THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-97-04

HeROD Flavoured Oct-Trees: Scientific Computation with a Multicomputer Persistent Object Store

Stephen Fenwick and Chris Johnson

February 1997

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-97-03 Brendan D. McKay. *Knight's tours of an 8 × 8 chessboard.* February 1997.

TR-CS-97-02 Xun Qu and Jeffrey X. Yu. *Mobile file filtering.* February 1997.

TR-CS-97-01 Peter Arbenz and Markus Hegland. *The stable parallel solution of general narrow banded linear systems.* January 1997.

TR-CS-96-09 Ralph Back, Jim Grundy, and Joakim von Wright. *Structured calculational proof.* November 1996.

TR-CS-96-08 David Hawking and Paul Thistlewaite. *Relevance weighting using distance between term occurrences.* August 1996.

TR-CS-96-07 Andrew Tridgell, Paul Mackerras, David Sitsky, and David Walsh. *AP/Linux - initial implementation*. June 1996.

HeROD Flavoured Oct-Trees: Scientific Computation with a Multicomputer Persistent Object Store *

Stephen Fenwick and Chris Johnson
Department of Computer Science
Australian National University

February 19, 1997

Abstract

Using a persistent multicomputer object store should greatly simplify the writing of distributed memory parallel programs operating on irregular, object-structured data, by removing from the programmer the burden of managing data referencing, distribution and coherency. In this report we explore the development of such a program using the HeROD persistent multicomputer object store, in a computational science application. The computational code is part of the tree-code algorithm for the N-body problem. A number of solutions to the difficulties of combining flat transactions and cooperating parallel processes are explored, in both the application programming domain and the persistent store design. Actual performance measures of the implementation on a 128-processor multicomputer are reported, with scalability comparisons for a range of processor configurations (exploring sensitivity to the client-server balance) and program implementation strategies, particularly by varying transaction size.

*This work was performed as part of the HeROD project of the Advanced Computational Systems (ACSys) Cooperative Research Centre.

1 Introduction

Using a distributed persistent object store as a paradigm for programming multicomputers has the potential to greatly simplify the writing of parallel programs, by removing from the programmer the burden of managing data distribution and coherency. In this report we describe our experiences developing a computational science program for the HeROD distributed object store [JYS95].

We try to answer the question: "how well does the HeROD programming model support the writing of simple yet efficient concurrent programs?" The subject of study is the effects on parallel performance of those aspects of data management of persistent objects that are visible to the application programmer. We employ two measures: a coarse measure of limitations on ideally achievable parallelism, which assumes that the application algorithm provides sufficient potential parallel computation for some large available set of processors to execute concurrently, if data contention were ignored; and measured times and scalability comparisons of actual executions.

The data management strategies of the HeROD implementation depend on design assumptions about the patterns and degree of data contention for a notional target class of user programs. In turn, in particular application programs the degree of data contention can be modified to some extent by the programmer's choice of algorithm, made according to his or her knowledge of the implementation strategy. Here we study the range of program implementation strategies open to the application programmer above the HeROD persistent object store implementation, and within the HeROD flat, optimistic transaction model, but using our knowledge of the gross timing characteristics of some aspects of that implementation.

This study shows that the persistent programming paradigm is of benefit to multicomputer applications programmers in practical cases, but it also shows that complete transparency of the persistence mechanisms is not possible where performance is an issue.

2 N-body Problem

The application we use as an example is the oct-tree N-body algorithm described by Barnes and Hut [BH88]. A parallel implementation of this algorithm has already been developed by Warren and Salmon [WS92]. This implementation was carefully built to exploit a Thinking Machines Corporation CM5, and in fact did this well enough to win the Gordon Bell prize. A later implementation study by the same authors made use of an indexing scheme that in some way resembles a user-level implementation of the simple store-wide object addressing of HeROD, but greatly increased the complexity of the programming[WS93].

The N-body treecode has become established as a benchmark for distributed memory systems, for example Scales and Lam[SL94] using the the SAM virtual distributed object memory system: this system has a quite different concurrency control mechanisms that required a lot of programmer attention, but is the basis for the work reported here; a distributed implementation of the algorithm for the Amoeba distributed operating system is described by Romein and Bal[RB95]; and another, also on the Fujitsu AP1000, by Field and Boothroyd[FB96]. The complex coding of these implementations indicates how far the programmer must go from a simple programming model to achieve acceptably efficient execution. Since much of the complexity of the Warren and Salmon algorithm results from the issues of data referencing, sharing and distribution, the virtual shared object space provided by the distributed persistent store promises to greatly simplify the programming task.

The N-body problem occurs in many fields of astronomy, for example galactic and cos-

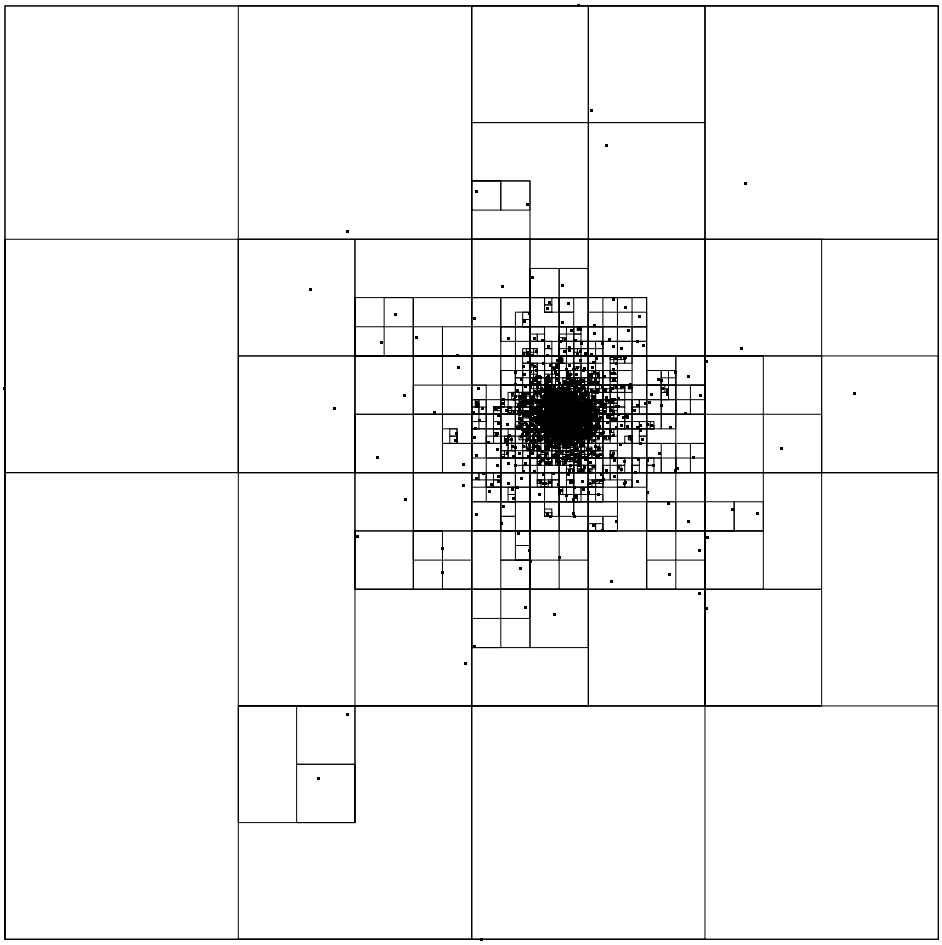

Figure 1: Quad-tree containing 2000 bodies

mological simulations. It involves calculating the gravitational forces acting between a large number of massive bodies, and thus simulating the evolution of the system over time. A naive way to calculate the force acting on each body is to sum the attraction of every other body in the system, but this approach leads to N^2 complexity.

The Barnes and Hut approach reduces an N-body force calculation from N^2 complexity to approximately $N \log N$, by arranging the bodies into a spatial oct-tree. This involves a hierarchical subdivision of the coordinate space into cubic cells, each of which is recursively divided into eight sub-cells whenever more than one body occupies the cell. The cost of calculating the forces on a particular body due to each other body can be reduced by using an approximation for the cluster of bodies in sub-trees that are sufficiently distant, the cluster then being treated as a single body with the sum of its constituent masses positioned at the centre of mass of the cluster.

The set of bodies is organised into a tree, where the internal nodes represent the cubic cells, and the leaf nodes each represent a body. The descendents of any given node are the nodes for the sub-cells and bodies that lie spatially within the corresponding cube, within a spatial neighbourhood; unrelated tree nodes therefore represent spatially distinct neighbourhoods. As an example of this recursive subdivision approach, Figure 1 shows the two-dimensional equivalent – a quad-tree – containing 2000 bodies.

3 Oct-Tree Implementation Using HeROD

HeROD provides a very simple a programming interface for persistence because its operational interface was designed as run-time support layer for a higher level programming language (which was never instantiated). However, the interface can be used to support experiments in programming over a parallel persistent object store. In this case study we looked at the programming of parallelism over HeROD's object store viewed as an implementation of a shared heap store, and the persistence properties were not essential to the application. The durability of persistence in this context provides a transparent checkpoint facility that could be useful for long-running executions of this algorithm, as discussed below in section 5.1.

The parallel programming model provided by HeROD is one of processes acting independently on a shared heap of objects, their concurrency being controlled by flat transactions. The programmer controls the order of access to objects, the "size" of transactions (the number of objects accessed in each), and the types of operations (read, update, create) in each. Our implementation of the N-body problem is based on the implementation by Scales and Lam for the SAM distributed memory system [SL94]. Because of the differing characteristics of SAM and HeROD the details of our implementation differ considerably from those of the SAM implementation. However, the overall structure is very similar. Each physical processor runs a single application process. Parallelism is achieved by partitioning the responsibility for operations that update the values of the bodies between processors, by allocating approximately equal sized subsets of the bodies to each process.

The implementation has the following phases of computation:

1. **Initialisation** The set of bodies to be simulated is created. In our example each body is given a random initial position with a clustering distribution. Each processor generates an equal number of bodies. Alternatively the bodies could be read in from an external file. While generating the initial data each processor calculates a cube that encloses all of the bodies that it generated. This phase involves no sharing of data.

2. **Build Oct-tree**

   (a) *Bounding cube* A bounding cube that encloses all the bodies is calculated. This phase involves each of the processors adding its cube's bounds to determine an overall enclosing cube.

   (b) *Insert bodies* Each process inserts its set of bodies into the common oct-tree. Each body is independently inserted into the shared tree, which is initially empty. Inserting each body into the tree involves descending from the root of the tree to a leaf node, at each level choosing one child to follow depending on the position of the body, and creating additional internal tree notes when two bodies would otherwise lie at the same leaf position.

   (c) *Combine Clusters* The total mass and centre of mass is calculated for all internal nodes. This is done serially by a single processor.

3. **Calculate Forces and Update** Each process chooses a set of bodies in a physical neighbourhood (contained in a particular subtree) and calculates the force on each one. The force calculation for each body traverses the tree, descending into each part of the tree as far as necessary to obtain an accurate force calculation. Once the force has been calculated the velocity and position of the body can be updated for the next iteration. A new set of bounding cubes is then calculated for each processor. This phase has not been implemented in our benchmark example.

4. The computation is repeated from step 2a using the updated positions and velocities.

4 Programming Issues

Writing the N-body application raised several issues concerning the way in which distributed object stores should be programmed. In this section we discuss some of these issues and their implications for the design of distributed object stores.

4.1 Synchronization

One of the apparent advantages of concurrent processing over distributed object stores is that they avoid the need for explicit synchronization. HeROD provides optimistic flat transactions for concurrency control at the individual object level using a mechanism described in a companion report [Joh97]. Before each transaction is committed the data it has accessed is checked for consistency with the current state of the store, and the transaction is aborted if it is found to be inconsistent. Since the store is always in a consistent state it should be possible for an application to execute operations at any time without regard to other operations that are taking place concurrently. If there are any conflicts between operations they are resolved by the transaction mechanism.

In practice things are not quite that simple. Many applications have a notion of consistency that extends beyond the normal scale of transactions, which are designed to ensure read–write consistency over objects or groups of objects. For example, Section 3 described the N-body program in terms of phases, each of which must be executed in sequence.

Phase 2b, inserting the bodies into the oct-tree, is possible only once phase 2a, the generation of a cube enclosing the entire set of bodies, has been completed.

Phase 2a in turn cannot be completed until *all* processes have finished generating bodies in phase 1 or updating them in phase 3.

It is not possible to make the whole of a phase a single transaction for each process, since this would serialise all the processing in this phase, because they read many objects and update some objects in the common tree. The insertion of a single object is a suitably sized operation for a transaction, so that many insertions can proceed concurrently; processes only collide and back out (retry) if two processes simultaneously (in a transaction timescale) insert into one interior node, or insert a new node above a leaf. But at this scale it is possible that one processor may complete several transactions in phase 2b before another has completed all object generation and bounding cube calculation in phase 2a.

If some processes start inserting bodies into the oct-tree while other processes are still generating bodies there is a possibility that some of the new bodies will lie outside the initially computed bounds of the oct-tree, in which case the oct-tree may be ill-formed and the computation will fail. This is a type of conflict that will not be detected using the standard transaction mechanism. This problem is due to the fact that the computation contains distinct phases, and that different operations are valid in each phase. In phase 1 it is permissible to generate a new body but not to insert bodies into the oct-tree. In phase 2b on the other hand it is permissible to insert a new body into the oct-tree but not to generate new bodies. If either of these operations is performed in the wrong phase invalid results will be generated, even though no conflict has occurred at the object level.

To deal with this problem we needed to violate the pure “object store paradigm” by using traditional barrier synchronisation operations (`MPI_Barrier`) to separate the phases. In principle this could be avoided by spin locking on the value of a single object, but this would be

very inefficient, creating a high load on the single server that manages the spin lock object and probably interfering with other clients that are still doing useful work. The barrier synchronisation call is functionally equivalent to an implementation that uses the object store, but avoids overloading the servers.

4.2 Producer–Consumer/Shared work pool problem

As stated above, the central object-insertion phase of the N-body program repeatedly takes a body from a shared pool and inserts it into the oct-tree. An obvious way to implement this is for many processors to perform insertions concurrently, representing each body as an object and maintaining the pool as a shared collection structure such as a linked list. However, this creates a problem: it turns out that performing insertions and deletions on such common collection structures effectively serialises the computation.

To see how this happens, suppose that a linked list is used to store the pool of bodies, and that several clients are simultaneously executing a loop of the form:

```
while (more bodies) {
    repeat {
        start transaction;
            fetch body from list;
            add body to tree;
        end transaction;
    until (last transaction was successful);
}
```

Although this is a very natural way of implementing this stage of the problem, it turns out that it allows virtually no parallelism. The reason for this is that all transactions that insert an object into the linked list have to update the value of an object that points to the head of the list. Under most transaction models, including the one used by HeROD, concurrent transactions that read and update a common object are not allowed. Any other process that reads from the list before the first transaction is committed will read the same object, and will attempt to insert it again into the tree. When the second process's transaction tries to commit, it will fail (the pool object has been updated by the first to commit, so the second transaction is invalid) and the process must retry the transaction. This time it will fetch another object and possibly succeed, but only one object from the pool can be successfully worked with by any number of processes: the computation has been serialised.

This is an example of the very general worker-farm paradigm, and it is very desirable to have a general solution that allows this paradigm to exploit parallelism.

The rest of this section discusses possible solutions for this serious problem. Note that the same serialization problem does not occur in practice with the concurrent insertions of body objects into the oct-tree itself, since after the first few nodes have been created the processors are likely to be inserting into different nodes, at some distance down the tree. Our measurements of data contention below bear this out.

4.2.1 Ad-Hoc Solution

In writing the N-body program we were able to avoid the problem completely by partitioning the work pool between processors. Each processor generates objects into its own object set in phase 1, and inserts only its own objects from its set into the oct-tree, in phase 2b. The data structure required is a separate list of bodies for each process, representing the object sets, and

the operations of inserting generated bodies into the lists and deleting from it cause no object-contention at all. This is not ideal for two reasons:

- It can provide for very poor load balancing.

  Load balancing is one of the desirable properties of the worker-farm with a common pool of work items. In this case the number of work items that each processor must handle is fixed at the number generated into its own set. If some work items take much longer than others so that processors finish at different times there is no simple way for work items to be redistributed from the processors that are still busy to the ones that are idle.

- It assumes that each processor generates its own work items and is sole user of its object set.

  Although this is the case in the N-body program it will not generally be true. The work items may be generated by separate specialized processes, which may be desirable to improve functional parallelism.

4.2.2 Data Type Solution

Another solution to this problem is to introduce new specialized datatypes to the object store. For example, Schwartz and Spector [SS84] describe a data structure called a *Weakly FIFO Queue* (WQueue). A WQueue has similar operations to a standard FIFO Queue (**WQEnter** and **WQRemove**).

However, the strict FIFO semantics are relaxed.

WQRemove need not remove the element at the head of the queue; hence multiple **WQRemove** operations may be performed concurrently as long as they return different elements.

The requirement that all the items entered into a queue by a single transaction appear consecutively at the head of the queue is also relaxed, so that several transactions can perform **WQEnter** operations concurrently. It is important to note that operations on the WQueue are still atomic; improved concurrency results simply from relaxing the strict FIFO property of the queue.

This WQueue datatype solves the producer–consumer problem by allowing simultaneous accesses to the data structure. Thus multiple producers and consumers can access the set of work items simultaneously, as long as sufficient work items are available to provide all concurrent consumer transactions with a unique work item.

Unfortunately it is extremely difficult to implement this datatype efficiently using the operations provided by HeROD (and most other object stores). One solution is to extend the object store to include a WQueue datatype. This datatype would provide the operations **WQEnter** and **WQRemove** rather than the **read** and **write** operations that are provided by normal objects.

These operations have been implemented in a separate extension to the HeROD store by one of us [Fenwick]. We do not discuss this aspect further in this report.

4.2.3 Programming Solution

Another way around the producer–consumer problem is to retain the common list structure, but to split the transaction into two:

```
while (more bodies) {
    repeat {
        start transaction;
            fetch body from list;
        end transaction;
    until (last transaction was successful);
    repeat {
        start transaction;
            add body to oct-tree;
        end transaction;
    until (last transaction was successful);
}
```

This does not completely solve the problem, since the transactions that fetch the bodies from the common list are still serialized. However, the oct-tree insertions are now in separate transactions and can take place in parallel. If the tree insertion takes considerably longer than the list fetch, this may be a significant performance gain.

However, this has created a new problem: since the operation of fetching and inserting is no longer atomic there is a potential data recovery problem if the system crashes in the time between the first and second transactions being committed. HeROD does not provide full orthogonal persistence: the program stack and its extension the program heap are not persistent, and references that are not in the object store will not be protected from a system crash. If the system crashes between the two transactions the store will be left in an inconsistent state with respect to this algorithm, since a new body has been removed from the new body list but not inserted into the tree. The body will effectively have disappeared.

A more sophisticated approach that deals with the problem of data recovery is to use temporary staging areas per processor, for each body as it is in transit:

```
while (more bodies) {
    repeat {
        start transaction;
            fetch body from list;
            insert body into private per processor area;
        end transaction;
    until (last transaction was successful);
    repeat {
        start transaction;
            add body to tree;
            delete body from private per processor area;
        end transaction;
    until (last transaction was successful);
}
```

This deals with the above problems since if the system crashes between transactions the body is not lost but is stored in a known location. For full recovery this requires that additional crash recovery code be written to move bodies from the private areas back into the common list; such code is notoriously error-prone because it is difficult for users to be fully aware of system-level considerations, and because it is rarely exercised.

Although the producer–consumer / shared pool problem can be solved using programming techniques such as these, the complexity of the solution increases programming effort and results in more opportunities for errors in the program.

4.2.4 Transaction Mechanism Solution

A more far-reaching solution to this problem is to alter the transaction model so that information can be shared between transactions. For example, nested transactions [Mos90] provide this facility.

We have implemented a version of HeROD that provides a simple form of nested transaction and using this were able to implement a datatype similar to the WQueue described by Schwartz and Spector.

5 Performance Issues

We have studied the performance of the tree construction phase of this problem, and not the force calculation phase. Although the latter is expected to be much more time consuming in a real instance of this problem, it is the case that once the shareable tree structure has been set up by the construction phase, the force calculation involves very little movement of data and no contention. This phase is therefore expected to scale very well and run at near full computational speed. It is the tree construction phase that reveals the problematic issues of a multicomputer object store, since it is data intensive and has many opportunities for contention.

Tuning the performance of the tree construction phase of the N-body application raised two important considerations in the use and design of distributed object stores. These are the relationship between data coherency and crash recovery (both of which HeROD currently addresses using transactions), and the trade-off between data replication and the cost of maintaining replicated data, which again is closely related to HeROD's transaction mechanism. Both of these considerations resulted in us using larger transactions than would otherwise have been desirable. This section describes these issues and their implications for object store design. Some of this discussion relies on performance results that are presented in Section 6, where we report a series of experiments for insertion of 10,000 bodies into an oct-tree. The hardware configuration was the ANU Fujitsu AP1000 with 128 processors, each with 16M bytes of RAM; the filesystem had 16 disks attached to separate processors, each of 4 Gbytes.

Our experiments use fewer than 100 processors as clients and servers. The 16 processors with disks attached were included in this set, but the effect of the additional workload on these processors was not separated out.

5.1 Data Coherency and Crash Recovery

The atomic transactions provided by HeROD have four properties described by the acronym ACID: *atomicity*, *consistency*, *independence*, and *durability*. Atomicity refers to the property that a transaction is never seen in a half-completed state. Consistency and independence refer to the properties that each transaction sees a consistent view of the store and executes independently of other transactions.

Durability refers to the property that a successfully committed transaction is permanent. The property of durability includes *crash resiliency* which means that data will not be lost if the system crashes. However, strict durability is unnecessary for many classes of problem, and it has a significant performance overhead that is not justified by its benefits. The cost arises from the fact that to be durable every transaction must be written to disk when it is committed. If transactions are small, which is the case in the N-body problem, this results in many writes of small amounts of data to disk, and may lead to programs becoming disk-bound for this reason alone.

This writing of data to disk accounts for a major proportion of the total execution time in our case study, as shown by comparing the two lines labelled “1 insertion / transaction” in Figure 6. The upper line (by default) shows the time for the program if it writes the result of each transaction to disk; the lower (labelled “no write to disk”) shows 30-50% better performance if all disk writes are omitted.

The nature of the N-body program means that strict durability is unnecessary. The ACID transaction model was developed for databases that describe, and frequently interact with, the outside world. Losing information in the database is unacceptable because then the state of the database no longer reflects the state of the outside world. For example, a bank *never* wants to lose track of the fact that a customer has made a withdrawal from an account, an operation that is modelled (and implemented) by a committed transaction: crash recovery should be to the state of the store at the last committed transaction. For the same reasons of real-world interaction, crash recovery should be fast.

The N-body application, along with many other scientific multicomputer applications, does not interact with the outside world except through the object store interface to the filesystem. Thus if the application or system crashes it is acceptable to restore the store to any consistent past state, not necessarily the most recent one, since the computation performed since that state was valid can easily be repeated. It is unnecessary to restore the state to the last committed transaction. Since system crashes are usually very infrequent, it may well be acceptable to use a slow recovery method, traded off against time saved at every transaction. This resembles traditional checkpoint mechanisms for long-running computations, but with transactions automatically providing coherence of the saved state.

A more satisfactory solution is to completely decouple crash resiliency from data coherency so that data is not written to disk when a transaction is committed. Instead explicit checkpoints would be used to update the state of the object store on disk. Thus transactions could be made small to avoid contention, without resulting in the overhead of writing every transaction to disk. The programmer could also chose when to update the data on disk so as to provide appropriate resilience while minimizing I / O costs. Resilient transactions could still be provided if required by the programmer. Mechanisms that allow short transactions to be combined with infrequent checkpoints include explicit checkpoints and nested transactions, such as those being explored for Persistent Java by Daynes[Day96].

There are some exceptions to this rule that the system state can be automatically restored, although they do not significantly affect the above discussion. Many multicomputer applications do in fact interact with the outside world, either to input problem parameters or an initial simulation state, or to output partial results. If input is read from a file, as is often the case, it is simple to recreate the data by simply resetting the state of the file descriptor. If part of the computation is repeated after a crash a portion of the output data may be repeated, but it would be straightforward to remove the duplicate portion.

5.2 Server Contention

At present a major limit to the scalability of the system is the way in which HeROD commits transactions to disk at the end of each transaction. In this application transactions are quite small. Larger transactions are likely to lead to more efficient use of the servers and filesystem and deliver higher throughput.

Analysis reveals a high cost of committing a transaction. After the transaction has been validated, each server that owns any objects that were modified by the transaction writes them to disk. This causes those servers to block for the duration of the disk write (on average about this takes about 15ms). The server is unable to service requests from other clients during this

time. The first consequence of this is that if the typical transaction is big enough that it typically includes an update to an object that is maintained by each server then each commit operation results in all servers being blocked for around 15ms. Thus the whole system will be unable to complete more than about 60 transactions per second. The counteracting effect is that a commit operation blocks the server for a time that is less than proportional to the number of objects being updated. Firstly the server writes pages, not individual objects, to the filesystem; hence any degree of locality that reduces the number of pages in proportion to the number of objects will reduce the number of filesystem write requests. Secondly the filesystem has multiple disks which one server can access concurrently. On a lightly loaded filesystem multiple pages can be written in little more elapsed time than one page, the disk seek and rotation latencies overlapping.

A more general observation is that in any client–server system it is desirable to avoid having the servers perform long, non-preemptable operations. The problem of server contention is particularly severe in this architecture because the server cannot be pre-empted, leading to queueing of all kinds of client requests for the full duration of preceding operations.

5.3 Transaction Size, Data Caching and Data Contention

Section 5.1 described how increasing the size of transactions results in data being written to disk less frequently.

Larger transactions have a further benefit by reducing a performance hit that results from the fact that client processors do not maintain local copies of objects between transactions. Using larger transactions means that the client's copies of objects are kept longer and the same object is fetched less frequently from the server. Since fetching an object from the server is expensive compared to using a local copy, this can have a significant impact on performance. Once again a trade-off results from the fact that larger transactions result in more data contention, which reduces performance.

We made an experimental investigation of this issue. The gains in performance by inserting numbers of bodies greater than one in each transaction can be seen in Figure 2.

The increased data contention on tree nodes is revealed in the rising number of aborted transactions as the transaction size is increased, in Figure 3. The increased performance is apparent up to approximately 10 bodies inserted per transaction; the effect of more aborted transactions, because of increasing data contention on tree nodes, is clearly apparent, but this does not affect overall performance below about 12 insertions per transaction.

This is because an aborted and retried transaction requires no disk I/O, and is relatively cheap compared to the disk I/O required to commit a successful transaction.

Analysis by the CREST performance analysis system [Fen97] revealed another bottleneck that is related to small transactions. Writing data to disk involves a fixed delay of around 15ms, during which the server is unable to service other requests from clients. If a single commit requires a single server to write multiple pages to disk these writes are performed in parallel, with the result that this delay is largely independent of the size of the transaction. Since a client committing a transaction does not wait for the server to complete the write before proceeding with other computation this does not affect the time taken to commit a transaction as seen by the client. However, if many small transactions are committed a large part of each server's time is taken waiting for writes to disk to complete, causing contention for the server and slowing down other types of operation.

As mentioned above, the HeROD implementation discards all objects in the client's cache at the end of a transaction. Any object that is needed again must be fetched again, but it is up to date at that time. This points to a basic tradeoff in the design of distributed object stores. It

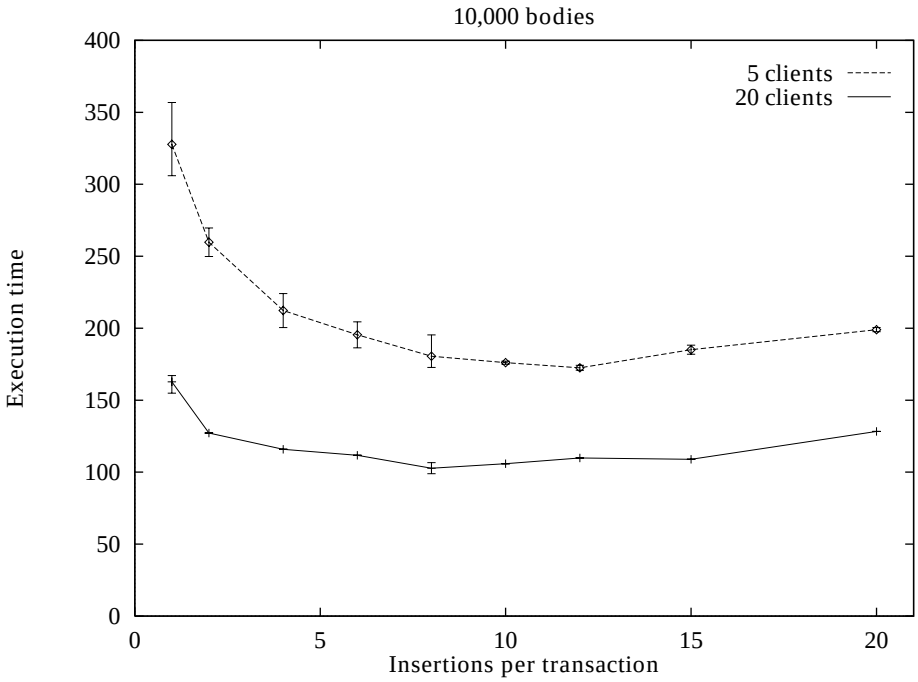

Figure 2: Effect of transaction size on performance, with 5 servers and 5 or 20 clients

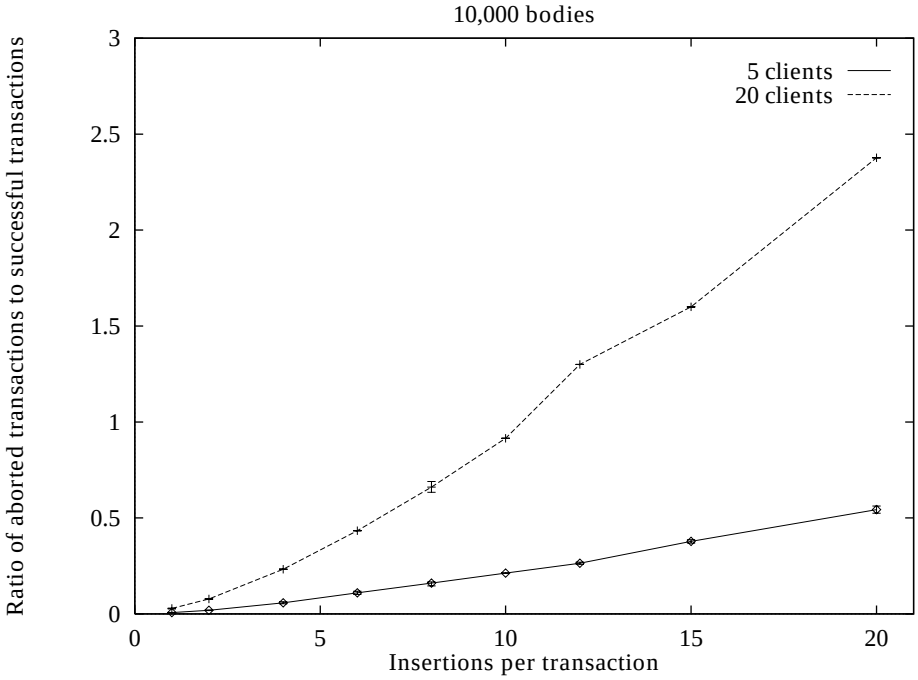

Figure 3: Effect of transaction size on proportion of aborted transactions, with 5 servers and 5 or 20 clients

appears attractive to consider maintaining the local copies between transactions, particularly in a program like N-body where after the initial flurry of in-fill in the top of the tree, many tree nodes change little thereafter but are used by many transactions. On reflection, however, in general this would either lead to more transactions being aborted (since the local objects would be more likely to become out of date) or require additional work to determine whether the objects were valid at the start of the transaction. In the case of such small objects as we see in this study, the cost of a re-validation message exchange for each object is little different from the cost of flushing the local cache and re-fetching all objects on demand.

The comparison of large and small sizes of transactions in Figures 2 and 3 points to performance gains from retaining replicas longer, for this N-body program. Further studies are needed to reveal whether this is a general phenomenon, and to what scale of transaction it applies.

6 Scalability Performance Measurements

In multicomputer studies the scalability of performance of an application or software system, with respect to problem size or numbers of processors, is a significant measure of quality. We report on varying three parameters of the N-body program: the number of client processes against a fixed number of servers, the number of client/server processors in a fixed ratio, and the number of client/server processes in different ratios. Unlike the normal distributed database studies where the number of servers is small and nearly fixed, in a multicomputer with a parallel filesystem the number of server processors can be freely varied, and choice of which processors are servers is not tied to the processor having a disk directly attached.

6.1 Client Scalability

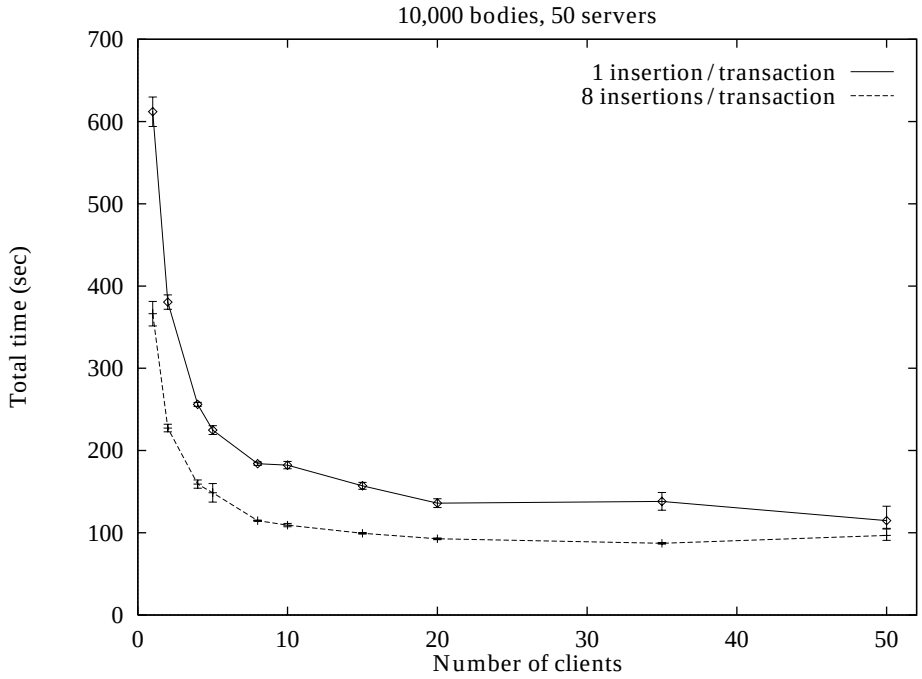

Figure 4: Execution time, varying number of client processors

First we look at the effects of varying the number of clients while holding the number of servers constant. Figure 4 shows how the time to insert 10,000 bodies into the oct-tree varies with the number of clients while the number of servers is held at 50. Curves for execution time with both one and eight insertions per transaction are shown. Performance increases well up to around 10 processors and then starts to level off. This levelling off is due to servers becoming overloaded, as well as increased data contention caused by the large number of processors simultaneously accessing the oct-tree.

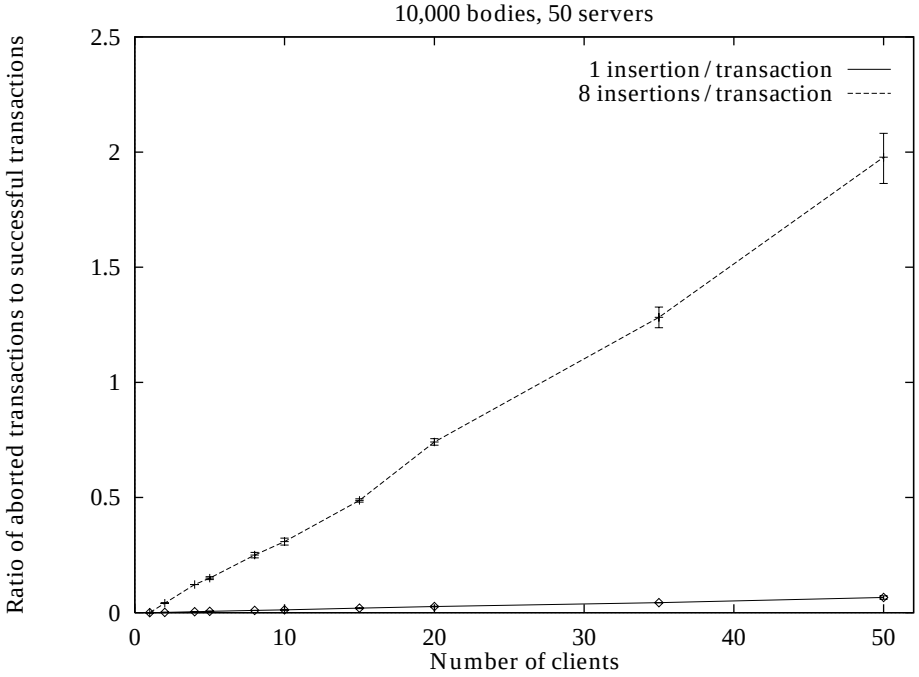

Figure 5: Effect of number of client processors on proportion of transactions aborted

Another point to note is that with a small number of clients increasing the transaction size results in a considerable performance improvement, but that this effect diminishes as the number of clients is increased. This can be explained by the fact that when large transactions are combined with a large number of clients data contention becomes much more common. This effect is shown more clearly in Figure 5, which shows the proportion of aborted transactions to successfully committed transactions. Since in HeROD the only reason for transaction abort is data contention, this effectively shows the level of data contention. With small transactions the level of data contention is relatively low, affecting less than 10% of transactions even with 50 client processors. With larger transactions, for example containing eight oct-tree insertions per transaction, data contention is a much more significant problem. With 50 clients concurrently executing transactions that each contain eight oct-tree insertions, aborted transactions outnumber successful transactions by a factor of almost 2 to 1. The fact that relatively good overall performance is still possible in this situation, as shown in Figure 4, is due to disk I/O accounting for a large portion of the cost of a successful transaction. Aborted transactions are not written to disk, and so are much cheaper than successful ones.

6.2 Overall Scalability

Next we present the effects of varying the number of client and server processes together. Figure 6 shows the time to insert 10,000 bodies into the oct-tree varying the total number of client

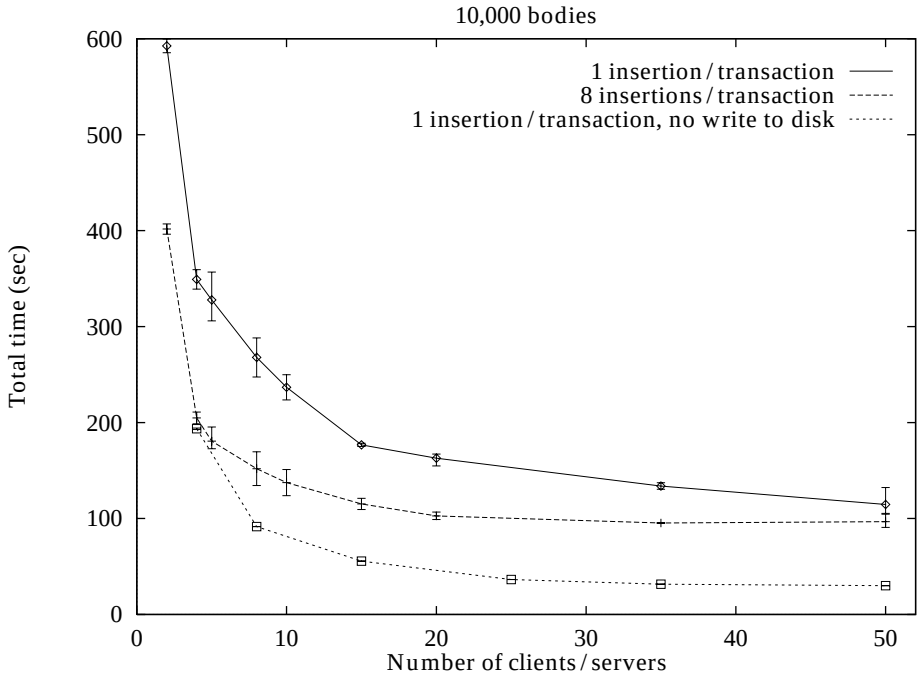

Figure 6: Execution time, varying number of client and server processors together

and server processors, but holding the number of each type equal. The x-axis of this graph shows the number of clients – or equivalently the number of servers. The total number of processors is thus twice this value. This graph shows the time required for a transaction size of a single insertion, and for eight insertions.

Again we see that the larger transaction size causes a significant increase in performance for small numbers of processors, but that this increase is far less for large number of processors. This is because of increased data contention, as described in Section 6.1. Figure 6 also shows the effect of disk I/O on system performance. HeROD includes an option that prevents data being written to disk at the end of each transaction. The "no disk write" data shows performance with this option disabled.

However, the data set shows the expected dramatic improvement in performance, and improved scalability (approaching 20 times), that results from avoiding disk activity at the end of each transaction.

Figure 7 shows the speedup for the single insertion per transaction case relative to a single processor.

Unfortunately, because of a local server cache size limitation of the HeROD implementation, it was not possible to run the 10,000 body problem using a single server. For this reason we have estimated the time for one client and one server as being twice the time for two clients and two servers. Thus the graph is likely to slightly overestimate the actual speedup, although this error should not be large. The speedup approaches a value of only 10, which is not unexpected given the intensive data activity and committing many very small transactions to disk, in this insertion phase.

6.3 Client–Server Balance

Figure 8 shows the effect of holding the total number of processors at 100, but varying the balance between clients and servers.

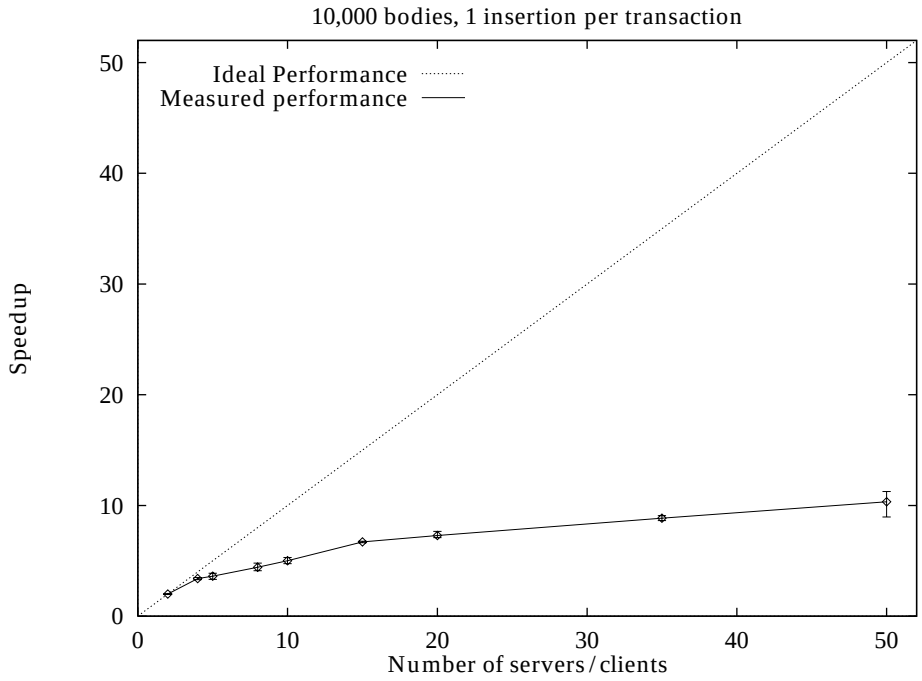

Figure 7: Speedup, varying number of client and server processors together, with one insertion per transaction

Figure 8: Execution time, varying balance of client and server processors

Although performance decreases significantly at extreme imbalances, we see that relatively good performance is obtained from a ratio of 2 servers for every client, through a 1:1 ratio, through to a ratio of 2 clients:1 server.

As the number of clients is increased the number of concurrent processors inserting into the oct-tree is increased, tending to improve performance. At the same time two factors decrease performance: the smaller number of servers results in increased server contention, and the larger number of clients results in more data contention. The effect of increased data contention is particularly severe when many clients are used, with the result that with 90 clients performing eight insertions per transaction gives poorer performance than inserting a single body per transaction.

6.4 Client-Server Specialisation

The implementation of HeROD used in this study has the property of specialising processors to act as either client or server, but not both. This has the advantage of simplicity in the system implementation, given that the underlying operating system has only non-preemptable scheduling; but it risks losing a large amount of potential processor power to idleness when a client processor is awaiting an reply to a fetch or transaction commit request from a server. Generalising the processors to act as both client and server is possible using some local extensions to MPI[Bla96] and combining the client and server processes as pseudo-threads in one process. It is expected that this application would scale much better in this case measured against numbers of processors, rather than against pairs of processors as we have shown here.

7 Conclusions

This document has described the implementation of part of an N-body simulation algorithm on the HeROD distributed object store.

The main points that arose from this exercise are:

- Distributed object stores provide an effective and elegant way of sharing data between processors.

  In addition the use of transactions avoids much of the complexity of shared memory programs and this advantage is delivered in simplifying the programming of this treecode algorithm.

- In some situations, careful design of the program is needed to avoid the effects of data contention.

  This was particularly significant when designing a structure to contain the bodies to be inserted into the oct-tree. It was very difficult to find a satisfactory solution to this problem using the services provided by HeROD, leading us to the conclusion that a distributed object store needs to provide specialized data structures to deal efficiently with common programming problems such as the producer–consumer problem.

- The issues of data coherency and crash recovery need to be separated.

  HeROD solves both problems using transactions by writing all updates in a transaction to disk at the end of the transaction. This causes a significant system overhead. Given that system crashes are generally rare, and in this type of application any lost computation can easily be recreated, a much coarser grain of crash recovery is appropriate. However,

fine grain transactions are still needed to avoid data contention. Thus transactions should be used only to maintain coherency, and data written to disk much less frequently than at present, possibly using a separate checkpoint mechanism.

References

[BH88] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 14(8):1071–1080, 1988.

[Bla96] S. M. Blackburn. Multicomputer object stores. In S. M. Blackburn, editor, *3rd IDEA Workshop (Integrated Data Environments Australia)*, July 1996.

[Day96] Laurent Daynes. A flexible transaction model for persistent Java. In M. Jordan and M. Atkinson, editors, *First International Workshop on Persistence and Java (PJ1)*, 1996.

[FB96] Anthony J. Filed and Simon Boothroyd. An evaluation of software cacheing in astrophysical n-body simulation codes for the Fujitsu AP1000. In *Proceedings of Fifth Fujitsu Parallel Computing Workshop*, Kawasaki, Japan, 1996. Fujitsu Parallel Computing Research Centre.

[Fen97] Stephen Fenwick. *State-based Performance Analysis of Multicomputer Object Store Applications* forthcoming. PhD thesis, Australian National University, 1997.

[Joh97] Christopher W. Johnson. The HeROD multicomputer persistent object store *to appear*. Technical Report TR-CS-97-nn, Dept of Computer Science, Australian National University, 1997.

[JYS95] C. W. Johnson, J. X. Yu, and R. B. Stanton. Architecture of a high-performance persistent object store. In J. Darlington, editor, *Proceedings of the Fourth International Parallel Computing Workshop*, pages 297–306, London, U.K., 1995. Imperial College Fujitsu Parallel Computing Centre.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object system. *Transactions on Information Systems*, 8(2):103–139, 1990.

[RB95] John W. Romein and Henri E. Bal. Parallel N-body simulation on a large-scale homogeneous distributed system. In *EuroPar'95*, August 1995.

[SL94] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[WS92] Michael S. Warren and John K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing '92*, pages 570–576. IEEE Computer Society, 1992.

[WS93] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing '93*, pages 12–21. IEEE Computer Society, 1993.