CWI

Centrum Wiskunde & Informatica

Sequential sparse matrix–vector multiplication and tomography

Jan-Willem Buurlage (CWI)

MasterMath: Parallel Computing (2018)

Sparse matrices

Sparse and dense matrices

- Sparse matrices are sparsely populated by nonzero elements.
- Dense matrices have mostly nonzeros.
- Sparse matrix computations save time: operations with zeros can be skipped or simplified; only the nonzeros must be handled.
- Sparse matrix computations also save memory: only the nonzero elements need to be stored (together with their location).

Sparse matrix example

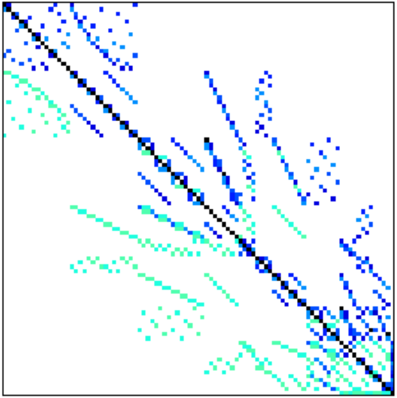


Figure 1: 93 rows and columns, 785 nonzeros, 8.4 nonzeros per row, 9.1% density.

Matrix statistics

- Number of nonzeros:

$$nz \equiv nz(A) \equiv |\{a_{ij} \mid 0 \leq i, j < n \text{ and } a_{ij} \neq 0\}|.$$

- Average number of nonzeros per row or column

$$c \equiv c(A) \equiv \frac{nz(A)}{n}.$$

- Nonzero density:

$$d \equiv d(A) \equiv \frac{nz(A)}{n^2}.$$

- A matrix sparse if $nz(A) \ll n^2$. Or, equivalently, when $c(A) \ll n$ or $d(A) \ll 1$.

Structure of sparse matrices

- If $a_{ij} \neq 0 \iff a_{ji} \neq 0$ then we say the matrix is structurally symmetric.
- This does not mean their values have to be equal. Computationally, the nonzero pattern is most important, not the the values.
- *Diagonal*, *tridiagonal* or more general banded matrices are also sparse.
- Sparse block matrices have a limited number of blocks, but these blocks can themselves be dense.

Irregular vs regular

- Regular algorithms have a computational cost that does not depend on the input. Examples of such algorithms you are familiar with are the FFT, LU, and dense matrix–matrix multiplication.
- Irregular algorithms, however, depend on the input. For sparse computations, they usually depend on the nonzero pattern of the matrix.
- Designing efficient irregular algorithms is a challenge. The ultimate goal is to make the algorithm as *efficient as possible for any input*.

Sequential algorithm

- An example of an irregular algorithm is the sparse matrix–vector product (SpMV).
- Given a sparse matrix A, and a dense vector v, compute $u \equiv Av$.

```
$\textbf{\textit{forall}}\ (i,j)\ \textit{such that}\ 0 \le i,j < n\ \textit{and}\ a_{ij} \neq 0\ \textbf{\textit{do}}$
    $u_i \leftarrow u_i + a_{ij} v_j$
```

- The nonzero test $a_{ij} \neq 0$ is never executed in practice. Rather, a sparse data structure is used, or the nonzeros are generated on-the-fly.

Applications of SpMV

- Sparse matrices are the rule rather than the exception.
- In many applications, variables are connected to only a few others, leading to sparse matrices.
- Sparse matrices occur in various application areas:
  - transition matrices in Markov models;
  - finite-element matrices in engineering;
  - linear programming matrices in optimisation;
  - weblink matrices in Google PageRank computation.
  - molecular dynamics
- The sequential computation is simple, but its parallelisation is a big challenge.

Power method

- Power methods are based on repeated application of A to some initial vector. It finds the dominant eigenvector.
- Let A be a transition matrix, and $\vec{x}$ a vector of state frequencies (i.e., x_i is the relative frequency of state i).
- Computing $A\vec{x}, A^2\vec{x}, A^3\vec{x}, \ldots$ until convergence, we find a vector satisfying $A\vec{x} = \vec{x}$. This is the steady state.

Iterative methods

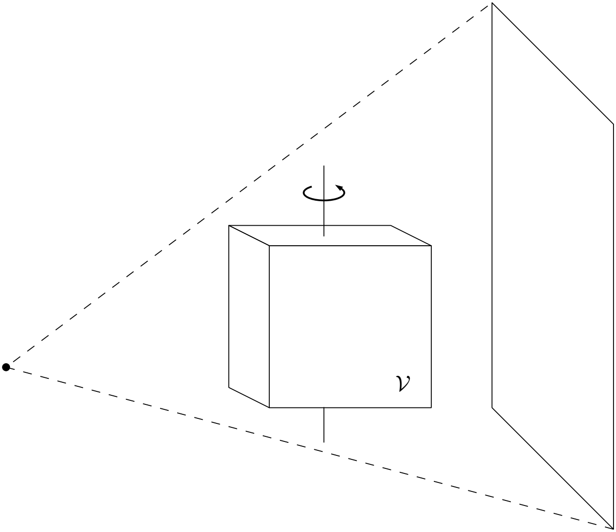
- More generally, sparse matrix—vector multiplication is the main computation step in iterative solution methods for linear systems or eigensystems.
- Iterative methods start with an initial guess x_0 and then successively improve the solution by finding better approximations x_k, $k = 1, 2, \ldots$, until the error is tolerable.
- Examples:
  - Linear systems $Ax = b$, solved by the conjugate gradient (CG) method or MINRES, GMRES, QMR, BiCG, Bi-CGSTAB, IDR, SOR, FOM, . . .
  - Eigensystems $Ax = \lambda x$ solved by the Lanczos method, Jacobi–Davidson, . . .

Tomography

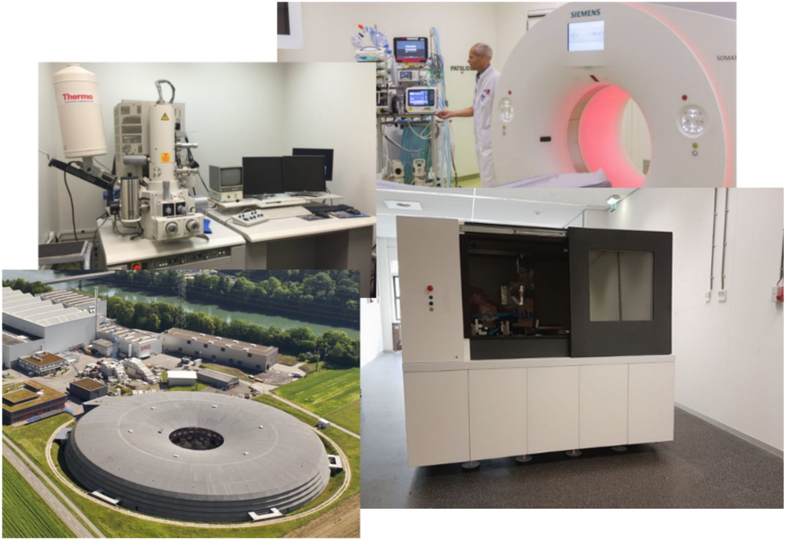
Introduction

- Tomography is a non-destructive imaging technique
- Penetrating rays (e.g. X-rays) are sent through an object from various angles, and their intensity is measured
- Leads to 2D projection images, from which a 3D volume is reconstructed

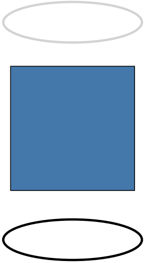
Example of tomographic measurement



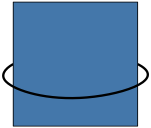
Applications of tomography



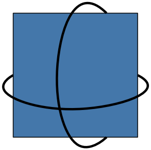
Acquisition geometries



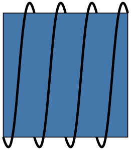
Laminography



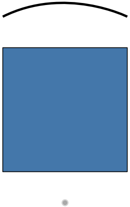
Single axis



Dual axis



Helical cone beam



Tomosynthesis

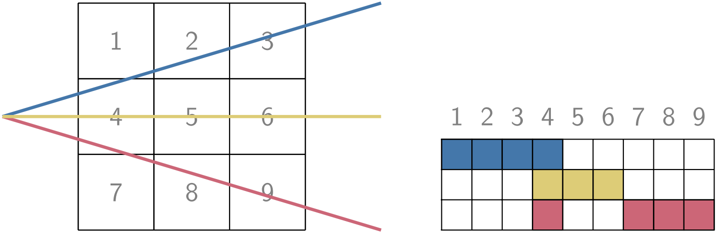
Tomographic reconstruction

- Projection matrix W, solve:

  $$W\vec{x} = \vec{b},$$

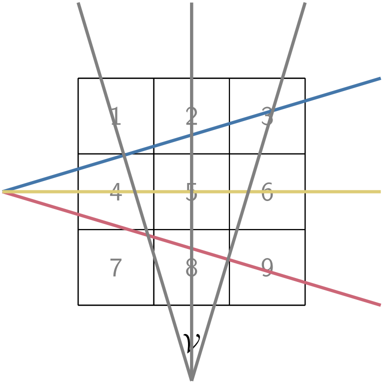
  with $\vec{x}$ the image, and $\vec{b}$ the projection data.
- The projection data consists of a series of 2D images (the 'X-ray shadows' of the object), and are measured. The 3D image is unknown, and is to be reconstructed.
- Rows correspond to rays, from a source to a detector pixel. Columns correspond to volume elements, or voxels.
- Intersections of rays with voxels, give rise to nonzeros in W.
- *Note:* W is sparse, for n voxels we have $\mathcal{O}(n^{1/3})$ nonzeros in each row.

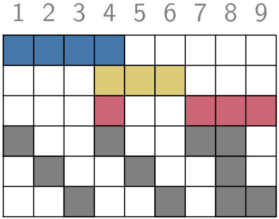
Example of Projection Matrix



$\mathcal{V}$ A

Example of Projection Matrix (II)





1	2	3	4	5	6	7	8	9
🟦	🟦	🟦	🟦					
			🟨	🟨	🟨			
			🟥			🟥	🟥	🟥
⬛			⬛			⬛	⬛	
	⬛			⬛			⬛	
		⬛			⬛		⬛	⬛

A

Spy plot of a projection matrix

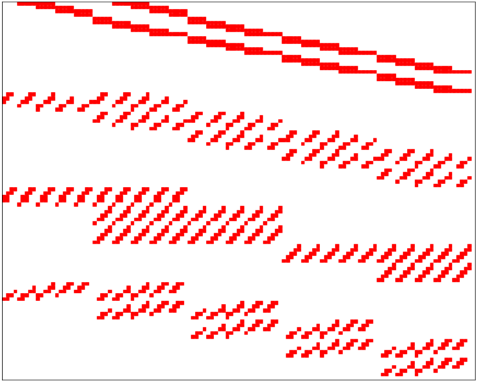


Figure 2: Parallel beam geometry matrix of size 100 × 125.

Large-scale tomography

- For tomographic reconstruction, the SpMVs $W\vec{x}$ and $W^T\vec{y}$ are the most expensive operations.
- 3D volumes with at least 1000^3 voxels. W then has $\geq \mathcal{O}(10^{12})$ entries $\Rightarrow$ TBs of data!
- Not stored explicitly, generated from the acquisition geometry.

Distributed-memory tomographic image reconstruction

- In tomography, we are reconstructing (i.e. compute based on projection data) a 3D image.
- If we want to do this in parallel, we can make each processor responsible for reconstructing only a (small) part of the volume.
- However, rays cross the the entire volume, coupling these parts together. *How do we partition the image to minimize the coupling?*

Geometric partitioning problem

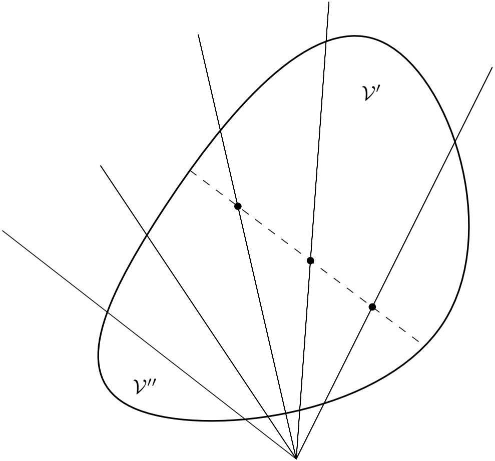
- We are given a cuboid, and a set of lines intersecting this cube.
- This cuboid is to be partitioned into p parts.
- A line crossing n parts has $n - 1$ cuts.
- What partitioning minimizes the total number of cuts?

Recursive bisectioning

- Idea: Split the volume into two subvolumes recursively.
- Straightforward to show that this can be done independently from previous splits.
- When splitting a subvolume, the effect on the overall communication volume is the same as that of the subproblem.

Interface intersection

- Communication volume equals number of lines through interface



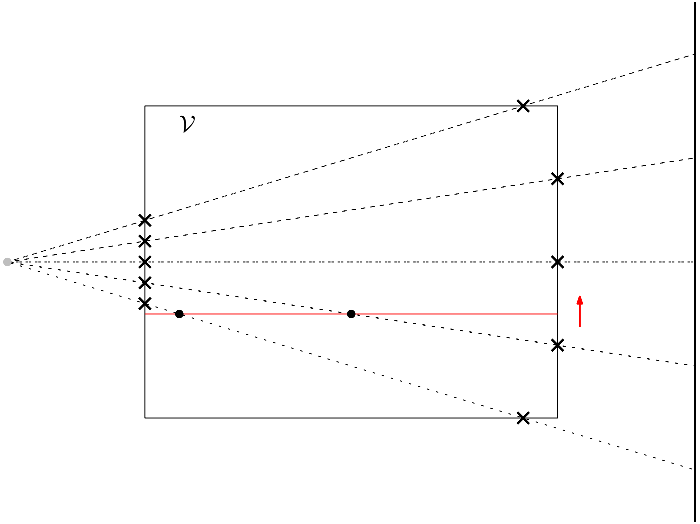
Bisectioning algorithm

- Choose the splitting interface with the minimum number of rays passing through it.
- Evenly distribute the workload
- Computational weight of a voxel is the number of lines crossing the voxel, i.e. number of nonzeros in its column
- Total computational weight of a subvolume can be computed using 3D prefix sums and application of inclusion-exclusion principle.

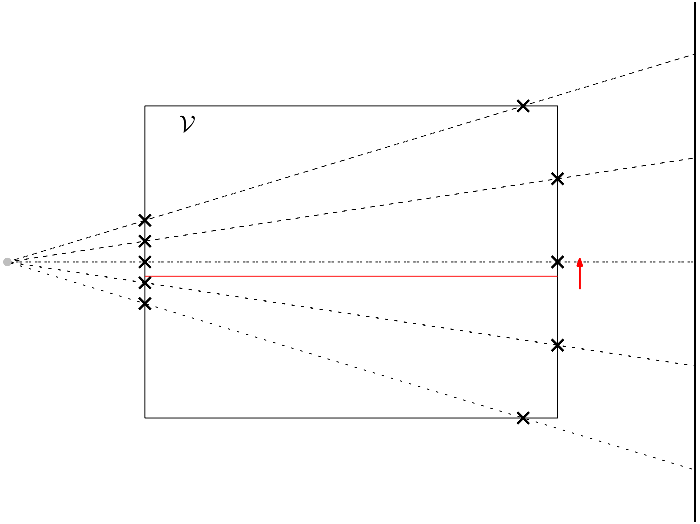
Plane sweep

- We sweep a candidate interface along the volume, and keep track of the current number of rays passing through it.
- Communication volume only changes at coordinates where a ray intersects the boundary!
- Compute intersections once, sweep for all three axes sorting the coordinates each time.

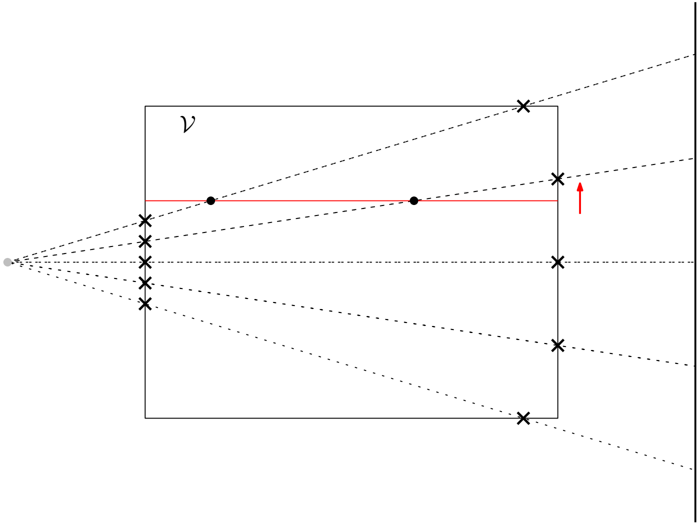
Example of plane sweep (I)



Example of plane sweep (II)



Example of plane sweep (III)

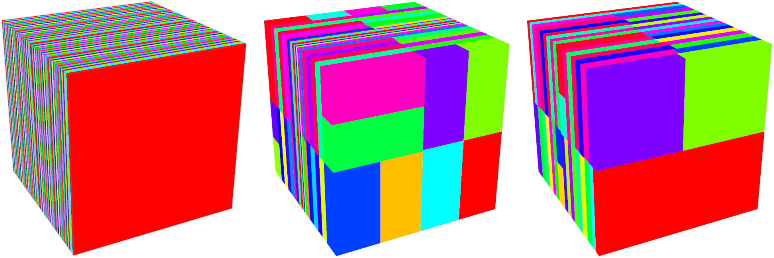


Results

- This gives us an efficient partitioning algorithm, runtime dominated by the sorting of coordinates: $\mathcal{O}(m \log(m))$.
- Geometric recursive coordinate partitioning (GRCB).
- Currently, slab partitionings of the volume along the rotation axis are used.

Partitioning results

- What constitutes a good partitioning depends heavily on the acquisition geometry.



- With a good partitioning, the amount of data that is communicated between processors is low.

Conclusion

- Sparse matrices are everywhere in scientific computing.
- Tomographic imaging is an important technique for science, medicine, cultural preservation and industry.
- Exploiting sparsity can save a lot of computational work. It requires, however, the design of specialized and irregular algorithms.
- Sequential sparse computations are relatively straightforward, but their parallelisation is a big challenge. Communication considerations often lead to interesting partitioning problems.

A geometric partitioning method for distributed tomographic reconstruction. Jan-Willem Buurlage, Rob Bisseling, Joost Batenburg. (Submitted to Parallel Computing)

CWI

Centrum Wiskunde & Informatica

Bulk: a Modern C++ BSP Interface

Jan-Willem Buurlage (CWI)

MasterMath: Parallel Computing (2018)

BSP today

- For high-performance computing on distributed-memory systems, BSP is still a (if not *the*) leading model.
- In the last 10 years or so, it has grown again in popularity. It has also found widespread use in industry (MapReduce / Pregel).
- BSP programming usually done using MPI or the various Apache projects (Hama, Giraph, Hadoop).

Google MapReduce

- *Standard example*: word count. The map takes a (file, content) pair, and emits (word, 1) pairs for each word in the content. The reduce function sums over all mapped pairs with the same word.
- The map and reduce are performed in parallel, and are both followed by communication and a bulk synchronization, which means MapReduce $\subset$ BSP![1]

[1] MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat (2004)

Google Pregel

BSP for graph processing, used by Google[2] and Facebook[3].

> *The high-level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model. Pregel computations consist of a sequence of iterations, called supersteps ... It can read messages sent to V in superstep S 1, send messages to other vertices that will be received at superstep $S + 1$ ...*

[2] Pregel: A System for Large-Scale Graph Processing – Malewicz et al. (2010)
[3] One Trillion Edges: Graph Processing at Facebook-Scale - Avery Ching et al (2015)

Modern BSP

- These frameworks are good for big data analytics, not for high-performance scientific computing.
- $\Longrightarrow$ Most scientific software still built on top of MPI.
- Modern programming languages have novel features (safety, abstractions) which can aid parallel programming.

BSP interfaces

- There are mature implementations of BSPlib for shared and distributed-memory systems[4].
- Many *Big Data* frameworks are based on (restricted) BSP programming, such as MapReduce (Apache Hadoop), Pregel (Apache Giraph) and so on.
- BSP interfaces that are not based on BSPlib include BSML and Apache Hama.

[4] e.g. Multicore BSP (for C) by Albert Jan Yzelman and BSPonMPI by Wijnand Suijlen

BSPlib

```c
#include <bsp.h>

int main() {
  bsp_begin(bsp_nprocs());
  int s = bsp_pid();
  int p = bsp_nprocs();
  printf("Hello World from processor %d / %d", s, p);
  bsp_end();

  return 0;
}
```

BSPlib: Registering and using variables

```
int x = 0;
bsp_push_reg(&x, sizeof(int));
bsp_sync();

int b = 3;
bsp_put((s + 1) % p, &b, &x, 0, sizeof(int));

int c = 0;
bsp_get((s + 1) % p, &x, 0, &c, sizeof(int));

bsp_pop_reg(&x);
bsp_sync();
```

BSPlib: Sending messages

```
int tagsize = sizeof(int);
bsp_set_tagsize(&tagsize);
bsp_sync();

int tag = 1;
int payload = 42 + s;
bsp_send((s + 1) % p, &tag, &payload, sizeof(int));
bsp_sync();
```

BSPlib: Receiving messages

```
int packets = 0;
int accum_bytes = 0;
bsp_qsize(&packets, &accum_bytes);

int payload_in = 0;
int payload_size = 0;
int tag_in = 0;
for (int i = 0; i < packets; ++i) {
    bsp_get_tag(&payload_size, &tag_in);
    bsp_move(&payload_in, sizeof(int));
    printf("payload: %i, tag: %i", payload_in, tag_in);
}
```

A modern BSP interface

- Modern programming languages focus on safety and zero-cost abstractions to increase programmer productivity, without sacrificing performance.
- A modern BSP interface should also have this focus. We want correct, safe and clear implementations of BSP programs without taking a performance hit.
- *Modern C++* has a large user base, is widely supported, with a good set of features and (support for) abstractions.

Bulk: A modern BSP interface

- Bulk is a modern BSPlib replacement.
- Focuses on memory safety, portability, code reuse, and ease of implementation of BSP algorithms.
- Flexible backend architecture. Bulk programs target shared, distributed, or hybrid memory systems.
- Support for various *algorithmic skeletons*, and utility features for logging, benchmarking, and reporting.

Bulk: Basics

- A BSP computer is captured in an `environment` (e.g. an MPI cluster, a multi-core processor or a many-core coprocessor).
- In an environment, an SPMD block can be spawned.
- The processors running this block form a parallel `world`, that can be used to communicate, and for obtaining information about the local process.

```
bulk::backend::environment env;
env.spawn(env.available_processors(), spmd);

void spmd(bulk::world& world) {
  world.log("Hello world from %d / %d\n",
            world.rank(),
            world.active_processors());
}
```

Bulk: Distributed variables (I)

- Registering and deregistering (`bsp_push_reg`) is replaced by *distributed variables*.

```
auto x = bulk::var<int>(world);
auto y = x(t).get();
x(t) = value;
```

- These variables are `var` objects. Their value is generally different on each processor.
- References to remote values are captured in `image` objects, and can be used for reading and writing.

Bulk: Distributed variables (II)

```
auto x = bulk::var<int>(world);
auto t = world.next_rank();
x(t) = 2 * world.rank();
world.sync();
// x now equals two times the previous ID

auto b = x(t).get();
world.sync();
// b.value() now equals two times the local ID
```

Bulk: Coarrays (I)

- Distributed variables work well for communicating single values.
- For communication based on (sub)arrays we have `coarray` objects, loosely inspired by Coarray Fortran.

```
auto xs = bulk::coarray<int>(world, 10);
xs(t)[5] = 3;
auto y = xs(t)[5].get();
```

- Images to remote subarrays of a coarray `xs`, are obtained as for variables by `xs(t)`, and can be used to access the remote array.

Bulk: Coarrays (II)

```
auto xs = bulk::coarray<int>(world, 4);
auto t = world.next_rank();
xs[0] = 1;
xs(t)[1] = 2 + world.rank();
xs(t)[{2, 4}] = {123, 321};
world.sync();
// xs is now [1, 2 + world.prev_rank(), 123, 321]
```

Bulk: Message passing queues (I)

- One-sided mailbox communication using message passing, which in Bulk is carried out using a queue. Greatly simplified compared to previous BSP interfaces, without losing power or flexibility.

```
// single integer, and zero or more reals
auto q1 = bulk::queue<int, float[]>(world);
// sending matrix nonzeros around (i, j, a_ij)
auto q2 = bulk::queue<int, int, float>(world);
```

- Message structure is defined in the construction of a queue: optionally attach tags, or define your own record structure.

BSPlib: Sending messages

```
int tagsize = sizeof(int);
bsp_set_tagsize(&tagsize);
bsp_sync();

int tag = 1;
int payload = 42 + s;
bsp_send((s + 1) % p, &tag, &payload, sizeof(int));
bsp_sync();
```

Bulk: Sending messages

```
auto q = bulk::queue<int, int>(world);
q(world.next_rank()).send(1, 42 + s);
world.sync();
```

BSPlib: Receiving messages

```
int packets = 0;
int accum_bytes = 0;
bsp_qsize(&packets, &accum_bytes);

int payload_in = 0;
int payload_size = 0;
int tag_in = 0;
for (int i = 0; i < packets; ++i) {
    bsp_get_tag(&payload_size, &tag_in);
    bsp_move(&payload_in, sizeof(int));
    printf("payload: %i, tag: %i", payload_in, tag_in);
}
```

Bulk: Receiving messages

```
for (auto [tag, content] : queue) {
    world.log("payload: %i, tag: %i", content, tag);
}
```

Bulk: Beyond tags

- In addition, Bulk supports sending arbitrary data either using custom structs, or by composing messages on the fly. For example, to send a 3D tensor element with indices and its value.

```
auto q = bulk::queue<int, int, int, float>(world);
q(world.next_rank()).send(1, 2, 3, 4.0f);
q(world.next_rank()).send(2, 3, 4, 5.0f);
world.sync();

for (auto [i, j, k, value] : queue) {
    world.log("element: A(%i, %i, %i) = %f", i, j, k, value);
}
```

- Multiple queues can be constructed, which eliminates a common use case for tags.

Bulk: Skeletons

```
// dot product
auto xs = bulk::coarray<int>(world, s);
auto ys = bulk::coarray<int>(world, s);
auto result = bulk::var<int>(world);
for (int i = 0; i < s; ++i) {
    result.value() += xs[i] * ys[i];
}
auto alpha = bulk::foldl(result,
    [](int& lhs, int rhs) { lhs += rhs; });

// finding global maximum
auto maxs = bulk::gather_all(world, max);
max = *std::max_element(maxs.begin(), maxs.end());
```

Bulk: Example application (I)

- In parallel regular sample sort, there are two communication steps.
  1. Broadcasting p equidistant samples of the sorted local array.
  2. Moving each element to the appropriate remote processor.

```
// Broadcast samples
auto samples = bulk::coarray<T>(world, p * p);
for (int t = 0; t < p; ++t)
    samples(t)[{s * p, (s + 1) * p}] = local_samples;
world.sync();

// Contribution from P(s) to P(t)
auto q = bulk::queue<int, T[]>(world);
for (int t = 0; t < p; ++t)
    q(t).send(block_sizes[t], blocks[t]);
world.sync();
```

Bulk: Word count

- The *word count* example (MapReduce) can be implemented in Bulk as follows. First the map phase:

```cpp
auto words = bulk::queue<std::string>(world);
if (s == 0) {
  auto f = std::fstream("examples/data/alice.txt");
  std::string word;
  while (f >> word) {
      words(hash(word) % p).send(word);
  }
}
world.sync();
```

Word count (II)

- Then the reduce phase:

```
auto counts = std::map<std::string, int>{};
for (auto word : words) {
    if (counts.find(word) != counts.end()) {
        counts[word]++;
    } else {
        counts[word] = 1;
    }
}
auto report = bulk::queue<std::string, int>(world);
for (auto [word, count] : counts) {
    report(0).send(word, count);
}
world.sync();
```

Bulk: Shared-memory results

Table 1: Speedups of parallel sort and parallel FFT compared to `std::sort` from libstdc++, and the sequential algorithm from FFTW 3.3.7, respectively.

	n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Sort	2^{20}	0.93	1.95	3.83	6.13	8.10	12.00
	2^{21}	1.01	2.08	4.11	7.28	10.15	15.31
	2^{22}	0.88	1.82	3.58	5.99	10.27	13.92
	2^{23}	0.97	1.90	3.63	6.19	11.99	16.22
	2^{24}	0.93	1.79	3.21	6.33	8.47	14.76
FFT	2^{23}	0.99	1.07	2.08	2.77	5.60	5.51
	2^{24}	1.00	1.26	2.14	3.07	5.68	6.08
	2^{25}	1.00	1.23	2.22	3.09	5.80	6.05
	2^{26}	0.99	1.24	2.01	3.28	5.48	5.97

Bulk: Shared-memory benchmarks

Table 2: The BSP parameters for MCBSP and the C++ thread backend for Bulk.

Method	r (GFLOP/s)	g (FLOPs/word)	l (FLOPs)
MCBSP (spinlock)	0.44	2.93	326
MCBSP (mutex)	0.44	2.86	10484
Bulk (spinlock) *new*	0.44	5.55	467
Bulk (mutex)	0.44	5.65	11702

Outlook

- Further performance improvements for the `thread` and the MPI backends.
- Implementing popular BSP algorithms to provide case studies as a learning tool for new Bulk users.
- Applications: tomography, imaging science, sparse linear algebra.
- Currently working on syntax/support for distributions: partitionings, multi-indexing, 2D/3D computations.

Bulk: Partitionings

```
auto phi = bulk::cyclic_partitioning<1>({size}, {p});
auto psi = bulk::cyclic_partitioning<2, 2>({n, n}, {M, N});
auto chi = bulk::block_partitioning<2, 2>({n, n}, {M, N});
// And: irregular, cartesian, tree, ...

// In LU decomposition: is a_kk assigned to us?
if (phi.owner({k, k}) == world.rank())
// What is the global index of local element (i, j)
phi.local_to_global({i, j}, {s, t})
// What is the size of my local data
phi.local_size(world.rank())
// What is my 'multi-index'?
auto [s, t] = bulk::unflatten<2>(phi.grid(), world.rank());
// What processor owns global element (i, j)?
phi.grid_owner({i, j})
```

Conclusion

- Modern interface for writing parallel programs, safer and clearer code
- Works together with other libraries because of generic containers and higher-level functions.
- Works across more (mixed!) platforms than other libraries.
- Open-source, MIT licensed. Documentation at `http://jwbuurlage.github.io/Bulk`. Current version: v1.1.0.

MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat (2004)

Pregel: A System for Large-Scale Graph Processing – Malewicz et al. (2010)

One Trillion Edges: Graph Processing at Facebook-Scale - Avery Ching et al (2015)

Buurlage JW., Bannink T., Bisseling R.H. (2018) Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs. Euro-Par 2018: Parallel Processing.