TECHNISCHE FAKULTÄT

Department Informatik

Technical Report CS-2010-04

Bernhard Haumacher, Michael Philippsen, and Walter F. Tichy

Irregular data-parallelism in a parallel object-oriented language by means of Collective Replication

January 2010

Please cite as:
Bernhard Haumacher, Michael Philippsen, and Walter F. Tichy, “Irregular data-parallelism in a parallel object-oriented language by means of Collective Replication,” University of Erlangen, Dept. of Computer Science, Technical Report CS-2010-04, January 2010.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.informatik.uni-erlangen.de

Irregular data-parallelism in a parallel object-oriented language by means of Collective Replication

Bernhard Haumacher,* Michael Philippsen,# and Walter F. Tichy*
(haui@haumacher.de, philippsen@informatik.uni-erlangen.de, tichy@ira.uka.de)

*: University of Karlsruhe
Computer Science Department, IPD
Am Fasanengarten 5, 76133 Karlsruhe, Germany

#: University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3, 91058 Erlangen, Germany

***Abstract*—In parallel object-oriented languages it is hard to *elegantly* express *efficient* data-parallel operations on objects of an irregularly-shaped object structure that is spread across the parallel computing environment.**

This paper presents a new programming model that smoothly integrates both task and data parallelism in a distributed object-oriented context. So called Collective Replication combines enhanced data locality for parallel tasks with data-parallel computations on irregular data structures. Collectively replicated objects exploit the bulk-synchronous data-parallel pattern in an object-oriented language and relieve the programmer from explicitly coding the communication step even for irregular data structures. To make the consistency protocol efficient, only modified fields are shipped instead of whole objects. Moreover, a novel graph coloring approach is used to broadcast updates to all replicas; this technique avoids bottlenecks and is more efficient than known approaches.

We suggest Java language extensions that can be handled by a pre-processor and achieve good performance on a set of benchmark applications.

I. Introduction

To program parallel computing systems, e.g. clusters of workstations, often a task-parallel, i.e. process- or thread-based, approach is used. In this MIMD mode, processes or threads are spread over the parallel system and perform their activities in isolation, except for explicit synchronization barriers. The distributed address space is either visible to the programmer and explicit communication messages must be used to bridge it, as for example in MPI [1]. Or a distributed object model is available [2], [3], [4], [5] that lets the programmer access remote objects by means of remote method invocations through local proxy objects [6], [7]. Such remote method invocations are always slower than local invocations.

The alternative is the data-parallel programming model (SPMD or SIMD) in which the same operations are applied to all elements of a data set in parallel. The most prominent representative of this approach is High Performance Fortran [8] in which the programmer only needs to specify the distribution of data structures.

Bal and Haines have surveyed approaches that integrate task and data parallelism [9]. Some of them enrich a data-parallel language with tasks; others add data-parallel constructs to a task-parallel object-oriented language. Unfortunately, none of these systems is capable of dealing with irregular non-array object structures.

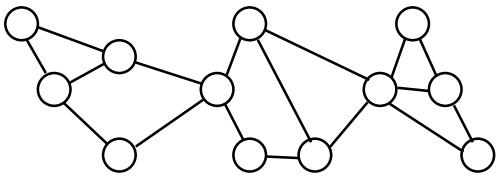

Fig. 1. Irregular data struct.

Fig. 1 shows a wire-frame model typical for finite element simulations. Although the solving process on such an irregular data-structure is defined in a data-parallel way (the state of each object is updated concurrently and depends on the values of its neighbors), traditional array-based data-parallel programming systems do not lead to a straightforward implementation.

Astonishingly at first glance, also a task-parallel approach is inappropriate. In such systems, one would distribute the data structure and simulate data-parallelism by partitioning the data structure among a set of worker threads, where all workers compute the state updates within their partitions in parallel. Unfortunately, this approach is neither efficient nor elegant. A naive ap-

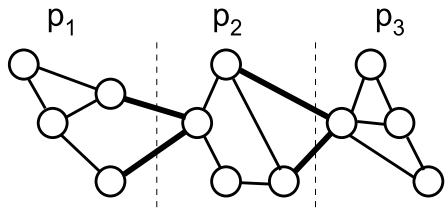

Fig. 2. Distrib. wire-frame.

proach would use a remote class for the graph nodes so that their methods can be accessed by remote method invocation. Fig. 2 shows how the data structure can be partitioned into three parts $p_1 \ldots p_3$, one for each thread. All graph nodes are connected by remote references but only some of the references (denoted by thick lines) actually cross partition boundaries. A thin line also requires an indirect access through a proxy object since in this naive approach all graph nodes are instances of a common remote class. This is slow, even though there are ways to optimize local accesses to potentially remote objects [10].

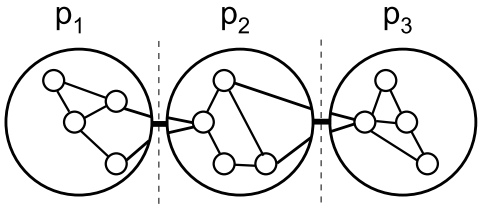

Fig. 3. Data partitions enclosed in remote objects.

Fig. 3 shows a more advanced solution that encapsulates each subgraph into a single remote object, one per thread. The objects within the capsule can only be accessed locally but access is much faster. However, this approach requires a complete redesign of the algorithm to add the extra hierarchy level to the data structure. This is not just inelegant but renders the resulting code error-prone and hard to maintain since it hides the original algorithmic idea.

In addition to the shortcomings mentioned above, both approaches also cause a high frequency of tiny remote accesses at runtime. Every single remote access to an object in the neighboring partition causes network traffic and latency. Hence, in both approaches, the programmer has to add message combining to achieve efficiency. This distorts the algorithmic idea even further.

Collective Replication, as proposed in this paper, guarantees fast local access without the need to redesign the algorithm due to the memory hierarchy, while it also performs node-to-node communication in burst mode.

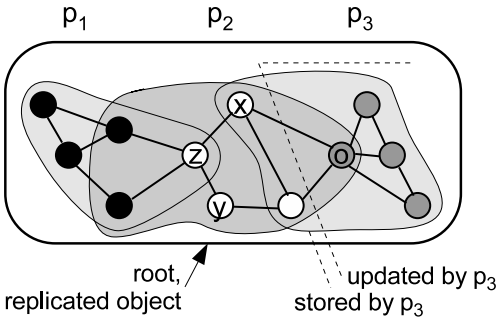

Fig. 4. Data structure with replicated object.

In our approach, all the threads *conceptually* have access to a single *replicated* object namely the root of the complete wire-frame model. But at runtime, only those objects are stored locally on a processor that are actually needed there. All the threads that perform the computation form a *collective*. During a data-parallel computation step, each worker of the collective concurrently updates its part of the replicated object (black, white, or gray objects in Fig. 4). Because information of the neighboring nodes is necessary workers concurrently access (and redundantly store) overlapping regions of nodes (shaded areas in Fig. 4). After such a step, the local modifications are merged *automatically* into a consistent view that is required before the next step. For example, if p_2 updates an object `x` that is also redundantly stored at p_3, p_3 needs the fresh values before it can access the object in the subsequent computation. Collective Replication ensures that each worker can see and update its segment of an irregular data structure in a data-parallel fashion while in the overlapping regions, objects are automatically replicated and kept consistent. Collective Replication is slightly similar to the idea of ghost cells, that in many MPI programs hold the most recent values of array fields assigned to neighboring processors. But Collective Replication is neatly integrated into an object-oriented language (replica consistency is defined according to the Java memory model), it is not restricted to array data, and it automatically and efficiently updates the ghost values, e.g., by combining and sending all update messages in burst-mode.

The remainder of this paper is structured as follows: Section II discusses related work on virtually shared memory and object replication. Section III presents Collective Replication and its implementation in more detail. Section IV covers efficiency issues of the consistency protocol and the graph coloring approach. Section V

gives performance numbers.

II. Related work

Data-parallel Orca [11] and Braid [12] integrate data-parallelism into a task-parallel object-oriented language. In Orca arrays can be wrapped into a shared object on which parallel operations can be performed. In Braid classes can be declared data-parallel. Methods are invoked in parallel on collections of instances of such classes. Neither of these languages is able to deal with data-parallelism on irregular data structures. It is the central contribution of this paper that our novel type of replication is a means to bridge both approaches.

In the literature, the most important reason for adding replication to a programming system is the aim to speed-up concurrent read accesses by making copies of shared data available locally. There are several known approaches to keep the replicated data consistent, i.e., to avoid that threads work on outdated data after some other threads have written to their copy of the data.

One extreme approach is function shipping, i.e., to invoke a modifying operation on all copies [13], [14]. The problem and the reason for us not to use it, is that replicated data needs to be completely separated from regular data. Since shipped functions cannot interact with local data whose values might differ between nodes, there may be no pointers from the replicated data to any local data. Pointers from local data to replicated objects can only be indirect through proxies. Hence, local access to replicated data remains more expensive than purely local access. The necessary clear separation of data structures requires distortions of the algorithms.

The other, more traditional approach, is to ship objects around, e.g. [15], [16], [17], [18], [19]. Whenever a local copy is modified its new contents must eventually be shipped to other nodes before the changed data is used there. Approaches vary with respect to where update operations can be performed: Either any node can apply for the right to update or there are fixed home nodes (and therefore bottlenecks). Independent of the consistency protocol, accesses to local copies must be guarded by proxies or wrapper objects to prevent access to outdated values. There are systems, e.g. [20], that grant direct access to the local copies, but those systems open themselves up to race conditions. Instead, our approach grants overhead-free direct access to local copies. By tightly merging replication with the memory consistency model of the language and its synchronization primitives and by adopting the bulk-synchronous parallel model [21] we prevent access to outdated values. Another frequent performance problem is that whole objects are shipped around even if only a single instance field is changed. It is unique to our approach, that we only ship the modified data instead of whole objects.

It is of course also possible to rely on the data replication mechanisms of an underlying (software) DSM system, e.g. [22], [23], [24], [25]. There are however, two main problems with that approach. First, since object granularity does not fit to page sizes, in general too much data is shipped. Specific object placement strategies and optimizations of page sizes [26], [27] improve performance but do not solve the general problem. Moreover, in general updated data is shipped too often since even a single changed value triggers the DSM system's general purpose consistency protocol. Most DSM systems do not provide any form of replication. Those that do, usually do not allow our flexible form of concurrent modifications to shared objects. It has been shown in [28] that the semantics of a language can be exploited to hold off flushing up to the end of a transaction, up to a synchronized block, or up to other language structures. Another language based approach [29] is to use a static analysis to determine that shipment of updates can be postponed and bundled if it can be shown that the new values are not immediately needed by other threads.

Our collectively replicated objects bridge the gap between data and task parallelism. We rely on Java's consistency control mechanisms to identify places where updated values need to be shipped. This allows for a proxy-free access to local copies that does not risk access to outdated values. Moreover, we only ship the modified instance variables in a burst-mode instead of shipping whole objects instantly whenever they are touched.

III. Collective Replication

Although collective replication is amenable to other parallel object-oriented programming languages as well, this section covers how to add it to JavaParty [2], an extension of Java for programming workstation clusters. JavaParty adds to Java so called remote objects that can reside on a remote node. Method invocations on a remote object are transparently transformed by the JavaParty pre-processor into RMI invocations through a local proxy object.

Replicated Objects. This paper adds replicated objects, yet another quality of objects. Whereas a remote object may reside on a remote node, a replicated object resides on all remote nodes (unless the degree of replication is restricted). Whereas remote objects are accessed remotely through proxy objects, replicated objects are

implemented by regular local Java objects, one per node; these local objects can only be accessed locally, i.e., conceptually there is no proxy object for remotely accessing a replicated object.

```
replicated class P {
   int x;
   String s;
   P p; //another replicated object
   static int y = 42;
   P(P p, int x) {
      this.p = p;
      this.x = x;
   }
   void foo() { ... }
   static void bar() { ... }
}
```

Fig. 5. Example declaration of a class of replicated objects.

As shown in Fig. 5, the programmer declares a class of replicated objects by means of the extra modifier `replicated`. The expressive power of the class is not restricted. (JavaParty's remote objects can also be used as fields, which is left out of this paper for brevity.) When an object of that class is created, as an initial distribution, a copy of it with all its instance variables is created on every node of the underlying parallel computing system. Threads access this copy, called *replica* in the remainder of this paper, locally on their node as they would access regular Java objects, i.e., there is no proxy indirection and no runtime overhead in accessing a replica.

For example, when the replicated object `root` shown in Fig. 4 is created and all the local objects of the wire-frame are added later on, each of the three machines p_1 to p_3 stores an exact copy of the complete wire-frame.

Below we show how all replicas of such a replicated object are kept in a consistent state and what that is.

Partial Replication. Since such a *full* replication (one replica on each node) consumes memory on every node, the programmer might choose to restrict the degree of replication.

```
void distribute(ReplicatedObject p, Object o,
                int[] distribution);
```

Fig. 6. Library routine to specify non-standard distributions.

By means of the library function `distribute` (Fig. 6) the programmer can specify that a local object `o` that can transitively be reached through references fields of the replicated object `p` will only be available on those cluster nodes that have their number mentioned in the array (3rd parameter). On all other nodes, an instance variable that would otherwise refer to that object is set to `null`. (As usual, dereferencing `null` will cause a runtime error.) This is called *partial* replication.

Assume an initial full replication of the example wire-frame model in Fig. 4, i.e., the three shaded areas would contain all nodes. To restrict the degree of replication, let the programmer then call (on an arbitrary machine):

```
distribute(root, x, new int[]{2,3});
distribute(root, y, new int[]{2});
```

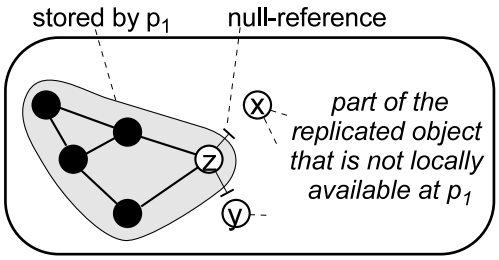

Fig. 7. Replica of node p_1 after `distribute`.

Then the replica at node p_1 will no longer consist of the complete wire-frame. Instead, some nodes are pruned and replaced by `null`-links as shown in Fig. 7. (The two invocations do not change the shaded areas for p_2 and p_3. They still contain all nodes.)

There is no need to explicitly `distribute` every object. If an object `x` refers to another object `o`, this object implicitly takes on the distribution of `x`. In the running example, the above `distribute` for `x` will implicitly make `o` locally available on node p_2 as well.

In all the applications we have looked at, explicit distribution has been used quite rarely, whereas often implicit distribution was sufficient. The rationale for a runtime library instead of a static specification is the need to change replication depending on algorithmic phases. The runtime overhead of a more comfortable solution that would create replicas on demand was too high.

Replica Consistency. From a bird's eye view, a replicated object is in a consistent state, if all its replicas are identical. In more detail, a consistent state requires that a deep comparison of all the local objects that can be reached through the replicated object's fields have identical values in their fields of primitive types. References to local objects may, however, be set to `null` on some nodes due to an explicit `distribute`, while on other nodes regular objects are reached.

Similar to Java where a thread is only guaranteed to see the changes made by different threads if both threads have seen an explicit synchronization, nodes can work on their replica in isolation, i.e. without an explicit synchronization no other node will ever see a change.

A. Flavors of Synchronization

With replicated objects, a single type of synchronization that guarantees exclusive access to an object is no longer sufficient since the main reason for replication is to allow concurrent access to locally available replicas (with as little overhead as possible). Hence, we added a multiple-reader, single-writer locking for replicated objects.

```
shared synchronized(p) { //reader
   ... //read access to replica
}
...
exclusive synchronized(p) { //writer
   ... //update replica and notify about change
}
```

Fig. 8. Shared and exclusive replica synchronization.

Fig. 8 shows our extension to Java's synchronized blocks that allows concurrent threads in a *shared* block as long as there is no thread in an *exclusive* block at the same time. Synchronized methods can also be flagged as `shared` or `exclusive`. As usual, at some point in time, after a modifying thread leaves the exclusively locked block and before another thread enters some synchronized block on the same object, all changes must be made available to the second thread. Note, that `wait` and `notify` work on replicated objects as known from regular Java, even though the signaling needs to span node boundaries, see [30].[1]

Our object replication is called *collective* replication because of the third flavor of synchronization, called **collective synchronization**. It is signaled by means of the modifier `collective`. In contrast to shared and exclusive synchronization, a thread must enter a collectively synchronized block of code on every node that carries a replica. Together these threads form a *collective* that cooperatively work on the replicated state. In the wire-frame example three threads are working on the replicated object, each on its portion of the replicated state, to collectively perform an updating step. After acquiring a collective lock, each thread is allowed to modify its portion of the replicated object's state, while concurrently, other threads modify their portions. Within the synchronized block, a thread does not see the updates performed by other threads. Only at the point of lock release (but not earlier), all the modifications are merged and are then made available on all the nodes.

Consider Fig. 4 again. Both p_1 and p_2 can access the white local object `z`. If either node changes some fields of Z within a collectively synchronized block, the other node will not see these changes immediately. Due to the bulk synchronous paradigm, the modifications are only shipped to it when all nodes leave the block. This is in line with Java's memory model, where changes made by one thread must be made visible to other threads only after a synchronization operation. If both p_1 and p_2 modify different instance fields of `z` concurrently, only the modified fields are exchanged. However, it is in general considered a program error if the modified portions of the replicated state overlap, i.e., if both p_1 and p_2 alter the same field of `z` to different values. In this case it is unspecified which of the two values survives.

```
class A {
   /** @merge AdditiveInt */
   int counter;
}
class AdditiveInt extends MergeValueInt {
   public boolean chngd(int orig,int copy) {..}
   public int diff(int orig,int copy) {..}
   public int patch(int orig,int delta) {..}
}
```

Fig. 9. Merging of conflicting updates in collective synchronization.

However, the programmer can specify merging routines that handle such concurrent updates of a single instance variable. See Fig. 9. Whenever multiple threads in a collectively synchronized block change the instance variable `counter` in their local replica, the methods of class `AdditiveInt` are used at the end of the synchronized block to merge the concurrent updates.[2]

B. Transformation and Compilation

Pre-processor. Our pre-processor compiles the language extension into regular Java. Whereas a `replicated` class is straightforwardly turned into a class that inherits from a library class `ReplicatedObject`, transformation of synchronization is slightly more involved.

To maximize efficiency for shared synchronization, we use the trivial translation of a shared synchronized

[1]The annotation of synchronization with the modifiers `shared` and `exclusive` serves a similar purpose as the `ReadWriteLock` classes introduced to Java in [31]. The difference is more a matter of taste. We wanted to stress that the annotation of synchronization targets a distributed environment, while the Java synchronization utilities are limited to a single JVM. This decision is similar to us adding a modifier `remote` to the class declaration where some interface technique might have been used instead. The collective replication introduced in this paper is independent of such cosmetic decisions.

[2]In general, for merging an instance variable of type `V` the annotation must specify a class that inherits from `MergeValue<V>`. Since Java's generics exclude basic types, our library provides special super classes for basic types, e.g. `MergeValueInt`.

block into a regular Java synchronization on the local replica. This has two implications: Firstly, shared synchronization is not as shared as it could be. Only if requested from threads on different nodes, access is granted concurrently. If several threads *from the same* VM try to acquire a shared lock, the VM sequentializes these threads. Secondly, exclusive synchronization has to remotely acquire the Java locks of all the replicas. Although this seems to cause a performance problem, all our experiments with hand-made reader/writer locks or centralized lock managers turned out to be slower, especially for the much more frequent case of acquiring shared locks.

```
Object handle = p.preAcquire();
synchronized (p) {
   p.postAcquire(handle);
   //exclusively synchronized code
   p.preRelease(handle);
}
p.postRelease(handle);
```

Fig. 10. Transformation of an exclusive synchronization.

Fig. 10 shows the transformation result for an empty exclusively synchronized block. The `pre` and `post` calls that are wrapped around regular Java synchronization, remotely acquire locks of all other replicas. To guarantee deadlock freeness we use a ticketing mechanism, see [32], [33].[3] We choose to use a push-based approach to achieve replica consistency, i.e., at the end of the exclusively synchronized code block in `preRelease()`, modifications done to the local replica are collected, bundled, and distributed to all other replicas where they are then re-played in the same way as they were done to the local replica. Several modifications can be replayed in any temporal order or at once since observers cannot enter a synchronized block before the re-play is completed. For more details see section IV.

```
synchronized (p) {
   //collective modification
   p.collectiveUpdate();
}
```

Fig. 11. Transformation of a collective synchronization.

To transform a collective synchronization (see Fig. 11) we use Java's regular synchronization on the local replica with an additional library call for differencing and merging modifications at the end of the synchronized block.

Notifications on replicated objects. In contrast to notification within a single VM, a notification on a replicated object must distribute the signal to all replicas where threads may wait. Waiting in an exclusively synchronized region has to temporarily release the remotely acquired locks on all replicas. While waiting is allowed within shared and exclusively synchronized blocks, notification can only be used within exclusive synchronization. Otherwise, a runtime exception is thrown. This is not only in line with Java's throwing of an `IllegalMonitorStateException` if notification is attempted outside of synchronized code. But there is a semantic reason as well: notification always announces a state change that sensibly can only occur within an exclusively synchronized context.

Since `wait()` and `notify()` cannot be overridden in class `Object` we need to add `replicatedWait()` and `replicatedNotify()` instead.

Marshaling of references. References to replicated objects can be used as method arguments. For remote methods, it is necessary to slightly extend the RMI implementation to handle such arguments. We have extended the KaRMI [34] fast remote method invocation library for cluster computing.

Let us look at JavaParty's regular (non-replicated) remote objects first. Whenever a direct reference to a remotely accessible object is passed as an argument, this reference needs to be replaced by a proxy object during marshaling. Unlike local objects, the remotely accessible object itself must not be transmitted along with the call. Instead, the new proxy object forwards all received calls back to the original object.

For the replicated objects introduced in this paper, a reference to a local replica must be replaced by a globally unique identifier on the sending side. This identifier is then turned into a reference to the (already present) replica on the remote side of the RMI implementation. (Remember that for a remote object a local replica exists on every node.) Hence, in contrast to remote objects, there are replacement operations on both sides of the object serialization. Note, that references to replicated objects cannot be passed to nodes that do not store a replica due to a manual non-standard distribution. Otherwise, a runtime error will be triggered.

IV. Efficient consistency propagations

A. Recording and playback of modifications

In contrast to earlier consistency protocols that ship modified replicas as a whole, we ship modified fields only. There are two options to do this: Either all field-modifying operations can be recorded at the time they

[3] See [30] for how to separately call `monitorenter` and `-exit` on remote objects without violating Java verifier constraints.

happen and stored for later transmission, or modifications can be reconstructed after they have happened by computing the difference of the source object graph to a backup copy.

We choose the latter approach because the necessary amount of extra memory space is known statically (while in the first approach it depends on the number of modifications that happen at runtime) and because it does not slow down regular operations. This is especially important because we consider the whole local subgraph of objects rooted at the replica. Since potentially every type of object can be referenced from a replica, all modifying instructions of a program would otherwise have to be instrumented with code for recording.

Unlike serialization, object differencing is no standard feature of Java. For its implementation there are also two options: Either the state of matching objects can be compared by means of dynamic introspection, or classes can implement differencing methods. Since dynamic introspection is slow we choose the second option requiring that each object that is (directly or indirectly) referred to by a replica must implement the interface `Patchable` as sketched in Fig. 12. Of course, the methods of the `Patchable` interface need not be handcrafted by the programmer. Instead, our preprocessor automatically injects the required code into a class when needed.[4]

```
public interface Patchable {
   public void createPatch(Object copy,
                           PatchOutput s);
   public void applyPatch(Object copy,
                          PatchInput s);
}
```

Fig. 12. Interface required for differencing.

The method `createPatch()` compares each instance variable of `this` with the corresponding instance variable of the backup `copy`. Differences are reported to `PatchOutput` that encodes them for transmission. The inverse operation is performed by `applyPatch()` which applies all modifications supplied by `PatchInput` to both `this` and its backup.

Our differencing algorithm (that only monitors truly replicated local objects that belong to more than one replica, i.e., that lie in intersections of more than one shaded area in Fig. 4) can deal with all kinds of modifications: Besides modifications of instance variables of basic type, new objects can be added to the graph, references can be changed within the graph, and objects can be removed from the graph. The algorithm is similar to and closely integrated into the cycle resolution of Java's serialization [35]. (Some identifier tables are even used for both purposes). Java's serialization already established a mapping between objects and Id's that allows an exact reconstruction of the graph structure at the receiving side. We also build such a mapping, but in contrast to serialization, we keep it between different updates of the replicated object. This builds a relation between objects belonging to two or more replicas. During an update, all modified shared objects are updated by applying a patch regardless of the number of modified instance variables. Even if all instance variables are affected by the modification, the object cannot be marshaled by regular serialization because this would create a fresh copy of that object at the receiving side, and would cause the risk of dangling pointers to the outdated copy.

[4]For library classes, e.g. strings, arrays, and collections, there is an alternative way of specifying the differencing functionality in a separate descriptor class.

B. Update propagation after exclusive blocks

When collective replication is available, there is not much need for exclusive updates – at least in our experiments. Nevertheless, they must be implemented with sufficient performance.

Only in case of full replication, all replicas are identical. Hence, it would be possible to broadcast a single patch to all other nodes to complete the exclusive block. But in practice, partial replication is much more common. Since only a few local objects are stored on more than one node, it turned out to be more efficient to compute a specific difference for each target node (that takes into account which local objects live on both nodes) and to send the resulting target-specific patches in a point-to-point scheme. Therefore, while synchronization barriers themselves are implemented by means of a logarithmic communication tree, the update propagation after an exclusive block scales linear with the degree of replication. We found that this rather naive scheme is sufficient when a fast collective update is available.

C. Collective Update

At the end of a collective synchronization, potentially all replicas have been modified locally. Therefore conceptually, an all-to-all communication is needed, in which each node computes a target-specific patch for every other node. Although in theory, a quadratic number of communication messages is needed, properly distributed data with little overlap between nodes in practice often

results in much less network load. Often only neighboring processors have to exchange patches for the data shared among them.

But it is not just the network load that causes problems in collective updating. It must be guaranteed that the concurrent propagation of patches does make progress even if system resources are limited. The first problem is that the nodes may need a lot of storage to store the patches. A node that waits to send its patches away might not have enough space to receive patches from other nodes. A second problem is that if for performance reasons only the threads are available that form the collective, they can at one time either generate and send a patch or receive and process a patch. Both problems can easily cause deadlocks if within the collectively synchronized block a lot of data has been modified by all the nodes: For any given size and number of communication buffers, a replicated object can be constructed whose replicas are roots of enough local object graphs that cause the buffers to be too small to store the patch information. Resizing the OS level buffer pool, or allowing the OS to use swapping, or adding additional service threads obviously are no options for performance reasons. (With upcoming multi-core architectures the latter might change, though.)

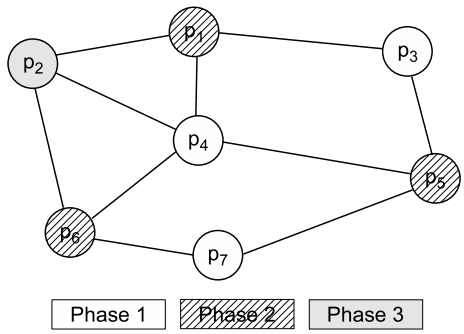

Fig. 13. Update in phases.

Thus, with any fixed size communication buffers and if only one thread per node is available, collective update must be performed in phases. Within a phase, a single node can either generate and send a patch or that node can receive and process a patch. Fig. 13 shows seven replicas as circles. Two replicas are connected iff they share at least one local object, i.e., if some update information needs to be exchanged. Two replicas that are connected must not be sending in the same phase. In the example, nodes p_3, p_4, and p_7 generate patches and send them to their neighbors in the first phase. In phase 2, nodes p_1, p_5, and p_6 are actively sending patch information. In the last phase, only node p_2 sends patches to its neighbors. The similarity to the graph coloring problem is apparent.

Starting from the parallel graph coloring in [36] that operates in synchronous steps, in our new algorithm the nodes operate asynchronously. This is much more appropriate for a cluster environment. Moreover, whereas known graph coloring algorithms compute the coloring from scratch, our new algorithm is incremental, i.e., if the last found coloring is still valid it terminates instantly. Only if a changed distribution and a changed partial replication causes different edges in the update graph, a new coloring is derived from the old one. Our new incremental and asynchronous graph coloring algorithm is sketched in more detail in the appendix.

Of course, the collective update and the necessary graph coloring are performed automatically by the runtime system, i.e., completely transparent to the programmer.

V. Evaluation

We have used a Myrinet-connected 16 node cluster. Each node has two 800 MHz Pentium III and runs SuSE Linux 2.4.21-smp4G, GM 2.016 and the server version of Suns 1.4.2-06 JVM. Since with newer JVMs with more advanced just-in-time compilers we got better results, we present rather conservative numbers here.

A. Microbenchmarks

Fig. 14 shows the duration of an exclusive synchronization on a replicated object. The x-axis gives the number of replicas. The first bar of each group shows the cost of a synchronization on an empty object. For the next two bars we used an 8×8 square wire-frame of linked objects, each of which holds a float (in addition to the links to neighboring objects). Only for the last bar, some values have been changed so that these changes are communicated to update the other replicas.

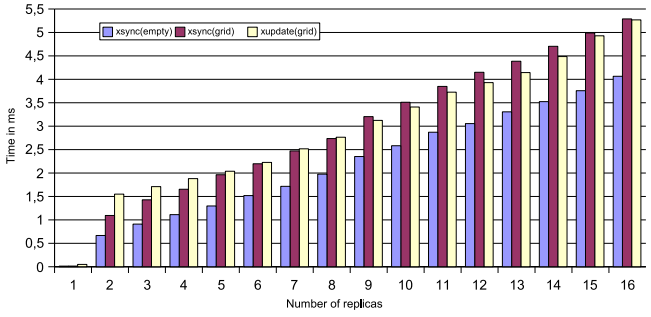

Fig. 14. Cost of exclusive synchronization; worst case.

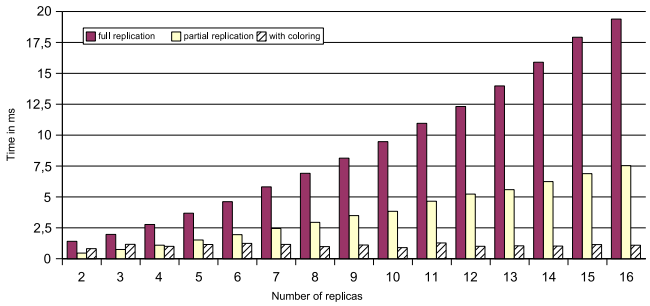

Fig. 15. Cost of collective synchronization.

Only if there actually exist at least two replicas at runtime, extra time is needed on top of Java's regular synchronization. Only if it is statically unknown for a partial replication how many replicas exist and on which nodes they reside, our implementation uses a 1:1-form of messaging that causes the worst case behavior, i.e., the linear growth of synchronization cost shown in Fig. 14. The size of the accessed object is not very important. And it does not affect the runtimes a lot if in addition to synchronization, patch data must be shipped. For a full replication, a logarithmic broadcast is used instead. With a fan-out of 4 (that turned out to be best) an exclusively synchronized access to 16 replicas takes less than 1 ms (not shown in Fig. 14).

In case of full replication, a collective synchronization takes a quadratic time to update all replicas. Fig. 15 shows the numbers (first bar in each group). For partial replication we consider a 2D-array that is distributed in a row-wise fashion; each node also stores copies of the two rows that belong to neighboring nodes. Updates are much faster (2nd bar). The times still grow in a linear way since graph coloring has been switched off for the 2nd bar. But because of the cyclic data dependence in that case the graph can be colored with two or three colors depending on the node count being even or odd. This leads to two or three steps for the parallel updates of the collective synchronization, i.e., a constant time, regardless of the number of replicas of the replicated object (3rd bar).

B. Application Benchmarks

Traveling Sales Person, TSP. Finding the shortest Hamiltonian circle in a weighted graph is NP-complete. When enumerating all routes, the search space can be pruned by skipping initial route segments that are already longer than the currently best route. In a distributed implementation, first this current minimum is read frequently and hence needs to be replicated; updates require an exclusive synchronization. Second, the search space needs to be distributed among the nodes. Unfortunately, it is statically unknown how many routes unfold from an initial route segment. Hence, to achieve a good load balance at runtime, a work stealing approach is needed.

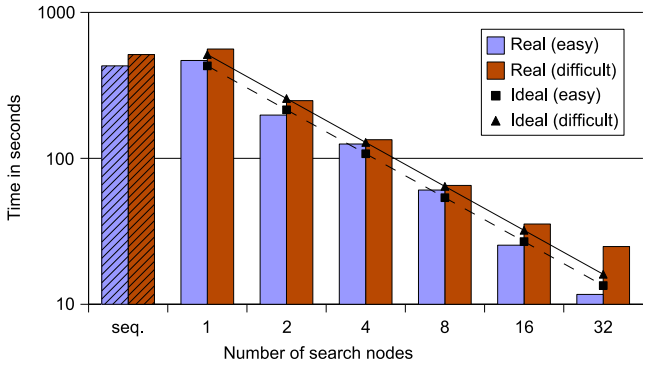

Fig. 16. TSP for an easy and a difficult problem.

With only minimal changes to the code and without distorting the original algorithm, we achieve amazing performance results. Fig. 16 shows the performance we achieve on both an easy and a difficult problem set. In easy problems good current minima are found soon. Hence, the search space is pruned in a speedy way. Note, that local access to our potentially replicated objects is very fast – therefore, the parallel runtimes on one node (2nd group of bars) is not much slower than a purely sequential TSP implementation (1st group of bars). The two lines show the ideal speedup. Superscalar speedup can be seen if a parallel search finds a better current minimum sooner. Reduced efficiency is caused by work stealing attempts that transfer fragments of the search space that turn out to be too small.

N-Body. When n particles move in space, every particle's position and gravity influences every other particle. Barnes and Hut [37] have suggested an $O(n \cdot \log(n))$ approximation that groups particles hierarchically and allows to consider few virtual masses instead of many individual particles. The farther a particle is away, the more likely it is that a replacement mass can be used instead. The algorithm groups the particles into a quad-tree. This tree structure needs to be (partially) replicated, since each node only needs to access a part of this quad-tree to compute the new positions and velocities of the particles assigned to it. Since the quad-tree changes due to the moving particles, collective synchronization is a perfect match for this problem. After each time step, the new distribution can be derived from the particles' positions in a straightforward way. It just takes a few

invocations of `distribute` to adapt the quad-tree for the next step of the simulation.

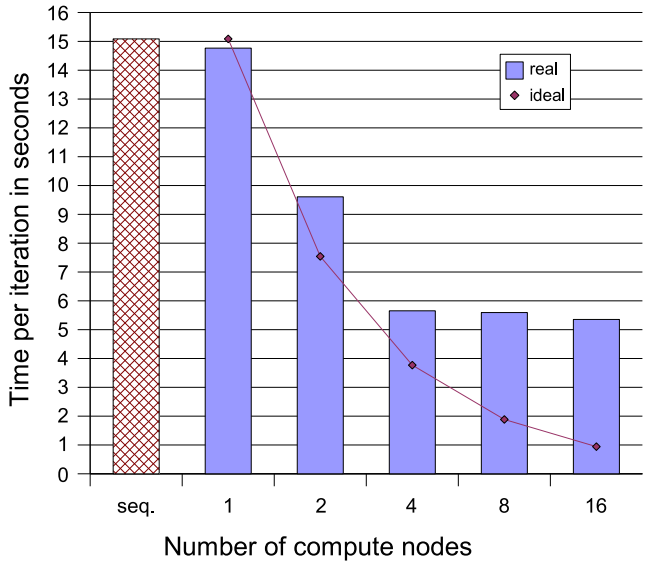

Fig. 17. N-body with 100.000 particles.

The implementation with a replicated quad-tree and collective synchronization scales well up to 4 nodes, see Fig. 17. Then our problem size becomes too small and too many objects in the quad-tree are replicated to too many nodes. Hence, the implementation suffers (a) from extensive update messages for the collective synchronization and (b) from memory and garbage collection pressure. We intentionally did not implement the following two optimizations: we did not (re-)distribute the work load dynamically and we did not leave out empty squares of the quad-tree. The reason is that we wanted to see what level of performance can be reached by a straightforward implementation of the ideas of Barnes and Hut on a cluster with a distributed data structure, i.e., without an involved manual restructuring of the code that all other n-body implementations known to us need to achieve their performance, e.g. [38], [39].[5]

Impact. Finally we have studied the public domain finite element simulation Impact [40] that can be used to predict dynamic events such as car crashes or metal sheet punch operations. The simulation works on a wire-frame model (similar to the one shown in Fig. 4) and computes forces and their effects. We have used partial replication and collective synchronization to express the underlying data-parallel algorithm.

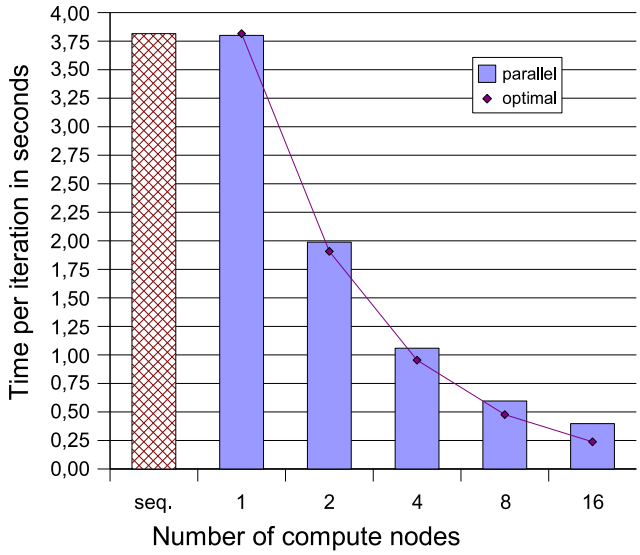

Fig. 18. Finite element simulation impact on a cluster.

In each step, the forces are applied from neighboring nodes and are transformed into acceleration, velocity, and movements. Since the forces of all surrounding nodes add up, we have used a merging routine (similar in spirit to the code shown in Fig. 9) to compute the combined effect of the collective update. Again, little work was necessary to make the irregular data-parallel algorithm work on a cluster with almost ideal efficiency.

Fig. 18 compares the runtimes of a single computation step of the sequential Impact version (1st bar) and its parallelized version (other bars). The simulation works on a wire-frame of 8399 nodes to compute the deformation of a hollow cylinder when being compressed along its axis. As the sequential version and the parallelized version have almost the same speed on a single processor the overhead of maintaining the replicated state is negligible. Partial replication with a graph coloring update strategy results in a near optimal scaling behavior.

VI. Conclusion

Collective replication, i.e. partial replication and the coupling of update messages for consistency of replicas to the memory consistency mechanisms of the language, is an efficient way to add data-parallelism for irregular problems to an object-oriented programming language with threads. Not only stays local access to potentially replicated objects overhead-free; updating of replicas to achieve consistency can also be made efficient, if

[5]Note, that we have implemented a 2-dimensional version of n-body, while the literatur often deals with 3-dimensional n-body problems. Hence, we see the common scalability problems from half as many nodes onwards.

only altered values instead of whole objects are shipped and if graph coloring is used to keep the number of update messages low and to avoid deadlocks. Collective replication made it straightforward to port a collection of irregular data-parallel application benchmarks to run with good speedups on a cluster.

APPENDIX

Iterative, asynchronous distributed graph coloring. Several heuristics are known that assign "few" colors to nodes of a graph so that no two neighboring nodes have the same color. E.g., in each "round" of a distributed algorithm from [36], the uncolored nodes suggest their own color. Then they exchange these suggestions with their neighbors. The proposed color is fixed if it is not in conflict with any of the colors selected by the neighbors. The authors have shown a complexity of $O(\Delta^2 log(n))$ for n nodes. The pseudo-code of the Asynchronous Distributed Largest First coloring algorithm (**ADLF**, see pseudocode below) adds a bidding mechanism and piggy-bags synchronization onto asynchronous color exchange messages. A suggested color is only fixed if the proposing node has the highest priority (largest rank and/or largest random number). The other nodes continue their bidding in the next round.

ADLF for node k:

Require: Node k is not colored. All neighbors of k are marked `active`.
  while node k is not colored **do**
    Choose color $c \in \mathbb{N}_0$ as $\min(\mathbb{N}_0 \backslash \{c_n \mid$ color c_n is `used`$\})$.
    Choose a random number r.
    Send CHECK$(c, \deg_k, r)$ to all `active` neighbors.
    Receive CHECK$(c_n, \deg_n, r_n)$ from all `active` neighbors n.
    Compare $(c, \deg_k, r)$ against all received parameters.
    if k has highest priority or color c produces no conflict **then**
      Send message COLOR(c) to all neighbors.
      Set node k colored.
    else
      Send CANCEL() to non-conflicting and lower priority neighbors.
    Receive reply messages.
  Receive missing color messages.
Ensure: k is colored, all neighbors got a color and are not `active`.

Receive reply messages:

  for all `active` neighbors n with higher priority or no conflict **do**
    Receive message m from neighbor n.
    if message m = COLOR(c_n) **then**
      Assign color c_n to neighbor n.
      Mark color c_n as `used`.
      Mark neighbor n as not `active`.
    else $\{m$ = CANCEL()$\}$
      Do nothing.

Receive missing color messages:

  for all `active` neighbors n **do**
    Receive message COLOR(c_n) from neighbor n.
    Assign color c_n to neighbor n.
    Mark color c_n as `used`.
    Mark neighbor n as not `active`.

ADLF must start from scratch if the structure of the graph is altered by adding or deleting nodes or edges. This is inefficient in our domain since our graphs often only change slightly. Hence, we have extended ADLF to start from a given coloring and to re-color only the area of the graph whose coloring is affected by the alterations. Our contribution is how we determine this modified area, in which nodes forget their previous color and become active again for a subsequent ADLF. Inactive nodes keep their previous color. See **IADLF** pseudo-code below.

A node checks whether it is affected by an alteration of the graph by comparing its color to the colors of its neighbors. If it would be possible to select a smaller color number, a bidding is started. Coarsely, a node is affected if it loses the bidding because of conflicting colors or neighbors with higher priority. The exact conditions are given in the pseudo-code. They ensure that recursively all neighbors of affected nodes with smaller rank will also forget their previous color and become active again for a subsequent invocation of ADLF. Area detection is no more complex than a single "round" of ADLF. Hence, while the overall asymptotic complexity does not change, IADLF often terminates quickly after area detection if the graph has only changed slightly.

IADLF for node k:

Require: k is colored with color c, all neighbors are marked `active`.
  if there is a color $c_u < c$ that is not `used` **then**
    Set c to c_u.
  Choose a random number r.
  Send CHECK$(c, \deg_k, r)$ to all `active` neighbors.
  Receive CHECK$(c_n, \deg_n, r_n)$ from all `active` neighbors n.
  Compare $(c, \deg_k, r)$ against all received parameters.
  if color c produces no conflict **then**
    Set `accept` to TRUE.
    for all `active` neighbors n with higher degree **do**
      Receive reply message m from neighbor n.
      if received message m was CANCEL() **then**
        Set `accept` to FALSE.
    if accept **then**
      Send message COLOR(c) to all neighbors.
    else
      Send message CANCEL() to all `active` neighbors.
    for all `active` neighbors n with lower or equal degree **do**
      Receive reply message m from neighbor n.
    if `accept` **then**
      Receive missing color messages.
    else
      Collective DLF for node k.
  else if k has highest priority **then**
    Send message COLOR(c) to all neighbors.
    Set node k colored.
    Receive reply messages.
    Receive missing color messages.
  else
    Send CANCEL() to all neighbors with lower priority or no conflict.
    Receive reply messages.
    Collective DLF for node k.

REFERENCES

[1] M. Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994.

[2] M. Philippsen and M. Zenger, "JavaParty - transparent remote objects in Java," *Concurrency: Practice & Exp.*, vol. 9, no. 11, pp. 1225–1242, Nov. 1997.

[3] P. Launay and J. Pazat, "A framework for parallel programming in Java," in *Intl. Conf. on High Perf. Computing and Networking*, Amsterdam, Netherlands, 1998, pp. 628–637.

[4] O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic layout of distributed applications in FarGo," in *Intl. Conf. on Software Engineering*, Los Angeles, CA, May 1999, pp. 163–173.

[5] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *ECOOP Conf.*, ser. LNCS, vol. 2374, Mlaga, Spain, June 2002, pp. 178–204.

[6] O. M. Group, "CORBA/IIOP specification (2.6)," Dec. 2001.

[7] S. Microsystems, "Java remote method invocation (RMI)," http://java.sun.com/products/jdk/rmi/.

[8] H. Forum, "High Performance Fortran language specification," *Scientific Programming*, vol. 2, no. 1-2, pp. 1–170, 1993.

[9] H. Bal and M. Haines, "Approaches for integrating task and data parallelism," *IEEE Concurrency*, vol. 6, no. 3, pp. 74–84, 1998.

[10] B. Haumacher and M. Philippsen, "Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters," in *CPC2001, 9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, June 2001, pp. 83–94.

[11] S. Hassen and H. Bal, "Integrating task and data parallelism using shared objects," in *Intl. Conf. on Supercomputing*, Philadelphia, PA, May 1996, pp. 317–324.

[12] E. West and A. Grimshaw, "Braid: Integrating task and data parallelism," in *5th Symp. on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995, pp. 211–219.

[13] L. Baduel, F. Baude, and D. Caromel, "Efficient, flexible, and typed group communications in Java," in *ACM-ISCOPE Conf. on Java Grande*, Seattle, WA, Nov. 2002, pp. 28–36.

[14] J. Maassen, T. Kielmann, and H. Bal, "Efficient replicated method invocation in Java," in *ACM Conf. on Java Grande*, San Francisco, CA, June 2000, pp. 88–96.

[15] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metacomputing on the web," in *9th Intl. Conf. on Parallel and Distributed Computing Systems*, Dijon, France, 1996.

[16] D. Hagimont and D. Louvegnies, "Javanaise: distributed shared objects for internet cooperative applications," in *Middleware'1998*, The Lake District, England, 1998, pp. 339–355.

[17] J. James and A. Singh, "Design of the Kan distributed object system," *Concurrency: Practice & Exp.*, vol. 12, no. 8, pp. 755–797, 2000.

[18] Y. Sohda, H. Nakada, S. Matsuoka, and H. Ogawa, "Implementation of a portable software DSM in Java," in *ACM-ISCOPE Conf. on Java Grande*, Palo Alto, CA, 2001, pp. 163–172.

[19] B. Topol, M. Ahamad, and J. Stasko, "Robust state sharing for wide area distributed applications," in *18th Intl. Conf. on Distributed Computing Systems*, Amsterdam, Netherlands, May 1998, pp. 554–561.

[20] M. Herlihy and M. Warres, "A tale of two directories: implementing distributed shared objects in Java," *Concurrency: Practice & Exp.*, vol. 12, no. 7, pp. 555–572, May 2000.

[21] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[22] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "TreadMarks: Distributed shared memory on standard workstations and operating systems," in *Winter 1994 USENIX Conf.*, 1994, pp. 115–131.

[23] J. Protić, M. Tomašević, and V. Milutinović, "An overview of distributed shared memory," in *Distributed Shared Memory: Concepts & Systems*, Aug. 1997, pp. 12–41.

[24] W. Yu and A. Cox, "Java/DSM: A platform for heterogeneous computing," *Concurrency: Practice & Exp.*, vol. 9, no. 11, pp. 1213–1224, Nov. 1997.

[25] W. Zhu, C. Wang, and F. Lau, "JESSICA2: A distributed Java virtual machine with transparent thread migration support," in *IEEE Intl. Conf. on Cluster Computing*, Chicago, IL, Sep. 2002, pp. 381–389.

[26] G. Antoniu, L. Boug, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst, "The Hyperion system: Compiling multithreaded Java bytecode for distributed execution," *Parallel Computing*, vol. 27, no. 10, pp. 1279–1297, Sep. 2001.

[27] M. Lobosco, C. Amorim, and O. Loques, "A Java environment for high-performance computing," Univ. Federal Fluminense, Rio de Janeiro, Brazil, Tech. Rep. RT-03/01, May 2001.

[28] X. Chen and V. H. Allan, "MultiJav: A distributed shared memory system based on multiple Java virtual machines," in *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, July 1998, pp. 91–98.

[29] R. Veldema, R. Bhoedjang, and H. Bal, "Distributed shared memory management for Java," Vrije Universiteit Amsterdam, Netherlands, Tech. Rep., Nov. 1999.

[30] B. Haumacher, T. Moschny, J. Reuter, and W. Tichy, "Transparent distributed threads for Java," in *5th Intl. Workshop on Java for Parallel and Distributed Computing at Intl. Parallel and Distributed Processing Symp.*, Nice, France, April 2003.

[31] D. Lea, "Java specification request (JSR) 166: Concurrency utilities," Sep. 2004, java Community Process, http://jcp.org/en/jsr/detail?id=166.

[32] M. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal, "An efficient reliable broadcast protocol," *Operating Systems Review*, vol. 23, no. 4, pp. 5–19, 1989.

[33] J. Kim and C. Kim, "A total ordering protocol using a dynamic token-passing scheme," *Distributed Systems Engineering*, vol. 4, no. 2, pp. 87–95, June 1997.

[34] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and RMI for Java," *Concurrency: Practice & Exp.*, vol. 12, no. 7, pp. 495–518, May 2000.

[35] S. Microsystems, "Java object serialization specification," Aug. 2001, available online from http://java.sun.com/j2se/.

[36] J. Hansen, M. Kubale, L. Kuszner, and A. Nadolski, "Distributed largest-first algorithm for graph coloring," in *EuroPar Conf.*, ser. LNCS, vol. 3149, Pisa, Italy, 2004, pp. 804–811.

[37] J. Barnes and P. Hut, "A hierarchical O(N Log N) force calculation algorithm," *Nature*, vol. 324, no. 4, pp. 446–449, Dec. 1986.

[38] H. Lu, A. L. Cox, and W. Zwaenepoel, "Contention elimination by replicating sequential sections in distributed shared memory programs," in *Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001, pp. 53–61.

[39] D. J. Scales and M. S. Lam, "The design and evaluation of a shared object system for distributed memory machines," in *1st USENIX Conf. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994, pp. 101–114.

[40] J. Forssell, "Impact," 2007, impact.sourceforge.net.