Implementing "Pliris-C/R" Resiliency Features Into the EIGER Application

Mike Davis
Customer Service
Cray, Inc.
Albuquerque, NM
u3186@cray.com

William Tucker
Los Alamos, NM
wwtucker@gmail.com

Joseph D. Kotulski
Electromagnetic Theory
Sandia National Laboratories
Albuquerque, NM
jdkotul@sandia.gov

***Abstract*— EIGER is a frequency-domain electromagnetics simulation code based on the boundary element method. This results in a linear equation whose matrix is complex valued and dense. To solve this equation the Pliris direct solver package from the Trilinos library is used to factor and solve this matrix. This code has been used on the Cielo XE6 platform to solve matrix equations of order 2 million requiring 5000 nodes for 24 hours.**

This paper describes recent work to implement "Pliris-C/R", a set of checkpoint/restart and other resilience features for Pliris. These include: targeting multiple file systems in parallel; striping controls; checkpoint period controls; turnstiling; open-file-descriptor sharing across processes; checkpointing on imminent job termination; application relaunch within the job; and scripts to monitor application progress. Timing data for runs using Pliris-C/R will also be presented.

Keywords-linear algebra, dense matrix, checkpoint, I/O

I. INTRODUCTION

The U.S. National Nuclear Security Administration (NNSA) has tasked its Advanced Simulation and Computing (ASC) program with providing high-performance simulation capabilities "to analyze and predict the performance, safety, and reliability of the nation's nuclear weapons" [1]. To fulfill this mission more effectively, two of NNSA's laboratories, Los Alamos and Sandia, formed the New Mexico Alliance for Computing at Extreme Scale (ACES) to design, procure, and deploy the Cielo supercomputer [2]. Cielo is a Cray XE6 system with 9216 nodes, rated at 1.38 Petaflops of peak performance [3]. Cielo is designated as an advanced-technology system [4], and as such it is tasked with handling workloads in which the typical compute job consumes a large fraction of the system's available resources [5] and runs for multiple days. Since 2012, one of the applications making up this workload has been the EIGER code.

EIGER is a frequency-domain electromagnetics simulation code based on the boundary element method [6]. This results in a linear equation whose matrix is complex-valued and dense. To solve this equation, the Pliris direct solver package [7, 8] from the Trilinos library [9] is used to factor and solve the matrix. EIGER is used on Cielo to solve matrix equations of order 2 million, requiring 5000 nodes for 24 hours or more per run. EIGER uses an MPI-everywhere method of parallelism; thus, the 5000 nodes host 80000 MPI processes.

This paper describes work undertaken in late 2013 to implement "Pliris-C/R", a set of checkpoint/restart and other resilience features in the Pliris solver package and the EIGER job stream for use on Cielo. Section 2 provides a high-level view of the Pliris design. Section 3 describes the high-level design of Pliris-C/R, and describes the approach taken to insert the C/R logic into the code, with a focus on how it exploits the I/O and file-system architectures of Cielo to achieve balance, regularity, and minimal contention. Section 4 details the user controls available to tune the behavior of Pliris-C/R, and discusses the considerations that go into choosing settings for some of the tuning parameters. Section 5 describes some of the lower-level design features of Pliris-C/R. Section 6 outlines the other resilience features implemented in the EIGER job stream. Section 7 shows some timing results from recent runs of EIGER on Cielo since the implementation of Pliris-C/R. Section 8 discusses potential areas of future work.

II. PLIRIS DESIGN

A brief description of Pliris is given in [7]. Pliris is capable of solving systems in either of two modes, depending on whether the right-hand-side vectors are available before or after factorization; for the EIGER calculations described here, only the "before" mode is used. The library distributes the augmented matrix in 2D blocks across the processes so that the blocks are as close to the same size as possible. To achieve good load balance, the factorization procedure operates on the matrix elements in block-cyclic fashion as if they were distributed in a torus-wrap mapping, as described in [10]; as a result, the factored matrix takes on a block-cyclic triangular form, such that each block's elements are updated and eliminated in a fashion much like that seen on a serial factorization of a monolithic matrix. Then, after the solve operation is completed, a permutation (shuffle) operation is performed on the result vector(s) to undo the torus-wrap mapping.

III. PLIRIS-C/R DESIGN

The four principal design factors of C/R are: where, when and how to checkpoint; where and how to restart; what partial results to transfer; and what kind of I/O to

perform. This section discusses all of these factors except when to checkpoint, which will be covered in Section 4.

A. *Where to Checkpoint*

The typical Pliris matrix solve operation spends the vast majority (over 90%) of its time in the factor () function. This function executes a loop that steps through the columns of the global operand matrix (augmented with one or more operand right-hand-side vectors) to perform the pivoting, scaling, interchange and row elimination operations. All of the code to implement the C/R capability is contained within this function; thus, no other parts of matrix processing (such as matrix fill, back substitution or permutation) are covered.

The checkpoint operation occurs at the bottom of the loop over the columns of the matrix, prior to the loop exit test, when an appropriate checkpoint period has elapsed. On alternating checkpoint events, the write operation is directed to one of a pair of alternate checkpoint sets (called "pink" and "blue"). At the end of every successful checkpoint operation, a control file named **intact** is written. This file contains four data items. The first two are integers that represent the size of the run. The third is an integer that represents the current value of the column loop index. The fourth is a character string that refers to the checkpoint set just written (pink or blue).

The restart operation occurs prior to entry into the column loop. Here, the C/R code checks for the presence of the **intact** file; its presence indicates that the run is a restart run, whereas its absence indicates an *ab initio* run. If the file is present, its contents are read. The size-of-run values are used to verify that the parameters for the current run match those from the prior run that generated the checkpoint. The checkpoint sets are then read, and the calculation proceeds at the appropriate column index (as specified by the third item in the file).

B. *What to Checkpoint*

The partial results to be transferred by each MPI process in a C/R operation (collectively referred to as a checkpoint image) include the local operand matrix, some local work vectors, some pointers, and some loop-carried scalars. Only those items that have a read-then-write reference pattern within the scope of the loop are included. The operand matrix is by far the largest piece of the checkpoint image; however, the fraction of the matrix that must be saved decreases as the factorization proceeds. Pliris-C/R is designed to perform "decrementing checkpoints," saving only the relevant fraction of the matrix at checkpoint events. Details of this design are discussed later in this section.

C. *How to Checkpoint*

The Pliris-C/R operations perform parallel unbuffered POSIX I/O. The I/O calls used (preadv and pwritev) allow for the specification of an offset/position at which to write and a vector of I/O requests to perform. For Pliris-C/R, the scalar and pointer members of the checkpoint set are packed into a single I/O vector element to minimize the I/O vector length.

On Cielo, the POSIX I/O used by Pliris-C/R is handled by Lustre 1.8 file system software [11]. Cielo has three Lustre parallel file systems, represented in the diagram of Fig. 1. Their mount points are named /lscratch2, /lscratch3, and /lscratch4. The /lscratch2 and /lscratch4 file systems are each comprised of 24 object storage servers (OSSs) and 1 active metadata server (MDS). The /lscratch3 file system is comprised of 48 OSSs and 1 active MDS. Each OSS is comprised of 6 object storage targets (OSTs), which are directly addressable by user software. Each of these file systems is made up of identical hardware components with identical theoretical performance characteristics at the OST level. The default stripe size setting is also consistent across the three file systems.

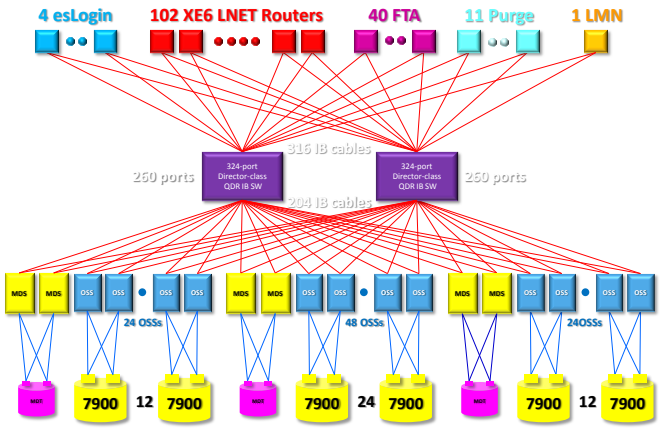

Figure 1: Cielo Lustre Architecture

Results from benchmarks of Lustre performance on Cielo [12] show bandwidths achieved from performing I/O on /lscratch3 (288 OSTs) from the I/O benchmark "fs_test". Fig. 3 of [12] shows N-N effective write bandwidth, and Fig. 4 shows N-N raw write bandwidth on a scaled set of MPI rank counts. These bandwidths are reproduced in Table 1 below. (The referenced source expresses the bandwidths as MB/s, as it is reported by fs_test, but the actual units are MiB/s.) Note how the performance decreases as N grows past 2048 MPI ranks; this tends to argue for an optimal load of 2048/288 = ~7 concurrent writers per OST. Note that in the raw case, the ratio of bandwidth at 2048 ranks to bandwidth at 65536 ranks is 1.29. This 29% overhead penalty is presumably due to overheads on the OST associated with having to service I/O requests on 65536/288 = ~227 files concurrently. In the effective case, the ratio is 1.78. The difference in these ratios is presumably attributable to the cost of metadata

operations, which imposes a significant limit on I/O efficiency at scale for the N-N regime.

Table 1 shows effective bandwidth of 57600 MiB/s with 32768 MPI ranks and 43600 MiB/s with 65536 MPI ranks. If we were interested in how this benchmark might perform on 40000 MPI ranks, we could interpolate linearly between these two points, and produce an estimate of 54500 MiB/s. Then, if we were interested in how it might perform on 80000 MPI ranks writing across all three file systems concurrently (using 500 OSTs), we could scale this figure by 500/288 to arrive at 94600 MiB/s. Granted, this estimate is favorably biased, since the ratio of writers to OSTs increases (80000/500 = 160 versus 40000/288 = ~138); but it is also unfavorably biased, since the number of metadata servers increases by a factor of three rather than two. We will assume that these biases cancel each other out, and will use this performance figure later, to compare with Pliris-C/R I/O bandwidth achieved in EIGER runs.

Table 1: Cielo Lustre /lscratch3 I/O Bandwidths (MiB/sec)

Processes	Eff. BW	Raw BW
1024	73900	74400
2048	77400	78500
4096	76200	75500
8192	72000	75900
16384	64000	72000
32768	57600	69400
65536	43600	60900

To mitigate the performance problems associated with large N-N I/O, Pliris-C/R uses a subsetting strategy called turnstiling [13], also called baton-passing. The idea behind turnstiling is that the benefits of presenting a more contiguous I/O load to the OST will overcome the cost of imposing some serialization on the I/O requests. Turnstiling also offers the benefit of potentially appending multiple checkpoint images to a single file, thus reducing the overall file count in a checkpoint set. This technique is used in other codes that run on Cielo [14]. Fig. 2 below illustrates a simple turnstiling arrangement consisting of two Lustre OSTs, each hosting three files. The processes colored red are proceeding through their respective turnstiles and doing I/O concurrently, while those in blue are waiting in line for their turn. The figure suggests that the complete I/O operation will consist of four turns.

In Pliris-C/R, each MPI process is assigned a specific file within the checkpoint set on which it will perform its I/O, a specific offset within the file, and a "turn" during which it will be allowed to proceed with its I/O. In general, many processes will be assigned the same file. There is one turnstile for every file in the checkpoint set. Each file resides wholly within a single Lustre OST. Processes assigned the same turn are not synchronized, but each is allowed to proceed independently, acting on its assigned file as its turn comes up and its turnstile becomes available. Processes assigned the same file proceed in turn, and in order according to their offset. Note that processes do not necessarily arrive in the turnstile queue in order according to their offset; thus, processes may be forced to wait even though the turnstile is available.

The initialization code for C/R calculates the amount of storage to be allotted to a process's checkpoint image in its assigned C/R file, and rounds this value up to the next multiple of the default Lustre stripe size; this image size is identical across processes and is used for all C/R I/O operations during the course of the run. The invariance of this value allows for regularities in file size and growth, as well as file offset assignment across processes.

The choice of turnstiling as an I/O strategy provides opportunities for other optimizations as well. Pliris-C/R assigns files and turns to the MPI processes by rank in round-robin fashion, with turns varying faster. It does this for two reasons: first, it helps keep I/O traffic on the compute node from hitting the node's network injection bandwidth limit (since ranks within a node are sequential by default); and second, it allows processes within a node to share open file descriptors across turns, which reduces Lustre metadata load and some of the overhead associated with serializing I/O requests. (This second optimization will be discussed in more detail in Section 5.)

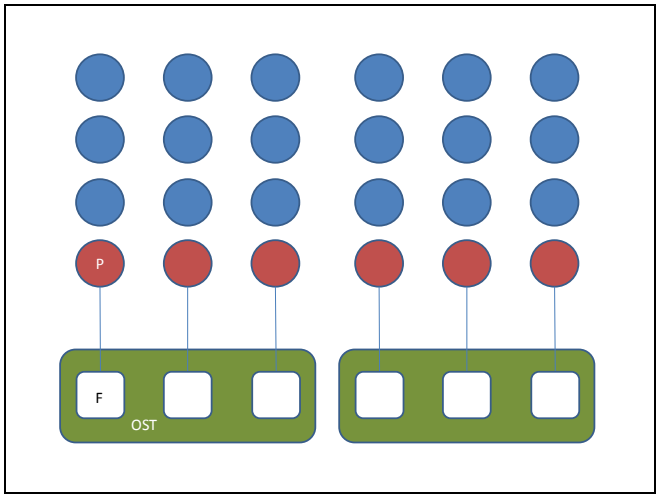

Figure 2: Turnstiling I/O

IV. PLIRIS-C/R USER CONTROLS

Pliris-C/R provides several user controls to tune the I/O behavior to serve the size of the application and the architecture of the Cray XE file system(s). These controls are available as environment variables with the prefix PLIRIS_CR. Table 2 summarizes the variables and their meanings.

Table 2: Pliris-C/R User Controls

Variable	Description
PLIRIS_CR_NFS	Number of file systems across which to spread the checkpoint set
PLIRIS_CR_DIR	List of directories (one per file system) to contain checkpoint sets
PLIRIS_CR_NS	List of OST counts (one per file system) across which to spread the checkpoint set
PLIRIS_CR_NF	Number of files that comprise the checkpoint set
PLIRIS_CR_COUNT	Number of checkpoint operations to perform during factor loop
PLIRIS_CR_SIGNUM	Signal indicating imminent job termination (default 23)

The PLIRIS_CR_NFS variable is used to specify the number of file systems across which the checkpoint set is (to be) spread. As discussed in Section 3, the Cielo system has three Lustre parallel file systems. By setting PLIRIS_CR_NFS=3, the user can specify that checkpoint I/O be performed across all three file systems in parallel.

The PLIRIS_CR_DIR variable is used to specify the list of directories, one per file system and space-delimited, in which checkpoint files (will) reside. (The **intact** file, described in Section 3, also resides in the first component of PLIRIS_CR_DIR.) Taking the Cielo system as an example, one might set PLIRIS_CR_DIR="/lscratch2/${USER} /lscratch3/${USER} /lscratch4/${USER}" to specify three user directories, one within each of the three file system mount points.

The PLIRIS_CR_NS variable is used to specify the list of OST counts, one per file system and space-delimited, across which the checkpoint set is (to be) spread. The C/R code uses these values to set the stripe origin characteristic for each of the files in the checkpoint set. Again taking the Cielo system as an example, one could set PLIRIS_CR_NS="144 288 144" to specify spreading across all of the available stripes of each of the three file systems.

The PLIRIS_CR_NF variable is used to specify the number of files that (will) make up a checkpoint set. The stripe count for each file is fixed at 1. The stripe size for each file is set to the default stripe size for the file system. Considerations for choosing a good value for this variable can be expressed as a set of tuning targets. To express these targets, first we define some terms in Table 3. Note that for the Cielo system, PPN=16, and for the EIGER runs described earlier, P=80000 and N=5000.

Given the definitions specified in Table 3, the tuning targets can be set out as shown in Table 4. The EIGER runs on Cielo are configured with S=500 and F=2500, which yields T=32, O=5, ION=1, and D=16.

Table 3: Pliris-C/R Tuning Parameters

Term	Derivation	Meaning
P		The number of MPI processes in the application
PPN		The number of MPI processes on a compute node
N	P/PPN	The number of compute nodes in the application
S		The sum of the components of PLIRIS_CR_NS
F		The value of PLIRIS_CR_NF, also the number of turnstiles
T	P/F	the number of turns, also the maximum number of processes operating on the same file, also the maximum number of checkpoint images in a file
O	F/S	the maximum number of concurrent I/O operations active per OST, also the maximum number of files hosted by each OST
ION	MAX(F/N,1)	The maximum number of concurrent I/O operations active on a Cielo compute node
D	GCF(PPN,T)	A measure of the efficiency of sharing open file descriptors

Table 4: Pliris-C/R Tuning Targets

Target	Explanation
4 <= O <= 8	This has been established experimentally as a good I/O load for an OST on Cielo.
S | F	This spreads the file load (and turnstiles) evenly across the OSTs. If this target cannot be met, then it is best if mod (F, S) is as close to S as possible.
F | P	This assures that all turns use the full bandwidth of all OSTs, and helps minimize the number of turns. If this target cannot be met, then it is best if mod (P, F) is as close to F as possible.
ION < 4	This assures that the I/O load on a compute node does not oversubscribe the node's network injection bandwidth.
D >> 1	Since file descriptor sharing is limited to the processes within a single OS image (compute node), it is optimal if PLIRIS_CR_NF is chosen so that PPN and T have a greatest common factor as large as possible.

A. When to Checkpoint: Coordination of Checkpoints

As discussed in Section 3, one of the principal design factors of a C/R scheme is when to checkpoint. The PLIRIS_CR_COUNT variable is used to specify the number of checkpoint operations to perform during execution of the loop over global matrix columns in the factor () function. The loop over columns, however, does not contain equal amounts of work across iterations. In addition, there is no explicit global synchronization event within the loop over columns that can be used to coordinate the checkpoint operation. Fortunately, there is no requirement that the processes coordinate their checkpointing on the basis of simulation time or wall-clock time. The only requirement is that the processes coordinate their checkpointing on the basis of agreed-upon progress points in the factorization, and column index is the most reasonable measure of these progress points.

The Pliris-C/R initialization code computes the set of column indexes at which to perform checkpoint operations so that the amount of factorization work performed between checkpoints is constant. A mathematical derivation of this algorithm starts with the observation that the amount of work W_J needed to perform the factorization on an individual column J (dominated by the outer product update step) is of the order $(N\text{-}J)^2$, where N is the size of the matrix. Let $V_J = J^2$ and note that V_J is a reflection of W_J across the midpoint $J_M = N/2$ of the domain [0: N].

Let $a_0, a_1, a_2, \ldots, a_{k+1} \varepsilon \{0, 1, \ldots, N\}$ be the bounds of k+1 equal subareas under the curve of V_J with $a_0 = 0$ and $a_{k+1} = N$ such that

$$\int_{a_i}^{a_{i+1}} J^2 \, dJ = \left(\frac{1}{k+1}\right)\left(\frac{N^3}{3}\right). \quad (1)$$

Evaluating for the case of i = 0, and solving for a_1:

$$a_1 = \sqrt[3]{1}\left[\frac{N}{\sqrt[3]{k+1}}\right]. \quad (2)$$

For the general case of i:

$$a_i = \sqrt[3]{i}\left[\frac{N}{\sqrt[3]{k+1}}\right]. \quad (3)$$

Let $b_0, b_1, b_2, \ldots, b_{k+1} \varepsilon \{0, 1, \ldots, N\}$ be the bounds of k+1 equal subareas under the curve of W_J with $b_0 = 0$ and $b_{k+1} = N$. Since W_J is just a reflection of V_J across the midpoint $J_M = N/2$ of the domain [0: N], the values b_i must be reflections of a_i across the domain:

$$b_i = N - \sqrt[3]{k+1-i}\left[\frac{N}{\sqrt[3]{k+1}}\right]. \quad (4)$$

Thus $b_1, b_2, \ldots, b_k$ are the column index values at which k equally-timed checkpoint sets should be written.

B. Decrementing Checkpoint of Matrix

As mentioned earlier in this section, Pliris-C/R is designed to save only the active portions of the operand matrix on checkpoint events. As factorization proceeds, the position within a process's block of the matrix that marks the active portion advances as shown in equation (5), where p^2 is the number of MPI processes used in the factorization.

$$E_i = \frac{Nb_{i-1}}{p^2}. \quad (5)$$

Fig. 3 shows a graphical representation of a process's block of the matrix after factorization, as seen by factor (), a C function. The colored pieces indicate portions of each matrix block that are written as the matrix is factored. When the index of the loop over columns (j) equals the value at which the first checkpoint is written (b_1), Pliris-C/R directs each process to write out its entire block of the matrix (starting at E_1) to the pink checkpoint set. When j equals b_2, processes write only the portion of the matrix starting at E_2 to the blue checkpoint set, since the elements above were either updated (above the diagonal) or eliminated (below the diagonal) before the pink checkpoint set was written and have not changed. Similarly, when j equals b_3, processes write the portion starting at E_3 to the pink checkpoint set. Thus, the pink and blue sets will contain alternating portions of the factored matrix. On the occasion of a restart, then, generally both sets must be read to reconstruct the operand matrix out of its constituent saved portions. Note that with this scheme, there are still static portions of the matrix (eliminated elements under the diagonal) being saved, but the cost to omit them would be prohibitive due to the excessive fragmentation of the I/O requests. (The values of E are shown here on row boundaries of the matrix, purely for convenience of illustration.)

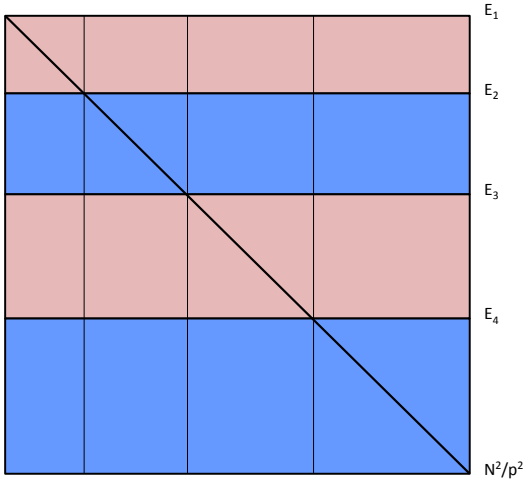

Figure 3: Decrementing Checkpoint of Process Block of Matrix

C. Choosing the Optimal Checkpoint Count

In discussing the considerations for choosing a good value for the PLIRIS_CR_COUNT variable, we start with the work of Daly [15]. Following the notation of this work, we define M to be the mean time between unscheduled interrupts that cause the application to terminate, δ to be the time to dump (write) a checkpoint set, N to be the number of dump-delimited compute segments making up the factorization, τ to be the time to execute a compute segment, R to be the restart time, and T_S to be the time required to compute the factorization. We use equation (20) from [15] to derive total work time:

$$T_W(\tau) = M * e^{R/M} * \left(e^{(\tau+\delta)/M} - 1\right) * T_S/\tau \quad (6)$$

In the Pliris-C/R implementation, τ is not an independent variable; rather, it is determined by $\tau = T_S / N$, where N is determined by PLIRIS_CR_COUNT. The restart time R is comprised principally of matrix fill time F and checkpoint read time ρ. Thus we refine the work time equation:

$$T_W(N) = M * e^{(F+\rho)/M} * \left(e^{(T_S/N+\delta)/M} - 1\right) * N \quad (7)$$

The next refinement will account for the fact that checkpoint write time δ is not constant, but rather decreases as the run progresses, due to the decrementing checkpoint of the matrix. We define the function δ(i) to be the time to write the checkpoint set associated with column index b_i and note that ρ is the time to read (potentially) both checkpoint sets and assemble the operand matrix. We set δ(N) to 0 since no checkpoint is written for the last segment. The work time equation then becomes:

$$T_W(N) = M * e^{(F+\rho)/M} * \sum_{i=1}^{N}\left(e^{(T_S/N+\delta(i))/M} - 1\right). \quad (8)$$

We derive M as follows. From [16] we use the component MTBF of 25 hours and system MTTI of 200 hours to obtain $\lambda_1 = (25*3600)^{-1} = 90000^{-1}$ and $\lambda_2 = (200*3600)^{-1} = 720000^{-1}$. Since we are interested in running on only 5000 of the 8944 nodes [3], the risk of experiencing a component failure is lower than if we ran on the full system, so we will adjust λ_1 accordingly, to $\lambda_1 = (90000*8944/5000)^{-1} = 160992^{-1}$. This yields $M = (\lambda_1+\lambda_2)^{-1} = 131572$. Note that scheduled interrupts are not included in this derivation, since the recovery cost is mitigated by the fact that Pliris-C/R is coded to checkpoint on these events.

Experience in running Pliris-C/R on 5000-node EIGER applications indicates that the matrix fill time F is 900 seconds, the checkpoint read time ρ is 1440 seconds, the factorization time T_S is 81573 seconds, and the time to write checkpoint i , where i ε {1, 2, …, N-1}, is as shown in equation (9).

$$\delta(i) = 960 * \sqrt[3]{\frac{N+1-i}{N}}. \quad (9)$$

Table 5 shows the values of T_W for the given values of the work equation parameters and various values of N. The optimal value for N is 6, thus the proper value for PLIRIS_CR_COUNT is 5. Note that the true optimum lies just off of N=6, but the current implementation does not allow the user to achieve it. This is a potential area of future work.

Table 5: Work times (sec) as a Function of Checkpoint Count

N	T_W
1	116858
2	99744
3	95334
4	93631
5	92946
6	92572
7	92832

The job's wall clock time limit is not a factor in determining the best value for PLIRIS_CR_COUNT. For example, if the problem's expected time spent doing factorization were 45 hours, the job's wall clock time limit were 24 hours, and the optimal checkpoint count were set at 9, then the resulting checkpoint period would be 5 hours, and at most 5 checkpoints before the *ab initio* job terminates (including the one triggered by PLIRIS_CR_SIGNAL).

The "checkpoint period" is the period of time spent calculating between the completion of the last checkpoint and the start of the next checkpoint; that is, the time spent writing the checkpoint set is not included in the checkpoint period. Thus, to continue the example, if the time to write a checkpoint set is 15 minutes, then the checkpoints will occur 5 hours and 15 minutes apart.

For the hypothetical job described above, the checkpoint set would allow the follow-on job to restart from a point 24 hours into the 45-hour calculation, thus the follow-on job will complete the factorization in only 21 hours. However, the checkpoint period for the follow-on job will remain the same as that for the *ab initio* run, namely 5 hours. This is a limitation of the algorithm used to calculate the checkpoint events in the current implementation.

V. PLIRIS-C/R FEATURES

A. File Descriptor Sharing

As mentioned in Section 3, once the decision has been made to adopt the turnstiling strategy, the opportunity for other optimizations arises. One of the optimizations implemented in Pliris-C/R is the use of sharing file descriptors among processes that share the same OS image (i.e., the same compute node) and share the same turnstile. The technique of sharing open file descriptors among

processes is documented in [17]. Use of this technique results in fewer file open and close operations, thus reducing load on the file system's metadata server(s). In addition, since processes queueing at a turnstile must communicate somehow in order to serialize their operations, the passing of the open file descriptor serves this function as well.

The Pliris-C/R initialization code forms the processes into MPI groups according to their turnstile index values, then assigns each process a rank within the group using a combination of the process's host node ID and MPI rank within the node. MPI rank within this group then becomes the process's assigned offset within the checkpoint file, and also its turn index. Each process also determines if its rank within the group is the lowest or highest on its node, as this indicates that it has special duties. The lowest-ranked process is an "opener"; that is, it will open the file itself, rather than relying on the open file descriptor from another process. The highest-ranked process is a "closer"; that is, it will close the file and refrain from sharing the file descriptor with any other process.

During a checkpoint operation, each process with group rank 0 opens the file for its group and goes first through its turnstile, while the other processes wait their turn. When a process receives an MPI "go on turnstile" message from its predecessor (by turn index), it either receives the file descriptor from its predecessor, or opens the file if it is an "opener"; then it proceeds through the turnstile. After a process finishes its I/O, it either sends a copy of its open file descriptor to its successor (by turn index), or closes the file if it is a "closer"; then it sends an MPI "go on turnstile" message to its successor.

Testing of the effectiveness of turnstiling and file-descriptor sharing was performed [18] on Cielo. The tests were run using 160 MPI processes, with 16 ranks on each of 10 compute nodes. The tests were sized to present the same I/O load to one OST of Cielo as the EIGER application presents to 500 OSTs spread across the three file systems. Each test was run in five modes. In the first mode (NXN), each process writes to its own file. In the second mode (NX1), all processes write to a single shared file, and each process is assigned a distinct offset within the file. In the third mode (NX5), processes are split into five groups of 32 each, and each group writes to its own file, and each process is assigned a distinct offset within its group's file. The fourth mode (TURN5) is just like the third, except that the processes in each group write in turnstile fashion, each in sequence according to its assigned offset. The fifth mode (TURN5_SFD) is just like the fourth, except that the processes within a group (or turnstile) share the file descriptor associated with their assigned file. In all modes, the I/O demand presumably exceeds the node's injection bandwidth limit. The test guides each process through a write loop that iterates eight cycles, and each process writes a checkpoint image of size 1.879e9 bytes every cycle; thus, the total amount of data moved in each test is 2.405e12 bytes. The write operation is timed every cycle using gettimeofday (), with the start time collected by the first process to visit the write call, and the end time collected by the last process to complete the write call. Write times are accumulated and reported at the end of the test. Table 6 shows the timings for the five different modes; these timings are averaged over eight separate runs of the test.

Table 6: Checkpoint Times (sec) on Single OST, as a Function of I/O Strategy

Test	Avg	Std Dev
NXN	11640	367
NX1	7697	721
NX5	7747	697
TURN5	6918	800
TURN5_SFD	6718	665

Note that the TURN5 case shows a marked reduction in time spent writing compared to NX5, confirming the hypothesis that the benefit of presenting contiguous requests to the OST overcomes the cost of serializing the writes. There is also a small reduction in time spent writing for the TURN5_SFD case compared to TURN5. In addition, the variation in timings is smaller. The typical EIGER run involves 80000 processes running across 5000 compute nodes, writing to 2500 files spread across 500 OSTs, with the two smaller file systems each hosting 625 files on 125 OSTs and one MDS, and the larger file system hosting 1250 files on 250 OSTs and one MDS. In a TURN5 mode, the three metadata servers will service 20000, 20000, and 40000 opens per checkpoint operation, whereas in a TURN5_SFD mode they will service 1250, 1250, and 2500 opens. This results in a tradeoff of small but predictable cost in on-node communication to share file descriptors versus moderate but potentially variable cost in metadata operations associated with file opens.

B. Determining First-In and Last-Out of Code Regions

Another feature in Pliris-C/R, more of an optimization than a resiliency feature, involves the use of shmem calls to determine the first rank into, and last rank out of, certain regions of code. These calls are used in regions where time stamps are collected at the start and end of checkpoint operations, to support the reporting of timing information at the end of factorization. They are also used in the region of code that writes the **intact** file. The implementation relies on the declaration of a shared-memory atomic counter variable on MPI rank 0. On entering a first-in region, each rank atomically queries and increments the variable (shmem_int_finc ()), and the rank to see a zero value takes on the duties of the first-in rank (e.g., collecting the "start" timing data). On exiting a last-out region, each rank atomically queries and decrements the variable, and the rank to see a value of 1 takes on the duties of the last-out rank (e.g., collecting the "end" timing data, creating the **intact** file). Typically, only ranks at the front or the end of a

turnstile queue perform the shmem calls; thus, the overheads of accessing the shared variable are much lower than if all ranks were to participate. Since the factorization operation contains no regular global synchronizations, the shmem implementation is deemed low-impact compared with one that would rely on MPI barriers.

Testing was performed on Cielo [19] to compare the costs of reporting first-in and last-out events using shmem-based versus barrier-based methods. In this test, a synthetic application is executed on 80000 MPI processes running on 5000 nodes. The application is run in two modes. In the first mode, a "storm" of simultaneous shmem_int_finc () calls is performed from 5000 processes on 5000 separate nodes. Wall-clock times of microsecond resolution are collected from each participating process using gettimeofday (). The time to execute the "storm" is computed as the difference of the minimum start time and the maximum finish time. In the second mode, each of the 80000 processes is directed to execute a workload designed to mimic that of the EIGER application (i.e., a set of local matrix operations followed by a checkpoint operation). At the end of the workload, 5000 processes on 5000 separate nodes execute an MPI_Barrier () call. Each mode is repeated six times. Wall-clock times are collected from each of the 5000 processes using the same method as in the shmem-based mode. The timings for the shmem-based test were on the order of 2.10e-1 +/- 1.0e-3 seconds, whereas the barrier-based timings were on the order of 8.8e1 +/- 2.5e1 seconds. The results indicate that, at large scale, a "storm" of shmem updates on a single rank is more efficient than an MPI barrier for determining first-rank-in and last-rank-out of a code region.

VI. EIGER Job Stream Resiliency Features

A. Recovery from Compute Node Failures

The MOAB job script in Fig. 4 illustrates how to set up the EIGER batch job to detect and recover from a compute node failure. This is the most common condition encountered by a failing EIGER run. The script is annotated with line numbers down the left column; these are not part of the actual script file.

```
…   # top of script
3   #MSUB -l nodes=2308:ppn=16
4   #MSUB -l walltime=24:00:00
5   #MSUB -l signal=23@10:00
…   # middle of script
30  for try in `seq 1 4`
31  do
32  aprun -n 36864 -N 16 ./a.out 2>&1 | tee out.${try}
33  grep "ec_node_failed" out.${try} >/dev/null || break
34  done
…   # end of script
```

Figure 4: Job Script Resilient to Node Failures

Lines 3-5 specify the MOAB parameters for job allocation size, job time limit, job name, and job output disposition. Also included in these parameters is a specification of which signal the job expects to receive when its wall clock time limit is near, and how long before the time limit the signal should be sent. As described in Section 3, the user control variable PLIRIS_CR_SIGNUM specifies the signal number that will be sent to signal imminent job termination or scheduled system shutdown. Lines 30-31 and 33-34 provide the logic to perform multiple successive application launches within the job, each potentially restarting from a checkpoint set generated from the prior launch, in the event that a prior launch was terminated due to a node failure. Note that the job allocates 2308 nodes, but the application launches on only 2304 nodes (36864 / 16), leaving four spare nodes available in the event of node failure. This resilience technique allows a job to re-launch the application up to four times within the same allocation. Each successive launch writes its standard output to a separate file, to aid in determining the exit status of the launch.

The Pliris-C/R code performs a check from MPI rank 0 after every iteration of the loop over matrix columns in factor () to see if the signal has been received, and if it has, the code performs a checkpoint write. In line 5 of the script, the signal 23 is specified.

This resiliency feature has also been used in other job scripts on Cielo [20].

B. Managing Checkpoint Storage Areas

Pliris-C/R is coded to check for the existence of the checkpoint files, and if necessary, create the files and assign their stripe characteristics so that subsequent reads and writes will be performed in an optimal fashion. As mentioned in Section 3, the creation process can be costly when the system is busy; this is especially true of stripe assignment. As a result, the solver process can be delayed while this creation process takes place. To mitigate this delay, a standalone serial program called **pliris_cr** is available to allow the user to prepare the checkpoint directories with restart files before the application executes its checkpoint code (i.e., while the job is waiting in the queue, or while the application is factoring the first batch of columns).

The **pliris_cr** program takes three command-line arguments: the matrix size (known within the Pliris code as ncols_matrix), the number of processes across which the matrix is partitioned (npes_per_col), and the operation to perform (setup, verify, or cleanup). The program also reads the environment variables PLIRIS_CR_NFS, PLIRIS_CR_DIR, PLIRIS_CR_NS, PLIRIS_CR_NF, and PLIRIS_CR_COUNT. From this information, the program is able to form the names of the checkpoint files that the solver application will write. The program then goes through the list of files and, for the operation 'setup', creates each one and sets its proper stripe characteristics. (The

'cleanup' operation removes the checkpoint files and directories, and the 'verify' argument directs the program to verify that all files are present on their assigned OSTs.)

Once the checkpoint files are all in place, Pliris-C/R in the factor () function will sense their presence and choose the more efficient code path for opening the files.

Fig. 5 shows the set of commands that could be executed to prepare the checkpoint directories using **pliris_cr**.

```
1   #!/bin/bash
2   export PLIRIS_JOBNAME=cr_test
3   export PLIRIS_CR_NFS=3
4   export PLIRIS_CR_NFS=3
5   DIR=${PLIRIS_JOBNAME}
6   DIR2=/lscratch2/${USER}/${DIR}
7   DIR3=/lscratch3/${USER}/${DIR}
8   DIR4=/lscratch4/${USER}/${DIR}
9   PLIRIS_CR_DIR="${DIR2} ${DIR3} ${DIR4}"
10  export PLIRIS_CR_DIR
11  export PLIRIS_CR_NS="144 288 144"
12  export PLIRIS_CR_NF=4608
13  export PLIRIS_CR_COUNT=6
14  ./pliris_cr 1920000 192 setup
15  # All done
```

Figure 5: Sample pliris_cr Script

C. *Detecting and Managing Job Hangs*

Over the course of the past three years in which EIGER has run on Cielo, there have been instances where the application has experienced a hang condition, and consumed several hours tying up nodes while making no progress. To assist in monitoring the long-running EIGER applications and detecting such conditions, the **pliris_watch** program has been implemented. This program takes as input four parameters: the batch job ID of the job running the application; the expected time (in seconds) required to perform the factorization; a grace period, in seconds; and an action to perform (report-only or report-and-signal). The program reads the environment table for the PLIRIS_CR variables (which should match those used in the job) and computes the expected times when checkpoint sets should appear on the system. It then watches for the appearance/update of the **intact** file associated with the checkpoint sets. If the file fails to appear/update at the expected times (plus the specified grace period), then the program performs the specified action. This program can be run within the batch job, in the background alongside the EIGER application itself, or from an interactive login session.

VII. Results From Recent EIGER Runs

The typical production EIGER run factors a matrix of 2474989 double-precision complex elements. A matrix of this size implies a checkpoint set of size approximately 9.801e13 bytes. The best checkpoint times observed so far were in job 1474501, run on 4/14/2014, where the range across six checkpoint operations was 871 to 956 seconds, writing to 500 OSTs spread across the three Lustre file systems. This yields an effective bandwidth of 1.050e11 bytes/second, which compares favorably to the projected effective N-N bandwidth of 94600 MiB/s on the fs_test benchmark from Section 2 (i.e., 1.050e11/500 = 2.1e8 versus 94600*1048576/576 = 1.722e8). The EIGER bandwidth also compares favorably with the bandwidth reported in [14], where 57 GB/s was observed when writing to 288 OSTs of Cielo's /lscratch3 file system from 65536 processes using turnstiling (i.e., 2.1e8 versus 5.7e10/288 = 1.979e8). The worst checkpoint times observed so far were in job 1568851, run on 11/25/2014, where the range across seven checkpoint operations was 2004 to 2826 seconds. Investigations into the cause for the low performance in this case are ongoing.

VIII. Future Work

With the upcoming installation and deployment of the Trinity XC system, with its DataWarp and DNE technologies, we will be interested to see how useful or necessary the Pliris-C/R optimizations will be on that system. In the interest of reducing restart time, we will be looking at ways to reduce or eliminate the matrix fill step for runs that read from a checkpoint set. There may be ways to overlap I/O on static portions of the matrix with factorization of the active portion, and we will investigate this possibility. We would also like to explore the value of adding a first-come first-served scheme to the queueing of processes at a turnstile. Finally, we will investigate improvements that would allow the user to specify a checkpoint interval that comes closer to achieving optimal work time; this may become important on future systems, where the reliability parameters (M/δ) will likely become even more of a factor in resiliency analysis.

Acknowledgment

The authors thank the following people for their assistance in this work. Courtenay Vaughn (Sandia) and Brett Kettering (LANL) reviewed an early draft of this paper and provided valuable comments. Dan Poznanovic (Cray) reviewed a final draft and pointed out where some key points could be clarified. Any errors that remain are the responsibility of the authors.

References

[1] http://www.nnsa.energy.gov/aboutus/ourprograms/defenseprograms/futurescienceandtechnologyprograms/asc

[2] http://nnsa.energy.gov/aboutus/ourprograms/defenseprograms/futurescienceandtechnologyprograms/asc/supercomputers#cielo

[3] http://www.lanl.gov/projects/cielo/index.php

[4] http://nnsa.energy.gov/aboutus/ourprograms/defenseprograms/futurescienceandtechnologyprograms/asc/ascnewsletters-0 , p. 4.

[5] http://www.nnsa.energy.gov/aboutus/ourprograms/defenseprograms/futurescienceandtechnologyprograms/asc/ascnewsletters/ascmar13

[6] W.A. Johnson et al., "EIGER™: an open-source frequency-domain electromagnetics code," Antennas and Propagation Society International Symposium, 2007 IEEE, pp. 3328-3331, doi: 10.1109/APS.2007.4396249.

[7] http://trilinos.org/docs/dev/packages/pliris/doc/html/index.html

[8] J. D. Kotulski, "Pliris: review and new capabilities," Trilinos User Group Meeting, November 2005, Albuquerque, NM. http://trilinos.sandia.gov/events/trilinos_user_group_2005/presentations/kotulski.pdf

[9] http://trilinos.org

[10] B. A. Hendrickson and D.E. Womble, "The torus-wrap mapping for dense matrix calculations on massively parallel computers," SIAM Journal of Scientific Computing, vol. 15, no. 5, September 1994, pp. 1201-1226, doi:10.1137/0915074.

[11] Lustre 1.8 Operations Manual. Sun Microsystems, Inc., Santa Clara, CA, 2010. https://docs.oracle.com/cd/E19495-01/821-0035-12/821-0035-12.pdf.

[12] B. M. Kettering, D. Bonnie, A. Torrez, and D. Shrader, "Lustre and PLFS parallel I/O performance on a Cray XE6," Cray User Group Conference, May 2014, Lugano, Switzerland. https://cug.org/proceedings/cug2014_proceedings/includes/files/pap101.pdf

[13] L. Crosby, "Parallel I/O techniques and performance optimization," NICS Spring Training, National Institute for Computational Sciences, March 2012. http://www.nics.tennessee.edu/sites/www.nics.tennessee.edu/files/pdf/Lonnie.pdf

[14] S. Langer, A. Bhatele, and C. H. Still, "pF3D simulations of laser-plasma interactions in National Ignition Facility experiments," Computing in Science and Engineering, vol. 99, August 2014, IEEE Computer Society, doi: 10.1109/MCSE.2014.79. http://charm.cs.illinois.edu/~bhatele/pubs/pdf/2014/cise2014.pdf

[15] J.T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," Future Generation Computer Systems, Vol. 22, Elsevier B.V., Amsterdam, 2006, pp. 303–312.

[16] J.A. Ang et al., "Alliance for Computing at Extreme Scale," Cray User Group Conference, May 2010, Edinburgh, Scotland. https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/16-18Thursday/17A-Dosanjh-slidesACES-CUG.pdf

[17] W.R. Stevens, Advanced Programming in the UNIX® Environment. Addison-Wesley, Reading, MA, 1993, pp. 479-489.

[18] Cray Bug 805877. https://crayport.cray.com.

[19] Cray Bug 824359. https://crayport.cray.com.

[20] J.O. Stevenson et al., "Reliable Computation Using Unpredictable Components, JOWOG-34 Spring 2012 Conference, May 2012, Los Alamos, NM.