Aztec User's Guide
Version 1.0

Scott A. Hutchinson, John N. Shadid, Ray S. Tuminaro

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.
Aztec User’s Guide*
Version 1.0

Scott A. Hutchinson†  John N. Shadid§  Ray S. Tuminaro‡
Massively Parallel Computing Research Laboratory
Sandia National Laboratories
Albuquerque, NM  87185

Abstract

Aztec is an iterative library that greatly simplifies the parallelization process when solving the linear systems of equations $Ax = b$ where $A$ is a user supplied $n \times n$ sparse matrix, $b$ is a user supplied vector of length $n$ and $x$ is a vector of length $n$ to be computed. Aztec is intended as a software tool for users who want to avoid cumbersome parallel programming details but who have large sparse linear systems which require an efficiently utilized parallel processing system. A collection of data transformation tools are provided that allow for easy creation of distributed sparse unstructured matrices for parallel solution. Once the distributed matrix is created, computation can be performed on any of the parallel machines running Aztec: nCUBE 2, IBM SP2 and Intel Paragon, MPI platforms as well as standard serial and vector platforms.

Aztec includes a number of Krylov iterative methods such as conjugate gradient (CG), generalized minimum residual (GMRES) and stabilized biconjugate gradient (BiCGSTAB) to solve systems of equations. These Krylov methods are used in conjunction with various preconditioners such as polynomial or domain decomposition methods using LU or incomplete LU factorizations within subdomains. Although the matrix $A$ can be general, the package has been designed for matrices arising from the approximation of partial differential equations (PDEs). In particular, the Aztec package is oriented toward systems arising from PDE applications.

* This work was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-94AL85000. The Aztec software package was developed by the authors at Sandia National Laboratories and is under copyright protection.
† Parallel Computational Sciences Department; sahutch@cs.sandia.gov; (505) 845-7996
§ Parallel Computational Sciences Department; jnshadi@cs.sandia.gov; (505) 845-7876
‡ Applied & Numerical Mathematics Department; tuminaro@cs.sandia.gov; (505) 845-7298
# Contents

1 Overview 1

2 Aztec: High Level View 1
   2.1 Aztec Options 2
   2.2 Aztec parameters 6
   2.3 Return status 7

3 Data Formats 7
   3.1 Distributed Modified Sparse Row (DMSR) Format 9
   3.2 Distributed Variable Block Row (DVBR) Format 9

4 High Level Data Interface 10

5 Examples 12

6 Advanced Topics 15
   6.1 Data Layout 15
   6.2 Reusing factorizations 17
   6.3 Important Constants 18
   6.4 AZ_transform Subtasks 18

7 Aztec Functions 19
   AZ_broadcast 20
   AZ_check_input 21
   AZ_check_msr 21
   AZ_check_vbr 22
   AZ_defaults 22
   AZ_exchange_bdry 23
   AZ_find_index 23
   AZ_find_local_indices 24
   AZ_find_procs_forExterns 24
   AZ_free_memory 25
   AZ_gavg_double 25
   AZ_gdot 25
   AZ_gmax_double 26
   AZ_gmax_int 26
   AZ_gmax_matrix_norm 26
   AZ_gmax_vec 27
   AZ_gmin_double 27
   AZ_gmin_int 27
   AZ_gsum_double 28
   AZ_gsum_int 28
   AZ_gsum_vec_int 28
   AZ_gvector_norm 29
   AZ_init_quick_find 29
   AZ_matvec_mult 30
   AZ_msr2vbr 31
AZ_order_ele ............................................. 31
AZ_print_error ........................................... 32
AZ_processor_info ....................................... 33
AZ_quick_find ............................................ 33
AZ_read_msg_matrix ..................................... 34
AZ_read_update .......................................... 34
AZ_reorder_matrix ....................................... 35
AZ_set_message_info ................................... 36
AZ_solve .................................................... 37
AZ_sort ..................................................... 37
AZ_transform .............................................. 38
Notation Conventions

Different fonts are used to indicate program fragments, keywords, variables, or parameters in order to clarify the presentation. The table below describes the meaning denoted by these different fonts.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>typewriter</td>
<td>File names, code examples and code fragments.</td>
</tr>
<tr>
<td>sans serif</td>
<td>C language elements such as function names and constants when they appear embedded in text or function definition syntax lines.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Parameter and variable names when they appear embedded in text or function definition syntax lines.</td>
</tr>
<tr>
<td>AZ-</td>
<td>C language elements such as function names and constants which are supplied by the Aztec library.</td>
</tr>
</tbody>
</table>

Code Distribution

Aztec is publicly available for research purposes and may be licensed for commercial application. The code is distributed along with technical documentation, example C and Fortran driver routines and sample input files via the internet. It may be obtained by contacting one of the authors listed on page i of this report.
1. Overview. Aztec is an iterative library that greatly simplifies the parallelization process when solving the linear system of equations

\[ Ax = b \]

where \( A \) is a user supplied \( n \times n \) sparse matrix, \( b \) is a user supplied vector of length \( n \) and \( x \) is a vector of length \( n \) to be computed. Aztec is intended as a software tool for users who want to avoid cumbersome parallel programming details but who have large sparse linear systems requiring efficient use of a parallel processing system. The most complicated parallelization task for an Aztec user is the distributed matrix specification for the particular application. Although this may seem difficult, a collection of data transformation tools are provided that allow creation of distributed sparse unstructured matrices for parallel solution with ease of effort that is similar to a serial implementation. Background information regarding the data transformation tools can be found in [5]. Once the distributed matrix is created, computation can occur on any of the parallel machines running Aztec: nCUBE 2, IBM SP2, Intel Paragon, and MPI platforms. In addition, Aztec can be used on standard serial and vector platforms such as SUN, SGI and CRAY computers.

Aztec includes a number of Krylov iterative methods such as conjugate gradient (CG), generalized minimum residual (GMRES) and stabilized biconjugate gradient (BiCGSTAB) to solve systems of equations. These Krylov methods are used in conjunction with various preconditioners such as polynomial preconditioners or domain decomposition using LU or incomplete LU factorizations within subdomains. Background information concerning the iterative methods and the preconditioners can be found in [4]. Although the matrix \( A \) can be general, the package has been designed for matrices arising from the approximation of partial differential equations (PDEs). In particular, the preconditioners, iterative methods and parallelization techniques are oriented toward systems arising from PDE applications. Lastly, Aztec can use one of two different sparse matrix notations – either a point-entry modified sparse row (MSR) format or a block-entry variable block row (VBR) format. These two formats have been generalized for parallel implementation and, as such, are referred to as “distributed” yielding DMSR and DVBR references.

The remainder of this guide describes how Aztec is invoked within an application. Aztec is written in ANSI-standard c and as such, all arrays in the descriptions which follow begin indexing with 0. Also, all function prototypes (loosely, descriptions) are presented in ANSI c format. Section 2 discusses iterative method, preconditioning and convergence options. Section 3 explains vectors and sparse matrix formats supported by Aztec. In Section 4 we discuss the data transformation tool for creating distributed vectors and matrices. A concrete detailed programming example using this tool is given in Section 5 and some advance topics are discussed in Section 6. Finally, Section 7 gives a glossary of Aztec functions available to users.

2. Aztec: High Level View. The following tasks must be performed to successfully invoke Aztec:

- describe the parallel machine (e.g. number of processors).
- initialize matrix and vector data structures.
- choose iterative methods, preconditioners and the convergence criteria.
- initialize the right hand side and initial guess.
- invoke the solver.
A sample C program is shown in Figure 1 omitting declarations and some parameters. The functions init_matrix_vector_structures, init_options, and init_guess_and_rhs are supplied by the user. In this section, we give an overview of Aztec's features by describing the user input arrays, options and params, that are set by the user in the function init_options. A discussion of the other subroutines is deferred to Sections 4 and 5.

2.1. Aztec Options. options is an integer array of length AZ_OPTIONS_SIZE set by the user. It is used (but not altered) by the function AZ_solve to choose between iterative solvers, preconditioners, etc. Below we discuss each of the possible options. In some of these descriptions, reference is made to a user-defined options or params value which is yet be introduced. These descriptions will follow but the reader may wish to "jump ahead" and read the descriptions if the immediate context is not clear.

Specifications

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ_cg</td>
<td>Conjugate gradient (only applicable to symmetric positive definite matrices).</td>
</tr>
<tr>
<td>AZ_gmres</td>
<td>Restarted generalized minimal residual.</td>
</tr>
<tr>
<td>AZ_cgs</td>
<td>Conjugate gradient squared.</td>
</tr>
<tr>
<td>AZ_tfqmr</td>
<td>Transpose-free quasi-minimal residual.</td>
</tr>
</tbody>
</table>

---

1 The entire main program with specific sample problems is distributed with the package in the file az_main.c
AZ_bicgstab: Bi-conjugate gradient with stabilization.
AZ.lu: Sparse direct solver (single processor only).

options[AZ.scaling]: Specifies scaling algorithm. The entire matrix is scaled (overwriting the old matrix). Additionally, the right hand side, the initial guess and the final computed solution are scaled if necessary. DEFAULT: AZ.none.

AZ_none: No scaling.
AZ_Jacobi: Point Jacobi scaling.
AZ_BJacobi: Block Jacobi scaling where the block size corresponds to the VBR blocks. Point Jacobi scaling is performed when using the MSR format.
AZ_row_sum: Scale each row so the magnitude of its elements sum to 1.
AZ_sym_diag: Symmetric scaling so diagonal elements are 1.
AZ_sym_row_sum: Symmetric scaling using the matrix row sums.

options[AZ.precond]: Specifies preconditioner. DEFAULT: AZ_none.

AZ_none: No preconditioning.
AZ_Jacobi: $k$ step Jacobi (block Jacobi for DVBR matrices where each block corresponds to a VBR block). The number of Jacobi steps, $k$, is set via options[AZ.poly_ord].
AZ_Neumann: Neumann series polynomial where the polynomial order is set via options[AZ.poly_ord].
AZ_Is: Least-squares polynomial where the polynomial order is set via options[AZ.poly_ord].
AZ_lu: Domain decomposition preconditioner (additive Schwarz) using a sparse LU factorization in conjunction with a drop tolerance params[AZ.drop] on each processor's submatrix. The treatment of external variables in the submatrix is determined by options[AZ_overlap]. The current sparse lu factorization is provided by the package y12m [6].
AZ_ilu: Similar to AZ_lu using ilu(0) instead of LU.
AZ_bilu: Similar to AZ_lu using block ilu(0) instead of LU where each block corresponds to a VBR block.
Non-overlapping domain decomposition (additive Schwarz) \( k \) step symmetric Gauss-Siedel. In particular, a symmetric Gauss-Siedel domain decomposition procedure is used where each processor independently performs one step of symmetric Gauss-Siedel on its local matrix, followed by communication to update boundary values before the next local symmetric Gauss-Siedel step. The number of steps, \( k \), is set via \texttt{options[AZ.polyOrd]}. 

\texttt{options[AZ.conv]} determines the residual expression used in convergence checks and printing. DEFAULT: \texttt{AZ.r0}. The iterative solver terminates if the corresponding residual expression is less than \texttt{params[AZ.tol]}: 

\[
\begin{align*}
\text{AZ.r0} & \quad \|r\|_2/\|r^{(0)}\|_2 \\
\text{AZ.rhs} & \quad \|r\|_2/\|b\|_2 \\
\text{AZ.Anorm} & \quad \|r\|_\infty/\|A\|_\infty \\
\text{AZ.sol} & \quad \|r\|_\infty/(\|A\|_\infty \cdot \|x\|_1 + \|b\|_\infty) \\
\text{AZ.weighted} & \quad \|r\|_{WRMS}
\end{align*}
\]

where \( \|r\|_{WRMS} = \sqrt{(1/n) \sum_{i=1}^{n} (r_i/w_i)^2} \), \( n \) is the total number of unknowns, \( w \) is a weight vector provided by the user via \texttt{params[AZ.weights]} and \( r^{(0)} \) is the initial residual.

\texttt{options[AZ.output]} specifies information (residual expressions - see \texttt{options[AZ.conv]}) to be printed. DEFAULT: 1.

\begin{itemize}
\item \texttt{AZ.all} print out the matrix and indexing vectors for each processor. Print out all intermediate residual expressions.
\item \texttt{AZ.none} no intermediate results are printed.
\item \texttt{AZ.last} print out only the final residual expression.
\item > 0 print residual expression every \texttt{options[AZ.output]} iterations.
\end{itemize}

\texttt{options[AZ.pre_calc]} indicates whether to use factorization information from previous calls to \texttt{AZ.solve}. DEFAULT: \texttt{AZ.calc}.

\begin{itemize}
\item \texttt{AZ.calc} use no information from previous \texttt{AZ.solve} calls.
\item \texttt{AZ.recalc} use preprocessing information from a previous call but recalculate preconditioning factors. This is primarily intended for factorization software which performs a symbolic stage.
\end{itemize}
**AZ_reuse**

Use preconditioner from a previous AZ_solve call, do not recalculate preconditioning factors. Also, use scaling factors from previous call to scale the right hand side, initial guess and the final solution.

**options[AZ_max_iter]**

Maximum number of iterations. DEFAULT: 500.

**options[AZ_poly_ord]**

The polynomial order when using polynomial preconditioning. Also, the number of steps when using Jacobi or symmetric Gauss-Seidel preconditioning. DEFAULT: 3.

**options[AZ_overlap]**

Determines the submatrices factored with the domain decomposition algorithms: AZ.lu, AZ.ilu, AZ.bilu. DEFAULT: AZ_none.

**AZ_none**

Factor the local submatrix defined on this processor discarding column entries that correspond to external elements.

**AZ_diag**

Factor the local submatrix defined on this processor augmented by a diagonal (block diagonal for VBR format) matrix. This diagonal matrix corresponds to the diagonal entries of the matrix rows (found on other processors) associated with external elements. This can be viewed as taking one Jacobi step to update the external elements and then performing domain decomposition with AZ_none on the residual equations.

**AZ_full**

Factor the local submatrix defined on this processor augmented by the rows (found on other processors) associated with external variables (discarding column entries associated with variables not defined on this processor). The resulting procedure is an overlapped additive Schwarz procedure.

**options[AZ_kspace]**

Krylov subspace size for restarted GMRES. DEFAULT: 30.

**options[AZ_orthog]**

GMRES orthogonalization scheme. DEFAULT: AZ_classic.

**AZ_classic**

Classical Gramm-Schmidt orthogonalization.

**AZ_modified**

Modified Gramm-Schmidt orthogonalization.

**options[AZ_aux_vec]**

Determines $\check{r}$ (a required vector within some iterative methods). The convergence behavior varies slightly depending on how this is set. DEFAULT: AZ_resid.

**AZ_resid**

$\check{r}$ is set to the initial residual vector.
AZ_rand \cdot \bar{r} \text{ is set to random numbers between } -1 \text{ and } 1.

NOTE: When using this option, the convergence depends on the number of processors (i.e. the iterates obtained with } x \text{ processors differ from the iterates obtained with } y \text{ processors if } x \neq y).

2.2. Aztec parameters. params is a double precision array set by the user and normally of length AZ_PARAMS_SIZE. However, when a weight vector is needed for the convergence check (i.e. options[AZ_conv] = AZ_weighted), it is embedded in params whose length must now be AZ_PARAMS_SIZE + # of elements updated on this processor. In either case, the contents of params are used (but not altered) by the function AZ.solve to control the behavior of the iterative methods. The array elements are specified as follows:

Specifications

<table>
<thead>
<tr>
<th>params[AZ_tol]</th>
<th>Specifies tolerance value used in conjunction with convergence tests. DEFAULT: \text{10}^{-6}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>params[AZ_drop]</td>
<td>Specifies drop tolerance used in conjunction with LU preconditioner. DEFAULT: 0.0.</td>
</tr>
<tr>
<td>params[AZ_weights]</td>
<td>When options[AZ_conv] = AZ_weighted, the i'\text{th} local component of the weight vector is stored in the location params[AZ_weights+i].</td>
</tr>
</tbody>
</table>

Figure 2 illustrates a sample function init.options where the Aztec function AZ_defaults sets the default options.

Example

```c
void init_options(int options[AZ_OPTIONS_SIZE],
                  double params[AZ_PARAMS_SIZE])
{
    AZ_defaults(options, params);
    options[AZ_solver] = AZ_cgs;
    options[AZ_scaling] = AZ_none;
    options[AZ_precond] = AZ_ls;
    options[AZ_output] = 1;
    options[AZ_max_iter] = 640;
    options[AZ_poly_ord] = 7;
    params[AZ_tol] = 0.0000001;
}
```

\textbf{Fig. 2. Example option initialization routine (init_options).}
2.3. Return status. *status* is a double precision array of length AZ_STATUS_SIZE returned from AZ_solve. The contents of *status* are described below.

**Specifications**

<table>
<thead>
<tr>
<th><strong>status[AZ_its]</strong></th>
<th>Number of iterations taken by the iterative method.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>status[AZ_why]</strong></td>
<td>Reason why AZ_solve terminated.</td>
</tr>
<tr>
<td><strong>AZ_normal</strong></td>
<td>User requested convergence criteria is satisfied.</td>
</tr>
<tr>
<td><strong>AZ_param</strong></td>
<td>User requested option is not available.</td>
</tr>
<tr>
<td><strong>AZ_breakdown</strong></td>
<td>Numerical breakdown occurred.</td>
</tr>
<tr>
<td><strong>AZ_loss</strong></td>
<td>Numerical loss of precision occurred.</td>
</tr>
<tr>
<td><strong>AZ_maxits</strong></td>
<td>Maximum iterations taken without convergence.</td>
</tr>
<tr>
<td><strong>status[AZ_r]</strong></td>
<td>The true residual norm corresponding to the choice options[AZ_conv] (this norm is calculated using the computed solution).</td>
</tr>
<tr>
<td><strong>status[AZ_scaled_r]</strong></td>
<td>The true residual ratio expression as defined by options[AZ_conv].</td>
</tr>
<tr>
<td><strong>status[AZ_rec_r]</strong></td>
<td>Norm corresponding to options[AZ_conv] of final residual or estimated final residual (recursively computed by iterative method). Note: When using the 2-norm, tfqmr computes an estimate of the residual norm instead of computing the residual.</td>
</tr>
</tbody>
</table>

When AZ_solve returns abnormally, the user may elect to restart using the current computed solution as an initial guess.

3. **Data Formats.** In this section we describe the matrix and vector formats used internally by Aztec. In Section 4 we discuss a tool that transforms data from a simpler format to this format. Here, the terms “element” and “component” are used interchangeably to denote a particular entry of a vector.

The sparse matrix-vector product, \( y \leftarrow Ax \), is the major kernel operation of Aztec. To perform this operation in parallel, the vectors \( x \) and \( y \) as well as the matrix \( A \) must be distributed across the processors. The elements of any vector of length \( n \) are assigned to a particular processor via some partitioning method (e.g. Chaco [2]). When calculating elements in a vector such as \( y \), a processor computes only those elements in \( y \) which it has been assigned. These vector elements are explicitly stored on the processor and are defined by a set of indices referred to as the processor’s update set. The update set is further divided into two subsets: internal and border. A component corresponding to an index in the internal set is updated using only information on the

---

2 All integer information returned from AZ_solve is cast into double precision and stored in *status*. 7
current processor. As an example, the index $i$ is in internal if, in the matrix-vector product kernel, the element $y_i$ is updated by this processor and if each $j$ defining a nonzero $A_{ij}$ in row $i$ is in update. The border set defines elements which would require values from other processors in order to be updated during the matrix vector product. For example, the index $i$ is in border if, in the matrix-vector product kernel, the element $y_i$ is updated by this processor and if there exists at least one $j$ associated with a nonzero $A_{ij}$ found in row $i$ that is not in update. In the matrix-vector product, the set of indices which identify the off-processor elements in $x$ that are needed to update components corresponding to border indices is referred to as external. They are explicitly stored by and are obtained from other processors via communication whenever a matrix-vector product is performed. Figure 3 illustrates how a set of vertices in a partitioning of a grid would be used to define these sets. Since these sets of indices are used exclusively to reference specific vector components, the same names (i.e., update, internal, border and external) are sometimes used below to describe the vector elements themselves. Having generalized these labels, the three types of vector elements are distinguished by locally storing the internal components first, followed by the border components and finally by the external components. In addition, all external components received from the same processor are stored consecutively. Below we summarize the nomenclature for a processor with $N$ total elements where $N_{\text{internal}}$, $N_{\text{border}}$, and $N_{\text{external}}$ elements are distributed over the sets internal, border and external respectively.

**Fig. 3.** Example partitioning of a finite element grid.
<table>
<thead>
<tr>
<th>set</th>
<th>description</th>
<th>local numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>updated w/o communication</td>
<td>0 to $N_{\text{internal}} - 1$.</td>
</tr>
<tr>
<td>border</td>
<td>updated with communication</td>
<td>$N_{\text{internal}}$ to $N_{\text{internal}} + N_{\text{border}} - 1$.</td>
</tr>
<tr>
<td>external</td>
<td>not updated but used to update border</td>
<td>$N_{\text{internal}} + N_{\text{border}}$ to $N - 1$. Elements received from the same processor are numbered consecutively.</td>
</tr>
</tbody>
</table>

Similar to vectors, a subset of matrix non-zeros is stored on each processor. In particular, each processor stores only those rows which correspond to its update set. For example, if vector element $i$ is updated on processor $p$, then processor $p$ also stores all the non-zeros of row $i$ in the matrix. Further, the local numbering of vector elements on a specific processor induces a local numbering of matrix rows and columns. For example, if vector element $k$ is locally numbered as $k_l$, then all references to row $k$ or column $k$ in the matrix would be locally numbered as $k_l$. Thus, each processor contains a submatrix whose row and column entries correspond to variables defined on this processor.

The remainder of this section describes the two sparse matrix formats that are used to store the local renumbered submatrix. These two sparse matrix formats correspond to common formats used in serial computations.

### 3.1. Distributed Modified Sparse Row (DMSR) Format

The DMSR format is a generalization of the MSR format [3]. The data structure consists of an integer vector $bindx$ and a double precision vector $val$ each of length $N_{\text{nonzeros}} + 1$ where $N_{\text{nonzeros}}$ is the number of nonzeros in the local submatrix. For a submatrix with $m$ rows the DMSR arrays are as follows:

**$bindx$:**

$$bindx[0] = m + 1,$$

$$bindx[k+1] - bindx[k] = \text{number of nonzero off-diagonal elements in } k^{\text{th}} \text{ row, } k < m,$$

$$bindx[k_s,...,k_e] = \text{column indices of the off-diagonal nonzeros in row } k_i \text{ where } k_s = bindx[k] \text{ and } k_e = bindx[k+1]-1.$$

**$val$:**

$$val[k] = A_{kk}, k < m,$$

$$val[k_i] = \text{the } (k, bindx[k_i])^{\text{th}} \text{ matrix element where } k_s \leq k_i \leq k_e \text{ with } k_s \text{ and } k_e \text{ as defined above.}$$

Note: $val[m]$ is not used. See [1] for a detailed discussion of the MSR format.

### 3.2. Distributed Variable Block Row (DVBR) Format

The Distributed Variable Block Row (DVBR) format is a generalization of the VBR format [1]. The data structure consists of a double precision vector $val$ and five integer vectors: $indx$, $bindx$, $rpntr$, $cpntr$ and $bpntr$. The format is best suited for sparse block matrices of
the form

\[ A = \begin{pmatrix}
A_{00} & A_{01} & \cdots & A_{0k} \\
A_{10} & A_{11} & \cdots & A_{1k} \\
\vdots & \ddots & \vdots \\
A_{m0} & \cdots & \cdots & A_{mk}
\end{pmatrix} \]

where \( A_{ij} \) denotes a block (or submatrix). In a sparse block matrix, some of these blocks would be entirely zero while others may be dense. The DVBR vectors are described below for a matrix with \( M \times K \) blocks.

\( rpntr[0 \ldots M] : \)
\[ rpntr[0] = 0 \]
\[ rpntr[k+1] - rpntr[k] = \text{number of rows in } k^{\text{th}} \text{ block row} \]

\( cpntr[0 \ldots K] : \)
\[ cpntr[0] = 0 \]
\[ cpntr[k+1] - cpntr[k] = \text{number of columns in } k^{\text{th}} \text{ block column} \]

\( bpntr[0 \ldots M] : \)
\[ bpntr[0] = 0 \]
\[ bpntr[k+1] - bpntr[k] = \text{number of nonzero blocks in the } k^{\text{th}} \text{ block row} \]

\( bindx[0 \ldots bpntr[M]] : \)
\[ bindx[k_s \ldots k_e] = \text{block column indices of nonzero blocks in block row } k \]
where \( k_s = bpntr[k] \) and \( k_e = bpntr[k+1]-1 \)

\( indx[0 \ldots bpntr[M]] : \)
\[ indx[0] = 0 \]
\[ indx[k_i+1] - indx[k_i] = \text{number of nonzeros in the } (k, bindx[k_i])^{\text{th}} \text{ block} \]
where \( k_s \leq k_i \leq k_e \) with \( k_s \) and \( k_e \) as defined above.

\( val[0 \ldots indx[bpntr[M]]] : \)
\[ val[i_s \ldots i_e] = \text{nonzeros in the } (k, bindx[k_i])^{\text{th}} \text{ block stored in column major order where } k_i \text{ is as defined above,} \]
\[ i_s = indx[k_i] \text{ and } i_e = indx[k_i+1]-1 \]

See [1] for a detailed discussion of the VBR format.

4. High Level Data Interface. Setting up the distributed format described in Section 3 for the local submatrix on each processor can be quite cumbersome. In particular, the user must determine a mapping between the global numbering scheme and a local scheme which facilitates proper communication. Further, a number of additional variables must be set for communication and synchronization (see Section 6). In this section we describe a simpler data format that is used in conjunction with a transformation function to generate data structures suitable for Aztec. The new format allows the user to specify the rows in a natural order as well as to use global column numbers in the bindx array. To use the transformation function the user supplies the
update set and the submatrix for each processor. Unlike the previous section, however, the submatrix is specified using the global coordinate numbering instead of the local numbering required by Aztec. This procedure greatly facilitates matrix specification and is the main advantage of the transformation software.

On a given processor, the update set (i.e. vector element assignment to processors) is defined by initializing the array update on each processor so that it contains the global index of each element assigned to the processor. The update array must be sorted in ascending order (i.e. \( i < j \Rightarrow \text{update}[i] < \text{update}[j] \)). This sorting can be performed using the Aztec function AZ_sort. Matrix specification occurs using the arrays defined in the previous section. However, now the local rows are defined in the same order as the update array and column indices (e.g. bindx) are given as global column indices. To illustrate this in more detail, consider the following example matrix:

\[
A = \begin{pmatrix}
  a_{00} & a_{01} & a_{03} & a_{04} \\
  a_{10} & a_{11} & a_{13} & \\
  & a_{22} & a_{23} & a_{24} & a_{25} \\
  a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
  a_{40} & a_{42} & a_{43} & a_{44} & \\
  & & a_{52} & a_{53} & a_{55}
\end{pmatrix}.
\]

Figure 4 illustrates the information corresponding to a particular matrix partitioning that is specified by the user as input to the data transformation tool. Using this

### Examples

**proc 0:**

- N_update: 3
- update: 0 1 3
- bindx: 4 7 9 14 1 3 4 0 3 1 0 4 2 5
- val: a00 a11 a33 - a01 a03 a04 a10 a13 a30 a31 a32 a34 a35

**proc 1:**

- N_update: 1
- update: 4
- bindx: 2 5 0 3 2
- val: a44 - a40 a42 a43

**proc 2:**

- N_update: 2
- update: 2 5
- bindx: 3 6 8 4 3 5 3 2
- val: a22 a55 - a23 a24 a25 a52 a53

**FIG. 4.** User input (MSR format) to initialize the sample matrix problem.

information, AZ_transform

- determines the sets internal, border and external.
- determines the local numbering: update_index[i] is the local numbering for update[i] while extern_index[i] is the local numbering for external[i].
permutes and renumbers the local submatrix rows and columns so that they now correspond to the new ordering.

- computes additional information (e.g. the number of internal, border and external components on this processor) and stores this in data_org (see Section 6).

A sample transformation is given in Figure 5 and is found in the file az_app_utils.c.

Example

```
init_matrix_vector_structures(bindx, val, update, external,
    update_index, extern_index, data_org);
{
    AZ_read_update(update, N_update);
    create_matrix(bindx, val, update, N_update);
    AZ_transform(bindx, val, update, external, update_index,
        extern_index, data_org, N_update);
}
```

Fig. 5. init_matrix_vector_structures.

AZ_read_update is an Aztec utility which reads a file and assigns elements to update. The user supplied routine create_matrix creates an MSR or VBR matrix using the global numbering. Once transformed the matrix can now be used within Aztec.

5. Examples. A sample program is described by completing the program fragments given earlier (Figures 1, 2 and 5). In Figure 1, AZ_processor_info is an Aztec utility which initializes the array proc_config to reflect the number of processors being used and the node number of this processor. The function AZ_solve is also supplied by Aztec to solve the user supplied linear system. Thus, the only functions that the user must supply which have not already been discussed include: init_guess_and_rhs in Figure 1 and create_matrix in Figure 5.

The function init_guess_and_rhs initializes the initial guess and the right hand side.

Example

```
void init_guess_and_rhs(x, rhs, data_org, update, update_index)
{
    N_update = data_org[AZ_N_internal] + data_org[AZ_N_border];
    for (i = 0; i < N_update ; i = i + 1) {
        rhs[update_index[i]] = (double) update[i];
        x[i] = 0.0;
    }
}
```

Fig. 6. init_guess_and_rhs.

In Figure 6, a sample routine is given which sets the initial guess vector to zero and sets the right hand side vector equal to the global indices (where the local element update_index[i] corresponds to global element update[i], see Section 4).
A create_matrix function to initialize an MSR matrix is illustrated in Figure 7. Different matrix problems can be implemented by changing the function add_row which computes the MSR entries corresponding to a new row of the matrix. The specific

Example

```c
void create_matrix(bindx, val, update, N_update)
{
    N_nonzeros = N_update + 1;
    bindx[0] = N_nonzeros;
    for (i = 0; i < N_update; i = i + 1)
        add_row(update[i], i, val, bindx);
}
```

**FIG. 7. create_matrix.**

add_row function for implementing a 5-point 2D Poisson operator on an n x n grid is shown in Figure 8 (n is a global variable set by the user). With these few lines of code

Example

```c
void add_row(row, location, val, bindx)
{
    k = bindx[location];
    /* check neighboring points in each direction and add nonzero */
    /* entry if neighbor exists. */
    bindx[k] = row + 1; if (row%n != n-1) val[k++] = -1.;
    bindx[k] = row - 1; if (row%n != 0) val[k++] = -1.;
    bindx[k] = row + n; if ((row/n)%n != n-1) val[k++] = -1.;
    bindx[k] = row - n; if ((row/n)%n != 0) val[k++] = -1.;

    bindx[location+1] = k;
    val[location] = 4.; /* matrix diagonal */
}
```

**FIG. 8. add_row for a 2D Poisson problem**

and the functions described earlier, the user initializes and solves a 2D Poisson problem. While for simplicity of presentation this specific example is structured the Aztec library does not assume any structure in the sparse matrix. All the communication and variable renumbering is done automatically without the assumption of structured communication.
Other add_row functions corresponding to a 3D Poisson equation and a high order 2D Poisson equation are distributed with Aztec (file az_examples.c). We recommend that potential users review at these examples. In many cases, new applications can be written by simply editing these programs. The interested reader should note that only a few lines of code are different between the functions for the 5-pt Poisson, the high order Poisson and the 3D Poisson codes. Further, the add_row routines are essentially identical to those that would be used to set up sparse matrices in serial applications and that there are no references to processors, communications or anything specific to parallel programming.

While Aztec simplifies the parallel coding associated with structured problems, it is for unstructured problems that Aztec makes a significant programming difference.

To illustrate this, a 2D finite element example is given where the underlying grid is a triangulation of a complex geometry. Unlike the previous example create_matrix defines a sparsity pattern (i.e. bindx) but not the actual nonzero entries (i.e. val) as interprocessor communication is required before they can be computed. Thus, in this example AZ_transform takes the sparsity pattern and initializes the communication data structures. Using these structures, communication can be performed at a later stage in computing the matrix nonzeros.

Figure 9 depicts create_matrix while Figure 10 depicts an additional function ma-

Example

```c
void create_matrix(bindx, val, update, N_update);
{
    read_triangles(T, N_triangles);
    init_msr(val, bindx, N_update);

    for (triangle = 0; triangle < N_triangles; triangle = triangle + 1)
        for (i = 0; i < 3; i = i + 1) {
            row = AZ_find_index(T[triangle][i], update, N_update);
            for (j = 0; j < 3; j = j + 1) {
                if (row != NOT_FOUND)
                    add_to_element(row, T[triangle][j], 0.0, val, bindx, i==j);  }  
        }  
    compress_matrix(val, bindx, N_update);  
}  
```

**FIG. 9.** create_matrix for the Poisson finite element problem.

trix_fill that must be included before AZ_solve is invoked in Figure 1. We have not made any effort to optimize these routines. In both figures the new lines that have been added specifically for a parallel implementation are underlined. That is, create_matrix and matrix_fill have been created by taking a serial program that creates the finite element discretization, splitting this program over the two functions and adding a few new lines necessary for the parallel implementation. The only additional change is to replace the single data file containing the triangle connectivity read using read_triangles
by a set of data files containing the triangle connectivity for each processor. We do not
discuss the details of this program but only wish to draw the readers attention to the
small number of lines that need changing to convert the serial unstructured application
to parallel. Most of the main routines such as setup_Ke which computes the element
contributions and add_to_element which stores the element contributions in the MSR
data structures remain the same. In fact, almost all the new lines of code correspond
to adding the communication (AZ_exchange_bdry) (which was the main reason that the
calculation of the matrix nonzeros was deferred) and the conversion of global index
values by local index values with the help of AZ_find_index. As in the Poisson example,
all of the details with respect to communication are hidden from the user.

6. Advanced Topics.

6.1. Data Layout. The Aztec function AZ_transform initializes the integer array
data_org. This array specifies how the matrix is set up on the parallel machine. In many
cases, the user need not be concerned with the contents of this array. However, in some
situations it is useful to initialize these elements without the use of AZ_transform, to
access these array elements (e.g. determine how many internal components are used),
or to change these array elements (e.g. when reusing factorization information, see
Section 6.2). When using the transformation software, the user can ignore the size of
data_org as it is allocated in AZ_transform. However, when this is not used, data_org
must be allocated of size AZ_COMM_SIZE + number of vector elements sent to other
processors during matrix-vector multiplies. The contents of data_org are as follows:

<table>
<thead>
<tr>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_org[AZ_matrix_type]</td>
</tr>
<tr>
<td>AZ_VBR_MATRIX</td>
</tr>
<tr>
<td>AZ_MSR_MATRIX</td>
</tr>
<tr>
<td>data_org[AZ_N_internal]</td>
</tr>
<tr>
<td>data_org[AZ_N_border]</td>
</tr>
<tr>
<td>data_org[AZ_N_external]</td>
</tr>
<tr>
<td>data_org[AZ_N_int_blk]</td>
</tr>
<tr>
<td>data_org[AZ_N_bord_blk]</td>
</tr>
</tbody>
</table>
Example

```c
void matrix_fill(bindx, val, N_update, update, update_index,
    N_external, external, extern_index)

    /* read the x and y coordinates from an input file */

    for (i = 0; i < N_update; i = i + 1)
        read_from_file(x[update_index[i]], y[update_index[i]]);
    AZ_exchange_bdry(x);
    AZ_exchange_bdry(y);

    /* Locally renumber the rows and columns of the new sparse matrix */

    for (triangle = 0; triangle < N_triangles; triangle = triangle + 1)
        for (i = 0; i < 3; i = i + 1)
            row = AZ_find_index(T[triangle][i], update, N_update);
            if (row == NOT_FOUND) {
                row = AZ_find_index(T[triangle][i], external, N_external);
                T[triangle][i] = extern_index[row];
            } else T[triangle][i] = update_index[row];
    }

    /* Fill the element stiffness matrix Ke */

    for (triangle = 0; triangle < N_triangles; triangle = triangle + 1){
        setup_Ke(Ke, x[T[triangle][0]], y[T[triangle][0]],
            x[T[triangle][1]], y[T[triangle][1]],
            x[T[triangle][2]], y[T[triangle][2]]);
    }

    /* Fill the sparse matrix by scattering Ke to appropriate locations */

    for (i = 0; i < 3; i = i + 1) {
        for (j = 0; j < 3; j = j + 1){
            if (T[triangle][i] < N_update){
                add_to_element(T[triangle][i], T[triangle][j], Ke[i][j],
                    val, bindx, i==j);
            }
        }
    }
```

FIG. 10. matrix_fill for the Poisson finite element problem.
**6.2. Reusing factorizations.** When solving a problem, Aztec may create certain information that can be reused later. In most cases, this information corresponds to either matrix scaling factors or preconditioning factorization information for LU or ILU. This information is saved internally and referenced by the matrix name given by `data.org[AZ_name]`. By changing `options[AZ_pre_calc]` and `data.org[AZ_name]` a number of different Aztec possibilities can be realized. As an example, consider the following situation. A user needs to solve the linear systems in the order shown below:

\[
A_1x = b, \quad A_2y = x, \quad \text{and} \quad A_3z = y.
\]

The first and second systems are solved with `options[AZ_pre_calc]` set to `AZ_calc`. However, the name (i.e. `data.org[AZ_name]`) is changed between these two solves. In this way, scaling and preconditioning information computed from the first solve is not overwritten during the second solve. By then setting `options[AZ_pre_calc]` to `AZ_reuse` and `data.org[AZ_name]` to the name used during the first solve, the third system is solved reusing the scaling information (to scale the right hand side, initial guess, and rescale
the final solution\(^3\) and the preconditioning factorizations (e.g. ILU) used during the first solve. While in this example the same matrix system is solved for the first and third solve, this is not necessary. In particular, preconditioners can be reused from previous nonlinear iterates even though the linear system being solved are changing. Of course, many times information from previous linear solves is not reused. In this case the user must explicitly free the space associated with the matrix or this information will remain allocated for the duration of the program. Space is cleared by invoking AZ\_clear(data\_org/AZ\_name).

6.3. Important Constants. Aztec uses a number of constants which are defined in the file az\_aztec\_defs.h. Most users can ignore these constants. However, there may be situations where they should be changed. Below is a list of these constants with a brief description:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ_MAX_NEIGHBORS</td>
<td>Maximum number of processors with which information can be exchanged during matrix-vector products.</td>
</tr>
<tr>
<td>AZ_MSG_TYPE</td>
<td>All message types used inside Aztec lie between AZ_MSG_TYPE and AZ_MSG_TYPE + AZ_NUM_MSGS - 1.</td>
</tr>
<tr>
<td>AZ_NUM_MSGS</td>
<td>Maximum message information that can be sent by any processor at any given time before receiving. This is used to subdivide large messages to avoid buffer overflows.</td>
</tr>
<tr>
<td>AZ_MAX_BUFFER_SIZE</td>
<td>Maximum available memory. Used primarily for the LU-factorizations where a large amount of memory is first allocated and then unused portions are freed after factorization.</td>
</tr>
<tr>
<td>AZ_TEST_ELE</td>
<td>Internal algorithm parameter that can effect the speed of the AZ_find_procs_for_externs calculation. Reduce AZ_TEST_ELE if communication buffers are exceeded during this calculation.</td>
</tr>
</tbody>
</table>

6.4. AZ\_transform Subtasks. The function AZ\_transform described in Section 4 is actually made up of 5 subtasks. In most cases the user need not be concerned with the individual tasks. However, there might arise situations where additional information is available such that some of the subtasks can be omitted. In this case, it is possible for the user to edit the code for AZ\_transform located in the file az\_tools.c to suit the application. In this section we briefly describe the five subroutines which make up the transformation function. More detailed descriptions are given in [5]. Prototypes for these subroutines as well as for AZ\_transform are given in Section 7.

AZ\_transform begins by identifying the external set needed by each processor. Here, each column entry must correspond to either an element updated by this processor or the matrix does not need to be rescaled as the scaling during the first solve overwrites the original matrix.

\(^3\) The matrix does not need to be rescaled as the scaling during the first solve overwrites the original matrix.
an external component. The function AZ_find_local_indices checks each column entry. If a column is in update, its number is replaced by the appropriate index into update (i.e. update[new column index] = old column index). If a column number is not found in update, it is stored in the external list and the column number is replaced by an index into external (i.e. external[new column index - N_update] = old column index).

AZ_find_procs_forExterns queries the other processors to determine which processors update each of its external components. The array extern_proc is set such that extern_proc[i] indicates which processor updates external[i].

AZ_order_ele reorders the external components such that elements updated by the same processor are contiguous. This new ordering is given by extern_index where extern_index[i] indicates the local numbering of external[i]. Additionally, update components are reordered so the internal components precede the border components. This new ordering is given by update_index where update_index[i] indicates the local numbering of update[i].

AZ_set_message_info initializes data_org (see Section 6.1) This is done by computing the number of neighbors, making a list of the neighbors, computing the number of values to be sent and received with each neighbor and computing the list of elements which will be sent to other processors during communication steps.

Finally, AZ_reorder_matrix permutes and reorders the matrix nonzeros so that its entries correspond to the newly reordered vector elements.

7. Aztec Functions. In this section we describe the Aztec functions available to the user. Certain variables appear many times in the parameter lists of these frequently used functions. In the interest of brevity we describe these variables at the beginning of this section and then proceed with the individual function descriptions.

Frequently Used Aztec Parameters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_org</td>
<td>Array describing the matrix format (Section 6.1). Allocated and set AZ_set_message_info and AZ_transform.</td>
</tr>
<tr>
<td>extern_index</td>
<td>extern_index[i] gives the local numbering of global element external[i]. Allocated and set by AZ_order_ele and AZ_transform.</td>
</tr>
<tr>
<td>extern_proc</td>
<td>extern_proc[i] is updating processor of external[i]. Allocated and set by AZ_find_procs_forExterns.</td>
</tr>
<tr>
<td>external</td>
<td>Sorted list (global indices) of external elements on this node. Allocated and set by AZ_find_local_indices and AZ_transform.</td>
</tr>
<tr>
<td>N_external</td>
<td>Number of external components. Set by AZ_find_procs_forExterns and AZ_transform.</td>
</tr>
<tr>
<td>N_update</td>
<td>Number of update components assigned to this processor. Set by AZ_read_update.</td>
</tr>
<tr>
<td>options, params</td>
<td>Arrays describing AZ_solve options (Section 2).</td>
</tr>
</tbody>
</table>
**proc_config[AZ.node]**
Node i.d. of this processor.

**proc_config[AZ.N.proc]**
Total number of processors used in current simulation. Allocated and set by AZ.processor_info.

**update_index**
update_index[i] gives the local numbering of global element update[i]. Allocated and set by AZ.order.ele and AZ.transform.

**update**
Sorted list of elements (global indices) to be updated on this processor. Allocated and set by AZ.read.update.

**val, bindx, bpntr, cpntr, indx, rpntr**
Arrays used to store matrix. For MSR matrices bpntr, cpntr, indx, rpntr are ignored (Section 3).

---

**Prototype**

```c
void AZ_broadcast(char *ptr, int length, int *proc_config, int action)
```

**Description**

Used to concatenate a buffer of information and to broadcast this information from processor 0 to the other processors. The four possible actions are

- **action == AZ_PACK**
  - proc_config[AZ.node] == 0: store ptr in the internal buffer.
  - proc_config[AZ.node] ≠ 0: read from the internal buffer to ptr. If the internal buffer is empty, first receive the broadcast information.

- **action == AZ_SEND**
  - proc_config[AZ.node] == 0: broadcast the internal buffer (filled by AZ.broadcast) and then clear it.
  - proc_config[AZ.node] ≠ 0: clear internal buffer.

**Sample Usage:**

The following code fragment broadcasts the information in ‘a’ and ‘b’.

```c
if (proc_config[AZ_node] == 0) {
    a = 1;
    b = 2;
}
AZ_broadcast(&a, sizeof(int), proc_config, AZ_PACK);
AZ_broadcast(&b, sizeof(int), proc_config, AZ_PACK);
AZ_broadcast(NULL, 0, proc_config, AZ_SEND);
```
NOTE: There can be no other communication calls between the **AZ_PACK** and **AZ_SEND** calls to **AZ_broadcast**.

### Parameters

- **ptr**
  - On input, data string of size *length*. Information is either stored to or retrieved from *ptr* as described above.

- **length**
  - On input, length of *ptr* to be broadcast/received.

- **action**
  - On input, determines **AZ_broadcast** behavior.

### Prototype

```c
int AZ_check_input(int *data_org, int *options, double *params, int *proc_config)
```

### Description

Perform checks for iterative solver library. This is to be called by the user of the solver library to check the values in *data_org*, *options*, *params*, and *proc_config*. If all the values are valid **AZ_check_input** returns 0, otherwise it returns an error code which can be deciphered using **AZ_print_error**.

### Prototype

```c
void AZ_check_msr(int *bindx, int N_update, int N_external, int option,
                   int *proc_config)
```

### Description

Check that the number of nonzero off-diagonals in each row and that the column indices are nonnegative and not too large (see *option*).
AZ_LOCAL
On input, indicates matrix uses local indices. The number of nonzeros in a row and the largest column index must not exceed the total number of elements on this processor.

AZ_GLOBAL
On input, indicates matrix uses global indices. The number of nonzeros in a row and the largest column index must not exceed the total number of elements in the simulation.

Prototype

```c
void AZ_check_vbr(int N_update, int N_external, int option, int *bindz,
                   int *bpntr, int *cpntr, int *rpntr, int *proc_config )
```

Description

Check VBR matrix for the following:
- number of columns within each block column is nonnegative.
- \( rpntr[i] == cpntr[i] \) for \( i \leq N_{update} \).
- number of nonzero blocks in each block row is nonnegative and not too large.
- block column indices are nonnegative and not too large.

Parameters

`option`

- **AZ_LOCAL**
  On input, indicates matrix uses local indices. The number of block nonzeros in a row and the largest block column index must not exceed the total number of blocks columns on this processor.

- **AZ_GLOBAL**
  On input, indicates matrix uses global indices. The number of block nonzeros in a row and the largest block column index must not exceed the total number of blocks rows in the simulation.

Prototype

```c
int AZ_defaults(double *options, int *params )
```

Description
Set *options* and *params* so that the default options are chosen.

### Parameters

- **options**
  - On output, set to the default options.
- **params**
  - On output, set to the default parameters.

### Prototype

```c
void AZ_exchange_bdry(double *x, int *data_org)
```

### Description

Locally exchange the components of the vector \( x \) so that the *external* components of \( x \) are updated.

### Parameters

- **\( x \)**
  - On input, vector defined on this processor. On output, *external* components of \( x \) are updated via communication.

### Prototype

```c
int AZ_find_index(int key, int *list, int length)
```

### Description

Returns the index, \( i \), in \( list \) (assumed to be sorted) which matches the key (i.e. \( list[i] == key \)). If *key* is not found AZ_find_index returns -1. See also AZ_quick_find.

### Parameters

- **key**
  - On input, element to be search for in list.
- **list**
  - On input, sorted list to be searched.
- **length**
  - On input, length of list.
Prototype

```c
void AZ_find_local_indices(int N_update, int *bindx, int *update,
    int **external, int *N_external, int mat_type,
    int *bpnt)
```

Description

Given the global column indices for a matrix and a list of elements updated on this processor, compute the external set and change the global column indices to local column indices. Specifically,

- allocate external, compute and store the external components in external.
- renumber column indices so that column entry $k$ is renumbered as $j$ where either $update[j] == k$ or $external[j-N\_update] == k$.

Called by AZ\_transform.

Parameters

- **mat\_type**
  - On input, indicates whether matrix format is MSR (= AZ\_MSR\_MATRIX) or VBR (= AZ\_VBR\_MATRIX).
- **external**
  - On output, allocated and set to sorted list of the external elements.
- **bindx**
  - On input, contains global column numbers of MSR or VBR matrix (Section 3). On output, contains local column numbers as described above.

Prototype

```c
void AZ_find_procs_forExterns(int N_update, int *update, int *external,
    int N_external, int *proc_config, int **extern_proc)
```

Description

Determine which processors are responsible for updating each external element.

Called by AZ\_transform.

Parameters

- **extern_proc**
  - On output, extern\_proc[i] contains the node number of the processor which updates external[i].
Prototype

```c
void AZ_free_memory(int name)
```

Description

Free Aztec memory associated with matrices with `data_org[AZ_name] = name`. This is primarily scaling and preconditioning information that has been computed on earlier calls to `AZ_solve`.

Parameters

- `name` On output, all preconditioning and scaling information is freed for matrices which have `data_org[AZ_name] = name`.

Prototype

```c
double AZ_gavg_double(double value, int *proc_config)
```

Description

Return the average of the numbers in `value` on all processors.

Parameters

- `value` On input, `value` contains a double precision number.

Prototype

```c
double AZ_gdot(int N, double *r, double *z, int *proc_config)
```

Description

Return the dot product of `r` and `z` with unit stride. This routine calls the BLAS routine `ddot` to do the local vector dot product and then uses the global summation routine `AZ_gsum_double` to obtain the required global result.
Parameters

\[ N \]
On input, length of \( r \) and \( z \) on this processor.

\[ r, z \]
On input, vectors distributed over all the processors.

Prototype

\[
\text{double AZ_gmax_double(double } value, \text{ int } \ast \text{proc_config })
\]

Description

Return the maximum of the numbers in \( value \) on all processors.

Parameters

\[ value \]
On input, \( value \) contains a double precision number.

Prototype

\[
\text{int AZ_gmax_int(int } value, \text{ int } \ast \text{proc_config })
\]

Description

Return the maximum of the numbers in \( value \) on all processors.

Parameters

\[ value \]
On input, \( value \) contains an integer.

Prototype

\[
\text{double AZ_gmax_matrix_norm(double } \ast \text{val, int } \ast \text{indx, int } \ast \text{bindx, int } \ast \text{rpntr, int } \ast \text{cpntr,}
\text{ int } \ast \text{bpntr, int } \ast \text{proc_config, int } \ast \text{data_org})
\]
Description

Returns the maximum matrix norm $\|A\|_\infty$ for the distributed matrix encoded in $val$, $indx$, $bindx$, $rpntr$, $cpntr$, $bpntr$ (Section 3).

Prototype

```c
double AZ_gmax_vec(int N, double *vec, int *proc_config )
```

Description

Return the maximum of all the numbers located in $vec[i]$ ($i < N$) on all processors.

Parameters

- $vec$  
  On input, $vec$ contains a list of numbers.
- $N$  
  On input, length of $vec$.

Prototype

```c
double AZ_gmin_double(double value, int *proc_config )
```

Description

Return the minimum of the numbers in $value$ on all processors.

Parameters

- $value$  
  On input, $value$ contains a double precision number.

Prototype

```c
int AZ_gmin_int(int value, int *proc_config )
```
Return the minimum of the numbers in \( value \) on all processors.

**Parameters**

- \( value \)  
  On input, \( value \) contains an integer.

**Prototype**

\[
\text{double AZ\_gsum\_double(double } value, \text{ int } \ast \text{proc\_config })
\]

**Description**

Return the sum of the numbers in \( value \) on all processors.

**Parameters**

- \( value \)  
  On input, \( value \) contains a double precision number.

**Prototype**

\[
\text{int AZ\_gsum\_int(int } value, \text{ int } \ast \text{proc\_config })
\]

**Description**

Return the sum of the integers in \( value \) on all processors.

**Parameters**

- \( value \)  
  On input, \( value \) contains an integer.

**Prototype**

\[
\text{void AZ\_gsum\_vec\_int(int } \ast \text{values, int } \ast \text{wkspace, int } length, \text{ int } \ast \text{proc\_config })
\]
values[i] is set to the sum of the input numbers in values[i] on all processors (i < length).

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>On input, values contains a list of integers. On output, values[i] contains the sum of the input values[i] on all the processors.</td>
</tr>
<tr>
<td>workspace</td>
<td>On input, workspace array of size length.</td>
</tr>
<tr>
<td>length</td>
<td>On input, length of values and workspace.</td>
</tr>
</tbody>
</table>

Prototype

```c
double AZ_gvector_norm(int n, int p, double *x, int *proc_config)
```

Description

Returns the p norm of the vector x distributed over the processors:

\[ \|x\|_p = (x[0]^p + x[1]^p + \cdots + x[N-1]^p)^{1/p} \]

where \(N\) is the total number of elements in \(x\) over all processors. NOTE: For the \(\| \cdot \|_\infty\) norm, set \(p = -1\).

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>On input, number of update components of (x) on this processor.</td>
</tr>
<tr>
<td>p</td>
<td>On input, order of the norm to perform, i.e., (|x|_p).</td>
</tr>
<tr>
<td>x</td>
<td>On input, vector whose norm will be computed.</td>
</tr>
</tbody>
</table>

Prototype

```c
void AZ_init_quick_find(int *list, int length, int *shift, int *bins)
```
**Description**

*shift* and *bins* are set so that they can be used with AZ_quick_find. On output, *shift* satisfies

\[
\frac{\text{range}}{2^{\text{shift}-1}} \geq \left\lfloor \frac{\text{length}}{4} \right\rfloor \quad \text{and} \quad \frac{\text{range}}{2^{\text{shift}}} \leq \left\lfloor \frac{\text{length}}{4} \right\rfloor
\]

where \( \text{range} = \text{list[length - 1]} - \text{list[0]} \). The array *bins* must be of size \( 2 + \text{length}/4 \) and is set so that

\[
\text{bins}[k] \leq \text{list}[j] < \text{bins}[k+1]
\]

where \( k = (\text{list}[j] - \text{list}[0])/2^{\text{shift}} \).

This routine is used in conjunction with AZ_quick_find. The idea is to use *bins* to get a good initial guess as to the location of *value* in *list*.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>list</em></td>
<td>On input, sorted <em>list</em>.</td>
</tr>
<tr>
<td><em>length</em></td>
<td>On input, length of <em>list</em>.</td>
</tr>
<tr>
<td><em>shift</em></td>
<td>On output, <em>shift</em> is set as described above.</td>
</tr>
<tr>
<td><em>bins</em></td>
<td>On input, array of size ( 2 + \text{length}/4 ). On output, <em>bins</em> is set as described above.</td>
</tr>
</tbody>
</table>

**Prototype**

```c
void AZ_matvec_mult(double *val, int *indx, int *bindx, int *rpntr, int *cpntr,
                    int *bpntr, double *b, double *c, int exchange_flag,
                    int *data_org)
```

**Description**

Perform the matrix-vector multiply

\[ c \leftarrow Ab \]

where the matrix \( A \) is encoded in *val*, *indx*, *bindx*, *rpntr*, *cpntr*, *bpntr* (Section 3).

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>b</em></td>
<td>On input, distributed vector to use in multiplication.</td>
</tr>
</tbody>
</table>
On output, result of matrix-vector multiplication.

`exchange_flag`  
On input, dictates whether communication needs to occur. If `exchange_flag == 1`, communication occurs. If `exchange_flag == 0`, no communication occurs.

**Prototype**

```c
void AZ_msr2vbr(double *val, int *indx, int *rpntr, int *cpntr, int *bpntr, int *bindx,
                 int *bindx2, double *val2, int total_blk_rows, int total_blk_cols,
                 int blk_space, int nz_space, int blk_type)
```

**Description**

Convert the DMSR matrix defined in `(val2, bindx2)` to a DVBR matrix defined in `(val, indx, rpntr, cpntr, bpntr, bindx)`.

**Parameters**

- `val2, bindx2`  
  On input, DMSR arrays holding the matrix to be converted.

- `cpntr`  
  On input, `cpntr[i]` is the block size of the `i`th block in the resulting DVBR matrix. Columns 0 to `cpntr[0]` - 1 form the first block column, columns `cpntr[0]` to `cpntr[0] + cpntr[1]` - 1 form the second block column, etc. On output, `cpntr` corresponds to the resulting DVBR matrix.

- `val, indx, rpntr, bpntr, bindx`  
  On output, DVBR arrays of converted DMSR matrix.

- `total_blk_rows`  
  On input, number of block rows in resulting local VBR matrix.

- `total_blk_cols`  
  On input, number of block columns in resulting local VBR matrix.

- `blk_space`  
  On input, length allocated for `bindx` and `indx`.

- `nz_space`  
  On input, length allocated for `val`.

- `blk_type`  
  On input, if `blk_type > 0`, indicates that all block rows (and columns) have the same size given by `blk_type`. If `blk_type < 0`, the block rows have different sizes.
Prototype

```c
void AZ_order_ele(int *update_index, int *extern_index, int *N_internal,
                  int *N_border, int N_update, int *bpntr, int *bindx,
                  int *extern_proc, int N_external, int option, int mat_type)
```

Description

Find orderings for update and external. external are ordered so that elements updated by the same processor are contiguous. If option == AZ_ALL, update are ordered so that the internal components have the lowest numbers followed by the border components. Otherwise, the order of update is unchanged. The ordering information is placed in update_index and extern_index (Section 4). Called by AZ_transform.

Parameters

- **N_internal**: On output, number of internal components on processor.
- **N_border**: On output, number of border components on processor.
- **update_index**: On output, update_index[i] indicates the local index (or order) of update[i].
- **extern_index**: On output, extern_index[i] indicates the new local index (or order) of external[i].
- **option**: On input, indicates whether to reorder update.
  - AZ_ALL: Order update and external.
  - AZ EXTERNS: Order only external elements.
- **mat_type**: On input, indicates whether matrix format is MSR (= AZ_MSR_MATRIX) or VBR (= AZ_VBR_MATRIX).

Prototype

```c
void AZ_print_error(int error_code)
```

Description

32
Prints out an error message corresponding to \textit{error\_code}. Typically, \textit{error\_code} is generated by \textit{AZ\_check\_input}.

**Parameters**

\begin{itemize}
  \item \textit{error\_code} \quad On input, error code generated by \textit{AZ\_check\_input}.
\end{itemize}

**Prototype**

\begin{verbatim}
void AZ\_processor\_info(int *proc\_config)
\end{verbatim}

**Description**

\texttt{proc\_config[AZ\_node]} is set to the node name of this processor. \texttt{proc\_config[AZ\_N\_proc]} is set to the number of processors used in simulation.

**Prototype**

\begin{verbatim}
int AZ\_quick\_find(int key, int *list, int length, int shift, int *bins)
\end{verbatim}

**Description**

Return the index, \texttt{i}, in \texttt{list} (assumed to be sorted) which matches the key (i.e. \texttt{list[i] = key}). If \texttt{key} is not found \texttt{AZ\_quick\_find} returns -1. 

\textbf{NOTE:} This version is faster than \textit{AZ\_find} but requires \texttt{bins} to be set and stored using \textit{AZ\_init\_quick\_find}.

**Parameters**

\begin{itemize}
  \item \texttt{key} \quad On input, element to search for in \texttt{list}.
  \item \texttt{list} \quad On input, sorted list to be searched.
  \item \texttt{length} \quad On input, length of list.
  \item \texttt{shift} \quad On input, used for initial guess (computed by previous \textit{AZ\_init\_quick\_find} call).
\end{itemize}
On input, computed by AZ_init_quick_find for initial guess. \( \text{bins} \) is set so that \( \text{list[bins[k]]} \leq \text{key} < \text{list[bins[k + 1]]} \) where \( k = (\text{key} - \text{list[0]})/2^{\text{shift}} \).

Prototype

```c
void AZ_read_msr_matrix(int *update, double **val, int **bindx, int N_update,
                        int *proc_config )
```

Description

Read the file .data and create a matrix in the MSR format. Processor 0 reads the input file. If the new row to be added resides in processor 0’s \textit{update}, it is added to processor 0’s matrix. Otherwise, processor 0 determines which processor has requested this row and sends it to this processor for its local matrix.

The form of the input file is as follows:

```
num_rows
col_num1 entry1 col_num2 entry2
col_num3 entry3 -1
col_num4 entry4 col_num5 entry5
col_num6 entry6 -1
```

This input corresponds to two rows: 0 and 1. Row 0 contains entry1 in column \textit{col_num1}, entry2 in column \textit{col_num2} and entry3 in column \textit{col_num3}. Row 1 contains entry4 in column \textit{col_num4}, entry5 in column \textit{col_num5} and entry6 in column \textit{col_num6}.

NOTE: row and column numbers must start from 0.

NOTE: AZ_read_msr_matrix() is inefficient for large matrices.

Parameters

\textit{val, bindx} On output, these two arrays are allocated and filled with the MSR representation corresponding to the file .data.

Prototype

```c
void AZ_read_update(int *N_update, int **update, int *proc_config,
                    int N, int chunk, int input_option )
```
Description

This routine initializes update to the global indices updated by this processor and initializes N-update to the total number of elements to be updated.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-update</td>
<td>On output, number of elements updated by processor.</td>
</tr>
<tr>
<td>update</td>
<td>On output, update is allocated and contains a list of elements updated by this processor in ascending order.</td>
</tr>
<tr>
<td>chunk</td>
<td>Number of indices within a group. For example, chunk == 2 =&gt; chunk₀ = {0,1}, and chunk₁ = {2,3}.</td>
</tr>
<tr>
<td>N</td>
<td>Total number of chunks in the vector.</td>
</tr>
</tbody>
</table>

input_option

| AZ_LINEAR | Processor 0 is assigned the first \(\left\lfloor \frac{N+P-1}{P} \right\rfloor\) chunks, processor 1 is assigned the next \(\left\lfloor \frac{N+P-2}{P} \right\rfloor\) chunks, etc. where \(P = \text{proc_config}[AZ.N.proc]\). |
| AZ_BOX | The processor system is viewed as a \(p_2 \times p_1 \times p_0\) where \(p_i = 2^{(k+i)/3}\) (so \(\text{proc_config}[AZ.N.proc]\) must equal \(2^k\)). The chunks are viewed as an \(n \times n \times n\) cube where \(n\) is divisible by each \(p_i\). Chunks are distributed into uniform boxes such that each processor has the same number of chunks. |
| AZ_FILE | Read the \(\text{proc_config}[AZ.N.proc]\) lists contained in the file update. Each list contains a set of global indices preceded by the number of indices in this set. List 0 is sent to processor \(\text{proc_config}[AZ.N.proc] - 1\), list 1 is sent to processor \(\text{proc_config}[AZ.N.proc] - 2\), etc. Note: A graph partitioning package named Chaco [2] produces files in this format. |

Prototype

```c
void AZ_reorder_matrix(int N_update, int *bindx, double *val, int *update_index,
                        int *extern_index, int *indx, int *rpntr, int *bpntr,
                        int N_external, int *cpntr, int option, int mat_type)
```
Reorder the matrix so that it corresponds to the new ordering given by `update_index` and `extern_index`. Specifically, global matrix entry \((update[i], update[j])\) which was stored as local matrix entry \((i, j)\) is stored as \((update_index[i], update_index[j])\) on output. Likewise, global matrix entry \((update[i], external[k])\) which was stored as local matrix entry \((i, k + N\_update)\) is stored locally as \((update_index[i], extern_index[k])\) on output. Called by `AZ\_transform`.

IMPORTANT: This routine assumes that `update_index` contains two sequences of numbers that are ordered but intertwined. For example,

```
update_index: 4 5 0 6 1 2 3 7
sequence 1: 0 1 2 3
sequence 2: 4 5 6 7
```

### Parameters

**option**

On input, indicates whether to reorder update elements.

- **AZ\_ALL**
  All the rows and columns are renumbered.

- **AZ\_EXTERN**
  Only columns corresponding to external elements are renumbered.

**mat\_type**

On input, indicates matrix format.

- **AZ\_MSR\_MATRIX**
  DMSR matrix format.

- **AZ\_VBR\_MATRIX**
  DVBR matrix format.

**bindx, val, indx, rptr, bptr, cpntr**

On input, matrix ordered as described above. On output, matrix reordered using `update_index` and `extern_index` as described above.

### Prototype

```c
void AZ_set_message_info(int N\_external, int *extern\_index, int N\_update, int *external, int *extern\_proc, int *update, int *update\_index, int *proc\_config, int *cpntr, int **data\_org, int mat\_type)
```

### Description

Initialize `data\_org` so that local communications can occur to support matrix vector products. This includes:
• determine neighbors with which we send or receive.
• determine the total number of elements that we send and allocate data_org.
• initialize data_org as described in Section 6.1.
  Note: data_org[AZ.name] is set to a number (starting from 1) that is incremented each time AZ_set_message_info is called.

Called by AZ_transform.
NOTE: Implicitly the neighbors are numbered using the ordering of the external elements (which have been previously ordered such that elements updated by the same processor are contiguous).

Parameters

- **data_org**
  On output, data_org is allocated and completely initialized as described in Section 6.1.

- **mat_type**
  On input, indicates matrix format.

  - AZ_MSR_MATRIX: DMSR matrix.
  - AZ_VBR_MATRIX: DVBR matrix.

Prototype

```c
void AZ_solve(double *E, double *b, int *options, double *params, int *indx,
               int *bindx, int *rpntr, int *cpntr, int *bpntr, double *val,
               int *data_org, double *status, int *proc_config)
```

Description

Solve the system of equations \(Ax = b\) via an iterative method where the matrix \(A\) is encoded in \(indx, bindx, rpntr, cpntr, bpntr\) and \(val\) (see Section 3 and Section 2).

Parameters

- **x**
  On input \(x\) contains the initial guess. On output \(x\) contains the solution to linear system.

- **b**
  Right hand side of linear system.

- **options, params**
  Options and parameters used during the solution process (Section 2).

- **status**
  On output, status of iterative solver (Section 2).
Prototype

```c
void AZ_sort(int *list1, int N, int *list2, double *list3 )
```

Description

Sort the elements in list1. Additionally, move the elements in list2 and list3 so that they correspond with the moves done to list. NOTE: If list2 == NULL, list2 is not manipulated. If list3 == NULL, list3 is not manipulated.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list1</td>
<td>On input, values to be sorted. On output, sorted values (i.e. list1[i] ≤ list1[i+1])</td>
</tr>
<tr>
<td>N</td>
<td>On input, length of lists to be sorted.</td>
</tr>
<tr>
<td>list2</td>
<td>On input, a list associated with list1. On output, if list1[k] on input is now stored in list1[j] on output, list2[k] on input is also stored as list2[j] on output.</td>
</tr>
<tr>
<td>list3</td>
<td>On input, a list associated with list1. On output, if list1[k] on input is now stored in list1[j] on output, list3[k] on input is also stored as list3[j] on output. Note: if list3 == NULL on input, it is unchanged on output.</td>
</tr>
</tbody>
</table>

Prototype

```c
void AZ_transform(int *proc_config, int **external, int *bindx, double *val, int *update, int **update_index, int **extern_index, int **data_org, int N_update, int *indx, int *bpntr, int *rpntr, int **cpntr, int mat_type)
```

Description

Convert the global matrix description to a distributed local matrix format (see Section 2 and Section 6.4).

Parameters
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>external</code></td>
<td>On output, allocated and set to components that must be communicated during the matrix vector multiply.</td>
</tr>
<tr>
<td><code>bindx, val, index, bpntr, rpntr</code></td>
<td>On input, matrix arrays (MSR or VBR) corresponding to global format. On output, matrix arrays (DMSR or DVBR) corresponding to local format. See Section 2.</td>
</tr>
<tr>
<td><code>update_index</code></td>
<td>On output, allocated and set such that <code>update_index[i]</code> is the local numbering corresponding to <code>update[i]</code>.</td>
</tr>
<tr>
<td><code>extern_index</code></td>
<td>On output, allocated and set such that <code>extern_index[i]</code> is the local numbering corresponding to <code>external[i]</code>.</td>
</tr>
<tr>
<td><code>data_org</code></td>
<td>On output, allocated and set to data layout information, see Section 6.1.</td>
</tr>
<tr>
<td><code>cpntr</code></td>
<td>On output, allocated and set for VBR matrices to the column pointer array.</td>
</tr>
<tr>
<td><code>mat_type</code></td>
<td>On input, matrix format: either AZ_VBR_MATRIX or AZ_MSR_MATRIX.</td>
</tr>
</tbody>
</table>
REFERENCES


Fred Howes  
US Department of Energy  
OSC, ER-30, GTN  
Washington, DC 20585

Christopher R. Johnson  
Department of Computer Science  
3484 MEB  
University of Utah  
Salt Lake City, UT 84112

David Keyes  
Dept. of Mechanical Engineering  
Yale University  
PO Box 2159, Yale Station  
New Haven, CT 06520-2159

David Kincaid  
Center for Numerical Analysis  
RLM 13.150  
University of Texas  
Austin, TX  78713-8510

T. A. Kitchens  
US Department of Energy  
OSC, ER-30, GTN  
Washington, DC 20585

Vipin Kumar  
Computer Science Department  
Institute of Technology  
200 Union Street S.E.  
Minneapolis, MN 55455

Joanna Lees  
Intel Corp.  
Scalable Systems Division  
CO1-15  
15201 NW Greenbrier Parkway  
Beaverton, OR 97006

John Lewis  
Boeing Corp.  
M/S 7L-21  
P.O. box 24346  
Seattle, WA 98124-0346

T. A. Manteuffel  
Department of Mathematics  
University of Co. at Denver  
Denver, CO 80202

S. F. McCormick  
Computer Mathematics Group  
University of CO at Denver  
1200 Larimer St.  
Denver, CO 80204

Robert McLay  
University of Texas at Austin  
Dept. ASE-EM  
Austin, TX 78712

P. C. Messina  
158-79  
Mathematics & Comp Sci. Dept.  
Caltech  
Pasadena, CA 91125
Vineet Singh  
HP Labs, Bldg. 1U, MS 14  
1501 Page Mill Road  
Palo Alto, CA 94304

Anthony Skjellum  
Mississippi State University  
Computer Science  
PO Drawer CS  
Mississippi State, MS 38772

L. Smarr  
Director, Supercomputer Apps.  
152 Supercomputer Applications  
Bldg. 605 E. Springfield  
Champaign, IL 61801

Burton Smith  
Tera Computer Co  
400 N. 34th St., Suite 300  
Seattle, WA 98103

Barry Smith  
Department of Mathematics  
UCLA  
Los Angeles, CA 90024-1555

Harold Trease  
Los Alamos National Lab  
PO Box 1666, MS F663  
Los Alamos, NM 87545

C. VanLoan  
Department of Computer Science  
Cornell University, Rm. 5146  
Ithaca, NY 14853

John VanRosendale  
ICASE, NASA Langley Research Center  
MS 132C  
Hampton, VA 23665

Steve Vavasis  
Department of Computer Science / ACRI  
722 Engineering and Theory Center  
Cornell University  
Ithaca, NY 14853

R. G. Voigt  
MS 132-C  
NASA Langley Resch Cntr, ICASE  
Hampton, VA 36665

Phuong Vu  
Cray Research, Inc.  
19507 Franz Road  
Houston, TX 77084

Steven J. Wallach  
Convex Computer Corp.  
3000 Waterview Parkway  
PO Box 833851  
Richardson, TX 75083-3851
<table>
<thead>
<tr>
<th>#</th>
<th>MS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9018</td>
<td>Central Technical Files, 8523-2</td>
</tr>
<tr>
<td>5</td>
<td>0899</td>
<td>Technical Library, 13414</td>
</tr>
<tr>
<td>1</td>
<td>0519</td>
<td>Print Media, 12615</td>
</tr>
<tr>
<td>2</td>
<td>0100</td>
<td>Document Processing, 7613-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For DOE/OSTI</td>
</tr>
</tbody>
</table>