Highly Scalable Checkpointing for Exascale Computing

Christer Karlsson
Colorado School of Mines
Golden, CO, USA
Email: ckarlsso@mines.edu

Zizhong Chen
Colorado School of Mines
Golden, CO, USA
Email: zchen@mines.edu

***Abstract*—A consequence of the fact that the number of processors in High Performance Computers (HPC) continues to increase is demonstrated by the correlation between *Mean-Time-To-Failure*(T_{MTTF}) and application execution time. The T_{MTTF} is becoming shorter than the expected execution time for many next generation HPC applications. There is an ability to handle failure without a system-wide breakdown in most architecture, but many of the applications do not have a built-in ability to survive node failures. The purpose of this paper is to present an approach to develop a highly scalable technique to allow the next generation applications to survive node and/or link failure without aborting the computation. We will develop several strategies to improve the scalability of diskless checkpointing. The technique is scalable in the sense that when the number of processes increases, the overhead to handle k failures on p processes should remain as constant as possible. We will present the proposed technique, initial results together with remaining objectives and challenges.**

***Keywords*-diskless checkpointing; exascale; multi failure; topology aware;**

I. Introduction

The applications of today are driven toward attempting larger simulations and working with larger datasets. This has pushed the size of the high performance computers from tens of thousands of cores into the realm of hundreds of thousands. The fact that the T_{MTTF} has now become significantly shorter than the execution times is a reality which needs to be addressed [1].

The Southern California Earthquake Center introduced a simulation named TeraShake. This large scale earthquake simulation is designed to run on 40k BlueGene processors creating an output of 47 TB of time-varying volumetric data with up to 400,000 files from a single simulation [2]. The scientists desire a higher level of detail and accuracy in the simulations resulting in new simulations that are designed to run for long periods of time. To avoid restarting the computation after a failure, they must rely on the simulation's ability to continue execution regardless of failures. Even though most of today's architectures are robust enough to handle process failures without suffering a complete system failure, the techniques available to application developers to provide fault tolerance are usually limited to *checkpoint/restart*. The consensus is that this is inadequate for future needs on large scale simulations [1].

A possible solution would be an optimized diskless checkpointing from the application level that allows recovery from multiple failures. The rationale is that diskless checkpointing, in comparison to the traditional methods, has less overhead on stable media. We therefore want to focus our efforts on improving the scalability of diskless checkpointing and optimizing the checkpoint encoding with respect to both the latency and the bandwidth. We will also develop techniques to detect network topology and effectively take advantage of the detected network topology information to achieve lower overhead and higher scalability.

II. Related Work

A. Diskless Checkpointing

An application level checkpoint library would be a desirable tool to use when building any fault tolerant application. The goal is to design and implement a fully scalable algorithm using the diskless checkpointing concept where each process creates a local checkpoint and then stores an encoding of these local checkpoints. The fault tolerance overhead for the diskless checkpointing is low. On a parallel system let p be the number of processors, and let m represent the size in bytes of the checkpointing on each process. Further, let

978-1-4244-6534-7/10/$26.00

α be latency and $\frac{1}{\beta}$ be the network bandwidth. Then, for a binary-tree based encoding where γ is the rate in bytes per second for which the sum of two arrays can be calculated, the overhead can be approximated as follows [1]: $T_{diskless} \approx 2\lceil \log p \rceil \cdot ((\beta + \gamma)m + \alpha)$

B. Pipelined Encoding Algorithm for Diskless Checkpointing

The disklessless method advances the scalability of checkpointing dramatically on parallel and distributed systems. The main problem remains that the overhead to perform one checkpoint increases logarithmically with the number of processors due to the fact that $T_{diskless} \approx 2\lceil \log p \rceil \cdot (\beta + \gamma)m$.

Our attempt to overcome this problem is to implement a *pipelined encoding algorithm*. The algorithm was described by Chen and Dongarra [1]. It builds on the concept of segmentation relating to the messages and a simultaneous non-blocking transmission and reception of data. This allows a more complete use of the full duplex capability of the connected links in the parallel system and masks the inherent processor and network latencies.

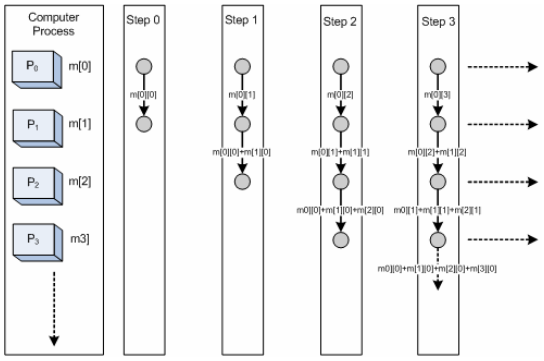

Figure 1. Checkpointing using Pipeline Algorithm

Let there be $p-1$ processors for computation and a single checkpoint processor. Further, denote the data on the i^{th} processor with $m[i]$, where $i = 0, 1, 2, \ldots, p-2$. The checkpoint encoding should at each checkpoint calculate:

$$\sum_{i=0}^{p-2} m[i]$$

and then store this value on the checkpoint processor.

The pipelined encoding algorithm in a p-processor system works as follows. First, create a chain consisting of the checkpoint processor and all $p-1$ computational processors. Second, chunk the data on each computational processor. For example, let the data on each processor be divided into n segments of size s. Let, $m[i][j]$ denote the j^{th} segment of $m[i]$. Thereafter,

$$\sum_{i=0}^{p-2} m[i] = \sum_{i=0}^{p-2} \sum_{j=0}^{t-1} m[i][j]$$

is calculated in a pipelined fashion. Finally, when the j^{th} segment of encoding

$$\sum_{i=0}^{p-2} m[i][j]$$

is available, it is sent and stored at the checkpoint processor.

III. Proposed Approach

The main goal of our research is to improve the scalability of checkpointing. The first step is to implement and test the *Pipelined Encoding Algorithm for Diskless Checkpointing*. This will give us some baseline data. The encoding and implementation will then be optimized. The first step of the optimization will be to change the message passing from a linear pipeline to a double binary tree. Our hope is to retain the simple implementation coupled with a good time-performance. The next step is to introduce an *Error Correction Code* (ECC) to enable the algorithm to recover multiple simultaneous process failures. This ECC has to be constructed in such a way that it can handle variables of any type. Also, some types of binary ECC will most likely be needed. The final step is to to look at the topology and investigate what can be done to improve the overall latency and scalability.

A. Checkpoint Encoding Based on Double Tree Algorithm

The linear pipeline has a large latency drawback. Its best performance is therefore when $m \gg p$. Our hope is to keep the broadcast time of the *Pipeline Encoding Algorithm* while improving on the the latency problem. A logical solution would be to implement a *Two Pipelined Binary Tree* as described by Sanders [3].

This algorithm should inherit both the low latency of a pipelined binary tree and the time complexity that is achieved with a linear pipeline. Two binary trees T_1 and T_2 are constructed. The construction is then completed by connecting the two trees at node $p-1$

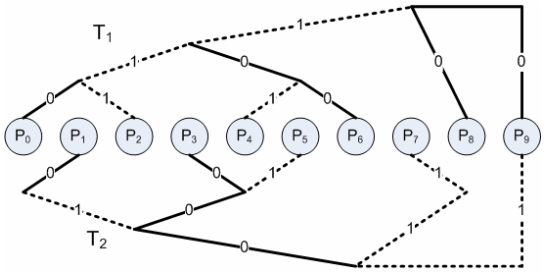

Figure 2. Double Tree

(see fig. 2). The checkpointing is done by pipelining half the data through each tree. With bidirectional communication it is possible to send data over both trees at the same time. The overall checkpointing time is estimated to be:

$$\begin{aligned} & 2\Big[2(h-1)\alpha + 4\sqrt{(h-1)\alpha}\sqrt{(\beta+\gamma)m/2} \\ & \quad +2(\beta+\gamma)m/2\Big] \\ = \;& 4(h-1)\alpha + 4\sqrt{(h-1)\alpha}\sqrt{(\beta+\gamma)m/2} \\ & \quad +(\beta+\gamma)m \end{aligned}$$

where h is such that $2^h \geq p-1$.

B. Multiple Simultaneous Process Failures

To handle multiple simultaneous process failures some kind of ECC is necessary. There are several workable routines such as the Reed-Solomon codes, BCH codes, the binary Golay code, the binary Goppa code or the Viterbi decoder. At this time it remains undecided as to which scheme will be used. One approach would be to view the Reed-Solomon code as a weighted checksum [4]. Let each processor create a local in-memory checkpoint. Then construct k equalities by storing weighted checksums of the local checkpoints into k checkpoint processors.

Should f failures occur (where $f \leq k$), the k equalities can then be used as k different equations with f unknowns. By an appropriate assignment of weights to the checksums, the data on f lost processors can be recovered through the solution to the equation system of the k equations. There is an issue however, because the number of processors is usually very large. The effect of this is that a simple weighted checksum code might not be sufficient as the overhead could increase significantly. Other potential schemes shall be investigated before committing to further development.

C. Topology Aware Checkpoint Encoding

The performance of MPI operations is heavily effected by the network's physical topology. We know that topology specific communication algorithms often outperform the communications protocols that are unaware of the topology [5]. There are implemented solutions to solve this problem. Automatic routines that take the topology information as an input and produces a topology specific MPI routines is an example. There are also routines that automatically discover the switch level topology through end-to-end measurements. This is due to the fact that the switch level topology is often unknown on most clusters [5]. Another approach has been to implement process cooperation algorithms that are easily and efficiently constructed on-line and still close to optimal in one-port fully-connected systems [6].

The decision has yet to be made as to what approach will be used, but we are aware of the fact that to achieve the highest performance the topology must be taken into account.

IV. Initial Results

All the experiments were performed on a cluster of 23 machines with 2x Dual Core Opteron 2218 2.6GHz. Each node on the cluster has 8 GB of memory and runs the Linux operating system with `Open MPI`. The nodes are connected with fast Ethernet (100Mbps) switched network links. The timer that was used in all measurements was the `MPI_WTime()`.

Three sets of experiments were conducted. The first experiment measures the communication overhead of both the `MPI_Reduce()` and the `Pipelined_Reduce()` regarding the message size. This experiment was setup over 16 processors by creating an array of `long int` with the amount elements set such that the total data equaled the predetermined message size. One of the functions was then randomly selected to be first. This was done to avoid that one function always established the communication. The time was then measured fifteen times and an average was calculated for each function and message size. The results indicate that for the built-in `MPI_Reduce()` there is an exponential relationship between the message size and the time it takes to complete the function. The pipelined function not only sends the message much faster but the exponential relationship seems to be less significant.

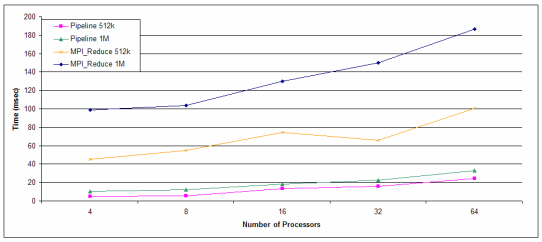

Figure 3. Overhead vs. Number of Processors

The second experiment measures the communication overhead of both the `MPI_Reduce()` and the `Pipelined_Reduce()` in respect of the number of processors. We used five different amounts of processors (4,8,16,32 and 64 on 3,6,9,12,18 and 21 nodes) and each test was also conducted on different message sizes. Once again, the starting function was randomly chosen and the time was an average of fifteen measurements. Figure 3 shows that there is an increase in time with an increase in processors. The increase is not as drastic for the `Pipelined_Reduce()` as it is for the built-in `MPI_Reduce()`, but it is still present.

The third experiment tested the checkpoint overhead, using only the pipeline-function. This function was placed in a program that executed 5000 iterations and checkpointing was done through the pipeline-function every 100^{th} iteration. Each test was conducted on different message sizes, and each time was averaged over ten measurements.

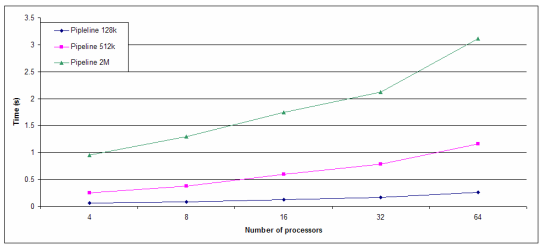

Figure 4. Checkpoint Overhead

V. Conclusion

This paper has presented an approach to create a scalable environment where the application will have the ability to handle k failures on p processes. The implementation strategy used was diskless checkpointing wherein we implemented and measured the overhead for a *Pipeline Encoding Algorithm* and a simple *Weighted Checksum Scheme*. From this base-implementation incremental improvements will be completed and augmented with an encoding algorithm which allows the application to survive k failures in p process. It shall be shown that an increase in p does not significantly increase the overall overhead.

Suggested improvements are to replace the *Pipeline Encoding Algorithm* with a *Two Pipelined Binary Tree*. Then, implement an ECC that allows recovery from multiple failures, and finally, account for the topology to reduce the latency overhead. There is also a need to create an interface which increases ease and usability for the application developer to implement the diskless checkpointing on all dynamic data they so desire.

References

[1] Z. Chen and J. Dongarra, "Highly scalable self-healing algorithms for high performance scientific computing," *IEEE Transactions on Computers*, vol. 58, pp. 1512–1524, 2009.

[2] Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. Day, Y. Hu, J. Zhu, A. Majumdar, and T. Jordan, "Enabling very-large scale earthquake simulations on parallel machines," in *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I, May 27–30, Beijing, China*. Springer-Verlag, 2007, pp. 46–53.

[3] P. Sanders, J. Speck, and J. L. Träff, "Full bandwidth broadcast, reduction and scan with only two trees," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, (PVM/MPI'07), Sep. 30 – Oct. 3, Paris, France*, vol. 4757. LNCS, 2007, pp. 17–26.

[4] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Softw. Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[5] J. Lawrence and X. Yuan, "An mpi tool for automatically discovering the switch level topologies of ethernet clusters," in *22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS'08), Apr. 17–18, Miami, Florida USA*. DBLP, 2008, pp. 1–8.

[6] B. Jia, "Process cooperation in multiple message broadcast," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, (PVM/MPI'07), Sep. 30 – Oct. 3, Paris, France*, vol. 4757. LNCS, 2007, pp. 27–35.