SAND99–8801J
Unlimited Release

Distribution
Category UC–405

Printed Nov 1999

# Official Aztec User's Guide[1] Version 2.1

Ray S. Tuminaro[2] Mike Heroux[3] Scott A. Hutchinson[4] John N. Shadid[5]
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**: workstation clusters (DEC, SGI, SUN, LINUX, etc.), Cray T3E, Intel TeraFlop, Intel Paragon, IBM SP2, nCUBE 2 as well as other MPI platforms, vector machines or serial machines.

**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).

[1]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

[2]Applied & Numerical Mathematics Department; tuminaro@cs.sandia.gov; (925) 294-2564
[3]Applied & Numerical Mathematics Department; maherou@cs.sandia.gov; (320) 845-7695
[4]Parallel Computational Sciences Department; sahutch@cs.sandia.gov; (505) 845-7996
[5]Parallel Computational Sciences Department; jnshadi@cs.sandia.gov; (505) 845-7876

# Contents

**1 Overview** **1**

**2 Aztec: High Level View** **2**
2.1 proc_config . . . 2
2.2 Aztec Options . . . 3
2.3 **Aztec** parameters . . . 7
2.4 Return status . . . 8

**3 Data Formats** **11**
3.1 Distributed Modified Sparse Row (DMSR) Format . . . 13
3.2 Distributed Variable Block Row (DVBR) Format . . . 13

**4 High Level Data Interface** **15**

**5 Examples** **17**

**6 Advanced Topics** **21**
6.1 Data Layout . . . 21
6.2 Reusing factorizations . . . 22
6.3 Important Constants . . . 23
6.4 AZ_transform Subtasks . . . 23

**7 Matrix-Free Capabilities** **25**
7.1 AZ_iterate . . . 25
7.2 Matrix-vector product examples. . . . 26
7.3 Preconditioning examples. . . . 27
7.4 **Aztec** preconditioning with user-defined matrices. . . . 28

**8 Aztec Functions** **30**
AZ_broadcast . . . 31
AZ_check_input . . . 32
AZ_check_msr . . . 32
AZ_check_vbr . . . 33
AZ_defaults . . . 33
AZ_exchange_bdry . . . 34
AZ_find_index . . . 34
AZ_find_local_indices . . . 35
AZ_find_procs_for_externs . . . 35
AZ_free_memory . . . 36
AZ_gavg_double . . . 36
AZ_gdot . . . 37
AZ_get_matvec_data . . . 37
AZ_get_matvec_data . . . 37
AZ_get_precond_data . . . 38
AZ_gmax_double . . . 38
AZ_gmax_int . . . 39
AZ_gmax_matrix_norm . . . 39
AZ_gmax_vec . . . 39
AZ_gmin_double . . . 40

AZ_gmin_int . . . 40
AZ_gsum_double . . . 41
AZ_gsum_int . . . 41
AZ_gsum_vec_int . . . 41
AZ_gvector_norm . . . 42
AZ_init_quick_find . . . 43
AZ_invorder_vec . . . 43
AZ_iterate . . . 44
AZ_matrix_create . . . 45
AZ_matrix_destroy . . . 45
AZ_matvec_mult . . . 46
AZ_MSR_matvec_mult . . . 46
AZ_msr2vbr . . . 47
AZ_order_ele . . . 48
AZ_precond_create . . . 49
AZ_precond_destroy . . . 49
AZ_print_error . . . 49
AZ_print_out . . . 50
AZ_quick_find . . . 51
AZ_read_msr_matrix . . . 52
AZ_read_update . . . 52
AZ_reorder_matrix . . . 53
AZ_reorder_vec . . . 54
AZ_set_message_info . . . 55
AZ_set_MATFREE . . . 56
AZ_set_MSR . . . 56
AZ_set_precond_print_string . . . 57
AZ_set_proc_config . . . 57
AZ_set_VB . . . 58
AZ_solve . . . 58
AZ_sort . . . 59
AZ_transform . . . 60
AZ_VBR_matvec_mult . . . 61
AZ_version . . . 61

## Notation Conventions

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

| Convention | Meaning |
|---|---|
| `typewriter` | File names, code examples and code fragments. |
| sans serif | C language elements such as function names and constants when they appear embedded in text or in function definition syntax lines. |
| *italics* | Parameter and variable names when they appear embedded in text or function definition syntax lines. |
| **AZ_** | C language elements such as function names and constants which are supplied by the **Aztec** library. |

## 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 or from the **Aztec** web site at `http://www.cs.sandia.gov/CRF/aztec1.html`.

# 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 [9]. Once the distributed matrix is created, computation can occur on any of the parallel machines running **Aztec**: workstation clusters (DEC, SGI, SUN, LINUX, etc.), Cray T3E, Intel TeraFlop, Intel Paragon, IBM SP2, nCUBE 2 as well as other MPI platforms, vector machines or serial machines.

**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 [8]. 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 work with user-supplied matrix-vector product routines or two specific sparse matrix formats (in which case **Aztec** provides the matrix-vector product) – 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 discusses Aztec's matrix-free interface and Section 8 gives a glossary of **Aztec** functions available to users.

**Example**

```
#include "az_aztec.h"

void main(void) {

  AZ_set_proc_config(proc_config, AZ_NOT_MPI );

  init_matrix_vector_structures(bindx, val, update, external,
                                update_index, extern_index, data_org);

  AZ_defaults(options,params);
  choose_solver_options(options, params);

  init_guess_and_rhs(x, b, data_org, update, update_index);

  AZ_solve(x, b, options, params, bindx, val, data_org, status,
           proc_config);
}
```

Figure 1: High level code for **Aztec** application.

# 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[6]. The functions init_matrix_vector_structures, choose_solver_options, and init_guess_and_rhs are supplied by the user. All functions beginning with AZ_ are **Aztec** functions. In this section, we give an overview of **Aztec**'s features by describing the user input arrays, *proc_config*, *options* and *params*, that are set by the user. A discussion of other parameters is deferred to Sections 4 and 5.

## 2.1 proc_config

The integer array *proc_config* of length AZ_PROC_SIZE is set by invoking AZ_set_proc_config(). This array contains the number of processors, the processor id, and an MPI communicator (if MPI is used[2]). Most users need not be concerned with the contents of this array. They must simply set it and pass it to other **Aztec** functions.

[6]The entire main program with specific sample problems is distributed with the package in the file `az_main.c`

### 2.2 Aztec Options

The integer array *options* of length AZ_OPTIONS_SIZE is set by the user. It is used (but not altered) by the function AZ_solve to choose between iterative solvers, preconditioners, etc. Default values for this array (as well as for *params*) are set by invoking AZ_defaults(). 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**

| | |
|---|---|
| *options[*AZ_solver*]* | Specifies solution algorithm. DEFAULT: AZ_gmres. |
| AZ_cg | Conjugate gradient (only applicable to symmetric positive definite matrices). |
| AZ_gmres | Restarted generalized minimal residual. |
| AZ_cgs | Conjugate gradient squared. |
| AZ_tfqmr | Transpose-free quasi-minimal residual. |
| 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. For symmetric scaling, this transforms $Ax = b$ into $SASy = Sb$ as opposed to $SAx = Sb$ when symmetric scaling is not used. NOTE: The residual within **Aztec** is now given by $S(b - Ax)$. Thus, residual printing and convergence checking are effected by scaling. 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_ls | Least-squares polynomial where the polynomial order is set via *options*[AZ_poly_ord]. |
| AZ_sym_GS | 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 *options*[AZ_poly_ord]. |
| AZ_dom_decomp | Domain decomposition preconditioner (additive Schwarz). That is, each processor augments its submatrix according to *options*[AZ_overlap] and approximately "solves" the resulting subsystem using the solver specified by *options*[AZ_subdomain_solve]. Note: *options*[AZ_reorder] determines whether matrix equations are reordered (RCM) before "solving" submatrix problem. |

*options[*AZ_subdomain_solve*]* Specifies the solver to use on each subdomain when *options*[AZ_precond] is set to AZ_dom_decomp DEFAULT: AZ_ilut.

| | |
|---|---|
| AZ_lu | Approximately solve processor's submatrix via a sparse LU factorization in conjunction with a drop tolerance *params*[AZ_drop]. The current sparse lu factorization is provided by the package y12m [10]. |
| AZ_ilut | Similar to AZ_lu using Saad's ILUT instead of LU [6]. The drop tolerance is given by *params*[AZ_drop] while the fill-in is given by *params*[AZ_ilut_fill]. |
| AZ_ilu | Similar to AZ_lu using ilu(k) instead of LU with k determined by *options*[AZ_graph_fill] |
| AZ_rilu | Similar to AZ_ilu using rilu(k,$\omega$) instead of ilu(k) with $\omega$ ($0 \geq \omega \geq 1$) given by *params*[AZ_omega] [3]. |

AZ_bilu — Similar to AZ_ilu using block ilu(k) instead of ilu(k) where each block corresponds to a VBR block.

AZ_icc — Similar to AZ_ilu using icc(k) instead of ilu(k) [5].

*options[*AZ_conv*]* — Determines the residual expression used in convergence checks and printing. DEFAULT: AZ_r0. The iterative solver terminates if the corresponding residual expression is less than *params*[AZ_tol]:

AZ_r0 — $\|r\|_2/\|r^{(0)}\|_2$

AZ_rhs — $\|r\|_2/\|b\|_2$

AZ_Anorm — $\|r\|_2/\|A\|_\infty$

AZ_noscaled — $\|r\|_2$

AZ_sol — $\|r\|_\infty/(\|A\|_\infty * \|x\|_1 + \|b\|_\infty)$

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

*options[*AZ_output*]* — Specifies information (residual expressions - see *options*[AZ_conv]) to be printed. DEFAULT: 1.

AZ_all — Print out the matrix and indexing vectors for each processor. Print out all intermediate residual expressions.

AZ_none — No intermediate results are printed.

AZ_warnings — Only Aztec warnings are printed.

AZ_last — Print out only the final residual expression.

$> 0$ — Print residual expression every *options[*AZ_output*]* iterations.

*options[*AZ_pre_calc*]* — Indicates whether to use factorization information from previous calls to AZ_solve. DEFAULT: AZ_calc.

AZ_calc — Use no information from previous AZ_solve calls.

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.

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_graph_fill*]*
: The level of graph fill-in (k) for incomplete factorizations: ilu(k), icc(k), bilu(k). DEFAULT: 0

*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 (see *options*[AZ_precond]). DEFAULT: 0.

  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.

  k
  : Augment each processor's local submatrix with rows from other processors. The new rows are obtained in k steps ($k \geq 0$). Specifically at each augmentation step, rows corresponding to external unknowns are obtained. These external unknowns are defined by nonzero columns in the current augmented matrix not containing a corresponding row on this processor. After the k steps, all columns associated with external unknowns are discarded to obtain a square matrix. The resulting procedure is an overlapped additive Schwarz procedure.

*options[*AZ_type_overlap*]*
: Determines how overlapping subdomain results are combined when different processors have computed different values for the same unknown. DEFAULT: AZ_standard.

  AZ_standard
  : The resulting value of an unknown is determined by the processor owning that unknown. Information from other processors about that unknown is discarded.

| | |
|---|---|
| AZ_symmetric | Add together the results obtained from different processors corresponding to the same unknown. This keeps the preconditioner symmetric if a symmetric technique was used on each subdomain. |
| *options[*AZ_kspace*]* | Krylov subspace size for restarted GMRES. DEFAULT: 30. |
| *options[*AZ_reorder*]* | Determines whether RCM reordering will be done in conjunction with domain decomposition incomplete factorizations. 1 indicates RCM reordering is used. 0 indicates that equations are not reordered. DEFAULT: 1. |
| *options[*AZ_keep_info*]* | Determines whether matrix factorization information will be kept after this solve (for example to solve the same system with another right hand side, see *options*[AZ_pre_calc]). 1 indicates factorization information is kept. 0 indicates that factorization information is discarded. DEFAULT: 0. |
| *options[*AZ_orthog*]* | GMRES orthogonalization scheme. DEFAULT: AZ_classic. |
| AZ_classic | 2 steps of classical Gram-Schmidt orthogonalization. |
| AZ_modified | Modified Gram-Schmidt orthogonalization. |
| *options[*AZ_aux_vec*]* | Determines $\tilde{r}$ (a required vector within some iterative methods). The convergence behavior varies slightly depending on how this is set. DEFAULT: AZ_resid. |
| AZ_resid | $\tilde{r}$ is set to the initial residual vector. |
| AZ_rand | $\tilde{r}$ is set to random numbers between -1 and 1. NOTE: When using this option, the convergence depends on the number of processors (i.e. the iterates obtained with x processors differ from the iterates obtained with y processors if x $\neq$ y). |

### 2.3 Aztec parameters

The double precision array *params* 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**

| | |
|---|---|
| *params*[AZ_tol] | Specifies tolerance value used in conjunction with convergence tests. DEFAULT: $10^{-6}$. |
| *params*[AZ_drop] | Specifies drop tolerance used in conjunction with LU or ILUT preconditioners (see description below for ILUT). DEFAULT: 0.0. |
| *params*[AZ_ilut_fill] | ILUT uses two criteria for determining the number of nonzeros in the resulting approximate factorizations. For examples, setting *params*[AZ_ilut_fill] = 1.3, requires that the ILUT factors contain no more than approximately 1.3 times the number of nonzeros of the original matrix. Additionally, ILUT drops all elements in the resulting factors that are less than *params*[AZ_drop]. Thus, when *params*[AZ_drop] is set to zero, nothing is dropped and the size of the matrix factors is governed only by *params*[AZ_ilut_fill]. However, positive values of *params*[AZ_drop] may result in matrix factors containing significantly fewer nonzeros. [6]<br>DEFAULT: 1. |
| *params*[AZ_omega] | Damping or relaxation parameter used for RILU. When *params*[AZ_omega] is set to zero, RILU corresponds to ILU(k). When it is set to one, RILU corresponds to MILU(k) where k is given by *options*[AZ_graph_fill]. [3]<br>DEFAULT: 1. |
| *params*[AZ_weights] | When *options*[AZ_conv] = AZ_weighted, the $i$'th local component of the weight vector is stored in the location *params*[AZ_weights+i]. |

Figure 2 illustrates a sample user function choose_solver_options that chooses specific solver options (by overwriting default values set with AZ_defaults).

### 2.4 Return status

The double precision array *status* of length AZ_STATUS_SIZE returned from AZ_solve[7]. The contents of *status* are described below.

**Specifications**

| | |
|---|---|
| *status*[AZ_its] | Number of iterations taken by the iterative method. |
| *status*[AZ_why] | Reason why AZ_solve terminated. |
| AZ_normal | User requested convergence criteria is satisfied. |
| AZ_param | User requested option is not available. |

[7] All integer information returned from AZ_solve is cast into double precision and stored in *status*.

**Example**

```
void choose_solver_options(int options[AZ_OPTIONS_SIZE],
                  double params[AZ_PARAMS_SIZE])
{
  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;
  params[AZ_drop]        = 0.;
  params[AZ_omega]        = 1.;

}
```

Figure 2: Example option initialization routine (choose_solver_options).

| | |
|---|---|
| AZ_breakdown | Numerical breakdown occurred. |
| AZ_loss | Numerical loss of precision occurred. |
| AZ_ill_cond | The Hessenberg matrix within GMRES is ill-conditioned. This could be caused by a number of reasons. For example, the preconditioning matrix could be nearly singular due to an unstable factorization (note: pivoting is not implemented in any of the incomplete factorizations). Ill-conditioned Hessenberg matrices could also arise from a singular application matrix. In this case, GMRES tries to compute a least-squares solution. |
| AZ_maxits | Maximum iterations taken without convergence. |
| *status*[AZ_r] | The true residual norm corresponding to the choice *options*[AZ_conv] (this norm is calculated using the computed solution). |
| *status*[AZ_scaled_r] | The true residual ratio expression as defined by *options*[AZ_conv]. |
| *status*[AZ_rec_r] | 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. |
| *status*[AZ_solve_time] | Utilization time in Aztec to solve system. |
| *status*[AZ_Aztec_version] | Version number of Aztec. |

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. User's who wish to supply their own matrix-vector product can skip the matrix description and instead read Section 7 where **Aztec**'s matrix-free interface is discussed.

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** [4]). 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 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 $N_internal$, $N_border$, and $N_external$ elements are distributed over the sets *internal*, *border* and *external* respectively.

| **set** | **description** | **local numbering** |
|---|---|---|
| *internal* | updated w/o communication | 0 to $N_internal - 1$. |
| *border* | updated with communication | $N_internal$ to $N_internal + N_border - 1$. |
| *external* | not updated but used to update *border*. | $N_internal + N_border$ to $N - 1$. Elements received from the same processor are numbered consecutively. |

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

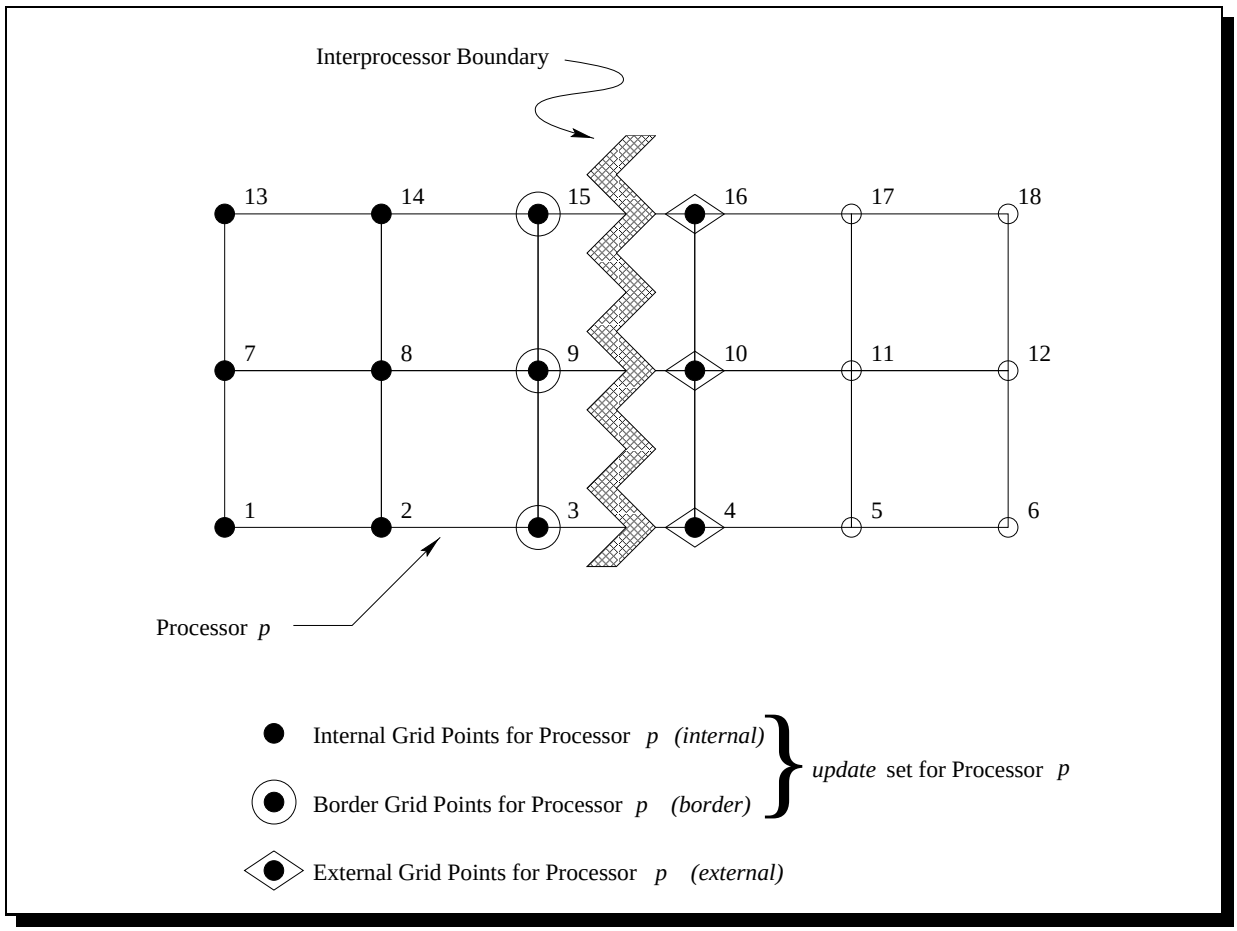


Figure 3: Example partitioning of a finite element grid.

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 [7]. The data structure consists of an integer vector *bindx* and a double precision vector *val* each of length *N_nonzeros + 1* where *N_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]* | = number of nonzero off-diagonal elements in $k$'th row, $k < m$ |
| *bindx[$k_s$...$k_e$]* | = column indices of the off-diagonal nonzeros in row $k$ where $k_s$ = *bindx[k]* and $k_e$ = *bindx[k+1]-1.* |

*val* **:**

| | |
|---|---|
| *val[k]* | = $A_{kk}, k < m$ |
| *val[$k_i$]* | = the ($k$, *bindx[$k_i$]* )'th matrix element where $k_s \leq k_i \leq k_e$ with $k_s$ and $k_e$ 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 ... M]* **:**

| | |
|---|---|
| *rpntr[0]* | = 0 |
| *rpntr[k+1] - rpntr[k]* | = number of rows in $k$'th block row |

*cpntr[0 ... K]* **:**

*cpntr[0]* = 0
*cpntr[k+1] - cpntr[k]* = number of columns in $k$'th block column

*bpntr[0 ... M]* :
*bpntr[0]* = 0
*bpntr[k+1] - bpntr[k]* = number of nonzero blocks in the $k$'th block row

*bindx[0 ... bpntr[M] - 1]* :
*bindx[$k_s$...$k_e$]* = block column indices of nonzero blocks in block row $k$ where $k_s$ = *bpntr[k]* and $k_e$ = *bpntr[k+1]*-1

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

*val[0 ... indx[bpntr[M]] - 1 ]* :
*val[$i_s$...$i_e$]* = nonzeros in the ($k$, *bindx[$k_i$]*)'th block stored in column major order where $k_i$ is as defined above, $i_s$ = *indx[$k_i$]* 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. $i < j \Rightarrow update[i] < update[j]$). 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 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`. 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**.

**Example**

proc 0:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N_update: | 3 | | | | | | | | | | | | |
| update: | 0 | 1 | 3 | | | | | | | | | | |
| bindx: | 4 | 7 | 9 | 14 | 1 | 3 | 4 | 0 | 3 | 0 | 1 | 2 | 4 | 5 |
| val: | $a_{00}$ | $a_{11}$ | $a_{33}$ | - | $a_{01}$ | $a_{03}$ | $a_{04}$ | $a_{10}$ | $a_{13}$ | $a_{30}$ | $a_{31}$ | $a_{32}$ | $a_{34}$ | $a_{35}$ |

proc 1:

| | | | | | |
|---|---|---|---|---|---|
| N_update: | 1 | | | | |
| update: | 4 | | | | |
| bindx: | 2 | 5 | 0 | 3 | 2 |
| val: | $a_{44}$ | - | $a_{40}$ | $a_{43}$ | $a_{42}$ |

proc 2:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N_update: | 2 | | | | | | | |
| update: | 2 | 5 | | | | | | |
| bindx: | 3 | 6 | 8 | 3 | 4 | 5 | 2 | 3 |
| val: | $a_{22}$ | $a_{55}$ | - | $a_{23}$ | $a_{24}$ | $a_{25}$ | $a_{52}$ | $a_{53}$ |

Figure 4: User input (MSR format) to initialize the sample matrix problem.

**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);
}
```

Figure 5: init_matrix_vector_structures.

**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[i] = (double) update[i];
    x[i] = 0.0;
  }
  AZ_reorder_vec(rhs,data_org,update_index,NULL);
  AZ_reorder_vec(  x,data_org,update_index,NULL);
                           /* reorder 'rhs' and 'x' to correspond to    */
                           /* AZ_transformed matrix. Note: convert      */
                           /* solution from AZ_solve (or AZ_iterate)    */
                           /* back to user ordering via AZ_invorder_vec */
}
```

Figure 6: init_guess_and_rhs.

**Example**

```
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);
}
```

Figure 7: create_matrix.

# 5 Examples

A sample program is described by completing the program fragments given earlier (Figures 1, 2 and 5). In Figure 1, AZ_set_proc_config 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. 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)[8].

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

[8] Alternatively, $i$ could replace *update_index[i]* by invoking *AZ_reorder_vec()* once *rhs* was initialized.

**Example**

```
void add_row(row, location, val, bindx)
{
   k = bindx[location];   val[location] = 4.; /* matrix diagonal */

   /* 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;
}
```

Figure 8: add_row for a 2D Poisson problem

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 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 matrix_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

**Example**

```
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);
}
```

Figure 9: create_matrix for the Poisson finite element problem.

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.

**Example**

```
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 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 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);
            }
         }
      }
   }
```

Figure 10: matrix_fill for the Poisson finite element problem.

# 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:

**Specifications**

| | |
|---|---|
| *data_org*[AZ_matrix_type] | Specifies matrix format. |
| AZ_VBR_MATRIX | Matrix corresponds to VBR format. |
| AZ_MSR_MATRIX | Matrix corresponds to MSR format. |
| *data_org*[AZ_N_internal] | Number of elements updated by this processor that can be computed without information from neighboring processors (*N_internal*). This also corresponds to the number of internal rows assigned to this processor. |
| *data_org*[AZ_N_border] | Number of elements updated by this processor that use information from neighboring processors (*N_border*). |
| *data_org*[AZ_N_external] | Number of *external* components needed by this processor (*N_external*). |
| *data_org*[AZ_N_int_blk] | Number of internal VBR block rows owned by this processor. Set to data_org[AZ_N_internal] for MSR matrices. |
| *data_org*[AZ_N_bord_blk] | Number of border VBR block rows owned by this processor. Set to data_org[AZ_N_border] for MSR matrices. |
| *data_org*[AZ_N_ext_blk] | Number of external VBR block rows on this processor. Set to data_org[AZ_N_external] for MSR matrices. |
| *data_org*[AZ_N_neigh] | Number of processors with which we exchange information (send or receive) in performing matrix-vector products. |
| *data_org*[AZ_total_send] | Total number of vector elements sent to other processors during matrix-vector products. |

| | |
|---|---|
| *data_org[*AZ_name*]* | Name of the matrix. This name is utilized when deciding which previous factorization to use as a preconditioner (see Section 6.2). (positive integer value). |
| *data_org[*AZ_neighbors*]* | Start of vector containing node i.d.'s of neighboring processors. That is, *data_org[*AZ_neighbors*+i]* gives the node i.d. of the (*i+1*)'th neighbor. |
| *data_org[*AZ_rec_length*]* | Start of vector containing the number of elements to receive from each neighbor. We receive from the (*i+1*)'th neighbor *data_org[*AZ_rec_length*+i]* elements. |
| *data_org[*AZ_send_length*]* | Start of vector containing the number of elements to send to each neighbor. We send to the (*i+1*)'th neighbor *data_org[*AZ_rec_length*+i]* elements. |
| *data_org[*AZ_send_list*]* | Start of vector indicating the elements that we will send to other processors during communication. The first *data_org[*AZ_send_length*]* components correspond to the elements for the first neighbor and the next *data_org[*AZ_send_length*+1]* components correspond to element indices for the second neighbor, and so on. |

## 6.2 Reusing factorizations

By solving a problem with *options[*AZ_keep_info*]* set to '1', **Aztec** will not remove information so that it 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:

$$A_1 x = b, A_2 y = x, \text{ and } A_1 z = y.$$

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[9]) 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_free_memory(*data_org[*AZ_name*]*).

[9]The matrix does not need to be rescaled as the scaling during the first solve overwrites the original matrix.

### 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:

| | |
|---|---|
| AZ_MAX_NEIGHBORS | Maximum number of processors with which information can be exchanged during matrix-vector products. |
| AZ_MSG_TYPE<br>AZ_NUM_MSGS | All message types used inside **Aztec** lie between AZ_MSG_TYPE and AZ_MSG_TYPE + AZ_NUM_MSGS - 1. |
| AZ_MAX_BUFFER_SIZE | 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. |
| AZ_MAX_MEMORY_SIZE | 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. |
| AZ_TEST_ELE | 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. |

### 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 [9]. Prototypes for these subroutines as well as for AZ_transform are given in Section 8.

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 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_for_externs 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 Matrix-Free Capabilities

Version 2.1 contains a matrix-free capability. However, this capability is still undergoing some changes. In this document we describe the **Aztec** interface and give concrete examples with user-defined matrix-vector products and user-defined preconditioners. At present, only the 'C' interface has been completed. A Fortran interface will follow.

## 7.1 AZ_iterate

To use **Aztec**'s solvers, a subroutine driver entitled *AZ_iterate* must be invoked. For those familiar with **Aztec1.1**, this subroutine replaces *AZ_solve*. It should be noted that:

- *AZ_solve* may still be used (when matrix-free is not needed).
- The interface to *AZ_iterate* will probably change a bit to incorporate new developments (including the integration of **Aztec** with eigenvalue solvers and multilevel solvers).

There are three conceptual differences between *AZ_iterate* and *AZ_solve*:

- the user's matrix is more generalized and is now passed through a structure,
- user-supplied preconditioning can be used,
- **two different matrices** can be supplied: one to solve and one to use within preconditioning. For example, **Aztec** can solve equations $Ax = f$ by applying an incomplete factorization to a matrix $B$ which is a sparse approximation to $A$.

The syntax for *AZ_iterate* is given below.

**Prototype**

```
void AZ_iterate(double *x, double *b, int *options, double *params, double *status,
                int *proc_config, AZ_MATRIX *Amat, AZ_PRECOND *prec,
                struct AZ_SCALING *scaling)
```

**Parameters**

| | |
|---|---|
| *x, b, options, params, status, proc_config* | These parameters are identical to those in *AZ_solve* and are described in Section 8. |
| *Amat* | On input, a structure representing the matrix to be solved (described below). |
| *prec* | On input, either NULL (implying **Aztec**'s preconditioners are applied to *Amat*) or a structure indicating the preconditioning routine and matrix to which preconditioning is applied (described below). |

| | |
|---|---|
| *scaling* | Currently not used. |

To complete the discussion, we must describe how *Amat* and *prec* are created and set.

### 7.2 Matrix-vector product examples.

The following examples use a variety of functions: *AZ_set_MSR()*, *AZ_set_VBR()*, etc. These functions are more fully described in Section 8.

Example 1: DMSR or DVBR matrices

*AZ_solve*() users with DMSR matrices can convert to *AZ_iterate*() by replacing

```
AZ_solve(x, b, options, params, indx, bindx, rpntr, cpntr, bpntr,
         val, data_org, status, proc_config);
```

with

```
AZ_MATRIX *Amat;
   .
   .
Amat = AZ_matrix_create(data_org[AZ_N_internal] +
                        data_org[AZ_N_border]);
AZ_set_MSR(Amat, bindx, val, data_org, 0, NULL, AZ_LOCAL);
AZ_iterate(x, b, options, params, status, proc_config, Amat, NULL, NULL);
AZ_matrix_destroy(&Amat);
```

Users with DVBR matrices would replace *AZ_set_MSR()* by

```
AZ_set_VBR(Amat, rpntr, cpntr, bpntr, indx, bindx,
           val, data_org, 0, NULL, AZ_LOCAL);
```

Example 2: user-supplied matrix-vector product

This case is almost identically to the DMSR or DVBR examples above except that *AZ_set_MATFREE* (as opposed to *AZ_set_MSR* or *AZ_set_VBR*) is used to associated the user-defined matrix-vector product with the matrix.

```
AZ_MATRIX *Amat;
   .
   .
Amat = AZ_matrix_create(N_equations);
AZ_set_MATFREE(Amat, my_data, my_matvec);
AZ_iterate(x, b, options, params, status, proc_config, Amat, NULL, NULL);
AZ_matrix_destroy(&Amat);
```

`N_equations` is the number of matrix rows implicitly kept on this processor (i.e. the length of the vector $y_{\text{local}}$ after performing $y_{\text{local}} = A_{\text{local}} x_{\text{local}}$). `my_data` is a user-defined data pointer that will be passed to the user-defined matrix-vector product function `my_matvec`. `my_matvec` should have the following proto-type:

```
void my_matvec(double *p, double *ap, AZ_MATRIX *Amat, int proc_config[])
```

where

| | |
|---|---|
| *p* | On input, *p* is a vector containing local vector components for this processor. NOTE: **Aztec**'s internal matrix-vector multiplies require additional space (see *AZ_solve*() description in Section 8). |
| *ap* | On output, *ap* is set to $(Amat) * p$. |
| *Amat* | On input, structure representing matrix used in matrix-vector multiplication. |
| *proc_config* | On input, *proc_config*[AZ_node] is set via the command *AZ_set_proc_config* . |

Whenever the user-defined data pointer, *my_data*, is needed inside *my_matvec*(), it can be obtained via the function *AZ_get_matvec_data (Amat)*.

### 7.3 Preconditioning examples.

The following examples use: *AZ_precond_create()*, and *AZ_precond_destroy()*. These functions are more fully described in Section 8.

Example 1: Standard **Aztec** preconditioning

Standard **Aztec** preconditioning can be invoked by creating a preconditioner,

```
AZ_PRECOND *prec;
    .
    .
prec = AZ_precond_create(Amat, AZ_precondition, NULL);,
```

passing it to *AZ_iterate*(),

```
AZ_iterate(x, b, options, params, status, proc_config, Amat, prec, NULL);,
```

and finally clearing it with

```
AZ_precond_destroy(&prec);
```

Example 2: Standard **Aztec** preconditioning applied to $B$

**Aztec**'s preconditioners can be applied to a matrix, $B$, which is different from the matrix being solved. In this case, the following might be used:

```
AZ_PRECOND *prec;
AZ_MATRIX  *Amat, *Bmat;
    .
    .
N_equations = data_org[AZ_N_internal]+data_org[AZ_N_border];
Amat = AZ_matrix_create(N_equations);
AZ_set_MSR(Amat, bindx, val, data_org, 0, NULL, AZ_LOCAL);
Bmat = AZ_matrix_create(N_equations);
AZ_set_MSR(Bmat, bindx2, val2, data_org2, 0, NULL, AZ_LOCAL);
```

```
prec = AZ_precond_create(Bmat, AZ_precondition, NULL);
AZ_iterate(x, b, options, params, status, proc_config, Amat, prec, NULL);
AZ_matrix_destroy(&Amat);
AZ_matrix_destroy(&Bmat);
AZ_precond_destroy(&prec);
```

Example 3: user-supplied preconditioner User-supplied preconditioning requires that the user's function pointer and data pointer be given to AZ_precond_create(). For example,

```
AZ_PRECOND *prec;
AZ_MATRIX  *Amat;
    .
    .
N_equations = data_org[AZ_N_internal]+data_org[AZ_N_border];
Amat = AZ_matrix_create(N_equations);
AZ_set_MSR(Amat, bindx, val, data_org, 0, NULL, AZ_LOCAL);
prec = AZ_precond_create(Amat, my_preconditioner, my_data);
AZ_iterate(x, b, options, params, status, proc_config, Amat, prec, NULL);
AZ_matrix_destroy(&Amat);
AZ_precond_destroy(&prec);
```

my_data is a user-defined data pointer that will be passed to the user-defined preconditioner function my_preconditioner. my_preconditioner should have the following proto-type:

```
void my_preconditioner(double *z, int *options, int *proc_config,
                       double *params, AZ_MATRIX *Amat,
                       AZ_PRECOND *prec)
```

where

| | |
|---|---|
| *z* | On input, $z$ is a vector containing all the local vector components for this processor. On output, $z$ is the preconditioned vector (local components). |
| *options, proc_config*<br>*params, Amat, prec* | Same as those given to *AZ_iterate*. |

### 7.4 Aztec preconditioning with user-defined matrices.

In general, **Aztec** needs to know something about the matrix in order to apply a preconditioner (see Table 1).

| Preconditioner | needed matrix information |
|---|---|
| Jacobi | matrix diagonal |
| least squares polynomial | infinity norm of matrix |
| Neumann polynomial | infinity norm of matrix |
| incomplete factorizations | algebraic preconditioners are used for the subdomain solvers. These algebraic preconditioners need matrix entries. |

Table 1: *Data requirements for various preconditioners.*

For the polynomial preconditioners, it is possible to give **Aztec** the infinity norm via the command AZ_set_MATFREE_matrix_norm(). For Jacobi preconditioning it is often possible to give the matrix diagonal in a separate DMSR matrix. However, for most of the other preconditioners, it is necessary to give all the matrix information. The current **Aztec** contains an experimental 'getrow' facility which allows this to be done. This 'getrow' is demonstrated in the example program `az_mat_free_main.c` but is not officially released as it is still undergoing changes.

# 8 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 integer variables at the beginning of this section and then proceed with the individual function descriptions.

### Frequently Used Aztec Parameters

| | |
|---|---|
| *data_org* | Array describing the matrix format (Section 6.1). Allocated and set AZ_set_message_info and AZ_transform. If allocated manually (e.g. fortran programs), must be of size AZ_COMM_SIZE $+k$ where $k$ is the total number of values sent to neighboring processors (to update their ghost nodes) when doing a matrix-vector product. |
| *extern_index* | *extern_index[i]* gives the local numbering of global element *external[i]*. Allocated and set by AZ_order_ele and AZ_transform. |
| *extern_proc* | *extern_proc[i]* is updating processor of *external[i]*. Allocated and set by AZ_find_procs_for_externs. |
| *external* | Sorted list (global indices) of external elements on this node. Allocated and set by AZ_find_local_indices and AZ_transform . |
| *N_external* | Number of *external* components. Set by AZ_find_procs_for_externs and AZ_transform. |
| *N_update* | Number of *update* components assigned to this processor. Set by AZ_read_update. |
| *options, params* | Arrays describing AZ_solve options (Section 2). |
| *proc_config[]* | Integer array of length AZ_PROC_SIZE consisting of primarily: the node id (*proc_config[AZ_node]*), the total number of processors (*proc_config[AZ_N_procs]*), and the MPI[2] communicator (AZ_get_comm(*proc_config*)). This array must be set by calling AZ_set_proc_config. |
| *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

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] $\neq$ 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] $\neq$ 0: clear internal buffer.

Sample Usage:

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

```
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

```
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

```
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*).

### Parameters

*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

```
void AZ_check_vbr(int N_update, int N_external, int option, int *bindx,
                  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

```
int AZ_defaults(int *options, double *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

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

### 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

```
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

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

### 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

```
void AZ_find_procs_for_externs(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

```
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

```
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

```
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

```
MPI_Comm* AZ_get_comm( int *proc_config )
```

### Description

Returns a pointer to the MPI communicator associated with *proc_config* (see AZ_set_proc_config).

### Parameters

| | |
|---|---|
| *proc_config* | On input, an integer array of size AZ_PROC_SIZE that has been initialized with AZ_set_proc_config. |

### Prototype

```
void* AZ_get_matvec_data( AZ_MATRIX *Amat )
```

### Description

Returns data pointer associated with matrix (that was set via AZ_set_MATFREE().

### Parameters

| | |
|---|---|
| *Amat* | On input, matrix created via AZ_matrix_create(). |

### Prototype

```
void* AZ_get_precond_data( AZ_PRECOND *prec )
```

### Description

Returns data pointer associated with preconditioner (that was set via AZ_precond_create().

### Parameters

| | |
|---|---|
| *prec* | On input, preconditioner created using the function AZ_precond_create(). |

### Prototype

```
double AZ_gmax_double(double value, int *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

```
int AZ_gmax_int(int value, int *proc_config )
```

### Description

Return the maximum of the numbers in *value* on all processors.

### Parameters

*value* On input, *value* contains an integer.

### Prototype

```
double AZ_gmax_matrix_norm(double *val, int *indx, int *bindx, int *rpntr, int *cpntr,
                           int *bpntr, int *proc_config, int *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

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

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

int AZ_gmin_int(int $value$, int *$proc_config$ )

### Description

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

### Parameters

| | |
|---|---|
| *value* | On input, *value* contains an integer. |

### Prototype

```
double AZ_gsum_double(double value, int *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

```
int AZ_gsum_int(int value, int *proc_config )
```

### Description

Return the sum of the integers in *value* on all processors.

### Parameters

| | |
|---|---|
| *value* | On input, *value* contains an integer. |

### Prototype

void AZ_gsum_vec_int(int *$values$, int *$wkspace$, int $length$, int *$proc_config$ )

### Description

*values[i]* is set to the sum of the input numbers in *values[i]* on all processors ($i <$ *length*).

### Parameters

| | |
|---|---|
| *values* | On input, *values* contains a list of integers. On output, *values[i]* contains the sum of the input *values[i]* on all the processors. |
| *wkspace* | On input, workspace array of size *length*. |
| *length* | On input, length of *values* and *wkspace*. |

### Prototype

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

| | |
|---|---|
| $n$ | On input, number of *update* components of $x$ on this processor. |
| $p$ | On input, order of the norm to perform, i.e., $\|x\|_p$. |

| | |
|---|---|
| $x$ | On input, vector whose norm will be computed. |

### Prototype

```
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

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

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

$$bins[k] \leq list[j] < bins[k+1]$$

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

| | |
|---|---|
| *list* | On input, sorted *list*. |
| *length* | On input, length of *list*. |
| *shift* | On output, *shift* is set as described in above. |
| *bins* | On input, array of size $2 + length/4$. On output, *bins* is set as described above. |

### Prototype

```
void AZ_invorder_vec(double *x , int *data_org, int *update_index, int *rpntr, double
*untrans_x )
```

## Description

Untransform the vector $x$ whose data layout corresponds to a transformed matrix (see AZ_transform or AZ_reorder_matrix) so that the new vector ($untrans_x$) corresponds to data in the original user-given ordering.

## Parameters

| | |
|---|---|
| $x$ | On input, distributed vector whose data layout corresponds to a transformed matrix. |
| $untrans_x$ | On output, the result of untransforming $x$ so that the data-layout now corresponds to the original ordering given by the user. |

## Prototype

```
void AZ_iterate(double *x, double *b, int *options, double *params, double *status,
                int *proc_config, AZ_MATRIX *Amat, AZ_PRECOND *prec,
                struct AZ_SCALING *scaling)
```

## Parameters

| | |
|---|---|
| $x$ | $x$ is the initial guess on input and the linear system solution on output. NOTE: when using **Aztec**'s DMSR or DVBR formats, $x$ should contain space for ghost elements or external variables (see *AZ_solve* description). Matrix-free users need not supply this additional space. |
| $b$ | $b$ is the right hand side vector. Note: $b$ does not need to contain space for ghost variables. |
| *options, params* | Options and parameters used during the solution process (Section 2). |
| *status* | On output, status of iterative solver (Section 2). |
| *proc_config* | On input, *proc_config*[AZ_node] is the node id of this processor and *proc_config*[AZ_N_procs] is the total number of processors. |
| *Amat* | On input, a structure representing the matrix to be solved (described below). |

| | |
|---|---|
| *prec* | On input, either NULL (implying that **Aztec**'s preconditioners are to be applied to *Amat*) or a structure indicating the preconditioning routine and matrix to which preconditioning is applied (described below). |
| *scaling* | Currently not used. |

### Prototype

```
AZ_MATRIX* AZ_matrix_create( int N_rows )
```

### Description

Creates a matrix with *N_rows* matrix rows (on this processor).

### Parameters

| | |
|---|---|
| *N_rows* | On input, number of matrix rows on this processor for the matrix to be created. |

### Prototype

```
void AZ_matrix_destroy( AZ_MATRIX **Amat )
```

### Description

Destroy matrix *Amat*. Note: this routine does not destroy arrays or data structures associated with this matrix by the set functions: AZ_set_MSR, AZ_set_VBR, or AZ_set_MATFREE.

### Parameters

| | |
|---|---|
| *Amat* | On input, matrix to be freed. On output, memory used for creating the matrix object is freed (see description above). |

### Prototype

```
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). Note this routine will eventually be replaced by *AZ_MSR_matvec_mult()* and *AZ_VBR_matvec_mult()*.

### Parameters

| | |
|---|---|
| *b* | On input, distributed vector to use in multiplication. NOTE: $b$ should contain $data_org[AZ_N_internal] + data_org[AZ_N_border] + data_org[AZ_N_external]$ elements (though external variables stored at the end of the vector do not need to be initialized). |
| *c* | On output, the result of matrix-vector multiplication. NOTE: $c$ should contain $data_org[AZ_N_internal] + data_org[AZ_N_border]$ elements. |
| *exchange_flag* | On input, dictates whether communication needs to occur. If $exchange_flag == 1$, communication occurs. If $exchange_flag == 0$, no communication occurs. |

### Prototype

```
void AZ_MSR_matvec_mult( double *b, double *c, AZ_MATRIX *A, int *proc_config)
```

### Description

Perform the matrix-vector multiply

$$c \leftarrow Ab$$

where the matrix $A$ is a DMSR matrix (see Section 7).

### Parameters

| | |
|---|---|
| *b* | On input, distributed vector to use in multiplication. NOTE: *b* should contain *A*-> *data_org*[AZ_N_internal] + *A*->*data_org*[AZ_N_border] + *A*-> *data_org*[AZ_N_external] elements (though external variables stored at the end of the vector do not need to be initialized). |
| *c* | On output, the result of matrix-vector multiplication. NOTE: *c* should contain *A*->*data_org*[AZ_N_internal] + *A*->*data_org*[AZ_N_border] elements. |

### Prototype

```
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

```
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

AZ_PRECOND * AZ_precond_create( AZ_MATRIX *$Pmat$, void (*)() $prec_function$, void *$data$ )

### Description

Creates a preconditioner and associates the preconditioning function $prec_function$, the matrix $Pmat$, and the data pointer $data$ with it.

### Parameters

| | |
|---|---|
| $Pmat$ | On input, matrix to which the preconditioner is to be applied. |
| $prec_function$ | On input, function to be invoked when preconditioning is needed. |
| $data$ | On input, data pointer to be associated with preconditioner. |

### Prototype

void AZ_precond_destroy( AZ_PRECOND **$prec$ )

### Description

Destroy preconditioner $prec$. Note: this routine does not destroy arrays or data structures associated with this preconditioner via AZ_precond_create.

### Parameters

| | |
|---|---|
| $prec$ | On input, preconditioner to be freed. On output, memory used for creating the preconditioner object is freed (see description above). |

### Prototype

void AZ_print_error(int *error_code*)

### Description

Prints out an error message corresponding to *error_code*. Typically, *error_code* is generated by AZ_check_input.

### Parameters

| | |
|---|---|
| *error_code* | On input, error code generated by AZ_check_input. |

### Prototype

void AZ_print_out(int **update_index*, int **extern_index*, int **update*, int **external*, double **val*, int **indx*, int **bindx*, int **rpntr*, int **cpntr*, int **bpntr*, int **proc_config*, int *choice*, int *matrix_type*, int *N_update*, int *N_external*, int *offset* )

### Description

Print **Aztec** matrices in one of several formats (see also the file `doc/AZ_capture_matrix_howto.txt` in the **Aztec** distribution):

- *choice* = *AZ_input_form*
  Output corresponding to Figure 4 (similar information for VBR matrices). The matrix can be printed either before or after being *AZ_transformed*.
- *choice* = *AZ_global_mat*
  Can only be used on *AZ_transformed* matrices. Nonzeros are printed as:
  - MSR: $a(I + offset, J + offset) = value$;
  - VBR: $a(I + offset(i), J + offset(j)) = value$;

  where $I, J$ are global (block) row and column indices and $i, j$ are row and column positions within the block.
- *choice* = *AZ_explicit*
  Similar to *AZ_global_mat* except $(I, J)$ are interpreted differently. If used on an *AZ_transformed* matrix and *NULL* is passed instead of *update*, $(I, J)$ are the local rows and columns on this processor. If used before *AZ_transform* and *update* is passed, $(I, J)$ correspond to the global matrix numbering.

NOTE: *update_index*, *extern_index*, *update*, *external*, and *N_external* are only used for the *AZ_global_mat* option. Also, if set to *NULL*, *cpntr* is not printed or used.

### Parameters

| | |
|---|---|
| *choice* | On input, selects output format as described above. |
| *matrix_type* | On input, matrix format: either AZ_VBR_MATRIX or AZ_MSR_MATRIX. |
| *N_update* | On input, number of (block) rows assigned to processor. |
| *N_external* | On input, $data_org[AZ_N_ext_blk]$. |
| *offset* | On input, row and column offset. See above description. |

### Prototype

```
int AZ_quick_find(int key, int *list, int length, int shift, int *bins )
```

### Description

Return the index, $i$, in *list* (assumed to be sorted) which matches the key (i.e. $list[i] = key$). If *key* is not found AZ_quick_find returns -1.
NOTE: This version is faster than AZ_find_index but requires *bins* to be set and stored using AZ_init_quick_find.

### Parameters

| | |
|---|---|
| *key* | On input, element to search for in *list*. |
| *list* | On input, sorted list to be searched. |
| *length* | On input, length of list. |
| *shift* | On input, used for initial guess (computed by previous AZ_init_quick_find call). |
| *bins* | On input, computed by AZ_init_quick_find for initial guess. *bins* is set so that $list[bins[k]] \leq key < list[bins[k+1]]$ where $k = (key - list[0])/2^{shift}$ . |

### Prototype

```
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 *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 `col_num1`, `entry2` in column `col_num2` and `entry3` in column `col_num3`. Row 1 contains `entry4` in column `col_num4`, `entry5` in column `col_num5` and `entry6` in column `col_num6`. When using exponential notation, 'E' or 'e' should be used instead of 'D' or 'd'. Additionally, it is important that row and column numbers be labeled from 0 to n-1 and not 1 to n where n is the number of rows.

NOTE: Spacing and carriage returns are not important.

NOTE: AZ_read_msr_matrix() is inefficient for large matrices. It is, however, possible to read binary files. In particular, if **Aztec** is compiled with a -Dbinary option, '.data' must contain binary integers and binary double precision numbers in the same form as above except without spaces or carriage returns between numbers. Since binary formats are not standard, '.data' should be created using a 'C' program via fread() and fwrite() executed on the same machine as **Aztec**.

### Parameters

| | |
|---|---|
| *val, bindx* | On output, these two arrays are allocated and filled with the MSR representation corresponding to the file `.data`. |

### Prototype

```
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

*N_update* — On output, number of elements updated by processor.

*update* — On output, *update* is allocated and contains a list of elements updated by this processor in ascending order.

*chunk* — Number of indices within a group. For example, $chunk == 2 \Rightarrow chunk_0 = \{0, 1\}$, and $chunk_1 = \{2, 3\}$.

*N* — Total number of chunks in the vector.

*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 $P =$ proc_config[AZ_N_procs]).

AZ_box — The processor system is viewed as a $p_2 \times p_1 \times p_0$ array while the matrix problem corresponds to a $n_2 \times n_1 \times n_0$ grid with 'chunks' unknowns at each grid point. The user is prompted for the $p_i$'s and the $x_i$'s. Assuming that the grid is ordered 'naturally' or lexicographically, the grid is subdivided into uniform boxes whose corresponding rows are to be assigned to each processor.

AZ_file — Read the proc_config[AZ_N_procs] lists contained in the file `.update`. Each list contains a set of global indices preceded by the of number of indices in this set. List 0 is sent to processor proc_config[AZ_N_procs] - 1, list 1 is sent to processor proc_config[AZ_N_procs] - 2, etc. Note: A graph partitioning package named **Chaco** [4] produces files in this format.

## Prototype

```
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)
```

## Description

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 |

See also AZ_reorder_vec and AZ_invorder_vec to transform the right hand side and solution vectors.

## Parameters

| | |
|---|---|
| *option* | On input, indicates whether to reorder update elements. |
| AZ_ALL | All the rows and columns are renumbered. |
| AZ_EXTERNS | 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, rpntr, bpntr, cpntr* | On input, matrix ordered as described above. On output, matrix reordered using *update_index* and *extern_index* as described above. |

## Prototype

```
void AZ_reorder_vec(double *x , int *data_org, int *update_index, int *rpntr )
```

### Description

Transform the vector $x$ whose data layout corresponds to the user-given ordering (i.e. $x[i]$ corresponds to global vector element $x[update[i]]$) so that now the data corresponds to a transformed matrix (see AZ_transform or AZ._reorder_matrix).

### Parameters

| | |
|---|---|
| $x$ | On input, distributed vector whose data layout corresponds to the original ordering given by the user. On output, the data layout now corresponds to a transformed matrix whose ordering is given by *update_index*. |

### Prototype

```
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

```
void AZ_set_MATFREE(AZ_MATRIX *Amat, void *data, void (*matvec)() )
```

### Parameters

| | |
|---|---|
| *Amat* | On input, matrix created via AZ_matrix_create(). On output, *Amat* is labeled as a user-supplied matrix and the matrix-vector product function *matvec* is associated with it as well as the data pointer *data*. |
| *data* | On input, user-defined data pointer. On output, this pointer is associated with *Amat* and is accessible inside the user's matrix-vector product routine (see AZ_get_matvec_data()). |
| *matvec* | On input, user-defined function pointer corresponding to a matrix-vector product. On output, this function pointer is associated with *Amat*. The exact proto-type description of this routine is given in Section 7. |

### Prototype

```
void AZ_set_MSR(AZ_MATRIX *Amat, int *bindx, double *val, int *data_org,
                int N_update, int *update, int option)
```

### Parameters

| | |
|---|---|
| *Amat* | On input, matrix created via AZ_matrix_create(). On output, *Amat* is labeled as a DMSR matrix and the the DMSR arrays *bindx*, *val*, *data_org* are associated with it. |
| *bindx,val, data_org* | On input, DMSR matrix arrays. On output, these arrays are associated with *Amat*. |
| *N_update, update, options* | On input, should be set to *0, NULL, AZ_LOCAL* respectively. In the future, these parameters will allow for matrices with global indices. |

### Prototype

```
void AZ_set_precond_print_string(AZ_PRECOND *prec, char *str)
```

### Description

Associate string with preconditioner. This string will be printed out by **Aztec** when AZ_iterate is invoked and output is turn on.

### Parameters

| | |
|---|---|
| *prec* | On input, preconditioner created using the function AZ_precond_create(). On output, a copy of *str* is made and associated with *prec*. |

### Prototype

```
void AZ_set_proc_config(int *proc_config, MPI_AZComm comm)
```

### Description

Sets *proc_config[]* for use by **Aztec**. *proc_config[]* contains the number of processors (in *proc_config[*AZ_N_procs*]*), the processor id (in *proc_config[*AZ_node*]*), and the communicator (via *AZ_get_comm(proc_config)* ) if MPI [2] is used.

### Parameters

| | |
|---|---|
| *proc_config* | On input, an integer array of size AZ_PROC_SIZE. On output, *proc_config[]* is filled as described above. |
| *comm* | On input, should be set to AZ_NOT_MPI if MPI is not used. Otherwise, *comm* should be the MPI communicator (if communicators are not used set this to MPI_COMM_WORLD). |

### Prototype

```
void AZ_set_VBR( AZ_MATRIX *Amat, int *rpntr, int *cpntr, int *bpntr, int *indx,
                 int *bindx, double *val, int *data_org, int N_update, int *update,
                 int option)
```

### Parameters

| | |
|---|---|
| *Amat* | On input, matrix created via AZ_matrix_create(). On output, *Amat* is labeled as a DVBR matrix and the the DVBR arrays *rpntr*, *cpntr*, *bpntr*, *indx*, *bindx*, *val*, and *data_org* are associated with it. |
| *rpntr, cpntr, bpntr, indx, bindx, val, data_org* | On input, DVBR matrix arrays. On output, these arrays are associated with *Amat*. |
| *N_update, update, options* | On input, should be set to *0, NULL, AZ_LOCAL* respectively. In the future, these parameters will allow for matrices with global indices. |

### Prototype

```
void AZ_solve(double *x, 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, $x$ is the initial guess. On output, $x$ is the linear system solution. NOTE: The size of $x$ is NOT equal to the number of matrix rows stored on the processor. Instead, it should contain $data_org[AZ_N_internal] + data_org[AZ_N_border] + data_org[AZ_N_external]$ elements (though external variables stored at the end of the vector do not need to be initialized). NOTE2: if AZ_transform was used to reorder the matrix $A$, AZ_reorder_vec can be used to transform the initial guess so that it is consistent with the reordered $A$ and AZ_invorder_vec can be used to transform the computed solution back to 'unreordered' form. |
| $b$ | A $data_org[AZ_N_internal] + data_org[AZ_N_border]$ length vector containing the right hand side. NOTE: if AZ_transform was used to reorder the matrix $A$, AZ_reorder_vec can be used to transform the right hand side so that it is consistent with the reordered $A$. |
| *options, params* | Options and parameters used during the solution process (Section 2). |
| *status* | On output, status of iterative solver (Section 2). |

## Prototype

```
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

| | |
|---|---|
| *list1* | On input, values to be sorted. On output, sorted values (i.e. *list1[i]* $\leq$ *list1[i+1]*) |
| *N* | On input, length of lists to be sorted. |
| *list2* | 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. |
| *list3* | 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. |

## Prototype

```
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). See also AZ_reorder_vec and AZ_invorder_vec to transform the right hand side and solution vectors.

## Parameters

| | |
|---|---|
| *external* | On output, allocated and set to components that must be communicated during the matrix vector multiply. |
| *bindx, val, index, bpntr, rpntr* | 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. |
| *update_index* | On output, allocated and set such that $update_index[i]$ is the local numbering corresponding to $update[i]$. |
| *extern_index* | On output, allocated and set such that $extern_index[i]$ is the local numbering corresponding to $external[i]$. |

| | |
|---|---|
| *data_org* | On output, allocated and set to data layout information, see Section 6.1. |
| *cpntr* | On output, allocated and set for VBR matrices to the column pointer array. |
| *mat_type* | On input, matrix format: either AZ_VBR_MATRIX or AZ_MSR_MATRIX. |

### Prototype

```
void AZ_VBR_matvec_mult( double *b, double *c, AZ_MATRIX *A, int *proc_config)
```

### Description

Perform the matrix-vector multiply

$$c \leftarrow Ab$$

where the matrix $A$ is a DVBR matrix (see Section 7).

### Parameters

| | |
|---|---|
| *b* | On input, distributed vector to use in multiplication. NOTE: *b* should contain *A->data_org*[AZ_N_internal] + *A->data_org*[AZ_N_border] + *A-> data_org*[AZ_N_external] elements (though external variables stored at the end of the vector do not need to be initialized). |
| *c* | On output, the result of matrix-vector multiplication. NOTE: *c* should contain *A->data_org*[AZ_N_internal] + *A->data_org*[AZ_N_border] elements. |

### Prototype

```
void AZ_version(char *str )
```

## Description

Record the current version of **Aztec** that is being used in *str*.

## Parameters

| | |
|---|---|
| *str* | On input, space for at least 15 characters must be allocated. On output, *str* is filled with a string representing the current version of **Aztec** that is being used. |

## References

[1] S. Carney, M. Heroux, and G. Li. A proposal for a sparse BLAS toolkit. Technical report, Cray Research Inc., Eagen, MN, 1993.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.

[3] I. Gustafsson. A class of first-order factorization methods. *Bit*, 18:142–156, 1978.

[4] B. Hendrickson and R. Leland. The Chaco user's guide - version 1.0. Technical Report Sand93-2339, Sandia National Laboratories, Albuquerque NM, 87185, August 1993.

[5] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for systems of which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31(137):148–162, 1977.

[6] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[7] J. N. Shadid and R. S. Tuminaro. Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation. *Concurrency: Practice and Experience*, 4(6):481–497, September 1992.

[8] J. N. Shadid and R. S. Tuminaro. Aztec - a parallel preconditioned Krylov solver library: Algorithm description version 1.0. Technical Report Sand95:in preparation, Sandia National Laboratories, Albuquerque NM, 87185, August 1995.

[9] R. S. Tuminaro, J. N. Shadid, and S. A. Hutchinson. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, August 1997.

[10] Z. Zlatev, V.A. Barker, and P.G. Thomsen. SSLEST - a FORTRAN IV subroutine for solving sparse systems of linear equations (user's guide). Technical report, Institute for Numerical Analysis, Technical University of Denmark, Lyngby,Denmark, 1978.