A General Sparse Sparse Linear System Solver and Its Application in OpenFOAM

Murat Manguoglu*

Middle East Technical University, Department of Computer Engineering, 06800 Ankara, Turkey

Abstract

Solution of large sparse linear systems is frequently the most time consuming operation in computational fluid dynamics simulations. Improving the scalability of this operation is likely to have significant impact on the overall scalability of application. In this white paper we show scalability results up to a thousand cores for a new algorithm devised to solve large sparse linear systems. We have also compared pure MPI vs. MPI-OpenMP hybrid implementation of the same algorithm.

1. Introduction

The solution of sparse linear systems is among the most time consuming operations in many CFD codes, including OpenFOAM. This is more pronounced when the mesh is fine (i.e. the coefficient matrix is large). With the introduction of multicore processors and emergence of large-scale clusters in which a single node contains many cores, it is inevitable to come up with new algorithms that are aware of the memory and cache hierarchies and communication is mostly limited between the neighboring nodes. Classical preconditioned iterative solvers, although scalable are known to be not robust while direct solvers are robust yet they only provide limited scalability [1,2]. Some examples of well-known direct solver implementations are Pardiso [3], MUMPS [4,5], WSMP [6,7] and SuperLU [8] that are all based on the Gaussian elimination. There are many classical preconditioned iterative solvers, two main groups of iterative solvers are (i) Krylov subspace techniques and (ii) preconditioned Richardson iterations. Iterative solvers are usually not as robust as direct solvers even if they are used with the recent improved variations of the incomplete LU or approximate inverse based preconditioners [9,10,11,12].

We have recently developed a general sparse solver based DS factorization as opposed to LU factorization. The solver can be either used as a direct solver or as a solver for a preconditioned linear system where the preconditioner is obtained by ignoring small elements in absolute value. DS factorization goes back to 1970s where the banded DS factorization (or the Spike algorithm) was introduced in [13,14,15,16]. Later, the banded DS factorization is used as a solver for the banded preconditioned linear system [2,17,18]. More recently, a generalized DS factorization with incomplete LU factorization based solvers for the diagonal blocks for CFD problems is introduced [19] and the algorithm was combined with direct solvers for the diagonal blocks to obtain a much more robust variation applied to a variety of problems [20]. It was also applied to fluid-structure interaction problems in [21] and its relation to sparse graphs was studied in [22]. In all of these, an inner and outer iterative scheme exists. The local solves are done in parallel using either a direct solver or incomplete LU factorization but it is possible to use sparse approximate inverse or other types of approximate solvers as well. This white paper is a contribution from work package 7.5.
2. Algorithm

The algorithm used (see [19,20,21,22,23] for more details) in this white paper is based on sparse DS factorization and called domain-decomposing parallel sparse solver (DDPS). Given a sparse linear systems of equations

\[ Ax = f \]  

(1)

and number of partitions \( p \), we partition \( A \in \mathbb{R}^{n \times n} \) into \( p \) block rows \( A = [A_1, A_2, \ldots, A_p]^T \). Let

\[ A = D + R, \]  

(2)

where \( D \) consists of \( p \) block diagonals of \( A \),

\[
D = \begin{pmatrix}
A_{11} & & \\
& A_{22} & \\
& & \ddots \\
& & & A_{pp}
\end{pmatrix}.
\]  

(3)

Premultiplying both sides of (1) by \( D^{-1} \) from left

\[ D^{-1}Ax = D^{-1}f, \]  

(4)

we obtain a modified system

\[ Sx = g. \]  

(5)

Let us call the index of the unknowns, \( c \), corresponding to the nonzero columns of the matrix \( G = S - I \). Then, from Equation (5) the following independent and smaller reduced system can be extracted

\[ \hat{S}\hat{x} = \hat{g}, \]  

(6)

where \( \hat{S} = S(c,c) \), \( \hat{x} = x(c) \), and \( \hat{g} = g(c) \). After solving the reduced system for unknowns \( \hat{x} \) overall solution vector can be obtained

\[ x = g - \hat{G}\hat{x}, \]  

(7)

in which \( \hat{G} \) is a matrix that consists of only the nonzero columns of \( G \). The algorithm can be used as a solver for a preconditioned linear system with an outer iterative scheme where the preconditioning matrix is obtained from the coefficient matrix in Equation (1) by dropping small off-diagonal entries. The small reduced system in Equation (6) can also be solved employing an iterative scheme. This gives rise to inner-outer iterative method in which DS factorization is used as a solver for the preconditioned linear systems at each iteration. Comparison of DDPS algorithm with the well-known Direct and Iterative solvers (using AMG and ILU based preconditioners) is given in [19,20,21,22,23]. The DDPS solver has several advantages compared to traditional direct and iterative methods. Perhaps the most significant one is that it is a hybrid solver that combines direct and iterative solvers. Hence, it is as scalable as most iterative solvers and as robust as a direct solver. The solver is not publicly available currently but planned to be made available under GPL license under the homepage of the author of this white paper.

OpenFOAM comes with the variety of preconditioners: Diagonal Incomplete Cholesky (DIC), Faster Diagonal Incomplete Cholesky (FDIC), Diagonal Incomplete LU (DILU), diagonal, and Geometric Algebraic Multigrid (GAMG). We note that the DIC, FDIC, and DILU are block Jacobi type preconditioners (diagonal preconditioner is the Jacobi preconditioner) and they do not consider the elements outside the off-diagonal blocks. Geometric Multigrid preconditioners are known to be fast. However, since they require some information about the geometry they are not applicable to all problems. Hence, Algebraic Multigrid (AMG) preconditioners are developed to alleviate this weakness of Geometric Multigrid preconditioners. Algebraic Multigrid preconditioners, however, lack strong scalability [24].
3. Test problem and computing platform

The Lid Driven Cavity Flow is most probably one of the most studied CFD problems. Due to the simplicity of the cavity geometry, applying a numerical method on it is quite easy in terms of coding. Nonetheless the problem retains a rich fluid flow physics manifested by multiple counter rotating recirculating regions on the corners of the cavity depending on the Reynolds number. We have extracted a linear system from the problem with Reynolds’s number of 5000 and using a mesh with 4 million unknowns using OpenFOAM v2.0 [25]. For the following numerical test CURIE cluster located at CEA, France is used. CURIE includes 360 “fat” nodes where each node has 4 eight core Intel EX X7560 processors running at 2.26 GHz and 128 GB of memory. The nodes are connected with an InfiniBand QDR Full Fat Tree network.

4. Numerical results

We have used the BiCGStab as the outer and inner iterative solver in DDPS. The stopping criteria are (relative residuals) $10^{-2}$ and $10^{-10}$, respectively for the outer and inner BiCGStab iterations. For the step in Equation (4) we use Incomplete-LU factorization (not threaded) with drop tolerance $10^{-2}$ and fill-in 5. The right hand side is formed as a vector that consists of all ones. In the following runs we split the stages of the algorithm into two parts: (i) factor and (ii) solve. First stage does not require the right hand side and need not to be repeated if multiple systems are to be solved with the same coefficient matrix. This step mainly consists of obtaining the coefficient matrices in Equations (5) and (6). The solution step, on the other hand, consists of iterative solution of the linear system using a preconditioner. In each iteration, a preconditioned linear system is solved which involves modifying the right hand side with $D^{-1}$ and obtaining the reduced system solution (Equation (6)). Once the reduced system solution is obtained, it is used for retrieving the full solution (Equation (7)).

In Tables 1 and 2, we present the parallel scalability of the DDPS solver using only MPI (Table 1) and MPI combined with OpenMP (Table 2). The objective of the comparing these two tables is to see if for a fixed number of MPI processes (which is equal to the number of partitions) using more threads, would improve the performance. We note that the algorithm is implemented such that it mostly relies on efficient BLAS kernels. For both cases we use Intel Fortran compiler and for BLAS we use single and multithreaded Math Kernel Library (both the compiler and MKL are part of Intel Composer XE ver. 201.3.174), Bull MPI (ver. 1.1.10.1) with “-O2” compiler optimization level and “-openmp” to enable OpenMP threading. MKL routines are not called inside an OpenMP parallel region as it would result in a poor performance. The hybrid runs are obtained by just enabling threads during runtime and using multithreaded version of MKL while the pure MPI runs use sequential MKL and one thread per MPI process. In both cases we use no more than 16 cores per node due to memory bandwidth limitations. The super linear speed improvement observed in Table 1 is also due to the memory bandwidth limitation which becomes less pronounced as the number of nodes increase. While the total solve time decrease as we increase the number of MPI processes up to 64 for the given problem size, it starts to increase beyond 32 processes for the MPI-OpenMP runs. The best time is obtained to be 1.55 seconds. If one, on the other hand, uses only MPI the scalability is not affected by the increase in the number of MPI processes and the best time is obtained to be 1.00 seconds. The most likely reason for the pure MPI implementation scaling better is the fact that the MPI processes run across a smaller number of nodes and hence the MPI communication is mostly handled via the shared memory. The hybrid implementation, however, places one MPI process per node and therefore the communication time starts to dominate the total time especially for large number of processes. Increase of the percentage of the time spent in MPI communication for OpenFOAM linear solver using a diagonal preconditioner is also observed in [26]. If one considers the overall efficiency of the algorithm, we observe that the efficiency is higher if one uses only MPI even though it makes sense to use some number of threads when the amount of work per MPI processes is large (up to 32 processes for this test problem). Notice that the factorization time is a very small fraction of the total time and it scales increasing the number of partitions (i.e. number of MPI processes).

In Table 3, we present the reduced system size, number of outer and average inner iterations, and the final relative residual. Both inner and outer numbers of iterations are very weakly dependent on the number of partitions. Half outer iterations are due to the variation of BiCGStab that exits the iterations early if convergence is detected [27].

5. Conclusions

We have presented scalability results of DDPS algorithm for a sparse linear system extracted from OpenFOAM 3D lid-driven cavity test problem on Curie cluster. Pure MPI implementation performs and scales better than the MPI-OpenMP hybrid implementation. DDPS algorithm can be viewed as an enhancement over the diagonal and block Jacobi preconditioned iterative schemes. This is due to the fact that the largest elements in the off diagonal blocks are also taken into account. Because of its robustness and scalability, the DDPS algorithm can be applied for solving sparse linear systems that arise in various application domains.
Table 1. Scalability (given as wall clock time in seconds for factor, solve and total) of DDPS using one MPI process per core on CURIE

<table>
<thead>
<tr>
<th>nodes</th>
<th>processes</th>
<th>cores</th>
<th>factor</th>
<th>solve</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0,41</td>
<td>22,66</td>
<td>23,08</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>16</td>
<td>0,20</td>
<td>19,38</td>
<td>19,58</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>32</td>
<td>0,09</td>
<td>1,82</td>
<td>1,91</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>64</td>
<td>0,03</td>
<td>0,97</td>
<td>1,00</td>
</tr>
</tbody>
</table>

Table 2. Scalability (given as wall clock time in seconds for factor, solve and total) of DDPS using 1 MPI process per node and 16 threads per process on CURIE

<table>
<thead>
<tr>
<th>nodes</th>
<th>processes</th>
<th>cores</th>
<th>factor</th>
<th>solve</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>128</td>
<td>0,37</td>
<td>7,06</td>
<td>7,44</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>256</td>
<td>0,18</td>
<td>6,57</td>
<td>6,75</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>512</td>
<td>0,09</td>
<td>1,46</td>
<td>1,55</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>1024</td>
<td>0,03</td>
<td>3,71</td>
<td>3,74</td>
</tr>
</tbody>
</table>

Table 3. Size of the reduced system, number of outer iterations, average number of inner iterations and the final relative residual for DDPS solver using different number of partitions

<table>
<thead>
<tr>
<th>part</th>
<th>red. sys. size</th>
<th>outer</th>
<th>avg. inner</th>
<th>rel.res</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3200</td>
<td>24,5</td>
<td>1,16</td>
<td>3,47E-11</td>
</tr>
<tr>
<td>16</td>
<td>9600</td>
<td>24,5</td>
<td>1,19</td>
<td>2,57E-11</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>26,5</td>
<td>1,83</td>
<td>1,27E-11</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
<td>28</td>
<td>1,72</td>
<td>2,65E-11</td>
</tr>
</tbody>
</table>

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resource Curie at CEA, France. I would like to thank Okan Dogru for extracting sparse linear systems from a 3D lid-driven problem in OpenFOAM and Joerg Hertzer for his comments and suggestions.

References


