Fault Tolerance in Coarse Grain Data Flow[1]

Anh Nguyen-Tuong, Andrew S. Grimshaw and John F. Karpovich

University of Virginia, Thornton Hall, Department of Computer Science

Charlottesville, VA 22903

Email: {nguyen | grimshaw | karp}@virginia.edu

URL: http://www.cs.virginia.edu/~an7s, http://www.cs.virginia.edu/~grimshaw, http://www.cs.virginia.edu/~jfk3w

1. Introduction

Powerful virtual machines consisting of thousands of hosts, both workstations and MPP's, connected by fast wide-area networks are becoming a reality. The objective of the Legion project at the University of Virginia[2] is to construct such a machine — a virtual computer capable of supporting distributed and parallel transnational applications within a single logical namespace. A small-scale prototype, the campus-wide virtual computer (CWVC) [11], is operational and is currently configured with over 80 workstations and an IBM SP-2. Already we are experiencing frequent host failures in the CWVC. On the scale of the envisioned nation-wide virtual computer, host failures will be a fact of life. Therefore, fault-tolerance — both at the system and application level — will be necessary to exploit the potential of such a large system.

In addition to the usual arguments for achieving a high degree of parallelism, the data-flow model [1][16] presents natural advantages for fault-tolerance. Recall that data-flow computations are modelled 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 distinguishing feature of actors in terms of fault tolerance is their functional nature: an actor presented with the same tokens will produce the same result. Thus, fault-tolerance for actors can be easily achieved in two non-mutually exclusive ways. The first is to replicate the actor *k* times

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

2. Information about Legion can be obtained via the world-wide web at URL: `http://www.cs.virginia.edu/~legion`

and use the first available result, discarding later arriving results. The second is to detect failure and then restart the failed actor.

In this paper we show that building fault-tolerant applications need not be hard if programmers are provided with the right tools and abstractions. We illustrate our point with Mentat[3] [7][10], an object-oriented, data-flow based, parallel processing system. Using Mentat, we demonstrate two orthogonal methods of providing application fault-tolerance. The first method provides transparent replication of actors and requires modification to the existing Mentat run-time system. Providing direct support for replicating actors enables the programmer to easily build fault-tolerant applications regardless of the complexity of their data-flow graph representation. The second method — the checkboard method — is applicable to applications that contain independent and restartable computations such as "bag of tasks", Monte Carlo's, and pipelines, and involves some simple restructuring of code. While these methods are presented separately, they could in fact be combined. For both methods, we present experimental data to illustrate the trade-offs between fault-tolerance, performance and resource consumption. Resource consumption is often neglected but must be taken into account as we move towards an environment in which machines are often shared.

The rest of this paper is organized as follows: in Section 2, we present a brief overview of Mentat and its execution model. In Section 3, we present an extension to the Mentat run-time system for supporting the transparent replication of actors while the checkboard method is described in Section 4. Section 5 contains Mentat pseudo-code for three versions of a synthetic pipeline application — a non fault-tolerant version, a version exploiting extensions to Mentat to provide transparent replication of actors, and a version using the checkboard method. In Section 6 and Section 7 we present our experimental data and analyze the results.

2. Mentat

Mentat 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

3. For additional publications and information about Mentat, see the URL: `http://www.cs.virginia.edu/~mentat`

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. By splitting the responsibility between the compiler and the programmer, we exploit the strengths and avoid the weaknesses of each.

Mentat classes are denoted by the inclusion of the keyword “mentat” in the class definition, as in the mentat class `sw_worker` shown in Figure 1. The keyword `mentat` tells the compiler that the member functions of the class are worth executing in parallel. Mentat classes may be defined as either *persistent* or *regular*. Instances of regular Mentat classes are logically stateless, 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.

Figure 1 Mentat class definition

```
regular mentat class sw_worker {
// private data and function members
public:
       result_list*compare(sequence*,libstruct*,paramstruct);
};
```

The keyword “mentat” tells the compiler to treat instances of this class differently. The “regular” modifier indicates that instances of this class are stateless, i.e., they are pure functions.

A Mentat object is an instance of a Mentat class, and possesses a name, a thread of control, and an address space. Because Mentat objects each have their own address space they are address space disjoint. Therefore, *all* communication between Mentat objects and between Mentat objects and main programs is via member function invocation.

2.1. Mentat Execution Model

Mentat is based on the macro data-flow model (MDF [8]), an extension of the pure data-flow model. MDF is one of several large grain data flow models [2][5][6] that expand on traditional data flow. 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 and regular actors are implemented by persistent and regular Mentat objects).

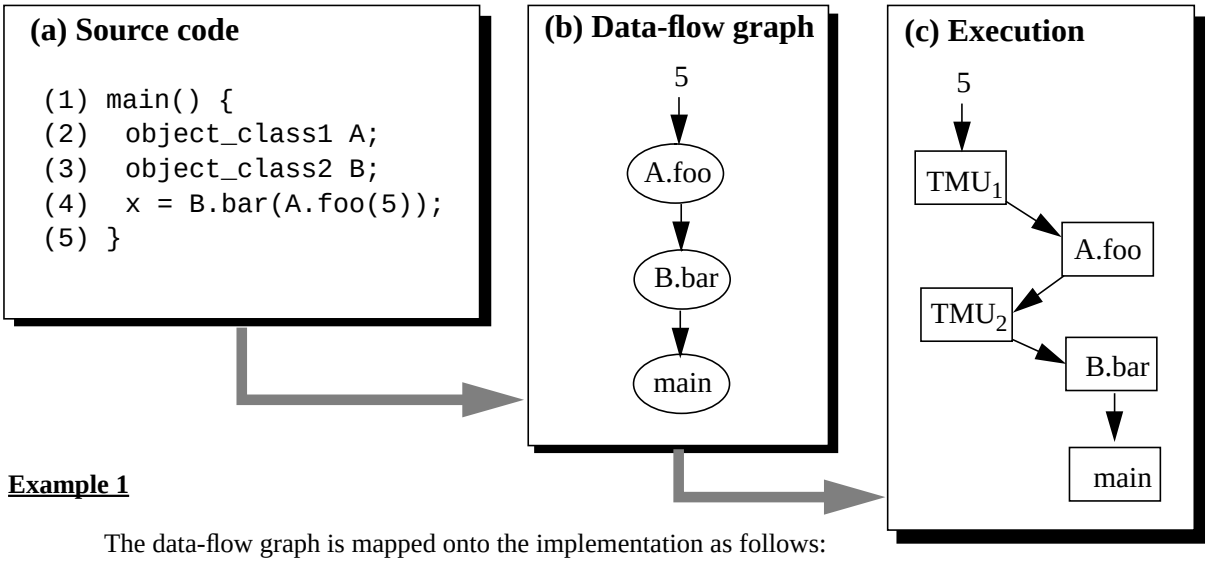
The Mentat run-time system implements a virtual macro data-flow machine that transparently constructs program data-flow graphs, schedules actors on processors, and manages communication and synchronization. The pure data-flow subset of the MDF model is implemented by the token matching unit (TMU) which is responsible for matching tokens and for enabling an actor when all its tokens are present. At run-time when a token is generated it is sent to a token matching unit. When all of the tokens required for an actor computation have arrived at the TMU, the TMU issues a scheduling request to the system scheduler. The scheduler selects a processor to service the request and notifies the TMU, which then forwards the tokens to the actor so that it may execute. To distribute the workload associated with regular actors, there is one TMU per host and work is divided among them via a simple hash function.

The mapping from a sample MPL source code fragment that uses regular objects to the implementation is shown in Figure 2. At run-time, calls made to regular object functions (actors) are transformed into a data-flow graph (Figure 2b), which is then acted on by the run-time system to deliver the proper arguments (tokens) to the appropriate object's function (Figure 2c). For more information on the Mentat system, including how it detects data dependence and builds data-flow graphs, see [8][10].

3. Extending the model to support fault-tolerance

To support fault-tolerance, the Mentat run-time system was modified to transparently handle the replication of regular objects. The goal was to let programmers build fault-tolerant applications without the need for learning complex protocols.

Figure 2 Transformation from source code to execution



Example 1

The data-flow graph is mapped onto the implementation as follows:

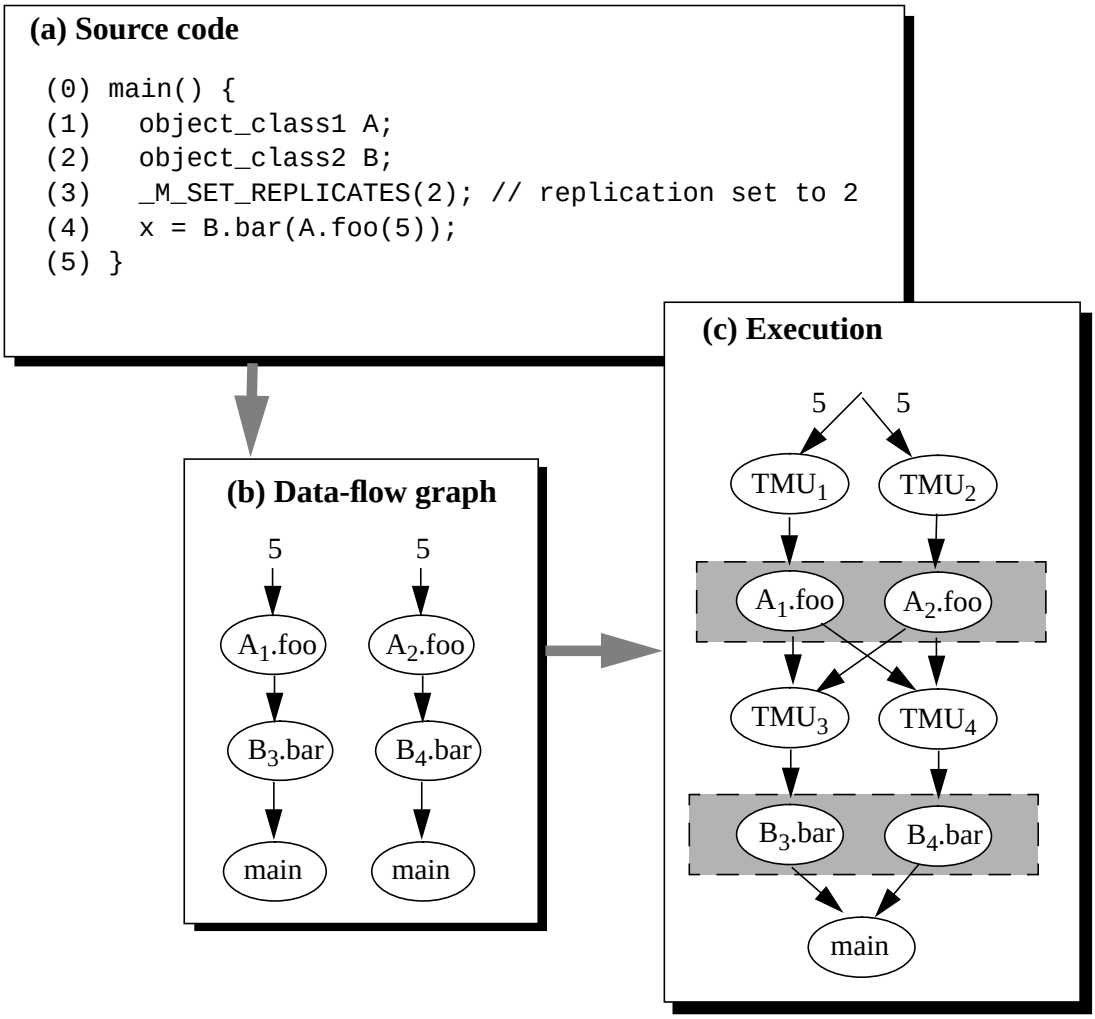
- A message containing the token "5" is sent to a TMU (TMU_1 in Figure 2c). The token contains a computation tag[1] that uniquely identifies the actor and the number of tokens required to enable the actor. The message also contains a copy of the program graph.
- Upon receiving the token, TMU_1 determines that the actor (A.foo()) may fire because all necessary tokens for the actor have arrived. TMU_1 makes a scheduling request to the system scheduler which creates an instance of object A and returns A's physical address.
- TMU_1 forwards the tokens and computation tags to object A.
- Object A executes function foo().
- When A.foo() finishes, it must send the result along all outgoing arcs in the program graph representation. Since B is a regular object, the result is passed to a TMU that is responsible for matching B.bar()'s tokens.
- B.bar() is handled similarly with the end results sent back to the main program.

1. Computation tags are similar to token colors.

Setting the desired level of replication is currently done with a macro called `_M_SET_REPLICATES`. This mechanism is temporary and is only a test vehicle — a cleaner language-based mechanism is being investigated. Figure 3 shows the same program as in Figure 2 with the replication level set to two.

Replication is accomplished by duplicating tokens when they are sent to the TMU. Rather than sending the tokens to only one TMU based on a hash function of the computation tag, the tokens are duplicated and the duplicates are sent to other TMUs. The choice of the additional TMUs is based on a hash function which is guaranteed to select distinct TMUs.

Figure 3 Implementing replication of regular objects



Example 2

The program of example 1 in Figure 2 has been modified to set the replication level to two (Figure 3a). Execution proceeds as follows (Figure 3c):

- A message is created that contains the token "5", the program graph, and the level of replication. This message is duplicated and sent to two distinct TMUs. The TMUs are chosen based on the computation tag for A.foo() using the primary and secondary hash function.
- Each TMU independently instantiates a copy of object A using the protocol described in Section 2.1. The actor corresponding to A.foo() is thus replicated.
- The result from each A.foo() is forwarded to each of the TMUs handling the next replicated actor (B.bar()). The TMU has been modified to detect duplicate tokens and discards the duplicates to prevent an exponential growth of objects instantiated.

Naive duplication of tokens will lead to an exponential growth of tokens and computations. Consider Figure 3b. If the token "5" to actor A is duplicated, two A's will execute. If their output tokens are also duplicated, four B's will be executed. For larger graphs this exponential growth would quickly overwhelm the system. To avoid exponential growth each TMU tracks the tokens it has already received. When, under normal circumstances, the duplicate tokens for a computation arrive, the TMU discards them. By discarding duplicates we avoid instantiating extra duplicate computations.

One critique of this technique is that one cannot know how long to "remember" which tokens have already been consumed. In fact this is not a problem. We use a fixed size table of past tokens. When a new slot is needed we throw away the token with the oldest timestamp. In the unlikely event that a duplicate token arrives after we have "forgotten" about it we simply schedule a redundant computation whose result will be thrown away later.

Note that only minor changes to the TMU were required. There is no coordination needed between replicated objects nor between TMUs. The failure of any *k-1* of the TMUs handling a *k*-replicated actor, or *k-1* of the replicated objects, does not prevent the successful completion of the program, though a current requirement is that the host where the main program is placed does not fail. A benefit of this method is that the replication algorithm is decentralized and hence scalable.

The TMUs responsible for a given replicated object, i.e. TMU_1 and TMU_2 or TMU_3 and TMU_4, (Figure 3) are placed on distinct hosts by the Mentat system. Mentat does not currently guarantee that the replicated objects themselves execute on separate hosts, though this is likely to be the case. Assuming a random placement of objects and one host failure, the probability that all objects are placed on the failed host is given by: $P(n, z) = \frac{1}{z^n}$ where n is the number of replicates and z the total number of hosts. Under saturation, the Mentat scheduler [9] effectively uses a random placement policy.

4. Checkboard Method

The basic idea is to register results with a "checkboard" as results are generated by the application (Figure 4). The checkboard is then periodically checked for progress. When no progress is made in a given time interval R, the missing computations are restarted. If the checkboard receives duplicate results, it simply discards them. An arbitrary number of host failures

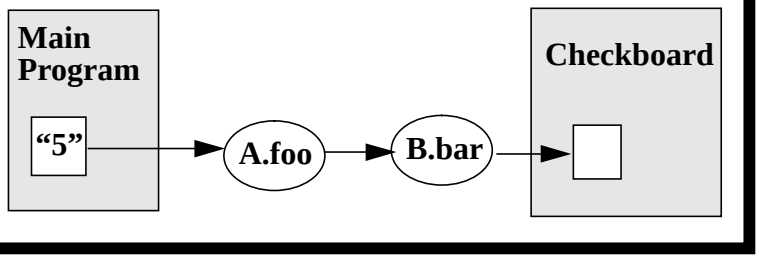
can be tolerated with this method provided that the hosts where the main program and the checkboard objects are started do not fail. To reduce the risk of this occurring, we place the checkboard object and the main program together on the same host. Moreover, if this host does not fail, the checkboard method can also tolerate network partitioning.

Figure 4 Checkboard method illustrated

(a) Pseudo-code

```
(00) main() {
(01)   object_class1 A; object_class2 B;
(02)   checkboard_class checkboard;
(03)   checkboard.create(MY_HOST); //place checkboard on same host as main
(04)
(05)   // start computation and register result with checkboard
(06)   checkboard.register_results(1, B.bar(A.foo(5)));
(07)
(08)   // check for progress and restart computations if needed
(09)   while (num_completed = checkboard.num_completed() < 1) {
(10)      sleep(restart_interval);
(11)      if (num_completed == save_num) {
(12)         checkboard.register_results(1, B.bar(A.foo(5)));
(13)      }
(14)      save_num = num_completed;
(15)   }
(16)   x = checkboard.get_result(1); // retrieve result from checkboard
(17) }
```

(b) Program graph



Example 3

The checkboard method, though conceptually simple, is considerably more complex than the transparent replication method of Section 3.

On line 6, we start the computation and register the result with the checkboard object. The result is tagged with the value 1 so that it can later be retrieved.

In lines 8-15, we check for progress and, if necessary, restart the computation (line 12).

On line 16, we retrieve the result from the checkboard object.

The checkboard method is applicable for programs or subsets of programs that are composed of idempotent computations. While the concept can be applied to any parallel system, the functional nature of Mentat regular objects guarantees that they can be executed again and again[4]. With parallel systems that do not provide first class support for restartable objects or entities, the user is responsible for correctly implementing restartable computations. While this is trivially accomplished with simple program graphs, Mentat allows the user to easily build applications with a more complex graph representation.

Determining a good value for R *a priori* requires knowledge about the available set of hosts, their computing power, and the nature of the computation. Such knowledge is often difficult to obtain in an environment such as Legion in which the available set of hosts may change and where underlying hosts are often shared with other users. The problem is that setting R too low can lead to computations being restarted prematurely, causing computing resources to be unnecessarily consumed even in the absence of host failure. Setting R too high does not hurt performance in the absence of failure, but it can lead to poor performance when hosts fail, as it takes longer to detect failure and restart the needed computations.

5. The Synthetic Pipeline

To compare the replication and checkboard techniques we have created a synthetic pipeline application. In this application, the work is divided into independent pieces that flow through a two-stage pipeline with the results being collected by the main program.

We have implemented three versions. The first version is a non fault-tolerant version using standard MPL code and the non-enhanced run-time system. The second and third versions are fault-tolerant employing the transparent replication and checkboard methods. The following sections describe the implementations in more detail and Sections 6 & 7 discuss our performance results and demonstrate the trade-offs between fault-tolerance, resource consumption and performance.

---

4. Programmers can also design persistent object to be idempotent. However, there is presently no linguistic, and hence, no direct run-time support for idempotent persistent objects.

5.1. Original MPL version

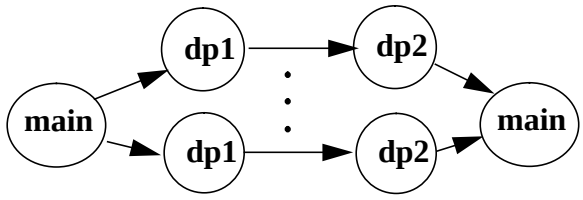
The MPL version and its data-flow graph representation are shown in Figure 5. On line 2, we declare two regular objects that are the data processing filters. The number of iterations in the pipeline is `NUM_RESULTS`. The data flows through the two filters in a pipeline fashion before being stored in an array back in the main program. The data-flow graph generated from the source code implicitly shows the independence of the computations. Thus, the equivalent of a DO_ALL loop is automatically achieved.

Figure 5 Original source code and data-flow graph

(a) Source code for original non fault-tolerant Mentat version

```
(0) main() {
(1)    my_data the_data[NUM_RESULTS];
(2)    data_processor dp1, dp2;          // these are regular objects
(3)    for (i = 0; i < NUM_RESULTS; ++i)
(4)       result[i] = dp2.process(dp1.process(the_data[i]));
(5) }
```

(b) Data-flow graph representation



5.2. Transparent replication version

For the replication method, the code looks similar to the original code except for the addition of the macro _M_SET_REPLICATES on line 4 (Figure 6). The number of replicates is related to the level of fault-tolerance desired. By default, all regular objects are 1-replicated. In general, a k-replicated object will tolerate k-1 failures assuming that all objects are placed on a different host. Setting the number of replicates high will improve the fault-tolerant characteristics of the application at the cost of higher resource consumption and possibly worse performance (depending on the number of hosts available).

Figure 6 Source code for transparent replication method

```
(0) main() {
(1)    data_processor dp1, dp2;          // these are regular objects
(2)    my_data the_data[NUM_RESULTS];
(3)
(4)    _M_SET_REPLICATES (number_of_replicates);
(5)
(6)    for (i = 0; i < NUM_RESULTS; ++i)
(7)      result[i] = dp2.process(dp1.process(the_data[i]));
(8) }
```

5.3. Checkboard version

The checkboard implementation of the pipeline application is considerably more complicated. The interface for the checkboard object and the code for the main program are shown in Figure 7. In lines 7-8, we invoke the computations and register the results with the checkboard. Lines 10-22 check for progress. If no progress is made within `restart_interval` time units, then all unfinished computations are restarted. Finally, in lines 24-25 we consume the results.

6. Results

When evaluating a fault-tolerance mechanism for a parallel computing environment we must keep firmly in mind that performance is the *raison d'être* of parallel computing. In a shared computing environment with multiple users, resource consumption is also an issue; any resources used to ensure fault-tolerance for one application cannot be used by another application. A design that incurs high overhead, recovers failed computations slowly, or uses large amounts of resources will not be used if the price is too high.

To determine the relative strengths and weaknesses of the fault-tolerant methods presented, we tested the performance of the synthetic pipeline application using both fault tolerant implementations and the non-fault tolerant implementation as a baseline. We tested the fault tolerant implementations under no failure and single failure scenarios to determine the recovery time characteristics of the different methods, while using a range of values for key parameters to the mechanisms, e.g. R the restart value for the checkboard method.

Figure 7 Checkboard object and main program

```
(a) mentat class checkboard {
(b)    public:
(c)      void register_results(int workerID, resultType result);
         // register result with the checkboard
         // workerID associates a tag with a computation
(d)      int isDone(int workerID);// is worker done?
(e)      resultType get_result(int workerID); // get result for one worker
(f)      int num_completed(); // number of completed computations
(g) }

(00) main() {
(01)   data_processor dp1, dp2;
(02)   checkboard board;      // the checkboard is a persistent object
(03)   my_data the_data[NUM_RESULTS];
(04)   board.create(MY_HOST); // place checkboard on same host as main program
(05)
(06)   // fire off computations and register the results
(07)   for (i = 0; i < NUM_RESULTS; ++i)
(08)        board.register_results(i, dp2.process(dp1.process(the_data[i])));
(09)
(10)   int save_num = 0; // keep track of the number of finished computations
(11)
(12)   // check for progress and if necessary, restart computations
(13)   while (num_completed = board.num_completed() < NUM_RESULTS) {
(14)     sleep (restart_interval);
(15)     if (num_completed == save_num) { // was there any progress?
(16)        // restart computations
(17)        for (i = 0; i < NUM_RESULTS; ++i)
(18)           if (!board.isDone(i))
(19)              board.register_results(i,
                     dp2.process(dp1.process(the_data[i]));
(20)     }
(21)     save_num = num_completed;
(22)   }
(23)
(24)   for (i = 0; i < NUM_RESULTS; ++i)
(25)        results[i] = board.get_result(i);
(26) }
```

We tested each configuration on a variety of workloads, ranging from 1-32 pipeline iterations with each stage of the pipe taking approximately 13 seconds. The times presented below

are the average start-to-finish wall clock times over 25 runs on a dedicated network of 8 Sun SparcStation2 workstations.

6.1. Non fault-tolerant baseline case

In Table 1 we show the average wall clock time elapsed for the baseline case with no failures. Notice that for 1 to 4 iterations of the pipe performance remains nearly constant. This is because Mentat automatically detects the independence of each iteration of the main loop and immediately schedules the first stage of the pipeline across all iterations. The theoretical limit for the pipeline is reached with 8 iterations. In practice, this limit is often not achieved because the scheduler may place multiple objects on the same host and thus the performance degrades as the number of iterations increases to 8.

TABLE 1 Non fault-tolerant Mentat baseline version

Iterations	Time (sec)	Total Resources Consumed (CPU sec)
1	27	26
2	28	52
4	33	104
8	50	208
16	71	416
32	120	832

6.2. Transparent Replication Method

The primary advantages of the transparent replication method are that it is generic and easy to use. Applications are not limited to a particular structure, as the Mentat run-time system can handle arbitrarily complex data-flow graphs. The programmer sets the replication level for objects and does not need to worry about the fault-tolerance protocols involved. The main drawback of this method is its high usage of CPU resources.

Table 2 shows performance and resource consumption with the level of replication set to 2. Resource consumption is directly related to the replication level and thus twice as many resources are consumed. When the available set of hosts is saturated with objects, performance decreases in relation to the non fault-tolerant baseline case as replicates delay the execution of other objects. This effect can be seen especially for 16 and 32 iterations, where performance respectively

degrades by 28% and 55%. On the other hand, when the number of hosts exceeds the number of objects, there is almost no performance penalty.

TABLE 2 Performance and CPU resources consumed with 0 & 1 host failure simulated

	No host failure				1 host failure	
	Baseline		2-replicated		2-replicated	
Iterations	Time (sec)	Total Resources (CPU sec)	Time (sec)	Total Resources (CPU sec)	Time (sec)	Total Resources (CPU sec)
1	27	26	26	52	28	50
2	28	52	27	104	29	95
4	33	104	36	208	39	196
8	50	208	58	416	58	370
16	71	416	91	832	93	770
32	120	832	186	1664	191	1540

There is no significant difference in performance between the 0 and 1 host failure case with the replication level set to two. This is expected since the objects that were placed on the failed host have a duplicate on another host. Resource consumption is slightly lower with one host failure as the objects that are placed on the failed host only partially execute or do not execute at all.

6.3. Checkboard Method

In the checkboard method, computations are restarted if no progress is made within a specified time interval. In the following set of experiments, the restart interval is varied from 15 to 60 seconds at 15 second intervals under both the 0 and 1 host failure cases. The restart values are chosen to bracket the expected completion time of one iteration (approximately 30 seconds). To make sure that the main program detects completion within a reasonable time interval, we check for completion every 5 seconds.

6.3.1. No host failures

Table 3 shows performance and resource consumption for the no failure case. Setting the restart interval too low (R=15) leads to both wasteful use of computation resources and poorer performance for all numbers of iterations tested relative to the baseline non fault-tolerant case. Resource consumption increases by 100-159%,with relative consumption rising with higher numbers of iterations. With `R=15`, the prematurely restarted objects compete with other objects for

CPU resources. Therefore, performance degrades (6-36%) but not as drastically as resource consumption.

As one would expect, as R increases, resource consumption quickly drops back to the levels of the baseline case. At the same time, performance improves but is still impacted by the overhead of the checkboard object and the fact that we only check for completion every 5 seconds.

TABLE 3 No host failures

	Baseline		Restart = 15		Restart = 30		Restart = 45		Restart = 60	
Itera-tions	Time (sec)	Total Resources CPU sec	Time (sec)	Total Resources CPU sec	Time (sec)	Total Resources CPU sec	Time (sec)	Total Resources CPU sec	Time (sec)	Total Resources CPU sec
1	27	26	31	52	31	26	31	26	31	26
2	28	52	32	104	33	52	32	52	32	52
4	33	104	45	208	44	104	41	104	40	104
8	50	208	53	416	52	208	52	208	51	208
16	71	416	94	884	74	416	72	416	75	416
32	120	832	158	2158	133	995	128	832	127	832

6.3.2. One host failure

With one host failure (Table 4), the trade-off between R, performance, and resource consumption is apparent. When R=15, we obtain the best performance at the cost of high resource consumption as most of the computations are prematurely restarted. Starting at R=30, resource consumption truly reflects the cost of restarting failed objects. However, with R≥45, we incur a

TABLE 4 One host failure

	Restart = 15		Restart = 30		Restart = 45		Restart = 60	
Iterations	Time (sec)	Resources CPU sec	Time (sec)	Resources CPU sec	Time (sec)	Resources CPU sec	Time (sec)	Resources CPU sec
1	31	52	36	29	32	26	31	26
2	36	104	33	55	56	57	58	57
4	53	234	47	107	64	112	66	109
8	66	442	91	224	89	218	90	221
16	117	832	138	468	160	463	158	471
32	202	2080	208	1186	223	926	234	931

performance penalty because it takes longer to detect failure with these relatively high restart

values. A good choice for the restart interval value in this case is `R=30` as it provides good usage of resources and performance.

6.3.3. Dynamic setting of restart interval

The performance results for various values of R show the necessity of finding a “good” value for the restart interval. However, it may be difficult to do so *a priori*. In fact, a “good” value for R may change for different executions of the same program or even during the execution of a single program (e.g. if resource availability changes). Therefore, we feel that setting R dynamically at run-time is essential to balancing resource consumption and performance. To test this hypothesis, we implemented a very simple heuristic - setting R to be the average of the three fastest iterations to complete. We tested this dynamic approach for the same iterations and failure modes as the static R method and present the results in Table 5.

TABLE 5 Dynamic reset of restart interval

	No Host Failures						One Host Failure					
	Dynamic Restart			Best Static Case			Dynamic Restart			Best Static Case		
Itera-tions	Time (sec)	Final R	Resour. CPU sec	Time (sec)	Static R	Resour. CPU sec	Time (sec)	Final R	Resour. (CPU sec)	Time (sec)	Static R	Resour. CPU sec
1	30	26	26	31	30-60	26	30	26	26	31	60	26
2	32	26	52	32	45-60	52	34	26	55	33	30	55
4	40	26	104	40	60	104	59	27	109	47	30	107
8	50	26	208	51	60	208	77	27	218	66	15	442
16	70	28	416	72	45	416	144	32	473	117	15	832
32	127	35	832	127	60	832	207	36	927	202	15	2080

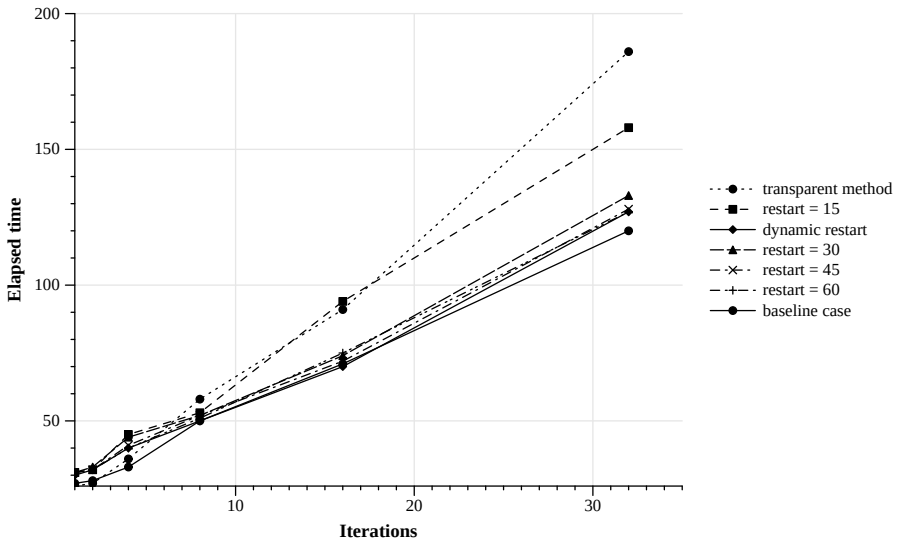
For the no failure case, resource consumption is optimal (i.e. there were no restarts) while performance is quite good, equalling or exceeding that of any of the values chosen for the static tests. In the one failure case, the dynamic algorithm utilizes resources about as well as the best results obtained across the different R values used in the static tests. In terms of performance, the best static case (`R=15`) outperforms the dynamic algorithm by up to 35% depending on the number of iterations. However, `R=15` is not efficient in its usage of resources. The dynamic algorithm settles on a value of `R` in the range [26..36], and thus, performance is similar to the static case of `R=30`.

Overall, setting R dynamically with our simple algorithm achieves a good balance between performance and an efficient utilization of resources. With no host failures, the dynamic algorithm performs as well or better than picking a static R value, while with one host failure, the dynamic algorithm does not perform as well as the best static case, but finds a good compromise between resource consumption and performance.

7. Observations

The graph in Figure 8 shows the performance of all the methods used in this paper in the no host failure case.

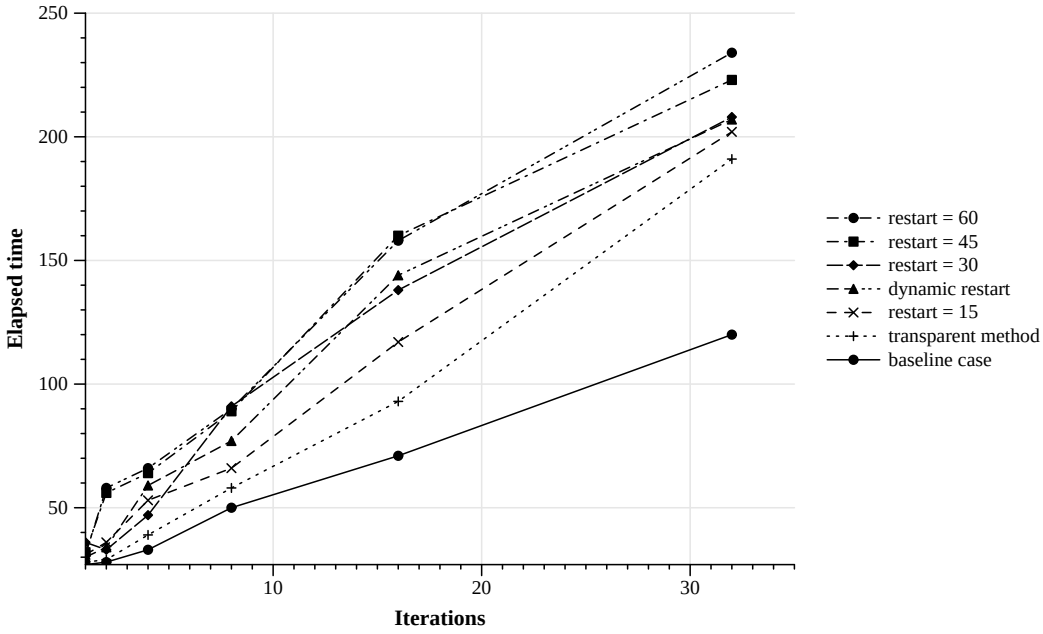
Figure 8 Performance comparison (no failure)



From 1 to 8 iterations, performance is similar for all methods. After 8 iterations, the graph bifurcates into two groups. The first group is composed of the baseline case and the checkboard method with restart values from 30 to 60 seconds. The second group is composed of the transparent method and the checkboard method with the restart value set to 15 seconds. The bifurcation marks the saturation point at which the CPUs are overwhelmed with objects. Past 8 iterations, replicated objects compete with other objects for CPU resources and delay their execution. The worst performers in failure-free mode are the transparent method and the checkboard method with the lowest restart value (R=15).

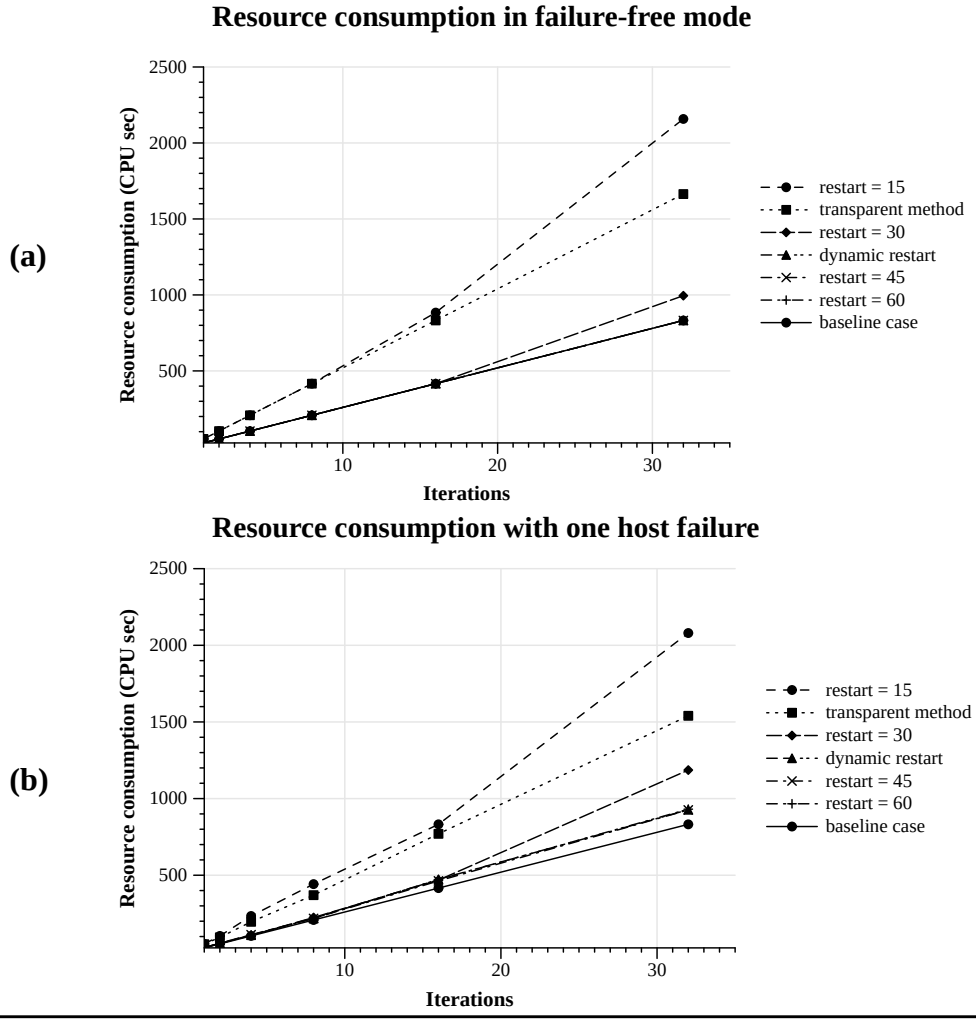
In the case of one host failure, the situation is reversed (Figure 9). The best performer is the transparent method followed by the checkboard method with `R=15`. In general, a higher restart value implies worse performance since the recovery time directly depends on the restart value chosen. In effect, these methods can be classified as pessimistic or optimistic. The pessimistic methods, i.e. the transparent method and the checkboard method with `R=15`, perform well when hosts fail, but are outperformed by the optimistic methods in failure-free mode.

Figure 9 Performance comparison (One host failure)



The resource consumption for all methods under both the 0 and 1 host failure cases are shown in Figure 10a and Figure 10b respectively. In failure-free mode, the checkboard method with $R \geq 30$ consumes essentially the same amount of resources as the baseline case. However, setting R too low (`R=15`) more than doubles the amount of resources consumed. The transparent method with the replication level set to two consumes exactly twice the amount of resources as the baseline case. With one host failure, resource consumption increases across all values of `R` for the checkboard method. However, for the transparent replication method, resource consumption slightly decreases as the failed host prevents objects from executing. Overall, the checkboard method with `R=15` and the transparent method are the least efficient in their CPU resource consumption.

Figure 10 Resource consumption for all methods



8. Related work

R. Jagannathan [12] classifies functional languages into extensional computing models (e.g. parallel graph reduction or term rewriting) and intensional computing model (dataflow) and argues that intensional computing models lend themselves to an efficient fault-tolerant implementation. Our experience with extending a data-flow based system, i.e. Mentat, corroborates this claim. Mentat and macro data-flow (MDF) differ from other large grained data-flow models such as CDF [2], HeNCE [5] and Code/Rope [6] 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 [5][6], where the programmer is responsible for generating the program graphs using a graphical interface.

There is a rich literature in fault-tolerance for distributed and real-time systems. See for examples the proceedings of Fault-Tolerant Computing Symposium (FTCS) and Real-Time Operating Systems (RTOS). 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 [3], PLinda [13], Orca [14], Calypso [4], and Fail-safe PVM [15]. These systems use a combination of well known mechanisms such as replication, transactions, message logging, or checkpoints and rollbacks to provide fault-tolerance. In Mentat, regular objects do not hold state and thus the overhead associated with maintaining consistency among replicates, of logging messages, or of taking checkpoints and rolling back processes are not incurred. Our philosophy is that users should only pay for the level of fault-tolerance they need.

9. 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 designed with fault-tolerance as a design objective.

The data-flow model, long a mainstay of parallel processing, offers hope. The model's 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 developed and presented two non-mutually exclusive methods for writing parallel fault-tolerant applications using a pure data-flow subset of Mentat. In the first method, the Mentat run-time system was modified to provide transparent replication of data-flow actors. The advantages of this method is that it is general and easy to use. Programmers simply set the level of replication desired in the parts of their program that need fault-tolerance. The main drawback of this method is its high usage of resources. While setting the level of replication high can improve

the fault-tolerance characteristics of an application, it can also have adverse effects on performance. We have found that when hosts are saturated with objects, performance actually decreases as replicated objects compete with other objects for CPU resources.

The checkboard method is the second method presented and is more difficult to use. The main idea behind this method is that applications register their results with a checkboard which is then periodically checked for progress. If none is made within a time interval R, the missing computations are then restarted. Choosing a good value for R is of paramount importance as it affects the performance and amount of resources consumed. In an environment where the available set of hosts can change during the execution of a program and where hosts are shared with other users, R may be difficult to estimate *a priori*. Therefore, we presented a simple scheme to dynamically set R and found that it achieves a good compromise between resource consumption and performance. Furthermore, the checkboard method tolerates an arbitrary number of host failures and network partitioning provided that the host where the main program is started does not fail.

By varying the level of replication in the transparent method or the restart interval in the checkboard method, programmers have at their disposal a "dial" with which to trade-off fault-tolerance, performance and resource consumption. Where programmers choose to set the "dial" ultimately depends on the relative importance that they attach to fault-tolerance, performance and resource consumption.

Now that we have demonstrated that robust techniques can be added to our existing system, the next steps are to improve the performance and resource consumption of the methods and to provide mechanisms to support fault-tolerance for persistent objects. For example, we have already begun work on algorithm changes in the transparent method that will significantly reduce resource consumption.

10. Acknowledgments

We would like to thank Adam Ferrari and Mark Hyett for their comments and suggestions in writing this paper.

11. References

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

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

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

[4] 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.

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

[6] 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.

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

[8] 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.

[9] A. S. Grimshaw and V. E. Vivas, “FALCON: A Distributed Scheduler for MIMD Architectures”, *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March, 1991.

[10] 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*.

[11] 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

[12] R. Jagannathan and E. A. Ashcroft, “Fault Tolerance in Parallel Implementations of Functional Languages,” *FTCS-21*, pp.256-263, Montreal, Canada, June 1991.

[13] 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.

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

[15] J. Leon, A. L. Fisher, P. Steenkiste, “Fail-save 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.

[16] A. H. Veen, “Dataflow Machine Architecture,” *ACM Computing Surveys*, pp. 365-396, vol. 18, no. 4, December, 1986.