Fault Tolerance in Distributed Parallel Computing

M. Long, longm@cs.colostate.edu Graduate Student
CS530DL Spring 2011, Colorado State University

***Abstract* — Modern applications are requiring more processing power than ever before. The use of parallel development for software that runs across a distributed network of computers is becoming much more common. However, when failures occur in supercomputing clusters there can be substantial time lost if they system is unable to recover. Parallel processing jobs can run for weeks or months to compute the desired results. We look at existing solutions to preventing parallel computing failures in parallel shared-memory systems as well as more modern MapReduce systems and finally propose an M-Modular Redundant version of MapReduce for error free and fault tolerant computing.**

I. Introduction

Parallel programming is rapidly becoming the most important method to gaining computing performance from modern computing hardware. Processing speed improvements will slow from the previous meteoric increases over the last few decades and systems will need to extract performance by doing tasks in parallel across many nodes. Parallel programming is a complicated paradigm to master when compared to sequential programming specifically when detecting and recovering from faults. Faults in parallel systems can come from a variety of sources such as hardware failures at the processor or memory level, environmental failures due to power loss, or even geographic failures caused when a datacenter is lost for some reason. Adding redundancy and fault tolerance becomes more important than simply increases the speed with which calculations are completed since a failure during the calculations will cause the system to start over and delay the results substantially.

This paper intends to explore the current state of fault tolerant computing in parallel computing systems. These systems include shared memory such as those used with OpenMP and MPI parallel frameworks as well as newer technology such as MapReduce systems implemented by the open-source Hadoop system. Finally, we intend to propose a N-Modular Redundancy system for use in large-scale MapReduce parallel jobs.

II. Checkpointing for Fault Tolerace

Currently, one of the most common fault-tolerant systems used is checkpointing. Checkpointing is a method of saving processor state at various points during the computation to allow the system to recover from failures. The checkpoint is somewhat analogous to a core dump and provides sufficient information to recover the state and restart computations on another node. Checkpointing works well in a variety of parallel and distributed processing systems from multicore systems to cluster computing and massive parallel processing systems. There has been substantial research around minimizing the number of processes required to complete a consistent checkpoint [3] as well as evaluations of the performance of various checkpointing techniques [3]. As checkpointing requires time and adds to the overhead of the system minimizing it can be beneficial to reducing the overall computation time of any given task.

There are two commonly used forms of checkpointing: Independent checkpointing and coordinated checkpointing. Independent checkpointing is often used in "scientific calculations and corporate administration systems" [6]. These are systems that are considered non-critical and operation repeatable applications [6]. In these checkpointing systems the application threads are allowed to create checkpoints independent of other threads and there is no synchronization between threads doing the checkpointing. Independent checkpointing presents many advantages, as there is no synchronization there is no communication overhead and no delay in agreeing on checkpoint. The processing can continue without the extended blocking of the checkpoint. Additionally, if the checkpointing occurs in an asynchronous fashion the systems can reduce the I/O constraints on the storage subsystem by checkpointing one at a time to improve overall performance.

While independent checkpointing has advantages there is a major drawback. Should the system need to roll back the computation to a stable checkpoint, independent checkpoints can cause a cascading rollback of other nodes. This can occur when the systems rolls back to a checkpoint causing other threads to also have to roll back to a checkpoint to get the entire systems to a stable point in the calculations. This chain reaction of rollbacks will cause substantial delays in rollback time but also in lost work. It is possible, although unlikely, that the roll back could erase all the work in the system to that failure point and restart the entire calculation.

Coordinated checkpointing is the second form of widely used checkpointing and is often used in mission-critical applications. These include "life-supporting systems in a hospital and non-idempotent applications such as ticketing systems" [6]. This system ensures that the checkpoints are consistent. This consistent checkpointing can be achieved by using a two-phase commit protocol. This method results in a global checkpoint being taken of the system at a given time. This requires the processes to communicate and synchronize among themselves for each system-wide checkpoint. This obviously results in overhead however, the main advantage to using this method is that it checkpoints

[2] Elmozahy et al used the terms consistent and optimistic checkpointing in the paper "The Performance of Consistent Checkpointing." These terms generally equate to independent and coordinated checkpointing respectively. The terms have been updated for consistency in this paper.

the system in a working and stable state and can avoid the cascading roll back problem seen in independent checkpointing. This increased overhead gains improved recoverability should a fault occur. The complete snapshot is a working checkpoint that the system can revert to and being processing after a failure. While the checkpoint is completing, execution of the parallel tasks is blocked by the system to prevent state changes. Once the checkpoint completes, operation resumes normally.

III. Performance of Checkpointing

A major concern with checkpointing is the overhead and time penalty for performing the checkpointing operation. Elmozahy et al in the paper [3] "The Performance of Consistent Checkpointing[2]" addressed this issue. According to the paper, there are three main causes of checkpointing overhead: Saving checkpoints to stable storage, interference between checkpointing and the execution processes, and the cost of communication between processes.

Table 1 gives an overview of the general memory footprint of the programs tested along with the observed running time without checkpointing of any sort. The result of the performance experiments, shown in table 2, indicate that checkpointing, as performed for the tests, did not produce a significant performance penalty on a variety of tested programs. The largest increase in running time resulted in a 5.8% increase. In fact, the nqueens program experienced no discernible increase in runtime under a checkpointing system. These results indicate that checkpointing is a cost-effect and performance-effective method for improving fault tolerance in distributed and parallel computing environments.

Program Name	Running Time (minutes)	Per Process Memory (Kbytes)		
		Code	Data	Total
fft	186	21	555	576
gauss	48	20	576	596
grid	59	21	2163	2184
matmult	137	20	2348	2368
nqueens	77	18	22	40
prime	53	38	74	112
sparse	65	22	2954	2976
tsp	73	21	27	48

Table 1 – Program Statistics (Reproduced from [3])

[2] Elmozahy et al used the terms consistent and optimistic checkpointing in the paper "The Performance of Consistent Checkpointing." These terms generally equate to independent and coordinated checkpointing respectively. The terms have been updated for consistency in this paper.

Program Name	Without Checkpoint (sec)	With Checkpoint (sec)	Difference	
			sec	%
fft	11157	11184	27	0.2
gauss	2875	2885	10	0.3
grid	3552	3618	66	1.8
matmult	8203	819	16	0.2
nqueens	4600	4600	0	0.0
prime	3181	3193	12	0.4
sparse	3893	4119	226	5.8
tsp	4362	4362	0	0.0

Table 2 – Running time increase with Coordinated checkpointing using copy-on-write (reproduced from [3])

The authors utilized copy-on-write schemes to avoid blocking while writing the checkpoints to stable storage. Table 3 shows the running times without using copy-on-write compared with copy-on-write and the performance degradation is largely dependent on the number of checkpoints being written.

Program Name	% Increase in running time	
	Blocking	Copy-on-write
fft	0.2	0.2
gauss	13.7	0.3
grid	85.0	1.8
matmult	3.7	0.2
nqueens	1.8	0.0
prime	2.9	0.4
sparse	20.0	5.8
tsp	1.8	0.0

Table 3 – Blocking versus copy-on-write checkpointing (reproduced from [3])

In addition to the copy-on-write scheme, checkpointing can be done in full or incremental checkpoints. The goal of the incremental checkpoint is to reduce the amount of data written at each checkpoint. With the reduction in data the time required to checkpoint should be reduced and thus the program performance increased. Table 4 shows the improvement observed in running times from full checkpointing to incremental checkpointing.

Program Name	% Increase in running time		
	Full Checkpoint	Incremental Checkpoint	% Reduction
fft	17.6	2.0	89
gauss	17.8	14.1	21
grid	60.2	60.2	0
matmult	66.1	3.3	95
nqueens	2.6	1.5	42
prime	4.0	2.8	30
sparse	86.9	25.7	70
tsp	2.2	0.2	91

TABLE 4 – FULL CHECKPOINTING VERSUS INCREMENTAL CHECKPOINTING (REPRODUCED FROM [3])

Incremental checkpointing improves performance in virtually all cases as seen in table 4. The final measurement the authors made was to compare optimistic versus coordinated checkpointing. Table 5 shows the performance difference between the two checkpointing schemes outlined. The table shows that the performance varies based on the program and neither independent nor coordinated checkpointing provide guaranteed performance improvements over the others.

Program Name	% Increase in running time	
	Coordinated Checkpointing	Independent Checkpointing
fft	0.2	0.2
gauss	1.0	0.3
grid	1.6	1.8
matmult	0.1	0.2
nqueens	0.0	0.0
prime	0.2	0.4
sparse	3.0	5.8
tsp	0.0	0.0

TABLE 5 – COORDINATED VERSUS INDEPENDENT CHECKPOINTING (REPRODUCED FROM [3])

IV. CHECKPOINTING INTERVALS

An important consideration when performing checkpoints is the frequency with which the system performs checkpoints. Frequent checkpoints, while helpful in restarting calculations on a working node without excessive loss of data, can cause performance degradation, as the system is constantly checkpointing and not getting as much work done. However, reducing the interval to checkpointing can result in too much work needing to be performed again once a node or core fails. The balance exists in an optimal checkpointing interval.

Bressoud and Kozuch performed an analysis on checkpointing using data provided by the Los Alamos National Laboratories on three separate cluster-computing systems over the span of many years. Two systems comprised of 1024 nodes and 4096 cores and one system of 256 nodes and 1024 cores. Bressoud and Kozuch presented substantial data regarding the execution time of programs in the presence of faults running on fault tolerant and non-fault tolerant systems. The execution time, E_F, is given by the following in equation 1.

$$E_F = \frac{(e^{\lambda F} - 1)}{\lambda}$$

EQUATION 1

As the number of cores n increase in a system without checkpointing, the running times of programs on those systems increases due to unrecoverable faults. The programs must be restarted on the system and attempt to complete prior to a failure. The data from Bressoud and Kozuch indicates that running time can increase exponentially as shown in figure 1. Execution time is plotted as a function of n, the number of cores in the system. The graphs show the data from two of the three systems from Los Alamos National Laboratories, S18 and S20 as well as two programs, P1 and P2 and the results of their simulation.

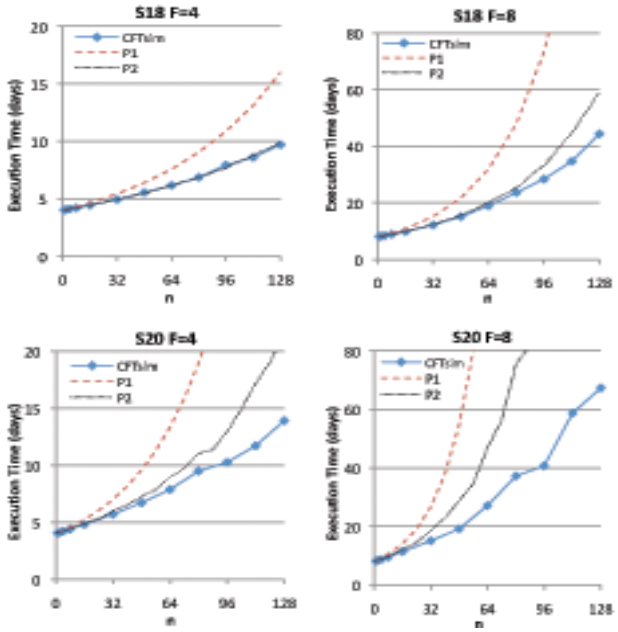

FIGURE 1 EXECUTION TIME FOR COORDINATED APPLICATION WITH NO CHECKPOINTING - REPRODUCED FROM [13]

As the graphs in figure 1 clearly show, the execution time increases with the number of cores available due to the lack of fault tolerance. Jobs are not completed in their simulations prior to faults occurring forcing a complete restart of the calculations in hopes that they complete prior to a fault.

Once checkpointing is introduced the performance of long-running jobs in the presence of failures improved drastically. Figure 2 shows the running times of an application in days as a function of I, the frequency of checkpointing in minutes. The various core values n are plotted on separate lines.

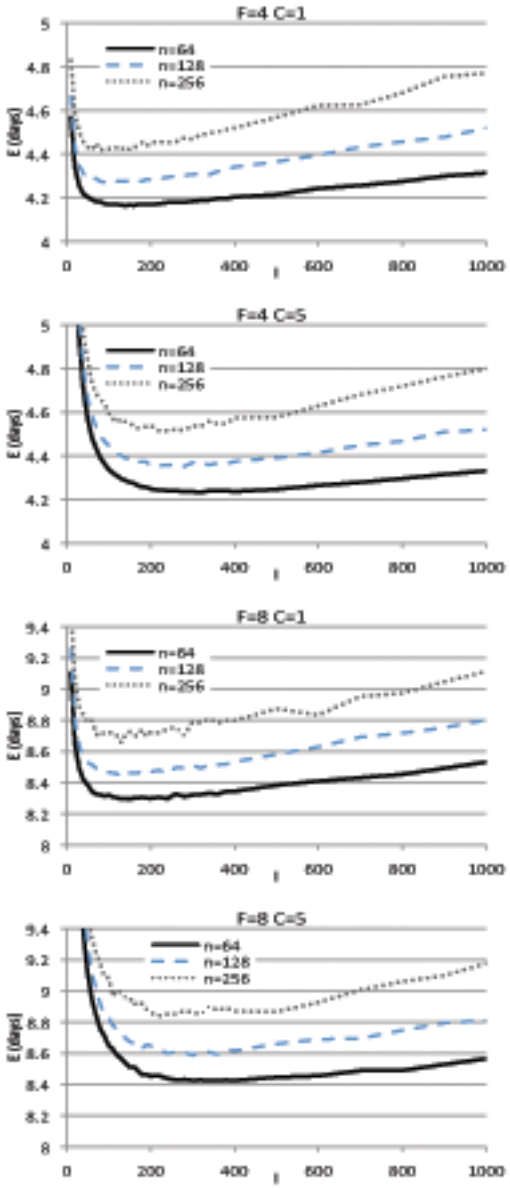

FIGURE 2 – RUNNING TIME OF AN APPLICATION AS A FUNCTION OF I, THE FREQUENCY OF CHECKPOINTING IN MINUTES. – REPRODUCED FROM [13]

As is obvious from the graphs, the running time has decreased dramatically as the number of cores n increases (shown as separate lines on the graphs). The data also show that the frequency of checkpointing can have an impact on the performance of the system. If the system checkpoints too often the overall performance is degraded sharply. However, if the system does not checkpoint enough, the time to recover from a failure will degrade the performance as well thought not quite as dramatically. Bressoud and Kozuch describe a equation for determining the optimum checkpointing interval in minutes as a function of the failure rate λ and C, the checkpoint overhead during which no task work is accomplished in minutes as seen in equation 2.

$$I_{opt} = \sqrt{2C/\lambda}$$

EQUATION 2

Additionally, we can see from equation 2 that as values of C increase the optimal checkpointing interval will increase. To determine the appropriate checkpointing interval a programmer must be very aware of the checkpoint overhead to ensure the best performance in addition to the failure rate of the system.

V. HYBRID CHECKPOINTING

Given that each of the described checkpointing systems has its advantages and disadvantages it would be logical to combine these systems into a hybrid checkpoint system such that the weaknesses of each are masked by the strengths of the other. Cao et al proposed such a system in their paper [6] "Checkpointing in hybrid distribution systems." In this hybrid system the authors proposed separating the system into subsystems of coordinated (CO) subsystems and independent (IN) subsystems based on the checkpointing schemes employed. The hybrid system has subsystems perform coordinated checkpoints at a low level. Each subsystem then propagates up to a single independent checkpointing system where the system can make checkpoints of each subsystem for recovery. The goal of the hybrid system is such that a rollback in the independent checkpointing subsystem will not lead to a rollback in the coordinated checkpointing system and that a rollback within the coordinated checkpointing subsystem will not propagate to a separate coordinated checkpointing subsystem.

This system allows low-level subsystems to make coordinated checkpoints to help reduce the possibility of a cascading rollback during recovery. Each subsystem then becomes stable in the event of a failure. Reducing the size of a coordinated checkpoint versus a system-wide coordinated checkpoint will help control the overhead of such an operation. The subsystems are loosely coupled and the coordinated subsystems do not need to communicate with other coordinated subsystems. This ensures subsystem stability and recoverability however, the addition of a single independent checkpointing system at a higher level ensures system stability.

The independent checkpointing system (IN) captures checkpoints of the coordinated subsystems. The independent checkpoint system does not know the details of the CO subsystems. It manages checkpoints for the CO subsystems as a whole. Given the loosely coupled nature of the CO subsystems the entire system is no longer subject to the possibility of a cascading rollback should a failure occur. Even if an entire subsystem fails the independent checkpoints can recover the subsystem and restore the system to a stable state.

VI. Fault Tolerance and MapReduce

MapReduce was originally developed by Google to handle large-scale data sets on distributed computing systems. Hadoop implements MapReduce as an open source project. The system consists of a map step whereby the master or controller node "maps" the input by dividing it into smaller sub-problems and distributes the tasks to worker nodes. Once the worker nodes complete their designated tasks the controller node accepts the results of the sub-problems during the reduce step and combines them in some meaningful fashion so as to provide the solution to the original problem.

MapReduce has been used by the New York Times to convert 4 TB of tiff images into PDF format by mapping conversion tasks to worker nodes and aggregating the response [11]. These tiff images comprised the New York Times articles scanned from the original paper from 1851 through 1922. Google uses MapReduce technology to process end-user queries via google.com. Google indexes the Internet using scripts known as spiders. These spiders crawl websites and save the data on those websites. The data is housed on a common file system available to the MapReduce nodes. When a user queries google.com, Google servers map the query to any number of nodes. Each node is allocated a small subset of the documents found on the Internet by the spider. The results are returned to the controller node and aggregated for display to the user. This happens hundreds of millions of times each day. O'Malley and Murthy [15] have provided research on the performance of MapReduce using Hadoop. Their data indicates a 1406 node Hadoop MapReduce cluster was able to perform the Graysort benchmark on 500 gigs of data in 59 seconds.

MapReduce and Hadoop can be used for long running calculations on large data sets such as manipulating large sets of spatial data, analyzing weather data, or modeling various data-intensive problems. Failure of a single node will cause the entire process to fail in certain workflows, as there will be a portion of data unaccounted for in the final results. Given a task of sorting many petabytes of data, a failure of a single node would leave a gap in the results. A controller can restart a failed job on an idle node or wait for the tasks to complete and remap the job to a finished node. However, on long-running jobs the delay this would introduce could be catastrophic. If the execution of a single map operation takes on the order of weeks or months restarting that failed task on a functioning node would slow the result while restarting the task on a node once it has completed its original task would double the execution time. For time-sensitive tasks this sort of fault tolerance may not yield acceptable results.

According to Kavulya et al, most jobs fail within 150 seconds of the first aborted task and a maximum error latency of 4.3 days [16]. However, the first aborted task may not occur until much further into the calculations. Additionally, the majority of errors were seen in the map phase versus the reduce phase of the operation when the available processor count is low and the impact of a fault is going to be more significant. This impact increases depending on the nature of the fault. In a two-core system, a permanent fault will reduce the available cores to one resulting in a substantial increase in running time. However, on large systems with larger number of available cores to execute tasks on, a permanent failure has a less significant impact on the execution of the overall job.

VII. N-Modular Redundant MapReduce

In order to avoid failures we propose applying the same N-modular redundancy seen in embedded systems and other fault tolerant areas to the MapReduce programming paradigm. The goal of an N-Modular Redundant MapReduce system would be to facilitate highly reliable parallel processing performed in a heterogeneous environment via commodity off the shelf hardware. The system would require no expensive or customized hardware. This system would allow for two modes of operation: Fault tolerant and highly reliable. In fault tolerant mode it would map the job to two or more nodes and simply respond with the first result. This protects against a failed node. The highly reliable mode would compare the results and return the majority. The system would allow for variable redundancy in the configuration of jobs prior to the mapping operation.

There has been work around developing NMR processors for distributed systems to address Byzantine failures [12] however these systems require specialized processors and components. Given the distributed nature and ability for MapReduce to run on heterogeneous hardware it makes the environment well suited to the overlay of an NMR system on top of existing MapReduce constructs. MapReduce is fault tolerant by design but it is unable to handle faults and still produce results in at a desired time. When calculations must by accurate such as mission or life critical application or results must be complete by a desired time, MapReduce cannot meet those requirements in the presences of faults. The goal of an NMR fault tolerant MapReduce cluster is to facilitate large data-set operations using open-source software running on commodity hardware.

In figure 3 we show the normal state of a MapReduce cluster. The controller maps tasks to nodes, the nodes complete those tasks, and the response is typically returned to the controller during the reduce phase. Figure 4 shows a failed node in MapReduce where no results from that node are ever returned to the controller. The goal of the NMR MapReduce is to mask this failure and allow for normal operation to continue without any lost time.

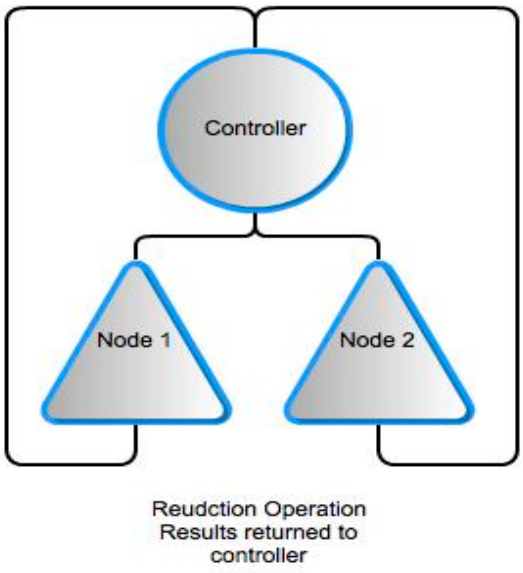

FIGURE 3 – NORMAL MAPREDUCE ARCHITECTURE

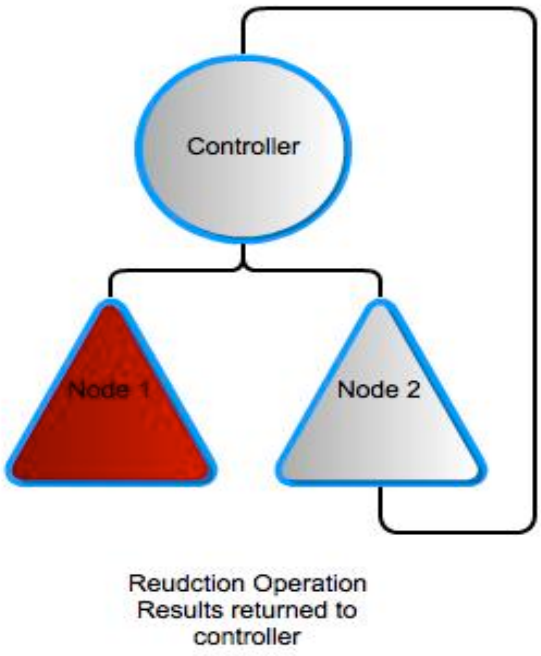

FIGURE 4 –MAPREDUCE WITH A FAILED NODE

In figure 6 we show the proposed architecture for the NMR MapReduce system. Each task is mapped to three nodes. This instance is a triple modular redundant system but the idea can scale beyond three nodes should the operator require it. Each node performs the calculations independently. The results are then sent to a voter. In the diagram the voter is logically shown as an independent node however in practice the voter could be the controller node. The voter will take the results of each node and compare them. The voter will relay the proper results to the controller based on the criteria specified in the configuration. The operator can specify how many redundant nodes the operation executes on as well as how many need to match for a consensus. For instance, the operator can specify five redundant nodes and three of those five must agree.

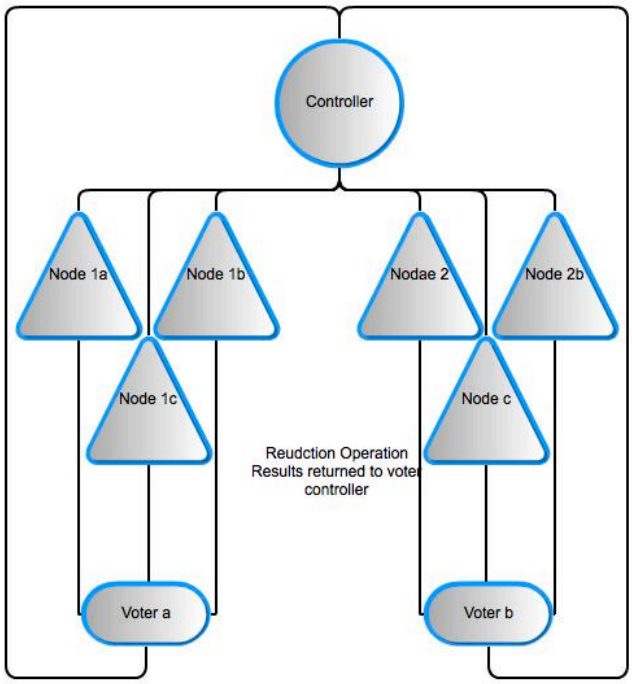

Figure 3 – Proposed NMR MapReduce cluster.

VIII. VOTING IN THE N-MODULAR REDUNDANT SYSTEM

Voting in an N-modular redundant MapReduce system can be complicated. Voting in embedded systems is simpler as the inputs are logic inputs consisting typically of 1s and 0s. An embedded voter could simply compare each signal to the rest and return the majority or for simple fault tolerance, return the first response. The voter in a higher-level MapReduce operation would have to contend with larger result sets and the comparisons could be complicated. One attempt to minimize the complication of the voting algorithm is to hash the results to a simple string value for comparison.

There are two obvious modes of operations for a NMR MapReduce cluster. The first being simple fault tolerance. A cluster could be run in a 2*N configuration such that every map task is executed on two MapReduce nodes. Given equivalent hardware, the tasks should complete in the

same time and the result is a simple first response. The voter is reduced to simply passing along the first result it receives. This system can be extended depending on the number of redundant nodes required. For calculations that simply cannot be lost the cluster could be configured in a 3*N or higher would improve the reliability. The reliability in the case of parallel execution is a simple parallel calculation as seen in equation 3 where R_n is the reliability of a given system in the TMR configuration.

$$R_s = 1 - (1 - R_1)(1 - R_2)$$

EQUATION 3 – RELIABILITY OF A PARALLEL SYSTEM

Alternatively, should highly reliable calculations be required of the system, the voter system could be employed to compare the results of each node and ensure that the system returns the majority. This would be used where component reliability is already high however incorrect results would present a significant problem for the operators.

```
1.  For each Ri in Rn
2.  Compare Ri with Ri through Rn
3.    If Ri = Rj counter = counteri + 1
4.    next j
5.  if MAX(counteri) >=MIN_CONSENSUS
6.      return Ri
7.  else
8.      return null
```

FIGURE 6 – VOTING ELECTION ALGORITHM

Figure 6 shows a simple algorithm to count the number of hashed results that are equal. There is no need to compare all values to each other simply compare each result to the remaining results and count the total number that are equal. We then return the result with the maximum number of equal results in the entire set as long as that number is greater than the minimum required consensus as specified in the job configuration. This could be as low as one if the aim is durable computing and there are complete node failures or the ceiling of n/2 for the majority to mask faults in the results.

IX. SPATIAL CONSIDERATIONS

In an attempt to facilitate maximum node availability we evaluate physical and spatial considerations for the MapReduce cluster. To maintain high availability the MapReduce system should be designed to take advantage of some spatial distribution. With modern networks capable of high-speed data transmission MapReduce nodes could theoretically be located throughout datacenters across the world. However, at a minimum, the system would need to be configured to have nodes in separate racks in the data center. Each rack should be connected to separate power systems such that if the power fails in rack the other racks continue operation. At a minimum, no more than two nodes in the NMR configuration should be in the same rack. This allows the loss of one rack in the data center due to power failure or network failure while still maintaining a running node. In a properly configured system the reduction object from the running node would be used as the only survivor of the failure and the operation would continue. Obviously this is dependent on proper power and network redundancy, which is outside the scope of this paper.

Moving beyond simple spatial considerations within the datacenter, in ultra high reliability operations the nodes could be moved to an offsite datacenter. For instance, in a cluster of 3,072 nodes, one third of those nodes would be allocated to the local datacenter while one third each could be placed in offsite datacenters to allow for the loss of two of the three datacenters completely in this system. For true operational stability and reliability however the controller node would likely need to be placed a fourth location. The main fault of this design is the single point of failure the controller node presents.

Common storage is the underpinning of MapReduce. The Hadoop file system presents all the nodes with a common view of the data. This allows the nodes to perform operations on the data without having to replicate the data. This is a major constraint on the spatial distribution of nodes, as it requires a network capable of replicating or operating on massive data sets. Within a very small geographic region this is typically not an issue as local area networks and fiber interconnections make data transfer fast and reliable. However, moving beyond the scope of a few city blocks and into the realm of distributing nodes to various states or countries would become dramatically more challenging.

X. CONTROLLER CONSIDERATIONS

At the heart of the MapReduce model is a master node or a controller. This node is responsible for the mapping operation and distributes the workload to the nodes. In the NMR MapReduce model proposed there is no consideration or controller node redundancy. The general idea is that the controller is not the same as the nodes and is, in and of itself, highly reliable. Should the controller fail for some reason, the entirety of the calculations would be lost and the entire cluster itself would completely fail. However, the system could be designed in such a way that instead of partitioning the nodes into a mirrored or NMR configuration the design would build MapReduce systems in an NMR or mirrored configuration. For instance, in a simple mirrored system, there would be two complete and functioning MapReduce clusters complete with their own controllers and own sets of nodes. These clusters would have no understanding or context of the redundancy applied. Each controller would be given the exact same tasks to complete and would then return their results independently. A voter node or array of voter nodes would than take the results and return the proper data based on the configuration. In the case of the mirror, it would simply return the first response. A single node failure at the cluster level of either of the clusters would remove that cluster from service and the computation would be dependent on the remaining operational cluster.

Applying the NMR style redundancy to the higher-level MapReduce clusters there would be three distinct and complete clusters. They would each get a complete working problem set and return responses to the voter node or nodes. The voters would then return the majority answer to the user.

Moving the redundancy from the node level to the cluster level provides obvious benefits. No longer is the controller the single point of failure. Additionally, allocating jobs could be easier since we are no longer altering the fundamental operation of a MapReduce system by injecting changes into the core of the operation. In operating redundant nodes we would need to reconfigure the code to allocate the tasks to three nodes, the controller would need to have an advanced understanding of the nodes not normally required (e.g. spatial data to distribute work based on location and machine data to match like machines). The controller would also need to understand how to perform the reduction on multiple identical datasets returned from redundant nodes. Bringing the redundancy up one level allows us to add the functionality onto existing MapReduce clusters. No longer is the off-the-shelf cluster aware of fault tolerance it simply operates as normal. In this scenario we return results to a second controller that implements our redundancy. This voter or controller would aggregate the responses from multiple MapReduce clusters and present the results. This effectively commoditizes the MapReduce cluster and allows for the fault tolerant cluster user to add clusters to the fault tolerant cluster. It's conceivable that a user at one national laboratory would be able to "borrow" another lab's MapReduce cluster for added fault tolerance. The clusters can be completely heterogeneous and unaware of the other. The operator would run software that is aware of the clusters, the software would dispatch the jobs, and finalize the responses. The biggest draw back in this scenario is the loss of common data storage.

Again, the issue of data locality becomes an issue in the redundant cluster configuration. Truly independent clusters would not share a common file system making MapReduce not a viable option. The dependence on a common file system is fundamental to the MapReduce configuration. The arrays of MapReduce clusters could be configured in a very confined geographic area such as a building or within a city block or two. This would allow for high-speed fiber interconnections and the sharing of a common file system. The nodes would be allocated into various clusters and the clusters controlled via master the proposed fault tolerant software. This provides some spatial distribution but not so much as to make the data transfer and common file system impossible.

Obviously, these are scenarios and configurations motivated by not only high-performance parallel computing but also by ultra high reliability computing. These configurations would be cost prohibitive for a sizable portion of the general high-performance user base.

XI. Cost Considerations

Any system implementing mirroring or N-modular redundancy in computations incurs a cost penalty as a result. Adding redundancy requires adding components, interconnections, additional power, cooling, and management in a MapReduce environment. The traditional high-performance computing user would not require the kinds of redundant systems described here when using MapReduce. However, for highly valuable and long running tasks, the benefits would justify the costs. Given that Hadoop MapReduce clusters can be built with simple off the shelf servers the barrier to entry is dramatically reduced. Replicating complete MapReduce clusters in a high reliability configuration would double, triple, or more the cost of the cluster however it would provide dramatic improvements in the reliability. This sort of system would be appealing to organizations that need results from large datasets within time constraints and thus cannot wait for tasks to be restarted and the delivery delayed as well as organizations for which data accuracy is or paramount importance.

XII. Conclusion

We have evaluated the widely used parallel computing fault tolerant schemes in shared memory systems as well as MapReduce. Shared memory systems make use of two different checkpointing schemes to save execution state to a stable storage and recover later in the event of a node or core failure. MapReduce is inherently fault tolerant however it often does not recover quickly as it needs to wait for a node to become free before restarting, from the beginning, the failed computation on that node. We proposed a scheme whereby the computations are started simultaneously on two, three, or more nodes and the results are either taken in a first return wins or via a voting algorithm to ensure correctness in the results. The results are hashed in this case and compared. Typically this is a majority wins scenario but the operator can configure the output of the voting mechanism for the desired reliability. Additionally, we proposed moving the redundancy from the node level up to the cluster level and creating arrays of clusters for fault tolerance. While the major constraint in the node-level configuration was introducing new code to the proven MapReduce systems, moving it to the cluster level presents the problem of the common file system.

MapReduce is a technology with substantial promise to change the way we deal with large data sets. It is a technology that is just beginning to show what it can do in business and laboratories around the world. We have shown that while it comes at a considerable cost, high-performance fault tolerant computing using MapReduce is possible in using a mirrored or NMR style configuration.

REFERENCES

[1] Srinivasan, A.; Shoja, G.C.; , "Time-cost analysis of fault-tolerant parallel programs with timing constraints," *Communications, Computers, and Signal Processing, 1995. Proceedings. IEEE Pacific Rim Conference on* , vol., no., pp.343-346, 17-19 May 1995

[2] Maier, J.; , "Fault tolerance lessons applied to parallel computing," *Compcon Spring '93, Digest of Papers.* , vol., no., pp.244-252, 22-26 Feb 1993

[3] Elnozahy, E.N.; Johnson, D.B.; Zwaenepoel, W.; , "The performance of consistent checkpointing," *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on* , vol., no., pp.39-47, 5-7 Oct 1992 doi: 10.1109/RELDIS.1992.235144

[4] Silva, L.M.; Silva, J.G.; , "The performance of coordinated and independent checkpointing," *Parallel and Distributed Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings* , vol., no., pp.280-284, 12-16 Apr 1999

[5] Whisnant, K.; Kalbarczyk, Z.; Iyer, R.K.; , "Micro-checkpointing: checkpointing for multithreaded applications ," *On-Line Testing Workshop, 2000. Proceedings. 6th IEEE International* , vol., no., pp.3-8, 2000

[6] Jiannong Cao; Yifeng Chen; Kang Zhang; Yanxiang He; , "Checkpointing in hybrid distributed systems," *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on* , vol., no., pp. 136- 141, 10-12 May 2004

[7] Oliner, A.; Sahoo, R.; , "Evaluating cooperative checkpointing for supercomputing systems," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , vol., no., pp.8 pp., 25-29 April 2006

[8] Bicer, T.; Wei Jiang; Agrawal, G.; , "Supporting fault tolerance in a data-intensive computing middleware," *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* , vol., no., pp.1-12, 19-23 April 2010

[9] Qin Zheng; , "Improving MapReduce fault tolerance in the cloud," *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* , vol., no., pp.1-6, 19-23 April 2010

[10] Goiri, I.; Julià, F.; Guitart, J.; Torres, J.; , "Checkpoint-based fault-tolerant infrastructure for virtualized service providers," *Network Operations and Management Symposium (NOMS), 2010 IEEE* , vol., no., pp.455-462, 19-23 April 2010

[11] Gottfrid, Derek (November 1, 2007). "Self-service, Prorated Super Computing Fun!". The New York Times. Retrieved May 4, 2010.

[12] I-Ling Yen; , "Specialized N-modular redundant processors in large-scale distributed systems," *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on* , vol., no., pp.12-21, 23-25 Oct 1996

[13] Bressoud, T.C.; Kozuch, M.A.; , "Cluster fault-tolerance: An experimental evaluation of checkpointing and MapReduce through simulation," *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on* , vol., no., pp.1-10, Aug. 31 2009-Sept. 4 2009

[14] Gunarathne, T.; Tak-Lon Wu; Qiu, J.; Fox, G.; , "MapReduce in the Clouds for Science," *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* , vol., no., pp.565-572, Nov. 30 2010-Dec. 3 2010

[15] O. O'Malley and A. C. Murthy, "Winning a 60 second dash with a yellow elephant," 2009. [Online]. Available: http://sortbenchmark.org/Yahoo2009.pdf

[16] Kavulya, S.; Tan, J.; Gandhi, R.; Narasimhan, P.; , "An Analysis of Traces from a Production MapReduce Cluster," *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on* , vol., no., pp.94-103, 17-20 May 2010

[17] Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G.; Kozyrakis, C.; , "Evaluating MapReduce for Multi-core and Multiprocessor Systems," *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* , vol., no., pp.13-24, 10-14 Feb. 2007

[18] Wang, Li; Ni, Zhiwei; Zhang, Yiwen; Wu, Zhang Jun; Tang, Liyang; , "Pipelined-MapReduce: An Improved MapReduce Parallel Programing Model," *Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on* , vol.1, no., pp.871-874, 28-29 March 2011

[19] Chao Tian; Haojie Zhou; Yongqiang He; Li Zha; , "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on* , vol., no., pp.218-224, 27-29 Aug. 2009

[20] Sandhya, S.V.; Sanjay, H.A.; Netravathi, S.J.; Sowmyashree, M.V.; Yogeshwari, R.N.; , "Fault–Tolerant Master-Workers Framework for MapReduce Applications," *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on* , vol., no., pp.931-933, 27-28 Oct. 2009

[21] Maier, J.; , "Fault tolerance lessons applied to parallel computing," *Compcon Spring '93, Digest of Papers.* , vol., no., pp.244-252, 22-26 Feb 1993