ANURAG SAH

Parallel Computation Structures Group
Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley

Parallel Language Support on Shared Memory Multiprocessors

May 22, 1991

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and
Computer Sciences, University of California, Berkeley,
in partial satisfaction of the requirements for the degree
of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: ______________________ Research Advisor

______________________ Date

______________________ Second Reader

______________________ Date

This work was supported in part by the National Science Foundation PYI Award (CCR-90-58432), with matching funds from Motorola Inc. and the TRW Foundation. Computing resources were provided in part by NSF CDA-8722788.

Abstract

The study of general purpose parallel computing requires efficient and inexpensive platforms for parallel program execution. This helps in ascertaining tradeoff choices between hardware complexity and software solutions for massively parallel systems design. In this report, we present an implementation of an *efficient parallel execution model* on shared memory multiprocessors based on a Threaded Abstract Machine. We discuss a k-way generalized locking strategy suitable for our model. We study the performance gains obtained by a queuing strategy which uses multiple queues with reduced access contention. We also present performance models in shared memory machines, related to lock contention and serialization in shared memory allocation. A bin-based memory management technique which reduces the serialization is presented. These issues are critical for obtaining an efficient parallel execution environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The study of general purpose parallel computing requires efficient and inexpensive platforms for executing parallel programs so that a better understanding of what it takes to execute them ensues. This may result in the right hardware choices for massively parallel systems design, as well as an appreciation of what aspects are best dealt at the software level in order to minimize the hardware real estate and complexity.

Parallel program development on most machines currently requires substantial programming effort. The algorithm needs to be partitioned, so that the work can be appropriately distributed among the available processors. Scalability results only when the tasks are partitioned such that there is little interaction among the processors. Often this interaction can be subtle such as cache line contention when various processors write to locations that fall in the same line, causing the cache line to bounce from one cache to another. This can be difficult for a programmer to address, since the line size is invisible in the instruction set architecture and may differ from one model of the machine to another. Similarly, the costs of low-level synchronization primitives, such as lock mechanisms [Thakkar 90], barrier synchronization, etc. are difficult to ascertain. These factors make programming parallel machines much more difficult than programming conventional uniprocessors. These low-level performance determining issues are best dealt with as part of the language system, so the programmer can concentrate on the logical aspects of the program development.

In this report we discuss the Threaded Abstract Machine (TAM) developed by the Parallel Computation Structures group at the University of California, Berkeley and present issues related to its implementation and realization in bus-based Shared memory multiprocessors. One of the objectives of the Threaded Abstract Machine (TAM) [Culler 91] is to provide a parallel program development environment across a diverse spectrum of platforms including uniprocessors, shared memory multiprocessors and distributed memory multiprocessors. The emphasis is on an environment in which a programmer can develop and debug parallel programs, to run efficiently on a variety of machines. Thus the programmer does not have to study the subtleties of different architectures before he can begin programming. Another advantage is the accrued gain of portability of the same high level programs across machines with different architectures.

TAM is a model of execution for parallel programs. It is supported by an abstract machine language called Thread Language Zero (TL0). It provides an explicit scheduling hierarchy and storage model. Interprocessor communication latency and remote memory access latency are addressed by split-phase operations. An operation which requires an uncertain duration due to any of the latencies mentioned above, is carried out in two phases. The first phase initiates the operation by sending a message. Return of the value(s) requested starts the second phase of the operation. Thus, the TAM model tries to keep the execution pipeline full, provided there is enough work to do.

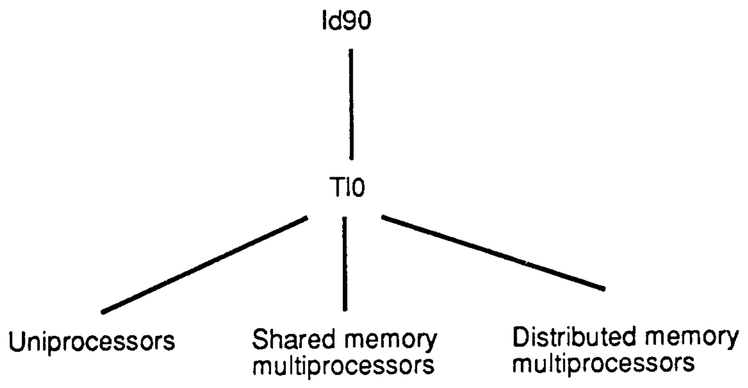

Id90
Tl0
Uniprocessors
Shared memory multiprocessors
Distributed memory multiprocessors

Figure 1.1: Translation path from Id90 to target architectures

TL0 serves as an intermediate representation of a program within the TAM execution model. Programs written in a high level language are translated into TL0. The TL0 representation is then translated into native code or C code for various machines. This provides execution platforms on several current machines. We start from Id90 [Nikhil 90] which is a parallel programming language with non-strict semantics. Figure 1.1 shows the translation path. Implementation on a particular platform involves generation of run-time code for the machine starting from TL0. Run-time support for scheduling, synchronization and heap access needs to be provided. We present implementation strategies suited towards shared memory multiprocessors. We also discuss the issues faced in realizing an efficient implementation on such machines. These relate to efficient shared memory management, generalized locking strategies, management of scheduling queues and frameworks for synchronization of mutually exclusive access of shared structures. A study of performance issues in shared memory multiprocessors from the point of view of processor scalability *i.e.* getting increasing performance with processors is presented.

The next chapter discusses issues which affect scalable performance in shared memory multiprocessors. A family of machine characterization and software tuning tests are developed to identify key performance issues in a TAM implementation. Chapter 3 describes the TAM execution model and its implementation on bus based shared memory machines. The programs considered are shown in Chapter 4, while Chapter 5 presents the performance achieved under various implementation strategies.

Chapter 2

Performance models in Shared memory multiprocessors

Shared memory multiprocessors provide a convenient environment for execution of tasks in which the data workspace can be partitioned amongst the processors. In order to get the best performance, the workspace division must be equitable *i.e.* the processors get equal amounts of work. This is generally done by dynamic data partitioning whereby blocks of work are assigned to a scheduling queue from which the processors remove work. This ensures load balancing among the processors. Access from the global work queue has to be done in a mutually exclusive fashion. This is done by converting the access code into a critical section executed under the ownership of a lock by a processor and its subsequent release.

Providing support for a parallel language on shared memory multiprocessors requires an understanding of several performance related issues. The nature of the locking mechanism and the contention observed plays a vital role. Efficient shared memory management techniques need to be provided that prevent processor serialization in memory allocation. Separate cache line alignment of frequently accessed data needs to be done to prevent unnecessary cache transfers. In this chapter, we analyze these issues which are important for obtaining performance on shared memory multiprocessors. We use some simple C programs to demonstrate their effects. Ways of overcoming these sources of possible performance loss are presented. The tests have been carried out on the Sequent Symmetry, with 14 on-line processors and 32 Kbytes cache on each processor. In the next section we describe the architecture of the Sequent Symmetry. Section 2.2 discusses the performance related issues.

2.1 Sequent Symmetry Architecture

The Sequent Symmetry is a shared memory multiprocessor based on a single bus architecture. This bus is shared by various modules which are directly connected to it. Dual processor boards with private caches and floating point units for each processor are connected to this bus. Memory boards are also plugged in to this bus. The processors are tightly coupled and share a single memory address space. The Sequent Symmetry S81 that we have used has fourteen 386 processors (16 Mhz) and 24 Mbytes of memory. Each processor has 32 Kbytes of cache RAM. The bus has a clock rate of 10 Mhz.

2.2 Performance Models

In this section, we consider issues which can degrade performance in shared memory multiprocessors if not understood properly. First, is the issue of lock contention among the various processes spawned by a parent process. Second, is the serialization of processes during allocation of requested shared memory. We have also presented a study of the effects of cache transfer on preformance and processor utilization. The following subsections provide the motivation for considering these issues and the experimental results.

2.2.1 Lock contention

Lock contention is an important direct source of performance inhibition. It not only causes processes to serialize, but affects system bus usage depending on the locking algorithm used.[1] We consider snooping spin locks, the C code for which is described in Figure 2.1.

The a_exchg_b(lock, CONST) function atomically exchanges the value in the variable lock with the const value CONST. While obtaining the lock, we do an atomic exchange of the value in the lock variable with the const LOCK. If we return LOCK, then someone else has the lock and we just spin waiting for its release. Note that an atomic exchange locks the system bus during the course of the instruction execution and hence is costly from bus resource point of view. Thus if a first lock attempt fails, snooping is done and the next attempt is made only when the lock is released. While unlocking, we just clear the lock with an atomic write of UNLOCKED to lock. This study uses lock macros hand written in assembly code using the xchg instruction which atomically exchanges the contents of a register with that of a memory location.

[1][Thakkar 90] presents an extensive study of lock performance in high contention environments. They study the performance of various locking mechanisms.

The system bus can be unnecessarily loaded in high lock contention environments even with snooping locks. This is because a flurry of lock attempts by waiting processes accompany a lock release. Each attempt requires an atomic lock test which is achieved by the atomic byte exchange instruction. This instruction locks the system bus for the duration.

Spin_lock:

```
while (a_exchg_b(lock, LOCK) == LOCK) {
        while (lock == LOCK) {
        }
}
```

Spin_unlock:

```
a_exchg_b(lock, UNLOCK);
```

Figure 2.1: Snooping lock and unlock

We present measurements of effective lock rate across the spectrum from high contention locks to a low lock contention environment. Measurements are obtained by executing a program that forks processes which repeatedly grab lock(s), increment a counter, and release lock(s). The first process to reach a fixed number of locks causes all processes to exit. The lock rate is simply the number of locks obtained divided by the time to reach this point. The lock rate measured is the rate when all the processors are attempting the locks. We consider four cases. In the first case, the processes contend for a single lock; thus high contention ensues. This case is referred to as SL. In the second case, each process grabs a lock from a finite set of locks equal to the number of processes. The lock to be grabbed is determined randomly. Thus, there may be slight contention. This is referred to as FL,RL. The third case has a million locks, and each process grabs locks at random. Thus there is minimal lock contention. This is referred to as ML,RL. The last case has zero lock contention where each process grabs its own private lock and is referred as FL,OL. These four programs are given in Appendix A.1.

Figure 2.2 presents the variation of the number of locks obtained per second as a function of the number of processors. Simple, singleton lock performs poorly. The overall lock rate decreases with additional processors. The increase observed from going from 1 processor to 2 (notice that the locks obtained/sec. almost double) is due to the fact that contention has not become a major issue, as the two processes after the first contention can run out of phase with respect to lock attempts. What this means is that the subsequent attempts are likely not to be made together. Beyond two processors, the contention becomes significant and overcomes the gains of additional processors. The lock rate decreases down to a third (0.15 to 0.05 million/sec) for 12 processors. Only ML,RL shows a linear increase in the locks obtained per second (lock rate). This case depicts the independent operation of processes with virtually no influence on the performance of each other. Thus, the number of locks obtained is a linear multiple of the number of processes. FL,OL should seemingly work as well as ML,RL, if not better, because the lock corresponding to each processor should remain cached after the first attempt. However we see clearly this is not the case. The reason is the way we have declared the locks. Each lock is an element of an array. Four locks occupy a cache line size which is 16 bytes, and is the unit of ownership and transfer among the various caches of the processors. Each cache line bounces among at most four processors when attempts are made to access a processor's lock. This slows down each of the processes. If the locks in this case are declared in each process' private stack then FL,OL performs better than ML,RL because there is no cache to cache transfers of cache lines containing locks. FL,RL also shows an increase but due to some degree of contention the increase is less compared to FL,OL.

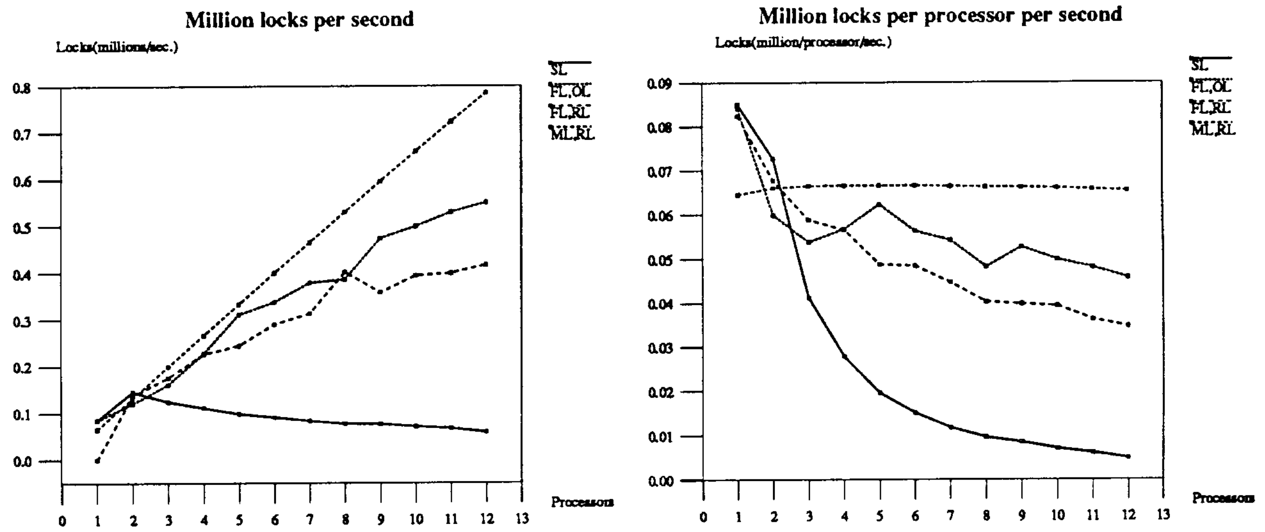


Figure 2.2: Locks obtained per second and per processor per second

Figure 2.2 also shows the average number of locks obtained per processor per second. This figure shows more clearly the lock performance degradation due to changes in the degree of contention. The lock rate per processor drops rapidly for SL by almost a factor of 17. ML,RL remains flat as the number of processors are increased. FL,OL shows an interesting phenomenon. There is a peak at 5 processors and again at 9 processors. These peaks arise due to the fact that the lock for the fifth process occupies a fresh cache line. This line remains cached and the process speeds through grabbing a million locks before others and increasing the overall lock rate. The same is true for the ninth process.

2.2.2 Shared memory allocator serialization

Access to system shared data structures has to be done under ownership of a lock which provides mutual exclusion. An example of such an access is the shared memory allocator which allocates a requisite number of bytes in shared memory. Shared memory is used for sharing data between processes. Dynamic allocation of shared memory by processors requires use of the allocator.

Appendix A.2 presents an example program, maxpar, which requires dynamic shared memory allocation. The program has a large degree of parallelism. Each of the processes gets a certain number of bytes of shared memory and touches successive bytes of it for *delay* iterations. Thus after the allocation and caching of the bytes, each process operates out of its cache. This is repeated a large number of times. The max_par program contains parallelism which is ideal from the point of view of scalable shared memory multiprocessors. Each process gets the bytes requested and then runs independent of each other. Thus the only interaction is at the time of shared memory request.

Appendix A.2.1 contains the code which uses the shared memory allocator call provided as part of the parallel programming library, for dynamic shared memory allocation. Figures 2.3 and 2.4 show the speedup and utilization per processor obtained for *delay* of 10 and 100 for a total of 100,000 memory allocation requests of size 16 bytes distributed equally among the various processes (almost equally). The shared memory allocator is referred to as Mem_alloc in the graphs. A *delay* of 10 shows poor speedup and processor utilization. The maximum speedup obtained is merely 2.5. Increasing the *delay* to 100, improves the performance. The reason for the performance observed is that if multiple processes request the shared memory allocator simultaneously, there is a high degree of lock contention for a single lock. This lock is needed to provide exclusive access to the system shared data structures. This results in serialization of processes, which causes serious performance degradation especially if there is a high frequency of requests for shared memory allocation. Thus the shared memory allocator becomes a bottleneck for the processes and causes a sharp decline in processor utilization as the number of active processes is increased.

The serialization seen above can be reduced by decreasing the effective rate at which the shared memory allocator gets called. We have developed a bin-based shared memory manager which achieves this. Appendix A.2.2 contains the code with our version of the shared memory manager. This manager allocates a large chunk of memory in fixed size blocks, when the first demand occurs, and maintains them as a linear list in a bin. Each process has its own private bin and hence access from this bin can be done without locking. The manager provides memory blocks requested by the process and reuses freed blocks. Further allocation of shared memory through the shared memory allocator is done only when the bin of a process gets empty and a further request is made.

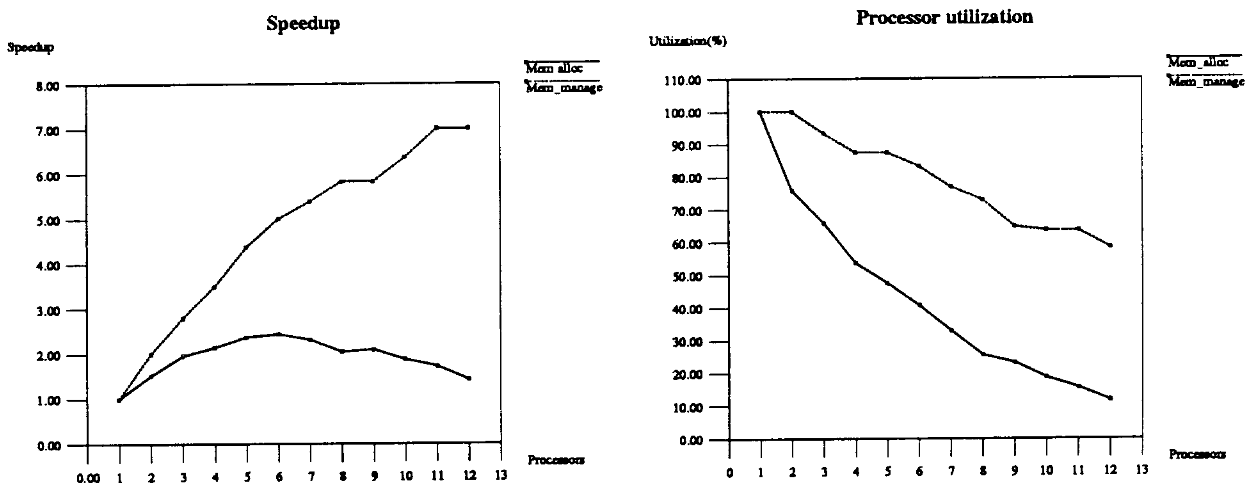


Figure 2.3: Speedup and processor utilization in max_par for 100000, 16, 10

Figures 2.3 and 2.4 show the speedup and utilization for the memory manager. It is referred to as Mem_manage in the graphs. A *delay* of 100 shows almost linear increase in the speedup with the shared memory manager whereas with the parallel programming library shared memory allocator the speedup gradually tapers down. Thus the memory manager reduces the process serialization in the shared memory allocation process and yields a better performance.

2.2.3 Cache to Cache transfer

To study the effect of cache transfers, we consider the program presented in Appendix A.3 which generates processes that repeatedly write to the same main memory line. This causes the corresponding cache line to transfer from one cache to another. Table 2.1 shows the time taken when each process undergoes 1 million iterations of the write loop. The processor utilization is also shown. Processor utilization has been computed by first obtaining the execution time for a total of 1 million iterations.

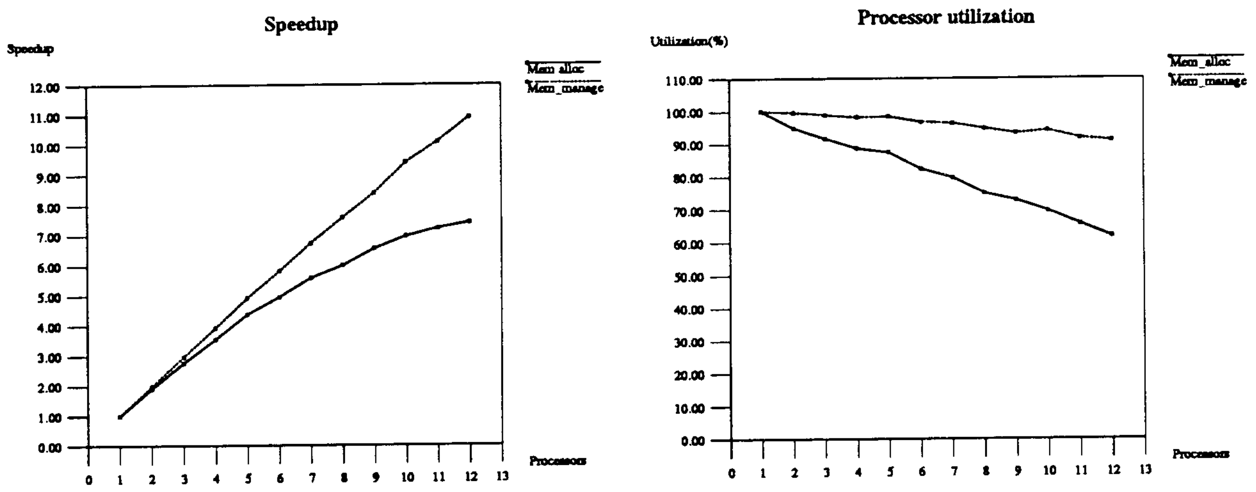


Figure 2.4: Speedup and processor utilization in max_par for 100000, 16, 100

Processors	1	2	4	6	8	10	12
Time(sec)	1.6	4.7	7.4	11.0	14.8	18.3	22.0
Utilization	1.00	0.34	0.21	0.15	0.11	0.09	0.07

Table 2.1: Execution time for cache transfer program

This is done by dividing the execution time by the number of processors (Note that N processors execute N million iterations overall). Then the execution time on 1 processor is divided by the execution time on N processors and by the number of processors, N.

Table 2.1 shows the serialization of processes due to cache transfers and the rapid decay in processor utilization. The effects are so significant that on 2 processors the utilization drops to 0.34. Thus when there is contention for any main memory line by processors, cache transfers causes reduction in performance and processor utilization. Shared data has to be placed such that frequently accessed ones do not occur in the same cache line. This can be achieved by padding the rest of the cache line in which the data is placed by a dummy shared variable.

2.3 Conclusions

This study shows three issues that must be considered while implementing a parallel programming language support environment on bus based shared memory machines in order to prevent their serving as sources of possible performance loss. Choice of an appropriate locking strategy overcomes poor lock performance. Use of a shared memory manager which provides a lower effective shared memory allocator function call rate by maintaining shared memory blocks in private bins, reduces process serialization when the shared memory request rate is high. Transfer of actively sought cache lines can cause performance loss. This is reduced by placing frequently referenced shared data in different main memory lines. These issues are important from the point of view of scalability as they ensure maximum parallelism among the processes with little interference.

Chapter 3

Implementation of TAM on Shared memory multiprocessors

The Threaded Abstract machine (TAM) is realized by an abstract machine language called Thread Language Zero (TL0). It serves as a common intermediate representation in the translation of Id programs to code supported by the TAM model which can be executed on target machines. The model of execution specifies the run-time environment and the associated data structures. TL0 is machine independent. The actual implementation of the run-time environment and the data structures, however, depends upon the target machine serving as the host for the execution model. Different machines require diverse implementations because the means of achieving efficiency depends on the architecture and its preferred programming style. The compilation process can be divided into two steps. A front-end does program graph analysis and optimizations to generate TL0 code. The back-end then generates optimal code for execution on the target machine. The front-end is thus independent of the target machine, while the back-end varies from machine to machine.

3.1 TAM model of execution

A TL0 program is represented by *codeblocks*, *threads* and *inlets*. Storage models are provided in the form of *codeblocks*, *frames* and *istructures*. A codeblock (Figure 3.1) is a code fragment which is a collection of threads and inlets. When a codeblock is invoked, a frame is allocated from the heap. The threads and inlets in the codeblock execute in the context of this frame. The idea of a codeblock is that of a static entity whereas a frame is a dynamic run-time entity much like the call frame in conventional strict programming languages.

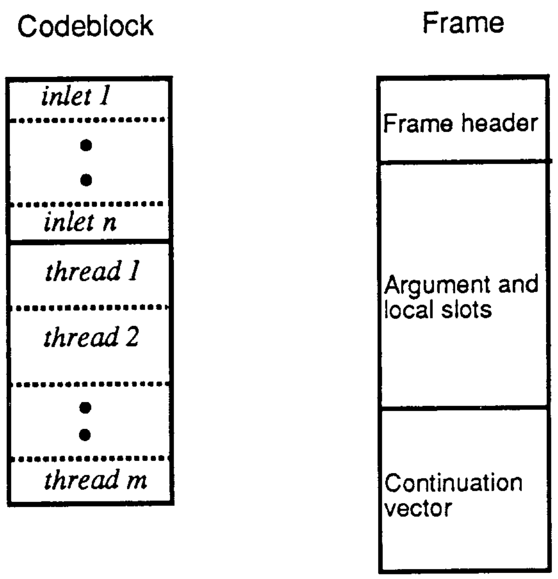


Figure 3.1: Codeblock and Frame structure

A thread is a sequence of instructions which when started run to completion. A thread cannot be interrupted in the middle, except to execute higher priority inlets. Threads are a convenient way to hide latency of remote memory access. A processor can maintain a collection of threads to execute while a request is pending. This is a result of the fact that the compiler guarantees that threads may be executed out of order. Threads which are ready for execution are stored in a *continuation vector*. There are two continuation vector areas associated with a frame: the local continuation vector, which resides on the processor executing the threads of the frame, and the remote continuation vector, which is a part of the frame. New threads are generated by means of a *fork* operation which pushes a thread number onto the local continuation vector. They are also generated by the RCVPUSH operation which atomically pushes threads to the remote continuation vector. Thread dependencies are taken care of by means of synchronizing forks. Synchronizing forks decrement a counter and allow the fork to occur only when the counter reaches zero. A thread which has to be executed after a specific collection of threads have completed, is synchronizingly forked by those threads. It does not become ready for execution until all those threads finish execution.

A codeblock may invoke several other codeblocks even before they complete. This is due to non-suspension of the caller. The invocation profile represents a tree of frames rather than a stack. Each

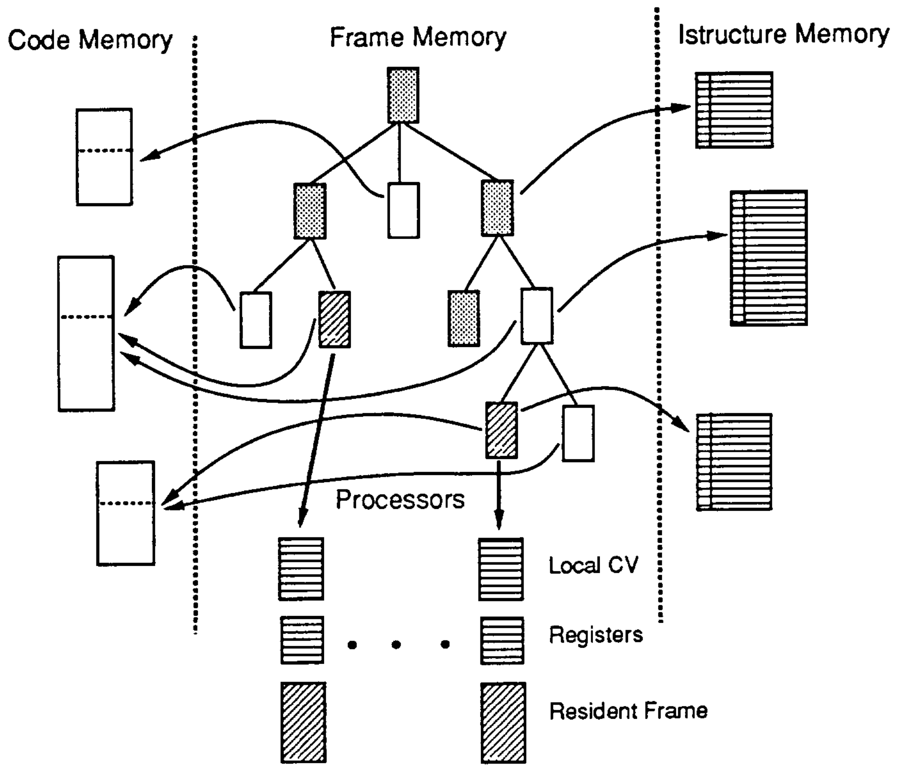


Figure 3.2: Representation of a TAM program execution

invocation of a codeblock is referred to as an *activation*. An activation is *ready* when there are threads in its continuation vector. Activations are removed from the ready pool and made *resident* on processors. A single codeblock can have multiple activations active at the same time. Each activation represents a context within which threads of the codeblock execute.

Figure 3.1 shows the structure of frames. Frames consist of a frame header, slots for arguments and locals, and a continuation vector area. The frame header consists of slots that are common to all frames. The local slots are used as data slots and the argument slots pass arguments to codeblock invocations. The continuation vector in the frame is the area where inlets of the activation push threads. This is different from the continuation vector out of which the activation removes threads for execution. The former is referred to as the *remote* continuation vector while the latter is the *local* continuation vector. The local continuation vector resides on each processor.

Figure 3.2[1] shows the run-time structures associated with a TAM program. Codeblocks reside in code memory. Associated with each codeblock are frames which reside in frame memory. Istructures are discussed later.

Inlets serve to provide the external interface to activations. They are specialized message handlers which interpret a received message and take appropriate actions like frame slot store and thread forks. They permit activations to receive values, arguments and results. They are also used when a split-phase operation completes and returns the value to the activation that requested it, by executing an inlet of the activation. This inlet stores the value in a frame slot and pushes a thread into the remote continuation vector area of the activation. Inlets are part of codeblocks.

Istructures also form part of the TAM storage model. They are accessed by means of split-phase operations like IFETCH and ISTORE. An istructure is allocated when an activation makes an IALLOC request. It consists of data slots and tags associated with each slot. The tags can be in one of the following states : EMPTY, FULL and DEFERED. On allocation, an istructure slot has a state of EMPTY. An ISTORE operation on an EMPTY slot causes it to move to a FULL state. An ISTORE operation on the istructure slot before an IFETCH is performed causes the tag to move to a DEFERED state. The frame pointer and the inlet number of the requesting activation is stored in a deferred read list associated with the istructure slot. ISTORE on a DEFERED tag state causes the inlets in the deferred read list to be executed, stores the value in the slot and changes the state to FULL. ISTORE on a FULL state causes an error.

The TAM execution model provides for explicit scheduling of frames and threads. There is a two

[1]Taken from [Culler 91]

level scheduling hierarchy. At the upper level in the hierarchy is the frame scheduling queue. Ready frames are assigned to this queue and are picked up by processors for execution. This process of accessing a ready frame from the scheduling queue is referred to as making the frame resident on the processor. The thread scheduling in the local continuation vector forms the lower level of the scheduling hierarchy.

When an activation is removed from the scheduling queue and made resident on a processor, it runs until there are no threads in its continuation vector to execute. At this point a new activation is removed from the queue. This run of an activation is referred to as a *quantum*. Larger the average quantum size of the program, lesser is the fraction of time spent in removing the ready activations from the scheduling queue. This reduces the scheduling overhead. Thus, before the activation is dropped due to lack of threads in its local continuation vector area, its remote continuation vector area is checked to see if threads have been pushed by inlets in the meantime. If there are, then they are copied to the local continuation vector area and the quantum continues.

3.2 TAM model on shared memory multiprocessors

As described before, the implementation of the TAM run-time environment is machine specific. This section discusses the issues related to an implementation on bus-based shared memory multiprocessors. In order to maintain data consistency, operations on shared data structures that may be manipulated simultaneously by more than one processor, need to guarantee mutually exclusive access to them. The concept of a generalized lock strategy with multiple states that guarantees this, has been discussed. For our purposes, it has been found to perform much better than two-state locks which were used for previous versions. Implementation of frame and istructure tag transitions have been discussed. We also present the concept of parallelizing the frame scheduling queue and indicate the advantages obtained. A bin-based memory management technique compared to shared memory allocator (shmalloc) function call to the parallel programming library on the Sequent, has been discussed.

3.2.1 A Generalized Locking Strategy

Traditionally, only two-state locks have been used to provide exclusive access to shared data structures. In the TAM model, some shared structures have a notion of state associated with them. During execution, they make transitions from one state to another. We generalize the two-state locking model to a k-state model in which the data structure can have k-1 states. The extra state is used to serve as a synchronization point and serves the same role as the locked state in a two-state lock. An operation which needs exclusive access to this structure, attempts an atomic exchange of the transient state with the current state. The exchange succeeds only when the structure is not already in the transient state when the attempt was made. Thereafter the required operation may be carried out and the state restored to a non-transient state on completion. Failure leads to a snooping spin and a reattempt when the state is restored to a non-transient state. Figure 3.3 shows these ideas. Operation Op1 attempts a transition from state A to transient. On success, the operation is carried out and a transition to state B occurs. The darker arrow represents that Op1 must take the state to a non-transient state by the lighter arrow after the operation is complete. Operation Op2 causes the sequence of transitions from state C to transient to state B. Each operation is thus responsible for two transitions, one from a non-transient state to a transient state, and then back to a non-transient state. The transient state forms a synchronization point where all operations requiring exclusive access to the structure need to bring the state before the guarantee can be provided. We will discuss specific instances of the use of generalized locks in the following subsections.

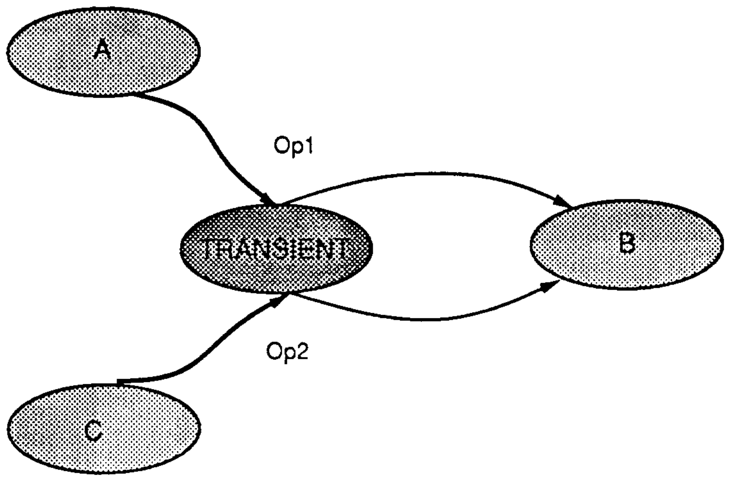


Figure 3.3: Generalized locks

3.2.2 Frames, States and Transitions

Frames may be accessed by any processor even though they may not be resident on the accessing processor at the current point of time. An example of this is executing an inlet on behalf of another processor on which the frame may be currently resident. Frames are thus allocated in shared memory heap. The remote continuation vector of a frame may be accessed by any processor to push ready threads. Even the processor on which the frame is resident, accesses it to copy the threads to its local continuation vector. Hence thread pushes and pops from the remote continuation have to be done in an exclusive access manner. We use generalized locks to implement the remote continuation access. The rest of the frame may be accessed without the lock overhead even though it resides in shared memory, because it is guaranteed by the compiler that simultaneous write access will not occur.

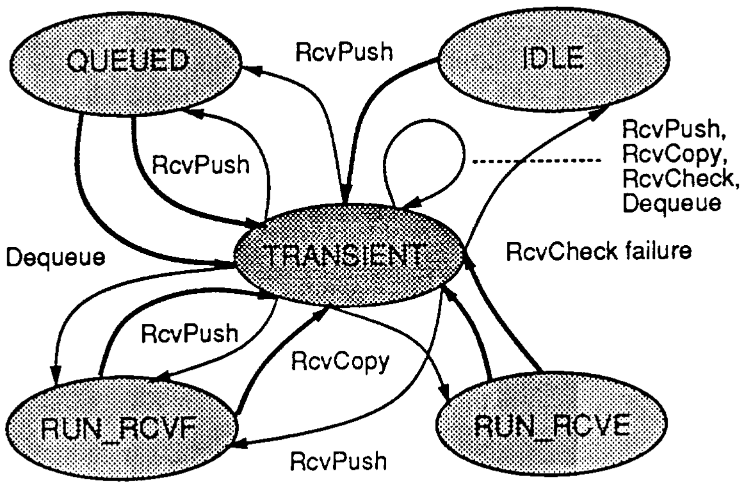


Figure 3.4: Frame states and transitions

Frames after allocation can remain in one of five states, of which one is a transient state. The others are referred to as static states. During the course of execution frames make atomic transitions from one state to another. They can however remain in the transient state only for a short duration The processor making the frame transition from a static state to the transient state guarantees the return to a static state after requisite frame operations have been done. The static states are : IDLE, QUEUED, RUN_RCVF and RUN_RCVE. The transient state is TRANSIENT. Refer to Figure 3.4. A frame in an IDLE state is inactive. There are no threads active because it is waiting for threads to be posted into its remote continuation vector area by other processors executing inlets for this frame. Rcvpush operations push ready threads into the remote continuation. Once threads are posted, the frame becomes active and moves to the QUEUED state. This state reflects that the frame is waiting in a queue for a processor to pick it up and start executing its threads. The act of dequeueing by a processor moves the frame to the RUN_RCVF state. When the local continuation vector area has been updated with the threads posted into the remote continuation vector area, the frame makes a transition to the RUN_RCVE state. The rcvcopy operation is responsible for this transition. Rcvpush in RUN_RCVE or RUN_RCVF state takes the frame to RUN_RCVF. Rcvcheck operation is carried out when the local continuation of a resident frame is empty. If there are threads in the remote continuation, they are copied to the local continuation and the quantum continues. Otherwise the frame is made IDLE and dropped. All transitions first bring the frame into the TRANSIENT state.

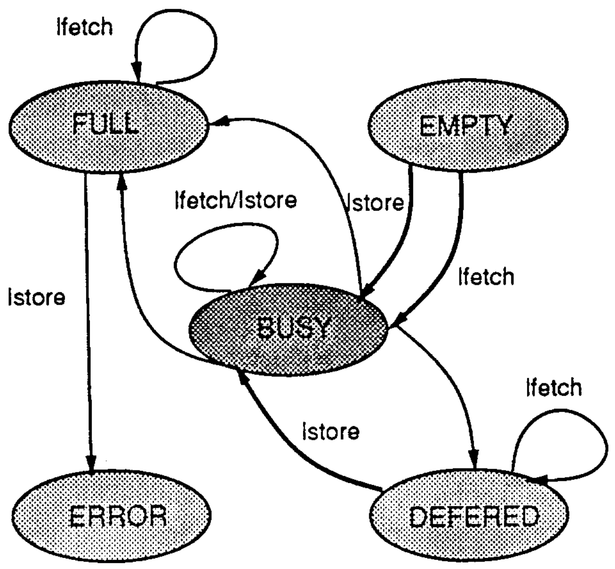


Figure 3.5: Istructure tag transitions

3.2.3 Istructures and Tag state transitions

Istructure slot tags as discussed before can be in one of the following states: EMPTY, FULL or DEFERED. Split-phase istructure operations may cause the tags of the slot to make a transition from one state to another. These state transitions on a shared memory multiprocessor have to be atomic operations because simultaneous access and modifications of the tag state can occur. We implement this transition diagram with generalized locks by introducing a transient state called BUSY. A state transition occurs by an atomic exchange of the BUSY state into the tag state. Requisite operations are carried out and then a transition to the final state occurs. Thus the BUSY state serves as a synchronization point for the istructure operations that have to be performed atomically. It guarantees exclusive operation privileges and blocks others from similar access. Refer to Figure 3.5 for the state transition diagram.

3.2.4 Parallelizing the scheduling queue

The TAM model of execution provides for explicit queueing of activations which are ready to execute. When an RCVPUSH operation is carried out, the state of the frame into whose remote continuation vector area the threads are being pushed, is checked. If the frame is IDLE it is placed in the scheduling pool. Processors which finish executing a quantum for a frame, access new ready frames from this scheduling pool. The access rate to this pool then becomes an important issue in performance with scalability of processors in a shared memory multiprocessor. If the access rate is low and there are plenty of frames in the pool then a single queue serves fine and provides no bottleneck. However if the access rate is high then this queue may end up serializing all the processors, causing them to spend most of their time trying to access the scheduling queue. This provides the motivation for parallelizing the queue structure.

A parallel queue structure has several advantages. As discussed above, the queue lets processors run in parallel on useful sections of the code rather than being serialized in the queue access. Thus a fast queue access rate *i.e.* due to small quantum sizes, does not reduce performance. It also results in a possibly larger pool of activations because each of the processors can access the queue at a faster rate, and hence execute the activations at a faster rate, thereby generating more activations over a period of time.

The queue model that we have chosen to implement is a parallel queue structure with processors placing ready activations onto almost private queues. They remove activations from their own queue. If the queue is empty then they try to steal activations from other queues. Thus, if there are a large number of activations distributed among the queues, then most of the processors run from their own queue except in the beginning when the frames have to be distributed among the queues. The path in the code that gets frames from the personal queue has been optimized because this is the frequent case when there is enough parallelism. It takes slightly longer for a processor to execute the code that hunts around for frames from other queues when its queue is empty. An advantage of working out of ones own queue and placing frames onto it is that it can provide for better cache locality due to possible spatial binding of a frame to a processor. This is however not strictly true for the entire frame because parts of the frame are accessed by other processors also, like the remote continuation vector area. However the portions of the frame that are manipulated only when the frame becomes resident are cached better. This reduces bus traffic due to a reduction in the cache to cache transfer.

3.2.5 Memory management

Allocation of an activation requires a block of shared memory to be allocated. This can be done with the help of the shared memory allocation routine provided. Sequent Symmetry provides the shmalloc function call in the parallel programming library. Since shmalloc manipulates shared data structures, it is carried out under lock so that exclusive access may be provided to only one of the requesting processors at a point of time. Problems with this have been discussed in Chapter 2. It can result in serialization of processors trying to obtain shared memory for frames. This is a severe problem when activation generation rate is large.

The way to overcome the serialization problem is to reduce the rate of call to the shared memory allocator. This can be done with the following model of shared memory management for frames. Each processor maintains bins of fixed sizes in multiples of 128. When a request for a frame of a particular size is made, the processor's private bin corresponding to the closest size in which the requested size fits is checked. If it is empty then a certain multiple of the bin size is allocated by the shared memory allocator function call. This memory block is partitioned into frames of the bin size and kept in a linear list. One of these is allocated to satisfy the frame request. However, if the bin of that size contains frames, then the process of going through the shmalloc call is eliminated and a frame is immediately allocated. Thus the processors can run without blocking each other in the lock associated with the shmalloc call. This reduces the effective shared memory allocator access rate. A further optimization in the process is that the compiler precomputes the bin number into which the frame will fit and provides this bin number. This reduces the run-time overhead in calculation of the bin number into which the frame can fit. This scheme causes some amount of internal fragmentation but the gains from the reduced lock contention in the shared memory access are immense to overcome this.

Chapter 4

Programs under consideration

This chapter discusses the various programs considered. It discusses the algorithms and shows from where the parallelism stems. The programs have been written in Id90 which is the current version of the Id parallel programming language developed at MIT. Id90 differs from previous versions of Id in that it supports static typing and mutable objects.

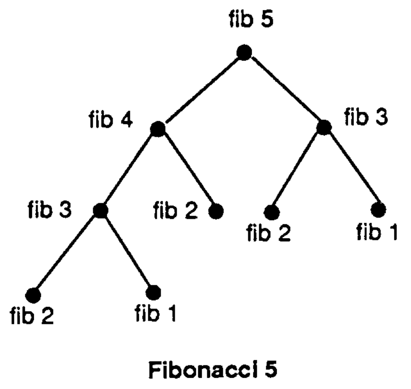
4.1 Id programs

4.1.1 Fibonacci, Fib

This is the standard fibonacci series calculation problem. The high level Id90 program is shown in Figure 4.1. The first fib activation checks whether the argument N is less than or equal to 2. If it is, it sends a result of 1 to the print activation. Otherwise it generates two new fib activations and sends them arguments of N-1 and N-2 respectively. These two activations calculate fibonacci of N-1 and N-2 respectively and send the result and completion signal to the calling activation. This is done by executing an inlet on behalf of the caller activation, which stores the result in a frame slot and forks a thread. This thread in turn forks a 2-way synchronizing thread. The first result merely causes the synchronizing thread not to be forked, while the second one causes the synchronizing constraint to be met and the thread is forked. This thread calculates the sum and returns the result to the caller. Thus, at a point of time there are several activations from several portions of the activation tree either resident on a processor or in the scheduling pool. Clearly fibonacci has a high activation generation rate. Parallelism results from the large pool of activations present to be executed.

```
def fib n = if n <= 2 then 1
            else (fib (n-1)) + (fib (n-2));
```

Figure 4.1: Fibonacci



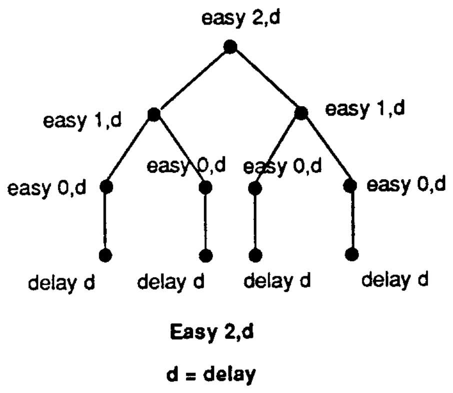


Figure 4.2: Activation trees for fib 5 and easy 2,d

4.1.2 Easy

Easy is like Fibonacci in activation generation behavior except that when the leaf activations of the split activation subtree are reached, each leaf activation generates a new delay activation. The quantum size of the delay activation can be controlled by varying the second parameter to easy. This argument determines how many iterations of a for loop are executed (Figure 4.3). Thus the average quantum size in easy can be tuned. This helps in providing an understanding of the interplay between quantum size and speedup performance.

```
def split n m = if n == 0 then delay m
                          else (split (n-1) m) + (split (n-1) m);

def delay m = {sum = 0;
              in {for i <- 1 to m do
                      next sum = sum + 1
                    finally m - sum}};
```

Figure 4.3: Easy

4.1.3 Dynamic Time Warp, Dtw

Dynamic Time Warp (Appendix B.1) is a larger program that implements an algorithm used in discrete word speech recognition for time alignment of uttered word templates referred to as test templates and a template from a speech dictionary referred to as the reference template. A test template and a reference template matrix are first generated. The rows of these matrices correspond to speech frames and columns represent the speech coefficients of degree K where K is the number of columns.

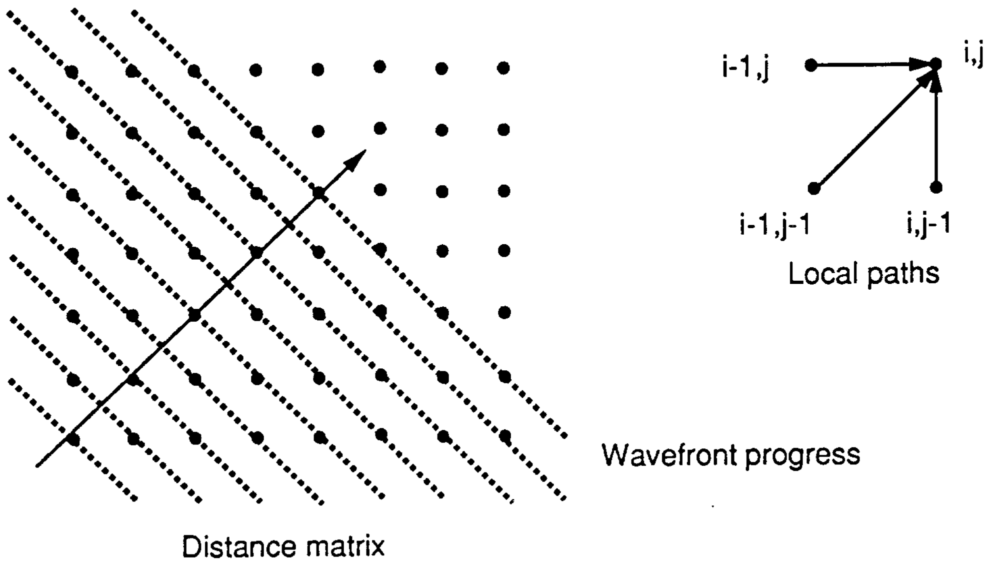


Figure 4.4: Dynamic Time Warp

A distance matrix for local distance computations between different frames of the speech template and reference is computed. The local distance matrix element computation proceeds by row partitioning. The topmost level activation in the local distance activation tree subdivides the matrix into two halves. This subdivision continues until the leaf activations calculate the local distances along a row. The core of the algorithm is a global distance calculation which uses the local distance values computed and finds the minimum path distance from element $(1, 1)$ to element (N, M) where N is the template size and M is the reference size. Figure 4.4 shows how this distance is computed. The element (i, j) is computed as

```
D[i,j]  =  ld[i,j] if i = 1 and j = 1
        =  D[i-1,j] + ld[i,j] if j = 1
```

```
    = minimum(D[i-1,j-1], D[i-1,j], D[i,j-1]) + ld[i,j]
```

Thus, the computation follows a wavefront in time dependency. Since Id is a non-strict language, the progress of this wavefront is not explicit. If values in the local distance matrix or the global distance matrix are referred to before they are computed then the access gets deferred. The global distance computation also follows row partitioning of size one.

4.1.4 Speech

Speech (Appendix B.2) is a fairly large program for computing cepstral coefficients from speech samples. It applies a preemphasis function to the speech samples and then subdivides them into overlapping speech frames. A hamming window smoothing function is calculated and applied to each frame. Then a set of autocorrelation coefficients are calculated for each frame. Each correlation coefficient calculation corresponds to an activation. Hence there is a substantial pool of ready activations. Each frame also has a corresponding leaf activation for cepstral coefficients determination. The computation per frame for the cepstral coefficients is in the form of an iterative loop which generates a substantial amount of scratch arrays in the process. Cepstral calculation uses row partitioning of size one while the autocorrelation is computed by row and column partitioning of size $(1,1)$.

4.1.5 Recursive Matrix Multiply, Rmm

Recursive matrix multiply (Appendix B.3) partitions a matrix into subblocks until finally blocks of size $(1,1)$ are obtained. Thereafter one activation corresponds to one element of the result matrix. Each element is calculated by an inner product activation which uses a for loop. There are thus N^2 leaf activations in the multiply activation subtree.

Chapter 5

Performance of TAM on Sequent shared memory multiprocessor

This chapter presents results related to the performance of TAM on the Sequent. Execution times and speedup have been shown for the programs under consideration. The implications of the scheduling queue strategies as well as that of the memory manager are shown. Other statistics of interest are also presented.

5.1 Speedup and Normalized Speedup

Speedup($\mathcal{S}_N$) is the time it takes to execute a program on one processor to the time taken on N processors. A program meant to be executed only on a single processor, does not require the multiprocessor overhead needed to guarantee mutually exclusive access to shared data structures that may be modified by more than one processor simultaneously. Normalized speedup ($\mathcal{NS}_N$) is thus the execution time for the uniprocessor version to the time taken by the multiprocessor version on N processors. This normalization represents the actual gain obtained from using multiple processors.

$$\begin{aligned} \mathcal{S}_N &= \frac{T_{m1}}{T_{mN}} \\ \mathcal{NS}_N &= \frac{T_u}{T_{mN}} \end{aligned} \tag{5.1}$$

where T_u is the time for execution on the uniprocessor version and T_{mN} is the time taken in the multiprocessor version on N processors.

5.2 Single Queue and Multiple Queue Performance

We have previously discussed the advantages of parallelizing the frame scheduling queue. In this section, we present results that confirm them. The execution time, speedup and normalized speedup for the programs considered are shown in Figures 5.1,5.2,5.3,5.4 and 5.5. Both the single queue implementation (SQ) and the multiple queue implementation (MQ) are shown. The memory manager was used to obtain these results.

Figure 5.1 shows the execution time and speedups for Fibonacci of 25. The execution times for MQ are significantly lower than SQ. MQ shows a good linear speedup. A speedup of 9.9 is obtained on 12 processors. This corresponds to a normalized speedup of 7.2. Fibonacci has a large amount of parallelism. There is a large pool of active frames present during most of the execution time which causes the processes to essentially work from their own queues with negligible stealing. Stealing occurs at the start and towards the end. The start time stealing causes the distribution of subtrees in the fibonacci activation tree among the processors. Thereafter the processes work essentially within their own subtree and out of their own queue. There is a certain amount of increase in queue steals towards the end when some processes finish their subtrees and look around for more work from processes which have not yet completed.

The SQ version shows an increase in speedup, though much less than MQ, until 8 processors. The peak speedup achieved is 4 corresponding to a normalized speedup of 2.9. The reason that the MQ version does better than the SQ version is the active frame access bottleneck posed by the single queue compared to parallel queues. Processes get serialized during frame access from a single scheduling queue. The scheduling queue bottleneck is evident from the percentage scheduling queue access failure (PSQF) which measures the percentage of queue access attempts that failed. SQ has PSQF of 4.9% on 2 processors which rapidly increases to 85.4% on 12 processors.

Results for Easy are shown in Figures 5.2 and 5.3. The second parameter to Easy which is the delay parameter decides the quantum size for the leaf *delay* activations. A higher delay value results in a larger quantum size for the *delay* activations. Easy with parameters of 15,20 shows speedup curves similar to Fibonacci. The SQ version shows poor speedup compared to the MQ version. It peaks at 9 processors at a value of 4.6. Increasing the delay reduces the contention for the scheduling queue. Hence both the single queue and the multiple queue versions perform almost as well for a delay of 200. The speedup observed on 12 processors is 10.1 corresponding to a normalized speedup of 9.2 for MQ, and 9.1 corresponding to 8.3 normalized speedup for SQ.

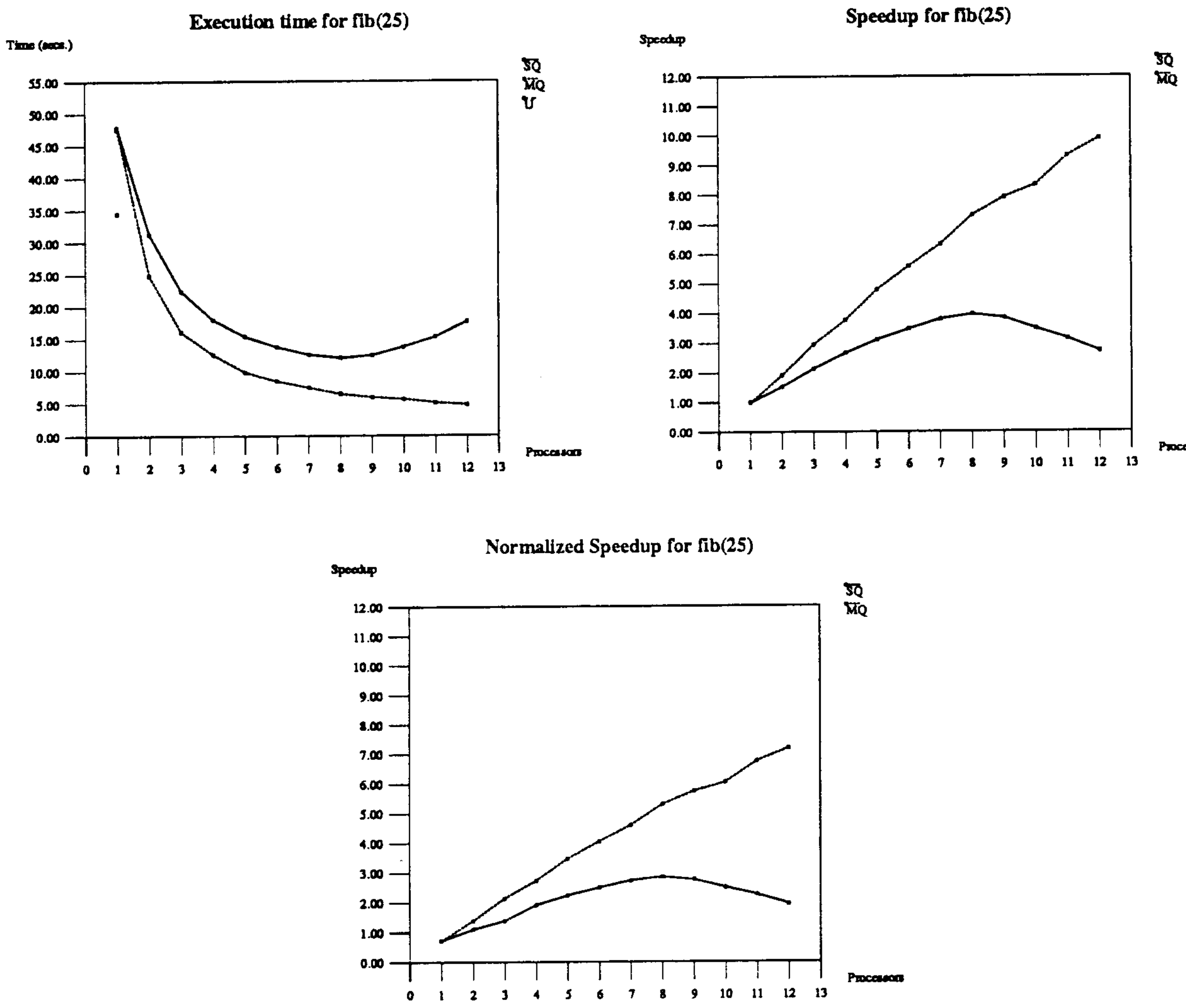


Figure 5.1: Fibonacci 25

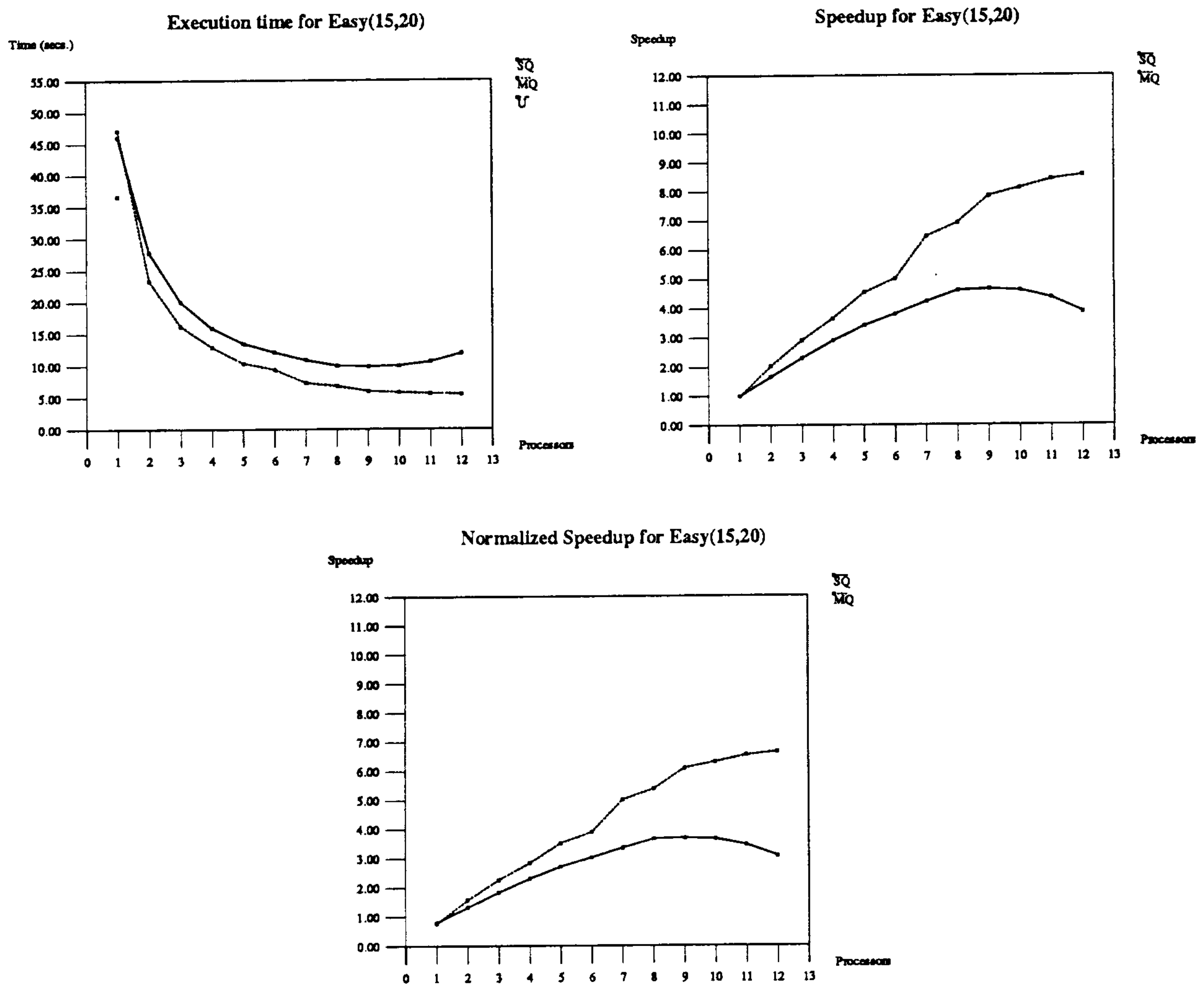
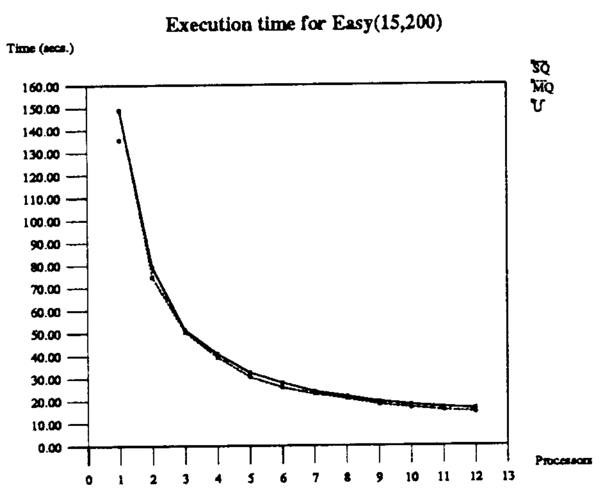
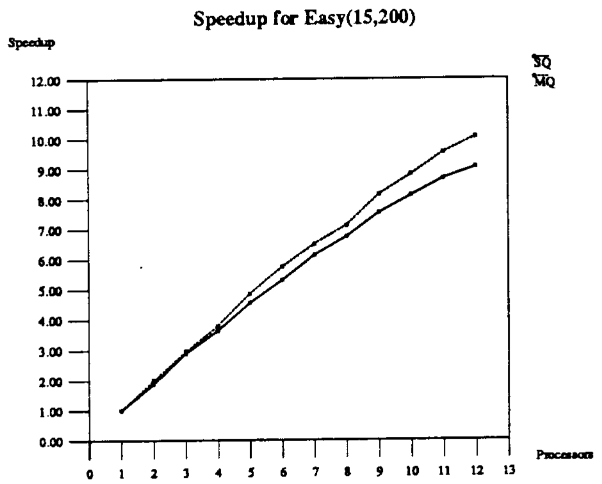


Figure 5.2: Easy 15,20





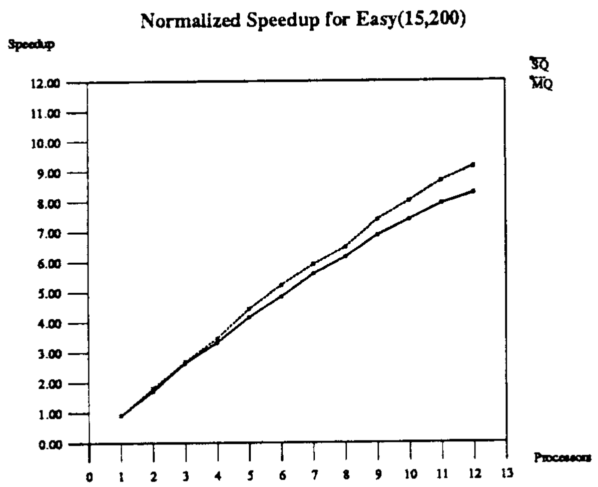
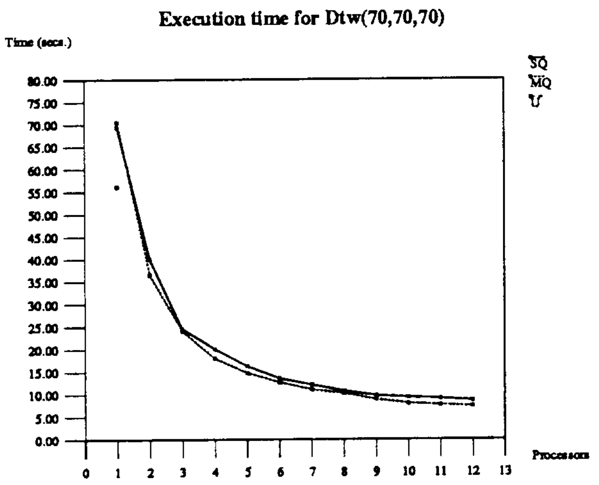
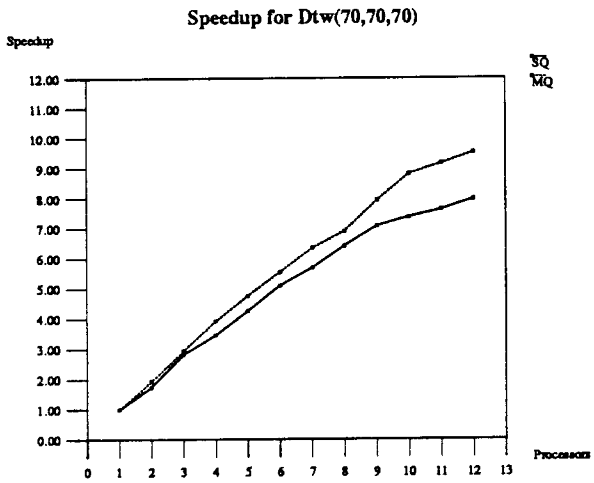


Figure 5.3: Easy 15,200





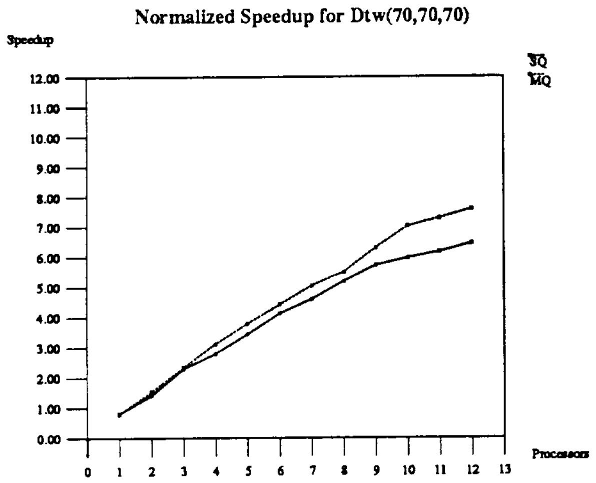
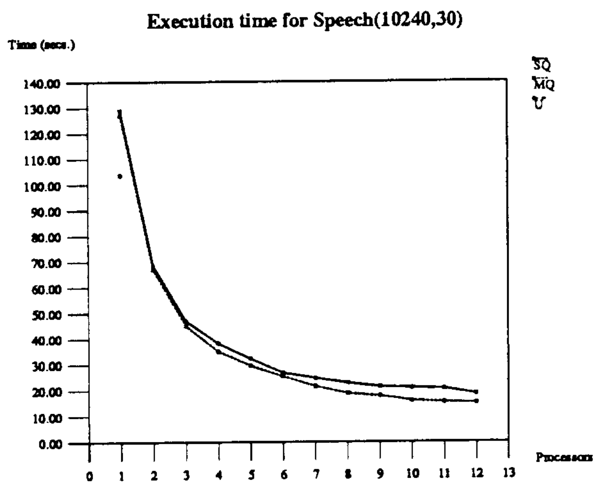
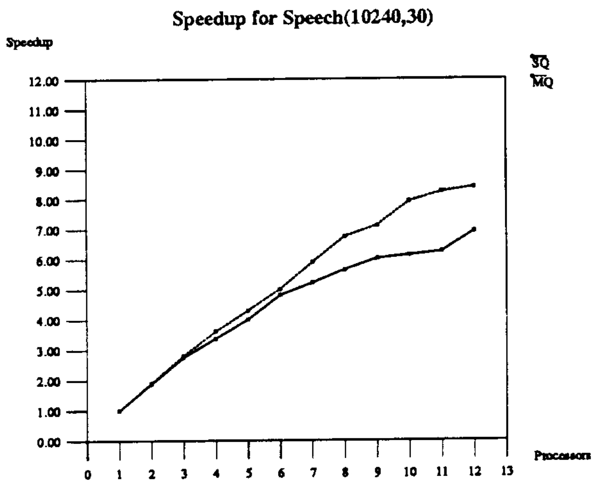


Figure 5.4: Dynamic Time Warp 70,70,70





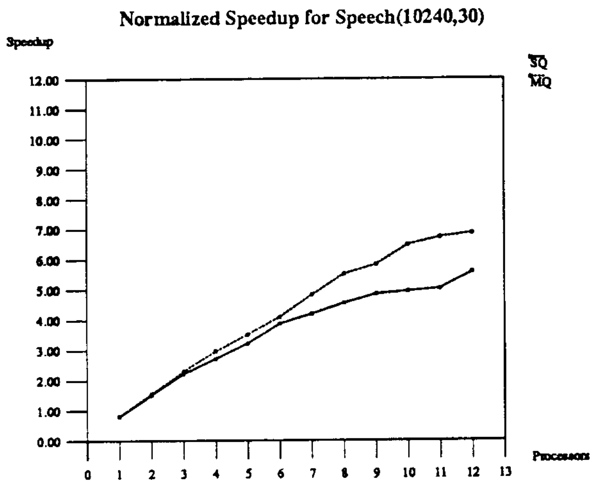


Figure 5.5: Speech 10240,30

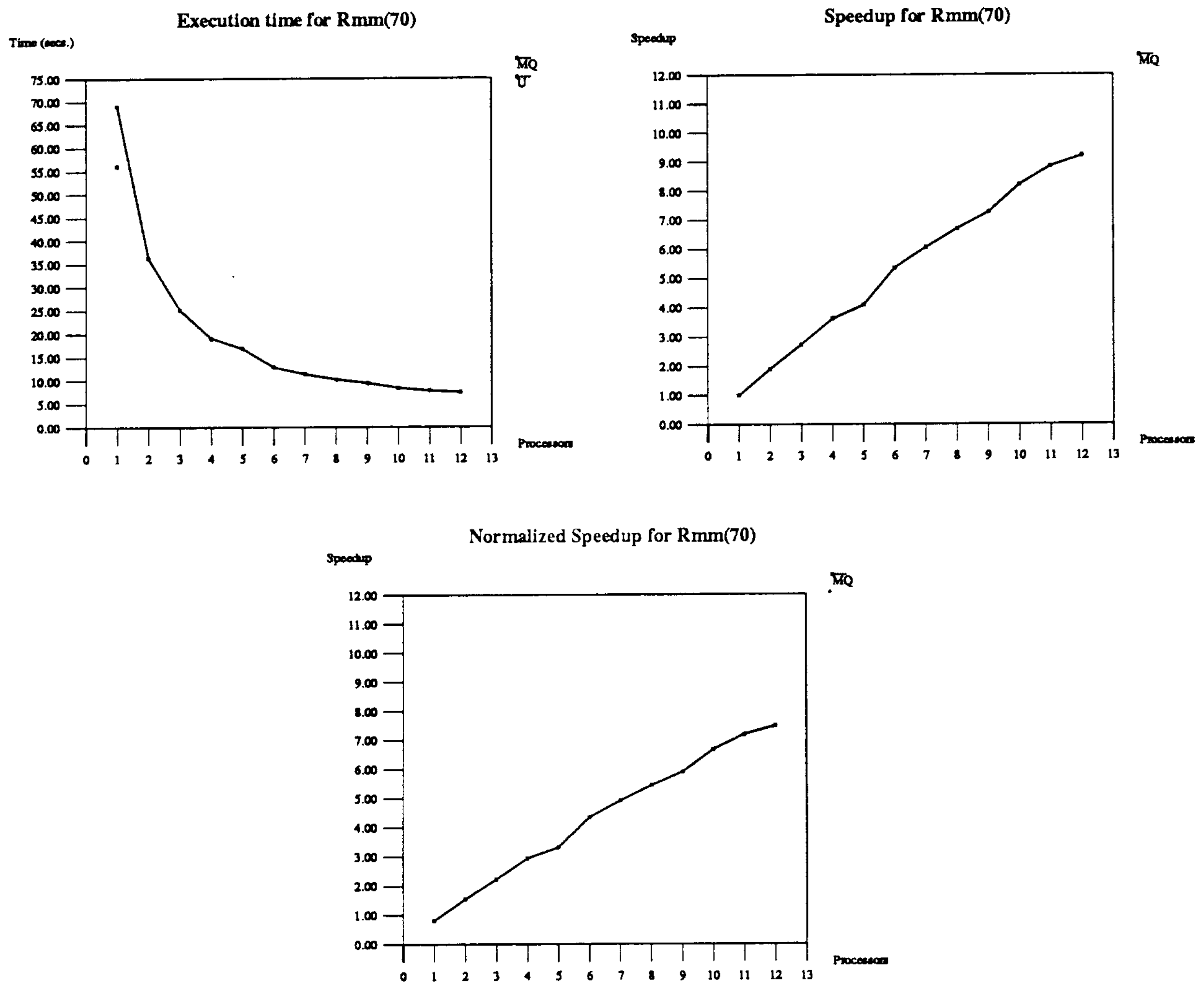


Figure 5.6: Recursive Matrix Multiply 70

Dynamic Time Warp and Speech show a better speedup in the SQ version than that seen in Fibonacci or Easy. The speedup curves do not peak but show a steady increase as seen in Easy with a delay of 200. The reason is that the average quantum size obtained in these programs is higher. Section 5.4 discusses the effects of quantum size on the speedup observed in SQ in more detail. The speedup seen on 12 processors is 8.0 for dynamic time warp and 7.0 for speech. MQ shows a speedup of 9.5 for Dtw and 8.4 for Speech.

The recursive matrix multiply program generates a large number of activations for generation of the matrices and for calculation of the inner product for the elements of the result matrix. Figure 5.6 show the performance results for the multiple queue version. A speedup of 9.2 on 12 processors corresponding to a normalized speedup of 7.5 is observed.

5.3 Frame Memory Management

Figure 5.7 shows the performance obtained with the memory manager and the shared memory allocator function call (shmalloc) to the parallel programming library. Speedup for Fibonacci of 25 is presented. In SQ , the memory manager yields slightly higher performance than shmalloc but beyond 9 processors

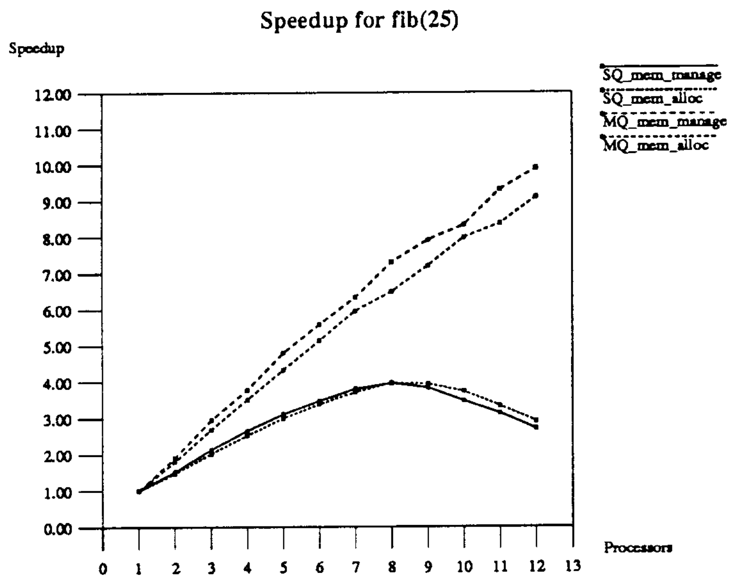


Figure 5.7: Speedup for Fibonacci 25 with memory manager and with shmalloc

turning the manager off gives better performance. This is due to the fact that with the manager, the

Processors		1	2	4	8	12
Fib(25)	SQ	7.25	6.84	6.38	6.09	6.14
	MQ	7.25	7.26	7.26	7.27	7.27
Easy(15,20)	SQ	15.17	14.09	12.73	12.20	12.21
	MQ	15.17	15.17	15.17	15.18	15.18
Easy(15,200)	SQ	75.17	69.26	64.85	62.14	60.79
	MQ	75.17	75.17	75.15	75.15	75.14
Dtw(70,70,70)	SQ	94.25	74.57	62.13	58.16	57.37
	MQ	94.25	92.45	90.66	88.09	82.77
Speech(10240,30)	SQ	404.97	315.38	358.83	321.59	283.42
	MQ	404.97	362.85	376.02	353.62	313.11

Table 5.1: Threads per quantum for Fib, Easy, Dtw and Speech

processes are not serialized in the frame allocation process and return to the scheduling queue at a faster rate. This increases the contention for the queue decreasing performance. The manager does much better in MQ where faster queue access does not result in increase of queue contention. The speedup performance gain obtained on 12 processors is 8%. Thus we see that proper frame management is essential for preventing the frame allocation process from becoming a bottleneck.

5.4 Effects of Quantum Size

The quantum size decides the amount of time the processors execute threads before the scheduling pool is accessed. It plays a significant role in the single queue version where a lower number of threads executed per quantum results in a faster return by the processors to the scheduling queue. This increases the queue access contention and significantly reduces speedup performance. Table 5.1 shows the variation of the threads executed per quantum (TPQ) for Fibonacci, Easy and Dynamic Time Warp. Fibonaci shows a fairly small TPQ. In MQ, the TPQ remains almost constant at 7.26 as the number of processors are increased. SQ however shows decreasing TPQ figures. This is due to the fact that the processors tend to serialize each other. As a result, less threads are effectively posted to the remote continuation vector area which results in the decrease of TPQ.

The effect of quantum size on the single queue performance is more clearly demonstrated by Easy in which the average quantum size can be tuned with the help of the delay parameter. The TPQ seen in Easy on one processor with a delay of 20 is 15.17. Increasing the delay to 200 increases the quantum size to 75.17 threads per quantum. This reduces the scheduling queue contention and improves the speedup significantly.

Dynamic Time Warp has a fairly large average TPQ. As a consequence, SQ does not perform significantly worser than MQ. The quantum size decreases for both SQ and MQ unlike Fibonacci and Easy where it remains constant for MQ. This is due to the istructure operations in Dynamic Time Warp which are not present in the other two programs. Istructure IFETCH operation is carried out in two parts. The first part initiates the action of accessing the istructure slot while the second part executes an inlet of the requesting activation which stores the value in a frame slot and pushes threads into the remote continuation vector. If the access gets deferred, the inlet is executed when an istore operation is carried out on the istructure slot. Higher the number of resident frames, higher is the probability that the access gets deferred. The threads posted by the inlet then have a higher probability of executing in another quantum. This causes the reduction in TPQ. TPQ seen in SQ on 1 processor is 94.25 and drops to 57.37 on 12 processors. Another reason for the larger reduction in TPQ for SQ, is that in the multiple queue case the frames are scheduled on a processor's own queue. This activation remains ready until the same processor picks it up for execution, except in the case when stealing occurs. However if there is substantial parallelism, stealing is rare. This increases the time that a ready activation spends on a scheduling queue and hence the probability of having larger number of threads in its continuation vector when it is finally picked up for execution. In MQ, the decrease is much less from 94.25 to 82.77. This is the basis for the better performance of MQ compared to SQ. Speech also shows a reduction in the quantum size for SQ and MQ due to the extensive istructure operations. The MQ version shows a smaller reduction in quantum size and hence a better speedup performance. Thus we see that multiple queues with stealing provides a significantly better quantum behavior compared to the single queue version.

The effects of quantum size are also seen in the comparison of execution times of the uniprocessor version with the multiprocessor version running on one processor. Table 5.2 depicts the percentage overhead in executing the uniprocessor version over the multiprocessor version with multiple queues on one processor. Easy with a delay of 20 shows 28.7% higher execution time. This drops to 9.96% with a delay of 200. One of the primary reasons for this difference is that when the quantum size is small, the fraction of time spent in the multiprocessor overhead to the total time is significantly higher. Fibonacci shows an overhead of 37.68%.

Even though the quantum size is higher in Dynamic Time Warp than Easy with a delay of 200, the percentage overhead between the uniprocessor version and MQ on a single processor is much higher (25.44%) compared to 9.96% for Easy. The reason is not hard to find. It must be realized that a single quantum does not necessarily mean that during the quantum no multiprocessor overhead

Programs	Percentage overhead
Fib(25)	37.68
Easy(15,20)	28.69
Easy(15,200)	9.96
DTW(70,70,70)	25.44
Speech(10240,30)	22.25
Rmm(70)	22.99

Table 5.2: Percentage overhead between uniprocessor version and multiple queue multiprocessor version on one processor

Programs	NS
Fib(25)	3.688
Easy(15,20)	2.164
Easy(15,200)	1.108
DTW(70,70,70)	1.176
Speech(10240,30)	1.232

Table 5.3: Ratio of speedups between single queue and multiple queue on 12 processors

occurs. Dynamic Time Warp does istructure operations. These operations require an overhead while checking for tags and data values giving an overall higher overhead than Easy.

Table 5.3 portrays the ratio of the normalized speedups on 12 processors for the multiple queue and the single queue case for various programs. Fibonacci under MQ shows a 268.8% improvement in normalized speedup than under SQ. Similarly, Easy with 20 delay shows 116.4% improvement. These two cases show such a marked improvement because of the high rate at which new activations are generated and the low TPQ value which causes the scheduling queue to become the bottleneck in the SQ version. Easy with 200 delay, Dynamic Time Warp and Speech show less marked but significant improvements ranging from 10-25%.

Chapter 6

Conclusions

In this report, we have presented an implementation of the Threaded Abstract Machine on bus-based shared memory multiprocessors. The implementation provides an efficient performance environment for parallel programs. We also present a study of performance metrics in shared memory multiprocessors related to lock contention and shared memory management. We have shown that the scheduling queue bottleneck posed by a single queue implementation is eliminated by using a multiple queue strategy. Multiple queue with stealing further provides better frame caching and also a better quantum behavior for the activations. Also using a bin based shared memory manager rather than the shmalloc function call directly, reduces the effective shmalloc call rate and provides for better processor utilization and speedup. Reduction of cache transfers by padding cache lines of frequently used data is essential for performance. In short, implementing efficient support for a parallel language on shared memory multiprocessors requires doing everything the right way. A single factor can cause significant loss in performance.

6.1 Acknowledgements

We would like to thank the members of the Parallel Computation Structures group for their help and support throughout the project. In particular, we wish to thank David E. Culler, John Wawrzynek, Thorsten von Eicken, Klaus E. Schauser and Seth C. Goldstein.

Appendix A

C programs

A.1 Programs for lock contention study

A.1.1 Single lock

```
/*
 * Single lock
 */
#include <parallel/parallel.h>
#include <usclkc.h>

#define MAX_CPUS        14
#define MIL_LCK         1000000

shared slock_t  *lck;
shared int  flag = 0;
shared int  cnt[MAX_CPUS];
shared sbarrier_t  *bp;
shared usclk_t   start_time, finish_time, total_time;

main(argc, argv)
int  argc;
char  **argv;
{
        int  i;
        int  avg = 0;
        int  num = atoi(argv[1]);
        void  test();

        S_INIT_LOCK(&lck);
        s_init_barrier(&bp,num);

        m_set_procs(atoi(argv[1]));
        m_fork(test,atoi(argv[2]));
        m_kill_procs();
        for(i=0;i<atoi(argv[1]);i++)
                printf("Id = %d Count = %d\n", i, cnt[i]);
        for(i=0;i<atoi(argv[1]);i++)
                avg += cnt[i];
        printf("Time = %f secs. MLocks per sec %f\n",
            (float)total_time/(float)1000000 , (float)avg/(float)total_time);
}

void
test(iter)
int  iter;
{
        int  count = 0;
        int  myid = m_get_myid();
        int  rnd;

        srand(myid);
        S_WAIT_BARRIER(&bp);

        if(!myid) {
                usclk_init();
                start_time = getusclk();
                flag = 1;
        }
        while (flag == 0) { .
        }

        while (flag == 1) {
                rnd = rand()%MIL_LCK;
                S_LOCK(&lck);
                count++;
                if (count == iter) flag = 2;
                S_UNLOCK(&lck);
        }
        cnt[myid] = count;
        if(!myid) {
                finish_time = getusclk();
                total_time = finish_time - start_time;
        }
}
```

A.1.2 Unique lock per process

```
/*
 * As many locks as processes.
 * Each process grabs its own unique lock
 */
#include <parallel/parallel.h>
#include <usclkc.h>

#define CPUS 14
#define MIL_LCK         1000000

shared slock_t  *lck[CPUS];
shared int  flag = 0;
shared int  cnt[CPUS];
shared sbarrier_t  *bp;
shared usclk_t  start_time, finish_time, total_time;

main(argc, argv)
int  argc;
char  **argv;
{
        int  i;
        int  num = atoi(argv[1]);
        int  avg = 0;
        void  test();

        for(i=0;i<atoi(argv[1]);i++)
                S_INIT_LOCK(&lck[i]);
        s_init_barrier(&bp,num);

        m_set_procs(atoi(argv[1]));
        m_fork(test,atoi(argv[2]));
        m_kill_procs();

        for(i=0;i<atoi(argv[1]);i++)
                printf("Id = %d Count = %d\n", i, cnt[i]);
        for(i=0;i<atoi(argv[1]);i++)
                avg += cnt[i];
        printf("Time = %f secs. MLocks per sec %f\n",
            (float)total_time/(float)1000000 , (float)avg/(float)total_time);
}

void
test(iter)
int  iter;
{
        int  count = 0;
        int  myid = m_get_myid();
        slock_t **lck_addp = &lck[myid];
        slock_t **lck_addp1;

        srand(myid);
        S_WAIT_BARRIER(&bp);

        if(!myid) {
                usclk_init();
                start_time = getusclk();
                flag = 1;
        }
        while (flag == 0) {
        }

        while (flag == 1){
                lck_addp1 = &lck[rand()%MIL_LCK];
                S_LOCK(lck_addp);
                count++;
                if (count == iter) flag = 2;
                S_UNLOCK(lck_addp);
        }
        cnt[myid] = count;
        if(!myid) {
                finish_time = getusclk();
                total_time = finish_time - start_time;
        }
}
```

A.1.3 Random locks from a finite number of locks

```
/*
 * As many locks as processes.
 * Each process grabs random locks.
 */
#include <parallel/parallel.h>
#include <usclkc.h>

#define MAX_CPUS  14
#define MIL_LCK         1000000

shared slock_t  *lck[MAX_CPUS];
shared int  flag = 0;
shared int  num;
shared int  cnt[MAX_CPUS];
shared sbarrier_t  *bp;
shared usclk_t  start_time, finish_time, total_time;

main(argc, argv)
int  argc;
char  **argv;
{
        int  i;
        int  avg = 0;
        void  test();

        num = atoi(argv[1]);
        for(i=0;i<atoi(argv[1]);i++)
                S_INIT_LOCK(&lck[i]);
        s_init_barrier(&bp,num);

        m_set_procs(atoi(argv[1]));
        m_fork(test,atoi(argv[2]));
        m_kill_procs();

        for(i=0;i<atoi(argv[1]);i++)
                printf("Id = %d Count = %d\n", i, cnt[i]);
        for(i=0;i<atoi(argv[1]);i++)
                avg += cnt[i];
        printf("Time = %f secs. MLocks per sec %f\n",
            (float)total_time/(float)1000000 , (float)avg/(float)total_time);
}

void
test(iter)
int  iter;
{
        int  count = 0;
        int  myid = m_get_myid();
        slock_t **lck_addp;

        srand(myid);
        S_WAIT_BARRIER(&bp);

        if(!myid) {
                usclk_init();
                start_time = getusclk();
                flag = 1;
        }
        while (flag == 0) {
        }

        while (flag==1){
                lck_addp = &lck[rand()%num];
                S_LOCK(lck_addp);
                count++;
                if (count == iter) flag = 2;
                S_UNLOCK(lck_addp);
        }
        cnt[myid] = count;
        if(!myid) {
                finish_time = getusclk();
                total_time = finish_time - start_time;
        }
}
```

A.1.4 Million locks

```
/* Million locks.
 * Each process grabs random locks.
 */
#include <parallel/parallel.h>
#include <usclkc.h>

#define MAX_CPUS 14
#define MIL_LCK 1000000

shared slock_t  *lck[MIL_LCK];
shared int  flag = 0;
shared int  num;
shared int  cnt[MAX_CPUS];
shared sbarrier_t  *bp;
shared usclk_t  start_time, finish_time, total_time;

main(argc, argv)
int  argc;
char  **argv;
{
        int  i;
        int  avg = 0;
        void  test();

        num = atoi(argv[1]);
        for(i=0;i<atoi(argv[1]);i++)
                S_INIT_LOCK(&lck[i]);
        s_init_barrier(&bp,num);

        m_set_procs(atoi(argv[1]));
        m_fork(test,atoi(argv[2]));
        m_kill_procs();

        for(i=0;i<atoi(argv[1]);i++)
                printf("Id = %d Count = %d\n", i, cnt[i]);
        for(i=0;i<atoi(argv[1]);i++)
                avg += cnt[i];
        printf("Time = %f secs. MLocks per sec %f\n",
            (float)total_time/(float)1000000 , (float)avg/(float)total_time);
}

void
test(iter)
int  iter;
{
        int  count = 0;
        int  id;
        int  myid = m_get_myid();
        slock_t **lck_addp;

        srand(myid);
        S_WAIT_BARRIER(&bp);

        if(!myid) {
                usclk_init();
                start_time = getusclk();
                flag = 1;
        }
        while (flag == 0) {
        }

        while (flag==1){
                lck_addp = &lck[rand()%MIL_LCK];
                S_LOCK(lck_addp);
                count++;
                if (count == iter) flag = 2;
                S_UNLOCK(lck_addp);
        }
        cnt[myid] = count;
        if(!myid) {
                finish_time = getusclk();
                total_time = finish_time - start_time;
        }
}
```

A.2 Programs for max_par

A.2.1 Shared memory allocator

```
main(argc, argv)
int argc;
char **argv;
{
        void delay();

        m_set_procs(atoi(argv[1]));
        m_fork(delay, atoi(argv[2])/atoi(argv[1]), atoi(argv[3]), atoi(argv[4]));
        m_kill_procs();
}

void
delay(iter, size, delay_time)
int iter;
int size;
int delay_time;
{
        int i,j;
        char dm_var;
        char *fp;

        for(i = 0; i < iter; i++) {
                fp = (char *)shmalloc(size);
                for(j = 0; j < delay_time; j++) {
                        dm_var = *(fp + j%size);
                }
                shfree(fp);
        }
}
```

A.2.2 Shared memory manager

```
#define MEM_SIZE        128     /* Number of buffers of size X to allocate */
#define SIZE128         16
#define NULL 0

typedef struct tlrun_membuf tlrun_membuf;
struct tlrun_membuf {
        tlrun_membuf *next;
};

/* Shared memory bins */
typedef struct {
        int count;
        tlrun_membuf *memp;
} tlrun_membin;

private tlrun_membin shmem_bin;
shared int cpus;

/*
 * Initialise shared memory bins' counters.
 */
void
mshmalloc_init(pid)
int pid;
{
        shmem_bin.count = 0;
}

/*
 * Return shared memory chunk from bin
 * corresponding to my process id. If bin is
 * empty, then fill bin with MEM_SIZE chunks.
 */
char *
mshmalloc(sz)
int  sz;
{
        tlrun_membuf    *tmp_memp;
        int size = SIZE128;

        if (shmem_bin.count) {
                tmp_memp = shmem_bin.memp;
                shmem_bin.memp = shmem_bin.memp->next;
                shmem_bin.count--;
        } else {
                int i;
                tmp_memp = shmem_bin.memp = (tlrun_membuf *)shmalloc(size);
                for (i = 1; i < (MEM_SIZE-1); i++) {
                        tmp_memp->next = (tlrun_membuf *)shmalloc(size);
                        if (tmp_memp->next == NULL)
                               printf("Out of memory in alloc\n");
                        tmp_memp = tmp_memp->next;
                }
                tmp_memp->next = NULL;
                tmp_memp = (tlrun_membuf *)shmalloc(size);
                if (tmp_memp == NULL)
                        printf("Out of memory in alloc\n");
                shmem_bin.count = MEM_SIZE - 1;
        }
        return (char *)tmp_memp;
}

/*
 * Place shared memory to be deallocated back in bin
 * corresponding to my procrss id.
 */
void
mfree(memp, sz)
char *memp;
int   sz;
{
        tlrun_membuf *tmp_memp = shmem_bin.memp;

        shmem_bin.memp = (tlrun_membuf *)memp;
        shmem_bin.memp->next = tmp_memp;
        shmem_bin.count++;
}

main(argc, argv)
int argc;
char **argv;
{
        void delay();

        cpus = atoi(argv[1]);
        mshmalloc_init();
        m_set_procs(atoi(argv[1]));
        m_fork(delay, atoi(argv[2])/atoi(argv[1]), atoi(argv[3]), atoi(argv[4]));
        m_kill_procs();
}

void
delay(iter, size, delay_time)
```

```
int iter;
int size;
int delay_time;
{
        int i,j;
        char dm_var;
        char *fp;

        for(i = 0; i < iter; i++) {
                fp = (char *)mshmalloc(size);
                for(j = 0; j < delay_time; j++) {
                        dm_var = *(fp + j%size);
                }
                mfree(fp, size);
        }
}
```

A.3 Cache transfer

```
shared int *touch;

int spin();

void main(argc, argv)
int argc;
char **argv;
{
    int procs = atoi(argv[1]);
    int iter = atoi(argv[2]);
    touch = (int*)shmalloc(sizeof(int));

    m_set_procs(procs);
    m_fork(spin, iter);
    m_kill_procs();
}

spin(iter)
int iter;
{
    int i;
    for (i=0; i<iter; i++)
        *touch = i;
}
```

Appendix B

Id programs

B.1 Dynamic Time Warp

```
%%
%% Dynamic time warp algorithm
%% Author : Anurag Sah
%%

typeof rmat1_plus = (I_Matrix F) -> I -> I -> I -> I -> I;
def rmat1_plus M rl rh cl ch =
  if (rl < rh) then {rm = div (rl + rh) 2;
                     _ = rmat1_plus M rl rm cl ch;
                     _ = rmat1_plus M (rm + 1) rh cl ch;
                    in 1 }
  else
    {for j <- cl to ch do
       M[rl,j] = float (rl + j);
     finally 1};

typeof rmat_plus = I -> (I_Matrix F);
def rmat_plus n = {M = (I_matrix ((1,n),(1,n)));
              _ =  rmat1_plus M  1 n 1 n;
              in M};

typeof rmat1_minus = (I_Matrix F) -> I -> I -> I -> I -> I;
def rmat1_minus M rl rh cl ch =
  if (rl < rh) then {rm = div (rl + rh) 2;
                     _ = rmat1_minus M rl rm cl ch;
                     _ = rmat1_minus M (rm + 1) rh cl ch;
                    in 1 }
  else
    {for j <- cl to ch do
       M[rl,j] = (float (rl - j));
     finally 1};
```

```
typeof rmat_minus = I -> (I_Matrix F);
def rmat_minus n = {M = (I_matrix ((1,n),(1,n)));
              _ =  rmat1_minus M  1 n 1 n;
              in M};

typeof fillr_local_distance = (I_Matrix F) -> (I_Matrix F) -> (I_Matrix F)
        -> I -> I -> I -> I -> I;
def fillr_local_distance loc_dist reference template refl refu temp_sz coeffs =
    if (refl < refu) then
        {refm = div (refl + refu) 2;
         _    = fillr_local_distance loc_dist reference template
                                 refl refm temp_sz coeffs;
         _    = fillr_local_distance loc_dist reference template
                                 (refm+1) refu temp_sz coeffs;
         in 1}
    else
        {for j <- 1 to temp_sz do
            dist_sum = 0.0;
            tmp_sum = {for k <- 1 to coeffs do
                        next dist_sum = dist_sum + (reference[refl,k]
                                                   - template[j,k]);
                       finally dist_sum};
            loc_dist[refl,j] = tmp_sum;
         finally 1};

typeof distance = (I_Matrix F) -> I -> I -> F;
def distance loc_dist ref_sz temp_sz =
    {glob_dist = (I_Matrix ((1,ref_sz),(1,temp_sz)));
     _ = rdistance glob_dist loc_dist 1 ref_sz temp_sz;
     in glob_dist[ref_sz,temp_sz]};

typeof rdistance = (I_Matrix F) -> (I_Matrix F) -> I -> I -> I -> I;
def rdistance glob_dist loc_dist refl refu temp_sz =
    if (refl < refu) then
        {refm = div (refl + refu) 2;
         _ = rdistance glob_dist loc_dist refl refm temp_sz;
         _ = rdistance glob_dist loc_dist (refm+1) refu temp_sz;
         in 1}
    else
        {for j <- 1 to temp_sz do
            min = if ((refl == 1) and (j == 1)) then
                      0.0
                  else if (refl == 1) then
                      glob_dist[refl,j-1]
                  else if (j == 1) then
                      glob_dist[refl-1,j]
                  else
                      {min1 = minf glob_dist[refl,j-1] glob_dist[refl-1,j];
                  in (minf min1 glob_dist[refl-1,j-1])};
         glob_dist[refl,j] = loc_dist[refl,j] + min;
          finally 1};

typeof minf = F -> F -> F;
defsubst minf x y = if (x < y) then x else y;

def main m n coeffs =
    {R = rmat_plus m;
     T = rmat_minus n;
     L = (I_Matrix ((1,m),(1,n)));
     _ = fillr_local_distance L R T 1 m n coeffs;
     in distance L m n};
```

B.2 Speech

```
%%
%% Front end Speech processing to determine Cepstral coefficients.
%% Author : Anurag Sah
%%

typeof S = (I_Array F);
typeof Ps = (I_Array F);
typeof Hw = (I_Array F);

typeof preemphasis = (I_Array F) -> (I_Array F) -> I -> I;
def preemphasis S Ps size =
    {S[0] = 0.0;
     _  = rec_preemphasis S Ps 1 size;
     in
        1};

typeof rec_preemphasis = (I_Array F) -> (I_Array F) -> I -> I-> I;
def rec_preemphasis S Ps lb ub =
     if (ub-lb) > 50 then
        {mb = lb + (div (ub-lb) 2);
         _  = rec_preemphasis S Ps lb mb;
         _  = rec_preemphasis S Ps (mb+1) ub;
         in
            1}
     else
        {for j <- lb to ub do
            S[j] = 1.0;
            Ps[j] = S[j] - 0.95*S[j-1];
         finally 1};

typeof hamming = (I_Array F) -> I -> I;
```

```
def hamming Hw window_size =
    {_  = rec_hamming Hw window_size 1 window_size;
     in
        1};

typeof rec_hamming = (I_Array F) -> I -> I -> I -> I;
def rec_hamming Hw window_size lb ub =
     if (ub-lb) > 10 then
        {mb = lb + (div (ub-lb) 2);
         _  = rec_hamming Hw window_size lb mb;
         _  = rec_hamming Hw window_size (mb+1) ub;
         in
            1}
     else
        {pi = 3.1415927;
         _  = {for j <- lb to ub do
                    Hw[j] = 0.54 -
                        0.46*(cos (2.0*pi*(float j)/((float window_size)-1.0)));
                finally 1};
         in
            1};

typeof auto_correlation = (I_Array F) -> (I_Array F) -> (I_Matrix F) -> I
        -> I -> I -> I -> I;
def auto_correlation Ps Hw Cm frames coeffs novrlp ws =
    {_  = recr_auto_correlation Ps Hw Cm 1 frames 0 coeffs novrlp ws;
     in
        1};

typeof recr_auto_correlation = (I_Array F) -> (I_Array F) -> (I_Matrix F)
        -> I -> I -> I -> I -> I -> I -> I;
def recr_auto_correlation Ps Hw Cm lbr ubr lbc ubc novrlp ws =
     if (ubr > lbr) then
        {mbr = lbr + (div (ubr-lbr) 2);
         _   = recr_auto_correlation  Ps Hw Cm lbr mbr lbc ubc novrlp ws;
         _   = recr_auto_correlation  Ps Hw Cm (mbr+1) ubr lbc ubc novrlp ws;
         in
            1}
     else
        {_  = recc_auto_correlation  Ps Hw Cm lbr ubr lbc ubc novrlp ws;
         in
            1};

typeof recc_auto_correlation = (I_Array F) -> (I_Array F) -> (I_Matrix F)
        -> I -> I -> I -> I -> I -> I -> I;
def recc_auto_correlation Ps Hw Cm lbr ubr lbc ubc novrlp ws =
     if (ubc > lbc) then
        {mbc = lbc + (div (ubc-lbc) 2);
        _   = recc_auto_correlation Ps Hw Cm lbr ubr lbc mbc novrlp ws;
        _   = recc_auto_correlation Ps Hw Cm lbr ubr (mbc+1) ubc novrlp ws;
        in
           1}
    else
       {correlation = 0.0;
        window_start = (lbr-1)*novrlp + 1;
        Corr = {for j <- window_start to (window_start+ws-1-lbc) do
                   next correlation = correlation + Ps[j]*Ps[j+lbc]
                             *Hw[j-window_start+1]*Hw[j-window_start+1+lbc];
                finally correlation};
        Cm[lbr,lbc] = corr;
        in
           1};

typeof cepstral = (I_Matrix F) -> (I_Matrix F) -> I -> I -> I;
def cepstral Cm Cp frames coeffs =
    {_  = rec_cepstral Cm Cp 1 frames coeffs;
     in
        1};

typeof rec_cepstral = (I_Matrix F) -> (I_Matrix F) -> I -> I -> I -> I;
def rec_cepstral Cm Cp lfb ufb coeffs =
    if (ufb > lfb) then
       {mfb = lfb + (div (ufb-lfb) 2);
        _   = rec_cepstral Cm Cp lfb mfb coeffs;
        _   = rec_cepstral Cm Cp (mfb+1) ufb coeffs;
        in
           1}
     else
       {E = I_array (0,coeffs);
        E[0] = Cm[lfb,0];
        k = I_array (1,coeffs);
        alpha = I_matrix ((1,coeffs),(1,coeffs));
        _ = {for i <- 1 to coeffs do
               k[i] = Cm[lfb,i] - (sigma alpha i Cm lfb);
               alpha[i,i] = k[i];
               _ = if (i==1) then 0
                   else
                      {for j <- 1 to (i-1) do
                          alpha[i,j] = alpha[i-1,j] - k[i]*alpha[i-1,i-j];
                       finally 1};
               E[i] = (1.0 - k[i]*k[i])*E[i-1];
             finally 1};
        Cp[lfb,1] = alpha[coeffs,1];
        _ = {for i <- 2 to coeffs do
               Cp[lfb,i] = Cp[lfb,1] + (lpc_sigma i alpha coeffs Cp lfb);
             finally 1};
```

```
         in
            1};

typeof sigma = (I_Matrix F) -> I -> (I_Matrix F) -> I -> F;
def sigma alpha i Cm frame =
     if (i==1) then 0.0
     else
        {sum = 0.0;
         in
            {for j <- 1 to (i-1) do
                next sum = sum + alpha[i-1,j]*Cm[frame,i-j];
             finally sum}};

typeof lpc_sigma = I -> (I_Matrix F) -> I -> (I_Matrix F) -> I -> F;
def lpc_sigma i alpha coeffs Cp frame =
    {sig = 0.0;
     in
        {for k1 <- 1 to (i-1) do
            next sig = sig +
                (1.0 - ((float k1)/(float i)))*alpha[coeffs,k1]*Cp[frame,i-k1];
         finally sig}};

typeof frontprocess = I -> I -> F;
def frontprocess samples coeffs =
    {novrlp = 384;
     window_size = 512;
     smpls = (div samples novrlp)*novrlp;
     frames = (div samples novrlp) - 2;
     S  = (I_array (0,smpls));
     Ps = (I_array (1,smpls));
     Hw = (I_array (1,window_size));
     Cm = (I_matrix ((1,frames),(0,coeffs)));
     Cp = (I_matrix ((1,frames),(1,coeffs)));
     _  = preemphasis S Ps smpls;
     _  = hamming Hw 512;
     _  = auto_correlation Ps Hw Cm frames coeffs novrlp window_size;
     _  = cepstral Cm Cp frames coeffs;
     in
        Cp[1,1]};

def main samples coeffs = frontprocess samples coeffs);
```

B.3 Recursive Matrix Multiply

```
%%
%% Recursive matrix multiply.
%%
```

```
typeof rmat1 = (I_Matrix F) -> I -> I -> I -> I -> I;
def rmat1 M rl rh cl ch =
  if (rl < rh) then {rm = div (rl + rh) 2;
                     _ = rmat1 M rl rm cl ch;
                     _ = rmat1 M (rm + 1) rh cl ch;
                    in 1 }
  else
    {for j <- cl to ch do
       M[rl,j] = float (rl + j);
     finally 1};

typeof rmat = I -> (I_Matrix F);
def rmat n = {M = (I_matrix ((1,n),(1,n)));
              _ =  rmat1 M  1 n 1 n;
              in M};

typeof ip = (I_Matrix F) -> (I_Matrix F) -> I -> I -> I -> F;
defsubst ip A B i j n = {sum = 0.0;
                         in {for k <- 1 to n do
                                next sum = sum + A[i,k]*B[k,j]
                             finally sum}};

typeof mmatr = (I_Matrix F) -> (I_Matrix F) -> (I_Matrix F) -> I -> I ->
               I -> I -> I -> I;
def mmatr C A B il ih cl ch n =
    if (il < ih) then {im = div (il + ih) 2;
                       _ = mmatr C A B il im cl ch n;
                       _ = mmatr C A B (im+1) ih cl ch n;
                       in 1}
    else
        { _ = mmatc C A B il cl ch n;
         in 1};

typeof mmatc = (I_Matrix F) -> (I_Matrix F) -> (I_Matrix F) -> I -> I ->
               I -> I -> I;
def mmatc C A B r cl ch n =
    if (cl < ch) then {cm = div (cl + ch) 2;
                       _ = mmatc C A B r cl cm n;
                       _ = mmatc C A B r (cm + 1) ch n;
                       in 1}
    else
        {C[r,cl] = ip A B r cl n;
         in 1};

typeof rmm = (I_Matrix F) -> (I_Matrix F) -> I -> (I_Matrix F);
def rmm A B n = {C = I_matrix ((1,n),(1,n));
                 _ = mmatr C A B 1 n 1 n n;
                 in C};

typeof main = F -> F;
def main n =  {A = rmat n;
               R = rmm A A n;
               in R[n,n]};
```

Bibliography

[Culler 91] David.E.Culler, Anurag Sah, Klaus E.Schauser, T.von Eicken, John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. *Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[Nikhil 90] R.S.Nikhil. Id (Version 90.0) Reference Manual. MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, September 1990.

[Thakkar 90] Shreekant Thakkar, Gary Graunke. Synchronization algorithms for shared-memory multiprocessors. Computer, v23, June 1990.