Object based Collective Communication in Java

Dhruvika Sharma, Aastha Sharma
Student, Information Technology, Maharishi Dayanand University
New Delhi, Delhi, India

***Abstract*- CCJ is a communication library that adds MPI-like collective operations to Java. Instead of trying to use the precise MPI syntax, CCJ focus on integrating 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, so that it can be used with any Java virtual machine. This paper discusses three parallel Java applications that use collective communication technique. 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.**

I. INTRODUCTION

Java a viable platform for high-performance computing due to recent improvements in compilers and communication mechanisms. As Java support multithreading and Remote Method Invocation (RMI) therefore it becomes suitable for writing parallel programs.RMI uses a familiar abstraction (object invocation), integrated in a 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 with support for object replication.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. Several projects have proposed to extendJava with MPI-like collective operations. Unlike RMI, the MPI primitives are biased towards array-based data structures.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.In this paper we present the CCJ library (Collective Communication in Java) which integrates the core of MPI's collective operations to Java's object model. CCJ maintains thread groups that can collecively 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. Ittherefore does not suffer from JNI overhead and it can be used with any Java virtual machine. 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.
CCJ's support for arbitrary data structures is useful for example in implementing sparse matrices.

II. OBJECTBASED 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 collective communication, as inspired by the MPI standard, into Java's object-based model. CCJ integrates MPI-like collective 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 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 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 userimplemented
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 usedto 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
to groupSize □

___

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.

2.3 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 applicationspecific
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 mpi-
Java 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.

```
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

```
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

III. 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 an internal 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 experimentation 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 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 operations 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.
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 barrier implementation is dominated by the cost of the underlying
RMI mechanism.

```
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

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
□
participating members are arranged in
a hypercube structure, performing remote method invocations in
_____ □
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.

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
□
participating members in
a binomial tree, resulting in ___ _ □
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 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.

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 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.

IV. 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 Cbased
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 Allpairs
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 □ , all processors need the value of the □ th 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.

V. 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 [26] 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, 27]. 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.

VI. CONCLUSIONS

In this paper we discussed design and implementation of CCJ, a library that integrates MPI-like collective operations into Java. CCJ allows Java applications to use collective communication, in a similar way like RMI provides two-party client/server communication. In particular, any data structure (not just arrays) can be communicated.. The issue of how to map MPI's communicator-based process group model onto Java's multithreading model is solved with a new model that allows two-phase construction of immutable thread-groups at runtime. Another issue of how to express user-defined reduction operators, given the lack of first-class functions in Java is solved with the usage of 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 JavaVirtual 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. In conclusion, we have shown that CCJ is an easy-to-use library for adding MPI-like collective operations to Java. With efficient RMI implementation, CCJ results in application runtimes that are competitive to other implementations.