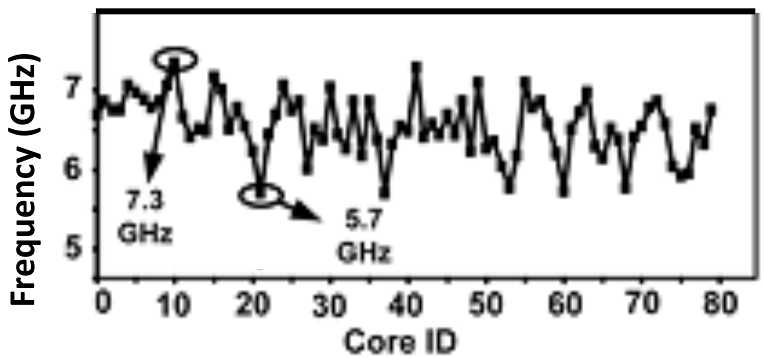
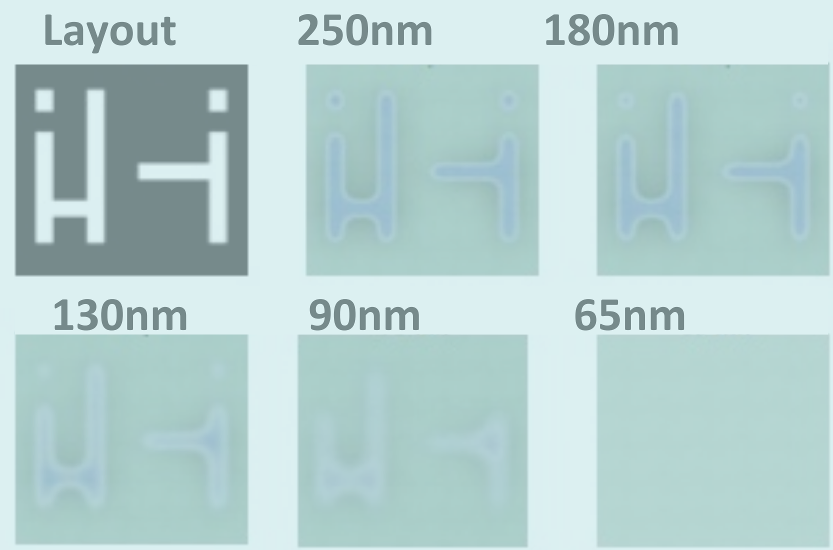
Automated Algorithmic Error Resilience for Structured Grid Problems based on Outlier Detection

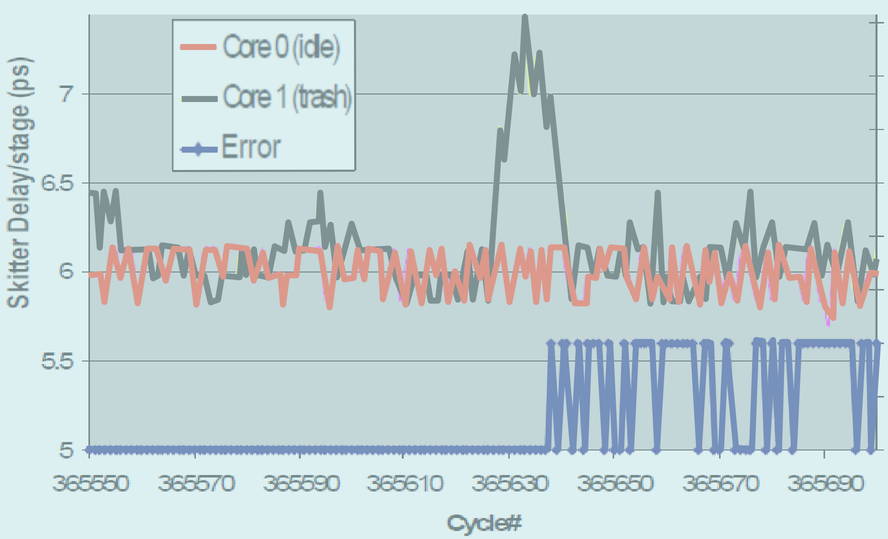
Amoghavarsha Suresh and **John Sartori**

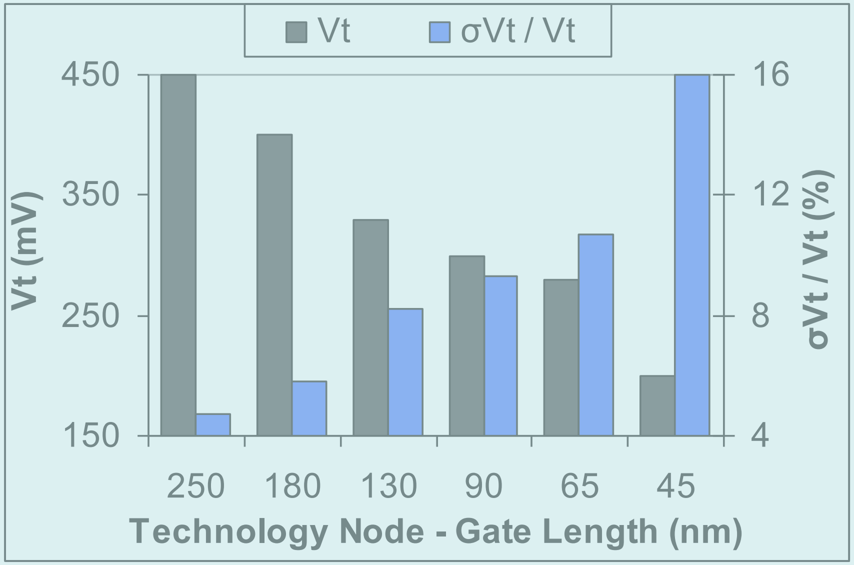
University of Minnesota

Non-determinism is (*getting more*) expensive

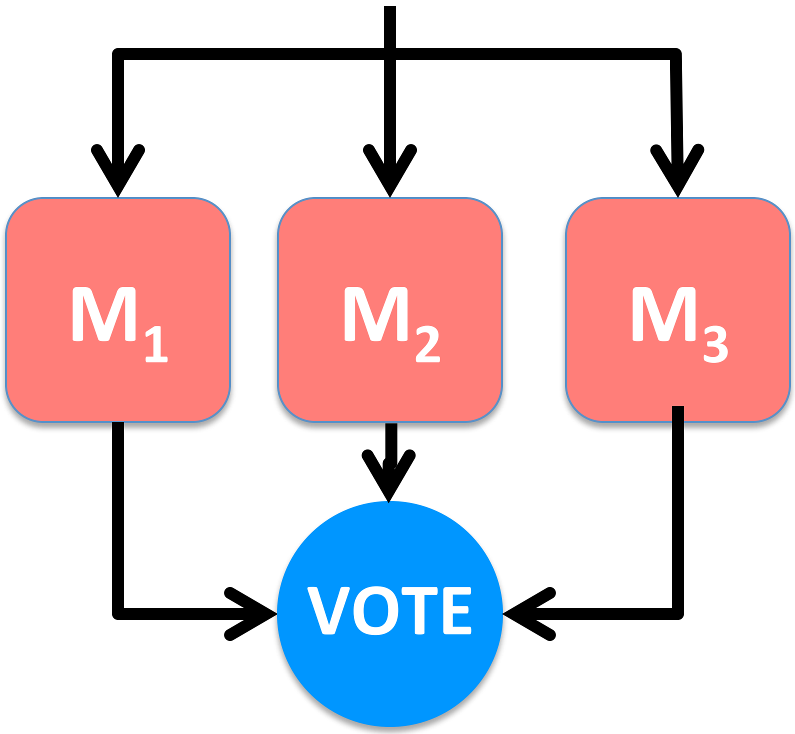




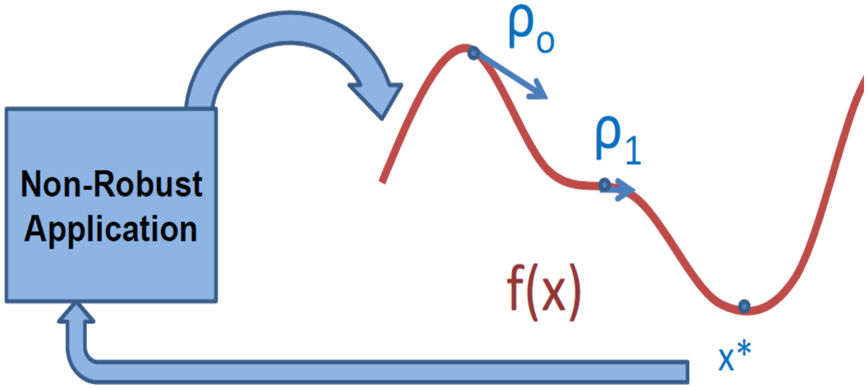




Error resilience as a solution to variability



Triple Modular Redundancy



Application Robustification

We want error resilience that addresses the drawbacks of previous error resilience approaches

Error resilience requirements

- Low overhead
  - Applicable to many applications
    - Easy to implement
      - Automated

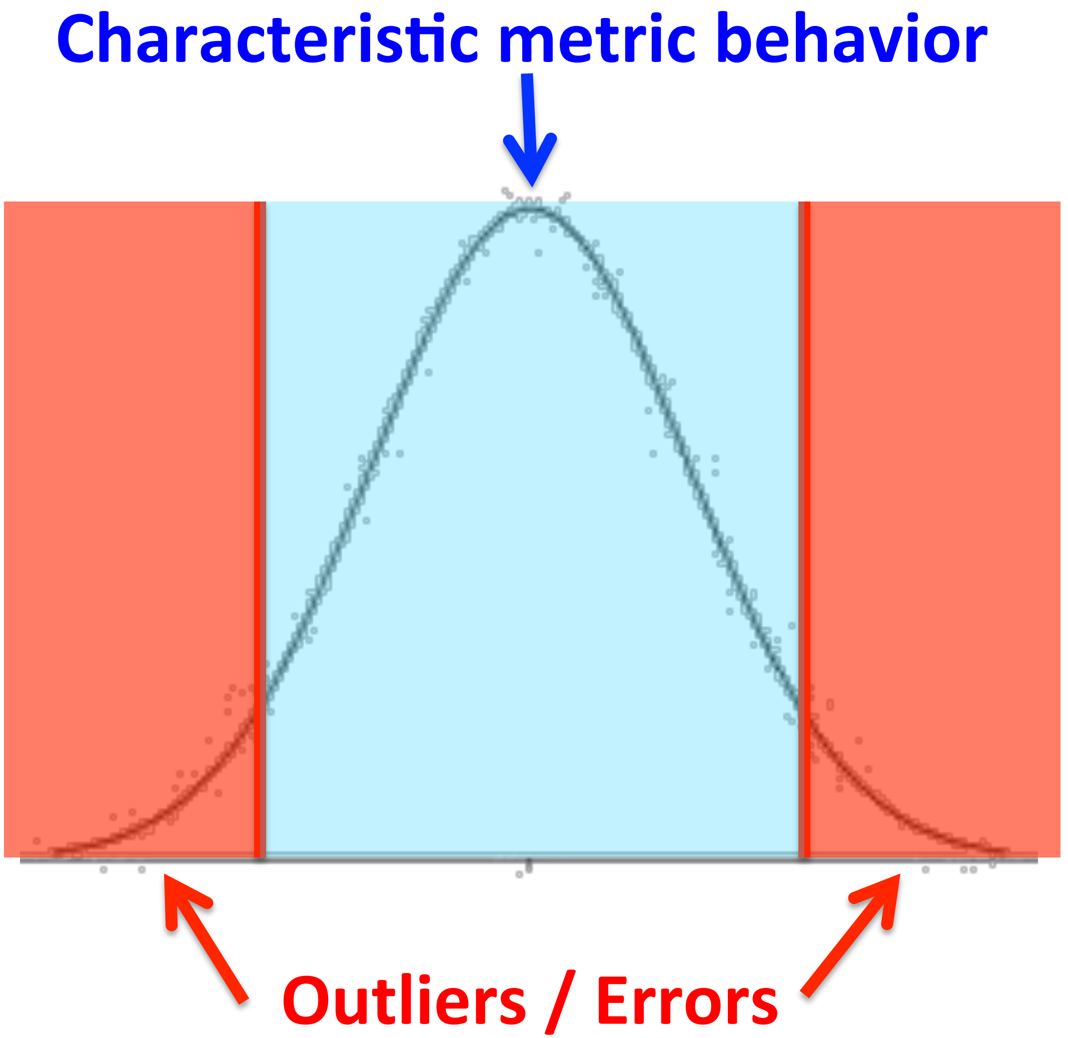
Our solution:

Automated

Algorithmic Error Resilience based on

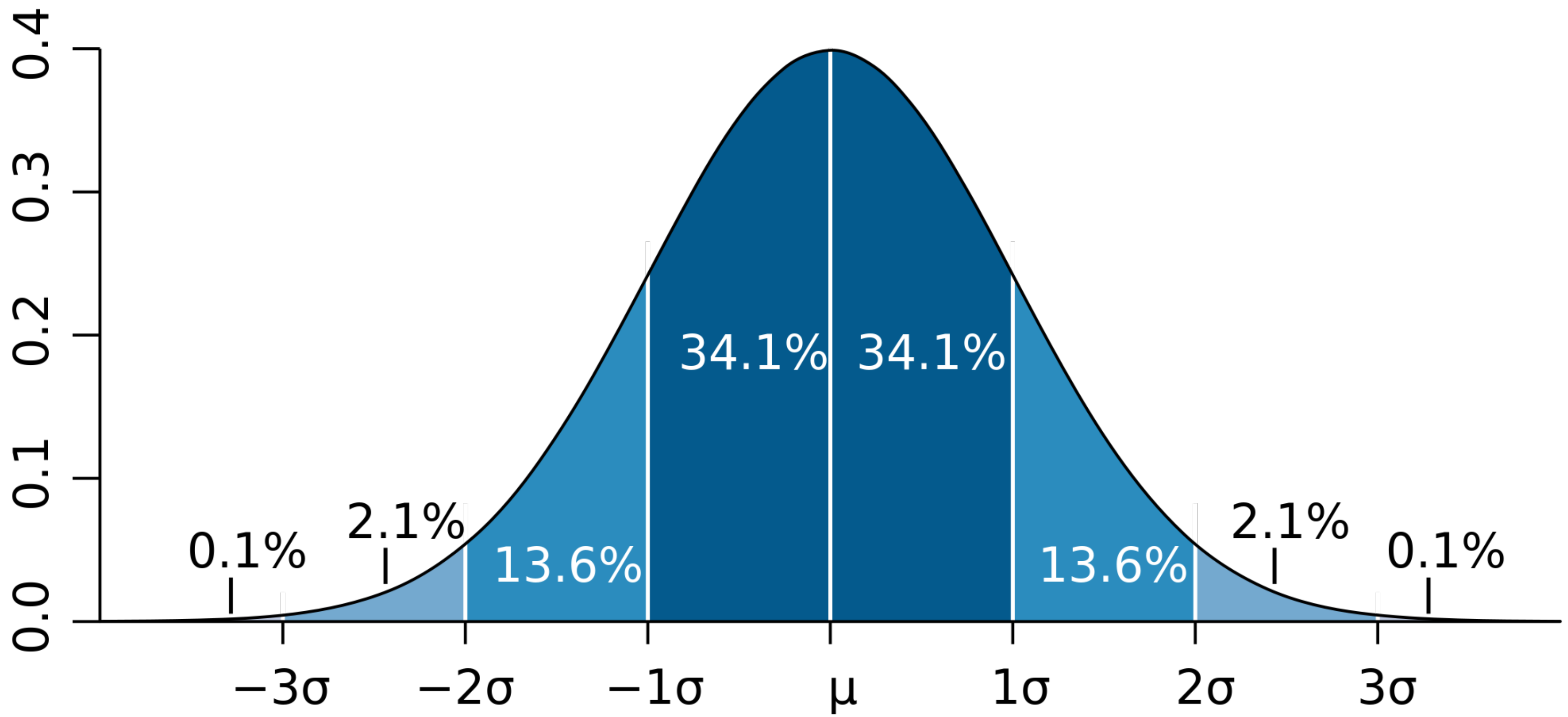
Outlier Detection

Outlier detection-based error resilience

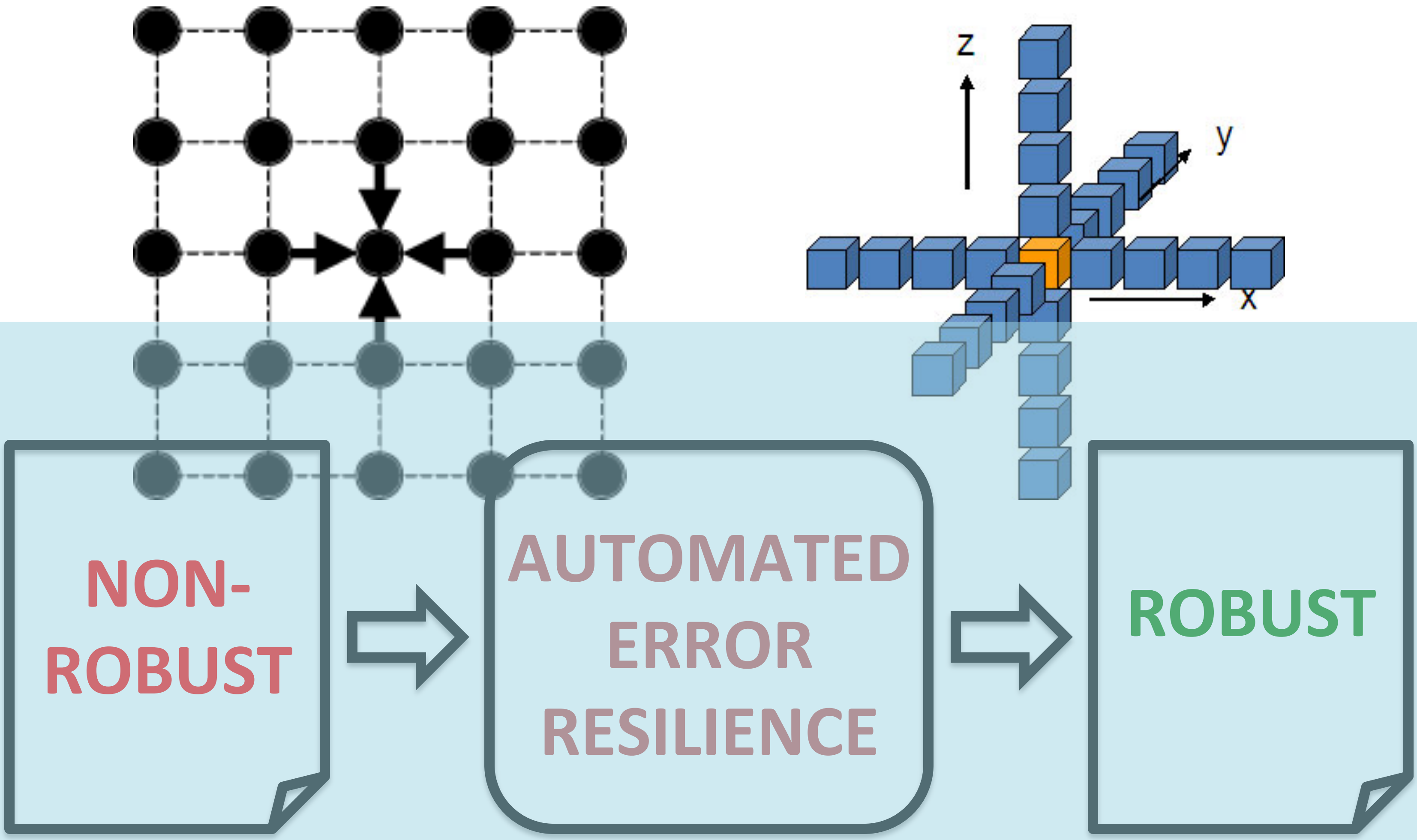


Outlier detection can be used to perform error detection

Leveraging statistically-rigorous techniques

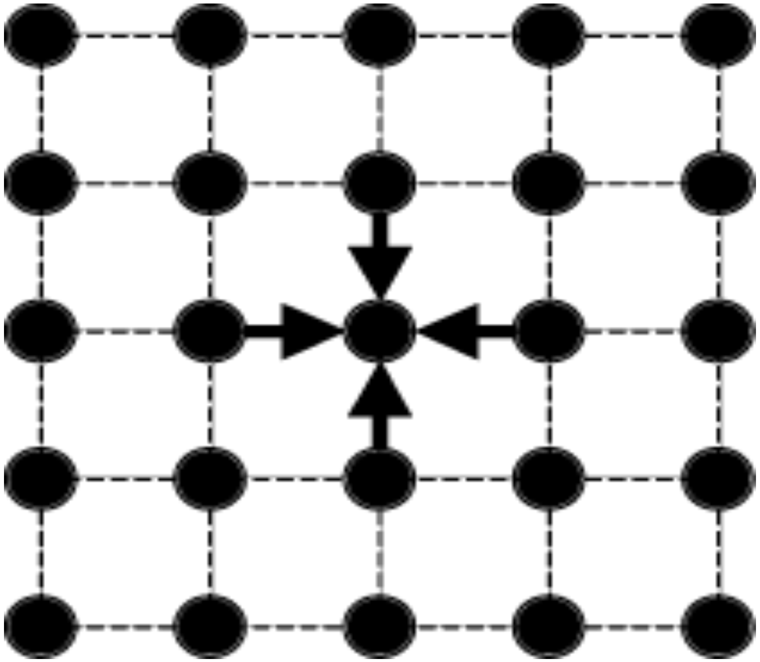


99.7% of data within three sigma


z
y
x
NON-ROBUST
AUTOMATED ERROR RESILIENCE
ROBUST

Structured Grid (SG) Problems

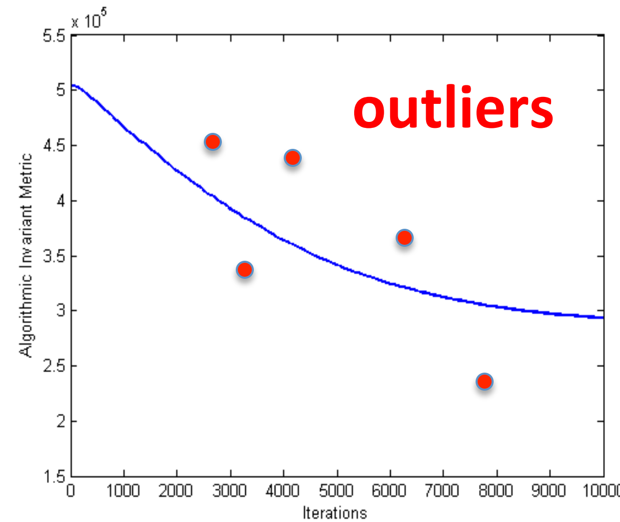
- Non-iterative SG
  - Convolution, stencil computation
  - e.g., Image and Video Processing
- Iterative SG
  - PDE Solvers
  - e.g., Heat transfer, Wave propagation

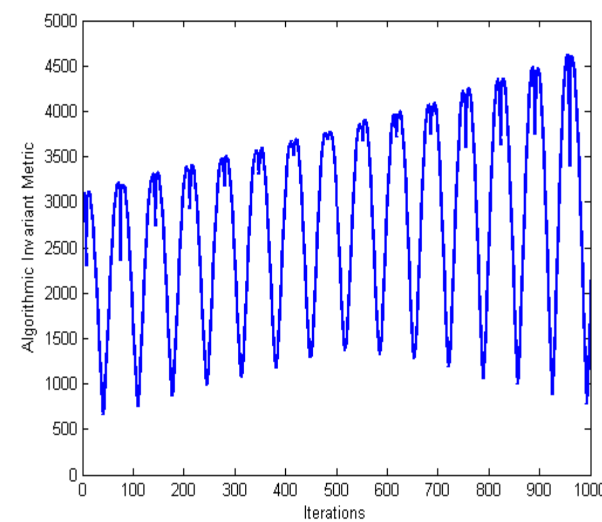


Updates to grid depend on neighbors → errors can propagate

Algorithmic Invariants: Iterative Applications

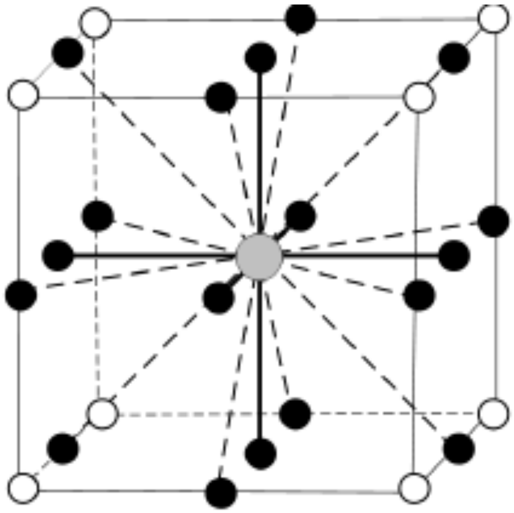
- **Time independent**
  - Poisson equations
  - Laplacian equations
  - Characteristic convergence rate
- **Time dependent**
  - Heat dissipation
  - Wave equation
  - Characteristic frequency



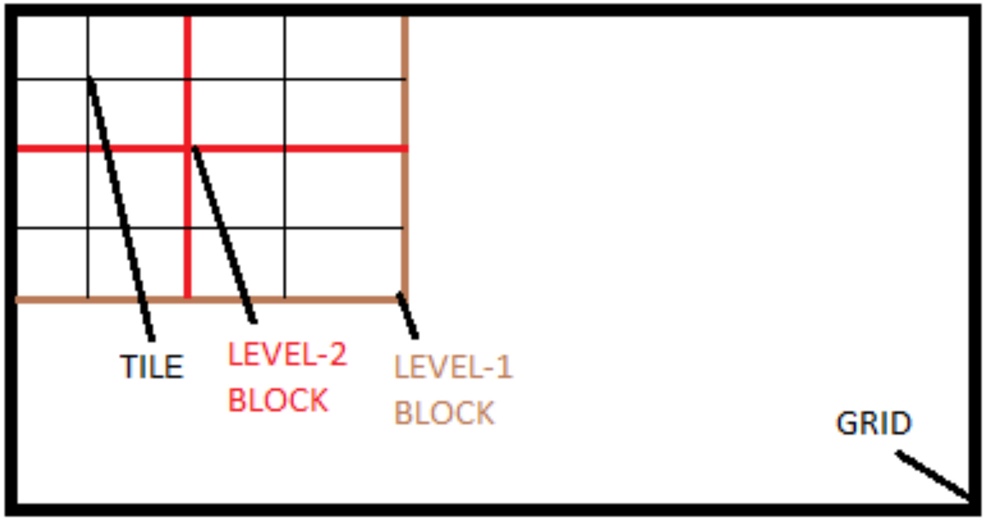


Approach: Check invariant at points of the grid for outliers

Algorithmic Error Resilience: Iterative Apps



Grid can be huge (e.g., 10^9 points)

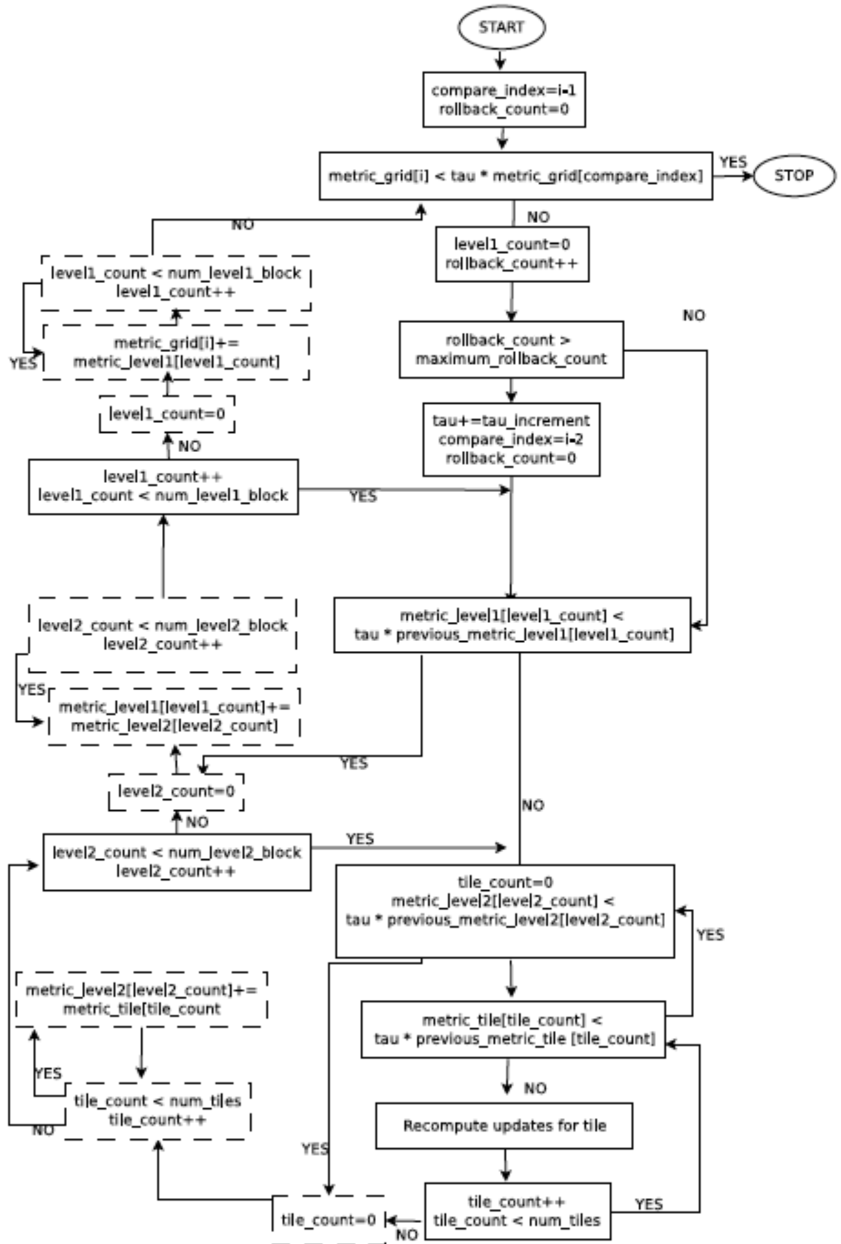


Grid decomposition reduces overhead of error resilience

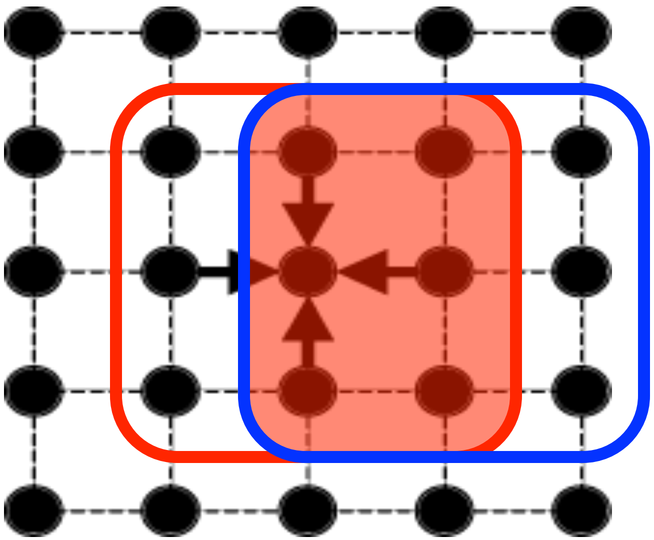
Error detection at coarse granularity.

Error recovery recover at fine granularity.

Detection, localization, and recovery



Algorithmic Invariants: Non-iterative Apps



- $Convution[i][j] = \sum_{k=0}^{2n} \sum_{l=0}^{2m} Kernel[k][l] * Grid[i-n+k][j-m+l]$

Significant data reuse in both the grid and the kernel

Defining bounds for outlier detection

$$
\begin{array}{l}
\texttt{for}(i = 0, \;\; i < 2m) \\
\quad quotient \;\; = \;\; (int) \;\; kernel[0][i] \;\; / \;\; kernel[0][i+1] \\
\quad max_{remainder} = 0 \\
\quad \texttt{for}(j = 0, j < (2n+1)) \\
\qquad remainder = (((int)kernel[j][i]/kernel[j][i+1]) \\
\qquad\qquad\qquad\qquad -quotient) \\
\qquad \texttt{if}(remainder \;\; > \;\; max_{remainder}) \\
\qquad\quad max_{remainder} \;\; = \;\; remainder \\
\quad C_{max}[i] \;\; = \;\; quotient \;\; + \;\; max_{remainder} \\
\quad C_{min}[i] \;\; = \;\; quotient \;\; - \;\; max_{remainder}
\end{array}
$$

Example: Canny Edge Detection

$$\text{kernel} = 1/159 * \begin{bmatrix} 2 & 5 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 5 & 5 & 4 & 2 \end{bmatrix}$$

$$\text{kernel}[\,][3] = [4 \quad 9 \quad 12 \quad 9 \quad 4]^T$$
$$\text{kernel}[\,][2] = [5 \quad 12 \quad 15 \quad 12 \quad 5]^T$$

Characteristic behavior of kernel used to define expected data distribution for outlier detection.

Example: Canny Edge Detection

- Represent one column in terms of another.

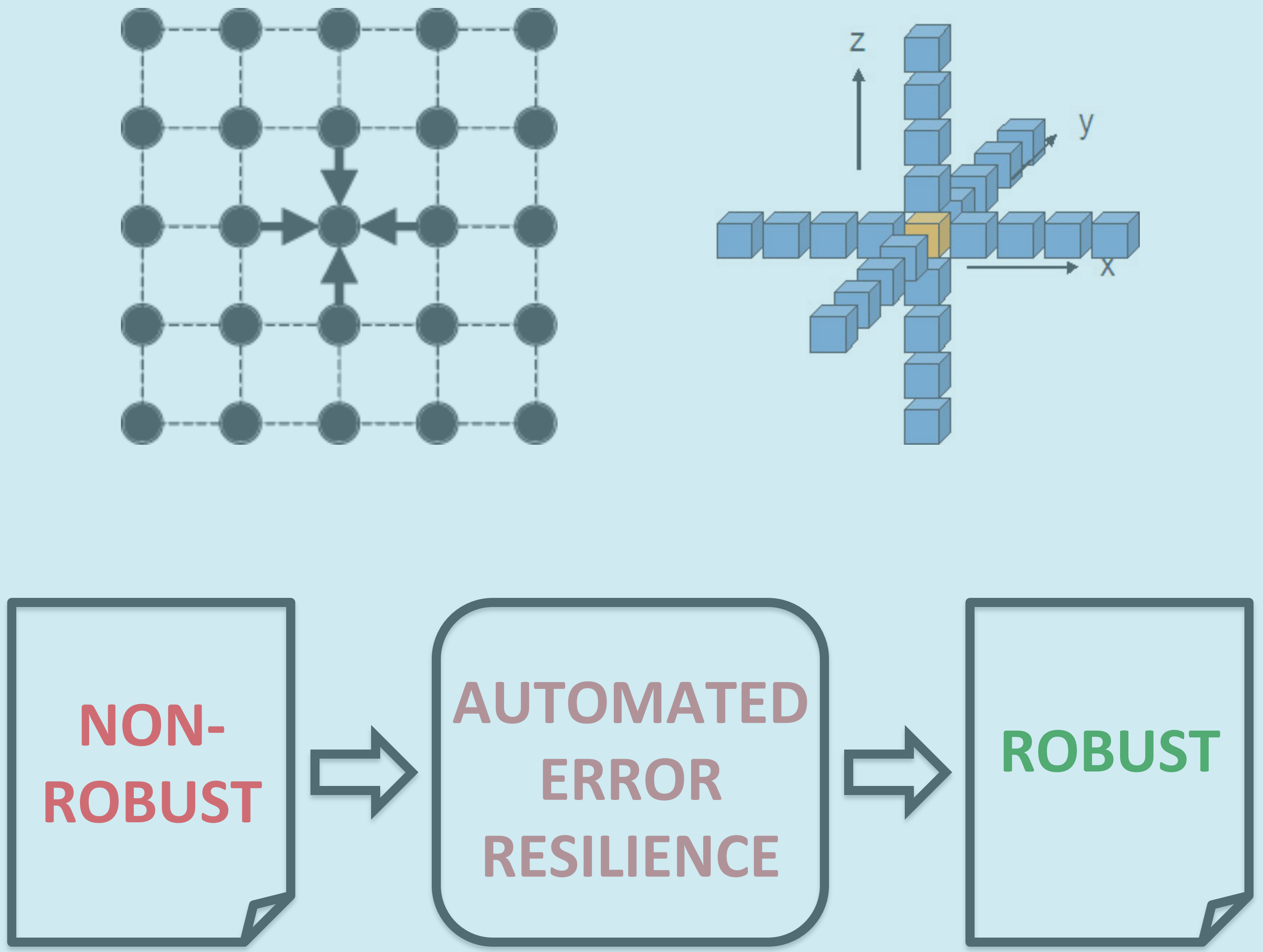
$$\text{kernel[][2]= 1*kernel[][3] +} \begin{bmatrix} \text{0.25 * kernel[0][3]} \\ \text{0.33 * kernel[1][3]} \\ \text{0.25 * kernel[2][3]} \\ \text{0.33 * kernel[3][3]} \\ \text{0.25 * kernel[4][3]} \end{bmatrix}$$

- Bounds on computed results define expected data distribution

$$C_{max}[2] = 1 + 0.33,\ C_{min}[2] = 1 - 0.33$$

$$\text{min(abs(grid[][k] . kernel[][2])) = abs(}C_{min}\text{ * grid[][k] . kernel[][3])}$$

$$\text{max(abs(grid[][k] . kernel[][2])) = abs(}C_{max}\text{ * grid[][k] . kernel[][3])}$$


z
y
x
NON-ROBUST
AUTOMATED ERROR RESILIENCE
ROBUST

Automatic Program Transformation

Previous work on Application Robustification

- *If possible*, reformulate app as stochastic optimization
- Requires expertise in stochastic optimization
- Transformation performed manually and individually
- Different procedures for different applications

Automated Algorithmic Error Resilience

- Minimal programmer intervention required
- Minimal programmer expertise required
- Transformation is automated by tool
- Approach applicable for all applications in class

Mark variables in non-robust version

```
#define ROWS 1000000
#define COLS 1000000
#define STENCIL_LENGTH 1
#define PADDED_ROWS ROWS+2*STENCIL_LENGTH
#define PADDED_COLS COLS+2*STENCIL_LENGTH
#define DATA_LAYOUT 1

// Variable declarations
int m = ROWS;
int n = COLS;
int sl = STENCIL_LENGTH;
int dl = DATA_LAYOUT;
float heat_mat1[PADDED_ROWS*PADDED_COLS];
float heat_mat2[PADDED_ROWS*PADDED_COLS];
```

- **Grid Size**
- **Grid Layout**
- **Current and Previous**
- **End of grid update code**

```
// Programmer-inserted preprocessor directives
#pragma struct_grid size m n
#pragma struct_grid curr_array heat_mat1
#pragma struct_grid prev_array heat_mat2
#pragma struct_grid stencil_length sl
#pragma struct_grid data_layout dl
#pragma struct_grid iterator t
for (int t = 0; t < num_iters; t++)
{
   ...
   // Beginning of grid update code
   for(int row_idx = STENCIL_LENGTH;
   row_idx < (PADDED_ROWS - STENCIL_LENGTH);
   row_idx++){
      for(int col_idx = STENCIL_LENGTH;
      col_idx < (PADDED_COLS - STENCIL_LENGTH);
      col_idx++){
         int index = row_idx*PADDED_COLS + col_idx;
         #pragma struct_grid equation
 heat_mat1[index] =
         0.125*(heat_mat2[index+PADDED_COLS]
         - 2.0*heat_mat2[index]
         + heat_mat2[index-PADDED_COLS])
         + 0.125*(heat_mat2[index+1]
         - 2.0*heat_mat2[index]
         + heat_mat2[index-1])+heat_mat2[index];
      }
   }
   // End of grid update code
   #pragma struct_grid end_grid_update
}
```

Transformed robust version

```
#include "RobustSG.h"
...
for (int t = 0; t < num_iters; t++)
{
    ...
    // Beginning of grid update code
    for (int row_idx = STENCIL_LENGTH;
    row_idx < (PADDED_ROWS - STENCIL_LENGTH);
    row_idx++){
        for(int col_idx = STENCIL_LENGTH;
        col_idx < (PADDED_COLS-STENCIL_LENGTH);
        col_idx++){
            // ...grid update equation...
        }
    }
    // End of grid update code
    // Error detection and correction function
    grid_outlier_based_resilience(heat_mat1,
                                  heat_mat2,
                                  m, n, sl, dl, t);
}
```

Any application in the structured grid class can be automatically robustified by our tool

Methodology

- **Fault model:** Models numerical effects of faults
  - Symmetric (bimodal, unimodal)
  - Memory (bitflip)
  - Non-symmetric (one-sided, trimodal)
- **Fault injection**: Binary instrumentation via Pin
- **Applications**
  - Iterative (Poisson, Laplacian, Heat dissipation, Wave propagation)
  - Non-iterative (Canny edge detection, Gaussian filter)

Methodology: Metrics

Performance

$$O_{FLOPs} = FLOPs_{error\_injected\_run} / FLOPs_{pristine\_run}$$

$$O_{iterations} = O_{FLOPs} \cdot \frac{N_{iterations\_error\_injected}}{N_{itertions\_pristine}}$$

Output Quality

Average deviation =

$$(1/N) \sum_{i=1}^{N} |X_i - X_i'| / X_i$$

Results: Non-iterative applications

- Average overhead of error resilience: 22%
- Output quality: 2x improvement over no error resilience

Metric	Error Resilience?	64x64	128x128	256x256
AD	No	0.00468	0.00461	0.00451
	Yes	0.00199	0.00221	0.00206
O_{FLOPs}	No	1.245	1.350	1.345
	Yes	1.187	1.191	1.303

* Results shown for most challenging fault models (bitflip, unimodal)
→ Our technique has no output quality degradation for other fault models

Overhead is significantly lower than traditional HW and SW error resilience techniques.

Results: Non-iterative applications

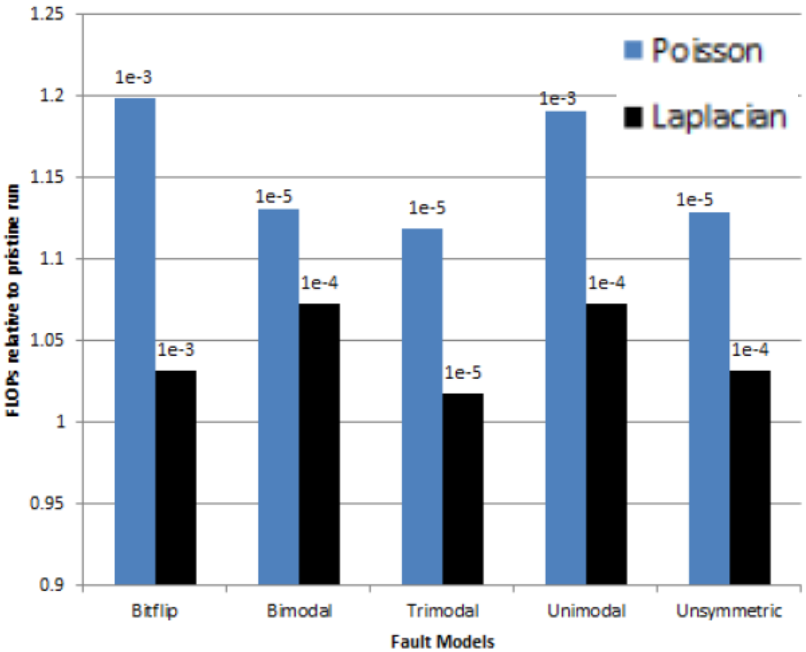
- Average overhead of error resilience: 22%
- Output quality: 2x improvement over no error resilience

Metric	Error Resilience?	64x64	128x128	256x256
Avg Dev	No	0.00468	0.00461	0.00451
	Yes	0.00199	0.00221	0.00206
O_{FLOPs}	Yes	1.187	1.191	1.303

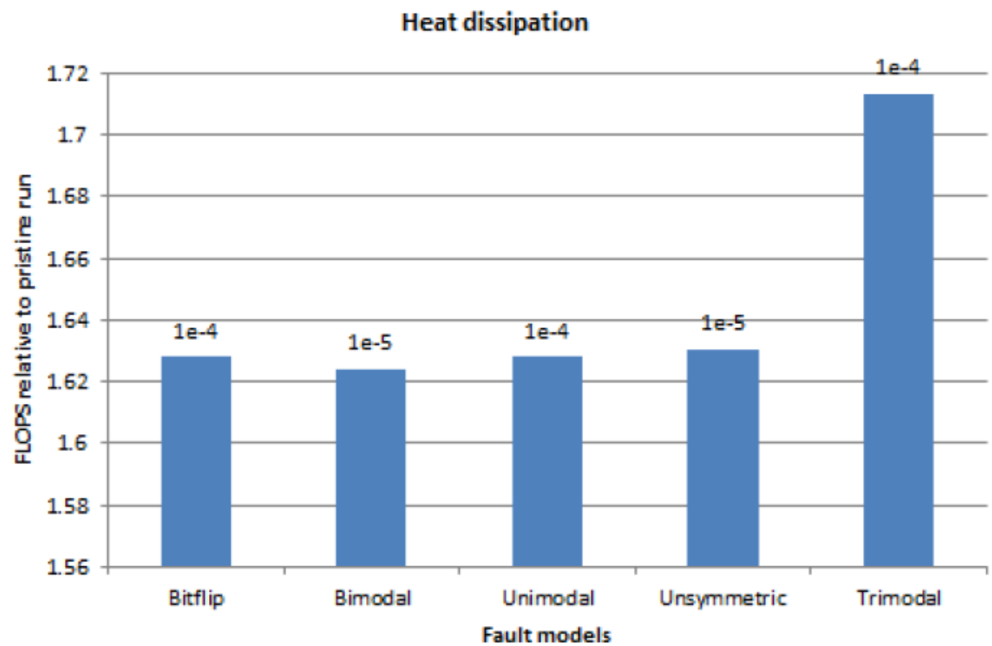
* Results shown for most challenging fault models (bitflip, unimodal)
→ Our technique has no output quality degradation for other fault models

Overhead is significantly lower than traditional HW and SW error resilience techniques.

Results: Iterative applications



Time-independent



Time-dependent

Average overhead = 4% - 15%

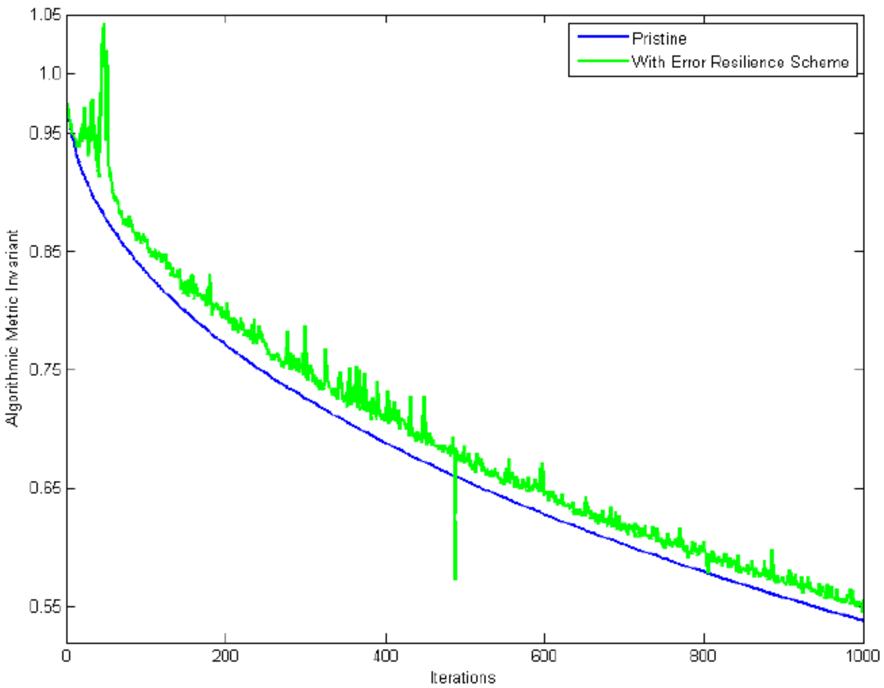
Average overhead = 64%

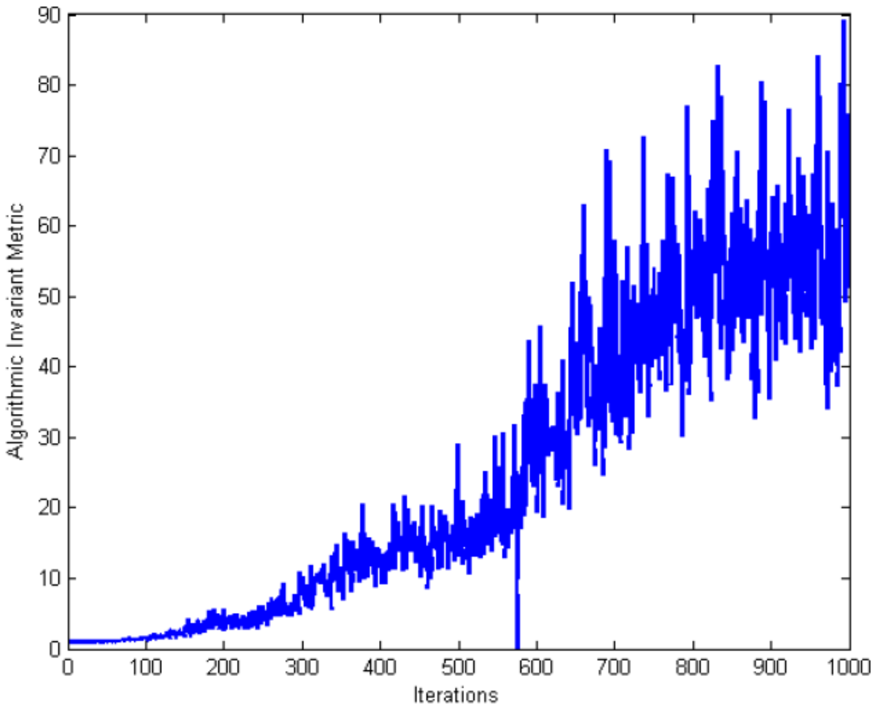
Overhead is lower for lower fault rates

Overhead vs Grid Decomposition

(Grid-size,#$L1$,#$L2$,L2-Size)	Fault Rate	O_{iter}
(320x320,1,25,64x64)	$1E-3$	1.83787
(384x384,1,36,64x64)	$1E-3$	2.00794
(384x384,1,16,96x96)	$1E-3$	1.53335
(480x480,1,25,96x96)	$1E-3$	1.96161
(576x576,3,9,64x64)	$1E-3$	1.48059
(576x576,3,9,64x64)	$1E-4$	1.24327
(576x576,2,9,96x96)	$1E-4$	1.23814
(768x768,4,9,64x64)	$1E-4$	1.27950
(768x768,3,9,96x96)	$1E-4$	1.27810

Error Resilience Required





With
Error Resilience

Without
Error Resilience

We achieve same output quality as pristine application

Lessons Learned and Ongoing Directions

- **Lessons Learned**
  - Exploit an algorithm's native features
  - Variables that characterize majority of data are good candidates for metric creation
  - Error resilience should be automated
- **Ongoing research**
  - Utilize dynamic invariant detection tools
  - Robust libraries
  - Automate existing application robustification
  - Target broader range of algorithms

Conclusion

- High-performance and energy-efficient computing systems are error-prone and energy-constrained
  - May require error resilience to ensure productivity
- Outlier-based error resilience is effective and efficient
- Automated error resilience facilitates application robustification for large classes of applications with minimal programmer intervention
- Low overhead (4% – 64%, on average, for same output quality as error-free application)
- 2x – 3x improvement in output quality over non-robust version of application