# On Scalable Matrix Multiply via OpenMP Workqueuing

A Technical Paper Submitted to ICPADS'06

Robert A. van de Geijn
Field G. Van Zee
The University of Texas at Austin
Austin, TX 78712

January 2006

**Abstract**

Multicore processors with a few cores are already becoming available and processors with numerous cores are reportedly not far behind. Initially, we can expect to program such processors much like SMP architectures via multi-threading and OpenMP. Eventually we anticipate numerous threads to be executing on numerous cores which raises the important question of how to program such systems. Surprisingly, even for the otherwise well-studied matrix-matrix multiplication operation little on this can be found in the literature. In this paper we present initial results from an empirical study on how to program and attain high performance for this operation on SMP systems with a large number of processors. We believe this gives some insight into how multicore processors with a moderate number of cores (16-64) may need to be programmed. Performance on a 16 CPU Itanium2 server is reported.

## 1 Introduction

The scientific computing community is unequivocally obsessed with high-performance matrix-matrix multiplication. This obsession is well justified, however; it has been shown that fast matrix-matrix multiplication is also an essential building block of many other linear algebra operations like those supported by the Level-3 Basic Linear Algebra Subprograms (BLAS) [9, 19] and the Linear Algebra Package (LAPACK) [2]. As new architectures become available, this obsession continues.

The high-performance implementation of matrix-matrix multiplication on serial architectures is well understood [14, 11]. So is the practical, scalable parallelization of this operation on distributed memory parallel architectures [1, 27, 12]. Surprisingly, little or no literature exists on the scalable implementation of this operation on symmetric multiprocessors (SMPs). With the advent of multicore processors, it is conceivable that SMPs with hundreds of threads of simultaneous execution will be widespread within a decade, since each processor of an SMP will have multiple cores. Programmability as well as scalability is an issue that must thus be addressed for this and related operations.

The Formal Linear Algebra Methods Environment (FLAME) project at UT-Austin pursues a new approach to deriving and implementing high-performance algorithms [3]. Algorithms for linear algebra operations are systematically (and, more recently, often mechanically) derived and then expressed in code in a way that hides indexing details. The methodology has been shown to be applicable to all of the BLAS, most of LAPACK, and many similar operations [13, 4, 22]. More recently, it has been extended to support SMP parallelism using OpenMP directives [21, 20].

In this paper, we demonstrate how FLAME supports SMP parallelism by utilizing the workqueuing model. We show empirical evidence that available processors need to be viewed as forming a two-dimensional grid of processors to which work is distributed. This is similar to the two-dimensional data and work distribution required for scalability on distributed memory architectures [8, 1, 27, 28, 25]. Performance results on a 16 CPU Itanium2 server are given to support this insight. An important part of this paper is the conclusion in which the presented results are used to suggest future directions that need to be pursued.

# 2 General matrix-matrix multiply

In this section we review the different special cases of matrix-matrix multiply that arise in practice as well as algorithms that support these different cases.

## 2.1 Basic algorithmic variants

In this paper, we focus on the *general matrix-matrix multiply*, or GEMM operation: $C := \alpha AB + \beta C$ where $A$, $B$, and $C$ are general (non-symmetric, non-triangular) matrices of dimensions $m \times k$, $k \times n$, and $m \times n$, respectively. For simplicity, we will assume $\alpha = \beta = 1$.

Partition each matrix by rows and by columns:

$$X = \left( \begin{array}{c|c|c} X_0 & X_1 & \cdots \end{array} \right) = \left( \begin{array}{c} \check{X}_0 \\ \hline \check{X}_1 \\ \hline \vdots \end{array} \right), X \in \{A, B, C\},$$

which represent a partitioning by columns and rows, respectively. Then

$$\left( \begin{array}{c} \check{C}_0 \\ \hline \check{C}_1 \\ \hline \vdots \end{array} \right) = \left( \begin{array}{c} \check{A}_0 \\ \hline \check{A}_1 \\ \hline \vdots \end{array} \right) B = \left( \begin{array}{c} \check{A}_0 B \\ \hline \check{A}_1 B \\ \hline \vdots \end{array} \right)$$

so that $\check{C}_i = \check{A}_i B$. Also,

$$\left( \begin{array}{c|c|c} C_0 & C_1 & \cdots \end{array} \right) = A \left( \begin{array}{c|c|c} B_0 & B_1 & \cdots \end{array} \right) = \left( \begin{array}{c|c|c} AB_0 & AB_1 & \cdots \end{array} \right)$$

so that $C_j = AB_j$. Finally,

$$C = \left( \begin{array}{c|c|c} A_0 & A_1 & \cdots \end{array} \right) \left( \begin{array}{c} \check{B}_0 \\ \hline \check{B}_1 \\ \hline \vdots \end{array} \right)$$

so that $C = A_0 \check{B}_0 + A_1 \check{B}_1 + \cdots$. We will refer to these algorithms as GEMM_VARm, GEMM_VARn, and GEMM_VARk, respectively, to indicate that the matrices are partitioned along the $m$, $n$, and $k$, dimensions, respectively.

## 2.2 Variant affinity to matrix shapes

Matrix-matrix multiplication almost always occurs in context. Frequently, the "shape" of the matrix operands is dictated by this context. For most high-performance dense linear algebra operations, like those supported by the BLAS and LAPACK, one of the dimensions ($m$, $n$, $k$) is relatively small. In Figs. 1 and 2,

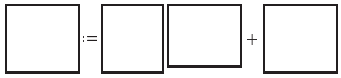
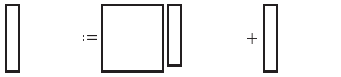
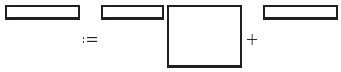
| $m$ | $n$ | $k$ | Illustration | Label | Affinity |
|---|---|---|---|---|---|
| large | large | large |  | GEMM | GEMM_VARm<br>GEMM_VARn<br>GEMM_VARk |
| large | large | small |  | GEPP | GEMM_VARm<br>GEMM_VARn |
| large | small | large |  | GEMP | GEMM_VARm<br>GEMM_VARk |
| small | large | large |  | GEPM | GEMM_VARn<br>GEMM_VARk |
| small | large | small |  | GEBP | GEMM_VARn |
| large | small | small |  | GEPB | GEMM_VARm |
| small | small | large |  | GEPDOT | GEMM_VARk |
| small | small | small |  | GEBB | |

Figure 1: Special shapes of GEMM.

| Letter | Shape | Description |
|---|---|---|
| M | Matrix | Both dimensions are large or unknown. |
| P | Panel | One of the dimensions is small. |
| B | Block | Both dimensions are small. |

Figure 2: The labels in Fig. 1 have the form GEXY where the letters chosen for X and Y indicate the shapes of matrices $A$ and $B$, respectively, according to the above table.

we give all commonly encountered shapes, and give a special label to each resulting matrix-matrix multiplication. The term "small" is, of course, subjective. In papers on the high-performance implementation of sequential matrix-matrix multiplication it has been shown that an optimal small dimension is roughly proportional to the square-root of the size of the L2 cache [11] [1].

Our goal for partitioning is simple: the matrices should be partitioned so as to create independent subproblems that may be executed in parallel by multiple threads of computation. Clearly, it is along a "large" dimension that one should partition, since partition a "small" dimension creates subproblems that have at least one suboptimal dimension. In Table 1 we indicate that a matrix shape has an *affinity* to algorithms GEMM_VARm, GEMM_VARn, or GEMM_VARk based on this insight.

[1] Later in this paper, we will take small to equal 128, a reasonable number for our target platform, the Itanium2 architecture.

### 2.3 Composite variants

Algorithms GEMM_VARm, GEMM_VARn, and GEMM_VARk partition along only one of the three dimensions ($m$, $n$, and $k$). We can synthesize *composite* variants that perform partitioning along two of the three dimensions by recognizing that the subcomputation to be performed is itself a matrix-matrix multiplication and that it therefore can be accomplished by invoking one of the three algorithms. Naturally, invoking an algorithm that partitions in the same dimension for this subproblem is equivalent to partitioning only one dimension with a finer partitioning, and therefore not interesting. Let us refer to composite variant GEMM_VARij as the basic variant GEMM_VARi which invokes variant GEMM_VARj to compute the subproblems. This allows nine composite variants to be formed from the three basic variants. However, as mentioned we wish to avoid partitioning along a given dimension more than once. This leaves only six composite algorithms. We will see that for our study GEMM_VARij is for all practical purposes equivalent to GEMM_VARji which further reduces the number of composite algorithms to three.

## 3 Partitioning for parallelism

The previous section has laid the groundwork for the discussion on parallel algorithms for matrix-matrix multiplication. Extensive work in the field of distributed-memory matrix-matrix multiplication shows that a two-dimension partitioning is needed for scalable parallelism [8, 1, 27, 28, 25]. In this paper, we conjecture that a two-dimensional *work* partitioning is also necessary to achieve scalable performance in a shared memory environment. We discuss this further in this section.

### 3.1 One-dimensional work partitioning

Parallelizing matrix-matrix multiplication on an SMP architecture with $t$ processors is a matter of partitioning the problem into $t$ subproblems (tasks) to be executed in parallel. Let us examine the three original approaches in detail:

**Algorithms** GEMM_VARm **and** GEMM_VARn**:** These algorithms can proceed by partitioning the target dimension $m$ or $n$, respectively, into $t$ (almost) equal parts. After this, $t$ completely independent computations can proceed: $\check{C}_i := \check{A}_i B + \check{C}_i$ on processor $i$ for GEMM_VARm and $C_j := AB_j + C_j$ on processor $j$ for GEMM_VARn. The only concern is to ensure that the subproblem doesn't have a dimension that is too small.

**Algorithm** GEMM_VARk**:** This algorithms can proceed by partitioning the target dimension $k$ into $t$ (almost) equal parts. After this, $t$ completely independent computations can proceed: $C^{(p)} := A_p \check{B}_p$ is computed on processor $p$. However, now these partial contributions must be added together to update $C$: $C := C^{(0)} + C^{(1)} + \cdots + C^{(t-1)} + C$. Thus, the updating by different threads must be carefully orchestrated. Again, one must ensure that the subproblem doesn't have a dimension that is too small.

### 3.2 Two-dimensional work partitioning

For the composite algorithms it suffices to factor the number of threads, $t$, into a local two-dimensional grid, $r \times c = t$, and then to partition two of the three dimensions ($m$, $n$, and/or $k$) into $r$ and $c$ (almost) equal parts.

**Algorithm** GEMM_VARmn**:** In this case, dimensions $m$ and $n$ are partitioned into $r$ and $c$ (almost) equal parts. Algorithm GEMM_VARm is employed to create $r$ equal subproblems, each of which is computed using

algorithm GEMM_VARn which subdivides each of these tasks into $c$ subproblems for a total of $t$ tasks. Each of these tasks updates a separate part of matrix $C$ so that they can all execute without further interaction.

**Algorithms GEMM_VARkm and GEMM_VARnk:** For algorithm GEMM_VARkm dimensions $m$ and $k$ are partitioned into $r$ and $c$ (almost) equal parts, respectively. Notice that since the $k$ dimension is partitioned, in that dimension contributions of smaller matrix-matrix multiplications must be summed to the appropriate submatrix of $C$. Similarly, for algorithm GEMM_VARnk dimensions $k$ and $n$ are partitioned into $r$ and $c$ (almost) equal parts, respectively. Again, the $k$ dimension is partitioned and in that dimension contributions of smaller matrix-matrix multiplications must be summed to the appropriate submatrix of $C$. This summation requires more synchronization between tasks and can thus be expected to incur a higher overhead.

# 4 Implementation with FLAME/C and Workqueuing

In this section we discuss implementation details related to the Formal Linear Algebra Methods Environment (FLAME) Application Programming Interface (API) for the C programming language, FLAME/C, and how it is supported by a model that has been proposed for addition to OpenMP, workqueuing.

## 4.1 Formal Linear Algebra Methods Environment

The FLAME project embodies a methodology for deriving algorithms for dense linear algebra operations as well as APIs for representing the resulting algorithms in code. The derivation process presents algorithms using a somewhat unconventional notation illustrated in Fig. 3(left) for GEMM_VARm. Notice that the notation manages movement through operands by tracking submatrices. Details can be found in [6]. In Fig. 3(right), we present the FLAME/C code. The algorithmic *Partition*, *Repartition* and *Continue With* statements exist within the FLAME programming interface as subroutines that hide all indexing variables inside matrix objects (also called "descriptors"). All detailed complexity related to indexing is abstracted away from the programmer within the FLAME matrix descriptors. In addition, the programmer can manipulate matrix data simply by passing the appropriate matrix subpartition descriptors to routines such as `FLA_Gemm` which perform the same operations corresponding to the BLAS call to `dgemm`. In this way the FLAME/C API frees the programmer to think at a higher level of abstraction where the computation is expressed in terms of separate partitioned regions within the matrices. A full discussion of this API can be found in [5].

## 4.2 OpenMP `task` queues

OpenMP is a set of compiler directives and library routines that aids in parallel programming on shared memory systems [24]. This allows the programmer to explicitly specify regions of code that can be executed by simultaneous threads of execution. OpenMP directives are expressed in C source code as preprocessor `#pragma` directives. An OpenMP-aware compiler recognizes these directives and then performs source code transformations to insert the "nuts and bolts" that implement the threaded parallelism.

Shah et al. [23] establish the limitations of the two most common forms of OpenMP parallelism: the `for` and `sections` directives. The authors point out that the number of iterations in OpenMP `for` loops must be computable upon first entering the loop. Similarly, the `sections` construct is "non-iterative" and statically identifies independent regions of computation. In other words, only one parallel `section` can be instantiated for every *textual* occurrence of the `section` directive within the code. The authors of [23] proposed the workqueuing model specifically to overcome limitations in the existing set of constructs.

OpenMP workqueuing, proposed for inclusion in the OpenMP 3.0 standard, provides a novel way to obtain parallelism [26]. The model consists of two new OpenMP directives: `taskq` and `task`. Conceptually,

**Algorithm:** $C :=$ GEMM_VARm( A, B, C )

**Partition**

$$A \rightarrow \left(\frac{A_T}{A_B}\right), \quad C \rightarrow \left(\frac{C_T}{C_B}\right)$$

**where** $A_T$ has 0 rows, $C_T$ has 0 rows

**while** $m(A_T) < m(A)$ **do**

**Determine block size** $b_m$

**Repartition**

$$\left(\frac{A_T}{A_B}\right) \rightarrow \left(\frac{\frac{A_0}{A_1}}{A_2}\right), \quad \left(\frac{C_T}{C_B}\right) \rightarrow \left(\frac{\frac{C_0}{C_1}}{C_2}\right)$$

**where** $A_1$ **has** $b_m$ **rows,** $C_1$ **has** $b_m$ **rows**

$C_1 := A_1 B + C_1$

**Continue with**

$$\left(\frac{A_T}{A_B}\right) \leftarrow \left(\frac{\frac{A_0}{A_1}}{A_2}\right), \quad \left(\frac{C_T}{C_B}\right) \leftarrow \left(\frac{\frac{C_0}{C_1}}{C_2}\right)$$

**endwhile**

```
void GEMM_VARm( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nthreads )
{
  FLA_Obj AT, A0,     CT, C0,
          AB, A1,     CB, C1,
              A2,         C2;
  int m, bm;

  FLA_Part_2x1( A,    &AT,
                &AB,            0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                &CB,            0, FLA_TOP );
  m = FLA_Obj_length( A );
#pragma intel omp parallel taskq
{
  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){
    bm = min( FLA_Obj_length( AB ), m/nthreads );
    FLA_Repart_2x1_to_3x1( AT,      &A0,
                          /* ** */  /* ** */
                                    &A1,
                             AB,    &A2,  bm, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,      &C0,
                          /* ** */  /* ** */
                                    &C1,
                             CB,    &C2,  bm, FLA_BOTTOM );
    /*-----------------------------------------------*/
#pragma intel omp task captureprivate( A1, C1 )
{
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              ONE, A1, B, ONE, C1 );
}  // end of task scope
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,      A0,
                                        A1,
                             /* ** */  /* ** */
                               &AB,     A2,    FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,      C0,
                                        C1,
                             /* ** */  /* ** */
                               &CB,     C2,    FLA_TOP );
  }
} // end of taskq scope
}
```

Figure 3: Left: Algorithm GEMM_VARm expressed as a FLAME algorithm. Right: Representation in C using the FLAME/C API annotated with proposed OpenMP task queue directives.

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 (determined by the runtime system) begin dequeuing and executing tasks from the queue according to a first-in/first-out (FIFO) policy. When all tasks have been completed, the threads synchronize at the end of the `taskq` scope and continue through the program.

Some code implementations, including those presented in Sec. 2.3, require an additional clause to the `task` directive: `captureprivate`. This clause allows the programmer to specify a list of variables whose values must be recorded when the task is enqueued. Typically, a variable is included in a `captureprivate` clause because the the task in question resides within a loop. The OpenMP compiler schedules the value of this variable to be "captured" at the time the task is enqueued so that sequential semantics may be preserved.

### 4.3 Parallelizing of our algorithms

An example of how the `taskq` and `task` pragmas can be used to parallelize our algorithms is given in Fig. 3. The subtasks, themselves matrix-matrix multiplications, can be executed in parallel. Naturally, the dimension, $m$, is partitioned into subparts of size `m/nthreads` in order to achieve load balance. Clearly, `gemm_varn` can be parallelized similarly.

The parallelization of `gemm_vark` is slightly more complex: Here the results of independent matrix-matrix multiplications must be added to the original matrix $C$. For details of how FLAME/C handles this we refer the reader to [20].

The composite algorithms are achieved by inlining the loop that computes the subproblem so that only a single task queue is required for the implementation. In theory, nested task queues could be used so that this in-lining needs not be performed. We have not yet experimented with this.

## 5 Performance Experiments

For our experiments, the host architecture is an NEC ccNUMA Express 5800/1000 series SMP equipped with 16 Itanium2 "Madison" core CPUs and 64GB of shared main memory. Each CPU is clocked at 1.5GHz and contains an on-chip 6MB level-3 (L3) data and instruction cache. Because the Itanium2 can execute 4 floating-point operations per clock period, each Itanium2 CPU has a theoretical peak performance of 6 GFLOPS/sec. Combined peak performance of the system utilizing all CPUs is 96 GFLOPS/sec. In our graphs, this theoretical peak performance is represented by the maximum y-axis value.

All experiments use double precision (64-bit) arithmetic and corresponding invocation of the level-3 BLAS routine `dgemm` (via the `FLA_Gemm` wrapper routine). For the subproblems assigned to each processing thread, our implementations invoke a single-threaded call `dgemm` that is part of the popular GotoBLAS library (release 0.95) and is tuned for the Itanium2 architecture [10, 11]. Reference data is obtained from the GotoBLAS multithreaded `dgemm` routine. We note that the GotoBLAS are continuously improved and that therefore the reference numbers may not be representative of the latest release of that library. Indeed, Kazushige Goto regularly attends our meetings and often makes improvements in response to our experimental insights.

Only performance for composite variants GEMM_VARmn, GEMM_VARkm, and GEMM_VARnk is reported. We choose to not observe the other composite variants because of the algorithmic symmetry discussed in Sec. 2.3. We have implemented a convenient mechanism within the FLAME/C API to allow the programmer to set the $t = r \times c$ thread factorization at runtime. Once a factorization is set, the algorithms may compute appropriate values for the subproblems. Since we were constrained by the 16 processors of the platform, $t = r \times c = 16$.

For each composite variant GEMM_VARij factorizations of the form $t \times 1$ or $1 \times t$ are algorithmically equivalent to applying only variant GEMM_VARi or GEMM_VARj, respectively. These special cases capture one-dimensional partitionings. For this reason, we do not report separate results for the basic (non-composite) variants.

Performance results are given in Figs. 4-7 for different shapes of operands. Each data point was computed from the best wall clock time of three trials. A reference curve is given in each graph corresponding to the performance results of the GotoBLAS multithreaded `dgemm`. The results are arranged to allow comparison between various thread factorizations and composite variants for each matrix shape scenario.

In general, GEMM_VARmn is the best performing variant. There is always a choice of $r \times c$ for which this variant outperforms the reference implementation. However, for some matrix shapes there is a benefit to choosing a different variant. For example, when $m$ and $n$ are both small, further partitioning of these dimensions creates subproblems for which the $m$ and $n$ dimensions are suboptimal. In this case the other two variants, which allow the $k$ dimension to be partitioned, yield better performance, as is demonstrated in Fig. 6(right).

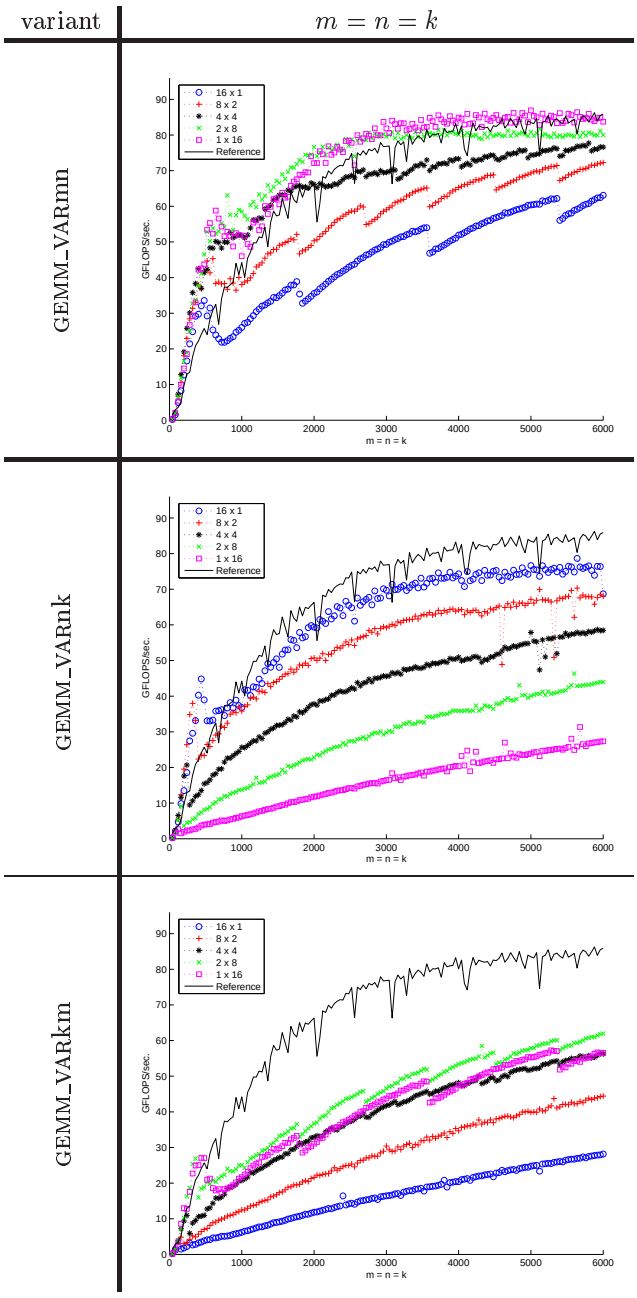



Figure 4: Performance for square matrices.

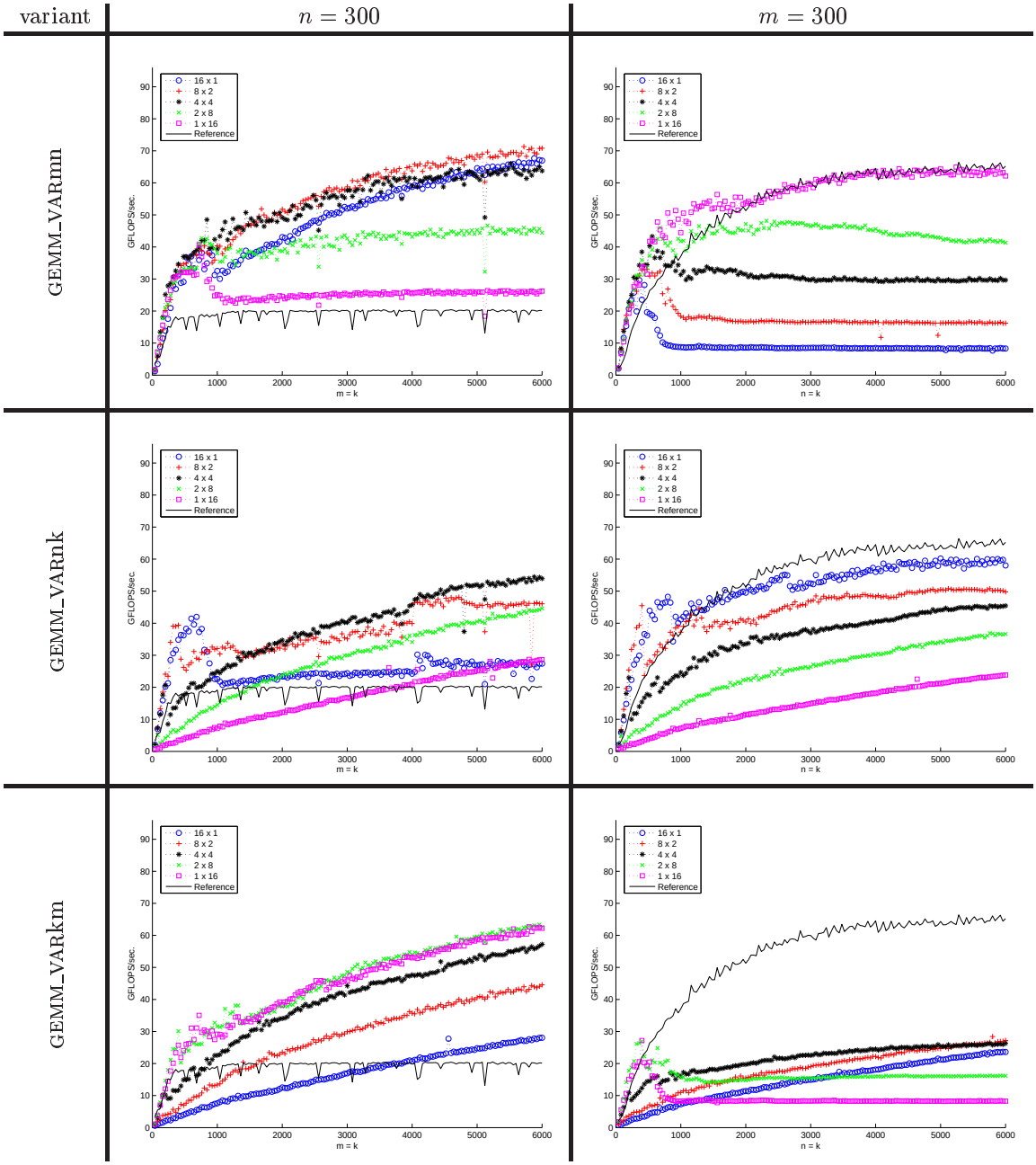



Figure 5: Left: Performance when $n$ is "small" (GEMP). Right: Performance when $m$ is "small" (GEPM).

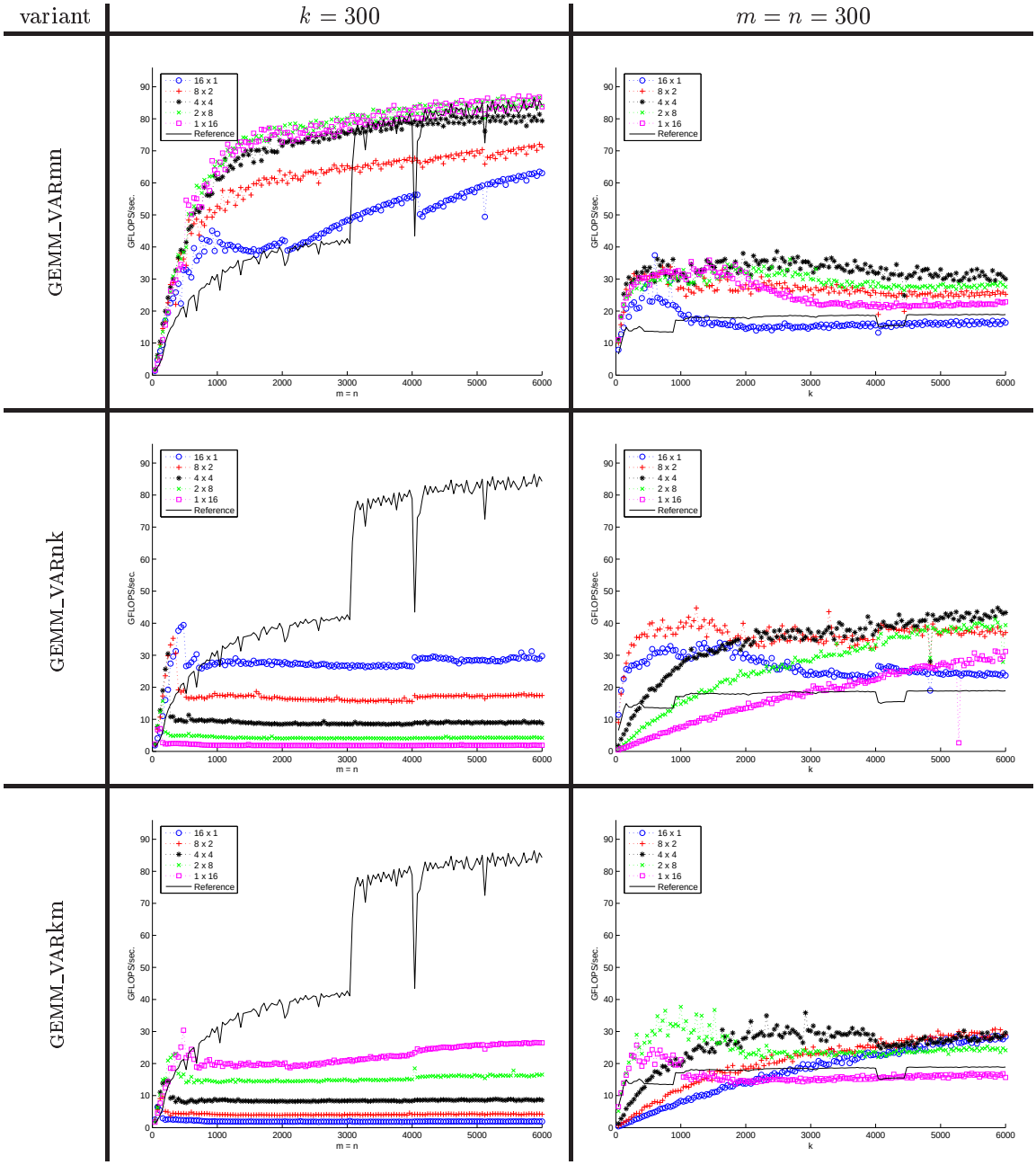



Figure 6: Left: Performance when $k = 300$ (GEPP). Right: Performance when $m = n = 300$ (GEPDOT).

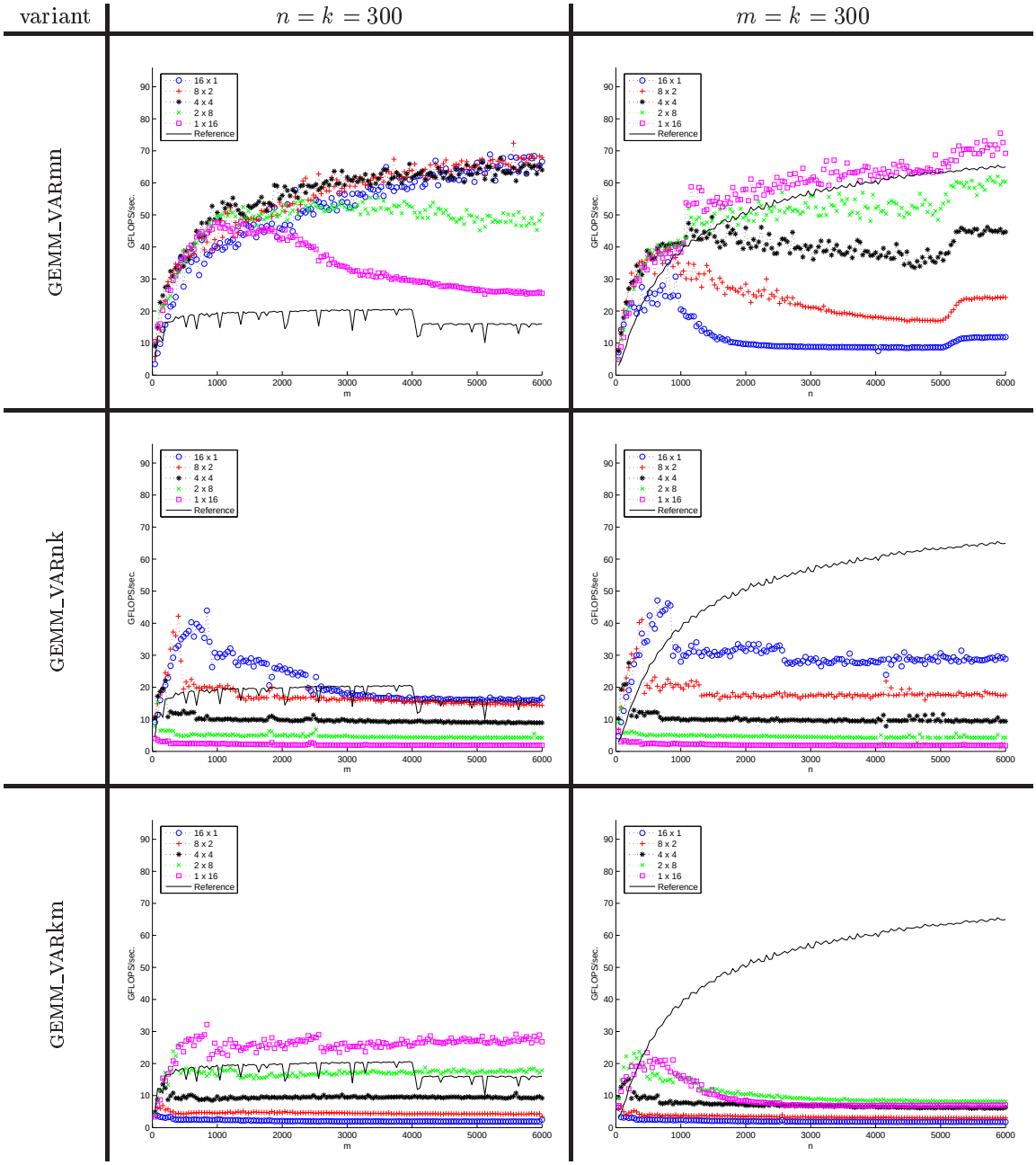



Figure 7: Left: Performance when $n = k = 300$ (GEPB). Right: Performance when $m = k = 300$ (GEBP).

**Note to the referees: We have realized that rather than setting small equal to 300 in these experiments, we should have take it to equal 128 so that partitioning in the small dimension indeed would create subproblems with an unfavorably small dimension. We believe that this would show much more dramatically the point we are trying to make. Unfortunately, the sixteen CPU system on which we had performed our experiments was dismantled before we realized this. We have every intention of identifying another sixteen CPU Itanium2 system so that we can substitute such data in the final version of this paper.**

# 6 Conclusions

We have illustrated issues related to the parallel implementation of the matrix-matrix multiplication operation on SMP architectures. Initial empirical evidence has been given that partitioning work along two dimensions is necessary for scalability. The elegance and effectiveness of the work queuing model as proposed for OpenMP 3.0 was also illustrated, especially for FLAME code.

A number of issues are not discussed in this paper and are ripe for future research.

- How can workqueuing be supported in the absence of OpenMP task queues or in the absence of OpenMP altogether? We have prototyped a mechanism to support this for FLAME which will be the topic of a future paper.
- How can compiler technology be used to reduce some of the overhead associated with the peculiar indexing mechanism used by FLAME? We believe that abstracting away from detailed indexing is beneficial and makes more information available to a compiler like Broadway [16, 18, 17, 15].
- How can one determine optimal parameters? In this paper we merely illustrate that parameters for partitioning the threads into a grid and other blocking parameters impact performance. We do not believe that approaches like PHiPAC [7] and ATLAS [29] are feasible. What we do believe is that the underlying matrix-matrix multiply provided by the GotoBLAS [11] can be highly accurately modeled, which than can facilitate the highly accurate modeling of the demonstrated SMP algorithms. Such a model can then be used to motivate heuristics for choosing the best algorithms.
- How can redundant effort be avoided? High-performance matrix-matrix multiplication inherently involves the copying of data to make it contiguous [11]. In the proposed algorithms, this copying is duplicated within the call to `FLA_Gemm`, which is itself a wrapper to the `dgemm` BLAS routine. Thus, such copying is duplicated by each of the threads which is clearly a waste. Worse than that, this copying causes the different threads to compete for memory bandwidth which further amplifies overhead.
- How do the insights impact other BLAS operations? Initial results, reported in [20], suggest that load-balancing is somewhat more complex for the other level-3 BLAS operations.
- What will multi-core architectures with many cores look like? And that is the billion dollar question yet to be answered.

All of these questions are part of our future research.

## Further information

Additional information regarding the FLAME project may be found by visiting
`http://www.cs.utexas.edu/users/flame/` .

### Acknowledgments

The FLAME working group became aware of the OpenMP workqueuing model through Dr. Timothy Mattson (Intel). The high-level constructs afforded by this model were key in guiding our research.

Many thanks go to NEC Solutions (America), Inc for providing access to their two 16 CPU Itanium2 compute servers on which many of the performance experiments were performed. NEC Solutions (America) also provided partial funding for this project. Additional funding came from Dr. James Truchard, President, CEO, and Cofounder of National Instruments. This research was also 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.

We thank Kazushige Goto for his technical advice and acknowledge Paolo Bientinesi, Sukant Hajra, Tze Meng Low, and Dr. Kent Milfeld for their valuable feedback throughout the course of this research.

## References

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development*, 38(6), 1994.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* SIAM, 1994.

[3] Paolo Bientinesi, Kazushige Goto, Tze Meng Low, Enrique Quintana-Orti, Robert van de Geijn, and Field Van Zee. Flame 2005 prospectus: Towards the final generation of dense linear algebra libraries. Technical Report FLAME Working Note 15, CS-TR-05-15, Department of Computer Sciences, The University of Texas at Austin, December 2005. `http://www.cs.utexas.edu/users/flame/pubs/`.

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[5] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft.*, 2005. to appear.

[6] Paolo Bientinesi and Robert van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. *SIAM Review.* submitted.

[7] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing.* ACM SIGARC, July 1997.

[8] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, Sept. 1997.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] Kazushige Goto. `http://www.cs.utexas.edu/users/kgoto`, 2004.

[11] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.* submitted.

[12] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[14] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[15] Samuel Z. Guyer, Emery Berger, and Calvin Lin. Customizing software libraries for performance portability. In *$10^{th}$ SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[16] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.

[17] Samuel Z. Guyer and Calvin Lin. *Broadway: A Software Architecture for Scientific Computing*, pages 175–192. Kluwer Academic Press, October 2000.

[18] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, pages 221–238, August 2000.

[19] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[20] Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. Parallelizing FLAME code with OpenMP task queues. Technical Report FLAME Working Note 15, CS-TR-04-50, Department of Computer Sciences, The University of Texas at Austin, December 2005. `http://www.cs.utexas.edu/users/flame/pubs/`.

[21] Tze Meng Low, Robert van de Geijn, and Field Van Zee. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of PPoPP'05*.

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

[23] Sanjiv Shah, Grant Haab, Paul Peterson, and Joe Throop. Flexible control structures for parallelism in OpenMP. In *EWOMP*, 1999.

[24] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[25] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[26] Ernesto Su, Xinmin Tian, Milind Girkar, Grant Haab, Sanjiv Shah, and Paul Peterson. Compiler support of the workqueuing execution model for Intel SMP architectures. In *EWOMP*, 2002.

[27] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

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

[29] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.