Efficient sparse matrix computations and their generalization to graph computing applications

Albert-Jan Yzelman
Parallel Computing and Big Data
Paris Research Centre
10th of February, 2017

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?

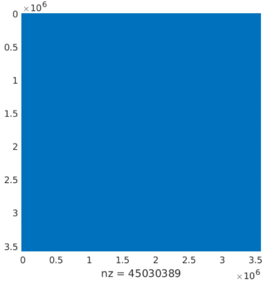


Figure: Wikipedia link matrix ('07) with on average ≈ 12.6 nonzeroes per row.

Central obstacles for SpMV multiplication

Shared-memory:

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

Distributed-memory:

- inefficient network use.

Central obstacles for SpMV multiplication

Shared-memory:

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

Distributed-memory:

- inefficient network use.

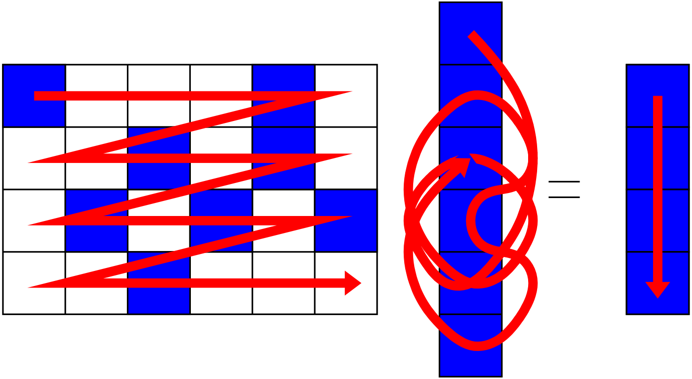
Shared-memory and distributed-memory share their objectives:

minimisation of data movement.

Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods by A. N. Yzelman & Rob H. Bisseling in SIAM Journal of Scientific Computation 31(4), pp. 3128-3154 (2009).

Inefficient cache use

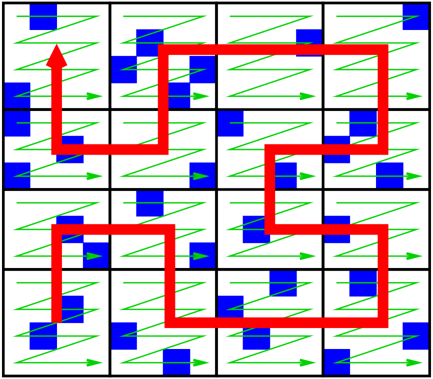
Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:



Accesses on the input vector are completely unpredictable.

Enhanced cache use: nonzero reorderings

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

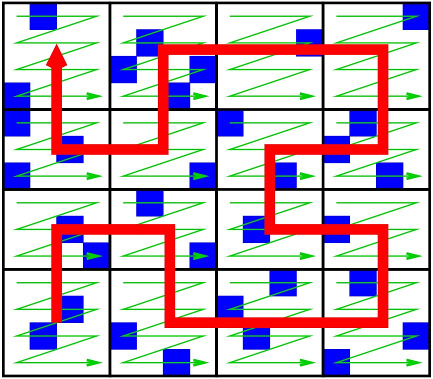


Other approaches: no blocking (Haase et al.), Morton Z-curves and bisection (Martone et al.), Z-curve within blocks (Buluç et al.), composition of low-level blocking (Vuduc et al.), ...

Ref.: Yzelman and Roose, “High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication”, IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Enhanced cache use: nonzero reorderings

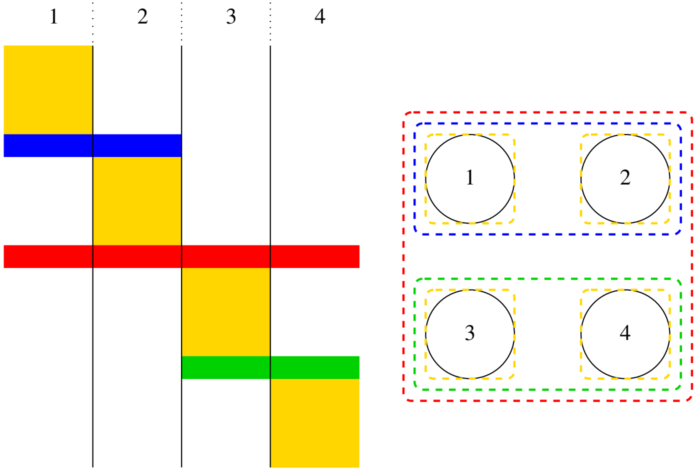
Blocking to cache subvectors, and **cache-oblivious traversals**.



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

Ref.: Yzelman and Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication", IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Enhanced cache use: matrix permutations



Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods by A. N. Yzelman & Rob H. Bisseling in SIAM Journal of Scientific Computation 31(4), pp. 3128-3154 (2009).

Enhanced cache use: matrix permutations

Practical gains:

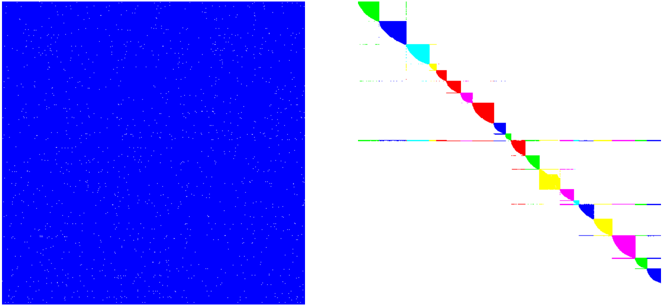


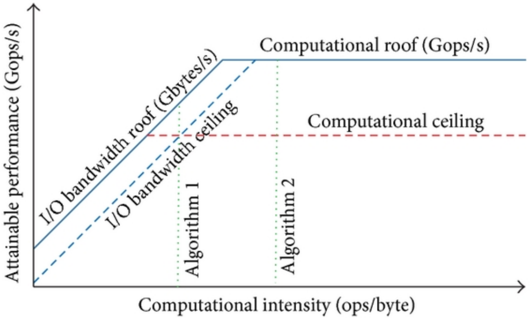
Figure: the Stanford link matrix (left) and its 20-part reordering (right).

Sequential execution using CRS on Stanford:

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

Ref.: Two-dimensional cache-oblivious sparse matrix-vector multiplication by A. N. Yzelman & Rob H. Bisseling in Parallel Computing 37(12), pp. 806-819 (2011).

Bandwidth



Theoretical turnover points: Intel Xeon E3-1225

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

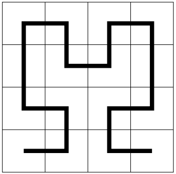
(Image taken from da Silva et al., DOI 10.1155/2013/428078, Creative Commons Attribution License)

Bandwidth

Consequence: compression leads to better performance.

- Coordinate format storage: $\Theta(3nz)$
- Compressed Row Storage (CRS): $\Theta(2nz + m + 1)$
- Bi-directional Incremental CRS: $\Theta(2nz + row\_jumps + 1)$

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



Need to consider the whole picture; good cache efficiency but no compression or compression but no cache optimisation? **No gain!**

Ref.: Yzelman and Bisseling, “A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve”, Progress in Industrial Mathematics at ECMI 2010, pp. 627-634 (2012).

Efficient bandwidth use

With BICRS you can

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

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

Ref.: Buluç, Fineman, Frigo, Gilbert, Leiserson (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (pp. 233-244). ACM.

Ref.: Yzelman and Bisseling (2009). Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. In SIAM Journal of Scientific Computation 31(4), pp. 3128-3154.

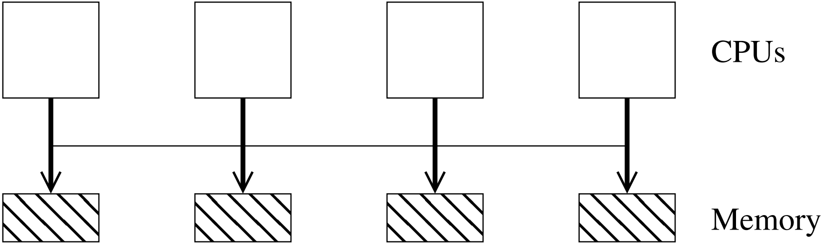
Ref.: Yzelman and Bisseling (2012). A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve". In Progress in Industrial Mathematics at ECMI 2010, pp. 627-634.

Ref.: Yzelman and Roose (2014). High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication. In IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31.

Ref.: Yzelman, A. N. (2015). Generalised vectorisation for sparse matrix: vector multiplication. In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. ACM.

NUMA

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

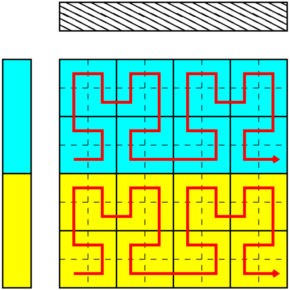


Memory access between sockets is slower, leading to *non-uniform memory access* (NUMA).

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,

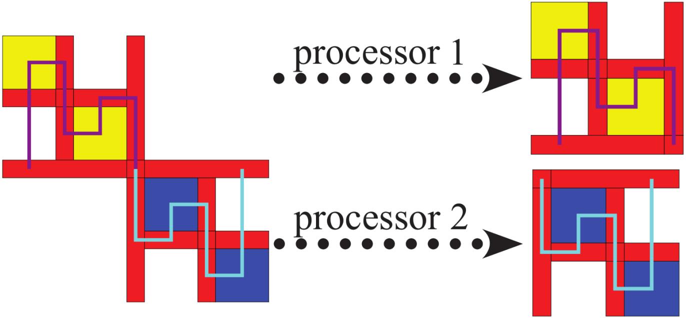


Ref.: Yzelman and Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication", IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

Two-dimensional data placement

Distribute row- *and* column-wise (individual nonzeroes):

- most work touches only local data,
- inter-process communication minimised by partitioning;
- incurs cost of partitioning.



Ref.: Yzelman and Roose, *High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication*, IEEE Trans. Parallel and Distributed Systems, doi:10.1109/TPDS.2013.31 (2014).

Ref.: Yzelman, Bisseling, Roose, and Meerbergen, *MulticoreBSP for C: a high-performance library for shared-memory parallel programming*, Intl. J. Parallel Programming, doi:10.1007/s10766-013-0262-9 (2014).

Results

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

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

Results

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

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

Average speedup on six large matrices:

	2 x 6	4 x 10	8 x 8
–, 1D fine-grained, CRS*	4.6	6.8	6.2
Hilbert, Blocking, 1D, BICRS*	5.4	19.2	24.6
Hilbert, Blocking, 2D, BICRS†	–	21.3	30.8

†: uses an updated test set. (Added for reference versus a good 2D algorithm.)

As NUMA scales up, 1D algorithms lose efficiency.

*: Yzelman and Roose, *High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication*, IEEE Trans. Parallel and Distributed Systems, doi:10.1109/TPDS.2013.31 (2014).

†: Yzelman, Bisseling, Roose, and Meerbergen, *MulticoreBSP for C: a high-performance library for shared-memory parallel programming*, Intl. J. Parallel Programming, doi:10.1007/s10766-013-0262-9 (2014).

Usability

Wish list:

- Performance and scalability.
- Better usability. Standardised API? Generalised Sparse BLAS:

GraphBLAS.org

- Interoperability with Big Data:

EYWA, Spark, Hadoop, DSLs, ...

- Interoperability with classic HPC:

MPI + { PThreads, Cilk, OpenMP, ...}

Ref.: Kepner & Gilbert, Linear Algebra in the Language of Linear Algebra, ISBN 978-0-898719-90-1, 2011

Ref.: Stepanov & McJones, The Elements of Programming, ISBN 978-0-321-63537-2, 2009

Ref.: Buluç & Gilbert, The Combinatorial BLAS, ICHPCA, 2011

Ref.: Zhang, Zalewski, Lumsdaine, Misurda, & McMillan, GBTL-CUDA, IPDPS, 2016

Ref.: Ekanadham, Horn, Kumar, Jann, Moreira, Pattnaik, Serrano, Tanase, & Yu, Graph programming interface (GPI), ACM ICCF, 2016

GraphBLAS

A ‘generalised’ semiring is given by

$$< D_1, D_2, D_3, D_4, \oplus, \otimes, 0, 1 >,$$

with

$$\begin{aligned} \oplus: &\quad D_3 \times D_4 \to D_4 \\ \otimes: &\quad D_1 \times D_2 \to D_3 \end{aligned}$$

These operators have to follow some basic rules, such as: $\oplus(a, b) = \oplus(b, a)$, $\oplus(\oplus(a, b), c) = \oplus(a, \oplus(b, c))$, $\otimes(\otimes(a, b), c) = \otimes(a, \otimes(b, c))$, $\otimes(a, \oplus(b, c)) = \oplus(\otimes(a, b), \otimes(b, c))$, $\oplus(a, 0) = \oplus(0, a) = a$, $\otimes(a, 1) = \otimes(1, a) = a$, $\otimes(a, 0) = \otimes(0, a) = 0$.

If these are true, (sparse) linear algebra ‘works’; we can apply all of our performance optimisations regardless of the operators selected!

Bridging HPC and Big Data

Platforms like Spark allow programmers to ignore data placement issues, thus negatively impacting performance. It's a classic tradeoff:

automatic mode vs. direct mode
ease-of-use vs. performance

Ref.: Valiant, L. G. (1990). A bridging model for parallel computation. Communications of the ACM, 33(8).

Bridging HPC and Big Data

Platforms like Spark allow programmers to ignore data placement issues, thus negatively impacting performance. It's a classic tradeoff:

automatic mode vs. direct mode

ease-of-use vs. performance

Ref.: Valiant, L. G. (1990). A bridging model for parallel computation. Communications of the ACM, 33(8).

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 direct mode;
- Set up and enable inter-process RDMA communications.

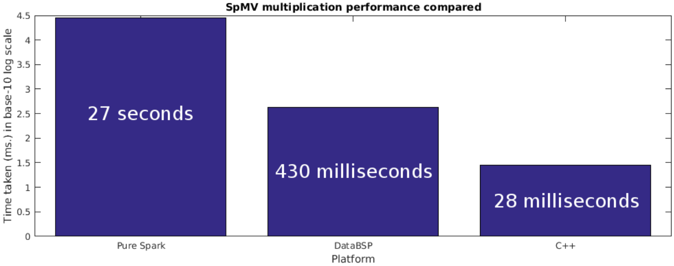
Both usable *and* performant!

Bridging Big Data and HPC

We have a shared-memory prototype. Preliminary results:

- SpMM multiply, SpMV multiply, and basic vector operations;
- one machine learning application, plus one on graph analysis.

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



This is ongoing work. Performance will be improved, functionality extended.

Conclusions and future work

We know how to do fast sparse computations

- **use same techniques for graph computing**.

Future work:

- faster partitioning to enable scalable 2D sparse computations,
- sparse power kernels,
- symmetric matrix support,
- hypergraph and sparse tensor computations,
- support various hardware and execution platforms (Hadoop?).

The high performance (non-generalised) SpMV multiplication codes are free:

`http://albert-jan.yzelman.net/software#SL`

Thank you!

Backup slides

GraphBLAS

A working example:

```
#include <graphblas.hpp>
int main() {
  const size_t num_cities = ... //some input matrix size
  grb::init();
  grb::Matrix< double > distances( num_cities, num_cities );
  grb::build( distances, ... ); //input data from file
                                //or memory
  grb::Vector< double > x( num_cities ), y( num_cities );
  grb::set( x, 0.0, 4 );   //set city number 4 to
                           //have distance 0.0
  ...
```

GraphBLAS

A working example (continued):

```
...
//declare an alternative semiring on doubles:
grb::Semiring< double, double, double, double,
               grb::operators::min,      //‘plus’
               grb::operators::add,      //‘multiply’
               grb::identities::infinity //‘0’
               grb::identitites::zero    //‘1’
> ring;

//calculate the shortest distances from all cities to
//city #4, allowing only a single path
grb::mxv( y, distances, x, ring );
...
```

GraphBLAS

A working example (continued):

```
  ...
  //calculate the shortest distances from all cities to
  //city #4, allowing only a single path
  grb::mxv( y, distances, x, ring );

  //calculate the shortest distances from all cities to
  //city #4, allowing two ‘hops’
  grb::mxv( x, distances, y, ring );

  //example output via iterators and exit:
  writeResult( x.cbegin(), x.cend(), ... );
  grb::finalize();
  return 0;
}
```

Results: cross platform

Cross platform results over 24 matrices:

	Structured	Unstructured	Average
Intel Xeon Phi	21.6	8.7	15.2
2x Ivy Bridge CPU	23.5	14.6	19.0
NVIDIA K20X GPU	16.7	13.3	15.0

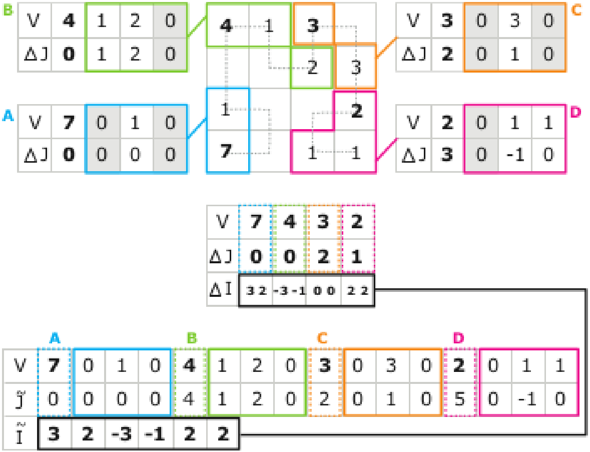
no one solution fits all.

If we must, some generalising statements:

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

Ref.: Yzelman, A. N. (2015). Generalised vectorisation for sparse matrix: vector multiplication. In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms. ACM.

Vectorised BICRS



B

V	**4**	1	2	0
ΔJ	**0**	1	2	0

C

V	**3**	0	3	0
ΔJ	**2**	0	1	0

A

V	**7**	0	1	0
ΔJ	**0**	0	0	0

D

V	**2**	0	1	1
ΔJ	**3**	0	-1	0

V	**7**	**4**	**3**	**2**
ΔJ	**0**	**0**	**2**	**1**
ΔI	**3 2**	**-3 -1**	**0 0**	**2 2**

	A				B				C				D			
V	**7**	0	1	0	**4**	1	2	0	**3**	0	3	0	**2**	0	1	1
J̃	0	0	0	0	4	1	2	0	2	0	1	0	5	0	-1	0
Ĩ	**3**	**2**	**-3**	**-1**	**2**	**2**										