

Diamond matrix powers kernels

Emil Vatai*

vatai@inf.elte.hu
emil.vatai@gmail.com
The University of Tokyo
Graduate School of Information
Science and Technology
Tokyo, Japan
ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Utsav Singhal

utsavsinghal5@gmail.com
The University of Tokyo
Graduate School of Information
Science and Technology
Tokyo, Japan
Indian Institute of Technology, Delhi
New Delhi, India

Reiji Suda

reiji@is.s.u-tokyo.ac.jp
The University of Tokyo
Graduate School of Information
Science and Technology
Tokyo, Japan

ABSTRACT

Matrix powers kernel calculates the vectors $A^k v$, for $k = 1, 2, \dots, m$ and they are the heart of various scientific computations, including communication avoiding iterative solvers. In this paper we propose diamond matrix powers kernel - DMPK, which has the purpose to apply the “diamond tiling” stencil algorithm to general matrices. It can also be considered as an extension of the PA1 and PA2 algorithms, introduced by Demmel et al. Our approach enables us to control the balance between the amount of communication avoidance and redundant computation inherently present in communication avoiding algorithms. We present a proof of concept implementation of the algorithm using MPI routines. The experiments we performed show that the control of the amount of computation and communication is achievable, and with more thorough optimisations, DMPK is a promising alternative to existing MPK approaches.

ACM Reference Format:

Emil Vatai, Utsav Singhal, and Reiji Suda. 2020. Diamond matrix powers kernels. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020)*, January 15–17, 2020, Fukuoka, Japan. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3368474.3368494>

1 INTRODUCTION

Iterative solvers are the de-facto standard to solve sparse linear equations. However, for large linear systems, which are meant to be solved on supercomputers, the performance is hampered by the inter-process communication. To tackle that problem, various modifications of Krylov subspace methods have been proposed [7]. Among them, communication-avoiding algorithms [6, 9, 11] employ aggressive techniques to reduce communication, such as step methods, Matrix Powers Kernel (MPK), Tall-Skinny QR (TSQR) and block Gram-Schmidt. In this paper, we focus on MPK, which

is also called AkX (for $A^k x$) in some literature [2], and is actively used in current research e.g. [1].

MPK calculates the m vectors obtained by multiplying m times a vector v by the (sparse) matrix A , i.e. the vectors: $v^{(0)}, v^{(1)}, \dots, v^{(m)}$ where $v^{(i)} = A^i v$ (with the initial vector $v = v^{(0)}$) as described in Algorithm 1. The obvious disadvantage of this naive implementation, is the requirement for communication in each iteration. Demmel et

Algorithm 1 MPK(v, A, m)

```
 $v^{(0)} \leftarrow v$   
for  $k = 1$  to  $m$  do  
   $v^{(k)} \leftarrow Av^{(k-1)}$ 
```

al. [6] proposes several variations of MPK. Here we only review two of them: PA1 and PA2. In PA1, each process computes a disjoint subset of $v^{(m)}$. Before that, the process computes the subset of $v^{(m-1)}$ that is referred to in the computation of $v^{(m)} = Av^{(m-1)}$. The subset of $v^{(m-1)}$ computed here is nothing but the process’s domain plus its halo region. In a conventional method, the elements in the halo region are computed by the processes they are assigned to, but in PA1, they are computed in duplication by multiple processors. Similar discussion is done from $v^{(m-2)}$ to $v^{(1)}$. The consequence is that the process computes all the elements of $v^{(1)}, v^{(2)}, \dots, v^{(m-1)}$ that have dependency to one or more elements in the finally computed subset of $v^{(m)}$. The required part of $v^{(0)}$ is sent to the processor before the computations (but some overlap of computation and communication is possible). In the PA1, communication is done only once at the beginning, and a significant part of computations is done in duplication by two or more processors. PA2 is a modification of PA1. The difference is that, each process first does as much local computations as possible without communication. The *partial results*, i.e. the vertices computed (locally) up to some level k (where $k < m$) are sent to the processes that need them, so a part of redundant computation is removed. However, the amount of duplicated computation is still significant. In our algorithm, the duplicated computations is reduced, but multiple steps of communication are needed. Thus ours is not absolutely better than PA1 and PA2, but we provide more freedom of trade-off to optimise performance.

MPK is closely related to the temporal blocking technique in stencil computations [14]. PA1 corresponds to the overlapped tiling. In stencil computations, there are several methods, such as the

*Emil Vatai is an International Research Fellow of Japan Society for the Promotion of Science (Postdoctoral Fellowships for Research in Japan (Standard)). This work was supported by JSPS Grant-in-Aid for JSPS Research Fellow Grant Number JP18F18786.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

HPCAsia2020, January 15–17, 2020, Fukuoka, Japan

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7236-7/20/01...\$15.00

<https://doi.org/10.1145/3368474.3368494>

diamond tiling, to reduce or totally remove duplicated computations in temporal blocking [3]. However, as far as the authors know, there is no known parallel MPK algorithm for general matrices except PA1 and PA2 (and the one presented here). In this paper, we propose an MPK that avoids most of the duplicated computations, corresponding to the diamond tiling in stencil computations. This work is an extension of our earlier work reported in [13].

The rest of the paper is organised as follows. Section 2 describes the problem and some previous research done to address it. In Section 3 we present our idea and give an overview of our algorithm. The details of our implementation are provided in Section 4. We describe the experiments performed in Section 5, and we present some results, how our algorithm effects the amount of communication and redundant computation. In Section 6 we state our conclusions and in Section 7 we describe our future work.

2 COMMUNICATION IN MPK

An (m dimensional) Krylov subspace, defined by the matrix A and vector v is:

$$\mathcal{K}_m(v, A) = \text{span}\{v, Av, A^2v, \dots, A^{m-1}v\}$$

The basic principle behind Krylov subspace projection methods for solving sparse linear systems is (in each iteration) to find the best approximation of the exact solution in a Krylov subspace, and then to expand the Krylov subspace [12]. Usually this is a one dimensional expansion, which requires one matrix multiplication: $A^m v \leftarrow A \cdot (A^{m-1}v)$ i.e. $v_{\text{new}} \leftarrow A \cdot v_{\text{old}}$. This is followed by an inner product operation, which demands communication, because all elements of the vector are needed. To avoid communication in such algorithms, various modifications have been developed, which expand the Krylov subspace by $m > 1$ dimensions (see [5]). These methods require m matrix multiplications, i.e. the vectors $Av, A^2v, \dots, A^m v$ need to be calculated. Hence the procedure to calculate these vectors is called matrix powers kernel.

In this paper, $A \in \mathbb{R}^{n \times n}$ is a general (sparse) matrix (stored in compressed sparse row format) with the component $a_{i,j}$ at the i -th row and j -th column of A and $v \in \mathbb{R}^n$ a (dense) column vector with v_i as its i -th component (for $0 < n \in \mathbb{N}$). Our goal is to calculate the m vectors $Av, A^2v, \dots, A^m v$, where m denotes a positive integer (usually $m > 1$). For reasons stated later, we should assume the pattern (i.e. position of nonzero elements) of A is symmetric. This condition is easy to satisfy by storing zeros in the missing position.

2.1 Graph representation

To better understand the data dependency in MPK, the problem should be reformulated in graph theoretical terms.

Adjacency matrices are a conventional way matrices and graphs are assigned to each other. This correspondence implies a natural labelling of the vertices of the graph with the components of a vector v of length n where n is the number of vertices of the graph.

Definition 2.1 (G graph). The graph $G = G(A, v)$ associated with the vector v and matrix A , has n vertices and e edges, where e is the number of nonzero elements in A . The i -th vertex of G is labelled with the value of v_i , and there is a directed edge from v_j to v_i if and only if $a_{i,j} \neq 0$, and this edge is labelled with $a_{i,j}$.

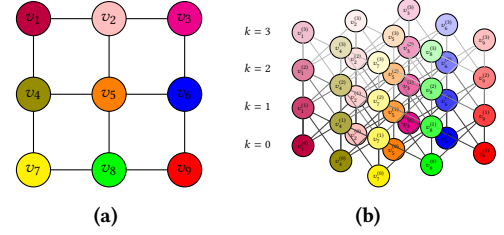


Figure 1: G and G_m graph of a 3×3 mesh

It can be useful to treat v_i both as the i -th component of the vector v , and the corresponding vertex in G . In this context, calculating a vertex v_i , means calculating the new label of that vertex, that is the value of v_i in the vector v .

This representation needs a simple extension to allow the visualisation of the flow of information in MPK: m new sets of vertices are needed (because each matrix multiplication by A produces a new vector), and the edges should be pointing to vertices which are calculated, from vertices which are needed to calculate them.

Definition 2.2 (G_m graph). The graph $G_m = G_m(v, A)$, consists of $(m + 1) \times n$ vertices labelled with $v_i^{(k)}$ (i.e. the i -th component of the vector $v^{(k)} = A^k v$, for $1 \leq i \leq n$ and $0 \leq k \leq m$) and $e \times m$ number of edges, each labelled with $a_{i,j}$ and directed from $v_j^{(k-1)}$ to $v_i^{(k)}$ for $a_{i,j} \neq 0$ (for each $1 \leq k \leq m$). In this case the vertex $v_j^{(k-1)}$ is the parent of $v_i^{(k)}$.

The vertices of $v^{(k)}$ in G_m are the k -th level of MPK, and the graph should be visualised with the vertices of $v^{(0)}$ at the bottom, and the vertices of $v^{(k)}$ for $k > 0$ as identical copies (with different vertex labels) stacked on each other. In this arrangement, the edges on top of each other have the same labels. In the matrix vector multiplication

$$v_i^{(k)} = \sum_{j=1}^n a_{i,j} v_j^{(k-1)} \quad (1)$$

we can see that in the sum the terms with $a_{i,j} = 0$ can be omitted, so to calculate $v_i^{(k)}$, we need to sum up all the vertices with an edge pointing to $v_i^{(k)}$ from level $k - 1$ and multiply them with the edge label. In the adjacency matrix G , the sum in (1) visits all the neighbours of the calculated element $v_i^{(k)}$. To calculate $Av, A^2v, \dots, A^m v$, MPK needs to “reach” the vertices at the top level. As an example, Figure 1a shows a 3×3 2D mesh and how the vector v of length $n = 9$ is associated to the vertices, while Figure 1b shows the data dependency of MPK with $m = 3$ (the edges are directed from lower levels i.e. lower values of k to higher levels with higher values of k). The directed graph G_m represents the flow of information in MPK. The edges represent a floating point multiplication and an addition. G can be useful for a simpler discussion of our algorithm.

2.2 Parallel computation and communication in MPK

The basic approach to communication avoiding MPK roughly follows the classical divide and conquer scheme: the vertices of G are

partitioned and each process is assign one partition to compute. Henceforth, Π will denote a partitioning of the set $\{1, 2, \dots, n\}$, i.e. $\cup \Pi = \{1, 2, \dots, n\}$ and for each $p, q \in \Pi$, if $p \neq q$ then $p \cap q = \emptyset$. For each partition $q \in \Pi$, if $i \in q$ we say $v_i^{(k)}$ belongs to partition q and sometimes we may abuse notation and write $v_i^{(k)} \in q$ or $v_i \in q$ (for each k). This implies a partitioning of G and G_m . If P is the number of available computation nodes, then usually $|\Pi| = P$ is chosen, and each partition is assigned a process number between 1 and P , and in this context, partitions and processes are interchangeable.

Given a partitioning Π , two concepts will be useful: the set of required vertices and the set of locally computable vertices. Let $d(i, j)$ denote the distance between vertices v_i and v_j in G .

Definition 2.3 (Required vertices). Given a partition $q \in \Pi$ and m , the set of *required vertices* (to compute MPK to level m) is defined as

$$R(q, m) = \{i : \exists j \in q \text{ such that } d(i, j) \leq m\}$$

This is the set of indices of the input vector $v^{(0)}$ which need to be sent the partition q to compute the vertices up to level m if no partial results are computed.

Definition 2.4 (Halo of a partition). Given a partition $q \in \Pi$ and m , the *halo* (or *skirt*) of the partition (to compute MPK to level m) is

$$H(q, m) = \{(i, k) : \exists j \in q \text{ s.t. } d(i, j) \leq m \wedge k > 0\}$$

These are the (i, k) pairs, such that $v_i^{(k)}$ elements are calculated when the vertices in q are calculated up to level m (again assuming no partial results computed). PA1 transfers $R(q, m)$ to each partition q and calculates elements corresponding to $H(q, m) \setminus (q \times \mathbb{N})$ redundantly.

Definition 2.5 (Locally computable vertices). Given a partition $q \in \Pi$ the set of *locally computable vertices* is defined as

$$L(q) = \{(i, k) : i \in q \wedge d(i, j) \leq k \implies j \in q \text{ is true for } \forall j\}$$

These are the (i, k) pairs, such that $v_i^{(k)}$ elements can be calculated from $v_j^{(0)}$ elements where $j \in q$.

Given a set $X \subset \mathbb{N} \times \mathbb{N}$, let $\ell_i(X)$ be $\max\{k : (i, k) \in X\}$ if there is a k such that $(i, k) \in X$ or 0 otherwise. Applying ℓ to $L(q)$, we obtain the *levels* of vertices reached, i.e. if $\ell_i(L(q)) = k$ then $v_i^{(k')}$ is computed/computable in q for $k' \leq k$. The levels of all partitions can be considered simultaneously, and we can use a global ℓ sequence such that $\ell_i = \sum_{q \in \Pi} \ell_i(L(q))$ which is the same as $\ell_i(\cup_{q \in \Pi} L(q))$, given a partitioning Π , every vertex $v_i^{(k)}$ can be calculated, for $k \leq \ell_i$ using only local vertices.

Now we can define the halo and required vertices with partial results computed.

Definition 2.6 (Halo with partial results). Given a partition $q \in \Pi$, a sequence ℓ ($0 \leq \ell_i \leq m$ for $1 \leq i \leq n$) and the integer $m > 1$, the halo (or skirt) of the partition (to compute MPK to level m , with partial results specified by ℓ) is the set

$$H(q, m, \ell) = \{(i, k) : \exists j \in q \text{ s.t. } d(i, j) - \ell_j \leq m \wedge \ell_j < k \leq m - d(i, j)\}$$

Definition 2.7 (Required vertices with partial results). Given a partition $q \in \Pi$, a sequence ℓ ($0 \leq \ell_i \leq m$ for $1 \leq i \leq n$) and the

integer $m > 1$, the required vertices partition to compute MPK to level m , with partial results specified by ℓ is the set

$$R(q, m, \ell) = \{(i, k) : \exists(j, k+1) \in H(q, m, \ell) \text{ s.t. } k \leq \ell_i \wedge d(i, j) \leq 1\}$$

These are extensions of Definition 2.4 and Definition 2.3 for the case when we have calculated vertices $v_i^{(k)}$ for $k \leq \ell_i$. It should be noted, that Definition 2.3 of required vertices is a subset of indices, while Definition 2.7 is a set (i, k) pairs, since the same vertex at different level might be required, but the idea of an extension still holds, by adding a zero component to the elements of $R(q, m)$ i.e. to take $(i, 0)$ instead of i when $i \in R(q, m)$.

The PA2 algorithm, given a partitioning Π , calculates the vertices up to level $\ell_i = \ell_i(L(q))$ in the process/partition q using only vertices available in that partition, and then computes the vertices defined by $H(q, m, \ell)$ by obtaining the vertices of $R(q, m, \ell)$ from the other partitions. Both PA1 and PA2 have redundant computation which we tend to reduce in our approach, described in the following section.

3 DIAMOND MATRIX POWERS KERNEL

The PA2 algorithm can be considered as an approximation of the diamond tiling algorithm for general matrices, obtained from PA1 by extending the definition of halo and required vertices to consider partial results, i.e. vertices which can be computed up to some intermediate level k (where $k < m$).

Following this idea, the two ingredients needed by diamond tiling for general matrices, i.e. our algorithm, are:

- (1) the extension of locally computable vertices to include partial results, and
- (2) the moving index domains of the diamond tiling scheme (reassignment of indices to different processes at different levels/phases of the computation).

Definition 3.1 (Locally computable vertices with partial results). Given a partition $q \in \Pi$, a sequence ℓ ($0 \leq \ell_i \leq m$ for $1 \leq i \leq n$) the set of *locally computable vertices* (with partial results specified by ℓ) is defined as

$$L(q, \ell) = \{(i, k) : i \in q \wedge d(i, j) - \ell_j \leq k \implies j \in q \text{ is true for } \forall j\}$$

Similarly to the diamond tiling algorithm, DMPK also runs in phases: after each phase diamond tiling moves the index domains. This can be very efficient for stencil computations because of the regular access pattern, but for general matrices, this movement of the index domain corresponds to the repartitioning of the vertices. With the additional restriction, to make the first and last partitioning the same, the PA1 and PA2 are the special cases of DMPK with zero or one startup phase ($s = 0$ or 1) and the final (halo/skirt) phase. A rough description of DMPK is given in Algorithm 2.

Line 8 and 4 deserve some explanation. After calculating $L(q, \ell)$ the vertices in q (for each partition q) are further advanced, to higher levels, and after this operation ℓ should be updated to these new, higher levels.

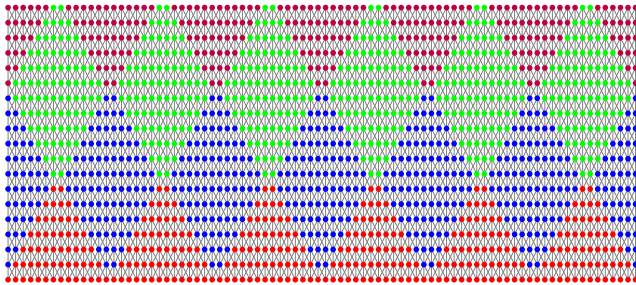
Generating a new partitioning corresponds to moving the index domains in stencil computations, and deserves some explanation. Phases and moving domains come naturally to stencil computations because of their regular access pattern, while it is not so clear-cut for general matrices. Diamond tiling can fit the diamond tiles naturally

Algorithm 2 $\text{DMPK}(v, A, m, s)$

```

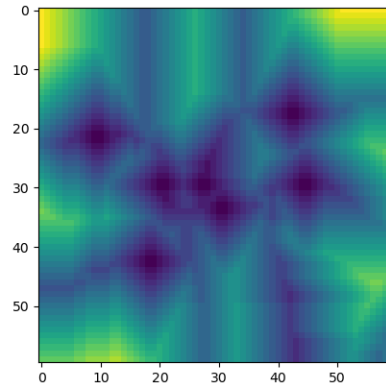
1:  $v^{(0)} \leftarrow v$ 
2:  $\ell_i \leftarrow 0$  (for all  $i$ )
3: for  $r = 0$  to  $s - 1$  do
4:   Generate a new  $\Pi_r$  partitioning
5:   Perform communication
6:   for  $q \in \Pi_r$  do
7:     Calculate vertices in  $L(q, \ell_i)$ 
8:     Update  $\ell_i$ 
9:   Perform communication
10:  for  $q \in \Pi_0$  do
11:    Calculate  $H(\Pi_0, m, \ell)$ 

```

**Figure 2: Iterations resembling stacked diamonds**

onto each other, but this is not possible for general matrices. Figure 2 shows these “diamond tiles” of a tridiagonal matrix and what we call phases of the algorithm. The vertices at different levels are arranged above each other. The initial phase is red (these vertices are calculated when $r = 0$ in Algorithm 2), followed by blue ($r = 1$), then green ($r = 2$) and finally purple (the final halo phase). The “moving index domains” are defined by the left and right sides of the diamond shapes: basically in this example of a tridiagonal matrix, the domain alternates between two positions, shifted by half on diamonds width.

A different approach needs to be taken, to see what needs to be done for general matrices. The initial phase, when the algorithm calculates only “tops” of the diamonds i.e. the red triangles, is simply calculating the vertices in $L(q)$ for each partition q . For a 2D mesh, the $L(q)$ sets (for certain partitioning) look like pyramids and if the underlying graph is more complicated one could describe $L(q)$ as some kind a hill or mountain. In subsequent phases of the diamond tiling algorithm, diamonds are fitted between these triangles. By dividing the diamonds (again) into upper and lower halves, one could argue that the lower halves fill the space between the triangles of the previous phase, while the upper halves make a new relief of hills, similar to the one obtained in the previous phase, only moved to a different position. Translating this into case of general matrices, valleys between the hills need to be filled, and then new hills are to be build on this flat surface. The basis of these new hills correspond to the partitions, and now the question is how to determine these partitions. Following the diamond tiling scheme, the repartitioning on line 4 of Algorithm 2 should choose the new partitions trying to satisfy the following two conditions: the boundaries of new partitions are at the tallest points of the

**Figure 3: Hills and valleys**

previous phase and the centre of the partitions is over the centre of valleys. This can get a bit complicated: Figure 3 shows the levels of vertices our algorithm reached after 2 phases on a 60×60 five point 2D mesh with 8 partitions. Each pixel corresponds to a vertex in the 2D mesh. Brighter colours represent higher, darker colours represent lower levels. The details of the implementation including the details of repartitioning are discussed in Section 4.

4 DETAILS

Algorithm 2 shows a general idea about what are the things DMPK needs to do. Our assumption is that DMPK will be used in each iteration of a larger application, and will efficiently calculate the vectors v, A^1v, \dots, A^Mv , for some $M \gg m$, in chunks of m by executing DMPK M/m times. The generation of a new partition (line 4 in Algorithm 2) and determining how to perform the communication (line 5 in Algorithm 2) are computationally very expensive, and would not lead to an efficient implementation. For this reason, DMPK in our implementation performed in several stages. This approach enables us to determine the communication patterns in the first stages, which can be considered as preprocessing and then execute these communication patterns with the actual data, by only running the final execution stage M/m times in the application. Separating the execution into a preprocessing and execution stage illuminates the fact that numerical data is needed only in the final, execution stage, not in the communication patter determining stage. This has the hidden benefit, that if there are multiple matrices, with different (nonzero) values with in the same positions (with the same structure), we can use the previously calculated communication patterns. Also, we managed to implement the the preprocessing in such a way, that it is basically identical to a regular SpMV program, only the content of the CSR arrays (ptr, col and val) are different.

The repartitioning stage is divided into further substages. The first stage of preprocessing calculates the levels and determines the repartitioning. The second stage of preprocessing determines the communication patterns. The final stage of DMPK is the execution

of the communication using the previously computed communication patterns and numerical data i.e. the input vector v and (nonzero) values of the matrix A .

4.1 Repartitioning the graph

The input for the first preprocessing stage, which calculates the levels and the partitions for each phase consists of the n , which is the size of the vector (and the dimensions of the A matrix), m which is the target level the algorithm should reach, and the number of phases denoted by s . The output consists of a s different partitionings Π_r (or just a single partitioning Π_0 if $s = 0$), $\ell^{(r)}$ for $0 \leq r \leq s - 1$, and the halo data. The halo data has no partition because of the restriction that the first and the last partitioning must be the same. Instead of one ℓ sequence of length n , the halo is represented as $|\Pi_0|$ number of sequences. For each $q \in \Pi_0$ the sequence $\sigma^{(s,q)}$ (of length n) represents the halo of q , by storing the values $\sigma_i^{(s,q)} = m - \ell_i^{(s-1)}$, that is the number of levels each vertex needs to advance to reach the final level m (within q).

Metis [10] is used for partitioning the graph G . Metis is an open source, fast, high quality partitioner library for irregular graphs. For the first partitioning, the only objective is to find approximately equal size partitions, while in subsequent phases (for $r > 0$), the ability to find partitions with minimal edgecuts is also used. As stated in the previous section, the objectives for phases when $r > 0$, is to have the edges of the new partitions at vertices with higher levels (top of hills) and the centre of the partitions at vertices with lower levels in the previous phase. Since the minimal edge cut functionality of the Metis library is to be used, the level information from vertex labels needs to be converted into edge labels. For this purpose (2) is used:

$$w_{i,j} = \frac{1}{\ell_i + \ell_j - 2 \min(\ell) + 1} \quad (2)$$

The term $2 \min(\ell)$ removes the levels obtained from previous phases, i.e. emphasises only the relative differences between levels. If both levels of vertex i and j , ℓ_i and ℓ_j are large, then the weight of the edge between them, $w_{i,j}$ is going to be small. This weight increases the possibility of the edge to be on the boundary of partitions, which corresponds to the objective of having partition borders where the values from ℓ_i are large. In the opposite case, when both ℓ_i and ℓ_j are small, $w_{i,j}$ will be large, and this will reduce the chance of v_i and v_j ending up in different partitions, which is objective of having the interior of partitions where the levels of the previous phase are low.

The following technical details are worth mentioning. Metis uses undirected graphs, that is the reason for the requirement, mentioned at the beginning of Section 2, that the pattern of the matrix A should be symmetric, since this corresponds to a undirected graph. Also, to use the minimal edgecut functionality of Metis, the edge weights $w_{i,j}$ need to be integers, so $w_{i,j}$ multiplied by a large integer is stored in the implementation.

This preprocessing stage is separate from the next because this stage is executed sequentially, while the next preprocessing stage, and the final execution stage are executed for each partition in parallel.

4.2 Determining communication patterns

The input for the second preprocessing stage, which determines the communication patterns for the execution stage consists of the same input as the first stage (n , m , s and the pattern of A) in addition to the output of the previous phase (Π_r , $\ell^{(r)}$ and $\sigma^{(s,q)}$ for $0 \leq r \leq s - 1$). The output consists of buffers (or arrays) which can be categorised into two major categories: some describe computation and some communication. There are two kinds of communication buffers: send and receive buffers. As mentioned earlier the algorithm described here is executed for each partition $q \in \Pi^{(r)}$ for each phase r .

The first step towards solving the issue of orchestrating the communication is to have all $v_i^{(k)}$ elements in a single array, with a single index. This can be done trivially, by treating the n long vectors at $m + 1$ levels (including level 0) as a single $(m + 1) \times n$ matrix, and store the elements in a single array \hat{v} in the row-major order. This way every element $v_i^{(k)}$ can be identified with a single index $\xi = nk + i$, i.e. $\hat{v}_\xi = v_i^{(k)}$.

A single phase of DMPK can be treated as a single program with input, intermediate results and output. The input is the data received from the previous phase, the intermediate results, i.e. the work it has to perform is $L(q, \ell^{(r)})$ for $q \in \Pi_r$, and the output is the data needed by other processes/partitions in the following phase.

4.2.1 Buffers describing computation. Assuming communication is complete, and the received data (the “input” of the phase) is available, the task of process q is to calculate the vertices in $L(q, \ell^{(r)})$. A vertex $v_i^{(k)}$ is calculated from the matrix values $a_{i,*}$ and from $v_j^{(k-1)}$ such that vertex j is adjacent to vertex i in G . This is the same operation as SpMV, with this slight difference, that some of the calculated values $v_i^{(k)} = \hat{v}_\xi$ can become the input of another $v_{i'}^{(k+1)} = \hat{v}_{\xi'}$ element of $L(q, \ell^{(r)})$. Therefore, the final “execution” stage of the implementation runs Algorithm 3, which is essentially a regular (sequential) SpMV, with the parameters modified as explained later (for now they are just prefixed with an extra ‘m’ character). For simplicity, we will only consider this case, when the both the input and the output are in the same buffer. The reason why this assumption is possible will be discussed in the next section.

Algorithm 3 SpMV($v, mptr, mcol, mval$)

```

1: for  $i = 0$  to  $\dots$  do
2:    $s \leftarrow 0$ 
3:   for  $t = mptr[i]$  to  $mptr[i + 1]$  do
4:      $s \leftarrow mval[t] \times v[mcol[t]]$ 
5:    $v_i \leftarrow s$ 

```

The contents of these modified parameters is basically obtained by simulating the required computations and recording the proper values. For each partition q and phase r the program iterates (sequentially) through all $\xi = nk + i$ values, and if $(i, k) \in L(q, \ell^{(r)})$, that is if $\ell_i^{(r-1)} < k \leq \ell_i^{(r)}$, then records ξ in an index-buffer $\mu = \mu(q, r)$. In other words, if $v_i^{(k)}$ is the vertex which should be calculated in step v of phase r in partition q , the $\mu_v(q, r) = \xi = nk + i$.

The index-buffer μ is not used directly, but enables the construction of the parameters $mptr$, $mcol$, $mval$ used by Algorithm 3. The original (ptr , col , val) and modified ($mptr$, $mcol$, $mval$) parameters have similar purpose.

The original ptr is an $n+1$ long array which satisfies the property that $ptr[i+1] - ptr[i]$ is the number of vertices adjacent to v_i in G . The contents of $mptr$ is modified accordingly. The length of $mptr$ is $|\mu(q, r)| + 1$ and if $\mu_v(q, r) = \xi = nk + i$ then $mptr[\xi + 1] - mptr[\xi]$ is equal to $ptr[i + 1] - ptr[i]$.

The length of col and val equals to the last entry in the ptr array. The modified $mcol$ and $mval$ have length equal to the last entry of $mptr$. For $T \in [ptr[i], ptr[i + 1])$, $col[T] = j$ is the index of the t -th neighbour of v_i where $t = T - ptr[i]$ and $val[T]$ is the element $a_{i,j}$, from the i -th row and j -th column of A . Similarly, for $\tau \in [mptr[\xi], mptr[\xi + 1])$ where $\xi = nk + i$, using previous notation, if $mcol[\tau] = v'$, then $\mu_{v'}(q, r) = \xi' = n(k - 1) + j$ where j is index of the t' -th neighbour of i and $mval[\tau] = a_{i,j}$ for $t' = \tau - mptr[\xi]$. The values of $mcol$ are filled, using a lookup table, which for each possible index ξ stores the corresponding v index such that $\mu_v = \xi$.

Unlike the original val array, $mval$ is most likely to have duplicates, not just in the sense of two different entries having the same value (that is possible within the case of the original val when two different elements of A are the equal), but in the sense that the same $a_{i,j}$ element is stored more than once in $mval$. This can be remedied at the price of introducing another indirection: an array $midx$ which stores indices of the original val array, instead of storing the values themselves. The modification to Algorithm 3 is trivial: $val[midx[t]]$ should be used instead of $mval[t]$ in line 4.

It should be noted, that at this point are the values of the matrix used for the first time. This can be postponed by a modification similar to the one described above which uses the $midx$ array. This has the advantage, that if there are multiple matrices $A^{(t)}$, which have different values but the same pattern/structure, than by reading the doing the preprocessing stages for only one matrix, with the $midx$, the val arrays off the other matrices can be easily swapped out.

4.2.2 Buffers describing communication. The communication is implemented as an `MPI_Alltoallv` call. This approach is not very efficient, but since this implementation is only a proof of concept, it aims to ensure that all the information that is required to describe the communication is gathered and available. The most important parameters¹ of a the `MPI_Alltoallv` routine are:

send (receive) count arrays with length equal to the number of processes ($P = |\Pi|$), which at the q -th position stores the number of elements sent to (received from) the q -th process/partition.

send (receive) buffers a pointer to an allocated memory region to (from) where the current sends sends (receives) the the data.

send (receive) displacements array of length $P = |\Pi|$, which at the i -th position specifies the offset from the send (receive) buffer, from (to) where the i -th process can send (receive) data.

Smiliar buffers are computed in DMPK as well. This is the information which a full description of what data (including the amount) should be sent from which to which process.

The send and receive information is loosely speaking the same. The global communication (which includes both at the same time the send and the receive information) could be summarised in a $P \times P$ communication table, where the number in the q -th row and p -th column would tell how many elements are sent from process q to p . The “send” related information is stored in the rows and the “receive” related information is stored in the columns of this communication table.

In the discussion of the computation buffers above, the assumption was made, that only the buffers which contained only the intermediate results of a phase are used both as input and output. However this implies no input data for the intermediate results which would be meaningless. This paradox can easily be resolved by having both the receive buffers and the computation buffers in one big buffer. For convenience, in the implementation, $\mu = \mu(q)$ for each partition q is a single array, which starts which the parts of v which belongs to q would be the read buffer for phase $r = 0$, continues with the computation buffer and further continues to alternate between the read and computation buffers for each phase.

Alternating arrangement of read and computation buffers is possible because it is irrelevant for the receiver what data comes from which partition. However this is not the case for the send buffers, because the sender needs to know what to send where, and this is the main difference between the send and receive buffers. The same data might need to be sent to multiple partition and this implies the need to have multiple copies of the same data in the send buffer. For this reason, in the implementation the send buffers is stored in a different part of memory that the receive and computation buffers.

The communication consists of two steps: copy the required data from the computation buffers to the **send buffer**, and call `MPI_Alltoallv`. To be able to decide which elements need to be sent, besides the, above mentioned `MPI_Alltoallv` parameters which describe the amount and the destination of data, one more *fill buffer* is needed, to describe fill the payload, i.e. the `MPI_Alltoallv` parameter **send buffer** from the computation buffers. The same lookup table is used to create this fill buffer, which was used to create $mcol$. If the program determines that the vertex needed by an entry in the computation buffer if in another partition, an index v is added to the fill buffer, such that $\mu_v = \xi$ if \hat{v}_ξ was the needed vertex. This way, for each partition and each phase, the fill buffer contains the indices of the elements in computation buffer, which need to be copied to the **send buffer** when they have been calculated. This is analogous to the $mcol$ buffer storing the information on how to utilise the information in the read and computation buffer.

4.2.3 Optimising communication. There is an import possibility for optimisation at this point. Metis, partitions G by assigning an integer from 0 to $P - 1$ to each vertex v_i . The structure of the partitioning is optimal, however there is a possibility to reduce communication, by choosing different labels for the partitions. The optimisation simply consists of going through all the permutations $\pi : \{0, \dots, P-1\} \rightarrow \{0, \dots, P-1\}$, and selecting the one, which results it the communication table (which includes the “self-communication”,

¹For the detailed description consult [8].

i.e. the number of vertices from the previous phase used by the same process) with maximal trace. The trace being maximal, is equivalent to most of the data having the same source and destination partition, i.e. most of the data is kept on the partition.

4.3 Execution: performing the calculations

The input of the execution phase is the input vector v and the output of previous stage: the communication and computation buffers. The output of the execution stage should be the final output of DMPK: the vectors $v, Av, \dots, A^m v$. As mentioned earlier the algorithm described here is executed for each partition $q \in \Pi^{(r)}$ for each phase r . The assumption is that DMPK will be executed multiple times with the same matrix, and the amortised execution time depends mostly on the speed of this final execution stage. Therefore the execution stage is made to be as simple as possible, since this code is the main target for optimisation. However, since this is just exploratory research, to understand if this approach is worth pursuing, most of the possible optimisations are not implemented. Algorithm 4 shows approximately the actions performed by this stage.

Algorithm 4 EXECUTION(v , comm. and comp. buffers)

SPMV($v, mptr, mcol, mval$)

for $r = 1$ **to** s **do**

 Update buffer pointers based on the r

 COMMUNICATE(communication buffers)

 SPMV($v, mptr, mcol, mval$)

The intended use of DMPK consists of the following. First the preprocessing stages are run which yields the communication and communication buffers. These buffers encode the instructions the execution stage should perform. After these buffers are obtained, they are fed into the execution stage with the actual values of the input vector, and then Algorithm 4 is executed multiple times. In the beginning, based on the initial partition Π_0 , the v_i values are distributed to the different processes. After the execution stage finishes, the same parts of the results are obtained in the same partitions, because of the restriction for the last partitioning to be the same as the initial partition. This is convenient, because it enables the multiple executions with the input always being the output of the previous execution.

5 EXPERIMENTS AND RESULTS

We now describe the results of our measurements on how DMPK affects the amount of communication for various matrices. Our main concern is the amount of communication and the amount of redundant computation (naturally, lower is better for both). We performed these measurements of various matrices, some artificially generated such as 5 or 9 point 2D meshes, and some from the SuiteSparse Matrix Collection [4].

5.1 Experiments

The matrices measured for this paper are listed in Table 1. Each matrix is described by its spyplot, name, description and the triple showing number of rows, nonzeros and nonzeros/rows.









			
m5p10 100 (100, 460, 5)	m5p100 10,000 (10K, 49.6K, 5)	m9p10 100 (100, 784, 9)	m9p100 10,000 (10K, 88K, 9)
			
bmw7st-1 Stiffness matrix (141K, 7.3M, 51)	cfd2 Pressure matrix (123K, 3.1M, 25)	gearbox Aircraft flap actuator (153K, 9.1M, 59)	xenon2 Complex zeolite sodalite crystals (157K, 3.9M, 25)

Table 1: Measured matrices

For each matrix we measured the communication and the redundant computation by setting different values for the parameters described in Table 2.

Param.	description	values
P	the number of partitions/processes	4, 8
m	the target level	10, 20
s	the (maximal) number of phases	0, 1, \dots , 4

Table 2: Experiment parameters

5.2 Results

The results are presented in Figures 4 to 11. Each Figure contains eight plots showing the number of transfers and redundantly computed nodes as a function of s , the number of phases. For each (P, m) pair where $P \in \{4, 8\}$ and $m \in \{10, 20\}$ the number of redundant computations is represented by a solid line and the number of transfers by a dashed line of the same colour. The correspondence between the colours and (P, m) pairs is given in each figure as a legend.

The main observation, is that the amount of communication grows, while the redundant computation shrinks as the number of phases is increased. This is inline with our expectations.

As for the rest of the parameters for the greater value of $m = 20$ levels, the overall amount of work is greater, resulting in more communication and more redundant computation. For the other parameter of P , i.e. the number of partitions/processes, more partitions, result in more communication and more redundant computations.

In general, the number of phases has a greater (positive) effect on the amount of redundant computation than on communication. That is, for different values of s , the amount of communication only changes by a factor of approximately $\times 1 - 1.5$ (in the extreme case of the gearbox matrix a factor of $\times 3$), while in all experiments, the amount of redundant computation is reduced by a factor of at least $\times 3$, but usually it is much higher. For some of the matrices, such as the m5p100, in some cases, the amount of redundant computation is reduced to 0. This happens when the matrix is large and the

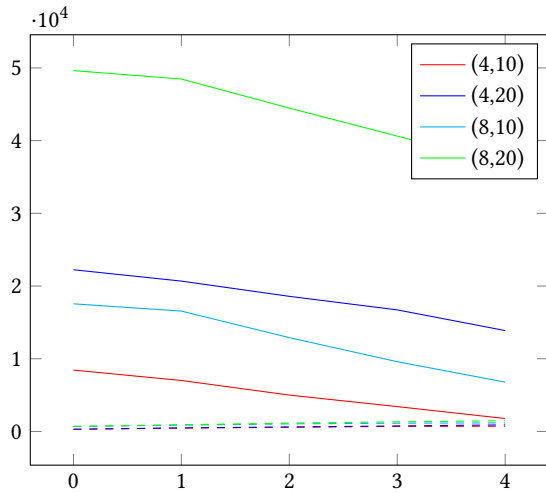


Figure 4: Comm. and redundant comp. for m5p10

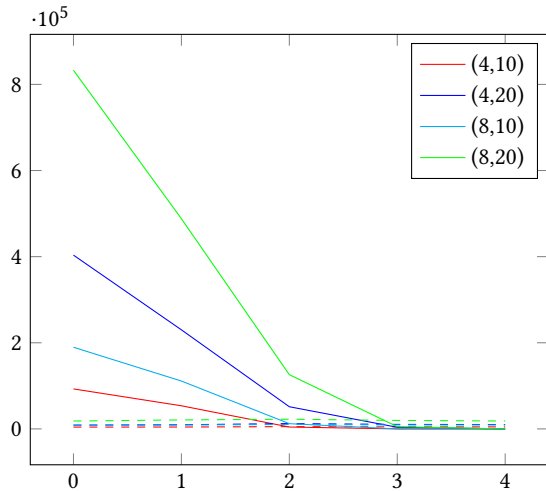


Figure 5: Comm. and redundant comp. for m5p100

number of levels m is small. As a result, the phases climb too fast, and reach the target level before the last phase, leaving no work, hence no communication for the last phases.

The exact numbers are provided in a table for each matrix. The columns for each matrix are the number of processes/partitions P , the target level m , the number of phases s , the number of elements transmitted during the execution in the *comm* column, number of redundant multiplications performed in the *red.calc* column, the minimum number of multiplications needed by the sequential algorithm. As indicated earlier the case when $s = 0$ corresponds to the PA1, and $s = 1$ to the PA2 algorithm.

6 CONCLUSIONS

Our conclusions is that we demonstrated the ability to control the balance between communication and redundant computation,

P	m	s	comm	red.calc	mincalc
4	10	0	300	8,440	4,600
4	10	1	464	7,012	4,600
4	10	2	573	5,003	4,600
4	10	3	694	3,418	4,600
4	10	4	723	1,779	4,600
4	20	0	300	22,240	9,200
4	20	1	492	20,680	9,200
4	20	2	621	18,583	9,200
4	20	3	759	16,720	9,200
4	20	4	904	13,876	9,200
8	10	0	679	17,555	4,600
8	10	1	869	16,554	4,600
8	10	2	1,042	12,903	4,600
8	10	3	1,137	9,595	4,600
8	10	4	1,180	6,785	4,600
8	20	0	700	49,627	9,200
8	20	1	924	48,472	9,200
8	20	2	1,151	44,475	9,200
8	20	3	1,321	40,625	9,200
8	20	4	1,474	36,845	9,200

Table 3: Matrix m5p10, $n = 100$

P	m	s	comm	red.calc	mincalc
4	10	0	4,204	92,885	496,000
4	10	1	4,353	53,705	496,000
4	10	2	5,283	4,390	496,000
4	10	3	4,703	0	496,000
4	10	4	4,703	0	496,000
4	20	0	8,794	403,550	992,000
4	20	1	9,644	230,115	992,000
4	20	2	12,412	51,485	992,000
4	20	3	10,490	3,805	992,000
4	20	4	10,094	0	992,000
8	10	0	8,634	189,600	496,000
8	10	1	9,083	111,185	496,000
8	10	2	10,365	12,170	496,000
8	10	3	9,233	0	496,000
8	10	4	9,230	0	496,000
8	20	0	18,301	832,873	992,000
8	20	1	20,844	488,075	992,000
8	20	2	22,483	125,960	992,000
8	20	3	19,704	5,225	992,000
8	20	4	18,365	25	992,000

Table 4: Matrix m5p100, $n = 10,000$

P	m	s	comm	red.calc	mincalc
4	10	0	300	16,200	7,840
4	10	1	492	13,632	7,840
4	10	2	662	9,960	7,840
4	10	3	748	8,493	7,840
4	10	4	738	4,944	7,840
4	20	0	300	39,720	15,680
4	20	1	492	37,152	15,680
4	20	2	662	33,480	15,680
4	20	3	748	32,013	15,680
4	20	4	822	27,945	15,680
8	10	0	700	36,491	7,840
8	10	1	882	35,140	7,840
8	10	2	1,098	29,197	7,840
8	10	3	1,144	24,333	7,840
8	10	4	1,280	18,548	7,840
8	20	0	700	91,371	15,680
8	20	1	882	90,020	15,680
8	20	2	1,098	84,077	15,680
8	20	3	1,146	79,205	15,680
8	20	4	1,319	73,234	15,680

Table 5: Matrix m9p10, $n = 100$

P	m	s	comm	red.calc	mincalc
4	10	0	5,231	205,740	888,040
4	10	1	5,489	119,784	888,040
4	10	2	6,245	16,227	888,040
4	10	3	5,582	0	888,040
4	10	4	5,582	0	888,040
4	20	0	11,152	907,854	1,776,080
4	20	1	12,306	516,519	1,776,080
4	20	2	13,736	163,530	1,776,080
4	20	3	11,369	0	1,776,080
4	20	4	11,553	0	1,776,080
8	10	0	10,206	393,639	888,040
8	10	1	10,837	232,092	888,040
8	10	2	12,672	37,080	888,040
8	10	3	10,977	225	888,040
8	10	4	10,861	0	888,040
8	20	0	22,747	1,796,226	1,776,080
8	20	1	27,346	1,063,077	1,776,080
8	20	2	28,116	426,261	1,776,080
8	20	3	28,096	93,120	1,776,080
8	20	4	25,673	7,515	1,776,080

Table 6: Matrix m9p100, $n = 10,000$

P	m	s	comm	red.calc	mincalc
4	10	0	103,734	21,279,430	73,396,670
4	10	1	92,848	9,370,179	73,396,670
4	10	2	198,216	3,393,539	73,396,670
4	10	3	234,067	821,530	73,396,670
4	10	4	246,197	397,594	73,396,670
4	20	0	206,391	100,903,516	146,793,340
4	20	1	245,413	50,270,950	146,793,340
4	20	2	385,189	30,505,549	146,793,340
4	20	3	541,531	15,052,740	146,793,340
4	20	4	613,673	9,579,194	146,793,340
8	10	0	196,097	38,216,211	73,396,670
8	10	1	189,634	18,517,632	73,396,670
8	10	2	308,348	8,769,146	73,396,670
8	10	3	346,686	3,265,894	73,396,670
8	10	4	343,883	916,410	73,396,670
8	20	0	424,522	196,086,374	146,793,340
8	20	1	581,491	112,243,853	146,793,340
8	20	2	651,959	81,637,790	146,793,340
8	20	3	776,980	53,805,529	146,793,340
8	20	4	821,521	29,624,723	146,793,340

Table 7: Matrix bmw7st-1, $n = 141,347$

P	m	s	comm	red.calc	mincalc
4	10	0	74,764	7,899,661	30,878,980
4	10	1	72,752	4,261,138	30,878,980
4	10	2	115,006	1,394,205	30,878,980
4	10	3	112,926	63,738	30,878,980
4	10	4	113,732	0	30,878,980
4	20	0	162,102	36,679,638	61,757,960
4	20	1	179,461	19,871,171	61,757,960
4	20	2	241,504	11,842,488	61,757,960
4	20	3	242,271	3,784,788	61,757,960
4	20	4	283,448	1,109,227	61,757,960
8	10	0	160,910	16,385,185	30,878,980
8	10	1	165,716	9,315,419	30,878,980
8	10	2	236,076	3,944,445	30,878,980
8	10	3	230,694	755,601	30,878,980
8	10	4	231,601	75,759	30,878,980
8	20	0	356,158	80,296,593	61,757,960
8	20	1	461,567	48,875,192	61,757,960
8	20	2	519,581	32,421,385	61,757,960
8	20	3	532,937	16,330,783	61,757,960
8	20	4	612,157	9,061,078	61,757,960

Table 8: Matrix cfd2, $n = 123,440$

P	m	s	comm	red.calc	mincalc
4	10	0	99,692	24,294,608	90,804,040
4	10	1	95,352	9,871,755	90,804,040
4	10	2	198,999	5,181,359	90,804,040
4	10	3	264,897	2,614,449	90,804,040
4	10	4	295,686	883,211	90,804,040
4	20	0	183,428	110,204,862	181,608,080
4	20	1	264,229	67,021,869	181,608,080
4	20	2	368,755	53,890,878	181,608,080
4	20	3	477,365	41,613,472	181,608,080
4	20	4	587,471	28,533,642	181,608,080
8	10	0	217,601	49,363,088	90,804,040
8	10	1	228,729	24,352,678	90,804,040
8	10	2	324,833	14,308,198	90,804,040
8	10	3	396,593	7,301,514	90,804,040
8	10	4	441,470	3,800,870	90,804,040
8	20	0	456,246	254,922,118	181,608,080
8	20	1	668,806	164,982,291	181,608,080
8	20	2	766,438	133,312,370	181,608,080
8	20	3	813,534	102,416,480	181,608,080
8	20	4	859,458	78,584,864	181,608,080

Table 9: Matrix gearbox, $n = 153,746$

P	m	s	comm	red.calc	mincalc
4	10	0	124,366	13,408,903	38,666,880
4	10	1	135,012	7,774,501	38,666,880
4	10	2	242,836	3,303,020	38,666,880
4	10	3	247,330	387,953	38,666,880
4	10	4	241,865	4,262	38,666,880
4	20	0	267,843	59,836,338	77,333,760
4	20	1	317,809	35,868,447	77,333,760
4	20	2	465,274	26,053,918	77,333,760
4	20	3	531,210	12,186,619	77,333,760
4	20	4	568,840	2,911,517	77,333,760
8	10	0	242,719	25,532,527	38,666,880
8	10	1	269,925	15,186,539	38,666,880
8	10	2	401,719	6,452,540	38,666,880
8	10	3	424,061	1,307,478	38,666,880
8	10	4	428,071	251,333	38,666,880
8	20	0	530,344	117,765,880	77,333,760
8	20	1	777,237	80,374,888	77,333,760
8	20	2	851,761	55,166,197	77,333,760
8	20	3	897,783	32,323,057	77,333,760
8	20	4	935,267	19,230,184	77,333,760

Table 10: Matrix xenon2, $n = 157,464$

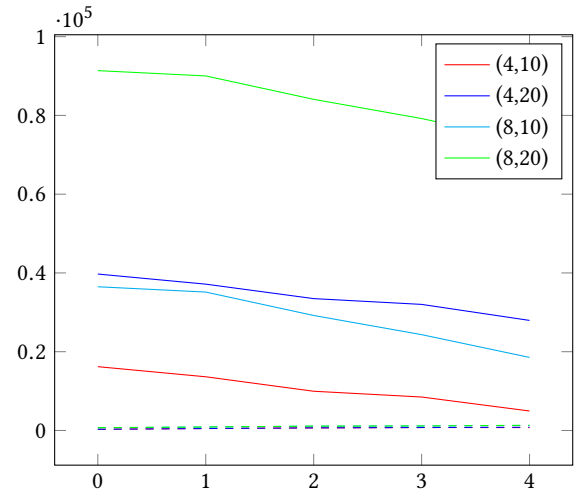


Figure 6: Comm. and redundant comp. for m9p10

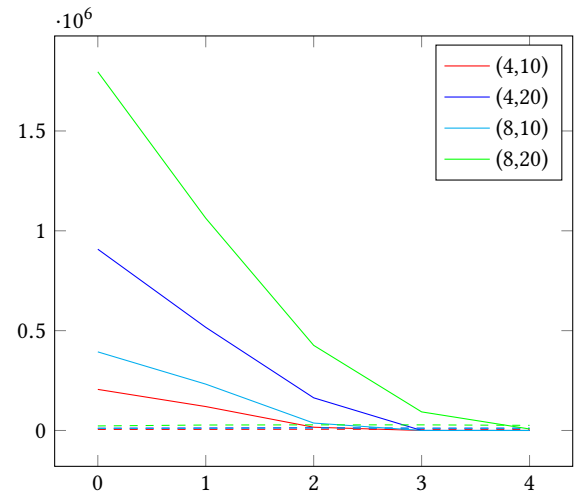


Figure 7: Comm. and redundant comp. for m9p100

furthermore reduction of redundant computation is greater than the increase of communication.

To achieve this, we had to find a good approach to repartition the graph in the intermediate phases. This was done, by transforming the information contained in the levels of vertices into edge weights, and so transform the problem of minimising communication to a problem of finding minimal edgecut.

All the code developed for this research is available at <https://github.com/vatai/mpk/>.

7 FUTURE WORK AND LIMITATION

Since this research is in its preliminary stages, it has limitations. We don't present any performance measurements (in terms of execution speed), since the code is not optimised at all.

There is a lot to be done to make use of these ideas in real applications.

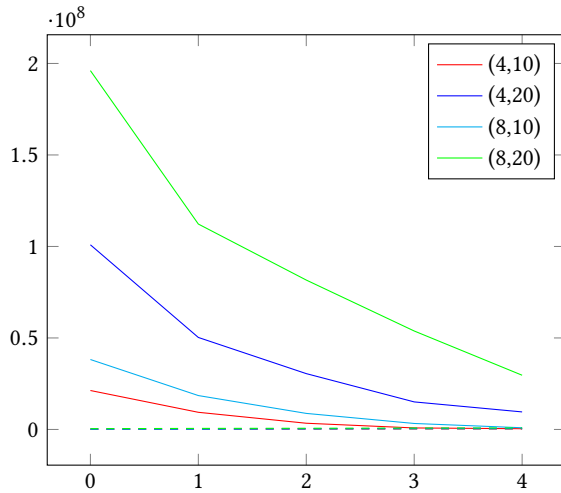


Figure 8: Comm. and redundant comp. for bmw7st-1

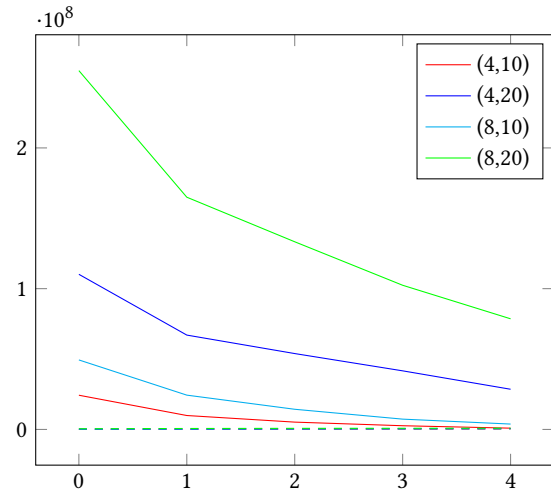


Figure 10: Comm. and redundant comp. for gearbox

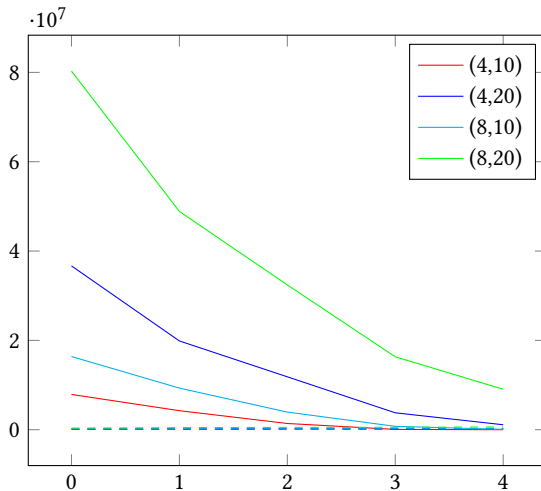


Figure 9: Comm. and redundant comp. for cfd2

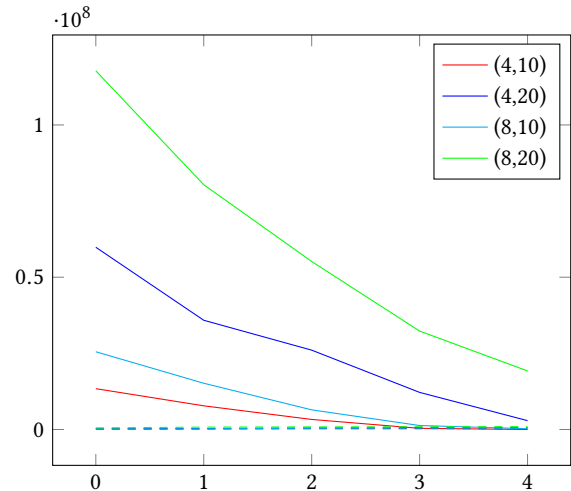


Figure 11: Comm. and redundant comp. for xenon2

The most obvious bottleneck in the current implementation is the lack of hiding the communication. The MPI routine used is synchronous call, which does not allow any computation to be done while communication is in progress. This is an obvious discrepancy and also the least trivial problem. The basic idea to solve this, is to somehow cut the execution buffer in half (or smaller chunks) and start the communication as soon as these chunks are computed, without waiting for the commutation of the whole phase to finish.

The other problem would be load balancing. Simply looking at size of the files which store the communication and computation buffers for each process, it is obvious that the amount of work performed by the different process is significant which degrades the performance.

And finally, the code to do the computation in each phase after the communication was completed is sequential. If a computation node has multiple cores, there is no reason not to use all of them by

implementing a hybrid MPI/OpenMP parallel approach. However, this situation is mitigated somewhat by the fact that the execution stage, which would need to run on multiple threads is very similar to the traditional SpMV algorithm, and there is ample amount of literature which deals with this issue such as [15] for example.

REFERENCES

- [1] Zhaojun Bai, Jack Dongarra, Ding Lu, and Ichitaro Yamazaki. 2019. Matrix Powers Kernels for Thick-Restart Lanczos with Explicit External Deflation. In *International Parallel and Distributed Processing Symposium (IPDPS)*.
- [2] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* (2014), 1–155.
- [3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *Proc. SC12*.
- [4] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. <https://doi.org/10.1145/2049662.2049663>
- [5] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2007. Avoiding communication in computing Krylov subspaces. *ECS Dept., UC*

- Berkeley, Tech. Rep. UCB/Eecs-2007-123 (2007).
- [6] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. 2008. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–12.
- [7] Stefan Feuerriegel and H. Martin B ker. 2015. The Non-Symmetric s -step Lanczos Algorithm: Derivation of Efficient Recurrences and Synchronization-Reducing Variants of BiCG and QMR. *Int. J. Appl. Math. Comput. Sci.* 4 (2015), 769–785.
- [8] Message Passing Interface Forum. 2015. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart. <https://books.google.co.jp/books?id=Fbv7jwEACAAJ>
- [9] Mark Hoemmen. 2010. *Communication-avoiding Krylov subspace methods*. Ph.D. Dissertation. University of California, Berkeley.
- [10] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.
- [11] Marghoob Mohiyuddin. 2012. *Tuning Hardware and Software for Multiprocessors*. Ph.D. Dissertation. UC Berkeley.
- [12] Y. Saad. 1996. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company. <https://books.google.co.jp/books?id=jLtiQgAACAAJ>
- [13] Reiji Suda. 2016. Domain Decomposition Algorithm for Generalized Diamond Matrix Powers Kernel. In *IPSJ SIG Technical Report*.
- [14] David G. Wonnacott and Michelle Mills Strout. 2013. On the Scalability of Loop Tiling Techniques. In *Proc. IMPACT 2013*.
- [15] Katherine Yelick, James Demmel, Leonid Oliker, Samuel Williams, Richard Vuduc, and John Shalf. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194. <https://doi.org/10.1016/j.parco.2008.12.006>