

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M3

“Interactive Animation of Fault Tolerant Parallel Algorithms”

by
Scott W. Apgar

Interactive Animation of Fault Tolerant Parallel Algorithms *

Scott W. Apgar
Digital Equipment Corporation,
Codman Hill Road, Boxborough, MA 01719, USA
apgar@oblio.enet.dec.com,
Dept. of Computer Science, Brown University,
Providence, RI 02912, USA

January 31, 1992

Abstract

Animation of algorithms makes understanding them intuitively easier. This paper describes the software tool *Raft* (Robust Animator of Fault Tolerant Algorithms). The *Raft* system allows the user to animate a number of parallel algorithms which achieve fault tolerant execution. In particular, we use it to illustrate the key *Write-All* problem. It has an extensive user-interface which allows a choice of the number of processors, the number of elements in the *Write-All* array, and the adversary to control the processor failures. The novelty of the system is that the interface allows the user to create new on-line adversaries as the algorithm executes.

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science
in the Department of Computer Science at Brown University

*This research was supported in part by ONR grant N00014-91-J-1613, and Digital Equipment Corporation.

This thesis by Scott Wade Apgar
is accepted in its present form by the Department of
Computer Science as partial fulfillment of
the requirements for the Degree of Masters of Science

Date: Jan 31 92 Advisor: 
Paris C. Kanellakis

1 Introduction

Algorithm visualization is a powerful tool in helping the designer gain insight into the execution of an algorithm. It provides a whole new dimension to the study of algorithms, exposing problems and properties of the algorithm which may not be discovered without the animation. Animation techniques allow the display of changing data-structures and the execution flow of an algorithm, exposing bottlenecks as well as potential enhancements.

In this paper we stress that allowing interaction with the animation makes it possible to gain a clearer understanding of exactly how the algorithm is working, as well as the effect particular parameters have on the algorithm. The ability to change certain algorithmic parameters interactively, along with the visualization of these changes, provides immediate feedback to the designer. Parallel fault tolerant algorithms provide a rich context for illustrating the capability to change parameters based on the information being portrayed on the screen.

The main contribution of this work is the ability to interact with the animation as the algorithm is executing, changing run-time characteristics, in particular the adversary used to fail processors.

Algorithm animation is a relatively new area of system development. [Brow87] describes the animation tool *Balsa*, considered to be the pioneering effort in algorithm animation. Many of the animation tools available today follow the general design guidelines established by *Balsa*. In short, these guidelines suggest annotating the algorithm code with *interesting events* in order to identify the possible operations which would be interesting to view. The tools typically provide a high level graphical language composed of routine calls which remove the user from the intricacies of the actual graphics code. These routines are utilized to provide the animation actions designated for each *interesting event*. Of course, the systems vary greatly in the amount of support offered the user. *Balsa* has been successfully used to visually demonstrate data structures and algorithm designs in an educational context. It has no interaction capability during the execution of the algorithm, but provides facilities for prespecifying animation scenarios. The animation tool *Zeus* is interesting for its use of objects, strong-typing, parallelism, color and sound. [BH91] describes *Zeus* in detail. This MIMD multiprocessor simulator *Proteus* [BCDW91] provides integrated graphical output through the use of a trace file and an X-based graph program. It provides line graphs, bar graphs, and tables, however, it does not have the ability to perform animation of the algorithm. [Lin91] outlines a methodology for automatic insertion of animation code into an algorithm, (annotating the *interesting events*) removing the designer from manually analyzing the algorithm for insertion points.

Tango [Stas89][SH90] is the animation tool used by *Raft*. *Tango* follows the *Balsa* design guide-

line, providing the user with a high level interface to a graphics package. Routine calls are added to the algorithm at user-specified places requesting the animation window be updated by user written routines which make use of the *Tango* facilities. *Tango* provides an I/O function which allows basic interaction with the animation window via the mouse. The user clicks on a point in the window, and the location is returned to the user-provided animation routines. *Tango* was chosen for its modularity and simplicity of use. Also, *Tango* was developed at Brown, thus is closely tied to the software development environment here. The message passing facility, *MSG*, was also developed here at Brown, and *Tango* was designed to adhere to the interface *MSG* specified.

Now let us describe the class of algorithms animated using this tool. *Efficient* parallel algorithms exist for many problems (integer manipulation, manipulating lists and trees, etc). By *efficient* we mean that the algorithms can solve those problems with near linear speed-up. The efficiency of these algorithms requires them to be very carefully designed to make the most of the parallel architecture. This leaves very little computation time for use towards making the algorithms fault tolerant (ie performing error detection and load rescheduling). The algorithms provided for animation by *Raft* have been designed to provide *robust efficient* [KS89] execution of the original parallel algorithms. In short, this means that the original parallel-time \times processors product, N , of the original algorithm, will be increased by at most a *polylog* factor of N , despite the presence of failures (see Section 2).

Raft allows the algorithm designer to easily interact with the algorithm, receiving immediate visual and textual feedback to any perturbations. This is accomplished through a user interface which controls execution of the algorithm in a simulator-like keyboard environment, and the annotation of the algorithm with high-level calls to the animation tool providing the visual feedback.

Allowing interaction with the adversaries, which control the insertion of faults, makes it possible to visualize the performance of the algorithm under user-designed fault models. The interaction makes it possible to enact “what-if” scenarios, which has the benefit of exposing performance bottlenecks in the algorithm. The interaction via the adversaries also facilitates the verification of any unproven performance bounds.

This introduction is complemented by a short example of an animation using *Raft*. Figure 1, page 4, shows a sample run of *Raft* including a subset of the animation displays and the keyboard interface. The example shows an animation of the X algorithm (see Section 2.1.2) running under the control of an interactive user-defined adversary (see Section 4.4), with $P = 64$, $N = 64$, and no restarts allowed. Note that some of the informational output has been removed from the script and ellipses inserted in the interest of brevity. User responses are in bold-face.

The remainder of the paper is organized as follows, the model of computation used by the algorithms is described in Section 2. Section 3 describes the main contribution of this work, the user interface portion of *Raft*. Section 4 gives detailed descriptions of each of the adversaries offered by *Raft*. The implementation details are discussed in Section 5, and we end with a discussion of how to add algorithms or adversaries to this tool in Section 6.

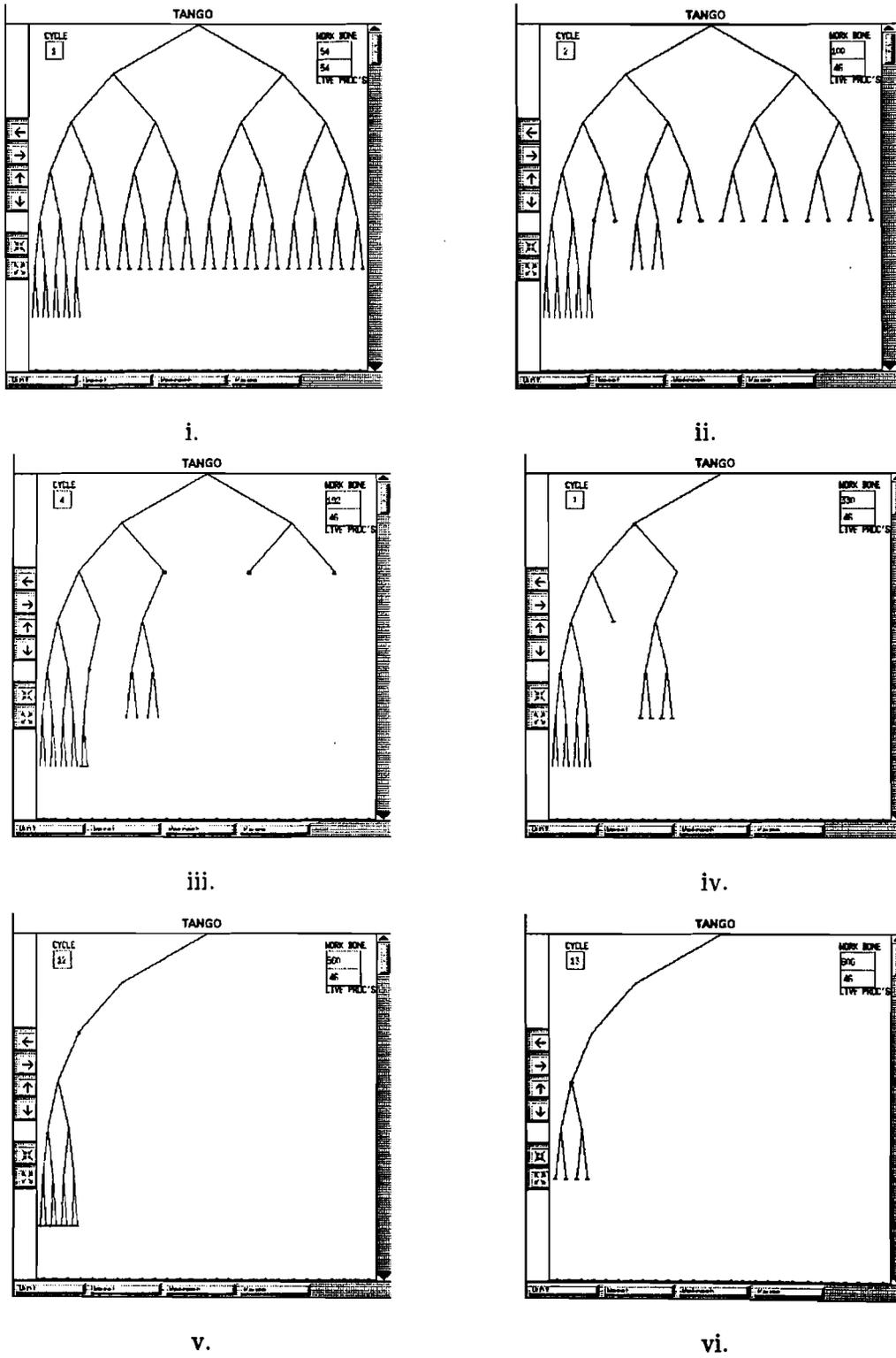


Figure 1: Example animation displays of the X algorithm, $P = 64$, $N = 64$, and an interactive adversary

2 Model of Computation

We use as a basis the parallel random access machine or PRAM model [FW78], with the assumption that all concurrently writing processors write the same value (COMMON CRCW). The PRAM is a model that has attracted much research and many efficient algorithms have been designed for it; see [KR88]. The PRAM is a convenient abstraction that combines the power of parallelism with the simplicity of a RAM, but it has some unrealistic features, e.g., broad bandwidth memory access, and freedom from faults. The gap between the PRAM and realizable parallel computers is being bridged by current research. For example, memory access simulation in other architectures is addressed in [AHMP87, DKM+88, HB88, Herl89, Herl90, HP89, KR88, KU88, Kuck77, LPP88, LPP89, Meye86, PSW91, Ran87, Upfa84, UW87]. The processors used in this model are subject to stop-failures and restarts as in [SS 83]. There are P processors, with each processor being assigned a unique identifier (PID) in the range $0, \dots, P-1$. The memory model used is one of **shared** memory, thus it is accessible to all processors. Each processor also has a constant size **private** memory. The input to the algorithm being modelled is stored in N cells in shared memory, with the rest of memory being cleared.

To guarantee algorithm termination and sensible accounting of resources, we introduce an *update cycle*, that generalizes the standard PRAM read/compute/write cycle. Each cycle consists of reading a small fixed number of shared memory cells, performing some fixed time computation, and writing a small number of shared memory cells. The parameters of the update cycle, i.e., the number of reads and writes, are fixed, but depend on the instruction set of the PRAM.

We assume that the shared memory writes of $O(\log \max\{N, P\})$ bit words are atomic. Algorithms using this assumption can be easily converted to use only single bit atomic writes as in [KS89]. We use the *fail-stop with restart* model, where time instances are the PRAM clock-ticks:

The following assumptions are made about the fault model employed in this computational model:

1. A failure pattern F (i.e., failures and restarts) is determined by an *on-line adversary*, that knows everything about the algorithm and is unknown to the algorithm.
2. Any processor may fail at any time in any update cycle, and it may later restart at any time, provided:
 - (i) at any time at least one processor is executing an update cycle that successfully completes;
 - (ii) single bit writes are *atomic*, i.e., failures can occur before or after a write of a single bit.
3. Failures do not affect the shared memory, but the failed processors lose their private memory. Processors are restarted at their initial state with their PID as their only knowledge.

Condition 2 (i) is necessary to ensure that termination of the algorithm is possible. Update cycles are thus sufficient to enable termination. In addition, they serve as units of accounting. They do not constrain the instruction set of the PRAM, however (see Def. 2.2 below), the processors are not charged for the instructions of the update cycles that are not completed.

We now present some definitions which are a necessary precursor to our definition of the key measure of the *robust efficiency* of the algorithms, completed work, given in Definition 2.2 .

Definition 2.1 is a formal definition of the adversaries, or failure patterns, used.

Definition 2.1 A *failure pattern* F is a set of triples $\langle tag, PID, t \rangle$ where *tag* is either **failure** indicating processor failure, or **restart** indicating a restart, *PID* is the processor identifier, and *t* is the time when the processor either stops or restarts. The *size* of F is defined as the cardinality $|F|$.

□

Our measure, S , of *completed work* generalizes the *Parallel-time* \times *Processors* product and the available processor steps of [KS89]; [KS91]).

Definition 2.2 Consider an algorithm with P initial processors that terminates in parallel-time τ after completing its task on some input data I and in the presence of a failure pattern F . If $P_i(I, F) \leq P$ is the number of processors completing an update cycle at time i , and c is the time required to complete one update cycle, then we define the completed work $S(I, F, P)$ as: $S(I, F, P) = c \sum_{i=1}^{\tau} P_i(I, F)$. \square

Definition 2.3 A P -processor algorithm on any input data I of size $|I| = N$, and in the presence of any pattern F of failures and restarts of size $|F| \leq M$, uses *completed work* $S = S_{N,M,P} = \max_{I,F} \{S(I, F, P)\}$. \square

We are now ready to outline the problem solved by the algorithms presented here. After this description of the problem in general, we give specific solutions for the problem in the following sections. In [KS89] it is shown that it is possible to combine efficiency and fault-tolerance in many key PRAM algorithms in the presence of arbitrary dynamic fail-stop processor errors. The solution of a key problem can be used in transporting PRAM algorithms to architectures where processor failures are present. This problem, called *Write-All*, is:

Given a P -processor PRAM and a 0-valued array of N elements, write a 1 into all array locations.

This problem captures the computational progress that can be naturally accomplished in unit time by a PRAM (when $P = N$). In the absence of failures, it is easily solved by an $O(N)$ work parallel assignment. “Efficient” solutions (with work N times a “small” factor) in the presence of failures are non-obvious.

Given an efficient solution for the *Write-All* problem, it is possible to efficiently simulate any N -processor synchronous PRAM on P restartable fail-stop processors ($P \leq N$). The simulations of the individual PRAM steps are based on replacing the trivial array assignments in a *Write-All* solution with the appropriate components of the PRAM steps. For the details on this technique, the reader is referred to [KS89, KPS90].

2.1 Algorithms for the *Write-All* problem

In the presentation of the algorithms we assume that the number of array elements N and the number of processors P are powers of 2 (nonpowers of 2 can be handled using padding) and all logarithms are base 2. The algorithms involve traversals of data structures by processors. For simplicity we give high level descriptions. It is easy to implement these algorithms using update cycles, so that when a processor fails and then restarts, it resumes the traversal of the data structures at the point where it failed.

2.1.1 Algorithms V , W : global allocation paradigm

Algorithm W , implemented in *Raft*, of [KS89] is a fail-stop (no restart) *Write-All* solution. It uses two full binary trees and it consists of a loop in which the active processors synchronously iterate through the following phases: W1: enumerate the processors in a bottom-up traversal of the processor tree, W2: allocate the processors in a divide-and-conquer top-down traversal of the progress tree, W3: work at the leaves, and W4: evaluate progress in a bottom-up traversal of the progress tree. To avoid a complete restatement, the reader is urged to refer to [KS89].

Algorithm W has *efficient* work subject to failures without restarts. It can be extended to handle restarts, but, it may not terminate if no processors complete a full iteration of all four phases. In addition, restarts invalidate the proof framework of [KS89]: the processor allocation W1 becomes incorrect, since restarts prevent accurate estimates of active processors. However, as shown in [KS91], processors can be assigned in $O(\log N)$ time by using the processor PID in the top-down divide-and-conquer allocation. [KS91a] presents a modified version of algorithm W , called V , that has a simpler allocation strategy without the enumeration phase.

```

01 forall processors PID=0..P - 1 parbegin
02   Perform initial processor assignment to the leaves of the progress tree
03   Traverse the progress tree bottom-up and update it to evaluate progress
04   while there is still work left in the tree do
05     W1: Enumerate processors in a bottom up traversal of processor tree
06     W2 (V1): Allocate processors using enumerated ids (PIDs) in a top-down traversal of
progress tree
07     W3 (V2): Perform work at the leaves reached in phase W2 (V1)
08     W4 (V3): Continue from the leaves of the progress tree and update it bottom up
08   od
09 parent

```

Figure 2: A high level view of algorithms W and V – global allocation paradigm.

```

01 forall processors PID=0..P - 1 parbegin
02   Perform initial processor assignment to the leaves of the progress tree
03   while there is still work left in the tree do
04     if current subtree is done then move one level up
05     elseif this is a leaf then perform the work at the leaf
06     elseif this is an interior tree node then
07       if both subtrees are done then update the tree node
08       elseif only one is done then go to the one that is not done
09       else move to the left/right subtree according to PID bit values fi fi
10   od
11 parent

```

Figure 3: A high level view of the algorithm X – local allocation paradigm.

[KS89] outlines a further optimization of W , which is implemented in *Raft*, algorithm W_{opt} : W_{opt} uses full binary trees of $\frac{N}{\log N}$ leaves with $\log N$ array elements mapped to each leaf. When using $P > \frac{N}{\log N}$ processors, it is sufficient for each processor to take its PID modulo $\frac{N}{\log N}$ to assure uniform initial assignment of processors. Choosing $P \leq N/\log^2 N$ is shown to yield optimal results, ie $S = O(N)$, in [KS89]. In order to add the restart capability to this algorithm there must be some form of processor synchronization. This can be realized through the use of an iteration wrap-around counter based on the PRAM clock. For an outline, see Figure 2.

Theorem 2.1 The completed work of algorithm V using $P \leq N$ processors subject to an arbitrary failure and restart pattern F of size M is: $S = O(N + P \log^2 N + M \log N)$. The completed work of algorithm W using $P \leq N$ processors subject to an arbitrary failure pattern without restarts is $S = O(N + P \frac{\log^2 N}{\log \log N})$. The completed work of algorithm W_{opt} , using $P \leq N/\log^2 N$ is $O(N)$.

2.1.2 Algorithm X : local allocation paradigm

Algorithm X utilizes a progress tree of N leaves that is traversed by the processors independently, not in synchronized phases. This illustrates the local processor assignment paradigm. Each processor searches for work in the smallest immediate subtree which needs work done.

The algorithm consists of initialization and a *loop* whose body is implemented as an update cycle (see Figure 3).

The *loop* (lines 03-10) consists of a multi-way decision (lines 04-09). If the current node is marked done (has value 1), the processor moves up (line 04). If the processor is at a leaf, it performs work (line 05). If the current node is an unmarked interior node and both of its subtrees are done, the interior node is marked done (line 07). If a single subtree is not done, the processor moves there (line 08).

The last case (line 09) is where a non-trivial decision is made(see *italics* in line 09). At depth h of the tree, the value of the h^{th} most significant bit of the processor PID is used to choose the direction: 0 sends the processor to the left, and 1 to the right. This is where the distinction is made between the 2 X algorithms offered for animation by *Raft*. Alg. X traverses to the leaf nodes visiting every node in between, in a “smooth” fashion. Alg. X_j jumps directly from the intermediate node to the leaf node when it can determine which leaf node needs work done.

Theorem 2.2 Algorithm X with $P \leq N$ processors is a correct and fault-tolerant solution for the *Write-All* problem of size N that solves the problem using completed work $S = O(N \cdot P^{\log \frac{3}{2}})$. There is an adversary that forces algorithm X to have $S = \Omega(N \cdot P^{\log \frac{3}{2}})$. (see Section 4.10)

2.1.3 *Raft* Algorithms

Raft currently implements only a subset of the known *Write-All* solutions. Future work would include adding more of the known algorithms to *Raft*. *Raft* offers the following algorithms:

1. Algorithm X
2. Algorithm X_j , a slight modification of X
3. Algorithm W
4. Algorithm W_{opt} , an optimal version of W

In general, tight upper bounds are still to be derived for both the no-restart and the restartable setting. [KS91a] *conjectures* that the fail-stop (no restart) performance of X is $S = O(N \log N \log \log N)$ using N processors. One of the reasons the *Raft* tool was created was to provide a flexible, intuitive aid for verifying the *conjectured* bounds. While *Raft* does not provide proof of these bounds, if the bound can not be beaten, while using *Raft* to change the adversaries to be the “worst” imaginable, the *conjectured* bounds seem to stand up. Note, the *conjectured* bounds have not yet been disproven through the extensive use of the *Raft* system.

3 User Interface

The user interface portion of *Raft* is the main contribution of the system. The interaction with the adversaries and algorithms adds tremendous value to the animation. It allows users to dynamically adjust the adversary to affect the algorithm in the desired ways. There are a number of options available which allow the user to tailor the execution of the algorithm in many way. These options are specified below. First a description of how the algorithms are represented by the animation window is needed.

3.1 Animation Window

This section gives an overview of the use of the animation window and how the work done by the algorithms is represented.

The tool used to perform the animation is *Tango*, developed at Brown by John Stasko, see [Stas89]. The goal of the *Tango* tool was to make it possible for users to create animations without understanding all the underlying graphics code necessary to control them. By making use of the provided *Tango* routines it is possible to create simple animations without getting deep into the details of the graphics code.

The animation window (see Figure 4) runs in its own process, separate from *Raft*. It need not be restarted after each pass of *Raft*, as there is a reset button that allows the user to clear all the

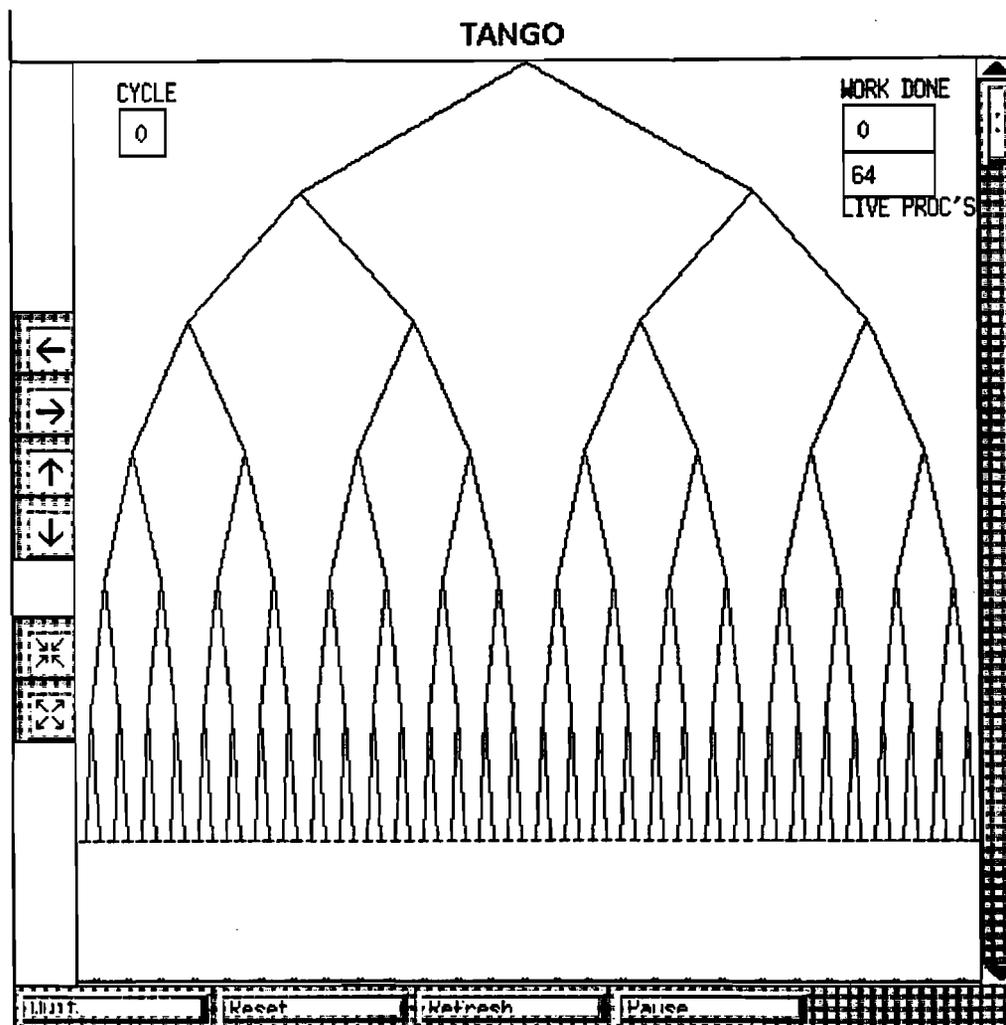


Figure 4: Sample *Tango* animation window, 64 processors

necessary *Tango* data structures. After each execution of *Raft*, use the mouse to click on the reset button in the *Tango* window, and *Raft* can be run again.

The control structure of the algorithms being animated is a binary tree. This is what is displayed in the animation window. The leaf nodes of the tree represent one of the tasks to be performed. In general, when all the work of a subtree is determined to be complete, the edge between the root of the subtree and its parent is deleted. In this way, the user can visualize the progress of the algorithm.

The processors are represented as small horizontal red lines residing at a node of the tree. Initially each processor is assigned to a leaf node, modulo the number of leaves. That is if there are 16 leaf nodes and 17 processors then leaf node 0 will have 2 processors assigned. When processors are killed by an adversary the red line representing them is deleted; when they are restarted at a node a red line is created. As processors move up and down the control tree, potentially being assigned to the same node as other processors from the subtree, the thickness of the red lines is increased (by a factor of the log of the number of processors present at the node) to show the presence of more than one processor.

The number of steps the algorithm has taken to date is written in the CYCLE box in the top left corner of the animation window before each step is animated. The amount of work done to date, S (see Definition 2.2), is written in the WORK DONE box in the top right hand corner of the animation window. Note that this is different from the number of cycles. An algorithm may have only one processor alive, and take N cycles to perform the algorithm, where an algorithm with N processors alive may take only one cycle to perform the algorithm. The amount of work done in each case is the same. The number of processors currently alive is written in the LIVE PROC's box just below the WORK DONE box.

After the simulation has completed, the user has the option of tracing the path of any processor through the binary tree. The path is highlighted using colored arrows in the animation window, and is also printed in the *Raft* window. This feature allows the user to get an idea of the path each processor takes through the tree during the execution of the algorithm. This is useful as an aid to understanding how the algorithm works, and to give a better understanding of which nodes would be likely candidates for faults.

The user can control the animation window in a variety of ways using the buttons *Tango* provides. There are ZOOM IN, ZOOM OUT, PAN LEFT, PAN RIGHT, PAN UP, and PAN DOWN buttons to control the view the user gets of the animation window.

The first query *Raft* makes of the user is whether or not to include the *Tango* animation window in the execution of the algorithm. It may seem obvious that the user would want to see the animation, but there are cases where it is advantageous to run the simulation without the *Tango* animation.

Due to the fact that the screen to be used for animation is of a fixed size, one can fit only so much information on it. This fact limits the *Tango* window to be able to use only as much information as can be stored in a screen. This software was developed on SUN Sparcstation1 GX machines. The screen on this machine limited the animation to using $P = 1024$ processors. Once this number was used, the leaves of the binary tree have lost all resolution and appear as a thick black band across the bottom of the *Tango* animation window. The *Tango* software allows the user to zoom in and out on the window, thus it is possible to see what is happening at a local level, but when viewing the entire tree, the lowest level appears to be solid black.

Also, due to the message passing facility [Reis89] used, and the design of *Tango*, each change to the animation screen requires that a message be sent to the animation window (see Section 5). With large numbers of processors the number of messages that must be passed, and the memory required to store all the animation actions slow the execution of *Raft* dramatically.

Thus if the user wants to simulate an algorithm using a larger number of processors ($P > 1024$) it

is possible to run without the *Tango* animation window. The user can still gather useful information on the runtime statistics of the algorithm, but will not have the intuitive aid of the animation screen. In single step mode the user would still have the capability to query for the same information available when the animation window is running. See Section 3.5 for a detailed description of the information available via query.

3.2 Processor/Array Element Enumeration

The user is able to specify the number of processors and the number of elements in the *Write-All* array for each execution of an algorithm. These numbers can range from 1 to 2^{18} . If the user inputs a number larger than 2^{18} the software loops asking for a smaller number. Note that all the data structures are created from heap memory according to the size of these parameters. It is suggested that the number of array elements given be a power of 2 to make the binary tree data structure complete. Each processor is initially assigned to a leaf of the binary tree, starting from the left and going right. In the normal case, the number of processors would be equal to the number of array elements.

If the user chooses to use more processors than array elements, the additional processors are assigned the the array elements modulo N . One could also choose to use fewer processors than array elements. This causes some of the leaf nodes to have no processors assigned in the first step. The assignment starts at leaf node 0 (on the left of the tree) and continues until the processors are depleted.

In the optimized version of algorithm W , the number of processors is calculated based on the number of array elements and the user specified number of processors is ignored.

3.3 Algorithm Selection

The next choice to be made is which *Write-All* algorithm will be used. There is currently a menu of 4 choices available. The user simply needs to enter the number of the algorithm to be simulated. The choices (1-4) are as follows: X , X_j , W , W_{opt} . See Section 2 for a description of the algorithms.

This choice can only be specified once per each execution of *Raft*. The algorithms share some of the same data structures, thus it is not possible to switch between algorithms during the execution. See Section 6.2 for details on adding a new algorithm.

3.4 Adversary Selection

The user now has the chance to select which adversary will interact with the simulation. The adversaries are listed in section 4. If the user chooses to run continuously, this is the only chance to change adversaries. If the simulation is run in single step mode, then the user can choose to change the adversary each step (see Section 3.5). Due to the model being used (*fail-stop restartable PRAM*), the adversaries will be used to fail as well as restart processors. The tool also offers the capability of running without restarts in the case that the algorithm requires the *fail-stop PRAM* model (see Section 2.1.1). To differentiate between these two models the user also is queried regarding whether the *revive* flag should be set or not. This flag controls whether processors are allowed to restart once they have been failed. The Lower Bound adversary (see Section 4.10) will never attempt to restart processors, but all the other adversaries will restart processors if this flag is set.

3.5 Interactive Mode

The simulation can be run in one of two modes, *single step*, or *continuous*. The *continuous* mode does not allow the user to interact with the simulation except through the adversary. If running *Raft* with an interactive adversary (see Section 4) each step the adversary will prompt the user, otherwise no interaction is needed. An example of this is shown in Section 4.7.

In *single step* mode the user has the ability to interact with the simulation between each PRAM step taken. *Single step* mode allows the user to change adversaries, quit the simulation, enter *continuous* mode if all interaction is complete, and query about the state of the simulation (work done, processors alive, details on specific nodes of the tree). This portion of *Raft* makes it extremely effective as an intuitive aid to understanding the action of the animated algorithms.

The options made available to the user in *single step* mode are: (the characters in parentheses are the proper responses to the *Raft* prompt :)

- (s)ingle step - This allows the user to step the algorithm any number of times. Typing s #, where # is some integer, will step the algorithm # times. Typing s will step the algorithm once by default.
- (c)ontinuous execution - This allows the user to let the algorithm run to completion with one command. The only interaction necessary from the user is that which may be required by the adversary. There is no output given in this mode until the algorithm is complete and the schedule of which processor wrote which array element in what cycle is printed out. (This is printed out upon completion in interactive mode also)
- query - Typing (?) will give the user some choices as to what information is desired. The choices are as follows:
 1. Expand a node in the animation window. Using the mouse the user clicks on a node in the animation window. *Raft* then prints out the following information on the node:
 - node index - this is the index of the node in the tree.
 - value - this is the value the node has, representing the amount of work done in the subtree rooted at this node.
 - live processor count - The number of live processors that are assigned to this node.
 - dead processor count - The number of dead processors that are assigned to this node.
 - assigned processors - the processors assigned to this node are listed by their unique identifiers with their state (alive/dead).
 2. total number of live processors left
 3. total work done to date
 4. return to simulation prompt
- (q)uit - this stops the animation immediately. Note that this has no effect on the *Tango* window, except that it won't receive any more messages from *Raft*. The user must reset the window if another algorithm is to be run.
- change (a)dversary - This allows the user to change the adversary before the next step is taken. Note that one of the adversary selections is to run completely free of faults.

3.6 Processor Trace Facility

Once the simulation is complete, the user is offered the opportunity to trace the path of any processors as they traveled through the tree. If the user chooses to trace the path of a processor through the tree a new tree is drawn to replace the edges deleted in the original tree and the path is highlighted by overlaying green or yellow arrows on the edges the processor traversed (see figure 5). Figure 5 was taken from an execution of *Raft* using $P = 64$, $N = 64$, the *X* algorithm and the Lower Bound Adversary. *X* took 19 cycles to finish all the work, there were 46 processors alive at

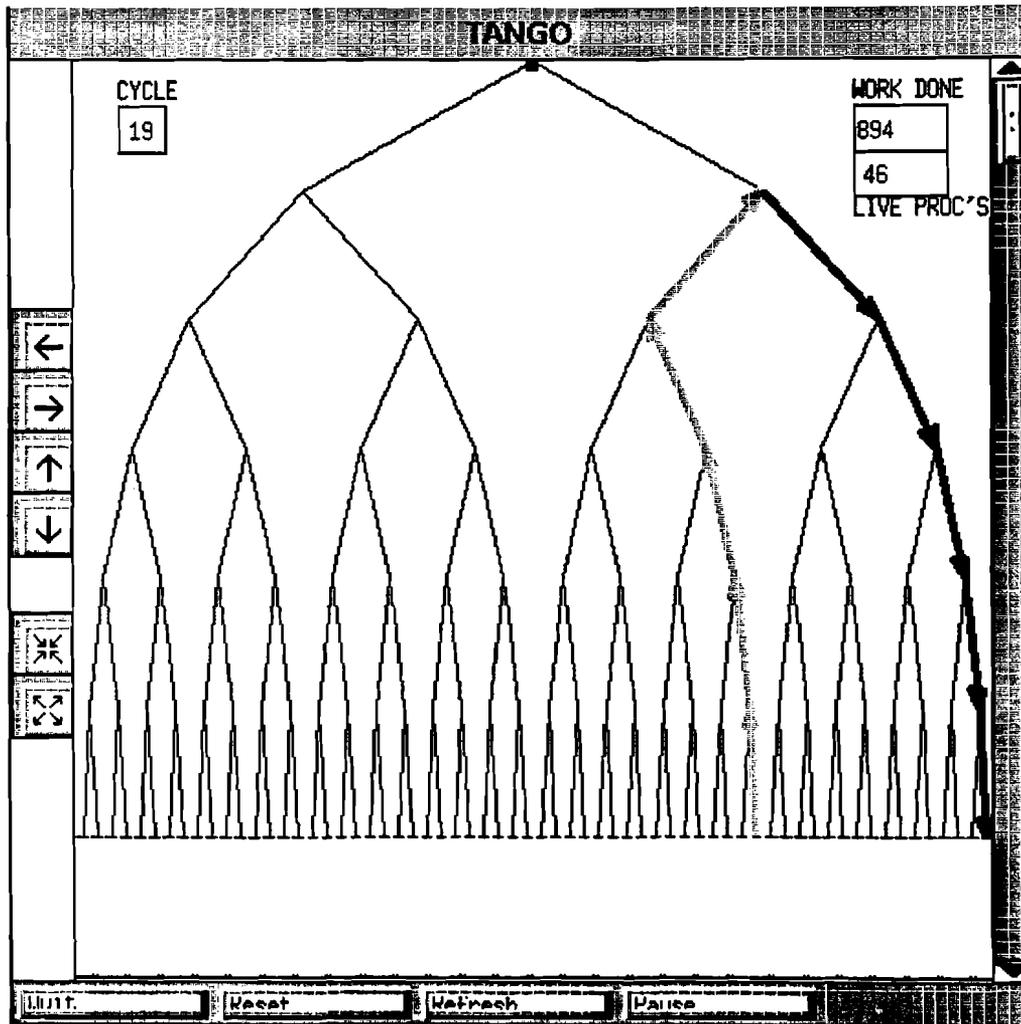


Figure 5: Trace of processor

the finish of the algorithm, and they had performed work $S = 894$. The user chose to trace the path of processor 47 in this figure. Although the picture is not color, it is evident that there are arrows traversing the edges from the 47th leaf node up to the root of the subtree containing unfinished work, and down to the 63rd leaf node (processor 63 was killed by the lower bound adversary in step 1). When processor 47 reached leaf node 63 it was killed by the adversary, thus its path ended at leaf node 63.

Note that if the user asks to see the path of a processor which was killed in the first cycle, no edges will be highlighted. At the same time the animation window is being updated, there is also data being printed in the User Interface window, describing which nodes were visited, with the leaf nodes indicated, in the order the processor visited them

4 Adversary Options

The adversaries are used to control when each processor is subject to failure during the simulation.

There are currently 11 adversaries available to the user. They range from totally interactive to completely specified before execution. In sections 4.1 - 4.11 these adversaries are described completely. Note that the user has the ability to change adversaries at any point in the simulation if running in *single step* mode (as opposed to continuous mode where the algorithm runs through to completion allowing interaction only through the adversary, see section 3.5). The user also has the option of running the animation with no adversary, i.e. with no failures.

The adversaries work on two different principles: failing specific processors, and failing processors assigned to specific nodes in the tree. Most adversaries are designed to fail specific processors. The adversaries which make use of the *Tango* I/O facility allow the user to use the mouse to choose specific nodes in the animation window which are to have the assigned processors failed. Note that in general there may be any number of processors assigned to a node. If the user needs to know specifically which processors are assigned to each node he/she can use the query function to determine exactly which processors are assigned to which nodes. Thus, instead of choosing specific processor numbers to be toggled, the user is choosing specific nodes in the tree which will have all/some (deterministic/probabilistic adversaries) processors assigned to them toggled. Note, depending on the number of processors being simulated, the resolution of the cross hairs in the *Tango* screen is not good enough to pick out single nodes. At around 256 nodes it starts being very difficult to select single nodes at the leaves of the tree. At 1024 processors there is simply a solid dark band at the leaves of the tree. The user would be encouraged to use adversary #4 until the nodes in the tree are thinned out some. Alternatively, the user could make use of the ZOOM functionality of the *Tango* window to focus on smaller portions of the tree, making it possible to select single nodes. The user would then use the PAN LEFT, PAN RIGHT, PAN UP, and PAN DOWN buttons to select nodes outside the visible portion of the *Tango* window. If this technique is to be used it is necessary to ZOOM to the interesting part of the tree before the adversary is waiting for input each step. If the adversary has been invoked, and is waiting for the user to choose nodes in the tree, it is not possible to use the mouse to click on the ZOOM or PAN buttons until the adversary is complete.

The adversary is invoked at different times depending on which algorithm is being run. For the X algorithms, the adversary is invoked before each local step is taken. For the W algorithms each step consists of 4 phases, and the adversary is optionally invoked by the user between each phase. Note that at the beginning of the simulation all processors are assumed to be alive. It is also assumed that the adversary never fails all the processors in any one cycle. There should always be at least one processor alive to continue the execution of the algorithm.

The numbers given in parentheses in the following headings are the numbers used to identify the adversaries by *Raft*. When running *Raft*, use those numbers to choose the different adversaries.

Examples of each adversary are shown below. Two of these examples, Random Adversary, and Adversary 6 (mouse driven), have been extended and include some of the animation windows.

4.1 Random Adversary (#1)

This adversary is designed to choose processors to fail by random techniques. The user is prompted to enter a number between 0 and 100, Pr . Each time this adversary is invoked the pseudo-random number generator is used to get a random number between 0 and $2^{30} - 1$. This number is then normalized such that it lies between 0 and 100. If the normalized number is less than Pr the processor is toggled, otherwise the processor is not affected in this step. Pr is intended to reflect the percentage of processors that could be toggled given that the normalized number is equally likely to be any integer in the range 0 – 100. During each step, due to the random nature of this adversary, it is possible that no processors are toggled, or that all processors (except one) are toggled.

The user is also prompted for a second parameter for this adversary. The second parameter is used as a factor by which to decrease Pr between each parallel step of the algorithm. This parameter, a number between 0 and 1, is used to reduce Pr after each step. A choice of 0 will reduce the Pr to 0 in the first step, and a choice of 1 will never reduce Pr . The reason this reduction is available is that the adversary is applied *each step*. It takes only $\log_{\frac{1-Pr}{1-100}} P$ steps (where P is the number of processors) before the number of processors remaining is reduced to one. For example, assuming $Pr = 50$, and a parallel machine of 256 processors, it could take as few as $\log_{\frac{1-50}{1-100}} 256 = 8$ steps before there would only be one processor remaining. Again, note that since the adversary is random, it may never fail any processors. By allowing the user to choose a value by which to monotonically decrease Pr each step, a larger number can be chosen to affect the early steps of the algorithm, when much of the work done is unique to a processor, without penalizing the later steps of the algorithm, when much of the work being done is duplicated as processors are assigned to the same nodes while traveling up and down the tree.

The script in Figure 6, page 18, is typical of one seen when choosing the random adversary. Note that once the adversary prompts the user for the value of Pr it never interferes with the execution again. The user responses are in boldface. Figure 6 also shows some of the animation displays associated with this simulation script.

4.2 Table Driven Adversary (#2)

This adversary allows the user to completely specify the action of each processor before the algorithm is run. The user creates a file which contains a line for each step of the algorithm. Each line contains P 0's and 1's, with the position in the line corresponding to the number of the processor. Note that the user must have at least as many lines in the file as there are possible cycles in the algorithm. This is a hard thing to know apriori, it is suggested that the last line the user is interested in be repeated some number of times for padding to ensure that the program does not encounter EOF while using this adversary.

A future enhancement to *Raft* would be to add a routine which created the file for the user, based on some input specifications, or based on the action of the just completed user driven interactive adversary.

A sample session for adversary # 2 would look as follows:

```
Choice: 2

The user must input the name of the file containing
the table representation of the adversary.
Each line of file represents one cycle.
A 1 should be in column n if the user wants processor n
alive during that step, a 0 if it should be dead.

File Name: adversary.input

Should processors be allowed to restart after a failure? (y/n): n

Run in single step mode, or continuous execution? (s/c)

Mode: s
:
:
```

4.3 Interactive Adversary, prompting for each processor (#3)

On each cycle this adversary queries the user whether there are any changes to be made to the state (alive,dead) of any of the processors. An answer of yes will start a processor by processor prompting of the user. If the processor is alive, the user will be asked if it should be killed. If the processor is dead, and restart is enabled, the user will be asked if the processor should be restarted. If the user wishes to stop the prompting before all processors have been processed, an answer of q to any of the questions will stop the adversary for this cycle.

A sample session with this adversary might look as follows: (user responses are in bold).

```
Run in single step mode, or continuous execution? (s/c)
Mode: s
In response to the : prompt,
type the character s to single step,
type s # to step # steps,
type c for continuous execution,
type ? to query a hidden node,
type q to stop the animation.
type a to change adversaries.
: s
Any changes? ([y]/n)
y
Kill processor #0: (y/n/q)
n
Kill processor #1: (y/n/q)
n
Kill processor #2: (y/n/q)
y
Kill processor #3: (y/n/q)
y
Kill processor #4: (y/n/q)
n
Kill processor #5: (y/n/q)
y
Kill processor #6: (y/n/q)
q
: s
Any changes? ([y]/n)
y
Kill processor #0: (y/n/q)
y
Kill processor #1: (y/n/q)
y
Kill processor #4: (y/n/q)
n
Kill processor #6: (y/n/q)
q
:
:
:
```

Note that in the second step the user is again asked if processor 0 should be killed, but not asked if processors 2,3 should be killed, as they are already dead. If the *revive* flag were set (see section 3), then the user would be queried whether to revive the dead processors also.

<pre> : : Which type of Adversary would you like? : : : Choice: 1 The random number generator is used to fail a percentage of processors in each step. The probability of failure is reduced by a user-chosen factor between steps. The probability is multiplied by the factor after each step. This allows the user to fail a high number of processors early in the simulation without causing all processors to die in a small number of steps. Probability of failure : 50 Factor to reduce Pr by: (between 0 and 1) .50 Should processors be allowed to restart after a failure? (y/n): n Run in single step mode, or continuous execution? (s/c) Mode: S In response to the : prompt, type the character s to single step, type s # to step # steps, type c for continuous execution, type ? to query a hidden node, type q to stop the animation. type a to change adversaries. : S : ? Please enter the number of the operation you would like to perform: </pre>	<pre> 1) expand a given node 2) total number of live processors 3) work done to date 4) return to simulation prompt Operation: 1 Please click on the node you wish to expand? (user clicks on node 22 in animation window) Node index = 22, Value = 0, Live_proc_count = 3, Dead_proc_count = 0 The processors assigned to this node are: 28 (Live) 30 (Live) 31 (Live) Please enter the number of the operation you would like to perform: 1) expand a given node 2) total number of live processors 3) work done to date 4) return to simulation prompt Operation: 4 : C Work Done =389, NlogN=390, NlogNloglogN=1170 Num Cycles = 36 Do you want to trace the path of any processor? (y/n) Answer: n </pre>
---	---

Notes on Example Animation with Adversary #1
 $P = 64, N = 64, \text{Algorithm} = X_j$

1. In figure 6.i, 37 of 64 processors have been killed by the adversary. This is slightly more than 50% (57%), but fairly close.
2. In cycle 2, figure 6.ii, 10 more processors were killed. Note the user chosen percentage has been reduced by half to be 25%. $10/27 = 37\%$, again slightly larger than the 25%, but close.
3. In cycle 3, the percentage of processors to fail went to 12.5%, and only 4 processors were failed, $4/17 = 23\%$, close to the 12.5%. In the following cycles, 1 processor was killed each cycle, until cycle 7 when no more processors are killed, as the probability has been reduced to less than 1.
4. The difference in number of cycles between figures 6.iv and 6.v can be attributed to the small number of processors alive, and the length of the paths that need to be traversed to get to the unfinished leaf nodes.
5. Figure 6.vi shows the skeletal remains of the tree in cycle 26. Note that it takes another 10 cycles for all work to be completed, as the processors must travel down to the unfinished leaf nodes to complete the work there. As the X_j algorithm is being used, the processors do not need to traverse each edge, but may jump to the leaf when they recognize it needs work. If the path from the nodes containing live processors down to the unfinished leaf node, and back to the root is counted, there are more than 10 edges.
6. Note in the script that the user never needs interact with the animation at all. The Random adversary needs no interaction. Unless the user wishes to query or change adversaries there is no need for interaction. The user chose to query for information about node 22 in the tree during cycle 2 (see Figure 6.ii, root is node 0, children 1,2, grandchildren 3,4,5,6, etc.)

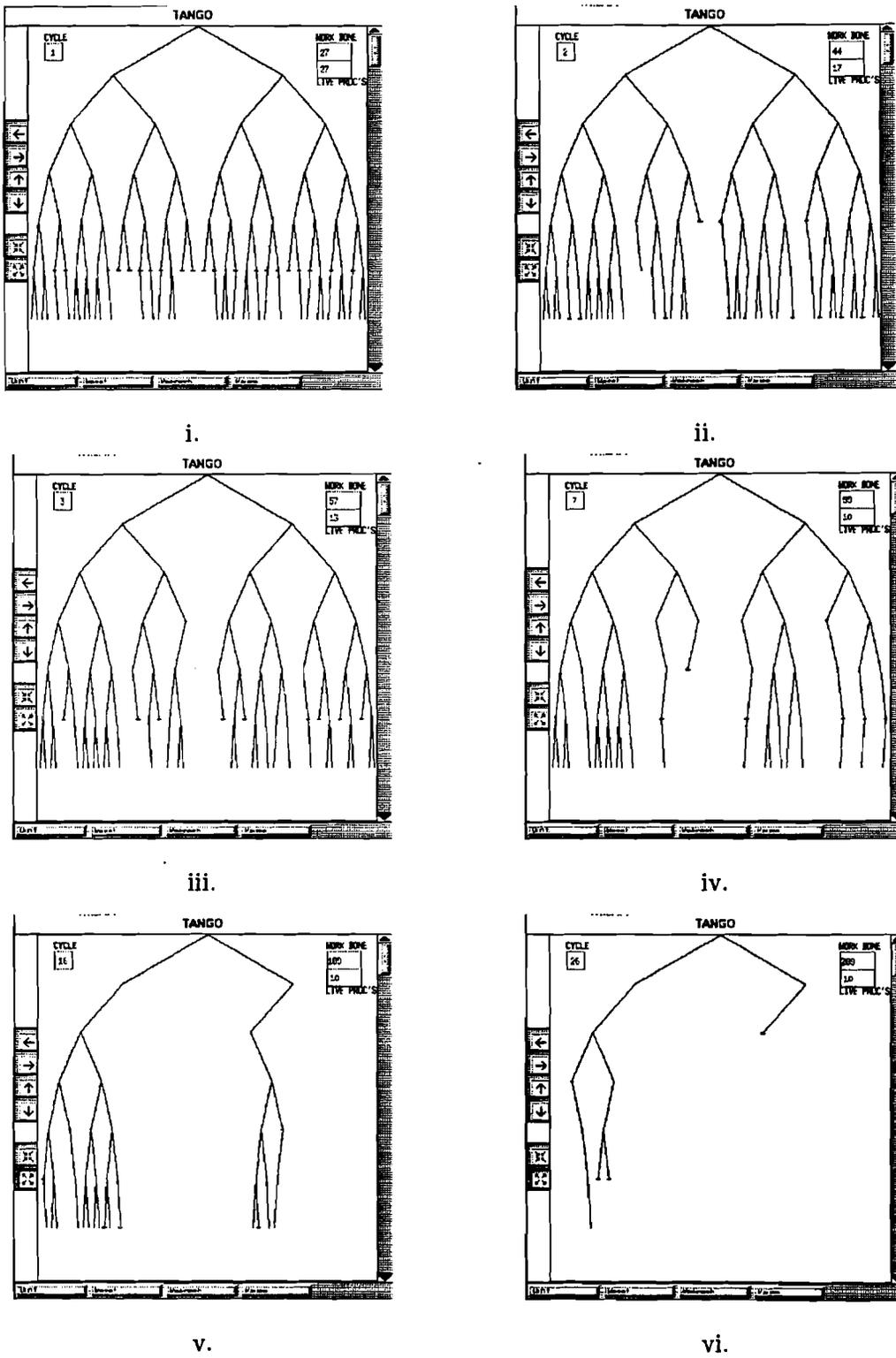


Figure 6: Example animation displays of the X_j algorithm, $P = 64$, $N = 64$, and the Random adversary

4.4 Interactive Adversary, Toggle listed processors (#4)

This interactive adversary prompts the user for a list of processor numbers, separated by commas, which will have their state toggled in the current cycle. If the *revive* flag is set then the dead processors will be revived, otherwise, they are ignored. If there are no changes to be made this cycle, an answer of *n* will cause the simulation to continue.

A sample session would look as follows: (user responses in boldface)

```
Run in single step mode, or continuous execution? (s/c)
Mode: S
In response to the : prompt,
type the character s to single step,
type s # to step # steps,
type c for continuous execution,
type ? to query a hidden node,
type q to stop the animation.
type a to change adversaries.
: S
Toggle which processors?
(give processor number (s) separated by commas)
(enter n to continue to next cycle with no changes)

Cycle Number: 1, Proc's: 0,1,2,3,4,5,6,7,8

: S

Cycle Number: 2, Proc's: 0,1,2,3,4,5,6,7,8

: S

Cycle Number: 3, Proc's: n

: S

:
```

In this example, *revive* is set. In cycle 1, the first 9 processors are dead. In cycle 2 and 3, all *P* processors are alive. If processor number 4 were not included in the command line for cycle number 2, then during cycle two it would still be dead. If *revive* were not set, then the first 9 processors would be dead for all cycles.

4.5 Interactive Adversary, bit vector representing state (#5)

This adversary is the interactive version of the table driven adversary. Each cycle the user is prompted for a string of 0's and 1's representing the state of each of the *P* processors. Note, this is not the adversary to use if you are simulating with lots of processors, as it is necessary to enter a number of 0's and 1's equal to the index of the largest processor being changed. The user is given the choice of entering *n* if there are no changes in this cycle, or of ending a line prematurely with a *q* if there are no more processors to be toggled. There is a limit of 128 processors with this adversary, as it is felt to be too difficult to type in that many 0's and 1's. The user is prompted to use the previous adversary for values of *P* larger than 128.

A sample session would look as follows: (user responses in boldface)

```
Run in single step mode, or continuous execution? (s/c)
Mode: S
In response to the : prompt,
type the character s to single step,
type s # to step # steps,
type c for continuous execution,
type ? to query a hidden node,
type q to stop the animation.
type a to change adversaries.
: S
Enter a string of 0's (dead processors) and 1's (alive processors)
Either enter P 0's and 1's, or end the string with a 'q'
after the last processor to be toggled is specified.
(enter n to continue to next cycle with no changes)

Cycle Number: 1, 0's/1's: 11000000000011110000q

: S
```

```

Cycle Number: 2, 0's/1's: 00000000000000000000000000000000q
: s
Cycle Number: 3, 0's/1's: n
: s
:
:

```

4.6 Interactive, Mouse Driven, select nodes singly, deterministic (#6)

This adversary makes use of the I/O features of *Tango* mentioned in section 5.3. The user positions the cross hairs over a node of the binary tree and left-clicks on the mouse. This selects all processors assigned to that node to be toggled. A box is drawn around the node in the animation window to highlight the choices made. The user clicks on as many nodes as desired each cycle, clicking on the *CYCLE* box when done. Each cycle the user is asked to confirm that the node (s) selected should indeed have all assigned processors toggled. If a mistake was made, the user should answer no to this question. *Raft* then gives the user the option of selecting processors again. The boxes highlighting the chosen nodes are deleted once this question is answered. The figures in Figure 7, page 22, contain some of the animation screens seen with this adversary. The simulation script leading to these pictures is also shown. Note that we actually are implementing the Lower Bound Adversary using interactive techniques here.

4.7 Interactive, Mouse Driven, select nodes singly, probabilistic (#7)

This adversary is similar to the previous one, except that the processors assigned to the selected nodes are subjected to a probabilistic adversary, instead of all being toggled. The user is prompted for a probability of failure, and the same algorithm that is applied in adversary #1 is then applied here. Note that the reduction factor of the probability does not apply in this adversary, as the user chooses a new probability at each step. See section 4.1 for a description of the random adversary.

A sample session with this adversary would be:

```

Run in single step mode, or continuous execution? (s/c)

Mode: c
Click on the node (s) you would like to select, click in CYCLE box when done

(user clicks on 8 nodes in the Tango animation window)

Kill a random percentage of processors assigned to nodes:
63 64 65 66 67 68 69
70
(y/n): y
Probability of failure : 35
Click on the node (s) you would like to select, click in CYCLE box when done

(user clicks on 4 nodes in the Tango animation window)

Kill a random percentage of processors assigned to nodes:
35 36 37 38

(y/n): y
Probability of failure : 50
Click on the node (s) you would like to select, click in CYCLE box when done
Kill a random percentage of processors assigned to nodes:

(user clicks on CYCLE box in the Tango animation window)

No processors selected.

:
:

```

It should be noted that this session was run in *continuous* mode, not *single step* mode. There was no chance to change adversaries, nor to query about any of the nodes. See section 3.5 for more details.

Simulation script for adversary # 6, some text removed, indicated by ellipse.

<pre> : : Which type of Adversary would you like? : : : Choice: 6 This adversary allows the user to select nodes in the tree which will have all processors assigned to them killed. The user will use the mouse to select the nodes. The user must select a single node at a time until all desired nodes have been selected. Should processors be allowed to restart after a failure? (y/n): n Run in single step mode, or continuous execution? (s/c) Mode: S In response to the : prompt, type the character s to single step, type s # to step # steps, type c for continuous execution, type ? to query a hidden node, type q to stop the animation. type a to change adversaries. Click on the node (s) you would like to select, click in CYCLE box when done (user clicks on the nodes seen surrounded by the boxes in figure i) Kill all processors assigned to node (s)? 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 (y/n): Y : S Click on the node (s) you would like to select, click in CYCLE box when done (user clicks on nodes with no processors assigned) Kill all processors assigned to node (s)? No Nodes selected. : a Which type of Adversary would you like? : : : Choice: 11 This choice causes the algorithm to be run adversary free. No more processors will be killed. </pre>	<pre> : S 8 : a Which type of Adversary would you like? : : : Choice: 6 : : Click on the node (s) you would like to select, click in CYCLE box when done Kill all processors assigned to node (s)? No Nodes selected. : S Click on the node (s) you would like to select, click in CYCLE box when done Kill all processors assigned to node (s)? 63 64 (y/n): n Would you like to select more nodes? (y/n) Y Click on the node (s) you would like to select, click in CYCLE box when done Kill all processors assigned to node (s)? 77 78 (y/n): Y : S Click on the node (s) you would like to select, click in CYCLE box when done Kill all processors assigned to node (s)? No Nodes selected. : a Which type of Adversary would you like? : : : Choice: 11 This choice causes the algorithm to be run adversary free. No more processors will be killed. : C : : Work Done =894, NlogN=390, NlogNloglogN=1170 Num Cycles = 19 Do you want to trace the path of any processor? (y/n) Answer: n </pre>
--	---

Notes on Example Animation with Adversary #6

1. Note in figure 7.i, the first ten nodes have been selected by the user for failure as indicated by the boxes surrounding the nodes. These nodes were selected using the mouse in the Tango window, left-clicking on the desired nodes.
2. In figure 7.ii, the 54 live processors have moved up the tree in step one, and the user has just chosen the nodes 17-20 to fail in step 2. Note there are no processors assigned to these nodes, thus none are failed.
3. Figure 7.iii shows cycle 9, when the live processors have traversed through the tree and are ready to descend to unfinished leaf nodes.
4. Figure 7.iv shows that the user has selected leaf nodes 0 and 1 for failure. For illustration purposes, these nodes were cancelled and 2 different nodes, leaf nodes 14 and 15, were selected for failure in step 11 (see figure 7.v). This mimics the behavior of the lower bound adversary.
5. Figure 7.vi shows the last two leaf nodes just after the live processors have reached them. After completing the work at these nodes the processors traverse up to the root and X has completed.

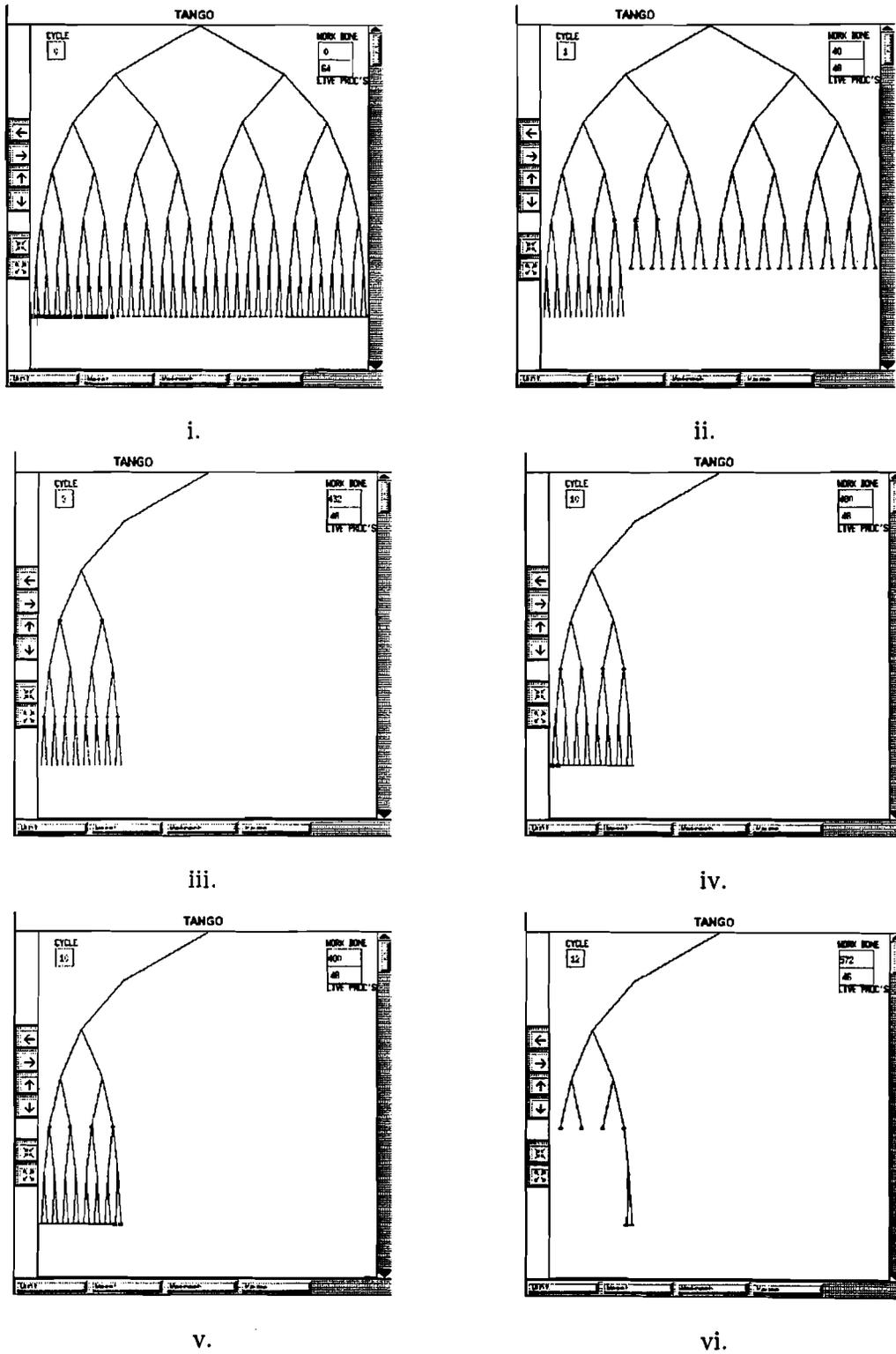


Figure 7: Example animation displays of the X algorithm, $P = 64$, $N = 64$, and an interactive mouse driven adversary

4.8 Interactive, Mouse Driven, select regions of nodes, deterministic (#8)

The following two adversaries are identical to the previous two, except that the user selects regions of nodes with the mouse as opposed to single nodes. A region is some consecutively numbered set of nodes, all at the same level of the tree. Left click on the left most node of the region, then left click on the right most node of the region. All nodes between those two nodes, inclusive, will have the processors assigned to them toggled. The user can select as many regions as desired during each cycle. Left-click twice on the CYCLE box to end the region selection for this cycle.

A sample session would look as:

```
Run in single step mode, or continuous execution? (s/c)

Mode: s
In response to the : prompt,
type the character s to single step,
type s # to step # steps,
type c for continuous execution,
type ? to query a hidden node,
type q to stop the animation.
type a to change adversaries.
: s
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user clicks on 2 nodes (6 nodes apart) in the Tango animation window)

Kill all processors assigned to node (s)?
63 64 65 66 67 68 69
70
(y/n): n
Would you like to select different nodes? (y/n)
y
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user clicks on 2 nodes (5 nodes apart) in the Tango animation window)

Kill all processors assigned to node (s)?
63 64 65 66 67 68 69
(y/n): y
Select another region, or double click on the cycle box if done.

(user double clicks on CYCLE box in the Tango animation window)

: s
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user clicks on 2 nodes (4 nodes apart) in the Tango animation window)

Kill all processors assigned to node (s)?
35 36 37 38
39 40
(y/n): y
:
:
```

4.9 Interactive, Mouse Driven, select regions of nodes, probabilistic (#9)

This adversary applies a probabilistic adversary to the regions of nodes the user has selected each cycle. Selection is done exactly as for the previous adversary. The user will be prompted for a probability of failure after each region is selected. Double-clicking on the CYCLE box will stop region selection for this cycle.

A sample session would be:

```
Run in single step mode, or continuous execution? (s/c)

Mode: s
In response to the : prompt,
type the character s to single step,
type s # to step # steps,
type c for continuous execution,
```

```

type ? to query a hidden node,
type q to stop the animation.
type a to change adversaries.
: S
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user clicks on the 2 boundary nodes in the Tango animation window)

Kill a percentage of processors assigned to node (s)?
63 64 65 66 67 68 69
70 71 72 73 74 75 76
77 78
(y/n): Y
Probability of failure : 50
Select another region, or double click to end selection

(user double clicks on CYCLE box in the Tango animation window)

: S
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user clicks on the 2 boundary nodes in the Tango animation window)

Kill a percentage of processors assigned to node (s)?
39 40 41 42

(y/n): Y
Probability of failure : 100
Select another region, or double click to end selection

(user double clicks on CYCLE box in the Tango animation window)

: S
Click first on the left most node in the region,
followed by the right most node in the region.
Select as many regions as you like.
Double click on the CYCLE box when you are done.

(user double clicks on CYCLE box in the Tango animation window)

: S
:
:

```

4.10 Lower Bound Adversary (#10)

This adversary is designed to cause the worst possible behavior of the X algorithms. The adversary has the following behavior.

Select a number, U_i , of the tasks remaining to be done (in this case, array elements waiting to be written) which have the fewest number of processors, including 0 processors, assigned to complete them. These tasks are at the leaves of the tree by definition of X. Thus, the only processors eligible for failure by this adversary are those assigned to the leaf nodes. The number of nodes which have their assigned processors failed each step is U_i , where $U_0 = N$, and $U_i = U_{i-1} / \log U_0$. Thus in the first step, $N / \log N$ nodes are selected. Note that if $N = P$ then $N / \log N$ processors are failed as there is a single processor assigned to each leaf node.

The index, i , is incremented only when U_i processors are actually failed in a step. It is possible that there are no processors assigned to the leaf nodes, or that the number of nodes with processors assigned is less than U_i . In either of these cases, i is not incremented. After a small number of iterations of i , the number of processors to be failed goes to 0. $\left\lceil U_i = \frac{U_{i-1}}{\log U_0} \right\rceil$ goes to 0 after $i = \left\lceil \log_{\log U_0} U_0 \right\rceil + 1$ steps.

The position of the nodes selected for failure has significance to the efficiency of the adversary. The performance of the algorithms is worst when the processors to be failed are grouped according to the largest possible subtree of height $\leq \log U_i$. If there are more than U_i nodes which are eligible for failure (ie. the U_i^{th} node can be chosen from a set of nodes with the an equal number of assigned processors), then the adversary is most effective if the nodes are selected to be in consecutively

numbered leaves, starting on the boundary of a subtree of height $\log U_i$ (if possible).

If the nodes to be failed were not grouped, then neighboring nodes would be able to finish the uncompleted work 2 steps later, one step to get to the root of the subtree, one step to get down to the unfinished leaf and perform the work. Then all processors would only have to ascend to the root of the tree and the algorithm would be finished.

There is a slight improvement which can be added to the lower bound adversary to cover all possible values of N . If the value of $U_1 = \lfloor \frac{N}{\log N} \rfloor$ is a power of 2 (this is true for all values of $N = 2^y$ such that $\log N = 2^x$) then an entire subtree (call it subtree B) will have the processors at its leaf nodes failed. This leaves the processors at the leaves of the neighboring subtree of equal size (call it subtree A) free to traverse up the tree and down to the failed nodes in $2 * \log U_1$ steps. If $U_1 = \lfloor \frac{N}{\log N} \rfloor$ is not a power of 2, then there will still be processors alive in a portion of subtree B. These processors will proceed to perform work as they travel through subtree B, without having to wait the $2 * \log U_1$ steps for the processors of subtree A to arrive to do the work. Thus if the adversary fails an entire subtree of processors it will cause more work to be done. So each step, set U_i equal to the smallest power of 2 greater than U_i .

When this adversary is selected there is no interaction between the adversary and the user during the execution of the algorithm. The user still has the opportunity to use the query functionality, but as the action of this adversary is completely specified by the selection of P and N there is no more input required by the adversary here. Simply type 10 in response to the prompt for an adversary.

4.11 Non-existent Adversary (#11)

This choice allows the algorithm to run without any processors being failed. The user simply types 11 in response to the adversary prompting and no other input is needed by the adversary.

5 Implementation

5.1 General

There are 3 modules of code used to build the *Raft* tool.

1. *Raft.c* - This module contains all the user-interface code, all the algorithm routines, all the adversary routines, and the code written to interface with the *Tango* software. This module is roughly 4000 lines of documented C code, including 43 subroutines.
2. *Raftscenes.c* - This module contains all the code written to perform the animation. It is compiled into an object file, then linked in directly with the *Tango* software. *Tango* calls these routines when messages are passed from *Raft* via the [Reis89] facility. This module is composed of roughly 1000 lines of documented C code, containing the 6 routines mentioned in Section 5.3 below, with numerous calls to the *Tango* package routines.
3. *Raft.h* is an include file which keeps track of all the variables and constants required by this software.

The *Tango* and *MSG* software is also necessary for the animation window to be activated. *Tango* is comprised of 14 modules of commented C code, totalling roughly 8500 lines. The *MSG* software provides a rich set of message passing utilities for use between processes. The utility used by *Raft* is a single routine of roughly 150 lines of C code which sends a message via a routine call and returns a string to the caller.

The next two sections go into some more detail on the user interface software, and the *Tango*-algorithm interface routines.

5.2 User Interface Implementation Notes

The keyboard interface employed by *Raft* is very simple by design. It consists of a main loop which performs all the necessary initialization, prompting the user for the necessary parameters (see Section 3, and setting up all the global data structures from heap memory. It then calls the routine which implements the user chosen algorithm. The code written to implement the algorithm takes care of all the remaining interaction during execution. All the algorithm routines follow the general outline given below:

1. call the algorithm specific initialization routine. This sets up the animation window, allocates and initializes any algorithm specific data structures, and initializes the global data structures.
2. enter a loop controlling the execution of the algorithm. Each iteration of the loop represents one PRAM step of the algorithm. (The implemented algorithms are controlled by the value of the work done in the root node being equal to N .)
3. Inside the loop the following steps are taken:
 - (a) Call the adversary routine to toggle any processors this step.
 - (b) If in *single step* mode, call single step routine to allow the user to query before step is taken. (Note, user may change adversaries at this point, if so, routine must clear out any previously toggled processors for this step, and then invoke new adversary).
 - (c) Enter the synchronous loop, allowing each processor to perform the algorithm, making calls to the animation window to update the screen with any changes. (Note, the changes to the animation window are being composed by the user provided *Tango* routines at this point so that they will appear to happen all at the same time, see Section 5.3)
 - (d) Send a message to the *Tango* window to perform all the composed actions, then copy the write data structure onto the read data structure to implement the shared memory (see below).
4. Go back to the top of the loop

In order to implement the PRAM model used by these algorithms, it is necessary to simulate shared memory access. As the processors loop synchronously to perform a single PRAM step, it is necessary to keep two copies of the controlling data structure. The first copy stores the current value of the work done at each node. The processors read from this data structure while performing the algorithm. The second copy of the data structure is used for all writes. Thus, each processor reads the same value and can write any new value into the second data structure. At the end of each synchronous loop, the write data structure is copied into the read data structure, effectively simulating shared memory. Note, as mentioned in Section 2, each processor is concurrently writing the same value to memory, thus there is no priority scheme necessary.

Adding an algorithm to *Raft* is discussed in Section 6.2.

The adversary routines are all resident in the user interface module. They all make use of the global arrays *Live* and *changed* to keep track of the state of each processor. The code written varies greatly depending on the nature of the adversary. However, in general, there is a loop of P iterations which writes either a 1 or a 0 in the *Live* array, and sets a bit in the *changed* array if the state toggled (this is necessary to inform the main loop which processors need to send animation information across to the *Tango* window). It is important to note that there must always be at least one processor which remains alive, else the algorithm's will not execute properly. Adding an adversary to *Raft* is discussed in Section 6.1.

5.3 Tango-Algorithm Interface Implementation Notes

Raft interfaces to *Tango* through a message passing facility. This message passing facility is a layer of software as described in [Reis89]. A routine is called which passes messages from *Raft* to the *Tango* software. The message passed is a character string which is then parsed by *Tango* to determine which of the user-provided routines to invoke. The user provided animation scene routine is then passed the parameters which *Tango* parses out of the character string and run. There is also an X11 based version of *Tango* which avoids this extra layer of software. A version of *Raft* has been created to work with *Xtango* also. See [SH90] for a detailed description of the differences between *Tango* and *Xtango*. The main difference is that the entire animation is run in a single process, thus there is no need for the message passing facility.

There are some requirements on *Raft* which are outlined here, as are it's capabilities to affect the animation window.

In order to keep track of the state of the animation window, there is a data structure which all the algorithms make use of. Since the data structure displayed in the animation window is a binary tree, it makes sense that the algorithms keep track of the state of the animation window using a binary tree. In order to keep the animation as fast as possible, the execution of the algorithm is not animated fully. Anytime there is duplication by the algorithm, (for example 2 processors traveling together over an edge, or multiple processors resident at a single node of the tree) there is only a single image shown in the animation window. The duplicate image would be overlaid on the screen and appear as a single image anyway. This provides a speedup in animation time, and requires much less memory to store the animation actions. This tree is represented as an array of records. The information stored in the tree is important because *Tango* is very particular about creating and/or deleting already existing images. If *Raft* sent a message to the *Tango* window requesting to delete a processor line at a node, and the line didn't exist, the *Tango* window would crash. Each record, one per node of the tree, contains the following pieces of information:

1. *proc_line_present* - This flag keeps track of whether there is a red line representing the fact that there is a processor alive at the node.
2. *num_procs_at_node* - This field is used to keep track of the required thickness of the lines at each node. The line is scaled by the log of the number of processors at the node.
3. *parent_edge* - When the algorithm completes the task (s) of a subtree, the parent edge to that subtree are deleted. This flag keeps track of the fact that the parent edge has already been deleted.

There are 6 routines which are used to control the animation window. Using these routines the user may perform the following actions:

- Routine *BalTree*
 - Create a balanced binary tree with red processor lines at all leaf nodes.
- Routines *TreeNode* & *KillTreeNode*
 - Select specific nodes of the tree using the mouse and return the number of the node (root is 0, children are 1,2, grandchildren are 3,4,5,6, etc.) (useful for adversaries or queries)
 - Select a range of nodes in the tree and return the number of the boundary nodes on the left and right.

- Routine MoveProcLine
 - Create a red line representing a processor present at a node.
 - Move an already created red line from one node to another.
 - Thicken an already created red line to represent more than one processor at a node
 - Delete a red processor line at a specified node.
 - Delete the thickened red processor line at a specified node.
 - Remove an edge from the binary tree to represent work in that subtree done.
- Routines Highlight & UnHighlight
 - Highlight/Unhighlight an edge between two nodes

In order to change the capabilities of the animation, it would be necessary to write new code for the animation scenes. These animation scenes make use of routines provided by the *Tango* software. Note that all these routines utilize the MALLOC library routine in C to grab memory from the heap. *Tango* keeps all this memory unless specifically told to free it. The routines that are linked in with the *Tango* image to perform the tasks listed above were carefully designed to return memory whenever possible. If new routines are designed, the author must be very aware of the memory requests made, else the animation slows down tremendously.

Tango keeps each element (line, circle, text, etc.) of the animation window in its own record in heap memory. It also stores all the actions performed on each element in heap memory. These records are not released back to the heap after the action has been performed, unless explicitly requested. Special attention need be paid to the *composition* and *concatenation* of *Tango* actions. These functions are used to make two single actions appear to occur at the same time (*composition*, or sequentially *concatenation*, in the animation window. Composing/concatenating two transactions into one creates an entirely new record in heap memory, even if the code asks for the composed action to be pointed to by one of the original actions. This causes memory requirements to expand dramatically if care is not taken to assign the new actions to temporary pointers, delete the old actions, reassign the temporary actions, then delete the temporary actions.

5.3.1 BalTree routine

This routine must be called first to initialize the animation window. It sets up all the data structures and necessary *Tango* hooks, then draws a binary tree with N leaves ($N/\log N$ leaves for the optimized version of W) in the animation window. As parameters to this routine the calling program must send the number of processors to be animated, P (they are represented as red lines at the leaf nodes of the tree), the number of leaf nodes to be drawn, N , and a third parameter, *trace*, which is used to differentiate between the initial call to this routine and calls generated when using the processor trace facility. Because the algorithms destroy the image of the binary tree in order to represent work being completed, it is necessary to redraw the tree when the algorithm is complete if the user would like to trace the path any processor took during the execution of the algorithm. *trace* should be 0 when calling *BalTree* initially, and a 1 when the trace facility is redrawing the tree.

Tango uses a feature called an *association* to keep track of all the different line segments, text, shapes, etc. which create the figure (s) in the animation window. All the *Tango* provided routines make use of these associations with a hashing technique to get pointers to the images when deleting, creating, or moving an image in the animation window. This routine creates the initial definitions of these associations to be used later.

This routine also tells *Tango* to draw in the **CYCLE** box, **WORK DONE** box, and the **LIVE PROCESSORS** box.

5.3.2 MoveProcLine routine

This routine is responsible for changing the image of the binary tree when the algorithms are being executed. It takes care of telling *Tango* when to move which parts of the animation window. Due to the varied nature of the tasks this routine performs, it has 8 parameters passed to it.

- *pos* - the number of the node in the tree representing the position of the processor being simulated. The nodes are numbered consecutively starting at the root (# 0) going left to right across levels of the tree.
- *newpos* - the number of the node in the tree representing the position that the processor being simulated wants to move to.
- *pn* - Processor number. This is not actually used in the animation window itself, but is very important as an aid in differentiating between the associations made by *Tango*. Two lines sharing the same space in the animation window can be differentiated by the association if they have been stored with unique identifiers, which are the processor numbers in this case.
- *op* - This parameter tells the routine which of the 6 different operations to perform on the animation window.
 1. REMV_EDGE - This tells the routine that the edge from node *pos* to node *newpos* in the tree is to be removed. This represents the fact that all the work in the subtree with node *pos* as root has been completed.
 2. REMV_PROCLINE - This tells the routine that the red line representing the presence of a processor at node *pos* is to be removed. This is used if the node a processor is being moved to already has a red line drawn to avoid duplication in the animation window.
 3. MOVE_PROCLINE - This tells the routine that the red line representing the presence of a processor at node *pos* is to be moved to node *newpos*. This is used to animate the path a processor takes through the tree.
 4. CREATE_PROCLINE - This tells the routine that a red line needs to be created at a node to represent the arrival of a processor. It is used when the node being moved from no longer has a red line resident (some other processor previously at this node has already moved to another node).
 5. INCR_WIDTH - The width of the red line at a node represents the number of processors there this step. The width of the line is scaled to the log of the number of processors there.
 6. ALG_W - When the W algorithm is being animated, this tells the routine to move a red processor line from the root node to the node in *newpos* (a leaf node). This represents the path the node took from the root down to the leaf node. Note that in the interest of speed of animation the entire path is not shown as it is unique.
- *size* - This parameter holds the number of processors at node *pos*. It is used by the routine in conjunction with the *op* INCR_WIDTH when drawing the thick red line representing the number of processors at the node.
- *cyc_count* - This tells the routine the amount of work done to this point. It is only valid when the last parameter, *done*, is set to 1. The routine writes this number in the WORK DONE box.

- *steps* - This tells the routine how many steps have been simulated so far. It is written in the CYCLE box when the last parameter, *done*, is set to 1.
- *done* - This parameter tells the routine that there are no more actions to be added to the animation this step. The routine then goes off and issues the message to *Tango* that it should perform all the stored actions. Due to the speed at which *Tango* performs the animation it is necessary to use a *Tango* feature known as *composing* to make one large animation action out of all the animation requests sent each step. There are a set of linked lists kept of all animation requests. When the *done* parameter is set to 1, these linked lists are processed with the actions being composed into one large action for *Tango* to perform. The memory associated with these actions is then given back to the heap. If the composition feature wasn't used, each action would be performed consecutively, leading to a dramatic slowdown in animation, and a loss in the appearance of the algorithm being executed in parallel.

5.3.3 Highlight & UnHighlight routines

These two routines are used to highlight the path a processor took through the binary tree during the execution of the algorithm. These routines are passed three parameters: the numbers of the two nodes between which the path should be highlighted, and the direction the processor was going when traversing the edge.

A green/yellow arrow is drawn between the two nodes depending on whether the processor is going up or down in the tree. A processor may pass through the same edge a number of times in either direction. If a processor is passing over an edge for the second time a new arrow is not drawn, instead, the previously drawn arrow is moved to the top plane of the animation window.

Once the entire path a particular processor took through the tree is highlighted the user is offered the chance to view another processors path. If the user chooses to do so, the currently highlighted path is erased using the UnHighlight routine. The parameters are exactly the same. They are used to fetch the images of the arrow for deletion. Note that all these messages requesting arrows be deleted are composed into a single animation action and performed only when the *done* parameter is set to a 1. This makes the edges appear to disappear together. When highlighting the path, the animation messages are handled immediately, in a synchronous fashion to give the appearance of traveling through the tree.

5.3.4 TreeNode & KillTreeNode routines

These routines are used to interact with the mouse and the *Tango* animation window. They make use of a message passing facility to pass back to the calling routine a pointer to a character string. The contents of this string are set up by these routines to contain the number of the node the user clicked on with the mouse.

The *TreeNode* routine takes as a parameter the number of leaf nodes in the tree. It uses the *Tango* IO facility to allow the user to choose a node. The node number is passed back to the calling routine and all known information about that node is displayed to the user. There is a slight margin of error that is present with the mouse. The node with the smallest node number that is within the margin of error is passed back to the user. Thus if the nodes of the tree are quite dense, the user will have some difficulty choosing them uniquely.

The *KillTreeNode* routine is used in conjunction with the adversaries. There are two parameters to this routine: the number of leaf nodes, *num_nodes* and a boolean, *done*. This routine allows the user to choose either one or two nodes, depending on the value of *done*. If *done* is equal to 0, then the routine was called by an adversary selecting single nodes. If *done* is equal to 2 then the routine was called by an adversary selecting a region of nodes. In either case the chosen nodes have a red

box drawn around them to indicate that they have been selected, and the node number is passed back to the calling routine. When the last parameter, *done*, is set to 1 all the boxes outlining the chosen nodes are deleted.

6 Enhancements

Additional algorithms, or adversaries could be added to *Raft* to increase it's functionality. There are certain requirements that must be followed in order to fit new algorithms, or adversaries, cohesively into the *Raft* tool. These requirements are briefly addressed below.

6.1 Adding Adversaries

It is fairly straightforward to add an adversary to *Raft*. There is a single data structure which is used to hold the state (alive/dead) of each processor. Each time the new adversary is invoked it should leave the state of each processor in the dynamically allocated global array, *Live*. If processor *i* is alive, there should be a 1 in element *i* of the array, if processor *i* is dead there should be a 0 in element *i* of the array. It should also update element *i* of the global array *changed*. This array tells *Raft* whether there has been any change in the state of a processor since the last step. This information is used to determine whether to call the animation routines for this processor in this step; a 1 in element *i* means the state has changed for processor *i*, a 0 means the state remained the same. Note that the adversary must ensure that at least one processor remains alive in each cycle, else the algorithms cease to work correctly.

The new adversary should be added to the CASE statement which selects the adversary to perform. This is done in routine *adversary_setup*. A brief textual description of the adversary should be added here also. The routine which performs the adversary should then be added to the CASE statement in the routine *count_live_procs_and_perform_adversary* under the appropriate switch value.

6.2 Adding an algorithm

There are certain requirements that are imposed by *Raft* on the algorithms to be animated. These requirements are due to the rigid method in which the *Tango* animation scenes must be coded. Of course, by writing additional *Tango* animation scene routines, any algorithm may be animated.

In order to add an algorithm to the suite of those the user may select, without having to write new *Tango* animation scene routines, the following prerequisites must be met:

1. The algorithm must make use of a binary tree as the displayed data structure, or not make use of the animation facility. If it is desired to animate an n-ary tree it would be necessary to write new *Tango* routines to control the n children at each node of the tree. The existing *Tango* routines provide for creation of binary trees or graphs.
2. The algorithm needs to have calls to the *Tango* animation scenes inserted at the appropriate points. These routines are described in section 5.3. The syntax of the message passing is described in [Reis89]. There is also additional code required to update the data structure representing the state of the animation display. This is where the bulk of the work will be.
3. The algorithm needs to be added to the existing CASE statement in routine *algorithm_setup*, and in routine *main*, a new initialization routine needs to be written for the required data structures (all are dynamically allocated), and a brief description of the algorithm written for display upon user request.

4. There is a defined interface with the adversaries that must be adhered to. The adversaries keep an array **Live** where each element represents the state of a processor. This array must be used in the algorithm to control the action of live/dead processors. Also, element i of array **changed** needs to be checked to see if any animation action updates need to be sent across to the *Tango* process. Calls to the routine *count_live_procs_and_perform_adversary* should be inserted in the algorithm at the appropriate points. Alternatively, a completely new adversary could be written to make use of existing structures in the proposed algorithm.
5. In order to provide the processor trace capability, it is necessary to keep a history of the nodes each processor visits. This is to be stored in a linked list of linked lists, **history**, with one list per processor. The i th element of the primary list represents the initial position of the i th processor. Each element of the secondary lists represents the next change in position of the processor.

6.3 Future Considerations

A future enhancement to *Raft* would be to add a routine which creates a file for the Table Driven adversary automatically. It could do this based on some input specifications, or based on the action of the just completed user driven interactive adversary.

It would also be nice to add all known algorithms solving the fault tolerant *Write-All* problem.

An interesting modification would be to update *Raft* for execution on a parallel architecture, such as the Connection Machine. This would allow direct simulation of the algorithms, without the overhead associated with simulating on a single cpu. This would add some additional complexity in the memory access, but as mentioned earlier, this problem has been addressed extensively in the literature.

It would be nice to update the interface to be menu driven, via Motif, or some such toolkit.

If you are interested in more details, it is suggested that you contact the author at swa@cs.brown.edu, or apgar@oblio.enet.dec.com. Enhancements to the tool are encouraged!

References

- [AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F.P. Preparata. "Deterministic simulation of idealized parallel computers on more realistic ones," *SIAM Journal of Computing*, 16 (5):808-835, Oct 1987.
- [AW 91] R. Anderson and H. Woll, "Wait-Free Parallel Algorithms for the Union-Find Problem", *Proc. of the 23rd ACM Symp. on Theory of Computing*, pp. 370-380, 1991.
- [BCDW91] E. A. Brewer, C. N. Dellarocas, A. Colbrok, W. E. Weihl. "PROTEUS: A High-performance Parallel-Architecture Simulator," Massachusetts Institute for Technology, Laboratory for Computer Science, Technical Report MIT/LCS/TR-516, Sept. 91.
- [BKRS 91] J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman, "Parallel algorithms with processor failures and delays", Brown Univ. Tech. Report CS-91-54, August 1991.
- [Brow87] M. H. Brown. "Algorithm Animation." Ph.D Dissertation, Computer Science Department, Brown University, May 1987.
- [BH91] M. H. Brown, J. Hershberger. "Algorithm Animation using Zeus." Digital Equipment Corp. DEC Systems Research Center, Palo Alto, CA
- [DKM+88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Hiede, H. Rohnert, and R.E. Tarjan. "Dynamic Perfect Hashing: Upper and lower bounds." In *Proceedings of the 29th Annual Symposium on the foundations of Computer Science, White Plains, New York*, pp 524-531, Oct 1988.
- [FW78] S. Fortune, J. Wyllie. "Parallelism in random access machines," *Proc. 10th ACM STOC*, pp. 114-118, 1978.
- [HB88] K.T. Herley and G. Bilardi. "Deterministic simulations of P-RAMs on bounded-degree networks." In *Proceedings of the 26th Annual Allerton Conference on Communication, Control, and Computation, Monticello, Illinois*, pp 1084-1093, Sept 1988.
- [Herl89] K.T. Herley. "Efficient simulation of small shared memories on bounded degree networks." In *Proceedings of the 30th Annual Symposium of the Foundations of Computer Science, Research Triangle Park, North Carolina*, pp 390-395, Oct 1989.
- [Herl90] K.T. Herley. "Space-Efficient Representations of Shared Data for Parallel Computers." In *Journal of the ACM*, (7):407-415 1990.
- [HP89] S.W. Hornick and F.P. Preparata. "Deterministic PRAM Simulation with Constant Redundancy." In *Information and Computation*, Vol 92, No. 1, pp. 81-96, May 1991.
- [KPS90] Z. M. Kedem, K. V. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," in *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 138-148, 1990.
- [KPRS 91] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, "Combining Tentative and Definite Executions for Dependable Parallel Computing," in *Proc. 23rd ACM Symp. on Theory of Computing*, 1991.
- [KR88] R. M. Karp and V. Ramachandran. "A Survey of Parallel Algorithms for shared-Memory Machines," Technical Report UCB/CSD 88/408, Computer Science Department, University of California at Berkeley, CA.

- [KS89] P. Kanellakis, and A. Schvartsman. "Efficient Parallel Algorithms Can Be Made Robust," Technical Report CS-89-35, Department of Computer Science, Brown University, Providence RI.
- [KS91] P. Kanellakis, and A. Schvartsman. "Efficient Parallel Algorithms On Restartable Fail-stop Processors," Technical Report CS-91-36, Department of Computer Science, Brown University, Providence RI.
- [KS91a] P. Kanellakis, and A. Schvartsman. "Robust Computing with Fail-stop Processors," Department of Computer Science, Brown University, Providence RI.
- [KU88] A. R. Karlin and E. Upfal. "Parallel Hashing: An efficient Implementation of Shared Memory." In *Journal of the ACM*, Vol. 35, No. 4 pp. 876-892, Oct 1988.
- [Kuck77] D.J. Kuck. "A Survey of parallel machines organization and programming," In *ACM Comput. Survey*. 9,1 (1977),29-59.
- [Lin91] Y. Lin "A Framework for Automatic Algorithm Animation", Technical Report CS-91-37, Computer Science Department, Brown University, May 1991.
- [LPP88] F. Luccio, A. Pietracaprina, and G. Pucci. "A Probabilistic Simulation of PRAMS on a Bounded Degree Network," In *Information Processing Letters* 28, pp. 141-147, 1988.
- [LPP89] F. Luccio, A. Pietracaprina, and G. Pucci. "A New Scheme for the Deterministic Simulation of PRAMS in VLSI", In *Algorithmica* (1990) 5: 529-544.
- [Meyer86] F. Meyer auf der Heide. "Efficient simulations among several models of parallel computation." In *SIAM Journal of Computing*, 15 (1):106-119, Feb 1986.
- [MSP 90] C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," in *Proc. 32d IEEE Symp. on Found. of Computer Science*, pp. 590-599, 1990.
- [PSW91] M. Palis, S. Rajasekaran, D. Wei. "Emulation of a PRAM on Leveled Networks," In *1991 International Conference on Parallel Processing, Vol I*, pp. 418-421, 1991.
- [Ran87] A. Ranade. "How to emulate shared memory", In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, Los Angeles, California, pp. 185-194, Oct 1987.
- [Reis89] S. Reiss. User Manual, MSG Facility. Brown University.
- [Shv91] A. A. Shvartsman, "Achieving Optimal CRCW PRAM Fault-Tolerance", in *Information Processing Letters*, vol. 39, pp. 59-66, 1991.
- [SS 83] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: an Approach to Designing Fault-tolerant Computing Systems", *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222-238, 1983.
- [Stas89] J. Stasko. "*Tango*: A Framework and System for Algorithm Animation", Ph.D Dissertation, Computer Science Department, Brown University, May 1989.
- [SH90] J. Stasko, D. Hayes. "Xtango Algorithm Animation Designers Package", User manual, College of Computing, Georgia Institute of Technology, Atlanta GA. Dec 1990

- [Upfa84] E. Upfal. "A Probabilistic Relation Between Desirable and Feasible Models of Parallel Computation." In *Proceedings of the 16th ACM Symposium of Theory of Computing* (Washington, D.C., Apr. 30-May 2). ACM, New York, 1984, pp. 258-265.
- [UW87] E. Upfal and A. Wigderson. "How to Share Memory in a Distributed System." *Journal of the ACM*, 34 (1):116-127, Jan 1987.