Available on-line at www.prace-ri.eu
Partnership for Advanced Computing in Europe

Scaling Block Conjugate Gradient Variants Orthomin and Orthodir

Oguz Selvitopi[a], M. Ozan Karsavuran[a], Cevdet Aykanat[a,*]

[a]Bilkent University, Computer Engineering Department, 06800 Ankara, TURKEY

Abstract

Iterative solvers based on Krylov subspace methods are widely used for the solution of the problems that appear in large-scale parallel scientific simulations. These solvers, when parallelized, often suffer from global synchronization overheads due to the collective communication operations. Block CG variants have the advantage of reduced communication overheads at the expense of increased computation per iteration. The aim of this project is the scalable parallelization of two such block CG variants, Orthomin and Orthodir, proposed by FET-HPC project NLAFET to enable the use of these methods on future exascale systems through reducing number of the synchronization points. We investigate 1D- and 2D-partitioning of the sparse coefficient matrix for encapsulating the minimization of the communication overhead as well as one- and two-constraint partitioning for computational load balancing. Two different parallel codes for Orthomin and Orthodir variants are developed. The relative performance of different partitioning techniques are evaluated by running the parallel Orthomin and Orthodir codes on two different HPC systems up to 1024 processors with 26 symmetric positive definite matrices. The number of rows in these matrices ranges between 13,681 and 1,391,349, whereas the number of nonzero entries ranges between 340,431 and 77,651,847.

1. Introduction

Iterative solvers based on Krylov subspace methods are widely used for the solution of the problems that appear in large-scale parallel scientific simulations. These solvers, when parallelized, often suffer from global synchronization overheads due to the collective communication operations as discussed in several works [6, 7, 8, 9, 14, 15, 16, 18].

Recently, FET-HPC project NLAFET (Parallel Numerical Linear Algebra for Future Extreme Scale Systems) has proposed new iterative methods based on enlarged Krylov subspaces that are variants of the Conjugate Gradient (CG) method [10, 11]. Block CG variants have the advantage of reduced communication overheads at the expense of increased computation per iteration. These methods are based on Orthomin and Orthodir variants and they dynamically reduce the number of search directions compared to the existing block CG methods. It is shown that these two variants converge faster and reduce the computation and memory overheads compared to the existing block CG variants for solving symmetric positive definite linear systems with a single right hand side.

The aim of this project is to enable the scalable parallelization of Orthomin and Orthodir methods proposed by FET-HPC project NLAFET for future exascale systems. Both methods contain the following kernel operations: sparse matrix dense matrix multiply (SpMM), dense matrix update (DMU), dense matrix matrix transpose multiply (DMMT), and forward and backward substitution (FS/BS) on triangular factors of block matrices. Intelligent partitioning of the sparse coefficient matrix is crucial for the efficient parallelization of these kernel operations. The input and output dense matrices of the SpMM operation should be partitioned conformably in order to avoid redundant communication during DMU and DMMT operations. In this setting, SpMM incurs irregular point-to-point communications, DMU incurs no communication, DMMT incurs a regular collective communication, and FS/BS incurs no communication based on sequential factorization of a block matrix .

We investigated and implemented one-dimensional (1D) and two-dimensional (2D) partitioning models with single-constraint and multi-constraint variants. In 1D partitioning, we implemented row-wise partitioning of the sparse coefficient matrix, utilizing the standard graph model, that aims to minimize the total inter-processor communication volume while maintaining computational load balance. In 2D partitioning, we adopted a two-phase approach in which we utilized the described 1D partitioning in the first phase and 2D cyclic matrix distribution in the second phase [3]. The 2D model has the nice property of providing $O(\sqrt{K})$ bound on the number of messages sent/received by a processor, where K denotes the number of processors. Note that the

* Corresponding author: e-mail. aykanat@cs.bilkent.edu.tr tel. +90 (312) 290 1213 fax. +90 (312) 266 4152 March 2019

bound for the same metric in 1D partitioning is $O(K)$. In both 1D- and 2D-partitioning, using a single constraint aims at maintaining balance on the computational loads of processors during the parallel SpMM operations, whereas using two constraints aims at maintaining balance on the computational loads of processors during both parallel SpMM and DMU-DMMT-FS/BS operations.

Parallel Orthomin and Orthodir codes were developed for both 1D- and 2D-partitioned sparse coefficient matrices. The row-parallel [1, 19, 20] and the row-column-parallel [2, 17, 19] SpMM algorithms were utilized for the former and latter parallelization schemes. The parallel codes follow the steps of the sequential MATLAB code that is provided by NLAFET. The relative performance of the above-mentioned four different partitioning techniques were evaluated by speedup values, obtained through running the parallel Orthomin and Orthodir codes on two distributed memory systems, Sariyer and Juwels, with up to 1024 processors for 26 symmetric positive definite matrices from SparseSuite Matrix Collection [5]. The number of rows in these matrices ranges between 13,681 and 1,391,349, whereas the number of nonzero entries ranges between 340,431 and 77,651,847.

2. Block Conjugate Gradient Variants: Orthomin and Orthodir

Algorithm 1 Block CG: orthomin

```
Require: $A, B, X, \epsilon$
 1: $R \leftarrow B - AX$
 2: $P \leftarrow A - R$
 3: $Q \leftarrow AP$
 4: $[P, Q] \leftarrow$ A_Chol_QR$(P, Q)$        ▷ orthonormalize $P$ and $Q$
 5: while $||R|| > \epsilon$ do
 6:     $\alpha \leftarrow P^T R$        ▷ DMMT
 7:     $X \leftarrow X + P\alpha$        ▷ DMU
 8:     $R \leftarrow R - Q\alpha$        ▷ DMU
 9:     $\beta \leftarrow Q^T R$        ▷ DMMT
10:     $P \leftarrow R - P\beta$        ▷ DMU
11:     $Q \leftarrow AP$        ▷ SpMM
12:     $[P, Q] \leftarrow$ A_Chol_QR$(P, Q)$        ▷ orthonormalize $P$ and $Q$
```

Algorithm 2 Block CG: orthodir

```
Require: $A, B, X, \epsilon$
 1: $R \leftarrow B - AX$
 2: $P \leftarrow A - R$
 3: $Q \leftarrow AP$
 4: $[P, Q] \leftarrow$ A_Chol_QR$(P, Q)$        ▷ orthonormalize $P$ and $Q$
 5: $Q_p \leftarrow 0$
 6: $P_p \leftarrow 0$
 7: while $||R|| > \epsilon$ do
 8:     $\alpha \leftarrow P^T R$        ▷ DMMT
 9:     $X \leftarrow X + P\alpha$        ▷ DMU
10:     $R \leftarrow R - Q\alpha$        ▷ DMU
11:     $\beta \leftarrow Q^T Q$        ▷ DMMT
12:     $\gamma \leftarrow Q_p^T Q$        ▷ DMMT
13:     $Q_p \leftarrow Q$        ▷ memcopy
14:     $Q \leftarrow Q - P\beta$        ▷ DMU
15:     $Q \leftarrow Q - P_p\gamma$        ▷ DMU
16:     $P_p \leftarrow P$        ▷ memcopy
17:     $P \leftarrow Q$        ▷ memcopy
18:     $Q \leftarrow AP$        ▷ SpMM
19:     $[P, Q] \leftarrow$ A_Chol_QR$(P, Q)$        ▷ orthonormalize $P$ and $Q$
```

Algorithm 3 A_Chol_QR

```
Require: $P, Q$
 1: $C \leftarrow P^T Q$        ▷ DMMT
 2: $L \leftarrow$chol$(C)$        ▷ decompose (Cholesky) C into its triangular factors ($C = LL^T$)
 3: $P \leftarrow P/L$  $(LL^T P^T = P^T)$        ▷ triangular solve with multiple RHS (for each row of P)
 4: $Q \leftarrow Q/L$  $(LL^T Q^T = Q^T)$        ▷ triangular solve with multiple RHS (for each row of Q)
 5: return $P, Q$
```

Algorithms 1 and 2 display the pseudocodes for the Orthomin and Orthodir algorithms, respectively. Algorithm 3 displays the pseudocode for the orthonormalization operation performed in both Orthomin and Orthodir algorithms. These pseudocodes are important to understand the basic computation steps that are repeatedly performed at each iteration of Orthomin and Orthodir algorithms. There are five different basic operations in Orthomin and Orthodir algorithms, which are summarized in Table 1.

Table 1: Basic operations performed at each iteration of Ortomin and Orthodir algorithms. A is a $n \times n$ sparse matrix that contains nnz(A) nonzeros. X, Y and Z are dense matrices of size $n \times t$. α, β, and C are dense block matrices of size $t \times t$, where $t \ll n$.

abbreviation	full name	form	# of flops	omin	odir
SpMM	sparse matrix dense matrix multiply	$Y \leftarrow AX$	$2 \cdot t \cdot \text{nnz}(A)$	1	1
DMU	dense matrix update operation	$Z \leftarrow Z + \alpha X$	$n \cdot t^2$	3	4
DMMT	dense matrix matrix transpose multiply	$\beta \leftarrow X^T Y$	$2 \cdot n \cdot t^2$	3	4
chol	cholesky decomposition into triangular factors	$C = LL^T$	$t^3/3$	1	1
FS/BS	forward and backward substitution	$LL^T X^T = X^T$	$\approx 2 \cdot n \cdot t^2$	1	1

SpMM can be formulated as $Y \leftarrow AX$, where A is an $n \times n$ sparse matrix whereas X and Y are $n \times t$ dense matrices. Here, t is assumed to be much smaller than n. SpMM is repeatedly performed in line 11 of Algorithm 1 and in line 17 of Algorithm 2.

DMU is used in updating the descent matrix P, residual matrix R and solution matrix X at each iteration. The DMU operation can be formulated as $Z \leftarrow Z + \alpha X$. Here, Z and X are dense matrices of size $n \times t$, whereas α is a block matrix of size $t \times t$. The DMU operation involves the multiplication of an $n \times t$ matrix by a $t \times t$ block matrix followed by a linear matrix addition/subtraction operation for the update. DMU is performed in lines 7, 8 and 10 of Algorithm 1 and in lines 9, 10, 14 and 15 of Algorithm 2.

DMMT is used in computing the block matrices α, β, and γ . The DMMT operation can be formulated as $\beta \leftarrow X^T Y$. Here, X and Y are dense matrices of size $n \times t$, whereas β is a block matrix of size $t \times t$. DMMT is performed in lines 6, 9 and 12 of Algorithm 1, where the last one is actually inside the orthonormalization step, which can be seen in line 1 of Algorithm 3. In Algorithm 2, DMMT is performed in lines 8, 11, 12 and 18, where the last one is again inside the orthonormalization.

Both orthomin and orthodir algorithms involve the A-orthonormalization operation at the end of each iteration. This orthonormalization operation involves the cholesky decomposition of the $t \times t$ block matrix $C = P^T Q$ into its lower and upper triangular factors. Then, these triangular factors are used to update P and Q matrices through FS/BS operations.

In Table 1, the "# of flops" column shows the complexity of the above-mentioned basic operations in terms of the number of floating point operations performed. The "omin" and "odir" columns display the number of times these basic operations are performed at each iteration of Orthomin and Orthodir algorithms, respectively.

Table 2 summarizes the communication and computational-load-balancing requirements of the basic operations SpMM, DMU, DMMT, and FS/BS, when they are performed in parallel. We should note here that Cholesky factorization of the $t \times t$ block matrix C is performed sequentially at each processor. Since $t \ll n$ this is a reasonable implementation which is not expected to disturb scalability for large scale matrices.

Table 2: Communication and computational-load-balancing requirements of the basic operations.

abbreviation	communication	computational load balancing
SpMM	irregular P2P communication	number of nonzeros
DMU	no communication (if conformal)	number of rows
DMMT	collective communication	number of rows
chol	no communication (sequential)	–
FS/BS	no communication (if conformal)	number of rows

In SpMM, the irregular sparsity pattern of A results in irregular point-to-point communication for transferring either X-matrix rows or partial results for Y-matrix rows, respectively, before or after the local SpMM computations. Since the flop counts in SpMM is proportional to the number of nonzeros in matrix A, the balance on computational loads of processors during SpMM is achieved by obtaining balance on the number of A-matrix nonzeros assigned to processors.

For avoiding redundant communication in DMU and FS/BS operations, it is common practice to partition rows of input and output $n \times t$ matrices of SpMM operation in a conformal way. That is, the processor that owns row i of Z also owns row i of X. Provided that the partitions of all other $n \times t$ dense matrices in parallel Orthomin and Orthodir algorithms are conformal, DMU and FS/BS operations are communication-free. The flop counts in DMU, DMMT, and FS/BS operations are proportional to the number of rows in the dense matrices involved, hence, the balance on computational loads of processors during DMU, DMMT, and FS/BS is achieved by obtaining balance on the number of dense-matrix rows assigned to processors. In DMMT with conformal partitions of X and Y, processors do not need to communicate anything before the local computation starts. However, each processor produces a partial result for β of size $t \times t$, which are then reduced to the final value by an all-to-all reduction. This collective communication operation is unavoidable.

Recall from Table 1 that a single Orthomin iteration contains one SpMM, three DMUs, three DMMTs, and one FS/BS, whereas a single Orthodir operation contains one SpMM, four DMUs, four DMMTs, and one FS/BS. Among these four different operations, SpMM is considered to be the most expensive one, however, since DMU and DMMT operations are repeated at least three times as well as FS/BS operation performed once, while SpMM is performed once, they may have a detrimental effect on the overall performance if they are not optimized at all. For achieving the best performance in the parallelization of Orthomin and Orthodir algorithms, one should optimize the communication costs of SpMM while balancing the computational loads of processors for both SpMM and DMU-DMMT-FS/BS operations.

3. Partitioning Techniques for Parallel Orthomin and Orthodir Algorithms

In this section, we describe different techniques that we use for parallelizing the Orthomin and Orthodir algorithms. Before describing these techniques, first, we give some insights on their motivations.

The irregular memory access pattern during SpMM necessitates distributing the sparse matrix among processors carefully. Graph/hypergraph partitioning models [4, 12, 13] have been used for partitioning sparse matrices for decades so that the matrix elements (rows/columns or even nonzeros) with similar memory access patterns are assigned to the same processor to exploit the locality. Using these models, the communication overhead is almost always reduced significantly compared to utilizing a simple uniform block partitioning. The partitions obtained by these models are also well-balanced in terms of processors' computational loads since partitioning constraint of maintaining balance on the part weights correctly encode computational load balance via proper vertex weighting.

The communication overhead can and should be defined by multiple communication cost metrics. For parallel SpMM operation, in which the pattern of the communication can be as irregular as the sparsity pattern of the input sparse matrix, defining the communication by multiple cost metrics is especially important. One metric is the bandwidth cost, which is associated with the communication volume. Another metric is the latency cost, which is associated with the number of messages. The number of columns of the dense block matrices in the Orthomin and Orthodir algorithms, i.e., t value, may dramatically affect the overall communication cost. As t value increases, since the rows of dense matrices that are communicated during the parallel SpMM operation become more loaded, the bandwidth cost becomes important again. Therefore, depending on the sparsity pattern of the input matrix as well as block size t, one may have to consider both bandwidth- and latency-based communication cost metrics in the partitioning models used for Orthomin and Orthodir algorithms.

Recall that for SpMM operation, balance should be maintained on the number of nonzeros, whereas for DMU, DMMT and FS/BS operations, it should be maintained on the number of rows. These two constraints often conflict with each other. Considering a sparse matrix A whose rows have high variability in the number of nonzeros, a row-balanced partition of A exhibits a poor nonzero balance. In a dual manner, a nonzero-balanced partition exhibits a poor row balance. It is possible to obtain balance on both row and nonzero counts, thanks to the multi-constraint partitioning tools such as METIS [12]. However, since we have more constraints in the optimization problem solved by the partitioner, the solution quality is not as good as the case where there is only one constraint. That is, the total communication volume (the quality metric of the partitioner), when both row and nonzero counts are balanced and it is generally higher than the communication volume, when only row counts or nonzero counts are balanced. Therefore, while obtaining balance on both it is expected to perform better for a computation-bound instance. Obtaining balance on only one is expected to perform better for a communication-bound instance.

We consider four partitioning techniques that are different from each in terms of the above-mentioned communication-related and computation-related issues. All these methods rely on graph partitioning. Sections 3.1., 3.2., 3.3., and 3.4. describe these methods in detail.

3.1. 1D-sc: One-Dimensional Single-Constraint Partitioning

In this method, the rows of the given $n \times n$ sparse matrix A is partitioned into K parts/processors using single-constraint graph partitioning. First, an undirected graph $G = (V, A)$ is formed with vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set $E = \{e_{i,j} : a_{i,j} \neq 0 \text{ and } a_{j,i} \neq 0\}$. Each row i of A is represented by a vertex v_i in V. In a similar manner, nonzero entries $a_{i,j}$ and $a_{j,i}$ are represented by an edge $e_{i,j}$ in E if $i \neq j$. Note that since matrix A is symmetric we have $a_{i,j} = a_{j,i}$ and so $a_{i,j} \neq 0$ iff $a_{j,i} \neq 0$. The diagonal nonzero entries of A are not represented by edges in G. In this method, we maintain balance only on the number of nonzeros assigned to parts/processors. So, each vertex v_i in V is assigned a weight $w(v_i)$ which is equal to the number of nonzeros in row i. Each edge $e_{i,j}$ is associated with unit a cost, that is, $c(e_{i,j}) = 1$.

Consider a K-way partition $\Pi = \{V_1, V_2, \ldots, V_K\}$ of G. An edge $e_{i,j}$ is said to be cut in Π if v_i and v_j are assigned to different parts. Then the cutsize of Π is defined as the sum of the costs of the cut edges in Π, that is,

$$cutsize(\Pi) = \sum_{e_{i,j} \text{ is cut}} c(e_{i,j}).$$

The weight of part $V_k \in \Pi$ is defined as the sum of the weights of vertices in part V_k, that is, $W(V_k) = \sum_{v_i \in V_k} w(v_i)$. The graph partitioning problem is defined as finding a balanced partition Π of a given G with the objective of minimizing the cutsize. The partitioning constraint, which requires balanced parts, is formulated as

$$W(V_k) < W_{avg}(1+\epsilon) \text{ for } k = 1, 2, \ldots, K,$$

where W_{avg} denotes average part weight and ϵ denotes a maximum allowable imbalance ratio.

A K-way partition Π of G can be utilized to distribute the rows of A among K processors so that part V_k corresponds to the kth processor for $k = 1, 2, \ldots, K$. That is, row i of A is assigned to processor k if and only if $v_i \in V_k$. Since we utilize conformal partitions, row partitions of the input and output dense matrices of SpMM are also the same as the row partition of A, that is the ith rows of A, Y, and X are all assigned to processor k if $v_i \in V_k$. Then, the partitioning objective of minimizing the cutsize of Π relates to minimizing the total communication volume of parallel SpMM. The partitioning constraint of maintaining balance on the part weights corresponds to maintaining balance on the number of nonzeros assigned to processors, hence on processors' computational loads during parallel SpMM.

3.2. 1D-mc: One-Dimensional Multi-Constraint Partitioning

This method is very similar to 1D-sc. The only difference is that we maintain balance on two quantities instead of one, that is, we have two partitioning constraints. The first one is the same as the partitioning constraint of 1D-sc, that is, we maintain balance on the number of nonzeros when we assign rows to processors. The second constraint in 1D-mc deals with the numbers of the rows itself, instead of the nonzeros they contain. That is, we also maintain balance on the number of rows assigned to processors. Although maintaining balance on this quantity is not important for SpMM's parallel performance, since we utilize conformal partitions, it determines the balance achieved on the number of dense-matrix rows assigned to processors, which is important for the parallel performances of DMU, DMMT, and FS/BS.

In this method, we utilize the same graph partitioning model used in 1D-sc, except for the following difference. To be able to achieve balance on both, the number of nonzeros and the number of rows, we add an additional constraint to the graph partitioning problem. For this purpose, vertices are assigned a secondary weight, which is a unit weight, to encode the number or rows assigned to processors. Then, the first and second weights of vertex v_i are formulated as follows:

$$w^1(v_i) = |\{a_{i,j} \neq 0\}| \quad \text{and} \quad w^2(v_i) = 1.$$

The cth constraint of the two-constraint graph partitioning problem can be formulated as

$$W^c(V_k) < W^c_{avg}(1+\epsilon^c) \quad \text{for} \quad k = 1, 2, \ldots, K.$$

Here, $W^c(V_k)$ denotes the sum of the cth weights of vertices in part V_k, whereas W^c_{avg} denotes the average part weight for the cth weight. In some partitioning tools such as METIS [12], it is possible to use different maximum allowable imbalance ratios for different constraints, hence the maximum allowable imbalance ratio for the cth constraint is denoted by ϵ^c.

Since 1D-mc has an additional constraint compared to 1D-sc, its cutsize and hence the total communication volume of parallel SpMM is expected to be higher than that of 1D-sc.

3.3. 2D-sc: Two-Dimensional Single-Constraint Partitioning

In this method, instead of assigning all nonzero entries in a row to a single processor, we assign them to multiple processors. Since the nonzero partitioning of the input sparse matrix A is along both row and column dimensions, this method is called two-dimensional. We utilize the method proposed in [3] to obtain the 2D partition of A. It is a two-phase method where a 1D partitioning is obtained in the first phase and it is converted into a 2D one in the second phase. In the first phase of 2D-sc, we use the 1D partitioning obtained in 1D-sc. At the end of the first phase, the input matrix A can be reordered to form A^π in a K-way block-structure form. To do so, the rows that are represented by the vertices that belong to the same part in Π are reordered consecutively in A^π. Fig. 1 displays a 6-way partition of the graph (on the left) and the reordered matrix A^π in 6-way-block-structure form (on the right). As seen in the figure, each part/processor is shown in a different color. The dashed arrow lines in this figure correspond to the messages that yellow processor sends. Note that it can send at most $K-1$ messages with this 1D partitioning layout.

The second phase of 2D-sc redistributes the nonzeros in the off-diagonal blocks of A^π in a cyclic manner. To do so, it assumes that the K processors in the parallel system is organized as a virtual 2D mesh of processors of size $K_r \times K_c$, where K_c and K_r respectively denote the number of rows and columns in the processor mesh. Then it assigns the blocks of matrix A^π to processors according to their coordinates in the mesh, where A^π is the reordered matrix that can be obtained after applying 1D-sc. Fig. 2 displays how the blocks of A^π given in Fig. 1 are redistributed for a 2×3 virtual mesh of processors. Note that K_r horizontally-consecutive blocks are assigned to each processor in the corresponding row of the processor mesh and the assignments are repeated in a vertical manner.

In this 2D method, each processor is guaranteed to send at most K_r–1 messages for communicating the input dense matrix rows and to send at most K_c-1 messages for communicating the partial results for the output dense matrix rows. The dashed arrow lines in this figure show the messages sent by the yellow processor. For

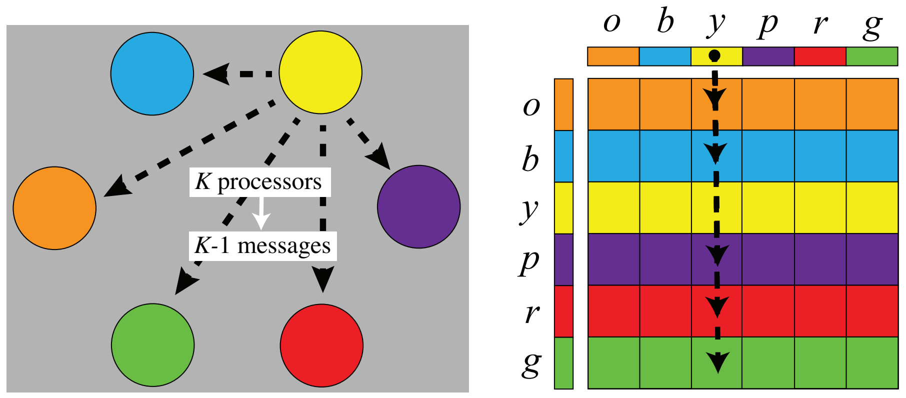

Fig. 1: Matrix A^{π} which is obtained by symmetrically reordering rows and columns of A according to a K-way partition Π obtained in the first phase.

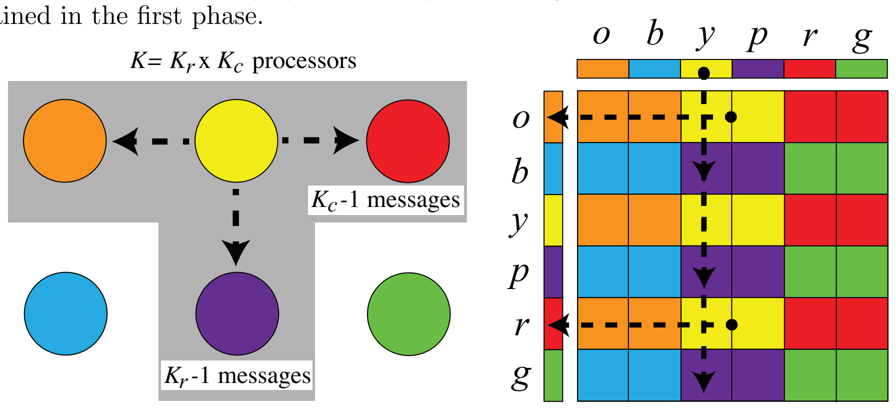

Fig. 2: The 2D partitioning of A^{π} for a 2×3 mesh.

communicating the input dense matrix rows, at worst, it sends a single ($K_r-1=1$) message, whose destination is the purple processor. For communicating the partial results for output dense matrix rows, at worst, it sends two ($K_c-1=2$) messages, whose destinations are the orange and red processors. These numbers are the same for the messages that it receives. It can be generalized that each processor only communicates with the processors in the same column of the mesh for communicating the input data and only communicates with the processors in the same row of the mesh for communicating the output data.

For a $\sqrt{K} \times \sqrt{K}$ processor mesh, observe that the number of messages that a processor send/receives is bounded above by $\sqrt{K}-1$. Matrices with a very dense row/column generally suffer from a high maximum message count, when partitioned in a one-dimensional way, hence, they are latency-bound. For the one-dimensional partition of such matrices, the number of messages that a processor sends/receives can be as high as $K-1$, which then hinders the scalability. Therefore, the two-dimensional matrix partitioning that provides $O(\sqrt{K})$ upperbound on the number of messages, handled by a processor, is expected to be very beneficial for improving the parallel performance on the instances that are latency-bound.

Converting a 1D matrix partition to a 2D partition, for obtaining the nice upper bound of $O(\sqrt{K})$ on the message count per processor, comes at the following expenses. The computational balance obtained on the number of nonzeros in 1D-sc is not valid anymore in 2D-sc. It is because the rows are not assigned to the processors as a whole anymore, which was the assumption of 1D-sc in maintaining the computational balance. The total communication volume of 2D-sc may be higher than 1D-sc since there are two phases of communication in 2D-sc (one for input and one for output communication) in constrast to the single communication phase in 1D-sc (for input communication).

3.4. 2D-mc: Two-Dimensional Multi-Constraint Partitioning

This method is very similar to 2D-sc, except for the fact that in this one, we use 1D-mc in the first phase instead of 1D-sc. So, 2D-mc has balanced row counts while 2D-sc does not. Note that for the 2D methods, it is not guaranteed that processors are assigned balanced number of nonzeros due to the conversion from 1D to 2D partitioning that takes place in the second phase.

4. Experiments

In our experiments, we compare the four partitioning techniques discussed in Section 3. in terms of the speedup values obtained for parallel Orthomin and Orthodir codes. For this comparison, we consider sparse real-world problems obtained from SuiteSparse Matrix Collection [5].

Table 3: Properties of test matrices.

matrix	problem kind	nrows	nnzs	avg	min	max	sparsity (%)
gyro_m	duplicate model reduction	17,361	340,431	19.61	4	120	0.11295
cbuckle	structural	13,681	676,515	49.45	26	600	0.36145
gyro_k	duplicate model reduction	17,361	1,021,159	58.82	12	360	0.33880
gyro	model reduction	17,361	1,021,159	58.82	12	360	0.33880
vanbody	structural	47,072	2,336,898	49.65	6	232	0.10547
smt	structural	25,710	3,753,184	145.98	52	414	0.56780
pdb1HYS	weighted undirected graph	36,417	4,344,765	119.31	18	204	0.32761
thread	structural	29,736	4,470,048	150.32	48	306	0.50553
ship_001	structural	34,920	4,644,230	133.00	18	438	0.38086
shipsec8	structural	114,919	6,653,399	57.90	15	132	0.05038
bmw7st_1	structural	141,347	7,339,667	51.93	1	435	0.03674
shipsec1	structural	140,874	7,813,404	55.46	24	102	0.03937
ship_003	structural	121,728	8,086,034	66.43	18	144	0.05457
m_t1	structural	97,578	9,753,570	99.96	48	237	0.10244
shipsec5	structural	179,860	10,113,096	56.23	12	126	0.03126
x104	structural	108,384	10,167,624	93.81	30	324	0.08655
crankseg_1	structural	52,804	10,614,210	201.01	48	2,703	0.38068
crankseg_2	structural	63,838	14,148,858	221.64	48	3,423	0.34719
nd12k	2D/3D	36,000	14,220,946	395.03	126	519	1.09730
bundle_adj	computer vision	513,351	20,208,051	39.36	3	12,588	0.00767
Fault_639	structural	638,802	28,614,564	44.79	15	318	0.00701
nd24k	2D/3D	72,000	28,715,634	398.83	110	520	0.55393
inline_1	structural	503,712	36,816,342	73.09	18	843	0.01451
PFlow_742	2D/3D	742,793	37,138,461	50.00	1	137	0.00673
Serena	structural	1,391,349	64,531,701	46.38	15	249	0.00333
audikw_1	structural	943,695	77,651,847	82.28	21	345	0.00872

We performed our experiments on 64, 256, and 1024 processors, that is, $K \in \{64, 256, 1024\}$. Note that each considered processor count is a perfect square, which is more convenient for comparing our 2D methods against 1D methods. The tested block sizes, t values, are 4, 8, 16, and 32.

The parallel Orthomin and Orthodir codes were implemented in C using MPI for inter-processor communication. The parallel codes follow the steps of the sequential MATLAB code that is provided by NLAFET. We used Intel MKL's BLAS and LAPACK routines for performing the factorization during the orthonormalization step.

We performed our parallel experiments on two different systems. The first one, Sariyer, is a Tier-1 system where each node contains 28 cores (two Intel Xeon E5-2680 v4 CPUs) running at 2.40 GHz clock frequency and 128 GB memory. The nodes are connected by an InfiniBand FDR 56 Gbps network. On this system, we used ICC (version 17.0.4) with OpenMPI (version 3.0.1) and Intel MKL (version 2017.0.098). The second system, JUWELS, is a Tier-0 system where each node contain 48 cores (two Intel Xeon Platinum 8168 CPUs) running at 2.70 GHz clock frequency and 96 GB memory. The nodes are connected by an EDR InfiniBand, fat-tree interconnection network. On this system, we used GCC (version 8.2.0) with ParaStation MPI (version 5.2.1, based on MPICH v3) and Intel MKL (version 2019.0.117).

We used METIS [12] for partitioning the graphs in 1D-sc and 1D-mc methods. Recall that 2D-sc and 2D-mc use the partitions obtained by 1D-sc and 1D-mc, respectively, in their first phases. In 1D-sc, the maximum allowable imbalance ratio ϵ is set to 3%. Similarly, in 2D-mc, both ϵ values are set to 3%, that is $\epsilon^1 = \epsilon^2 = 0.03$.

In our experiments, we considered 26 test matrices obtained from SuiteSparse Matrix Collection [5]. All test matrices we considered are symmetric positive definite. Table 3 displays the properties of these matrices in the increasing order of the number of nonzero entries. As seen in the table, the number of rows in these matrices ranges between 13,681 and 1,391,349, whereas the number of nonzero entries ranges between 340,431 and 77,651,847. Note that these matrices exhibit various charasteristics in terms of average, minimum, and maximum numbers of nonzeros in a row, as well as the sparsity rate. Also note that all matrices in our dataset exhibit irregularity in their sparsity pattern to a certain extent.

Table 4 displays the average run times obtained by the sequential Orthomin and Orthodir algorithms on Sariyer. In the table, a running time value refers to the geometric average of the running times over all 26 matrices.

Table 4: Average sequential run times of Orthomin and Orthodir algorithms.

	run time (seconds)			
	t=4	t=8	t=16	t=32
Orthomin	0.051	0.072	0.138	0.348
Orthodir	0.056	0.084	0.169	0.437

Table 5: Speedup values obtained by the parallel Orthomin algorithm averaged over all 26 matrices.

		speedup values				normalized values w.r.t. 1D-sc		
K	t	1D-sc	1D-mc	2D-sc	2D-mc	1D-mc	2D-sc	2D-mc
64	4	39.63	39.83	40.29	40.71	1.01	1.02	1.03
	8	39.74	40.13	40.93	42.01	1.01	1.03	1.06
	16	39.91	40.55	41.58	42.72	1.02	1.04	1.07
	32	40.69	42.51	41.34	42.99	1.04	1.02	1.06
256	4	51.19	56.16	64.97	63.08	1.10	1.27	1.23
	8	60.91	65.00	68.33	71.92	1.07	1.12	1.18
	16	74.65	76.78	84.21	85.62	1.03	1.13	1.15
	32	89.45	94.88	96.42	101.98	1.06	1.08	1.14
1024	4	49.85	53.27	74.61	69.62	1.07	1.50	1.40
	8	68.73	66.61	89.65	85.37	0.97	1.30	1.24
	16	99.83	104.91	113.53	118.39	1.05	1.14	1.19
	32	132.02	143.43	150.22	150.00	1.09	1.14	1.14
average values for different K values								
$K = 64$		39.99	40.75	41.03	42.11	1.02	1.03	1.05
$K = 256$		69.05	73.21	78.48	80.65	1.06	1.14	1.17
$K = 1024$		87.61	92.06	107.00	105.84	1.05	1.22	1.21
average values for different t values								
$t = 4$		46.89	49.75	59.96	57.80	1.06	1.28	1.23
$t = 8$		56.46	57.25	66.30	66.43	1.01	1.17	1.18
$t = 16$		71.46	74.08	79.77	82.24	1.04	1.12	1.15
$t = 32$		87.39	93.61	95.99	98.32	1.07	1.10	1.13
average values for all								
all						**1.05**	**1.15**	**1.16**

Tables 5 and 6 display the average speedup values obtained by the parallel Orthomin and Orthodir algorithms, respectively, on Sariyer. Here, a speedup value for a given matrix and (K, t) pair is computed as the ratio of the parallel running time of the algorithm on K processors to the sequential running time both with the same block size of t. In the tables, a speedup value for a (K, t) pair refers to the geometric average of the speedup values over all 26 matrices. In these tables, columns 3, 4, 5, and 6 display the actual average speedup values obtained by 1D-sc, 1D-mc, 2D-sc, and 2D-mc, respectively, whereas columns 7, 8, and 9 display the average speedup values of 1D-mc, 2D-sc, and 2D-mc, respectively, normalized with respect to those of 1D-sc.

As seen in Table 5, for parallel Orthomin algorithm, 1D-sc, 1D-mc, 2D-sc, and 2D-sc achieve average speedup values of 132, 143, 150, and 150 for $t = 32$ on 1024 processors, respectively, averaged over all matrices. Note that, compared to 1D-sc, 1D-mc improves the parallel performance by 9% on average, whereas 2D-sc and 2D-mc improve it by 14%, on average. Note that the improvement obtained by the multi-constraint partitioning is more visible for 1D partitionings compared to 2D partitionings. On overall average, 1D-mc performs 5% better than 1D-sc, whereas 2D-mc performs only 0.9% better than 2D-sc.

In Table 5, on 64 processors, all the speedup values are around 40 although they are obtained by different methods and using different t values. This can be attributed to the coarse computational granularity in SpMM operation dominating the parallel runtime so that neither the change in communication requirements across 1D and 2D methods nor the change in computational balance requirements for DMU, DMMT, and FS/BS operations across single-constraint and multi-constraint methods affect the performance significantly. Nevertheless, on $K = 64$, it is still possible to observe that 2D and multi-constraint methods achieve better performance compared to 1D and single-constraint methods, respectively, by a slight amount. Note that the performance difference among different methods becomes more visible as the number of processors increases.

Recall that 2D methods are superior to 1D methods in terms of the latency cost. The results given in Table 5 are in agreement with that statement, that is, the performance gap between 2D and 1D methods increases in favor or 2D methods as the number of processors increases and t value decreases, i.e., latency cost becomes more important than the bandwidth cost. On average, the 2D models perform better than their 1D counterparts by up to 3% for $K = 64$, by up to 14% for $K = 256$, and by up to 22% for $K = 1024$. These average improvement rates are 28% for $t = 4$, 17% for $t = 8$, 12% for $t = 16$, an 10% for $t = 32$.

Recall that we expect multi-constraint methods to perform better than single-constraint ones for computational load balancing during DMU, DMMT, and FS/BS operations. The results given in Table 5 validate this statement for 1D methods on all t values, except for $t = 8$ on $K = 1024$ processors. However for 2D methods, there are more exceptions to this statement, e.g., for $t = 4$ and $t = 8$ on 1024 processors. This is probably because the balance achieved on DMU, DMMT, and FS/BS operations do not pay the increase in total communication volume for these settings.

Table 6: Speedup values obtained by the parallel Orthodir algorithm averaged over all 26 matrices.

		speedup values				normalized values w.r.t. 1D-sc		
K	t	1D-sc	1D-mc	2D-sc	2D-mc	1D-mc	2D-sc	2D-mc
64	4	40.01	39.83	40.26	40.39	1.00	1.01	1.01
	8	39.73	39.75	40.82	41.44	1.00	1.03	1.04
	16	40.45	41.33	40.91	42.23	1.02	1.01	1.04
	32	40.64	42.19	41.01	43.31	1.04	1.01	1.07
256	4	58.33	55.75	60.71	62.64	0.96	1.04	1.07
	8	69.20	67.92	68.51	68.77	0.98	0.99	0.99
	16	79.71	82.93	84.20	87.21	1.04	1.06	1.09
	32	93.00	98.32	98.52	104.38	1.06	1.06	1.12
1024	4	50.81	50.59	59.40	64.63	1.00	1.17	1.27
	8	70.44	73.65	80.86	91.84	1.05	1.15	1.30
	16	106.06	114.08	112.17	116.13	1.08	1.06	1.09
	32	138.43	154.71	141.26	159.50	1.12	1.02	1.15
average values for different K values								
$K = 64$		40.21	40.77	40.75	41.84	1.01	1.01	1.04
$K = 256$		75.06	76.23	77.98	80.75	1.02	1.04	1.08
$K = 1024$		91.44	98.26	98.42	108.03	1.07	1.08	1.18
average values for different t values								
$t = 4$		49.72	48.72	53.46	55.89	0.98	1.08	1.12
$t = 8$		59.79	60.44	63.39	67.35	1.01	1.06	1.13
$t = 16$		75.41	79.45	79.09	81.86	1.05	1.05	1.09
$t = 32$		90.69	98.41	93.60	102.40	1.09	1.03	1.13
average values for all								
all						**1.04**	**1.05**	**1.12**

As seen in Table 6, for parallel Orthodir algorithm, 1D-sc, 1D-mc, 2D-sc, and 2D-sc achieve average speedup values of 138, 155, 141, and 160 for $t = 32$ on 1024 processors, respectively, averaged over all matrices. Note that compared to 1D-sc, 1D-mc improves the parallel performance by 12% on average, whereas 2D-sc and 2D-mc improve it by 5% and 12%, respectively, on average.

Most of the observations made on the speedup values obtained by the Orthomin algorithm are also valid for the speedup values obtained by the Orthodir algorithm. The major differences are summarized as follows. Note that, in constrast to the Orthomin algorithm, the improvement obtained by the multi-constraint partitioning in Orthodir algorithm is almost the same for both 1D and 2D partitionings. On overall average, 1D-mc performs 4% better than 1D-sc and 2D-mc performs 6.5% better than 2D-sc.

Another interesting observation is that the performance improvement obtained by 2D-sc compared to 1D-sc on small t values and large K values is not as high in parallel Orthodir as in parallel Orthomin. Normalized speedup values of 1.50 and 1.30 obtained by 2D-sc on $K = 1024$ with $t = 4$ and $t = 8$ on Orthomin reduce to 1.17 and 1.15 for Orthodir, respectively. As the last observation, note that as t value increases, the improvement obtained by the multi-constraint partitioning becomes much more than the improvement obtained by the 2D partitioning. This is because latency cost becomes less important with increasing t value, whereas the computational balance on DMU, DMMT, and FS/BS operations becomes more important.

Fig 3 displays the speedup curves for parallel Orthomin algorithm on nine sample matrices from our dataset. As seen in the figure, the 2D methods generally scale the parallel Orthomin code better than the 1D methods. Matrices `bmw7st_1`, `crankseg_2`, `Fault_639`, and `Serena` constitute good examples for this observation. While single-constraint and multi-constraint methods may be interleaved for 1D and 2D methods on some matrices, it is generally the case that multi-constraint methods perform better than single-constraint methods. The exceptions for this observation are `inline_1` and `audikw_1`, where the best performer is 2D-sc.

Fig. 4 displays the speedup curves for parallel Orthomin on JUWELS on the same matrices given in Fig. 3. Observe that the parallel performances, obtained by the different partitioning methods, are closer to each other on JUWELS compared to Sariyer. This is especially true across 1D and 2D methods, that is, they behave similar on all matrices, except for `crankseg_2`. The decrease in the performance discrepancy between 1D and 2D methods can be attributed to the fast interprocessor network used in JUWELS, that is, improving the communication overhead on such a network does not considerably alter the parallel performance up to 1024 processors. The more visible performance gap exists across single-constraint and multi-constraint methods in favor of multi-constraint ones. Matrices `bmw7st_1`, `bundle_adj`, and `PFlow_742` constitute good examples for this gap. Recall that multi-constraint methods performing better than single-constraint methods is explained by the fact that they achieve balance on the computational loads of the processors on every kernel operation (SpMM, DMU, DMMT, and FS/BS), while single-constraint methods achieve it only on SpMM.

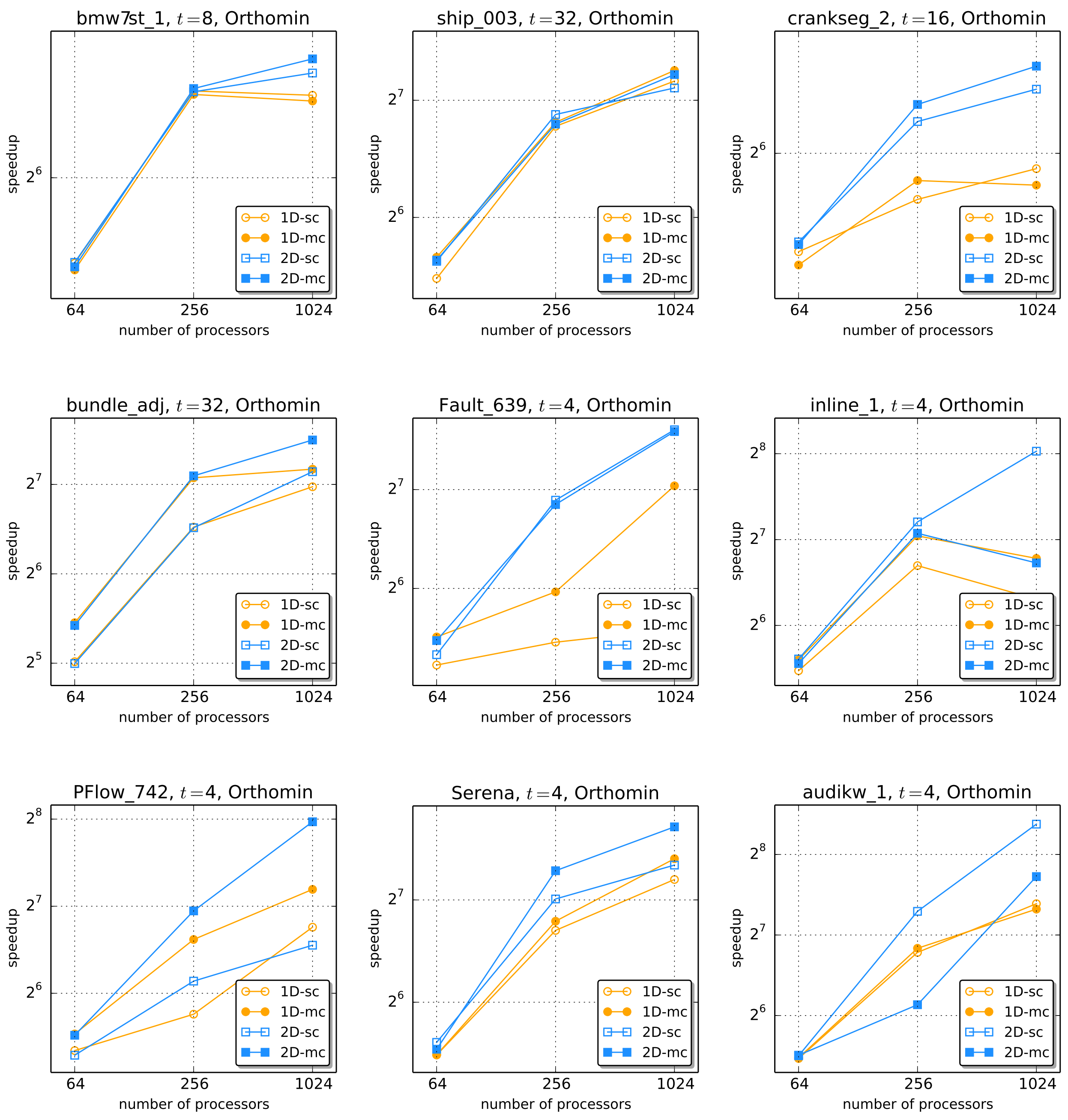

Fig. 3: Speedup curves of Orthomin comparing 1D-sc, 1D-mc, 2D-sc, and 2D-mc on Sariyer.

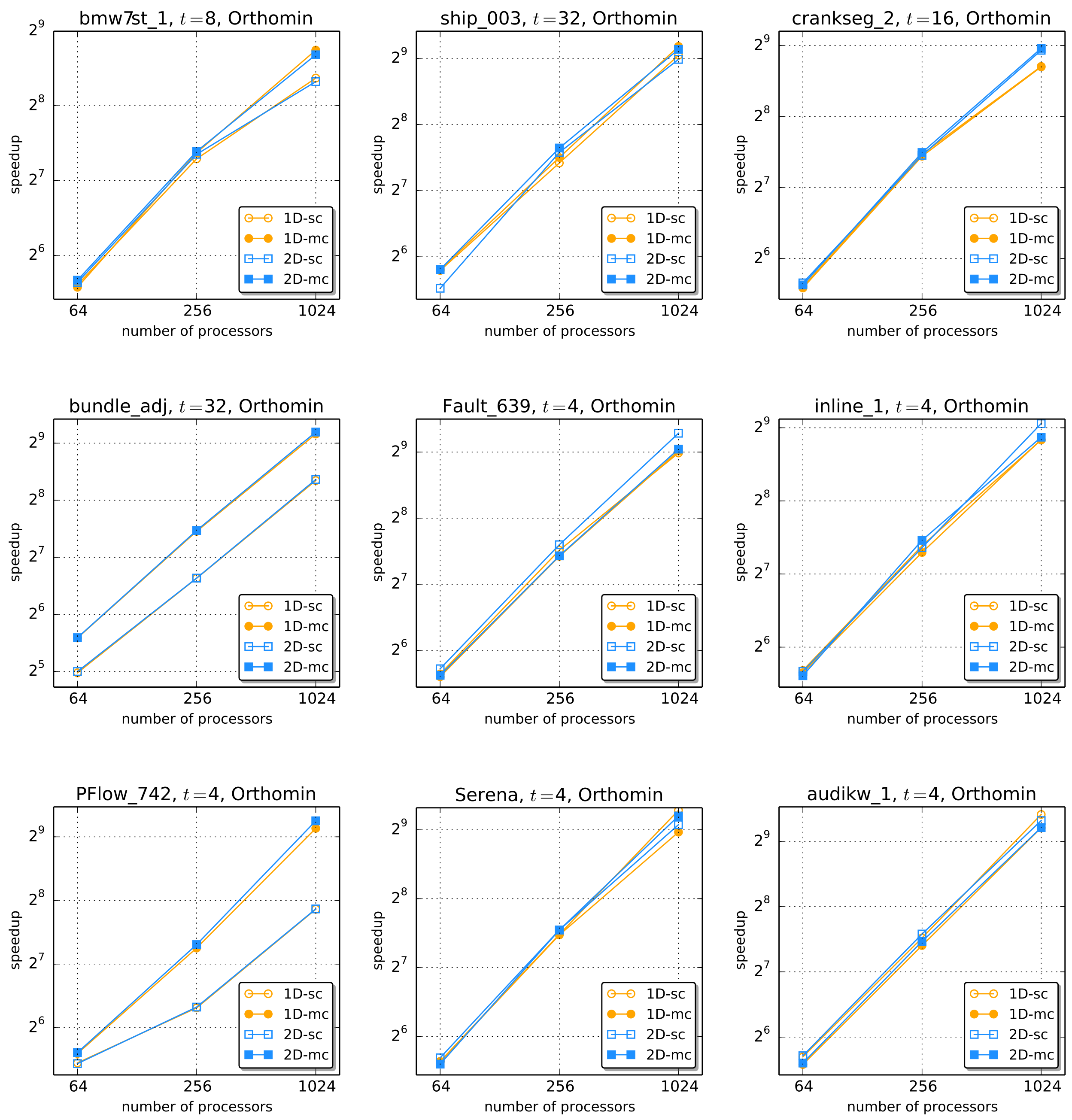

Fig. 4: Speedup curves of Orthomin comparing 1D-sc, 1D-mc, 2D-sc, and 2D-mc on JUWELS.

5. Conclusion

Scalable parallelization of block CG variants Orthomin and Orthodir, proposed by FET-HPC project NLAFET, is investigated to enable the use of these methods on future exascale systems. 1D- and 2D-partitioning models with single-constraint and two-constraint variants are realized for partitioning the sparse coefficient matrix as well as the dense matrices involved in the Ortodir and Orthomin iterations. Parallel Orthomin and Orthodir codes that utilize 1D and 2D sparse matrix partitioning models are developed and the performance of these methods are evaluated on two different HPC systems with four different partitioning techniques. Experimental findings show that encapsulating computational load balance during sparse matrix and dense matrix computations simultaneously via two-constraint partitioning formulation leads to better scalability. 2D-partitioning models are found to perform better in general for smaller block sizes and larger number of processors.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913. We acknowledge PRACE for awarding us access to resource JUWELS based in Germany at Jülich Supercomputing Centre (JSC). We acknowledge UHeM for awarding us access to resource Sariyer based in Turkey. We thank Olivier Tissot and Laura Grigori as they provided the sequential MATLAB implementation of Orthomin and Orthodir variants.

References

1. S. Acer, O. Selvitopi, and C. Aykanat. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Computing*, 59:71–96, 2016.
2. S. Acer, O. Selvitopi, and C. Aykanat. Optimizing nonzero-based sparse matrix partitioning models via reducing latency. *Journal of Parallel and Distributed Computing*, 122:145–158, 2018.
3. E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.
4. U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999.
5. T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.
6. E. De Sturler and H. A van der Vorst. Reducing the effect of global communication in gmres (m) and cg on parallel distributed memory computers. *Applied Numerical Mathematics*, 18(4):441–459, 1995.
7. EF DAzevedo, V Eijkhout, and CH Romine. Lapack working note 56 conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical report, Citeseer, 1999.
8. P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Technical Report 12.2012.1*, 2012.
9. P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.
10. L. Grigori, S. Moufawad, and F. Nataf. Enlarged krylov subspace conjugate gradient methods for reducing communication. *SIAM Journal on Matrix Analysis and Applications*, 37(2):744–773, 2016.
11. L. Grigori and T. Oliver. Reducing the communication and computational costs of Enlarged Krylov subspaces Conjugate Gradient. *Working Note 13, FET-HPC NLAFET*, 2017.
12. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
13. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.
14. G. Meurant. Multitasking the conjugate gradient method on the cray x-mp/48. *Parallel Computing*, 5(3):267–280, 1987.
15. Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.
16. Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.
17. O. Selvitopi and C. Aykanat. Reducing latency cost in 2d sparse matrix partitioning models. *Parallel Computing*, 57:1–24, 2016.

18. O. Selvitopi, M. M. Ozdal, and C. Aykanat. A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):632–645, March 2015.
19. B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.
20. B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 29(4):1683–1709, 2007.