Using Chapel to Implement Dense Linear Algebra Libraries

Joe Elizondo and Samuel Palmer
Department of Computer Science University of Texas at Austin
Email: {elizondo, spalmer}@cs.utexas.edu

May 16, 2010

Abstract

This paper evaluates the parallel global-view language Chapel's ability to implement dense linear algebra libraries. Both programmability and performance characteristics at the current stage of Chapel's implementation are considered but since many of Chapel's compiler optimizations are not implemented our emphasis is on programmability. Our conclusion is that in its current development state Chapel lacks the data locality features required to implement efficient dense linear algebra operations but should be considered again once the complete language and compiler optimizations are implemented.

1 Introduction

For the scientific computing community interest in fast linear algebra libraries is alive and well because scientists are always seeking to reduce the computation time required to solve large problems. The authors of these libraries are always looking for optimizations in software, compilers, and hardware to make their libraries faster. To this select group of scientific library developers a new partitioned global address space language like Chapel is of interest for a number of reasons. One reason is the possibility for Chapel to have dense linear algebra functions built in that allow the programmer to write code quickly and exploit parallelism with little effort. If so then the question becomes performance. How do these operations compare to the performance achieved by their current library? Otherwise library developers might see an opportunity to implement a library on this new platform. Programming a parallel dense linear algebra library in a global-view language is presumably easier than using a local-view language like MPI combined with C++. In addition, the most popular parallel dense linear algebra libraries, e.g. ScaLAPACK, are surprisingly inefficient and a library that is properly implemented can outperform these libraries without many additional optimizations[5]. If a library written in Chapel can achieve performance comparable to ScaLAPACK then ease of programming may make it preferable in cases where a matrix operation does not already exist in the library and the problem size is such that any additional computation time is negligible. For library developers with no formal introduction to Chapel all of these scenarios are possible and make experimenting with the language worthwhile.

For the Formal Linear Algebra Methods Environment (FLAME) group at the University of Texas at Austin, Chapel poses interesting possibilities. Much labor has gone into writing parallel linear algebra libraries using MPI and C (PLAPACK[7]) and currently MPI and C++ (Elemental[5, 6]). These libraries were built using matrix theory and methodology that results in some of the fastest linear algebra algorithms available. These libraries, however, are far from simple and if a new operation needs to be introduced a user would need a decent amount of experience and understanding of collective communication with MPI in order to successfully add the operation to the library. The FLAME framework and API, however, is one of the cleanest ways to express dense linear algebra operations. Matrix indices are abstracted away and more attractive methods are provided to partition matrices for computations. libFLAME already lends itself to parallelization through the use of matrix partitioning so it seems natural to implement it in a parallel language like Chapel. The question then becomes: is it possible to use the FLAME approach for deriving algorithms and the FLAME/C API with Chapel to write a parallel linear algebra library? Finding the answer to this question is the goal of this work. We were informed by Chapel developers from the start that the compiler was not optimized so achieving comparable performance would not be possible and competing against any benchmark (including ScaLAPACK) would not be valuable. Despite this we set out to learn, under the assumption of a compiler that generates optimized code, whether Chapel presents a good layer of abstraction for parallel dense linear algebra algorithms; whether it allows us to express the formalisms of the FLAME methodology; whether it allows us to express the necessary distributions required to partition a matrix over a mesh of processors for parallel computation; and whether it allows us to express the communication required to perform matrix operations on matrices that have been partitioned across many processors. We attempt to determine if Chapel satisfies these conditions by implementing several variations of general matrix-matrix multiply (GEMM) algorithms from libFLAME and a scalable matrix multiplication algorithm. The results of our findings are reported in this paper. Section 2 gives an overview of the FLAME API. Section 3 discusses our Chapel shared memory implementation of a matrix multiplication algorithm from libFLAME. Section 4 discusses our attempt to implement a blocked multi-locale matrix multiplication algorithm in Chapel. Section 5 discusses our Chapel implementation of a scalable matrix multiplication algorithm called SUMMA and Section 6 summarizes our conclusions.

2 FLAME Overview

We now discuss the FLAME API [10] and the motivation for implementing parallel algorithms in Chapel via the API. The algorithms used in the FLAME libraries are derived using a formal methodology and verification process. They are expressed using a notation that applies the mathematical specification for the routine in a step by step manner that is easy to visualize and understand. The API was designed so that the clarity of this derivation process is not lost when notation is translated to code form. Using the API the algorithm can be coded using a high level of abstraction that hides the indices that would usually be involved when coding the algorithms. The code partitions and repartitions matrices while performing the mathematical computation in the same manner expressed by the FLAME notation. This high level of abstraction makes it easier to think about different variations of the algorithms. The API also exposes critical sections in the code which provides hints to how they could be parallelized. Figures 1 and 2 show the similarities between FLAME algorithm notation and code expressed in the FLAME API.

Aside from the linear algebra routines that do the bulk of the computation, libFLAME's API is made up of numerous auxiliary functions, objects, type definitions, partitioning functions, validation functions, etc. All of this code had to be rewritten in Chapel. We were disappointed by Chapel's current level of support for calling C routines and manipulating C data. The insufficient support for external C code meant that we could not reuse any existing code and in many cases the changes that were made during translation were purely syntactic.

3 Shared Memory GEMM Implementation

3.1 Description of Algorithm

We implemented a shared memory GEMM algorithm, $C := AB + \hat{C}$, [8] which partitions A along its first dimension into a collection of vectors, a_i^T. The algorithm then performs a series of general matrix vector multiplication (GEMV) operations, $c_i^T := a_i^T B + \hat{c}_i^T$, to compute the result. In the FLAME methodology this partitioning is done by creating "views" of the matrix which allow the programmer to treat partitioned

```
Algorithm: $C = AB + \hat{C}$    (Variant 1, Unblocked)
Partition $C \rightarrow \left(\frac{C_T}{C_B}\right)$, $A \rightarrow \left(\frac{A_T}{A_B}\right)$
    where $C_T$ has 0 rows, $A_T$ has 0 rows
while $m(C_T) < m(C)$ do
    Repartition
    $\left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline c_1^T \\ C_2 \end{pmatrix}$, $\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline a_1^T \\ A_2 \end{pmatrix}$
        where $c_1$ has 1 row, $a_1$ has 1 row

    $c_1^T = a_1^T B + c_1^T$

    Continue with
    $\left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ c_1^T \\ \hline C_2 \end{pmatrix}$, $\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ a_1^T \\ \hline A_2 \end{pmatrix}$
endwhile
```

Figure 1: Unblocked GEMM algorithm expressed in FLAME notation

blocks as independent matrices or vectors[1]. In the sequential implementation of libFLAME this algorithm would partition the matrix to extract one vector from it at a time, multiply the other matrix by this vector, and merge the result back into the result matrix.

3.2 Implementation

In order to exploit parallelism our implementation had to deviate from the algorithm's sequential order of execution. Our implementation, instead, completes all of the partitioning before it performs any GEMV operations. After one matrix is completely partitioned into vectors we use the *forall* Chapel construct to assign the independent GEMV operations to different threads. Once the GEMV operations complete the results are merged back into the answer matrix. Completely partitioning the matrix before performing the GEMV operations requires more memory but allows us to execute the GEMV operations in parallel.

Figure 3 shows the relative speedup when multiplying two 1000×1000 matrices using our shared memory GEMM implementation as the number of threads increase. The speedup is measured relative to the output produced when the *--serial* flag is used to compile the same code. The figure shows that the shared memory implementation does in fact exhibit speedup as the number of threads increases. In general, the Chapel implementation of this algorithm compiled with the Chapel version 1.1 compiler is much slower than the sequential C libFLAME implementation. The average time it took our Chapel GEMM implementation to multiply two 1000×1000 matrices on a 16-way SMP compute node with AMD Opteron processors clocked at 2.3GHz was 4.7238 seconds while the average time it took the sequential libFLAME implementation to multiply two 1000×1000 matrices on the same hardware was 0.2658 seconds.

3.3 Issues

The initial goal did not involve rewriting the entire algorithm including the BLAS implementation of GEMV but upon a survey of Chapel's interoperability with C we concluded that it would be necessary since Chapel does not provide enough guarantees on the representation of data. Specifically, the standard BLAS GEMV requires array parameters to be passed as pointers which do not exist in Chapel. This forced us to write everything including a simplified version of the BLAS GEMV in Chapel. The immature support for interoperability with the C language causes us to lose the benefit of using highly optimized BLAS routines such as those found in the widely used GotoBLAS[4].

```
FLA_Error FLA_Gemm_nn_unb_var1( FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C )
{
  FLA_Obj AT,              A0,
          AB,              a1t,
                           A2;

  FLA_Obj CT,              C0,
          CB,              c1t,
                           C2;

  FLA_Scal_external( beta, C );

  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );

  FLA_Part_2x1( C,    &CT,
                      &CB,            0, FLA_TOP );

  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){

    FLA_Repart_2x1_to_3x1( AT,                &A0,
                        /* ** */            /* *** */
                                              &a1t,
                           AB,                &A2,        1, FLA_BOTTOM );

    FLA_Repart_2x1_to_3x1( CT,                &C0,
                        /* ** */            /* *** */
                                              &c1t,
                           CB,                &C2,        1, FLA_BOTTOM );

    /*------------------------------------------------------------*/

    /* c1t  = a1t * B + c1t    */
    /* c1t' = B' * a1t' + c1t' */
    FLA_Gemv_external( FLA_TRANSPOSE, alpha, B, a1t, FLA_ONE, c1t );

    /*------------------------------------------------------------*/

    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  a1t,
                            /* ** */           /* *** */
                              &AB,                A2,     FLA_TOP );

    FLA_Cont_with_3x1_to_2x1( &CT,                C0,
                                                  c1t,
                            /* ** */           /* *** */
                              &CB,                C2,     FLA_TOP );

  }

  return FLA_SUCCESS;
}
```

Figure 2: Unblocked GEMM algorithm implemented in C using the FLAME API

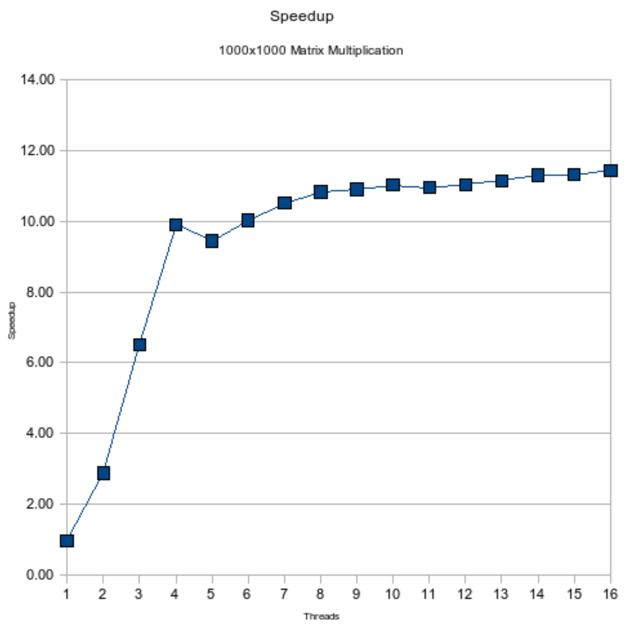

Figure 3: Relative speedup achieved as the number of threads increases in our shared memory implementation. Speedup is relative to the same Chapel code compiled with the *--serial* flag.

4 Blocked Multi-Locale GEMM Implementation

4.1 Description of Algorithm

Even though support for multiple locales is still in its early stages we decided to implement a blocked GEMM algorithm, $C := AB + \hat{C}$, that used multiple locales to evaluate the ease of reasoning about and programming with Chapel's data locality features. The blocked GEMM algorithm that we implemented partitions A into panels along the first dimension and B into panels along the second dimension. The algorithm then performs a series of general panel-panel multiplies, $C := A_i B_i + \hat{C}$. According to [8] this algorithm performs better than the other blocked GEMM algorithms in libFLAME.

4.2 Implementation and Issues

The largest roadblock that we encountered was that while the Chapel developers intend to add support for user defined domain maps there is currently only support for standard block distributions and standard cyclic distributions. These were inadequate for our needs so we implemented an algorithm that explicitly moved data between locales when necessary. Once the appropriate data was moved to a locale we made use of the *local* construct in Chapel to explicitly tell the compiler that all data references were local. The implementation of the *local* construct enables the compiler to perform performance optimizations but does not statically check if the data references will be local. In Chapel if a programmer makes a non-local reference inside a local block a runtime error will occur. The compiler should have enough information to be able to statically decide at compile-time if non-local references are made and give a compile-time error instead of a runtime error. The Chapel Language Specification 0.795 [3] does not in fact define the semantics of the *local* construct but it can be deduced from the discussion in [2] that it provides a boost in performance to explicitly assert that all references are local.

Using the current implementation of the Chapel compiler (version 1.1) our implementation induced

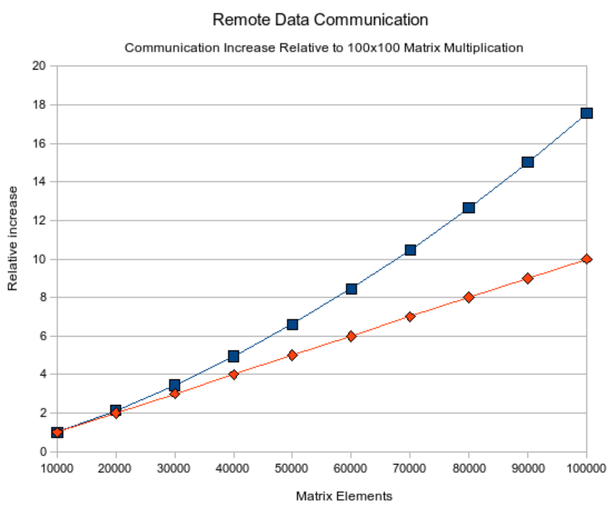

Figure 4: Remote communication increases faster than the problem size in our implementation. The red (bottom) line shows a linear increase and the blue (top) line shows the increase observed in our implementation.

much more data movement than we had intended. When multiplying two 100×100 matrices with a block size of ten our implementation induced 228,400 remote data communications. Without the use of user defined domain maps it is very difficult to reason about how many remote operations will be used to implement a given statement in Chapel which results in poor performance. Figure 4 shows that in our implementation of the blocked GEMM algorithm the amount of data communication grows faster than the problem size. This would most likely not have been the case if a user defined domain map had been used to manage our data locality. For this reason it is too early to decide if Chapel will eventually be able to efficiently handle the data movement involved in this blocked GEMM algorithm.

5 SUMMA Implementation

5.1 Description

Though not part of libFLAME we decided to implement a scalable matrix multiplication algorithm called SUMMA [9]. SUMMA has a number of benefits in that it not only achieves high performance but is also relatively simple compared to its predecessors. This algorithm is only practical in a parallel environment so it wouldn't make sense for the sequential FLAME library to include it. However, SUMMA tests multiple facets of Chapel's programmability so its implementation is a good exercise. The key to achieving performance benefits is replication. We give a high level explanation of the algorithm for the sake of clarity before discussing our implementation and issues encountered.

5.1.1 Unblocked SUMMA

Assume we have a group of computation nodes that can be logically arranged into a $r \times c$ mesh, each node can be indexed by P_{ij}, where i is the node's row and j is the node's column. To simplify the explanation assume we have two matrices A and B where A is $r \times n$ and B is $n \times c$. In practice this simplification can be removed by using a block-cyclic distribution along the rows and columns of nodes. We replicate row a_i along node row p_i and column b_j along node column p_j. Each node can then perform a rank-one update in parallel to compute $C_{ij} = a_i * b_j$. The idea is that the communication overhead for replicating each row

```
// replicated distributions aren't implemented yet, but imagine that
// they look something like the following:
// var replAbD: domain(2)
//                        dmapped new Dimensional(BlkCyc(blkSize),Replicated))
//                        = AbD[start.., 1..#blkSize];

// actual code to achieve the distributed replication
// Notice the replicated domains must be declared INSIDE the "on" block
for pnl in 1..n by blkcols
{
  loc1 = (loc1 + 1) % numLocales;
  loc1cnt = loc1cnt + 1;
  on Locales(loc1)
  {
    var start: int = pnl;
    var replAbD: domain(2) = AbD[1.., start..start+blkcols-1];
    var replAb : [replAD] real = Ab[1.., start..start+blkcols-1];


                                        ⋮
```

Figure 5: Example adapted from Chapel HPL HPC Challenge 2009 entry [2] demonstrating how to replicate data across locales.

and column should be small compared to the performance gain achieved by having to perform only a single rank-one update on each node.

5.1.2 Blocked SUMMA

Van de Geijn and Watts [9] found that even better performance can be achieved by formulating the computations as matrix-matrix multiplications instead of rank-one updates. The idea is that communication is reduced because more data is being sent to each node and smaller blocks can take advantage of data locality better than entire rows or columns. Our implementation of the blocked SUMMA algorithm takes in two arguments for block size call them s and t. It partitions matrix A into $m \times t$ groups of columns, or column panels, and partitions matrix B into $t \times n$ row panels. Each panel of matrix B is replicated across its respective row in the mesh of nodes. For example, B_0^j, where j is the node column index, is replicated across node row p_0. The $m \times t$ panels of matrix A are then partitioned further into $s \times t$ blocks that are block-mapped to their corresponding logical position in the mesh of nodes. The algorithm then performs a GEMM operation in parallel on each node to compute block $C_{ij} = A_{ij} * B_l^j$, where l is the index of a block of rows in B.

5.2 Issues

We encountered significant programmability issues during all aspects of the algorithm's implementation. Chapel's current limitations with domain mapping forced us to do replication by placing data onto a locale (node) explicitly using the "on" keyword, as shown in Figure 5. In addition to being difficult to read this approach meant that to compute C all computation had to be done at every iteration of the innermost loop otherwise the names of the replicated data fall out of scope and can no longer be accessed. This also increases the communication required to compute an answer. We would ideally like to leave C partitioned over the mesh of nodes so that the only communication cost is in the replication of data. In Chapel however, if we leave C partitioned we have no way to access C's component blocks. In order to return an answer we have to send all the computed data from each node back to a single node that holds the original reference to C. Figure 6 shows the increase in communication that this incurs. Again, it would be nice to see if Chapel's

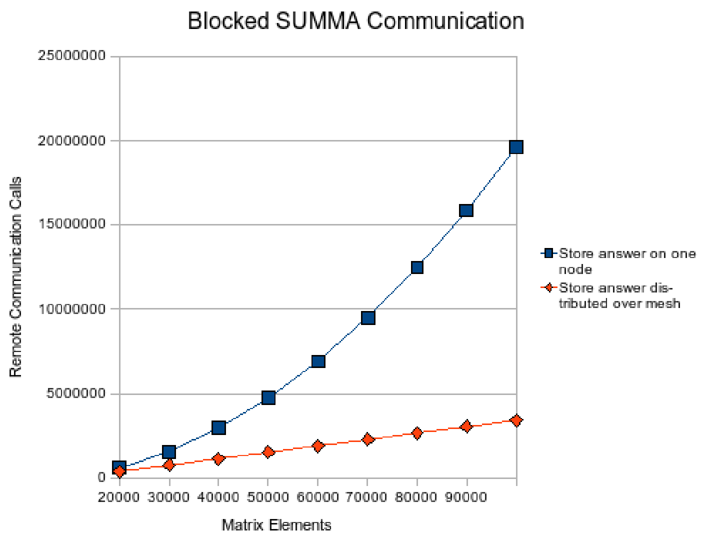

Figure 6: Remote communication required for our blocked SUMMA Chapel implementation measured using the same technique as in Figure 4. Squares show the large increase in communication required to store C on one node. Diamonds show communication for the ideal case where C is partitioned across the mesh of nodes.

user defined domain maps and replicated distributions could handle this algorithm elegantly, however, at this stage in Chapel's development this was not possible.

6 Conclusions

It is still too early in the development of Chapel to reach any final conclusions regarding Chapel's ability to implement parallel linear algebra libraries. The results from our shared memory GEMM implementation show that relative speedup is possible as the number of threads increases which is promising. If optimizations continue to be added to the Chapel compiler it may become possible to achieve performance that surpasses the sequential C implementation of libFLAME. From a programmability point of view the biggest deterrent for library developers at this point in Chapel's development is the lack of data locality features. The inability to create user defined domain maps across locales is a major issue when attempting to implement efficient linear algebra algorithms. Once this feature is implemented in the Chapel compiler another evaluation should be conducted. In the compiler's current state it is too difficult to manage data locality for some of the proven parallel GEMM algorithms.

References

[1] Bientinesi, P., Quintana-Ortí, E. S., and van de Geijn, R. A. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software 31*, 1 (Mar. 2005), 27–59.

[2] Chamberlain, B. L., Choi, S.-E., Choi, S.-E., and Iten, D. Hpc challenge benchmarks in chapel. *HPC Challenge Awards Competition at SC09* (November 2009).

[3] Cray Inc. *Chapel Language Specification 0.795*, April 2010.

[4] Goto, K. Texas advanced computing center: Tacc projects.

[5] Poulson, J. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. Supercomputing 2010, Apr. 2010.

[6] Poulson, J., and van de Geijn, R. A. Rethinking Distributed Memory Dense Linear Algebra. Apr. 2010.

[7] van de Geijn, R. A. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[8] van de Geijn, R. A., and Quintana-Ortí, E. S. *The Science of Programming Matrix Computations.* `www.lulu.com`, 2008.

[9] van de Geijn, R. A., and Watts, J. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience 9*, 4 (April 1997), 255–274.

[10] Zee, F. G. V. `libflame`*: The Complete Reference.* `www.lulu.com`, 2009.