Resilient Matrix Multiplication of Hierarchical Semi-Separable Matrices

Brian Austin, Eric Roman, Xiaoye Li
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{baustin, eroman, xsli}@lbl.gov

ABSTRACT

The hierarchical semi-separable (HSS) matrix factorization has useful characteristics for representing low-rank operators on extreme scale computing systems. To prepare for the higher error rates anticipated with future architectures, this paper introduces new fault-tolerant algorithms for HSS matrix multiplication that maintain efficient performance in the presence of high error rates. The measured runtime overhead for error checking and data preservation using the Containment Domains library is exceptionally small and encourages the use of frequent, fine-grained error checking when using algorithm based fault tolerance.

Categories and Subject Descriptors

I.6.3 [**Computing Methodologies**]: SIMULATION AND MODELING—*Applications*

Keywords

HSS, ABFT, numerical methods, error detection

1. INTRODUCTION

Emerging extreme-scale architectures pose new challenges for parallel algorithms. These architectures, due to the expected hardware fault rate, require the use of resilient algorithms, and the high cost of data movement demands increased computational intensity. In this paper, we explore the implications of these challenges for solving linear systems with resilient solvers and preconditioners. A new class of structured sparse factorization methods employing numerically low-rank structures, hierarchically semi-separable (HSS) matrices, may be suitable for broad classes of large partial differential equations (PDEs) systems that are often too difficult for current methods. An HSS-sparse solver applies HSS compression techniques to the dense submatrices appearing in traditional sparse factorization methods. This compression reduces the number of floating point operations, leading to nearly linear complexity for certain PDEs, even in 3D geometry [20]. In addition to the flops reduction, a more significant benefit is that HSS compression reduces the data volume and the communication/flops ratio on parallel machines [18, 12]; this reduction is mainly due to the nearly linear size of the compact HSS data structure. Our interest in exploiting these algorithms on extreme-scale architectures has led us to investigate fault-tolerant schemes for HSS matrix operations. This paper focuses on the matrix multiplication algorithms of an HSS matrix with a dense vector/matrix.

Many types of faults occur in computer systems [1]. The impact of these faults and the techniques for detetecting or recovering (or not recovering) from the errors they induce depend on the timing and location of the fault. Permanent and intermittent faults cause frequent failures at the same location and are mitigated by replacing hardware after uncorrectable permanent errors are detected [15]. Transient faults, such as particle-induced bit flips, are more likely to be random and not reproduceable. Many transient faults can be detected and corrected in hardware by ECC and memory scrubbers. Rollback recovery, based on checkpoint restart is commonly enployed on parallel systems to recover from uncorrected faults that lead to fail-stop errors. When undetected, "soft" (i.e. transient) faults can modify data without causing the program to halt or providing other error notification, leading to so-called "silent data corruption" (SDC) that can propagate through a calculation and result in incorrect output. In future HPC systems, silent faults are expected to occur at higher rates than fail-stop errors [14].

Our main goal is to handle SDC, which must be detected before it can be corrected. The methods we introduce, based on algorithm-based fault tolerance (ABFT), provide the ability to detect faults and handle errors in software. Since these error-correction techniques based on data encoding provide limited recovery from possible errors, we combine encoding techniques with a software rollback scheme to achieve full recovery. We have designed several resilient algorithms for HSS matrix-vector multiplication, which differ in the points during the computation at which they detect and recover from errors. Our implementation of the rollback scheme uses Containment Domains (CDs) [4, 16], a hierarchical recovery mechanism that corresponds to the hierarchical nature of HSS algorithms. We introduce a finite automata model to describe our performance, and an approximate Markov model for comparison. We analyze our results with these models, and compare our results across error rates to determine their effectiveness. The main contributions of this paper are the FT-HSSmv algorithms, an analysis of their costs, and measurements of their effectiveness.

2. METHOD

In this section, we describe two methods for detecting and recovering from errors during the multiplication of an HSS matrix with a dense vector (HSSmv) These methods are easily extensible to matrix multiplication with a dense matrix (HSSmm). HSSmv

FTXS'15 June 15 2015, Portland, OR, USA
ACM 978-1-4503-3569-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2751504.2751507.

and HSSmm are indispensable operations when HSS factorization is used as a preconditioner in iterative solvers. HSSmv and HSSmm are also used in the randomized sampling algorithm for HSS construction [11]. In section 2.1, we discuss the advantages and disadvantages of resilience by preserving data to safe storage relative to methods based on checksum encoding. We review the HSS factorization and an HSSmv algorithm for matrix-vector multiplication in section 2.2, followed by a description of resilient HSSmv algorithms: FT-HSSmv by preservation-restoration in section 2.3 and FT-HSSmv by encoding in section 2.4.

2.1 Models for algorithm-based fault tolerance

We focus on two different FT schemes for these algorithms: one is based on preservation-restoration, and the other is based on checksum encoding.

Preservation-restoration consists of the following components:

- Preservation: Select uncorrupted data to preserve in a "safe" store.
- Detection: During computation, check for violations of expected invariants.
- Recovery: When errors are detected, restore the correct data from the store, and rollback the computation to the last correct stage.

Preservation methods enjoy generality. They can be applied to any algorithmic structure, but they require a safe store. The safe store may fail, but store must be available for recovery. A fault detected by the preservation method must not cause a failure of the safe store. Several software packages implement a safe store. In our work, we use Containment Domains (CDs) [16]. Zheng [21] uses GVR [17]. We chose CDs because they provide a composable, hierarchical means for preservation, as well as local recovery. This matches well with the nature of our hierarchical algorithms, our applications, and the deep hierarchies expected in future exascale platforms.

Checksum encoding matrices, introduced by Huang and Abraham [6], consists of the following components:

- Encoding: Additional (redundant) data is added in some form of encoding.
- Processing: Redesign of algorithm to operate on encoded data.
- Detection: Check the encoded data for errors.
- Recovery: Correct identified errors from the encoding information.

This technique is used in FT-ScaLAPACK [19], for dense matrix operations, such as MM, LU and QR factorization.

Checksum encoding methods provide self-recovery — the (computed) encoded data provides sufficient information for error dection and correction. There is no need for a safe store, therefore no other software support is required. The main drawback of these approaches is that only limited error patterns can be corrected. For example, in Huang's MM row/column checksum scheme, only one error per row or column can be corrected. In some cases, the algorithm can detect more errors, but cannot recover. In order to tolerate more errors, more encoded data is needed, which may be costly both in memory and in runtime. A second drawback is that the checksum encoding, detection and recovery methods are specific to particular algorithms. A new FT scheme needs to be designed and proved mathematically for each new operation.

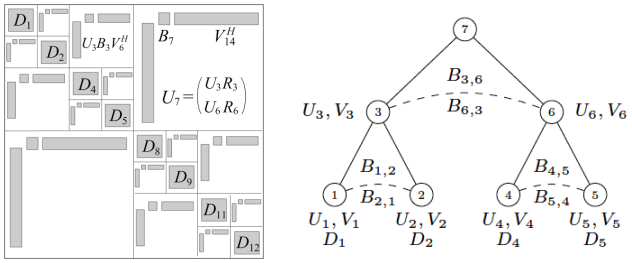

Figure 1: HSS representation of A, HSS tree

2.2 HSS matrix-vector multiplication: HSSmv

The HSS structure of a matrix A can be represented as a recursive structure through a telescoping factorization [11]:

$$A^{(0)} = B^{(0)} \tag{1}$$

$$A^{(l)} = U^{(l)} A^{(l-1)} (V^{(l)})^* + B^{(l)}, l = 1, 2, \ldots, L-1 \tag{2}$$

$$A \approx A^{(L)} = U^{(L)} A^{(L-1)} (V^{(L)})^* + D^{(L)}, \tag{3}$$

where the matrices $U^{(l)}$, $V^{(l)}$ and $B^{(l)}$ are block diagonal matrices at each level l. The superscript l refers to the level number in the recursion. We will omit it in writing when there is no ambiguity in the context. The number of levels, L, is a tunable parameter. The recursive structure is illustrated in Figure 1, which shows how matrix A is partitioned. The $\{U, V, B\}$ generators at each level of the partition are associated with the HSS tree nodes on the right. At the l-th level, there are 2^l nodes in the HSS tree ($l = 0$ at the root). We use $U_\tau^{(l)}$ to denote one of the diagonal blocks of $U^{(l)}$ associated with node τ at level l of the tree. Thus, $U^{(l)} = diag(U_1^{(l)}, \ldots, U_{2^l}^{(l)})$. This is similarly defined for block diagonal matrices $V^{(l)}$ and $B^{(l)}$.

One key advantage of the HSS structure over other non-hierarchical structures is the use of *nested bases*. That is, at any intermediate node τ in the HSS tree, the actual basis U_τ^{big} is not stored explicitly, but only represented as the unevaluated product of the bases of the children (ν_1 and ν_2) and the node τ's (small) basis U_τ, as follows:

$$U_\tau^{\text{big}} = \begin{bmatrix} U_{\nu_1}^{\text{big}} & 0 \\ 0 & U_{\nu_2}^{\text{big}} \end{bmatrix} U_\tau \equiv \begin{bmatrix} U_{\nu_1}^{\text{big}} R_{\nu_1} \\ U_{\nu_2}^{\text{big}} R_{\nu_2} \end{bmatrix}, \tag{4}$$

where U_τ is written as $[R_{\nu_1}\ R_{\nu_2}]^T$. In this notation, only the basis matrix U_τ without superscript "big" is explicitly stored. Here, R_{ν_1} and R_{ν_2} are the two (stored) components of the basis matrix U_τ, each of which is of size $r \times r$, with r being the numerical rank of the block. For simplicity of the cost analysis in this paper, we use the maximum rank of all the blocks, which is called *HSS-rank*. Our code allows for variable ranks with different blocks [12].

With this hierarchical, unevaluated product representation, the stored basis matrices are asymptotically smaller than U_τ^{big}, and the compression results at the children are reused at the parent node, hence also reducing operation count asymptotically.

Our HSS construction algorithm uses randomized sampling and the Interpolative Decomposition, which is an efficient way to obtain a low-rank approximation of the form $A_{\nu_1,\nu_2} \approx U_{\nu_1} B_{\nu_1,\nu_2} V_{\nu_2}^*$ for each off-diagonal block. The number of sampling (random) vectors needs to be slightly larger than the maximum rank r, e.g., $r + 10$ is sufficient, see [11] for details. The implementation we use is from [12], where an adaptive algorithm was developed when r is not known *a priori*.

Algorithm 1 HSSmv—multiplication of an HSS matrix with a dense vector.

Input: all generators U_τ, V_τ, B_{ν_1,ν_2}, and D_τ of an HSS matrix A, a dense vector x.
Output: $b = Ax$.

1. For every leaf node τ, calculate
$$\tilde{x}_\tau = V_\tau^* x(I_\tau) \tag{5}$$
2. Looping over all non-leaf nodes τ, from finer to coarser, calculate
$$\tilde{x}_\tau = V_\tau^* \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix} \tag{6}$$
3. For the root node τ, compute
$$\begin{bmatrix} \tilde{b}_{\nu_1} \\ \tilde{b}_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_1,\nu_2} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix}$$
4. Looping over all non-leaf nodes τ, from coarser to finer, calculate
$$\begin{bmatrix} \tilde{b}_{\nu_1} \\ \tilde{b}_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_1,\nu_2} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix} + U_\tau \tilde{b}_\tau \tag{7}$$
where ν_1 and ν_2 are the children of τ.
5. For every leaf node τ, calculate
$$b(I_\tau) = U_\tau \tilde{b}_\tau + D_\tau x(I_\tau). \tag{8}$$

Using the HSS structure of A, the product $b = Ax$ can be evaluated with Algorithm 1. The procedure involves one pass up the tree, multiplying the V generators, followed by one pass going down the tree, multiplying the $\{U, B\}$ generators. The computational complexity of HSSmv is $\mathcal{O}(nr)$ [11].

In the following two subsections, we describe our approaches to making Algorithm 1 resilient to failures. We expect the HSSmv algorithm to be used repeatedly in an iterative solver. Errors may occur in this long stretch of computation.

We make the following assumptions in our ABFT algorithms:

- The HSS construction is computed correctly.[1]
- Checksums are computed correctly at the initial stage.[2]
- During computation, both matrix data and checksums are susceptible to SDC.

2.3 FT-HSSmv via preservation-restoration

When using the preservation-restoration model two questions need to be answered: (1) What data must be preserved? Our objective is to protect against SDC in the HSS matrix, so we preserve all the U, V, B and D blocks of the HSS tree. The meta-data associated with the tree is much smaller than the data and therefore less likely to be effected by uniformly distributed errors, so we chose not to preserve the HSS meta-data. (2) What invariant conditions can we check? All of our algorithms rely on the invariance of matrix multiplication with respect to associativity. In particular, for any two vectors p and q, $(p^*A)q = p^*(Aq)$. For dense matrices, the methods of Huang and Abraham [6] can be applied directly. For HSS matrices, we introduce related checksum tests that can be verified at different stages of the HSSmv algorithm. By including or excluding some intermediate tests, we can explore the trade-offs between overhead costs for checking and preservation and error-induced costs for restoration and error correction. Below, we describe the checksum tests and associated costs of coarse, medium and fine-grained error checking algorithms.

[1] Interpolative decomposition is also susceptible to errors. We intend to design FT algorithms for HSS construction in our future work.

[2] The checksum encoding scheme in section 2.4 could be used to justify this assumption.

2.3.1 Coarse-grained approach

In this approach, we perform correctness checking only at the end of Algorithm 1. To prepare for coarse-grained checking, we precompute a checksum vector $c^* = e^*A$ immediately after the HSS construction, where e is a vector of all ones. The procedure for computing c is a simple adaptation of HSSmv in Algorithm 1 in which we sweep through the telescope factorization from left to right. At the end of the calculation (following Step 5), we check that $c^*x = e^*b$. If this is not satisfied, we restore the entire HSS matrix, (i.e., all the $\{U, V, B, D\}$ generators), and restart the algorithm from Step 1.

The memory overhead for this coarse-grained error check is small—we need only one vector c of size $\mathbf{n}$. The drawback is the long time to recover from errors; this is especially important on systems with high error rates.

2.3.2 Medium-grained approach

The medium-grained approach adds two checks to the coarse-grained version. The two additional checks verify that the intermediate quantities $\tilde{x}_\tau$ are computed correctly and are performed after Step 1 and Step 2 of Algorithm 1 respectively.

The checksums used to validate $\tilde{x}_\tau$ are based on the observation that
$$\tilde{x}^{(l)} = V^{(l)*} \dots V^{(L-1)*} V^{(L)*} x = V^{(l)^*} \tilde{x}^{(l+1)},$$
where $\tilde{x}^{(l)}$ is the concatenation of all $\tilde{x}_\tau$ on level of the HSS tree. Suppose we introduce a weight vector w. (In practice, w can be a vector of all ones.) We pre-compute the intermediate *check vectors* at each level l:
$$\tilde{w}^{(l)*} = w^* V^{(0)*} \dots V^{(l-1)*} = \tilde{w}^{(l-1)^*} V^{(l-1)^*} \tag{9}$$
By associativity of multiplication, we obtain the following *invariant condition* for any level l:
$$\begin{aligned} \tilde{w}^{(l)*} \tilde{x}^{(l)} &= (w^* V^{(0)*} \dots V^{(l-1)*})(V^{(l)*} \dots V^{(L)*} x) \\ &= (w^* V^{(0)*} \dots V^{(l-1)*} V^{(l)*} \dots V^{(L)*}) x \\ &= \tilde{w}^{(L+1)*} x = \sum_\tau \tilde{w}_\tau^{(L+1)*} x_\tau \end{aligned}$$
Furthermore, a partial w-x checksum holds for a subset of nodes:
$$\tilde{w}_\tau^{(l)*} \tilde{x}_\tau^{(l)} = \sum_{\nu \in \text{descendants of } \tau} w_\nu^{(l')*} \tilde{x}_\nu^{(l')}, \text{ for some } l' > l.$$
Note that the sum over ν is not over all descendents of τ, but only a set of descendants at the same level l'.

Algorithm 2 is the medium-grained FT HSSmv algorithm. The text in blue highlights the error detection/recovery procedures. After Steps 1 and 2, the checks are performed using the precomputed check vectors $\tilde{w}$. The third check, after Step 5, uses the precomputed c check vector introduced for the coarse algorithm.

Memory is needed for the check vectors $\tilde{w}$ at all levels of the HSS tree. Denote r as the HSS-rank, and b as the block size at the finest level partition. At each level l, there are 2^l vectors of size r each. The memory required for the check vectors is:
$$r \sum_{l=1}^{L} 2^l \approx r 2^{L+1} = 2rn/b.$$

Algorithm 2 FT-medium-HSSmv: HSSmv with error detection/recovery at leaves and root.

Input:

- all generators U_τ, V_τ, B_{ν_1,ν_2}, and D_τ of HSS matrix A,
- a dense vector x,
- precomputed checksum vector $c^* = e^*A$ and the weighted check vectors $\tilde{w}^{(l)}$ in Eqn. (9).

Output:

- $b = Ax$. Errors are recovered after the leaf-node calculations, after the root-node calculation, and after the final calculation.

1. For every leaf node τ, calculate
$$\tilde{x}_\tau = V_\tau^* x(I_\tau) \tag{10}$$
$$p_\tau = \tilde{w}_\tau^{(L+1)^*} x_\tau, \quad q_\tau = \tilde{w}^{(L)^*} \tilde{x}_\tau.$$
If $p_\tau \neq q_\tau$ then restore V_τ for all leaves and return to step 1.

2. Looping over all non-leaf nodes τ, from finer to coarser, calculate
$$\tilde{x}_\tau = V_\tau^* \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix} \tag{11}$$
At root τ, compute $p_\tau = \tilde{w}_\tau^{(L+1)^*} x_\tau, q_\tau = \tilde{w}^{(1)^*} \tilde{x}_\tau$.
If $p_\tau \neq q_\tau$ then restore V_τ for all non-leaf nodes and return to step 2.

3. For the root node τ, compute
$$\begin{bmatrix} \tilde{b}_{\nu_1} \\ \tilde{b}_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_1,\nu_2} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix}$$

4. Looping over all non-leaf nodes τ, from coarser to finer, calculate
$$\begin{bmatrix} \tilde{b}_{\nu_1} \\ \tilde{b}_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_1,\nu_2} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix} + U_\tau \tilde{b}_\tau \tag{12}$$
where ν_1 and ν_2 are the children of τ.

5. For every leaf node τ, calculate
$$b(I_\tau) = U_\tau \tilde{b}_\tau + D_\tau x(I_\tau). \tag{13}$$
Let $c^* = e^*A$ be the precomputed checksum vector. If $c^*x \neq e^*b$ then restore U_τ and B_{ν_1,ν_2} for all nodes and return to Step 3.

In addition, a vector of size n is needed to store the column checksum vector $c = e^*A$. Therefore, the total storage required for the medium-grained approach is $\mathbf{n + 2rn/b}$.

2.3.3 *Fine-grained approach*

In the fine-grained approach, we verify correctness of the matrix-vector product for each operation involving the $\{U, V, B\}$ generators. To do so, we associate a column checksum vector with each block, e.g., e^*U_τ. During the matrix-vector multiply operation, after we compute each product $U_\tau x$, we test the invariant condition $(e^*U_\tau)x = e^*(U_\tau x)$. If this is not violated, the U_τ block is restored, and the computation is repeated.

The storage used by the checksum vectors can be calculated as follows. Additional storage is required for each matrix $\{U, V, B, D\}$. For the blocks in V, at the bottom level L, we need 2^L vectors of size $b \approx n/2^L$ each (total is n). At the intermediate levels $l < L$, we need 2^l vectors of size $2r$ each. Summing up all the intermediate levels:
$$2r \sum_{l=1}^{L-1} 2^l \approx 2r2^L = 2rn/b\,.$$
Adding the above two, we find the total checksum memory for the V components is: $n + 2rn/b$. Going down the tree for the B components, at each level l, need 2^{l-1} vectors of size $2r$ each, for a total of
$$2r \sum_{l=1}^{L} 2^{l-1} \approx 2rn/b\,.$$
For the U components at each level l, we need 2^l vectors of size r each, so the storage required is:
$$r \sum_{l=2}^{L-1} 2^l \approx rn/b\,.$$
Finally, at bottom level for D, the total storage is n. Adding all the above, the total size of the checksum vectors for the fine-grained approach is $\mathbf{2n + 5rn/b}$.

2.4 FT-HSSmv via checksum encoding

In the HSSmv algorithm, the basic building block is a dense matrix-vector product, where the dense matrices are the $\{U_\tau, V_\tau, B_\tau\}$ generators associated with the HSS tree nodes. We devised a generic ABFT matrix-matrix product algorithm that uses Huang-Abraham's checksum encoding [6], but adapted to tolerate SDC in the input matrix.

Consider GEMM $C = A * B$.[3] We augment the matrix A with column sum encoding, $A_c = (A; [e^*A])$, and the matrix B with row sum encoding, $B_r = (B \quad [Be])$. Here, we use the square bracket $[\cdots]$ to denote the checksum encodings that are precomputed correctly. Carrying on the multiplication with A_c and B_r, we obtain the full checksum matrix
$$C_{cs} = \begin{pmatrix} AB & A[Be] \\ [e^*A]B & [e^*A][Be] \end{pmatrix}.$$
Assume that A is corrupted from silent errors, becoming $\tilde{A}$, but B is still correct. Then, the erroneous calculation results in
$$\tilde{C}_{cs} = \begin{pmatrix} \tilde{A}B & \tilde{A}[Be] \\ [e^*A]B & [e^*A][Be] \end{pmatrix}.$$
After the computation, we proceed with the following three steps to handle potential errors. First, we compute the column sum of $\tilde{A}B$ and compare this sum vector with the last row of $\tilde{C}_{cs}$, (i.e., $[e^*A]B$). If the two vectors exceed a prescribed threshold, we conclude some entries of $\tilde{A}$ are incorrect. Second, given the correctly precomputed row checksum Ae and column checksum e^*A, Steps 1-4 of Algorithm 3 correct up to one error per row (or column) of A. Step 5 updates $\tilde{C}_{cs}$ to reflect the corrections to $\tilde{A}$.

Compared to the preservation-restoration methods in prevision section, this FT-GEMM has the potential to correct a limited number of errors more efficiently than recomputing the entire result. The full-checksum encoding approach requires both row- and column-checksums and therefore requires roughly twice as much memory for storing checksums as the fine-grained preservation scheme described above, but does not require preserving a second "safe" copy of the input matrix.

An noteworthy limitation of FT-GEMM is the restricted range of errors from which it can recover. In particular, it can recover from

[3] B is a single vector in case of matrix-vector product.

Algorithm 3 Using checksum encoding to recover silent errors in a matrix $\tilde{A}$ and propagating corrections to the product $\tilde{C} = \tilde{A}B$.

Given a full checksum matrix $\tilde{A}^{(full)} = \begin{bmatrix} \tilde{A} & [Ae] \\ [e^*A] & [e^*Ae] \end{bmatrix}$, possibly including errors as described above, this algorithm corrects the erroneous elements of $\tilde{A}$ if possible, and returns the number of errors. An error code is returned if the errors $\tilde{A}$ cannot be corrected.

1. Compute: $r = \tilde{A}e - [Ae]$,
   Count: N_r = number of nonzero elements in r.
   Record each nonzero row index i: $p[j] = i, 0 \le j < N_r$.

2. Compute: $c^* = e^*\tilde{A} - [e^*A]$
   Count: N_c = number of nonzero elements in c.
   Record each nonzero column index i: $q[j] = i, 0 \le j < N_c$.

3. If $N_r \neq N_c$, return error code for multiple errors per row or column.

4. Co-locate the error at row $p[i]$, column $q[j]$
   for $i = 1 : N_r$
   for $j = 1 : N_c$
   if ($r[p[i]] = c[q[j]]$), then recover:
   $\tilde{A}_{p[i],q[j]} = \tilde{A}_{p[i],q[j]} - r[p[i]]$.
   endfor
   endfor

5. Propagate corrections from $\tilde{A}$ to $\tilde{C}$
   for $i = 1 : N_r$
   for $j = 1 : N_c$
   if ($r[p[i]] = c[q[j]]$), then recover:
   for $k = 1 : N_B$
   $\tilde{C}_{p[i],k} = \tilde{C}_{p[i],k} - r[p[i]]B_{q[j],k}$.
   endfor
   endfor
   endfor

only one error per row or column.[4] However, both high efficiency and strong resilience could be achieved by hybrid methods that first attempt to recover using Algorithm 3 and resort to restoration when necessary.

Table 1 summarizes the memory requirement for various HSSmv algorithms presented in this section, including the extra storage needed for the FT algorithms. The second column is the baseline of the size of HSS matrix A itself. The next two columns contain various checksum vectors and the data size in the safe store.

We also explored the possiblity of appending checksum rows to matrices prior to HSS factorizatation. This approach does provide an invariant condition, but the invariant is preserved only within the interpolation error of the HSS factorization. Preliminary experiments indicated that these error bounds were not tight enough to permit efficient error detection.

3. IMPLEMENTATION AND EVALUATION

In this section, we discuss how we implement the new ABFT algorithms and how we assess their performance.

3.1 Containment Domains

Containment Domains (CDs) provide an API for An efficient implementation of our ABFT algorithms. CDs provide a software layer to facilitate the preservation-restoration model, including nesting control constructs, and durable storage. We used a newly developed, low-overhead implementation from UT Austin [16].

Containment domains are a programming construct that enables applications to express, tune, and specialize error detection, state preservation and restoration, and recovery to satisfy application specific resilience needs [4]. The following features are attractive for our algorithms. First, CDs respect the deep machine and application hierarchies expected in exascale systems. This matches well with our hierarchical algorithms. Second, CDs allow software to preserve and restore state selectively within the storage hierarchy, to support local recovery, which is desirable for large-scale applications, since it enables preservation to exploit locality of storage, and rather than requiring every process to recover from an error, limits the scope of recovery to only the affected processors. Third, since CDs nest, they are composable. Errors can be completely encapsulated, or escalated to calling routines through a well-defined interface. Finally, the CD interfaces are flexible enough to tailor error detection and recovery mechanisms to suit our needs. We can easily implement hybrid algorithms that combine both preservation-restoration and data encoding. Thus, we need only a minimum amount of rollback and re-execution.

Algorithm 4 lists the pseudo-code including the calls to the CD routines for Step 1 of the FT-medium-HSSmv algorithm (Algorithm 2).

Table 1: The storage requirement (in number of real values). Notation: n is the matrix dimension, b is the block size at the finest level partition, and r is the HSS-rank (assuming uniform everywhere). The total size of {U, V, B} generators plus D is $M = 18r^2n/b + nb$. The variable $s = rn/b$ is introduced for brevity.

	HSS matrix	Checksum	Safe store	Total
Without FT	M	0	0	M
Coarse	M	n	M+n	2(M+n)
Medium	M	n+2s	M+n+2s	2(M+n+s)
Fine	M	2n+5s	M+2n+5s	2(M+2n+5s)
Encoded	M	4n+10s	0	M+4n+10s

3.2 Performance metrics for resilience

The evaluation metrics for any fault-tolerant algorithm should include the following: the runtime overhead incurred by the resilience mechanism when no fault occurs (failure-free overhead), the runtime overhead when errors occur, as well as the memory cost introduced for fault handling. For runtime analysis, we divide the runtime of our algorithm into three states. A *working* state indicates normal work (forward progress) of our algorithm, without error checks. Failure-free overhead is represented by the *checking* state, which indicates that the algorithm is performing preventive action (checking), but no error was detected. We account for fault handling time in the *recovering* state, which includes both time spent detecting an error, and the time recovering from failures. We use the fraction of time spent working as our main performance metric.

We describe the performance of our algorithms with a vector $\Pi = (\Pi_W, \Pi_R, \Pi_C)$, where the subscripts W, R, and C denote the working, recovering, and checking states, and each component Π_i represents the fraction of time that the system is in the state i. Errors occur at random times, so the transitions among states form a stochastic process. A statistical model, such as a continuous-time

[4] A higher density of errors could be tolerated if additional checksum rows/columns were appended to the matrix.

Algorithm 4 Details of Algorithm 2 Step 1 including CDs.

Input:

- V_τ generators for leaf-nodes of HSS matrix A,
- a dense vector x
- weighted check vectors $\tilde{w}^{(L)}$ in Eqn. (9)

Output:

- $\tilde{x}^{(L)} = V^{(L)*}x$.
  Errors are recovered after the leaf-node calculations.

1. CD_Begin( )
2. //Preserve selected data for this CD.
   For every leaf node, τ:
   CD_Preserve(V_τ)
3. For every leaf node, τ, calculate:
   $\tilde{x}_\tau = V_\tau^* x(I_\tau)$
   $p_\tau = \tilde{w}_\tau^{(L+1)^*} x_\tau, \quad q_\tau = \tilde{w}^{(L)^*} \tilde{x}_\tau.$
4. //If the assertion fails, control returns to CD_Begin.
   //On second pass, CD_Preserve will restore data.
   CD_Assert($p_\tau = q_\tau$)
5. CD_Complete( )

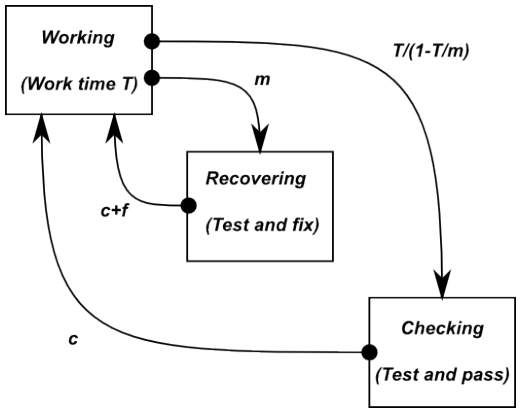

Figure 2: State transition diagram.

Markov chain, can be used to estimate the performance of such systems, by associating a transition rate with each allowed transition between states. The system remains working state for an average time T, after which it performs error-related work. Errors are discovered at a rate of $1/m$. In the Markov model, m represents the mean time spent in the working state before an error is found. It takes a time c to test for an error (check), and a time f to recover from (fix) an error. The time f includes any time for reloading data, performing rework, or correcting errors. When multiple recovery trials are necessary, perhaps due to errors during recovery, we incorporate these effects in this model by increasing f. If we use these rates in a Markov model of our finite-state machine, and solve for the steady-state probabilities, then we find:

$$\Pi = \left(\frac{m}{D}, \frac{f+c}{D}, \frac{(m/T)c - c}{D} \right)$$
$$D = m + f + (m/T)c \tag{14}$$

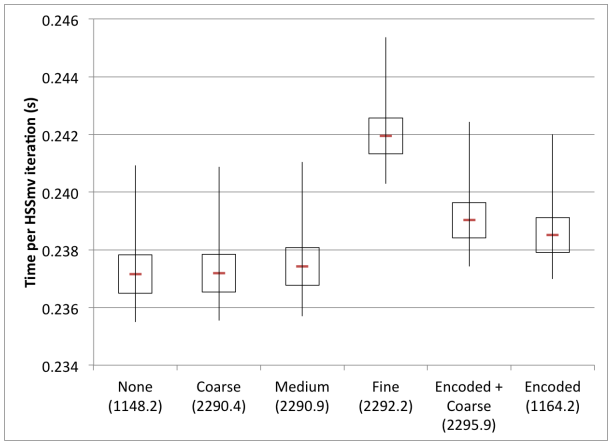

Figure 3: Runtime overhead for fault-tolerant HSSmv algorithms is smaller than the system's natural performance variation. Red markers are averages over eight runs. The fastest and slowest runs are indicated with vertical lines. Boxes show the standard error.

If no failures occur, then $m \gg T$, and from (3.2)

$$\Pi = \left(\frac{T}{T+c}, 0, \frac{c}{T+c} \right), \tag{15}$$

which matches what we expect intuitively. In the failure-free case, we spend no time recovering ($\Pi_R = 0$), but experience some slowdown due to spending c out of every $c+T$ cycles running checks. On the other hand, suppose every check fails, then $m = T$, and from (3.2) we find that

$$\Pi = \left(\frac{T}{T+f+c}, \frac{f+c}{T+f+c}, 0 \right). \tag{16}$$

The system executes for a time T before being taken out of service for a time $f+c$ for recovery. The quantity Π_W is analogous to the availability metric Λ used in two-state maintenance models. Noting that $f+c$ is the mean time for repairs (MTTR), and m is a mean time to fail (MTTF), from the first component of (3.2) we recover the familiar expression $\Lambda = MTTF/(MTTF + MTTR)$.

4. RESULTS AND DISCUSSION

We tested our FT HSSmv algorithms by injecting errors into the HSS factors at random times during the HSSmv operation. Before the HSSmv iterations begin, we register every U, V, B and D block with an ErrorInjector class. When matrix encoding is used, the entire encoded matrix is registered. The error injection module schedules 1,000 errors with frequencies sampled from an exponential probability distribution and memory locations uniformly distributed across the registered arrays. During the iterations, errors injection is triggered according to the schedule by comparing the current time to the schedule after each preserve call (or block-mv when matrix encoding is used). A new schedule is generated whenever the previous schedule is completed.

Our initial (uncompressed) matrix was a square H-matrix with side lengths of 20,000, generated for each off-diagonal block with rank of 5% of its size. The HSS construction used 1,000 random sampling vectors. The relative error in the matrix L^∞ norm due to compression of 10^{-10}. We then performed 10,000 iterations of the HSSmv algorithm, each with a different random vector.

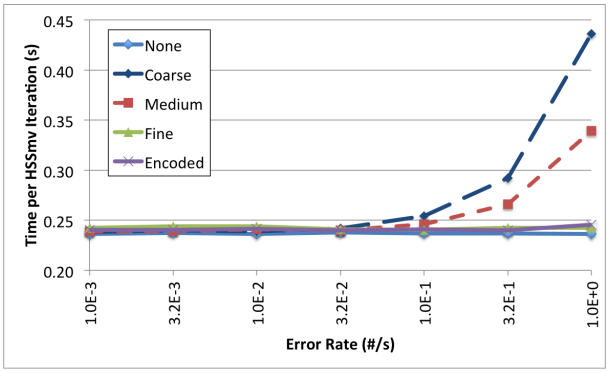

Figure 4: The runtime of fault-tolerant HSSmv algorithms increases with error rate. Fine-grained error checking minimizes recovery costs.

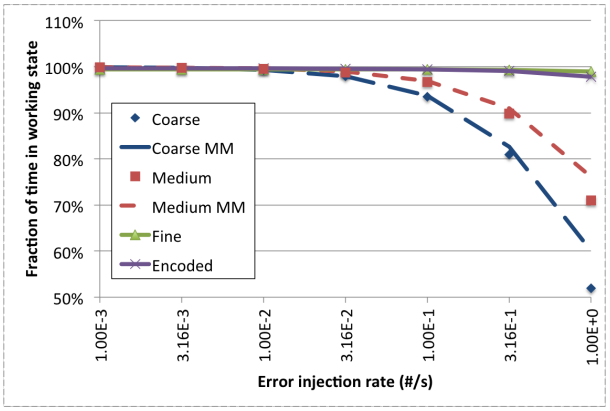

Figure 5: Efficiency of fault-tolerant algorithms as measured (points) and as predicted by the Markov model (lines).

Figure 3 shows the runtime overhead for each FT algorithm when no errors are injected. Encoded+coarse is a hybrid method that uses matrix encoding as its primary fault tolerance mechanism and resorts to coarse-grained restoration when FT-GEMM is unsuccessful. Note that these timings include only the HSSmv iterations and exclude *a)* the time to construct the HSS matrix (167 s), *b)* the time to store a nonvolatile copy of the HSS matrix (0.2 s), and *c)* time to precompute the checksum arrays (0.2 s). In every case, the additional costs for data preservation and error checking are less than 2% of the runtime without FT (labeled "None" in the figure). Runtime overhead for the fine-grained CDs is higher than the others, but is nevertheless small—nearly within the natural performance variation of the system. Figure 3 also lists in parenthesis, the total memory used by each algorithm. Memory use doubles when CDs are used because an additional copy of the matrix required for recovery. The fine-grained matrix encoding algorithm requires only 1% more memory than the non-resilient algorithm, but is somewhat less robust than using CDs (see Section 2.4).

The fine-grained error checking provided the best performance at high error rates. Figure 4 shows that the runtime of the coarse- and medium-grained FT algorithms increased steeply when the error rate exceeded the HSSmv iteration rate. At the highest error rate tested (1 per second), performance of the coarse-grained algorithm is roughly half that of the fine-grained CD and encoding schemes, which are nearly the same speed as the unprotected algorithm.

To extend our measurements to lower, more realistic error rates, we analyzed our timing data using a Markov model. During the runs described in Figures 3 and 4, we logged the value of *gettimeofday* for each error injection event or transition between work, check and recovery states of the FSM. We then post-processed this log to determine the total time in each state and the transition rate between states. Figure 5 compares the fraction of the total time spent in the working state as measured by the log files to the fraction predicted by the Markov model using the transition rates from the runs with 10^{-3} errors/second. Agreement between the two is excellent at low error rates. At higher error rates, errors may be injected during the recovery state and, when detected, cause reentry into the recovery state. This relationship between the recovery rate and the error rate is not accounted for by the three-state Markov model and explains the diminishing fidelity of the model at higher error rates.

5. RELATED WORK

Earlier work with ABFT for dense linear systems includes Huang's checksum encoding schem for error detection and recovery for matrix multiplication [6]. Chen used distributed checksums to enable tolerance of fail-stop failures on distributed systems [3]. Matrix encoding techniques were extended to LU and QR decompositions by Luk and Park, who used low-rang updates to acheive fast error recovery [10, 9]. This was applied to distributed parallel LU decomposition by Du [5]. FT algorithms have also been developed for two-sided decompostions such as Hessenberg reduction [7] and bidiagonal reduction [8].

A growing volume of work incorporates FT into iterative solvers with sparse matrices. Bronevetsky and de Supinski combined ABFT for encoding and detecting errors with checkpoint based recovery to tolerate errors in sparse iterative methods [2]. Shantharam developed a checksum encoding scheme to detect errors in sparse matrix-vector multiplication and triangular solve operations, and used these to construct a FT PCG solver [13]. Zhang combined an inner-outer solver (that is relatively insensitive to faults during inner iterations) with preservation-restoration techniques to build a FT iterative solver [21].

Our work is distinguished from other FT algorithms by its focus on the HSS matrix format and the use of CDs to perform fine-grained error correction *within* the HSSmv operation.

6. CONCLUSIONS AND FUTURE WORK

Our data shows that FT-HSSmv algorithms effectively handle errors at high error rates, about one error per second. Since the failure-free overhead of our algorithms is small, the overall runtime overhead when failures do occur remains small until the time between errors approaches the recovery time of the algorithm. We showed that the additional runtime costs of medium-grained and fine-grained error checking are small. Our analysis of these results with the Markov model shows that we can identify the limits of these algorithms' effectiveness by performing measurements at low error rates, and extrapolating these results to higher error rates. Finally, our experience developing these algorithms shows that the preservation methods are robust and straightforward, at the expense of requiring a form of safe storage, whereas the encoding techniques require no safe storage, but required changes to data structures and modifications to the algorithms to ensure that the additional encoding information is maintained.

We next speculate about what our results imply for making parallel HSSmv algorithms [12, 18] resilient. In a parallel HSS construction, the HSS blocks are distributed across processors. The steps in Algorithm 2 remain essentially unchanged. The coarse method,

for example, only requires collective agreement of the checksum condititon at the last step, even though pairwise exchanges are required at the non-leaf nodes to propagate the results between levels. As a result, we expect the error detection and recovery methods described here to be directly applicable. Further, we expect the computational and preservation costs for SDC detection to be similar, and the total rate of all (parallel) SDCs to be of the same order, a few seconds, as the rates described in this paper.

In the future, we will develop fault-tolerant algorithms for the other HSS matrix operations, including HSS construction and ULV factorization. Since all these algorithms follow the same HSS tree structure, our hierarchical style preservation-restoration can be employed in a similar fashion, but new invariant conditions must be developed for effective error detection.

7. ACKNOWLEDGEMENTS

Thanks to Francois-Henry Rouet and the CD team for giving us access to the pre-released HSS code and the CD library. Thanks to Osni Marques and Alex Druinsky for valuable discussions. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[2] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 155–164, New York, NY, USA, 2008. ACM.

[3] Zizhong Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1628–1641, Dec 2008.

[4] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC). IEEE.*, 2012.

[5] P. Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. In *Proc. of 2011 IEEE International Conference on Cluster Computing*, pages 272–280, 2011.

[6] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518–528, June 1984.

[7] Y. Jia, G. Bosilca, P. Luszczek, and J. Dongarra. Parallel Reduction to Hessenberg Form with Algorithm-Based Fault Tolerance. In *Proc. of SC13*, Denver, CO, USA, November 17-21 2013.

[8] Y. Jia, P. Luszczek, G. Bosilca, and J. Dongarra. CPU-GPU Hybrid Bidiagonal Reduction With Soft Error Resilience. In *Proc. of ScalA'13 (4th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems)*, Denver, CO, USA, November 17-21 2013.

[9] Franklin T. Luk and Haesun Park. An Analysis of Algorithm-Based Fault Tolerance Techniques. *J. Parallel and Distributed Computing*, 5:172–184, November 1988.

[10] Franklin T. Luk and Haesun Park. Fault-Tolerance Matrix Trianglations on Systolic Arrays. *Computers, IEEE Transactions on*, 37(11):1434–1438, November 1988.

[11] P.G. Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM J. Matrix Analysis and Applications*, 32(4):1251–1274, 2011.

[12] F.-H. Rouet, X.S. Li, and P. Ghysels. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Mathematical Software*, 2015. (submitted).

[13] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 69–78, New York, NY, USA, 2012. ACM.

[14] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, Pavan Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, Andrew A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, Sriram Krishnamoorthy, Sven Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. 2013.

[15] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[16] CD Team. Containment Domains API v0.1 (C++), 2014.

[17] GVR Team. Global View Resilience, API Documentation R0.8.1-rc0. Technical Report TR-2014-05, University of Chicago, University of Chicago, 2014.

[18] S. Wang, X.S. Li, J. Xia, Y. Situ, and M.V. de Hoop. Efficient parallel algorithms for solving linear systems with hierarchically semiseparable structures. *SIAM J. Scientific Computing*, 35(6):C519–C544, 2013.

[19] Panruo Wu and Zizhong Chen. FT-ScaLAPACK: Correcting Soft Errors On-line for ScaLAPACK Cholesky, QR, and LU Factorization Routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 49–60, New York, NY, USA, 2014. ACM.

[20] J. Xia. Randomized sparse direct solvers. *SIAM J. Matrix Anal. Appl.*, 34(1):197–227, 2013.

[21] Z. Zheng, A. A. Chien, and K. Teranishi. Fault Tolerance in an Inner-outer Solver: A GVR-enabled Case Study. In *Proc. of VECPAR'14 11th International Meeting on High Performance Computing for Computational Science*, Eugene, Oregon, USA, June 30-July 3 2014.