Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2023) 31: 596 – 611
© TÜBİTAK
doi:10.55730/1300-0632.4004

Quadratic programming based partitioning for Block Cimmino with correct value representation

Zuhal TAŞ, F. Şükrü TORUN*
Department of Computer Engineering , Yıldırım Beyazıt University, Ankara, Turkey

Received: 22.09.2022 • **Accepted/Published Online:** 30.04.2023 • **Final Version:** 28.05.2023

Abstract: The block Cimmino method is successfully used for the parallel solution of large linear systems of equations due to its amenability to parallel processing. Since the convergence rate of block Cimmino depends on the orthogonality between the row blocks, advanced partitioning methods are used for faster convergence. In this work, we propose a new partitioning method that is superior to the state-of-the-art partitioning method, GRIP, in several ways. Firstly, our proposed method exploits the Mongoose partitioning library which can outperform the state-of-the-art methods by combining the advantages of classical combinatoric methods and continuous quadratic programming formulations. Secondly, the proposed method works on the numerical values in a floating-point format directly without converting them to integer format as in GRIP. This brings an additional advantage of obtaining higher quality partitionings via better representation of numerical values. Furthermore, the preprocessing time is also improved since there is no overhead in converting numerical values to integer format. Finally, we extend the Mongoose library, which originally partitions graphs into only two parts, by using the recursive bisection paradigm to partition graphs into more than two parts. Extensive experiments conducted on both shared and distributed memory architectures demonstrate the effectiveness of the proposed method for solving different types of real-world problems.

Key words: Parallel computing, graph partitioning, quadratic programming, Mongoose, recursive bisection, block Cimmino.

1. Introduction

Solution of systems of linear equations is required in a variety of domains, including structural analysis [1], chemical engineering [2], network theory [3], fluid dynamics [4], data analysis [5], and circuit theory [6]. These domains give rise to linear systems of equations in each of which the coefficient matrices are large sparse matrices. Although much progress has been made in computer hardware over the last few years in order to perform as many processes per second as possible, solving these systems still requires quite an amount of time. One way to alleviate this problem is to perform computations by using parallel computing platforms in an efficient manner.

There are mainly two types of methods for solving systems of linear equations: direct and iterative methods. Direct methods work by factorizing a permutation of the input coefficient matrix. These methods have the potential for large memory usage when solving large linear problems. However, direct methods are known to be robust and reliable. On the other hand, iterative methods are based on matrix-vector operations. Thus iterative methods have the advantage of low and predictable memory consumption. Iterative methods generally require preconditioning techniques for faster convergence by enhancing the numerical properties of

*Correspondence: fstorun@aybu.edu.tr

the matrix. For large and sparse linear systems iterative methods are preferred generally due to the minimal memory usage and amenability to parallelism.

Hybrid methods emerged later as an alternative method that can combine the robustness of direct methods and the memory economy of iterative methods. Domain Decomposition Methods (DDM) are one of the categories of hybrid methods that aim to exploit all computing resources of parallel platforms, by dividing the entire linear system into subproblems that can be addressed separately. A direct solver is then applied in parallel to each subproblem, and a global iterative approach is used to integrate the obtained partial solutions to assure the consistency of the global solution.

The block Cimmino method, which is a block-row projection method, is one of the well-studied hybrid methods [7–12]. Among the row projection methods, Kaczmarz [13] and Cimmino [14] are two well-known instances. Kaczmarz uses the product of orthogonal projection to obtain the solution, while Cimmino uses the sum of orthogonal projections. Usually, the Cimmino method is preferred for the parallel solution of large linear systems [7] since it is more amenable to parallelism than the Kaczmarz method due to the summation of independent orthogonal projections. The Cimmino method, on the other hand, requires a larger number of iterations than Kaczmarz. The block Cimmino method [8] is a block-row version and requires a relatively small number of iterations for convergence since the projections are computed on row blocks instead of all rows one by one. To compute the projections of the row blocks one needs to use additional methods which can be classical direct and iterative methods. If a direct method is preferred then the block Cimmino method for solving sparse linear systems has the advantage of using both direct and iterative methods together to solve smaller independent systems in a simple iterative framework.

The number of iterations in the block Cimmino method depends on the orthogonality between row blocks. In order to decrease the number of iterations, several partitioning methods are proposed [9, 10]. The main objective of these tools is the closer to orthogonal row blocks, the less number of iterations [7]. The most recent method [10] (GRIP) proposes a graph theoretical approach for determining the row blocks of the matrix. GRIP aims to minimize inner products between row blocks which leads to significantly fewer iterations for convergence.

In this work, a new method is proposed and implemented to fulfill some drawbacks of the implementation of GRIP. GRIP uses a state-of-the-art partitioning tool (METIS [15]) to partition the row-inner-product graph. However, METIS cannot keep the floating-point values of the edge weights of the graph in floating-point format since the data structures used in METIS can store only integer-type values. In GRIP, to alleviate this limitation of METIS, floating-point values were scaled and then rounded to a certain integer range (e.g., [1,100]). This results in the loss of important numerical information due to the use of a much smaller numerical range instead of using double-precision floating-point representation. In order to identify row blocks correctly during the partitioning phase, the numerical values of the graph are crucial and should be used as are. Although using a larger integer range (e.g.,$[1, 1{,}000{,}000]$) may overcome this issue to some degree, we observe integer overflow problems in large matrices during partitioning. Furthermore, this scaling and rounding process also causes an extra preprocessing overhead which increases the execution time of the method.

In the new method[1], we utilize Mongoose [16] multilevel graph partitioning library. Mongoose is a sophisticated library such that it can leverage classical combinatoric methods and continuous quadratic programming formulations together for graph partitioning. Besides these advantages of Mongoose, it allows floating-point values in the edge weights of its graph data structure. This in turn leads to better and more direct graph rep-

[1]ABCD with Mongoose [online]. Website https://github.com/AYBU-ParLab/ABCD_Mongoose [accessed 9 March 2023]

resentation for the partitioning problem of block Cimmino by keeping the numerical values in double-precision floating-point format without casting them to integer as in METIS. Additionally, it has also been shown that Mongoose can provide better quality partitioning than METIS due to its advanced features such as quadratic programming formulations and some advanced coarsening methods [16].

Furthermore, in the new method, we adopt and implement a recursive bisection paradigm to obtain more than two parts since the current implementation of Mongoose allows only binary graph partitioning. Then, we compare our new partitioning method against GRIP in terms of the number of iterations required for the convergence in block Cimmino and the total parallel solution time by using two different parallel architectures.

The rest of the paper is organized as follows. We give background information about the block Cimmino method, graph partitioning via the GRIP method, Mongoose, and its features in Section 2. In Section 3, we explain the proposed approach and its implementation details. In Section 4, we demonstrate the performance of the proposed method by comparing it with the state-of-the-art method in two different parallel computing architectures. Finally, we conclude the paper with a summary and discussion in Section 5.

2. Background

2.1. Block Cimmino method

In this study, we consider a linear system of equations of the form

$$Ax = f, \tag{1}$$

where A is a $n \times n$ sparse nonsymmetric nonsingular matrix and x and f are column vectors of size n. One popular choice to solve (1) is block Cimmino method [9–12, 17]. In block Cimmino, the blocks are defined by partitioning the system (1) into p blocks of rows with p≤n as follows:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_p \end{pmatrix}. \tag{2}$$

Algorithm 1 shows the steps of the classical block Cimmino method. At line 4, the row block projections are computed independently. Here, A_k^+ denotes the minimum 2-norm solution of an underdetermined linear system where the right-hand-side vector is $(f_k - A_k x^{(j)})$. At line 6, the computed projections are summed up and used for updating the new solution vector after being scaled by the relaxation parameter ϕ. The algorithm is very convenient for parallel computation since the most time-consuming part can be computed perfectly parallel without any communication. Only at line 6, the communication across processors is needed to gather δ_k vectors.

Algorithm 1 Block Cimmino method

```
1: Choose $x^{(0)}$
2: while $j = 0, 1, 2, \ldots,$ until convergence do
3:     for $k = 1, \ldots, p$ do
4:         $\delta_k = A_k^+(f_k - A_k x^{(j)})$
5:     end for
6:     $x^{(j+1)} = x^{(j)} + \phi \sum_{k=1}^{p} \delta_k$
7: end while
```

The block Cimmino iteration is described by

$$x^{(j+1)} = x^{(j)} + \phi \sum_{i=1}^{p} A_i^{+}(f_i - A_i x^{(j)}) \tag{3}$$

$$= \left(I - \phi \sum_{i=1}^{p} A_i^{+} A_i \right) x^{(j)} + \phi \sum_{i=1}^{p} A_i^{+} f_i \tag{4}$$

$$= (I - \phi H)\, x^{(j)} + \phi \sum_{i=1}^{p} A_i^{+} f_i \tag{5}$$

$$= \hat{Q} x^{(j)} + \phi \sum_{i=1}^{p} A_i^{+} f_i, \tag{6}$$

where $\hat{Q}$ is called the iteration matrix. In [7, 8], the Conjugate Gradient (CG) acceleration is proposed to enhance the convergence rate of the block Cimmino method. The CG accelerated block Cimmino method is applied to the following system

$$\phi H x = \phi \sum_{k=1}^{p} A_i^{+} f_k, \tag{7}$$

where $H = \sum_{i=1}^{p} A_i^{+} A_i$ (see (5)) and x is the same solution vector in (1). We take ϕ as one since it exists on both sides of the equation in (7) and does not affect the convergence of CG applied to (7). It has been shown that the CG accelerated block Cimmino method has high robustness and successfully solves large sparse linear systems [7].

2.2. Partitioning

The row-block partitioning of block Cimmino, that is, the decision of which rows of the matrix will be in which row blocks, plays an important role in the convergence. We note that internal ordering within row blocks does not affect the convergence [9]. In the solution of (7) via CG, the number of iterations is directly related to the eigenvalue spectrum of H, which is only affected by the row-block partitioning of A. Column ordering of A does not have any impact on the convergence of the system (7) [9]. In fact, the convergence of CG accelerated block Cimmino is associated with the convergence of CG applied on (7).

Let the QR factorization of A_i^T be defined as $Q_i R_i = A_i^T$. Then, the H matrix can be rewritten by using the Q_i factor of each A_i^T as follows [8];

$$\begin{aligned} H &= \sum_{i=1}^{p} A_i^T (A_i A_i^T)^{-1} A_i \\ &= \sum_{i=1}^{p} Q_i Q_i^T \\ &= (Q_1, \ldots, Q_p)(Q_1, \ldots, Q_p)^T. \end{aligned} \tag{8}$$

Since the spectrum of $(Q_1, \ldots, Q_p)(Q_1, \ldots, Q_p)^T$ is identical to the spectrum of $(Q_1, \ldots, Q_p)^T (Q_1, \ldots, Q_p)$

[18], H can be shown as

$$\begin{pmatrix} I_{n_1 \times n_1} & {Q_1}^T Q_2 & \dots & {Q_1}^T Q_p \\ {Q_2}^T Q_1 & I_{n_2 \times n_2} & \dots & {Q_2}^T Q_p \\ \vdots & \dots & \ddots & \vdots \\ {Q_p}^T Q_1 & {Q_p}^T Q_2 & \dots & I_{n_p \times n_p} \end{pmatrix}, \tag{9}$$

where n_i is the row size of A_i and the eigenvalues of each off-diagonal block ${Q_i}^T Q_j$ represent the cosines of the principal angles between the subspaces spanned by the rows of A_i and A_j [19]. In [20], these angles ($\{\theta_i\}_{i=1}^q$ where $q \geq 1$ and $q = min(dim(\mathcal{R}(A_i^T), dim(\mathcal{R}(A_j^T))))$ between these subspaces are also defined successively by,

$$cos(\theta_k) = \max_{\substack{u \in \mathcal{R}(A_i^T) \\ u^T[u_1,\dots,u_{k-1}]=0}} \max_{\substack{v \in \mathcal{R}(A_j^T) \\ v^T[v_1,\dots,v_{k-1}]=0}} \frac{u^T v}{||u||\ ||v||}, \tag{10}$$

where u_i and v_i are the principal vectors and the principal angles ($\{\theta_i\}_{i=1}^q$) satisfy $0 \leq \theta_1 \leq \dots \leq \theta_q \leq \pi/2$. If two subspaces $\mathcal{R}(A_i^T)$ and $\mathcal{R}(A_j^T)$ are orthogonal to each other, then all principal angles between these subspaces $\{\theta_i\}_{i=1}^q = \pi/2$. In the (9) matrix, the wider angles between subspaces, the closer the matrix is to the identity matrix. In the extreme case, if all row blocks are orthogonal to each other, in other words, all inner products between different A_i blocks are zero, then H will be an identity matrix, and the solution obtained through only one iteration of block Cimmino. If the off-diagonal blocks of the (9) matrix are smaller, more eigenvalues would be clustered around one. Consequently, this leads to a fewer number of CG iterations.

Several partitioning methods [9, 10, 21] are proposed to decrease the number of iterations required for the convergence of the block Cimmino method. The most recent work [10] proposes a novel graph-based partitioning method (GRIP) that minimizes the row inner product values between different A_i blocks. In their work, the numerical values of the matrix are taken into account to find a good partitioning that leads to fewer iterations. In [10], a graph $G(A) = (V, E)$ is constructed, where V holds vertex v_i for each row r_i of A. There is an edge $(v_i, v_j) \in E$ if an inner product of row r_i and r_j is different from zero. Each edge (v_i, v_j) is associated with a cost $cost(v_i, v_j) = |\langle r_i, r_j \rangle|$.

In the edge cut graph partitioning problem, a graph is partitioned into balanced subgraphs (parts) while minimizing cut edges between the parts and it is known as an NP-complete problem [22]. If v_i and v_j are located in different parts after the partitioning then the edge (v_i, v_j) becomes a cut edge. The cut size is defined as the sum of the costs of cut edges. The balance between parts is determined by the sum of vertex weights in each part. Edge-cut partitioning aims to reduce the cut size, which is the sum of the costs of the cut edges. With this definition, the aim of GRIP corresponds to obtaining a row-block partitioning of A where the row inner products between row blocks are minimized. After GRIP partitioning, we expect wider angles between A_i row blocks by putting the rows that have larger inner product values into the same part and the rows that have smaller values to the different parts. This results in a significantly decreased number of iterations in block Cimmino for the convergence.

In GRIP, the cost of each edge (v_i, v_j) is equal to the cosine of the angle between the rows r_i and r_j

assuming the rows are scaled to have 2-norm equal to one. Then, the minimizing cut size objective of GRIP corresponds to minimizing the sum of cosines of the angle between all row pairs which are located in different row blocks.

In GRIP, firstly, the row-inner product graph (G_{RIP}) is constructed and then partitioned by the METIS [15] multilevel graph partitioning tool. However, METIS is limited to using only integer edge weight values whereas the edge weight values of G_{RIP} are floating-point values in the half-closed interval of (0,1]. As a result, METIS is not capable of representing the cost of edge weights in G_{RIP} directly. In [10], to get rid of this limitation of METIS, floating-point values are scaled and then rounded to a certain integer range (e.g., [1,100]). It is clear that rounding to integer values can cause a dramatic loss of information which is important to correctly identify row blocks during the partitioning. Furthermore, this scaling process also requires extra preprocessing time which increases the execution time of the method. In addition, we observe that the cut size in METIS can be out of range of integer data type for large graphs, which causes integer overflow problem and decrease the quality of partitioning.

2.3. Mongoose

Mongoose [16] is a library for binary graph partitioning which computes edge cuts on a graph in a multilevel way. The binary graph partitioning problem is defined as the process of dividing a graph into balanced two subgraphs (parts) while minimizing the edges between parts. Similar to METIS, Mongoose adopts a multilevel approach rather than computing an edge cut on the input graph directly. There are mainly 3 levels; coarsening, initial partitioning, and refinement.

In the coarsening level, Mongoose coarsens the graph to obtain a relatively smaller but structurally similar graph. Here Mongoose offers some more advanced strategies besides heavy-edge matching, such as Brotherly matching and Community matching. Brotherly matching allows to group vertices that share a neighbor even if there is no edge physically linking them. Community matching basically allows matching two vertices if their neighbors are matched together. Figure 1 shows coarsening of a sample graph after applying heavy-edge, brotherly, and community matchings.

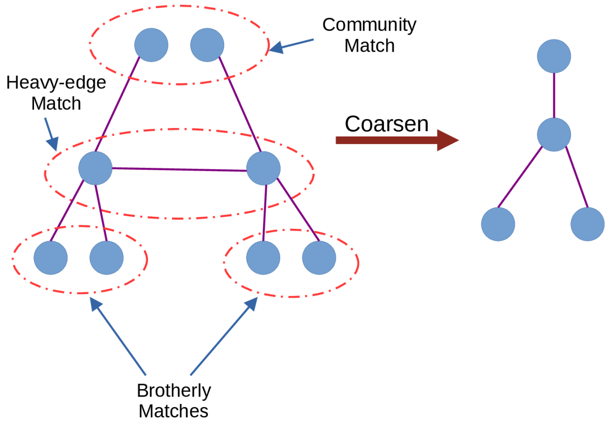

Figure 1. Heavy-edge, brotherly, and community matchings in coarsening.

After coarsening phase, an initial partitioning is computed on the small graph. Mongoose offers an option to use the quadratic programming solver to compute an initial partitioning. Finally, in the refinement phase, an inverse operation of graph coarsening is performed. In the refinement, the quadratic programming solver and combinatorial methods are used together as a default approach.

Mongoose is an advanced graph partitioning tool since it combines different partitioning methodologies to obtain high-quality partitioning. These are quadratic programming solutions and a state-of-the-art iterative refinement heuristic. In [23], a continuous optimization formulation is proposed, which depicts that the binary graph partitioning problem is equivalent to the continuous quadratic programming problem as;

$$\min_{x\in\mathbb{R}^n} (1-x)^T (A+I)\, x \quad \text{subject to} \quad 0 \le x \le 1,\ \ l \le 1^T x \le u, \tag{11}$$

where A is the adjacency matrix of the graph, and l and u are the lower and upper bounds of a partition at the target size. In Mongoose, a gradient projection algorithm is used to solve this formulation.

In addition to quadratic programming, the Fiduccia-Mattheyses (FM) method [24], which is a traditional iterative refinement heuristic, is also utilized in the refinement phase of Mongoose. FM works iteratively and in each iteration (pass) set of vertices is moved from one part to another aiming to improve the edge cut quality. These moves continue even if there is no immediate improvement in order to discover better quality cuts and not to get stuck in local optima. If there is no improvement after some predefined number of moves, the partitioning with the best cut is reverted by looking at the past moves. The next pass is started from the best cut obtained from the previous pass, and these passes are repeated a predefined number of times. Until reaching the final graph partitioning, both the FM and the quadratic programming algorithms are used to refine the graph at each level of the iterative refinement phase.

In this work, we exploit Mongoose due to its three main features. Firstly, the weights of the edges in the graph can be kept as floating-point values. In this way, row inner product values of the matrix can be directly used in the graph, instead of rounding them to integer values as in METIS. In this way, better quality partitioning can be obtained which leads to faster convergence in block Cimmino. Recall that METIS can use only integer values for the edge weights in its graph structure. The second reason is that since METIS uses integer data type to keep the cut size value, for large graphs this can easily cause an integer overflow problem. Therefore, it can result in low-quality partitioning because of the negative cut size values which must always be positive typically. The last reason is that since GRIP scales and rounds all floating-point values of the row inner product values to integer values for METIS, it is clear that the preprocessing overhead is decreased without this operation. For instance, we observe that on average 5% of the preprocessing time is consumed by this data type conversion operation in GRIP.

The main drawback of Mongoose is that it enables partitioning into only two parts, whereas METIS can partition a graph into any number of parts. To alleviate this drawback of the Mongoose, in this work, we utilize a recursive bipartitioning approach [25–27]. Thus, we extend the Mongoose library to partition a graph into more than two parts. In the current implementation of the proposed work, Mongoose is utilized to partition graphs only into powers of two parts. We leave the implementation of partitioning a graph into any number of parts by using Mongoose as future work.

3. Proposed recursive bipartitioning based algorithm

In the proposed method, we partition a graph into more than two parts by utilizing the Mongoose graph partitioning tool which can actually partition a graph into only two parts. Our method applies a recursive bipartitioning approach in which 2-way partitioning on a subgraph is performed recursively at each level.

Firstly, the entire graph is partitioned into two parts by using Mongoose. Then, two subgraphs are created according to the partitioning information of Mongoose. After creating these subgraphs, the relation of the original indexes with local indexes in each subgraph is kept in a *map* data structure. This allows us to find the global index of each vertex by transforming its local index after all recursive calls are completed. The recursion continues until the target total number of parts is reached.

Algorithm 2 shows the steps of the proposed recursive bipartitioning method. In the algorithm, the *edge_cut* function is the only function that we adopt from the Mongoose library. At line 1, we maintain the desired maximum allowed imbalance ratio (imb) among vertex parts by setting the target_split parameter in Mongoose. The imbalance ratio is calculated as the ratio of the maximum loaded part over the average load,

$$imb = 1 - \frac{load_{max}}{load_{avg}}. \tag{12}$$

Therefore if we allow a maximum load imbalance of 1%, the load of the largest part can be at most 1% larger than the average load. In Mongoose, the default value for target_split of 0.5, which leads to perfectly balanced parts. Since at the end of the recursive calls we desire at most imb percent imbalance among parts, we set target_split $= 0.5 - (imb^{(1/log(p))} - 1)/2$, where $log(p)$ shows the number of levels which is required to get the target number of parts. The output of the recursive function is *partvector* is a vector that stores the partitioning information of the graph, where the value of each index specifies the part of the respective vertex. After successful successive recursive calls of the *RecursiveBipartitioning* method, *partvector* will be the output of the proposed method.

The main parameters passed to the `RecursiveBipartitioning` function are shown at line 3 of Algorithm 2, where p represents the number of parts to be split, G is the first graph created with Mongoose, *target_split* is the load imbalance threshold and *partvector* is the output parameter that keeps partitioning information. In `RecursiveBipartitioning`, the same value for *target_split* is used in each recursive call, whereas parameters of G, p, and *partvector* are modified in each call. At lines 4–6, the stopping rule of the recursion is shown. At line 7, the $Mongoose :: edge\_cut$ function in Mongoose API returns a *result* object in which the *partition* vector keeps the partitioning information of the vertices. Note that the *edge_cut* function partitions G into only two parts. At line 8, in the *divideGraph* function, two subgraphs are created named G_{left} and G_{right} by using the information of *partition* vector. In G_{left} and G_{right}, the vertices and internal edges that connect to those vertices are extracted from G according to the partitioning. In addition, in the *divideGraph* function, we create *map* data structure in which the relation of the global indexes with local indexes in each subgraph is stored. We use *map* to recover the global indexes of the original graph after the successive recursive calls and relabeling the vertices of subgraphs. At lines 9–11, part_left and part_right are created and initialized, then the graph bipartitioning process continues by calling the same function twice recursively for each subgraph G_{left} and G_{right}. Meanwhile, the number of parts is halved (i.e. $p/2$) at each level and this process continues until p becomes 1. At line 12, the vectors part_left and part_right, which respectively contain the partitioning information of G_{left} and G_{right}, are used to update *partvector* through *map*. Finally, we return the summation of level information (l_lvl and r_lvl) of the recursive bipartitioning functions to correctly differentiate part ids of vertices in each recursive call for updating partvector (at line 12). At the end of the recursive calls, *partvector* keeps the final partitioning information with p parts of G.

In the implementation of the proposed method, firstly, the input matrix A is read and rows of A are scaled such that 2-norm of each row is equal to one. In this way, the value of the inner product of two rows

Algorithm 2 The proposed method via recursive bipartitioning approach

```
Input: Graph $G$
Output: partvector
1: target_split = 0.5 – (( $\sqrt[log(p)]{imb}$) – 1)/2
2: RecursiveBipartitioning($G$, target_split, partvector, $p$)
3: function LVL = RECURSIVEBIPARTITIONING($G$, target_split, partvector, $p$)
4:     if p≤1 then
5:         return 1
6:     else
7:         result = Mongoose::edge_cut($G$, target_split)          ▷ Bipartitioning with Mongoose
8:         [$G_{left}$,$G_{right}$, $map$] = divideGraph($G$, result)   ▷ Divide $G$ into 2 subgraphs using result
9:         Create and initialize part_left and part_right
10:        l_lvl = RecursiveBipartitioning($G_{left}$, target_split, part_left, $p/2$)
11:        r_lvl = RecursiveBipartitioning($G_{right}$, target_split, part_right, $p/2$)
12:        partvector = Update_partvector(part_left, part_right, $map$, l_lvl, r_lvl )
13:        return l_lvl + r_lvl
14:    end if
15: end function
```

directly corresponds to the cosine angle between the respective rows and the partitioning objective of obtaining less connected rows between row-blocks corresponds to more orthogonal row-blocks. Then we construct the row-inner-product graph G and create the respective graph data structure by using the `Graph::create` function in Mongoose API. Here, we can directly represent G with Mongoose's graph data structure thanks to the allowing edge weights in double-precision floating-point format. `Graph::create` takes several parameters to create a graph such as the number of vertices, the number of edges, the adjacency lists of the graph, and the lists of edge and vertex weights.

In Mongoose API, the `EdgeCut_Options::create()` method creates an *option* object which is initially filled with default values. We can set user-defined options by modifying this object. For instance, we set our desired load imbalance threshold by modifying the *target_split* member of *option*. We note that, since we partition the graph recursively, the load imbalance threshold should be chosen carefully at each level of the recursion. Otherwise, the imbalance ratio among the final parts can overflow the maximum allowed partitioning imbalance. Therefore, we use the below formula (Eqn. (13)) in each recursive call of the bipartitioning so that the target load imbalance (imb_ratio) among parts is ensured within the desired ratio

$$imb\_ratio = (\sqrt[log(p)]{imb}) - 1. \tag{13}$$

For Mongoose one needs to subtract imb_ratio from 0.5 to define the maximum allowed load imbalance among parts

$$target\_split = 0.5 - \frac{imb\_ratio}{2}. \tag{14}$$

4. Experimental results

In the experiments, we adopt the CG accelerated block Cimmino implementation of the ABCD Solver package[2]. The current implementation of the ABCD solver includes several methods to obtain row blocks. Among these,

[2]The Augmented Block Cimmino Distributed Solver [online]. Website https://bitbucket.org/apo_irit/abcd/src/Dev-IRIT/, 2022. ABCD Solver v1.1. [accessed 1 September 2022]

the GRIP partitioning method is superior to the others by achieving a faster convergence rate for most of the matrices [10]. We have embedded our proposed partitioning method as a new partitioning method in ABCD Solver. We use the maximum number of iterations as 5000 and the relative norm-wise backward error [28] for the convergence checking at iteration (j) as

$$\frac{||Ax^{(j)} - f||_\infty}{||A||_\infty ||x^{(j)}||_1 + ||f||_\infty} < 10^{-11}. \tag{15}$$

In the experiments, we compare the performance of our proposed row-block partitioning method with the state-of-the-art row-block partitioning method GRIP which uses METIS as a partitioner. As a partitioning constraint, we allow at most 1% load imbalance among row-block sizes in both methods. Other options are left as their default values. In the implementation, C/C++ programming language and pure MPI [29] based parallelism is used.

Table 1 shows the properties of 15 large sparse unsymmetric matrices used in the experiments from the SuiteSparse matrix collection [30]. Those matrices arise from 9 different kinds of real-world applications whose categories are shown in the last column of the table. In the table, matrices are sorted in ascending order according to their sizes.

Table 1. The properties of matrices (n: size, nnz: number of nonzeros).

Matrix name	n	nnz	Kind
rajat26	51,032	247,528	Circuit Simulation Problem
ecl32	51,993	380,415	Semiconductor Device Problem
2D_54019_highK	54,019	486,129	Semiconductor Device Problem
bayer01	57,735	275,094	Chemical Process Simulation Problem
TSOPF_RS_b39_c30	60,098	1,079,986	Power Network Problem
venkat01	62,424	1,717,792	Computational Fluid Dynamics Problem Sequence
shyy161	76,480	329,762	Computational Fluid Dynamics Problem
ASIC_100ks	99,190	578,890	Circuit Simulation Problem
torso2	115,967	1,033,473	2D/3D Problem
cage12	130,228	2,032,536	Directed Weighted Graph
majorbasis	160,000	1,750,416	Optimization Problem
shar_te2-3	200,200	800,800	Combinatorial Problem
cage13	445,315	7,479,343	Directed Weighted Graph
rajat30	643,994	6,175,244	Circuit Simulation Problem
cage14	1,585,705	27,130,349	Directed Weighted Graph

4.1. Experimental framework

In the experiments, we have used two parallel architectures; shared and distributed memory. In our lab, we have a shared memory machine that consists of 40 GB DDR4 memory and two NUMA sockets each has Intel Xeon E5-2620 v3 6-core CPU. On the other hand, the distributed memory architecture is a subset of *sariyer* HPC cluster of UHEM [3]. In the HPC experiments, we use 10 distributed nodes of *sariyer* cluster each node has 128 GB DDR4 memory and two NUMA sockets. In each NUMA socket, Intel Xeon E5-2680 v4 14-cores CPU is installed. Extensive numerical experiments are conducted on the shared memory architecture to test the

[3]UHEM. National Center for High Performance Computing. http://www.uhem.itu.edu.tr, 2021.

influence of the proposed method against GRIP by comparing the required number of iterations for convergence on all test matrices. Since we have no core-hour limitation in this machine, we have run exhaustive experiments with different combinations of part counts and parameters. However, this machine has a relatively small number of cores therefore we perform timing experiments only in the distributed memory system. In the distributed memory system, we have conducted time-critical tests by using only 256 cores due to the core-hour limitation of the HPC system.

4.2. Experiments on shared memory architecture

Table 2 shows the results of experiments in terms of the number of iterations for 2, 4, 8, 16, 32, 64, 128, and 256 parts (blocks) on the shared memory architecture. Experiments are conducted with linear systems of equations whose coefficient matrices are partitioned according to one of the following methods; GRIP, Mongoose, and Mongoose with Community Matching enabled. The best results for each test matrix are shown in bold and blue text. In the table, N denotes the associated linear system that does not reach the desired accuracy in 5000 iterations, F denotes the failures of the solver, and M denotes the failure of insufficient memory.

As seen in Table 2, the proposed method can decrease the required number of iterations in 14 matrices out of 15. Only in `ASIC_100ks`, the proposed method cannot achieve fewer iterations for all part counts. Since this matrix has very dense columns and exhibits power-law distribution, our recursive bipartitioning scheme seems to have difficulty in discovering high-quality cuts. However, out of 120 test instances (15 matrices with 8 different numbers of parts), the proposed method with classical Mongoose gives better results in 92 instances which shows 77% better performance overall. When we enable community matching in Mongoose, we have not seen any remarkable improvement in the results. The last 3 rows of the table show the number of best results for each method. For most of the part count values, the proposed method that uses the default Mongoose gives the best results. Mongoose with community matching enabled does not outperform both methods in any number of parts. Thus, in the rest of the experiments, we do not enable this feature of Mongoose.

Two factors play crucial roles in the success of the proposed method; utilizing floating point values in G_{RIP} instead of integer values and using Mongoose instead of METIS. To see the effect of utilizing floating point values on the convergence of the block Cimmino method, we conduct another experiment with `rajat30`. In this experiment, we construct G_{RIP} with integer edge-weight values as in GRIP, however, this time we use Mongoose to partition G_{RIP} instead of METIS. We call this new method *Mongoose-int*. We have seen a dramatic degradation in the performance with Mongoose-int. For 256 parts, Mongoose-int required 206 iterations, whereas the proposed method (Mongoose + floating points edge-weight values) and GRIP require 84 and 237 iterations, respectively. Similarly, for 128 parts, Mongoose-int required 133 iterations, whereas the proposed method and GRIP require 62 and 234 iterations, respectively. These results also confirm the validity of the importance of the use of floating-point values versus integers in edge weights.

In Table 2, as the number of parts increases, the number of iterations required for convergence increases in general. This is mainly due to the fact that, with the increasing number of parts, the conditioning of H gets worsens in general. This behavior is common for the block Cimmino algorithm. However, for some matrices, it seems there are decreases in iteration counts for increased part counts. We believe that the reason for this issue stems from the heuristics used in Mongoose and METIS. For some part counts, these partitioners cannot obtain good partitionings due to being stuck in local optima which causes more iterations in block Cimmino.

Table 2. The number of iterations for varying numbers of parts on the shared memory architecture.

	The number of parts								
Matrix	Method	2	4	8	16	32	64	128	256
rajat26	GRIP	122	1585	N	N	N	N	N	N
	Proposed	96	373	**513**	**661**	1090	**1291**	**1612**	**2797**
	Proposed(with CM)	**91**	**342**	525	896	**849**	1604	1806	F
2D_54019_highK	GRIP	322	**387**	**187**	618	317	701	642	775
	Proposed	**233**	715	579	**415**	1213	286	430	692
	Proposed(with CM)	286	798	488	633	**196**	**214**	**325**	**480**
ecl32	GRIP	316	580	729	941	1135	1807	2378	3022
	Proposed	**137**	319	470	**521**	**718**	**909**	1155	**1313**
	Proposed(with CM)	199	**307**	**425**	591	725	920	**1082**	1392
bayer01	GRIP	95	133	**260**	740	1286	3420	**3363**	N
	Proposed	93	**65**	366	**657**	**769**	**1714**	3477	**4792**
	Proposed(with CM)	**35**	110	293	909	863	F	F	F
TSOPF_RS_b39_c30	GRIP	769	1396	1847	2343	2393	2690	2836	2411
	Proposed	**44**	**116**	**285**	**616**	**706**	**1192**	1994	**2316**
	Proposed(with CM)	56	141	362	688	709	1448	**1908**	2514
venkat01	GRIP	30	**33**	38	**39**	**43**	48	**51**	**57**
	Proposed	30	34	**36**	**39**	44	**46**	52	58
	Proposed(with CM)	**29**	**33**	37	**39**	**43**	51	**51**	58
shyy161	GRIP	9	F	F	29	25	**25**	26	**27**
	Proposed	**8**	F	F	**24**	25	**25**	26	31
	Proposed(with CM)	10	F	F	31	**24**	**25**	**25**	32
ASIC_100ks	GRIP	**17**	**18**	**20**	**20**	**22**	**20**	**22**	**22**
	Proposed	26	44	53	50	106	96	89	92
	Proposed(with CM)	23	66	F	50	98	98	104	F
torso2	GRIP	**11**	17	17	17	19	20	19	20
	Proposed	**11**	**13**	**15**	**15**	**17**	18	**18**	19
	Proposed(with CM)	12	**13**	**15**	16	18	**17**	F	**18**
cage12	GRIP	**7**	**9**	13	13	**13**	**14**	16	**15**
	Proposed	F	10	11	**12**	**13**	**14**	**14**	**15**
	Proposed(with CM)	8	11	**10**	**12**	**13**	**14**	F	F
majorbasis	GRIP	17	**18**	**19**	**19**	21	**21**	25	25
	Proposed	**16**	19	20	**19**	**20**	**21**	**24**	25
	Proposed(with CM)	17	F	**19**	**19**	**20**	22	**24**	**24**
shar_te2-3	GRIP	F	F	F	F	F	F	F	F
	Proposed	F	**21**	22	**21**	**22**	**23**	**22**	**21**
	Proposed(with CM)	F	**21**	**21**	23	23	24	23	22
cage13	GRIP	M	M	M	13	14	14	16	16
	Proposed	M	F	**12**	**12**	**13**	14	**14**	**15**
	Proposed(with CM)	M	F	**12**	**12**	F	**13**	15	F
rajat30	GRIP	**19**	**20**	**22**	66	67	123	234	237
	Proposed	26	28	56	43	64	67	**62**	**84**
	Proposed(with CM)	21	22	24	**33**	**46**	**45**	75	F
cage14	GRIP	M	M	M	M	M	**14**	**16**	16
	Proposed	M	M	M	F	F	F	F	**15**
	Proposed(with CM)	M	M	M	F	F	F	F	F
# Bests	GRIP	4	**6**	5	2	3	5	4	4
	Proposed	**6**	4	5	**12**	**8**	**9**	**7**	**9**
	Proposed(with CM)	3	5	5	4	7	5	6	3

CM: Community Matching, N: does not converge in 5000 iterations, F: failure of the solver, M: insufficient memory.

4.3. Experiments on HPC

Table 3 shows the results of the experiments using 256 cores on the HPC system. Here, we partition the matrices into 256 blocks each of which is assigned to a core. In HPC experiments, in addition to the iteration count analysis that is done in Table 2, a parallel solution time analysis is performed by measuring the parallel execution time (in seconds) of the solution of the linear systems via CG accelerated block Cimmino excluding the sequential preprocessing steps of conversion and partitioning of the matrix. According to the experiments, the decrease in the number of iterations directly reflects the parallel solution time. The proposed method achieves a better number of iterations and faster parallel solution times in 11 matrices. On the other hand, only in 3 matrices, GRIP gives better results. The block Cimmino method requires the same number of iterations and similar parallel solution times in `cage12` for both methods.

Table 3. Number of iterations and parallel solution times in seconds on HPC system.

	GRIP		Proposed Method	
	# iterations	Parallel time	# iterations	Parallel time
rajat26	N		**2797**	**49.9**
2D_54019_highK	775	12.7	**692**	**10.8**
ecl32	3022	50.7	**1313**	**22.3**
bayer01	N		**3374**	**59.1**
TSOPF_RS_b39_c30	2411	70.8	**2316**	**65.1**
venkat01	**57**	**1.2**	58	1.2
shyy161	**27**	**0.7**	31	0.9
ASIC_100ks	**22**	**0.8**	92	3.3
torso2	20	0.9	**19**	**0.8**
cage12	14	0.1	14	0.1
majorbasis	24	1.5	**23**	**1.5**
shar_te2-3	F		**21**	**2.0**
cage13	16	3.3	**15**	**2.9**
rajat30	237	289.0	**84**	**86.8**
cage14	16	8.3	**15**	**7.9**

N: does not converge in 5000 iterations, F: failure of the solver.

We present Figure 2 to compare the two methods comprehensibly. Figure 2 shows the normalized time and the number of iterations improvements of the proposed method with respect to the GRIP method. According to the experiments, the proposed method achieves the best improvement in `rajat30` with 3.33 times faster parallel solution time. On the other hand, the worst performance is obtained in `ASIC_100ks` with 4.16 times slower time. For `rajat26`, `bayer01`, and `shar_te2−3`, our method is a clear winner due to the failure of the other method, thus, gradient unlimited column bars are used to represent the results of those matrices in the figure. These results confirm the validity of the proposed method by achieving better performance in most of the problems in terms of the number of iterations and parallel solution time through higher-quality row-block partitionings of the block Cimmino method.

5. Conclusion

In this work, we propose a new partitioning method for the block Cimmino method. The proposed method which is based on a recursive bipartitioning paradigm is employed to obtain more orthogonal row blocks in

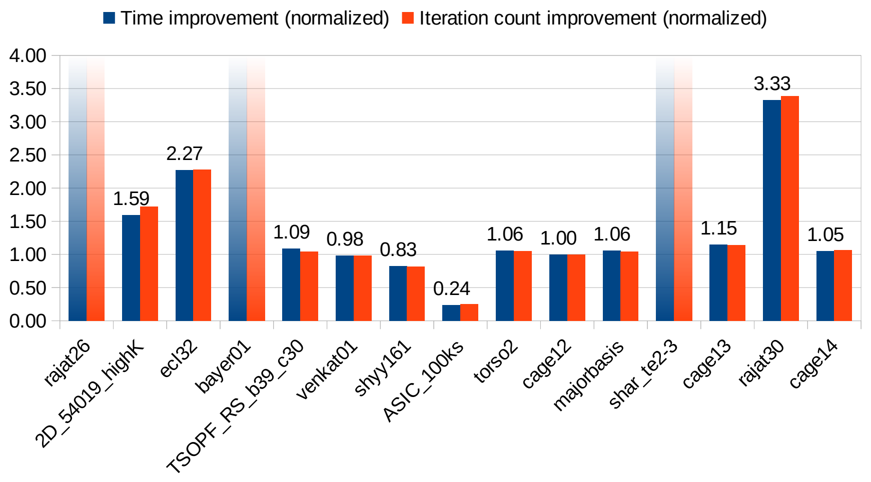

Figure 2. Normalized parallel solution time and the number of iterations improvements of the proposed method with respect to GRIP.

the block Cimmino method. The effectiveness of the proposed method is evaluated by comparing it with the state-of-the-art partitioning method. The proposed method is superior to the other method due to a better representation of numerical values which is quite important to determine orthogonality between row blocks. With this contribution, the qualities of the partitionings are improved considerably. Furthermore, since the direct representation of numerical values of the matrices does not require an extra step to convert them to other data formats, the preprocessing time is also improved by 5% on average. According to the experiments, the proposed method achieves faster parallel solution time by decreasing the number of iterations thanks to the higher quality row-block partitioning in most of the test matrices.

Acknowledgment

Computing resources used in this work were provided by the National Center for High Performance Computing of Turkey (UHeM) under grant number 4012882022.

References

[1] Wilson EL, Bathe K, Peterson F, Dovey H. Sap—a structural analysis program for linear systems. Nuclear Engineering and Design 1973; 25 (2). https://doi.org/10.1016/0029-5493(73)90048-4

[2] Liang Yz, Fang Kt, Xu Qs. Uniform design and its applications in chemistry and chemical engineering. Chemometrics and Intelligent Laboratory Systems 2001; 58 (1): 43–57. https://doi.org/10.1016/S0169-7439(01)00139-3

[3] Borgatti SP, Halgin DS. On network theory. Organization science 2011; 22 (5): 1168–1181. https://doi.org/10.1287/orsc.1100.0641

[4] Anderson JD, Wendt J. Computational fluid dynamics, volume 206. Springer, 1995. https://doi.org/10.1007/978-3-540-85056-4

[5] Wang JL, Chiou JM, Müller HG. Functional data analysis. Annual Review of Statistics and Its Application 2016; 3: 257–295. https://doi.org/10.1146/annurev-statistics-041715-033624

[6] Dickson BG, Albano CM, Anantharaman R, Beier P, Fargione J et al. Circuit-theory applications to connectivity science and conservation. Conservation Biology 2019; 33 (2): 239–249. https://doi.org/10.1111/cobi.13230

[7] Bramley R, Sameh A. Row projection methods for large nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing 1992; 13 (1): 168–193. https://doi.org/10.1137/0913010

[8] Arioli M, Duff I, Noailles J, Ruiz D. A block projection method for sparse matrices. SIAM Journal on Scientific and Statistical Computing 1992; 13 (1): 47–70. https://doi.org/10.1137/0913003

[9] Drummond T, Duff IS, Guivarch R, Ruiz D, Zenadi M. Partitioning strategies for the block cimmino algorithm. Journal of Engineering Mathematics 2015; 93 (1): 21–39. https://doi.org/10.1007/s10665-014-9699-0

[10] Torun FS, Manguoglu M, Aykanat C. A novel partitioning method for accelerating the block Cimmino algorithm. SIAM Journal on Scientific Computing 2018; 40 (6): C827–C850. https://doi.org/10.1137/18M1166407

[11] Dumitrasc A, Leleux P, Popa C, Ruede U, Ruiz D. Extensions of the augmented block cimmino method to the solution of full rank rectangular systems. SIAM Journal on Scientific Computing 2021; 43 (5): S516–S539. https://doi.org/10.1137/20M1348261

[12] Duff I, Leleux P, Ruiz D, Torun FS. Row replicated block cimmino. Technical report, CERFACS, 2022.

[13] Karczmarz S. Angenaherte auflosung von systemen linearer glei-chungen. Bull. Int. Acad. Pol. Sic. Let., Cl. Sci. Math. Nat. 1937; pages (in German). 355–357.

[14] Cimmino G. Estensione dell'identita di Picone alla piu generale equazione differenziale lineare ordinaria autoaggiunta: nota. Bardi, 1929 (in İtalian).

[15] Karypis G, Kumar V. Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed computing 1998; 48 (1): 96–129. https://doi.org/10.1006/jpdc.1997.1404

[16] Davis TA, Hager WW, Kolodziej SP, Yeralan SN. Algorithm 1003: Mongoose, a graph coarsening and partitioning library. ACM Transactions on Mathematical Software (TOMS) 2020; 46 (1): 1–18. https://doi.org/10.1145/3337792

[17] Duff IS, Guivarch R, Ruiz D, Zenadi M. The augmented block cimmino distributed method. SIAM Journal on Scientific Computing 2015; 37 (3): A1248–A1269. https://doi.org/10.1137/140961444

[18] Golub G, Kahan W. Calculating the singular values and pseudo-inverse of a matrix. Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis 1965; 2 (2): 205–224. https://doi.org/10.1137/0702016

[19] Björck Å, Golub GH. Numerical methods for computing angles between linear subspaces. Mathematics of computation 1973; 27 (123): 579–594. https://doi.org/10.1090/S0025-5718-1973-0348991-3

[20] Golub GH, Van Loan CF. Matrix computations. JHU press, 2013.

[21] Zenadi M. Méthodes hybrides pour la résolution de grands systèmes linéaires creux sur calculateurs parallèles. Ph.D. thesis, École Doctorale Mathématiques, Informatique et Télécommunications (Toulouse); 142547247, 2013 (in French).

[22] Johnson DS, Garey MR. Computers and intractability: A guide to the theory of NP-completeness. WH Freeman, 1979.

[23] Hager WW, Krylyuk Y. Graph partitioning and continuous quadratic programming. SIAM Journal on Discrete Mathematics 1999; 12 (4): 500–523. https://doi.org/10.1137/S0895480199335829

[24] Fiduccia CM, Mattheyses RM. A linear-time heuristic for improving network partitions. In 19th design automation conference. IEEE, 1982; pages 175–181. https://doi.org/10.1109/DAC.1982.1585498

[25] Chamberlain BL. Graph partitioning algorithms for distributing workloads of parallel computations. Technical report, University of Washington, 1998.

[26] Drechsler R, Gunther W, Eschbach T, Linhard L, Angst G. Recursive bi-partitioning of netlists for large number of partitions. In Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools. IEEE, 2002; pages 38–44. https://doi.org/10.1109/DSD.2002.1115349

[27] Hendrickson B, Leland R. An improved spectral graph partitioning algorithm for mapping parallel computations. SIAM Journal on Scientific Computing 1995; 16 (2): 452–469. https://doi.org/10.1137/0916028

[28] Arioli M, Duff I, Ruiz D. Stopping criteria for iterative solvers. SIAM Journal on Matrix Analysis and Applications 1992; 13 (1): 138–144. https://doi.org/10.1137/0613012

[29] Snir M, Gropp W, Otto S, Huss-Lederman S, Dongarra J et al. MPI–the Complete Reference: the MPI core, volume 1. MIT press, 1998.

[30] Davis TA, Hu Y. The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS) 2011; 38 (1): 1–25.