A foundational communication layer and a linear algebraic programming methodology

A. N. Yzelman
Computing Technology Lab
Huawei Technologies Zürich

30th of October, 2019
Contents

Lightweight Parallel Foundations
   Communication
   Performance model
   Execution, interoperability, and more

GraphBLAS
   How to achieve high performance: past lessons learned
   Lessons learned applied to GraphBLAS

Most work done while at the Paris R&D centre.
Huawei

Huawei is a large company with expanding business areas.

- around 180,000 employees worldwide
  - around 70,000 engage in R&D
  - 23 research centres in Europe alone
- 100+ billion USD yearly revenue
  - 25–30 percent year-over-year revenue increases

Computing Technologies Lab in Zürich (est. late 2019):

- long-term fundamental research
- hardware, architecture, software, tools, theory, and models:
  - microarchitectures, systems, interconnects, storage, memory, ...
  - compilers, HW/SW co-design, HW design tools, SW tools, ...
  - model-driven design, lower bounds, programming models, ...
- Unique (in Huawei)
  - novel lab to foster open research.
Lightweight Parallel Foundations

Communication
Performance model
Execution, interoperability, and more

GraphBLAS
How to achieve high performance: past lessons learned
Lessons learned applied to GraphBLAS
Lightweight Parallel Foundations

LPF is a communications layer alike BSPlib.

What is LPF *not* about:

► politics
► making parallel programming easier
► replacing BSPlib, MPI, MapReduce, Spark, . . .

What it is about:

► 'close to the metal' performance
► guaranteed performance
► enhanced robustness
► formal semantics
► interoperability

Joint work with Wijnand Suijlen (Huawei Technologies France).

Lightweight Parallel Foundations

LPF is a communications layer alike BSPlib.

What is LPF not about:
- politics
- making parallel programming easier
- replacing BSPlib, MPI, MapReduce, Spark, ...

What it is about:
- ‘close to the metal’ performance
- guaranteed performance
- enhanced robustness
- formal semantics
- interoperability

Joint work with Wijnand Suijlen (Huawei Technologies France).
Application Programming Interface

Communication:

- `lpf_get, lpf_put`: one-sided, non-buffered, non-blocking
- `lpf_sync`: block until local comms (in & out) completed
- `lpf_msg_attr_t, lpf_sync_attr_t`: allows for extensions

Alike `bsp_{hpget,hpput,sync}`

No other comms primitives!

Alike BSPlib, require memory registration:

- `lpf_memslot_t`: LPF registers slots instead of addresses
- `lpf_register_{local,global}`: local vs. remote reference
- `lpf_deregister`: takes both local & global slots

Changes are valid immediately.
Overlapping writes

Suppose $p > 1$ processes all execute this BSPlib code:

- `bsp_push_reg( &l_err, sizeof(l_err) );`
- `bsp_push_reg( &g_err, sizeof(g_err) );`
- `bsp_sync();`
- `bsp_hpput( 0, &l_err, 0, &g_err, 0, sizeof(l_err) );`
- `bsp_sync();`

Then `g_err` at process 0 becomes undefined.

LPF has clear semantics for such cases:

- For each memory region subject to overlapping writes
- Consider all incoming communication requests
- LPF must guarantee the result is a serialisation of $M$

Overlapping reads and writes remain undefined(!)
Overlapping writes

Suppose $p > 1$ processes all execute this BSPlib code:

- `bsp_push_reg( &l_err, sizeof(l_err) );`
- `bsp_push_reg( &g_err, sizeof(g_err) );`
- `bsp_sync();`
- `...`
- `bsp_hpput( 0, &l_err, 0, &g_err, 0, sizeof(l_err) );`
- `bsp_sync();`

Then `g_err` at process 0 becomes undefined.
Overlapping writes

Suppose $p > 1$ processes all execute this BSPlib code:

- `bsp_push_reg( &l_err, sizeof(l_err) );`
- `bsp_push_reg( &g_err, sizeof(g_err) );`
- `bsp_sync();`
- `...`
- `bsp_hpput( 0, &l_err, 0, &g_err, 0, sizeof(l_err) );`
- `bsp_sync();`

Then `g_err` at process 0 becomes undefined.

LPF has clear semantics for such cases:

- for each memory region subject to overlapping writes
- consider all incoming communication requests $M$
- **LPF must guarantee the result is a serialisation of $M$**
Overlapping writes

Suppose \( p > 1 \) processes all execute this BSPlib code:

\[
\begin{align*}
&\text{bsp\_push\_reg( } & \&l\_err, \text{ sizeof}(l\_err) \text{ );} \\
&\text{bsp\_push\_reg( } & \&g\_err, \text{ sizeof}(g\_err) \text{ );} \\
&\text{bsp\_sync();} \\
&... \\
&\text{bsp\_hpput( } 0, \&l\_err, 0, \&g\_err, 0, \text{ sizeof}(l\_err) \text{ );} \\
&\text{bsp\_sync();}
\end{align*}
\]

Then \texttt{g\_err} at process 0 becomes undefined.

LPF has clear semantics for such cases:

\[
\begin{align*}
&\text{for each memory region subject to overlapping writes} \\
&\text{consider all incoming communication requests } M \\
&\textbf{LPF must guarantee the result is a serialisation of } M \\
&\text{Overlapping reads and writes remain undefined(!)}
\end{align*}
\]
Communication example

Suppose $p > 1$ processes, $k$ of which have a local error $l_{err}$. 
Communication example

Suppose \( p > 1 \) processes, \( k \) of which have a local error \( l_{\text{err}} \).

Each process executes:

- \texttt{lpf\_err\_t l\_err, g\_err; bsp\_memslot\_t s\_lerr, s\_gerr;}
- \texttt{lpf\_register\_local( ctx, &l\_err, sizeof(l\_err), &s\_lerr );}
- \texttt{lpf\_register\_global( ctx, &g\_err, sizeof(g\_err), &s\_gerr );}
- ... 
- \texttt{if l\_err != LPF\_SUCCESS then}
  \texttt{lpf\_put( ctx,}
  \texttt{ s\_lerr, 0,}
  \texttt{ 0, s\_gerr, offset,}
  \texttt{ sizeof(l\_err), LPF\_MSG\_DEFAULT );}
- \texttt{lpf\_sync( ctx, LPF\_SYNC\_DEFAULT );}
Communication example

Suppose $p > 1$ processes, $k$ of which have a local error $l_{err}$.

Each process executes:

- $lpf_{err\_t} l_{err}, g_{err}; bsp\_memslot\_t s_{lerr}, s_{gerr};$
- $lpf\_register\_local( ctx, &l_{err}, sizeof(l_{err}), &s_{lerr} );$
- $lpf\_register\_global( ctx, &g_{err}, sizeof(g_{err}), &s_{gerr} );$
- ...
- **if** $l_{err} \neq LPF\_SUCCESS** then**
  $lpf\_put( ctx,$
  $s_{lerr}, 0,$
  $0, s_{gerr}, offset,$
  $sizeof(l_{err}), LPF\_MSG\_DEFAULT );$
- $lpf\_sync( ctx, LPF\_SYNC\_DEFAULT );$

By conflict resolution, $g_{err}$ at PID 0 will be a valid error code.
Performance model

Communication is not free.

Bulk Synchronous Parallel (BSP, Valiant 1990):

- A current superstep $i$ is ended by an $lpf$ sync

- Let $t_s^i$ the number of bytes transmitted by PID $s$ at step $i$

- Let $r_s^i$ the number of bytes received by PID $s$ at step $i$

- Each call to $lpf$ put at process $s$ to $k$ increases $t_s^i$ and $r_k^i$

- Each call to $lpf$ get at process $s$ from $k$ increases $r_s^i$ and $t_k^i$

- The subsequent $lpf$ sync takes at most $h^i + l$ time,

- with $h^i = \max_s \{t_s^i, r_s^i\}$, the superstep's $h$-relation and

- with $g$, $l$ machine-specific parameters.

- Majority of effort goes into ensuring compliance to model.

Some questions:

- How does an algorithm know the value for $g$ and $l$?

- Can we guarantee anything about the other LPF primitives?
Performance model

Communication is not free.

Bulk Synchronous Parallel (BSP, Valiant 1990):
- A current superstep $i$ is ended by an lpf_sync
- Let $t_i^s$ the number of bytes transmitted by PID $s$ at step $i$
- Let $r_i^s$ the number of bytes received by PID $s$ at step $i$
Performance model

Communication is not free.

Bulk Synchronous Parallel (BSP, Valiant 1990):

- A current superstep $i$ is ended by an $lpf_{\text{sync}}$
- Let $t_i^s$ the number of bytes transmitted by PID $s$ at step $i$
- Let $r_i^s$ the number of bytes received by PID $s$ at step $i$
- Each call to $lpf_{\text{put}}$ at process $s$ to $k$ increases $t_i^s$ and $r_i^k$
- Each call to $lpf_{\text{get}}$ at process $s$ from $k$ increases $r_i^s$ and $t_i^k$
Performance model

Communication is not free.

Bulk Synchronous Parallel (BSP, Valiant 1990):

- A current superstep \( i \) is ended by an \texttt{lpf\_sync}
- Let \( t^s_i \) the number of bytes transmitted by PID \( s \) at step \( i \)
- Let \( r^s_i \) the number of bytes received by PID \( s \) at step \( i \)
- Each call to \texttt{lpf\_put} at process \( s \) to \( k \) increases \( t^s_i \) and \( r^k_i \)
- Each call to \texttt{lpf\_get} at process \( s \) from \( k \) increases \( r^s_i \) and \( t^k_i \)
- The subsequent \texttt{lpf\_sync} takes at most \( h_i g + l \) time,
  - with \( h_i = \max_s \max\{t^s_i, r^s_i\} \), the superstep’s \( h \)-relation and
  - with \( g, l \) machine-specific parameters.
- Majority of effort goes into ensuring \textbf{compliance} to model.
Performance model

Communication is not free.

Bulk Synchronous Parallel (BSP, Valiant 1990):

- A current superstep $i$ is ended by an lpf_sync
- Let $t_i^s$ the number of bytes transmitted by PID $s$ at step $i$
- Let $r_i^s$ the number of bytes received by PID $s$ at step $i$
- Each call to lpf_put at process $s$ to $k$ increases $t_i^s$ and $r_i^k$
- Each call to lpf_get at process $s$ from $k$ increases $r_i^s$ and $t_i^k$
- The subsequent lpf_sync takes at most $h_i g + l$ time,
  - with $h_i = \max_s \max\{t_i^s, r_i^s\}$, the superstep’s $h$-relation and
  - with $g, l$ machine-specific parameters.
- Majority of effort goes into ensuring compliance to model.

Some questions:

- How does an algorithm know the value for $g$ and $l$?
- Can we guarantee anything about the other LPF primitives?
Immortal algorithms: reduction

Every process has a number $\alpha$ to be reduced at process 0:

- direct all-to-one: $(p - 1)(g + 1) + l$ flops.
- binary tree: $\lceil \log_2 p \rceil (g + l + 1)$ flops.
Immortal algorithms: reduction

Every process has a number $\alpha$ to be reduced at process 0:

- **direct all-to-one**: $(p - 1)(g + 1) + l$ flops.
- **binary tree**: $\lceil \log_2 p \rceil (g + l + 1)$ flops.
- **optimal**: $\min_{b \in \{2, 3, \ldots, p\}} \lceil \log_b p \rceil ((b - 1)(g + 1) + l)$ flops.

Examples:

- $p = 100, g = 250, l = 10000$, cost in flops:
  - $b = 2$ (binary tree), $71757$ flops,
  - $b = 10$, $24518$ flops,
  - $b = 100$ (direct), $34849$ flops.

- $p = 100, g = 25, l = 10000$, direct is optimal at $12574$ flops.

- $p = 10000, g = 250, l = 100000$, $b = 100$ is optimal.
**Immortal algorithms: reduction**

Every process has a number $\alpha$ to be reduced at process 0:

- direct all-to-one: $(p - 1)(g + 1) + l$ flops.
- binary tree: $\lceil \log_2 p \rceil (g + l + 1)$ flops.
- **optimal**: $\min_{b \in \{2, 3, \ldots, p\}} \lceil \log_b p \rceil ((b - 1)(g + 1) + l)$ flops.

I.e., optimal algorithm depends on $p$, $g$, and $l$. Examples:

- $p = 100$, $g = 250$, $l = 10\,000$, cost in flops:
  - 71 757 for $b = 2$ (binary tree),
  - **24 518** for $b = 10$,
  - 34 849 for $b = 100$ (direct).

Rationale: an immortal algorithm requires run-time introspection of the performance model's parameters.
Every process has a number $\alpha$ to be reduced at process 0:

- direct all-to-one: $(p - 1)(g + 1) + l$ flops.
- binary tree: $\lceil \log_2 p \rceil (g + l + 1)$ flops.
- optimal: $\min_{b \in \{2, 3, \ldots, p\}} \lceil \log_b p \rceil ((b - 1)(g + 1) + l)$ flops.

I.e., optimal algorithm depends on $p$, $g$, and $l$. Examples:

- $p = 100$, $g = 250$, $l = 10\,000$, cost in flops:
  - 71 757 for $b = 2$ (binary tree),
  - **24 518** for $b = 10$,
  - 34 849 for $b = 100$ (direct).
- $p = 100$, $g = 25$, $l = 10\,000$, direct is optimal at 12 574 flops.
Immortal algorithms: reduction

Every process has a number $\alpha$ to be reduced at process 0:

- **direct all-to-one**: $(p - 1)(g + 1) + l$ flops.
- **binary tree**: $\lceil \log_2 p \rceil (g + l + 1)$ flops.
- **optimal**: $\min_{b \in \{2, 3, \ldots, p\}} \lceil \log_b p \rceil ((b - 1)(g + 1) + l)$ flops.

I.e., optimal algorithm depends on $p$, $g$, and $l$. Examples:

- $p = 100$, $g = 250$, $l = 10\,000$, cost in flops:
  - 71 757 for $b = 2$ (binary tree),
  - **24 518** for $b = 10$,
  - 34 849 for $b = 100$ (direct).

- $p = 100$, $g = 25$, $l = 10\,000$, direct is optimal at 12 574 flops.
- $p = 10\,000$, $g = 250$, $l = 100\,000$, $b = 100$ is optimal.
Immortal algorithms: reduction

Every process has a number $\alpha$ to be reduced at process 0:

- **direct all-to-one:** $(p - 1)(g + 1) + l$ flops.
- **binary tree:** $\lceil \log_2 p \rceil (g + l + 1)$ flops.
- **optimal:** $\min_{b \in \{2, 3, \ldots, p\}} \lceil \log_b p \rceil ((b - 1)(g + 1) + l)$ flops.

I.e., optimal algorithm depends on $p$, $g$, and $l$. Examples:

- $p = 100$, $g = 250$, $l = 10000$, cost in flops:
  - 71 757 for $b = 2$ (binary tree),
  - **24 518** for $b = 10$,
  - 34 849 for $b = 100$ (direct).

- $p = 100$, $g = 25$, $l = 10000$, direct is optimal at 12 574 flops.
- $p = 10000$, $g = 250$, $l = 100000$, $b = 100$ is optimal.

**Rationale:** an immortal algorithm requires run-time introspection of the performance model’s parameters.
lpf_probe returns the number of parallel processes the machine supports, how many are currently free, as well as two functions:

- double g( bsp_pid_t p, size_t wordsize, bsp_sync_attr_t );
- double l( bsp_pid_t p, size_t wordsize, bsp_sync_attr_t );

<table>
<thead>
<tr>
<th></th>
<th>w = 8</th>
<th>w = 64</th>
<th>w = 1024</th>
<th>w = 1048576</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy-8 r (ns/byte)</td>
<td>1.18</td>
<td>0.874</td>
<td>0.864</td>
<td>0.777</td>
</tr>
<tr>
<td>Hybrid-RB g (×)</td>
<td>332 ±0.39</td>
<td>82.8 ±0.15</td>
<td>22.4 ±0.23</td>
<td>6.83 ±0.14</td>
</tr>
<tr>
<td>128 procς l (words)</td>
<td>5877 ±351</td>
<td>725 ±3.9</td>
<td>54 ±0.28</td>
<td>0.06 ±0.0005</td>
</tr>
<tr>
<td>Ivy-6 r (ns/byte)</td>
<td>0.806</td>
<td>0.730</td>
<td>0.719</td>
<td>0.653</td>
</tr>
<tr>
<td>Hybrid-RB g (×)</td>
<td>303 ±0.11</td>
<td>80.8 ±0.046</td>
<td>13.5 ±0.056</td>
<td>2.75 ±0.01</td>
</tr>
<tr>
<td>120 procς l (words)</td>
<td>7717 ±178</td>
<td>706 ±5.2</td>
<td>179 ±31</td>
<td>0.06 ±0.0003</td>
</tr>
<tr>
<td>BigIvy r (ns/byte)</td>
<td>0.844</td>
<td>0.806</td>
<td>0.769</td>
<td>0.825</td>
</tr>
<tr>
<td>Pthreads g (×)</td>
<td>51.9 ±0.26</td>
<td>10.7 ±0.060</td>
<td>5.63 ±0.041</td>
<td>5.43 ±0.52</td>
</tr>
<tr>
<td>120 procς l (words)</td>
<td>6231 ±74</td>
<td>1086 ±11</td>
<td>100 ±0.93</td>
<td>4.3 ±3.2</td>
</tr>
</tbody>
</table>

**Table 3.** The system constants $g, l$ normalised w.r.t. $r$, the speed of a memcpy. The unit of communication is $w$ bytes. The ± indicate the size of a 95% confidence interval.
Other performance guarantees

Time spent in communication (lpf_sync) defined by:
▶ algorithm’s puts/gets (source, destination, and size);
▶ machine parameters captured by $g$, $l$;
▶ BSP’s performance model.

Time spent in computation:
▶ user code;
▶ calls to LPF primitives(!).
Other performance guarantees

Time spent in communication (lpf\_sync) defined by:

▶ algorithm’s puts/gets (source, destination, and size);
▶ machine parameters captured by $g, l$;
▶ BSP’s performance model.

Time spent in computation:

▶ user code;
▶ calls to LPF primitives(!).

Some performance guarantees:

▶ lpf\_register\_{\{local,global\}}: $O(size)$
▶ lpf\_get, lpf\_put, lpf\_deregister: $\Theta(1)$
▶ lpf\_probe: $\Omega(1)$

**Rationale:** Asymptotic behaviour must be defined or process-local complexity analysis becomes impossible.
Memory guarantees & buffer control

Memory may be constrained:
▶ small manycore device, or
▶ core counts that grow faster than available memory.

Even taking $\Theta(p)$ memory per process may be too much.

LPF’s internal buffers are explicitly controlled at run-time:
▶ `lpf_resize_memory_register( lpf_t context, size_t max_regs )`
▶ `lpf_resize_message_queue( lpf_t context, size_t max_msgs )`

Compare these to calling to reserve ahead for a date.

Initial values (fresh contexts): 0 memslots, 0 messages(!)
▶ first calls in an LPF program are likely these two primitives.

LPF doesn’t forgive:
▶ registering without reservation: undefined behaviour (UB).
▶ sending (or receiving!) a message without reservation: UB.
Some performance results

We have a POSIX Threads LPF implementation:

HPBSP FFT compared to FFTW3 and Intel MKL

- 8-socket shared-memory Intel Xeon E7-8890 v2
  - Local memory: 2.8 Gbyte/s/core.
- Immortal FFT algorithm: Valiant ('90), Inda & Bisseling ('01).
Some performance results

Ibverbs LPF implementation + PThreads LPF = Hybrid LPF:

HPBSP FFT compared to FFTW3 and Intel MKL

- 8 node times 2 socket Intel Xeon E5-2650
  - Local memory: 4.3 Gbyte/s/core, FDR IB: 0.4 Gbyte/s/core.
- Immortal FFT algorithm: Valiant ('90), Inda & Bisseling ('01).
Execution types

An LPF SPMD program has the following signature:

- void (*f)(lpf_t context, lpf_pid_t s, lpf_pid_t P, lpf_args_t args);

An LPF SPMD program is started through one of:

- lpf_exec( lpf_t context, bsp_pid_t P, lpf_spmd_t program, lpf_args_t args );
- lpf_hook( lpf_init_t init, lpf_spmd_t program, lpf_args_t args );
- lpf_rehook( lpf_t ctx, lpf_spmd_t program, lpf_args_t args );

These primitives block until the LPF program completes.

lpf_args_t consists of six fields:

- const void * input, size_t input_size,
- void * output, size_t output_size,
- const lpf_func_t f_symbols, size_t f_size.
Hello World

```c
void hello_world( lpf_t ctx, lpf_pid_t s, lpf_pid_t P, lpf_args_t args )
{
    (void) ctx; (void) args;
    (void) printf( "Hello world from PID %d / %d\n", s, P );
}
```
Hello World

```c
void hello_world( lpf_t ctx, lpf_pid_t s, lpf_pid_t P, lpf_args_t args )
{
    (void) ctx; (void) args;
    (void) printf( "Hello world from PID %d / %d\n", s, P );
}

int main( int argc, char ** argv )
{
    const lpf_err_t err = lpf_exec( LPF_ROOT, LPF_MAX_P, &hello_world, LPF_NO_ARGS );
    return err == LPF_SUCCESS ? 0 : 255;
}
```
Encapsulation

Libraries that implement algorithms using LPF should

- define an lpf_spmd_t entry function, and
- perform I/O using lpf_args_t.

The library should be aware of exec vs. hook:

- in exec, should distribute input or read input in parallel
- in (re)hook, transfer input 1:1 between host and slave

Example:

- double * x, * y; size_t n;
- ... // initialise x, y, n and retrieve or compute x
- lpf_args_t fft_args; fft_args.in = x; fft_args.out = y;
- fft_args.in_size = fft_args.out_size = 2 * n * sizeof(double);
- lpf_rehook( context, &lpf_fft, fft_args );
- ... //use the FFT of x, now stored in y
Interoperability

Past decade, a wildgrowth of parallel programming frameworks:

▶ Big Data: MapReduce, Spark, Giraph,

▶ HPC: MPI, OpenMP, BSPlib, PThreads, Cilk, TBB, Legion, OCR, Chapel, Charm++, PaRSec, OmpSs, StarPU, HPX,

LPF aims not to make all these obsolete. Big data’s success due to programmer efficiency, not performance efficiency. LPF instead enables interoperability: immortal algorithms should transparently integrate with any parallel framework, thus be enabled for use as widely as possible.

Rationale: users should use tools best suited for the job at hand.
Interoperability

Past decade, a wildgrowth of parallel programming frameworks:

- Big Data: MapReduce, Spark, Giraph, Flink, Ligra, PowerGraph,
- HPC: MPI, OpenMP, BSPlib, PThreads, Cilk, TBB, Legion, OCR, Chapel, Charm++, PaRSec, OmpSs, StarPU, HPX,

LPF aims not to make all these obsolete. Big data's success due to programmer efficiency, not due to performance efficiency. LPF instead enables interoperability: immortal algorithms should transparently integrate with any parallel framework, thus be enabled for use as widely as possible.

Rationale: users should use tools best suited for the job at hand.
Interoperability

Past decade, a wildgrowth of parallel programming frameworks:

- Big Data: MapReduce, Spark, Giraph, Flink, Ligra, PowerGraph, Storm, Hive, Presto, Samza, Heron, Kudu, ...
- HPC: MPI, OpenMP, BSPlib, PThreads, Cilk,
Interoperability

Past decade, a wildgrowth of parallel programming frameworks:

- **Big Data:** MapReduce, Spark, Giraph, Flink, Ligra, PowerGraph, Storm, Hive, Presto, Samza, Heron, Kudu, ...
- **HPC:** MPI, OpenMP, BSPlib, PThreads, Cilk, TBB, Legion, OCR, Chapel, Charm++, PaRSec, OmpSs, StarPU, HPX, ...

LPF aims **not** to make all these obsolete

Big data’s success due to **programmer efficiency**.

- not due to performance efficiency.

LPF instead enables **interoperability**: immortal algorithms should

- transparently integrate with *any* parallel framework, thus
- be enabled for use as widely as possible.

**Rationale:** users should use tools best suited for the job at hand.
Interoperability

The lpf_hook is like lpf_rehook

- but from arbitrary parallel contexts, not only LPF contexts;
- requires a valid lpf_init_t instance;
- creating an lpf_init_t is implementation-defined.

In our LPF implementation, can retrieve one using TCP/IP:

- char * hostname = NULL, * portname = NULL;
- lpf_pid_t process_id = 0, nprocs = 0;
- lpf_args_t args = LPF_NO_ARGS;
- ... // user code decides the above variables
- lpf_init_t init = LPF_INIT_NONE;
- lpf_mpi_initialize_over_tcp( hostname, portname, 30 000, process_id, nprocs, &init );
- lpf_hook( init, &spmd, args );
- ... //user code continues making use of args.out
Interoperability

A bridge between Big Data and HPC (Y., PMAA ’16):

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

cage15, \( n = 5154859 \), \( nz = 99199551 \)
Robustness

LPF allows for the following return codes (lpf_err_t):

- LPF_SUCCESS
- LPF_OUT_OF_MEMORY
- LPF_ERR_FATAL

All errors except LPF_ERR_FATAL:

- must have no side-effects
- must be mitigable

On encountering LPF_ERR_FATAL:

- any subsequent call to any LPF primitive results in UB;
- user code can only clean up and return.

All errors are local;

- no globally consistent error codes, even for lpf_sync(!)
Formal semantics

LPF’s 12 primitives have their semantics formally defined.

Enables use of formal methods in parallel software. E.g.,
- verification of (correct) use of LPF primitives
- automatic cost analysis

Combined with sequential verification tools:
- verification of user programs
- verified parallel code generation

State of an LPF machine is a vector of sequential states;
- only the lpf_sync induces global change,
- calling any other primitive incurs local changes only.

BSPlib: Tesson & Loulergue, 2007; Gava & Fortin, 2008; Jokabsson et al., 2017; Jokabsson, 2018. For LPF:
- modified to work with unbuffered DRMA
Extensibility

Functional and performance semantics extended through attributes:

▶ lpf_msg_attr_t: an attribute attached to a put or get
▶ lpf_sync_attr_t: an attribute attached to a superstep

Example of a change in functional semantics:
▶ Message attribute changing a put into an accumulate

Example of a change in performance semantics:
▶ optimisations for pre-defined communication patterns
▶ zero-cost synchronisation (Alpert & Philbin, ’97)

Example of a change in both:
▶ message attribute introducing staleness (Xing et al., ’15)
Summary

LPF is a communication layer for immortal algorithms:
1. allows implementing portable immortal algorithms
2. allows interoperable use of immortal algorithms

LPF is robust:
- formal specification of primitives
- error codes and mitigation

LPF supports easier-to-use, higher-level libraries:
- Encapsulation (lpf_rehook) aids library design (FFT, e.g.)
- Bulk-Synchronous Message Passing (BSMP, send/move)
- Collectives library (Suijlen, 2019)
- BSPlib interface

LPF is extensible.
GraphBLAS

Lightweight Parallel Foundations
  Communication
  Performance model
  Execution, interoperability, and more

GraphBLAS
  How to achieve high performance: past lessons learned
  Lessons learned applied to GraphBLAS
Motivation

What is GraphBLAS *not* about:

- politics
- replacing BSPlib, MPI, MapReduce, Spark, ...
Motivation

What is GraphBLAS *not* about:
- politics
- replacing BSPlib, MPI, MapReduce, Spark, ...

What it is about:
- making parallel programming easier
- close-to-the-metal performance
- guaranteed performance
- interoperability

Community:
http://www.graphblas.org

Note that the standard API is C11, ours is C++11.
- joint work with Jonathan M. Nash & Daniel Di Nardo
History

APIs & Software:

- **Basic Linear Algebra Subroutines**, Lawson et al., 1979
- **Sparse BLAS**, Remington & Pozo, 1996
- **Combinatorial BLAS** (CombBLAS), Buluç & Gilbert, 2011
- **GraphBLAS**, GraphBLAS.org, 2015 onwards
  - GraphBLAS Template Library (GBTL), C++, CPU + GPU
    McMillan et al., CMU and others, 2015 onwards;
  - **This work**, C++11, CPU + cluster + mobile
    Y. et al., Huawei Paris, 2016 onwards;
  - **SuiteSparse::GraphBLAS**, C11, CPU
    Davis, TAMU, 2018 onwards;
  - **GraphBLAST**, C11 (+Gunrock), GPU
    Yang et al., U. Illinois and others, 2019 onwards.
Conceptually, the idea of using math concepts in programming, or generalised linear algebraic concepts, seems somewhat recurrent:

- **Introduction to Algorithms** (first edition?), Cormen, Leiserson, Rivest (1990)
Core concepts

▶ scalars, $\alpha \in D$: standard C++ (POD) types
▶ vectors, $x \in D^n$, sparse and dense: `grb::Vector< D > x`;
▶ matrices, $A \in D^{m\times n}$, sparse: `grb::Matrix< D > A`;
▶ In the above templates, D may be void for pattern-only data.
Core concepts

- Scalars, $\alpha \in D$: standard C++ (POD) types
- Vectors, $\mathbf{x} \in D^n$, sparse and dense: `grb::Vector<D> x;`
- Matrices, $A \in D^{m \times n}$, sparse: `grb::Matrix<D> A;`
- In the above templates, D may be void for pattern-only data.

- Unary operators, $f : D_1 \rightarrow D_2$
- Binary operators, $g : D_1 \times D_2 \rightarrow D_3$
- Monoid, $< D_1, D_2, D_3, \oplus, 0 >$
  - $\oplus : D_1 \times D_2 \rightarrow D_3$
  - $\forall a \in D_1, b \in D_2 : \oplus(a, 0) = a, \oplus(0, b) = b$
  - $\oplus$ must be associative

- Semiring, $< D_1, D_2, D_3, D_4, \oplus, \otimes, 0, 1 >$
  - $\otimes : D_1 \times D_2 \rightarrow D_3$, forms a monoid with 1
  - $\oplus : D_3 \times D_4 \rightarrow D_4$, forms a commutative monoid with 0
  - Left-distributive: $\otimes(m, \oplus(a, b)) = \oplus(\otimes(m, a), \otimes(m, b))$
  - Right-distributive: $\otimes(\oplus(a, b), m) = \oplus(\otimes(a, m), \otimes(b, m))$
  - $\forall a \in D_1, b \in D_2 : \otimes(a, 0) = 0, \otimes(0, b) = 0$
GraphBLAS

Graph algorithms in the language of linear algebra:

▶ a graph \((V, E)\) is a sparse matrix \(A \in D_E^{|V| \times |V|}\)
▶ an edge is a nonzero \((i, j)\) coordinate in \(A\)
▶ an edge weight is a nonzero \(a_{ij} \in A\)
GraphBLAS

Graph algorithms in the language of linear algebra:

- a graph $(V, E)$ is a sparse matrix $A \in D_E^{\lvert V \rvert \times \lvert V \rvert}$
- an edge is a nonzero $(i, j)$ coordinate in $A$
- an edge weight is a nonzero $a_{ij} \in A$
- a vertex is a coordinate $i$, $0 \leq i < \lvert V \rvert$
- a vertex weight is a vector nonzero $x_i$, $x \in D_V^{\lvert V \rvert}$

Kepner & Gilbert: GA in the Language of LA
DOI: 10.1137/1.9780898719918 (SIAM, 2011)
GraphBLAS

Graph algorithms in the language of linear algebra:

- a graph \((V, E)\) is a sparse matrix \(A \in D_E^{\left|V\right| \times \left|V\right|}\)
- an edge is a nonzero \((i, j)\) coordinate in \(A\)
- an edge weight is a nonzero \(a_{ij} \in A\)
- a vertex is a coordinate \(i, 0 \leq i < \left|V\right|\)
- a vertex weight is a vector nonzero \(x_i, x \in D_V^{\left|V\right|}\)

Graph algos as (sparse) linear algebra, parametrised in semirings:

- different from C++ overloaded ‘+’, ‘*’ operators
  - same sparse container may be subject to different semirings, reinterpreting computation for statically typed \(A\) and \(x\):
    - \texttt{grb::mxv( y, A, x, ring1 );}
    - \texttt{grb::mxv( y, A, x, ring2 );}
GraphBLAS

Graph algorithms in the language of linear algebra:

- a graph \((V, E)\) is a sparse matrix \(A \in D_E^{V \times V}\)
- an edge is a nonzero \((i, j)\) coordinate in \(A\)
- an edge weight is a nonzero \(a_{ij} \in A\)
- a vertex is a coordinate \(i, 0 \leq i < |V|\)
- a vertex weight is a vector nonzero \(x_i, x \in D_V^{|V|}\)

Graph algs as (sparse) linear algebra, parametrised in semirings:

- different from C++ overloaded ‘+’, ‘*’ operators
  - same sparse container may be subject to different semirings, reinterpreting computation for statically typed \(A\) and \(x\):
    - \texttt{grb::mxv( y, A, x, ring1 );}
    - \texttt{grb::mxv( y, A, x, ring2 );}

Kepner & Gilbert: GA in the Language of LA
DOI: 10.1137/1.9780898719918 (SIAM, 2011)
Examples

(1) PageRank

▶ canonical example of a graph LA algo
▶ ‘regular’ LA \((\mathbb{R}, +, \ast, 0, 1)\): dot products, sparse matrix–dense vector multiplication, element-wise vectors ops, ...
▶ With \(L\) the link matrix, \(row\_sum\) its row sums, inner loop only, pseudo code and standard+GraphBLAS C++ primitives:

1: \textbf{for iter} = 0 to \textbf{max_iter} while \(r > tol\) do
2: \hspace{1em} \(\delta = (pr, row\_sum), \text{grb::dot}\)
3: \hspace{1em} \(pr\_next = pr \ast row\_sum, \text{grb::apply}\)
4: \hspace{1em} \(\delta = 1/n(\alpha \ast \delta + 1 - \alpha), \text{standard C++ scalar ops}\)
5: \hspace{1em} \(pr\_next = pr\_next L, \text{grb::vxm}\)
6: \hspace{1em} \(pr\_next = pr\_next + \delta, \text{grb::foldl}\)
7: \hspace{1em} \(r = \|pr - pr\_next\|_1, \text{grb::norm}\)
8: \hspace{1em} \(pr = pr\_next, \text{std::swap}\)
(2) Nearest-neighbours

▶ a canonical graph algorithm example
▶ modified semiring: \((\mathbb{N}_0, \text{min}, +, \infty, 0)\)
▶ starting from vertex \(d\):

\[
\begin{pmatrix}
0 \\
\end{pmatrix}
\begin{pmatrix}
\times & 7 & \textbf{5} \\
7 & \times & 8 & 9 & 7 \\
8 & \times & 5 \\
5 & 9 & \times & 15 & 6 \\
7 & 5 & 15 & \times & 8 & 9 \\
6 & 8 & \times & 11 \\
9 & 11 & \times \\
\end{pmatrix}
\begin{pmatrix}
0 \\
\end{pmatrix}
\]

\[
x^\text{in}_d = 0, \ a_{ad} = 5, \ x^\text{in}_a = \text{“zero”} = \infty, \ so \ x^\text{out}_a = \min\{0 + 5, \infty\}.
\]

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours

- a canonical graph algorithm example
- modified semiring: $(\mathbb{N}_0, \min, +, \infty, 0)$
- starting from vertex $d$:

\[
\begin{pmatrix} 5 \\ 0 \end{pmatrix} + \begin{pmatrix} x & 7 & 5 \\ 7 & 8 & 9 & 7 \\ 8 & 5 \\ 5 & 9 & 15 & 6 \\ 7 & 5 & 15 & 8 & 9 \\ 6 & 8 & 11 \\ 9 & 11 & \end{pmatrix} \begin{pmatrix} A \\ x \end{pmatrix} = \begin{pmatrix} 0 \end{pmatrix}
\]

Now move to the next row and see how vertex $b$ interacts with $d$...

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours

- a canonical graph algorithm example
- modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
- starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
0
\end{pmatrix}
+ \begin{pmatrix}
x \\
7 \\
7 \\
8 \\
5 \\
9 \\
7 \\
5 \\
9 \\
15 \\
8 \\
11
\end{pmatrix} \begin{pmatrix}
x \\
x + Ax
\end{pmatrix}
\]

\[w(d) = 0, \ w(\{a, b\}) = 9, \ x_b = \infty \ (\text{‘zero’}), \ \text{so} \ y_b = \min\{0 + 9, \infty\}.
\]

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours

▶ a canonical graph algorithm example
▶ modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
▶ starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
0
\end{pmatrix}
\begin{pmatrix}
x \\
7 \\
8 \\
0
\end{pmatrix}
\begin{pmatrix}
5 \\
9 \\
15 \\
6
\end{pmatrix}
\begin{pmatrix}
0
\end{pmatrix}
\]

\[
x \rightarrow \begin{pmatrix}
\times & 7 & 5 \\
\times & 8 & 9 \\
\times & 15 & 6
\end{pmatrix}
\begin{pmatrix}
7 \\
5 \\
9 \\
8 \\
6 \\
11
\end{pmatrix}
\begin{pmatrix}
x
\end{pmatrix}
\]

We’ll now finish the iteration in one go...

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours
▶ a canonical graph algorithm example
▶ modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
▶ starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6
\end{pmatrix} + \begin{pmatrix}
x & 7 & 5 \\
7 & x & 8 & 9 & 7 \\
8 & x & 5 \\
5 & 9 & x & 15 & 6 \\
7 & 5 & 15 & x & 8 & 9 \\
6 & 8 & x & 11 \\
9 & 11 & x & 0
\end{pmatrix} = \begin{pmatrix}
x \\
A \\
x
\end{pmatrix}
\]

\(x\) now are the shortest 1-hop distances from \(d\). Now on to \(A^2y\)...

Examples

(2) Nearest-neighbours

- a canonical graph algorithm example
- modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
- starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6
\end{pmatrix} +
\begin{pmatrix}
x \\
7 \\
8 \\
5 \\
5
\end{pmatrix}
\begin{pmatrix}
7 \\
8 \\
9 \\
5 \\
6
\end{pmatrix}
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6
\end{pmatrix}
\]

From \(a\) to \(b\): \(\min\{5 + 7, 9\} = 9\). From \(a\) to \(d\): \(\min\{5 + 5, 0\} = 0\).

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours

> a canonical graph algorithm example
> modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
> starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
17 \\
0 \\
15 \\
6
\end{pmatrix}
\begin{pmatrix}
x & 7 & 5 \\
7 & x & 9 & 7 \\
8 & x & 5 \\
5 & 9 & x & 15 & 6 \\
7 & 5 & 15 & x & 8 & 9 \\
6 & 8 & x & 11 \\
9 & 11 & x
\end{pmatrix}
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6
\end{pmatrix}
\]

\(x = Ax\)

From \(b\) to \(\{a, c, d, e\}\), followed by \(d\) to \(\{a, b, e, f\}\)

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours
- a canonical graph algorithm example
- modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
- starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
17 \\
0 \\
15 \\
6 \\
24 \\
\end{pmatrix}
\xrightarrow{x + = Ax}
\begin{pmatrix}
x 7 5 \\
7 x 8 9 7 \\
8 x 5 \\
5 9 x 15 6 \\
7 5 15 x 8 9 \\
6 8 x 11 \\
\end{pmatrix}
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6 \\
\end{pmatrix}
\]

From \(e\) to \(\{b, c, f, g\}\); now only \(f\) remains to do in this iteration

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours
▶ a canonical graph algorithm example
▶ modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
▶ starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
17 \\
0 \\
14 \\
6 \\
17
\end{pmatrix}
+ \begin{pmatrix}
x & 7 & 5 \\
7 & x & 9 & 7 \\
8 & x & 5 \\
5 & 9 & x & 15 & 6 \\
7 & 5 & 15 & x & 8 & 9 \\
6 & 8 & x & 11 \\
9 & 11 & x
\end{pmatrix}
\begin{pmatrix}
5 \\
9 \\
0 \\
15 \\
6 \\
x
\end{pmatrix}
\]

\[x = Ax\]

From \(f\) to \(\{e, g\}\). \(x\), the shortest distances from \(d\) within two hops.

graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Nearest-neighbours

- a canonical graph algorithm example
- modified semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\)
- starting from vertex \(d\):

\[
\begin{pmatrix}
5 \\
9 \\
17 \\
0 \\
14 \\
6 \\
17
\end{pmatrix} + \begin{pmatrix}
x \\
7 \\
7 \\
8 \\
5 \\
9 \\
6
\end{pmatrix} = \begin{pmatrix}
x \\
7 \\
8 \\
5 \\
9 \\
6 \\
9
\end{pmatrix}
\]

Stop condition: keep iterating until \(x\) does not change

Graph illustration in Tikz from http://www.texample.net/tikz/examples/prims-algorithm/
Examples

(2) Single-source shortests paths

- Modify semiring: \((\mathbb{N}_0, \min, +, \infty, 0)\) and start from vertex \(d\)
- In C++, with uint an unsigned int:

```cpp
1: grb::semiring< uint, grb::operators::min, grb::operators::add,
   grb::identities::infinity, grb::identities::zero > spR;
2: n = grb::nrows(A); grb::vector< bool > mask( n );
3: grb::vector< uint > x( n ), y( n );
4: grb::set( x, d, 0 ); grb::set( mask, x ); grb::set( y, x );
5: grb::Vector< bool > eWiseEq( n ); bool eq = false;
6: while !eq do
7:   grb::mxv< descriptors::invert_mask | descriptors::no_casting
      | grb::descriptors::in_place >( y, mask, A, x, spR );
8:   grb::apply( eWiseEq, x, y, operators::is_equal< uint >( ) );
9:   eq = true; std::swap( x, y );
10:  grb::foldl( eq, eWiseEq, operators::is_equal< bool >( ) );
11: return x
```
Examples

Gabor Szarnyas recently produced an overview of algos:
- Breadth-First Search, $\Theta(|E|)$
- Single-source Shortest Paths, Bellman-Ford, $\Theta(|V||E|)$
- All-pairs shortest paths, $\Theta(|V|^3)$
- Minimum spanning tree, Boruvka, $|E| \log |V|$
- Maximum flow, $\Theta(|V||E|^2)$

Some algorithms not at best known complexities:
- Dijkstra (SSSP), Prim (APSP), max. independent sets...

Some recent work:
- Linear Algebraic Depth-First Search, Spampinato et al., 2019
- DFS previously a canonical ‘counter-example’ to GraphBLAS
- Adds permutation-based stack semantics to GraphBLAS
Summary

GraphBLAS is:

- a way to express graph algorithms in linear algebra
- a C11 standard with two compliant implementations
  - Tim Davis’ SuiteSparse::GraphBLAS
  - IBM’s GPI with a C11 GraphBLAS wrapper (Moreira et al.)
- two native C++ implementations (CMU, Huawei)
- increasingly many algorithms: LAGraph
Summary

GraphBLAS is:

- a way to express graph algorithms in linear algebra
- a C11 standard with two compliant implementations
  - Tim Davis’ SuiteSparse::GraphBLAS
  - IBM’s GPI with a C11 GraphBLAS wrapper (Moreira et al.)
- two native C++ implementations (CMU, Huawei)
- increasingly many algorithms: LAGraph

Debate:

- LA is the ‘right way’ to look at graph algorithms
- one alternative: ‘think like a vertex’, a lot of adoption
Summary

GraphBLAS is:
- a way to express graph algorithms in linear algebra
- a C11 standard with two compliant implementations
  - Tim Davis’ SuiteSparse::GraphBLAS
  - IBM’s GPI with a C11 GraphBLAS wrapper (Moreira et al.)
- two native C++ implementations (CMU, Huawei)
- increasingly many algorithms: LAGraph

Debate:
- LA is the ‘right way’ to look at graph algorithms
- one alternative: ‘think like a vertex’, a lot of adoption

Less debatable:
- Sparse LA software techniques can speed up graph algos
Central obstacles for SpMV multiplication performance

**Shared-memory:**
- limited memory throughput,
- inefficient cache use, and
- non-uniform memory access (NUMA) issues.

**Distributed-memory:**
- inefficient network use.
Central obstacles for SpMV multiplication performance

**Shared-memory:**
- limited memory throughput,
- inefficient cache use, and
- non-uniform memory access (NUMA) issues.

**Distributed-memory:**
- inefficient network use.

Initial analysis of $Ax = b$ (SpMV) by **roofline**:
- read input matrix once
- two flops per nonzero

Arithmetic intensity is very low.
Central obstacles for SpMV multiplication performance

**Shared-memory:**
- limited memory throughput,
- inefficient cache use, and
- non-uniform memory access (NUMA) issues.

**Distributed-memory:**
- inefficient network use.

Initial analysis of $Ax = b$ (SpMV) by **roofline**:
- read input matrix once
- two flops per nonzero

Arithmetic intensity is very low:
- minimise footprint of $A$
- vectorisation shouldn’t help

(Image taken from da Silva et al., DOI 10.1155/2013/428078, Creative Commons Attribution License)
Bandwidth

Compression leads to better performance:

- Coordinate format storage (COO):

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

for \( k = 0 \) to \( nz - 1 \)

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

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

\( \Theta(3nz) \) storage.
**Bandwidth**

Compression leads to better performance:

- **Compressed Row Storage (CRS):**

  \[
  i = (0, 0, 1, 1, 2, 2, 2, 3) \\
  i_{\text{start}} = (0, 2, 4, 7, 8) \\
  j = (0, 4, 2, 4, 1, 3, 5, 2) \\
  v = (a_{00}, a_{04}, \ldots, a_{32})
  \]

  \[
  \text{for } i = 0 \text{ to } m - 1 \\
  \text{for } k = i_{\text{start}i} \text{ to } i_{\text{start}i+1} - 1 \\
  y_i := y_i + v_k \cdot x_{jk}
  \]

  The CRS format:

  From $\Theta(3nz)$ storage to $\Theta(2nz + m + 1)$.
Bandwidth

Compression leads to better performance:

- **Compressed Row Storage (CRS):**

  \[
  i = (0, 0, 1, 1, 2, 2, 2, 3) \\
  i_{\text{start}} = (0, 2, 4, 7, 8) \\
  j = (0, 4, 2, 4, 1, 3, 5, 2) \\
  v = (a_{00}, a_{04}, \ldots, a_{32})
  \]

  \[
  \text{for } i = 0 \text{ to } m - 1 \\
  \text{for } k = i_{\text{start}} \text{ to } i_{\text{start}} + 1 - 1 \\
  y_i := y_i + v_k \cdot x_{jk}
  \]

  The CRS format:

  From $\Theta(3nz)$ storage to $\Theta(2nz + m + 1)$.

  Can do the same column-wise, leading to CCS.
Inefficient cache use

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

Accesses on the input vector are completely unpredictable.
Inefficient cache use

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

Two orthogonal solution classes:
- reordering matrix rows and/or columns
- reordering matrix nonzeroes (not compatible with CRS!)
Matrix permutations

Goes back to the 70s, linked to cache reuse for SpMV by the 90s:

- Das et al. (1994). The design and implementation of a parallel unstructured Euler solver;
- Sivan Toledo. (1997). Improving the memory-system performance of sparse-matrix vector multiplication;

Network data movement during distributed-memory SpMV and cache misses during shared-memory SpMV are both bounded by

\[ \sum_i (\lambda_i - 1), \]

the \( \lambda - 1 \)-metric (row-net model and zig-zag storage; Y & B, ’09).

Matrix permutations

Row-net model:

- columns correspond to vertices, rows to hyperedges.

Matrix permutations

Can be done in 2D (medium- & fine-grain models) too. In practice:

Sequential execution using CRS on Stanford:

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

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

Ref.: A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices by Çatalyürek & Aykanat (2001)
Two-dimensional cache-oblivious sparse matrix-vector multiplication by Yzelman & Bisseling (2011)
A medium-grain method for fast 2D bipartitioning of sparse matrices by Pelt & Bisseling (2014)
A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric SpMV by Alappat et al. (2019)
Nonzero reorderings

2D permutations cannot rely on CRS:
- vertical separators pollute cache; thus
- need to store nonzeroes in specific order.

This makes sense by itself as well:

Blocking combined with \textit{cache-oblivious traversals}
Nonzero reorderings

2D permutations cannot rely on CRS:
▶ vertical separators pollute cache; thus
▶ need to store nonzeros in specific order.

This makes sense by itself as well:

Hilbert on blocked dense matrix storage: Lorton & Wise, 2007
Nonzero reorderings

2D permutations cannot rely on CRS:
▶ vertical separators pollute cache; thus
▶ need to store nonzeroes in specific order.

This makes sense by itself as well:

Hilbert with COO: Haase, Liebmann, & Plank, 2007
Nonzero reorderings

2D permutations cannot rely on CRS:

- vertical separators pollute cache; thus
- need to store nonzeros in specific order.

This makes sense by itself as well:

Hilbert with compression: Yzelman & Bisseling, ECMI ’09
Nonzero reorderings

2D permutations cannot rely on CRS:

▶ vertical separators pollute cache; thus
▶ need to store nonzeroes in specific order.

This makes sense by itself as well:

Blocking with Morton inside blocks, COO+CRS: Buluç et al., 2009
Nonzero reorderings

2D permutations cannot rely on CRS:
- vertical separators pollute cache; thus
- need to store nonzeroes in specific order.

This makes sense by itself as well:

Morton-ordered blocks, quadtree store: Martone et al., 2010
Nonzero reorderings

2D permutations cannot rely on CRS:

- vertical separators pollute cache; thus
- need to store nonzeroes in specific order.

This makes sense by itself as well:

Hilbert-ordered blocks, fully compressed: Y & Bisseling, 2011
Nonzero reorderings

2D permutations cannot rely on CRS:
- vertical separators pollute cache; thus
- need to store nonzeroes in specific order.

This makes sense by itself as well:

Blocking with Hilbert-ordered blocks: Y & Roose, 2014
Nonzero reorderings

2D permutations cannot rely on CRS:
▶ vertical separators pollute cache; thus
▶ need to store nonzeroes in specific order.

This makes sense by itself as well:

Sequential SpMV multiplication on the Wikipedia ’07 link matrix:
345 (CRS), 203 (Hilbert), 245 (blocked Hilbert) ms/mul.
Cache-efficiency and bandwidth

Need to consider the whole picture; good cache efficiency but no compression? Compression but no cache optimisation? No gain!

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
0 & 0 & 0 & 2 \\
7 & 5 & 1 & 1
\end{pmatrix}
\]

Bi-directional incremental CRS (BICRS):

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

Allows arbitrary traversals. Storage: \(\Theta(2nz + \text{row}_jumps + 1)\).
Vectorisation: no use, right?

Much faster with vectorisation on Xeon Phi 7120A (KNC). Why?

Latency-bound, not bandwidth-bound!

Gather/scatter is critical.

Vectorisation: no use, right?

Much faster with vectorisation on Xeon Phi 7120A (KNC). Why?
Vectorisation: no use, right?

Much faster with vectorisation on Xeon Phi 7120A (KNC). Why? Latency-bound, not bandwidth-bound! **Gather/scatter** is critical.

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

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

Access to only one socket: limited bandwidth
Interleave memory pages across sockets: emulate uniform access
Explicit data placement on sockets: best performance
One-dimensional data placement

Coarse-grain row-wise distribution, compressed, cache-optimised:

- explicit allocation of separate matrix parts per core,
- explicit allocation of the output vector on the various sockets,
- interleaved allocation of the input vector.

Two-dimensional data placement

Distribute row- and column-wise (individual nonzeros):

- most work touches only local data,
- inter-process communication minimised by partitioning;
- incurs cost of partitioning.


Some kernels illustrated

Fine-grained CRS, using OpenMP:

- no pre-processing required (vs. sequential)
- use numactl –interleave=all on NUMA system

```c
#pragma omp parallel for private(i, k) schedule(dynamic, 8)
1:   for i = 0 to m − 1 do
2:     for k = \hat{i}_i to \hat{i}_{i+1} − 1 do
3:       add V_k · x_{J_k} to y_i
```

Compressed Sparse Blocks, using Cilk:

- Block A into $\beta \times \beta$ submatrices,
- use numactl –interleave=all on NUMA system,
- omitted: need buffer if multiple threads on a row.

```c
for each row of blocks
1:   cilk_for each block
2:     for each block
3:       do SpMV with nonzeros in Z-curve order
```
Some kernels illustrated

1D SpMV multiplication:
  ▶ when loading in $A$, distribute rows over threads
  1. perform local SpMV
(That’s really it!)

Caveat:
  ▶ take care of vector operations— they’re distributed!
Some kernels illustrated

2D SpMV multiplication:

- partition and reorder $A$

1. **for each** $j$ s.t. $\exists a_{ij}$ local to $s$ while $x_j$ is not local **do**
2. $bsp\_get x_j$ from remote process
3. $bsp\_sync()$
Some kernels illustrated

2D SpMV multiplication:
- partition and reorder $A$

1: for each $j$ s.t. $\exists a_{ij}$ local to $s$ while $x_j$ is not local do
2: $\text{bsp\_get } x_j$ from remote process
3: $\text{bsp\_sync()}$
4: Perform SpMV $y = Ax$, using only those $a_{ij}$ local to $s$
Some kernels illustrated

2D SpMV multiplication:

- partition and reorder $A$

1. for each $j$ s.t. $\exists a_{ij}$ local to $s$ while $x_j$ is not local do
2. $bsp\_get x_j$ from remote process
3. $bsp\_sync()$
4. Perform SpMV $y = Ax$, using only those $a_{ij}$ local to $s$
5. for each $i$ s.t. $\exists a_{ij}$ local to $s$ while $y_i$ is not local do
6. $bsp\_send (y_i, i)$ to the owner of $y_i$
7. $bsp\_sync()$
Some kernels illustrated

2D SpMV multiplication:

- partition and reorder $A$

1: for each $j$ s.t. $\exists a_{ij}$ local to $s$ while $x_j$ is not local do
2: \quad bsp\_get $x_j$ from remote process
3: \quad bsp\_sync()
4: Perform SpMV $y = Ax$, using only those $a_{ij}$ local to $s$
5: for each $i$ s.t. $\exists a_{ij}$ local to $s$ while $y_i$ is not local do
6: \quad bsp\_send ($y_i$, $i$) to the owner of $y_i$
7: \quad bsp\_sync()
8: while $bsp\_qsize() > 0$ do
9: \quad ($\alpha$, $i$) = $bsp\_move()$
10: add $\alpha$ to $y_i$
Results

Sequential CRS on Wikipedia ’07: 472 ms/mul. 40 threads BICRS:

$$21.3 \text{ (1D), } 20.7 \text{ (2D) ms/mul. Speedup: } \approx 22x.$$ 

4 sockets, 10 core Intel Xeon E7-4870
### Results

Average speedup on six large matrices:

<table>
<thead>
<tr>
<th></th>
<th>2 x 6</th>
<th>4 x 10</th>
<th>8 x 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>–, 1D fine-grained, CRS*</td>
<td>4.6</td>
<td>6.8</td>
<td>6.2</td>
</tr>
<tr>
<td>Blocking, Morton, 1D FG, CSB</td>
<td>7.9</td>
<td>24.3</td>
<td>26.3</td>
</tr>
<tr>
<td>Hilbert, Blocking, 1D, BICRS*</td>
<td>5.4</td>
<td>19.2</td>
<td>24.6</td>
</tr>
<tr>
<td>Hilbert, Blocking, 2D, BICRS†</td>
<td>–</td>
<td>21.3</td>
<td><strong>30.8</strong></td>
</tr>
</tbody>
</table>

†: uses an updated test set. (Added for reference versus a good 2D algorithm.)

As NUMA scales up, 1D algorithms lose efficiency.
## Results

**Cross-platform:**

<table>
<thead>
<tr>
<th></th>
<th>Structured</th>
<th>Unstructured</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi</td>
<td>21.6</td>
<td>8.7</td>
<td>15.2</td>
</tr>
<tr>
<td>2x Ivy Bridge CPU</td>
<td>23.5</td>
<td>14.6</td>
<td>19.0</td>
</tr>
<tr>
<td>NVIDIA K20X GPU</td>
<td>16.7</td>
<td>13.3</td>
<td>15.0</td>
</tr>
</tbody>
</table>


Application to GraphBLAS

In summary: we know how to do high performance parallel SpMV.

▶ what about sparse matrix–sparse vector multiply (SpMSpV)?
▶ what about masks?

\[ y = Ax, \quad x \text{ sparse, no mask.} \]

▶ Best data structure? Column-major!

\[ y = Ax, \quad x \text{ dense, but masked (} y(\text{find}(m == 0)) = 0). \]

▶ Best data structure? Row-major!

\[ y = Ax, \quad x \text{ sparse, and masked?} \]

▶ ...
Application to GraphBLAS

In summary: we know how to do high performance parallel SpMV.

- what about sparse matrix–sparse vector multiply (SpMSpV)?
- what about masks?

Suppose $y = Ax$, $x$ sparse, no mask.

- Best data structure?
In summary: we know how to do high performance parallel SpMV.

▶ what about sparse matrix–sparse vector multiply (SpMSpV)?
▶ what about masks?

Suppose \( y = Ax \), \( x \) sparse, no mask.
▶ Best data structure? Column-major!
Application to GraphBLAS

In summary: we know how to do high performance parallel SpMV.
- what about sparse matrix–sparse vector multiply (SpMSpV)?
- what about masks?

Suppose $y = Ax$, $x$ sparse, no mask.
- Best data structure? Column-major!

Suppose $y = Ax$, $x$ dense, but masked ($y(find(m == 0)) = 0$).
- Best data structure?
Application to GraphBLAS

In summary: we know how to do high performance parallel SpMV.

▶ what about sparse matrix–sparse vector multiply (SpMSpV)?
▶ what about masks?

Suppose $y = Ax$, $x$ sparse, no mask.

▶ Best data structure? Column-major!

Suppose $y = Ax$, $x$ dense, but masked ($y(find(m == 0)) = 0$).

▶ Best data structure? Row-major!
Application to GraphBLAS

In summary: we know how to do high performance parallel SpMV.

▶ what about sparse matrix–sparse vector multiply (SpMSpV)?
▶ what about masks?

Suppose \( y = Ax \), \( x \) sparse, no mask.

▶ Best data structure? Column-major!

Suppose \( y = Ax \), \( x \) dense, but masked (\( y(find(m == 0)) = 0 \)).

▶ Best data structure? Row-major!

Suppose \( y = Ax \), \( x \) sparse, and masked?

▶ ...
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

- $y = Ax$, loop over nonzero indices of $x$: scatter, column-major
- $y = Ax$, loop over mask indices: gather, row-major
- $y = xA$, loop over nonzero indices of $x$: scatter, row-major
- $y = xA$, loop over mask indices: gather, column-major

Store matrix twice (row- and column-major). At runtime:

- choose variant with smallest loop-size.

Vector data structure, $\Theta(1)$ ops required for:

- checking if the $i$th vector entry is (non)zero
- jump to the next nonzero (iteration);

Use both an array & a stack to maintain vector nonzero indices.
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

- $y = Ax$, loop over nonzero indices of $x$: scatter, column-major
- $y = Ax$, loop over mask indices: gather, row-major

Store matrix twice (row- and column-major). At runtime:

- choose variant with smallest loop-size.

Vector data structure, $\Theta(1)$ ops required for:

- checking if the $i$th vector entry is (non)zero
- jump to the next nonzero (iteration)

Use both an array & a stack to maintain vector nonzero indices.
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

- $y = Ax$, loop over nonzero indices of $x$: scatter, column-major
- $y = Ax$, loop over mask indices: gather, row-major
- $y = xA$, loop over nonzero indices of $x$: scatter, row-major
- $y = xA$, loop over mask indices: gather, column-major

Store matrix twice (row- and column-major). At runtime:
- choose variant with smallest loop-size.

Vector data structure, $\Theta(1)$ ops required for:
- checking if the $i$th vector entry is (non)zero
- jump to the next nonzero (iteration)

Use both an array & a stack to maintain vector nonzero indices.
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

▶ \( y = Ax \), loop over nonzero indices of \( x \): scatter, column-major
▶ \( y = Ax \), loop over mask indices: gather, row-major
▶ \( y = xA \), loop over nonzero indices of \( x \): scatter, row-major
▶ \( y = xA \), loop over mask indices: gather, column-major

Store matrix twice (row- and column-major). At runtime:

▶ choose variant with smallest loop-size.
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

- $y = Ax$, loop over nonzero indices of $x$: scatter, column-major
- $y = Ax$, loop over mask indices: gather, row-major
- $y = xA$, loop over nonzero indices of $x$: scatter, row-major
- $y = xA$, loop over mask indices: gather, column-major

Store matrix twice (row- and column-major). At runtime:
- choose variant with smallest loop-size.

Vector data structure, $\Theta(1)$ ops required for:
- checking if the $i$th vector entry is (non)zero
- jump to the next nonzero (iteration);
Application to GraphBLAS

Sequential mode, four different SpM(Sp)V kernels:

- $y = Ax$, loop over nonzero indices of $x$: scatter, column-major
- $y = Ax$, loop over mask indices: gather, row-major
- $y = xA$, loop over nonzero indices of $x$: scatter, row-major
- $y = xA$, loop over mask indices: gather, column-major

Store matrix twice (row- and column-major). At runtime:
- choose variant with smallest loop-size.

Vector data structure, $\Theta(1)$ ops required for:

- checking if the $i$th vector entry is (non)zero
- jump to the next nonzero (iteration);

Use both an array & a stack to maintain vector nonzero indices.
Application to GraphBLAS

We start simple. For shared-memory parallel:

- use OpenMP-based parallelisation

For distributed-memory parallel:

- 1D row-wise block-cyclic distribution of matrices
- matching block-cyclic distribution of vectors
- rely on OpenMP (or sequential) backend

Due to row-wise 1D distribution over $p$ processes:

- $y = Ax$ results in $p$ vectors $y_s$, while $y = \sum_k y_k$

Use stack-based synchronisation/reduction when useful:

- choose variant with lowest communication cost,
- requires standard collectives: allreduce, allgather, alltoall(v).
Application to GraphBLAS

We start simple. For shared-memory parallel:
  - use OpenMP-based parallelisation

For distributed-memory parallel:
  - 1D row-wise block-cyclic distribution of matrices
  - matching block-cyclic distribution of vectors
  - rely on OpenMP (or sequential) backend

Due to row-wise 1D distribution over $p$ processes:

$$ y = Ax \text{SpM(Sp)V requires globally synchronised/replicated}$$

$$ y = xA \text{SpM(Sp)V results in} p \text{ vectors}$$

Use stack-based synchronisation/reduction when useful:
  - choose variant with lowest communication cost,
  - requires standard collectives: allreduce, allgather, alltoall(v).
Application to GraphBLAS

We start simple. For shared-memory parallel:
- use OpenMP-based parallelisation

For distributed-memory parallel:
- 1D row-wise block-cyclic distribution of matrices
- matching block-cyclic distribution of vectors
- rely on OpenMP (or sequential) backend

Due to row-wise 1D distribution over \( p \) processes:
- \( y = Ax \ SpM(Sp)V \) requires globally synchronised/replicated \( x \)
- \( y = xA \ SpM(Sp)V \) results in \( p \) vectors \( y_s \), while \( y = \sum_{k} y_k \)
Application to GraphBLAS

We start simple. For shared-memory parallel:
▶ use OpenMP-based parallelisation

For distributed-memory parallel:
▶ 1D row-wise block-cyclic distribution of matrices
▶ matching block-cyclic distribution of vectors
▶ rely on OpenMP (or sequential) backend

Due to row-wise 1D distribution over $p$ processes:
▶ $y = Ax \text{ SpM}(Sp)V$ requires globally synchronised/replicated $x$
▶ $y = xA \text{ SpM}(Sp)V$ results in $p$ vectors $y_s$, while $y = \sum_k y_k$

Use stack-based synchronisation/reduction when useful:
▶ choose variant with lowest communication cost,
▶ requires standard collectives: allreduce, allgather, alltoall(v).
In relation to graph frameworks

Since Google’s Pregel, a plethora of graph frameworks:

- CombBLAS, Gunrock, GraphX, Giraph, Ligra, PowerGraph, PowerLyra, Galois, GraphChi, TigerGraph, Venus, Neo4j, ArangoDB, Titan, ...
In relation to graph frameworks

Since Google’s Pregel, a plethora of graph frameworks:
▶ CombBLAS, Gunrock, GraphX, Giraph, Ligra, PowerGraph, PowerLyra, Galois, GraphChi, TigerGraph, Venus, Neo4j, ArangoDB, Titan, ...

One of the main Ligra features:
▶ automatic push/pull ‘direction-optimisation’
▶ also done by us: scatter/gather & row/col kernel selection
In relation to graph frameworks

Since Google’s Pregel, a plethora of graph frameworks:

- CombBLAS, Gunrock, GraphX, Giraph, Ligra, PowerGraph, PowerLyra, Galois, GraphChi, TigerGraph, Venus, Neo4j, ArangoDB, Titan, ...

One of the main Ligra features:

- automatic push/pull ‘direction-optimisation’
- also done by us: scatter/gather & row/col kernel selection

In relationship to other frameworks:

- MapReduce maps to BLAS level 1 (only vector ops)
- Pregel’s vertex-centric programs map directly to SpM(Sp)V
- See, e.g., GraphMAT by Sundaram et al. (2015).
Interoperability

One way of dealing with many auxiliary frameworks:

▶ be compatible with most of them
▶ matches LPF’s philosophy (Suijlen & Y, 2019)
Interoperability

One way of dealing with many auxiliary frameworks:

▶ be compatible with most of them
▶ matches LPF’s philosophy (Suijlen & Y, 2019)
▶ distributed-memory GraphBLAS built on top of LPF
Interoperability

One way of dealing with many auxiliary frameworks:

- be compatible with most of them
- matches LPF’s philosophy (Suijlen & Y, 2019)
- distributed-memory GraphBLAS built on top of LPF
- allows interfacing with any framework over TCP/IP
- LPF processes reside in same process space as the host
Interoperability

One way of dealing with many auxiliary frameworks:
- be compatible with most of them
- matches LPF’s philosophy (Suijlen & Y, 2019)
- distributed-memory GraphBLAS built on top of LPF
- allows interfacing with any framework over TCP/IP
- LPF processes reside in same process space as the host

Applications:
- Spark ML acceleration using LPF
- Graph Analytics acceleration using LPF + GraphBLAS
- GraphBLAS on Spark
- Graph DB on Edge
Future work & Outlook

Implementation:
▶ use of Suijlen’s new BSP collectives (2019)
▶ compatibility layer with the C11 standard

Future Work:
▶ incorporate more advanced matrix partitioning methods?
▶ how to extend to multi-linear algebra (tensor computations)?
   ▶ applicable to hypergraph computations?
▶ graph- and sparse neural networks
▶ for which other (graph) algorithms is GraphBLAS applicable?
   ▶ which additions would enlarge suitable areas significantly?
Backup slides
A working example:

```cpp
#include <graphblas.hpp>
int main() {
    const size_t num_cities = ... //some input matrix size
    grb::init();
    grb::Matrix< double > distances( num_cities, num_cities )
    grb::build( distances, ... ); //input data from file
                                   //or memory
    grb::Vector< double > x( num_cities ), y( num_cities );
    grb::set( x, 0.0, 4 );    //set city number 4 to
    //have distance 0.0
    ...
```
A working example (continued):

...  
//declare an alternative semiring on doubles:
grb::Semiring< double, double, double, double, 
grb::operators::min,     //‘plus’
grb::operators::add,     //‘multiply’
grb::identities::infinity //‘0’
grb::identities::zero    //‘1’
> ring;

//calculate the shortest distances from all cities to
//city #4, allowing only a single path
grb::mxv( y, distances, x, ring );
...
A working example (continued):

...  
//calculate the shortest distances from all cities to  
//city #4, allowing only a single path  
grb::mxv( y, distances, x, ring );

//calculate the shortest distances from all cities to  
//city #4, allowing two ‘hops’  
grb::mxv( x, distances, y, ring );

//example output via iterators and exit:  
writeResult( x.cbegin(), x.cend(), ... );  
grb::finalize();  
return 0;
Vectorised BICRS
Vectorised BICRS: \( l = p \times q = 2 \times 2 \)

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

while there are nonzero blocks do

load next \( l \) row indices into \( r_5 \)

\[
r_5 = (0, 1, 2, 3)
\]

gather output vector elements into \( r_6 \) (using \( r_5 \))

\[
r_6 = (y_0, y_1, y_2, y_3)
\]

for \( offset = 0 \) to \( l - 1 \) step \( p \)

set \( r_0 \) to all zero

handle all nonzero blocks sharing these rows

...
Vectorised BICRS: $2 \times 2$

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

for each nonzero block do

load nonzeros into $r_3$, load nonzero column indices in $r_4$

$$r_3 = (4, 1, 2, 3), \quad r_4 = (0, 1, 2, 3)$$
Vectorised BICRS: $2 \times 2$

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

for each nonzero block do

load nonzeros into $r_3$, load nonzero column indices in $r_4$

$$r_3 = (4, 1, 2, 3), \quad r_4 = (0, 1, 2, 3)$$

gather corresponding elements from $x$ into $r_2$ (using $r_4$)

$$r_2 = (x_0, x_1, x_2, x_3)$$
Vectorised BICRS: $2 \times 2$

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

**for** each nonzero block **do**

load nonzeros into $r_3$, load nonzero column indices in $r_4$

$$r_3 = (4, 1, 2, 3), \quad r_4 = (0, 1, 2, 3)$$

gather corresponding elements from $x$ into $r_2$ (using $r_4$)

$$r_2 = (x_0, x_1, x_2, x_3)$$

do vectorised multiply-add

$$r_1 = r_1 + r_2 \circ r_3$$
Vectorised BICRS: $2 \times 2$

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

while there are nonzero blocks do

... 

set $r_0$ to all zero

handle all nonzero blocks sharing these rows

reduce output $r_1$ into $r_6$

$$r_6 = (y_{offset} + (r_1)_1 + (r_1)_2, y_{offset+1} + (r_1)_3 + (r_1)_4, \ldots)$$

end for

scatter $r_6$ unto $y$ (using $r_5$)

end while
Sequential SpMV: reordering
Sequential SpMV: reordering

Column partitioning
Sequential SpMV: reordering

Column permutation
Permuting to SBD form

Mixed row detection
Permuting to SBD form

Row permutation
Sequential SpMV: reordering

- No cache misses
- 1 cache miss per row
- 3 cache misses per row
- 1 cache miss per row
Sequential SpMV: reordering

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
Sequential SpMV: reordering

1D (\(p = 20, \epsilon = 0.1\))  Finegrain (\(p = 100, \epsilon = 0.1\))