Fault Tolerance Techniques for Sparse Matrix Methods

Simon McIntosh-Smith
Rob Hunt

An Intel Parallel Computing Center

Twitter: @simonmcs
Acknowledgements

• Funded by FP7 Exascale project: Mont Blanc 2

• Also supported by the Numerical Algorithms Group (NAG) and EPSRC

• My PhD student, Rob Hunt, did all the hard work
Prior work in Bristol

Performance portability across many-core architectures using OpenCL:

"High Performance in silico Virtual Drug Screening on Many-Core Processors", S. McIntosh-Smith, J. Price, R.B. Sessions, A.A. Ibarra, IJHPCA 2014
DOI: 10.1177/1094342014528252
CloverLeaf: Peta→Exascale hydrodynamics mini-app

- Developed in collaboration with AWE in the UK
- CloverLeaf is a bandwidth-limited, structured grid code and part of Sandia's "Mantevo" benchmarks.
- Solves the compressible Euler equations, which describe the conservation of energy, mass and momentum in a system.
- Optimised parallel versions exist in OpenMP, MPI, OpenCL, OpenACC, CUDA and Co-Array Fortran.
CloverLeaf sustained bandwidth

S.N. McIntosh-Smith, M. Boulton, D. Curran, & J.R. Price, “On the performance portability of structured grid codes on many-core computer architectures”, ISC, Leipzig, June 2014. DOI: 10.1007/978-3-319-07518-1_4
CloverLeaf (Peta)-scaling

Weak scaling performance of 960x960 instance of Cloverleaf CUDA on Titan

- Weak scaled across 16,000 GPUs on Oak Ridge's Titan
- Represented ~1.9 PetaBytes/s of memory bandwidth
Motivating application - TeaLeaf

- Will complement the Mantevo-CloverLeaf hydrodynamics mini-app
- Heat diffusion simulation
- 2D (3D coming)
- Implicit **sparse matrix** solver
- Written in FORTRAN, C, CUDA/OpenCL, OpenMP, MPI etc.
Fault tolerance – a crucial Exascale issue

• Identified as one of the top 10 technical challenges facing Exascale computing
  - Feb 2014 DoE Exascale report

• Many different kinds of "fault" can cause errors
  (G. Gibson, Proc. of the DSN2006, June, 2006):
  • Soft errors (bit flips in memory etc)
  • Hard errors (component breakage)
  • Power outages
  • OS errors
  • System software errors
  • Administrator error (human)
  • User error (human)
Research Status Anatomy

Checkpointing & Restart (C/R)
Diskless Checkpointing
Algorithm Based Fault Tolerance (ABFT)

Large
Overhead
Small
Application Specificity
Small
Large

Jack Dongarra, ISC, Leipzig, June 2014
ABFT: Application Based Fault Tolerance

• One of the main new techniques to enable FT Exascale applications without always resorting to naïve checkpoint/restart

• Potentially has great advantage over non-application based approaches:
  • Much lower overhead than checkpoint/restart
  • User knowledge enables wider range of fault recovery techniques
ABFT existing examples


- This approach was recently implemented by Dongarra and others in dense linear algebra libraries (ScaLAPACK etc)
• Before the factorization starts, a checksum is taken and Algorithm Based Fault Tolerance (ABFT) is used to carry the checksum along with the computation.
ABFT for sparse matrix computations

- Most of the matrix elements are zero
- Stored in a compressed format
- Which elements are zero may change over time

So we need a different approach for sparse matrices…
Sparse matrix compressed formats

- Sparse matrices are typically mostly 0
- E.g. in the University of Florida sparse matrix collection (~2,600 real, floating point examples), the median fill of non-zeros is just ~0.24%
- Therefore stored in a compressed format, such as COOrdinate format (COO) and Compressed Sparse Row (CSR)
**COO sparse matrix format**

- Conceptually think of each sparse matrix element as a 128-bit structure:
  - Two 32-bit unsigned coordinates \((x,y)\)
  - One 64-bit floating point data value
- **Observation 1:** *In a COO format sparse matrix, there is as much data in the indices as in the floating point values*
Protecting sparse matrix indices

- It turns out almost all sparse matrices store their elements in sorted order

- Observation 2: *We can exploit this ordering, along with the sparse matrix structure, to define a set of index relationships, or criteria, which can then be tested as elements are accessed*
Sparse matrix index criteria 1

For an $m \times n$ sparse matrix:

- $0 < x_k \leq m$
- $0 < y_k \leq n$

Does this help us?

- Largest matrix in UoFlorida set: $\sim 118M^2$
- Only uses bottom 27 bits of $(x,y)$
- Top 5 bits (at least) must always be 0 (15%)
- We have reduced the number of susceptible bits
Exploit the ordering of sparse matrix elements:

- \( x_{k-1} \leq x_k \leq x_{k+1} \)
- \( y_{k-1} < y_k \) when \( x_{k-1} = x_k \)
- where \( 1 < k < \text{NNZ} \)

Harder to evaluate how much these help us, as the answer depends on the distribution of the non-zeros in the matrix.
**Distributions of non zeros**

When non zeros are very spread out, potentially many bits of $y_k$ could be flipped while still satisfying the ordering constraint.

When non zeros are closer together, there are far fewer *susceptible* bits, i.e. bits of $y_k$ that can be flipped without the ordering constraint spotting the fault.
Non zero distributions

- Many real-world sparse matrices contain a lot of "clumping" of the non-zeros

"nasasrb"  "circuit5M"
Statistical analysis of the UoFlorida sparse matrix collection

- Analysed ~2,600 matrices in collection

- The scheme looks promising, protecting many elements completely, and most bits in most sparse matrices
Results from "nasasrb"

The number of protected bits as a proportion of all row index elements

All indices have at least 17 of their 32 bits protected

Nearly 70% of all indices fully protected
Results from "circuit5M"

The number of protected bits as a proportion of all row index elements

All indices have at least 9 of their 32 bits protected

About 45% of all indices fully protected
Exploiting index constraints

- Most constraints can be implemented with very simple integer operations
  - Arithmetic, bit shifts, comparisons
- These can be implemented in just a few instructions on most modern computer architectures
- Sparse matrix element accesses tend to cause cache misses
  - Opportunity to perform constraint checks in parallel with long latency DRAM accesses
Going beyond index constraint checking

**Advantages** of proposed approach:
- Fast to test, enables some correction
- Software implementation
- Catches majority of errors in many cases

**Disadvantages:**
- Doesn't catch all bit flip errors
- Only protects the indices, not the data
Software ECC protection of sparse matrix elements

- Remember that most sparse matrices only use 27 bits of their 32-bit indices
  - And most only use 24 bits

- **Observation 3:** *This leave 10-16 bits that could be "repurposed" for a software ECC scheme*

- A software ECC scheme could save considerable energy, performance and memory (all in region of 10-20%)
COO sparse matrix format

- Using 8 bits of the 128-bit compound element would allow a full single error correct, double error detect (SECDED) scheme in software.
- Use e.g. 4 unused bits from the top of each index:
  - Limits their size to "just" $0..2^{27}$ ($0..134$M)
- Can be used in conjunction with the index constraint checking approach for even greater protection.
Future work

• Have a stand-alone implementation which looks promising
• Overheads look low
• Want to implement this in a real library like PETSc
• Then want to test at scale in the presence of injected faults to measure real impact on performance
• Might be interesting to look at deliberately structuring the matrix to aid its resilience
Conclusions

- Fault tolerance / resilience is set to become a **first-order concern for Exascale**
- Application-based fault tolerance (**ABFT**) is one of the most promising techniques to address this issue
- ABFT can be applied at the **library-level** to help protect **large-scale sparse matrix operations**
Related Publications


