MUMPS Based Approach to Parallelize the Block Cimmino Algorithm

Carlos Balsa[1], Ronan Guivarch[2], João Raimundo[2], and Daniel Ruiz[2]

[1] Instituto Politécnico de Bragança, Portugal
balsa@ipb.pt
[2] IRIT–ENSEEIHT, Toulouse, France
{guivarch, jraimund, ruiz}@enseeiht.fr

Abstract. The Cimmino method is a row projection method in which the original linear system is divided into subsystems. At every iteration, it computes one projection per subsystem and uses these projections to construct an approximation to the solution of the linear system.
The usual parallelization strategy applied in block algorithms is to distribute the different blocks on the different available processors. In this paper, we follow another approach where we do not perform explicitely this block distribution to processors whithin the code, but let the multi-frontal sparse solver MUMPS handle the data distribution and parallelism. The data coming from the subsystems defined by the block partition in the Block Cimmino method are gathered in an unique matrix which is analysed, distributed and factorized in parallel by MUMPS. Our target is to define a methodology for parallelism based only on the functionalities provided by general sparse solver libraries and how efficient this way of doing can be.
We relate the development of this new approach from an existing code written in Fortran 77 to the MUMPS-embedded version. The results of the ongoing numerical experiments will be presented in the conference.

Avant-propos

Our original proposal for this conference was called "Parallelization of BlockCGSI Algorithm" and was based on former work still under progress.

For various reasons, we had to change the direction of our investigations and developments. Actually, we focus on the parallelization of the Block Cimmino Algorithm, which is the inner iterative part of the BlockCGSI Algorithm mentioned above. Starting from a sequential Fortran 77 version of a solver based on this algorithm, and following the idea discussed in the abstract, we are currently working on the first parallel version of the method.

This change of direction which occured after some preliminary investigations has induced some delays in our former schedule and, in particular, in the experimental phase which remains to be performed and will only be shown in a future release of this article. For this inconvenience, we please ask the readers of this extended abstract to forgive us.

1 Introduction

As we have already stated in abstract, the Cimmino method is a row projection method in which the original linear system is divided into subsystems. At each iteration, it computes one projection per subsystem and uses these projections to construct an approximation to the solution of the linear system. The Block-CG method can also be used inside the Block Cimmino Iteration to accelerate its convergence rate. Therefore, we present an implementation of a parallel distributed Block Cimmino method were the Cimmino iteration matrix is used as a preconditioner for the Block-CG.

In this work we propose to investigate a non usual methodology for parallelization of the Block Cimmino method where the direct solver package MUMPS (MUltifrontal Massively Parallel sparse direct Solver [1, 2]) is incorporated in order to schedule and perform most of the parallel tasks. This sparse solver library offers to the user the facilities to call the different phases of the solver without taking care of the distribution of data and processes. It also implements various functionalities, like residual computation (for iterative refinement), etc, which can also be exploited at different steps of the iterative solver.

The outline of the paper is the following: the Block Cimmino Algorithm is described in section 2, the parallelization strategy will be exposed in section 3 and the current status of the work is presented in section 4. We finish by a conclusion where we list all the improvements and developments still to be done.

2 Block Cimmino Algorithm

The Block Cimmino method is a generalization of the Cimmino method [3]. Basically, we partition the linear system of equations:

$$\mathbf{A}x = b, \tag{1}$$

where $\mathbf{A}$ is a $m \times n$ matrix, into l subsystems, with $l \leq m$, such that:

$$\begin{pmatrix} \mathbf{A}^1 \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^l \end{pmatrix} x = \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^l \end{pmatrix} \tag{2}$$

The block method [4] computes a set of l row projections, and a combination of these projections is used to build the next approximation to the solution of the linear system. Now, we formulate the Block Cimmino iteration as:

$$\begin{aligned} \delta^{i^{(k)}} &= \mathbf{A}^{i^+} b^i - \mathbf{P}_{\mathcal{R}(A^{i^T})} x^{(k)} && (3) \\ &= \mathbf{A}^{i^+} \left(b^i - \mathbf{A}^i x^{(k)} \right) \\ x^{(k+1)} &= x^{(k)} + \nu \sum_{i=1}^{l} \delta^{i^{(k)}} && (4) \end{aligned}$$

In (3), the matrix $\mathbf{A}^{i^+}$ refers to the classical pseudoinverse of $\mathbf{A}^i$ defined as: $\mathbf{A}^{i^+} = \mathbf{A}^{i^T} \left(\mathbf{A}^i \mathbf{A}^{i^T} \right)^{-1}$. However, the Block Cimmino method will converge for any other pseudoinverse of $\mathbf{A}^i$ and in our parallel implementation we use a generalized pseudo-inverse [5], $\mathbf{A}^{i-}_{\mathbf{G}^{-1}} = \mathbf{G}^{-1} \mathbf{A}^{i^T} \left(\mathbf{A}^i \mathbf{G}^{-1} \mathbf{A}^{i^T} \right)^{-1}$, were $\mathbf{G}$ is some symmetric and positive definite matrix. The $\mathbf{P}_{\mathcal{R}(A^{i^T})}$ is an orthogonal projector onto the range of $\mathbf{A}^{i^T}$.
We use the augmented systems approach, as in [6] and [7], for solving the subsystems (3)

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}^{i^T} \\ \mathbf{A}^i & 0 \end{bmatrix} \begin{bmatrix} u^i \\ v^i \end{bmatrix} = \begin{bmatrix} 0 \\ b^i - \mathbf{A}^i x \end{bmatrix} \tag{5}$$

with solution:

$$\begin{aligned} v^i &= -\left(\mathbf{A}^i \mathbf{G}^{-1} \mathbf{A}^{iT} \right)^{-1} r^i \\ u^i &= \mathbf{A}^{i-}_{\mathbf{G}^{-1}} (b^i - \mathbf{A}^i x) && (6) \\ &= \delta^i \end{aligned}$$

The Block Cimmino method is a linear stationary iterative method with a symmetrizable iteration matrix [8]. The symmetrized iteration matrix is symmetric positive definite (SPD), and we can accelerate its rate of convergence with the use of Block Conjugate Gradient method (Block-CG).

The Block-CG method [9, 10] simultaneously searches for the next approximation to the system's solution in a given number of Krylov subspaces, and this number is given by the *block size* of the Block-CG method. The Block-CG method converges in a finite number of iterations in abscense of roundoff errors.

3 Parallelization strategy

With block algorithms, a way to parallelize, which seems to be the more naturaly exploited in general, is to distribute the different blocks on the different available processors as developed in [11]. An easy distribution could be, in this case, to assign a block per processor. This natural way becomes not so easy when we try to find a distribution which aims to improve the load-balancing:

– if we have a high number of processors, we can create the same number of blocks; but the convergence of blocked algorithms often decreases when the number of blocks increases;
– if the blocks are too small, the cost of communication will be prohibitive relatively to the computations.

In this paper, we follow another approach where we will forget this block distribution to processors by letting the sparse direct solver package MUMPS handle the data distribution and the parallelization. The data coming from the subsystems defined by the block partition (2) in the Block Cimmino method are gathered into an unique matrix. This block diagonal matrix is then given to MUMPS to be distributed, analyzed and factorized in parallel. Then, at each Block-CG iteration, MUMPS solves the system involved during the preconditioner step. Finally, we let MUMPS taking care of the distribution with respect of the structure and properties of the matrix in order to have the best factorization and load balancing. The added advantage of this way of doing, is that the sparse linear solver will handle (without any extra development for us) all the levels of parallelism available, and in particular those coming from sparsity structure and BLAS3 kernels on top of the block partitioning. This also give us more degrees of freedom when partioning the matrix as in (2), with the possibility to define less blocks than processors but larger ones, that may help to increase the speed of convergence of the method with still good parallelism in the end since the three levels of parallelism above are managed together.

After each solution of the preconditioner step, there is some operations to perform in the Block-CG algorithm, such as daxpy, ddot, residual computation, ... The first step in our development was to start from an existing code written in Fortran 77 by improving it (dynamic memory allocation, see section 4). We then implemented the preconditioner step with the use of the MUMPS package. However, after this step, we gather the distributed results on a a master processor to perform the remaining Block-CG operations (daxpy, ddot, etc) still in sequential. The next step to achieve our goal will be to perform most of these operations in parallel by letting the data in place after distribution and factorization by MUMPS, and these operations will be performed by means of the MUMPS-embedded functionalities for data management and communications.

4 Parallelization

In the existing code written in Fortran 77, all the memory allocation is made statically; the dynamic memory allocation being first available only in Fortran 90. To handle the memory in a static manner, a particular strategy is adopted. Two arrays of size defined by the programmer are created, *IPool* and *DPool*, and used as a memory stack to store all integer and double precision major variables respectively. In this way, it is assured that during the execution of the program no more memory than that specify by the programmer will be used. There are two extra variables, *FinIPool* and *FinDPool* indicate at any time during the execution the amount of memory occupied in each array. This is flexible enough to have variables that are no longer needed being overridden by new variables, thus allowing some memory management and less memory waste.

The main issue about the original code is that, in order to use the direct solver MUMPS, it is necessary to have dynamic memory allocation management of the variables used by the solver. So, to allow a greater flexibility in the program, we started to change all variables stacked in *IPool* and *Dpool* to dynamic allocatable arrays.

For a better structuration and understanding of the code, we took advantage of another feature of FORTRAN 90, the modules. The module created in this case contains all the structure prototypes, which, in turn contains all variables used with dynamic allocation.

With dynamic allocation implemented, the next step was to move from the original sequential solver MA57 [12] to the parallel solver MUMPS. The MUMPS package has two versions, a sequential and a parallel one. Because MA57 and MUMPS are direct solvers, the switch was not very difficult: we just had to change the calls to these solvers in the three main phases, i.e Analysis, Factorization and Solve.

However the data had to be transformed completely, and in particular matrix blocks and right hand sides. These data are stored in MUMPS variables along with the three phases as they are needed but, MUMPS doesn't accept the matrices by blocks so, we had to group all the blocks from the partitioning of the Cimmino method, in the according array variable in MUMPS.

After the sequential MUMPS was implemented, the next step was to implement the parallel version. We just needed a new compilation of MUMPS with the parallel libraries (ScaLAPACK) and several MPI function calls. One of the nice feature of this implementation is that, to go from the sequential to parallel version of our code, only few changes are necessary, the interface of MUMPS being the same in the two cases.

5 Ongoing works

We are currently working on the validation of the firt parallel version of the code described above. We plan to perform various tests for performance, scalability, parallelism with some classical matrices from Matrix Market.

These tests will enable us to differentiate the parallel part, the cost of communication and the weight of the remaining sequential operations, with varying numbers of processors, matrices with different sizes and structure, etc. We will also investigate different partitioning strategies, some with few but large blocks and others with more but smaller blocks, and compare the trade-off between the three levels of parallelism managed alltogether, the fill-in when factorization, and the speed of convergence that may vary from one to the other partitioning strategy.

After that, to implement the final targeted parallel version, we will identify the basic embedded functionalities already available in the MUMPS package, such as residual computation used in iterative refinement for instance, and design new ones whenever necessary. We will also define the user interfaces to adress directly these functionalities within the parallel iterative solver so as to exploit the data distribution already established and handled by MUMPS.

References

1. Amestoy, P., Buttari, A., Combes, P., Guermouche, A., L'Excellent, J.Y., Pralet, S., Ucar, B.: Multifrontal massively parallel solver - user's guide of the version 4.7. Technical Report 957, Toulouse, France (2007)
2. Amestoy, P., Duff, I., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23** (2001) 15–41
3. Cimmino, G.: Calcolo approssimato per le soluzioni dei sistemi di equaziono lineari. In: Ricerca Sci. II. Volume 9, I. (1938) 326–333
4. Arioli, M., Duff, I.S., Noailles, J., Ruiz, D.: A block projection method for sparse matrices. SIAM J. Sci. Stat. Comput. (1992) 47–70
5. Campbell, S.L., Meyer, J.C.D.: Generalized inverses of linear transformations. Pitman, London (1979)
6. Bartels, R.H., Golub, G.H., Saunders, M.A.: Numerical techniques in mathematical programming. In J.B. Rosen, O. L. Mangasarian, K.R., ed.: Nonlinear Programming. Academic Press, New York (1970)

7. Hachtel, G.D.: Extended applications of the sparse tableau approach - finite elements and least squares. In Spillers, W.R., ed.: Basic question of design theory. North Holland, Amsterdam (1974)
8. Hageman, L.A., Young, D.M.: Applied Iterative Methods. Academic Press, London (1981)
9. O'Leary, D.P.: The block conjugate gradient algorithm and related methods. Linear Algebra Appl. **29** (1980) 293–322
10. Arioli, M., Ruiz, D.: Block conjugate gradient with subspace iteration for solving linear systems. In: Second IMACS International Symposium on Iterative Methods in Linear Algebra, Blagoevgrad, Bulgaria (1995) 64–79
11. Arioli, M., Drummond, A., Duff, I.S., Ruiz, D.: A parallel scheduler for block iterative solvers in heterogeneous computing environments. In: Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing, Philadelphia, USA, SIAM (1995) 460–465
12. Duff, I.: MA57 - A new code for the solution of sparse symmetric definite and indefinite systems. Technical Report RAL-TR-2002-024 (2002)