Fast sparse matrix–vector multiplication by partitioning and reordering

Albert-Jan Yzelman

September, 2011

Central question:

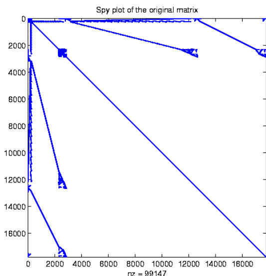
how to calculate

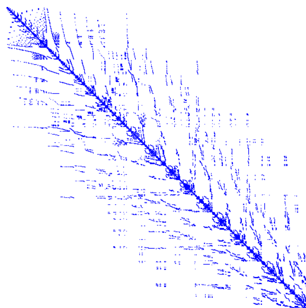
$$y = Ax$$

on a (parallel) computer, as fast as possible?

Which sparse matrices?

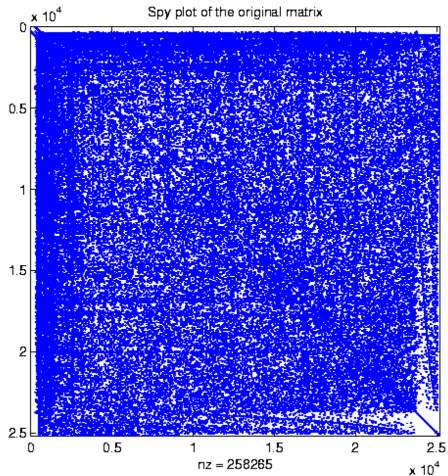
Chip industry / Markov chain modelling in chemistry





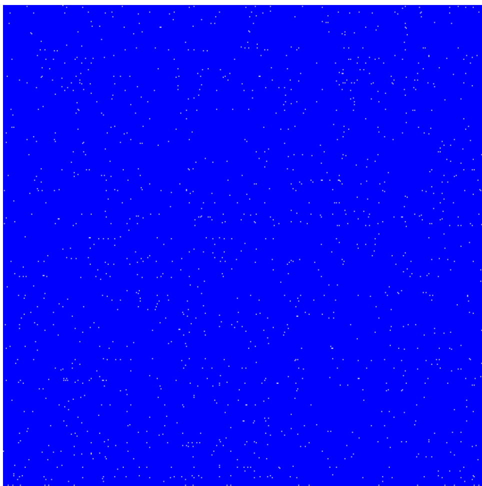
Which sparse matrices?

Chip industry

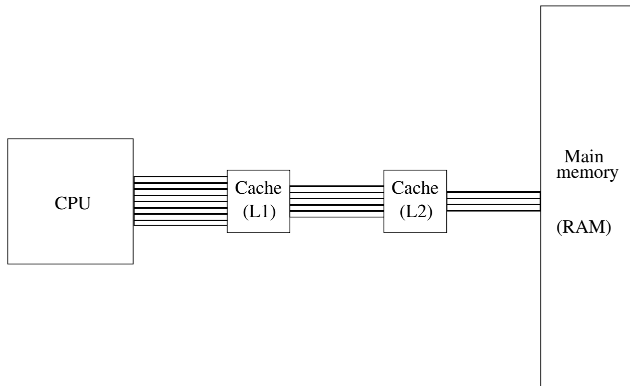


Which sparse matrices?

Link matrix



Cause of inefficiency

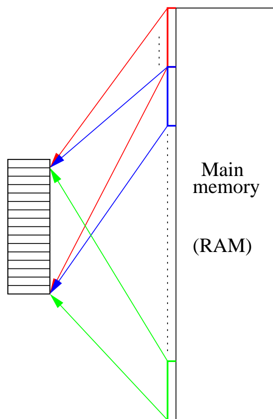


Intel Core2 (Q6600)			AMD Phenom II (945e)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$

Cause of inefficiency

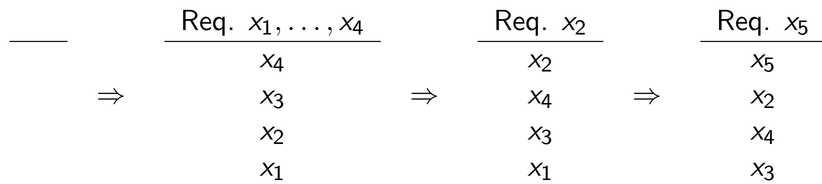
modulo mapped cache ($k = 1$)

Memory (of length L_S) from RAM with start address x is stored in cache line number $x \mod L$:



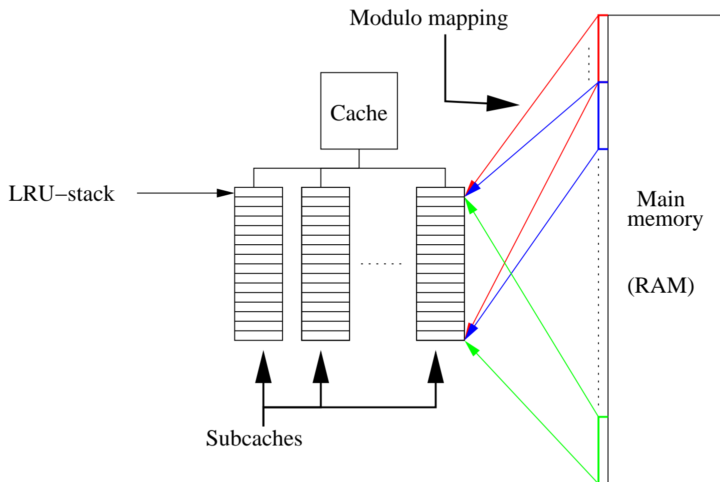
Cause of inefficiency

Instead of using a naive modulo mapping, we use a smarter policy. We use 'Least Recently Used' (LRU) policy (with $k = L$):



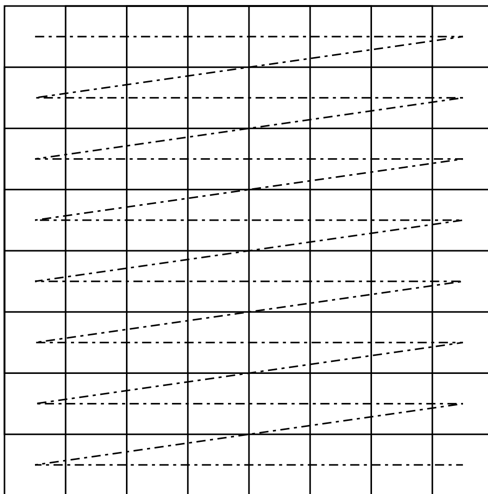
Cause of inefficiency

Realistic cache: both modulo-mapping and the LRU policy ($1 < k < L$)



Cause of inefficiency

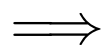
Standard datastructure: Compressed Row Storage (CRS)



Cause of inefficiency

Sparse matrix–vector multiplication (SpMV)

$x_?$



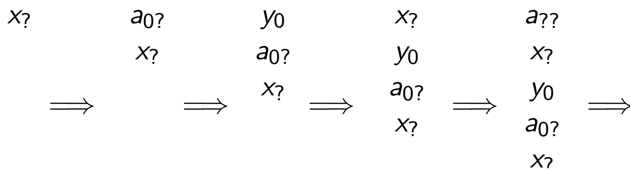
Cause of inefficiency

Sparse matrix–vector multiplication (SpMV)

$x_?$ $\Longrightarrow$ $a_{0?}$ $x_?$ $\Longrightarrow$ y_0 $a_{0?}$ $x_?$ $\Longrightarrow$

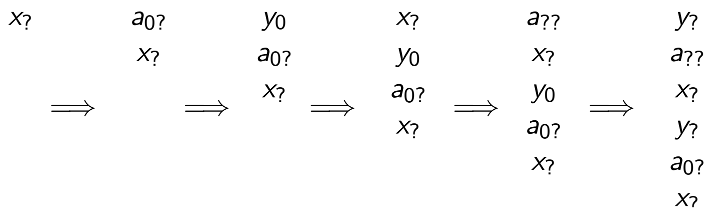
Cause of inefficiency

Sparse matrix–vector multiplication (SpMV)



Cause of inefficiency

Sparse matrix–vector multiplication (SpMV)



Memory accesses cannot be predicted!

Outline

Solutions:

1. Parallel multiplication: partitioning
2. Sequential multiplication: reordering

Bulk Synchronous Parallel

1. Bulk Synchronous Parallel
2. Partitioning
3. Sequential SpMV
4. Parallel cache-friendly SpMV
5. Experimental results
6. Outlook

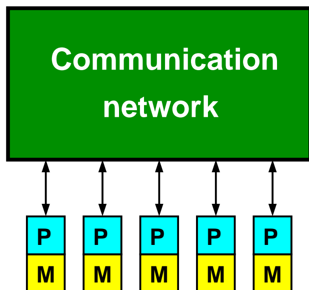
Parallel bridging models

- Message Passing Interface (MPI)
- **Bulk Synchronous Parallel** (BSP)

Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 (1990), pp. 103–111

Bulk Synchronous Parallel

A BSP-*computer*:

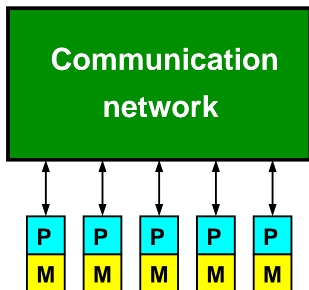


- consists of P processors, each with local memory
- does r flops each second *per processor*
- takes l time to synchronise
- has a communication speed of g

The model thus uses only four parameters (P, r, l, g).

Bulk Synchronous Parallel

A BSP-*algorithm*:



- executes a Single Program on Multiple Data (SPMD)
- performs no communication during calculation (supersteps)
- communicates only during *barrier synchronisation*

Superstep 0

Sync

Superstep 1

Bulk Synchronous Parallel

The BSP cost model:

- let $w_i^{(s)}$ be the *work* to be done by processor s in superstep i,
- let $v_i^{(s)}$ be the amount of data *received* by processor s between superstep i and $i+1$,
- let $t_i^{(s)}$ be the amount of data *transmitted* by processor s.

Define $c_i = \max\left\{\max_s v_i^{(s)}, \max_s t_i^{(s)}\right\}$ and $w_i = \max_s w_i^{(s)}$.

If T is the number of supersteps, the cost of a BSP algorithm is:

$$\sum_{i=0}^{T} \frac{1}{r} w_i + \sum_{i=0}^{T-1} (l + g \cdot c_i)$$

Bulk Synchronous Parallel

Using a BSP library, one can:

- `bsp_nprocs()`
  `bsp_pid()`
- `bsp_sync()`
- `bsp_put(source, dest, dest_pid)`
  `bsp_get(source, source_pid, dest)`
- `bsp_send(data, dest_pid)`
  `bsp_qsize()`
  `bsp_move()`

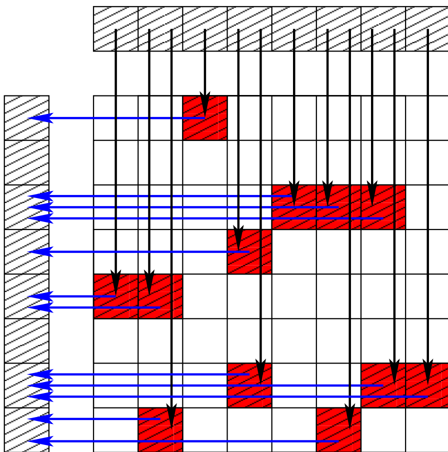
Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas, Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, Volume 24(14), pp. 1947–1980 (1998)

Example: sparse matrix, dense vector multiplication

`y=Ax`:

for each nonzero k **from** A

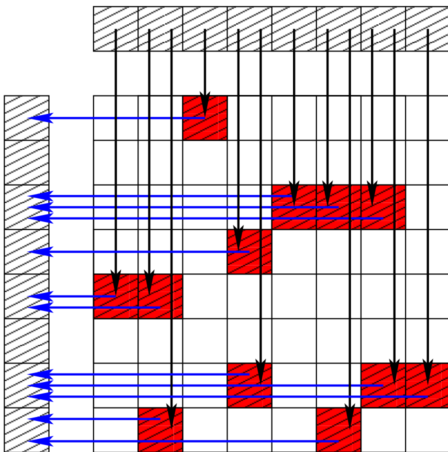
add $x[k.column] \cdot k.value$ **to** $y[k.row]$



Example: sparse matrix, dense vector multiplication

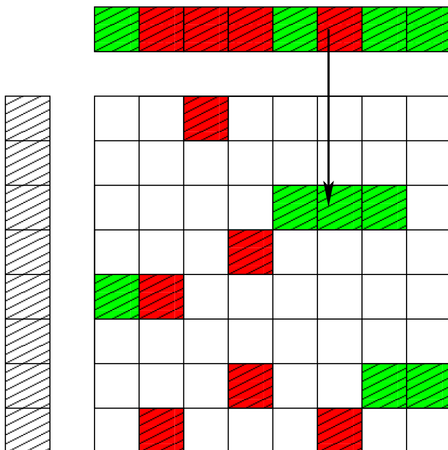
To do this in parallel:

Distribute the nonzeroes of A, but also distribute x and y;
each processor should have about $1/P$th of the total data.



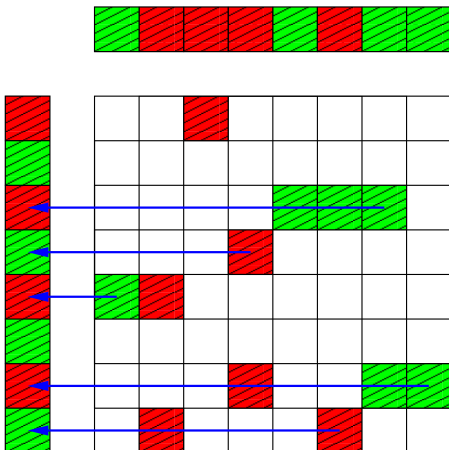
Example: sparse matrix, dense vector multiplication

Step 1 (*fan-out*): not all processors have the elements from x they need; processors need to get the missing items. Here, only one message is needed, x is distributed well.



Example: sparse matrix, dense vector multiplication

Step 2 (*mv*): use received elements from x for multiplication.
Step 3 (*fan-in*): send local results to the correct processors; here, y is distributed cyclically, obviously a bad choice.



Example: sparse matrix, dense vector multiplication

The algorithm:

1. **for all** nonzeroes k **from** A
   if column of k is not local
   request element from x from the appropriate processor
   synchronise
2. **for all** nonzeroes k **from** A
   do the SpMV for k
   send all non-local row sums to the appropriate processor
   synchronise
3. **add** all incoming row sums to the corresponding $y[i]$

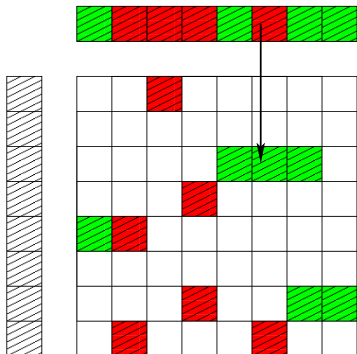
Partitioning

1 Bulk Synchronous Parallel
2 Partitioning
3 Sequential SpMV
4 Parallel cache-friendly SpMV
5 Experimental results
6 Outlook

Automatic nonzero partitioning

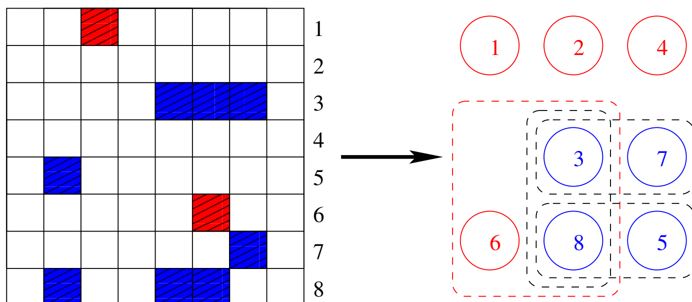
What causes the communication?

- nonzeroes on the same column distributed to different processors: *fan-out* communication



Automatic nonzero partitioning

"Shared" columns: communication during *fan-out*

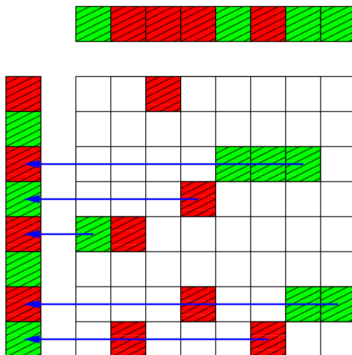


Column-net model; a cut net means a shared column

Automatic nonzero partitioning

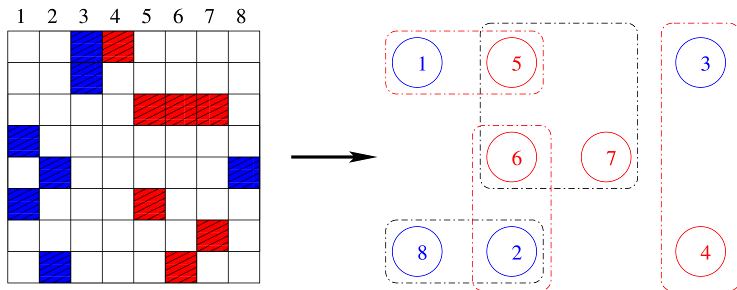
What causes the communication?

- nonzeroes on the same row distributed to different processors: fan-in communication



Automatic nonzero partitioning

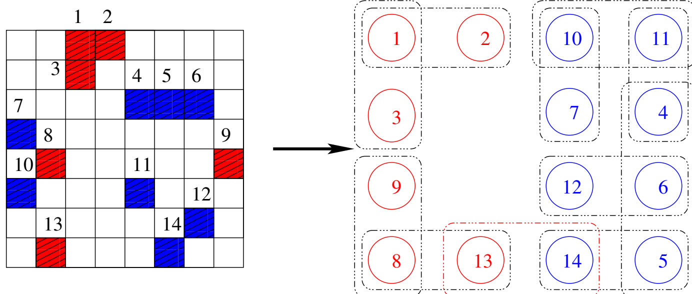
"Shared" rows: communication during fan-in



Row-net model; a cut net means a shared row

Automatic nonzero partitioning

Catch both types of communication:



Fine-grain model; a cut net means either a shared row or column

Automatic nonzero partitioning

A cut net n_i means communication. The number of processors involved in processing the net is:

$$\lambda_i = \#\{\mathcal{V}_i \cap n_i \neq \emptyset\}.$$

So the quantity to minimise is:

$$C = \sum_i (\lambda_i - 1).$$

Partitioning strategy:

- Model the sparse matrix using a hypergraph
- Partition the *vertices* of that hypergraph in two so that C is minimised under the additional constraint of load balance.

Automatic nonzero partitioning

Partitioning strategy:

- **Model** the sparse matrix using a hypergraph
- Partition the *vertices* of that hypergraph in two.

Catalyürek & Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel Distributed Systems 10 (1999).

Catalyürek & Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, Proc. IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (2001).

Bisseling & Vastenhouw, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review Vol. 47(1), 2005.

Automatic nonzero partitioning

Partitioning strategy:

- Model the sparse matrix using a hypergraph
- **Partition** the *vertices* of that hypergraph in two.

Kernighan & Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical Journal 49 (1970).

Fiduccia & Mattheyses, *A linear-time heuristic for improving network partitions*, Proceedings of the 19th IEEE Design Automation Conference (1982).

Catalyürek & Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool*, Bilkent University, Ankara (1999–now)

Bisseling, Fagginger Auer, van Leeuwen, Meesen, Vastenhouw, Yzelman, *Mondriaan for sparse matrix partitioning*, Utrecht University (2002–now).

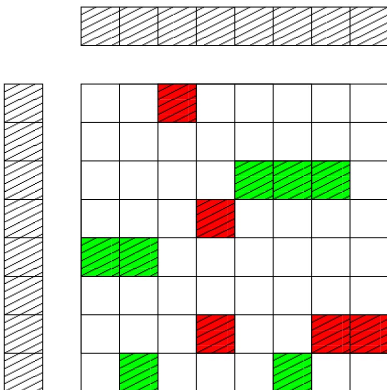
Mondriaan partitioning strategy

- **Model the sparse matrix using a hypergraph**
- Partition the vertices of the hypergraph (in two)

Try both row- and column-net, and choose best

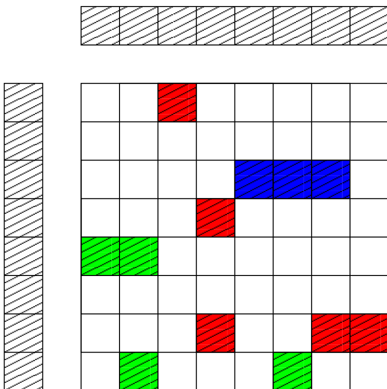
Mondriaan partitioning strategy

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



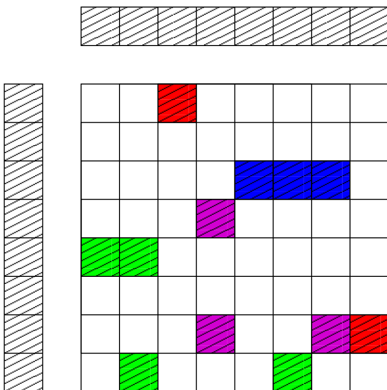
Mondriaan partitioning strategy

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



Mondriaan partitioning strategy

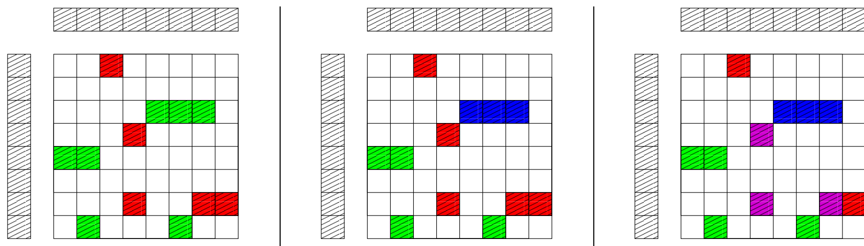
- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



Mondriaan partitioning strategy

Mondriaan:

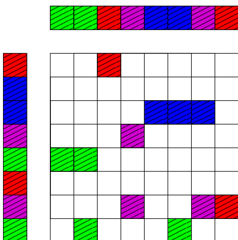
- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts



Mondriaan partitioning strategy

Mondriaan:

- Model the sparse matrix using a hypergraph
- Partition the vertices of the hypergraph (in two)
- Recursively keep partitioning the vertex parts
- **Partition the vector elements**



Bisseling and Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, Vol. 21 (2005) pp. 47-65

Sequential SpMV

1. Bulk Synchronous Parallel
2. Partitioning
3. Sequential SpMV
4. Parallel cache-friendly SpMV
5. Experimental results
6. Outlook

Realistic cache

$1 < k < L$, combining modulo-mapping and the LRU policy

Compressed Row Storage (CRS)

CRS

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

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]
col: [0 1 2 2 3 0 3 0 2 3] , $2nnz + (m+1)$ accesses
row: [0 3 5 7 10]

Incremental CRS

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

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]
col_increment: [0 1 1 4 1 1 3 1 2 1] , $2nnz + m$ accesses
row_increment: [0 1 1 1]

Note: accesses like plain CRS, but requires less instructions for SpMV

Joris Koster, *Parallel templates for numerical linear algebra, a high-performance computation library*, Masters Thesis, Utrecht University, 2002

Blocked CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$, dense blocks: $4, 1, 3$ / $2, 3$ / 1 / 2 / $7, 0, 1, 1$

Stored as:

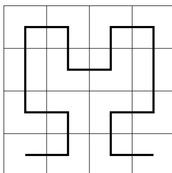
nzs: [4 1 3 2 3 1 2 7 0 1 1]
blk: [0 3 5 6 7 11]
col: [0 2 0 3 0]
row: [0 1 2 4 5]

, $nnz + (2nblk + 1) + (m + 1)$ accesses

Pinar and Heath, *Improving Performance of Sparse Matrix-Vector Multiplication*, 1999

Fractal datastructures (triplets)

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



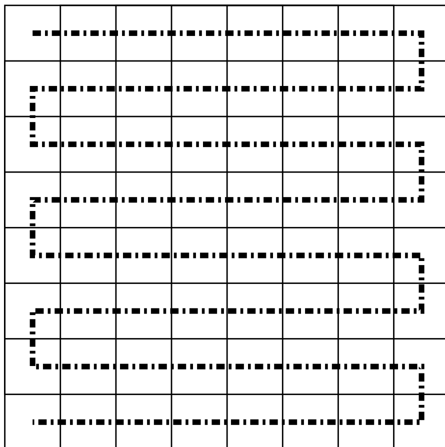
Stored as:

nzs: [7 1 4 1 2 2 3 2 1]
i : [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero
j : [0 0 0 1 1 3 3 3 2]

Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005

Zig-zag CRS

Change the order of CRS:



Zig-zag CRS

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

Stored as:

nzs: [4 1 3 3 2 1 2 1 1 7]
col: [0 1 2 3 2 0 3 3 2 0] , $2nnz + (m+1)$ accesses
row: [0 3 5 7 10]

Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SIAM Journal on Scientific Computing (2009)

Why not also change the input matrix structure?

- Assume zig-zag CRS ordering (theoretically)
- Allow only row and column permutations

Separated Block Diagonal form

Separated Block Diagonal form

No cache misses

1 cache miss per row

3 cache misses per row

1 cache miss per row

Separated Block Diagonal form

No cache misses

1 cache miss per row

3 cache misses

1 cache miss per row

7 cache misses per row

1 cache miss per row

3 cache misses per row

1 cache miss per row

Separated Block Diagonal form

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

Separated Block Diagonal form

In 1D, row and column permutations bring the original matrix A in *Separated Block Diagonal* (SBD) form as follows.

A is modelled as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, with

- $\mathcal{V}$ the set of columns of A,
- $\mathcal{N}$ the set of *hyperedges*, each element is a subset of $\mathcal{V}$ and corresponds to a row of A.

A partitioning $\mathcal{V}_1, \mathcal{V}_2$ of $\mathcal{V}$ can be constructed; and from these, three hyperedge categories can be constructed:

- $\mathcal{N}_{-}^{\text{row}}$ as the set of hyperedges with vertices only in $\mathcal{V}_1$,
- $\mathcal{N}_{c}^{\text{row}}$ as the set of hyperedges with vertices both in $\mathcal{V}_1$ and $\mathcal{V}_2$,
- $\mathcal{N}_{+}^{\text{row}}$ the set of remaining hyperedges.

Separated Block Diagonal form

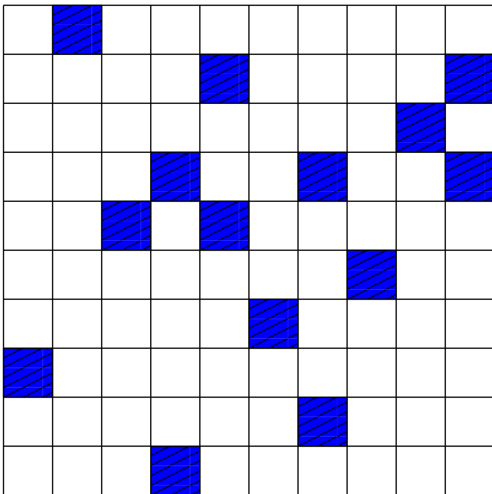
$\mathcal{N}_{-}^{\text{row}}$

$\mathcal{N}_{c}^{\text{row}}$

$\mathcal{N}_{+}^{\text{row}}$

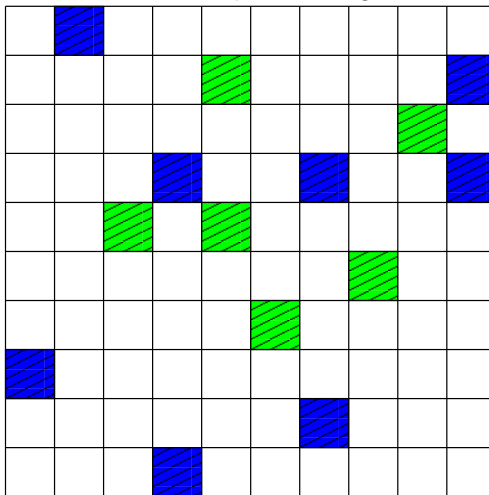
$\mathcal{V}_1$ $\mathcal{V}_2$

Permuting to SBD form

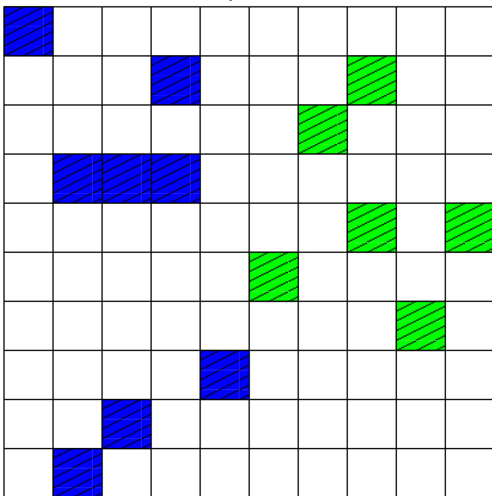


Permuting to SBD form

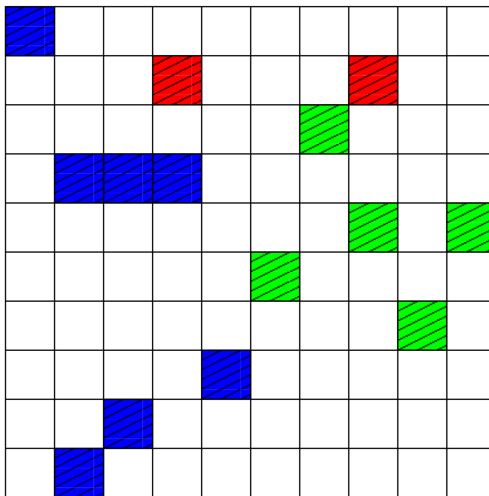
Column partitioning



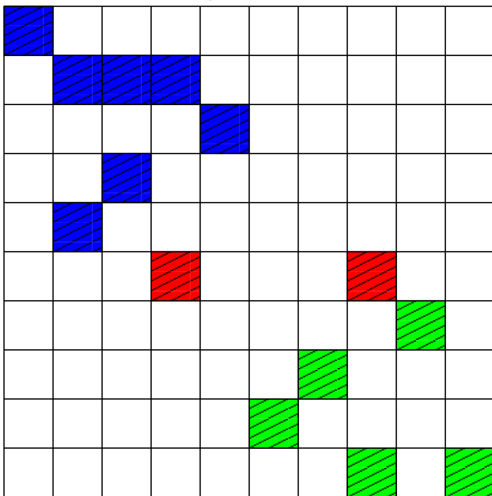
Permuting to SBD form



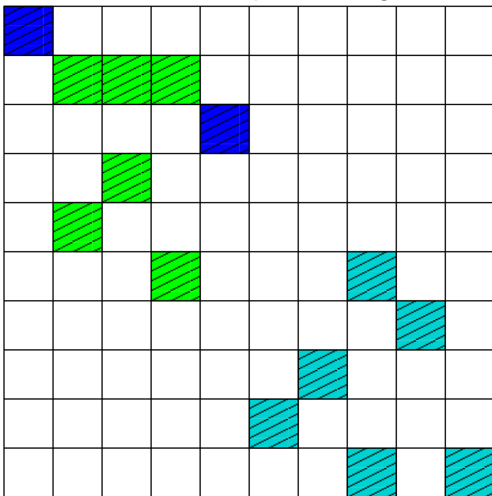
Permuting to SBD form



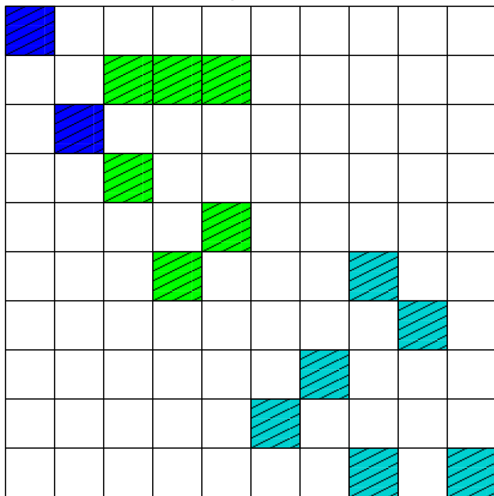
Permuting to SBD form



Permuting to SBD form

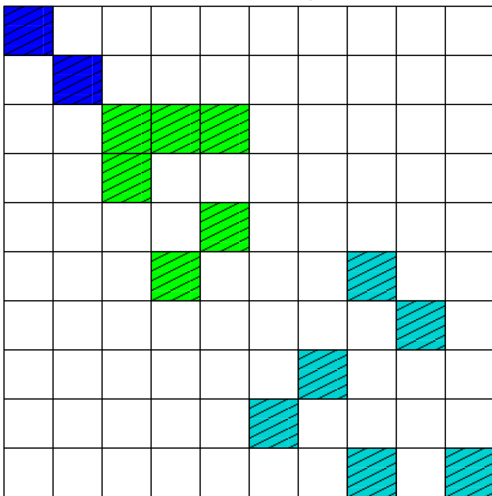


Permuting to SBD form

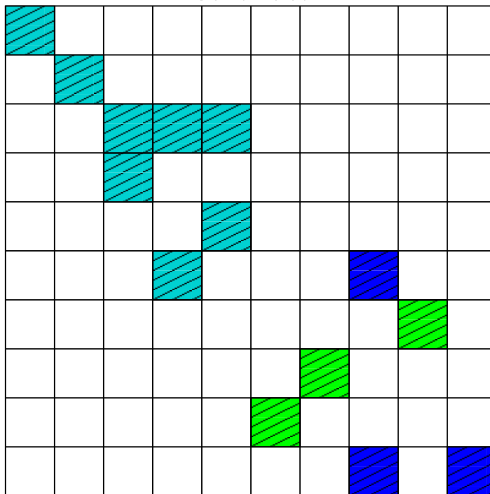


Permuting to SBD form

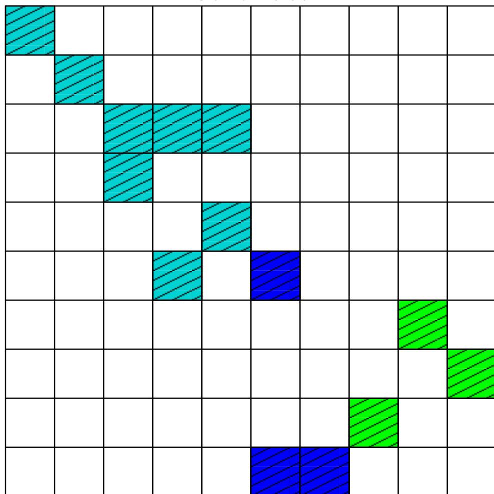
No mixed rows - row permutation



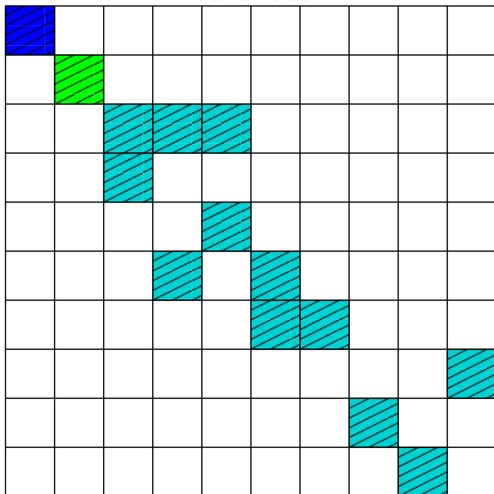
Permuting to SBD form



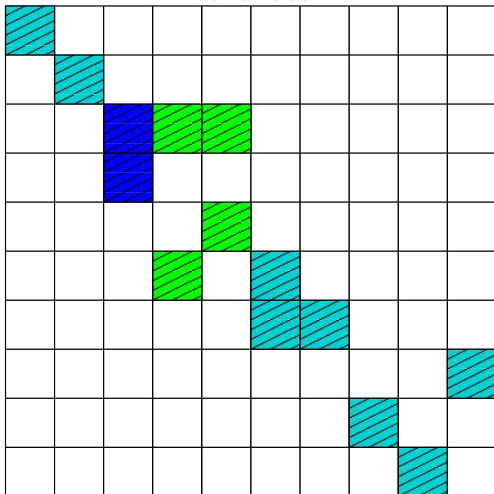
Permuting to SBD form



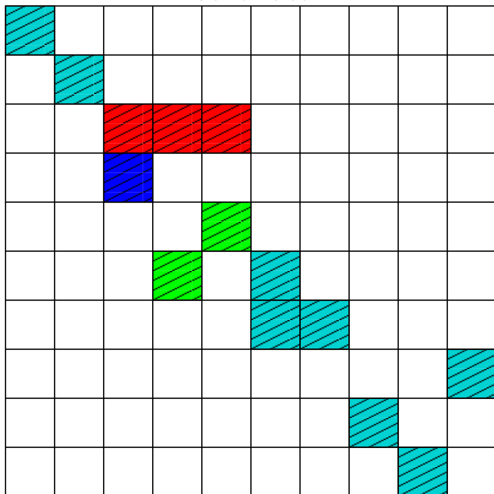
Permuting to SBD form



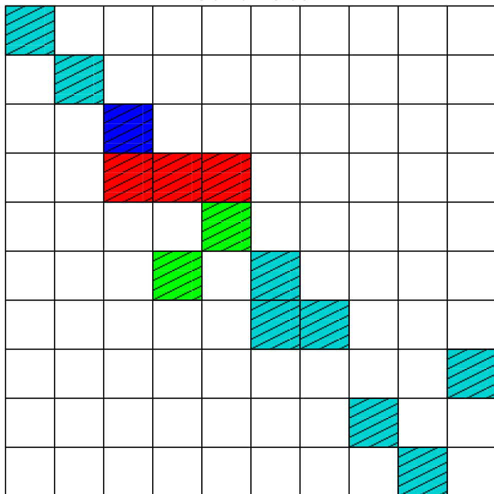
Permuting to SBD form



Permuting to SBD form

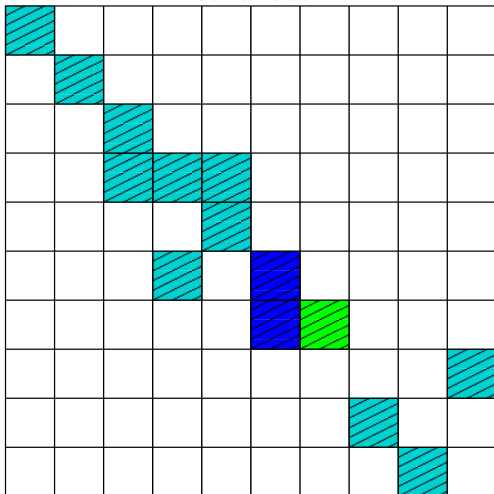


Permuting to SBD form

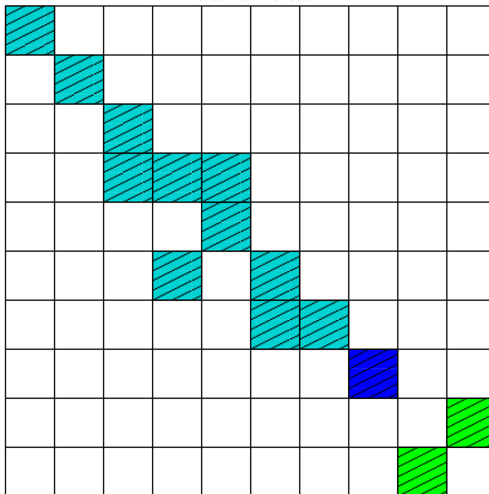


Permuting to SBD form

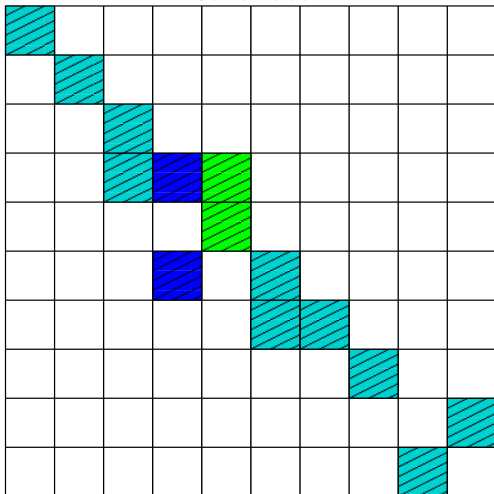
Continued



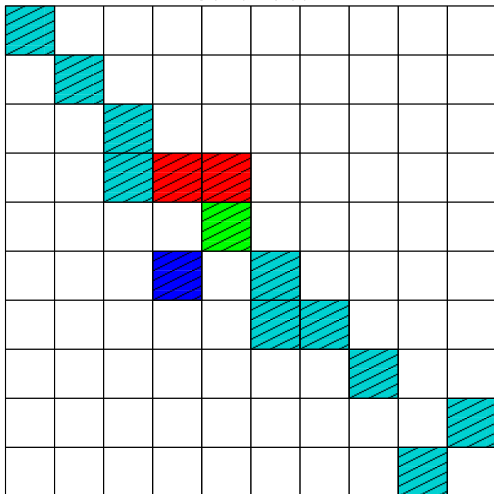
Permuting to SBD form



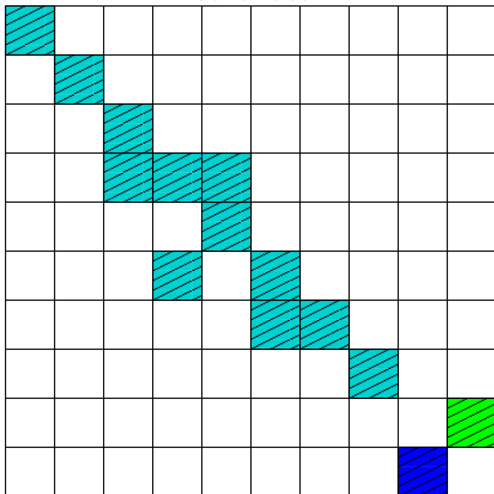
Permuting to SBD form



Permuting to SBD form

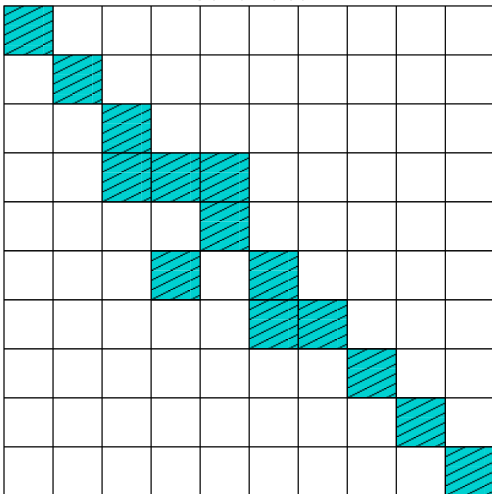


Permuting to SBD form



Permuting to SBD form

Continued



Reordering parameters

$$\text{Taking } p = \frac{n}{S},$$

the number of cache misses is strictly bounded by

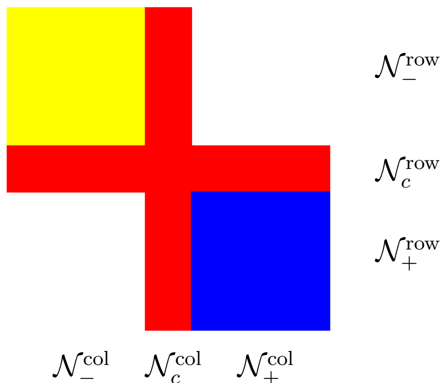
$$\sum_{i:\ n_i \in \mathcal{N}} (\lambda_i - 1);$$

taking $p \to \infty$ yields a *cache-oblivious* method with the same bound.

- Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SIAM Journal on Scientific Computing, 2009
  (Chapter 1 of the thesis)

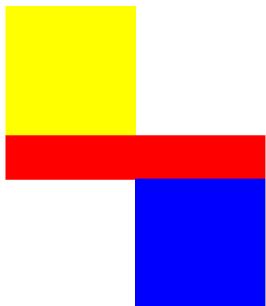
Two-dimensional SBD (doubly separated block diagonal)

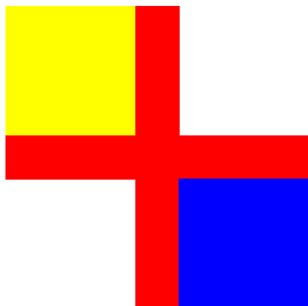
Using a fine-grain model of the input sparse matrix, individual nonzeros each correspond to a vertex;
each row and column has a corresponding net.



The quantity minimised remains $\sum_i (\lambda_i - 1)$.

Two-dimensional SBD (doubly separated block diagonal)





1D **2D**

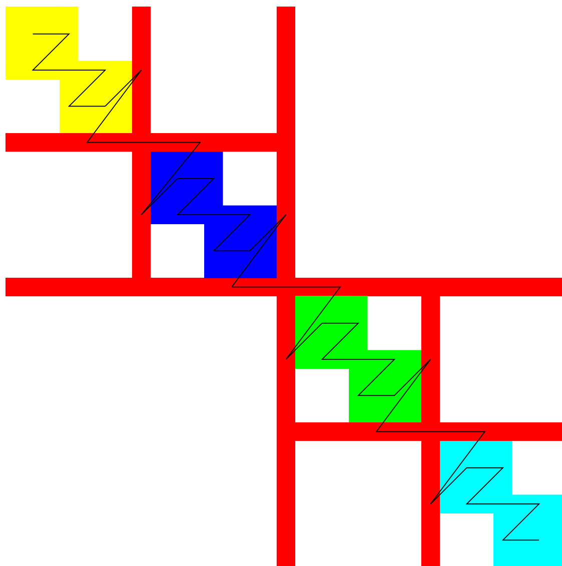
Yzelman and Bisseling, *Two-dimensional cache-oblivious sparse matrix–vector multiplication*, Parallel Computing, 2011; in press (Chapter 2 of the thesis)

Two-dimensional SBD (doubly separated block diagonal)

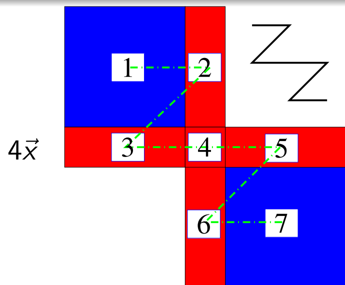
Zig-zag CRS is not suitable for handling 2D SBD!

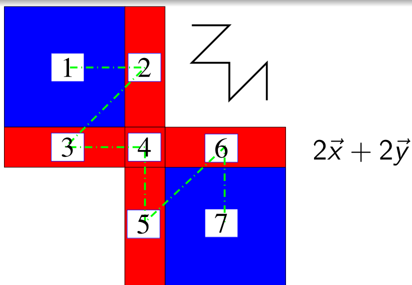
Two-dimensional SBD

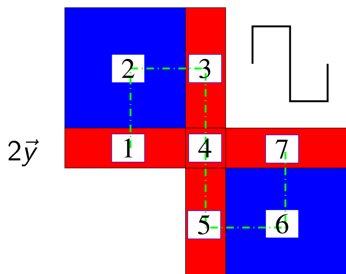
Two-dimensional SBD

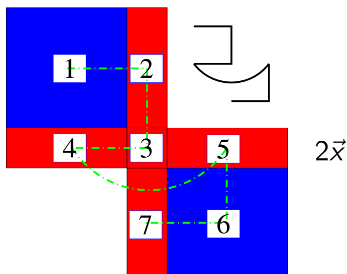


Two-dimensional SBD; block ordering



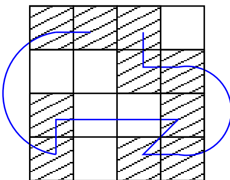






Bi-directional Incremental CRS (BICRS)

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



Stored as:

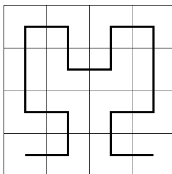
nzs:	[3 2 3 1 1 2 1 7 4 1]
col_increment:	[2 4 1 4 -1 5 -3 4 4 1] ,
row_increment:	[0 1 2 -1 1 -3]

$2nnz + (row_jumps + 1)$ accesses

BICRS and fractal storage

Uncompressed (triplets):

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



Stored as:

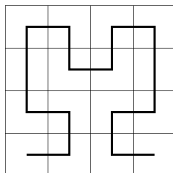
nzs: [7 1 4 1 2 2 3 2 1]
i : [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero
j : [0 0 0 1 1 3 3 3 2]

Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005

BICRS and fractal storage

Compressed (BICRS):

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



Stored as:

nzs: [7 1 4 1 2 2 3 2 1]
i : [3 -1 -2 1 -1 1 1 1] , $\mathbf{2nnz} + (\mathbf{row_jumps} + \mathbf{1})$ accesses
j : [0 4 4 1 4 2 4 4 3]

Yzelman and Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, Proceedings of the ECMI 2011; in press (Chapter 3 of the thesis)

Parallel cache-friendly SpMV

1. Bulk Synchronous Parallel
2. Partitioning
3. Sequential SpMV
4. Parallel cache-friendly SpMV
5. Experimental results
6. Outlook

What kind of parallel machines?

Different kinds of parallelism:

1. distributed-memory ('traditional' supercomputer)
2. shared-memory (multicore PC)
3. stream processing (GPU)

What kind of parallel machines?

Different kinds of parallelism:

1. distributed-memory (‘traditional’ supercomputer)
2. shared-memory (multicore PC)
3. stream processing (GPU)

Yzelman and Bisseling, *An Object-Oriented BSP Library for Multicore Programming*, Concurrency and Computation: Practice and Experience, 2011; in press. (Chapter 4 of the thesis.)

MulticoreBSP

BSP programming explicitly for shared-memory architectures:

`http://www.multicorebsp.com`

Programmed in standard Java, this is a fully object-oriented library which contains only 12 functions and 2 interfaces. One function is new:

- *bsp_nprocs()*
  bsp_pid()
- *bsp_sync()*
- *bsp_put(source, dest, dest_pid)*
  bsp_get(source, source_pid, dest)
  bsp_direct_get*(source, source_pid, dest)*
- *bsp_send(data, dest_pid)*
  bsp_qsize()
  bsp_move()

MulticoreBSP

The efficiency of MulticoreBSP has been tested by implementing examples for the following scientific computing operations:

1. dense vector inner-product calculation,
2. dense LU decomposition,
3. the fast Fourier transformation,
4. sparse matrix–vector multiplication

(examples are adapted from: Bisseling, *Parallel Scientific Computation: A structured approach using BSP and MPI*, Oxford University Press, 2004)

On shared-memory architectures

The original (3-step) BSP algorithm (also for distributed-memory):

1. **for all** nonzeroes k **from** A
   if column of k is not local
   request element from x from the appropriate processor
   synchronise
2. **for all** nonzeroes k **from** A
   do the SpMV for k
   send all non-local row sums to the appropriate processor
   synchronise
3. **add** all incoming row sums to the corresponding $y[i]$

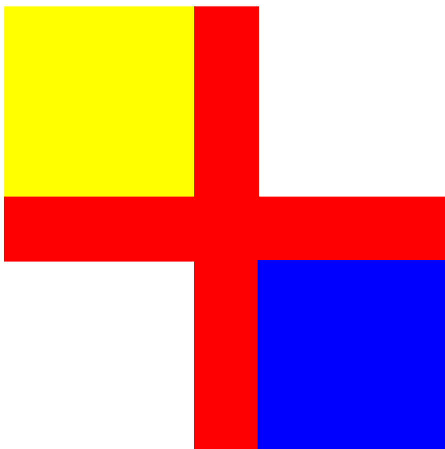
On shared-memory architectures

Alternative (2-step) SpMV algorithm in MulticoreBSP:

1. **for all** nonzeroes k **from** A
   if both row and column of k are local
   add do the SpMV for k
   if column of k is not local
   direct get element from x, and do SpMV for k
   send all non-local row sums to the correct processor
   synchronise
2. **add** all incoming row sums to the corresponding $y[i]$

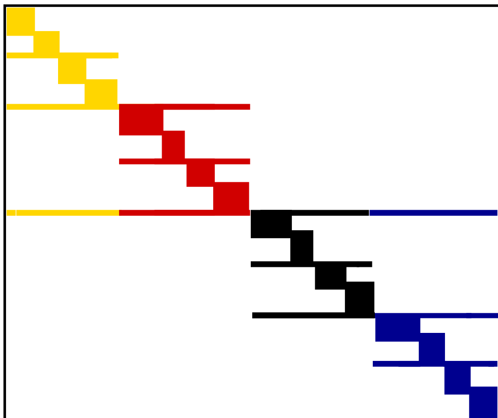
On shared-memory architectures

Both these algorithms directly use the partitioner output:



On shared-memory architectures

Alternatively: use both partitioner and reordering output, i.e., partition for $p \to \infty$ but distribute only over the actual number of processors:



(This is Chapter 5 of the thesis)

On shared-memory architectures

Alternatively:

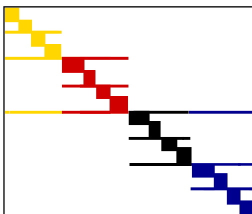
- global version of the matrix A, stored in BICRS,
- global input vector x,
- global output vector y.

On shared-memory architectures

Alternatively:

- global version of the matrix A, stored in BICRS,
- global input vector x,
- global output vector y.

- Multiple threads work simultaneously on contiguous blocks in the BICRS data structure;
- conflicts only arise on the row-wise separator areas.

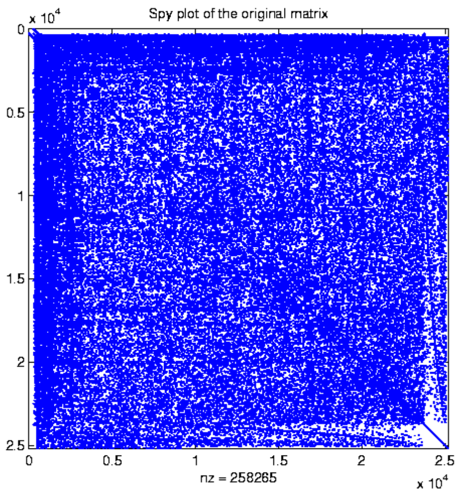


Use $t-1$ synchronisation steps to prevent concurrent writes.

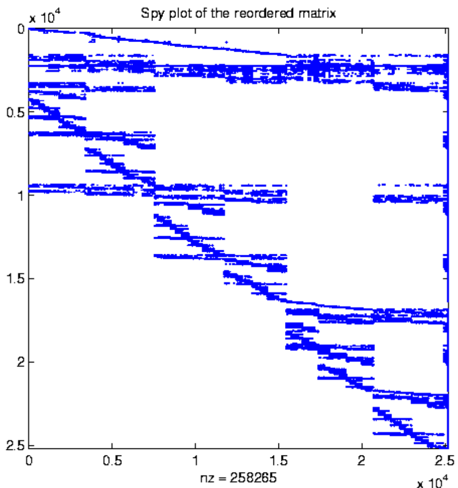
Experimental results

1. Bulk Synchronous Parallel
2. Partitioning
3. Sequential SpMV
4. Parallel cache-friendly SpMV
5. Experimental results
6. Outlook

Chip industry



Chip industry – 1D reordering



$p = 100, \quad \epsilon = 0.1$

Chip industry – 2D reordering

$p = 100, \quad \epsilon = 0.1$

Link matrix

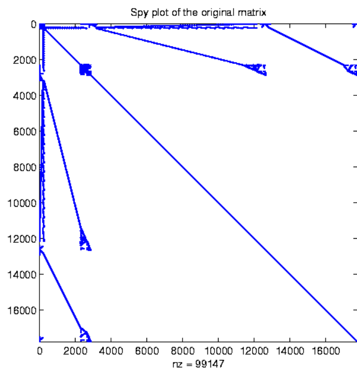
Link matrix – 1D reordering

$p = 20, \quad \epsilon = 0.1$

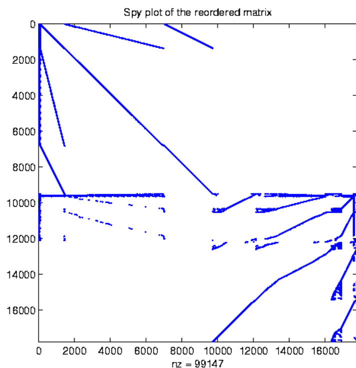
Link matrix – 2D reordering

$p = 20, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering

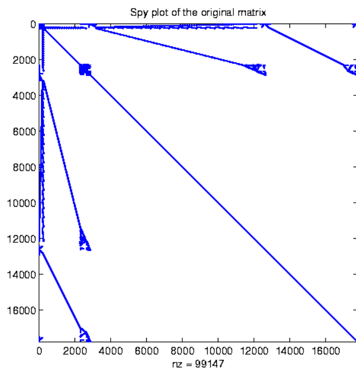


$p = 1 \quad$ (original)

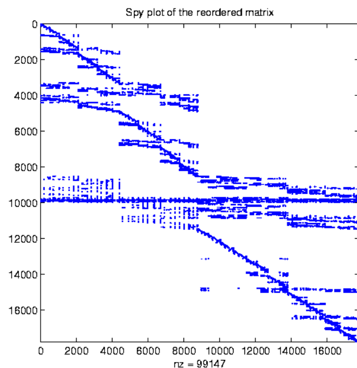


$p = 2, \quad \epsilon = 0.1$

The memplus matrix – 1D reordering

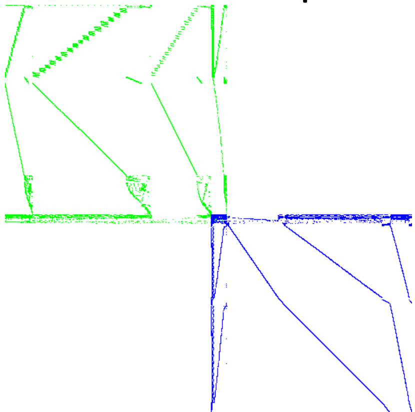


$p = 1$ (original)

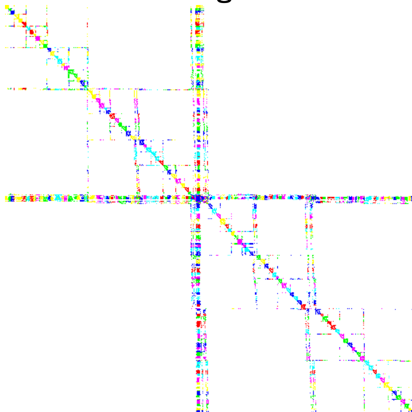


$p = 100, \quad \epsilon = 0.1$

The memplus matrix – 2D reordering

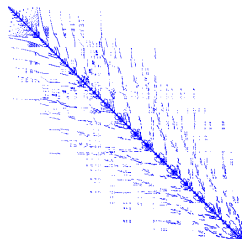


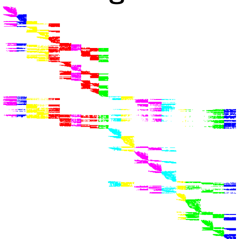
$p = 2$

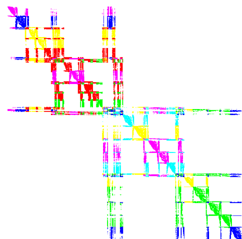


$p = 100$

The cage14 matrix


Original


1D ($p = 20$, $\epsilon = 0.1$)


Finegrain ($p = 20$, $\epsilon = 0.1$)

Sequential SpMV times without reordering

	Intel Q6600		AMD 945e	
	s3dkt3m2	GL7d18	s3dkt3m2	GL7d18
Triplet	18	1323	12	466
CRS	14	780	12	437
ICRS	13	856	9	610

All timings are in milliseconds.
s3dkt3m2 is a 90449×90449 structured sparse matrix with about 1.9 million nonzeroes.
GL7d18 is a 1955309×1548650 unstructured sparse matrix with about 35.6 million nonzeroes.

Sequential SpMV times with reordering

Intel Q6600

	s3dkt3m2		GL7d18	
Original	10	(OSKI)	780	(CRS)
Hilbert	28	(BICRS)	372	(BICRS)
1D reordering	10	(OSKI)	553	(OSKI)
1D blocking	13	(Block)	613	(BICRS)
2D Mondriaan	11	(Block)	550	(CRS)
2D finegrain	13	(ICRS)	568	(BICRS)

All timings are in milliseconds.

Sequential SpMV times with reordering

AMD 945e

	s3dkt3m2		GL7d18	
Original	9	(ICRS)	730	(CRS)
Hilbert	16	(BICRS)	453	(BICRS)
1D reordering	9	(ICRS)	452	(ICRS)
1D blocking	9	(Block)	412	(BICRS)
2D Mondriaan	8	(BICRS)	425	(BICRS)
2D finegrain	9	(Block)	423	(BICRS)

All timings are in milliseconds.

Pre-processing and SpMV times

Matrix	Reordering time	SpMV time (old/1D/2D)
memplus, $p = 50$:	4 seconds	(0.4 / 0.3 / 0.3 ms.)
rhpentium, $p = 50$:	1 minute	(0.9 / 0.7 / 0.9 ms.)
cage14, $p = 10$:	30 minutes	(111.6 / 130.4 / 130.4 ms.)
wiki2005, $p = 10$:	2 hours	(347.4 / 212.5 / 136.7 ms.)
GL7d18, $p = 10$:	2 hours	(780.3 / 552.5 / 549.5 ms.)
wiki2006, $p = 9$:	21 hours	(745.0 / 495.0 / 311.8 ms.)

Black indicates use of a regular data structure, green the use of block ordering, blue the use of the OSKI auto-tuning library.

(reordering on an AMD Opteron 2378, SpMV on an Intel Q6600)

Parallel: distributed-memory architectures

Directly use partitioner output:

Matrix	$p = 1$	$p = 4$	$p = 16$	$p = 64$
cage13	372.2	120.7 (3.0x)	37.1 (10x)	*16.1* (23.1x)
stanford_berkeley	552.6	169.3 (3.2x)	71.2 (7.7x)	*21.4* (25.8x)

Using the BSPonMPI library with the 3-step BSP SpMV multiplication code, on two nodes of 16 IBM Power6+ processors each.

Bisseling, van Leeuwen, Çatalyürek, Fagginger Auer, Yzelman, *Two-dimensional approach to sparse matrix partitioning* in *Combinatorial Scientific Computing* by Schenk and Naumann (eds.), 2011; in press.

Parallel: shared-memory architectures

Directly use partitioner output:

Matrix	$p = 1$	$p = 2$	$p = 3$	$p = 4$
cage14	232.8	272.5 (0.8x)	*249.7* (0.9x)	297.1 (0.7x)
wiki2005	564.2	285.3 (1.9x)	*244.5* (2.3x)	255.0 (2.2x)

Using the Java MulticoreBSP library on an Intel Q6600; two superstep algorithm with full synchronisation.

Combined parallel with reordering

Intel Core 2 Q6600:

s3dkt3m2 $_t\backslash^p$	4	16	32	64
1	17	*16*	18	17
2	17	*16*	18	17
4	20	*18*	22	21

GL7d18 $_t\backslash^p$	4	16	32	64
1	906	633	492	*486*
2	718	347	345	*285*
4	583	491	398	*385*

Maximum speedup: 3.1x using 2 cores

Combined parallel with reordering

AMD Phenom II 945e:

s3dkt3m2 $_t\backslash^p$	4	16	32	64
1	*11*	*11*	14	*11*
2	8	*7*	9	8
4	*6*	7	*6*	*6*

GL7d18 $_t\backslash^p$	4	16	32	64
1	482	373	*352*	372
2	333	376	*236*	357
4	250	200	*199*	237

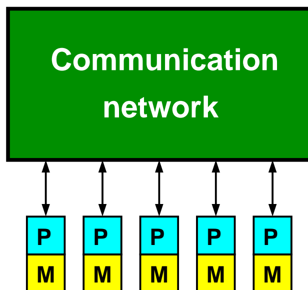
Maximum speedup for s3dkt3m2: 1.8x using 2 cores
Maximum speedup for GL7d18: 2.4x using 4 cores

Outlook

1. Bulk Synchronous Parallel
2. Partitioning
3. Sequential SpMV
4. Parallel cache-friendly SpMV
5. Experimental results
6. Outlook

MulticoreBSP

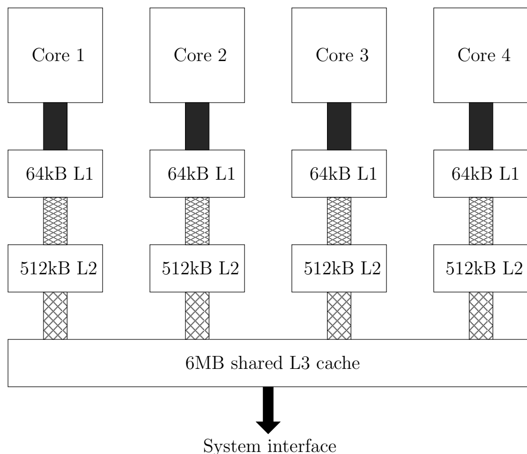
For shared-memory (multicore) architectures, the original BSP model



may no longer apply.

MulticoreBSP

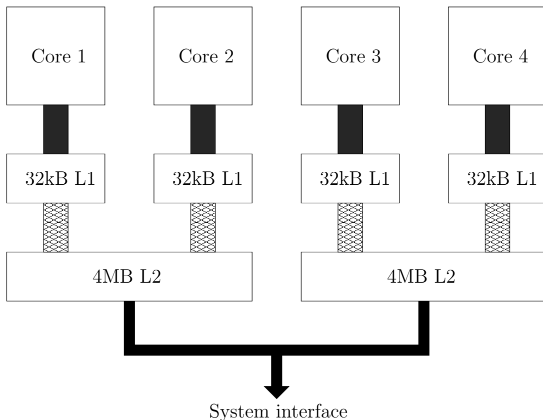
The AMD Phenom II 945e processor has uniform memory access:



This is modelled well by BSP.

MulticoreBSP

However, the Intel Core 2 Q6600 processor has non-uniform memory access (NUMA):



This is *not* modelled well by BSP.

ManycoreBSP?

Hierarchical BSP model (Intel Q6600):

$$p = 2,\ r = 3\text{GHz},\ l = l_1,\ g = g_1,\ M = 8\text{GB}$$

$$p = 2,\ r = 3\text{GHz},\ l = l_2,\ g = g_2,\ M = 4\text{MB}$$

$$p = 1,\ r = 3\text{GHz},\ l = l_3,\ g = g_3,\ M = 32\text{kB}$$

Leslie G. Valiant, *A bridging model for multi-core computing*, *Lecture Notes in Computer Science*, vol. 5193, Springer (2008); pp 13–28.

ManycoreBSP?

Hierarchical BSP model (AMD 945e):

- $p = 1,\ r = 3\text{GHz},\ l = l_1,\ g = g_1,\ M = 4\text{GB}$
  - $p = 4,\ r = 3\text{GHz},\ l = l_2,\ g = g_2,\ M = 6\text{MB}$
    - $p = 1,\ r = 3\text{GHz},\ l = l_3,\ g = g_3,\ M = 512\text{kB}$
      - $p = 1,\ r = 3\text{GHz},\ l = l_4,\ g = g_4,\ M = 64\text{kB}$

Conclusion

Thank you for your attention!

My current location:

K.U.Leuven, Dept. of Computing Sciences
Intel ExaScience Laboratory at IMEC

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

Software locations:

- http://www.math.uu.nl/people/bisseling/Mondriaan
- http://albert-jan.yzelman.net/software
- http://www.multicorebsp.com