

# 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 operation. 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

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

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  $\mathbf{C}$  and  $\mathbf{A}$  are descriptors for the matrices  $C$  and  $A$ , respectively. The various routines facilitate the creation of *views* into the data described by  $\mathbf{C}$  and  $\mathbf{A}$ . 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 absence 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`, form a queue and units of work (tasks) for parallel execution, respectively. A single thread executes the `taskq` block, enqueueing 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:

<b>Algorithm:</b> $C := \text{SYRK\_BLK\_VAR1\_2}(A, C)$	
<b>Partition</b> $C \rightarrow \left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right), A \rightarrow \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right)$	
where $C_{TL}$ is $0 \times 0$ , $A_T$ has 0 rows	
<b>while</b> $m(C_{TL}) < m(C)$ <b>do</b>	
<b>Determine block size</b> $b$	
<b>Repartition</b>	
$\left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c c c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right), \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$	
where $C_{11}$ is $b \times b$ , $A_1$ has $b$ rows	
<hr style="border: 0.5px solid black;"/>	
<b>Variant 1:</b> $C_{10} := A_1 A_0^T + C_{10}$ $C_{11} := A_1 A_1^T + C_{11}$	<b>Variant 2:</b> $C_{21} := A_2 A_1^T + C_{21}$ $C_{11} := A_1 A_1^T + C_{11}$
<hr style="border: 0.5px solid black;"/>	
<b>Continue with</b>	
$\left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c c c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right), \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$	
<b>endwhile</b>	

<b>Algorithm:</b> $C := \text{SYRK\_BLK\_VAR3\_4}(A, C)$	
<b>Partition</b> $C \rightarrow \left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right), A \rightarrow \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right)$	
where $C_{BR}$ is $0 \times 0$ , $A_B$ has 0 rows	
<b>while</b> $m(C_{BR}) < m(C)$ <b>do</b>	
<b>Determine block size</b> $b$	
<b>Repartition</b>	
$\left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c c c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right), \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$	
where $C_{11}$ is $b \times b$ , $A_1$ has $b$ rows	
<hr style="border: 0.5px solid black;"/>	
<b>Variant 3:</b> $C_{21} := A_2 A_1^T + C_{21}$ $C_{11} := A_1 A_1^T + C_{11}$	<b>Variant 4:</b> $C_{10} := A_1 A_0^T + C_{10}$ $C_{11} := A_1 A_1^T + C_{11}$
<hr style="border: 0.5px solid black;"/>	
<b>Continue with</b>	
$\left( \begin{array}{c c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c c c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right), \left( \begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left( \begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$	
<b>endwhile</b>	

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.

```

1  #include "FLAME.h"
2
3  int Syrk_blk_var1( FLA_Obj C, FLA_Obj A, int nb_alg )
4  {
5      FLA_Obj CTL,   CTR,   C00, C01, C02,
6              CBL,   CBR,   C10, C11, C12,
7              C20, C21, C22;
8      FLA_Obj AT,
9              AB,
10             A0,
11             A1,
12             A2;
13
14     int b;
15
16     FLA_Part_2x2( C,   &CTL, &CTR,
17                 &CBL, &CBR,   0, 0, FLA_TL );
18     FLA_Part_2x1( A,   &AT,
19                 &AB,      0, FLA_TOP );
20
21     while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
22         b = min( FLA_Obj_length( CBR ), nb_alg );
23
24         FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
25                               /*****/ /*****/
26                               &C10, /**/ &C11, &C12,
27                               CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
28                               b, b, FLA_BR );
29         FLA_Repart_2x1_to_3x1( AT,
30                               /* ** */          &A0,
31                               /* ** */          &A1,
32                               AB,              &A2,   b, FLA_BOTTOM );
33         /*-----*/
34         FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A1, A0, ONE, C10 );
35         FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
36         /*-----*/
37         FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,   C00, C01, /**/ C02,
38                                   C10, C11, /**/ C12,
39                                   /*****/ /*****/
40                                   &CBL, /**/ &CBR,   C20, C21, /**/ C22,
41                                   FLA_TL );
42         FLA_Cont_with_3x1_to_2x1( &AT,
43                                   A0,
44                                   A1,
45                                   /* ** */          /* ** */
46                                   &AB,              A2,      FLA_TOP );
47     }
48 }

```

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  $b^2n(A)$  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 proportionally. 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  $1200 \times 1200$  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.

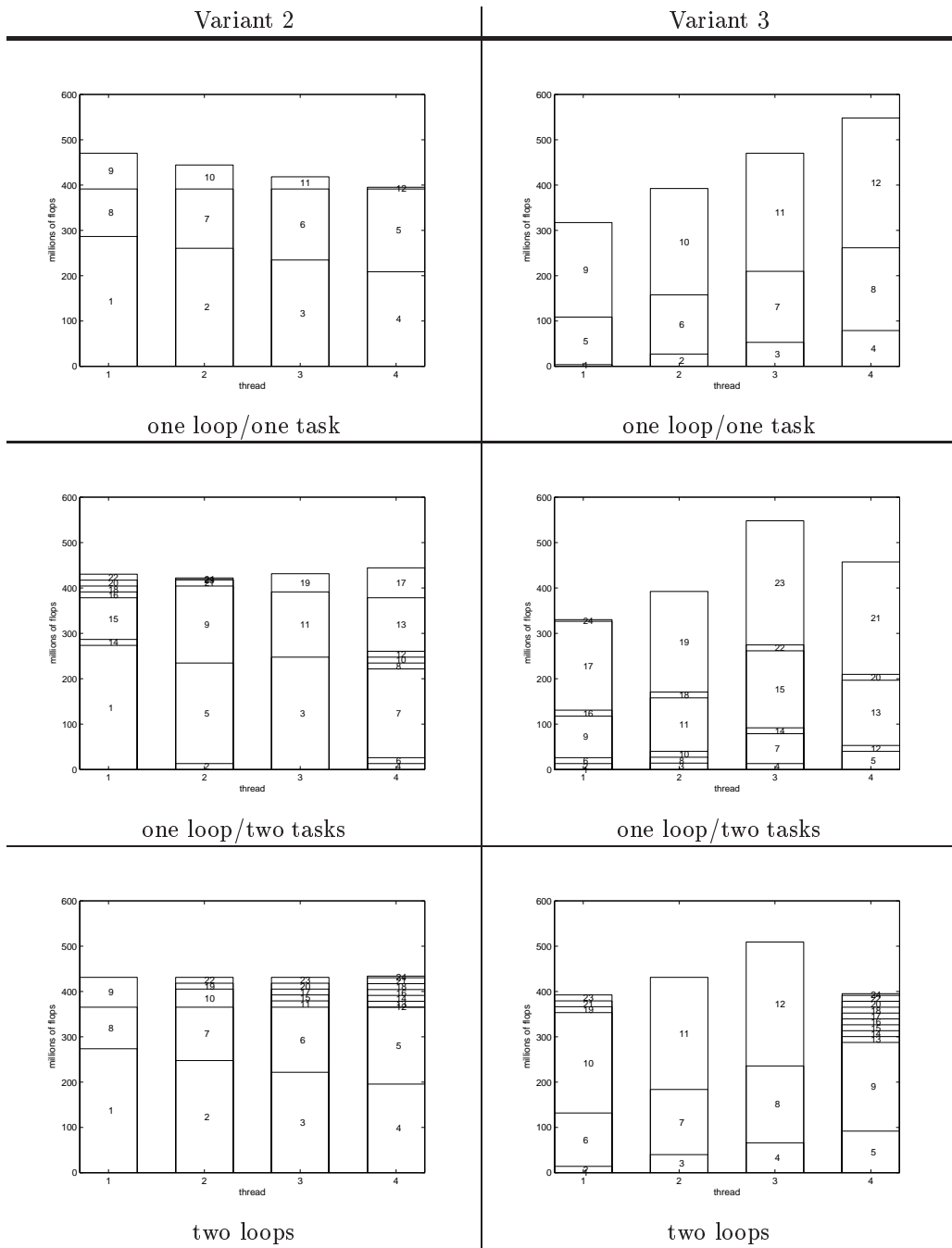


Figure 5: Scheduling of tasks for Variants 2 and 3 ( $C$  and  $A$  both  $1200 \times 1200$ ).

<b>Algorithm:</b> $C = \text{SYRK\_BLK\_VAR5}(A, C)$
<b>Partition</b> $A \rightarrow (A_L \mid A_R)$ <b>where</b> $A_L$ has 0 columns
<b>while</b> $n(A_L) < n(A)$ <b>do</b> <b>Determine block size</b> $b$ <b>Repartition</b> $(A_L \mid A_R) \rightarrow (A_0 \mid A_1 \mid A_2)$ <b>where</b> $A_1$ has $b$ columns
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/> <hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>
$C := C + A_1 A_1^T$
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/> <hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>
<b>Continue with</b> $(A_L \mid A_R) \leftarrow (A_0 \mid A_1 \mid A_2)$
<b>endwhile</b>

Figure 6: Blocked algorithm for  $C := AA^T + C$  (Variant 5).

Because all processes share access to the array that stores  $A$  while computing the SYRK operation,  $C := AA^T + C$ , it is also reasonable to partition  $A$  by blocks of columns:  $A = (A_0 \mid A_1 \mid \dots \mid A_{N-1})$ . Then

$$C := A_0 A_0^T + A_1 A_1^T + \dots + A_{N-1} A_{N-1}^T + C. \quad (2)$$

An algorithm for computing SYRK using this insight is given in Fig. 6 and the FLAME/C implementation is given in Fig. 7.

In Fig. 8 we show how task queues can be used to parallelize this algorithm. Clearly, it is best to choose  $N$  in (2) to be the number of available processors, so that the computational work of each  $A_i A_i^T$  is equal. Private updates to  $C$  are accumulated by each task in MyC, and then added to the global  $C$  matrix, in shared array  $C$ . We employ the routine `FLA_Axpy_local_to_shared` in Fig. 9 to atomically add the elements of MyC to  $C$ .

There are several other ways to synchronize the updates in the `FLA_Axpy_local_to_shared` utility routine. For instance, explicit indexing with critical regions can be employed for the atomic updates. Since it is a utility routine, our usual objection to indices does not apply, as is the case for our lower level computational kernels, like DGEMM, that also use explicit indexing.

## 5 Experiments

While the Workqueuing Model simplifies the parallelization of task-oriented programs, load balancing the tasks can still be a challenge. The FLAME framework and FLAME/C API is ideal for testing and evaluating the impact that algorithmic variants have on load balancing. When there are multiple independent routine calls in a loop and their work varies in a well-defined way (linearly for `FLA_Gemm`, constant for `FLA_Syrk`) the routine order within the loop and the order of separate loops become another dimension in the variant space that affects load balancing. Knowing the characteristics of the work associated with an computation is often enough to accurately predict the optimal looping arrangement. However, memory contention and false sharing can complicate any prediction. The purpose of the following experimental investigations is to illustrate how the variants *and* options affect the algorithm performance for task queuing. Even though the results readily reveal optimal variants, the experiments are not meant to be an exhaustive search to achieve

```

1  int Syrk_blk_var5( FLA_Obj C, FLA_Obj A, int nb_alg )
2  {
3      FLA_Obj AL,    AR,    A0,  A1,  A2;
4      int b;
5
6      FLA_Part_1x2( A,    &AL,  &AR,    0, FLA_LEFT );
7
8      while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){
9          b = min( FLA_Obj_width( AR ), nb_alg );
10
11         FLA_Repart_1x2_to_1x3( AL,  /**/ AR,    &A0, /**/ &A1, &A2,
12                                b, FLA_RIGHT );
13         /*-----*/
14
15         FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C );
16
17         /*-----*/
18         FLA_Cont_with_1x3_to_1x2( &AL,  /**/ &AR,    A0, A1, /**/ A2,
19                                   FLA_LEFT );
20     }
21 }

```

Figure 7: FLAME code for a blocked implementation of Variant 5.

```

7  #pragma intel omp parallel taskq
8  {
9      while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){
10         b = min( FLA_Obj_width( AR ), nb_alg );
11
12         FLA_Repart_1x2_to_1x3( AL,  /**/ AR,    &A0, /**/ &A1, &A2,
13                                b, FLA_RIGHT );
14         /*-----*/
15         #pragma intel omp task captureprivate(A1) private(MyC)
16         {
17             /* Create a local copy of C for this task */
18             FLA_Obj_create_conf_to( FLA_NO_TRANSPOSE, C, &MyC );
19
20             /* MyC := 0 */
21             FLA_Obj_set_to_zero( MyC );
22
23             /* MyC := A1 * A1' */
24             FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, MyC );
25
26             /* C := C + MyC */
27             FLA_Axpy_local_to_shared( ONE, MyC, C );
28
29             FLA_Obj_free( &MyC );
30         } /* end of task */
31         /*-----*/
32         FLA_Cont_with_1x3_to_1x2( &AL,  /**/ &AR,    A0, A1, /**/ A2,
33                                   FLA_LEFT );
34     }
35 } /* end of task queue */

```

Figure 8: OpenFLAME code for the loop in Fig. 7.

```

1  #include "FLAME.h"
2
3  int FLA_Axpy_local_to_shared( FLA_Obj alpha, FLA_Obj X, FLA_Obj Y )
4  {
5      FLA_Obj XL,   XR,   X0, X1, X2;
6      FLA_Obj YL,   YR,   Y0, Y1, Y2;
7
8      int b, i, nb_alg, n_stages;
9
10     /* Set the number of stages equal to the number of processes */
11     n_stages = omp_get_num_procs();
12
13     FLA_Part_1x2( X,   &XL, &XR,   0, FLA_LEFT );
14     FLA_Part_1x2( Y,   &YL, &YR,   0, FLA_LEFT );
15
16     /* Compute the width of one lockable partition */
17     if( n_stages == -1 ) nb_alg = FLA_Obj_width(X);
18     else                  nb_alg = max( FLA_Obj_width(X)/n_stages + 1, 1 );
19
20     while ( FLA_Obj_width( XL ) < FLA_Obj_width( X ) ){
21         b = min( FLA_Obj_width( XR ), nb_alg );
22
23         FLA_Repart_1x2_to_1x3( XL, /**/ XR,      &X0, /**/ &X1, &X2,
24                               b, FLA_RIGHT );
25
26         FLA_Repart_1x2_to_1x3( YL, /**/ YR,      &Y0, /**/ &Y1, &Y2,
27                               b, FLA_RIGHT );
28         /*-----*/
29
30         /* Get the index of the current partition */
31         i = FLA_Obj_width(XL)/nb_alg;
32
33         /* Acquire lock[i] (the lock for X1 and Y1) */
34         omp_set_lock( &lock[i] );
35
36         /* Y1 := alpha * X1 + Y1 */
37         FLA_Axpy( alpha, X1, Y1 );
38
39         /* Release lock[i] (the lock for X1 and Y1) */
40         omp_unset_lock( &lock[i] );
41
42         /*-----*/
43         FLA_Cont_with_1x3_to_1x2( &XL, /**/ &XR,      X0, X1, /**/ X2,
44                                   FLA_LEFT );
45
46         FLA_Cont_with_1x3_to_1x2( &YL, /**/ &YR,      Y0, Y1, /**/ Y2,
47                                   FLA_LEFT );
48     }
49
50     return FLA_SUCCESS;
51 }

```

Figure 9: OpenFLAME code for adding a private matrix with descriptor X to a shared matrix with descriptor Y. This particular version uses locks to create a pipeline.

optimal performance for the SYRK operation.

## 5.1 Details

To demonstrate the effect of algorithmic variants and/or parallelization options on performance we implemented all five variants. For each of the first four variants we implemented three different options: a simple insertion of the task queue mechanism with one task in the loop-body (one loop/one task); the separation of the two updates to create two tasks in the loop-body (one loop/two tasks); and the separation of the two tasks into two separate loops (two loops).

All computation are in double precision (64 bit) arithmetic. The performance of the different implementations was measured on a 4 CPU Itanium2 (1.5GHz) workstation, with a peak performance of 6 GFLOPS ( $10^9$  flops/sec.) per processor, for a total peak of 24 GFLOPS. The Intel C compiler was used, since it supports the proposed OpenMP task queue construct even though, at this time, that construct is not yet part of the standard.

The implementations were linked to the BLAS libraries by Kazushige Goto [7]. The calls to `FLA_Gemm` and `FLA_Syrk` in the loop-body were implemented as wrappers to sequential implementations of the BLAS calls `DGEMM` and `DSYRK`. For Goto’s BLAS a block size of 104 is known to yield good performance from the matrix-matrix product routine `DGEMM`<sup>1</sup> and was therefore used in our experiments.

For computing the rate of computation, the operation count is  $m^2k$  flops for  $C \in \mathbb{R}^{m \times m}$  and  $A \in \mathbb{R}^{m \times k}$ . The GFLOPS rate reported in the graphs was computed by the formula

$$\text{GFLOPS attained} = \frac{m^2k}{\text{time (in sec.)}} \times 10^{-9}.$$

## 5.2 Results

The resulting performance is reported in Figs. 10–12. So that the performances can be visually evaluated relative to the theoretical peak performance of the machine, the range of the y-axis of the graphs is 0–24 GFLOPS. We organized the results so that Variants 2 and 3 are displayed on one page, as are Variants 1 and 4. These pairs of variants perform the same updates in the loop-body, but march through the matrices in opposite directions. Variants 2 and 4 create smaller tasks later in the computation, which explains the better and smoother performance attained by these implementations. Variants 2 and 3 are rich in matrix-matrix products that have the form  $A_2A_1^T$  while Variants 1 and 4 are rich in  $A_1A_0^T$ . The difference here is that  $A_2A_1^T$  yields a matrix with a large row dimension and a small column dimension, while  $A_1A_0^T$  yields a matrix with a small row dimension and a large column dimension. As of this writing the `DGEMM` matrix-matrix product in the BLAS by Goto is less optimized for this second case than for the first, which explains the lower performance attained by Variants 1 and 4. Given these insights, Variant 2 should perform best overall, as is observed in these experiments.

In Fig. 12 we compare Variant 5 only to the “two loops” version of Variant 2. In this figure, we explore the effects of picking different shapes of matrices as input to SYRK. The two top graphs report performance for square matrices  $C$  and  $A$ . In that case, Variant 2 performs best, likely due to the added synchronization overhead incurred by Variant 5 since contention occurs as contributions are added to  $C$  by different tasks. The center two graphs show drastically different results for the special case where  $C$  is  $104 \times 104$  and dimension  $k$  is varied. Now no parallelism is achieved by Variant 2, since it is executed with a block size of 104: only one iteration of the loop is executed, and therefore only one task is created. For this case, Variant 5 achieves reasonable performance as  $k$  increases, due to the fact that the contention during the addition

---

<sup>1</sup>Optimal performance of `DGEMM` is reported to require a block size of 1024. However, that is too large to achieve effective parallelization.

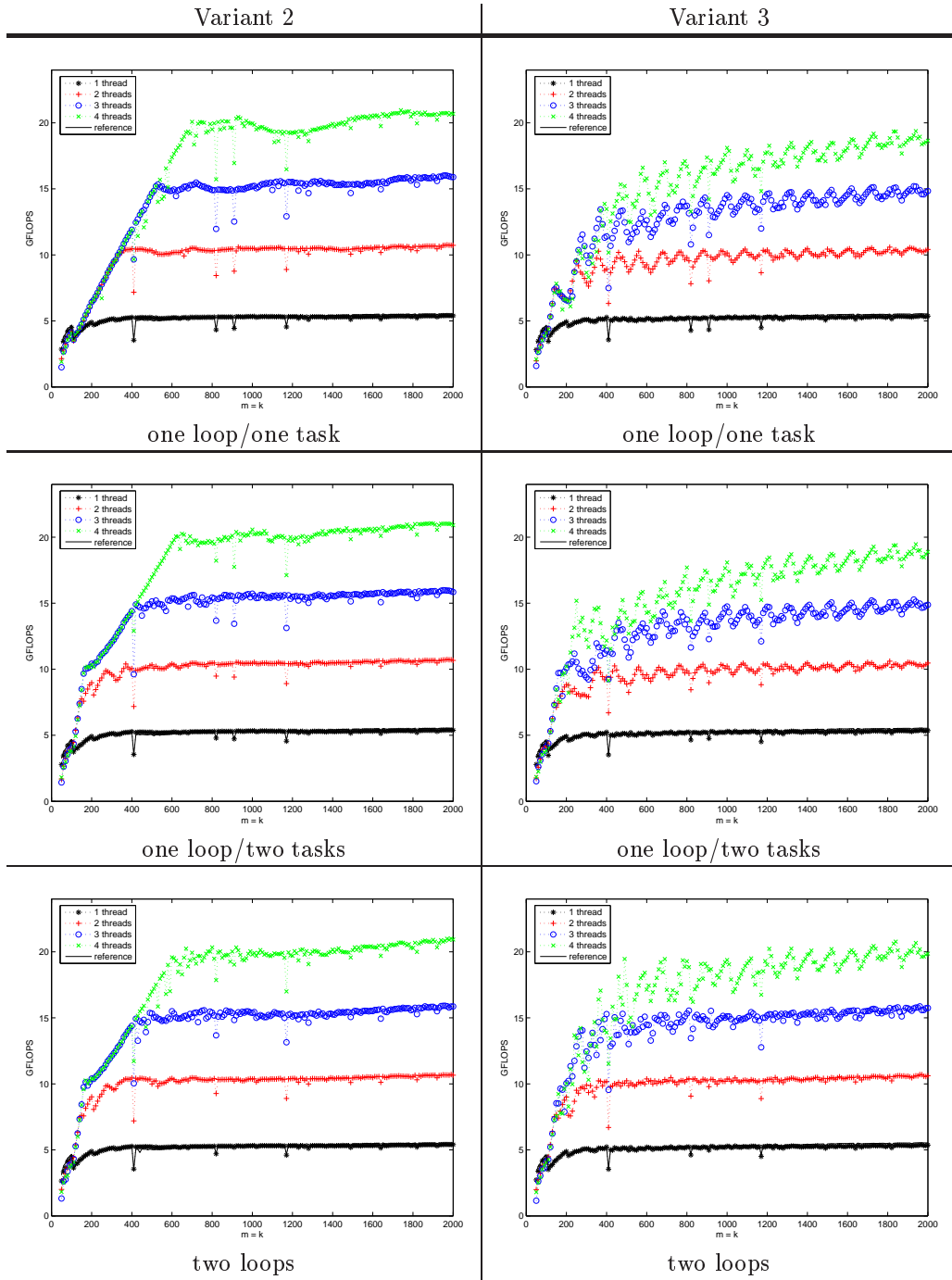


Figure 10: Performance of Variants 2 and 3 when  $C$  and  $A$  are square.

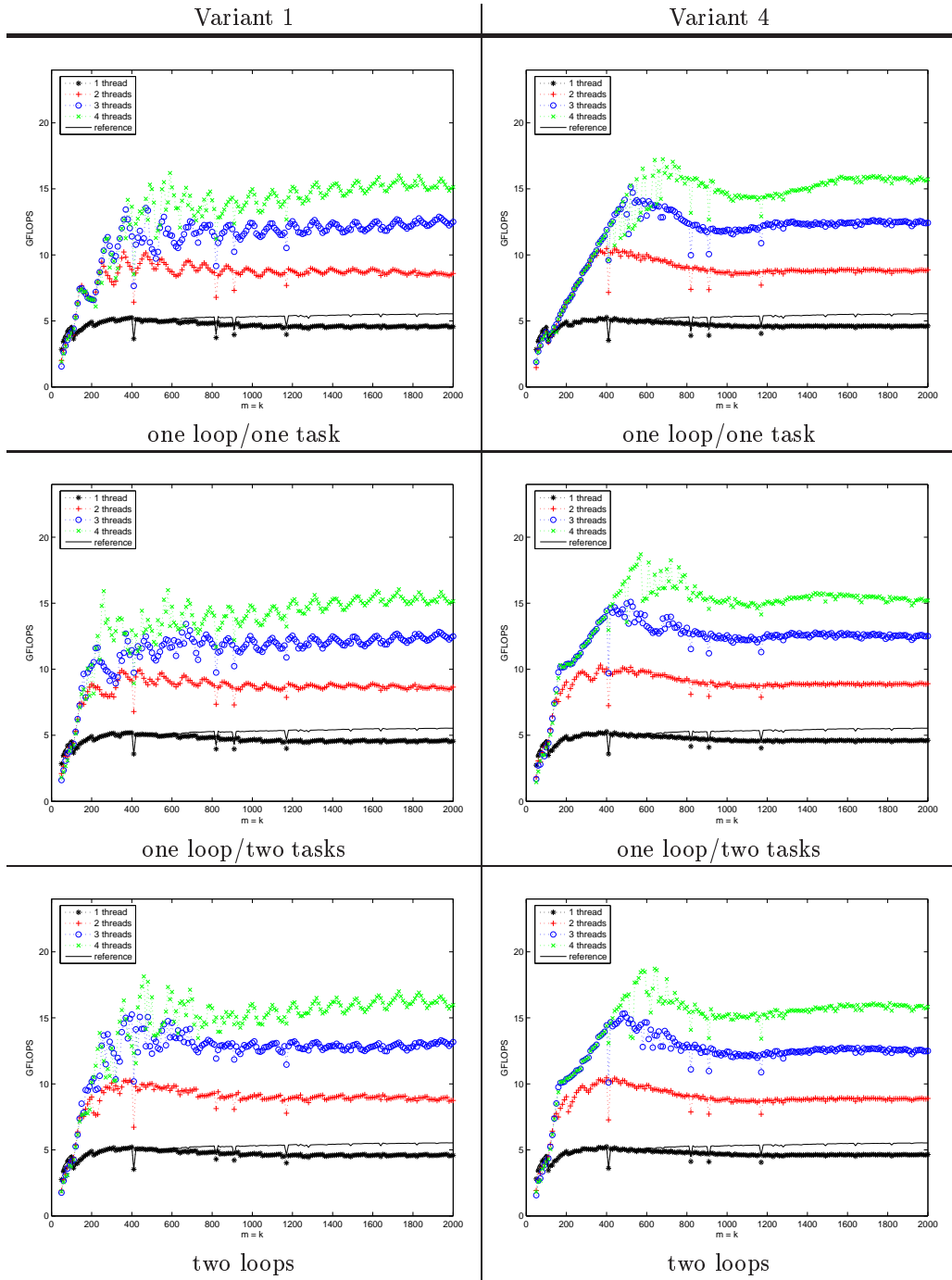


Figure 11: Performance of Variants 1 and 4 when  $C$  and  $A$  are square.

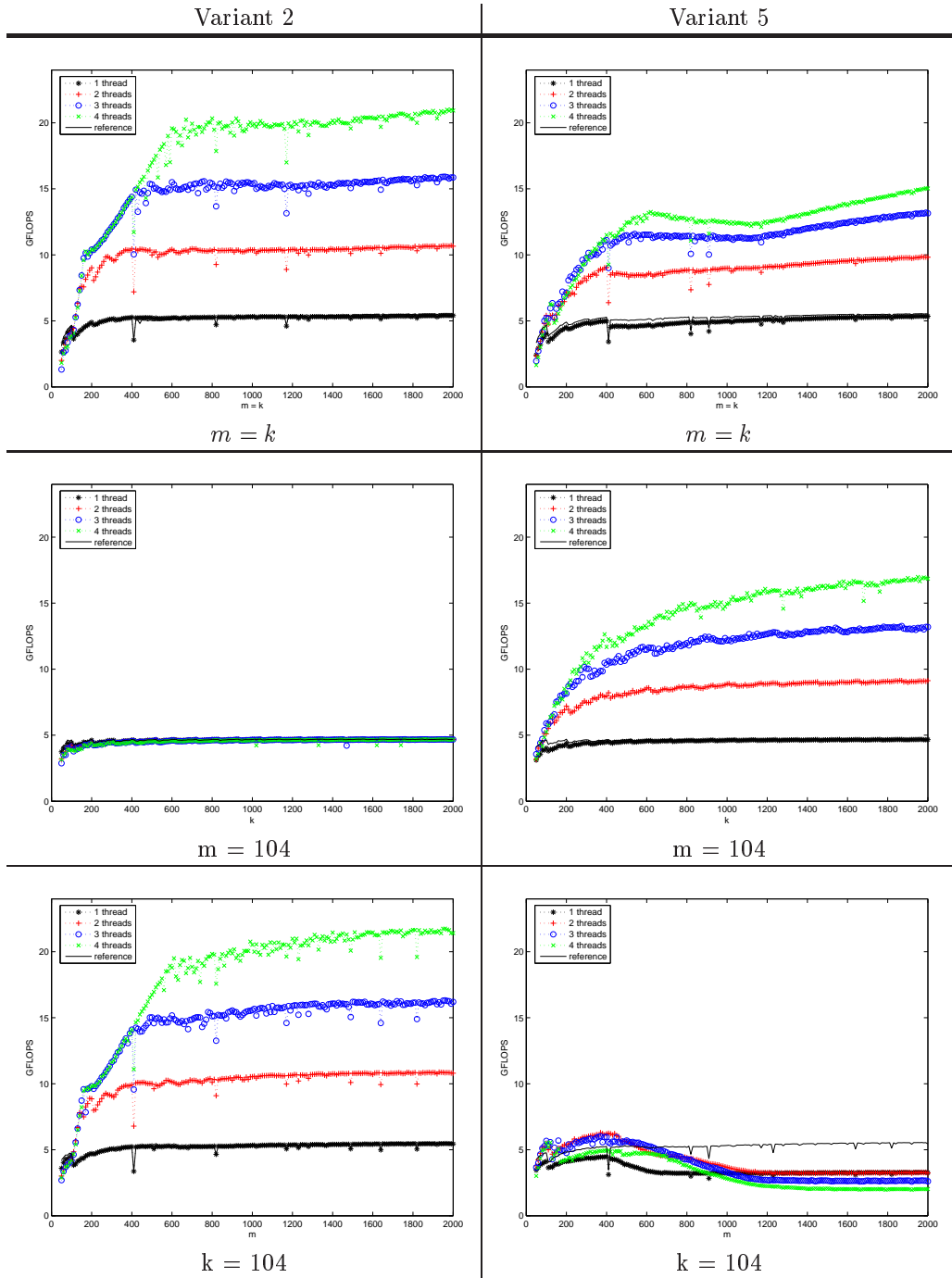


Figure 12: Performance of Variants 2 and 5 for different shaped matrices.

of the contributions to  $C$  is less significant since  $C$  is small. When  $k$  equals 104 and  $m$  is varied, reported in the bottom two graphs, Variant 2 again achieves excellent performance. Variant 5 fails to attain good performance since it is the  $k$  dimension that is split among the tasks, and relatively little computation is performed relative to the overhead associated with the addition of the contributions from different tasks to  $C$ . These last two cases,  $m = 104$  with large  $k$  and  $k = 104$  with large  $m$ , occur as subproblems in different algorithmic variants for operations like the Cholesky factorization.

### 5.3 Simulated results

In Section 3.4 we presented load simulations for task scheduling on four processors. We now show that the simulator accurately predicts the behavior of the implementations, except for small matrices. We calculate the performance as the product of a predicted speedup and the actual asymptotic performance for a serial execution on a single CPU. The speedup is determined by the work of the processor that has to perform the maximum scheduled work. We assume that each user thread is bound to a processor. Using the scheduled task assignments in Section 3.4, the computational rate (in GFLOPS) is estimated by the formula

$$\begin{aligned} \text{GFLOPS} &= \text{predicted speedup} \times \text{GFLOPS by one CPU} \\ &= \frac{\text{total flops performed}}{\max_{i=1}^t (\text{flops performed by processor } i)} \times \text{GFLOPS by one CPU}, \end{aligned}$$

where  $t$  is equal to the number of processors (and threads under our assumptions).

From the graphs in Fig. 10 we determined that a single CPU attains about 5.4 GFLOPS out of a possible 6 GFLOPS for this operation. With this, the graphs in Fig. 13 were generated. In that figure, the measured performance is depicted on the left, while the corresponding simulated performance is predicted on the right. The behavior of the measured and the simulated data is quite similar, except for small matrices.

## 6 Conclusion and Future Directions

In this paper, we have demonstrated how task queues, a proposed feature for OpenMP 3.0, allows code that is devoid of indexing to be elegantly and effectively parallelized. This allows algorithms to be coded at a much higher level of abstraction and improves almost all stages of library development, in our experience. The methodology was applied to the symmetric rank- $k$  update operation, coded using the FLAME/C API. The resulting code was shown to be a minor modification of the FLAME/C implementation. Very good speedup was reported on small SMP systems.

The FLAME approach to deriving and implementing linear algebra operations has been shown to apply to a large number of operations in linear algebra, including most that are supported by the BLAS and LAPACK. We believe that the constructs discussed in this paper apply broadly to algorithms derived and implemented using the FLAME APIs. Thus, the results reported in this paper have broad impact on the development of libraries in the area of dense linear algebra.

The real challenge will be to extend this approach to SMP systems with a large number of processors. Experience with distributed memory architectures tells us that, both theoretically and in practice, eventually it does not suffice to extract parallelism out of the update of blocks of columns or blocks of rows alone. We envision using the presented approach to overcome this problem by viewing the available threads as forming two dimensions. Let us assume that there are  $t$  threads available. Factor  $t$  so that  $r \times s = t$ . Then the presented approach can be used to obtain  $s$ -way parallelism within the presented loop. Within the calls to DGEMM and/or the smaller DSYRK,  $r$ -way parallelism can be achieved. This is in effect similar to the two-dimensional data (and work) decompositions utilized by distributed memory parallel linear algebra packages like ScaLAPACK and PLAPACK [4, 13]. We will investigate this in future research.

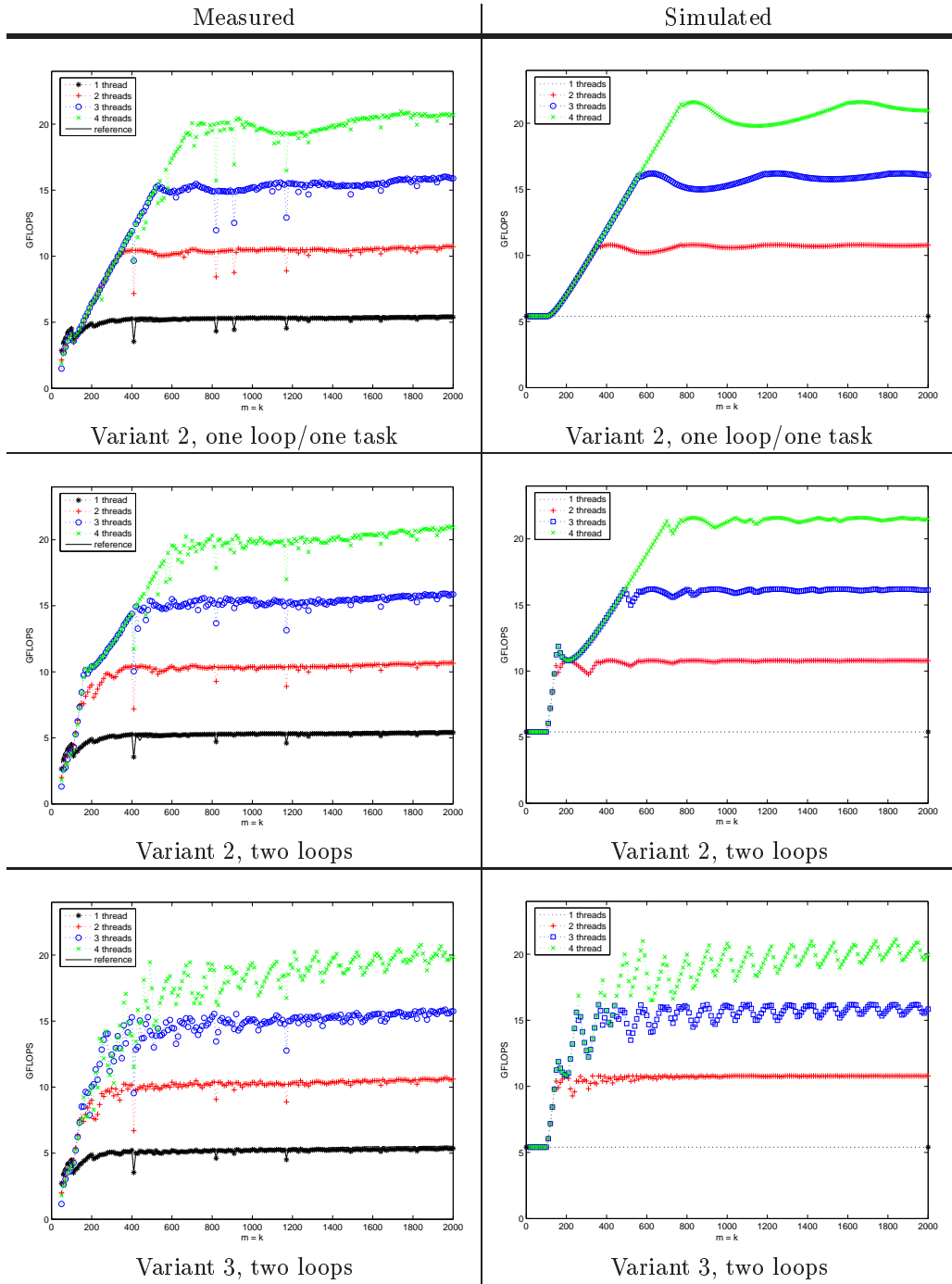


Figure 13: Comparison of measured performance to simulated performance.

## Further information

Additional information regarding the FLAME project can be found at  
<http://www.cs.utexas.edu/users/flame/>

## Acknowledgments

The OpenMP task queue construct was brought to our attention by Dr. Timothy Mattson (Intel). This was the key insight that has allowed us to avoid the reintroduction of indices.

This research was partially sponsored by NSF grants ACI-0305163 and CCF-0342369. The 4 CPU Itanium2 (1.5 GHz) server on which the experiments were conducted was generously donated to our research by the Hewlett-Packard and is administered by UT-Austin's Texas Advanced Computing Center. We also thank Dr. Andrew Chapman and Thuan Cao (both with NEC Solutions (America), Inc.) for their technical advise.

We would like to acknowledge input from Dr. Enrique Quintana-Ortí and Paolo Bientinesi on an advanced draft of this paper.

## References

- [1] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.
- [2] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 2005. to appear.
- [3] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft.*, 2005. to appear.
- [4] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.
- [5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
- [6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.
- [7] Kazushige Goto. <http://www.cs.utexas.edu/users/kgoto>, 2004.
- [8] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.
- [9] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.
- [10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.
- [11] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software*, 29(2):218–243, June 2003.

- [12] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.
- [13] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

## A Additional performance graphs

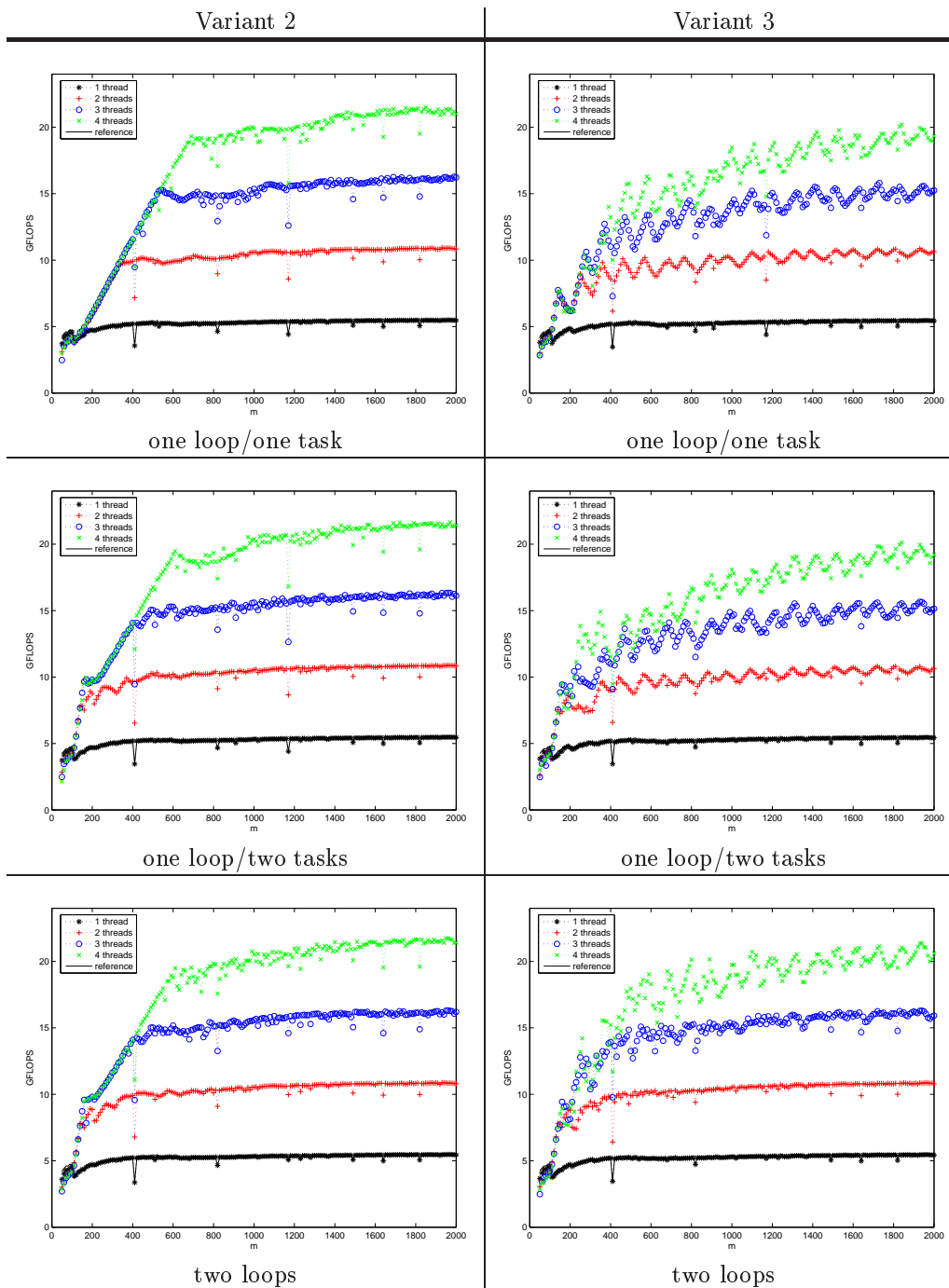


Figure 14: Performance of Variants 2 and 3 when  $A$  is  $m \times 104$ .

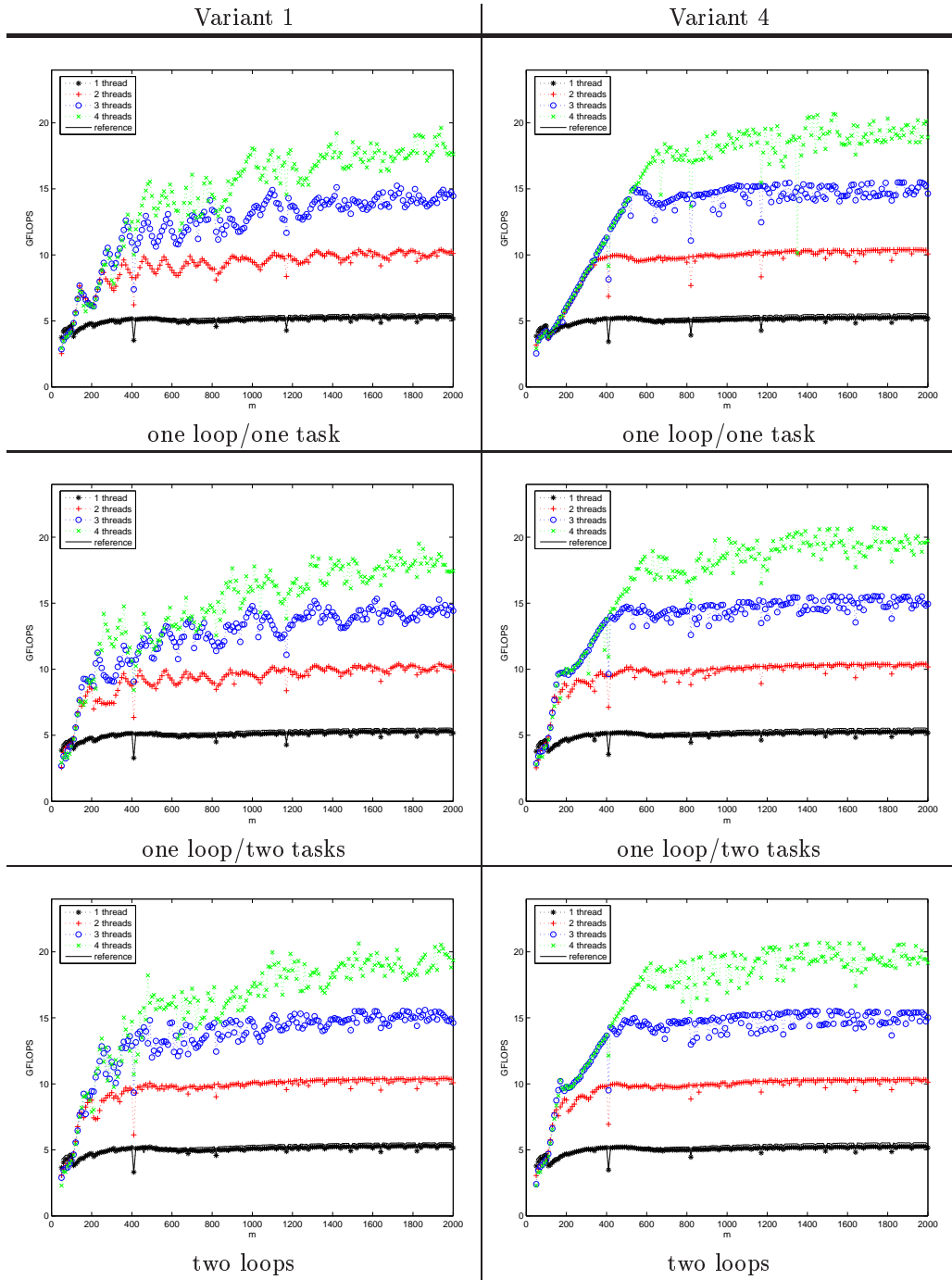


Figure 15: Performance of Variants 1 and 4 when  $A$  is  $m \times 104$ .

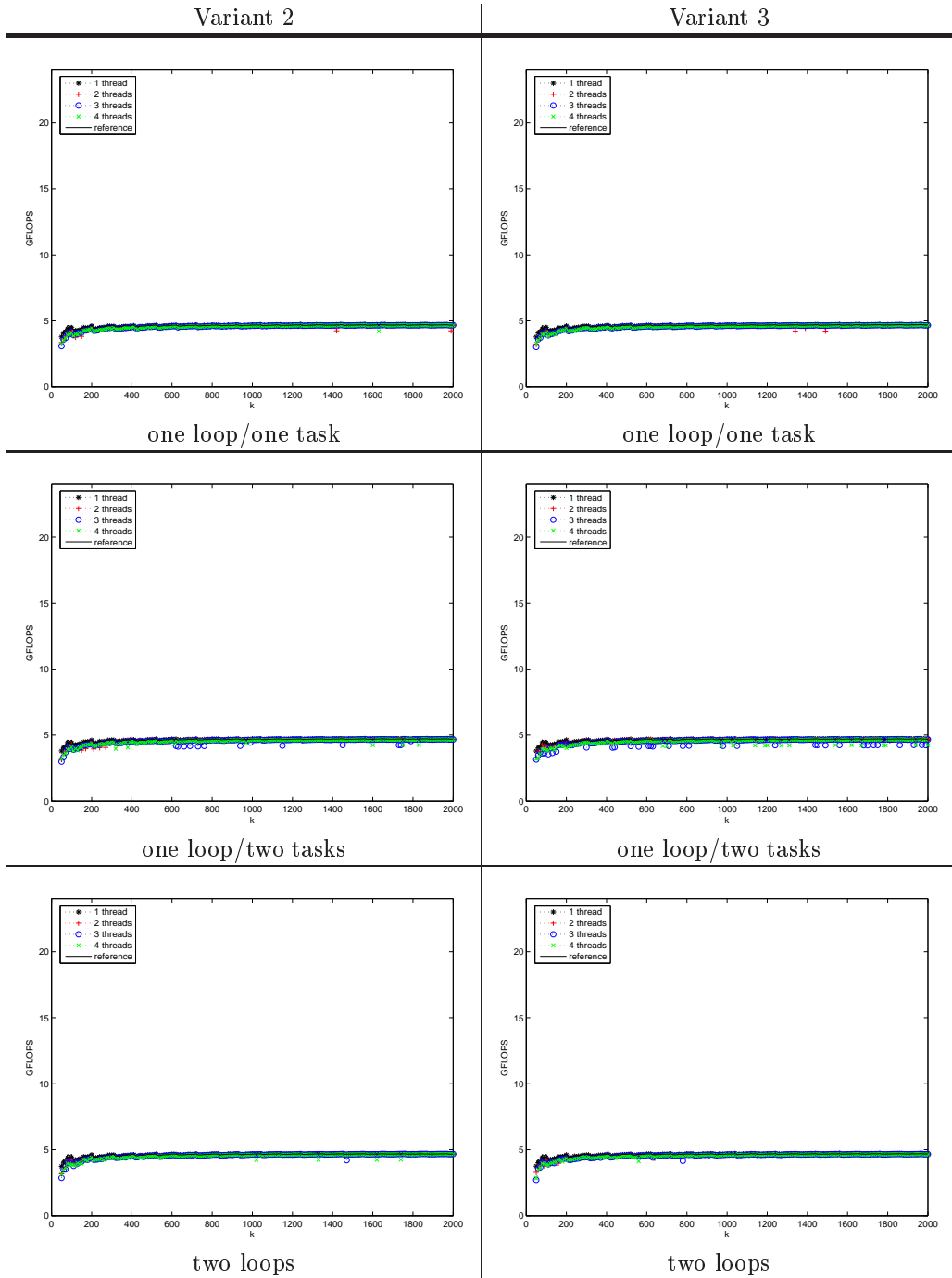


Figure 16: Performance of Variants 2 and 3 when  $A$  is  $104 \times k$ .

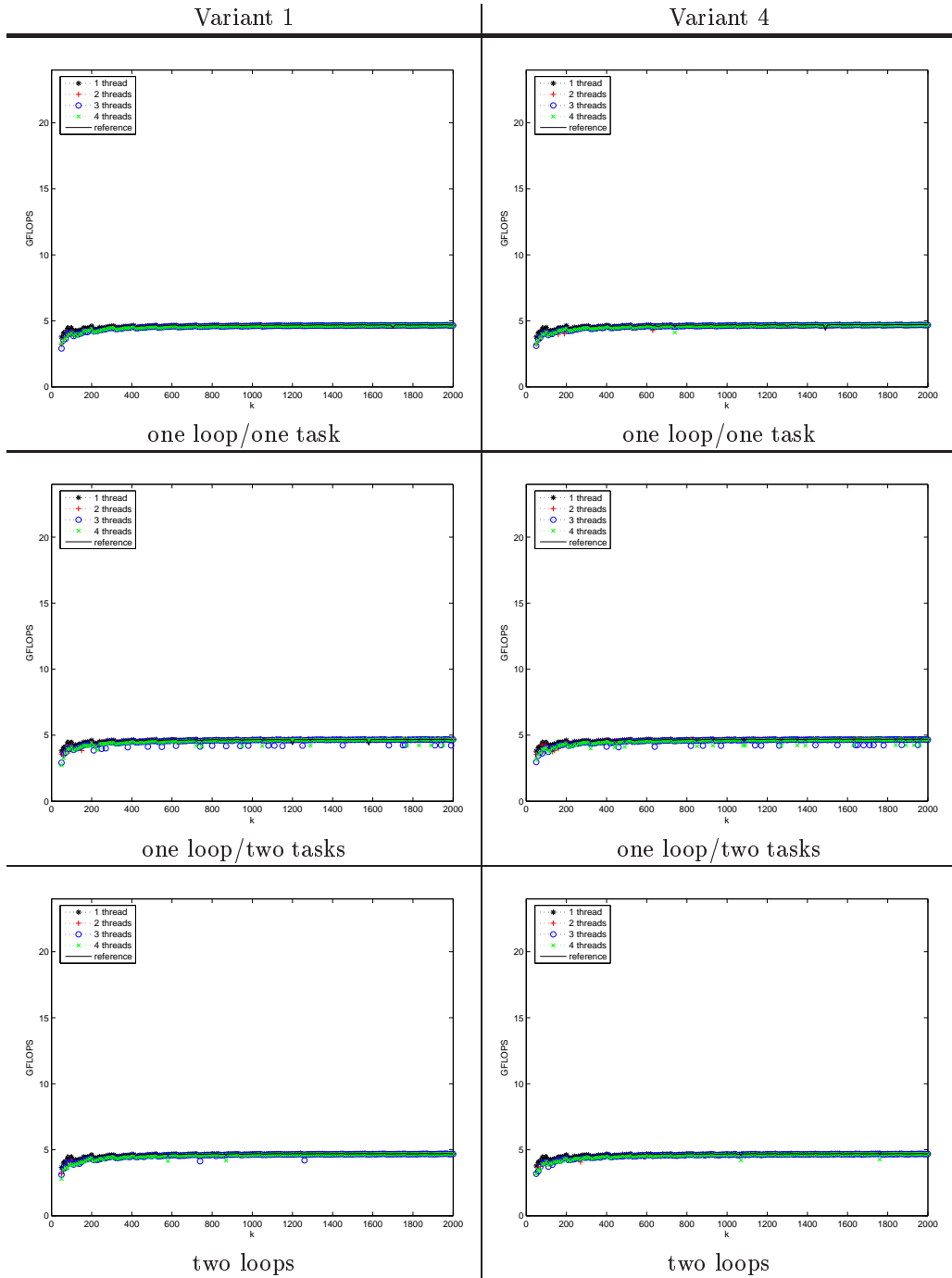


Figure 17: Performance of Variants 1 and 4 when  $A$  is  $104 \times k$ .