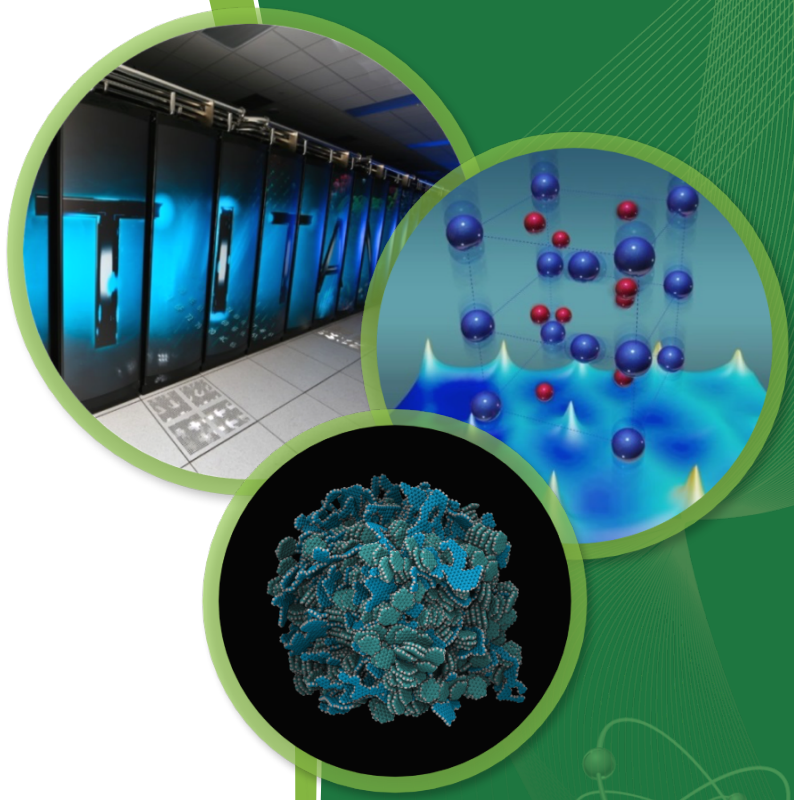
Havens: Explicit Reliable Memory Regions for HPC Applications



Saurabh Hukerikar, Christian Engelmann
Computer Science & Mathematics Division
Oak Ridge National Laboratory

ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory

Outline

- Motivation: Trends in HPC architectures
- Challenges for future extreme-scale systems
- Havens:
  - Memory-management with emphasis on reliability
  - Context: region-based memory management
  - Models of reliability through the use of havens
  - Interfaces: Abstract and Library Implementation
  - Experimental Evaluation: Fault injection and performance studies

The Changing Landscape of Supercomputing

- End of era of node scaling multicore with complex cores
  - ~20% of the Top500 today are heterogeneous (GPU/Accelerator)
- Memory hierarchies are becoming more complex and diverse
  - *Interface Standards:* HMC, HBM/2/3, LPDDR4, GDDR5X, WIDEIO2
  - *Manufacturing processes:* 2.5D, 3D Stacking
  - *New devices:* ReRAM, PCRAM, STT-MRAM
- Scientific applications becoming increasingly complex

ORNL's Next Machine: Summit

ATTRIBUTE	TITAN	SUMMIT
Compute Nodes	18,688	~3,400
Processor	(1) 16-core AMD Opteron per node	(Multiple) IBM POWER 9s per node
Accelerator	(1) NVIDIA Kepler K20x per node	(Multiple) NVIDIA Volta GPUs per node
Memory per node	32GB (DDR3)	>512GB (HBM+DDR4)
CPU-GPU Interconnect	PCI Gen2	NVLINK (5-12x PCIe3)
System Interconnect	Gemini	Dual Rail EDR-IB (23 GB/s)
Peak Power Consumption	9 MW	10 MW

Each Summit node will contain 512GB HBM + DDR4, and an additional 800GB NVRAM

ASCR Computing: Upcoming Systems

System attributes	NERSC Now	OLCF Now	ALCF Now	NERSC Upgrade	OLCF Upgrade	ALCF Upgrade	
Planned Installation	Edison	TITAN	MIRA	Cori 2016	Summit 2017-2018	Theta 2016	Aurora 2018-2019
System peak (PF)	2.4	27	10	>30	>150	>8.5	>150
Peak Power (MW)	3	8.2	4.8	< 3.7	10	1.7	13
System memory per node	64 GB	38 GB	16 GB	~1 PB DDR4 + High Bandwidth Memory (HBM)+1.5PB persistent memory	> 1.74 PB DDR4 + HBM + 2.8 PB persistent memory	>480 TB DDR4 + High Bandwidth Memory (HBM)	> 7 PB High Bandwidth On-Package Memory Local Memory and Persistent Memory
Node performance (TF)	0.460	1.452	0.204	> 3	> 40	> 3	> 17x Mira
	Intel Ivy Bridge	AMD Opteron NVidia Kepler 64-bit	PowerPC A2	Intel Knights Landing many core CPUs Intel Haswell CPU in data partition	Node processors Multiple IBM Power9 CPUs & multiple NVidia Voltas GPUS	Intel Knights Landing Xeon Phi many core CPUs	Knights Hill Xeon Phi many core CPUs
System size (nodes)	5,600 nodes	18,688 nodes	49,152	9,300 nodes, 1,900 nodes in data partition	~3,500 nodes	>2,500 nodes	>50,000 nodes
System Interconnect	Aries	Gemini	5D Torus	Aries	Dual Rail EDR-IB	Aries 2 nd Generation	Intel Omni-Path Architecture

Challenges for Next Generation HPC

Old Optimization Targets	New Optimization Targets
Peak: clock frequency as primary limiter for performance improvement	**Power:** is first-order design constraint for future HPC system design
Cost: FLOPs are biggest cost for system: optimize for compute	**Cost:** Data movement dominates: optimize to minimize data movement
Concurrency: Modest growth of parallelism by adding nodes	**Concurrency:** Exponential growth of parallelism within chips
Memory scaling: maintain byte per flop capacity and bandwidth	**Memory Scaling:** Compute growing 2x faster than capacity or bandwidth
Locality: MPI+X model (uniform costs within node & between nodes)	**Locality:** must reason about data locality and possibly topology
Uniformity: Assume uniform system performance	**Heterogeneity:** Architectural and performance non-uniformity increase
Reliability: It's the hardware's problem	**Reliability:** Cannot count on hardware protection alone

Need for software-based techniques to navigate these emerging challenges

Havens: Reliable Memory Regions

- **Memory-Management Style**
- Havens represent areas of memory
- Program objects are allocated inside the havens
- **Reliability of the haven implicitly guaranteed**
- **Makes no assumptions about hardware-based protection schemes**
- Various deallocation policies possible

```
Haven h = new_haven();

double* x = haven_alloc(h, N * N * sizeof(double));

work(x);

delete_haven();
```

Memory Management Paradigms

Explicit individual allocation/deallocation

- programmer must request each memory block allocation/deallocation
- malloc/free discipline of C, or new/delete operators in C++

Garbage Collection

- Run-time component (garbage collector) periodically deallocates memory blocks whose addresses are not known to the executing program

Reference Counting

- Form of lightweight garbage collection scheme - memory blocks are deallocated as soon as there are no more pointers pointing to them.

Stack Allocation

- Pre-scheduling of allocation/deallocation of memory blocks on LIFO discipline

Havens: Abstract Interface

haven_create:
- request for the creation of a haven by an application
- returns a handle to the memory region, but no memory is allocated.
- error protection for haven scheme is specified

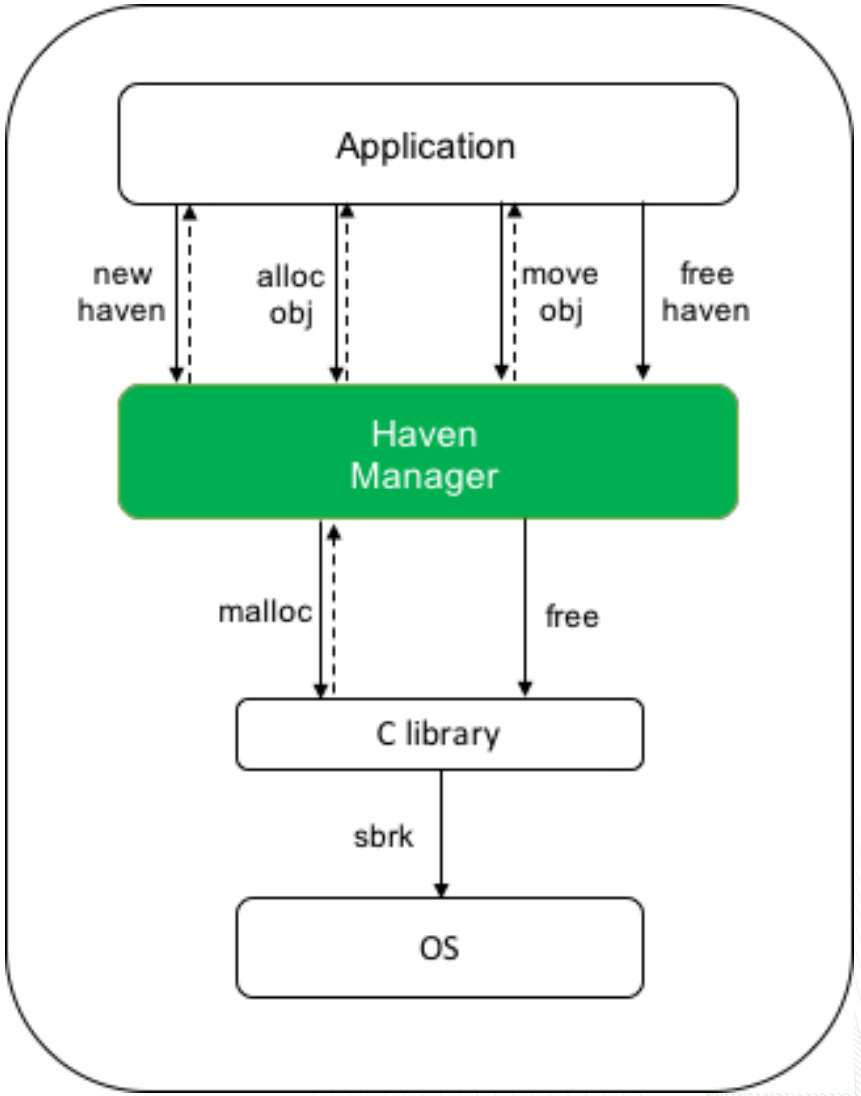
haven_alloc:
- request for a specified block of memory “within” a haven
- results in the allocation of the memory
- initialization of state related to the protection scheme

haven_delete:
- indicates intent to delete an object within a haven
- memory is not released until the haven is destroyed

haven_read, haven_write:
- read/update the program objects contained in the haven
- state of haven’s resilience mechanism updated

haven_destroy:
- results in all memory blocks allocated in the region to be deallocated; memory is available for reuse.
- no further operation on the haven are permitted

haven_relax, haven_robust:
- enable the error protection scheme applied to a haven to be turned on and off



Historical Context

- Regions [Tofte & Talpin, 1994]
  - Proposed as an alternative to garbage collection
  - Intended for functional languages (Implemented in MLKit Compiler for Standard ML)
  - Static lifetime analysis for grouping objects into regions
- Improvements to Region-based Memory Management
  - Deallocation Policies
    - per object deallocation, hybrid garbage collection
  - Safety
    - Reference counting, Reachability (GC), Per region reference counting (RC)
    - Sans stack discipline (Aiken et al.)
  - Expressivity
    - Prolog, Cyclone, Haskell, RC

Havens: Allocation & Deallocation

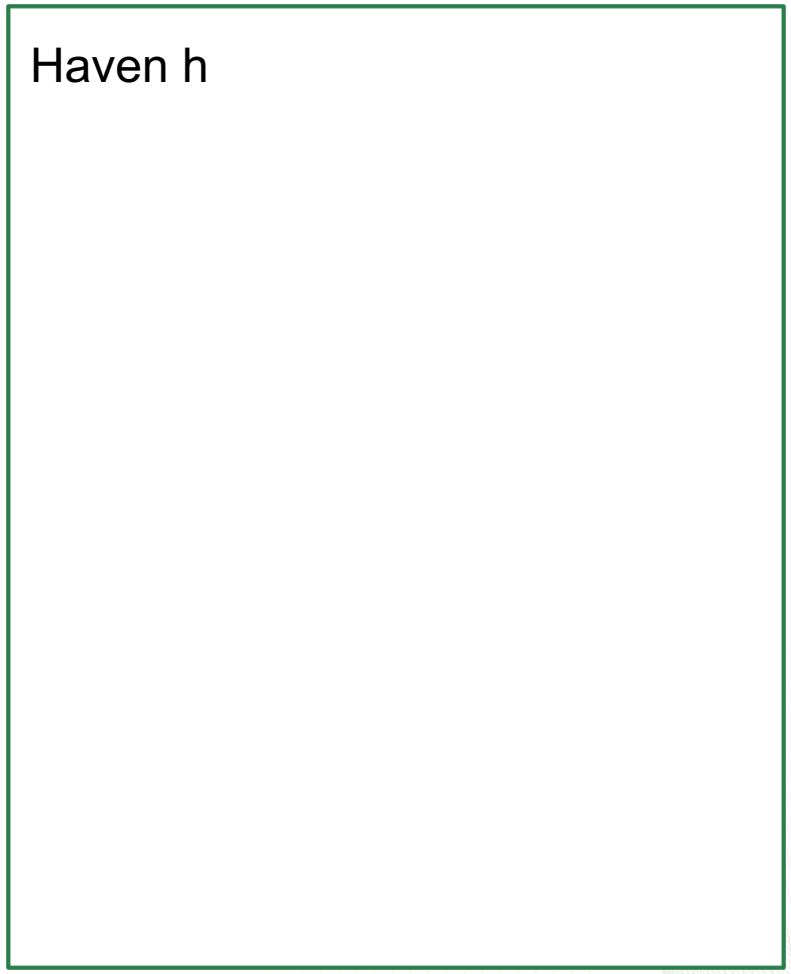
- Basic strategy
  - Resilience characteristic of the haven defined
  - Reliability features apply to all objects “inside” the haven
  - Individual allocations for program objects
  - Per-region deallocation
    - Per-object deallocation is an illegal operation
    - Haven delete operation frees entire memory of the haven

Library Implementation of Interface

- Haven-type handle
  - used to create handle objects
  - contains the bounds of page addresses that make up the memory region.
  - passed as argument to the allocation and deallocation functions
- Heap memory divided into fixed-size pages
  - Each new haven creation is aligned on a page-size boundary.
  - Haven manager maintains a linked list of these pages.
  - No changes on the representation of regular pointers
  - Dangling-pointer dereferences - *not supported (yet)*
- Library functions
  - haven_alloc(), haven_new() implement the allocation of objects inside a haven.
  - haven_delete() operation concatenates all the haven's page list to a global list of free pages

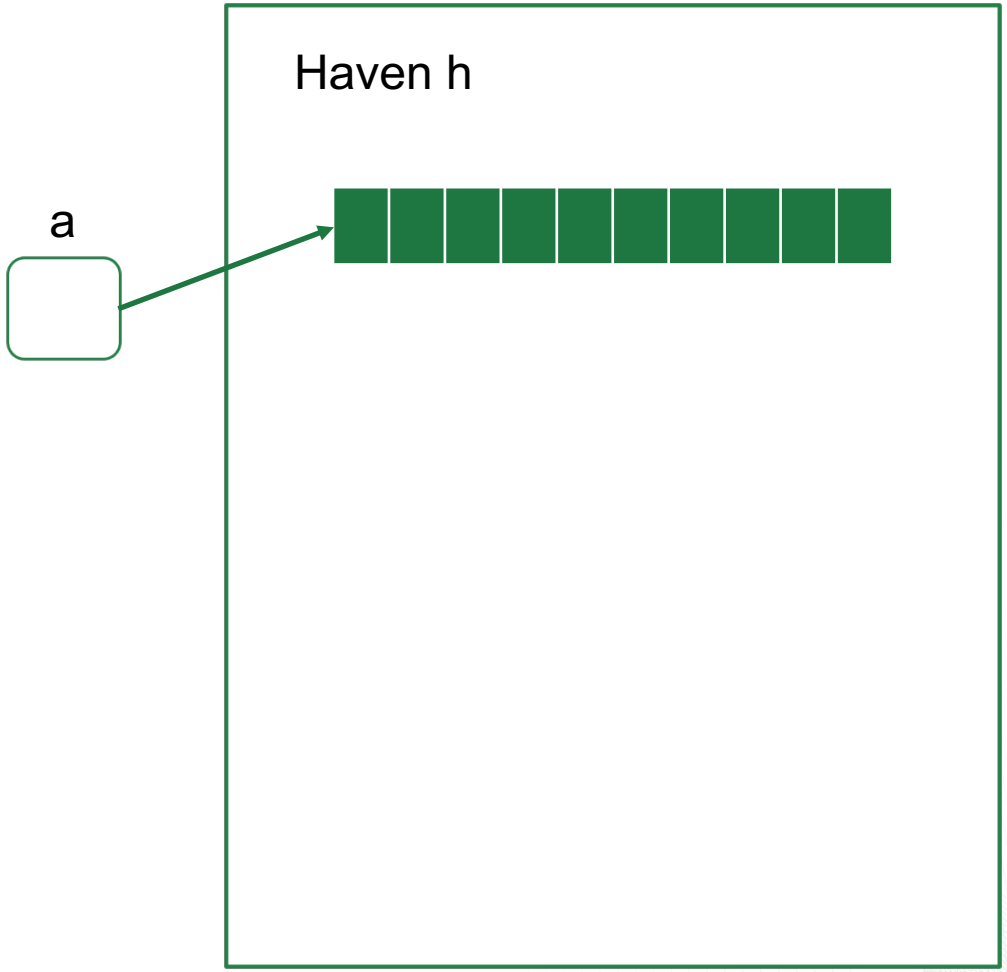
Using Havens: Vector Addition Example

```
Haven h = new_haven();
```



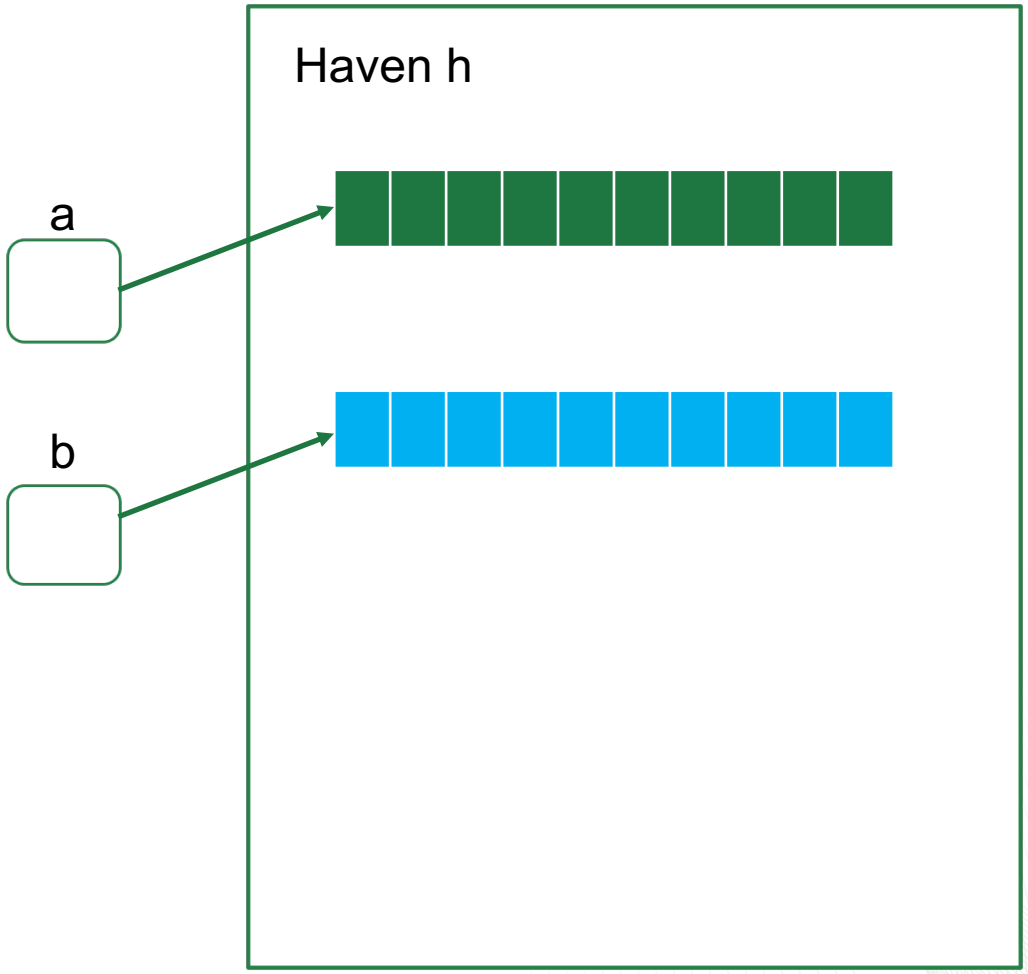
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
```



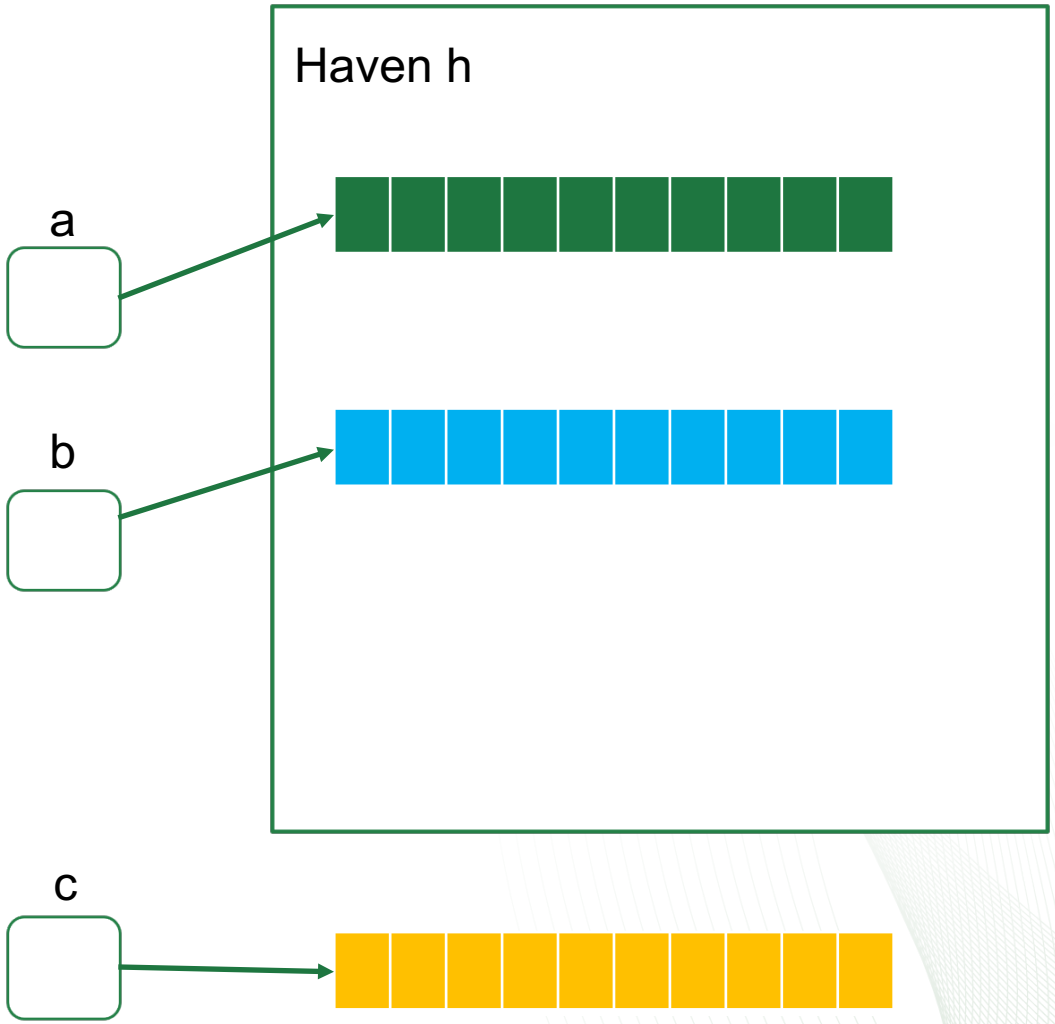
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
double* b = haven_alloc(h, N *
sizeof(double));
```



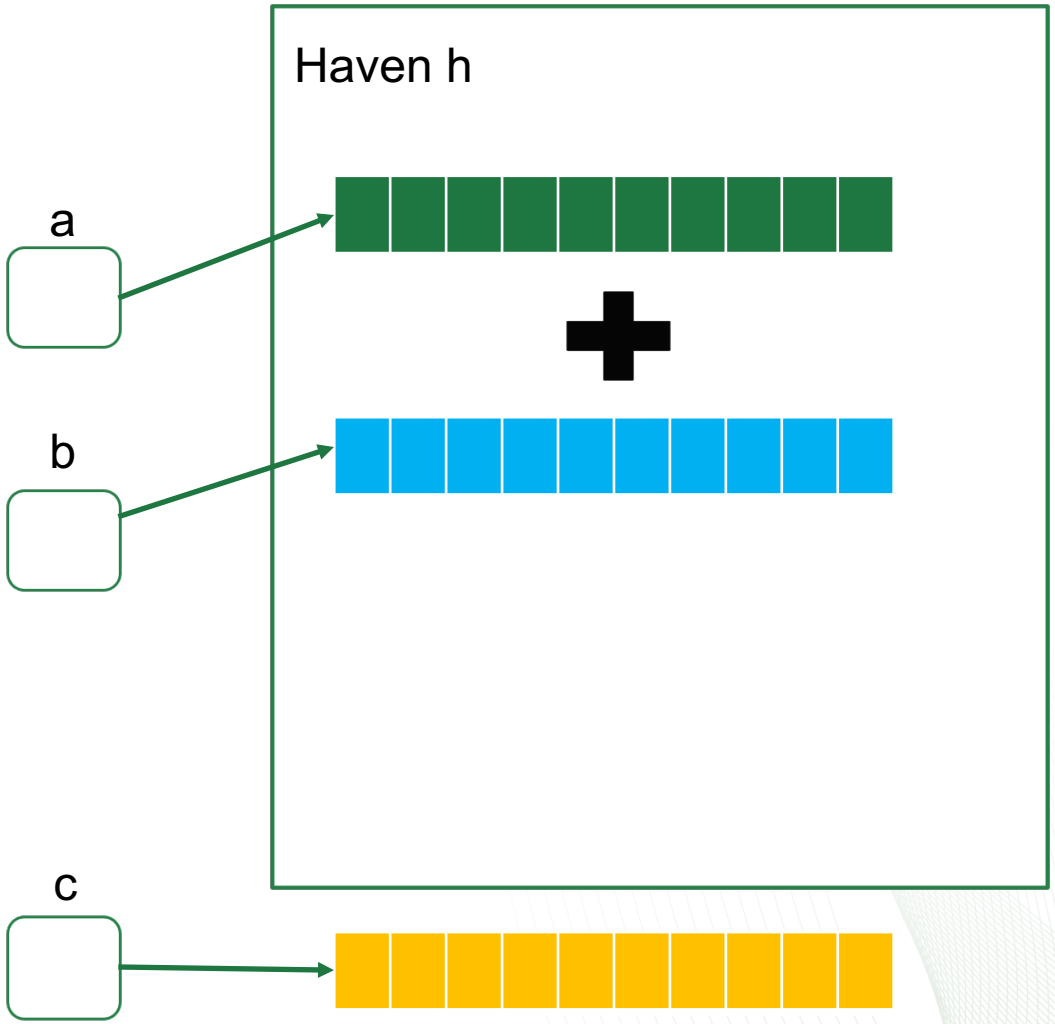
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
        sizeof(double));
double* b = haven_alloc(h, N *
        sizeof(double));
double* c = malloc(N *
        sizeof(double));
```



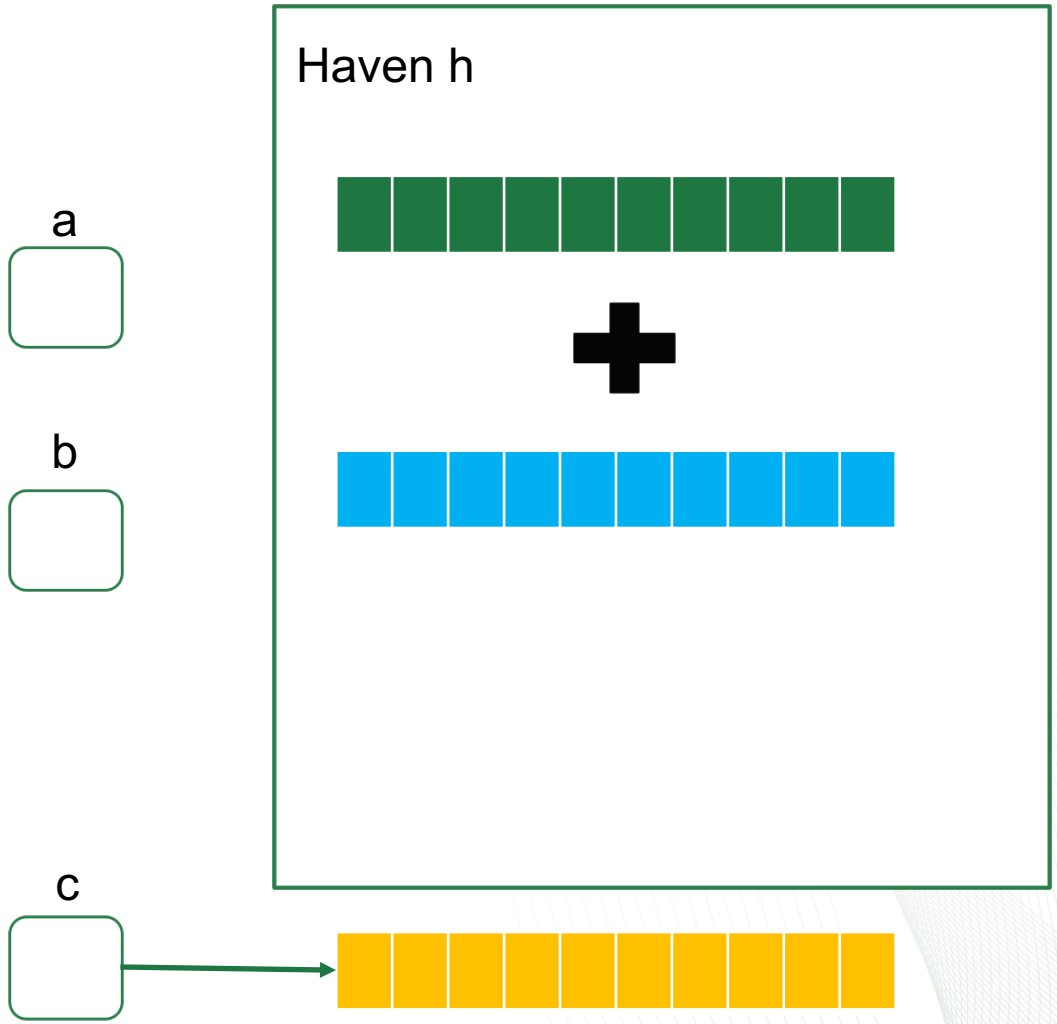
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
double* b = haven_alloc(h, N *
sizeof(double));
double* c = malloc(N *
sizeof(double));

vector_addition(c, a, b);
```



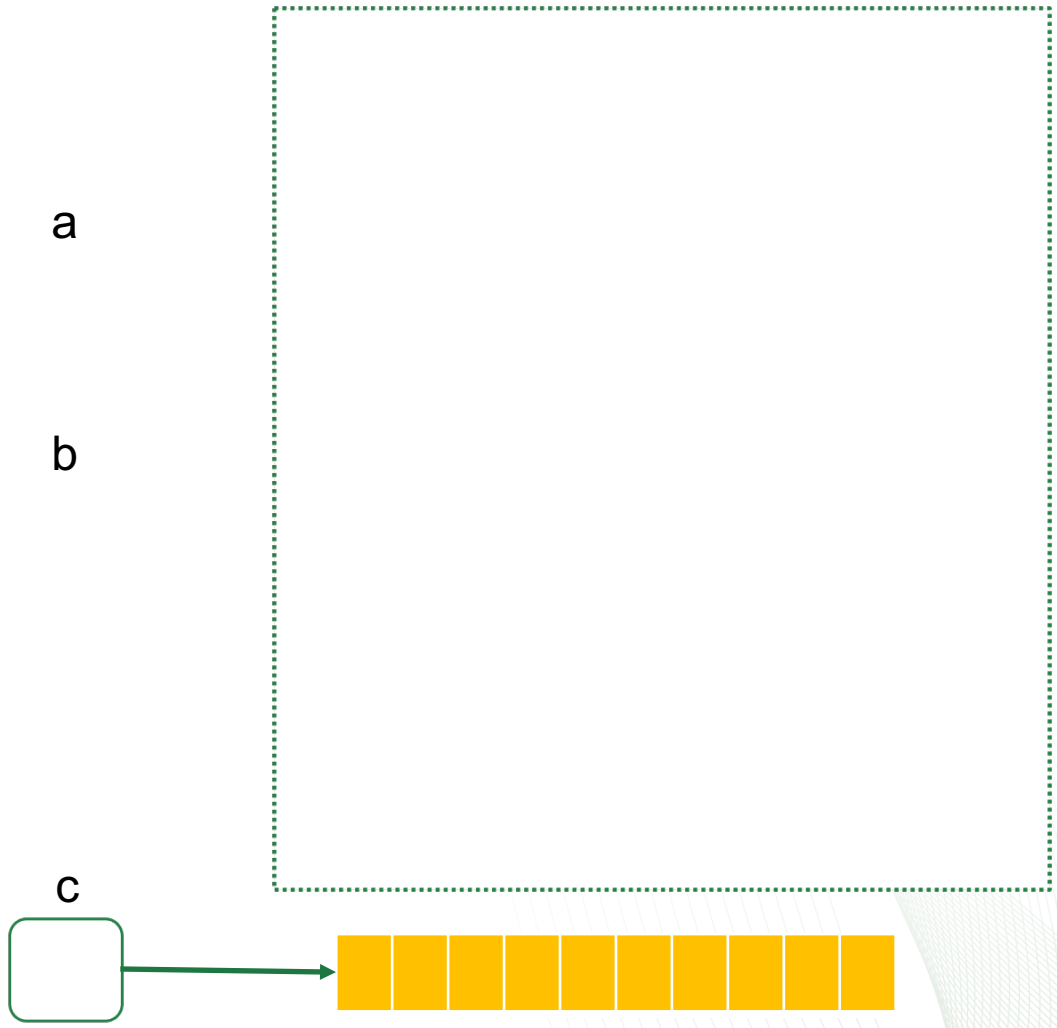
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
double* b = haven_alloc(h, N *
sizeof(double));
double* c = malloc(N *
sizeof(double));

vector_addition(c, a, b);

a = null; b = null;
```



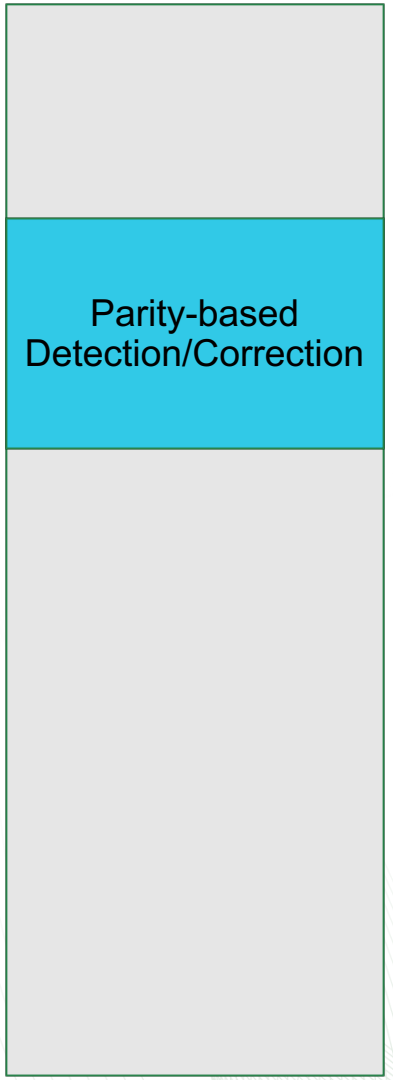
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
double* b = haven_alloc(h, N *
sizeof(double));
double* c = malloc(N *
sizeof(double));

vector_addition(c, a, b);

a = null; b = null;

delete_haven();
```



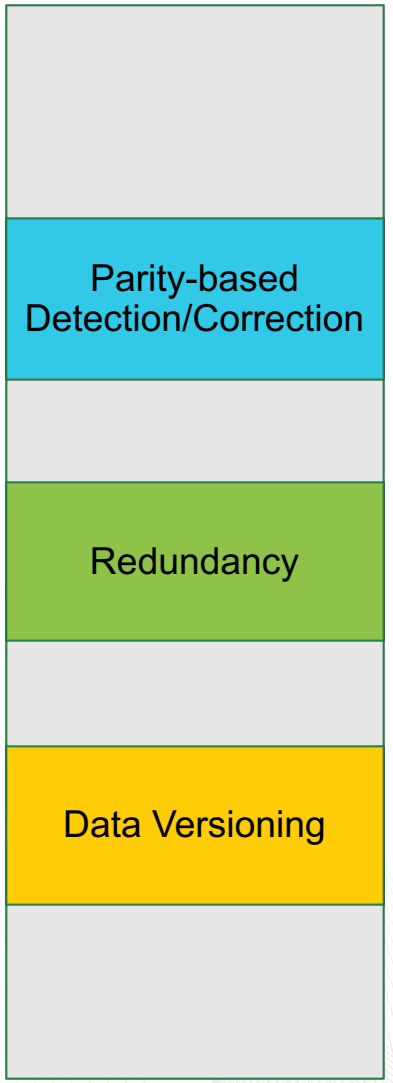
Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



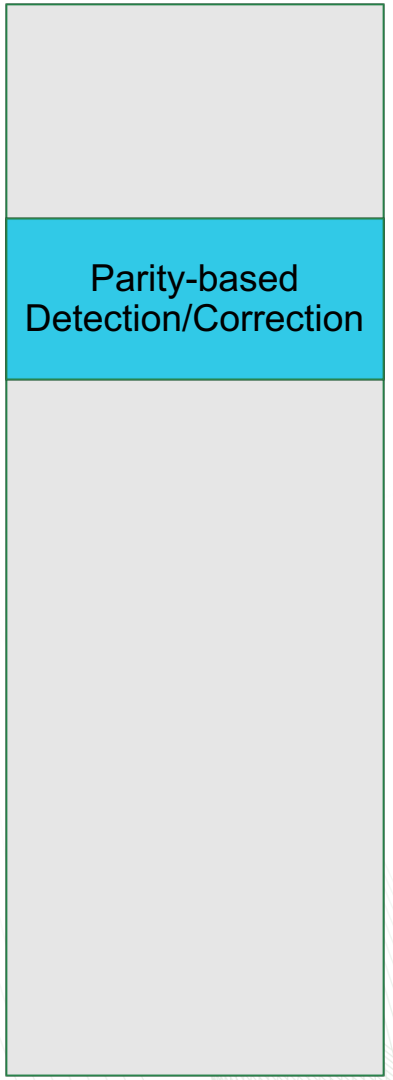
Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.

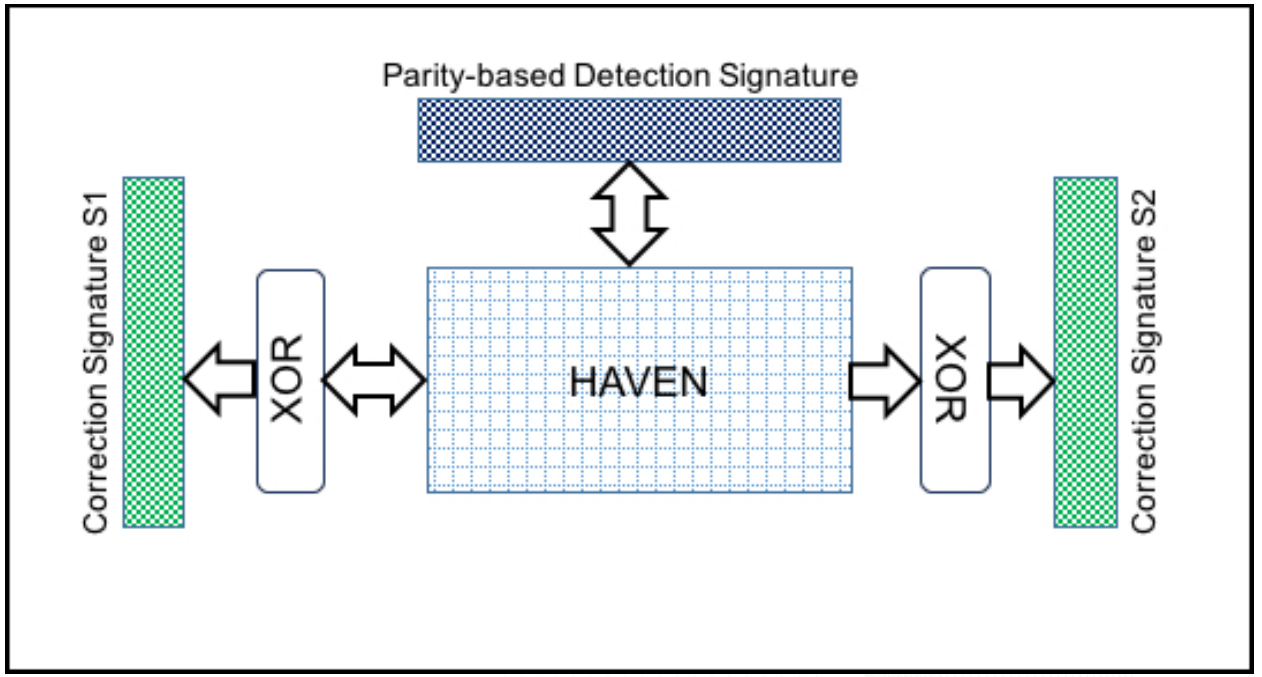
Havens: Expressing Locality

- Havens may be used to express locality:
  - Allocations within a haven may be optimized for cache/memory usage
  - Aggregation of several small objects into a single haven

Parity-based Detection/Correction Scheme

Default Resilience Scheme for Havens
(based on erasure codes)

- 64-bit parity detection signature
- Pair of 64-bit parity correction signatures
- Location of corrupted address within haven determined using detection signature
- S1: XOR of all words written to the haven memory
- S2: XOR of all word updates to the haven memory
- Recovery of corrupted location in haven ➔ S1 XOR S2



Experimental Evaluation: Conjugate Gradient

Selective Reliability Model:

Key Data Structures considered for placement in havens:

- matrix A
- vector b
- vector x (solution, initialized to random values)
- matrix M (preconditioner)
- vector r (auxiliary vector)
- vector p (auxiliary vector)

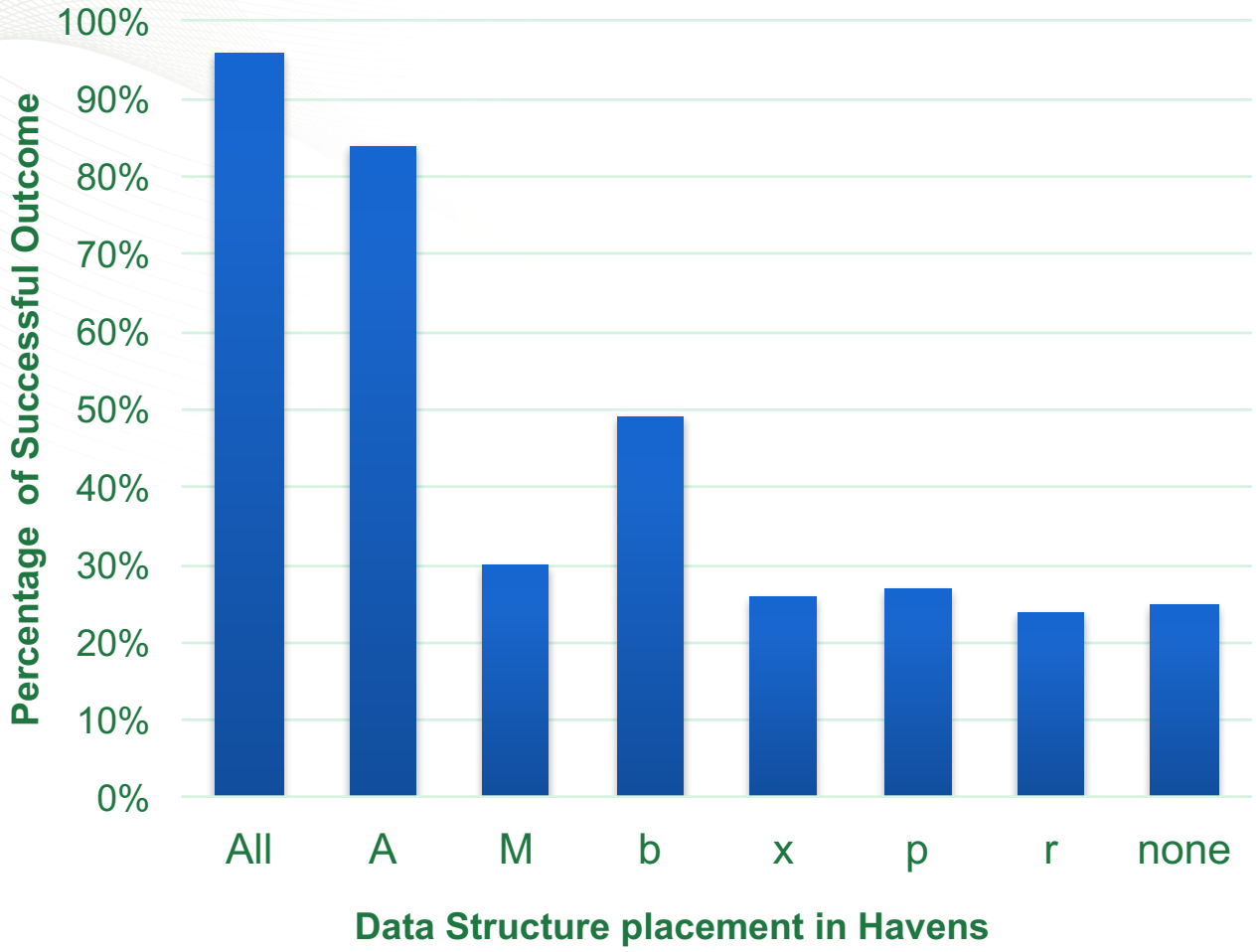
Fault Injection Experiments:

- Objects allocated selectively in havens
- Multiple fault injections per CG run
  - Monte Carlo fault injection

$r_0 = b - Ax_0$
$z_0 = M^{-1} r_0$
$p_0 = z_0$
$k = 0$
procedure REPEAT(A, M, r, z, p, x)
  $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  if r_{k+1} **is sufficiently small then**
    exit loop
  end if
  $z_{k+1} = M^{-1} z_{k+1}$
  $\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$
  $k = k + 1$
end procedure

Algorithm: Preconditioned Conjugate Gradient

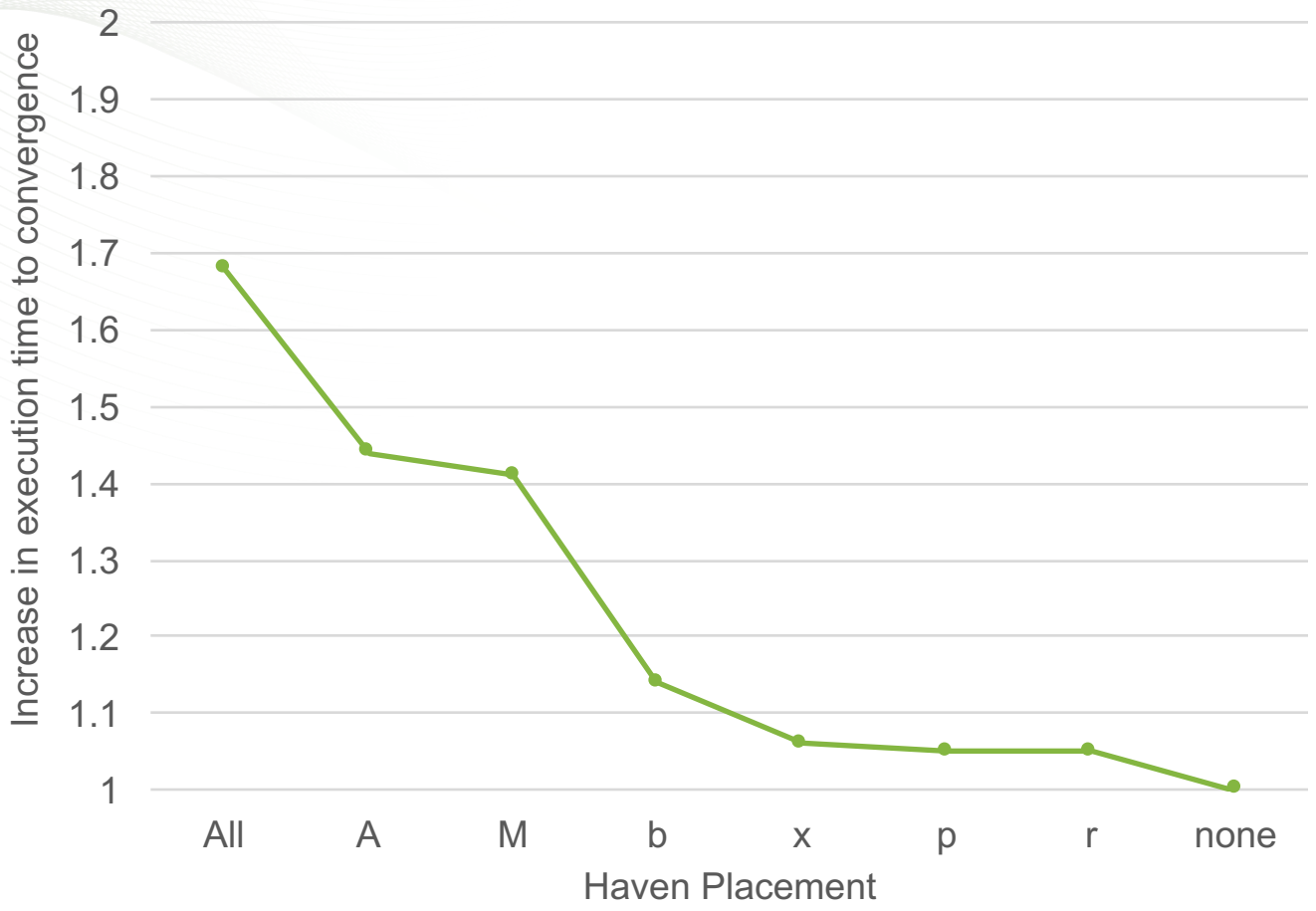
Results: Conjugate Gradient Solver



Individual Allocation of CG objects into Havens

- matrix A occupies dominant part of the solver's memory
  - Over 50% of the active address space
- matrix A, vector B higher error sensitivity:
  - errors in the operand matrix A or vector b fundamentally changes the linear system being solved
- preconditioner matrix M, vectors x, p, r demonstrate lower sensitivity to errors
  - errors typically lead to performance degradation, but solver converges.

Results: Performance Overhead

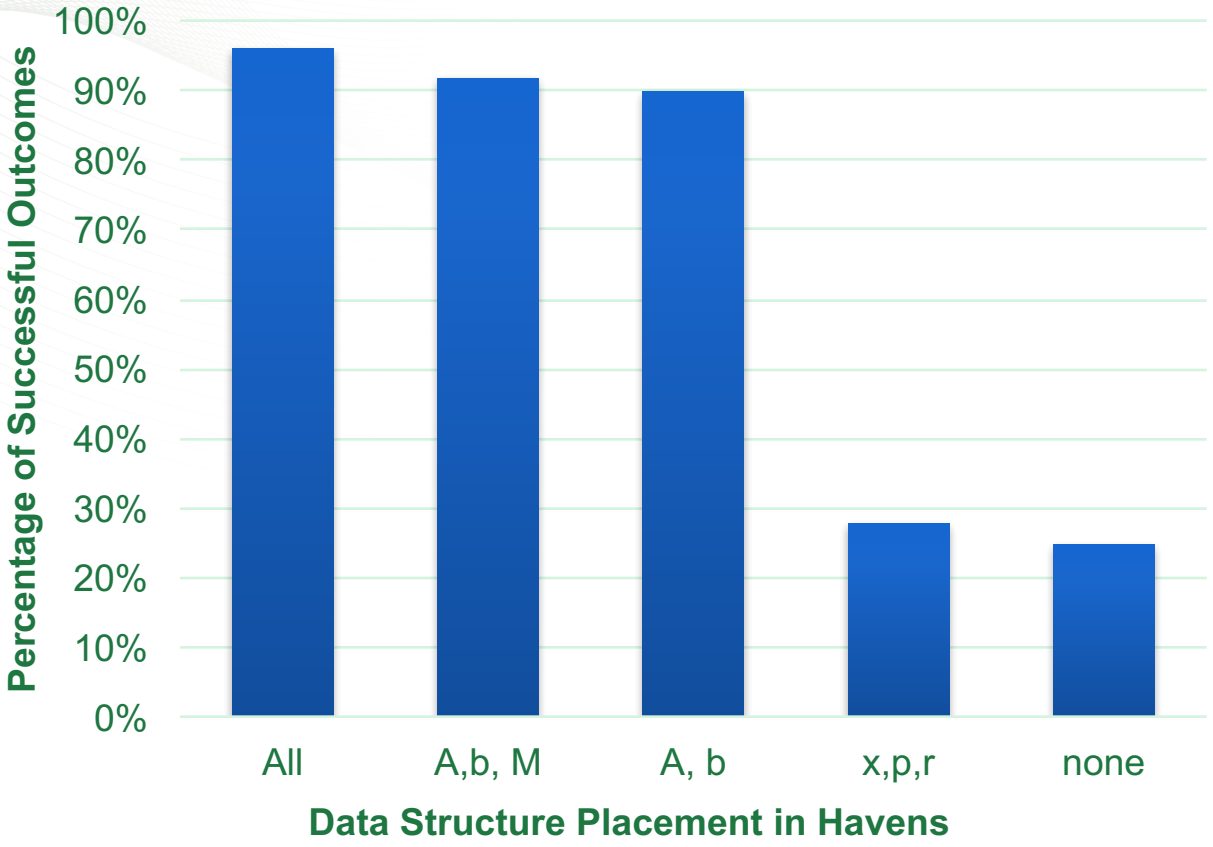


Baseline: memory allocations using standard malloc/free interfaces

- The performance overhead generally tracks the memory footprint of the program objects
- Naïve placement of **all** objects into havens proves very expensive given their impact on overall application resilience.

Results: Selective Placement

Higher Fault Coverage ⇏ Higher Application Resilience



Strategic Placement:

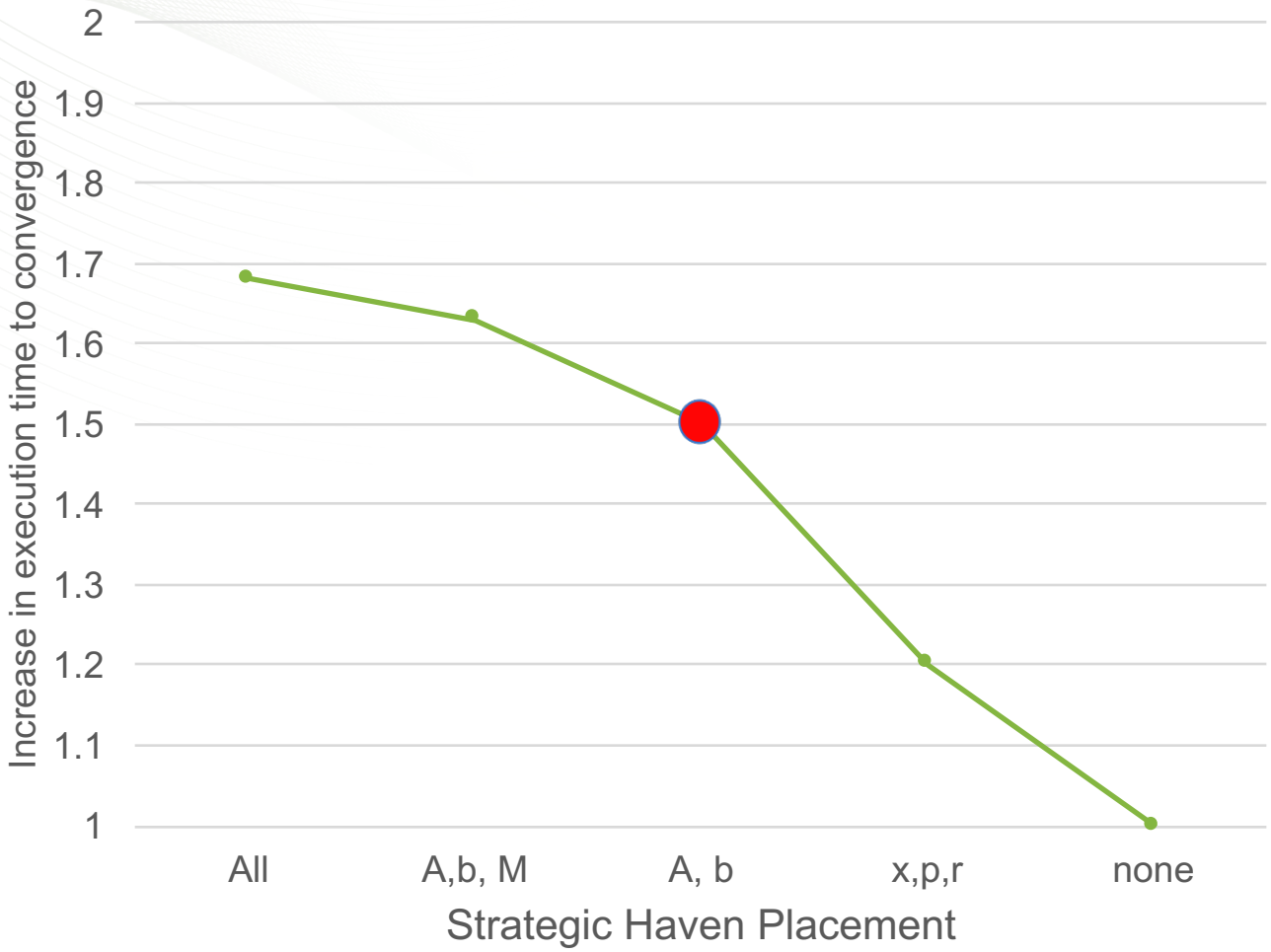
Static state: program variables initialized at the start and remain unmodified during the computation

Dynamic state: data structures that are continuously updated

Computational environment: program code, environment variable, pointer variables, etc.

- Protecting the static state (operand matrix A, vector b, matrix M) provides higher application resilience
- Removing M from the protection domain of havens makes only a small difference to application resilience

Using Havens in CG: Performance Overhead



Baseline: memory allocations using standard malloc/free interfaces

- Based on fault injection study and performance overhead: *Protecting the operand matrix A, vector b, yields most reasonable balance between application resilience v increase in time to convergence.*

Future Directions

- Expressiveness*
  - Structured memory management
  - Limits number of code changes
  - Movement of objects between regions
- Safety
  - Dangling references to objects
- Complement hardware-based mechanisms
  - Present implementation makes no assumptions about hardware-based resilience schemes.
- Optimizations for resilience schemes

* Language Support for Reliable Memory Regions at the International Workshop on Languages and Compilers for Parallel Computing, Rochester, NY, September 28-30, 2016

Summary

- HPC node and system architectures are changing
- Need for software techniques to navigate heterogeneity & complexity
- Havens: Memory management with reliability as a prominent feature
- Enables flexibility
  - Allocation of program objects based on their individual error sensitivity
  - Each haven may have separate protection scheme
  - Allows complementing hardware-level solutions

Questions?