PARLab Parallel Boot Camp

Computational Patterns
and Autotuning

Jim Demmel
EECS and Mathematics
University of California, Berkeley
• Productive parallel computing depends on recognizing and exploiting useful patterns
  • Computational (7 Motifs) and Structural
• Simplest case: use “best” existing highly tuned implementation
  • Best: Fastest? Most accurate? Fewest keystrokes?
• Optimizing (some of) the 7 Motifs
  • To minimize time or energy, minimize communication (moving data)
    • Between levels of the memory hierarchy
    • Between processors over a network
  • Autotuning to explore large design spaces
    • Too hard (tedious) to write by hand, let machine do it
• SEJITS – how to deliver autotuning to more programmers
• For more details, see
  • CS267: www.cs.berkeley.edu/~demmel/cs267_Spr13
  • 10-hour short course: issnla2010.ba.cnr.it/Courses.htm
  • Papers at bebop.cs.berkeley.edu, parlab.eecs.berkeley.edu
Phil Colella (LBL) identified 7 kernels of which most simulation and data-analysis programs are composed:

1. Dense Linear Algebra
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where $A$ is a dense matrix

2. Sparse Linear Algebra
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where $A$ is a sparse matrix (mostly zero)

3. Operations on Structured Grids
   - Ex: $A_{\text{new}}(i,j) = 4*A(i,j) - A(i-1,j) - A(i+1,j) - A(i,j-1) - A(i,j+1)$

4. Operations on Unstructured Grids
   - Ex: Similar, but list of neighbors varies from entry to entry

5. Spectral Methods
   - Ex: Fast Fourier Transform (FFT)

6. Particle Methods
   - Ex: Compute electrostatic forces on $n$ particles

7. Monte Carlo
   - Ex: Many independent simulations using different inputs
Phil Colella (LBL) identified 7 kernels of which most simulation and data-analysis programs are composed:

1. **Dense Linear Algebra**
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where $A$ is a dense matrix

2. **Sparse Linear Algebra**
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where $A$ is a sparse matrix (mostly zero)

3. **Operations on Structured Grids**
   - Ex: $A_{new}(i,j) = 4*A(i,j) - A(i-1,j) - A(i+1,j) - A(i,j-1) - A(i,j+1)$

4. **Operations on Unstructured Grids**
   - Ex: Similar, but list of neighbors varies from entry to entry

5. **Spectral Methods**
   - Ex: Fast Fourier Transform (FFT)

6. **Particle Methods**
   - Ex: Compute electrostatic forces on n particles

7. **Monte Carlo**
   - Ex: Many independent simulations using different inputs
Organizing Linear Algebra Motifs - in books and on-line

www.netlib.org/lapack

www.netlib.org/scalapack

www.netlib.org/templates

www.cs.utk.edu/~dongarra/etemplates
Why Minimize Communication? (1/2)

- Running time of an algorithm is sum of 3 terms:
  - \# flops * time_per_flop
  - \# words moved / bandwidth
  - \# messages * latency

- Time_per_flop $\ll 1/\text{bandwidth} \ll \text{latency}$

- Gaps growing exponentially with time [FOSC]

- Minimize communication to save time

<table>
<thead>
<tr>
<th>Annual improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time_per_flop</td>
</tr>
<tr>
<td>Network</td>
</tr>
<tr>
<td>DRAM</td>
</tr>
</tbody>
</table>

- Minimize communication to save time
Why Minimize Communication? (2/2)

Source: John Shalf, LBL
Minimize communication to save energy

Source: John Shalf, LBL
“New Algorithm Improves Performance and Accuracy on Extreme-Scale Computing Systems. On modern computer architectures, communication between processors takes longer than the performance of a floating point arithmetic operation by a given processor. ASCR researchers have developed a new method, derived from commonly used linear algebra methods, to minimize communications between processors and the memory hierarchy, by reformulating the communication patterns specified within the algorithm. This method has been implemented in the TRILINOS framework, a highly-regarded suite of software, which provides functionality for researchers around the world to solve large scale, complex multi-physics problems.”
Obstacle to avoiding communication: Low “computational intensity”

• Let $f = \#\text{arithmetic operations in an algorithm}$
• Let $m = \#\text{words of data needed}$
• Def: $q = f/m = \text{computational intensity}$
• If $q$ small, say $q=1$, so one op/word, then algorithm can’t run faster than memory speed
• But if $q$ large, so many ops/word, then algorithm can (potentially) fetch data, do many ops while in fast memory, only limited by (faster!) speed of arithmetic
• We seek algorithms with high $q$
  – Still need to be clever to take advantage of high $q$
DENSE LINEAR ALGEBRA MOTIF
• In the beginning was the do-loop...
  – Libraries like EISPACK (for eigenvalue problems)
• Then the BLAS (1) were invented (1973-1977)
  – Standard library of 15 operations on vectors
    • Ex: $y = \alpha \cdot x + y$ ("AXPY"), dot product, etc
  – Goals
    • Common pattern to ease programming, efficiency, robustness
  – Used in libraries like LINPACK (for linear systems)
    • Source of the name "LINPACK Benchmark" (not the code!)
  – Why BLAS 1? 1 loop, do $O(n^1)$ ops on $O(n^1)$ data
  – Computational intensity $= q = 2n/3n = 2/3$ for AXPY
    • Very low!
  – BLAS1, and so LINPACK, limited by memory speed
  – Need something faster ...
So the BLAS-2 were invented (1984-1986)

- Standard library of 25 operations (mostly) on matrix/vector pairs
  - Ex: \( y = \alpha \cdot A \cdot x + \beta \cdot y \) (“GEMV”), \( A = A + \alpha \cdot x \cdot y^T \) (“GER”), \( y = T^{-1} \cdot x \) (“TRSV”)
  - Why BLAS 2? 2 nested loops, do \( O(n^2) \) ops on \( O(n^2) \) data
  - But \( q = \) computational intensity still just \( \sim (2n^2)/(n^2) = 2 \)
    - Was OK for vector machines, but not for machine with caches, since \( q \) still just a small constant
• The next step: BLAS-3 (1987-1988)
  – Standard library of 9 operations (mostly) on matrix/matrix pairs
    • Ex: $C = \alpha \cdot A \cdot B + \beta \cdot C$ ("GEMM"), $C = \alpha \cdot A \cdot A^T + \beta \cdot C$ ("SYRK"), $C = T^{-1} \cdot B$ ("TRSM")
  – Why BLAS 3? 3 nested loops, do $O(n^3)$ ops on $O(n^2)$ data
  – So computational intensity $q=(2n^3)/(4n^2) = n/2$ – big at last!
    • Tuning opportunities machines with caches, other mem. hierarchy levels
• How much faster can BLAS 3 go?
Matrix-multiply, optimized several ways

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

Optimized Implementations:
- Vendor (Sun) and Autotuned (PHiPAC)

Reference Implementation; Full compiler opt.
Faster Matmul $C=A*B$ by "Blocking"

- Replace usual 3 nested loops ...

\[
\begin{align*}
&\text{for } i=1 \text{ to } n \\
&\quad \text{for } j=1 \text{ to } n \\
&\quad \quad \text{for } k=1 \text{ to } n \\
&\quad \quad \quad C(i,j) = C(i,j) + A(i,k)B(k,j)
\end{align*}
\]
Lower bounds on Communication for Matmul

- Assume sequential $n^3$ algorithm for $C = A \cdot B$
  - i.e. not Strassen-like
- Assume $A$, $B$ and $C$ fit in slow memory, but not in fast memory of size $M$
- Thm: Lower bound on #words_moved to/from slow memory, no matter the order $n^3$ operations are done, $= \Omega \left( \frac{n^3}{M^{1/2}} \right)$ [Hong & Kung (1981)]
- Attained by “blocked” algorithm
  - Some other algorithms attain it too
  - Widely implemented in libraries (e.g., Intel MKL)
How hard is hand-tuning, anyway?

Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09
Students given “blocked” code to start with
Still hard to get close to vendor tuned performance (ACML)
For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/
Naïve matmul: just 2% of peak
How hard is hand-tuning, anyway?
What part of the Matmul Search Space Looks Like

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-Ill, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)
Autotuning DGEMM with ATLAS (n = 500)

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
- ATLAS written by C. Whaley, inspired by PHIPAC, by Asanovic, Bilmes, Chin, D.
LAPACK – “Linear Algebra PACKage” - uses BLAS-3 (1989 – now)
   - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of each row to other rows – BLAS-1
   - Need to reorganize GE (and everything else) to use BLAS-3 instead

Contents of current LAPACK (summary)
   - Algorithms we can turn into (nearly) 100% BLAS 3 for large n
     - Linear Systems: solve $Ax=b$ for $x$
     - Least Squares: choose $x$ to minimize $\sqrt{\sum_i r_i^2}$ where $r=Ax-b$
   - Algorithms that are only up to ~50% BLAS 3, rest BLAS 1 & 2
     - “Eigenproblems”: Find $\lambda$ and $x$ where $Ax = \lambda x$
     - Singular Value Decomposition (SVD): $A^T Ax = \sigma^2 x$
   - Error bounds for everything
   - Lots of variants depending on A’s structure (banded, $A=A^T$, etc)

Widely used (list later)

All at www.netlib.org/lapack
• Is LAPACK parallel?
  – Only if the BLAS are parallel (possible in shared memory)

• ScaLAPACK – “Scalable LAPACK” (1995 – now)
  – For distributed memory – uses MPI
  – More complex data structures, algorithms than LAPACK
    • Only subset of LAPACK’s functionality available
    • Work in progress (contributions welcome!)
  – All at www.netlib.org/scalapack
Success Stories for Sca/LAPACK

• Widely used
  – Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, ...
  – >172M web hits (in 2013, 157M in 2012)
    @ Netlib (incl. CLAPACK, LAPACK95)

• New science discovered through the solution of dense matrix systems
  – Nature article on the flat universe used ScaLAPACK
  – 1998 Gordon Bell Prize
  – [Link](www.nersc.gov/news/reports/newNERSResults050703.pdf)

• Currently funded to improve, update, maintain Sca/LAPACK
Lower bound for all “$n^3$-like” linear algebra

- Let $M$ = “fast” memory size (per processor)

  $\#\text{words\_moved (per processor)} = \Omega(\#\text{flops (per processor)} / M^{1/2})$

- Parallel case: assume either load or memory balanced
  - Holds for
    - Matmul
Lower bound for all “$n^3$-like” linear algebra

- Let $M$ = “fast” memory size (per processor)

  $$\#\text{words\_moved} (\text{per processor}) = \Omega(\#\text{flops} (\text{per processor}) / M^{1/2})$$

  $$\#\text{messages\_sent} \geq \#\text{words\_moved} / \text{largest\_message\_size}$$

- Parallel case: assume either load or memory balanced

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, ...
  - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg $A^k$)
  - Dense and sparse matrices (where $\#\text{flops} \ll n^3$)
  - Sequential and parallel algorithms
  - Some graph-theoretic algorithms (eg Floyd-Warshall)
Lower bound for all “$n^3$-like” linear algebra

- Let $M$ = “fast” memory size (per processor)

  \[
  \text{#words\_moved (per processor)} = \Omega\left(\frac{\text{#flops (per processor)}}{M^{1/2}}\right)
  \]

  \[
  \text{#messages\_sent (per processor)} = \Omega\left(\frac{\text{#flops (per processor)}}{M^{3/2}}\right)
  \]

- Parallel case: assume either load or memory balanced

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, ...
  - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg $A^k$)
  - Dense and sparse matrices (where #flops $< n^3$)
  - Sequential and parallel algorithms
  - Some graph-theoretic algorithms (eg Floyd-Warshall)
Can we attain these lower bounds?

- Do conventional dense algorithms as implemented in LAPACK and ScaLAPACK attain these bounds?
  - Mostly not

- If not, are there other algorithms that do?
  - Yes, for much of dense linear algebra
  - New algorithms, with new numerical properties, new ways to encode answers, new data structures
  - Not just loop transformations (need those too!)

- Only a few sparse algorithms so far
- Lots of work in progress
Example: “2.5D” Matrix multiply

Lower bound decreases as M increases, even beyond minimum needed ($3n^2/p$)
2.5D Matrix Multiply Timing Breakdown

$c = 16$ copies

Matrix multiplication on 16,384 nodes of BG/P

95% reduction in comm

Execution time normalized by 2D

Distinguished Paper Award, EuroPar’11 (Solomonik, D.)

(SC’11 paper by Solomonik, Bhatele, D.)
TSQR: QR of a Tall, Skinny matrix

\[ W = \begin{bmatrix}
W_0 \\
W_1 \\
W_2 \\
W_3 \\
\end{bmatrix} \]

\[
\begin{bmatrix}
R_{00} \\
R_{10} \\
R_{20} \\
R_{30} \\
\end{bmatrix} = \begin{bmatrix}
Q_{01} & R_{01} \\
Q_{11} & R_{11} \\
\end{bmatrix} \]

\[
\begin{bmatrix}
R_{01} \\
R_{11} \\
\end{bmatrix} = \begin{bmatrix}
Q_{02} & R_{02} \\
\end{bmatrix} \]
TSQR: QR of a Tall, Skinny matrix

\[
W = \begin{pmatrix}
W_0 \\
W_1 \\
W_2 \\
W_3
\end{pmatrix} = \begin{pmatrix}
Q_{00} & R_{00} \\
Q_{10} & R_{10} \\
Q_{20} & R_{20} \\
Q_{30} & R_{30}
\end{pmatrix} = \begin{pmatrix}
Q_{00} & \\
Q_{10} & \\
Q_{20} & \\
Q_{30} & 
\end{pmatrix} \cdot \begin{pmatrix}
R_{00} \\
R_{10} \\
R_{20} \\
R_{30}
\end{pmatrix}
\]

\[
\begin{pmatrix}
R_{00} \\
R_{10} \\
R_{20} \\
R_{30}
\end{pmatrix} = \begin{pmatrix}
Q_{01} & R_{01} \\
Q_{11} & R_{11}
\end{pmatrix} = \begin{pmatrix}
Q_{01} & \\
Q_{11} & 
\end{pmatrix} \cdot \begin{pmatrix}
R_{01} \\
R_{11}
\end{pmatrix}
\]

\[
\begin{pmatrix}
R_{01} \\
R_{11}
\end{pmatrix} = \begin{pmatrix}
Q_{02} & R_{02}
\end{pmatrix}
\]

Output = \{ Q_{00}, Q_{10}, Q_{20}, Q_{30}, Q_{01}, Q_{11}, Q_{02}, R_{02} \}
TSQR: An Architecture-Dependent Algorithm

Parallel: \( W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} \rightarrow R_{00} \rightarrow R_{10} \rightarrow R_{01} \rightarrow R_{02} \)

Sequential: \( W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} \rightarrow R_{00} \rightarrow R_{01} \rightarrow R_{02} \rightarrow R_{03} \)

Dual Core: \( W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} \rightarrow R_{00} \rightarrow R_{01} \rightarrow R_{01} \rightarrow R_{02} \rightarrow R_{03} \)

Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

Can choose reduction tree dynamically
TSQR Performance Results

• Parallel
  – Intel Clovertown
    – Up to 8x speedup (8 core, dual socket, 10M x 10)
  – Pentium III cluster, Dolphin Interconnect, MPICH
    • Up to 6.7x speedup (16 procs, 100K x 200)
  – BlueGene/L
    • Up to 4x speedup (32 procs, 1M x 50)
  – Tesla C 2050 / Fermi
    • Up to 13x (110,592 x 100)
  – Grid – 4x on 4 cities vs 1 city (Dongarra, Langou et al)
  – Cloud (Gleich, Benson) – cost ~ 2 MapReduce operations, orders of magnitude faster

• Sequential
  – “Infinite speedup” for out-of-Core on PowerPC laptop
    • As little as 2x slowdown vs (predicted) infinite DRAM
    • LAPACK with virtual memory never finished

• Joint work with Grigori, Hoemmen, Langou, Anderson, Ballard, Keutzer, others
• Communication-Avoiding for everything (open problems...)
  – Perfect strong scaling for time and energy
  – Extensions to Strassen-like algorithms

• Extensions for multicore
  – PLASMA – Parallel Linear Algebra for Scalable Multicore Architectures
    • Dynamically schedule tasks into which algorithm is decomposed, to minimize synchronization, keep all processors busy
    • Release 2.5.1 at icl.cs.utk.edu/plasma/

• Extensions for heterogeneous architectures, eg CPU + GPU
  – “Benchmarking GPUs to tune Dense Linear Algebra”
    • Best Student Paper Prize at SC08 (Vasily Volkov)
    • Paper, slides and code at www.cs.berkeley.edu/~volkov
  – Lower, matching upper bounds (tech report at bebop.cs.berkeley.edu)
  – MAGMA – Matrix Algebra on GPU and Multicore Architectures
    • Release 1.4 at icl.cs.utk.edu/magma/

• How much code generation can we automate?
  – MAGMA, and FLAME (www.cs.utexas.edu/users/flame/)
Beyond Dense Linear Algebra

• Recall matmul: \#words\_moved = Ω (\#flops/M^{1/2}), attained by “block algorithm” with \( M^{1/2} \times M^{1/2} \) blocks
  – Where do all the 1/2’s come from?

• “Thm”: Lower bound extends to any algorithm that
  – “smells like” nested loops over \( i_1, i_2, i_3, \ldots, i_k \)
  – Accesses arrays like \( A(i_1, i_2+3*i_3-2*i_4, i_3+4*i_5, \ldots), B(pntr(i_2), \ldots) \)
  – Lower bound becomes \( Ω (\#loop\_iteration/M^S) \)
  – Can write down optimal algorithms in many cases of interest
    • Linear Algebra, N-body, data-base-join, ...
    • Open problem: Does an algorithm attaining lower bound always exist?
SPARSE LINEAR ALGEBRA MOTIF
Sparse Matrix Computations

• Similar problems to dense matrices
  – Ax=b, Least squares, Ax = λx, SVD, ...

• But different algorithms!
  – Exploit structure: only store, work on nonzeros
  – Direct methods
    • LU, Cholesky for Ax=b, QR for Least squares
    • See crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf for a survey of available serial and parallel sparse solvers
    • See crd.lbl.gov/~xiaoye/SuperLU/index.html for LU codes
  – Iterative methods – for Ax=b, least squares, eig, SVD
    • Use simplest operation: Sparse-Matrix-Vector-Multiply (SpMV)
    • Krylov Subspace Methods: find “best” solution in space spanned by vectors generated by SpMVs
Choosing a Krylov Subspace Method for $Ax=b$

- A symmetric?
  - $A^T$ available?
    - No
      - Storage expensive?
        - No: Try GMRES
        - Yes: Try CGS, BiCGStab, or GMRES(k)
    - Yes: A definite?
      - No
        - A well-conditioned?
          - No: Try QMR
          - Yes: Try CG on normal eqns.
      - Yes: Largest/smallest eigenvalues known?
        - No: Try CG
        - Yes: Try CG with Chebyshev acceleration

- A definite?

- All depend on SpMV
- See [www.netlib.org/templates](http://www.netlib.org/templates) for $Ax=b$
- See [www.cs.ucdavis.edu/~bai/ET/contents.html](http://www.cs.ucdavis.edu/~bai/ET/contents.html) for $Ax=\lambda x$ and SVD
Sparse Outline

• Approaches to Automatic Performance Tuning
• Results for sparse matrix kernels
  – Sparse Matrix Vector Multiplication (SpMV)
  – Sequential and Multicore results
• OSKI = Optimized Sparse Kernel Interface
  – pOSKI = parallel OSKI
• Tuning Entire Sparse Solvers
  – Avoiding Communication
• What is a sparse matrix?
Approaches to Automatic Performance Tuning

• Goal: Let machine do hard work of writing fast code

• Why is tuning dense BLAS “easy”?  
  – Can do the tuning off-line: once per architecture, algorithm  
  – Can take as much time as necessary (hours, a week...)  
  – At run-time, algorithm choice may depend only on few parameters (matrix dimensions)

• Can’t always do tuning off-line  
  – Algorithm and implementation may strongly depend on data only known at run-time  
  – Ex: Sparse matrix nonzero pattern determines both best data structure and implementation of Sparse-matrix-vector-multiplication (SpMV)  
  – Part of search for best algorithm must be done (very quickly!) at run-time

• Tuning FFTs and signal processing  
  – Seems off-line, but maybe not, because of code size  
Source: Accelerator Cavity Design Problem (Ko via Husbands)
<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>Data 1</td>
<td>Data 2</td>
<td>Data 3</td>
</tr>
<tr>
<td>Row 2</td>
<td>Data 4</td>
<td>Data 5</td>
<td>Data 6</td>
</tr>
<tr>
<td>Row 3</td>
<td>Data 7</td>
<td>Data 8</td>
<td>Data 9</td>
</tr>
</tbody>
</table>
A Sparse Matrix You Use Every Day
Matrix-vector multiply kernel: \( y(i) \leftarrow y(i) + A(i,j) \cdot x(j) \)

for each row \( i \)

\[
\text{for } k = \operatorname{ptr}[i] \text{ to } \operatorname{ptr}[i+1] \text{ do}
\]
\[
y[i] = y[i] + \operatorname{val}[k] \cdot x[\operatorname{ind}[k]]
\]

Only 2 flops per 2 memRefs: Limited by getting data from memory
Example: The Difficulty of Tuning

- $n = 21200$
- $\text{nnz} = 1.5 \text{ M}$
- Kernel: SpMV
- Source: NASA structural analysis problem
Example: The Difficulty of Tuning

- \(n = 21200\)
- \(\text{nnz} = 1.5 \text{ M}\)
- Kernel: SpMV

- Source: NASA structural analysis problem

- 8x8 dense substructure: exploit this to limit \#\text{mem_refs}
### Speedups on Itanium 2: The Need for Search

The performance results for Matrix #02-raefsky3.rua on Itanium 2 (900 MHz) are shown in the table below, with a reference performance of 274.3 Mflop/s.

<table>
<thead>
<tr>
<th>Row Block Size (r)</th>
<th>Column Block Size (c)</th>
<th>Mflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.39</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.55</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.24</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3.35</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.09</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.16</td>
</tr>
</tbody>
</table>

Best performance: 4x2

Reference: 1.00 Mflop/s
Register Profile: Itanium 2

SpMV BCSR Profile [ref=294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]

1190 Mflop/s

190 Mflop/s
Register Profiles: Sun and Intel x86

Ultra 2i - 11%
- 72 Mflop/s
- 35 Mflop/s
- 108 Mflop/s
- 42 Mflop/s
- 122 Mflop/s

Ultra 3 - 5%
- 90 Mflop/s
- 50 Mflop/s

Pentium III - 21%
- 42 Mflop/s

Pentium III-M - 15%
- 108 Mflop/s

Profile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]
Profile [ref=50.3 Mflop/s; 500 MHz Sun Ultra 3, Sun C v6.0]
Profile [ref=80 Mflop/s; 800 MHz Pentium III-M, Intel C v7.0]
Another example of tuning challenges

- More complicated non-zero structure in general

- $N = 16614$
- $NNZ = 1.1M$
• More complicated non-zero structure in general

• $N = 16614$

• $\text{NNZ} = 1.1M$
3x3 blocks look natural, but...

- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
- But would lead to lots of “fill-in”
Extra Work Can Improve Efficiency!

- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
  - Fill-in explicit zeros
  - Unroll 3x3 block multiplies
  - “Fill ratio” = 1.5

(688 true non-zeros) + (383 explicit zeros) = 1071 nz
Selecting Register Block Size $r \times c$

- **Off-line benchmark**
  - Precompute $\text{Mflops}(r,c)$ using dense $A$ for each $r \times c$
  - Once per machine/architecture

- **Run-time “search”**
  - Sample $A$ to estimate $\text{Fill}(r,c)$ for each $r \times c$
  - Control cost $= O(s \cdot \text{nnz})$ by controlling fraction $s \in [0, 1]$ sampled
  - Control $s$ automatically by computing statistical confidence intervals, by monitoring variance

- **Run-time heuristic model**
  - Choose $r, c$ to minimize $\text{time} \sim \text{Fill}(r,c) / \text{Mflops}(r,c)$

- **Cost of tuning**
  - Lower bound: convert matrix in 5 to 40 unblocked SpMVs
  - Heuristic: 1 to 11 SpMVs

- **Tuning only useful when we do many SpMVs**
  - Common case, eg in sparse solvers
Accuracy of the Tuning Heuristics [Itanium 2]

NOTE: “Fair” flops used (ops on explicit zeros not counted as “work”)

See p. 375 of Vuduc’s thesis for matrices.
Accuracy of the Tuning Heuristics [Itanium 2]

DGEMV

Performance (Mflop/s)

Matrix No.

FEM

FEM (var)

Other

LP
Performance Bounds on Register Blocked SpMV [Itanium 2]

Upper bound counts only compulsory memory traffic

PAPI upper bound counts true traffic

Performance (Mflop/s)

Fraction of machine peak

matrix no.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

D FEM FEM (var) Other LP

0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500

0 0.0556 0.0833 0.1111 0.1389 0.1667 0.1944 0.2222 0.25 0.2778 0.3056 0.3333 0.3611 0.3889 0.4167
Summary of Other Sequential Performance Optimizations

• Optimizations for SpMV
  – **Register blocking (RB):** up to *4x* over CSR
  – **Variable block splitting:** *2.1x* over CSR, *1.8x* over RB
  – **Diagonals:** *2x* over CSR
  – **Reordering** to create dense structure + **splitting:** *2x* over CSR
  – **Symmetry:** *2.8x* over CSR, 2.6x over RB
  – **Cache blocking:** *2.8x* over CSR
  – **Multiple vectors (SpMM):** *7x* over CSR
  – And combinations...

• Sparse triangular solve
  – Hybrid sparse/dense data structure: *1.8x* over CSR

• Higher-level kernels
  – **A·Aᵀ·x, Aᵀ·A·x:** *4x* over CSR, 1.8x over RB
  – **A²·x:** *2x* over CSR, 1.5x over RB
  – **[A·x, A²·x, A³·x, .., A^k·x]** .... more to say later
Can we reorder the rows and columns to create dense blocks, to accelerate SpMV?
Post-RCM (Breadth-first-search) Reordering

Moving nonzeros nearer the diagonal should create dense block, but let’s zoom in and see…
Here is the top 100x100 submatrix before RCM
“Microscopic” Effect of RCM Reordering

Here is the top 100x100 submatrix after RCM: red entries move to the blue locations. More dense blocks, but could be better, so let’s try reordering again, using TSP (Travelling Salesman Problem)
“Microscopic” Effect of Combined RCM+TSP Reordering

**Before:** Green + Red
**After:** Green + Blue

Here is the top 100x100 submatrix after RCM and TSP: red entries move to the blue locations. Lots of dense blocks, as desired!

Speedups (using symmetry too):

- Itanium 2: 1.7x
- Pentium 4: 2.1x
- Power 4: 2.1x
- Ultra 3: 3.3x
Multicore SMPs Used for Tuning SpMV

**Intel Xeon E5345 (Clovertown)**
- 4MB L2 Cache
- 4 MB Shared L2 (16 way) (64b interleaved)
- 4 Coherency Hubs
- 2x128b controllers
- MCH (4x64b controllers)
- FSB 10.66 GB/s
- 2 MB victim
- 8 x 667 MHz FBDIMMs
- 8 x 667 MHz DDR2 DIMMs
- 10.66 GB/s

**AMD Opteron 2356 (Barcelona)**
- HyperTransport
- 4 Gb/s
- 2 x 64b controllers
- 667 MHz DDR2 DIMMs
- 667 MHz DDR2 DIMMs
- 2 MB victim
- SRI/xbar
- 10.66 GB/s

**Sun T2+ T5140 (Victoria Falls)**
- MT SPARC
- MT SPARC
- MT SPARC
- MT SPARC
- MT SPARC
- MT SPARC
- 179 GB/s
- 90 GB/s
- 8 x 6.4 GB/s (peak)
- 4 Coherency Hubs
- 2x128b controllers
- 21.33 GB/s
- 10.66 GB/s
- 667 MHz FBDIMMs

**IBM QS20 Cell Blade**
- VMT PPE
- MFC 256K
- SPE 256K
- EIB (ring network)
- 512 MB XDR DRAM
- XDR memory controllers
- 25.6 GB/s
- 512K L2
- 512K L2
### Multicore SMPs Used for Tuning SpMV

<table>
<thead>
<tr>
<th>Multicore SMPs</th>
<th>Processor Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5345 (Clovertown)</td>
<td>Cache based, 8 Threads, 75 GFlops, 21/10 GB/s R/W BW</td>
</tr>
<tr>
<td>AMD Opteron 2356 (Barcelona)</td>
<td>Cache based, 8 Threads, NUMA, 74 GFlops, 21 GB/s R/W BW</td>
</tr>
<tr>
<td>Sun T2+ T5140 (Victoria Falls)</td>
<td>Cache based, 128 Threads (CMT), NUMA, 19 GFlops, 42/21 GB/s R/W BW</td>
</tr>
<tr>
<td>IBM QS20 Cell Blade</td>
<td>Local-Store based, 16 Threads, NUMA, 29 Gflops (SPEs only), 51 GB/s R/W BW</td>
</tr>
</tbody>
</table>
Set of 14 test matrices

- All bigger than the caches of our SMPs

2K x 2K Dense matrix stored in sparse format

Well Structured (sorted by nonzeros/row)
- Protein
- FEM / Spheres
- FEM / Cantilever
- Wind Tunnel
- FEM / Harbor
- QCD
- FEM / Ship
- Economics
- Epidemiology

Poorly Structured hodgepodge
- FEM / Accelerator
- Circuit
- webbase

Extreme Aspect Ratio (linear programming)
- LP
SpMV Performance: Naive parallelization

- Out-of-the-box SpMV performance on a suite of 14 matrices
- Scalability isn’t great: Compare to # threads
  
<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5345 (Clovertown)</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>Opteron 2356 (Barcelona)</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>UltraSparc T2+ T5140 (Victoria Falls)</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>QS20 Cell Blade (PPEs)</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
- NUMA-aware allocation is essential on NUMA SMPs.
- Explicit software prefetching can boost bandwidth and change cache replacement policies
- used **exhaustive** search
SpMV Performance: “Matrix Compression”

- Compression includes
  - register blocking
  - other formats
  - smaller indices
- Use **heuristic** rather than search
SpMV Performance: cache and TLB blocking

Xeon E5345 (Clovertown)

Opteron 2356 (Barcelona)

UltraSparc T2+ T5140 (Victoria Falls)

QS20 Cell Blade (PPEs)

- +Cache/LS/TLB Blocking
- +Matrix Compression
- +SW Prefetching
- +NUMA/Affinity
- Naïve Pthreads
- Naïve
SpMV Performance: Architecture specific optimizations

- Xeon E5345 (Clovertown)
- Opteron 2356 (Barcelona)
- UltraSparc T2+ T5140 (Victoria Falls)
- QS20 Cell Blade (SPEs)

Graphs showing performance (GFLOP/s) for different benchmarks and optimizations:
- Cache/LS/TLB Blocking
- Register Block
- Prefetch
- NUMA
- Parallel
- Naïve Pthreads
- Naïve

Benchmark categories include:
- Dense
- Protein
- FEM-Sphr
- FEM-Cant
- Tunnel
- QCD
- Econ
- Epidem
- FEM-Circuit
- Webbase
- LP

The graphs display performance variations across different architectures and optimizations.
SpMV Performance: max speedup

• Fully auto-tuned SpMV performance across the suite of matrices
• Included SPE/local store optimized version
• Why do some optimizations work better on some architectures?

- Cache/LS/TLB Blocking
- Matrix Compression
- SW Prefetching
- NUMA/Affinity
- Naïve Pthreads
- Naïve

<table>
<thead>
<tr>
<th>System</th>
<th>SpMV Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5345 (Clovertown)</td>
<td>2.7x</td>
</tr>
<tr>
<td>Opteron 2356 (Barcelona)</td>
<td>4.0x</td>
</tr>
<tr>
<td>UltraSparc T2+ T5140</td>
<td>2.9x</td>
</tr>
<tr>
<td>QS20 Cell Blade (SPEs)</td>
<td>35x</td>
</tr>
</tbody>
</table>
Optimized Sparse Kernel Interface - pOSKI

• Provides sparse kernels automatically tuned for matrix & machine
  – BLAS-style functionality: SpMV, $Ax$ & $A^Ty$
  – Hides complexity of run-time tuning

• Faster than previous implementations
  – Up to 7.8x over reference serial implementation on Sandy Bridge E
  – Up to 4.5x over OSKI on Sandy Bridge E
  – Up to 2.1x over MKL on Nehalem

• bebop.cs.berkeley.edu/poski

• Ongoing work by the Berkeley Benchmarking and Optimization (BeBop) group
Optimizations in pOSKI, so far

- Fully automatic heuristics for
  - Sparse matrix-vector multiply ($Ax, A^T x$)
    - Register-level blocking, Thread-level blocking
    - SIMD, software prefetching, software pipelining, loop unrolling
    - NUMA-aware allocations

- “Plug-in” extensibility
  - Very advanced users may write their own heuristics, create new data structures/code variants and dynamically add them to the system

- Other kernels just in OSKI so far
  - $A^T A x, A^k x$
  - $A^{-1} x$: Sparse triangular solver (SpTS)

- Other optimizations under development
  - Cache-level blocking, Reordering (RCM, TSP), variable block structure, index compressing, Symmetric storage, etc.
How pOSKI Tunes (Overview)

1. Build for Target Arch.
2. Benchmark
   - Sample Dense Matrix
   - Generated Code Variants
     \((r,c,d,imp,\ldots)\)
   - Benchmark Data & Selected Code Variants
     \((r,c)\)

Library Install-Time (offline) → Application Run-Time

1. Partition
   - User’s Matrix
   - User’s hints
   - Workload from program monitoring

2. Evaluate Models
3. Select Data Struct. & Code
   - History

\((r,c)\) = Register Block size
\((d)\) = prefetching distance
\((imp)\) = SIMD implementation

To user: Matrix handle for kernel calls
How pOSKI Tunes (Overview)

- **At library build/install-time**
  - Generate code variants
    - Code generator (Python) generates code variants for various implementations
  - Collect benchmark data
    - Measures and records speed of possible sparse data structure and code variants on target architecture
  - Select best code variants & benchmark data
    - prefetching distance, SIMD implementation
  - Installation process uses standard, portable GNU AutoTools

- **At run-time**
  - Library “tunes” using heuristic models
    - Models analyze user’s matrix & benchmark data to choose optimized data structure and code
    - User may re-collect benchmark data with user’s sparse matrix (under development)
  - Non-trivial tuning cost: up to ~40 mat-vecs
    - Library limits the time it spends tuning based on estimated workload
      - provided by user or inferred by library
    - User may reduce cost by saving tuning results for application on future runs with same or similar matrix (under development)
How to call pOSKI: Basic Usage

• May gradually migrate existing apps
  – Step 1: “Wrap” existing data structures
  – Step 2: Make BLAS-like kernel calls

```c
int* ptr = ..., *ind = ...;  double* val = ...; /* Matrix, in CSR format */
double* x = ..., *y = ...; /* Let x and y be two dense vectors */

/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
    my_matmult( ptr, ind, val, α, x, β, y );
```
• May gradually migrate existing apps
  – Step 1: “Wrap” existing data structures
  – Step 2: Make BLAS-like kernel calls

```c
int* ptr = ..., *ind = ...;  double* val = ...; /* Matrix, in CSR format */
double* x = ..., *y = ...; /* Let x and y be two dense vectors */

// Step 1: Create a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();

// Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
                                           nnz, SHARE_INPUTMAT, poski_thread, NULL, ...);
poski_vec_t x_view = poski_CreateVec(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVec(y, nrows, UNIT_STRIDE, NULL);

// Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
    my_matmult( ptr, ind, val, α, x, β, y );
```
How to call pOSKI: Basic Usage

• May gradually migrate existing apps
  – Step 1: “Wrap” existing data structures
  – Step 2: Make BLAS-like kernel calls

```c
int* ptr = ..., *ind = ...; double* val = ...; /* Matrix, in CSR format */
double* x = ..., *y = ...; /* Let x and y be two dense vectors */
/* Step 1: Create a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();
/* Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
                                        nnz, SHARE_INPUTMAT, poski_thread, NULL, ...);
poski_vec_t x_view = poski_CreateVec(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVec(y, nrows, UNIT_STRIDE, NULL);

/* Step 3: Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
    poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```
How to call pOSKI:
Tune with Explicit Hints

• User calls “tune” routine (optional)
  – May provide explicit tuning hints

```c
poski_mat_t A_tunable = poski_CreateMatCSR( ... );
   /* ... */
/* Tell pOSKI we will call SpMV 500 times (workload hint) */
poski_TuneHint_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view,500);
/* Tell pOSKI we think the matrix has 8x8 blocks (structural hint) */
poski_TuneHint_Structure(A_tunable, HINT_SINGLE_BLOCKSIZE, 8, 8);

/* Ask pOSKI to tune */
poski_TuneMat(A_tunable);

for( i = 0; i < 500; i++ )
   poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```
How to call pOSKI: Implicit Tuning

- Ask library to infer workload (optional)
  - Library profiles all kernel calls
  - May periodically re-tune

```c
poski_mat_t A_tunable = poski_CreateMatCSR( ... );
/* ... */

for( i = 0; i < 500; i++ ) {
    poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
    poski_TuneMat(A_tunable); /* Ask pOSKI to tune */
}
```
Performance on Intel Sandy Bridge E

- Jaketown: i7-3960X @ 3.3 GHz
- #Cores: 6 (2 threads per core), L3:15MB
- pOSKI SpMV (Ax) with double precision float-point
- MKL Sparse BLAS Level 2: mkl_dcsrmv()
Avoiding Communication in Sparse Linear Algebra

- Computational intensity of one SpMV \( \leq 2 \), limits performance
- \( k \)-steps of typical iterative solver for \( Ax=b \) or \( Ax=\lambda x \)
  - Does \( k \) SpMVs with starting vector (e.g., with \( b \), if solving \( Ax=b \))
  - Finds “best” solution among all linear combinations of these \( k+1 \) vectors
  - Many such “Krylov Subspace Methods”
    - Conjugate Gradients, GMRES, Lanczos, Arnoldi, ...
- Goal: minimize communication in Krylov Subspace Methods
  - Assume matrix “well-partitioned,” with modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: \( O(k \log p) \) messages, because \( k \) calls to SpMV
    - **New: \( O(\log p) \) messages - optimal**
  - Serial implementation
    - Conventional: \( O(k) \) moves of data from slow to fast memory
    - **New: \( O(1) \) moves of data – optimal**
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation, numerical stability issues
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)

- Example: A tridiagonal, \(n=32, k=3\)
- Works for any “well-partitioned” \(A\)
Communication Avoiding Kernels:  
The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$ 

- Replace $k$ iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$ 

- Example: A tridiagonal, $n=32$, $k=3$
Communication Avoiding Kernels: The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)

Example: A tridiagonal, \(n=32\), \(k=3\)
Communication Avoiding Kernels:
The Matrix Powers Kernel: $[Ax, A^2x, \ldots, A^kx]$

- Replace $k$ iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

Example: A tridiagonal, $n=32$, $k=3$
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)

Example: A tridiagonal, \(n=32\), \(k=3\)
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)

Example: A tridiagonal, \(n=32\), \(k=3\)
Communication Avoiding Kernels: The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)
- Sequential Algorithm

Example: A tridiagonal, \(n=32, k=3\)
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)
- Sequential Algorithm

Example: A tridiagonal, \(n=32\), \(k=3\)
Communication Avoiding Kernels: The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)
- Sequential Algorithm

Example: A tridiagonal, \(n=32\), \(k=3\)
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)
- Sequential Algorithm

\[
\begin{array}{cccc}
A^3 \cdot x & A^2 \cdot x & A \cdot x & x \\
\text{Step 1} & \text{Step 2} & \text{Step 3} & \text{Step 4}
\end{array}
\]

- Example: A tridiagonal, \(n=32, k=3\)
Communication Avoiding Kernels:
The Matrix Powers Kernel: \([Ax, A^2x, \ldots, A^kx]\)

- Replace \(k\) iterations of \(y = A\cdot x\) with \([Ax, A^2x, \ldots, A^kx]\)
- Parallel Algorithm

Example: A tridiagonal, \(n=32\), \(k=3\)
- Each processor communicates once with neighbors
Communication Avoiding Kernels: The Matrix Powers Kernel: \([Ax, A^2x, ..., A^kx]\)

- Replace \(k\) iterations of \(y = A \cdot x\) with \([Ax, A^2x, ..., A^kx]\)
- Parallel Algorithm

[Diagram of processor distribution and matrix multiplication]

- Example: A tridiagonal, \(n=32\), \(k=3\)
- Each processor works on (overlapping) trapezoid
Communication Avoiding Kernels: The Matrix Powers Kernel: $[Ax, A^2x, \ldots, A^kx]$

Same idea works for general sparse matrices

Partitioning by rows $\Rightarrow$ Graph partitioning

Processing left to right $\Rightarrow$ Traveling Salesman Problem
What about multicore?

- Two kinds of communication to minimize
  - Between processors on the chip
  - Between on-chip cache and off-chip DRAM

- Use hybrid of both techniques described so far
  - Use parallel optimization so each core can work independently
  - Use sequential optimization to minimize off-chip DRAM traffic of each core
Speedups on Intel Clovertown (8 core)
Test matrices include stencils and practical matrices
See SC09 paper on bebop.cs.berkeley.edu for details
### Classical GMRES for Ax=b

for i=1 to k  
  w = A * v(i-1)  
  MGS(w, v(0),...,v(i-1))  
    ... Modified Gram-Schmidt  
    ... to make w orthogonal  
  update v(i), H  
    ... H = matrix of coeffs  
    ... from MGS  
endfor  

solve LSQ problem with H for x

**Communication cost =**  
k copies of A, vectors from slow to fast memory

### Communication-Avoiding GMRES, ver. 1

W = [ v, Av, A²v, ... , A^k v ]  
[Q,R] = TSQR(W)  
  ... “Tall Skinny QR”  
  ... new optimal QR discussed before  
Build H from R  
solve LSQ problem with H for x

**Communication cost =**  
O(1) copy of A, vectors from slow to fast memory

Let’s confirm that we still get the right answer …
Matrix diag-cond-1.000000e-11: rel. 2-nrm resid.

- Right answer (converges)
- Oops, doesn’t converge
Minimizing Communication of GMRES 
(and getting the right answer)

Communication-Avoiding GMRES, ver. 2

\[ W = [ v, p_1(A)v, p_2(A)v, \ldots, p_k(A)v ] \]
... where \( p_i(A)v \) is a degree-\( i \) polynomial in \( A \) multiplied by \( v \)
... polynomials chosen to keep vectors independent

\[ [Q, R] = \text{TSQR}(W) \]
... “Tall Skinny QR”
... new optimal QR discussed before

Build \( H \) from \( R \)
... slightly different \( R \) from before
solve LSQ problem with \( H \) for \( x \)

Communication cost still optimal:
\( O(1) \) copy of \( A \), vectors from
slow to fast memory
Right answer (converges)

Oops, doesn’t converge

Converges again!

Right answer (converges)
Speed ups on 8-core Clovertown

CA-GMRES = Communication-Avoiding GMRES

Runtime per kernel, relative to CA-GMRES(k,t), for all test matrices, using 8 threads and restart length 60

Paper by Mohiyuddin, Hoemmen, D. in Supercomputing09
Summary of what is known, open

• GMRES
  – Can independently choose $k$ to optimize speed, restart length $r$ to optimize convergence
  – Need to “co-tune” $A_kx$ kernel and TSQR
  – Know how to use more stable polynomial bases
  – Proven speedups

• Can similarly reorganize other Krylov methods
  – Arnoldi and Lanczos, for $Ax = \lambda x$ and for $Ax = \lambda Mx$
  – Conjugate Gradients (CG), for $Ax = b$
  – Biconjugate Gradients (BiCG), CG Squared (CGS), BiCGStab for $Ax=b$
  – Other Krylov methods?

• Preconditioning – how to handle $MAx = Mb$
What is a sparse matrix?

<table>
<thead>
<tr>
<th>Values</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>LBM, Stencils</td>
<td>Laplacian of a Graph</td>
<td>-</td>
</tr>
<tr>
<td>on structured grids</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

- CBIR's SpMV
  - extremely large & complex stencil
  - e.g., CSR

- PIC
  - Histograms sparse matrix of #grid rows and #particles columns

- How much infrastructure (for code creation, tuning or interfaces) can we reuse for all these cases?
Sparse Conclusions

• Fast code must minimize communication
  – Especially for sparse matrix computations because communication dominates

• Generating fast code for a single SpMV
  – Design space of possible algorithms must be searched at run-time, when sparse matrix available
  – Design space should be searched automatically

• Biggest speedups from minimizing communication in an entire sparse solver
  – Many more opportunities to minimize communication in multiple SpMVs than in one
  – Requires transforming entire algorithm
  – Lots of open problems

• For more information, see bebop.cs.berkeley.edu
STRUCTURED GRID MOTIF

Source: Sam Williams
Applying the finite difference method to PDEs on structured grids produces **stencil operators** that must be applied to all points in the discretized grid.

Consider the 7-point Laplacian Operator

Challenged by bandwidth, temporal reuse, efficient SIMD, etc...

but trivial to (correctly) parallelize

**most optimizations can be independently implemented,**

*(but not performance independent)*

core (cache) blocking and cache bypass were clearly integral to performance
Structured Grids
Lattice Boltzmann Methods

• LBMHD simulates charged plasmas in a magnetic field (MHD) via Lattice Boltzmann Method (LBM) applied to CFD and Maxwell’s equations.

• To monitor density, momentum, and magnetic field, it requires maintaining two “velocity” distributions
  – 27 (scalar) element velocity distribution for momentum
  – 15 (Cartesian) element velocity distribution for magnetic field
  – = 632 bytes / grid point / time step

• Jacobi-like time evolution requires ~1300 flops and ~1200 bytes of memory traffic
Structured Grids

Lattice Boltzmann Methods

- Challenged by:
  - The higher flop:byte ratio of ~1.0 is still bandwidth-limiting
  - TLB locality (touch 150 pages per lattice update)
  - cache associativity (150 disjoint lines)
  - efficient SIMDization

- easy to (correctly) parallelize
- explicit SIMDization & SW prefetch are dependent on unrolling
- Ultimately, 2 of 3 machines are bandwidth-limited

---

**Opteron 2356 (Barcelona)**

**Xeon E5345 (Clovertown)**

**UltraSparc T2+ T5140 (Victoria Falls)**
Structured Grids
Lattice Boltzmann Methods

- **Distributed Memory & Hybrid**
  - MPI, MPI+pthreads, MPI+OpenMP (SPMD, SPMD², SPMD+Fork/Join)

- Observe that for this large problem, **auto-tuning flat MPI delivered significant boosts (2.5x)**

- Extending auto-tuning to include the domain decomposition and balance between threads and processes **provided an extra 17%**

- 2 processes with 2 threads was best (true for Pthreads and OpenMP)
DELIVERING AUTOTUNING WITH SEJITS

Source: Shoaib Kamil
What is SEJITS?

• Goal: Let non-expert programmers quickly write their algorithms in an easy-to-use language, but still get high performance
  – First example: Python
• By using common “patterns” to write algorithms, and hints about tuning opportunities, enable system to autotune
• SEJITS = Selective Embedded Just-in-time Specialization
Delivering Autotuning via SEJITS

Several examples exist now:
- Structured Grids/Stencils
- CA-Conjugate Gradient
- Tuned SpMV over other semirings
Summary

• “Design spaces” for algorithms and implementations are large and growing
• Finding the best algorithm/implementation by hand is hard and getting harder
• Ideally, we would have a database of “techniques” that would grow over time, and be searched automatically whenever a new input and/or machine comes along
• Lots of work to do…