CCJ: Object-based Message Passing and Collective Communication in Java

Arnold Nelisse, Jason Maassen, Thilo Kielmann, Henri E. Bal
Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
arnold@cs.vu.nl jason@cs.vu.nl kielmann@cs.vu.nl bal@cs.vu.nl
http://www.cs.vu.nl/manta

Abstract

CCJ is a communication library that adds MPI-like message passing and collective operations to Java. Rather than trying to adhere to the precise MPI syntax, CCJ aims at a clean integration of communication into Java's object-oriented framework. For example, CCJ uses thread groups to support Java's multithreading model and it allows any data structure (not just arrays) to be communicated. CCJ is implemented entirely in Java, on top of RMI, so it can be used with any Java virtual machine. The paper discusses three parallel Java applications that use collective communication. It compares the performance (on top of a Myrinet cluster) of CCJ, RMI and mpiJava versions of these applications, and also compares the code complexity of the CCJ and RMI versions. The results show that the CCJ versions are significantly simpler than the RMI versions and obtain a good performance. A detailed performance comparison between CCJ and mpiJava is given using the Java Grande Forum MPJ benchmark suite.

1 Introduction

Recent improvements in compilers and communication mechanisms make Java a viable platform for high-performance computing [8]. Java's support for multithreading and Remote Method Invocation (RMI) is a suitable basis for writing parallel programs. RMI uses a familiar abstraction (object invocation), integrated in a clean way in Java's object-oriented programming model. For example, almost any data structure can be passed as argument or return value in an RMI. Also, RMI can be implemented efficiently [21, 25] and it can be extended seamlessly with support for object replication [20].

A disadvantage of RMI, however, is that it only supports communication between two parties, a client and a server. Experience with other parallel languages has shown that many applications also require communication between multiple processes. The MPI message passing standard defines collective communication operations for this purpose [22]. Several projects have proposed to extend Java with MPI-like collective operations [9, 13]. For example, MPJ [9] proposes MPI language bindings to Java, but it does not integrate MPI's notions of processes and messages into Java's object-oriented framework. Unlike RMI, the MPI primitives are biased towards array-based data structures, so collective operations that exchange other data structures are often awkward to implement. Some existing Java systems already support MPI's collective operations, but often they invoke a C-library from Java using the Java Native Interface, which has a large runtime overhead [13].

In this paper we present the CCJ library (Collective Communication in Java) which adds the core of MPI's message passing and collective communication operations to Java's object model. CCJ maintains thread groups the members of which can communicate by exchanging arbitrary object data structures. For example, if one thread needs to distribute a list data structure among other threads, it can invoke an MPI-like scatter primitive to do so. CCJ is implemented entirely in Java, on top of RMI. It therefore does not suffer from JNI overhead and it can be used with any Java virtual machine. We study CCJ's performance on top of a fast RMI system (Manta [21]) that runs over a Myrinet network. Performance measurements for CCJ's collective operations show that its runtime overhead is almost negligible compared to the time spent in the underlying (efficient) RMI mechanism. We also discuss CCJ applications and their performance. CCJ's support for arbitrary data structures is useful for example in implementing sparse matrices. We also compare CCJ's performance to mpiJava in detail using the Java Grande Forum MPJ benchmark suite.

The rest of the paper is structured as follows. In Sections 2 and 3, we present CCJ's design and implementation, respectively. In Section 4, we discuss code complexity and performance of three application programs using CCJ, mpiJava, and plain RMI. In Section 5, we present the result from the Java Grande Forum benchmarks. Section 6 presents related work, Section 7 concludes.

2 Object-based message passing and collective communication

With Java's multithreading support, individual threads can be coordinated to operate under mutual exclusion. However, with collective communication, *groups* of threads cooperate to perform a given operation collectively. This form of cooperation, instead of mere concurrency, is used frequently in parallel applications and also enables efficient implementation of the collective operations.

In this section, we present and discuss the approach taken in our CCJ library to integrate message passing and collective communication, as inspired by the MPI standard, into Java's object-based model. CCJ integrates MPI-like operations in a clean way in Java, but without trying to be compatible with the precise MPI syntax. CCJ translates MPI processes into active objects (threads) and thus preserves MPI's implicit group synchronization properties. In previous work, we discussed the alternative approach of using groups of passive objects [20].

2.1 Thread groups

With the MPI standard, *processes* perform point-to-point and collective communication within the context of a *communicator* object. The communicator defines the group of participating processes which are ordered by their *rank*. Each process can retrieve its rank and the size of the process group from the communicator object. MPI communicators can not be changed at runtime, but new communicators can be derived from existing ones.

In MPI, immutable process groups (enforced via immutable communicator objects) are vital for defining sound semantics of collective operations. For example, a barrier operation performed on an immutable group clearly defines which processes are synchronized; for a broadcast operation, the set of receivers can be clearly identified. The ranking of processes is also necessary to define operations like scatter/gather data re-distributions, where the data sent or received by each individual process is determined by its rank. Unlike MPI, the PVM message passing system [12] allows mutable process groups, trading clear semantics for flexibility.

The MPI process group model, however, does not easily map onto Java's multithreading model. The units of execution in Java are dynamically created threads rather than heavy-weight processes. Also, the RMI mechanism blurs the boundaries between individual Java Virtual Machines (JVMs). Having more than one thread per JVM participating in collective communication can be useful, for example for application structuring or for exploiting multiple CPUs of a shared-memory machine. Although the MPI standard requires implementations to be thread-safe, dynamically created threads can not be addressed by MPI messages, excluding their proper use in collective communication.

CCJ maps MPI's immutable process groups onto Java's multithreading model by defining a model of thread groups that constructs immutable groups from dynamically created threads. CCJ uses a two-phase creation mechanism. In the first phase, a group is *inactive* and can be constructed by threads willing to join. After construction is completed, the group becomes immutable (called *active*) and can be used for collective communication. For convenience, inactive copies of active groups can be created and subsequently modified. Group management in CCJ uses the following three classes.

ColGroup Objects of this class define the thread groups to be used for collective operations. ColGroup provides methods for retrieving the rank of a given ColMember object and the size of the group.

ColMember Objects of this class can become members of a given group. Applications implement subclasses of ColMember, the instances of which will be associated with their own thread of control.

ColGroupMaster Each participating JVM has to initialize one object of this class acting as a central group manager. The group master also encapsulates the communication establishment like the interaction with the RMI registry.

For implementing the two-phase group creation, ColGroupMaster provides the following interface. Groups are identified by String objects with symbolic identifications.

void addMember(String groupName, ColMember member)

Adds a member to a group. If the group does not yet exist, the group will be created. Otherwise, the group must still be inactive; the getGroup operation for this group must not have completed so far.

ColGroup getGroup(String groupName, int numberOfMembers)

Activates a group. The operation waits until the specified number of members have been added to the group. Finally, the activated group is returned. All members of a group have to call this operation prior to any collective communication.

2.2 Message Passing

For some applications, simple message exchange between two group members can be beneficial. Inspired by the MPI standard, we added the following operations for synchronous and asynchronous message sending, for receiving, and for a combined send-receive. We also added a rendezvous message exchange, which is equivalent to two nodes performing send-receive operations with each other. This rendezvous can be implemented very efficiently by a single RMI request/reply pair of messages.

void send_sync(ColGroup group, Serializable object, int destination)
: Sends object to the member destination of the group. Waits until the object has been received using the receive operation.

void send_async(ColGroup group, Serializable object, int destination)
: Same as send_sync, but only delivers the object at the receiving member's node, without waiting for the receiver to call the receive operation.

Serializable receive(ColGroup group, int source)
: Receives and returns an object from the group's member source. Waits until the object is available.

Serializable send_receive(ColGroup send_group, Serializable send_object, ColGroup receive_group, Serializable receive_object)
: Simultaneously performs a send_async and an unrelated receive operation.

Serializable rendezvous(ColGroup group, Serializable object, int peer)
: Sends object to the group's member peer and returns an object sent by that member.

2.3 Collective communication

As described above, CCJ's group management alleviates the restrictions of MPI's static, communicator based group model. For defining an object-based framework, also the collective communication operations themselves have to be adapted. MPI defines a large set of collective operations, inspired by parallel application codes written in more traditional languages such as Fortran or C. Basically, MPI messages consist of arrays of data items of given data types. Although important for many scientific codes, arrays can not serve as general-purpose data structure in Java's object model. Instead, collective operations should deal with serializable objects in the most general case.

The implementation of the collective operations could either be part of the group or of the members. For CCJ, we decided for the latter option as this is closer to the original MPI specification and more intuitive with the communication context (the group) becoming a parameter of the operation.

From MPI's original set of collective operations, CCJ currently implements the most important ones, leaving out those operations that are either rarely used or strongly biased by having arrays as general parameter data structure. CCJ currently implements Barrier, Broadcast, Scatter, Gather, Allgather, Reduce, and Allreduce. We now present the interface of these operations in detail. For the reduce operations, we also present the use of function objects implementing the reduction operators themselves. For scatter and gather, we present the DividableDataObjectInterface imposing a notion of indexing for the elements of general (non-array) objects. CCJ uses Java's exception handling mechanism for catching error conditions returned by the various primitives. For brevity, however, we do not show the exceptions in the primitives discussed below. Like MPI, CCJ requires all members of a group to call collective operations in the same order and with mutually consistent parameter objects.

void barrier(ColGroup group)

Waits until all members of the specified group have called the method.

Object broadcast(ColGroup group, Serializable obj, int root)

One member of the group, the one whose rank equals root, provides an object obj to be broadcast to the group. All members (except the root) return a copy of the object; to the root member, a reference to obj is returned.

MPI defines a group of operations that perform global reductions such as summation or maximum on data items distributed across a communicator's process group. MPI identifies the reduction operators either via predefined constants like "MPI_MAX," or by user-implemented functions. However, object-oriented reduction operations have to process objects of application-specific classes; implementations of reduction operators have to handle the correct object classes.

One implementation would be to let application classes implement a reduce method that can be called from within the collective reduction operations. However, this approach restricts a class to exactly one reduction operation and excludes the basic (numeric) data types from being used in reduction operations.

As a consequence, the reduction operators have to be implemented outside the objects to be reduced. Unfortunately, unlike in C, functions (or methods) can not be used as first-class entities in Java. Alternatively, Java's reflection mechanism could be used to identify methods by their names and defining class (specified by String objects). Unfortunately, this approach is unsuitable, because reflection is done at runtime, causing prohibitive costs for use in parallel applications. Removing reflection from object serialization is one of the essential optimizations of our fast RMI implementation in the Manta system [21].

CCJ thus uses a different approach for implementing reduction operators: *function objects* [19]. CCJ's function objects implement the specific ReductionObjectInterface containing a single method Serializable reduce(Serializable o1, Serializable o2). With this approach, all application specific classes and the standard data types can be used for data reduction. The reduction operator itself can be flexibly chosen on a per-operation basis. Operations implementing this interface are supposed to be associative and commutative. CCJ provides a set of function objects for the most important reduction operators on numerical data. This leads to the following interface for CCJ's reduction operations in the ColMember class.

Serializable reduce(ColGroup group, Serializable dataObject,
ReductionObjectInterface reductionObject, int root)

Performs a reduction operation on the dataObjects provided by the members of the group. The operation itself is determined by the reductionObject; each member has to provide a reductionObject of the same class. reduce returns an object with the reduction result to the member identified as root. All other members get a null reference.

Serializable allReduce(ColGroup group, Serializable dataObject,
ReductionObjectInterface reductionObject)
Like reduce but returns the resulting object to all members.

The final group of collective operations that have been translated from MPI to CCJ is the one of scatter/gather data re-distributions: MPI's scatter operation takes an array provided by a root process and distributes ("scatters") it across all processes in a communicator's group. MPI's gather operation collects an array from items distributed across a communicator's group and returns it to a root process. MPI's allgather is similar, however returning the gathered array to all participating processes.

Although defined via arrays, these operations are important for many parallel applications. The problem to solve for CCJ thus is to find a similar notion of indexing for general (non-array) objects. Similar problems occur for implementing so-called iterators for container objects [11]. Here, traversing (iterating) an object's data structure has to be independent of the object's implementation in order to keep client classes immune to changes of the container object's implementation. Iterators request the individual items of a complex object sequentially, one after the other. Object serialization, as used by Java RMI, is one example of iterating a complex object structure. Unlike iterators, however, CCJ needs random access to the individual parts of a dividable object based on an index mechanism.

For this purpose, objects to be used in scatter/gather operations have to implement the DividableDataObjectInterface with the following two methods:

Serializable elementAt(int index, int groupSize)
Returns the object with the given index in the range from 0 to groupSize -1

void setElementAt(int index, int groupSize, Serializable object)
Conversely, sets the object at the given index.

Based on this interface, the class ColMember implements the following three collective operations.

Serializable scatter(ColGroup group,
DividableDataObjectInterface rootObject, int root)
The root member provides a dividable object which will be scattered among the members of the given group. Each member returns the (sub-)object determined by the elementAt method for its own rank. The parameter rootObject is ignored for all other members.

DividableDataObjectInterface gather(ColGroup group,
DividableDataObjectInterface rootObject,
Serializable dataObject, int root)
The root member provides a dividable object which will be gathered from the dataObjects provided by the members of the group. The actual order of the gathering is determined by the rootObject's setElementAt method, according to the rank of the members. The method returns the gathered object to the root member and a null reference to all other members.

DividableDataObjectInterface allGather(ColGroup group,
DividableDataObjectInterface resultObject,
Serializable dataObject)

Like gather, however the result is returned to all members and all members have to provide a resultObject.

2.4 Example application code

We will now illustrate how CCJ can be used for application programming. As our example, we show the code for the All-Pairs Shortest Path application (ASP), the performance of which will be discussed in Section 4. Figure 1 shows the code of the Asp class that inherits from ColMember. Asp thus constitutes the application-specific member class for the ASP application. Its method do_asp performs the computation itself and uses CCJ's collective broadcast operation. Before doing so, Asp's run method first retrieves rank and size from the group object. Finally, do_asp calls the done method from the ColMember class in order to de-register the member object. The necessity of the done method is an artifact of Java's thread model in combination with RMI; without any assumptions about the underlying JVMs, there is no fully transparent way of terminating an RMI-based, distributed application run. Thus, CCJ's members have to de-register themselves prior to termination to allow the application to terminate gracefully.

Figure 2 shows the MainAsp class, implementing the method main. This method runs on all JVMs participating in the parallel computation. This class establishes the communication context before starting the computation itself. Therefore, a ColGroupMaster object is created (on all JVMs). Then, MainAsp creates an Asp member object, adds it to a group, and finally starts the computation. Our implementation of the ColGroupMaster also provides the number of available nodes, which is useful for initializing the application. On other platforms, however, this information could also be retrieved from different sources.

For comparison, Figure 3 shows some of the code of the mpiJava version of ASP. We will use this mpiJava program in Section 4 for a performance comparison with CCJ. A clear difference between the mpiJava and CCJ versions is that the initialization code of CCJ is more complicated. The reason is that mpiJava offers a simple model with one group member per processor, using the MPI.COMM_WORLD communicator. CCJ on the other hand is more flexible and allows multiple active objects per machine to join a group, which requires more initialization code. Also, the syntax of mpiJava is more MPI-like than that of CCJ, which tries to stay closer to the Java syntax.

3 The CCJ library

The CCJ library has been implemented as a Java package, containing the necessary classes, interfaces, and exceptions. CCJ is implemented on top of RMI in order to run with any given JVM. We use RMI to build a basic message passing layer between the members of a given group. On top of this messaging layer, the collective operations are implemented using algorithms like the ones described in [15, 18]. This section describes both the messaging layer and the collective algorithms of CCJ.

CCJ has been implemented using the Manta high performance Java system [21]. Our ex-

```
class Asp extends ColMember {
ColGroup group;
int n, rank, nodes;
int[][] tab; // the distance table.
Asp (int n) throws Exception {
   super();
   this.n = n;
}
void setGroup(ColGroup group) {
   this.group = group;
}
void do_asp() throws Exception {
   int k;
   for (k = 0; k < n; k++) {
      // send the row to all members:
      tab[k] = (int[])
              broadcast(group, tab[k], owner(k));
      // do ASP computation...
   }
}
public void run() {
   try {
      rank = group.getRank(this);
      nodes = group.size();
      // Initialize local data
      do_asp();
      done();
   } catch (Exception e) {
     // handle exception... Quit.
   }
}
}
```

Figure 1: Java class Asp

perimentation platform, called the *Distributed ASCI Supercomputer* (DAS), consists of 200 MHz Pentium Pro nodes each with 128 MB memory, running Linux 2.2.16. The nodes are connected via Myrinet [5]. Manta's runtime system has access to the network in user space via the Panda communication substrate [3] which uses the LFC [4] Myrinet control program. The system is more fully described in *http://www.cs.vu.nl/das/*. All performance numbers reported in this work have been achieved on the DAS platform.

For comparison, we also provide completion times using the RMI implementation from Sun's JDK 1.1.4. We have ported this to Manta by replacing all JNI calls with direct C function calls. By

```
class MainAsp {
int N;
void start(String args[]) {
   ColGroup group = null;
   int numberOfCpus;
   Asp myMember;
   try {
      ColGroupMaster
         groupMaster = new ColGroupMaster(args);
      numberOfCpus = groupMaster.getNumberOfCpus();
      // get number of rows N from command line
      myMember = new Asp(N);
      groupMaster.addMember("myGroup", myMember);
      group = groupMaster.getGroup("myGroup",
                                    numberOfCpus);
      myMember.setGroup(group);
      (new Thread(myMember)).start();
   } catch (Exception e) {
      // Handle exception... Quit.
   }
}
public static void main (String args[]) {
   new MainAsp().start(args);
}
}
```

Figure 2: Java class MainAsp

compiling Sun RMI using the Manta compiler, all performance differences can be attributed to the RMI implementation and protocol, as both the sequential execution and the network (Myrinet) are identical. We did not investigate the performance impact of having multiple group members per node because this is only sensible on shared-memory nodes (SMP) which are not available to us.

3.1 Message passing subsystem

CCJ implements algorithms for collective communication based on individual messages between group members. The messages have to be simulated using the RMI mechanism. The basic difference between a message and an RMI is that the message is asynchronous (the sender does *not* wait for the receiver) while RMIs are synchronous (the client has to wait for the result from the server before it can proceed). Sending messages asynchronously is crucial for collective communication performance because each operation requires multiple messages to be sent or received by a single group member. CCJ simulates asynchronous messages using multithreading: send opera-

```
class Asp {
int n, rank, nodes;
int[][] tab;
Asp (int n) throws Exception {
    this.n = n;
}
void do_asp() throws Exception {
    int k;
    for (k = 0; k < n; k++) {
        // send the row to all other members
        if (tab[k] == null) tab[k] = new int[n];
        MPI.COMM_WORLD.Bcast(tab[k], 0, n,
                             MPI.INT, owner(k));
        // do ASP computation...
    }
}
public void run() {
    rank = MPI.COMM_WORLD.Rank();
    nodes = MPI.COMM_WORLD.Size();
    // initialize local data
    do_asp();
}
public static void main(String args[]) {
    int N;
    try {
        // get number of rows from command line
        MPI.Init(args);
        MPI.Finalize();
        System.exit(0);
    } catch (MPIException e) {
        // Handle exception... Quit.
    }
}
}
```

Figure 3: mpiJava code for ASP

tions are performed by separate sending threads. To reduce thread creation overhead, each member maintains a thread pool of available sending threads.

Unfortunately, multiple sending threads are run subject to the scheduling policy of the given JVM. Thus, messages may be received in a different order than they were sent. To cope with unordered message receipt, each member object also implements a list of incoming messages, for faster lookup implemented as a hash table. For uniquely identifying messages, CCJ not only uses the group and a message tag (like MPI does), but also a message counter per group per collective operation.

Table 1: Timing of CCJ's ping-pong messages

	time (μs)		
	Manta RMI		Sun RMI
ints	CCJ	RMI	CCJ
1	84	59	660
2	87	66	660
4	88	68	660
8	88	69	695
16	90	70	700
32	93	72	705
64	101	78	715
128	115	91	750
256	147	121	750
512	177	142	875
1024	259	206	975
2048	456	334	1250
4096	763	590	1655
8192	1400	1289	2725
16384	2662	2378	5010

We evaluated the performance of CCJ's messaging layer by a simple ping-pong test, summarized in Table 1. For CCJ, we measured the completion time of a member performing a send operation, directly followed by a receive operation. On a second machine, another member performed the corresponding receive and send operations. The table reports half of this round trip time as the time needed to deliver a message. To compare, we also let the same two machines perform a RMI ping-pong test.

We performed the ping-pong tests for sending arrays of integers of various sizes. Table 1 shows that with short messages (1 integer), CCJ's message startup cost (using Manta RMI) causes an overhead of 42 %. This is mainly caused by thread switching. With longer messages (16K integers, 64K bytes) the overhead is only about 12 % (again for Manta RMI) because in this case object serialization has a larger impact on the completion time. In Section 4 we compare CCJ-based applications with pure RMI versions of the same codes, showing that CCJ results in at least competitive application speed with less programming complexity.

Table 1 also shows the respective ping-pong times for CCJ using Sun RMI. These times are an order of magnitude higher and are clearly dominated by the Sun RMI software overhead. In the following discussion of CCJ's collective operations, we also show completion times using Sun RMI which are much higher, as can be expected from the ping-pong measurements. For brevity, we do not discuss them individually.

3.2 Collective communication operations

We will now present the implementations of CCJ's collective communication operations. CCJ implements well known algorithms like the ones used in MPI-based implementations [15, 18]. The performance numbers given have been obtained using one member object per node, forcing all communication to use RMI.

3.2.1 Barrier

In CCJ's barrier, the M participating members are arranged in a hypercube structure, performing remote method invocations in $\log M$ phases. The RMIs have a single object as parameter. If the number of members is not a power of 2, then the remaining members will be appended to the next smaller hypercube, causing one more RMI step. Table 2 shows the completion time of CCJ's barrier, which scales well with the number of member nodes. The barrier implementation is dominated by the cost of the underlying RMI mechanism.

Table 2: Completion time of CCJ's barrier

	time (μs)	
members	Manta RMI	Sun RMI
1	<1	<1
2	78	580
4	166	1170
8	273	1840
16	380	2800
32	478	5510
64	605	11700

3.2.2 Broadcast

CCJ's broadcast arranges the group members in a binomial tree. This leads to a logarithmic number of communication steps. Table 3 shows the completion times of CCJ's broadcast with a single integer and with an array of 16K integers. Again, the completion time scales well with the number of member objects. A comparison with Table 1 shows that the completion times are dominated by the underlying RMI mechanism, as with the barrier operation.

3.2.3 Reduce/Allreduce

CCJ's reduce operation arranges the M participating members in a binomial tree, resulting in $\log M$ communication steps. In each step, a member receives the data from one of its peers and reduces it with its own data. In the next step, the then combined data is forwarded further up the tree.

Table 3: Completion time of CCJ's broadcast

	time (μs)			
	Manta RMI		Sun RMI	
members	1 int	16K int	1 int	16K int
1	<1	1	< 1	1
2	86	2306	760	4490
4	156	4562	1440	8960
8	222	6897	2160	13840
16	292	9534	3020	18940
32	374	11838	5950	26400
64	440	14232	13700	41700

Table 4 shows the completion time for four different test cases. Reductions are performed with single integers, and with arrays of 16K integers, both with two different reduce operations. One operation, labelled *NOP*, simply returns a reference to one of the two data items. With this non-operation, the reduction takes almost as long as the broadcast of the same size, caused by both using binomial communication trees. The second operation, labelled *MAX*, computes the maximum of the data items. Comparing the completion times for *NOP* and *MAX* shows the contribution of the reduction operator itself, especially with long messages.

Table 4: Completion time of CCJ's reduce

	time (μs)					
	Manta RMI				Sun RMI	
	MAX		NOP		NOP	
members	1 int	16K int	1 int	16K int	1 int	16K int
1	1	1	1	1	1	1
2	90	3069	88	2230	740	4460
4	158	6232	152	4539	1450	9160
8	223	9711	225	6851	2200	14460
16	294	13520	290	9359	3190	20080
32	368	17229	356	12004	5570	27420
64	453	21206	437	14657	11010	46020

CCJ's Allreduce is implemented in two steps, with one of the members acting as a root. In the first step, a Reduce operation is performed towards the root member. The second step broadcasts the result to all members. The completion times can thus be derived from adding the respective times for Reduce and Broadcast.

3.2.4 Scatter

MPI-based implementations of Scatter typically let the root member send the respective messages directly to the other members of the group. This approach works well if messages can be sent in a truly asynchronous manner. However, as CCJ has to perform a thread switch per message sent, the related overhead becomes prohibitive, especially with large member groups. CCJ thus follows a different approach that limits the number of messages sent by the root member. This is achieved by using a binomial tree as communication graph. In the first message, the root member sends the data for the upper half of the group members to the first member in this half. Both members then recursively follow this approach in the remaining subgroups, letting further members forward messages. This approach sends more data than strictly necessary, but this overhead is almost completely hidden because the additional sending occurs in parallel by the different group members.

Table 5: Completion time of CCJ's scatter

	time (μs)			
	Manta RMI			Sun RMI
	1 int × mbr.	16K int × mbr.		16K int × mbr.
mbr.	scatter	scatter	broadcast	scatter
1	3	1251	<1	1290
2	188	4381	4480	6740
4	375	12790	16510	17330
8	595	26380	48920	39510
16	935	55196	126490	84350
32	1450	112311	315840	178630
64	2523	225137	798150	426010

Table 5 shows the completion time for the scatter operation. Note that, unlike with broadcast, the amount of data sent increases with the number of members in the thread group. For example, with 64 members and 16K integers, the size of the scattered rootObject is 4MB. But still, the completion time scales well with the number of group members. To compare CCJ's scatter with an upper bound, the table also shows the completion time for broadcasting the same (increasing) amount of data to the same number of members. The scatter operation clearly stays far below the time for broadcasting, except for the trivial case of a single member where broadcast simply has to return a reference to the given object.

3.2.5 Gather/Allgather

CCJ implements the gather operation as the inverse of scatter, using a binomial tree structure. With gather, the messages are combined by intermediate member nodes and sent further up the tree. Table 6 shows that the completion times are comparable to the ones of the scatter operation. However, times vary because the sending of the individual members towards the root member happens in a less synchronized fashion, allowing for more overlap. In almost all cases, gather performs slightly faster than scatter. CCJ's allgather operation is implemented by a gather towards one of the members, followed by a broadcast. Like with allreduce, the completion times can be derived from adding the respective timings.

Table 6: Completion time of CCJ's gather

	time (μs)		
	Manta RMI		Sun RMI
mbr.	1 int $\times$ mbr.	16K int $\times$ mbr.	16K int $\times$ mbr.
1	< 1	433	410
2	113	4239	5930
4	209	11646	16450
8	345	25514	37400
16	568	52902	79590
32	985	106965	166370
64	1663	248827	412630

3.3 Using non-array data structures

With Broadcast and Reduce, non-array data structures are transparently handled by Java's object serialization. However, for Scatter and Gather operations, CCJ's DividableDataObjectInterface has to be implemented by the respective object classes. To evaluate this interface, we have implemented and benchmarked two different matrix data structures, DenseMatrix and SparseMatrix.

3.3.1 DenseMatrix

The DenseMatrix data structure consists of an object which contains an ordinary 2-dimensional array of doubles (see Figure 4). Since real multi-dimensional arrays are not supported in Java, the data is actually stored in an array of arrays.

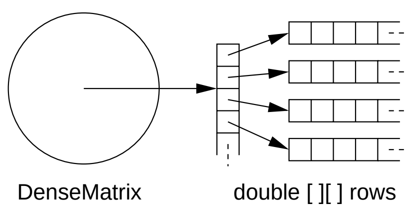

Figure 4: DenseMatrix

To allow the use of the Scatter and Gather operations of CCJ, the DenseMatrix object implements the DividableDataObjectInterface. It therefore has to implement two methods, elementAt (not shown), which is used in the scatter operation, and setElementAt, which is used in the Gather operation. (See Figure 5.)

When the members need to combine their local DenseMatrix objects into a single DenseMatrix , each of them calls the gather method, passing their local objects as a parameter. The root node of the gather also passes an extra DenseMatrix object as a parameter, which will contain the result of the gather operation. The setElementAt method will repeatedly be called on this result object, each time with one of the local objects as a parameter. The data inside the local object will then be copied into the correct position in the result object.

```
public void setElementAt(int index,
                         int groupSize,
                         Serializable object) {

        DenseMatrix src = (DenseMatrix) object;

        for (int i = 0 ; i < src.size() ; i++) {
            row[src.offset+i] = src.row[i];
        }
}
```

Figure 5: setElementAt method of DenseMatrix

3.3.2 SparseMatrix

A sparse matrix is a matrix containing mostly zeros, which, to save memory, should not be stored. We have implemented a SparseMatrix object, which is shown in Figure 6. The SparseMatrix object contains an array of Row objects, which are used to store the matrix rows in a compressed form. Every Row object contains two arrays, a data array, and an index array. The data array stores all the non-zero values of the row. The index array is used to store the original position of each of these data values, (e.g. the position it would have in the row of a non-sparse matrix).

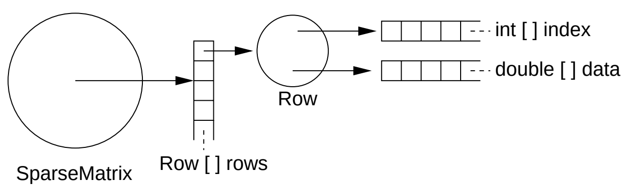

Figure 6: SparseMatrix

The SparseMatrix object requires more memory per data item than the DenseMatrix. However, if the amount of non-zero data inside the SparseMatrix is small enough, the memory saved by not storing zeros is greater than the extra cost of the more complex data structure. In this particular case (with doubles as data), the SparseMatrix is more efficient if more than approximately 19 % of the data consists of zero values.

The SparseMatrix also implements the DividableDataObjectInterface to allow the use of the scatter and gather operations of CCJ. Figure 7 shows the elementAt method.

To distribute a SparseMatrix over the members, the scatter operation of CCJ can be used. The CCJ library will then repeatedly invoke elementAt on the SparseMatrix object, each time passing it a member number as an index. The elementAt method calculates the sub matrix to send to this member, and creates a new SparseMatrix object containing this sub matrix. This new object can then be sent to the destination member.

```
public Object elementAt(int index,
                        int groupSize) {

    /* calculate the i-th part of the matrix */
    int size     = n / groupSize;
    int leftover = n % groupSize;
    int offset   = index * size;

    if (index >= (groupSize - leftover)) {
        size   += 1;
        offset += index - (groupSize - leftover);
    }
    /* return a new sub-matrix */
    return new SparseMatrix(this, offset, size);
}
```

Figure 7: elementAt method of class SparseMatrix

3.3.3 Performance

As a benchmark, we have measured the time required by the Scatter operation to distribute a DenseMatrix and a SparseMatrix across a number of members. Each matrix object contains 512x512 doubles. We have used two different SparseMatrix objects, one containing 95 % zeros, and one containing 50 % zeros.

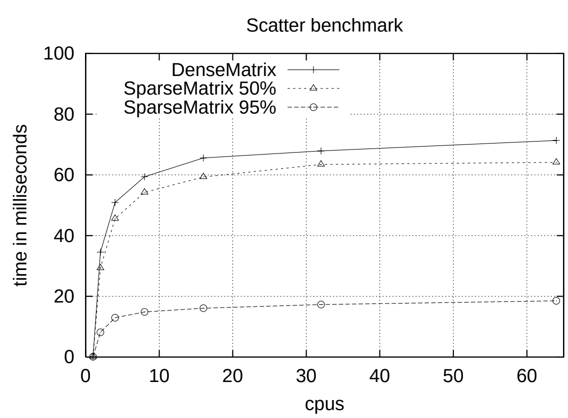

Figure 8: Matrix scatter benchmark

As Figure 8 shows, the time required to scatter the objects grows rapidly with the number of members (CPUs). The DenseMatrix, which contains the most data, takes 35 milliseconds to scatter the object to two members. When we scatter to 64 members, the time required grows to 71 milliseconds. As expected, the SparseMatrix containing 95 % zeros requires significantly less time, 8 milliseconds when scattering to two members, 19 milliseconds when scattering to 64. If we decrease the number of zeros in the SparseMatrix to 50 %, it still requires less time per scatter than the DenseMatrix, varying from 29 to 64 milliseconds.

4 Application programs

In this section we discuss the implementation and performance of three applications of CCJ, running both over Manta RMI and Sun RMI. We also compare the code complexity and performance of these programs with RMI versions of the same applications, measured using Manta RMI. Furthermore, we compare runtimes to *mpiJava* versions of our applications. For this purpose, we ported the *mpiJava* library [2] to Manta. Originally, *mpiJava* calls a C-based MPI library (in our case MPICH) via the Java native interface (JNI). We compiled *mpiJava* with the Manta compiler after replacing all JNI calls to direct C function calls, the latter to eliminate the high JNI overhead [13]. Unfortunately, *mpiJava* is not thread safe; so we had to disable Manta's garbage collector to avoid application crashes. Taking these two changes (direct C calls and no garbage collection) into account, the given results are biased in favour of *mpiJava*. We report speedups relative to the respectively fastest of the four versions on one CPU.

4.1 All-pairs Shortest Paths Problem

The All-pairs Shortest Paths (ASP) program finds the shortest path between any pair of nodes in a graph, using a parallel version of Floyd's algorithm. The program uses a distance matrix that is divided row-wise among the available processors. At the beginning of iteration k, all processors need the value of the kth row of the matrix. The processor containing this row must make it available to the other processors by broadcasting it.

In the RMI version, we simulate this broadcast of a row by using a *binary tree*. When a new row is generated, it is forwarded to two other machines which store the row locally and each forward it to two other machines. As soon as a row is forwarded, the machine is able to receive a new row, thus allowing the sending of multiple rows to be pipelined. The forwarding continues until all machines have received a copy of the row. In the CCJ and *mpiJava* versions, the row can be broadcast by using collective operations, as shown in Figures 1 and 3.

Figure 9 shows the speedups for a 2000x2000 distance matrix. The speedup values are computed relative to the CCJ/Manta RMI version on one node, which runs for 1074 seconds. The fastest parallel version is *mpiJava* with a speedup of 60.4 on 64 nodes, followed by the RMI version (59.6), CCJ/Manta RMI (57.3), and finally CCJ/Sun RMI (30.1).

We have also calculated the code size of the CCJ and RMI versions of ASP, by stripping the source of comments and whitespace, and then counting the number of bytes required for the entire program. The RMI version of ASP is 32 % bigger than the CCJ version. This difference in size is caused by the implementation of the broadcast. In the RMI version, this has to be written by the application programmer and contributes 48 % of the code. The communication related code in the CCJ version is used to partition the data among the processors, and takes about 17 % of the code. The broadcast itself is already implemented in the library.

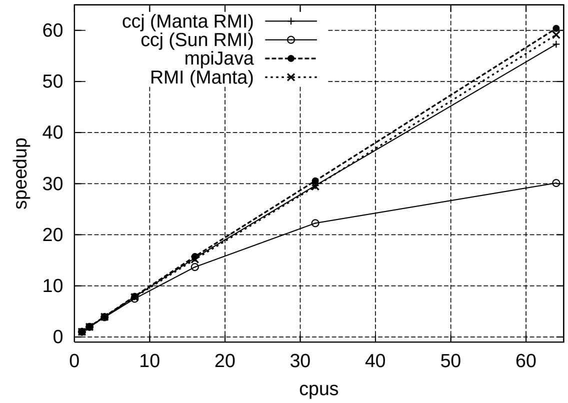

Figure 9: Speedup for the ASP application

4.2 QR Factorization

QR is a parallel implementation of QR factorization. In each iteration, one column, the *Householder vector* H, is broadcast to all processors, which update their columns using H. The current upper row and H are then deleted from the data set so that the size of H decreases by 1 in each iteration. The vector with maximum norm becomes the Householder vector for the next iteration. To determine which processor contains this vector, an allreduce collective operation (using an object as parameter) is used. In the RMI version, both the broadcast and allreduce operations are implemented using a binary tree algorithm.

Figure 10 shows the results for a 2000x2000 matrix. All speedup values are computed relative to the CCJ version on one node, which runs for 1991 seconds. As Figure 10 shows, the CCJ/Manta RMI version of QR has a better speedup than the RMI version, 41.4 against 31.6. This difference is caused by the efficient implementation of the allreduce operation in the CCJ library. The mpiJava and CCJ/Sun RMI versions have significantly lower speedups, 15.7 and 13.8 on 64 cpus. This is caused by serialization overhead of the object parameter used in the allreduce operation. Both the CCJ/Manta RMI and the RMI version use the efficient serialization offered by Manta, while the mpiJava and CCJ/Sun RMI version can only use the (much slower) standard serialization.

The code size of the RMI version is 44 % larger than the CCJ version. Furthermore, 41 % of the code of the RMI version is communication related. The CCJ version has only 10 % communication related code. Only an implementation of the *ReductionObjectInterface* is required. The actual implementations of the allreduce and the broadcast are hidden in the CCJ library. In the RMI version, however, this has to be implemented by the application programmer.

4.3 Linear Equation Solver

Linear equation solver (LEQ) is an iterative solver for linear systems of the form $Ax = b$. Each iteration refines a candidate solution vector x_i into a better solution x_{i+1}. This is repeated until the difference between x_{i+1} and x_i becomes smaller than a specified bound.

The program is parallelized by partitioning a dense matrix containing the equation coefficients over the processors. In each iteration, each processor produces a part of the vector x_{i+1}, but needs

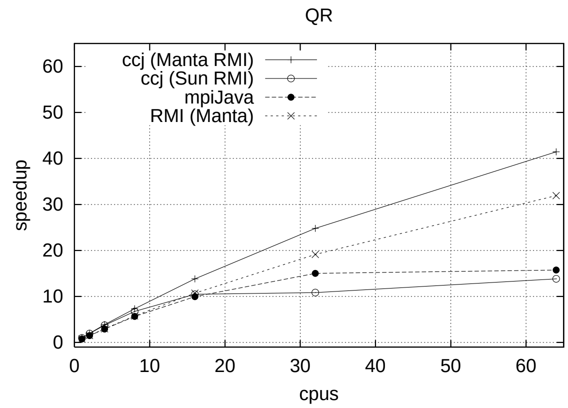

Figure 10: Speedup for the QR application

all of vector x_i as its input. Therefore, all processors exchange their partial solution vectors at the end of each iteration using an allgather collective operation. Besides exchanging their vectors, the processors must also decide if another iteration is necessary. To do this, each processor calculates the difference between their fragment of x_{i+1} and x_i. An allreduce collective operation is used to process these differences and decide if the program should terminate.

Figure 11 shows the results for a 1000x1000 matrix. All speedup values are computed relative the CCJ/Manta RMI version on one node, which runs for 1708 seconds.

In the RMI version, the vector fragments and values to be reduced are put in a single object, which is broadcast using a binary tree. Each processor can then locally assemble the vector and reduce the values. Unlike the previous programs, in which one processor was broadcasting, in LEQ all processors are required to broadcast data. This requires a large number of RMIs to complete the communication, causing more overhead than in the previous programs. For example, on 64 processors, 4032 RMIs are needed per iteration, while ASP only needs 63 RMIs per iteration. Due to this overhead the speedup of the RMI version is only 13.9 on 64 processors.

In the CCJ versions of LEQ, both the allgather and allreduce collective operations can be called directly from the library. Using the efficient allgather and allreduce implementations of CCJ, only 252 RMIs are required on 64 nodes. For CCJ/Manta RMI, the result is a better speedup than the RMI version: 16.3 on 64 nodes. However, for CCJ/Sun RMI, hardly any speedup is achieved (2.3 on 64 nodes). The *mpiJava* version is the fastest one with a speedup of 32.4 on 64 nodes, due to a better (ring) algorithm for allgather inside MPICH. However, CCJ can be improved by adopting this algorithm.

The RMI version of LEQ is 72 % larger than the CCJ version. As with QR, this is is caused by the communication related code, which makes up 67 % of the RMI version, but only 29 % of the CCJ version. The CCJ version only requires an implementation of the interfaces ReductionObjectInterface and DividableDataObjectInterface. The implementation of allreduce and allgather are hidden in the CCJ library.

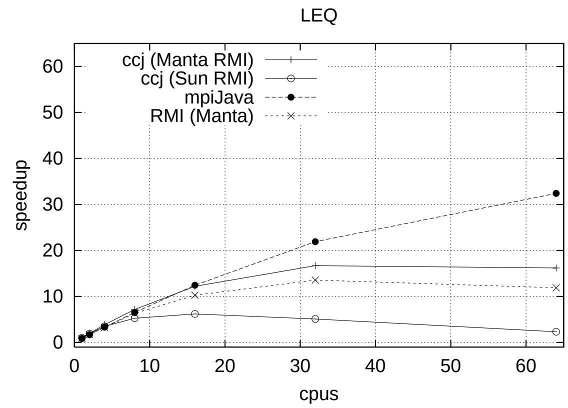

Figure 11: Speedup for the LEQ application

5 The Java Grande Forum Benchmarks

In the previous section, we evaluated the performance of CCJ with three applications. We emphasized the impact of the underlying RMI mechanism, by comparing Sun RMI with Manta RMI. We also compared to application versions using mpiJava or plain RMI for communication between processes. In this section, we give a more detailed assessment of CCJ's performance based on the MPJ benchmark suite from the Java Grande Forum [26]. We compare CCJ using the fast Manta RMI with the mpiJava version described in the previous section. From the MPJ benchmark suite, we generated two versions. The first one adapts the MPJ syntax to mpiJava's syntax. The second one uses CCJ's syntax. Before benchmarking, we also had to fix problems in the source codes of the SOR kernel, and of the applications MolDyn, MonteCarlo, and RayTracer.

The Java Grande MPJ benchmark suite consists of three sections. Section 1 is benchmarking low level operations, like message pingpong and collective communication operations like barrier and broadcast. Section 2 consists of five application kernels carrying out specific operations frequently used in Grande applications. Section 3, finally, consists of three larger codes, representing complete Grande applications. In the following, we present results for all benchmarks from the MPJ suite, except for the Alltoall communication benchmark from Section 1 because CCJ does not implement an Alltoall operation.

5.1 Low Level Operations

The Java Grande low level benchmarks produce a great variety of result data. For brevity, we restrict our discussion on the completion times of the investigated operations. They are shown in Figures 12 and 13. The original benchmarks send arrays of integer values of varying sizes. This is the most simple case for communication benchmarks. As the focus of CCJ is on objects, we wrote a second set of low level benchmarks with a more complex object structure. We chose to benchmark the sending of two-dimensional arrays (matrices), which in Java are objects with vector sub-objects. Benchmarking with two-dimensional arrays is attractive because they trigger the mechanisms for transmitting complex objects (instead of simple arrays). They can also be easily sized to have the same number of integer elements as the linear arrays used in the original Java Grande suite.

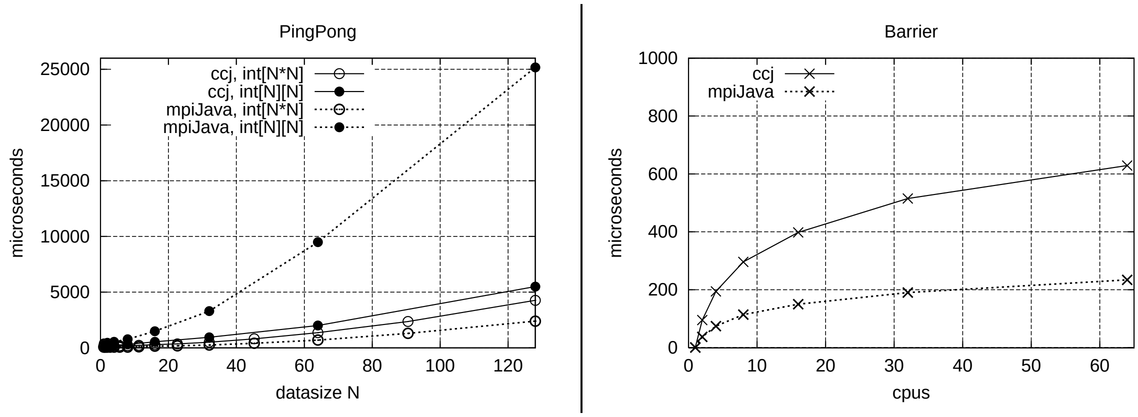

Figure 12: Completion times of low-level operations from the Java Grande Forum benchmarks

Figure 12 (left) shows the completion times of sending (**PingPong**) messages of varying size between two nodes. For sending one-dimensional arrays, mpiJava is faster than CCJ, because in this case no object serialization is necessary for mpiJava. Instead, the data can be copied directly from the array to the network and, at the receiver side, in the reverse order from the network into the array. CCJ, however, always uses object serialization because it is implemented on top of RMI. This implies that the receiver has to create a new array into which the data can be received. This additional (copying) overhead is the reason why mpiJava is faster than CCJ when sending simple arrays.

However, when sending two-dimensional matrices, mpiJava suddenly becomes significantly slower than CCJ. This is because, in this case, mpiJava first has to serialize the matrix into a byte stream to be sent to the receiver. The receiver, first has to create a new byte array for receiving and to deserialize the objects of the matrix in turn. For CCJ, hardly anything changes with matrices; just the number of transmitted objects slightly increases.

On the right side, Figure 12 shows the completion times of the **Barrier** operation with a varying number of CPUs. Although both implementations use the same basic algorithm, MPICH's Barrier (used by mpiJava) is more efficient than CCJ's message passing on top of RMI.

Figure 13 shows the completion times of the low level benchmakrs for the operations Bcast, Scatter, Gather, and Reduce. On the left side, times are shown for short messages (1 integer). On the right side, times are shown for long messages (16K integers). The results basically confirm he findings from the pingpong and barrier tests. The **Bcast** of simple arrays is faster with mpiJava than with CCJ. For example, a broadcast of a short (1 integer) array to 64 CPUs takes 63 μs with mpiJava, and 572 μs with CCJ. However, sending complex (matrix) objects is much more inefficient with the mpiJava implementation. Broadcasting a matrix of size 128×128 to 64 CPUs only takes 16,456 μs with CCJ, but 531,000 μs with mpiJava.

With the **Scatter** benchmark, mpiJava has an even higher advantage over CCJ, when scattering linear arrays. This is because the underlying MPI implementation (MPICH) can send parts of arrays, without further copying. With RMI, however, only complete objects can be transferred, so

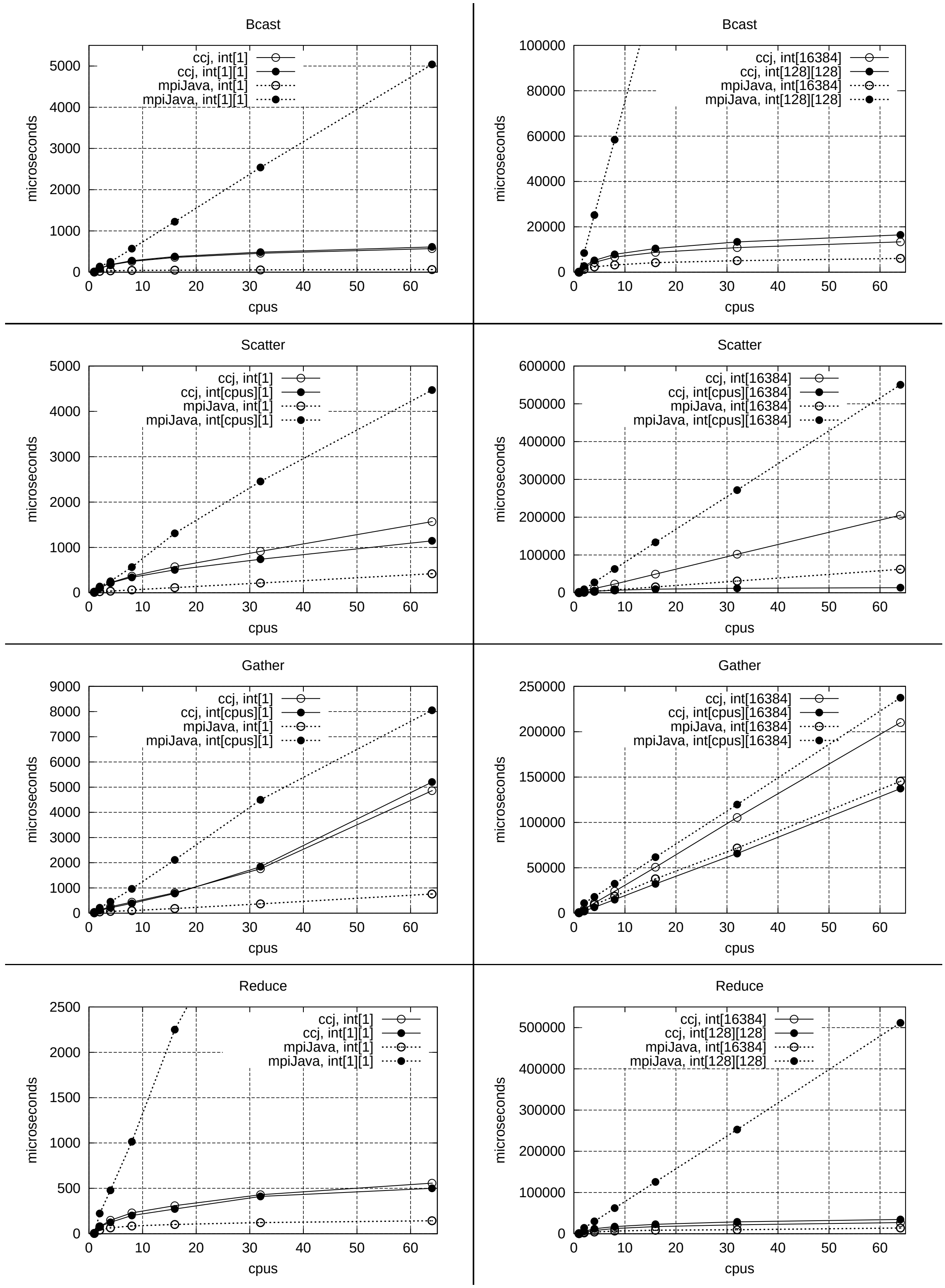

Figure 13: Completion times of low-level operations from the Java Grande Forum benchmarks

CCJ first has to create new objects for each receiver, which can then be transferred. But when scattering matrices, CCJ can directly serialize the row objects, without further copying, so it becomes faster than mpiJava with matrix objects.

With the **Gather** benchmark, the issue with partial objects is the same as with Scatter. However, all the copying overhead is concentrated at the receiver side. With short messages, CCJ performs equally with arrays and with matrices. mpiJava, however, is much faster for gathering a short array, but it is also much slower with a small matrix. With large messages, CCJ is slightly faster than mpiJava for both array and matrix because of the smaller copying overhead at the receiver side.

For the **Reduce** benchmark, we use element-wise summation both for array and matrix. The completion times are similar than with other low-level benchmarks. With arrays, mpiJava is faster than CCJ, but is much slower with matrices. With mpiJava, reducing a small matrix from 64 CPUs takes 8,680 μs while reducing a small array only takes 142 μs.

5.2 Kernels

Figure 14 shows the speedups achieved with both kernels and applications from the Java Grande MPJ benchmarks (Sections 2 and 3). We report speedups relative to the respectively fastest version on a single CPU. The kernels come in three problem sizes each, ranging from A (small), over B (medium), to C (large).

The **Series** kernel computes Fourier coefficients of a function in a given interval. It constitutes an embarrassingly parallel application, because the CPUs only compute at the end of the run. Here, arrays of double values are sent from all nodes to CPU 0 using individual messages. As expected, both mpiJava and CCJ achieve almost linear speedups with all problem sizes.

The **LUFact** kernel performs a parallel LU factorization, followed by a sequential triangular solve. The CPUs communicate by broadcasting arrays of doubles and integers. As can be expected from the low-level benchmarks, mpiJava is somewhat faster than CCJ in this case.

The **SOR** kernel performs 100 iterations of successive over-relaxation. At the beginning, matrix blocks are distributed to all CPUs. For each iteration, the neighbors exchange arrays of double values. Because of the transmission of matrix blocks, CCJ achieves higher speedups than mpiJava.

The **Crypt** kernel performs IDEA (International Data Encryption Algorithm) encryption and decryption on a byte array. The array is created on CPU 0 and then sent to the other CPUs by individual messages. At the end of the application, the computed results are then sent back to CPU 0 by individual messages. It is unclear why this kernel does not use the Scatter and Gather operations instead. Because only simple arrays are transferred, the mpiJava versions are moderately faster than the CCJ versions. With the largest problem size C, the CCJ version shows a degraded speedup which is due to memory problems, caused by necessary, additional array copying.

The **SparseMatmult** kernel performs multiplication of a sparse matrix stored in compressed-row format, using one array of double values and two integer arrays. First, CPU 0 creates the data and distributes the sparse matrix across the CPUs, using individual messages. The results of the individual computations are combined by an Allgather operation. Unfortunately, this kernel does not achieve any speedups, neither with mpiJava, nor with CCJ.

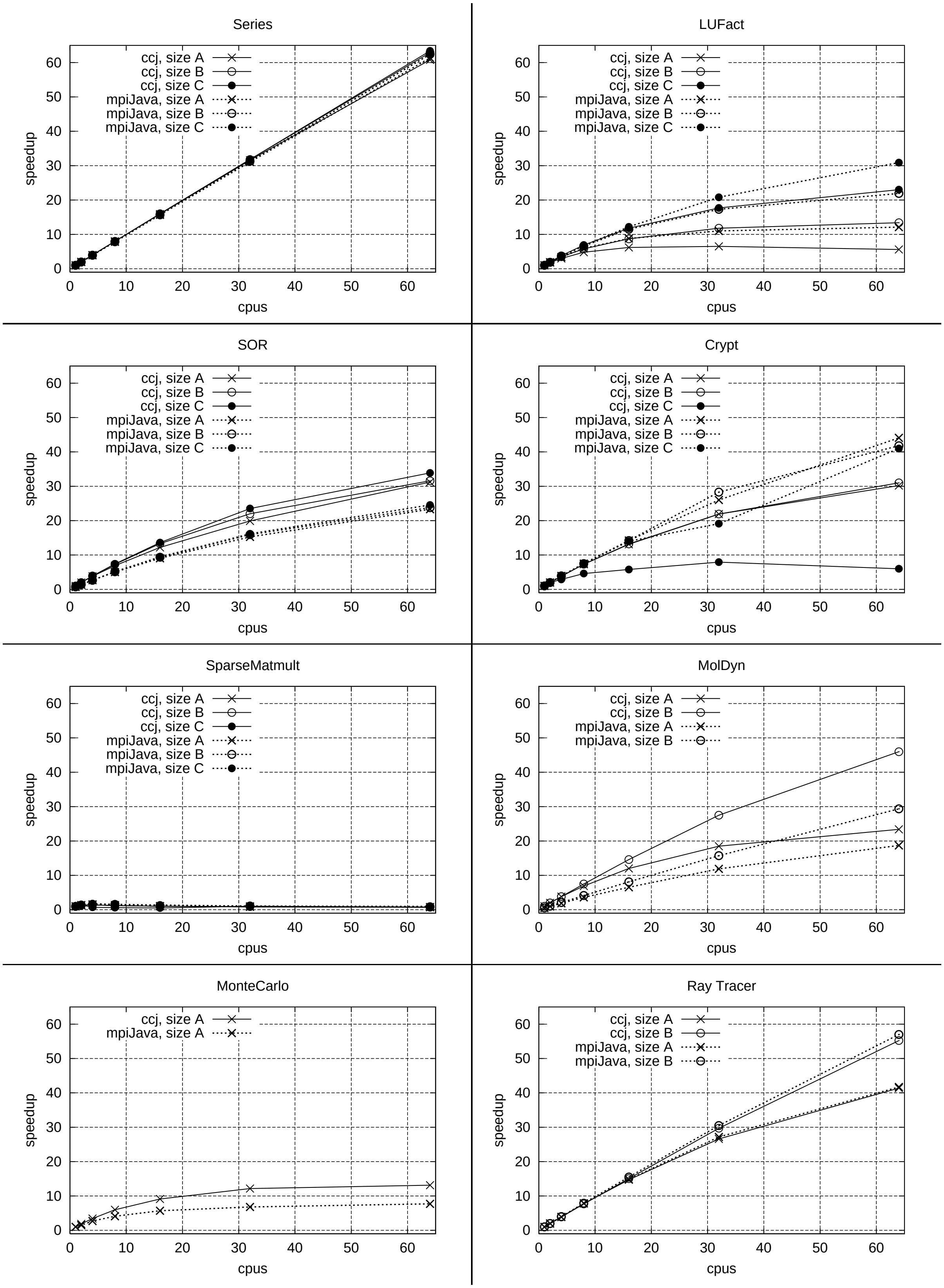

Figure 14: Speedups of kernels and applications from the Java Grande Forum benchmarks

5.3 Large Applications

Figure 14 also shows the speedups achieved with the applications from the Java Grande MPJ benchmarks (Section 3). We report speedups relative to the respectively fastest version on a single CPU. The applications come in two problem sizes each. A denotes small, and B denotes large.

The **MolDyn** application is an N-body code modeling particles interacting under a Lennart-Jones potential in a cubic spatial volume with periodic boundary conditions. For each iteration, 6 Allreduce (summation) operations are used to update the particles (3 times with double arrays, 2 times with a double value, and once with an integer value). With MolDyn, CCJ clearly outperforms mpiJava. The advantage could be even bigger if the individual Allreduce operations would be combined to a single operation on a complex object, which is possible with CCJ but not with mpiJava.

The **MonteCarlo** application is a financial simulation for pricing products derived from the price of an underlying asset. At the beginning of the computation, all nodes read in a file and simulate parts of the problem. The result on each node is an array of java.util.Vector objects. These arrays of complex objects are sent to CPU 0 by individual messages. With problem size A, CCJ achieves better speedups than mpiJava. We could not run problem size B because it exceeds the memory size of our compute nodes.

The **Ray Tracer** application renders a scene of 64 spheres. Each CPU renders part of the scene which is simultaneously generated on all nodes. The CPUs send the rendered pixels to CPU 0 by individual messages. The speedups achieved by mpiJava and by CCJ are almost identical.

6 Related work

The driving force in high-performance Java is the Java Grande Forum (www.javagrande.org). There are also many other research projects for parallel programming in Java [1, 6, 7, 14, 16, 25]. Most of these systems, however, do not support collective communication. Taco [24] is a C++ template library that implements collective operations, however without exploiting MPI's concept of collective invocation by the participating processes. JavaNOW [27] implements some of MPI's collective operations on top of a Linda-like entity space; however, performance is not an issue.

In our previous work on parallel Java, we implemented several applications based on RMI and RepMI (replicated method invocation) [20, 21, 28]. There, we identified several MPI-like collective operations as being important for parallel Java applications. We found that collective operations both simplify code and contribute to application speed, if implemented well. CCJ implements efficient collective operations with an interface that fits into Java's object-oriented framework.

An alternative for parallel programming in Java is to use MPI instead of RMI. MPJ [9] proposes MPI language bindings to Java. These bindings merge several earlier proposals [2, 10, 17, 23]. This approach has the advantage that many programmers are familiar with MPI and that MPI supports a richer set of communication styles than RMI, in particular collective communication. However, the current MPJ specification is intended as "…initial MPI-centric API" and as "…a first phase in a broader program to define a more Java-centric high performance message-passing environment." [9] CCJ is intended as one step in this direction.

7 Conclusions

We have discussed the design and implementation of CCJ, a library that integrates MPI-like message passing and collective operations in a clean way into Java. CCJ allows Java applications to use collective communication, much like RMI provides two-party client/server communication. In particular, any data structure (not just arrays) can be communicated. Several problems had to be addressed in the design of CCJ. One issue is how to map MPI's communicator-based process group model onto Java's multithreading model. We solve this with a new model that allows two-phase construction of immutable thread-groups at runtime. Another issue is how to express user-defined reduction operators, given the lack of first-class functions in Java. We use function objects as a general solution to this problem.

CCJ is implemented entirely in Java, using RMI for interprocess communication. The library thus can run on top of any Java Virtual Machine. For our performance measurements, we use an implementation of CCJ on top of the Manta system, which provides efficient RMI. We have implemented three parallel applications with CCJ and we have compared their performance to mpiJava and hand-optimized RMI versions. For all three applications, CCJ performs faster or equally fast as RMI. Compared to mpiJava, CCJ performs equally fast with ASP and significantly faster with QR. For LEQ, the performance is worse than mpiJava, which is caused by a less-efficient allgather implementation. We have also compared the code complexity of the CCJ and RMI versions of the applications. The results show that the RMI versions are significantly more complex, because they have to set up spanning trees in the application code to do collective communication efficiently. We have shown that CCJ is an easy-to-use library for adding MPI-like collective operations to Java. Given an efficient RMI implementation, CCJ results in application runtimes that are competitive to other implementations.

We finally compared CCJ's performance to mpiJava in detail, using the Java Grande Forum MPJ Benchmark suite. We found that CCJ's simulation of individual messages with RMI and threads is moderately slower than sending individual messages directly. Also, when sending arrays of primitive data types, using an underlying MPI library (in our case MPICH) has less communication overhead than RMI with its object serialization. However, when transferring complex objects, CCJ causes less overhead, leading to better speedups for those kernels and applications from the Java Grande Forum benchmark that actually use objects instead of plain arrays. To conclude, CCJ is a viable alternative to existing message passing platforms for Java, because it combines competitive performance with a clean integration of message passing and collective communication into Java's object-based model.

8 Acknowledgements

This work is supported in part by a USF grant from the Vrije Universiteit. The DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, and Ceriel Jacobs for their contributions to this research. We thank Kees Verstoep and John Romein for keeping the DAS in good shape.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. A Java interface to MPI. In *Proc. First UK Workshop on Java for High Performance Network Computing*, Sept. 1998.

[3] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[4] R. Bhoedjang, T. Rühl, and H. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[5] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[6] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Santa Barbara, CA, Feb. 1998.

[7] S. Brydon, P. Kmiec, M. Neary, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. In *ACM 1999 Java Grande Conference*, pages 171–180, San Francisco, CA, June 1999.

[8] M. Burke, J.-D. Choi, S. Fink, D. Grove, M.Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.

[9] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[10] G. Crawford, Y. Dandass, and A. Skjellum. The jmpi commercial message passing environment and specification. http://www.mpi-softtech.com/publications/JMPI_121797.html, Dec. 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object–Oriented Software*. Addison Wesley, 1994.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[13] V. Getov. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Supercomputing'99*, Portland, OR, November 1999.

[14] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance Parallel Programming in Java: Exploiting Native Libraries. In *ACM 1998 workshop on Java for High-performance network computing*, Feb. 1998.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[16] T. Haupt, E. Akarsu, G. Fox, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, and C.-H. Youn. The Gateway System: Uniform Web Based Access to Remote Resources. In *ACM 1999 Java Grande Conference*, pages 1–7, San Francisco, CA, June 1999.

[17] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice and Experience*, 10:977–983, 1998.

[18] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999.

[19] T. Kühne. The function object pattern. *C++ Report*, 9(9):32–42, Oct. 1997.

[20] J. Maassen, T. Kielmann, and H. E. Bal. Efficient Replicated Method Invocation in Java. In *ACM 2000 Java Grande Conference*, pages 88–96, San Francisco, CA, June 2000.

[21] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[22] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[23] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In *Recent Advances in PVM and MPI*, number 1332 in Lecture Notes in Computer Science, pages 135–142. Springer, 1997.

[24] J. Nolte, M. Sato, and Y. Ishikawa. Template Based Structured Collections. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 483–491, Cancun, Mexico, 2000.

[25] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[26] The Java Grande Forum. The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/.

[27] G. K. Thiruvathukal, P. M. Dickens, and S. Bhatti. Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concunrrency: Practice and Experience*, 12:1093–1116, 2000.

[28] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming using the Remote Method Invocation Model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.