Low-overhead diskless checkpoint for hybrid computing systems

Leonardo BAUTISTA GOMEZ[1,3], Akira NUKADA[1], Naoya MARUYAMA[1],
Franck CAPPELLO[3,4], Satoshi MATSUOKA[1,2]
[1]*Tokyo Institute of Technology* - [2]*National Institute of Informatics*
[3]*INRIA* – [4]*University of Illinois*

Abstract*—As the size of new supercomputers scales to tens of thousands of sockets, the mean time between failures (MTBF) is decreasing to just several hours and long executions need some kind of fault tolerance method to survive failures. Checkpoint\Restart is a popular technique used for this purpose; but writing the state of a big scientific application to remote storage will become prohibitively expensive in the near future. Diskless checkpoint was proposed as a solution to avoid the I/O bottleneck of disk-based checkpoint. However, the complex time-consuming encoding techniques hinder its scalability. At the same time, heterogeneous computing is becoming more and more popular in high performance computing (HPC), with new clusters combining CPUs and graphic processing units (GPUs). However, hybrid applications cannot always use all the resources available on the nodes, leaving some idle resources such us GPUs or CPU cores. In this work, we propose a hybrid diskless checkpoint (HDC) technique for GPU-accelerated clusters, that can checkpoint CPU/GPU applications, does not require spare nodes and can tolerate up to 50% of process failures with a low, sometimes negligible, checkpoint overhead.

I. Introduction

The huge amount of computational power that current scientific problems require has led the scientific community to develop high performance computers that can execute more than one peta of floating operations per second (FLOP) using tens of thousands of processors [1]. While the reliability of a single processor guarantee several years of usage, the reliability of a system with hundreds of thousands of processors can only guarantee a MTBF of several hours.

In these circumstances it is necessary to use some fault tolerance technique for long executions. In HPC, checkpoint/restart is the most used technique to deal with failures. In the classic disk-based checkpoint strategy, the processes coordinate to create a coherent cut of the parallel application and then they save their checkpoint data on remote stable storage. Since new supercomputers can treat more data and scientific applications are getting larger, the amount of checkpoint data is also getting larger creating an I/O bottleneck when writing that data on remote storage. The overhead of this technique is about 20% of the execution time and is increasing dramatically [2, 3]. Also, even for low memory consuming applications, the checkpoint time can increase dramatically depending on the pattern used to write the checkpoint data on the file system [4].

In order to avoid this I/O bottleneck, diskless checkpoint proposes to save the checkpoint data on memory and to use replication or encoding techniques with spare nodes to guarantee the reliability of the system. However, the spare nodes, the low fault tolerance capability (i.e. one failure tolerated for XOR encoding) and the hardly scalable encoding algorithms discourage the use of this technique. In our previous work [5] we proposed a fault tolerant model able to tolerate up to 50% of process failures without spare nodes using solid-state drives (SSDs) and a scalable encoding algorithm. Using this model the checkpoint takes several minutes which is a tolerable overhead for data intensive application when the checkpoint interval is about one hour; but in the future the checkpoint frequency need to be increased to deal with the decreasing MTBF of next generation supercomputers. Furthermore, in GPU-accelerated clusters, the fault tolerance issue remains almost unexplored and there is not standard solution to checkpoint GPU applications today.

A. Contributions

In this work we propose a hybrid diskless checkpoint technique and we evaluate our technique in a heterogeneous architecture using a Nbody simulation and the Himeno benchmark which has been implemented on CPUs [6] and GPUs [7].

- We explain how one can do a better usage of the idle resources in GPU-accelerated clusters to implement scalable low-overhead diskless checkpoint.
- Our proposed technique can checkpoint CPU and GPU applications at a user level, checkpointing only the necessary data and delegating the encoding process to idle resources present on the node.
- We present a strategy for hybrid architectures where the encoding process is done in parallel with the application execution decreasing the checkpoint overhead at the point to be negligible in some cases, as presented in our evaluation.
- We propose a hybrid node manager with our in-memory checkpoint that reduces the diskless checkpoint process to a simple in-memory copy of the checkpoint data.
- We evaluate our technique in different configurations with a high checkpoint frequency and we show that for some configurations, it decreases the checkpoint overhead significantly in comparison with a non-hybrid diskless checkpoint technique.

The rest of this paper is organized as follows. Section II present the motivations of this work. In section III we explain the GPU architecture and programming model. Section V explains our hybrid diskless checkpoint technique, which is

978-1-4244-8520-8/10/$26.00 ©2010 IEEE

evaluated in section VI and finally section VII concludes this paper.

II. Motivations

A large study [2, 3] done at Los Alamos National Laboratory (LANL) over nine years shows that the frequency of failures increases with the size of the system. In this study we can see that 60% of the failure have as root cause the hardware and on 25% of the cases the root cause is software. The reliability of a single processor can guarantee its usage for years without failures, but a system with tens of thousands of these processors will have a mean time to failure (MTTF) of only a couple of hours [9].

Another large-scale field study made in cooperation with Google [10] analyses the frequency of memory errors in production environments. There are several causes for errors in dynamic random access memory (DRAM) such as electrical or magnetic interference, hardware problems or corruption between the processors and the memory. Current clusters have error correcting codes (ECC) making the nodes able to detect and correct one or several bit errors; but in some cases, when the ECC cannot correct the error, the entire node will shutdown causing multiple process failures.

A. Disk-based checkpoint

Most of current large clusters and supercomputers are composed of hundreds or thousands of multi-socket computing nodes connected by a network such as Infiniband and communicate with the parallel file system trough dedicated I/O nodes. There are several different file systems such as Panasas file system [11], GPFS [12] or Lustre [13] that can reach several tens or hundreds of GB/s of I/O bandwidth. However, the constantly increasing system size is making the checkpoint data of current applications so large, that the I/O bandwidth becomes a bottleneck at the checkpoint time, generating more than 20% of checkpoint overhead [9].

B. Heterogeneous computing

During the last five years, accelerators have taken an important place in HPC. Several supercomputers in the world use accelerators to improve the performance of highly parallel applications and the energy efficiency of the machine. A remarkable example of these heterogeneous systems is the LANL's supercomputer, Roadrunner [1], which became the first supercomputer to reach the Petaflop barrier in 2008. Roadrunner is composed by AMD Opteron processors and IBM/Sony/Toshiba PowerXCell accelerators.

Another example is the Tokyo Tech's supercomputer Tsubame with AMD Opteron 280s processors and nVidia Tesla GPUs cards [16]. The GPUs have increased the performance of several libraries and applications [7] such as the dense linear algebra (Linpack) [18] and the sparse finite difference Himeno benchmark [6] in single precision. GPUs have become an important part of commodity clusters, in particular with the last generation of general purpose GPUs (GPGPUs) such as Tesla and Fermi [20]. The Fermi GPUs not only achieve good performance in single-precision but also in double-precision. Moreover, this new GPU card include ECC to tolerate bit-flip errors. In addition, the compute unified device architecture (CUDA) developed by nVidia really ease the programmer work to implement algorithms in those GPUs. For these reasons, GPU-accelerated clusters are getting more popular in HPC.

However, not all the scientific applications can achieve good performance on current GPUs. Most of data-intensive applications will have a large communication overhead when computing on GPUs. These applications will rather use CPUs on heterogeneous clusters leaving the GPU cards idle. On the other hand, GPU applications usually don't use all the CPU cores present on the nodes. Most of the GPU applications usually launch one MPI process per GPU or even one MPI process to manage several GPUs. Since most of the heterogeneous architectures have more CPU cores than GPUs in each node and GPU applications usually use one CPU core per GPU, a wide variety of hybrid applications can achieve good performance on heterogeneous clusters while leaving some idle resources on each node. These idle resources can be used for other purpose such as fault tolerance. In this work we propose an efficient way to use these resources to achieve a low-overhead checkpoint in order to deal with the decreasing MTBF of future supercomputers.

When several techniques propose a way to accommodate multiple applications to use the idle cores, parallel GPU applications are not suitable for such best effort policies. Although some CPU cores are really idle in those applications, providing those cores to other users may cause a large overhead due to contentions at memory bus, network, and so on. Important jobs, which are expected to be checkpointed, should use the compute nodes exclusively. This is why many production clusters, such as Tsubame, have several queues with different priority levels and some of them use the compute nodes exclusively.

Furthermore, even in the scenario where there are not idle resources, our technique is still useful. Indeed, we can imagine that the users may sacrifice a thread per node to increase the reliability of the execution. Taking as example the future Blue Waters supercomputer where we should be able to launch 128 threads per node, sacrificing 1 of them for fault tolerance means an overhead of only 0.78%; the fault tolerance dedicated thread will be part of the user reservation. In doing so, no any assumption is made on the idle resources.

III. Related work

The scientific community have noticed the importance of fault tolerance in HPC from long time ago. For this reason, there is a vast literature on checkpoint/restart techniques. There are several possible implementations for checkpoint/restart such us kernel-level checkpoint or user-level checkpoint. Each one of these has several advantages and disadvantages. One well-known kernel-level checkpoint library is Berkley Linux Checkpoint Restart (BLCR) [21-24]. BLCR propose a disk-based checkpoint that has the advantage to be transparent for

the user. In order to deal with the I/O bottleneck, incremental checkpoint [25] has been proposed as a spatial reduction of the checkpoint data; but not all the applications can get a significant speed-up with this technique because of the large amount of data modified between two checkpoints. Also, speculative checkpoint [26] has been proposed as a temporal reduction of the checkpoint process, but again, the accuracy of the predictions is a complex problem for most of the scientific applications. In addition, some works propose local checkpoints with new technologies such us Phase Change Memory (PCM) [33]. However, these approaches do not solve the fundamental problem of disk-based checkpoint.

A. Diskless checkpoint

In 1997, diskless checkpoint [28, 29] has been proposed as a completely new way of taking checkpoints. Diskless checkpoint propose to store the checkpoint data in the memory or local disk of the computing nodes. In this way, the parallel file system and the I/O nodes don't participate in the checkpoint process avoiding the overhead caused by the I/O bottleneck [19]. However, if one of the computing nodes fail, the checkpoint data stored on that node will be unavailable. For this reason, diskless checkpoint proposes two strategies to guarantee the reliability of the system, replication and redundancy codes. When using replication, the computing nodes will create a virtual ring or another topology, and each node will send its checkpoint data to its neighbor. This technique cannot tolerate the failure of a node and its neighbor. In order to tolerate m simultaneous failures, each checkpoint should be replicated m times generating a large amount of extra data. For this reason, the redundancy codes are usually preferred when implementing diskless checkpoint.

Several encoding techniques [19, 30] can be used when implementing diskless checkpoint. The simplest encoding technique used in diskless checkpoint is the exclusive-OR (XOR) encoding [35]. Another encoding technique is the Reed-Solomon encoding [14, 15, 27]. Reed-Solomon encoding can tolerate several simultaneous failures within the same group but it requires a complex and time consuming encoding algorithm. In order to increase the number of simultaneous failures tolerated and to deal with scalability, the system can be partitioned in groups that will execute the encoding process in parallel increasing the checkpointing performance [28, 32]. However, it is important to notice, that using this approaches each group will have one or several spare nodes to encode and store the encoded data. In every cluster, the users have a limited number of computing nodes and dedicating a significant number of these for fault tolerance implies losing performance.

B. Spare-free diskless checkpoint

As presented above, the redundancy codes can tolerate several simultaneous failures while generating a low amount of encoded data in comparison with the replication approach. However, the spare nodes and the complex encoding process usually discourage the use of this technique. Several works have proposed spare-free diskless checkpoint techniques. An example of these works is the scalable checkpoint restart (SCR) library [31] that propose XOR encoding in a pipeline fashion, so the encoding process is not done on spare nodes but in the computing nodes. Once the encoded checkpoint is generated it is spread in blocks and replicated among the computing nodes; in this way the system does not need any spare node. However, the SCR library still have the same fundamental issue of XOR encoding, it only tolerates one failure per group.

Another example is the localized weighted checksum (LWC) [32] that proposes a Reed-Solomon encoding, also using a pipeline algorithm and replication of the encoded checkpoints among the computing nodes. This model can tolerate $\sqrt{k}$ failures in a group of k processes without need of spare nodes. Finally, in our previous work we presented the distributed diskless checkpoint (DDC) model [5] that can tolerate $\frac{k}{2}$ simultaneous failures in a group of k processes using a Reed-Solomon encoding. In this work we extend the DDC model by implementing a low overhead hybrid encoding algorithm and evaluate it with several benchmarks and real applications.

C. GPU Reed-Solomon encoding

GPU technologies have been proposed as a solution to improve the efficiency of storage systems in HPC [17]. Also, the Reed-Solomon encoding have been already implemented on GPU previously. In order to increase the reliability of storage system, a project at Sandia [34] is proposing GPU Reed-Solomon encoding on RAID-type systems. They motivate the need for triple-disk redundancy and their evaluation shows how GPUs outperform CPUs for this Reed-Solomon encoding. However, the encoding techniques used for storage systems usually need a low level of redundancy (Double-parity for RAID6) and the data to encode is a small set of blocks that can be stored in one single node before the encoding work starts. In diskless checkpoint, the level of redundancy is higher and the data to encode is an ultra-large set of files distributed among thousands of nodes. These fundamental differences make the encoding techniques used for storage systems not suitable for diskless checkpoint and vice-versa. We do not know at this moment, any GPU implementation of the Reed-Solomon encoding suitable for diskless checkpoint.

D. Positioning HDC with respect our previous work

In our previous work [5] we proposed a fault tolerance model able to tolerate up to 50% of simultaneous failures using Reed-Solomon encoding. We also proposed a partitioning strategy that virtually distributes the failures among the groups in order to increase the fault tolerance of the system. In addition, a scalable encoding algorithm was proposed and evaluated. In this paper, we do a study of high frequency diskless checkpoint for large GPU-accelerated clusters, such as Tsubame 1.2, with an adaptive CPU/GPU Reed-Salomon encoding depending on the application's configuration. Moreover, the paper presents a time-division (DDC) vs. spatial-

division (HDC) comparison of Reed-Solomon encoding on such architectures.

We propose a new technique that allows us to encode the checkpoints in parallel with the application execution and focus on the issues that are particularly related to hybrid architectures (See section V.E.). In our evaluation we compare our previous DDC model with the current hybrid approach in different configurations, and we present a model that allows the user to predict whether or not HDC is more advantageous than DDC for a given checkpointing requirements and a given machine.

IV. Working with GPUs

In order to have a better understanding of our hybrid technique, it is important to know the architecture of GPUs and understand the interactions between the CPU host and the GPU device. In this section we introduce the GPU architecture, the CUDA programming environment and the memory model. Though the architecture of the new generation of GPUs, Fermi, is already known, we are going to present the architecture of the GPU we used in our evaluation. The new features of the Fermi GPU give to this card an important gain in performance, but the main ideas and concepts of GPU computing remain basically the same in both cases.

A. The GPU architecture

The GPU architecture is designed for massively parallel computation thanks to its 240 cores, achieving 1 teraflop of performance in single precision. In contrast with the classic CPU architecture, that is optimized for low-latency access to cached data sets, the GPU architecture is optimized for high data parallel throughput computation. The cores are managed by the thread manager, who can spawn and manage over 30000 threads simultaneously. Fig. 1 shows the GPU Architecture. The cores of the GPU (240 for the GeForce GTX 280) are called Streaming processors (SP) and they have two Arithmetic Logic Units (ALU) and a Floating Point Unit (FPU). A group of eight SP plus two Special Function Units (SFUs) and a 16KB shared memory block make a Streaming Multiprocessor (SM). Two or three of these SM plus a texture block and some control logic blocks make a Texture Processor Cluster (TPC). Finally, a group of eight or ten TPCs make a Streaming Processor Array (SPA). A GPU has a SPA, an interconnection network, the DRAM global memory, the L2 caches and some other rasterization processors and control units.

The CPU, usually called the host, and the GPU, usually called the device, communicate through the PCI bus. First, the required data is copied from the main memory to the DRAM of the GPU card, then the GPU program, usually called the kernel, is copied from the CPU to the GigaThread manager on the GPU. When the kernel execution starts, the data is copied from the global memory of the GPU to the registers and shared memory. The management of the data through the memory hierarchy of the GPU is an important factor in order to optimize GPU applications. Finally, when the kernel execution is finished, the data is copied back from the device to the host.

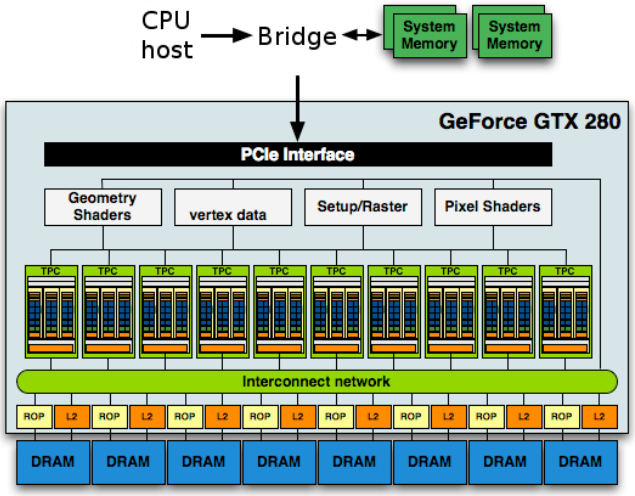

Fig. 1. GPU architecture

B. The CUDA programming model

CUDA support multiple standard languages such as Fortran, C and C++. When programming in with CUDA, the massively parallel parts of the application has to by written in one or several CUDA kernels, which will be executed on the GPU by thousands of simultaneous lightweight CUDA threads, each one of them with its own ID. In a CUDA kernel, all the threads execute the same code but can take different paths. The threads are grouped in blocks and the blocks are grouped in to a grid. The threads use the registers and the local memory to store the application data. Threads belonging to the same block can cooperate using the low latency shared memory. Finally, all the threads of all the blocks can access to the global memory. The CUDA API allows the host to allocate memory on the device and make data transfers before and after the kernel execution. In particular, the CUDA C Runtime API offers a high level of abstraction to the user, making easy to program these GPUs.

V. Hybrid diskless checkpoint

As explained in section III, diskless checkpoint can be implemented in different ways with different fault tolerance capabilities. The most reliable strategy is Reed-Solomon encoding but it is also the most complex and time consuming. In this section we start by explaining how Reed-Solomon encoding make a group of k nodes able to tolerate m simultaneous node failures and how to recover the lost data after a failure.

A. The weighted checksum encoding

The classic parity-based checkpoint scheme can also be implemented with floating-point numbers using floating-point number addition, this is called the weighted checksum encoding scheme [32]. In the weighted checksum encoding scheme, each node takes a local checkpoint, then they generate m weighted checksums establishing m equalities. Assuming that Ch_{ij} is the j^{th} element of the i^{th} node in a group of k nodes, the j^{th} data vector to encode will be composed by Ch_{xj} for x

between 1 and k. When this vector is multiplied by the row lth of the distribution matrix M, M_{lx} for x between 1 and k, the encoded value C_{lj} is generated. When the distribution matrix has $k+m$ rows, m weighted checksums are generated. In general, for a data vector Ch_{xj} we will have the next equalities:

$$\begin{cases} M_{11}Ch_{1j} + M_{12}Ch_{2j} + \cdots + M_{1k}Ch_{kj} = C_{1j} \\ \vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ M_{m1}Ck_{1j} + M_{m2}Ch_{2j} + \cdots + M_{mk}Chkj = C_{mj} \end{cases}$$

When $f < m$ node failures occur, the system will have m equations and f unknowns. If the weights of the weighted checksums were chosen properly, it is possible to recover the lost data by solving these equations. Notice that in this scheme we assume that the weighted checksums were stored in another m nodes dedicated to store these weighted checksums.

B. The weighted checksum recovery

In order to understand the weighted checksum recovery in a general case, we can analyze the next example. Let there be n_1 to nm the failed computing nodes and n_{m+1} to n_k the non-failed computing nodes. As result of these failures, Ch_{1j} to Ch_{1k} become unknowns in the system presented above. By re-structuring the equations we get the following system:

$$\begin{cases} M_{11}Ch_{1j} + \cdots + M_{1m}Ch_{mj} = C_{1j} - (M_{1m+1}Ch_{m+1j} + \cdots + M_{ik}Ch_{kj}) \\ \vdots \\ M_{m1}Ck_{1j} + \cdots + M_{mm}Ch_{mj} = C_{mj} - (M_{mm+1}Ch_{m+1j} + \cdots + M_{mk}Ch_{kj}) \end{cases}$$

Lets call M' the matrix composed by the coefficients M_{ij} in this new linear system. If M' has full column rank then the lost data in the nodes n_1 to n_m can be recovered by solving this linear system. Please notice that some of the checkpoint nodes could fail also and that this case is also tolerated as far as the total amount of failures does not exceed the number of encoded checkpoints generated.

From this example, we can understand that the data can be recovered only if the number of failures f is lower or equal to m and if the matrix M' has full column rank. But this last condition will depend on the failures' distribution, so to tolerate any $f < m$ failures, the matrix M has to verify the condition that any square sub-matrix, including minor, of M has to be non-singular. For this reason, it is necessary to use a Cauchy matrix or a Vandermonde matrix as distribution matrix of the encoding process [28, 29].

C. System topology

As presented in section V.A., the weighted checksum encoding generates a number of weighted checksums that are stored in spare-nodes and used to recover the lost data in case of failure. In section III we presented several models that propose to store the weighted checksums on the computing nodes and use some replication to guarantee the reliability of the system.

In our previous work [5], we proposed a new spare-free diskless checkpoint technique. The approach used in this work is similar but not identical as explained in section V.D. In order to deal with scalability, the encoding techniques, including the weighted checksum, are usually implemented in groups. By partitioning the system in groups, the encoding work can be done in parallel for different groups. In addition, the system can tolerate more simultaneous failures if the failures are distributed among the groups. If the system is partitioned in ten groups and each group generates two encoded checkpoints, then each group can tolerate two simultaneous failures, making the complete system able to tolerate twenty failures in total.

With our HDC technique we propose to divide the system in groups of k nodes. Each node will take its checkpoint and they will generate $m = k$ encoded checkpoints. In theory, generating m=k encoded checkpoints should allow each group to tolerate 100% failures. However, we will not store the encoded checkpoints on spare nodes but among the computing nodes. For this reason, a node failure will cause two erasures, a checkpoint and an encoded checkpoint, decreasing the fault tolerance capability of each group from 100% to 50%. Tolerating 50% of simultaneous failures within a group makes a system highly reliable, but it is also important to guarantee a low overhead.

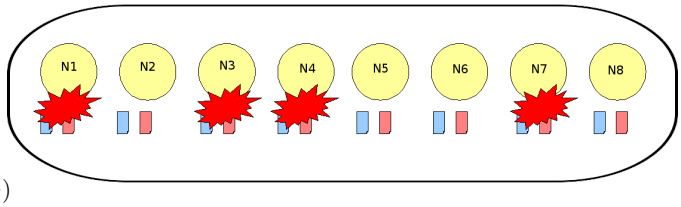

Fig. 2. HDC encoding group

The size of the groups k is a parameter that can be fixed by the cluster administrator or it can be chosen at the execution time by the user. However, it is important to define some limits to this parameter. In theory, $k = m > 1$ is enough to have an encoding system working, but if we chose a number too small, for example 2, then the system could fail with only two simultaneous node failures if they occur within the same group. On the other hand, if we chose a number too big, the group of nodes will need to generate a large amount of encoded checkpoints, increasing the checkpoint overhead. In Fig. 2 we can see an example of a group of 8 nodes with the checkpoints (in blue) and the encoded checkpoints (in red), the group can tolerate up to 4 simultaneous node failures.

Another important factor to analyze is the physical distance between the nodes belonging to the same virtual group. As much as possible, administrators should provide some architecture information in order to create groups with nodes that are "physically far". Physically far could mean nodes belonging to different racks or nodes connected to different network switches, or another material factor linking the two nodes, that could lead to a simultaneous failure of both nodes. In this way, when the rack or the switch fails the two or more nodes will be unavailable but if they belong to different virtual groups, the probability of tolerating such a failure increase dramatically.

D. HDC implementation

As we can see in sub-section V.B, we talk about node failures instead of process failures. This is because we can gather all the checkpoint data of every process of a node in a node checkpoint. When doing so, we need to map each process data on the node checkpoint, to be able to restart the failed processes. We use this approach in our technique for several reasons that will be explained below, but first we explain our implementation.

In our prototype, we divided a node in two parts, the head and the body of the node, as presented in Fig. 3. The head of the node will be a MPI process that manage the whole checkpoint work. The body of the node is composed by the several MPI processes participating in the application. At the checkpoint time, the head will receive the checkpoint data from every process in the node and will map it, in order to be able to re-scatter the checkpoint data of each process at the restart time.

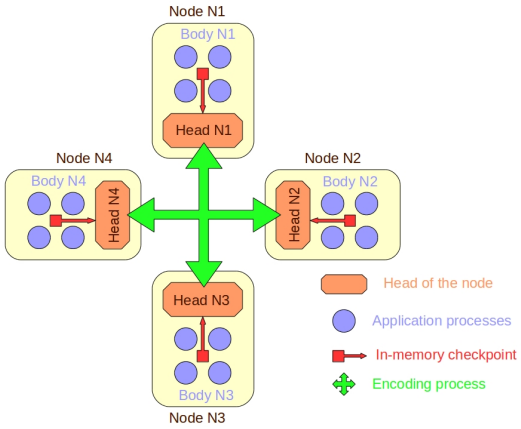

Fig. 3. HDC prototype

Once the checkpoint data has been gathered and mapped, the encoding process can start. The head will then communicate with other heads to encode the checkpoints. Notice that after the data has been gathered, the application processes are able to continue the execution and they do not need to wait for the encoding process. When the encoding process is done by the same hardware resources that participate in the application, the cost of diskless checkpoint is usually the sum between the time to make the local checkpoint and the encoding time. However, when the encoding process is done by extra idle hardware resources present on the nodes, the encoding process can be done in parallel and the checkpoint cost will be reduced to the local checkpoint plus the extra data transfered on the network (See section VI.A.). These are the advantages of dividing the nodes in two parts and gather the checkpoint data in one single node checkpoint. However, it is important to notice that this technique works well only when the checkpoint data of the whole node can be held in memory by one MPI process. This is the case of most GPU application, that are limited by the memory size of the GPU cards which usually lower than the amount of memory present on each node. Please notice that extra computing ressources on the computing nodes is different from spare nodes because spare nodes are dedicated to hold the parity data and they do not participate in the application execution causing much more overheads than our scheme.

E. Checkpointing GPU applications

Many GPU applications require an MPI process per GPU. Since the number of GPUs per node is most of the times smaller than the number of cores per node, the nodes participating in the applications will have extra idle CPU cores. These CPU cores can be used for fault tolerance, in this case to calculate the Reed-Solomon encoding for diskless checkpoint, but where and how remain difficult questions when working with hybrid applications. First we need to analyze the communication pattern of these hybrid applications. Multi-GPU applications work by using the following data transfer scheme:

- The data is copied from the device memory to the host memory by calling cudamemcpy.
- Then the data is transferred from a CPU to another using MPI communications.
- Finally, the data is copied from the host memory to the device memory by calling cudamemcpy again.

Between those data transfers, the CUDA kernels are executed one or several times. The CUDA kernels represent the core of the computing part of the applications and are executed by thousands of parallel thread inside the GPU. Interrupting a CUDA kernel execution to make a checkpoint is not a good strategy because to save and restore the data distributed among the different memory levels of the GPU can be a very complex problem. To see the memory hierarchy of GPUs, please refer to section IV.

We checkpoint the application during the data transfers. When a CUDA kernel execution is finished and the data is copied from the device to the host, is an ideal moment to make a fast copy of the data to the head of the node, and then let the application process continue its execution. In our prototype, the checkpoint of an application process is done by calling a function that require as parameters the data to checkpoint and the size of the data. We assume that the user coordinates the communications or use an MPI implementation that supports coordinated checkpoint to guarantee a consistent global state.

It is important to be able to choose the appropriate moment to make the checkpoint in order to avoid the interruption of the CUDA kernels, this is an advantage of user-level checkpoint. Notice that kernel-level checkpoint libraries, such as BLCR (See section III), cannot currently checkpoint GPU applications. Another advantage of user-level checkpoint is that the user can choose only the necessary data to checkpoint decreasing the memory the size of the checkpoint data significantly, which is consistent with the in-memory checkpoint strategy used in our HDC technique. A disadvantage of user-level checkpoint is that is not always easy to manage the checkpoint interval. However, in our prototype is easy to

control by adding a timer in the head of the node, then the application processes will ask permission to the head before copying the checkpoint data. This can happen several times within the same checkpoint interval, but the head will only allow the checkpoint after the timer expiration, then it will reset it for the next interval.

Finally, when the checkpoint data has been gathered by the head of the node, the encoding process can start. The Reed-Solomon encoding will be done in parallel with the application execution, using the idle CPU cores. The application can then continue without waiting for the coordination between head of the nodes. The CPU encoding algorithm is basically the same presented in our previous work [5]. The checkpoint data transfer to the head of the node and the extra data present on the network generated by the encoding process are the only overhead that this technique has. When the checkpoint data size is not too large, the overhead imposed to the GPU application should be almost negligible.

F. Checkpointing CPU applications

CPU applications are easier to checkpoint than GPU applications and there is a large literature about it (See section III). When checkpointing CPU applications using our HDC technique the Reed-Solomon encoding will be done with the idle GPU cards. The scheme presented in the previous subsection is basically the same one used for CPU applications. The distribution matrix is spread among the nodes of each group, then sent to the device memory at the beginning of the application and kept there all along the execution. The checkpoint data is divided in blocks, when a block is being encoded, another one is being transferred, overlapping communications and computation. An extra CPU core and a GPU card will be used per node for the encoding process. All the computations of the encoding algorithm have been implemented in one single CUDA kernel so the CPU only manage the communications.

Some applications require a specific number of MPI processes, such as power of two processes, that does not match the number of MPI processes that the user can launch within the number of nodes allocated at the reservation time. Obviously, the user will reserve the minimum number of nodes necessary to launch its application and leave some idle CPU cores. In this case, the checkpoint overhead generated should be similar to the checkpoint overhead for GPU applications.

Other applications do not have any number of processes restriction and can use all the CPU cores of each node, leaving idle only the GPU cards. In this case, one CPU core per node, that could participate in the application, has to be used for fault tolerance in our HDC technique. The overhead generated in this case will be larger than the overhead mentioned in the two precedent cases. However, when the number of MPI processes that can be launched per node increases, the checkpoint overhead decreases. All these three cases have been treated in our evaluation.

VI. Evaluation

A comparison of several diskless checkpoint schemes was made in our previous work [5] showing better fault tolerance rates and lower risk of catastrophic failure. A performance comparison between different techniques with different levels of reliability (e. g. XOR encoding, partner replication) is presented in [31]. The goal of our current experimentation is to evaluate the overhead of the technique proposed in section V for different applications and configurations and compare it with our previous model DDC [5], in order to understand in which conditions HDC can be more advantageous. The results of this time-division (DDC) vs. spatial-division (HDC) comparison of Reed-Solomon encoding should coincide with the theoretical overheads explained in the previous section. All our evaluation has been done on Tsubame 1.2 supercomputer with nodes composed by 8 AMD dual core Opteron processors (in total 16 cores) and 32 GB of memory. Each node is connected to two nVidia Tesla GPUs with over 1Teraflops of performance in single precision and over 100 GB/s of memory bandwidth. Tsubame has a 100 Gigabit network with a performance of 20 Gbps (10Gbps infiniband x2 per node).

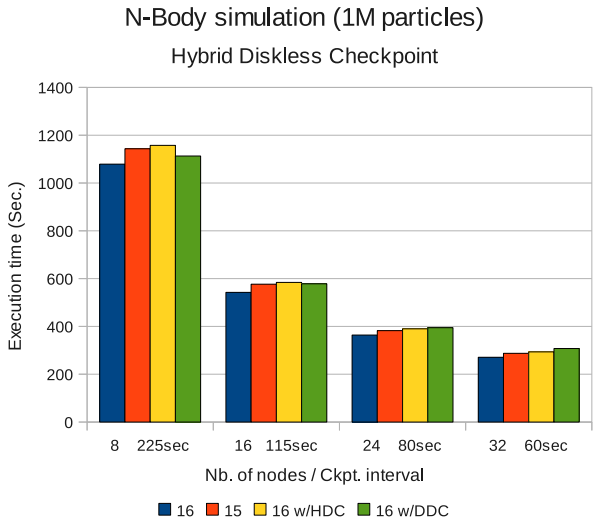

Fig. 4. N-Body simulation

A. N-Body simulation

First, we study the case when a CPU application can use all the CPU cores available in each node and the batch constraints do not allow the user to share a CPU core between application process and checkpointing process. To illustrate this case we will use a classic N-Body simulation. In this example we scale from 8 to 32 nodes, using a very simple implementation [36] and launching the application in four different configurations. In the first and second configurations, we launch the application using 16 and 15 CPU cores per node without checkpointing. Then, we make a third experiment launching 15 application processes per node and using an extra CPU core for fault tolerance (16 w/HDC). In the last experiment too, we

launch 16 processes per node but this time we checkpoint the application using the model (DDC) presented in our previous work [5]. The result of this evaluation is presented in Fig. 4.

In order to understand easily this results, we present in Fig. 5 a time decomposition for the case of 8 nodes of Fig. 4. First, we can see the execution time when launching 16 and 15 processes per node, obviously the former is faster. Then, we can see that the in-memory copy to the head of the node of HDC technique takes only a few seconds but the execution time (in blue) is slightly longer than the 15 processes case, which is consistent with the theoretical analysis given in section V. Indeed, since the encoding process is done in parallel with the application execution, the extra data transferred on the network by the encoding process will generate a short extra overhead. Also, the coordination between processes at every checkpoint interval should slightly increase the execution time. In the last experiment, we can see that the encoding process is much more time consuming that the in-memory copy of the HDC technique. However, in this case the sample checkpointed with DDC technique is faster than the one with HDC technique because in the HDC technique we sacrifice a CPU core as head of the node. It is important to notice that for this last experiment (16 w/DDC), the execution time is also slightly longer than the first one because of the coordination at every checkpoint interval.

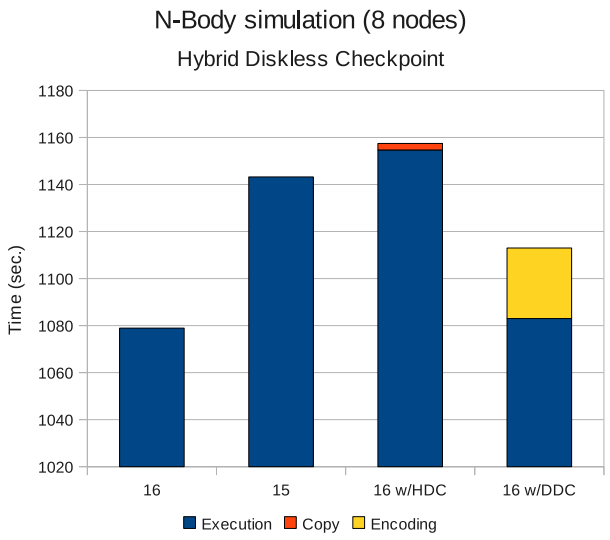

Fig. 5. Time decomposition for N-Body

In the HDC technique, we are losing 1/16 of performance plus the overhead mentioned above. The result of our evaluation shows an average overhead of 7% for HDC. As we can see in Fig. 4, in the DDC case, the overhead grows slightly with the number of nodes. We measured the encoding time and it remains constant from 8 nodes to 32 nodes. However, the coordination between processes at the checkpoint time gets more and more complex as the system size grows. An important difference between both techniques are the coordination and dependencies between processes at checkpoint time. In the HDC technique, the processes just have to copy the data to the head of the node and the heads will coordinate to encode the checkpoints, so there is no dependency between application processes added by the HDC technique. On the other hand, the DDC technique add dependencies between application processes, because processes belonging to the same encoding group will have to coordinate before starting the encoding work and processes belonging to different groups may have to wait for each other if the encoding work of both groups finish at different moments.

The hierarchical strategy (head-body) used in HDC decreases the number of MPI communications needed and the dependencies between processes. By decreasing the number of processes participating in the encoding work, this technique gets more scalable but the whole encoding work is delegated to a single CPU core per node making it more time consuming. This is why GPU-acceleration is important, in order to have a fast encoding work that allows the user to use smaller checkpoint intervals. The combination of these two features contributes to the scalability of the hybrid approach, leading to better performance for larger number of nodes.

Notice that when the number of cores per node is larger the overhead of HDC decreases; for example a cluster with 16 cores per node loses 7% of performance with HDC, but a cluster with 32 cores per node loses only around 3% of performance with HDC, making HDC significantly faster. On the other hand, a larger checkpoint interval will benefit more a non-hybrid approach such us DDC, than our HDC technique. We developed the following model in order to understand when HDC is more advantageous than non-hybrid diskless checkpoint for applications that use all the CPU cores on the nodes:

$$\frac{1}{N_{cpn}} + \frac{t_{cpy} * 100}{T_{ChI}} < \frac{t_{enc} * 100}{T_{ChI}}$$

where N_{cpn} is the number of CPU cores per node, t_{cpy} is the time to copy the checkpoint data to the head of the node, t_{enc} is the time to encode the checkpoints and T_{ChI} is the checkpoint interval. The time to copy and encode the checkpoints will depend on the size of the checkpoints and the characteristics of the machine, in our previous work [5] we defined a model to calculate this values. As explained above, the overhead of HDC in percentage is the CPU core sacrificed in every node ($\frac{1}{N_{cpn}}$) plus the overhead of the data copy work at each checkpoint interval ($\frac{t_{cpy} * 100}{T_{ChI}}$). On the other side of the formula we can see the overhead of the encoding work for DDC. By using this model, the user can understand whether HDC is faster or not for his checkpointing requirements and the characteristics of the machine where he wants to execute his application.

B. Himeno benchmark on CPU

This benchmark [6] was created by R. Himeno in order to evaluate the performance of computational fluid dynamics (CFD). It uses Jacobi iteration method to solve Poisson's equation systems and it measures the speed of the major loops.

We chose the Himeno benchmark because it has a total number of processes restriction (power of two) that may lead in many cases to extra idle resources on the nodes and because it is a very bandwidth demanding code and it reveals the worst-case scaling scenario for bandwidth intensive applications, even when production CFD applications would be larger and more complex. Scores of the Himeno benchmark are been published for a wide variety of architectures [8].

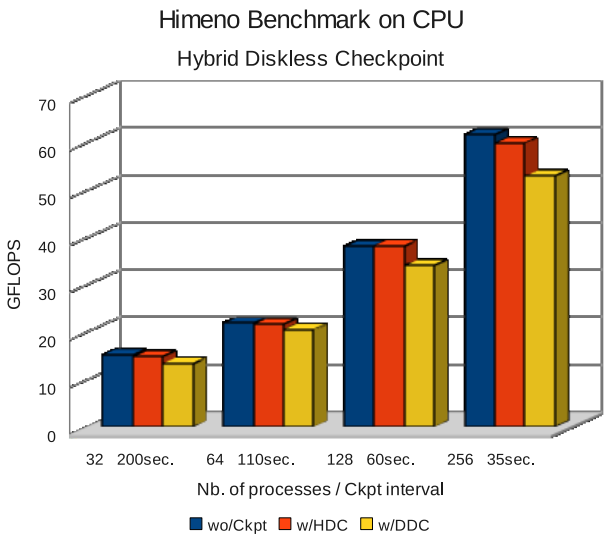

Fig. 6. Himeno benchmark on CPU

For our evaluation, we created two copies of the Himeno benchmark and we checkpointed the application using HDC in one copy and DDC in the other, with a group size of 8 in both cases. The checkpoint is done at the end of the major loop in the Jacobi function. The HDC sample will be launched in the same number of nodes but using an extra CPU core and a GPU card per node. Since the checkpoint interval t must decrease when the number of processes p increase, we tried to keep the product p*t almost constant during our evaluation. We used in all the cases a large grid size (257 x 257 x 513) of Himeno benchmark. In the DDC sample, the encoding process is not done by GPU, but by the same MPI processes that participate in the application, so the application must be stopped during the encoding process as explained above. As we can see in Fig. 6, the overhead generated by DDC slightly grows from 9% to 11%, when the overhead generated by HDC is less than 2%. This is consistent with our analysis because the encoding process is more complex and time consuming than the in-memory copy to the head of the node, particularly when checkpointing at a high frequency like in this experiment. In these circumstances, the HDC technique has a significantly lower overhead, comparing with a diskless checkpoint strategy where the application is stopped during the encoding work.

C. Himeno benchmark on GPU

The Himeno benchmark has been implemented also in CUDA [7] for GPU clusters. First, one-dimensional block distribution is done along the X-axis across multiple nodes. Then, MPI communications and Jacobi iterations are overlapped to optimize the algorithm. Also, the benchmark has been optimized within a GPU using shared memory and coalesced memory access. In this experiment too, we chose the large grid size (257 x 257 x 513) and we maintain the problem size for 4 and 8 GPUs.

In the checkpointed versions of this Himeno GPU benchmark, the checkpoint is done after the execution of the CUDA kernels, just after the data is copied back from the device to the host, before starting the MPI communications of the current iteration. Again, the product p*t is kept constant. As result of this experiment presented in Fig. 7, we get a checkpoint overhead not larger than 1% for the HDC technique. As mentioned in section V.E, this negligible checkpoint overhead was expected because the only two sources of overhead are the in-memory copy of the checkpoint data and the extra data circulating on the network. The infiniband network of Tsubame can easily absorb the overhead of the extra data transferred on the network during the encoding process in this case because only a few MPI processes are launched per node. For the DDC technique, we see again the overhead generated by the synchronizations after and before the encoding work, but in this case with a higher impact because of the high performance of GPUs.

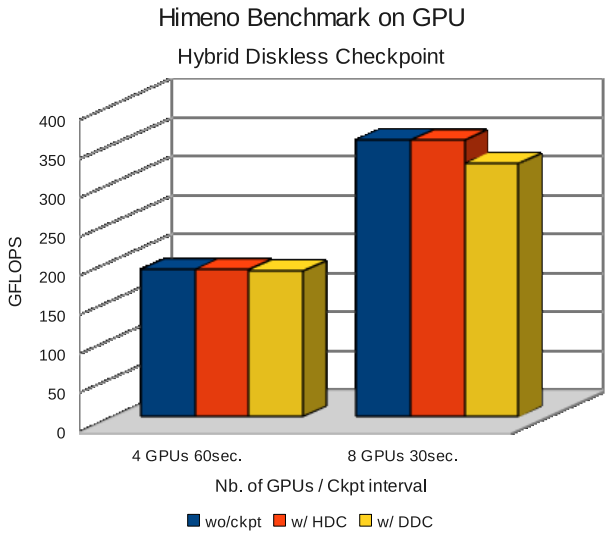

Fig. 7. Himeno benchmark on GPU

VII. Conclusions

Nowadays, heterogeneous computing is an important topic in HPC. In GPU-accelerated clusters, many hybrid applications cannot use all the hardware resources available on the nodes in an efficient way. For this reason, extra idle resources are usually present on the nodes and they can be used for fault tolerance purposes. On the other hand, reliability is an important open problem for next generation of supercomputers including hybrid machines. In this work, we propose a scalable

technique that uses those idle resources to tolerate up to 50% simultaneous failures with a high checkpoint frequency and guarantee a very low overhead.

As presented in our evaluation, the overhead is not larger than 7% in most of the cases and when the application does not use all the CPU cores per node, this checkpoint overhead becomes negligible, increasing significantly the checkpoint performance in comparison with other diskless checkpoint strategies. For applications that use all the CPU cores of each node, such us the N-Body simulation presented in our evaluation, the hierarchical strategy and the GPU-acceleration increase the scalability of our hybrid technique, leading to a low checkpoint overhead that is comparable with the DDC technique.

For our future work, we want to propose a hybrid diskless checkpoint library and evaluate it with petascale applications in petascale machines.

REFERENCES

[1] K. Barker , K. Davis , A. Hoisie , D. Kerbyson , M. Lang , S. Pakin , J. Sancho, Entering the petaflop era: the architecture and performance of Roadrunner, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, November 15-21, 2008, Austin, Texas

[2] B. Schroeder, G. A. Gibson, A large-scale study of failures in high-performance computing systems, Proceedings of the International Conference on Dependable Systems and Networks (DSN'06), p.249-258, June 25-28, 2006.

[3] B. Schroeder and G. A. Gibson, Understanding failures in petascale computers, SciDAC 2007 J. Phys.: Conf. Ser., vol. 78, no. 012022, 2007.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," SC Conference, vol. 0, 2009.

[5] L. Bautista Gomez, N. Maruyama, F. Cappello, S. Matsuoka, "Distributed Diskless Checkpoint for large scale systems", IEEE/ACM International Symposium on Cluster, Cloud and Grid computing (CCGrid2010), Melbourne, Australia, May 2010.

[6] The Riken Himeno CFD Benchmark. http://accc.riken.jp/HPC_e.html

[7] Matsuoka S, Aoki T, Endo T, Nukada A, Kato T, Hasegawa A GPU-accelerated computing—from hype to mainstream, the rebirth of vector computing. J Phys Conf Ser 180 (2009)

[8] The Riken Himeno CFD Benchmark (scores) http://accc.riken.jp/HPC_e/HimenoBMT_e/scoretop_e.html

[9] F. Cappello, Fault tolerance in Petascale/Exascale systems: current knowledge, challenges and research opportunities International Journal on High Performance Computing Applications, SAGE, Volume 23, Issue 3, 2009.

[10] B. Schroeder, E. Pinheiro, W. Weber. DRAM errors in the wild: A Large-Scale Field Study. SIGMETRICS, Seattle, June 2009.

[11] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.

[12] F. Schmuck , R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, Proceedings of the Conference on File and Storage Technologies, p.231-244, January 28-30, 2002

[13] S. Microsystems. Lustre file system, October 2008

[14] J. S. Plank, Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications, Technical Report CS-07-603, University of Tennessee, September, 2007.

[15] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST), San Francisco, CA, 2009.

[16] S. Matsuoka, The Road to TSUBAME and beyond, Petascale Computing: Algorithms and Applications, Chapman & Hall Crc Computational Science Series, 2008, pp. 289-310

[17] A GPU Accelerated Storage System, Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, Matei Ripeanu, IEEE/ACM International Symposium on High Performance Distributed Computing (HPDC 2010), Chicago, IL, June 2010. (To appear)

[18] A. Petitet, R. Whaley, J. Dongarra and A. Cleary. HPL – a portable implementation of the highperformance Linpack benchmark for distributed computers. http://www.netlib.org/benchmark/hpl

[19] NA Kofahi, S Al-Bokhitan, A Al-Nazer, On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis - Information Technology Journal, 2005.

[20] http://www.nvidia.com/object/fermi_architecture.html

[21] J. Duell, P. Hargrove and E. Roman, Requirements for Linux Checkpoint/Restart Lawrence Berkeley National Laboratory Technical Report LBNL-49659, 2002.

[22] E. Roman, A Survey of Checkpoint/Restart Implementations Lawrence Berkeley National Laboratory Technical Report LBNL-54942, 2003.

[23] J. Duell, P. Hargrove and E. Roman, The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart Lawrence Berkeley National Laboratory Technical Report LBNL – 54941, 2002.

[24] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove and E. Roman, The LAM/MPI checkpoint/restart framework: system-initiated checkpointing Proc. Los Alamos Computer Science Institute (LACSI) Symp. Santa Fe, New Mexico, USA, October 2003.

[25] J. S. Plank, M. Beck, G. Kingsley and K. Li, Libckpt: Transparent checkpointing under UNIX. In Proceedings of the USENIX, Technical Conference, 213—223, 1995.

[26] S. Matsuoka, I. Yamagata, H. Jitsumoto, H. Nakada, Speculative Checkpointing: Exploiting Temporal Affinity of Memory Operations, HPC Asia 2009, pp. 390–396, 2009.

[27] Z. Chen and J. J. Dongarra. Algorithm-Based Checkpoint-Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources. Rhodes Island, Greece, april 2006.

[28] J. Plank, K. Li, M. A. Puening, Diskless Checkpointing, IEEE Transactions on Parallel and Distributed Systems, v.9 n.10, p.972-986, October 1998.

[29] J. S. Plank and L. Xu, Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications, NCA-06: 5th IEEE International Symposium on Network Computing Applications, Cambridge, MA, July, 2006.

[30] C. Lu, Scalable diskless checkpointing for large parallel systems, PhD. Thesis, University of Illinois at Urbana-Champaign, IL, 2005.

[31] A. Moody, G. Bronevetsky, Scalable I/O Systems via Node-Local Storage: Approaching 1 TB/sec File I/O LLNL, TeraGrid Fault tolerance for Extreme-Scale Computing, 2009.

[32] Z. Cheng, J. Dongarra, A scalable Checkpoint Encoding Algorithm for Diskless Checkpointing, Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium, 2008.

[33] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, Y. Xie. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. In Super Computing Conference, Portland 2009.

[34] M. Curry, L. Ward, T. Skjellum, and R. Brightwell. Accelerating reed-solomon coding in raid systems with gpus. In International Parallel and Distributed Processing Symposium, April 2008.

[35] W. D. Gropp, R. Ross, and N. Miller. Providing efficient I/O redundancy in MPI environments. Lecture Notes in Computer Science, 3241:7786, September 2004.

[36] Gropp, W., Thakur, R., and Lusk, E. 1999 Using Mpi-2: Advanced Features of the Message Passing Interface. 2nd. MIT Press.