RC23715 (W0509-039) September 12, 2005
Mathematics

IBM Research Report

Algorithm Compiler Architecture Interaction Relative to Dense Linear Algebra

Fred G. Gustavson
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Algorithm Compiler Architecture Interaction Relative to Dense Linear Algebra

Fred G. Gustavson
IBM T.J. Watson Research Center
phone: 1-914-945-1980
email: fg2@us.ibm.com

September 7, 2005

Abstract

We present several ideas for the development of sequential and parallel dense linear algebra software. The algorithms of Linpack and Eispack and later LAPACK and ScaLAPACK have stood the test of time in terms of robustness and accuracy. We focus on producing high performance versions of these algorithms. Our main results use the Algorithms and Architecture Approach. It will be seen that these ideas affect both Architecture and Compiler Design. The paper briefly discusses the following topics:

1. The Linear Transform Approach as a general way to produce traditional algorithms.
2. The underlying role of Matrix Multiplication.
3. The use of matrix partitioning to describe traditional algorithms.
4. The two standard Data Structures of Dense Linear Algebra hurt performance.
5. The Packed Data Format has poor performance relative to Standard Full Format.
6. Standard full format meshes well with Industry Standards.
7. Standard full format waste half the storage for Triangular matrices.
8. A novel hybrid full format for Triangular matrices.
9. The LAPACK and Level 3 BLAS approach has a basic flaw.
10. The ScaLAPACK and PBLAS approach has the same basic flaw.
11. New Block-based Data Structures for Matrices remove the flaw.
12. Industry Standards relative to Dense Linear Algebra Software.
13. Industry Standards relative to Distributed Memory Computing
14. The role of concatenating submatrices to facilitate hardware streaming.
15. A need for Architecture to enlarge the Floating Point Register File.

We describe a *new* result by showing that representing a matrix A as a collection of square blocks can reduce the amount of data reformating required by LAPACK factorization algorithms from O(n^3) to O(n^2). *New* results are presented for rectangular full packed format which is a standard full format array using minimal storage for representing a symmetric or triangular matrix. The LAPACK library contains some 125 times two (full and packed storage) routines for symmetric or triangular matrices. Equivalent routines, written for this new format, usually consist of just calls to Level 3 BLAS and existing LAPACK full format routines. Hence they are trivial to produce. We give several examples for Cholesky factorization and an example for both triangular inverse and Cholesky inverse. Finally, we introduce two *new* distributed memory near minimal storage algorithms using square block packed format for Cholesky factorization. We only give performance

results for Square Block Format Cholesky, on an IBM Power 3, and Rectangular Full Format, on an IBM Power 4.

1 Introduction

We present a novel way to produce Dense Linear Algebra Factorization Algorithms (DLAFAs) for serial and Distributed Memory Computing (DMC). The current Most Commonly Used (MCU) Dense Linear Algebra (DLA) algorithms for such systems have a performance inefficiency and hence they give sub-optimal performance for most of LAPACK's factorizations. We show that standard Fortran and C two dimensional arrays are the main reason for the inefficiency. For the other standard format (packed one dimensional arrays for symmetric and/or triangular matrices) the situation is much worse. We introduce Rectangular Full Packed (RFP) format which represents a packed array as a full array. This means that performance of LAPACK's packed format routines becomes equal to or better than their full array counterparts. Hence, RFP format should replace packed format. Returning to full format, we also show how to correct these performance inefficiencies by using New Data Structures (NDSs) along with so-called kernel routines. The NDS generalizes the current storage layouts for both standard layouts. The BLAS [32, 11, 12] (Basic Linear Algebra Subroutines) were introduced to make the algorithms of DLA performance-portable. However, a relationship exists between Level 3 BLAS use in most of level 3 factorization routines of the LAPACK library. This relationship introduces a performance inefficiency in LAPACK algorithms and we will now discuss the Level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix) to illustrate this fact. This paper is a condensation and continuation of [23]. To make it self contained we shall repeat or re-formulate certain essential parts of [23]. A re-formulation and condensation of it is to appear in [24]. At the end of the paper we include a Glossary of the Acronyms that we will be using.

In [2, 7, 41, 21] design principles for producing a high performance "Level 3" `DGEMM` BLAS are given. A key design principle for `DGEMM` is to partition its matrix operands into submatrices and then call an L1 kernel routine multiple times on its submatrix operands. The suffix i in Li stands for level i cache. Li is not to be confused with Level i BLAS. Another key design principle is to change the data format of the submatrix operands so that each call to the L1 kernel can operate at or near the peak Million FLoating point OPerations per Second (MFlops) rate. This format change and subsequent change back to standard data format is a cause of a performance inefficiency in `DGEMM`. The `DGEMM` interface definition requires that its matrix operands be stored as standard Fortran or C two-dimensional arrays. Any LAPACK factorization routine of a matrix A calls `DGEMM` multiple times with *all* its operands being submatrices of A. For each call data copy will be done; the principle inefficiency is therefore multiplied by this number of calls. However, this inefficiency can be eliminated by using the NDS to create a substitute for `DGEMM`, e.g. its analogous L1 kernel routine, which does *not* require the aforementioned data copy.

The MCU software for DLA is probably the LAPACK Library. Thus, the standard data layout for matrices becomes, for the most part, the two dimensional rectangular full array of the Fortran and C languages. The Level 1,2,3 BLAS are an Industry Standard and they support the DLA algorithms of the LAPACK library. Also, for the most part, the input and output formats of the BLAS are the two dimensional arrays of the Fortran and C languages. Now the LAPACK library and the Level 2 BLAS are partly written in terms of the packed data format that was introduced in 1950 and supported since by the DLA community. Their purpose was to represent symmetric and triangular matrices using minimal storage. Note that placing these matrices in a standard full array wastes half the storage of the full array. At this point we want to dispense with the packed data format and Level 2 packed BLAS as this paper demonstrates that packed data format can be represented by RFP format. RFP format is a standard full array that requires minimal storage and it is easy to write LAPACK library software for this format; see also [20]. Now, looking at BLAS written for the full format, we find they are designed for shared memory processors which use the Industry Standards Threads and Open MP. The point of this paragraph was to argue that the Industry Standards for DLA and the standard full format arrays of Fortran and C mesh together very well on shared memory machines; ie, to cover Point 6

of the Abstract.

We introduce a NDS as a replacement for standard Fortran or C array storage. One of the key insights is to see that storing a matrix as a collection of submatrices (eg, square blocks of size `NB`) leads to very high performance on today's, RISC type, processors. Some NDS order the blocks in standard Fortran or C order; ie, store the blocks either in column-major or row-major order. However, to facilitate hardware streaming, [31, 38], it may be necessary to reformat the interior of each block. The main benefit of the simpler data layout is that addressing of an arbitrary $a(i, j)$ element of matrix A can be easily handled by a compiler and/or a programmer. We call the NDS *simple* if the ordering within the blocks as well as the ordering of the blocks themselves follows the standard row / column major order.

For level 3 algorithms, the basis of the ESSL (Engineering and Scientific Subroutine Library) is a set of kernel routines that achieve peak performance when the underlying arrays fit into L1 cache [2, 27]. If one were to adopt the new, simple NDS then BLAS and LAPACK type algorithms become almost trivial to write. At this point we shall follow [23] closely as we want to clarify how NDS differs from standard full format. Briefly, NDS are a good format for BLAS kernels and full format is not. Also, the combination of using the NDS with kernel routines is a general procedure and for matrix factorization it helps to overcome the current performance problems introduced by having a non-uniform, deep memory hierarchy. We use the Algorithms and Architecture (AA) approach, see [2], to illustrate what we mean and in doing so we cover Point 3 of the Abstract. We shall make eight points below. Points 1 to 3 are commonly accepted architecture facts about many of today's processors. Points 4 to 6 are DLA algorithms facts that are easily demonstrated or proven. Points 7 and 8 are a natural conclusion based on the AA approach.

1. Floating point arithmetic usually cannot be done unless the operands involved first reside in the L1 cache.

2. Two-dimensional Fortran and C arrays do *not* map nicely into L1 cache.

   (a) The best case happens when the arrays are contiguous and properly aligned.

   (b) At least a three-way set associative cache is required when matrix multiply is being done as `DGEMM` has three operands.

3. For peak performance, all matrix operands must be used multiple times when they enter L1 cache.

   (a) This assures that the cost of bringing an operand into cache is amortized by its level 3 multiple re-use.

   (b) Multiple re-use of all operands only occurs when all matrix operands map well into L1 cache.

4. Each scalar $a(i, j)$ factorization algorithm has a square submatrix counterpart `A(I:I+NB-1,J:J+NB-1)` algorithm. See

   (a) Golub and Van Loan's "Matrix Computations" book, [18].

   (b) The LAPACK library.

5. Some submatrices are both contiguous and fit into the L1 cache.

6. Dense matrix factorization is a level 3 computation.

   (a) Dense matrix factorization, in the context of Point 4, is a series of submatrix computations.

   (b) Every submatrix computation (executing any kernel routine) is a level 3 computation that is done in the L1 cache.

   (c) A level 3 L1 computation is one in which each matrix operand gets used multiple times.

From Points 1-6, we conclude Points 7 and 8:

7. Map the input Fortran/C array (matrix A) to a set of contiguous submatrices each fitting into the L1 cache.

   (a) For portability (using block hybrid format) perform the inverse map after applying Point 8 (below).

8. Apply the appropriate submatrix algorithm.

The book [18], Point 4a, gives a detailed listing of the scalar algorithms and describes (with references to the research literature) their block submatrix counterparts. The LAPACK library, Point 4b, gives code for the submatrix counterpart algorithms. The block submatrix codes of Point 4b use Fortran and C to input their matrices, so Point 5 does *not* hold for MCU algorithms. See page 739 of [22] for more details. Point 5 does hold for the NDS described here. Assuming both Points 5 and 6 hold, we see that Point 3 holds for every execution of the kernel routines that make up the factorization algorithm. This implies that near peak performance will be achieved. Points 7 and 8 suggest an algorithmic change that is justified by Points 1 to 6. Point 7 is pure overhead for the new algorithms. Using the new data formats reduces this cost to zero. By only doing Point 8 we see that we can get near peak performance as every subcomputation of Point 8 is a Point 6b computation. Note that each kernel call of the submatrix algorithm is a level 3 call done in L1 cache and so, on average, every scalar of each submatrix gets used multiple times.

Now we discuss the use of kernel routines in concert with NDS. Take any standard linear algebra factorization code, say Gaussian elimination with partial pivoting or the QR factorization of an `M` by `N` matrix, A. It is quite easy to derive the block equivalent code from the standard code. In the standard code a floating point operation is usually a Fused Multiply Add (FMA), ($c = c - ab$), whose block equivalent is a call to a `DGEMM` kernel. Similar analogies exist; eg, for $b = b/a$ or $b = b * a$, we have a call to either a `DTRSM` or a `DTRMM` kernel. In the simple block equivalent codes we are led to one of the variants of IJK order, [10]. For these types of new algorithms the BLAS are simply calls to kernel routines. It is important to note that no data copying need be done.

There is one type of kernel routine that deserves special mention. It is the factor kernel. Neither LAPACK nor the research literature treat factor kernels in sufficient depth. For example, the factor part of LAPACK level 3 factor routines are level 2 routines; they are named with the suffix `TF2`, and they call Level 2 BLAS repetitively. On the other hand ESSL, [2], and more recently, [22, 25, 14, 26, 4, 15, 16, 3], where sometimes recursion is used, have produced level 3 factor routines that employ level 3 factor kernels to yield level 3 factor parts. The above four paragraphs and its Points 1 to 8 are also a basis of some of the MCU LAPACK library. We are referring to LAPACK's level 3 full format DLAFAs. However, these routines are based on using Level 3 BLAS which uses standard full format Fortran or C 2-D arrays and on level 2 factorization parts of these routines which uses Level 2 full BLAS. Besides Point 3 of the Abstract these four paragraph cover Points 4, 9, 11 and 12 of the Abstract.

Besides full storage, there is packed storage, which is used to hold symmetric/triangular matrices. Using the NDS instead of the standard packed format [23] describes new algorithms that save "half" the storage of full format for symmetric matrices and outperform the current block based level 3 LAPACK algorithms done on full format symmetric matrices. Also, we introduce RFP format which is a variant of Hybrid Full Packed (HFP) format. HFP format is described in [20]. RFP format is a rearrangement of a standard full storage array holding a symmetric / triangular matrix A into a compact full storage rectangular array AR that uses minimal storage `NT=N(N+1)/2`. Therefore, Level 3 BLAS can be used on AR. In fact, using AR instead of A on an equivalent LAPACK algorithm gives slightly better performance. This offers the possibility to replace all packed or full LAPACK routines with equivalent LAPACK routines that work on AR. We present a new algorithm and indicate its performance for Cholesky factorization and inverse using AR instead of full A or packed AP; see also [20]. This paragraph is covering a *new* result called RFP format and it properties and Points 4, 5 and 7 of the Abstract.

Let us now consider some Industry Standards for Single Program Multiple Data (SPMD) machines. For DLA and ScaLAPACK [39] in particular one uses a `P`$\times$`Q` mesh of processors (processes). The programming model is a Block Cyclic Layout (BCL) of a rectangular full format array. As such, symmetric and triangular matrices are represented in full arrays which waste half the storage on all `PQ` processors. Currently, the LAPACK and ScaLAPACK project, [9], has a NSF funded project to investigate and produce a "packed array solution" with supporting PBLAS. A major result of this paper is to present two solutions to this problem; one is a detailed solution and the second is only a sketched solution. Both solutions use

nearly minimal storage on each of the PQ processors. Again, we argue that the Industry Standards for DLA and standard full *rectangular* format arrays of Fortran and C mesh together very well for SPMD machines (see Item 13 of the Abstract). However, there are no distributed memory industry standards for minimal storage arrays holding symmetric and triangular matrices. Our new results in Section 4 suggest new industry standards.

In paragraphs one and two of this Introduction we briefly discussed a flaw in the LAPACK and Level 3 BLAS approach. The NDS, discussed in this Introduction, removes this flaw. It turns out that a Square Block (SB), these SB's make up our new NDS, is the fundamental building block of the standard BCL. Therefore, it should not be surprising that our NDS can lead to high performing DMC ScaLAPACK type algorithms for the SPMD programming model. Also, it is fairly easy to see that the ScaLAPACK PBLAS approach has the same flaw as the LAPACK and Level 3 BLAS approach. We suggest that this flaw is compounded as ScaLAPACK represents the distributed global rectangular matrix A on each processor of a $\mathtt{P} \times \mathtt{Q}$ mesh as a single full format array. These single full format arrays are also, by definition, a rectangular array consisting of just SB's; ie, our NDS.

So, another generalization of using NDS applies or relates to ScaLAPACK. The standard BCL on a $\mathtt{P} \times \mathtt{Q}$ mesh of processors has parameter NB. Therefore, it is natural to view the square submatrices of order NB that arise in these layouts as atomic units. Now, many ScaLAPACK algorithms can be viewed as right looking LAPACK algorithms: factor and scale a pivot panel, broadcast the scaled pivot panel to all processors, and then perform a Schur Complement Update (SCU) on all processors. Three example are $LU = PA$, PDGETRF; $QR = A$, PDGEQRF; $LL^T = A$, PDPOTRF of ScaLAPACK. We describe in Section 4 some benefits:

1. since P and Q are arbitrary integers, the square blocks can move about the mesh as contiguous atomic units

2. it is possible to eliminate the PBLAS layer of ScaLAPACK as only standard Level 3 BLAS are needed

3. for triangular / symmetric matrices one only needs to use about half the storage.

In Section 4, we outline two DMC Right Looking Algorithms (RLAs) for Cholesky Factorization on a $\mathtt{P} \times \mathtt{Q}$ mesh of processors for the SPMD programming model. They both have the features (1-3) above. However, our two $LL^T = A$ algorithms will use near minimal storage and so can be codes for the newly proposed PDPPTRF ScaLAPACK algorithm [9]. In Sections 4.1 and 4.2 we give full details on one of these algorithms. The above three paragraphs are covering Points 10, 11, 12 and 13 of the Abstract.

In Section 2 we describe some basic algorithmic and architectural results as a rationale for the work we are presenting. The idea is to briefly elucidate the AA approach [2]. An architecture with an FMA instruction has a real advantage and we argue this assertion in Section 2. We describe the linear transformation approach to producing DLA algorithms and use it as a foundation for producing their high performance implementations. Section 2.1 describes a *new* concept which we call the L1 cache / L0 cache interface [5]. The L0 cache is the register file of a Floating Point Unit. Today, many architectures possess special hardware to support the streaming of data into the L1 cache from higher levels of memory [31, 38]. In fact with a large enough floating point register file it may be possible to do, say, a L2 or L3 cache blocking for a DGEMM kernel; ie, completely bypass the L1 cache. This is the case in [5] where a 6 by 6 register block for the C matrix can be used as this processor has 32 dual SIMD floating point registers. To do L0 register blocking we can concatenate tiny submatrices to faciltate streaming; ie, to reduce the number of streams. In effect, at the L0 level we have a concatenation of tiny submatrices behaving like a single long stride one vector that passes through L1 and into L0 in an optimal way. Sections 2.1, 2.2 and 2.21 gives details about this technique. Using this extra level of blocking does not negate the benefits of using SB's. It is still essential that $\mathtt{NB}^2$ elements of a SB be contiguous. However, the SB's are now no longer simple according to our definition. And using non-simple SB's as described here allows us to claim in Section 3.2 that data copy for RLAs using SBs can be O(N^2) instead of O(N^3) which occurs for standard Fortran or C two dimensional arrays. This paragraph is addressing an application of the AA approach and Points 1, 2, 4, 11, 12, 14 and 15 of the Abstract.

In Section 3.1 we describe Square Block Packed (SBP) format for symmetric/triangular arrays and show that they generalize both the standard packed and full arrays used by DLA algorithms. This Section is covering SBP format which is the NDS of Point 8 of the Abstract. In Section 3.1.1 there is a short discussion about four serial Cholesky factorization algorithms. In general, three of these algorithms, called left looking, right looking and recursive, [10, 22, 19, 16] apply to most DLAFA's. For each DLAFA one can use any appropriate matrix data structure. Thus, data structures and DLAFA's are independent components and hence performance of a factorization code becomes the product of the number of elements in each catorgory. It follows that an implementer or compiler writer should try many possibilities in order to find an optimally performing one. Using the Level 3 BLAS and standard full format data structures the FLAME approach [19] describes an automatic procedure to select a "best" performing DLA algorithm. We have just covered another application of the AA approach and also Point 4 of the Abstract. We describe RFP format arrays in Section 3.3. They can be used to replace both the standard packed and full arrays used by DLA algorithms. A minimal new coding effort is required as existing LAPACK routines would constitute most of the new code. Section 3.3 demonstrates this for Cholesky factorization, and Section 3.4 demonstrates this for LAPACK Cholesky inverse and triangular inverse codes. Performance results of RFP format versus LAPACK for Cholesky factorization and inverse are also given. In the comparison, only LAPACK and Level 2,3 BLAS codes execute for both RFP and the standard full and packed formats. Section 3.5 gives conclusions about RFP format. This paragraph is addressing Points 4, 5, 6 and 7 of the Abstract.

We have said very little about the Compiler in regard to the AA approach. What we have in mind is to try and automate the production of the kernel routines by a Compiler which we think can be done.

2 Rationale and Underlying Foundations of our Approach

The purpose of this section is to argue that DLAFA's are nothing more than matrix multiplication in disguise. Many reseachers have made this observation; eg, [28]. Our ammunition goes back to the ground breaking work of the originator of matrix theory, Cayley. He invented the matrix and first defined matrix multiplication. Again, we follow part of the description given in [23].

For solving a set of linear equations $Ax = b$ there are two points of view. The more popular view is to select an algorithm, say Gaussian elimination with partial pivoting, and use it to compute x. The other view, which we adopt here, is to perform a series of linear transformations on both A and b so that the problem, in the new coordinate system, becomes simpler to solve. Both points of view have their merits. We use the second as it demonstrates some reasons why the AA approach, [2], is so effective. Briefly, the AA approach states that the key to performance is to understand the algorithm and architecture interaction. Furthermore a significant improvement in performance can be obtained by matching the algorithm to the architecture and vice-versa. In any case, it is a very cost-effective way of providing a given level of performance.

The fundamental reason or idea behind the coordinate transformation approach is the concept of Equivalence and Elementary Matrices. In [6], pages 170-173, matrices and row-equivalence are discussed. In particular, elementary row operations of three types are cited on page 172. The most important is the addition of any multiple of one row to any other row of a matrix. See also [34], Chapter 6, Elementary Operations and the concept of Equivalence, for another treatment. Closely related to elementary operations (there are both row and column types) are elementary matrices E which are a rank one modification of the identity matrix I : $E = I + \sigma uv^T$, where u and v are vectors and σ is a scalar. We have the following:

THEOREM Let $Ax = b$ represent an m by n linear system of equations. Let T represent an elementary operation or an elementary matrix. Let $A_1x = b_1$ represent the m by n linear system of equations after applying T to both sides of

$Ax = b$, i.e., $A_1 = TA$ and $b_1 = Tb$. Then the solution properties of both systems are the same.

Note that a more general form of an elementary operation can be considered a linear transformation.

COROLLARY Let $T_i, 1 \leq i \leq k$ represent k linear transformations where each T_i is elementary. Let $T = T_k...T_1$. Then $Ax = b$ and $Cx = d$ where $C = TA$ and $d = Tb$ have the same solution properties.

As an example, we relate this second approach to the first approach of Gaussian elimination with partial pivoting, i.e., $LU = PA$. We get $C = U$ and $k = n$ when the linear transformations are $T_i = L_i$ or $k = \lceil n/NB \rceil$ when a blocked method is used. A second example would be $A = QR$ factorization where $C = R$ and the T_i would be elementary Householder matrices or the compact WY representation, [14, 15]. We remark that Section 2, pages 939-942 of [8] shows that product of n L_i to produce L requires *no* additional work, i.e., L is obtained via concatenation of the n L_i. This is *not* the case with elementary Householder matrices; see Section 2, pages 606-615 of [14].

Now we examine a single elementary column operation. Let the two columns be represented by vectors x and y and the scalar multiple by α. Then this operation is the Level 1 BLAS `DAXPY` operation $y = y + \alpha x$. Note that `DAXPY` is a series of multiply-add operations. In fact the dot-product $x^T y$ operation, another pervasive operation, is also a series of multiply-add operations. For DLA we can conclude that multiplies and additions occur equally often and almost always in multiply-add pairs. Hence, from the architecture point of view, the use of the FMA instruction, `T=B+A*C`, is a natural architecture choice for DLA. We mention that the FMA instruction costs slightly more than the multiply instruction and that doing a multiply and an add costs about 1.7 times more than a FMA, [35, 17, 36].

Next we claim that matrix multiplication is pervasive in the algorithms of DLA; eg, see the book [18], and the `LAPACK` library. Let R and S be linear transformations with a common set of basis vectors. Let $T = S(R)$ be the composition of the two linear transformations where we want T to be linear. This restriction, i.e., that T be linear, on the basis for T, in terms of the common set of basis vectors, *defines* matrix multiplication. In fact, in the 1840's Cayley first described a matrix as a rectangular two-dimensional array. According to Meyer [33], Cayley also defined matrix multiplication as the result of the composition of two linear coordinate transformations. We take the same view here. Our dual point of view lets us describe DLA algorithms as a series of linear coordinate transformations with a common set of basis vectors. And for each such composition of transformations to be linear we *must* perform matrix multiplication. See [6], Chapter 8, pages 209-214 of for more details. We also note that matrix multiply is by *definition* a series of parallel dot-product and hence just a series of FMAs, which can be done independently.

We have just seen why matrix multiply repeatedly shows up in say, `LAPACK` algorithms. In fact, the Level 3 BLAS, `DGEMM`, is called the most important Level 3 BLAS. Our next point relates to the current block based (submatrix) matrix multiplication used by most `DGEMM` implementations. These implementations are optimal in the following sense:

THEOREM (Toledo, [40]) Any algorithm that computes $a_{i,k}b_{k,j}$ for all $1 \leq i, j, k \leq n$ must transfer between memory and an M word cache $\Omega(n^3/\sqrt{M})$ words if $M < n^2/5$.

The current block based algorithms transfer $O(n^3/\sqrt{M})$ words. The point here is that we need not search for better ways to perform matrix multiplication via other `DGEMM` implementations as our current algorithms are achieving the lower bound complexity measure. Also, practical experimental evidence, (eg ESSL's `DGEMM` achieves better than 90 % of peak performance), tells us the same thing in a very concrete way.

We end this section with brief remarks about blocking. The general idea of blocking is to get information to a high speed storage and use it multiple times to amortize the cost of moving the data. In doing so, we satisfy Points 1 and 3 of the Introduction. We only touch upon the Translation Look-aside Buffer (TLB), cache, and register blocking. The TLB contains a finite set of pages. These pages are a good approximation of the *current working set* of the computation. If the computation addresses only memory in the TLB then there is no penalty. Otherwise, a TLB miss occurs resulting in a large performance penalty; see [2]. Cache blocking reduces traffic between the memory and or a higher level cache and the L1 cache. Analogously, register blocking reduces traffic be-

tween the L1 cache and the registers of the CPU; ie, the floating point register file. Cache and register blocking are further discussed in [2]. Section 2 is covering an application of the AA approach and Points 1, 2 and 3 of the Abstract.

2.1 The Need to Reorder a Contiguous Square Block

NDS represent a matrix A as a collection of SB's of order NB. Each SB is contiguous in memory. In [37] it is shown that a contiguous block of memory maps best into L1 cache as it minimizes L1 and L2 cache misses as well as TLB misses for matrix multipy and other common row and column matrix operations. When using standard full format on a DLAFA one does an O($(N/NB)^2$) amount of data copy in calling DGEMM in an outer do loop: j=1,N,NB. Over the entire DLAFA this becomes O(N^3).

Computer manufacturers have recently introduced hardware, eg. [5], that initiates multiple floating point operations (two to four) in a single cycle. Also, each floating point operation requires several cycles (five to ten) to complete. Therefore, one needs to be able to schedule many (ten to forty) independent floating point operations every cycle. Thus, if one wants to run their floating point applications at their peak MFlops rate it is sufficient for hardware to introduce larger floating point register files (storage for the operands and results of the floating point units). We call this tiny memory the L0 cache.

We now want to discuss a *new* concept which we call the L1 cache / L0 cache interface. On some RISC processors there are floating point multiple load and store instructions associated with the multiple floating point operations; see [2, 5]. A multiple load / store operation usually requires that its multiple operands be contiguous in memory. Some newer processors with multiple floating point operations require their operands to be contiguous; eg, [5]. So, data that enters L1 may also have to be properly ordered to be able to enter L0 in an optimal way. Unfortunately, layout of a SB in standard row / column major order may *no longer* lead to an optimal way. In some cases it is sufficient to reorder a SB into submatrices which we call register blocks. Doing this produces a new data layout that will still be contiguous in L1 but can also be loaded into L0 from L1 in an optimal manner. Of course, the order and size in which the submatrices (register blocks) are chosen will be platform dependent. This section is addressing another application of the AA approach and Points 8, 9, 10, 11, 14 and 15 of the Abstract.

2.2 A DGEMM kernel based on Square Block Format

Currently, on some platforms, register blocks can be considered as submatrices of a SB. This fact is very important as it means one can still more easily use the AA approach. To see this let A, B and C be three SB's and suppose we want to apply DGEMM to A, B and C. If we partition A, B and C into conformable submatrices that are register blocks then we can use Points 1 to 6 of the Introduction at the register block level to obtain a near optimal kernel for DGEMM.

Let A^T be K $\times$ M, B be K $\times$ N and C be M $\times$ N. Here one should think that M, N, K are of the order of NB. Note that any SB stored in column major order is identical to its transpose stored in row major order and vice-versa. So, we want to compute $C = C - A^T B$ as matrix multiply is stride one across the rows and columns of A and B respectively. Let the sizes of the register blocks be kb $\times$ mb, kb $\times$ nb and mb $\times$ nb. Thus A^T, B and C are matrices of register blocks of sizes $k_1 \times m_1$, $k_1 \times n_1$ and $m_1 \times n_1$ respectively. Now consider a fundamental DGEMM kernel building block which consists of multiplying k_1 register blocks of A^T by k_1 register blocks of B and summing them to form the update of a register block of C. The entire kernel will therefore consist of executing $m_1 \times n_1$ fundamental building blocks in succession. And, each element of A, B and C gets N, M, K reuse respectively; see Point 3 of the Introduction.

2.2.1 The Fundamental DGEMM Kernel Building Block and Hardware Streaming

If we use simple SB format we would need mb rows of A^T and nb columns of B and C to execute the fundamental building block. This would require mb + 2nb stride one streams of matrix data to be present and working during the execution of a single building block. Many architectures do *n*ot possess special hardware to support this number of streams. Now the minimum number of streams is three; one each for matrix operands A, B and C. Is three possible? An answer emerges if one

is willing to change the data structure away from simple SB order. We prefer the new data structures to consist of submatrices (register blocks); this is another application of the AA approach.

In Figure 1 we describe a data layout of a fundamental register block computation. Initially, a register block of C is placed in $\mathtt{mb} \times \mathtt{nb}$ floating point registers $\mathtt{T}(0:\mathtt{mb}-1, 0:\mathtt{nb}-1)$. An inner `do loop` on `l=0:K-1,kb` consists of performing kb sets of $mb \times nb$ independent dot products on `T`. For a given single value of `l` vectors $\mathtt{u}, \mathtt{v}$ of lengths $\mathtt{mb}, \mathtt{nb}$ from A and B respectively are used to update $T = T - uv^T$. This update is a `DAXPY` outer product update. However, and this is important, since the `T`'s are in registers there are *no* loads and stores of the `T`'s. The entire update is $\mathtt{T} = \mathtt{T} - \mathtt{A}^{\mathtt{T}}(0:\mathtt{K}-1, \mathtt{i}:\mathtt{i}+\mathtt{mb}-1) \times \mathtt{B}(0:\mathtt{K}-1, \mathtt{j}:\mathtt{j}+\mathtt{nb}-1)$. If A and B were simple SB's we would need to access vectors $\mathtt{u}, \mathtt{v}$ with stride `NB` and also there would be $\mathtt{mb}+\mathtt{nb}$ streams. Luckily, if we transpose $\mathtt{K} \times \mathtt{mb}$ A^T and $\mathtt{K} \times \mathtt{nb}$ B we will simultaneously access $\mathtt{u}, \mathtt{v}$ stride one, just get two streams and be able to address A, B in the standard way. These two transpositions accomplishes a matrix data rearrangement that allows for an excellent L1 / L0 interface of matrix data for the `DGEMM` kernel fundamental building block computation. Sections 2.1 and 2.1.1 cover an application of the AA approach and Points 2, 11, 14 and 15 of the Abstract.

3 SBP and RFP Formats for Symmetric/Triangular Arrays

SBP format emerges from applications of the AA approach. Point 3 of the Abstract gives one a free parameter `NB` to describe DLAFAs. Unlike standard 2-D arrays contiguous SBs of order `NB` map optimally into a L1 cache [37]. A drawback of SBP format is that the MCU LAPACK library does not recognise this format. To get around this problem one can transform to SBP format, execute the DLAFA on SBF format and then transform back to standard 2-D format. In [23, 3] and elsewhere this approach is used and the performance results are better, sometimes decidedly so, despite having to incur the penalty of two data transformations. RFP format is a standard 2-D array and so Level 3 BLAS and triangular LAPACK DLAFAs work directly on this format. Again, the LAPACK library does not accept this format for its full triangular matrices. In Sections 3.3-5 we give some reasons why this would be a good idea.

```
      Packed Lower
 1
 2 11
 3 12 20
 4 13 21 28
 5 14 22 29 35
 6 15 23 30 36 41
 7 16 24 31 37 42 46
 8 17 25 32 38 43 47 50
 9 18 26 33 39 44 48 51 53
10 19 27 34 40 45 49 52 54 55

      Packed Upper

 1  2  4  7 11 16 22 29 37 46
    3  5  8 12 17 23 30 38 47
       6  9 13 18 24 31 39 48
         10 14 19 25 32 40 49
            15 20 26 33 41 50
               21 27 34 42 51
                  28 35 43 52
                     36 44 53
                        45 54
                           55
```

Figure 2: Order `n=10` Packed Format Arrays

3.1 SBP Formats Generalize Standard Full and Packed Formats

SBP formats are a generalization of packed format for triangular arrays. They are also a generalization of full format for triangular arrays. A major benefit of the SBP formats is that they allow for level 3 performance while using about half the storage of the full array cases. In packed format, the elements of a triangular matrix of order $n = 10$ would be stored as shown in Figure 2; the number in location (i, j) of Figure 2 is $a(i, j)$'s storage position in A.

For SBP formats there are two parameters, `TRANS` and `NB`, where usually $n \geq \mathtt{NB}$. For these formats, we first choose a block size, `NB`, and then we lay out the matrix elements in squares of size `NB`. Each square block can be in column-

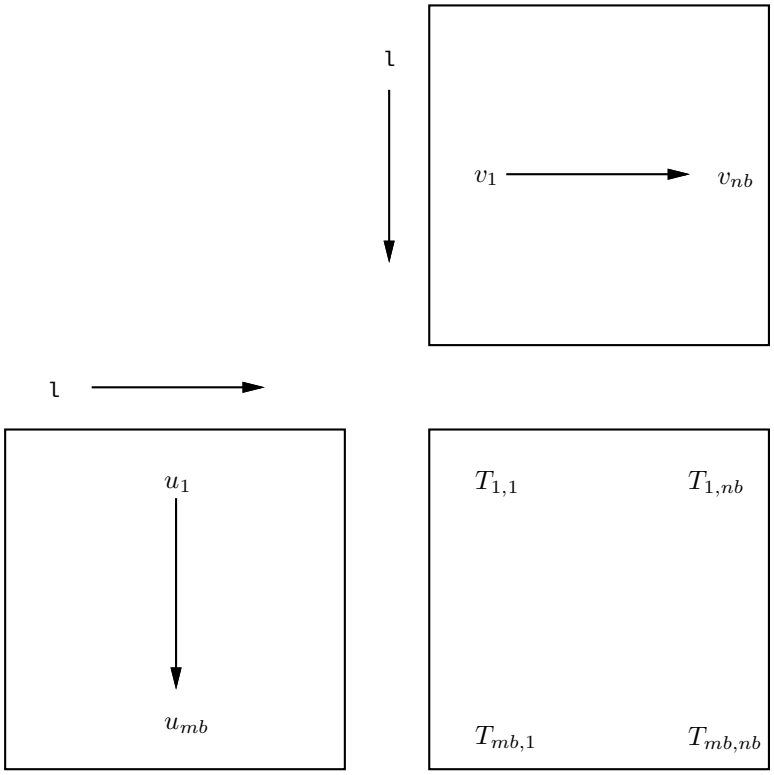


Figure 1: Fundamental GEMM Kernel Building Block.

major order (TRANS = 'N') or row-major order (TRANS = 'T'). These formats support both uplo = 'L' or 'U'. We shall only cover the case uplo = 'L'. For uplo = 'L', the first vertical stripe is n by NB and it consists of n_1 square blocks where $n_1 = \lceil n/\mathtt{NB} \rceil$. It holds the first trapezoidal n by NB part of L. Here we rename matrix A matrix L to remind the reader that our format is lower triangular. The next stripe has $n_1 - 1$ square blocks and it holds the next trapezoidal n - NB by NB part of L, and so on, until the last stripe consisting of the last leftover triangle is reached. The total number of square blocks is $n_1(n_1+1)/2$.

The top of Figure 3 gives an example of Square Blocked Packed Lower (SBPL) format with TRANS = 'T'. Here n = 10, TRANS = 'T' and NB = 4 and the numbers represent the position within the array where the matrix element $a(i,j)$ is stored. Note the missing numbers (eg, 2, 3, 4, 7, 8, and 12) which correspond to the upper right corner of the first stripe. This square blocked, lower, packed array consists of 6 SB arrays. The first three SB's hold submatrices that are 4 by 4, 4 by 4, and 2 by 4. The next two blocks hold submatrices that are 4 by 4 and 2 by 4. The last SB holds a 2 by 2 submatrix. Note that we have added padding, which we have done for ease of addressing. It is straightforward to address this set of six SB's as a composite block array.

Now we turn to full format storage. We continue the example with a matrix A of order N = 10, in an array A of LDA = 12. To get SBP format one simply sets NB = LDA = 12 and one obtains the full format array A; ie, SBP format gives a single block triangle which happens to be full format (see bottom of Figure 3). Thus, it should be clear that SBP format generalizes standard full format.

```
 1  *  *  *
 5  6  *  *
 9 10 11  *
13 14 15 16
-----------|
17 18 19 20|49  *  *  *
21 22 23 24|53 54  *  *
25 26 27 28|57 58 59  *
29 30 31 32|61 62 63 64
-----------|-----------|
33 34 35 36|65 66 67 68|81  * * *
37 38 39 40|69 70 71 72|85 86 * *
 *  *  *  *| *  *  *  *| *  * * *
 *  *  *  *| *  *  *  *| *  * * *

 1  *  *  *  *  *  *  *  *   *
 2 14  *  *  *  *  *  *  *   *
 3 15 27  *  *  *  *  *  *   *
 4 16 28 40  *  *  *  *  *   *
 5 17 29 41 53  *  *  *  *   *
 6 18 30 42 54 66  *  *  *   *
 7 19 31 43 55 67 79  *  *   *
 8 20 32 44 56 68 80 92  *   *
 9 21 33 45 57 69 81 93 105  *
10 22 34 46 58 70 82 94 106 118
 *  *  *  *  *  *  *  *  *   *
 *  *  *  *  *  *  *  *  *   *
```

Figure 3: Square Blocked Lower Packed Format for NB=4 and LDA=NB

3.1.1 Benefits of SBP Formats

We believe a main innovation in using the SBP formats is that one can translate, verbatim, standard packed or full factorization algorithms into a corresponding SBP format algorithm by replacing each reference to an i,j element by a reference to its corresponding SB submatrix. This is an easy application of Point 4 in the Introduction. Because of this storage layout, the beginning of each SB is easily located. Another key feature of using SB's is that SBP format supports Level 3 BLAS. Hence, old, packed and full codes are easily converted into square blocked, packed, level 3 code. Therefore,

```
do j = 0, n-nb, nb
  factor a(j:j+nb-1,j:j+nb-1) ! kernel routine for potrf
  do i = j + nb, n-nb, nb
    a(i:i+nb-1,j:j+nb-1) =
      a(i:i+nb-1,j:j+nb-1)*aT(j:j+nb-1,j:j+nb-1) ! BLAS trsm
  end do
  do i = j +nb, n-nb, nb  ! THE UPDATE PHASE
    a(i:i+nb-1,i:i+nb-1) = a(i:i+nb-1,i:i+nb-1) -
      a(i:i+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! BLAS syrk
    do k = i + nb, n-nb, nb ! The Schur Complement update phase
      a(k:k+nb-1,i:i+nb-1) = a(k:k+nb-1,i:i+nb-1) -
        a(k:k+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! BLAS gemm
    end do
  end do
end do
```

Figure 4: Block Version of Right Looking Algorithm for Cholesky Factorization

one keeps "standard packed or full" addressing so the library writer/user can handle his own addressing in a Fortran/C environment. Figure 4 describes a RLA for block Cholesky factorization and illustrates what we have just said. For clarity, we assume that `n` is a multiple of `nb`. Lines 2, 4, 7 and 9 of Figure 4 are calls to kernel routines.

3.1.2 Performance for SBP Cholesky

In Figure 5 the graphs plot MFlops versus matrix order N. Note that the x-axis is log scale; we let N range from 10 to 2000. The graphs compare the code of Figure 4 versus `DPOTRF`. In [23], a complete Fortran 77 code of Figure 4 is given. Data for the graphs were obtained on a 200 MHz IBM Power 3 with a peak performance of 800 MFlops. The performance of the SBP Cholesky algorithm of Figure 4 at order $N \geq 200$ is over 720 MFlops and then reaches 735 MFlops at $N = 500$. Kernel routines for Cholesky factor and three BLAS `DTRSM, DSYRK, DGEMM` were used in Figure 4. Using conventional full format LAPACK `DPOTRF` with ESSL BLAS, performance first gets to 600 MFlops at $N \geq 600$ and only reaches a peak of 620 MFlops. We do not include the cost of transforming the data format. This is perhaps unfair. Nonetheless, we did it to demonstrate what type of performance is possible. Note that the performance of Figure 4 shows some choppy behavior especially when N is small. The matrix orders where this occurs are *not* multiples of four. For example, when $N = 70$ the performance is about the same as $N = 60$. This is because Figure 4 is solving an order $N = 72$ problem, while the MFlops calculation is being done for $N = 70$. However, the kernel routines are much simpler when there is no fixup code. In the kernel codes register blocking with `mb=4`, `nb=4` and `kb=1` is used. Note that Figure 4 is always faster than `DPOTRF` by as much as a factor of four when $N = 60$ and at least 15 % for N = 2000.

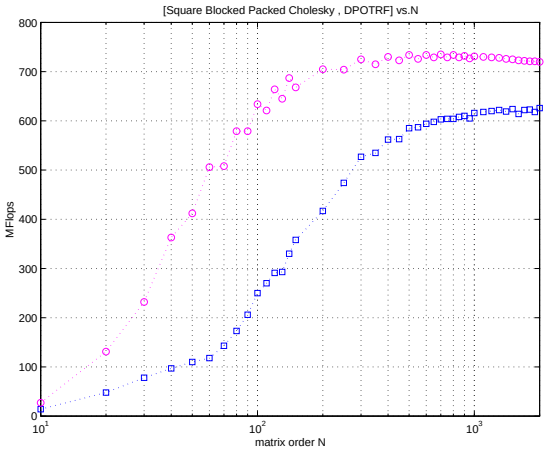


Figure 5: Performance of SBP format with `nb=88` versus LAPACK `DPOTRF`

3.1.3 Serial Cholesky Factorization Algorithms

More generally this section might be entitled Serial DLAFA's. However, our focus is on Cholesky factorization and what we say about Cholesky factorization here applies to many other DLAFA's. There are many Cholesky DLAFA's. We only mention left and right looking as well as hybrid and recursive [30, 29, 22] ones. A left, right looking algorithm does the least, most amount of computation at outer `do loop` stage `j`, respectively. The recursive algorithm uses the divide-and-conquer paradigm. The hybrid algorithm is a combination of the left and right looking algorithms. The current version of LAPACK [29], uses the hybrid algorithm. The paper [3] examines all these four algorithms using SBF, packed recursive and standard full and packed formats. Performance studies on six platforms, Alpha, IBM P4, Intel x86, Itanium, SGI and SUN were made. Overall, the hybrid algorithm using SBF was best. However, it was not a clear winner.

3.2 Data Copy of RLA's can be O(N^2)

The result we now give holds generally for RLAs for DLAFAs. And similar results hold for Left Looking Algorithms (LLAs). Here we shall be content with demonstrating that the Cholesky RLA on SBPF can be done by only using O(N^2) data copies.

In Figure 4 the O(N^3) part of the RLA has to do with the SCU; ie, the inner `DGEMM do loop` over variable `k`. We assume each call to `DGEMM` will do data copy on each of its three operands A, B and C. Now the number of C SB's that get SCUed over the entire RLA is $n_1(n_1-1)(n_1-2)/2$ where $n_1 = \lceil \mathtt{N}/\mathtt{NB} \rceil$ and `N` is the order of A. It is therfore clear that O(N^3) data copies will occur.

In Section 2.2.1 we indicated that it is now usually necessary to reformat each SB every time `DGEMM` is called if simple SB's are used. We now demonstrate that we can reduce this data copy cost to O(N^2). What we intend to do is to store the C operands of `DGEMM` in the register block format that was indicated in Section 2.2.1. Hence, the format of these C operands is then fixed throughout the algorithm of Figure 4 and no additional data copy occurs for them during the entire execution of the RLA of Figure 4. And clearly, an initial formatting cost, if necessary, is only O(N^2). Now we examine the A and B operands of the SCU for the outer loop variable `j`. SB's $A(\mathtt{j}:\mathtt{n_1},\mathtt{j})$ whose total is $\mathtt{n_1}-\mathtt{j}$ are needed for the SCU as they constitute all the A, B operands of the SCU at iteration `j`. Summing from `j=1` to $\mathtt{j} = \mathtt{n_1}$ we find just $\mathtt{n_1}(\mathtt{n_1}-1)/2$ SB's in all that need reformatting (data copying) over the course of the entire RLA of Figure 4. And since there are both A and B operands we may have to double this amount to $n_1(n_1-1)$ SB's. However, in either case this amount of data copy is clearly O(N^2).

3.3 Cholesky Factorization using Rectangular Full Packed Format.

RFP format is a standard full array of size `NT=n(n+1)/2` that holds a symmetric / triangular matrix A of order n. It is closely related to HFP format, see [20], which represents A as the concatenation of two standard full arrays whose total size is also `NT`. A basic simple idea leads to both formats. Let A be an order n symmetric matrix. Break A into a block 2×2 form

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} \qquad (1)$$

where A_{11} and A_{22} are symmetric. Clearly, we need only store the lower triangles of A_{11} and A_{22} as well as the full matrix A_{21}. When $n = 2k$ is even, the lower triangle of A_{11} and the upper triangle of A_{22}^T can be concatenated together along their main diagonals into an $(k+1) \times k$ dense matrix. This last operation is the crux of the basic simple idea. The off-diagonal block A_{21} is $k \times k$, and so it can be appended below the $(k+1) \times k$ dense matrix. Thus, the lower triangle of A can be stored as a single $(n+1) \times k$ dense matrix AR. In effect, each block matrix A_{11}, A_{21} and A_{22} is now stored in "full format". This means all entries of AR can be accessed with constant row and column strides. So, the full power of LAPACK's use of Level 3 BLAS are now available for symmetric and triangular computations. Additionally, one is using the minimal amount of storage. Finally, AR^T which is $k \times (n+1)$ has these same two desirable properties; see Figure 6, where $n = 10$. In Figure 6 we have added vertical `|`'s to try to visually delineate triangles `T1,T2` representing lower, upper triangles of A_{11} , A_{22}^T respectively and square or near square `S1` representing matrix A_{21}. The elements of A are represented

using 0 indexing. Now A has a block 2×2 form Cholesky factorization

$$LL^T = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \quad (2)$$

where L_{11} and L_{22} are lower triangular. Equation 2 is the basis of a Simple Related Partition Algorithm (SRPA) on RFP format. We now illustrate this by using existing LAPACK routines and Level 3 BLAS. The SRPA with partition sizes k and k and $n = 2k$ is: (see equations 1, 2 and Figure 6).

1. factor $L_{11}L_{11}^T = A_{11}$
   `call dpotrf('L',k,AR(1,0), n+1,info)`
2. solve $L_{21}L_{11}^T = A_{21}$
   `call dtrsm('R','L','T','N',k,k, one,AR(1,0),n+1,AR(k+1,0),n+1)`
3. update $A_{22}^T \leftarrow A_{22}^T - L_{21}L_{11}^T$
   `call dsyrk('U','N',k,k,-one, AR(k+1,0),n+1,one,AR(0,0),n+1)`
4. factor $U_{22}^T U_{22} = A_{22}^T$
   `call dpotrf('U',k,AR(0,0), n+1,info)`

This covers RFP format when `uplo` = 'L' and n is even. A similar result holds for n odd. We let $n_1 = \lceil n/2 \rceil$ and $n_2 = \lfloor n/2 \rfloor$ so that $n_1 + n_2 = n$ and $n_1 = n_2 + 1$ and symmetric matrices A_{11} and A_{22} have orders n_1 and n_2 respectively. Again, equation 1 applies. The lower triangle of A_{11} and the upper triangle of A_{22}^T can be concatenated together along their main diagonals into an order n_1 dense matrix. The off-diagonal block A_{21} is $n_2 \times n_1$, and so it can be appended below the order n_1 dense matrix. Thus, the lower triangle of A can be stored as a single $n \times n_1$ dense matrix AR. As before, AR and AR^T have the same two desirable properties; see Figure 6 where $n = 9$. Again, equation 2 is the basis of a SRPA on RFP format. This time the SRPA partition sizes are n_1 and n_2: (see equations 1, 2 and Figure 6).

1. factor $L_{11}L_{11}^T = A_{11}$
   `call dpotrf('L',n1,AR(0,0), n,info)`
2. solve $L_{21}L_{11}^T = A_{21}$
   `call dtrsm('R','L','T','N',n2, n1,one,AR(0,0),n,AR(n1,0),n)`
3. update $A_{22}^T \leftarrow A_{22}^T - L_{21}L_{11}^T$
   `call dsyrk('U','N',n2,n1,-one, AR(n1,0),n,one,AR(0,1),n)`
4. factor $U_{22}^T U_{22} = A_{22}^T$
   `call dpotrf('U',n2,AR(0,1), n,info)`

Also, for `uplo` = 'U' and n even similar results hold. To see this, break an order n symmetric matrix A into a block 2×2 form

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} \quad (3)$$

where A_{11} and A_{22} are symmetric. We need only store the upper triangles of A_{11} and A_{22} as well as the full matrix A_{12}. When $n = 2k$ is even, the lower triangle of A_{11}^T and the upper triangle of A_{22} can be concatenated together along their main diagonals into an $(k+1) \times k$ dense matrix. The off-diagonal block A_{12} is $k \times k$, and so it can be appended above the $(k+1) \times k$ dense matrix. Thus, the upper triangle of A can be stored as a single $(n+1) \times k$ dense matrix AR. As before, each block matrix A_{11}, A_{12} and A_{22} is now stored in "full format", meaning its entries can be accessed with constant row and column strides. Note that AR^T which is $k \times (n+1)$ also has these two desirable properties; see Figure 7 where $n = 10$. In Figure 7 we have added vertical `|`'s to try to visually delineate triangles `T1,T2` representing lower, upper triangles of A_{11}^T , A_{22} respectively and square or near square `S1` representing matrix A_{12}. The elements of A are represented using 0 indexing. This time A has a block 2×2 form Cholesky factorization

$$U^T U = \begin{bmatrix} U_{11}^T & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \quad (4)$$

where U_{11} and U_{22} are upper triangular. We now give the SRPA that equation 4 generates: (see equations 3, 4 and Figure 7).

1. factor $L_{11}L_{11}^T = A_{11}^T$
   `call dpotrf('L',k,AR(k+1,0), n+1,info)`

```
     LRFP AR                 LRFP AR transpose
00|55 65 75 85           00 10 20 30 40|50 60 70 80
10 11|66 76 86           55|11 21 31 41|51 61 71 81
20 21 22|77 87           65 66|22 32 42|52 62 72 82
30 31 32 33|88           75 76 77|33 43|53 63 73 83
40 41 42 43 44           85 86 87 88|44|54 64 74 84
50 51 52 53 54
60 61 62 63 64
70 71 72 73 74
80 81 82 83 84

     LRFP AR                 LRFP AR transpose
55 65 75 85 95        55|00 10 20 30 40|50 60 70 80 90
00|66 76 86 96        65 66|11 21 31 41|51 61 71 81 91
10 11|77 87 97        75 76 77|22 32 42|52 62 72 82 92
20 21 22|88 98        85 86 87 88|33 43|53 63 73 83 93
30 31 32 33|99        95 96 97 98 99|44|54 64 74 84 94
40 41 42 43 44
50 51 52 53 54
60 61 62 63 64
70 71 72 73 74
80 81 82 83 84
90 91 92 93 94
```

Figure 6: Lower Rectangular Full Packed formats when $n = 9, 10$, `LDAR = n,n+1`

```
     URFP AR                 URFP AR transpose
04 05 06 07 08           04 14 24 34|44|00 01 02 03
14 15 16 17 18           05 15 25 35|45 55|11 12 13
24 25 26 27 28           06 16 26 36|46 56 66|22 23
34 35 36 37 38           07 17 27 37|47 57 67 77|33
44 45 46 47 48           08 18 28 38|48 58 68 78 88
00|55 56 57 58
01 11|66 67 68
02 12 22|77 78
03 13 23 33|88

     URFP AR                 URFP AR transpose
05 06 07 08 09        05 15 25 35 45|55|00 01 02 03 04
15 16 17 18 19        06 16 26 36 46|56 66|11 12 13 14
25 26 27 28 29        07 17 27 37 47|57 67 77|22 23 24
35 36 37 38 39        08 18 28 38 48|58 68 78 88|33 34
45 46 47 48 49        09 19 29 39 49|59 69 79 89 99|44
55 56 57 58 59
00|66 67 68 69
01 11|77 78 79
02 12 22|88 89
03 13 23 33|99
04 14 24 34 44
```

Figure 7: Upper Rectangular Full Packed formats when $n = 9, 10$, `LDAR = n,n+1`

2. solve $L_{11}U_{12} = A_{12}$

```
call dtrsm('L','L','N','N',k,k,
one,AR(k+1,0),n+1,AR(0,0),n+1)
```

3. update $A_{22} \leftarrow A_{22} - U_{12}^T U_{12}$

```
call dsyrk('U','T',k,k,-one,
AR(0,0),n+1,one,AR(k,0),n+1)
```

4. factor $U_{22}^T U_{22} = A_{22}$

```
call dpotrf('U',k,AR(k,0),
n+1,info)
```

This covers RFP format when `uplo` = 'U' and n is even. A similar result holds for n odd. We let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$ so that $n_1 + n_2 = n$ and $n_2 = n_1 + 1$ and symmetric matrices A_{11} and A_{22} have orders n_1 and n_2 respectively. Again, equation 3 applies. The lower triangle of A_{11}^T and the upper triangle of A_{22} can be concatenated together along their main diagonals into an order n_2 dense matrix. The off-diagonal block A_{12} is $n_1 \times n_2$, and so it can be appended above the order n_2 dense matrix. Thus, the upper triangle of A can be stored as a single $n \times n_2$ dense matrix AR. As before, AR and AR^T have the same two desirable properties; see Figure 7 where $n = 9$. Again, equation 4 is the basis of a SRPA on RFP format. This time the SRPA partition sizes are n_1 and n_2: (see equations 3, 4 and Figure 7).

1. factor $L_{11}L_{11}^T = A_{11}^T$

```
call dpotrf('L',n1,AR(n2,0),
n,info)
```

2. solve $L_{11}U_{12} = A_{12}$

```
call dtrsm('L','L','N','N',n1,
n2,one,AR(n2,0),n,AR(0,0),n)
```

3. update $A_{22} \leftarrow A_{22} - U_{12}^T U_{12}$

```
call dsyrk('U','T',n2,n1,-one,
AR(0,0),n,one,AR(n1,0),n)
```

4. factor $U_{22}^T U_{22} = A_{22}$

```
call dpotrf('U',n2,AR(n1,0),
n,info)
```

We close the Section by giving the above four SRPA's on $B = AR^T$. To help understand how these four SRPA's are arrived at note that transposition changes lower to upper and vice versa and left to right and vice versa. For $n = 2k$ and equation 2 we get

```
call dpotrf('U',k,B(0,1),k,info)
call dtrsm('L','U','T','N',k,k,one,\\
           B(0,1),k,B(0,k+1),k)
call dsyrk('L','T',k,k,-one,       \\
          B(0,k+1),k,one,B(0,0),k)
call dpotrf('L',k,B(0,0),k,info)
```

For n odd, $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$ and symmetric matrices A_{11} and A_{22} having orders n_1 and n_2 we get with equation 2

```
call dpotrf('U',n1,B(0,0),n1,info)
call dtrsm('L','U','T','N',n1,n2, \\
          one,B(0,0),n1,B(0,n1),n1)
call dsyrk('L','T',n2,n1,-one,     \\
          B(0,n1),n1,one,B(1,0),n1)
call dpotrf('L',n2,B(1,0),n1,info)
```

We shall give more details in just this one case. Apply transposition to equation 1 with $n = n_1 + n_2$ odd to get equation 3 with $n = n_2 + n_1$; ie, the roles of n_1, n_2 interchange. Now we have A_{11}, A_{22} upper, lower respectively and due to symmetry $A_{12} = A_{21}^T$. Also, due to symmetry and the nature of Cholesky factorization, $U_{12} = L_{21}^T$. Using these facts, we get below:

1. factor $U_{11}^T U_{11} = A_{11}^T$
2. solve $U_{11}^T U_{12} = A_{21}^T$
3. update $A_{22} \leftarrow A_{22} - L_{21}^T L_{21}$
4. factor $L_{22}L_{22}^T = A_{22}^T$

For $n = 2k$ and equation 4 we get

```
call dpotrf('U',k,B(0,k+1),k,info)
call dtrsm('R','U','N','N',k,k,one,\\
          B(0,k+1),k,B(0,0),k)
call dsyrk('L','N',k,k,-one,       \\
          B(0,0),k,one,B(0,k),k)
call dpotrf('L',k,B(0,k),k,info)
```

For n odd, $n_1 = \lfloor n/2 \rfloor$, $n_2 = \lceil n/2 \rceil$ and symmetric matrices A_{11} and A_{22} having orders n_1 and n_2 we get with equation 4

```
call dpotrf('U',n1,B(0,n2),n2,info)
call dtrsm('R','U','N','N',n2,n1, \\
          one,B(0,n2),n2,B(0,0),n2)
call dsyrk('L','N',n2,n1,-one,     \\
          B(0,0),n2,one,B(0,n1),n2)
call dpotrf('L',n2,B(0,n1),n2,info)
```

3.4 Performance of RFP

We now consider performance aspects of using RFP format in the context of using LAPACK routines on triangular matrices stored in RPF format. Let X be a Level 3 LAPACK routine that operates either on standard packed or full format. X has a full L3 LAPACK block algorithm, call it FX. Write a SRPA with partition sizes $n1$ and $n2$. Apply the new SRPA on the new RPF data structure. The new SRPA almost always has four major steps consisting entirely of calls to existing full format LAPACK routines in two steps and calls to Level 3 BLAS in the remaining two steps:

```
FX('L',n1,T1,lda)
BLAS3(n1,n2,'L',T1,lda,S,lda)
BLAS3(n1,n2,S,lda,'U',T2,lda)
FX('U',n2,T2,lda)
```

The SRPA of Section 3.2 operating on RFP format should perform about the same as the corresponding full format LAPACK routines. This is because both the SRPA code and the corresponding LAPACK code is nearly the same and both data formats are full format. Also the SRPA code should outperform the corresponding LAPACK packed code by about the same margin as does the corresponding LAPACK full code. We give performance results for the IBM Power 4 Processor. The gain of full code over packed code is anywhere from roughly a factor of one to a factor of seven.

There are two performance graphs, one for Cholesky factor, suffix C and the other for Cholesky inverse, suffix I; see Figure 8. Before continuing we need to give the SRPA for Cholesky inverse. In equation 2 we gave the block algorithm for computing the lower Cholesky factor L from A. LAPACK computes A^{-1} in two steps: (1) compute L^{-1} using `dtrtri`; (2) compute $A^{-1} = L^{-T}L^{-1}$ using `dlauum`. So, there are actually two SRPA's, one for `dtrtri` and one for `dlauum`. The SRPA for `dtrtri` is based on the following equation 5

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}^{-1} = \begin{bmatrix} L_{11}^{-1} & 0 \\ -L_{22}^{-1}L_{21}L_{11}^{-1} & L_{22}^{-1} \end{bmatrix} \quad (5)$$

The SRPA for `dtrtri` follows:

```
call dtrtri('L','N',k,AR(1,0),n+1,info)
call dtrmm('R','L','N','N',k,k,-one,\\
           AR(1,0),n+1,AR(k+1,0),n+1)
call dtrtri('U','N',k,AR(0,0),n+1,info)
call dtrmm('L','U','T','N',k,k,one,\\
           AR(0,0),n+1,AR(k+1,0),n+1)
```

Next the SRPA for `dlauum` is based on the following equation 6

$$\begin{bmatrix} W_{11}^T & W_{21}^T \\ 0 & W_{22}^T \end{bmatrix} \begin{bmatrix} W_{11} & 0 \\ W_{21} & W_{22} \end{bmatrix} = \begin{bmatrix} W_{11}^T W_{11} + W_{21}^T W_{21} & 0 \\ W_{22}^T W_{21} & W_{22}^T W_{22} \end{bmatrix} \quad (6)$$

where $W = L^{-1}$. The SRPA for `dlauum` follows:

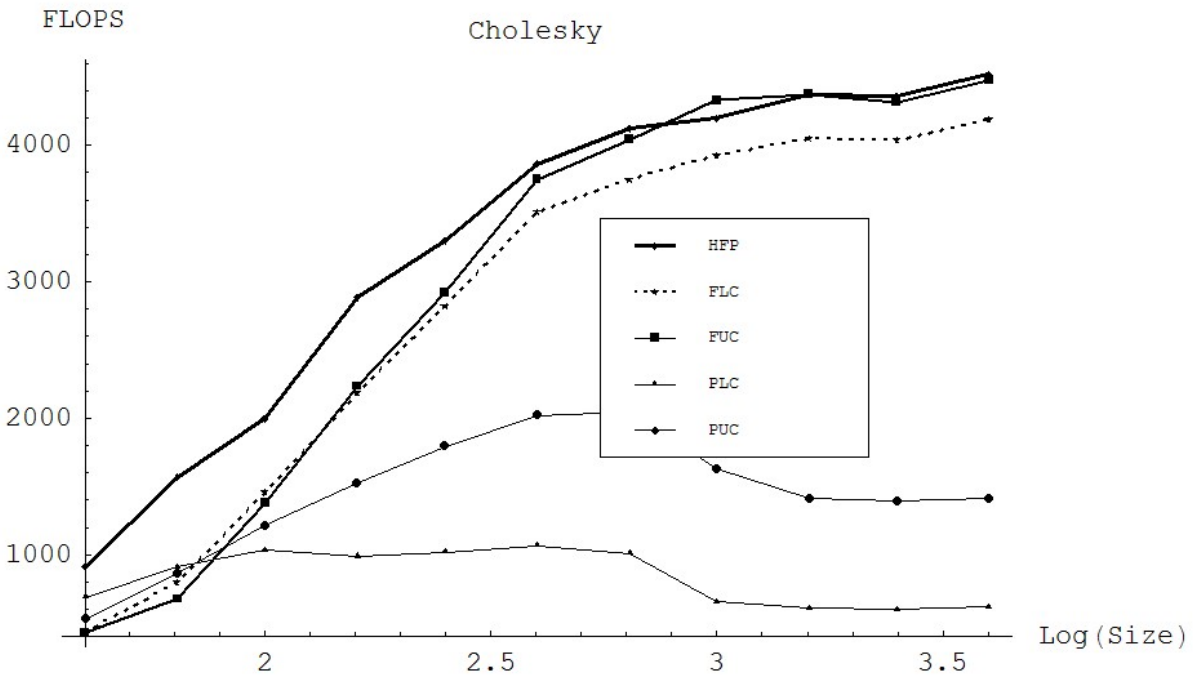
```
call dlauum('L',k,AR(1,0),n+1,info)
call dsyrk('L','N',k,k,one, \\
     AR(k+1,0),n+1,one,AR(1,0),n+1)
call dtrmm('L','U','T','N',k,k, \\
     one,AR(0,0),n+1,AR(k+1,0),n+1)
call dlauum('U',k,AR(0,0),n+1,info)
```

For both of the above two SRPA's we have assumed $n = 2k$ is even. Of course, similar SRPA's are obtainable for n being odd.

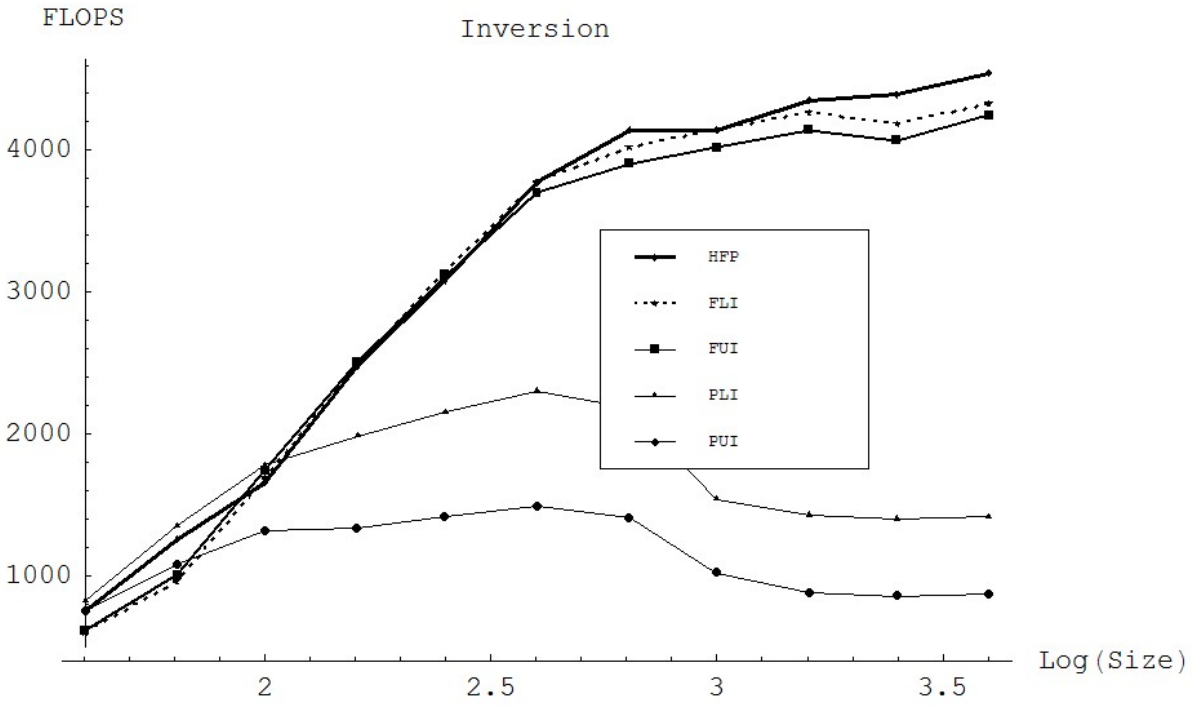
For each graph in Figure 8 we give four curves for LAPACK called FL, FU, PL, PU corresponding to Full,Uplo='L', Full,Uplo='U', Packed,Uplo='L', and Packed,Uplo='U' and a single curve RFP corresponding to SRPA. This is because the SRPA replaces each of these four LAPACK subroutines. Actually, we ran the SRPA routine four times and averaged their times. Performance is given in MFlops. We chose the order N of the matrices to follow a base 10 log distrbution. The values chosen were N = 40, 64, 100, 160, 250, 400, 640, 1000, 1600, 2500, and 4000. The corresponding log N values are 1.60, 1.81, 2, 2.20 2.40, 2.60, 2.81, 3, 3.20. 3.40, and 3.60.

As can be seen from Graph 1, SRPA performance is greater than FLC, PLC, and PUC performance. The suffix C stands for Cholesky. Also, SRPA performance is greater than FUC performance except at N = 1000 where FUC is 3 % faster. For graph two, with N $\geq$ 400 the SRPA curve is faster than the other four curves. For N $\leq$ 250 the performance ratios range from .92 to 2.18 (see Figure 8). Returning to Graph 1, we see that SRPA is 1.33 to 7.35 times faster than PLC and 1.64 to 3.20 times faster than PUC. Similarly, for Graph 2, SRPA is .92 to 3.21 times faster than PLI and 1.01 to 5.24 times faster than PUI. The suffix I stands for Inverse.

In Figure 9 we give performance ratios of RFP to the four LAPACK routines, FL, FU, PL, and



(a)



(b)

Figure 8: Absolute performance of algorithms (a) Cholesky Factorization. (b) Inversion.

N	40	64	100	160	250	400	640	1000	1600	2500	4000
RFP	914	1568	1994	2883	3302	3861	4123	4198	4371	4358	4520
FLC	2.12	1.97	1.37	1.32	1.17	1.10	1.10	1.07	1.08	1.08	1.08
FUC	2.14	2.32	1.45	1.29	1.13	1.03	1.02	.97	1.00	1.01	1.01
PLC	1.33	1.72	1.93	2.92	3.24	3.62	4.08	6.42	7.18	7.30	7.35
PUC	1.73	1.82	1.64	1.89	1.84	1.91	2.02	2.58	3.10	3.13	3.20
RFP	755	1255	1656	2481	3081	3775	4141	4141	4351	4394	4544
FLI	1.26	1.31	.98	1.00	.98	1.00	1.03	1.00	1.02	1.05	1.05
FUI	1.23	1.25	.95	.99	.99	1.02	1.06	1.03	1.05	1.08	1.07
PLI	.92	.93	.93	1.25	1.43	1.64	1.90	2.69	3.05	3.14	3.21
PUI	1.01	1.16	1.26	1.86	2.18	2.54	2.94	4.06	4.97	5.13	5.24

Figure 9: Relative Performance Comparison of Algorithms

PU. The row labeled RFP give MFlops values for that routine. To obtain the MFlops values for the other four routines, simply divide MFlops value by its associated ratio. For example, for $N = 640$, the MFlops for suffix C are 3818, 4042, 1011, 2041 and for suffix I, they are 4020, 3907, 2179, 1409.

3.5 Conclusions for RFP Format

We have described a novel data format, RFP, that can replace both standard full and packed formats for triangular and symmetric matrices. We showed that codes for the new data format RFP can be written by simply making calls to existing LAPACK routines and Level 3 BLAS. Each new SRPA operating on RFP format data replaces two corresponding LAPACK routines (four if you count dual cases of uplo='L' and 'U'). Performance on an IBM Power 4 of SRPA routines on RFP format is slightly better than LAPACK full routines while using half the storage and is roughly one to seven times faster than LAPACK packed routines while using the same storage.

We have described four distinct versions of RFP format. Two are for matrix A having full formats uplo='L' or 'U'. The other two are for the transposes of the first two. Therefore, for uplo = 'L', 'U' we have two layouts each. For codes like Cholesky factorization and triangular inverse each version should be trivial to code; ie, we just need to write SRPA's. For other codes; eg, symmetric indefinite factorization (LAPACK `DSYTRF, DSPTRF`) the AR format is probably best in terms of a coding effort. The reason is that symmetric indefinite factorization requires searching entire columns and rows of A and the `LDA` of AR is either `n+1` or `n`. Even so, this code will become intricate as it requires either one by one or two by two pivoting. In terms of RFP format this means one has to deal with boundary effects that are introduced by placing the three full arrays `T1, T2, S1` into the single full array `AR` or $\texttt{AR}^{\texttt{T}}$.

4 Distributed Memory Computing with SB and SBP Formats

DMC is concerned with a $P \times Q$ mesh of processors and with the SPMD programming model of a BCL of a global rectangular array. This mesh could be virtual or the actual hardware could be interconnected in a toroidal 2-D or 3-D array. The algorithms we describe here appear to work well on such hardwares. Now, full format symmetric and triangular arrays will waste half the storage on all processors. Using the results of Section 3.3 and additionally SBP format of Section 3.1 we can save this wasted storage if we are willing to use SB's of order $\texttt{NB}^2$. This is a natural thing to do as `NB` is the free parameter of our programming model. Let $\texttt{n}_1 = \lceil \texttt{N}/\texttt{NB} \rceil$ where `N` is the order of our global symmetric matrix A. For the time being we shall be concerned with A being a matrix of SB's of block order n_1. Now P and Q *can* have nothing to do with n_1. This fact is an additional reason why we should treat each SB of A as an *atomic unit* because when P and Q are relatively prime the SB's of A will move about the processors as single contiguous blocks. However, this separation of the SB's into single SB's will only

occur for symmetric or triangular matrices. The reason has to do with the diagonal of A separating a BroadCast (BC) into both a horizontal (row) and a vertical (column) part. We want these SB's that move during BC's to various processors to be part of our data layout so that the Send / Receive buffers of MPI or of ScaLAPACK's BLACS [13] can be treated as contiguous blocks of storage. This allows one to avoid copying matrix data to a Send buffer and copying a Receive buffer to matrix data.

We now describe a right looking algorithm (RLA) that is especially tailored to DMC. We explain it for a global matrix A. The usual way to factor A is $A = (F_1U_1)(F_2U_2)\dots(F_{n_1}U_{n_1})$ where $U_{n_1} = I$. Here F_i, U_i are the factor and update parts at stage i of the RLA. A second equally good way has $A = (F_1)(U_1F_2)\dots(U_{n_1-1}F_{n_1})$. A benefit of the second way is that it allows one to overlap the computation of F_{i+1} with the computation of U_i; see [1]. Let processor column (pc) J hold the pivot panel (pp) of F_{i+1}. Here $0 \le J < Q$. Now $pc(J)$ will do four computations: update $pp(F_{i+1})$ with U_i, factor $pp(F_{i+1})$, Send or broadcast $pp(F_{i+1:n_1})$ to all other $pc(K)$, $0 \le K < Q$ and finally will update its remaining column panels. Simultaneously, the remaining $pc(K), K \ne J$ will just update all of their column panels.

The RLA above can be expressed as three separate sub algorithms: factor F_{i+1}, Send / Receive $F_{i+1:n_1}$ and update on processor $p(I, J)$ for all I and J. The update algorithm is called the Schur Complement Update: Each active SB on $p(I, J)$ gets a `DGEMM` update. What is missing are the `A,B` operands of `DGEMM` for each active SB, the `C` operand of `DGEMM`. We add these `A,B` operands to our data structure by placing on each $p(I, J)$ West and South border vectors that will hold all of $p(I, J)$ SB `A,B` operands. These borders, now part of our data layout, are the Send / Receive buffers referred to above. Now `SB(il,jl)=SB(il,jl)-W(il)*S(jl)` becomes the generic `DGEMM` update on $p(I, J)$ where `il,jl` are local coordinates on $p(I, J)$. Thus, the Schur Complement update is just a sum of `DGEMM's` over all active SB's and our DMC paradigm guarantees that almost perfect load balance will occur.

There are two DMC SBP Cholesky factorization algorithms that emerge for A. One is based on using a BCL of RFP format. Since AR is rectangular, it should not be hard to see that each $p(I, J)$ will hold a rectangular matrix. The layout is made up of pieces of two triangles `T1, T2` and a square `S1` that make up global AR; see Section 3.3 or [20] for the meaning of `T1, S1, T2`. In the layout the SB's of `T2` are reflected in `T2`'s main diagonal. We introduce North and East border vectors to hold the `A,B` operands of `T2`'s reflected SB's. The coding becomes intricate because AR consists of three distinct pieces of A.

The second algorithm is simpler to code because we have found a novel way to layout A. Global A consists of `NT1` = $n_1(n_1 + 1)/2$ SB's. Because we have made each SB atomic we may layout A on the $P \times Q$ mesh in the standard manner. The processor $p(I, J)$ will hold a quasi lower triangular matrix. We represent it as a one dimensional array of SB's along with a Column Pointer (`CP`) array that points at the first block in local column `jl`, $0 \le$ `jl` $\le$ `qe`(`J`); `qe(J)` is defined in Section 4.2. It turns out that row indices are *not* required. This is because the last row index on each $p(I, J)$ for $0 \le J < Q$ is the same for each I.

This paper describes DMC for Symmetric/Triangular matrices. However, it should be clear that our paradigm of using SB's works for rectangular matrices as well. Thus, most of ScaLAPACK's factorization codes, eg. $LU = PA$ and $QR = A$, also work under this paradigm. Our paradigm is based on the second view point of Section 2. What we have done is to isolate the major matrix multiply part of our DMC algorithm and to relate it to the Schur complement update. We have introduced West and South border vectors to our data layout. In so doing we have produced a way to code ScaLAPACK type algorithms without a PBLAS layer. However, this is not to say that we should avoid the PBLAS layer.

4.1 Overview of the SBPL DMC Algorithm

We shall give a complete overview of an example problem and this description is general. We need to introduce some notation. Upper case letters (eg `I, J`) represent the processors and the lower case letters (eg `jp`) represent block indices. Thus, `p(I,J)` will denote the process or processor at row `I` and column `J` of a processor mesh. We assume there are `P` process rows and `Q` process

columns, so $0 \le I < P$ and $0 \le J < Q$. Lower case letters (eg, i,j,k,l) denote indices or subscripts of the blocks that make up a given block matrix. There are both global indices and local indices.

Figure 10 depicts a Block Packed Global (BPG) matrix ABPG of block order n = 18. Note that we are using zero indexing; ie, the row and columns of our matrices and arrays are labeled $0, \ldots, n-1$. Note that in Section 4 we used n_1 to denote the n we are using from now on. Matrix ABPG will be laid out on a P=5 by Q=3 processor mesh. In the three left columns, we give the mesh R(ow) label, local row label, and global row label of the n rows of ABPG (letters a, b to h stand for the numbers 10, 11 to 17). These row labels are the row indices of the NT=n(n+1)/2=171 NB by NB blocks making up ABPG. The last three rows give global column labels, local column labels, and mesh column labels of the column indices of ABPG.

In Figure 11 we give the Block Packed Cyclic (BPC) layout of Figure 10 whichs gives array block packed cyclic (abpc). We directly map ABPG onto the P by Q process mesh. We shall need the W and S border block vectors (buffers) to hold the Scaled Pivot Blocks (SPB's). These block vectors r,c while hold the A and B operands of GEMM. These two borders vectors contain the jp=3 SPB blocks of ABPG(jp+1:n-1,jp). There are n-jp-1=14 SPB blocks. The jp = 3 pivot column J=0 lies on p(0:4,J) and the pivot block (jp,jp) is on p(3,J). These SPB blocks reside on local column jl=1 of p(0:4,J). Later, we shall describe in general how ABPG(jp+1:n-1,jp) is BC from p(0:P-1,J) to all SPB buffers on p(0:P-1,0:Q-1). We continue with Figure 11. The local indices of abpc are given by a one dimensional addressing scheme plus a set of column pointers. For example, on p(1,2) there are five local columns (0:4) that have 3, 3, 2, 2, 1 blocks, respectively. These columns start at addresses 0, 3, 6, 8, 10 of a vector (one dimensional packed block format) of abpc blocks whose length is 11. Now, bl(9) on p(1,2) is the second block of column 3. This block is bl(g,b) of Figure 10.

In Figure 11 the processes are considered as occurring on process rows identified from 0 to 4 and process columns identified from 0 to 2. The processor memory contents after the cyclic distribution is represented by the global lower block indices inside the $mp = \lceil n/P \rceil = 4$ by $np = \lceil n/Q \rceil = 6$ rectangle that just contain a quasi lower packed array of blocks. Also, the left column block vector and the bottom block row vector represent the send buffers for the processor. This figure represents the condition for which jp = 3, wherein the column on the left and the row at bottom are in receipt of a row BC from the zeroth process column (eg, J=0). The asterisks in the block border vectors (eg, " ** ") represent positions in which nothing is presently occurring.

In general, the west and south boundaries in each process will hold the SPB's and active SB's residing to the north and east of these buffers will get updated by them during a SCU..

The conventional RLA for execution by blocks of data was given in Figure 4 of Section 3.1. For the distributed memory version, after the ABPG is distributed onto a P by Q mesh in block cyclic fashion, the RLA is carried out with a slight modification; see Figure 12. In Section 4 we gave a reason to prefer the global algorithm of Figure 12 over the global algorithm of Figure 4 for the case of DMC. The reason was to overlap the factor part of pivot panel j with the SCU of stage j-1. Clearly, this is what Figure 12 is doing in its outer do loop j=1,n. Note that steps 2 and 3 of the elseif clause forces one to use two border column and row block vectors as buffers. For clarity, we disregard this technicality in what follows. Figure 12 can be structured into three sub algorithms:

- Factor a Pivot Panel
- Send and Receive the Pivot Panel
- Perform a Schur Complement Update

For factoring a pivot panel, the following steps are executed; see Figure 13. The pivot panel j is on some column J, p(0:P-1,J), of the mesh. This group of P processors update, (step 1.), pivot panel j via a SCU of this single panel. In step 2., p(I,J) does a NS BC of the global block bl(j,j), (pivot), to all p(0:P-1,J). Here I is the row processor of the group on which bl(j,j) resides. In step 3. all p(0:P-1,J) factors and scales the pivot panel j. In last step 4. all p(0:P-1,J) do a row BC of the just completed SPB.

```
                    ABPG has block size order n   18
m  l  g
e  o  l
s  c  o
h  a  b
   l  a
R     l

0  0  0 | 00
         |
1  0  1 | 10 11
         |
2  0  2 | 20 21 22
         |
3  0  3 | 30 31 32 33
         |
4  0  4 | 40 41 42 43 44
         |
0  1  5 | 50 51 52 53 54 55
         |
1  1  6 | 60 61 62 63 64 65 66
         |
2  1  7 | 70 71 72 73 74 75 76 77
         |
3  1  8 | 80 81 82 83 84 85 86 87 88
         |
4  1  9 | 90 91 92 93 94 95 96 97 98 99
         |
0  2  a | a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa
         |
1  2  b | b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb
         |
2  2  c | c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc
         |
3  2  d | d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd
         |
4  2  e | e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee
         |
0  3  f | f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
         |
1  3  g | g0 g1 g2 g3 g4 g5 g6 g7 g8 g9 ga gb gc gd ge gf gg
         |
2  3  h | h0 h1 h2 h3 h4 h5 h6 h7 h8 h9 ha hb hc hd he hf hg hh
         |
-------------------------------------------------------------

global     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  g  h

local      0  0  0  1  1  1  2  2  2  3  3  3  4  4  4  5  5  5

mesh C     0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2
```

Figure 10: The Global Layout of a SBP order 18 matrix

Lower Block Packed Cyclic layout(P,Q)= 5 3

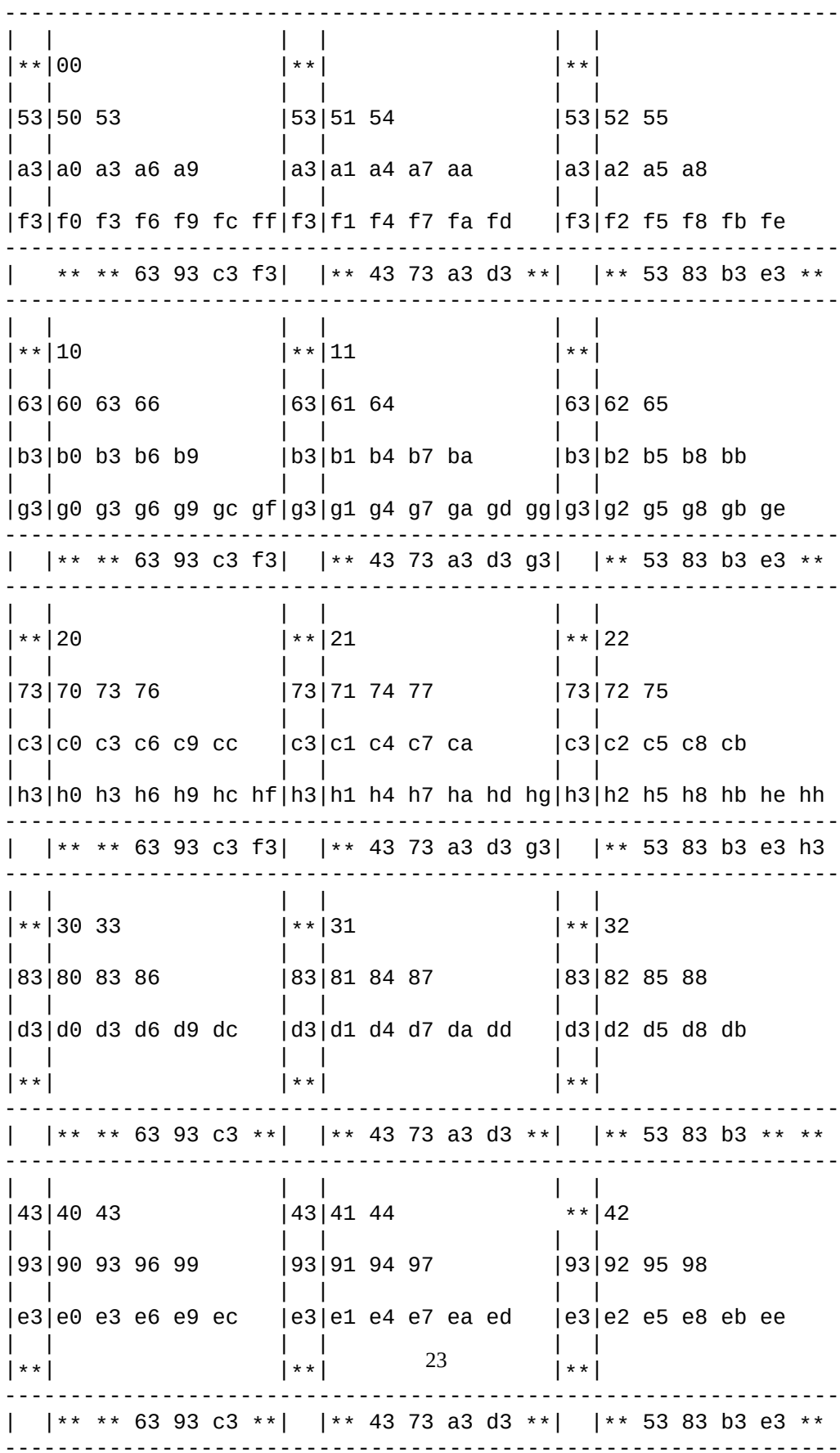


Figure 11: 5 by 3 Lower Block Packed Cyclic Layout of a SBP order 18 matrix

```
Factor and Scale first column
  panel j=0 on p(0:P-1,0).
Row BC the completed 0-th pivot
  column panel on p(0:P-1,0).
do j = 1, n-1
  Receive scaled blocks cols of
    previous pivot col j-1
    in the W and S borders.
  if(p(0:P-1,J) does NOT hold
    the j-th pivot panel)then
    Update all trailing blocks
    with the W and S borders
    (holding scaled column
    panel j-1).
  elseif(p(0:P-1,J) holds the
    j-th pivot panel)then
    Update the j-th pivot col-
      umn panel, Factor it,
      and Scale it.
    Row BC the completed j-th
      pivot column panel to all
      W and S borders.
    Update the remaining trail-
      ing blocks of p(0:P-1,J)
      with the W and S borders
      (holding scaled column
      panel j-1).
  endif
enddo
```

Figure 12: Distributed Memory Cholesky Factorization

```
1. p(0:P-1,J) updates the pivot
   col j with their W,S borders
   (scaled column panel j-1).
2. p(I,J) sends pivot block
   (j,j) to all p(0:P-1,J).
3. p(0:P-1,J) factors pivot
   block (j,j) and scales pivot
   col (j+1:n-1,j)
   (column panel j).
4. p(0:P-1,J) row BC's the
   scaled pivot column
   (j+1:n-1,j).
```

Figure 13: Distributed Memory Cholesky Factorization of a Pivot Panel

```
SPB's (j+1:n-1,j) have been row
  BC (last step of Figure 11).
p(I,0:Q-1) receives all scaled
  blocks.
A block (i,j) is column BC when
  it reaches p(I,K) that is
  holding block(i,i); Row BC of
  block(i,j) stops at p(I,K)
  holding block (i,i) if i<j+q-1;
  "stopping" means that a BC is
  not done to p(I,K+1), as it
  would hold non existent block
  (i,i+1).
Col BC of block (i,j) stops at
  p(L,K) holding block (n-1,i) if
  i+p=n; "stopping" means that a
  BC is not done to p(L+1,K), as
  it would hold non existent
  block (n,i).
```

Figure 14: Distributed Memory Send / Receive of a Pivot Panel

An overview of the Send and Receives are given in Figure 14. We shall describe this overview in Section 4.1.1.

For the update of the trailing matrix see also Section 4.1.1.

4.1.1 The Schur Complement Update

The Schur complement is another name for the blocks `ABPG(jp+1,n-1,jp+1:n-1)`. These blocks are updated during the update phase. They are the `GEMM` and `SYRK` updates. Almost all of the floating point operations are done in this phase which has 100 % parallelization.

Looking at the figure 10, we see that blocks `ABPG(jp+1:n-1,jp+1:n-1)` are to be updated by the current scaled pivot blocks `ABPG(jp+1:n-1,jp)`. Let $B_{i,j}$ be any Schur complement block and $B_{i,jp}$ and $B_{j,jp}$ be its associated scaled pivot blocks. The update operation is $B_{i,j} = B_{i,j} - B_{i,jp}B_{j,jp}^T$ if are all three operand blocks are stored in column major order. Now $B_{i,jp}, i > jp$ is used to `GEMM` update SB's $B_{i,jp+1:i-1}$ as an `A` operand, SB's $B_{i+1:n-1,i}$ as a `B` operand and to `SYRK` update SB $B_{i,i}$. In all, each $B_{i,jp}, i > jp$ is used to update $n-1-jp$ times during the SCU stage jp. For example, in Figure 10, let $i = 9, jp = 3$ and note that $B_{i,jp}$

is used six times to update $B_{i,jp+1:i}$ along row i and eight times to update $B_{i+1:n-1,i}$ along column i. Now in Figure 11 $B_{i,jp}$ resides on p(I,J) where I=4,J=0. And mesh row p(I,0:Q-1) contains block row i of ABPG including diagonal block $B_{i,i}$ which is on some p(I,K), 0 $\leq$ K $<$ Q. In this case, K=0. This means mesh column p(0:P-1,K) contains column i of ABPG. These simple observations tells us that $B_{i,jp}$ only needs one row and one column BC on the P $\times$ Q mesh to fully distribute it for the DMC SCU. These remarks serve as an explanation of Figure 14.

Figure 15 overviews the SCU operation with the West and South border vectors containing the A and B operands of GEMM. In Figure 11 each p(I,J) has W and S borders that hold update operands $B_{i,jp}$ and $B_{j,jp}$ where jp=3. The inactive blocks are all $B_{i,j}$ for j $<$ jp + 1. Thus, global row i indices with i $<$ 4 and global column j indices with j $<$ 4 are inactive. For the time being, note that each active Schur complement block has two operands present for updating purposes (ie, a W and S operand which is found by projecting W and S from its (i,j) position on p(I,J).

Therefore, in summary, an update of any SB $B_{i,j}$ requires operands $B_{i,jp}$ and $B_{j,jp}$. We have assumed above that all NT blocks of ABPG have been stored in column major order. However, we can also store all of these blocks in row major order; see Section 3.1. In that case, each block is the transpose of the column major case and vice versa. This means we can use a transpose identity of GEMM; ie, C = C$-$AB^T if and only if D = D$-$E^{T}F where D, E, F are the transposes of C, B, A respectively. Here, we prefer to store the blocks in row major format because it will lead to the T,N form of GEMM as opposed to the N,T form of GEMM which one gets by using column major order.

4.2 Details of the SBPL DMC Algorithm

We now give details about the Block Packed Cyclic Lower (BPCL) algorithm. We shall not cover the upper case, as it is quite similar to the lower case. First we describe the mapping of a SBPL array to our new BPCL layout on a P by Q rectangular mesh. Before doing so, we must define our new BPCL layout on a rectangular mesh. The block order of our BPG symmetric matrix ABPG is n. On a P by Q mesh, p(I,J) gets rows I + il $*$ P, il = 0, ..., pe(I) and columns J + jl $*$ Q, jl = 0, ..., qe(J). Here pe(I) and qe(J) stand for the local end index values of il and jl. On the West border of p(I,J) we lay out pe(I)+1 send receive buffers, block vector r of row positions (row indices), and on the South border of p(I,J) we lay out qe(J)+1 send receive buffers, block vector c of column positions (column indices). The row and column positions are the same as the standard BCL indices. ABPG consists of an isosceles triangle of SB's. As a full symmetric matrix it holds n^2 SB's. Now the BCL of a full rectangular array of SB's is standard and well understood. Conceptually, the two sides of the isosceles triangle can play the role of the two sides of the rectangle in a standard BCL. Here these sides are represented by the two border vectors r and c. These two block vectors along with a one dimensional column pointer array CP, described in the next paragraph, are sufficient to determine our lower (upper) packed BCL. In Figure 11, P=5,Q=3, n=18, pe(0:4)=3,3,3,2,2 and qe(0:2)=5,5,5. The block vectors r,c will receive SPB's with row, column indices I + il $*$ P, il = 0, ..., pe(I), J + jl $*$ Q, jl = 0, ..., qe(J) respectively.

Since ABPG can be viewed as a full matrix of order n, we could layout all n^2 blocks. This is what ScaLAPACK currently does. However, we shall only layout the lower triangular blocks (eg, just the blocks of ABPG). To do this, we use a column pointer array CP(0:np,0:P-1,0:Q-1), where np = $\lceil$n/Q$\rceil$. Actually, on p(I,J)

```
do for all p(0:P-1,J), 0 = J < Q.
  Receive row and col BC scaled
  pivot col blocks (j+1:n-1,j) on
  the W and S borders.
  Do SYRK and GEMM updates on all
  trailing blocks
  (j+1,n-1,j+1,n-1) using the
  scaled pivot col blocks
  (j+1:n-1,j) on the W and S
  borders.
enddo
```

Figure 15: Distributed Memory Schur Complement Update

np is an upper bound of the size of array CP; see Section 4.2.3 and array jle for the actual size of CP on each p(I,J). On p(I,J), CP(jl,I,J) points to the first block of global column j=J+jl*Q. The column pointer array is found by counting the number of SB's of global column j of ABPG that reside on p(I,J). It is clear that the last row of column j will have process last (pl) row index pl(I) independent of the global column j and the process column J. These independent values are equal to the last row index values of the block vectors r, namely, I+pe(I)*P. For ABPG these values are 15,16,17,13,14. Using this fact and the nature of the standard block cyclic mapping, one can show that a row index array is *not* necessary. In Figure 11 , pl(I)=f,g,h,d,e. To see this, note that CP(jl,I,J) gives the first nonzero in local column jl of p(I,J). The last index in column jl is pl(I) which is pointed at by CP(jl+1,I,J)-1. There are NU=CP(jl+1,I,J)-CP(jl,I,J) elements in column jl. Let i1 be the first global row index in column jl. Then i1=pl(I)-NU*P. The NU global indices in column j are therefore $\mathtt{i1}, \mathtt{i1}+\mathtt{P}, \ldots, \mathtt{pl(I)}$ which are of course the index set of block vector r. Hence, row indices are not required.

We can now define mapping global local packed (glp) and local global packed (lgp), which are described in Figures 16 and 17, respectively. For both mappings we assume the ABPG array is SBPL triangular. In Figures 16, 17 ijp is the global packed block index of standard lower packed format and ijl is the corresponding local packed index on p(I,J). The idea behind describing both mappings is to move to full coordinates, use the standard full block cyclic maps for global local (gl) and/or local global (lg), and then move back to packed coordinates.

Looking now at the glp mapping shown in Figure 16, we start with block order n and $0 \le \mathtt{ijp} < \mathtt{NT} = \mathtt{n}(\mathtt{n}+1)/2$. From n and ijp we can compute global full coordinates (i,j) using the standard mapping from lower packed format to full format. Now we have (i,j). The standard block cyclic mapping for a rectangular global array gives il=i/P and I=i-il*P and jl=j/Q and J=j-jl*Q. Now we have the full local coordinates. Using the properties of our BPCL layout, namely arrays CP,pL, we can find the unique

1. Enter with n,ijp
2. Find (i,j) from n,ijp:
   - $\mathtt{j} = (\mathtt{n}+1/2) - [(\mathtt{n}+1/2)^2 - 2\mathtt{ijp}]^{.5}$
   - $\mathtt{i} = \mathtt{ijp} - \mathtt{j}(2\mathtt{n}-1-\mathtt{j})/2$
3. Use (i,j) to compute (il,I) and (jl,J)
   - il=i/P and I=i-il*P
   - jl=j/Q and J=j-jl*Q
4. Find ijl from il,I,jl,J,CP,pl,i; ie, $\mathtt{IND} = \mathtt{CP}(\mathtt{jl}+1) - 1 \rightarrow \mathtt{pl}(\mathtt{I})$ and ijl=IND-(pl(I)-i)/P

Figure 16: The Global Local Packed Mapping

1. Enter with I,J,ijl
2. Find jl such that $\mathtt{CP}(\mathtt{jl}) \le \mathtt{ijl} < \mathtt{CP}(\mathtt{jl}+1)$; eg, use a binary search
3. il=CP(jl+1)-1-ijl
4. Use (il,I) and (jl,J) to find (i,j); ie, i=I+il*P and j=J+jl*Q
5. Use (i,j) to compute ijp=j(2n+1-j)/2+i-j

Figure 17: The Local Global Packed Mapping

ijl associated with I,J,jl,i. See Step 4 of Figure 16 for details.

Now look at Figure 17 showing the lgp mapping. We start with I,J, and element ijl on p(I,J). Index ijl corresponds to SB(il,jl) on p(I,J). From ijl, we need to find il,jl, the full coordinates associated with our BPCL layout. In step 2, we can use a binary search to find jl, using input I,J,ijl and array CP. Knowing jl,ijl, and CP, we can find il (see step 3). In step 4, we use the standard block cyclic inverse mapping for a rectangular array to obtain the global i,j full coordinates of ABPG. Finally, in step 5, we compute ijp from i,j, using the standard map of full lower, L(i,j), to packed lower LP(ijp).

In Section 4.1 we saw that the DMC RLA for the Cholesky Factorization Algorithm of Figure 12 consisted of three sub algorithms given in Figures 13, 14, 15. In the next three sub-sections we shall give the details of these three sub algorithms. Experts can probably skip over these three sections as the information we gave about Figures 13, 14, 15 was sufficient for their complete understanding of the three sub algorithms. We are going into more detail for two reasons. First, we want an overly complete description to demonstrate how efficient this algorithm will be. Second, some of our intended audience is not performance oriented about algorithms or they are not familiar with the area of DLAFA. The very detailed descriptions to be given will serve the purpose of demonstating the efficiency and allow various readers to stop reading when their understanding is complete. We shall use the technique of a general description followed by covering a detailed example. The detailed example, with jp=3, will be relegated to the Appendix. Finally, before we move on we first we give very brief details of our programming model which is the SPMD model. This model works well on a BCL. For our application, my=p(I,J) is the unique identifier of one of the P $\times$ Q processors; ie, (I,J) is another name for this identifier.

4.2.1 Factor Pivot Panel jp Producing n-jp-1 SPB's

In Figure 18 we give explicit pseudo code for DMC of factoring a pivot panel. This code is quite simple. Every process p(I,J) has a copy of global variable jp which is the current global pivot column of Figure 18. Given jp we compute il=jp/P, pr=jp-il*P, jl=jp/Q and pc=jp-jl*Q. Since we are on p(I,J) we can find out if our I,J equals pr,pc. So, the first if-then-else clause can be executed and pb(I,J) will then hold the pivot block bl(jp,jp) for J=pc and 0 $\leq$ I $<$ P. From now on, the process column J will execute in parallel and there will be almost perfect load balance during this parallel sub-computation: First, p(0:P-1,J) factors its copy of bl(jp,jp) by calling factor kernel dpofu. This kernel is a level 3 implementation of LAPACK code DPOTF2. Now, we come to the second if-then-else clause of Figure 18. If I=pr then the factored bl(jp,jp) is copied back to abpc(ijl,I,J). This clause also locates the local starting pointer ijl on p(0:P-1,J) of the first SB that is to be scaled. Also, in parallel local end pointer ijle is set. Finally, ijle-ijl+1 SB's are scaled to become SPB's in the last parallel do loop of Figure 18. This completes the description of Figure 18. However, see the Appendix, Section 7.1, if you want to follow the above code via an example.

4.2.2 Send and Receive all SPB's to all p(0:P-1,0:Q-1)

After a pivot and scaling step completes a BC SEND RECEIVE commences on p(0:P-1,J). We now illustrate with explicit details how the SEND/RECEIVE algorithm, works on p(0:P-1,J), using Figure 19 below. Input (I,J) is the identifier of the processor. Input jls is the local starting column in apbc and c. Input ils is the local starting row in r. Input ilj is the starting index of abpc corresponding to row ils of r and column jls of c.

In Section 4.1.1 we saw that a single SPB $B_{i,jp}$ on p(I,J) needs to be row BC from p(I,J) to SB $B_{i,i}$ on p(I,K) and then column BC to SB $B_{n-1,i}$ on p(L,K). Usually these two BC's are full BC's of lengths Q-1,P-1 respectively. The row BC starts by copying SPB $B_{i,jp}$ from abpc(ijl,I,J) to r(il,I,J). The length of the row BC is determined and the BC commences. Now p(I,K) holds $B_{i,i}$. When r(il,I,J) is received on p(I,K) in r(il,I,K) it is copied to c(jl,I,K) where jl corresponds to global index i. The length of the column BC is determined and this BC is ex-

```
Input:
  I,J,jp,CP(jl:jl+1,I,J)
  abpc(CP(jl,I,J):CP(jl+1,I,J)-1,I,J)
from jp compute (il,pr),(jl,pc) ! pc=J
if(I.eq.pr)then
  ijl=CP(jl,I,J) ! -> bl(jp,jp) on P(I,J)
  pb(I,J)=abpc(ijl,I,J) ! pb is pivot send buffer on p(I,J)
  DO a NS BC from pb(I,J)
else !
  pb(I,J) receives bl(jp,jp) from pb(pr,pc)
endif
! bl(jp,jp) now resides on p(0:P-1,J) in pb(0:P-1,J)
call dpofu(pb(I,J),nb,nb,info) ! pb is now factored
ijl=CP(jl,I,J)
if(I.eq.pr)then
  abpc(ijl,I,J)=pb(I,J) ! return factored bl(jp,jp)
  ijl=ijl+1
endif
ijle=CP(jl+1,I,J)-1 ! last block to be scaled on p(I,J)
do ii=ijl,ijle
  scale abpc(ii,I,J) ! dtrsm
enddo
```

Figure 18: BPCL Pivot Panel Factoring and Scaling Pseudo Code

ecuted. This completes the description of Figure 19. However, see the Appendix, Section 7.2, if you want to follow the above code via an example.

4.2.3 Execute a Schur Complement Update on all p(0:P-1,0:Q-1).

Finally, as Figure 11 shows when `jp=3`, one is ready to perform a Schur complement update. The algorithm for doing this is given in the BPCL SCU Algorithm of Figure 20 given below.

On `p(I,J)` the basis of Figure 20 rests on the relationships between the SB's of `r,c` and the active part of the SB's of `abpc`. We have seen that the generic update formula is

$$B_{i,j} = B_{i,j} - B_{i,jp}B_{j,jp}^T \qquad (7)$$

Now, on `p(I,J)` the SB's, $B_{i,j}$, are represented by the one dimensional array of SB's `abpc` and its associated column pointer array `CP`. The SB's $B_{i,jp}, B_{j,jp}$ on `p(I,J)` are represented by the one dimensional block vectors `r,c` respectively. Given three local indices `ijl,il,jl` of `abpc,r,c` we need to relate them to the global indices of equation 7. Clearly, one could use the mappings `glp` and `lgp`. This approach has a neglicable cost. However, the cost is not zero and we wish to find a better approach with a near zero cost. On `p(I,J)` we shall have two `do loops`, the outer on `jl` and the inner on `il` as Figure 20 shows. In the outer do loop we need to know if the first SB in column `jl` is a diagonal block. We do this via a cheaper computation of the global `i,j` associated with the local `il,jl`. Our data structure makes it easy to relate `jl` with `ijl` via the use of the `CP` array. Also, the `CP` array defines a linear relation between the starting value of `il`, called `ils(jl)`, and `jl`; ie, `ils(jl)=pe(I)+1 -CP(jl+1,I,J)+CP(jl,I,J)`. See also the explanation in the second paragraph of Section 4.2. As said above, in the outer `do loop`, we compute global `i,j` as a way to ascertain whether the first SB in column `jl` is on the diagonal of `ABPG`. All other SB's in column `jl` must be `GEMM` blocks. This completes the description of Figure 20. However, see the Appendix, Section 7.3, if you want to follow the above code via an example.

```
Input:
  I,J,jp,ils,jls,ijl,pe(I),abpc(ijl:ijl+pe(I)-ils,I,J)
Output
  r(ils:pe(I),I,J),c(jls:qe(J),I,J)
Do il=ils to pe(I)
  i=I+il*P ! global i
  slr=min(i-jp,Q-1) ! length of West East BC
  abpc(ijl,I,J) to r(il,I,J) ! bl(i,jp) to W send buffer
  DO East West BC from r(il,I,J)
  r(il,I,(J+1:J+slr)mod Q) receive bl(i,jp)
  K=mod(i,Q) ! p(I,K) holds bl(i,i)
  jl=i/Q ! c(jl,I,K) will hold bl(i,jp)
  slc=min(n-i,P)-1 ! length of North South BC
  r(il,I,K) to c(jl,I,K) ! bl(i,jp) to S send buffer
  DO N S BC from c(jl,I,K)
  c(jl,(I+1:I+slc)mod P,K) receive bl(i,jp)
  ijl = ijl +1
enddo
```

Figure 19: BPCL SEND/RECEIVE Algorithm

```
Input
  jp,I,J,jls,jle(I,J),ils(jls),pe(I),CP(jls:jle(I,J)+1,I,J),
  abpc(CP(jls,I,J):CP(jle(I,J)+1)-1,I,J),
  r(ils(jls):pe(I),I,J),c(jls:jle(I,J),I,J)
Output
  abpc(CP(jls,I,J):CP(jle(I,J)+1)-1,I,J)
-------------------------------------------------------------
ijl=CP(jl,I,J) ! first active block
do jl=jls,jle(I,J)
  j=J+jl*Q ! global j index
  ils=pe(I)+1-CP(jl+1,I,J)+ijl ! local row start column jl
  i=I+ils*P ! global i index
  if(i.gt.j)then ! gemm bl(i,j)=bl(i,j)-bl(i,jp)*bl(j,jp)^t
    abpc(ijl,I,J)=abpc(ijl,I,J)-r(ils,I,J)*c(jl,I,J)^t
  else ! i=j, syrk bl(i,i)=bl(i,i)-bl(i,jp)*bl(i,jp)^t
    abpc(ijl,I,J)=abpc(ilj,I,J)-r(ils,I,J)*c(jl,I,J)t
  endif
  ijl=ijl+1 ! bump pointer to abpc
  do il=ils+1,pe(i)
    abpc(ijl,I,J)=abpc(ijl,I,J)-r(il,I,J)*c(jl,I,J)^t ! gemm
    ijl=ijl+1 ! bump pointer to abpc
  enddo
enddo
```

Figure 20: BPCL Schur Complement Update Algorithm

5 Summary and Conclusions

This paper contains three new results. RFP format, a variant of HFP format was described as a standard minimal full storage array for representing both symmetric and triangular matrices. Hence, for these types of matrices it is a replacement for both the standard formats of DLA, namely full and packed storage. It possesses three really good features: it is supported by Level 3 BLAS and LAPACK full format routines and it requires minimal storage.

Secondly, by using a combination of sub matrix partitioning and standard packed format; ie, the SBP format of Section 3.1 we have produced a near minimal storage distributed memory algorithm for Cholesky factorization on a `P` $\times$ `Q` mesh of processors. Full details were given. Also, by combining SB format and RFP format we have sketched a second such algorithm.

The final result quantifies the amount of data reformating done by right looking LAPACK factorization algorithms such as $LU = PA$ (`DGETRF`), $LL^T = A$ (`DPOTRF`) and $QR = A$ (`DGEQRF`). LAPACK factorization algorithms all call Level 3 BLAS and in particular `DGEMM`. The data reformating we are referring to occurs in multiple calls to the Level 3 BLAS and it is done to improve the performance of the BLAS. We show this data reformatting cost is O(N^3). On the other hand, we also demonstrate that this data reformatting cost can be O(N^2) when SB format is used to implement the same right looking algorithms.

The other part of our paper centered on making the ideas of the paper [23] relate well to the fourteen topics listed in the abstract. In Section 2 we demonstrated that matrix multiplication was a fundamental algorithm of DLA. Hence, `DGEMM` becomes the major component of almost every DLAFA. Also, we showed the standard full format data structure of DLA is *not* the best one for `DGEMM` performance. We showed that using SB's was. Also, programming a DLAFA using SB format was just as easy, if not easier, as using conventional full format. Finally, it was demonstrated that DMC also adapted very well to using SB format because the atomic unit of a BCL was indeed a SB.

6 Acknowledgements

The continuation part of [23], mainly Sections 2.1, 2.2, 2.2.1 and all of Section 4, as applied $LU = PA$ factorization, are recent results obtained with Sid Chatterjee, Jim Sexton and mainly John Gunnels. John implemented a Linpack benchmark that obtained 70.72 TFlops in the fall of 2004 that placed IBM number one in the TOP500 list. We thank Cleve Ashcraft for suggesting a way to describe Section 3.2. I thank Bernie Rudin for his advice and a careful reading of the paper. I thank Jerzy Wasniewski for discussions. Finally, we thank Erik Altman for inviting me to submit this paper to the IBM Journal Issue on PACT.

References

[1] R. C. Agarwal, F. G. Gustavson. A Parallel Implementation of Matrix Multiplication and LU factorization on the IBM 3090. *Proceedings of the IFIP WG 2.5 Working Group on Aspects of Computation on Asychronous Parallel Processors*, book, Margaret Wright, ed. Stanford CA. 22-26 Aug. 1988, North Holland, pp. 217-221.

[2] R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.

[3] B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid and J. Wasnieski. A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. *ACM TOMS*, Vol. 31, No. 2 June 2005, pp. 201-227.

[4] B. S. Andersen, F. G. Gustavson, and J. Wasnieski. A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage. *ACM TOMS*, Vol. 27, No. 2 June 2001, pp. 214-244.

[5] S. Chatterjee et. al. Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. *IBM Journal of Research and Development*, Vol. 49, No. 2-3, March-May 2005, pp. 377-391.

[6] G. Birkhoff and S. MacLane. A Survey of Modern Algebra, revised edition 1953, Book. *Macmillan Company* New york.

[7] J. Bilmes, K. Asanovic, C. Whye Chin, J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. *Proceedings of International Conference on Supercomputing*, Vienna, Austria, 1997.

[8] R. K. Brayton, F. G. Gustavson and R. A. Willoughby. Some Results on Sparse Matrices. *Mathematics of Computation*, Vol. 24, No. 112, Oct. 1970, pp. 937,954.

[9] J. W. Demmel and J. J. Dongarra. ST-HEC: Reliable and Scalable Software for Linear Algebra Computations on High End Computers *NSF Award No.0325873*, Sept. 2005, pp. 1,13 and 1,4

[10] J. J. Dongarra, F. G. Gustavson, A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, Vol. 26, No. 1, Jan. 1984, pp. 91,112.

[11] J. J. Dongarra and J. Du Croz, S. Hammarling and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *TOMS*, Vol. 14, No. 1, Mar. 1988, pp. 1-17.

[12] J. J. Dongarra and J. Du Croz, S. Hammarling and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms, *TOMS*, Vol. 16, No. 1, Mar. 1990, pp. 1-17.

[13] J. J. Dongarra, R. C. Whaley. The BLACS v1.1 User's Guide. *LAWN Report #94*, May, 2000, pp. 1-66.

[14] E. W. Elmroth and F. G. Gustavson. Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. *IBM Journal of Research and Development*, Vol. 44, No. 4, July 2000, pp. 605,624.

[15] E. W. Elmroth and F. G. Gustavson. A New Much Faster and Simpler Algorithm for LAPACK DGELS. *BIT*, Vol. 41, No. 5, 2001, pp. 936,949.

[16] E. Elmroth, F. G. Gustavson, B. Kagstrom, and I. Jonsson. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, Vol. 46, No. 1, Mar. 2004, pp. 3,45.

[17] B. M. Fleisher. Private Communication. IBM, Yorktown Heights. Sep. 2001.

[18] G. Golub, C. VanLoan. Matrix Computations, Book. *Johns Hopkins Press* Baltimore, 1996

[19] J. Gunnels, F. G. Gustavson, G. Henry, and R. van de Geijn. Formal linear algebra methods environment (FLAME). *ACM TOMS*, Vol. 27, No. 4 Dec. 2001, pp. 422-455.

[20] J. A. Gunnels, F. G. Gustavson. A New Array Format for Symmetric and Triangular Matrices. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 247-255, 2004.

[21] J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn. A Family of High-Performance Matrix Multiplication Algorithms. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 256-265, 2004.

[22] F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, Vol. 41, No. 6, Nov. 1997, pp. 737,755.

[23] F. G. Gustavson High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. *IBM Journal of Research and Development*, Vol. 47, No. 1, Jan. 2003, pp. 31,55.

[24] F. G. Gustavson. New Generalized Data Structures for Matrices Lead to a Variety of High performance Dense Linear Algorithms. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 11-20, 2004.

[25] F. G. Gustavson, A.Henriksson, I. Jonsson, B. Kagstrom, P. Ling. Recursive Blocked

Data Formats and BLAS Dense Linear Algebra Algorithms. In B. Kagstrom et.al., editors *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, No. 1541, 1998, pp. 195,206.

[26] F. G. Gustavson and I. Jonsson. Minimal Storage High Performance Cholesky via Blocking and Recursion *IBM Journal of Research and Development*, Vol. 44, No. 6, Nov. 2000, pp. 823,849.

[27] IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. *IBM Pub. No. SA22-7272-04* Dec. 2001.

[28] B. Kagstrom, P. Ling, C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM TOMS*, Vol. 24, No. 3 Sep. 1998, pp. 268-302.

[29] 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 3.0*, SIAM, Philadelphia, 1999, (http://www.netlib.org/lapack/lug/lapack_lug.html).

[30] J. J. Dongarra, C. B. Moler, J. R. Bunch, G. W. Stewart. *LINPACK Users' Guide Release 2.0*, SIAM, Philadelphia, 1979.

[31] R. Kalla, B. Sinharoy, J. Tendler. Power 5 *HotChips-15*, August 17-19, 2003. Stanford, CA

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage, *TOMS*, Vol. 5, No. 3, Sep. 1979, pp. 308-323.

[33] C. D. Meyer. Matrix Analyis and Applied Linear Algebra, Book. it SIAM Philadelphia, 2001.

[34] L. Mirsky. An Introduction to Linear Algebra, revised edition 1972, Book. *Dover Publications* Mineola, New york.

[35] R. K. Montoye, E. Hokenek, S. L. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, Vol. 34, No. 1, Jan. 1990, pp. 59,70.

[36] R. K. Montoye. Private Communication. IBM, Austin. Sep. 2001.

[37] N. Park, B. Hong, V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel and Distributed Systems*, 14(7):640-654, 2003.

[38] B. Sinharoy, R.N. Kalla, J.M Tendler, R.G. Kovacs, R.J. Eickemeyer, J.B. Joyner. POWER5 System Microarchitecture *IBM Journal of Research and Development*, to appear Vol. 49

[39] L. S. Blackford, et. al. , *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997

[40] S. Toledo., A survey of out-of-core algorithms in numerical linear algebra. Book, *External Memory Algorithms*, eds., J. M. Abello and J. S. Vitter, pp. 161,179, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999.

[41] R. C. Whaley, A. Petitet, J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 2001(1-2), pp. 3-35.

7 Appendix

7.1 Illustration of factoring a pivot panel

We now illustrate how the algorithm for factoring a pivot panel works. Referring to Figure 11, we have local column 1 on `p(0:4,0)` as the `jp=3` the pivot panel. Input to the coding of Figure 18 is basically `jp=3`. Given `jp` we compute `il=jp/P`, `I=jp-il*P`, `jl=jp/Q` and `pc=jp-jl*Q`. Thus, `il,pr,jl,pc` = 0,3,1,0. Let us choose our processor as `p(3,0)` of the mesh. Having these four values we find `J=0,jl=1`; `CP(1:2,3,J)=3,6` and `apbc(3:5,3,J)=bl((3,8,13),3)`. Now we start the execution of Figure 18. We have already found `il,pr` and `jl,pc` in line 1 of the code. Now `I=pr` so `ijl=3` and `pb(3,0)=abpc(3,3,0)`. From Figure 11 we

have pb(3,0)=bl(jp,jp). After completion of the NS BC we have pb(0:4,J)=pb(3,0). Now, pb(3,0) is Cholesky factored by kernel routine dpofu. Next, ijl=3 and factored pb(3,0) is stored back into abpc(3,3,0) and ijl is set to 4. Now, ijle=5 and the do loop scales abpc(4:5,3,0) via two calls to dtrsm. From Figure 11 these two scalings are on global blocks bl((8,13),3).

7.2 Illustration of Send and Receive the Pivot Panel

To illustrate Figure 19 we consider jp=3 and p(4,0) as our processor; ie, I,J = 4,0. Other input ils,jls,ijl,pe(I) = (0,1,3,2) and abpc(3:5,I,J) = bl((4,9,14),3). The SPB's on p(4,0) are bl((4,9,14),3) of Figures 10 and 11. Figure 11 shows all r,c border vector with their final contents; ie, after Figure 19 has finished on all p(0:4,0). Therefore, initially one should view r,c as containing garbage. What Figure 19 does on p(4,0) is initiate the filling in bl((4,9,14),3) into all r,c that is appropriate on the entire mesh. The outer do loop is over SPB's bl((4,9,14),3) as il takes on values 0,1,2. We now step through this outer loop three times :

il,i,slr,ijl=0,4,1,3 and abpc(ijl,4,0) = bl(4,3). Since slr = 1 we do not have a full row BC. Now, bl(4,4) is on p(4,1) so K = 1 = mod(4,3). jl,slc = 1,4 and we have a full column BC from c(1,4,1).

il,i,slr,ijl=1,9,2,4 and abpc(ijl,4,0) = bl(9,3). Since slr = 2 we do have a full row BC. Now, bl(9,9) is on p(4,0) so K = 0 = mod(9,3). jl,slc = 3,4 and we have a full column BC from c(3,4,0).

il,i,slr,ijl=2,14,2,5 and abpc(ijl,4,0) = bl(14,3). Since slr = 2 we do have a full row BC. Now, bl(14,9) is on p(4,2) so K = 2 = mod(14,3). jl,slc = 4,3 and we do not have a full column BC from c(4,4,2).

This completes the illustration of Figure 19. In Figure 11 the r,c border vectors filled in with bl((4,9,14),3) are the final result of p(4,0)'s contribution to the DMC Send / Recieve of pivot panel jp = 3.

```
l  g  r in <--active-->
o  l    ac.
    ----------------------
    |  |  |              |
0  4|43|41|44            |
    |  |  |              |
1  9|93|91|94 97         |
    |  |  |              |
2  e|e3|e1|e4 e7 ea ed   |
    |  |  |              |
    |**|  |              |
    ----------------------
c   |  |** 43 73 a3 d3 **|
    ----------------------
global   1  4  7 10 13 16
local    0  1  2  3  4  5
```

Figure 21: jp=3 BPCL SCU on p(4,1)

7.3 Illustration of Perform a Schur Complement Update

To illustrate the coding of Figure 20, consider p(I,J) of Figure 21 with I,J= 4,1. We have added local and global column and row labels to aid in the clarity of our description. As described above Figure 20 has a double do loop structure. We have introduced an array jle(I,J) which denotes the last local j column on processor p(I,J). Note that jle(I,J) can be strictly less than qe(J). In Figure 21 this inequality occurs. The outer do loop on jl is therefore jls,jle(I,J) = 1:4. Local column 0 of c,abpc is inactive as $1 \leq$ jp. The active part of abpc is abpc(3:9,I,J). The active part of r is ils(jls):pe(I) = 0:2. The relevant part of CP is CP(jls:pe(I,J)+1,I,J) = 3,6,8,9,10. We now walk through the code: ijl=CP(jls,I,J)=3. do jl=jls,ije(I,J) translates to do jl=1,4.

jl=1, j=1+1*3=4, ils=3-6+3=0, i=4+0*5=4 and i=j. Since i=j, we do a SYRK update: $B_{4,4} = B_{4,4} - B_{4,3}B_{4,3}^T$ and ijl becomes 4. The inner do il=1,2 is next and we have two GEMM updates: $B_{9,4} = B_{9,4} - B_{9,3}B_{4,3}^T$ and $B_{14,4} = B_{14,4} - B_{14,3}B_{4,3}^T$. And ijl has become 6.

jl=2, j=1+2*3=7, ils=3-8+6=1, i=4+1*5=9 and i>j. Since i>j, we do a GEMM update: $B_{9,7} = B_{9,7} - B_{9,3}B_{7,3}^T$ and

`ijl` becomes 7. The inner `do il=2,2` is next and we have one `GEMM` update: $B_{14,7} = B_{14,7} - B_{14,3}B_{7,3}^T$ and `ijl` has become 8.

`jl=3, j=1+3*3=10, ils=3-9+8=2, i=4+2*5=14` and `i>j`. Since `i>j`, we do a `GEMM` update: $B_{14,10} = B_{14,10} - B_{14,3}B_{10,3}^T$ and `ijl` becomes 9. The inner `do il=3,2` is next and is empty.

`jl=4, j=1+4*3=13, ils=3-10+9=2, i=4+2*5=14` and `i>j`. Since `i>j`, we do a `GEMM` update: $B_{14,13} = B_{14,13} - B_{14,3}B_{13,3}^T$ and `ijl` becomes 10. The inner `do il=3,2` is next and is empty.

The outer loop is now complete and the code is done. It should be noted that extremely little extra operations are required in Figure 20. They are all fixed point operations (on `j,ils,i` and the logic of the `if-then-else` clause) and they all occur in the outer `do loop` on `jl`.

8 Glossary

Acronym	Meaning	Page #
AA	Algorithms and Architecture	3
abpc	array block packed cyclic	20
BC	BroadCast	19
BCL	Block Cyclic Layout	4
BPC	Block Packed Cyclic	20
BPG	Block Packed Global	20
BPCL	Block Packed Cyclic Lower	25
CP	Column Pointer	20
DGEMM	Dbl. prec. GEneral Matrix Multipy	2
DLA	Dense Linear Algebra	2
DLAFA	Dense Linear Algebra Factorization Algorithm	2
DMC	Distributed Memory Computing	2
FMA	Fused Multiply Add	4
gl	global local	26
glp	global local packed	25
HFP	Hybrid Full Packed	4
lg	local global	26
lgp	local global packed	25
LLA	Left Looking Algorithm	12
MCU	Most Commonly Used	2
MFlops	Million FLoating-point OPerations per Second	2
NDS	New Data Structure(s)	2
pe	local end index (row)	25
pl	global end index (row)	25
qe	local end index (column)	25
RFP	Rectangular Full Packed	2
RLA	Right Looking Algorithm	5
SB	Square Block	5
SBP	Square Block Packed	5
SBPL	Square Block Packed Lower	10
SCU	Schur Complement Update	5
SPB	Scaled Pivot Block	20
SPMD	Single Program Multiple Data	4
SRPA	Simple Related Partition Algorithm	13
TLB	Translation Lookaside Buffer	7