A quad-tree based Sparse BLAS implementation for shared memory parallel computers

Michele Martone

Università di Roma “Tor Vergata”, Italy

Rome, May 31, 2011
numerical matrices which are *large* and populated mostly by zeros

ubiquitous in scientific/engineering computations (e.g.: PDE)

used in *information retrieval* and *document ranking*

the *performance* of sparse matrix codes computation on modern CPUs can be problematic (a fraction of peak)!

there is no "silver bullet" for performance

jargon: *performance* = *time efficiency*
An example application

Simulation of an automotive engine performed with the aid of the PSBLAS linear solver software (See [BBDM$^+$05]). Courtesy of Salvatore Filippone.
Our Focus

The numerical solution of linear systems of the form $Ax = b$ (with $A$ a sparse matrix, $x, y$ dense vectors) using iterative methods requires repeated (and thus, fast) computation of (variants of):

- **SpMV**: “$y \leftarrow A \times$”
- **SpMV-T**: “$y \leftarrow A^T \times$”
- **SpSV**: “$x \leftarrow L^{-1} \times$”
- **SpSV-T**: “$x \leftarrow L^{-T} \times$”
high performance programming cache based, shared memory parallel computers requires:

- **locality of memory references**—for the memory hierarchy has:
  - limited memory bandwidth
  - memory access latency
- programming multiple cores for coarse-grained *workload partitioning*
  - high synchronization and cache-coherence costs
Sparse matrices require indirect addressing
e.g. (in C): \( k = \text{RP}[i]; x[i] = x[i] + VA[k] * y[JA[k]] \);

- additional latency
  ("random" accessing the first element of line \( i \) in a CSR matrix requires two dependent memory accesses)
- "wasting" cache lines (indices \( JA[k] \) and nearby will be cached but not reused, and so will indices \( y[JA[k]] \) and nearby)
- with many active cores, saturation of the memory subsystem traffic capacity
we could *mitigate* using...

- **index compression**: less memory traffic
- **cache blocking**: more cached data reuse
- dense (or register) blocking: less indices (⇒ less memory traffic)
We draw knowledge from:

- existing sparse matrix codes (See Filippone and Colajanni [FC00], Buttari [But06])
- classical algorithms (See Barrett et al. [BBC+94, § 4.3.1])
- novel techniques (See Buluç [BFF+09], Nishtala et al. [NVDY04], Vuduc [Vud03])
Basic representation: Coordinate (COO)

\[ A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & 0 & a_{4,4} \end{bmatrix} \]

- \( VA = [a_{1,1}, a_{1,2}, a_{1,3}, a_{2,2}, a_{2,3}, a_{3,3}, a_{4,4}] \) (nonzeroes)
- \( IA = [1, 1, 1, 2, 2, 3, 4] \) (nonzeroes row indices)
- \( JA = [1, 2, 3, 2, 3, 3, 4] \) (nonzeroes column indices)
- so, \( a_{i,j} = VA(n) \) if \( IA(n) = i, JA(n) = j \)
Pros and Cons of COO

▶ + good for layout conversion
▶ + easy implementation of many simple operations
▶ - parallel $SpMV/SpMV-T$ requires sorting of elements
▶ - efficient $SpSV$ is complicated, in parallel even more
Standard representation: Compressed Sparse Rows (CSR)

\[
A = \begin{bmatrix}
  a_{1,1} & a_{1,2} & a_{1,3} & 0 \\
  0 & a_{2,2} & a_{2,3} & 0 \\
  0 & 0 & a_{3,3} & 0 \\
  0 & 0 & 0 & a_{4,4}
\end{bmatrix}
\]

- \( VA = [a_{1,1}, a_{1,2}, a_{1,3}, a_{2,2}, a_{2,3}, a_{3,3}, a_{4,4}] \) (nonzeroes)
- \( JA = [1, 2, 3, 2, 3, 3, 4] \) (nonzeroes column indices)
- \( RP = [1, 4, 6, 7, 8] \) (row pointers, for each row)
- so, elements on line \( i \) are in positions \( VA(RP(i)) \) to \( VA(RP(i + 1)) - 1 \)
- so, \( a_{i,j} = VA(n) \) if \( JA(n) = j \)
Pros and Cons of CSR

- + common, easy to work with
- + parallel SpMV is feasible and reasonably efficient
- - parallel SpMV-T is feasible ...but inefficient with large matrices
- - impractical for parallel SpSV

\[^{1}\text{That is, when a matrix memory representation approaches to occupy the machine's memory size.}\]
we propose:

▶ a *quad-tree* of sparse *leaf* submatrices
▶ outcome of recursive *partitioning* in *quadrants*
▶ submatrices are stored *row oriented* (in CSR)
▶ an *unified* format for Sparse *BLAS*\(^2\) operations

\(^2\)Basic Linear Algebra Subprograms
we stop subdivision of prospective leaves on an expected work/efficiency basis:

- leaves **too small** (e.g.: comparable to the cache capacity)
- leaves **too sparse** for CSR (e.g.: when $nnz < rows$)
Instance of an Information Retrieval matrix (573286 rows, 230401 columns, 41 \cdot 10^6 nonzeroes):

Courtesy of Diego De Cao.
Dual threaded recursive \textit{SpMV}

We compute $y_1$ in the first thread, $y_2$ in the second:

\[
\begin{vmatrix}
  y_1 \\
  y_2
\end{vmatrix} = A \begin{vmatrix}
  x_1 \\
  x_2
\end{vmatrix} = \begin{vmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{vmatrix} \begin{vmatrix}
  x_1 \\
  x_2
\end{vmatrix}
\]

\[
= \begin{vmatrix}
  A_{11} & A_{12} \\
  0 & 0
\end{vmatrix} \begin{vmatrix}
  x_1 \\
  x_2
\end{vmatrix} + \begin{vmatrix}
  0 & 0 \\
  A_{21} & A_{22}
\end{vmatrix} \begin{vmatrix}
  x_1 \\
  x_2
\end{vmatrix}
\]

\[
= \begin{vmatrix}
  A_{11}x_1 + A_{12}x_2 \\
  0
\end{vmatrix} + \begin{vmatrix}
  0 \\
  A_{21}x_1 + A_{22}x_2
\end{vmatrix}
\]

Recursion continues on each thread
Single threaded recursive $SpSV$

\[
Lx = b \implies \begin{bmatrix}
L_1 & 0 & x_1 \\
M & L_2 & x_2
\end{bmatrix} = \begin{bmatrix}
b_1 \\
b_2
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
x_1 \\
x_2
\end{bmatrix} = L^{-1}b = \begin{bmatrix}
L_1 & 0 \\
M & L_2
\end{bmatrix}^{-1} \begin{bmatrix}
b_1 \\
b_2
\end{bmatrix} = \begin{bmatrix}
L_1^{-1}b_1 \\
L_2^{-1}(b_2 - Mx_1)
\end{bmatrix}
\]

This computation is executed recursively.
Pros/Cons of RCSR

Experimentally, on two threads:

- + compares well to CSB (an efficient research prototype)
- + coarse partitioning of workload (especially good for parallel transposed $SpMV$)
- + locality in the access of the matrix vectors
- - some matrix patterns may lead to *unbalanced* subdivisions
- - some nonempty leaf matrices may contain empty row intervals
Multi-threaded SpMV

\[ Y \leftarrow Y + A \times X \]

\[ Y_0 \quad Y_0 \quad A_0 \quad X_0 \]
\[ Y_1 \quad Y_1 \quad A_1 \quad X_1 \]

\[ y \leftarrow y + \sum_i A_i \times x_i \text{, with leaves } A_i; \quad A = \sum_i A_i \]
Multi-threaded SpMV

\[ y \leftarrow y + \sum_i A_i \times x_i \]

Threads \( t \in \{1..T\} \) execute concurrently:

\[ y_{it} \leftarrow y_{it} + A_{it} \times x_{it} \]

we prevent race conditions performing idle cycles if needed; we use\(^3\):

- per-submatrix visit information
- per-thread current submatrix information

\(^3\)See extra: slide 29
Multi-threaded SpSV

\[
L^{-1} \times X
\]

\[
X \leftarrow L^{-1} \times X
\]
Multi-threaded $SpSV$

\[ x \leftarrow L^{-1}x = (\sum L_i)^{-1}x \]

A thread $t \in \{1...T\}$ may perform either:

- $x_i t \leftarrow x_i t + L_i t \times s_i t$ (forward substitution)
- $x_i t \leftarrow L_i^{-1}x_i t$ ($x_i t = s_i t$) (solve)
- idle (wait for dependencies)

Consistency with $SpMV$'s techniques plus honouring dependencies\(^4\).

\(^4\)See extra:slide 30
Improving RCSR?

- tuning the leaf submatrices representations
  - *compressing* numerical indices on leaves
  - using a coordinate representation on *some* leaves
The idea:

- when a submatrix is less than $2^{16}$ columns wide, we use a 16 bit integer type for column indices\(^5\)
- overall, up to 16\% memory saving on double precision CSR
- overall, up to 32\% memory saving on float precision CSR

⇒ likely speedup due to reduced memory traffic!

\(^5\)Instead of a standard 32 bit integer.
But how?

- Our code is large (hundreds of thousands of LOC for all of our variants\(^6\))!
- ⇒ using custom **code generators** for all variants
- Our matrices are usually dimensioned \(\gg 2^{16}\)!
- ⇒ if possible, subdividing **until** submatrices are narrower than \(2^{16}\)

---

\(^6\)A BLAS implementation shall several different operation variants
Consequences — a finer partitioning—RCSRH

instance of $kkt\_power$ $(2063494 \times 2063494, 813034$ nonzeroes) in RCSR (left) and in RCSRH (right)
Pros/Cons of RCSRH

- speedup due to reduced memory overhead
- for some matrices, RCSRH requires more memory than RCSR (a consequence of submatrices hypersparsity; see Buluç and Gilbert [BG08])
We introduce selectively COO (coordinate format) leaf submatrices to:
  - use less memory than CSR in particular submatrices (more rows than nonzeroes)
  - subdivide *more* very sparse submatrices, for a better workload partitioning
Recursive Sparse Blocks: a hybrid sparse matrix format.

- recursive quad-partitioning
- CSR/COO leaves
- 16 bit indices whenever possible
- partitioning with regards to both the underlying cache size and available threads
Pros/Cons of RSB

- + scalable parallel $SpMV/SpMV-T$
- + scalable parallel $SpSV/SpSV-T$
- + many other common operations (e.g.: parallel matrix build algorithm)
- + native support for symmetric matrices
- - a number of known cases (unbalanced matrices) where parallelism is poor
- - some algorithms easy to express/implement for CSR are more complex for RSB
Experimental time efficiency comparison of our RSB prototype to the proprietary, highly optimized Intel’s Math Kernels Library (MKL r.10.3-0) sparse matrix routines.

We report here results on an Intel Xeon 5670 and publicly available matrices.
Comparison to MKL, SPMV

Figure: Transposed/Non transposed SpMV performance on M4, versus MKL, 12 threads, large unsymmetric matrices.
Comparison to MKL, Symmetric SPMV

Figure: $SpMV$ performance on M4, versus MKL, 12 threads, symmetric matrices.
Comparison to MKL, SPSV

Figure: Transposed/Non transposed SpSV performance on M4, versus MKL, single thread.

(MKL SpSV is not multithreaded)
Conclusions

A shared memory parallel Sparse BLAS library implementation:

- on large matrices, better performance than Intel’s highly optimized, proprietary CSR implementation
- provides primitives for sparse solvers
- usable from within the open source PSBLAS library
- may be further tuned


Forthcoming:

- interfacing with Matlab/Octave (via the PSBLAS solver)
- interfacing with Octave (standalone)
- study of further tuning of $SpMV/SpMV-T/SpSV/SpSV-T$

Possible:

- new specialized computational kernels
- conversion tools/routines
- new formats for leaf submatrices

Gino Bella, Alfredo Buttari, Alessandro De Maio, Francesco Del Citto, Salvatore Filippone, and Fabiano Gasperini.  
Fast-evp: An engine simulation tool.  

Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks.  
In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, SPAA, pages 233–244. ACM, 2009.

Aydın Buluç and John R. Gilbert.  
On the Representation and Multiplication of Hypersparse Matrices.  
In IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), April 2008.
Alfredo Buttari.
**Software Tools for Sparse Linear Algebra Computations.**

Salvatore Filippone and Michele Colajanni.
**PSBLAS: A library for parallel linear algebra computation on sparse matrices.**

**When cache blocking sparse matrix vector multiply works and why.**

Richard Wilson Vuduc.
**Automatic performance tuning of sparse matrix kernels (phd thesis).**
Thank you for your attention!

(Extra slides following)
We compute $y_1$ in the first thread, $y_2$ in the second:

$$
\begin{vmatrix}
    y_1 \\
    y_2
\end{vmatrix} = A^T x =
\begin{pmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{pmatrix}^T
\begin{vmatrix}
    x_1 \\
    x_2
\end{vmatrix} =
\begin{pmatrix}
    A_{11}^T & A_{21}^T \\
    A_{12}^T & A_{22}^T
\end{pmatrix}
\begin{vmatrix}
    x_1 \\
    x_2
\end{vmatrix} =
\begin{pmatrix}
    A_{11}^T x_1 + A_{21}^T x_2 \\
    0
\end{vmatrix} +
\begin{pmatrix}
    0 \\
    A_{12}^T x_1 + A_{22}^T x_2
\end{pmatrix}
$$

Recursion continues on each thread
Multi-threaded SPMV

\[ y \leftarrow y + \sum_i A_i \times x_i \]

Concurrently on threads \( t \in \{1..T\} \):

\[ y_{it} \leftarrow y_{it} + A_{it} \times x_{it} \]

For threads \((t, u)\), no \((y_{it}, y_{iu})\) shall intersect at a given time — using lock+usage bitmap to prevent \textit{race conditions}
Multi-threaded SPSV

\[ x \leftarrow L^{-1}x = (\sum L_i)^{-1}x \]

On threads \( t \in \{1...T\} \) either task:

1) \( x_{it} \leftarrow x_{it} + L_{it} \times s_{it} \) (forward substitution)

or

2) \( x_{it} \leftarrow L_{it}^{-1}x_{it} \) (solve) \( (x_{it} = s_{it}) \)

\( x_i \): subrow on \( L'_{it} \) s rows range
\( s_i \): subrow on \( L'_{it} \) s columns range (using lock+usage map+array)
Figure: The relative performance of some linear scan primitives on M2 (an AMD Opteron 2354), M4 (Intel Xeon 5670). We have parameters $w_s = 8$, $\kappa = 1024$, $w_n = 128 \cdot 1024$. 
Figure: The relative performance of some linear scan primitives on M6 (an Atom 450N), M8 (a Pentium III). We have parameters $w_s = 8, \kappa = 1024, w_n = 128 \cdot 1024$. 

Relative Memory Scan Speeds

<table>
<thead>
<tr>
<th>stride (#of words)</th>
<th>words per second to 1-stride (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>64</td>
<td>5</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
</tr>
<tr>
<td>512</td>
<td>0.5</td>
</tr>
<tr>
<td>1024</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Legend:
- M6-IND-O0
- M6-IND-O3
- M6-LIN-O0
- M6-LIN-O3
- M6-RND-O0
- M6-RND-O3
- M8-IND-O0
- M8-IND-O3
- M8-LIN-O0
- M8-LIN-O3
- M8-RND-O0
- M8-RND-O3
End of Extra slides

Thank you for your extra attention!