PARALLELIZATION SCHEMES FOR PHYSICS CODES

USING THE INDEPENDENT TIME STEP METHOD

Lise Storc
B.S., University of Texas, 1977

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

July, 1989

The thesis "Parallelization Schemes for Physics Codes Using the Independent Time Step Method" by Lise Storc has been examined and approved by the following Examination Committee:

Robert G. Babb II
Associate Professor, Thesis Advisor
Oregon Graduate Center

Michael Wolfe
Associate Professor
Oregon Graduate Center

Peter Eltgroth
Senior Physicist
Lawrence Livermore National Laboratory

Acknowledgements

I thank the following people for the great deal of encouragement, support, and kicks in the pants that it took for me to finish this thesis: my advisor Robert Babb, committee members Mike Wolfe and Pete Eltgroth, fellow current and former graduate students Dave DiNucci, Dan Lulich, Jacob Stein, Steve Rehfuss, Hitomi Ohkawa, Harry Porter, and Doug Pase, departmental staff Kathy Hammerstrom, Marie LaBonte, Kelly Atkinson and Bruce Jerrick, and registrar Margaret Day. Thanks also to Miki and Madhavi Desai for insisting, to Jane Allen for helping me to arrange the time, to Christopher English for true love, and to Mom and Dad for being there to see this happen at last.

Table of Contents

Trademarks

DYNIX is a registered trademark of Sequent Computer Systems, Inc.

HEP is a registered trademark of Denelcor, Inc.

Sun Workstation is a registered trademark of Sun Microsystems, Inc.

Symmetry and Pdbx are trademarks of Sequent Computer Systems, Inc.

UNIX is a registered trademark of AT&T Bell Laboratories.

Abstract

Parallelization Schemes for Physics Codes Using
The Independent Time Step Method

Lise Storc, M.S.
Oregon Graduate Center, 1989

Supervising Professor: Robert G. Babb, II

The Independent Time Step Method (ITSM) exploits the underlying parallelism in physics simulations by using local information to advance physical values in time. Spatial regions containing physical data are allowed to update asynchronously. Regions communicate with near neighbors, and mutual exclusion is required to assure consistency of updates. Also, the neighborhood of a region can change with time. ITSM is an example of a tightly coupled and relatively fine grained parallel algorithm based on an arbitrary dynamic graph.

Early attempts at parallelization of ITSM codes focused on allocation of work to processes. This was the starting point for this study of alternative scheduling and synchronization strategies for these codes, using models that incorporated the major features of the ITSM problem space. Experiments measuring the performance of the models over a range of parameter settings were carried out on a Sequent Symmetry. The models are differentiated based on the strategy a region uses when blocked by an executing neighbor, and by the time at which synchronization occurs. The performance of each model is discussed, and the results are compared. Results are also given for a full ITSM code based on a version of one of the models. This code achieved linear speedup.

CHAPTER 1

Introduction

In the real world, physical processes such as hydrodynamics, diffusion, and heat conduction occur in parallel. Early models for the physics of this class of problems were developed for sequential computers and relied on global information to advance the state of the computation, using sweeps across arrays. The Independent Time Step Method (ITSM) is a new method for the time advance of computation, and was developed to restore potential parallelism to the physical model, both for aesthetic reasons and for the possible speedup of execution when the model is implemented on a parallel processor. ITSM allows different spatial regions to advance at different rates, based on local information only. Independent regions can then potentially advance in parallel. ITSM in two or more dimensions has a dynamic neighbor relationship that significantly affects its parallel processing implementation. The parallel work management problem is thus dynamic and general enough to be of relevance to a large variety of parallel processing problems.

Attempts at parallelizing ITSM pre-dating this study focused on the allocation of work to multiple processors. The strategies used tended to suffer from parallel bugs or inefficiency. The difficulty in parallelizing ITSM lies in the dynamic neighbor relation and the fact that neighbors and potential neighbors cannot be allowed

execute simultaneously. ITSM is also inherently tightly coupled and fine grained, so that performance is quite sensitive to the efficiency of the code added to manage parallelism.

The primary goal of this study was to increase efficiency while maintaining correctness and avoiding parallel bugs. The first step in this process was to identify the major sources of inefficiency in previous parallelization strategies. The second step was to explore alternate strategies that dealt with these causes of inefficiency. A simplified model of an ITSM code was developed to allow multiple strategies to be tested and compared more easily than would have been possible using full ITSM codes, and to allow more general conclusions to be drawn. Also, the use of simpler testing codes reduced the frequency and severity of parallel bugs.

The remainder of this chapter describes the specific hardware and environments used for the early parallel codes and the subsequent model codes developed in this work. Chapter 2 gives a description of the Independent Time Step Method and the early attempts at parallelization of ITSM codes. Chapter 3 describes the sources of inefficiency in parallel ITSM codes, and describes the design of the simplified ITSM model and the parallelization strategies that were investigated using this model. Chapter 4 gives further details of the implemented strategies and the experimental results, including a strategy that was applied to a full ITSM 2D hydrodynamics code. Chapter 5 contains conclusions and future directions.

1.1. Overview of Hardware and Environment

Each implementation discussed in this paper was carried out on one of three computers: the Sun Workstation, the Denelcor HEP, and the Sequent Symmetry. The Sun Workstation is a uniprocessor and will not be discussed further. The Denelcor HEP was the first commercially available parallel processor, and will be covered briefly. The Sequent Symmetry is the parallel machine that was used for all experimental models and for the later full ITSM codes, and is discussed in more detail.

1.1.1. The Denelcor HEP

The Denelcor Heterogeneous Element Processor (HEP) [Den84a] was a shared memory machine running the HEP/UPX operating system, which was based on UNIX. Multitasking and sequential applications could execute simultaneously. Sequential applications ran transparently, while parallel applications made use of simple language extensions to create and synchronize multiple processes.

1.1.1.1. HEP Architecture

The HEP could consist of from 1 to 16 Process Execution Modules (PEMs), each with a general purpose 64-bit microprocessor, 64-bit floating point unit, and several local memories. A PEM could also access the shared global memory, called the Data Memory Module (DMM). Each PEM supported multiple processes by time-multiplexing its control and execution hardware and using pipelined logic. All

PEMs, memory modules and I/O controllers, were plugged into a single high-speed packet switching network.

The HEP had an access state for every data word in the DMM and for registers in a PEM. This state could be set to FULL or EMPTY. In addition, atomic operations were available to wait until EMPTY and write and set FULL and to wait until FULL and read and set EMPTY. An instruction encountering the wrong access state in a data memory or register was suspended until the access state changed. Thus, the FULL and EMPTY bits allowed for simple mutual exclusion of access to shared data.

1.1.1.2. Parallel Programming on the HEP

The HEP provided language extensions to FORTRAN 77 [Den84b] for the creation and synchronization of parallel processes and for accessing shared memory. Parallel processes were created via the CREATE statement. This statement was the same as a CALL, except that the calling program ran in parallel with the subroutine. The new process terminated at the RETURN statement. Synchronization was accomplished via the access states in data memory. Control of the access state of a variable was gained by giving the variable a name beginning with $. Various FORTRAN statements were provided for reading, writing, testing, and manipulating these asynchronous variables. Sharing of data was handled through common blocks, or through arguments to subroutine and function calls.

1.1.2. The Sequent Symmetry

The Sequent Symmetry Series [Seq87c] computers are shared memory multiprocessors running the DYNIX operating system, a version of UNIX 4.2bsd. Multitasking and sequential applications coexist in a multi-user environment. The Symmetry S81 incorporates up to 30 general purpose 32-bit microprocessors[1]. DYNIX achieves dynamic load balancing via a system-wide run queue used to schedule executable processes. Even if no effort is made to parallelize a user application, all DYNIX processes can potentially run in parallel. For example, Sequent has modified certain UNIX utilities, such as `make`, to allow them to run multitasked automatically. In addition, users can specify explicit parallelism in which DYNIX processes can share memory. Multitasking libraries are available to assist in writing parallel applications [Seq87a] [Sto88]. Sequent also offers an interactive parallel debugger, `pdbx` [Seq87b], based on the UNIX 4.2bsd debugger, `dbx`.

1.1.2.1. Sequent Architecture

The Symmetry Series computers contain up to 30 32-bit general purpose microprocessors, two to a board. A Symmetry system can contain up to 240 Mbytes of shared system memory. All processors, together with memory modules and I/O controllers, are plugged into a single high-speed bus, the Sequent System Bus. This bus supports pipelined I/O and memory operations and can sustain a data transfer rate of 53.3 Mbytes per second.

[1]Another model, the Symmetry S27, supports up to 10 processors.

The CPU used is the 16-MHz Intel 80386. All CPUs operate independently, and can execute both user code and system code. All processors share a single copy of the DYNIX kernel. Communication with other processors and subsystems is managed by System Link and Interrupt Controller (SLIC) chips via a special SLIC bus. A processor has a cache subsystem, which contains copies of the most recently read blocks of main memory. Each processor also has an Intel 80387 floating point unit and uses the memory management unit in the 80386 to handle 256 Mbytes of virtual address space per process.

1.1.2.2. The DYNIX Operating System

DYNIX schedules processes from a system-wide run queue. A process will run without interruption on a processor until it blocks (e.g., to wait for an I/O operation to complete) or terminates, or is pre-empted by another process with greater priority. At any point, the user process with lowest priority is pre-empted to service the interrupts. Therefore, a parallel application runs best with a higher priority than other user processes, and a number of processes one less than the number of processors available. It may also be advantageous to suspend processes that busy wait for a lock longer than some specified amount of time.

DYNIX allows two or more processes to share a common region of system memory. Blocks of shared memory are dynamically allocated to processes or process groups. This memory can either be local to a process or shared by a process group. From application languages, shared memory is read and written in the same way as ordinary program memory. In Fortran programs, shared variables are

statically allocated in COMMON blocks. In C programs, each shared variable can be either statically or dynamically allocated. This memory can be used for interprocess communication (between both related and unrelated processes), and is more efficient than using UNIX system calls to share data between processes. Shared memory is implemented as a portion of a process's virtual address space.

New processes are created via the `fork()` system call, which has an overhead of about 55 ms. The new (child) process is a duplicate copy of the current (parent) process, and inherits data, register contents, program counter, and any shared memory and file access possessed by the parent. In a typical application, a parent process acquires a region of shared memory and one or more locks, then forks some number of child processes to share the work. No child process is terminated until the program is complete.

1.1.2.3. Locks and Synchronization

In the Symmetry series, any byte of memory can be used as a lock via special Symmetry assembly language instructions that perform atomic test-and-set and test-and-clear operations [Seq87a]. These form the basis of routines to allow mutual exclusion of shared data structures. A lock can support a variety of synchronization techniques, including busy waits, counting/queueing semaphores, and barriers.

The lock routine `s_lock` in the DYNIX Parallel Programming Library (see Section 1.2.5 below) implements a busy wait. If the memory location serving as the lock is found locked, the processor spins until it is unlocked. The spin wait is carried out in cache memory, and puts no load on the System Bus. When a processor

unlocks a lock using `s_unlock`, the address of the lock is sent over the System Bus, and processors spinning on that lock will try again to lock it. A processor can be relinquished during potentially long busy waits by using a system call such as `sigpause()`, with an overhead of a few milliseconds. Such a processor can also be reacquired later.

1.1.2.4. Program Development

The Sequent parallel programming documentation [Seq87a] describes two basic types of applications, along with techniques for analyzing and dealing with order dependencies within them. In general, an order dependency is a point where a task depends on the result of a previous task and cannot proceed until the previous task is finished.

Function partitioning is appropriate when multiple processes perform different operations on the same data set. One model for this is the *fork-join*, where processes are forked, do separate tasks, then meet at a barrier. This is the correct model when tasks are relatively independent. Another model is the *pipeline*, which handles dependencies by allowing a process to perform calculations on a subset of the data, write the results to shared memory, then notify the downstream process that the results are available for further work. This is then repeated on the next subset of data. A process can suspend or terminate when there is no more work to do.

Data partitioning is appropriate when multiple identical processes can perform the same operation on different data, e.g., loops that perform calculations on arrays. This is also referred to as *homogeneous multitasking*. Data partitioning tends to lead

to automatic load balancing.

All programs undergoing parallelization must undergo some degree of dependency analysis to uncover potential problems. The *Sequent Guide to Parallel Programming* [Seq87a] contains guidelines to aid this process. In addition, the `gprof` utility can be used to analyze a program to determine which loops are good candidates for dependency analysis and attempts at parallelization. Running `gprof` on a program results in a call graph and a profile indicating where the program spends its time. If no loop that is reasonably independent and compute-bound can be found, then function partitioning can be tried.

1.1.2.5. The DYNIX Parallel Programming Library

The DYNIX Parallel Programming Library contains routines for creation, synchronization, and termination of parallel processes, and for management of shared memory in C, Fortran, and Pascal programs [Seq87a] [Sto88]. A preprocessor can be used to interpret compiler directives for loops in Fortran codes. The preprocessor automatically inserts code to create and synchronize processes executing the body of the loop in parallel.

A loop to be executed in parallel is placed in a subprogram and called using the `m_fork` function. This function forks a set of child processes and assigns an identical copy of the subprogram (and any non-shared data) to each process for parallel execution. The parent process also gets a copy to execute. The default number of child processes forked is half the number of CPUs online. The number of child processes can be set explicitly by first calling `m_set_procs`. If desired, the

number given to `m_set_procs` can be dependent on the number of actual CPUs available, which is determined by the function `cpus_online`.

After execution of the loop, the default action is to let the child processes spin until the next `m_fork` call. `M_fork` will then reuse the existing processes, saving the overhead of process creation. Alternatively, the child processes can be suspended via a call to `m_park_procs` and later resumed via `m_rele_procs`. This saves the overhead of spinning processes at a minimal cost. After the last call to `m_fork`, all child processes should be terminated using `m_kill_procs`.

Loop iterations within a subprogram can be allocated statically or dynamically. Library routines available to set up scheduling are `m_get_myid`, which returns the process identification number assigned by `m_fork`, `m_get_numprocs`, which returns the total number of child processes, and `m_next`, which increments a global counter.

Static scheduling assigns each process an equal number of iterations, and is appropriate if each iteration involves the same amount of computation. A typical algorithm for this is to get `n`, the total number of child processes, and execute every `n`th iteration starting with the process id.

Dynamic scheduling allows each process to obtain one or more loop iterations, execute them, and return to an iteration queue for more work. This type of scheduling is appropriate when the amount of computation in each iteration varies. If the loop iterations are sufficiently computationally intense, dynamic scheduling can be implemented easily and efficiently by using the global counter as the iteration queue. Otherwise, the queue can be implemented by creating a shared loop

index protected by a lock. Lock overhead and contention can be minimized by allowing a process to grab a large number of iterations at each visit to the queue.

The DYNIX Parallel Programming Library contains lock and barrier functions for use in synchronization. Lock routines `m_lock` and `m_unlock` interface to a single lock. `S_init_lock`, `s_lock`, `s_clock`, and `s_unlock` are used when multiple active locks are needed. `M_sync` sets up a single barrier for all active processes, while `s_init_barrier` and `s_wait_barrier` can be used to set multiple barriers and synchronize subsets of the processes.

A single-process section can be set up using `m_single` and `m_multi`. `M_single` halts the execution of child processes, allowing the parent process to perform some function such as I/O. Child process execution is resumed by calling `m_multi`.

`Shmalloc` and `shfree` are used for dynamic memory allocation in C. `Shbrk` and `shsbrk` change the size of a process's shared data segment, and `brk` and `sbrk` change the size of the private data segment.

1.1.2.6. Timing and I/O

The DYNIX system calls `gettimeofday()` and `getrusage()` are available from within programs and provide user, system, and wall clock times. Program executions can also be timed using the DYNIX system call `/bin/time`. However, this will also time I/O.

Multiple processes writing to the same file can cause problems. Also, processes tend to relinquish processors during disk accesses. Thus, it is best to put I/O in a sequential part of the code or to have an I/O server.

CHAPTER 2

The Independent Time Step Method

For many classes of physics problems, the goal of study is the behavior of a medium in the presence of some physical event or process. The system exists in an initial state, and subsequent states evolve in time via the interaction of regions within the medium. Models of the evolution of the system must obey appropriate physical constraints, such as *conservation of energy.* In addition, each state of the system must represent well-formed initial conditions. For example, the *Courant condition* says that a region's history cannot be allowed to influence the past history of another region, since this typically leads to numerical instability. To maintain computational accuracy, regions of the problem that are undergoing significant physical change typically must advance frequently, and by small amounts.

Since the propagation of information within a system is limited by travel time, it is possible to model the physics underlying its behavior with local information only. The maximum speed of propagation of information, such as the speed of light for heat conduction problems or the speed of sound for fluid flow problems, may vary during the evolution of the problem depending on the physical problem being considered. The *cone of information* of an updating region consists of all past updates that can affect the outcome of the update and all future updates that will

be influenced by the outcome. The new information at a region can thus be based on the most recent information from neighboring regions, which limits the amount of advance.

In computer modeling of physical systems, the methods chosen for propagating information and performing updates can constrain the overall efficiency of the computation. Since simulations typically involve large numbers of regions and advances of variables through time, parallelization can be crucial for achieving reasonable execution time. This is especially important in a multi-dimensional problem space. Also, since the underlying physical processes are inherently parallel, a parallel model is more aesthetically pleasing.

Cellular automata have been applied to simulation of physical processes such as diffusion and hydrodynamics [ToMa]. Solutions produced by such simulations do not suffer from numerical instabilities or diverging solutions. The computation is inherently parallel and localized. Problem partitioning and load balancing issues are avoided. However, work is done on regions that do not need updating, and, with existing hardware, a region is highly restricted in the complexity of computation it can perform. Physics computations must sometimes be over-idealized to accommodate this.

Implicit methods [Gr61] rely on old information in all regions of the problem to compute the new information at a given region. The corresponding computational models require global synchronization and thus restrict potential parallelism. The time advances are large and stable, limited only by accuracy requirements. However, because information is used from the entire problem, non-physical rates of

information transfer can cause degradation of accuracy.

Explicit methods use old information in a given region and neighboring regions to compute new information, allowing maximum independence of regions. The Independent Time Step Method (ITSM) is an explicit method developed by Peter Eltgroth of Lawrence Livermore National Laboratory for application to physics problems on parallel computers [EP85]. ITSM relies on the fact that most physical systems involve the exchange of information at some limited rate between space-time regions that are close together. In particular, hyperbolic systems (characterized by speed of sound) and parabolic systems (characterized by rate of information diffusion) can be treated within the framework of ITSM [Elt85] [Elt86]. Computation for such problems needs to take into account only a limited set of nearby regions to advance local information, with the amount of advance limited by the Courant condition. Therefore, regions can advance without reference to global information. Independent regions can update asynchronously, increasing potential parallelism. For typical problems, many regions share no information and can be advanced simultaneously.

In computational models of ITSM, the amount of advance is locally determined. Thus, the timestep can vary from region to region, and a given region can use different timesteps at each advance. There is no ordering imposed by the algorithm other than that implied by physical cause and effect. In a typical problem state, only a few regions are interacting in an interesting way. These regions use small timesteps to maintain accuracy, and physical bottlenecks are forced to advance. Uninteresting areas are updated only when necessary and can advance

much further in time when an update finally occurs. This reduces the total amount of computation. If there is enough work per update to justify the overhead, faster execution times can be achieved even for sequential code. In addition, the initial configuration for a problem modeled with ITSM can contain arbitrary start times for each region. Regions far from the action can be pre-advanced by the user in the problem input, and the work required to update them to the start time can be avoided.

The remainder of the chapter describes ITSM in more detail, and covers the previous approaches to parallelization of ITSM done by Peter Eltgroth and associates. Section 2.1 describes the ITSM computation, including implementation and parallelization issues as considered in the early code development. Section 2.2 discusses the scheduling strategies differentiating these codes and gives results. Section 2.3 gives conclusions about the early approaches.

2.1. Implementation of ITSM

Space is represented by a discrete set of cells (nodes) for computation. Fundamental data is kept with the nodes[1], which have attributes such as position, velocity, acceleration, and mass. Nodes also have a set of neighbor relationships with other nodes. Associated with each relationship are attributes such as volume, energy, and pressure, which are computed using knowledge of the partitioning of

[1]The choice of data structures in a computational model can also be the connections that carry information about physical relationships, or vertices that carry information about the physical partitioning of space. The choice of data structure strongly influences synchronization considerations.

space and time between nodes.

Nodes involved in the computation are considered to reside in a *work pool.* A process acquires a node from the pool and allows the node to advance the state of the problem by computing its *timestep,* then carrying out an *advance* of physical variables. As a result of the advance, more work is created and placed in the pool. In addition, neighbor relationships may change. The work routine is outlined in Figure 2.1 and discussed in detail in the following sections: timestep calculation and advance (2.1.1), neighbor relationships (2.1.2) and mutual exclusion of neighbors (2.1.3).

2.1.1. Timestep Calculation and Advance

For a node to be considered for update, it must first determine whether it is ready to advance. If it can advance, a timestep for the advance is computed. The decision to update physical variables depends on the result of the timestep calculation. If the timestep is necessary to maintain accuracy, physical variables such as time, acceleration, velocity and position are advanced to the new time. The node

```
get a node from the work pool
identify working nodes
compute the timestep for the node
if the node can advance
   advance the node
   remap the working nodes
put nodes into the work pool
(repeat)
```

Figure 2.1. Work routine executed by each process

uses only its own data and that of its neighbors in the computations. The only global information needed is the maximum rate of information propagation in the system. This is either a constant or an estimate based on values propagated to the node by neighbors.

In the timestep calculation, small amounts of shared data are read from neighbors and some of the node's own data is modified. The calculation involves two phases. The *timestep for information propagation* is computed as the minimum of the global timestep and the local timestep. The local timestep is based on the spread of information from neighbors only, and allows regions to advance with larger timesteps than a one pass global restriction would permit. The global timestep is based on the spread of information from all regions of the problem, and is required to prevent instability from the propagation of distant information. The simplest approach to this calculation would require global communication. However, ITSM avoids this by using a timestep estimate based on the minimum of the current estimates of neighboring regions. Accuracy is maintained if these estimates are updated with sufficient frequency. As repeated estimates of the global information are made, the global timestep increases. When it gets large enough, local timestep conditions dominate. At this point, a node can usually advance.

An *accuracy timestep* is also computed, which restricts an advance that would change the physical variables under study by more than a specified amount. The region can advance according to this timestep if it is smaller than the timestep for information propagation. Otherwise, an advance is not needed for accuracy, and the node advances by the timestep for information propagation only if it is restrict-

ing the advance of other regions. A small timestep for information propagation means the Courant condition is restraining update. The need for the accuracy timestep is to keep the computed solution close to that implied by the original equations.

2.1.2. Neighbor Relationships

A key feature of ITSM codes is the dynamic neighbor relationship. The connectivity of nodes in the underlying physical system is allowed to change with time. A node has a variable set of neighbors (both number and identity). In addition, nodes can be created dynamically. Fixed grids, whether Eulerian (fixed in space) or Lagrangian (fixed in mass or some other system property), are not imposed on a problem. The relaxation of time coordinate constraints is analogous to the relaxation of spatial coordinate constraints in the Free Lagrange approach to a hydrodynamics code. In the Free Lagrange approach, information is evaluated at spatial points (nodes) that move in concert with the fluid motion.

The neighbor relations between nodes must be maintained at runtime. The basis for the identification of neighbors is the Voronoi construction process for triangularization of the nodes described in [Duk81]. This process partitions space into regions that are closer to a given node than to any other. A node exchanges information with neighboring nodes only. The nodes involved in advancing a given node are the node itself, all immediate neighbors of the node, and all potential neighbors (neighbors of neighbors) of the node. After a node has advanced, these nodes must be remapped. First, the new connections are computed. Then, using geometric and

physical data, new vertices and volumes are computed for the node, its neighbors, and nodes that have added or deleted a connection. Quantities shared by the nodes in time, such as pressure, sound speeds, and timestep information, are advanced.

2.1.3. Mutual Exclusion of Neighbors

Since neighbor information must be read consistently during advance of physical variables, and since new neighbors may arise as a result of advance, a node must prevent both current and potential neighbors from updating simultaneously. It is also possible that a consistent state must be read from neighbors during timestep calculation. In all implementations discussed in this thesis, mutual exclusion was handled independently of work allocation by weaving locks into the work routine executed by all processes in parallel. This was done with the goal of minimizing operations inside scheduler critical sections, since it is not known until part way through an update which nodes will become new neighbors or ex-neighbors of the selected node.

In the early codes, nodes were grouped into *boxes* at initialization, each with an associated lock. Nodes were placed into boxes in near contiguous order[2]. Once the set of *working nodes* (the node itself together with current and potential neighbors) is identified, the set of boxes containing these nodes is determined. These boxes are then locked, and processes busy wait until all locks are obtained. To

[2]A dynamic alternative is to test nodes when they change neighbors, and swap boxes if a count of current neighbor boxes suggests a preference (e.g., Node A has more neighbors in the box of Node B and vice-versa).

prevent deadlock, the boxes are sorted and locked in increasing order, and unlocked in the same order. Boxes were introduced to minimize the number of locks and the potential for deadlock in the earliest version of a parallel ITSM code, and were retained in later versions. However, locking a node requires all nodes in its box to also be locked, including nodes that are not potential neighbors.

The locking scheme used in all early implementations was single phase. The scheme locks the working neighbors (and their boxes) of a node against any action before beginning the timestep calculation. The node is then updated, changing data and relationships as appropriate. Nodes that need further work are placed back into the work pool. Finally, the locking against simultaneous update is removed. An outline of the code with this locking scheme is given in Figure 2.2. This simple scheme was used to simplify coding, guarantee correct results, and reduce the chance of synchronization difficulties.

```
  get a node from the work pool
  identify working neighbors
* lock working neighbors
  compute the timestep for the node
  if the node can advance:
      advance the node
      remap the working nodes
  put nodes into the work pool
* unlock working neighbors
```

Figure 2.2. Single Phase Locking Scheme

2.2. Scheduling

The early implementations of ITSM can be differentiated primarily by the method of allocating nodes to processes. Other differences exist, such as the particular physical process modeled, but these are not relevant to the higher level view of ITSM computation. The static allocation of nodes to processes was never considered. The load balance for most problems would be poor since the regions of significant physical change are dynamic and unpredictable. Therefore, nodes needing work are placed in a pool, from which processes remove and insert nodes.

When a process seeks a node to work on, it consults the scheduler, which either returns a node or indicates termination. If the process receives a node, it proceeds with the ITSM computation. If the node could not advance, the restricting neighbor is placed in the pool and the process seeks another node. Otherwise, the advance is carried out and all nodes whose data has changed (including the node that has advanced) are returned to the pool. A node is not inserted if it is already in the pool or has been delivered to another process.

The first parallel coding ever attempted was carried out on the (now defunct) Denelcor HEP, in FORTRAN. The HEP had available as basic synchronization primitives EMPTY/FULL bits on variables. The sequential code for a one dimensional hydrodynamics problem was ported to a 1-PEM HEP by Peter Eltgroth. EMPTY/FULL bits were used for pertinent variables to achieve data update synchronization, creating a large number of critical sections. Each process determined its next task, and acquired nodes from a circular list. Critical sections surrounded task acquisition, task insertion, and decision making code. The code as originally

written never worked, suffering badly from deadlock. It also proved difficult to debug. Repeated attempts were made with coarser and coarser grain size, until the code finally worked when little parallelism was left. The critical sections were too large, causing much overhead and inefficiency.

In response to the coding and debugging problems encountered when using the HEP's asynchronous variables, Lusk and Overbeek of Argonne National Laboratory implemented *monitors*, a high level synchronization construct developed for operating systems. Their approach was to use macro-expansion of monitor constructs implemented using the low-overhead HEP primitives to achieve easier coding, debugging, and portability (across versions of HEP FORTRAN), without losing the efficiency possible with the use of asynchronous variables [Lusk83]. They were particularly interested in maintaining efficiency in tightly coupled algorithms (small granularity and high degree of communication). The macro package was later expanded to support a variety of machines, message passing and clustered system paradigms, and the C programming language [Boy87].

A monitor consists of shared data, initialization code and a set of operations on this data, plus the basic monitor operations *menter*, *mexit*, *delay* and *continue*. The *menter* and *mexit* commands together form a critical section for entering and exiting a monitor. Only one process can be active inside a monitor at a given time. A process attempting to enter a monitor must wait if the monitor is owned by another process. The *delay* command suspends the process active in the monitor and places it in a named queue (of arbitrary length). A process can delay itself if it cannot do required work. A process executing the *continue* command exits the mon-

itor, simultaneously activating a process from a named queue (which continues where it left off). If the named queue is empty, the effect of the *continue* is the same as *mexit*.

The standard monitor operations can be used to construct monitors such as self-scheduling loops and synchronization barriers. Problem termination can also be handled inside monitors. For locking simple shared variables, macros *lock* and *unlock* are provided as shortcuts to writing monitors for each variable.

The ASKFOR monitor was developed by Lusk and Overbeek to be general enough to handle all algorithms with the simple high level description of processes retrieving work from a work pool and inserting work into the pool (such as ITSM). However, specific ASKFOR monitors may vary considerably. For example, the work pool for different problems can require different representations. The decision making code for selection of a task from the pool can range widely in complexity. Another example is that termination may occur when a solution is found, or when all subcomputations of a decomposed problem are complete. Since a significant amount of problem dependent code can be included, a complex ASKFOR monitor can be as difficult to implement as a reasonably modularized "bare knuckles" parallel code.

The next three sections describe monitor use in ITSM: the basic ASKFOR, the queue of queues, and the distributed ASKFOR.

2.2.1. The Basic ASKFOR

This strategy uses a single ASKFOR monitor with a single critical section. A work pool of nodes is initialized and divided into subpools (the number of subpools is selected at startup), implemented as circular queues. Each node is assigned to a fixed queue for the duration of the execution. The division into sub-pools groups neighboring nodes together in an attempt to minimize the collisions that occur when two neighbors try to update simultaneously. Each process is assigned a home pool from which to withdraw and insert nodes. When a process is ready to remove or insert new work into the work pool, the ASKFOR monitor is called (with a busy wait if the monitor is blocked), and the critical section entered. The process first checks its home pool for nodes. If the pool is empty, the process is allowed to search other pools [3]. If all pools are empty, the process checks for termination, and either delays until work reaches the pools or terminates. If the process finds work, the node is removed from the pool and the process exits the monitor.

The basic ASKFOR strategy for a one dimensional hydrodynamics code was run on a 4-PEM HEP, and was the first working parallel version of the code. However, the single source of work allocation was a serious bottleneck. For problems with small grain size, the serial code within the critical section becomes a significant impediment to speedup.

---

[3]Other implementations of this strategy divided nodes based on the node id mod the number of queues and allowed processes to access pools in circular fashion.

2.2.2. The Queue of Queues

This strategy incorporated a simple improvement to the basic ASKFOR. It was possible that much effort was spent checking queues that held no tasks on which to work, so a master queue was added to keep track of non-empty queues. There is one critical section that is entered when work is needed for a process. The master queue is consulted to determine which work queue to use. A node is removed from that work queue and the critical section is exited. On completion of work on a node, a second critical section is entered, and appropriate nodes are placed into queues.

For the one dimensional hydrodynamics problem, the queue of queues strategy showed less parallelism than the basic ASKFOR or the distributed ASKFOR (see Section 2.2.3), but times were competitive. It was the most efficient parallelization strategy of the three [Elt86].

All early implementations of ITSM for two dimensional problems used the queue of queues strategy. The sequential version of a two dimensional Free Lagrange hydrodynamics code was ported from the HEP to a Sun Workstation by Peter Eltgroth, and used monitor macros developed for the Sun. The monitor macros were then rewritten for the Sequent, and the code was run in parallel. This code was also run sequentially on a Sun at OGC. There were two test problems, one the exact analog of the one dimensional hydrodynamics problem, the other an 80x5 rectangle.

At the same time that the two dimensional hydrodynamics code was under development, a code for two dimensional heat conduction was written for the Sun,

developed from scratch by Peter Eltgroth and Dave Turner. The monitor macros for the Sequent were used later used to run the code in parallel.

2.2.3. The Distributed ASKFOR

This strategy attempts to ease the bottleneck of a single critical section for work allocation by dividing the critical section of the basic ASKFOR into multiple critical sections protected by different locks. One critical section is used for each pool. No synchronization is necessary for checking the master queue to see whether a pool is empty. This reduces collisions at the monitor while adding some overhead (it is still possible for processes to collide when searching pools other than their home pool). Different pools could be manipulated in parallel.

The distributed ASKFOR strategy for the one dimensional hydrodynamics problem got a speedup of 21 on a 4-PEM HEP, but needed more processes than the queue of queues because of the additional overhead incurred from distributing the monitor. It also had the lowest overall execution times.

2.3. Conclusions about Early Approaches

The progression of ITSM implementations discussed in this chapter gives an interesting demonstration of the problems encountered in parallelizing a large and complex code. The first code used large numbers of fine grained critical sections in an attempt to maximize efficiency, but suffered from massive deadlock and debugging difficulties. The next code used a higher level construct for work allocation and

a large grained critical section for mutual exclusion during computation to prevent the deadlock and debugging problems. However, this caused the work allocation bottleneck to become apparent. The remaining codes attempted to avoid this bottleneck by trying more complex work allocation strategies, and showed the performance tradeoff between more complex scheduler strategies and scheduler efficiency.

The tendency to parallelize codes by starting at the lowest level and then proceeding by reaction to the most apparent problem underscores the desperate lack of useful tools and models for simplifying the parallelization process. Though some such models and tools exist [DiB89] [Ger85] [Jor87], they are all limited in the types of algorithms that can make effective use of them. ITSM is a highly dynamic, tightly coupled and fine grained algorithm. In addition, computational tasks operate under tight constraints. There are so many possible sources of inefficiency that it would take an exceptional parallel programming model or approach to capture all of the complexity of the algorithm. Although use of high level constructs for ease of coding and portability is desirable, it must be weighed against the possibility of using problem dependent constraints to reduce overhead and increase efficiency.

The abstraction provided by the ASKFOR monitor proved to be of limited use in speeding up the tightly coupled ITSM codes. The barrier to speedup is in some other area than allocation of nodes. However, use of monitors and the macroexpansion strategy are of great help in code organization and debugging, and add the bonus of portability.

The major difficulty with the ITSM implementations is the persistence of underlying assumptions and remnants of earlier codes into the next code. The goal was to decouple work allocation and computation to simplify the monitor procedure, both for ease of programming and to reduce the work allocation bottleneck. The most obvious sources of inefficiency in the resulting codes are the synchronization during ITSM computation, busy waits, degeneration of neighborhoods and excessive locking.

Synchronization is required at two stages, work allocation and node update. The first busy wait is encountered by a process attempting to remove (or insert) a node when the pool is locked. This wait cannot be avoided, but can potentially be relieved by distributing the pool critical section as in the Distributed ASKFOR strategy. However, the results obtained with this strategy were not as good as expected. One reason for this is the natural load balancing of ITSM. The process was allowed to search other pools rather than idle on an empty pool. Another reason is the degeneration of neighbor relationships within the pools. Since neighbor relations are dynamic, no initial allocation of work to pools can guarantee that work is spread across all pools for the duration of the problem. Therefore, processes eventually start to search other pools for work. The Distributed ASKFOR degenerates into the Queue of Queues strategy, but with greater overhead.

The second busy wait is encountered by processes attempting to lock the neighbors of a node against simultaneous update. There is potential for much loss of efficiency by processes colliding during computation, since a node obtained from the pool can be restricted or blocked by a neighbor (this possibility becomes more

acute in higher dimensions). A restricted node busy waits on locks before doing any computation, then blocks neighbors (who busy wait) until it is determined that the node cannot advance. Since neighborhoods degenerate at runtime, there is no way to group nodes to minimize blocking. The busy waits executed by blocked nodes are especially detrimental in these codes since each lock blocks out groups of nodes. And, again, since neighbor relations change, these groups of nodes eventually contain nodes that have no need to be excluded from updating. Also, the distribution of schedulable nodes across the node neighbor graph is dynamic, and problem dependent. This would cause deterioration of static node to queue assignment even if the node neighbor graph itself was static. The use of boxes of nodes limits the potential benefit of using multiple processes with a fixed number of nodes, and is only relieved by scaling the problem upward in size to reduce the chance of collision.

The fairness of the early strategies depends on whether a node seeking access to a lock is guaranteed to possess this lock eventually. There is nothing in these scheduling strategies to provide this guarantee. A process is allowed to busy wait until all locks needed by a node are acquired. However, it is theoretically possible for competing nodes to win repeatedly since there is no way to distinguish two nodes. Each lock controls an entire group of nodes. Therefore, the node and competing nodes may not have any relationship other than seeking to lock nodes belonging to the same group. Fairness is instead achieved by practical means. The waiting node is distinguished from competing nodes when these nodes become idle in the work pool. This is guaranteed by a reasonable node to process ratio [Fra86]. Unfortunately, allowing blocked nodes to suspend while retaining acquired locks degrades

performance severely as the percentage of locks held by blocked nodes gets large.

CHAPTER 3

ITSM Bottlenecks and Solution Strategies

The early parallelizations of ITSM codes focused on the bottleneck of allocating nodes to processes, and achieved moderate speedup. However, it was felt that better results should be possible. The goal of this work was first to uncover other sources of inefficiency and then to develop alternative parallelization strategies that might remedy them. A model capturing the most relevant aspects of full ITSM codes was used so that more parallelization strategies could be tested with greater ease, due the size of a typical ITSM code: 18,000 lines of C.

This chapter begins in Section 3.1 by listing the major bottlenecks in parallel ITSM codes and summarizes any approaches for reducing these bottlenecks taken prior to this work. Section 3.2 describes the simplified ITSM model developed here, along with alternative strategies for overcoming some of the bottlenecks. Finding an efficient parallelization strategy for ITSM involves many competing issues. Care must be taken not to introduce complex strategies that would add overhead in excess of any gain.

3.1. Bottlenecks

The primary targets for more efficient parallelization schemes for ITSM are the following bottlenecks:

- *Scheduling*
- *Blocked nodes*
- *Time restricted nodes*
- *Size of time advance*

Scheduling refers to the allocation of nodes to processes. A blocked node is one that attempts to obtain a lock that has been locked by a neighboring node. A time restricted node is unable to advance due to the impact of neighbors on the computation of its timestep. The size of time advance refers to the size of the timestep taken by an advancing node. These bottlenecks are discussed further in the following sections.

3.1.1. Scheduling

A primary bottleneck in ITSM codes is the allocation of nodes to processes. The probability of processes conflicting at the scheduler and being forced to busy wait should be reduced. At the cost of high overhead, the scheduling strategy might also attempt to prevent scheduling of blocked or restricted nodes. However, the increased overhead must be weighed against potential gain. The amount of overhead that is tolerable in a scheduler depends heavily on the granularity of the computation being scheduled. ITSM is a fine grained algorithm, and is especially

sensitive to scheduling overhead. Care must also be taken not to add non-essential synchronization or reintroduce global bottlenecks or deadlock.

The early parallelization schemes discussed in Chapter 2 tried several strategies for scheduling nodes. The first ASKFOR monitors made use of single critical sections and kept scheduling overhead down by their simplicity. The Distributed ASKFOR increased overhead while reducing the likelihood of contention at the scheduler by using multiple critical sections.

3.1.2. Blocked Nodes

The early codes also tried to reduce the probability of scheduling nodes blocked by neighbors by using various queue initialization strategies for multiple queues. However, since the node neighbor graph is dynamic in ITSM, any static assignment of nodes to queues deteriorates during execution. It is difficult to determine in advance which distribution of nodes to queues would result in a reduction of blocking.

An alternative to static assignment is to allow nodes to migrate into different queues as neighbor relationships change. In this case the initialization strategy becomes more important since the node space can be partitioned into neighborhoods whose integrity is maintained, reducing the chance of blocking. However, dynamic queue management adds overhead to the scheduling process, and to node computation, since a node must compute the best new neighborhood.

The possibility of node to node conflict cannot be removed altogether by any queue strategy, since nodes straddling the boundary of a neighborhood must still be scheduled. Also, processes must be allowed to access other queues should the home pool become empty, to achieve load balancing. It would be interesting, however, to study the effectiveness of various queue initialization and maintenance strategies by plotting the number of blocked nodes as a function of execution time. Since scheduling using multiple queues was studied in the early work, this was not pursued further. For simple schedulers that do not prevent the scheduling of blocked nodes, a random queue initialization scheme should perform as well as any other. The scheduling queue could still be broken up into multiple queues with separate locks as in the Distributed ASKFOR, in order to reduce the probability of contention at the scheduler.

3.1.3. Time Restricted Nodes

A node that is time restricted by a neighbor cannot do any real work (time advance the node and neighbors and update physics variables). In addition, the neighbors of a time restricted node are delayed from executing if they are blocked by the restricted node. It is desirable to prevent too much execution time from being taken up by these nodes. This issue was not addressed in the early schemes discussed in Chapter 2.

Avoiding the scheduling of time restricted nodes is difficult and would likely have high overhead. A better approach is to reduce the impact of a restricted node on overall execution time by such strategies as delaying locking until after the node

is known not to be time restricted.

3.1.4. Size of Time Advance

A node that is available to be worked on may not be able to advance very far in time. In some cases, the advance is required to maintain accuracy and its size will not change if the node is postponed. However, it is sometimes the case that the node would take a much larger timestep if it *was* postponed. This is preferable to a sequence of smaller timesteps, since the cost of computation is independent of the size of the timestep advance. This is referred to as *physics efficiency*. If better physics efficiency could be achieved, then the speedup could in fact become "superlinear" since the total number of advances would decrease.

The physics efficiency bottleneck is a complex problem and will not be discussed further. Solving it would likely entail a great increase in the size of scheduling overhead. Giving priority to areas where the most physics is taking place may involve significant runtime analysis of the problem state, and could even degenerate into a global algorithm.

3.2. Model Design and Parallelization Strategies

The following sections outline the design of a simplified model of ITSM and strategies for parallelization. The model uses a static grid of nodes with a simple queue for allocating nodes to processors. Time delays are used to represent various aspects of the physics computation. Input parameters, such as number of nodes,

number of neighbors, time to compute the timestep for a node and time to advance a node, are available to allow a wide range of ITSM problems to be modeled.

Several strategies for dealing with node-neighbor conflict are developed, along with a strategy for delaying node-neighbor synchronization. The ultimate goal of this work was to increase efficiency by reducing the impact of blocked and restricted nodes through the use of these alternative strategies.

3.2.1. Node Graph and Neighbor Relations

For ITSM codes with two spatial dimensions, the average number of neighbors for a node is approximately six, so the underlying node graph is roughly a hexagonal grid. The number of neighbors is dynamic, however, and can range from fewer than three to more than twenty. In higher dimensions, the average number of neighbors is much larger. In any case, a node must exclude current neighbors and potential future neighbors from simultaneous update. To precisely study the effect of dynamic neighborhoods on efficiency, a model could be based on an arbitrary dynamic graph, with parameters for number of nodes and average number of neighbors. These parameters affect the time required to identify neighbors, the time for remapping the neighborhood after update, and the probability of a node being blocked by a neighboring node.

For modeling purposes, however, nodes were organized in a static two-dimensional grid to simplify coding and analysis of results. This works as long as no assumptions are made that do not hold for the case of arbitrary dynamic neighborhoods. The total number of nodes was specified by input parameters for the grid

dimensions.

The number of neighbors is an input parameter and is the same for all nodes, except that nodes on or near the boundary will omit from their neighborhood those neighbors that lie outside the grid boundary. Legal values for the number of neighbors range from 0 to 25. In the case of 0 neighbors the neighborhood is empty. For other values, the neighborhood is constructed from the set of nearest neighbors according to the scheme in Figure 3.1.

The times required to identify neighbors and to remap neighbors after update were added to the model as delays computed as a function of the number of neighbors. These delays were also input parameters.

In the models, potential neighbors were considered a subset of the regular neighborhood. A node is blocked unless it can lock its own lock and the locks of its neighbors. The probability of a node blocking increases with the number of neighbors. In the case where nodes busy wait on locks, deadlock is a possibility. This is prevented by assigning each lock a unique identification number. Once the neighborhood of a node has been determined, the locks are sorted by ID and the locks acquired in increasing order. A lock ID is computed as a function of its grid

24 20 09 13 21
19 08 01 05 14
12 04 00 02 10
18 07 03 06 15
23 17 11 16 22

Figure 3.1. Neighborhood Scheme - For n neighbors, n>0, the neighborhood of the node at position 00 consists of the node itself together with nodes 01 through n-1.

position. The ID for a node at position `i,j is i*J+j`, where `J` is the total number of columns in the grid.

3.2.2. Scheduling

In ITSM, all nodes are placed in the work pool initially. Though it is possible to preadvance nodes in the input data, this would increase the sequential initialization portion of the code significantly and limit speedup. Thus, the models also place all nodes in the pool at initialization.

Since the problem of contention at the scheduler was studied in the early parallelization schemes, the models made use of a single FIFO queue for scheduling nodes, protected by a lock against simultaneous update. As discussed earlier, the dynamic nature of the node graph in ITSM renders efficient queue initialization and maintenance strategies ineffective in reducing the likelihood of blocking. The queue used in the models was initialized by placing nodes in the queue in random order. This scheme should be as effective as any other for the full dynamic ITSM code, and is essential in both full ITSM and the static grid models to prevent parallel processes from repeatedly accessing sets of neighbors should they remain in synchrony.

3.2.3. Blocked Nodes

Another possibility for reducing the impact of blocked nodes is to use different strategies in the event a node is blocked. For simple schedulers, several strategies

can be tried:

- *process busy waits on the lock*
- *process suspends node, retains locks*
- *process suspends node, relinquishes locks*

In the first strategy, the number of processes is a critical parameter, since processes are timesliced during the busy wait. This is the strategy that was used in the early parallelization schemes described in Chapter 2. In the latter strategies, there is no need for more processes than processors since each process treats nodes as virtual processes. Note that the second approach is equivalent to allowing processes to suspend themselves when the node they are working on becomes blocked. Deadlock is an issue in the first two strategies, while fairness is the main issue in the third approach.

The first and third strategies were explored in the models, together with a strategy combining these two approaches. The second strategy was not explored. Any simple scheduling scheme using this strategy becomes highly inefficient as the number of suspended nodes gets large, since the probability of a node blocking increases with the number of locks held by other nodes. This also applies to the first strategy when the number of processes grows larger than the number of processors. More complex schedulers were assumed to add too much overhead to the scheduler for the grain size of the node computation. The strategies used in the models are discussed in the following sections.

3.2.3.1. Busy Wait

The *busy wait* strategy allows a node to spin wait on each neighbor lock (see Section 1.1). A single level locking scheme where a node accesses the lock directly forces the node to busy wait. With the Sequent machine, spinning on hard locks can strain the bus, so that soft locks are used instead.

In the busy wait strategy, no scheduling overhead or neighbor identification and locking time is wasted. However, a node that is blocked on a neighbor that is in the early stages of computation may have a potentially long wait that is only relieved by the process holding the node being timesliced by the operating system. This relief might then be offset by the increase in the likelihood of blocking caused by locks held by suspended processes. Also, neighbor locks must first be sorted to prevent deadlock.

This strategy was modeled for comparison with the other strategies, and is essentially the strategy used in the early schemes discussed in Chapter 2. However, this strategy is more efficient than that used in the early work, since boxes are not used to group locks. In the early schemes, the use of boxes increased the probability of blocking significantly, thus reducing efficiency.

3.2.3.2. Relinquish

The *relinquish* strategy attempts to reduce the effect of nodes busy waiting while neighbors do computation by allowing a process to return a blocked node to the scheduling queue and continue searching for an unblocked node. All acquired

locks are released before the node is returned to the queue. Since the time required to identify neighbors is small in ITSM compared to the subsequent computation, it was possible that this strategy might be more efficient than the busy wait strategy as long as scheduler overhead and neighbor locking time are also small enough.

The locking scheme used in the busy wait strategy cannot be used when a process needs to take alternate action with a blocked node, though some scheme is essential to prevent read/write conflict. Synchronization cannot be accomplished on a binary flag alone, since it would be possible for two processes to read the flag as UNLOCKED before either sets it to LOCKED, and fail to achieve mutual exclusion [Ray86]. However, a simple two level locking scheme can be used. The critical section is associated with a flag protected by a lock. A process trying to enter the critical section must obtain the lock (a busy wait) before reading the flag. If the flag is set to LOCKED, the process is blocked from entering the critical section, and can proceed with an alternate action after releasing the lock. If the flag is set to UNLOCKED, the process can update it to LOCKED, release the lock, and enter the critical section. See Figure 3.2. At the end of the critical section, the flag can be directly updated to UNLOCKED, since the process that owns the flag is the only one that can write it. Since deadlock is not an issue in this strategy, no sorting of neighbor locks is necessary.

```
lock the node lock
if the node flag is UNLOCKED, change it to LOCKED
unlock the node lock
```

Figure 3.2. Locking a node with two level scheme

3.2.3.3. Timestamps

The *timestamp* strategy is a compromise between the busy wait and relinquish strategies. A node acquires a unique timestamp before acquiring neighbor locks. During the locking process, a node timestamps all acquired locks. If the node blocks, it follows the busy wait strategy if it is the first node to block on that neighbor. Otherwise it follows the relinquish strategy. See Figure 3.3 for an outline of how a single neighbor is locked. The routine is executed repeatedly until the lock is obtained or the node is blocked by a node with an earlier timestamp.

The timestamp strategy reduces the probability that a node will have to wait through computations by a sequence of other nodes, as is possible in the busy wait strategy, while also reducing some of the waste incurred from blocked nodes releasing locks in the relinquish strategy. The version of the two level locking scheme shown in Figure 3.3 is used to allow the alternative action to busy wait for blocked nodes.

```
lock the neighbor lock
if the neighbor is unlocked
   timestamp the neighbor
   lock the neighbor flag
   unlock the neighbor lock
   return unblocked
else the neighbor is locked
   if the neighbor stamp is earlier
      unlock the neighbor lock
      return blocked
unlock the neighbor lock
```

Figure 3.3. Locking with the Timestamp Scheme

3.2.4. Time Restricted Nodes

In ITSM, the two fundamental operations carried out on the data associated with a node are the timestep calculation, in which it is determined whether and how far a node can advance in time, and the advance itself. There are several possible strategies for synchronizing neighboring nodes during these operations that achieve proper mutual exclusion. Depending on the strategy used, the impact of time restricted neighbors on efficiency can vary. Two strategies are discussed in the following sections.

3.2.4.1. Early Synchronization

The *early synchronization* strategy requires a node to lock all neighbor nodes before carrying out any computation. No two neighboring nodes can work in parallel. This strategy was used in the early parallelization schemes discussed in Chapter 2. For an outline of early synchronization, see Figure 3.4.

```
 get a node from the work pool
 identify neighbor nodes
*lock neighbor nodes
 compute the timestep for the node
 if the node can advance:
     advance the node
     remap the neighbor nodes
*unlock neighbor nodes
 put nodes into the work pool
```

Figure 3.4. Early Synchronization

3.2.4.2. Late Synchronization

The *late synchronization* strategy takes advantage of additional information to reduce the effect of time restricted nodes on efficiency. Since the node computation can be taken in two phases, the goal is to identify the potential parallelism available in both phases of the update, so that a more efficient parallel scheme can be employed. If possible, a node would not block on a neighbor that is not actually being updated.

An exclusion scheme must normally synchronize nodes to prevent both read-write and write-write conflicts. However, in ITSM, it may be possible to allow a node to carry out the first phase of the update reading data belonging to a node that is being advanced. This read-write conflict can be allowed if the node in the first phase does not need a consistent state from the data being read in order to do the timestep calculation. (However, additional small critical sections are necessary when the write of a single datum is not atomic). By delaying the exclusion of a node until it is in the advance phase, it is possible for the node to execute the first phase in parallel with other nodes executing the second phase. In addition, delayed exclusion also allows nodes in the first phase to execute in parallel. This scheme is outlined in Figure 3.5.

The early synchronization scheme not only prevents write-write conflict between nodes in the advance phase, and read-write conflict between nodes in the check phase and nodes in the advance phase, but also prevents read-read actions between nodes in the check phase.

```
get a node from the work pool
identify neighbor nodes
compute the timestep for the node
if the node can advance:
   *lock neighbor nodes
    advance the node
    remap the neighbor nodes
   *unlock neighbor nodes
put nodes into the work pool
```

Figure 3.5. Late Synchronization

With early synchronization, a node that is time restricted prevents neighbors from executing either computational phase. In the late synchronization scheme, such a node has no impact on efficiency other than that which can only be prevented by reducing or eliminating the scheduling of time restricted nodes. Late synchronization also improves efficiency by reducing the probability of blocking, since the amount of computation in the critical section is smaller.

3.2.5. Timesteps and Advance

In ITSM, a node that has advanced puts neighboring nodes whose data has changed into the work pool. A time restricted node puts the restricting node into the pool. However, in the model, a node does not actually change the data of neighbors, so that the neighbor nodes are not returned to the work pool after an update. A time restricted node releases all locks and is returned to the scheduling queue.

To eliminate the physics efficiency question from the models, timesteps of constant size were used so that no order of update is more efficient than any other. Each node represented its state of advancement by a count of the number of times

it had advanced. Time restriction of a node was modeled using constraints of varying tightness. A node was allowed to advance as long no neighbor had seen more than n fewer updates, where n was a parameter for the tightness of the constraint.

The statistics on execution in one dimensional ITSM typically show timestep calculation taking 10-50% of the execution time of physical variable update, and occurring 3-6 times as often. No statistics were available for two dimensional ITSM. The times for computing the timestep, determining whether the node could advance, and carrying out the advance itself were modeled by delays given by input parameters.

Since models with constant timesteps were used, termination occurred when each node advanced a preset number of times, given by an input parameter. This was handled by using a critical section to allow processes to update a shared counter after each advance. Nodes were not returned to the scheduling queue after they had been advanced enough times. The scheduler detected termination by examining the shared counter to check the total number of advances.

CHAPTER 4

Experiments and Results

Though the issues and ideas raised in the previous chapter seem plausible, experiments were carried out in an attempt to verify the ideas. This chapter gives the results of the experiments, and includes comparative analysis. The experiments are differentiated based on the action taken when a node is blocked (busy wait, relinquish, timestamp) and the synchronization scheme employed (early, late). A version of a strategy used with the full ITSM code is described, and the results are discussed. First, implementation details of the simplified ITSM model are given, including an overall description of the trials.

4.1. Implementation Details

All experiments were carried out on an eight processor Sequent Symmetry S81, running Dynix V3.0.12. Code was written in C, and used the DYNIX Parallel Programming Library (see Section 1.1.2).

Input Parameters The set of input parameters used in the experiments is summarized in Figure 4.1. The parallelization strategies were tested on a range of values of these parameters. At first, single runs were timed for a very wide range of

number of processors	1,3,5,7
number of nodes (grid dimensions)	100 (10x10)
number of neighbors of each node	1,9,25
total timesteps for each node	100,200
constraint tightness	1
time to compute timestep and check advance	0-25 ms
time to advance and remap neighbors	0-25 ms

Figure 4.1. Input parameters

values. The results were examined to determine which sets of parameters were useful, redundant, or unrealistic as regards ITSM. The useful sets of parameters (using values given in Figure 4.1) were then tested extensively and averaged over all runs. The goal was to select a parameter range covering the full spectrum of ITSM problems, and to push the simplified ITSM model and parallelization strategies through worst case scenarios still potentially relevant to full ITSM.

Performance results depended primarily on the particular strategy, the probability of blocking and the check/advance ratio. The *check/advance ratio* refers to the ratio of the time to compute timestep and check advance to the time to advance and remap neighbors. The *probability of blocking* is a function of the number of processors, the number of nodes, the number of neighbors of each node, and also on the check/advance ratio and the *granularity* of the computation (essentially, the check time plus the advance time). In order to assess *speedup*, the number of processors was varied. This, plus allowing neighbors to range in number, produced a full range of blocking probabilities in a fixed number of nodes.

The times to compute timestep and check advance and the times to advance and remap neighbors were collapsed since the actions occur consecutively in full

ITSM codes and are either both outside of critical sections or occur in the same critical section. These times were modeled by variable delays. The identification of neighbors must be carried out by every strategy, and always takes place prior to neighbor locking, so that the time to identify neighbors can be subsumed for modeling purposes in the time to compute timestep and check advance. Identification time is still relevant when comparing strategies, since the busy wait and timestamp blocking strategies must sort neighbors to prevent deadlock while the relinquish strategy does not. However, this was accounted for by a quicksort in the code, and not by a variable delay.

The total number of timesteps for each node was varied significantly, but had a primarily quantitative effect for numbers relevant to ITSM codes. Also, computation time increases roughly linearly with the number of timesteps. Therefore, the number used in most experiments was a relatively small 100, with an increase to 200 to illustrate the effect as the number of timesteps increases. Loose constraints have a quantitative rather than a qualitative effect on the comparative runtimes of the various strategies. Thus, constraint tightness was fixed at 1.

Shared Data Structures A node is a structure containing its grid coordinates. The scheduling queue consists of a linked list of nodes, together with pointers to the head and tail of the queue and a lock to enforce mutual exclusion. Initially, all nodes are placed in the queue in random order. Depending on the locking scheme used, there is either a grid of node locks (one lock for each set of node coordinates), or a grid of node locks plus a corresponding grid of node flags. Where deadlock is a problem, there is also a corresponding grid of node IDs that is initialized once. A

(non-shared) dynamic length queue of nodes is available to each process to store neighborhoods. A grid of node timestep counts is used to update the state of time advancement of a node.

Scheduling The scheduling queue is protected by a queue lock. Access to the queue results in a busy wait if another process is accessing it. If the queue is nonempty, a node is obtained from the head of the queue. Otherwise, the scheduler checks for termination. The queue lock is then unlocked. This procedure is executed repeatedly until either a node is returned or termination is detected. Nodes are replaced at the tail of the queue under protection of the queue lock[1].

Main Procedure The main procedure reads input parameters, initializes the scheduling queue and other shared data, then forks off child processes using `m_fork` (see Section 1.1.2.5) and joins in the work. The number of processes forked is equivalent to the number of processors. At termination, the child processes are killed and timing and statistics are printed.

Timing and Statistics The experiments were timed using the DYNIX `getrusage` call, which allows the user times for the parent process and the total user times for all child processes to be accessed. The calls were made in the main routine after termination of the child processes, and the result for the main routine was added to that of the children to get the total user time. The average time per process was then computed. The results reported in the performance graphs of this

[1]Note that separate locks can be used for getting a node from the head of the queue and putting a node at the tail of the queue whenever the head and tail pointers are distinct. This would reduce conflict at the queue, but was not implemented.

chapter show the average user time per process as a function of the number of processors. `getrusage` does not allow the times for individual processes to be accessed. However, both ITSM and the model strategies are naturally load balanced so that differences between processes was not an issue. All initialization and I/O was done outside of the timed code so that only the comparative performance of the strategies in the parallel work routine was assessed.

Statistics on runtime behavior were recorded to help understand poor performance and suggest improvements. These included the number of nodes accessed, restricted, blocked and advanced. To prevent the introduction of additional critical sections, each process kept local statistics in a global shared array indexed by the process ID. These statistics were then printed and totaled in the main routine after child process termination and timing.

4.2. Busy Wait

This section describes the performance of the busy wait strategy. Section 4.2.1 covers early synchronization, while Section 4.2.2 covers late synchronization. Section 4.2.3 discusses the addition of noise into these two strategies.

4.2.1. Early Synchronization

In the busy wait strategy with early synchronization, a process first obtains a node from the scheduling queue. The neighbor set for the node is computed and the neighbors are sorted to prevent deadlock. Neighbor locks are acquired in increasing

order using the single level locking scheme described in Section 3.2.3.1. This requires the process to busy wait on a node that has been locked by another process. The node is checked to see if it can advance by comparing its timestep with those of its neighbors. If the node is not restricted, it is advanced. After the advance, each neighbor lock is unlocked. Finally, the node is returned to the scheduling queue.

A representative graph showing the performance of this strategy is given in Figure 4.2. For all input parameter sets, the execution times for a fixed number of processors increased with the number of neighbors, as in the sample graph. This difference is at least partly the result of the increase in computation time for locating and sorting the neighbor set, and the increase in granularity from longer check and advance times. However, the strategy also suffers from general speedup difficulty when there is a positive probability of blocking (with only one neighbor, the scheduled node locks only itself and there is no blocking). The loss of speedup

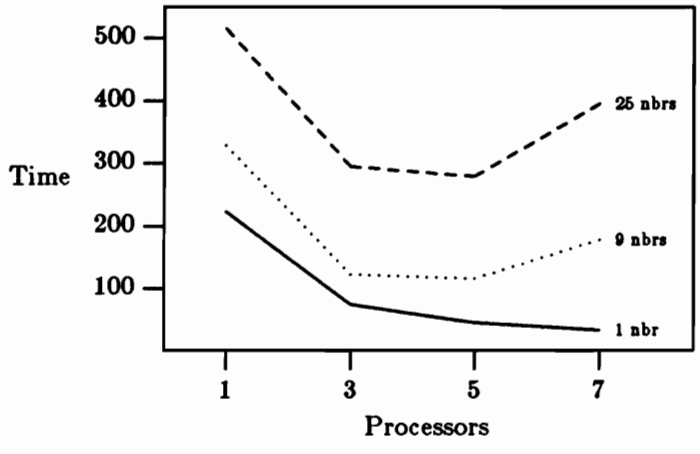

Figure 4.2. Busy Wait with Early Synchronization

can be partially attributed to the overhead incurred by blocked nodes, which must busy wait through part of the computation cycle of a neighbor. Though the number of blocked nodes and the time spent in busy wait are not recorded in this strategy, an increase in the number of processors must result in a corresponding increase in the total time spent in busy wait. This increase will also occur with the increased number of neighbors, so that part of the difference between neighbors with a fixed number of processors can be attributed to busy wait.

Another factor contributing to the loss of speedup and the differences between different numbers of neighbors is the number of restricted nodes. With 1 processor, no restricts will occur since the queue is accessed sequentially. Each node is advanced in turn. With multiple processors, the state of time advance of nodes in the queue can become more mixed. This increases the likelihood of restriction. However, the number of restricts remains low until the probability of blocking gets quite high. In the sample graph shown in Figure 4.2, the percentage of scheduled nodes that are restricted is less than 1% until the case 25 neighbors and 7 processors, when 7% are restricted. Although the queue is initially random with respect to the neighbor relation, all nodes have an equal state of time advancement. A lack of blocking results in synchronous queue access by the multiple processes — no nodes are restricted because advances are occurring in lock-step. As the probability of blocking increases, the the queue access becomes more randomized and the state of time advance of the nodes in the queue becomes less homogeneous. The total number of restricts also remains low since a node that restricts another node but is not itself restricted is guaranteed to advance because of the busy wait strategy.

Thus, a restricted node will be unrestricted when it is next scheduled.

If the delay time for checking a node is positive, any restricted nodes add this delay time to the overall execution time without advancing the state of the computation. In any case, higher numbers of restricted nodes result in more node accesses, neighbor sorts, neighbors that busy wait during a useless computation, and restricted neighbors that block non-restricted nodes from executing.

Busy wait with early synchronization is the strategy closest to the early ITSM parallelization strategies described in Chapter 2. However, this strategy uses one lock per node rather than one lock for a group of nodes. Though the increased number of locks can add to the overhead of locking neighbors and adds to the time required to sort locks, this is small compared to the effect of the increased probability of blocking caused by using one lock for a group of nodes. Thus, the busy wait with early synchronization strategy discussed here and used in comparison with the other strategies should perform significantly better than would a strategy derived directly from the early ITSM parallelizations.

4.2.2. Late Synchronization

Busy wait with late synchronization is identical to the previous strategy except that synchronization is delayed. A node is first checked to see if it can advance. If the node is not restricted, the neighbor set is locked just prior to advance and unlocked immediately afterward. If the node is restricted, no neighbor locking occurs.

The performance of busy wait with late synchronization is qualitatively similar to that of the previous strategy. Increased numbers of neighbors result in increased execution time for all sets of input parameters, and there is the same difficulty with speedup. However, late synchronization consistently improves on execution time over early synchronization. A sample graph is comparing early and late synchronization is given in Figure 4.3. Late synchronization typically results in fewer restricted nodes. In Figure 4.3 (a), the percentage of restricted nodes with 7 processors is 4% for early synchronization and less than 1% for late. In Figure 4.3 (b), the percentage of restricted nodes is 8% for early but only 5% for late. In the sample graphs, some of the decrease in execution time is due to the decrease in restricted nodes. However, differences in execution times between early and late synchronization remain significant even for cases where the number of restricted nodes in late synchronization is larger than that in early synchronization. This indicates that

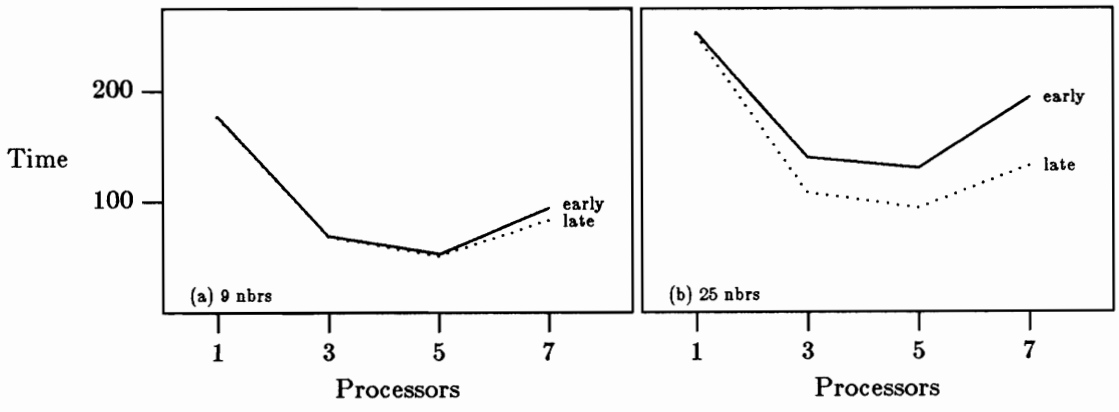

Figure 4.3. Early vs. Late: Busy Wait

there is more loss of efficiency in early synchronization from blocked nodes spin waiting on neighbors than is the case in late synchronization. This is certainly to be expected, since late synchronization reduces the size of the node computation critical section by the total time to check for restriction. Also, the "check phase" of the computation for a node can occur in parallel with the "advance phase" of its neighbor set.

4.2.3. Noise

Though all versions of the strategies are nondeterministic due to system activity and process switching, experiments were run that purposefully injected noise into the delay time parameters. A random number generator was used to modify these parameters at each node access, resulting in times that varied from 0 to the value of the parameter. This more closely resembles real ITSM problems, where the time to perform computations with a node are variable due to branch decisions in the physics routines and the variable number of neighbors (though the strategies do have some variation at the grid boundaries).

The addition of noise to busy wait with early and late synchronization tended to increase the probability of restricting, since the state of time advancement of the nodes grew even less homogeneous. This resulted in greater execution times for both the early and late strategies. See Figure 4.4 for comparisons of early and late results with the equivalent noisy versions. In general, the addition of noise increases the number of restricts, and restricted nodes make their appearance at lower blocking probabilities. In Figure 4.4 (a), the percentage of restricts for 7 processors

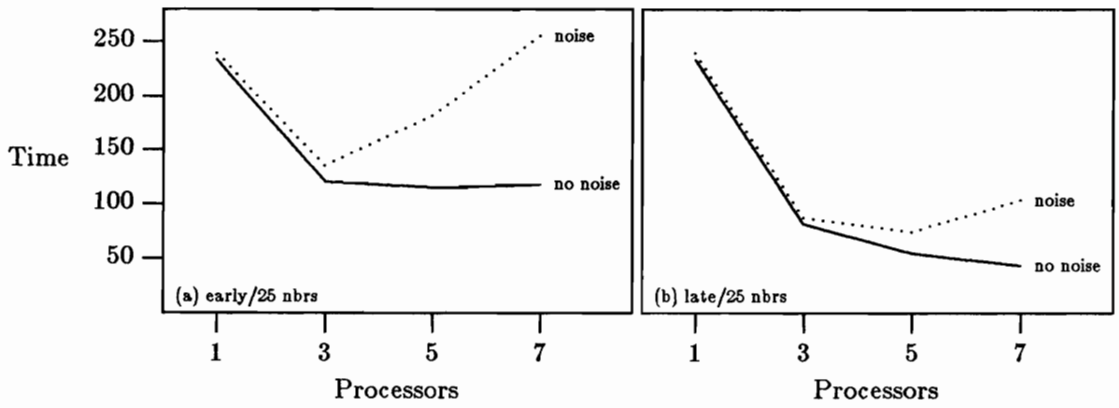

Figure 4.4. Effect of Noise on Busy Wait

jumps from 5% for no noise to 13% for noise. In Figure 4.4 (b), the percentage is less than 1% for no noise but climbs to 11% for noise. Still, the increases were not as widespread as expected. No restricted nodes appeared in the case of 1 neighbor, for any input parameter set. The reason for this is unclear, but apparently the injected noise was insufficient to randomize the queue in these circumstances.

It should be noted that the graphs in Figure 4.4 use a check/advance ratio of 3 (3:1), which is unrealistic for ITSM. The exact ratio for ITSM problems is unknown, but it *is* known to be less than 1. This example is interesting, however, in that busy wait performs better in these circumstances and even achieves speedup for high blocking probabilities in the case of late synchronization. Unfortunately, this speedup was lost with the introduction of noise, which more realistically models ITSM.

4.3. Relinquish

This section describes the performance of the relinquish strategy. Section 4.3.1 covers early synchronization, while Section 4.3.2 covers late synchronization. Section 4.3.3 discusses the addition of noise into these two strategies.

4.3.1. Early Synchronization

In the relinquish strategy with early synchronization, a process first obtains a node from the scheduling queue and then computes the neighbor set for the node. The neighbor set does not need sorting since deadlock is not possible in this strategy. Neighbor locks are acquired using the two level locking scheme described in Section 3.2.3.2. If the process finds the node flag is locked, any neighbor locks acquired up to that time are unlocked and the node is returned to the scheduling queue. In the case that the process successfully acquires all locks, the node is checked to see if it can advance. If the node is not restricted, it is advanced, the neighbor set is unlocked and the node is returned to the queue.

The graph in Figure 4.5 gives representative performance for this strategy. As in the busy wait strategies, execution time increases with the number of neighbors. However, in this case, speedup is achieved for 9 neighbors in spite of large numbers of blocked and restricted nodes. In Figure 4.5, the percentage of scheduled nodes that are blocked is already 43% for 3 processors, with another 31% restricted. Blocked nodes account for an increasing percentage of accesses as the number of processors increases. For 7 processors, 82% of accessed nodes are blocked, and

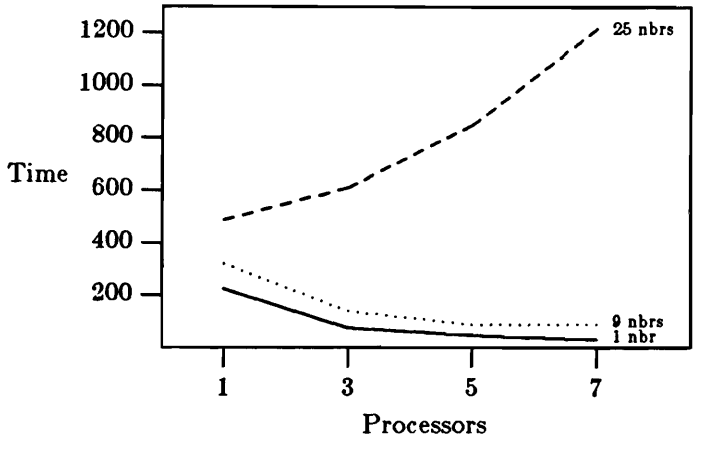

Figure 4.5. Relinquish with Early Synchronization

another 13% restricted. Twenty nodes are accessed for every one that advances. However, speedup is maintained in spite of these increases. Since the majority of nodes are blocked, most of the accessed nodes waste only the time to acquire and relinquish a subset of their neighborhood, and lock acquisition has low overhead. Restricted nodes waste lock acquisition and lock release on the full neighbor set, but this is still low overhead. The problem with restricted nodes is primarily in the computation phase. Since neighbor sets are not sorted, this is the check time alone. With a small granularity, a large number of nodes can be checked without overwhelming all benefits from increased numbers of processors.

With 25 neighbors, the number of blocked and restricted nodes increases roughly by a factor of 20 over the number for 9 neighbors, and the strategy becomes compute bound. The number of blocked nodes increases so dramatically with an increase from 9 to 25 neighbors because of the relinquish strategy. The probability of blocking is higher for 25 neighbors with 3 processors than with 9 neighbors and 7

processors. Because of the high probability of blocking, it is very likely that neighbors will interfere repeatedly with each other, so that acquisition of all neighbor locks is rare. The reason for the corresponding dramatic increase in restricted nodes is that a node that is restricted may be scheduled many times before restricting nodes advance, since the probability of blocking is so high. When a node that was restricted on its last scheduling finally acquires all locks, it is likely that the nodes that restricted it previously were blocked at last scheduling and still restrict the node.

4.3.2. Late Synchronization

Relinquish with late synchronization is identical to the previous strategy except that synchronization is delayed. A node is first checked to see if it can advance, without locking any neighbors. If the node is not restricted, the neighbor set is locked just prior to advance (with relinquish in case of block) and unlocked immediately afterward. If the node is restricted, no neighbor locking occurs and the node is returned to the scheduling queue. This strategy was previously reported in [BaSE88].

As before, the execution time increase with the number of neighbors for a fixed number of processors. And, as in busy wait, late synchronization outperforms early synchronization. Sample graphs are given in Figure 4.6.

Relinquish with late synchronization achieves speedup even for 25 neighbors, where the probability of blocking is high. For 9 neighbors, early and late synchronization have approximately equal execution times (recall that the performance of

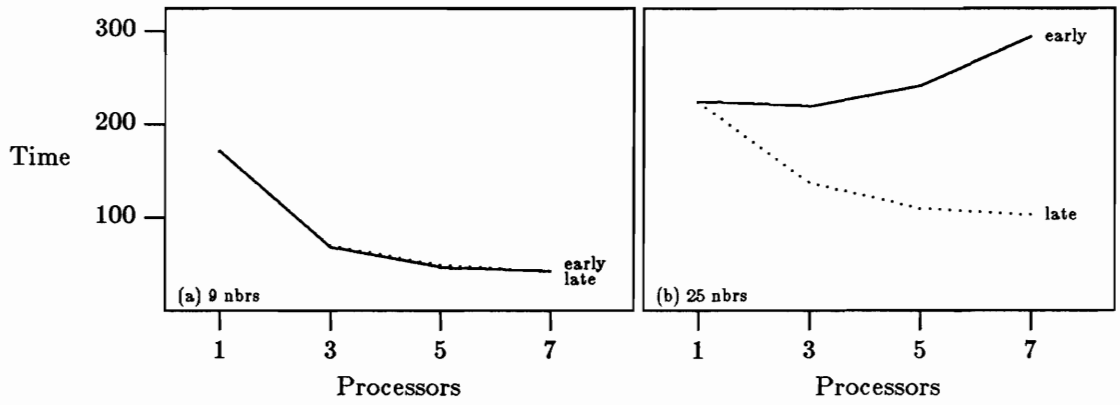

Figure 4.6. Early vs. Late: Relinquish

early synchronization was also good in this case). This is true even though the total number of nodes accesses in early synchronization is over twice that of late synchronization. This is because blocked and restricted nodes break down differently. With late synchronization, there are far fewer blocked nodes because the probability of blocking is actually lower due to the late synchronization. In Figure 4.6 (a), the percentage of scheduled nodes that are blocked decreases from 73% for early synchronization to 28% for late (with 7 processors). A node only blocks when a neighbor is advancing. On the other hand, late synchronization has an slightly increased number of restricted nodes. The reason for a higher number of restricted nodes is unknown, and may be a minor statistical variance due to the nondeterministic nature of the model. Since the overhead for a blocked node is much lower than for a restricted node, the small difference in restricted nodes is equivalent to a much larger difference in blocked nodes with this granularity of computation.

With 25 neighbors, late synchronization provides a major improvement over early synchronization. Early synchronization accesses an order of magnitude more nodes (with roughly twice as many restricted nodes). The probability of blocking is lower with late synchronization because of the delayed synchronization and resulting increase in parallelism, as discussed earlier. The difference in restricted nodes is more significant, and is probably lower with late synchronization as a direct consequence of the lower probability of blocking. It is far more likely that a node that was restricted when last scheduled will not be restricted when next scheduled.

4.3.3. Noise

The addition of noise in the check and advance times made no difference to the relinquish strategy with either early or late synchronization. The occurrence of large numbers of blocks and restricts at all non-zero blocking probabilities successfully randomizes the state of time advancement of nodes in the scheduling queue. The case of multiple processes with zero blocking probability remained elusive, since the addition of noise produced no restricts.

When the model of ITSM was originally constructed and tested on the various strategies, noise in the delay times was used in all cases in order to match real ITSM problems more closely. Later, the model was extended and reorganized in preparation for further testing. Noise was turned into a parameter and several tests were run with the noise turned off. In testing the relinquish strategy, some small problems that should have had a brief execution time ceased to terminate (in reasonable time). It became apparent that there was a problem with livelock, or

deadly embrace, in this strategy. If neither injected noise nor system noise is present, it is possible for a group of processes to become synchronized in accessing a cycle of neighboring nodes. These nodes repeatedly block each other, return to the scheduling queue, and are rescheduled to block each other again. This can occur with both early and late synchronization.

Although the deadly embrace is theoretically a major problem, the probability of its occurrence can be easily reduced to near zero by introducing noise. In real ITSM problems, there are variations in every aspect of the node computation, including computation that takes place before synchronization. Without the use of a model, the problem with the relinquish strategy might never have been noticed.

System noise alone is probably sufficient to prevent the deadly embrace. In fact, the many attempts at reproducing the experiments that experienced livelock have failed. However, benchmarking applications running exclusively on some system would have more need for injected noise.

The relinquish strategy is comparable to the action taken by Ethernet [MetB76] when stations collide when attempting to transmit a packet. A station that has detected a collision abandons the attempt and tries again after a random time period. In the relinquish strategy, a processor abandons a node when there is a conflict with a neighboring node, and the node is eventually rescheduled. When noise is injected into the relinquish strategy, the time period before rescheduling is random.

4.4. Timestamp

This section describes the performance of the timestamp strategy. Section 4.4.1 covers early synchronization, while Section 4.4.2 covers late synchronization. Section 4.4.3 discusses the addition of noise into these two strategies.

4.4.1. Early Synchronization

The timestamp strategy is a compromise between the busy wait and relinquish strategies. In the timestamp strategy with early synchronization, a process first obtains a node from the scheduling queue and then computes the neighbor set for the node. The neighbor set must be sorted to prevent deadlock. A unique timestamp is then acquired for the node. Neighbor locks are acquired using the timestamp locking scheme described in Section 3.2.3.3. If the process finds a node flag locked by a node with an earlier timestamp, any neighbor locks acquired up to that time are unlocked and the node is returned to the scheduling queue. If the node flag is found locked by a node with a later timestamp, then the locking routine is again executed. If the node flag is unlocked (the process that had held the lock either acquired all locks and completed computation or was blocked and relinquished), then the node flag is locked and the node is timestamped. In the case that the process successfully acquires all locks, the node is checked to see if it can advance. If the node is not restricted, it is advanced, the neighbor set is unlocked and the node is returned to the queue.

A representative graph for the timestamp strategy with early synchronization is given in Figure 4.7. In this timestamp strategy, speedup is achieved for 9 neighbors even though the number of blocks and restricts is large. With 25 neighbors, execution time per processor holds steady in general as the number of processors increases, rather than falling and rising as in early busy wait or simply rising as in early relinquish. The number of scheduled nodes typically goes up by a factor of 5 when increasing from 9 to 25 neighbors. Also, the percentage of node accesses caused by blocked nodes rises as the number of processes increases.

With the timestamp strategy, the majority of blocked nodes will relinquish. The bulk of the remainder must busy wait through at most one neighbor computation. An insignificant number of nodes will busy wait through multiple computations, or busy wait through a computation only to relinquish to another competing neighbor with an earlier timestamp. Restricted nodes may be scheduled multiple

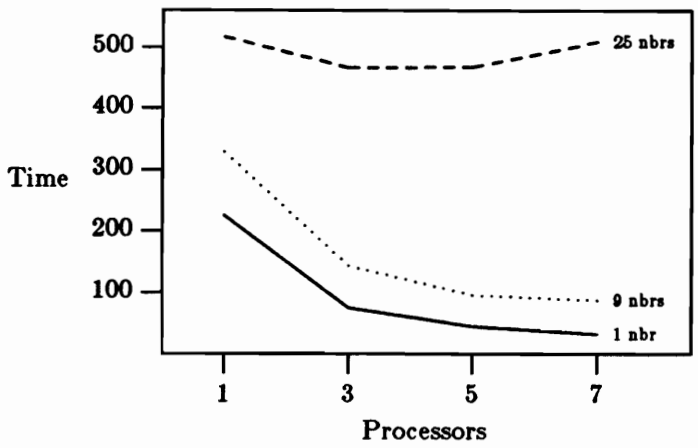

Figure 4.7. Timestamp with Early Synchronization

times before becoming unrestricted, but these numbers are somewhat controlled by the persistence of nodes with early timestamps.

4.4.2. Late Synchronization

Timestamp with late synchronization is identical to the previous strategy except that synchronization is delayed. A node is first checked to see if it can advance. If the node is not restricted, the neighbor set is locked just prior to advance and unlocked immediately afterward. If the node is restricted, no neighbor locking occurs.

Representative graphs for this strategy are given in Figure 4.8. The comparison between early and late synchronization roughly follows the pattern exhibited in the relinquish strategy. However, the number of blocks that equate to a fixed number of restricts is smaller in the timestamp strategy. The cost of a

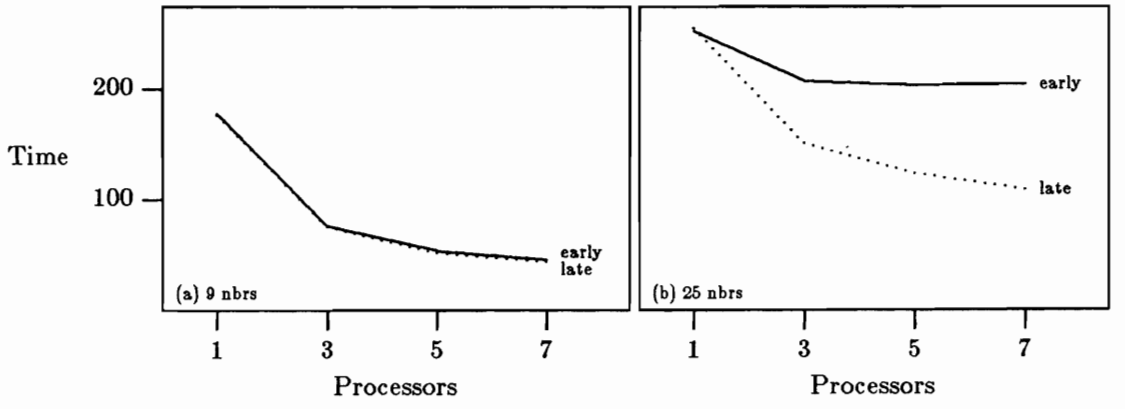

Figure 4.8. Early vs. Late: Timestamp with 25 Neighbors

restricted node is higher in timestamp, since neighbor sets must be sorted to prevent deadlock. The cost of a blocked node is also higher due to the occurrence of busy wait. The change in relative cost of blocks in terms of restricts may indicate that the effect of the busy wait is more significant. In any case, late synchronization greatly outperforms early synchronization with 25 neighbors (see Figure 4.8 (b)). The number of restricted nodes increases with early synchronization, and the number of blocked nodes goes up by an order of magnitude.

4.4.3. Noise

The timestamp strategy is similar to the relinquish strategy in that the addition of noise had no impact on execution times.

4.5. Comparison of Strategies

In the previous sections it has been observed that increasing numbers of neighbors result in higher overall execution times due to the corresponding increase in granularity and probability of blocking. Busy wait has equal speedup difficulties for 9 and 25 neighbors, early and late synchronization, but achieves some speedup before losing it with larger numbers of processors. Relinquish and timestamp achieve no speedup with 25 neighbors and early synchronization, though timestamp does not lose much in wall clock time. They both improve significantly with late synchronization and achieve speedup. Late synchronization with a given blocking strategy has better performance when the probability of blocking is high due to

increased parallelism and reduced probability of blocking. The addition of noise into the model has an effect only in the busy wait strategy, due to the higher degree of randomization of queue access.

When the various blocking strategies are compared directly, no significant difference is seen for the case of 1 neighbor (the probability of blocking is 0). The barely perceptible differences favor the relinquish strategy, where execution of two phase locking apparently has a slight edge over a call to the quicksort routine, and penalize the timestamp strategy, where a more complex locking routine is used in addition to the neighbor sort.

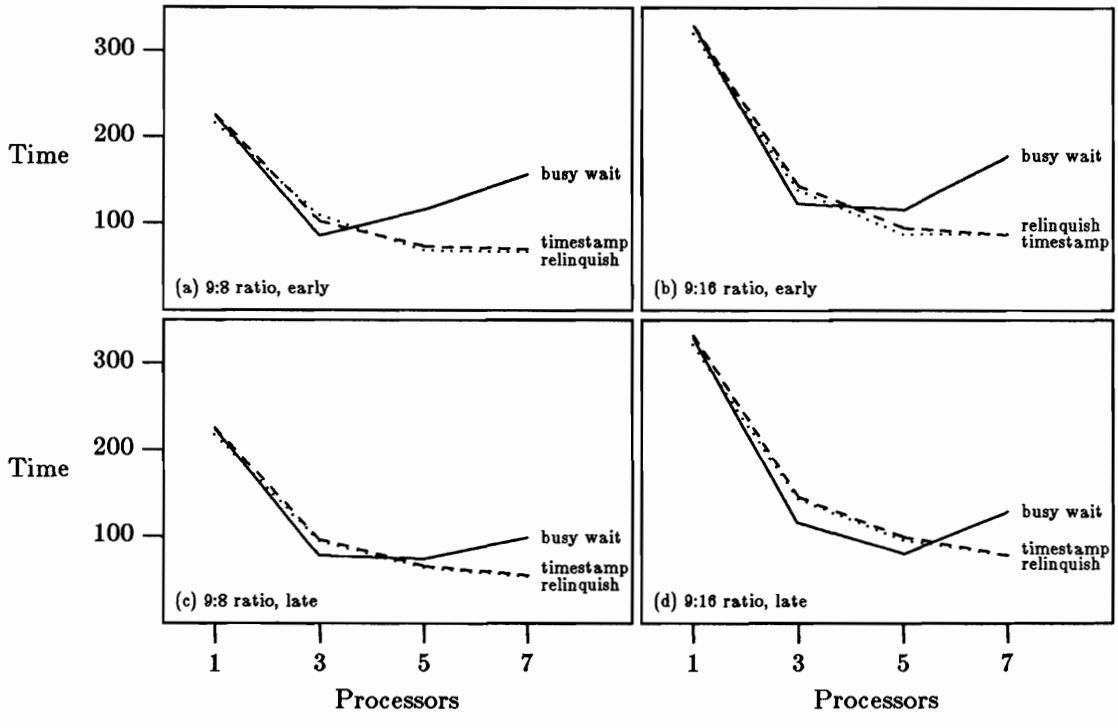

Figure 4.9. Check/Advance Ratio: 9 Neighbors

Check/Advance Ratio The performance of the strategies was tested at various ratios of check time to advance time. In general, a smaller ratio favored the relinquish and timestamp strategies. The comparison of overall performance varies somewhat according to the number of neighbors and the synchronization strategy. Figure 4.9 gives a sample of the three blocking strategies with early and late synchronization for the case of 9 neighbors. Relinquish and timestamp outperform busy wait in spite of having far more blocked and restricted nodes. This difference cannot be attributed to the neighbor sort, since timestamp also sorts neighbors. The poor performance of busy wait must be attributable to excessive busy waiting during neighbor computations.

Figure 4.10 compares the three blocking strategies with early and late synchronization for the case of 25 neighbors. With early synchronization, the massive numbers of blocked and restricted nodes, especially for the relinquish strategy, give

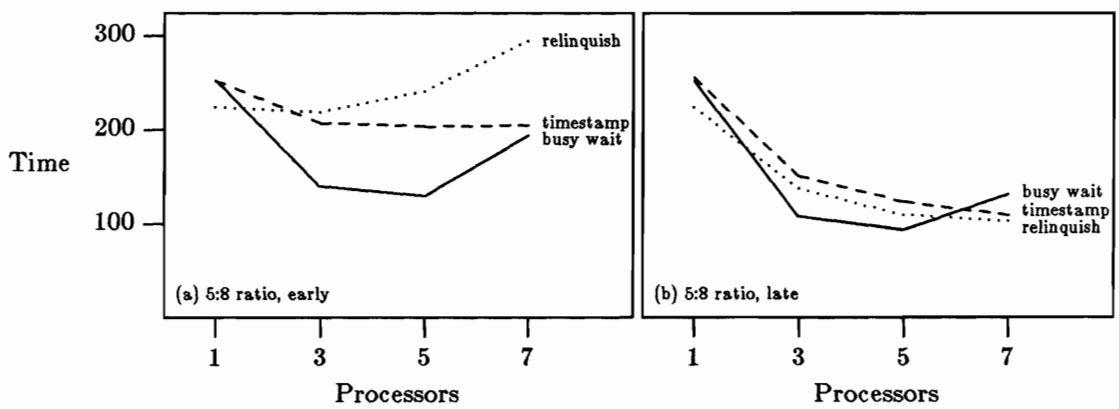

Figure 4.10. Check/Advance Ratio: 25 Neighbors

the edge to busy wait. However, as blocking increases, busy wait loses ground to the timestamp strategy.

Granularity The strategies were tested using several granularities (check time plus advance time). As the number of neighbors increases, granularity also increases proportionally. The effect of increased granularity from this source has been discussed previously. The effect of granularity increase from varying the check and advance times per neighbor had no qualitative effect on the comparative performance of the strategies and served only to uniformly increase overall execution times, with some slight exaggeration of differences in favor of timestamp and

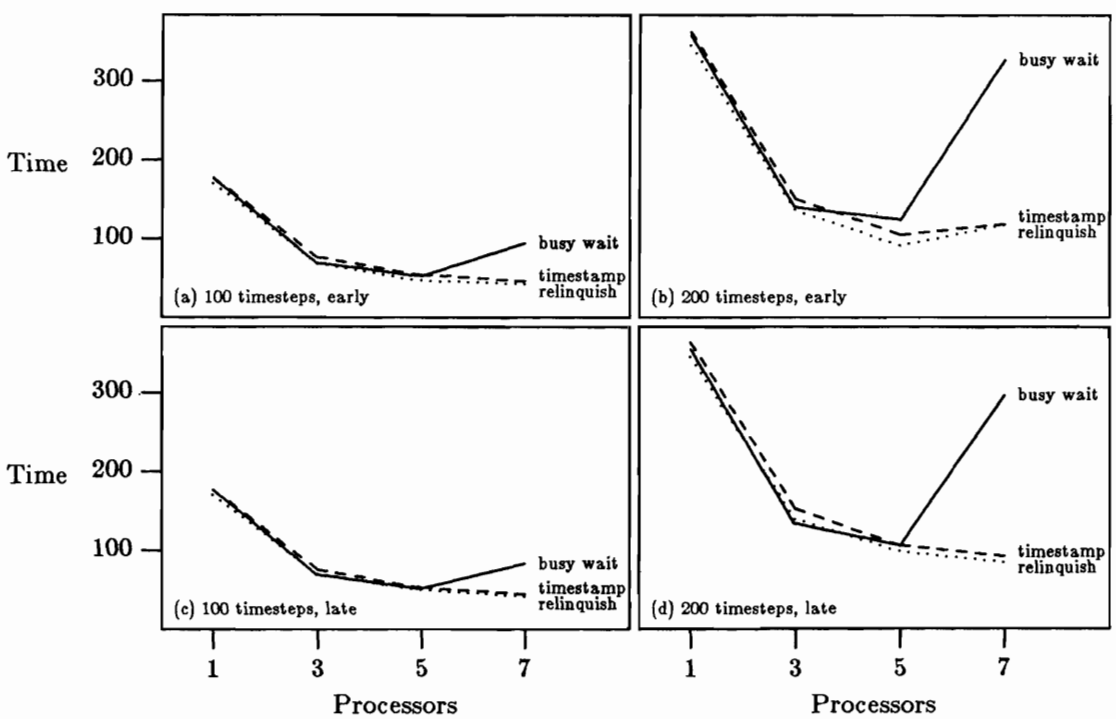

Figure 4.11. Timesteps: 9 Neighbors

relinquish. Thus, these times were not tested sufficiently to discover qualitative changes in behavior for a fixed number of neighbors.

Timesteps An increase in the number of timesteps taken per node also exaggerated differences in favor of the relinquish and timestamp strategies. Figure 4.11 shows the effect of increased timesteps for early and late synchronization in the case of 9 neighbors. Busy wait shows a dramatic increase in execution time. Again, this must be due to the accumulated penalty of busy waiting during neighbor computations. Figure 4.12 gives the corresponding comparisons for the case of 25 neighbors. In early synchronization, the performance of relinquish deteriorates rapidly.

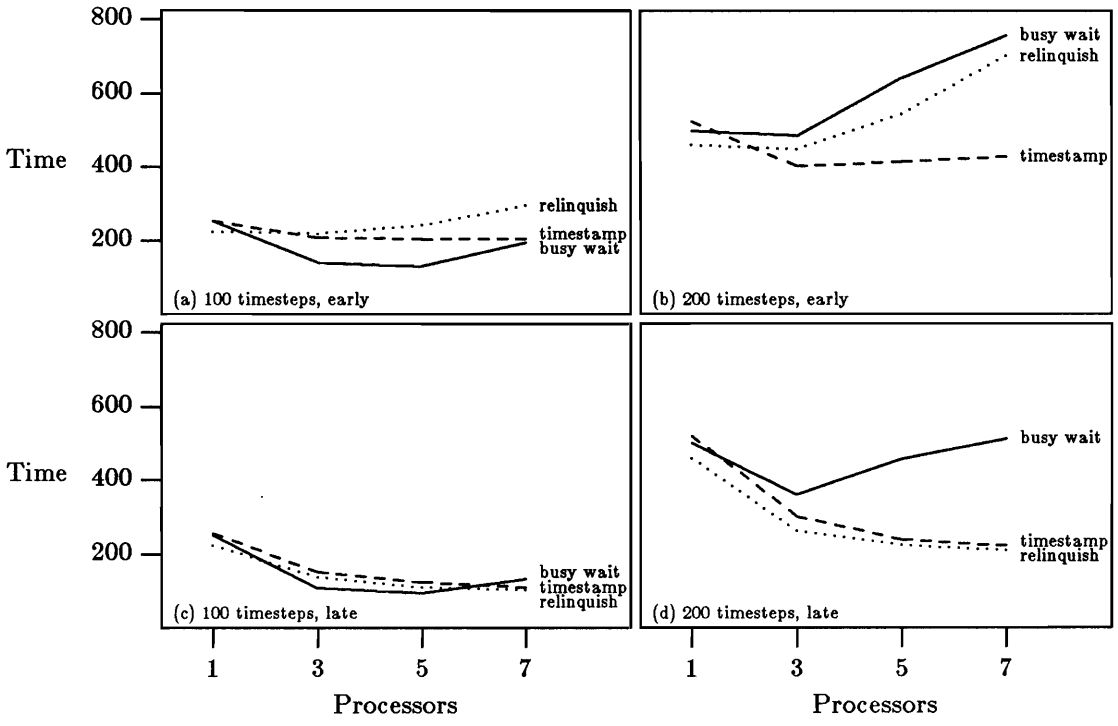

Figure 4.12. Timesteps: 25 Neighbors

However, busy wait loses even more ground due to the accumulated effect of blocked nodes. With late synchronization, relinquish and timestamp hold ground, while busy wait again loses ground more rapidly.

Summary The relinquish and timestamp strategies show better speedup potential and overall performance than busy wait for a wide range of input parameter sets, especially for parameter choices that are more probable in real ITSM problems. In the cases where relinquish suffers the most problems, the timestamp strategy manages to hold ground over busy wait, and would be a good choice of strategies in those cases.

4.6. Full ITSM Code

In real ITSM problems, the probability of blocking and restricting is unknown. However, since timestamps are not constant, allowing nodes away from the "action" to advance less frequently, most scheduled nodes are in "hot" regions where timesteps are frequent and small. Thus, the assumption can be made that real ITSM problems are subject to a relatively high probability of blocking and are likely to be restricted by neighbors.

The assumptions are supported by the performance of a variant of the relinquish strategy with early synchronization that was implemented on the full ITSM code, which achieved linear speedup on a 12 processor Sequent. A process obtained a node from the scheduler and identified the neighbor set. Using the two level locking scheme, the process then scanned the node flags *without locking the node locks*, and if any were set to `locked`, the node was returned to the scheduler. If all

neighbors were found to be unlocked, the process then executed a busy wait on each neighbor lock, by repeatedly calling two level locking until the flag was successfully locked. With this strategy, it is possible that a node in the neighbor set will be locked by another node in the period between the scan of neighbor flags and the actual locking, so that the process must busy wait throughout a computational cycle. However, the probability of this occurring will be low even when the probability of blocking is high, as long as the granularity of the computation is sufficiently larger than the overhead of locking the neighbor set.

Although the relinquish strategy with early synchronization did not do well in some of the experiments with the simplified ITSM model, the good performance of this strategy over early busy wait in full ITSM codes may be explained in several ways. The greatest difficulty with the relinquish strategy was the dramatic increase in blocked and restricted nodes due to multiple scheduling of restricted nodes when the probability of blocking was high. Since full ITSM does not return restricted nodes to the scheduler, the relinquish strategy as implemented on the ITSM model is a worst case scenario for alternatives to the busy wait strategy, for which multiple scheduling of restricted nodes was not a problem. Also, in full ITSM, the probability of blocking may be large, but not as large as the 25 neighbor case. Full ITSM is likely more analogous to the case of 9 neighbors, where the relinquish strategy did much better in comparison with busy wait. Lastly, an increased number of timesteps caused a much more rapid deterioration in the performance of busy wait than that of relinquish, so that part of the good performance of the relinquish strategy on full ITSM codes may be due to increased timesteps.

CHAPTER 5

Conclusions

Efficient parallelization of ITSM codes is a broad problem with many issues and parameters to study. The prior approaches to parallelizing ITSM were haphazard and inefficient. The early strategies that were developed were a result of working directly with a half megabyte of lightly documented C code. Nodes were grouped into boxes to reduce parallel bug problems and make lock grains explicit, and excess busy waits were used. These early strategies focused on the efficiency of allocation of work to processes but failed to exploit some of the sources of improved efficiency available with ITSM.

The use of a model for testing parallelization strategies for large and unwieldy codes is to be recommended. If statistics are available for such parameters as computation times for major code modules, degree of connectivity and likelihood of blocking, models can provide a relatively accurate simulation of the full code through the proper setting of input parameters. Strategies can be implemented more easily, with less likelihood of bugs that can interfere with obtaining results on the true performance of a given strategy. Worst case scenarios can be easily tested. More importantly, an abstract view of the problem can be taken, hopefully resulting in more widely applicable solutions. The range of applications for which a

particular strategy is most effective can be determined by testing over a wide range of input parameters. This will be useful in ITSM, where application of ITSM to different physics problems and higher dimensional spaces can drastically alter the characteristics of the code.

In this work, the model was tailored to ITSM. This allowed the application dependent information to be exploited in the search for increased efficiency. The two phase nature of a node update in ITSM allowed for the possibility of using late synchronization to increase potential parallelism and reduce the probability of blocking by shrinking the size of the main critical section. On the down side, use of problem dependent information reduces the general applicability of the parallelization strategies.

The relinquish strategy of giving up when blocked is counterintuitive because of the possibility that useful work will be done and then thrown away. However, successful parallelization strategies often result in wasted or duplicated work. This is particularly true in dynamic graph problems [ChLa82]. ITSM is in fact related to this class of problems, since the underlying space is an arbitrary dynamic graph. Perhaps other characteristics of ITSM such as the tightly coupled nature of the algorithm contribute to the success of such an approach. The relinquish strategy is not tied in any way to ITSM and may be applicable to a wide range of problems.

5.1. Future Directions

Average runtime statistics on real ITSM problems would be useful in setting input parameters for the model strategies and would help the decision as to which strategy would be most effective in the new code. The necessary statistics are dictated by the model design.

The correctness of late synchronization in ITSM codes needs to be determined. This may have to be done experimentally by implementing a late synchronization strategy on a full code. If it is determined to be incorrect, then a compromise between early and late synchronization can be studied.

The problem of reducing or eliminating the scheduling of nodes that are blocked when neighbor relationships are dynamic is difficult and could involve significant overhead. This may be worth investigating as the dimension of the problem grows. In higher dimensions, there are many more neighbors to locate and lock, though the total number of nodes also increases by an order of magnitude or more. Overall, the likelihood of contention may decrease so that excessive scheduling overhead could be avoided.

The scheduling of nodes that are time restricted could be reduced by storing such nodes separately until the restricting node updates. However, this would also add to the complexity of the work allocation procedure, and possibly result in too much scheduling overhead to justify the effort. Entirely eliminating the scheduling of restricted nodes would require too much computation in the scheduler, and should not be considered.

Enforcing fairness in an application can result in degradation of performance due to added scheduling complexity and synchronization, especially in the case of a tightly coupled and fairly fine grained dynamic graph algorithm. This work did not attempt to address this issue, since *probabilistic fairness* [Fra86] gets the job done with greater efficiency. However, fair solutions such as the Drinking Philosophers algorithm [Ch84] [Mu88] are applicable to ITSM, and could be tested and compared to the ITSM model strategies to verify or reject the intuition as regards ITSM.

The use of noise can reduce the need for synchronization in a wide range of applications. Random search algorithms for dynamic graphs reduce synchronization needs and allow the possibility of superlinear speedup [MeG85]. Ethernet [MetB76] uses random delays between schedulings of conflicting packets to avoid use of synchronization between distributed stations. In ITSM, random queue initialization removes the need for complex and dynamic queue maintenance strategies. Also, the relinquish strategy with random physics computation times removes the need for excess busy wait due to synchronizing neighbors.

References

[BaSE88]

Babb, R. G., Storc, L. and Eltgroth, P. G., "Parallelization Schemes for 2-D Hydrodynamics Codes Using the Independent Time Step Method," *Parallel Computing*, vol. 8 (1988), pp. 5-9.

[Boy87]

Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J. and Stevens, R., *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, Inc., 1987.

[Ch84]

Chandy, K. M. and Misra, J., "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, vol. 6 (1984), pp. 632-646.

[ChLa82]

Chin, F. Y., Lam, J. and Chen, I., "Efficient Parallel Algorithms for Some Graph Problems," *Communications of the ACM*, vol. 25 (1982), pp. 659-665.

[Den84a]

Denelcor HEP/UPX Reference Manual, Denelcor, Inc., August, 1984.

[Den84b]

Denelcor FORTRAN 77 Reference Manual, Denelcor, Inc., June, 1984.

[DiB89]

DiNucci, D. C. and Babb, R. G., "Design and Implementation of Parallel Programs with LGDF2," *Digest of Papers, Compcon Spring 1989*, pp. 102-107.

[Duk81]

Dukowicz, J. K., "Lagrangian Fluid Dynamics Using the Voronoi-Delaunay Mesh," Report LA-UR-81-1421, Los Alamos National Laboratory, 1981.

[Elt85]

Eltgroth, P. G., "Free Lagrangian Methods, Independent Time Steps, and Parallel Processing," in *The Free Lagrange Method*, M. J. Fritts, W. P. Crowley, and H. Trease (eds.), Springer-Verlag (1985), pp. 114-121.

[Elt86]

Eltgroth, P. G., "Physics Codes on Parallel Computers," in *Proceedings 1985 International Conference on Parallel Computing*, M. Feilmeier, G. Joubert, and U. Schendel (eds.), Elsevier Science Publishers B.V., North-Holland, Amsterdam (1986), pp. 213-220.

[EP85]

Eltgroth, P. G. and Porter, A. P., "The Independent Time Step Method for Hydrodynamics," Report UCRL-89853, Lawrence Livermore National Laboratory, 1985.

[Fra86]

Francez, N., *Fairness*, Springer-Verlag, 1986.

[Ger85]

Gelernter, D., "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7 (1985), pp. 80-112.

[Gr61]

Grandey, R. A., "Application of Finite Difference Methods to Problems in Two Dimensional Hydrodynamics," Report U-1130, Feltman Research and Engineering Laboratories, 1961.

[Jor87]

Jordan, H., "The Force", in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass (eds.), MIT Press (1987), pp. 395-436.

[Lusk83]

Lusk, E. L. and Overbeek, R. A., "An Approach to Programming Multiprocessing Algorithms on the Denelcor HEP," Report ANL-83-96, Argonne National Laboratory, 1983.

[MeG85]

Mehrotra, R. and Gehringer, E. F., "Superlinear Speedup Through Randomized Algorithms," *International Conference on Parallel Processing*, 1985, pp. 291-300.

[MetB76]

Metcalfe, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19 (1976), pp. 395-404.

[Mu88]

Murphy, S. L. and Shankar, A. U., "A Note on the Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, vol. 10 (1988), pp. 178-188.

[Ray86]

Raynal, M., *Algorithms for Mutual Exclusion*, MIT Press, 1986.

[Seq87a]

Sequent Guide to Parallel Programming, Sequent Computer Systems, Inc., 1987.

[Seq87b]

Sequent Pdbx User's Manual, Sequent Computer Systems, Inc., 1987.

[Seq87c]

Symmetry Technical Summary, Sequent Computer Systems, Inc., 1987.

[Sto88]

Storc, L., "Sequent Balance Series," in *Programming Parallel Processors*, Robert G. Babb II (ed.), Addison-Wesley (1988), pp. 143-154 and 317-377.

[ToMa]

Toffoli, T. and Margolus, N., *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, 1987.

Biographical Note

The author was born in Austin, Texas and graduated from Lockhart High School in 1974. She graduated from the University of Texas at Austin with a B.S. in Mathematics in 1977. She then did graduate work in mathematics at the University of North Carolina at Chapel Hill and taught courses at UNC for five years. She moved to Oregon in 1983 to attend graduate school in computer science at OGC. She has been employed full time for the past three years at Emanuel Hospital in Portland, where she runs the Medical Computing Lab.