Concepts and Models of Parallel and Data-centric Programming

BSP V (Bulk: Distributed Variables & Coarrays)

Lecture, Summer 2021

Simon Schwitanski
Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

High Performance Computing

Outline

0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. **Bulk-Synchronous Parallelism**
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. Motivation
b. BSP Computer
c. BSP Programming Model
d. BSP Cost Model
e. **Bulk Library**
   a. Introduction
   b. Data Distribution
   c. **Distributed Variables**
   d. **Coarrays**
   e. **Further Features**

Bulk Library: Distributed Variables

Distributed Variables

- *Distributed variables* enable communication between processors
  - `var` object captures a distributed variable
- Has to be created in the same superstep by each processor
- Each processor has a *local value* of that variable and can access the concrete values on remote processors as so called *remote values*
- Accessing a remote value of a distributed variable x
  - `bulk::future<Type> y = x(t).get()` (read value of `x` at processor `t`)
  - `x(t) = value` (write `value` to `x` at processor `t`)

Question: Why is the `get()` operation encapsulated in a future?

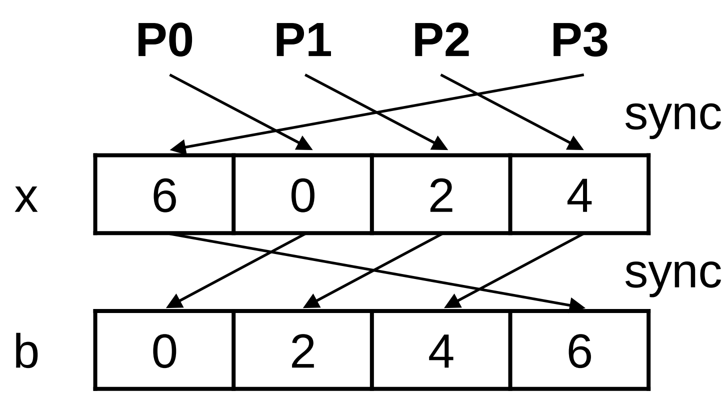
- Because the communication (reading / writing a value) is only guaranteed to be completed after a `world.sync()` operation!

Distributed Variables – Example

```
env.spawn(env.available_processors(), [](auto& world) {
    auto x = bulk::var<int>(world);
    auto t = world.next_rank();
    x(t) = 2 * world.rank();
    world.sync();
    // x now contains 2 * world.prev_rank() at the current rank
    bulk::future<int> b = x(t).get(); // get value of next rank
    world.sync();
    // b.value() now contains two times the local rank ID
});
```

- Note: All communication via distributed variables is nonblocking!
  - `x(t).get();` and `x(t) = value` do <u>not</u> wait until a value has been read / or written

Example for 4 processes:



Translating BSP Algorithm to Bulk – Distributed Variables (1)

```
env.spawn(env.available_processors(),
          [](auto& world) {
    auto s = world.rank();
    auto p = world.active_processors();

    // computation of local inner product
    // assume: x and y cyclic partitioned arrays
    double local_product = 0.0;
    for (int i = 0; i < local_size; i++)
        local_product += x.local(i) * y.local(i);

    // create distributed variable for each rank
    std::vector<bulk::var<double>> remote_products;
    for (int t = 0; t < p; t++)
        remote_products.
            push_back(bulk::var<double>(world));

    // put value to other processes
    for (int t = 0; t < p; t++)
        remote_products(t)[s] = local_product;

    world.sync();

```

// compute local product

$\alpha_s := 0;$

for (i := s; i < n; i += p) do

  $\alpha_s := \alpha_s + x_i y_i;$

// broadcast to all processors t

for (t := 0; t < p ; t++) do

  put α_s in $P(t)$;

barrier();

// sum up received α values

$\alpha := 0;$

for (t := 0; t < p; t++) do

  $\alpha := \alpha + \alpha_t;$

`local_size` can be determined using the `local_count()` member function of the `partitioning` (not shown here)

Translating BSP Algorithm to Bulk – Distributed Variables (2)

```
    // computation of global inner product
    double inner_product = 0;
    for (int t = 0; t < p; t++)
        inner_product += remote_products[t];

    // inner product available at each processor
    // end of second superstep
});
```

`// compute local product`
$\alpha_s := 0;$
`for (i := s; i < n; i += p) do`
  $\alpha_s := \alpha_s + x_i y_i;$

`// broadcast to all processors t`
`for (t := 0; t < p; t++) do`
  `put` α_s `in` $P(t);$

`barrier();`

`// sum up received α values`
$\alpha := 0;$
`for (t := 0; t < p; t++) do`
  $\alpha := \alpha + \alpha_t;$

Bulk Library: Coarrays

Coarrays (1)

- **Problem:** Implementing a “broadcast” like shown before with distributed variables is ugly
- What we want to achieve: Put our local value to a defined place on the other processors
- Idea: Allocate an array of size p on each processor
  - Each processor s puts its local value at the s-th place in the array of all other processors

Coarrays (2)

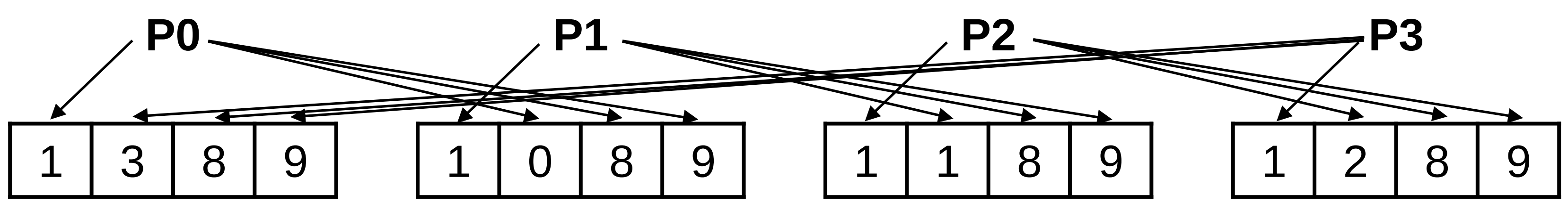
- Approach: Coarray
  - Model distributed data as 2D array
  - First dimension: Processor
  - Second dimension: Chosen processor’s 1D array (according to first dimension)
- Examples:
  - `coarray(t)[41] = value`
    - Put `value` to the 42-th element in the 1D array of processor `t`
  - `coarray(t)[{start,end}] = {value1, value2, ...}`
    - Put `value1, value2, …` to the elements `start, start + 1, …, end - 1` at processor `t` (modifying *slices* of arrays)

Coarrays (3)

```cpp
env.spawn(env.available_processors(), [](auto& world) {
    auto xs = bulk::coarray<int>(world, 4); // coarray of local size 4

    // each processor sets its first local element to 1
    xs[0] = 1;
    auto t = world.next_rank();
    // set second element of processor t's array to rank ID
    xs(t)[1] = world.rank();
    // set third and fourth element of processor t's array to 8 and 9
    xs(t)[{2,4}] = {8, 9};
});
```

Example for 4 processes:



Translating BSP Algorithm to Bulk – Coarrays (1)

```cpp
env.spawn(env.available_processors(),
          [](auto& world) {
    auto s = world.rank();
    auto p = world.active_processors();

    // computation of local inner product
    // assume: x and y cyclic partitioned arrays
    double local_product = 0.0;
    for (int i = 0; i < local_size; i++)
        local_product += x.local(i) * y.local(i)

    // broadcast to all processors t
    auto remote_products =
                    bulk::coarray<double>(world, p);
    for (int t = 0; t < p; t++)
        remote_products(t)[s] = local_product;

    world.sync(); // end of first superstep

```

// compute local product
$\alpha_s := 0;$
for (i := s; i < n; i += p) do
  $\alpha_s := \alpha_s + x_i y_i;$

// broadcast to all processors t
for (t := 0; t < p ; t++) do
  put α_s in $P(t)$;

barrier();

// sum up received α values
$\alpha := 0;$
for (t := 0; t < p; t++) do
  $\alpha := \alpha + \alpha_t;$

Translating BSP Algorithm to Bulk – Coarrays (2)

```cpp
    // computation of global inner product
    double inner_product = 0;
    for (int t = 0; t < p; t++)
        inner_product += remote_products[t];

    // inner product available at each processor
    // end of second superstep
});
```

`// compute local product`
$\alpha_s := 0;$
`for (i := s; i < n; i += p) do`
   $\alpha_s := \alpha_s + x_i y_i;$

`// broadcast to all processors t`
`for (t := 0; t < p ; t++) do`
   `put` α_s `in` $P(t);$

`barrier();`

`// sum up received` α `values`
$\alpha := 0;$
`for (t := 0; t < p; t++) do`
   $\alpha := \alpha + \alpha_t;$

Data Exchange in Bulk: Summary

Object	Image	Description	Code
`var`	local	set	`x = 5`
		use	`auto y = x + 3`
	remote	put	`x(t) = 5`
		get	`auto y = x(t).get()`
`coarray`	local	set	`xs[idx] = 5`
		use	`auto y = xs[idx] + 3`
	remote	put	`xs(t)[idx] = 5`
		get	`auto y = xs(t)[idx].get()`
		put slice*	`xs(t)[{start, end}] = {values...}`
		get slice*	`auto ys = xs(t)[{start, end}].get()`

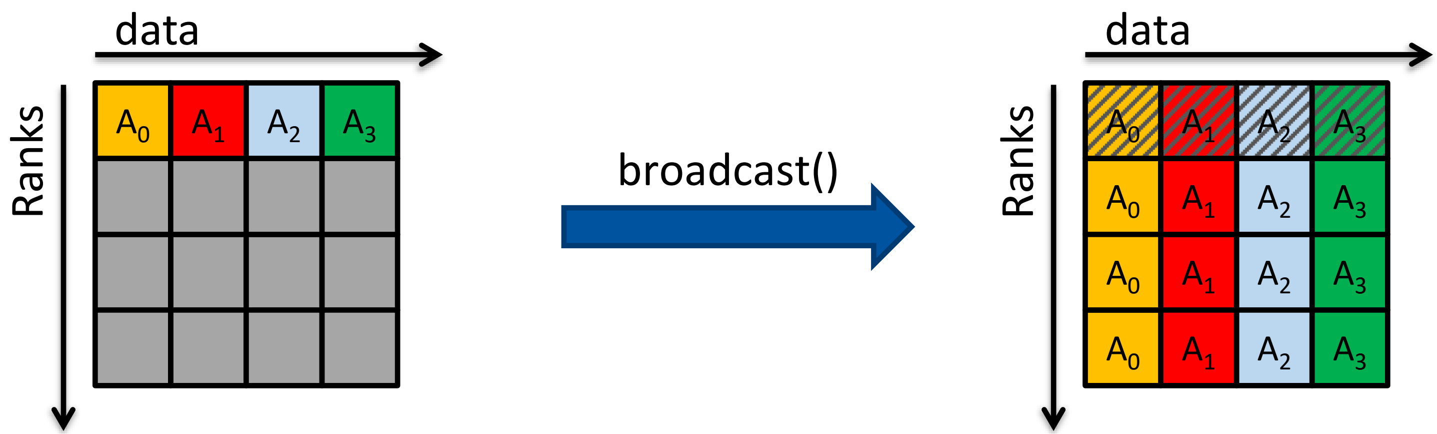
Source: Buurlage, J. W., Bannink, T., Bisseling, R. H.. *Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs*.

*: Slices are represented as `std::vector` containers

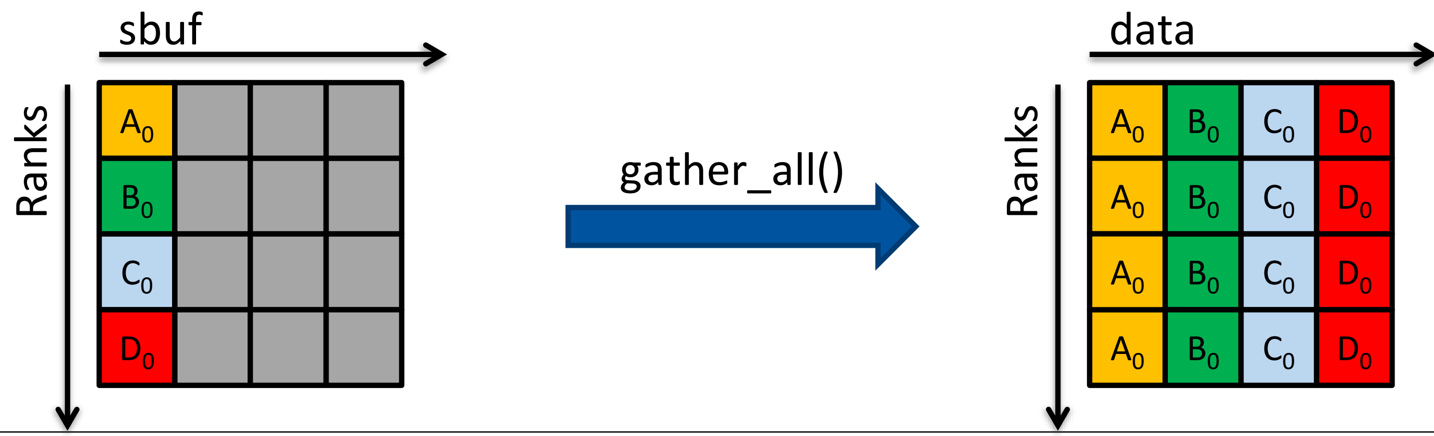
Bulk Library: Further Features

Further Features (1)

- Common communication patterns also available, e.g.:
  - `broadcast()`



  - `gather_all()` (what we did manually in the inner product algorithm)



Further Features (2)

- Mailbox communication: queue object enables message passing
  – Similar to MPI message passing
- Special-purpose objects for vectors, matrices, …
- Pre-implemented algorithms (inner product, sorting, LU decomposition)
- Further reading:
  – https://jwbuurlage.github.io/Bulk/
  – Buurlage, J. W., Bannink, T., Bisseling, R. H.. *Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs*. In EuroPar 2018 (pp. 519-532). Springer
    - https://doi.org/10.1007/978-3-319-96983-1_37

What you have learnt

- Bulk: C++ library implementing the BSP model
  - Supporting different backends: MPI, C++ threads, coprocessors
- Data distribution objects
- Communication mechanisms
  - Distributed variables
  - Coarrays
- Implementation of inner product computation in Bulk
- Other supported communication patterns