Implementing Automatic Coordination on Networks of Workstations

Christian Weiß
TU München
Institut für Informatik

Jürgen Knopp
Siemens AG
Private Networks

Hermann Hellwagner
TU München
Institut für Informatik

Abstract

Distributed shared objects are a well known approach to achieve independence of the memory model for parallel programming. The illusion of shared (global) objects is a convenient abstraction which leads to ease of programming on both kinds of parallel architectures, shared memory and distributed memory machines. In this paper, we present several different implementation variants for distributed shared objects on distributed platforms. We have considered these variants while implementing a high level parallel programming model known as coordinators [10]. These are global objects coordinating accesses to the encapsulated data according to statically defined access patterns. Coordinators have been implemented on both shared memory multiprocessors and networks of workstations (NOWs). In this paper, we describe their implementation as distributed shared objects and give basic performance results on a NOW.

1 Introduction

A key element of architecture-independent parallel programming has always been to hide the specific memory model of a parallel system – distributed memory or shared memory – from the programmer. In addition to the convenience and support for portability resulting from this, a maximum of performance should also be achievable using such a programming system. However, this goal of supporting both, independence of the memory model and highest efficiency, has not been fully reached to date.

Several approaches to memory model independent parallel programming systems have been pursued. Some examples are:

- Distributed shared memory (DSM): This well known approach of abstracting the physical distribution of memory has been proposed and implemented for years, mostly based on replicating and maintaining consistency of fixed-size blocks (cache lines or pages) and extending cache and virtual memory management [15]. Albeit several refinements have been introduced, DSM may still suffer from the drawback known as false sharing. This especially holds for programming languages such as C++ where objects tend to be spread over different memory regions (e.g., when objects are allocated partially on both the stack and the heap).

- Global pointers: Global pointers, as used in languages such as Split-C [3], allow for data distribution and remote accesses. While Split-C has efficient implementations, it partially exposes communication (i.e., cost differences) on shared data accesses.

- Distributed shared objects: The basic idea of DSM can be extended to objects: instead of fixed memory blocks, variable-size user-defined objects are considered. Languages such as parallel variants of C++ [19] and Orca [1] rely on this generalization. Although distributed objects can hardly be supported by the operating system or the hardware (due to the object model being an integral part of the language), we believe that distributed shared objects are an important concept.

This paper presents implementation considerations and results for a special case of distributed shared objects, called *coordinators*. Coordinators are global in the sense that they are shared between parallel tasks (regardless of whether threads or processes). As such, they can be implemented either as truly shared or as distributed shared objects. Moreover, coordinators allow implicit coordination (synchronization or remote coordination, depending on the implementation) of the tasks accessing the shared object. Aside from independence of the memory model, coordinators provide a high level of abstraction which allows to conveniently specify the parallel program behaviour. The programming model provided by coordinators as well as related work has been described in some detail in earlier papers [10, 11]. In contrast, in this paper we focus on coordinator implementations on distributed platforms.

The remainder of the paper is structured as follows. Section 2 reviews the basic concepts of coordinators. Section 3

introduces the problems associated with a distributed implementation of coordinators and presents several design considerations and trade-offs. An implementation on a network of workstations (NOW) is described, combining several of these implementation variants. In section 4, basic performance results are given and discussed. Section 5 provides some conclusions.

2 Automatic Coordination as a Means for Memory Independence

Parallel computations often follow certain algorithmic patterns which can be easily defined on an abstract level. In data parallel programs, several tasks are performing similar work on different objects or parts thereof. In scientific computing, for instance, concepts like fan-in and fan-out are being used. More precisely, one often finds patterns like: all tasks write or read an object a certain number of times before the object may be read or written, respectively. Thus, in physical or virtual shared memory, such objects must be protected in order to avoid races (sometimes called shared memory hazards). In distributed memory programming, these hazards do not occur. Instead, data must be moved, and coordination must be achieved, by means of remote communication, typically message passing.

The declarative approach of coordinators is based on the declarative specification of the above mentioned access patterns to objects and allows parallel programming in both shared and distributed memory environments without the need of explicit synchronization or remote communication. For instance, in C++ syntax,

> **coordinator**<**float**>
> myVal(**write**(**arbitrary(1)**),**read**(**each**(gr,**2**)));

specifies a coordinator which is being written once by an arbitrary task before it is read twice by each task that is a member of the task group gr. This pattern includes iteration: after being read twice by every task in gr, the coordinator may be written again. This coordinator can be used, for instance, to conveniently implement a producer-consumer scenario:

> **processGroup** gr(*list of procIds*);
> **coordinator**<**float**>
> myVal(**write**(**arbitrary(1)**),**read**(**each**(gr,**2**)));
>
> // *executed by potential producer processes*
> **while(true)** {
>   // *write access*
>   myVal=**produceValue();**
> }

> // *executed by each process in consumer group*
> **while(true)**
> {
>   // *read accesses*
>   **consumeValue**(myVal**);**
>   **consumeValue**(myVal**);**
> }

Notice that there is no need to explicitly synchronize producer and consumer actions; this is implicitly effected by the coordinator myVal.

Access patterns to a coordinator are declared separately for writing and reading. The parallel semantics is defined by switching between a *writable* and a *readable* state of the coordinator; the computation proceeds in terms of *write* and *read* phases. The patterns statically describe the conditions that trigger phase switches. These conditions are defined in terms of "who reads or writes an object how many times". Inappropriate (premature) accesses are delayed until the state switches.

Coordinators allow the main patterns *each* and *arbitrary*. The *each* pattern is used for a homogeneous group of tasks. Assume that the overall job within the write or read phase of a computational step can be defined by specifying the job of every single task within the phase. The intended standard way to specify such a job is to declare how many times the process will access the coordinator within the phase. For example, when a vector is normalized (see figure 1), each task writes the coordinator once before reading it once. If the number of accesses cannot be given, each task has to state explicitly that its job within the phase is finished, by invoking a special coordinator method.

In contrast, the *arbitrary* pattern is used if an arbitrary, possibly varying number of possibly inhomogeneous tasks share the coordinator. Within this cooperation mode, only global switch conditions can be provided. That is, the state of the coordinator switches after a statically defined number of accesses or when the state is explicitly switched by a task via a special coordinator method.

The interleaved accesses of all tasks must obey the statically defined access pattern. For example, in figure 1 each task must have finished its write access before any task is allowed to read the global value sum. The defined access pattern is enforced by the coordinator by blocking premature accesses. Note that aside from blocking of read accesses during the write phase, and vice versa, there may be blocking of reads in the read phase and of writes in the write phase. This is the case when the access actually belongs to the next phase of the same access type. In addition to blocking, mutual exclusion of accesses is guaranteed for all write accesses.

Readers familiar with *futures* [5] and *I-structures* might see some similarity here. Coordinators extend the implicit

```
void
normalize(vectorSection s, processGroup gr)
{

    double partial=0.0;
    coordinator<double>
     sum(write(each(gr,1)), read(each(gr,1)));

    // calculating partial norm
    for(int j = s.lower() ; j<= s.upper() ; j++ )
      partial += s[j] * s[j];

    // writing global sum
    sum += partial;

    // reading global sum - may block until all
    // group members have written once
    double norm = sqrt(sum);

    for( j = s.lower() ; j <= s.upper() ; j++ )
      s[j] /= norm;

}
```

Figure 1. Normalizing a Vector

coordination idea from single assignment patterns to arbitrary user-defined patterns. In order to support irregular patterns, explicit switches between phases are possible as well.

3 Distributed Implementation of Coordinators

Coordinators are a means of architecture independent parallel programming as they provide a global object and global access control in the form of access patterns that ensure proper parallel semantics. In this section, we will focus on the issues related to implementing the global object and the global access control for coordinators on distributed memory architectures.

3.1 The Problem

A coordinator must be accessible from every task involved in the parallel computation regardless of the physical location of the task. Therefore, a coordinator must provide an interface to its methods across address space boundaries. In addition, the coordinator must control and synchronize all incoming accesses according to the statically defined access pattern to ensure proper semantics.

Figure 2 shows a situation in which three tasks are involved in the parallel computation. Two of them share an address space; the other is in a different context. Each task accessing the coordinator via method calls must first pass through the coordinator interface which ensures adherence to the access pattern. Then, the method call is forwarded to an object inside the coordinator which actually implements its behavior. In fact, the interface is a mirror image of the user defined object inside the coordinator. Finally, the method call is processed and the return values are delivered to the caller.

Therefore, the following questions arise for the implementation of distributed coordinators:

1. How is the desired access pattern established in a situation where several, potentially distributed, tasks access the (global) coordinator?

2. Where does the user defined object physically reside and how is the method call forwarded to it?

3. Since coordinators are passive objects, which "agent" will process the method call?

The implementation of coordinators in a shared memory environment is straightforward, since all tasks share a common address space. Well known mechanisms – mutex and condition variables – can be used for synchronization. Also, forwarding of the method calls can be achieved by simple method wrappers, since the user object is already in place and the callee itself can process the method.

In a distributed environment, however, the communication between tasks across address space boundaries requires explicit message passing. Distributed locking facilities are rarely available or applicable, and direct method calls are not possible unless automatic task migration is available. Question 1 leads to the need for global access control which will be discussed in the following. The interrelated questions 2 and 3 address the problem of implementing a global shared object which is dealt with in section 3.3.

3.2 Distributed Access Control

The access control component must solve two problems. First, write accesses must be made exclusive so that two tasks will not modify the user object simultaneously. The second problem is that premature accesses must be delayed to meet the statically defined access pattern of the parallel program. For example, write accesses are not allowed to occur in read phases and must be delayed until the next write phase. The same applies to accesses that fit the current access type, yet are in excess of the number of accesses expected for the current phase. Such accesses apparently must be delayed until the next phase of the same type.

In the following, we discuss the concepts *dedicated node*, *token passing*, and *decentralized control* as possible implementations of distributed access control. The first two methods work with one control server which is responsible for coordination, the last scheme works with several cooperating servers.
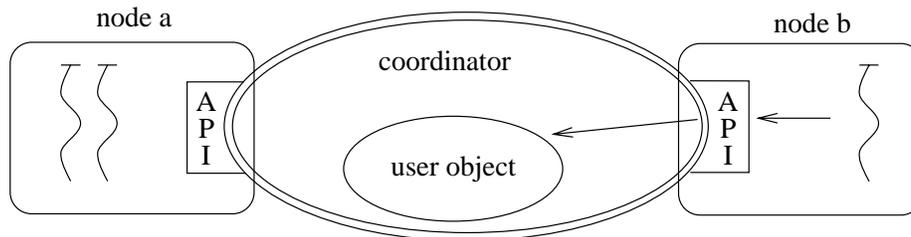
Figure 2. Coordinator Model

3.2.1 Dedicated Node

The concept of the *dedicated node* uses one fixed control server for each coordinator, maintaining it for its whole lifetime. When a task wants to access a given coordinator, it sends a *requireAccess* message to its control server and waits until the control server grants the access with an *allowAccess* message. After the access, the task sends an *accessFinished* message to signal the termination of the access. In general, the messages are sent via a remote communication layer unless shared memory can be used as a shortcut. Note that this implies that the communication layer must support *request-reply* communication patterns between arbitrary tasks.

The main disadvantage of this technique is that the node with the control server may become a bottleneck since all nodes involved in the coordination must communicate with the central server. Therefore, the node of the control server must be carefully chosen. This means that the control servers of all coordinators should achieve good load balancing on all nodes. Moreover, the placement of the control servers should optimize the utilization of shared memory for the communication of tasks with the control server. Another disadvantage of this concept is that each access requires communication, although the access patterns may allow some freedom (see section 3.2.3) which should be used to eliminate some requests. For example, an *each* pattern allows read accesses with no communication because *read* accesses can be performed in parallel without synchronisation. Also, no communication is needed for *each* patterns before the phase switch because each task has to fulfill only local conditions to meet the global switch condition.

On the other hand, a centralized control server is a well known and robust concept which is easy to implement and only needs a simple *request-reply* communication model. Also, the global knowledge of the implementation allows optimizations w.r.t. throughput and response time. For example, accesses of a task with high communication demands can be preferred and the last (pending) access in a phase can be granted before the task actually requests access permission.

3.2.2 Token Passing

The *token passing* scheme generalizes the concept of the *dedicated node*. There is always one control server responsible for a coordinator, but the node where the control server resides varies over the lifetime of the coordinator. The node is identified by a token which can be migrated together with the data needed for access coordination.

The migration of a control server allows dynamic load balancing to minimize bottlenecks and to reduce remote communication between participating tasks and the control server. For example, when there are several tasks on one node, the server can migrate to this node and reside there until all tasks have finished their accesses using shared memory communication with the server. Also, the control server can keep an access history to identify and predict recurrent access patterns to further improve load balancing.

Unfortunately, this scheme may involve significant costs, both in terms of efficiency and implementation effort. Since the location of the control server can change, the tasks must locate the server before each access. If the communication layer offers no cheap broadcast primitive, this can cause a high penalty for each access. Also, the potential bottleneck is not really removed; it simple moves among control nodes. Indeed, the bottleneck can become even worse since there is no control server available during the time the token is in transit. Moreover, the implementation of the *token passing* concept is more difficult since requests in transit must be correctly forwarded to the control server during migration. In the worst case, this can produce a race among forwarded requests and token migrations.

As a consequence, use of the *token passing* method must be considered carefully.

3.2.3 Decentralized Control

The concepts *dedicated node* and *token passing* both suffer from a potential bottleneck constituted by a central control server. Therefore the idea of the *decentralized control* is to use several cooperating control servers. They communicate using the remote communication platform to enforce

the access pattern.

Obviously, decentralized control has no advantage compared to a centralized access control if all control servers are needed for the decision. Indeed, to ensure exclusive write accesses, communication of all servers to realize a distributed mutex or barrier [13, 16] is needed. Also, all access patterns requiring a global counter need communication involving all servers to ensure its proper implementation.

Fortunately, the semantics of coordinator access patterns allow some freedom which can be used to reduce communication between control servers:

- A server can grant access without further communication if the access type is *read* and the access pattern includes no global counters.

- A server can delay granting an access if the access type does not match the current phase of the coordinator. E.g., a *read* access must be delayed in a *write* phase.

- A server can delay granting an access if the requesting task has already fulfilled its access pattern in the current phase.

- A server can delay granting an exclusive access if one of the tasks the server controls already performs an exclusive access. The server may grant the access immediately after the exclusive access is finished.

The *decentralized control* has some advantages in *read* phases but produces a global bottleneck of all servers in *write* phases.

3.2.4 Summary

To conclude, none of the concepts fulfills all demands of the coordinator model, but a combination of the models is promising:

1. Use the *dedicated node* concept to handle exclusive accesses.

2. Take advantage of freedoms in the access pattern whenever possible, especially in *read* phases to reduce the communication with the control server.

3. Use very rarely *token passing* to optimize load balancing. Perform the migration on a phase change. Notify all nodes of the changed location before the beginning of the next phase to avoid the need of forwarding access requests.

3.3   Global Object Representation

A coordinator is a global object; hence, it has to be visible in the address space of every task involved in the parallel computation. However, the location of the user defined object, i.e. the actual data, is hidden from the tasks. An implementation can freely decide where to place the user object as long as an access to a coordinator object is not distinguishable from an access to a local object. On invocation of a coordinator method, a prolog is executed which asks one of the control servers for access permission and then invokes the related method of the user defined object, according to one of the schemes discussed below. After the method call is finished, an epilog is executed which informs the control server of the termination of the access.

There are conceptionally two possible ways to invoke the method of the user defined object. First, the call can be forwarded to the address space of the user defined object and then be processed by another task. This is called *remote method invocation* (or remote procedure call). The other concept provides the object in the address space of the task before the access is performed. Therefore, the method can be processed by the calling task itself. This can either done by *migration* or by *replication* of the user object.

3.3.1   Remote Method Invocation

The concept of *remote method invocation* (RMI) is well known. In the context of coordinators it can be used to implement a transparent access to the user defined object. On invocation of a coordinator method, the prolog is executed. Then, the parameters of the call are packed into a buffer which is sent to the node where the user object resides. The calling task is blocked meanwhile. On arrival, the buffer is received and unpacked. Then, the related method of the user object is invoked. The results of the call are packed into a buffer and sent back to the waiting task which unpacks the results. Subsequently, the epilog is executed and the coordinator method terminates. Since the calling task cannot do the work on a remote node, there must be a special task on the target node which performs the work on behalf of the caller.

Initially, this seems to be a bad scheme since the remote node may become a bottleneck and processing of all method calls is actually not done in parallel but sequentially on one node. However, if the method call consumes less time than the transfer of the user object, it may be worthwhile to employ RMI. Also, RMI provides an update mechanism. That is, instead of the entire user object, only a method's parameters need to be transfered (which are usually much smaller) and the object is modified in place. As an example, consider setting a value of a large matrix; with RMI, this requires the transmission of two integers and one double instead of the

whole matrix. In addition, if the access control server is located on the same node as the user object, the RMI can be piggy-backed onto the permission request, saving the latency of two additional remote communication operations.

Yet, there are some more problems involved with RMI. First, RMI does not provide method calls that are fully equivalent to local calls since the pack and unpack functions need an interface specification. Also, call-by-reference call semantics must be simulated by call-by-value and copy-back. Finally, the RMI concept is very difficult to implement when an additional coordinator is given as a parameter of the method invocation. The reason is that if the user objects of both coordinators are not located on the same node it is not possible to process the method call directly. As a consequence one of the coordinators must be migrated or replicated, so that both coordinators are located on the same node again.

3.3.2 Migration

The key idea of *migration* is that, if a task tries to access the coordinator and if the user object is located on another node, the user object is transfered to the node of the caller. The access of the task is delayed until the user object arrives at the node so that the method can be processed locally. Since the method call is processed in the environment of the calling task, it is equivalent to a local access except for the delay for access control and migration.

After the method call terminates, the user object can be migrated to another node if needed. If it migrates immediately back to its former node, the *migration* has *call-by-visit* semantics. If the object stays in place until another task accesses the user object, causing it to migrate, then the *migration* has *call-by-move* semantics. In general, call-by-visit semantics is easier to implement since all nodes can request the user object from one (known) node. Using call-by-move semantics implies that each node must locate the user object to initiate the migration. This costs at least one additional message, e.g. to forward a migration request. However, in most cases call-by-move will cause less network traffic than call-by-visit semantics (see section 4.2).

Although the concept of *migration* is well suited for write accesses it has disadvantages for read accesses: there is always only a single user object and only the tasks that share a common address space with the user object can access it concurrently. Tasks on other nodes have to wait until the object has been migrated to their node. This means that read accesses are strictly sequentialized if we assume one task per node.

3.3.3 Replication

Both concepts, *remote method invocation* and *migration*, do not support parallelism very well. This is acceptable in write phases, but definitely not in read phases. With *replication*, it is possible to have an instance of the user object on each node so that parallel read accesses become possible. However, having multiple copies of one object causes consistency problems [12] in write phases which are well known from distributed shared memory platforms [2, 9]. This implies that write accesses are more expensive since additional messages are needed to ensure consistency of all copies. Since the coordinator model permits only exclusive write accesses, it is not reasonable to provide, and maintain consistency of, multiple copies of the user object during *write* phases.

In a *read* phase however, *replication* is recommended. When a coordinator enters the read phase, the control server can send a copy of the user object to each node involved in the parallel computation. This is acceptable if a cheap broadcast primitive is available or if the access pattern is regular, e.g. an *each* pattern. If not, this may block the control server an unreasonable amount of time. Moreover, if the pattern is irregular, not all nodes involved in the parallel computation will actually access the coordinator in this phase, so that some of the messages will have been wasted. If the pattern is irregular, it is therefore better to use *replication on demand*, i.e. the copy of the user object is sent to the task the first time it accesses the coordinator.

3.3.4 Summary

Again, a combination of the alternatives seems advantageous:

1. Use *replication on demand* in *read* phases. If a cheap broadcast primitive is available, replicate the user object on a phase change.

2. Use *migration* in *write* phases.

3. Use *remote method invocation* in write phases to replace *migration* if the parameters are small in comparison to the user object, or if the method executes in a short amount of time.

3.4 Implementation

The implementation of coordinators on distributed memory architectures is based on the existing implementation on shared memory platforms [11]. The implementation includes a C++ preprocessor and a class library. The preprocessor reads a program which uses coordinators, adds calls to the class library that implements the access control, and generates an ordinary C++ program.

The distributed version enhances the shared memory version with remote communication. Hence, it is able to work

in hybrid configurations with several SMPs connected together. In [18] we have examined several remote communication platforms: *Active Messages* [17, 14], *Nexus* [6], *Chant* [7], *TPVM* [4]. The distributed coordinator implementation has been decided to use *Nexus* since this package supports both multithreaded shared memory and distributed memory applications on heterogeneous platforms.

The implementation uses the *dedicated node* concept for access control. For the global object representation, *migration* is used in *write* phases and *replication on demand* in *read* phases. The user can decide between the two different variants *call-by-move* and *call-by-visit* of the *migration* concept. Also, the user may use *remote method invocation* on top of the other concepts. The variants are specified at compile time by C++ `defines`.

Whenever an object is sent to another node, overloaded functions are called to marshal and unmarshal the object. The functions have to be written by the user using a basic set of functions provided by the implementation. Support for automatic marshaling [8] however is not yet available.

We have tested the implementation on a cluster of SUN workstations with UltraSparc and SuperSparc processors. The results of the tests are presented in the next section.

4 Experimental Results

In this section, we present results obtained with the implementation of coordinators on a cluster of four SUN workstations with 170 MHz UltraSparc CPUs connected with 10 Mbit/s Ethernet. The hit statistics in figure 5 were obtained with three additional SUN workstations with SuperSparc CPUs. The benchmark program for the experiments is outlined in figure 3. It was compiled with optimizations enabled. The distributed coordinator implementation uses the optimized versions of the *Nexus* library routines.

During the access time measurements, the benchmark program was executed by one task per node and the amount of work performed by each task between consecutive accesses was reduced to one floating point operation. The number of accesses of the access type to be measured, was increased to a value high enough to yield reasonable time measurements. Only a single access of the second access type was performed.

4.1 Remote Access Times

Figure 4 shows the average times (in milliseconds) of *remote* coordinator accesses from any of the involved remote nodes, as a function of the size of the user defined object. The *dedicated node* which hosts the coordinator is excluded from these measurements because it would use the local shared memory for communication. The graphs depict access times for remote *read (RD)* accesses, *write* ac-

```
void f(int noWrites, int noReads, int workAmount)
{

    static coordinator<object>
     d(write(each(gr,noWrites)),
        read(each(gr,noReads)));

    for( i=0 ; i < noWrites ; i++ )
    {
        doCalc(workAmount);
        doWriteAccess(d);
    };
    for( i=0 ; i < noReads ; i++ )
    {
        doCalc(workAmount);
        doReadAccess(d);
    };

}
```

Figure 3. Outline of the Benchmark Program

cesses using *call-by move (CBM)* and *call-by-visit (CBV)*, and *write* accesses using *remote method invocation (RMI)*. In case of RMI, the access times are given as a function of the size of the RMI parameters since the user object itself is not moved. The number of nodes (processes) involved is shown in brackets. We measured on a cluster with two, three and four nodes.

The average remote read and write access times for a very small user object are about three milliseconds for all methods on a two node cluster; the times increase with increasing object size.

A closer analysis of the access times reveals that the disappointing times result from the low performance of *Nexus* on our configuration. Using three nodes, the average access times are smaller because the multithreaded implementation of the coordinator communication layer with *threaded remote service requests* is able to hide more of the network latencies. The access times on four nodes (not shown in the figures) are similar to the three node cluster.

For very small object/parameter sizes, RMI turns out to be the best method to implement *write* accesses. RMI is better in this case because it saves two remote messages since the access permission is requested and granted directly on the target node, which coincides with the *dedicated node*. However, if the parameters of the method calls are bigger than 300 double words, RMI gets worse than migration with call-by-move which still has an average access time of about three milliseconds due to the smaller increase in the access times then. Notice however, that RMI tends to move smaller amounts of data (as parameters) than migration, which ships an entire object.
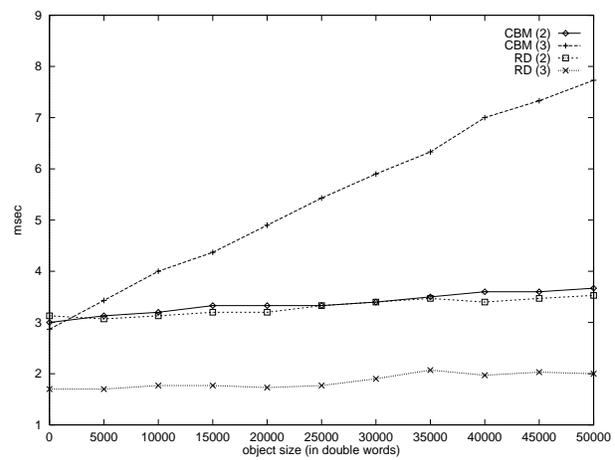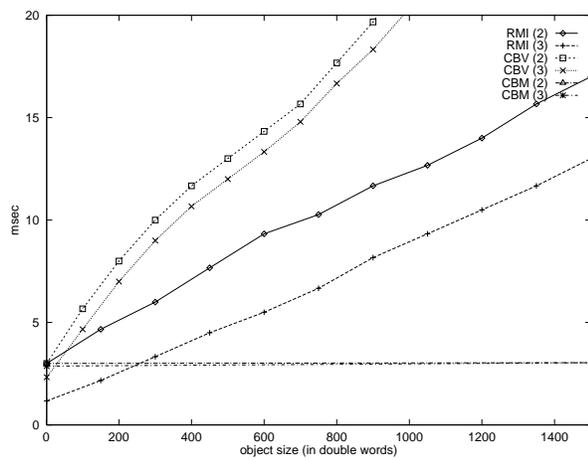
The average access times of *read* accesses (RD) are not

Figure 4. Remote Access Times

very high since the user object is sent only once in a phase due to the implementation of *replication on demand*. Only request and grant messages are needed for each access. However, each read access *does* require communication.

Some preliminary tests with numerical applications (e.g. *Jacobi* method) indicate that good speedups are only achievable if the amount of computation between two read accesses is high. Of course, the same applies to write accesses, but the problem with read accesses could be avoided with a *decentralized control*.

4.2   Analysis of Migration Schemes

The average access times of *write* accesses with migration increases with increasing object size regardless of the implementation variant. However, the average access times of *call-by-visit migration* grows more rapidly then those of *call-by-move migration*. So, call-by-visit is only useful for very small object sizes. Indeed, if the object size is very small and many nodes are used, call-by-visit is faster then call-by-move because no forwarding messages are needed; a small object is always sent without costs, piggy-backed with the access grant.

For larger user objects, however, the *dedicated node* becomes a bottleneck. Using call-by-visit, the user object is migrated twice per remote access. In general, there are $2 * (n - 1)/n * no\_accesses$ object migrations needed in an n-node cluster. With call-by-move, however, the object is sent only when the object is not already on the requesting node. In the worst case $no\_accesses$ object migration are needed. Whenever a write access occurs, there are five possible situations:

1. The object is located on the *dedicated node* and a task on this node tries to access the coordinator. Since the

user object is in place, no migration is needed (*local hit*).

2. The object is located on a remote node, i.e., not the *dedicated node*, and a task on that node tries to access the coordinator. Again, no migration is needed (*remote hit*).

3. The object is located on the *dedicated node* and a task on a remote node wants to access the coordinator. The object must be migrated to the remote node (*send*).

4. The object is located on a remote node and a task on the *dedicated node* accesses the coordinator. So, the object must be migrated back to the *dedicated node* (*copyback*).

5. The object is located on a remote node and a task on another remote node accesses the coordinator. Hence, the user object must be migrated (*forward*).

The distribution of these alternatives for various system sizes (number of nodes) is shown in figure 5. In the experiment, each node hosts one task and all tasks take part in a regular *each* access pattern. The curves for *copyback* (not shown in the figure) and *send* are always identical because the object is located on the *dedicated node* in the beginning of a phase and is migrated back at the end of a phase. Unfortunately, the ratio of the forwarding messages clearly increases with the amount of nodes so that call-by-move will become more expensive. Figure 4 on the right side illustrates this behaviour since the access times for three nodes increase more rapidly than those for two nodes. Nevertheless, the amount of object migrations will always be smaller using call-by-move than using call-by-visit semantics.
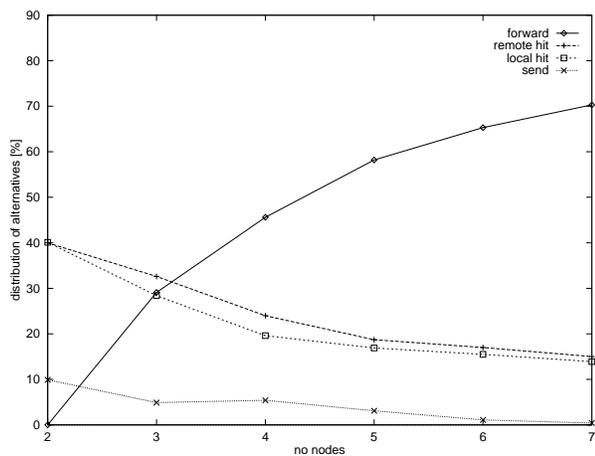
Figure 5. Call-by-Move Migration Statistics

5   Conclusions

We have presented several different variants to implement a global object abstraction and global access control on distributed memory platforms while studying coordinators. We have discussed the concepts *dedicated node*, *token passing* and *decentralized control* as possible implementations of a distributed access control. We have shown benefits and drawbacks of each method. The discussion has shown that only a combination of all methods can satisfy the requirements of a coordinator implementation. We have also discussed *remote method invocation*, *migration* and *replication* as implementation variants for global objects. Similarly, none of the methods satisfies all requirements, but a combination of all of them is promising.

We have also presented our implementation on SUN workstation clusters using *Nexus* as the remote communication platform. The implementation is based on the *dedicated node* concept for access control and uses a combination of all of the discussed global object implementation variants. The results show that there is no single best global object implementation variant. Indeed, this depends highly on the application. If the size of the parameters of a coordinator method call is small, RMI may be a good solution. However, as the size of the parameters grows, call-by-move migration should be preferred.

Our results also show that a centralized access control results in suboptimal performance for read phases. Hence, decentralized elements are highly recommended and should be considered for implementation in the future.

References

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[2] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming With Entry Consistency for Distributed Memory Multiprocessors. Technical Report CS-91-170, Carnegie Mellon University, September 1991.

[3] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing'93*, November 1993.

[4] A. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. *Proc. Fourth IEEE International Symposium on High Performance Distributed Computing*, 1994.

[5] C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimizations. *ACM Principles of Programming Languages*, 1995.

[6] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computing Systems. Technical report, Argonne National Laboratory, January 1996.

[7] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of chant: A talking thread package. In *Proceedings of Supercomputer 94*, pages 350–359, Washington, D.C., November 1994.

[8] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems 4*, 1982.

[9] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–131, January 1994.

[10] J. Knopp. High Level Parallel Programming based on Automatic Coordination. In *Proc. EUROPAR, Lyon*, August 1996.

[11] J. Knopp and M. Reich. A Data Model For Architecture Independent Parallel Programming. In *Workshop on High-Level Programming Models and Supportive Environments*, held in conjunction with the IEEE International Parallel Processing Symposium, Hawaii, 1996. IEEE Conputer Press.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[13] Mamoru Maekawa. A $\sqrt{n}$ algorithm for mutal exclusion in decentralized systems. In *ACM Transactions on Computer Systems*, May 1985.

[14] A. Mainwaring and D. Culler. Active Messages: Organization and Applications Programming Interface. Technical report, University of California at Berkeley, Computer Science Department, 1995.

[15] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, Summer 1996.

[16] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. IEEE Computer Society Press.

[18] Christian Weiß. Realisierung von Koordinatoren auf verteilten Parallelrechnern und Rechnernetzen. Master's thesis, TU München, October 1996.

[19] G.V. Wilson and P. Lu, editors. *Parallel Programming Using C++*. MIT Press, Cambridge, Massachusetts, and London, England, 1996.