Demonstration of the Remote Exploration and Experimentation (REE) Fault-Tolerant Parallel-Processing Supercomputer for Spacecraft Onboard Scientific Data Processing

Fannie Chen
Jet Propulsion Laboratory
Fannie.Chen@jpl.nasa.gov

Loring Craymer
Jet Propulsion Laboratory
Loring.Craymer@jpl.nasa.gov

Jeff Deifik
Jet Propulsion Laboratory
Jeff.Deifik@jpl.nasa.gov

Alvin J. Fogel
Jet Propulsion Laboratory
Alvin.J.Fogel@jpl.nasa.gov

Daniel S. Katz
Jet Propulsion Laboratory
Daniel.S.Katz@jpl.nasa.gov

Alfred G. Silliman, Jr.
Jet Propulsion Laboratory
Alfred.G.Silliman@jpl.nasa.gov

Raphael R. Some
Jet Propulsion Laboratory
Raphael.R.Some@jpl.nasa.gov

Sean A. Upchurch
Jet Propulsion Laboratory
sau@alumni.caltech.edu

Keith Whisnant
University of Illinois
kwhisnan@uiuc.edu

Abstract

This paper is the written explanation for a demonstration of the REE Project's work to-date. The demonstration is intended to simulate an REE system that might exist on a Mars Rover, consisting of multiple COTS processors, a COTS network, a COTS node-level operating system, REE middleware, and an REE application. The specific application performs texture processing of images. It was chosen as a building block of automated geological processing that will eventually be used for both navigation and data processing. Because the COTS hardware is not radiation hardened, SEU-induced soft errors will occur. These errors are simulated in the demonstration by use of a software-implemented fault-injector, and are injected at a rate much higher than is realistic for the sake of viewer interest. Both the application and the middleware contain mechanisms for both detection of and recovery from these faults, and these mechanisms are tested by this very high fault-rate. The consequence of the REE system being able to tolerate this fault rate while continuing to process data is that the system will easily be able to handle the true fault rate.

1. Introduction

The goal of the Remote Exploration and Experimentation (REE) Project [1] is to move supercomputing into space in a cost effective manner and to allow the use of inexpensive, state of the art, commercial-off-the-shelf (COTS) components and subsystems in these space-based supercomputers. The motivation for the project is the lack of bandwidth and long round trip communication delays which severely constrain current space science missions. Unlike typical radiation-hardened space-based systems, the use of COTS hardware will require the REE system to withstand relatively high rates of single event upset (SEU) induced errors. Depending on mission environments and component technologies, an REE system will be required to withstand average fault rates of between 1 and 100 SEU-induced soft errors per CPU-MB-day with occasional peaks of up to 1000 soft errors per CPU-MB-day[2]. Unlike traditional fault tolerant computer systems, however, the REE computer need not provide 100% reliability, but is instead, as with many sampled data or convergent-computation systems, allowed to occasionally fail in a computation. Periodic resets to flush latent errors, and other techniques which provide less than 100% availability, are also permissible. Further, the REE computer need not support hard real time or mission critical computation, as these tasks can be off-loaded to the spacecraft control computer.

The flexibility afforded by the above requirements allows the system to be optimized for high-performance, low-power, supercomputing rather than for "hard" fault tolerance. Thus, REE seeks to maximize simplex operation and minimize resource replication, redundant executions and other high-overhead strategies. (We should note that software-implemented triple-modular redundancy (TMR) and other high-overhead techniques

0-7695-0707-7/00 $10.00 © 2000 IEEE

will be developed and integrated into a suite of operational options for flexible fault tolerance, but it is expected that these will not be the primary operating modes of the system. It is, however, expected that a small subset of nodes may be called upon, from time to time, to operate in a highly reliable and real time manner.)

Another project goal is to allow scientists to develop science applications in their laboratories and to easily port the resulting software to the REE computer with minimal or no re-engineering for fault tolerance or for the spacecraft computing environment. In addition to COTS hardware, the project thus seeks to utilize a commercial operating system and to support standard commercial application development tools (compilers, debuggers, etc.) and methods to the maximum extent practical.

The REE computer architecture is a Beowulf-type[1] [3] parallel processing supercomputer comprising a multiplicity of processing nodes interconnected by a high speed, multiply redundant communication fabric. In the current instantiation of the system, dual Power PC 750 based computational nodes containing 128MB of main memory and dual redundant Myrinet [4] interfaces are interconnected via a redundant Myrinet fabric. The node level operating system is Lynx [5] Operating System (OS), to which multiple versions of MPI [6] have been ported. The current system may contain up to 20 nodes (40 processors) and is extensible to at least 50 nodes with a power:performance of better than 30MOPS/Watt. The applications are written so that they may be automatically configured to execute on up to 50 processors with the system being informed, by the application, of the optimal number of processors for maximum throughput and the system assigning the number of processors available based on system status and operational constraints such as available power, spares availability and mission phase.

There are currently 5 science teams writing applications for potential future NASA missions which may incorporate the REE computer. To aid application developers, a library of fault-detection-enabled scientific subroutines for linear algebra and Fast-Fourier Transform (FFT) routines has been developed. Work is ongoing to determine the utility of an error-correction-enabled library. In addition, continued analysis of application fault tolerance requirements and determination of the applications' native error tolerance is ongoing, as is the development of a generalized taxonomy of scientific software structure and the applicability (and overhead costs) of various software-implemented fault-tolerance (SIFT) mechanisms to these constructs.

[1] Beowulf-class computers were originally defined as parallel clusters of commodity hardware and open-source operating systems and tools. This definition has grown to include most clusters composed of personal computer central processing units (CPUs) and commodity operating systems and tools.

While we are currently in the process of expanding and documenting guidelines for application software developers, and while some of the SIFT strategy is the responsibility of the applications themselves, three system software layers have thus far been defined to aid in achieving the required fault tolerance:

A middleware layer which, conceptually, resides between the OS and the application,
a reliable communications layer which ensures that all system level communications are either error free or error-noted and which, conceptually, is viewed as a series of driver level enhancements to the node OS, and
A global coordination system which manages the overall system.

The combination of node operating system, reliable communications software, middleware, and global coordination layers are simply referred to as the REE System Software. Some of the responsibilities of the REE system software include:

1. Managing system resources (maintaining state information about each node and about the global system, performing system resource diagnostics, etc.).
2. Job scheduling (globally scheduling jobs across the system, local job scheduling within the node, allocation of resources to jobs, etc.).
3. Managing the scientific applications (launching the applications, monitoring the applications for failure, initiating recovery for applications, etc.)

The key components of the REE systems software are shown in Figure 1.

The immediate concern of the Applications Manager is to oversee the execution of the scientific applications. As the applications represent the ultimate "customer" of the REE environment, efficiently supporting their required dependability level is paramount. The Applications Manager monitors the science application for externally visible signs of faulty behavior as well as for messages generated internally by the applications requesting fault tolerance services.

Fault tolerance concerns for the REE System Software must also be addressed since these components ultimately ensure the correct operation of the REE environment. Several of its operations, such as scheduling and resource allocation are considered to be critical and therefore must be protected at all costs. We currently envision that these operations will therefore be run under the software-implemented TMR system previously discussed. Another module which must be protected is the Applications Manager. This software module, which is resident on each node engaged in applications processing, must be

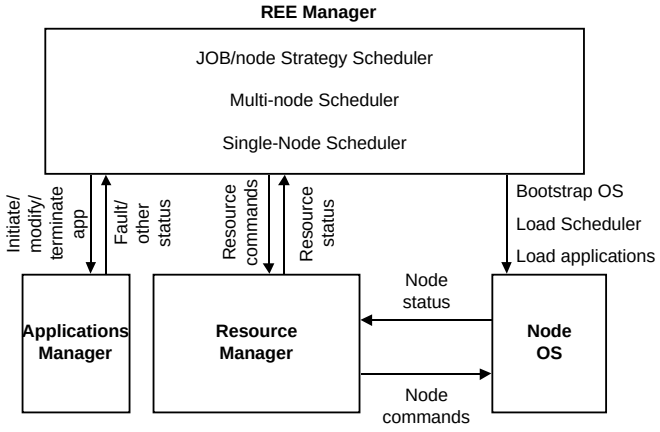

Figure 1. REE system software block diagram

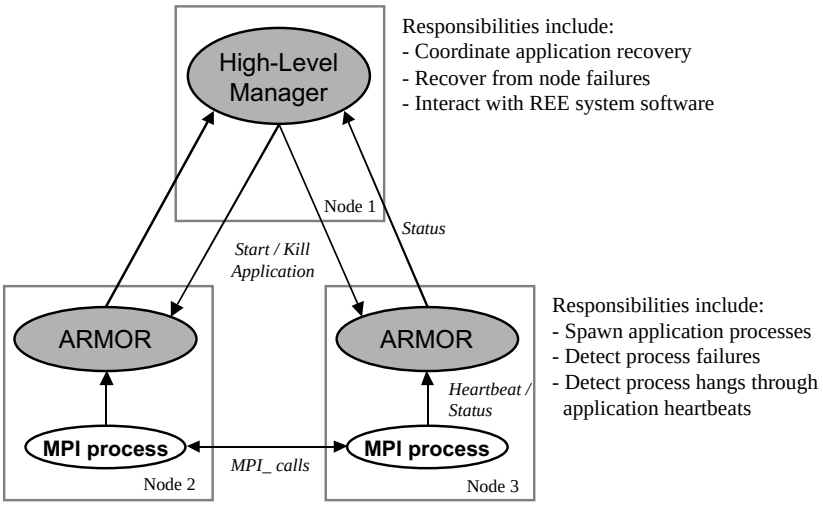

Figure 2. MPI application manager

self-checking to ensure correct operation of this "middleware" layer.

This demonstration will show one of REE's scientific applications executing under the Applications Manager. Faults will be injected into the nodes executing the application by a software implemented fault injector, and the cooperative interactions between the application, the Applications Manager and the COTS node operating system to protect the integrity of the computation will be shown.

2. Application Manager

The scientific applications executing on an REE platform are programmed using MPI [6], a standardized messaging interface used to implement parallel applications. These are typically computationally intensive programs that perform such actions as on-board image filtering and signal processing.

Core routines within each application, such as matrix multiplication, employ algorithm-based fault tolerance (ABFT) to help protect against data faults. Internal ABFT techniques, however, do not mitigate the need for an external entity controlling the applications. Capabilities such as launching application processes, terminating rogue application processes, detecting failures in application processes, and migrating failed processes to functioning nodes are some responsibilities that must be relegated to an external controlling entity. The Application Manager fulfills this role. REE currently uses the Chameleon application manager written by Prof. Ravi Iyer et. al at the University of Illinois [7] [8].

The Application Manager provides its fault tolerance services to the scientific MPI applications through ARMOR processes (Adaptive Reliable Mobile Objects of Reliability). ARMORs are built from a library of reusable components that implement specific services and techniques for providing fault tolerance. An overriding

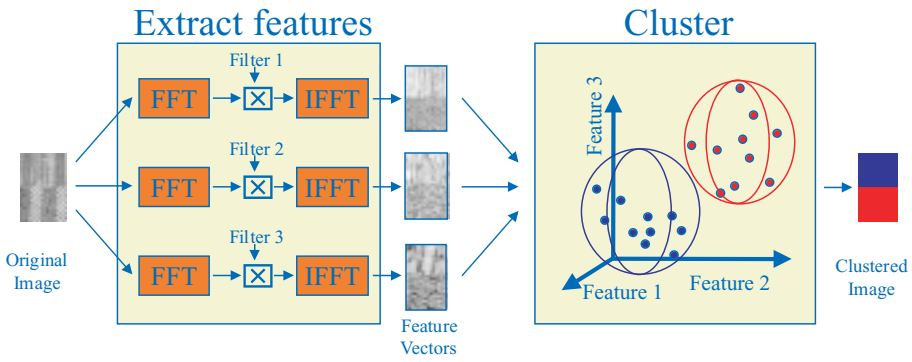

Figure 3. Rover texture analysis application

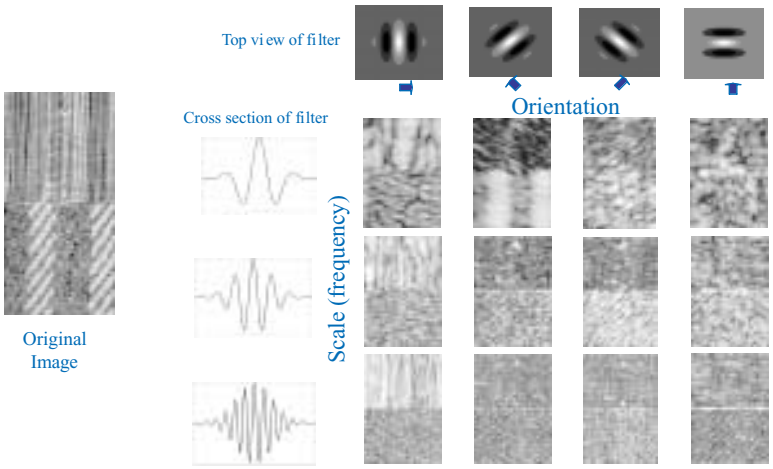

Figure 4. Image response to several filters

goal of the ARMOR concept is to have a uniform architecture through which customized levels of fault tolerance can be achieved. For REE, the ARMORs have been customized to provide oversight to MPI applications.

Because the target MPI applications often consist of several processes and because these applications cannot sacrifice performance, replication is not viewed as an acceptable approach for ensuring fault tolerance in an environment with constrained processing resources. Effective reporting and detection of errors is considered most important, as the target MPI applications can tolerate occasional restarts and rollbacks to previous checkpoints.

A challenge to the applications manager is to provide these error detection and recovery services to the target applications as transparently as possible. For the most part, the Applications Manager treats the MPI application as a black box entity. Each MPI process is directly overseen by an ARMOR executing on the same node, as shown in Figure 2. Failures in the application are detected by the ARMOR and communicated to a high-level ARMOR that coordinates recovery.

For tolerating non-crash failures, the overseeing ARMOR exposes a non-intrusive API (application programming interface) to the MPI application. Non-intrusive mean that the MPI application only need be lightly instrumented with API calls; no fundamental redesign of the application is necessary. Through this interface, the application can communicate vital information to the overseer ARMOR process so that the Application Manager can better gauge the health of the application. Examples include the reporting of correctable and uncorrectable ABFT errors directly detected by the application, as well as periodic updates to the ARMOR concerning the application's progress. Also

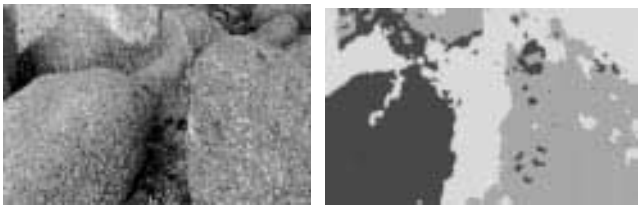

Figure 5. Example of texture segmentation input and output

currently being investigated is applying some of the error detection techniques found within ARMOR processes to the MPI applications as well [7]. (These include such things as control flow signature checking on the process's execution.) The ultimate goal of these techniques is to improve a process's self-checking capabilities, and the MPI applications outfitted with these techniques would work in tandem with the overseeing ARMOR to accomplish this goal.

Recovery of an MPI application is complicated by the fact that current MPI implementations do not allow single MPI processes to be restarted; instead, all processes must be launched again to restart the application. For this reason, there exists a single high-level ARMOR named the Fault Tolerance Manager (FTM) that coordinates the actions among all other ARMORs (the ARMORs that directly oversee each MPI process). The FTM is also responsible for handling node failures that affect one or more MPI applications. Whenever the FTM detects a node failure through its heart-beating mechanism, it must migrate all affected processes to another node. Selecting a spare node is done with the assistance of Resource Manager, an REE System Software component that is external to the Applications Manager. Again, the MPI applications themselves are oblivious to the exact recovery actions taken by the FTM and other ARMORs.

3. REE Application Demonstration

We will demonstrate one of the REE Science Applications, the Texture Analysis application developed by the Rover Science Team [9]. This parallel (MPI) application segments images according to texture information. This is one of the methods that a Mars Rover would use to determine rock types. The application can process any number of images. The processing steps for each image are shown in Figure 3. First, a number of filters are applied to the image. Each filter is a combination of a frequency and an orientation, as shown in Figure 4. The results of each filter are a feature vector. The feature vector measures the response of each pixel in the original image to the filter. After completing the filtering, clusters are segmented in the feature vector space. Then, each pixel is painted to show the cluster to which it belongs, as seen in Figure 5.

In order to test this application, two levels of fault-protection have been applied. The first level is Chameleon, used as the application manager. Once the application starts successfully, Chameleon ensures that it continues running until it has completed. The application has been slightly modified to make heartbeat calls to the Chameleon ARMORs, and Chameleon is aware of how often these heartbeats should occur. If one fails to occur within the response window, Chameleon assumes that the application has hung, or is stuck in a loop, and restarts it. The application also writes out its status to a log file. It can then read this file when it is started or restarted to know what images and filters have already been processed, so that it can start on the first image or filter that has not yet been completed. This "checkpointing" could be done at a finer level such as each FFT, but this current level is sufficient for testing and demonstration.

The second level of fault-protection is inside the application, though the application code itself is not modified. Instead, an ABFT [10] version of the FFT library is used. The ABFT versions of the FFT routines have the same calling sequences as the basic routines, but they check to see if the FFT was completed successfully before returning. If the FFT was not successful, they retry once. If this retry is also unsuccessful, the ABFT version of the FFT calls exit, which essentially promotes the problem to Chameleon to deal with by restarting the application on the current rock or filter. A flowchart of the ABFT operation is shown in Figure 6.

The demonstration to be shown will use the application running on an embedded system at the Jet Propulsion Laboratory (JPL). Through a series of scripts, output files will be transferred to the demonstration machine and displayed. An application-based fault injector named SWIFI (developed at JPL) will be used to insert random SEUs into memory and registers at 10 to 100 times the expected fault rate on the Martian Surface [2], which will exercise the two existing levels of fault protection. The demonstration machine will compare the application outputs from the code running with random fault injection to outputs previously generated from an unfaulted application. This will show the effects and overhead of the fault-protection layers.

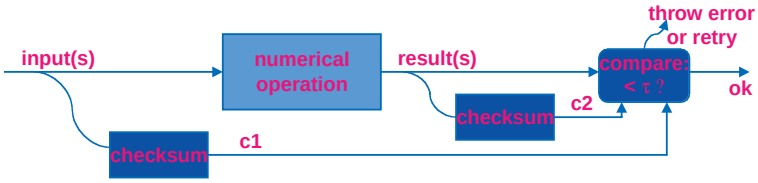

Figure 6. ABFT flowchart

4. Conclusions

The REE project requires that dependability be provided through software. Extensive error detection and recovery services are provided to the target applications through a variety of mechanisms including checkpointing, TMR, and ABFT-enabled scientific subroutine libraries. The applications are built and tested without fault-tolerance features, and then modified to use substantially off-the-shelf fault-tolerance components. The initial application manager being used by REE is built from a series of ARMORs, and is controlled by the prototype REE system software. Through the application manager and system software, the application is able to tolerate process failures, process hangs, and node failures. The ABFT-enabled libraries are essentially transparent to the application and provide high fault coverage of the mathematical routines themselves, though not of logical or arithmetic codes outside the library routines. Additional fault detection strategies will be required to protect the remainder of the application codes, the node operating system and system software.

It is possible that, in addition to having ARMOR technology protect the application manager, comparable error detection and recovery techniques can be extended to the REE system software as well. Because the REE system software does not have the strict requirement of being completely off-the-shelf, additional customizations can be made to the REE processes that allow them to take advantage of ARMOR technology. Specifically, the underlying ARMOR architecture can be embedded into the REE processes, allowing REE system software to take advantage of the reconfigurable error detection and recovery services currently found in standalone ARMORs. It is the intent of the REE project that the integration of the REE system software into the overall SIFT layer will result in a unified approach toward providing dependability to all facets of the REE software, including the target scientific applications.

Over the next 18 months, the REE project will continue the development of SIFT approaches for space-based parallel COTS supercomputing. The Project will culminate in the development of a final flight-capable prototype hardware/software system during the 2003-2004 time frame.

5. Acknowledgements

The work described in this publication was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology under a contract with the National Aeronautics and Space Administration (NASA).

6. References

[1] R. Ferraro, "NASA Remote Exploration and Experimentation Project," http://www-ree.jpl.nasa.gov/

[2] R. Ferraro, R. R. Some, J. Beahan, A. Johnston, and D S. Katz, "Detailed Radiation Fault Modeling of the REE First Generation Testbed Architecture," to appear in Proceedings of *2000 IEEE Aerospace Conference.*

[3] T. Sterling, J. Salmon, D. Becker, D. Savarese, *How to Build a Beowulf*, The MIT Press, 1999.

[4] Myrinet is a class of products of Myricom, Inc. (http://www.myricom.com/).

[5] Lynx OS is a product of Lynx Real Time Systems, Inc. (http://www.lynx.com/)

[6] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, J. Dongarra, *MPI: The Complete Reference*, The MIT Press, 1996.

[7] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, R. Iyer, "Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment," to appear in *IEEE Transactions on Knowledge and Data Engineering*, March 2000.

[8] Z. Kalbarczyk, S. Bagchi, K. Whisnant, R. Iyer, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Transactions on Parallel and Distributed Computing*, June 1999.

[9] R. Castaño, T. Mann and E. Mjolsness, "Texture Analysis for Mars Rover Images," *Applications of Digital Image Processing XXII*, Proc. of SPIE, Vol. 3808, Denver, July, 1999.

[10] M. Turmon, R. Granat, "Algorithm-Based Fault Tolerance for Spaceborne Computing: Basis and Implementations," to appear in Proceedings of *2000 IEEE Aerospace Conference.*