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%?

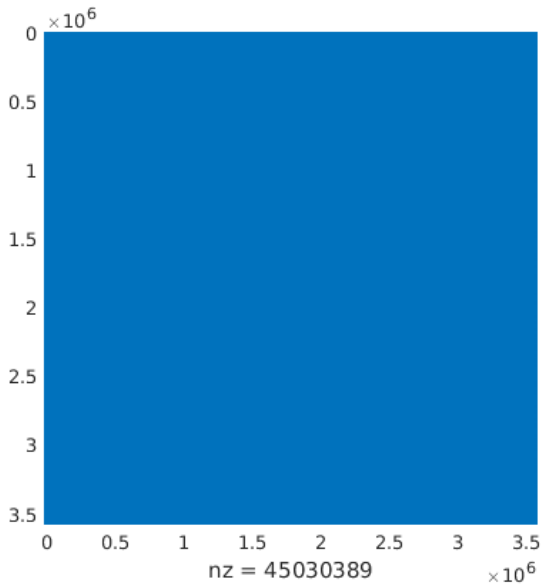


Figure: Wikipedia link matrix ('07) with on average $\approx$ 12.6 nonzeroes per row.

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:

cache misses

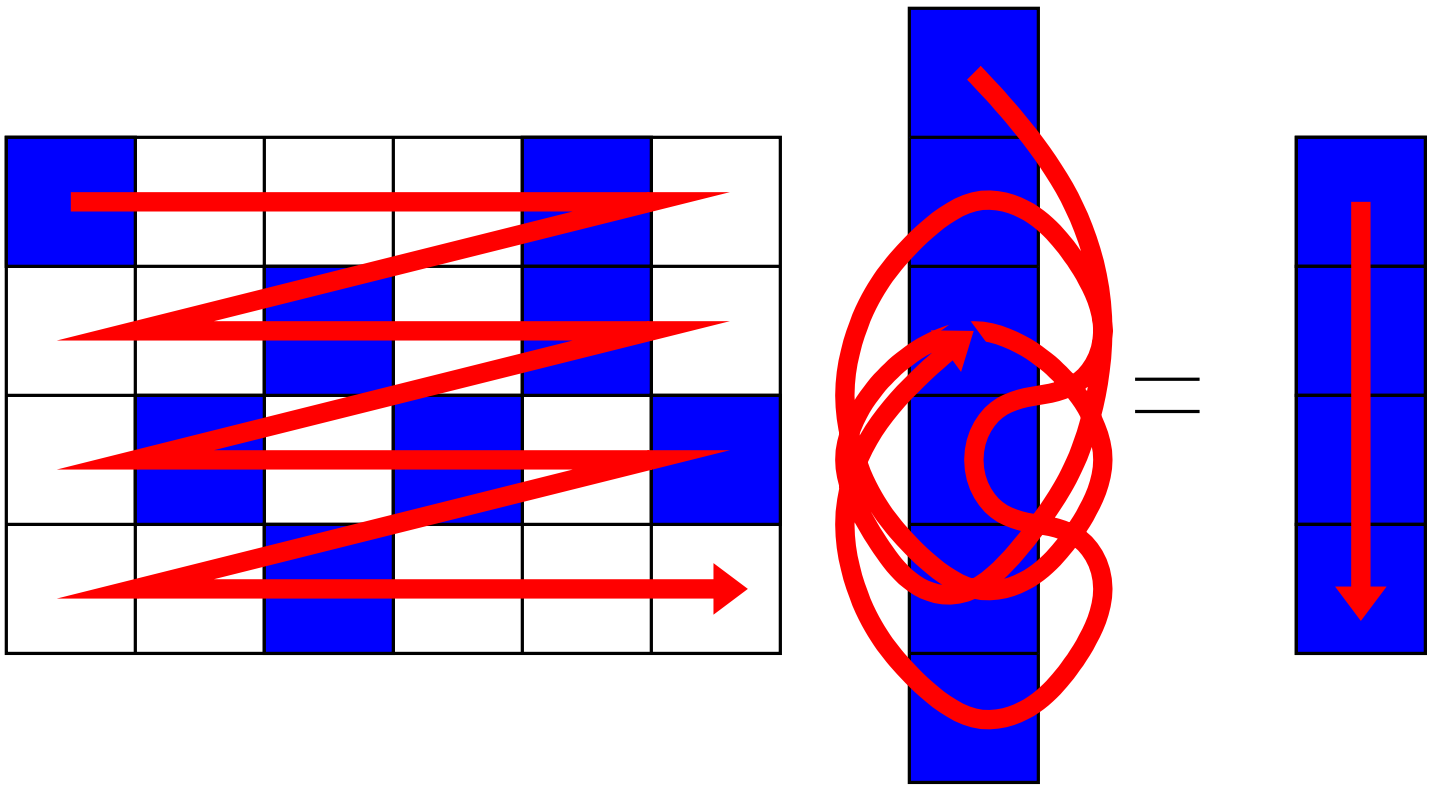
==

communication volume

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

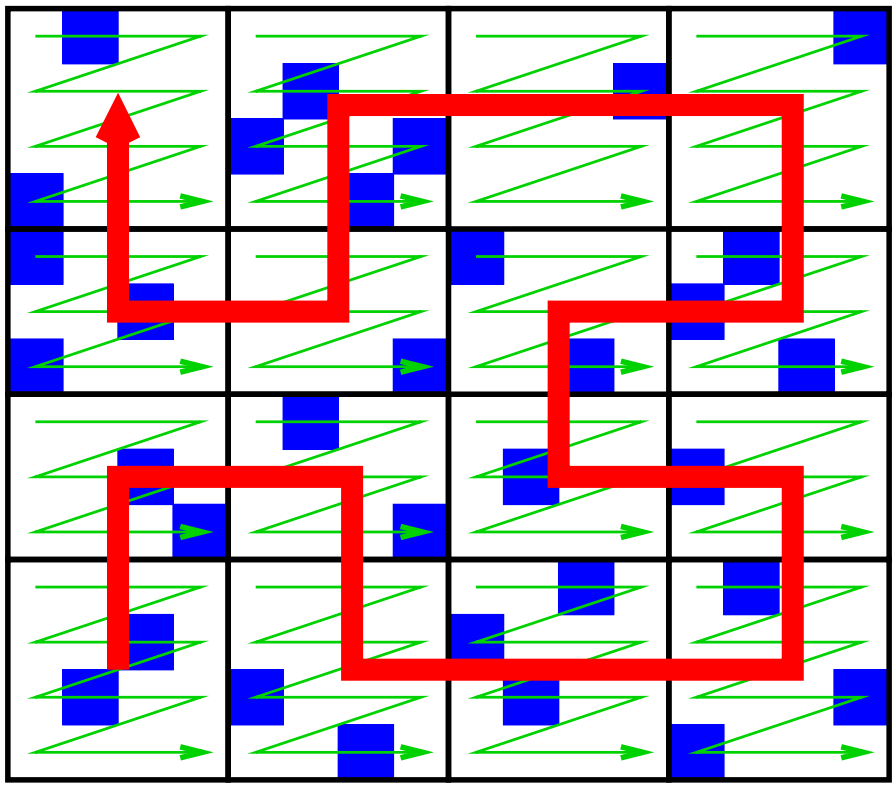
Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:



Accesses on the input vector are completely unpredictable.

Enhanced cache use: nonzero reorderings

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

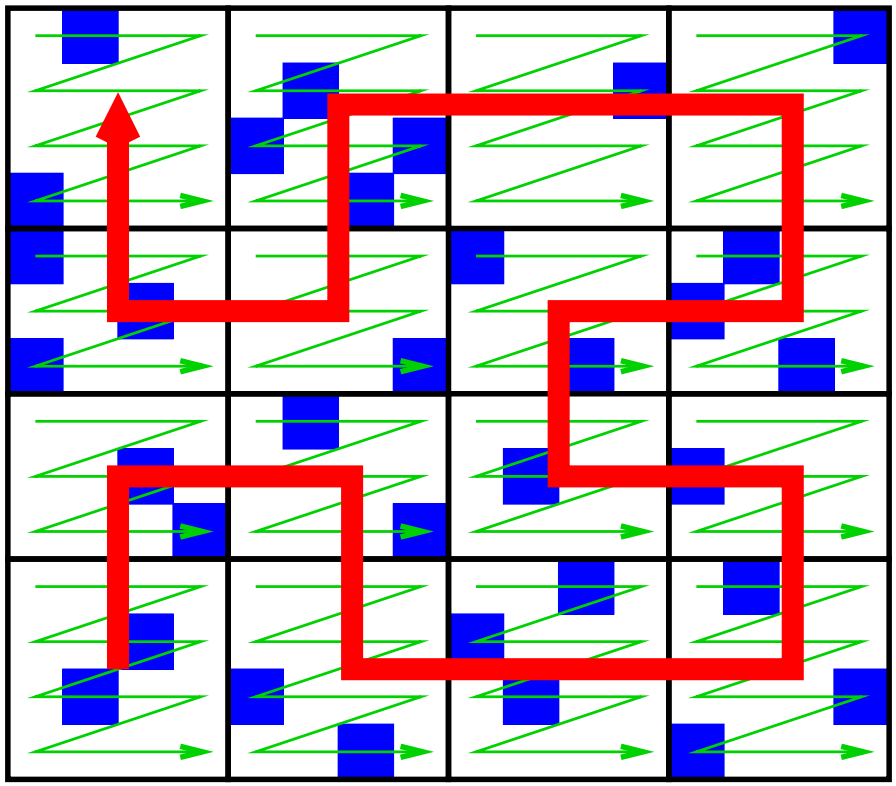


Other approaches: no blocking (Haase et al.), Morton Z-curves and bisection (Martone et al.), Z-curve within blocks (Buluç et al.), composition of low-level blocking (Vuduc et al.), ...

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

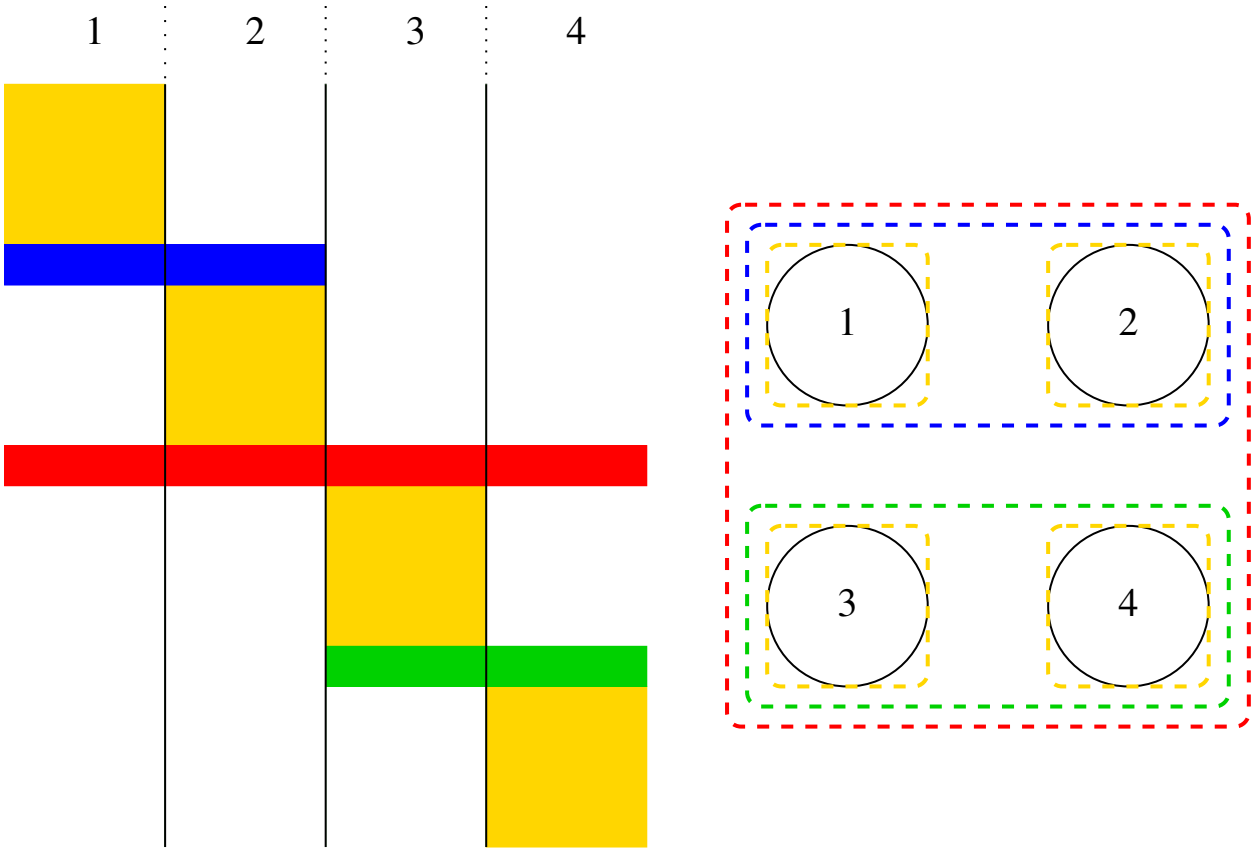
Blocking to cache subvectors, and **cache-oblivious traversals**.



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



(Upper bound on) the number of cache misses: $\sum_i (\lambda_i - 1)$

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

$$\text{cache misses} \leq \sum_i (\lambda_i - 1) = \text{communication volume}$$

- Lengauer, T. (1990). Combinatorial algorithms for integrated circuit layout. Springer Science & Business Media.
- Çatalyürek, Ü. V., & Aykanat, C. (1999). Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems, 10(7), 673-693.
- Çatalyürek, Ü. V., & Aykanat, C. (2001). A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices. In IPDPS (Vol. 1, p. 118).
- Vastenhouw, B., & Bisseling, R. H. (2005). A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM review, 47(1), 67-95.
- Bisseling, R. H., & Meesen, W. (2005). Communication balancing in parallel sparse matrix-vector multiplication. Electronic Transactions on Numerical Analysis, 21, 47-65.

Should we program shared-memory as though it were distributed?

Enhanced cache use: matrix permutations

Practical gains:

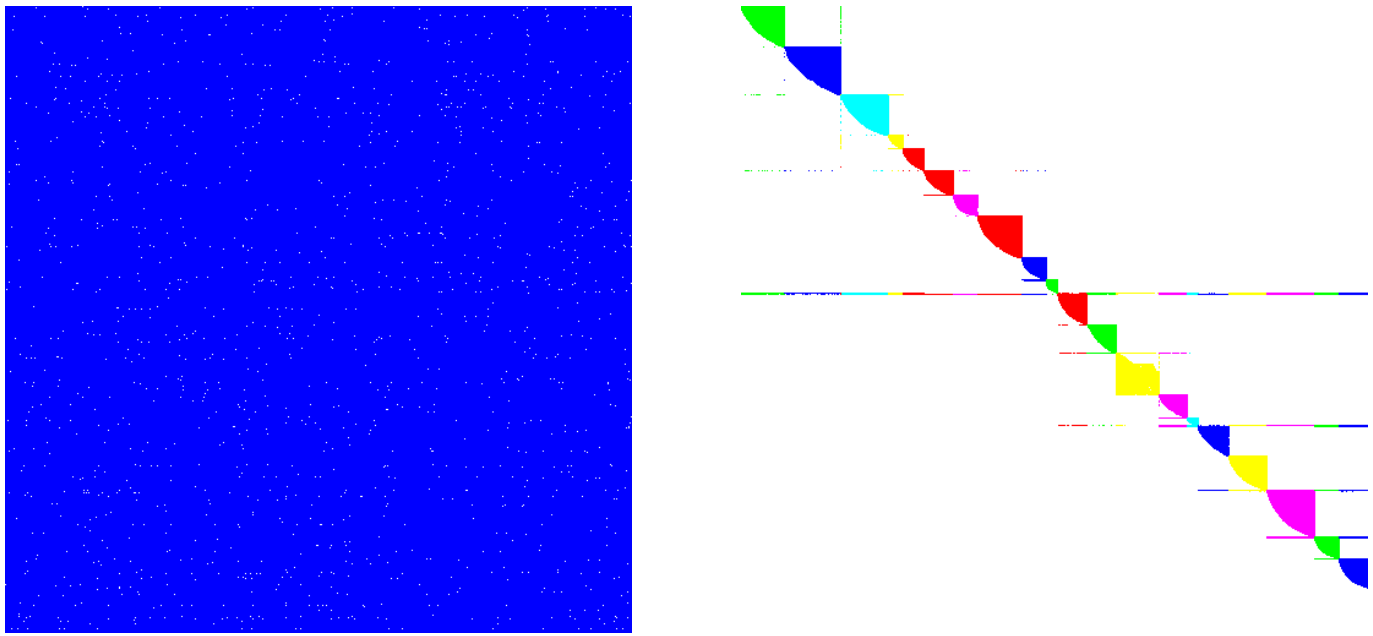


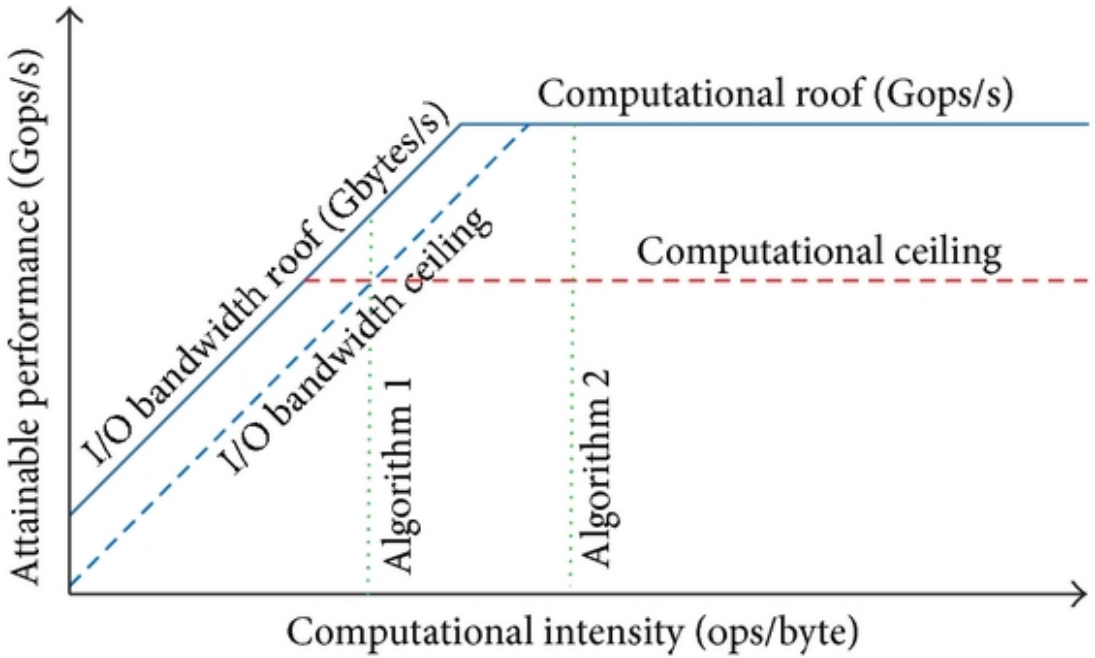
Figure: the Stanford link matrix (left) and its 20-part reordering (right).

Sequential execution using CRS on Stanford:

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

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



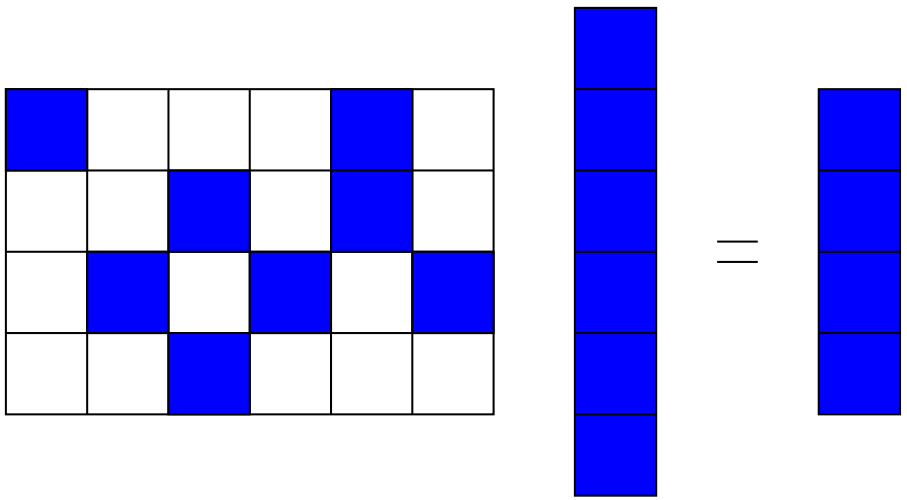
Theoretical turnover points: Intel Xeon E3-1225

- 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

Exploiting sparsity through computation using only nonzeroes:



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

for $k = 0$ **to** $nz - 1$

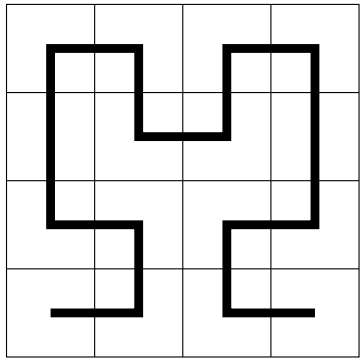
$$y_{i_k} := y_{i_k} + v_k \cdot x_{j_k}$$

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

$\Theta(3nz)$ storage. CRS: $\Theta(2nz + m)$.

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):

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

Storage requirements, allowing **arbitrary traversals**:

$$\Theta(2nz + row\_jumps + 1).$$

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, 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.

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.

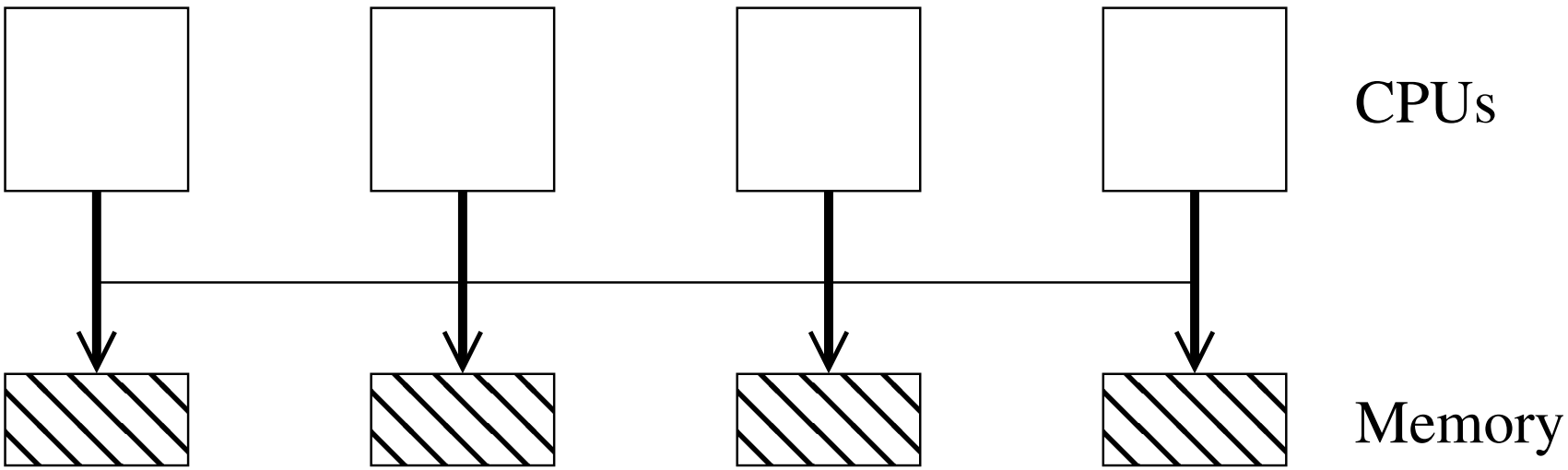
Ref.: Yzelman and Bisseling (2012). A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve". In Progress in Industrial Mathematics at ECMI 2010, pp. 627-634.

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

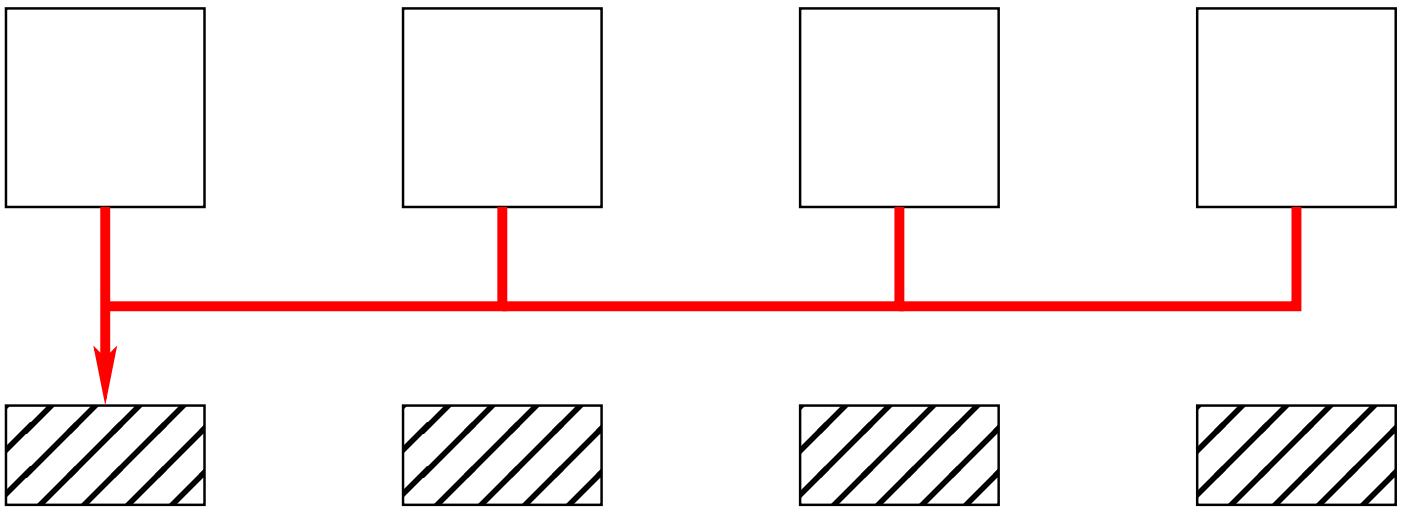
Each socket has **local** main memory where access is **fast**.



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

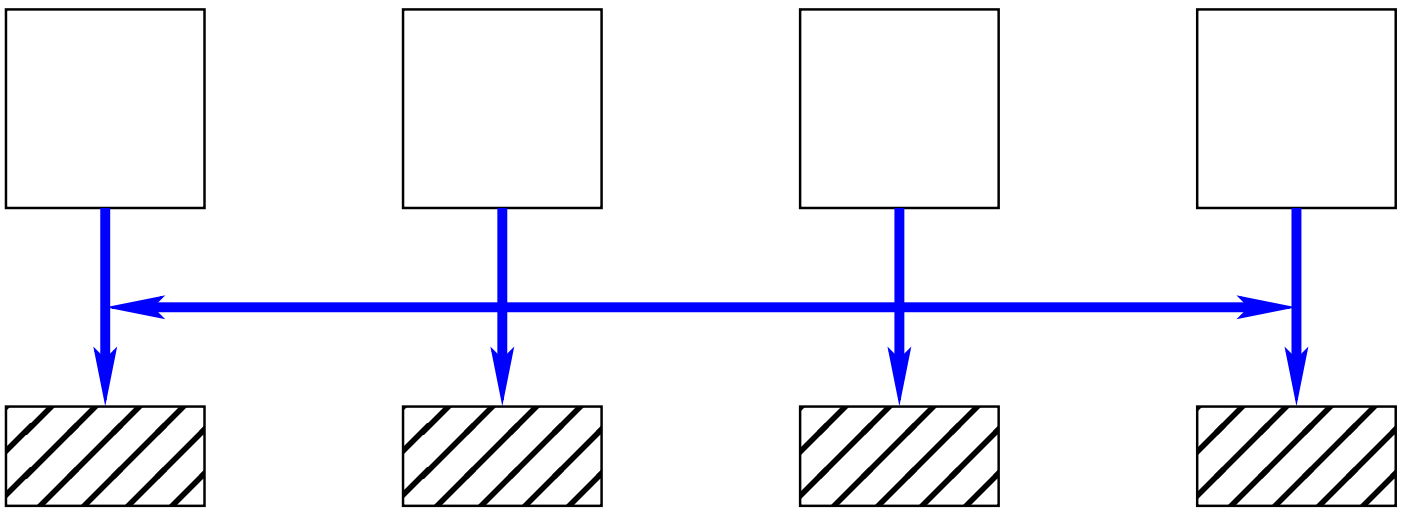
NUMA

Access to only one socket: limited bandwidth



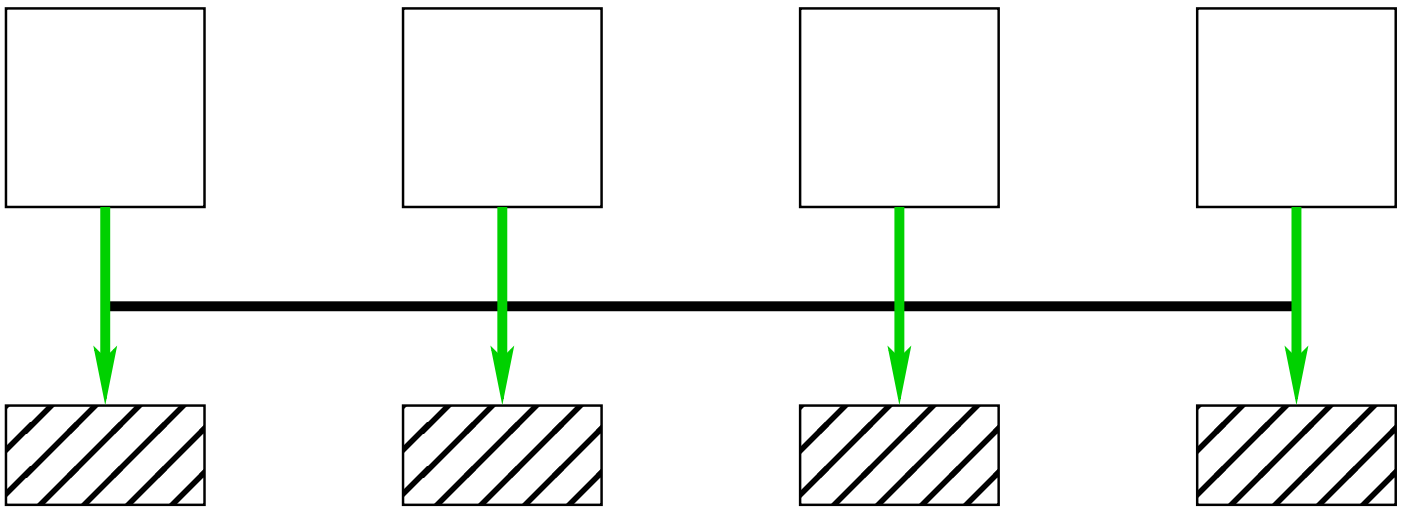
NUMA

Interleave memory pages across sockets: emulate uniform access



NUMA

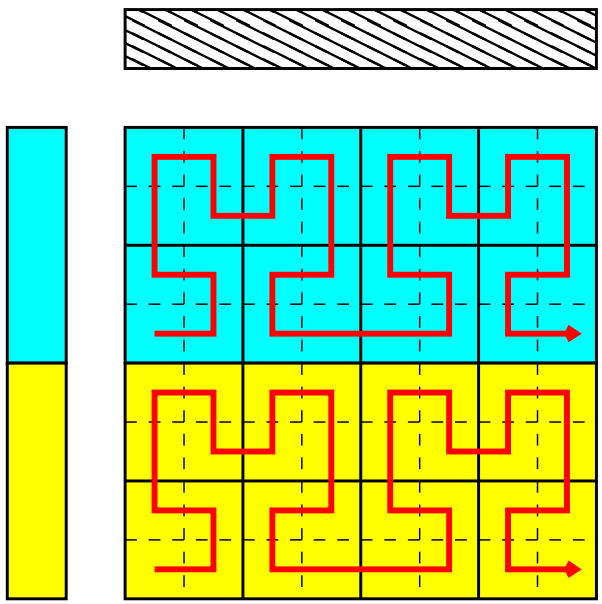
Explicit data placement on sockets: best performance



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,

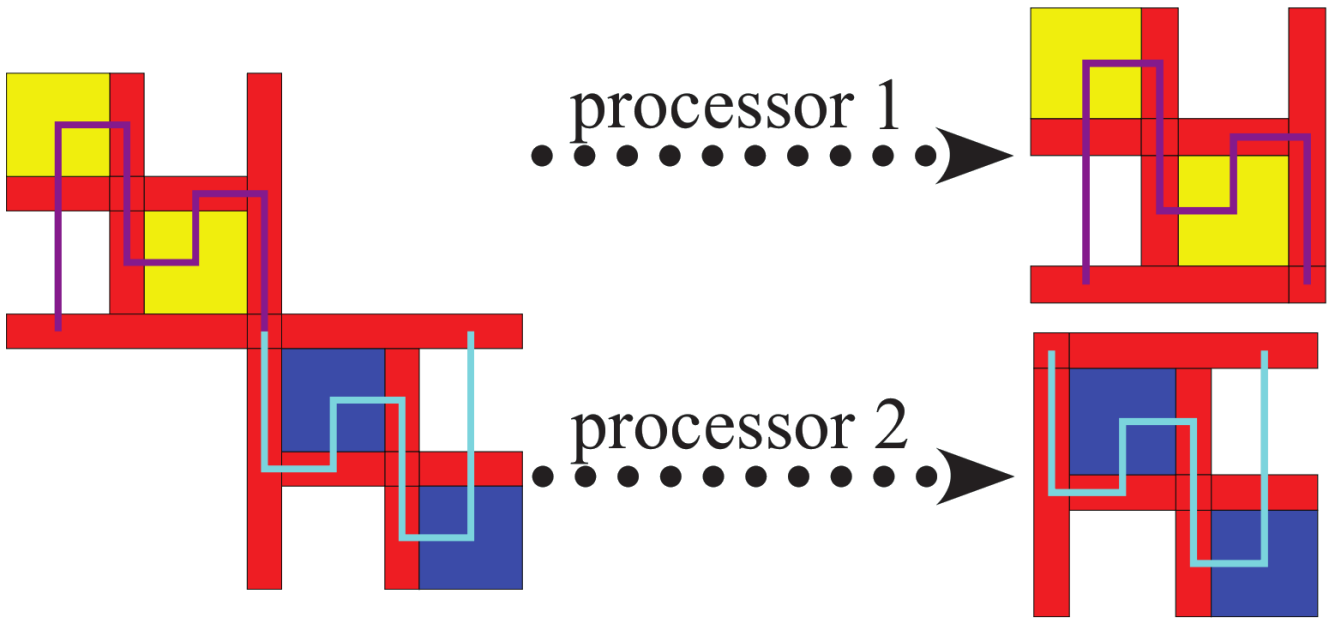


Ref.: Yzelman and Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication", IEEE Transactions on Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2014).

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: $\approx$ 22x.

	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

Average speedup on six large matrices.

† uses an updated test set, added for reference versus a good 2D algorithm.

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

*: 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

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!)

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> val A = sc.textFile( "A.txt" ).map( x => x.split(" ") 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.

Ref.: Zaharia, M. (2013). An architecture for fast and general data processing on large clusters. Dissertation, UCB.

Bridging HPC and Big Data

Platforms like Spark essentially perform PRAM simulation:

automatic mode vs. direct mode

easy-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 essentially perform PRAM simulation:

automatic mode vs. direct mode
easy-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:

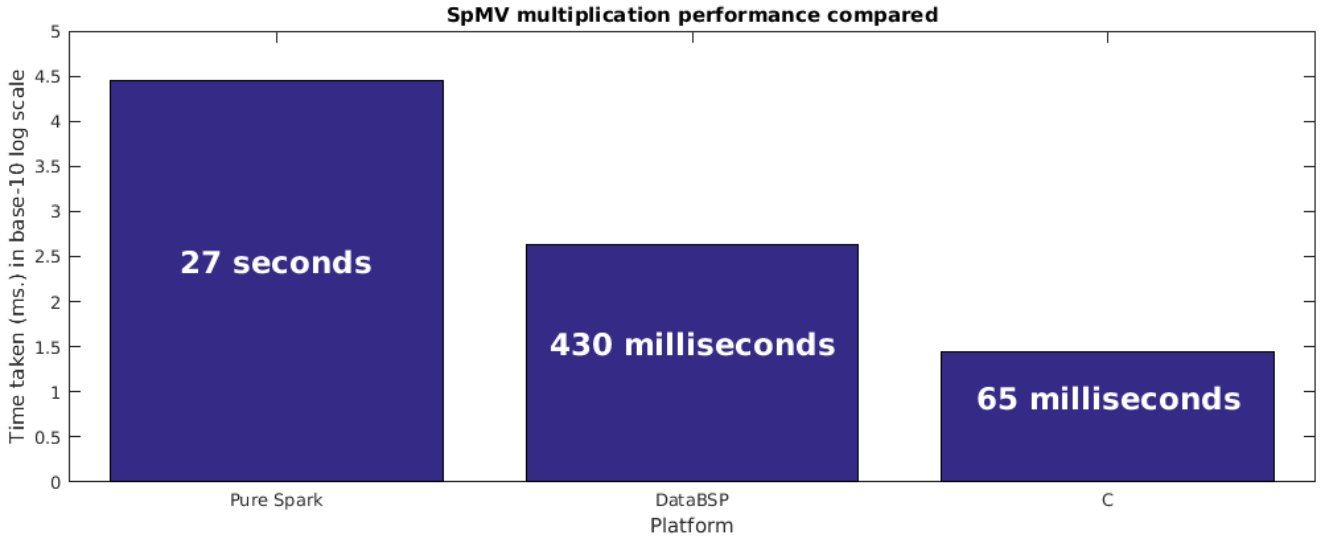
- Spark I/O via native RDDs and native Scala interfaces;
- Rely on serialisation and the JNI to switch to C;
- Intercept Spark's execution model to switch to SPMD;
- Set up and enable inter-process RDMA communications.

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.




```
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 );                                  //type: DenseVector (returns y)
scala> val y_rdd = toRDD( sc, y );                          //type: RDD[(Long, Double)]
```

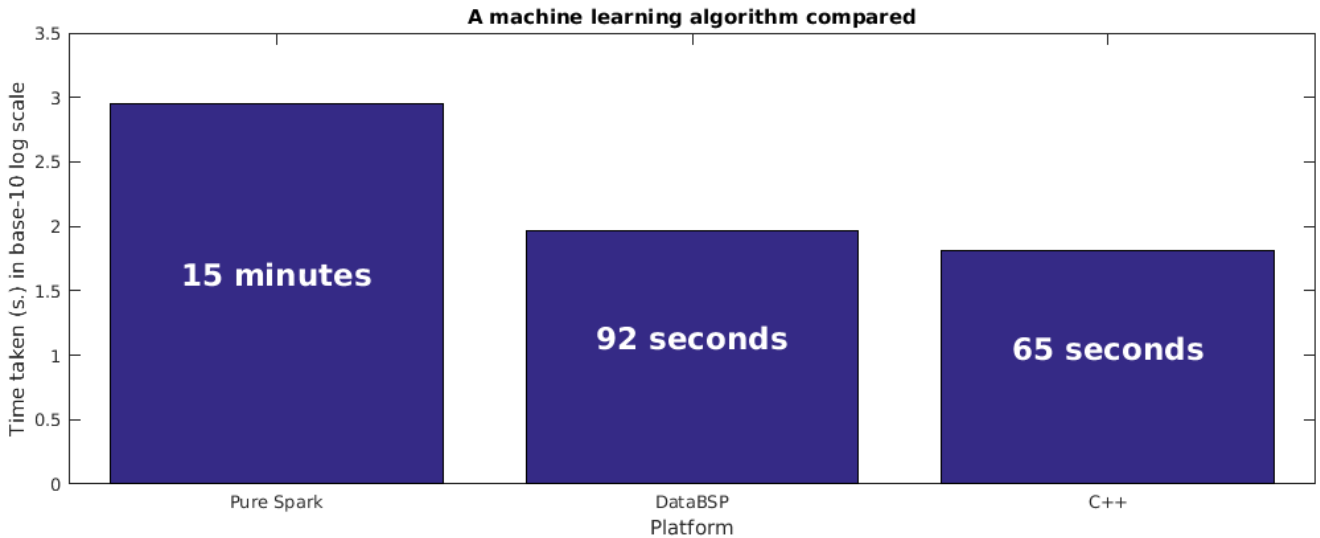
This is ongoing work; we are improving performance, extending functionality.

Bridging HPC and Big Data

Some preliminary shared-memory results, Spark.

- a machine learning application:

Training stage, internal test data.




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

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:

	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