



# STRATEGIES FOR PARALLEL SpMV MULTIPLICATION

ALBERT-JAN YZELMAN

JUNE 2012

# ACKNOWLEDGEMENTS

This presentation outlines the pre-print

*'High-level strategies for parallel shared-memory sparse matrix-vector multiplication'*, tech. rep. TW-614, KU Leuven (2012), which has been submitted for publication.

This is joint work of Albert-Jan Yzelman and Dirk Roose, Dept. of Computer Science, KU Leuven, Belgium.



# ACKNOWLEDGEMENTS

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology (IWT), in the framework of the Flanders ExaScience Lab, part of Intel Labs Europe.



# SUMMARY

- 1 SUMMARY
- 2 CLASSIFICATION
- 3 STRATEGIES
- 4 EXPERIMENTS
- 5 CONCLUSIONS AND OUTLOOK

# CENTRAL QUESTION

Given a *sparse*  $m \times n$  matrix  $A$  and an  $n \times 1$  input vector  $x$ .

How to calculate

$$y = Ax$$

on a shared-memory parallel computer, as fast as possible?

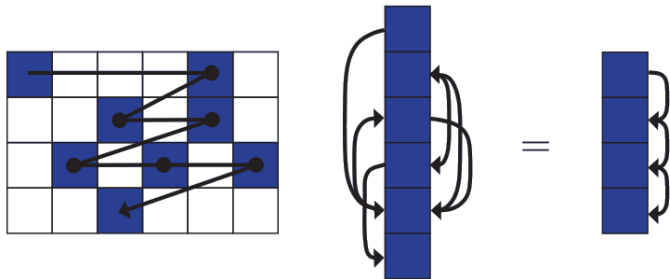
# CENTRAL OBSTACLES

Three obstacles oppose an efficient shared-memory parallel sparse matrix–vector (SpMV) multiplication kernel:

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

# CENTRAL OBSTACLES

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



Accesses on the input vector are completely unpredictable.

# CENTRAL OBSTACLES

The *arithmetic intensity* of an SpMV multiply lies between

$$\frac{2}{4} \text{ and } \frac{2}{5} \text{ flop per byte.}$$

On an 8-core 2.13 GHz (with AVX), and 10.67 GB/s DDR3:

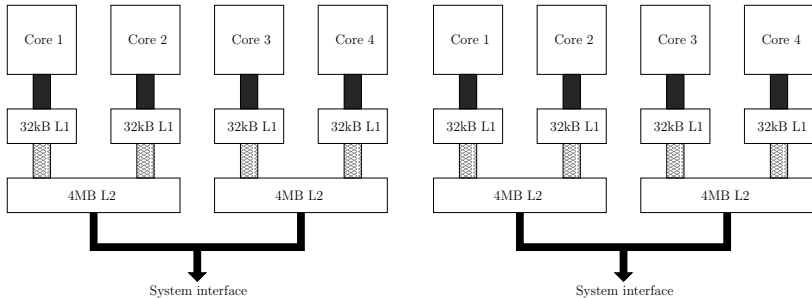
	CPU speed	Memory speed
1 core	$4 \cdot 2.13 \cdot 10^9 \text{ nz/s}$	$\frac{2}{5} \cdot 10.67 \cdot 10^9 \text{ nz/s}$
8 cores	$32 \cdot 2.13 \cdot 10^9 \text{ nz/s}$	$\frac{2}{5} \cdot 10.67 \cdot 10^9 \text{ nz/s}$

The CPU-speed exceeds by far the memory speed.



# CENTRAL OBSTACLES

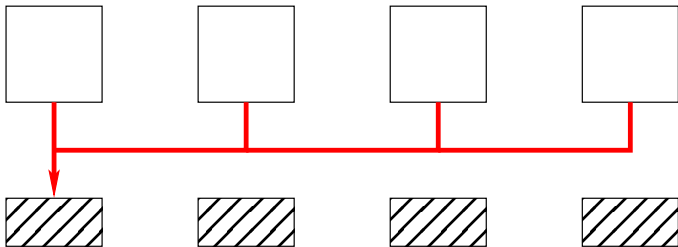
E.g., a dual-socket machine, with two quad-core processors:



Pairs of cores may have different bandwidths.

# CENTRAL OBSTACLES

If each processor moves data from and to the same memory element, the effective bandwidth is shared.



# STARTING POINT

Assuming a row-major order of nonzeros:

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

CRS storage:

$$A = \begin{cases} V & [4 \ 1 \ 3 \ 2 \ 3 \ 1 \ 2 \ 7 \ 1 \ 1] \\ J & [0 \ 1 \ 2 \ 2 \ 3 \ 0 \ 3 \ 0 \ 2 \ 3] \\ \hat{I} & [0 \ 3 \ 5 \ 7 \ 10] \end{cases}$$

Kernel:

```
for  $i = 0$  to  $m - 1$  do  
  for  $k = \hat{I}_i$  to  $\hat{I}_{i+1} - 1$  do  
    add  $V_k \cdot x_{J_k}$  to  $y_i$ 
```

# STARTING POINT

Assuming a row-major order of nonzeros:

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

CRS storage:

$$A = \begin{cases} V & [4 \ 1 \ 3 \ 2 \ 3 \ 1 \ 2 \ 7 \ 1 \ 1] \\ J & [0 \ 1 \ 2 \ 2 \ 3 \ 0 \ 3 \ 0 \ 2 \ 3] \\ \hat{I} & [0 \ 3 \ 5 \ 7 \ 10] \end{cases}$$

```
#omp parallel for private( i, k ) schedule( dynamic, 8 )
```

```
for  $i = 0$  to  $m - 1$  do  
  for  $k = \hat{I}_i$  to  $\hat{I}_{i+1} - 1$  do  
    add  $V_k \cdot x_{J_k}$  to  $y_i$ 
```

# RESULTS

Methods	DL-2000	DL-580	DL-980
OpenMP CRS (1D)	2.6 (8)	3.5 (8)	3.0 (8)
CSB (1D)	4.9 (12)	9.9 (30)	8.6 (32)
Interleaved CSB (1D)	6.0 (12)	18.2 (40)	17.7 (64)
pOSKI (1D)	5.4 (12)	14.8 (40)	12.2 (64)
Row-distr. block CO-H (1D)	7.0 (12)	24.7 (40)	34.0 (64)
Fully distributed (2D)	4.0 (12)	8.3 (40)	6.9 (32)
Distr. CO-SBD, $qp = 32$ (2D)	4.0 (8)	7.4 (32)	6.4 (32)
Distr. CO-SBD, $qp = 64$ (2D)	4.2 (8)	7.0 (16)	6.8 (64)
Block CO-H+ (ND)	2.5 (12)	3.1 (16)	3.2 (16)

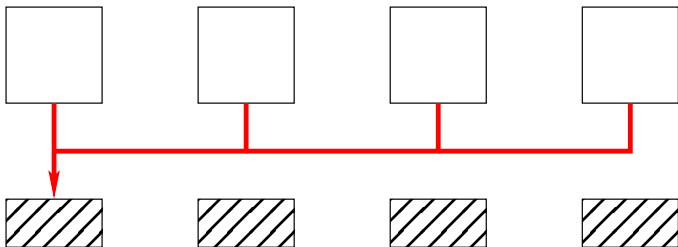
Average speedups relative to sequential CRS. Actual number of threads used is in-between brackets.

# CLASSIFICATION

- 1 SUMMARY
- 2 CLASSIFICATION**
- 3 STRATEGIES
- 4 EXPERIMENTS
- 5 CONCLUSIONS AND OUTLOOK

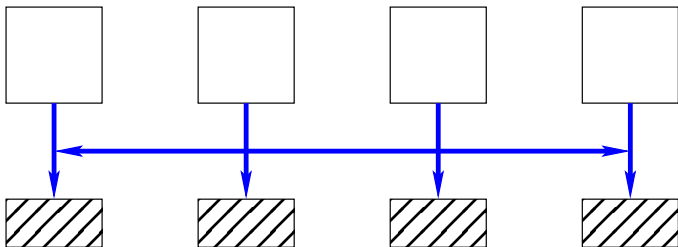
# DISTRIBUTION TYPES

**Implicit** distribution, centralised local allocation:



# DISTRIBUTION TYPES

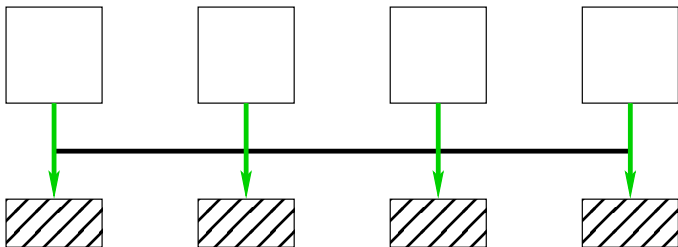
**Implicit** distribution, centralised interleaved allocation:





# DISTRIBUTION TYPES

**Explicit** distribution, distributed local allocation:



If each processor moves data from and to its own unique memory element, *the bandwidth multiplies with the number of available elements.*

# DISTRIBUTION TYPES

Parallelisation maps nonzeros to processes:

$$\pi_A : \{0, \dots, m-1\} \times \{0, \dots, n-1\} \rightarrow \{0, \dots, p-1\};$$

process  $\pi_A(i, j)$  performs  $y_i = y_i + a_{ij}x_j$ .

Vectors *can* be distributed similarly:

$$\begin{aligned} \pi_y &: \{0, \dots, m-1\} \rightarrow \{0, \dots, p-1\}, \quad \text{and} \\ \pi_x &: \{0, \dots, n-1\} \rightarrow \{0, \dots, p-1\}. \end{aligned}$$

# DISTRIBUTION TYPES

ND (no distribution):  $\pi_x$  and  $\pi_y$  are left undefined;  
maximum freedom for choosing  $\pi_A$ .

1D distribution:  $\pi_A(i, j)$  depends on either  $i$  or  $j$ ;  
typically,  $\pi_A(i, j) = \pi_A(i) = \pi_y(i)$ .

2D distribution:  $\pi_A, \pi_x$  and  $\pi_y$  are all defined.

---

Implicit distribution: process  $s$  performs only computations  
with nonzeros  $a_{ij}$  s.t.  $\pi_A(i, j) = s$ .

Explicit distr. (of  $A$ ): process  $s$  additionally allocates storage  
for those  $a_{ij}$  for which  $\pi(i, j) = s$ , on  
locally fast memory. Explicit distribution  
of  $x$  or  $y$  is similar.

# CACHE-OPTIMISATION

- Cache-aware: auto-tuning, or coding for specific architectures
- Cache-oblivious: runs well regardless of architecture
- Matrix-aware: exploits structural properties of  $A$  (usually combined with auto-detection within cache-aware schemes)

# PARALLELISATION TYPE

Coarse-grained:  $p$  equals the available number of units of execution (UoEs; e.g., cores).

Fine-grained:  $p$  is much larger than the number of available UoEs.

A coarse grainsize easily incorporates explicit distributions; a fine grainsize spends less effort to attain load-balance.

# STRATEGIES

- 1 SUMMARY
- 2 CLASSIFICATION
- 3 STRATEGIES**
- 4 EXPERIMENTS
- 5 CONCLUSIONS AND OUTLOOK

# No VECTOR DISTRIBUTION

We define  $\pi_A$  such that for all  $s \in \{0, \dots, p-1\}$ ,

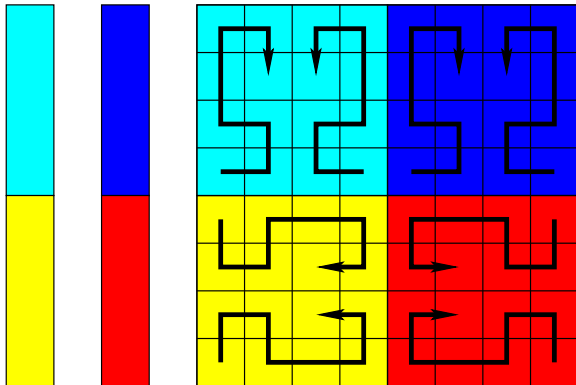
$$|\{a_{ij} \in A \mid \pi(i, j) = s\}| = \lfloor nz/p \rfloor + \begin{cases} 1, & \text{if } s \bmod p < nz \bmod p \\ 0 & \text{otherwise} \end{cases} ;$$

i.e., perfect load-balance.

Which nonzeros go where, and the order of processing of nonzeros, is determined by the Hilbert-curve.

# NO VECTOR DISTRIBUTION

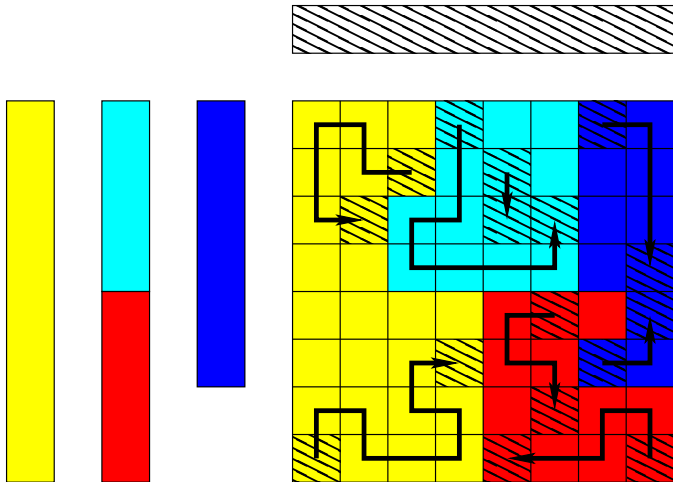
$x$  is implicitly distributed (using interleaved allocation):





# No VECTOR DISTRIBUTION

$y$  is explicitly distributed, but must be replicated:



# NO VECTOR DISTRIBUTION

This strategy is called block CO-H+, and is

- non-distributed,
- implicit,
- fully cache-oblivious, and
- coarse-grained.

# 1D DISTRIBUTION

Many existing methods fall in this distribution type.

The openMP parallelisation shown at the start is:

- 1D,
- implicit,
- not optimised for cache,
- fine-grained.

the parallel Optimised Sparse Kernel Interface (pOSKI, by Byun et al., UCB) is better as it is (by default):

- 1D,
- explicit,
- cache- and matrix-aware optimised (auto-tuned),
- coarse-grained.

# 1D DISTRIBUTION

For a good fine-grained 1D method we refer to *Compressed Sparse Blocks* (CSB, by Buluç et al., UCSB/LBNL/MIT):

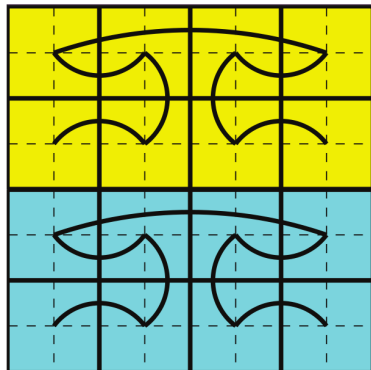
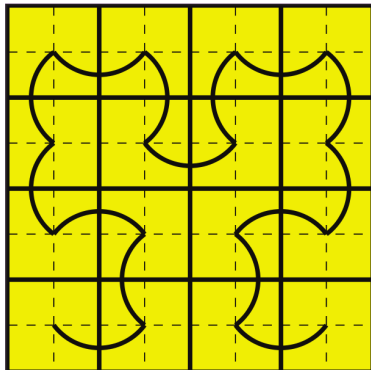
- 1D,
- implicit (originally locally allocated, but can be interleaved),
- cache-oblivious,
- fine-grained.

We also constructed the *row-distributed block CO-H* strategy, which is:

- 1D,
- explicit,
- fully cache-oblivious,
- coarse-grained.

# 1D DISTRIBUTION

Sketch of the row-distributed block CO-H strategy:

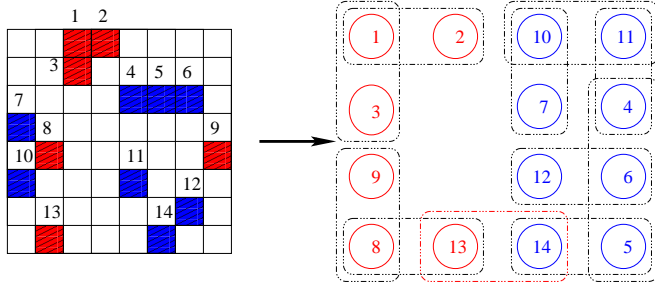


(Left:  $p = 1$ , right:  $p = 2$ .)

This method uses  $\pi_A = \pi_y$ , allocates  $x$  in an interleaved fashion, and allocates  $A$  and  $y$  in a distributed, explicit way.

# 2D DISTRIBUTION

Pre-processing: full matrix partitioning.



The communication metric minimised (the  $\lambda - 1$ -metric), is exact (see Yzelman & Bisseling, 2009). Available software:

- Mondriaan (Bisseling et al., UU),
- Zoltan (Boman et al., Sandia),
- PaToH (Çatalyürek & Aykanat, Ohio State), and
- Scotch (Pellegrini, Bordelais).

# 2D DISTRIBUTION

Full matrix partitioning yields the three  $\pi_{\{A,x,y\}}$  maps.

These are used in an explicit way; each process allocates its own local  $A^{(s)}$  matrix and  $x^{(s)}$  and  $y^{(s)}$  vectors.

Local vectors overlap with each other; the number of redundantly stored elements is given by the  $(\lambda - 1)$ -metric.

That is also the measure for communication, and is minimised by partitioning.

# 2D DISTRIBUTION

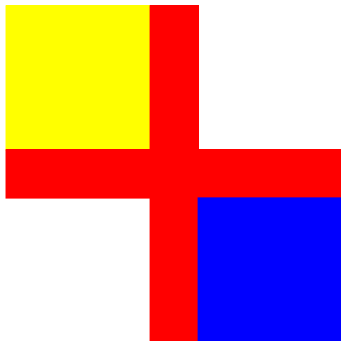
The classical parallel SpMV algorithm consists out of three steps and one synchronisation. Each process  $s$  executes:

- copy remote input vector elements (all  $j$  with  $\pi_x(j) \neq s$ ),
- perform a local (optimised) SpMV multiply, *synchronise*,
- get and add remote contributions to local output vector.



# 2D DISTRIBUTION

In the paper, we augment this strategy with the Separated Block Diagonal (SBD) form.



We denote the local indices of the red separator cross by  $r_s^-, r_s^+, c_s^-, c_s^+$  and the local matrix size by  $m^s \times n^s$ .

# 2D DISTRIBUTION

In the paper, we augment this strategy with the Separated Block Diagonal (SBD) form.

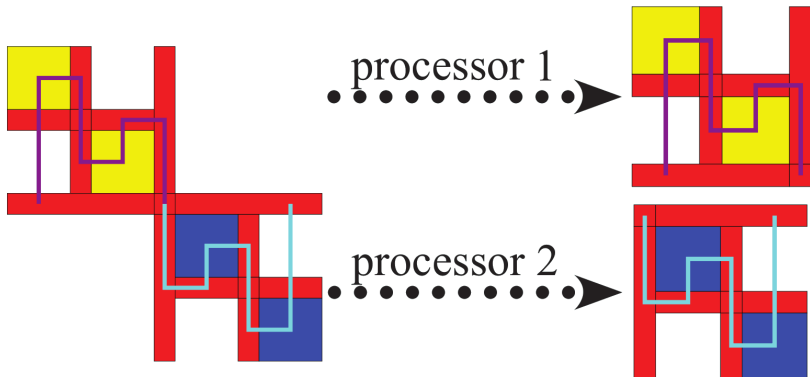
Each processor  $s \in \{0, 1, \dots, p - 1\}$  executes:

- 1: **for**  $j = c_-^s$  to  $c_+^s$  **do**
- 2:   get  $x_j^s$  from remote process
- 3:
- 4:   calculate  $y^s = A^{(s)}x^s$
- 5:   synchronise
- 6: **for**  $i = r_-^s$  to  $r_+^s$  **do**
- 7:   write  $y_i^s$  to remote process

We can add in locally refined SBD forms as well.

# 2D DISTRIBUTION

Locally refined SBD:



This is the full explicitly 2D distributed CO-SBD strategy.

# 2D DISTRIBUTION

In the paper, we augment this strategy with the Separated Block Diagonal (SBD) form.

Each processor  $s \in \{0, 1, \dots, p - 1\}$  executes:

- 1: **for**  $j = c_-^s$  to  $c_+^s$  **do**
- 2:   get  $x_j^s$  from remote process
- 3:
- 4:   calculate  $y^s = A^{(s)}x^s$
- 5:   synchronise
- 6: **for**  $i = r_-^s$  to  $r_+^s$  **do**
- 7:   write  $y_i^s$  to remote process

Adding in locally refined SBD forms...

# 2D DISTRIBUTION

In the paper, we augment this strategy with the Separated Block Diagonal (SBD) form.

Each processor  $s \in \{0, 1, \dots, p - 1\}$  executes:

- 1: **for**  $j = c_-^s$  to  $c_+^s$  **do**
- 2:   get  $x_j^s$  from remote process
- 3:   calculate  $y^s = L^{(s)}x^s$  (Cache-oblivious SBD)
- 4:   calculate  $y^s = S^{(s)}x^s$  (Compressed BICRS)
- 5:   synchronise
- 6: **for**  $i = r_-^s$  to  $r_+^s$  **do**
- 7:   write  $y_i^s$  to remote process

# OVERVIEW

Strategy	Characteristics	Overheads		
		Data	Time	NUMA
Block CO-H+	impl. ND coarse	$\mathcal{O}(mp)$	$\mathcal{O}(m)$	$x, y$
openMP CRS	impl. 1D fine	-	low	$x$
CSB	impl. 1D fine	low	low	$x$
pOSKI	expl. 1D coarse	-	?	$x$
RD block-H	expl. 1D coarse	-	-	$x$
2D CO-SBD	expl. 2D coarse	$\mu_{\lambda-1} + qp$	$\mu_{\lambda-1} + l$	-

Data overhead: extra storage required over  $\Theta(2nz + m)$ .

Time overhead: extra real time required for 1 SpMV.

NUMA overhead: potentially unbounded data movement across the memory hierarchy (sockets).

Not-scalable overhead is printed **red**. ‘Low’ overhead is negligible in practice.  $l$  is the latency of a global synchronisation.

# EXPERIMENTS

- 1 SUMMARY
- 2 CLASSIFICATION
- 3 STRATEGIES
- 4 EXPERIMENTS**
- 5 CONCLUSIONS AND OUTLOOK

# SETUP

We use 23 matrices from four different categories:

- Structured matrices
  - small matrices (vectors fit into L3 cache)
  - large matrices
- Unstructured matrices
  - small matrices
  - large matrices

We ran experiments on three different architectures:

- DL-2000: two-socket six-core Intel Xeon X5660
- DL-580: four-socket ten-core Intel Xeon E7-4870
- DL-980: eight-socket eight-core Intel Xeon E7-2830

All machines have a bandwidth of 10.67 GB/s per socket.



# RESULTS PER CATEGORY

<i>Small structured</i>	DL-2000	DL-580	DL-980
Interleaved CSB (1D)	6.0 (12)	12.6 (40)	8.7 (48)
pOSKI (1D)	4.7 (12)	10.9 (20)	8.4 (16)
Row-distr. block CO-H (1D)	10.1 (12)	36.1 (40)	52.4 (64)
Fully distributed (2D)	3.1 (12)	2.3 (8)	2.1 (8)
Distr. CO-SBD, $qp = 64$ (2D)	3.6 (8)	3.1 (8)	2.8 (8)
<hr/>			
<i>Small unstructured</i>			
Interleaved CSB (1D)	4.8 (12)	17.9 (40)	13.8 (64)
pOSKI (1D)	6.3 (12)	20.3 (40)	15.5 (48)
Row-distr. block CO-H (1D)	8.7 (12)	32.6 (40)	49.0 (56)
Fully distributed (2D)	4.4 (12)	9.2 (40)	6.1 (16)
Distr. CO-SBD, $qp = 64$ (2D)	4.4 (8)	7.8 (16)	9.0 (16)

# RESULTS PER CATEGORY

<i>Large structured</i>	DL-2000	DL-580	DL-980
Interleaved CSB (1D)	5.4 (12)	18.0 (40)	22.0 (64)
pOSKI (1D)	3.9 (12)	12.2 (40)	11.9 (64)
Row-distr. block CO-H (1D)	3.8 (12)	10.9 (40)	16.4 (64)
Fully distributed (2D)	3.5 (12)	10.6 (30)	10.2 (40)
Distr. CO-SBD, $qp = 64$ (2D)	3.6 (8)	7.8 (32)	—
<i>Large unstructured</i>			
Interleaved CSB (1D)	7.9 (12)	24.3 (40)	26.3 (64)
pOSKI (1D)	6.6 (12)	19.4 (40)	16.2 (64)
Row-distr. block CO-H (1D)	5.4 (12)	19.2 (40)	24.6 (64)
Fully distributed (2D)	5.1 (10)	13.6 (40)	12.3 (64)
Distr. CO-SBD, $qp = 64$ (2D)	5.4 (8)	11.7 (32)	—

# 2D-SBD WITH $qp$ UP TO 512

Partitioning for  $qp = 512$  (using Mondriaan with  $\epsilon = 0.3$ ) was only successful on two structured matrices.

We test on the DL-980:

	cage15				adaptive			
$q \setminus p$	8	16	32	64	8	16	32	64
1	3.0	3.9	5.7	4.8	6.0	11.1	11.7	15.3
4	2.5	3.5	—	5.5	6.4	9.7	—	15.8
8	2.5	—	5.9	6.7	5.8	—	14.8	16.3
16	—	3.7	5.3	—	—	10.9	17.5	—
32	2.2	3.4	—	—	5.5	10.9	—	—
64	2.0	—	—	—	6.0	—	—	—
iCSB	3.2	6.0	10.6	14.6	8.4	15.9	28.5	38.1
1D-H	5.2	6.9	9.2	14.7	4.4	6.5	10.7	18.8

# CONCLUSIONS AND OUTLOOK

- 1 SUMMARY
- 2 CLASSIFICATION
- 3 STRATEGIES
- 4 EXPERIMENTS
- 5 CONCLUSIONS AND OUTLOOK**

# CONCLUSIONS

We categorised existing techniques, and used this classification to construct new SpMV multiplication methods.

Of the considered methods, the row-distributed block CO-H strategy performs best, on average.

On highly NUMA systems, explicitly distributed strategies continue to speed up, while implicit methods start to stall.

The overhead of 2D methods is costlier than the uncontrolled input vector accesses of 1D methods, but this gap decreases when more sockets are added; 2D methods do seem mandatory for future large-scale computing.

# FUTURE WORK

Incorporate low-level tuning. The row-distributed block CO-H strategy then may display a gain similar to that of parallel CRS with pOSKI presented here.

Load-balancing for 1D coarse-grained schemes must improve in order to compete with CSB.

Investigation of hybrid shared-memory schemes; e.g., the overhead of the 2D schemes increase with  $p$ , but applying 2D distributions only over sockets reduces  $p$  tremendously.

Implementing overlap of communication in the 2D scheme.

# THANK YOU

Thank you for your attention!

Pre-print and software are available at:

<http://people.cs.kuleuven.be/~albert-jan.yzelman>