Supplementary Material for “Blending Extensibility and Performance in Dense and Sparse Parallel Data Management”

Javier Fresno, Arturo Gonzalez-Escribano, and Diego R. Llanos, *Senior Member, IEEE*

Abstract—Dealing with both dense and sparse data in parallel environments usually leads to two different approaches: To rely on a monolithic, hard-to-modify parallel library, or to code all data management details by hand. In this paper we propose a third approach, that delivers good performance while the underlying library structure remains modular and extensible. Our solution integrates dense and sparse data management using a common interface, that also decouples data representation, partitioning, and layout from the algorithmic and parallel strategy decisions of the programmer. Our experimental results in different parallel environments show that this new approach combines the flexibility obtained when the programmer handles all the details with a performance comparable to the use of a state-of-the-art, sparse matrix parallel library.

Index Terms—Data partition, mapping techniques, sparse structures, parallel libraries.

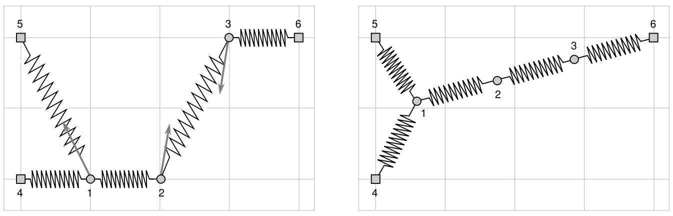


Fig. 1. Spring system example: Initial node position (left) and final equilibrium position (right).

1 HITMAP API

The basic Hitmap library API is summarized in Table 1. The new elements of the Hitmap API to support the sparse domains are summarized in Table 2.

2 FEM BENCHMARK

Our FEM benchmark calculates the equilibrium position of a 3-dimensional spring system, represented as a graph. Fig. 1 shows a 2-D example of a simple spring system. The first part of the figure contains the initial node position and the second one contains the final equilibrium position after the benchmark execution.

Systems from real civil-engineering structures usually have a high-degree of sparsity, but we may also generate more artificial systems with any degree of sparsity. The parallel algorithm parts the system into irregular pieces, and executes several computation stages which need neighbor synchronization.

The benchmark uses the Finite Element Method (FEM). The FEM method is a numerical procedure to analyze structures, finding approximate solutions for huge problems that cannot be solved by classical analytical methods, due to its complexity. The FEM method reduces the complexity of the problem by performing a discretization, using simple connected parts called finite elements. The elements are represented by interconnected nodes. There are N free nodes whose positions ($\vec{r_i}$) are arranged in a ρ vector, and M fixed nodes.

• *Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: {jfresno, arturo, diego}@infor.uva.es.*

2.1 Mathematical background

We briefly show the mathematical background of the FEM benchmark. To obtain the equations that determine the equilibrium position, we have to start from the potential energy. In this system the potential energy is composed by the strain energy of elastic distortion in each spring. The following equation calculates the potential energy. The parameters are the current positions of the free nodes.

$$V(\vec{\rho}) = \frac{1}{2}\sum_{i<j} k(l - \|\vec{r}_i - \vec{r}_j\|)^2 \qquad \begin{matrix} i\in[1,N] \\ j\in[1,N+M] \end{matrix}$$

Equilibrium position corresponds with the configurations that make the gradient of the potential energy function equal to zero, that is: $\nabla V(\vec{\rho}) = 0$.

We have selected the Newton iterative method to find the root in the previous equation. The Newton method, also called the Newton-Raphson method, requires the evaluation at arbitrary points of both the function and its derivative. The Newton method geometrically extends the tangent line at a current point x_i until it crosses zero, and then sets the next guess x_{i+1} to the abscissa of that zero-crossing. Its great advantage is that it converges quadratically [1]. The generalized version of the Newton-Raphson method for multiple dimensions needs the derivative of the gradient function, that is, the

Object	Method	Description
SigShape	SigShape(dims, [begin,end, stride]*)	Signature Shape constructor. A SigShape is defined by a selection of indexes in each of its dimensions.
Topology	Topology(plugin)	Topology constructor. This constructor creates a new topology object using the selected plugin to arrange the available processors in a virtual topology. It could be one of the Hitmap predefined plugins or a used-defined one.
Layout	Layout(plugin, topology,shape)	A Layout constructor determines the data distribution of a shape over a virtual topology. Like in the topology objects, Hitmap offers several predefined plugins.
	getShape()	This method returns the local shape assigned to the current processor.
Comm	Comm(type, TileIn,TileOut)	Creates a new Comm object that communicates data from tiles. The type parameter define which communication to be performed.
	do()	Performs the communication encapsulated by the Comm object.
Pattern	Pattern(type)	Creates a new communication pattern, which can be executed ordered or unordered.
	add(comm)	Adds a new communication to the pattern.
	do()	Performs the communications associated in the pattern.
MatrixTile	Tile(shape, datatype)	Tile constructor. It creates a tile object using the domain defined by a shape object. The datatype parameter determines the type of data of each single element.
	allocate()	Allocates the data for the tile.
	elemAt(x,y,...)	Method to access a element in the tile.

TABLE 1
Original classes and methods of the Hitmap API. Original classes that were internally redesigned without changing the API are highlighted in grey.

Object	Method	Description
SparseShape	SparseShape()	Sparse Shape constructor.
	addVertex(x)	Adds a new vertex to the structure.
	addElem(x,y), addEdge(x,y)	Equivalent methods to add an element to the matrix, or an edge to the graph.
	hasVertex(x)	Checks whether a vertex is present in the sparse structure.
	hasElem(x,y), hasEdge(x,y)	Equivalent methods to check whether an element is present in the matrix, or an edge in the graph.
GraphTile	GraphTile (Shape, datatype, vertex/edges)	GraphTile constructor. It creates a graph tile object using the domain defined by a shape object. The datatype parameter determines the type of data of each single element. A GraphTile object could store data for the vertex and/or egdes of a graph.
	allocate()	Allocates the data for the tile.
	vertexAt(x)	Method to access a vertex in the tile.
	edgeAt(x,y)	Method to access a edge in the tile. Equivalent to elemAt(x,y) of Tile class.

TABLE 2
New classes and methods of the Hitmap API to handle sparse domains.

hessian matrix of the potential energy function: $V(\vec{\rho})$. A hessian matrix is a square matrix composed by the second partial derivatives of the function. The iterative method uses successive approximations to obtain a more accurate solution, in each step it calculates:

$$\vec{\rho}_{k+1} = \vec{\rho}_k - \vec{\xi}_k$$

where $\vec{\xi}_k$ is:

$$\mathrm{Hess}\, V(\vec{\rho}_k) \cdot \vec{\xi}_k = \nabla V(\vec{\rho}_k)$$

The previous equation is a linear system that can be rewritten in the $Ax = b$ form, where the hessian matrix function evaluated at the current position is the coefficient matrix, the vector $\vec{\xi}_k$ has the unknown variables and the gradient function evaluated at the current position is the constant vector. To solve this system we have selected the Jacobi iterative method [2].

2.2 Parallel algorithm for the FEM benchmark

The parallel algorithm of the FEM benchmark performs the following stages:

1) Graph load: A sparse graph representing the structure of the spring system, and the position coordinates of the nodes, are loaded from files.
2) Node initialization: 10% of the nodes are randomly set as fixed. The coordinates of the remaining nodes (free nodes) are the variables of the benchmark.
3) Graph partition: The data partition is calculated.
4) Graph distribution: Nodes coordinates and their *free/fixed* status are sent to the appropriate processor.
5) Approximation computation: A loop performs N iterations of the Newton method to get an approximation of the equilibrium position. At each iteration, the current gradient and hessian matrix is calculated to generate a linear system which

solution determines the new approximation. This stage includes the communication of the new approximation coordinates.

6) System solution: For each iteration of the main loop, the Jacobi method performs J iterations to solve the linear system. This stage includes the communication of the intermediate solutions.
7) Result check: The final result is checked using the norm of the gradient vector to verify that the benchmark has reached the equilibrium position.

2.3 Hitmap implementation

In this section we discuss how to implement the FEM parallel algorithm using Hitmap, paying special attention to the functionalities to automatically compute the data-layout, and communicate the neighbor's vertices values.

Fig. 2 shows the main function that contains the *complete parallel code*. Line 3 calls a function that loads the global graph information and coordinates positions from files. Line 5 transparently generates a virtual topology of processors using the internal information available about the real topology. We have selected a plain topology, that does not define specific neighborhood relationships. Lines 7 to 12 correspond to the graph domain partition. In Line 7, the data-layout is generated with a Hitmap call. The layout parameters are: (a) The layout plug-in name; (b) the virtual topology of processors generated previously; and (c) the shape with the domain to distribute. The plug-in internally calls the Metis function to determine the distribution of vertices. The result is a HitLayout object. In line 9, we obtain the shape of the local part of the graph containing just the local nodes. On the following line, we use the layout to obtain an extended shape with local nodes plus neighbor nodes located at other processors. Line 12 calls a function that declares and allocates the local extended graph.

In lines 14 to 16, three HitCom objects are created to be used in the following stages. The first object (comA) is used to perform the communication that will distribute the original node values to the processor where the data has been mapped. The second object (comB) contains the information for the communication that will update the neighbor nodes values in the Newton method. The last object (comC) is created for the communication that will update the neighbor approximation values in the Jacobi method. Lines 18 and 19 invoke the starting communications previous to the first step of the algorithm.

The main loop for the Newton method starts at line 21. This loop uses an iterator defined in the library to traverse the local vertices of the graph (lines 24 to 27), calculating the gradient and hessian for the Newton method with the function *calculate_GH*. The inner loop for the Jacobi method starts at line 29. This loop traverses the free nodes to get an approximation for the linear system. Then, the new solution of the linear system is copied from the tile *new_e* to the tile *e* (line 36) and the line 37 invokes the communication to update the neighbor values of this tile. At this point, the inner loop for the Jacobi method is over. Line 40 uses an iterator to update the positions of the Node. Finally, line 45 updates the neighbor nodes position with a communication.

3 CSR AND BITMAP COMPARISON

To compare the efficiency of the different Shape implementations, we have executed the FEM benchmark using two hundred, randomly generated graphs of 1 000 vertices with different density degrees as input sets. The graphs have been generated using the method explained in Section 8.1 of the paper. The first plot at Fig. 3 shows the total execution time of the FEM benchmark in terms of density degree. The CSR implementation outperforms the Bitmap implementation for any density degree. CSR is more efficient when accessing the whole structure using iterators.

The second plot at Fig. 3 shows the memory space used by the shapes. Bitmap shapes are more efficient in storage space. In addition, their access methods are also more efficient for shape specific operations (union, intersection, adding/deleting elements), and for algorithms which do random accesses, like some graph traversing algorithms.

Figures 4 and 5 show excerpts of code with the structure of the CSR and Bitmap shapes, and the code of the iterators used to traverse them. There are two macro functions for each structure type that build the loops for the row and column iterator. The CSR iterator simply access the row indices array and the compressed column array returning the current element. The Bitmap iterator has the same interface but uses an additional inline function to locate the next active bit in the bitmap matrix.

4 CODE COMPLEXITY

To compare Hitmap code complexity with respect to the parallel manual and PETSc implementations, we present several code complexity and development-effort metrics. Fig. 6 compares the number code lines and tokens. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts and communications), internal code of the solvers, and non-essential lines (input-output, timers, etc).

For the manual, Hitmap and PETSc versions, we analyze the code containing the upper abstraction level and the solvers that implement the parallel algorithms. In the case of PETSc, they are included into the library as functions called directly at the top level. In this analysis, we skip the internal functions of Hitmap and PETSc devoted to implement data partitions, data accesses, marshaling, communications, etc.

Taking into account only essential line, our results for the matrix-vector multiplication benchmark show that the use of Hitmap library leads to a 56% reduction on the total number of code lines with respect to the

```
int i, j, vertex;
// Load the graph from the file.
HitGraphTileCSR global_graph = init_graph();
// Create the topology object.
HitTopology topo = hit_topology(plug_topPlain);
// Distribute the graph among the processors.
HitLayout lay = hit_layout(plug_layMetis, topo, hit_tileShape(global_graph));
// Get the local shape and the extended shape.
HitShapeCSR local_shape = hit_layShape(lay);
HitShapeCSR ext_shape = hit_cShapeExpand(local_shape, hit_tileShape(global_graph), 1);
// Allocate memory of all the variables.
HitGraphTileCSR local_graph = init_allocated_graph(&ext_shape);
// Create the communication objects.
HitCom comA = hit_comSparseScatter(lay, &global_graph, &local_graph, HitNode);
HitCom comB = hit_comSparseUpdate(lay, &local_graph, HitNode);
HitCom comC = hit_comSparseUpdate(lay, &e, HitVector);
// Send all the data from the root proc to the other procs.
hit_comDo( &comA );
hit_comDo( &comB );
// Main loop for the newton method
for(i=0; i<ITER1; i++){
    // Iterate trough all the vertices and
    // obtain the gradient and the hessian.
    hit_cShapeVertexIterator(vertex, local_shape){
        Node current = hit_gcTileVertexAt(local_graph, vertex);
        if (!current.fixed) calculate_GH(vertex);
    }
    // Loop for the jacobi method to solve the system.
    for(j=0; j<ITER2; j++){
        // Perform a iteration.
        hit_cShapeVertexIterator(vertex, local_shape){
            Node current = hit_gcTileVertexAt(local_graph, vertex);
            if (!current.fixed) solve_system_iter(vertex);
        }
        // Update the displacement.
        hit_gcTileCopyVertices(&e, &new_e);
        hit_comDo( &comC );
    }
    // Update the position with the final displacement.
    hit_cShapeVertexIterator(vertex, local_shape){
        Node * current = & hit_gcTileVertexAt(local_graph, vertex);
        if (!current->fixed)
            subV(current->r, current->r, hit_gcTileVertexAt(e, vertex));
    }
    hit_comDo(&comB);
}
```

Fig. 2. Complete parallel kernel code of the Hitmap implementation for the FEM benchmark.

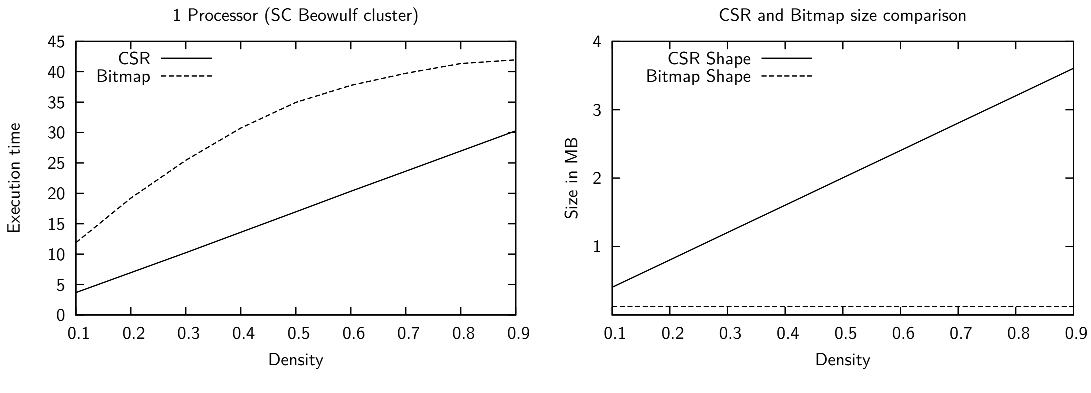


Fig. 3. CSR and Bitmap performance and size comparison for random graphs with different densities.

```c
/**
 * CSR sparse shape, implements HitShape
 */
typedef struct{
    int cards[HIT_MAXDIMS];          /**< Cardinalities       */
    idxtype * xadj;                  /**< Row indices         */
    idxtype * adjncy;                /**< Compressed columns */
    HitNameList names[HIT_MAXDIMS]; /**< Name lists          */
} HitCShape;

/**
 * CSR row iterator
 */
#define hit_cShapeRowIterator(var, shape) \
    for(var=0; var<hit_cShapeCard(shape, 0); var++)

/**
 * CSR column iterator
 */
#define hit_cShapeColumnIterator(var, shape, row)    \
    for(                                             \
        var = hit_cShapeFistColumn(shape, row);      \
        var < hit_cShapeLastColumn(shape, row);      \
        var++                                        \
    )
```

Fig. 4. CSR structure and iterators.

manual MPI version. Regarding line devoted specifically to parallelism, the percentage of reduction is up to 87%.

Taking into account the multiplication solver that is part of PETSc and that has to be programmed for the other versions, PETSc has only a reduction of 30%.

For the FEM benchmark, Hitmap has a total reduction of 41% code lines. The PETSc version needs a higher number of lines to program the Jacobi solver within the library.

Tables 3 and 4 show the McCabe's cyclomatic complexity metric for each function. As can be seen, the total cyclomatic complexity of Hitmap is less than half the value of the manual version for both benchmarks. PETSc has also a good reduction for the matrix-vector multiplication benchmark. Howevever, it increases the complexity for the FEM benchmark. As PETSc does not have a complete support for graph operations, it needs similar code than the Manual C version to generate the hessian matrix from the elements positions, to feed the solver on each iteration. The complexity increasement is caused by the implementation of the functions for the linear solvers and matrix operations.

5 Experimental Results

Table 5 shows the properties of the matrices used as examples in this paper. Fig. 7 shows a representation of the location of non-zero elements in the matrices. The first four ones are used for the matrix-vector multiplication benchmark. They have a similar high number of non-zero elements to create a big load. However, they present a large difference in the number of rows and densities, deriving in very different communication

Function	C	Hitmap	PETSc
Main	6	5	2
Matrix and vector initialization	8	6	3
Matrix and vector partition	15	-	-
Multiplication	3	1	-
Mult solver	-	-	10
Mult solver parallel	-	-	2
Mult solver computation	-	-	11
Vector redistribution	12	-	2
Vector norm	2	3	-
Other (Frees, Timers, etc)	4	2	-
Total	50	17	28

TABLE 3
Cyclomatic complexity of the multiplication benchmark.

Function	C	Hitmap	PETSc
Main	11	7	2
Init graph	6	5	6
Init structures	5	4	-
Read coordinates	11	12	14
Preallocate system matrix	-	-	8
Graph partition	44	-	-
Calculate system	6	5	12
Solve system	5	4	-
KSP solver	-	-	88
Jacobi solver	-	-	6
Mult solver	-	-	10
Mult solver parallel	-	-	2
Mult solver computation	-	-	25
Vector mult solver	-	-	13
Gradient norm	5	4	-
Scatter graph	5	-	-
All to all graph	6	-	-
Other	6	10	3
Total	110	51	189

TABLE 4
Cyclomatic complexity of the FEM benchmark.

```
/**
 * Bitmap sparse shape, implements HitShape
 */
typedef struct{
    int cards[HIT_MAXDIMS];         /**< Cardinalities. */
    int nz;                         /**< Non-zero elements. */
    HIT_BITMAP_TYPE * data;         /**< Bitmap array */
    HitNameList names[HIT_MAXDIMS]; /**< Name lists */
} HitBShape;

/**
 * Bitmap row iterator
 */
#define hit_bShapeRowIterator(var,shape) \
    for(var=0; var<hit_bShapeCard(shape, 0); var++)

/**
 * Bitmap column iterator
 */
#define hit_bShapeColumnIterator(var,shape,row)              \
    for(                                                     \
        var=hit_bShapeColumnIteratorNext(shape, -1, row);   \
        var<hit_bShapeCard(shape,1);                         \
        var=hit_bShapeColumnIteratorNext(shape, var, row)   \
    )

/**
 * Return the next column (next non-zero element of the bitmap)
 */
static inline int hit_bShapeColumnIteratorNext(HitShape shape, int var){

    size_t i;

    // 1. Index of the element and Offset of the bit in the element
    size_t xind = hit_bitmapShapeIndex(var);
    size_t xoff = hit_bitmapShapeOffset(var);

    // 2. Check if there is a 1 bit in the current element
    HIT_BITMAP_TYPE element = hit_bShapeData(shape)[xind];
    HIT_BITMAP_TYPE mask = HIT_BITMAP_1 >> xoff;
    for(i=0; i<HIT_BITMAP_SIZE-xoff; i++){
        if( (mask & element) != 0 ){
            return var + (int) i;
        }
        mask >>= 1;
    }

    // 3. Find the next element that have 1s.
    xind ++;
    while( hit_bShapeData(shape)[xind] == 0 ){
        xind++;
    }

    // 4. We do the same as 2. to get the bit location
    element = hit_bShapeData(shape)[xind];
    mask = HIT_BITMAP_1;
    for(i=0; i<HIT_BITMAP_SIZE; i++){
        if( (mask & element) != 0 ){
            return (int) (xind * HIT_BITMAP_SIZE + i);
        }
        mask >>= 1;
    }

    // Bit not found
    return -1;
}
```

Fig. 5. Bitmap structure and iterators.

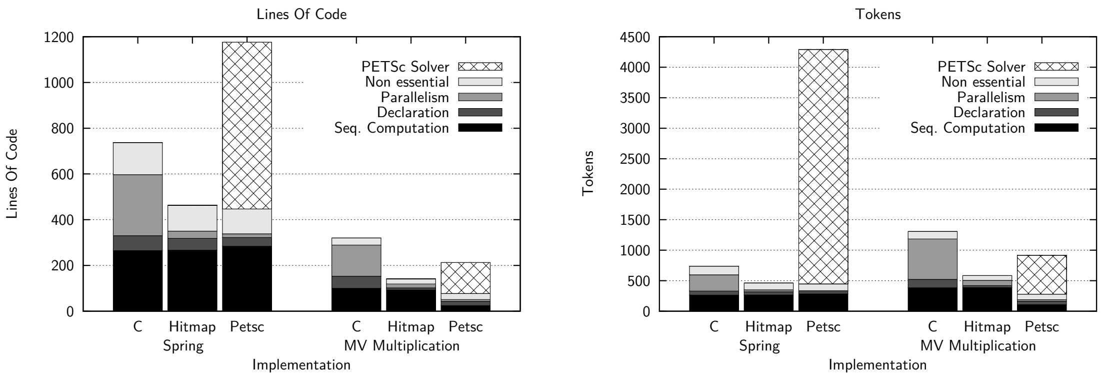
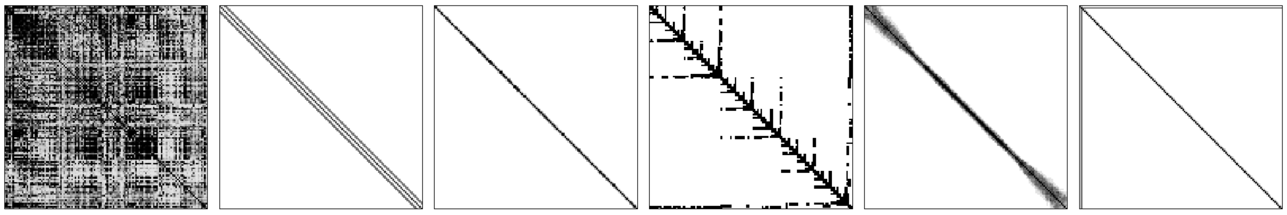


Fig. 6. Comparison of number of code lines and number of tokens.



(a) human_gene2 (b) atmosmodm (c) af_shell1 (d) pkustk14 (e) filter3D (f) lung2

Fig. 7. Example matrices structure: Representation of non-zero elements location.

Matrix/Graph	Rows/Nodes	Non-zero	d
(a) human_gene2	14,340	18,068,388	8.8×10^{-2}
(b) atmosmodm	1,489,752	10,319,760	4.6×10^{-6}
(c) af_shell1	504,855	17,562,051	6.9×10^{-5}
(d) pkustk14	151,926	14,836,504	6.4×10^{-4}
(e) filter3D	106,437	2,707,179	2.4×10^{-4}
(f) lung2	109,460	492,564	4.1×10^{-5}

TABLE 5
Characteristics of some input set examples for: (a)-(d) vector-matrix multiplication, and (e)-(f) FEM benchmark.

sizes. The second set is used for the FEM benchmark. In this case the number of rows is similar to produce similar partition sizes. We select examples with very different number of edges (non-zero elements) to produce very different computational and communication load in this benchmark.

Fig. 8 shows the execution times (not considering initialization times) obtained for the matrix-vector multiplication benchmark. In the cluster architectures, the times for all versions with one process are the same. PETSc obtains slightly better performance for small number of processors, but the scalability trend is the same for all versions when the number of processors approximates to the number of nodes. In the shared-memory architecture we observe the same results in the three versions for some example matrices (e.g. human_gene2 or atmosmodm). For other example matrices (e.g. af_shell1 or pkustk) the PETSc version leads to half the execution time of the sequential part of the code. This difference appear to be related with the PETSc storage policy and/or internal optimizations. It is not possible to determine the exact reason without a deeper analysis of the PETSc internals, as this effect appears for matrices with different structures and densities, and it does not appear in other similar ones (see Table 5 and Fig. 7). As the number of processes increases the communication times are relatively higher and the effect of the sequential optimizations have less impact.

Fig. 9 shows the execution times obtained with the FEM benchmark. The different experiments confirm the results discussed in the paper.

Finally, Fig. 10 shows the execution times obtained with the FEM benchmark for different input graph densities. The results are similar for all the three architectures. As discussed in the paper, the Hitmap implementation produces the same results than the manual implementation, obtainig better results than PETSc for growing density degrees. The efficient communications improve the results even more when the number of processes grow.

REFERENCES

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1992.
[2] G. H. Golub and C. F. Loan Van, *Matrix computations*, 3rd ed. The Johns Hopkins University Press, 1996.

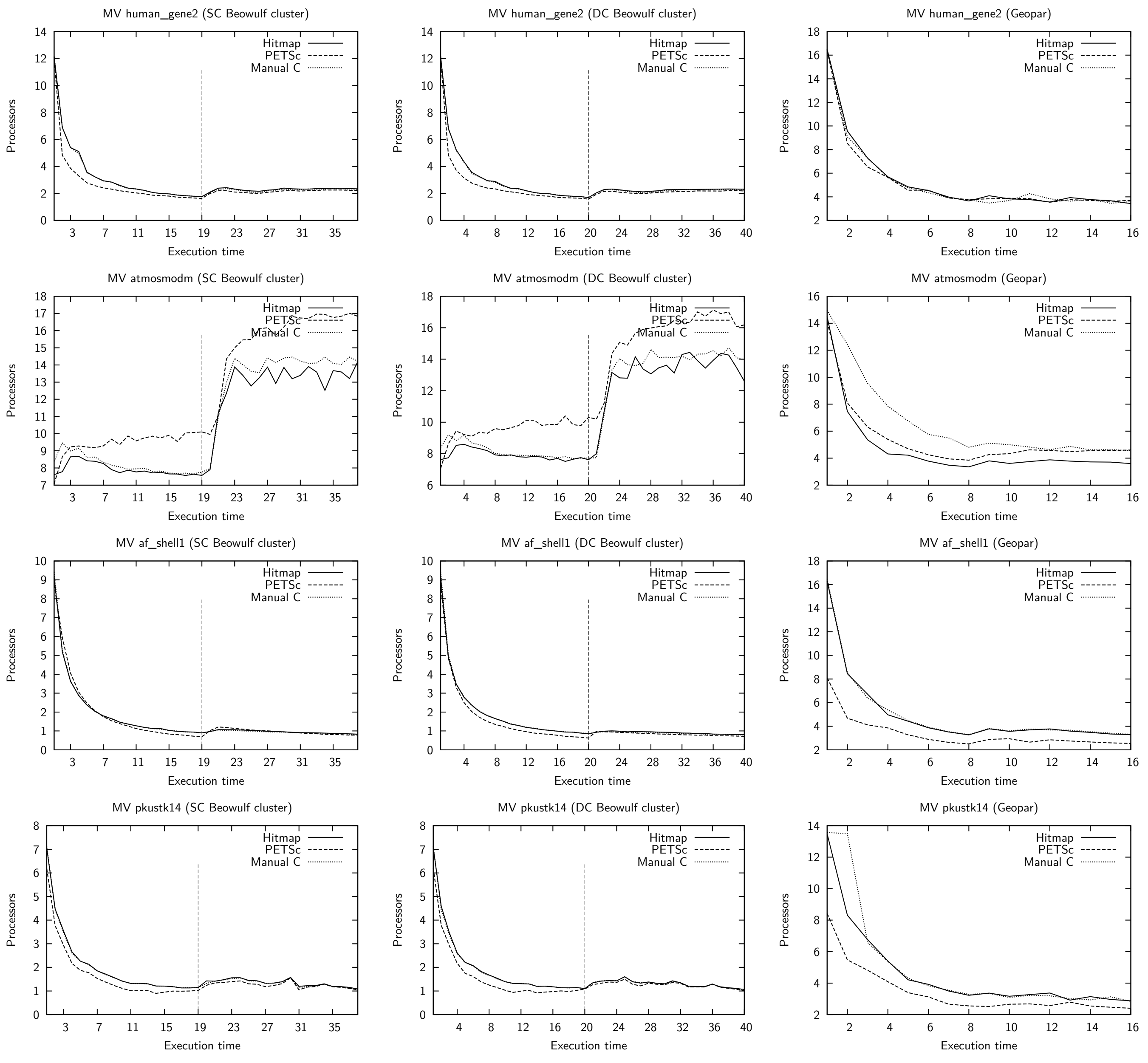


Fig. 8. Execution time comparison for Hitmap, manually developed, and PETSc implementations of the MV (matrix-vector multiplication) benchmark using the (a), (b), (c), and (d) matrix examples.

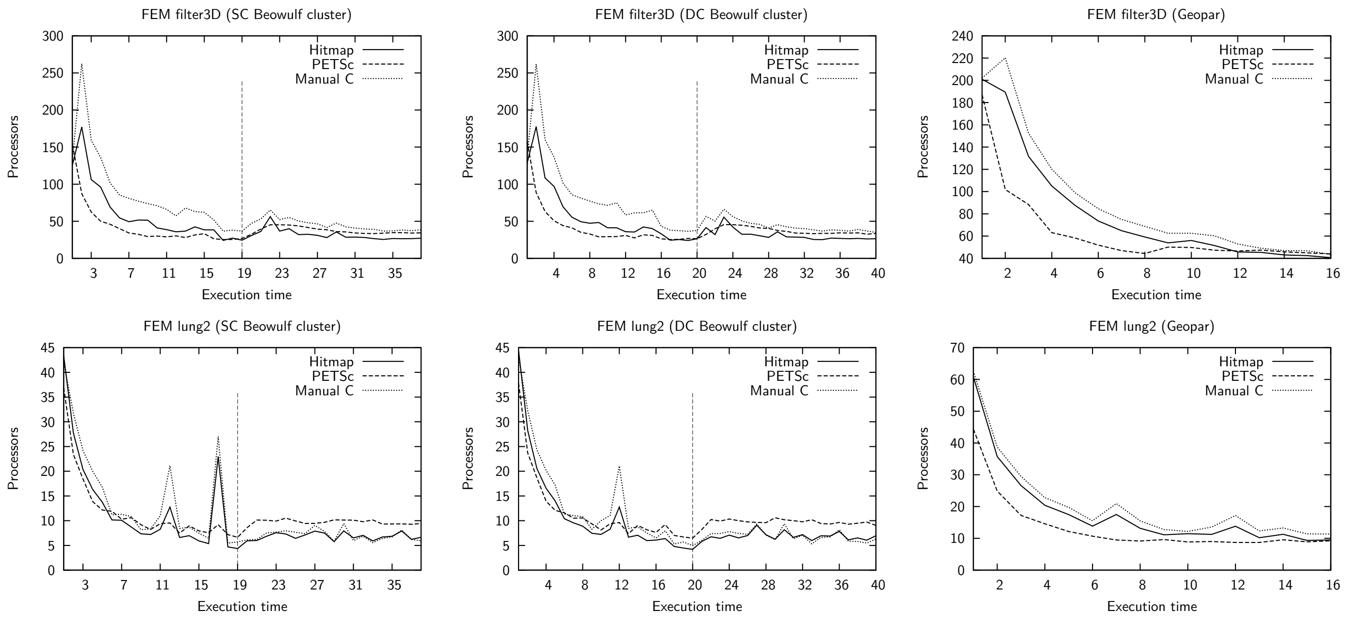


Fig. 9. Speedup comparison for Hitmap, manually developed, and PETSc implementations of the FEM (Finite Element Method) benchmark using the (e) and (f) matrix examples.

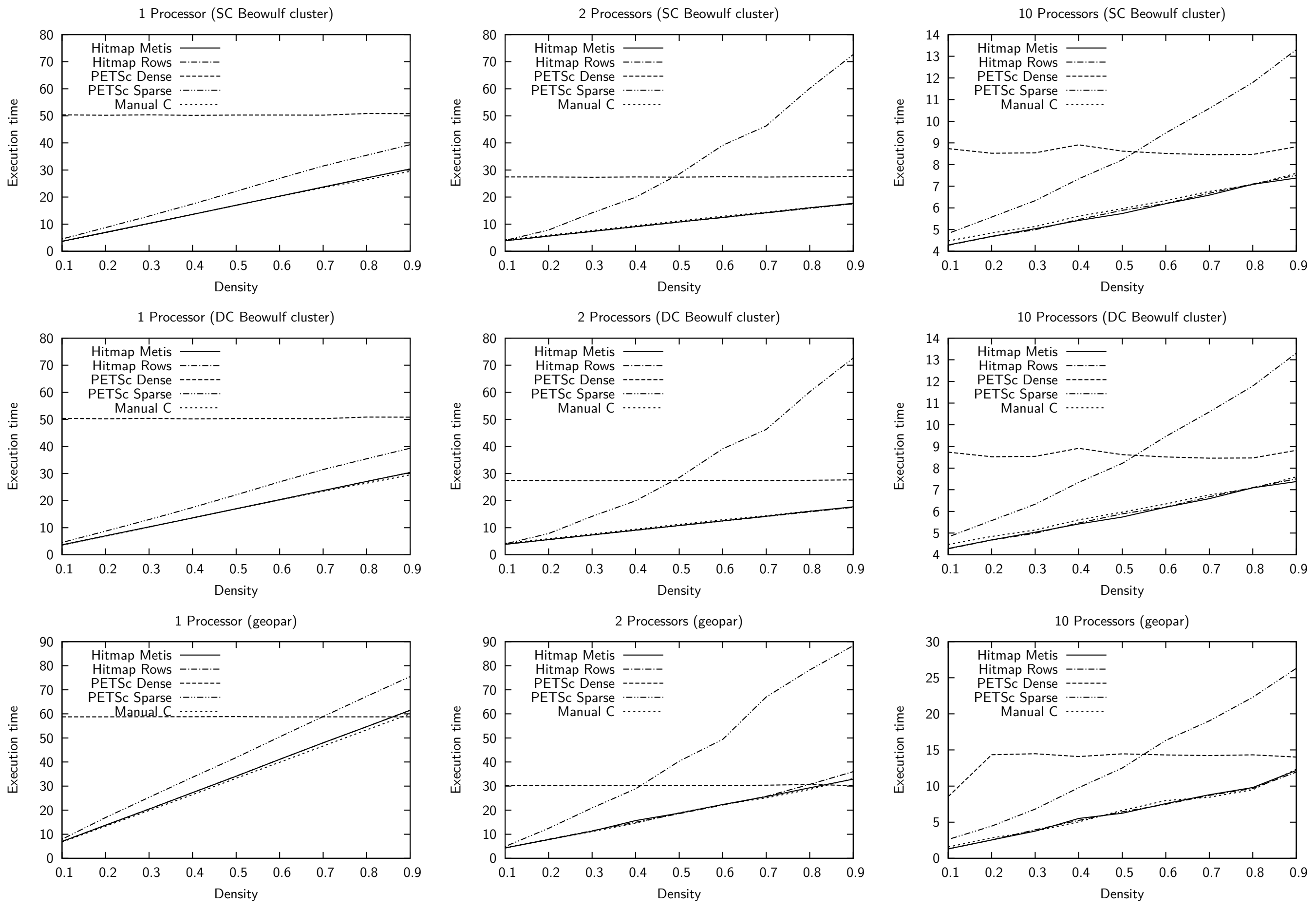


Fig. 10. Execution time comparison for Hitmap (Metis and Row partition), manually developed, and PETSc (Dense and Sparse data structures) implementations of the FEM (Finite Element Method) benchmark using random generated matrices with different densities.