Dense and Iterative Concurrent Linear Algebra in the Multicomputer Toolbox*

Purushotham V. Bangalore Anthony Skjellum
Computer Science Department & NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

Chuck Baldwin
Center for Supercomputing Research
& Development
University of Illinois
Urbana, IL 61801

Steven G. Smith
Center for Computational Sciences
& Engineering
Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract

The Multicomputer Toolbox includes sparse, dense, and iterative scalable linear algebra libraries. Dense direct, and iterative linear algebra libraries are covered in this paper, as well as the distributed data structures used to implement these algorithms; concurrent BLAS are covered elsewhere. We discuss uniform calling interfaces and functionality for linear algebra libraries.

We include a detailed explanation of how the level-3 dense LU factorization works, including features that support data distribution independence with a blocked algorithm. We illustrate the data motion for this algorithm, and for a representative iterative algorithm, PCGS. We conclude that data distribution independent libraries are feasible and highly desirable. Much work remains to be done in performance tuning of these algorithms, though good portability and application-relevance have already been achieved.

1 Introduction

The Multicomputer Toolbox is a unique effort intended to provide data-distribution-independent programming for a large number of important algorithms, while supporting many real machines. The *Toolbox* framework, together with carefully designed message passing primitives, and concurrent data structures (that encapsulate data layout of matrices and vectors), provides an effective basis for building and then tuning the performance of whole applications, rather than single computational steps.

We recognize three distinct methodologies for solving linear systems[1]: direct methods (*i.e.*, LU factorization), semi-iterative or Krylov-subspace methods (*e.g.*, GMRES, QMR, PCG) and stationary iterative methods (*e.g.*, Jacobi, SOR). The *Toolbox* provides libraries to solve linear systems using the first two methods through the *Cdense*, *Csparse* and *Citer* libraries, and will soon include Jacobi and SOR algorithms (with support for dense and sparse data structures). In what follows, we illustrate *Cdense* and *Citer* data structures, and subsets of applicable calls. The high-level uniform calling interface for these algorithms is described in section 2. The functionality and interfaces of the *Cdense* library and the detailed explanation of the level-3 dense LU factorization algorithm are presented in section 3. Section 4 describes the functionality and interfaces of the *Citer* library with details about the PCGS algorithm. Conclusions and future work appear in section 5.

2 Uniform Calling Interface

A coherent linear solver interface is needed to ease application programming effort and to increase the portability of user programs to the latest generation

*The first two authors acknowledge financial support by the NSF Engineering Research Center for Computational Field Simulation (NSF ERC), Mississippi State University.

[1]Concurrent Basic Linear Algebra Subprograms (CBLAS) are described elsewhere (see [4]).

of *Toolbox* linear solvers. This is our first version of such an interface. If there were no high-level uniform calling interface for these diverse methods, the user would be forced to spend considerable effort to interface these libraries to application programs. Furthermore, in view of the need for "poly-algorithms" designed to increase application scalability over a range of problem sizes and concurrencies, uniform calling interfaces gain substantially increased importance.

The linear library interface consists of one structure and several calls:

```
typedef struct tbx_linear_system
{
  void  *Info;  /* "A", methods, data */
  void  *b; /* 1 or more right-hand sides */
  void  *x; /* 1 or more unknown vectors  */

  Method *linear_solver; /* specific method
             from Info for solving system */
  Extra *extra;
} Tbx_linear_system;
```

The information content is broadly as follows:

- `Info` — Everything to do with the matrix (or generalized linear operator)
- `b` — Everything to do with right hand sides
- `x` — Everything to do with unknowns
- `linear_solver` — The method for solving the system
- `extra` — Other state information needed by solver

The constructor for this object is as follows:

```
void (*new_linear_solver)
     (void *Info, *b, *x, linear_solver);
```

This constructor returns the `Tbx_linear_system` structure with all the default parameters set for the specified solvers. After the user has obtained this structure, problems can be solved by specifying the compact call:

```
error =
   Tbx_Solve_Linear_System(linear_system);
```

which expands at compile-time to

```
error = (*linear_system->linear_solver)
          (Info, b, x, extra);
```

The destructor is as follows:

```
void Tbx_free_linear_system
     (Tbx_linear_system *linear_system);
```

This operation destroys only the top-level data structure created by the previous constructor.

We find this high-level encapsulation to be good as long as there is no need for a broader interface between the solver and a higher-level accuracy check (such as if an inexact Newton method were coupled to a linear solver). For such cases, the Newton solver will have to be more intimately tied to the underlying linear solver, and a set of these methods will be needed, rather than a single Newton method. We comment on this further under section 4.

3 *Cdense* Functionality and Interfaces

The *Toolbox*'s concurrent dense matrix structure incorporates a local dense matrix represented in each process, the global problem size (`M, N`), and the matrix distribution `CMdistrib *Mdis`, which describes where each coefficient maps across a logical grid of processes. The latter contains the two-dimensional grid mailer defined by *Zipcode* (see [6]), as well as the row and column data distribution mappings (see [7]). As such, the matrix so represented is general, and is as follows:

```
typedef struct _Cmatrix
{
  matrix  *a; /* The local matrix. */
  int     M, N; /* Dimensions of Cmatrix. */

  /* Data distribution on grid */
  CMdistrib *Mdis;
} Cmatrix;
```

The local `matrix` data structure in turn includes the local storage, local size, and orientation (row- or column-major):

```
typedef struct _matrix
{
   int    m, n;
   double **s;
   int    orient_flag; /* row or
                          column-major */
} matrix;
```

Linear system data structures include the dense matrix structure; they are provided for both the level-2 and level-3 variants of LU factorization. The level-3 variant of this data structure is depicted here:

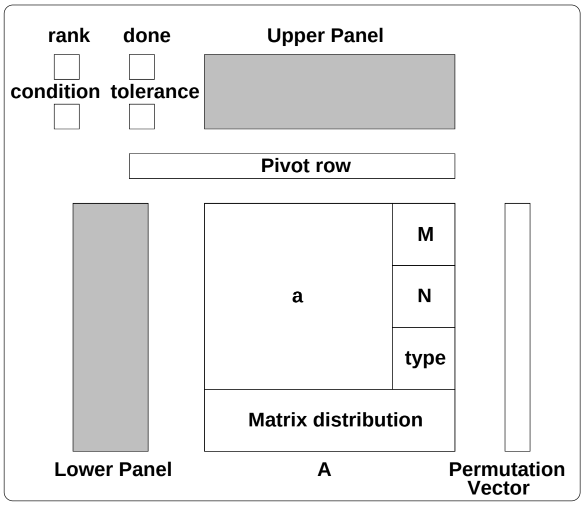

Figure 1: Depiction of data structure for level-3 algorithm. (For the level-2 algorithm, the "Lower Panel" is replaced by a single vector in each process column whereas "Upper Panel" disappears totally.)

```
typedef struct _Clu_info_lvl_3
{
  Cmatrix *A;
  /* permutation info [stored scalably]: */
  int *perm;
  /* temporary data space for pivot row: */
  double  *pivot;
  /* dynamic panel sizing function: */
  int   (*panel_size)();
  /* L temporary data space */
  double  *lower;
  /* U temporary data space */
  double  *upper;
  /* estimated rank after factorization */
  int     rank;
  /* "is factorization done?" flag */
  int     done;
  /* pivot tolerance coefficients: */
  double  condition;
  double  tolerance;
} Clu_info_lvl_3;
```

In the foregoing structure, the pointer to function `(*panel_size)()` determines the size of the next panel dynamically, permitting a dynamic tradeoff of bandwidth, latency, and load balancing during the level-3 factorization. Van de Geijn utilizes the blocking of the data distribution to achieve a fixed panel size [10]. Bischof has also explored variable blocking algorithms [2, 3].

The following function calls implement LU factorization within *Cdense*:

```
void lu_factor_Cmatrix_lvl_2(
       Clu_info_lvl_2 *LU,
       Cmatrix *B, Cvector *rhs)
void lu_factor_Cmatrix_lvl_3(
       Clu_info_lvl_3 *LU,
       Cmatrix *B, Cvector *rhs)
```

Both single (row-replicated) and multiple right-hand sides are supported. The following function calls implement parallel triangular solves:

```
/* forward only */
void fwd_solve_lu_Cmatrix_lvl_{2,3}(
       Clu_info_lvl_{2,3} *LU,
       Cvector *rhs, *sol)
/* back only */
void back_solve_lu_Cmatrix_lvl_{2,3}(
       Clu_info_lvl_{2,3} *LU,
       Cvector *rhs, *sol)
/* forward/back */
void solve_lu_Cmatrix_lvl_{2,3}(
       Clu_info_lvl{2,3} *LU,
       Cvector *rhs, *sol)
```

Other variants (such as those that can exploit pipelined back-solve techniques, despite data distribution independence) are in development.

3.1 Algorithm Development

The basic algorithm (right-looking, outer-product version) reduces an arbitrary real, non-singular matrix A to the familiar PLU product where L is unit lower triangular, U is upper triangular, and P is a permutation matrix (not explicitly formed). The following pseudo-code borrowed from Golub and Van Loan summarizes the sequential algorithm (see [5]):

$r = \min(m, n-1)$;
for $k = 1, r$
  $\sigma = \min(\mid A(i,k) \mid : i = k, r)$;
  $A(k, k:n) \leftrightarrow A(\sigma, k:n)$; $piv(k) = \sigma$;
  if $A(k,k) \neq 0$
    $A(k+1:m, k) = A(k+1:m, k)/A(k,k)$;
    $A(k+1:m, k+1:n) = A(k+1:m, k+1:n)$
      $-A(k, k+1:n)A(k+1:m, k)$;

For the sake of illustration here, the given matrix A of size $M \times N$ is distributed in a row- and column-scatter data distribution over the process grid of 2x2 processes. We have concentrated on the scattered distribution in our discussion, though our algorithm is

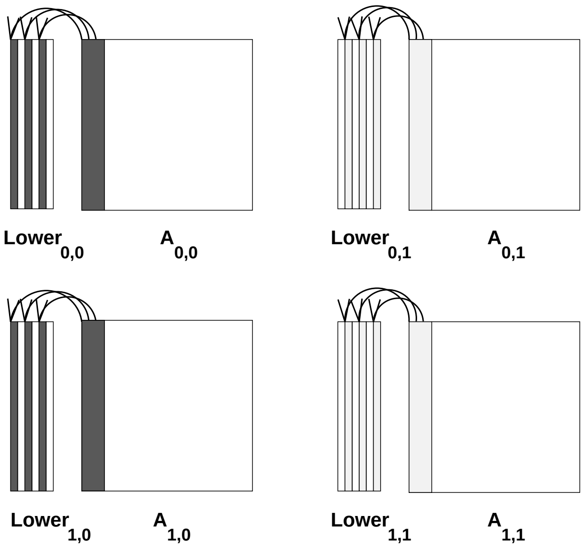

Figure 2: Snapshot of the local matrices and the lower panels after data has been assembled locally in each panel by copying from the A matrix.

data distribution independent. Each process will have a local matrix A of size $m \times n$ and the subscripts indicate the process row and column.

Panel Size Selection: The panel size k is chosen dynamically by the panel sizing function. Steps $1 \ldots 5$ are then performed for columns $K = J, J+1 \ldots, J+k-1$, where J is the first column of the panel. (The size of the panel and the data distribution of the matrix columns are independent in this algorithm.)

Step 1: All the processes copy their part of the columns to the lower data structure. Since we have two process columns and a scatter-scatter distribution, processes along process column 0 copy matrix columns 0, 2, and 4, while processes along process column 1 copy matrix columns 1, 3, and 5, as shown in figure 2. Now each process row performs a concatenation (currently implemented with combine and sum) to get a copy of the panel as shown in figure 3. (If we had selected a linear column data distribution then the lower panel in the process column 1 would be empty.) At this point, each process column will have a copy of the panel. Next, each process column performs the LU factorization of the panel while maintaining A's columns coherent with the developing panel.

Step 2: At the Kth step of the factorization, the following sequence of sub-steps advances the factorization of the current panel:

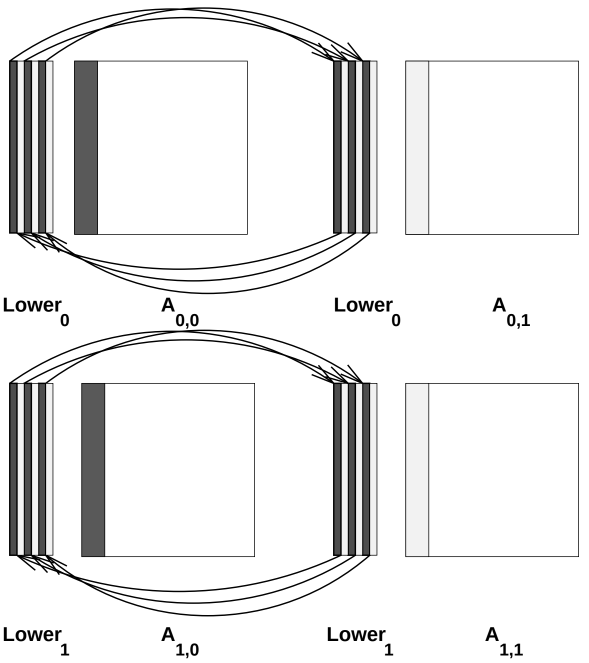

Figure 3: Snapshot of the local matrices and the lower panels after the row-wise concatenation of the columns. Observe that the panel names lose a subscript, signifying their replication in each process column.

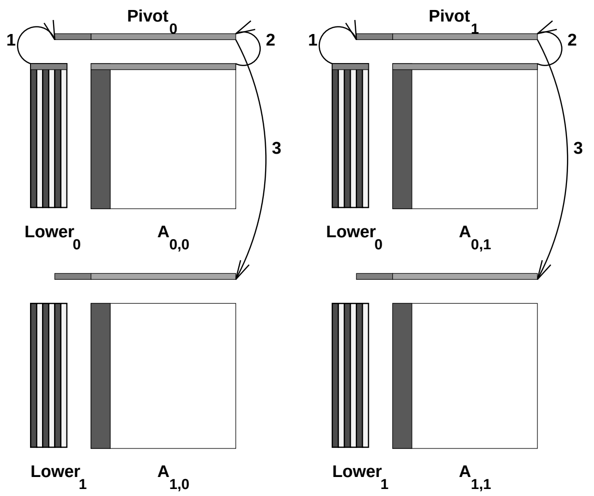

Figure 4: Operations performed to obtain the pivot row I.: 1) copy the Kth row of the panel to the pivot data structure; 2) copy the Kth row of A to the pivot data structure; 3) broadcast the pivot data structure. All process columns replicate work.

1. All processes in the process row owning the Kth row of A copy it to the pivot data structure and perform a broadcast of the pivot row along the process column as shown in figure 4. Now every process row has the Kth row of A, though this may not ultimately be the pivot row.

2. Each process owning a part of the Kth column finds the local maximum absolute value and subsequently performs a combine over the process column to find the global maximum (as shown in figure 5). Now each process in the column knows the global pivot row's number, and the value of the Kth pivot.

3. Since every process knows the global pivot, the distributed permutation vector is modified accordingly.

4. If the process row owning the Kth row owns (resp, does not own) the pivot row, then first perform an intra-process (resp, inter-process) row swap. Second, broadcast the correct pivot row across the process rows. Overwrite the Kth row in A with the pivot row. This is shown in figure 6.

5. Scale the Kth column (by the inverse of the pivot value) in the panel and perform a rank-one update (DGER) on the remaining part of the panel using the pivot and update. If the process column owns the Kth column, then update A as well.

Since we have replicated panels in each process column, there is no need to broadcast the updated column across rows. In figure 7 the updated column 0 is stored back in the A matrix by process column 0, whereas in figure 8 the updated column 1 is stored by process column 1. Figure 9 shows the state of the matrices after the entire panel has been factorized.

Step 3: Now we perform a vertical accumulation of the rows in each process row as shown in figure 10 to obtain the column-distributed, row-replicated upper panel. (If we had used a linear row data distribution then process row 0 will have all the rows.) This operation is similar to that of the accumulation of the columns to the lower panel, except that it is along the rows in this case.

Step 4: After the vertical accumulation, each process performs the triangular solve with multiple right-hand sides on the upper panel using the BLAS level-2 operation DTRSM as shown in figure 11. Observe that the submatrices appear as blocks instead of columns

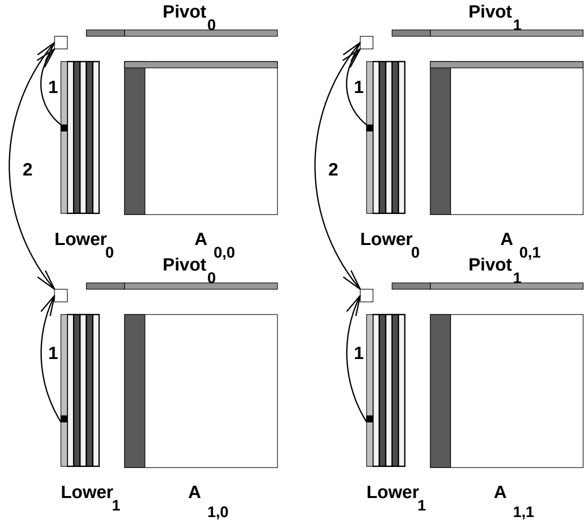

Figure 5: Operations performed to obtain the pivot row II.: 1) find the maximum element in column K locally; 2) combine over process rows for global concensus on maximum.

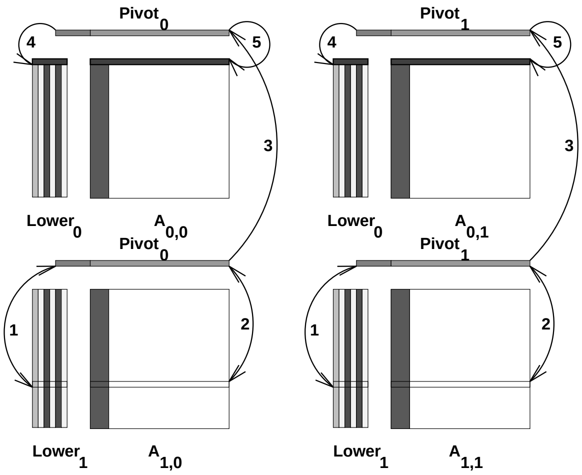

Figure 6: Operations performed to obtain the pivot row III.: 1) & 2) inter- or intra-process-row swap of the Kth row of the panel and matrix with the pivot row data structure; 3) broadcast the pivot row data structure, from the appropriate root, to update it in each process row. Now each process row has the correct pivot row in the pivot row data structure; 4) & 5) store the pivot row data structure back into the panel and A matrix.

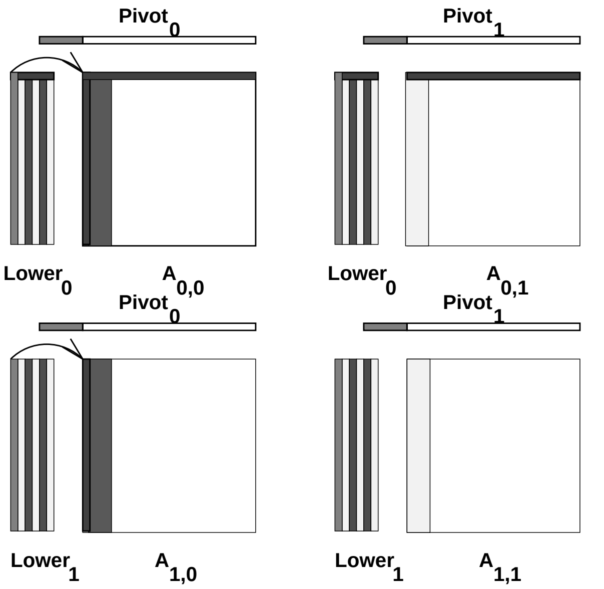

Figure 7: Column 0 in A is scaled by the inverse of the pivot value.

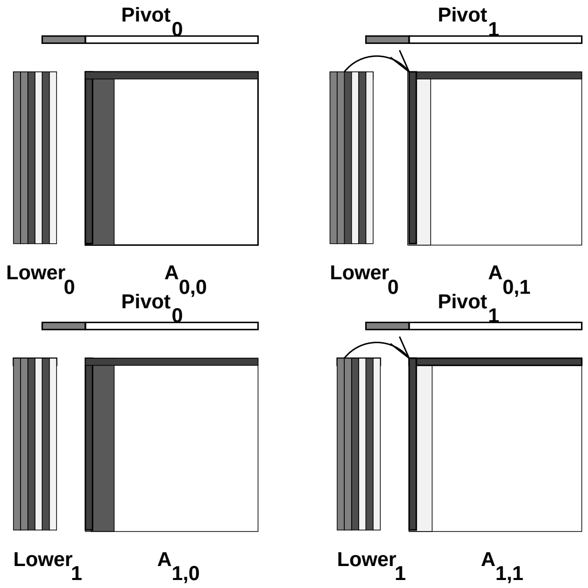

Figure 8: Column 1 in A is scaled by the inverse of the pivot value.

or rows to indicate their participation in the block operation.

Step 5: Finally, each process independently performs a rank-k matrix-matrix multiplication (DGEMM) to update the subblock of A using the upper and lower panels (see figure 12).

Things to note about the factorization algorithm (see [1]):

- We swap rows only within the panel, and the as-yet unfactored part of A. (This implies that we do not create a canonical, contiguous lower-triangular data structure L for the one-process case.)
- We currently implement a simple backsolve strategy; more sophisticated algorithms are possible, that take advantage of the opportunity to redistribute L and U at low cost during factorization in order to improve the triangular solves. We have not implemented these synergistic redistribution variants as yet.
- The choice of panel-size function remains a topic of research; we do not yet have a definitive algorithm for picking this function form based on the size of problem, characteristics of the hardware, and number of processes.

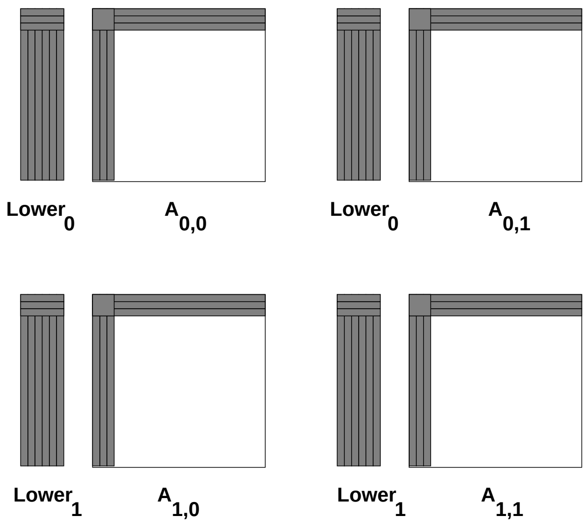

Figure 9: Snapshot of the local matrices and the lower panels after the factorization of the panel.

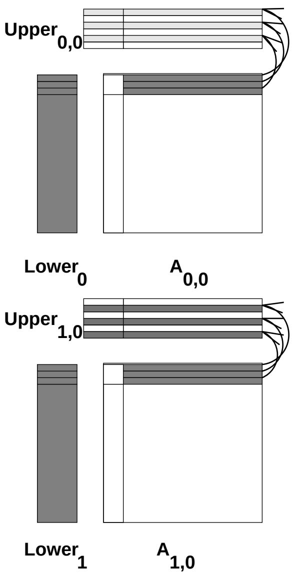

Figure 10: Snapshot of the local matrices and the upper panels after data has been assembled locally in each upper panel by copying rows from the A matrix.

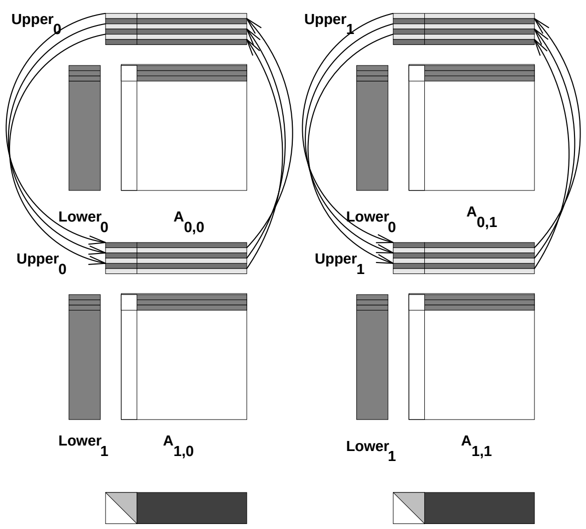

Figure 11: Snapshot of the local matrices and the lower panels after the column-wise concatenation of the rows. (Panel names drop a subscript, signifying their replication in the process rows.) The DTRSM operation on the upper panels is also indicated: this operation completes the deferred update of the A matrix to complete k rows of U.

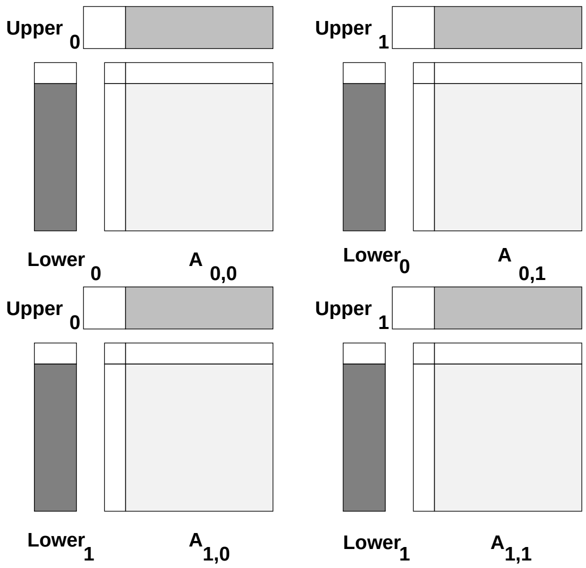

Figure 12: Snapshot of the upper and lower panels and A matrix after the DGEMM operation (matrix-matrix multiplication).

4 *Citer* Functionality and Interfaces

Citer currently supports the following Krylov subspace algorithms:

- GMRES — General Minimized Residual Method
- PCGS — Preconditioned Conjugate Gradient Squared
- PCG — Preconditioned Conjugate Gradient
- PTFQMR — Preconditioned Transpose-Free Quasi-Minimum Residual Method

When other methods demonstrate distinct advantages compared to these four algorithms, such methods can readily be added. For now, these appear to be a reasonable set of methods from which to choose. One exception to this is our intent to add the ability to support polynomial preconditioned iterative methods, specifically polynomial preconditioned conjugate gradient method (PPCG). On another note, we intend to add stationery methods such as Jacobi, SOR/Gauss-Seidel. These will be done as soon as possible, that is, before the *Citer* library has its first public release (1st quarter of 1994).

4.1 General Structures for *Citer*

Significant encapsulation occurs within *Citer*. For instance, the following structure houses the concurrent inner products to be supported for a typical iterative method:

```
typedef struct _inner_product_bundle
{
  /* inner_product(x,y,&ip,extra)    */
   Method *inner_product;
  /* as above but for skew vectors   */
   Method *skew_inner_product;
  /* multi_inner_product(v1,v2,ip,num,extra)
   * v1, v2 are arrays with num vectors.
   * ip is array of doubles for results */
   Method *multi_inner_product;
   /* as above but for skew vectors  */
   Method *skew_multi_inner_product;
} Inner_Product_Bundle;
```

(Standard instantiations of these methods reside in the *Cvector* library.) Furthermore, the following structure packages matrix-vector multiplication functions for iterative solvers:

```
typedef struct _matvec_bundle
{
  /* x := b * A * y + c * z
   *      matvec(b,A,y,c,z,x,extra) */
  Method *matvec;
  /* x := b * y^T * A + c * z
   *      matvec_T(b,A,y,c,z,x,extra) */
  Method *matvec_T;
} Matvec_Bundle;
```

It should be noted that, separate from *Citer*, one supports specific matrix data structures and matrix-vector multiplications. We currently support dense matrix-vector multiplication for the `Cmatrix` data structure defined by the *Cdense* library, and we are considering the addition of a general sparse technology as well. This remains to be done. However, since applications normally provide their own parallel matrix-vector product (representing a linear operator), the current interface is sufficient to make *Citer* useful in practice.

Within *Citer*, we add the loose notion of a "matrix," though matrix-free methods are equally well supported:

```
typedef struct _citer_matrix
{
  void           *A;
  CMdistrib      *Mdis;
  Matvec_Bundle  *mv_bundle;
  long            rows, columns;
} Citer_Matrix;
```

The culmination comes in the top-level data structure `Citer_Info`; the following structure describes the entire iterative linear system:

```
typedef struct _citer_info
{
  Citer_Matrix *coeff_matrix;
  Citer_Matrix *left_precond;/* z := M r */
  Citer_Matrix *right_precond;
  /* Solves Ax = b
   * iter_solver(iter_info,x,b)
   * method dependent data is located in
   * iter_solver -> extra                 */
  Method       *iter_solver;
  /* called at each iteration to store
   * results                              */
  Method       *store_results;
  /* called at start of each iterations to
   * determine if solver should be
   *  terminated                          */
  Method       *term_cond;
} Citer_Info;
```

Note that we support both left and/or right preconditioning in the formalism. Furthermore, we provide a general format in which the user can elect to store arbitrary information about some or all of the past iterates through the `store_results` mechanism. Finally, "extra information" for the implementation of the termination condition method is also supported:

```
typedef struct _citer_residual_test_extra
{
   int              max_iter;
   double           tol;
} Citer_Residual_Test_Extra;
```

4.2 Data Structures for Preconditioned Conjugate Gradient Squared (PCGS)

In the foregoing section, we defined the framework for iterative solvers within *Citer*. In this section, we describe the additional data structures for one of the supported solvers, Preconditioned Conjugate Gradient Squared (PCGS).

```
typedef struct _citer_pcgs_extra
{
   Cvector        *rt, *p, *z, *r;
   Cvector        *q, *u, *v, *w, *t;
   Inner_Product_Bundle *ip_bundle;
   /* a*x + b*y; scaled vector sum:*/
   Method         *axpby;
   double          breakdown_tolerance;
} Citer_PCGS_Extra;
```

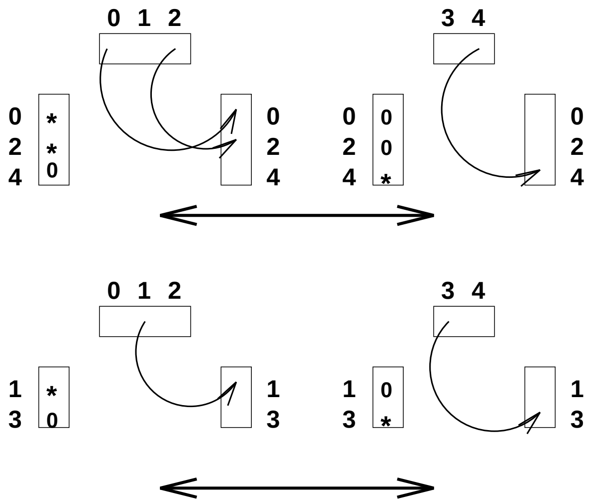

Figure 13: The skew inner product used in PCGS. A linear-row, scatter-column decomposition is depicted. In this figure, the row-replicated, column-distributed vector is copied in two phases into a column-replicated, row-distributed vector. The destination is first set to zero, then elements that remain within process boundaries are copied locally; finally, a horizontal concatenation (currently implemented as combine with summation) completes the conversion. Stars implied that final values have been copied into the destination vector, while zeroes indicate the initial state of the destination vector elements.

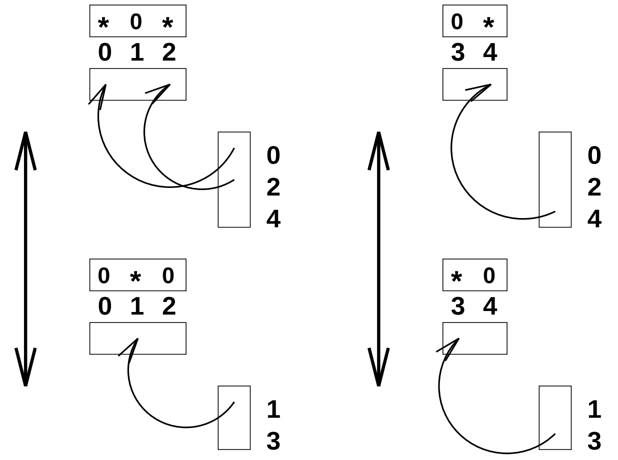

Figure 14: The skew inner product used in PCGS. A linear-row, scatter-column decomposition is depicted. The operation shown in figure 13 is reversed by this operation.

The constructor, destructor, and the extra information needed by PCGS are as follows:

```
Extra *citer_new_PCGS_extra(
       Inner_Product_Bundle *ip_bundle,
       Method *axpby, CMdistrib *matrix_dis,
       long row_size,long col_size,
       double tol);

void citer_free_PCGS_extra(Extra *extra);
```

Finally, the actual call for this solver is as follows:

```
int citer_PCGS(Citer_Info *PCCG,
                  Cvector *rhs, Cvector *sol);
```

which is the same calling sequence used by all *Citer* solvers.

Here is a high-level representation of the Preconditioned Conjugate Gradient method, with a symmetric preconditioner C:

$r_0 = b - Ax_0$; $p_{-1} = 0$; $\rho_{-1} = 1$;
$n = 0$;

while (not done)

$$z_n = (C^T C)^{-1} r_n = M^{-1} r_n;$$
$$\rho_n = r_n^T z_n;\ \beta_n = \tfrac{\rho_n}{\rho_{n-1}};$$
$$p_n = z_n + \beta_n p_{n-1};$$
$$\sigma_n = p_n^T A p_n;\ \alpha_n = \tfrac{\rho_n}{\sigma_{n-1}};$$
$$r_{n+1} = r_n - \alpha_n A p_n;$$
$$x_{n+1} = x_n + \alpha_n p_n;$$
$$n = n + 1;$$

In the foregoing, termination of the loop is most often based on the norm of the residual. The code fragment below provides termination through a general termination function. The code fragment for the Preconditioned Conjugate Gradient method is as follows:

```
cons_Cvector(p, 0.0);
(*matvec) (-1.0, A, sol, 1.0, rhs, r,
           matvec_extra);
new_rho = sqrt((*inner_prod)
               (r, r, inner_prod_extra));
(*store_results) (PCG, 0, new_rho, r,
                  store_results_extra);
orig_rho = new_rho;
new_gamma = 1;
for (k = 0; !(*term_cond) (PCG, k, orig_rho,
          new_rho, r, term_cond_extra); k++)
{
      (* precond) (1.0, A, r, 0.0, NULL, z,
                   precond_extra);
      old_gamma = new_gamma;
      new_gamma  = (*skew_inner_prod)
```

```
            (r, z, skew_inner_prod_extra);
    beta = new_gamma / old_gamma;
    (*axpby) (1.0, z, beta, p, p);
    (*matvec) (1.0, A, p, 0.0, NULL, w,
               matvec_extra);
    sigma = (*skew_inner_prod)
            (p, w, skew_inner_prod_extra);
    alpha = new_gamma / sigma;
    (*axpby)(1.0, sol, alpha, p, sol);
    (*axpby)(1.0, r, -alpha, w, r);
    old_rho = new_rho;
    new_rho = sqrt((*inner_prod)
               (r, r, inner_prod_extra));
    (*store_results) (PCG, k+1, new_rho,
                 r, store_results_extra);
}
```

In the above code fragment, the `(*axpby)()` vector scaling operations are communication free; the `(*matvec)()` operations utilize the user-specified matrix-vector multiplications. The `(*inner_prod)()` and `(*skew_inner_prod)()` implement inner products, depending on the relative orientation of the concurrent vectors. figures 13, 14 show the data motion of skew inner products. The function `(*term_cond)()` permits the user to specify complex termination criteria for the solver. The function `(*store_results)()` allows applications to save intermediate iterations, if necessary.

5 Summary and Commentary

In this paper, we have presented the data structures, uniform calling interface and data motion for representative algorithms from the *Multicomputer Toolbox*'s dense direct, and iterative linear solver libraries.

The following issues remain to be studied and further demonstrated in the future:

- It is possible to obtain good performance and functionality from a data distribution independent algorithms (see [8]),
- Sometimes it is necessary to "rethink" traditional ideas in order to design a scalable data distribution independent algorithm,
- Performance is sometimes not the only guiding principle in designing parallel libraries,
- A software paradigm that has this kind of capability, like the *Multicomputer Toolbox*, fits well into application software.

References

[1] Purushotham V. Bangalore, Anthony Skjellum, Chuck Baldwin, and Steven G. Smith. Data-Distribution-Independent, Concurrent Block LU Factorization. *In preparation.*, January 1994.

[2] Christian H. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3(3):193–208, 1989.

[3] Christian H. Bischof and Philippe G. Lacroute. An adaptive blocking strategy for matrix factorizations. In H. Burkhart, editor, *Lecture Notes in Computer Science 457*, pages 210–221, New York, NY, 1990. Springer Verlag.

[4] Robert D. Falgout, Anthony Skjellum, Steven G. Smith, and Charles H. Still. The Multicomputer Toolbox Approach to Concurrent BLAS. *Submitted to Concurrency: Practice & Experience.*, October 1993.

[5] Gene Golub and Charles Van Loan. *Matrix Computations*. John Hopkins University Press, 1991. 2nd edition.

[6] Anthony Skjellum. The Design and Evolution of Zipcode. *Parallel Computing*, 1993. (Invited paper for special issue on message passing, to appear).

[7] Anthony Skjellum. The Multicomputer Toolbox: Current and Future Directions. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, October 1993.

[8] Anthony Skjellum and Chuck H. Baldwin. *The Multicomputer Toolbox:* Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[9] Anthony Skjellum, Alvin P. Leung, Charles H. Still Steven G. Smith, Robert D. Falgout, and Chuck H. Baldwin. The Multicomputer Toolbox – First-Generation Scalable Libraries. In *Proceedings of HICSS–27*. IEEE Computer Society Press, 1994. HICSS–27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[10] Robert A. van de Geijn. Massively Parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 Systems. Technical report, Department of Computer Sciences, University of Texas, 1991.