

An Introduction to Superpositional Parallelization

Kenn K. Q. Zhang*

Department of Physics,
University of Illinois at Chicago,
845 West Taylor Street,
Chicago, IL 60607, United States

Abstract

A new superposition-based parallelization is introduced. The main idea is to extract the influence of external guest processors as source terms for the local host processor. The algorithm is illustrated through a parallel model then is derived rigorously. Excellent parallel scale up is demonstrated through two-dimensional diffusion problems.

1 Introduction

The past three decades of years witnessed a monotonic growth in high performance computing on distributed computers for tackling large scale problems such as those arising from fluid flows. Domain decomposition [1, 2, 4, 3] stands out as the primary parallelization method for solving field problems. In a typical domain decomposition, the overall domain is decomposed into several subdomains consisted of many elements. Computation of each subdomain is typically conducted on each processor, and the inter-subdomain information is communicated among processors. In the present work, this approach is also called *juxtapositional parallelization*, simply because the overall domain is a juxtaposition of all subdomains. Exactly due to the same nature, juxtapositional domain decomposition is restricted to field problems, where an actual physical domain must be clearly identified. Also, the method requires sophisticated pre-processing for the processing-stage data communications. For simulation of moving boundary problems, these pre-processing level communications need to be nested inside time marching, and this makes computer programming more error-prone. One of the bottlenecks in juxtapositional parallelization is the communication overhead. An example on aeroacoustics is illustrated in [4], where

* kenn.kq.zhang@gmail.com

a significant drop in parallel scale up is demonstrated as the number of processors increases. Since data to be communicated is determined during the early stage of system construction, it is very common in the juxtapositional domain decomposition to communicate excessive number of variables/quantities (in continuous sense). If this is handled in the system solving stage, where the data is more consolidated, the number of messages can be reduced to some extent. Three extra motivations for search of a new parallelization method are as follows.

First, the existing successfully demonstrated domain decomposition is implemented with discontinuous type of numerical methods with inter-subdomain patches. In contrast, no comparable success is fully demonstrated for continuous numerical methods, which still grip a firm control in many sectors of computational continua. Second, through numerical experiments it is observed the commercial software such as “Fluent” and the open source software “OpenFoam” display unsatisfactory parallel scale up. Both software use finite volume method and patches. Third, the existing successfully demonstrated domain decomposition is mostly implemented for explicit time schemes. This is typically fine for compressible flows. However, incompressible flows and many other field equations, including Maxwell’s equations, require to solve, for instance, poisson equations. Hence, iterations are unavoidable and parallel method for explicit time schemes is hard to be extended to more general situations.

In the present work we introduce the superpositional parallelization, which needs no preprocessing stage domain decomposition. Instead, the overall domain is directly discretized into elements as serial computation; then each processor handles a subset of all elements, grouped by some rules such as by numbering of elements. Each processor stores local matrices, local vectors (hereafter ‘vectors’ excludes ‘solution vectors’), and local solution vectors. For flexibility, all data is stored randomly (and compressed). The elements operated by a particular processor may scatter in the physical space, a great flexibility over domain decomposition. The superpositional parallelization takes a more abstract approach and achieves both the simplicity and the efficiency for a broad class of problems. Plus, the technique can be applied to non-field problems.

For convenience, here let us define three length scales, l^0 , l^1 , and l^2 , for computation and communications. l^0 is defined as the order of the global number of discrete unknowns (which is assumed as the same order of global nodes). l^1 is defined as the order of the local (that is, processor-level) number of discrete unknowns. l^2 is defined as the order of the number of discrete unknowns on interfaces. A serial computation costs l^0 operations in single processor. We wish an ideal parallelization costs l^1 computation and l^2 communications. For implicit time schemes, a local computation may involve several iterations, then an inter-processor communication is conducted. Hence the cost on communications may reduce further, compared with local computation. However, the iterative structure of the

overall computing is altered so that the parallel mode may take more iterations compared with the serial counterpart.

2 A parallel model

Figure 1 shows a simple two-processor model, where elements e_0 and e_1 reside in processor 0 and element e_2 resides in processor 1. Nodes 0 and 1 belong to processor 0, node 3 belongs to processor 1, and node 2 is shared by both processors. In the mimic of serial computation, with superposition-based data structure we can construct a discrete system in the symbolic form of $Ax = b$. The more detailed structure of the discrete system $Ax = b$ is as follows

$$\begin{array}{|c|c|c|c|} \hline a_{00}^0 & a_{01}^0 & & \\ \hline a_{10}^0 & a_{11}^0 & a_{12}^0 & \\ \hline & a_{21}^0 & a_{22}^0 + a_{22}^1 & a_{23}^1 \\ \hline & & a_{32}^1 & a_{33}^1 \\ \hline \end{array} \begin{array}{l} \left\{ \begin{array}{l} x_0 \\ x_1 \\ x_2 \\ x_3 \end{array} \right\} = \left[\begin{array}{l} b_0^0 \\ b_1^0 \\ b_2^0 + b_2^1 \\ b_3^1 \end{array} \right] \end{array} \quad (1)$$

where superscripts denote processors. In Eq. (1), terms such as b_1^0 are already the consequence of element-level superposition in processor 0. For this parallel model, the superposition-based data structure are

$$A^0 = \begin{array}{|c|c|c|c|} \hline a_{00}^0 & a_{01}^0 & & \\ \hline a_{10}^0 & a_{11}^0 & a_{12}^0 & \\ \hline & a_{21}^0 & a_{22}^0 & \\ \hline & & & \\ \hline \end{array} \quad A^1 = \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{22}^1 & a_{23}^1 \\ \hline & & a_{32}^1 & a_{33}^1 \\ \hline \end{array}$$

and

$$b^0 = \begin{array}{|c|} \hline b_0^0 \\ \hline b_1^0 \\ \hline b_2^0 \\ \hline 0 \\ \hline \end{array} \quad b^1 = \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline b_2^1 \\ \hline b_3^1 \\ \hline \end{array} \quad x^0 = \begin{array}{|c|} \hline x_0 \\ \hline x_1 \\ \hline x_2 \\ \hline 0 \\ \hline \end{array} \quad x^1 = \begin{array}{|c|} \hline 0 \\ \hline 0 \\ \hline x_2 \\ \hline x_3 \\ \hline \end{array}$$

Obviously, $A = \sum_j A^j$ and $b = \sum_j b^j$, but $x \neq \sum_j x^j$. On processor 0, we can recast Eq. (1) into

$$\begin{array}{|c|c|c|c|} \hline a_{00}^0 & a_{01}^0 & & \\ \hline a_{10}^0 & a_{11}^0 & a_{12}^0 & \\ \hline & a_{21}^0 & a_{22}^0 & \\ \hline & & & \\ \hline \end{array} \begin{array}{l} \left\{ \begin{array}{l} x_0 \\ x_1 \\ x_2 \\ 0 \end{array} \right\} = \left[\begin{array}{l} b_0^0 \\ b_1^0 \\ b_2^0 \\ 0 \end{array} \right] + \left(\left[\begin{array}{l} 0 \\ 0 \\ b_2^1 \\ b_3^1 \end{array} \right] - \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{22}^1 & a_{23}^1 \\ \hline & & a_{32}^1 & a_{33}^1 \\ \hline \end{array} \right) \left\{ \begin{array}{l} 0 \\ 0 \\ x_2 \\ x_3 \end{array} \right\} \end{array} \quad (2)$$

Row 3 (the 4th row) in Eq. (2) is irrelevant to processor 0, therefore the discrete equation reduces to

$$\begin{aligned}
\begin{array}{|c|c|c|c|} \hline a_{00}^0 & a_{01}^0 & & \\ \hline a_{10}^0 & a_{11}^0 & a_{12}^0 & \\ \hline & a_{21}^0 & a_{22}^0 & \\ \hline & & & \\ \hline \end{array} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \end{pmatrix} &= \begin{pmatrix} b_0^0 \\ b_1^0 \\ b_2^0 \\ 0 \end{pmatrix} + R^0 \left(\begin{pmatrix} 0 \\ 0 \\ b_2^1 \\ b_3^1 \end{pmatrix} - \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{22}^1 & a_{23}^1 \\ \hline & & a_{32}^1 & a_{33}^1 \\ \hline \end{array} \begin{pmatrix} 0 \\ 0 \\ x_2 \\ x_3 \end{pmatrix} \right) \\
&= \begin{pmatrix} b_0^0 \\ b_1^0 \\ b_2^0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ b_2^1 \\ 0 \end{pmatrix} - \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{22}^1 & a_{23}^1 \\ \hline & & & \\ \hline \end{array} \begin{pmatrix} 0 \\ 0 \\ x_2 \\ x_3 \end{pmatrix}
\end{aligned}$$

where the filtering operator R^0 screens out irrelevant data from processor 1 to processor 0. Similarly, we have

$$\begin{aligned}
\begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{22}^1 & a_{23}^1 \\ \hline & & a_{32}^1 & a_{33}^1 \\ \hline \end{array} \begin{pmatrix} 0 \\ 0 \\ x_2 \\ x_3 \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \\ b_2^1 \\ b_3^1 \end{pmatrix} + R^1 \left(\begin{pmatrix} b_0^0 \\ b_1^0 \\ b_2^0 \\ 0 \end{pmatrix} - \begin{array}{|c|c|c|c|} \hline a_{00}^0 & a_{01}^0 & & \\ \hline a_{10}^0 & a_{11}^0 & a_{12}^0 & \\ \hline & a_{21}^0 & a_{22}^0 & \\ \hline & & & \\ \hline \end{array} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \end{pmatrix} \right) \\
&= \begin{pmatrix} 0 \\ 0 \\ b_2^1 \\ b_3^1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ b_2^0 \\ 0 \end{pmatrix} - \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & a_{21}^0 & a_{22}^0 \\ \hline & & & \\ \hline \end{array} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \end{pmatrix}
\end{aligned}$$

From this parallel model, we find that only row 2 (the third row) is correlated between two processors, which corresponds to the fact that node 2 is shared by both processors. However, node 3 which only belongs to processor 1 also plays a role in processor 0 through the source term. From this parallel model we find that the inter-processor correlation between any two processors is extremely simple: the rows (in the global sense) appear in both processors.

3 Algorithm

Let i and j denote specific processors. On processor i , we begin with the global system

$$\begin{aligned}
& Ax = b \\
\Leftrightarrow & \left(\sum_j A^j \right) x = \sum_j b^j \\
\Leftrightarrow & \sum_j (A^j x) = \sum_j b^j \\
\Leftrightarrow & \sum_j (A^j x^j) = \sum_j b^j \\
\Leftrightarrow & A^i x^i = b^i + \sum_{j \neq i} (b^j - A^j x^j) \\
\Leftrightarrow & A^i x^i = b^i + \sum_{j \neq i} [R^i (b^j - A^j x^j)] \\
\Leftrightarrow & \hat{A}^i \hat{x}^i = \hat{b}^i + \sum_{j \neq i} [R^i (\hat{b}^j - \hat{A}^j \hat{x}^j)]
\end{aligned} \tag{3}$$

where R stands for the inter-processor correlation or the filtering operator, and terms with $\hat{}$ are condensed versions of local (processor level) random data structure. A graphical illustration of Eq. (3) is already presented in the previous parallel model. The superposical parallelization is fully localized. Each processor solves a local problem, with influence of other processors coming into play through the source term. Since every processor owns a portion of the global solution vector, at some stage the solution must be homogenized. The superposical parallelization is the superset of its serial counterpart, in terms of iteration. If the local iteration is conducted only once, then it reduces the same iterative structure as its serial counterpart. The superposical parallelization can be extended to situations where the right-hand-side contains *discrete unknowns*

$$\begin{aligned}
& Ax = By \\
\Leftrightarrow & \hat{A}^i \hat{x}^i = \hat{B}^i \hat{y}^i + \sum_{j \neq i} [R^i (\hat{B}^j \hat{y}^j - \hat{A}^j \hat{x}^j)],
\end{aligned}$$

which is the case when an operator splitting technique is selected for incompressible flows.

4 Performance test

The standard diffusion problem can be written as $\frac{\partial u}{\partial t} = \nabla^2 u + f$. Figure 2 shows the configurations of diffusion problems for the performance test of superposical parallelization. The exact analytical solution to diffusion I problem is

$$u = y + 2 \sum_{n=1}^{\infty} \frac{(-1)^n}{n\pi} \sin(n\pi y) e^{-n^2 \pi^2 t}$$

and the solution to diffusion II problem is

$$u = \sin\left(\frac{\pi}{2}x\right) \sin\left(\frac{\pi}{2}y\right) \left(e^{-\frac{\pi^2}{2}t} - 1\right)$$

A relaxed Gauss-Seidel is used for linear iteration, where the increase is relaxed by 0.5. Table 1 demonstrates an excellent scale up of superposical parallelization, where the altered iterative procedure in parallel mode is monitored. Figure 3 graphically demonstrates what presented in Table 1.

5 Extension to quantum mechanics

Schrodinger's equation reads

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \nabla^2 \Psi + V\Psi$$

Let $\Psi = \phi + i\psi$, we obtain

$$\begin{aligned} \hbar \frac{\partial \phi}{\partial t} &= -\frac{\hbar^2}{2m} \nabla^2 \psi + V\psi \\ \hbar \frac{\partial \psi}{\partial t} &= \frac{\hbar^2}{2m} \nabla^2 \phi - V\phi \end{aligned}$$

These two real-value quantum equations are essentially coupled diffusion equations, and can be solved in a very similar way.

References

- [1] P. F. Fischer. *Spectral element solution of Navier-Stokes equations on high performance distributed-memory parallel processors*. PhD thesis, Massachusetts Institute of Technology, 1989.
- [2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 2nd edition, 1999.
- [3] G. B. Jacobs. *Numerical simulation of two-phase turbulent compressible flows with a multidomain spectral method*. PhD thesis, University of Illinois at Chicago, 2003.
- [4] D. Stanescu. *A multidomain spectral method for computational aeroacoustics*. PhD thesis, Concordia University, 1999.

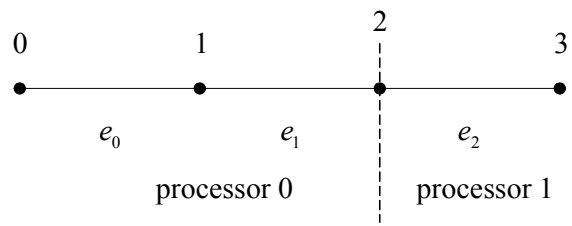


Figure 1: A two-processor parallel model.

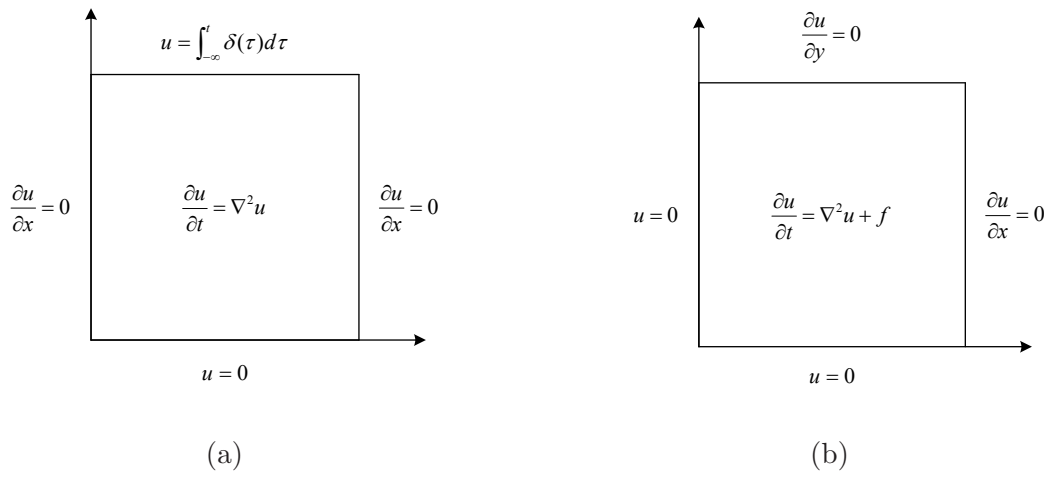


Figure 2: Configuration for the diffusion problems, (a) diffusion I problem, (b) diffusion II problem where the source $f = -\frac{\pi^2}{2} \sin\left(\frac{\pi}{2}x\right) \sin\left(\frac{\pi}{2}y\right)$. For both cases, $u = 0$ initially.

number of processors	1	2	4	8
cpu time (seconds)	511.	237.	117.	61.
speed up (versus serial)	1	2.16	4.37	8.38
number of inter-processor iterations		1062	1062	1058

(a)

number of processors	1	2	4	8
cpu time (seconds)	465.	208.	112.	56.
speed up (versus serial)	1	2.24	4.15	8.30
number of inter-processor iterations		801	801	799

(b)

Table 1: Scale up study of superposical parallelization for 2D diffusion, (a) diffusion I problem; (b) diffusion II problem.

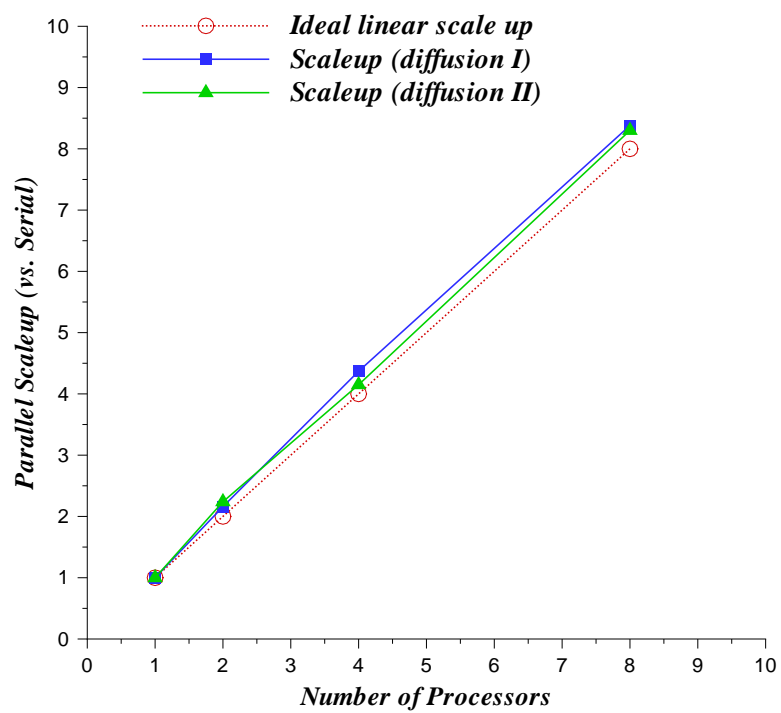


Figure 3: Demonstration of parallel scale up of superpositional parallelization.

6 Appendix: List of Subroutines/Functions for Superpositional Parallelization

6.1 *process_superpositional_MPI.cpp*

```
1 // the solver and the caller of parallel subroutines
2 #include "globalDefined.h"
3
4 #if parallel
5 #include "mpi.h"
6 #endif
7
8 #include <iostream>
9 #include <fstream>
10 #include <iomanip>
11 #include <cmath>
12 #include <ctime>
13
14 using namespace std;
15
16
17
18 // subroutines
19 int assemblerComRand_ebe ( double matrixHe [], int positionHe [], int counterHe [], double
    vectorHe [], int &rowCounter, int rowToNode [], int nodeToRow [] );
20
21 void MPI_correlation ( int rowCounter, int rowToNode [], int numDofHe, int
    counterFilteredList [], int nodeFilteredList [][MPI_MAXSIZE] );
22
23 void MPI_superpositionFiltered ( int rowCounter, int rowToNode [], int nodeToRow [],
    double vector_host [], int counterFilteredList [], int
    rowToNodeFilteredList [][MPI_MAXSIZE], double vector [] );
24
25 void MPI_homogenizationFiltered ( int rowCounter, int rowToNode [], int nodeToRow [],
    double vector_local [], int counterFilteredList [], int
    rowToNodeFilteredList [][MPI_MAXSIZE] );
26
27 void MPI_homogenization ( int rowCounter, int rowToNode [], double vector_local [], int
    length_vector_global, double vector_global [] );
28
29 void MPI_iterationTermination ( int rowCounter, double u_local [], double uOldSource [],
    int &agreement );
30
31 void utility_bubble_address ( int a_random [], int a_address [], int size );
32
33 void linearGS_comRand ( int maxIter, double omega, int rowCounter, int rowToNode [], int
    nodeToRow [], int rowAddressOrdered [], int bandwidth, double a [], int columnPosition [],
    int columnCounter [], double b [], double x [] );
```

```

34
35
36
37 // global variables
38 extern int      MaxIterLinear, MaxIterInterprocessor, NumDofHe, MPI_MaxNumRow, BandwidthHe;
39 extern double   dof_he [NUMDOFHE];
40 extern int      MPI_CounterFilteredList [MPLMAXSIZE],
41                MPI_RowToNodeFilteredList [MPLMAXNUMROW] [MPLMAXSIZE];
42
43
44
45
46 void process_superposical_MPI ()
47 {
48     int      i, j, iterInterprocessor;
49     int      agreement;
50     double   omega = 0.5;
51
52     char     *outfile;
53     outfile = "output_monitor.dat";
54
55
56
57 // begin dynamic
58 // random data retrieval
59 int      rowCounter;
60 int      *rowToNode = new int [MPI_MaxNumRow];
61 int      *nodeToRow = new int [NumDofHe];
62 int      *rowAddressOrdered = new int [MPI_MaxNumRow];
63
64 // local matrices
65 double   *matrixHe = new double [MPI_MaxNumRow*BandwidthHe];
66 int      *columnPositionHe = new int [MPI_MaxNumRow*BandwidthHe];
67 int      *columnCounterHe = new int [MPI_MaxNumRow];
68
69 // local vectors
70 double   *vectorHe = new double [MPI_MaxNumRow];
71 double   *bHe = new double [MPI_MaxNumRow];
72 double   *he = new double [MPI_MaxNumRow];
73 double   *he_old = new double [MPI_MaxNumRow];
74
75 // intermediate vectors
76 double   *residual_host = new double [MPI_MaxNumRow];
77 double   *residual_guess = new double [MPI_MaxNumRow];
78 // end dynamic
79
80
81
82 #if parallel

```

```

83     MPI::COMM_WORLD.Barrier();
84 #endif
85
86
87
88     assemblerComRand_ebe ( matrixHe, columnPositionHe, columnCounterHe, vectorHe,
89                           rowCounter, rowToNode, nodeToRow );
90
91
92 //     initialize local solution vectors
93 for ( i = 0; i < rowCounter; i++ ) {
94     he[i] = dof_he[rowToNode[i]];
95     he_old[i] = he[i];
96 }
97
98
99
100 //     index vector ordered by node
101 utility_bubble_address ( rowToNode, rowAddressOrdered, rowCounter );
102
103
104
105 //     let each other know matrix occupancy
106 MPI_correlation ( rowCounter, rowToNode, NumDofHe, MPI_CounterFilteredList,
107                 MPI_RowToNodeFilteredList );
108
109
110
111 if ( MPI_Rank == MPI_Rank_Print ) {
112     ofstream output_monitor ( outfile, ios::app);
113     output_monitor << "right_after_matrix_formation\n";
114     output_monitor.close();
115 }
116
117
118 //     part II, the interprocessor iterative procedure
119 for ( iterInterprocessor = 0; iterInterprocessor < MaxIterInterprocessor;
120     iterInterprocessor++ ) {
121
122 //         step 0: local contribution to the source term
123 for ( i = 0; i < rowCounter; i++ ) {
124     bHe[i] = vectorHe[i];
125 }
126
127
128 //         step 1: external contribution to the source term
129 for ( i = 0; i < rowCounter; i++ ) {

```

```

130         residual_host[i] = bHe[i];
131         for ( j = 0; j < columnCounterHe[i]; j++ ) {
132             residual_host[i] -= matrixHe[i*BandwidthHe+j] *
                he[nodeToRow[columnPositionHe[i*BandwidthHe+j]]];
133         }
134     }
135
136     MPI_superpositionFiltered ( rowCounter, rowToNode, nodeToRow,
        residual_host, MPI_CounterFilteredList, MPI_RowToNodeFilteredList,
        residual_guess );
137     for ( i = 0; i < rowCounter; i++ ) {
138         bHe[i] += residual_guess[i];           // not the superposition of
        host and guess residuals!!!
139     }
140
141
142
143     // step 2: linear solving
144     linearGS_comRand ( MaxIterLinear, omega, rowCounter, rowToNode, nodeToRow,
        rowAddressOrdered, BandwidthHe, matrixHe, columnPositionHe,
        columnCounterHe, bHe, he );
145
146
147
148     // step 3: interprocessor homogenization of the discrete unknowns
149     MPI_homogenizationFiltered ( rowCounter, rowToNode, nodeToRow, he,
        MPI_CounterFilteredList, MPI_RowToNodeFilteredList );
150
151
152
153     // termination of source-term iteration
154     MPI_iterationTermination ( rowCounter, he, he_old, agreement );
155     if ( agreement == 1 ) {
156         goto outsideInterprocessorIteration;
157     }
158
159
160
161     // for the next iteration
162     for ( i = 0; i < rowCounter; i++ ) {
163         he_old[i] = he[i];
164     }
165
166     } // end of interprocessor iteration
167
168
169
170
171     outsideInterprocessorIteration:
172

```

```

173
174 // communicate: global solution homogenization
175 MPI_homogenization ( rowCounter, rowToNode, he, NumDofHe, dof_he );
176
177
178
179 if ( MPI_Rank == MPI_Rank_Print ) {
180     cout << "interprocessor_iters:_" << setw(10) << iterInterprocessor <<
181         "\n";
182     ofstream output_monitor ( outfile, ios::app );
183     output_monitor << "interprocessor_iters:_" << setw(10) <<
184         iterInterprocessor << "\n";
185     output_monitor.close();
186 }
187
188 delete rowToNode;
189 delete nodeToRow;
190 delete rowAddressOrdered;
191
192 delete matrixHe;
193 delete columnPositionHe;
194 delete columnCounterHe;
195
196 delete vectorHe;
197 delete bHe;
198 delete he;
199 delete he_old;
200
201 delete residual_host;
202 delete residual_guess;
203 }

```

6.2 *MPI_correlation.cpp*

```
1 // send to all external processors the local occupancy information
2 // receive from all external processors the occupancy information
3 // make a filter for each external processor
4
5 #include "globalDefined.h"
6
7 #if parallel
8 #include "mpi.h"
9 #endif
10
11
12
13 extern int      MPI_Rank, MPI_Size;
14 extern int      MPI_SendList[MPLMAXSIZE], MPI_ReceiveList[MPLMAXSIZE];
15 extern int      MPI_CounterList[MPLMAXSIZE];
16
17 extern int      MPI_Inf[2], MPI_Inf_Buffer[2];
18 extern int      MPI_Position[MPLMAXNUMROW], MPI_Position_Buffer[MPLMAXNUMROW];
19
20
21
22 void utility_bubble_simple ( int a[], int size );
23
24
25
26 void MPI_correlation ( int rowCounter, int rowToNode[], int numDofHe, int
    counterFilteredList[], int rowToNodeFilteredList[][MPLMAXSIZE] )
27 {
28     int      i;
29     int      *flag_local_stretched = new int [numDofHe];
30
31
32
33 #if parallel
34     int      j;
35     int      tag = 0;
36     MPI::Status      status;
37     MPI::Request      requestIsend, requestIrecv;
38 #endif
39
40
41
42 // for correlation purpose
43 for ( i = 0; i < numDofHe; i++ ) {
44     flag_local_stretched[i] = 0;
45 }
46 for ( i = 0; i < rowCounter; i++ ) {
47     flag_local_stretched[rowToNode[i]] = 1;
```

```

48     }
49
50
51
52 #if parallel
53     MPI::COMMWORLD.Barrier();
54 #endif
55
56
57
58 #if parallel
59 // copy to standard fixed-length data structure for outgoing communication, and to
avoid data modification
60     MPI_Inf[0] = rowCounter;
61     MPI_CounterList[MPI_Rank] = MPI_Inf[0];
62     for ( i = 0; i < MPI_Size-1; i++ ) {
63         requestIsend = MPI::COMMWORLD.Isend ( &MPI_Inf, 1, MPI::INT,
64             MPI_SendList[i], tag );
65         requestIrecv = MPI::COMMWORLD.Irecv ( &MPI_Inf_Buffer, 1, MPI::INT,
66             MPI_ReceiveList[i], tag );
67         requestIrecv.Wait ( status );
68         MPI_CounterList[MPI_ReceiveList[i]] = MPI_Inf_Buffer[0];
69         requestIsend.Wait ( status );
70     }
71 #endif
72
73 #if parallel
74     MPI::COMMWORLD.Barrier();
75 #endif
76
77
78
79 // broadcast occupancy by all processors
80 #if parallel
81     for ( i = 0; i < rowCounter; i++ ) {
82         MPI_Position[i] = rowToNode[i];
83     }
84     for ( i = 0; i < MPI_Size-1; i++ ) {
85         requestIsend = MPI::COMMWORLD.Isend ( &MPI_Position,
86             MPI_CounterList[MPI_Rank], MPI::INT, MPI_SendList[i], tag );
87         requestIrecv = MPI::COMMWORLD.Irecv ( &MPI_Position_Buffer,
88             MPI_CounterList[MPI_ReceiveList[i]], MPI::INT, MPI_ReceiveList[i], tag
89             );
90         requestIrecv.Wait ( status );
91
92     // make a filter ASAP
93     counterFilteredList[MPI_ReceiveList[i]] = 0;
94     for ( j = 0; j < MPI_CounterList[MPI_ReceiveList[i]]; j++ ) {

```

```

92         if ( flag_local_stretched[MPI_Position_Buffer[j]] == 1 ) {
93             rowToNodeFilteredList [
94                 counterFilteredList [MPI_ReceiveList [i]]
95                 ][MPI_ReceiveList [i]] = MPI_Position_Buffer [j]; //
96                 content the global position, ordered by external
97                 processor
98                 counterFilteredList [MPI_ReceiveList [i]]++;
99             }
100     }
101
102
103
104     #if parallel
105         MPI::COMM_WORLD.Barrier ();
106     #endif
107
108
109
110     //      sorting
111     #if parallel
112         int         temp_scalar;
113         int         *temp_vector = new int [numDofHe];
114
115         for ( i = 0; i < MPI_Size-1; i++ ) {
116
117             temp_scalar = counterFilteredList [MPI_SendList [i]];
118             for ( j = 0; j < temp_scalar; j++ ) {
119                 temp_vector [j] = rowToNodeFilteredList [j][MPI_SendList [i]];
120             }
121             utility_bubble_simple ( temp_vector, temp_scalar );
122             for ( j = 0; j < temp_scalar; j++ ) {
123                 rowToNodeFilteredList [j][MPI_SendList [i]] = temp_vector [j];
124             }
125
126             temp_scalar = counterFilteredList [MPI_ReceiveList [i]];
127             for ( j = 0; j < temp_scalar; j++ ) {
128                 temp_vector [j] = rowToNodeFilteredList [j][MPI_ReceiveList [i]];
129             }
130             utility_bubble_simple ( temp_vector, temp_scalar );
131             for ( j = 0; j < temp_scalar; j++ ) {
132                 rowToNodeFilteredList [j][MPI_ReceiveList [i]] = temp_vector [j];
133             }
134         }
135
136         delete temp_vector;
137     #endif

```

138

139

140

141 *delete* *flag_local_stretched*;

142 }

6.3 MPI_homogenization.cpp

```
1 // superposition of condensed local vectors, make sure incoming local vector is not
   // modified
2
3 #include "globalDefined.h"
4
5 #if parallel
6 #include "mpi.h"
7 #endif
8
9
10
11 extern int MPI_Rank, MPI_Size;
12 extern int MPI_SendList[MPLMAXSIZE], MPI_ReceiveList[MPLMAXSIZE];
13
14 extern int MPI_CounterList[MPLMAXSIZE];
15 extern int MPI_Inf[2], MPI_Inf_Buffer[2];
16
17 extern double MPI_Vector[MPLMAXNUMROW], MPI_Vector_Buffer[MPLMAXNUMROW];
18 extern int MPI_Position[MPLMAXNUMROW], MPI_Position_Buffer[MPLMAXNUMROW];
19
20
21
22 void MPI_homogenization ( int rowCounter, int rowToNode[], double vector_local[], int
   // length_vector_global, double vector_global[] )
23 {
24     int i;
25     int *redundancyCounter = new int [length_vector_global];
26
27 #if parallel
28     int j;
29     int tag = 0;
30     MPI::Status status;
31     MPI::Request requestIsend, requestIsend2, requestIrecv, requestIrecv2;
32 #endif
33
34
35
36 // initialization
37 for ( i = 0; i < length_vector_global; i++ ) {
38     vector_global[i] = 0.0;
39     redundancyCounter[i] = 0;
40 }
41
42 // local contribution
43 for ( i = 0; i < rowCounter; i++ ) {
44     vector_global[rowToNode[i]] += vector_local[i];
45     redundancyCounter[rowToNode[i]]++;
46 }
```

```

47
48
49
50 //      copy to standard data structure for communication
51 #if parallel
52     MPI_Inf[0] = rowCounter;
53     for ( i = 0; i < rowCounter; i++ ) {
54         MPI_Vector[i] = vector_local[i];
55         MPI_Position[i] = rowToNode[i];
56     }
57 #endif
58
59
60
61 #if parallel
62     MPI::COMM_WORLD.Barrier();
63 #endif
64
65
66
67 //      broadcast row counters to all other processors by all processors
68 #if parallel
69     MPI_CounterList[MPI_Rank] = MPI_Inf[0];
70     for ( i = 0; i < MPI_Size-1; i++ ) {
71         requestIsend = MPI::COMM_WORLD.Isend ( &MPI_Inf, 1, MPI::INT,
72         MPI_SendList[i], tag );
73         requestIrecv = MPI::COMM_WORLD.Irecv ( &MPI_Inf_Buffer, 1, MPI::INT,
74         MPI_ReceiveList[i], tag );
75         requestIrecv.Wait ( status );
76         MPI_CounterList[MPI_ReceiveList[i]] = MPI_Inf_Buffer[0];
77         requestIsend.Wait ( status );
78     }
79 #endif
80
81 #if parallel
82     MPI::COMM_WORLD.Barrier();
83 #endif
84
85
86
87 //      broadcast values and positions by all processors
88 #if parallel
89     for ( i = 0; i < MPI_Size-1; i++ ) {
90         requestIsend = MPI::COMM_WORLD.Isend ( &MPI_Vector,
91         MPI_CounterList[MPI_Rank], MPI::DOUBLE, MPI_SendList[i], tag );
92         requestIsend2 = MPI::COMM_WORLD.Isend ( &MPI_Position,
93         MPI_CounterList[MPI_Rank], MPI::INT, MPI_SendList[i], tag );

```

```

92     requestIrecv = MPI::COMM_WORLD.Irecv ( &MPI_Vector_Buffer ,
          MPI_CounterList[MPI_ReceiveList[i]], MPI::DOUBLE, MPI_ReceiveList[i],
          tag );
93     requestIrecv2 = MPI::COMM_WORLD.Irecv ( &MPI_Position_Buffer ,
          MPI_CounterList[MPI_ReceiveList[i]], MPI::INT, MPI_ReceiveList[i], tag
          );
94     requestIrecv.Wait ( status );
95     requestIrecv2.Wait ( status );
96
97     for ( j = 0; j < MPI_CounterList[MPI_ReceiveList[i]]; j++ ) {
98         vector_global[MPI_Position_Buffer[j]] += MPI_Vector_Buffer[j];
99         redundancyCounter[MPI_Position_Buffer[j]]++;
100     }
101
102     requestIsend.Wait ( status );
103     requestIsend2.Wait ( status );
104 }
105 #endif
106
107
108
109 #if parallel
110     MPI::COMM_WORLD.Barrier();
111 #endif
112
113     for ( i = 0; i < length_vector_global; i++ ) {
114         vector_global[i] = vector_global[i]/redundancyCounter[i];
115     }
116
117     delete redundancyCounter;
118 }

```

6.4 *MPI_homogenizationFiltered.cpp*

```
1 #include "globalDefined.h"
2
3 #if parallel
4 #include "mpi.h"
5 #endif
6
7
8
9 extern int      MPI_Rank, MPI_Size;
10 extern int     MPI_SendList[MPI_MAXSIZE], MPI_ReceiveList[MPI_MAXSIZE];
11
12 extern int     MPI_Inf[2], MPI_Inf_Buffer[2];
13 extern double  MPI_Vector[MPLMAXNUMROW], MPI_Vector_Buffer[MPLMAXNUMROW];
14
15
16
17 void MPI_homogenizationFiltered ( int rowCounter, int rowToNode[], int nodeToRow[],
18     double vector[], int counterFilteredList[], int rowToNodeFilteredList[][MPI_MAXSIZE] )
19 {
20     int      i;
21     int      *redundancyCounter = new int [rowCounter];
22     double   *vector_superposed = new double [rowCounter];
23
24 #if parallel
25     int      j, index;
26     int      tag = 0;
27     MPI::Status      status;
28     MPI::Request     requestIsend, requestIrecv;
29 #endif
30
31
32 //      initialization
33 for ( i = 0; i < rowCounter; i++ ) {
34     vector_superposed[i] = vector[i];
35     redundancyCounter[i] = 1;
36 }
37
38
39
40 #if parallel
41     MPI::COMM_WORLD.Barrier();
42 #endif
43
44
45
46 //      broadcast filtered values and positions by all processors
47 #if parallel
```

```

48     for ( i = 0; i < MPI_Size-1; i++ ) {
49
50         for ( j = 0; j < counterFilteredList[MPI_SendList[i]]; j++ ) {
51             MPI_Vector[j] =
52                 vector[nodeToRow[rowToNodeFilteredList[j][MPI_SendList[i]]]];
53         }
54
55         if ( counterFilteredList[MPI_SendList[i]] != 0 ) {
56             requestIsend = MPI::COMM_WORLD.Isend ( &MPI_Vector,
57                 counterFilteredList[MPI_SendList[i]], MPI::DOUBLE,
58                 MPI_SendList[i], tag );
59             requestIsend.Wait ( status );
60         }
61
62         if ( counterFilteredList[MPI_ReceiveList[i]] != 0 ) {
63             requestIrecv = MPI::COMM_WORLD.Irecv ( &MPI_Vector_Buffer,
64                 counterFilteredList[MPI_ReceiveList[i]], MPI::DOUBLE,
65                 MPI_ReceiveList[i], tag );
66             requestIrecv.Wait ( status );
67
68             for ( j = 0; j < counterFilteredList[MPI_ReceiveList[i]]; j++ ) {
69                 index =
70                     nodeToRow[rowToNodeFilteredList[j][MPI_ReceiveList[i]]];
71                 vector_superposed[index] += MPI_Vector_Buffer[j];
72                 redundancyCounter[index]++;
73             }
74         }
75     }
76
77     #endif
78
79     #if parallel
80         MPI::COMM_WORLD.Barrier();
81     #endif
82
83
84     for ( i = 0; i < rowCounter; i++ ) {
85         vector[i] = vector_superposed[i]/redundancyCounter[i];
86     }
87
88     delete redundancyCounter;
89     delete vector_superposed;
90 }

```

6.5 *MPI_superpositionFiltered.cpp*

```
1 // filtered superposition of condensed local vector, make sure incoming data not modified
2
3 #include "globalDefined.h"
4
5 #if parallel
6 #include "mpi.h"
7 #endif
8
9
10
11 extern int      MPI_Rank, MPI_Size;
12 extern int      MPI_SendList[MPLMAXSIZE], MPI_ReceiveList[MPLMAXSIZE];
13
14 extern int      MPI_Inf[2], MPI_Inf_Buffer[2];
15 extern double   MPI_Vector[MPLMAXNUMROW], MPI_Vector_Buffer[MPLMAXNUMROW];
16
17
18
19 void MPI_superpositionFiltered ( int rowCounter, int rowToNode[], int nodeToRow[],
    double vector_host[], int counterFilteredList[], int
    rowToNodeFilteredList[][MPLMAXSIZE], double vector[] ) // rowToNodeFilteredList[][]
    indexed by outgoing data
20 {
21     int i;
22
23 #if parallel
24     int j;
25     int tag = 0;
26     MPI::Status status;
27     MPI::Request requestIsend, requestIrecv;
28 #endif
29
30
31
32 // initialization
33 for ( i = 0; i < rowCounter; i++ ) {
34     vector[i] = 0.0;
35 }
36
37
38
39 #if parallel
40     MPI::COMM_WORLD.Barrier();
41 #endif
42
43
44
45 // broadcast filtered values and positions by all processors
```

```

46 #if    parallel
47    for ( i = 0; i < MPI_Size-1; i++ ) {
48
49        for ( j = 0; j < counterFilteredList[MPI_SendList[i]]; j++ ) {
50            MPI_Vector[j] = vector_host[
51                nodeToRow[rowToNodeFilteredList[j][MPI_SendList[i]] ]];
52
53        }
54
55        if ( counterFilteredList[MPI_SendList[i]] != 0 ) {
56            requestIsend = MPI::COMM_WORLD.Isend ( &MPI_Vector,
57                counterFilteredList[MPI_SendList[i]], MPI::DOUBLE,
58                MPI_SendList[i], tag );
59            requestIsend.Wait ( status );
60        }
61
62        if ( counterFilteredList[MPI_ReceiveList[i]] != 0 ) {
63            requestIrecv = MPI::COMM_WORLD.Irecv ( &MPI_Vector_Buffer,
64                counterFilteredList[MPI_ReceiveList[i]], MPI::DOUBLE,
65                MPI_ReceiveList[i], tag );
66            requestIrecv.Wait ( status );
67            for ( j = 0; j < counterFilteredList[MPI_ReceiveList[i]]; j++ ) {
68                vector[
69                    nodeToRow[rowToNodeFilteredList[j][MPI_ReceiveList[i]]
70                    ] += MPI_Vector_Buffer[j];
71            }
72        }
73    }
74
75    #endif
76
77    #if parallel
78        MPI::COMM_WORLD.Barrier ();
79    #endif
80 }

```