Linear Algebra Library for Mentat Applications

Laurie MacCallum and Andrew S. Grimshaw

Technical Report No. CS-93-43

August 6, 1993

This work was partially funded by NSF grants ASC-9201822, CDA-8922545-01, and grant JPL-959303.

Table of Contents

1. Introduction

Programming CPU intensive applications is a difficult task, and parallelizing the code is even more difficult. The programmer must not only program the application but manage synchronization, memory management and communication as well. Many of these applications require the use of matrices. What is provided is a set of linear algebra routines that make this task easier. What is provided is a library of classes that manipulate two-dimensional arrays that are contained within memory or files. From these classes, two Mentat classes were created to perform the standard linear algebra functions in parallel. These routines can be used to implement more interesting and difficult applications easily while achieving modest performance gains over sequential programs.

The class, *DD_array,* forms the base for the linear algebra library. DD_array is a C++ class that serves as a useful tool in parallel applications. It provides an interface for memory-contiguous matrices and vectors. The class also provides several means of decomposing arrays. Finally, it provides the basic linear algebra operations.

However, the class *DD_array* operates solely in memory. Many interesting problems require the use of matrices that are too large to fit into memory. In order to reduce the performance degradation due to disk swapping, another C++ class, *DD_array_file,* was developed to allow users to manipulate files or portions of files containing *DD_arrays*. The user is provided with an interface that will read and write from or to files directly into or out of *DD_arrays*.

The classes *DD_array* and *DD_array_file* form the basis for the Mentat classes that will allow the computationally intensive linear algebra functions to execute in parallel. These classes were used to create a regular Mentat class, *matrix_ops*. This class forms the basis of a linear algebra library while providing transparent parallelism to the user. Currently, the library includes member functions for addition and multiplication by a scalar, transpose, matrix-vector multiply, matrix-matrix multiply and Gaussian elimination using partial pivoting.

One of largest setbacks to achieving performance gains is the cost of distributing the data to the various processors. In many cases, performance gains occur only when the data remains distributed as successive operations are performed on it. To realize this potential for performance improvement, a persistent Mentat class, *p_matrix*, was created. The class provides the user with the ability to specify the manner in which the data is to be distributed among the processors. The functions defined on this class include addition and multiplication by a scalar value, transpose, matrix-vector multiply, matrix-matrix multiply, and Gaussian elimination using partial pivoting.

Performance measurements for both the regular and persistent classes for the operations matrix-vector multiply, matrix-matrix multiply, and Gaussian elimination using partial pivoting perform as expected. Matrix-vector multiply is not computationally intensive enough to justify the cost of parallelization. Matrix-matrix multiply displays relatively good performance gains. The results for Gaussian elimination are not as large as for matrix-matrix multiply, but they are encouraging considering the amount of synchronization necessary. Overall, the regular class performs better than the persistent class. However, the persistent class can provide better performance results when successive operations are performed on the data.

In what follows, the base classes *DD_array, dense_vector, sparse_vector,* and

DD_array_file are presented along with their interfaces. The class *matrix_ops* is then introduced, and performance measurements for the linear algebra library functions are presented. Finally, the persistent class, *p_matrix is* introduced, and its interface is discussed. Finally, performance measurements for linear algebra library functions are presented.

2. Class DD_array

The foundation for the Linear Algebra Library is the class *DD_array. DD_arrays* provide an interface for the manipulation of arrays and vectors that meet two requirements for programming within the Mentat environment. The data structures are stored contiguously in memory. Further, the user is provided with a wide range of methods to decompose the larger data structures into smaller parts. The classes derived from *DD_array* support the standard linear algebra operations.

DD_array serves as a base class from which various types of arrays can be derived. The element types of these matrices can be one of integer, single precision floating point, double precision floating point, or character. The actual type names of these classes are: *DD_intarray, DD_floatarray, DD_doublearray,* and *DD_chararray,* respectively. Also derived from this class are row or column vectors of type integer, single precision floating point or double precision floating point, *int_vector, sp_dvector and dp_dvector,* respectively. The relationship among the hierarchy is shown below. Additional types for either arrays or vectors, such as complex numbers, can be derived from the base class, *DD_array.*[1]

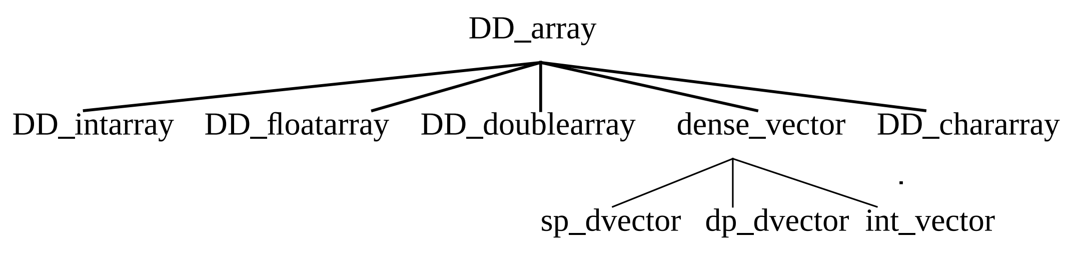


Figure 1: DD_array Class Hierarchy

One of the most important services that can be provided by a library for use in a parallel/distributed environment is the ability to decompose the data to be sent to the various processors. The class DD_array provides functions that allow the user to decompose the matrix into one of five forms: by row, by column, cyclically by row, cyclically by column, or by block.

One of the most useful means of decomposing a matrix is to "slice" the matrix by rows or columns into even pieces. These row and column decompositions are used

1. In the examples that follow, the type *DD_floatarray* is referenced, but may be substituted with any of the other type classes except where noted.

extensively in the Mentat classes described below. A row decomposition consists of dividing the array into a number of subarrays, each of which represents a number of consecutive rows of the array. Each of the subarrays consists of the number of rows divided by the number of pieces. In the event that the number of pieces does not divide evenly into the number of rows, the subarrays, starting at the first, each contain an extra row until the difference is covered (See Figure 2). The analogous operation can be performed in a column-wise fashion as well. The *DD_array* methods, *decompose_by_rows* and *decompose_by_columns* return an array of pointers to each of the subarrays, where each subarray is a part of the decomposed array.

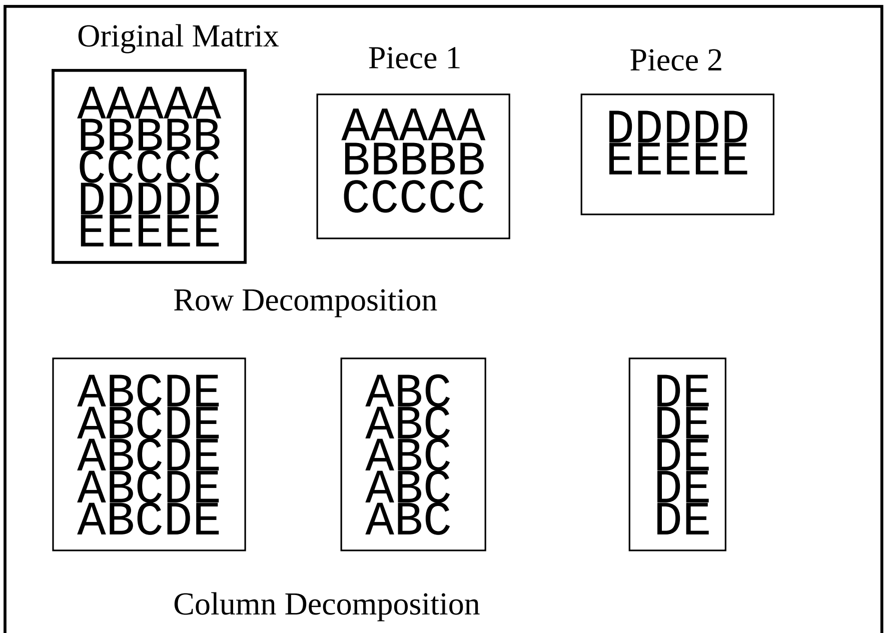


Figure 2: DD_array Decomposition Types

When manipulating arrays that have the property that they are diagonally dominant, it is often useful for load balancing reasons to cyclically decompose the matrix in a manner that permutes the order of the diagonal entries. A row-cyclic decomposition consists of subarrays of the same size had a row decomposition occurred, but each of the rows in the subarrays does not represent consecutive rows of the parent array. The first row of each of the subarrays represents the first rows of the parent array, and likewise through the array (See Figure 3). An analogous function is provided for a column-cyclic decomposition. The DD_array member functions, *cyclic_decompose_by_row* and *cyclic_decompose_by_column* produce an array of pointers to sub-arrays that represent a portion of the decomposition.

Decomposing a matrix into blocks can be useful for stencil algorithms. A block decomposition is a combination of a row and column decompositions. The user specifies the number of pieces into which the rows should be divided and likewise for the number of columns. The number of rows in each subarray is the number of rows in the entire matrix

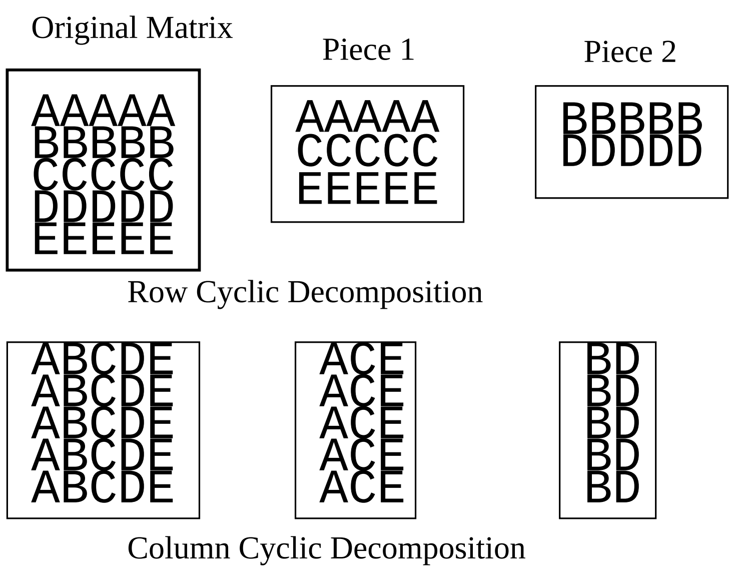


Figure 3: Cyclic Decomposition Types

divided by the number of row pieces; the number of columns in each subarray is the number of columns in the entire matrix divided by the number of column pieces. Should the rows and columns not divide evenly into the number of pieces for specified for each direction, then the excess will be placed in the upper-most rows and left most columns of the subarrays (See Figure 4). The *DD_array* member function *decompose_by_block* returns an array of pointers to the subarrays comprising a block decomposition. The pointers are ordered in a "row major" fashion: the subarrays representing the lowest numbered rows for increasing numbered columns are represented first in the return array.

Further, the user is not limited to the decompositions provided. The member functions *extract_region* and *overlay_region* can be used to create other decompositions. The member function *extract_region* takes as its arguments the upper left row and column indices and the lower right row and column indices and returns an array consisting of the elements contained within those boundaries. The member function *overlay_region* takes as its arguments the upper left row and column indices and a source array. The elements in the destination array starting at the boundary are replaced with the elements in the source array (See Figure 5).

Finally, the class *DD_array* provides the user with several linear algebra functions: addition, subtraction, and multiplication by a scalar, addition of two matrices, and multiplication of two matrices and the multiplication of a matrix and a vector.

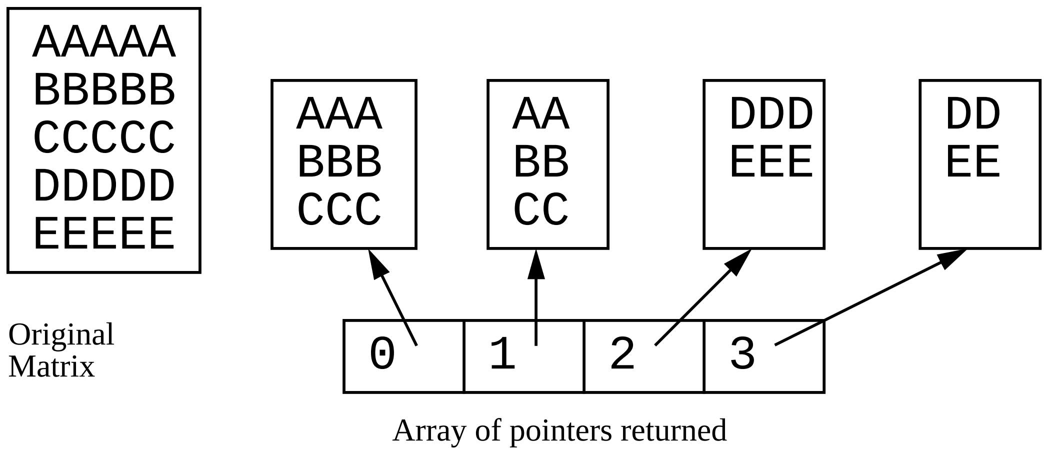


Figure 4: Block Decomposition into Four Pieces

2.1. Interface

The following describes the methods defined within the class. For explanatory purposes, *DD_arrays* of type float are used, yet the methods are identical for *DD_chararray, DD_doublearray,* and *DD_intarray* except where noted. It is important to bear in mind that array indices follow the C convention: for example, if an array is of dimension five, its indices range from zero to four.

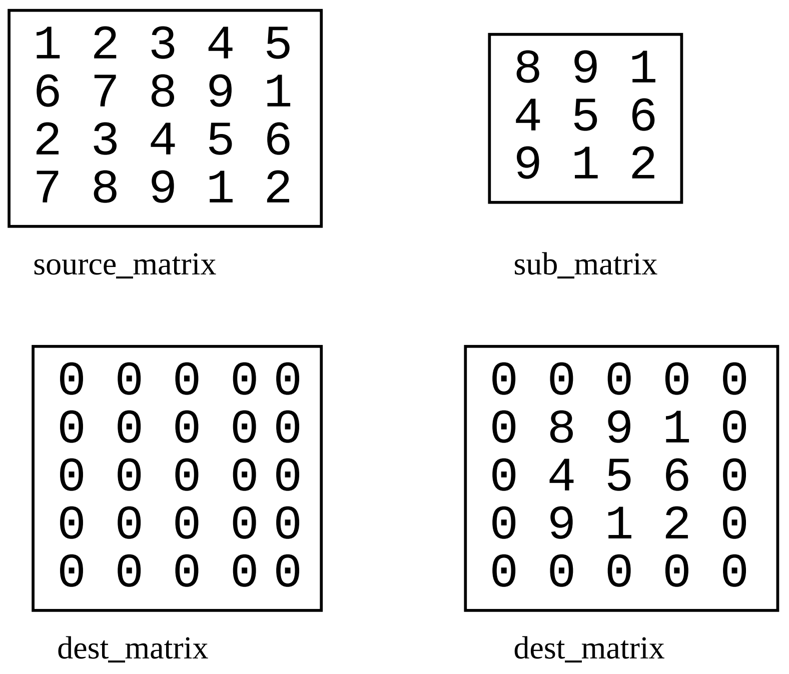
2.1.1. Creating a Two Dimensional Array Instance

DD_floatarray(DD_floatarray_file *file_class_name)

This constructor creates a new array stored in the file represented by *file_class_-name*. *File_class_name* is an instance of the class *DD_floatarray_file* described below. The file represented must have previously been opened for either READ or READ_AND_WRITE. In the event that the file has not been properly opened, an array of size 0 and status -1 is created.
An equivalent method exists for each of the classes *DD_chararray, DD_doublear-ray,* and *DD_intarray.*

DD_floatarray (int rows, int cols)
DD_floatarray(int rows, int cols, float *ptr)

These constructors create an array instance having *rows* rows and *cols* columns. In the former case, the array is zero-filled. In the latter case, the array is filled with the



```
sub_matrix = source_matrix->extract_region(1,2,4,4);
dest_matrix->overlay_region(sub_matrix,1,1);
```

Figure 5: Example of *extract_region and overlay_region*

values stored at address *ptr* in row major order.
Equivalent methods exist for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

2.1.2. Creating an Instance of DD_array from an Existing Instance

DD_floatarray(DD_floatarray *src)

This constructor creates an array instance which duplicates the array pointed to by *src*.
An equivalent constructor exists for the class *DD_doublearray.*

DD_floatarray *extract_region(int ul_row, int ul_col, int lr_row, int lr_col)

This method extracts a region of the array bounded by [*ul_row, ul_col*] on the upper left corner, and [*lr_row, lr_col*] on the lower right corner of the source matrix. The resulting region is returned as a *DD_floatarray.*[1] Figure 5 displays an example of such a decomposition.

1. Note that the order of the arguments has been changed. In prior releases, the column index was specified first.

In the event that the boundaries specified are out of range in the source matrix, the array is returned with zeros filled into the rows and/or columns that are out of range.
An equivalent method exists for each of the classes *DD_chararray, DD_doublearray*, and *DD_intarray*.

DD_floatarray *extract_piece_by_columns(int pieces, int the_piece)
DD_floatarray *extract_piece_by_rows(int pieces, int the_piece)
These methods decompose the array into *pieces DD_floatarrays*, column-wise or row-wise, and returns the *the_piece* piece from the decomposition. If the number of columns or rows in the array does not divide evenly, the excess columns or rows are added to each of the pieces, beginning with the first. The desired part of the decomposition is returned as a *DD_floatarray*. Figure 2 displays an example of such a decomposition where the return value for this method is one of the subarrays.

2.1.3. Accessing Array Elements

float *get_r_ptr(int row_num)
This method returns a pointer to the element in row *row_num*, column 0. The following statement demonstrates how the pointer returned can be indexed to access an element of a particular row.

```
float element = source_matrix->get_r_ptr(1)[3];
/*If source_matrix is defined as above, then element = 7.*/
```

An equivalent method exists for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray* each of which return a character, double, or integer pointer, respectively.

float &element(int row, int column)
This method returns a reference to element in row *row*, column *column*.The following sequence of statements demonstrates how this method can be used to access and change values in an array.
An equivalent method exists for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray* each of which return a character, double, or integer reference, respectively.

2.1.4. Decomposing DD_arrays

DD_floatarray **decompose_by_column(int pieces)
DD_floatarray **decompose_by_row(int pieces)
These methods decompose the source array into *pieces* arrays, row-wise or column-wise as shown in Figure 2. An array of pointers to each of the subarrays, in their respective orders within the source array.

Equivalent methods exist for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

DD_floatarray **cyclic_decompose_by_column(int pieces)
DD_floatarray **cyclic_decompose_by_row(int pieces)

These methods decompose the source array into *pieces*. Each column or row is distributed, in order, among each of the subarrays as shown in Figure 3. An array of pointers to each of the subarrays is returned.
Equivalent methods exist for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

DD_floatarray **decompose_by_block(int row_pieces, int col_pieces)

This method decomposes the source array into *row_pieces * col_pieces* subarrays. The source array is decomposed first as in a row decomposition, dividing the array into row_pieces pieces. A column decomposition is then performed creating the appropriate number of pieces. An example of such a decomposition is shown in Figure 4. An array of pointers to the newly created subarrays is returned. The subarrays are ordered by column decomposition order within row decomposition order.
An equivalent method exists for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

2.1.5. Array Manipulation Methods

void overlay_region(int ul_row, int ul_col, DD_floatarray *data)
void overlay_region(int ul_row, int ul_col, sp_dvector *data)

These methods copy the array or vector, *data*, onto the source array starting at row *ul_row*, column *ul_col*. In the event that data is too large to be completely copied, only those regions of data are copied onto the source.This is an in place operation, therefore nothing is returned.[1] An example is shown in Figure 5.
Equivalent methods exist for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

DD_floatarray *transpose()

This method returns the transpose of the source array.
An equivalent method exists for each of the classes *DD_chararray, DD_doublearray,* and *DD_intarray.*

void print()

This method prints the contents of the source array to standard output. For a Mentat application, this would be the window in which the Mentat system was started.

1. Note that the order of the arguments has been changed. In previous releases, the column index was specified first.

2.1.6. Mathematical Functions

void operator += (float scalar)
void operator += (DD_floatarray *data)

These operators performs in-place addition of either a scalar value or the array *data.* For the latter method, if the dimensions of the source and the argument do not match, no action is taken. The following statement increments each array element by 3.4:

```
(*source_matrix)+=3.4;
/*This statement adds the value 3.4 to all of the array elements*/
```

Equivalent methods exist for each of the classes *DD_doublearray*, and *DD_intarray*

void operator -= (DD_floatarray *data)

This method performs in-place subtraction of *data* from the source array. If the dimensions of the source and the argument do not match, no action is taken.
Equivalent methods exist for each of the classes *DD_doublearray*, and *DD_intarray*

DD_floatarray operator *= (float scalar)

This method performs in-place multiplication of the source array with *scalar.*

An equivalent method exists for each of the classes *DD_doublearray*, and *DD_intarray*

DD_floatarray * operator * (DD_floatarray *data)
sp_dvector * operator(sp_dvector *data)

These methods perform matrix-matrix multiply and matrix-vector multiply of the source matrix and *data* and returns the result.
In the event that the arrays are non-conformant (the number of columns of the source does not equal the number of columns of the destination), a 0 x 0 array is returned with a status of -1.Equivalent methods exist for the classes *DD_chararray, DD_doublearray*, and *DD_intarray* each of which return a character, double, or integer pointer, respectively.
Equivalent methods exist for each of the classes *DD_doublearray*, and *DD_intarray.*

2.2. Dense Vectors

Often, when manipulating arrays, there is a need to manipulate vectors as well. The classes, *sp_dvector, dp_dvector*, and *int_vector* have been developed for this purpose. An instance of *sp_dvector* represents a single precision floating point dense vector. Likewise, an instance of *dp_dvector* represents a double precision floating point dense vector. Finally, an instance of *int_vector* represents an integer vector. With each dense vector is associated the number of elements and a status code.

2.2.1. Creating a Dense Vector Instance

sp_dvector(sp_dvector_file *file_class_name)

This constructor creates an instance of *sp_dvector* where the elements are stored in the file represented by *file_class_name*. This method is analogous to the method *DD_floatarray(DD_floatarray_file *file_class_name)*.
An equivalent constructor exists for each of the classes *dp_dvector* and *int_vector.*

sp_dvector(int cols)
sp_dvector(int cols, float *ptr)

These constructors create an instance of *sp_dvector* where *cols* is the number of elements in the vector. The latter method initializes the vector with the values stored at address *ptr.* These methods are analogous to the corresponding methods for the class *DD_floatarray.*
All vectors are created as row vectors. To create a column vector, the constructor needs to be followed with a call to *transpose* (described below).
Equivalent methods exist for the classes *dp_dvector* and *int_vector.*

2.2.2. Creating an Instance of Dense Vector from an Existing Instance

sp_dvector *duplicate()

This method creates a new vector that is a duplicate of its source. Equivalent methods exist for the classes *dp_dvector* and *int_vector.*

sp_dvector *extract_subvec(int first, int last)

This method extracts a subvector from the source vector starting with the element at *first* and ending with element at *last*.
Equivalent methods exist for the classes *dp_dvector* and *int_vector.*

sp_dvector *extract_piece(int pieces, int the_piece)

This method extracts a portion of the source vector to create a new vector. This method is analogous to the method *row_decompose* for the class *DD_floatarray.*
Equivalent methods exist for the classes *dp_dvector* and *int_vector.*

2.2.3. Accessing Vector Elements

float *get_vec()

This method returns a pointer to the first element in the vector. Elements of the vector can be accessed by indexing from this pointer as with the method *get_r_ptr* in the class *DD_floatarray.*
Equivalent methods exist for the classes *dp_dvector* and *int_vector*, each of which

return a character, double, or integer pointer, respectively.

2.2.4. Accessing the State of a Dense Vector

int is_row_vec()
This method returns TRUE if the vector is a row vector; FALSE otherwise.

int is_col_vec()
This method returns TRUE if the vector is a column vector; FALSE otherwise.

int num_elements()
This method returns the number of elements contained in the vector.

2.2.5. Decomposing Dense Vectors

sp_dvector **decompose(int pieces)
This method decomposes the vector into *pieces* subvectors and returns an array of pointers to the subvectors. This method is analogous to the method *decompose_by_row* for the class *DD_floatarray.*
An equivalent method exists for the classes *dp_dvector* and *int_vector.*

sp_dvector **cyclic_decompose(int pieces)
This method performs a cyclic decomposition of the vector. This method is analogous to the method *row_cyclic_decompose* for the class *DD_floatarray.*
Returns an array of pointers to the decomposed subvectors.
An equivalent method exists for the classes *dp_dvector* and *int_vector.*

sp_dvector *cyclic_reorder(int num_pieces)
This method reorders the elements of a vector assuming the order was permuted cyclically. It returns the vector whose elements are reordered.
An equivalent method exists for the classes *dp_dvector* and *int_vector.*

2.2.6. Dense Vector Manipulation Methods

void transpose()
This method performs an in-place transpose of the vector.

void overlay_region(int offset, sp_dvector *data)
This method overlays the elements of data starting at element index *offset*.The source vector is modified, so there is no return value.
An equivalent method exists for the classes *dp_dvector* and *int_vector.*

void print()
This method prints the elements of the vector to standard output.

void operator+=(float arg)
void operator+=(sp_dvector *arg)
void operator*=(float arg)

These operators perform scalar addition, vector-vector addition, and scalar multiplication, respectively.
Equivalent methods exist for the classes *dp_dvector* and *int_vector.*

float dot_product(sp_dvector *arg)

This method performs the dot product of the source and *arg*. The result is returned. If the vectors are not the same size, the value 0.0 is returned.
An equivalent method exists for the classes *dp_dvector* and *int_vector.*

3. Class DD_array_file

It is often desirable to save the results from one set of computations for future use. In addition, many scientific problems involve arrays too large to fit in memory at one time. Their component parts may fit in memory, yet the whole array may not, resulting in performance degradation due to disk swapping. By allowing the user to easily manipulate files or portions of files, the results of computation can be preserved, and the entire array need not be brought into memory, only the portions necessary at a particular host.

The class *DD_array_file* provides a class of file operations that make this task simpler. These classes are designed to be used with the class *DD_array.*

The class *app_file* defines the low level file manipulation routines and can be used as a base for any type of data to be stored on disk. This class maintains the file pointer, the magic number, the file type, the current file status, and the mode in which the file was opened. Each of these items is discussed in greater detail below. The application progammer need never be concerned with the methods involving this class. The methods used to manipulate this class are protected and cannot be accessed by a user. However, examination of these methods is useful when deriving new file formats off of this class.

The class *DD_array_file* implements the file interface for two dimensional arrays. Derived from *DD_array_file* are five classes that can be used by the application programmer. These classes are: *DD_intarray_file, DD_floatarray_file, DD_doublearray_file, DD_chararray_file, dense_vector_file,* and contain arrays of integers, single precision floating point, double precision floating point, characters, and vectors, respectively. The class *dense_vector_file* serves as a base class for three vector classes: *int_vector_file, sp_dvector_file,* and *dp_dvector_file.*

A file consists of a header and the array elements stored in row major order. The file header contains: the magic number, the file type, the size of the data elements (in bytes), the number of rows and columns of the array, and the array elements. The magic number is a means of ensuring that the file version is consistent with the library functions. It is currently set to 556. The file type is a means of ensuring that the type of structure contained within the file is consistent with the library functions. Currently, only files of two dimensional arrays and vectors (types *DD_array* and *dense_vector*) are supported. The file type number is 575. The state of a file consists of its status, the mode in which it was opened, and the number of rows and columns contained in the file. The status of a file can any one of the following: OPEN, CLOSED, MAGIC_NUM_ERROR, OPEN_ERROR, HEADER_READ_ERROR, HEADER_WRITE_ERROR, FILE_TYPE_HEADER_ERROR, and BAD_HEADER. Each of these status codes are discussed below. The modes in which a file can be opened are: READ, READ_AND_WRITE, and WRITE. These modes are also discussed in detail below.

When using these classes, it is important that the file permissions allow for global access or at least Mentat group access. This is due to the fact that when a user runs a Mentat application, it runs under the Mentat user id, not that of the individual user.

3.1. File Initialization Methods

DD_floatarray_file(string *fname, int mode)

This constructor creates and opens the file, where *file_name* is the name of the file and *mode* is the mode in which the file is to be opened. The allowable modes are: READ, READ_AND_WRITE, and WRITE. A file opened for READ allows the file to be read. A file opened for WRITE allows the file to be written to. However, if a file is opened for WRITE, any existing file will be erased. This is the only mode in which a file will be created. A file opened for READ_AND_WRITE may be either read from or written to. However, the file must already exist. If the mode is READ or READ_AND_WRITE, the header information is read from the file and the file is ready to be manipulated. If the mode is WRITE, then the header information must be explicitly written using a call to *file_alloc* (described below). The status is set to OPEN if the open command is successful. In the event that the open command is not successful, the status is set to one of the following error codes: OPEN_ERROR, indicating that the file does not exist, the file permissions have not been properly set, or the disk is full, MAGIC_NUM_ERROR, indicates that the version number contained in the file is inconsistent with the version number expected by the class, FILE_TYPE_HEADER_ERROR, indicating that the file type for is inconsistent, or HEADER_READ_ERROR or HEADER_-WRITE_ERROR, indicating that there was an error while reading or writing the header information (maybe the file is malformed). Further, an open command may fail if the system-defined limit for number of file pointers allowed for one file is exceeded. SUCCESS is returned upon successful completion, FAILURE otherwise.

An equivalent method exists for the classes *DD_chararray_file*, *DD_doublearray_file*, and *DD_intarray_file*.

int file_alloc(DD_floatarray *data)
int file_alloc(int num_row, int num_col)

These methods prepare a file that has been opened for write to accept data. *Data* is a *DD_floatarray* from which the number of rows and columns can be extracted. The header information is written to the file.

They return SUCCESS if the header is successfully written. In the event that the header information is not written, FAILURE is returned, and the status of the file is set to HEADER_WRITE_ERROR.

This function only prepares the file for writing, in order to write to the file, a call to one of the methods for writing to a file (described below) must be made. The file must already have been opened, and its mode must be WRITE.

Equivalent methods exist for the classes *DD_chararray_file, DD_doublearray_-file,* and *DD_intarray_file*.

int close_file()

This method closes the file, resets the file status to CLOSED and mode to 0. Returns a 0 on SUCCESS, or 1 on FAILURE, depending on the outcome of the close.

3.1.1. Accessing the State of a File

int get_row_size()
int get_col_size()
int get_data_type_size()
int get_header_size()

The first two methods return the number of rows and columns contained within the source file, respectively. The latter two methods return the size of the data to be stored in the file, and the size of header within the file, respectively.

void set_row_size(int rows)
void set_col_size(int cols)
void set_data_type_size(int size)
void set_header_size(int size)

Each of these methods set the private member variables *num_rows*, *num_cols*, *data_type_size*, and *header_size*.

3.1.2. Reading from a File

DD_floatarray *read_row(int row)

This method reads the row specified by the parameter *row* from the file, places it in a 1 x *num_columns DD_floatarray* and returns the array. The arrays is returned after a successful read, otherwise, an empty array whose status is -1 is returned. *Row* must be a valid row within the range specified in the header of the file.

An equivalent method exists for the classes *DD_chararray_file, DD_doublear-ray_file,* and *D*D_intarray_file.

DD_floatarray *read_row(int row, int start_col, int end_col)

This method reads the row specified by the parameter row from the file, places it in a 1 x (*end_col-start_col*) *DD_floatarray* and returns the array. The arrays is returned after a successful read, otherwise, an empty array whose status is -1 is returned. I *Row* must be a valid row within the range specified in the header of the

file. *Start_col* and *end_col* must also be within the appropriate ranges.
Equivalent methods exist for the classes *DD_chararray_file, DD_doublearray_-file*, and *DD_intarray_file*.

DD_floatarray *read_col(int column_number)
This method reads the column specified by *column_number* into an appropriately sized *DD_floatarray.* The array is then returned. In the event of an error, an empty array whose status is -1 is returned. *Column_number* must be within valid ranges as specified in the file header.
Equivalent methods exist for the classes *DD_chararray_file*, *DD_doublearray_-file*, and *DD_intarray_file*.

DD_floatarray *read_region(int ul_row, int ul_col, int lr_row, int lr_col)
This method reads the portion of the array bounded by [*ul_row, ul_col*], [*lr_row, lr_col*], creates and places it a new *DD_floatarray,* and returns the result. In the event of a read error, an empty array whose status is -1 is returned. The region must be a valid region within the range specified in the header of the file.
An equivalent method exists for the classes *DD_chararray_file*, *DD_doublearray_file,* and *DD_intarray_file*.

DD_floatarray *read_array()
Reads the entire array from a file, creates and places it in a new *DD_floatarray* and returns the result. In the event of a read error, an empty array, whose status is -1, is returned.
An equivalent method exists for the classes *DD_chararray_file*, *DD_doublearray_file,* and *DD_intarray_file*.

3.1.3. Writing to a File

int write_row(DD_floatarray *data, int row)
This method writes row *row* from the *DD_floatarray data* to the corresponding position in the file and returns SUCCESS or FAILURE depending on the outcome of the write.
Row must be a valid row within the range specified in the header of the file.
An equivalent methods exists for the classes *DD_chararray_file, DD_doublearray_file,* and *DD_intarray_file*.

write_row(DD_floatarray *data, int row, int start_col, int end_col)
This method writes row *row* between columns *start_col* and *end_col* from the *DD_floatarray data* to the corresponding position in the file. It returns SUCCESS or FAILURE depending on the outcome of the write. *Row, start_col, end_col* must be a valid within the ranges specified in the header of the file.
An equivalent method exists for the classes *DD_chararray_file, DD_doublearray_file,* and *DD_intarray_file*.

int write_col(DD_floatarray *data, int column_number)

This method writes column *column_number* of *data* to the corresponding position in the file. It returns SUCCESS or FAILURE depending on the outcome of the write.

Column_number must be within valid array bounds, and the number of rows in the array must equal those specified in the header.

An equivalent method exists for the classes *DD_chararray_file, DD_doublearray_file,* and *DD_intarray_file*.

int write_region(DD_floatarray *data, int ul_row, int ul_col, int lr_row, int lr_col)

This method writes the region of the *DD_floatarray, data,* bound by *[ul_row, ul_col], [lr_row, lr_col]* to the corresponding position in the file. It returns SUCCESS or FAILURE depending on the outcome of the write.

An equivalent method exists for the classes *DD_chararray_file*, *DD_doublearray_file*, and *DD_intarray_file*.

int write_array(DD_floatarray *data)

This method writes the entire *DD_floatarray*, *data*, to the file, and returns SUCCESS or FAILURE depending on the outcome of the write. The number of rows and columns in data must match those specified in the file header.

Equivalent methods exist for the classes *DD_chararray_file*, *DD_doublearray_-file*, and *DD_intarray_file*.

void print(string *file_name)

This method pens the file specified by *file_name*, prints the contents in row-major order to the standard output, and closes the file.

4. Regular Mentat Class Matrix_ops

The regular Mentat class *matrix_ops* utilizes both of the above classes to provide a means of performing computationally intensive array operations in parallel in a manner that is transparent to the user. The class *matrix_ops* provides several of the standard linear algebra functions. The class *matrix_ops* provides the operations for addition, subtraction, and multiplication by a scalar. Because the granularity of these operations is relatively small, these operations are performed sequentially. The linear algebra operations matrix-vector multiply, matrix-matrix multiply, and Gaussian Elimination using partial pivoting are performed in parallel. These operations are deemed to be computationally intensive enough to justify the overhead of communication.

Matrix-vector multiply is the first of the linear algebra functions implemented in parallel. The function *vec_mult* is invoked using either a file name, or a *DD_floatarray,* a vector, *sp_dvector*, and the number of pieces into which to decompose the computation. The matrix is decomposed by row into the specified number of pieces, and the member function *mult_work* is performed on each of the subarrays and the entire vector. Upon return, the newly calculated sub-vector is overlaid into the appropriate section of the resultant vector and the result is returned to the caller.

```
sp_dvector *vec_mult(string *file_name, sp_dvector *vector, int pieces)
 {
   //Performs Matrix-Vector Multiplication
   Create pieces instances of matrix_ops objects
    For each object, call mult_work to read from file and compute subvector
    Coalesce the subvectors
    Return the resultant vector
  }

sp_dvector *mult_work(string *file_name, sp_dvector *vector, int pieces,
                                   int piece_number)
  {
    Calculate region to be read from file based on a row decomposition and
         piece_number
    Read the region from the file
    Multiply the region with vector
     Return the subvector
   }
```

Figure 6: Pseudocode for Parallel Matrix-Vector Multiply

Matrix-matrix multiply is another linear algebra operation that is conducive to parallelization. The function *mult_mat* is invoked using either two *DD_floatarrays,* two file names, or a *DD_floatarray* and a file name. In all cases, the number of pieces into which the problem is to be decomposed is also included. The left hand matrix is then decomposed by row into the square root of *pieces* subarrays, and the right hand matrix is decomposed by columns into *pieces / x_pieces* subarrays. Each row “slice” is then multiplied by each column slice using the member function *mult_work*. The resulting arrays form a block decomposition of the resultant matrix. The result of each of these multiplications is then written to the output file. Unfortunately, this last step proves to be a bottleneck. Because the file server (NFS) does not follow Unix semantics in terms of file coherency, the file regions must be written sequentially. A future implementation is planned that will utilize an additional Mentat class to enable pipelining of file access to increase performance.

The functions matrix-vector multiply and matrix-matrix multiply are relatively easy to parallelize because there is no communication of intermediate results necessary between the various subarrays. Gaussian Elimination, on the other hand, requires communication between all of the subarrays at each iteration of the algorithm. The row with the largest value in the column corresponding to the iteration must be determined from amongst all of the subarrays, and this row must be sent to each of the subarrays. The function *solve* is invoked using either a *DD_floatarray* or a file name, an *sp_dvector*, and the number of pieces into which to decompose the computations. *Pieces* instances of a persistent Mentat class, *matrix_sub_block*, are created to hold the rows while the reduction is performed. The

```
int mult_mat(string *input_fi leA, string *input_fi leB, string *output_fi le, int
        pieces)
 {
   //Performs Matrix-Matrix Multiply
   Calculate the number of pieces for row decomposition for A matrix
   Calculate the number of pieces for column decomposition for B matrix
   For each row piece
     For each column piece
         Call mult_work to read the appropriate regions and calculate results
   Write each resulting subarray to the fi le
 }

DD_floatarray *mult_work(string *input_fi leA, string *input_fi leB, string
        *output_fi le, int x_pieces, int y_pieces, int my_x_piece, int my_y_piece)
 {
    Calculate the row region for fi rst matrix using my_x_piece
    Read the appropriate row block from input_fi leA
    Calculate the column region for fi rst matrix using my_y_piece
    Read the appropriate column block from input_fi leB
    Perform the multiplication
    Return the result
  }
```

Figure 7: Pseudocode for Parallel Matrix-matrix multiply

matrix is decomposed by row into *pieces* subarrays, which are then kept as part of the state within *matrix_sub_block*. Each of the *matrix_sub_blocks* calculates its local candidate pivot row, the row with largest value in the column being reduced, which is then reported back to the *matrix_ops* object through the return to future mechanism provided by Mentat while the rest of the subarray is reduced. The maximum of all of local maximums is determined and communicated to the subarrays, and each subarray reduces its rows using the pivot row. The process is repeated until all of the rows have been reduced. Back substitution is performed by the regular class, where the actual pivot rows have been saved between each iteration.

4.1. Interface

The following describes the methods defined in the regular Mentat class *matrix_ops.* The methods are currently defined for arrays and files of type *DD_floatarray.*

DD_floatarray *scal_add_mat(DD_floatarray *mat, float scalar)
DD_floatarray *scal_add_mat(string *mat_file_name, float scalar)

Performs scalar addition on the DD_floatarray mat or the array contained in mat_-file_name. These operations are performed sequentially due to their low computation granularity.
Returns the resultant DD_floatarray.

```
sp_dvector *solve(string *input_fi le, sp_dvector *vector, int pieces)
 {
   //Performs Gaussian Elimination using Partial Pivoting

   Create pieces instances of persistent objects, matrix_sub_block
   Have each of the persistent objects read the appropriate region from the fi le in
       a row decomposition
   Find the initial pivot row

   For each column in the array
     For each persistent object call reduce with current pivot row
       Find the maximum of the potential pivots
       Communicate new pivot row  to matrix_sub_block objects

   Perform back substitution
   Return result
 }

DD_floatarray *reduce(DD_floatarray *current_pivot)
 {
    Determine next candidate pivot with a partial reduction on next column
   Return new candidate pivot to caller
    Reduce the rest of the  subarray  with current pivot
   Return to caller
  }
```

Figure 8: Pseudocode for Parallel Gaussian Elimination with Partial Pivoting

DD_floatarray *scal_mult_mat(DD_floatarray *mat, float scalar)
DD_floatarray *scal_mult_mat(string *mat_file_name, float scalar)

Performs scalar multiplication on the DD_floatarray mat or the array contained in mat_file_name. These operations are performed sequentially due to their low computation granularity.
Returns the resultant DD_floatarray.

DD_floatarray *add_mat(DD_floatarray *mat1, DD_floatarray *mat2)
DD_floatarray *add_mat(string *mat_file_name, DD_floatarray *mat2)
DD_floatarray *add_mat(DD_floatarray *mat1, string *mat_file_name)
DD_floatarray *add_mat(string *mat_file_nameA, string *mat_file_nameB)

Performs matrix addition where the argument arrays are either in memory or contained in a file. Due to small amount of computation, these operations are performed sequentially.
Returns the resultant array.
In the event that the arrays are not the same size, an array of size zero and status of -1 is returned.

sp_dvector *mult_work(DD_floatarray *mat, sp_dvector *vect)
sp_dvector *mult_work(string *file_name, sp_dvector *vect, int pieces, int my_piece_number)

Multiplies the *DD_floatarray* represented by either *mat* or *file_name* by the vector *vect* and returns the result. The additional integer arguments for the latter method are used to determine which part of the file should be read.

sp_dvector *vec_mult(DD_floatarray *matrix, sp_dvector *vector, int pieces)
sp_dvector *vec_mult(string *matrix_file_name, sp_dvector *vector, int pieces)

Performs matrix vector multiply where the argument matrices are either in core or contained within the file *matrix_file_name*. The computation is split as evenly as possible among pieces workers in a row decomposition. This method determines the decomposition of the problem and calls *mult_work* to perform the actual computation.

Returns the resultant vector.

In the event that the number of columns of the matrix does not equal the number of rows of *vector*, a vector of size zero and status -1 is returned.

DD_floatarray *mult_work(DD_floatarray *mat1, DD_floatarray *mat2)
DD_floatarray *mult_work(DD_floatarray *mat, string *file, int pieces)
DD_floatarray *mult_work(string *file, DD_floatarray *mat, int piece, int my_piece_num)
DD_floatarray *mult_work(string *file1, string *file2, int x_pieces, int y_pieces, int my_piece_num)

Performs the multiplication between the argument *DD_floatarrays* which are represented by mat, mat1, or mat2, or files named file, file1, or file2. The resultant *DD_floatarray* is returned.The additional integer arguments are used to determine which part of the file should be read.

DD_floatarray *mult_mat(DD_floatarray* mat1,DD_floatarray* mat2,int pieces)
DD_floatarray *mult_mat(string *mat_file_name, DD_floatarray *mat2, int pieces)
DD_floatarray *mult_mat(DD_floatarray *mat1, string *mat_file_name, int pieces)
int mult_mat(string *input_fileA, string *input_fileB, string *output_file, int pieces)

Performs Matrix-matrix multiplication where the arguments are either in core DD_floatarrays or contained within files. The computation is divided into pieces sub-problems where the A matrix is decomposed into $\sqrt{pieces}$, row-wise, and the B matrix is decomposed into $(pieces) / (\sqrt{pieces})$, column-wise. This method determines the means by which the problem is decomposed and calls *mult_work* to perform the actual multiplication.

For the former three methods, it assumed that the result is small enough to fit into memory easily. Thus, the result is returned as a *DD_floatarray.* In the latter case, the resultant matrix is written to the file, *output_file,* and the integer value SUCCESS is returned.

In the event that the number of columns of the A matrix does not equal the number of rows of the B matrix, an array of size zero with status is returned or FAILURE in the latter case.

sp_dvector *solve(DD_floatarray *mat, sp_dvector *vect, int pieces)
sp_dvector *solve(string *mat_file_name, sp_dvector *vect, int pieces)

Performs Gaussian Elimination using partial pivoting as outlined above.
Returns the solution vector.
If the matrix is not square or not invertible, a vector of size zero and status -1 is returned.

The class *matrix_sub_block* is used to maintain the state of the decomposed, reduced matrices while performing Gaussian Elimination.

void initialize(DD_floatarray *the_block, sp_dvector *vec)
void initialize(string *file_name, sp_dvector *vec, int pieces, int my_piece_num)

Initializes each of the submatrices with either *the_block* or the *DD_floatarray* contained in *file_name*. The arguments *pieces* and *my_piece_num* are used in the latter method to determine which part of the matrix should be read from the file.

DD_floatarray *get_max(int column)

Returns a *DD_floatarray* containing the row with the largest value in column *column*.

DD_floatarray *reduce(DD_floatarray *by_row)

Reduces the submatrix using the pivot row, *by_row*. The next candidate pivot is returned before the rest of the matrix is reduced.

4.2. Performance Analysis

In order to determine the effectiveness of the class *matrix_ops* in completing computationally intensive operations, several performance measurements were taken on a network of sixteen SUN IPCS using the NFS file server. Measurements were taken for matrix-vector multiply, matrix-matrix multiply, and Gaussian elimination with partial pivoting. Each problem was run using matrices that were contained in a file (file-based). Matrices of sizes 512 x 512 and 1024 x 1024, 1536 x 1536, and 2048 x 2048 floating point numbers were used. They were decomposed into four, six, eight, ten, twelve, and fourteen pieces, respectively. The Mentat run-time system managed the scheduling of each of the subarrays onto a processor. The scheduler was configured to base its placement decisions on CPU utilization (with a threshold of 50%), and round robin selection. The tests were run five times independently of one another during periods of low network load.

Measurements were taken of the total time to complete the operation, including the time to read the data in from the file. The sequence of events was: start the timer, invoke the function, use the result in some quick manner, stop the timer. It was necessary to "touch" the result to force synchronization of all of the objects. Between each test of the functions, the regular objects were removed.[1]

1. The Mentat run time system does not normally delete regular objects between invocations in order to reduce overhead incurred for object creation, should subsequent objects be needed.

The sequential times were taken from a heavily optimized C++ version of the respective operations, *not* the performance of the Mentat version run with one processor. The sequential times were measured for 512 x 512 and 1024 x 1024 matrices. The sequential results for the larger sized matrices were then obtained by multiplying the serial time for 1024 case by the square of the ratios of the sizes for matrix-vector multiply, and the cube of the sizes for matrix-matrix multiply and Gaussian Elimination. For the larger matrices, this approach is extremely conservative in favor of the serial times: on one processor, the amount of time needed to perform a matrix-matrix multiply for matrices of size 2048 x 2048 is much greater than a factor of 8. This is due to the effects of paging because the SUN IPC's memory cannot hold sixteen megabytes of data in memory at once. In both cases, CC_saber's compiler level two optimization option was used.

Tables 1 through 3 display the average execution times over five trials and the lowest execution time of the trials. Table 4 displays the lowest execution time over two trials. Tables 5 through 8 display the corresponding speedups. Speedup is calculated as the ratio of the serial execution time to the parallel execution time.

Table 1: Execution Times of Class matrix_ops 512 x 512 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	2.47	6.20	2.84	3.27	2.29	1.53	2.41
Lowest time		1.92	1.52	1.57	1.61	1.10	2.77
Matrix-Matrix Multiply	279.58	85.24	52.67	47.37	44.54	40.93	46.20
Lowest time		80.56	49.09	44.36	32.90	32.94	31.67
Gaussian Elimination	78.91	43.29	28.30	31.30	36.08	41.62	49.25
Lowest time		30.53	26.77	27.58	33.19	37.87	43.69

Table 2: Execution Times of Class matrix_ops 1024 x 1024 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	20.148	7.87	7.59	6.78	6.82	6.17	5.15
Lowest time		5.59	5.55	4.36	4.45	4.72	3.76
Matrix-Matrix Multiply	2270.93	608.29	383.45	385.75	299.56	340.26	307.10
Lowest time		597.86	365.42	320.95	246.72	235.58	232.49
Gaussian Elimination	626.78	183.94	136.09	132.63	113.97	147.37	146.31
Lowest time		179.21	131.04	109.34	108.07	118.33	126.81

Table 3: Execution Times of Class matrix_ops 1536 x 1536 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	45.33	45.96	19.19	20.51	21.67	19.78	20.18
Lowest time		17.21	17.13	14.07	16.12	16.05	16.75
Matrix-Matrix Multiply	7664.40	2617.53	2509.49	1667.91	1511.40	1353.24	1143.32
Lowest time		2138.52	1528.97	1160.70	1130.92	954.56	805.91
Gaussian Elimination	2115.39	906.80	496.40	532.79	435.23	329.65	402.23
Lowest time		565.58	406.29	325.47	317.28	299.08	305.68

Table 4: Execution Times of Class matrix_ops 2048 x 2048 Matrix (in seconds)[1]

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	80.59	24.66	22.67	22.67	29.94	32.97	35.56
Matrix-Matrix Multiply	18167.47	5004.95	3281.97	3246.70	2339.31	2062.77	2068.13
Gaussian Elimination	5014.26	1393.96	917.96	783.46	687.67	723.20	1064.80

Table 5: Speedup of Class matrix_ops for 512 x 512 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	0.40	0.87	0.76	1.08	1.62	1.03
Lowest Time	1.29	1.63	1.58	1.54	2.27	2.24
Matrix-Matrix Multiply	3.28	5.31	5.90	6.28	6.83	6.05
Lowest Time	3.47	5.69	6.30	8.50	8.49	8.91
Gaussian Elimination	1.82	2.79	2.54	2.19	1.90	1.60
Lowest Time	2.58	2.95	2.86	2.38	2.08	1.81

1. This table displays the best (lowest) execution time for two independent tests.

Table 6: Speedup of Class matrix_ops 1024 x 1024 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	2.56	2.65	2.97	2.96	3.27	3.91
Lowest Time	3.61	3.63	4.62	4.53	4.27	5.35
Matrix-Matrix Multiply	3.73	5.92	5.89	7.58	6.67	7.39
Lowest Time	3.80	6.21	7.08	9.20	9.64	9.77
Gaussian Elimination	3.41	4.61	4.73	5.50	4.25	4.28
Lowest Time	3.50	4.78	5.73	5.80	5.30	4.94

Table 7: Speedup of Class matrix_ops for 1536 x 1536 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	0.99	2.36	2.21	2.09	2.29	2.25
Lowest Time	2.63	2.65	3.22	2.81	2.83	2.71
Matrix-Matrix Multiply	2.93	3.05	4.60	5.07	5.6	6.70
Lowest Time	3.58	5.01	6.60	6.78	8.03	9.51
Gaussian Elimination	2.33	4.26	3.97	4.86	6.42	5.26
Lowest Time	3.74	5.21	6.50	6.67	7.07	6.92

Table 8: Speedup of Class matrix_ops for 2048 x 2048 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	3.27	3.53	3.55	2.97	2.78	2.78
Matrix-Matrix Multiply	3.63	5.54	6.16	7.99	9.48	8.78
Gaussian Elimination	3.60	5.46	6.40	7.82	8.21	5.39

4.3. Observations

Overall, the performance gains are mixed. Matrix-vector multiply exhibits only a slight performance gain. Matrix-matrix multiply performs exceptionally well. The performance gains for Gaussian Elimination with partial pivoting are modest.

For regular Mentat objects, performing matrix vector multiply in parallel results in a small performance gain. For each matrix size, the speedup quickly reaches a peak speedup and does not increase, irrespective of the number of processors used. Two factors weigh heavily in this result. The operation itself is not computationally intensive. Its approximate running time is $O(n^2)$ where n is the size of the matrix. For a larger number of submatrices (and consequently more processors), the cost of communication is too great to be amortized over this small amount of computation. Further, the cost of computation is dwarfed by the cost of reading the data in from a file. Therefore, performing matrix-vector multiply using regular file-based classes is not recommended.

For Gaussian elimination, the results are more encouraging. The maximum speedup attained increases with the size of the matrix. Further, this maximum occurs for a larger number of pieces as the size of the matrix increases. However, before and after the maximum speedup has been achieved, the speedup appears to remain close to the maximum. This type of result is to be expected due to the need to communicate intermediate results (the next pivot row) for each iteration. This forces a synchronization point which limits the amount of performance gain that can be achieved.

It would be expected that of the three computations, matrix-matrix multiply would exhibit the largest speedups. This is due in part to the fact that the granularity of the computation is larger than for matrix-vector multiply. Further, there is no data dependency between the component parts of the computation as with Gaussian elimination. The results bear this assumption out. The speedups remain almost linear for all problem sizes. The performance gains are only limited by the synchronization needed at the end of the computation to write the resulting matrix to the output file sequentially. Yet, despite this needed synchronization point, matrix-matrix multiply performs well, as would be expected for any trivially data parallel problem.

Although the performance results are good, in all likelihood, they can be improved. The current implementation relies on NFS to allow each *matrix_ops* object to read its part of the file. For a large number of processors, this translates into heavy network load for the Ethernet. By creating a Mentat class to pipeline file access, the performance degradation due to initialization of the private member variables should be diminished. Further, the effect of the necessary synchronization for writing the results to the file for matrix-matrix multiply would be reduced, further improving performance.

In general, the results presented here are very conservative. As was stated before, the speedups were based on an extrapolated estimate of the sequential running time for a problem size that could fit entirely in memory. For larger sized problems, however, the sequential time may be quite a bit slower as the result of disk swapping. Thus, the ability to decompose the problem into parts that fit entirely in memory in and of itself offers a performance gain.

5. Persistent Mentat Class P_matrix

Although the regular Mentat class *matrix_ops* offers modest performance improvements for matrix-matrix multiplication and Gaussian elimination, they are of limited utility when successive operations are to be performed on the same matrices. Often, the user knows the best means of decomposing a problem, or would like to perform a sequence of operations on the data. In this situation, it is useful to amortize the cost of the initialization and communication overhead by preserving the state of the decomposed matrix between invocations. Although the overhead of initialization dwarfs that of the regular classes, this cost is negligible when compared to the cost of decomposing and communicating the data for each function invocation.

To realize the performance gain between successive operations, each *p_matrix* instantiation must maintain state between invocations. State information for this class includes the total number of rows and columns, the manner in which the matrix was decomposed, the number of subarrays into which the matrix was decomposed, an indicator as to whether or not the matrix originated from a file and its file name, and pointers to each of the subarray instances. This state information is named *total_rows, total_cols, num_pieces, decomp_type, file_flag, filename,* and *sub_arrays*, respectively.

The subarrays themselves also comprise a class, *p_matrix_sub_block*. The use of this class is completely transparent to the user. The *p_matrix_sub_block* instance maintains each of the subarrays provided by the *p_matrix* class. The state information includes the upper left row and column and lower right row and column of its subarray, relative to the entire matrix, and the appropriate portion of the *DD_floatarray* itself. This state information is named *ul_row, ul_col, lr_row, lr_col,* and *the_matrix,* respectively.In addition, the state may also contain two additional structures for use while performing Gaussian elimination: a *DD_floatarray* consisting of the reduced matrix (named *reduced_matrix*), and an array of integers denoting which of the rows of the reduced matrix have actually been reduced (named *used*), and the current column on which the reduction is taking place (named *current_col*).

The user specifies how many subarrays to create, and how the array is to be decomposed: by row, by column, by block, cyclic by row, or cyclic by column. Either *DD_floatarrays* or file names may be used in the initialization. The subarrays are created and the data is distributed as per the user's directives. The subarrays become instances of the class *p_matrix_sub_block*, which actually performs all of the desired operations.

With the persistent class, all of the operations are performed in parallel. The operations provided are scalar addition and multiplication, which change the value of the distributed subarrays. An extract function is also provided which will return a specified portion of the matrix irrespective of the original decomposition. The array can also be transposed in place. For debugging purposes, the function retrieve returns each of the subarrays in the manner that they have been decomposed.

5.1. Interface

The following describes the methods defined within the class. The class *p_matrix* serves as a "master", distributing the data and coordinating activities between the "slaves", the class *p_matrix_sub_block*. The class is currently only defined for matrices of type single precision floating point.

int initialize(DD_floatarray *data, int decomp_type, int pieces)
int initialize(string *fname, int decomp_type, int pieces)

> Initializes the private member variables, and distributes the matrix contained in the *DD_floatarray data* or file named *fname* based into pieces number of subarrays among the class *p_matrix_sub_block*. The subarrays are distributed in a manner determined by *decomp_type*. The decomposition is one of: BY_ROW, BY_COLUMN, CYCLIC_BY_ROW, CYCLIC_BY_COL, BY_BLOCK.
> Returns SUCCESS when the initialization is completed.
> In the event that the number of pieces is less than the number of rows or columns, the number of subarrays created will be the number of rows or columns, depending on the decomposition specified.

int initialize_UT_matrix()

> Creates subsidiary data structures necessary for Gaussian elimination. The structures created are the reduced_matrix which contains the portion of the subarray that has been reduced by Gaussian Elimination with Partial Pivoting and an array of flags denoting which rows have been reduced. The structures are maintained between invocations of *solve* to reduce execution time of subsequent operations.
> Returns SUCCESS.

int destroy_UT_matrix()

> Destroys the data structures created by *initialize_UT_matrix*.
> Returns SUCCESS.

int get_total_rows()
int get_total_cols()
int get_num_pieces()
int get_decomp_type()

> Returns the values of the private member variables *total_rows*, *total_cols*, *num_pieces*, and *decomp_type*, respectively.

int scalar_add_mat(float scalar)
int scalar_mult_mat(float scalar)

> Multiplies the subarrays by the value of scalar. The value of the subarrays is updated.
> Returns SUCCESS.

int transpose()

> Performs an in-place transpose of the matrix. The decomposition type of the matrix is adjusted to reflect this change.
> Matrices decomposed CYCLIC_BY_ROW and CYCLIC_BY_COL are not transposed.
> Returns SUCCESS.

DD_floatarray *extract(int ul_row, int ul_col, int lr_row, int lr_col)

> Returns a portion of the original matrix bounded by rows *ul_row* and *lr_row* and columns *ul_col* and *lr_col*. The *DD_floatarray* is returned in the expected (original) order, irrespective of the decomposition of the subarrays.

DD_floatarray *retrieve()

> Returns the entire matrix in its original order, irrespective of the decomposition of the subarrays.

sp_dvector *vec_mult(sp_dvector *arg_vect)

> Performs matrix-vector multiply. Returns the resultant vector.
> In the event that the matrix and *arg_vect* are not conformant, a vector of size zero and status of -1 is returned.
> There is a performance penalty for matrices whose decomposition type is not BY_ROW or CYCLIC_BY_ROW as the subarrays will be redistributed in a row decomposition.

DD_floatarray *mat_mult(DD_floatarray *data)

> Performs matrix-matrix multiply where the first matrix is the *p_matrix* and the second is the *DD_floatarray data*.
> Returns the resultant matrix. In the event that the matrices are not conformant, a matrix of size zero and status of -1 is returned.
> There is a performance penalty for matrices whose decomposition type is not BY_ROW or CYCLIC_BY_ROW as the subarrays will be redistributed in a row decomposition.

int mat_mult(string *input_file, string *output_file)

Performs matrix-matrix multiply where the first matrix is the *p_matrix* and the second is contained in the file *input_file*.

The result is written to file *output_file*. In the event that the matrices are not conformant, FAILURE is returned. In the event that the files cannot be opened, FAILURE is returned.There is a performance penalty for matrices whose decomposition type is not BY_ROW or CYCLIC_BY_ROW as the subarrays will be redistributed in a row decomposition.

sp_dvector *solve(sp_dvector *rhs_vector)

Performs Gaussian Elimination using partial pivoting.

Returns the resultant vector. In the event that the matrices are not conformant, a vector of size one and status -1 is returned. There is a performance penalty for matrices whose decomposition type is not BY_ROW or CYCLIC_BY_ROW as the subarrays will be redistributed in a row decomposition.

void print()

Prints the values of the entire matrix to standard output. For Mentat applications, this is the window in which Mentat was initialized.

int clean_up()

Destroys all of the subarrays and returns any dynamically allocated memory to the heap.

5.2. CLASS P_MATRIX_SUB_BLOCK

5.2.1. Interface

The following describes the methods defined within the class *p_matrix_sub_block*. The class *p_matrix* serves as a “master”, distributing the data and coordinating activities between the “slaves”, the class p_matrix_sub_block. The class is currently only defined for matrices of type single precision floating point.

int initialize(DD_floatarray *array)
int initialize(string *fname)
int initialize(DD_floatarray *array, int first_row, int first_col, int last_row, int last_col)
int initialize(string *fname, int first_row, int first_col, int last_row, int last_col)

Initializes the private member variables for an instance of *p_matrix_sub_block*. In the first two cases, the member variable *the_matrix* is initialized to the entire array and the member variables, *ul_row, ul_col, lr_row*, and *lr_col* are set to the dimensions of the array. In the latter two cases, *the_matrix* initialized to the portion of the array specified by the input parameters, and the member variables are set accordingly.

int row_cyclic_decompose(string *file_name, int the_pieces, int my_piece_num)
int col_cyclic_decompose(string *file_name, int the_pieces, int my_piece_num)

Initializes the private member variables *the_matrix, ul_row, ul_col, lr_row,* and *lr_col* by reading from the file in a row or column cyclic decomposition. (See Figure 3).

int return_ul_row()
int return_ul_col()
int return_lr_row()
int return_lr_col()

Returns the private member variable corresponding to the coordinates relative to the entire matrix.

int return_num_rows()
int return_num_cols()

Returns the number of rows or columns of *the_matrix,* respectively.

DD_floatarray *retrieve_matrix()

Returns the private member variable, *the_matrix*, to the caller. To be used primarily as a debugging mechanism.

sp_dvector *get_column(int col_num)
sp_dvector *get_row(int row_num)

Returns the column or row of *the_matrix* as specified by the argument. The column or row is returned as a column or row vector.

DD_floatarray *get_max_row(int column)

Returns the row of *the_matrix* with the largest value in column *column*. The row is returned as a *DD_floatarray.*

DD_floatarray *extract(int first_row, int first_col, int last_row, int last_col)

Returns a *DD_floatarray* consisting of those rows and columns that lie within the boundary specified by the arguments.

int transpose()

Performs an in-place transpose of *the_matrix.*

int scalar_add_mat(float scalar)
int scalar_mult_mat(float scalar)

Performs in-place scalar addition and multiplication on *the_matrix*, respectively.

sp_dvector *vec_mult(sp_dvector *arg_vect)

Performs matrix-vector multiply. The resulting subvector is returned. In the event of an error (the matrix and vector sizes are non-conformant), a vector of size 0 and status -1 is returned.

DD_floatarray *mat_mul(DD_floatarray *ip_matrix)
DD_floatarray *mat_mul(string *input_file, int pieces)

Performs matrix-matrix multiply. The_matrix serves as the left-hand operand, and the array *ip_matrix* or the array stored in the file input_file is used for the right-hand argument. The resulting matrix is returned as a *DD_floatarray.* In the event of an error (the matrices are not matching sizes or a file access error), a *DD_floatarray* of size 0 and status -1 is returned.

int initialize_reduced_matrix()
int destroy_reduced_matrix()

Initializes and destroys the private member variable, *reduced_matrix*, with *the_-matrix*, and creates the array, *used*. This is called in preparation for Gaussian Elimination where the reduced matrix contains the rows reduced with pivoting and the array *used* is a flag to determine which rows have been used as the pivot row.

int setup_solver(sp_dvector (rhs_vector, int decomp_meth)

Prepares the instance for Gaussian elimination. *Reduced_matrix* is initialized with *rhs_vector.*

DD_floatarray *gauss_eliminate(DD_floatarray *pivot_row)

Performs one iteration of the Gaussian elimination algorithm. This method determines the next candidate pivot returns this row to the *p_matrix* instance and reduces all of its rows using *pivot_row.*

5.3. Performance Analysis

In order to determine the effectiveness of the class *p_matrix* in completing computationally intensive operations, several performance measurements were taken on a network of sixteen SUN IPCS using the NFS file server. Measurements were taken for matrix-vector multiply, matrix-matrix multiply, and Gaussian elimination with partial pivoting. Each problem was run using matrices that were contained in a file (file-based). Matrices of sizes 512 x 512 and 1024 x 1024, 1536 x 1536, and 2048 x 2048 floating point numbers were used. They were decomposed into four, six, eight, ten, twelve, fourteen, and sixteen pieces, respectively. The Mentat run-time system managed the scheduling of each of the sub-arrays onto a processor. The scheduler was configured to base its placement decisions on CPU utilization (with a threshold of 50%), and round robin selection. The tests were run five times independently of one another during periods of low network load.

Four separate measurements were taken: the initialization time, matrix-vector multiply, matrix-matrix multiply, and Gaussian elimination. Initialization time includes the amount of time to initialize each instance of *p_matrix_sub_block*, the determine which part of the matrix each worker was to manage, and read the portion of the file from disk. The latter three times include the amount of time needed to complete the operation and return its result.

The sequential times were taken from a heavily optimized C++ version of the respective operations, *not* the performance of the Mentat version run with one processor. The sequential times were measured for 512 x 512 and 1024 x 1024 matrices. The results for the larger sized matrices were then obtained by multiplying the serial time for 1024 case by the square of the ratios of the sizes for matrix-vector multiply, and the cube of the sizes for matrix-matrix multiply and Gaussian Elimination. For the larger matrices, this approach is extremely conservative in favor of the serial times: on one processor, the amount of time needed to perform a matrix-matrix multiply for matrices of size 2048 x 2048 is much greater than a factor of 8. This is due to the effects of paging because the SUN IPC's memory cannot hold sixteen megabytes of data in memory at once. In both cases, CC_saber's compiler optimization option was used.

Tables 9 through 12 display the execution times, and Tables 13 through 16 display the speedups.Tables 9 through 11 display the average execution times over five trials and the lowest execution time of the trials. Table 12 displays the lowest execution time over two trials. Tables 13 through 16 display the corresponding speedups. Speedup is calculated as the ratio of the serial execution time to the parallel execution time.

Table 9: Execution Times of Class p_matrix 512 x 512 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Initialization Time		3.33	6.01	5.56	5.27	4.69	4.82
Lowest time		1.76	3.86	4.20	4.04	4.08	4.57
Matrix-Vector Multiply	2.47	3.56	6.22	5.79	5.48	4.92	5.08
Lowest time		1.97	4.07	4.40	4.24	4.29	4.80
Matrix-Matrix Multiply	279.58	88.57	67.00	58.29	48.78	50.33	47.10
Lowest time		85.06	61.08	53.00	44.94	39.56	41.51
Gaussian Elimination	78.91	32.19	30.78	32.41	36.88	42.61	48.64
Lowest time		28.85	27.73	28.21	33.34	39.11	46.57

Table 10: Execution Times of Class p_matrix 1024 x 1024 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Initialization Time		8.10	13.57	16.64	13.47	18.22	11.85
Lowest time		3.97	12.21	10.14	11.29	9.54	9.97
Matrix-Vector Multiply	20.15	8.83	14.12	17.20	13.93	18.67	12.33
Lowest time		4.65	12.70	10.57	11.68	9.92	10.35
Matrix-Matrix Multiply	2270.93	676.46	438.90	419.17	372.62	347.50	304.85
Lowest time		621.02	411.29	374.70	295.76	289.41	248.38
Gaussian Elimination	626.78	192.80	145.09	127.87	130.18	136.97	143.23
Lowest time		176.78	135.38	112.61	110.76	117.51	136.07

Table 11: Execution Times of Class p_matrix 1536 x 1536 Matrix (in seconds)

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Initialization Time		24.28	29.45	36.96	36.55	30.63	30.80
Lowest time		19.26	20.51	25.06	27.85	28.97	20.50
Matrix-Vector Multiply	45.33	26.20	30.81	38.91	37.51	33.13	32.66
Lowest time		20.73	21.51	25.85	28.54	29.58	21.07
Matrix-Matrix Multiply	7664.40	2391.32	2002.37	1633.42	1338.56	1309.29	1092.20
Lowest time		2144.74	1721.36	1466.38	1209.59	1029.60	938.91
Gaussian Elimination	2115.39	603.64	449.58	361.55	330.85	324.29	339.31
Lowest time		568.15	406.08	323.20	286.37	282.42	292.54

Table 12: Execution Times of Class p_matrix 2048 x 2048 (in seconds)[1]

Problem Description	Sequential Execution	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Initialization Time		36.52	35.46	50.55	53.22	50.65	51.54
Matrix-Vector Multiply	80.59	43.59	53.60	52.29	54.50	51.84	53.10
Matrix-Matrix Multiply	18167.47	5452.52	3615.95	3542.02	3461.82	3365.08	4166.65
Gaussian Elimination	5014.26	1380.27	1024.17	769.22	666.73	654.26	680.18

1. Note that the times listed in this table represent the best (lowest) execution time for two executions.

Table 13: Speedup of Class p_matrix for 512 x 512 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	0.69	0.40	0.43	0.45	0.50	0.49
Lowest Time	1.25	0.60	0.56	0.58	0.58	0.52
Matrix-Matrix Multiply	3.16	4.17	4.80	5.73	5.56	5.94
Lowest Time	3.29	4.58	5.27	6.22	7.07	6.74
Gaussian Elimination	2.45	2.56	2.43	2.14	1.85	1.62
Lowest Time	2.73	2.85	2.80	2.36	2.02	1.69

Table 14: Speedup of Class p_matrix for 1024 x 1024 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	2.28	1.43	1.17	1.45	1.08	1.63
Lowest Time	4.34	1.59	1.91	1.73	2.03	1.95
Matrix-Matrix Multiply	3.36	5.17	5.42	6.09	6.54	7.45
Lowest Time	3.66	5.52	6.06	7.68	7.85	9.14
Gaussian Elimination	3.25	4.32	4.90	4.81	4.58	4.38
Lowest Time	3.55	4.63	5.57	5.66	5.33	4.61

Table 15: Speedup of Class p_matrix for 1536 x 1536 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	1.73	1.47	1.16	1.21	1.37	1.39
Lowest Time	2.19	2.11	1.75	1.59	1.53	2.15
Matrix-Matrix Multiply	3.21	3.83	4.69	5.73	5.85	7.02
Lowest Time	3.57	4.45	5.23	6.34	7.44	8.16
Gaussian Elimination	3.50	4.71	5.85	6.39	6.52	6.23
Lowest Time	3.72	5.21	6.55	7.39	7.49	7.23

Table 16: Speedup of Class p_matrix for 2048 x 2048 Matrices

Problem Description	Four Pieces	Six Pieces	Eight Pieces	Ten Pieces	Twelve Pieces	Fourteen Pieces
Matrix-Vector Multiply	1.85	1.50	1.54	1.48	1.55	1.52
Matrix-Matrix Multiply	3.33	5.02	5.13	5.25	5.40	4.36
Gaussian Elimination	3.63	4.90	6.52	7.52	7.66	7.37

5.4. Observations

Overall, the performance results parallel those obtained with the regular class *matrix_ops*. Matrix-vector multiply exhibits very poor speedups for smaller sized problems. Matrix-matrix multiply results in a performance gain of slightly less than 50% of the number of processors at its best. Gaussian elimination exhibits better performance for larger problem sizes.

The performance gains for Matrix-vector multiply are relatively poor. The speedups hover around two. As was mentioned above, the problem is trivially data parallel with communication restricted to data distribution and result collection, the amount of computation is small relative to the amount of communication. Therefore, it would not be expected that performing matrix-vector multiply in parallel would significantly improve its performance.

Matrix-matrix multiply, on the other hand, displays good performance gain. Almost linear speedups are obtained at smaller number of processors. However, for a larger number of processors, the speedups, although increasing, do not increase as quickly as would be expected with a data parallel problem. One hindrance to what would be expected to be linear speedups is the effect of file I/O for both input of the argument matrix and output of the result matrix. The argument matrix each of the *p_matrix_sub_block* instances must read the entire matrix from the file in a column-wise fashion. Each of the these column accesses necessitates reading the entire file into the file cache due to the row-major layout of the array on disk. Further, it is necessary to write the resultant array to the file sequentially within the *p_matrix* object. This forces synchronization at the end of the computation. Because matrix-matrix multiply is trivially data parallel, good speedups can be obtained despite the necessary synchronization for file access.

The performance gains were also good for Gaussian Elimination with partial pivoting. The algorithm requires a great deal of communication (not in terms of the message size but in the number of messages required) and is necessarily data parallel, as severe load balance could be a problem. It is interesting to note that for each of the problem sizes, there is little deviation from the peak speedup. This is due to the fact that the speedup is limited by the need to calculate and communicate the new pivot row to each of the processors which limits the speedup obtained.

In general, the results presented here are very conservative. As was stated before, the speedups were based on an extrapolated estimate of the sequential running time for a problem size that could fit entirely in memory. For larger sized problems, however, the sequential time may be quite a bit slower as the result of disk swapping. Thus, the ability to decompose the problem into parts that fit entirely in memory in and of itself offers a performance gain.

6. Comparison of Regular and Persistent Matrix Operations

The large question is whether or not to use regular or persistent classes. In almost every instance the regular classes perform better than the persistent classes. This is due to the fact that there is a great deal of overhead associated with persistent classes. The full utility of the persistent classes can be seen when successive operations are performed on the data.

All of the tables of the execution times list out the initialization time and the execution time of the respective operation. The initialization time is only incurred once for the persistent classes. Thus, when successive operations are performed, the larger initialization time is amortized over each of the operations. With the regular classes, the initialization time must be paid for each operation performed. For matrix-vector multiply, for example, the 1536 x 1536 case with 8 pieces, a speedup of 5.5 is obtained for the persistent classes as opposed to a speedup of 2.21 for the regular classes when the function is performed four times. Likewise, for Gaussian elimination, given the same problem size number of pieces, the speedup is 6.92 as opposed to a speedup of 6.50 when the problem is run four times.

For matrix-matrix multiply, the persistent classes are considerably slower than the regular classes. For the persistent class, the matrix is already decomposed. Thus, each of the instances is sequentially multiplying its subarray the entire argument matrix. For the regular class, this is done in parallel. Therefore, the most compelling reason to use the persistent class for matrix-matrix multiply is when is used in conjunction with other matrix operations, provided the penalty for the persistent matrix-matrix multiply is overcome.

7. Conclusions

We have created a set of base classes that provide an easy to use class of two-dimensional arrays suitable for use with parallel and distributed applications because of their memory contiguous structure and the ability to decompose an array into several subarrays. In addition, we have provided a file structure and interface to effectively manipulate large arrays stored in files. With these two classes, two different linear algebra libraries were created. The first uses transparent parallelism through the use of Regular Mentat objects. The second library allows the user more control over the parallelism through the use of Persistent Mentat objects. Although both libraries offer reasonable speedups for matrix-matrix multiply and Gaussian Elimination with partial pivoting, the regular classes achieve greater performance gains. However, the persistent classes offer the advantage that the objects maintain state through each invocation. Therefore, performance gains can be realized when successive operations are performed on the data. Using these libraries as a base, a user may build larger, more complex applications easily while achieving performance gains.

8. Directions for Future Work

In order to further improve the Linear Algebra several modifications to the existing library will be made. The class *DD_array* will be expanded to include sparse matrices. Further, other data structures may be added so the library may be used in applications involving other types of data structures. The Gaussian Elimination function *solve* will be modified to allow for subsequent solutions using only one iteration of the LU factorization. Further, a function to compute the inverse may also be added.

The most important modification will involve the pipelining of file access. In order to achieve this, the class *DD_array_file* will be promoted to a Mentat class. This new class will not only pipeline file access but allow for more efficient access for columns within the array. In the current implementation, accessing a column of the array within a file requires that the entire file be read. With this modification in place, the performance of the classes *matrix_ops* and *p_matrix* should improve dramatically.