Parallel and Multi-Locale Computing in Scientific Computing Algorithms using Chapel

Brian Becker

May 22, 2020
Version: Final Draft
Advisor: David G. Wonnacott

Abstract

Parallel and clustered implementations of scientific computing algorithms, while fast, can be very difficult for a human to parse. This is because the industry standard method of writing parallel code is to use complicated libraries that extend existing languages such as OpenMP C. Sometimes a programmer even needs more than one library at a time. Attempts have been made to avoid this problem via automatic parallelizing compilers that use mathematical abstractions of loops in code to recognize dependencies and generate parallel code accordingly. Unfortunately, these compilers have their limitations. We propose that a solution to this problem could come in the form of the Chapel programming language. Chapel is a language developed by Cray Inc. that was specifically designed to be used for parallel and clustered programming. Because of this design choice, all the parallel features in Chapel are standard, requiring no additional libraries. In this thesis, we review existing approaches to parallelizing code, using BLAS as our primary example, and present how we might use Chapel to create fast code that humans can understand.

Contents

Introduction

1

There are many subroutines that are encountered so often in scientific computing that optimizing them to run as quickly as possible can make a significant impact on the efficiency of an experiment. This is especially important considering that these algorithms often have nontrivial complexities that can quickly become unreasonable as the scale of the problem increases. The naive approach to matrix multiplication for example has a complexity of $O(n^3)$ (actually this can be improved to $O(n^{2.376})$ using block matrices and reducing the number of multiplications (Coppersmith and Winograd 1990)), which can be a significant time sink for algorithms that have to compute a large number of multiplications on high-dimensional matrices. Unsurprisingly, there have been efforts to use parallel and multi-locale computing to assuage the cost of these subroutines. One example of these efforts is the development of the Pluto compiler, which translates sequential C programs with nested loops into more efficient parallel and multi-locale programs in OpenMP (Bondhugula et al. 2008), which is a C library that supports parallel computing accross multiple cores. Additionally, there was the development of a compiler that used so-called Declarative Transformations in the polyhedral model used by Pluto (Zinenko et al. 2018). This compiler takes an approach of trying to recognize specific known kernels and optimizing them at a low level. The amount that these tools aid in the production of fast code cannot be overstated. They are extremely general and automatic; all the programmer needs to do is write simple code and the compiler magically generates a fast version. What's more, these compilers give us benchmarks to compare against when writing our own parallel code. However, automatic compilers are not perfect. While the black box they provide is convenient, it makes it difficult for programmers to interact with their strategies when trying to further speed up code. This also means that it takes a significant amount of work to implement and improve upon these compilers.

A recent development in the field of parallel computing is the advent of the Chapel programming language. The general philosophy of Chapel is that it aims to be an interpretable, accessible language that provides a full suite of parallel and multi-locale features (Balaji 2015). The features it offers are comprehensive enough that in principle, one could write a simply expressed algorithm in Chapel that could take full advantage of different multi-core architectures across a cluster of computers. This is different from previously existing methods to accomplish this in that those methods require the user to learn libraries developed for languages that previously existed. For example, to write a C program that takes advantage of multiple cores and clusters, one must learn the OpenMP library and the MPI library. In essence, there are a lot of moving parts to keep track of. Because Chapel avoids this by incorporating all these features into the base language, it lends itself well to the problem of optimizing algorithms in scientific computing. The hope is that hand-tuning these optimizations with Chapel would lead to programs that are both as fast as those yielded by the compilers that parallelize automatically, and that also use parallel features in a way that is legible to the average programmer. In this thesis I will provide an exposition of the background in this field, as well as a strategy for a Chapel implementation of an optimized general matrix multiplication algorithm.

2 Literature Review

2.1 A Set of Level 3 Basic Linear Algebra Subprograms

We begin with some background on the subroutines we will be trying to optimize. The main program we are interested in is called GEMM, which stands for General Matrix Multiplication, or alternatively General Matrix-Matrix products. GEMM was proposed in a set of important subroutines by (J. J. Dongarra et al. 1990) known as level 3 BLAS, or Basic Linear Algebra Subprograms. BLAS is an evergrowing library of such subroutines that was first proposed in 1973. The idea behind BLAS is that the linear algebraic programs it contains are useful accross a wide range of software. Therefore it would be efficient, both for the implementation and clarity of such software, to have the BLAS subroutines be standardized and optimized.

The noteworthy thing about level 3 BLAS, as compared to the previous levels, is that level 3 BLAS was designed with parallel computing in mind. For example, computing the product of matrices A and B can be reduced to the smaller problem of computing the products of different dot products of rows of A and columns of B. The computation of these products can be run in parallel, and across different machines. Of course, before we can think about how BLAS is implemented we must first address the exact subroutine we are trying to compute. We note that the BLAS level 3 paper by (J. J. Dongarra et al. 1990) does not provide implementations for any subroutine. Rather, it assembles a compendium of deemed important linear algebra subroutines involving matrix-matrix computations. Since the publication of this paper, this compendium has indeed become a recognized industry standard. The Matrix-Matrix products accounted for in the set are presented as follows:

$$\begin{aligned} C &\leftarrow \alpha AB + \beta C \\ C &\leftarrow \alpha A^T B + \beta C \\ C &\leftarrow \alpha AB^T + \beta C \\ C &\leftarrow \alpha A^T B^T + \beta C \end{aligned}$$

such that α and β are scalars, and A, B, and C are matrices. We will focus primarily on the first of these programs. We next examine a sequential implementation of GEMM in C from the PolyBench project at the Ohio State University.

```
// => Form C := alpha*A*B + beta*C
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
for (i = 0; i < _PB_NI; i++) {

    for(j = 0; j <_PB_NJ; j++)
        C[i][j] *= beta;

    for(k = 0; i < _PB_NK; k++) {
        for(j = 0; j <_PB_NJ; j++)
            C[i][j] += alpha * A[i][k] * B[k][j]
    }
}
```

Fig. 2.1: GEMM Implementation

This implementation works by iterating through the rows of C and multiplying each element by β. The kernel also computes $A_{i*} \cdot B_{*j}$ and adds it to $C[i][j]$. We trace the algorithm on a small example. We aim to compute:

$$\underbrace{100}_{\alpha} \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 1 & 3 \\ 1 & 5 \end{bmatrix}}_{B} + \underbrace{5}_{\beta} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{C}.$$

We first multiply the 1st row of C by $\beta = 5$. We then add $\left(100 \begin{bmatrix} 1 & 2 \end{bmatrix}\right) \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ to $C[0][0]$, and $\left(100 \begin{bmatrix} 1 & 2 \end{bmatrix}\right) \begin{bmatrix} 3 \\ 5 \end{bmatrix}$ to $C[0][1]$. After this first iteration, we have

$$C = \begin{bmatrix} 305 & 1300 \\ 0 & 1 \end{bmatrix}.$$

We then repeat this exact process, this time updating the second row of C by first multiplying by β and then adding the appropriate dot products to each element in the second row.. This yields a final result of

$$C = \begin{bmatrix} 305 & 1300 \\ 700 & 295 \end{bmatrix}.$$

This is the most naive way to implement matrix multiplication, and we can see that there is inefficiency in this algorithm as calculations that do not at all depend on earlier calculations still wait for those earlier calculations to terminate before they can be processed. We'll see in upcoming sections different ways to get around this obstacle.

In general, as we have mentioned, BLAS provides a compendium of problems that beg to be approached with parallel design. One future direction I can imagine for BLAS is to consider general tensor-matrix and tensor-tensor operations (I learned about these operations in a talk given by Elizabeth Newman at Haverford on December 9, 2019). In particular, a tensor is just an array with arbitrarily many dimensions. For example $\mathcal{X} \in \mathbb{R}^{3\times 3\times 3}$. would be a cubic tensor. The two operations of note are the mode-k product, which is a tensor-matrix operation, and the tensor m-product which is a tensor-tensor operation. The mode-k product essentially computes k matrix multiplications (1 for each *layer* of the tensor with the given matrix), and the m-product uses linear transformations to consider tensors as matrices of *tubes* (vectors, matrices or tensors themselves), and proceeds using using traditional matrix multiplications. In essence these tensor products are just a series of matrix products, just as matrix products are a series of dot products. Because these products do not depend on one another, it makes sense that they could be parallelized.

It is exciting to think about the future directions BLAS might take, but for the rest of this thesis, we will focus on matrix-matrix operations as there is far more existing literature on the topic.

2.2 Numerical Linear Algebra Algorithms and Software

This (Jack J. Dongarra and Eijkhout 2000) paper covers various improvements and speedup techniques that can be made for algorithms involving matrices. This paper shows us why BLAS is important by reducing other problems in linear algebra to a series of BLAS operations. The primary example in the paper is Cholesky factorization. The goal of Cholesky factorization is to factor an symmetric positive definite matrix A, as $A = U^T U$, where U is upper triangular. It turns out that one way to write this algorithm is to iterate through the diagonal of U and compute the row and column that passes through that diagonal. In doing this, each step is reduced to a series of Matrix-Vector multiplication systems of equations. Solving such systems is a BLAS level 2 operation, so as we hoped, Cholesky factorization boils down to BLAS. The paper then describes a block version of the algorithm that operates similarly but computes whole blocks of U at once instead of just a row and column. This uses BLAS level 3 operations, and leads to significant speedups of the algorithm (a factor of 5 megaflops).

The paper then goes on to discuss various design choices that can be made in implementing BLAS. This is of interest to us, as we hope to implement our own BLAS kernels in Chapel. The first principle is to try and use elements of the matrix that are close together in memory. One provided reason for this is to make use of *cache lines*. The cache is a smaller, faster memory block than main memory. It is this speed that we hope to take advantage of. Say a matrix A is too large to fit in the cache (this is not uncommon when dealing with large amount of data), so it is instead stored in main memory. Say also that I need to access each element of A to multiply it by B. When I access each element, it is copied into the cache, which is a smaller, faster memory block than main memory. The key factor is that in fact, when I access an element a_{ij} of A, a number of consecutive elements (say a_{ij+1} through a_{ij+k}) are also copied into the cache in a *cache line*. This way, when I next access those elements, I need not look back in main memory. If I were to access random elements of A, or even if I were to access elements in some order that avoids using consecutive matrix entries, I would not be able to take advantage of cache lines. The other main principle to speed up BLAS is cache reuse. We want to use the data in the cache as much as possible before loading in new data. To this end, BLAS algorithms can be partitioned into computations on different portions or *tiles* of the given matrix or vector. We will see how this notion of tiling becomes more formal later in this thesis.

2.3 A Practical Automatic Polyhedral Parallelizer and Locality Optimizer (Pluto)

Automatic parallelization is a very attractive option as it would allow programmers to write clear and concise code that would still take advantage of parallel and multi-locale features once compiled. In implementing such a compiler the key is to think about how to process nested loops, as these comprise the majority of running times for many computationally expensive algorithms. The Pluto compiler as detailed in (Bondhugula et al. 2008) attempts to tackle this problem. The general strategy is to think of each computation of the body of a given nested loop as a point in the loop's overall iteration space, which is called the *polyhedron*. The general idea is that each index variable in an n-nested loop can be thought of as an axis in $\mathbb{Z}^n$. The polyhedron is just the set of points in $\mathbb{Z}^n$ that correspond to valid loop indices. Once the loop is abstracted to this mathematical model, optimizing the code happens in three steps.

1. First, the polyhedron is analyzed for dependencies. A dependency is a connection between two points in the iteration space to indicate that one relies on the other to occur first. This is important, because we wouldn't want to run two dependent sections of code simultaneously.

2. After this analysis is complete, the polyhedron is *tiled*. That is, the iteration space is partitioned into different portions that can be executed on different processors.

3. Finally, this transformed expression of the code is translated into OpenMP C, which is a C library that allows for parallel computing.

We begin our analysis of this process by presenting the polyhedral model presented in the paper.

Definition 1. For some $\vec{a} \in \mathbb{Z}^n$, an *affine hyperplane* is the set of vectors $\vec{x} \in \mathbb{Z}^n$, such that $\vec{a} \cdot \vec{x} = 0$.

A hyperplane is essentially a generalization of a linear function in n variables. For instance a hyperplane with 2 variables is a line, and one with 3 variables is a plane. To explain why we can define a hyperplane with a single vector, consider some line $y = mx$. We can write this as $mx - y = 0$, which is equivalent to $\begin{bmatrix} m \\ -1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = 0$. Here the $\vec{a}$ is the vector $(m, -1)$. The generalization of this is just to have a sum with n terms, one for each variable, which can be encoded by the constant vector $\vec{a}$ and the vector of variables $\vec{x}$.

Definition 2. For some matrix $A \in M_{m\times n}(\mathbb{Z})$, a *polyhedron* is the set of $\vec{x} \in \mathbb{Z}^n$ such that $A\vec{x} + \vec{b} \geq \vec{0}$. If the polyhedron is a bounded set, it is also a *polytope*.

As an example, we solve the following inequality:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

This multiplies out to the following system of inequalities.

$$x_1 + x_2 \geq 2 \qquad x_2 \geq 3$$

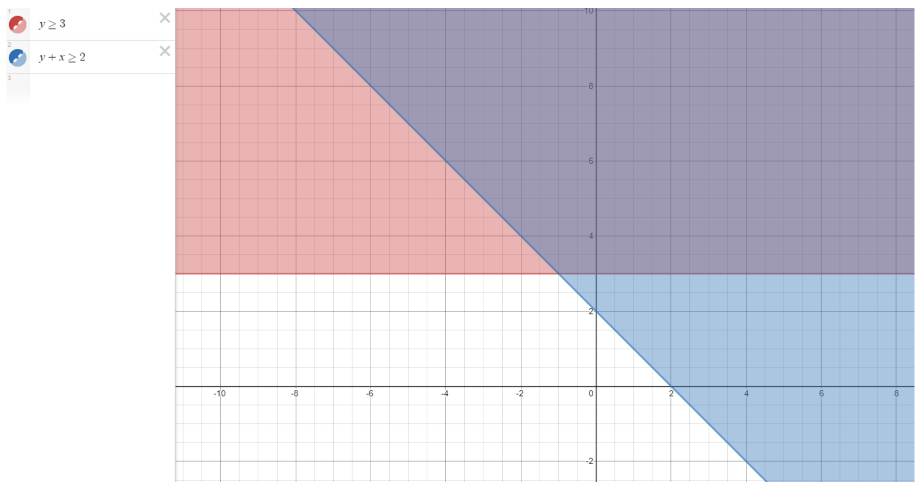

Fig. 2.2: The purple region is the polyhedron. Note that it is *not* a polytope.

We can see the solution set to these equations expressed by the following graph.

As it happens, if we are given a statement S within some nested loop in some program P such that the bounds of the loops are defined in terms of existing variables and outer loops bounds, we can associate a polytope with S. We call this polytope $\mathcal{D}_S$. This polytope is defined by considering the *iteration vector* $\vec{i}$ of S. This vector defines which dynamic instance of S is being executed, and $\mathcal{D}_S$ is defined by $A\vec{i} \geq 0$. Where A appropriately introduces constraints on $\vec{i}$. We consider the following example to help with intuition.

```
for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
        S1: A[i,j] = x[i]+y[j];
```

$$\mathcal{D}_{S1} : \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ N \\ M \\ 1 \end{bmatrix} \geq \vec{0}$$

If we multiply out $\mathcal{D}_{S1}$, we see that it implies that $i \geq 0, j \geq 0, i \leq N-1$ and $j \leq M-1$, which are exactly the constraints of the nested `for` loop above. As expected, this construction does define a polytope. Namely, the rectangle on the ij-plane with width N and height M whose lower left vertex is at (0,0). Moreover, each point in the polytope corresponds to a specific instance of $S1$ within the iteration space of the `for` loop.

The next step in constructing the polyhedral model is to create the Data Dependence Graph for our program.

Definition 3. Given a program P, with statements $S_1, \ldots, S_k$, the *Data Dependence Graph* or (DDG) is a graph (V, E), where V is the set of statements, and an edge (S_i, S_j) is in E iff there is at least one dynamic instance where S_j depends on S_i.

To illustrate this concept we consider the following extension to our above example.

```
for(i = 0; i < N; i++)
    for(j = 0; j < M; j++)
        S1: A[i,j] = x[i]+y[j];

for(p = 0; k < M; i++)
    for(q = 0; l < N; j++)
        S2: z[p*N+q] = A[q,p];
```

Here, the very simple DDG would have 2 nodes, one for each of S_1 and S_2. It would also have an edge from S_1 to S_2. It turns out that we can associate a polyhedron with each edge in the DDG. For some edge e, this polyhedron is appropriately denoted the *dependence polyhedron of* e, and is expressed $\mathcal{P}_e$.

So for example, the dependence polyhedron of the edge in our example would be

$$\mathcal{P}_e : \left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \end{array}\right] \begin{bmatrix} i \\ j \\ p \\ q \\ N \\ M \\ 1 \end{bmatrix} \begin{array}{c} \geq 0 \\ \geq 0 \\ \geq 0 \\ \geq 0 \\ \hline = 0 \\ = 0 \end{array}$$

We see that the constraints this polytope imposes on the indices parallel the constraints imposed by each statement's individual polytope. The difference is that this polytope adds the constraint that $i = q$ and $j = p$. This is because it is when these conditions are met that S_1 writes the specific data that S_2 needs.

The big result here is that we can use these dependence polyhedrons to tile the iteration space of our programs. Intuitively, a tiling is a way to break up the execution of a program into discrete chunks that do not depend on each other. This would allow us to execute the individual tiles in parallel which would speed up the runtime of the algorithm. The general mathematical notion of a tiling over an affine space is a change of basis. Given a polytope, changing basis partitions it based on the new axes. In our case, the new basis is a new set of iterators that encode how we move through the iteration space.

The way this happens mathematically, is that an affine transformation is associated with each iterator. These transformations have to satisfy a mathematical condition derived from $\mathcal{P}_e$. In particular, the difference of the transformations of iterators whose statements have a dependence must be non-negative. Together these transformations constitute the change of basis to a new set of iterators.

The Pluto compiler can generate these tilings (just polytopes under the new basis), and subsequently generate the corresponding parallel code.

At the end of this paper, results of experiments using the Pluto compiler are presented. In our work, we could attempt to replicate some of these experiments in Chapel and compare our results to these results. We could also write and run comparison tests of our own. Ideally, we will show that the Chapel can provide a readable customizable alternative to parallel programming that can compete with Pluto.

2.4 Declarative Transformations in the Polyhedral Model

While automatic parallelization is a powerful concept, and while the Pluto compiler is a useful tool, it still has its drawbacks. For instance, the polyhedral representation of nested loops is a somewhat limited concept that cannot be applied to all parts of any program. As a result, a lot of code is still hand tuned for optimization, which can be a lengthy and challenging process. Additionally, because these improvements are often implemented in somewhat inaccessible libraries like OpenMP, they can often be very difficult to interpret. A compiler is proposed in (Zinenko et al. 2018) that combines the polyhedral technique with more classical compiler optimization techniques. Such classical techniques include, for example, the recognition of abstract syntax trees within a program that lend themselves to optimization.

Polyhedral compilers in part function by creating schedules for the execution of the different parts of their iteration spaces. These schedules are represented in schedule trees, which have the following types of nodes (Zinenko et al. 2018):

- *domain* - this is the representation of the iteration space; the polyhedron. This node appears once at the root of the tree.

- *band* - this is a partial schedule that represents the iteration of some number of loops

- *filer* - these nodes restrict their children to certain parts of the polyhedron.

- *sequence* - this forces its children to occur in order.

- *extension* - adds commands that did not previously exist in the code.

These schedule trees seek to reduce reliance on scheduling via affine functions.

The idea is that just as classical compilers can optimize based on the abstract syntax tree of their programs, polyhedral compilers can optimize based on the schedule trees they produce. This is achieved by an implementation of specific "tree-matchers" which recognize schedule trees and replace them with versions that are optimized for parallel execution.

Another tactic of this compiler is to make use of Polyhedral relation matchers. Polyhedron relations are defined by the user and are essentially a list of properties that a polyhedron must satisfy. Once these relations are defined, the compiler employs an algorithm to match them in the provided code and optimize from there. In particular, the compiler can search through the domain of the schedule tree and create different bands for each pattern in matches. The paper then goes on to present an example of these relation matchers that we will outline here. This is because the example they provide is on a GEMM-like kernels so it is particularly relevant to us.

To qualify for a transformation, the GEMM-like kernel must satisfy:

1. There are three nested one-dimensional for loops.

2. At the inner-most point, there is a statement

$$C_{\pi C(IJ)} = E(A_{\pi A(IK)}, B_{\pi B(KJ)}, C_{\pi C(IJ)})$$

   Here, A, B, and C are tensors and π defines which element of them is being accessed. E is some operation that reads from each tensor at least once. For example, in the case of GEMM, E is a multiplication of the elements from A and B and an addition of the element from C.

Once such a pattern is matched, the compiler tiles the schedule tree so as to be able to run the GEMM-like kernel in parallel. The writers of this paper argue that by defining matchers explicitly, they save a lot of difficult work in the compiler. They also argue that their method using matchers and builders requires less code than existing methods (such as the Polly loop optimizer).

2.5 Computing BLAS Level-2 Operations on Workstation Clusters Using the Divisible Load Paradigm

In addition to parallelizing the GEMM kernel, we also aim to optimize our code to run on a cluster of computers. If done efficiently, this could theoretically increase the overall efficiency of the kernel by a significant margin. The challenges that come with undertaking this task are related to what's known as *communication delay*. Communication delay is the phenomenon that transmitting data and instructions across different nodes in a computer clusters takes a certain amount of time. For example, if we were to run each tile of a GEMM polyhedron on a different node, the runtime of our algorithm would be dominated by communication delay. To solve this problem, we turn to existing literature on divisible load scheduling (Ghose and Kim 2005). This paper presents a mathematical framework both to describe the problem of load scheduling, as well as to present equations for the optimal load distribution. The example of choice in this paper are BLAS level-2 operations, (i.e Matrix-Vector multiplication). We aim to understand these results and gain the framework we need to apply this paradigm to BLAS level 3.

We begin by presenting the problem. The first thing we must consider is the architecture of our computer cluster. We assume that we have $p + 1$ separate processors, $P_0, P_1, \ldots, P_p$, that are all connected by some bus. P_0 is the root node, so it receives the initial data, distributes tasks, and stores the final result. This architecture is referred to as a $p-$processor network. The network is assumed to be *homogenous*, which just means that each processor is identical in memory and speed.

At this point we show how to decompose the problem of Matrix-Vector multiplications. If A is an $m \times n$ matrix, and x is an $n \times 1$ vector, then calculating the product Ax takes mn multiplications and $m(n-1)$ additions. This is because dotting x with a row of A takes n multiplications, (one for each element of x), and $n-1$ additions, (it takes $n-1$ additions to add n numbers). Because multiplying A and x is achieved by computing one such dot product for each of the m rows of A, we get mn and

$m(n-1)$ multiplications and additions respectively overall. This realization about the different dot products required to calculate Ax yields a natural way to break up the problem. In particular we notate

$$A = (A_1, A_2, \dots, A_m)^T$$

where A_i is the ith row of A. We can now express our matix-vector product as

$$b = Ax = (A_1 \cdot x, A_2 \cdot x, \dots, A_m \cdot x)^T$$

We now have a natural way to divide the work of this multiplication among the processors. Namely, we define the *load* of a processor P_i to be the m_i elements of b it is responsible for calculating. This implies the constraints

$$m_0 + m_1 + \cdots + m_p = m \qquad m_i \geq 0.$$

From this, we are motivated to define a *load partition* to be these row-assignments which we can represent by the tuple $(m_0, m_1, m_2, \dots, m_p)$.

At this point, we present some useful reference variables included in the paper that we will use to denote how much time the different calculations are taking.

T_{cm} = the time it takes a processor to communicate one element of a matrix or vector to another processor.
T_{cp}^a = the time it takes a processor to compute one addition.
T_{cp}^m = the time it takes a processor to compute one multiplication.
$T_{cp} = nT_{cp}^m + (n-1)T_{cp}^a$ = the time it takes a processor to compute one $n-$dimensional vector dot product.

The paper also introduces normalizing constants z and w which normalize the time for data transmission and arithmetic calculation respectively. So for instance wT_{cp} is the normalized time it takes to compute a dot product. At this point, the paper asserts a linear relationship between the amount of data transmitted, and the time it takes to transmit it. Particularly, it asserts that the time it takes to transmit m data points is mzT_{cm}. We hope to examine whether this linear relationship in fact holds.

The final step in defining the problem is to define the runtime and speed up of the processor network on a given task with distributed loads. If t_i is the time it takes P_i to complete its load, then the overall time of the task is defined

$$T_p = max(t_0, \dots, t_p).$$

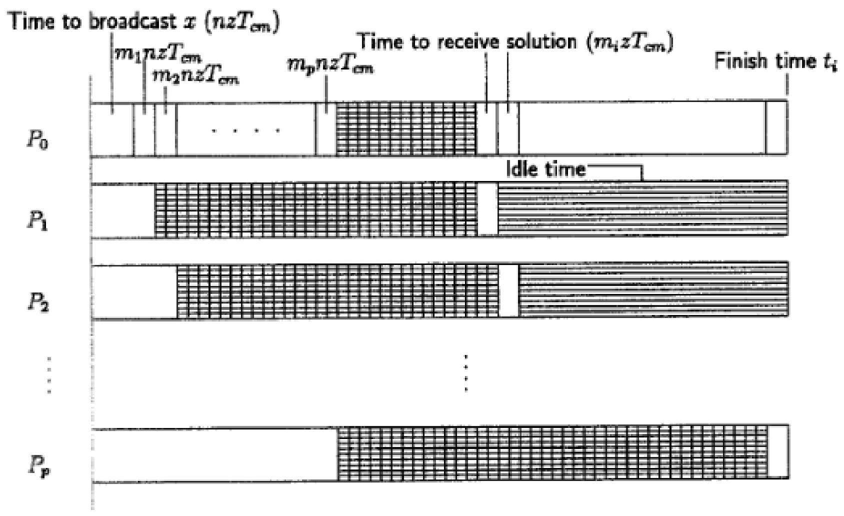

Fig. 2.3: Timing diagram: multicast communication (Ghose and Kim 2005).

The speedup is the ratio of the time it would take a 0-processor network (that is, a standard non-distributed calculation), to the time it would take the p-processor network. It is defined

$$S(0,p) = \frac{T_0}{T_p}.$$

At this point, the paper proceeds with a mathematical analysis of load partitions. In particular it examines the following 3 cases.

1. The input data is already stored on all processors.

2. The root broadcasts the input data to all processors.

3. The root multicasts only the data needed by each processor to each processor.

In each case, the paper explores what the optimal load partition would look like, and most interestingly examines under what conditions it is actually beneficial to add more processors to speed up the calculation. Throughout this, one might ask how it relates to GEMM. After all, GEMM is a matrix-matrix operation, not a matrix-vector one. However, matrix-matrix multiplication is just a series of matrix vector multiplications. In particular, the ith column of the matrix product AB is just A multiplies by the ith column of B. Therefore, if we understand how to distribute the n dot products of one matrix-vector product appropriately, we can perhaps apply the same (or a similar) paradigm to the n matrix-vector products it takes to calculate one matrix-matrix product. Getting back to the paper, we examine a diagram representing the load partition in the 3rd case above. In this model, each set of m_i rows is sent from P_0 to P_i. Once P_i receives the data it needs, it computes its m_i dot products and returns the results back to P_0. The goal is to choose $\vec{m}$ such that P_i finishes returning its data when P_{i+1} starts returning its data. If this is achieved then at all times while one processor is returning its solution, all subsequent processors are still working. Exactly how to achieve this is described by load distribution equations in the paper. These equations are quite technical so we will not include them here. The point is that they provide an ideal jumping off point in designing a way to distribute the different matrix-vector products of a matrix-matrix multiplication.

2.6 Performance Evaluation of BLAS on a Cluster of Multi-Core Intel Processors

This paper (Soliman and Sayed 2016) presents speedup results of running BLAS levels 1, 2, and 3 using many different architectures and techniques. They found that in the case of BLAS levels 1 and 2, it is actually most efficient to run a standard SIMD parallelization on one computer. This is because when trying to use multithreading or clusters, the communication delay dominates the runtime. However, they also found that for level 3 BLAS, using multithreading SIMD-blocking on multiple clusters indeed speeds up the algorithm by a factor of over 10. This is both because the data is large enough that the delay is worth it, and because they can use blocking techniques to avoid cache misses at scale. This is a promising result as far as our own experiments are concerned, as it shows that working with clusters is in fact practical for GEMM.

2.7 Programming Models for Parallel Computing

This chapter in (Balaji 2015) is a primer on the Chapel programming language. As mentioned earlier, we hope to be able to use chapel to write readable code with performance that can compete with the standing optimizing compilers.

"Chapel was designed by Cray Inc. for parallel computing at scale. It has a multi-thread execution model, and provides both low level features as well as the ability to abstract."

Chapel's Motivating Themes Chapel aims to support general parallel programming. The aim is that there should be no tasks where a user would have to switch back to using frameworks like MPI (a C++ library to write on multiple locales). Chapel aims to include all types of parallel computing. Not just data-parallelism or task-parallelism. Additionally, before Chapel, to fully take advantage of the parallelism that some hardware may be capable of, users would have to outsource to different tools in different occasions. For example, they might use OpenMP for loop parallelism across multiple cores, and MPI for executable-level parallelism. Chapel seeks to incorporate all of this in one convenient interface.

Chapel has a multithreaded execution model. The idea is that confining to a single thread can limit the creativity of programmers and prevent them from taking full advantage of parallel computing. The language also supports global-view data structure which are customizable. It starts with a single task, and branches. This is unlike SPMD (single program, multiple data) approaches.

As previously mentioned, Chapel institutes a multi-resolution design, allowing for different levels of abstraction. Because the high-level constructs are implemented in terms of the lower level ones, the user can choose what level of abstraction is fit for a particular task. The language also allows a customization of locality. That is, the user can choose where data is stored, and where commands are executed. This enables efficient lookups. Users can also customize locality without necessarily introducing a parallel computation (and vice-versa). Data-centric synchronization is supported. There is no reliance on "heroic compilation," but there is room for modular design. Low level experts can implement the parallel aspects of an algorithm, while other team members makes the high level contributions. We hope this design framework would allow us to implement our optimizations in an intuitive and efficient manner.

Basic Language Features -

Chapel has C-style syntax for the most part, (semicolons, curly braces, if/while). Declarations are more similar to ML (left-to-right). All types use 64 bits, but users can specify alternatives. E.g "int (8)" is equivalent to a "byte" in java. Records use local memory, classes use heap-allocated memory. Records are like C++ structs, and classes like Java classes. (Classes pass by reference, records do not).

Parallelism in Chapel is implemented through tasks. (Initiated with begin keyword). Sync keyword ensures tasks within it conclude before current task continues. Because the sync keyword is so absolute, it can sometimes work against overall efficiency. For this purpose Chapel has sync variables (var x: sync int) that store whether they are full or empty. Tasks accessing them will wait until they are full to read, or empty to write. Doing either operation inverts the state of the variable. But there are special methods that allow one, for example, to read a sync variable when full without setting it to empty.

Our main tools in Chapel will likely be custom iterators and domains. Custom iterators are essentially functions that return specific ranges that we can then iterate through with for loops. If we figure out an efficient way to iterate through matrices when computing GEMM for example, we can encode that in a custom iterator and then write a very clear for loop to compute the matrix product. (This is inspired by work presented in (Bertolacci et al. 2015)). Chapel Domains are data structures that store the ranges custom iterators can produce, (as well as standard Chapel ranges). As we would hope, Domains can be multi-dimensional.

We also hope to take advantage of Chapel's locality support. Chapel has a built in *Locale* class, that allows programmers store data and run code on different cores or machines. Ideally, we could use this feature to run algorithms efficiently on clusters of computers via our own implementation of the load distribution paradigms presented by (Ghose and Kim 2005), and the blocking techniques described in (Soliman and Sayed 2016).

Ideally, at this point we would provide an example comparing Chapel code for some subroutine to existing OpenMP or MPI code, but unfortunately, we were unable to find such an example. It it precisely for this reason that we hope to develop such code in the future. We think that Chapel would make it easy and elegant to implement speedup techniques like all those described in this thesis, while keeping code modular and readable.

3 Conclusion and Future Work

After reading through the literature we have found first of all, that there have been many strides and achievements in speeding up scientific computations through parallelization and locality. In our context, this began when the community recognized the value of the ubiquitous Basic Linear Algebra Subprograms (BLAS). The key insight in doing this was essentially that most scientific computations end up boiling down to the execution of different BLAS subroutines, therefore speeding up BLAS would speed up computations in the field generally. The specific important development for BLAS was paralellization. For instance, the compiler in (Zinenko et al. 2018) specifically recognizes GEMM-like code blocks, and translates them into parallel code. The Pluto compiler Bondhugula et al. 2008, while a bit more general, still focuses on the idea that the bulk of an algorithm's runtime happens in loops or nested loops. Bondhugula constructs the polyhedral model with this notion in mind, which causes the Pluto compiler to target GEMM-like code as well. Additionally, (Soliman and Sayed 2016) found that it is experimentally viable to run GEMM on a cluster of computers.

There are a few issues with these existing methods. One is that speeding up code does not end with the polyhedral model. As (Zinenko et al. 2018) points out, most critical sections of code are still optimized by hand using libraries like OpenMP or MPI. These libraries can be obtuse, and can produce ugly code that is all but unintelligible to the average programmer. The tiled code in (Bondhugula et al. 2008) for instance has almost three times as many for loops as the original code. There are also many seemingly random loop bounds and calls to odd statement functions which take a lot of work for a human to parse. Deciding whether this messy code even achieves the same result as the original expression would be an arduous task.

Our goal is to work towards solving this accessibility problem by developing GEMM, and possibly other BLAS kernels in Chapel. We will implement tilings based on the polyhedral model using Chapel iterators, and will use the built in *Locale* class to assign load distributions among different processors. As presented in (Balaji 2015), Chapel is specifically *designed* for the development of parallel and multi-locale code, so it shouldn't actually require too much code to implement these kernels. What's more, the abstractions and modularity provided by Chapel should make this code human-readable. If our kernels can compete with existing ones, this project could make the implementation of these subroutines accessible to more people in the field, and could open the door to future understandable implementations of other important algorithms.

Bibliography

Balaji, Pavan (2015). *Programming Models for Parallel Computing*. The MIT Press. ISBN: 0262528819, 9780262528818.

Bertolacci, Ian J., Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout (2015). “Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators”. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS ’15. Newport Beach, California, USA: ACM, pp. 197–206. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751226. URL: http://doi.acm.org/10.1145/2751205.2751226.

Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan (2008). “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: ACM, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

Coppersmith, Don and Shmuel Winograd (1990). “Matrix multiplication via arithmetic progressions”. In: *Journal of Symbolic Computation* 9.3. Computational algebraic complexity editorial, pp. 251–280. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(08)80013-2. URL: http://www.sciencedirect.com/science/article/pii/S0747717108800132.

Dongarra, J. J., Jeremy Du Croz, Sven Hammarling, and I. S. Duff (Mar. 1990). “A Set of Level 3 Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 16.1, pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: http://doi.acm.org/10.1145/77626.79170.

Dongarra, Jack J. and Victor Eijkhout (2000). “Numerical linear algebra algorithms and software”. In: *Journal of Computational and Applied Mathematics* 123.1. Numerical Analysis 2000. Vol. III: Linear Algebra, pp. 489–514. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00400-3. URL: http://www.sciencedirect.com/science/article/pii/S0377042700004003.

Ghose, D. and Hyoung Joong Kim (2005). “Computing BLAS level-2 operations on workstation clusters using the divisible load paradigm”. In: *Mathematical and Computer Modelling* 41.1, pp. 49–70. ISSN: 0895-7177. DOI: https://doi.org/10.1016/j.mcm.2004.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0895717705000312.

Soliman, Mostafa and Fatma Sayed (Mar. 2016). “Performance Evaluation of BLAS on a Cluster of Multi-Core Intel Processors”. In:

Zinenko, Oleksandr, Lorenzo Chelini, and Tobias Grosser (Dec. 2018). *Declarative Transformations in the Polyhedral Model*. Research Report RR-9243. Inria ; ENS Paris - Ecole Normale Supérieure de Paris ; ETH Zurich ; TU Delft ; IBM Zürich. URL: `https://hal.inria.fr/hal-01965599`.