Extending Distributed Functionality in Phylanx

Master's Defense

Maxwell Reeser

March 24, 2020

Division of Computer Science and Engineering
School of Electrical Engineering and Computer Science
Louisiana State University

Previous Publications/Presentations

- This presentation was delivered at the SCALA 2020 conference

Outline

Introduction to Phylanx and Background

Algorithms

Results & Future Work

Introduction to Phylanx and Background

Phylanx: An Asynchronous Distributed C++ Array Processing Toolkit

- Write Python code, run it in distributed
- Targeting machine learning mainly
  - Focus on linear algebra
  - Lower barrier to entry for ML Practitioners
    - NumPy API
- Heavy use of HPX
  - Standard's compliant distributed C++ runtime
  - Product of Stellar Group
- Blaze data structures
  - Parallel linear algebra

Distributed Phylanx Roadmap

- Map operations
  - No Data Dependencies
- Distributed Data Structures
  - distributed_object
    - UPC++
  - distributed_vector/matrix
  - Annotations
- Distributed Primitives
  - Previously implemented:
    - Matrix transpose
    - Vector-vector product
    - Matrix-vector product
  - Implemented in this project:
    - Matrix-matrix product

Tiling in Phylanx

- Distributed computation means distributed data
  - Distributed data structures have local (single node) tiles
  - These can be tiles of vectors, matrices, or other data structures
  - Splitting up of data must be intentional
  - We mostly focus on tiling of matrices
- Different tilings can have different costs
  - Eventually we want to minimize the cost

Tiling Example: $A = B + C$

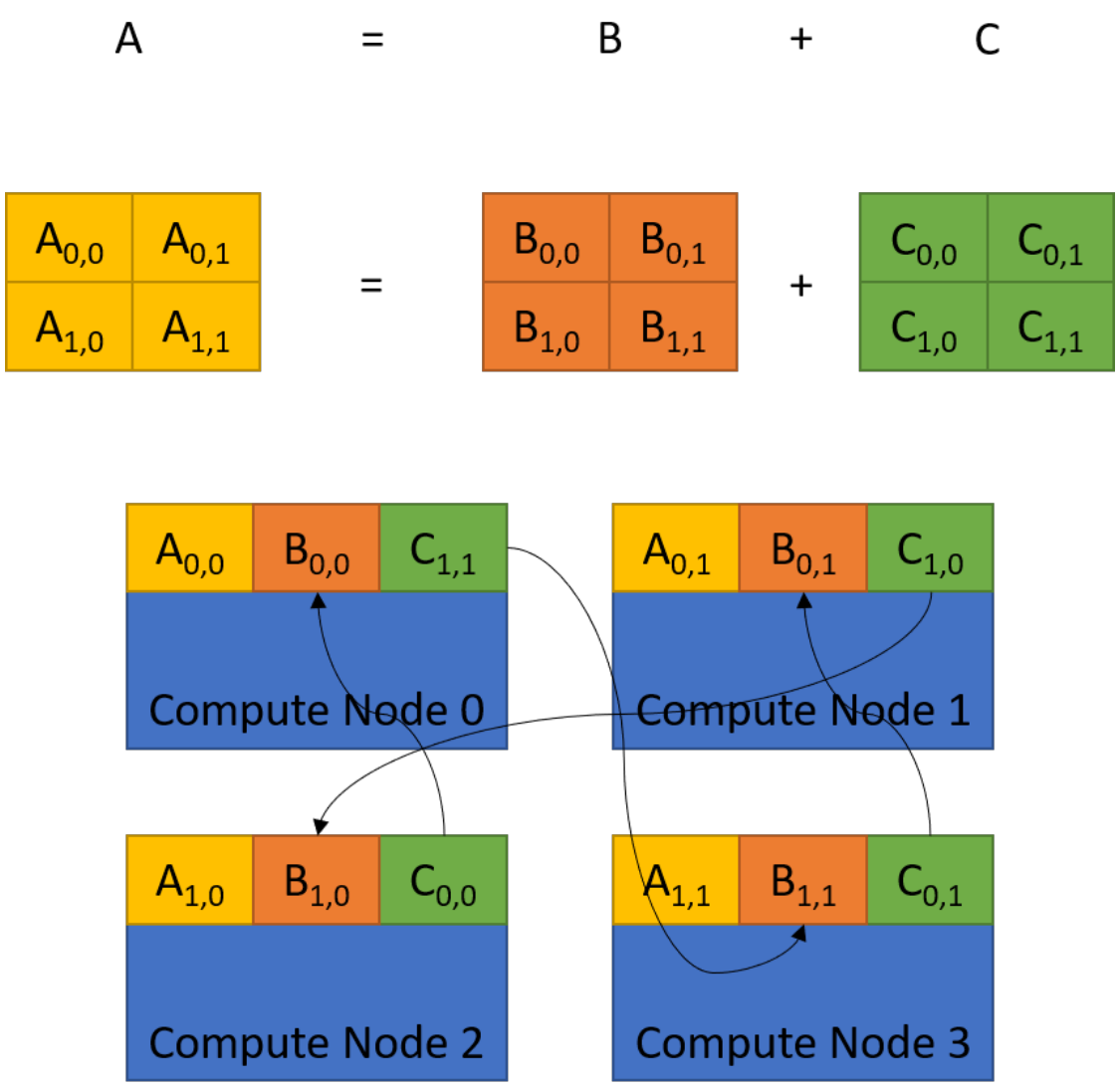


Figure 1: Tiling Mismatch

distributed_matrix

- Object-oriented distributed data organization
- Enabled by HPX's Active Global Address Space (AGAS)
- Maintains a list of participating nodes
- Allows one local tile to "fetch" a non-local tile over the network

distributed_matrix Example

```
util::distributed_matrix<T> lhs_data(lhs_localities.annotation_.name_,
    lhs.matrix(), lhs_localities.locality_.num_localities_,
    lhs_localities.locality_.locality_id_);

std::size_t remote_id = 2;
std::size_t row_start = 0;
std::size_t row_stop = 100;
std::size_t col_start = 0;
std::size_t col_stop = 100;
hpx::lcos::future<blaze::DynamicMatrix<T>> lhs_tmpl =
    lhs_data.fetch(remote_id, row_start, row_stop, col_start, col_stop);
```

Figure 2: Tiling Mismatch

Algorithms

dot_d 2d2d & Cannon Product

- dot_d 2d2d
  - Supports rectangular, non-overlapping tiling on an arbitrary number of nodes
  - Designed by Hartmut Kaiser
- Cannon Product
  - Requires uniform tiling on a perfect square number of nodes
  - Chosen for its simplicity and space efficiency

dot_d 2d2d

- Very flexible matrix multiplication algorithm
- Iterates through all tiles of RHS operand
    - Performs multiplication if intersection detected
- Local partial result matrices may be large
- May require partial result row-aggregation in order to compute final result

dot_d Example: $A = B \cdot C$

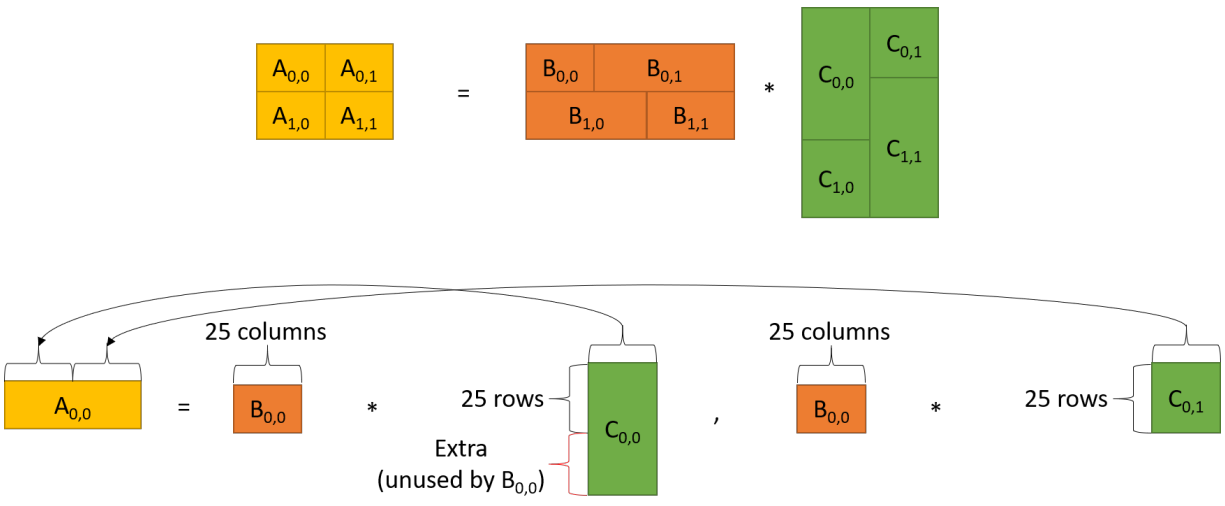
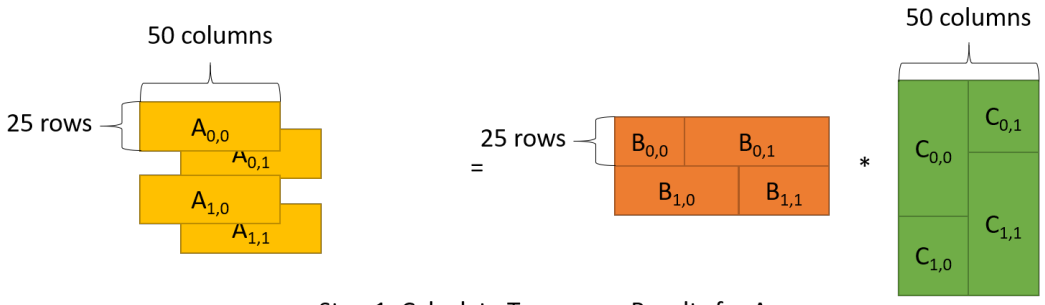
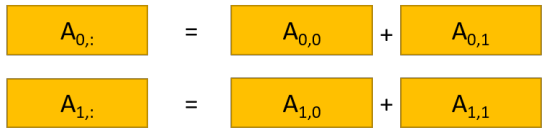


Figure 3: Calculating Intermediate Result

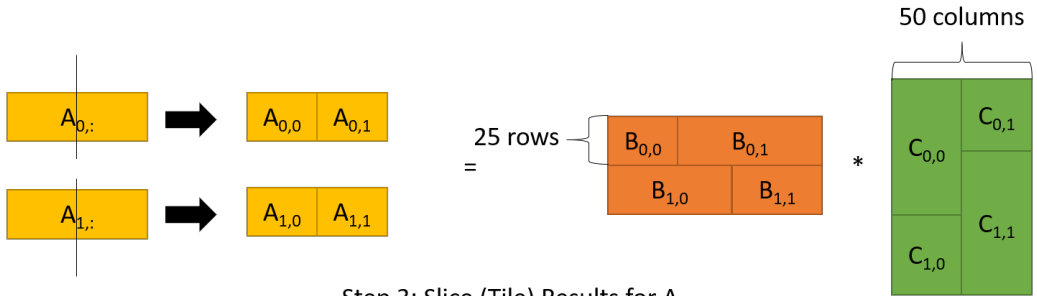
dot_d Example: $A = B \cdot C$



Step 1: Calculate Temporary Results for A



Step 2: Calculate Row Results for A



Step 3: Slice (Tile) Results for A

Figure 4: Calculating Final Tiled Result

dot_d Example: $A = B \cdot C$

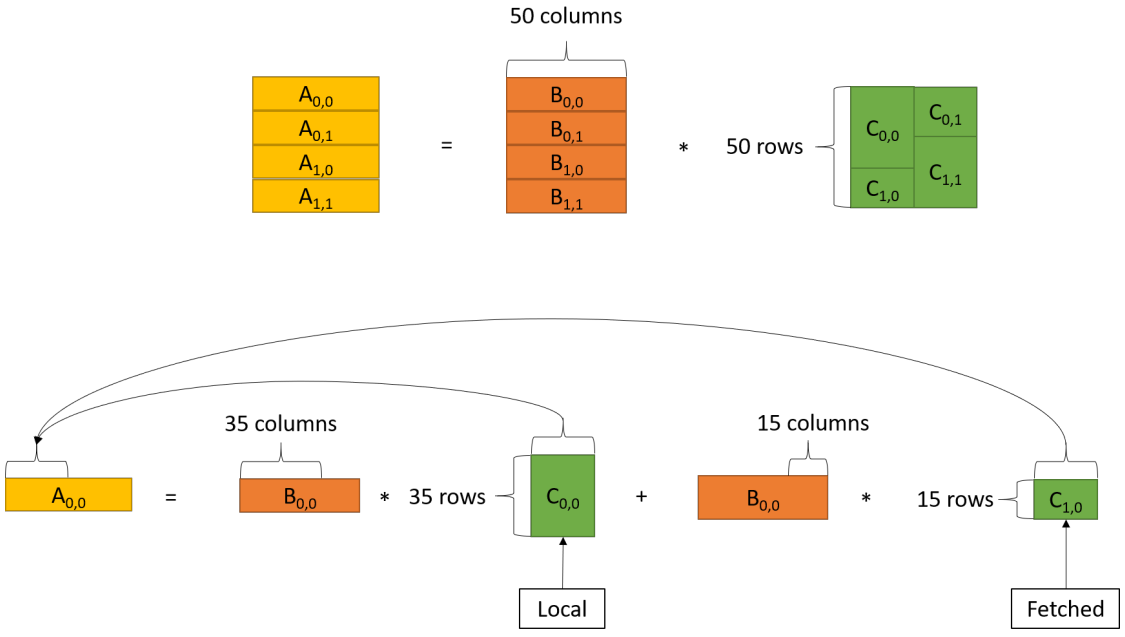


Figure 5: Row Major LHS

Cannon's Algorithm

- Space efficient matrix multiplication algorithm
- Moves both input matrix tiles at every step
  - $\sqrt{P}$ number of iterations (on P processors)
- End result does not require row-aggregation

Cannon Example: $A = B \cdot C$

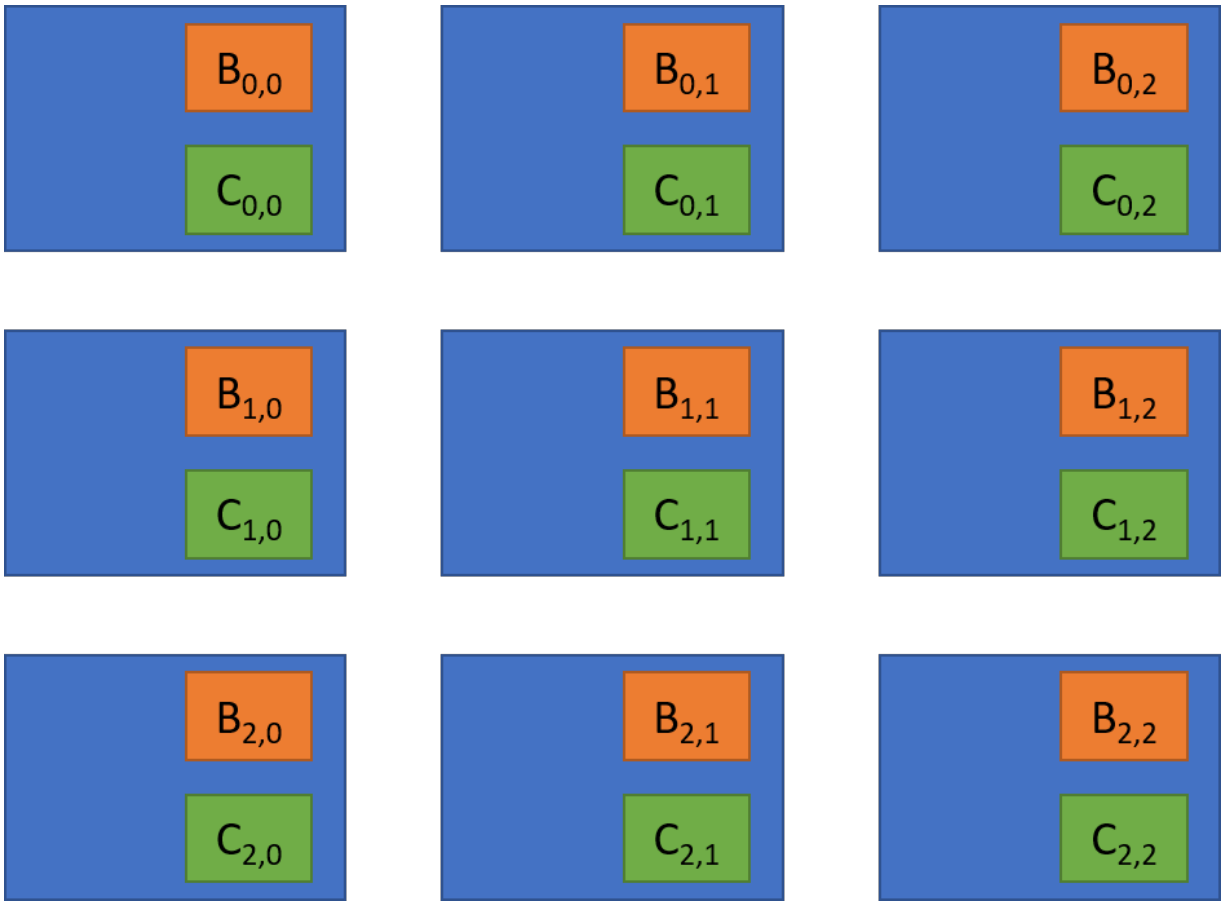


Figure 6: Alignment

Cannon Example: $A = B \cdot C$

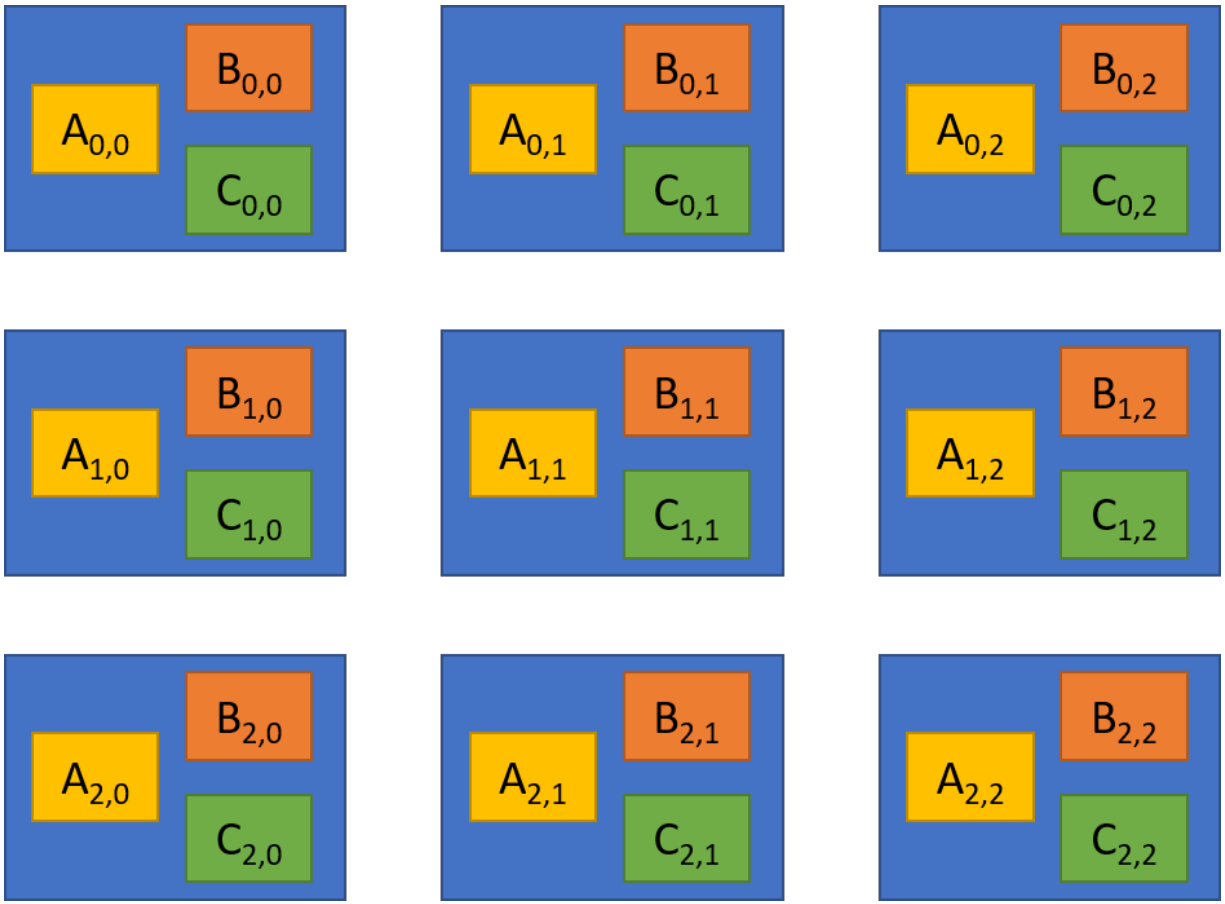


Figure 7: Multiply Local Values

Cannon Example: $A = B \cdot C$

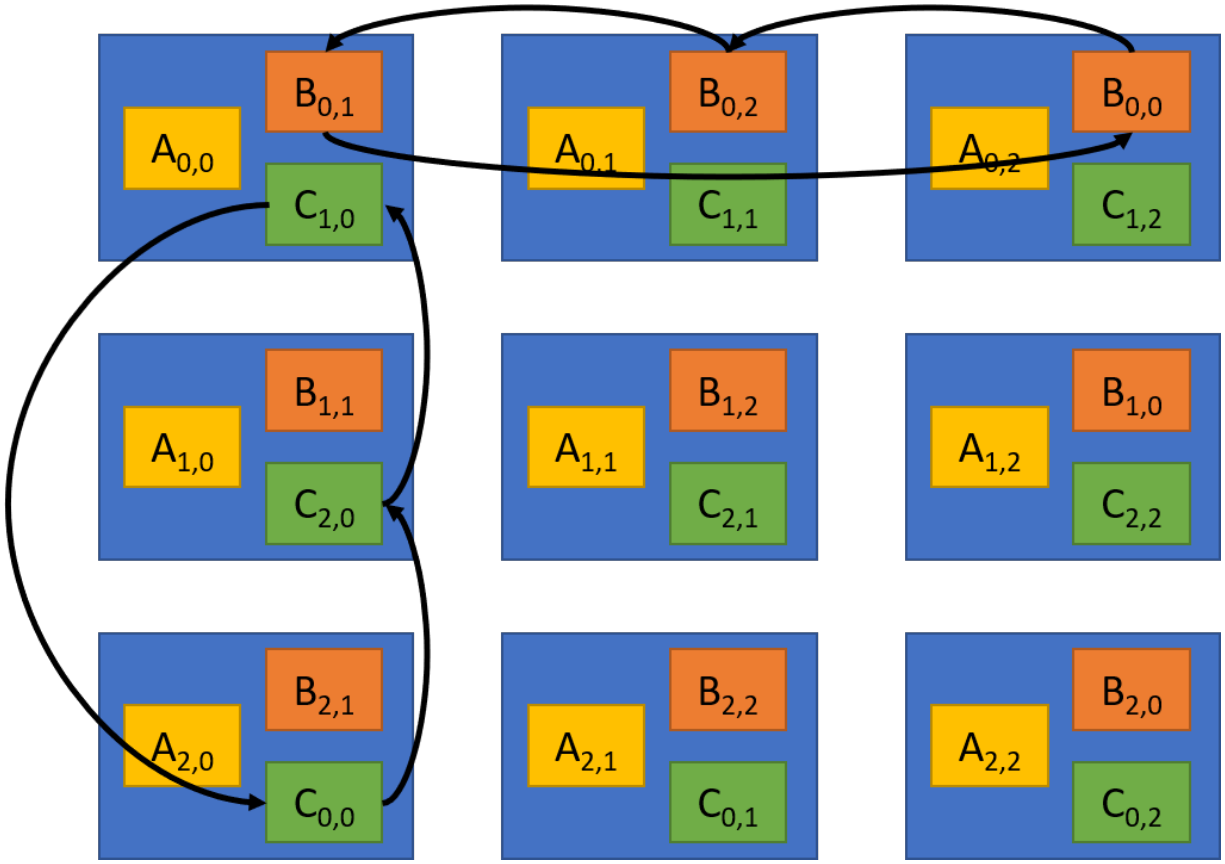


Figure 8: Shift Data & multiply

Cannon Example: $A = B \cdot C$

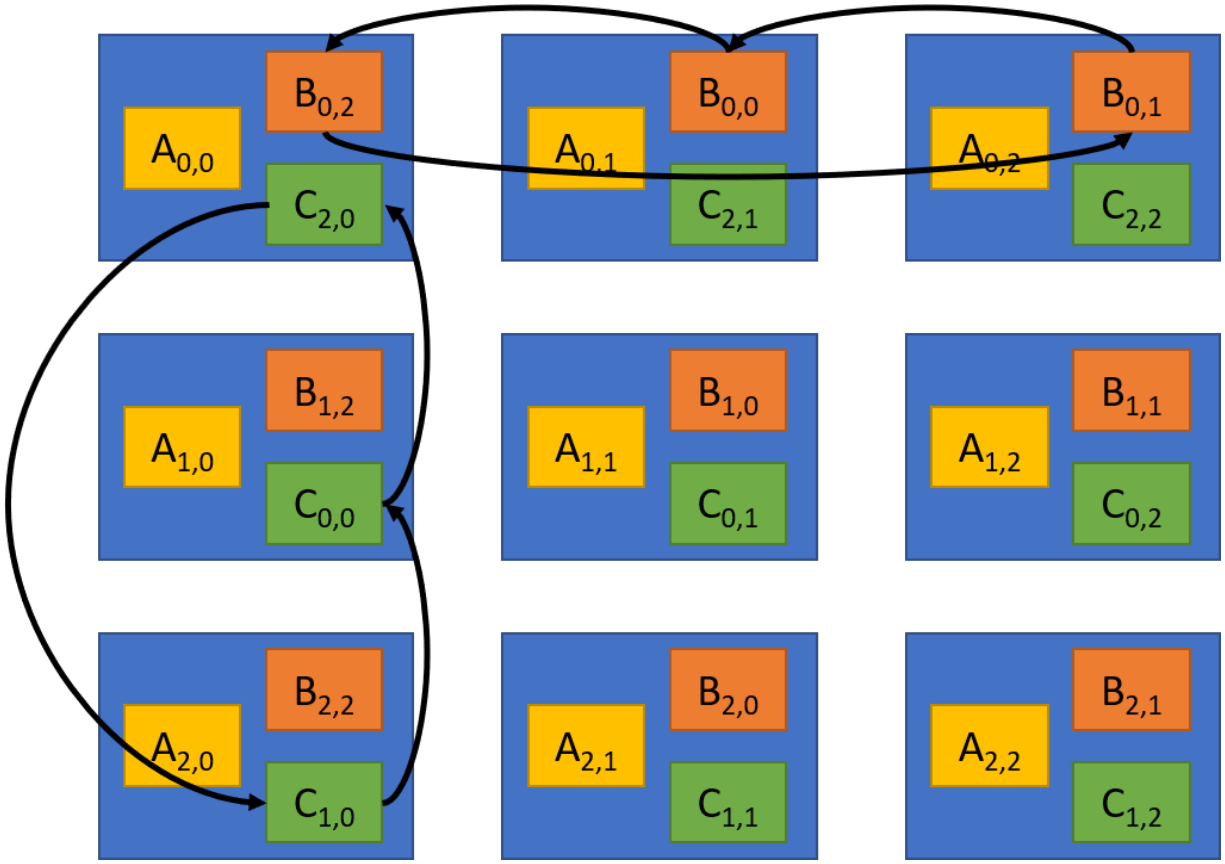


Figure 9: Shift Data & multiply

Futurizing Cannon's Algorithm

- No permanent data moving
  - Pulling instead
  - Doubles memory cost
- Allows pulling to be done one cycle ahead
  - Computation runs while data is being fetched

Results & Future Work

Preliminary Results

- Both distributed algorithms outperformed the pure serial version
- Cannon performed substantially better than dot_d

Matrix Size	dot_d	cannon	dot (serial)
500	5351.08	3131.345	8538.37
1000	36946.15	25889.2	67425.2
2000	282916.5	170091.5	539491

Speedup

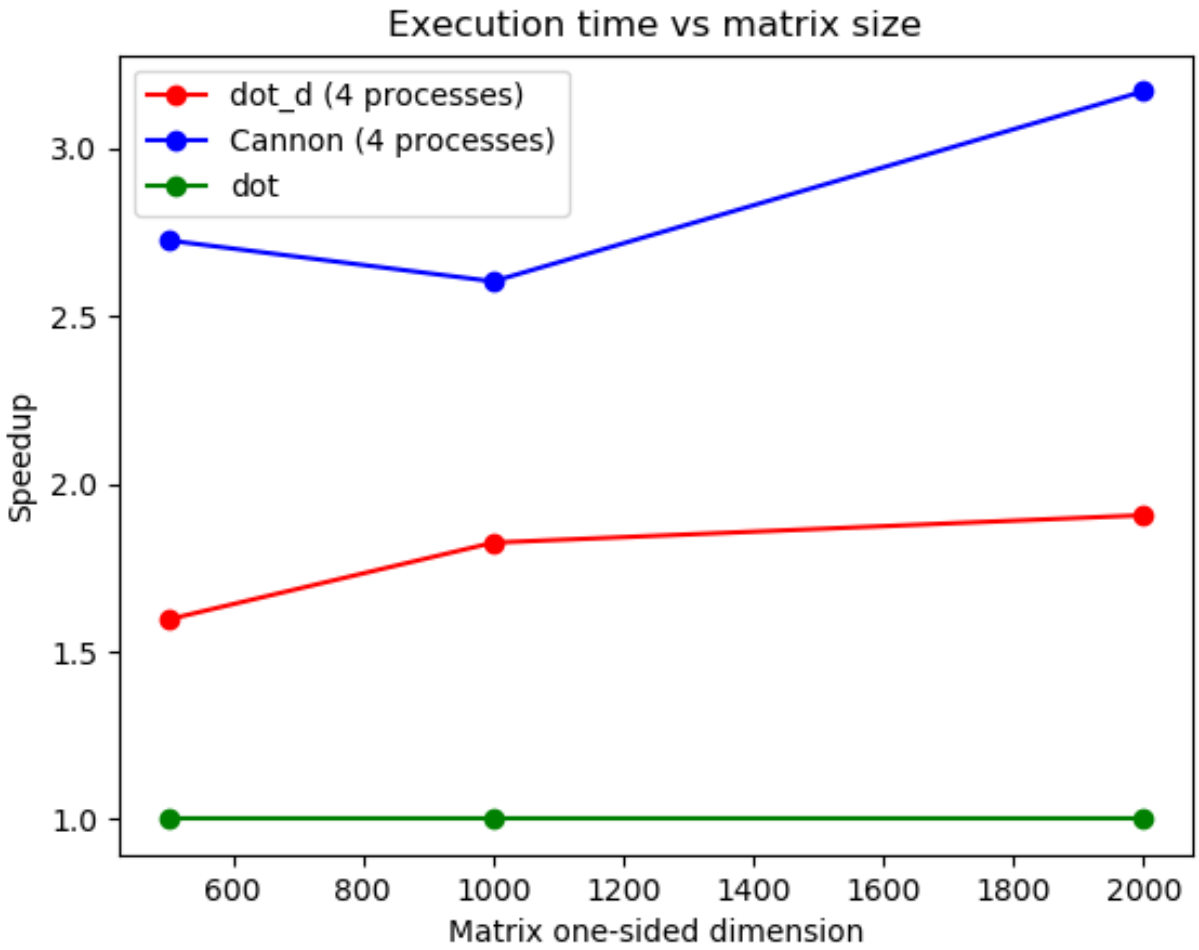


Figure 10: Speedup Plot

Contributions of this work

- Explored performance and implementation differences between two different matrix multiplication algorithms
- Learned more about how tiling structure impacts performance
  - In how it is derived from the algorithm
  - In how we might choose algorithms for their tiling benefits or constraints
- Learned more about our requirements for distributed primitives in Phylanx going forward
- Highlighted known deficiencies in HPX that, if resolved, would improve distributed functionality

Future Work

- Confirm results in a cluster environment
- Tiling testing
- Tiling optimizer

Questions?

Complexity

- In an operation, $A = B \cdot C, B \in M_{NxL}, C \in M_{LxM}$, with B, C containing doubles (8 bytes)
- Cannon's algorithm always transfers $8 * (\sqrt{P} * (N * L/P) + \sqrt{P} * (L * M/P))$ bytes of data at a speed of α bytes/sec, with latency β, meaning it takes $\sqrt{P} * \beta$ time in latency, due to the futurization, with a total maximum memory footprint of $(N * M) + 2 * (N * L) + 2 * (L * M)$.
- dot_d transfers up to $8 * P * (P - 1) * (L * M)$ bytes of data in the main multiplication step, and up to $8 * N * log(L) * M$ bytes in the row-aggregation step, with total latency $P * (P - 1) * \beta + N * log(L) * \beta$ in the worst case. The memory footprint in the worst case is $P * (N * M) + (N * L) + 2 * (L * M)$