**Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System**

Anh Nguyen-Tuong
Andrew S. Grimshaw
Mark Hyett

Computer Science Report No. CS-96-13
August 20, 1996

# Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System[1]

Anh Nguyen-Tuong, Andrew S. Grimshaw and Mark Hyett
University of Virginia, Department of Computer Science
Charlottesville, VA 22903

E-mail: {nguyen | grimshaw | mrh2e} @ virginia.edu
URL: http://www.cs.virginia.edu/~an7s, http://www.cs.virginia.edu/~grimshaw,
http://www.cs.virginia.edu/~mrh2e

### *Abstract*

*Wide-area parallel processing systems will soon be available to researchers to solve a range of problems. In these systems, it is certain that host failures and other faults will be a common occurrence. Unfortunately, most parallel processing systems have not been designed with fault-tolerance in mind. Mentat is a high-performance object-oriented parallel processing system that is based on an extension of the data-flow model. The functional nature of data-flow enables both parallelism and fault-tolerance. In this paper, we exploit the data-flow underpinning of Mentat to provide easy-to-use and transparent fault-tolerance. We present results on both a small-scale network and a wide-area heterogeneous environment that consists of three sites: the National Center for Supercomputing Applications, the University of Virginia and the NASA Langley Research Center.*

## 1. Introduction

Recent advances in network technology promise to make gigabit-per-second bandwidth between remote hosts a reality in the near future. This increase in bandwidth paves the way for increased exploitation of distributed computing resources. Coupled with advances in distributed memory parallel compiler technology, there is strong reason to believe that wide-area distributed parallel processing will be an increasingly popular and important programming paradigm. Parallelizing and distributing program sub-tasks has the potential of increasing performance for many applications while also improving the overall utilization of system resources. Unfortunately, there is a downside. When a program is partitioned into sub-tasks, each sub-task may be distributed among different processors. As the number of processors employed by an application increases, so does the chance that the application will fail due to a host failure.

At the University of Virginia, we have first hand experience the problems caused by host failures in distributed systems while developing and using a prototype for the Legion project [12][13] (information on Legion is available on the WWW at http://www.cs.virginia.edu/~legion). The objective of Legion is to construct the software environment to enable a nation-wide or world-wide virtual computer capable of supporting distributed and parallel applications. Our current prototype, that we call the Campus-Wide Virtual Computer (CWVC), contains a mix of over 90 workstations and an IBM SP-2 multicomputer. Even in this relatively small environment, we are frequently experiencing host failures. On the scale of the envisioned nation-wide system, host failures will simply be a fact of life and must be dealt with accordingly. User applications, especially those that are critical or are composed of many distributed components, must be resilient to host failures. Fortunately developing fault tolerant parallel applications does not need to be difficult.

In this paper we show that by developing applications using the data-flow model of parallel computation there is a simple method for providing fault-tolerance. The key to our approach is in exploiting the functional nature of data-flow programs in the fault-tolerance mechanisms. Recall that data-flow computations are modeled by actors, arcs, and tokens. Actors are computation primitives, tokens carry data or control information, and arcs are used to model the dependencies between actors. The

[1] This work is partially funded by NSF grants ASC-9201822, NRaD contract N00014-94-1-0882, and ARPA grant J-FBI-93-116

distinguishing feature of actors in terms of fault tolerance is their idempotent nature: an actor presented with the same tokens will always produce the same result. Thus, fault-tolerance can be easily achieved through actor replication, i.e. replicate an actor k times and use the first available result (discard later arriving results).

In an earlier paper [18], we implemented actor replication and showed its performance and resource consumption characteristics using a synthetic pipeline application on a small homogeneous network of workstations. The main problem with our previous technique was its high consumption of resources. Furthermore, we found that under a saturated computational environment increasing the level of replication actually decreased performance as replicates competed with each other for the same finite pool of resources.

In this paper, we extend our work by introducing the concept of dormant actors. Using dormant actors, programmers can increase the fault-tolerance characteristics of their applications with very little overhead. We tested this new approach using a DNA/Protein sequence comparison application in two different environments. The first environment consisted of a local-area dedicated network of Intel 80486 machines. The second environment consisted of a wide-area, heterogeneous environment comprised of three autonomous sites: the National Center for Supercomputing Applications (NCSA), the University of Virginia (UVa) and the NASA Langley Research Center (LaRC).

The remainder of the paper is organized as follows. We first present a brief overview of the Mentat system and its execution model (Section 2). We describe the interface for specifying a fault-tolerance policy and discuss the protocol used to transparently replicate both active and dormant actors. We also illustrate the mapping of Mentat source code to actual run-time implementations (Section 3). We then describe the DNA/Protein sequence comparison application (Section 4) and analyze its performance and recovery characteristics using several fault-tolerance policies (Section 5). Finally, we discuss related work (Section 6) and conclude (Section 7).

## 2. Mentat

Mentat [9] is a high performance, object-oriented parallel processing system. There are two primary aspects of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. MPL is an object-oriented programming language based on C++. The granule of computation is the Mentat class member function. The programmer is responsible for identifying those object classes whose member functions are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used like C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment.

The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

Mentat classes are denoted by the inclusion of the keyword `mentat` in the class definition. Mentat classes may be qualified as either `regular` or `persistent`. Instances of regular Mentat classes do not maintain state between invocation, thus the implementation may create a new instance to handle every member function invocation. Persistent Mentat classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data or that require persistent semantics.

Mentat objects are active entities and possess a name, a thread of control, and their own address space. From the user's perspective, Mentat objects communicate via asynchronous member function invocation.

### 2.1 Mentat execution model

The Mentat execution model is based on the macro data-flow model (MDF [10]) an extension of the pure data-flow model. MDF is one of several large grain data flow models [1][3] that expand on traditional data flow [22]. The salient features of MDF are that it incorporates the notion of state, adds the ability to dynamically create graphs, and provides coarse grained actors. In MDF, actors with states are said to be persistent actors while stateless actors are called regular actors. Persistent actors that share state map directly onto member functions of a persistent Mentat class. Similarly regular actors map onto member functions of a regular Mentat class. For the rest of the paper, we may thus use a class name and a member function to denote actors.

The Mentat run-time system implements a virtual macro data-flow machine that transparently constructs program data-flow graphs, schedules actors on processors, and manages communication and synchronization. The token matching unit (TMU) implements the pure data-flow subset of the MDF model and is responsible for matching tokens and for enabling an actor when all its tokens are present. When an actor is enabled, the TMU calls on the instantiation manager (IM) to schedule a regular object for the actor. The TMU then forwards the tokens to the object so that the actor may fire, i.e. execute. To distribute the workload associated with regular actors and scheduling, there is one TMU and one IM per host in a Mentat system. Note that by delaying the instantiation of regular objects until the tokens are matched, the Mentat

scheduler can make better placement decision by having access to up-to-date information (e.g. load).

The macro data-flow underpinning of Mentat is completely transparent to end users. In Figure 1, we illustrate the mapping from a sample MPL source code to its execution. At run-time, calls made to object functions (actors) are transformed into a macro data-flow graph that is then acted on by the run-time system to deliver the proper arguments (tokens) to the appropriate object's function. The mechanism for building graphs and detecting data dependence is fully described in [10][11].

The graph (Figure 1b) is constructed at run-time and maps onto the implementation as follows :

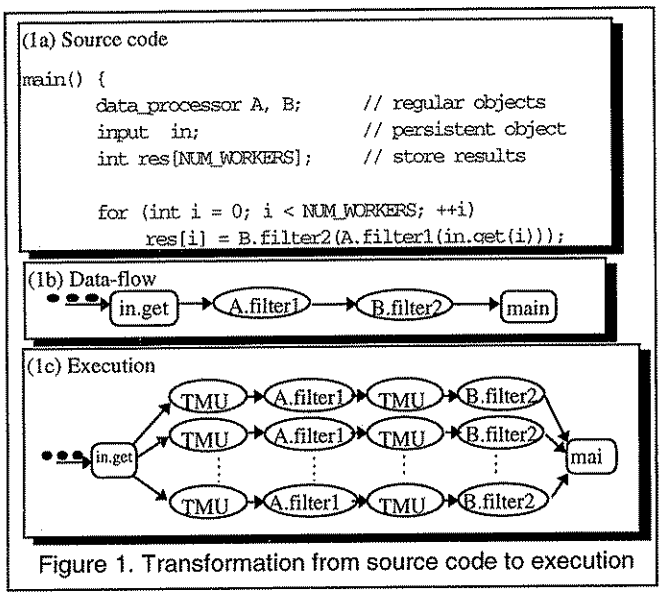


Figure 1. Transformation from source code to execution

For each iteration,

- A message that contains the token *i* and a copy of the data-flow graph is sent to the persistent actor `in.get`. Tokens carry a computation tag that uniquely identifies an actor and the number of tokens required to enable that actor. Computation tags are equivalent to token colors in the data-flow litterature.
- `in.get` then sends a message containing its output token to a TMU along with the program graph. The TMU is selected by a hash function on the computation tag for actor `A.filter1`.
- Upon receiving the token, the TMU enables the actor `A.filter1` and then makes a scheduling request to the Mentat scheduler. The scheduler instantiates object A and returns A's physical address (host id and port number).
- The TMU forwards the token and program graph to object A.
- Object A executes function filter1().
- When `A.filter1` finishes, it must send the result along all outgoing arcs in the data-flow graph representation. Since B is also a regular object, A forwards its output token to the TMU handling `B.filter2`. Again, the TMU is selected by hashing on the computation tag for `B.filter2`.
- `B.filter2` is handled similarly with the end results sent back to the main program.

This simple example represents a pipeline computation that consists of three stages. In a traditional pipeline, the rate of output is determined by the slowest stage. For example, if `in.get` produces an output token every second, `filter1` takes one second, and `filter2` five seconds, then we will get an output every five second.

Mentat exploits the fact that the two filters are regular objects and instantiates a new copy to service each request. The fully elaborated execution (Figure 1c), represents a pipeline with multiple functional units. Thus, throughput is no longer limited by the slowest stage and we can obtain an average throughput rate of one output per second per iteration once the pipe is full.

Another feature of Mentat is that results from function invocations do not return to the caller unless they are needed. In the pipeline example above, the output from the first filter is never returned to the main program and is automatically forwarded to the second filter.

## 3. Supporting fault-tolerance

In a previous paper [18] we described modifications to the Mentat run-time system for supporting an *active* replication policy -- actors fired as soon as they were enabled -- and noted the resulting inefficient use of CPU resources. We now extend our earlier work by introducing the concept of *dormant* actors. Unlike active actors, dormant actors do not fire right away when enabled but wait until they detect the failure of an ongoing computation. Dormant actors serve a similar function to backup replicates in a primary/backup replication scheme. The main difference here is that there is no need for explicit synchronization protocols or state transfer primitives.

The combination of active and dormant actors provide users with the mechanisms for trading-off fault-tolerance, resource consumption and performance. By using dormant actors, application designers can reduce their consumption of CPU while increasing the level of fault-tolerance. However, there is a trade-off between resource consumption and performance. In the presence of failures, active actors provide instantaneous recovery, while dormant actors will take longer to recover as they must

first detect failure and then restart the computation from the beginning.

The replication strategies presented in this paper only apply to program graphs composed of regular actors. If the program graph contains persistent actors, then they will not be replicated and present single failure points (transparently replicating persistent actors is much more difficult and will be addressed in future research). This is not a major problem as applications that use regular objects typically exhibit a master/worker or a pipeline structure. Thus, persistent actors in a graph usually bracket a composition of regular actors, and it is that subset of the program graph that we are replicating. In fact, the main program is usually both the initiator of a parallel computation and its recipient.

Furthermore, we assume that both user and system objects (IM and TMU) are correctly implemented (i.e. no logical errors) and fail only when the host on which they are placed fails. We also assume that communication links do not fail.

### 3.1 Specifying the fault-tolerance policy

One of our key goals is to provide application writers with an easy-to-use mechanism for specifying fault-tolerance. The interface for selecting a replication policy simply consists of creating an instance of the class `ft_policy`, setting the number of active and dormant replicates, and specifying a ping value. The policy is then valid within the scope of the declaration. This gives users the flexibility of tailoring their fault-tolerance policies to different parts of their code.

The interface to `ft_policy` is shown below:

```
(1) class ft_policy {
(2) public:
(3)   ft_policy (int active,
          int dormants=0, int ping=0);
(4)   ~ft_policy();
(5) };
```

The following Mentat code fragment illustrates the use of `ft_policy`:

```
(1) main() {
(2)   int a;
(3)   regular_class Y;
(4)
(5)   // 2 active only
(6)   ft_policy replication(2,0);
(7)
(8)   a = Y.op2(3);
(9)   printf("a = %d\n", a);
(10)
(11)  { // new scope, 1 active,
(12)    // 3 dormants, 30 sec ping
(13)    ft_policy replication2(1,3,30);
(14)    int b;
(15)    regular_class2 D, F;
(16)
(17)    b = D.op1( F.op2(4));
(18)  }
(19)  //policy restored to 2 active
(20) }
```

Line (6) specifies a replication policy of two active. The run-time system transparently replicates `Y.op2(3)` twice. On line 13 the user has selected another policy that is enforced within the scope of the declaration (lines 13-17). The original policy is automatically restored when the flow of control exits scope (line 18).

The fault-tolerance policy only applies to objects that are directly invoked by the caller -- objects Y, D, F in this example -- and does not propagate to called objects. If Y.op2() invokes another object Z before returning its result, Z will not be replicated by default.

### 3.2 Active actor replication

Before describing the algorithm for dormant actor replication, we first review the implementation of active actor replication. Both replication methods use the same underlying mechanisms for transparently replicating regular actors.

To implement replication, the Mentat run-time system duplicates tokens and sends them to distinct TMUs. The additional TMUs are selected by using a parameterized hash function on the computation tag that, given a number x, returns $TMU_x$ and guarantees that $TMU_0..TMU_{x-1}$ are all on distinct hosts.

Using the same pipeline example as before, we show the process of transforming MPL source code to a replicated execution graph (Figure 2).

For each iteration,

- A message that contains the token *i*, a copy of the data-flow graph, and the fault-tolerance policy is sent to the persistent actor `in.get`. Note that the token is not replicated since `in.get` is persistent.
- `in.get` then sends a message containing its output token to k distinct TMUs. The level of replication k is extracted from the fault-tolerance policy. In this particular case, k = 2.

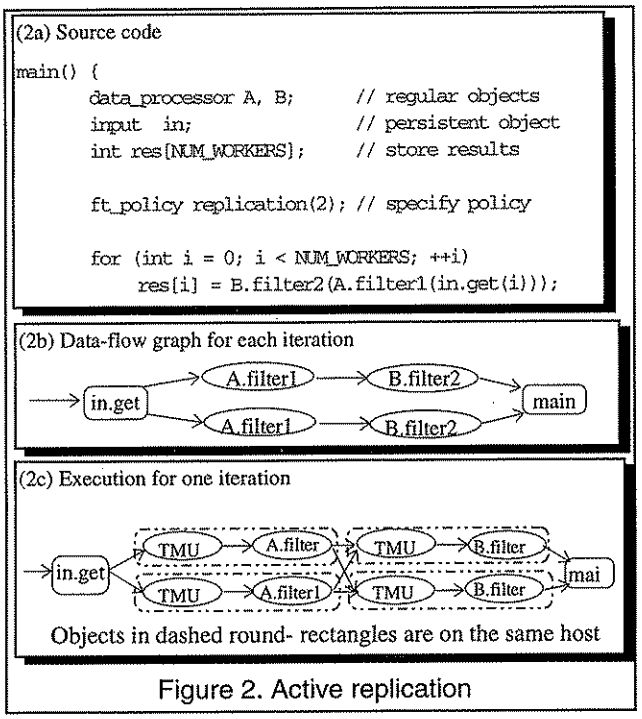



Figure 2. Active replication

- Upon receiving the token, each TMU independently enables the actor `A.filter1` and makes a scheduling request. The scheduler instantiates object A on the same host as the TMU and returns A's physical address (host id and port number). There are now two copies of A.
- Each TMU forwards the token, the program graph, and the fault-tolerance policy to its respective A.
- Each A executes function filter1().
- When each `A.filter1` finishes, it duplicates the result token and sends it to the two TMUs handling `B.filter2`.
- Each TMU receives a total of two tokens but discards one. At this point, `B.filter2` is handled normally with the end results sent back to the main program.

In order to make a k-replicated object tolerate k - 1 host failures, we must ensure that they are placed on distinct hosts. This is accomplished by scheduling regular objects on the same host as the TMUs handling them. Since the TMUs participating in replicating a regular actor are guaranteed to be on distinct hosts (assuming that the number of hosts is at least k), it follows that the objects themselves will be on different hosts.

An arbitrary graph composed of k-replicated actors can tolerate k - 1 host failures. As long as one of each of the individual replicated actor succeeds, there will be a path along which the overall computation may complete. Consider the execution graph in Figure 2 where k = 2. To prevent `main` from receiving its result would require at least two failures, otherwise there would be a path from `in.get` to `main`.

### 3.3 Algorithm for dormant actor replication

The implementation of active replication was a relatively simple extension to the Mentat run-time system. Its simplicity stems from the fact that TMUs need only make local decisions and do not coordinate their operations. Implementing dormant replication is much more complex as it requires TMUs to cooperate amongst themselves.

The mechanisms for replicating dormant actors and sending tokens to multiple destinations are the same ones used in active replication (Figure 2). To aid in the description of the implementation, we distinguish between active and dormant TMUs, i.e. TMUs that handle active or dormant actors respectively. Note that in the following discussion TMUs are active or dormant with respect to a single computation.

The computation tag and the fault-tolerance policy determine the set of participating TMUs. The replication level, k, is the number of active actors plus the number of dormant actors. We use the parameterized hash function to order the set of participating TMUs ($TMU_0..TMU_{k-1}$). By convention, $TMU_0..TMU_{a-1}$ are active and $TMU_a..TMU_{k-1}$ are dormant.

To simplify the description of the algorithm, we assume that there is only one active and d dormant TMUs. Recall that an active TMU schedules an actor as soon as it is enabled. The task of the dormant TMUs is to monitor the progress of the scheduled actor and restart it when a failure is detected. To minimize the number of ping messages, the dormant TMUs do not all monitor the active actor. Instead, the dormant TMUs elect a leader that is responsible for monitoring the scheduled actor. The leader assumes failure when the actor does not respond within the specified ping interval $P_0$.

The dormant leader also sends a "pulse" message every $P_0$ seconds to each of its followers. When a follower does not receive a pulse within a time interval $P_i$, it elects itself leader. The value $P_i$ is set according to the rank of each follower and is a multiple of $P_0$. By staggering the values of $P_i$, we attempt to elect a unique leader. In the event that multiple leaders are elected, they will start monitoring the same object, which will increase the number of ping messages sent to the object. In the unlikely worse case scenario, all followers turn leaders and reschedule the same failed computation. While this may waste resources it does not affect the correctness of the computation. The net result is that we are back to an active replication policy.

We now describe the steps taken by a TMU when it has matched the tokens for an actor:

**TMU**:

(T1) TMU determines whether it is active or dormant. Since we have assumed a policy of k replication (one active and d dormants), $TMU_0$ is active and $TMU_1..TMU_{k-1}$ are dormant.

**Active**:

(A2) Schedules regular object and forwards physical address to all dormant TMUs.

**Dormant**:

(D2) TMU determines whether it is a leader or a follower. We have assumed only one active TMU, thus the leader is $TMU_1$ and the followers are $TMU_2..TMU_{k-1}$.

**Dormant Leader**:

(L3) If the dormant leader does not receive a physical address within $P_0$ seconds, it changes status from dormant to active and reschedules the computation (A2).

(L4) Pings instantiated object every $P_0$ seconds using physical address received from active TMU (A2).

(L5) If object does not reply within $P_0$ seconds, it changes the status to active and reschedules the computation (A2).

(L6) Sends pulse to followers every $P_0$ seconds.

**Dormant Follower**:

(F3) Determines rank $F_i$ within the set of followers. $F_0$ is the highest ranking follower and is initially $TMU_2$.

(F4) If no pulse is received within $P_0 * (i+1)$ seconds, it elects itself leader and sends a pulse message to lower ranked followers.

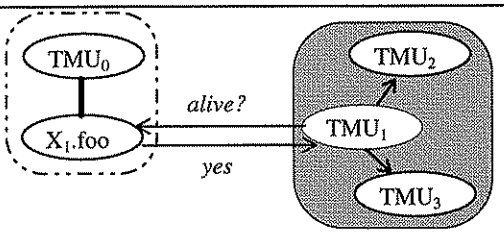
We also present the protocol for regular objects:

**Regular object**:

(R1) Asynchronously responds to pings sent by dormant leader TMU.

(R2) Upon completing a computation, it notifies dormant TMUs so that they can stop tracking this computation.

Using dormant replicates in conjunction with active replicates provides the same guarantees as active replication alone since all dormant TMUs have the potential of changing their status to active TMUs. Indeed, a dormant TMU with a ping value set to 0 is essentially equivalent to an active TMU. If the active actor fails, steps L3 and L5 guarantee that a dormant leader TMU turns active and reschedules the actor computation. If the leader TMU fails, step F4 guarantees that at least one of the dormant follower TMU elects itself leader within a bounded time interval.




$TMU_0$ is the active TMU. $TMU_1$, $TMU_2$, $TMU_3$ are dormant TMUs. $TMU_1$ is the dormant leader and monitors the health of $X_1$ by periodically sending ping messages. $TMU_1$ also sends "pulse" messages to its followers.

Figure 3. Active and dormant replication

## 4. DNA/Protein sequence comparison

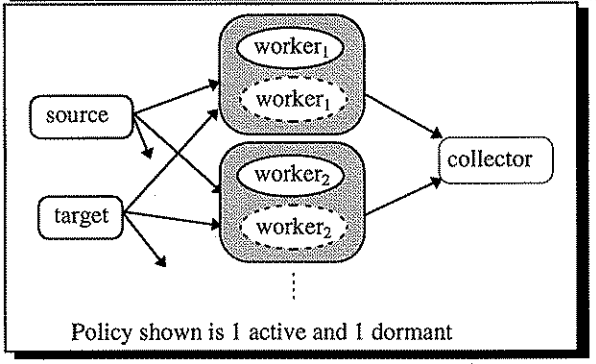
Our test application compares two protein or DNA sequence libraries. Each library contains one or more sequences, each of which consists of a sequence name and a variable-length string of characters (also known as residues). Each sequence in the first library, called the source library, is compared against each sequence in the second, called the target library. For each sequence comparison, a score is generated reflecting sequence

```
// Mentat source code
// This is the heart of the application
//
docomplib (genome_lib src, collector_class collector)
{
        worker      sw;          // Smith-Watermann
        chunk *tgt_chunks;
        chunk       source_chunk;

        ft_policy  replication(active, dormant, ping);

        tgt_chunk = tgt.get_chunk(num_workers);
        src_chunk = src.get_chunk();

        for (i=0; i < num_workers; ++i) {
           collector.register(
              sw.compare(src_chunk,tgt_chunk[i]));
```




Policy shown is 1 active and 1 dormant

Figure 4. Mentat source code and program graph

commonality using the Smith-Waterman [21] algorithm.

Once all of the scores have been generated for all sequence comparisons, they are sorted and statistical information is generated. An important attribute of the algorithm is that all comparisons are independent of one another and if many sequences are to be compared, they can be compared in any order. This parallelism is easy to exploit.

The program consists of several Mentat objects, the source and target libraries, a collector object that monitors the progress of the comparisons, a recorder object to gather statistical information, and finally, workers to perform the actual comparisons.

In Figure 4, we show the heart of the application. For each worker, we compare the source library with a subset of the target library. The results of the comparisons are then forwarded to the collector object. Note that the workers are the only objects that are instances of regular Mentat classes. All other objects are persistent.

## 5. Experiment

We ran our experiments on two very different environments. The first consisted of a dedicated network of 26 33MHz 80486 PCs running Linux connected by Ethernet. This controlled environment allowed us to analyze our data with a high degree of confidence. The second environment consisted of shared computing resources at three sites: the SGI Power Challenges at NCSA, high-end SGI workstations at NASA Langley and several SUN workstations running both SunOS & Solaris at the University of Virginia (UVa).

To minimize the impact on other users, we ran our experiments over the course of several nights. Even then, the machines were often heavily utilized (especially the Power Challenges at NCSA and the compute servers at UVa) and the load varied widely. In addition, we limited ourselves to using at most 5 processors of the 16-processor Power Challenges and 2 processors of the 4-processor Solaris compute servers at UVa. Table 1 lists the resources used at each site.

| MACHINE | #MACHINES | PROCS. USED |
|---|---|---|
| *Dedicated Homogeneous Testbed* | | |
| Intel 80486-33 (Linux) | 26 | 1 / 1 |
| *Shared Heterogeneous Testbed* | | |
| SGI Power Challenge Array | 2 | 5 / 16 |
| SGI Indigo | 8 | 2 / 2 |
| Sparc 20 (SunOS) | 2 | 1 / 1 |
| Sparc 10 (Solaris) | 4 | 2 / 4 |
| Sparc 10 (Solaris) | 1 | 2 / 2 |

Table 1. Testbeds configuration

To determine the relative strengths and weaknesses of various replication strategies, we tested their performance in both environments with the DNA/protein sequence comparison application developed at the University of Virginia. Performance is given in terms of millions of matrix entries per second (MEPS) and is a standard benchmark in the biochemistry community. Each matrix entry corresponds to one residue from one sequence compared against one residue from another sequence.

We explored several replication strategies: 1 active, 1 active and 1 dormant (30 second ping interval), and 2 active. For the last two policies, we tested both the 0 and 1 host failure case. For all configurations, we varied the number of workers from 8 to 24.

Baseline sequential times and their corresponding MEPS are given in Table 2. Note that we used different libraries for the testbeds. The MEPS rating for the Intel platform corresponds to 905,079,926 residue comparisons whereas performance for the other platforms were computed with 3,087,930,284 residue comparisons. This was a practical decision as the Intel based machines are relatively slower.

| MACHINE | CPU TIME (SEC) | MEPS |
|---|---|---|
| SGI Power Challenge Array | 1455 | 2.12 |
| SGI Indigo | 1103 | 2.80 |
| Sparc 20 (SunOS) | 1987 | 1.55 |
| Sparc 10 (Solaris) | 1887 | 1.64 |
| Intel 80486-33 (Linux) | 2985 | 0.30 |

Table 2. Relative performance of each class of machine using sequential version of the application

In Table 3 we show the average wall clock time elapsed for the baseline case (1 active, no dormant) with no failures on both testbeds.

| WORKERS | MIN TIME (S) | MAX TIME (S) | MEAN TIME(S) ± STDDEV | MEPS |
|---|---|---|---|---|
| *Dedicated Homogeneous Testbed* | | | | |
| 8 | 368 | 424 | $379 \pm 18$ | 2.39 |
| 16 | 217 | 241 | $225 \pm 9$ | 4.02 |
| 24 | 166 | 207 | $176 \pm 13$ | 5.13 |
| *Shared Heterogeneous Testbed* | | | | |
| 8 | 276 | 343 | $299 \pm 27$ | 10.33 |
| 16 | 160 | 283 | $199 \pm 31$ | 15.50 |
| 24 | 164 | 2678 | $212 \pm 37$ | 14.57 |

Table 3. Performance with no failures

The figures obtained on the shared heterogeneous network exhibit a wide variance due to unpredictable resource availability and the vastly different processing capabilities of each platform. The performance in the homogeneous environment was good. However the focus

of this paper is not on the absolute performance of the application but rather on exploring the costs associated with using various fault-tolerance replication policies.

### 5.1 Execution in failure-free mode

We found that the additional performance overhead incurred using 1 dormant actor is very low as compared to the non-fault tolerant base case (Table 4). In all cases, the difference is within 6%. This should be expected since a dormant actor does not fire unless a fault is detected. In the cases of 2 active replicates in a dedicated homogenous environment, we find that as the number of workers increases, performance degradation occurs.

This performance penalty is a symptom of a saturated computational environment. As the number of workers increases, the chances of multiple objects being scheduled on the same host also increases. With our 26 host homogeneous test environment, using 2 active replicates with more than 13 workers guarantees that at least two workers will be scheduled on the same host.

| Workers & Policy | Min Time(s) | Max Time(s) | Mean Time(s) | Meps |
|---|---|---|---|---|
| *Dedicated Homogeneous Testbed* | | | | |
| 8 (1,0) | 368 | 424 | 379 ± 18 | 2.39 |
| 8 (1,1) | 385 | 434 | 401 ± 16 | 2.26 |
| 8 (2,0) | 376 | 398 | 384 ± 6 | 2.36 |
| 16 (1,0) | 217 | 241 | 225 ± 9 | 4.02 |
| 16 (1,1) | 215 | 260 | 227 ± 13 | 3.99 |
| 16 (2,0) | 380 | 414 | 387 ± 9 | 2.34 |
| 24 (1,0) | 166 | 207 | 176 ± 13 | 5.13 |
| 24 (1,1) | 170 | 207 | 176 ± 13 | 5.14 |
| 24 (2,0) | 279 | 452 | 314 ± 57 | 2.88 |
| *Shared Heterogeneous Testbed* | | | | |
| 8 (1,0) | 276 | 343 | 299 ± 27 | 10.33 |
| 8 (1,1) | 287 | 312 | 299 ± 9 | 10.32 |
| 8 (2,0) | 274 | 331 | 297 ± 20 | 10.40 |
| 16 (1,0) | 160 | 283 | 199 ± 31 | 15.50 |
| 16 (1,1) | 163 | 248 | 211 ± 31 | 14.62 |
| 16 (2,0) | 177 | 380 | 280 ± 65 | 11.04 |
| 24 (1,0) | 164 | 268 | 212 ± 37 | 14.57 |
| 24 (1,1) | 184 | 218 | 201 ± 13 | 15.33 |
| 24 (2,0) | 157 | 235 | 211 ± 27 | 14.65 |

Table 4. Effects of using active and dormant replicates for fault tolerance (Policy in parenthesis denote the level of active and dormant replication)

We do not observe any performance advantage to using multiple active replicates in our heterogeneous testbed, however because of the high variance within the heterogeneous network, we cannot draw any definite conclusion.

### 5.2 Execution with host failure

Table 5 presents the effects of a simulated 1 host failure in a variety of configurations. In order to simulate host failure, we terminated all of the Mentat objects on a single host. Since our replication strategy does not include persistent objects, the main program, collector, and libraries are not fault tolerant. To ensure that these persistent objects did not fail, they were all placed on the same host and this host was never a candidate for a simulated host failure. The simulated host failure occurred 45 seconds after the application began.

As expected the system successfully recovered from a single host failure using 2 active or 1 active and 1 dormant replicate. The main difference between these two fault tolerant strategies is their recovery characteristics. In the dormant replicate case the completion time depends on the time it takes to detect failure, the amount of work lost, where the object is rescheduled, and the cost of restarting the object. Active replication essentially provides instantaneous recovery. As can be seen though, active replication is not an efficient solution in a saturated computational environment.

| Workers & Policy | Min Time(s) | Max Time(s) | Mean Time(s) | Meps |
|---|---|---|---|---|
| *Dedicated Homogeneous Testbed* | | | | |
| 8 (1,1) | 463 | 466 | 408 ± 2 | 2.22 |
| 8 (2,0) | 377 | 408 | 394 ± 13 | 2.30 |
| 16 (1,1) | 294 | 364 | 321 ± 32 | 2.82 |
| 16 (2,0) | 365 | 406 | 380 ± 11 | 2.38 |
| 24 (1,1) | 244 | 313 | 280 ± 28 | 3.23 |
| 24 (2,0) | 277 | 396 | 314 ± 56 | 2.88 |
| *Shared Heterogeneous Testbed* | | | | |
| 8 (1,1) | 394 | 469 | 432 ± 53 | 7.15 |
| 8 (2,0) | 282 | 356 | 309 ± 23 | 9.99 |
| 16 (1,1) | 215 | 280 | 248 ± 23 | 12.45 |
| 16 (2,0) | 234 | 407 | 295 ± 69 | 10.46 |
| 24 (1,1) | 189 | 270 | 212 ± 25 | 14.57 |
| 24 (2,0) | 195 | 376 | 259 ± 58 | 11.93 |

Table 5. Effects of using active and dormant replicates for fault tolerance with one host failure (Policy in parenthesis denote the level of active and dormant replication)

The data suggests that using dormant replication, fault tolerance can be obtained without negatively impacting resource consumption or application performance significantly. The advantages of dormant replication over active replication is its more efficient use of resources. In failure free mode, there is no significant performance penalty (less than 6%). Theoretically, the potential benefit to active replication is its instant recovery characteristic. Unfortunately, this is only true when the number of available resources is sufficiently high. Otherwise, in a

saturated environment, replicated workers may compete with each other for the same available resources; thus negatively impacting performance. In a highly shared environment such as our heterogeneous testbed, active replication may consume resources that other users could have used.

## 6. Related work

While there is a rich literature in fault-tolerance for distributed and real-time systems, there has been much less done in the area of fault-tolerant parallel processing systems. Most of the work has concentrated on fault-tolerant hardware, e.g. fault-tolerant networks and system reconfiguration after a fault. There has been some though, for example, FT-Linda [4], PLinda [14], Orca [15], Calypso [5], and Fail-safe PVM [16]. These systems use a combination of well known mechanisms such as replication, transactions, message logging, or checkpoints and rollbacks to provide fault-tolerance.

Mentat differs from these systems in that its underlying computational model is based on data-flow. Moreover, Mentat and macro data-flow (MDF) differ from other large grained data-flow systems such as Paralex [2], CDF [3], HeNCE [6], and Code/Rope [8] in that program graphs in MDF are dynamic and generated at runtime. In Mentat, the program graphs are generated by the compiler and run-time system, unlike [2][6][8], where the programmer is responsible for generating the program graphs using a graphical interface. Paralex uses the ISIS toolkit [7] to provide fault-tolerance and to our knowledge is one of the few data-flow parallel processing system that provides direct support for fault-tolerance. ATAMM [19] is another but its application domain is embedded real-time systems.

The techniques described in this paper are easily applicable to any coarse grain data-flow systems. Replication is not novel and is a well understood concept even in the general case of objects/processes with state [17][20]. Our work differs in that we have focused on the special case, i.e. stateless objects, and have exploited their idempotent nature to provide easy-to-use fault-tolerance. The combination of active and dormant replication form a very special kind of replicated group that does not require group communication protocols. We have tailored our group abstraction to work with objects that do not maintain state and this considerably simplifies our design.

## 7. Conclusion

Wide-area parallel processing systems will soon be available to researchers to solve a range of problems. It is certain that host failures and other faults will be an every day occurrence in these systems. Unfortunately contemporary parallel processing systems were not constructed with fault-tolerance as a design objective.

The data-flow model offers hope. Its functional nature, which makes it so amenable to parallel processing, also facilitates straight-forward fault-tolerant implementations. It is the combination of ease of parallelization and fault-tolerance that we feel will increase the importance of the model in the future and lead to the widespread use of functional components.

We have demonstrated the mechanisms by which the Mentat run-time system transparently replicates data-flow actors to implement a user defined fault-tolerance policy. We have also introduced the concept of dormant actors. Unlike active actors, dormant actors do not execute as soon as they are enabled but wait until failure occurs before firing their computation. We have shown that by using dormant actors, programmers can add fault-tolerance to their code without a significant impact on resource consumption, and in failure-free mode, without paying a high performance penalty (less than 6% in all cases).

We have presented results from a production biochemistry application running over both a local, dedicated network of Intel based machine and a wide-area heterogeneous environment consisting of shared resources from three sites: NCSA, NASA Langley and the University of Virginia. The wide-area heterogeneous testbed represents an early prototype of the envisioned nation-wide Legion system.

Future work consists of incorporating persistent actors in our replication strategies and on extending our replication mechanism to handle network partitioning.

## 8. Acknowledgments

We would like to thank the National Center for Supercomputing Application and the NASA Langley Research Center for the use of their resources.

## 9. References

[1] T. Agerwala and Arvind, "Data Flow Systems," IEEE Computer, vol. 15, no. 2, pp. 10-13, February, 1982.

[2] O. Babaoglu et. al., "Paralex: An Environment for Parallel Programming in Distributed Systems," Technical Report UBLCS-92-4, Laboratory for Computer Science, University of Bologna, Oct. 1992.

[3] R. F. Babb, "Parallel Processing with Large-Grain Data Flow Techniques," IEEE Computer, pp. 55-61, July, 1984.

[4] D. Bakken and R. Schlichting, "Supporting fault-tolerant parallel programming in Linda," Technical Report TR93-18, The University of Arizona, 1993.

[5] A. Baratloo, P. Dasgupta and Z. M. Kedem, "CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms," Proceedings of the Fourth IEEE International Symposium on High

Performance Distributed Computing, pp. 122-129, Washington, D.C., August 1995.

[6] A. Beguelin et al., "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing," Proceedings SHPCC-92, pp. 129-136, Williamsburg, VA, May, 1992.

[7] K. Birman et. al., "Implementing Fault-Tolerant Distributed Objects," IEEE Transactions on Software Engineering, Vol. SE-11, No. 6, June 1985.

[8] J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, pp. 111-120, vol. 16, no. 2, Feb., 1990.

[9] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," IEEE Computer, pp. 39-51, May, 1993.

[10] A. S. Grimshaw, "The Mentat Computation Model - Data-Driven Support for Dynamic Object-Oriented Parallel Processing," Computer Science Technical Report, CS-93-30, University of Virginia, May, 1993.

[11] A. S. Grimshaw, J. B. Weissman and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing", To appear in the ACM Transactions of Computer Systems.

[12] A. S. Grimshaw, A. Nguyen-Tuong and W. A. Wulf, "Campus-Wide Computing: Early Results using Legion at the University of Virginia", Technical Report CS-95-19, Department of Computer Science, University of Virginia, 1995.

[13] A. S. Grimshaw et. al., "Legion: The Next Logical Step Toward a Natiowide Virtual Computer," Computer Science Technical Report, CS-94-21, June 8, 1994.

[14] K. Jeong and D. Shasha, "Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda," Proceedings of the 13th Symposium on Reliable Distributed Systems, 1994.

[15] M. Kaashoek et. al., "Transparent fault-tolerance in parallel Orca programs," Symposium on Experiences with Distributed and Multiprocessor Systems, 1992.

[16] J. Leon, A. L. Fisher, P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery", Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, PA, February 1993.

[17] M.C. Little and S.K. Shrivastava, "Replicated K-Resilient Objects in Arjuna", Proceedings of the 1st IEEE Workshop on the Management of Replicated Data, Houston, pp. 53-58, November 1990.

[18] A. Nguyen-Tuong, A. S. Grimshaw, J. F. Karpovich, "Fault-Tolerance via Replication in Coarse Grain Data-Flow", Proceedings of the International Workshop on Parallel Symbolic Languages and System, Lecture Notes in Computer Science 1068, Springer-Verlag, pp. 250-265, 1996.

[19] R. A. Obando and J. W. Stoughton, "A Performance Prediction Model for a Fault-Tolerant Computer During Recovery and Restoration," NASA Contractor Report 195074, NASA Langley Research Center, Virginia, February 1995.

[20] D. Powell, "Delta-4: A Generic Architecture for Dependable Distributed Computing," ESPRIT project 2252 Research Report, Springer Verlag, 1991.

[21] F. Smith and M. S. Waterman, "Identification of common molecular subsequences", J. Mol. Biol., 147, pp. 195-197, 1981.

[22] H. Veen, "Dataflow Machine Architecture," ACM Computing Surveys, pp. 365-396, vol. 18, no. 4, December, 1986.