

# *Using Tensor Methods, PETSc, and SLEPc to Obtain Exact Cumulative Reaction Probability*

**M. Minkoff and D. Kaushik**



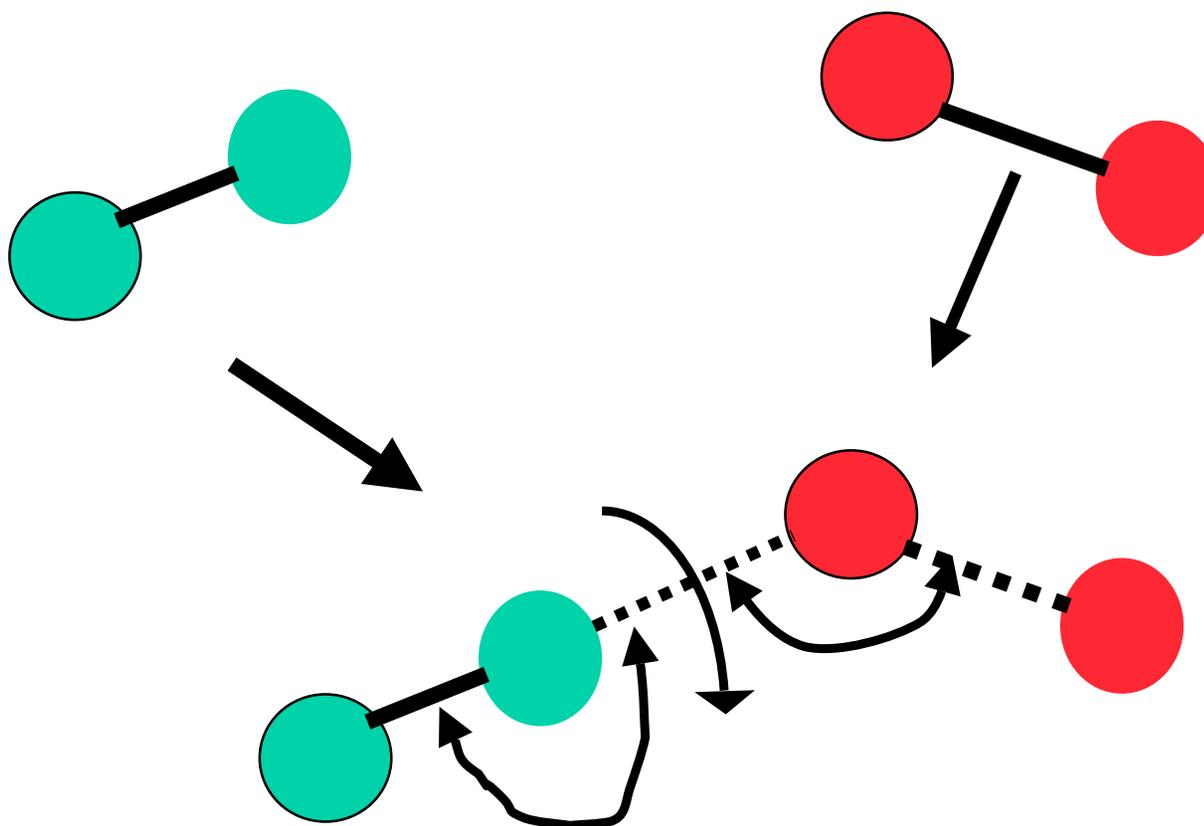
# *Outline*

- **Overview**
- **Description of CRP Simulation Problem**
- **Using PETSc and SLEPc for Application to CRP**
- **Computations for Sparse Matrices**
- **Results for Banded Preconditioning**
- **Future Directions**
- **Tensor Matrix Multiplication**



# *Chemical Dynamics Theory*

*3 angles, 3 stretches  
6 degrees of freedom*



# Chemical Dynamics Theory

- **Reaction rates are related to**

- Cumulative Reaction Probability (CRP),  $N(E)$

$$N(E) = 4 \text{Tr} [ \varepsilon_r^{1/2} G(E)^\dagger \varepsilon_p G(E) \varepsilon_r^{1/2} ]$$

- Where Tr is the trace of the matrix,  $\dagger$  is the adjoint,  $\varepsilon_r$  and  $\varepsilon_p$  are the absorbing potential in the reactant and product regions.
- $\varepsilon_r + \varepsilon_p = \varepsilon$ , the given total absorbing potential.

The Green's functions have the form:

$G(E) = (E + i\varepsilon - H)^{-1}$ , where  $i$  is imaginary and  $H$  is the Hamiltonian.

We need to solve two linear systems (at each iteration):

$(E + i\varepsilon - H)y = x$  and its adjoint where  $x$  is known.

This system is solved via GMRES with preconditioning methods (initially diagonal scaling).

# Hamiltonian Sparsity Pattern

- Sparsity for  $d=2$  and  $d>2$

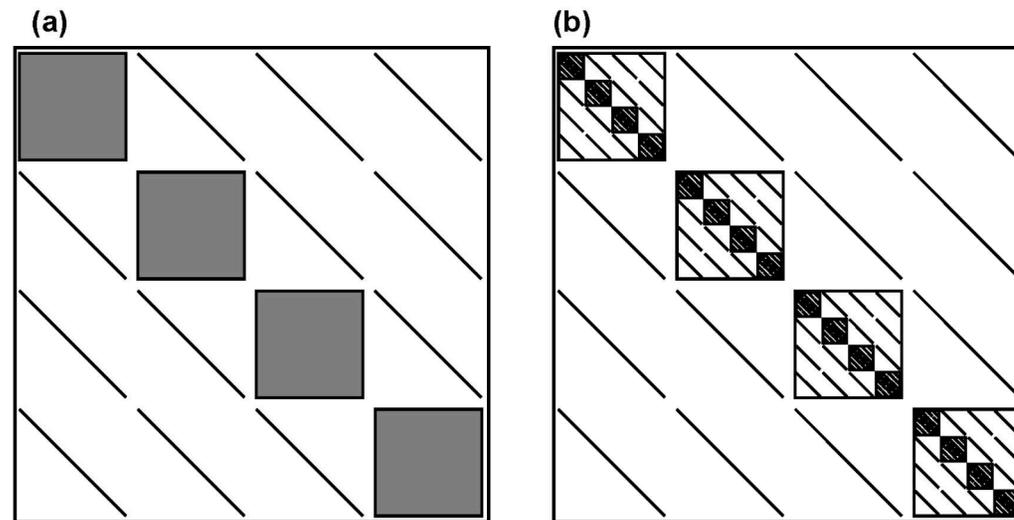
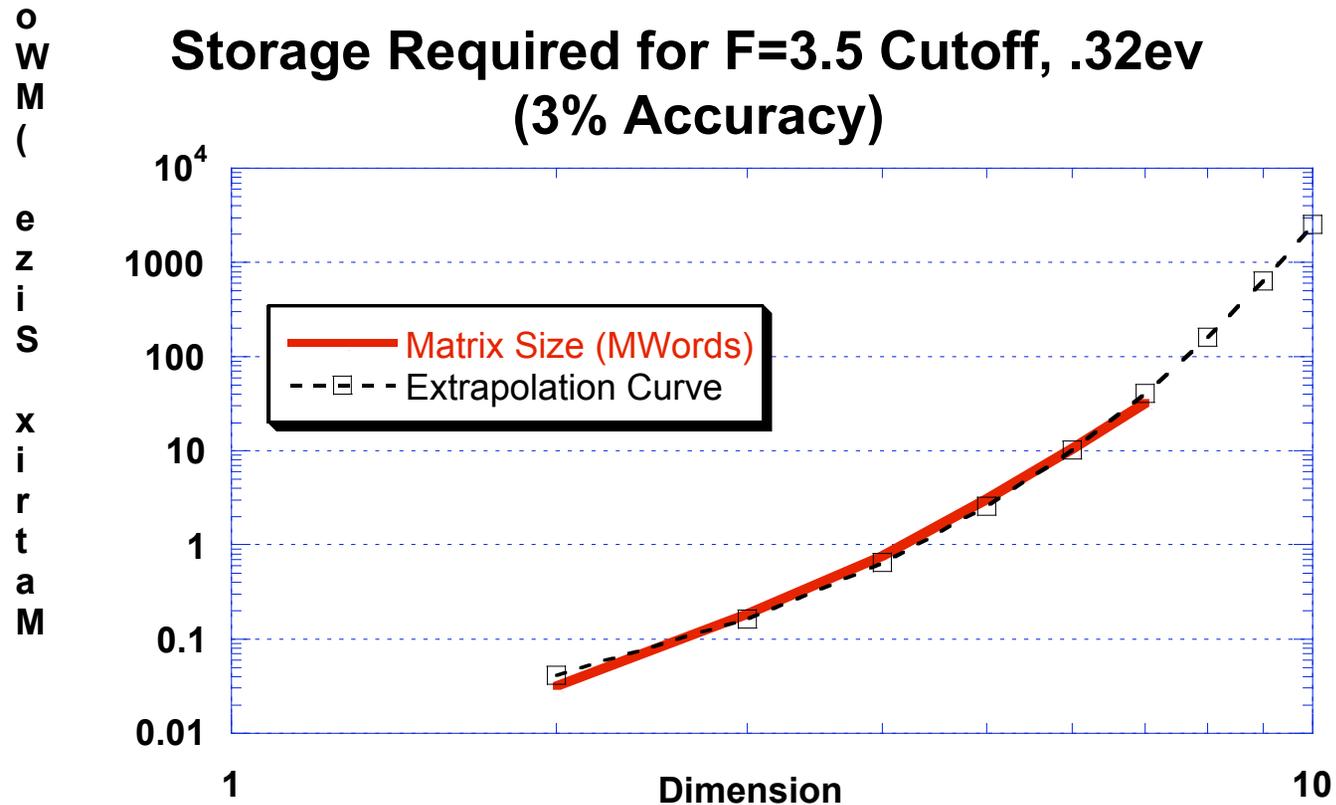
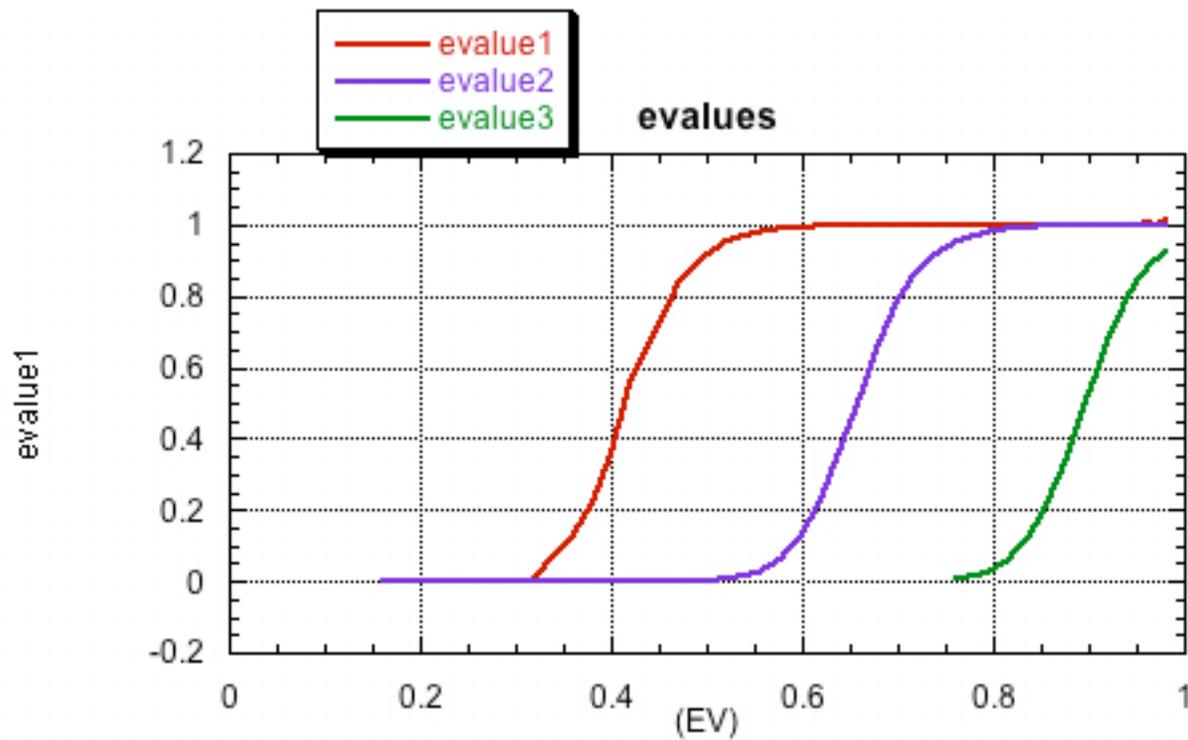


Fig. 1

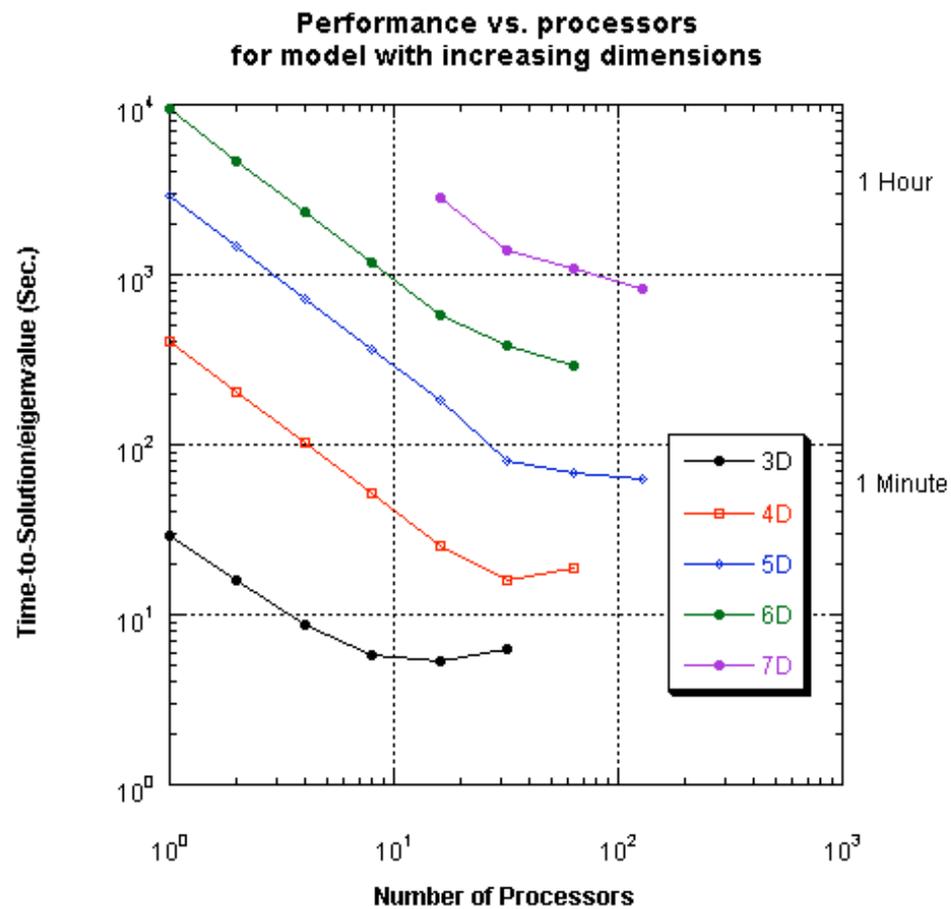
# Non-zero Storage for Green's Matrix



# Eigenvalues vs. Total Energy



# Performance vs. Processors

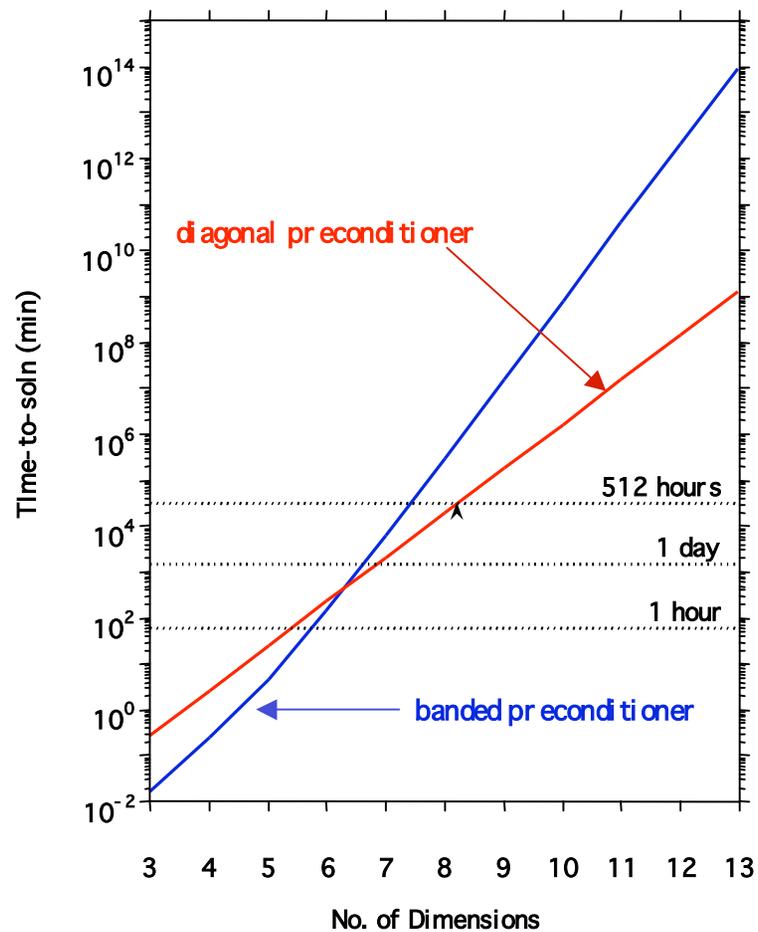


## ***Banded Single-Processor Approach***

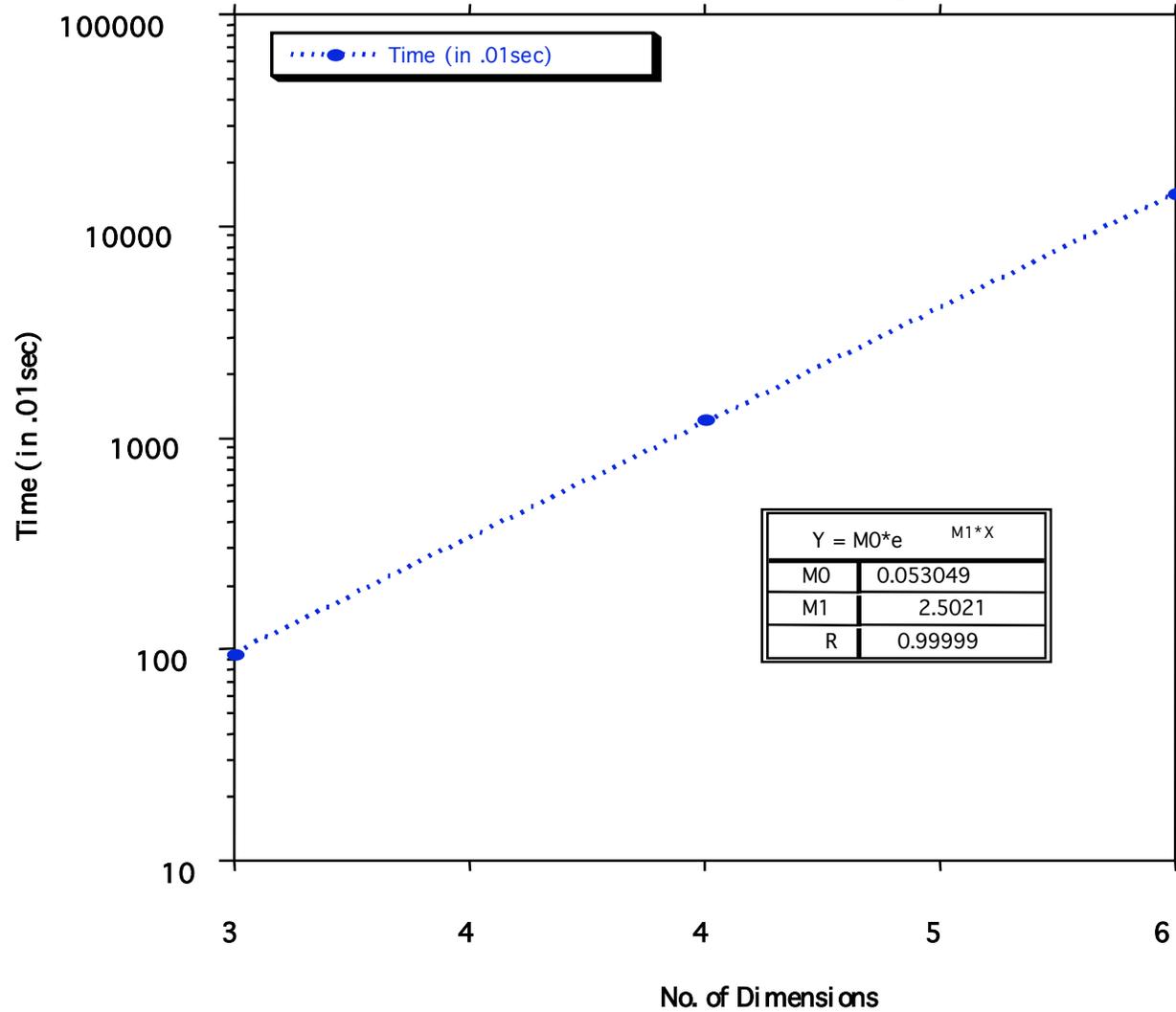
- **Compare diagonal and banded preconditioner in terms of reducing total iteration count and cost**
- **Compare iterative methods (Davidson, GMRES) to benchmark PETSc results**
- **Evaluate relative cost of banded operations with sparse-matrix approach in PETSc**



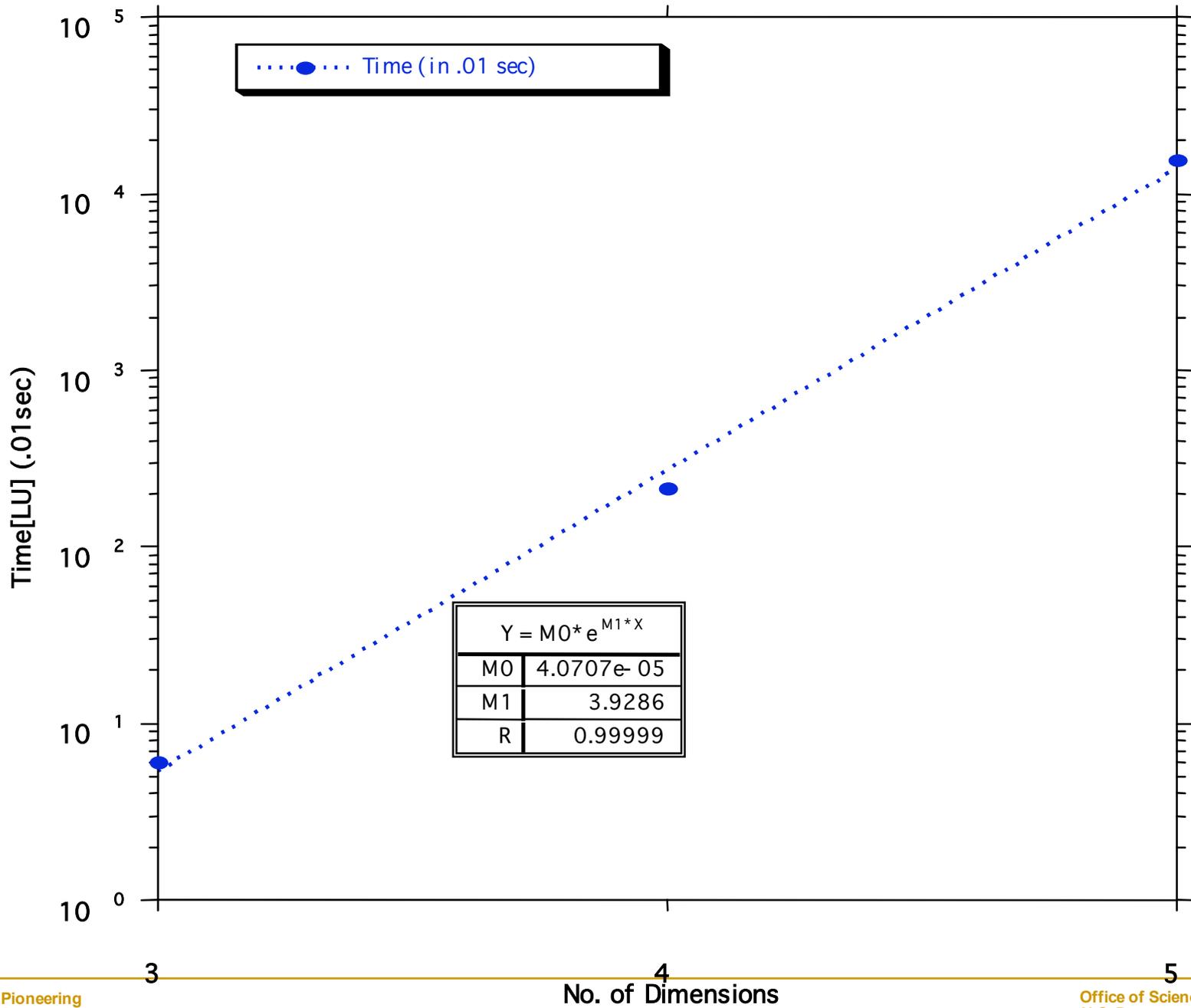
## Diagonal vs. Banded Preconditioner



## Non LU Decomposition



# Time for LU Preconditioner



## ***Results and Future Work***

- **Developing global and block orthogonal (W. Poirier) preconditioning methods**
- **Use SLEPc for Lanczos iteration and tensor products for efficient matrix-vector multiplies**
- **Use NLCF IBM BGL to solve 10 DOF problems**





# *Improving the Performance of Tensor Matrix Vector Product*

*Dinesh Kaushik*  
*Argonne National Laboratory*

**Argonne National Laboratory**



A U.S. Department of Energy  
Office of Science Laboratory  
Operated by The University of Chicago



# Tensor Matrix Vector Product

- Operator comes from the tensor product of a dense matrix with the identity matrix

$$A = A_z \otimes I \otimes I + I \otimes A_y \otimes I + I \otimes I \otimes A_x$$

- $A_x, A_y, A_z$  are one directional operators (dense)
- $v$  and  $w$  are vectors of size  $n^3$

$$w = Av$$



## Two Ways

- **Build the large sparse matrix**
  - Large sparse matrix of size ( $n^3 \times n^3$  for 3D case)
  - Slow memory bandwidth limited performance
- **Just evaluate the action of A on v (without explicitly forming A)**
  - Done as dense matrix-matrix multiplication
  - Very efficient implementation
  - Huge savings in memory

## *Performance Issues for Sparse Matrix Vector Product*

- **Little data reuse**
- **High ratio of load/store to instructions/floating-point ops**
- **Stalling of multiple load/store functional units on the same cache line**
- **Low available memory bandwidth**



# *Sparse Matrix Vector Algorithm: A General Form*

```
for every row, i {  
  fetch ia(i+1)  
  for j = ia(i) to ia(i + 1) { // loop over the non-zeros of the  
    row  
    fetch ja(j), a(j), x1(ja(j)), .....xN(ja(j))  
    do N fmaddd (floating multiply add)  
  }  
  Store y1(i) .....yN(i)  
}
```

# Estimating the Memory Bandwidth Limitation

## Assumptions

- **Perfect Cache (only compulsory misses; no overhead)**
- **No memory latency**
- **Unlimited number of loads and stores per cycle**

## Data Volume (AIJ Format)

$$\begin{aligned} & \mathbf{m * sizeof(int) + N * (m+n) * sizeof(double)} \\ & \quad \text{// ia, N input (size n) and output (size m) vectors} \\ & \mathbf{+ Nnz * (sizeof(int) + sizeof(double))} \\ & \quad \text{// ja, and a arrays} \\ & \mathbf{= 4 * (m+nnz) + 8 * (N * (m+n) + Nnz)} \end{aligned}$$



# Estimating the Memory Bandwidth Limitation (Contd.)

- Number of Floating-Point Multiply Add (fmadd) Ops =  $N \cdot n_z$
- For square matrices,

$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N}\right) * \frac{n}{N_{nz}} + \frac{12}{N}$$

(Since  $N_{nz} \gg n$ , Bytes transferred / fmadd  $\sim 12/N$ )

- Similarly, for Block AIJ (BAIJ) format

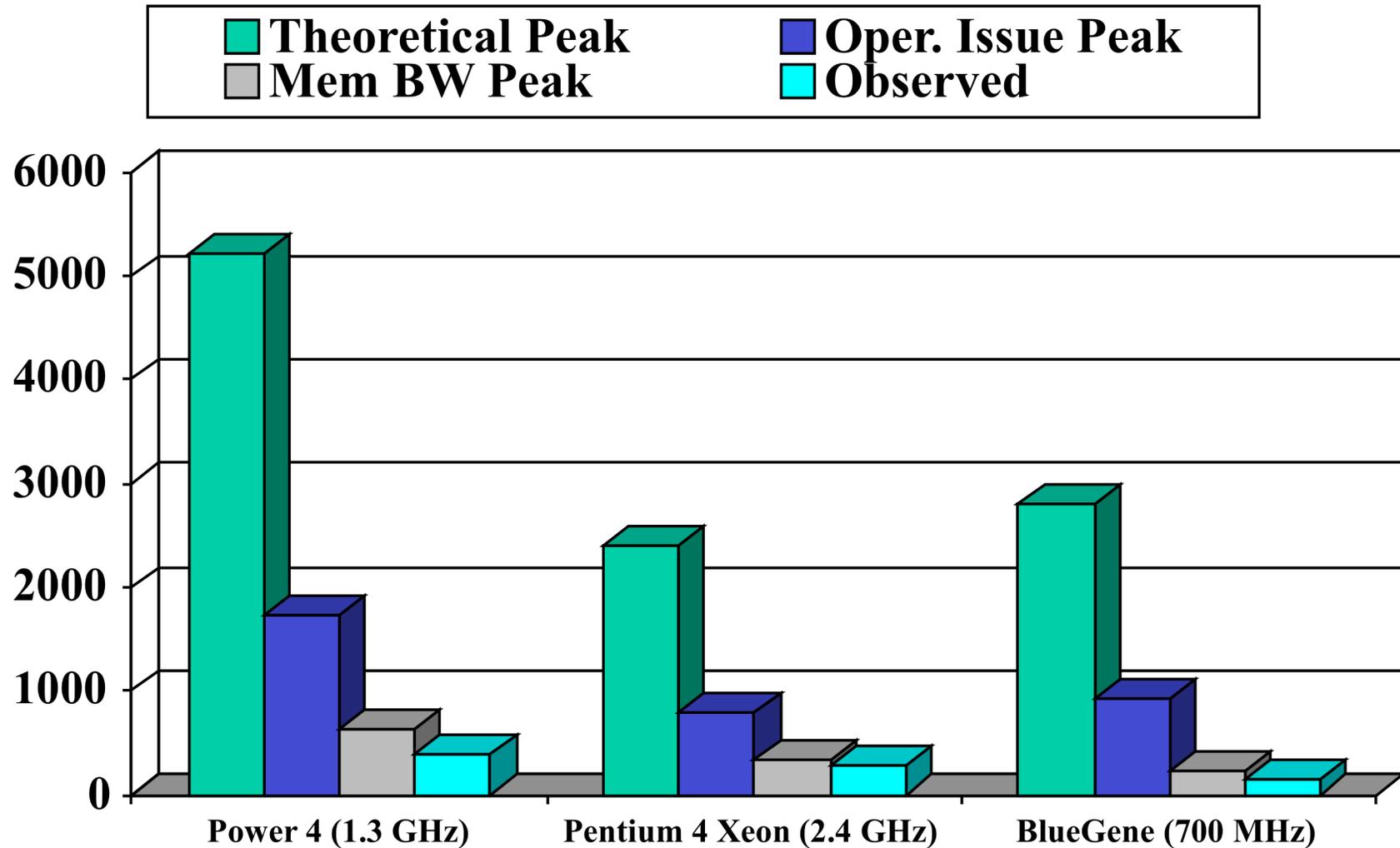
$$\text{Bytes transferred/fmadd} = \left(16 + \frac{4}{N * b}\right) * \frac{n}{N_{nz}} + \left(\frac{4}{N * b} + \frac{8}{N}\right)$$



# Realistic Measures of Peak Performance

Sparse Matrix Vector Product

One vector, matrix size,  $m = 90,708$ , nonzero entries  $nz = 5,047,120$



## ***Second Choice: Dense Matrix-Matrix Multiplication***

- **We just need to store the small dense matrices of size  $n \times n$** 
  - for 3 dimensions memory needed is  $3n^2$
  - Good ratio of flops to bytes:  $O(n^4)$  operations  $O(n^3)$  doubles
  - Gets better for higher dimensions

# Evaluating the Tensor Product Terms

- **Type 1**

$$(I_m \otimes A_n)v_{mn} = [A_n]_{n \times n} [V]_{n \times m}$$

- **Type 2**

$$(A_n \otimes I_m)v_{mn} = [V]_{m \times n} [A_n]_{n \times n}^T$$

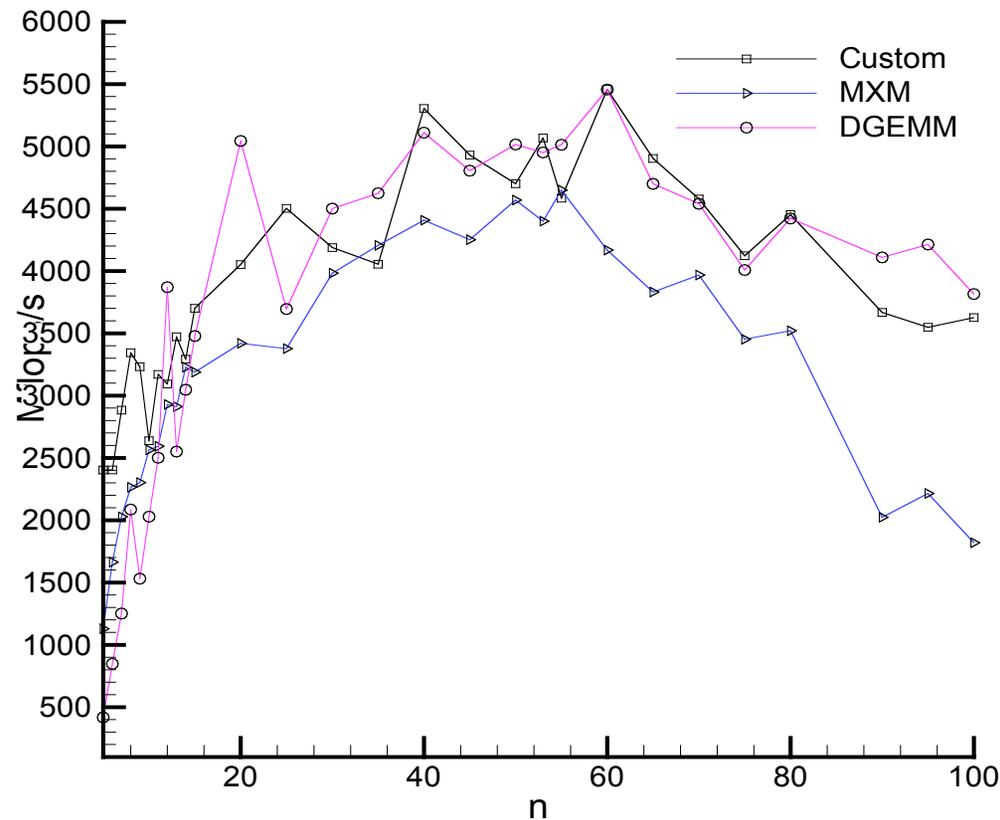
- **Type 3**

$$(I_p \otimes A_n \otimes I_m)v_{mn}$$

- Loop over Type 2 for  $i = 1, p$

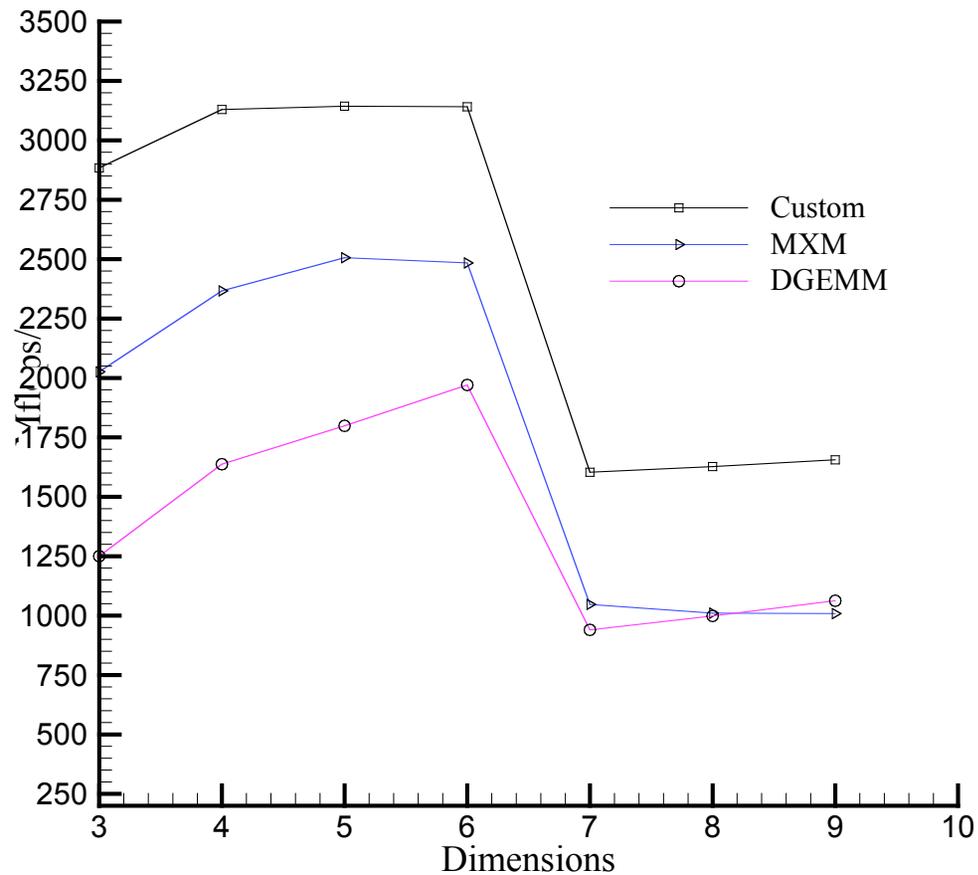
# Performance of Tensor Matrix-Vector Multiplication – 3D case

(Intel Madison Processor 1.5 GHz, 6 Gflops/s Peak, 4 GB Memory)  
Memory Bandwidth Limited Bound 670 Mflops/s



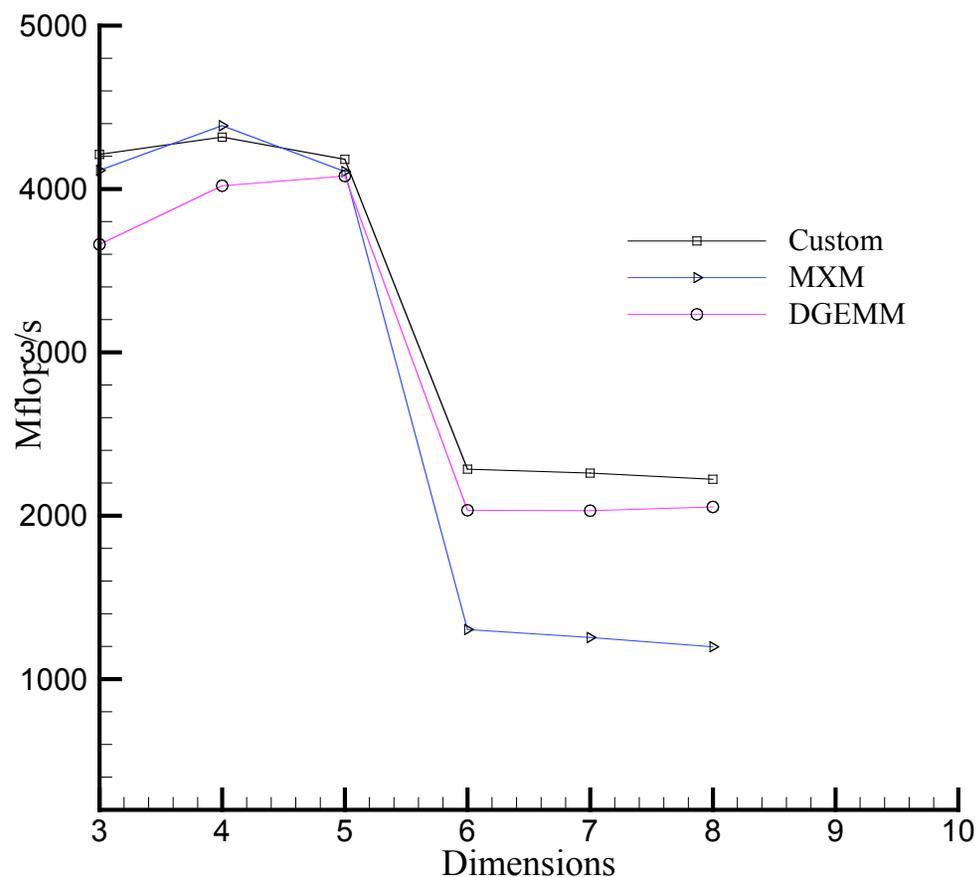
# Performance of Tensor Matrix-Vector Multiplication – Fixed Mesh Points ( $n=7$ )

(Intel Madison Processor 1.5 GHz, 6 Gflops/s Peak, 4 GB Memory)



# Performance of Tensor Matrix-Vector Multiplication –Long Reaction Co-ordinate

(51 points along reaction path and 7 points in other dimensions)



## *Conclusions and Future Work*

- **Very efficient implementation**
  - Sparse matvecs take about 80% of execution time
  - We expect that tensor product implementation can improve the performance by a factor of three to five
- **Possible to solve much larger problems because of huge savings in memory requirement**
- **Parallel implementation**



# *Acknowledgements*

- **Barry Smith, William Gropp, and Paul Fischer for many helpful discussions**

