RICE UNIVERSITY

Message Passing Versus Distributed Shared Memory on Networks of Workstations

by

Honghui Lu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Willy Zwaenepoel, Chairman
Professor
Computer Science

Dr. Sarita Adve
Assistant Professor
Electrical and Computer Engineering

Dr. John K. Bennett
Associate Professor
Electrical and Computer Engineering

Houston, Texas

May, 1995

Abstract

Message Passing Versus Distributed Shared Memory on Networks of Workstations

Honghui Lu

We compared the message passing library Parallel Virtual Machine (PVM) with the distributed shared memory system TreadMarks, on networks of workstations. We presented the performance of nine applications, including Water and Barnes-Hut from the SPLASH benchmarks; 3-D FFT, Integer Sort and Embarrassingly Parallel from the NAS benchmarks; ILINK, a widely used genetic analysis program; and SOR, TSP, and QuickSort.

TreadMarks performed nearly identical to PVM on computation bound programs, such as the Water simulation of 1728 molecules. For most of the other applications, including ILINK, TreadMarks performed within 75% of PVM with 8 processes. The separation of synchronization and data transfer, and additional messages to request updates for data in the invalidate-based shared-memory protocol were two of the reasons for TreadMarks's lower performance. TreadMarks also suffered from extra data communication due to false sharing. Moreover, PVM benefited from the ability to aggregate scattered data in a single message.

Acknowledgments

I would like to express my thanks to the members of my committee, Dr. Zwaenepoel, Dr. Adve, and Dr. Bennett. I also want to thank Sandhya, Pete, Alan, Dr. Schäffer, and Weimin. Their knowledge and experience, together with their patience, has been a great help to me. My thanks to Nenad and Edmar, for developing the parallel versions of the Embarrassingly Parallel benchmark and Integer Sort. We thank Dr. Lori A. Sadler and Dr. Susan H. Blanton for contributing disease family data for ILINK. Development of the RP data was supported by grants from the National Retinitis Pigmentosa Foundation and the George Gund Foundation. This research was supported in part by the National Science Foundation under Grant CCR-9116343, and by the Texas Advanced Technology Program under Grants 003604014 and 003604012.

Contents

Tables

Chapter 1

Introduction

Parallel computing on networks of workstations has been gaining more attention in recent years. Because workstation clusters use "off the shelf" products, they are cheaper than supercomputers. Furthermore, high–speed general–purpose networks and very powerful workstation processors are narrowing the performance gap between workstation clusters and supercomputers.

Since no physical memory is shared in workstation clusters, all communication between processes in such a system must be performed by sending messages over the network. Currently, the prevailing programming model for parallel computing on networks of workstations is message passing. Libraries such as PVM [GS92], TCGMSG [Har90] and Express [Par92] were developed at different research institutions. A message passing standard MPI [Mes94] has also been published. With the message passing paradigm, the distributed nature of the memory system is fully exposed to the application programmer. The programmer needs to keep in mind where the data are, decide *when* to communicate with other processes, *whom* to communicate with, and *what* to communicate. This makes programming for message passing paradigm hard, especially for large applications with complex data structures.

Recent distributed shared memory(DSM) systems[Li86, BCZ90, BZ91, KDCZ94] provide a shared memory abstraction on top of message passing in workstation clusters. An application programmer can write the program as if it is executing in a shared memory multiprocessor and access shared data with ordinary read and write operations. The chore of message passing is left to the underlying DSM system to handle. However, DSM systems are less efficient than message passing systems. This is because under the message passing paradigm, communication is handled by the programmer, who has complete knowledge of the data usage pattern. Under the DSM paradigm, the DSM system has little knowledge of the application program, and therefore must be conservative in determining when to communicate data. Since sending messages between workstations is very expensive, additional communication is the major drawback resulting in poor DSM performance.

Although much work has been done to improve the performance of DSM systems in the past decade, DSM systems still do not have the same popularity as do the message passing systems. Most published performance evaluations of DSM systems only show the results of toy programs with simple data access patterns. We need to run DSM systems on a large variety of more practical applications, and compare them with equivalent message passing systems, both to show the potential of DSM systems, and to determine the causes of the lower performance of DSM systems. This information can then be used to further improve current DSM systems.

In this paper, we compare the message passing system PVM [GS92] with the DSM system TreadMarks [KDCZ94]. We ported nine parallel programs to both TreadMarks and PVM, and ran them on eight DECstation-5000/240 workstations connected by a 100Mbits per second ATM network. The programs are Water and Barnes-Hut from the SPLASH benchmark suite [SWG92]; 3-D FFT, Integer Sort and Embarrassingly Parallel(EP) from the NAS benchmarks [BBLS91]; ILINK, a widely used genetic analysis program; and SOR, TSP, and QuickSort.

The performance results show that TreadMarks performs well in comparison to PVM on some practical problems. In the Water simulation of 1728 molecules, TreadMarks achieves 99% of the performance of PVM. For ILINK, with 8 processes, TreadMarks's speedup is 87% of that in PVM. In general, the TreadMarks versus PVM performance ratio is closely related to computation/communication ratio and the granularity of shared data. For programs with a high computation/communication ratio, and large granularity of sharing, such as EP, SOR, and the Water simulation of 1728 molecules,TreadMarks and PVM have nearly identical performance. For programs with little access locality and a large amount of shared data, such as Barnes-Hut, PVM performs about twice as well as TreadMarks with 8 processes. Most of our programs lie between these two extremes. For these programs, the TreadMarks speedup with 8 processes is 76% to 87% of that in PVM. The separation of synchronization and data transfer, and additional messages to request updates for data in the invalidate based shared-memory protocol are two of the reasons for TreadMarks's lower performance. TreadMarks also suffers from extra data communication due to false sharing. In addition, PVM benefits from the ability to aggregate scattered data in a single message, an access pattern that would result in several miss messages in the invalidate-based TreadMarks protocol.

In terms of programmability, since most of our test programs are relatively simple, it was not difficult to port them to PVM. However, for two of the programs, namely

3 - D FFT and ILINK, the message passing versions were significantly harder to develop than the DSM versions. We will discuss our experience with the programs later in this thesis.

The rest of this thesis is organized as follows. In Chapter 2 we briefly describe the two paradigms, and give an overview of PVM and TreadMarks. The application programs and their parallel versions are described in Chapter 3. The performance results are presented in Chapter 4. Chapter 5 presents the conclusions.

Chapter 2

Message Passing Versus DSM on Workstation Clusters

Message passing and distributed shared memory are the major paradigms for parallel programming on networks of workstations. With the message passing paradigm, the application programmer must keep in mind that there is no memory accessible by multiple processes. To share data between processes, the application programmer must write codes that explicitly exchange messages. Programming with message passing is hard, because the application programmer must decide when to communicate with other processes, which process to communicate with, and what data to communicate.

The distributed shared memory (DSM) systems provide the application programmer with the image of a shared memory. An application programmer can write the program as if it is executing in a shared memory multiprocessor and leave the DSM system to handle the underlying message passing. It is easier to program with the shared memory paradigm, especially when the algorithm is very complicated, because the programmer can concentrate more on the algorithm side rather than on moving data among processes.

Although it is more complicated to program with message passing, message passing is more efficient than DSM. With some programmer effort, communication in message passing is less frequent than in DSM. Since it is very expensive to pass messages between user level processes on different workstations, DSM systems must be highly optimized to avoid additional communication. In this section, we introduce two effective methods to reduce the number message in run time DSM systems – relaxed memory consistency models [AH93] and multiple-writer protocols [BCZ90]. Both of them are used in TreadMarks.

The rest of this chapter is organized as follows. A brief introduction to programming styles in the two paradigms, along with some simple examples are given in

Section 2.1. Section 2.2 introduces the two systems analyzed in this thesis – the message passing system PVM, and the distributed shared memory system TreadMarks.

2.1 Different Programming Styles in Message Passing and DSM

2.1.1 Message Passing

With the message passing paradigm, the only way for parallel processes to communicate with each other is to exchange messages via the network. The basic primitives in the message passing paradigm are *send* and *receive*.

A *send* is used to send a message containing data from one process to another. The send can be either *blocking* or *nonblocking*. A process executing a nonblocking send continues processing immediately after dispatching the message. In a blocking send, the process waits until the message has been received by the other process. In general, a blocking send is used only in the presence of unreliable communications, or when it is important to send the messages in a particular order.

A *receive* is used to read a message sent from another process. The receive can also be either *blocking* or *nonblocking*. A blocking receive waits until the message arrives. A nonblocking receive returns if the message is not available. The nonblocking receive can be used for asynchronous input, in which the process repeatedly checks for the presence of an incoming message. Depending on the availability of the message, the process can either read the message or do some other work.

The best way to get a feeling for what the message passing paradigm implies is to look at some simple programs. In the message passing versions of these programs, we assume that there is an initialization procedure to start up tasks on different machines. After the initialization, each process knows its process identifier, `proc_id`, and the number of processes in the system, `nprocs`. The variable `proc_id` ranges from `0` to `nprocs-1`. We also assume the following syntax for send and receive: `send(proc_id, start_address, length)` and `receive(proc_id, start_address, length)`. The semantics of `send` is to send to `proc_id` the `length` elements starting from `start_address`. In a receive, the process receives `length` elements from `proc_id`, and loads them into memory beginning at `start_address`.

The first example is Successive Over-Relaxation (SOR). It solves partial differential equations. A simple form of SOR iterates over a two-dimensional array. During each iteration, every matrix element is updated to the average of the values of its

nearest neighbors (above, below, left and right). Two separate arrays are used. One array `new` is used to store the newly computed values, the other array `old` is used to store the initial values or results from the last iteration. The array `new` is copied to `old` at the end of each iteration. The sequential SOR program is shown in Figure 2.1. The message passing version of SOR appears in Figure 2.2. Each process is assigned to work on approximately the same number of consecutive rows. At the beginning of the program, process 0 initializes the array `old` and distributes it to other processes. At the end of each iteration, a process exchanges its highest and/or lowest numbered rows with its neighbors. The `send` and `receive` here are blocking.

Another example with a bit more complicated communication is integer addition, which sums an array `A` of 1,000,000 integers. The sequential code of integer addition is shown in Figure 2.3 . A straightforward method to parallelize this algorithm is to divide the array into equal-sized bands, assigning one to each process. Each process computes the sum of the sub-array into a variable `LocalSum`. The values of `LocalSum` are added together to `GlobalSum` at the end. The message passing version of integer addition is in Figure 2.4. At the beginning of the program, process 0 initializes array `A` and distributes it to other processes. Each process then sums its sub-array into `LocalSum`. At the end of the program, each process sends its value of `LocalSum` to process 0, and process 0 sums them up. Both `send` and `receive` in this program are blocking.

2.1.2 Distributed Shared Memory

Distributed Shared Memory(DSM) provides the programmer with the abstraction of a globally shared memory. The fact that the memory is distributed is hidden from the user. DSM requires a set of primitives different from those used in message passing. First, there is a need to distinguish data that are private to each process from those shared by all processes. Second, because data is shared, synchronization is required to prevent out-of-order accesses to shared variables. For instance, a *critical section* contains code that can only be accessed by one process at a time. Critical sections can be used for reduction operations such as summing values into a global variable. A *barrier* is a point in the program where all processes must have arrived before any one can proceed. Barriers are used to keep the processes working in lock step.

```
float new[M][N], old[M][N];
main()
{
   Initialize old;
   for C iterations {
       for (i=1; i++; i<M)
           for (j=1; j++; i<N)
               new[i][j] = (old[i-1][j]+old[i+1][j]+old[i][j-1]+old[j][i+1])/4;

       for (i=0; i++; i<M)
           for (j=0; j++; i<N)
               old[i][j] = new[i][j];
   }
} /* End of main */
```

Figure 2.1 Sequential SOR

Consider again the examples in the previous section. Besides the routine to start up processes, suppose we have `barrier()`, `begin_critical(i)` and `end_critical(i)`. The last two functions specify a critical section.

The DSM version of SOR appears in Figure 2.5. In the DSM version, the array `old` is shared, `new` is private. There are three barriers in the program. The first barrier is right before the start of computation. This barrier is used in order to make sure that data have been initialized by process 0 before other processes start computation. The second barrier occurs before copying `new` to `old`. Because `old` is shared, a process can not update its part of `old` until all its neighbors have finished the computation. The third barrier is at the end of each iteration. It is there to make sure that each process has finished updating its part of the shared array `old` before any of them can go on to the next iteration.

The DSM version of integer addition is shown in Figure 2.6. In the DSM version, the array `A` and the variable `GlobalSum` are shared. At the beginning of the program, process 0 initializes `A`, while the other processes are blocked waiting at a barrier. After the barrier, every process sums its part of the array into a private `LocalSum`. Finally, all the `LocalSum` variables are added to the shared variable `GlobalSum`. This operation must be serialized, i.e., protected by a critical section.

```
float new[M][N], old[M][N];
main()
{
   int len;
   Initialization();
   len = M/nprocs;
   if (proc_id == 0) {
       Initialize old;
       for (i=1; i<nprocs; i++) {
           send(i, &len, 1);
           send(i, new[len×i], len×N); }}
   else {
       receive(0, &len, 1);
       receive(0, new[len×proc_id], len×N); }

   /* All processes */
   /* Low and high are the lower and upper bounds of a process's rows */
   for C iterations {
       for (i=low; i<=high; i++)
           for (j=1; j<N; j++)
               new[i][j] = (old[i-1][j]+old[i+1][j]+old[i][j-1]+old[j][i+1])/4;
       for (i=low; i<=high; i++)
           for (j=0; j++; i<N)
               old[i][j] = new[i][j];

       send(proc_id-1, old[low], N);
       send(proc_id+1, old[high], N);
       receive(proc_id-1, old[low-1], N);
       receive(proc_id+1, old[high+1], N); }
} /* End of main */
```

Figure 2.2 Message Passing Version of SOR

```
#define N 1000000
int A[N];
main()
{
    int GlobalSum;
    Initialize A;
    GlobalSum = SumSub(A, N);
}

int SumSub(array, len)
int *array, len;
{
    int LocalSum, i;
    LocalSum = 0;
    for (i = 0; i < len; i++)
        LocalSum = LocalSum + array[i];
    return(LocalSum);
}
```

Figure 2.3 Sequential Integer Add

2.2 PVM Versus TreadMarks

In this thesis, we compare the performance of Parallel Virtual Machine (PVM [GS92]) with TreadMarks [KDCZ94]. PVM is a message passing system originally developed at Oak Ridge National Laboratory. It runs on Unix, and provides the programmer with a set of user level library routines. Although there exist other message passing systems such as TCGMSG [Har90], which provide higher bandwidth than PVM, we chose PVM because of its popularity. PVM version 3.2.6 is used in our experiments. TreadMarks is a software DSM system built at Rice University. Although many DSM implementations have been reported in literature [NL91], none of them is widely available. One reason is that many of them run on experimental operating systems, rather than general available operating systems, or require kernel modifications. Early DSM systems also suffer from performance problems. TreadMarks overcomes most of these problems. It is an efficient user level DSM system that runs on commonly available Unix systems.

```
#define N 1000000
int A[N];
main()
{
    int SUM, len, i, LocalSum;
    Initialization();
    len = N/nprocs;
    if (proc_id == 0){
        Initialize A;
        for (i = 1; i < nprocs; i++)
            send(i, &A[len×i], len); }
    else
        receive(0, A, len);

    LocalSum = SumSub(A, len);
    if (proc_id == 0) {
        SUM = LocalSum;
        for (i = 1; i < nprocs; i++) {
            receive(i, &LocalSum, 1);
            SUM = SUM + LocalSum; } }
    else
        send(0, &LocalSum, 1);
}
```

Figure 2.4 Message Passing Version of Integer Addition

```
shared floatold[M][N];
float   new[M][N];
main()
{
    Initialization();
    if (proc_id == 0)
        initialize old;
    barrier();

    for C iterations {
        for (i=low; i<=high; i++)
            for (j=1; j<N; j++)
                new[i][j] = (old[i-1][j]+old[i+1][j]+old[i][j-1]+old[j][i+1])/4;
        barrier();

        for (i=low; i<=high; i++)
            for (j=0; j++; i<N)
                old[i][j] = new[i][j];
        barrier(); }
} /* End of main().  */
```

Figure 2.5 DSM Version of SOR

```
#define N 1000000
shared int A[N];
shared int GlobalSum;
main()
{
    int len, i, LocalSum;
    Initialization();
    GlobalSum = 0;
    if (proc_id == 0)
        initialize A;
    /* All processes */
    barrier();
    len = N/nprocs;
    LocalSum = SumSub(&A[proc_id × len], len);
    begin_critical();
        GlobalSum = GlobalSum + LocalSum;
    end_critical();
    barrier(); }
} /* End of main() */
```

Figure 2.6 DSM Version of Integer Addition

2.2.1 PVM

PVM Interface PVM allows heterogeneous computers in a network to appear as a single concurrent computational resource. It provides data type abstraction and buffers for messages. PVM assigns a unique task identifier to every process in a virtual machine. There are both send and receive buffers. A send dispatches the contents of the send buffer to its destination. A receive places an incoming message in a receive buffer. User data are packed into the send buffer before sending and unpacked from the receive buffer after receiving. Data types are specified at these times. PVM has both C and FORTRAN libraries. We focus on C because that is the language we used in application programs.

In PVM, the parent process uses `pvm_spawn()` to start children on different machines. The format is: `int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)`. The routine `pvm_spawn()` starts up `ntask` copies of an executable file `task` on the virtual machine. The parameters `flag` and `where` are used together to specify what architectures and machines to use. Each process is assigned a unique task identifier in the virtual machine, `tid`. The `tid`s are used to specify source and destination of a message in message passing. The list of `tid`s are returned in `tids`. The number of processes created is returned in `numt`.

PVM provides the routine `pvm_initsend()` to clear the old send buffer and create a new one: `int bufid = pvm_initsend(int encoding)`. The new buffer identifier is returned in `bufid`. The variable `encoding` specifies the encoding scheme, which will be discussed in the implementation of PVM.

The routines `pvm_pkType()` are used to pack data into the send buffer. The `Type` can either be `byte` for characters, `int` for integers, or `float` for floating point numbers, etc. They all have the same syntax: `int info = pvm_pkType(Type *ptr, int nitem, int stride)`. The variable `ptr` is a pointer to the first element to be packed, and `nitem` is the total number of items to be packed. The variable `stride` is the stride to use when packing.

The routines `pvm_send()` and `pvm_mcast()` send messages. Their formats are: `int info = pvm_send(int tid, int msgtag)` and `int info = pvm_mcast(int *tids, int ntask, int msgtag)`. The routine `pvm_send()` labels the message with an integer identifier `msgtag` and sends it to the process `tid`. The routine `pvm_mcast()` multicasts the message to all processes specified in the integer array `tids` except

itself. The `tids` array is of length `ntask`. Both `pvm_send` and `pvm_mcast` are non-blocking.

There are both blocking and nonblocking receives in PVM. They are `pvm_recv()` and `pvm_nrecv()` respectively:
`int bufid = pvm_recv(int tid, int msgtag)`, `int bufid = pvm_nrecv(int tid, int msgtag)`. The routine `pvm_recv()` waits until a message with label `msgtag` has arrived from `tid`. A value of `-1` in `msgtag` or `tid` will match any tag value or any process identifier, respectively. It then clears the previous receive buffer, places the message in the newly created receive buffer, and returns the receive buffer identifier in `bufid`. The nonblocking `pvm_nrecv()` returns `0` in `bufid` if the expected message has not arrived. The routine `pvm_nrecv()` does the same thing as `pvm_recv()` if a message with label `msgtag` has arrived from `tid`. This routine can be called multiple times to check the presence of the same message, while performing other work between calls. When there is no useful work to do, `pvm_recv()` can be called for the same message.

Data in the receive buffer are unpacked into a user data structure by using `pvm_upkType()`. The routines `pvm_upkType()` are similar to `pvm_pkType()`: `int info = pvm_upkType( Type *ptr, int nitem, int stride)`. The variable `ptr` points to the address of the first element to be unpacked. The unpack should match the corresponding pack calls in types and number of items.

The PVM version of SOR appears in Figure 2.7. Because the entire program is too long to fit in a simple example, details of the initialization subroutine and computation are omitted. During initialization, after the master process spawns the slaves, it broadcasts the number of processes started and an array containing task identifiers of the processes. Then the master process initializes the array `old` and distributes it to the other processes.

PVM Implementation PVM consists of two parts: a daemon process on each host and a set of library routines. The daemons connect with each other by UDP, and a user process connects with its local daemon using TCP. If a user process p_1 on host H_1 wants to send a message to another user process p_2 on host H_2, the usual way to do this is through the daemons on their hosts. However, p_1 and p_2 can set up a direct TCP connection between them in order to reduce overhead. We use direct connections between the user processes in our experiments, because it gives better performance.

```
float old[M][N], new[M][N];
main() {

     Initialization();
     /* All processes */
     for C iterations {
         msgtag = C ;
         for my rows of matrix {
             compute values of new;
             copy new to old; }

         /* Send to upper neighbor */
         if (proc_id != 0) {
             pvm_initsend(PvmDataRaw);
             pvm_pkfloat(&old[my_start_row][0], N, 1);
             pvm_send(tids[proc_id-1], msgtag++); }

         /* Send to lower neighbor */
         if (proc_id != nprocs-1) {
             pvm_initsend(PvmDataRaw);
             pvm_pkfloat(&old[my_end_row][0], N, 1);
             pvm_send(tids[proc_id+1], msgtag); }

         /* Receive from lower neighbor */
         if (proc_id != 0) {
             pvm_recv(tids[proc_id+1], msgtag);
             pvm_upkfloat(&old[my_end_row+1][0], N, 1);}

         /* Receive from upper neighbor */
         if (proc_id != nprocs-1) {
             pvm_recv(tids[proc_id-1], msgtag);
             pvm_upkfloat(&old[my_start_row-1][0], N, 1);}
     }
} /* End of main */
```

Figure 2.7 PVM version Version of SOR

Because PVM is designed to work on a set of heterogeneous machines connected by the network, it provides conversion to and from external data representation (XDR). This conversion is avoided if all machines in the PVM are identical.

In PVM, data are usually packed into a message buffer and remain there until they are dispatched. At the receiving end, the entire message is buffered until a receive call accepts it. There exists in-place packing in the newest version of PVM, but it is restricted to data with a stride of 1. The new version is not very stable on our system, so we decided to use message buffers instead.

2.2.2 TreadMarks

TreadMarks Interface TreadMarks is a software distributed shared memory system that allows shared memory programs to run on a cluster of workstations connected by general–purpose networks such as the Ethernet. The fact that the memory in the system is physically distributed is transparent to the user.

TreadMarks provides the user with two variables, `Tmk_nprocs` and `Tmk_proc_id`. The variable `Tmk_nprocs` specifies the number of processes in this system. The variable `Tmk_proc_id` is unique for each process, it ranges from 0 to `Tmk_nprocs-1`.

The routine `Tmk_startup()` starts up processes and initializes TreadMarks data structures. A call to this routine must precede all other TreadMarks calls. After `Tmk_startup()`, the contents of both private and shared memory are identical across all processes in the system, except that the values in `Tmk_proc_id` are different for each process. The function `Tmk_startup` takes no parameters. However, the number of processes and host names can be specified on the command line.

There is no statically allocated shared memory segments in TreadMarks, shared memory must be allocated dynamically. The routines `char *Tmk_malloc(int size)` and `char *Tmk_sbrk(int size)` allocate memory on shared memory. If the pointers to shared memory structures are in private memory, the user can use `Tmk_distribute` to distribute values of these pointers to all the other processes in the system. The routine `Tmk_distribute(char *addr, int size)` sends `size` bytes of private memory at address `addr` to all processes in the system, so that they all have the same value at this address in private memory.

Application threads synchronize via two primitives: barriers and exclusive locks. Barriers are used to synchronize all processes. Exclusive locks are used to control accesses to critical sections. The routine `Tmk_barrier(int num)` stalls the calling

process until all processes in the system have arrived at the same barrier. Barrier indices `num` are integers in a certain range. Locks are used to control access to critical sections. A lock is *acquired* before entering a critical section and *released* after the critical section is finished. The routine `Tmk_lock_acquire(int num)` acquires a lock for the calling process, and the routine `Tmk_lock_release(int num)` releases it. No process can acquire a lock if another process is holding it. Integer `num` is a lock index assigned by the programmer.

TreadMarks guarantees memory consistency only at certain synchronization points in order to reduce communication among processes. It is imperative to use built-in synchronizations in TreadMarks rather than rolling your own. In particular, neither spin locks nor setting and checking flags in shared memory works, because data is only moved from node to node in response to explicit TreadMarks synchronizations.

Furthermore, TreadMarks features a variant of *release consistent*(RC) shared memory model [GLL+90]. With this programming abstraction, memory accesses are divided into normal accesses and synchronization accesses. Synchronization accesses are further divided into *acquires* and *releases*. The basic idea is that shared memory modifications by a process p_1 only need to become visible to another process p_2 when a subsequent *release* of p_1 becomes visible at p_2 via some chain of mutual synchronizations. This programming abstraction is slightly different from that of a shared memory multiprocessor, because changes to the shared memory do not go anywhere until a release is performed.

In TreadMarks, lock acquires and barrier departures are modeled as *acquires*, lock releases and barrier arrivals are modeled as *releases*. In barriers, shared memory modifications by a process before a barrier are guaranteed to be visible to other processes only after the barrier. For example, in Figure 2.8 the second read in p_2 returns 4. The first read does not return 4 even if in "wall clock time" the write precedes that read. For locks, look at the example in Figure 2.9. The read by p_2 after the lock acquire will return a value of 4. However, if either p_1's release or p_2's acquire is not present, the value returned by the read may not be 4 even though in "wall clock time" the write precedes the read.

The TreadMarks version of SOR is given in Figure 2.10. The array `old` is allocated on shared memory. The explicit message passing at the end of each iteration in the PVM version is replaced by a call to `Tmk_barrier()`.

```
P1                      P2
w(x) 4                  r(x) ?
Tmk_barrier(1)          Tmk_barrier(1)
                        r(x) 4
```

Figure 2.8 Release Consistency With Barriers

```
P1                      P2
w(x) 4
lock_release(l)         r(x) ?
                        lock_acquire(l)
                        r(x) 4
```

Figure 2.9 Release Consistency With Locks

```
float *old[M], new[M][N];
main()
{
    Tmk_startup();
    /* Master process initializes A1 */
    if (Tmk_proc_id == 0)
        for (i = 0; i<N; i++) {
            old[i] = (float *)Tmk_malloc(sizeof(float)*N);
            Tmk_distribute(old[i], sizeof(float*));
        initialize old; }
    Tmk_barrier(1);

    for C iterations {
        for my rows of matrix {
            Compute values of new;
            Tmk_barrier(2);
            copy new to old; }
        Tmk_barrier(3); }
}
```

Figure 2.10 TreadMarks Version of SOR

TreadMarks Implementation TreadMarks relies on the operating system's virtual memory page protection mechanism to detect accesses to the shared pages. In TreadMarks, processes communicate either through UDP on an Ethernet or an ATM LAN, or through AAL3/4 on an ATM LAN. Because sending message between two processes is very expensive, TreadMarks takes great effort to minimize synchronization messages.

Release Consistency and Multiple-Writer Protocol

TreadMarks provides the user with the illusion of a globally shared memory and does the underlying message passing to keep the shared memory consistent. It is TreadMarks's responsibility to decide when to send messages, what to send, and whom to send to. In order to keep the shared memory consistent, one way would be sending out messages whenever writing to a shared variable that is also remotely cached. This method is implemented by most snoopy-cache, bus-based multiprocessors, but DSM systems cannot afford such a high communication rate because of the high overhead per message.

As mentioned before, TreadMarks features a *release consistent* (RC) [DKCZ93, AH93] shared memory model. In the RC model, shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a process p to become visible to another process q only when a subsequent release by p becomes visible to q via some chain of mutual synchronizations. In practice, this model allows a process to buffer multiple writes to shared data in its private memory until the release. In TreadMarks, `Tmk_lock_acquire(i)` is modeled as an acquire, and `Tmk_lock_release(i)` is modeled as a release. `Tmk_barrier(i)` is modeled as a release followed by an acquire, where each process releases at barrier arrival, and acquires at barrier departure.

False sharing also causes frequent communication. False sharing occurs when two or more processes access different variables within the same page, with at least one of the accesses being a write. If only one process is allowed to write to the page, the shared page will ping-pong back and forth among processes, because a write to any variable of a page causes the entire page to become invalid on all other processes that cache the page. A subsequent access by any of these processes incurs an access miss and causes the modified copy to be brought in over the network. The original copy of

the page may suffice in this case, because the write was to a variable different from the one that was accessed locally. This problem occurs in snoopy-cache multiprocessors as well, but it is more prevalent in software DSM because the consistency protocol operates on pages that are much larger than cache blocks.

TreadMarks uses a *multiple-writer* protocol to address this problem. With the multiple-writer protocol, two or more processes can simultaneously modify their own copy of the shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing. In order to distinguish changes made by different processes, at the time the process sends out update to the shared page, instead of sending the whole page, only the modified values are sent. Those modified values are called *diffs*, meaning difference between the modified page and the page before the modification.

A Lazy Invalidate Implementation of Release Consistency

TreadMarks implements a *lazy invalidate* version of release consistency [KCZ92]. The propagation of modifications is postponed until the time of the *acquire*. The releaser is *lazy*, it does not propagate modifications to the shared pages. Furthermore, instead of sending new data to the acquirer, the releaser notifies the acquirer of which pages have been modified, causing the acquirer to *invalidate* its local copies of these pages. A process has an page fault on the first access to an invalidated page, and gets diffs for that page from previous releasers.

The acquiring process determines which modifications it needs to see according to the definition of RC. To do so, the execution of each process is divided into *intervals*. A new interval begins every time a process executes a release or an acquire. Each process has an *interval index*, which is incremented every time a new interval starts on this process. Intervals on different processes are partially ordered [AH93]: (i) intervals on a single process are totally ordered by program order, (ii) an interval on process p *precedes* an interval on process q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p, and (iii) an interval precedes another interval by transitive closure. In locks, the interval corresponding to the release of a lock directly precedes the interval beginning with a subsequent acquire to the same lock. In barriers, any interval corresponding to the barrier arrival precedes all intervals corresponding to the subsequent barrier departures. However, no ordering exist among the barrier arrivals, or among the barrier departures.

Each interval has a vector *timestamp* to record its knowledge of intervals in other processes that precede it. A timestamp contains an entry for each process. For example, in the timestamp of the ith interval of process p, the entry for process p is equal to i. The entry for process q other than p denotes the most recent interval of process q that precedes interval i of process p according to the partial order.

RC requires that before a process p may continue past an acquire, the updates of all intervals preceding the current interval must be visible at p. Therefore, at an acquire, p sends its current interval timestamp to the previous releaser q. The releaser then compares the corresponding entries of both timestamps, and sends a message to p including *write notices* for all intervals named in q's current interval timestamp but not in the timestamp it received from p. Process p computes a new vector timestamp according to the pair-wise maximum of its previous timestamp and the releaser's timestamp. A *write notice* is an indication that a page has been modified in a particular interval.

Implementation Details

TreadMarks uses *diffs* to record modifications to a page made by different processes. In order to capture the modifications to a shared page, it is initially write protected. At the first write, a protection violation occurs. The TreadMarks makes a copy of the page (a *twin*), and removes the write protection so that further writes to the page can occur without any TreadMarks intervention. The *twin* and the current copy can later be compared to create a *diff*. At a release, a write notice is created for each page that was twinned since the last remote synchronization.

In TreadMarks, each lock has a statically assigned manager. The manager records which process has most recently requested the lock. All lock acquire requests are directed to the manager, and, if necessary, forwarded to the process that last requested the lock. A lock acquire request contains the current timestamp of the acquiring process. When the lock is released, the releaser informs the acquirer of all intervals between the vector timestamp in the acquirer's lock request message, and the releaser's current vector timestamp. The acquiring process then invalidates all pages for which write notices were received. A lock acquire takes up to 3 messages. A lock release does not incur any message in this lazy implementation.

Barriers have a centralized manager. At the barrier arrival, each client informs the barrier manager of its timestamp and all write notices created since the last time

that the client and the manager synchronized. The manager sets its new timestamp according to the pair-wise maximum of its previous timestamp and the timestamps of the clients. When all processes have arrived at the barrier, the manager then informs all clients of write notices of all intervals between their vector timestamp and the manager's new timestamp. The clients then invalidate the pages for which write notices were received. The number of messages sent in a barrier is $2(n-1)$, where n is the number of processes.

On an access miss, if the faulting process does not have a copy of the page, it requests a copy from a member of the page's approximate copyset. The approximate copyset for each page is initialized to contain process 0. If write notices are present for the page, the faulting process obtains the missing diffs and applies them to the page in increasing timestamp order. It is usually unnecessary to send diff requests to all the processes who have modified the page, because if a process has modified a page during an interval, it must have all the diffs of all intervals that precede it, including those from other processes. In TreadMarks, all write notices without corresponding diffs are examined. TreadMarks then sends diff requests to the subset of processes for which their most recent interval is not preceded by the most recent interval of another process.

2.3 Summary

This chapter introduced two programming paradigms for parallel computing on networks of workstations, namely message passing and distributed shared memory(DSM). In message passing, messages are exchanged explicitly in application programs. The DSM systems provide a shared memory abstraction on top of message passing. The DSM paradigm is easier to program with than the message passing paradigm, but also incur more overhead than the latter.

We discussed user interface and implementation of the message passing system PVM [GS92] and the DSM system TreadMarks [KDCZ94]. They are both user level libraries running on commonly available Unix systems. PVM allows heterogeneous computers in a network to appear as a single concurrent computational resource. It provides data type abstraction and buffering for messages. TreadMarks features release consistency and multiple-writer protocol to reduce communication in DSM systems.

Chapter 3

Application Programs

In this chapter, we present both the PVM and the TreadMarks versions of nine programs. The programs are Successive Over-Relaxation (SOR), Traveling Salesman Problem (TSP), and QuickSort (QSORT); Water and Barnes-Hut from the SPLASH benchmarks [SWG92]; Embarrassingly Parallel (EP), Integer Sort (IS) and three dimensional FFT (3-D FFT) from the NAS benchmarks [BBLS91]; and ILINK, which is a widely used genetic linkage analysis program. For each program, we tried to achieve the best performance for each paradigm.

3.1 EP: An Embarrassingly Parallel Benchmark

EP [BBLS91] is a heavily computation bound benchmark. EP generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive square annuli. This problem is typical of many Monte-Carlo simulation applications.

The program first generates $2n$ ($n = 2^{24}$ in our test) pseudo random floating point values r_i in the interval $(0, 1)$ for $1 \leq i \leq 2n$. Then for $1 \leq j \leq n$, set $x_j = 2r_{2j-1} - 1$ and $y_j = 2r_{2j} - 1$. Next set $k = 0$, and beginning with $j = 1$, test if $t_j = x_j^2 + x_j^2 \leq 1$. If not, reject this pair and proceed to the next j. If this inequality holds, then set $k = k + 1$, $X_k = x_j\sqrt{(-2\log t_j)/t_j}$ and $Y_k = y_j\sqrt{(-2\log t_j)/t_j}$. Approximately $n\pi/4$ pairs will be constructed in this manner. Finally, for $0 \leq l \leq 9$, tabulate Q_l as the count of the pairs (X_k, Y_k) that lie in the square annulus $l \leq \max(|X_k|, |Y_k|) \leq l + 1$, and output the ten Q_l counts.

Because separate sections of the uniform pseudo random numbers can be independently generated by each process, this program can be parallelized so that the only requirement for communication is the combination of the ten Q_l sums from various processes at the end.

In the TreadMarks version of EP, there is a shared array of Q_l sums, and each process also has a private array of Q_l. The Q_l sums are accumulated locally and added

to the shared array only at the end of the program. Updates to the shared sum are protected by a lock, and the processes wait at a barrier after the modification. After the barrier, the master prints out the result.

In the PVM version, each process has a Q_l sum in its private memory. At the end of program, each process sends its Q_l array to the master. The master process adds them together and outputs the result.

3.2 Red-Black SOR

Successive Over-Relaxation (SOR) is a method of solving partial differential equations. Our test program iterates over a two dimensional array. During each iteration, every matrix element is updated to the average of its nearest neighbors (above, below, left and right).

In red-black SOR, the elements are painted either red or black, such that all the nearest neighbors of a red element are black and vice versa. Each iteration is divided into two phases. The red elements are updated in the first phase, and the black elements are updated in the second phase. In practice, each row in the array is split into two adjacent rows, with one containing all the red elements and another containing all the black elements in the row. We have described a naive SOR algorithm in chapter 2. Compared to the naive algorithm, red–black SOR does not need a scratch array, and converges faster. In the second phase, all the red neighbors of a black point contain new values from the first phase, unlike in the naive algorithm, where only the old values from the last iteration are used.

SOR is parallelized by dividing the matrix into roughly equal size bands of rows, assigning each band to different process. During an iteration, every process works on its own band and synchronizes with others by a barrier at the end of each phase. In the TreadMarks version, the matrix is allocated in shared memory, and processes synchronize with barriers at the end of each phase. With PVM, each process allocates its band of rows in its private memory and explicitly sends the shared row to its neighbor at the end of each phase.

3.3 IS: Integer Sort

Integer Sort (IS) [BBLS91] requires ranking an unsorted sequence of N keys. The *rank* of a key in a sequence is the index value i that the key would have if the sequence

of keys were sorted. All the keys are integers in the range $0 \leq x \leq B_{max}$ and the method used is counting, or bucket sort.

The parallel version of IS divides up the keys among processes. First, each process counts its keys and writes the result in the private bucket. Then, the values in the private buckets are summed up. At last, the processes read the sum and rank their keys. The amount of computation required for this benchmark is relatively small – linear in the size of the array. The amount of communication is proportional to the size of the key range, because the bucket is passed around among processes.

In the TreadMarks version of IS, the only shared structure is a bucket. Besides the shared bucket, each process also has a private bucket and a private array containing keys owned by this process. After counting locally, a process locks the shared bucket, adds the values of its private bucket to the shared bucket, releases the lock, and waits at a barrier until all others have finished their updates. Each process then reads the final result in shared bucket and ranks its keys.

In the PVM version of IS, each process has a bucket and its part of the keys in private memory. After counting locally, the processes form a chain, in which process 0 sends its local bucket to process 1, process 1 adds the values in its local bucket to the values in the bucket it receives and forwards the result to the next process, etc. The last process in the chain calculates the final result and broadcasts it to all the other processes.

3.4 TSP

The Traveling Salesman Problem (TSP) finds the shortest path that starts at a designated node, passes through every other node exactly once and returns to the original node. A complete path is known as a tour.

A brute force algorithm would try all possible path permutations and select the shortest one, but the time to check all possible permutations is prohibitive. A simple optimization is to use a branch-and-bound algorithm. In this solution, if the length of a partial tour plus a lower bound of the remaining portion of the path is longer than the current shortest tour, the partial tour will not be explored further, because it cannot lead to a shorter tour than the current minimum length tour.

The version of TSP used in the evaluation maintains a priority queue of partially evaluated tours, with the one having the shortest lower bound on its length at the head. The evaluation of a partial tour is composed mainly of two procedures,

`get_tour` and `recursive_solve`. The subroutine `get_tour` deletes the most promising path from the queue. If the path contains more than a threshold number of cities, it returns this path. Otherwise, `get_tour` extends the path by one node, puts the promising ones generated by the extension back on the priority queue, and calls itself recursively. The subroutine `get_tour` returns either when the most promising path is longer than a threshold, or when the priority queue becomes empty. The procedure `recursive_solve` takes the path returned by `get_tour`, and tries all permutations of remaining nodes recursively.

In the TreadMarks version, the major shared data structures are the global minimum tour and its length, a tour array of structures describing both partially evaluated and unused tours, a priority queue containing pointers to partly evaluated tours, and a stack of pointers to unused tour structures. The `get_tour` is guarded by a lock to guarantee exclusive access to the priority queue and the stack. At the end of `recursive_solve`, if the process finds a shorter tour than the global minimum tour, it acquires the lock for global minimum tour and sets the new minimum value.

In the PVM version, the processes are divided into master and slaves. The master executes `get_tour` and keeps track of the optimal solution. The slaves execute `recursive_solve`. The slaves send requests to the master either to request solvable tours, or to update the shortest tour.

3.5 QuickSort

QuickSort (QSORT) is a recursive sorting algorithm that operates by repeatedly partitioning an unsorted input list into a pair of unsorted sublists, such that all elements in one of the lists are strictly greater than those in the other, and then recursively invokes itself on the two sublists. In the implementation used in our evaluation, the base case of the recursion happens when a list is sufficiently small, at which time it is sorted directly using bubble-sort. The QSORT is parallelized using a work queue that contains descriptions of unsorted sublists, from which worker thread continuously removes the lists.

In the TreadMarks version of QSORT, the major data structures are: an array to be sorted, a task queue that contains range indices of the unsorted sub-array, and a count of the number of worker threads blocked waiting for work. The TreadMarks version of QSORT differs from the TreadMarks version of TSP in that the QSORT worker releases the task queue without subdividing the sub-array it remove from the

queue. The QSORT worker may further divide the sub-array. It then reacquires control of the task queue and places the newly generated sub-arrays back into the task queue. In contrast, TSP workers do not relinquish control of the task queue until they have removed a partial tour that can be solved directly. As a consequence, the task queue in QSORT is accessed more frequently per unit of computation.

In the PVM version, the master maintains the work queue and performs the partitioning on demand. The slaves send request messages to the master, which responds either with a sublist to be sorted directly or an indication that there is no more work. The slaves ship the sorted sublist along with the next request.

3.6 Water

Water [SWG92] is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a number of user-specified time steps. Both intramolecular and intermolecular potentials are computed in each iteration. In order to avoid computing all the $n^2/2$ pairwise interactions among molecules, a spherical cutoff range is used.

The main data structure used in water is an one-dimensional array of molecules called VAR. The structure of a molecule contains the center of mass of the molecule, plus for each of its three atoms the forces acting on them, their displacements, and the first six derivatives of the displacements. In order to make sure that each interaction is computed only once, only the interactions between a molecule and the $n/2$ molecules ahead of it in the VAR (wrapped around) are computed.

Water is parallelized by dividing VAR into equal segments, assigning one to each process, regardless of the possible load imbalance caused by cutoff radius. Most of the communication happens at the intermolecular force computation. At this time, a process reads the center of mass and the displacement of two molecules, computes the intermolecular force, and updates the force acting on each of the molecules. Only the center of mass, the displacement and the force in the molecule structure are accessed by multiple processes.

In the TreadMarks version of Water, the center of mass, the displacement and the force of the VAR are allocated in shared memory, the other variables in VAR are allocated in private memory. Each process also maintains a private copy of the forces. There is a lock associated with each process. At the time of intermolecular force computation, each process first zeros its private copy of the forces, and subsequent

changes to forces are accumulated locally into the private copy. Forces in the shared VAR are updated after all processes have finished their computation. If a process has updated molecules belonged to process i in the private copy, it acquires lock i and adds all modifications to molecules owned by process i. In the PVM version, the processes must exchange displacement and mass of molecules explicitly. The forces are also accumulated locally.

3.7 Barnes-Hut

Barnes-Hut [SWG92] simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which every body is modeled as a point mass and exerts forces on all other bodies in the system. If all pairwise forces are calculated directly, this has a complexity of $O(n^2)$ in the number of bodies, which is impractical for simulating large systems. Barnes-Hut is a hierarchical tree-based method that reduces the complexity to $O(n \log n)$.

The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The root of the tree represents a space cell containing all bodies in the system. The tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The result is a tree whose internal nodes are cells and leaves are individual bodies. The tree is therefore adaptive in that it extends to more levels in regions that have high particle densities. The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm starts at the root of the tree and conducts the following test recursively for every cell it visits: If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell, and the force this center of mass exerts on the body is computed. However, if the center of mass is not far enough away, the cell must be opened and each of its sub-cells visited.

The major data structures used in this algorithm are two arrays, one is the array of bodies that are leaves of the tree, the other is an array of internal cells in the tree. Load balance is achieved using the cost-zone method, where the cost of each body is defined by the work associated with it, and the cost of a cell is the sum of the costs of the bodies in the subtree rooted at this cell. The total work in the system is divided among processes so that every process has a contiguous, equal zone of work.

There are five major phases in each time step. In the sequential version, most of the time is spent in step 4.

1. MakeTree : Construct Barnes-Hut tree.
2. C_OF_M : Compute center of mass for each cell.
3. Get_my_bodies: Partition bodies among processes.
4. Force Computation: Compute forces on my own bodies.
5. Update: Update position and velocities of my bodies.

In the TreadMarks version, both body and cell arrays are shared. The subroutines MakeTree and C_OF_M are executed only by the master process, because these incur a lot of traffic if executed in parallel. There are barriers after C_OF_M and force computation. No synchronization is necessary during the force computation.

The PVM version looks much like the TreadMarks version. The MakeTree and C_OF_M are sequential, which means every process obtains all the bodies and creates a complete tree. Then each process goes on to do the next three steps on its private copy of the Barnes-Hut tree. At the end of each iteration, the processes broadcast their bodies.

3.8 3-D FFT

3-D FFT [BBLS91] numerically solves a partial differential equation using three dimensional forward and inverse FFT's. After initializing an array U of size $n_1 \times n_2 \times n_3$ with pseudo random numbers, we need to

Compute forward 3-D FFT of U, and call the result V.
Adjust V to obtain X
DO for some fixed number of iterations:
    Compute an inverse 3-D FFT on X.
    Output the checksum of X.

The 3-D FFT applies one dimensional FFT on each of the three dimensions. The parallel 3-D FFT algorithm we used is as follows. Assuming the input array $A_{n_1 \times n_2 \times n_3}$ is organized in row-major order. We distribute the array elements along the first dimension of A, so that for any i, all elements of the complex matrix $A_{i,j,k}$, $0 \leq j < n_2, 0 \leq k < n_3$ are contained within a single process. The 3-D FFT first

performs a n_3-point 1-D FFT on each of these $n_1 n_2$ complex vectors. Then it performs a n_2-point 1-D FFT on each of the $n_1 n_3$ vectors. No communication is needed up to now, because each of the n_3-point vector or the n_2-point vector is owned by a single process. Last, it transposes the resulting array into an $n_2 \times n_3 \times n_1$ complex array B and applies a n_1-point 1-D FFT on each of the $n_2 n_3$ complex vectors. The processes communicate with each other at the transpose, because each process owns a different set of elements after it.

In the TreadMarks version, both the original array A and the resulting array B are allocated in shared memory. Since the original array has been modified before the transpose, a barrier is called before each transpose to make sure that the new values can be propagated to other processes. In the PVM version, messages are sent explicitly at transpose. This includes figuring out where each part of the result array comes from, where each part of the original array goes to, and sending messages correspondingly. This index calculation on a 3-dimensional array is much more error-prone than simply swapping the indices.

3.9 ILINK

The sequential version of ILINK is a program from version 1.0 of the FASTLINK version [CIS93, SGSC94] of the LINKAGE [LLJO84] package for genetic linkage analysis. The LINKAGE/FASTLINK package contains programs used by geneticists around the world to find the approximate location of disease genes. ILINK is one program in the package. ILINK's input is one or more family trees showing phenotypes at some genes and the disease status for those individuals where it is known. It iteratively estimates a vector which represents a maximum likelihood estimate of how close the disease gene lies to the other specified genes on the genome.

Given a set of pedigrees and a fixed value of the probability, the outer loop of the likelihood evaluation iterates over all the pedigrees calculating the likelihood for each one. Within a pedigree, the program visits each nuclear family and updates the probability of each genotype for each individual. The update to each individual's probability is parallelized. The TreadMarks version of ILINK comes from [DSC+94], and we base our message passing version on it.

The main data structure in ILINK is a pool of `genarrays`, where each genarray is indexed by genotype numbers. The genarray is sparse, and an index array is associated with each one of them. Each update to an individual's genarray either

updates one parent conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings. Before each update, the master allocates a genarray from the pool of genarrays and initializes it for each person in the family. Then, the master assigns the nonzero elements in the parent's genarray to available processes in round robin fashion. Each process works on its share of nonzero values, and updates the individual's genarray accordingly. The master process summarizes the modifications made by each process at the end of each update.

In the PVM version of ILINK, each process has a local copy of genarray, and messages are passed explicitly between the master and the slaves at the beginning and the end of each update. Since the genarray is sparse, only the nonzero elements are sent. In TreadMarks, genarray is allocated in shared memory, and barriers synchronize the processes at the beginning and the end of the update to each person. The sparsity of genarray is addressed automatically by the TreadMarks system. Since only the nonzero elements are modified in each update, when the master collects updates at the end, the diffs it receives only contain the nonzero elements.

3.10 Summary

From our experience with PVM and TreadMarks, we conclude that it is easier to program using TreadMarks than using PVM. Although there is little difference in programmability for simple programs, for programs with complicated communication patterns, such as ILINK and 3-D FFT, it takes a lot of effort to figure out what to send and whom to send to.

Chapter 4

Performance Results

In this chapter, we present and analyze the performance results of both the PVM and the TreadMarks versions of the programs discussed in Chapter 3. The system used to evaluate PVM and TreadMarks consists of 8 DECstation-5000/240 workstations, each with a 40MHz MIPS R3000 processor and 24Mbytes of main memory. The workstations are connected to a high-speed ATM network using a Fore Systems TCA-100 network adapter card supporting communication at 100Mbits/second. The ATM interface connects point-to-point to a Fore Systems ASX-100 ATM switch, providing a high aggregate bandwidth because of the capability for simultaneous full-speed communication between disjoint workstation pairs. In TreadMarks, the user processes connect with each other with UDP. In PVM, processes set up direct TCP connections with each other. Both UDP and TCP are built on top of IP, with UDP being connectionless and TCP being connection oriented. TCP is a reliable protocol while UDP does not ensure reliable delivery. Using UDP for TreadMarks has the advantage of the ability to use light-weight, operation-specific, user-level protocols to ensure reliable delivery. For PVM, since the type of communication is not known, there is no significant disadvantage in using TCP over UDP.

Some of the runtime statistics are given. For the PVM versions, we counted the number of user level messages and amount of user level data. For the TreadMarks versions, we counted the total number of UDP messages, the total amount of data sent in these messages, and the amount of data sent for transferring diffs and pages.

4.1 EP: An Embarrassingly Parallel Benchmark

EP is completely compute-bound, so the overhead added by TreadMarks has little impact on performance. In our test, the program generates 2^{24} pairs of random numbers. This is much smaller than the 2^{28} pairs suggested in the NAS benchmark suites. The results are shown in Table 4.1 and Table 4.2. The TreadMarks version sends out much more data than the PVM version, because on the first access to

the shared Q_l, a process faults and brings in the whole page, rather than simply initializing the local array as is done by the PVM version. The additional messages in TreadMarks come from barriers, lock acquires, page and diff requests. In PVM, communication only happens at the end of the program, when all processes send their 40 byte Q_l to process 0. However, compared to the overall execution time, the communication overhead is negligible. With 8 processes, the program runs for 36 seconds, and only 7 messages, a total of 308 bytes are sent in PVM. Although TreadMarks sends more messages than PVM, the number of messages is still small at 86, and the total amount of data is only 33 Kbytes. Consequently, both TreadMarks and PVM achieve a speedup of 7.9 with 8 processes.

4.2 Red-Black SOR

We ran red-black SOR on a 512×2048 matrix of floating point numbers for 101 iterations. The first iteration is excluded from timing and statistics so as to eliminate cold start effect on our results. With this matrix size, the red or black elements of a row occupy exactly one page, so at the beginning of each phase, only one page fault is needed to get the red or black elements of the shared row.

In the first test, the elements in the middle of the array are initialized to zeroes, and those on the edges are ones. The results are shown in Table 4.3 and Table 4.4. Because of load imbalance, neither PVM nor TreadMarks achieves good speedup with 8 processes. The load imbalance happens because floating point computation involving zeroes takes longer time than others. The TreadMarks version performs within 2% of the PVM version. This is a consequence of the low communication rate in SOR, and the use of lazy release consistency in TreadMarks. Because of lazy release consistency, although the processes keep writing to the shared pages between two barriers, the number of messages sent in TreadMarks is always three times of

nprocs	2	4	6	8
Tmk	1.99	3.97	5.96	7.94
PVM	1.99	3.99	5.96	7.92
Tmk/PVM	1.0	1.0	1.0	1.0
Sequential time: 286 *sec*				

Table 4.1 EP speedup

nprocs		2	4	6	8
Tmk	Messages	9	33	62	86
	Total Bytes	4K	13K	23K	33K
	User Bytes	4K	13K	22K	31K
PVM	Messages	1	3	5	7
	Bytes	44	132	220	308

Table 4.2 EP Message and Data

that in PVM. Assume there are n processes, the number of shared rows is $2(n-1)$. In PVM, $2(n-1)$ messages are sent in each phase, because processes send the shared rows directly to their neighbors. In TreadMarks, since a request is sent to get the diff of a row, a total of $4(n-1)$ messages are sent to obtain the data. Besides, there are $2(n-1)$ messages sent in the barrier at the end of each phase. Altogether, $6(n-1)$ messages are sent in each phase, which is three times of those sent in PVM.

Notice that the amount of data sent in TreadMarks is much less than that in PVM. This is a result of multiple-writer protocol. Since all elements in the middle of the array are zeroes, changes are propagated from the edge to the center of the array. In TreadMarks, only the diffs are sent, compared to PVM, where the whole shared row is transferred. However, this has little effect on the performance, because wire time is negligible compared to the time to initiate a message.

We have also run SOR with all values in the matrix initialized to nonzero, such that all elements in the matrix are changed in each iteration. The results are shown in Table 4.5 and Table 4.6. The TreadMarks version and the PVM version send the same amount of data, and the number of messages sent in each version remains unchanged from the first test. Because the numbers are nonzero, the single process time drops from 279 seconds to 122 seconds. Due to better load balance, this test also has higher speedup than the first one. Consequently, communication rate has larger influence on performance. The TreadMarks version achieves 91% of the speedup of the PVM version with 8 processes, compared to 99% in the first test.

4.3 Integer Sort

We tested IS on two sets of parameters. In the first test, we sorted 2^{20} keys ranging from 0 to 2^7 for 10 iterations. In the second test, the keys range from 0 to 2^{15}, and

nprocs	2	4	6	8
Tmk	1.98	2.80	2.94	3.32
PVM	1.98	2.85	3.01	3.36
Tmk/PVM	0.99	0.98	0.98	0.99
Sequential time: 279 *sec*				

Table 4.3 SOR-Zero speedup

nprocs		2	4	6	8
Tmk	Messages(K)	1.2	3.6	6.0	8.4
	Total Bytes(KB)	157	801	2482	3613
	User Bytes(KB)	137	713	2292	3285
PVM	Messages(K)	0.4	1.2	2.0	2.8
	Bytes(KB)	1640	4920	8200	11480

Table 4.4 SOR-Zero Message and Data

nprocs	2	4	6	8
Tmk	1.88	3.52	5.05	6.30
PVM	1.97	3.77	5.48	6.91
Tmk/PVM	0.95	0.93	0.92	0.91
Sequential time: 122*sec*				

Table 4.5 SOR-Nonzero speedup

nprocs		2	4	6	8
Tmk	Messages(K)	1.2	3.6	6.0	8.4
	Total Bytes(KB)	1660	5008	8390	11808
	User Bytes(KB)	1640	4920	8200	11480
PVM	Messages(K)	0.4	1.2	2.0	2.8
	Bytes(KB)	1640	4920	8200	11480

Table 4.6 SOR-Nonzero Message and Data

we measured 5 iterations. The results are shown in tables 4.7 to 4.10. We did not try the 2^{23} keys, and 2^{20} buckets as suggested in NAS, because the extremely low computation/communication ratio is not suitable for workstation clusters.

The results show the potential performance degradation caused by diff accumulation in the current TreadMarks implementation. Assuming the bucket size is b, in PVM, the amount of data sent in each iteration is $2(n-1)b$. In TreadMarks, since updates to the shared bucket are protected by a lock, each process that modifies the shared bucket must get all the diffs created by previous modifiers in this iteration. The same thing happens after the barrier, where every process reads the final values in the shared bucket. At this time, each process gets all the diffs made by the processes who modified the shared bucket after it. These add up to $n(n-1)b$ in each iteration. We call this phenomenon *diff accumulation*. Although in each case, the process can obtain all the diffs from one process, this accumulation causes more messages to be sent when the sum of the diffs exceeds the maximum size of a UDP message. These results indicate that the diffing mechanism is not suitable for migratory data. Coalescing the diffs before sending them out would eliminate this problem.

Diff accumulation is not a serious problem when the bucket size is 2^7 integers, because 8 diffs of the bucket can be sent in one message. The overheads in the first test come mainly from synchronization messages and diff requests. Consequently, with 8 processes, the TreadMarks version sends out 4 times more data, 8 times more messages than the PVM version, and achieves 86% of the speedup in PVM. In the second test, since the bucket of 2^{15} integers is much larger than the message size, sending more diffs means sending more messages. Because of the high communication/computation ratio, with 8 processes, PVM achieves a speedup of 1.12, and TreadMarks achieves 48% of the speedup of PVM.

nprocs	2	4	6	8
Tmk	1.94	3.6	4.7	5.2
PVM	1.94	3.7	5.1	6.1
Tmk/PVM	1.0	0.98	0.93	0.86
Sequential time: 10 *sec*				

Table 4.7 IS speedup, $N = 2^{20}$, $B_{max} = 2^7$

nprocs		2	4	6	8
Tmk	Messages	165	483	823	1141
	Total Bytes(KB)	16	81	217	378
	User Bytes(KB)	14	72	197	348
PVM	Messages	20	60	100	140
	Bytes(KB)	10	31	51	71

Table 4.8 IS Message and Data, $N = 2^{20}$, $B_{max} = 2^7$

nprocs	2	4	6	8
Tmk	1.43	1.26	0.83	0.53
PVM	1.54	1.85	1.56	1.12
Tmk/PVM	0.93	0.68	0.54	0.48
Sequential time: 10 *sec*				

Table 4.9 IS speedup, $N = 2^{20}$, $B_{max} = 2^{15}$

nprocs		2	4	6	8
Tmk	Messages	705	2491	4981	8355
	Total Bytes(MB)	1.3	7.4	18.8	35.4
	User Bytes(MB)	1.3	7.4	18.7	35.2
PVM	Messages	10	30	50	70
	Bytes(MB)	1.3	3.9	6.6	9.2

Table 4.10 IS Message and Data, $N = 2^{20}$, $B_{max} = 2^{15}$

4.4 TSP

We solved a 18 city problem. The results are shown in Table 4.11 and Table 4.12. With 8 processes, TreadMarks sends 5 times more messages and 72 times more data than PVM, and achieves 84% of the speedup of PVM. The performance gap comes from the cost of accessing the priority queue.

In the PVM version of TSP, only the solvable tours and the minimum tour are exchanged between the slave and the masters. It takes 2 messages each for a slave to obtain a solvable tour, or to update the global minimum tour. The number of messages and the amount data sent in PVM changes little with increasing numbers of process, because the total amount of work remains stable.

The additional messages in TreadMarks come from both lock acquires and diff requests. Furthermore, in `get_tour`, it takes 3 page faults to obtain the priority queue, because it is composed of 3 distinct data structures, each of which takes more than one page. Overall, TreadMarks should send 4.5 times more messages than PVM, provided that every lock acquirer gets the lock from a remote process. This is not true with 2 processes, where 90 of the 165 locks are remote. Consequently, TreadMarks only sends 3 times more messages than PVM. With 8 processes, since 170 of the 200 locks are remote, PVM sends 5 times more messages than PVM. As for the amount of data, in TreadMarks, besides the additional data movement to get the shared tour structures, because accesses to the shared data are protected by a lock, on each access miss, a process gets all the diffs created since its last release of the lock. On average, this means getting diffs created by all the other processes.

nprocs	2	4	6	8
Tmk	1.73	3.07	4.03	4.74
PVM	2.01	3.6	4.82	5.63
Tmk/PVM	0.86	0.85	0.84	0.84
Sequential time: 27.1 *sec*				

Table 4.11 TSP speedup

nprocs		2	4	6	8
Tmk	Messages(K)	0.7	1.3	1.9	2.3
	Total Bytes(KB)	76	241	531	802
	User Bytes(KB)	67	217	492	748
PVM	Messages(K)	0.3	0.4	0.4	0.4
	Bytes(KB)	10	11	11	11

Table 4.12 TSP Message and Data

4.5 QuickSort

We used two sets of parameters, where the array size is 256K, and the bubble sort thresholds are 1024 and 512 respectively. The results are shown in Tables 4.13 to 4.16. Because the coarse-grained version runs for a longer amount of time, and sends less messages than the fine-grained version, it allows for better speedup than the latter. In the fine-grained version, although both PVM and TreadMarks get lower speedup, with 8 processes, TreadMarks only achieves 50% of the performance of PVM, compared to 78% in the coarse-grained version. Most of the performance dropoff comes from the different ways that work is distributed in each version.

In the PVM version, the only data sent are sublists that can be bubble sorted, and each sublist is sent exactly twice. Consequently, the amount of data transferred is always twice the size of the total array, and the number of messages is twice the size of the array divided by the bubble sort threshold. For example, in the coarse-grained version, 2,103 Kbytes of data are sent in 1K messages, compared to the fine-grained version, where 2,110 Kbytes of data are sent in 2K messages.

TreadMarks sends more data than PVM, because the intermediate sub-arrays and the task queue are also shifted among processes. As with TSP, due to diff accumulation, the amount of data sent increases with the number of processes. The additional messages come from lock acquires and diff requests. Because the processes continually re-acquire the task queue as they divides the array until it is smaller than the threshold, the lock acquires are more frequent than the message passing in PVM. Since the size of the sublists is not an integral of a page, extra messages are sent due to false sharing.

nprocs	2	4	6	8
Tmk	1.84	3.37	4.32	5.33
PVM	1.92	3.66	5.32	6.79
Tmk/PVM	0.96	0.92	0.81	0.78
Sequential time: 81.33 *sec*				

Table 4.13 QSORT - Coarse-grained speedup

nprocs		2	4	6	8
Tmk	Messages(K)	2.9	5.6	8.9	10.0
	Total Bytes(KB)	2,709	7,251	12,121	13,969
	User Bytes(KB)	2,671	7,136	11,890	13,632
PVM	Messages(K)	1	1	1	1
	Bytes(KB)	2,103	2,103	2,103	2,103

Table 4.14 QSORT Coarse-Grained Message and Data

nprocs	2	4	6	8
Tmk	1.60	2.43	2.58	2.81
PVM	1.73	3.2	4.49	5.58
Tmk/PVM	0.92	0.76	0.57	0.50
Sequential time: 41.31 *sec*				

Table 4.15 QSORT - Fine-grained speedup

nprocs		2	4	6	8
Tmk	Messages(K)	4.9	12.4	18.7	18.9
	Total Bytes(KB)	3,216	10,594	19,163	20,950
	User Bytes(KB)	3,147	10,321	18,636	20,245
PVM	Messages(K)	2.1	2.1	2.1	2.1
	Bytes(KB)	2,110	2,110	2,110	2,110

Table 4.16 QSORT Fine-Grained Message and Data

4.6 Water

We used two data set sizes, 288 molecules and 1728 molecules, and ran for 5 time steps. The 288 molecule simulation is used in the SPLASH. The results are shown in tables 4.17 to 4.20.

With 288 molecules, TreadMarks achieves 76% of the speedup of PVM with 8 processes. In addition to synchronization messages and repeated page faults, false sharing causes extra messages in TreadMarks. In PVM, two user level messages are sent for each pair of processes that interact with each other. In TreadMarks, if a process computes the forces between its molecules and the molecules belonging to process i, it sends diff requests when reading the displacements, and when updating the forces of molecules belonging to process i. When process i reads the new values of its molecules after the barrier synchronizing these updates, it may fault again if process i is not the last one to update these molecules before the barrier. At this point, false sharing causes the faulting process to send diff requests to two different processes, because molecules belonging to different processes are protected by different locks. With 8 processes, since each process owns 1.48 pages, false sharing occurs on 7 of the 11.8 pages of the VAR array. Consequently, TreadMarks sends 5977 messages compared to 620 messages in PVM.

False-sharing also causes the TreadMarks version to send more data than the PVM version. Another cause of the additional data sent in TreadMarks is diff accumulation. Because updates to the shared array are protected by locks, diff accumulation causes TreadMarks to send $(n/2 + 1)/2$ times more data than the PVM, where n is the number of processes. Adding the two factors, with 8 processes, TreadMarks sends 3.4 times more data than PVM.

With 1728 molecules, TreadMarks achieves 99% of the speedup in PVM with 8 processes. This is the results of increased computation/communication ratio and reduced false-sharing in TreadMarks. Because the reduced false-sharing, with 8 processes, TreadMarks sends 2.8 times more data than PVM, compared to 3.4 times more with 288 molecules.

4.7 Barnes-Hut

We ran Barnes-Hut with 4096 bodies for 5 steps, and $fcell = 0.8$, $\theta = 0.6$. The results are shown in Table 4.21 and Table 4.22. TreadMarks achieves 90% of the speedup in

nprocs	2	4	6	8
Tmk	1.8	3.4	4.6	5.3
PVM	1.9	3.6	5.4	6.9
Tmk/PVM	0.98	0.93	0.85	0.76
Sequential time: 42.9 *sec*				

Table 4.17 Water speedup, 288 molecules, 5 time steps

nprocs		2	4	6	8
Tmk	Messages	727	2295	3938	5977
	Total Bytes(KB)	668	1935	3397	5195
	User Bytes(KB)	659	1894	3311	5039
PVM	Messages	50	180	370	620
	Bytes(KB)	379	759	1139	1520

Table 4.18 Water Message and Data, 288 molecules, 5 time steps

nprocs	2	4	6	8
Tmk	1.94	3.80	5.64	7.47
PVM	1.98	3.82	5.73	7.55
Tmk/PVM	0.99	0.99	0.99	0.99
Sequential time: 1568 *sec*				

Table 4.19 Water speedup, 1728 molecules, 5 time steps

nprocs		2	4	6	8
Tmk	Messages	2805	6074	9779	14399
	Total Bytes(MB)	4.01	9.66	16.98	26.05
	User Bytes(MB)	3.98	9.58	16.83	25.79
PVM	Messages	50	180	370	620
	Bytes(MB)	2.28	4.56	6.84	9.12

Table 4.20 Water Message and Data, 1728 molecules, 5 time steps

PVM with 2 processes, and 58% of that in PVM with 8 processes. With 8 processes, TreadMarks sends 70% more data and about 200 times more messages than PVM.

The difference in the amount of data is due to the different programming styles in TreadMarks and PVM. In the PVM version, each process broadcasts its bodies at the end of the iteration. With 8 processes, this takes 56 user level messages, and the total amount of data sent is 7 times the size of the array of bodies. However, in TreadMarks, the master process first collects bodies from all the others and builds the tree. Then the whole tree is sent to each of the processes, where the size of the tree is about half of the size of the array of bodies. Moreover, during the computation step, each process reads most of the bodies. Overall, the data sent in the TreadMarks version is about 50% more than that in the PVM version.

As for the number of messages, TreadMarks sends a lot more messages than PVM, mostly because the large data size and false-sharing. Since the size of the array of bodies is 147K bytes, it takes a lot of page faults to get the bodies and the Barnes-Hut tree. Furthermore, although the set of bodies owned by a process are adjacent in physical space, they are not adjacent in memory, which causes false-sharing. Because of the false-sharing, in MakeTree, each page fault causes the process to send out diff requests to several processes. All these add up to about 1400 messages in each iteration.

4.8 3-D FFT

The results are obtained by running on a $32 \times 64 \times 64$ array of double precision complex numbers for 6 iterations, excluding the time for distributing the initial value at the beginning of program. This matrix size if one fourth of that used in the NAS benchmark. The statistics are shown in Table 4.23 and Table 4.24. TreadMarks

nprocs	2	4	6	8
Tmk	1.63	2.44	2.73	2.78
PVM	1.82	3.21	4.06	4.83
Tmk/PVM	0.9	0.76	0.67	0.58
Sequential time: 69.1 *sec*				

Table 4.21 Barnes-Hut speedup

nprocs		2	4	6	8
Tmk	Messages(K)	4.7	17.7	32.6	52.5
	Total Bytes(KB)	3,493	9,766	14,576	19,555
	User Bytes(KB)	3,449	9,593	14,249	19,020
PVM	Messages(K)	0.01	0.06	0.15	0.28
	Bytes(KB)	1,805	5,426	9,059	12,704

Table 4.22 Barnes-Hut Message and Data

achieves 98% and 76% of the speedup in PVM with 2 processes and 8 processes respectively.

Because of release consistency, TreadMarks almost sends the same amount of data as PVM, with the exception of 6 processes. However, because of diff requests, many more messages are sent in TreadMarks than in PVM. These diff requests play a more important role with increasing number of processes, both because the computation/communication ratio is lower, and because more data are transferred in transposition with the increase of the number of processes.

An abnormally happens with 6 processes. We attribute this to false-sharing. During the transposition in inverse 3-D FFT, each page modified by one process is read by two other processes. Although the two processes read disjoint parts in the page, the same diff is sent to both of them. Since this happens only when running with 6 processes, in TreadMarks, 36% more messages and 45% more data are sent with 6 processes than with 8 processes. Consequently, TreadMarks only achieves 71% of the speedup in PVM, compared to 76% with 8 processes.

4.9 ILINK

nprocs	2	4	6	8
Tmk	1.48	2.40	2.54	3.88
PVM	1.51	2.63	3.58	5.08
Tmk/PVM	0.98	0.91	0.71	0.76
Sequential time: 39.6 *sec*				

Table 4.23 3-D FFT speedup

nprocs		2	4	6	8
Tmk	Messages	3639	6363	11569	8505
	Total Bytes(MB)	7.39	12.68	22.92	15.76
	User Bytes(MB)	7.35	12.60	22.78	15.63
PVM	Messages	230	1362	2130	3178
	Bytes(MB)	7.34	11.01	12.23	12.85

Table 4.24 3-D FFT Message and Data

nprocs	2	4	6	8
Tmk	1.69	2.72	3.20	3.33
PVM	1.73	2.85	3.67	3.84
Tmk/PVM	0.98	0.96	0.87	0.87
Sequential time: 910 *sec*				

Table 4.25 ILINK speedup

nprocs		2	4	6	8
Tmk	Messages(K)	40	161	331	545
	Total Bytes(MB)	20.9	65.4	111.6	158.0
	User Bytes(KB)	20.5	63.7	107.1	152.1
PVM	Messages(K)	2.1	6.3	10.5	19.6
	Bytes(MB)	18.8	47.9	75.1	101.9

Table 4.26 ILINK Message and Data

In this test, we use the data set RP01-3 [BHC+91], with an allele product $2 \times 6 \times 6$. The results are shown in Table 4.25 and Table 4.26. With 8 processes, TreadMarks sends 55% more data and 27 times more messages than PVM, and achieves 87% of the speedup of PVM.

Sparsity of the genarray is the main reason that makes TreadMarks send more messages than PVM. In our test, the size of the genarray is about 16 pages. In PVM, the processes send the nonzero values in one message, which is much smaller than 16 pages. Because the nonzero elements are scattered on 16 pages, TreadMarks sends 32 messages in respond to the 16 page faults.

One source of the additional data sent in TreadMarks is false sharing. False sharing happens because the nonzero values in the parent's genarrays are assigned to processes in round robin fashion. When the parent's genarrays are distributed to the slaves, in PVM, each slave receives only its part of the genarray, but in TreadMarks, a slave gets all the nonzero elements in the page, including those belonging to other processes. False sharing has a large effect when there is only one child in the family, where among the three genarrays of the family members to be distributed, both of the parent's genarrays are assigned in round robin fashion. With more children in the family, more genarrays are distributed, and only one parent's genarray is partitioned. We assume that for 50% of the families, there is only one child in the family.

Another source of additional data in TreadMarks is diff accumulation. Since the genarrays are initialized at the beginning of computation for each nuclear family. Although it is correct for the processes to get newly initialized data only, in TreadMarks, they also get the diffs created during previous computations.

4.10 Summary

Our results show that because of the use of release consistency and the multiple-writer protocol, TreadMarks performs comparable with PVM on a large variety of problems. For example, in SOR-zero, the Water simulation of 1728 molecules, and EP, TreadMarks performs within 2% of PVM. In most of the other tests, TreadMarks performs within 76% to 91% of PVM with 8 processes. In three of the thirteen tests, PVM performs about twice as well as TreadMarks with 8 processes. These are Barnes-Hut, the fine-grained QuickSort, and the Integer Sort with a bucket size of 2^{15}. In Integer Sort, PVM achieves a speedup of 1.12 with 8 processes. We do not think this is a great advantage compared to the speedup of 0.53 achieved by TreadMarks.

The separation of synchronization and data transfer, and the additional diff requests in TreadMarks are two of the causes of its lower performance for all the programs. In PVM, data communication and synchronization are integrated together. The send and receive operations not only exchange data, but also regulate the progress of the processes. In TreadMarks, synchronization is through locks/barriers, which do not communicate data. Data movement is triggered by expensive page faults, and diff requests are sent out in order to get the modifications.

Although the multiple-writer protocol addresses the problem of simultaneous writes to the same page, false sharing still affects the performance of TreadMarks. This occurs in QuickSort, 3-D FFT, the Water run of 288 molecules, and Barnes-Hut. Although multiple processes write to disjoint parts of the same page without interfering with each other, when a process reads the data written by one of the writers, diff requests are sent to all the writers, which causes a lot of redundant communication.

In addition, PVM also benefits from the ability to aggregate scattered data in a single message, an access pattern that would result in several miss messages in the invalidate-based TreadMarks protocol. This occurs in Barnes-Hut and ILINK.

In the current implementation of TreadMarks, diff accumulation causes additional communication. Diff accumulation is the problem of transmitting multiple overlapping diffs as a result of several processes modifying the same data. This occurs in migratory data, and in data that are reinitialized, such as Integer Sort, QuickSort, TSP, Water, and ILINK.

Chapter 5

Conclusions

This thesis presents two contributions. First, our results show that, on a large variety of programs, the performance of a well optimized DSM system is comparable to a message passing system. Especially for practical problems, such as ILINK and the Water simulation of 1728 molecules, TreadMarks performs within 15% of PVM.

Second, we summarize four main causes for the lower performance of TreadMarks compared to PVM. The first is the separation of synchronization and data transfer in TreadMarks, the second is additional access misses in the invalidate based TreadMarks protocol, the third is false sharing in TreadMarks, and finally, PVM benefits from the ability to aggregate scattered data in a single message, an access pattern that would result in several miss messages in the invalidate-based TreadMarks protocol. To alleviate these problems, we suggest a combination of performance analysis tools and compiler annotation. To avoid expensive page faults, we can use the compiler to determine the data that will be accessed in an interval. The runtime system can then prefetch this data with the acquires. We can also use the compiler to detect data migration and re-initialization, so that we can use more efficient runtime protocols in these cases. For numerical computations with static shared memory access patterns, such as 3-D FFT, we can use the compiler to detect sharing patterns, and avoid false sharing. Finally, if the application is programmed with poor locality, compiler optimization may not be possible. In this case, a performance analysis tool could assist the user in improving the parallel program.

In terms of programmability, our experience indicates that it is easier to program using TreadMarks than using PVM. Although there is little difference in programmability for simple programs, for programs with complicated communication patterns, such as ILINK and 3-D FFT, a lot of effort is required to determine what data to send and whom to send the data to. Distributed shared memory, on the other hand, provides an easier path to developing parallel programs on networks of workstations.

Bibliography

[AH93] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.

[BCZ90] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176, March 1990.

[BHC+91] S.H. Blanton, J.R. Heckenlively, A.W. Cottingham, J. Friedman, L.A. Sadler, M. Wagner, L.H. Friedman, and S.P. Daiger. Linkage mapping of autosomal dominant retinities pigmentosa (RP1) to the pericentric region of human chromosome 8. *Genomics*, 11:857–869, 1991.

[BZ91] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[CIS93] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[DKCZ93] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

[DSC+94] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GS92] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. In *Concurrency: Practice and Experience*, pages 293–311, June 1992.

[Har90] R.J. Harrison. Portable tools and applications for parallel computers. In *International Journal of Quantum Chemistry*, volume 40, pages 847–863, February 1990.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KDCZ94] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[LLJO84] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proc. Natl. Acad. Sci. USA*, 81:3443–3446, June 1984.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.0, May 1994.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[Par92] Parasoft Corporation, Pasadena, CA. Express user's guide, version 3.2.5, 1992.

[SGSC94] A. A. Schäffer, S. K. Gupta, K. Shriram, and R. W. Cottingham Jr. Avoiding recomputation in linkage analysis. *Human Heredity*, 44:225–237, 1994.

[SWG92] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, March 1992.