

Performance Issues for Sparse Iterative Solvers

Michael A. Heroux
Sandia National Laboratories



Outline of the Talk

1. Scope of Efforts.
2. Iterative Methods 101.
3. Important kernel operations.
4. Kernel performance issues.
5. Summary.



Scope of Effort

- Sandia has many engineering applications.
- A large fraction of newer apps are *implicit* in nature:
 - Requires solution of many large nonlinear systems.
 - Boils down to many sparse linear systems.
- Linear system solves are large fraction of total time.
 - Small as 30%.
 - Large as 90+%.
- Iterative solvers most commonly used.
- Iterative solvers have small handful of important kernels.
- We focus on performance issues for these kernels.
 - Caveat: These parts do not make the whole, but are a good chunk of it...

Problem Definition

- A frequent requirement for scientific and engineering computing is to solve:

$$Ax = b$$

where A is a known large (sparse) matrix,
 b is a known vector,
 x is an unknown vector.

- Goal: Find x .
- Method:
 - Use Preconditioned Conjugate Gradient (PCG) method,
 - Or one of many variants, e.g., Preconditioned GMRES.

Other Types of Problems

- Nonlinear problems: $f(u) = 0$:
 - Example: $u(x)u(x)' - \sin(x)\cos(x) = 0$.
- Eigenvalue problems: $Ax = \lambda x$.

$$\begin{bmatrix} 1 & -2 & 1 \\ 0 & -2 & 2 \\ 2 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 0 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

- Many variations.
- Sparse matrix multiplication: Basic op for all above.
- Linear solver often basic component for all.
- Iterative linear solvers important on parallel machines.

Classic Prototype: The CG Method

```
i = 0; xi-1 = 0; ri-1 = b; A given by user;
while norm(ri) > tol {
    i ++;
    rtri-1 = ddot(ri-1, ri-1);
    if (i=1) pi = ri-1;
    else {
        bi = rtri-1 / rtri-2;
        pi = ri-1 + bi * pi-1;
    }
    Api = sparsemv(A, pi);
    MApi = applyPrec(M, Api);
    ai = rtri-1 / ddot(pi, MApi);
    xi = xi-1;    xi = xi-1 + ai * pi;
    ri = ri-1;    ri = ri-1 - ai * MApi;
}
x = xi; // When norm(ri) ≤ tol, stop and set x to xi
```



Three Categories of Operations

- Vector reductions and updates:
 - Dot products (*ddot*).
 - Norms (*norm*).
 - AXPYs.
 - Multitude of AXPY variations.
- Sparse matrix–dense vector product (*sparsemv*).
 - Compressed Row–oriented.
 - Compressed Column oriented.
- Preconditioner:
 - Setup (one–time cost):
 - Form coarse grid operator, or
 - Compute incomplete factorization, or
 - Not much.
 - Apply: Repeated proportional to sparse MV (*applyPrec*)

Vector Reductions and Updates

Kernel	Reads	Writes	Mults	Adds	Bytes Per Clock Memory Bandwth
ddot	2n	1	n	n	16
norm	n	1	n	n	8
axpy	2n	n	n	n	24

- Per Clocktick Memory Bandwidth Requirement:
 - Assumes simultaneous mult/add (two FP ops/clk).
 - Number of bytes needed per clk for optimal performance.
- Data window of solver is large:
 - Little chance of temporal re-use,
 - Except multi-MB cache with small problems.
- Bottom line:
 - Optimal vector kernel performance requires lots of bandwidth.
 - Any improvements would please us.
 - A lot is required to satisfy us.



Row-based sparse matrix- vector multiply

```
c.....do sequence of SPDOTs (sparse sdots)
      jend = pntr(0)
      do 10 j = 0, m-1
        jbgn = jend
        jend = pntr(j+1)
        sum = 0.0
        do 20 i = jbgn, jend-1
          sum = sum + val(i) * x(indx(i))
20      continue
        y(j) = sum
10      continue
```

- Notes:
 - Written in Fortran (still better than C). Called from C++ wrappers.
 - `val` and `indx` of length number of nonzeros in matrix.
 - `pntr`, `x` and `y` of length matrix dimension.
 - `val`, `pntr`, `indx` and `y` accessed sequentially and used once.
 - `x` accessed indirectly, typically some effective cache use.
 - Loop 20 is of average length 10s, regardless of problem size.
- Optimal bandwidth: 3 reads & 1 write per clock
 - Assuming simultaneous mult/add and no re-use of `x` values.
 - Some `x` values will be re-used.
 - Some `x` values will be read into cache and flushed without being used.
- Sparse triangular solve also important:
 - Nearly identical kernel.
 - Needed for many preconditioners.



Sparse MV Observations

- \mathbf{x} is the only array that benefits from cache:
 - Some temporal re-use from row-to-row.
 - Some spatial locality that acts as pre-fetching.
- All other arrays are:
 - Accessed sequentially.
 - Used once and discarded.
- Sparse column variant has similar properties:
 - \mathbf{y} is cacheable. All others not.
- General observations:
 - A sophisticated cache memory system is mostly inappropriate for these kernels.
 - Some kind of streamed access with cache bypass would be very attractive.

Preconditioners

- Preconditioners tend to rely on kernels already mentioned:
 - Sparse MV, vector updates, Triangular solves.
- One additional kernel is sparse matrix triple product for multi-level preconditioners (called RAP):

$$A_C = RA_F P.$$

- RAP is part of preconditioner setup:
 - Done once per solve.
 - Still cost can be substantial.
 - This kernel not well-studied (by us).
 - Temporal re-use of data is higher than other kernels.
 - Spatial re-use also, but probably offset by unused cache line entries.

Where We Are: One Data Point

- Example: AMD Opteron (242)
 - 1.6GHz, 1MB L2 cache, 3100 BogoMIPS.
 - Use DGEMM as practical achievable peak:
 - 2760 MFLOPS (using Hammer-specific ATLAS BLAS)
 - 10X effective bandwidth increase would satisfy us.

Kernel	Asymptotic Peak MFLOPS	Percent of DGEMM Peak
ddot	400	14.5%
axpy	300	10.9%
sparsemv	250–310	9.0%–11.2%



Observations/Questions

- Sophisticated cache memory systems are greatly underused by sparse iterative solvers:
 - Very few kernels can benefit.
 - Even when useful:
 - Performance gains from temporal locality can be offset by unused cache line data.
 - Partial cache line fill mode?
 - Much of the time it gets in the way.
- We could really use a high bandwidth streaming memory system.
 - Much of our memory traffic is long-array unit-stride.
 - Some kind of heavily interleaved memory system that could bypass cache?

Summary

- Many Sandia applications are implicit.
- Implicit apps need (sparse iterative) solvers.
- Sparse iterative solvers:
 - Use between 30–90+% of total application run-time.
 - Cost is similar across major Sandia frameworks. (e.g., SIERRA, NEVADA, XYCE)
- Sparse iterative solver kernels are a challenge:
 - Most memory access is sequential one-time use.
 - Indirect memory accesses can utilize cache, but...
 - Re-use is offset at least somewhat by unused cache-line entries.
- Any increase in bandwidth is welcome.
- Less aggressive spatial prefetch and streaming memory capabilities seem attractive.