Proof of Concept for Radiation-Tolerant Distributed Computing Incorporating Non-Hardened Nodes

Paul Hudgins, *Virginia Commonwealth University*

***Abstract*—The goal of this project is to develop a software framework for incorporating non-hardened processors into a distributed system on spacecraft. A full implementation proved to be far beyond the scope of this project. However, a proof-of-concept framework for defining and assigning tasks was successfully implemented, as well as error-handling for the most common error case. This framework introduces significant computational overhead, which is far outweighed by the benefits of using high-performance hardware.**

I. Problem

RADIATION-HARDENED processors for use in space applications are both extremely expensive and very slow. For example the commonly-used BAE RAD750 processor costs $200,000 and performs at 133-200 MHz. Most aspects of spacecraft command and control are not computationally intensive, but computation is a limiting factor for some areas. Computation-intensive applications include image processing for real-time landing site refinement on rovers, earth atmospheric re-entry, and data preprocessing on deep space probes with very limited bandwidth.

The primary problem caused by radiation in non-hardened processors is Single-Event Effects (SSEs) which flip random bits during computation. At low altitudes these happen rarely. SpaceX uses redundant non-hardened computers on Falcon 9 rockets, and applies commands by majority vote, tolerating isolated errors or the failure of one computer. At high altitudes and in deep space, errors are so frequent that this is not feasible.

II. Proposed Solution

The proposed framework is based on the procedures of an artillery Fire Direction Center. An FDC is a distributed computing system whose nodes are human beings, and can be very prone to error under some conditions. However, the output of an FDC always be perfectly accurate and timely. The solution is to divide a problem into many steps, ensure redundancy in each step, and so catch errors as early as possible. Optimizations for this sort of procedure include distributing tasks in a way that minimizes communication overhead, and increasing redundancy when error rates increase, to minimize the chance of having to recompute data.

In this framework, a computational problem is divided into a graph of tasks, each of which can have multiple input and output buffers. This is referred to as the task graph. A task is specified by a function pointer, which will be passed input and output buffers. A task is performed redundantly as soon as all of its prerequisites have been computed, and recomputed if necessary. If two nodes agree on the output of the task and one disagrees, the erroneous node can be corrected. The program flow is effectively a breadth-first search through the task graph. Loops in control flow are specified as loops in the task graph. Buffers are automatically garbage-collected when no longer needed.

The proposed architecture uses one low-performance radiation-hardened node referred to as the chief, and any number of high-performance non-hardened nodes. This enables existing flight software tools to run on the hardened chief node, and computationally intensive tasks to be offloaded onto the non-hardened nodes. It also allows the chief to perform all direct communication with hardware, and to maintain reliable data structures for directing the efforts of the non-hardened nodes.

A. Benefits of Parallelism

Parallelism can be easily specified with a task graph that divides a task into many parallel subtasks, which can be done programmatically. Without radiation, there would be no benefit to subdividing a problem more than the number of available cores. However, by subdividing a large problem into many smaller problems, errors can be detected sooner, decreasing their impact on runtime and increasing the level of radiation which can be tolerated. This benefit is not truly from parallelism, but it means that embarrassingly parallel problems can be made extremely tolerant to errors. The limiting factor for this benefit is the error-checking overhead.

B. Error cases

The problems caused by Single Event Effects are divided into three categories, all of which must be handled by a full implementation:

1) Computation Errors: Errors in computation, which will cause a task to produce incorrect output once. This is the most frequent radiation induced error, and the main focus of this project.

2) Input Buffer Errors: Errors in input buffers, which will cause a task to continue to produce incorrect output on a node until the error is corrected. Radiation effects have less impact on memory than on processors, so error case 1 will be more common than error case 2.

3) Node Failure: Errors which cause process or system failure, or prevent sending of a response. Any time one process takes significantly more time to complete a task than its peers, error case 3 can be assumed and a hardware reset can be performed. This requires that the non-hardened nodes either have a bare-metal implementation or an extremely minimal operating system, minimizing the time required for hardware reset. The maximum level of radiation that the system can tolerate is when the mean time between node resets approaches the node restart time. If the nodes have a limited operating system and the large majority of computation is devoted to producing output, instead of maintaining the system, then error case 1 will be more common than error case 3.

III. Implementations

This project was implemented with MPI. Asynchronous communication is required for responses from non-hardened nodes to the chief. Debugging with MPI proved to be the most significant challenge of the project. Algorithms such as navigating the task graph and garbage collection, which are implemented only on the chief, can become very problematic because of the communication required and the difficulty of debugging. Two separate proof-of-concept processes were created, one that implements single-threaded task graph navigation, and one that distributes a single task to nodes and checks it.

A. Single-Threaded Task Graph

The core functionality of defining and navigating a task-graph was successfully implemented. This is key because that structure can be used by the chief node to automatically handle dependencies, identify sources of memory errors, and distribute computation appropriately. The task-graph data-structure is hard-coded in this demo and is difficult to read. A preprocessing script for specifying task graphs would be needed in a final implementation. The task graph in the demo implements a loop, as well as mapping a function to many buffers and using another to reduce them.

Garbage collection of data buffers is implemented. This not only saves memory, it is key to the implementation of loops in the task graph. Task assignment constantly tries to compute all tasks for which prerequisites are ready, and garbage collection deletes all prerequisites for which tasks are complete.

B. Distribution of Tasks to Nodes

Distributing tasks to nodes and error checking proved very difficult. Each node has its own copy of the task-graph, and the radiation-hardened chief node sends commands to nodes telling them to execute tasks and check buffers. Radiation effects are simulated by adding random bits to the output of a task. The chief asynchronously receives responses that include a hash of output buffers, and checks them for equality.

The following key steps were not completed: Sending commands to non-hardened nodes to trigger peer-to-peer transfer of buffers, and integration of this command structure with the task-graph navigation structure.

C. Future Optimizations/Improvements

1) Moving computation to data: The chief currently assigns the next task to the next available node, and all buffers are kept current on all nodes. Network overhead could be significantly reduced by an algorithm on the task-graph structure that minimizes the amount of peer-to-peer communication required, by assigning tasks to nodes that already have prerequisites.
2) Task queues on nodes: Assigning nodes a queue of tasks can prevent error-checking from becoming a synchronization barrier.
3) Inverting references in the task graph will prevent the overhead of searching that occurs several times in this implementation.
4) A preprocessor is necessary to make task graphs more readable.

IV. Conclusion

This project was much more difficult and complex than expected. Based on lessons learned, I believe this is the best approach to the problem, and I plan on continuing to develop it. This approach will probably never be as easy to use as other distributed computing frameworks, but it will allow spacecraft to perform computation at a level that is currently impossible.