D6.6 Final report on deployment of improved software fault-tolerance techniques in the target WP3 application
Version 1.0

Document Information

Contract Number	610402
Project Website	www.montblanc-project.eu
Contractual Deadline	M24
Dissemination Level	PU
Nature	R
Author	Simon McIntosh-Smith (University of Bristol)
Contributors	Inria and BSC.
Reviewer	Gilles Sassatelli, Anastasiia Butko & Florent Bruguier (CNRS)
Keywords	Fault tolerance, resilience

Notices:
The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 610402.

Change Log

Version	Description of Change
V0.1	Initial Draft released for internal review
V1.0	Version for release to EU

Table of Contents

Executive Summary

Mont Blanc's WP6 was set up to address the problem of the expected greater error rates due to increased component counts, smaller silicon geometries and other factors, that are expected in future Exascale systems. D6.6 summarises the results so far from research into new fault tolerant iterative sparse solvers based on the Conjugate Gradient (CG) method. These types of solver are very commonly used in scientific applications, and so any advance in improving the built-in fault tolerance of sparse iterative CG solvers should have a material impact for Exascale systems and for several of the Mont Blanc applications. For example, Berlin Quantum Chromodynamics (BQCD), spends ~80% of its execution time in a CG solver, while EUTERPE also spends a significant portion of its run-time in a Jacobi Preconditioned Conjugate Gradient solver.

The results show that it is possible to build-in fault tolerance checks within a CG library such that the amount of performance impact can be traded off versus the level of fault protection provided. Performance overheads range from 5 to 20% dependent on the protection when measured at the pure solver level – once the overhead is amortized within the whole application it will represent a smaller fraction of performance. The results also suggest that a software-based fault tolerance scheme for sparse iterative solvers might be a faster, more energy efficient solution than a pure hardware ECC scheme. This is because application software can immediately respond to the error at the individual word-level, without having to invoke system-wide checkpoint/restart cycles when faults do occur.

1 Introduction

As microprocessor and memory devices adopt silicon manufacturing processes with ever smaller feature sizes, the likelihood of errors occurring in data sets due to transient faults increases. These faults have a large number of causes: they could be due to physical faults in the underlying hardware, faults in the software, or even transient errors due to a silicon device being struck by cosmic radiation or other electromagnetic interference. Memory devices are one of the main victims of this category of transient error. According to Zeigler and Lanford [1], errors affecting memory devices can be divided into two basic groups: hard errors are those caused by a physical defect, while soft errors are transient in nature and caused by some kind of electromagnetic interaction, such as a cosmic ray strike. Considerable work has been carried out on understanding the causes and effects of cosmic rays on silicon devices [1], [2], [3], [4], and in particular on their effects on DRAM devices [5], [6], [7], [8].

A 2009 study by Schroeder et al. at Google recorded between 2,000 and 6,000 memory errors per GByte of DRAM per year [9]. DRAM cells are now largely resistant to faults, due to the design of their capacitor-based cells [10]. While these errors are increasingly rare for DRAM devices, the sheer number of these devices in an Exascale system means that this class of soft error will always be a concern. Beyond DRAM, contemporary processors now include tens of megabytes of on-chip SRAM, which, as a transistor-based memory technology, is increasingly prone to errors caused by cosmic rays as the size of transistors continues to shrink. In practice, the use of the most common single error correct double error detect (SECDED) hardware improves the reliability of computer systems by detecting and correcting single bit errors (historically accounting for 98% of all memory errors [11]), as well as detecting (but not correcting) double bit errors.

Despite the refining of error correcting code (ECC) hardware mechanisms over the years, these can require significant extra hardware, storage and energy. For example, a typical single error correction, double error detection (SECDED) hardware implementation for a memory will require 8 bits of additional storage for each 64 bits of data, representing a 12.5% storage overhead for supporting ECC. In addition, the memory controller adds complexity in order to calculate the appropriate 8 parity bits whenever a 64-bit location is written to, and has to check the 8 parity bits whenever a 64-bit location is read from.

The additional hardware complexity and bandwidth requirement increase the energy required for every memory access by at least 12.5% [21]. Some implementations may provide the ECC support within the memory controllers themselves, but even in this case, there is 12.5% more data to move around, and this will take at least 12.5% more energy to move these extra bits. At Exascale, where the first systems have the challenging goal of using no more than 20MW of power, it is estimated that memories will consume 20% of this power, while the processors will be responsible for 52% of the total power consumption [12]. From the 52% used by the processors, one fifth is expected to be consumed by on-chip memories and off-chip memory bandwidth, and so in total, memory access will account for roughly 30% of the 20MW power budget of the first Exascale systems. Therefore, being able to reliably compute without ECC hardware could present a distinct advantage in terms of energy efficiency. As future hardware Mont Blanc machine may well continue to rely on embedded processor technologies. This ability to achieve the effect of ECC in software at reduced performance penalty would be advantageous. Our approach also benefits application performance, as errors caused by bit-flips will be identified — and in many cases corrected — within the sparse matrix solver itself, saving the substantial performance penalty that would have otherwise been necessary for an operating-system level ECC recovery scheme or for a machine-wide checkpoint-restart.

In this work we focus on sparse matrix calculations, where explicit matrix cell location data must be stored along with the respective matrix cell values. This is in contrast to dense matrix computations of dimension m × n, where the address of any element i,j in the array can be calculated simply from the base address, i, j, m and n. Thus dense matrices are stored as contiguous regions of memory, with the addresses of required cells trivially computed as they are required. With sparse matrices, typically only a small fraction of the matrix elements are non-zero, and so the matrix is stored as corresponding arrays of indices and values. Therefore, in a two-dimensional sparse matrix with one double-precision floating-point value and two 32-bit unsigned integer indices per non-zero element, there can be as much data to describe the structure of the matrix (two 32-bit indices per non-zero) as there is for the actual values of the non-zero elements of the matrix. Furthermore, when binary (or *pattern*) matrices are considered, the indices themselves represent *all* of the stored data.

Whilst there are iterative algorithms for sparse matrix calculations that are known to be tolerant of some errors caused by bit-flips in the element *values* [13], there is almost no existing literature which considers the effect of bit-flips on the element indices, which as we have established, can themselves represent up to half of the data in a sparse matrix – for applications which use sparse matrix solvers, the sparse matrices might typically be the majority of all data in memory, and thus any soft error that does affect a running sparse matrix solver is likely to affect the sparse matrix itself, with roughly equal probabilities of a soft error effecting the index data vs. the value data. An uncorrected bit-flip in an index within a sparse matrix structure could cause a catastrophic failure, potentially resulting in a memory fault and a subsequent checkpoint-restart recovery sequence. Perhaps worse, this kind of data corruption could instead result in a silent error, where incorrect results might erroneously be believed to be valid. This class of sparse-matrix memory index error has largely been overlooked in the literature to date, yet for any bit-flip affecting a sparse matrix, that bit flip is equally as likely to occur in the 64-bits of index data as it is in the 64-bits of value data.

2 **Related Work**

While almost no literature has yet addressed the issue of errors in sparse matrix index data, there is a growing body of work exploring the potential benefits of software-based fault tolerance techniques. Recent work by Hukerikar et al. on software-based resilience to bit-flips has explored high-level, transparent techniques that enable a software developer to specify which variables and operations within their code *must* be performed in a resilient manner, and which have some degree of natural fault tolerance [15]. Mitigating the effects of bit-flips through the use of type qualifiers and a library of resilient functions, this approach allows the user to tailor their code to either ensure accuracy, or to mask those bit-flips which would have a negligible effect on their program. In Hukerikar et al.'s work, the focus is on masking bits where a flip can be tolerated — in the unused most significant bits of the index data or in the relatively insignificant lowest bits of the floating-point data. By contrast, in our work we use the constraints that naturally arise due to the structure inherent in most sparse matrices to identify when a bit flip has occurred.

While no previous work has directly considered the effect of bit-flips on sparse matrix index data, Elliott and Mueller have analyzed the effects of bit-flips on floating-point values [16]. Their work showed that a bit-flip in a floating-point value has a high probability of occurring in the least significant bits of the fraction, and thus many of these events would likely result in small rounding errors which have little effect in many kinds of scientific calculation. In our work we apply a similar approach but to the index values of the matrix, and then develop mechanisms to spot and potentially correct such errors when they occur.

Other work by Maruyama et al. has looked at the need for fault tolerance in software running on commodity GPUs [17], which are increasingly being used to help accelerate HPC applications. Commodity GPUs, unlike their higher-end HPC counterparts, tend not to include hardware ECC support, and so any bit-flips that occur in the GPU DRAM or on-chip SRAM are potentially a serious problem. To address this issue, Maruyama et al. have developed schemes that combine elements of software ECC and grid- based parity check bits with checkpoint-based methods. Our work differs in that it presents solutions for sparse matrix methods running on any hardware platform. Our methods can be used to help augment alternatives to ECC hardware solutions, efficiently protecting sparse matrix indices and values, with no additional storage overhead and low performance overhead. As such, our methods can benefit any processors without hardware ECC support, such as commodity GPUs or consumer-level CPUs.

3 Sparse Matrix Indexing

Throughout rest of this report we work with Fortran-style numbering of arrays, i.e., an array of length N starts with element 1 and runs to element N.

The amount of data used to store the row and column locations in sparse matrices is significant yet often overlooked. From a study of over 2,600 sparse matrices from the University of Florida's Sparse Matrix Collection [14], [18], the average percentage fill of a given sparse matrix was ~2%, although the median fill rate is much lower, at just ~0.24%. This low fill rate is why it is much more efficient to store only the non-zero elements in a sparse matrix, and thus why they are commonly stored in compressed formats, such as the so-called Coordinate (COO) format or the Compressed Sparse Row (CSR) format, which both store location data for each (non-zero) matrix element, along with the (non-zero) matrix values themselves[1].

In the rest of this section we present a detailed analysis of the effects of bit-flips on sparse matrix index data, before moving on to develop new techniques for detecting and correcting these errors which require no additional storage and have a low performance penalty. We believe that this work is the first that considers the issue of errors in sparse matrix index data.

3.1 Sparse Matrix Storage Formats

The simplest sparse matrix storage scheme is the COO (Coordinate) format [19], whilst the CSR (Compressed Sparse Row) format is often used for its advantages over COO in terms of space and simplicity in indexing.

In COO, when working with two-dimensional matrices, there are three arrays of length NNZ (Number of Non- Zeros): two 32-bit unsigned integer arrays to hold the index dimensions and a (single- or double-precision) floating-point array to hold the value at each non-zero point. (If the matrix is only filled with integers, this third array could also be an integer array, whilst if it is a binary or pattern matrix, there is no need for the third array at all.)

In CSR, when working with two-dimensional matrices, there are two arrays of length NNZ whilst the third is of length m + 1 (for a matrix of dimension m × n). As before, we have one array that stores the values at each non-zero point, whilst the other two arrays store sufficient data to calculate the index dimensions. In this case, the index array of length NNZ holds the column index of each non-zero element. The (m + 1)-length array stores the position in the other two arrays of each element that starts a new row, with the (m + 1) element always set to NNZ + 1.

[1] https://en.wikipedia.org/wiki/Sparse_matrix

$$A = \begin{pmatrix} 2.4 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.5 & 0.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.8 & 0.0 \\ 0.0 & 0.0 & 0.0 & 3.3 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.7 \end{pmatrix}$$

Figure 1 – An example sparse matrix

We can illustrate the COO and CSR schemes by considering the matrix in Figure 1. In the COO scheme, we represent the non-zero elements using two coordinate vectors, x and y, as well as the array of non-zero values, v, to which x and y directly correspond (i.e., v[i] is located at (x[i], y[i])). Here we store the matrix A as in Table I.

Table I - The sparse matrix A in the COO format.

v	2.4	3.5	4.0	1.0	1.8	3.3	1.0	0.7
x	1	2	2	3	3	4	4	5
y	1	2	4	3	4	4	5	5

In the CSR scheme, we again have an array of values, v, as well as two coordinate vectors, but one of the coordinate vectors (x_{off}) represents the m offsets into v of the starting elements of each row, along with the (m+1) element storing the value NNZ + 1. This gives us v[i] located at (j, y[i]), where j is the unique integer that satisfies the relation x[j] ≤ i < x[j + 1]. Thus we store the matrix A in the CSR scheme as in Table II. The CSR scheme assumes that there are no empty rows in the matrix.

Table II - The sparse matrix A in the CSR format.

v	2.4	3.5	4.0	1.0	1.8	3.3	1.0	0.7
x	1	2	4	6	8	9		
y	1	2	4	3	4	4	5	5

In the next section we will show a range of valid index constraints that naturally arise from the structure often inherent in sparse data sets. We will initially present our new methods in relation to the simpler COO format, before showing how they extend to CSR.

4 Sparse Matrix Index Constraints

4.1 Exploiting Sparse Matrix Size Constraints

A first constraint that must always apply is that the indices i, j corresponding to each element a_{ij} of an m × n sparse matrix A must have a value between one and the size of their dimension. Intuitively, this is obvious: assuming that each non-zero element e_{ij} is located at (x_k,y_k),where $x_k = x[k]=i$ and $y_k = y[k]=j$, then

$$0 < x_k \leq m \qquad (1)$$
$$0 < y_k \leq n \qquad (2)$$

While this might not at first appear to be that useful, an analysis of the University of Florida's collection of ∼2,600 sparse matrices shows that the largest dimension of any of their sparse matrices is ∼118 million, which requires unsigned indices of 27 bits. However, matrices of this size are rare in the collection, and excluding the largest 13 matrices reduces the maximum dimension to 16,777,216 (i.e., 2^{24}), which needs only 24 bits for representation. The important implication is that while there are 2^{32} valid unsigned integers, the range of valid indices is much smaller, the exact number depending on the maximum size of the matrices under consideration.

We can apply this constraint to immediately improve the fault tolerance of a sparse matrix-based computation. As a sparse matrix is processed, we check that each of the indices still satisfies their relevant constraint as defined in either Equation 1 or 2. This method will automatically detect any bit-flips that have occurred in the most significant part of each index: those in the bit positions higher than those used to represent the size of each dimension. In our examples from the Florida collection, this approach will detect bit-flips in at least 5 of the 32 index bits (15.5% of potential index bit-flips), or at least 8 of the index bits (25.0%) if we exclude the largest 13 matrices. The method is even more effective for smaller matrices.

In addition, these single-bit flips, once detected by the constraint, could also be corrected in a similar manner to the masking used by Hukerikar et al.; zeroing the desired number of bits in the most significant 'guarded' range of the index would restore the correct index value, and the sparse matrix calculation could continue without interruption, and without resorting to a checkpoint-restart sequence. When the size of each dimension is not an exact power of 2, these constraints will detect more errors than simply masking the top bits; whilst the index cannot be guaranteed to be corrected, the error can be detected if it violates one or both of the constraints.

The overhead of checking the constraints is low. The constraint checking can be implemented as either a simple conditional test requiring a 32-bit unsigned integer comparison, or it can be implemented as a bitwise test, whichever is fastest on the target architecture. This constraint checking can also be performed in parallel with other parts of the sparse matrix computation. For example, if one were to assume that bit-flips in the indices would not result in memory faults, but would merely index the wrong data, then one could envisage an implementation where the indices of an element would be used to access the element value, likely requiring a long-latency memory load from DRAM, during which time the indices could be checked against the relevant constraints. If during the constraint checking an error was found, the indices could be corrected and the element value load reissued, with the erroneous element value discarded. In general, it should be possible to overlap constraint checking with element value loads, as the latter are likely to be cache misses and require long latency DRAM accesses. Our new approach should be considerably faster than checkpoint/restart schemes, as only a few simple tests are needed and only a small, local correction required to a single 32-bit index in the event of an error.

4.2 Exploiting Sparse Matrix Ordering Constraints

Having established that we can use a sparse matrix's size to define constraints on valid element indices, and that these constraints can subsequently be used to detect and potentially correct bit-flips in the index data, we can now look for more opportunities to establish further constraints that would enable us to detect a wider range of errors.

The first set of constraints was simple and considered only the matrix size. A second simple yet useful set of constraints can be applied to sparse matrices in the COO format when the non-zero elements are stored in a consistent order. Assuming the indices are always in increasing order (an assumption that is true for all ∼2,600 sparse matrices in the University of Florida collection), this gives a monotonically non-decreasing sequence for the major dimension and, within that, a strictly increasing sequence for the minor dimension (the wrap-around when the major dimension increases being the only exception to this constraint). If matrices are stored in this fashion, it gives us two new simple constraints that must always be true:

$$x_{k-1} \leq x_k \leq x_{k+1} \qquad (3)$$
$$y_{k-1} < y_k \quad \text{when} \quad x_{k-1} = x_k (4)$$

where $1 < k < NNZ$. These constraints will be useful if a bit-flip in an index causes it to break the apparent ordering of the indices. The probability that a bit-flip in an index will cause a violation of these new constraints depends on the nature of the indices in real sparse matrices. In practice, constraints 3 and 4 can provide significant additional index data error detection and, potentially, an aid to correction. Further, the major index has even greater protection since its non-decreasing sequence changes much more slowly and — for non-degenerate matrices — changes by at most one. This means that the vast majority of bit-flips in the major index (31 out of 32, or ∼97%) would violate a constraint and would therefore be detectable and trivially correctable.

So far we have only considered sparse matrices in the COO format. When we are dealing with matrices in the CSR format, we can add to the above constraints since we would have strict inequality in the x_{off} array:

$$x_{off}[k] < x_{off}[k+1] \qquad (5)$$

where $1 < k \leq m$, with m the number of rows. However, it should be noted that in this case the changes to x_{off} will be at least one, and often greater.

Ordering constraints 3 and 4 can be implemented in a similar manner to the simple dimensional constraints (1 and 2) described earlier, using simple 32-bit unsigned integer comparisons, executed concurrently with the element value accesses themselves. When an ordering constraint violation is detected, it is potentially possible to correct the error. If we assume that only a single bit has been flipped in the index that has violated the ordering constraint, in many cases it will be possible to determine which bit has caused the violation.

4.3 Exploiting Sparse Matrix Symmetry Constraints

So far the constraints we have developed have depended only on the size and ordering of the sparse matrix elements; the constraints have not considered any other structure that might be present in the data. From a statistical analysis of the sparse matrices in the University of Florida's collection, we see that there is often significant structure that underlies the matrix data. For a number of numerical algorithms, such as those derived from stencil-based methods, this intuitively follows, as the non-zeros in the sparse matrix will occur in a regular pattern that is some function of the stencil --- a 2D 5-point stencil operation usually results in a sparse matrix with a non-zero band three wide centred on the diagonal, with two more non-zero diagonals, one either side of the diagonal, a short distance away. Sparse matrices often have other useful structure, such as symmetry.

Symmetric matrices have the advantage that only half the off-diagonal elements need to be stored, reducing the overall storage by nearly 50%. It also presents a new constraint for us to add to constraints 3 and 4. For COO formatted matrices, assuming that the non-zero elements are stored in the same fashion as those of a lower-triangular matrix, we have a constraint involving both the x and y index arrays:

$$x[k] \geq y[k] \qquad (6)$$

where $1 < k < NNZ$. (This constraint does not have much meaning when applied to CSR formatted matrices, since x_{off} represents an offset within the other two arrays, rather than a fixed index value.)

There are a few advantages of storing only half the off-diagonal elements of symmetric matrices. First, there is the obvious savings in space of roughly one half. Secondly, the probability of a bit-flip affecting an element's index or value is also approximately halved, since the matrix only occupies half as much space. Finally, the use of constraint 6 can reduce the number of susceptible bits in a given index by at least 1. For example, we see this if we have a non-zero element on a diagonal with at least one valid index (along the minor dimension) between it and the next non-zero element. In this case, at the times when the least significant bit is zero in this index we now have additional protection for the least significant bit. Similarly, if any of the most significant zero bits could have been changed (if the matrix had been stored fully, as with an unsymmetric matrix) and still yield a valid index, these bits too would now be protected by constraint 6.

4.4 Exploiting Sparse Matrix Banding Constraints

Banded matrices are another common type in the Florida collection. Banded matrices comprise of regions of non-zero elements that are grouped along diagonals, often the main diagonal. These regions of the matrices, if viewed alone, appear densely packed. Banded matrices can also be exhibited through blocks of diagonals, yielding (banded) block-diagonal matrices. In these sparse matrix structures, values of index positions that are between the extremities of the band (the *left* and *right bandwidths*) have little freedom for a bit-flip to affect their index data yet remain bound within the band.

When looking at banded matrices, constraints 3 and 4 can detect bit-flip errors in a majority of the bits of the indices, with only errors in the least significant index bits potentially going undetected. The more densely packed the elements within the band, the greater the number of index bits that will be protected by our scheme. With such close non-zero index grouping, the more chance an index has of being correctable if and when a detectable bit-flip occurs. For bit-flips that occur in the most significant index bits for elements within the bandwidth (a tighter bound than the dimension), there is a higher probability of successful correction since there is a higher probability that there is only 1 bit of the index which could be re-flipped in order to satisfy all the constraints. Conversely, the more sparsely distributed the elements within the band, the less the protection the constraints can offer.

Considering specific classes of banded sparse matrices, we can develop tighter bounds for the ordering constraints. In a single-banded matrix, or one that is close to it in structure, the number of valid index positions within the band will likely be small, typically under 1% of the appropriate dimension. This structure results in the non-zeros being very close together within the bands, with subsequently close indices. For sparse matrices in the Florida collection, which are all of dimension up to 2^{27}, this would mean that a bit- flip to any of the most significant 7 bits of the index (top 12 bits of the whole 32-bit number) would immediately violate a constraint, and thus over a third (37.5%) of all potential single bit-flip errors affecting the indices could be corrected by this single constraint alone. The smaller the bandwidths, the tighter the effect of the constraints.

In this way, bit-flips to the guarded index values are limited in the damage they can cause, with spatially closer indices better protected against bit-flips than those spread further apart. If the non-zero elements are consecutively spaced within the bands, the danger of bit flips in the indices is removed almost entirely; single bit-flips in those elements not at either bandwidth extremity would violate one of constraints 3 or 4 and be both detectable and correctable, whilst those occurring in the few elements at the band extremities would have an improved chance of detectability and possibly correctability. For non-zero elements that are more spread out, the danger of a ‘silent’ bit-flip in an index which creates another legitimate index that retains the same ordering is increased. In this case, the index bits — especially the least significant ones — are more vulnerable and may require further protection.

The discussion above can be generalized to matrices with more than one band. The greater the distance between the bands, the more likelihood there is of bit-flips to the indices of the band extremities changing into another valid index, and therefore causing a change in an index that our constraints would not detect. However, the more bands there are, the smaller these distances may be, so despite matrices with a greater number of bands containing a greater number of susceptible elements (those on the band extremities), the smaller the chances are that these susceptible elements are affected by bit-flips without violating constraints. In the vast majority of cases the number of non-zero elements at the band extremities is a small proportion of the total number of non-zeros, and therefore banded sparse matrices should benefit even more from our scheme than non-banded ones.

5 Further Index Fault Tolerance Techniques

The largest sparse matrix in the University of Florida collection has a maximum dimension of $\sim$118 million, and all the rest are at most of dimension $\sim$50 million. These can be represented with 27 bits and 26 bits respectively, thus leaving the top 5 or 6 bits available for alternative uses. In fact, if we eliminate the thirteen largest matrices (representing 0.5% of the entire collection), we need only 24 bits to represent each of the indices, freeing up the most significant byte of each index for alternative uses. These most significant unused index bits could therefore be used for additional fault tolerance schemes, such as storing a redundant copy of a selection of the rest of the bits of the index. A redundant copy could be easily exploited via a simple masking and bit-wise comparison process, requiring no additional data.

In this way, we can provide redundancy for those parts of the index that would benefit most from it. For those matrix elements that are closer together, this may be the most significant index bits, whilst for sparse matrices with a greater 'spread' (or lack of inherent structure) protecting the least significant bits of the index with a redundant copy may be of more use.

An alternate use for the spare index bits would be to provide some form of soft SECDED scheme, either on a per-index basis, or perhaps even at the level of a compound sparse matrix element (two 32-bit indices and one 64- bit value, 128-bits in total). This alternate approach might potentially provide even greater levels of protection, but might also require higher levels of overhead to compute and check the ECC codes for each element being protected. We are currently investigating this approach and will report our findings in a future report.

An additional technique that would improve the efficacy of our scheme would be to apply transforms to the matrix to change its structure into a form that makes it even more 'constraint friendly'. This might be in the form of a preconditioner which modifies the structure of the matrix into some equivalent form, but which will better exploit the constraints. For this to work, the preconditioner could, for example, move elements closer together or introduce more bands. The preconditioner would be focused on the fault tolerance of the structure of the matrix, rather than the more traditional uses of preconditioners, which typically focus on improving the convergence of the solver.

6 Results

In the following section we detail the results from applying all of the COO-applicable constraints detailed above (Equations 1 – 4). These constraints were applied to all of the real matrices within the University of Florida sparse matrix collection – roughly ~60% of the 2,600 example matrices. We found that, as expected, matrices exhibited different levels of amenability to our constraint-based fault tolerance scheme, with some benefiting extremely well and others receiving much less protection. In this section, we present results for the number of protected bits (Pb) in the indices from three exemplar sparse matrices chosen to illustrate the benefits our scheme can deliver across a range of different matrix sizes and types (small, medium and large). These examples were also chosen as being representative of the results we observed across the whole collection.

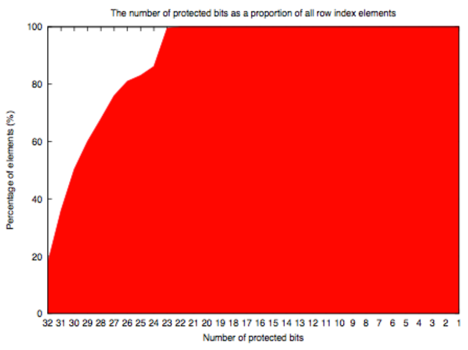
In each of the three cases, we provide two graphs. The first of each pair is a graph which shows the percentage of bits in each index which are protected by applying constraints 1 – 4 from our scheme. If our scheme was to fully protect all bits in every index, this graph would show 100% across the full range of index bits on the x axis. If our scheme protected only the least significant 16 bits in every index, it would show 0% for 32-17 bits and then 100% for 16-1 bits. If half the indices had every index bit fully protected and the other half had no index bits protected at all, then the graph would show 50% right across the x axis. The ratio of the red area under the line (the protected bits) to the white area above the line (the unprotected bits) shows how well the scheme is doing.

The second graph in each pair is a histogram based on the bit specific positions in each index which remain unprotected by our scheme. This graph takes into account the susceptible bit positions of each of the indices in the matrix and, assuming that a bitflip error occurs, plots the expected frequency (or likelihood) in which each susceptible bit position would be altered on the y axis against the susceptible bit positions in the x axis. A bin showing 15% for bit position 1 means that across all the susceptible bit positions in all of the indices there is a 15% chance that an occurring bitflip would alter the least significant bit of an index. The more skewed the graph is, the greater the likelihood that bitflips will affect specific bits. If our scheme worked perfectly one would see all bit positions showing 0%.

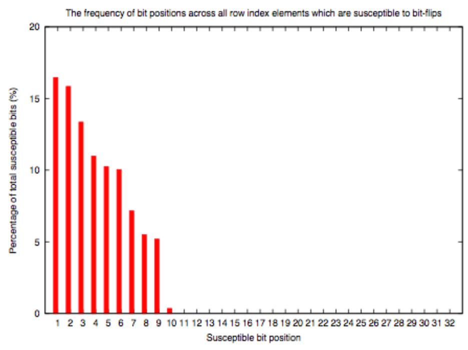
The first pair of graphs is from a small, unsymmetric, rectangular sparse matrix, named lp_agg in the University of Florida collection. This matrix is relatively small compared to the rest of the collection, being just 488 × 615 in size and containing 2,862 non-zero elements (a 0.95% fill rate). Due to its relatively small size, there is very little clustering of this matrix's non-zeros, and therefore many relatively large gaps between elements. Figure 2a shows that, whilst all elements have at least 22 bits of each index fully protected from our first set of constraints, the other constraints in our scheme make less of a difference in this scenario. Even with this very small size our scheme has protected over two thirds of the index bits, and from Fig. 2b, one can see that the index bits that remain unprotected are almost all in the least significant 9 bits of the indices. Even though our scheme has worked well in this case, a matrix as small as this is not really a candidate for an Exascale-class computation.

The second pair of graphs is from an average sized, symmetric, square sparse matrix, named nasasrb. This matrix is of dimension 54,870 × 54,870 elements, and contains 2,677,324 non-zeros (a 0.089% fill rate). Here we see an excellent result for our scheme. Figure 2c shows that the first set of constraints fully protect 16 bits of every index, with the remaining constraints fully protecting an additional 4 bits, as well as ensuring nearly 70% of all elements are fully protected. One of the reasons we achieve such a strong result in this case is that this matrix's symmetry enables additional constraints and thus protection. Very few index bits are left unprotected in matrices of this class. Figure 2d shows that the majority of the index bits that remain unprotected are in the least significant byte of the index.

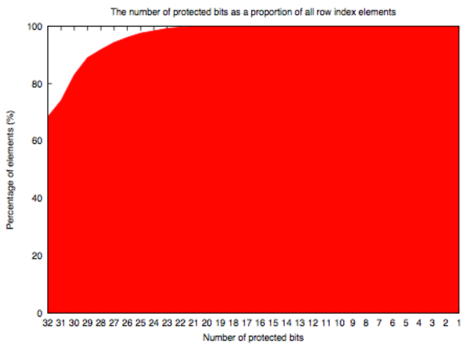
The final pair of graphs is from a large, non-symmetric, square sparse matrix, named circuit5M. This is one of the largest real matrices in the collection, being 5,558,326 × 5,558,326 elements in size and containing 59,524,291 non-zeros (a 0.00019% fill rate). Our scheme also works well with this matrix. Figure 2e shows that we are able to fully protect all 32 bits of around 43% of all indices, and in the worst case any single index has at least 10 bits protected by our scheme. Figure 2f shows a fairly even spread of index bits which remain unprotected, with each susceptible bit position almost as likely as any other (around 5% likelihood). The fraction of index bits left unprotected then drops off sharply, and by bit position 24 all the remaining index bits are fully protected in every index.



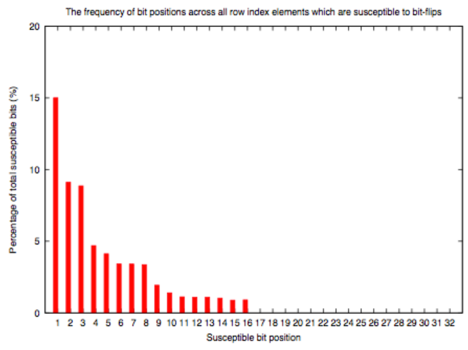
(a) Pb graph for a small, unsymmetric matrix (lp_agg).



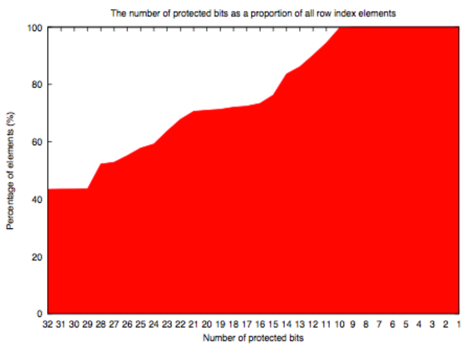
(b) Index bit positions (lp_agg).



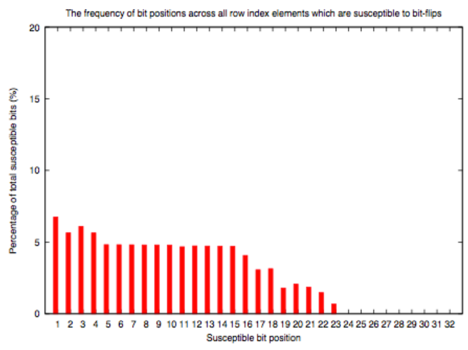
(c) Pb graph for a mid-sized, symmetric matrix (nasasrb).



(d) Index bit positions graph (nasasrb).



(e) Pb graph for a large, unsymmetric matrix (circuit5M).



(f) Index bit positions (circuit5M).

Figure 2 – Graphs for three representative (real) matrices of different sizes from the University of Florida collection.

7 Discussion

The work presented in this report has focused on developing software-based fault tolerance techniques for spotting and in some cases correcting single-bit errors in the index data associated with sparse matrices. As yet we have not addressed multi-bit errors in this work, but observed that multi-bit errors are both much less likely to occur (less than 2% of all memory errors according to [11]), and easier to spot and correct with our scheme (multi-bit errors tend to affect adjacent bits and thus will create larger changes in afflicted indices which will be more likely to violate a constraint).

The overheads of the constraint checking scheme are yet to be rigorously investigated. An early prototype in Bristol suggests an upper bound of ∼10% on the performance of a CG solver, but we estimate that careful coding should bring this down to closer to ∼5%. This overhead needs to be balanced against the overhead of the alternate checkpoint-restart scheme, which is widely predicted to be increasingly costly for Exascale machines, potentially requiring a performance overhead of ∼20% or more. The performance of our scheme in the presence of faults is also much greater than a checkpoint-restart traditional approach: in our scheme, a user-level software routine can discover exactly which 32-bit index has become corrupted and can restore just this one word. In contrast, a traditional checkpoint-restart scheme involves heavyweight OS-level intervention and potentially a large amount of data transfer from disk across the network.

The potential energy savings of our software-based fault tolerance scheme had yet to be quantified, but we can already show that our scheme reduces the amount of data that needs to be moved to support ECC and instead performs an increased number of simple arithmetic operations (32-bit integer subtracts, compares and conditional tests). As data movement is expected to be energy expensive compared to integer operations, this is the right trade-off to make in the bid to improve energy efficiency for Exascale.

The results show that a purely constraint-based scheme protects the majority, but not all of the index bits in a sparse matrix. This is an interesting result. To fully depend on this scheme as an alternate to hardware-based ECC, one might prefer that all indices should be fully protected. Further investigation may show that there is an acceptable tradeoff – a combination of judicious matrix preconditioning and constraint selection might deliver close to 100% index protection and this might be good enough for all but the most critical scientific applications, if the performance and energy efficiency benefits are significant enough. For the cases where 100% protection is absolutely necessary, we are exploring the potential for using the unused index bits for a completely software-based SECDED scheme that protects at the sparse matrix compound element level. This should provide the same level of soft-error protection as hardware-based ECC while retaining the benefits of reduced data movement and machine-wide operating system intervention when an error occurs. We will study this point and publish the results in a future report.

To conclude, when working with sparse matrices whose elements are stored in a consistent order, we have shown that it is possible to define schemes to detect and potentially correct bit-flip errors from affecting the underlying index bits. Through a series of constraints, we can help to ensure that the indices which locate the data within the sparse matrices are not inadvertently changed by bit-flips. Whilst these constraints in isolation do not guarantee a full protection from bit-flips in the indices, it is possible to catch a majority of index errors in this way. The constraints can be implemented with simple operations, and can help to avoid the otherwise costly checkpoint-restart alternative to dealing with faults. The Application-Based Fault Tolerance (ABFT) techniques we have presented in this paper could be added to existing sparse matrix solver libraries, and would help improve their built-in fault tolerance as a step towards making Exascale-class computations a reality.

8 Considerations for Fault Tolerant Mont Blanc Applications and Future Work

We now have a well-developed scheme for a fault-tolerant CG solver. This CG solver could use the index-protecting criteria described above, and also the software-based SECDED scheme which we have mentioned in brief and that will be implemented in the next stage of our work. Early stand-alone prototypes of a fault-tolerant CG solver suggest that, considering the time for the CG solve in isolation, the overhead of the FT scheme is in the region of 10-20%, depending on the level of protection required.

The next step will be to plug these techniques into the CG solvers of representative applications. Several Mont Blanc applications rely on CG, including BQCD, which spends ~80% of its time in an appropriate solver. The FT overhead should be less once integrated into the rest of a real application, and so this is a set of measurements we wish to take soon. Selection of Mont Blanc applications for testing will take place at the October face-to-face project meeting at Jülich in Germany.

These ABFT techniques are complimentary to other FT techniques. For example, D6.7 is developing nano-checkpointing, a method which should significantly reduce the overhead of using checkpoints. The techniques presented in this report could be readily integrated alongside this approach, providing further benefits and reduced overhead.

The Bristol team has also been involved in developing the Mantevo[2] suite of mini-apps. In particular we have contributed TeaLeaf, a 2D/3D heat diffusion benchmark, which spends the vast majority of its time in a sparse iterative CG solver. We have developed several different solvers for this work, including a communication-avoiding Chebyshev Polynomially Preconditioned CG (CPPCG) solver, which would also make an ideal candidate testbed for our FT-CG work. We will explore the effectiveness of our FT techniques on TeaLeaf's CPPCG solver in addition to the Mont Blanc applications. We have recently demonstrated TeaLeaf's ability to strong scale across thousands of nodes of Titan at Oak Ridge and Piz Daint at CSCS. So TeaLeaf will also make a good backup plan if integrating our FT techniques into one of the other Mont Blanc applications proves too difficult to achieve for any reason. A paper describing the TeaLeaf work has been submitted to the PMBS workshop at SC'15.

Beyond FT techniques for sparse iterative CG solvers, Bristol has also been developing FT techniques for an N-body class code. We have developed the BUDE molecular docking code [20], a new application for Mont Blanc 2. BUDE uses an evolutionary algorithm to find the

[2] https://mantevo.org

optimal 'pose' for a drug molecule to bind, or 'dock' with a target molecule, usually a protein of medicinal interest. The evolutionary algorithm within BUDE exhibits some properties that should give it natural fault tolerance – many poses are docked and evaluated, with the best, or 'fittest', being used to spawn subsequent generations of poses to evaluate. If some fraction of poses become corrupted or even lost altogether, this should have a limited impact on the overall outcome of the algorithm.

BUDE's implementation does not expose the algorithm's natural fault tolerance yet, so as future work in the third period of the Mont Blanc project we are going to modify the software to enable the following:

1. We will investigate what fraction of each population could be lost before the evolutionary optimization fails – with the FT-capable version of BUDE we hope to make these measurements.
2. We will also explore the impact of silent data corruptions (SDCs) on individual poses – if we corrupt the position or orientation of a pose, is it then naturally ignored by scoring badly, or is there a chance that the corrupted pose might look artificially good, generating a false optimum?

These are the questions we'll ask and hopefully answer in the third period of the project, but we believe that this class of naturally Application Based Fault Tolerant (ABFT) techniques should show great promise.

9 Acknowledgements

The work described in this report has been supported by funding from the European Community's Seventh Framework Programme [FP7/2007- 2013] under the Mont-Blanc 2 Project (www.montblanc-project.eu), grant agreement no 610402, and by the UK funding council, EPSRC. We also received financial support from NAG Ltd.

10 References

[1] J. Ziegler and W. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, November 1979.

[2] J. Ziegler, "Terrestrial cosmic rays," *IBM Journal of Research & Development*, vol. 40, no. 1, pp. 19–39, January 1996.

[3] J. Ziegler, H. Curtis, and et al, "IBM experiments in soft fails in computer electronics," *IBM Journal of Research & Development*, vol. 40, no. 1, pp. 3–18, January 1996.

[4] J.ZieglerandW.Lanford,"The effect of sea level cosmic rays on electronic devices," *Journal of Applied Physics*, vol. 52, no. 6, pp. 4305–4312, June 1981.

[5] L. Borucki, G. Schindlbeck, and C. Slayman, “Comparison of accelerated DRAM soft error rates measured at component and system level,” in *IEEE International Reliability Physics Symposium*, 2008.

[6] W. McKee and H. McAdams, “Cosmic ray neutron induced upsets as a major contributor to the soft error rate of current and future generation DRAMs,” in *IEEE International Reliability Physics Symposium*, 1996.

[7] Y.-P. Fang, B. Vaidyanathan, and A. Oates, “Soft error rate cross-technology prediction on embedded DRAM,” in *IEEE International Reliability Physics Symposium*, 2009.

[8] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, “Cosmic rays don’t strike twice: Understanding the nature of DRAM errors and the implications for system design,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[9] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM errors in the wild: a large-scale field study,” in *ACM International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.

[10] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, “The international exascale software project roadmap,” *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[11] IBM, “Fault tolerance decision in DRAM applications,” IBM, Tech. Rep., 1997.

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[13] M. Hoemmen and M. Heroux, “Fault-tolerant iterative methods via selective reliability,” Sandia National Laboratories, Sandia National Laboratories, Albuquerque, NM 87175, Tech. Rep., June 2011. [Online]. Available: http://www.sandia.gov/ maherou/docs/FTGMRES.pdf

[14] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Soft- ware*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[15] S. Hukerikar, P. Diniz, and R. Lucas, “A programming model for resilience in extreme scale computing,” in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, 2012.

[16] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, “Quantifying the impact of single bit flips on floating point arithmetic,” Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, Tennessee 37831, Tech. Rep. ORNL/TM-2013/282, August 2013. [Online]. Available: http://info.ornl.gov/sites/publications/files/Pub44838.pdf

[17] N. Maruyama, A. Nukada, and S. Matsuoka, “A high-performance fault-tolerant software framework for memory on commodity GPUs,” in *IEEE International Symposium on Parallel & Distributed Processing*, 2010, pp. 1–12.
[18] University of Florida Sparse Matrix Collection.
http://www.cise.ufl.edu/research/sparse/matrices/.

[19] Y.Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[20] S. McIntosh-Smith, J. Price, R.B. Sessions, A.A. Ibarra, “High Performance in silico Virtual Drug Screening on Many-Core Processors”, IJHPCA, April 2014. DOI: 10.1177/1094342014528252
http://hpc.sagepub.com/content/29/2/119

[21] P. K. Lala, P. Thenappan, and M. T. Anwar, “Single error correcting and double error detecting coding scheme,” IET Electronics Letters, vol. 41, Issue 13, pp. 758–760, Feb. 2005.