Concepts and Models of Parallel and Data-centric Programming

BSP IV (Bulk: Introduction & Data Distribution)

Lecture, Summer 2021

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

Outline

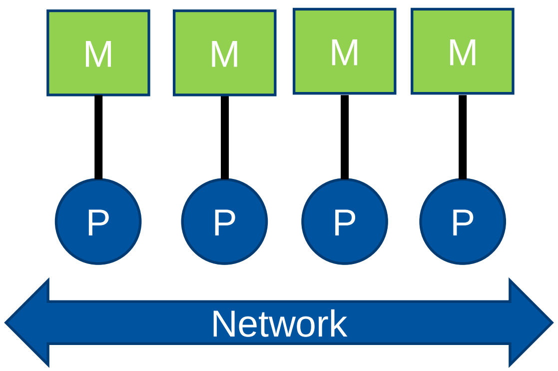
0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. **Bulk-Synchronous Parallelism**
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

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

Recap: BSP Model Components

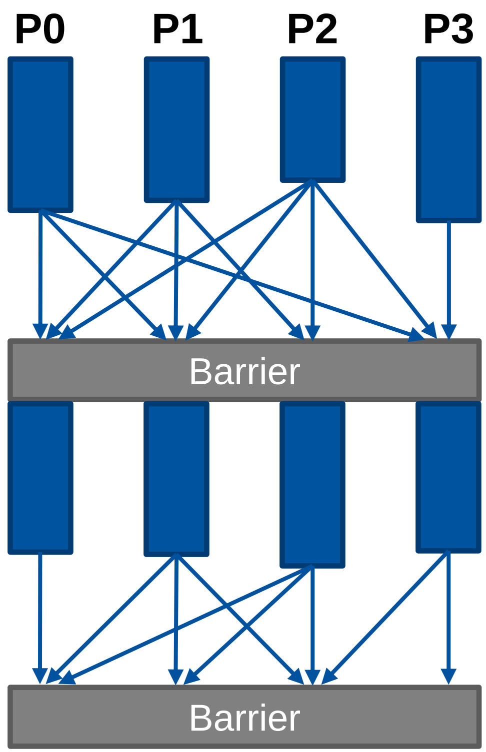
BSP Computer

(Distributed Memory Computer)

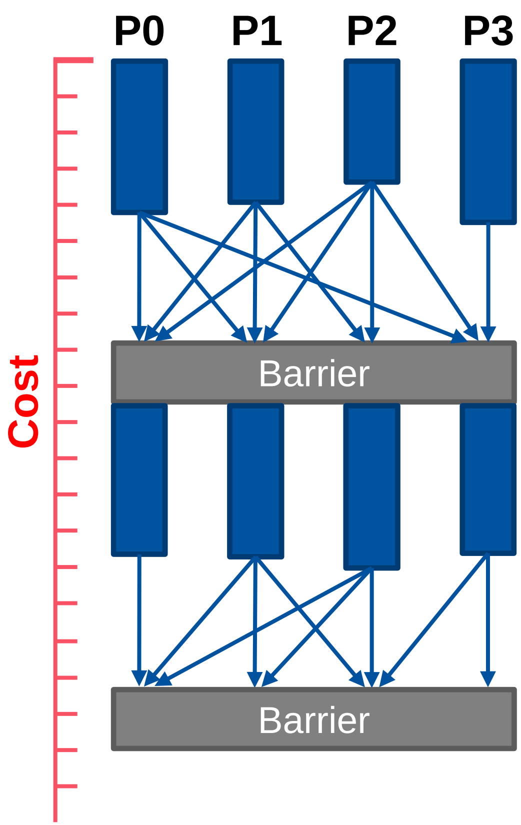


Programming Model

(Algorithmic Framework)

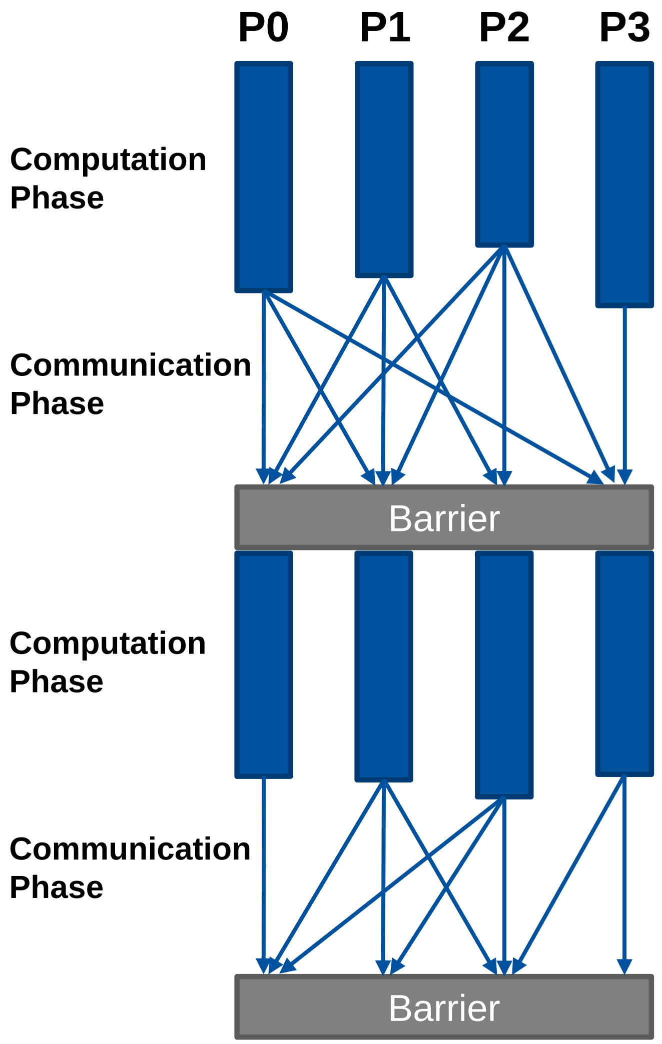


Cost Model



Recap: Supersteps

- BSP algorithm: Series of supersteps
- Superstep: Computation and communication
  - *Computation phase:* Perform local calculations with available data (e.g., FP operations)
  - *Communication phase:* Transfer data (e.g., results) between the different processors
- End of each superstep: Barrier
  - Each processor has to wait until all other processors have reached the barrier.
    → Bulk synchronization
  - Ensures that communication between processors has finished



Bulk Library: Introduction

BSP Libraries

- BSP Libraries: Enable implementation of BSP algorithms
- BSPlib: Definition of a library interface for BSP programming (1998)
  - Implementations in C and Fortran available (as for MPI)
  - BSPonMPI: BSPlib implementation on top of MPI https://github.com/wijnand-suijlen/bsponmpi)
  - Paderborn University BSP-Library (C implementation, compliant with BSPLib): http://publibrary.sourceforge.net
- **Bulk**: Modern implementation of BSP Model in C++17:
  - Uses common idioms and features of C++
  - Increased memory safety, code reuse, less boilerplate code
  - https://jwbuurlage.github.io/Bulk

Bulk Interface

- BSP computer is represented in an `environment` object
  - Captures information about the underlying hardware (could be MPI cluster (distributed memory) or a multi-core processor (shared memory))
- SPMD block using all available processors can be spawned in this `environment`
- All processors in an SPMD block form a *parallel world* captured in a `world` object
  - Enables communication and synchronization between processors
  - Can be queried for information about the local process (e.g., process number, also called `rank`)

Bulk Interface – Simple Example

```
bulk::backend::environment env;
env.spawn(env.available_processors(), [](auto& world) {
    world.log("Hello world from %d / %d\n",
              world.rank(), world.active_processors());
});
```

SPMD Block

- Output on a single CLAIX-2018 node (48 cores)

```
Hello world from 0 / 48
Hello world from 1 / 48
Hello world from 2 / 48
Hello world from 11 / 48
Hello world from 12 / 48
Hello world from 23 / 48
Hello world from 4 / 48
[...]
```

Bulk – Backends

- Bulk supports multiple *backends* on which the BSP program can run
- Backends use different underlying programming models / libraries
- MPI backend: `bulk::mpi::environment`
  - Running on a distributed memory computer / cluster
- Thread backend: `bulk::thread::environment`
  - Running on a shared memory computer / single node, C++ threads
- Further backends for coprocessors (e.g., Xeon Phi)
- We will focus on the MPI backend, because we want to run our programs on a cluster.

Methods for `environment` and `world` objects

Class	Method	Description
`environment`	`spawn`	starts an SPMD block
	`available_processors`	returns maximum p
`world`	`active_processors`	returns chosen p
	`rank`	returns local processor ID s
	`next_rank`	returns $s + 1 \pmod p$
	`prev_rank`	returns $s - 1 \pmod p$
	`sync`	ends the current superstep
	`log`	logs a string message

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

Supersteps in Bulk

- SPMD section contains the supersteps of our BSP algorithm
- `world.sync()` ends the current supersteps by performing required communication and synchronization, next superstep starts afterwards

```
bulk::backend::environment env;
env.spawn(env.available_processors(), [](auto& world) {
    // superstep 0 (comp)
    world.sync(); // comm + barrier
    // superstep 1 (comp)
    world.sync(); // comm + barrier
    // superstep 2 (comp)
    world.sync(); // comm + barrier
    // ...
});
```

Branching Statements

- Use processor identity `world.rank()` to perform special tasks / code for certain processors

```
bulk::backend::environment env;
env.spawn(env.available_processors(), [](auto& world) {
    if (world.rank() == 0) {
        // only executed by P0
    } else if (world.rank() == 1) {
        // only executed by P1
    } else {
        // executed by all processors but P0 and P1
    }

    // executed by all processors
});
```

Bulk Library: Data Distribution

Data Distribution (1)

- Different data partitionings possible
  - `bulk::cyclic_partitioning<D,G>(global_size, grid_size)`
  - `bulk::block_partitioning<D,G>(global_size, grid_size)`
  - …
- `D`: Dimension of partitioning, e.g. 1 for a vector, 2 for a matrix etc.
- `G`: Dimension of processor grid
- `global_size`: Number of elements (per dimension)
- Member function `local_count(rank)` gives the number of elements locally assigned to the given rank

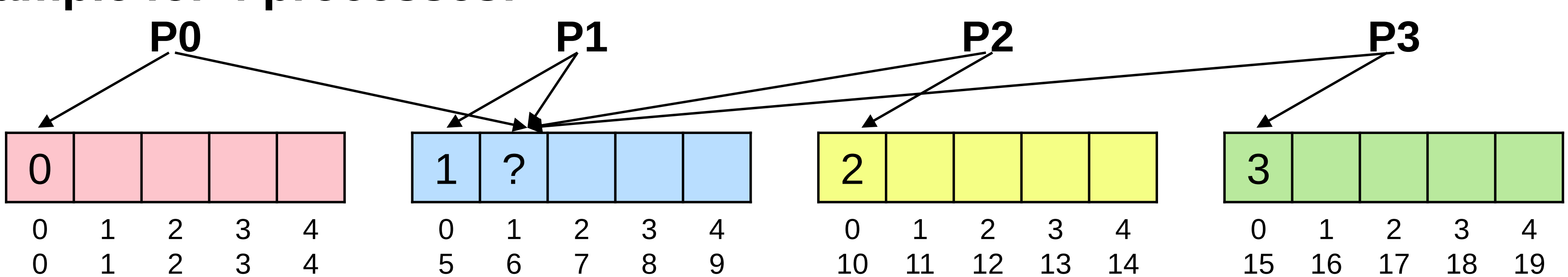
Data Distribution (2)

- `grid_size`: Controls which processors get a chunk of data
  - Only the first `grid_size` processors will get a chunk
  - Has to be smaller than or equal to the number of available processors
  - Set `grid_size` to number of available processors to ensure that every processor gets a chunk
- Special `partitioned_array` object has to be created with a matching partitioning (matching `D` and `G`)
  - `bulk::partitioned_array<Type, D, G>(world, partitioning)`
  - Access values with `local(index)` and `global(index)` member functions

Data Distribution (3)

```
env.spawn(env.available_processors(), [](auto& world) {
      auto n = world.active_processors();
      // block partitioning with 20 elements distributed on n processors
      auto part1 = bulk::block_partitioning<1,1>(20, n);
      // block partitioning with 20 x 20 elements
      auto part2 = bulk::block_partitioning<2,1>({20,20}, n);
      // distributed array with part1
      auto arr = bulk::partitioned_array<int, 1, 1>(world, part1);
      // accessing first local (!) element
      arr.local(0) = world.rank();
      // accessing seventh global (!) element (note: data race)
      arr.global(6) = world.rank(); });
```

Example for 4 processes:



What you have learnt

- Different implementations of the BSP model available
- Bulk: C++ library implementing the BSP model
  - Supports different backends: MPI, C++ threads, coprocessors
- Data distribution with different pre-defined partitionings possible
  - Fine-grained control about data distribution and processor grid
  - E.g. cyclic or block partitioning