(HPSC **5576** ELIZABETH JESSUP)

HIGH PERFORMANCE SCIENTIFIC COMPUTING

:: Homework / 5

:: Student / F. Rappl

2 problems / **25** points

Problem 1

Task:

Programming assignment 06.10.03 from Pacheco's PPMPI textbook: Dense transpose (p. 110), *5 pts*

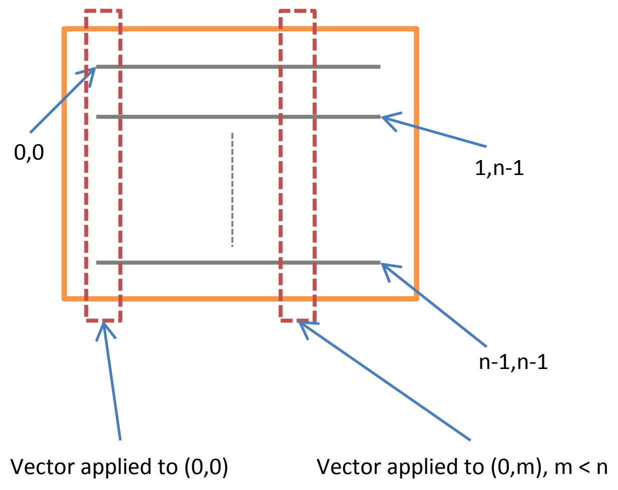
Write a function that transposes a dense matrix while sending it from one process to another. To do this, create a derived datatype (MPI_Type_vector of, say, MPI_INT) representing a column in the matrix. The sending process should transmit data using the column derived datatype, but the receiving process should accept the data using a standard datatype (e.g., MPI_INT).

Solution:

By just using the implemented type vector we are able to get certain values out of a big array, which can be a matrix. By settings the starting pointer to a different location we are therefore able to send a column but receive it as a row – i.e. doing a transpose operation. This can be very useful in general (not the transpose operation itself – but such operations that can be implemented in a send / receive) – in order to save computation time.

My solution is pretty straight forward. Most of the code (like the command line parser) was used in previous homework. Therefore I will just write about the interesting parts. First of all we have to declare our own datatype as MPI_Datatype. After this is done we have to initialize it using the MPI_Type_vector function, which declares how this vector is used (how many blocks should be taken – how many skipped, etc.). In the final step we synchronize this declaration with all other running processes, in order to keep consistency.

Then we built the original matrix and send it using our own datatype (in this case *column_t*). For the send receive we make a little trick. We send using a pointer to the i-th column, while we receive using a pointer to the i-th row. This procedure can be explained using the definition of our own datatype:



SEND

MPI_Type_vector(n, 1, n, …):
Result has n components
Always take only 1 component then skip
Do n skips

RECEIVE

Treat result as array of MPI_INT:
Therefore the logic behind column_t will not apply but we get the data. By using a pointer to (m,0) we get the starting pointer for the m-th row. Then we fill the row with the n MPI_INT vector (from the array).

Program output:

```
D:\Documents\Visual Studio 2005\Projects\Whatever\Debug>mpiexec -n 2
whatever -n 6
New matrix:
1       7       13      19      25      31
2       8       14      20      26      32
3       9       15      21      27      33
4       10      16      22      28      34
5       11      17      23      29      35
6       12      18      24      30      36
Original matrix:
1       2       3       4       5       6
7       8       9       10      11      12
13      14      15      16      17      18
19      20      21      22      23      24
25      26      27      28      29      30
31      32      33      34      35      36
```

Code printout:

```c
#define STANDARDCOUNT 5
#include <stdio.h>
#include <string.h>
#include "mpi.h"

void printMatrix(int** matrix, int dim);

int main(int argc, char* argv[])
{
	int		my_rank;	/* rank of process	*/
	int		p;		/* number of processes	*/
	int		tag = 0;	/* tag for messages	*/
	char		my_name[64];	/* name of machine	*/
	int		my_name_len;	/* length of my_name	*/
	MPI_Status	status;		/* return status (recv) */
	int		n, k, i, j;	/* loop counters	*/
	int**		matrix;		/* matrix of int values */
	MPI_Datatype column_t;		/* own datatype - column*/
	/* dimension of the matrix */
	n = STANDARDCOUNT;
	/* Command line args parser */
	for(i = 1; i < argc; i++)
	{
		/* if we have the -n */
		if(strcmp(argv[i],"-n") == 0)
		{
			/* but nothing else specified */
			if(i == argc - 1)
			{
				printf("Dimension of Mat. Expected -n.\n");
				break;
			}
			/* or we probably have a number */
			n = atoi(argv[++i]);
			if(n < 1)
			{
				/* but that is not a valid number */
				printf("Wrong input for n.\n");
				n = STANDARDCOUNT;
			}
			}
		}
		/* Start up MPI */
		MPI_Init(&argc, &argv);
		/* Find out process rank  */
		MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
		/* Find out number of processes */
		MPI_Comm_size(MPI_COMM_WORLD, &p);
		if(p < 2)
		{
			/* Shut down MPI if p < 2 */
			MPI_Finalize();
			/* and end program */
			return 0;
		}
		/* Get machine name */
		MPI_Get_processor_name( my_name, &my_name_len );
		/* define new MPI datatype */
		MPI_Type_vector(n, 1, n, MPI_INT, &column_t);
		MPI_Type_commit(&column_t);
		/* allocate matrix */
		matrix = (int**)malloc(sizeof(int*) * n);
		/* allocate everything without spaces etc. */
		matrix[0] = (int*)malloc(sizeof(int) * n * n);
		k = 0;
		for(i = 0; i < n; i++)
		{
			/* set pointers correctly */
			matrix[i] = matrix[0] + ( i * n );
			/* set arbitrary matrix */
			if(my_rank == 1)
				for(j = 0; j < n ; j++)
					matrix[i][j] = ++k;
		}
		if(my_rank == 0)
		{
			/* receive the columns but spawn them as rows */
			for(i = 0; i < n; i++)
				MPI_Recv(&(matrix[i][0]), n, MPI_INT, 1, tag,
						MPI_COMM_WORLD, &status);
			/* print out transpose matrix */
			printf("New matrix:\n");
			printMatrix(matrix, n);
		}
		else if(my_rank == 1)
		{
			/* print out original matrix */
			printf("Original matrix:\n");
			printMatrix(matrix, n);
			/* send the original matrix columns */
			for(i = 0; i < n; i++)
				MPI_Send(&(matrix[0][i]), 1, column_t, 0, tag,
						MPI_COMM_WORLD);
		}
		/* Shut down MPI */
		MPI_Finalize();
		return 1;
} /* main */

void printMatrix(int** matrix, int dim)
{
	int i, j;
	for(i = 0; i < dim; i++)
	{
		for(j = 0; j < dim; j++)
			printf("%d\t", matrix[i][j]);
		printf("\n");
	}
	return;
} /* printMatrix */
```

Problem 2

Task:

Programming assignment 06.10.04 from Pacheco's PPMPI textbook: Sparse transpose (p. 110), *10 pts*

Write a function that transposes a sparse matrix while sending it from one process to another using pack and unpack.

Solution:

The problem here is based on matrices which do have lots of zeros. I wrote a little random sparse matrix function to generate the matrix used for output and testing purposes. While I will explain the *MPI_Pack() / MPI_Unpack()* calls later on I first want to focus on another important aspect of this program. One requirement was to not have a full matrix at any point in program. At the same time we should transpose the matrix by just sending. Since the datatype for the sparse matrix rows stores only nonzero value count, column positions of values and the values itself which are not zero, we cannot directly transpose this. Therefore we just have two possibilities:

- Sending all the data first, storing them in a sparse matrix and inverting this matrix
- Using the information on the senders side, gathering any column vector out of the sparse matrix and then sending the specified column vector

Since the assignment in Pacheco states that a function should be written I decided for the last one. So I've written a function called *getColumn()* which takes the sparse matrix, the desired column and the dimension of the matrix (number of rows) as in parameters and the outgoing sparse row (to be used as a column) as an out parameter.

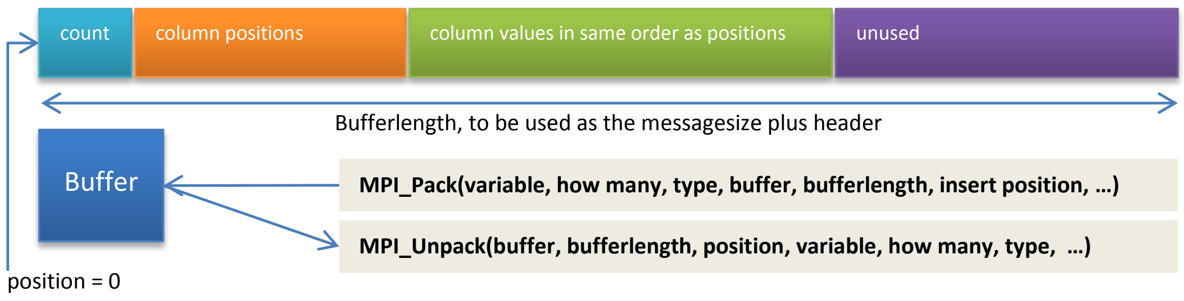
The assignment does not state that we have to use an optimized buffer, so I just use a buffer of 1024 bytes. This can be too less when we deal with a real big sparse matrix but in this case (and for testing purposes) it is definitely enough. Else one has to do the computation for the buffer with

$$B_{max} = i \cdot (1 + n) + d \cdot n,$$

where i is the size of an integer, d is the size of a double and n is the dimension of the matrix. The function calls for dealing with the data package is explained after the program output. For sending / receiving we use the datatype *MPI_Packed* to declare the package. Important is to always have the position variable right, i.e. setting it 0 when recv a new package. Else it is pretty straight forward.

Program output:

```
D:\Documents\Visual Studio 2005\Projects\Whatever\Debug>mpiexec -n 2
whatever -n 6
Original sparse matrix:
0.0     0.0     9.0     0.0     0.0     0.0
0.0     0.0     0.0     0.0     0.0     0.0
0.0     4.0     0.0     0.0     6.0     0.0
8.0     0.0     0.0     0.0     0.0     6.0
0.0     0.0     0.0     0.0     0.0     6.0
0.0     6.0     0.0     0.0     0.0     9.0
Transpose sparse matrix:
0.0     0.0     0.0     8.0     0.0     0.0
0.0     0.0     4.0     0.0     0.0     6.0
9.0     0.0     0.0     0.0     0.0     0.0
0.0     0.0     0.0     0.0     0.0     0.0
0.0     0.0     6.0     0.0     0.0     0.0
0.0     0.0     0.0     6.0     6.0     9.0
```


count
column positions
column values in same order as positions
unused
Bufferlength, to be used as the messagesize plus header
Buffer
MPI_Pack(variable, how many, type, buffer, bufferlength, insert position, …)
MPI_Unpack(buffer, bufferlength, position, variable, how many, type, …)
position = 0

Code printout:

```
#define BUFFERSIZE 1024
#define STANDARDCOUNT 5
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "mpi.h"

typedef struct
{
      int         count;              /* non-zero entries       */
      int*        colpos;             /* positions in the row */
      double*     data;               /* data values            */
} SPARSE_R;

void printMatrix(SPARSE_R* matrix, int dim);

void getColumn(SPARSE_R* matrix, int dim, int col, SPARSE_R* vec);

int main(int argc, char* argv[])
{
      int          my_rank;           /* rank of process      */
      int          p;                 /* number of processes */
      char         my_name[64];       /* name of machine      */
      int          my_name_len;       /* length of my_name    */
      MPI_Status   status;            /* return status (recv)*/
      int          n, k, i, j;        /* loop counters        */
      int          position = 0;      /* buffer position      */
```

```
	char		buffer[BUFFERSIZE];/* buffer content      */
	SPARSE_R*	matrix;		  /* sparse matrix array */
	/* dimension of the matrix */
	n = STANDARDCOUNT;
	/* Command line args parser */
	[...] - same as before / look at task 1.)
	/* Start up MPI */
	MPI_Init(&argc, &argv);
	/* Find out process rank  */
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
	/* Find out number of processes */
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	if(p < 2)
	{
		/* Shut down MPI if p < 2 */
		MPI_Finalize();
		/* and end program */
		return 0;
	}
	/* Get machine name */
	MPI_Get_processor_name( my_name, &my_name_len );
	/* create new sparse matrix - array of sparse rows */
	matrix = (SPARSE_R*)malloc(sizeof(SPARSE_R) * n);
	/* Generate new random sparse matrix */
	if(my_rank == 1)
	{
		srand(time(NULL));
		for(k = 0; k < n; k++)
		{
			/* create new sparse row */
			SPARSE_R tmp;
			/* get random entry count - max 50% occupied */
			tmp.count = rand() % (n/2);
			/* allocate the other pointers */
			tmp.colpos = (int*)malloc(sizeof(int)*tmp.count);
			tmp.data = (double*)malloc(sizeof(double)*tmp.count);
			/* spawn some values for number of values */
			for(i = 0; i < tmp.count; i++)
			{
				int colpos = 0;
				int ava = 0;

				/* check to not double occupy a column in row */
				do
				{
					ava = 0;
					colpos = rand() % n;
					for(j = 0; j < i; j++)
					{
						if(tmp.colpos[j] == colpos)
						{
							ava = 1;
							break;
						}
					}
				} while(ava == 1);
				/* set values - column position and value */
				tmp.colpos[i] = colpos;
				tmp.data[i] = (double)(rand() % 9 + 1);
			}
			/* set the k-th sparse row in the matrix */
```

```c
                  matrix[k] = tmp;
            }
            /* print out the generated matrix */
            printf("Original sparse matrix:\n");
            printMatrix(matrix, n);
            /* pack and send the matrix (columns) */
            for(i = 0; i < n; i++)
            {
                  /* allocate our sparse column - type sparse row */
                  SPARSE_R tmp;
                  position = 0;
                  /* get the i-th column from the rows */
                  getColumn(matrix, n, i, &tmp);
                  /* pack the data to the buffer */
                  MPI_Pack(&tmp.count, 1, MPI_INT, buffer, BUFFERSIZE,
                              &position, MPI_COMM_WORLD);
                  MPI_Pack(tmp.colpos, n, MPI_INT, buffer, BUFFERSIZE,
                              &position, MPI_COMM_WORLD);
                  MPI_Pack(tmp.data, n, MPI_DOUBLE, buffer, BUFFERSIZE,
                              &position, MPI_COMM_WORLD);
                  /* send the packed data */
                  MPI_Send(buffer, BUFFERSIZE, MPI_PACKED, 0, i,
                              MPI_COMM_WORLD);
            }
      }
      else if(my_rank == 0)
      {
            /* receive and unpack the matrix (rows) */
            for(i = 0; i < n; i++)
            {
                  /* allocate our sparse row - type sparse row */
                  SPARSE_R tmp;
                  tmp.colpos = (int*)malloc(sizeof(int)*n);
                  tmp.data = (double*)malloc(sizeof(double)*n);
                  position = 0;
                  /* receive the packed data */
                  MPI_Recv(buffer, BUFFERSIZE, MPI_PACKED, 1, i,
                              MPI_COMM_WORLD, &status);
                  /* unpack the data to the buffer */
                  MPI_Unpack(buffer, BUFFERSIZE, &position, &tmp.count, 1,
                              MPI_INT, MPI_COMM_WORLD);
                  MPI_Unpack(buffer, BUFFERSIZE, &position, tmp.colpos, n,
                              MPI_INT, MPI_COMM_WORLD);
                  MPI_Unpack(buffer, BUFFERSIZE, &position, tmp.data, n,
                              MPI_DOUBLE, MPI_COMM_WORLD);
                  /* set the i-th row */
                  matrix[i] = tmp;
            }
            /* print out new matrix */
            printf("Transpose sparse matrix:\n");
            printMatrix(matrix, n);
      }
      /* Shut down MPI */
      MPI_Finalize();
      return 1;
} /* main */

void printMatrix(SPARSE_R* matrix, int dim)
{
      int i, j, k;
      /* go through all rows */
      for(i = 0; i < dim; i++)
      {
            /* go thorugh all columns */
            for(j = 0; j < dim; j++)
            {
                  /* default value is zero for no column given */
                  double value = 0.0;
                  /* go through all column positions */
                  for(k = 0; k < matrix[i].count; k++)
                  {
                        /* if a column position == current column */
                        if(matrix[i].colpos[k] == j)
                        {
                              /* set the right cell value */
                              value = matrix[i].data[k];
                              break;
                        }
                  }
                  /* print the cell value */
                  printf("%2.1f\t", value);
            }
            printf("\n");
      }
      return;
} /* printMatrix */

void getColumn(SPARSE_R* matrix, int dim, int col, SPARSE_R* vec)
{
      int i, k;
      /* allocate arrays for temporary data storage */
      int* colpos = (int*)malloc(sizeof(int)*dim);
      double* values = (double*)malloc(sizeof(double)*dim);
      vec->count = 0;
      /* go through all rows */
      for(i = 0; i < dim; i++)
      {
            /* go through all columns with values */
            for(k = 0; k < matrix[i].count; k++)
                  /* if the column that has a value is equal to the
requested col */
                  if(matrix[i].colpos[k] == col)
                  {
                        /* save the rowindex i and the value of the cell */
                        colpos[vec->count] = i;
                        values[vec->count] = matrix[i].data[k];
                        /* increment counter */
                        vec->count++;
                        break;
                  }
      }
      /* format the data accordingly */
      vec->colpos = (int*)malloc(sizeof(int)*vec->count);
      vec->data = (double*)malloc(sizeof(double)*vec->count);
      for(i = 0; i < vec->count; i++)
      {
            vec->colpos[i] = colpos[i];
            vec->data[i] = values[i];
      }
} /* getColumn */
```