Portable, usable, and efficient sparse matrix–vector multiplication

Albert-Jan Yzelman
Parallel Computing and Big Data
Huawei Technologies France

8th of July, 2016
Introduction

Given a **sparse** $m \times n$ matrix $A$, and corresponding vectors $x$, $y$.

- How to calculate $y = Ax$ as fast as possible?
- How to make the code usable for the 99%?

Figure: Wikipedia link matrix ('07) with on average $\approx 12.6$ nonzeros per row.
Central obstacles for SpMV multiplication

**Shared-memory:**
- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

**Distributed-memory:**
- inefficient network use.
Usable sparse matrix–vector multiplication

Central obstacles for SpMV multiplication

**Shared-memory:**
- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

**Distributed-memory:**
- inefficient network use.

Shared-memory and distributed-memory share their objectives:

\[
\text{cache misses} = \text{communication volume}
\]

Usable sparse matrix–vector multiplication

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeros processed in row-major order:

Accesses on the input vector are completely unpredictable.
**Usable sparse matrix–vector multiplication**

**Enhanced cache use: nonzero reorderings**

**Blocking** to cache subvectors, and **cache-oblivious traversals**.

Other approaches: no blocking (Haase et al.), Morton Z-curves and bisection (Martone et al.), Z-curve within blocks (Buluç et al.), composition of low-level blocking (Vuduc et al.), ...

Usable sparse matrix–vector multiplication

**Enhanced cache use: nonzero reorderings**

**Blocking** to cache subvectors, and **cache-oblivious traversals**.

Sequential SpMV multiplication on the Wikipedia '07 link matrix:
345 (CRS), 203 (Hilbert), 245 (blocked Hilbert) ms/mul.


A. N. Yzelman
Usable sparse matrix–vector multiplication

Enhanced cache use: matrix permutations

(Upper bound on) the number of cache misses: \( \sum_{i} (\lambda_i - 1) \)

Usable sparse matrix–vector multiplication

Enhanced cache use: matrix permutations

\[ \text{cache misses} \leq \sum_{i} (\lambda_i - 1) = \text{communication volume} \]


Should we program shared-memory as though it were distributed?
Enhanced cache use: matrix permutations

Practical gains:

Figure: the Stanford link matrix (left) and its 20-part reordering (right).

Sequential execution using CRS on Stanford:

18.99 (original), 9.92 (1D), 9.35 (2D) ms/mul.

Usable sparse matrix–vector multiplication

Bandwidth

Theoretical turnover points: Intel Xeon E3-1225

- 64 operations per word (with vectorisation)
- 16 operations per word (without vectorisation)

(Image taken from da Silva et al., DOI 10.1155/2013/428078, Creative Commons Attribution License)
Exploiting sparsity through computation using only nonzeros:

\[ i = (0, 0, 1, 1, 2, 2, 2, 3) \]
\[ j = (0, 4, 2, 4, 1, 3, 5, 2) \]
\[ v = (a_{00}, a_{04}, \ldots, a_{32}) \]

\[
\text{for } k = 0 \text{ to } nz - 1
\]

\[
y_{i_k} := y_{i_k} + v_k \cdot x_{j_k}
\]

The coordinate (COO) format: two flops versus five data words.

\[ \Theta(3nz) \] storage. CRS: \[ \Theta(2nz + m) \].
Usable sparse matrix–vector multiplication

Efficient bandwidth use

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1
\end{pmatrix}
\]

Bi-directional incremental CRS (BICRS):

\[
A = \begin{cases}
V & [7 1 4 1 2 3 3 2 1 1] \\
\Delta J & [0 4 4 1 5 4 5 4 3 1] \\
\Delta I & [3 -1 -2 1 -1 1 1 1]
\end{cases}
\]

Storage requirements, allowing **arbitrary traversals**:

\[
\Theta(2nz + \text{row\_jumps} + 1).
\]

Usable sparse matrix–vector multiplication

Efficient bandwidth use

With BICRS you can, distributed or not,

- **vectorise**, 
- **compress**, 
- **do blocking**, 
- **have arbitrary nonzero or block orders**.

Optimised BICRS takes less than or equal to $2nz + m$ of memory.


Usable sparse matrix–vector multiplication

**NUMA**

Each socket has **local** main memory where access is **fast**.

Memory access between sockets is slower, leading to *non-uniform memory access* (NUMA).
Usable sparse matrix–vector multiplication

NUMA

Access to only one socket: limited bandwidth
Usable sparse matrix–vector multiplication

NUMA

Interleave memory pages across sockets: emulate uniform access
Explicit data placement on sockets: best performance
Usable sparse matrix–vector multiplication

One-dimensional data placement

Coarse-grain row-wise distribution, compressed, cache-optimised:

- explicit allocation of separate matrix parts per core,
- explicit allocation of the output vector on the various sockets,
- interleaved allocation of the input vector,

Usable sparse matrix–vector multiplication

Two-dimensional data placement

Distribute row- and column-wise (individual nonzeros):
- most work touches only local data,
- inter-process communication minimised by partitioning;
- incurs cost of partitioning.


Usable sparse matrix–vector multiplication

Results

Sequential CRS on Wikipedia ’07: 472 ms/mul. 40 threads BICRS:

21.3 (1D), 20.7 (2D) ms/mul. Speedup: \(\approx 22x\).
Usable sparse matrix–vector multiplication

Results

Sequential CRS on Wikipedia '07: 472 ms/mul. 40 threads BICRS:

21.3 (1D), 20.7 (2D) ms/mul. Speedup: $\approx 22x$.

<table>
<thead>
<tr>
<th></th>
<th>2 x 6</th>
<th>4 x 10</th>
<th>8 x 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-$, 1D fine-grained, CRS*</td>
<td>4.6</td>
<td>6.8</td>
<td>6.2</td>
</tr>
<tr>
<td>Hilbert, Blocking, 1D, BICRS*</td>
<td>5.4</td>
<td>19.2</td>
<td>24.6</td>
</tr>
<tr>
<td>Hilbert, Blocking, 2D, BICRS†</td>
<td>$-$</td>
<td>21.3</td>
<td>30.8</td>
</tr>
</tbody>
</table>

Average speedup on six large matrices.
† uses an updated test set, added for reference versus a good 2D algorithm.

Efficiency $\rightarrow 0$ as NUMA $\rightarrow \infty$, if not 2D.


Usable sparse matrix–vector multiplication

Usability

Problems with integration into existing codes:

- SPMD (PThreads, MPI, ...) vs. others (OpenMP, Cilk, ...).
- Globally allocated vectors versus explicit data allocation.
- Conversion between matrix data formats.
- **Portable** codes and/or APIs: GPUs, x86, ARM, phones, ...
- Out-of-core, streaming capabilities, dynamic updates.
- User-defined overloaded operations on user-defined data.
- Ease of use.

Wish list:

- Performance and scalability.
- Standardised API? Updated and generalised Sparse BLAS: GraphBLAS.org
- Interoperability (PThreads + Cilk, MPI + OpenMP, DSLs!)
Usable sparse matrix–vector multiplication

Usability

Very high level languages for large-scale computing: resilient, scalable, huge uptake; **expressive** and **easy to use**.

MapReduce/Hadoop, Flink, **Spark**, Pregel/Giraph

```scala
scala> val A = sc.textFile( "A.txt" ).map( x => x.split( "\n" ) match {
  case Array( a, b, c ) => ( a.toInt - 1, b.toInt - 1, c.toDouble )
}) .groupBy( x => x._1 ) ;
A: org.apache.spark.rdd.RDD[(Int, Iterable[(Int, Int, Double)])] = ShuffledRDD[8] ...

scala>
```

- **RDDs** are **fine-grained** data distributed by hashing;
- Transformations (map, filter, groupBy) are **lazy** operators;
- **DAGs** thus formed are resolved by actions: reduce, collect, ...
- Computations are **offloaded as close to the data as possible**;
- **all-to-all** data shuffles for communication required by actions.

Spark is implemented in Scala, runs on the JVM, relies on serialisation, and commonly uses HDFS for distributed and resilient storage.

Platforms like Spark essentially perform PRAM simulation:

- **automatic mode** vs. **direct mode**
- **easy-of-use** vs. **performance**

Platforms like Spark essentially perform PRAM simulation:

- **automatic mode** vs. **direct mode**
- **easy-of-use** vs. **performance**


**A bridge between Big Data and HPC:**

- **Spark I/O via native RDDs** and native Scala interfaces;
- Rely on serialisation and the JNI to **switch to C**;
- Intercept Spark’s execution model to **switch to SPMD**;
- Set up and enable **inter-process RDMA communications**.
Usable sparse matrix–vector multiplication

Bridging HPC and Big Data

Some preliminary shared-memory results, Spark.

- **SpMV multiply** (plus basic vector operations):

  Cage15, $n = 5\ 154\ 859$, $nz = 99\ 199\ 551$. Using the 1D method.

![SpMV multiplication performance compared](image)

```scala
scala> val A_rdd = readCoordMatrix("..."); // type: RDD[(Long, Iterable[(Long, Double)])]
scala> val A = createMatrix(A_rdd, P); // type: SparseMatrix
scala> val x = InputVector(sc, A); val y = OutputVector(sc, A); // type: DenseVector
scala> vxm(sc, x, A, y);
scala> val y_rdd = toRDD(sc, y);
```

This is ongoing work; we are improving performance, extending functionality.

A. N. Yzelman
Usable sparse matrix–vector multiplication

Bridging HPC and Big Data

Some preliminary shared-memory results, Spark.

- a machine learning application:

  Training stage, internal test data.

```
sca\la > val in = sc.textFile("...");  //type: RDD[String]
sca\la > val out = MLalgo(in, 16);     //type: RDD[(Int, Double)]
sca\la > val nfea = out.count;
res6: Long = 1285593
```

![A machine learning algorithm compared](chart)
Conclusions and future work

Needed for current algorithms:
- faster partitioning to enable scalable 2D sparse computations,
- integration in practical and extensible libraries (GraphBLAS),
- making them interoperable with common use scenarios.

Extend application areas further:
- sparse power kernels,
- symmetric matrix support,
- graph and sparse tensor computations,
- support various hardware and execution platforms (Hadoop?).

Thank you!

The basic SpMV multiplication codes are free:
http://albert-jan.yzelman.net/software#SL
Backup slides
Results: cross platform

Cross platform results over 24 matrices:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Structured</th>
<th>Unstructured</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi</td>
<td>21.6</td>
<td>8.7</td>
<td>15.2</td>
</tr>
<tr>
<td>2x Ivy Bridge CPU</td>
<td>23.5</td>
<td>14.6</td>
<td>19.0</td>
</tr>
<tr>
<td>NVIDIA K20X GPU</td>
<td>16.7</td>
<td>13.3</td>
<td>15.0</td>
</tr>
</tbody>
</table>

no one solution fits all.

If we must, some generalising statements:

- Large structured matrices: GPUs.
- Large unstructured matrices: CPUs or GPUs.
- Smaller matrices: Xeon Phi or CPUs.

Vectorised BICRS