Parallel Computing for Science & Engineering: Sparse matrix - vector product

Name: Pushkar Kumar Jain

Date: May 4 2014

This report presents the analysis for the sparse matrix-vector product by peroforming weak scaling study, wherein the amount of data owned per processor is constant. Storage and manipulation of data for sparse matrices require special algorithms. In the context of multiplication of a sparse matrix and a vector, it is beneficial to use the sparsity and have a greater control of the memory per processor and the reusability of data. When dealing with the numerical techniques like the finite difference method and the finite element method, we often come across some special sparse matrix structures like banded, diagonal or symmetrical. Manuipulation of data in such cases is of importance as it directly effects the execution time. Matrix vector product is one such operation that is used in many of the present iterative solvers. This report presents the analysis for effect of domain decomposition on efficiency of the sparse-matrix vector product.

1 Introduction

While solving partial/ordinary differential equations using techniques like FEM, FDM etc, one encounters the problem:

$$AX = B \tag{1}$$

$$A \rightarrow \begin{pmatrix} a_{00} & a_{01} & \cdots & \cdots & a_{0,n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & \cdots & a_{n-1,n-1} \end{pmatrix} \qquad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ .. \\ .. \\ x_{n-1} \end{pmatrix} \quad B \rightarrow \begin{pmatrix} b_0 \\ b_1 \\ .. \\ .. \\ b_{n-1} \end{pmatrix}$$

where A is a matrix, X is the unknown vector and B is the known right hand solution.

When the matrix A is sparse, certain storage techniques can considerably reduce the computation time. Unlike the dense matrix-vector product case, where each processor has to communicate with every other processor, the sparse matrix vector product is governed by minimum amount of communication of each processor. Specifically, every processor communicates with just its neighboring processors.

This fact is exploited in the area of high perofrmance linear algebra, to find the solution X iteratively in the Equation 1 . Iterative schemes employ an intiial guess value of vector X, that has to be pre-multiplied by the matrix A, giving a new vector X, for the next iteration. This is recursively performed till the difference in the solution reaches a required tolerance value.

2 Application

Multiplication of a sparse matrix A with a vector X to give a vector Y can be wriiten as

$$y \leftarrow Ax : y_i = \sum a_{ij} x_j$$

, where a, x and y are given in Equation 1. Sparsity of A implies that only a certain number of $a_{ij} \times x_j$ operations are to be carried and summed to give y_i. In case of parallel computing point of view, if the matrix A and vector X is distributed among the processors p, then, appropriate values of a_{ij} or x_j are to be communicated and the operation must be performed. Since vector memory is much less compared to the matrix memory, the matrix is usually distributed amongst the p processors. Thus, the constraint of the matrix-vector product is shown in Figure 1.

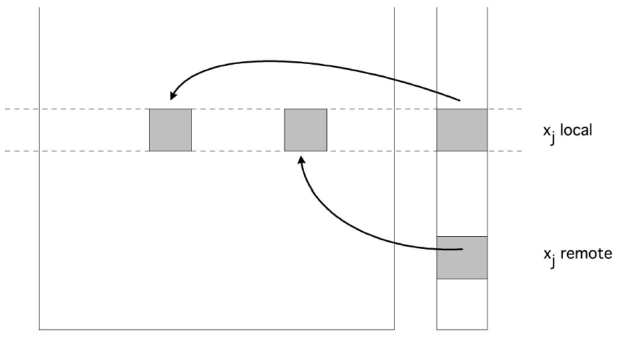


Figure 1: Matirx-vector data communication constraint (Ref [1])

While solving finite differences or finite element problem, this has a direct correlation with a given point 'interacting' with its neighboring points. The 2D poissons equation is given by

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \qquad + \qquad \textit{boundary condition} \tag{2}$$

, where u represents the variable to be solved, x and y are the spacial coordinates of the square domain and f is the forcing function. For the present analysis f is taken to be -3 and the boundary values are taken as 0. Discretization by finite differences with 5 point stencil gives the relation of the value of the point with respect to the neighboring points.

$$u_{i,j}^{k+1} = \frac{1}{4}\Big((u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - h^2 b_{i,j}\Big) \tag{3}$$

Following the above convention and expanding into the equation form gives followed by forming the matrix gives,

$$A = \begin{pmatrix} D & -I & 0 & 0 & ... \\ -I & D & -I & 0 & ... \\ 0 & -I & D & ... & ... \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & D \end{pmatrix} \quad D = \begin{pmatrix} 4 & -1 & 0 & 0 & ... \\ -1 & 4 & -1 & 0 & ... \\ 0 & -1 & 4 & ... & ... \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 4 \end{pmatrix} \quad I = \begin{pmatrix} 1 & 0 & 0 & 0 & ... \\ 0 & 1 & 0 & 0 & ... \\ 0 & 0 & 1 & ... & ... \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 1 \end{pmatrix}$$

Thus, a direct correlation of sparse matrix-vector product is given with iterated value being obtained from the neighboring points. This means that to obtain the the next iterated value of u, the above given sparse matrix A must be multiplied with the vector with all the values as -2. This is equivalent of saying that the first iteration (Equation 3 over all points) that represents operation over tne neighboring values is the matrix-vector product: Au_0 .

3 Problem definition

With the above mentioned theory and from the point of parallel computing, it is of the utmost importance the manner in which the matrix A is distributed among the processors. This sets the problem definition as the following -

The problem that is being solved is the finite difference based poisson equation over a square domain $(0, 1)$. The domain is given by the size $n \times n$ and the number of processors is given by p.

Weak scaling of the sparse matrix-vector product that is governed by the decomposition of the domain is studied based on two cases -

1. Row wise decomposition into $n \times (n/p)$ slabs.
2. Block wise decomposition into $(n/\sqrt{p}) \times (n/\sqrt{p})$ patches as shown in Figure 2.

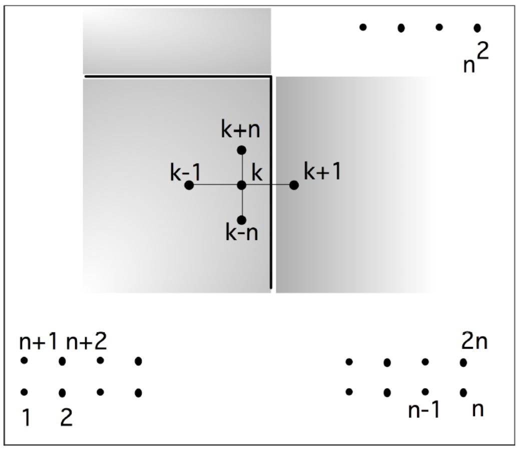


Figure 2: Block wise decomposition (Ref [1])

With n^2/p as constant, it is to be proved that

- Case 1 does not scale weakly : efficiency goes down with speed up still going up as $\sqrt{p}$.

- Case 2 does scale weakly.

In both the cases, the key to the problem is the computation versus communication at the boundaries. The communication between the processors is given in the following section which further states the algorithm.

4 Algorithm

As mentioned, it is assumed that the initial guess value for the iteration is the vector given with its elements as -2. It is to be noted that since the objective is to analyze the matrix-vector product, the poissons equation is actually not being solved by the conventional rule of converging towards a value. Rather a fixed amount of iterations are performed so that speed up and efficiency can be measured iteration wise. Also, even though the physical domain is given in 2D, the problem is dealt with the treating it as a single dimensional array. It has been appropriately flattened by making the size of the rows and columns as function with variables - number of processors and size of the matrix.

4.1 Row decomposition

The partition of domain is given by -

$$A \rightarrow \begin{pmatrix} A_0 \\ A_1 \\ .. \\ .. \\ A_{p-1} \end{pmatrix} \qquad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ .. \\ .. \\ x_{p-1} \end{pmatrix} \qquad y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ .. \\ .. \\ y_{p-1} \end{pmatrix},$$

where A_i represents the block of rows of size $n \times (n/p)$.

The computation is characterized by the fact that k^{th} processor needs to exchange data with the at most two processors: $(k-1)^{th}$ and the $(k+1)^{th}$ processor. Thus in addition to the memory size of $(n/p) \times n$, a buffer space of size $n \times 1$ is given to each processor that can be visualized being above and below the physical domain. This buffer space is the area which communicates the data between the processors. The following algorithm is followed -

1. Input the number of nodes n on the square domain
2. *MPI_Bcast(...)* n to p processors
3. Allocate memory of size $n \times (n/p)$ that holds u^k.
4. Allocate memory of size $n \times (n/p)$ that holds the dummy value u^{k+1}.
5. Allocate top buffer and bottom buffer of size $1 \times n$
6. Set boundary condition
7. Initialize the A_k matrix
8. For user iterations perform -
    a) if (first processor): No data communication to top buffer of p_1

b) else if (last processor): No data exchange to bottom buffer of p_n

c) else: Send last row of A_{k-1} to top buffer of p_k and first row of $A_{k+1}$to bottom buffer of p_k using MPI_SendRecv(...)

d) See the left neighbor value (in processor p_k itself), right neighbor value (in processor p_k itself), top neighbor value (in top buffer) and bottom neighbor value (in bottom buffer)

e) Compute $u^{(k+1)}$ using Equation 3 and store it in dummy memory

f) Copy u^{k+1} to u^k

9. Display time

10. *MPI_ Gather(...)* and display solution (optional)

4.2 Block decomposition

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \dots & \dots & A_{0,p-1} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ A_{p-1,0} & A_{p-1,1} & \dots & \dots & A_{p-1,p-1} \end{pmatrix} \qquad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ .. \\ .. \\ x_{p-1} \end{pmatrix} \qquad y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ .. \\ .. \\ y_{p-1} \end{pmatrix}$$

where A_{ij} represents the block of sub-matrix of size $n/\sqrt{p} \times n/\sqrt{p}$. Further note that A_{ij} is p_{ij}. The computation is characterized by the fact that k^{th} processor needs to exchange data with the at most four neighbor processors. Thus in addition to the memory size of $(n/\sqrt{p}) \times (n/\sqrt{p})$, a buffer space of size $1 \times n/\sqrt{p}$ is given to each processor that can be visualized each being above, below, left and right to the physical domain. This buffer space is the area which communicates the data between the processors. The following algorithm is followed -

1. Input the number of nodes n on the square domain

2. *MPI_Bcast(...)* n to p processors

3. Allocate *memory* of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ that holds u^k.

4. Allocate memory of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ that holds the dummy value u^{k+1}.

5. Allocate top buffer, left buffer, right buffer and bottom buffer each of size $(1 \times (n/\sqrt{p})$

6. Initialize the A_{ij} matrix

7. Set boundary condition

8. For user iterations perform -

   a) if (first processor): No data communication to top buffer of p_1

   b) else if (last processor): No data exchange to bottom buffer of p_n

   c) else: Send last row of A_{k-1} to top buffer of p_k and first row of $A_{k+1}$to bottom buffer of p_k using MPI_SendRecv(...)

   d) See the left neighbor value (in processor p_k itself), right neighbor value (in processor p_k itself), top neighbor value (in top buffer) and bottom neighbor value (in bottom buffer)

   e) Compute $u^{(k+1)}$ using Eq 3 and store it in dummy memory

   f) Copy u^{k+1} to u^k

9. Display time

10. *MPI_ Gather(...)* and display solution (optional)

5 Parallel Code

Based on the algorithm in Section 4, this section provides the important snippets of codes that involve the MPI calls. Two parallel programs by the name - *row.c* and block.c are programmed that perform row decomposition and block decomposition respectively. It is compiled by the commands *make row* and *make block* respectively. The serial code can be run by *specifying* number of processors as 1.

- Broadcasting number of nodes `n_node` to p processors -
  - *MPI_ Bcast(&n_ node, 1,MPI_ LONG_ LONG_ INT, 0, comm);*
- Allocation of memory u (to ghost regions also) -
  - *float *u, *new_ val, *top_ buffer, *bot_ buffer, *left_ buffer, *right_ buffer;*
  - *u = malloc((rpb * cpb) * sizeof(float));*
  - *new_ val = malloc((rpb * cpb) * sizeof(float));*
  - *top_ buffer = malloc(cpb * sizeof(float));*
  - *bot_ buffer = malloc(cpb * sizeof(float));*
  - *left_ buffer = malloc(rpb * sizeof(float));*
  - *right_ buffer = malloc(rpb * sizeof(float));*
- Sending data from u to buffer values. rpb and cpb represent the rows per blocks and columns per blocks respectively. For row decomposition $rpb = n/p$, $cpb = n$ and for block decomposition, $rpb = cpb = n/\sqrt{p}$. Each is a function that calls MPI_Sendrecv with appropriate source and destination.
  - *send_ to_ left(&u[0], &left_ buffer[0], rpb, cpb, nprocs, id, comm);*
  - *send_ to_ right(&u[0], &right_ buffer[0], rpb, cpb, nprocs, id, comm);*
  - *send_ to_ bot(&u[0], &bot_ buffer[0], rpb, cpb, nprocs, id, comm);*
  - *send_ to_ top(&u[0], &top_ buffer[0], rpb, cpb, nprocs, id, comm);*
- Compute the iteration by using Eq 3 and store in `new_val`-
  - *check_ neighbor(&u[0], &top_ buffer[0], &bot_ buffer[0], &right_ buffer[0], &left_ buffer[0], cpb, nprocs, id, rpb, &new_ val[0], f, h);*
- Update u^{k+1} to u^k -

– *update_u(&u[0], &new_val[0], rpb, cpb);*

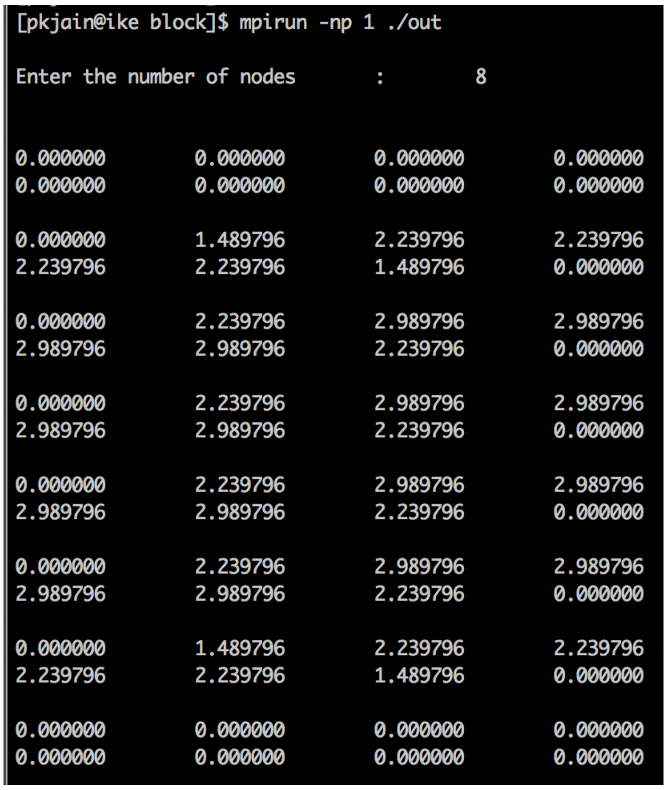
A typical communication to buffer looks like -

- *MPI_Sendrecv(&u[0], cpb, MPI_FLOAT, dest, 0, &bot_buffer[0], cpb, MPI_FLOAT, src, 0, comm, MPI_STATUS_IGNORE);*

6 Results

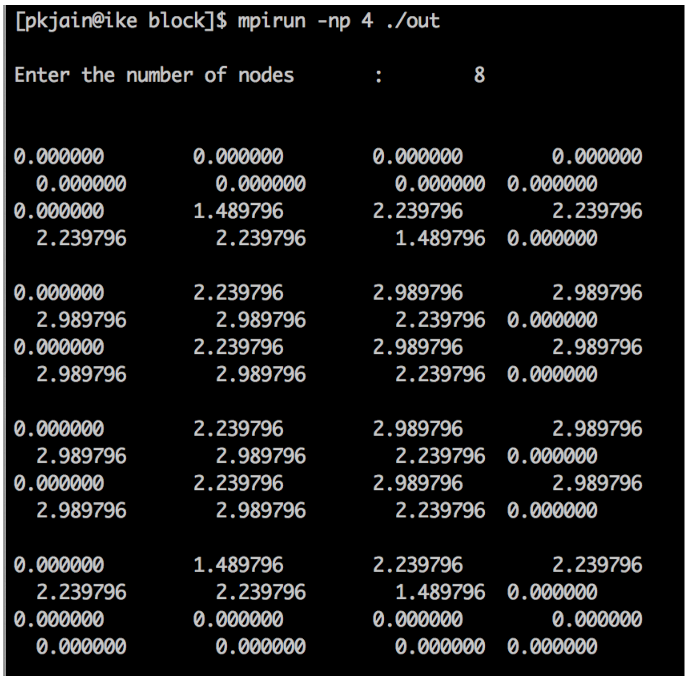
6.1 Correct execution of code

The serial and parallel version of matrix-vector code (both row decomposition and block decomposition) is performed. The results are in Figures 3, 4 and 5 . Note that the results are same. The manner in which it is being printed is different. The code is executed for 4 processors and side the square has 8 nodal points. The result represents the first iteration or equivalently, one matrix-vector product.



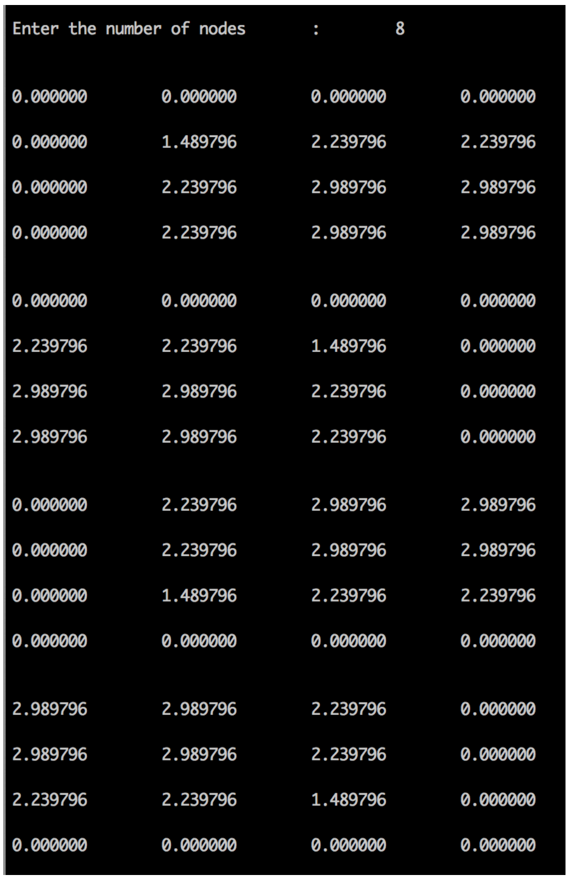
```
[pkjain@ike block]$ mpirun -np 1 ./out

Enter the number of nodes       :       8


0.000000        0.000000        0.000000        0.000000
0.000000        0.000000        0.000000        0.000000

0.000000        1.489796        2.239796        2.239796
2.239796        2.239796        1.489796        0.000000

0.000000        2.239796        2.989796        2.989796
2.989796        2.989796        2.239796        0.000000

0.000000        2.239796        2.989796        2.989796
2.989796        2.989796        2.239796        0.000000

0.000000        2.239796        2.989796        2.989796
2.989796        2.989796        2.239796        0.000000

0.000000        2.239796        2.989796        2.989796
2.989796        2.989796        2.239796        0.000000

0.000000        1.489796        2.239796        2.239796
2.239796        2.239796        1.489796        0.000000

0.000000        0.000000        0.000000        0.000000
0.000000        0.000000        0.000000        0.000000
```

Figure 3: Serial execution



```
[pkjain@ike block]$ mpirun -np 4 ./out

Enter the number of nodes       :       8


0.000000        0.000000        0.000000        0.000000
  0.000000        0.000000        0.000000  0.000000
0.000000        1.489796        2.239796        2.239796
  2.239796        2.239796        1.489796  0.000000

0.000000        2.239796        2.989796        2.989796
  2.989796        2.989796        2.239796  0.000000
0.000000        2.239796        2.989796        2.989796
  2.989796        2.989796        2.239796  0.000000

0.000000        2.239796        2.989796        2.989796
  2.989796        2.989796        2.239796  0.000000
0.000000        2.239796        2.989796        2.989796
  2.989796        2.989796        2.239796  0.000000

0.000000        1.489796        2.239796        2.239796
  2.239796        2.239796        1.489796  0.000000
0.000000        0.000000        0.000000        0.000000
  0.000000        0.000000        0.000000  0.000000
```

Figure 4: Parallel execution with row decomposition



```
Enter the number of nodes       :       8


0.000000        0.000000        0.000000        0.000000

0.000000        1.489796        2.239796        2.239796

0.000000        2.239796        2.989796        2.989796

0.000000        2.239796        2.989796        2.989796


0.000000        0.000000        0.000000        0.000000

2.239796        2.239796        1.489796        0.000000

2.989796        2.989796        2.239796        0.000000

2.989796        2.989796        2.239796        0.000000


0.000000        2.239796        2.989796        2.989796

0.000000        2.239796        2.989796        2.989796

0.000000        1.489796        2.239796        2.239796

0.000000        0.000000        0.000000        0.000000


2.989796        2.989796        2.239796        0.000000

2.989796        2.989796        2.239796        0.000000

2.239796        2.239796        1.489796        0.000000

0.000000        0.000000        0.000000        0.000000
```

Figure 5: Parallel execution with patch decomposition

6.2 Limitation of domain size

During the execution of the code, it was found that execution for both types of decomposition, is limited to approximately 50000 nodal points or 25×10^8 size for every processor. Around this value, the spawning of parallel processes was giving error. Below this, there was right execution. Moreover, the same was found for serial execution of the code. This initially was thought as a problem due to the limited range of *int* datatype. This was tried to be resolved by using long long int. But it did not help. This made to conclude that memory per processor might be the issue. The amount of data held by every processor can be approximately calculated as $2 \times 25 \times 10^8 \times sizeof(long) \rightarrow 200Gb$. Note that it is multiplied by 2 to accommodate the dummy variable new_val of equivalent size. Thus, this might be the possible reason which limits the computation.

It is to be noted that this limitation in size is for single processor. Therefore the parallel code can solve much larger domain with ease. But for the same size, it is not possible to calculate for serial execution and hence speed ups cannot be calculated. Hence, further analysis is performed keeping in mind, the limitation of the size.

6.3 Scaling analysis

Weak analysis is performed by considering $n \times n/p$ as constant. With the constant set as 50, execution using a single iteration does not give dependable results because the execution time is extremely low to the order of 10^{-5}. Thus, the iterations are set to 20 for good estimate for time comparison.

Analysis is done for number of iterations is 20 with $n \times n/p = 422500$. This value is taken so that good amount of data points is obtained such that both p and $\sqrt{p}$ divide n. This is an essential criteria governed by the square domain. The result for row patch wise decomposition is given in Table 1 and Figure 6 gives a good estimate of the trend. The same is provided for row wise decomposition in Table 2 and Figure 7.

The following can be concluded from the values -

1. The efficiency in some cases are greater than one. This is due to the fact that for such a large n value, a lot of time is being spent by the processor to perform computation such that communication time is over-shadowed.

2. It can be seen from the plot that in case of patch wise distribution, the efficiency is high in addition to the fact that it is almost constant with value varying between 1.0 and 1.2. This is not the case for row wise distribution where we can see the efficiency dropping gradually.

3. As mentioned in section 6.2, due to limitation of the domain size, even though parallel code can be run for $n > 50000$, the same is not possible for serial execution. Hence, the entire behavior of dropping efficiency in case of row wise decomposition is not captured. Theoretically, it must decrease asymptotically.

4. Thus, for row wise decomposition, even though the speed up increases, the efficiency actually decreases. This is not the case for block wise decomposition. Therefore, patch wise decomposition for sparse matrix vector product is desirable.

Table 1: Speed up and efficiency for block domain decomposition for $n \times n/p = 422500$ and 20 iterations

n	p	Time (Serial)	Time (Block)	Speed up (Block)	Efficiency (Block)
650	4	1.292	0.2608	4.953845	1.238461
1300	25	8.061	0.2678	30.0942	1.203771
3250	100	32.240	0.2740	117.6300	1.176300
6500	169	54.422	0.3105	175.2376	1.03690
8450	625	202.06	0.3112	649.2837	1.038854
16900	676	217.70	0.274	791.6835	1.171129
32500	2500	807.727	0.3020	2674.583	1.069833

Table 2: Speed up and efficiency for row domain decomposition for $n \times n/p = 422500$ and 20 iterations

n	p	Time (Serial)	Time (Row)	Speed up (Row)	Efficiency (Row)
650	4	1.292	0.3216	4.017	1.004323
1300	25	8.061	0.3325	24.2449	0.969796
3250	100	32.240	0.3328	96.8599	0.968599
6500	169	54.422	0.3415	159.323	0.942742
8450	625	202.06	0.3426	589.763	0.943620
16900	676	217.70	0.3459	629.218	0.930796
32500	2500	807.727	0.3623	2229.19	0.891676

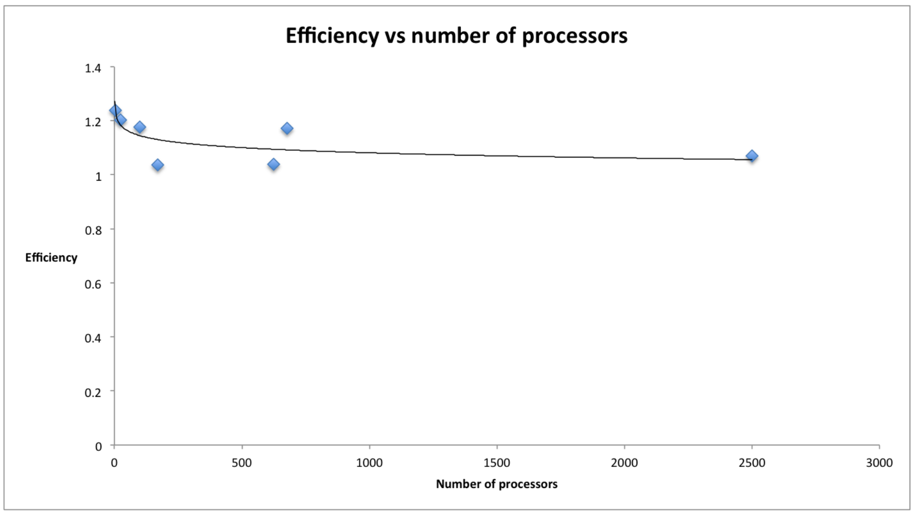


Figure 6: Efficiency vs Number of processors for block decomposition of $n \times n/p = 422500$

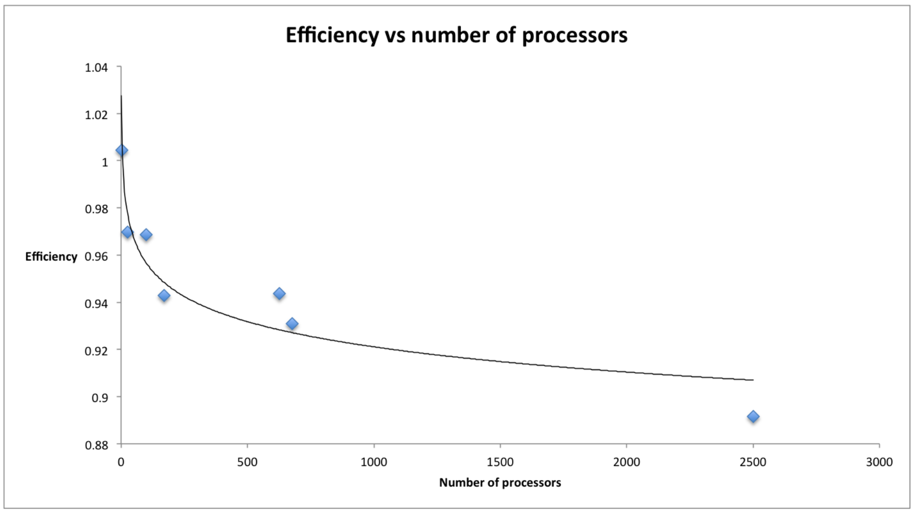


Figure 7: Efficiency vs Number of processors for block decomposition of $n \times n/p = 422500$

An attempt was made to capture the behavior, by juggling with the n and p values. The following result (Table 3) was obtained for $n \times n/p = 400000$ and 20 iterations again. This time it was possible to find that efficiency went down to 84%.

Table 3: Speed up and efficiency for row domain decomposition for $n \times n/p = 400000$ and 20 iterations

n	p	Time (Serial)	Time (Row)	Speed up (Row)	Efficiency (Row)
2000	10	3.055589	0.311599	9.806157	0.980615
4000	40	12.21287	0.314525	38.8295	0.97071
8000	160	48.829533	0.322256	151.5240	0.947025
10000	250	76.376846	0.321507	237.5588	0.950235
16000	640	195.279007	0.322394	605.7153	0.946430
20000	1000	304.857346	0.329782	924.4208	0.924420
40000	4000	1221.684589	0.361443	3380.020	0.845005

7 Conclusion

The analysis for sparse matrix-vector product is performed based on the type of domain decomposition - patch wise and row wise.The limitation of size of data per processor is an important criteria that does not allow the serial execution of the same code. It has been found that patch wise decomposition gives better results in terms of speed up and efficiency. Further, with amount of data per processor held constant (weak scaling). the efficiency drops for row wise decomposition even though the speed up rises. Thus block wise decomposition scales weakly, whereas row wise decomposition does not scale weakly.

8 References

[1] Introduction to High Performance Scientific Computing, Evolving Copy, Victor Eijkhout with Edmond Chow, Robert van de Geijn