

# Adapting to Memory Pressure from within Scientific Applications on Multiprogrammed Environments

Richard Tran Mills, Andreas Stathopoulos, Dimitrios S. Nikolopoulos

Department of Computer Science  
The College of William & Mary  
July 20, 2004



W&M Computer  
Science  
WILLIAM AND MARY

Work supported by NSF and a DOE Computational Science Graduate Fellowship administered by Krell Institute. Performed using facilities made possible by Sun Microsystems, NSF, the Commonwealth of Virginia, and the College of William and Mary.



# Research goals

---

Goal: Enable scientific codes to run efficiently in non-dedicated, shared resource environments:

- Loose networks of workstations (may share everything!)
- SMP machines (share memory, even w/ space-shared CPUs)
- Big buzzword: "Grid Computing"

Work has focused on two shared resources (CPU + RAM):

- CPU: Dynamic load balancing via algorithmic modifications
  - Specific class of inner-outer iterative methods
  - Simple *algorithmic* modifications
  - Very low overhead
- RAM: Dynamic adaptation to memory pressure



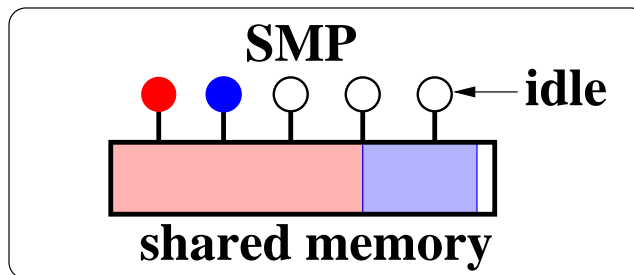
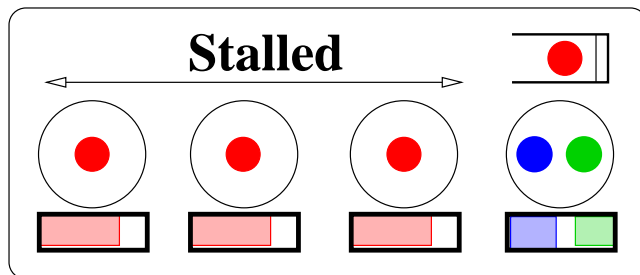
# Memory pressure in shared environments

Over-committing memory can cause severe performance penalties! E.g.:

- Multigrid, no memory pressure: 14 seconds per iteration
- With Matlab QR factorization: 472 seconds per iteration

System response to memory shortage:

- The system may move a process to wait queue



Synchronous parallel programs stall, despite load balancing!



# Memory pressure in shared environments

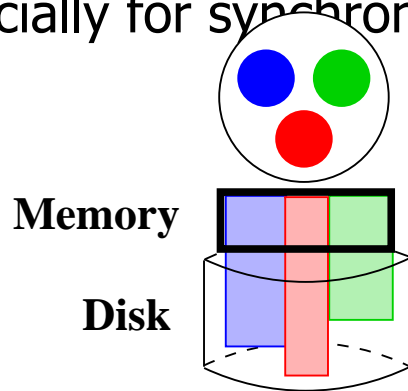
---

Over-committing memory can cause severe performance penalties! E.g.:

- Multigrid, no memory pressure: 14 seconds per iteration
- With Matlab QR calculation: 472 seconds per iteration

System response to memory shortage:

- The system may move a process to wait queue.
- Or, all jobs run, resulting in thrashing.
- Either case is disastrous, especially for synchronous parallel programs!



Many CPU cycles wasted!



# Avoiding/coping with memory pressure

---

Many approaches exist at OS, compiler, middleware levels:

- Adaptive schedulers, VM modifications, compiler support, middleware memory managers, ....

We explore an **application-level** memory adaptation framework.

Why? Application has full knowledge of

- Granularity
- Communication pattern
- Memory access pattern
- Dynamic memory size requirements

Goal: Applications exploit this knowledge to adapt memory requirements to memory availability.



# Memory-adaptation framework

---

Target scientific apps, where large, repetitive data access typical.

Many scientific apps use blocked access pattern:

```
for  $i = 1:P$   
    Get panel  $p_i$  from lower memory level  
    Work on  $p_i$   
    Write results and evict  $p_i$  to lower level
```

Suggests an adaptation framework:

- Use named `mmap()` to access data set
- Cache (`mmap`) as many panels as possible in-core
- Control RSS by varying number of panels cached
- User-defined replacement policy
- Encapsulate caching decisions within `get_panel()`



## Detecting memory shortage/surplus

---

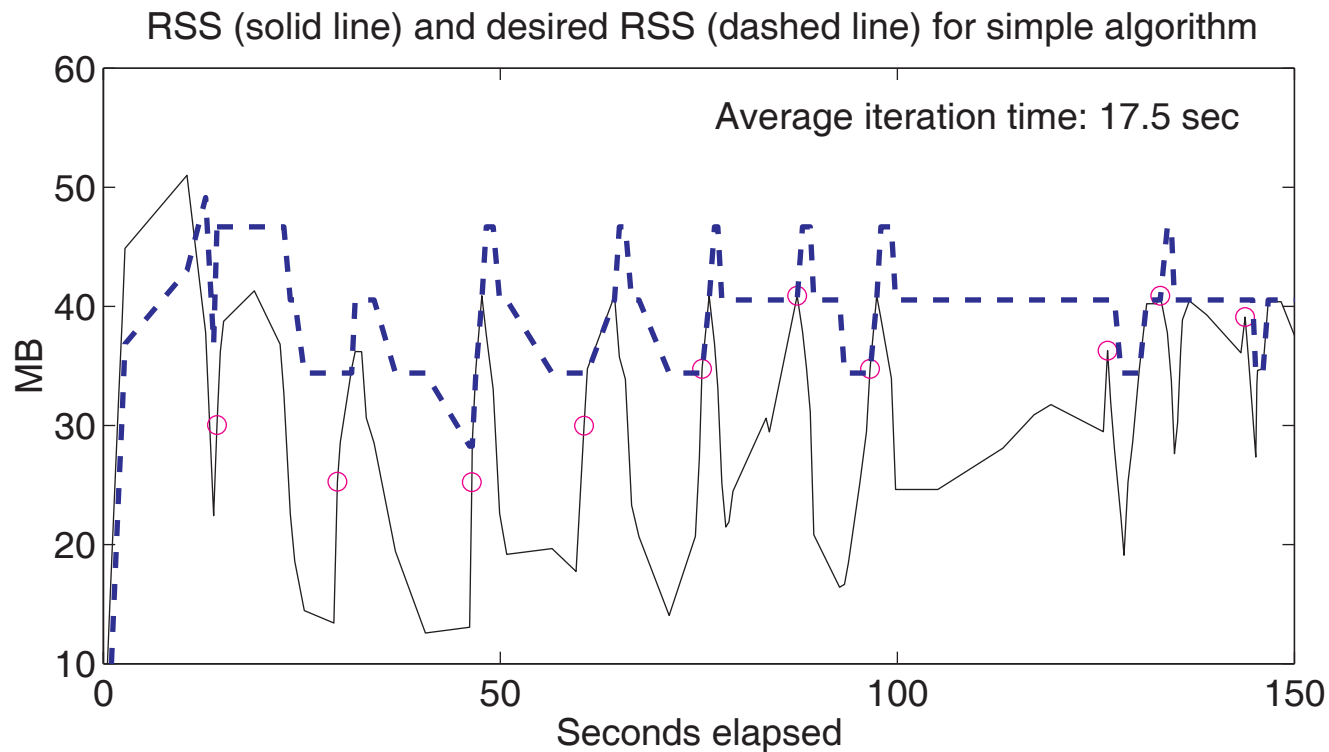
OS provides no accurate estimate of available memory!

- We infer shortage if RSS decreases w/o unmapping by us.
- Detecting surplus is more complicated:
  - Periodically probe for more memory
    - Map new panel w/o unmapping another (increase panels\_in)
  - If memory available, RSS grows by one panel
  - Otherwise, RSS stays constant or decreases
  - Note: Don't probe if  $RSS < dRSS = \text{panels\_in} * \text{panel\_size}$
- Simplest policy: Probe when  $RSS = dRSS = \text{panels\_in} * \text{panel\_size}$



## Simplest policy too aggressive

- Safe RSS/dRSS is 30-35 MB; regularly exceeded!
- Unsuccessful probes drop RSS dramatically.



70 MB adaptive job vs. 70MB dummy job; 128 MB system RAM.



## Use dynamic delay to reduce aggressiveness

---

We must reduce frequency of probes for surplus memory:  
After a detected shortage, delay probing for some time.

Choice of delay must consider two penalties:

- Probe too often → VM system takes memory (“incursion penalty”)  
penalty:  $(\text{maxRSS} / B_w)$
- Probe too infrequently → Wasteful reads from disk (“inaction penalty”)  
penalty:  $((\text{maxRSS} - \text{RSS}) / B_w)$

Can balance two penalties by scaling some base delay by their ratio.

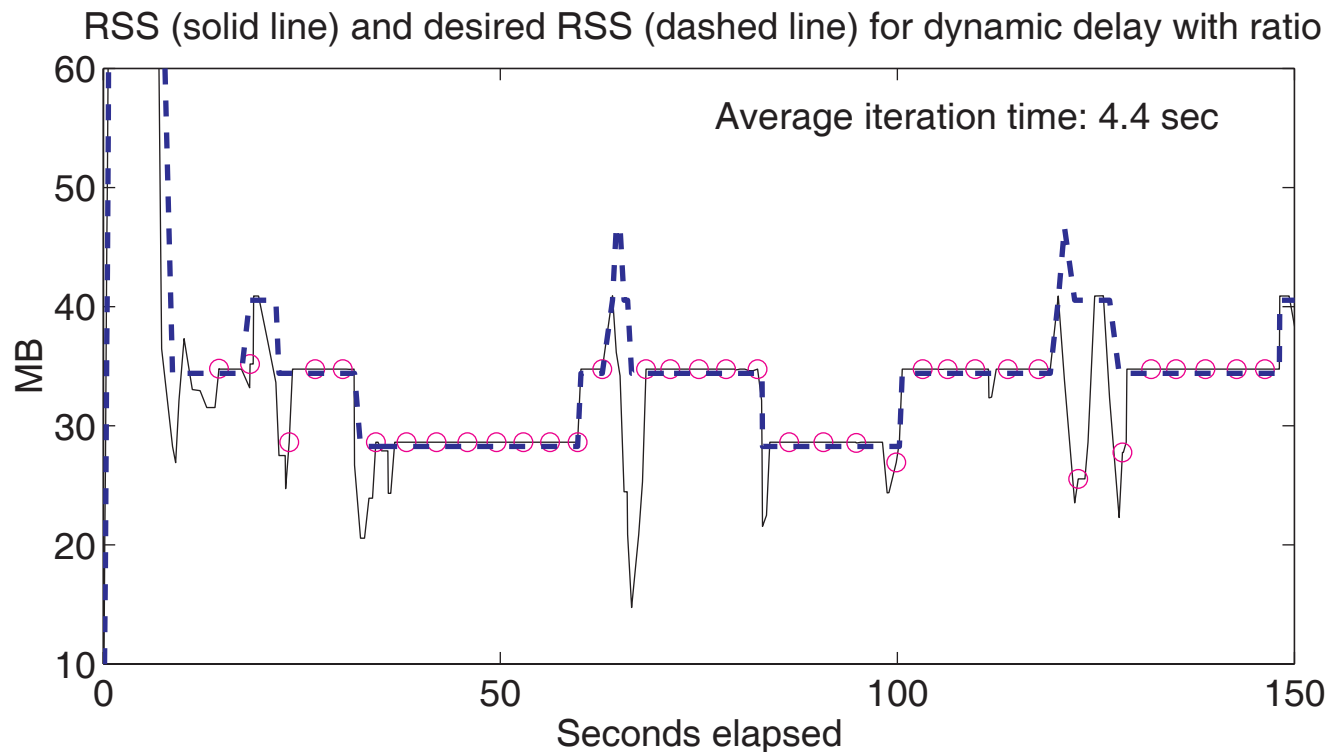
- After shortage, must touch panels to re-load lost pages.  
→ Use time for sweep through all panels as base delay.

$$\text{Delay} = (\text{Time for last full sweep}) * (\text{maxRSS} / (\text{maxRSS} - \text{RSS}))$$



# Dynamic delay improves performance

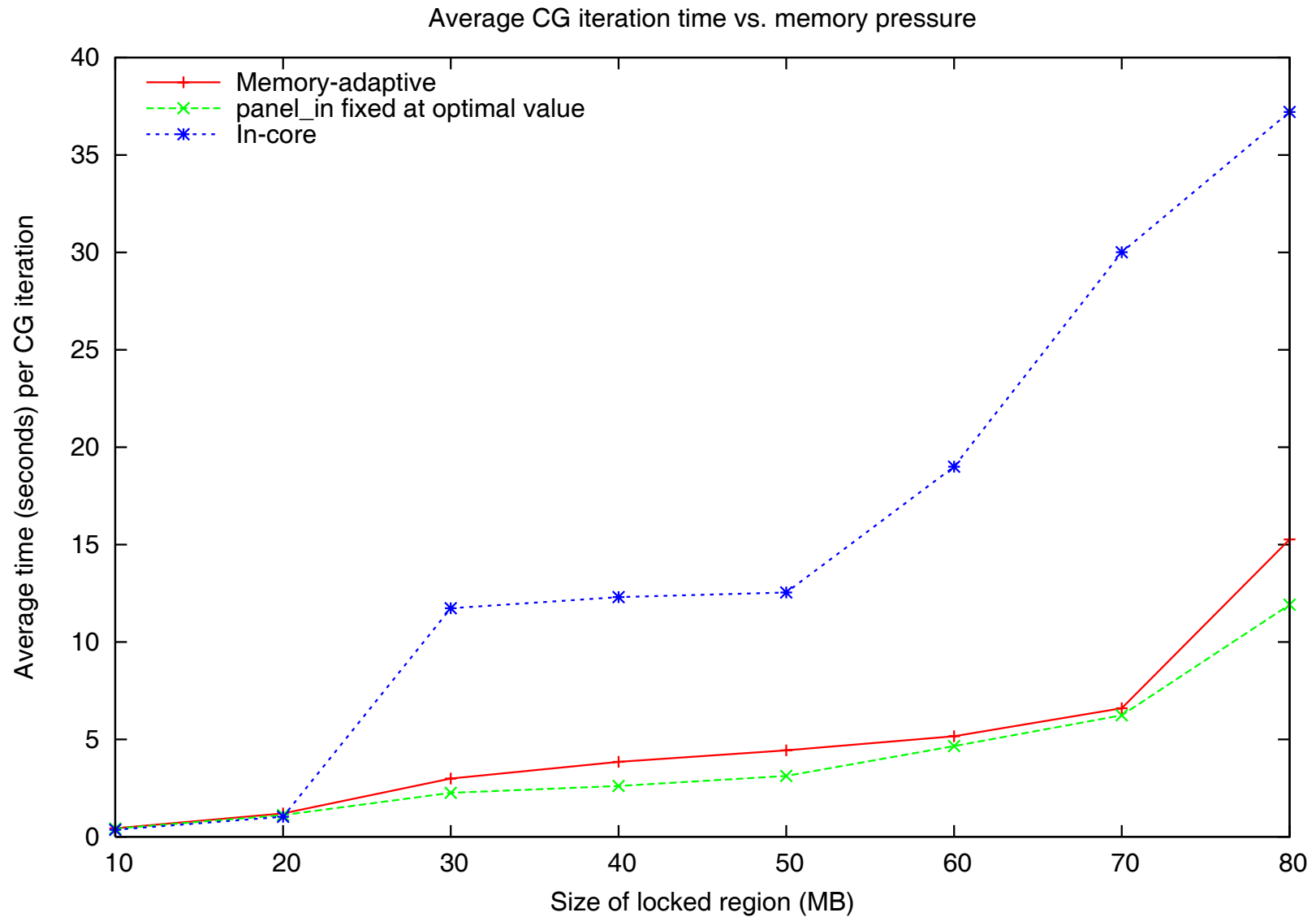
- dRSS quickly finds safe value.
- Fluctuations in RSS greatly reduced.
- Time per matvec reduced from 17.5 sec to 4.4 sec.



70 MB adaptive job vs. 70MB dummy job; 128 MB system RAM.

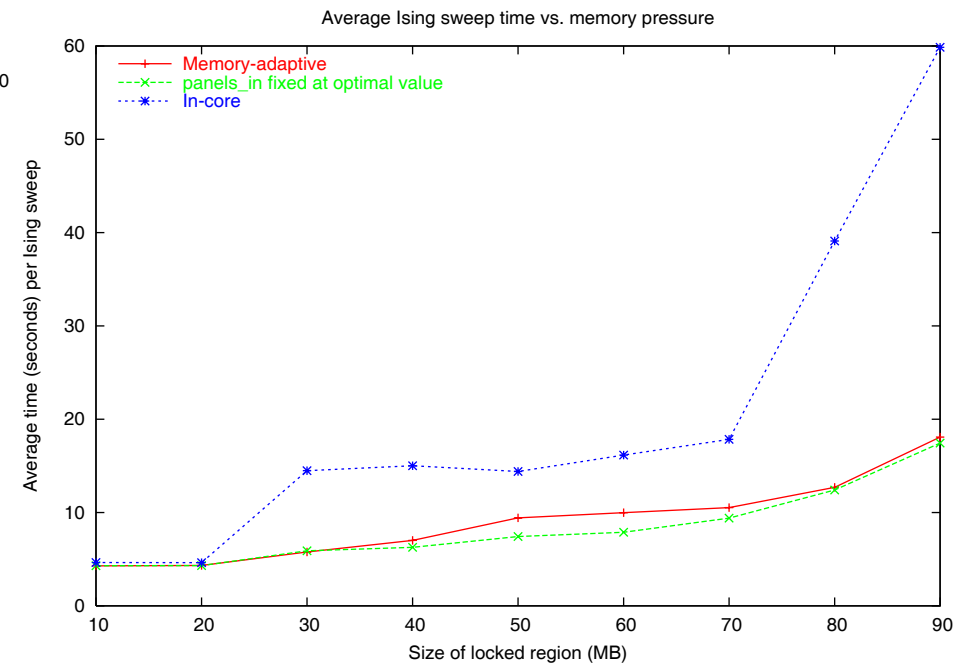
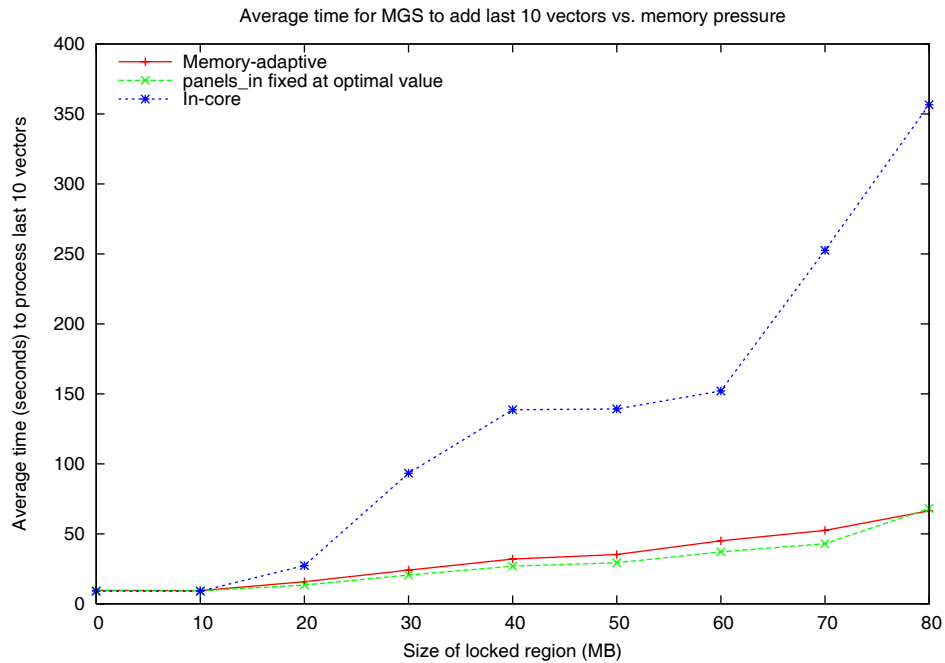


# Graceful performance degradation in CG





# Modified Gram-Schmidt; Ising via Metropolis





## In summary...

---

### Application-level memory-adaptation framework:

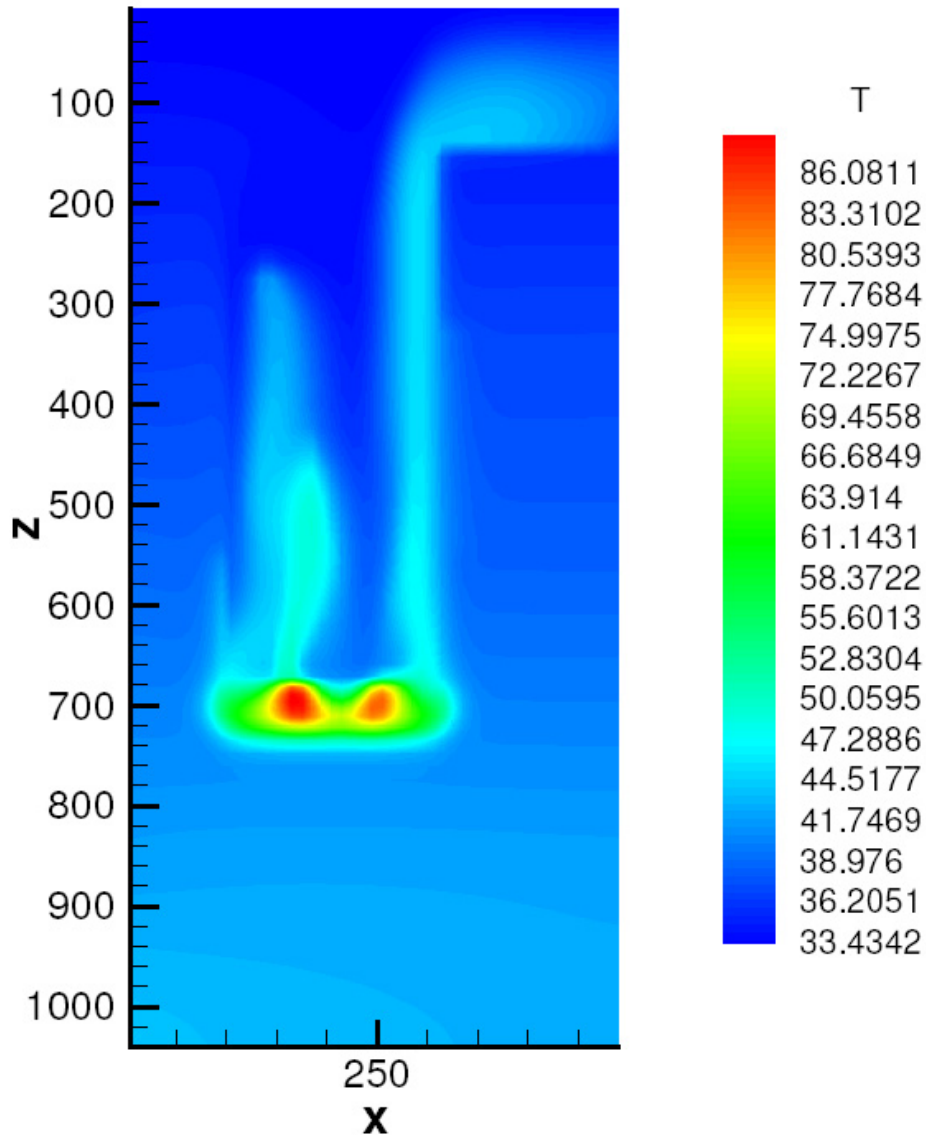
- Enables graceful performance degradation under memory pressure (order of magnitude speedups not atypical)
- Minimal reliance on OS-provided information
- Requires minimal code changes to many scientific kernels
- Suited to non-centrally administered, open systems

### Key contributions:

- Algorithm judges memory availability with single metric
- Demonstrated in three application kernels
- Modular, flexible supporting library



## PFLOW: Multiphase subsurface flow



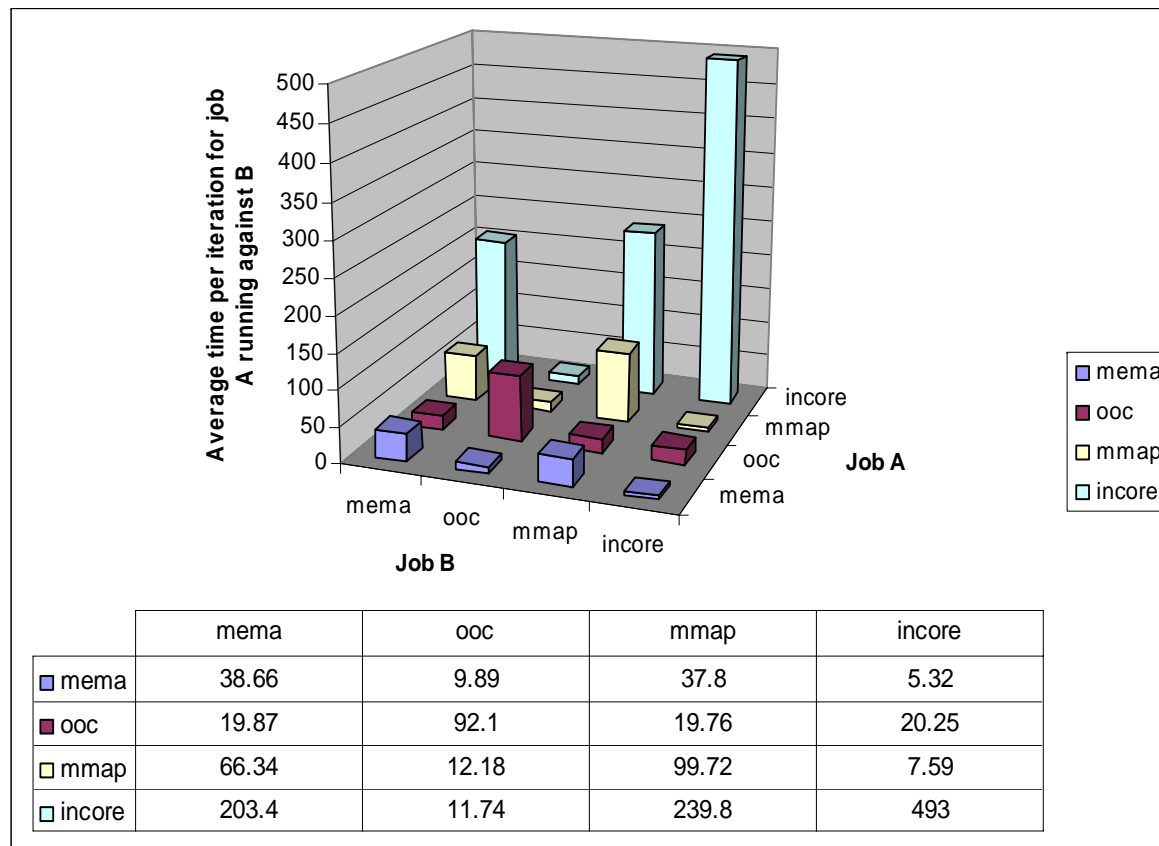
- Parallel, multiphase, fully-implicit subsurface flow
- Object-oriented Fortran 90 code: uses derived types, modules to provide encapsulation
- Uses PETSc iterative solvers, communication constructs



# Comparison of different CG implementations

Compared memory adaptive CG (mema) with other CG codes:

- Conventional in-core (incore)
- Conventional out-of-core (ooc)
- Memory-mapped I/O (mmap)





# Modified Gram-Schmidt

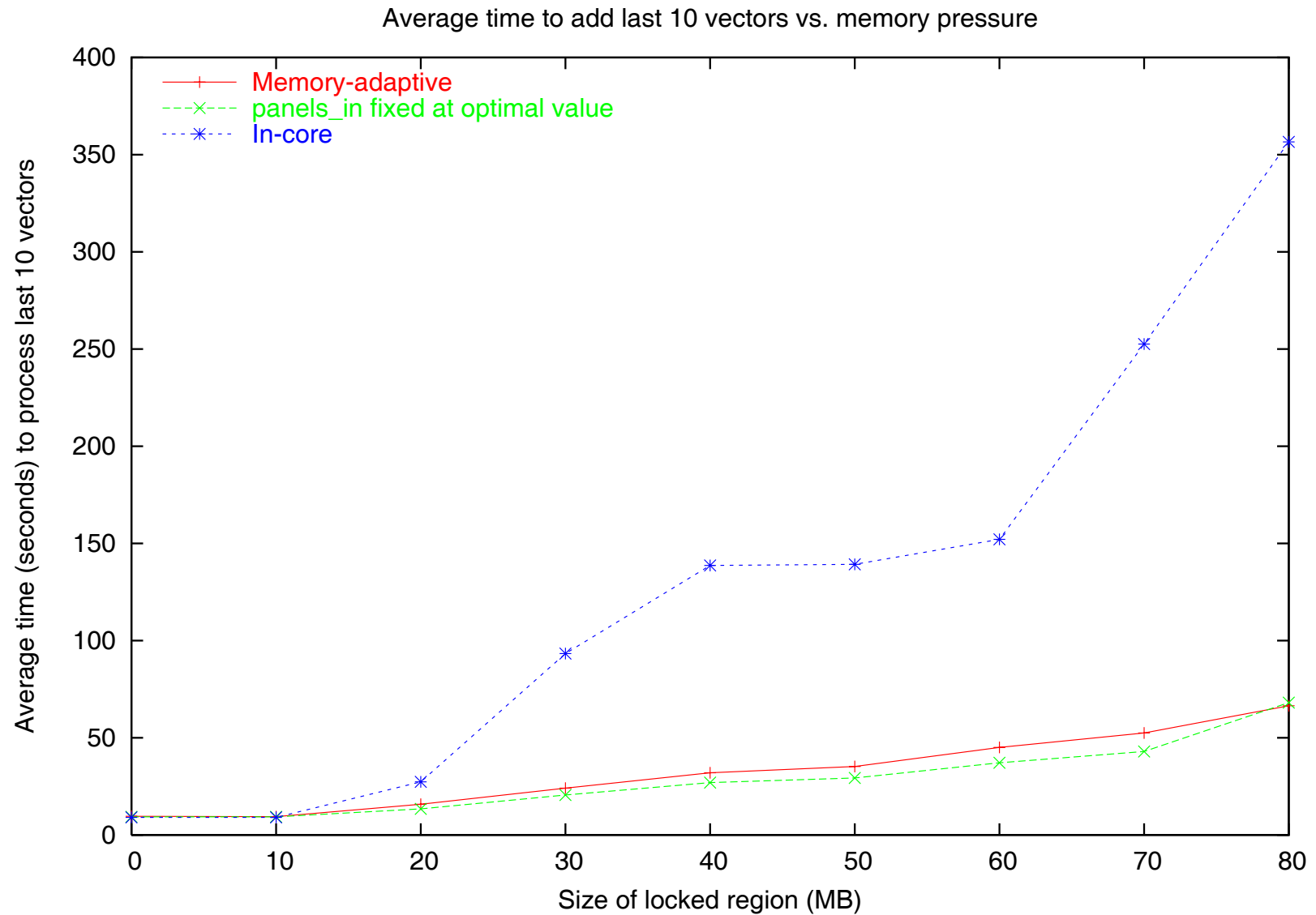
---

From vectors  $W$ , generate orthonormal vector basis  $V$ :

- Orthogonalize  $w_2$  against  $v_1$ :  $v_2 = w_2 - (v_1^* w_2) v_1$
  - Orthogonalize  $w_3$  against  $v_1$  and  $v_2$
  - Orthogonalize  $w_i$  against  $v_1, v_2, \dots, v_{i-1}$
- 
- Working set increases as vectors added to basis.
  - Once added to basis, vector not written again.
- 
- Our test code:
    - Generates random vector, orthonormalizes, adds to basis.
    - Restarts (discards vectors) when max basis size reached.
    - Note: Like GMRES, but doesn't build Krylov space.
    - Can specify min basis size to guarantee memory pressure.



# Modified Gram-Schmidt





## Ising Model via Metropolis

---

- Ferromagnet modeled as rectangular lattice.
- Sites possess only spin up (+1) or down (-1).
- Interaction with four nearest-neighbors only.
- Similar to a wide range of physics models.
  
- Monte-Carlo simulation:
  - Generate configurations that represent equilibrium.
  - Sweep through the lattice, making trial spin flips.
    - Accept new spin if  $\Delta E < 0$  OR
    - Accept with probability  $\exp(-\Delta E/kT)$  otherwise.
  
- Partition lattice row-wise into panels.
- At panel boundaries, need both panels for  $\Delta E$  calc.



# Metropolis Ising model simulation

