Software fault recovery for real-time signal processing on massively parallel computers*

*James Lebak, Jim Daly, Hank Hoffmann,
Jeremy Kepner, Jan Matlis, Patrick Richardson,
Ed Rutledge, Glenn Schrader
MIT Lincoln Laboratory*

1 Abstract

As massively parallel computers are increasingly used in real-time embedded signal processing applications, fault tolerance becomes an important concern. One of the most stringent requirements is that the system must remain available after a fault is detected. We have developed software technologies for fault recovery within a signal processing application. In this paper, we implement an example parallel signal processing application and compare the complexity and performance of three different strategies for implementing fault recovery.

2 Introduction

Real-time signal processing is one of the largest and most stressing compute intensive applications. In addition, many of these applications are of considerable importance to the government [3]. In the past, many signal processing applications have been addressed by using custom hardware. The increasing speed of commercial programmable processors now makes it possible to address many of these applica-

*This work is sponsored by the U.S. Navy under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the U.S. Air Force or the U.S. Navy.

tions with commodity massively parallel computers. A primary benefit of using commercial-off-the-shelf (COTS) systems is the ability to implement applications in software.

One of the key challenges in bringing massively parallel computing to real-time signal processing is how to implement fault tolerance capabilities in software. In theory, massively parallel systems present the opportunity for greatly increasing the fault tolerance of the system by adding only a few additional processing nodes. Exploiting this capability requires sophisticated software that can easily move processing tasks from one node to another. In this paper, we present the software technology we have developed for addressing this problem.

2.1 Fault tolerance in real-time signal processing

In a typical real-time signal processor, the flow of tasks is known when the system is constructed, and processing elements are dedicated to particular tasks at system initialization time. The failure of a particular processing node can therefore destroy the entire signal processing chain, requiring replacement of the failed node and reboot of the system. Unfortunately, the probability of a failure increases as the number of nodes in the system increases. In addition, the availability requirements of many signal processing systems make it impractical to shut down for the length of time required for replacement and reboot.

The simplest approach to meet the requirements of recovery time and availability is to double the size of the signal processor and switch to the redundant signal processor when a failure is detected. The failed processor can then be repaired while the redundant processor is operating. Obviously, this solution is quite expensive. Furthermore, if a unit in the second processor fails before the first is repaired, then the system still will not meet availability requirements.

A more cost-effective solution to meet availability and reliability requirements would be to maintain a small number of spare nodes in the system.[1] When an individual node fails, its role in the system can be assumed by one of the spare nodes. Although such a strategy will not allow recovery of data that was being processed at the time of the failure, in most signal processing systems the data rate is high enough that any individual frame of data is usually not critical. The focus is rather on getting the system back on-line as quickly as possible.

In this paper, we compare fault recovery strategies that allow one node to take over the role of another in a real-time signal processing application. We assume that mechanisms for detection of a fault and isolation to a particular node are in place and that the failure of individual nodes will not prevent the system from operating. Fault detection and isolation mechanisms are an active research area [1].

2.2 Parallel vector library (PVL)

The experimental apparatus that we use is an object-oriented parallel signal processing library, the parallel vector library or PVL. Written in C++, this library is based

[1]The replaceable unit will be referred to as a "node", though it could be an individual processor, a daughtercard, a board, a module, etc.

on our previous work with the space-time adaptive processing library (STAPL, [2]). PVL allows us to quickly and efficiently implement parallel signal processing applications with strict real-time latency requirements. The key concept of PVL is that it separates the operations to be performed from the description of the nodes that will be performing them.

Within PVL, there is a *task* object that represents a set of operations to be performed on a node or a set of nodes. Tasks can be thought of as coarse-grain entities in a signal processing pipeline; they define a control scope for a computation. The nodes that the task is running on are specified by another object referred to as a *map*.

Within a task, the user performs operations on data objects such as matrices and vectors. These are distributed objects that may also be assigned to particular nodes using the map object. Maps allow the user to describe an arbitrary block-cyclic distribution for each matrix or vector. Maps may be characterized as containing two types of information: *shape* information, describing the layout and the number of nodes, and *location* information, describing the exact positions of nodes.

Data objects are communicated between different tasks using objects called *conduits*. An individual data object exists only in the scope of one particular task at a time; using a conduit, these items can be sent to and received from other tasks in a non-blocking way.

The use of tasks and conduits provides an effective framework for integrating modules developed by different software engineers. Application developers can concentrate on the signal processing and linear algebra requirements of the algorithm and postpone mapping until later. Maps can be made to suit the individual platform that is being worked on. This allows the application to become portable, reducing lifecycle costs as systems are upgraded. Portable applications need not undergo debugging on embedded target hardware, but can be debugged on inexpensive platforms such as networks of workstations.

In terms of this library, the fault recovery problem can be summarized as finding an efficient way to give an object a new map after a failure has been detected. In the following section, we consider methods for solving this problem.

3 Fault Recovery Strategies

Consider a system with three nodes, named n_0, n_1, and n_2. Assume that initially n_0 and n_1 are performing useful signal processing work, and that they share a vector `v` of length `L` distributed in block fashion between them. Node n_2 is a spare node. At some point, assume a failure is detected in node n_1, and we wish to replace it with n_2. We define three strategies for recovering from the failure, referred to as *remap*, *redundant objects*, and *rebind*.

One strategy for failure recovery would be to free memory on previously mapped nodes and re-allocate memory on new nodes. This is the strategy we refer to as *remapping*. It is potentially a very slow process, and any pre-computed items such as coefficient tables for the fast Fourier transform (FFT, [7]) must be re-computed when using this strategy.

A second strategy attempts to reduce the time penalty of the remap method by allocating more memory. This strategy, which we will refer to as the method of *redundant objects*, requires the user to create objects at system initialization time corresponding to the various failure scenarios that may occur. When a failure occurs, the system uses a different set of objects based on the component that failed. Using this approach, no memory allocation has to be done at recovery time. However, the application has paid a substantial cost in terms of memory use. In general, the number of redundant objects that has to be maintained for complete fault coverage in a system with N nodes in which f nodes can fail is the binomial coefficient

$$\binom{N}{f} = \frac{N!}{f!(N-f)!}.$$

Therefore, this approach does not scale well enough to be considered for use in a real system. However, it has the smallest performance impact that can be imagined, making it useful for comparison purposes.

We refer to our compromise strategy between these two extremes as *rebinding*. In this approach, we constrain the new map to have the same number of nodes and the same distribution as the original map but allow the particular nodes to vary. Recalling that maps contain shape and location information, another way to look at the constraint is that the shape information in the map cannot change but the location information can.

When using rebinding, the programmer must give each distributed object a scope. At initialization time, the library will allocate on each node in the scope sufficient memory to play the role of any of the nodes in the object's map. There is no memory allocation performed at fault recovery time, and the only cost is extra memory allocation on the "spare" nodes.

Figure 1 gives pseudo-code examples of how a failure recovery would be accomplished using each method. Vector `v` is originally declared with a map `Map1` that includes nodes n_0 and n_1. Memory is allocated for $\frac{L}{2}$ elements of v on each of nodes n_0 and n_1. The new map `Map2` of `v` after the failure occurs will include nodes n_0 and n_2. The parameters to the vector constructor are the vector length and the map used to describe the distribution. Notice the third (scope) argument to the constructor for the rebind method, and the additional objects that must be built using the method of redundant objects.

4 Experiments

For each of these fault recovery methods, we are interested in measuring two kinds of overhead, performance and complexity. We compared the performance of the three methods on two different experiments of interest in signal processing; results of these experiments are detailed in Section 4.1. We then examined the application code for the second experiment to obtain a sense of the complexity impact of each recovery method; this comparison is shown in Section 4.2.

Remap	**Redundant Objects**	**Rebind**
Setup Phase		
`Map1` $=\{n_0, n_1\}$; `Map2` $=\{n_0, n_2\}$;		
`Vector v(L, Map1);`	`Map3` $=\{n_1, n_2\}$; `Vector v1(L, Map1);` `Vector v2(L, Map2);` `Vector v3(L, Map3);` `Vector *v=&v1;`	`Vector v(L, Map1,`$\{n_0, n_1, n_2\}$`);`
Recovery Phase		
`v.remap(Map2);`	`v=&v2;`	`v.rebind(Map2);`

Figure 1. *Pseudo-code for each of the three fault recovery strategies. A vector v is created on nodes n_0 and n_1 and is moved to nodes n_0 and n_2 after node n_1 fails.*

4.1 Performance Experiments

In this section, we describe the two experiments that we used to compare the performance of the three fault recovery strategies. Experiment 1 was an all-to-all communication or *corner turn* operation, a common stressing scenario in signal processing. Experiment 2 represented a small signal processing application.

Our platform for the performance experiment was an eight-node cluster of Pentium III PC's running Linux [8]. The cluster was connected by Gigabit ethernet and an eight-port switch, isolated from normal day-to-day network traffic. The underlying communication library was the message-passing interface, MPI [5]. The Linux kernel used was release 2.2.14 and the GNU C++ compiler was used.

Experiment 1

In this experiment, a vector of length L was moved from a block distribution on P_S source processors to a cyclic distribution on P_D destination processors. This operation requires all-to-all communication, an important operation in signal processing [4, 6]. The source and destination sets are disjoint. We measured the average time to perform this operation over 10000 iterations, using the function `MPI_WTIME` to measure time. After having measured a baseline time with no simulated failures, we assumed that during each iteration, a single destination node would fail, requiring a recovery. We simulated this recovery operation by changing the operating map in the destination after every iteration. We used $P_D + 1$ processors in the destination set; on even iterations, processor P_D was left out of the operating map, and on odd iterations, processor $P_D - 1$ was left out of the operating map.

The number of source processors P_S was fixed at 2, and the number of destination processors P_D was either 2 or 4. We used single precision floating-point vectors of length $L \in \{400, 1600, 6400, 25600\}$.

Times for this first experiment for $P_S = 2$ and $P_D = 4$ on the Linux cluster

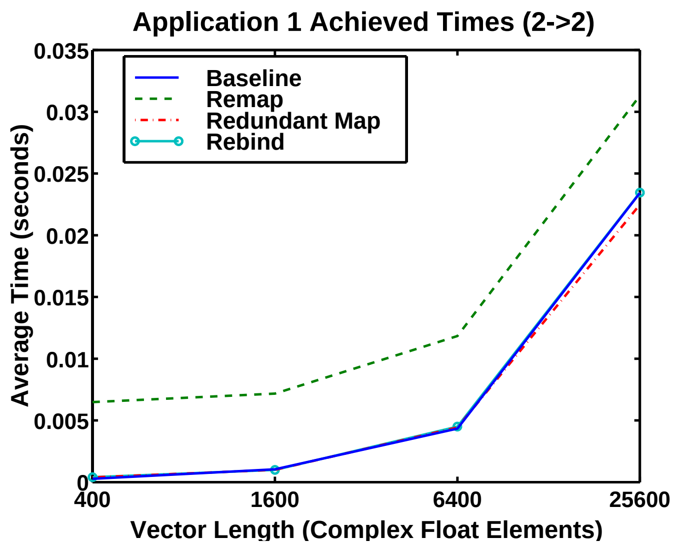

Figure 2. *Achieved times for the corner turn experiment with two source and two destination processors.*

are shown in Figure 2. An example bandwidth curve is shown in Figure 3. Notice that the baseline, redundant map, and rebind performance curves are all very close to each other in each of these figures, whereas the remap curve shows a substantial performance penalty in each case. The penalty appears to be independent of the vector length, meaning that the bandwidth penalty for using this method is greatest for small vectors.

Experiment 2

The first experiment was purely stressing on the communication fabric of the machine. To stress computation as well, a small signal processing application was used. The block diagram for this application is shown in Figure 4.

The application consists of two tasks, a front-end input task that generates a data matrix, and a back-end filtering task that processes the matrix. The two tasks are connected by a single conduit. In the filtering task, two filters are used, a coarse and a fine filter, with a two-to-one decimation between them. The matrix is size $X \times 2X$, and we again used disjoint processor sets, with P_i processors in the input task and P_f processors in the filtering task. We measured the average time in seconds and throughput in samples per second over 1000 iterations. To measure the fault recovery overhead, we used $P_f + 1$ total processors in the filtering task and alternated between two sets of P_f processors on consecutive iterations, similar to the first experiment.

In these experiments, $P_i = 2$, $P_f \in \{2, 4\}$, and $X \in \{10, 20, 40, 80\}$. Results for the two values of P_f looked very similar, and so only the case $P_f = 4$ is shown.

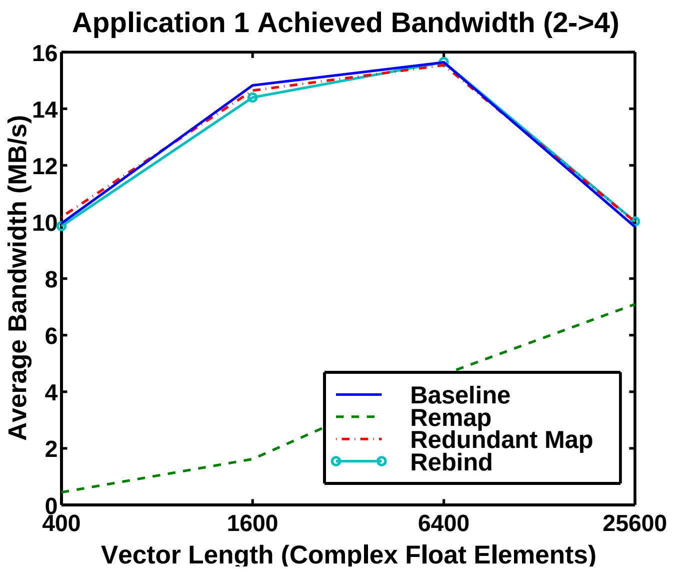

Figure 3. *Achieved bandwidths for the corner turn experiment with two source and four destination processors.*

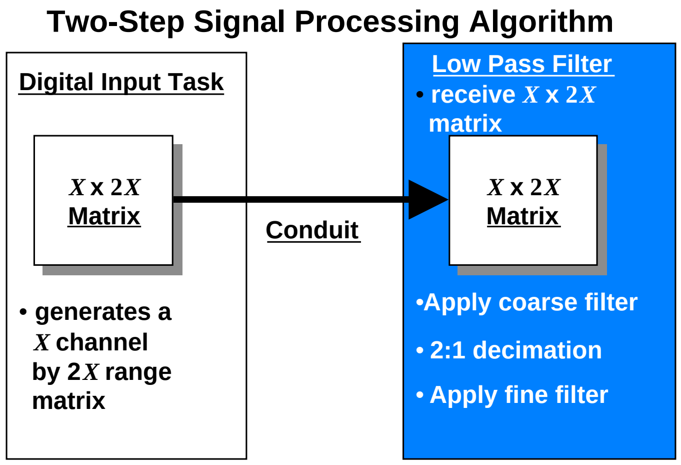

Figure 4. *Block diagram of the simple signal processing application used in experiment 2.*

Achieved times and throughputs are shown in Figures 5 and 6.

As in the first experiment, the redundant object and rebind methods have very similar performance to the baseline case, meaning that these methods introduce very little additional overhead for fault recovery. The remap recovery method introduces a constant overhead on the order of 10 ms. Notice that this is larger than the penalty

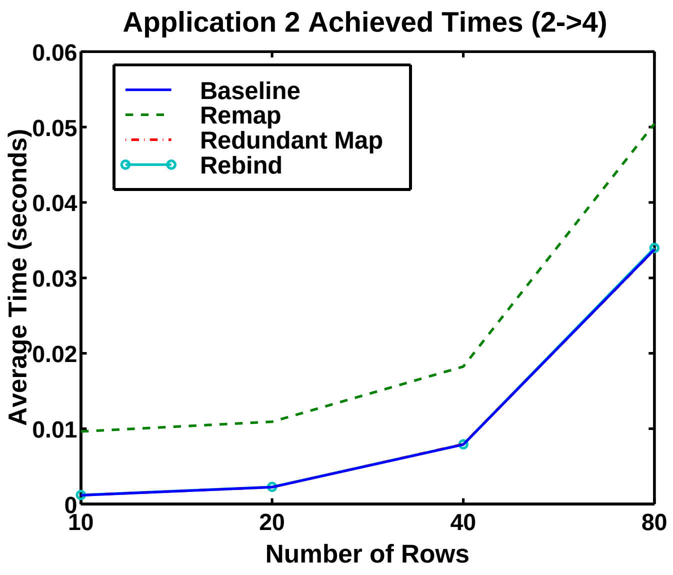

Figure 5. *Achieved times for the simple signal processing application with two source and four destination processors.*

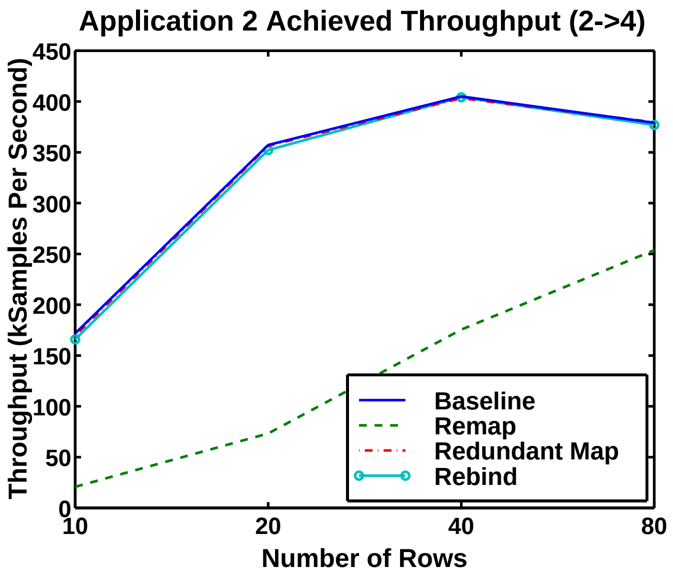

Figure 6. *Achieved throughput in ksamples per second for the simple signal processing application with two source and four destination processors.*

Table 1. *Line Count for Iteration and Fault Recovery Code.*

Method	Lines of Code
Baseline	90
Remap	93
Redundant Object	103
Rebind	96

for remap in the first experiment, because more objects need to be initialized and coefficients for the FFT are re-calculated.

4.2 Complexity

To compare the complexity of the various approaches, we examined the framework program that constitutes the "outer loop" in experiment 2. In the baseline case, the framework declares and initializes the task and conduit objects, performs the iterations, and records the time. The 90 lines of C++ program in the framework are designed to be independent of what the tasks themselves are doing. This framework is where code is added or changed to perform fault recovery by each of our three techniques. In Table 1, we compare the number of lines of code in the framework program for each fault recovery technique.

The code overhead is very small in this case because only a small number of objects are involved, and a small number of possible maps are used. Nonetheless, the redundant object method involves about a 10% penalty in terms of code overhead even in this simple case. We stated in Section 3 that the number of objects that has to be maintained to support this method in the general case grows exponentially.

5 Conclusions

We have investigated three potential methods for fault recovery. The remap method is slow, but uses no additional system memory in preparing for fault recovery. The redundant object method is very fast but uses lots of system memory. The third method, rebinding, represents a compromise between these two extremes, and obtains performance very similar to the redundant object method without using a great deal of system memory. This method has the additional advantage that the complexity of fault recovery is largely moved from the application programmer's domain into that of the library programmer.

Obviously, conclusions about the proper method for a particular embedded system depend on a number of different system parameters, such as the system timeline and the complexity of the processing. We intend to test these methods on more complex applications and on different parallel machines, including a Cray T3E and several embedded multiprocessors. We will evaluate the results in light of system requirements for a number of different embedded signal processing programs. We believe that the rebind method is a good first step toward a practical fault recovery technique for these systems.

Bibliography

[1] D. R. Aversky and D. R. Kaeli, *Fault-tolerant parallel and distributed systems*, Kluwer Academic Publishers, 1998.

[2] C. M. DeLuca, C. W. Heisey, R. A. Bond, and J. M. Daly, *A portable object-based parallel library and layered framework for real-time radar signal processing*, in ISCOPE '97: First Conference on International Scientific Computing in Object-Oriented Parallel Environments, Dec. 1997.

[3] C. J. Holland, *DoD perspective on the future of embedded software development*, in Proceedings of the Fourth Annual High-Performance Embedded Computing Workshop, Sept. 2000.

[4] J. M. Lebak and A. W. Bojanczyk, *Design and performance evaluation of a portable parallel library for space-time adaptive processing*, IEEE Transactions on Parallel and Distributed Systems, 11 (2000), pp. 287–298.

[5] MPI Forum, *MPI: A message-passing interface standard*, tech. rep., University of Tennessee, Apr. 1994.

[6] N. Park, V. K. Prasanna, and C. Raghavendra, *Efficient algorithms for block-cyclic array redistribution between processor sets*, in Proceedings of SC98: Tenth High Performance Networking and Computing Conference, 1998.

[7] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, 1992.

[8] M. Welsh, M. K. Dalheimer, and L. Kaufman, *Running Linux*, O'Reilly and Associates, 3rd ed., 1999.