Matrix Multiplication Specialization in STAPL

Adam Fidel, Lena Olson, Antal Buss, Timmie Smith, Gabriel Tanase,
Nathan Thomas, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{afidel,lolson,abuss,timmie,gabrielt,nthomas,bmm,amato,rwerger}@cs.tamu.edu

Abstract

The Standard Template Adaptive Parallel Library (STAPL) is a superset of C++'s Standard Template Library (STL) which allows high-productivity parallel programming in both distributed and shared memory environments. This framework provides parallel equivalents of STL containers and algorithms enabling ease of development for parallel systems. In this paper, we will discuss our methodology for implementing a fast and efficient matrix multiplication algorithm in STAPL. Our implementation employs external linear algebra libraries, specifically the Basic Linear Algebra Subprograms (BLAS) library which includes highly optimized sequential matrix operations. The paper will describe the benefits of creating a parallel matrix multiplication algorithm whose library calls are specialized based on both the matrix storage and traversal. This specialization technique ensures that the most appropriate implementation in terms of data access and structure will be used, resulting in increased efficiency compared to a non-specialized approach.

1 Introduction

Parallel libraries such as the Standard Template Adaptive Parallel Library (STAPL) [1, 5, 7, 6] allow developers to focus their programming efforts on higher-level abstract issues, rather than the intricacies of the parallelization itself. By providing a standard set of operations and procedures to the programmer, parallel code can be produced in a manner comparable to the development of sequential programs. With this in mind, we sought to incorporate a matrix multiplication algorithm which took advantage of parallelization in a way that is transparent to the end user.

In STAPL, one of the major components is the pAlgorithms, which are the direct equivalents of the Standard Template Library's sequential algorithms. STAPL also provides pContainers (parallel containers) and an abstraction of a container's data access called views. In terms of STL, the view defines an iteration space over a pContainer. We deal specifically with the pMatrix, where a column view can be taken over a container that is stored row-major. The input for a pAlgorithm in STAPL is a view and a work function which specifies the operations to be executed on the data. In the case of matrix multiplication, a view can be taken over an entire row, an entire column or a block of both rows and columns. This paper will focus mainly on specializing a matrix-matrix multiplication operation on row-major views and column-major views.

Interoperability is a key goal for STAPL's ability to work seamlessly across multiple platforms. For a framework to be interoperable, it must take advantage of routines and operations from external libraries and the framework's own methods should have the ability to be invoked from other frameworks. We focus on STAPL's ability to incorporate well-known and highly optimized libraries effectively into its framework.

2 Background

Matrix multiplication is a key part of many scientific applications where matrices can be dense and sufficiently large. For this reason, it is important that a parallel framework be able to handle distributed matrix multiplication in an efficient way. The manner in which the data is stored should also be flexible, in order to fit the needs of various applications.

For serial matrix multiplication, the Basic Linear Algebra Subroutines (BLAS) [4] libray can be used. BLAS is a collection of highly optimized FORTRAN subroutines for matrix and vector operations, including the level 3 subroutine general matrix multiply (gemm). By basing a parallel implementation on gemm, it is unnecessary to optimize the serial multiplication, instead relying on BLAS to provide an efficient implementation. However, because BLAS is a non-STAPL external library and requires the input data to be laid out in a specific format, it is not always possible to use the library. Additionally, BLAS only handles four data types: float, double, complex, and complex-double. When data has other types or is laid out in a non-conformable format, a different and often slower method of matrix multiplication must be used.

Partial specialization is a technique in generic programming in which a general function is provided along with several versions of the same function based on traits determined statically. The advantage of partial specialization is that when the data layout and type is appropriate, BLAS may be used, while still providing a general algorithm which can handle cases that do not conform to BLAS requirements. This allows the benefits of having fast methods which only work in a few cases while still having a fall-back operation which will always provide correct results. The choice between specializations and the general case is made statically at compile time, providing no additonal run-time cost.

3 Proposed Method

In STAPL, pContainers are distributed across multiple processors in many different partitioning strategies. The pMatrix container defines three types of partitions: block, block-cyclic and block-band. Block-band partitioning is a method of distribution in which each node contains blocks that consist of an entire row or column of the matrix. Our algorithm works most effectively for block-band distributions, but also provides functionality for block partitions.

Each matrix can be stored in block-band format in one of two directions,

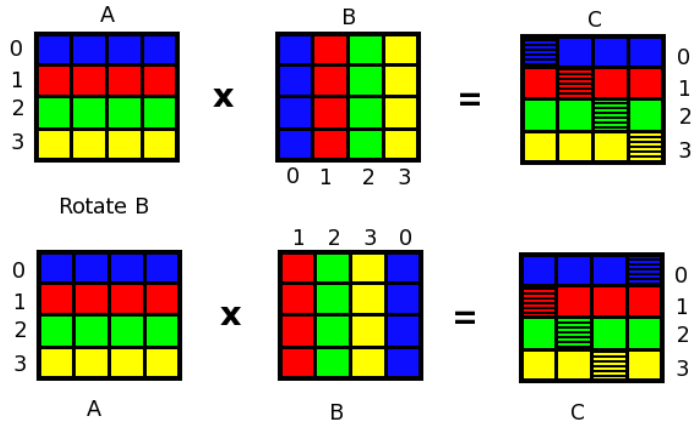


Figure 1: An illustration of the roll algorithm using RCR matrix multiplication

p_matrix_multiply(A, B, C)

```
1. FOR 1 to number of blocks
2.   multiply sub-blocks of A by sub-blocks of B using gemm
3.   store result in C
4.   rotate sub-blocks of B
5. END FOR
```

Figure 2: Proposed matrix multiplication `pAlgorithm`

hence there are 2^3 combinations of data partitioning possible for matrix multiplication. In this paper they will be referred to by a three-letter abbreviation, where R stands for row-major and C for column-major block-band partitioning, and ordering is ABC for multiplication A $\times$ B = C. For example, the case where A and C are row-major and B is column-major will be referred to as RCR. In the general algorithm, a row-view of A and C and a column-view of B is taken, irrespective of the actual data distribution. The multiplication is then performed using the Standard Template Library's `innerproduct`.

The general algorithm is very inefficient because it ignores the physical data partitioning, thus disallowing the use of `BLAS` due to data incontiguity. Therefore, more specific algorithms that exploit data locality can be used to improve performance. A different algorithm is needed for each case, because the data locality determines which of the three matrices should be rotated, as well as which elements to multiply and where to store the result. Eight algorithms are needed, yet six of these algorithms are very similar; only when the matrices to be multiplied are stored partitioned row-wise and the result is stored partitioned column-wise, or vice versa, is a substantially different algorithm needed. For

purposes of this paper, we will explain in detail one example: RCR.

Our proposed `pAlgorithm` employs the use of a temporary matrix in which to store the rotated data. As shown in Figure 2, the entirety of the algorithm is encased inside of a `for` loop. Each iteration consists of a block of A being multiplied with B utilizing `gemm`. Note that these two blocks are both local to the same processor and thus the proper conditions for `BLAS` multiplication are met. The result of this sequential multiplication is then stored in a sub-block of C which is also local. Then the matrix B is "rotated" meaning each block is sent to the processor directly to its right. At this point, the loop repeats itself until each block of the matrix A is multiplied with each block in B.

The other versions of the algorithm are similar to the one in Figure 1. In each case, one of the three matrices is rotated. In some cases partial results are stored in an entire row- or column-band in C, rather than the complete result being stored in one sub-block after each multiplication.

In the cases where both A and B are stored with a different major than C (i.e. RRC and CCR), the algorithm is substantially different. These cases are fundamentally more difficult than the other six, and the equivalent algorithm would require a rotation of two matrices and thus a squaring of the total number of rotations and multiplications. This approach is undesirable because as the number of blocks increases linearly, the number of rotates and multiplications increases quadratically. This results in an algorithm that does not scale. An alternate approach is to transpose one of the matrices and then call one of the six fast specializations. The efficiency of transposition, if we focus solely on moving data between processors and ignore rearranging it within each one and assume that each processor has only one block, depends on the communication costs. If there are p processors, the number of elements in an $n \times n$ matrix that must be transferred by each processor is

$$\frac{n^2 \cdot (p-1)}{p^2} \tag{1}$$

with each processor communicating $(\frac{n}{p})^2$ elements with each other processor. The cost of communication for the transpose, $C_{transpose}$, for a start-up time t_s and a transfer rate of t_w is thus

$$C_{transpose} = (p-1)(t_s + (\frac{n}{p})^2 \cdot t_w) \approx t_s \cdot p + \frac{n^2}{p} \cdot t_w \tag{2}$$

Therefore, when the start-up cost is relatively high, the communication cost will be large because although the amount of data communicated decreases with increasing processor count, the number of messages sent increases, and t_s will dominate. However, the communication cost for the transpose operation is always less than the cost of the sum of the rotate steps in the multiplication algorithms.

$$C_{rotate} = t_s \cdot p + n^2 \cdot t_w > C_{transpose} \tag{3}$$

However, this relies on a well-implemented transpose function which sends as many elements as possible at a time, and a reasonably large n. Otherwise,

the start-up costs will dominate the communication and the transpose will not achieve reasonable performance. We implement the transpose using `STAPL`'s built-in `p_copy`, which is the parallel version of `STL`'s `copy`, and we pass it a view consisting of $(\frac{n}{p})^2$ elements.

In addition to the eight algorithms discussed above, each algorithm also has a number of specializations. Currently, there are specializations based on the major in which local blocks are stored. These specializations simply determine the correct values and transpose flags to pass to the `gemm` call. In order for BLAS to be used, the following conditions must be met:

- Contiguous and local data in memory
- Laid out row-major or column-major
- Elements are of BLAS-recognized data type

In STAPL, the most appropriate version of our algorithm is determined during compilation with the help of Boost metafunctions. Through the use of templated generic programming, we provide six functions of the same operation, each function being specific to a certain data traversal and block-band layout.

```
template <typename vA, typename vB, typename vC>
struct matrix_multiply<vA, vB, vC,
  typename enable_if<and_<
              blas_conformable<vA, vB, vC>,
              row_view<vA>,
              column_view<vB>,
              row_view<vC>
            > >::type >
{
  void operator()(vA& va, vB& vb, vC& vc)
  { // determines BLAS descriptors }
};
```

Figure 3: Example of specialization based on view type

In the future it may be desirable to extend the algorithms to handle local blocks stored in other formats, such as smaller blocks. This could be accomplished by adding a new, more complicated specialization which would multiply smaller blocks together. At the moment, these other layouts are handled by a slower general implementation using `innerproduct`. In addition to the specializations based on data layout, there are also specializations by data type. `gemm` has four versions: `sgemm`, `dgemm`, `cgemm`, and `zgemm` for handling `float`, `double`, `complex`, and `complex-double` data types, respectively. If the data type is recognized by `BLAS`, then the appropriate version of `gemm` is called; otherwise the `innerproduct` general algorithm is used.

4 Experimental Results

Our experimental setup is 8192×8192 matrix multiplication of doubles on a cluster consisting of 40 p5-575 nodes, each with 16 Power5+ CPUs with 32 GB of DDR2 DRAM and 2-plane HPS interconnect.

The six non-transposing algorithms that we implemented showed similar results in cases where `BLAS` was called.

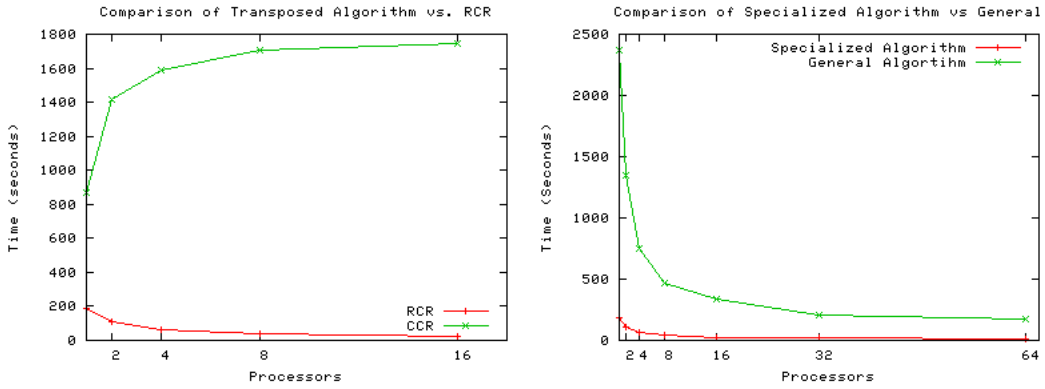
The difference between the general, `innerproduct` implementation and the `BLAS` specializations was dramatic in terms of execution time. For a 8192x8192 matrix, the specialized version for RCR executed approximately 12.62 times faster than the general implementation on one processor, and approximately 11.45 times faster on 64 processors. In our experiments, the general algorithm was executed with optimal data locality (i.e. RCR partitioning and multiplying with innermost loop having stride of one), in order to make full use of cache and minimize communication. The general implementation showed similar scalability compared with the `BLAS` specializations as reflected in Figure 5 (b).

As indicated in Figures 6 (a) and (b), the six implemented algorithms showed very similar scalability and almost identical execution times. This can be accounted for by the static nature of the algorithm decision and the similar philosophy used within the algorithms themselves.

The two algorithms involving transpose were significantly worse than the non-transposing versions as shown in Figure 5 (a). As these call the same algorithm as the rest after the transpose, it is the transpose that is negatively affecting the performance. These cases take increasing amounts of time as more processors are added and are, except for when only one processor is used, significantly worse than the general implementation. Therefore these cases are not worth specializing, at least until a more efficient transpose routine is implemented in the `STAPL` framework.

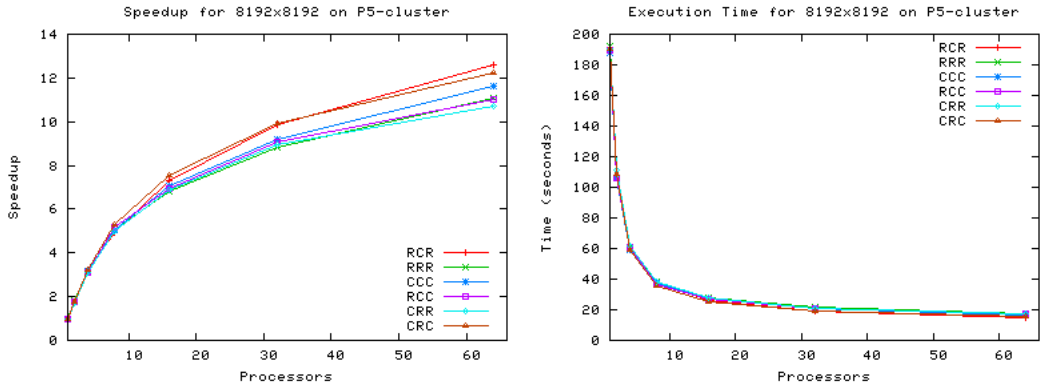
CPUs	RCR (sec)	General (sec)	Speedup
1	188.09	2373.34	12.62
2	108	1349.38	12.49
4	59.82	750.92	12.55
8	38.07	466.11	12.24
16	25.74	336.73	13.08
32	19.03	208.42	10.95
64	14.96	171.2	11.45

Figure 4: Comparison of RCR (`BLAS`) and the general implementation (`innerproduct`) in terms of execution time in seconds



(a) (b)

Figure 5: Comparison of (a) RCR vs. CCR which employs matrix transposition and (b) specialized algorithm vs. `innerproduct` implementation



(a) (b)

Figure 6: Execution time (a) and relative speedup (b) of several matrix multiplication specializations on P5-cluster (doubles)

5 Related Work

Many parallel matrix multiplication algorithms have been developed to achieve high performance with regard to specific matrix layouts. The creators of the Basic Linear Algebra Subprograms library created a parallel implementation of their general matrix multiplication algorithm included in `PBLAS` [2] which is limited to block-cyclic distributions. `PBLAS` differs from our matrix multiplication technique in that it provides exceptional performance at the cost of working in much fewer cases and with arguable difficulty with invocation and integration in frameworks. `STAPL` is unique in that it provides an adaptive framework to decide the best method of performing a task during both run-time and compile-time. For example, if a specific data distribution conforms to `PBLAS` requirements, `STAPL` will adaptively invoke `PBLAS`'s matrix multiplication. If this is not possible, it will attempt to utilize our algorithm to provide the best possible performance while simultaneously increasing the number of situations that `STAPL` can handle matrix multiplication.

To cover the currently implemented data distributions in `pMatrix`, there exist multiple matrix multiplication algorithms for blocks or 2- and 3-dimensional meshes. [3] mentions a number of algorithms including Cannon's algorithm and the DNS algorithm that work on this type of data distribution. These algorithms differ from ours in that they rely on a specific block partioning whereas our algorithm is dependent on a block-band distribution.

6 Summary

In this paper, we showed that matrix multiplication using `innerproduct` is far slower than what is possible using well-known, highly optimized, third-party libraries specifically created for matrix multiplication, such as `BLAS`, thus showing the benefit of interoperabilty and specialization. We demonstrated the increased performance that can be achieved by using external libraries for the sequential portion of several matrix multiplication algorithms and the benefits of interoperably in `STAPL`. We further showed that partial specialization can be used to determine with little to no run-time overhead the most appropriate routine based on factors determinable statically.

In the future, it would be interesting to examine the causes of poor performance from `STAPL`'s transpose method used in two of the algorithms. An efficient and scalable implementation of matrix transposition would allow the last two algorithms (CCR and RRC) to achieve performance similar to the other six.

References

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel

programming library for C++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug 2001.

[2] J. Choi, J. J. Dongarra, L. S. Ostrouchov, Petitet, A. P., D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, Fall 1996.

[3] P. S. H. Gupta. Communication efficient matrix-multiplication on hypercubes. In *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, volume 22 of *Extended version in Parallel Computing*, pages 75–99, 1994.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[5] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *Proceedings of the 2007 Workshop on Memory Performance (MEDEA)*, pages 73–80, Brasov, Romania, 2007.

[6] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Associative parallel containers in STAPL. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, 2007, to appear.

[7] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, Chicago, IL, USA, 2005. ACM.