Extracting SMP Parallelism for Dense Linear Algebra Algorithms from High-Level Specifications

Tze Meng Low
Robert A. van de Geijn
Field G. Van Zee
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{ltm,rvdg,field}@cs.utexas.edu

ABSTRACT

We show how to exploit high-level information, available as part of the derivation of provably correct algorithms, so that SMP parallelism can be systematically identified. Recent research has shown that loop-based dense linear algebra algorithms can be systematically derived from the mathematical specification of the operation. Fundamental to the methodology is the determination of loop-invariants (in the sense of Dijkstra and Hoare) from which correct loops can be systematically derived. We show how the high-level specification of the operation together with these loop-invariants can be exploited to detect the independence of loop iterations. This in turn then allows a Workqueuing Model to be used to implement and parallelize the algorithms using a feature proposed for OpenMP 3.0, task queues. Although performance is not the main feature of this paper, performance is reported on a 4 CPU Itanium2 server for a concrete example, the symmetric rank-k update operation.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.2.11 [**Software Architectures**]: Domain-specific architectures; D.3.4 [**Processors**]: Code generation

General Terms

Algorithms, Design, Theory

Keywords

Formal derivation, linear algebra, code generation, SMP Parallelism

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

We describe a systematic way of generating provably correct dense linear algebra implementations for SMP systems from the mathematical (high level) specification of the operation. Our method of generating these implementations for SMP systems involves prescribed steps: (1) We examine a high-level description of the operation to determine if an algorithm with independent iterations can exist. (2) If it is determined that there can be an algorithm with independent iterations, we find a loop-invariant where the corresponding algorithm has independent iterations from the mathematical specification of the operation. (3) We derive an algorithm from the loop-invariant via the Formal Linear Algebra Methodology Environment (FLAME) approach [5]. (4) We translate the algorithm to code using the FLAME API for the C programming language [2]. (5) We parallelize the sequential code by inserting task queue directives, a proposed construct for OpenMP 3.0 [11, 10, 7].

The primary contribution of the paper is in the theory that allows us to perform Steps 1 and 2. In the interest of making this paper self-contained, brief introductions to FLAME and the the use of task queues under the FLAME framework have been included. Details regarding Steps 3 through 5 are described in the literature.

Our approach towards extracting SMP parallelism is very different from that taken by current compilers, which analyze code in order to determine how iterations can be parallelized. This typically requires indices into arrays in order to perform dependence analysis on the code. After determining whether a code can be parallelized, the compiler will then perform program transformations in order to transform the code into one that has better parallelism [6, 12]. Instead, we analyze the loop-invariant, which the FLAME approach to deriving algorithms makes available to us.

We illustrate our implementation generation process via a concrete example: the computation of the symmetric rank-k update (`SYRK`). This operation is supported by the Basic Linear Algebra Subprograms (BLAS) [3] and is employed by algorithms for computing the Cholesky factorization and reduction to tridiagonal form. More importantly, the `SYRK` operation is representative of many algorithms that are part of the BLAS and LAPACK and that can be systematically derived via the FLAME approach.

This paper is organized as follows: In the following section, we review FLAME as a means of deriving algorithms from a high level specification of the problem. Next, we describe how the use of task queues, a feature proposed for inclusion into OpenMP 3.0, allows us to parallelize algorithms without having to deal with intricate indexing. In Section 4, we discuss how to determine *a priori* if algorithms can be parallelized using task queues. We show performance resulting from the use of task queues in Section 5. Concluding remarks are given in the final section.

2. FLAME

FLAME is a methodology for deriving families of provably correct linear algebra algorithms[5, 1]. FLAME allows users to reason about linear algebra algorithms at a level that avoids the use of explicit indexing. This is achieved by partitioning matrices into disjoint submatrices and casting the required updates as updates of the various submatrices. We illustrate the process via a concrete example.

2.1 Deriving an algorithm

Consider the `SYRK` operation, $C := AA^T + C$, where C is symmetric matrix stored in the lower triangular part of matrix C, A is a general matrix, and we overwrite the original value of C with the computed value. To distinguish between the input and computed values of C, we shall use C and $\hat{C}$ to denote the output and input values, respectively.

To derive an algorithm for $C := AA^T + \hat{C}$, we first partition C and A conformally in the following manner:

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

Here C_{TL} and C_{BR} are symmetric submatrices. The thick lines are meant to indicate how the computation proceeds through the data. The $*$ indicates the symmetric part that is not stored. Upon completion, matrix C should contain, in terms of the submatrices,

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

This implies

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv A_B A_B^T + \hat{C}_{BR} \end{array}\right),$$

which we call the *partitioned matrix expression (PME)*.

The PME gives all computation that must be performed to compute C in terms of the submatrices. To derive a loop, we need a loop-invariant. A loop-invariant should describe a state that is a partial result towards the final result. The FLAME approach derives a loop-invariant by removing some of the computation from the PME. Some of these states will not yield valid loops. Those that do we call *feasible* loop-invariants. For our example, four feasible loop invariants are given in Fig. 1.

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv \hat{C}_{BL} & C_{BR} \equiv \hat{C}_{BR} \end{array}\right)$$

Loop-invariant 1

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv \hat{C}_{BR} \end{array}\right)$$

Loop-invariant 2

$$\left(\begin{array}{c|c} C_{TL} \equiv \hat{C}_{TL} & * \\ \hline C_{BL} \equiv \hat{C}_{BL} & C_{BR} \equiv A_B A_B^T + \hat{C}_{BR} \end{array}\right)$$

Loop-invariant 3

$$\left(\begin{array}{c|c} C_{TL} \equiv \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv A_B A_B^T + \hat{C}_{BR} \end{array}\right)$$

Loop-invariant 4

Figure 1: Four feasible loop-invariants for algorithms that compute $C := AA^T + \hat{C}$.

The process of deriving an algorithm from a specific loop invariant via the FLAME approach is a systematic process consisting of eight steps. We refer the reader to [1] for details. Using the following loop-invariant,

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv \hat{C}_{BR} \end{array}\right)$$

the derivation process derives an algorithm to compute $C := AA^T + \hat{C}$ given in Figure 2. That figure shows a *blocked* algorithm, which, for performance reasons, casts most computation in terms of matrix-matrix products.

We believe the algorithm to be self-explanatory. In short, the thick lines are used to show progress through the matrices. Thin lines show submatrices that will be updated and/or used. The function $m(X)$ returns the row dimension of matrix X. The reader may wish to consult [1].

2.2 Translating the algorithm to code

Having derived an algorithm, a translation from the algorithm into code is required. The FLAME APIs [2] allow users to represent FLAME algorithms in code. The correctness of the derived algorithm is captured in its implementation by making the code look similar to the algorithm that results from the derivation. This is achieved by hiding details such as indices, storage

Algorithm: $[D, E, F, \ldots] := \text{op}(A, B, C, D, \ldots)$

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

where A_T has 0 rows, C_{TL} is 0×0

while $m(A_T) \neq m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} C_{TL} & * \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & * & * \\ \hline C_{10} & C_{11} & * \\ C_{20} & C_{21} & C_{22} \end{array}\right)$$

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

$C_{11} := A_1 A_1^T + C_{11}$
$C_{21} := A_2 A_1^T + C_{21}$

Continue with

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} C_{TL} & * \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & * & * \\ C_{10} & C_{11} & * \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$$

end while

Figure 2: Algorithm for computing $C := AA^T + \hat{C}$ that maintains loop-invariant 2.

methods, and matrix dimensions from the programmer through the use of object based programming, similar to that used in the Message Passing Interface (MPI) [13]. Currently, the FLAME APIs include APIs for Fortran, C, and Matlab[8]. PLAPACK [14], an API for coding linear algebra algorithms that targets distributed memory architectures, has also been extended to allow code targeting those architectures to resemble FLAME algorithms.

The FLAME/C API is a FLAME API for C. The implementation of the derived algorithm in Figure 2 using FLAME/C is shown in Figure 3. Notice that by using APIs that make the code look like the description of the algorithm, the implementation does not include intricate indexing.

A consequence of removing explicit loop counters is that traditional compiler approaches to code analysis and automatic parallelization cannot be applied to the resulting FLAME code.

3. OPENMP AND FLAME/C

Since the FLAME APIs do not encourage the use of explicit loop counters, directives that are part of OpenMP 2.0 do not allow us to easily generate code for SMP systems. Fortunately, we can utilize the Workqueueing Model introduced in [11, 10] to extract SMP parallelism [7]. A construct, the task queue, that supports this model has been proposed for inclusion into OpenMP 3.0.

3.1 The Workqueueing Model

Conceptually, a `taskq` pragma creates an empty task queue. Within the `taskq` block, tasks that can be enqueued onto the task queue are identified by the `task` pragma. A single thread executes the code within the `taskq` block sequentially. Each time a `task` pragma is encountered, a task is created and enqueued onto the task queue. Other threads would then dequeue tasks from the task queue and execute the tasks in parallel.

3.2 Parallelizing FLAME/C with OpenMP task queues

In Figure 4, we present how task queue constructs can be added to the FLAME/C implementation. We highlight the following lines in the code:

- Line 16: The task queue is initialized.
- Line 31: This line indicates the start of a task that is to be enqueued. Since the values of C_{11}, C_{21}, A_1 and A_2 change with every iteration, we need to preserve their values with the use of the `captureprivate` directive.
- Line 35: This indicates the end of the task.
- Line 37: This indicates the end of the task queue. An implicit synchronization of the threads is performed to ensure that all tasks have been completed.

Notice that by inserting two lines of OpenMP directives, and a few curly-brackets, a sequential C code is turned into code for SMP systems. However, in order to parallelize code using task queues, the programmer has the responsibility of ensuring that the tasks are independent. Here we use the term *update* for the expressions between the lines "`/* ---- ··· ---- */`". Since a task in Figure 4 is essentially all the updates in the loop, it follows that to parallelize FLAME algorithms using task queues, we need to ensure that (updates in) the iterations of the loops are independent. In other words, we need to ensure that the result of updates in previous iterations are not needed by computations in future iterations and data required by computations in future iterations are not overwritten by updates of previous iterations.

As mentioned previously, compilers use indices to analyze the code to determine if the iterations of the loop being parallelized are independent. However, since APIs are used to capture the correctness of a FLAME algorithm, the lack of indices require us to analyze the FLAME/C code via other means.

4. DETERMINING INDEPENDENCE OF ITERATIONS A PRIORI

The purpose of using APIs to implement a FLAME algorithm is to ensure that the code is similar in appearance to the algorithm. Therefore, an analysis of

```
int Syrk_blk( FLA_Obj A, FLA_Obj C, int nb_alg )
{
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;

  FLA_Obj CTL,   CTR,      C00, C01, C02,
          CBL,   CBR,      C10, C11, C12,
                           C20, C21, C22;

  int b;

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

  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );

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

    b = min( FLA_Obj_length( AB ), nb_alg );

    FLA_Repart_2x1_to_3x1( AT,                &A0,
                        /* ** */            /* ** */
                                              &A1,
                           AB,                &A2,        b, FLA_BOTTOM );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00, /**/ &C01, &C02,
                        /* ************* */   /* ******************** */
                                                &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,       &C20, /**/ &C21, &C22,
                           b, b, FLA_BR );

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

    FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
             ONE, A2, A1, ONE, C21);

    FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
             ONE, A1, ONE, C11);

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

    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */           /* ** */
                              &AB,                A2,     FLA_TOP );

    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,       C00, C01, /**/ C02,
                                                     C10, C11, /**/ C12,
                            /* ************** */  /* ****************** */
                              &CBL, /**/ &CBR,       C20, C21, /**/ C22,
                              FLA_TL );

  }

  return FLA_SUCCESS;
}
```

Figure 3: FLAME/C implementation of the algorithm in Figure 2.

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

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

 #pragma intel omp parallel taskq
  {
    while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
      b = min( FLA_Obj_length( AB ), nb_alg );

      FLA_Repart_2x1_to_3x1( AT,                &A0,
                             /* ** */            /* ** */
                             &A1,
                             AB,                &A2,        b, FLA_BOTTOM );
      FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00, /**/ &C01, &C02,
                             /* ************* */   /* ******************** */
                             &C10, /**/ &C11, &C12,
                             CBL, /**/ CBR,       &C20, /**/ &C21, &C22,
                             b, b, FLA_BR );
      /*------------------------------------------------------------*/
     #pragma intel omp task captureprivate(C11, A1, A2, C21)
      {
        FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A2, A1, ONE, C21);
        FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11);
      } /* end of task */
      /*------------------------------------------------------------*/
      FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                    A1,
                                /* ** */           /* ** */
                                &AB,                A2,     FLA_TOP );
      FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,       C00, C01, /**/ C02,
                                                       C10, C11, /**/ C12,
                                /* ************** */  /* ****************** */
                                &CBL, /**/ &CBR,       C20, C21, /**/ C22,
                                FLA_TL );
    }
  }  /* end of task queue */
  return FLA_SUCCESS;
}
```

Figure 4: OpenMP implementation of the algorithm in Figure 2.

the algorithm is equivalent to an analysis of the code. In addition, since the algorithm is derived from the loop-invariant, the loop-invariant is the essence of the algorithm. The PME describes the operation to be computed and the loop-invariant describes the algorithm being used. It follows that by analyzing the PME and the loop-invariant, we analyze the code but at a higher level of abstraction and with more information than traditional code presents to a traditional compiler.

In this section, we describe how an analysis of the PME and the loop-invariant allows the *a priori* determination of the independence of iterations of a FLAME algorithm. We assume throughout that the reader understands the meaning of *true dependence, anti-dependence* and *output dependence* as described in the compiler literature.

4.1 Relating the PME, loop-invariant, and dependences

Recall that the PME is a specification of the operations in terms of partitioned operands. It tells us how data from different parts of the input matrices are used to compute the different submatrices of the output matrix. By studying the flow of data in the PME, we are essentially performing data dependence analysis on the PME. In addition, because the thick lines both indicate movement through the matrix and partition the matrix into submatrices, a dependence between submatrices could potentially be a dependence across iterations. In order to determine whether a dependence between submatrices is a dependence between loops, we have to study the loop invariant.

A loop-invariant is a description of a partial result towards the computation described by the PME. In particular, it states which part of the matrix has been computed and which part will be computed later. A loop-invariant, by definition, must be true at the start of every iteration. Therefore, any submatrix of the output matrix that has a computed value must have been computed in previous iterations and any submatrix that still contains the original value must be updated in future iterations. Therefore, if a dependence between two submatrices is such that one submatrix has a computed value and the other will be updated in future iterations, then we know that that dependence occurs between iterations. Furthermore, if the loop-invariant is such that there exists one submatrix with an partial result, we know that there exists a dependence between iterations. This is because the same matrix was partially updated in previous iterations but will have to be updated again in future iterations.

Consider a PME for the operation, $C = AA^T + \hat{C}$, given below:

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv A_B A_B^T + \hat{C}_{BR} \end{array}\right)$$

Notice that in order to compute C_{TL}, only data from $\hat{C}_{TL}$ and A_T are needed. Therefore, we can conclude that C_{TL} does not depend on any other submatrices of C being computed first. A similar analysis of the other submatrices of C shows that the values in $\hat{C}_{TL}$ and the result of C_{TL} are not needed in the computation of the other submatrices. This indicates that the submatrices of C can be computed independently and in any order.

For the SYRK example: (1) The thick lines are separators that divides the submatrices of C into two disjoint sets, those that were computed in previous iterations and those to be computed in future iterations. (2) The loop-invariant describes which submatrices of C belong to which of the two sets. (3) Since the submatrices can be computed independently, we know that no result of previous iterations are needed to compute the result of future iterations. For this example we can therefore conclude that the iterations must be independent.

4.2 Conditions necessary for independent iterations

Now, let us attempt to generalize the above analysis, for a typical operation $C := \mathrm{op}(A, C)$ where A represents all matrices whose values are read but not written to. Due to the lack of space, we will only show general results for the case where there is one output matrix and the matrix is partitioned as follows:

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right).$$

How matrix A is partitioned is not important for this analysis since it will not change in value. Since the output matrix is partitioned into four quadrants, it follows that the PME and loop-invariant each must have four quadrants as well. Let the PME and loop-invariant for the operation be

$$\left(\begin{array}{c|c} C_{TL} \equiv \mathsf{PME}_{TL} & C_{TR} \equiv \mathsf{PME}_{TR} \\ \hline C_{BL} \equiv \mathsf{PME}_{BL} & C_{BR} \equiv \mathsf{PME}_{BR} \end{array}\right) \text{ and}$$

$$\left(\begin{array}{c|c} C_{TL} \equiv \mathsf{INV}_{TL} & C_{TR} \equiv \mathsf{INV}_{TR} \\ \hline C_{BL} \equiv \mathsf{INV}_{BL} & C_{BR} \equiv \mathsf{INV}_{BR} \end{array}\right)$$

where PME_X and INV_Y are mathematical functions defined on matrices and $X, Y \in \{TL, TR, BL, BR\}$, respectively. PME_X has the general form

$$\mathsf{PME}_X(A, \hat{C}_{TL}, \ldots, \hat{C}_{BR}).$$

In addition, let REM_X be a mathematical function such that

$$\mathsf{PME}_X \equiv \mathsf{REM}_X(\mathsf{INV}_X)$$

, in other words, REM_X represents the remaining computation to be performed on quadrant X.

We begin our generalization with some definitions.

Definition 1. *The state of a quadrant X, denoted $\sigma(X)$, can be catagorized into*

1. Fully Updated. *$\sigma(X)$ is fully updated if and only if $\mathsf{PME}_X \equiv \mathsf{INV}_X$ and REM_X is the identity function.*
2. Partially Updated. *$\sigma(X)$ is partially updated if $\mathsf{PME}_X \neq \mathsf{INV}_X$ and $\mathsf{INV}_X \neq \hat{C}_X$.*
3. Not Updated. *$\sigma(X)$ is not updated if $\mathsf{INV}_X \equiv \hat{C}_X$ and REM_X is not the identity function.*

Since every algorithm derived using the FLAME approach has a corresponding loop-invariant, we need to know what properties of the loop-invariant are necessary and sufficient in order to conclude that the derived algorithm has independent iterations. We start with two lemmas, stating properties a loop-invariant must possess in order to derive an algorithm with independent iterations.

Lemma 1. *If an algorithm derived using the FLAME approach has independent iterations, each quadrant in the loop-invariant must either be fully updated or not updated.*

Proof: A proof by contradiction is used. Assume that quadrant X in the loop invariant is partially updated. By definition, INV_X was computed in previous iterations and REM_X must be performed on INV_X in future iterations in order compute PME_X. Therefore, the iterations cannot be independent. Q.E.D. ∎

Lemma 2. *If an algorithm derived using the FLAME approach has independent iterations and there exists a dependence between quadrants X and Y in the PME, then quadrants X and Y in the loop-invariant must either both be fully updated or both be not updated.*

Proof: Assume that an algorithm has independent iterations and there exists a dependence between quadrants X and Y in the PME. In addition, assume that $\sigma(X) \neq \sigma(Y)$. Without loss of generality assume that the dependence between X and Y is such that X has to be computed before Y. Using Lemma 1, we can conclude that $\sigma(X)$ is fully updated while $\sigma(Y)$ is not updated. This implies that quadrant X was computed in previous iterations while quadrant Y is to be computed in future iterations. Since there is a dependence between X and Y, it follows that there must be a dependence between previous and future iterations. Therefore, the algorithm cannot have independent iterations and thus we have a contradiction. Q.E.D. ∎

Theorem 1. *An algorithm derived from a feasible loop-invariant will have independent iterations if and only if the loop-invariant possesses the following properties*

1. *Every quadrant in the loop-invariant must either be fully updated or not updated.*

2. *If there exists a dependence between quadrants X and Y in the PME, then quadrants X and Y in the loop-invariant must either both be fully updated or both be not updated.*

Proof: Lemmas 1 and 2 show that both properties are necessary. We now show that these properties are sufficient to show the independence of iterations. Assume that an algorithm and its associated loop-invariant have the two properties but there exists a dependence between iterations. We employ proof by contradition to show that true dependence, anti-dependence and output dependence cannot exist.

True Dependence: Assume that the dependence between iterations is a true dependence. Let X and Y be dependent such that the result of X is required to compute the result of Y. Since the true dependence occurs across iterations, X must be computed in previous iterations while Y must be computed in future iterations. This implies that $\sigma(X)$ is fully updated whereas $\sigma(Y)$ is not updated. This contradicts the property that if two submatrices are dependent then they have to be in the same state. Therefore, there cannot be a true dependence between iterations.

Anti-Dependence: Assume that the dependence between iterations is an anti-dependence. In a typical iteration, submatrix X is updated with the original value in another submatrix Y. Since the anti-dependence is across iterations, it must follow that submatrix Y is updated in some iteration in the future. Therefore, we can conclude that at the end of the current iteration, $\sigma(X)$ is fully updated whereas $\sigma(Y)$ is not updated. This contradicts the property that if two submatrices are dependent then they have to be in the same state. Therefore, there cannot be an anti-dependence between iterations.

Output Dependence: Assume that there exists an output dependence between iterations. By definition, a submatrix, X, is written with a certain value in one iteration and another value is stored in the X in future iterations. This correspond to the $\sigma(X)$ being partially updated. Since the loop-invariant can only have quadrants that are either fully updated or not updated, it follows that there are not output dependence between iterations.

Since there cannot be true dependence, anti-dependence and output dependence across iterations, then it follows that the iterations must be independent. Therefore, we have proved that the two properties are sufficient to show that the derived algorithm has independent iterations. Q.E.D. ∎

4.3 (Non-)existence of algorithms with independent iterations

Given that a data dependence analysis of the PME is the first step in determining dependences between submatrices of the PME, we show how performing data dependence analysis on the PME allows us to *a priori* determine the (non-)existance of algorithms.

Again, we start with a definition.

Definition 2. *Two sets of submatrices, S_0 and S_1, are said to be independent if*

- $S_0 \cap S_1 = \emptyset$
- *For any $X \in S_0$ and $Y \in S_1$, there exists no dependence between X and Y.*

THEOREM 2. *If the submatrices of a PME cannot be divided into two or more independent sets of submatrices, then every algorithm whose loop-invariant is obtained by removing sub-expressions from the PME will have dependent iterations.*

Proof: We again prove this with a proof by contradiction. Assume that the submatrices of the PME cannot be divided into two or more independent sets of submatrices. In addition, assume that there exists a loop-invariant that is obtained by removing sub-expressions from the PME but the corresponding algorithm has independent iterations. We want to show that such an algorithm cannot exist.

Since the algorithm has independent iterations, Theorem 1 allows us to conclude that the loop-invariant has the property that submatrices that have a dependence between them must be of the same state. Since the PME cannot be divided into two independent sets, then there exists dependences between the four quadrants. Therefore, all four quadrants of the loop-invariant must be in the same state. This implies that either all four quadrants of the loop invariant are fully updated or not updated. In both cases, the loop-invariant is infeasible and thus the algorithm with independent iterations cannot exist. Q.E.D. ∎

An example of an operation whose PME cannot be separated into two independent sets of submatrices is the solution to the triangular Sylvester equation, a commonly encountered problem in Control Theory [9]. Consider the equation $AX + XB == C$ where A, B and C are input matrices such that A and B are triangular matrices and X is the output matrix. If A and B are both lower triangular matrix and we partition all matrices as follow:

$$A \left(\begin{array}{c|c} A_{TL} & 0 \\ \hline A_{BL} & A_{BR} \end{array} \right), B \left(\begin{array}{c|c} B_{TL} & 0 \\ \hline B_{BL} & B_{BR} \end{array} \right),$$
$$C \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right), \text{ and } X \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array} \right),$$

we obtain the PME in Figure 5.

Notice that X_{BL} is required to compute X_{TL} and X_{BR} while X_{TL} and X_{BR} are needed to compute X_{TR}. Therefore, it is not possible to split the submatrices of X into two independent sets. Thus, Theorem 2 allows us to conclude that no iterative algorithm that solves the triangular Sylvester equation can be parallelized with the mechanism as described in the previous section.

4.4 Another example

Let us now discuss a slightly more complex example than the SYRK operation. Let L and U be a lower triangular matrix and an upper triangular matrix respectively. Consider the operation $U := UL$. Letting $\hat{U}$ denote the original contents of U and partitioning

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BL} \end{array} \right), \text{ and } U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array} \right),$$

the PME for this operation is given by

$$\left(\begin{array}{c|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} + \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR}L_{BR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR}L_{BR} \end{array} \right).$$

Notice that $\hat{U}_{TR}$ is required to compute the top two quadrants while $\hat{U}_{BR}$ is required to compute the bottom two quadrants. This means that there are anti-dependences between quadrants on the same row. However, the rows are independent. By separating the quadrants of the PME into $\{TL, TR\}$ and $\{BL, BR\}$, we have separated the quadrants into two independent sets. More importantly, there are no other ways of separating the PME into two independent sets. Therefore, we know that there could be algorithms with independent iterations that compute $U := UL$. A list of feasible loop invariants to compute $U := UL$ is shown in Figure 6. As theory predicts, only algorithms that compute U row-wise have independent iterations.

5. PERFORMANCE

We re-emphasize that this is *not* a paper about performance. As such, we do not compare the performance obtained against other implementations. Nonetheless, we report performance to show that good performance can be achieved easily. Performance for a 4 CPU Intel Itanium2(R) (1.5GHz) server with a peak performance of 6 GFLOPS per processor, for a total peak of 24 GFLOPS, is reported. The Intel C compiler was used, since it supports the task queue construct that are proposed for OpenMP 3.0. Double precision (64 bits) arithmetic was used in all computation. The implementations were linked to the BLAS libraries by Kazushige Goto [4]. A block size of 104, known to yield good performance from the general matrix-matrix multiplication routine (`DGEMM`) in Goto's BLAS, was used in our experiments. For $C \in \mathbb{R}^{m\times m}$ and $A \in \mathbb{R}^{m\times k}$ the operation count for computing $C := AA^T + C$ (lower triangular part only) is taken to equal m^2k floating point operations (flops). We report the rate of computation as $\text{GLOPS} = m^2k/(\text{time in sec.}) * 10^{-9}$.

Blocked algorithms for all four loop-invariants given in Fig. 1 were implemented using FLAME/C. The calls `FLA_Gemm` and `FLA_Syrk` were implemented as wrappers to the BLAS routines DGEMM and DSYRK, respectively.

The performance results are shown in Figure 7. The top of the graph correspond to the theoretical peak performance of the machine. Notice that for all four algorithms, we obtained an approximately linear speed-up as we increase the number of threads to that of the number of processors on the system. The performance behavior of the different implementations is quite different. It is interesting to note that the algorithms corresponding to loop-invariants 1 and 4 are the same loop, but executed in reverse order. In [7], the authors show that the variations in behavior across variants can be explained by the scheduling properties inherent in OpenMP task queues. The algorithms corresponding to loop-invariants 2 and 3 are similarly identical but for

$$\left(\begin{array}{c|c} X_{TL} \equiv \Omega(A_{TL}, B_{TL}, C_{TL} - A_{TR}X_{BL}) & X_{TR} \equiv \Omega(A_{TL}, B_{BR}, C_{TR} - A_{TR}X_{BR} - X_{TL}B_{TR}) \\ \hline X_{BL} \equiv \Omega(A_{BR}, B_{TL}, C_{BL}) & X_{BR} \equiv \Omega(A_{BR}, B_{BR}, C_{BR} - X_{BL}B_{TR}) \end{array}\right)$$

where $X \equiv \Omega(A, B, C)$ represents the solution to $AX + XB == C$.

Figure 5: PME for the triangular Sylvester equation where A and B are lower triangular matrices.

Loop Invariant	Independent Iterations?	Type of Dependence
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BL} \equiv 0 & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	No	Output
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} + \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BR} \equiv 0 & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	No	Anti
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} + \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR}L_{BR} \\ \hline U_{BL} \equiv \hat{U}_{BL} \equiv 0 & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	Yes	-N.A.-
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	No	Output & Anti
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} + \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	No	Anti
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL}L_{TL} + \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR}L_{BR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR} \end{array}\right)$	No	Anti
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR}L_{BR} \end{array}\right)$	Yes	-N.A.-
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR} \\ \hline U_{BL} \equiv \hat{U}_{BR}L_{BL} & U_{BR} \equiv \hat{U}_{BR}L_{BR} \end{array}\right)$	No	Output & Anti
$\left(\begin{array}{c\|c} U_{TL} \equiv \hat{U}_{TR}L_{BL} & U_{TR} \equiv \hat{U}_{TR}L_{BR} \\ \hline U_{BL} \equiv \hat{U}_{BL}L_{BL} & U_{BR} \equiv \hat{U}_{BR}L_{BR} \end{array}\right)$	No	Output

Figure 6: Feasible loop-invariants for the operation $U := UL$

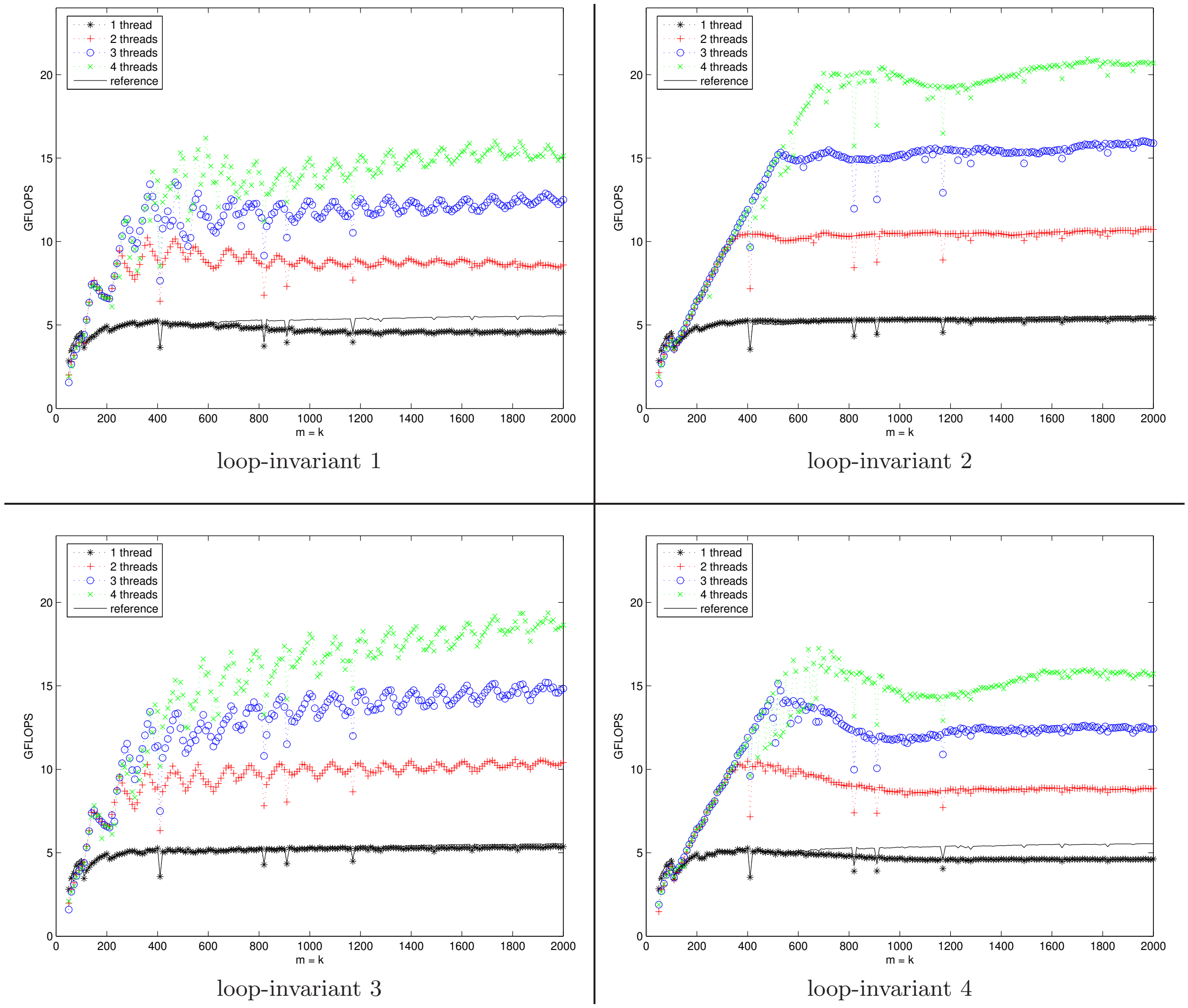

Figure 7: Performance Graphs for algorithms derived from different loop invariants.

the order of the loop. The lower performance attained by algorithms corresponding to loop-invariants 3 and 4 as compared to those with loop-invariants 2 and 1 is either due to the implementation of DGEMM or due to false sharing of data. For additional details and performance results, we refer the reader to [7].

6. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we described how implementations of dense linear algebra algorithms for SMP systems can be systematically obtained. The use of the FLAME approach allows one to derive provably correct dense linear algebra algorithms from a high level of specification. Task queues enable code to be parallelized without the use of an explicit loop index. We discussed conditions for which the PME and loop-invariants must have in order to derive algorithms with independent iterations. By applying dependence analysis on the PME, we can determine *a priori* if the operation can be parallelized.

The FLAME approach to deriving and implementing linear algebra operations have been shown to apply to a large number of operations in linear algebra [5, 1]. We believe that both the implementation/parallelizing process and the analysis of both the PME and loop-invariants can be similarly applied to algorithms derived and implemented using the FLAME approach.

More can be done through the analysis of the PME and the loop-invariants. Consider the two loop-invariants

$$\left(\begin{array}{c|c} C_{TL} \equiv A_T A_T^T + \hat{C}_{TL} & * \\ \hline C_{BL} \equiv A_B A_T^T + \hat{C}_{BL} & C_{BR} \equiv \hat{C}_{BR} \end{array}\right) \text{ and } \left(\begin{array}{c|c} C_{TL} \equiv \hat{C}_{TL} & * \\ \hline C_{BL} \equiv \hat{C}_{BL} & C_{BR} \equiv A_B A_B^T + \hat{C}_{BR} \end{array}\right),$$

which correspond to loop-invariants 2 and 3 in Fig. 1. The loop-invariant on the left computes C column-wise, starting from the left. The loop-invariant on the right computes C in the reverse order, column-wise starting from the right. In compiler terminology, the relation between the two corresponding algorithms is a loop transformation called loop reversal. From the performance graphs in Fig. 7, it is clear that the ability to choose the correct direction in which to execute the loop can greatly affect the performance of the resulting code. A sufficient condition for loop reversal to be applied to a loop is that the iterations of the loop are independent. The ability to determine independent iterations from the loop-invariant and PME allows us to view loop reversal as a function for mapping one loop-invariant to another. We believe that other loop transformations, such as loop fusion and fission, can be similarly related to and performed on loop-invariants. The ability to apply loop transformations to loop-invariant allows us to simplify the required analysis for performing the transformation within the FLAME framework while leveraging the knowledge from the compiler community so as to pick the loop-invariant that should yield better performance on a given machine. We are currently investigating applying loop transformations to loop-invariants instead of to code.

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 reintroduction of indices.

This research was partially sponsored by NSF grants ACI-0305163 and CCF-0342369. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

The 4 CPU Itanium2 (1.5 GHz) server on which the experiments were conducted was generously donated to our research by 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 advice.

We would like to thank our collaborators Paolo Bientinesi, Thierry Joffrain, Dr. Kent Milfeld, and Prof. Enrique Quintana-Ortí for their contributions to parts of this work.

7. REFERENCES

[1] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software 31*, 1 (Mar. 2005).

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

[3] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software 16* (1990), 1–28.

[4] Goto, K., 2004.

[5] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software 27*, 4 (2001), 422–455.

[6] Lim, A. W., and Lam, M. S. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput. 24*, 3-4 (1998), 445–475.

[7] Low, T. M., Milfeld, K., van de Geijn, R., and van Zee, F. Parallelizing flame code with openmp task queues. Tech. Rep. TR-2004-50, The University of Texas at Austin, Department of Computer Sciences, 2004.

[8] Moler, C., Little, J., and Bangert, S. *Pro-Matlab, User's Guide*. The Mathworks Inc., 1987.

[9] Quintana-Ortí;, E. S., and van de Geijn, R. A. Formal derivation of algorithms: The triangular sylvester equation. *ACM Trans. Math. Softw. 29*, 2 (2003), 218–243.

[10] Shah, S., Haab, G., Petersen, P., and Throop, J. Flexible control structures for parallelism in openmp. In *First European Workshop on OpenMP* (2002).

[11] Shah, S., Haab, G., Peterson, P., and Throop, J. Flexible control structures for parallelism in OpenMP. In *First European Workshop on OpenMP* (1999).

[12] Singhai, S., and McKinley, K. A parameterized loop fusion algorithm for improving parallelism andcache locality, 1997.

[13] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. *MPI: The Complete Reference*. The MIT Press, 1996.

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