Sparse Computations and Multi-BSP

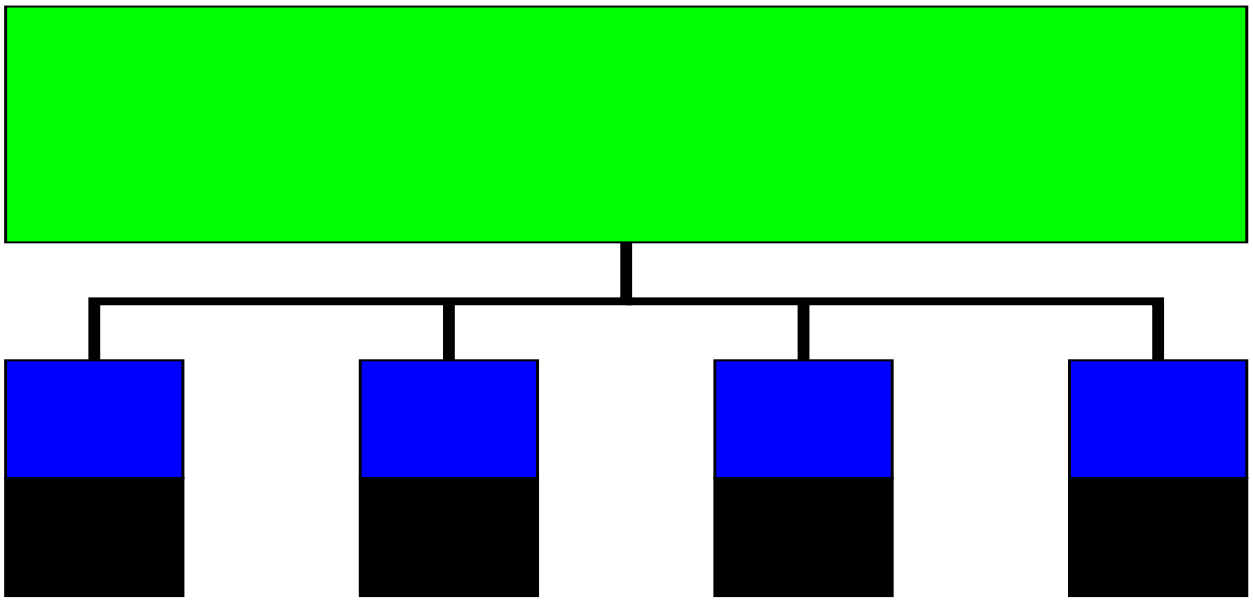
Albert-Jan Yzelman

October 11, 2016

Parallel Computing & Big Data
Huawei Technologies France

BSP

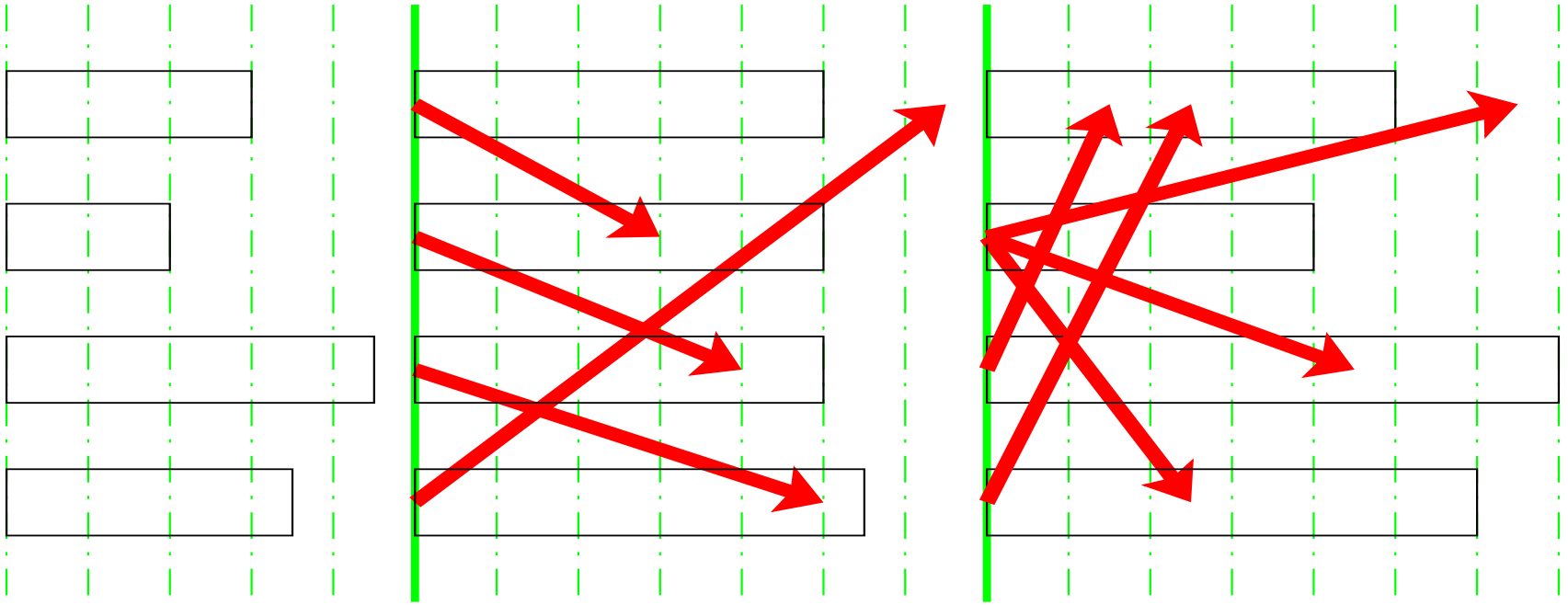
BSP machine $=$ { sequential processor } $+$ interconnect



The machine is described entirely by (p, g, L):

- strobing synchronisation,
- homogeneous processing,
- uniform full-duplex network,

BSP

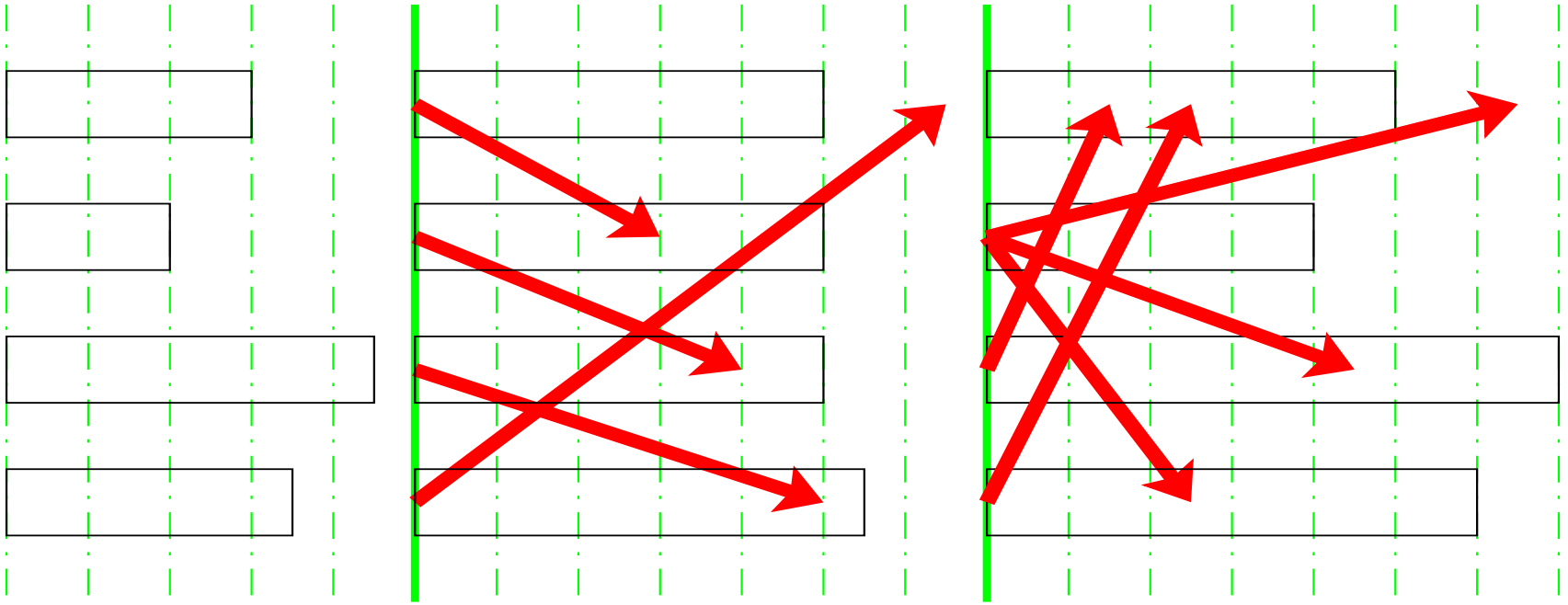


BSP algorithm:

- strobing barriers
- full overlap
- h-relation bottlenecks: $\max_s\{sent_s, recv_s\}$
- work balance

L. G. Valiant, *A bridging model for parallel computation*, CACM, 1990

BSP



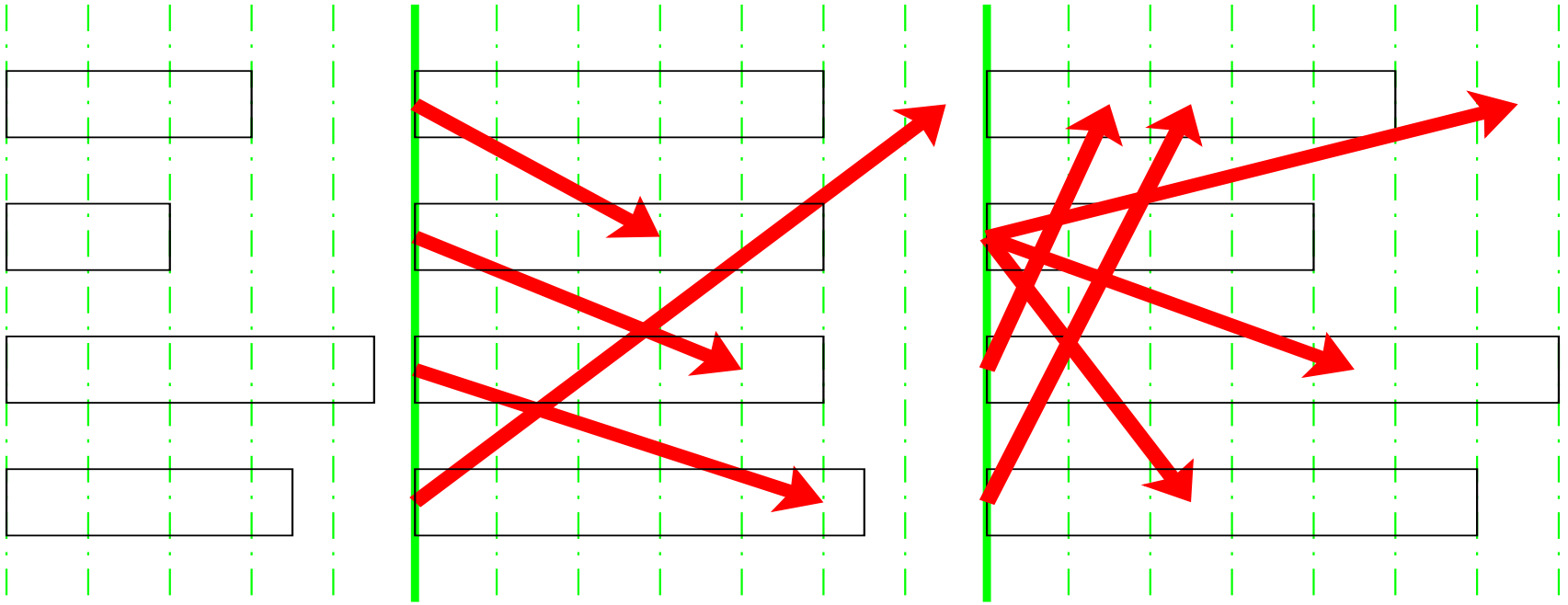
BSP cost:

$$T_p = \max_s w_s^{(0)} + L + \max\{\max_s w_s^{(1)} + L, \max_s h_s^{(1)} g + L\} + \ldots$$

Separation of computation vs. communication.

L. G. Valiant, *A bridging model for parallel computation*, CACM, 1990

BSP



BSP cost:

$$T_p = \max_s w_s^{(0)} + L + \max\{\max_s w_s^{(1)} + L, \max_s h_s^{(1)} g + L\} + \ldots$$

Separation of algorithm vs. hardware.

L. G. Valiant, *A bridging model for parallel computation*, CACM, 1990

Immortal algorithms

The BSP paradigm, allows the design of **immortal algorithms**:

- given a problem to compute
- given a BSP computer (p, g, l)
- find the BSP algorithm that attains provably minimal cost.

E.g., fast Fourier transforms, matrix-matrix multiplication.

Thinking in Sync: the Bulk-Synchronous Parallel approach to large-scale computing. Bisseling and Yzelman, ACM Hot Topic '16.

`http://www.computingreviews.com/hottopic/hottopic_essay.cfm?htname=BSP`

BSP sparse matrix–vector multiplication

Variables A_s, x_s, y_s are local versions of the global variables A, x, y distributed according to π_A, π_x, π_y.

BSP sparse matrix–vector multiplication

Variables A_s, x_s, y_s are local versions of the global variables A, x, y distributed according to π_A, π_x, π_y.

1: **for** $j \mid \exists a_{ij} \neq 0 \in A_s$ and $\pi_x(j) \neq s$ **do**
2: **get** $x_{\pi_x(j),j}$
3: **sync** {execute *fan-out*}

BSP sparse matrix–vector multiplication

Variables A_s, x_s, y_s are local versions of the global variables A, x, y distributed according to π_A, π_x, π_y.

1: **for** $j \mid \exists a_{ij} \neq 0 \in A_s$ and $\pi_x(j) \neq s$ **do**
2: **get** $x_{\pi_x(j),j}$
3: **sync** {execute *fan-out*}
4: $y_s = A_s x_s$ {local multiplication stage}

BSP sparse matrix–vector multiplication

Variables A_s, x_s, y_s are local versions of the global variables A, x, y distributed according to π_A, π_x, π_y.

1: **for** $j \mid \exists a_{ij} \neq 0 \in A_s$ and $\pi_x(j) \neq s$ **do**
2: **get** $x_{\pi_x(j),j}$
3: **sync** {execute *fan-out*}
4: $y_s = A_s x_s$ {local multiplication stage}
5: **for** $i \mid \exists a_{ij} \in A_s$ and $\pi_y(i) \neq s$ **do**
6: **send** $(i, y_{s,i})$ to $\pi_y(i)$
7: **sync** {execute *fan-in*}

BSP sparse matrix–vector multiplication

Variables A_s, x_s, y_s are local versions of the global variables A, x, y distributed according to π_A, π_x, π_y.

1: **for** $j \mid \exists a_{ij} \neq 0 \in A_s$ and $\pi_x(j) \neq s$ **do**
2: **get** $x_{\pi_x(j),j}$
3: **sync** {execute *fan-out*}
4: $y_s = A_s x_s$ {local multiplication stage}
5: **for** $i \mid \exists a_{ij} \in A_s$ and $\pi_y(i) \neq s$ **do**
6: **send** $(i, y_{s,i})$ to $\pi_y(i)$
7: **sync** {execute *fan-in*}
8: **for** all (i, α) received **do**
9: add α to $y_{s,i}$

Rob H. Bisseling, "Parallel Scientific Computation", Oxford Press, 2004.

BSP sparse matrix–vector multiplication

Suppose π_A assigns every nonzero $a_{ij} \in A$ to processor $\pi_A(i,j)$. If

1. $\pi_y(i) \in \{s \mid \exists a_{ij} \in A,\ \pi_A(i,j) = s\}$ and
2. $\pi_x(j) \in \{s \mid \exists a_{ij} \in A,\ \pi_A(i,j) = s\}$;

BSP sparse matrix–vector multiplication

Suppose π_A assigns every nonzero $a_{ij} \in A$ to processor $\pi_A(i,j)$. If

1. $\pi_y(i) \in \{s \mid \exists a_{ij} \in A,\ \pi_A(i,j) = s\}$ and
2. $\pi_x(j) \in \{s \mid \exists a_{ij} \in A,\ \pi_A(i,j) = s\}$;

then

- fan-out communication scatters $\sum_j \left(\lambda_j^{\text{col}} - 1\right)$ elements from x,
- fan-in communication gathers $\sum_i (\lambda_i^{\text{row}} - 1)$ elements from y,

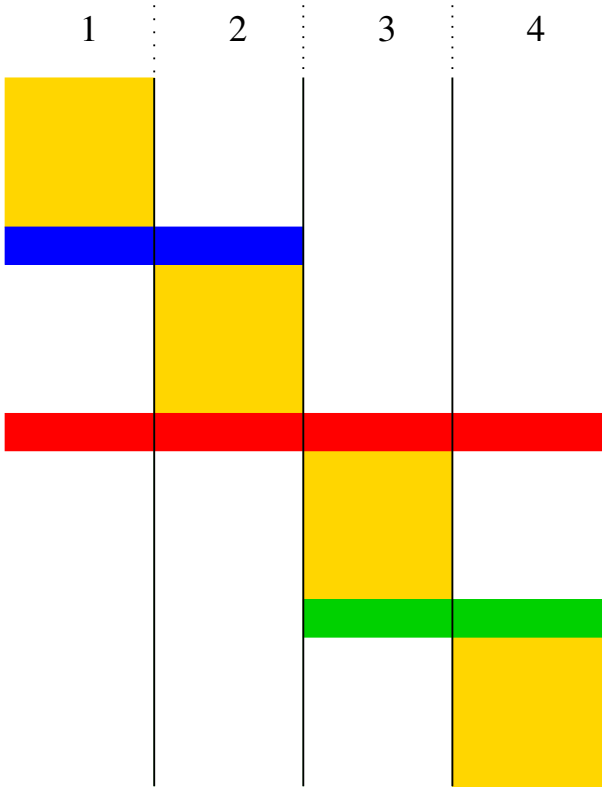
where

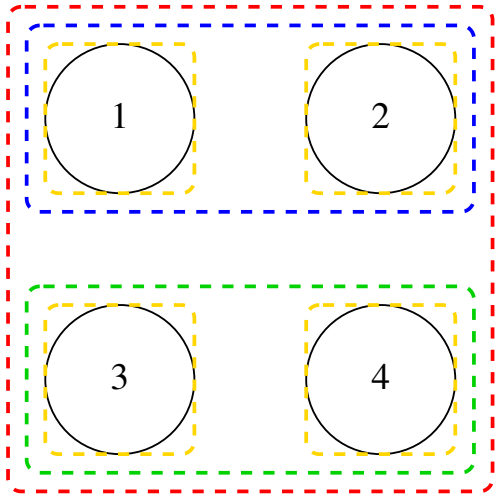
$$\begin{aligned} \lambda_i^{\text{row}} &= |\{s \mid \exists a_{ij} \in A_s\}| \text{ and} \\ \lambda_j^{\text{col}} &= |\{s \mid \exists a_{ij} \in A_s\}|. \end{aligned}$$

Minimising the $\lambda - 1$ metric minimises total communication *volume*.

BSP sparse matrix–vector multiplication

Partitioning combined with reordering illustrates clear separators:

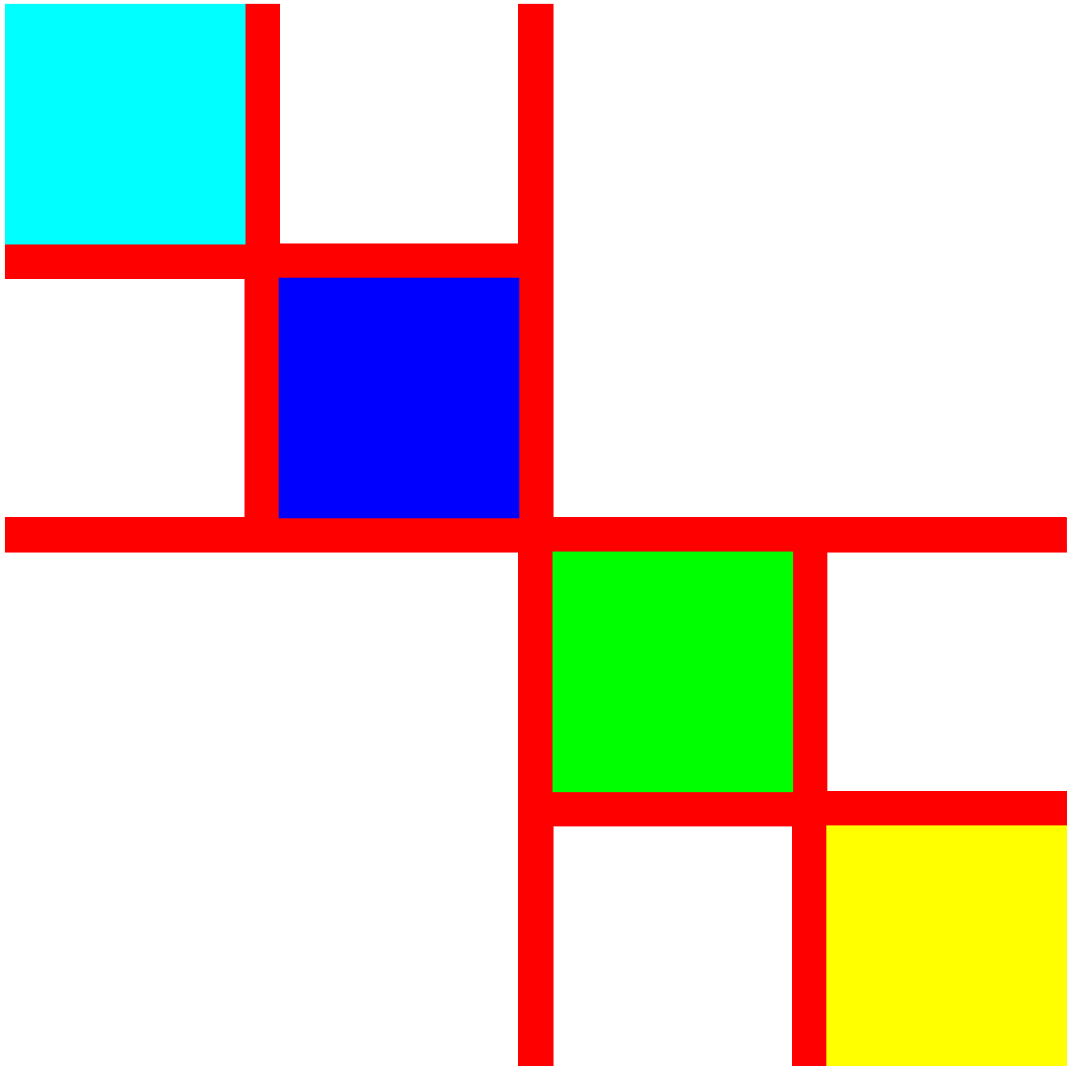




- Group nonzeroes a_{ij} for which $\pi_A(i) = \pi_A(j)$,
- permute rows i with $\lambda_i > 1$ in between,
- apply recursive bipartitioning.

BSP sparse matrix–vector multiplication

When partitioning in both dimensions:



BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

Block: $\frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l.$

BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

$$\text{Block:} \quad \frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l.$$

$$\text{Row 1D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + gh_{\text{fan-out}} + l.$$

BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

$$\text{Block:} \quad \frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l.$$

$$\text{Row 1D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + gh_{\text{fan-out}} + l.$$

$$\text{Col 1D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + gh_{\text{fan-in}} + l.$$

BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

$$\text{Block:} \quad \frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l.$$

$$\text{Row 1D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + gh_{\text{fan-out}} + l.$$

$$\text{Col 1D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + gh_{\text{fan-in}} + l.$$

$$\text{Full 2D:} \quad \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + g(h_{\text{fan-out}} + h_{\text{fan-in}}) + 2l.$$

BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

$$\begin{aligned}
\text{Block:} &\quad \frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l. \\
\text{Row 1D:} &\quad \frac{2nz(A)}{p}(1+\epsilon) + gh_{\text{fan-out}} + l. \\
\text{Col 1D:} &\quad \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + gh_{\text{fan-in}} + l. \\
\text{Full 2D:} &\quad \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + g(h_{\text{fan-out}} + h_{\text{fan-in}}) + 2l.
\end{aligned}$$

Memory overhead (buffers):

$$\Theta\left(\sum_i (\lambda_i^{\text{row}} - 1) + \sum_j \left(\lambda_i^{\text{col}} - 1\right)\right) = \mathcal{O}\left(p \sum_{\lambda:\lambda^{\text{row}}\cup\lambda^{\text{col}}} \mathbf{1}_{\lambda>1}\right).$$

BSP sparse matrix–vector multiplication

Classical worst-case bounds (in flops):

$$
\begin{aligned}
\text{Block:} \quad & \frac{2nz(A)}{p}(1+\epsilon) + n/p(\sqrt{p}-1)(2g+1) + 2l. \\
\text{Row 1D:} \quad & \frac{2nz(A)}{p}(1+\epsilon) + gh_{\text{fan-out}} + l. \\
\text{Col 1D:} \quad & \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + gh_{\text{fan-in}} + l. \\
\text{Full 2D:} \quad & \frac{2nz(A)}{p}(1+\epsilon) + \max_s recv_s^{\text{fan-in}} + g(h_{\text{fan-out}} + h_{\text{fan-in}}) + 2l.
\end{aligned}
$$

Memory overhead (buffers):

$$
\Theta\left(\sum_i (\lambda_i^{\text{row}} - 1) + \sum_j \left(\lambda_i^{\text{col}} - 1\right)\right) = \mathcal{O}\left(p \sum_{\lambda : \lambda^{\text{row}} \cup \lambda^{\text{col}}} \mathbf{1}_{\lambda > 1}\right).
$$

Depending on the higher-level algorithm:

- fan-in latency can be hidden behind other kernels,
- fan-out latency can be hidden as well.

Multi-BSP

Multi-BSP computer = p (subcomputers **or** processors) +
M bytes of local memory +
an interconnect

Multi-BSP

Multi-BSP computer $=$ p (subcomputers **or** processors) $+$
M bytes of local memory $+$
an interconnect

A total of $4L$ parameters: $(p_0, g_0, l_0, M_0, \ldots, p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Advantages:

- memory-aware,
- non-uniform!

Multi-BSP

Multi-BSP computer = p (subcomputers **or** processors) +
M bytes of local memory +
an interconnect

A total of $4L$ parameters: $(p_0, g_0, l_0, M_0, \ldots, p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Advantages:

- memory-aware,
- non-uniform!

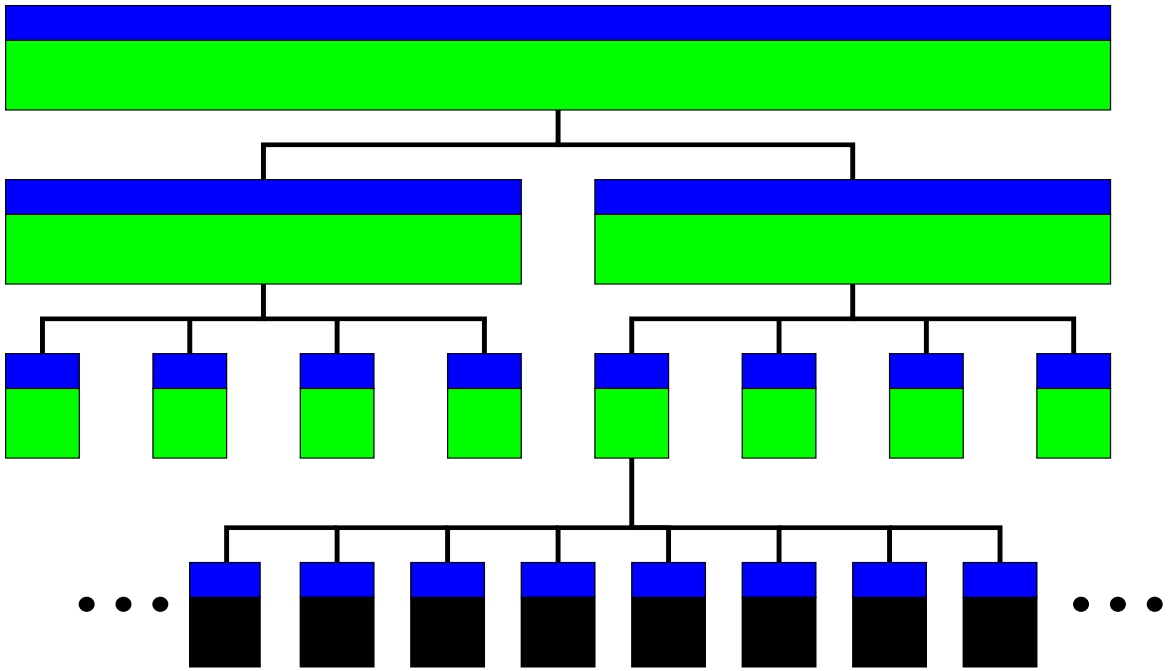
Disadvantages:

- (likely) harder to prove optimality.

L. G. Valiant, *A bridging model for multi-core computing*, CACM 2011.

Multi-BSP

An example with $L = 3$ quadlets (p, g, l, M):



$$\mathcal{C} = (2, g_0, l_0, M_0)\ (4, g_1, l_1, M_1)\ (8, g_2, l_2, M_2)$$

Each quadlet runs its own BSP SPMD program.

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

- define process 0 at level -1 as the Multi-BSP root.
- let process s at level k have parent t at level $k-1$.
- define $(A_{-1,0}, x_{-1,0}, y_{-1,0}) = (A, x, y)$, the original input.

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

- define process 0 at level -1 as the Multi-BSP root.
- let process s at level k have parent t at level $k-1$.
- define $(A_{-1,0}, x_{-1,0}, y_{-1,0}) = (A, x, y)$, the original input.
- variables $A_{k,s}, x_{k,s}, y_{k,s}$ are local versions of $A_{k-1,t}, x_{k-1,t}, y_{k-1,t}$,
- $\{A, x, k\}_{k-1,t}$ was distributed into $\tilde{p}_{k-1}$ parts,

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

- define process 0 at level -1 as the Multi-BSP root.
- let process s at level k have parent t at level $k-1$.
- define $(A_{-1,0}, x_{-1,0}, y_{-1,0}) = (A, x, y)$, the original input.
- variables $A_{k,s}, x_{k,s}, y_{k,s}$ are local versions of $A_{k-1,t}, x_{k-1,t}, y_{k-1,t}$,
- $\{A, x, k\}_{k-1,t}$ was distributed into $\tilde{p}_{k-1}$ parts,
- where $\tilde{p}_{k-1} \geq p_{k-1}$ is such that all $\{A, x, y\}_{k,s}$ fit into M_k bytes.

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

- define process 0 at level -1 as the Multi-BSP root.
- let process s at level k have parent t at level $k-1$.
- define $(A_{-1,0}, x_{-1,0}, y_{-1,0}) = (A, x, y)$, the original input.
- variables $A_{k,s}, x_{k,s}, y_{k,s}$ are local versions of $A_{k-1,t}, x_{k-1,t}, y_{k-1,t}$,
- $\{A, x, k\}_{k-1,t}$ was distributed into $\tilde{p}_{k-1}$ parts,
- where $\tilde{p}_{k-1} \geq p_{k-1}$ is such that all $\{A, x, y\}_{k,s}$ fit into M_k bytes.

```
1: do
2:     for j = 0 to p̃ step p
3:         get {A}_{k,j} from parent
4:         down
5: while(up)
```

Mandatory input data movement only.

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

```
1:  do
2:      ...
3:      for j = 0 to p̃ step p
4:          get {A, x, y}_{k,j} from parent
5:          ...
6:          if(not down)
7:              compute y_{k,j} = A_{k,j} x_{k,j} {only executed on leafs}
8:          ...
9:          put y_{k,j} into parent
10:         ...
11: while(up)
```

Mandatory and mixed mandatory/overhead data movement.
Minimal required work only.

Multi-BSP SpMV multiplication

SPMD-style Multi-BSP SpMV multiplication:

1: **do**
2: $\forall j$, **get** separator $\tilde{x}_{k,j}$ and initialise $\tilde{y}_{k,j}$ iff j mod $p = s$
3: **for** $j = 0$ **to** $\tilde{p}$ **step** p
4: **get** $\{A, x, y\}_{k,j}$ from parent
5: **sync**
6: **if**(not **down**)
7: compute $y_{k,j} = A_{k,j}x_{k,j}$ {*only executed on leafs*}
8: perform fan-in on seperator $\tilde{y}_{k,j}$
9: **put** $y_{k,j}$ into parent
10: **sync**
11: **put** $\tilde{y}_{k,j}$ into parent and **sync**
12: **while**(**up**)

Mandatory costs plus overhead. Split vectors: $\{x, y\}_s$ versus $\{\tilde{x}, \tilde{y}\}_s$.

Flat partitioning for Multi-BSP

Can we reuse existing partitioning techniques?

1 Partition $A = A_0 \cup \ldots A_{p-1}$ with $p = \pi_{l=0}^{L-1} p_l$?

No: A_s, x_s, y_s may not fit in M_{L-1}.

Flat partitioning for Multi-BSP

Can we reuse existing partitioning techniques?

1. Partition $A = A_0 \cup \ldots A_{p-1}$ with $p = \pi_{l=0}^{L-1} p_l$?
2. Find minimal k to partition A into s.t. $\{A, x, y\}_i$ fits into M_{L-1}?
   - Very similar to previous work!
     - Y. and Bisseling, "Cache-oblivous sparse matrix–vector multiplication by using sparse matrix partitioning", SISC, 2009.
     - Y. and Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication", Parallel Computing, 2011.

Flat partitioning for Multi-BSP

Can we reuse existing partitioning techniques?

1. Partition $A = A_0 \cup \ldots A_{p-1}$ with $p = \pi_{l=0}^{L-1} p_l$?
2. Find minimal k to partition A into s.t. $\{A, x, y\}_i$ fits into M_{L-1}?
   - Very similar to previous work!
     - Y. and Bisseling, "Cache-oblivous sparse matrix–vector multiplication by using sparse matrix partitioning", SISC, 2009.
     - Y. and Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication", Parallel Computing, 2011.
3. Hierarchical partitioning?
   - $A = A_0 \cup \ldots \cup A_{k_0}$,
   - $A_i = A_{i,0} \cup \ldots \cup A_{i,k_1}$, etc.
   - solves assignment issue.

Flat partitioning for Multi-BSP

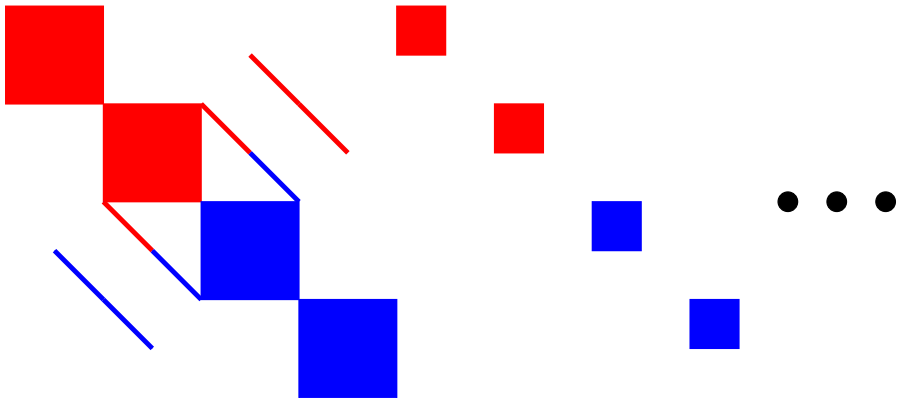
Can we reuse existing partitioning techniques?

1. Partition $A = A_0 \cup \ldots A_{p-1}$ with $p = \pi_{l=0}^{L-1} p_l$?
2. Find minimal k to partition A into s.t. $\{A, x, y\}_i$ fits into M_{L-1}?
   - Very similar to previous work!
     - Y. and Bisseling, "Cache-oblivous sparse matrix–vector multiplication by using sparse matrix partitioning", SISC, 2009.
     - Y. and Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication", Parallel Computing, 2011.
3. Hierarchical partitioning?
   - $A = A_0 \cup \ldots \cup A_{k_0}$,
   - $A_i = A_{i,0} \cup \ldots \cup A_{i,k_1}$, etc.
   - solves assignment issue.

However,

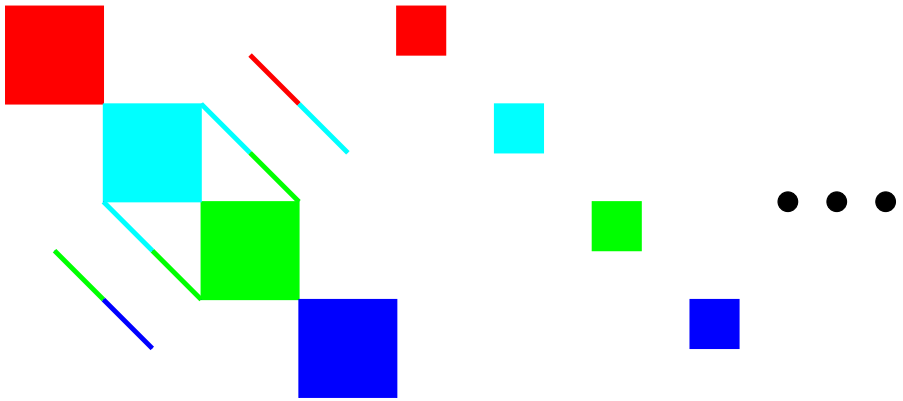
all of these do *not* take into account different g_l!

Hierarchical partitioning



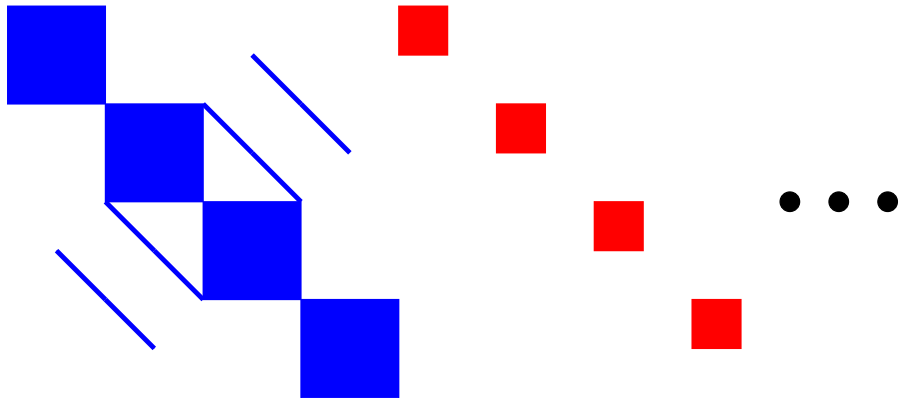
	Upper level	Lower level
Fan-out	$6g_0$	
Fan-in	$2g_0$	
Total:	$8g_0 + \ldots$	

Hierarchical partitioning



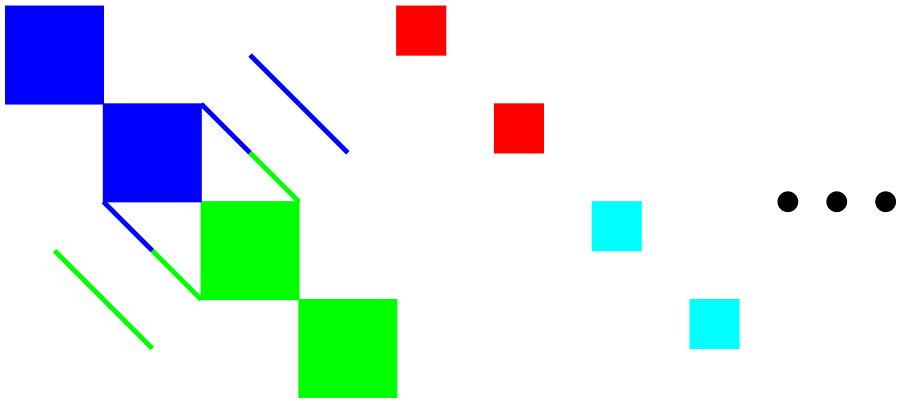
	Upper level	Lower level
Fan-out	$6g_0$	0
Fan-in	$2g_0$	0
Total:	$8g_0$	

Hierarchical partitioning



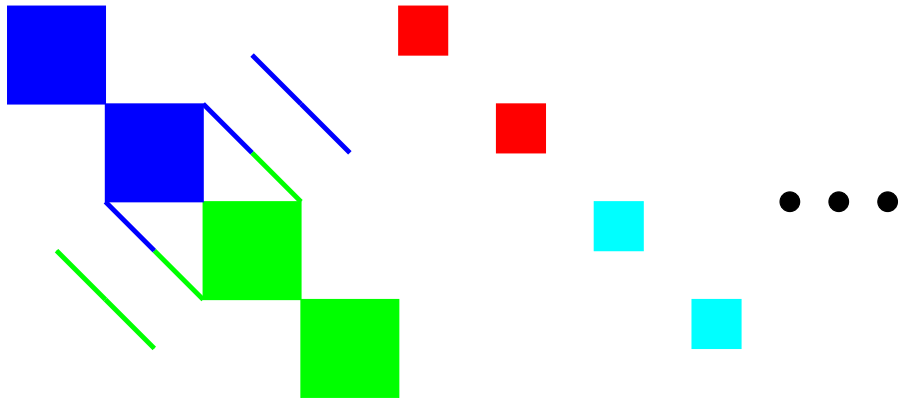
	Upper level	Lower level
Fan-out	0	
Fan-in	$4g_0$	
Total:	$4g_0 + \ldots$	
Previous:	$8g_0$	

Hierarchical partitioning



	Upper level	Lower level
Fan-out	0	$6g_1$
Fan-in	$4g_0$	$2g_1$
Total:	$4g_0 + 8g_1$	
Previous:	$8g_0$	

Hierarchical partitioning



If $g_0 < 2g_1$, greedy hierarchical partitioning is suboptimal.

	Upper level	Lower level
Fan-out	0	$6g_1$
Fan-in	$4g_0$	$2g_1$
Total:	$4g_0 + 8g_1$	
Previous:	$8g_0$	

Multi-BSP aware partitioning

Slightly modified V-cycle:

1. coarsen
2. recurse or randomly partition
3. do k steps of HKLFM
   - calculate gains taking $g_0, \ldots, g_{L-1}$ into account
4. refine

Multi-BSP aware partitioning

Slightly modified V-cycle:

1. coarsen
2. recurse or randomly partition
3. do k steps of HKLFM
   - calculate gains taking $g_0, \ldots, g_{L-1}$ into account
4. refine

Claim: *if $g_0 > g_1 > g_2 \ldots$, then HKLFM is a local operation.*

By enumeration of all possibilities ($L = 2$). At level-1 refinement:

- suppose we move a nonzero a_{ij} from A_{s_1,s_2} to A_{t_1,t_2} with $s_1 \neq t_1$:
  - $a_{ij} \in \tilde{A}_{s_1,s_2}$, $a_{ij} \notin \tilde{A}_{s_1}$: gain is $g_1 - g_0$ or $2(g_1 - g_0)$.
  - $a_{ij} \notin \tilde{A}_{s_1,s_2}$: gain is 0, $g_1 - g_0$, or $2(g_1 - g_0)$.

Hence it suffices to perform HKLFM steps on each level separately.

Summary

Differences from flat BSP:

- **different notion of load balance**
  - parts must fit into local memory.
- **non-uniform communication costs**
  - implies different partitioning techniques.

Non-uniform data locality...

Summary

Differences from flat BSP:

- **different notion of load balance**
  - parts must fit into local memory.
- **non-uniform communication costs**
  - implies different partitioning techniques.

Non-uniform data locality...

with **fine-grained** distribution.

How does it compare?

- ANSI C++11, parallelisation using std::thread,
- implementation relies on shared-memory cache coherency
- Mondriaan 4.0, medium-grain, symmetric doubly BBD reordering
- Global arrays without blocking, nonzero reordering, compression.

	matrix	original	$p = 1$	$p = \max$	Optimal
2x8	G3_circuit	33.3	26.7	10.5	2.77
2x8	FS1	83.5	65.3	22.0	10.3
2x8	cage15	523	387	77.1	29.8
2x10	G3_circuit	22.7	16.9	9.77	1.73
2x10	FS1	83.5	65.3	22.0	7.56
2x10	cage15	341	233	54.7	23.4

all numbers are in ms.

- Y. and Bisseling, Cache-oblivious sparse matrix–vector multiplication, SISC 2009
- Y. and Roose, High-level strategies for sparse matrix–vector multiplication, IEEE TPDS 2014

Conclusions and Outlook

Conclusions:

- not (yet) competitive on shared-memory
- programmability, usability?
  - do we need to program for explicit hierarchies? (No!)
  - is recursive SPMD general enough?
  - Generic API, portability
  - **interoperability**: call from MPI, BSP, Spark, ...

Future work:

- incorporate vector distribution
- distributed-memory, and shared memory without cache coherency:
  - requires explicit Multi-BSP programming
- extension to sparse matrix powers

Thank you!

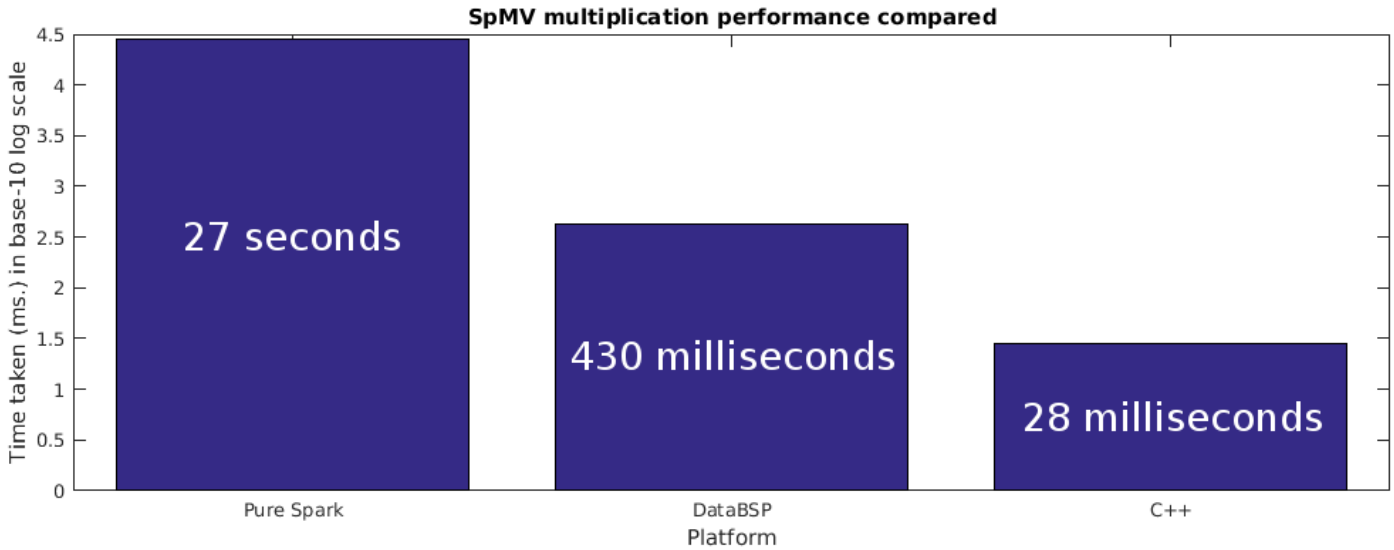
Backup Slides

Interoperable BSP

We have a shared-memory prototype. Preliminary results:

- SpMM multiply, SpMV multiply, and basic vector operations;
- one machine learning application.

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



Note: this is ongoing work. Performance will be improved, and functionality will be extended.

Interoperable BSP

Using an unified BSP guarantees interoperability. Going further:

- Call BSP algorithms from MPI;
- call BSP algorithms from MapReduce/Hadoop;
- call BSP algorithms from Spark;
- ...

Data I/O is a challenge. One example approach:

```
scala> val output_rdd = rdd.map( BSP_algorithm );
Hello from BSP, process number 0
Hello from BSP, process number 1
...
Hello from BSP, process number 11
scala>
```

Is this the best way to bridge HPC and Big Data?

Multi-BSP broadcast example

```
do {
    if ( val != NULL )
        bsp_put( val into process 0 );
    bsp_sync()
} while( bsp_up() );

do {
    if ( my process ID is not 0 )
        bsp_get( val from process 0 );
    bsp_sync();
} while( bsp_down() );
```

Automatically deploys over arbitrary hierarchies.

Results: cross platform

Cross platform results over 24 matrices:

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

no one solution fits all.

If we must, some generalising statements:

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

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