SAND2010-0441C

See Applications Run and Throughput Jump: The Case for Redundant Computing in HPC

Rolf Riesen
Sandia National Laboratories[*]
rolf@sandia.gov

Kurt Ferreira
Sandia National Laboratories
kbferre@sandia.gov

Jon Stearley
Sandia National Laboratories
jrstear@sandia.gov

Abstract

For future parallel computing systems with one hundred thousand nodes or more we propose redundant computing to reduce the number of application interrupts. The frequency of faults in exascale systems will be so high that traditional checkpoint/restart methods will break down. Applications will experience interruptions so often that they will spend more time restarting and recovering lost work, than computing the solution. We show that redundant computation at large scale can be cost effective and allows applications to complete their work in significantly less wall-clock time. On truly large systems, redundant computing can increase system throughput by an order of magnitude.

1 Introduction

The average number of processor cores per system on the top500 list is approaching ten thousand with the largest systems exceeding one hundred thousand cores [8]. Even though more and more of these cores are gathered on single integrated circuits, the overall component count of these enormous systems keeps increasing. With more cores, whether on a single chip or not, more memory and more supporting components are required. With an increased component count, the number of faults a system experiences increases as well.

Most large-scale parallel applications rely on checkpoint/restart techniques to recover from faults. Each fault in a component that the application is currently using, causes an application interrupt and the application aborts. Later, the application has to restart and resume from the last successful checkpoint. Several studies have shown that this approach, independent of the specific checkpoint method, does not scale beyond a few tens of thousands of nodes [9, 2].

This leads to a problem for users of extreme-scale systems: Their applications will spend most of their execution time storing checkpoints, performing restarts, and recomputing work that has been lost. For large-scale scientific applications the additional parallelism available in these systems becomes a hindrance instead of a performance boost. This is an inefficient use of resources and may forces applications to use fewer nodes than are available, thereby getting less work done, or taking longer to reach a solution.

The number of interrupts an application experiences is dependent on the Mean Time Between Failures (MTBF). An MTBF of four or five years for each node in a system seems to be the norm [12]. Applications are vulnerable not only to hardware failures, but also to software errors and environmental causes such as power outages and shutdowns due to things as mundane as the failure of a cooling fan or a clogged filter. Any such failure leads to an application interrupt and a subsequent restart.

Given a node MTBF Θ_{node} and the assumption that all nodes have the same MTBF, it is easy to compute the MTBF Θ_{sys} for an entire system consisting of n nodes [5]:

$$\Theta_{\text{sys}} = \frac{1}{\frac{1}{\Theta_1} + \frac{1}{\Theta_2} + \ldots + \frac{1}{\Theta_n}} = \frac{1}{n\frac{1}{\Theta}} = \frac{\Theta_{\text{node}}}{n} \qquad (1)$$

With a node MTBF of five years and a node count of one hundred thousand, the system MTBF is just over 25 minutes. An application spanning the entire system can expect to restart two or more times per hour. This outlook becomes even worse with larger systems and when the somewhat optimistic node MTBF of five years in our example is adjusted down. In addition, The time to write a checkpoint and to restart increases with the application

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

size. Values of tens of minutes are not uncommon at the peta or exascale. Therefore, such applications spend a significant portion of their time not doing the computations they were designed for.

That checkpoint/restart is no longer a solution for gigantic systems has been recognized [2, 10] and several research groups are working on the problem of reducing the frequency or overhead of checkpointing: [4, 9]. Our suggestion for making these huge systems practical is to use redundant computing. This approach has a long history in mission-critical systems and the time has come to apply it to large-scale High-Performance Computing (HPC) systems.

A side effect of redundant computing is that result verification can be done at no additional cost. The number of cores per socket keeps increasing and main memory capacity is also growing. In some main memories, error correcting codes are employed. However, CPU registers, caches, and the buses between these building blocks are largely unprotected and may introduce silent errors. Redundant computing can detect these errors and abort an application when an error occurs. With triple redundancy, error correction becomes an option.

In Section 2 we propose how redundant computing can be employed in HPC systems and why it reduces the number of interrupts an application experiences as the node count goes up. We analyze the payoff in Section 3 by comparing the modeled wall-clock time of an application with and without redundant nodes. We draw our conclusions in Section 4.

2 Redundant Computing

Redundant computation increases an application's resilience to faults by increasing the time between application interruptions; i.e., it increases the application Mean Time Between Interrupts (MTBI) Θ_{app}. In redundant computation each process is replicated a number of times throughout the system. Individual components and nodes will fail, but an application will continue without interruption, provided that, in a bundle of replicated processes, at least one of them is still functioning.

The potential benefits of redundant computation can be illustrated using a generalization of a common problem in probability theory called the birthday problem [7]. The birthday problem is concerned with the expected number of people needed to find two with the same month and day of birth. The birthday problem result is used in the analysis of many problems, including collisions and chaining in hashes [6].

We generalize the results of this problem to describe the impact of redundant computing on application fault tolerance and the increase in MTBF of our redundant system. If we consider each of the replicated bundles of nodes to be a bin with a capacity equal to the number of replicas, then asking how many faults this new system can handle without interruption is equivalent to asking what is the expected number of throws of random balls until one bin has been filled to capacity. In the case of two replica per process, the birthday problem tells us that the expected number of throws is $O(\sqrt{n})$ (where n is the number bins or unique processes). More precisely, the average number of faults F our redundant system of size n can absorb, assuming double redundancy, is [6, 3]:

$$F(n) = 1 + \sum_{k=1}^{n} \frac{n!}{(n-k)! \cdot n^k} \tag{2}$$

Figure 1 shows a plot of Equation 2 as a function of the number of nodes. From this figure we see the well known result for the birthday problem for $n = 365$ (around 24.16 people). We also see that adding redundant nodes to our system dramatically increases its ability to absorb faults, thereby increasing the effective MTBF of the application. For example, for $n = 200,000$ nodes, on average, we can sustain 561 faults before our application will be interrupted. Therefore, the MTBF will increase by a factor of 561 in the redundant case over the non-redundant case.

While the analysis above outlines the benefits of redundant computation, it does not indicate the performance overhead of such a mechanism. Performance overhead includes keeping the replica state consistent as the application progresses. To evaluate the overhead of the replica consistency we built a prototype library that handles the replica coordination protocol. This library is implemented at the MPI profiling layer and intercepts all MPI calls from the application. The simple protocol used by this library converts all messages between a source and a destination to message exchanges between: 1) the source and destination, and 2) between the redundant partners of the source and destination, assuming they exist.

Since the consistency protocol doubles message traffic in the system, point-to-point micro benchmark performance suffers. For higher level benchmarks and applications the impact is, in general, minimal. Figure 2 shows the overhead of the consistency protocol for HPCCG. The HPCCG mini-application, part of the Mantevo project [11], is a simple sparse conjugate gradient solver designed to capture an important component of Sandia National Laboratories production workload. The majority of its runtime is spent performing sparse matrix-vector multiplies, where the sparse matrix is encoded in compressed row storage format. The interprocessor communication consists of nearest neighbor boundary information, in addition to global `MPI_Allreduce` operations required for the scalar computations in the CG algorithm. As the figure shows, the performance impact of the pro-

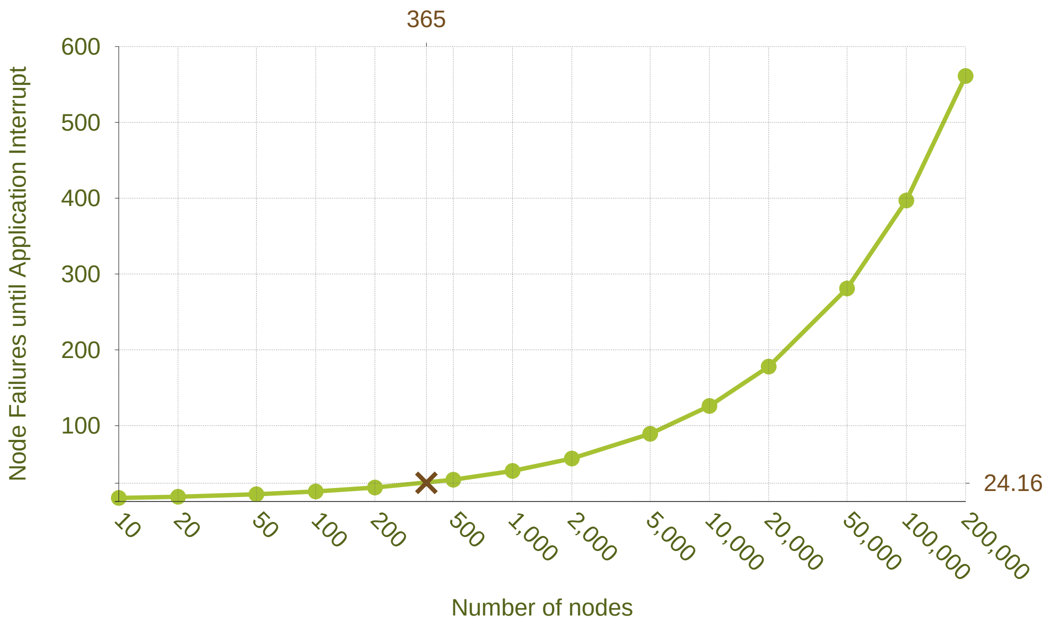

Figure 1: Expected number of node failures before an application interrupt in a system with redundant nodes. Numbers are calculated using the birthday problem Equation 2.

tocol on HPCCG when using redundant nodes is minimal. For applications where the additional bandwidth is an issue, alternative consistency protocols are being investigated.

3 Discussion

Initial reactions to applying redundancy to HPC systems often concern cost. This includes the cost of acquiring twice as many compute nodes and supplying twice as much power and cooling to the system. Another cost of concern is the software performance overhead associated with enabling redundant computing. Finally, but not least is the notion that purchasing the largest and most expensive computer in the world should not carry the stigma of getting at most half of the peak performance of such a system.

Acquisition and power costs must be weighed against the time for an application to reach a solution. If, due to application interrupts, a solution takes 200 hours to compute, but the same solution can be attained in 25 hours by using redundant computing, then the extra cost becomes negligible. The throughput of this example improves by a factor of eight, dwarfing the doubling in cost.

John Daly has proposed Equation 3 to calculate the execution time of an application [1]. Where $T_w(\tau)$ is the total wall-clock time when using the checkpoint interval τ. Θ is the MTBF for the application and T_s is the solve time; i.e, the amount of time required to complete the assigned work. δ is the time it takes to write a checkpoint, and R is the time to restart after an application interrupt.

$$T_w(\tau) = \Theta_{\text{app}} \cdot e^{\frac{R}{\Theta_{\text{app}}}} \left(e^{\frac{\tau+\delta}{\Theta_{\text{app}}}} - 1 \right) \frac{T_s}{\tau} \quad \text{for} \quad \delta << T_s \tag{3}$$

We can use Equation 3 to calculate how long an application will need to execute to solve a problem. We can perform this calculation for an increasing number of nodes and then compare it to the case where we use redundant nodes. Figure 3 shows the result of one such calculation. For this figure we assumed a perfectly weak-scaling application that requires $T_s = 168$ hours to complete its work. We chose a fixed checkpoint time $\delta = 5$ minutes and fixed the time to restart at $R = 10$ minutes.

For each node size we calculated the optimal checkpoint interval τ according to Equation 38 in [1]. The checkpoint interval τ is dependent on the MTBI Θ_{app} of the application. For a given number of nodes we used Equation 1 to calculate the system MTBF based on a node MTBF of five years. We use $T_w(\tau, n)$ to mean the wall-clock time of an application when run on n nodes. The dark green line in Figure 3 shows that the amount of time required to complete the application increases exponentially with the number of nodes used.

We then repeated these calculations for the case with redundant nodes. For a given number of nodes we used the same checkpoint interval τ as for the non-redundant case. That is not entirely correct, since Daly's calcula-

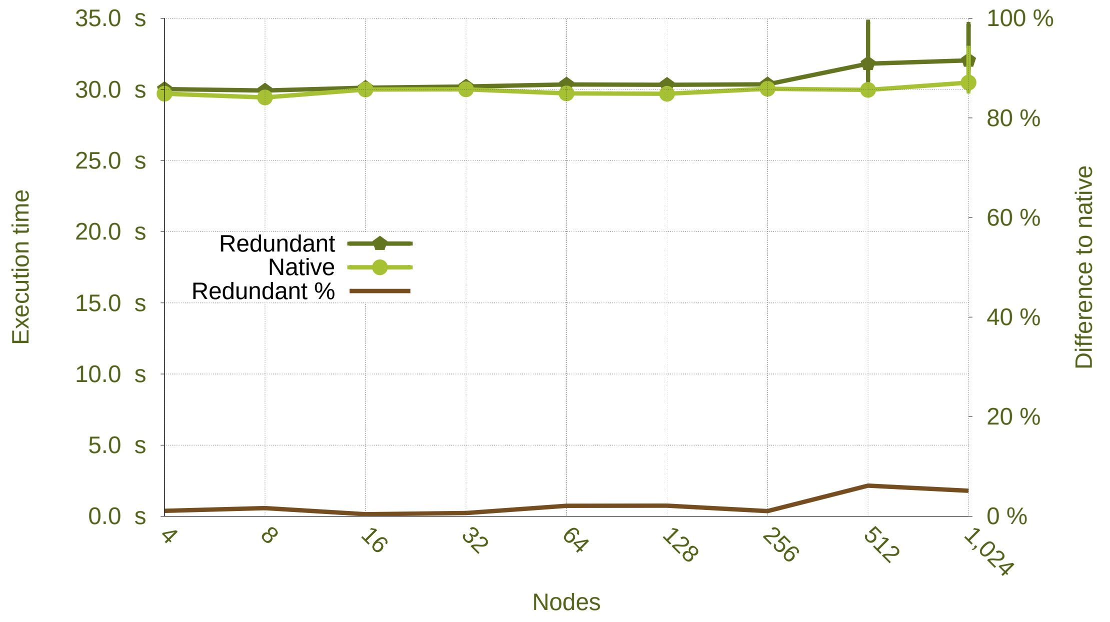

Figure 2: Performance and overhead of HPCCG application. The number of nodes on the *x*-axis is from the applications' perspective. Twice that many nodes are used to provide full redundancy.

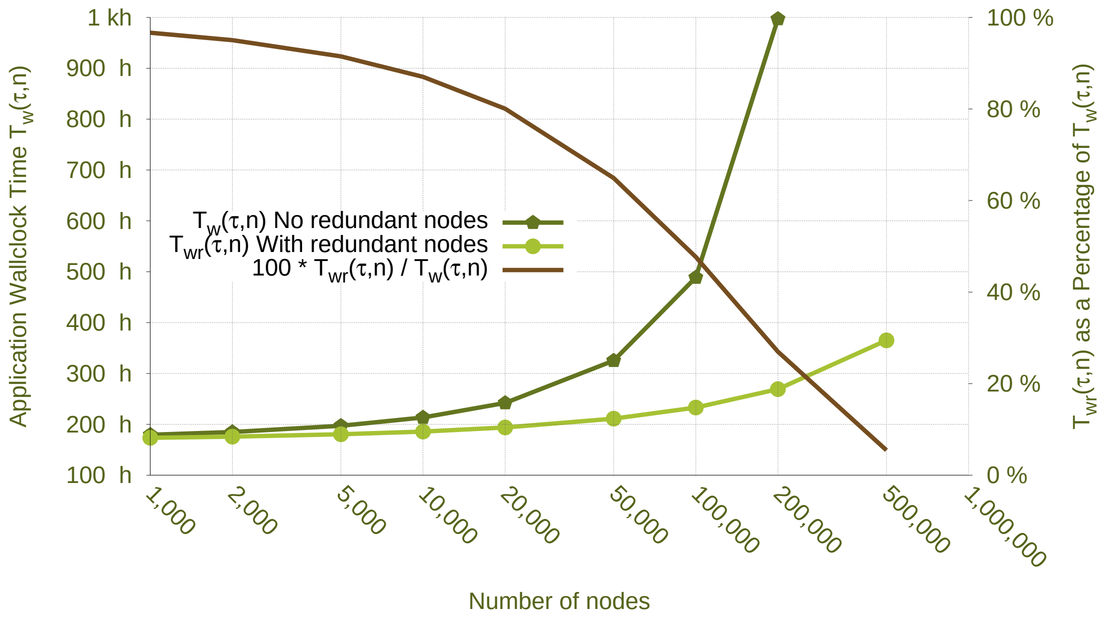

Figure 3: Comparing application wall-clock time with and without redundant nodes. When redundant computing consumes less than 50% of the time of a non-redundant run, then the extra cost of redundant computing may pay off.

tion of the optimal checkpoint interval may not be suitable for a redundant system. Since redundancy improves the MTBF of a system, our τ should probably be higher. Using the same τ as for the non-redundant case gives us a worst case scenario because we may be checkpointing too frequently.

We use $T_{\mathrm{wr}}(\tau, n)$ to mean the wall-clock time of an application when run on $2 \cdot n$ nodes (n nodes plus n redundant nodes). To compute $T_{\mathrm{wr}}(\tau, n)$ we need the MTBF of an application running on a system with redundant nodes. The birthday problem, Equation 2, gives us the ratio of faults to interrupts in a system with n nodes. When we multiply that ratio with the system MTBF of a non-redundant system (Equation 1), we can calculate the application MTBF:

$$\Theta_{\mathrm{app}} = \Theta_{\mathrm{sys}} \cdot F(n) = \frac{\Theta_{\mathrm{node}}}{n} \left(1 + \sum_{k=1}^{n} \frac{n!}{(n-k)! \cdot n^k} \right) \quad (4)$$

Using the values from Equation 4 in Equation 3 lets us calculate the wall-clock execution time for an application on a system with redundant nodes. The light green line in Figure 3 shows $T_{\mathrm{wr}}(\tau, n)$ as a function of the number of nodes. Running on larger number of nodes still introduces overhead due to the increased number of faults in the system. However, it is substantially less than the case without redundant computation. The brown curve in Figure 3 is the percentage of the wall-clock time with redundant computation of the wall-clock time without redundancy: $\frac{100 \cdot T_{\mathrm{wr}}(\tau,n)}{T_{\mathrm{w}}(\tau,n)}$. When that curve drops below 50%, using twice as many nodes for redundant computing may pay off. In Figure 3 that happens for 100,000 nodes and more.

Figure 3 shows that the benefit of redundant computation does not come into play until we reach a large number of nodes. The exact number depends on the node MTBF Θ_{node}, the time to write a checkpoint δ, and the time to restart R. We used fixed values for the latter two, but in a real system they are likely to increase with the number of nodes being used. For the redundant case it seems that we should be able to increase the checkpoint interval τ and further improve those results. Note that the overhead of the redundancy protocol is not included in Figure 3. Beyond five-hundred-thousand nodes, for which we do not have performance measurements, it is likely to have some impact. Nevertheless, current measurements indicate that it will be minimal when compared to the benefits of redundant computing.

In Section 2 we mentioned that there is significant overhead for point-to-point micro benchmarks, and have shown that the HPCCG application kernel suffers very little from the redundant computing overhead. Therefore the main cost of redundant computing is the doubling in the number of nodes and the resources, such as power and cooling, that go along with it. We have shown in this section that this cost is justified by the faster execution of applications, resulting in higher throughput of the system.

Note, Redundant computing is not a substitute for checkpoint/restart. It reduces the number of interruptions an application experiences and therefore reduces the overhead of checkpointing. Redundant computation also enables the detection of silent errors. These are errors that occur in registers, caches, and data paths that are not protected by error detection or correction codes. These errors are often silent because they produce a corrupted result but do not further disrupt the computation. Most applications assume that calculations done by the hardware are correct and do not or cannot check for invalid results. Double redundancy lets us flag silent errors, and triple redundancy would let us correct silent errors.

4 Conclusions

In this paper we take the position that redundant computing is a viable solution to the problem of the ever increasing number of application interruptions as massively parallel systems get larger. The problem, applications spending more time writing checkpoints and restarting rather than performing productive computations, is real and needs to be addressed. At first glance, redundant computing seems to waste half of the compute resources in a system. We have shown that redundant computing actually reduces the time to solution by such a large factor that not using it is actually a bigger loss in system throughput. There is a comparatively small overhead to pay to enable redundancy in software at the user level, and it does not completely replace checkpoint/restart since it only reduces the number of application interrupts; not eliminating them completely.

For future work we are considering improving the consistency protocol used by our library and reduce protocol overhead. Although the overhead seems to be acceptable for applications at the few-thousand-node scale, it may become worse at much larger scales. We are also looking at applying this idea to non-MPI applications, especially in a multicore world. Finally, we are looking at approaches where the data is replicated but the redundant computation is only carried out when necessary.

References

[1] J. T. Daly, *A higher order estimate of the optimum checkpoint interval for restart dumps*, Future Gener. Comput. Syst. **22** (2006), no. 3, 303–312.

[2] E.N. Elnozahy and J.S. Plank, *Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery*, Dependable and Secure Computing, IEEE Transactions on **1** (2004), no. 2, 97–108.

[3] Philippe Flajolet, Peter J. Grabner, Peter Kirschenhofer, and Helmut Prodinger, *On Ramanujan's Q-function*, Journal of Computational and Applied Mathematics **58** (1992), 103–116.

[4] Rinku Gupta, Pete Beckman, Byung-Hoon Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar Panda, Andrew Lumsdaine, and Jack Dongarra, *Cifts: A coordinated infrastructure for fault-tolerant systems*, ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing (Washington, DC, USA), IEEE Computer Society, 2009, pp. 237–245.

[5] Dimitri B. Kececioglu, *Reliability engineering handbook*, vol. 2, DEStech Publications, Inc, May 2002.

[6] Donald E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[7] Frank H. Mathis, *A generalized birthday problem*, SIAM Review **33** (1991), no. 2, 265–270. MR 1 108 591

[8] Hans Meuer, Erich Strohmaier, Horst Simon, and Jack Dongarra, *Top 500 supercomputer sites*, `http://www.top500.org/`, November 2009.

[9] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth, *Modeling the impact of checkpoints on next-generation systems*, 24th IEEE Conference on Mass Storage Systems and Technologies, September 2007, pp. 30–46.

[10] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta, *Performance implications of periodic checkpointing on large-scale cluster systems*, Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18, 2005, p. 299.2.

[11] Sandia National Laboratory, *Mantevo project home page*, `https://software.sandia.gov/mantevo`, Nov. 6 2008.

[12] Bianca Schroeder and Garth A. Gibson, *A large-scale study of failures in high-performance computing systems*, Proceedings of the International Conference on Dependable Systems and Networks (DSN2006), June 2006.