Parallelizing FLAME Code with OpenMP Task Queues

Tze Meng Low
Kent F. Milfeld
Robert A. van de Geijn
Field G. Van Zee
The University of Texas at Austin
Austin, TX 78712

FLAME Working Note #15

Dec. 3, 2004

Abstract

We discuss the OpenMP parallelization of linear algebra algorithms that are coded using the Formal Linear Algebra Methods Environment (FLAME) API. This API expresses algorithms at a higher level of abstraction, avoids the use of indices, and thus represents these algorithms as they are formally derived and presented. Traditional OpenMP directives require an explicit loop index, or explicit critical-region constructs on a variable, in order to indicate parallelism in loops and thus the lack of indices previously posed a challenge. A feature, *task queues*, that has been proposed for adoption into OpenMP 3.0 overcomes this problem. We illustrate the issues and solutions by discussing the parallelization of the symmetric rank-k update and report impressive performance on a 4 CPU Itanium2 server.

1 Introduction

The Formal Linear Algebra Methods Environment (FLAME) project pursues a systematic methodology for deriving and implementing linear algebra libraries [2, 9]. The methodology is goal-oriented: Given a mathematical specification of the operation to be implemented, prescribed steps yields a family of algorithms for computing the opreation. As part of the derivation, the proof of correctness of the algorithm is also given. The resulting algorithms are expressed at a high level of abstraction, much like one would present algorithms with pseudo-code in a classroom setting. Application Programming Interfaces (APIs) have been developed allow the code to closely resemble the formal algorithm structure so that the opportunity for the introduction of "bugs" in the translation from algorithm to implementation is reduced. APIs have been defined for the Matlab M-script language, for the C and Fortran programming languages, and even as an extension to the Parallel Linear Algebra Package (PLAPACK) [3, 13]. The scope of FLAME includes the Basic Linear Algebra Subprograms (BLAS) [10, 6, 5], most of LAPACK [1], and a large number of operations encountered in Control Theory [11].

Integrating OpenMP directives into the resulting code is a problem in that the code is devoid of indexing: OpenMP constructs for parallelizing loops usually require a loop-index to indicate how the loop is to be parallelized. Task queues, a construct that was recently proposed for inclusion in OpenMP 3.0, allow tasks to be defined by a single control structure. These tasks are then scheduled for execution on the different threads. We show in this paper how this Workqueuing Model naturally supports parallelism in C code written with the FLAME/C API. We refer to the resulting extension of FLAME/C as *OpenFLAME*. The

Workqueuing Model can be applied to many algorithms that are systematically derived via the FLAME approach for operations supported by the BLAS and LAPACK.

We demonstrate the general applicability of the approach with a concrete example: the computation of the symmetric rank-k update (SYRK) operation. This operation is supported by the BLAS and is important in higher-level operations like the Cholesky factorization and the formation of the normal equations in linear least-squares problems. For that example, impressive performance is reported on an Intel Itanium2 (R) Symmetric Multiprocessor (SMP).

The paper is organized as follows: In Section 2 we discuss the SYRK operation, four algorithmic variants for computing it, and the implementation of those algorithms using FLAME/C. The parallelization of the resulting implementations using OpenMP and task queues is discussed in Section 3. An additional algorithmic variant is presented in Section 4. The parallelization of that fifth variant requires partial results, computed by different tasks in the task queue, to be summed. Performance attained by the different implementations is presented in Section 5. Concluding remarks are given in the final section.

2 A Concrete Example

Consider the computation $C := AA^T + C$ where C is symmetric and hence only the lower triangular part of C is stored and updated. This operation is known as a *symmetric rank-k update* (SYRK).

In the FLAME approach to deriving algorithms, matrices are partitioned into regions:

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) \quad \text{and} \quad A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$$

where the thick lines indicate how far into the matrices the computation has reached. It is assumed that C_{TL} is square so that both C_{TL} and C_{BR} are symmetric. Here the '$\star$' symbol indicates the symmetric part of C that is not stored.

We will let $\hat{C}$ denote the original contents of C so that upon completion C should contain $C = AA^T + \hat{C}$, which is called the *postcondition*. It describes the state of the variables upon completion of the computation. Substituting the partitioned matrices into the postcondition yields

$$\begin{array}{rcl} \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) & = & \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)^T + \left(\begin{array}{c|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right) \\ & = & \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right). \end{array} \tag{1}$$

This shows that $m(C_{TL})$ should equal $m(A_T)$ and that $\hat{C}$ should be partitioned as is C, where $m(X)$ denotes the row dimension of matrix X.

The idea now is that (1) tells us *all* computations that must be performed in terms of the different submatrices of $\hat{C}$ and A. What we want to determine is the state of matrix C at the top of a loop that computes the result $C = AA^T + \hat{C}$. This state is referred to as the *loop-invariant*. If the loop computes the result, not all computation that is required has already been performed. This suggests the states given in Fig. 1 as states that can be maintained as loop-invariants at the top of a loop: they are partial results towards the final result.

What is important here is that for each loop-invariant there is a corresponding algorithmic variant: Loop-invariant k in Fig. 1 yields the algorithmic Variant k in Fig. 2, in which so-called *blocked* algorithms are given that in the loop-body update various submatrices of matrix C. An *unblocked* algorithm can be created by taking $m(C_{11}) = m(A_1) = 1$, in which case the updates in the body of the loop become simpler operations

	Loop-invariant
1	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$
2	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$
3	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$
4	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$

Figure 1: Loop-invariants for computing SYRK.

like the matrix-vector product and inner-product. In each of the loop-bodies there is the computation of a SYRK operation with smaller submatrices of A and C.

Having the ability to derive correct algorithms solves only part of the problem since translating those algorithms to code ordinarily required delicate indexing into arrays, which exposes opportunities for the introduction of errors. We now illustrate how appropriately defined APIs overcome this problem. In Fig. 3, we show an example of FLAME/C code corresponding to Variant 1 in Fig. 2. To understand the code, it suffices to know that `C` and `A` are descriptors for the matrices C and A, respectively. The various routines facilitate the creation of *views* into the data described by `C` and `A`. Think of a variable like `CTL` as a fancy pointer into the array C. Furthermore, the calls to `FLA_Gemm` and `FLA_Syrk` perform the same operations as the BLAS calls DGEMM (matrix-matrix multiplication) and `dsyrk` (symmetric rank-k update). *What is most striking about this code is the absence of intricate indexing and absense of a loop control with a single variable.*

3 OpenFLAME := FLAME/C + (OpenMP + Task Queues)

The strength of FLAME code is that it hides intricate indexing. For OpenMP Standard 2.0, however, this strength is a weakness: inherently current OpenMP directives require loop indices in order to express parallelism in the execution of loops and/or explicit critical-region blocks for atomically updating a loop variable. Fortunately, a feature, task queues, is proposed for OpenMP Standard 3.0. It is this feature that allows a large number of algorithms to be easily parallelized when implemented with the FLAME API.

3.1 Task queues

Conceptually, the Workqueuing Model forms a queue for distributing tasks. Two workqueuing pragmas, `taskq` and `task`, formx a queue and units of work (tasks) for parallel execution, respectively. A single thread executes the taskq block, enqueuing tasks within the `task` block. Other threads dequeue tasks and execute them in parallel.

3.2 Application to SYRK

In Fig. 4 we show how the while loop in Fig. 3 can be annotated with OpenMP directives to create parallel tasks via the task queue mechanism. In Fig. 4:

Algorithm: $C := \text{SYRK\_BLK\_VAR1\_2}(\text{A}, \text{C})$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{TL} is 0×0, A_T has 0 rows

while $m(C_{TL}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 1:	**Variant 2:**
$C_{10} := A_1 A_0^T + C_{10}$	$C_{21} := A_2 A_1^T + C_{21}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Algorithm: $C := \text{SYRK\_BLK\_VAR3\_4}(\text{A}, \text{C})$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{BR} is 0×0, A_B has 0 rows

while $m(C_{BR}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 3:	**Variant 4:**
$C_{21} := A_2 A_1^T + C_{21}$	$C_{10} := A_1 A_0^T + C_{10}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Figure 2: Blocked algorithms for computing $C := AA^T + C$. The top algorithm implements Variants 1 and 2, corresponding to Loop-invariants 1 and 2 in Fig. 1. The bottom algorithm implements Variants 3 and 4, corresponding to Loop-invariants 3 and 4 in Fig. 1. The top algorithm sweeps through C from the top-left to the bottom-right, while **the bottom algorithm traverses the matrix in the opposite direction.**

```
#include "FLAME.h"

int Syrk_blk_var1( FLA_Obj C, FLA_Obj A, int nb_alg )
{
  FLA_Obj CTL,   CTR,      C00, C01, C02,
          CBL,   CBR,      C10, C11, C12,
                           C20, C21, C22;
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  int b;

  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );

  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
                         /*************/  /*********************/
                                            &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
                           b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,               &A0,
                        /* ** */           /* ** */
                                            &A1,
                           AB,              &A2,   b, FLA_BOTTOM );
    /*-----------------------------------------------------------*/

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A1, A0, ONE, C10 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );

    /*-----------------------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                            C10, C11, /**/ C12,
                              /*************/ /******************/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */            /* ** */
                              &AB,                A2,     FLA_TOP );
  }
}
```

Figure 3: FLAME/C code for a blocked implementation of Variant 1.

```
17    #pragma intel omp parallel taskq
18    {
19      while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
20        b = min( FLA_Obj_length( CBR ), nb_alg );
21
22        FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
23                               /*************/  /*********************/
24                                                 &C10, /**/ &C11, &C12,
25                                CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
26                                b, b, FLA_BR );
27        FLA_Repart_2x1_to_3x1( AT,                &A0,
28                              /* ** */           /* ** */
29                                                  &A1,
30                                AB,               &A2,   b, FLA_BOTTOM );
31        /*-----------------------------------------------------------*/
32        #pragma intel omp task captureprivate( A0, A1, C10, C11 )
33        {
34          FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
35          FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
36        } /* end task */
37        /*-----------------------------------------------------------*/
38        FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
39                                                    C10, C11, /**/ C12,
40                                  /*************/ /******************/
41                                  &CBL, /**/ &CBR,  C20, C21, /**/ C22,
42                                  FLA_TL );
43        FLA_Cont_with_3x1_to_2x1( &AT,                 A0,
44                                                       A1,
45                                  /* ** */            /* ** */
46                                  &AB,                 A2,     FLA_TOP );
47      }
48    }   /* end of taskq */
```

Figure 4: FLAME/C code with task queuing OpenMP directives for the code in Fig. 3.

- **Line 17** creates the taskq block and forms a single-threaded taskqueue.
- **Line 32** starts a section of code that defines a task to be added to the task queue. The descriptors `A0`, `A1`, `C10`, and `C11` change from iteration to iteration. They need to be private (local) variables and to have value assigned (captured) from the taskq thread for use in the calls to `FLA_Gemm` and `FLA_Syrk`.
- **Line 36** ends the scope of the task being added to the queue.
- **Line 48** ends the scope of the taskq block. The threads are synchronized at that line.

Clearly, task queues provide a simple mechanism for directing the parallel execution in this code. Moreover, without the task queue mechanism indices would have had to be reintroduced into the code, making it substantially more complex and aesthetically less pleasing.

Especially for blocked algorithms, the cost of the indexing operations (`FLA_Repart_`... and `FLA_Cont_with_`...) is amortized over enough computation that the associated overhead is negligible. Thus it suffices to parallelize the useful computation in the loop and not these indexing operations.

3.3 Options

In Fig. 4 the calls to `FLA_Gemm` and `FLA_Syrk` are independent and can, therefore, be executed in any order and/or queued as separate tasks. One option is to split the single task in the loop-body of Fig. 4 into two

tasks:

```
#pragma intel omp task captureprivate(A0, A1, C10)
{
  FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
            ONE, A1, A0, ONE, C10 );
}
#pragma intel omp task captureprivate(A1, C11)
{
  FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
            ONE, A1, ONE, C11 );
}
```

This creates twice the number of tasks for the task queue to schedule.

A further observation is that the computations $C_{10} := A_1 A_0^T + C_{10}$ and $C_{11} := A_1 A_1^T + C_{11}$ (updating the lower triangle only) cost about $2bn(C_{10})n(A)$ and $b^2 n(A)$ floating point arithmetic operations (flops), respectively. Here $n(X)$ indicates the column dimension of matrix X. Since $n(C_{10})$ grows linearly with each iteration of the loop the number of flops required to update C_{10} increases propotionally. Thus is unfortunate, since costly tasks at the end of a scheduling queue can create a large load imbalance.

One option to overcome this problem is to execute the loop in reverse order (in compiler terms: apply a loop reversal transformation), since this would then create the more costly tasks first. Variants 4 and 3 in Fig. 2 execute the loops in Variants 1 and 2 in reverse, respectively. This illustrates the value of the FLAME methodology which can systematically find algorithmic variants that have different strengths and weaknesses. In fact, Variants 1 and 3 have the property that tasks become more costly as the loop proceeds while Variants 2 and 4 generate progressively less costly tasks. What we will later see is that differences in performance can be observed for different variants.

An alternative option is to create two loops (in compiler terms: apply a loop fission transformation), replacing the single loop in Fig. 4 with two loops: the first for computing all the updates to C_{10} and the second loop for computing the updates to C_{11}:

```
#pragma intel omp parallel taskq
{
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3(

               [ ... ]

    #pragma intel omp task captureprivate(A0, A1, C10)
    {
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                ONE, A1, A0, ONE, C10 );
    }
               [ ... ]
  } /* end of first while loop */
```

```
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3(

                [ ... ]

    #pragma intel omp task captureprivate(A1, C11)
    {
      FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                ONE, A1, ONE, C11 );
    }
                [ ... ]

  } /* end of second while loop */
} /* end of taskq */
```

The updates to C_{11} require less work and are all equal in cost, allowing them to be used to balance the workload among threads before the synchronization upon completion of the tasks.

3.4 An illustration of the benefits of different options

The expected differences in performance are illustrated for Variants 2 and 3 in Fig. 5. In that figure, we report a simulation of the scheduling of tasks to four threads for the different options described above. The matrices A and C are taken to be of dimension 1200×1200 and the block size b in Fig. 2 is taken to equal 104 (except possibly during the last iteration), which is a block size that we will use in our experimental section as well. Each of the tasks is represented by a box that has a height that is proportional to the number of flops performed by the task. The integers in the boxes indicate the order in which the tasks are queued in the task queue. The tasks are scheduled to threads as they become idle. Recall that these variants perform the same computations, but the loop is executed in reverse for Variant 3.

We see that Variant 2 in general performs better than Variant 3 since the costs of the tasks decrease towards the end, allowing them to be more easily balanced among the threads before synchronization. Splitting the task in the loop-body into two tasks improves the load-balance for Variant 2, but not for Variant 3. Both variants benefit from splitting the loop into two loops, with the smaller tasks scheduled by the second loop. These small tasks, generated by the second loop, will be executed by those threads that complete their share of the tasks generated by the first loop early.

4 Summing Contributions from Tasks

From experience with parallelizing algorithms on distributed memory architectures [13, 8, 12], we (and others) have concluded that there are two types of communications needed to support the parallelization of operations like those in the BLAS and LAPACK: the first is data duplication where data are communicated to different processors and followed by the execution of completely independent tasks on each processor. The second involves the reduction of locally computed contributions to a global result. Typically the reduction is in fact a summation of contributions (partial sums) from each processor.

The method for using OpenMP described so far supports the SMP equivalent of the first type of communication: It defines separate tasks that update parts of matrices that do not overlap, using data that are shared and may be accessed concurrently. In lieu of duplication, each separate task reads the same data, as needed, from shared storage. In order to support independent tasks contributing to an update of the same data via the task queue construct, it has to be possible to compute contributions independently using data that are not shared, and to then reduce the results into a shared matrix or vector. We illustrate now how to accommodate this via task queues by discussing a fifth variant for computing SYRK.