4 Object Serialization: A Case for Specialization

The ability to send and receive primitive-type arrays efficiently is essential for high-performance communication in Java. Jbufs enable the VI architecture to transfer the contents of arrays in a zero-copy fashion by exploiting two facts: primitive-type arrays are shallow objects (i.e. contain no pointers to other Java objects) and their elements are typically contiguous in memory[20]. Array-based cluster applications written in Java can take advantage of jbufs to improve their communication performance, as demonstrated in the previous chapter.

During a remote method invocation (RMI) in Java, however, arbitrary linked object data structures are frequently passed by copy and must be transmitted over the wire. Since the objects forming the data structure are not guaranteed to be contiguous in memory, they need to be serialized onto the wire on the sending side and de-serialized from the wire on the receiving side. Standard serialization protocols are designed first for flexibility, portability, and interoperability of the RMI layer and only second for performance.

---

[20] Virtually all JVM implementations lay out array elements contiguously in memory for efficiency purposes, although it is not guaranteed by the JVM specification.

This chapter argues that the costs of object serialization are prohibitively high for cluster applications. It evaluates several implementations of the JDK serialization protocol using micro-benchmarks and an RMI implementation over Javia-I/II. Although efficient array transfer improves the performance of RMI (because serialization of Java objects ultimately yields byte arrays) the overheads of serialization are still an order of magnitude higher than the basic send and receive overheads in Javia-II. The impact of serialization on point-to-point RMI performance is substantial: the zero-copy benefits achieved by jbufs become negligible. For some applications in an RMI benchmark suite, the cost of serialization is estimated to account for up to 15% of their total execution time.

4.1 Object Serialization

As seen in Figure 4.1, serializing an object consists of converting its in-memory representation into a stream of bytes. This conversion makes a deep copy of the object: all transitively reachable objects are also serialized. The resulting

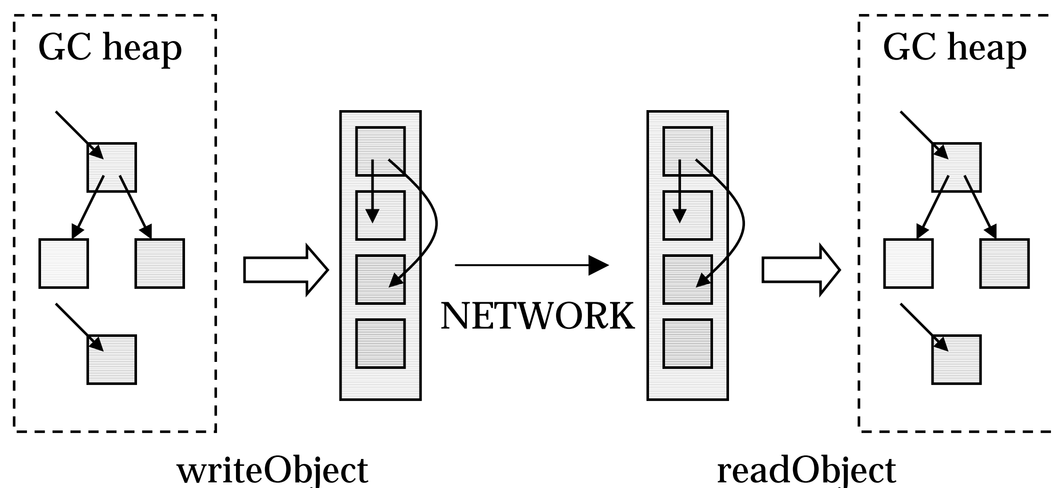

Figure 4.1 Object Serialization and De-serialization.

stream, typically stored in a Java byte array, is sent over the network. At the receiving end, the objects are retrieved (de-serialized) from the stream: for each de-serialized object, data is copied from the stream into its newly allocated storage.

Most publicly available JVMs implement the Java Object Serialization (JOS) protocol [Jos99] that is designed for flexibility and extensibility. JOS introduces object I/O streams (`ObjectInputStream/ObjectOutputStream`) with methods to write "serializable" Java objects into the stream (`writeObject`) and to read them from the stream (`readObject`). The protocol serializes the description of an object's class along with the object itself. If the class is available in the JVM during de-serialization, both the wire and the local versions are compared using "class compatibility" rules. If the class is not available or is incompatible, JOS provides a mechanism to annotate serialized classes (via the `annotateClass` method) so users can send along the original byte-code or an URL from where it can be fetched. Users can also define the external format of an object by overriding `read/writeObject` methods in object I/O stream classes with protocol-specific ones, or by providing object-specific implementations of `read/writeExternal` methods.

4.1.1 Performance

This section shows that the performance of JOS is inadequate for cluster computing using results from micro-benchmarks. Figures 4.2 and 4.3 show the performance of three implementations—Marmot, JDK1.2, and Jview3167 on a 450Mhz Pentium-II—of `writeObject` and `readObject` methods respectively. The types of objects used in the experiment are `byte` and `double` arrays with comparable sizes (around 100 and 500 bytes), an array of `Complex`

numbers (each element with a real and a imaginary field of type `double`), and a linked list (each element with a `int` value and a "next" pointer). The costs for the latter two are reported on a per-element basis. The numbers reported are averages with standard deviation of less than 5% (maximum is 4.7% in *byte[] 500*).

The results shown in Figures 4.2 and 4.3 lead to the following observations:

1. Serialization overheads are in tens of microseconds: an order of magnitude higher than basic send and receive overheads in Javia I/II (around 3μs). Reading a byte array of 500 elements costs around 20μs; in comparison, a `mempcy` of 500 bytes costs around 0.8μs;
2. Serialization of arrays with 16, 32, and 64-bit primitive-type elements

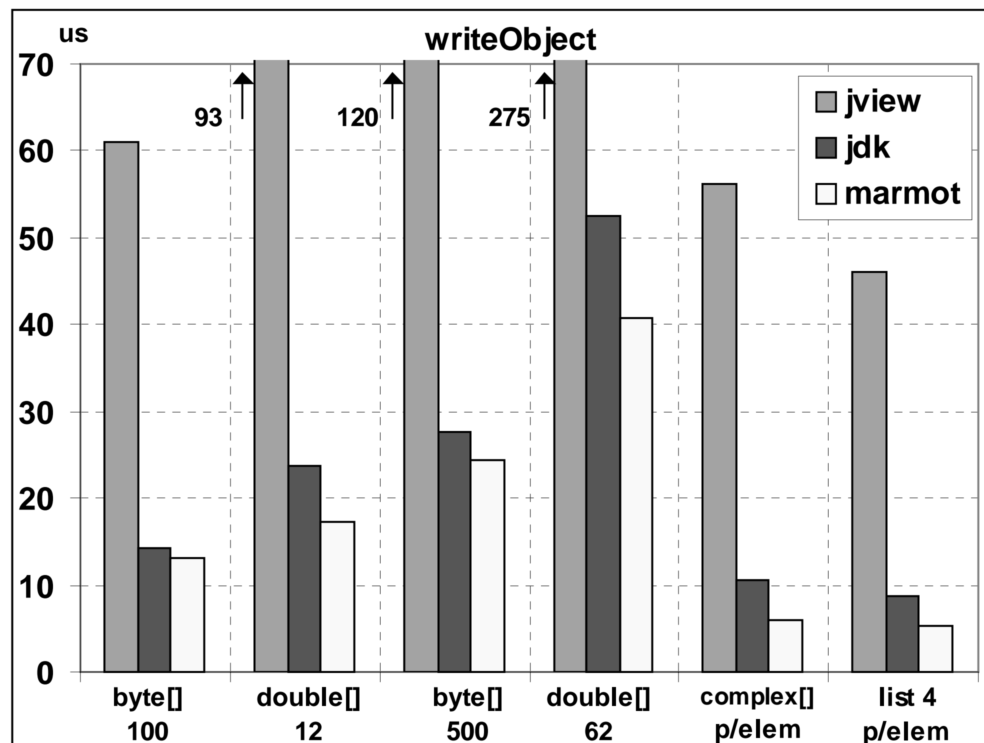

Figure 4.2 Comparing the cost of serialization in three implementations of Java Object Serialization.

into byte arrays takes a significant performance hit due to Java's type safety. Serializing a `double` array of 62 elements is nearly 50% more expensive than a byte array of 500 elements;

3. Costs grow as a function of object size both in `writeObject`, due to the deep-copy, and in `readObject`, due to storage allocation and data copying. It costs about 9µs to read one linked-list element with an `int` field out of the stream, and around 86µs to read one with 40 `int` fields.

It seems unlikely that better compilation technology will improve the performance of serialization in a substantial way. Tables 4.1 and 4.2 show the percentage change in the cost of `writeObject` and `readObject` on Marmot in the absence of method inlining, synchronization, and safety checks. Changes in cost are more significant for the array of complex numbers and the linked

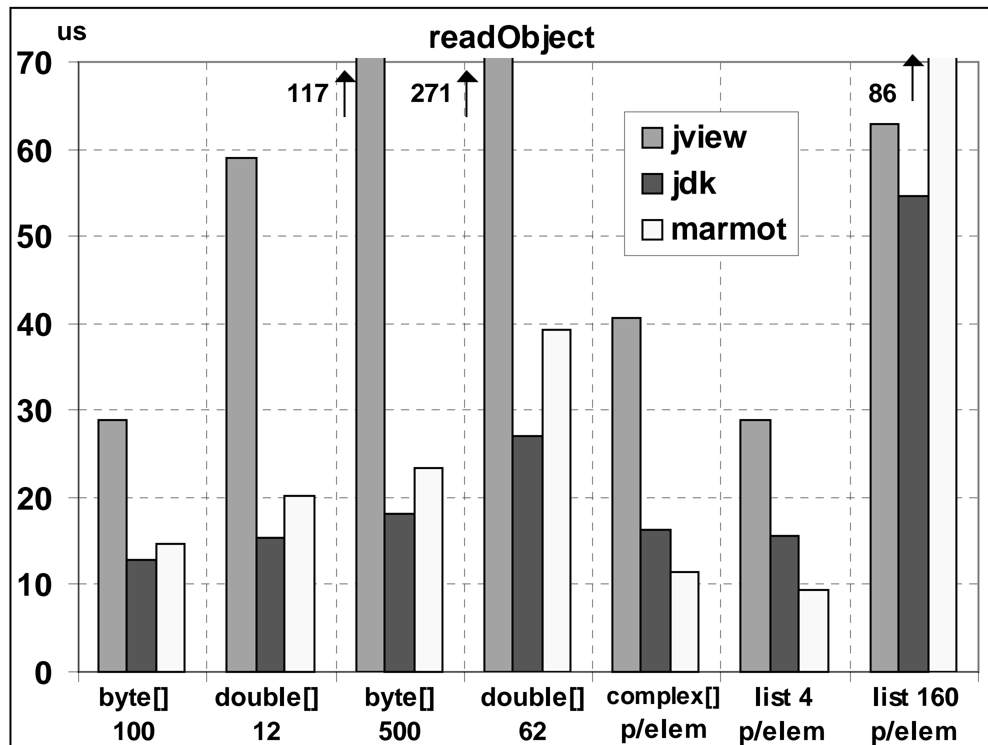

Figure 4.3 Comparing the cost of de-serialization in three implementations of Java Object Serialization.

list. Method inlining in Marmot already reduces the costs by 60%. Even if Marmot were able to successfully eliminate *all* safety checks and *all* synchronization, performance would improve by another 30% at best.

Table 4.1 Impact of Marmot's optimizations in serialization.

Cost Difference (writeObject)	*no method inlining*	*no locks*	*no array-bounds checks*	*no null pointer checks*	*no casts checks*	*no array-store checks*
byte[] 500	24.2%	-4.5%	-0.4%	0.0%	-2.5%	-1.3%
double[] 100	14.1%	-3.2%	-1.5%	0.0%	-3.3%	-0.5%
complex[] p/elem	56.7%	-12.9%	-0.7%	0.0%	-0.6%	-7.3%
list p/elem	61.7%	-12.9%	-0.7%	0.0%	0.0%	-6.7%

Table 4.2 Impact of Marmot's optimizations in de-serialization

Cost Difference (readObject)	*no method inlining*	*no locks*	*no array-bounds checks*	*no null pointer checks*	*no casts checks*	*no array-store checks*
byte[] 500	48.7%	-4.2%	0.0%	0.0%	-0.9%	0.0%
double[] 100	23.2%	-2.5%	0.0%	0.0%	-1.5%	0.0%
complex[] p/elem	80.5%	-12.6%	0.0%	0.0%	-6.1%	0.0%
list p/elem	77.7%	-20.7%	0.0%	-2.9%	-11.2%	-0.7%

4.2 Impact of Serialization on RMI

The high serialization costs reported in the previous section affect the performance of Java RMI significantly. This section starts with an overview of Java RMI and briefly describes an implementation over Javia-I/II. Readers familiar with RMI can jump to the section on micro-benchmark performance (Section 4.2.3).

4.2.1 Overview of RMI

RMI enables the creation of distributed Java applications in which methods of remote Java objects can be invoked from other JVMs, possibly on different hosts. A Java program can make a call on a remote object once it obtains a ref-

erence to the remote object, either by looking up the remote object in a name service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI implementations in publicly available JVMs are based on the Java RMI specification [Rmi99].

RMI relies on JOS to serialize and de-serialize remote objects (which are passed by reference) and regular objects (which are passed by value). RMI takes advantage of JOS' extensibility and class serialization protocol to support "polymorphic"[21] method invocations: an actual parameter object can be a subclass of the remote method's formal parameter class. This means that the receiver may not know the actual subclass of the argument, and may have to fetch it from the wire or from a remote location. This flexibility makes RMI applications potentially more tolerant to service upgrades and different versions of class files, and is the key distinction between RMI and traditional remote procedure call systems.

4.2.2 An Implementation over Javia-I/II

This section describes a straightforward RMI implementation over Javia-I/II based on the RMI specification.

Remote objects (that extend the `RemoteObject` class and implement the `Remote` interface) are bound (i.e. exported) to a simple RMI `Registry`. The registry creates corresponding stub and skeleton (i.e. server side stub) objects for the remote object, spawns a transport-dependent server thread that waits for incoming connections, and updates its service database. When a client binds to an exported remote object, the registry ships the stub to the client,

---

[21] This term is in quotes because there is no true polymorphism in Java [OW97].

which is instantiated in the client's JVM. During an RMI, the stub creates a transport-dependent `RemoteCall` object that connects to the server thread and initializes communication structures. The server thread spawns a new thread to service calls from that stub upon accepting the connection. The remote call object is cached by the stub for subsequent invocations to the same remote object in order to avoid creating a new connection for every RMI.

The implementation uses JOS for serialization and de-serialization of arguments and relies on a RMI protocol that is similar to the one described in the specification. It also uses a simple distributed GC scheme based on reference counting [BEN+94].

The system consists of about 4000 lines of Java and currently supports three transport layers: TCP/IP sockets, Javia-I and Javia-II. A remote call object using Javia-I connects to the server thread through a virtual interface that can be configured in four different send/receive combinations (Section 2.3.1). In the case of Javia-II, a connection is composed of two virtual interfaces: one for RMI headers (up to 40 bytes) and another for the payload. Jbufs posted on the header VI are accessed as `int` arrays; those posted on the payload VI are accessed as `byte` arrays by the object I/O streams.

Because of RMI's blocking semantics, the number of jbufs posted on each VI (which is a service parameter) essentially indicates the maximum number of concurrent calls (e.g. client threads) the remote object can handle for each connection. A remote call object tracks the number of outstanding RMIs to ensure that that number is not exceeded. When waiting for an incoming message (either a call or a reply), the thread polls for a while (around twice the round trip latency) before blocking.

4.2.3 Performance

The round-trip latency of an RMI between two cluster nodes is measured by a simple ping-pong benchmark that sends back and forth a byte array of size *N* as argument using a single RMI connection. The effective bandwidth is measured by sending 10MBytes of data one-way, using various byte array sizes as fast as possible. RMI implementations over several configurations of Javia-I and over Javia-II (labeled as *RMI jbufs*) are compared on Marmot and JDK1.2. Both experiments exclude context switch costs since unloaded machines are used and reception takes place primarily by polling (due to the above optimization).

Figure 4.4 shows the round-trip latencies. Although jbufs yield some improvement, the RMI performance is far from that of Javia-II. Table 4.3 shows the round-trip latencies of an RMI with an integer argument and includes the number for RMI over sockets as well. A significant fraction of the 150µs achieved by *RMI jbufs* goes to setting up object I/O streams for argument passing. For instance, the round-trip latency of a null RMI drops to slightly less than 30µs, which is about the same as that achieved by Jam.

Figure 4.5 shows effective bandwidth achieved by RMI. A peak bandwidth of about 22MBytes/s is attained by *RMI jbufs*, which is about 25% of total capacity. The bandwidth curve reaches the peak at a much slower rate than Javia-I and Javia-II because RMIs are not pipelined (due to their blocking semantics) during the experiment[22].

---

[22] Bandwidth experiments involving multiple client connections have not been carried out because the cluster nodes have a single processor.

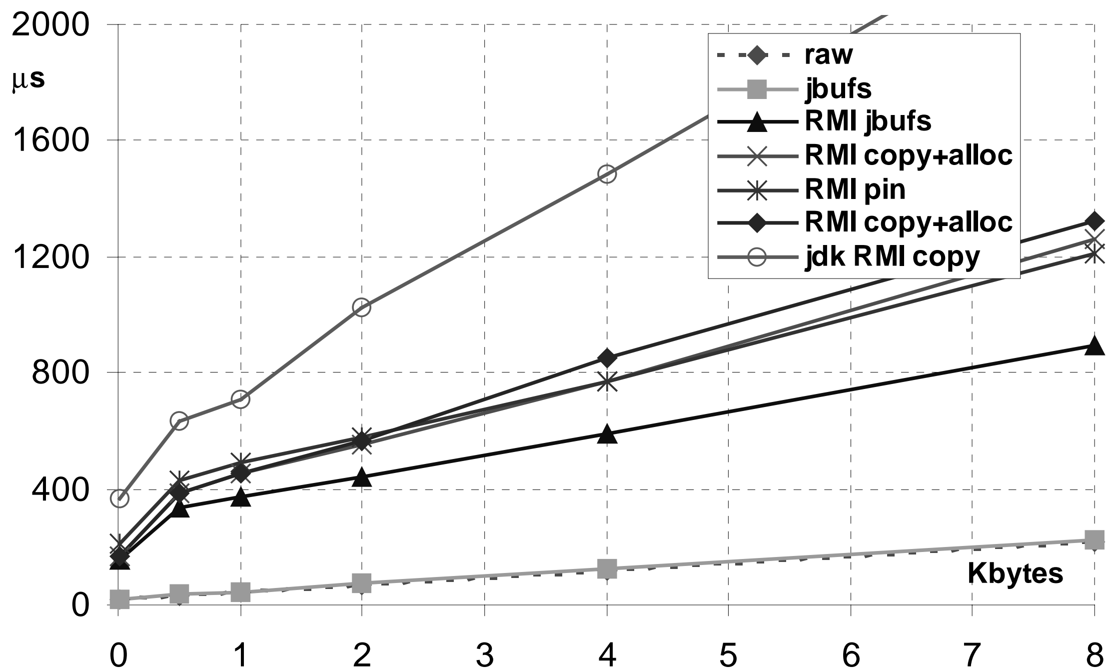

Figure 4.4 RMI round-trip latencies.

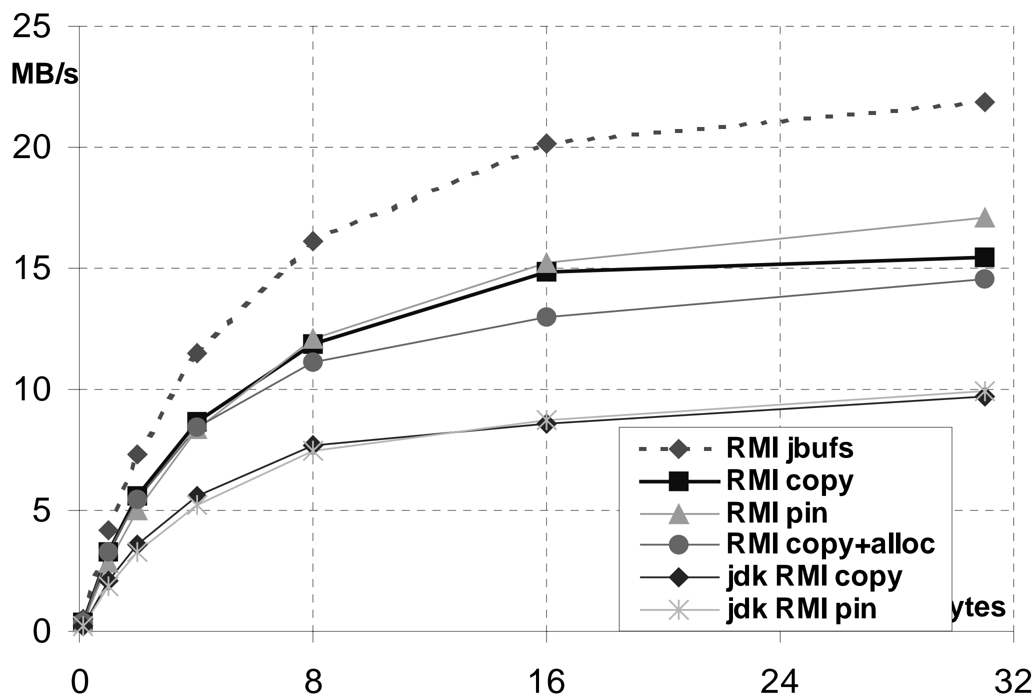

Figure 4.5 RMI effective bandwidth.

Table 4.3 RMI 4-byte round-trip latencies.

RMI	*4-byte (us)*
jbufs	150.4
copy+alloc	161.9
copy	164.5
pin	211.8
jdk copy	271.0
sockets	482.3
jdk sockets	520.1

4.3 Impact of Serialization on Applications

Poor RMI performance can have a significant effect on overall application performance. This section reports the impact of object serialization on a benchmark suite consisting of six RMI-based applications. Table 4.4 provides a summary of the applications used. A brief description of each application is presented in the following subsection.

4.3.1 RMI Benchmark Suite

Two applications, Traveling Salesman Problem (*TSP*) and Iterative Deepening A* (*IDA*), fall into the traditional producer-consumer model [NMB+99]. TSP computes the shortest path to visit all cities exactly once from a starting city by using a "branch-and-bound" algorithm. The algorithm prunes search sub-spaces by ignoring partial routes that are longer than the current shortest path. Because the amount of computation for a search sub-space is not known a-priori, the implementation adopts the producer-consumer model for (potentially) better load balancing. Workers running on cluster nodes repeatedly fetch jobs from a centralized job queue using RMIs. During execution, each worker keeps a local copy of the current best solution—if a worker finds a

Table 4.4 Summary of RMI benchmark suite.

Application	*Origin*	*Description*	*Model*	*Input*
TSP	Manta	shortest path to visit all other cities exactly once	Work Queue	17 cities
IDA	Manta	solving a 15-tile puzzle using repeated DFS	Work Stealing	depth of 58 moves
SOR	Manta	iterative method for Laplace equations	Sync Master Slave	1600-1600 grid, 100 iterations
EM3D arrays	Split-C Suite	simulation of EM wave propagation using RMI of `double[]`	Sync Master Slave	100K edges/proc, 100 iterations
FFT complex	Split-C Suite	1-D Fast Fourier Transform using `Complex[]`	Sync Master Slave	1 million points
FFT arrays	Split-C Suite	1-D Fast Fourier Transform using `double[]`	Sync Master Slave	1 million points
MM	Javia	MM using RMIs	Sync Master Slave	256x256 matrices

shorter solution, it updates the values of all other workers through RMI. The computation terminates when there are no jobs left in the job queue. The size of the input set used is 17 cities.

IDA solves the 15-tile puzzle using repeated depth-first searches. The program uses a decentralized job queue model with work stealing. Each job corresponds to a state in the search space, and each cluster node maintains a local job queue. When a node fetches a job from its job queue, it first checks whether the job can be pruned. If not, it expands the job by computing the successor states (e.g. making all possible next “moves”) and enqueues the new jobs. If the local job queue becomes empty, a node tries to “steal” jobs from

other nodes. The initial state of the puzzle is obtained by making 58 moves from the final state.

The remaining applications fall into the “structured” category: processing nodes have distinct computation and communication phases and are globally synchronized using barriers (*Barrier*). Upon reaching barrier point, program execution is blocked until all nodes reach a corresponding barrier point. To reduce the network traffic, each node communicate only with a parent node (if it is not the root) and (up to) two children nodes through RMIs.

Red-black Successive Over-relaxation (*SOR*) [NMB+99] is an iterative method for solving discrete Laplace equations: it performs multiple passes over a rectangular grid, updating each grid point using a stencil operation (a function of its four neighbors). The grid is distributed across all nodes in a row-wise fashion so each node receives several contiguous rows of the grid. Due to the stencil operation and the row-wise distribution, at each iteration every node (except for the first and last) needs to exchange its boundaries rows with its left and right neighbors using RMIs before updating the points. Each iteration has two exchange phases and two computation phases. The input used is a 1600x1600 grid of `double` values.

EM3D is a parallel application that simulates electromagnetic wave propagation [CDG+93]. The main data structure is a distributed graph. Half of its nodes represent values of an electric field (E) at selected points in space, and the other corresponds to values of the magnetic field (H). The graph is bipartite: no two nodes of the same type (e.g. E or H) are adjacent. Each of the processors has the same number of nodes, and each node has the same number of neighbors. Computation consists of a sequence of identical steps: each processor updates values of its local H- and E-nodes as a weighed sum of their

neighbors. A naïve version of EM3D performs an RMI to fetch the value from a remote node each time the value is needed. An optimized version uses a simple pre-fetching scheme: a ghost-node is introduced for each a remote node that is shared by many local nodes. During the pre-fetching phase, each ghost node fetches the data from its corresponding remote node, eliminating redundant RMIs. There are no remote accesses during the computation phase.

The version of EM3D used here aggregates ghost nodes on a per-processor basis and issues a single RMI per processor. It uses a `double` array as argument and explicitly copies data between the graph and the array itself. The benchmark uses a synthetic graph of 40,000 nodes distributed across 8 processors where each node has degree 20 for a total of 800,000 edges. The fraction of edges that cross processor boundaries is varied from 0% to 50% in order to change the computation to communication ratio.

Fast Fourier Transform (*FFT*) [CcvE99] computes the n-input butterfly algorithm for the discrete one-dimensional FFT problem using P processors. The algorithm is divided into three phases: (i) $log(n) - log(P)$ local FFT computation steps using a cyclic layout where the first row of the butterfly is assigned to processor 1, the second to processor 2, and so on; (ii) a data re-mapping phase towards a blocked layout where the n/P rows are placed on the first processor, the next n/P rows on the second processor, and so on; and (iii) $log(P)$ local FFT computation steps using the blocked layout. In the first and third phases, each processor is responsible for transforming n/P elements. Each processor allocates a single n/P-element vector to represent its portion of the butterfly. Communication occurs only in the data re-mapping phase where each processor uses RMIs to send a n/P^2-element chunk of data to each remote

processor[23]. The communication is staggered to avoid hot spots at the destination. Two versions—one using an array of `Complex` (with two `double` fields) and another using two `double` arrays—are run with an input of one million points.

The last application is a version of pMM (Section 3.3) where the communication using Javia-I/II is replaced with RMIs. pMM is run using 256x256 matrices on 8 processors.

4.3.2 Performance

RMI performance has little impact, if any, on the two irregular applications. Figure 4.6 shows the speedups of TSP and IDA. In TSP, the load is fairly balanced though coarse-grained; in IDA, idle workers ping other workers in a tight loop trying to steal work, congesting the network with RMIs.

The benefits of a fast transport layer are more pronounced in applications with distinct communication and computation phases. SOR (Figure 4.7) using RMI over Javia-II attains a speedup of 6.3 on 8 processors compared to about 1.8 if RMI over sockets are used. The per-edge cost of EM3D (Figure 4.8) grows as the percentage of remote edge grows, as expected. The array-based versions of FFT using RMI over Javia-I/II are able to achieve a peak 7Mflops (Figure 4.9). In comparison, the best FFT performance by a C program reported on a similar machine (a single 300Mhz Pentium-II) is about 70Mflops [Fft99].

---

[23] Because the FFT transfer size far exceeds the maximum transfer unit (MTU) of the RMI implementation (32Kbytes), data has to be further segmented to fit in MTU-sized chunks.

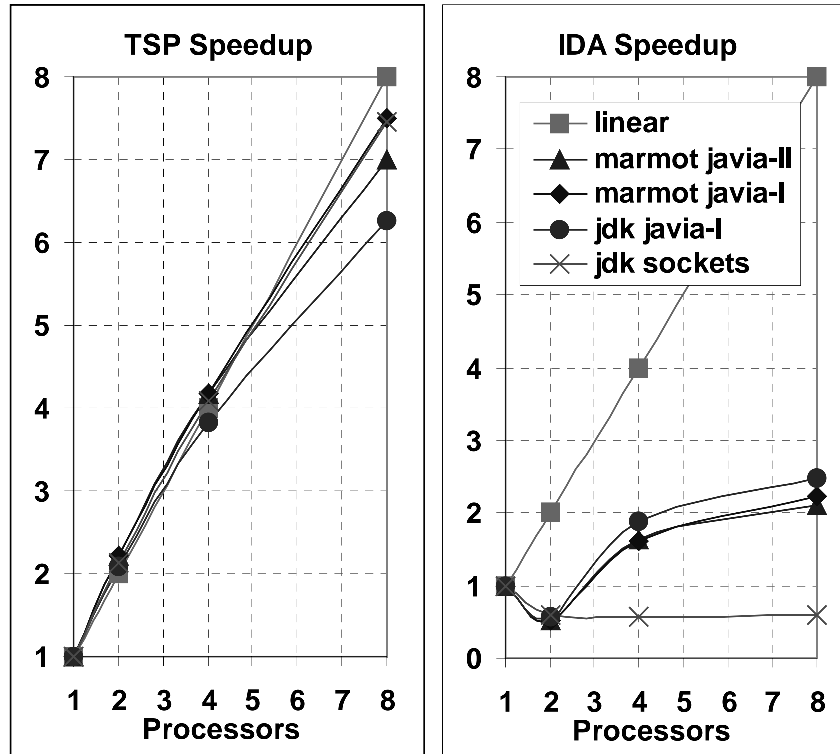

Figure 4.6 Speedups of TSP and IDA.

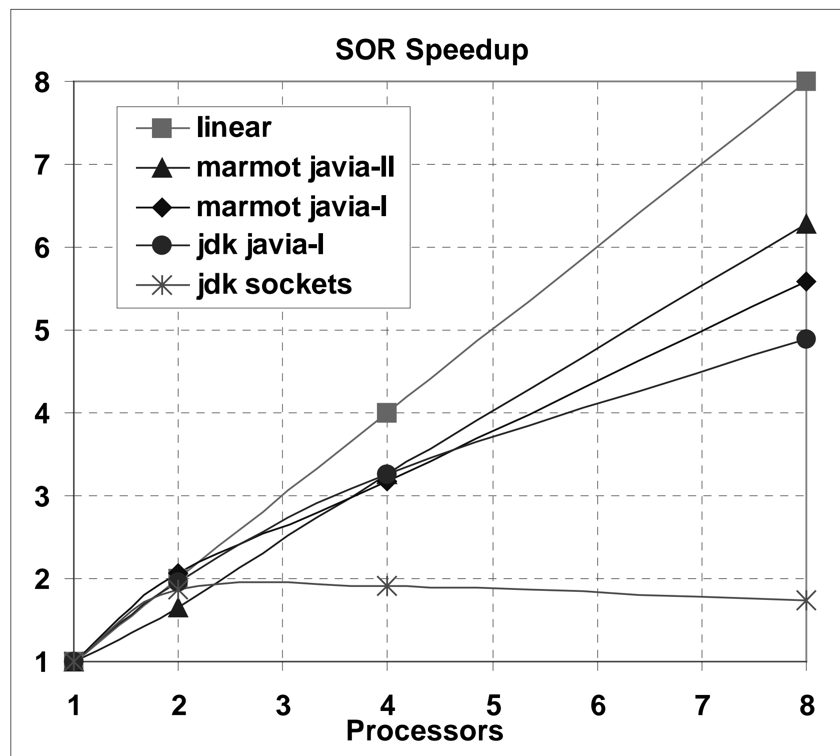

Figure 4.7 Speedup of SOR.

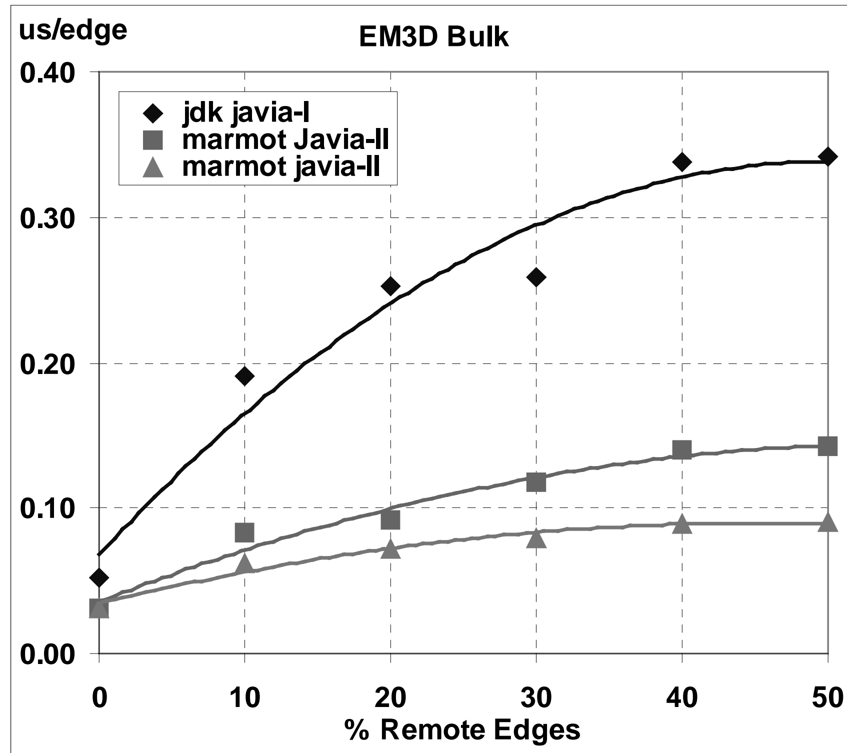

Figure 4.8 Performance of EM3D on 8 processors.

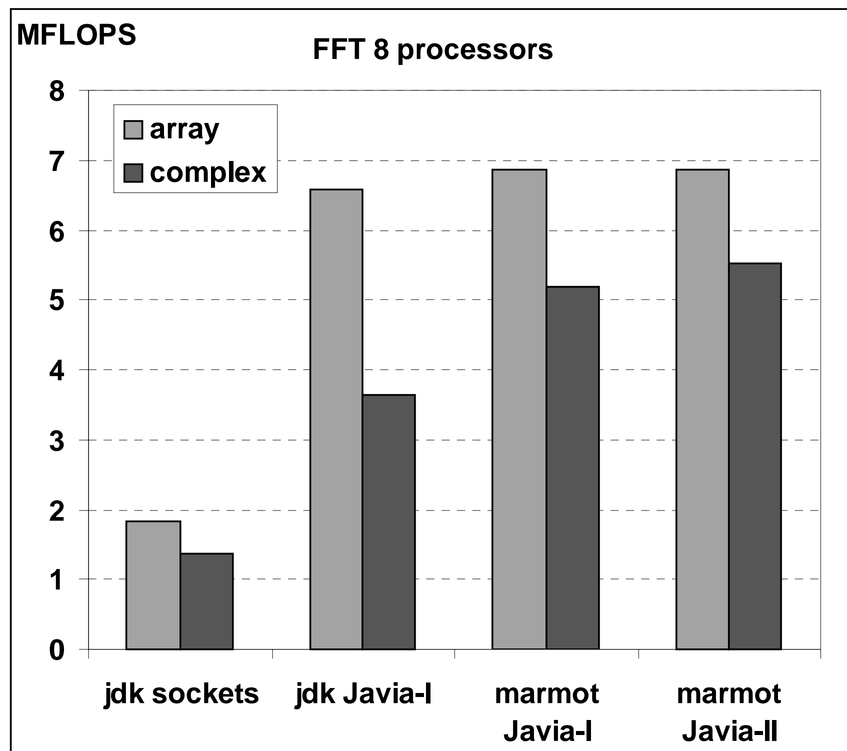

Figure 4.9 Performance of FFT on 8 processors.

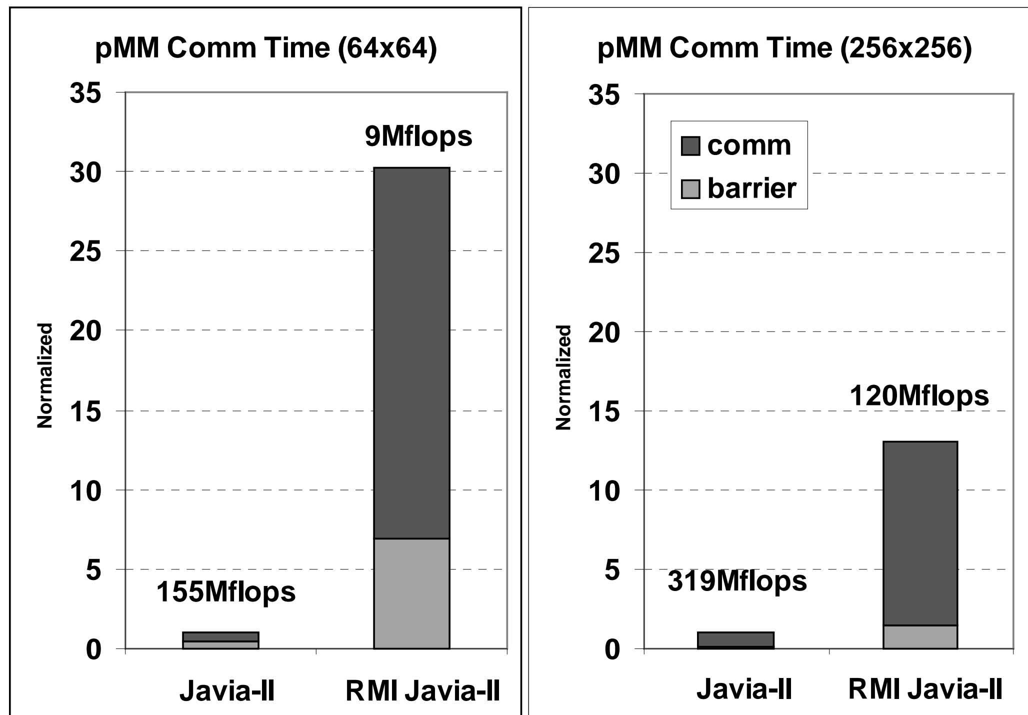

Figure 4.6 Performance of pMM over RMI on 8 processors.

Table 4.5 Communication Profile of Structured RMI Applications

Application	*#incoming RMIs*	*#outgoing RMIs*	*data type*	*data size (elements)*
SOR	400	400	`double[]`	1600
EM3D arrays	400	400	`double[]`	2300~2350
FFT complex	113	113	`Complex[]`	1024
FFT arrays	113	113	`double[]`	2 x 1024
pMM	22400	22400	`double[]`	256

Table 4.6 Estimated Impact of Serialization on Application Performance

Application	*comm meas. (secs)*	*total meas. (secs)*	*serial est. (secs)*	*serial est. (% comm)*	*serial est. (% total)*
SOR	4.59	19.78	0.54	11.76%	2.73%
EM3D arrays	2.20	4.60	0.24	10.90%	5.22%
FFT complex	18.30	19.03	2.61	14.28%	13.73%
FFT arrays	14.82	15.36	0.21	1.42%	1.37%
pMM	190.58	280.00	14.56	7.64%	5.20%

pMM using RMI over Javia-II (Figure 4.10) achieves a peak performance of 120Mflops, which is less than 40% of that achieved by pMM over Javia-II. The high communication time is partly attributed to context switches between the main and the remote object threads[24].

4.3.3 Estimated Impact of Serialization

To evaluate the effect of high serialization costs more precisely, we estimate the fraction of the communication time and of the total execution time in which the processor spends in serialization alone. The methodology relies heavily on the application's structured communication pattern and exploits two facts: (i) the cluster nodes have a single processor, and (ii) each application invokes a single remote method during the communication phase. For each processor, the total number of *incoming* RMIs during the communication phase is multiplied by the cost of *de-serializing* the arguments (both type and size are considered); the total number of *outgoing* RMIs[25] is multiplied by the total cost of *serializing* the arguments. The sum of the two resulting quantities is an estimate of the time spent in object serialization. Table 4.5 summarizes the communication profile of the structured applications in RMI benchmark suite.

Table 4.6 shows that the estimated serialization and de-serialization costs can account for as much as 15% of an application's execution time.

---

[24] Pipelining RMIs with multiple sender threads to hide network latency improved the communication time by less than 10%.

[25] The total number of incoming and outgoing RMIs reported in Table 4.5 have been validated by runtime RMI profiling.

4.4 Summary

The performance of object serialization is currently inadequate for cluster computing. Java's type safety causes array serialization to be over an order of magnitude higher than basic communication overheads as well as memory-to-memory transfer latencies. Better compiler technology will unlikely yield substantial improvements in serialization. Because of data copying, serialization costs grow as a function of object size. This essentially nullifies the "zero-copy" benefits offered by modern network interfaces.

The Java I/O model dictates that objects be serialized and de-serialized via cascading I/O streams, which leads to inefficient data access and buffering [NPH99]. Setting up these streams is also very costly: for example, experiments with RMI over Javia-II indicate that the round-trip latency of a null, optimized RMI is 5x faster than that of an RMI with one integer argument. Overall, serialization costs are estimated to account for 3% to 15% of total execution time of communication intensive applications.

4.5 Related Work

4.5.1 Java Serialization and RMI

KaRMI [NPH99] presents a ground-up implementation of object serialization and RMI entirely in Java. Unlike Manta (see below), the authors seek to provide a portable RMI package that runs on any JVM. On an Alpha500/ParaStation cluster, they report a point-to-point latency of 117μs and a throughput of over 2MBytes/s (compared to a raw throughput of 50MBytes/s). The low bandwidth is attributed to several data copies in the critical path: on each end, data is copied between objects and byte arrays in Java and then again between arrays and message buffers. The copying over-

head is so critical that the serialization improvements over JDK1.4 vanish quickly as transfer size increases.

Several other projects [JCS+99, CFK+99] have shown that the performance of serialization is poor in the context of messaging layers such as MPI.

Breg et al. [BDV+98] recognizes the poor performance of Java RMI but advocates a "top-down" solution: it designs a subset of RMI that can be layered on top of the HPC++ runtime system. Krishnaswamy et al. [KWB+98] improves the performance of RMI over UDP with clever caching.

4.5.2 High Performance Java Dialects

Manta [MNV+99] is a "Java-like" language and implements Java RMI efficiently over Panda, a custom communication system. Manta relies on compiler-support for generating marshaling and unmarshaling code in C, thereby avoiding type checking at runtime. It communicates using both JDK's serialization protocol for compatibility as well as a custom protocol for performance. Manta is able to avoid array copying in the critical path by relying on a non-copying garbage collector and scatter/gather primitives in Panda. The authors report a RMI latency of 35µs and a throughput of 51.3MBytes/s on a PII-200/Myrinet cluster, which is within 15% of the throughput achieved by Panda.

Titanium [YSP+98] is a Java dialect for parallel computing that is inspired by Split-C [CDG+93], a parallel extension to C with split-phase operations. Titanium is designed first for high performance on large-scale multiprocessors and clusters, and only second to safety, portability, and support for building complex data structures. Titanium supports contiguous

multi-dimensional arrays that map efficiently onto bulk transfers in Active Messages.

4.5.3 Compiler-Support for Serialization

More generally, previous work has demonstrated that optimizing stub compilers are required to reduce the serialization overheads that plague many distributed systems for heterogeneous environments. Schmidt *et al.* [SHA95, GS97] studied the performance of `rpcgen` and two commercial CORBA implementations. They reported that traditional stub compilers produced inferior code compared to hand-written stubs. The Flick IDL Compiler [EFF+97] uses custom intermediate representations and traditional compiler optimizations to produce stub code that is superior to most stub compilers. Object serialization in Java is inherently more expensive because object types (i.e. classes) have to be serialized as well.