Scalable Parallelization of FLAME Code via the Workqueuing Model

FIELD G. VAN ZEE
The University of Texas at Austin
and
PAOLO BIENTINESI
Duke University
and
TZE MENG LOW
The University of Texas at Austin
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

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 loop and array indices, and represents these algorithms as they are formally derived and presented. We report on two implementations of the workqueuing model, neither of which requires the use of explicit indices to specify parallelism. The first implementation uses the experimental `taskq` pragma, which may influence the adoption of a similar construct into OpenMP 3.0. The second workqueuing implementation is domain-specific to FLAME but allows us to illustrate the benefits of sorting tasks according to their computational cost prior to parallel execution. In addition, we discuss how scalable parallelization of dense linear algebra algorithms via OpenMP will require a two-dimensional partitioning of operands much like a 2D data distribution is needed on distributed memory architectures. We illustrate the issues and solutions by discussing the parallelization of the symmetric rank-k update and report impressive performance on an SGI system with 14 Itanium2 processors.

Categories and Subject Descriptors: D.1 [**Software - Programming Techniques - Concurrent Programming**]: Parallel Programming—

General Terms: Algorithms, Performance

Additional Key Words and Phrases: FLAME, OpenMP, SMP, parallel, scalability, workqueuing

Authors' addresses: Field G. Van Zee, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, `field@cs.utexas.edu`. Paolo Bientinesi, Department of Computer Science, Duke University, Durham, NC 27708, `pauldj@cs.duke.edu`. Tze Meng Low, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, `ltm@cs.utexas.edu`. Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, `rvdg@cs.utexas.edu`.

1. INTRODUCTION

FLAME. The Formal Linear Algebra Methods Environment (FLAME) project pursues a systematic methodology for deriving and implementing linear algebra libraries. The methodology is goal-oriented: given a mathematical specification of the operation to be implemented, prescribed steps yield a family of algorithms for computing the operation. A proof of correctness is also given as part of the derivation. The resulting algorithms are expressed at a high level of abstraction, much like one would present algorithms with pseudo-code in a classroom setting [**?**; Bientinesi et al. 2005; Bientinesi 2006]. Application Programming Interfaces (APIs) allow the code to closely resemble the formal algorithm structure, thereby reducing the opportunity for the introduction of "bugs" in the translation from algorithm to implementation. 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) [van de Geijn 1997; Bientinesi et al. 2005]. The scope of FLAME includes the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990], many operations from LAPACK [Anderson et al. 1992], and a large number of operations encountered in Control Theory [Quintana-Ortí and van de Geijn 2003; Bientinesi 2006].

The workqueuing model. Shah et al. [1999] proposed the workqueuing model specifically to overcome the limitations of the OpenMP `for` and `sections` constructs. In OpenMP, workqueuing parallelism would be derived through the use of two new directives: `taskq` and `task`. The workqueuing model offers distinct advantages over conventional parallel OpenMP constructs. Namely, workqueuing provides a method of parallelizing loops that abstracts completely from array and loop indexing; instead, the model is *work*-oriented, allowing the programmer to parallelize independent units of computation created within `for` and `while` statements. The virtues of workqueuing and the clean abstraction of FLAME allow the programmer to quickly parallelize any FLAME algorithm whose subproblems exhibit no inter-dependencies.

High performance. Contrary to conventional wisdom, elegant algorithms need not compromise on performance. Figure 1 gives the reader a general idea of the performance that we can attain in our algorithms with minimal effort. It is worth noting that while FLAME well outperforms the Intel Math Kernel Library (MKL), it is edged out by GotoBLAS for smaller problem sizes. However, this is not surprising. Kazushige Goto, author of GotoBLAS [Goto 2006], frequently collaborates with the FLAME project. Our open exchange of ideas allows him to combine our best findings with his own at a lower, more architecture-aware level.

Contributions. This paper makes the following contributions:

—We show how algorithms written with the FLAME/C API can be naturally parallelized for Symmetric Multi-Processor (SMP) systems by employing the workqueuing model.

—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

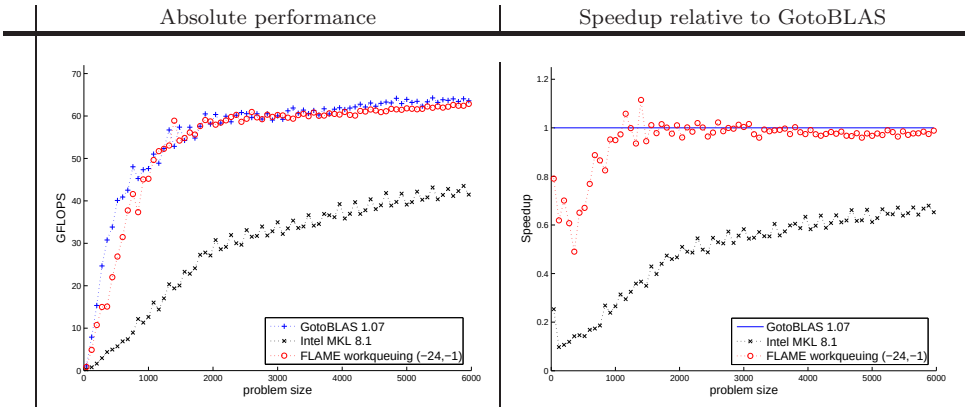

Fig. 1. Performance of parallel SYRK implementations (12 threads on 12 Itanium2 CPUs) using GotoBLAS 1.07, Intel MKL 8.1, and FLAME Variant 2 (parallelized with FLAME workqueuing) when m equals the problem size and $k = 200$. Further details of these experiments may be found in Section 6. *Bottom line:* An elegantly coded FLAME algorithm delivers impressive performance gains over a major vendor math library and quickly converges to perform on par with the cutting-edge GotoBLAS implementation.

such as the Cholesky factorization.

—Implementations are given as a case study for the experimental OpenMP `taskq` pragma and for workqueuing as a method of obtaining parallelism in general.

—A custom workqueuing mechanism is described that allows algorithms encoded with the FLAME/C API to be parallelized via the workqueuing model without relying on the `taskq` pragma, which is currently only implemented within the Intel compilers.

—A compelling argument is made for the need to provide the programmer with more control over task scheduling within the OpenMP workqueuing interface.

—Load-balancing issues are discussed that provide insight into the interplay between different algorithmic variants for computing the same linear algebra operations and the order in which tasks are enqueued.

—A case is made that a 2D work distribution is required for scalability on SMP systems with large numbers of processors much like a 2D data and work distribution is required on distributed memory architectures.

—Performance results are given for an SMP system based on the Intel Itanium2 architecture.

Together, these insights further the state-of-the-art in this area.

Overview. 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 task queues and a custom workqueuing solution (referred to as "FLAME workqueuing") is discussed in Sections 3 and 4. Various issues related to load-balancing and 1D/2D partitioning are discussed in Section 5. Performance

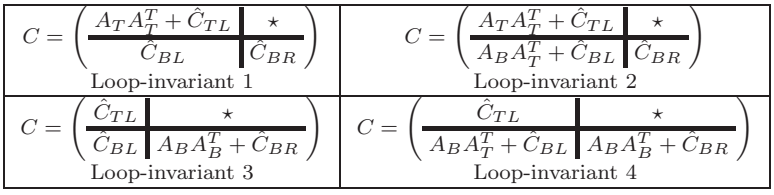

Fig. 2. Loop-invariants for computing SYRK.

experiments are discussed and analyzed in Section 6. Concluding remarks are given in the final section.

2. A CONCRETE EXAMPLE

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

The FLAME methodology describes how to derive linear algebra algorithms from predicates called *loop-invariants*, which describe intermediate states of the operation [Bientinesi et al. 2005]. Low et al. [2005] discuss how to arrive at four loop-invariants for the SYRK operation. These loop-invariants are given in Fig. 2.[1]

For each loop-invariant, the FLAME methodology yields a corresponding algorithmic *variant*. Specifically, Loop-invariant i in Fig. 2 yields algorithmic Variant i in Fig. 3.[2] The loop-body of each algorithm contains two subproblems: a SYRK operation and a GEMM operation, each of which operates on 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 requires delicate indexing into arrays, which exposes opportunities for the introduction of errors. We now illustrate how appropriately defined APIs overcome this problem [Bientinesi et al. 2005]. In Fig. 4, we show an example of FLAME/C code corresponding to Variant 1 in Fig. 3. To understand the code, it suffices to know that `A` and `C` are descriptors for the matrices A and C, respectively. The various routines facilitate the creation of *views* into the data described by `A` and `C`. Think of a variable like `CTL` as a fancy pointer into the array corresponding to matrix 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). The most attractive feature of this code is the *complete absence* of loop and array indexing.

3. WORKQUEUING VIA PROPOSED OPENMP `TASKQ` AND `TASK` DIRECTIVES

OpenMP is a set of compiler directives and library routines that facilitate parallel programming on shared memory systems by allowing a programmer to explicitly

[1]The sub-script notation used in Fig. 2 simply identifies subpartitions of the matrices. This notation is shown in fuller context in Fig. 3 and is further described in [Bientinesi et al. 2005].
[2]A fifth loop-invariant and corresponding algorithmic variant exist for computing SYRK. Early work in this area found that the parallelization of this variant inherently requires heavy synchronization, which significantly limits speedup[Low et al. 2004]. We omit this variant from our discussion due to space constraints.

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

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 $m(A_T) < m(A)$ **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: $C := \text{SYRK\_BLK\_VAR3\_4}(\text{ A, C })$

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 $m(A_B) < m(A)$ **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

Fig. 3. 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. 2. The bottom algorithm implements Variants 3 and 4, corresponding to Loop-invariants 3 and 4 in Fig. 2. Variants 1 and 2 share the same loop-body updates as Variants 4 and 3, respectively. Variants 1 and 2 sweep through C from the top-left to the bottom-right and A from top to bottom, while Variants 3 and 4 traverse the matrices in the opposite directions.

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

Fig. 4. FLAME/C code for a blocked implementation of Variant 1.

specify regions of code that can be executed by simultaneous threads of execution [OpenMP Architecture Review Board 2006].

Shah et al. [1999] point out limitations of the primary constructs for creating OpenMP parallelism: the `parallel for` and `sections` directives. They note that the number of iterations in OpenMP `for` loops must be computable upon first entering the loop, precluding its use in many applications, such as traversing a linked-list of unknown length. Similarly, a `sections` construct containing n independent regions of computation, each marked by a `section` directive, is limited

to achieving n-way parallelism [Shah et al. 1999]. The workqueuing model was proposed specifically to overcome these limitations.

3.1 The `taskq` and `task` pragmas

A proposed OpenMP instantiation of the workqueuing model consists of two new directives: `taskq` and `task`. Conceptually, encountering a `taskq` directive causes the main thread to create an empty workqueue (or task queue). The code within the `taskq` scope is executed sequentially. As `task` directives are encountered, the code associated with the `task` block is encapsulated and enqueued as a unit of work onto the task queue. A number of other threads begin dequeuing and executing tasks from the queue according to a first-in/first-out (FIFO) scheduling policy. The main thread joins the others in processing tasks as soon as enqueuing is complete. When all tasks have been completed, the threads synchronize at the end of the `taskq` scope and continue through the program.

This OpenMP instantiation of the workqueuing model features two noteworthy properties:

—Workqueuing is *dynamic*, unlike the `sections` directive which lexically encodes the degree of parallelism into the source code at compile time.

—Workqueuing is *flexible*, unlike the `parallel for` directive which provides parallelism only for indexed `for` loops and also requires the number of instantiated work-shared task units to be computable at runtime.

These two properties of OpenMP workqueuing enable an attractive new mechanism for expressing parallelism within FLAME/C algorithm implementations.

3.2 Parallelization of SYRK

In Fig. 5 we show how the `while` loop in Fig. 4 can be annotated with OpenMP directives to create parallel tasks via the task queue mechanism. In Fig. 5:

—**Line 13** establishes the `taskq` block.

—**Line 28** starts a section of code that defines a task to be added to the task queue. A single thread executes the `while` loop, enqueuing tasks as they are encountered. The descriptors `A0`, `A1`, `C10`, and `C11` change with each iteration of the loop. The values of these descriptors must be "captured" at the time each task is enqueued so that the thread that dequeues the task will have the correct values to pass along to `FLA_Gemm` and `FLA_Syrk`.

—**Line 34** ends the scope of the task being added to the queue.

—**Line 46** ends the scope of the `taskq` block. Here, all threads are synchronized.

Clearly, task queues provide a simple mechanism for directing the parallel execution in this code.

3.3 Options

In Fig. 5 the subproblems corresponding to the calls to `FLA_Gemm` and `FLA_Syrk` are independent and therefore can be executed in any order and/or queued as separate tasks. This is apparent by inspecting the algorithm itself. However, Low et al. [2005] discuss how to systematically detect the presence of independent loop

```
13  #pragma intel omp parallel taskq
14  {
15    while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
16      b = min( FLA_Obj_length( AB ), nb_alg );
17
18      FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
19                            /*************/  /*********************/
20                                               &C10, /**/ &C11, &C12,
21                             CBL, /**/ CBR,    &C20, /**/ &C21, &C22,
22                             b, b, FLA_BR );
23      FLA_Repart_2x1_to_3x1( AT,                &A0,
24                           /* ** */           /* ** */
25                                               &A1,
26                             AB,                &A2,   b, FLA_BOTTOM );
27      /*-----------------------------------------------------------*/
28      #pragma intel omp task captureprivate( A0, A1, C10, C11 )
29      {
30        FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
31                  FLA_ONE, A0, A1, FLA_ONE, C10 );
32        FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
33                  FLA_ONE, A1, FLA_ONE, C11 );
34      } /* end task */
35      /*-----------------------------------------------------------*/
36      FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
37                                                  C10, C11, /**/ C12,
38                                /**************/ /******************/
39                                &CBL, /**/ &CBR,  C20, C21, /**/ C22,
40                                FLA_TL );
41      FLA_Cont_with_3x1_to_2x1( &AT,                 A0,
42                                                     A1,
43                              /* ** */            /* ** */
44                                &AB,                 A2,     FLA_TOP );
45    }
46  }   /* end of taskq */
```

Fig. 5. FLAME/C code from Fig. 4 parallelized using OpenMP task queue directives.

iterations by inspecting the loop-invariants of the SYRK operation. Furthermore, we may observe that the two updates within the loop-body are independent of one another within a single iteration. Given these two observations, we may modify our original parallelization shown in Fig. 5 as follows.

One option is to split the single task in the loop-body of Fig. 5 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 );
}
```

A further observation is that the computations $C_{10} := A_1 A_0^T + C_{10}$ and $C_{11} :=$

```
#pragma intel omp parallel taskq
{
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) )
  {
    [ ... ]
    #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 ) )
  {
    [ ... ]
    #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 */
```

Fig. 6. Outline of how the loop in Fig. 5 can be implemented as two loops.

$A_1 A_1^T + C_{11}$ (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. Notice that $n(C_{11})$ remains constant across iterations. Thus, the number of FLOPs required to compute the update to C_{11} is fixed. In contrast, $n(C_{10})$ grows linearly with each iteration of the loop and therefore the number of FLOPs required to update C_{10} increases proportionally as the algorithm iterates. This is unfortunate since costly tasks at the end of a scheduling queue can create a large load imbalance as we will show later in Fig. 7.

One way to overcome this problem is to execute the loop in reverse order (in compiler terms: apply a *loop reversal* transformation), since this would create the more costly tasks first. Variants 4 and 3 in Fig. 3 execute the loops in Variants 1 and 2 in reverse, respectively.[3] 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. We will later show that differences in performance can be observed for different variants.

An alternative option replaces the single loop in in Fig. 5 with two loops (in compiler terms: apply a *loop fission* transformation): the first loop for enqueuing the tasks that update C_{10} and the second for enqueuing tasks that update C_{11}, as illustrated in Fig. 6. The updates to C_{11} incur a smaller cost and result in fixed-sized tasks, compared to the updates of C_{10} which result in larger, variable-sized tasks to be enqueued. These smaller tasks help balance the workload among

[3]This illustrates the value of the FLAME methodology which can systematically find algorithmic variants that have different strengths and weaknesses.

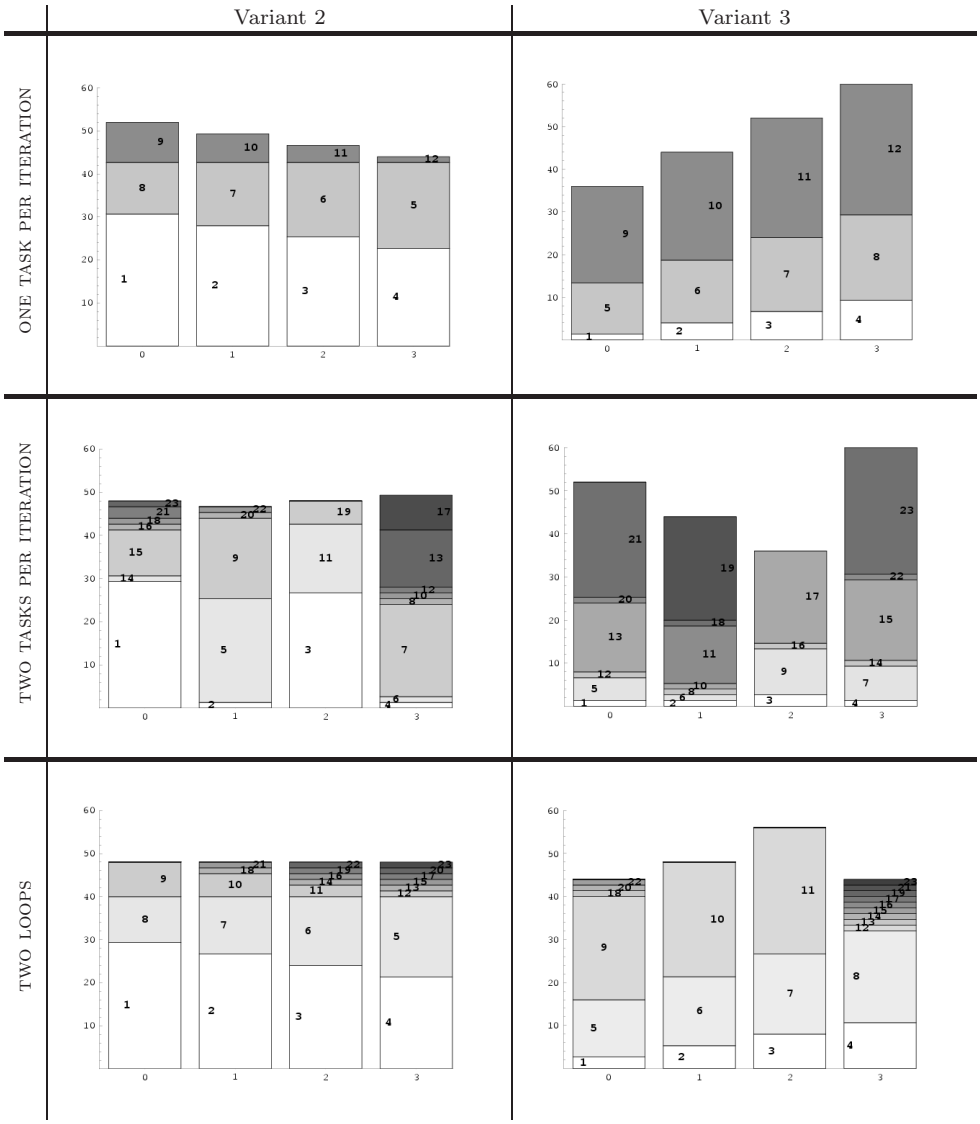

Fig. 7. Simulated OpenMP task queues scheduling of tasks to four threads for Variants 2 and 3 when $m = 2400$, $k = 200$, and the blocksize equals 200. The sum of the heights of the rectangles in each x-axis column correspond to the total amount of work assigned to each thread. The y-axis is in units of FLOPs $\times 10^6$ (millions of FLOPs). Load balance is ideal when all threads receive equal work. *Bottom line:* Load balance is determined by the number of tasks created, the cost of each task, and the order in which tasks are enqueued.

threads before the threads synchronize at the end of the `taskq` block.

3.4 An illustration of the benefits of different options

The expected differences in performance are illustrated for Variants 2 and 3 in Fig. 7. (Recall that Variants 2 and 3 are identical except that their loops iterate in opposite

directions.) In this figure, we simulate the scheduling of tasks to four threads for the different options described previously, where matrix C is 2400×2400, A is 2400×200, and the algorithmic blocksize b in Fig. 3 equals 200 (except possibly during the last iteration). Each of the tasks is represented by a box that has a height proportional to the number of FLOPs required to complete the task. The y-axis itself is represented in units of millions of FLOPs. The integers in the boxes indicate the order in which the tasks are enqueued in the task queue.

The simulation results shown in Fig. 7 were built upon a simplified workqueuing model that makes the following assumptions:

—Threads are idle when the computation begins.

—Enqueuing is instantaneous.

—When processing tasks, each thread computes at the same rate.

—Upon completing a task, the thread in question will dequeue a new task instantaneously. If the queue is empty, the thread becomes idle.

—The computation completes when the queue is empty and all threads are idle.

We see that Variant 2 in general performs better than Variant 3 since the cost of each variable-sized task decreases towards later iterations, allowing work 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. Variant 2 achieves near-perfect load-balance by splitting the loop into two loops, with variable-sized tasks enqueued in the first loop while the smaller fixed-sized tasks are enqueued by the second. This change provides only a modest improvement to Variant 3; the imbalance created by the increasing cost of variable-sized tasks is simply too large to erase with the smaller subproblems.

4. AN ALTERNATE IMPLEMENTATION: FLAME WORKQUEUING

As of this writing, the proposed OpenMP task queuing mechanism has two shortcomings. First, task queues are an experimental implementation: to our knowledge it is currently only supported by the Intel compiler [Su et al. 2002]. The current official OpenMP specification (version 2.5) does not provide any workqueuing constructs.[4] Second, our discussion in Section 3.4 suggests that the OpenMP task scheduling does not always result in good load-balance among threads. Later in this section, we discuss a custom implementation for FLAME that addresses these shortcomings.

4.1 Theoretical basis for dequeuing tasks largest to smallest

The problem of scheduling tasks among a set of threads is a variation of the Bin-Packing Problem in which a collection of objects are packed into a set of bins such that the total weight or volume of each bin does not exceed a given threshold [Weisstein 2006]. In our case of workqueuing, the computational tasks correspond to objects being packed (or scheduled) while the task volumes correspond to their computational costs, which can be approximated by counting FLOPs. Threads correspond to the bins into which the objects are packed. The target bin capacity corresponds

[4]It is possible that workqueuing support will be added to version 3.0 of the OpenMP specification.

to the sum of the tasks to be executed divided by the number of threads—that is, the ideal amount of computation per thread. J. D. Ullman [Garey et al. 1973] proves that a naive algorithm, one that packs objects into the first available bin with space, is suboptimal by as much as 70% [Weisstein 2006]. Johnson [1973] shows that an algorithm that first sorts the objects from largest to smallest will be *at most* 22% suboptimal [Weisstein 2006]. This suggests that, in general, an execution of tasks scheduled from largest to smallest (with respect to computational cost) will always perform reasonably well compared to executions based upon other task schedulings.

4.2 Motivation for sorting tasks over changing the algorithm

Ideally, an application loop will create independent subproblems of equal cost. Such cases usually parallelize well with minimal effort. Other loops may naturally create subproblems that decrease monotonically in cost. Given the discussion in the previous section, this is the next best scenario if subproblems of equal cost are not possible. If the loop creates subproblems that increase monotonically in cost, then a simple loop reversal will cause the tasks to be enqueued in the desired order. As we saw in Fig. 7, sometimes even more changes are needed, as the best scheduling also required splitting the main algorithm loop into two separate loops to enqueue variable-sized tasks first, followed by smaller fixed-sized tasks. But this method of changing the algorithm to induce a desirable scheduling is suboptimal for two reasons. First, it requires non-trivial changes to the algorithm code. Uniquely, FLAME codes resemble the underlying algorithm so closely that changing the source code will tend to also obscure the algorithm itself. More generally, changing the algorithm code is also less than desirable from a code maintenance perspective. Second, it is possible that there does not exist a reasonable loop or code transformation for a given algorithm that enqueues tasks from largest to smallest. For example, it is conceivable that some loops may create subproblems whose cost neither increases nor decreases monotonically.

An alternate way to ensure a desirable ordering of tasks within the queue, one that is less disruptive to the algorithm and therefore more portable, is to allow the application loop to enqueue tasks normally and then sort the queue before parallel execution. This solution is quite flexible and its disadvantages are minor.[5]

Unfortunately, the OpenMP `taskq` construct as specified provides no such sorting mechanism [Shah et al. 1999; Su et al. 2002]. The responsibility of ensuring load-balancing through a desirable task ordering is left to the programmer. Furthermore, experience, as well as results discussed later in this paper, show that the performance penalty for executing tasks from an unsorted queue can be quite severe.

4.3 FLAME workqueuing

To circumvent the shortcomings of OpenMP task queues we have implemented a custom workqueuing solution that behaves much like task queues (though it is

[5]Threads must wait for sorting to finish before parallel execution of the queue can begin. The sorting itself is an $O(nlog(n))$ operation that will most likely not adversely affect performance for our applications.

domain-specific to FLAME), featuring the following enhancements:

—*Portability.* Our implementation does not use the `taskq` or `task` constructs and thus works on any conventional implementation of OpenMP. In fact, FLAME workqueuing abstracts all implementation details from the API, potentially allowing us to replace OpenMP altogether with a more portable threading mechanism such as POSIX threads.

—*Task sorting.* FLAME workqueuing automatically sorts the task queue according to each task's estimated cost (FLOP count) before parallel execution begins.

A programmer enqueues a routine by replacing it with a corresponding preprocessor macro. The macro inserts an invocation to `FLA_Queue_push()`, which uses the data associated with the original function call to create a task structure. This structure is added to the queue, which is implemented as a linked list. After enqueuing is complete, the programmer signals that execution may begin by calling `FLA_Queue_exec()`, the definition of which is shown in Fig. 8. The linked list of tasks is indexed and then sorted according to approximate FLOP cost using Quicksort. Finally, the sorted queue's tasks are executed in parallel using a `parallel for` directive with dynamic scheduling.

The reader should note that FLAME workqueuing was not intended to replace the functionality of OpenMP task queues. Clearly the implementors of task queues are providing a generalized solution that leverages the privileged access that the compiler has to the code before compilation. Our workqueuing implementation was designed primarily to minimize disturbance to conventional algorithms implemented with the FLAME/C API while providing a mechanism to sort the contents of the queue prior to execution.

Though they are motivated by the same conceptual model, the FLAME workqueuing API uses a different syntax than the task queue constructs. Figure 9 shows how subproblems are enqueued as tasks under the FLAME workqueuing API. The programmer must initialize the workqueuing environment by invoking `FLA_Queue_init` and likewise call `FLA_Queue_finalize` to free internal resources when workqueuing is no longer needed. The API contains no direct analog to the `taskq` directive. Consequently, nested queuing is not supported and all tasks are implicitly enqueued onto the same global queue. Also, where as OpenMP task queues automatically dispatch threads as soon as the first task is enqueued, the programmer of the FLAME workqueuing API must explicitly invoke parallel execution after enqueuing is complete by calling `FLA_Queue_exec`.[6] Presumably, this may cause FLAME workqueuing to slightly under-perform a similar code that uses OpenMP task queues when the algorithm happens to enqueue tasks in the desired sorted order. However, performance results in Section 6.1 show that this penalty is negli-

[6]In order to guarantee a fully sorted queue, the programmer must allow enqueuing to finish. While dispatching threads immediately may sound desirable, it denies the workqueuing implementation the opportunity to sort the queue before execution. Figure 13 confirms that the benefits of deferring execution to allow sorting to take place dwarf the added serialization costs. Conceivably, there exist applications that create many small tasks for which this assertion may not hold. In that case, sorting is probably not necessary anyway due to the fact that many small tasks tend to inherently yield good load-balancing under a FIFO scheduling policy.

```
1  void FLA_Queue_exec()
2  {
3    int        i, n_tasks;
4    FLA_Task** task_array;
5    FLA_Task*  t;
6
7    /* The queue is full, so we may now create an index of each task. */
8    FLA_queue_create_task_array();
9
10   /* Sort the task_array with standard C library function qsort(). */
11   FLA_queue_sort_task_array();
12
13   /* Copy the task_array pointer and n_tasks integer locally to
14      satisfy the OpenMP compiler. */
15   n_tasks    = task_queue.n_tasks;
16   task_array = task_queue.task_array;
17
18   /* Iterate over the task queue using the random-access task_array.
19      Note that we iterate backwards because qsort() sorts in ascending
20      order, while we want to execute the tasks in descending order. */
21   #pragma omp parallel for shared( task_array, n_tasks ) \
22                            private( i, t ) \
23                            schedule( dynamic, 1 )
24   for( i = n_tasks - 1; i >= 0; --i )
25   {
26     t = task_array[ i ];
27     FLA_queue_exec_task( t );
28   }
29
30   /* Flush the queue: walk the task_array and free() each element. */
31   FLA_queue_flush();
32
33   /* Free the task_array now that all tasks have been executed. */
34   FLA_queue_free_task_array();
35 }
```

Fig. 8. Code fragment from the FLAME workqueuing implementation: definition of `FLA_Queue_exec()`.

gible given the relatively small number of tasks enqueued when executing a parallel SYRK operation.

5. BLOCKING AND PARTITIONING

In our previous discussions of the parallel workqueuing codes shown in Figs. 5 and 9 we did not mention a subtle but important detail: how does one determine the algorithmic blocksize b? In the workqueuing-enabled SYRK implementations, this blocksize determines the dimensions of the subproblems created, which corresponds directly to the cost of the tasks placed onto the queue. Determining an appropriate blocksize leads us to a somewhat more general discussion of how best to partition the computation into tasks. In this section we describe some methods of partitioning submatrices that may help us attain better load-balancing among the threads.

```
13    FLA_Queue_init();
14    [ ... ]
15    while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
16      b = min( FLA_Obj_length( AB ), nb_alg );
17
18      FLA_Repart_2x1_to_3x1( AT,                &A0,
19                           /* ** */            /* ** */
20                                                &A1,
21                             AB,                &A2,        b, FLA_BOTTOM );
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      /*-----------------------------------------------------------*/
28      ENQUEUE_FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
29                        FLA_ONE, A1, A0, FLA_ONE, C10 );
30      ENQUEUE_FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
31                        FLA_ONE, A1, FLA_ONE, C11 );
32      /*-----------------------------------------------------------*/
33      FLA_Cont_with_3x1_to_2x1( &AT,                A0,
34                                                    A1,
35                              /* ** */            /* ** */
36                                &AB,                A2,     FLA_TOP );
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    }
43    [ ... ]
44    FLA_Queue_exec();
45    [ ... ]
46    FLA_Queue_finalize();
```

Fig. 9. FLAME/C code from Fig. 4 parallelized using FLAME workqueuing.

5.1 Proportional blocking

Notice that for Variant 2 in Fig. 3 the cost (in FLOPs) of the variable-sized `FLA_Gemm` task created in the ith iteration decreases linearly with i while the cost of the `FLA_Syrk` task remains constant (and relatively small) across all iterations. Furthermore, the number of tasks generated varies with the m dimension. Naively, we may choose a blocksize *a priori* without regard to the problem size. We refer to this method of choosing b as an arbitrary fixed value as *constant* blocking. However, the simulation in Fig. 10 reveals that this method results in load-imbalance and diminished parallel performance for certain smaller matrix dimensions.

In order to circumvent this problem, let us chose the algorithmic blocksize b so that the algorithm cycles through $2t$ iterations, regardless of the problem size determined by the m dimension. Under this scenario, the algorithm creates $2t - 1$ variable-sized tasks and $2t$ fixed-sized tasks, where t equals the number of processor-

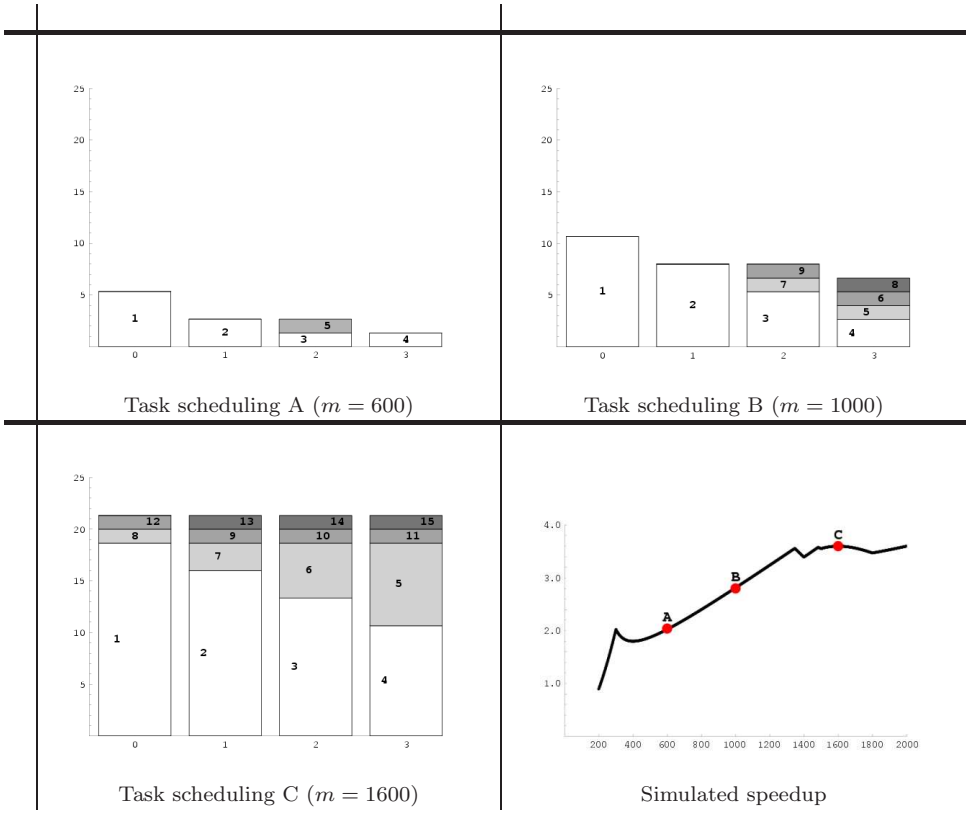

Fig. 10. Simulated OpenMP task queues scheduling of tasks to four threads for Variant 2 shown for select values of m when $k = 200$. The "two loop" parallelization for Variant 2 was used, which enqueues tasks in sorted order. Also, the blocksize used was 200. The axes' units of the three task scheduling graphs is similar to that of Fig. 7. The corresponding speedup of each task scheduling is marked on the curve shown in the lower-right graph, in which we assume `dgemm` and `dsyrk` consistently perform at 90% of peak efficiency. *Bottom line:* Given a one-dimensional constant blocking, some threads fall idle before others due to poor distribution of work, where the largest variable-sized GEMM tasks become bottlenecks. This load-imbalance hinders parallel speedup and thus results in limited performance for many smaller problem sizes.

bound[7] threads participating in the computation. Under our simplified task scheduling model, this careful choice of blocksize *always* yields an ideal load balance similar to the scheduling shown in the bottom-left graph of Fig. 10. Therefore, a good value for b may be chosen to equal $\frac{m}{2t}$. We refer to this method of choosing b as a function

[7]The operating system should, either by default or at the behest of the programmer, "bind" each thread to a single unique processor to ensure optimal performance. Under version 2.6 of the Linux kernel, programmers may explicitly request this behavior by setting the thread's CPU *affinity* via the `sched_setaffinity` function [Dow 2005]. Experience suggests that setting the CPU affinity of threads in a parallel computation is typically desirable. Otherwise, the scheduler may direct threads to migrate between processors. This situation may lead to significant performance degradation as a recently migrated thread may experience a higher latency while accessing data resident on the cache (or local memory) associated with its previous CPU.

of the partitioned dimension m and number of threads t as *proportional* blocking.[8]

Notice, however, that `dgemm` executes more efficiently the larger b is chosen. Thus, for SYRK problems with small dimensions, we expect proportional blocking will yield better load-balance but possibly at the expense of worse performance by each thread.

5.2 Partitioning in two dimensions

So far, we have only considered a single blocksize b in the SYRK algorithms shown in Fig. 3. Recall that this blocksize determines two properties of the tasks enqueued during each iteration. For Variant 2, these two properties are the dimension (order) of C_{11}, which is updated in the `FLA_Syrk` subproblem, and the dimensions of the C_{21} panel that is updated by the `FLA_Gemm` subproblem. The C_{11} submatrix is already small and creates a task of constant cost. However, the C_{21} submatrix varies in size and provides us with the opportunity to further partition along its m (row) dimension. Let us consider an alternate version of Variant 2 in which we replace the call to `FLA_Gemm` with a GEMM variant that partitions A_2 and C_{21} along the m dimension with a blocksize that is independent of the value of b used thus far. Partitioning A_2 and C_{21} causes the the parallelized algorithm to enqueue a larger number of somewhat smaller variable-sized tasks *for each* iteration of the Variant 2 `while` loop. Figure 11 shows the FLAME/C code that implements this variant of GEMM. This effectively allows the workqueuing analog of a two-dimensional data decomposition. The advantages of this approach are two-fold:

—First, attaining good load-balance is easier when enqueuing smaller partitioned `FLA_Gemm` tasks than when the subproblems are enqueued unpartitioned. This is simply a consequence of the fact that smaller tasks more easily allow a scheduling that distributes work equally across all threads. This behavior holds regardless of whether the task queue is sorted.

—Second, we may leverage proportional blocking so that the load-balance of a 2D partitioning remains ideal under our model. By using proportional blocking to partition A_2 and C_{21} along their m dimensions into q subpartitions of roughly equal size, we may choose the SYRK blocksize b to equal $\frac{mq}{2t}$ (resulting in $\frac{m}{b} = \frac{m}{(\frac{mq}{2t})} = \frac{2t}{q}$ iterations in the SYRK algorithm). This blocksize is larger than the value proposed in Section 5.1 and thus should allow the `dgemm` implementation to perform more efficiently, especially for smaller problems. However, this will also proportionally reduce the number of SYRK subproblem tasks from $2t$ to $\frac{2t}{q}$ and similarly increase their costs. This smaller number of more costly fixed-sized tasks may be more difficult to divide equally among t threads for $q > 2$.

We show later in Section 6.1 that a two-dimensional partitioning with proportional blocking allows us to sustain good performance per thread *and* good load-balance across threads for smaller problem sizes than would be possible with a one-dimensional partitioning.

[8]Given that $b = \frac{m}{2t}$, it follows that $m = 2bt$. The former equation suggests how to choose, as a function of the problem size, the largest possible b that still induces ideal load-balance. The latter equation reveals the smallest problem size m for which constant blocking will yield peak load-balance. The bottom-left graph in Fig. 10 illustrates both of these scenarios.

```
1  FLA_Error Gemm_blk_var1( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nb_alg )
2  {
3    FLA_Obj AT,        A0,        CT,        C0,
4            AB,        A1,        CB,        C1,
5                       A2,                   C2;
6    int b;
7
8    FLA_Part_2x1( A,    &AT,
9                        &AB,            0, FLA_TOP );
10   FLA_Part_2x1( C,    &CT,
11                       &CB,            0, FLA_TOP );
12
13   while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
14     b = FLA_Task_compute_blocksize( A, AT, FLA_TOP, nb_alg );
15
16     FLA_Repart_2x1_to_3x1( AT,                &A0,
17                          /* ** */            /* ** */
18                                               &A1,
19                            AB,                &A2,        b, FLA_BOTTOM );
20     FLA_Repart_2x1_to_3x1( CT,                &C0,
21                          /* ** */            /* ** */
22                                               &C1,
23                            CB,                &C2,        b, FLA_BOTTOM );
24     /*-------------------------------------------------------------*/
25     ENQUEUE_FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
26                       FLA_ONE, A1, B, FLA_ONE, C1 );
27     /*-------------------------------------------------------------*/
28     FLA_Cont_with_3x1_to_2x1( &AT,               A0,
29                                                  A1,
30                              /* ** */           /* ** */
31                                &AB,               A2,     FLA_TOP );
32     FLA_Cont_with_3x1_to_2x1( &CT,               C0,
33                                                  C1,
34                              /* ** */           /* ** */
35                                &CB,               C2,     FLA_TOP );
36   }
37   return FLA_SUCCESS;
38 }
```

Fig. 11. FLAME/C code, ready for use with FLAME workqueuing, implementing a variant of GEMM that transposes B and partitions A and C along the m dimension. This code may be called in our FLAME/C variant 2 of SYRK instead of calling `FLA_Gemm` directly.

6. EXPERIMENTS

Workqueuing comparison. To demonstrate the effect of algorithmic variants and task partitioning methods on performance we parallelized each of the four variants using OpenMP task queues and FLAME workqueuing. In the case of the variants using OpenMP task queues, we applied the code transformation options described in Section 3.3 to arrive at three parallelized configurations: a simple insertion of the task queue directives with one task in the loop-body; the separation of the two updates to create two tasks in the loop-body; and the separation of the fixed- and variable-sized tasks into two separate loops. For FLAME workqueuing, we implemented Variants 1 through 4 and replaced the invocation of `FLA_Gemm` with

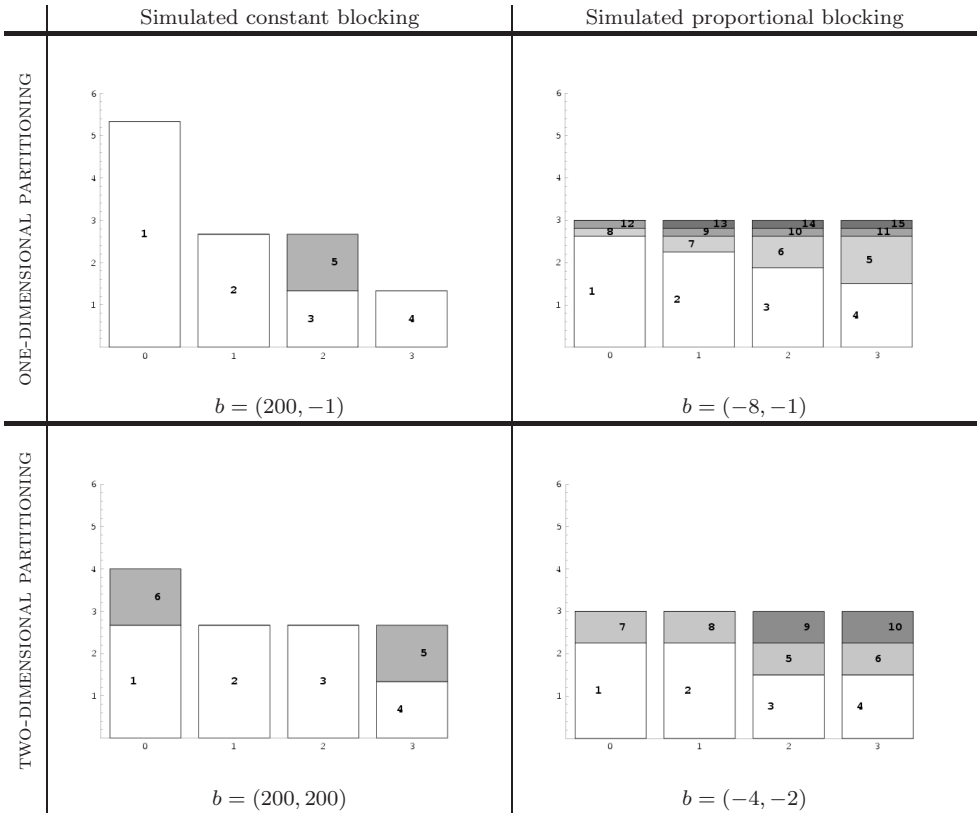

Fig. 12. Simulated scheduling of tasks to four threads for Variant 2 when $m = 600$ and $k = 200$ for various blocksize and partitioning schemes. The axes' units are similar to those of Figs. 7 and 10. *Bottom line:* Depending on the problem size, tasks created with constant blocking sometimes fail to distribute well among threads. Moving to a constant 2D partitioning helps, but remains sub-optimal. In constrast, both 1D and 2D proportional blocking create the opportunity for ideal load-balance regardless of problem size.

a call to a suitable GEMM variant to enable further partitioning the variable-sized tasks.

Blocking and Partitioning. Experiments were set up to accept constant or proportional algorithmic blocksizes in both one- and two-dimensional partitionings. Blocksizes are presented as follows: a blocksize *pair* $b = (x, y)$ indicates that a blocksize x is used in the overall SYRK algorithm while a blocksize y is used to further partition the GEMM subproblem. We denote constant blocksizes with positive numbers while proportional blocksizes are encoded as negative values. For negative x (or y), the algorithm completes $|x|$ (or $|y|$) iterations along the partitioned dimension where the actual blocksize value used is approximately equal for all iterations. If $y = -1$, then the algorithm does not partition the GEMM subproblem, which corresponds to an overall one-dimensional partitioning for the SYRK algorithm. Figure 12 conveniently illustrates the potential variations in task scheduling induced by one- and two-dimensional partitionings when combined with constant

and proportional blockings. Also, we have extended the FLAME/C programming interface to include a function, `FLA_Task_compute_blocksize`, that computes the value of `b` for each iteration based on the blocksize pair values provided as input. This routine is used to compute blocksizes in all experiments. Its use is demonstrated in Fig. 11.

Software. The Intel C compiler (version 9.0) was used to compile source code, since it is the only major compiler to support the proposed OpenMP task queue extensions. Calls to `FLA_Gemm` and `FLA_Syrk` in the loop-body were defined as wrappers to implementations of the BLAS routines `dgemm` and `dsyrk`, respectively. The code was linked to a sequential build of the GotoBLAS library (version 1.07) by Kazushige Goto [Goto 2006]. Also, threads were bound to unique processors using the SGI `dplace` utility. This method is easier and less intrusive (though less portable) than using the `sched_setaffinity` routine present in the Linux kernel.

Hardware. Performance was measured on an SGI Altix ccNUMA server consisting of seven dual-processor Itanium2 compute nodes, or *bricks*. Each brick contains approximately 2GB of local physical memory, but logically shares its memory with all other nodes via SGI's NUMAflex shared-memory architecture. Each CPU is clocked at 1.5GHz and may execute up to four double precision floating-point operations per clock cycle, yielding a peak performance of 6 GFLOPS (10^9 FLOPs/sec.) per processor. Thus, the total peak performance of the system is 84 GFLOPS. However, while 14 processors were available, we limited our tests to using 12 threads. Therefore, the maximum attainable peak of our experiments is 72 GFLOPS.

Computations. All computations were performed in double precision (64 bit) floating-point arithmetic. For the purposes of computing the rate of computation, the SYRK 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

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

6.1 Results

The resulting performance is reported in Figs. 13–17. For graphs reporting absolute performance, the maximum of the y-axis is set to 72 GFLOPS to allow the reader to visually evaluate the results relative to the theoretical peak of the experiments.

OpenMP task queues with 1D partitioning. Figure 13 shows two graphs containing results for OpenMP task queue parallelizations of Variants 1 through 4. Results are shown for only the "two loop" task partitioning option.[9] The left and right graphs illustrate using a constant blocksize of 200 (ie: $b = (200, -1)$) and a proportional blocking of -24 (ie: $b = (-2t, -1) = (-24, -1)$), respectively. Both graphs show results from one-dimensional partitionings, as indicated by the GEMM blocksize of -1 in the blocksize pairs. These results clearly show that a constant blocksize of 200 is suboptimal for most problem sizes tested. Performance is greatly

[9]We have omitted graphs for the other two options discussed in Section 3.3, as they exhibited performance signatures similar to those shown. The curious reader may find further discussion of all three task partitioning options in an earlier study of the topic presented in [Low et al. 2004].

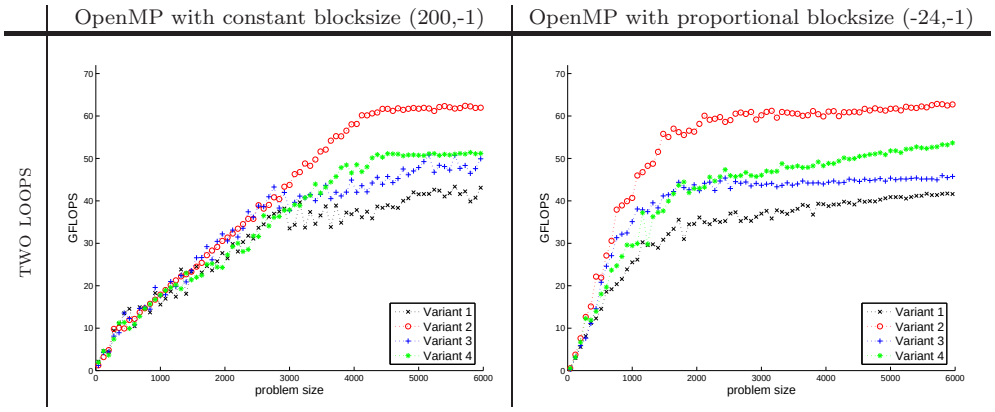

Fig. 13. Performance of OpenMP task queue parallelizations (12 threads) of SYRK Variants 1 through 4 when m equals the problem size and $k = 200$. The experiments on the left were performed with a constant blocksize of 200 while those on the right were performed with a proportional blocksize that partitioned the matrix equally through 24 iterations. Only results for the "two loop" task partitioning option are shown. *Bottom line:* Proportional blocking enables superior load balancing for small to medium-sized problems, allowing performance to ramp up quickly. Variant 2 outperforms all other variants due to a combination of enqueuing variable-sized tasks in descending order of cost and properties inherent in the `dgemm` implementation used to compute the matrix products associated with these tasks.

improved by partitioning with a proportional blocksize of -24. This observation holds regardless of how tasks are enqueued. Also of interest are the best and worst performing variants. Variants 2 and 4 both enqueue variable-sized tasks in descending (naturally sorted) order of cost. However, Variant 2 consistently outperforms Variant 4. This is likely due to the fact that the variable-sized GEMM tasks enqueued by Variant 2 update C_{21} with $A_2 A_1^T$, where C_{21} and A_2 are column-panel matrices. As of this writing, the sequential `dgemm` routine in GotoBLAS is more optimized for matrix multiplication on operands of this shape than the shape of operands in Variant 4, which updates C_{10} with $A_1 A_0^T$ where C_{10} and A_0^T are row-panel matrices.

OpenMP task queues v. FLAME workqueuing. Figure 14 shows the performance of SYRK Variant 2 using FLAME workqueuing and OpenMP task queues. The two graphs on the left show absolute performance while the graphs on the right show the speedup of FLAME workqueuing relative to OpenMP task queues. One set of graphs is given for each of the two blocksize pairs, $b = (200, -1)$ and $b = (-24, -1)$, used in Fig. 13. These results demonstrate that FLAME workqueuing does not incur significant overhead compared to the OpenMP task queue implementation in the Intel compiler. In fact, for a narrow range of small problems, FLAME workqueuing outperforms OpenMP task queues more often than not. Because the two implementations perform so similarly, we feel justified in limiting the remaining experiments to FLAME workqueuing.

Constant v. proportional blocking / 1D v. 2D partitioning. In Fig. 15, we highlight the differences among the four variants when constant and proportional blocksizes are used in one- and two-dimensional partitionings. For constant blocking, a

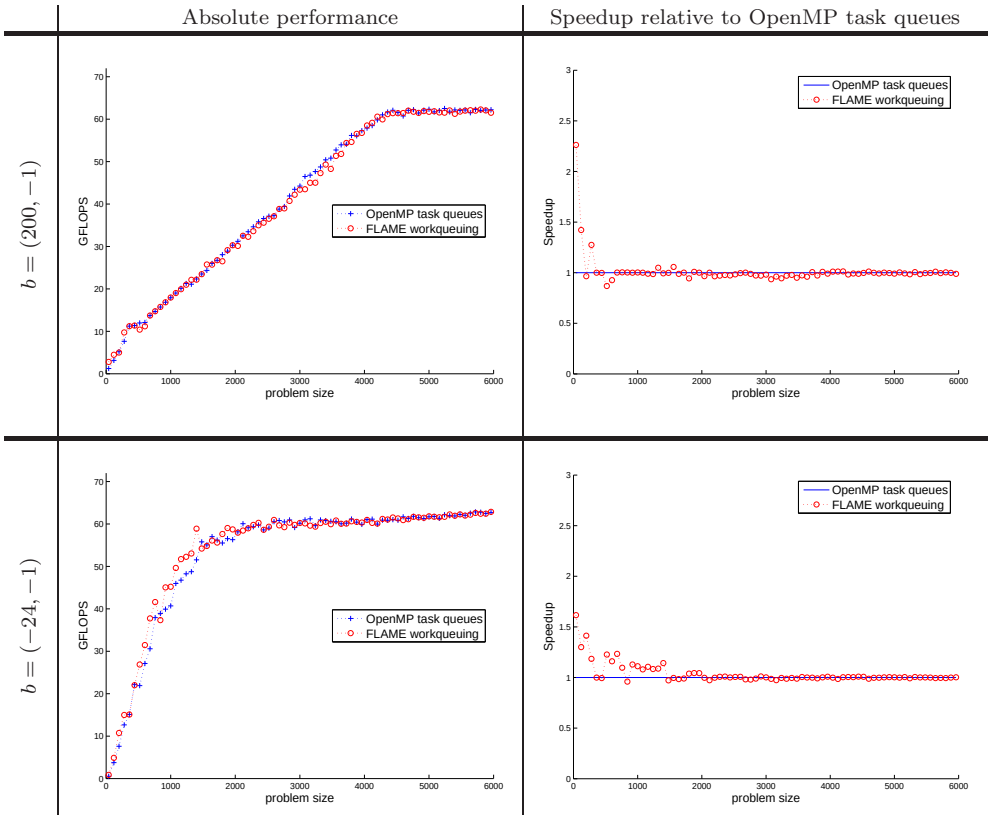

Fig. 14. Performance of OpenMP and FLAME workqueuing parallelizations (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. Absolute performance is shown in the left column for two one-dimensional partitionings while corresponding FLAME speedup relative to OpenMP is shown on the right. *Bottom line:* When compared to OpenMP task queues, FLAME workqueueing overhead is negligible (if not nonexistent).

blocksize of 200 is used. (We will see the effect of reducing this blocksize later on.) For proportional two-dimensional partitioning, we chose $b = (-t, -2) = (-12, -2)$. This blocksize pair has the special property that it partitions the SYRK algorithm into a minimal number of iterations such that enough variable-sized tasks are produced to distribute well when $t = 12$ while still creating an equal number of fixed-sized tasks for all threads. The results in Fig. 15 lead us to three interesting observations:

—Variants 1 and 4 perform nearly identically—likewise for Variants 2 and 3. The explanation is straightforward. Variants 1 and 2 share the same loop-body updates with Variants 4 and 3, respectively; the only difference within the algorithm pairs is the order in which subproblems are enqueued as tasks. FLAME workqueuing sorts the task queue automatically before threads begin dequeuing work, rendering Variants 1 and 2 equivalent and computationally indistinguishable to Variants 4 and 3, respectively. In addition, constant 2D partitionings cause variable-sized GEMM subproblems to be broken almost entirely into homo-

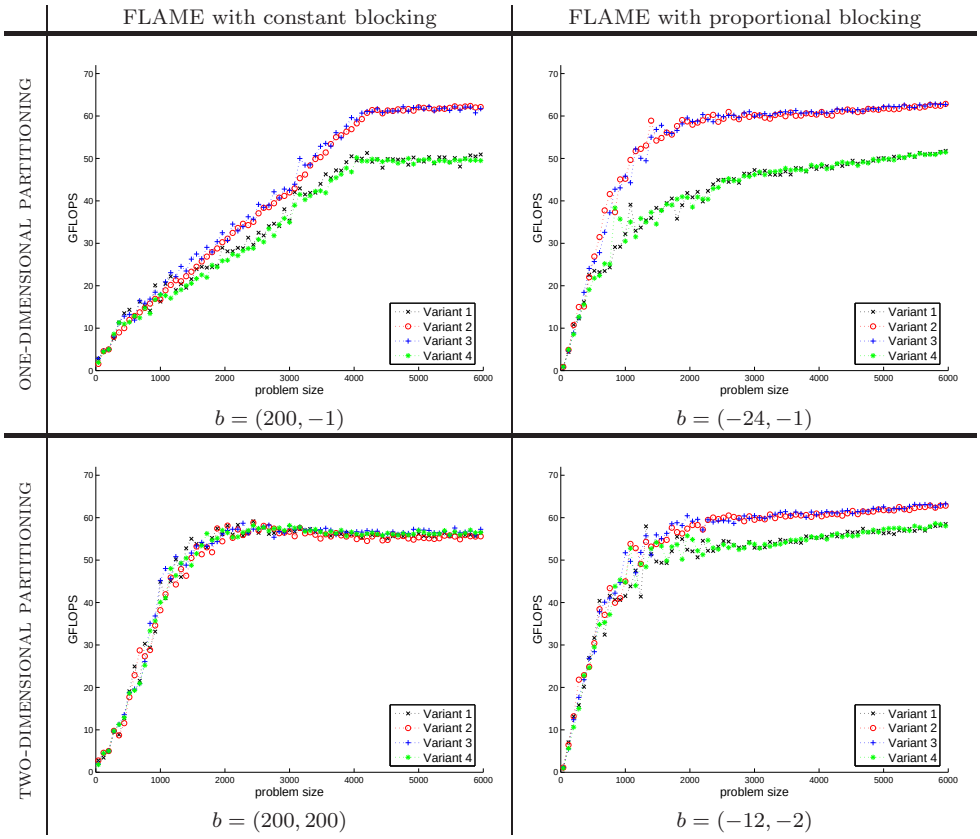

Fig. 15. Performance of FLAME workqueuing parallelizations (12 threads) of SYRK Variants 1 through 4 when m equals the problem size and $k = 200$. *Bottom line:* Sorting the workqueue renders Variants 1 and 2 computationally indistinguishable from Variants 4 and 3, respectively. Constant 2D partitioning performs better for many mid-sized problems but falls short of constant 1D performance for large problems due to limited `dgemm` efficiency on small 200×200 blocks. Proportional 2D partitioning performs similarly to that of proportional 1D for Variants 2 and 3; benefits appear mostly limited to improving lackluster Variants 1 and 4.

geneous 200×200 tasks, rendering the performance signatures of *all four* variants identical.

—The results show an overall performance advantage for 2D partitionings when a constant blocksize is used. Similarly, moving from a constant 1D partitioning to one that uses proportional blocking is sufficient to see a large jump in performance for a wide range of problems. In fact, the graphs suggest that Variants 2 and 3 need not partition both proportionally *and* in two dimensions, but rather only proportionally, in order to attain high performance for a wide range of problem sizes. This observation is predicted by the simulation of proportional 1D and 2D partitionings reported in Fig. 12.

—Lastly, it is interesting to note that a two-dimensional partitioning noticeably improves the performance of the otherwise mediocre Variants 1 and 4. This is

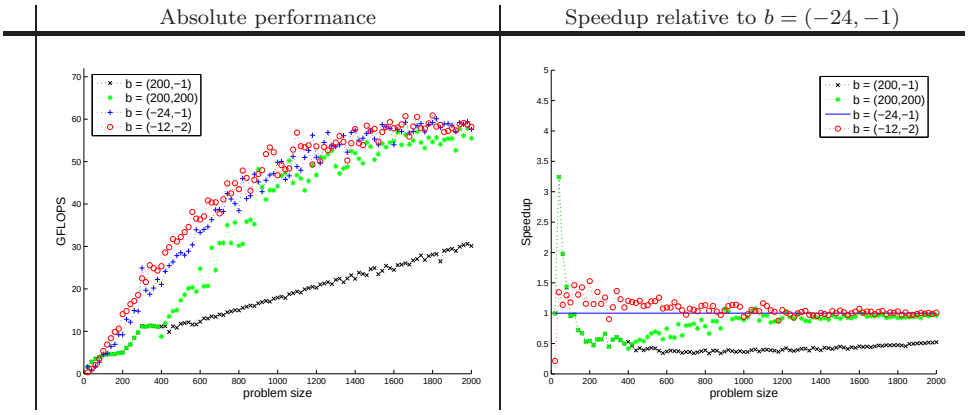

Fig. 16. Performance of FLAME workqueuing parallelization (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. The problem size range and increments have been reduced from Fig. 13 to show more detail for small problems. *Bottom line:* A proportional 2D partitioning performs noticably better for certain smaller problems and on par with a proportional 1D partitioning for larger problems.

likely a manifestation of the efficiency of the underlying sequential GotoBLAS `dgemm` in performing matrix-multiply on operands with certain shapes. As mentioned previously, the `dgemm` used tends to perform worse on the row-panel matrix multiply found in the GEMM subproblems of Variants 1 and 4.

Benefits of 2D partitioning for small problems. Figure 16 organizes the data for Variant 2 present in Fig. 15 in order to better contrast the four methods of task partitioning for small problems. In addition to showing absolute performance for each of the four task partitionings, the figure includes a graph showing speedup relative to the proportional one-dimensional partitioning given by $b = (-24, -1)$. The x-axis range and data point increments have been decreased in order to show more detail. The figure's right-hand graph reveals that a two-dimensional partitioning with proportional blocking ($b = (-12, -2)$) outperforms a similar one-dimensional partitioning ($b = (-24, -1)$) for small problems.

More on constant blocking / 2D partitioning. Finally, Fig. 17 shows the effect of moving from a one-dimensional to a two-dimensional partitioning, for both constant and proportional blocking. The following observations may be made:

—In the case of moving from $b = (200, -1)$ to $b = (200, 200)$, we see that the performance for the two-dimensional partitioning rises more sharply for smaller problems but levels off lower than that of the 1D partitioning. This most likely is due to the reduced `dgemm` efficiency that comes with casting most of the computations in terms of small 200×200 problems. By contrast, the 1D partitioning, while suffering from poor load-balance early on, maintains higher efficiency due to the variable-sized tasks creating GEMM operations where one dimension is, on average, still relatively large.

—Figure 17 also shows the effects of using a smaller constant blocksize. This is shown for blocksize pairs $b = (100, -1)$ and $b = (100, 100)$. Not surprisingly,

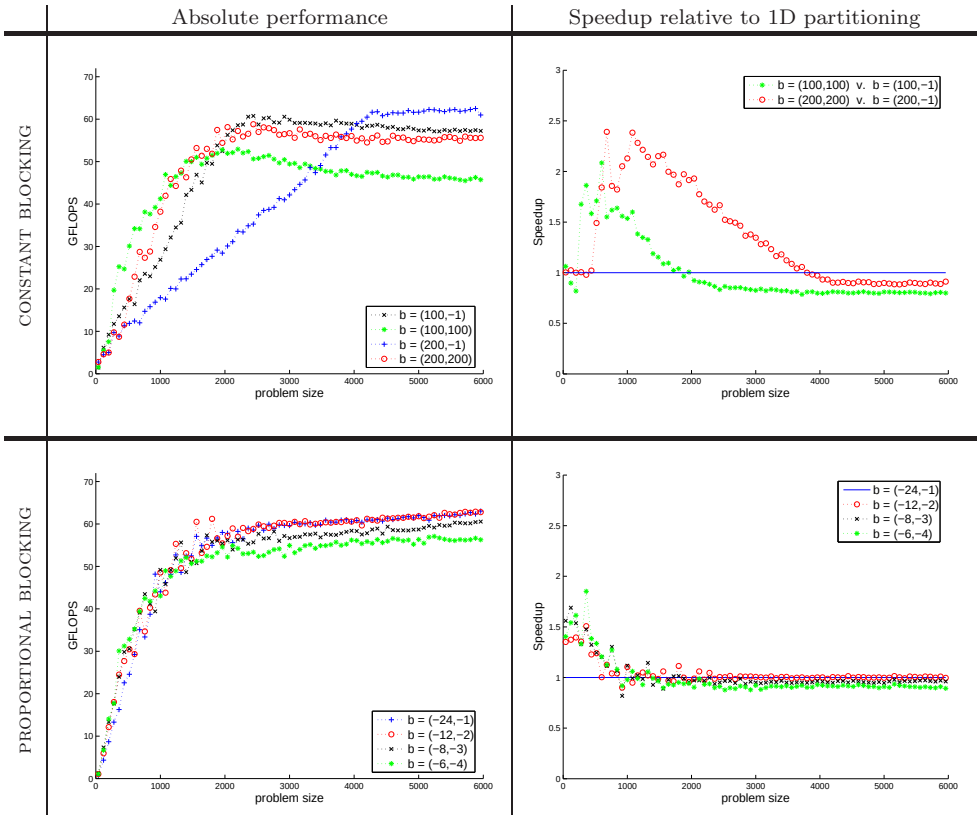

Fig. 17. Performance of FLAME workqueuing parallelizations (12 threads) of SYRK Variant 2 when m equals the problem size and $k = 200$. *Bottom line:* Two-dimensional partitioning is beneficial for smaller problems when compared to a similar 1D partitioning. A smaller constant blocksize generates parallelism more quickly, but at the expense of lower `dgemm` efficiency.

both outperform their larger counterparts for certain small problems. As the problem size increases, the blocksize pairs with smaller blocksize values create tasks more quickly, allowing more parallelism for smaller problems. However, these smaller blockings reach a lower peak performance due to reduced `dgemm` efficiency.

—For proportional blocking, we see once again that a two-dimensional partitioning is beneficial for small problems. Also included in these two graphs are data for $b = (-8, -3)$ and $b = (-6, -4)$. The performance of these partitionings roughly matches that of $b = (-24, -1)$ and $b = (-12, -2)$ for small problems but suffers slightly for large matrices. We suspect that this effect is not due to a loss of `dgemm` efficiency but rather suboptimal load-balancing. In our discussion of Fig. 14, we pointed out that the partitioning given by $b = (-12, -2)$ was more desirable than other proportional 2D partitionings. In this case, neither $b = (-8, -3)$ nor $b = (-6, -4)$ load-balances as well across 12 threads due to the fact that fewer fixed-sized SYRK subproblem tasks (8 and 6, respectively) are created than threads used in the computation. Furthermore, each of these SYRK

tasks is rather large, raising the potential for threads to become idle while some remaining portion of the SYRK computation is still in progress. Presumably, these factors conspire to prevent a favorable scheduling for most larger problems.

7. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we discussed the high-performance parallel implementation of the symmetric rank-k update operation, targeting SMP (and future multi-core) architectures. This operation is representative of how many level-3 BLAS and LAPACK-like operations are implemented with the FLAME/C API. Several contributions were reported that improve ease of implementation as well as performance.

We demonstrated how task queues, a proposed feature for OpenMP 3.0, allow code that is devoid of indexing to be elegantly and effectively parallelized. We identified and overcame two shortcomings present in the current Intel implementation of OpenMP task queues. First, we implemented a more portable domain-specific workqueuing solution for FLAME/C that uses only conventional OpenMP constructs. Second, we demonstrated the benefits of scheduling tasks in descending order of cost in the context of the SYRK operation. In addition, we demonstrated the merit in proportional blocking and found that a two-dimensional data partitioning gives way to better performance for smaller problems.

Both OpenMP and FLAME workqueuing implementations allow algorithms to be coded and parallelized at a much higher level of abstraction and, in our experience, improves almost all stages of library development. The resulting parallelized code was shown to require only minor modifications to the corresponding sequential FLAME/C implementation. Very good speedup was reported on a medium sized SMP system.

We believe this work provides the architects of OpenMP workqueuing with preliminary evidence that more control over task scheduling would benefit end-user performance. Specifically, these findings suggest that the workqueuing mechanism should allow the queue to be filled and sorted according to task cost before execution takes place. By including an optional `cost` clause in the `task` directive specification, a programmer could provide the implementation with an estimate for the cost of each task. This information would allow threads to dequeue tasks from largest to smallest, thereby potentially improving load-balance when tasks naturally vary in cost.

Future Work. After inspecting the GotoBLAS implementation of matrix-matrix multiplication [Goto and van de Geijn 2006], we have concluded that the proposed parallelization based on local GEMM operations causes threads to duplicate internal copying and packing of data. By exposing low-level interfaces to these underlying operations, it should be possible to schedule data movements so that duplication and/or memory contention can be reduced, yielding better performance yet. It should be feasible to incorporate these insights into the methodologies discussed in the present paper.

Further information

For additional information regarding the FLAME project, visit
`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 re-introduction of indices.

This research was partially sponsored by NSF grants CCF-0540926, CCF-0342369, and ACI-0305163. In addition, Dr. James Truchard (National Instruments) graciously made an unrestricted donation to our research.

Access to the 14 CPU Itanium2 (1.5 GHz) system on which the experiments were performed was provided by Dr. Gregorio Quintana-Ortí of Universidad Jaume I, Spain. Some experiments were prepared and tested on a 4 CPU Itanium2 (1.5 GHz) server which was generously donated to our research efforts by Hewlett-Packard and is administered by UT-Austin's Texas Advanced Computing Center.

We thank Kazushige Goto and Dr. Kent Milfeld (both with the Texas Advanced Computing Center) for their valuable feedback throughout our research. We also thank Dr. Andrew Chapman and Thuan Cao (both with NEC Solutions (America), Inc.) for their technical advice. We would like to acknowledge input from Dr. Enrique Quintana-Ortí on a draft of this paper.

REFERENCES

Anderson, E., Bai, Z., Demmel, J., Dongarra, J. E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. E., Ostrouchov, S., and Sorensen, D. 1992. *LAPACK Users' Guide*. SIAM, Philadelphia.

Bientinesi, P. 2006. Mechanical derivation and systematic analysis of correct linear algebra algorithms. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin. Also published as UTCS Technical Report TR-06-46.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft. 31,* 1 (March), 1–26.

Bientinesi, P., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft. 31,* 1 (March), 27–59.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1 (March), 1–17.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1 (March), 1–17.

Dow, E. 2005. Take charge of processor affinity. IBM developerWorks. `http://www.ibm.com/developerworks/linux/library/l-affinity.html`.

Garey, M. R., Graham, R. L., and Ullman, J. D. 1973. An analysis of some packing algorithms. In *Combinatorial Algorithms*, R. Rustin, Ed. New York: Algorithmics Press, 39–47.

Goto, K. 2006. `http://www.cs.utexas.edu/users/kgoto`.

Goto, K. and van de Geijn, R. A. 2006. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*. submitted.

Johnson, D. S. 1973. Approximation algorithms for combinatorial problems. In *Fifth Annual ACM Symposium on Theory of Computing*. New York: Assoc. Comput. Mach., 38–49.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5,* 3 (Sept.), 308–323.

Low, T. M., Milfeld, K. F., van de Geijn, R. A., and Van Zee, F. G. 2004. Parallelizing flame code with openmp task queues. Department of Computer Sciences Technical Report TR-04-05, The University of Texas at Austin. December.

Low, T. M., van de Geijn, R. A., and Van Zee, F. G. 2005. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *PPoPP '05: Proceedings of the*

tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM Press, New York, NY, USA, 153–163.

OpenMP Architecture Review Board. 2006. `http://www.openmp.org/`.

Quintana-Ortí, E. S. and van de Geijn, R. A. 2003. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software 29,* 2 (June), 218–243.

Shah, S., Haab, G., Peterson, P., and Throop, J. 1999. Flexible control structures for parallelism in OpenMP. In *EWOMP*.

Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., and Peterson, P. 2002. Compiler support of the workqueuing execution model for Intel SMP architectures. In *EWOMP*.

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

Weisstein, E. W. 2006. Bin-Packing Problem. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/Bin-PackingProblem.html`.

Received Month Year; revised Month Year; accepted Month Year