Object-based Collective Communication in Java

Arnold Nelisse, Thilo Kielmann, Henri E. Bal, Jason Maassen

Faculty of Sciences, Vrije Universiteit, Amsterdam, The Netherlands
{arnold,kielmann,bal,jason}@cs.vu.nl

An extended version will appear in Joint Java Grande - ISCOPE 2001 Conference, San Francisco, California, June 2-4, 2001.

Keywords: Collective Communication, RMI, MPI, Performance

Abstract

CCJ is a communication library that adds MPI-like collective operations to Java. CCJ provides a clean integration of collective communication into Java's object-oriented framework. 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 two parallel Java applications based on CCJ. It compares code complexity and performance of these applications (on top of a Myrinet cluster) with versions written using Java RMI. The CCJ versions are significantly simpler than the RMI versions and on average obtain better performance.

1 Introduction

Recent improvements in compilers and communication mechanisms make Java a viable platform for high-performance computing. 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 [9, 12] and it can be extended seamlessly with support for object replication [8].

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 [10]. Several projects have proposed to extend Java with MPI-like collective operations. For example, MPJ [4] 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.

In this paper we present the CCJ library (Collective Communication in Java) which adds the core of MPI's collective operations to Java's object model. CCJ maintains thread groups that can collectively 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 (calling native C functions from Java) and it can be used with any Java virtual machine. We study CCJ's performance on top of a fast RMI system (Manta [9]) 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.

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 two application programs using CCJ. Section 5 presents related work, Section 6 concludes.

2 Object-based collective communication

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. First, we deal with the use and management

of thread groups. Then we present the collective operations implemented within CCJ and discuss their integration into Java's object model.

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. The ranking of processes is also necessary to define operations like scatter/gather data redistributions, where the data sent or received by each individual process is determined by its rank.

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.

CCJ maps MPI's immutable process groups onto Java's multithreading model by defining a model of thread groups that constructs immutable groups at runtime 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 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

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.

A member object can be part of many groups. We thus implement the collective operations as methods of the group members. The communication context (the group) intuitively becomes 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. 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 the respective application classes implement a reduce method that can be called from within the collective reduction operations. However, this approach is not feasible because it restricts each class to exactly one reduction operation and it excludes the basic (numeric) data types from being used in reduction operations.

In 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 (specified by String objects). However, a serious problem is that reflection uses runtime interpretation, causing prohibitive costs for use in parallel applications.

CCJ thus uses a different approach for implementing reduction operators: *function objects* [7]. CCJ's function objects implement the specific ReductionObjectInterface containing a single method Serializable reduce(Serializable obj1, Serializable obj2). 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 required 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 but returns 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 [5]. 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 rootObj, 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 rootObj 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 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 to de-register the member object. The necessity of the done method is an artefact 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 main method running 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.

```
class Asp extends ColMember {
ColGroup group;
int n, rank, nodes;
int[][] tab; // the distance table.

void setGroup(ColGroup group) {
   this.group = group;
}
Asp (int n) throws Exception {
   super();
   this.n = n;
}
void do_asp() throws Exception {
   int k;
   for (k = 0; k < n; k++) {
      // send the row to all members:
      tab[k] = (int[])
            broad-
cast(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

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 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 [6]. This section describes both the messaging layer and the collective algorithms of CCJ.

CCJ has been implemented using the Manta high performance Java system [9]. 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. Manta's runtime system has access to the network in user space via the Panda communication substrate [1]. The system is more fully described in *http://www.cs.vu.nl/das/*.

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 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 re-

```
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

ceipt, 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)	
ints	CCJ	RMI
1	84	59
4	88	68
16	90	70
64	101	78
256	147	121
1024	259	206
4096	763	590
16384	2662	2378

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 roundtrip 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 (4 bytes each) of various sizes. Table 1 shows that with short messages (1 integer), CCJ's message startup cost 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 % because in this case object serialization (inside RMI) has a larger impact on the completion time.

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 [6]. The performance numbers given have been obtained using one member object per node, forcing all communication to use RMI.

Barrier In CCJ's barrier, the M participating members are arranged in a hypercube structure, performing remote method invocations with empty parameters in $\log M$ phases. 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 while being dominated by the cost of the underlying RMI mechanism.

Broadcast CCJ's broadcast arranges the group members in a binomial tree. This leads to a logarithmic number of communication steps. Table 2 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. Due to pipelining effects, the measured times are a bit less than expected from the ping-pong tests.

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 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

Table 2: Completion time of CCJ's barrier and broadcast

	time (μs)		
	barrier	broadcast	
members		1 int	16K int
1	< 1	<1	1
2	78	86	2306
4	166	156	4562
8	273	222	6897
16	380	292	9534
32	478	374	11838
64	605	440	14232

two data items. With this non-operation, the reduction takes almost exactly 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 3: Completion time of CCJ's reduce

	time (μs)			
	1 int		16K int	
members	MAX	NOP	MAX	NOP
1	1	1	1	1
2	90	88	3069	2230
4	158	152	6232	4539
8	223	225	9711	6851
16	294	290	13520	9359
32	368	356	17229	12004
64	453	437	21206	14657

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.

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 4: Completion time of CCJ's scatter

	time (μs)		
	1 int $*$ members	16K int $*$ members	
members	scatter	scatter	broadcast
1	3	1251	<1
2	188	4381	4480
4	375	12790	16510
8	595	26380	48920
16	935	55196	126490
32	1450	112311	315840
64	2523	225137	798150

Table 4 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.

Gather/Allgather CCJ implements the gather operation as the inverse of scatter, using a binomial tree structure. With gather, messages are combined by intermediate member nodes and sent further up the tree. The completion times for gather are almost identical to the scatter operation and thus not shown here. 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.

4 Application programs

In this section we discuss the implementation and performance of two CCJ applications. We compare code complexity and performance of these programs with RMI versions of the same applications. We report speedups relative to the respectively fastest version 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 rowwise 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. The forwarding continues until all machines have received a copy of the row. In the CCJ version the row can be broadcast by using the collective operation, as shown in Figure 1.

Figure 3 shows the speedups for a 2000x2000 distance matrix. The speedup values are computed relative the CCJ version on one node, which runs for 1082 seconds. The RMI version has a speedup of 59.6 on 64 nodes, slightly better than the speedup of 56.9 of the CCJ version.

We have also calculated the source code size in bytes of both ASP versions, after removing comments and whitespace. 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, the broadcast 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.

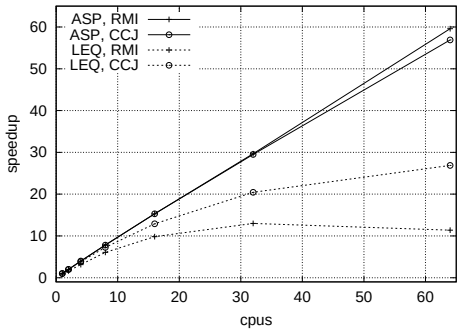

Figure 3: Speedup for ASP and LEQ

4.2 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 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 3 shows the results for a 1000x1000 matrix. All speedup values are computed relative the CCJ version on one node, which runs for 1716 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. In the CCJ version of LEQ, both the allgather and allreduce collective operations can be called directly from the library. With the efficient allgather and allreduce implementations of CCJ, the CCJ version achieves a speedup of 26.9 on 64 nodes, compares to only 11.4 of the RMI version.

The RMI version of LEQ is 72 % larger than the CCJ version. This is is caused by the communication code, which makes up 67 % of the RMI version, but only 29 % of the CCJ version.

5 Related work

There are many other research projects for parallel programming in Java [2, 3, 12]. Most of them, however, do not support collective communication. Taco [11] is a C++ template library that implements collective operations. JavaNOW [13] 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) [8, 9, 14]. 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 [4] proposes MPI language bindings to Java. 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 "...intial MPI-centric API" and as "...a first phase in a broader program to define a more Java-centric high performance message-passing environment." [4] CCJ is intended as one step in this direction.

6 Conclusions

We have discussed the design and implementation of CCJ, a library that integrates MPI-like 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 two parallel applications with CCJ and we have compared their performance and code complexity with RMI versions of the same 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. For one application (ASP), the RMI version is 4 % faster than the CCJ program. For the other program (LEQ), the CCJ version is significantly faster, up to a factor 2.4. In conclusion, we have shown that CCJ is an easy-to-use and efficient library for adding MPI-like collective operations to Java.

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] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.