Shared-memory parallel computing

Albert-Jan Yzelman

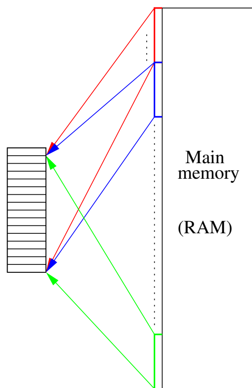
22nd of November, 2013

Shared-memory architectures and paradigms

1. Shared-memory architectures and paradigms
2. Applications
3. Case study: Intel Xeon Phi

Caches

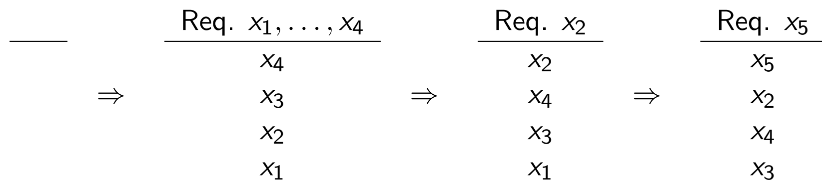
Divide the main memory (RAM) in stripes of size L_S.



The ith line in RAM is mapped to the cache line $i \bmod L$, where L is the number of available cache lines.

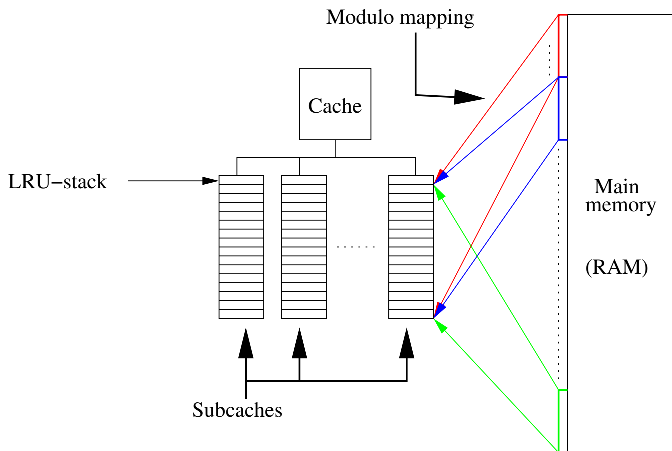
Caches

A smarter cache follows a pre-defined policy instead; for instance, the 'Least Recently Used (LRU)' policy:



Caches

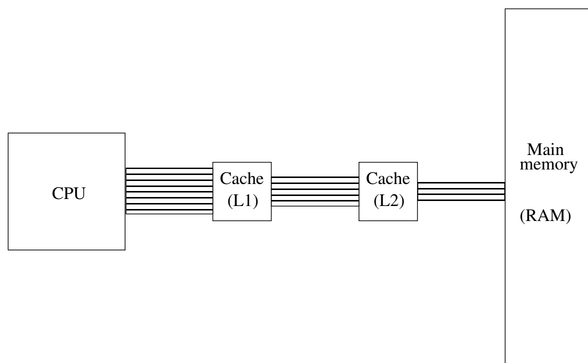
Realistic caches combine modulo-mapping and the LRU policy:



k is the number of subcaches; there are L/k LRU stacks.

Caches

Realistic caches are used within multi-level memory hierarchies:



	Intel Core2 (Q6600)		AMD Phenom II (945e)		Intel Westmere (E7-2830)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$	$S = 256$kB	$k = 8$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$	$S = 2$MB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$	$S = 24$MB	$k = 24$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:

x_0

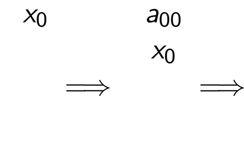
$\Longrightarrow$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:

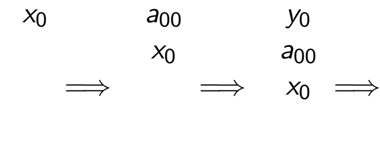


Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:

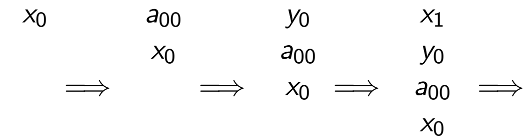


Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:



Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:

$$\begin{array}{c} x_0 \\ \\ \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ x_0 \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{01} \\ x_1 \\ y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow$$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching:

$$\begin{array}{c} x_0 \\ \\ \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{00} \\ x_0 \\ \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{00} \\ x_0 \\ \\ \hline \end{array} \Longrightarrow \begin{array}{c} x_1 \\ y_0 \\ a_{00} \\ x_0 \\ \hline \end{array} \Longrightarrow \begin{array}{c} a_{01} \\ x_1 \\ y_0 \\ a_{00} \\ \hline x_0 \end{array} \Longrightarrow \begin{array}{c} y_0 \\ a_{01} \\ x_1 \\ a_{00} \\ \hline x_0 \end{array}$$

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m-1$ rows.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m-1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A. But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each block of columns.

Caches and multiplication

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row; this causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A. But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each block of columns.

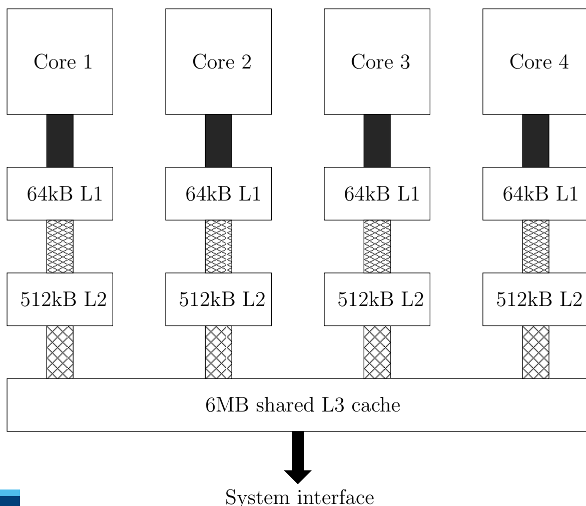
Fix:

- stop processing before an element from y is evicted; first do the remaining column blocks.

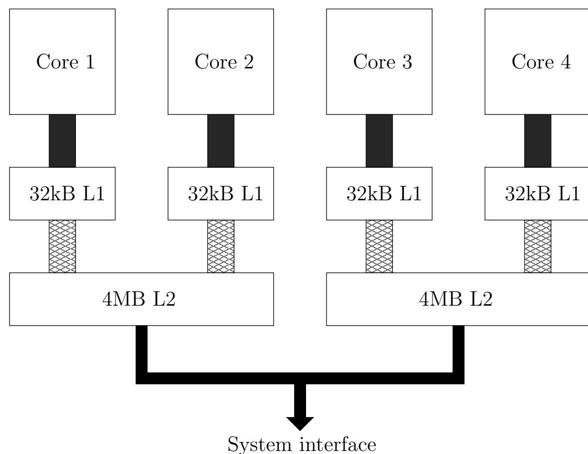
Consecutive processing of $p \times q$ submatrices (cache-aware blocking).

Caches and multicore

Most architectures employ shared caches; $(p, r, l, g) = (4, 3\text{GHz}, l, g)$:



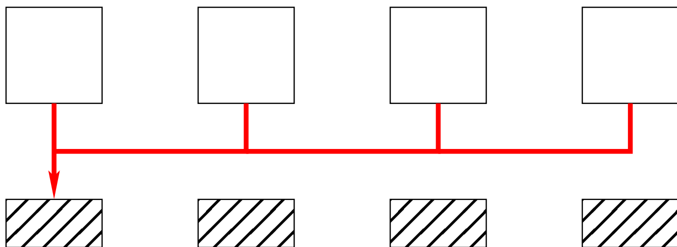
Caches and multicore: NUMA



$(4, 2.4\text{GHz}, l, g)$, but **Non-Uniform Memory Access** (NUMA)!

Dealing with NUMA: distribution types

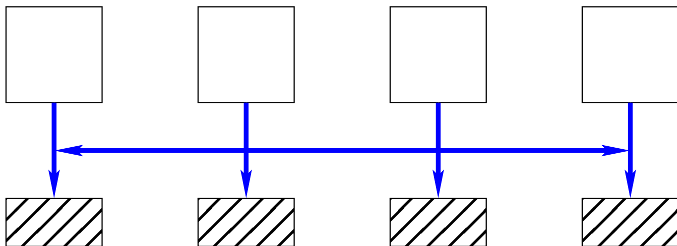
Implicit distribution, centralised local allocation:



If each processor moves data to the same single memory element, the **bandwidth is limited** by that of a single memory controller.

Dealing with NUMA: distribution types

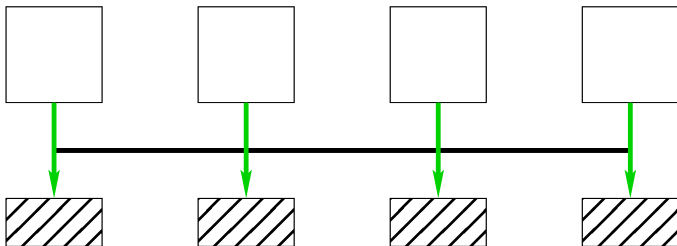
Implicit distribution, centralised interleaved allocation:



If each processor moves data from all memory elements, the bandwidth multiplies **if accesses are uniformly random**.

Dealing with NUMA: distribution types

Explicit distribution, distributed local allocation:



If each processor moves data from and to its own unique memory element, the **bandwidth multiplies**.

Bandwidth

CPU speeds stall, but Moore's Law now translates to an increasing amount of cores per die, i.e., **the effective flop rate of processors still rises** as it always has.

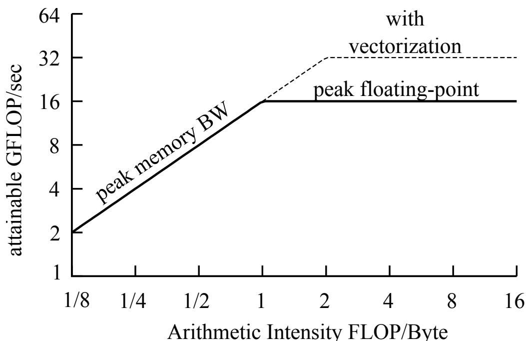
- But what about **bandwidth**?

Technology	Year	Speed
EDO	1970s	27 Mbyte/s
SDRAM	early 1990s	53 Mbyte/s
RDRAM	mid 1990s	1.2 Gbyte/s
DDR	2000	1.6 Gbyte/s
DDR2	2003	3.2 Gbyte/s
DDR3	2007	6.4 Gbyte/s
DDR3	2013	11 Gbyte/s

Will the **effective bandwidth per core** keep decreasing?

Bandwidth

Arithmetic intensity:



- If your computation has enough work per data element, it is **compute bound**. Otherwise it is **bandwidth bound**.
- If you are bandwidth bound, reducing your memory footprint, i.e., **compression**, directly results in faster execution.

(Image courtesy of Prof. Wim Vanroose, UA)

Applications

1 Shared-memory architectures and paradigms

2 Applications

3 Case study: Intel Xeon Phi

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s the current processor ID,
p the total number of processors (threads),
n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double α;' globally allocated:

- $\alpha = 0.0$
- for $i = s$ to n step p
- $\quad \alpha \mathrel{+}= x_i y_i$
- return α

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double α;' globally allocated:

- $\alpha = 0.0$
- for $i = s$ to n step p
- $\qquad \alpha \mathrel{+}= x_i y_i$
- return α

Data race! (for $n = p = 2$, output can be $x_0 y_0$, $x_1 y_1$, **or** $x_0 y_0 + x_1 y_1$)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[p]$;' globally allocated:

- for $i = s$ to n step p
- $\qquad \alpha_s \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_i$

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[p]$;' globally allocated:

- for $i = s$ to n step p
- $\quad\quad \alpha_s \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_i$

False sharing! (processors access and update the same cache lines)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $i = s$ to n step p
- $\quad\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $i = s$ to n step p
- $\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Inefficient cache use: $\Theta(pn)$ data movement.

(All threads access all cache lines)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $i = s \cdot \lceil n/p \rceil$ to $(s+1) \cdot \lceil n/p \rceil$
- $\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- synchronise
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

(Now inefficiency only at boundaries; $\mathcal{O}(n+p-1)$ data movement)

Central obstacles for SpMV multiplication

The second example application is the sparse matrix–vector multiplication

$$y = Ax.$$

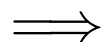
Three obstacles for an efficient shared-memory parallel sparse matrix–vector (SpMV) multiplication kernel:

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

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$



Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$

$\Longrightarrow$

$a_{0?}$

$x_?$

$\Longrightarrow$

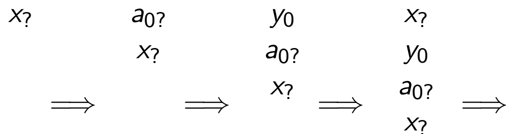
Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$$
\begin{array}{ccccccc}
x_? & & a_{0?} & & y_0 & & \\
 & & x_? & & a_{0?} & & \\
 & \Longrightarrow & & \Longrightarrow & x_? & \Longrightarrow &
\end{array}
$$

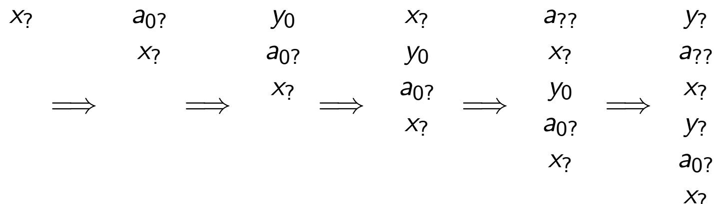
Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:



Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

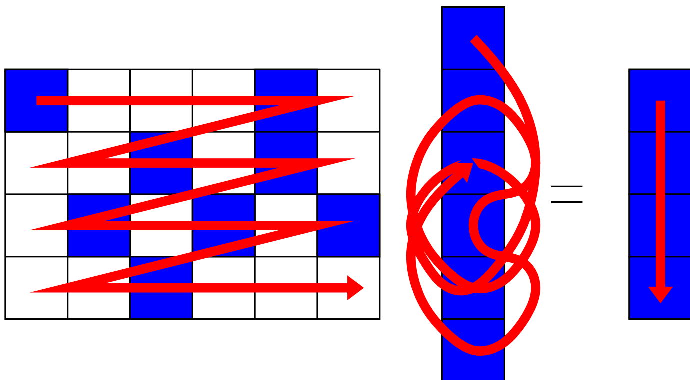


We cannot predict memory accesses in the sparse case:

- simple blocking is not possible.

Inefficient cache use

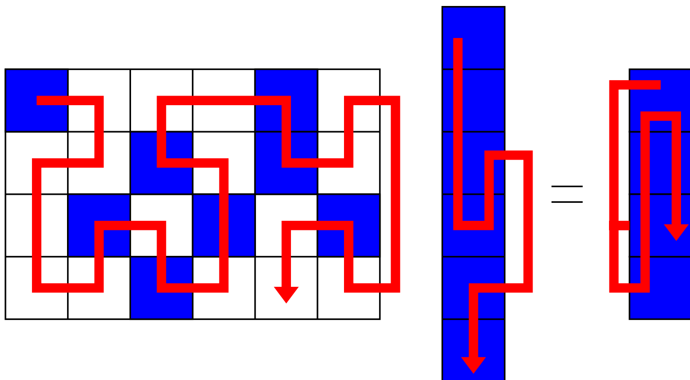
Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:



Accesses on the input vector are completely unpredictable.

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in an order defined by the **Hilbert curve**:



Accesses on both vectors have more **temporal locality**.

Bandwidth issues

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:

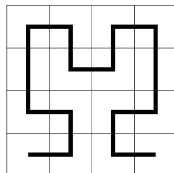
	CPU speed	Memory speed
1 core	$8.5 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s
8 cores	$68 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s

The SpMV multiplication is clearly bandwidth-bound on modern CPUs.

Sparse matrix storage

The coordinate format stores nonzeroes in arbitrary order:

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



COO:

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

Storage requirements:

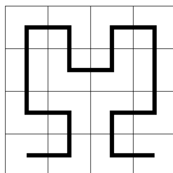
$$\Theta(3nz),$$

where nz is the number of nonzeroes in A.

SpMV multiplication

Multiplication using COO:

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



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

Sequential algorithm:

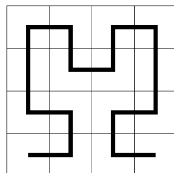
for $k = 0$ **to** $nz - 1$ **do**

  add $V_k \cdot x_{J_k}$ to y_{I_k}

SpMV multiplication

Multiplication using COO:

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



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

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

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

  add $V_k \cdot x_{J_k}$ to y_{I_k}

Is this OK?

Sparse matrix storage

Assuming a row-major order of nonzeroes enables **compression**:

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

CRS:

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

Storage requirements:

$$\Theta(2nz + m + 1).$$

SpMV multiplication

Multiplication using CRS:

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

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

Sequential 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

SpMV multiplication

Multiplication using CRS:

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

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

#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

SpMV multiplication

Parallel multiplication using Compressed Sparse Blocks:

$$A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,n/\beta-1} \\ \vdots & \ddots & \\ A_{m/\beta-1,0} & \cdots & A_{m/\beta-1,n/\beta-1} \end{pmatrix},$$

cilk_for $i = 0$ **to** $m/\beta - 1$
  for $k = \hat{I}_i$ **to** $\hat{I}_{i+1} - 1$
    do block-local SpMV using A_{i,J_k},
     the J_kth block of x, and
     the ith block of y

Ref.: Buluç, Fineman, Frigo, Gilbert, and Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks", Proc. 21st annual symposium on Parallelism in algorithms and architectures, pp. 233-244 (2009)

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Requires implicit (interleaved) allocation of all data.

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

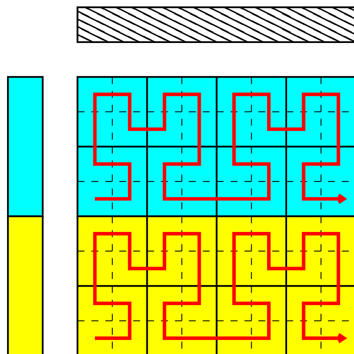
- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Requires implicit (interleaved) allocation of all data. But this does not play well with NUMA. Alternatives:

- 1D SpMV: distribute A and y rowwise.
- 2D SpMV: distribute A, x, and y.

NUMA: Parallel 1D SpMV

Distribute rows to processes, do local blocking and Hilbert ordering:



Allows for explicit (local) allocation of the sparse matrix A and the output vector y; x is implicitly distributed and interleaved.

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 (2013).

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory.

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory. Multiplication:

- Execute $y^{(s)} = A^{(s)}x$.

Implemented in POSIX Threads.

NUMA: Parallel 2D SpMV

2D SpMV

Input vector communication:

- retrieving values from x is called **fan-out**, and
- is implemented by using **bsp_get**.
- Elements from x are communicated in a one-to-many fashion.

Output vector communication:

- sending contributions to non-local y is **fan-in**.
- Implementation happens through **Bulk Synchronous Message Passing** (BSMP).
- Elements from y are communicated in a many-to-one fashion.

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

1: **for each** a_{ij} that is local to s **do**
2:     **if** x_j is not local **then**
3:         **bsp_get** x_j from remote process
4: **bsp_sync()**

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

1: **for each** a_{ij} that is local to s **do**
2: **if** x_j is not local **then**
3: **bsp_get** x_j from remote process
4: **bsp_sync()**
5: **for each** a_{ij} that is local to s **do**
6: add $a_{ij} \cdot x_j$ to y_i

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

1: **for each** a_{ij} that is local to s **do**
2:     **if** x_j is not local **then**
3:         **bsp_get** x_j from remote process
4: **bsp_sync()**
5: **for each** a_{ij} that is local to s **do**
6:     add $a_{ij} \cdot x_j$ to y_i
7:     **if** y_i is not local **then**
8:         **bsp_send** (y_i, i) to the owner of y_i
9: **bsp_sync()**

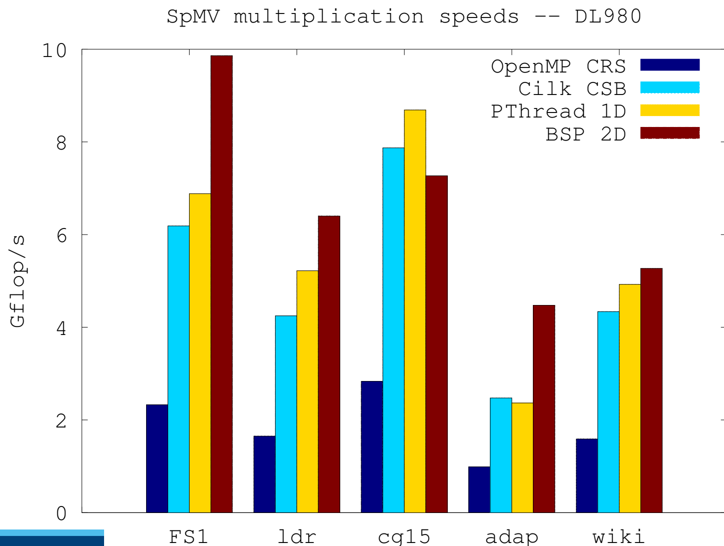
NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then, in BSP:

1: **for each** a_{ij} that is local to s **do**
2: **if** x_j is not local **then**
3: **bsp_get** x_j from remote process
4: **bsp_sync()**
5: **for each** a_{ij} that is local to s **do**
6: add $a_{ij} \cdot x_j$ to y_i
7: **if** y_i is not local **then**
8: **bsp_send** (y_i, i) to the owner of y_i
9: **bsp_sync()**
10: **while bsp_qsize()** > 0 **do**
11: $(\alpha, i) =$ **bsp_move()**
12: add α to y_i

everything is explicitly allocated!

Results



BSP 'direct get'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

BSP ‘direct get’

The ‘direct get’ is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP ‘get’ primitives,
- guarantee source data remains unchanged.

BSP 'direct get'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP 'get' primitives,
- guarantee source data remains unchanged.

Replacing those primitives with calls to `bsp_direct_get` allows **merging** this superstep with its following one, thus

saving a synchronisation step.

Ref.: Yzelman and Bisseling, "An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming", Concurrency and Computation: Practice and Experience 24(5), pp. 533-553 (2012).

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory.

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.

BSP 'hp send'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 2D SpMV

Step 1: fan-out. Request **contiguous** ranges of x.

```
typedef std::vector< fanQuadlet >::const_iterator IT;
for( IT it = fanIn.begin(); it != fanIn.end(); ++it ) {
    const unsigned long int src_P    = it->remoteP;
    const unsigned long int src_ind  = it->remoteStart;
    const unsigned long int dest_ind = it->localStart;
    const unsigned long int length   = it->length;
    bsp_direct_get( src_P,
                    x,
                    src_ind * sizeof( double ),
                    x + dest_ind,
                    length  * sizeof( double )
                  );
}
```

BSP 2D SpMV

Step 2: local SpMV multiplication:

```
if( A != NULL )
    A->zax( x, y ); //('zax' stands for z=Ax)
```

We use Compressed BICRS storage with the nonzeroes in row-major order. *A* is a pointer to an instance of a C++ class.

Yzelman and Roose, "High-level strategies for parallel shared-memory sparse matrix–vector multiplication", IEEE TPDS, 2013 (in press); paper: `http://dx.doi.org/10.1109/TPDS.2013.31`, software: `http://albert-jan.yzelman.net/software/#SL`

BSP 2D SpMV

Step 3: fan-in (I). Send **chunks** of row contributions.

```
//the tagsize is initialised to 2*sizeof( ULI )
//fanOut[ i ] has the following layout:
//{ ULI remoteP, localStart, remoteStart, length; }
typedef unsigned long int ULI;

for( ULI i = 0; i < fanOut.size(); ++i ) {
    const ULI dest_P    = fanOut[ i ].remoteP;
    const ULI src_ind   = fanOut[ i ].localStart;
    const ULI length    = fanOut[ i ].length;
    bsp_hpsend( dest_P,
                &( fanOut[ i ].remoteStart ),
                y + src_ind, length * sizeof( double ) );
}
bsp_sync();
```

BSP 2D SpMV

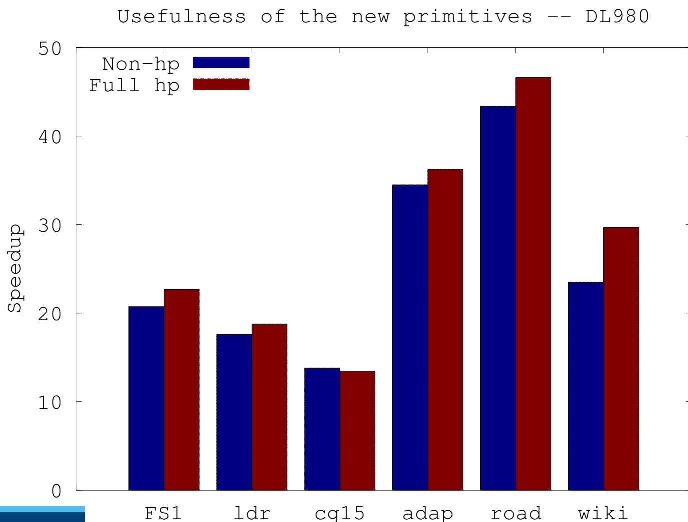
Step 3: fan-in (II). Handle incoming contributions.

```
unsigned long int *msg_tag;
double *msg_payload;
while( bsp_hpmove( (void**)&msg_tag,
                   (void**)&msg_payload
                 ) != SIZE_MAX ) {
    const unsigned long int y_dest = msg_tag[ 0 ];
    const unsigned long int length = msg_tag[ 1 ];
    for( unsigned long int i = 0; i < length; ++i )
        y[ y_dest + i ] += msg_payload[ i ];
}
```

This finishes our implementation of the 2D SpMV multiply.

Results – new primitives

We test the new primitives using the BSP 2D SpMV multiply:



Summary

We have seen

- hardware properties of modern shared-memory architectures,
- how this affects shared-memory programming and data locality,
- common pitfalls of non-BSP shared-memory programming like data races and false sharing (in OpenMP, Cilk, and PThreads),
- how shared-memory BSP programming avoids these issues, and
- how to attain high performance algorithms using BSP.

Case study: Intel Xeon Phi

1 Shared-memory architectures and paradigms

2 Applications

3 Case study: Intel Xeon Phi

Many integrated cores

The Xeon Phi is an accelerator which has

- 61 x86-type cores (one reserved for OS),
- support for 4 threads per core,
- on-board fast DDR5 memory (44 GB/s per controller),
- four groups of paired (2) memory controllers,
- the cores arranged around a ring interconnect,
- the memory groups uniformly distributed along this ring.

Thus one board effectively has 240 concurrent threads backed by a bandwidth of 350 GB/s.

Earlier work

Originally devised for vector computers:

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

ELLPACK:

$$A = \left\{ V : \begin{pmatrix} 4 & 1 & 3 \\ 2 & 3 & - \\ 1 & 2 & - \\ 7 & 1 & 1 \end{pmatrix}, \quad J : \begin{pmatrix} 0 & 1 & 2 \\ 2 & 3 & - \\ 0 & 3 & - \\ 0 & 2 & 3 \end{pmatrix} \right.$$

Formatted as an $m \times \max_j \{A \ni a_{ij} \neq 0\}$ dense matrix.

- causes **fill-in**.

Earlier work

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

Sliced ELLPACK (SELLPACK):

$$A = \begin{cases} V: & \left\{ \begin{pmatrix} 4 & 1 & 3 \\ 7 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix} \right\} \\ J: & \left\{ \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 3 \end{pmatrix}, \begin{pmatrix} 2 & 3 \\ 0 & 3 \end{pmatrix} \right\} \end{cases}$$

- Fixed-size slice parameter s (here, $s = 2$),
- improved performance when using row permutations P.

Earlier work

Other ELLPACK-based data structures:

- Hybrid (ELLPACK/COO)

  Ref.: Bell, Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors", Proc. High Performance Computing Networking, Storage and Analysis (2008)

- SELLPACK-R (zero-length encoded fill-in)

  Ref.: Vázquez, J. Fernández, and E. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs", Concurr. Comput.: Pract. Exper. 23, pp. 815–826 (2011).

- ELLPACK Sparse Block
  (ESB; restricted row partitioning, bitmasked fill-in)

  Ref.: Liu, Smelyanskiy, Chow, and Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors", Proc. 27th intern. conf. on supercomputing (ICS '13), doi: 10.1145/2464996.2465013 (2013).

The latter reminds us of similar efforts for CPUs:

- Bitmasked Compressed Sparse Blocks (mCSB)

  Ref.: Buluç, Williams, Oliker, and Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix–vector multiplication", Proc. of the Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, pp. 721-733 (2011).

Earlier work

Blocked CRS; integrated in OSKI (see Im and Vuduc).

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

$$A = \begin{cases} V_{\text{nz}}: & [4\ 1\ 3\ 2\ 3\ 1\ 2\ 7\ 0\ 1\ 1] \\ V_{\text{blk}}: & [0\ 3\ 5\ 6\ 7\ 11] \\ J: & [0\ 2\ 0\ 3\ 0] \\ \hat{I}: & [0\ 1\ 2\ 4\ 5] \end{cases}$$

Dense blocks: $4, 1, 3$ / $2, 3$ / 1 / 2 / $7, 0, 1, 1$,

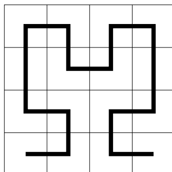
$\Theta(nz + 2nblk + m)$ storage.

Ref.: Pinar and Heath, "Improving Performance of Sparse Matrix-Vector Multiplication", Proc. ACM/IEEE Conf. on Supercomputing (1999).

New strategy

Correct data structure for row-wise 3-blocking:

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



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

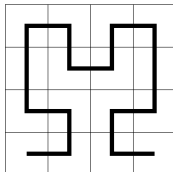
Fill-in in red, regular BICRS data in blue.

Note its similarity to Blocked CRS (it is different though!)

New strategy

Example multiplication: **block 1**

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



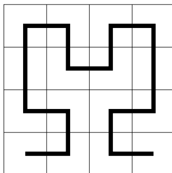
$$A = \begin{cases} V & [7\ 1\ 4\ /\ 0\ 0\ 1\ /\ 2\ 3\ 0\ /\ 3\ 0\ 2\ /\ 1\ 0\ 0\ /\ 1\ 0\ 0] \\ \Delta J & [0\ 1\ 5\ 1\ 3\ 1\ 4] \\ \Delta I & [3\ 2\ 0\ /\ \text{-}2\ \text{-}1\ 1\ /\ 2\ \text{-}\ \text{-}] \end{cases}$$

- Multiply $(7, 1, 4)$ with (x_0, x_0, x_0) and add to (y_3, y_2, y_0).

New strategy

Example multiplication: **block 2**

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



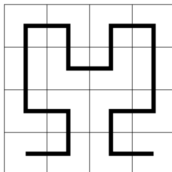
$$A = \begin{cases} V & [7\ 1\ 4\ /\ 0\ 0\ 1\ /\ 2\ 3\ 0\ /\ 3\ 0\ 2\ /\ 1\ 0\ 0\ /\ 1\ 0\ 0] \\ \Delta J & [0\ 1\ 5\ 1\ 3\ 1\ 4] \\ \Delta I & [3\ 2\ 0\ /\ \text{-2}\ \text{-1}\ 1\ /\ 2\ \text{-}\ \text{-}] \end{cases}$$

- Multiply $(0, 0, 1)$ with (x_1, x_1, x_1) and add to (y_3, y_2, y_0).

New strategy

Example multiplication: **block 3**

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



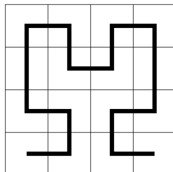
$$A = \begin{cases} V & [7\ 1\ 4\ /\ 0\ 0\ 1\ /\ 2\ 3\ 0\ /\ 3\ 0\ 2\ /\ 1\ 0\ 0\ /\ 1\ 0\ 0] \\ \Delta J & [0\ 1\ 5\ 1\ 3\ 1\ 4] \\ \Delta I & [3\ 2\ 0\ /\ \text{-2}\ \text{-1}\ 1\ /\ 2\ \text{-}\ \text{-}] \end{cases}$$

- Multiply $(2, 3, 0)$ with (x_2, x_2, x_2) and add to (y_1, y_0, y_2).

New strategy

Example multiplication: **block 4**

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



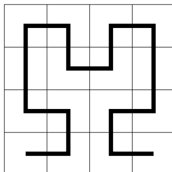
$$A = \begin{cases} V & [7\ 1\ 4\ /\ 0\ 0\ 1\ /\ 2\ 3\ 0\ /\ 3\ 0\ 2\ /\ 1\ 0\ 0\ /\ 1\ 0\ 0] \\ \Delta J & [0\ 1\ 5\ 1\ 3\ 1\ 4] \\ \Delta I & [3\ 2\ 0\ /\ \text{-}2\ \text{-}1\ 1\ /\ 2\ \text{-}\ \text{-}] \end{cases}$$

- Multiply $(3, 0, 2)$ with (x_3, x_3, x_3) and add to (y_1, y_0, y_2).

New strategy

Example multiplication: **block 5**

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



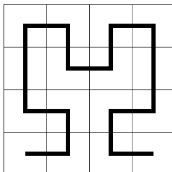
$$A = \begin{cases} V & [7\ 1\ 4\ /\ 0\ 0\ 1\ /\ 2\ 3\ 0\ /\ 3\ 0\ 2\ /\ 1\ 0\ 0\ /\ 1\ 0\ 0] \\ \Delta J & [0\ 1\ 5\ 1\ 3\ 1\ 4] \\ \Delta I & [3\ 2\ 0\ /\ \text{-}2\ \text{-}1\ 1\ /\ 2\ \text{-}\ \text{-}] \end{cases}$$

- Multiply $(1, 0, 0)$ with (x_2, x_2, x_2) and add to (y_3, y_-, y_-).

New strategy

Example multiplication: **block 6**

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



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

- Multiply $(1, 0, 0)$ with (x_3, x_3, x_3) and add to (y_3, y_-, y_-).

New strategy

The Xeon Phi supports **gather/scatter** instructions. This is the key technology for the new strategy. In **intrinsics**:

```
__m512d _mm512_i32logather_pd(
                                    indices,
                                    base_pointer,
                                    scale
                                  );
```

Suppose

- x is a vector of 512 doubles, and
- $I = (0, 9, 4, 1, 256, 511, 510, 509)$.

Then

```
_mm512_i32logather_pd( I, x, 8 ) ==
( x[0], x[9], x[4], x[1], x[256], x[511], x[510], x[509] ).
```

New strategy

The Intel Xeon Phi has a vectorisation width of 512 bits:

- arithmetic operations on vectors of 8 doubles only cost a single vectorised operation,
- data has to be loaded into vector registers first (no solution to bandwidth issues).

1D column blocking for the Xeon Phi, coding perspective (this is very close to the production code):

```
void oICRS< double, 1, 8, int16_t >::zax(    //z=Ax
        const double *__restrict__ pDataX,
              double *__restrict__ pDataZ
) {
        ...
```

New strategy

```
...
        double   *__restrict__ pDataA   = ds;
        int16_t *__restrict__ pIncRow  = r_ind;
        int16_t *__restrict__ pIncCol  = c_ind;

        //define buffers
        __m512i c_ind_buffer;
        __m512d input_buffer;
        __m512d value_buffer;
        __m512d outputbuffer;
        __m512i zeroF = _mm512_set_epi32(
                1, 1, 1, 1, 1, 1, 1, 0
        );
...
```

New strategy

```
...
        //initialise kernel
        outputbuffer = _mm512_setzero_pd();

        //fill c_ind_buffer
        c_ind_buffer = _mm512_extload_epi32( pIncCol,
                _MM_UPCONV_EPI32_UINT16,
                _MM_BROADCAST32_NONE,
                _MM_HINT_NT
        );

        //shift input vector
        pDataX += *pIncCol;
...
```

New strategy

```
...
        //move pIncCol up one block
        pIncCol += block_length;

        //reset start column index; c_ind_buffer[ 0 ] = 0;
        c_ind_buffer = _mm512_mullo_epi32(
                               c_ind_buffer, zeroF );

        //shift output vector
        pDataZ += *pIncRow++;

        //start kernel
        while( pDataA < pDataAend ) {
...
```

New strategy

```
...
        //process a single row
        while( pDataX < pDataXend ) {
            //fill input buffer
            input_buffer = _mm512_i32loextgather_pd(
                c_ind_buffer, pDataX, _MM_UPCONV_PD_NONE,
                8, _MM_HINT_NONE );

            //fill nonzero buffer
            value_buffer = _mm512_load_pd( pDataA );

            //do vectorised multiply-add
            outputbuffer = _mm512_fmadd_pd( value_buffer,
                               input_buffer, outputbuffer );
...
```

New strategy

```
...
                //shift input data
                pDataA += block_length;

                //fill c_ind_buffer
                c_ind_buffer = _mm512_extload_epi32(
                                pIncCol,
                               _MM_UPCONV_EPI32_UINT16,
                               _MM_BROADCAST32_NONE,
                               _MM_HINT_NT );

                //reset start column index
                c_ind_buffer = _mm512_mullo_epi32(
                               c_ind_buffer, zeroF );
...
```

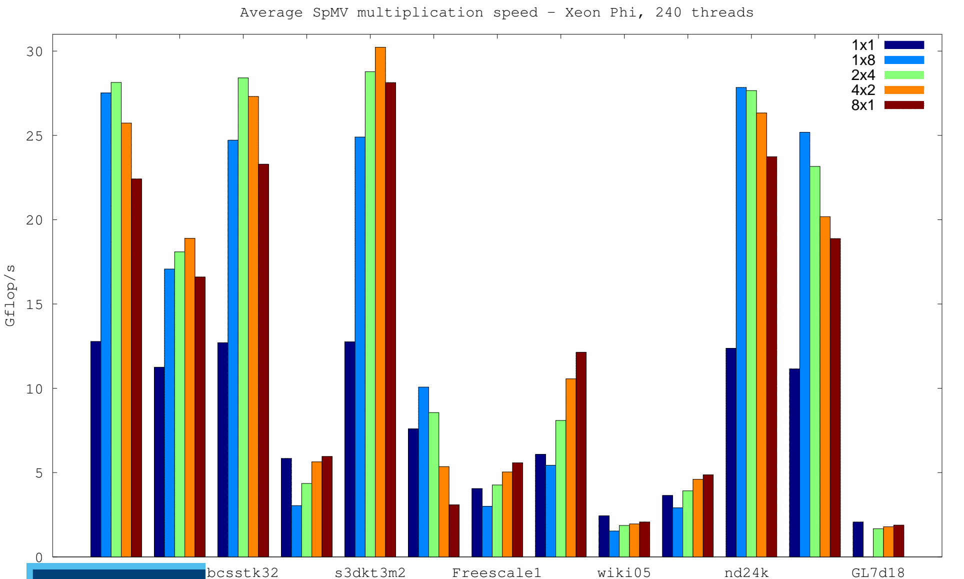
New strategy

```
...
            //shift input vector
            pDataX += *pIncCol;

            //move pIncCol up
            pIncCol += block_length;

        }  //finished processing this row

        //reduce row contribution
        *pDataZ +=_mm512_reduce_add_pd( outputbuffer );

        //reset sums buffer
        outputbuffer = _mm512_setzero_pd();
...
```

New strategy

```
...
            //row jump, shift back input vector
            pDataX -= this->noc;

            //shift output vector
            pDataZ += *pIncRow++;
      }
}
//end z=Ax specialisation for 16-byte unsigned integer
//indices, double data types, 1x8 blocks.
```

Results



Summary

We have seen:

- one of the newer shared-memory architectures,
- how existing methods may have to be tuned to new architectures,
- that the performance of some kernels depend heavily on its input.

New techniques:

- vectorisation,
- sparse matrix blocking,
- alternative data storage families.