Centrum Wiskunde & Informatica

Bulk: a Modern C++ Interface for Bulk-Synchronous Parallel Programs

---

Jan-Willem Buurlage (CWI)
Tom Bannink (CWI)
Rob Bisseling (Utrecht University)

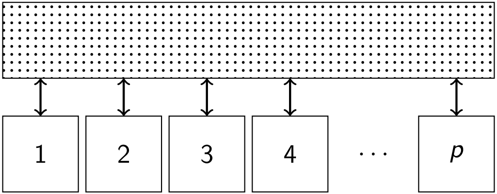
2018-08-29, Euro-Par 2018, Turin

Overview

- Introduction to BSP
- BSP programming interfaces
- Bulk
- Conclusion

BSP

- The BSP model provides a way to structure and analyze parallel computations.
- An (abstract) BSP computer has p processors, which all have access to a communication network.

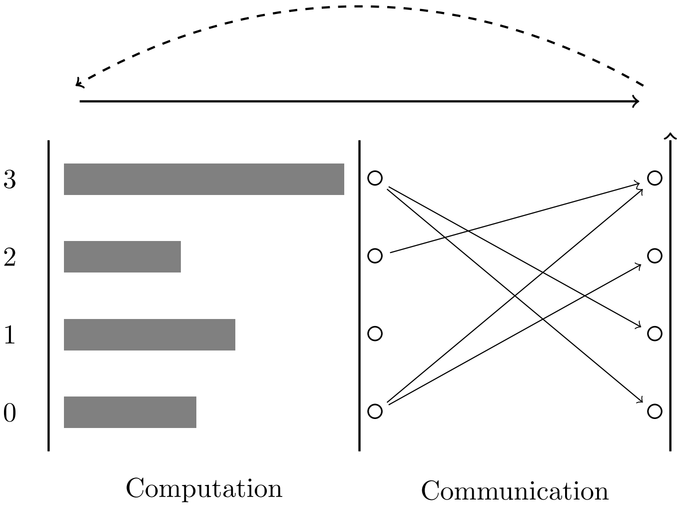


- Other parameters are the raw processing speed r, the communication time per data word g, and the latency l.
- The cost T of a BSP program is expressed in terms of the parameters (p, r, g, l).

BSP (II)

- A BSP program is structured in a number of supersteps.
- A superstep has a computation phase and a communication phase.
- After a superstep, a barrier (bulk) synchronization is performed. The next superstep begins after all communication has finished
- Each processor runs the same program, but on different data (SPMD).

BSP (Supersteps)



BSPlib

- The BSP model is a powerful abstraction for developing portable parallel algorithms.
- The BSPlib standard describes a collection of primitives which can be used for writing BSP programs.

```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 (II)

- In BSPlib, variables can be registered by their address. They can then be written to/read from remotely.

```
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();
```

Other BSP interfaces

- There are mature implementations of BSPlib for shared and distributed-memory systems[1].
- 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.

---

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

A modern BSP interface

- A focus of many modern (implementations of) programming languages is on safety and zero-cost abstractions that increase programmer productivity, without sacrificing performance.
- We think a modern BSP interface should also have this focus. We want correct, safe and clear implementations of BSP programs without taking a performance hit.
- For us, modern C++ is a good fit. Large user base, 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(), [](auto& world) {
  world.log("Hello world from %d / %d\n",
              world.rank(),
              world.active_processors());
});
```

Bulk: Distributed variables (I)

- Distributed 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.

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

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)

- For communication based on (sub)arrays we have `coarray` objects, loosely inspired by Coarray Fortran.
- 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.

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

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

```
// 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);
```

Bulk: Message passing queues (II)

```
// queue containing simple data
auto numbers = bulk::queue<int>(world);
numbers(t).send(1);
numbers(t).send(2);
world.sync();
for (auto value : numbers)
    world.log("%d", value);

// queue containing multiple components
auto index_tuples = bulk::queue<int, int, float>(world);
index_tuples(t).send({1, 2, 3.0f});
index_tuples(t).send({3, 4, 5.0f});
world.sync();
for (auto [i, j, k] : index_tuples)
    world.log("(%d, %d, %f)", i, j, k);
```

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

- 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: 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.
- Currently working on syntax/support for distributions: partitionings, multi-indexing, 2D/3D computations.
- Applications: tomography, imaging science, sparse linear algebra.

Conclusion

- Bulk is a modern BSP interface and library implementation.
- Many desirable features
  - Memory safety
  - Support for generic implementations of algorithms
  - Portability
  - Encapsulated state
  - ...
- Allows for clear and concise implementations of BSP algorithms. Furthermore, we show good scalability of BSP implementations of two $O(n \log n)$ algorithms, for which nearly all input data have to be communicated.
- The performance of Bulk is close to that of a state-of-the-art BSPlib implementation.
- Enables hybrid shared/distributed-memory programming with the efficiency of exploiting shared memory but without the pain of using two APIs (MPI+OpenMP).