The Shared Objects Net-interconnected Computer (SONiC)

Andreas Polze and Miroslaw Malek
Humboldt-University of Berlin
Department of Computer Science
Unter den Linden 6

10099 Berlin, GERMANY

e-mail: {apolze,malek]@informatik.hu-berlin.de

ABSTRACT

We present a novel approach to parallel computing in a workstation environment. We introduce *Replicated Shared Objects* as a programming paradigm which allows encapsulation of communication and synchronization operations within an object's implementation. It also provides an easy-to-use interface to the application programmer. *Shared Objects Memory* implements our programming paradigm as an object-based, replicated distributed shared memory system on top of network-transparent communication in the Mach operating system. Together with a *Remote Execution Service* it provides a complete environment for execution of shared-memory parallel programs in workstation clusters. Our system allows for definition and configuration of a virtual parallel machine called ‘‘Shared Objects Net-interconnected Computer’’ within such a cluster through an easy-to-use graphical interface.

Besides a description of our programming paradigm and the ‘‘Shared Objects Net-interconnected Computer’’ this paper includes promising runtime statistics from early experiments, showing that a cluster of four low-end workstations may have comparable performance with a four-processor shared memory multicomputer such as Sequent Balance.

Key words: *Shared Objects Net-interconnected Computer*, *distributed computing*, *memory management*, *object replication*.

1. Introduction and Motivation

Multi-threaded applications are becoming increasingly popular and one of the most prominent examples is the WWW browser “`netscape`” which shows much better interactive response than competitors’ due to its multi-threaded nature. Many more examples can be found in the context of the NeXTSTEP operating system. Obviously, application programmers are willing to undertake the extra effort of multi-threaded programming — dealing with critical sections, locks, barriers, and condition variables — for gaining better performance and interactive behavior of an application. As examples show, this effort tends to pay off.

However, much higher performance gains are possible if an application could be executed in multiple threads spread over a cluster of workstations, instead of running on a single one. For making an application programmer accept truly parallel programming techniques, these techniques have to be usable (at least) as easy as a threading package. Thus, a distributed threading package together with variables shared among different address spaces and an easily configurable distributed runtime system can be appealing to the programmers of performance hungry applications.

Contemporary distributed computing systems have a potential of providing high computing power and — with upcoming new networking technologies — fast communication links. However, until now network-based parallel systems which employ interconnected computers (PC’s, workstations, mainframes) as processing elements are not widely used. Poor programmability seems to be the main reason which inhibits widespread use of those systems for parallel computing.

To make parallel programs run in a distributed environment, three issues have to be addressed:

- initiation and termination of parallel activities (threads)
- communication between parallel threads
- synchronization among parallel threads.

In our approach, the *Remote Execution Service* addresses issue number one, whereas issues two and three are considered by the *Shared Objects Memory* system. Both components extended by a graphical front end form the *Shared Objects Net-interconnected Computer*.

The remainder of the paper is organized as follows: Section 2 describes shared objects as programming paradigm and presents two examples. Section 3 deals with implementation issues and describes our environment. In Section 4 we present some early results: a parallel prime number checker which actually runs faster in a distributed workstation environment than on a shared-memory multiprocessor. Section 5 describes related work and in Section 6 we finally draw our conclusions.

2. Shared Objects: Encapsulation of Communication and Synchronization

Message-passing and shared-memory are two major paradigms which are commonly used for parallel programming. In the message-passing programming model processes exchange information by messages. But the absence of global data is inconsistent with existing sequential programming styles. Writing programs with explicit message-passing routines has shown to be error prone and tedious. In contrast, the shared-memory model provides transparent data communication. Thus it is similar to the conventional programming style and seems to be more familiar to programmers.

Distributed-shared-memory systems — implemented either in software or hardware — provide the shared-memory programming model to the user. However, message-passing is internally used to support the vision of shared data. Speed of inter-node communication, data consistency model and the size of shared data chunks are main factors for remote memory access latency.

Classical distributed-shared-memory systems ([1] present an overview) rely on equally sized memory pages as units of sharing. Often those pages are relatively large chunks of data. Memory pages are not related to the concept of accessing memory through variables as supported by programming languages. Thus, the problem of false sharing may occur — several logically independent variables may reside in the same memory page and are treated like a single, big shared variable.

Weakly consistent data models can dramatically influence the amount of communication needed for execution of a parallel program and for keeping the illusion of a shared dataspace alive, a fact which is especially important in distributed environments. However, to make weakly consistent models work the programmer has to include ‘‘hints’’ to the memory management in his software. Synchronization variables have to be explicitly assigned to (sets of) shared data items. Operations on those variables like *acquire* and *release* have to be placed around accesses to shared data items.

Objects as implementation of abstract data types — which integrate a type with operations and equations on this type — have been established as a programming concept in a number of sequential languages. We believe that objects are the right place to integrate shared data items and the synchronization variables necessary to protect them as well. Our approach provides C++ classes whose instances can be used as simple as ordinary C++ objects in a sequential program. However, we provide a runtime system which treats objects in different address spaces as replicas. Thus, those objects can be seen as shared objects. Weakly consistent memory models are used to update all the replicas which make up a particular shared object.

Within ‘‘Shared Objects Memory’’ the programmer has full control over shared objects. He can use C++ language constructs to describe size and layout of objects — the units of sharing. Simply by using different base classes when instantiating an object the programmer can determine which memory consistency protocol should be used for managing a particular object. No false sharing of data can occur, unnecessary communication is avoided.

Consistency Models

Maintaining a predictable view of memory across machines is called memory consistency or memory coherency. If a memory management system implements full consistency, data modified by one processor will be immediately visible to all other processors sharing the memory. Alternately, memory management may implement weaker consistency, in which updates are deferred until absolutely needed or until triggered by a special mechanism.

Within a DSM-system replication allows for efficient data accesses in case of read sharing. However, this approach raises the *cache consistency problem*. Programmers often assume that memory is sequentially consistent [2]. But even in sequentially consistent systems, explicit synchronization is required for complex operations. So, most parallel programs define their own higher-level consistency requirements.

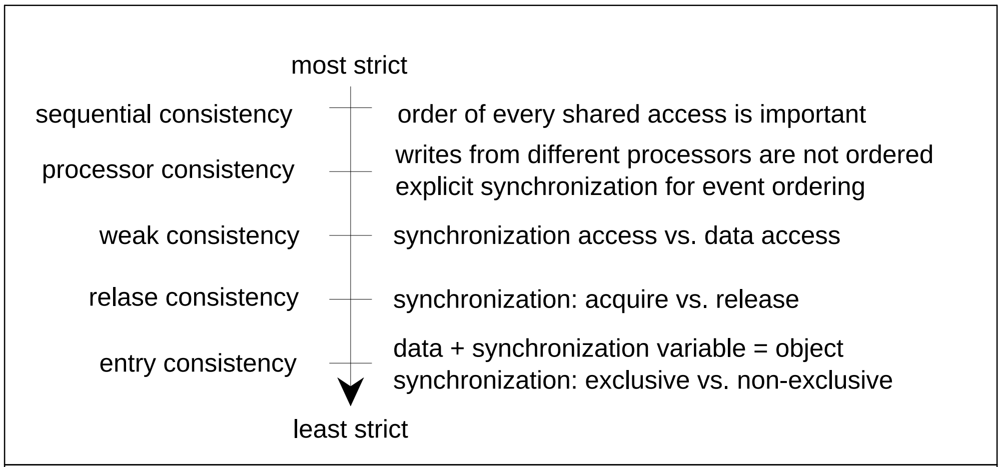

most strict
sequential consistency
order of every shared access is important
processor consistency
writes from different processors are not ordered
explicit synchronization for event ordering
weak consistency
synchronization access vs. data access
relase consistency
synchronization: acquire vs. release
entry consistency
data + synchronization variable = object
synchronization: exclusive vs. non-exclusive
least strict

Fig. 1*: Consistency Spectrum*

These observations have led to a class of weakly consistent protocols. Among them are processor consistency [3], weak consistency [4], release consistency [5] and entry consistency [6]. Such protocols distinguish between data accesses and synchronization accesses. The only accesses that must execute in sequentially consistent order are those relating to synchronization. Updates to shared data in distributed memories can be handled asynchronously. Fig. 1 illustrates how some consistency models are related to each other.

In the ‘‘Shared Objects Memory’’-system we support release consistency and entry consistency for shared objects. An association between data and synchronization variables is made on the basis of objects — each shared object is implicitly tied to a synchronization variable.

Two Example Classes

Our programming approach provides a whole hierarchy of classes whose instances are replicated shared objects. The base class `SOM` implements management functions for shared objects. Within this class code exists for communication between all the instances — distributed over different address spaces — which make up a shared object in our system. Class `SOM` ties a shared synchronization variable to each data object and provides functions for sending update or invalidation messages to an object's set of replicas. By interaction with the ‘‘object repository’’ (a component of the ‘‘Shared Objects Memory’’-runtime system) class `SOM` maintains the notion of read- and write-access rights to an object.

The classes `ENTRY_CONS` and `RELEASE_CONS` are derived from class `SOM`. Both classes implement weakly consistent memory protocols and provide the public member functions `acquire_write_lock`, `acquire_read_lock`, and `release_lock`. The programmer who implements a class of shared objects can use these functions to initiate synchronization operations where necessary. By simply choosing the appropriate base class the programmer can decide which consistency protocol his objects should be based on. Fig. 2 shows the part of the class hierarchy discussed in this paper.

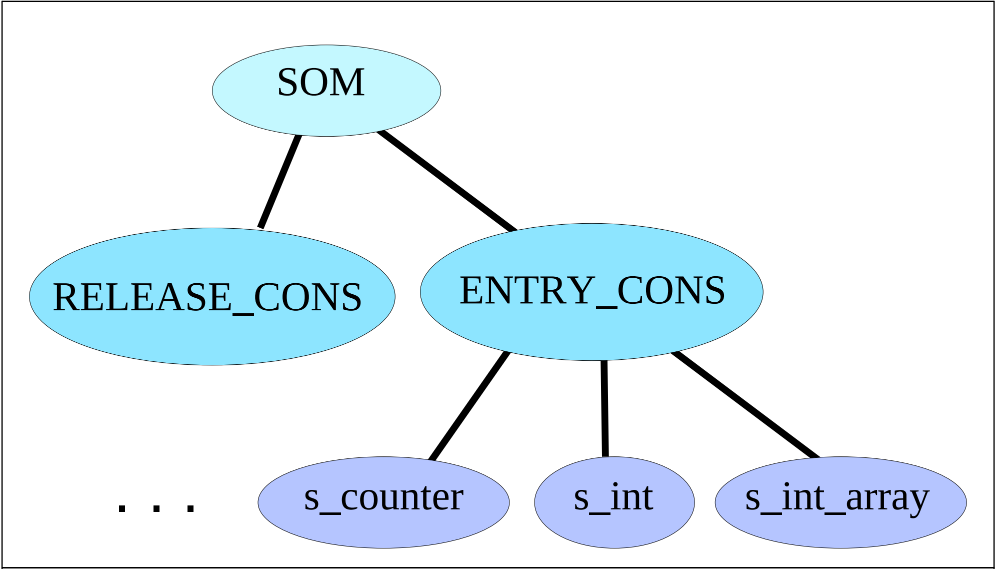


Fig. 2: *Class Hierarchy for Shared Objects*

Let us discuss classes `s_int` and `s_int_array` which implement shared integers and shared arrays of integers, respectively, in some detail. Both classes are derived from class `ENTRY_CONS`, thus the entry consistency protocol is used for maintaining consistency among instances of these classes.

From the class designers point of view implementation of a shared C++ class is simple. Besides identifying public, protected, and private member function as necessary when creating any C++ class, the designer has to decide which member functions act as write operations and therefore need exclusive access to an object and which functions are read operations, requiring non-exclusive access. Whereas a call to `acquire_write-_lock` has to be inserted at the beginning of the implementation of each write operation, each read operation should start with `acquire_read_lock`. Both types of operations have to be terminated by a call to `release_lock`. The programmer of a class of shared objects has no need to care about the actual details of copying object values from one address space to another. All those details are hidden in the implementation of classes `ENTRY_CONS` and `RELEASE_CONS`.

A new level of abstraction is reached if we look at shared objects from the application programmers point of view. From that point of view shared objects appear just like ordinary sequential C++ objects. The programmer only has to watch out for the appropriate public member functions which allow him to access a shared object. Since the implementation of those functions hides all synchronization-specific code, the programmer can treat each shared object as sequential consistent although weakly consistent memory protocols are used for its implementation.

In Fig. 3 we show the definition of class `s_int`. Besides a private data component making up the actual integer value class `s_int` provides a number of access functions. An assignment operator allows to assign ordinary integer values to an `s_int` object whereas the typecast-operator gives the opportunity to turn an `s_int` back into an integer object. Thus, objects of class `s_int` can be used as easily as ordinary integer variables within a C++ program.

```
# include "entry_cons.h"

class s_int : public ENTRY_CONS {
        int _val;

        virtual int size_of() { return sizeof( *this ); }
        virtual void* assign( void* p ) { *this = *((s_int*) p); }
public:
        s_int( int val = 0 );
        virtual ˜s_int();

        operator int();
        s_int & operator=( int s );
        s_int & operator=( s_int & s );
};
```

Fig. 3*: Definition of class s_int*

Besides the public access functions already mentioned class `s_int` contains two virtual functions `size_of()` and `assign()` which are used by code in class `SOM` to copy shared objects from one address space to another. By deriving class `s_int` from class `ENTRY_CONS` we ensure that the entry consistency protocol is used to keep all the replicas making up a shared object consistent. Synchronization functions are inherited from `ENTRY_CONS`. Thus, we can implement the typecast and assignment operations for class `s_int` as shown in Fig. 4 .

So far, one detail has been omitted from our discussion of shared objects. Within the ‘‘Shared Objects Memory’’ each shared object is identified by an unique integer identifier. All the replicas of a particular object carry the same identifier. At runtime, each object registers with the object repository, using its itentifier. Therefore, the repository can maintain associations of objects — copies which make up a shared object.

```
s_int::operator int() {
        acquire_read_lock();
        release_lock();
        return _val;
}

s_int & s_int::operator=( int s ) {
        acquire_write_lock();
        _val = s;
        release_lock();
        return * this;
}
```

Fig. 4*: Two operations of class s_int*

Somehow these unique identifiers have to be assigned to the shared objects. We use ‘‘Single Program Multiple Data (SPMD)’’ parallel computing paradigm which makes assignment of object identifier simple. Since the same program is running in several tasks, objects are created in the same order in each task. Therefore we can use a global counter per task to give each object an identifier corresponding to its occurence. Although this method is implemented as default for assigning identifiers to shared objects in our system, the programmer can override it and explicitly assign such identifiers. In the future, a preprocessor to the C++ compiler may generate object identifiers based on source code analysis.

Let us now discuss class `s_int_array`, which implements a shared array of integers. In contrast to `s_int` objects, which are equally sized, the size of instances of class `s_int_array` may vary. This is accomplished by an overloaded version of operator `new`. `s_int_array` objects have to be created dynamically and an object’s size has to be specified during creation. Synchronization for all the data items contained in a `s_int_array` object is handled once, the object is copied as a whole between address spaces when necessary. Using a `s_int_array` is therefore more efficient than using an array of shared integers (`s_int` objects). However, accesses to the `s_int_array` restrict the potential parallelism much more than accesses to an array of shared integers would.

```
class s_int_array : public ENTRY_CONS {
        static _curr_elem;
        int _nelem;
        int a[0];
        virtual int size_of() {
                 return sizeof( *this ) + _nelem * sizeof( int ); }
        virtual void* assign( void* p )
                                   { *this = *((s_int_array*) p); }
public:
        s_int_array(int objID = -1);
        ˜s_int_array();
        void * operator new(size_t siz)
                                 { return operator new( siz, 0 ); }
        void * operator new(size_t siz, int num_elem);
        void operator delete(void * o);
        s_int_array & operator=(s_int_array & s);
        int get( int pos );
        int put( int pos, int val );
};
```

Fig. 5: *Definition of class s_int_array*

In Fig. 5 we show the definition of class `s_int_array`. Besides the overloaded versions of operators `new` and `delete` the class provides two access functions `get` and `put`. Using the C++ technique of operator overloading once more, based on these functions one could provide overloaded array operators for the `s_int_array` class. Then, the shared class could be used exactly the same way like ordinary C++ arrays.

Container classes for co-located objects

As demonstrated with class `s_int`, user-defined shared objects may be implemented as instances of a class derived from `ENTRY_CONS` or `RELEASE_CONS`. A shared object is realized as a set of replicated C++ objects in different Mach-tasks. A weak consistency protocol is used to maintain a consistent view on shared data. With each of the replicas making up a shared object, a mutex-structure is associated to allow for implementation of the consistency protocol.

User-defined objects are often small, typically only a few bytes. Often those objects can be grouped into sets — like all the objects in a linked list or all the entries in a array. It seems to be inefficient to perform all the consistency-related operations for each object in such a set separately. With class `s_int_array` we have discussed how sets of related objects can be managed as a whole. However, the association among all the elements within a shared integer array can be established only at creation of the object and cannot be changed afterwards. In certain cases (e.g., when dealing with dynamic structures like linked lists) it seems to be useful to establish associations between objects dynamically. Now, we want to discuss how co-locations — associations among objects — can be managed dynamically.

Traditional distributed shared memory systems are page-based. Several data items are placed on the same page by the compiler — co-locations between data items are established implicitly. Read and write access rights are maintained once for all items on a page. This is more efficient than managing each data item separately — however, it raises the problem of false sharing of non-related items.

Within our *Shared Objects Memory*-system we allow the programmer to explicitly establish co-locations between objects. With container class `DYNAMIC_MEM` our system provides special, overloaded versions of C++ operators `new` and `delete`. Each instance of class `DYNAMIC_MEM` is itself a shared object of a particular, user-defined size. This object simply provides dynamic memory for the creation of smaller, co-located objects of arbitrary classes via the overloaded operation `new`.

Only one mutex structure is required for each `DYNAMIC_MEM` object to synchronize accesses to all the objects co-located on the particular `DYNAMIC_MEM` object. All the co-located objects are transferred as a big data chunk whenever the current value of one of the objects has to be obtained. Thus, traversal of a linked list of co-located objects causes transmission of a single shared object over the network instead of separate transmission of each of the objects within the list.

Our container class `DYNAMIC_MEM` serves a similar purpose like pages in a classical distributed shared memory system. However, in contrast to shared memory pages, with our approach the programmer has full control over co-location of objects on `DYNAMIC_MEM` objects. No false sharing of data may occur.

3. Implementation Issues: Our Environment

With our framework for parallel programming we focus on a number of interconnected workstations, PC's or mainframes as target environment. Co-operating computers can deliver all the support needed for parallel computations: processors capable of high sustained floating-point performance, networks with bandwidth that scales with the number of processors, parallel file I/O, and low overhead communication. With our programming environment we want to make parallel programming on distributed workstations as simple as multi-threaded programming on a single computer and therefore encourage programmers to make free computing power available to their applications.

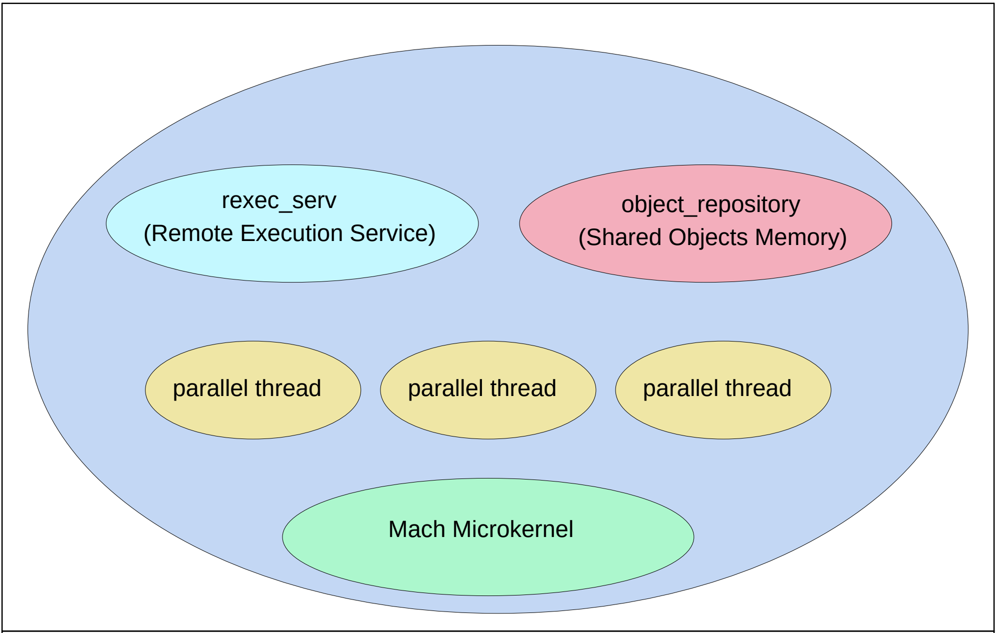


Fig. 6*: Node of the Shared Objects Net-interconnected Computer*

In our model a number of interconnected computing units forms a ‘‘Shared Objects Net-interconnected Computer’’. On each of its nodes several parallel tasks may run. Those tasks communicate using the shared-memory model. The ‘‘Shared Objects Memory’’, our memory management system, is based on network-transparent communication services as provided by the underlying Mach microkernel operating system. The ‘‘Remote Execution Service’’ allows for initiation and termination of parallel activities on every node of a virtual parallel machine, again, based on Mach’s network-transparent communication. It allows the programmer to write parallel programs following the fork-and-join style of parallel execution. Functions of both, the ‘‘Shared Objects Memory’’ and the ‘‘Remote Execution Service’’ runtime systems are accessible by programs through the parallel programming library already described. Here, we focus in the runtime system and show in Fig. 6 structure of a computing node in a ‘‘Shared Objects Net-interconnected Computer’’.

Interactions within the Shared Objects Memory

Besides a number of parallel threads, Fig. 6 shows an instance of the object repository running on a node. The repository serves a similar purpose like page managers in traditional distributed shared memory systems. Our current version of the repository is centralized on a per class base. Therefore for each class exactly one repository is run in the ‘‘Shared Objects Net-interconnected Computer’’.

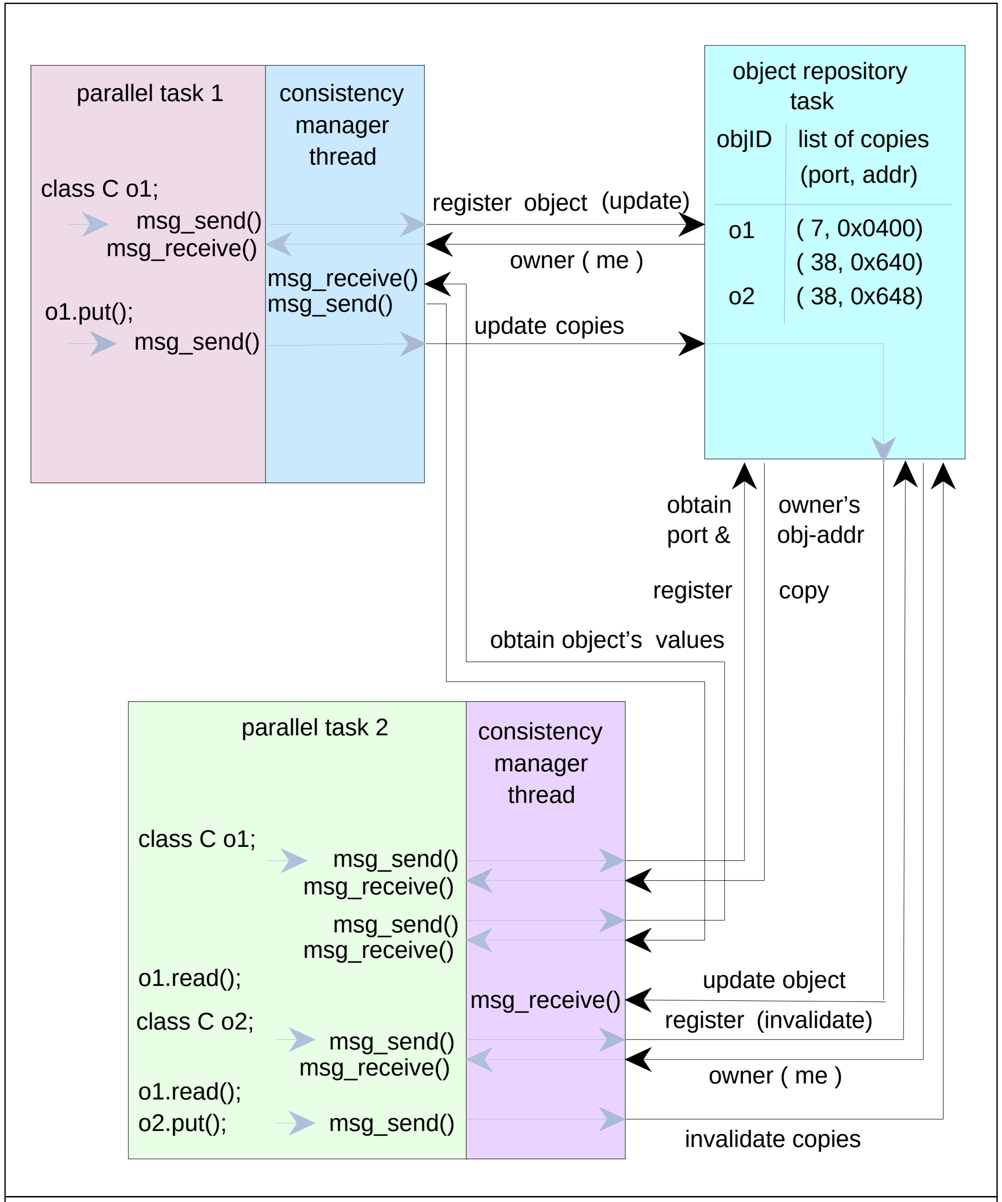


Fig. 7: *Shared Objects Memory — Architecture*

In Fig. 7 we show two parallel tasks running on top of the ‘‘Shared Objects Memory’’ (SOM). Both interact with a third component: the object repository task. The repository contains information about the locations of object's replicas. In addition to threads belonging to a parallel program, each parallel task has a consistency manager thread. This thread is implicitly created and managed by the SOM runtime.

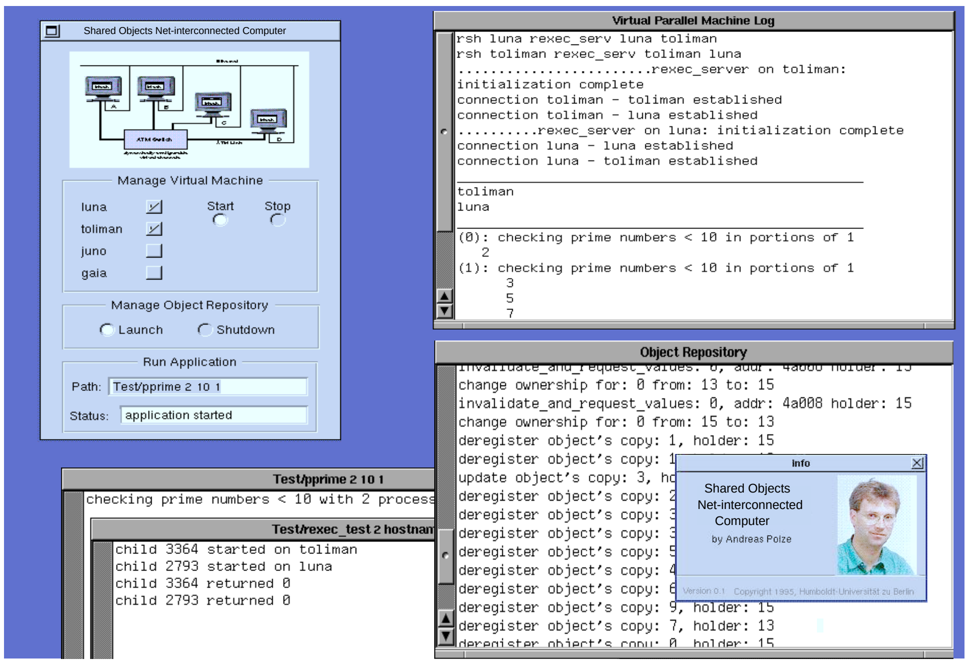
In Fig. 7 both parallel tasks create a C++ Object `o1`. The second tasks additionally creates an Object `o2`. These objects are instances of class C which has to be a subclass of class `SOM`, the ‘‘Shared Objects Memory’’-base class. After creation of an object each task (e.g., the object's constructor) registers the object with the object repository task. This task is accessible through the Mach `netmsgsever` under a well-known name. The repository maintains a table of object identifiers. For each object identifier a list of references to the object's copies is stored. Each entry within this list consists of a Mach port and an address. The port denotes a Mach task whereas the corresponding address specifies the location of an object's copy in the particular Mach task. A task which registers a newly created object receives a reply: the port and object address of the object's owner. A task which registers a newly created object receives a reply: the port and object address of the object's owner. In Fig. 7 Task 2 figures out that there is already an owner for Object `o1` registered with the repository. So it sends an additional message to obtain Object `o1`'s values from the other task.

Once an object has been created and registered with the repository the `SOM` base class provides two member functions to either update or invalidate all copies of the object. Both functions result in sending messages to other tasks. Those messages are received by the peer task's consistency manager thread. To avoid object inconsistencies due to consistency manager operations a mutex-structure coordinates local accesses to `SOM`-objects.

Class `SOM` does not associate any policy with its update and invalidate operations. However, `SOM`'s update and invalidate member functions can be used to implement a particular consistency protocol. For example, class `ENTRY_CONS` uses those member functions to implement the entry consistency protocol.

The Graphical User Interface

In order to make parallel programming in a distributed environment as simple and as usable as multi-threaded programming, a programming environment has not only to care for the application programmer but also for the users of parallel applications. Although we have simplified some details when discussing the runtime structure of our ‘‘Shared Objects Net-interconnected Computer’’ it has shown, that our system consists of a remarkable number of tasks spread all over the distributed workstations. To simplify management of all these tasks we have built a graphical front end to the ‘‘Shared Objects Net-interconnected Computer’’. Fig. 8 shows a screendump from one run of our system.


Shared Objects Net-interconnected Computer
Manage Virtual Machine
Start
Stop
Manage Object Repository
Launch
Shutdown
Run Application
Virtual Parallel Machine Log
Object Repository
Info
Shared Objects Net-interconnected Computer
by Andreas Polze
Version 0.1 Copyright 1995, Humboldt-Universität zu Berlin

luna [✓]
toliman [✓]
juno [ ]
gaia [ ]

Path: Test/pprime 2 10 1
Status: application started

```
rsh luna rexec_serv luna toliman
rsh toliman rexec_serv toliman luna
........................rexec_server on toliman:
initialization complete
connection toliman - toliman established
connection toliman - luna established
..........rexec_server on luna: initialization complete
connection luna - luna established
connection luna - toliman established
_____________________________________________
toliman
luna
_____________________________________________
(0): checking prime numbers < 10 in portions of 1
   2
(1): checking prime numbers < 10 in portions of 1
      3
      5
      7
```

```
invalidate_and_request_values: 0, addr: 4a000 holder: 15
change ownership for: 0 from: 13 to: 15
invalidate_and_request_values: 0, addr: 4a008 holder: 15
change ownership for: 0 from: 15 to: 13
deregister object's copy: 1, holder: 15
deregister object's copy: 1
update object's copy: 3, ho
deregister object's copy: 2
deregister object's copy: 3
deregister object's copy: 3
deregister object's copy: 5
deregister object's copy: 4
deregister object's copy: 6
deregister object's copy: 9, holder: 15
deregister object's copy: 7, holder: 13
deregister object's copy: 0, holder: 15
```

Test/pprime 2 10 1

```
checking prime numbers < 10 with 2 process
```

Test/rexec_test 2 hostnam

```
child 3364 started on toliman
child 2793 started on luna
child 3364 returned 0
child 2793 returned 0
```

Fig. 8: *Screendump showing the "Shared Objects Net-interconnected Computer's front end*

Currently this front end serves three different purposes:

- It allows to choose a number of computers for running the "Shared Objects Net-interconnected Computer" one, i.e., it serves as a configuration tool. Startup and shut-down of the "Remote Execution Service" is also handled and a log window is maintained.
- It allows to start and to stop object repository tasks. A second log window shows all the output from the repository.
- Finally the front end allows to start parallel applications. An own window is created for each parallel application. We assume that the main task of an application handles the application's input and output. In contrast an applications parallel subtasks do not receive terminal input, however, their output is multiplexed and appears in the "Remote Execution Service's" log window.

Fig. 8 shows the execution of the "Shared Objects Net-interconnected Computer" on two hosts, namely `luna` and `toliman`. In this example two applications have been run: the prime number checker running with two parallel tasks and checking prime numbers up to ten and a program called `rexec_test` which executed as subtasks the program `hostname` twice — on different hosts as the output shows.

The graphical user interface has been written in Objective-C using NeXTSTEP's fabulous class library. Its execution is therefore restricted to the (Mach 2.5-based) NeXTSTEP platform. However, all the other components of our parallel programming approach can be run on any Mach-system — our favourite candidate is the microkernel version Mach 3.0.

4. Early Results: Competing with a Shared Memory Multicomputer

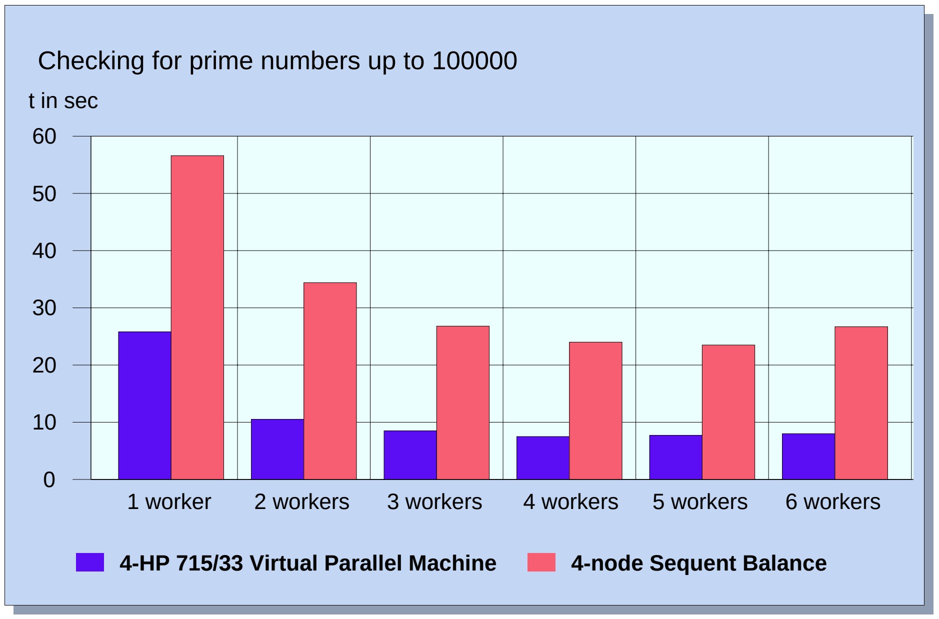


Fig. 9: *Shared Objects Net-interconnected Computer outperforms Sequent Balance*

Among the first parallel applications we have implemented within our environment is a prime number checker. This program originally came from a Sequent Balance and follows a classical shared memory approach. The program maintains a big shared array and stores for every number being checked a notice wether it is prime or not in the array. Several processes are used to check disjunct portions of the array for prime property. A shared counter is used to assign array portions to the different worker processes. We have run the program on a Sequent Balance with four processors (ns32000) and on our ‘‘Shared Objects Net-interconnected Computer’’ with four nodes (HP 715/33). Performance numbers for the HP (Linpack benchmark) measured on the level of a Mach-task are four to five times better than the same numbers for the Sequent. Fig. 9 shows execution times when running the program with up to six worker processes.

Although the structure of the algorithm — n worker processes, disjunct data portions to be written by different workers — allows an easy subdivision of the data structure used into a number of smaller, shared sub-structures, the hardware implementation of shared memory on the Sequent does not provide any means for transparently subdividing data structures. On a shared memory multicomputer the hardware designer and not the implementor of a particular algorithm decides which shared data structures will be available within a program.

On the other hand our object-based approach to parallel computing gives the programmer full control over the grainsize and organization of shared data structures. In our example we have re-implemented the big shared array from the Sequent-algorithm as an array made up of replicated shared subarrays. The C++ implementation of that data structure provides exactly the same interface as the one of the shared array class shown in Section 2.

Let us discuss the effect of that subdivision. For checking prime numbers up to 1 million one has to test prime factors up to 1 000. Let us assume that the shared data structure we use in our program is made up from subarrays of size 1 000. Let us further assume that blocks of 1 000 numbers are checked for prime property by a single task. Then, in a first step the first task has to check for all the prime numbers up to 1 000. Thus, that task is the single one writer to the first shared subarray. Shared objects in our approach are read-replicated. Therefore, when the first step is completed the first subarray is replicated onto all the nodes of our ‘‘Shared Objects Net-interconnected Computer’’. From that point on read accesses to that object happen with the same speed as accesses to completely local data structures. In fact, only the shared integer which tells what subproblem a particular task has to solve has to be transferred over the network during the second phase of our algorithm.

Based on our new implementation we have run the parallel prime number checker for different problem sizes and have seen varying speedups. Obviously computation to communication ratio varies in our case with problem size. Not surprisingly that our our measurements indicate, for small problem sizes a parallel algorithm shows no advantage over a sequential algorithm when running in a distributed environment. Further research is needed to figure out for which parallel algorithms and problem sizes our environment is most effective.

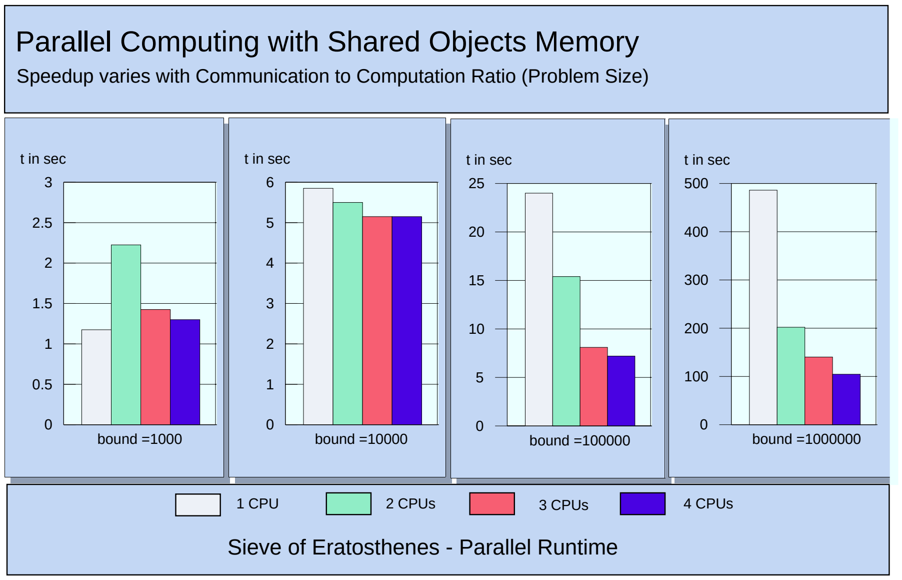


Fig. 10*: Speedups for one to four node Shared Objects Net-interconnected Computer*

The problem of prime number search can be partitioned very easily into subproblems. This partitioning is more difficult for other algorithms. However, we see similar chances for avoiding consistency-related communication in any parallel program if the implementor of a algorithm is not forced to use some shared data structures supported by hardware but can design his own efficient shared data structures. We feel that the object-based approach is very well suited for implementation of shared data with well defined interfaces. The encapsulation of synchronization and communication code within a shared object allows its transparent use in a C++ program.

5. Related Work

A number of projects based on the principle of using a collection of interconnected machines as a concurrent computing platform have been developed.

Many of those systems base on the message-passing programming model, with PVM [7] as the best known representative. A comprehensive list of such systems can be found in [8]. PVM provides a relatively small set of message-passing primitives and is oriented towards heterogeneous operation. Linda [9] is a concurrent programming model based on the concept of a ‘‘tuple-space’’, a distributed shared memory abstraction with operations `in`, `rd`, and `out`, via which cooperating processes communicate. However, the Linda programming model is similar to message-passing. A tuple written into ‘‘tuple-space’’ can be seen as a sent message whereas retrieving a tuple is comparable to receiveing a message.

In contrast to explicit message-passing, the shared-memory programming model provides transparent data communication. A number of distributed shared memory implementations have been described in literature, [1] presents an overview.

The MUNIN system [10] was one of the first software DSM systems which used release consistency as a model of memory coherence. Release consistency enables the system to merge page updates in order to propagate them in a single message on the next release operation. MUNIN offers multiple consistency protocols. Sharing annotations denote the protocol to be used with respect to a particular shared variable. Possible annotations are, for example, ‘read-only’, ‘write-shared’. ‘producer-consumer’, ‘migratory’ or ‘conventional’. However, to exploit all those sharing strategies, most of the consistency aspects have to be controlled by software and the advantage of cheap hardware support by the MMU is partly lost. If the grainsize of shared data items tends to be small, then the page-based release consistency is probably less suitable than true sharing at the level of programmer-defined objects.

MIDWAY [6] proposes the entry consistency model of page coherence. It tries to minimize communication costs by aggressively exploiting the relationship between shared objects and the synchronization variables which protect them. However, those relationships have to be established explicitly by calls to special functions of the MIDWAY runtime systems. So entry consistency depends on the correct use of synchronization primitives throughout the whole parallel program.

The PANDA system [11] provides a page-based distributed shared memory together with a C++ user-level thread implementation based on a pico-kernel. PANDA employs page differencing to reduce communication bandwidth requirements when transmitting shared memory pages. It supports migration of threads and user-level objects. With PANDA the programmer can dynamically specify clusters (co-locations) of related objects which indicate to the system that a set of objects should be treated as unit of sharing and mobility. This helps to lower the probability of false sharing.

The parallel programming language ORCA [12] provides sharing at the level of shared objects. ORCA introduces its own specialized programming model, this way it avoids some problems concerning compliance with existing programming languages. In a distributed prototype environment ORCA has been implemented based on object replication and reliable broadcast to achieve consistent object sharing. ORCA uses a write-update protocol to maintain consistency.

6. Conclusions

We have described the ‘‘Shared Objects Net-interconnected Computer’’, a new parallel programming environment for execution of shared memory parallel programs within a distributed computing environment. With our approach we provide a library with similar programming constructs like many of today’s thread packages. A user-friendly, graphical front end allows for management of the ‘‘Shared Objects Net-interconnected Computer’’ and for startup of parallel applications. Parallel programming in our environment is as easy as multi-threaded programming in traditional environments, however, much higher performance gains are possible.

Shared objects allow for communication between parallel tasks in our environment. A class library provides base classes which implement weakly consistent schemes of memory management — lowering the amount of consistency related communication in the system. Consistency and synchronization-related code can be encapsulated in the implementation of a shared class. Therefore, classes like the ‘‘shared integer’’-class described in Section 2 use weak consistency protocols for their implementation but provide a sequential consistent interface to their clients. Usage of shared objects in our environment is as easy as usage of ordinary C++ objects.

With a parallel prime number checker we have shown feasibility of our approach to parallel computing in distributed computing environments. By clever re-implementation of shared data structures we derived a program which outperformed a four processor Sequent Balance multicomputer with our ‘‘Shared Objects Net-interconnected Computer’’ running on four low-end HP 715/33 workstations. In the future we will implement a number of parallel benchmark programs — hopefully with similar results.

References

[1] N.Nitzberg, V.Lo; *Distributed Shared Memory: A Survey of Issues and Algorithms*; IEEE Computer, August 1991, pp. 52-60.

[2] L.Lamport; *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*; IEEE Transact. on Comp., C-28(9):241-248, Sept. 1979.

[3] J.R.Goodman, M.K.Vernon, P.J.Woest; *Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors*; Proc. of the 3rd Symp. on Arch. Support for Prog. Lang. and Op. Syst., pp. 64-75, 1989.

[4] M.Dubois, C.Scheurich, F.Briggs; *Memory Access Buffering in Multiprocessors*; Proc. of the 13th Annual Symp. on Comp. Arch., pp. 434-442, 1986.

[5] K.Gharachorloo, D.Lenoski, J.Laudon, P.Gibbons, A.Gupta, J.L.Hennessy; *Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors*; Proc. of 17th Ann.Symp.on Comp.Arch., pp. 15-26, 1990.

[6] B.N.Bershad, M.J.Zekauskas; *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*; Tech. Rep. CMU-CS-91-170, School of CS, CMU, Sept. 1991.

[7] V.S.Sunderam; *PVM: A Framework for Parallel Distributed Computing*; J. Concurrency: Practice and Experience, 2(4), pp. 315-339, Dec. 1990.

[8] O.A.McBryan; *An Overview of Message Passing Environments*; Parallel Computing, Vol. 20, No. 4, pp. 417-444, April 1994.

[9] N.Carriero, D.Gelernter; *Linda in Context*; Communications of the ACM 32(4), pp. 444-458, April 1989.

[10] J.B.Carter, J.K.Bennet, W.Zwaenepoel; *Implementation and Performance of Munin*; Proc. of the 13th ACM Symp. on Op. Syst. Principles, pp. 152-164, 1991.

[11] H.Assenmacher, T.Breitbach, P.Buhler, V.Hübsch, R.Schwarz; *PANDA — Supporting Distributed Programming in C++*; Proc. of ECOOP'93, LNCS vol. 707, 1993, pp. 361-383.

[12] H.E.Bal, F.Kaashoek; *Orca: A Language for Parallel Programming of Distributed Systems*; IEEE Transactions on Software Engineering, Vol.18, No.3, March 1992.

[13] M.Malek, A.Polze, M.Werner; *A Framework for Responsive Parallel Computing in Network-based Systems*;
in Proceedings of International Workshop on Advanced Parallel Processing Technologies, Bejing, pp. 335-343, September 1995.