Achieving Robust, Scalable Cluster I/O in Java

Matt Welsh and David Culler

University of California at Berkeley, Berkeley CA 94618, USA
{mdw,culler}@cs.berkeley.edu,
http://www.cs.berkeley.edu/~mdw

Abstract We present *Tigris*, a high-performance computation and I/O substrate for clusters of workstations that is implemented entirely in Java. Tigris automatically balances resource load across the cluster as a whole, shielding applications from asymmetries in CPU, I/O, and network performance. This is accomplished through the use of a *dataflow programming model* coupled with a work-balancing *distributed queue*. To meet the performance challenges of implementing such a system in Java, Tigris relies on *Jaguar*, a system that enables direct, protected access to hardware resources, including fast network interfaces and disk I/O. Jaguar yields an order-of-magnitude performance boost over the Java Native Interface for Java bindings to system resources. We demonstrate the applicability of Tigris through a one-pass, parallel, disk-to-disk sort exhibiting high performance.

1 Introduction

Realizing the performance potential of workstation clusters as a platform for incrementally scalable computing presents many challenges. While the performance aspects of communication [9,27], I/O [16], and process scheduling [4] have been addressed in specific settings, maintaining good performance across a range of different applications has proven difficult [3]. Applications tend to be fragile with respect to performance imbalance across the cluster; a single overloaded node can easily become a bottleneck for the entire application [2].

At the same time, Java has emerged as attractive platform allowing heterogeneous resources to be harnessed for large-scale computation. Java's object orientation, type and reference safety, exception handling model, code mobility, and distributed computing primitives all contribute to its popularity as a base upon which novel, component-based applications can be readily deployed. Increasingly, Java is becoming pervasive as a core technology supporting applications such as high-performance numerical computing [17], database connectivity [6], and scalable Internet services [14,19]. Unfortunately, most efforts in this direction focus on interfaces to legacy servers by encapsulating them in Java contexts, rather than the ability to construct large, I/O-intensive services directly in Java.

We present *Tigris*, a high-performance computation and I/O substrate for clusters of workstations, implemented entirely in Java. Tigris automatically balances resource load across the cluster as a whole, shielding applications from

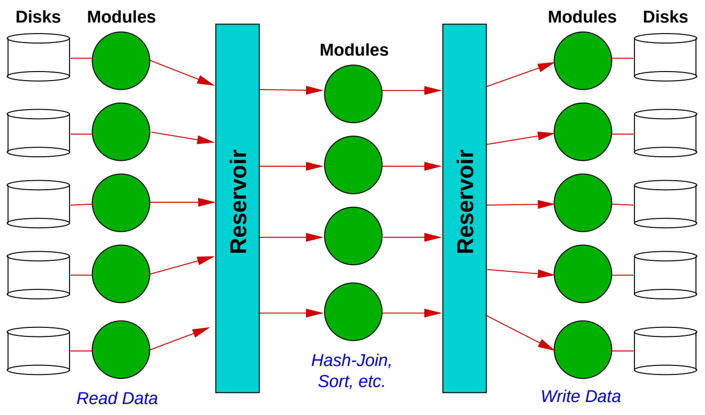

Figure 1. A sample Tigris application. *Tigris applications consist of a series of module stages connected by reservoirs. Reservoirs are realized as a distributed queue which allows data to flow from producers to consumers at autonomously adaptive rates.*

asymmetries in CPU, I/O, and network performance. This is accomplished through the use of a *dataflow programming model* coupled with a work-balancing *distributed queue*. By exploiting the use of Java as the native execution and control environment in Tigris, we believe that cluster application development is greatly simplified and that applications can take advantage of code mobility, strong typing, and other features provided by Java. The key ideas in Tigris build upon those River [5], a system which was implemented in C++ on the Berkeley Network of Workstations [25]. We describe the major differences between Tigris and River in Section 6.

Tigris achieves high-performance communication and disk I/O through the use of *Jaguar* [28], an extension of the Java programming environment which enables direct, protected access to hardware resources, such as fast network interfaces. Jaguar yields an order-of-magnitude performance boost over the Java Native Interface and eliminates the use of native methods, which raise protection concerns.

We evaluate the communication and load-balancing performance of Tigris on an 8-node workstation cluster and demonstrate its applicability through *Tigris-Sort*, a one-pass, parallel, disk-to-disk sort.

2 The Tigris System

The goal of Tigris is to automatically overcome cluster resource imbalance and mask this behavior from the application. This is motivated by the observation that cluster applications tend to be highly sensitive to performance heterogeneity; for example, if one node in the cluster is more heavily loaded than others, without some form of work redistribution the application may run at the rate of the slowest node. The larger and more heterogeneous a cluster is, the more evident this problem will be. Often, performance imbalance is difficult to prevent; for example, the location of bad blocks on a disk can seriously affect its bandwidth. This imbalance is especially serious for clusters which utilize nodes of varying CPU, network, and disk capabilities. Apart from hardware issues, software can cause performance asymmetry within a cluster as well; for example, "hot spots" may arise due to the distribution of data and computation in the application.

Tigris employs a dataflow programming model wherein applications are expressed as a series of *modules* each supporting a very simple input/output interface. Modules are organized into *stages*, each of which is a set of identical modules replicated across the nodes of a cluster. Increasing the number of modules in a stage can increase the effective bandwidth of that stage. Module stages communicate through the use of *reservoirs*, which are virtual communication channels through which data packets can be pushed or or pulled. A simple data-transformation application might consist of three distinct stages: one which reads data from disk and streams it out to a reservoir; one which reads packets from a reservoir and performs some transformation on that data; and one which writes data from a reservoir back onto disk. Figure 1 depicts this scenario.

Tigris addresses resource imbalance in a cluster by implementing reservoirs as a *distributed queue* (DQ), which balances work across the modules of a stage. The DQ allows data to flow at autonomously adaptive rates from producers to consumers, thereby causing data to "flow to where the resources are." In Tigris, the DQ is implemented by balancing outgoing data from a module across multiple communication channels.

2.1 Using Type-Safe Languages for Scalable Applications

Designers of robust, scalable systems can take advantage of the features provided by a type-safe language such as Java. Language protection and automatic memory management eliminate many common sources of bugs, and object orientation allows code to be built in a modular, reusable form. Within a cluster, Java's Remote Method Invocation (RMI) provides an elegant programming model for harnessing distributed resources. A JVM can export remote interfaces to local objects which are accessed through method invocation on a client-side stub. Java's portability simplifies the process of tying multiple heterogeneous architectures into a single application. The use of bytecode mobility allows an application to push new code modules into a JVM on demand. This is particularly valuable for building flexible cluster applications, as the binding between application modules and cluster nodes can be highly dynamic. For example, the number of nodes devoted to running a particular task can be resized based on application demand.

There are a number of challenges inherent in the use of Java for scalable server applications. A great deal of previous work has addressed problems with Java processor performance, including the efficiency of compiled code, thread

```
public class PrintModule implements ModuleIF {
  public void init(ModuleConfig cfg) { /* Empty */ }
  public void destroy() { /* Empty */ }
  public String getName() { return "PrintModule"; }

  public void doOperation(Water inWater, Stream outStream)
    throws ModuleException {
    for (int i = 0; i < inWater.getSize(); i++) {
      System.out.println("Read: " + inWater.readByte(i));
    }
    outStream.Put(inWater);
  }
}
```

Figure 2. An example of a Tigris module. *This module displays the contents of each packet it receives, and passes the packet along to the outgoing stream.*

synchronization, and garbage collection algorithms [20,22]. Java compilers, including both static and "just-in-time" (JIT) compilers, are now capable of generating code which rivals lower-level languages, such as C++, in performance [17]. However, obtaining high-performance I/O and efficient exploitation of low-level system resources remain as important performance problems. In this paper we describe *Jaguar*, our approach to obtaining high I/O performance in Java. Other issues include memory footprint, the binding between Java and operating system threads, and resource accounting. We believe that despite these issues, Java provides a compelling environment for the construction of cluster-based applications.

In Tigris, each cluster node runs a Java Virtual Machine which is bootstrapped with a receptive execution environment called the *MultiSpace* [14]. MultiSpace allows new Java classes to be "pushed into" the JVM remotely through Java Remote Method Invocation. A Security Manager is loaded into the MultiSpace to limit the behavior of untrusted Java classes uploaded into the JVM; for example, an untrusted component should not be allowed to access the filesystem directly. This allows a flexible security infrastructure to be constructed wherein Java classes running on cluster nodes can be given more or fewer capabilities to access system resources.

2.2 Design Overview

Tigris is implemented entirely in Java. Tigris modules are Java classes that implement the `ModuleIF` interface, which provides a small set of methods that each module must implement. The code for an example module is shown in Figure 2. `init` and `destroy` are used for module initialization and cleanup, and `getName` allows a module to provide a unique name for itself. The `doOperation` method is the core of the module's functionality: it is called whenever there is incoming data for the module to process, and is responsible for generating any outgoing data and pushing it down the dataflow path that the module is on.

Communication is managed by the `Stream` class, which provides two methods, `Get` and `Put`, allowing data items to be read from and written to a single, point-to-point communications channel. The `Water` class represents the unit of data which can be read from or written to a `Stream`; this is also the unit of work that is processed by the module `doOperation` method. A `Water` can be thought of as containing one or more data buffers that can be accessed directly (similarly to a Java array) or from which other Java objects can be allocated. This allows the contents of a `Water` to represent a structure with typed fields that have meaning to the Java application, rather than as an untyped collection of bytes or integers.

By subclassing `Stream` and `Water`, different communication mechanisms can be implemented in Tigris. Our prototype implementation includes three stream implementations:

– `ViaStream` provides reliable communications over Berkeley VIA [8], a fast communications layer implemented on the Myrinet system area network. This is accomplished through Jaguar (see below).
– `MemoryStream` implements communications between modules on the same JVM, passing the data through a FIFO queue in memory.
– `FileStream` associates the `Get` and `Put` stream operations with data read from and written to a file, respectively. This is a convenient way to abstract disk I/O. Each `Get` or `Put` operation accesses file data in FIFO order; random access is provided by a separate seek method.

`Water`s are initially created by a `Spring`, an interface which contains a single method: `createWater(int size)`. Every `Stream` has associated with it a `Spring` implementation that is capable of creating `Water`s which can be sent over that `Stream`. This allows a stream to manage allocation of `Water`s which will be eventually transmitted over it; for example, a stream may wish to initialize data fields in the `Water` to implement a particular communications protocol (e.g., sequence numbers). The implementation of `Water` can ensure that a module is unable to modify these "hidden" fields once the `Water` is created, by limiting the range of data items that can be accessed by the application.

2.3 Reservoir Implementation

Reservoirs are used as a virtual communication channel between stages of a Tigris application. Logically, modules pull data from their upstream reservoir, process it, and push data to their downstream reservoir. Reservoirs are implemented as a Distributed Queue (DQ), which dynamically balances communication between individual modules in a stage. Each Tigris module has multiple incoming and outgoing streams; the DQ is realized through the stream selection policy used by each module. Each module has an associated thread that is responsible for repeatedly issuing `Get` on one of the module's incoming streams, and invoking

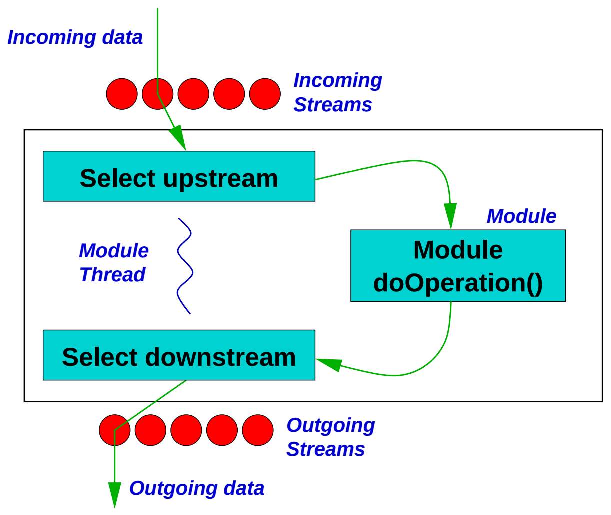

Figure 3. Tigris Reservoir implementation. *Tigris modules have an associated thread which repeatedly selects an incoming stream, reads data from it, invokes the module's* `doOperation` *method, and sends outgoing data to one of the outgoing streams. The reservoir is implemented through the stream selection policy used by this thread. Module authors are required only to implement the* `doOperation` *method.*

`doOperation` with two arguments: the input `Water`, and a handle to the outgoing stream to which any new data should be sent. This logic is invisible to the application, which is only responsible for implementing the `doOperation` method for each module.

By passing a handle to the current outgoing stream to `doOperation`, the module is capable of emitting zero or more `Water`s on each iteration. Also, this permits the module to obtain a handle to the stream's `Spring` to create new `Water`s to be transmitted. Note that the module may decide to re-transmit the same `Water` which it took as input; because a stream may not be capable of directly transmitting an arbitrary `Water` (for example, a `ViaStream` cannot transmit a `FileWater`), the stream is responsible for transforming the `Water` if necessary, e.g., by making a copy.

There are three implementations of the DQ scheduler in our prototype:

– *Round-Robin*: Selects the incoming and outgoing stream for each iteration in a round-robin fashion.
– *Random*: Selects the incoming stream for each iteration using round-robin, and the outgoing stream at random. The algorithm maintains a credit count for each outgoing stream. The credit count is decremented for each `Water` sent on a stream, and is incremented when the `Water` has been processed by the destination (e.g., through an acknowledgement). On each iteration,

a random stream S with a nonzero credit count is chosen from the set of downstream reservoirs.

– *Lottery*: Selects the incoming stream for each iteration using round-robin, and the outgoing stream using a lottery-based scheme. The algorithm maintains a credit count for each outgoing stream. On each iteration, a random stream S is chosen where the choice of S is weighted by the value $w = (c_S/C)$ where c_S is the number of credits belonging to stream S and $C = \sum c_S$. The intuition is that streams with more credits are more likely to be chosen, allowing bandwidth to be naturally balanced across multiple streams.

A reservoir may also be *Deterministic*, in which packets are sent between modules of a stage based on a deterministic routing. Deterministic reservoirs do not perform load balancing, but are useful for applications which require data to flow along a deterministic path between modules. The TigrisSort benchmark described later makes use of a deterministic reservoir.

2.4 Initialization and Control

A Tigris application is controlled by an external client which contacts the MultiSpace control interface of each cluster node through Java RMI, and communicates with the `TigrisMgr` service running on that node. `TigrisMgr` provides methods to create a module, to create a stream, to add a stream as an incoming or outgoing stream of a given module, and to start and stop a given module. In this way the Tigris application and module connectivity graph is "grown" at runtime on top of the receptive MultiSpace environment rather than hardcoded. Each cluster node need only be running the MultiSpace environment with the `TigrisMgr` service preloaded.

Execution begins when the control client issues the `moduleStart` command to each module, and ends when one of two conditions occur:

– The control client issues `moduleStop` to every module; or,
– Every module reaches the "End of Stream" condition.

"End of Stream" (EOS) is indicated by a module receiving a null `Water` as input. This can be triggered by a producer pushing a null `Water` down a stream towards a consumer, or by some other event (such as the DQ implementation itself declaring an EOS condition). A module may indicate to its control thread that EOS has been reached by throwing an `EndOfStreamException` from its `doOperation` method; this obviates the need for an additional status value to be passed between a module and its controlling thread.

3 Jaguar: High-Performance Communication and I/O in Java

Efficient communication and I/O in Tigris is provided by *Jaguar* [28], an extension of the Java environment that enables direct access to system resources such as fast network interfaces and disk I/O. Traditionally, Java applications make use of low-level system functionality through the use of *native methods*, which are written in a language such as C. To bind native method code to the Java application, a *native method interface* is used, which has been standardized across most JVMs as Sun Microsystems' Java Native Interface [21]. However, the use of native methods raises two important concerns. The first is performance: the cost of traversing the native method interface can be quite high, especially when a large amount of data must be copied across the Java-native code boundary. The second is safety: invoking arbitrary native code from a Java application effectively negates the protection guarantees of the Java Virtual Machine. These two problems conflate, as programmers tend to write more application code in the native language to amortize the cost of crossing the native interface.

Jaguar overcomes these problems by providing applications with efficient and safe access to low-level system resources. This is accomplished through a bytecode specialization technique in which certain Java bytecode sequences are translated to low-level code which is capable of performing functions not allowed by the JVM, such as direct memory access. Because this low-level code is inlined into the Java application at compile time, the overhead of the native interface is avoided. Also, the compiler can perform aggressive optimizations on the combined application and inlined Jaguar code. Low-level code is expressed as a type-exact, portable superset of Java bytecode, *Jaguar bytecode*, which includes additional instructions required for direct memory access. Application programmers are not permitted to make use of these instructions; they are only used within Java-to-Jaguar bytecode translation rules.

Tigris makes use of Jaguar in two ways: to implement efficient network and disk I/O, and to avoid Java object serialization. Network I/O is provided by the `ViaStream` class, which is implemented using the Jaguar interface to the Berkeley VIA communications architecture. Jaguar's VIA interface provides direct, zero-copy access to the Myrinet system area network, and obtains a round-trip time of 73 microseconds for small messages, and a peak bandwidth of over 488 mbits/second. This is identical to the performance of Berkeley VIA as accessed from C code. Jaguar translates access to certain Java classes (representing VIA registers, packet descriptors, and network buffers) into low-level code which directly manipulates these system resources.

Likewise, disk I/O in Tigris is provided by the `FileStream` class; this is implemented by memory-mapping the contents of a disk file (using the *mmap* system call) into the application address space. Jaguar specializes accesses to the `Water` objects representing file data to direct access to this memory-mapped region.

Jaguar is also used to map Java objects onto the raw byte streams transferred over network and disk streams. Rather than employ Java object serialization, which involves data copying and is very costly, Tigris uses Jaguar's *Pre-Serialized Objects* feature, which translates Java object field access into direct access to a low-level network, disk, or memory buffer. No copies are required to map a Pre-

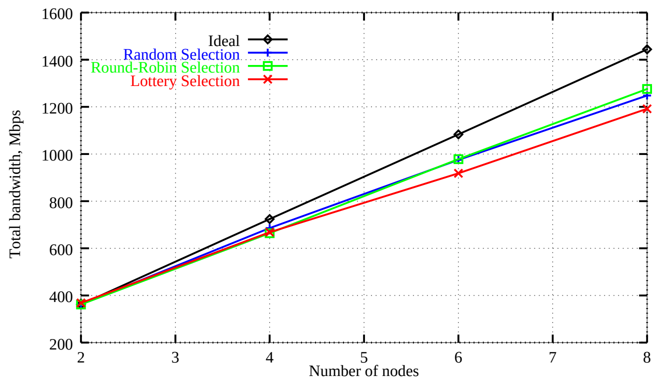

Figure 4. Reservoir performance under scaling. *This figure shows the aggregate bandwidth through the reservoir as the number of nodes reading and writing data through the reservoir is scaled. From 1 to 4 nodes write data into a reservoir implemented on top of the VIA network interface, and 1 to 4 nodes read data from the reservoir. The three DQ implementations (round-robin, randomized, and lottery) are shown, along with the ideal bandwidth under perfect scaling.*

Serialized Object onto an I/O buffer. In this way, Tigris streams are used to efficiently transfer Java objects rather than raw bytes.

4 Reservoir Performance

Figure 4 shows the performance of the Tigris reservoir implementations (round-robin, randomized, and lottery) as the number of nodes passing data through the reservoir is scaled up. All experiments were performed on a cluster of 500 MHz Pentium III systems with 512 MB of memory running Linux 2.2.13, connected using Myrinet with the Berkeley VIA communications layer. The Blackdown port of Sun JDK 1.1.7v3 is used along with a Jaguar-enabled JIT compiler. The `ViaStream` stream type is used, which implements a simple credit-based flow-control scheme over VIA. End-to-end peak bandwidth through a `ViaStream` is 368 Mbits/sec, or 75% of the peak bandwidth of raw VIA, which implements no flow-control or reliability.

In each case an equal number of nodes are sending and receiving data through the reservoir. The results show only a slight bandwidth loss (12% less than ideal) in the 8-node case, demonstrating that the presence of a reservoir does not seriously affect performance when the system is perfectly balanced. The bandwidth loss is partially due to the DQ implementation itself; in each case, the receiving node selects the incoming stream from which to receive data in a round-robin manner. Although the receive operation is non-blocking it does require the receiver to test for incoming data on each incoming stream until a packet arrives. We also believe that a portion of this bandwidth loss is due

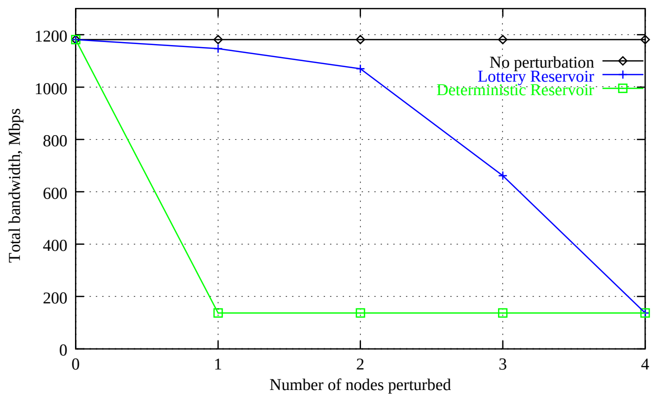

Figure 5. Reservoir performance under perturbation. *This figure shows the performance of the lottery reservoir as consumer nodes are artificially perturbed. 4 nodes are pushing data into a reservoir and 4 nodes are pulling data from the reservoir. When 3 out of 4 consumers are perturbed, 56% of the total bandwidth can be achieved. With the deterministic reservoir, performance drops as soon as a single receiver is perturbed.*

the VIA implementation being used; as the number of active communication channels increases, the network interface must poll additional queues to test for incoming or outgoing packets.

The second benchmark demonstrates the performance of the lottery reservoir in a scenario that models a performance imbalance in the cluster. This is accomplished by artificially loading nodes by adding a fixed delay to each iteration of the module's `doOperation` method. In this benchmark, 4 nodes are pushing data into the reservoir, and 4 nodes are pulling data from the reservoir; the receiving nodes are perturbed as described above. Figure 5 shows the aggregate bandwidth through the reservoir as the number of perturbed nodes is increased.

The total bandwidth in the unperturbed case is 1181.58 Mbits/second (4 nodes sending 8Kb packets at the maximum rate to 4 receivers through the reservoir), or 295.39 Mbits/sec per node. Perturbation of a node limits its receive bandwidth to 34.27 Mbits/sec. The lottery reservoir balances bandwidth automatically to nodes which are receiving at a higher rate, so that when 3 out of 4 nodes are perturbed, 56% of the total bandwidth can be achieved. Over 90% of the total bandwidth is obtained with half of the nodes perturbed. With the use of a non-load-balancing deterministic reservoir, the total bandwidth is limited by the slowest node, as shown in the figure.

5 TigrisSort: A Sample Application

In order to evaluate Tigris more generally, we have implemented *TigrisSort*, a parallel, disk-to-disk sorting benchmark. As with Datamation [12] and NOW-Sort [3], sorting is a good way to measure the memory, I/O, and communication

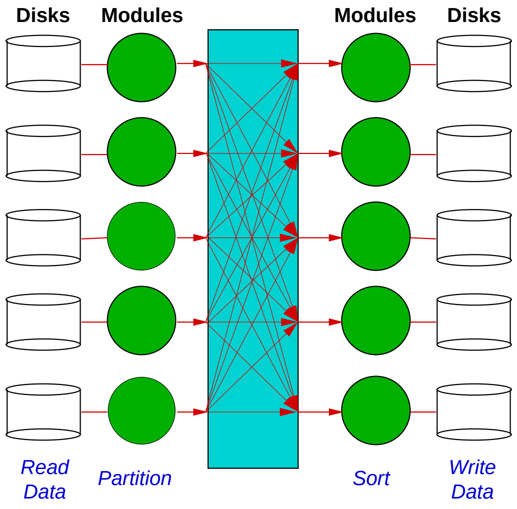

Figure 6. TigrisSort structure. *The application consists of two types of modules: partitioners and sorters. Partitioning nodes read data from their local disk and partition it into buckets based on the record key. Full buckets are transmitted to the appropriate sorting node, which sort the local data set and write it to the local disk. Communication is accomplished using a deterministic reservoir which routes packets to the appropriate sorting node based on the bucket's key value.*

performance of the complete system. While the existence of previous sorting results on other systems yields a yardstick by which the Tigris system can be compared, we were also interested in understanding the functional properties of the Tigris and Jaguar mechanisms in the context of a "real application."

5.1 TigrisSort Structure

The structure of TigrisSort is shown in Figure 6. TigrisSort implements a one-pass, disk-to-disk parallel sort of 100-byte records, each of which contains a 10-byte key. Data is initially striped across the input disks with 5 megabytes of random data per node.[1] The application consists of two sets of nodes: *partitioners* and *sorters*. Partitioning nodes are responsible for reading data from their local disk and partitioning it based on the record key; the partition "buckets" from each node are transmitted to the sorting nodes which sort the entire data set and write it to the local disk. This results in the sorted dataset being range-partitioned across the nodes, with each partition in sorted order. Communication is accomplished using a deterministic reservoir between the partitioning

[1] This amount is arbitrary; the software has no limitation in the amount of data that can be partitioned or sorted per node. 5 megabytes is a convenient value that limits the amount of memory that must be devoted to pinned VIA network buffers, which simplifies the benchmark structure somewhat.

# nodes	Amount sorted	Avg time/node	Total sort bw
2	5 MBytes	762 msec	52.49 Mbps
4	10 MBytes	734.5 msec	108.91 Mbps
6	15 MBytes	733 msec	163.71 Mbps
8	20 MBytes	725 msec	220.68 Mbps

Figure 7. TigrisSort performance results.

and sorting stages; this reservoir routes `Water`s representing full buckets to the appropriate sorting node based on the bucket's key value. This application cannot make use of a load-balancing reservoir, as data must be deterministically partitioned across the sorting modules.

File I/O is implemented by a class which maps a file into the address space of the JVM and exposes it directly to the Java application, through Jaguar, using methods such as `readByte` and `writeByte`. Operations on this class correspond to disk reads and writes through the memory-mapped file. This is the same mechanism used by the `FileStream` class. A special method is provided, `flush()`, which causes the contents of the memory-mapped file to be flushed to disk.

This approach has several limitations. One is that the operating system being used (Linux 2.2.13) does not allow the buffer cache to be circumvented using memory-mapped files, meaning that file data may be double-buffered. Another is that a particular write ordering cannot be enforced. Currently, Linux does not provide a "raw disk" mechanism which provides these features. Rather than concerning ourselves with these details, we assume that performance differences arising because of them will be negligible. This seems to be reasonable: first, disk I/O is just one component of the TigrisSort application, which does not appear to be disk-bandwidth limited. Secondly, double-buffering of sort data in memory is not problematic with the small (5 megabyte) per-node data partitions being dealt with. Third, write ordering is not important for implementing parallel sort; it is sufficient to ensure that all disk writes have completed.

The actual partitioning and sorting of records are implemented using native methods which manipulate I/O buffers directly, using pointers. This was necessary to avoid performance limitations of our Java compiler, which does not perform aggressive optimizations such as array bounds-check elimination. No data copying between Java and C is necessary as both Java (through Jaguar) and native code can directly manipulate the contents of the I/O buffers.

5.2 TigrisSort Performance

Figure 7 shows the performance of TigrisSort as the benchmark is scaled up from 2 to 8 nodes. In each case, half of the nodes are configured as partitioners, and half as sorters; 5 megabytes of data are partitioned or sorted per node. The total time to complete the sort averaged 738 milliseconds; as more nodes are added to the application, more data can be sorted in constant time. Given this result we feel that with careful tuning, TigrisSort can compete with the current world-record holder of the Datamation sort record, Millennium Sort [7], which was implemented on the same hardware platform using Windows NT and C++. However, the dominant cost of Datamation sort on modern systems is application startup; the results above do not include the cost of starting the Tigris system itself. Because Tigris is implemented in Java, which involves higher application startup cost (for class loading and JIT compilation), there is some question as to what should be included in the startup measurements. For instance, for traditional Datamation sort implementations, the cost of compiling the application and cold-booting the operating system are not measured.

6 Related Work

Tigris relates most closely to work in the areas of cluster programming frameworks and parallel databases.

Tigris' design is based on River [5], a robust cluster-based I/O system implemented in C++ on the Berkeley Network of Workstations [25]. While facially quite similar, Tigris differs from the original River system in a number of important respects, mainly stemming from the use of Java and Jaguar. In River, communication channels transmit raw byte streams (using Active Messages [9]), and the application must extract typed values from them through the use of a "data dictionary" class which maps field names onto byte offsets in the stream. In Tigris, modules map Java objects directly onto I/O buffers represented by the `Water` class through the use of Jaguar's *Pre-Serialized Object* mechanism. Type checking is therefore enforced by the compiler and modules operate directly on "true" Java objects, rather than using a library for indirection.

River made use of a single DQ algorithm (randomized), while Tigris introduces the round-robin and lottery DQ variants. Tigris additionally provides stream interfaces to shared memory segments and memory-mapped disk files, both enabled using Jaguar.

Other projects have considered support for parallel and distributed computing in a type-safe language. JavaParty [18] is an extension to Java providing transparent remote object access and object mobility. cJVM [1] is a cluster-based JVM which implements DSM-like remote object access and thread migration. DOSA [15] is a DSM system which provides fine-grain sharing at the granularity of objects which could be applied to languages such as Java. Titanium [30] is a dialect of Java for large-scale scientific computing; it is focused on static compilation and automated techniques for optimization of parallel programs expressed in the Titanium language. Other models for Java-based parallel computing, such as work stealing in JAWS [29] and agents in Ninflet [23], have also been considered. Tigris is the first dataflow and I/O-centric programming model to our knowledge to have been explored in the Java environment.

Parallel databases have made use of some of the dataflow concepts found in Tigris. Systems such as Bubba [10], Gamma [11], and Volcano [13] made use of static data partitioning techniques, rather than runtime adaption, to balance load across multiple physical resources. Static partitioning techniques form the foundation for all commercial shared-nothing parallel RDBMS products to our knowledge. Relational database queries have long been viewed as a dataflow graph; Eddies [6] build on the concepts in Tigris and River by dynamically reordering operators in a parallel database query plan to adapt to runtime performance fluctuations.

7 Conclusion

Cluster programming is by its nature a difficult task; obtaining good performance in the face of performance perturbations is even harder. Compounding this problem is the increasing use of clusters for irregular applications such as hosting Internet services and databases, which must tolerate fluctuations in resource availability and performance. Java has proven to be a viable platform for constructing such applications; what remains now is to bridge the gap between application demands and the mechanisms provided the underlying platform.

Tigris takes the idea of cluster programming in Java a step further by introducing dynamic resource adaptation to the programming model, as well as the use of high-performance networking and disk I/O through Jaguar. Several novel applications are being developed using Tigris as a base; the Telegraph [26] and Ninja [24] projects at UC Berkeley are both incorporating Tigris into their design for cluster-based scalable services.

We believe that through our experience with TigrisSort, as well as the low-level benchmarks of the Tigris DQ performance, that Tigris is an effective platform upon which to construct adaptive cluster applications. Moreover, the use of Jaguar allows us to build interesting data-intensive applications entirely in Java, which opens up new possibilities for developing flexible cluster programming environments.

References

1. Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A high performance cluster jvm presenting a pure single system image. In *Proceedings of the ACM 2000 JavaGrande Conference*, San Francisco, CA, June 2000.
2. R. H. Arpaci, A. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of SIGMETRICS/PERFORMANCE*, May 1995.
3. A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. Searching for the sorting record: Experiences in tuning NOW-Sort. In *Proceedings of the 1998 Symposium on Parallel and Distributed Tools (SPDT '98)*, 1998.
4. A. C. Arpaci-Dusseau, D. E. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *1998 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 233–243, June 1998.

5. R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *IOPADS '99*, 1999. `http://www.cs.berkeley.edu/~remzi/Postscript/river.ps`.
6. R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.
7. P. Buonadonna, J. Coates, S. Low, and D. E. Culler. Millennium Sort: A Cluster-Based Application for Windows NT Using DCOM, River Primitives and the Virtual Interface Architecture. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.
8. P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, November 1998.
9. B. Chun, A. Mainwaring, and D. Culler. Virtual network transport protocols for Myrinet. *IEEE Micro*, 18(1), January/February 1998.
10. G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. *SIGMOD Record*, 17(3):99–108, September 1988.
11. D. J. DeWitt, S. Ghanderaizadeh, and D. Schneider. A Performance Analysis of the Gamma Database Machine. *SIGMOD Record*, 17(3):350–360, September 1988.
12. Anon *et. al.* A measure of transaction processing power. In *Datamation, 31(7): 112-118*, February 1985.
13. G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record*, 19(2):102–111, June 1990.
14. S. Gribble, M. Welsh, D. Culler, and E. Brewer. Multispace: An evolutionary platform for infrastructural services. In *Proceedings of the 16th USENIX Annual Technical Conference*, Monterey, California, 1999.
15. Y. Charlie Hu, Weimin Yu, Dan Wallach, Alan Cox, and Willy Zwaenepoel. Run-time support for distributed sharing in typed languages. In *Proceedings of the Fifth ACM Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.
16. J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, July 1995.
17. J. Moreira, S. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, 1998. `http://www.research.ibm.com/ninja/`.
18. M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. In *Concurrency: Practice and Experience, 9(11):1225-1242*, November 1997.
19. Sun Microsystems Inc. Enterprise Java Beans Technology. `http://java.sun.com/products/ejb/`.
20. Sun Microsystems Inc. Java HotSpot Performance Engine. `http://java.sun.com/products/hotspot/index.html`.
21. Sun Microsystems Inc. Java Native Interface Specification. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html`.
22. Sun Microsystems Labs. The Exact Virtual Machine (EVM). `http://www.sunlabs.com/research/java-topics/`.
23. H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: A migratable parallel objects framework using Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998. `http://www.cs.ucsb.edu/conferences/java98/papers/ninflet.pdf`.

24. UC Berkeley Ninja Project. `http://ninja.cs.berkeley.edu`.
25. UC Berkeley NOW Project. The UC Berkeley Network of Workstations Project. `http://now.cs.berkeley.edu`.
26. UC Berkeley Telegraph Project. `http://db.cs.berkeley.edu/telegraph/`.
27. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, December 1995.
28. M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O from Java. *Concurrency: Practice and Experience*, 2000. Special Issue on Java for High-Performance Network Computing, To appear, `http://www.cs.berkeley.edu/~mdw/proj/jaguar`.
29. A. Woo, Z. Mao, and H. So. The Berkeley JAWS Project. `http://www.cs.berkeley.edu/~awoo/cs262/jaws.html`.
30. Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.