HIGH PERFORMANCE SPARSE COMPUTATIONS APPLIED TO A PARALLEL CONJUGATE GRADIENT SOLVER

A. N. YZELMAN

Abstract. Many advanced sparse matrix storage schemes have appeared in literature, demonstrating large gains in efficiency for basic operations such as the sparse matrix–vector (SpMV) multiplication. One class of methods, based on sparse matrix partitioning and reordering, sparse blocking, compression, and cache-oblivious nonzero traversals, gained two to three factors of increased performance on single SpMV multiplications, but had yet to be used in practice.

This work explores its use within a Conjugate Gradients (CG) iterative solver for linear systems, and analyses the resulting algorithm using the Bulk Synchronous Parallel (BSP) paradigm. It is implemented using C++11 and the high-performance MulticoreBSP for C programming framework, and tested on real-world problems on a shared-memory dual-socket architecture. Its performance is compared against Blaze, a C++ library for high-performance numerical linear algebra. The CG solver developed using the BSP paradigm and advanced sparse computation techniques results in a total speedup of almost 10x, close to a 3x performance increase over Blaze.

1. Introduction

This paper deals with the high performance implementation of the Conjugate Gradient (CG) solver shown in Algorithm 1, derived from O'Leary [26], here shown in Matlab. CG solves linear systems of equations $Ax = b$ for symmetric semi-positive definite $n \times n$ matrix A. We are interested in cases where A is sparse, i.e., containing $nz \ll n^2$ non-zeroes.

Instead of focusing on the CG method, this paper attempts to integrate many advanced techniques regarding sparse matrix computations into a CG algorithm. It does so following the Bulk Synchronous Parallel (BSP) paradigm briefly explained in Section 2, which provides a clear framework of algorithm design according to a prescribed cost model. The model clearly identifies which parts of any algorithm are performance bottlenecks in terms of data movement and other overheads, thus easily identifying areas for improvement.

Since the CG algorithm consists only of dense vector updates (BLAS1) and sparse matrix–vector (SpMV) multiplication, optimising the latter leads to direct performance gains for the entire algorithm. Section 3 summarises the state of the art in sequential SpMV multiplication and sparse matrix storage formats. We focus on shared-memory parallel machines to demonstrate data movement is a limiting factor on those architectures, where data distribution in principle is not required to obtain working parallel CG algorithms: the Matlab CG from Algorithm 1 can, for instance, be implicitly distributed by the use of parallel-for constructs. Section 4 describes this approach.

We apply advanced data formats and computational kernels for sparse computations within an explicitly distributed CG algorithm in Section 5. As the cost of data movement inside a shared-memory node is increasing as non-uniform memory access (NUMA) effects become increasingly pronounced while the effective bandwidth-per-core remains decreasing, we expect this distributed CG to outperform its implicitly distributing counterpart, especially on multi-socket architectures. An experimental comparison appears in Section 6, and we conclude in Section 7.

2. Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model provides a theoretical framework for parallel algorithm design. It encompasses (1) modelling of a parallel computer, (2) modelling of a parallel algorithm, and (3) a cost model that connects the two. The following paragraphs provides a

Algorithm 1 A Matlab CG implementation

Arguments:

in	*max*	the maximum number of CG iterations
in	*tol*	the accepted tolerance
in	A	the input matrix
in	b	the right-hand side of $Ax = b$
in	x	initial guess of the solution x
out	x	the solution x to $Ax = b$

matlab_cg:

```
1:  r = b - A*x;
2:  u = r;
3:  sigma = r' * r;
4:  for iteration=1:max
5:      e = A*u;
6:      omega = u'*e;
7:      alpha = sigma / omega;
8:      x += u * alpha;
9:      r -= e * alpha;
10:     omega = r' * r;
11:     if sqrt(omega < tol)
12:         break;
13:     end
14:     beta = omega / sigma;
15:     u = r + u * beta;
16:     sigma = omega;
17: end
```

short overview of BSP necessary for understanding the algorithm analysis later on. For full details on BSP, please refer to its introductory paper by Valiant [31], the introductory textbook by Bisseling [3], or the recent paper on MulticoreBSP [37].

A parallel computer consists of p processors that operate as sequential computers using local computational units and local memory. All processors are assumed homogeneous and are connected to a shared full-duplex network optimised for all-to-all communication. During all-to-all communication using this network, a processor can simultaneously send and receive a single data word at the cost g, the message gap. Preparing a processor to participate in an all-to-all exchange incurs a latency cost l.

A parallel algorithm consists of a single program that is executed on multiple processors concurrently. A BSP program is expressed as a sequence of *supersteps*, where every superstep consists of a sequential computation phase that uses only local resources, followed by a communication phase that only executes remote communication. The program is parametrised in the integers s and p, where $p > 0$ and $0 \leq s < p$. The same program is run on the p processors of a parallel computer, where s then becomes the local process ID. The program should function for any nonzero integer p such that it may run on any BSP computer (p, g, l).

We assume that the computation phases that take place on sequential processors are understood in terms of performance, and use the notion of h-relations to understand the communication phases: suppose the BSP program transmits $t_{s,p,i}$ data words at superstep i for a given s, p and receives $r_{s,p,i}$ data words at the same superstep, then the h-relation of this superstep is

$$h_{p,i} = \max\{\max_s t_{s,p,i}, \max_s r_{s,p,i}\};$$

BSP assumes the bottleneck of communication lies at its end points. Note that the h-relation is a global property that only depends on p, and not on s.

The BSP cost model combines all the above to express the cost T_p of executing a given BSP program on a given BSP computer. Let the N be the number of supersteps the BSP program

consists of. Then,

$$T_p = \sum_{i=0}^{N-1} \left(\max_s w_{s,p,i} + h_{p,i} g + l \right),$$

with $w_{s,p,i}$ the cost of the local compute phase at the ith superstep at the BSP program with s, p. Typically, costs also depend on the problem input which oftentimes may be expressed as a single integer, such as, in our case, n.

It is clear that any good parallel program minimises the following expressions:

$$T_{\text{sync},p} = Nl, \tag{1}$$

$$T_{\text{comp},p} = \sum_{i=0}^{N-1} \max_s w_{s,p,i}, \tag{2}$$

$$T_{\text{comm},p} = \sum_{i=0}^{N-1} h_{p,i} g. \tag{3}$$

An *immortal algorithm* provably minimises all three aspects for all BSP computers and for any input. Usually, however, an immortal algorithm makes trade offs between synchronisation, computation, and communication overheads.

A fourth expression of interest considers the maximum memory usage $M_{s,p}$ of given input program for given input, s, and p. Ideally, $p \max_s M_{s,p}$ equals the maximum memory usage M_{seq} of the best sequential program, although relaxing this requirement often results in lower bounds for one or more of Equations 1–3. Dense matrix–matrix multiplication is a well-known example for which an immortal algorithm can be optimal in up to two of the three bounds simultaneously, while continuous trade-offs exist, including improved bounds at the cost of additional memory usage [25].

3. Sequential SpMV multiplication

The simplest format for sparse computations is the COO data structure, consisting of three arrays I, J, V of length nz. Given an arbitrary nonzero ordering, the kth nonzero $(a_{ij})_k$ of A is stored in COO by $I[k] = i$, $J[k] = j$, and $V[k] = a_{ij}$. The Compressed Row Storage (CRS) [1], previously also known as the Yale format [13] and more recently also known as CSR [29], reduces the store requirement from $\Theta(3nz)$ to $\Theta(2 \cdot nz + m)$ bytes. Here, nonzeroes are sorted in a row-major order, such that a SpMV multiplication kernel may be written as in Algorithm 2.

Algorithm 2 Pseudocode for SpMV multiplication using the CRS format

crs_spmv:

```
1: for i = 0 to m − 1 do
2:     for k = row_start_i to row_start_{i+1} do
3:         Add v_k · x_{col_ind_k} to y_i
```

While CRS still remains the de-facto standard for sparse matrix storage, it does nothing to alleviate the two main obstacles to efficient SpMV multiplication: (1) poor cache efficiency and (2) being bandwidth-bound.

Efficient cache use. Since the matrix is read-in only once during SpMV multiplication, it cannot benefit from caching in all but the trivial case where the entire computation fits in cache. Vector elements, however, are re-used during the multiplication and for unstructured matrices of large dimensions this leads to high performance penalties. Solutions are either cache-aware and matrix-aware, such as with OSKI by Vuduc et al. who perform online sparse matrix analysis and machine-aware auto-tuning [33]. A different strategy uses recursive structures to obtain cache-oblivious behaviour on general sparse matrices, by, e.g., using space-filling curves. Nonzeroes ordered according to the Hilbert curve lead to better cache performance at a $\Theta(nz \log nz)$ expected cost of pre-processing due to sorting the nonzeroes [18]. The adapted traversal of nonzeroes during

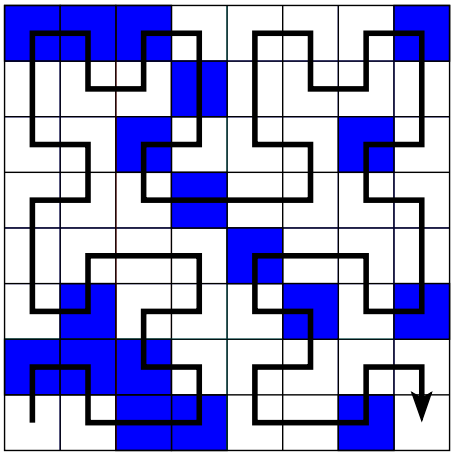

FIGURE 1. A sparse matrix with nonzeroes traversed in Hilbert order

SpMV multiplication, illustrated in Figure 1 disallows the use of CRS, which led to the development of the Bi-directional Incremental CRS (BICRS) data structure that retains compression but allows arbitrary nonzero orderings [36].

Buluç et al. divide a sparse matrix into smaller square blocks of size $\beta \ll n$. Nonzeroes within each block are ordered according to the Morton Z-curve, while processing blocks in a row-major order as this helps row-wise parallelisation [6]. Yzelman and Roose do the reverse and impose that all blocks are visited in a cache-oblivious Hilbert order. The nonzeroes within each block are ordered row-major to minimise the sparse matrix storage footprint [38]. Sparse blocking with cache-oblivious reordering on the top level helps improve efficient use of the top-level caches while additionally reducing the cost of sorting according to Hilbert coordinates to $\Theta(n \log n)$, as long as β is chosen proportionally to $\sqrt{n}$.

Compression. Since the SpMV multiplication is limited by the speed at which data arrives at the CPU, especially in a fully utilised parallel shared-memory architecture, every byte saved in the sparse matrix data structure directly results in an increased speed of the multiplication kernel.

Block-wise storage of sparse matrices using CRS would cost $\Theta\left(2nz + n^2/\beta^2 \cdot (\beta + 1)\right)$ bytes and thus does not scale due to the quadratic growth in n. Using COO at $\Theta(3nz)$ bytes scales, but loses compression. Buluç et al. show that under the weak condition of hypersparsity, blocked storage using only $\lceil \log_2 \beta \rceil$ bytes per index element uses the same amount of memory as non-blocked CRS. Yzelman and Roose combine this observation with the BICRS data structure to attain the memory footprint of CRS as a worst-case upper bound, while normally achieving a compression rate beyond that of CRS while continuing to support arbitrary nonzero orders [38].

Cost. The cost of sequential SpMV multiplication in flops is $2nz$, where 'flops' stands for *floating point operation*s such as the addition or multiplication of two scalars. Since the SpMV multiplication, and by extension the CG algorithm, are bandwidth bound, we also explicitly count the number of bytes needed to be moved during multiplication.

This amounts to streaming in the full data structure of A, which is bounded from above by the storage requirement of CRS: $nz \cdot (b_{\text{val}} + b_{\text{ind}}) + b_{\text{ind}}(n + 1)$ bytes, with b_{val} the number of bytes required to store a nonzero value and b_{ind} the number of bytes required to store an index value. In our analyses we drop lower-order terms such as single constants, and we assume nothing is cached, thus assuming that $2b_{\text{val}}nz$ bytes are read from the input and output vectors during SpMV multiplication. The total cost thus becomes

$$T_{\text{seq-spmv}} = 2nz \text{ flops} + nz \cdot (3b_{\text{val}} + b_{\text{ind}}) + nb_{\text{ind}} \text{ bytes}.$$

4. Non-distributed shared-memory parallel CG

The sequential CG from Algorithm 1 can be parallelised on shared-memory via algorithmic modifications that are low cost in terms of implementation effort: parallel-for constructs. For iterative solvers like CG, distributing matrices row-wise and distributing vectors accordingly avoids data race conditions. In case of fine-grained scheduling, aligned memory allocation and choosing an appropriate chunk size prevents inefficient cache use and false sharing conditions; on contemporary architectures this usually means aligning on 64 bytes and choosing a chunk size of 8 double-precision values.

Parallelisation is easily incorporated using OpenMP: A coarse-grained row-wise 1D distribution is easily incorporated using OpenMP by adding `#omp parallel for` before line 1 in Algorithm 2. For a fine-grained row-wise distribution, add `#omp parallel for schedule(dynamic,8)` instead, assuming all data structures are properly aligned. All (BLAS1) vector operations, i.e., additions and dot-products, can be parallelised similarly.

There exist several libraries that allow for Matlab-style syntax in C++ programs through operator overloading. The ease of programming is evident, providing sequential semantics using a linear algebra notation. Modern libraries provide loop fusion and automatically parallelise linear algebra operations: examples are Eigen [17] and Blaze [20]. Algorithm 3 shows the C++ code using Blaze that is mathematically equivalent to the Matlab code in Algorithm 1. Note that while the Blaze and Matlab codes look similar and the mathematics is equivalent, they certainly do result in a different algorithm due to different orders of execution due to work distribution and concurrency.

Algorithm 3 CG in C++ using Blaze

Arguments:

in	`size_t`	n	size of the matrix A
in	`size_t`	max	maximum CG iterations
in	`double`	tol	the accepted tolerance
in	`blaze::CompressedMatrix< double, blaze::rowMajor >`	A	the input matrix
in	`blaze::DynamicVector< double, blaze::columnVector >`	b	right-hand side of $Ax = b$
in	`blaze::DynamicVector< double, blaze::columnVector >`	x	initial guess to the solution
out	`blaze::DynamicVector< double, blaze::columnVector >`	x	the solution to $Ax = b$

blaze_cg:

```
1: double alpha, beta, sigma, omega;
2: blaze::DynamicVector< double, blaze::columnVector > r(n), e(n), u(n);
3: r = b − A ∗ x;
4: u = r;
5: σ = (r, r);
6: for( size_t iterations = 0; iterations < max; ++iterations ) {
7:         e = A ∗ u;
8:         omega = (u, e);
9:         alpha = sigma/omega;
10:        x += alpha ∗ u;
11:        r -= alpha ∗ e;
12:        omega = (r, r);
13:        if( std::sqrt(omega) < tol ) break;
14:        beta = omega/sigma;
15:        u = r + beta ∗ u;
16:        sigma = omega;
17: }
```

A good CG implementation, whether custom-coded or programmed using linear algebra template libraries, should fuse for-loops whenever possible. The three BLAS1 operations on lines 8–10 in Algorithm 1, for instance, should be implemented as in Algorithm 4, and not as three separate

for loops. This is especially critical for shared-memory parallel architectures where the bandwidth-per-core is severely limited; compute times of BLAS1 operations are bounded by bandwidth, hence streaming vectors as few times as possible directly translates to increased performance.

Algorithm 4 Fused BLAS1 implementation of lines 8–10 from Algorithm 1

```
1: set ω = 0
2: for i = 0 to n − 1 do
3:     add αu_i to x_i
4:     add −αe_i to r_i
5:     add r_i^2 to ω
```

Cost. The cost of this algorithm assuming that CRS storage is used, nothing is cached, and all *max* iterations are made, is

$$T_{\text{seq-cg}} = max\left((T_{\text{seq-spmv}} + 10n \text{ flops} + 10b_{\text{val}}n \text{ bytes}\right). \tag{4}$$

The data movement incurred, however, is oblivious to data location. This leads to suboptimal performance on NUMA architectures: data is accessed through shared caches or via off-socket memory controllers, taking much more time than when data were in local caches or in local memory banks only. An algorithm that quantifies local data movement (in bytes) versus non-local data movement (in the BSP g and l), is explored in the following section.

5. Distributed-memory parallel CG

We first proceed with sketching the state-of-the-art of sparse matrix partitioning, sparse matrix reordering, sparse matrix data structure compression, cache-oblivious process-local SpMV multiplication, and, finally, vector distribution. References to more detailed papers on each of the above sections are given in the corresponding sections. This section ends with a global view of the resulting parallel CG algorithm and its cost.

5.1. Sparse matrix distribution.

Let p be the number of available SPMD processes. Define the vector distribution $\pi_{\text{vec}} : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, p-1\}$; this map defines which process owns which element of the vectors used during the CG computation, e.g., element x_i is owned by process $\pi_{\text{vec}}(i)$. A vector element should be stored by at least the process that owns it, though more processes may store the same vector element as it may be needed during computation.

The derivation of π_{vec} depends on the distribution of the sparse matrix A. We take the most generic stance here, and stipulate that a distribution π_A of a sparse matrix A is a distribution of its nonzeroes, i.e.,

$$\pi_A : \ A \ni a_{ij} \to \{0, 1, \ldots, p-1\},$$

where we do *not* allow that the same nonzero element is distributed to multiple processes; though this, in principle, would be possible as a trade-off between (decreased) communication versus (increased) computational cost and memory use.

Load-balance is achieved by minimising

$$nz_p = \max_s |A_s| = \max_s |\pi_A^{-1}(s)|. \tag{5}$$

Now consider a single matrix row $a_{k:} = \{a_{ij} \in A \mid i = k\}$, A distributed according to π_A, and x, y distributed according to π_{vec}.

If all nonzeroes in $a_{k:}$ are owned by the same process s and $\pi_{\text{vec}}(i) = s$, then no communication would be necessary regarding updating y_i. Likewise, if all nonzeroes in the jth column $a_{:j}$ of A are owned by a single process s and $\pi_{\text{vec}}(j) = s$, then no communication would be necessary when retrieving the necessary element from x. If, however, during any of fan-out, $\lambda_{a_{:j}}$ processes have elements in the same column $a_{:j}$, then $\lambda_{a_{:j}} - 1$ values have to be communicated by process $\pi_{\text{vec}}(j)$. The same holds during fan-in for rows of A and their appropriate values for $\lambda_{a_{:j}}$.

Figure 2 illustrates this relationship between distribution and communication. The phase where input vector elements are distributed from their respective owners to the processes that require

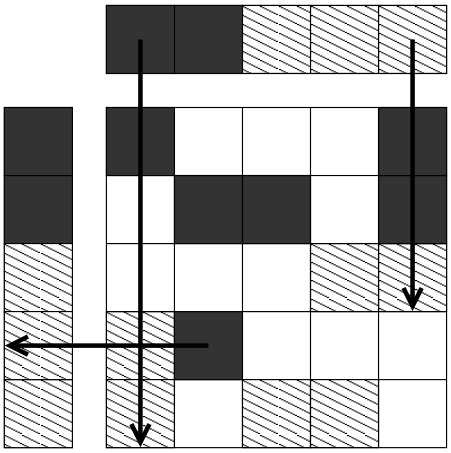

FIGURE 2. An example of a distributed sparse matrix and a matching vector distribution. and its effect on communication. Every matrix nonzero and vector element has been shaded differently to show to which out of two processes it is distributed. An arrow shows those cases where the connectivity λ_k corresponding to the kth row or kth column is larger than one, thus resulting in communication. In practice, values for λ_k can vary from 1 (no communication) to p (one-to-all or all-to-one communication) messages for row or column k.

them for local SpMV multiplication, i.e., the vertical arrows in Figure 2, is called the *fan-out* phase. The phase where y is updated according to remotely sent contributions, depicted by the horizontal arrow, is called the *fan-in* phase.

Fan-out and fan-in communication are naturally modelled using hypergraphs, where nonzeroes are vertices and rows $a_{i:}$ and columns $a_{:j}$ form hyperedges. Vertex-based hypergraph partitioning will then lead to a partitioning π_A, while the connectivity of each hyperedge correspond exactly to the $\lambda_{a_{k:}}$ and $\lambda_{a_{:k}}$ introduced above. The number $\mu_{\lambda-1}$ of vector elements communicated during SpMV multiplication depends only on these values: under the constraint that $\pi_{\text{vec}}(j) \in \pi_A^{-1}(a_{:j})$,

$$\mu_{\lambda-1} = \left[\sum_{a_{k:} \in A} \lambda_{a_{k:}} - 1 \right] + \left[\sum_{a_{:k} \in A} \lambda_{a_{:k}} - 1 \right]. \tag{6}$$

This measure is called the $\lambda - 1$ metric or connectivity-1 metric [24]. The corresponding algorithm for parallel SpMV multiplication is shown in Algorithm 5.

While minimising $\mu_{\lambda-1}$ minimises communication volume, it does not minimise the BSP measure of communication: the h-relations of the fan-out (superstep 0) and fan-in (superstep 1) in Algorithm 5. In the worst case, the h-relation is completely unbalanced with one process sending out all values of x and the same process receiving all contributions to y in which case h equals $\mu_{\lambda-1}$; the true cost is bounded by $\mu_{\lambda-1}$.

5.2. **Sparse matrix partitioning.** The cost model of SpMV multiplication described above, is the two-dimensional fine-grain hypergraph model by Çatalyürek and Aykanat [8]. Several other models exists that group together individual nonzeroes into single vertices, thus leading to smaller and easier-to-partition hypergraphs, at the loss of generality of the partitions they can model. Early examples are the 1D row-net model and the 1D column-net model, also by Çatalyürek and Aykanat [7]; there, nonzeroes are grouped by the column, resp., row they reside in. Partitioning thus proceeds at the granularity of matrix columns or rows, and not at the granularity of individual nonzeroes, thus reducing the number of vertices from nz to n only.

The generality of the fine-grain model and lower cost of partitioning of the row-net and column-net models were later combined in the Mondriaan algorithm by Vastenhouw and Bisseling [32], which allows switching hypergraph models during the partitioning process. Bisseling et al. found

Algorithm 5 Parallel distributed sparse matrix–vector multiplication $y = Ax$

Arguments:

in	`unsigned int`	s	local process ID
in	`unsigned int`	p	total number of processes
in	`double*`	x	local part of the input vector x
out	`double*`	y	local part of the output vector y

bsp_spmv:

1: **for all** a_{ij} for which $\pi_A(a_{ij}) = s$ **do**
2: **if** $\pi_{\text{vec}}(j) \neq s$ **then**
3: **bsp_get** x_j from process $\pi_{\text{vec}}(j)$
4: **bsp_sync** {end superstep 0}
5: call process-local sequential SpMV multiplication
6: **for all** a_{ij} for which $\pi_A(a_{ij}) = s$ **do**
7: **if** $\pi_{\text{vec}}(i) \neq s$ **then**
8: **bsp_send** (i, y_i) to process $\pi_{\text{vec}}(i)$
9: **bsp_sync** {end superstep 1}
10: **for all** received pairs (i, α) **do**
11: add α to y_i

its effectiveness to compare favourably to the earlier 1D and 2D models, and also show that minimising communication volume generally indeed leads to better performing distributed SpMV multiplication [4]. A method where nonzeroes can be grouped *partially* according to which row or column they reside in, dubbed the medium grain model, has recently been proposed by Pelt and Bisseling [28]. Note that by use of hypergraphs communication volume is modelled exactly; this is not the case for graph-based partitioning by software such as Metis [23].

We shall use the Mondriaan sparse matrix partitioning software using this new medium grain model. Like other state-of-the-art methods such as Zoltan [12], hMetis [22], or Scotch [27], Mondriaan relies on recursive multi-level bi-partitioning to create a p-partitioning of nonzeroes of A. Bi-partitioning is an NP-complete problem, indicating the need for heuristics to achieve reasonable run-times [24, Ch. 6.1.2]. In multi-level settings, partitioners work as follows:

(1) coarsen the hypergraph, i.e., reduce the number of vertices and hyperedges while attempting to retain the overall connection structure,
(2) recursively call this scheme, *or*, when small enough, (randomly) construct a coarse hypergraph partitioning,
(3) un-coarsen the hypergraph while refining the partitioning through, e.g., local search methods such as Kernighan-Lin [14], minimising communication (Equation 6) under constraint of load-balance (minimising Equation 5).

5.3. **Vector distribution.** For the communication volume during SpMV multiplication to equal $\mu_{\lambda-1}$, the vector distribution is assumed to have for any k and $\pi_{\text{vec}}(k) = s$, that $\exists\ a_{ik}$ such that $\pi_A(a_{ik}) = s$ and $\exists\ a_{kj}$ s.t. $\pi_A(a_{kj}) = s$. Other than that restriction, the choice of π_{vec} is free and defines the communication balance of the fan-out and fan-in supersteps.

Consider process s and write the number of messages sent during fan-out by $t_{\text{fanout},s,p}$, and the number of received messages by $r_{\text{fanout},s,p}$. Then

$$\begin{aligned} t_{\text{fan-out},s,p} &= \sum_{j:\ \pi_{\text{vec}}(j)=s} \left(\lambda_{a_{:j}} - 1\right), \\ r_{\text{fan-out},s,p} &= |\{j \in I \mid \exists a_{ij},\ \pi_A(a_{ij}) = s,\ \pi_{\text{vec}}(i) \neq s\}|. \end{aligned}$$

Due to symmetry of A, the $h_{\text{spmv,p}}$-relations of the fan-in and fan-out stages of the SpMV multiplication are equal, since $t_{\text{fan-in},s,p} = r_{\text{fan-out},s,p}$ and $r_{\text{fan-in},s,p} = t_{\text{fan-out},s,p}$. We write

$$h_{\text{spmv,p}} = \max\{\max_s t_{\text{fan-out},s,p}, \max_s r_{\text{fan-out},s,p}\}.$$

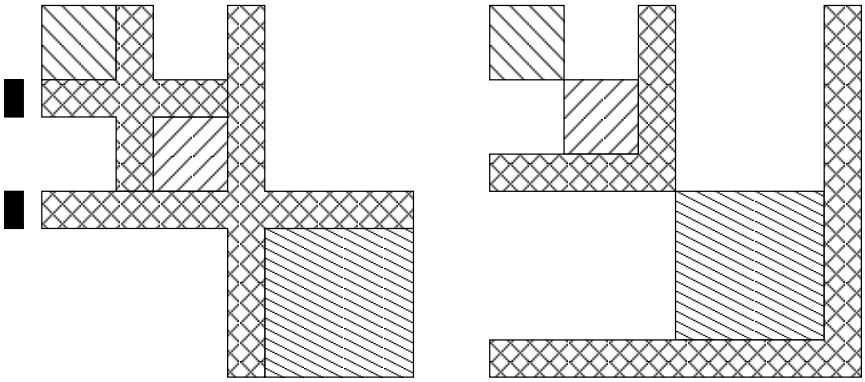

FIGURE 3. The Separated Block Diagonal (SBD, left) and the Bordered Block Diagonal (BBD, right) forms of a sparse matrix. Each shaded block corresponds to a sparse submatrix; white spaces in the matrix correspond to zero entries. Both forms correspond to a 3-partitioning, with diagonal and anti-diagonal pattern shades corresponding to nonzeroes in uncut rows and columns, and the cross-shaded area corresponding to nonzeroes that require communication during multiplication. The remaining matrix blocks are said to be part of the *separator cross* (left) or part of the matrix *border* (right). The solid bars on the left side of the matrix in SBD form accentuate the rows corresponding to the separator cross.

One algorithm to derive a good π_{vec} starts out with an empty set of locally owned vector elements, resulting in $t_{\text{fan-out},s,p} = 0$ and $r_{\text{fan-out},s,p} = n_A$, with n_A the number of columns in $\pi_A^{-1}(s)$. Adding a previously unassigned vector element into the set of locally owned elements, $r_{\text{fan-out},s,p}$ decreases while $t_{\text{fan-out},s,p}$ increases; this processes is repeated until $r_{\text{fan-out},s,p} \leq t_{\text{fan-out},s,p}$, which is a local lower bound on the fan-out h-relation. Processes with the highest local lower bounds get assigned the columns with the lowest connectivities. For full details, see Bisseling and Meesen [5].

This method is implemented in the Mondriaan sparse matrix partitioning software, which is available freely[1]. With the quantities $\mu_{\lambda-1}$ and the h-relations actively minimised by Mondriaan under the constraint of load balancing, the communication and computation costs of multiplication are

$$\max_s [2nz_p + r_{\text{fan-in},s,p}] \text{ flops} + 2h_{\text{spmv},p}g + 2l. \tag{7}$$

5.4. **Sparse matrix reordering.** No matter which hypergraph model is chosen, hyperedges correspond to matrix rows or columns. Yzelman and Bisseling link the communication volume $\mu_{\lambda-1}$ to a strict upper bound on the number of cache misses during sequential SpMV multiplication [34] by classifying matrix rows and columns according to their connectivity: if the ith row $a_{i:}$ has $\lambda_{a_{i:}} > 1$ then that row is *cut*. Cut rows can be grouped together and permuted to either the start, middle, or end of the matrix, and applying this reordering recursively during bipartitioning, and on the columns of A as well, induces cache-oblivious behaviour when properly exploiting the block structure during sequential SpMV multiplication [35]. Figure 3 illustrates this for $p = 3$, i.e., after two bipartitionings. The Separated Block Diagonal (SBD, left) form induces cache-friendly behaviour, while the Bordered Block Diagonal (BBD, right) form is useful for fill-in reduction during operations such as LU factorisation [16, 21]. Reordering to cache-obliviously increase performance has limited applicability due the pre-processing costs incurred by modern partitioners [35, 38]; we use reordering to achieve further data compression instead.

Let the process-local matrix $A_s = \pi_A^{-1}(s)$ be split disjoint matrices L_s and S_s such that $\forall a_{ij} \in L_s$, $\pi_{\text{vec}}(i) = \pi_{\text{vec}}(j) = s$; i.e., processing $y = L_s x$ is a process-local operation, while for the remainder nonzeroes S_s fan-out and fan-in are required. Note that the L_s correspond to the p

[1] via `http://www.math.uu.nl/bisseling/Mondriaan`.

square blocks with (anti)diagonal shading in Figure 3, while S_s corresponds to the remaining cross-shaped separator blocks when in SBD form.

Let $n_{A,s}$ be the number of rows in A_s, and let $n_{\text{vec},s} = |\pi_{\text{vec}}^{-1}(s)$ be the number of local vector elements. Note that $n_{\text{vec},s} \leq n_{A,s}$. Let $n_{S,s}$ be the number of rows for which $\lambda_{a_{i:}} > 1$ and $\pi_{\text{vec}}^{-1}(i) = s$. The maximum values of the above quantities are useful in bounding the cost of process-local data movement during multiplication: let $n_A = \max_s n_{A,s}$ and define n_{vec} and n_S similarly. Note that n_S is less or equal to the size of the horizontally oriented blocks of the separator cross as indicated by the solid bars on the far left in Figure 3.

The separation of L_s and S_s allows (1) the overlap of local computation $y = L_s x$ with the fan-out stage, as well as (2) compression of the information of communication patterns by use of reordering. Without reordering, executing the fan-out and fan-in communication patterns requires $\max_s (n_{A_s} - n_{\text{vec},s}) \leq n_S$ bytes of metadata per process. Additional compression may be obtained by organising the communication pattern as Q_s quadlets $(t, l, r, len)_{s,k}$, $0 \leq k < Q_s$, where reordering of the cut rows and columns is done to minimise Q_s [38]. During fan-out, a process s issues a *bsp_get* to request *len* consecutive elements from $x[r]$ at process t and store them at $x[l]$. During fan-in, a *bsp_send* transmits *len* consecutive elements from the local $y[l]$ to $y[r]$ at process t instead. We use SBD reordering for the compressed storage of the communication metadata only, resulting in a storage requirement of $b_{\text{ind}} \min\{b_{\text{ind}} Q_s, n_{S,s}\}$ bytes, and results, with Equation 7, in the following bound on the full cost of SpMV multiplication:

$$\begin{aligned} T_{\text{spmv},p} \quad \leq \quad & 2nz_p \text{ flops} + \max_s r_{\text{fan-in},s,p} \text{ flops} + \\ & 3nz_p b_{\text{val}} \text{ bytes} + (nz_p + n_A) b_{\text{ind}} \text{ bytes} + \\ & 2n_S(2b_{\text{val}} + 1) \text{ bytes} + 2h_{\text{spmv},p} g + 2l. \end{aligned} \tag{8}$$

This bound depends on the quality of the partitioning, and the difference with the true performance with regards to data movement increases with the quality of reordering.

5.5. **Modification of BLAS1 kernels.** The local matrix size $n_{A,s}$ is typically larger than the locally owned number of vector elements $n_{\text{vec},s}$. BLAS1 vector updates at a given process s, however, should only operate on the elements in π_{vec}^{-1}, i.e., on all elements corresponding to A_s, but only on several elements from the separator matrix S_s. Selecting the correct vector elements to prevent performing needless computations on the remaining vector elements can be done using $n_{S,s}$ bytes of metadata, but by using reordering similarly as with fan-out and fan-in, this metadata can be compressed by using P_s pairs $(p, q)_{s,k}$, $0 \leq k < P_s$ to encode the start and end locations of consecutive strings of local vector elements in S_s; the overhead of executing BLAS1 functions thus is

$$T_{\text{blas1}} = \min\{2b_{\text{ind}} P_s, n_{S,s}\} \text{ bytes}. \tag{9}$$

To ease programming, we define a function called *blas1* that takes a C++11 lambda function and maps that function unto all vector elements in π_{vec}^{-1}, thus ignoring the vector elements not owned by the process executing the BLAS1 operation; see the final parallel CG algorithm in Algorithm 6 for its example use.

An parallel inner product calculation of two vectors, called *bsp_dot* proceeds as a process-local inner product calculation $\alpha_s = x'y$ where x, y are the $n_{\text{vec},s}$ locally available vector elements. This is followed by a *bsp_allreduce* which broadcasts all local contributions to every other process, so that each process can proceed with calculating the global dot product afterwards:

1: **for** $k = 0$ **to** $p = 1$ **do**
2: **bsp_put** α_s into process k
3: **bsp_sync**
4: **return** $\sum_{k=0}^{p-1} \alpha_k$

The cost of this function is

$$T_{\text{allreduce},p} = p \text{ flops} + pb_{\text{val}} \text{ bytes} + pb_{\text{val}} g + l \tag{10}$$

so the cost of the full bsp_dot function becomes

$$T_{\mathrm{dot},p} = n_{\mathrm{vec}}\,(2 \text{ flops} + 2b_{\mathrm{val}} \text{ bytes}) + T_{\mathrm{allreduce},p}. \tag{11}$$

5.6. **Distributed-memory CG algorithm and its cost.** Algorithm 6 sketches the distributed-memory CG algorithm. Every vector depicted in that algorithm only relates to the parts of the global vector that are locally stored at the process (s, p); there is no process with a global view of all vector elements. The algorithm relies on the auxiliary functions described in the previous subsections and their costs as already derived. Ignoring the algorithm lead-in on lines 1–7 as these are lower-order terms only, the following total cost for the BSP CG algorithm can be derived:

$$\begin{aligned} T_{\mathrm{CG},p}/max \;=\; & T_{\mathrm{spmv},p} + T_{\mathrm{dot},p} + T_{\mathrm{allreduce},p} + 2T_{\mathrm{blas1},p} + \\ & 8n_S \text{ flops} + n_A b_{\mathrm{val}} \text{ bytes} + \\ & 8n_{\mathrm{vec}} b_{\mathrm{val}} \text{ bytes}. \end{aligned} \tag{12}$$

Algorithm 6 CG in C++ using BSP

Arguments:

in	`unsigned int`	s	local process ID
in	`unsigned int`	p	total number of processes
in	`size_t`	n	local complete vector size $n_{A,s}$
in	`size_t`	max	maximum iterations
in	`double`	tol	accepted tolerance
in	`double*`	b	local part of the RHS b
out	`double*`	x	local part of the solution x

bsp_cg:

```
mv(e, x);
sigma = 0.0;
blas1([&]( const size_t i ) {
      u[i] = r[i] = b[i] - e[i];
      sigma += r[i] * r[i];
} );
sigma = bsp_allreduce(sigma);
for( size_t iteration = 0; iteration < max;
   ++iteration ) {
     std::fill_n(e, n, 0);
     bsp_mv(e, u);
     bsp_dot(omega, u, e);
     alpha = sigma/omega;
     omega = 0.0;
     blas1([&]( const size_t i ) {
          x[i] += u[i] * alpha;
          r[i] -= e[i] * alpha;
          omega += r[i] * r[i];
     } );
     omega = bsp_allreduce(omega);
     if(std::sqrt(omega) < tol ) break;
     beta = omega/sigma;
     blas1([&]( const size_t i ) {
          u[i] = r[i] + u[i] * beta;
     } );
     sigma = omega;
}
```

6. Experiments

To demonstrate this approach indeed attains high performance in practice, the resulting BSP CG algorithm was implemented in C++11 using MulticoreBSP [37] and the Sparse Library[2] [34, 36, 35, 38]. We compare the results thus obtained against the Blaze CG implementation from Algorithm 3. All software is compiled using GCC 5.2.0 and is executed on the a two-socket Intel Xeon E5-2690 v2 which has 76.8 GByte/s of theoretical peak bandwidth available. Each processor consists of 10 cores that share a 25 MB L3 cache, and each core supports two hardware hyperthreads. Unlike Blaze, Eigen does not support parallel sparse computations nor parallel BLAS1, hence only Blaze is used in parallel performance comparisons.

We use two of the largest symmetric positive definite matrices available from the University of Florida sparse matrix collection [11]: (1) *G3_circuit* of size 1 585 478 with 7 660 826 nonzeroes coming from a circuit simulation application, and (2) *thermal2* of size 1 228 045 with 8 580 313 nonzeroes coming from a thermal steady-state FEM simulation. The thermal_2 matrix additionally has a right-hand side vector b available which ensures we are solving a real-world problem. For the G3_circuit matrix the 1-vector is taken as right-hand side. Both matrices are reasonably small as corresponding vectors take around 12 MB, meaning that four vectors fit entirely into cache. Both matrices are also reasonably structured, naturally inducing reasonable cache behaviour during SpMV multiplication; the matrix collection did not contain any fully unstructured SPD matrices of comparable size.

Both matrices were pre-processed using Mondriaan 4.0 to create distributions for $p = \{10, 20, 40\}$ with allowed load imbalance $\epsilon = 0.01$ using the medium grain hypergraph model. The settings *SquareMatrix_DistributeVectorsEqual* and *EnforceSymmetricPermutation* were set to *yes*, and the *Permute* option was set to *SBD*. Output was written to a single Extended Matrix-Market file which is directly parsed by the BSP CG program. All CG algorithms are run with $max = 10\ 000$ and $tol = 10^{-15}$, and all implementations measure the time taken for each of the iterations as well as the total time taken, the latter thus including the overhead of per-iteration timing. All found solutions were successfully verified by recalculating Ax and calculating the inf-norm and the 2-norm.

Results. For the thermal2 matrix, Blaze converges in 9 588, Eigen in 9 591, and BSP with $p = 1$ in 9 561 iterations. The number of iterations are different because computations are executed in different orders across the different algorithms; this number also changes with p. For G3_Circuit all methods use the maximum 10 000 iterations, which enables fair timing as all algorithms then are guaranteed to perform the same amount of computational operations. The results per iteration are summarised in Figure 4. Note that the BSP algorithm with $p = 1$ incurs overhead from parallel constructs while those for Blaze and Eigen do not.

We measure several parallel results on the shared-memory machine described above. The Blaze CG algorithm is run using $p = 20$ and $p = 40$ threads. Its BSP counterpart is run in three configurations: 1x20: one socket, with hyperthreading, 2x10: two sockets, no hyperthreading, and 2x20: two sockets, with hyperthreading. The parallel results per iteration are shown in Figures 5 and 6. The time taken for the complete parallel CG algorithm to complete are reported in Table 1. The performance gain of BSP CG over Blaze CG are 2.9x and 2.8x for the two matrices tested; the absolute attained speedups versus sequential Blaze are 10.5x and 8.5x.

Discussion. These results prove the scalability of the BSP CG and validate the approach leads to high-performance algorithms. By inspecting the per-iteration timings, information on performance stability has also become visible; for the sequential algorithm on G3_circuit, both Eigen and Blaze report highly varying performances between iterations, while BSP for $p = 1$ reports much more stable performance with sample standard deviations of one order of magnitude less. For thermal2, the respective sample standard deviations are still higher than that of BSP CG but only up to factors 2.3x (Blaze) and 2.5x (Eigen). The various sequential sparse matrix optimisations used thus lead to more stable performance.

[2]both are freely available from `www.multicorebsp.com` and `albert-jan.yzelman.net/software/#SL`, respectively.

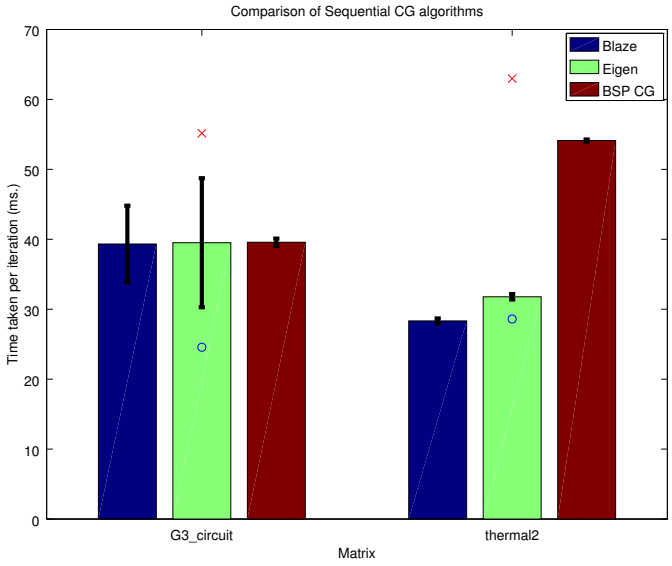

FIGURE 4. Comparison of sequential CG algorithms. The bar plot indicate the average time of an iteration, while the crosses indicate the time taken for the slowest iteration and the circles indicate that of the fastest iteration measured. The error bars centred at each bar have magnitude equal to the sample standard deviation measured in ms. across all iterations.

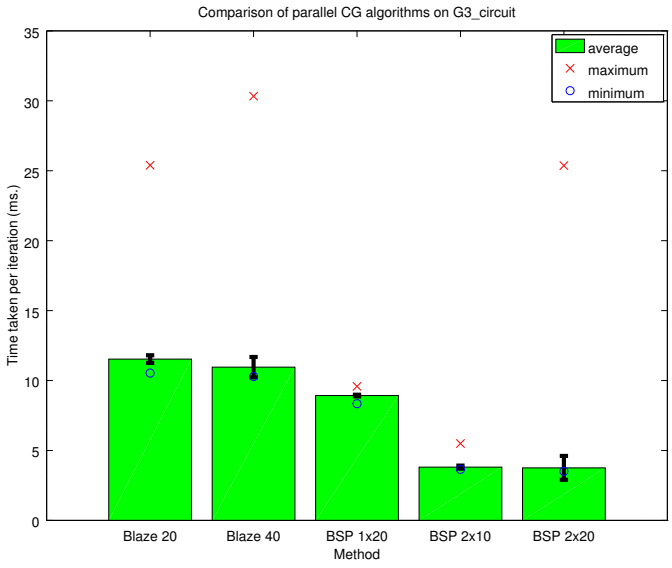

FIGURE 5. Comparison of parallel CG algorithms, G3_circuit

The parallel results for Blaze remain unstable when compared to on-socket BSP (1x20) and to two-socket BSP without hyperthreading (2x10); using hyperthreads on both sockets clearly causes performance instabilities to arise, for Blaze but also for BSP CG. The result on thermal2, where BSP 2x20 loses performance compared to BSP 2x10, indicates that the BSP CG algorithm is indeed bandwidth-bound as doubling the number of threads, and thus increasing the number of outstanding memory requests per cycle, failed to increase performance. On the G3_circuit matrix

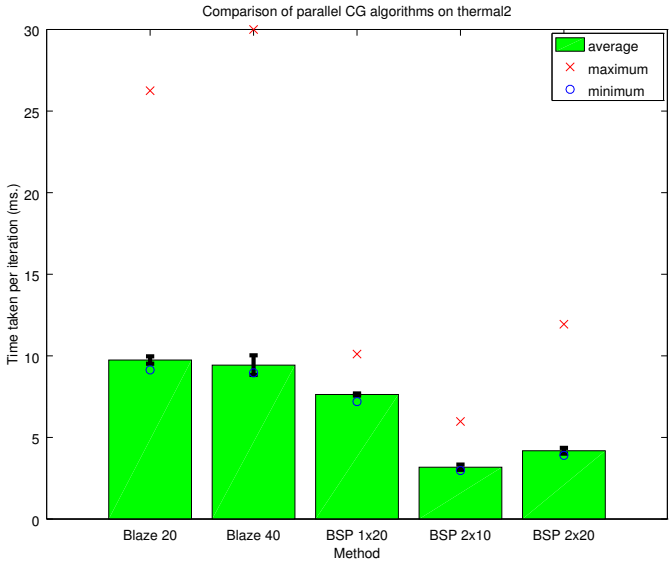

FIGURE 6. Comparison of parallel CG algorithms, thermal2

	G3_circuit	thermal2
Seq. Blaze	393.3 -	271.6 -
Blaze 40	109.6 (3.6x)	90.43 (3.0x)
BSP 2x10	38.15 (10.3x)	31.79 (8.5x)
BSP 2x20	37.61 (10.5x)	41.88 (6.5x)

TABLE 1. Selected timings for a complete CG algorithm run, in seconds. Speedups relative to sequential Blaze CG are in parenthesis.

BSP 2x20 did cause a small increase in performance, indicating local memory access latencies play a larger role than for thermal2. Since the sample standard deviation of iteration time increased about eight times, however, not using hyperthreads seems preferable.

7. CONCLUSION

We constructed a parallel CG algorithm based on many recent advances in sparse matrix computations. We provided a worst-case BSP cost analysis of the resulting algorithm, and expressed local compute times not only in the amount of floating point operations required, but also in terms of local data movement, which is the main cost factor in bandwidth-bound computations such as SpMV multiplications and sparse CG. This paper has provided a baseline analysis for a basic CG algorithm for which, the author hopes, improved methods will demonstrate improved bounds using similar metrics as used for analysis here.

Experiments have indicated the approaches used have led to a scalable, stable, and high performance CG algorithm that compares favourably against a CG algorithm written in a state-of-the-art numerical template library. Absolute speedups above a factor 10 on a 24-core machine have been observed, which is not trivial to achieve on problems bound by the performance of the SpMV multiplication kernel, and translate to a relative improvement of almost 3x over a non-distributing approach.

7.1. **Future work.** One bound of the current BSP CG algorithm that can be reduced immediately is the $4l$ cost per iteration. To lower this cost, computations can be reordered such that dot-products arise just before the SpMV product which saves the bottleneck of doing an allreduce,

effectively hiding the latency of a dot product behind that of the SpMV multiplication. Such a change is not only an algorithmic, but mathematical as well, and impacts the numerical properties as is known quite well in literature [10, 15]. This type of algorithmic restructuring is reminiscent of s-step methods where iterative methods are reworked to operate on matrices of rank s instead of on single vectors, which has nice parallel properties as the amount of work per superstep increases dramatically, again at the cost of less stable numerics [9]. Block CG also rewrites vector updates to higher-rank ones, but does not restructure the CG method as the goal is to solve for multiple right-hand sides [26]; the work per superstep has increased but the arithmetic intensity remains unchanged since the number of vectors used has multiplied with the the increase in work.

Other improvements are the use of tree-based reductions; though theoretically optimal in the BSP model, tree-based reductions are not practical due to the relatively large values for l on contemporary architectures. Since all presented methods rely on distributed-memory techniques, the BSP CG can scale out to multi-node clusters as well. It is worthwhile to investigate its performance on large-scale clusters, and to compare it against libraries such as Trilinos (i.e., Epetra and Aztec) [19] and PETSc [2]. This, however, also requires larger problems than those currently available in the University of Florida sparse matrix collection, where preferably also unstructured SPD matrices, would be of interest.

The techniques demonstrated here can furthermore be adapted to other iterative solvers, such as Block CG [26], IDR(s) [10], BiCGStab [30], s-Step methods [9], and many others. To increase the reach of the developed methods for sparse computation, further software engineering to increase the ease of integration is necessary. To exploit sparse matrix reordering for process-local efficient SpMV multiplication, the pre-processing cost of partitioning should be reduced as well.

References

[1] Z. Bai, D. Day, J. Demmel, M. Gu, J. Dongarra, A. Ruhe, and H. van der Vorst. Templates for linear algebra problems. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 115–140. Springer Berlin Heidelberg, 1995.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] R. H. Bisseling. *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford University Press, 2004.

[4] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 4, pages 95–127. CRC Press, Boca Raton, FL, USA, 2012.

[5] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[7] Ü. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Parallel algorithms for irregularly structured problems*, pages 75–86. Springer, 1996.

[8] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 1199–1204, April 2001.

[9] A. Chronopoulos and C. W. Gear. s-Step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[10] T. P. Collignon and M. B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 18(5):805–825, 2011.

[11] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[12] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[13] S. C. Eisenstat, M. Gursky, M. Schultz, and A. Sherman. Yale sparse matrix package ii: the nonsymmetric codes. Technical report, DTIC Document, 1977.

[14] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181. IEEE, 1982.

[15] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.
[16] L. Grigori, E. G. Boman, S. Donfack, and T. A. Davis. Hypergraph-based unsymmetric nested dissection ordering for sparse lu factorization. *SIAM Journal on Scientific Computing*, 32(6):3426–3446, 2010.
[17] G. Guennebaud and B. Jacob. Eigen. `http://eigen.tuxfamily.org/`, 2015. visited October 2015.
[18] G. Haase, M. Liebmann, and G. Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, 2007.
[19] M. Heroux and M. Sala. The design of trilinos. In J. Dongarra, K. Madsen, and J. Waniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 3732 of *Lecture Notes in Computer Science*, pages 620–628. Springer Berlin Heidelberg, 2006.
[20] K. Iglberger. Blaze. `https://bitbucket.org/blaze-lib/blaze`, 2015. visited October 2015.
[21] R. Ionuţiu, J. Rommes, and W. H. A. Schilders. SparseRC: sparsity preserving model reduction for RC circuits with many terminals. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(12):1828–1841, 2011.
[22] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.
[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
[24] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Chichester, UK, 1990.
[25] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24(3-4):287–297, 1999.
[26] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications*, 29:293–322, 1980.
[27] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.
[28] D. M. Pelt and R. H. Bisseling. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 529–539. IEEE, 2014.
[29] Y. Saad. *Iterative methods for sparse linear systems*. Siam, 2003.
[30] G. L. G. Sleijpen, H. A. Van der Vorst, and D. R. Fokkema. BiCGstab(l) and other hybrid Bi-CG methods. *Numerical Algorithms*, 7(1):75–109, 1994.
[31] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[32] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.
[33] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.
[34] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.
[35] A. N. Yzelman and R. H. Bisseling. Two-dimensional cache-oblivious sparse matrix–vector multiplication. *Parallel Computing*, 37(12):806 – 819, 2011.
[36] A. N. Yzelman and R. H. Bisseling. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, editors, *Progress in Industrial Mathematics at ECMI 2010*, Mathematics in Industry, pages 627–633. Springer, Berlin, Germany, 2012.
[37] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. MulticoreBSP for C: a high-performance library for shared-memory parallel programming. *International Journal on Parallel Programming*, 42:619–642, 2014.
[38] A. N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix–vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, 2014.

Huawei Technologies France, 20 quai du Point du Jour, 92100 Boulogne-Billancourt, France
E-mail address: `albertjan.yzelman@huawei.com`
URL: `http://albert-jan.yzelman.net`