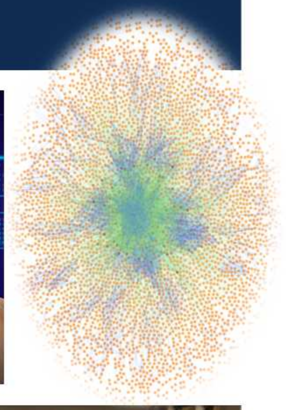


SAND2017-2085C

Programming Constructs for Transparent Silent-Error Mitigation in PDE Solvers

Maher Salloum, Jackson Mayo, and
Robert Armstrong

Sandia National Laboratories, Livermore, CA 94551
{mnsallo, jmayo, rob}@sandia.gov

February 27, 2017

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2017-XXXXC

Acknowledgment

- Karla Morris (Sandia National Laboratories) contributed to Fortran implementation

Problem: Future platforms will face tradeoffs imperiling correct hardware function

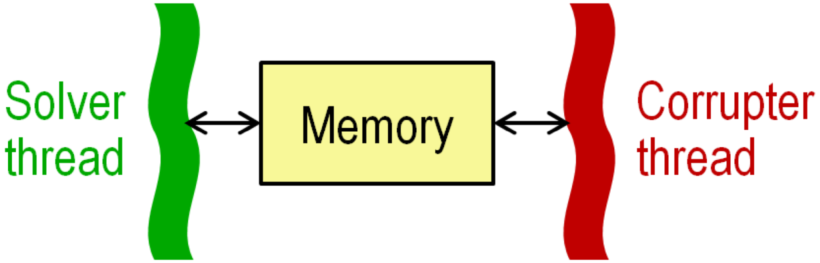
- Most HPC applications assume hardware that is reliable except for occasional *fail-stop* faults
  - For those faults, detection is simple, and generic recovery schemes are possible
- Hardware correction already attempts to hide many "out-of-nominal" behaviors from the application
  - Error correction for bit flips in DRAM and caches is important and largely effective
- Increasing scale and constrained power may push toward exposing new kinds of hardware errors – e.g., silent data corruption (SDC) that can cause wrong application results
  - Undetected DRAM errors at exascale for one type of ECC memory could be ~1 per day
  - Low-voltage processors and accelerators will likely have increased rates of arithmetic errors; ECC doesn't protect data transformation

Objective: Enable practical SDC mitigation targeted at physics simulation

- For Advanced Simulation & Computing (ASC) codes, we seek better understanding of how to anticipate, mitigate, and/or diagnose the effect of silent errors
- Ultimate goal is to contribute to a practical resilience toolbox for production codes
  - Leveraging existing PDE solvers and maintainable as they evolve
  - Adaptable to respond to future hardware characteristics
  - Flexible to unanticipated sources of silent errors
- There has been much study of SDC detection techniques, including for PDEs, but the recovery mechanism is crucial
- Our thesis: The dynamics of physical PDEs can support efficient ultralocal (within cache) detection and recovery, achieving stability to isolated occurrences of SDC
  - Handling silent errors quickly and transparently (like standard numerical errors) reduces the cost of a false positive

Current error model describes memory bit flips

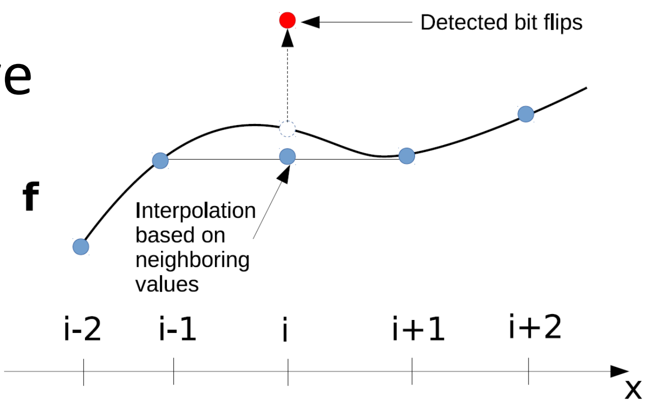
- In extreme-scale scientific computing, floating-point data are an obvious concern for SDC
  - Floating-point data often constitute the bulk of memory usage
  - Corruption in other places (control logic, pointers) is more likely to cause outright crashes, which will be mitigated by other means
- Our error-injection framework for solvers: Asynchronously perform raw memory bit flips in the solution array



  - Corrupter injects random bit flips based on a probability parameter
- Memory corruption is also a proxy for other silent-error sources that ultimately affect values in memory – e.g., processor arithmetic errors

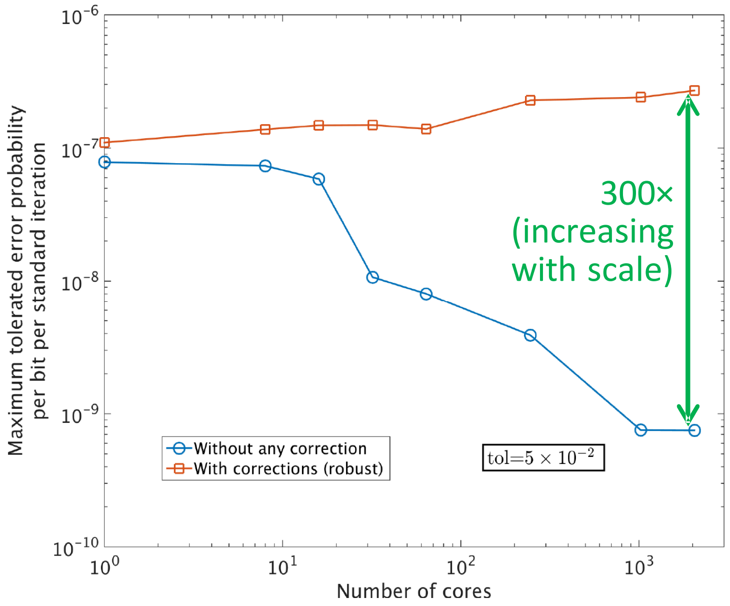
Outlier detection and interpolation can leverage smoothness properties of physics

- We have prototyped a simple, widely applicable SDC mitigation technique for PDE solvers (FTXS'16)
- Current scope: SDC in memory affecting floating-point data in structured meshes
- Aim is to correct accumulated bit flips in data values when they are loaded from memory, just before they are used – so that large corruptions will not propagate
- Corrupted values are detected via relative deviation from neighbors and replaced with an interpolation, as a computation loop is sweeping the array
  - One-sided correction at array boundaries
  - Detection/interpolation along the "fast" index in multidimensional arrays for cache efficiency



Interpolation is effective in tolerating bit flips

- Initially demonstrated on simple solvers, both explicit (1D Burgers equation) and iterative (HPCCG conjugate gradient mini-app for 3D elliptic equation)
  - Runtime overhead as low as a few percent in the presence of local source-term computations or intensive communication
- HPCCG example (FTXS'16)
  - Modify CG to use interpolation-based robust linear algebra "building blocks"
  - Robust CG method can tolerate higher SDC rates that prevent standard method from converging
  - Our approach is helpful for systems with SDC rates *between the blue and red curves* – a potentially very wide range of scenarios for co-design or unexpected faults

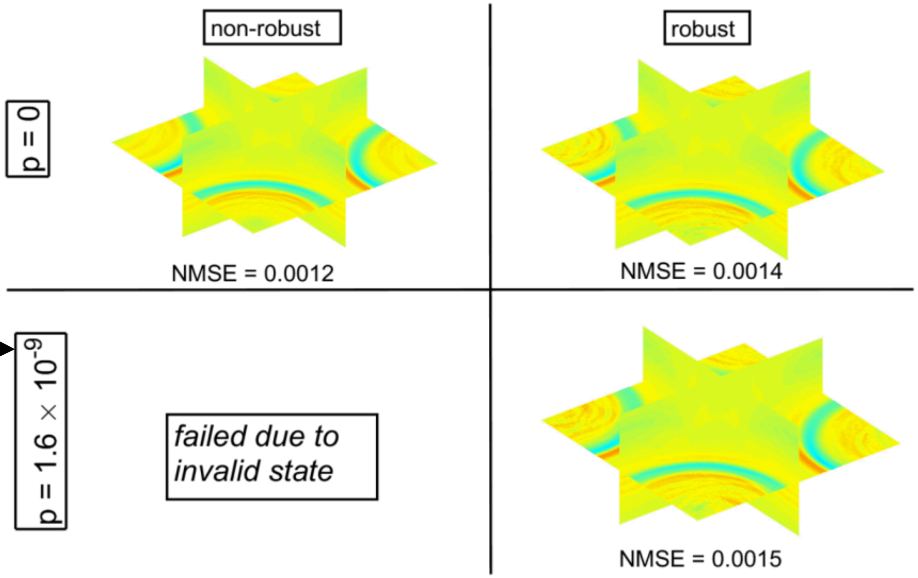


"Manual" SDC mitigation demonstrated in multiphysics solver

- SMC is a convenient example of production-like code
  - DOE coupled fluid dynamics and combustion solver
  - Representative of structured multiphysics simulation
- Linear algebra and stencils involve nested loops over spatial directions and field variables
  - Code modifications for robustness occur at the level of the inner loops

z-velocity field plots:

- SMC simulation with 9 species and 27 reactions running on 4 cores
- Robust solver tolerates 4.5× the error rate tolerated by the standard solver
- 20% runtime overhead
- 3% additional lines of code



Code maintainability is a key issue

- How easy is it to incorporate SDC mitigation in the code for low-level solver operations?
- For operations that are widely reused and rarely modified, robust versions can be written once and packaged in libraries; but robust custom solver operations (e.g., stencil code) may need to be written/modified by application experts rather than resilience experts
  - Even for packaged operations, we would like to ease the library writer's task
- Aim: Minimize additional programmer effort needed to implement and maintain a robust solver vs. a non-robust one

Linear algebra building blocks facilitate robustness in iterative solvers

- HPCCG top-level code incorporates mitigation by simply using overloaded BLAS functions with additional argument
  - The robust BLAS must implement error detection and correction

Non-robust pseudocode

```
std::vector<double> x(N), p(N);
std::vector<double> r(N), q(N);
HPC_Sparse_Matrix A;

// Loop until convergence
{
   HPC_sparsemv(A, p, q);
   ddot(p, q, &alpha);
   alpha = R / alpha;
   waxpby(1.0, x, alpha, p, x);
   waxpby(1.0, r, -alpha, q, r);
   ddot(r, r, &beta);
   beta = beta / R;
}
```

Robust pseudocode

```
std::vector<double> x(N), p(N);
std::vector<double> r(N), q(N);
HPC_Sparse_Matrix A;

// Error detection threshold
double t = 100.0;

// Loop until convergence
{
   HPC_sparsemv(A, p, q, t);
   ddot(p, q, &alpha, t);
   alpha = R / alpha;
   waxpby(1.0, x, alpha, p, x, t);
   waxpby(1.0, r, -alpha, q, r, t);
   ddot(r, r, &beta, t);
   beta = beta / R;
}
```

“Resilient view” class helps implement interpolation-based mitigation transparently

- In C++, preserves array syntax via overloaded operator[]
  - Simplified code shown, for error detection and correction when reading from the array
  - Templating allows use on any existing random-access array type
  - Extension: operator[] can return object with overloaded assignment operator so resilient view can be used on left-hand side too

```
class ResilientView1D {
public:
  double operator[](int index);
private:
  double* begin;
  int size;
  double threshold;
};
```

```
double ResilientView1D::operator[](int index) {
  const double* it = begin + index;
  double val = it[0];
  double diff = it[1] - it[-1];
  double interp = 0.5 * (it[1] + it[-1]);
  if(val != val ||
     std::fabs((val - interp) / diff) > threshold) {
    val = interp;
  }
  return val;
}
```

Resilient view further improves code maintainability for SDC mitigation

- Example: BLAS-like operation
- Original implementation: additional function calls
  - Must know and remember to insert in each loop for each array used
- New implementation: resilient views
  - Simply require that all array reads occur through view objects
  - Loops look very similar to the original code

Non-robust pseudocode

```
std::vector<double> x(N), y(N);

for (int i=0; i<N; i++)
    y[i] = a*x[i] + b*y[i];
```

Robust pseudocode *using additional function calls*

```
std::vector<double> x(N), y(N);

for (int i=0; i<N; i++) {

      detect_interp(x,i,t);
      detect_interp(y,i,t);

      y[i] = a*x[i] + b*y[i];
}
```

Robust pseudocode *using resilient views*

```
std::vector<double> x(N), y(N);

// RView is the resilient view
RView rx(x, t), ry(y, t);

for (int i=0; i<N; i++)
    y[i] = a*rx[i] + b*ry[i];
```

Resilient view also facilitates custom stencil operations

Non-robust pseudocode

```
std::vector<double> f(N), fn(N);

for (int i=1; i<N-1; i++)
    fn[i]=a*f[i-1]+b*f[i]+c*f[i+1];
```

Robust pseudocode
using additional function calls

```
std::vector<double> f(N), fn(N);

for (int i=1; i<N-1; i++) {

    detect_interp(f,i-1,t);
    detect_interp(f,i,t);
    detect_interp(f,i+1,t);

    fn[i]=a*f[i-1]+b*f[i]+c*f[i+1];
}
```

Non-robust pseudocode

```
std::vector<double> f(N), fn(N);
```

```
for (int i=1; i<N-1; i++)
    fn[i]=a*f[i-1]+b*f[i]+c*f[i+1];
```

Robust pseudocode
using resilient views

```
std::vector<double> f(N), fn(N);

RView rf(f, t);
```

```
for (int i=1; i<N-1; i++)
    fn[i]=a*rf[i-1]+b*rf[i]+c*rf[i+1];
```

Fortran implementation of resilient view is similar

- Requires type-bound procedure “%f” because array indexing syntax cannot be overloaded

```
type :: resilientView4D
    private
        double precision, pointer :: begin(:,:,:,:)=>NULL()
        double precision :: threshold
    contains
    procedure :: f
end type

contains
double precision function f(this,i1,i2,i3,i4) result(val)
    class(resilientView4D), intent(in) :: this
    integer, intent(in) :: i1,i2,i3,i4
    double precision :: diff, interp
    val = this%begin(i1,i2,i3,i4)
    diff = this%begin(i1+1,i2,i3,i4) - this%begin(i1-1,i2,i3,i4);
    interp = 0.5d0 * (this%begin(i1+1,i2,i3,i4) + this%begin(i1-1,i2,i3,i4))
    if( (val /= val) .or. (dabs((val - interp) / diff) > this%threshold) ) then
        val = interp
    endif
end function
```

Our previous mitigation in SMC required inserting function calls in each loop

Non-robust pseudocode

```
// Runge-Kutta explicit time-stepping
do m = 1, nc
   do k = lo(3),hi(3)
      do j = lo(2),hi(2)
         do i = lo(1),hi(1)
            u1p(i,j,k,m) = a*u1p(i,j,k,m) + b*u2p(i,j,k,m) + c*upp(i,j,k,m)
         end do
      end do
   end do
end do
```

Robust pseudocode *using additional function calls*

```
// Runge-Kutta explicit time-stepping
do m = 1, nc
   do k = lo(3),hi(3)
      do j = lo(2),hi(2)
         do i = lo(1),hi(1)
        // Error detection and correction steps
            u1p(i,j,k,m) = detect_interp(u1p,i,j,k,m,t)
            u2p(i,j,k,m) = detect_interp(u2p,i,j,k,m,t)
            upp(i,j,k,m) = detect_interp(upp,i,j,k,m,t)

            u1p(i,j,k,m) = a*u1p(i,j,k,m) + b*u2p(i,j,k,m) + c*upp(i,j,k,m)
         end do
      end do
   end do
end do
```

← Tests shown above used this version

Resilient view being incorporated into SMC

Sandia National Laboratories

Non-robust pseudocode

```
// Runge-Kutta explicit time-stepping
do m = 1, nc
   do k = lo(3),hi(3)
      do j = lo(2),hi(2)
         do i = lo(1),hi(1)
            u1p(i,j,k,m) = a*u1p(i,j,k,m) + b*u2p(i,j,k,m) + c*upp(i,j,k,m)
         end do
      end do
   end do
end do
```

Robust pseudocode *using resilient views*

```
RView u1pr(u1p, t)
RView u2pr(u2p, t)
RView uppr(upp, t)

// Runge-Kutta explicit time-stepping
do m = 1, nc
   do k = lo(3),hi(3)
      do j = lo(2),hi(2)
         do i = lo(1),hi(1)
            u1p(i,j,k,m) = a*u1pr%f(i,j,k,m) + b*u2pr%f(i,j,k,m) + c*uppr%f(i,j,k,m)
         end do
      end do
   end do
end do
```

Conclusion: Helping take SDC mitigation a step closer to routine use

- Resilient view can detect and correct SDC in structured PDE solvers while keeping code understandable & maintainable
  - Using mitigation technique previously found effective and efficient
  - Reducing chance for programmer to mistype or omit a mitigation step
- Future directions can bring this work further into practice
  - Evaluation at larger computational scale
    - We are moving toward advanced technology platform (100,000s of cores)
  - Broader error models
    - While memory error mitigation *can* address other silent error sources, more efficient targeted techniques are possible
  - Long-term potential to inform hardware choices (co-design)
    - Showing we can practically tolerate more errors could encourage vendors to “break the logjam” and increase their offerings of more efficient, less reliable hardware