GMI: Flexible and Efficient Group Method Invocation for Parallel Programming

Jason Maassen, Thilo Kielmann, Henri E. Bal

Division of Mathematics and Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands
{jason,kielmann,bal}@cs.vu.nl
http://www.cs.vu.nl/manta

Abstract. We present a generalization of Java's Remote Method Invocation (RMI) model providing an efficient group communication mechanism for parallel programming. Our *Group Method Invocation* (GMI) model allows methods to be invoked either on a single object or on a group of objects, the latter possibly with personalized parameters. Likewise, result values and exceptions can be returned normally (as with RMI), discarded, or, when multiple results are produced, combined into a single result. The different method invocation and reply handling schemes can be selected for each method individually at run time, and can be combined orthogonally. This allows us to express a large spectrum of communication mechanisms of which RMI is a special case. MPI-style collective communication can easily be implemented using GMI.

1 Introduction

Object-oriented languages like Java support distributed programming using the Remote Method Invocation (RMI) model. The key advantage of RMI is that it extends the object-oriented notion of method invocation to communication between processes. For parallel applications, however, many authors have argued that RMI is inadequate and that support for alternative forms of communication is also needed [6, 11]. For example, multicast may be implemented by using multiple RMI calls, but this is cumbersome and cannot exploit efficient low-level (hardware) multicast primitives. More complex communication, like personalized multicast or data reduction are even harder to express using RMI. For many parallel applications, communication with *groups* of objects is needed, both for code simplicity and application efficiency.

One approach to introduce group communication is to use a library such as MPI [6]. This increases expressiveness, but at the cost of adding a separate model based on message passing, which does not integrate with (method-invocation based) object models. MPI deals with static groups of processes rather than with objects and threads. In particular, MPI's collective operations must be explicitly invoked by all participating processes, forcing them to execute in lock-step, which is ill-suited for object-oriented programs.

In this paper, we will introduce an elegant approach to integrate flexible group communication[1] into an object-oriented language. Our goal is to express group communication using method invocations, just like RMI is used to express point-to-point communication. We therefore generalize the RMI model in such a way that it can express communication with a group of objects. This extended model is called *Group Method Invocation* (GMI). The key idea is to extend the way in which a method invocation and its result value are handled. A method invocation can be forwarded to a single object or to a group of objects (possibly personalizing the parameters for each destination). The result value (including exceptions) can be returned normally, discarded, forwarded to a separate object, or, when multiple results are produced, combined into a single result. All schemes for handling invocations can be combined with all schemes for handling results, giving a fully orthogonal design.

Due to this orthogonal approach, GMI is both simple and highly expressive. GMI can be used to implement MPI-style collective communication, where all members of a group collectively invoke the same communication primitive. Unlike MPI, GMI also allows a single thread to invoke a method on a group of objects, without requiring active involvement from other threads (e.g., to collect information from a group of objects). Because of GMI's orthogonal design, it can also express many communication patterns that have thus far been regarded as unrelated, special primitives. In particular, futures and voting can easily be expressed using GMI.

We have implemented GMI as part of the Manta high-performance Java system [12]. We will show that GMI can be implemented very efficiently. For example, invoking a method on a group of 64 remote objects takes about 70 μs on a Myrinet cluster.

The main contribution of the paper is a generalization of RMI to an expressive, efficient, and easy-to-use framework for parallel programming that supports a rich variety of communication patterns, while seamlessly integrating with object-oriented languages. In Sect. 2, we describe our GMI model and an example application. In Sect. 3, we discuss the performance of GMI operations and six applications on a 64-node Myrinet cluster. Sect. 4 presents related work, and Sect. 5 concludes.

2 The GMI Model

In this section, we will first summarize the RMI model and show how GMI generalizes it to support groups of objects. We will then describe how communication patterns within these groups can be used to implement many types of group communication.

[1] Please note that, in this paper, the term *group communication* refers to arbitrary forms of communication with groups of objects; in the operating systems community, the term is often used to denote *multicast*, which we regard as just one specific form of group communication.

2.1 Remote Method Invocation

RMI allows methods of an object to be invoked remotely. Such methods must be defined in a special *remote interface.* An object can be made suitable for receiving RMIs by implementing this remote interface. When this object is compiled, two extra objects are generated, a *stub* and a *skeleton.* The stub contains special, compiler-generated implementations of the methods in the remote interface, and can be used as a *remote reference* to the object. An example is shown in Fig. 1, where a stub acts as a remote reference to an object.

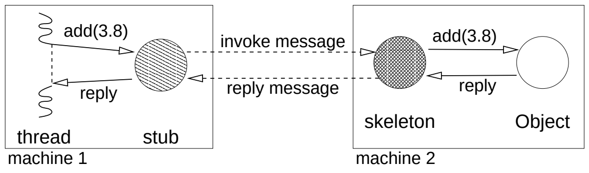

Fig. 1. A method invocation via a stub.

When a method of this stub is invoked, the stub *forwards* the invocation to the skeleton by sending a network message with a description of the method and its parameters (see Fig. 1). The skeleton waits for a message, decodes it, invokes the requested method on the object, and returns the method's result to the stub, which returns it to the invoker.

In RMI, *result handling* by stub and skeleton is fixed: they always operate synchronously. After the stub forwards the method to the skeleton, it waits for a reply message before continuing. The skeleton must therefore always send a reply back to the stub (even if the method has no result). Furthermore, a stub in RMI always serves as a remote reference to a single object, which can not be changed once the stub has been created.

2.2 Group Method Invocation

In GMI, we generalize the RMI model in three ways. First, a stub can be used as a reference to a group of objects, instead of just to a single one. In Fig. 2, the stub serves as a reference to a group of two objects.

Second, we generate communication code for stubs and skeletons that allows method invocations and result values to be handled in many different ways. As with RMI, exceptions are treated as special cases of result values.

Third, through a simple API, the programmer can configure the stubs and skeletons for each method individually *at run time.* Different forwarding and result-handling schemes can be combined, giving a rich variety of communication mechanisms. By configuring stubs at run time, the communication behavior of the application can easily be adapted to changing requirements.

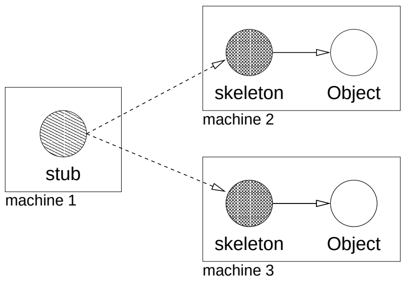

Fig. 2. A stub acting as group reference in GMI.

Below, we will first describe the notion of *object groups*, and then discuss the forwarding schemes, the result-handling schemes, and their combinations. We finally discuss synchronization issues on object groups.

Object Groups. An object group consists of several objects, possibly on different machines. Groups are created dynamically. For proper semantics of group operations, groups become immutable upon creation [14]. The objects in the group are ordered and can be identified by their *rank*. Ranks and the group size are available to the application.

The objects in a group may have different types, but they must all implement the same *group interface*, which serves a similar function as a remote interface: it defines which methods can be invoked on the group and makes the compiler generate stub and skeleton objects for this interface.

Forwarding Schemes. To support communication with multiple objects, GMI allows a stub to act as a reference to a group of objects. Depending on the application, the stub must be able to forward invocations to one or more objects in the group. GMI offers the following forwarding schemes:

– *single invocation*
  The invocation is forwarded to a single (possibly remote) object of the group, identified via its rank.
– *group invocation*
  The invocation is forwarded to every object in the group.
– *personalized group invocation*
  The invocation is forwarded to every object in the group, while the parameters are *personalized* for each destination using a user-defined method.

The first scheme is similar to a normal RMI. When configuring the stub, the programmer can specify to which group object the method invocations are forwarded. Unlike RMI, however, GMI allows this destination to be different for each invocation.

The second scheme can be used to express a multicast. The same method invocation (with identical parameters) is forwarded to each object in a group.

The last scheme is suitable for distributing data (or computations) over a group. Before the invocation is forwarded to each of the group objects, a user-defined *personalization* method is invoked. This method serves as a filter for the parameters to the group method. It gets as input the parameters to the group method, the rank of the destination object, and the size of the group. It produces a personalized set of parameters as output. These parameters are then forwarded to the destination object. Thus a *personalized* version of the call is sent to each group object.

In GMI, each method in a stub can be configured separately, at runtime, to use a specific forwarding scheme. As a result, the group stub can be configured in such a way that each of its methods behaves differently. By configuring a method in a stub, the programmer selects which compiler-generated communication code is used for method forwarding. For every method in a group interface, the compiler generates communication code for each forwarding scheme. Although this increases the code size of the stub, it allows the compiler to use compile-time information to generate optimized communication code for each method.

Result-Handling Schemes. RMI only supports synchronous method invocations, which is too restrictive for parallel programming. GMI therefore does not require the invoker to always wait for a result. Instead, it offers a variety of result-handling schemes that can be used to express asynchronous communication, *futures*, and other primitives. The skeletons offer the following result handling schemes:

- *discard results*
  No results are returned to the stub. They are discarded by the skeleton.
- *forward results*
  All results are returned to the stub, which forwards them to a user-defined *handler* object (rather than the original invoker).
- *return one result*
  A single result is returned to the stub. If the method is invoked on a single object, the result can be returned directly. Otherwise, one of the results (selected via a rank) is returned to the stub.
- *combine results*
  Combine all results into a single one (using a user-defined filter method). The final value is returned to the stub.

Discard results allows a method to be invoked asynchronously. The stub will forward the invocation to one or more skeletons and return immediately (without waiting for a reply). If the method returns a result, a default value (e.g., '0.0' or a *Null* reference) will be returned. Any result value returned (or exception thrown) by the method is directly discarded by the skeletons.

The second scheme, *forward results*, allows results (including exceptions) to be forwarded to a separate *handler* object. The stub will forward any method invocations, and return immediately. However, the skeletons do return their result values to the stub. The stub will forward them to a user-defined handler object, where the original invoker can retrieve the results later on. This mechanism can be used to implement futures (where the result value is stored until it is explicitly retrieved), voting (where all results are collected and the most frequently occurring one is returned), selection (of one result) and combining (of several results).

The third result handling scheme, *return one result*, is the default way of handling results in unicast invocations like RMI. The stub forwards a method invocation and blocks until the skeleton returns a result. If multiple results are produced, the user selects one *in advance* by specifying its rank. This has the advantage that only a single skeleton has to return a result, avoiding communication overhead.

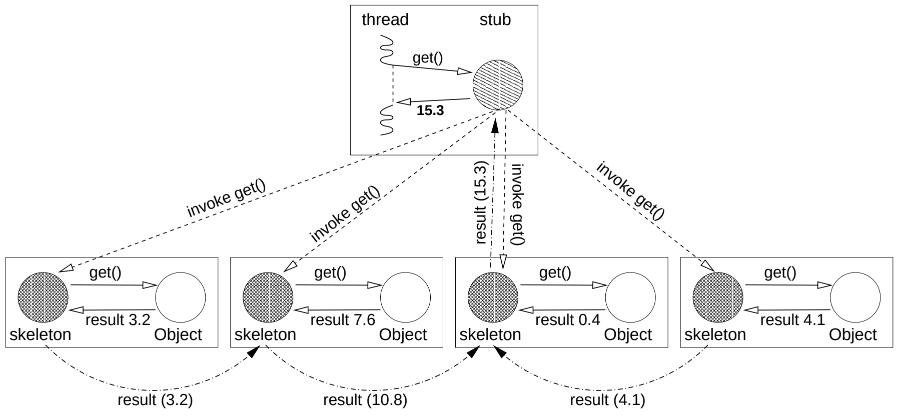

Fig. 3. An example of result combining

The final scheme, *combine results*, is useful when multiple results are produced. A user defined *combine* method is used to merge all results into a single value, which is returned. The results can either be combined by the stub (suitable for *gathering* results) or by the skeletons (suitable for *reducing* results). GMI selects the correct approach by analyzing the signature of the combine method.

Figure 3 shows an example of *reduce-style* result combining. A method invocation is multicast to all four objects in a group. After applying the method to their object, the skeletons communicate amongst each other to combine all results into a single value. The advantage of this scheme is that it can combine multiple results in parallel, for example by arranging the skeletons in a tree structure. This is important if the object group is large.

Any exception thrown during the execution of a combine operation will be forwarded instead of regular result values. The overall result of the combine operation will then be one of the thrown exceptions.

Combinations of the Forwarding and Reply-Handling Schemes. We have presented three different schemes to forward a method invocation and four schemes for handling the results. GMI allows these schemes to be combined orthogonally, as shown in Table 1. All twelve combinations result in useful communication patterns, making GMI highly expressive.

Table 1. Combinations of stub-skeleton behavior

	invocation		
result	single	group	personalized
discard	async. RMI	multicast	scatter
forward	future	async. voting / combine	scatter + async. combine
return one	RMI	synchronous multicast	synchronous scatter
combine	collective combine (allreduce/allgather)	group combine (reduce/gather)	scatter + group combine

In the table, *scatter, (all)gather,* and *(all)reduce* refer to the functionality of the respective collective operations from MPI. However, there is a major difference between the GMI communication and their MPI counterparts. In MPI, *all* members of a group must *collectively* invoke the communication operation.

Although the GMI model is different from the collective model used in MPI, it also provides communication similar to MPI-style allreduce and allgather. By using collective, single-object invocations and result combining (shown as *collective combine* in the table), all stubs of a group are required to forward their invocation to a different skeleton. Each skeleton will then produce a result and combine this result with all others. The overall result will finally be returned to all stubs.

GMI also allows a single thread to invoke a method on a group of objects without active involvement from other threads. For example, when combining a group invocation with result combining (shown as *group combine* in the table), a single thread forwards the invocation to the entire group and combines all results. In Sect. 2.3 we will show an example application that uses this style of communication.

Synchronization and Ordering of Method Invocation. In RMI, no guarantees are given about the order in which method invocations are received. For example, if two stubs simultaneously forward a method to the skeleton, the order in which the skeleton receives (or handles) these invocations is not defined. It may even run the two invocations concurrently, depending on the behavior of the

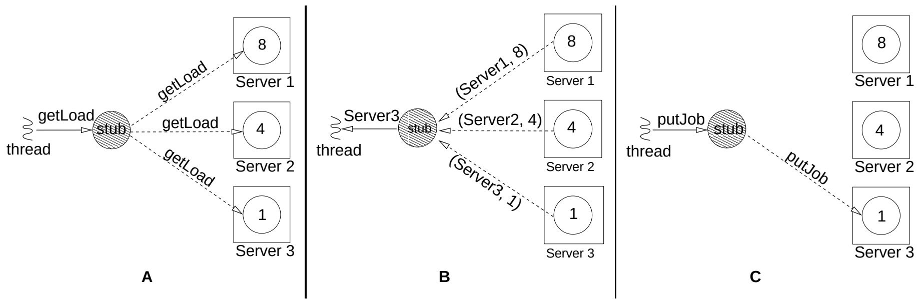

Fig. 4. Load balancing using GMI

network and the implementation of the skeleton. The programmer can use the regular Java monitor mechanism (*synchronized/wait/notify*) to ensure that the object behaves correctly, independent of the order in which method invocations are received.

Currently, the GMI model does not guarantee any ordering of group invocations. When two group invocations are done simultaneously on the same group, they may be received in different orders by different group objects. Semantically, a group invocation is equivalent to doing multiple RMI calls, and synchronization problems have to be addressed by the programmer in the same way as with RMI. The advantage of using group invocations, however, is that all the details of message forwarding and result processing are handled efficiently by the GMI implementation. For applications where group objects are merely intended as replicas of a single object, it is preferable to use the RepMI programming model [11]. Our native implementation of RepMI offers replicated objects that are kept consistent using totally-ordered multicast and a replication-aware thread scheduler. GMI was specifically designed for those applications where RepMI's consistency model is too strict and alternative communication forms (like personalized broadcast) are needed.

2.3 Example application

To illustrate how the stubs in GMI can be dynamically configured, we will now show an example application that uses GMI to implement a load balancing mechanism. This example also illustrates the difference between the GMI and MPI models.

In Fig. 4, a thread produces jobs for three servers, each containing an object which together form a group. These objects monitor the load of their server. The thread invokes the *getLoad* method on each of the objects using a group invocation (A). Each object executes this method and returns an estimate of the local load. Using a combine operation, the server with the least load is selected from these results (B). The job is then forwarded to this server (C). Note that

(A) and (B) together form a single group operation (i.e., a *group* invocation that *combines* the result).

This application is difficult to express using MPI-style collective communication, because it is not known in advance when the load information will be requested. Such asynchronous events are easily expressed with group invocations, but not with collective communication. The servers could either frequently poll for incoming *getLoad* messages, or periodically broadcast their latest load information, both of which would degrade performance.

```
import gmi.*;

interface ServerInterface extends GroupInterface {
  public Load getLoad();
  public void putJob(Job j);
}

class Client {
  public static void main(String [] args) {
    ServerInterface s = (ServerInterface) GMI.bind("ServerGroup");
    GroupMethod m = Group.findMethod(s, "getLoad()");
    m.groupInvoke();
    m.combineResult(new MyLoadCombiner());
    Load l = s.getLoad();
    m = Group.findMethod(s, "putJob(Job)");
    m.singleInvoke(l.rank);
    s.putJob(new Job(...));
  }
}
```

Fig. 5. A example GMI application

Figure 5 shows GMI pseudocode for the client side of the application. For simplicity, exception handling and the server code is ommitted from this example. The client starts by creating a new stub using the *bind* operation, which is similar in functionality to the *bind* operation of RMI. This stub can be used to communicate with the group of server objects called *"ServerGroup"*. The client then finds the *getLoad* method in the stub and configures it to use a *group invocation* and *combine* all results. The result combining is done using a *MyLoadCombiner* object (code not shown), which contains a method that selects the best result. Next, the *getLoad* method is invoked on the stub. This invocation behaves as shown in Fig. 4, parts A and B. After the results have been returned, the client configures the *putJob* method to be forwarded to the server with the lowest load, and invokes it.

3 Performance Results

We now evaluate the performance of our GMI implementation using the Manta high-performance Java system [12]. Manta has been specifically designed for parallel programming in Java and contains a native Java compiler (i.e., it compiles Java source code directly to native executables), an efficient runtime system, and a highly optimized implementation of RMI. The performance measurements were done on the DAS system, a cluster of 200 MHz Pentium Pro processors with 128 MByte of main memory. All boards are connected by a 1.2 Gbit/sec Myrinet network. The system runs RedHat Linux 6.2. We will analyze the performance of the basic group operations using micro benchmarks. Then, using several applications, we will show that the performance of GMI is similar to that of mpiJava (a Java language binding to a native MPI library [6]).

3.1 Micro benchmarks

Table 2 shows the latencies (completion times) of the basic group operations described in Sect. 2. Each measurement was performed by doing 10,000 operations, followed by a barrier synchronization.

Table 2. Group operations (time in μs)

group size	group invocation	collective combine	group combine	personalized invocation
1	26	13	29	37
2	54	95	102	53
4	56	135	136	72
8	55	177	170	108
16	57	227	205	182
32	59	274	252	343
64	71	333	300	642

In general, the operations have a low latency, which is explained by the efficient low-level Myrinet communication and by the many optimizations performed by the Manta system [12]. In comparison, Manta RMI has a round trip latency of about 40 μs on the same hardware. The latency for the group invocation increases only slightly with the group size. The reason is that this operation can be executed in a pipelined fashion, since the sender can continue as soon as a message has been delivered locally. However, on 64 processors (each running one executable), the flow control scheme of the underlying Myrinet multicast protocol sometimes has to block the sender, because the receivers run out of resources. This explains the increase in latency on 64 processors. The table also shows the latencies for two forms of result combining, collective and group, with roughly similar latencies. The high latencies for personalized group sends are

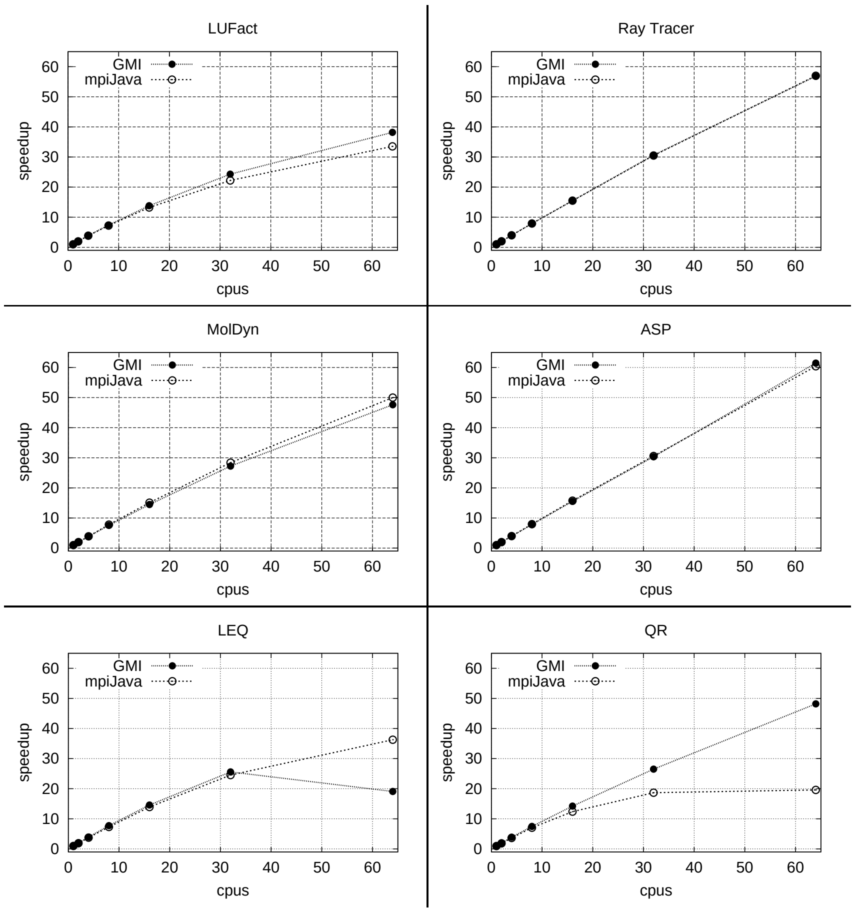

Fig. 6. Speedups of benchmarks

due to our current implementation, which is not yet optimal and serially sends the messages to the receivers.

3.2 Applications

Figure 6 shows the speedups achieved with the applications, relative to the fastest version on a single machine. The first three applications are taken from the Java Grande benchmark suite [5]. These applications come in different problem sizes, but we only show results for the largest problem size.

The **LUFact** kernel performs a parallel LU factorization, followed by a sequential triangular solver. The machines communicate by broadcasting integers and arrays of doubles. In the GMI version, these two broadcasts are expressed by forwarding a single *put* method to a group of objects. In mpiJava, the integer is stored in a spare entry of a double array to prevent an extra broadcast. The GMI version has a slightly better speedup (38) than the mpiJava version (34).

Ray Tracer renders a scene of 64 spheres. Each machine renders part of the scene which is simultaneously generated on all nodes. Each node calculates a checksum over its part of the scene. A reduce operation is used to combine these checksums into a single value. The machines send the rendered pixels to machine 0 using individual messages. The speedups achieved by mpiJava and by GMI are almost identical.

MolDyn is an N-body code, modeling argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. For each iteration, the mpiJava version uses six allreduce (summation) operations to update the atoms. In GMI, each machine stores its data in a single object. These objects are then combined using a collective combine operation. The speedup of MolDyn for GMI and mpiJava are almost identical, 47 for GMI and 50 for mpiJava (on 64 machines). The mpiJava version is faster because of its highly optimized summation operations for (arrays of) primitive types. Unlike GMI, mpiJava implements these operations completely in the library, and does not require the use of serialization to communicate, nor does it invoke user-defined methods to perform the operations. GMI uses a more general approach, which results in a slightly lower speedup.

ASP (All-pairs Shortest Paths) finds the shortest path between any pair of nodes in a graph, using a parallel version of Floyd's algorithm. The program uses a 2000×2000 distance matrix that is partitioned row-wise among the available processors. At the beginning of iteration k, the processor containing this row must broadcast it to the others. To implement this, mpiJava uses a broadcast operation, while GMI multicasts the invocation of a *put* method.

Both versions obtain excellent speedups, because the amount of communication needed for each iteration (broadcast a row) is small compared to the computation time (updating multiple rows of the distance matrix).

LEQ (Linear Equation Solver) is an iterative solver for linear systems of the form $Ax = b$. The program partitions a dense 1000×1000 matrix containing the equation coefficients over the processors. In every iteration, each processor produces a part of the candidate solution vector x_{i+1}, but needs all of vector x_i from the previous iteration as its input. Therefore, all processors need to combine their partial solution vectors at the end of each iteration. The processors also decide if another iteration is necessary, by calculating the difference between each element of x_{i+1} and x_i and computing the sum of these differences using a combine operation.

Both versions have similar speedups up to 32 processors. Unfortunately, the GMI version does not scale to 64 processors. This is caused by the algorithm used in the combine operation. The *mpiJava* library uses a ring algorithm, whereas our system currently uses a binomial tree.

QR is a parallel implementation of QR factorization. The program is parallelized by partitioning the 2000x2000 matrix which must be factorized over all processors. In each iteration, the column with the maximum norm is selected from the entire matrix, which will serve as the *Householder vector H*. Each processor selects the most suitable column from its local partition. From these columns the one with the maximum norm is selected (using a combine operation) as H and is broadcast to all processors.

GMI obtains much better speedups for QR than mpiJava (48 compared to 20, on 64 processors). mpiJava suffers from a high serialization overhead. This is caused by the allreduce operation that uses an object as data instead of an array. A case for which mpiJava is not optimized. The GMI implementation, however, is optimized to handle objects. It uses the highly-efficient serialization code generated by the Manta compiler.

4 Related Work

Java is increasingly recognized as a suitable platform for high-performance computing. The driving force is the Java Grande Forum (www.javagrande.org). Many research projects investigate parallel programming in Java [3,4,8,16]. Efficient communication mechanisms are a vital building block for high-performance Java [10]. One approach to sharing objects between parallel processes is to use either *object replication* [11] or *object caching* [1,18]. However, neither replication nor caching of objects provides the flexibility and expressiveness of GMI.

An alternative is to use a communication mechanism outside Java's object model. MPJ [6] proposes MPI language bindings to Java. MPI supports a rich set of communication styles, in particular collective communication. Unfortunately, MPI does not integrate cleanly into the Java object model. While communication between Java objects (even in sequential programs) is expressed using method invocations, MPJ is based on message-passing. MPJ's communication primitives are primarily designed for transmitting arrays of primitive data types (e.g., doubles), not for handling the complex object data structures often used in Java (e.g., lists and graphs). Also, MPJ uses a SPMD programming style, while Java is more suitable for a multi-threaded (MPMD) programming style.

In the field of distributed systems, group communication mainly serves the purposes of fault tolerance and location transparency [2,9,20]. Efficiency is less important in this context.

The Common Object Request Broker Architecture (CORBA) [13] is similar in functionality to RMI. In CORBA, *interceptor* functions can be inserted into a limited number of hooks in the runtime system. Interceptors allow the standard method invocation mechanism to be modified. They are applied to all methods that pass through that interception point. As a result, interceptors must be very general and able to handle any method invocation, limiting their usefulness for implementing complex group operations. However, the functionality of GMI heavily depends on an interceptor-like scheme, using function objects to modify the behavior of invoked methods. The difference is that GMI uses modification functions that are specific to a single method of a single stub, making them more flexible and easier to implement.

Smart proxies [19] change the behavior of an application by extending the stubs generated by the IDL compiler. Unfortunately, implementing smart proxies is quite complex, because it is up to the programmer to implement all communication code. With GMI, the stub is completely compiler generated (including all communication code). The programmer only needs to configure the stub at runtime, using function objects if necessary.

JavaSpaces [7] provides communication using Linda-like shared data *spaces*. However, no group operations can be applied to these *spaces*. JavaNOW [17] implements some of MPI-style collective operations on top of an *entity space*; however, performance is not an issue. Taco [15] is a C++ template library that implements some collective operations. Unlike these systems, GMI directly augments Java's object model by an orthogonal set of method invocation mechanisms. It provides maximal programming flexibility with efficient group communication.

5 Conclusions

We introduced a new model for group communication that generalizes RMI (Remote Method Invocation) by adding different schemes for forwarding invocations and for handling results. By combining these schemes orthogonally, a rich variety of useful communication primitives arises, which integrates seamlessly into object-oriented languages. The resulting model, Group Method Invocation (GMI), is highly expressive, easy-to-use, and efficient.

GMI's expressiveness is due to its orthogonal design. GMI supports invoking a method on single or multiple objects, personalized group invocations, and various ways of gathering or combining data from different objects. GMI supports RMI-style group operations that are invoked from a single thread, but can also be used to implement MPI-style collective operations, where all threads collectively call the same operation. We believe that the first model fits better into Java's multi-threaded programming model than MPI's collective communication, which was designed for SPMD-style parallelism. Many existing communication primitives (e.g., remote method invocations, asynchronous RMI, futures, and voting) appear naturally in GMI's design matrix, as opposed to being introduced as special cases.

Finally, GMI was designed for high performance, in particular to exploit efficient low-level multicast primitives provided by the network. Using microbenchmarks and several applications we have shown that the performance of GMI is similar to that of a Java language binding to a native MPI library (mpiJava).

References

1. G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.
2. A. Black and M. Immel. Encapsulating Plurality. In *ECOOP'93*, number 707 in Lecture Notes in Computer Science, pages 57–79. Springer, 1993.

3. 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.
4. 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.
5. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12:375–388, 2000.
6. 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.
7. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns, and Practice*. Addison Wesley, 1999.
8. 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.
9. R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System Support for Object Groups. In *Proc. OOPSLA'98*, pages 244–258, Vancouver, B.C., Oct. 1998.
10. T. Kielmann, P. Hatcher, L. Bougé, and H. E. Bal. Enabling Java for High-Performance Computing. *Commun. ACM*, 44(10):110–117, 2001.
11. 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.
12. J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. Prog. Lang. Syst.*, 2001.
13. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. June 1999. rev. 2.3.
14. A. Nelisse, T. Kielmann, H. E. Bal, and J. Maassen. Object-based Collective Communication in Java. In *Joint ACM JavaGrande-ISCOPE 2001*, pages 11–20, 2001.
15. J. Nolte, M. Sato, and Y. Ishikawa. Template Based Structured Collections. In *Int. Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 483–491, Cancun, Mexico, 2000.
16. M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.
17. 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). *Concurrency: Practice and Experience*, 12:1093–1116, 2000.
18. R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Joint ACM JavaGrande-ISCOPE 2001*, pages 153–162, 2001.
19. N. Wang, K. Parameswaran, and D. C. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio, 2001.
20. A. F. Zorzo and R. J. Stroud. A Distributed Object-Oriented Framework for Dependable Multiparty Interactions. In *OOPSLA'99*, pages 435–446, Denver, CO, Nov. 1999.