Parallel Sparse Matrix - Vector Product

Pure MPI and hybrid MPI-OpenMP implementation

Alexandersen, Joe; Lazarov, Boyan Stefanov; Dammann, Bernd

Publication date:
2012

Document Version
Publisher's PDF, also known as Version of record

IMM Technical Report 2012-10

Parallel Sparse Matrix - Vector Product

- Pure MPI and hybrid MPI-OpenMP implementation -

Authors: Joe Alexandersen

Boyan Lazarov
Bernd Dammann

Abstract:

This technical report contains a case study of a sparse matrix-vector product routine, implemented for parallel execution on a computer cluster with both pure MPI and hybrid MPI-OpenMP solutions. C++ classes for sparse data types were developed and the report shows how these classes can be used, as well as performance results for different cases, i.e. different types of sparse matrices.

DTU Informatics &
DTU Mechanics
1st of October 2012

Technical University of Denmark DTU

Preface

This report was written as part of the requirements for the DTU course '02616 Large Scale Modelling' with Boyan Lazarov and Bernd Dammann as supervisors.

The source code of the implementation shown in the report is available on-line at orbit.dtu.dk.

Contents

1 Introduction

1.1 Matrix-vector product

The matrix-vector product of linear algebra is one of the most important concepts in numerical modelling where linear systems are to be solved. When dealing with large systems of equations, iterative methods are most often used and these consist of repeatedly performing matrix-vector products to update the solution. Therefore, a fast implementation of the matrix-vector product is crucial if one seeks a fast and efficient iterative method.

1.2 Sparsity

A sparse matrix is where the number of non-zero entries is much smaller than the total number of entries. As most numerical discretisation schemes, e.g. finite difference, finite volume or finite elements, produce very sparse matrices, it can be extremely beneficial to exploit this sparsity. By storing only the non-zero entries, it is possible to reduce the required memory for storage significantly. Likewise, when dealing with parallel computations the amount of entries to be communicated between the processes is significantly smaller when dealing with sparse matrices. Although the mentioned discretisation schemes most often produce matrices with a certain structure, e.g. banded matrices, this project aims at an implementation that can work with general sparse matrices with no specific pattern.

1.2.1 Triplet format

Triplet format is the simplest way for storing general sparse matrices. For each non-zero value in the matrix, the column index, row index and value are stored in three arrays. This way of storing the non-zero entries is intuitively very easy to grasp, but the algorithms needed to perform, for example, a matrix-vector product are often slower than for other more advanced formats.

1.2.2 Compressed Sparse Row [CSR] format

The Compressed Sparse Row [CSR] format is a condensed and economic way to store a general sparse matrix. Instead of storing the non-zero entries and their corresponding row and column index, the CSR format consists of three arrays:

- `vals`: An array of doubles[1] containing the non-zero entries of the sparse matrix. The length of the array is equal to the number of non-zeroes.

[1] It could consist of other datatypes, e.g. floats or complex, but the current implementation is restricted to doubles only.

- `cols`: An array of integers containing the column index of the non-zero entries in the array of doubles. The length of the array is equal to the number of non-zeroes.
- `rows`: An array of integers containing the index, in the other two arrays, of the first non-zero entry for each row. The length of the array is equal to the number of rows plus one. The last entry of the array contains the number of non-zero entries, to facilitate an easy setup of algorithms.

To illustrate, the following sparse matrix:

$$\begin{bmatrix} 0.0 & 65.3 & 46.3 & 0.0 & 8.0 & 0.0 & 40.3 & 0.0 & 0.0 & 0.0 \\ 19.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -99.2 & 0.0 & 0.0 & 0.0 \\ -40.3 & 49.0 & 0.0 & 0.0 & 0.0 & 0.0 & 18.4 & 0.0 & 0.0 & 0.0 \\ 58.2 & 0.0 & 0.0 & 30.0.1 & 0.0 & -63.3 & 0.0 & 0.0 & 0.0 & -75.3 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 93.3 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -73.5 & 0.0 & 0.0 & 0.0 & 0.0 & -61.1 & 0.0 & 0.0 \\ 0.0 & 96.6 & 85.9 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -13.0 & 68.7 & 0.0 & 16.6 & 0.0 & 0.0 & -95.7 & -63.5 & -9.5 \\ 0.0 & 0.0 & 0.0 & -84.7 & 52.1 & 0.0 & 0.0 & -25.1 & -82.1 & 0.0 \\ 0.0 & 0.0 & -76.7 & 0.0 & -46.1 & 0.0 & 0.0 & 0.0 & 0.0 & -57.1 \end{bmatrix}$$

would have the following CSR array representation:

```
vals = {65.3,46.3,8.0,40.3,19.5,-99.2,-40.3,49.0,18.4,58.2,30.1,-
63.3,-75.3,93.3,-73.5,-61.1,96.6,85.9,-13.0,68.7,16.6,-95.7,-63.5,-
9.5,-84.7,52.1,-25.1,-82.1,-76.7,-46.1,-57.15}
cols = {1,2,4,6,0,6,0,1,6,0,3,5,9,6,2,7,1,2,1,2,4,7,8,9,3,4,7,8,2,4,9}
rows = {0,4,6,9,13,14,16,18,24,28,31}
```

The original full matrix contains 100 doubles, so the memory needed to store the full matrix is 800 bytes. By using the CSR format, only 31 doubles need to be stored. Although the two integer arrays also need to be stored, 42 entries in total, the total memory needed to store the matrix in CSR format is 416 bytes. That is a reduction of almost 50% even for a small 10 by 10 matrix and it is easy to imagine that the amount of data to store is significantly reduced when dealing with very large and very sparse matrices.

Another advantage of the CSR format is that it is very efficient for algorithms for arithmetic operations and matrix-vector products. It is important to note that better formats exist, especially for clustered data, but due to the implementation being aimed at general sparse matrices with no specific pattern, then the CSR format is used.

2 Implementation

The implementation has been carried out in the C++ programming language and is split up into several parts, all of which will be explained in this section. The code of the implementation can be found in appendix A.

The parallel vector and matrix representations are coded as objects and each have a number of subfunctions that help to initialise, control and manipulate the distributed data. The structure of the objects and the underlyings functions will be explained in each their section. Figure 1 illustrates how the

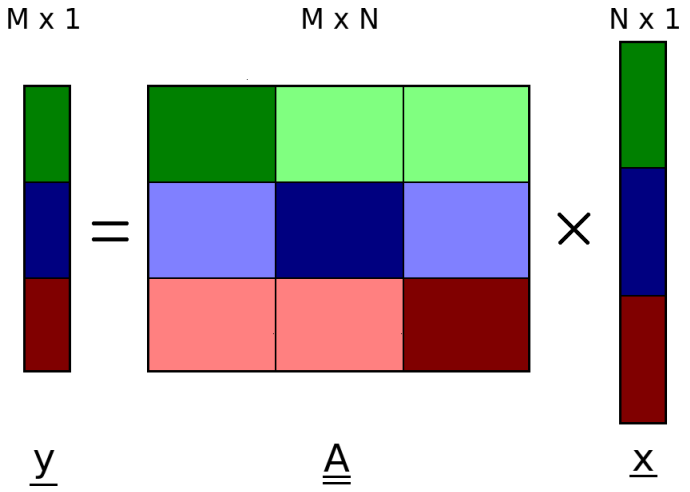


Figure 1 – Illustration of how the parallel matrix and vector representations are distributed over three processes.

parallel matrix $\underline{\underline{A}}$ and vectors $\underline{x}$ and $\underline{y}$ are distributed over three processes. The three colours; green, blue and red, represent each their process and which data is local to each individual process. The lightly coloured areas of the matrix is to illustrate which parts of the distributed matrix has be be multiplied by remote vector entries, these will be termed the local-to-global areas, whereas the fully coloured areas of the matrix are to be multiplied with the vector entries local to the process itself, these will be termed the local-to-local areas. Both the matrix and the vectors are partitioned by rows because then each process only has to pass results from the product to its own local part of vector $\underline{y}$. Furthermore, this partitioning allows for communications to be overlapped with computations, where the remote vector entries are being communicated while the local computations are performed. If the matrix was partitioned by columns instead, all communications would have to be at the end when the locally computed product contributions have to be sent to remote vector partitions.

2.1 Parallel vector

The parallel vector representation is composed of two main classes: `ParVectorMap` and `ParVector`. `ParVectorMap` controls the partioning of the parallel vectors, and matrices, and provides information about the distribution of the vector data. `ParVector` is the parallel vector object itself, which contains the actual distributed data and functions controlling the local data.

2.1.1 `ParVectorMap`: Vector index map

The `ParVectorMap` class is a vector index map that controls the partioning and distribution over the processes. The constructor header is as follows:

```
ParVectorMap(MPI_Comm ncomm, int lbound, int ubound);
```

Each process gets a lower bound and upper bound value attached to it, that describes the global index numbers of the beginning and end of the current process' local array. When a vector map is to be created, all processes in the specified MPI communicator have to call the constructor function. This is because in the constructor function, the `MPI_Allgather` function is used to share the lower and upper bounds of all other processes partaking in the distribution of the vector:

```
MPI_Allgather(&lower_bound,1,MPI_INT,lprocbound_map,1,MPI_INT,comm);
MPI_Allgather(&upper_bound,1,MPI_INT,uprocbound_map,1,MPI_INT,comm);
```

In the above function calls the local integers, `lower_bound` and `upper_bound`, are sent to all other processes in the communicator and all integers from the local and remote processes are collected in the local arrays of integers, `lprocbound_map` and `uprocbound_map`. The `MPI_Allgather` function is used because *all* of the processes need to have the global picture of the distributed vector and therefore collective communications are obvious to use. The class contains a series of getter functions that return various private data for the index map, such as the size of the local array, the size of the global array or the owner of a given global index. Furthermore, the index map also has functions to convert global indices to local indices and vice versa, which are useful because it is often easiest to work in local indices when performing local operations, but the global indices are needed when communicating with the global picture.

Each `ParVectorMap` object contains an integer value called `users` which is the number of active users of the vector map at any given time. The amount of users is updated every time a new object, e.g. an `ParVector` or `ParMatrixSparse` object, is associated with the given `ParVectorMap` index map object. This is to ensure, that a `ParVectorMap` is not deleted without having been released by all other objects using it.

2.1.2 `ParVector`: Vector class

The `ParVector` class is the representation of the distributed vector itself and utilises the vector index map class, described above, to control the distribution of the data. When a distributed vector is to be created, the `ParVector` constructor function is called by all processes in a given communicator for which the vector should be distributed over. This does not necessarily have to be the global MPI communicator, `MPI_COMM_WORLD`, but it is so throughout this project.

```
ParVectorMap(MPI_Comm ncomm, int lbound, int ubound);
```

Each local process calls the constructor with their specified lower and upper bound index and the first thing that is set up is the `ParVectorMap` objects that, as explained, control the distribution of the vector data. Each `ParVector` object has a local array, using local indices, attached to it in which the local part of the distributed vector is stored. As well as containing various simple getter functions, that return things such as the process index map object, the local array or bounds, the `ParVector` class also has some adding and setter functions. The setter functions are used if all entries are to be set to a given value, for instance all zeroes. The adder functions:

```
void AddValueLocal(int row, double value);
void AddValuesLocal(int nindex, int *rows, double* values);
```

allows single values or an array of values to be added to the local array by specifying the row indices and the corresponding values.

When dealing with numerical discretisation schemes, the vector entries would be added during the assembly of the system. However, for testing purposes, vector data is read in from an external input file, `vector.vec`, which contains the dense vector entries in triplet format.

```
void ReadExtVec();
```

The external input reader simply runs through the input file and adds any values that are within the local process bounds to the local array of data, using:

```
if ( (val1 >= lower_bound) && (val1 < upper_bound ) ) {
    AddValueLocal(index_map->Glob2Loc(val1),val2);
}
```

where `val1` is the row index and `val2` is the entry value. Notice that the index map is used to convert the global index read from the input file to a local index for inserting the value into the local array.

2.2 Parallel sparse matrix

The parallel matrix representation is composed of a single class: `ParMatrixSparse`. `ParMatrixSparse` contains the distributed data and functions to manipulate, control and perform the matrix-vector product. `ParMatrixSparse` objects utilise `MatrixCSR` structs to manage the sparse matrix information in CSR format.

2.2.1 `MatrixCSR`: CSR struct

The `MatrixCSR` struct is a simple structured type that contains the values and arrays necessary to store matrix entries in CSR format:

```
int nnz;
int* rows;
int* cols;
double* vals;
```

When a `MatrixCSR` object is to be created, it requires the number of non-zero entries and the number of rows as input. The constructor then directs the pointers to arrays that can contain the data as specified in section 1.2.2:

```
rows = new int[nrows_in+1];
cols = new int[nnz];
vals = new double[nnz];
```

The `MatrixCSR` struct is used as a simple way to contain the CSR formatted data, making it easier to send or return as output if needed by the user.

2.2.2 `ParMatrixSparse`: Matrix class

The `ParMatrixSparse` class is the representation of the distributed sparse matrix itself and utilises two vector index map objects to control the partitioning and distribution of data. The reason for utilising two seperate index maps for the row and columns, is so as to be able to accomodate non-square matrices that may arise in for instance coordinate transformation operations. The constructor:

```
ParMatrixSparse(ParVector* XVec, ParVector* YVec);
```

takes two `ParVector` objects as input so that the vector maps can be extracted from these and used to set up the partitioning of the parallel matrix. Another option could also be to make an overloaded constructor that takes two `ParVectorMap` objects as input, if only the parallel matrix is needed, but for the current problem at hand, two `ParVector` objects will already exist as the left- and right-hand side vectors of the matrix-vector product.

The `ParMatrixSparse` class contains many of the same functions as the `ParVector` class, such as the simple getters for bounds, but the essence of the parallel matrix implementation lies in the extended functionality. While creating and manipulating the data within the parallel sparse matrix objects, a dynamic matrix map is used. The dynamic matrix maps are objects of the `map` class from the C++ Standard Template Library [STL] and the smart thing about these are that they allow for dynamically growing storage, which is very useful while generating or assemblying the parallel matrix data.

```
std::map<int, double> *dynmat_lloc, *dynmat_gloc;
```

The data to be stored in the `ParMatrixSparse` object local to the process is split into two parts. The first part is the local-to-local part, where when performing the matrix-vector product only local data from both the matrix and the vector are needed, and the local-to-global part, where vector entries from remote processes are needed for performing the matrix-vector product. These two parts are illustrated in figure 1, where the fully coloured areas represent the local-to-local parts for each process and the lightly coloured areas represent the local-to-global parts. This splitting of the local domain is also used for the CSR representation of the local matrix and is done to easily facilitate the splitting of the matrix-vector product into two parts; the local-to-local product which is performed while the communications for the local-to-global part is taking place.

As for the `ParVector` class, the matrix entries would be added to the matrix during the assembly of the numerical discretisation, but for testing purposes the sparse matrices are read in from an external input file, `matrix.mat`. The `ReadExtMat` function works in much the same way as the vector equivalent, in that it checks whether the entries are within the local data range and then adds them to the local matrix representation. The `AddValueLocal` function is somewhat different to the vector equivalent, in that it checks whether the entries are within the local-to-local part or the local-to-global part and adds it to the respective dynamic matrix map using iterators:

```
std::map<int,double>::iterator it;
it = dynmat[row].find(col);
if ( it == dynmat[row].end() ){
    dynmat[row][col] = value;
    nnz++;
} else {
    it->second = it->second + value;
}
```

where general naming has been used. If the entry is within the local-to-local part then `dynmat_lloc` is used and `dynmat_gloc` if the entry is in the local-to-global part. After the parallel sparse matrix has been read in from

the input file and the matrix is finalised, the `ParMatrixSparse` class has a function to convert the dynamic matrix representation to the specified CSR format, which is needed by the implemented algorithm for performing the sparse matrix-vector product.

The `ParMatrixSparse` class also contains the functions for setting up the communications for the matrix-vector product along with the product routine itself and these will be explained in the following subsections.

2.3 Organisation of communications

The organisation of communications between the processes is most important, as having a fast local routine would not be of much use if the communications are not fast enough to keep up. The communications are orgranised in such a way that only a minimum number of vector entries need to be sent between the participating processes. This is achieved by sorting through the sparsity pattern of the local-to-global part of the local sparse matrix and exploiting the sparsity pattern only to send the required entries using a derived MPI datatype, namely `MPI_Type_indexed`.

2.3.1 Exchanging message sizes and indices

The `ParMatrixSparse` class contains a function called `FindColsToRecv`, which basically sorts through the local-to-global sparsity pattern and notes how many and which entries to receive from each process. This is done through the use of integer-to-integer `map` objects because these allow for easily finding existing values and adding new values, so that it is possible to sort the column indices of the sparsity pattern in such a way that repeating indices are only counted, and thus transferred, once.

Sends and receives for the number of entries to send and receive from each process are then posted, so that every process partaking in the matrix-vector product knows how many entries to send to and receive from each other remote process. When it is known how many entries that need to be received from each remote process, the indices of the entries to be transferred are loaded into a buffer and sent to each of the remote processes that need to transfer vector entries back to the local process. Meanwhile, the local process is also receiving which entries to send to each of the remote processes. When the non-blocking sends and receives are done, all processes now know which entries to send to which process for the distributed matrix-vector product to be performed.

2.3.2 Derived datatype: `MPI_Type_indexed`

In order only to transfer the required vector entries from one process to another, the derived MPI datatype `MPI_Type_indexed` is used. The MPI datatype `MPI_Type_indexed` is an easy way to communicate data that is not

stored in contiguous memory locations, e.g. as in the sparse matrix-vector product where only a select number of arbitrarily spaced entries of an array are to be sent from one process to another. `MPI_Type_indexed` allows for replication of an existing datatype, in this case `MPI_DOUBLE`, into a sequence of blocks of the old datatype with different displacements, all of which are specified in multiples of the old datatype [1]. The datatypes are tailored for each process-to-process transfer and they are set up within a loop over all remote processes. The call to set up the derived datatype for a given process-to-process transfer is quite simple:

```
MPI_Type_indexed(count,blength,displac,MPI_DOUBLE,&DTypeSend[i]);
```

where the function arguments are as follows:

- `count`: the number of entries to send to process `i`.
- `blength`: a pointer to an array of length `count` specifying the number of `MPI_DOUBLE`s contained in each block - in this case, an array of ones.
- `displac`: a pointer to an array of length `count` specifying the displacement, in number of `MPI_DOUBLE`s, of each block - in this case, simply the indices, in local numbering, of the entries to send to process `i`.
- `MPI_DOUBLE`: the MPI datatype which everything is specified in.
- `&DTypeSend[i]`: a pointer to where the new MPI datatype is to be stored - in this case, the datatypes for each process-to-process transfer is stored in an array of datatypes, so the MPI datatype for sending to process `i` is stored in array element `i`.

The above function call is to set up the datatype for 'current process'-to-'process number `i`' sending of entry values. The corresponding datatype for 'process number `i`'-to-'current process' receiving of entry values is very similar, except that the array of displacements, `displac`, instead are in global numbering. This is because the received entries, from all remote processes, are to be inserted into an array of global vector length, where the entries from each remote process are placed at the corresponding indices. This is to facilitate an easy setup of the sparse CSR matrix-vector product algorithm as explained in the next subsection.

2.4 Matrix-vector product

As already explained, the parallel sparse matrix-vector product is contained as a subfunction of the `ParMatrixSparse` class. The implementation utilises non-blocking and overlapping communications with the abovementioned derived MPI datatype. This was chosen as the initial implementation because

then the local-to-local product can be done while the transfers of vector entries from remote processes are being performed.

The algorithm for performing the sparse matrix-vector product using CSR formatted sparse matrix data and a dense vector is as follows using general naming:

```
for (i = 0; i < nrows; i++){
    for(k = CSR->rows[i]; k < CSR->rows[i+1]; k++){
        j = x_index_map->Glob2Loc(CSR->cols[k]);
        v = CSR->vals[k]*vecdat[j];
        YVec->AddValueLocal(i,v);
    }
}
```

where depending on whether it is the local-to-local or local-to-global product contribution to be calculated, `CSR` is either the local-to-local or local-to-global `MatrixCSR` object and `vecdat` is either the local vector array, `XVec->GetArray()`, or the received remote vector entries, `rBuf`.

2.4.1 Non-blocking communications

The workflow of the matrix-vector product routine using non-blocking communications is as follows:

1. Post all sends and receives using non-blocking communications.
2. Calculate the local-to-local contribution to the product, using the local-to-local CSR object and the local right-hand side vector, and add to the local result vector.
3. Wait for receives of remote vector entries to complete.
4. Calculate the local-to-global contribution to the product, using the local-to-global CSR object and the received right-hand side vector entries, and add to the local result vector.

As already mentioned, non-blocking communications are used because then the local-to-local product can be done while the transfers of vector entries from remote processes is being performed. Whether this has any significant effect, as opposed to blocking communications, has not been investigated.

2.5 MPI-OpenMP Hybrid

In this section, a hybrid implementation, where MPI is mixed with OpenMP, will be described. MPI is used to handle the coarse grain parallelism, that is the partitioning of the parallel matrices and vectors, and OpenMP is used to further split the work involved in the matrix-vector product into smaller grains. On a cluster, the configuration of a run can be tailored such

that MPI handles the communication between nodes and OpenMP is used locally inside each SMP node. The combination of MPI and OpenMP can be powerful if handled with care and implemented properly. The only part of the original implementation that has been considered is the matrix-vector product itself, in that this is the critical routine and the subject of this project.

In order to mix MPI with OpenMP, MPI needs to be initialised using the thread supporting version of `MPI_Init`, namely `MPI_Init_thread`. `MPI_Init_thread` initialises MPI in much the same way that `MPI_Init` does, but it also initialises the thread environment and sets up the required level of thread support. The required level of thread support in the hybrid implementation is specified as `MPI_THREAD_FUNNELED`, which means that the process can be multithreaded but the programmer has to make sure that only the master thread makes MPI calls. This will be returned to at a later stage.

The hybrid implementation of the matrix-vector product can be seen below:

```
#pragma omp parallel default(shared) private(i,j,k,v){
   // Start omp parallel region
   // Calculate local-local product
   if (CSR_lloc != NULL) {
        #pragma omp for schedule(guided)
        for (i = 0; i < nrows; i++){
           for(k = CSR_lloc->rows[i]; k < CSR_lloc->rows[i+1]; k++){
                j = x_index_map->Glob2Loc(CSR_lloc->cols[k]);
                v = CSR_lloc->vals[k]*sBuf[j];
                YVec->AddValueLocal(i,v);
           }
        }
   }
   // Wait for receiving of ghostvalues to complete
   #pragma omp barrier
   #pragma omp master
   {
        MPI_Waitall(nProcs-1,Rreqs,Rstat);
   }
   #pragma omp barrier
   // Calculate local-global product
   if (CSR_gloc != NULL) {
        #pragma omp for schedule(guided)
        for (i = 0; i < nrows; i++){
           for(k = CSR_gloc->rows[i]; k < CSR_gloc->rows[i+1]; k++){
                j = CSR_gloc->cols[k];
                v = CSR_gloc->vals[k]*rBuf[j];
                YVec->AddValueLocal(i,v);
           }
        }
   }
} // End of omp parallel region
```

The main workload, in the matrix-vector product routine, consists of the two sets of nested loops over the matrix entries; the first calculates the

local-to-local contribution to the product and the second the local-to-global contribution. Therefore, the parallel region is defined as covering these two sets of nested loops. The `default(shared)` clause is used to give all variables a default shared attribute. This is done so as to avoid having to explicitly define the data-sharing attribute for all variables, instead only those where it is critical, that they be private to each thread, need to be specified. This is done so using the `private` clause, which is used to specify that each thread needs to have a unique and local instance of the variables used to count through the matrix and vector entries.

The outer loops of the two sets of nested loops are chosen to be parallelised using OpenMP. This results in a partitioning of the matrix-vector product by rows and is done in order to assure that the threads do not attempt to write to the same location of the results vector, `YVec`, at the same time. By partitioning the product by rows, each thread handles a different part of the local result vector and simultaneous data writing is avoided. Furthermore, none of the instructions inside the loops depend on previous values of the result vector and therefore this can be done safely.

After the calculation of the local-to-local product contribution, there is a required call to the `MPI_Waitall` function in order to ensure that the receiving of the remote vector entries is finished, before beginning the calculation of the local-to-global product contribution. It is important that this call is only performed by one thread per process, here the main thread, and one approach to do this could be to simply have defined seperate parallel regions for the first and second product contributions. However, this would lead to unneccessary overhead due to the forking-and-joining of threads having to be done twice. Therefore, the parallel region is defined as covering both product contributions and it is ensured that only the master thread performs the `MPI_Waitall` call. This is ensured using the `master` construct and this is encapsulated in `barrier` constructs due to the lack of an implied barrier, both at the entry to and exit from the `master` construct.

The `schedule` clause is used to control the scheduling of the parallel loops, that is to control how the iterations are distributed over the threads. The initial thought was to use `static` scheduling, which divides the iterations over the threads in equal, or almost equal, sized chunks and this gives the `static` scheduling the least overhead [6]. However, by experimenting, the author has found that the `guided` scheduling performs the best for the used matrices and therefore this is used. The `guided` scheduling assigns iterations to threads as they are requested and the threads are assigned a new set of iterations when the first are done. The size of the assigned chunks decrease proportionally to the number of unassigned iterations. The reason as to why this scheduling type performs very well, may be the fact that the workloads are relatively poorly balanced and unpredictable for the current

matrices used in the investigation. When the banded matrices[2] are partioned using MPI, this can result in local-to-local and local-to-global parts of the local matrices where the number of entries per row is strongly dependent on the row number, as can be seen in figure 2. Likewise the completely random matrices[2], as the name suggests, are randomly distributed and the workload therefore is also randomly distributed between the processes *and* the threads.

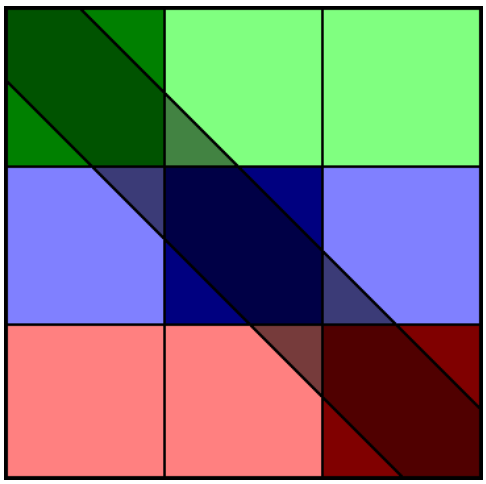


Figure 2 – Illustration of how the banded matrices are distributed over three processes and how the entries per row is strongly row-dependent for the global-to-local parts.

[2]The matrices are described in section 3.1.

3 Results

3.1 Generating input

To test the implementation, some sample matrices and vectors are generated using a MATLAB-script. Three types of matrices are generated randomly for testing the implementation. Firstly, diagonally banded matrices are generated, where the random entries are concentrated around the main diagonal of the matrix within a certain spread. Two subtypes of the banded matrices are generated; ones containing a maximum of 10 entries per row, which is roughly equivalent to a two-dimensional numerical discretisation, and ones containing a maximum of 100 entries per row, which is roughly equivalent to a three-dimensional numerical discretisation. Secondly, tridiagonally banded matrices are generated, where two additional diagonal bands are displaced a certain distance from the main diagonal. These contain a maximum of 20 entries per row. Thirdly, randomly distributed matrices are generated, where the random entries are spreadout over the entire matrix. The random matrices are generated containing a maximum of 10 or 100 entries per row.

For each matrix, a matching dense vector is generated with random values. The vectors and matrices are exported to a standard textfile in triplet-format using a modified MatrixMarket output script [2]. The MATLAB-scripts can be found in appendix B. Information on the matrices used for testing the implementation is tabulated in tables 1, 2 and 3 and figure 3 shows the sparsity pattern of two of the generated matrices, a banded matrix and a tri-banded matrix.

Type	Size	Non-zeroes
Banded “2D”	40k × 40k	394550
Banded “2D”	160k × 160k	1581304
Banded “2D”	320k × 320k	3159071
Tri-banded	160k × 160k	3089302
Tri-banded	320k × 320k	6216366

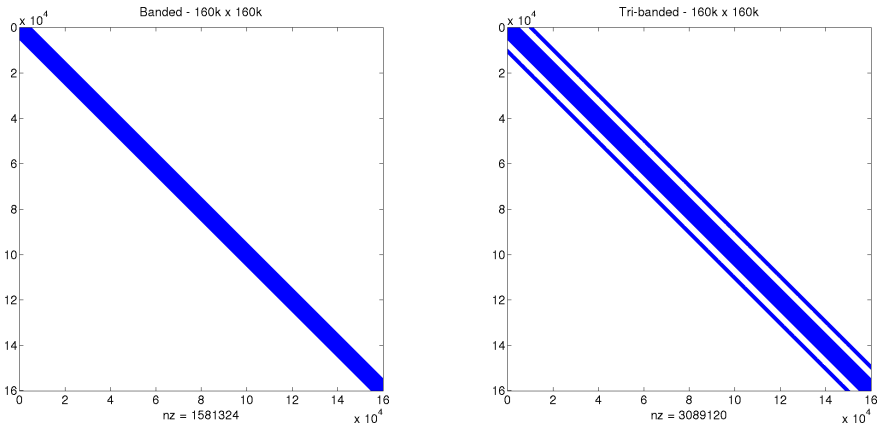
Table 1 – Information about the initial matrices used for testing the implementation.

Type	Size	Non-zeroes
Banded “2D”	160k × 160k	1581304
Banded “2D”	320k × 320k	3159071
Banded “3D”	160k × 160k	14652683
Banded “3D”	320k × 320k	28824180

Table 2 – Information about the banded matrices used for testing the implementation.

Type	Size	Non-zeroes
Random 10	160k × 160k	1599961
Random 10	320k × 320k	3199959
Random 100	160k × 160k	15994925
Random 100	320k × 320k	31995059

Table 3 – Information about the random matrices used for testing the implementation.



(a) Banded 160k × 160k **(b)** Tri-banded 160k × 160k

Figure 3 – Sparsity patterns for two sample matrices.

3.2 Timing runs

3.2.1 TopOpt Cluster

The timing runs were performed on the TopOpt cluster at the Section for Solid Mechanics at the Department of Mechanical Engineering. The cluster consists of 21 HP SL390s nodes, each equipped with two 6 core Intel Xeon X5660 CPUs (2.80 GHz) and 48GB RAM connected with QDR Infiniband. Access to the cluster was secured through the students affiliation with the TopOpt research group.

For each timing run, the parallel matrix-vector product is performed a total of 5000 times, to average out load fluctuations of the cluster, and the total walltime is saved and used to investigate the scalability of the implementation.

3.2.2 Pure MPI

Figure 4 shows the speed-up as a function of processors for the matrices listed in table 1. The speed-up is measured relative to the time taken for a single processor run and the timing runs are compared to linear scalability.

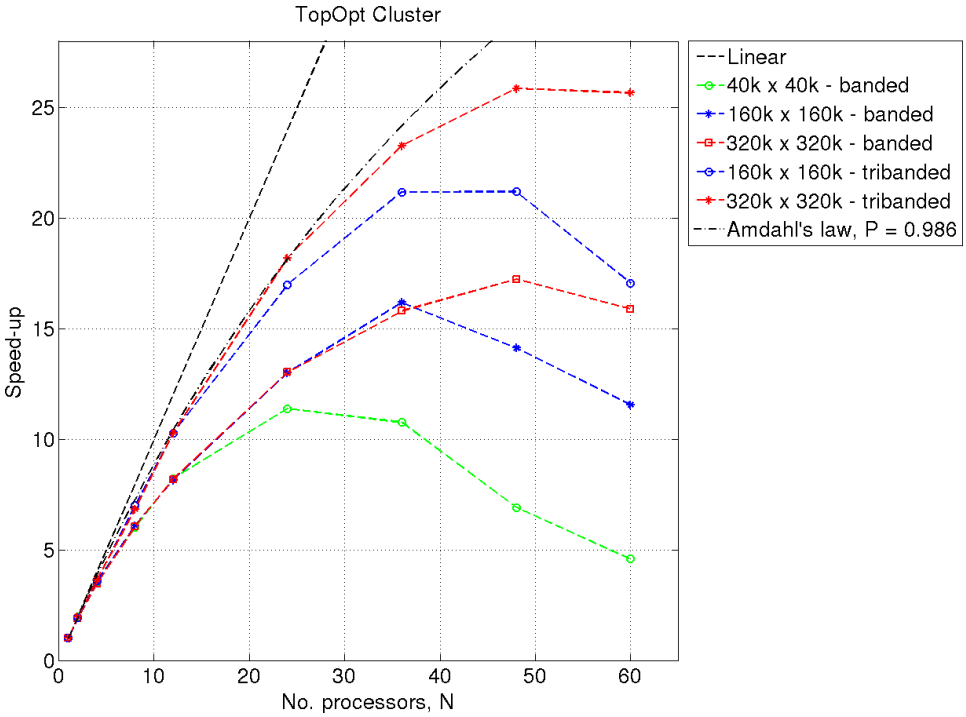


Figure 4 – Scalability plot for the matrices listed in tables 1.

It can be seen that the current implementation scales quite nicely but still has a lot of room for improvement. A trend can be observed, that as the problem size is increased for the banded matrices, the point at which the speed-up peaks is increased also. This makes sense as the amount of data to be treated is also getting larger and therefore more processes can be utilised before the increasing amount of communications begin to dominate the execution time.

The same trend can be said to be true for the tri-banded matrices and these both show better speed-up than for the banded matrices. The reason for the improved speed-up when working with tri-banded matrices, could be due to the increased amount of non-zeroes, which fits with the explanation posed for the banded matrices. It can also be postulated that combined with this, the fact that the extra bands are displaced from the main diagonal can explain the increased speed-up, as the workload thereby is balanced better than for the banded case. The banded 320k $\times$ 320k matrix has almost the same amount of non-zeroes as the tri-banded 160k $\times$ 160k matrix, but the later shows much larger speed-up. This could very well be due to the load balancing being improved, as just explained.

From figure 4 it can also be seen that the speed-up for the banded and tri-banded matrices, follow the others of the same type up until 12 processors. This makes sense as this is the point where inter-node communication is

beginning, whereas up until 12 processors everything has been contained within the same node. However, it is interesting to note that the 160k $\times$ 160k and 320k $\times$ 320k banded matrices actually follow each other up to 36 processors.

If one takes the 320k $\times$ 320k tri-banded matrix to be the limiting case for the current timing runs, it can be postulated using Amdahl's law that the parallelised portion of the program amounts to roughly 98.6%. This is of course a very rough estimate, seeing as it is impossible to know whether the 320k $\times$ 320k tri-banded matrix is the limiting case for the implementation.

Type	Size	Optimal N	Optimal speed-up
Banded	40k $\times$ 40k	24	11.38
Banded	160k $\times$ 160k	36	16.19
Banded	320k $\times$ 320k	48	17.24
Tri-banded	160k $\times$ 160k	48	21.19
Tri-banded	320k $\times$ 320k	48	25.86

Table 4 – Optimal number of processes and corresponding speed-up for the matrices used for testing the implementation.

Table 4 shows the optimal number of processors, and the corresponding speed-up, for the different matrices. It can be seen that for the larger problems, it appears that the optimal number of processors is approximately 48. Of course this number is found from the discrete points from the timing runs, so the optimal could theoretically be slightly lower or slightly higher, but this would not utilise all CPUs on a given number of nodes.

Figure 5 shows the speed-up as a function of processors for the matrices listed in table 2. The speed-up is measured relative to the time taken for a single processor run and the timing runs are compared to linear scalability. It can be seen that the "3D" matrices scale much better than the "2D" equivalents and close to linearly up to around 24-36 processors. This makes sense as the "3D" matrices contain more entries and therefore lead to more computations, in turn leaving more time for communications. Furthermore, the entries are distributed in a wider band around the diagonal than for the "2D" equivalents, so the workload is distributed over more processes.

Figure 6 shows the speed-up as a function of processors for the matrices listed in table 3. As for the banded matrices, it can be seen that the matrices with random-100 matrices scale much better than the random-10 matrices and close to linearly up to around 24-36 processors. It is interesting to see that the scalability of the random-100 matrices exhibits a form of snap after which the speed-up goes from being linear to become more or less constant. The sudden peak at $N = 84$ for the 160k $\times$ 160k random-100 matrix is also interesting and can possibly be explained to be an effect of a particularly favourable partitioning of the matrix. Given that the matrix entries are

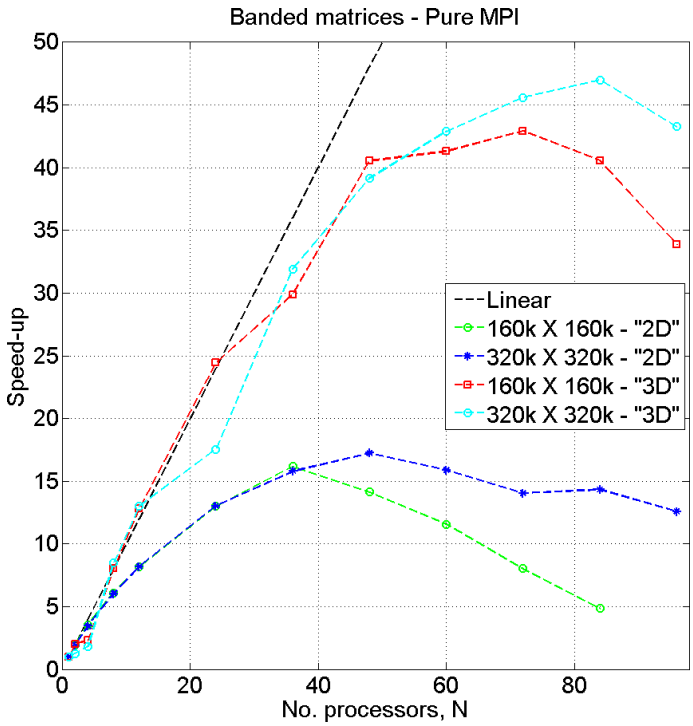


Figure 5 – Scalability plot for the matrices listed in table 2.

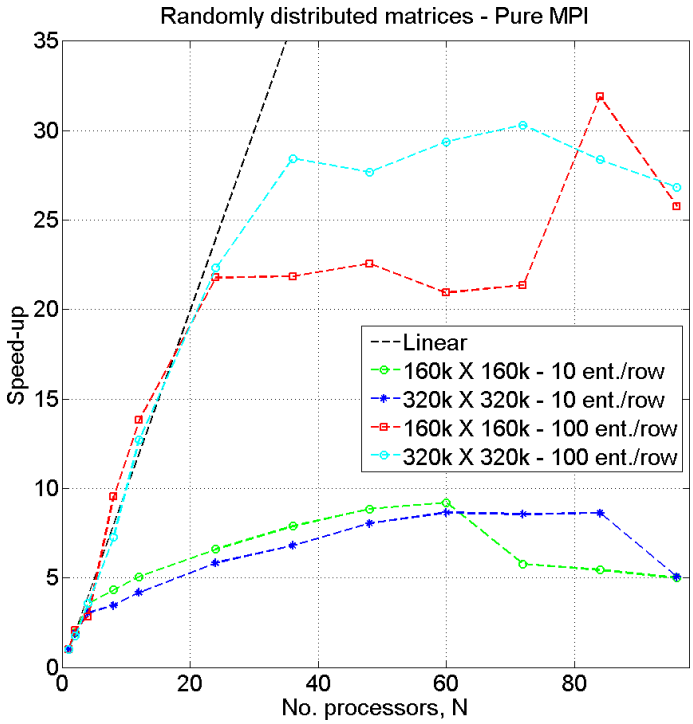


Figure 6 – Scalability plot for the matrices listed in table 3.

completely randomly distributed, particular sections of the partitioned matrix may be completely free of entries and therefore not be partaking in the communications.

When comparing figures 5 and 6, it is easily seen that the maximum speed-up for the random matrices is significantly lower than for the banded matrices. This makes sense, as the entries of the random matrices are spread over the entire matrix and therefore require the local processes to communicate with many, if not all, other processes to receive the needed remote vector entries to perform the local-to-global product.

3.2.3 Hybrid MPI-OpenMP

In this section, the hybrid implementation is compared to the original implementation. Four different configurations of utilising the processors are used, where the node to process and thread to process ratios are varied, and these are tabulated in table 5. All of the configurations utilise a total

Name	Processes/node	Threads/process
1×12	1	12
2×6	2	6
2×12 (overloaded)	2	12
4×3	4	3

Table 5 – Information about the random matrices used for testing the implementation.

number of threads per node according to the amount of processors of each node (12), except for the third configuration, 2×12, which is an attempt at checking whether overloading the processors of each node is beneficial. The scalability of the pure MPI implementation will be included in the following plots as a reference.

Figure 7 shows the scalability achieved for the 160k $\times$ 160k "3D" banded matrix using the hybrid implementation. It can be seen that all of the configurations, aswell as the pure MPI run, perform similarly and close to linear up until approximately 48 processors, except for the overloaded and 1×12 configurations. The 1×12 configuration is clearly inferior compared to all of the others and this makes sense. With this configuration, the matrix and vectors are distributed over the nodes with one partition per node, therefore everytime that information needs to be transfered between processes it is done over the network. This is of course slower as compared to the local transfer speed obtainable between processes on the same nodes when using configurations with several processes on the same node. Furthermore, when only one process is allocated per node, not all worker threads will be allocated to the same socket as the process. Therefore, some threads will

most likely have to access memory that is allocated to the other socket of the node, thus increasing the time taken to access the memory. The optimal configuration seems to be 2×6, which continues to scale after the other configurations have started to bend off, up until 108 processors yielding a speed-up of approximately 80.

The sudden drop in speed-up of the 2×6, 2×12 and 4×3 configurations at $N = 84$ is rather strange. Several runs, using different combinations of nodes, have been performed and yielded the same results and therefore currently no explanation can be offered for this anomaly.

Figure 8 shows the scalability achieved for the 320k $\times$ 320k "3D" banded matrix using the hybrid implementation. It can be seen that the pure MPI and 4×3 configurations perform similarly and better than the rest, up until $N = 48$ where they are overtaken by the 2×6 configuration. It is interesting to observe that the 4×3 configuration overtakes the 2×6 configuration for $96 < N \leq 120$. Therefore, the 4×3 configuration yields the optimal speed-up this time, of approximately 85 at 120 processors.

Again, as of yet no explanation can be offered for the sudden drop of the 2×6, 2×12 and 4×3 configurations around $N = 72 \pm 12$.

Figure 9 shows the scalability achieved for the 160k $\times$ 160k random-100 matrix using the hybrid implementation. It can be seen that the 2×6 configuration yields super-linear speed-up until $N = 36$ whereafter it starts to bend off. The 4×3 configuration also performs very well, yielding close to linear scalability up until $N = 48$. As always the 1×12 configurations performs inferior to the other configurations, but for this matrix it actually overtakes the pure MPI run in the interval of $48 \leq N \leq 96$. The configurations utilising 2 processes per node exhibit a lowered speed-up for $N = 48$ and $N = 60$, which again seems strange and the results remain the same for several different runs. However, due to the fact that both configurations utilises the same partitioning, that is 2 processes per node, it can be stipulated that the drop is due to a particularly bad partitioning of the distributed matrix resulting in increased communications and a poor distribution of workload.

Figure 10 shows the scalability achieved for the 320k $\times$ 320k random-100 matrix using the hybrid implementation. It can be seen that the 2×6 configuration yields super-linear speed-up, this time remaining above linear at least until $N = 96$, except for a drop at $N = 48$ and $N = 60$. It is interesting, and strange, to note that the scalability curves exhibit very similar behaviour for both of the random matrices. The configurations utilising 2 processes per node exhibit a lowered speed-up for $N = 48$ and $N = 60$ and the 4×3 configuration exhibits a slightly increased speed-up around the same interval. This seems strange and rather suspiscious, but unfortunately this has not been further investigated due to time constraints. Again, the 2×6 configuration yields the optimal speed-up of just under 120 at $N = 120$ and this would most likely increase for more processors.

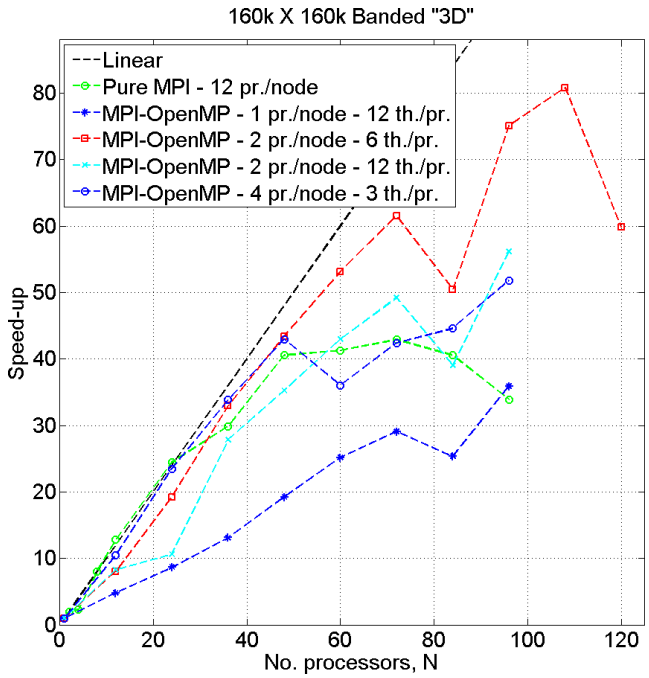


Figure 7 – Scalability plot for the 160k × 160k "3D" banded matrix, comparing hybrid implementation with pure MPI implementation.

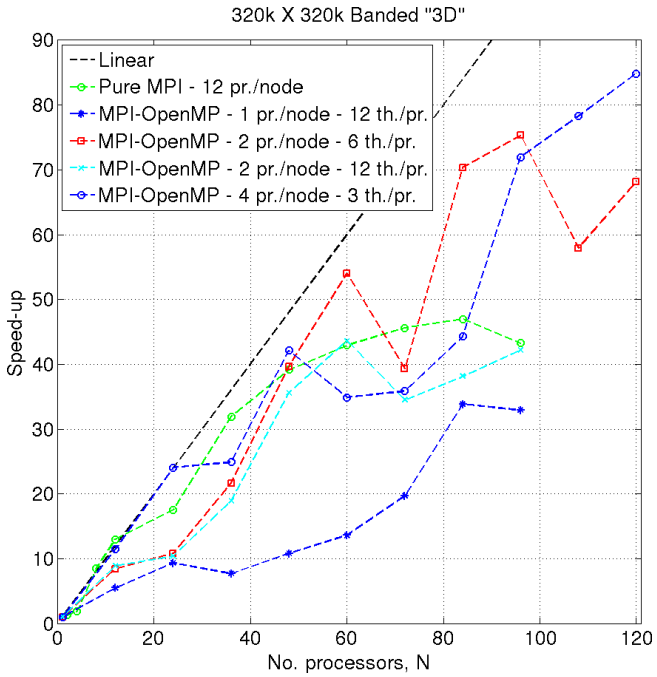


Figure 8 – Scalability plot for the 320k × 320k "3D" banded matrix, comparing hybrid implementation with pure MPI implementation.

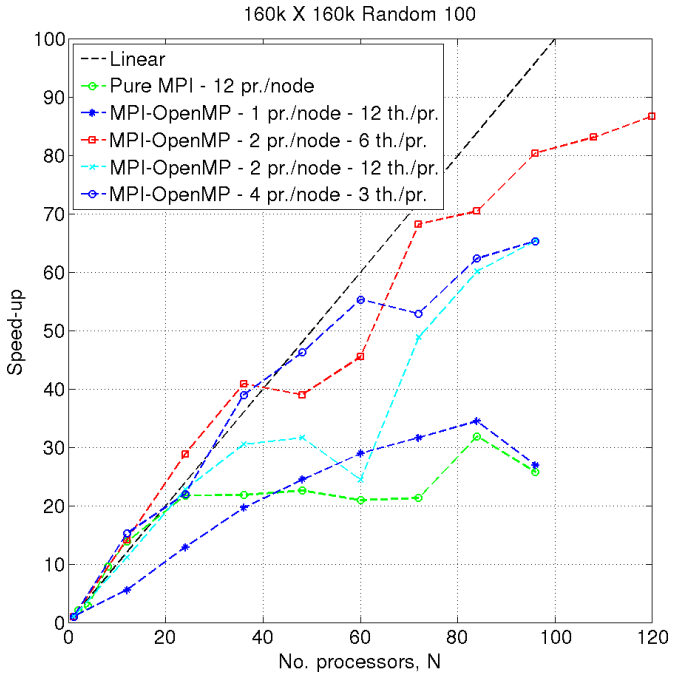


Figure 9 – Scalability plot for the 160k × 160k random-100 matrix, comparing hybrid implementation with pure MPI implementation.

Figure 11 shows the scalability achieved for the "2D" matrices using the hybrid implementation. It can be seen that for both the small and larger matrices, the pure MPI implementation performs better up until $N = 48$ and $N = 60$, respectively, where the 2×6 hybrid configuration takes over. From hereon forwards, the 2×6 configuration performs better than the pure MPI implementation, yielding an optimal speed-up of approximately 18 and 20, respectively for the small and large matrix, at $N = 120$ (with respect to the investigated interval). However, it does not seem like a viable option to approximately triple the amount of processors just to get an additional speed-up of around 2-3.

Figure 12 shows the scalability achieved for the random-10 matrices using the hybrid implementation. It is seen that the 2×6 hybrid configuration outperforms the pure MPI implementation for both matrices in the interval of $12 \leq N \leq 48$. For the 160k × 160k matrix, the 2×6 configuration does not yield a maximum speed-up that is larger than that of the pure MPI implementation - both are around 9. It does however yield it using less processors that the pure MPI implementation. For the 320k × 320k matrix, the 2×6 configuration yields a slightly higher maximal speed-up of 10, as compared to 9 using the pure MPI implementation, at $N = 24$ but drops to around the same level as the pure MPI implementation afterwards.

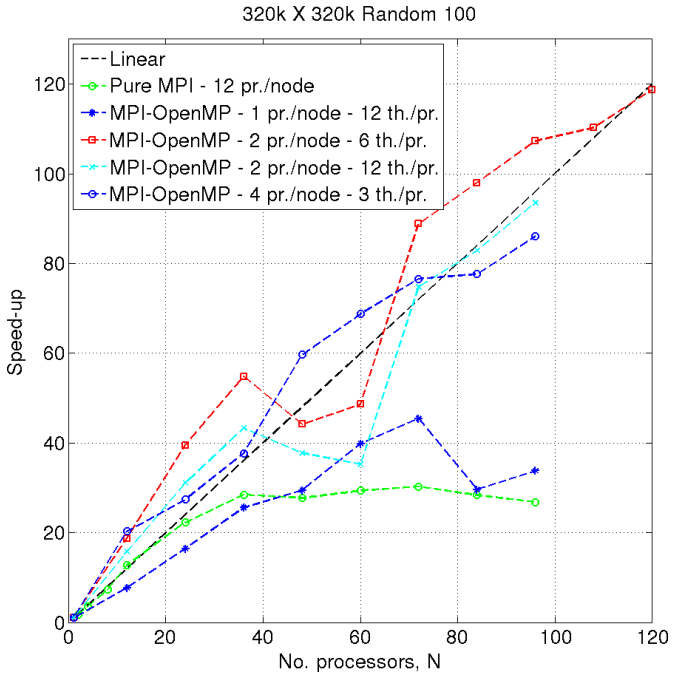
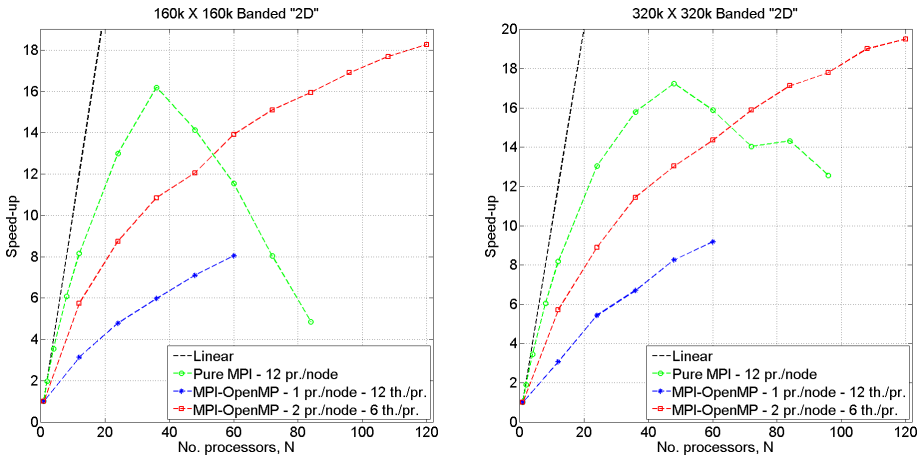
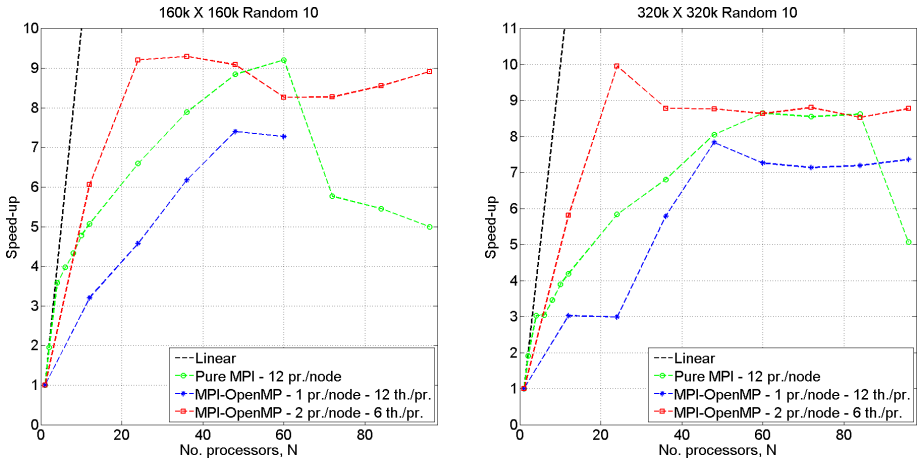


Figure 10 – Scalability plot for the 320k × 320k random-100 matrix, comparing hybrid implementation with pure MPI implementation.



(a) 160k × 160k **(b)** 320k × 320k

Figure 11 – Scalability plot for the "2D" banded matrices, comparing hybrid implementation with pure MPI implementation.

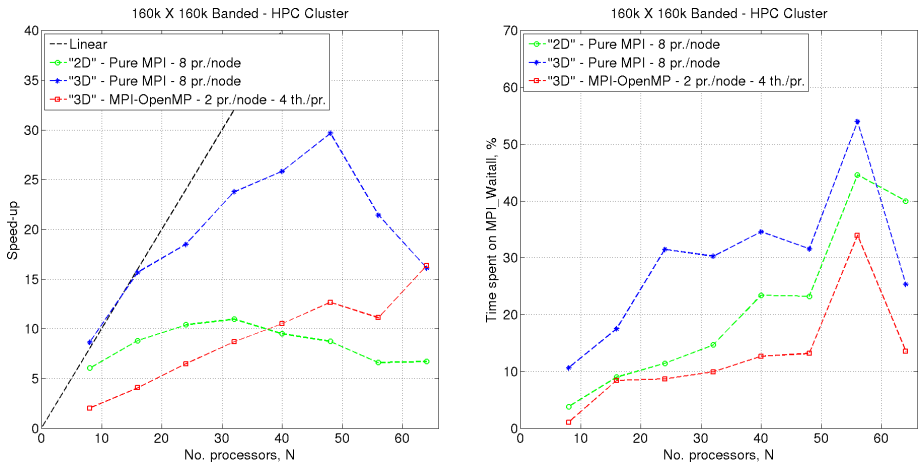


(a) 160k × 160k **(b)** 320k × 320k

Figure 12 – Scalability plot for the random-10 matrices, comparing hybrid implementation with pure MPI implementation.

3.3 Sun Studio Analyzer

The Sun Studio Analyzer is used to investigate the banded $160k \times 160k$ matrices. The analyzer is used to investigate how the percentage of time spent waiting for receiving the external vector entries, in the matrix-vector product routine, depends on the number of processors. Figure 13 shows



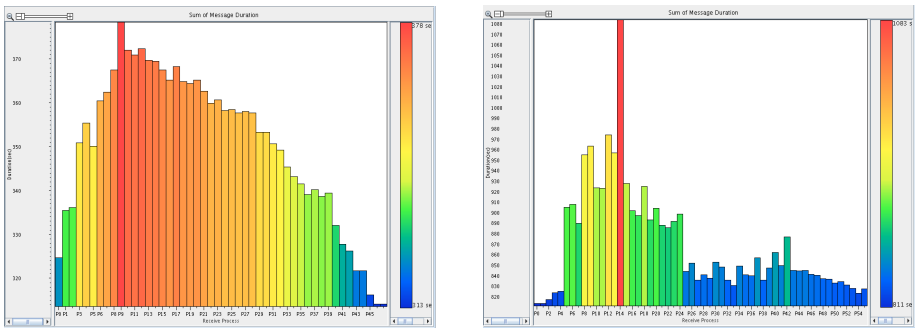
(a) Speed-up

(b) Percentage of time spent on `MPI_Waitall`

Figure 13 – Scalability plot for the "2D" banded matrices, comparing hybrid implementation with pure MPI implementation.

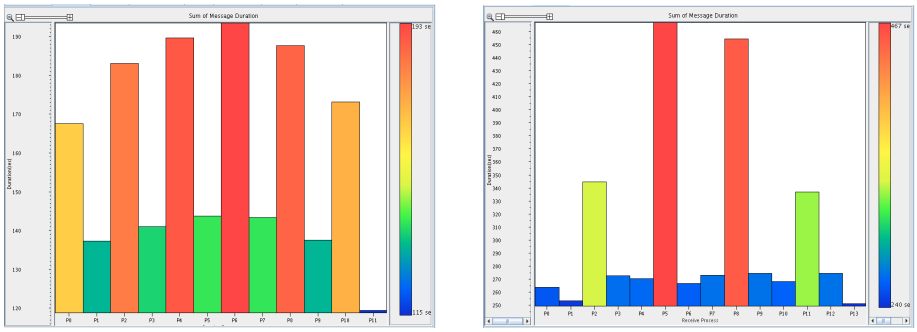
the results gained from the Sun Studio Analyzer runs. The pure MPI implementation is analysed on both the "2D" and "3D" $160k \times 160k$ banded matrices, whereas the hybrid MPI-OpenMP implementation only is analysed on the "3D" $160k \times 160k$ banded matrix. Figure 13a shows the scalability for the runs on the HPC cluster using the `collect` command. The same trend is seen as in section 3.2.2, that the "3D" scales much better than the "2D" equivalent. However, the hybrid implementation performs somewhat worse than in section 3.2.3, but it is important to remember that a different configuration has been used here, namely a 2×4 configuration with 2 processes on each node and 4 threads per process. This is due to the fact that the nodes of the HPC cluster have 8 processor cores, as compared to 12 for the TopOpt cluster. It is however interesting to note that the hybrid implementation exhibits the same behaviour as in section 3.2.3, namely the anomalous drop in speed-up at $N = 56$. Figure 13b shows the percentage of the time spent in the matrix-vector product routine, that is used on waiting for the incoming communications to finish. It is seen that all configurations in fact exhibit an increase in the percentage of time spent waiting at $N = 56$. This is rather interesting, in that it shows that something is affecting the communications when using 56 processors, or maybe rather the fact that 7

nodes are in play. The pure MPI runs have the partitioning of the data in common, in that the matrix and vectors are split into 56 partitons. The hybrid run on the other hand, is only split into 14 partitions.



(a) 6nodes – 8processes **(b)** 7nodes – 8processes

Figure 14 – Sum of durations spent on receiving data for each process for the pure MPI implementation. "3D" $160k \times 160k$ banded matrix.



(a) 6nodes – 2×4 **(b)** 6nodes – 2×4

Figure 15 – Sum of durations spent on receiving data for each process for the hybrid implementation. "3D" $160k \times 160k$ banded matrix.

Figures 14 and 15 show the sum of the duration spent on receiving data for each process. Figures 14a and 15a are for 6 active nodes and figures 14b and 15b are for 7 active nodes. It is easiliy seen that the communications are particularly unfavourable and collected to either one or two processes for the runs using 7 active nodes, as compared to a better distribution of communications for the runs using 6 active nodes. Unfortunately, time constraints have restricted further investigation into the cause of these anomalous drops in the performance, but there certainly seems to be a pattern.

4 Conclusion

It can be concluded that distributed parallel vector and sparse matrix representations have succesfully been implemented. The implementation utilises the CSR format for storing the sparse matrix entries which leads to an efficient matrix-vector product algorithm. As shown in section 3.2, the pure MPI implementation performs very well and is scalable up to approximately 48 processors for the smaller problems investigated. For the large problems, up until around 82 processors yielding a speed-up of approximately 45. The main conclusion is that the implemented hybrid MPI-OpenMP procedure for the parallel matrix-vector product shows very promising results for the tested matrices. The hybrid procedure performs especially well for the random matrices, where the entries are distributed over the entire matrix, yielding a maximum speed-up of approximately 85 and 120, for the $160k \times 160k$ and $320k \times 320k$ respectively, at 120 processors. The hybrid procedure also performs well for the banded matrices investigated, yielding a maximum speed-up of approximately 80 and 85, for the $160k \times 160k$ and $320k \times 320k$ respectively, at 108 and 120 processors. Furthermore, it can be concluded that generally it appears that the 2×6 configuration performs superior to the other configurations tested, but the 4×3 configuration also performs well and sometimes better than the prior.

5 Further work

It could also be interesting to investigate what effect the `MPI_Type_indexed` datatype actually has on the communication time and whether it can be effectivised. As it is now, the blocks are just assumed to be all of length one. But if several entries in index order are needed to be transferred, it is possible that it could be more efficient to create the datatype using blocks where entries in sequence are treated as a single block of a certain length.

Further investigation into the anomalous behaviour, exhibited by the hybrid configurations at various processor numbers, would be beneficial. It would be useful to know what triggers this behaviour, whether it is a poor distribution of workload and/or a particularly unfavourable distribution of the matrix entries.

Furthermore, it would have been very interesting to investigate a hybrid MPI-CUDA implementation, utilising the sparse matrix-vector product routine from NVIDIA's CUDA cuSPARSE library.

References

Documents

[1] Message Passing Interface Forum - MPI: A Message-Passing Interface Standard, Version 2.2 (2009), http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

Internet

[2] MatrixMarket input/output scripts for Matlab, http://math.nist.gov/MatrixMarket/mmio/matlab/mmiomatlab.html

Books

[3] N MacDonald, E Minty, T Harding, S Brown - Writing Message-Passing Parallel Programs with MPI, Course notes, University of Edinburgh

[4] WP Petersen, P Arbenz - Introduction to Parallel Computing: A Practical Guide with Examples in C, Oxford University Press (2004) ISBN: 0-19-851576-6

[5] TA Davis - Direct Methods for Sparse Linear Systems, SIAM (2006) ISBN: 0-89871-613-6

[6] B Chapman, G Jost, R van der Pas - Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press (2007) ISBN: 0-262-53302-7

A C++ code

A.1 ParVectorMap.h

```
// MPI vector map class - header

#include <mpi.h>
#include <map>
#include <vector>
#include <iostream>
#include <fstream>

class ParVectorMap
{

    private:
        // MPI variables
        MPI_Comm comm;
        int nproc;
        int rank;

        // Local bounds
        int lower_bound;
        int upper_bound;

        // Size of local and global vector
        int local_size;
        int loctot_size;
        int global_size;

        // Vector map describing process ID (rank) --> max_index+1
        std::map<int,int> vectormap;

        // Process lower and upper bound maps
        int *lprocbound_map, *uprocbound_map;

        // Maps mapping local index to global - and vice versa
        std::map<int,int> loc2glob;
        std::map<int,int> glob2loc;

        // Active "users" of vector map
        int users;

    public:

        // Constructor
        ParVectorMap(MPI_Comm ncomm, int lbound, int ubound);//, int↩
             nghost, int* global_index);
        // Destroyer
        ~ParVectorMap();

        // Convert local index to global - and vice versa
        int Loc2Glob(int local_index);
        int Glob2Loc(int global_index);

        // Getter functions
        int GetOwner(int index);
        // Quick and easy functions
        int GetRank(){return rank;};
```

```
        int GetLowerBound(){return lower_bound;};
        int GetUpperBound(){return upper_bound;};
        int GetLocalSize(){return local_size;};
        int GetGlobalSize(){return global_size;};
        int GetLocTotSize(){return loctot_size;};

        // Active "users" of vector map
        int AddUser(){users++;return users;};
        int DeleteUser(){users--; return users;};
        int GetUser(){return users;};

};
```

A.2 ParVectorMap.cc

```
// MPI vector map class
#include <ParVectorMap.h>

ParVectorMap::ParVectorMap(MPI_Comm ncomm, int lbound, int ubound)
{
    // Initialisation of MPI variables
    MPI_Comm_dup(ncomm,&comm);
    MPI_Comm_size(comm,&nproc);
    MPI_Comm_rank(comm,&rank);
    // Setting lower and upper bound
    lower_bound = lbound;
    upper_bound = ubound;
    // Calculating size of local vector
    local_size = upper_bound - lower_bound;
    // Initialising global size to 0
    global_size = 0;
    // Initialising process boundary maps to NULL
    lprocbound_map = NULL;
    uprocbound_map = NULL;

    // Acquire the global view of vector
    {
        // Allocate process boundary maps to number of processes
        lprocbound_map = new int[nproc];
        uprocbound_map = new int[nproc];

        // Gather all lower bounds and all upper bounds
        MPI_Allgather(&lower_bound,1,MPI_INT,lprocbound_map,1,↩
            MPI_INT,comm);
        MPI_Allgather(&upper_bound,1,MPI_INT,uprocbound_map,1,↩
            MPI_INT,comm);

        // Find global size and set vectormap to describe upper ↩
            bounds
        for (int i=0; i<nproc; i++) {
            vectormap[i] = uprocbound_map[i];
            if (uprocbound_map[i] > global_size) {
                global_size = uprocbound_map[i];
            }
        }

    }
```

```
    // TEMP UNTIL GHOST INTRODUCTION
    loctot_size = local_size;

}

ParVectorMap::~ParVectorMap()
{
    // Free communicator
    MPI_Comm_free(&comm);
    // Deallocate process bound maps
    if (lprocbound_map!=NULL) {
        delete [] lprocbound_map;
    }
        if (uprocbound_map!=NULL) {
        delete [] uprocbound_map;
    }

}

int ParVectorMap::GetOwner(int index)
{
    // If index is not outside of the global vector
    if ( (index < global_size) && (index >= 0) ) {
        // Check upper bounds for processes, until index is lower ↩
            and return the process.
        for (int i = 0; i<nproc; i++) {
            if (index < uprocbound_map[i]) {return i;}
        }
    } else {
        return -1;
    }

}

int ParVectorMap::Loc2Glob(int local_index)
{
    // If local index is within bounds of the local vector
    if ( (local_index < local_size)&&(local_index >= 0) ) {
        return lower_bound + local_index;
    } else {
        // until introduction of ghosts
        return -1;
    }

}

int ParVectorMap::Glob2Loc(int global_index)
{
    // If global index is within the bounds for the local process
    if ( (global_index >= lower_bound)&&(global_index < upper_bound)↩
         ) {
        return (global_index - lower_bound);
    } else {
        // until introduction of ghosts
        return -1;
    }

}
```

A.3 ParVector.h

```
// MPI vector class - header

#include <mpi.h>
// Input/output
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>


#include <ParVectorMap.h>

class ParVector
{

    private:

        // Local array and size
        double *array;
        int array_size;
        int local_size;

        // Vector index map
        ParVectorMap *index_map;

    public:

        // Constructors
        ParVector();
        ParVector(MPI_Comm ncomm, int lbound, int ubound);
        // Destructor
        ~ParVector();

        // Getters
        ParVectorMap* GetVectorMap(){return index_map;};
        int GetLowerBound();
                int GetUpperBound();
                int GetLocalSize();
                int GetGlobalSize();
                int GetArraySize();
                double* GetArray(){return array;};

        // Adders and setters
        void AddValueLocal(int row, double value);
        void AddValuesLocal(int nindex, int *rows, double* values);
        void SetToValue(double value);
        void SetToZero();


        // External reader
        void ReadExtVec();

                // Convert local index to global - and vice versa
                int Loc2Glob(int local_index);
                int Glob2Loc(int global_index);

                void RestoreArray(){};

};
```

A.4 ParVector.cc

```
// MPI vector class
#include <ParVector.h>

ParVector::ParVector()
{

    array = NULL;
    array_size = 0;
    local_size = 0;
    index_map = NULL;

}

ParVector::ParVector(MPI_Comm ncomm, int lbound, int ubound)
{

    // Generate vector index map
    index_map = new ParVectorMap(ncomm, lbound, ubound);
    index_map->AddUser();

    // Local size
    local_size = index_map->GetLocalSize();

    // Allocate array of size (local + ghosts)
    array_size = index_map->GetLocTotSize();
    array = new double[array_size];
}

ParVector::~ParVector()
{
    // If index_map has been defined
    if (index_map != NULL) {
        index_map->DeleteUser();
        // If there are no more users of index_map, then delete it
        if( index_map->GetUser() == 0 ) {
            delete index_map;
        }
    }

    // If array has been defined
    if (array != NULL) {
        delete [] array;
    }

}


int ParVector::GetLowerBound()
{

    // If index_map has been defined --> return maps lower bound
    if (index_map != NULL) {
        return index_map->GetLowerBound();
    // Else return 0
    } else {
        return 0;
    }
```

```
}

int ParVector::GetUpperBound()
{

        // If index_map has been defined --> return maps upper bound
        if (index_map != NULL) {
                return index_map->GetUpperBound();
        // Else return 0
        } else {
                return 0;
        }


}

int ParVector::GetLocalSize()
{

    return local_size;

}

int ParVector::GetGlobalSize()
{

    // If index_map has been defined --> return maps global size
        if (index_map != NULL) {
                return index_map->GetGlobalSize();
        // Else return 0
        } else {
                return 0;
        }

}

int ParVector::GetArraySize()
{

    return array_size;

}

void ParVector::AddValueLocal(int row, double value)
{
    // If specified location is within the local array size, add it ↩
        to existing value
    if (row < array_size) {
        array[row] = array[row] + value;
    }
}

void ParVector::AddValuesLocal(int nindex, int *rows, double* values↩
    )
{
    // For all specified locations, add to existing
    for (int i = 0; i < nindex; i++) {
        AddValueLocal(rows[i],values[i]);
    }

}
```

```
void ParVector::SetToValue(double value)
{
    // Set entire array to a specified value
    for (int i = 0; i < array_size; i++) {
        array[i] = value;
    }

}

void ParVector::SetToZero()
{
    // Set entire array to zero
    SetToValue(0.0);

}

int ParVector::Loc2Glob(int local_index)
{
    // Converts local index to global index using index map
        if ( index_map != NULL ) {
                return index_map->Loc2Glob(local_index);
        } else {
                return -1;
        }

}

int ParVector::Glob2Loc(int global_index)
{
    // Converts global index to local index using index map
        if ( index_map != NULL ) {
                return index_map->Glob2Loc(global_index);
        } else {
                return -1;
        }

}


void ParVector::ReadExtVec()
{

    std::ifstream file("vector.vec");
    std::string   line;

    int lower_bound = GetLowerBound();
    int upper_bound = GetUpperBound();
        int val1, dummy;
        double val2;
    int quit;

    // Read past first few lines of input file that are not numbers
    while (std::getline(file,line)) {
        val1 = 0; val2 = 0.0;
        std::stringstream linestream(line);
        linestream >> val1 >> dummy >> val2;
        if ( dummy != 0 && val1 != 0 && val2 != 0.0) {
            break;
        }
    }

    // Start reading in coordinates and if local -> add to vector
```

```
    while(std::getline(file, line))
    {

        val1 = 0; val2 = 0.0;
        std::stringstream linestream(line);
        linestream >> val1 >> dummy >> val2;
        val1 = val1 - 1;
        if ( (val1 >= lower_bound) && (val1 < upper_bound ) ) {
            AddValueLocal(index_map->Glob2Loc(val1),val2);
        }

    }



}
```

A.5 MatrixCSR.h

```
struct MatrixCSR{

        int nrows;
        int nnz;
        int* rows;
        int* cols;
        double* vals;

    MatrixCSR()
    {
                nnz=0;
                rows=NULL;
                cols=NULL;
                vals=NULL;
    };

        MatrixCSR(int nnz_in,int nrows_in)
        {
        if (nnz_in != 0 && nrows_in != 0) {
            nnz=nnz_in;
            rows = new int[nrows_in+1];
            cols = new int[nnz];
            vals = new double[nnz];
        } else {
            nnz =0;
            rows=NULL;
            cols=NULL;
            vals=NULL;
        }

        };

    ~MatrixCSR()
    {
        if(nnz!=0) {
            delete [] rows;
            delete [] cols;
            delete [] vals;
```

```
        }
    };

    void Free(){
        if(nnz!=0) {
            delete [] rows;
            delete [] cols;
            delete [] vals;
            nnz=0;
            rows=NULL;
            cols=NULL;
            vals=NULL;
            }
    };

};
```

A.6 ParMatrixSparse.h

```
// MPI vector class - header

#include <mpi.h>
// Input/output
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

//#include <ParVectorMap.h>
#include <ParVector.h>

// CRS Struct
#include <MatrixCSR.h>

class ParMatrixSparse
{

    private:
        // Dynamic maps for building matrix
        std::map<int, double> *dynmat_lloc, *dynmat_gloc;
        // Size of local matrix slab
        int ncols, nrows;

        // CSR structs for utilising matrix structure (for product)
        MatrixCSR *CSR_lloc, *CSR_gloc;
        // Number of nonzeroes
        int nnz_lloc, nnz_gloc;

        // Vector maps corresponding to x and y direction (or ↩
            vectors)
        ParVectorMap *x_index_map;
        ParVectorMap *y_index_map;

        // Values defining local-local domain
        int njloc;
        int lower_x, lower_y;
        int upper_x, upper_y;
```

```
        // MPI process ID (rank) and number of processes
        int ProcID, nProcs;

        // Pointers and buffers for setting up communications
        int *VNumRecv, *VNumSend;
        int maxRecv, maxSend;
        int **Rbuffer, **Sbuffer;

        // MPI datatype pointers for storing derived datatypes
        MPI_Datatype *DTypeRecv, *DTypeSend;

    public:

        // Constructors
        ParMatrixSparse();
        ParMatrixSparse(ParVector* XVec, ParVector* YVec);
        // Destructor
        ~ParMatrixSparse();

        // Getters
        ParVectorMap* GetXMap(){return x_index_map;};
        ParVectorMap* GetYMap(){return y_index_map;};
        int GetXLowerBound();
                int GetXUpperBound();
        int GetYLowerBound();
                int GetYUpperBound();
                void GetTrueLocalSize(int& rs, int& cs){rs=nrows; cs=↩
                    =njloc;};
                void GetLocalSize(int& rs, int& cs){rs=nrows; cs=↩
                    ncols;};
        std::map<int, double>* GetDynMatLocLoc(){return dynmat_lloc↩
            ;};
        std::map<int, double>* GetDynMatGlobLoc(){return dynmat_gloc↩
            ;};
        MatrixCSR* GetCSRLocLoc(){return CSR_lloc;};
        MatrixCSR* GetCSRGlobLoc(){return CSR_gloc;};

        // Adders
        void AddValueLocal(int row, int col, double value);
        void AddValuesLocal(int nindex, int *rows, int *cols, double↩
            * values);

        // External reader
        void ReadExtMat();
        // Convert from dynamic map to CSR
        void ConvertToCSR();

        // Find columns to receive from remote vector processes
        void FindColsToRecv();

        // Transfer ghost values from remote vector processes
        void SetupDataTypes();

        // Test of communication - for debugging
        void TestCommunication(ParVector* XVec, ParVector* YVec);

        // Matrix-vector product
        void MatVecProd(ParVector* XVec, ParVector* YVec);

};
```

A.7 ParMatrixSparse.cc

```
// MPI vector class
#include <ParMatrixSparse.h>

ParMatrixSparse::ParMatrixSparse()
{

    dynmat_lloc = NULL;
    dynmat_gloc = NULL;

    CSR_lloc = NULL;
    CSR_gloc = NULL;

    nnz_lloc = 0;
    nnz_gloc = 0;

    ncols = 0;
    nrows = 0;
    njloc = 0;

    lower_x = 0;
    lower_y = 0;
    upper_x = 0;
    upper_y = 0;

    x_index_map = NULL;
    y_index_map = NULL;

    MPI_Comm_rank(MPI_COMM_WORLD,&ProcID);
    MPI_Comm_size(MPI_COMM_WORLD,&nProcs);

    VNumRecv = NULL; VNumSend = NULL;
    Rbuffer = NULL; Sbuffer = NULL;

    DTypeRecv = NULL; DTypeSend = NULL;

}

ParMatrixSparse::ParMatrixSparse(ParVector* XVec, ParVector* YVec)
{

    dynmat_lloc = NULL;
    dynmat_gloc = NULL;

    CSR_lloc = NULL;
    CSR_gloc = NULL;

    nnz_lloc = 0;
    nnz_gloc = 0;

        x_index_map = NULL;
        y_index_map = NULL;

        ncols = 0;
        nrows = 0;
        njloc = 0;

        lower_x = 0;
        lower_y = 0;
```

```
        upper_x = 0;
        upper_y = 0;

    MPI_Comm_rank(MPI_COMM_WORLD,&ProcID);
    MPI_Comm_size(MPI_COMM_WORLD,&nProcs);

    VNumRecv = NULL; VNumSend = NULL;
    Rbuffer = NULL; Sbuffer = NULL;

    DTypeRecv = NULL; DTypeSend = NULL;


    // Get vector map for x and y direction
    x_index_map = XVec->GetVectorMap();
    x_index_map->AddUser();
    y_index_map = YVec->GetVectorMap();
    y_index_map->AddUser();

    if (x_index_map != NULL && y_index_map != NULL) {
        // Get number of rows and columns belonging to process
        ncols = x_index_map->GetGlobalSize();
        nrows = y_index_map->GetLocalSize();
        njloc = x_index_map->GetLocalSize();
        // Get upper and lower bounds
        lower_x = x_index_map->GetLowerBound();
        lower_y = y_index_map->GetLowerBound();
        upper_x = x_index_map->GetUpperBound();
        upper_y = y_index_map->GetUpperBound();

    }


}

ParMatrixSparse::~ParMatrixSparse()
{
    // If index_map has been defined
    if (x_index_map != NULL) {
        x_index_map->DeleteUser();
        // If there are no more users of index_map, then delete it
//      if( x_index_map->GetUser() == 0 ) {
            delete x_index_map;
//      }
    }
    if (y_index_map != NULL) {
        y_index_map->DeleteUser();
//      if( y_index_map->GetUser() == 0 ) {
            delete y_index_map;
//      }
    }

    // If dynmat has been defined
    if (dynmat_lloc != NULL) {
        delete [] dynmat_lloc;
    }
    if (dynmat_gloc != NULL) {
        delete [] dynmat_gloc;
    }
    if (CSR_lloc != NULL) {
        delete CSR_lloc;
    }
    if (CSR_gloc != NULL) {
```

```
        delete CSR_gloc;
    }
    if (VNumRecv != NULL) {
        delete [] VNumRecv;
    }
    if (VNumSend != NULL) {
        delete [] VNumSend;
    }
    if (Rbuffer != NULL) {
        int i;
        for (i=0;i<nProcs;i++){
            if (Rbuffer[i] != NULL) {
                delete [] Rbuffer[i];
            }
        }
        delete [] Rbuffer;
    }
    if (Sbuffer != NULL) {
        int i;
        for (i=0;i<nProcs;i++){
            if (Sbuffer[i] != NULL) {
                delete [] Sbuffer[i];
            }
        }
        delete [] Sbuffer;
    }

    if (DTypeRecv != NULL) {
        int i;
        for (i=0;i<nProcs;i++){
            if (DTypeRecv[i] != MPI_DATATYPE_NULL) {
                MPI_Type_free(&DTypeRecv[i]);
            }
        }
        delete [] DTypeRecv;
    }
    if (DTypeSend != NULL) {
        int i;
        for (i=0;i<nProcs;i++){
            if (DTypeSend[i] != MPI_DATATYPE_NULL) {
                MPI_Type_free(&DTypeSend[i]);
            }
        }
        delete [] DTypeSend;
    }

//  printf("ParMatrixSparse deleted.\n");

}


int ParMatrixSparse::GetXLowerBound()
{

    // If index_map has been defined --> return maps lower bound
    if (x_index_map != NULL) {
        return x_index_map->GetLowerBound();
    // Else return 0
    } else {
        return 0;
    }
```

```
}

int ParMatrixSparse::GetXUpperBound()
{

        // If index_map has been defined --> return maps upper bound
        if (x_index_map != NULL) {
                return x_index_map->GetUpperBound();
        // Else return 0
        } else {
                return 0;
        }


}

int ParMatrixSparse::GetYLowerBound()
{

    // If index_map has been defined --> return maps lower bound
    if (y_index_map != NULL) {
        return y_index_map->GetLowerBound();
    // Else return 0
    } else {
        return 0;
    }

}

int ParMatrixSparse::GetYUpperBound()
{

        // If index_map has been defined --> return maps upper bound
        if (y_index_map != NULL) {
                return y_index_map->GetUpperBound();
        // Else return 0
        } else {
                return 0;
        }


}

void ParMatrixSparse::AddValueLocal(int row, int col, double value)
{

    std::map<int,double>::iterator it;
    // If location is within local-local area then add to local-↩
        local dynamic map
    if ( (row < nrows && row >= 0) && (col < upper_x && col >= ↩
        lower_x && col >= 0) ) {
        if (dynmat_lloc == NULL) {dynmat_lloc = new std::map<int,↩
            double> [nrows];}
        it = dynmat_lloc[row].find(col);
        if ( it == dynmat_lloc[row].end() ){
            dynmat_lloc[row][col] = value;
            nnz_lloc++;
        } else {
            it->second = it->second + value;
        }
    // If location is within local-global area then add to local-↩
        global dynamic map
```

```
    } else if ( (row < nrows && row >= 0) && (col >= upper_x || col <
        lower_x) && (col >=0) ) {
        if (dynmat_gloc == NULL) {dynmat_gloc = new std::map<int,
            double> [nrows];}
        it = dynmat_gloc[row].find(col);
        if ( it == dynmat_gloc[row].end() ){
            dynmat_gloc[row][col] = value;
            nnz_gloc++;
        } else {
            it->second = it->second + value;
        }
    }

}

void ParMatrixSparse::AddValuesLocal(int nindex, int *rows, int *
    cols, double* values)
{

    std::map<int,double>::iterator it;

    for (int i = 0; i<nindex; i++) {
        AddValueLocal(rows[i],cols[i],values[i]);
    }

}

void ParMatrixSparse::ConvertToCSR()
{

    int count;
        int i, j, k;
    double v;
        std::map<int, double>::iterator it;


    if (dynmat_lloc != NULL) {
        // Allocate CSR structures
        CSR_lloc = new MatrixCSR(nnz_lloc,nrows);

        // Convert local-local to CSR format
        count = 0; CSR_lloc->rows[0] = 0;
        for (i = 0; i < nrows; i++) {
            CSR_lloc->rows[i] = count;
            for (it = dynmat_lloc[i].begin(); it != dynmat_lloc[i].
                end(); it++) {
                j = it->first;
                v = it->second;
                CSR_lloc->vals[count] = v;
                CSR_lloc->cols[count] = j;
                count++;
            }
        }
        CSR_lloc->rows[nrows] = nnz_lloc;
    }

    if (dynmat_gloc != NULL) {
        CSR_gloc = new MatrixCSR(nnz_gloc,nrows);
        // Convert global-local to CSR format
        count = 0; CSR_gloc->rows[0] = 0;
        for (i = 0; i < nrows; i++) {
            CSR_gloc->rows[i] = count;
```

```
            for (it = dynmat_gloc[i].begin(); it != dynmat_gloc[i].↩
                end(); it++) {
                j = it->first;
                v = it->second;
                CSR_gloc->vals[count] = v;
                CSR_gloc->cols[count] = j;
                count++;
            }
        }
        CSR_gloc->rows[nrows] = nnz_gloc;
    }
}

void ParMatrixSparse::ReadExtMat()
{

    // Reader
    std::ifstream file("matrix.mat");
    std::string line;
    int row, col;
    double value;
    row = 0; col = 0; value = 0.0;
    int quit = 0;

    // Read past start of input file that is not numbers
        while (std::getline(file,line)) {
                row = 0; col = 0; value = 0.0;
                std::stringstream linestream(line);
                linestream >> row >> col >> value;
                if ( row != 0 && col != 0 && value != 0.0) {
                        break;
                }
        }

    // Read in values from input file and add to matrix
    while (std::getline(file,line)) {
        std::stringstream linestream(line);
                linestream >> row >> col >> value;
        row = row - 1; col = col -1;
        if ( (row >= lower_y && row < upper_y) && (col < ncols) ) {
            AddValueLocal(y_index_map->Glob2Loc(row),col,value);
        }
    }


}



void ParMatrixSparse::FindColsToRecv()
{

    std::map<int,int> Rrows;
    std::map<int,int> Srows;
    std::map<int,int>::iterator vit;
    std::map<int,double>::iterator mit;
    int i, j, k;
    int count, count1; count = 0; count1 = 0;

    int nRecv;
    MPI_Comm_rank(MPI_COMM_WORLD,&ProcID);
    MPI_Comm_size(MPI_COMM_WORLD,&nProcs);
```

```
    // Initialise vector containing number of entries to send and ↩
        receive from each process
    VNumRecv = new int[nProcs];
    for (i=0;i<nProcs;i++) {VNumRecv[i]=0;}
    VNumSend = new int[nProcs];
    for (i=0;i<nProcs;i++) {VNumSend[i]=0;}

    MPI_Request *Rreqs, *Sreqs;
    MPI_Status status, *Rstat, *Sstat;
    int tag1, tag2; tag1 = 0; tag2 = 1;
    int Rtag, Stag;
    Rtag = 0; Stag = 1;

    int maxRecv, maxSend;

//////////////////////////////////////////////////////

    if (dynmat_gloc != NULL) {
        // Figure out which rows from remote vectors are needed and ↩
            to which process they belong
        for (i = 0; i<nrows; i++) {
            for (mit = dynmat_gloc[i].begin(); mit != dynmat_gloc[i↩
                ].end(); mit++) {
                j = mit->first;
                vit = Rrows.find(j);
                if (vit == Rrows.end()) {
                    Rrows[j] = x_index_map->GetOwner(j);
                    VNumRecv[Rrows[j]]=VNumRecv[Rrows[j]]+1;
                    count++;
                }
            }
        }
        nRecv = count;
    } else {
        for (i = 0; i < nProcs; i++) {
                    VNumRecv[i] = 0;
        }

    }

    // MPI non-blocking requests and statuses
    Rreqs = new MPI_Request[nProcs-1];
    Sreqs = new MPI_Request[nProcs-1];
    Rstat = new MPI_Status[nProcs-1];
    Sstat = new MPI_Status[nProcs-1];

    // Post sends and receives for the number of entries to send and↩
         receive from each process
    count = 0; maxRecv = 0; maxSend = 0;
    for (i = 0; i < nProcs; i++) {
        if (VNumRecv[i] > maxRecv){maxRecv = VNumRecv[i];}
        if (i != ProcID) {
            MPI_Isend(&VNumRecv[i],1,MPI_INT,i,tag1,MPI_COMM_WORLD,&↩
                Sreqs[count]);
            MPI_Irecv(&VNumSend[i],1,MPI_INT,i,tag1,MPI_COMM_WORLD,&↩
                Rreqs[count]);
            count++;
        }
    }

    // Wait for receives to finish
```

```
    MPI_Waitall(nProcs-1,Rreqs,Rstat);
    // Find max number to send
        for (i = 0; i < nProcs; i++) {
        if (VNumSend[i] > maxSend){maxSend = VNumSend[i];}
        }


    // Initialisation of send and receive buffers
    Rbuffer = new int*[nProcs];
    Sbuffer = new int*[nProcs];
    for (i = 0; i < nProcs; i++) {Rbuffer[i] = NULL;}
    for (i = 0; i < nProcs; i++) {Sbuffer[i] = NULL;}

    // Post sends for row indices
    // MPI non-blocking requests and statuses
    count = 0; count1 = 0;
    for (i = 0; i < nProcs; i++) {
        count = 0;
        if (ProcID != i) {
            Sbuffer[i] = new int[VNumRecv[i]];
            // Load Sbuffer with column nums needed for given ↩
                process and send to process
            for (vit = Rrows.begin(); vit != Rrows.end(); vit++){
                if (vit->second == i) {
                    Sbuffer[i][count] = vit->first;
                    count++;
                }
            }
            MPI_Isend(Sbuffer[i],VNumRecv[i],MPI_INT,i,tag1,↩
                MPI_COMM_WORLD,&Sreqs[count1]);
            count1 ++;
        } else {
            Sbuffer[i] = new int[1]; Sbuffer[i][0] = 0;
        }
    }

    count1 = 0;
    for (i = 0; i< nProcs; i++) {
        if (ProcID != i) {
            Rbuffer[i] = new int[VNumSend[i]];
            MPI_Irecv(Rbuffer[i],VNumSend[i],MPI_INT,i,tag1,↩
                MPI_COMM_WORLD,&Rreqs[count1]);
            count1++;
        } else {
            Rbuffer[i] = new int[1]; Rbuffer[i][0] = 0;
        }
    }


    // Wait for receives to finish
        MPI_Waitall(nProcs-1,Rreqs,Rstat);

    delete [] Rreqs;
        delete [] Sreqs;
        delete [] Rstat;
        delete [] Sstat;

}


void ParMatrixSparse::SetupDataTypes()
{
```

```
    int i, j, k;
    int count, *blength, *displac;

    // int MPI_Type_indexed(int count, int *array_of_blocklengths,
    //          int *array_of_displ, MPI_Datatype old, MPI_Datatype ↩
        new);
    // Sending:
    // count = VNumSend[i], array_of_blocklengths = vector of ones ↩
        of length count,
    // array_of_displ = vector of displacement found from ↩
        x_index_map->Glob2Loc of Rbuffer[i][j],
    // old = MPI_DOUBLE, new = DTypeSend[i]
    // Receiving:
    // count = VNumRecv[i], array_of_blocklengths = vector of ones ↩
        of length count,
    // array_of_displ = vector of displacements found from ↩
        x_index_map->Glob2Loc of Sbuffer[i][j],
    // old = MPI_DOUBLE, new = DTypeRecv[i]

    DTypeSend = new MPI_Datatype[nProcs];
    DTypeRecv = new MPI_Datatype[nProcs];

    for (i = 0; i < nProcs; i++) {
        count = VNumSend[i];
        blength = new int[count];
        displac = new int[count];
        // Set up arrays of blocklengths and displacements
        for (j = 0; j < count; j++) {
            blength[j] = 1;
            displac[j] = x_index_map->Glob2Loc(Rbuffer[i][j]);
        }
        // Initialise send datatype and commit
        MPI_Type_indexed(count, blength, displac, MPI_DOUBLE, &↩
            DTypeSend[i]);
        MPI_Type_commit(&DTypeSend[i]);

        count = VNumRecv[i];
        blength = new int[count];
        displac = new int[count];
        // Set up arrays of blocklengths and displacements
        for (j = 0; j < count; j++) {
            blength[j] = 1;
            displac[j] = Sbuffer[i][j];
        }
        // Initialise recv datatype and commit
        MPI_Type_indexed(count, blength, displac, MPI_DOUBLE, &↩
            DTypeRecv[i]);
        MPI_Type_commit(&DTypeRecv[i]);
    }


    delete [] blength;
    delete [] displac;


}


void ParMatrixSparse::TestCommunication(ParVector* XVec, ParVector* ↩
    YVec)
{
```

```
    // TEST OF COMMUNICATIONS FOR DEBUGGING

    int i, j, k, l, ng;
    int sender, receiver;
    sender = 2; receiver = 1;

    MPI_Status Rstat;

    double *rBuf, *sBuf;
//  rBuf = new double[VNumRecv[sender]];
    k = XVec->GetLocalSize();
    l = XVec->GetGlobalSize();
    rBuf = new double[l];
    for (i=0;i<l;i++){rBuf[i] = 0;}
    sBuf = XVec->GetArray();

    if (ProcID == sender) {
//      for (i=0;i<k;i++){printf("sBuf[%d] = %f\n",i,sBuf[i]);}
        MPI_Send(sBuf,1,DTypeSend[receiver],receiver,1,↩
            MPI_COMM_WORLD);
        printf("Sending complete.\n");
    } else if (ProcID == receiver) {
//      for (i=0;i<l;i++){printf("rBuf[%d] = %f\n",i,rBuf[i]);}
        MPI_Recv(rBuf,1,DTypeRecv[sender],sender,1,MPI_COMM_WORLD,&↩
            Rstat);
//      for (i=0;i<l;i++){printf("rBuf[%d] = %f\n",i,rBuf[i]);}
        printf("Receiving complete.\n");
    }

    if(ProcID == sender){for (i=0;i<k;i++){printf("sBuf[%d] = %f\n",↩
        i,sBuf[i]);}}
    MPI_Barrier(MPI_COMM_WORLD);
    if(ProcID == receiver){for (i=0;i<l;i++){printf("rBuf[%d] = %f\n↩
        ",i,rBuf[i]);}}

//  delete [] rBuf;
//  delete [] sBuf;

}


void ParMatrixSparse::MatVecProd(ParVector* XVec, ParVector* YVec)
{

    int i, j, k, l;
    int llength, glength;
    int count; count = 0;
    int count2; count2 = 0;
    int tag1; tag1 = 0;
    double v; v = 0.0;

    MPI_Request *Rreqs, *Sreqs;
    MPI_Status *Rstat, *Sstat;

    double *rBuf, *sBuf;
    // Get local and global length
    llength = XVec->GetLocalSize();
    glength = XVec->GetGlobalSize();
    // Setting up recv and send buffers
    rBuf = new double[glength];
    for (i=0;i<glength;i++){rBuf[i] = 0;}
```

```
    sBuf = XVec->GetArray();

        // MPI non-blocking requests and statuses
        Rreqs = new MPI_Request[nProcs-1];
        Sreqs = new MPI_Request[nProcs-1];
        Rstat = new MPI_Status[nProcs-1];
        Sstat = new MPI_Status[nProcs-1];

        // Post sends and receives each process
        count = 0;
        for (i = 0; i < nProcs; i++) {
            if (i != ProcID) {
            if (DTypeSend[i] != MPI_DATATYPE_NULL) {
                MPI_Isend(sBuf,1,DTypeSend[i],i,tag1,MPI_COMM_WORLD↩
                    ,&Sreqs[count]);
            } else {
                Sreqs[count] = MPI_REQUEST_NULL;
            }
            if (DTypeRecv[i] != MPI_DATATYPE_NULL) {
                MPI_Irecv(rBuf,1,DTypeRecv[i],i,tag1,MPI_COMM_WORLD↩
                    ,&Rreqs[count]);
            } else {
                Rreqs[count] = MPI_REQUEST_NULL;
            }
            count++;
                }
        }

#ifndef _OPENMP

    // Calculate local-local product
    if (CSR_lloc != NULL) {
        for (i = 0; i < nrows; i++){
            for(k = CSR_lloc->rows[i]; k < CSR_lloc->rows[i+1]; k++)↩
                {
                j = x_index_map->Glob2Loc(CSR_lloc->cols[k]);
                v = CSR_lloc->vals[k]*sBuf[j];
                YVec->AddValueLocal(i,v);
            }
        }
    }
    // Wait for receiving of ghostvalues to complete
    MPI_Waitall(nProcs-1,Rreqs,Rstat);

    // Calculate local-global product
    if (CSR_gloc != NULL) {
        for (i = 0; i < nrows; i++){
            for(k = CSR_gloc->rows[i]; k < CSR_gloc->rows[i+1]; k++)↩
                {
                j = CSR_gloc->cols[k];
                v = CSR_gloc->vals[k]*rBuf[j];
                YVec->AddValueLocal(i,v);
            }
        }
    }

#else
//  std::cout << "OpenMP run!!!!!!!!!!!!!!!!!!" << std::endl;
    #pragma omp parallel default(shared) private(i,j,k,v)
    { // Start omp parallel region
        // Calculate local-local product
        if (CSR_lloc != NULL) {
```

```
            #pragma omp for schedule(guided)//schedule(runtime)
                for (i = 0; i < nrows; i++){
                    for(k = CSR_lloc->rows[i]; k < CSR_lloc->rows[i↩
                        +1]; k++){
                        j = x_index_map->Glob2Loc(CSR_lloc->cols[k])↩
                            ;
                        v = CSR_lloc->vals[k]*sBuf[j];
                        YVec->AddValueLocal(i,v);
                    }
                }
        }
        // Wait for receiving of ghostvalues to complete
        #pragma omp barrier
        #pragma omp master
        {
            MPI_Waitall(nProcs-1,Rreqs,Rstat);
        }
        #pragma omp barrier
        // Calculate local-global product
        if (CSR_gloc != NULL) {
            #pragma omp for schedule(guided)//schedule(runtime)
                for (i = 0; i < nrows; i++){
                    for(k = CSR_gloc->rows[i]; k < CSR_gloc->rows[i↩
                        +1]; k++){
                        j = CSR_gloc->cols[k];
                        v = CSR_gloc->vals[k]*rBuf[j];
                        YVec->AddValueLocal(i,v);
                    }
                }
        }

    } // End of omp parallel region
#endif

    delete [] rBuf;
    delete [] Rreqs;
        delete [] Sreqs;
        delete [] Rstat;
        delete [] Sstat;

}
```

A.8 Timing driver: ParaMatVecTiming.cc

```
// // MPI library
#include <mpi.h>
// IO streams
#include <stdio.h>
#include <iostream>
// Maths library
#include <math.h>
// C Standard library
#include <stdlib.h>
// String library
#include <string.h>
// STL Map
#include <map>
```

```
// MPI Vector class
//#include <ParVector.h>
// MPI Sparse Matrix class
#include <ParMatrixSparse.h>


int main(int argc, char *argv[])
{

    // Printing switches
    int printEnds = 1, printMids = 1;

    // Variables
    int i, j, k;
    int span, lower_b, upper_b;

        // Declaration of MPI variables and constants
        const int tag1 = 0, tag2 = 1;
        int rank, size, intBuf1, intBuf2, number;
        double doubBuf1, start, finish, time;
        MPI_Status status;

    // Checking for commandline input
    int commMethod = 0;
    int probSize = 10;
    int maxCount = 1;
    if(argc>=2){
        for (int i = 0; i<argc; i++) {
            if (strcasecmp(argv[i],"-comm")==0) {
                commMethod = atoi(argv[i+1]);
            }
            if (strcasecmp(argv[i],"-size")==0) {
                                probSize = atoi(argv[i+1]);
                        }
            if (strcasecmp(argv[i],"-count")==0) {
                                maxCount = atoi(argv[i+1]);
                        }
        }
    }


    // MPI initialisation
#ifndef _OPENMP
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {std::cout << "Ordinary run!!!!!!!!!!!" << std::↩
        endl;}
#else
    int dummy; dummy = 0;
    MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,&dummy);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {std::cout << "OpenMP run!!!!!!!!!!!!!" << std::↩
        endl;}
#endif

    // Initialisation
        span = int(floor(double(probSize)/double(size)));
    if (rank == size-1) {
        lower_b = rank*span;
        upper_b = probSize;
```

```
    } else {
        lower_b = rank*span;
        upper_b = (rank+1)*span;
    }
    if (rank==0) {
        printf("------------------------------------------------------------\n");
        printf("-----      Parallel matrix-vector product      -----\n");
        printf("----- %3d processes - %6d x %6d matrix -----\n",size,probSize,probSize);
        printf("------------------------------------------------------------\n");
    }
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Proc. %d - Lower bound = %d - Upper bound = %d \n",rank,lower_b,upper_b);

    // INITIALISATION OF XVECTOR
    //ParVector XVector = ParVector(MPI_COMM_WORLD,lower_b,upper_b);
    ParVector *XVector = new ParVector(MPI_COMM_WORLD,lower_b,upper_b);
    XVector->ReadExtVec();
    if(rank == 0){printf("XVector initialised and read in.\n");}

    // INITIALISATION OF YVECTOR
    //ParVector YVector = ParVector(MPI_COMM_WORLD,lower_b,upper_b);
    ParVector *YVector = new ParVector(MPI_COMM_WORLD,lower_b,upper_b);
    YVector->SetToZero();
    if(rank == 0){printf("YVector initialised and zeroed.\n");}

    // INITIALISATION OF AMATRIX
    ParMatrixSparse *AMatrix = new ParMatrixSparse(XVector,YVector);
    if(rank == 0){printf("AMatrix initialised.\n");}

    AMatrix->ReadExtMat();
    if(rank == 0){printf("AMatrix read in.\n");}

    // CONVERTING TO CSR
    AMatrix->ConvertToCSR();
    if(rank == 0){printf("AMatrix converted to CSR format.\n");}

    // Setup communications
    AMatrix->FindColsToRecv();
    if(rank == 0){printf("AMatrix: Communications mapped.\n");}

    // Setup datatypes
    AMatrix->SetupDataTypes();
    if(rank == 0){printf("AMatrix: Datatypes created.\n");}

    MPI_Barrier(MPI_COMM_WORLD);

    // Matrix - vector product
    if(rank == 0){printf("AMatrix: MatVecProd initialised.\n");}
    start = MPI_Wtime();
    for (i = 0; i < maxCount; i++){
//      if(rank == 0){printf("i = %d\n",i);}
        AMatrix->MatVecProd(XVector, YVector);
    }
    finish = MPI_Wtime(); time = finish-start;
    if(rank == 0) {
```

```
        printf("------------------------------------------------------------\n")↩
            ;
        printf("AMatrix: MatVecProd completed in %f seconds.\n",time↩
            );
        printf("------------------------------------------------------------\n")↩
            ;
    }

    MPI_Barrier(MPI_COMM_WORLD);

    delete XVector;
    delete YVector;
    delete AMatrix;

    MPI_Finalize();

    return 0;
}
```

B Matlab code

B.1 GenBandedMatVec.m

```
%% Generate Random Banded Vector
clear all; close all; clc;

N = 64*5000
M = N;
minv = -100; maxv = 100;

vector = minv + (maxv-minv).*rand(N,1);
vector = sparse(vector);

filename = ['vector' num2str(N) '_matlab.mat'];
save(filename,'vector');
filename = ['vector' num2str(N) '.vec'];
mmwrite(filename,vector,'','real',5)


%% Generate Controlled Banded Random Matrix
matrix = sparse(M,N);
% maxspread = ceil(log(N)+log(M))
maxspread = ceil(log10(N)^log10(M))

% randnum = randi(10);
randnum = 10;
for i = 1:M;
    for j = 1:randnum
        di = randi([-maxspread,maxspread]);
        dj = randi([-maxspread,maxspread]);
        posi = min(max(1,i+di),M);
        posj = min(max(1,i+dj),N);
        matrix(i,posj) = minv + (maxv-minv).*rand();
    end
end
%spy(matrix);
```

```
filename = ['matrix' num2str(N) 'band_matlab.mat'];
save(filename,'matrix');
filename = ['matrix' num2str(N) 'band.mat'];
mmwrite(filename,matrix)

return
%% Check vector and matrix


N = 8*5000;
M = 8*5000;

filename = ['vector' num2str(N) '_matlab.mat'];
load(filename);
% full(vector)

filename = ['matrix' num2str(N) 'band_matlab.mat'];
load(filename);
% full(matrix)
figure(1);
set(gca,'FontSize',22);
spy(matrix);
title('Banded - 40k x 40k');

% matrix*vector
```

B.2 GenTriBandedMatVec.m

```
%% Generate Random Tribanded Vector
clear all; close all; clc;

N = 32*5000
M = N;
minv = -100; maxv = 100;

%% Generate Controlled Tri-Banded Random Matrix
matrix = sparse(M,N);
% maxspread = ceil(log(N)+log(M))
maxspread = ceil(log10(N)^log10(M))

% randnum = randi(10);
randnum = 10;
randnum2 = 5;
displ = ceil(5*log10(N)*sqrt(N))
maxspread2 = ceil(maxspread/log10(N))
for i = 1:M;
    if(mod(i,10000)==0)
        i
    end;
    for j = 1:randnum
        di = randi([-maxspread,maxspread]);
        dj = randi([-maxspread,maxspread]);
        posj = min(max(1,i+dj),N);
        matrix(i,posj) = minv + (maxv-minv).*rand();
    end
    for j = 1:randnum2
        di = randi([-maxspread2,maxspread2]);
        dj = randi([-maxspread2,maxspread2]);
```

```
        posj = min(max(1,i+displ+dj),N);
        matrix(i,posj) = minv + (maxv-minv).*rand();
        posj = min(max(1,i-displ+dj),N);
        matrix(i,posj) = minv + (maxv-minv).*rand();
    end
end

spy(matrix);

filename = ['matrix' num2str(N) 'triband_matlab.mat'];
save(filename,'matrix');
filename = ['matrix' num2str(N) 'triband.mat'];
mmwrite(filename,matrix)

return
%% Check vector and matrix


N = 32*5000;
M = 32*5000;

filename = ['vector' num2str(N) '_matlab.mat'];
load(filename);
% full(vector)

filename = ['matrix' num2str(N) 'triband_matlab.mat'];
load(filename);
% full(matrix)
figure(1);
set(gca,'FontSize',22);
spy(matrix);
title('Tri-banded - 160k x 160k');

% matrix*vector
```