EXTENDING DISTRIBUTED FUNCTIONALITY IN PHYLANX

A Report

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Computer Science and Engineering

by
Maxwell Reeser
B.S., Louisiana State University, 2018
February 2020

Acknowledgments

I would like to thank many people for their help over the years in teaching me about the HPX system, C++, software development, many other aspects of producing quality research, as well as how to be a good human. First of all I am indebted to my parents for inculcating in me an appreciation for learning, hard work, and beauty in the arts. I also would like to thank the many teachers, like Mrs. Breaux, Mrs. Jones, and Ms. Cain, who, when I was younger, took an interest in my future, even when I lacked vision for what that future could be. I would like to thank Dr. Dennis Castleberry for igniting my interest in software, and opening a door into a fascinating world full of difficult problems which provides not only the opportunity for satisfying work, but the opportunity to benefit others by its fruit. I would like to also thank Dr. Samuel Kellar, Dr. Bibek Wagle, and Adrian Lemoine for teaching me about HPX and C++ which, while never formally being their responsibility, they were always more than happy to do. Last, but certainly not least, I would like to thank Dr. Hartmut Kaiser, my mentor and friend, for taking an interest in my development as a software developer and researcher, and who was willing to give me a chance to continue my studies in the Master's program knowing that I would benefit from my involvement at CCT and the Ste||ar Group far more than he, or the Ste||ar Group, ever would.

Table of Contents

List of Tables

List of Figures

Abstract

The Phylanx Distributed Array Processing toolkit is a project being undertaken by the Ste||ar Group at LSU, built using the HPX runtime. This system has many single-node functions implemented, but before we started this project it had very few distributed primitives implemented. Because of matrix multiplication's foundational importance in machine learning algorithms, we chose it as our first major foray into distributed functionality. In order to provide options to users, we chose to implement two different algorithms, a tiling ambivalent one, as well as Cannon's algorithm. We found that in a single node distributed test, both of our algorithms outperformed the serial version.

Chapter 1. Introduction

Machine learning libraries like Google's Tensorflow, or Theano provide a mathematical backbone for the linear algebra operations which are at the heart of almost all statistical and machine learning methods. Without them it would be much more difficult to flexibly build and deploy complex machine learning models. As machine learning models continue to become more and more complex, and require larger amounts of data to train, the continued absence of libraries with robust integration of distributed and high performance systems unnecessarily hamstrings forward advancement. Phylanx [1], the distributed array processing toolkit, was created to address these problems. Similar to how other libraries like Tensorflow seek to expose a Python api to enable easier uptake by domain experts, Phylanx was built to enable Python code to be disassembled and optimized for running in a distributed environment, requiring no more complicated knowledge of Python than Numpy or Keras. Under the hood, Phylanx makes use of HPX [2], the C++ distributed runtime developed by the Ste||ar Group, to enable distributed functionality. For optimized shared memory linear algebra performance, Phylanx utilizes the Blaze library [3], developed by Klaus Iglberger. Prior to the work that this report references Phylanx had capitalized on only a few opportunities for distributed functionality, primarily matrix vector products, and matrix addition. In our work on Phylanx we extended this to include two matrix multiplication algorithms, a generalized version allowing irregular matrix tile sizes, and a modification on Cannon's algorithm.

This work was also undertaken with a goal of learning about tiling optimization, and the opportunities for it in a linear algebra system. When we make reference to 'tiling' we mean the result of distributing data across multiple computational nodes so that each data structure which has been distributed (most often a vector or matrix) is composed of many different 'tiles', or subsets, which are placed on many different machines in a cluster.

From preliminary testing we found that our distributed primitives (functions) ran faster than the serial versions in a shared memory distributed environment[1]. In Chapter 5 we discuss these results in detail.

[1] That is, running multiple processes at the same time on one machine, requiring each process to go through HPX's TCP/IP layer to communicate with one another. This meant that the computation was distributed in a sense, but also requires further distributed testing in a cluster environment to confirm our results

Chapter 2. Background and Motivation

2.1. Motivation

In recent years there have been several organizations which amass extremely large datasets. Whether this is browsing data from Amazon, industrial IoT sensor feedback, agricultural data sources, or self-driving car input data, there are some applications for which extremely large amounts of data can be analyzed for a variety of benefits. For certain of these applications, it is necessary to perform operations over the entire dataset at once. Examples of this include training the model for Tesla's self-driving cars, or Amazon or Netflix performing a matrix factorization on transaction or rating data. These examples and more make extensive use of linear algebra operations. Linear algebra libraries have a long history of being tuned to get optimum performance on single machines and in distributed environments using MPI, but the particular use-case of distributed linear algebra in systems like HPX[1], does not have libraries that are as well-standardized.

One example which may help us understand the need for a system like Phylanx is Linear Regression[2]. In order to get a set of weights for the coefficients in a linear predictor system with one dependent variable you must solve the following linear algebra problem.

$$A^T Ax = A^T b + \epsilon$$

Depending on the application, the amount of data involved in a computation like this could be extremely large. For example, with the average person checking their phone 80 times per day, if 2 of those check-ins included use of Facebook, Facebook could get, conservatively, 100 million tcp requests from the United States every day. If, for example, they wanted to test latency on connections based on location they could easily muster a dataset of one billion network requests. Trying to perform an ordinary least-squares (OLS) regression on a dataset with a billion rows on a single node would take a very long time,

[1]HPX is what is known as an Asynchronous Many Task (AMT) runtime

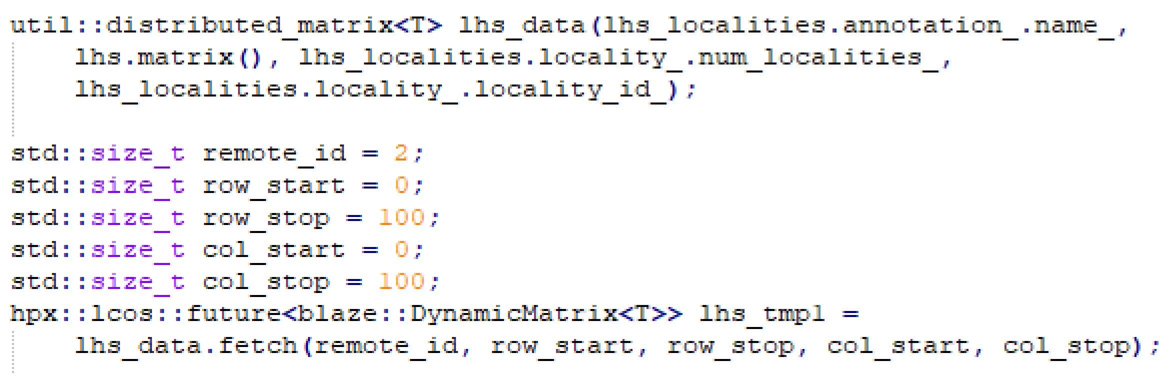
[2]For those who are unfamiliar with the goals of linear regression, please refer to Appendix I

but if they could distribute that application, something they could do easily with services like Amazon Web Services (AWS) or Microsoft's Azure, they could get the result back in much less time. Less time spent waiting on models being run translates to higher efficiency for analytics staff, and higher productivity.

Large linear algebra intensive computations using HPC systems are common in areas like Astrophysics, or meteorology, but less common in the machine-learning community. Most of the code bases which solve these large problems are tailor made for the particular problem using performant languages like C/C++ and Fortran, and often link to CUDA code for GPU acceleration. One of the problems then with extending the availability of HPC to ML researchers is the steep learning curve for developing distributed applications in these lower-level languages. Our goal with Phylanx then, is to provide them with a more familiar, more user-friendly API, which under the hood can take advantage of HPC resources automatically. Rather than create a separate standard, we chose to emulate the NumPy API for our front-end, since NumPy is the most commonly used library in Python, especially for Machine Learning practitioners.

2.2. Matrix Multiplication

In the above example of linear regression it is obvious that the matrix multiplication operation is an important part of the computation. Not only is this the case in linear regression, matrix multiplication is a very common element of Machine Learning algorithms in general. As a result of matrix multiplication's great importance for our target audience, we decided to focus our initial efforts for developing distributed functionality on exposing a distributed matrix multiplication operation. However, in order to better serve our goals with optimizing tiling we have implemented two different algorithms. One, what we are calling dot_d, is extremely flexible when it comes to the tiling, such that it searches for the parts of the tiles in the RHS of the operation which overlap with the relevant dimensions for the local tile of the LHS operand. The other is a modification of Cannon's algorithm, an algorithm for distributed matrix multiplication first developed by Lynn Elliot Cannon



```
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 2.1. Phylanx distributed_matrix example code

in 1969.

2.3. Background

For the algorithms we implemented, we relied on previous work on distributed data structures in HPX by Wei et al [4], work on tiling by Huang et al [5], much work done on the HPX system by Heller et al [2], and work on annotations in Phylanx by Hartmut Kaiser.

2.3.1. Distributed Object

One of the most important precursors for this project was the distributed_object. This is an abstraction which allowed us to coordinate a single data structure across many different localities, and enable retrieval of non-local elements. For the multiplication algorithms we made use of a specialized version of the distributed_object, the distributed_matrix, which allows submatrix fetching on a non-local tile. Figure 2.1 shows it in use.

As you can see, the distributed object exposes a fetch function which allows us to access non-local elements of the data structure in an object oriented way, using C++ Standard Library idioms (futures) similar to how one might access a data member of an ordinary local object. This convenient abstraction allows for distributed software development which is focused on the already difficult nature of structuring SPMD algorithms, instead of the nuts and bolts of remote data access. Another feature of the distributed_object is that it is "lazy", in the sense that it registers itself with AGAS, and only finds the precise location

```
int reduce_to_rank0(int my_hits) {
    // initialize this rank's part of the distributed object with the local value
    upcxx::dist_object<int> all_hits(my_hits);
    int hits = 0;
    // rank 0 gets all the values asynchronously
    if (upcxx::rank_me() == 0) {
        upcxx::future<int> f = upcxx::make_future(my_hits);
        for (int i = 1; i < upcxx::rank_n(); i++) {
            // get the future value from remote rank i
            upcxx::future<int> remote_rank_val = fetch(all_hits, i);
            // create a future that combines f and the remote rank's result
            upcxx::future<int, int> combined_f = upcxx::when_all(f, remote_rank_val);
            // get the future for the combined result, summing the values
            f = combined_f.then([](int a, int b) { return a + b; });
        }
        // wait for the chain to complete
        hits = f.wait();
    }
    upcxx::barrier();
    return hits;
}
```

Figure 2.2. UPC++ dist_object example code

of a colleague when that colleague's data is needed locally, and requested through the fetch function.

We were inspired to create Phylanx's distributed_object by the UPC++ [6] data structure of the same name. The main difference in its use is that Phylanx's distributed_object requires a unique basename to be passed at construction in order for each local portion to find its non-local colleagues through HPX's Active Global Address Space (commonly referrred to as AGAS). Figure 2.2 is sample code taken from the programmer's guide to UPC++ demonstrating the use of that library's version of a distributed_object.

2.3.2. Tiling

As mentioned previously, throughout our development of distributed functionality we had as one of our goals the eventual optimization of user-supplied programs with respect to tiling of distributed matrices. By tiling we mean the intentional distributed arrangement of data segments. For example, in the situation where we want to compute the matrix result

$$A = B + C$$

on a set of four computational nodes, we could have many different arrangements of that data. Figure 2.3 shows how each matrix involved must be intentionally allocated on a particular compute node in order for it to be used in the computation. Figure 2.4 shows a non-optimal tiling (for our addition operation) which will require extra network communication (represented by arrows for the fetching) in order to compute the result. For matrix addition with uniform tiling the optimal solution is simple to find, as there is only one, when all corresponding tiles are co-located on one node. In the case of non-uniform tiling however, there is the possibility that even this optimal solution is not trivial to find, if blocks of A and B do not have an exact dimensional correspondence[3].

For matrix addition, tiling decisions can be very easy. But for operations like matrix multiplication, there can be much greater complexity. Complicating factors include choice of algorithm, number of available processors, pre-existing tiling, as well as others. For instance, just in the algorithms we implemented, Cannon's algorithm requires a perfect square number of processors while dot_d will function with any number of processors. For situations with 35 available processors then, you may need to choose between the potential communication reduction benefits of tiling your matrices on 25 processors for compliance with Cannon's requirements, or trying to take advantage of all of the available processors with dot_d. These are the sorts of questions we hoped developing more distributed functionality in Phylanx might help us answer, whether we learn more in the design process or through the utilization of Phylanx as a testing environment.

2.3.3. Annotations

The data structure used to keep track of which blocks of each matrix are stored on which computational node is called the "annotation" in Phylanx. This is essentially a list of lists, which is established when the data structure is initially created, for example, when a matrix is loaded from disk and partitioned onto multiple nodes. At construction, the participating localities coordinate using AGAS, submitting their calculated share of the

[3]This extra difficulty in finding an optimal set of tiling choices while non-uniform tiles are present is one reason why, although dot_d allows us flexibility in this regard, we may not prefer to make use of it

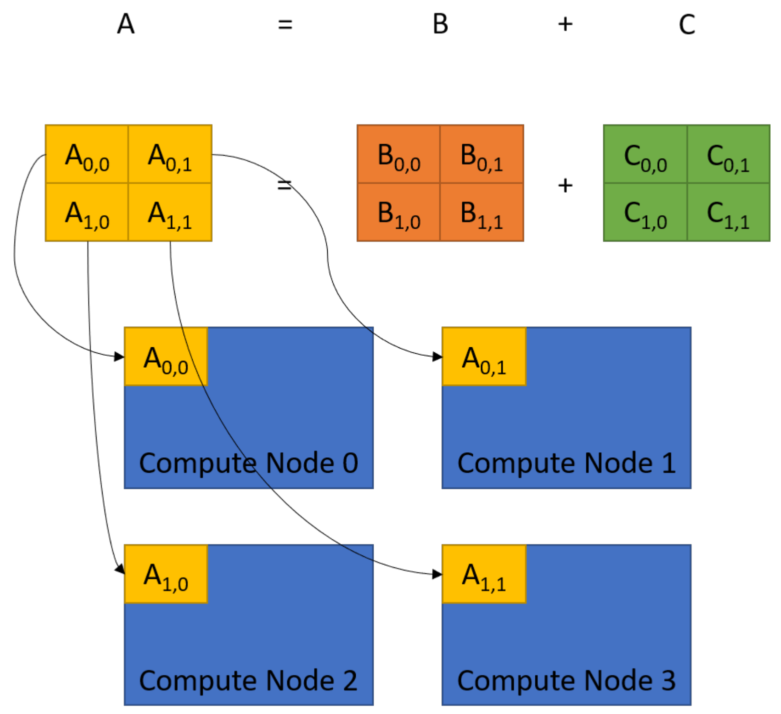


Figure 2.3. Creating a distributed object

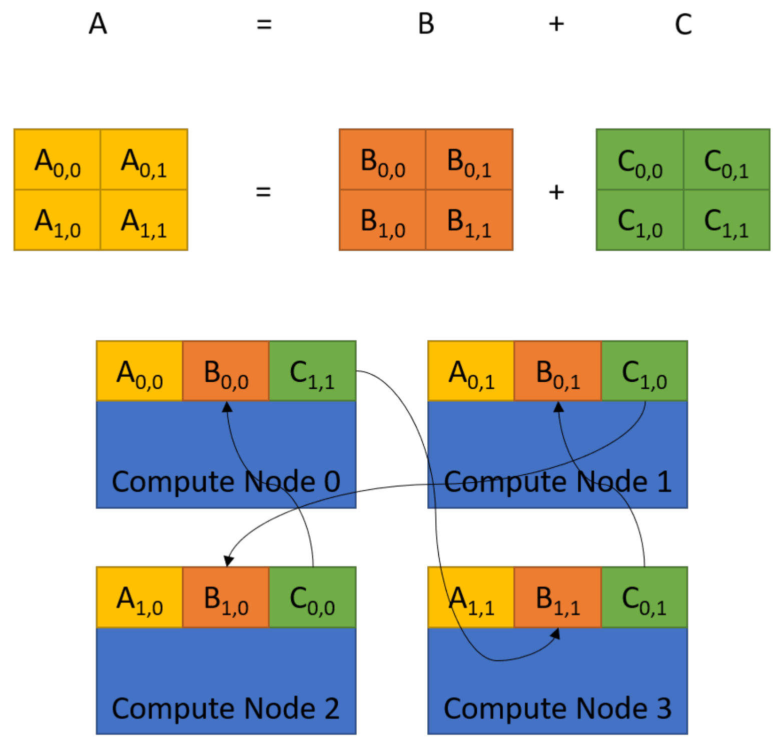
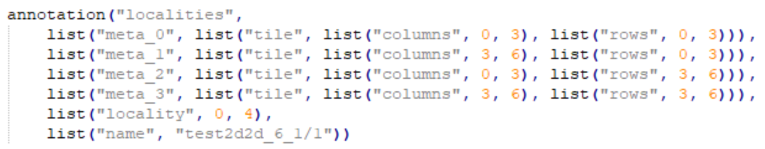


Figure 2.4. Network communication required to address tiling mismatch



```
annotation("localities",
    list("meta_0", list("tile", list("columns", 0, 3), list("rows", 0, 3))),
    list("meta_1", list("tile", list("columns", 3, 6), list("rows", 0, 3))),
    list("meta_2", list("tile", list("columns", 0, 3), list("rows", 3, 6))),
    list("meta_3", list("tile", list("columns", 3, 6), list("rows", 3, 6))),
    list("locality", 0, 4),
    list("name", "test2d2d_6_1/1"))
```

Figure 2.5. Four node annotation

macro-level data structure, so that after every node has chimed in each node receives a copy of the annotations for the macro-level data structure. This means if the data structure, for example a matrix, is used in an algorithm, say multiplication, the node can find from the annotation which node owns the data it needs to progress in the calculation. Figure 2.5 is an example of an annotation generated from a matrix of dimension 6x6 distributed across four localities. Again, each node stores a copy of this metadata, making it possible to identify the home node of any element in the macro-level matrix.

It's important to note that while annotations are used in concert with the distributed_matrix, the latter does not exist at all times. The underlying data, allocated locally on each node, persists as long as it is needed using RAII, and the distributed_matrix behaves similarly. That is, the distributed_matrix only exists as long as it is needed in the computation. Once the computation needing it is complete, it goes out of scope. Since the distributed object is "lazy", even if a matrix is distributed, and involved in many subsequent operations, each participating locality registers with AGAS only once per construction, and the IDs assigned by AGAS then are fetched as needed each time. Since all of these operations are occurring concurrently, with no object-wide barriers, this ensures that the program preserves its futurized nature (see chapter 4 for a discussion of futurization in AMTs). The annotations do still require a barrier on all participating localities at construction, in order for the annotation to be copied and distributed to each participating localities, but after that, the distributed_matrix requires no further hard barriers unless the algorithm itself necessitates it. The operation of the distributed_matrix was designed this way largely to avoid altering the most basic, persistent data-types in use in Phylanx, ensuring that they remained simpler shared memory data structures.

2.3.4. Summary

The Phylanx project was started with the goal of producing a high-quality, performant package which would enable a simple pathway for domain scientists to port their programs to a distributed environment. Prerequisites for our work towards this goal have included development of distributed data structures, and representations for, as well as a system for better understanding, tiling.

Chapter 3. Matrix Multiplication: dot_d

One of the two algorithms we implemented for matrix multiplication was what we call the dot_d algorithm. This algorithm takes full advantage of the flexibility of the annotations, and is capable of using inputs which are distributed across, among other things, arbitrary numbers of localities, and non-uniform (but still rectangular) tiling schemes. This is in contrast to the requirements of Cannon's algorithm, which requires a perfect square number of participating localities and a uniform tiling.

This algorithm uses the local tile of the LHS operand to set the pace for the computations it performs. What this means is that the LHS operand defines many aspects of the computation rather than depending on information about both the LHS and RHS. For one thing, the algorithm requests only portions of other tiles which overlap in a relevant way with the local LHS data. This could mean, for example, that some computational node X does not even make use of the entirety of the local RHS tile on X. It allows some other LHS residing on node Y which needs the data to request the leftovers of the RHS on X, rather than node X requesting the data missing from its own LHS from some other node Z, in order to take full advantage of X's local RHS data. Figure 3.2 displays this situation. Although most algorithms do not have this sort of flexibility as a requirement, as mentioned previously it pairs nicely with the very restrictive Cannon's algorithm for supplying options to users, or the tiling optimizer.

The dot_d primitive uses the annotations directly in the multiplication step by iterating through all tiles in the RHS, searching for those which have a relevant overlap, and calculating the submatrix result in its output matrix. In the process of matrix multiplication, when multiplying two matrices $A = B \cdot C$, with $B \in M_{\alpha\times\beta}$, and $C \in M_{\beta\times\kappa}$, we perform $\alpha \times \kappa$ total dot products (row of LHS times column of RHS). When we are performing tile multiplication, we are multiplying a subset of one matrix by a subset of another. We can represent this as tiles

$$B_i, B_i \in M_{\alpha_i\times\beta_i}, \sum_i \alpha_{B,i} = \alpha$$

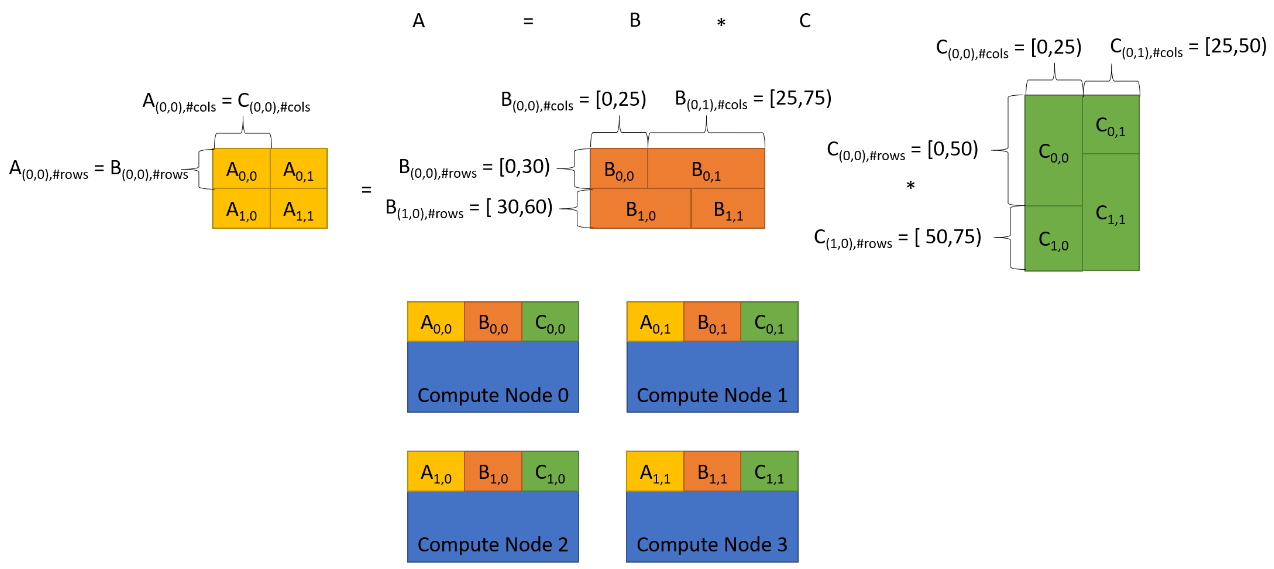

A = B * C
$A_{(0,0),\#cols} = C_{(0,0),\#cols}$
$A_{(0,0),\#rows} = B_{(0,0),\#rows}$
$A_{0,0}$ $A_{0,1}$ $A_{1,0}$ $A_{1,1}$
=
$B_{(0,0),\#cols} = [0,25)$
$B_{(0,1),\#cols} = [25,75)$
$B_{(0,0),\#rows} = [0,30)$
$B_{(1,0),\#rows} = [30,60)$
$B_{0,0}$ $B_{0,1}$ $B_{1,0}$ $B_{1,1}$
$C_{(0,0),\#cols} = [0,25)$
$C_{(0,1),\#cols} = [25,50)$
$C_{(0,0),\#rows} = [0,50)$
$C_{(1,0),\#rows} = [50,75)$
*
$C_{0,0}$ $C_{0,1}$ $C_{1,0}$ $C_{1,1}$
$A_{0,0}$ $B_{0,0}$ $C_{0,0}$ Compute Node 0
$A_{0,1}$ $B_{0,1}$ $C_{0,1}$ Compute Node 1
$A_{1,0}$ $B_{1,0}$ $C_{1,0}$ Compute Node 2
$A_{1,1}$ $B_{1,1}$ $C_{1,1}$ Compute Node 3

Figure 3.1. Calculating Tile Output Size

$$C_i, C_i \in M_{\beta_i \times \kappa_i}, \sum_i \kappa_{C,i} = \kappa$$

$$\sum_i \beta_{B,i} = \sum_i \beta_{C,i} = \beta$$

In order to compute the result matrix A, one node in the dot_d execution holds one B_i, its local LHS tile, as well as one C_i its local RHS tile. It iterates through all of the known C_j, searching for any tile which has any range of rows which is a subset of the range of columns which B_i posesses. When it has found one, it takes the subset of B_i which has the same range of columns as the range of rows which the C_j has, and stores their product as part of the output tile A_i. The dot_d primitive generates output tiles with dimensions of the number of rows in B_i by the number of columns in C_i. It is worth noting that although we have been referring to dot_d as an algorithm which can tolerate arbitrary tilings, it is required that the tiles of B form a tile row with homogeneous boundaries. That is, if one tile has rows $[25, 50)$, and another tile has rows $[24, 49)$, these boundaries are nonhomogeneous, and will cause an error. The same is true of C, albeit with respect to tile columns. Figure 3.1 demonstrates these aspects of the tile output size, and its relation to the tile input size.

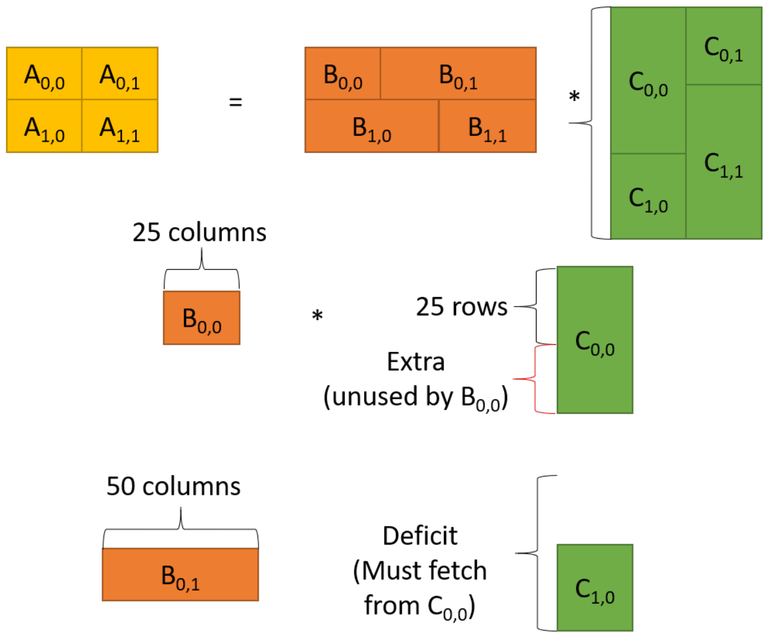


Figure 3.2. Dealing with Non-Uniform Tiling

When the LHS operand is not tiled such that the tile row is composed of only a single tile (i.e., the matrix is tiled in a row-major fashion), it is required that all elements in the tile row add together their partial results. This is because unless node i possesses all elements of the rows in the LHS, it will be performing only partial dot products. Figure 3.2 shows how this happens. In the figure, the tile $B_{0,0}$ does not possess all columns of the matrix for each row it has. As a result, it uses only a limited set of rows in the tiles fetched from C. This limitation means the dot products it performs are incomplete, and are only partial sums. Although in the graphic the output matrix is cut into even tiles, this can only happen at the end of the computation. In the middle of the computation, the node which contains $B_{0,0}$ has a partial sum of A which is of dimension (# of rows in $B_{0,0}$) x (# of columns in C). All such intermediate results which have the same set of rows then (in the example in figure 3.2, nodes (0,0), and (0,1)) must be summed to get the actual result for that section of A. At that point, the result can be sliced up and tiled across the nodes in the original tile row, giving us the nice output matrix A in 3.2. This process is visualized in figure 3.3.

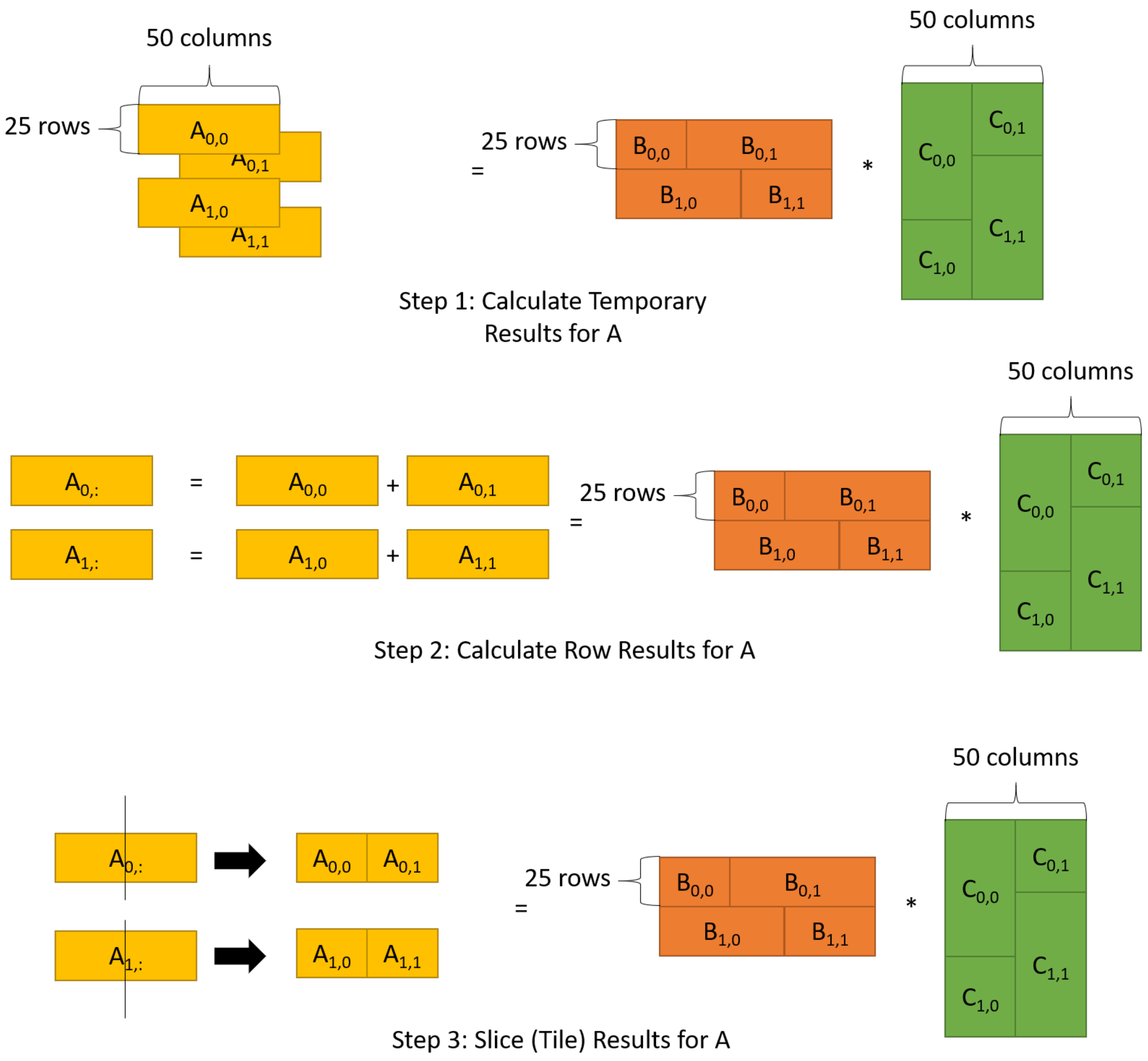

50 columns
25 rows
$A_{0,0}$
$A_{0,1}$
$A_{1,0}$
$A_{1,1}$
=
25 rows
$B_{0,0}$
$B_{0,1}$
$B_{1,0}$
$B_{1,1}$
*
50 columns
$C_{0,0}$
$C_{0,1}$
$C_{1,0}$
$C_{1,1}$
Step 1: Calculate Temporary Results for A
$A_{0,:}$ = $A_{0,0}$ + $A_{0,1}$
$A_{1,:}$ = $A_{1,0}$ + $A_{1,1}$
Step 2: Calculate Row Results for A
$A_{0,:}$
$A_{1,:}$
Step 3: Slice (Tile) Results for A

Figure 3.3. Calculating intermediate sums and tiling them

Chapter 4. Matrix Multiplication: Cannon's Algorithm

The second algorithm that we implemented for matrix multiplication was Cannon's algorithm. This algorithm avoids the reduction step which is necessary in the dot_d primitive in the case where the tiles do not encompass an entire row. This is the main advantage for Cannon's, and we believe is one of the reasons why it generally substantially outperforms the dot_d primitive in terms of speed. Cannon's does have more stringent tiling requirements than dot_d does, however. It requires that the matrices be tiled uniformly across a perfect square number of partitions. These two requirements mean that if a user wants to use this primitive in the middle of a long program with outputs of other operations, those outputs must either already be structured correctly, or be retiled before they can be passed as arguments to Cannon's.

4.1. Main Algorithm

Cannon's Algorithm is structured mainly to avoid communicating partial results. It achieves this by moving the input tiles rather than the output tiles. For a computation $A = B \cdot C$, distributed on 9 localities, we have the first iteration in figure 4.1.

Figure 4.2 shows how after the first iteration we move the tile rows of the LHS to the left (with wraparound) and the tile columns of the RHS up (with wraparound). In the traditional formulation of Cannon's algorithm, the only loaded copy of the data is moved in each iteration. Within the design ethos of Phylanx, this would complicate the algorithm, as it introduces a hard dependency tree of move operations. For example, if node (0,0) in the tiling structure is much faster than its neighbors, it could be forced to wait until nodes (1,0) and (0,1) were done with their data so it could be moved to calculate the next partial result for (0,0). This waiting could happen at every single iteration, severely hampering execution speed. What we decided to do was not to move the sole copy of the data, but to instead use a rolling copy retrieval method, so that when the execution started, every node fetches (copies) the data to its right and below first, then the data two nodes right, and two nodes down, and so on. This does carry the cost of duplicating the tiles, but allows

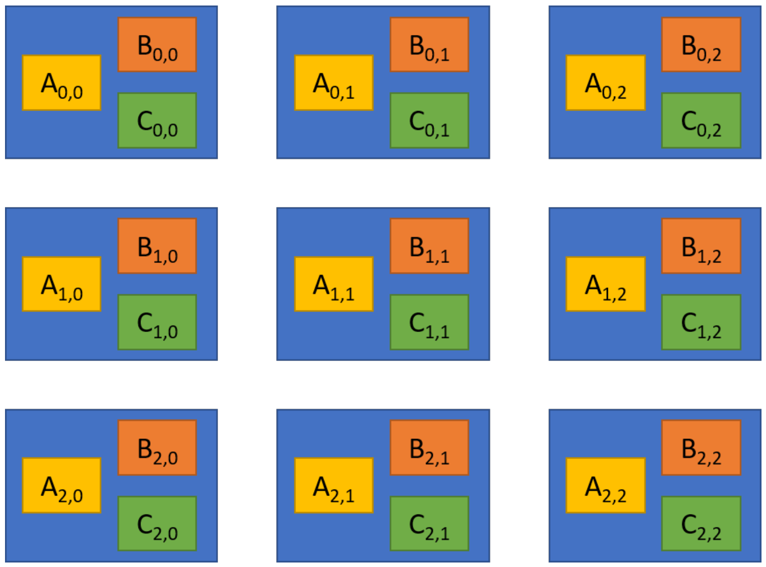


Figure 4.1. Starting Point of Cannon's Algorithm

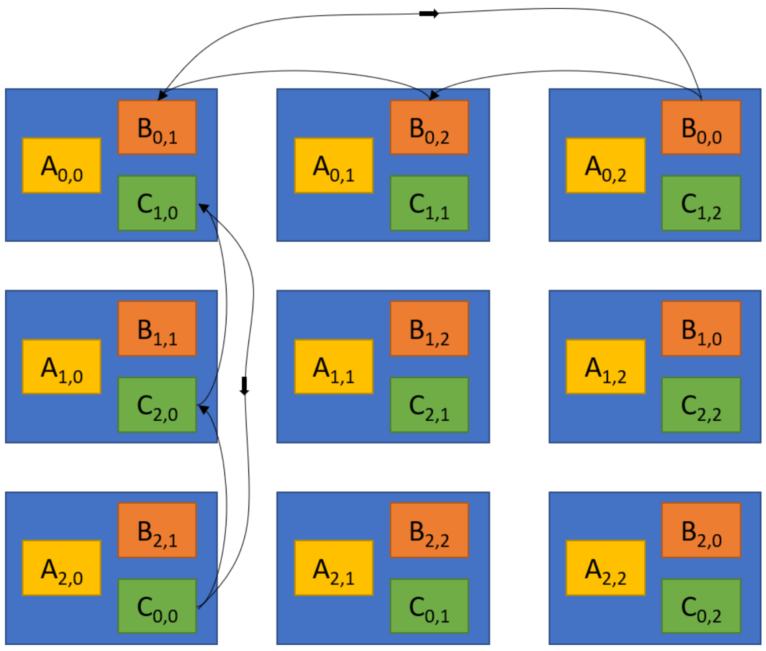


Figure 4.2. Data Moving

the the cluster to make use of all available computing capacity.

A common practice in AMTs is to turn a single program into an execution tree composed of discrete computational units. This process allows the system to first generate the specification for all the computations needed, and perform them in order of necessity. The process of representing the computation as an execution tree is referred to as "futurizing" a computation. The term refers to the fact that intermediate results become a "future," after the standard library convention, meaning that while the result is not complete yet, you can attach computation to the result which will execute when it is complete. In order to further optimize Cannon's algorithm we chose to futurize the network communication so that we issue fetch commands on the tiles α we will need next, perform the multiplication on the tiles we have, issue fetch commands for the next tiles β, and perform the multiplication on the α tiles we fetched previously, continuing until the entire multiplication is done.

Unlike the dot_d primitive, the Cannon product does not have substantial difficulty in determining output tile size. Since all tiles of the LHS must be equal to each other, and since this requirement similarly holds for tiles of the RHS, the output tiles are all of the same size, that is, they will be the number of rows in a tile of B, by the number of columns in a tile of C.

Chapter 5. Conclusion

At the outset of this project we had several goals in mind. These included learning more about how to implement distributed primitives in the future, learning more about how to design these primitives with tiling in mind, and the actual development of two performant, distributed matrix multiplication algorithms. With respect to the first goal we now know that distributed primitives may need auxiliary data structures to support efficient organization of tiling information, and that writing distributed primitives is a process that is uniquely challenging, in a way that serial primitives is not. With respect to auxiliary data structures, an example is how we were required to construct a representation of the tile row from the annotation for the LHS operand and a representation of the tile column for the RHS operand in the Cannon Product. This is directly related to how the Cannon Product runs, and other algorithms may need their own tailored way of traversing existing tiling. In the case of dot_d, we did not need an extra data structure for execution aside from the existing annotations. With respect to the second goal we learned that for tiling analysis with the goal of optimization, uniform tiling is much simpler to work with than non-uniform tiling. For the last goal, we produced two algorithms for matrix multiplication, as we set out to.

5.1. Results

With the two matrix multiplication algorithms completed, we were able to run some preliminary performance tests. Table 5.1 displays those results in tabular form, and figure 5.1 displays them in a graph. In figure 5.2 we can see the speedup achieved by the respective algorithms, calculated as $Speedup = t_1/t_N$ where t_1 is the time in the single process version and t_N is the version with N processes. In our case, as they are only preliminary results, we only tested on 1 process, and on 4 processes. We obtained these results running the application with 4 separate processes on a single Windows machine, versus a single process for the linear version. Due to unexpected technical complications, we were not able to use the threading ordinarily used in the Blaze linear algebra library for our tests. Although our

Matrix Size	dot_d	Cannon	dot (serial)
500	5351.08	3131.345	8538.37
1000	26946.15	25889.2	67425.2
2000	282916.5	170091.5	539491

Table 5.1. Preliminary Test Results

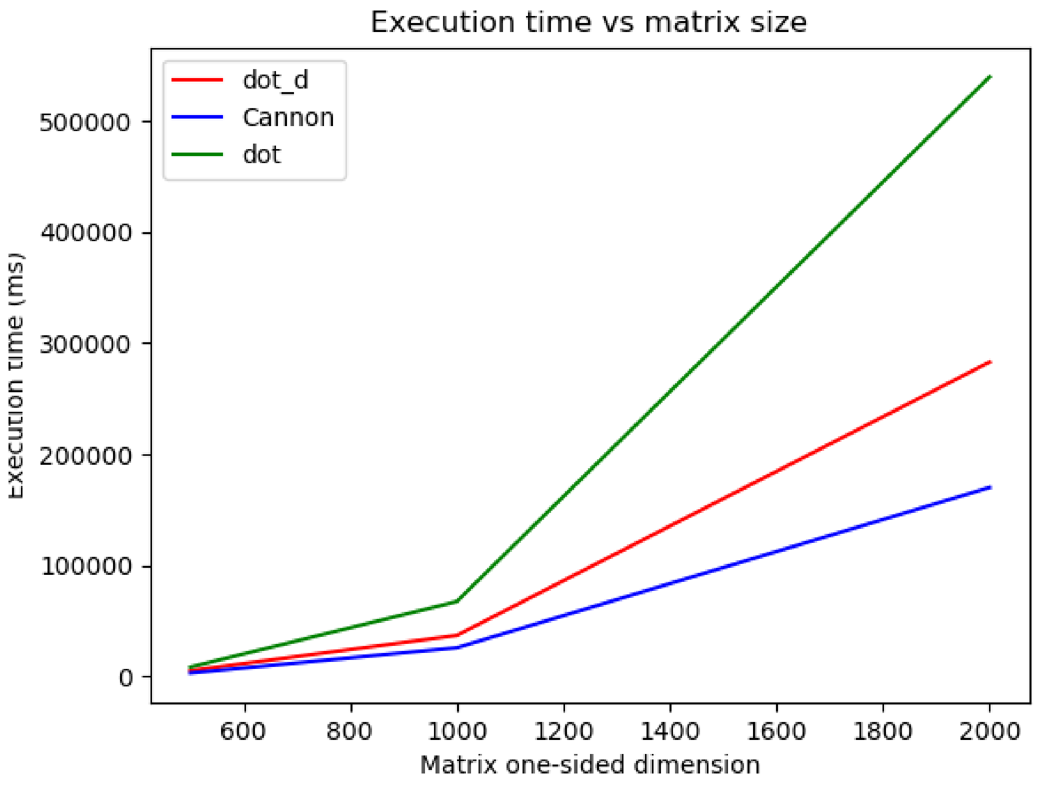


Figure 5.1. Preliminary Test Results - Raw timings

tests were performed in a shared memory environment, and thus could be contested, we believe that the requirement of communicating through the TCP/IP layer in HPX means that the results are approximately equivalent to running in a fully distributed environment. In the tests we ran, we found that the Cannon product was always the fastest, achieving a speed-up relative to the serial of between 2.6 and 3.2. Cannon consistently outperformed dot_d. As we mentioned before, we believe that the lack of a reduction step across tile rows in the output matrix substantially aided the Cannon product, as well as its use of futurization of the fetches. Of course, further testing is required to confirm these results.

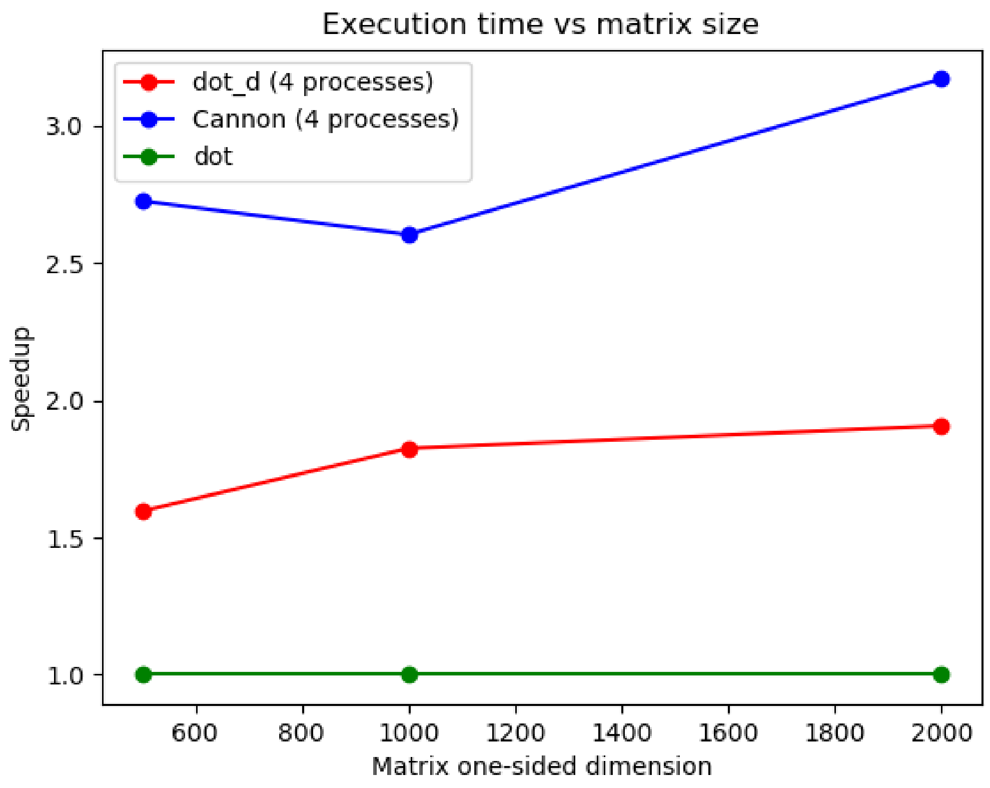


Figure 5.2. Preliminary Test Results - Speedup

5.2. Future work

Although these results are only preliminary, they are encouraging, and suggest that our work on this project has meaningfully advanced the state of the Phylanx project. In order to advance closer to having a robust system for performing linear algebra for ML applications, or otherwise, we have several new objectives. The main one is in expanding our lineup of distributed primitives, to allow for something of a "distributed computational basis" for ML applications. This necessitates functionality like distributed add/map, distributed file read/write, and other linear algebra primitives like matrix inverse, and matrix factorization. We also would like to be able to use this system to aid in providing testing data for the tiling optimization facet of Phylanx. The capacity to write many different algorithms with different tilings, and extract performance data will help give intuition, or experimental validation, in the development of a static tiling optimizer.

Appendix A. Linear Regression

Linear regression is a method of fitting a model to a data set in order to obtain a linear model minimizing error between observed and fitted values based on a number of input variables, known as regressors. The goal is to obtain a model of weights x, so that any input row, or set of input rows for which you have regressor data can be used to predict the value of another, related variable, the value of which you do not have access to, or which it may otherwise be too costly to obtain. A simple example is if you wanted to determine the likely productivity of a corn field. If you can create a model based on rain, fertilizing, or other variables, you can not only predict the productivity of the field at harvest, but also use the coefficients obtained in the model to compare the relative contribution of regressors on field productivity.

Appendix B. Copyright Information

References

[1] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck, S. Brandt, and H. Kaiser. Asynchronous execution of python code on task-based runtime systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 37–45, Nov 2018.

[2] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. HPX – An open source C++ Standard Library for Parallelism and Concurrency. In *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo'17)*, page 5, 2017.

[3] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. High performance smart expression template math libraries. In *2012 International Conference on High Performance Computing Simulation (HPCS)*, pages 367–373, July 2012.

[4] Weile Wei, Maxwell Reeser, Hartmut Kaiser, Adrian Serio, Avah Banerjee, and Rod Tohid. Distributed object abstraction in hpx, May 2019.

[5] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 1–15, Santa Clara, CA, July 2015. USENIX Association.

[6] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973, May 2019.

Vita

Maxwell Reeser received his Bachelor of Science in Computer Science in May 2018, from Louisiana State University. From 2018 to 2020 he pursued his MS in Computer Science at Louisiana State University while holding a student research position with the Ste||ar Group at LSU's Center for Computation and Technology where he was able to work on distributed data structures and primitives for the Phylanx Distributed Array Processing Toolkit project. He was also able to work on a variety of projects with Radiance Technologies including Python software development in graph algorithms and PySpark.