LATENCY-CENTRIC MODELS AND METHODS FOR SCALING SPARSE OPERATIONS

A DISSERTATION SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

By

Oğuz Selvitopi

July 2016

Latency-centric models and methods for scaling sparse operations

By Oğuz Selvitopi

July 2016

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Cevdet Aykanat (Advisor)

Özcan Öztürk

Murat Manguoğlu

Mustafa Özdal

Tayfun Küçükyılmaz

Approved for the Graduate School of Engineering and Science:

Levent Onural
Director of the Graduate School

ABSTRACT

LATENCY-CENTRIC MODELS AND METHODS FOR SCALING SPARSE OPERATIONS

Oğuz Selvitopi
Ph.D. in Computer Engineering
Advisor: Cevdet Aykanat
July 2016

Parallelization of sparse kernels and operations on large-scale distributed memory systems remains as a major challenge due to ever-increasing scale of modern high performance computing systems and multiple conflicting factors that affect the parallel performance. The low computational density and high memory footprint of sparse operations add to these challenges by implying more stressed communication bottlenecks and make fast and efficient parallelization models and methods imperative for scalable performance. Sparse operations are usually performed with structures related to sparse matrices and matrices are partitioned prior to the execution for distributing computations among processors. Although the literature is rich in this aspect, it still lacks the techniques that embrace multiple factors affecting communication performance in a complete and just manner. In this thesis, we investigate models and methods for intelligent partitioning of sparse matrices that strive for achieving a more correct approximation of the communication performance. To improve the communication performance of parallel sparse operations, we mainly focus on reducing the latency bottlenecks, which stand as a major component in the overall communication cost. Besides these, our approaches consider already adopted communication cost metrics in the literature as well and aim to address as many cost metrics as possible. We propose one-phase and two-phase partitioning models to reduce the latency cost in one-dimensional (1D) and two-dimensional (2D) sparse matrix partitioning, respectively. The model for 1D partitioning relies on the commonly adopted recursive bipartitioning framework and it uses novel structures to capture the relations that incur latency. The models for 2D partitioning aim to improve the performance of solvers for nonsymmetric linear systems by using different partitions for the vectors in the solver and uses that flexibility to exploit the latency cost. Our findings indicate that the latency costs should definitely be considered in order to achieve scalable performance on distributed memory systems.

Keywords: Parallel computing, distributed memory systems, combinatorial scientific computing, sparse matrices, load balancing, communication bottlenecks, graph partitioning, hypergraph partitioning.

ÖZET

TÜRKÇE BAŞLIK

Oğuz Selvitopi
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Cevdet Aykanat
Temmuz 2016

Seyrek çekirdeklerin ve işlemlerin büyük-ölçekli dağıtık bellekli sistemlerde paralelizasyonu modern yüksek performanslı hesaplama sistemlerinin sürekli artan ölçekleri ve paralel performansı etkileyen birbiriyle çakışan birçok etmenin varlığı nedenleriyle büyük bir zorluk olarak kalmaktadır. Seyrek işlemlerin düşük hesaplama yoğunlukları ve yüksek bellek izleri daha fazla vurgulanan iletişim darboğazlarını işaret ederek bu zorluklara yenilerini eklemekte ve ölçeklenebilir performans için hızlı ve verimli paralelizasyon model ve metotlarını zorunlu kılmaktadır. Seyrek işlemler genelde seyrek matrislerle ilgili veri yapıları üzerinde gerçekleştirilmekte ve matrisler koşma öncesinde işlemcilere dağıtılmak için bölümlenmektedir. Literatür bu alanda çok zengin olmasına karşın, literatürde iletişim performansını belirleyen birçok etmeni bu etmenlere hakedilen önemi atfederek tam anlamıyla aynı anda işleyebilecek yöntemlerin eksikliği bulunmaktadır. Bu tezde iletişim performansının daha doğru bir yakınsamasını elde etmek amacıyla seyrek matrislerin akıllı bölümlenmesini sağlayan model ve metotları incelemekteyiz. Paralel seyrek işlemlerin iletişim performansını arttırmak için bütün iletişim maliyetlerinde başlıca bir bileşen olarak kendisini gösteren gecikim darboğazlarının azaltılmasına odaklanmaktayız. Bunun yanı sıra, önerilen yaklaşımlar literatürde halihazırda kabul görmüş iletişim maliyet ölçütlerini de hesaba katarak olabildiğince fazla ölçütü aynı anda işlemeye çalışmaktadır. Birboyutlu (1D) ve iki boyutlu (2D) seyrek matrislerin bölümlenmesinde gecikim maliyetlerini azaltmak için sırasıyla bir-fazlı ve iki-fazlı bölümleme modelleri önermekteyiz. 1D bölümleme için önerilen model sıkça kullanılan özyinelemeli bölümleme metoduna dayanmakta olup gecikim gerektiren ilişkileri gösterebilmek için yeni yapılar kullanmaktadır. 2D bölümleme için önerilen bölümleme modelleri asimetrik lineer sistemler için kullanılan çözücülerin performanslarını, bu çözücülerde kullanılan vektörler üzerinde farklı bölümler aracılığıyla gecikim maliyetlerini azaltarak geliştirmeyi hedeflemektedir. Tezde elde ettiğimiz bulgular dağıtık bellekli sistemlerde ölçeklenebilir performans elde edilebilmesi için gecikim maliyetlerinin kesinlikle düşünülmesi gerektiğini göstermektedir.

Anahtar sözcükler: Paralel hesaplama, dağıtık bellekli sistemler, kombinasyonal bilimsel hesaplama, seyrek matrisler, yük dengeleme, iletişim darboğazları, çizge bölümleme, hiperçizge bölümleme.

Acknowledgement

I have met two great teachers throughout my entire education and I feel grateful and lucky for I had the opportunity to work and be advised by one of them for years. Professor Cevdet Aykanat's relentless dedication to good research, joy and excitement at the moment of discovery, perspective and insight on not only research but also on certain aspects of life showed me how an ideal academic should be. His approach to research has shaped my academic maturity and his cheerful mood has lifted my spirits up even in the most dire situations.

I would like to thank to Ata Turk for his efforts and valuable contributions in our joint works. He surely proved to be more than just a collaborator for me and directed me from the dead ends.

I would like to thank to Seher Acer whose way of seeing through things lead to fruitful discussions and improved the quality of our doings. Her views often raised questions about abandoning the control freak in me and certainly proved me sometimes doing so may lead to better conclusions.

I would like to thank to Kadir Akbudak for showing me collaboration may be easy when you expect it to be the hardest.

I also would like to thank to the professors Hakan Ferhatosmanoglu and Mustafa Ozdal who provided valuable contributions with their brilliant minds in our joint works.

Most importantly, I would like to thank to my mother, my father and my sister for being there whenever I needed, without whom, all my efforts would be in vain.

Finally, I would like to thank to TUBITAK for supporting me throughout my Ph.D. study under the national scholarship program 2211. I also thank TUBITAK 1001 program for supporting me in the project numbered EEEAG-114E545.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallelizing applications or kernels for distributed memory systems require extra care when the scale of modern high performance computing (HPC) systems taken into account. Ever-larger computing systems with hundreds of thousands of cores are built in order to tackle large-scale science and engineering problems in a fast and efficient manner. Parallelization of applications in such systems is a major challenge in the revitalized field of HPC because a plethora of factors need to be considered in assessing the application performance. Conventional models and methods used for parallelization in the recent decades often fall short when they meet the new or extended set of requirements brought by the scale of such systems.

Parallelization of computations on sparse structures is another challenge as low computational density makes them very difficult to scale. For example, the data sets related to sparse matrices include many zero elements which are not explicitly stored and usually compressed structures are utilized to represent these matrices, such as compressed column storage, block compressed sparse row storage, etc. High memory footprint and irregular access patterns in operations related to sparse matrices make certain optimization techniques commonly used in HPC area difficult to adapt, if not impossible. The low computational density moreover implies that there is not much room for overlapping computation and communication to hide the communication overheads. Hence the communication bottlenecks for sparse matrices become more pronounced on distributed systems compared to their dense counterparts. The closely related field of sparse linear algebra was surely listed as one of the seven dwarves that constitute a key computational challenge in the HPC field [1].

Sparse matrices and graphs are used interchangeably as the widely used adjacency list representation of a graph corresponds to a matrix. The operations on sparse matrices can be parallelized by representing the sparse matrix as a (hyper)graph and partitioning it to distribute the matrix among processors. Although the literature is rich in this aspect, it still lacks the techniques that embrace multiple factors affecting communication performance in a complete and just manner. The models and methods presented in this thesis rely on partitioning graphs and hypergraphs for the purpose of parallelizing operations on sparse matrices and they strive for achieving a more correct approximation of the communication performance in parallelizing sparse kernels and operations.

Many applications aim to obtain a good parallelization by following common conventions. Two rules of thumb in large-scale parallelizations are balancing computations and reducing communication bottlenecks. In the distribution of computations on sparse matrices, our approaches in this thesis always follow and respect these two principles to the fullest extent possible. Reducing communication bottlenecks happens to be a more important and complex issue compared to balancing computational loads as communication is more expensive compared to computation and communication shows itself to be inconstant in more cases compared to computation.

The communication bottlenecks on a distributed system are determined by many factors. The most common bottlenecks are typically related to bandwidth and latency costs. The bandwidth cost is proportional to the amount (volume) of data transferred and the latency cost is proportional to the number of messages communicated. In order to capture the communication requirements of parallel applications more accurately, both components should be taken into account in the partitioning models. Among these two components, the latency costs prove to be more vital for parallel performance, however, as they are generally harder to avoid and improve [2]. Although both costs are reduced within time, the gap between them gradually increases in favor of the bandwidth costs with an approximately 20% annual improvement over the latency costs [2, 3]. Furthermore, computation speeds evolve faster than communication speeds, making communication costs more critical for performance. With the latest developments in the scientific computing field, communication costs are likely to be a major factor in ranking fastest HPC systems [4], known as the Top500 list that measures the world's fastest systems.

In the following chapters, we investigate what can be done to reduce the latency cost in partitioning sparse matrices by representing them as graphs/hypergraphs. Our approaches do not only consider reducing the latency costs. Instead, they also take the already accepted communication cost metrics into account in the literature -such as total communication volume- and try to address as many cost metrics as possible. Note that, just as the bandwidth cost, there are multiple factors that determine the overall latency cost. Some of our models consider these multiple factors while some of them focus on only a single one.

In Chapter 2, we aim to assess the importance of latency on some of the modern large-scale systems. For this purpose, we conduct simple controlled experiments with predefined settings on different systems. We also describe the communication cost model used in the thesis. This model allows a general enough abstraction to estimate the communication costs and it is commonly accepted and widely used in the literature. We also give a simple partitioning experiment setting to obtain a number of parts for parallelization with a very common graph model and then we assess the properties of these partitions to evaluate the communication performance using the described communication cost model without an actual parallel code execution. We try to predict the cases where latency cost may play an important role in communication performance with evaluation on different matrices.

Sparse matrix partitioning is a common technique used for improving performance of parallel linear iterative solvers. Compared to solvers used for symmetric linear systems, solvers for nonsymmetric systems offer more potential for addressing different multiple communication metrics due to the flexibility of adopting different partitions on the input and output vectors of sparse matrix-vector multiplication operations. In this regard, there exist works based on one-dimensional (1D) and two-dimensional (2D) fine-grain partitioning models that effectively address both bandwidth and latency costs in nonsymmetric solvers. In Chapter 3, we propose two new models based on 2D checkerboard and jagged partitioning. These models aim at minimizing total message count while maintaining a balance on communication volume loads of processors; hence, they address both bandwidth and latency costs. We evaluate all partitioning models on two nonsymmetric system solvers implemented using the widely adopted PETSc toolkit and conduct extensive experiments using these solvers on a large-scale HPC system successfully scaling them up to 8192 processors. Along with the proposed models, we put practical aspects of eight evaluated models (two 1D- and six 2D-based) under thorough analysis. This chapter is unique in the sense that it analyzes practical performance of intelligent 2D sparse matrix partitioning models on such scale. Among the evaluated models, the models that rely on 2D jagged partitioning obtain the most promising results by striking a balance between minimizing bandwidth and latency costs.

Intelligent partitioning models are commonly used for efficient parallelization of irregular applications on distributed systems. These models usually aim to minimize a single communication cost metric, which is either related to communication volume or message count. There are only a few works that consider both of them and they usually address each in separate phases of a two-phase approach. In Chapter 4, we propose a recursive hypergraph bipartitioning framework that reduces the total volume and total message count in a single phase. In this framework, the standard hypergraph models, nets of which already capture the bandwidth cost, are augmented with *message nets*. The message nets encode the message count so that minimizing conventional cutsize captures the minimization of bandwidth and latency costs together. Our model provides a more accurate representation of the overall communication cost by incorporating both the bandwidth and the latency components into the partitioning objective. The use of the widely-adopted successful recursive bipartitioning framework provides the flexibility of using any existing hypergraph partitioner. The experiments on instances from different domains show that our model on the average achieves up to 52% reduction in total message count and hence results in 29% reduction in parallel running time compared to the model that considers only the total volume.

An idea proposed in [5] offers computation and communication rearrangements in certain iterative solvers to bound the number of messages communicated (hence addressing the latency cost). The downside of this approach is that it substantially increases the communication volume. In Chapter 5, we propose two iterative-improvement-based heuristics to alleviate the increase in the volume through one-to-one task-to-processor mapping. The main motivation of both algorithms is to keep the processors that communicate high volume of data close to each other in terms of communication pattern of collective operations so that the communicated vector elements cause less forwarding. The heuristics differ in their search space definitions. The first heuristic utilizes full space while the second one restricts it by considering only the directly communicating processors in collective communication operations. We show that the restricted space algorithm is feasible, and on the average, its running time remains lower than the partitioning time up to 2048 processors. Note that these heuristics respect the latency bounds provided by the [5].

We finally list the main findings of the thesis in Chapter 6 and give directions about future research related to the works presented in the preceding chapters.

Chapter 2

Importance of latency on large-scale parallel systems

We first review a common communication cost model that we utilize in the rest of the thesis. Then we evaluate the importance of latency on a few modern parallel systems with simple experiments. These evaluations will guide us through understanding the characteristics of the models and methods proposed for reducing latency costs in the following chapters. We also provide realistic partitioning examples with 20 matrices to assess whether and to which extent latency cost should be considered in order to reduce the communication costs.

2.1 Communication cost model

There are several communication cost models to measure the parallel performance of applications on distributed memory systems that rely on message passing paradigm for inter-process communication. Among these cost models are the Bulk-Synchronous Parallel Model (BSP) [6], LogP [7], LogGP [8], LoPC [9] and LoGPC [10]. All these models take various factors into account to provide a better approximation of the overall communication cost. For example, the

LogP communication cost model considers four important factors that determine the performance of a parallel application: computing bandwidth, communication bandwidth, communication delay and efficiency in overlapping communication and computation. Other cost models aim to extend this model by incorporating other less important factors such as contention on the network (LoPC, LoGPG) and the behavior of the hardware in the case of long messages (LogGP). All these models more or less require adjustments and settings in parameters according to the properties of the underlying parallel system. Other simple yet very effective communication cost model uses a linear function of message startup and unit data transfer time in order to estimate the communication costs [11, 12]. In this cost model, it is assumed there are three different factors that determine the overall communication cost:

- *Per-word transfer time, t_w*: If the channel bandwidth of the parallel system is b, then the cost of communicating a single data word between two processors is $1/b$. This cost contains the memory overheads related to data movement as well.
- *Per-hop time, t_h*: While sending a message from a processor to the another, it may be necessary that some other processor may need to forward this message to the destination processor due to the limitations of the network topology. The time spent between any such two hops is given by the per-hop time.
- *Latency, t_s*: Also known as the startup time, latency is given by the time spent by a processor's handling and preparation of a message that is either sent or received. This overhead contains tasks such as adding headers or tails, determination of the error correcting codes related to a single message, running of the routing algorithm and preparation of the interface between sender and receiver processors. Note that this cost is related to a transmitting a *single* message.

Taking these factors into account, the cost of communicating a message of size m between two processors in a setting where there are l hops between these two processors becomes

$$t_c = t_s + lt_h + mt_w. \tag{2.1}$$

In this formulation, the per-hop time lt_h is dominated by the transfer time mt_w if the message size is too big and by the latency time t_s if the message size is too short. In most of the modern large-scale parallel systems the network diameter is quite low and forwarding of messages is very fast due to the wormhole routing technique (regardless of the distance between any two nodes) [13]. For these reasons, we can safely omit the per-hop overhead without losing much accuracy and the cost of communicating a single message between two processors becomes

$$t_c = t_s + mt_w. \tag{2.2}$$

Although this communication cost model overlooks certain details in design of the parallel systems and algorithms, it provides a general enough abstraction for approximating the communication costs of the applications and codes to be run on a parallel system. This cost model is in our focus in rest of this thesis.

We refer to the mt_w component as bandwidth cost and the t_s component as latency cost. Note that the latency cost is proportional to the number of messages whereas the bandwidth cost is proportional to the number of words.

2.2 Assessment of latency

We try to measure the importance of latency on modern large-scale parallel systems. For this purpose, we conduct simple experiments with different settings. These experiments are centered around two processes repeatedly sending/receiving messages to each other, also called the *ping-pong* experiment. The experiments are performed with the message passing paradigm using MPI. The implementations in the parallel systems are based on MPI Chameleon [14] and according to the parallel system the vendors customized the MPI implementation to take advantage of the special hardware features.

We consider two send and receive schemes for the communication between two processors. These alternatives are presented as an application may necessitate any of these two schemes. The first scheme is called the *blocking* scheme and the processes P_0 and P_1 execute the following code snippet (unrelated details omitted for brevity):

```
/* blocking send/receive calls */
int proc;
MPI_Comm_rank(MPI_COMM_WORLD, &proc);
int pair = proc ^ 1;
...
/* timer start */
if (proc == 1)
  MPI_Recv(from pair);
else
  MPI_Send(to pair);
  MPI_Recv(from pair);
if (proc == 1)
  MPI_Send(to pair);
/* timer end */
...
```

The second scheme is called the *overlapping* scheme and the processes P_0 and P_1 execute the following code snippet:

```
/* nonblocking send/receive calls */
int proc;
MPI_Comm_rank(MPI_COMM_WORLD, &proc);
int pair = proc ^ 1;
...
/* timer start */
MPI_Irecv(from pair);
MPI_Isend(to pair);
MPI_Wait(send request);
MPI_Wait(receive request);
/* timer end */
```

```
...
```

These two simple code snippets will help us assess the extent of the importance of latency in a rather idealistic environment. Note that in both schemes the send and receive calls are repeated 10000 times and a warmup phase is included to ensure a precise timing.

Apart from blocking and nonblocking MPI primitives, we also consider two cases where the two processes either packed in a single node or scattered to two different nodes. This is important because some parallel systems take advantage of the process mapping and use techniques such as direct memory access (DMA) when processes are confined to the same node. In such cases, although the message preparation costs are expected to decrease, latency can still be an important factor. In the tables presented in the following paragraphs, the former case is referred to as *single-node* and the latter case is referred to as *multi-node*, respectively. We present the results obtained in three different large-scale parallel systems. Two of these systems are used in the experiments of the models and methods described in Chapters 3, 4 and 5. In all experiments the message sizes vary from 4 bytes to 128 Kb. All times are in microseconds.

Table 2.1 presents the results of the experiments with the aforementioned settings obtained on a BlueGene/Q system. The “time” column stands for communicating a single message between two processors and the column “L.O.” stands for the latency overhead as percentage of total communication overhead. The assumed latencies for this system are as follows: 3.86 microseconds for single-node communication and 3.37 microseconds for multi-node communication. This system is used in the experiments to evaluate the models and methods in Chapters 3, 4 and 5.

Table 2.2 presents the results of the experiments with the aforementioned settings obtained on a Cray XE6 system. The assumed latencies for this system are as follows: 0.36 microseconds for single-node communication and 2.07 microseconds for multi-node communication. This system is used in the experiments to

Table 2.1: Ping-pong experiments on IBM BlueGene/Q system (Juqueen). L.O. for “Latency Overhead”. Single-node latency overhead: 3.86 microseconds, multi-node latency overhead 3.37 microseconds.

	single-node				multi-node			
	blocking		overlapping		blocking		overlapping	
message size	time	L.O.	time	L.O.	time	L.O.	time	L.O.
4 bytes	3.95	98%	4.78	81%	3.47	97%	4.41	76%
8 bytes	3.86	100%	4.79	81%	3.39	99%	4.39	77%
16 bytes	3.87	100%	4.79	81%	3.40	99%	4.39	77%
32 bytes	3.96	97%	4.79	81%	3.37	100%	4.41	76%
64 bytes	3.98	97%	4.86	79%	3.52	96%	4.43	76%
128 bytes	6.72	57%	8.63	45%	5.08	66%	6.64	51%
256 bytes	6.71	58%	8.76	44%	5.33	63%	6.75	50%
512 bytes	6.79	57%	8.82	44%	7.21	47%	7.83	43%
1 Kb	6.82	57%	8.84	44%	7.54	45%	8.06	42%
2 Kb	7.04	55%	9.09	42%	8.33	40%	8.38	40%
4 Kb	7.41	52%	9.21	42%	10.17	33%	9.12	37%
8 Kb	9.08	43%	9.00	43%	13.54	25%	11.49	29%
16 Kb	11.66	33%	10.37	37%	18.37	18%	13.65	25%
32 Kb	13.39	29%	11.17	35%	27.45	12%	18.44	18%
64 Kb	16.57	23%	12.71	30%	45.80	7%	27.83	12%
128 Kb	22.97	17%	15.97	24%	82.81	4%	46.11	7%

evaluate the models and methods in Chapter 5.

Table 2.3 presents the results of the experiments with the aforementioned settings obtained on an IBM System x iDataPlex system. The assumed latencies for this system are as follows: 0.29 microseconds for single-node communication and 1.04 microseconds for multi-node communication.

We also present the communication times and the latency overheads as plots for IBM BlueGene/Q and Cray XE6 systems in Figures 2.1 and 2.2, respectively, to illustrate the effect of latency on overall communication performance. There are two plots regarding a single system, one for the single-node setting and one

Table 2.2: Ping-pong experiments on Cray XE6 system (Hermit). L.O. for "Latency Overhead". Single-node latency overhead: 0.36 microseconds, multi-node latency overhead 2.07 microseconds.

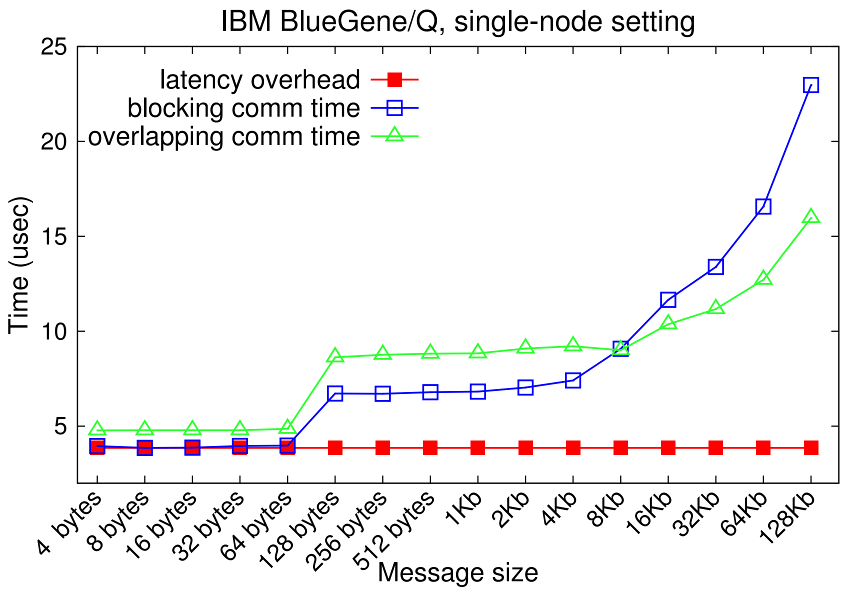
	single-node				multi-node			
	blocking		overlapping		blocking		overlapping	
message size	time	L.O.	time	L.O.	time	L.O.	time	L.O.
4 bytes	0.37	96%	0.37	98%	2.84	73%	2.08	100%
8 bytes	0.38	95%	0.36	100%	2.79	74%	2.07	100%
16 bytes	0.41	89%	0.37	98%	2.73	76%	2.21	93%
32 bytes	0.37	97%	0.36	99%	2.81	74%	2.12	98%
64 bytes	0.38	95%	0.37	98%	2.82	73%	2.37	87%
128 bytes	0.40	89%	0.38	96%	2.87	72%	2.53	82%
256 bytes	0.44	82%	0.40	89%	3.04	68%	2.35	88%
512 bytes	0.53	68%	0.46	78%	3.07	67%	2.17	95%
1 Kb	0.67	54%	0.56	65%	3.55	58%	2.17	95%
2 Kb	0.86	42%	0.84	43%	4.16	50%	2.52	82%
4 Kb	1.33	27%	1.32	27%	5.13	40%	3.43	60%
8 Kb	1.51	24%	1.56	23%	12.29	17%	7.33	28%
16 Kb	2.80	13%	2.38	15%	15.22	14%	9.07	23%
32 Kb	5.28	7%	3.84	9%	20.40	10%	12.14	17%
64 Kb	10.22	4%	6.77	5%	29.88	7%	19.02	11%
128 Kb	20.00	2%	12.47	3%	36.36	6%	24.89	8%

for the multi-node setting. The plots display the overall communication and latency times with respect to varying message sizes. As seen from these figures and tables, latency cost totally dominates the bandwidth cost up to message sizes 64 and 128 bytes on IBM BlueGene/Q and up to 512 bytes and 1 Kb on Cray XE6. On IBM BlueGene/Q from 256 bytes to 4 Kb, the latency cost and the bandwidth cost are comparable, while on Cray XE6 these values range from 2 Kb to 4 Kb. After those message sizes on both systems, the bandwidth cost dominates the latency cost. Judging from these figures, it can be said that for small-to-medium sized messages, latency cost is a factor that should definitely be considered while reducing communication costs. It is expected the latency cost to be more vital for blocking communication primitives as because the communication is not overlapped, it should be more difficult to hide the latency overhead. This seems

Table 2.3: Ping-pong experiments on IBM System x iDataPlex system (SuperMUC). L.O. for “Latency Overhead”. Single-node latency overhead: 0.29 microseconds, multi-node latency overhead 1.04 microseconds.

	single-node				multi-node			
	blocking		overlapping		blocking		overlapping	
message size	time	L.O.	time	L.O.	time	L.O.	time	L.O.
4 bytes	0.42	69%	0.30	99%	2.02	52%	1.07	98%
8 bytes	0.43	68%	0.30	99%	1.97	53%	1.04	100%
16 bytes	0.43	69%	0.29	100%	1.97	53%	1.04	100%
32 bytes	0.44	67%	0.30	97%	2.08	50%	1.11	94%
64 bytes	0.44	67%	0.31	95%	2.14	49%	1.14	92%
128 bytes	0.53	55%	0.34	86%	2.66	39%	1.40	74%
256 bytes	0.58	51%	0.39	76%	2.87	36%	1.52	69%
512 bytes	0.60	49%	0.39	75%	3.10	34%	1.64	64%
1 Kb	0.70	42%	0.46	63%	3.51	30%	1.86	56%
2 Kb	0.99	30%	0.61	48%	4.42	24%	2.32	45%
4 Kb	1.37	21%	0.80	37%	6.31	17%	3.33	31%
8 Kb	2.30	13%	1.28	23%	7.53	14%	3.98	26%
16 Kb	4.02	7%	2.20	13%	11.32	9%	6.58	16%
32 Kb	5.91	5%	3.94	7%	17.45	6%	10.06	10%
64 Kb	9.97	3%	7.43	4%	22.24	5%	14.15	7%
128 Kb	17.92	2%	14.49	2%	36.02	3%	21.20	5%

to be the case for the Cray XE6 and IBM System x iDataPlex systems. This can be seen by comparing the columns blocking L.O. and overlapping L.O. for single-node or by comparing the columns blocking L.O. and overlapping L.O. for multi-node in Tables 2.2 and 2.3, where a higher value indicates that the latency cost is more pronounced. It is not bluntly observed for the IBM BlueGene/Q system, which may be due to the on-board communication chips used in the design of this system. As a final note, the latency overhead on IBM BlueGene/Q seems to be higher compared to other two systems. This can be attributed to the fact that the processors used in this system (PowerPC chips) are slower compared to the ones used in other two (Intel Xeon processors).



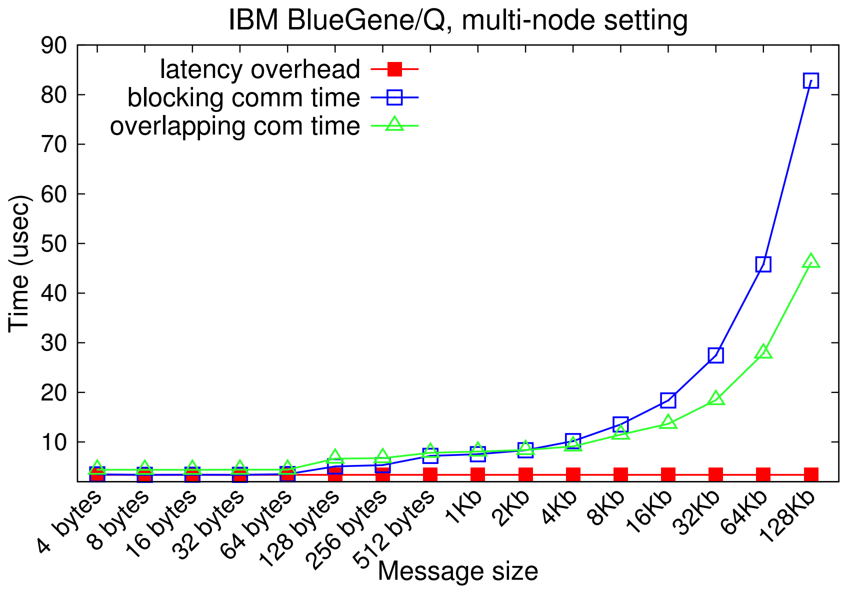
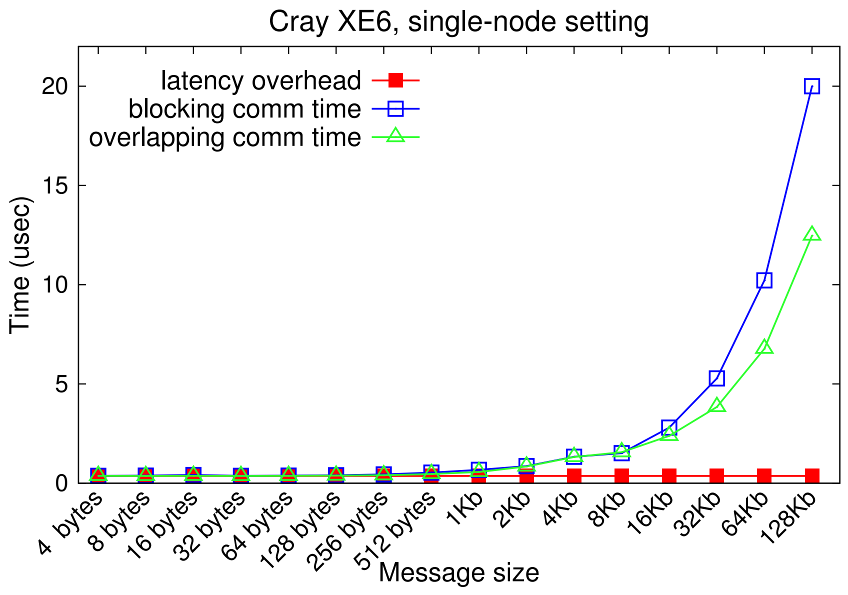


Figure 2.1: Blocking and overlapping communication times vs. latency overhead on IBM BlueGene/Q.



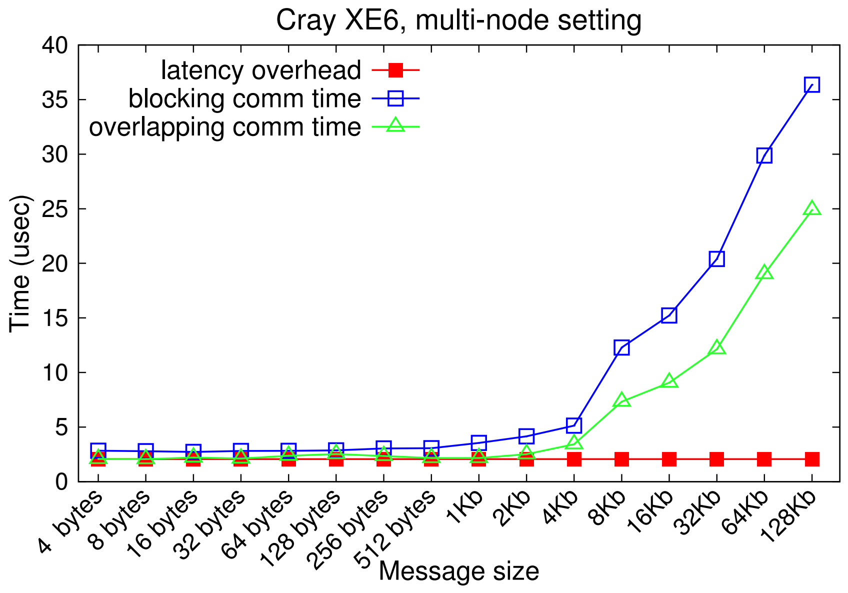


Figure 2.2: Blocking and overlapping communication times vs. latency overhead on Cray XE6.

2.3 Partitioning and latency

In a distributed setting, in any kernel or operation that contain computations related to matrices and vectors, matrices and vectors are usually distributed among processors. One very common technique to obtain a partition of the matrix is to represent it as a graph and use a graph partitioner to partition the graph. A partition of the graph induces a partition of the matrix, which is used to distribute the matrix among processors. Although there are various metrics considered by the partitioners, the most widely used metric is the total communication volume. In this section we partition 20 matrices into 512 parts (i.e., 512 processors) and assess their communication requirements to see if latency plays an important role in communication costs and whether it is worthwhile to exploit models and methods to reduce latency costs. The number of nonzeros in these matrices varies from one million to ten million and they are symmetric matrices. For the sake of simplicity, we confine ourselves to consider obtaining a one-dimensional partitioning of the matrix and consider the ubiquitous sparse matrix-vector multiplication (SpMV). The importance of latency should directly be observable from the communication statistics of the obtained partitions, i.e., there is no need for actually running the computations on the matrices in parallel. Our motivation is that, especially in the case of strong scaling, there will be small to medium sized messages and latency cost should manifest itself in such matrices. We use Metis [15] with default options to partition the matrices.

Table 2.4 presents the communication statistics for 20 matrices partitioned into 512 parts. There are two metrics related to the bandwidth cost, average volume and maximum volume, and two metrics related to the latency cost, average number of messages handled by a processor and maximum number of messages handled by a processor. We also present the average message length. Average volume, maximum volume and average message length are all in Kb. One word is assumed to be 8 bytes, i.e., double precision. With the help of the results of the experiments provided in Section 2.2, we can have an idea if latency will be a determining factor in the overall communication cost. We can consider two cases: (i) the matrices that have small average and maximum volume, i.e., smaller than

Table 2.4: Communication statistics for 20 matrices partitioned into 512 parts.

	volume (Kb)		message		
matrix	avg	max	avg	max	length (Kb)
apache2	3.2	4.3	6	8	0.6
az2010	0.4	0.9	6	14	0.1
bcsstk30	1.6	3.0	11	22	0.1
coAuthorsDBLP	4.4	9.5	201	322	0.0
cop20k_A	2.4	3.6	12	19	0.2
crystk03	1.8	2.8	15	26	0.1
Ga3As3H12	12.3	35.2	113	340	0.1
gupta3	51.0	104.8	418	511	0.1
gyro_k	1.3	2.7	14	32	0.1
net150	19.6	29.8	246	494	0.1
netherlands_osm	0.2	0.5	5	12	0.0
pkustk09	1.0	1.6	7	14	0.1
qa8fk	1.8	2.6	12	21	0.1
raefsky4	1.5	2.5	15	34	0.1
roadNet-CA	0.6	1.2	6	14	0.1
ship_001	3.4	5.1	16	33	0.2
shipsec1	2.2	3.5	8	15	0.3
sparsine	6.2	10.2	93	245	0.1
srb1	1.3	1.9	6	11	0.2
struct3	0.7	1.0	6	9	0.1

4/8 Kb, and (ii) the matrices that have high number of average or maximum messages. There are several matrices that fall into the former case, for example the matrices az2010, bcsstk30, crystk03, srb1, to name a few. Example of the matrices that fall into the latter case contain matrices such as coAuthorsDBLP, Ga3As3H12 and sparsine. In these matrices, reducing latency should pay off as it is a major contributor to the overall communication cost. In the following chapters, we investigate models and methods to reduce the latency overhead by trying to reduce the average number of messages and/or maximum number of messages.

Chapter 3

Reducing latency cost in 2D sparse matrix partitioning models

Many scientific and engineering applications necessitate solving a linear system of equations. The methods used for this purpose are categorized as direct and iterative methods. When the linear system is large and sparse, iterative methods are preferred to their direct counterparts due to their speed and flexibility. Most widely used iterative methods for solving large-scale linear systems are based on Krylov subspace iterations.

A single iteration in Krylov subspace methods usually consists of one or more Sparse Matrix–Vector multiplications (SpMV), dot product(s) and vector updates. In a distributed setting, SpMV operations require regular or irregular point-to-point (P2P) communication depending on the sparsity pattern of the coefficient matrix in which each processor sends/receives messages to/from a subset of processors. On the other hand, dot products necessitate global communication that involves a reduction operation on one or a few scalars in which all processors participate. Vector updates usually do not require any communication.

see [16] for the original work

3.1 Related work

Communication requirements of iterative solvers have been of interest for more than three decades. There are numerous works on reducing communication overhead of global reduction operations in iterative solvers. Several works in this category aim at decreasing the number of global synchronization points in a single iteration of the solver by reformulating it [17, 18, 19, 20, 21, 22, 23, 24, 25]. Another important area of study is s-step Krylov subspace methods, which focus on further reducing the number of global synchronization points by a factor of s by performing only a single reduction once in every s iterations [21, 26, 27, 28, 29, 30]. The performance gain of s-step methods comes at the cost of deteriorated stability and complications related to integration of preconditioners. However, these methods recently gained popularity again and promising studies address these shortcomings [26, 29, 31, 32]. Another common technique is to overlap communication and computation with the aim of hiding global synchronization overheads [33, 34]. Especially with the introduction of nonblocking collective constructs in the MPI-3 standard, this technique is gaining attraction [35, 36, 37]. Overlapping is commonly used for SpMV operations as well. In addition, a recent work proposed hierarchical and nested Krylov methods that constrain global reductions into smaller subsets of processors where they are cheaper [38]. Another recent work uses the idea of embedding SpMV communications into global reductions to avoid latency overhead of SpMV communications [5].

The performance of iterative solvers is also addressed by minimizing communication costs related to parallel SpMV operations, which is also addressed by this work. There are studies that can handle sparse matrices that are well-structured and have predictable sparsity patterns, generally arising from 2D/3D problems [29, 39, 40, 41]. However, the studies in this field generally focus on combinatorial models that are capable of exploiting both regular and irregular patterns to obtain a good partition of the coefficient matrix. In this regard, graph and hypergraph partitioning models are widely utilized with successful partitioning tools such as MeTiS [15], PaToH [42], Scotch [43], Mondriaan [44]. These models can broadly be categorized as one-dimensional (1D) and two-dimensional

(2D) partitioning models. In 1D models [15, 42, 45, 46, 47, 48, 49, 12], each processor is responsible for a row/column stripe, whereas in 2D models, each processor may be responsible for a submatrix block (generally defined by a subset of rows and columns) or as in the most general case, each processor may be responsible for an arbitrarily defined subset of nonzeros. Compared to 1D models, 2D models possess more freedom in partitioning the coefficient matrix. Some works on 2D models do not take the communication volume into account, however they provide an upper bound on the number of messages communicated [50, 51, 52, 53, 54]. On the other hand, there are 2D models that aim at reducing volume, with or without providing a bound on the maximum number of messages [44, 55, 56, 57, 58, 59, 60]. 2D partitioning models in the literature can further be categorized into three classes: *checkerboard* partitioning [57, 59, 60] (also known as coarse-grain partitioning), *jagged* partitioning [55, 59] and *fine-grain* partitioning [56, 58, 59]. Notably, a recent work [60] proposes a fast 2D partitioning for scale-free graphs via a two-phase approach. This method uses 1D partitioning to reduce volume in the first phase and an efficient heuristic in the second phase to obtain a bound on the maximum number of message. This work differs from ours as it does not explicitly minimize the message count, instead, it uses a property of the Cartesian distribution of the matrices to provide the mentioned upper bound.

3.2 Motivation and contributions

Most of the aforementioned and other existing partitioning models optimize the objective of minimizing total communication volume, which is an effort to reduce bandwidth costs. However, communication cost is a function of both bandwidth and latency, with the latter being at least as important as the former, as the current trends indicate. The need for partitioning models that also consider other cost metrics has been noted in other works [5, 45]. There are a few notable works that focus on different communication cost metrics. Balancing communication volume is one of them [44, 61, 62]. More important and overlooked work targets multiple communication metrics including latency [63], on which this study is based. Compared to [63], this study concentrates more on practical aspects.

In this work, we claim and show that attempting to minimize a single communication objective hurts parallel performance and achieving a tradeoff between bandwidth and latency costs is the key factor for achieving scalability. The basic motivation is to employ a nonsymmetric partition in the solver. Note that in parallel SpMV operations of the form $w = Ap$, one needs to partition the input vector p and the output vector w in addition to A. This can be achieved either by using a symmetric partition where the same partition is imposed on both input and output vectors, or by using a nonsymmetric partition where a distinct partition is employed for input and output vectors. The latter alternative is more appealing and it should be adopted whenever convenient since it is more flexible and allows operating in a broader search space. A nonsymmetric partition can be utilized in nonsymmetric linear system solvers such as the conjugate gradient normal equation error (CGNE) [64, 65, 66] and residual method (CGNR) [64, 65, 66, 67], and the standard quasi-minimal residual (QMR) [68] where the coefficient matrix is square and nonsymmetric. We constrain ourselves to nonsymmetric square matrices in this work, but all proposed models apply to certain iterative methods that involve rectangular matrices as well.

Our work is based on [63], which also achieves a nonsymmetric partition through a two-phase methodology with a model called *communication hypergraph*. Our contributions and differences from [63] are as follows:

1. We propose two new partitioning models for reducing latency which are based on 2D checkerboard and jagged partitioning. These models aim at reducing latency costs usually at the expense of increasing bandwidth costs. Similar models have been investigated [58, 63], but they are based on 1D and 2D fine-grain models.

2. All proposed and investigated partitioning models are realized on two iterative methods CGNE and CGNR implemented with the widely adopted PETSc toolkit [69]. We describe how to obtain a nonsymmetric partition on the vectors utilized in these solvers using the communication hypergraph

model and thoroughly evaluate partitioning requirements of them via experiments. In this manner, we differ from [63], in which the proposed methods were tested with a code developed by the authors that contains only parallel SpMV computations.

3. We conduct extensive experiments for the mentioned iterative solvers. Although better suited to large-scale systems, the communication hypergraph model was originally tested only for 24 processors on a local cluster and only for 1D partitioning. In this work, we test and show this model's validity on a modern HPC system (a BlueGene/Q machine) successfully scaling up to 8K processors.

4. We compare one 1D-based, three 2D-based models (checkerboard, jagged and fine-grain), and these four models' latency-improved versions, making a total of eight partitioning models. Among these, the 2D models are somewhat overlooked in the literature, never being tested in a realistic setting on a large-scale system. Although their theoretical merits are of no question, their practical merits are not appreciated. In our experiments, we put these methods' practical aspects into a thorough analysis. The experiments show surprising results with 2D jagged partitioning and its latency-improved version performing better in the majority of the matrices.

The rest of this chapter is organized as follows. In Section 3.3, we give background about 1D partitioning requirements, the basic communication hypergraph model and partitioning vectors in solvers CGNE and CGNR. Sections 3.4.1 and 3.4.2 describe the proposed partitioning models to reduce the latency overhead of checkerboard and jagged models, respectively. These two sections describe basic checkerboard and jagged models as well. We also briefly review the fine-grain model and its latency-improved version in Section 3.4.3, since they are included in our experiments. We compare communication properties of all partitioning models in Section 3.5. Section 3.6 contains the results and discussions of the extensive large-scale experimental evaluation of eight partitioning models on a BlueGene/Q system with 28 matrices. Our experiments range from 256 to 8192 processors.

3.3 Preliminaries

In this section, we describe 1D partitioning requirements and the basic communication hypergraph model to reduce latency overheads.

3.3.1 Hypergraph partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$ [70]. Each net $n_j \in \mathcal{N}$ connects a subset of vertices, which are referred to as pins of n_j. The set of nets that connect vertex v_i is denoted by $Nets(v_i)$. The degree of a vertex is equal to the number of nets that connect this vertex, i.e., $d_i = |Nets(v_i)|$. A weight value w_i is associated with each vertex v_i.

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is called a K-way partition of vertex set $\mathcal{V}$ if each part V_k is non-empty, parts are pairwise disjoint and the union of K parts is equal to $\mathcal{V}$. In Π, a net is said to connect a part if it connects at least one vertex in that part. The set of parts connected by a net n_j is called its connectivity set and is denoted by $\Lambda(n_j)$. The connectivity $\lambda(n_j) = |\Lambda(n_j)|$ of n_j is equal to the number of parts connected by this net. Net n_j is said to be an internal net if it connects only one part ($\lambda(n_j) = 1$), and an external net if it connects more than one part ($\lambda(n_j) > 1$). In Π, the weight of a part is the sum of the weights of vertices in that part. In the hypergraph partitioning (HP) problem, the objective is to minimize the *cutsize*, which is defined as

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}} (\lambda(n_j) - 1). \tag{3.1}$$

This objective function is known as the connectivity-1 cutsize metric and is widely used in the scientific computing community [42, 71, 72]. The partitioning *constraint* is to satisfy a balance on part weights:

$$(W_{max} - W_{avg})/W_{avg} \leq \epsilon, \tag{3.2}$$

where W_{max} and W_{avg} are the maximum and the average part weights, respectively, and ϵ is the user-defined imbalance ratio. The HP problem is known to be

NP-hard [73]. Nonetheless, there exist successful HP tools such as PaToH [42], hMeTiS [74] and Mondriaan [44].

A variant of the hypergraph partitioning problem is the multi-constraint hypergraph partitioning problem [57, 75], where multiple weights are associated with vertices. The partitioning objective is the same as defined in (3.1); however, the partitioning constraint is extended to maintain a balancing constraint with each vertex weight.

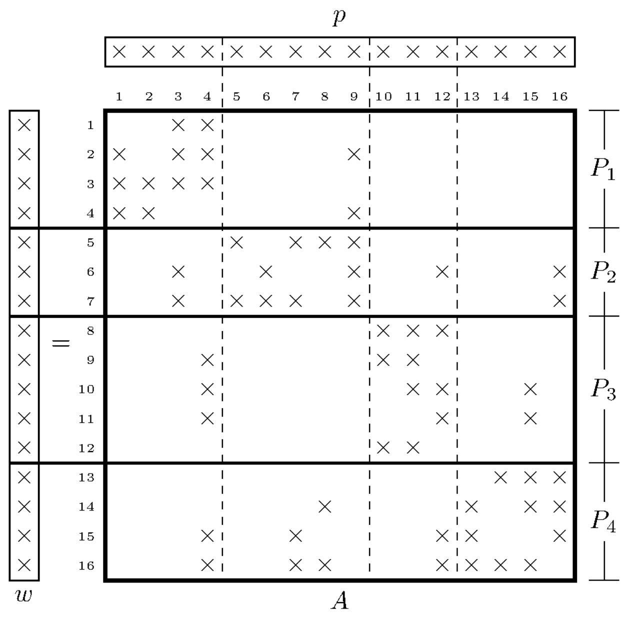
3.3.2 1D partitioning requirements

In 1D partitioning, $n \times n$ matrix A is partitioned either rowwise or columnwise. Assume that A is permuted into a $K \times K$ block structure as follows:

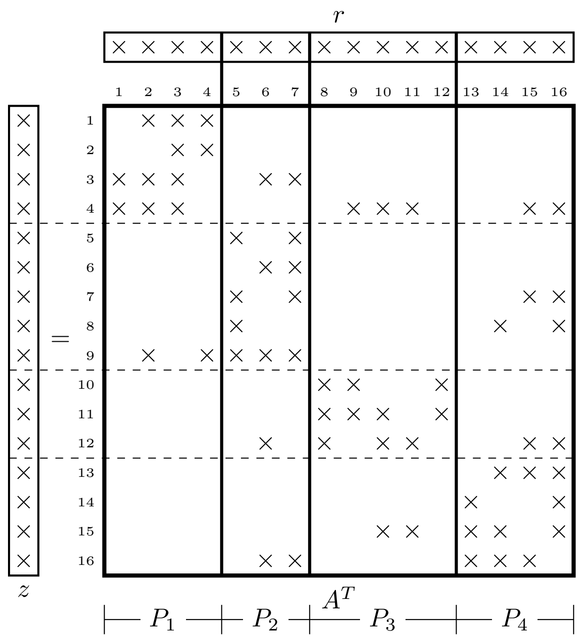
$$A_{BL} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1K} \\ A_{21} & A_{22} & \dots & A_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A_{K1} & A_{K2} & \dots & A_{KK} \end{bmatrix}, \tag{3.3}$$

where K denotes the number of processors in the parallel system and the size of block A_{kl} is $n_k \times n_l$. In rowwise partitioning, processor P_k is responsible for the kth row $[A_{k1} \dots A_{kK}]$ of size $n_k \times n$. In columnwise partitioning, processor P_k is responsible for the kth column block $[A_{1k}^T \dots A_{Kk}^T]^T$ of size $n \times n_k$. Throughout this section, without loss of generality, we assume a rowwise partition of A.

The vectors in an iterative solver should be partitioned conformally in order to avoid redundant communication during linear vector operations. For example, in the conjugate gradient solver, all vectors are partitioned conformally. In some solvers, we can utilize distinct vector partitions that separately apply to certain vectors. For example in CGNE and CGNR, it is possible to utilize two distinct partitions on the vectors. This enables utilization of a nonsymmetric partition for the coefficient matrix (see 3.3.4 for the details). The main motivation for adopting a nonsymmetric partition is that instead of enforcing the same partition on all vectors in the solver, we have more freedom by using a different partition on each distinct vector space – which accommodates more potential for reducing communication overheads in parallelization.



(a) Row-parallel $w = Ap$.



(b) Column-parallel $z = A^T r$.

Figure 3.1: Row-parallel matrix-vector and column-parallel matrix-transpose-vector multiplication.

In a parallel solver, inner product operations necessitate global collective communications whereas matrix-vector or matrix-transpose-vector multiplications necessitate P2P communications. Consider parallel $w = Ap$ and $z = A^T r$ multiplies. An example for these operations is illustrated in Figure 3.1 for $K = 4$ processors. Without loss of generality, assume that P_k is responsible for the kth row stripe of A, and thus the kth column stripe of A^T. Note that a rowwise partition on A induces a columnwise partition on A^T (3.3.4). Here, $w = Ap$ is performed with the row-parallel algorithm while $z = A^T r$ is performed with the column-parallel algorithm. The *row-parallel* algorithm necessitates a *pre-communication* stage in which the input vector elements are communicated. Each P_k sends the input vector elements that correspond to the nonzero *column* segments in off-diagonal blocks A_{ik}, $1 \leq i \neq k \leq K$. This is also referred to as the *expand* operation since the same vector element can be sent to multiple processors. The vector elements that correspond to the columns which have at least one nonzero column segment in off-diagonal blocks (called coupling columns) necessitate expand operations. In Figure 3.1a, eight elements of the input vector ($p[3]$, $p[4]$, $p[7]$, $p[8]$, $p[9]$, $p[12]$, $p[15]$, $p[16]$) need to be communicated. For example, P_3 sends $p[12]$ to P_2 and P_4, which need this element in their local computations. On the other hand, the *column-parallel* algorithm necessitates a *post-communication* stage in which the partial results of the output vector elements are communicated. Each P_k receives the output vector entries that correspond to the nonzero *row* segments in off-diagonal blocks A_{kj}, $1 \leq j \neq k \leq K$. This is also referred to as the *fold* operation since the partial results for the same vector element can be received from multiple processors. The vector elements that correspond to the rows which have at least one nonzero row segment in off-diagonal blocks (called coupling rows) necessitate fold operations. In Figure 3.1b, eight elements of the output vector ($z[3]$, $z[4]$, $z[7]$, $z[8]$, $z[9]$, $z[12]$, $z[15]$, $z[16]$) need to be communicated. For example, P_3 receives partial results for $z[12]$ from P_2 and P_4 to compute the final value of $z[12]$. Observe that the communication of $p[12]$ in the row-parallel algorithm is the dual of the communication of $z[12]$ in the column-parallel algorithm.

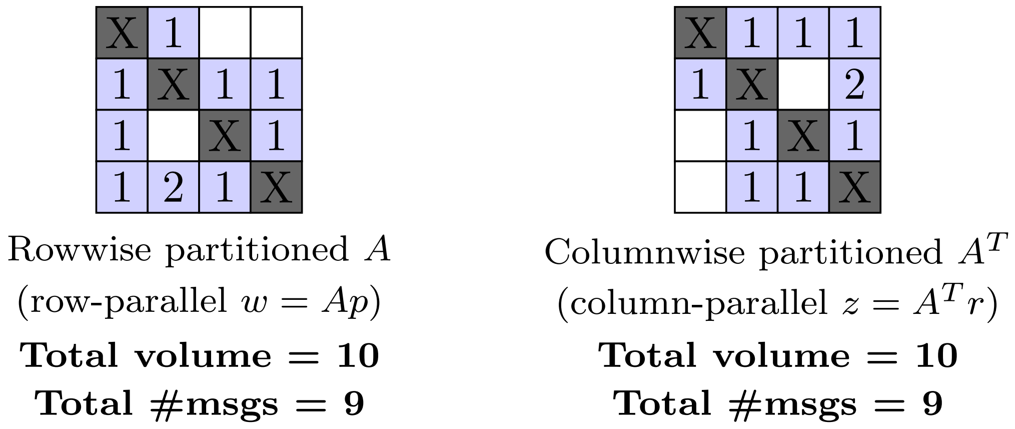


Figure 3.2: Communication block forms for 1D partitioning.

3.3.2.1 Matrix View

In matrix theoretical view, each off-diagonal block A_{kl} or A^T_{lk} with at least one nonzero necessitates a P2P message between processors P_k and P_l. In row-parallel algorithm, a non-empty off-diagonal block A_{kl} with x nonzero column segments requires a message from P_k to P_l with x words. In a dual manner, in column-parallel algorithm, a non-empty off-diagonal block A^T_{lk} with y nonzero row segments requires a message from P_k to P_l with y words. In Figure 3.1a, nine off-diagonal blocks A_{12}, A_{21}, A_{23}, A_{24}, A_{31}, A_{34}, A_{41}, A_{42}, A_{43} necessitate nine P2P messages, where the message corresponding to block A_{42} contains two words while each of the remaining eight messages contain one word, making a total of ten words of communication. Similarly, in Figure 3.1b, nine off-diagonal blocks A^T_{12}, A^T_{13}, A^T_{14}, A^T_{21}, A^T_{24}, A^T_{32}, A^T_{34}, A^T_{42}, A^T_{43} necessitate nine P2P messages, where the message corresponding to block A^T_{24} contains two words while each of the remaining eight messages contain one word, making a total of ten words of communication as well. The communication requirements of the 1D partitioning on example A and A^T matrices are summarized in Figure 3.2 as communication block forms (CBFs). In the example, the shaded blocks indicate the blocks that necessitate P2P messages with the numbers on them being the size of these messages in words. Note that total number of messages and total communication volume in row-parallel $w = Ap$ and column-parallel $z = A^T r$ are equal to each other since these operations are dual of each other.

3.3.3 Two computational hypergraph models for 1D sparse matrix partitioning

There are several ways of obtaining a 1D rowwise/columnwise partitioning of coefficient matrix A. We briefly discuss two hypergraph models since they are central to the models proposed in this work. These models are also referred to as *computational hypergraph* models.

The column-net hypergraph model $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, \mathcal{N}_\mathcal{C})$ can be used to obtain a rowwise partitioning of A [42]. In this model, vertex set $\mathcal{V}_\mathcal{R}$ represents the rows of A and net set $\mathcal{N}_\mathcal{C}$ represents the columns of A. There is a vertex $v_i \in \mathcal{V}_\mathcal{R}$ for each row r_i and there is a net $n_j \in \mathcal{N}_\mathcal{C}$ for each column c_j. Net n_j connects a subset of vertices that correspond to the rows that have a nonzero element in column c_j, i.e., $v_i \in n_j$ if and only if $a_{ij} \neq 0$. The weight w_i of vertex v_i is equal to the number of nonzeros in row r_i and represents the computational load associated with v_i.

The row-net hypergraph model $\mathcal{H}_\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{N}_\mathcal{R})$ can be used to obtain a columnwise partitioning of A [42]. In this model, vertex set $\mathcal{V}_\mathcal{C}$ represents the columns of A and net set $\mathcal{N}_\mathcal{R}$ represents the rows of A. There is a vertex $v_j \in \mathcal{V}_\mathcal{C}$ for each column c_j and there is a net $n_i \in \mathcal{N}_\mathcal{R}$ for each row r_i. Net n_i connects a subset of vertices that correspond to the columns that have a nonzero element in row r_i, i.e., $v_j \in n_i$ if and only if $a_{ij} \neq 0$. The weight w_j of vertex v_j is equal to the number of nonzeros in column c_j and represents the computational load associated with v_j.

Partitioning hypergraphs $\mathcal{H}_\mathcal{C}$ and $\mathcal{H}_\mathcal{R}$ with the objective of minimizing cutsize corresponds to minimizing total communication volume incurred in parallel sparse-matrix vector multiplication while maintaining the partitioning constraint on part weights corresponds to maintaining a balance on computational loads of processors.

Algorithm 1: CGNE and CGNR.

```
  Set initial $\mathbf{x}_0$
  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
  $\mathbf{p}_0 = \mathbf{A}^T\mathbf{r}_0$
1 for $i = 0, 1, \ldots$ do
2     $\alpha_i = \langle \mathbf{r}_i, \mathbf{r}_i \rangle / \langle \mathbf{p}_i, \mathbf{p}_i \rangle$    ▷ CGNE
      $\alpha_i = \langle \mathbf{A}^T\mathbf{r}_i, \mathbf{A}^T\mathbf{r}_i \rangle / \langle \mathbf{A}\mathbf{p}_i, \mathbf{A}\mathbf{p}_i \rangle$ ▷ CGNR
3     $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i\mathbf{p}_i$
4     $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i\mathbf{A}\mathbf{p}_i$
5     $\beta_i = \langle \mathbf{r}_{i+1}, \mathbf{r}_{i+1} \rangle / \langle \mathbf{r}_i, \mathbf{r}_i \rangle$    ▷ CGNE
      $\beta_i = \langle \mathbf{A}^T r_{i+1}, \mathbf{A}^T r_{i+1} \rangle / \langle \mathbf{A}^T\mathbf{r}_i, \mathbf{A}^T\mathbf{r}_i \rangle$ ▷ CGNR
6     $\mathbf{p}_{i+1} = \mathbf{A}^T\mathbf{r}_{i+1} + \beta_i\mathbf{p}_i$
```

3.3.4 Partitioning vectors in CGNE and CGNR solvers

We describe why it is possible to use different partitions on the vectors used in CGNE and CGNR solvers. For other solvers, refer to [76]. We make the distinction between input and output space for the vectors in the solver. A vector is said to be in the *input* space of A if it is multiplied with A or it participates with the vectors in the input space of A through linear vector operations. On the other hand, a vector is said to be in the *output* space of A if it is obtained by multiplying A with another vector or it participates with the vectors in the output space of A through linear vector operations.

We present CGNE and CGNR algorithms in Algorithm 1. In each iteration of the solvers, there are two inner products, three SAXPY operations (for forming vectors p, r, x), one matrix-vector multiply of the form $w = Ap$ and one matrix-transpose-vector multiply of the form $z = A^T r$. In $w = Ap$, vectors p and w are in the input and output space of A, respectively. In $z = A^T r$, vectors r and z are in the input and output space of A^T, respectively. Consider a rowwise (columnwise) partition of A. This induces a columnwise (rowwise) partition on A^T. Hence, the input space of A coincides with the output space of A^T, and vice versa. This implies that the partition on vector p is conformal with the partition on vector z, and the partition on vector w is conformal with the partition on vector r. Since x is involved in linear vector operations with vector p (line 3), it should be partitioned conformally with vectors p and z to avoid unnecessary communication. As a result, we can have two distinct vector partitions in CGNE and CGNR: one on vectors p, z and x, and another one on vectors w and r.

3.3.5 Communication hypergraph model

The communication hypergraph (CHG) model [63] is a means of distributing communication tasks among processors with the aim of minimizing latency. A communication task is defined as a subset of processors that involve in communicating a data object with a certain size. The CHG model strives to reduce the total number messages usually at the expense of increasing communication volume. However, although it increases the volume, it tries to obtain a balance on it. Reducing latency is a key factor to achieve scalability in large-scale systems as we show with our experiments. In this section, we review the CHG model for reducing latency overhead of 1D partitioned parallel $w = Ap$ and $z = A^T r$ multiplies.

3.3.5.1 Communication matrix

As the first step, we form communication matrices M_R and M_C to summarize the communication requirements of row-parallel $w = Ap$ and column-parallel $z = A^T r$, respectively. For row-parallel $w = Ap$, let p_C denote the p-vector elements that necessitate communication (via expand tasks). Communication matrix M_R is then a $K \times |p_C|$ matrix where the rows of M_R correspond to processors and the columns of M_R correspond to expand communication tasks. In M_R, $m_{kj} \neq 0$ if and only if the corresponding coupling column c_j has a nonzero column segment in the kth row stripe of A. For example in A (Figure 3.1a), column 12 has a nonzero at the second row stripe, thus there exists a nonzero at the corresponding entry

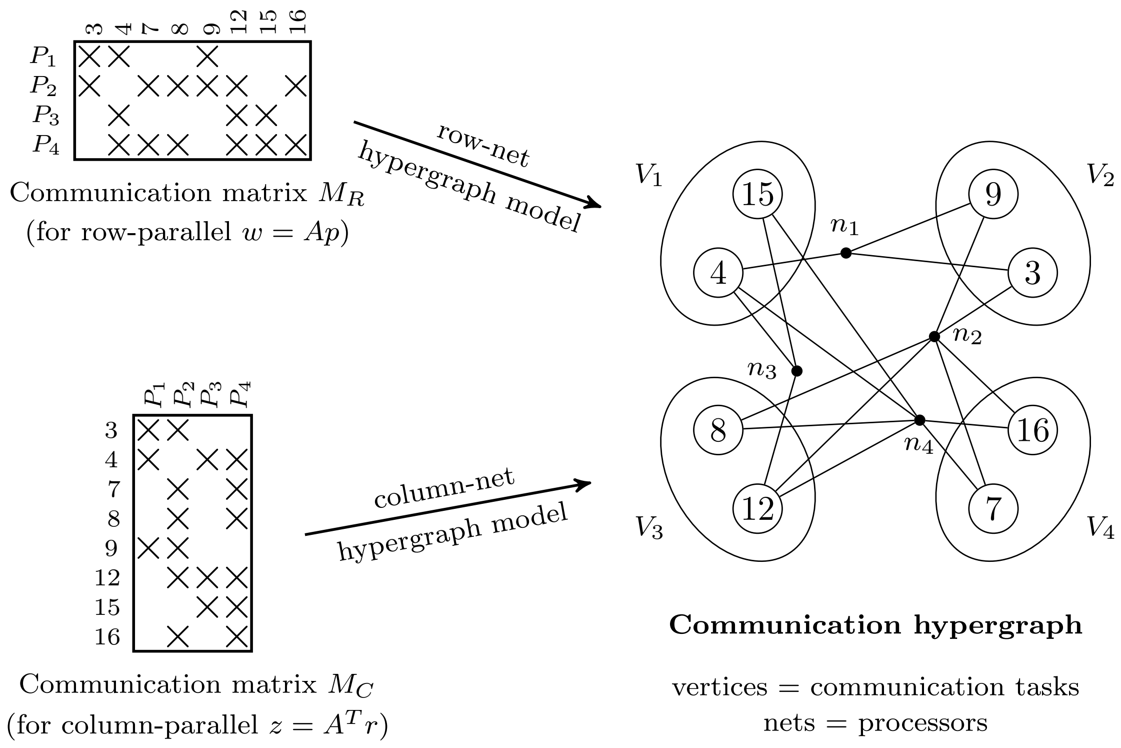

3 4 7 8 9 12 15 16
P_1 P_2 P_3 P_4
Communication matrix M_R
(for row-parallel $w = Ap$)
row-net hypergraph model
column-net hypergraph model
Communication matrix M_C
(for column-parallel $z = A^T r$)
V_1 V_2 V_3 V_4
n_1 n_2 n_3 n_4
Communication hypergraph
vertices = communication tasks
nets = processors

Figure 3.3: Formation of the communication hypergraph from communication matrix, and a four-way partition on this hypergraph. Matrices M_R and M_C summarize the communication requirements of $w = Ap$ and $z = A^T r$ operations illustrated in Figure 3.1.

in M_R in Figure 3.3 at the intersection of row P_2 and column 12. The nonzeros of column $c_j \in M_R$ signify the set of processors that participate in communicating $p_C[j]$. The nonzeros of row $r_k \in M_R$ signify all expand tasks that P_k takes part in. In Figure 3.3, the third row in M_R has nonzero elements corresponding to columns 4, 12 and 15, indicating that P_3 is involved in communicating $p[4]$, $p[12]$ and $p[15]$. Hence, a nonzero $m_{kj} \in M_R$ actually implies that P_k participates in the communication of $p_C[j]$.

For column-parallel $z = A^T r$, let z_C denote the z-vector elements that necessitate communication (via fold tasks). Communication matrix M_C is then a $|z_C| \times K$ matrix where the rows of M_C correspond to fold communication tasks and the columns of M_C correspond to processors. In M_C, $m_{ik} \neq 0$ if and only if the corresponding coupling row r_i has a nonzero row segment in the kth column stripe of A^T. For example in A^T (Figure 3.1b), row 12 has a nonzero at the second column stripe, thus there exists a nonzero at the corresponding entry in M_C in Figure 3.3 at the intersection of row 12 and column P_2. The nonzeros of row $r_i \in M_C$ signify the set of processors that participate in communicating $z_C[i]$. The nonzeros of column $c_k \in M_C$ signify all fold tasks that P_k takes part in. In Figure 3.3, the third column in M_C has nonzero elements corresponding to rows 4, 12 and 15, indicating that P_3 is involved in communicating $z[4]$, $z[12]$ and $z[15]$. Hence, a nonzero $m_{ik} \in M_C$ actually implies that P_k participates in the communication of $z_C[i]$.

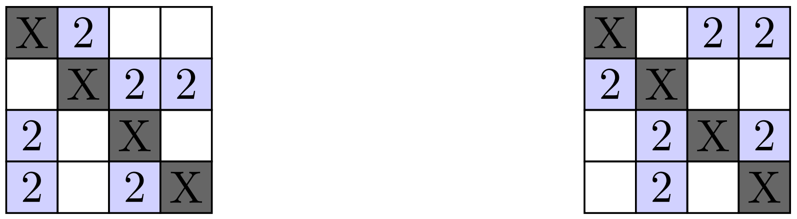
3.3.5.2 Formation of communication hypergraph

The communication matrix is then used to form a hypergraph called *communication hypergraph*. We apply the row-net hypergraph model to communication matrix M_R (vertices = columns, nets = rows) to obtain the communication hypergraph $\mathcal{H}_R^{CM}$ and we apply the column-net hypergraph model to communication matrix M_C (vertices = rows, nets = columns) to obtain the communication hypergraph $\mathcal{H}_C^{CM}$. The vertex and net set of both hypergraphs are the same (see Figure 3.3). In both hypergraphs, nets correspond to processors (there are K of them) and vertices correspond to communication tasks (there are $|p_C| = |z_C|$ of them). However, the semantics of these hypergraphs differ: the vertices in $\mathcal{H}_R^{CM}$ represent *expand* tasks in $w = Ap$, while the vertices in $\mathcal{H}_C^{CM}$ represent *fold* tasks in $z = A^T r$. A net n_k in both hypergraphs connects the set of vertices that correspond to communication tasks P_k participates in. Each vertex v_i is associated with a weight that signifies the volume of communication incurred by the corresponding expand or fold task. This value is generally equal to one less than the number of the nets v_i is connected by, i.e., $d_i - 1$.

3.3.5.3 Partitioning of the communication hypergraph

Obtaining a K-way partition $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ on $\mathcal{H}_R^{CM}$ or $\mathcal{H}_C^{CM}$ induces a communication task distribution for parallel matrix-vector or matrix-transpose-vector multiplies. Without loss of generality, assume that processor P_k is associated with part $\mathcal{V}_k$. Expand or fold communication tasks represented by the vertices in $\mathcal{V}_k$ are assigned to P_k by making this processor responsible for storing vector elements that necessitate these tasks. For instance in Figure 3.3, since $v_{12} \in \mathcal{V}_3$, P_3 is held responsible for storing $p[12]$ and $z[12]$, and expand and fold tasks necessitated by these elements. Consider a net n_k in $\mathcal{H}_R^{CM}$ with the connectivity set $\Lambda(n_k)$. All parts except $\mathcal{V}_k$ in this set correspond to the processors that send a message to P_k, hence, $\lambda(n_k) - 1$ (or $\lambda(n_k)$ if $\mathcal{V}_k \notin \Lambda(n_k)$) is equal to the number of messages P_k *receives*. In a dual manner, consider the same net in $\mathcal{H}_C^{CM}$ again with the connectivity set $\Lambda(n_k)$. All parts except $\mathcal{V}_k$ in this set correspond to the processors that receive a message from P_k, hence, $\lambda(n_k) - 1$ (or $\lambda(n_k)$ if $\mathcal{V}_k \notin \Lambda(n_k)$) is equal to the number of messages P_k *sends*. In Figure 3.3, the connectivity sets of nets are as follows: $\Lambda(n_1) = \{\mathcal{V}_1, \mathcal{V}_2\}$, $\Lambda(n_2) = \{\mathcal{V}_2, \mathcal{V}_3\, \mathcal{V}_4\}$, $\Lambda(n_3) = \{\mathcal{V}_1, \mathcal{V}_3\}$ and $\Lambda(n_4) = \{\mathcal{V}_1, \mathcal{V}_3, \mathcal{V}_4\}$, making a total of $(\lambda(n_1) - 1 = 1) + (\lambda(n_2) - 1 = 2) + (\lambda(n_3) - 1 = 1) + (\lambda(n_4) - 1 = 2) = 6$ messages. In [63], it is proven that partitioning a communication hypergraph with the aim of minimizing cutsize minimizes the total message count, while maintaining a balance among part weights preserves a balance on the communication volume.

By applying the CHG model, we obtain a different partition on rows and columns of the coefficient matrix and thus on its input and output space (for details, see 3.3.5.5). Adopting a different partition finds its application in non-symmetric sparse iterative solvers that allow distinct partitions on vectors and can be used to improve their scalability.



Rowwise partitioned A
(row-parallel $w = Ap$)
Total volume = 12
Total #msgs = 6

Columnwise partitioned A^T
(column-parallel $z = A^T r$)
Total volume = 12
Total #msgs = 6

Figure 3.4: Communication block forms after appplying CHG model.

3.3.5.4 Matrix View

The aim of CHG model is to reduce the number of non-empty off-diagonal blocks at the expense of increasing nonzero row/column segments in these blocks. Recall that each such block causes a P2P message to be communicated. In Figure 3.4, compare the final CBFs that are obtained after applying CHG model with the initial CBFs in Figure 3.2. For both $w = Ap$ and $z = A^T r$, number of non-empty off-diagonal blocks in the initial CBFs is nine. After applying CHG model and distributing the communication tasks, this value becomes six as seen in the final CBFs. CHG model reduces the latency overhead by reducing number of messages communicated, however, this generally leads to more off-diagonal row/-column segments in the matrix. Such segments necessitate vector entries to be communicated. This, as a result, corresponds to an increase in total volume of communication. Again comparing the initial and final CBFs in the example, observe that the communication volume increases from ten words to twelve words.

3.3.5.5 Obtaining different vector partitions using communication hypergraph model

The communication hypergraph (CHG) model can be regarded as a post-processing phase to distribute the communication tasks. Although any partitioning model could be used, assume that the column-net computational hypergraph model is used in the first phase for $w = Ap$ (3.3.3). As computational tasks are represented by vertices that correspond to rows of the matrix, partitioning this hypergraph actually induces a partition of the rows of the matrix. The result of this partitioning determines the communication tasks in the second phase. Applying the CHG in the second phase induces a partition of the columns of the matrix since communication tasks represented by the vertices are expand-type tasks performed on input vector elements. Hence, in the first phase, we minimize the communication volume by obtaining a partition of the rows, and in the second phase we minimize the number of messages by obtaining a different partition of the columns. An example of these two phases can be traced from Figures 3.1a and 3.3. Assume that the result of the first phase is the partition obtained on A in Figure 3.1a. Here, P_3 owns rows $8, 9, 10, 11$, 12 and columns 10, 11, 12. The column partition is subject to change after applying the CHG model, which assigns vertices 8 and 12 to $\mathcal{V}_3$, thus assigning columns 8 and 12 to P_3. Since columns 10 and 11 do not necessitate communication, they are not included in the CHG model and directly assigned to P_3 at the end of the first phase. As a result, P_3 owns columns 8, 10, 11, 12. Hence, we obtained a nonsymmetric partition of the rows and columns assigned to P_3.

3.4 Reducing latency cost in 2D partitioning models

2D models work at a finer level of partitioning granularity compared to 1D models by allowing nonzeros of a single row/column to be assigned to more than one processor. In this manner, they possess more flexibility in partitioning since they do not constrain the search space by assigning all nonzeros of a row/column to the same processor. This leads them to exploit existing partitioning tools better. 1D models necessitate a row-parallel/column-parallel algorithm (3.3.2), whereas 2D models necessitate a row-column-parallel algorithm. The fundamental difference between them is that the former necessitates a single communication stage in parallel SpMV operations, which is either pre-communication if the matrix is partitioned rowwise, or post-communication if the matrix is partitioned columnwise, whereas the latter necessitates two distinct communication stages: one before the local SpMV computations (on the input vector in a *pre*-communication stage via *expand* communication tasks) and one after the local SpMV computations (on the output vector in a *post*-communication stage via *fold* communication tasks), For more details on implementation issues regarding 2D partitioning, see [59].

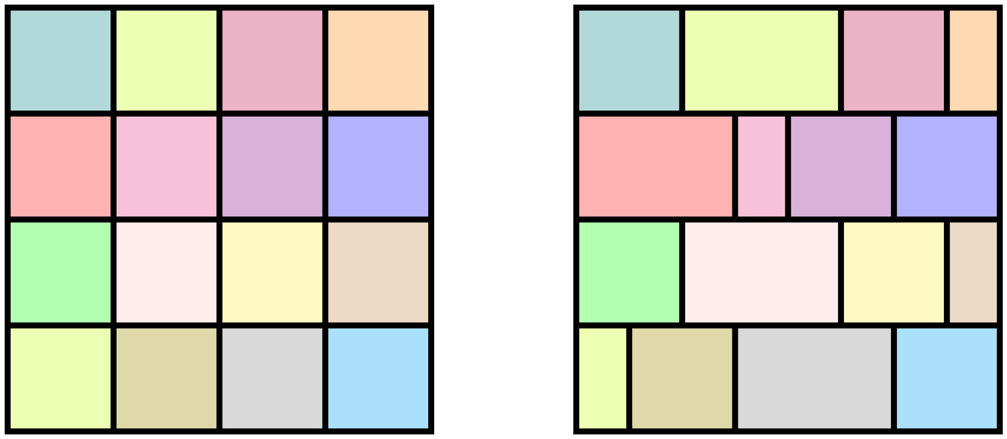

(a) Checkerboard (b) Jagged

Figure 3.5: A sample of 2D checkerboard and jagged partitionings on a $16 = 4 \times 4$ virtual processor mesh.

In this section, we describe how to reduce latency overhead of 2D checkerboard and jagged partitioning models. For these models, we assume a $K = P \times Q$ virtual processor mesh. A simple example depicting a matrix partitioned with these two models are given in Figure 3.5. Compared to their original counterparts, the proposed models are likely to increase the bandwidth costs by increasing communication volume. However, this issue is addressed by maintaining a balance on this metric. We also briefly review the 2D fine-grain model and compare it to checkerboard and jagged models as we evaluate it in our experiments. Note that a model to reduce latency overhead of fine-grain partitioning has already been investigated [58]. All proposed models are discussed on parallel $w = Ap$. However, the arguments are also valid for $z = A^T r$ as communication requirements of $w = Ap$ and $z = A^T r$ are the dual of each other and minimizing the objective function in $w = Ap$ is equivalent to minimizing the objective function in $z = A^T r$.

3.4.1 Checkerboard partitioning

Checkerboard partitioning is a two-phase process in which each phase utilizes a 1D partitioning model. The second phase depends on the first phase by using information obtained in the first phase to determine multiple vertex weights utilized in the second phase. For the rest of the section, we assume a 1D rowwise partition in the first phase and a 1D columnwise partition in the second phase. For the arguments made in this section, an analogous discussion holds for the dual scheme as well.

Consider a $K = P \times Q$ processor mesh and an $n \times n$ square matrix A. In the first phase, the column-net hypergraph model $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ is used to obtain a P-way partition $\Pi_{\mathcal{R}} = \{\mathcal{V}_1, \ldots, \mathcal{V}_P\}$, which induces a P-way rowwise partition $\{\mathcal{R}_1, \ldots, \mathcal{R}_P\}$ of A. Here, $\mathcal{R}_\alpha$ denotes the set of rows that correspond to vertices in $\mathcal{V}_\alpha$, for $\alpha = 1, \ldots, P$. The rows in $\mathcal{R}_\alpha$ form a row stripe A_α whose size is $n_\alpha \times n$, with $n_\alpha = |\mathcal{V}_\alpha|$. At the end of the first phase, the assignment of rows of A is determined by associating row stripe A_α with the Q processors in row α of the processor mesh, $P_{\alpha,*}$.

In the second phase, the row-net hypergraph model $\mathcal{H}_{\mathcal{C}} = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{R}})$ is used to obtain a Q-way partition $\Pi_{\mathcal{C}} = \{\mathcal{V}_1, \ldots \mathcal{V}_Q\}$, which induces a Q-way columnwise partition $\{\mathcal{C}_1, \ldots, \mathcal{C}_Q\}$ of A. Here, $\mathcal{C}_\beta$ denotes the set of columns that correspond to vertices in $\mathcal{V}_\beta$, for $\beta = 1, \ldots, Q$. The columns in $\mathcal{C}_\beta$ form column stripe A_β whose size is $n \times n_\beta$, with $n_\beta = |\mathcal{V}_\beta|$. At the end of the second phase, we complete the assignment of columns of A and actually obtain a Q-way columnwise partition of each row stripe A_α, forming Q submatrix blocks $A_{\alpha,1}, \ldots, A_{\alpha,Q}$. Hence, $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$ defines an assignment for rows and columns of A where processor $P_{\alpha,\beta}$

is responsible for the set of rows in $\mathcal{R}_\alpha$ and the set of columns in $\mathcal{C}_\beta$. In other words, nonzero a_{ij} is assigned to $P_{\alpha,\beta}$ if $r_i \in \mathcal{R}_\alpha$ and $c_j \in \mathcal{C}_\beta$.

This two-phase process aims at minimizing total communication volume for the pre- and post-communication stages in the first and second phases, respectively, while maintaining computational load balance [57, 59]. A notable property of checkerboard partitioning is that it confines the communication in expand and fold operations to the processors in the same column and row of the processor mesh, respectively. It achieves a Cartesian distribution of the matrix, in which each processor owns an intersection of a subset of rows and a subset of columns. A row (column) is said to be coherent if the nonzeros of this row (column) generate partial results for (require) the same w-vector (x-vector) element. Consider a row r_i that is assigned to $\mathcal{R}_\alpha$ at the end of the first phase. The coherency of this row is preserved at this point as it is modeled by v_i in $\mathcal{H}_\mathcal{R}$. In the second phase, the nonzeros of this row can be distributed among Q processors in row α of the processor mesh, which is also the case for all other rows in $\mathcal{R}_\alpha$. Hence, row coherency is respected in a *coarse* level by assigning nonzeros of rows in $\mathcal{R}_\alpha$ to the processors in the same row of the processor mesh, $P_{\alpha,*}$. A coarse level here implies that the nonzeros belonging to a subset of rows are distributed among the same subset of processors (in this case among P processors in a specific column of the processor mesh). This provides the upper bound $Q-1$ on the number of messages communicated in the post-communication stage as there are Q processors in row α of the processor mesh. With a similar argument, column coherency is also respected in a coarse level by assigning nonzeros of columns in $\mathcal{C}_\beta$ to the processors in the same column of the processor mesh, $P_{*,\beta}$. This provides the upper bound $P-1$ on the number of messages communicated in the pre-communication stage as there are P processors in column β of the processor mesh. Hence, the maximum number of messages handled by a single processor is bounded by $P+Q-2$. In checkerboard partitioning, the second phase is performed with P-way multi-constraint [74, 77] partitioning to balance computational loads.

Given a K-way checkerboard partition of A realized on a $P \times Q$ processor mesh, we use CHG (communication hypergraph) model to reduce communication costs in pre- and post-communication phases separately. Recall that we can classify the set of rows and columns as internal and coupling, where internal rows or columns have nonzeros in only a single part, while coupling rows or columns have nonzeros in more than one part.

3.4.1.1 Communication matrices

The expand communication tasks in the checkerboard model are bound to distinct columns of the processor mesh. For this reason, to summarize the communication requirements of expand tasks in the pre-communication stage, we form Q distinct communication matrices. Let p_{C_β} denote the vector elements that necessitate communication in column β of the processor mesh, for $1 \leq \beta \leq Q$. Note that at most P processors can participate in communicating p_{C_β}, confined to the set of processors in $P_{*,\beta}$. We summarize the communication operations in column β of the mesh with the $P \times |p_{C_\beta}|$ communication matrix M_β. Rows of M_β correspond to processors in $P_{*,\beta}$ and columns of M_β correspond to expand tasks on p_{C_β}. There exists a nonzero $m_{\alpha j} \in M_\beta$ if and only if there is a non-empty column segment in submatrix $A_{\alpha,\beta}$ at the respective column. The nonzeros in column j of M_β represent the set of processors that participate in communicating $p_{C_\beta}[j]$, which is a subset of processors in $P_{*,\beta}$. The nonzeros in row α of M_β represent the expand tasks processor $P_{\alpha,\beta}$ participates in. Note that vector elements corresponding to internal columns (those which have a single non-empty column segment in P row stripes) do not incur communication and they are not included in M_β. These vector elements should be assigned to the respective processors to avoid unnecessary communication. An example in Figure 3.6 is presented to illustrate the formation of communication matrix M_β for the third column of the processor mesh ($\beta = 3$) to summarize the expand tasks. There are four input vector elements that necessitate communication (denoted by p_{C_β}) and they form columns of M_β. For instance, the first column of M_β has nonzeros corresponding to processors $P_{1,3}, P_{2,3}$ and $P_{4,3}$ since in matrix A, there exist nonzero column segments in the respective submatrix blocks. Two vector elements–second and fifth–corresponding to internal columns do not incur communication and they are

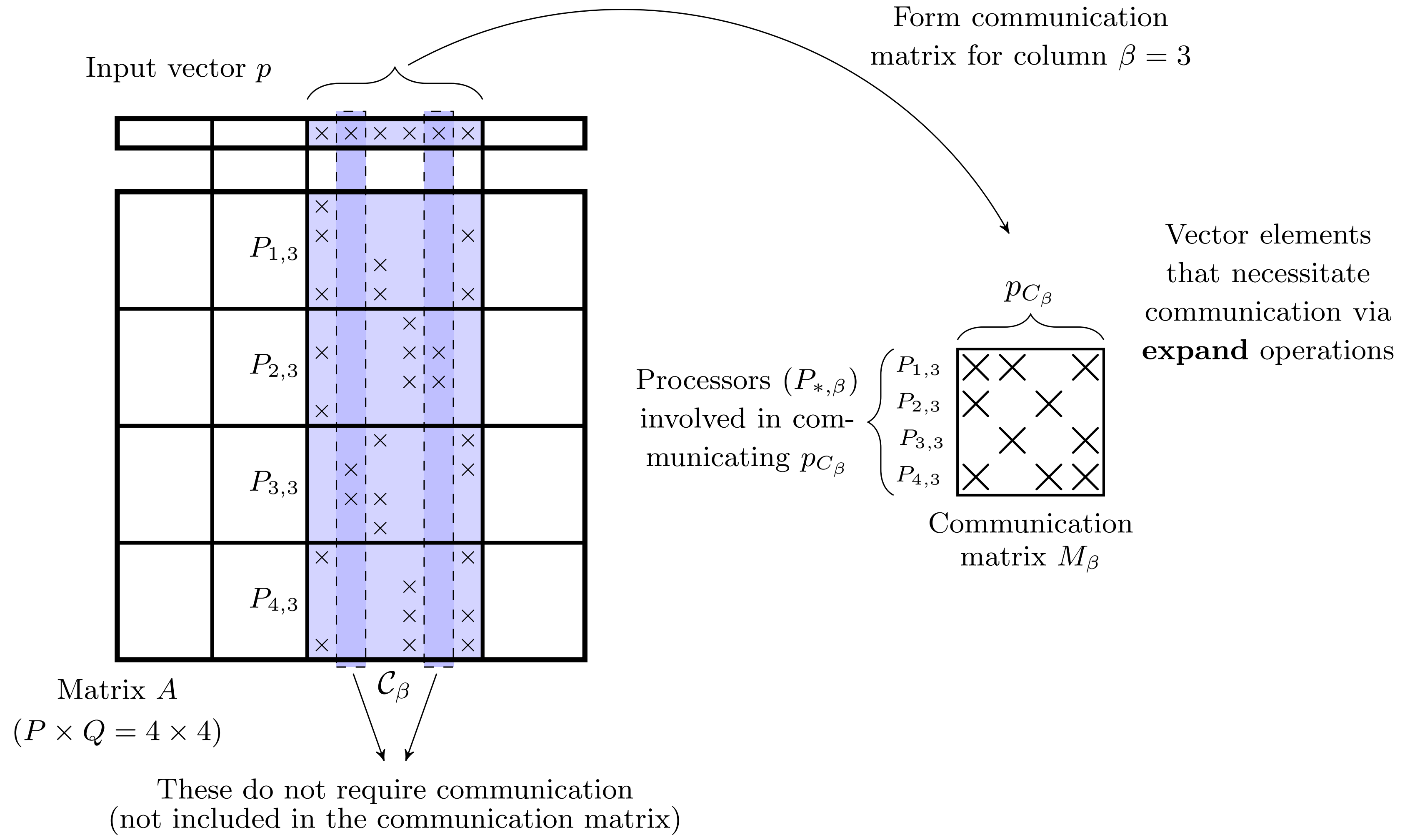
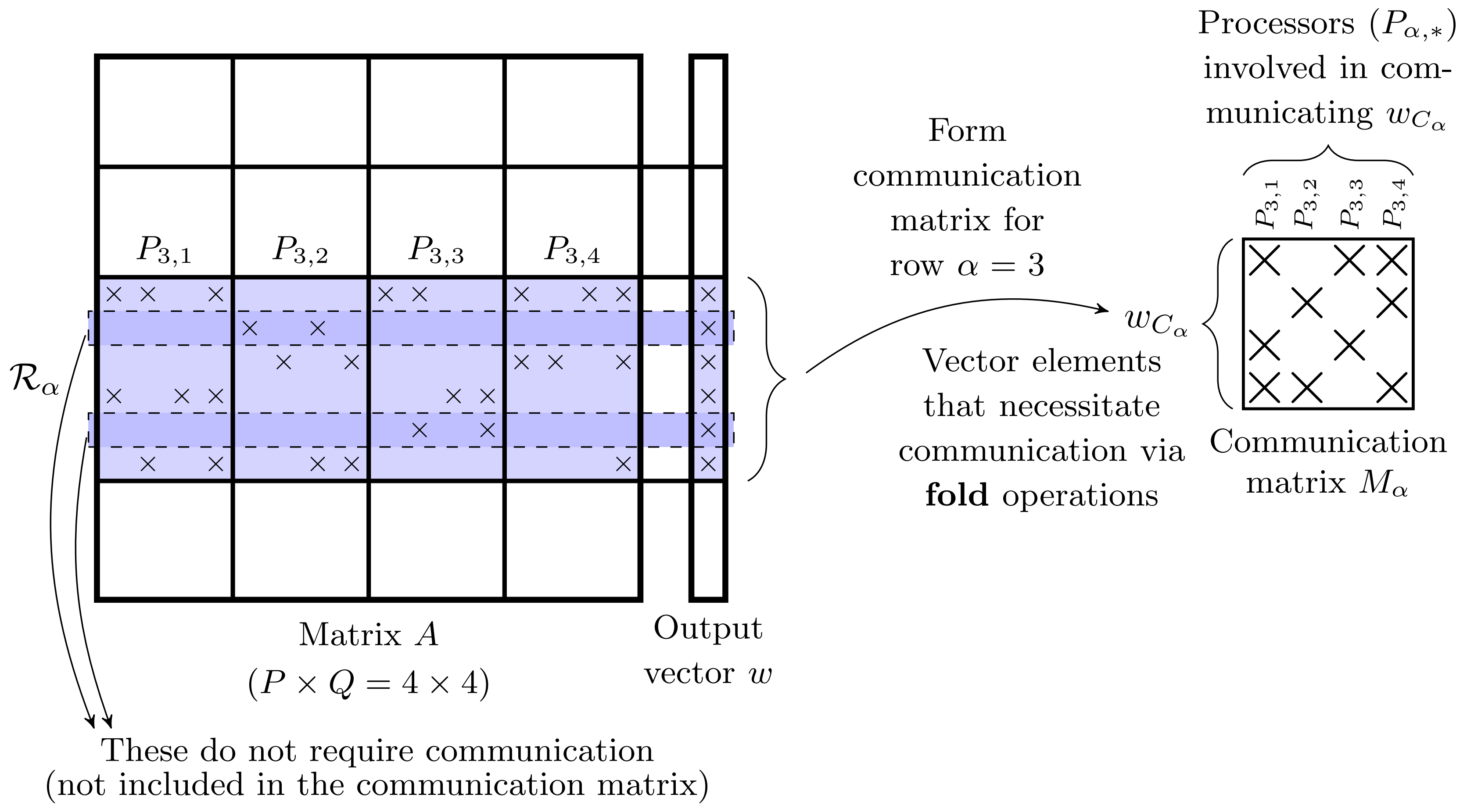

Input vector p
Form communication matrix for column $\beta = 3$
$P_{1,3}$
$P_{2,3}$
$P_{3,3}$
$P_{4,3}$
Matrix A $(P \times Q = 4 \times 4)$
$\mathcal{C}_\beta$
These do not require communication (not included in the communication matrix)
Processors $(P_{*,\beta})$ involved in communicating p_{C_β}
p_{C_β}
Communication matrix M_β
Vector elements that necessitate communication via **expand** operations

Figure 3.6: Formation of the communication matrix for the third column of the processor mesh ($\beta = 3$) to summarize expand operations in the pre-communication stage.

not included in M_β; these elements should be assigned to $P_{3,3}$ and $P_{2,3}$, respectively, to avoid unnecessary communication.

The fold communication tasks in checkerboard model are bound to distinct rows of the processor mesh. Following a similar approach, we form P distinct communication matrices to summarize the communication requirements of fold tasks in the post-communication stage. Let w_{C_α} denote the vector elements that necessitate communication in row α of the processor mesh, for $1 \leq \alpha \leq P$. At most Q processors can participate in communicating w_{C_α}, confined to the set of processors in $P_{\alpha,*}$. We summarize the communication operations in row α of the mesh with the $|w_{C_\alpha}| \times Q$ communication matrix M_α. Rows of M_α correspond to fold tasks on w_{C_α} and columns of M_α correspond to processors in $P_{\alpha,*}$. There exists a nonzero $m_{i\beta} \in M_\alpha$ if and only if there is a non-empty row segment in submatrix $A_{\alpha,\beta}$ at the respective row. The nonzeros in row i of M_α represent the set of processors that participate in communicating $w_{C_\alpha}[i]$, which is a subset of processors in $P_{\alpha,*}$. The nonzeros in column β of M_α represent the fold tasks processor $P_{\alpha,\beta}$ participates in. Again, internal rows are not included in M_α since they do not incur communication. An example is presented in Figure 3.7 to illustrate formation of communication matrix M_α in the third row of the processor mesh ($\alpha = 3$) to summarize the fold tasks. There are four output vector elements that necessitate communication (denoted by w_{C_α}) and they form rows of M_α. For instance, the first row of M_α has nonzeros corresponding to processors $P_{3,1}$, $P_{3,3}$ and $P_{3,4}$ since in matrix A, there exist nonzero row segments in respective submatrix blocks. Two vector elements–second and fifth–corresponding to internal rows do not incur communication and they are not included in M_α; these elements should be assigned to $P_{3,2}$ and $P_{3,3}$, respectively, to avoid unnecessary communication.

We form a total of $P+Q$ communication matrices to summarize communication requirements of checkerboard partitioning. We can address the communication requirements of both pre- and post-communication stages independently since communication operations in these stages are bound to distinct columns and rows of the processor mesh, respectively. Formation of these communication matrices is illustrated in Figure 3.8.



$P_{3,1}$
$P_{3,2}$
$P_{3,3}$
$P_{3,4}$
$\mathcal{R}_\alpha$
Matrix A
$(P \times Q = 4 \times 4)$
Output vector w
These do not require communication (not included in the communication matrix)
Form communication matrix for row $\alpha = 3$
w_{C_α}
Vector elements that necessitate communication via **fold** operations
Processors ($P_{\alpha,*}$) involved in communicating w_{C_α}
Communication matrix M_α

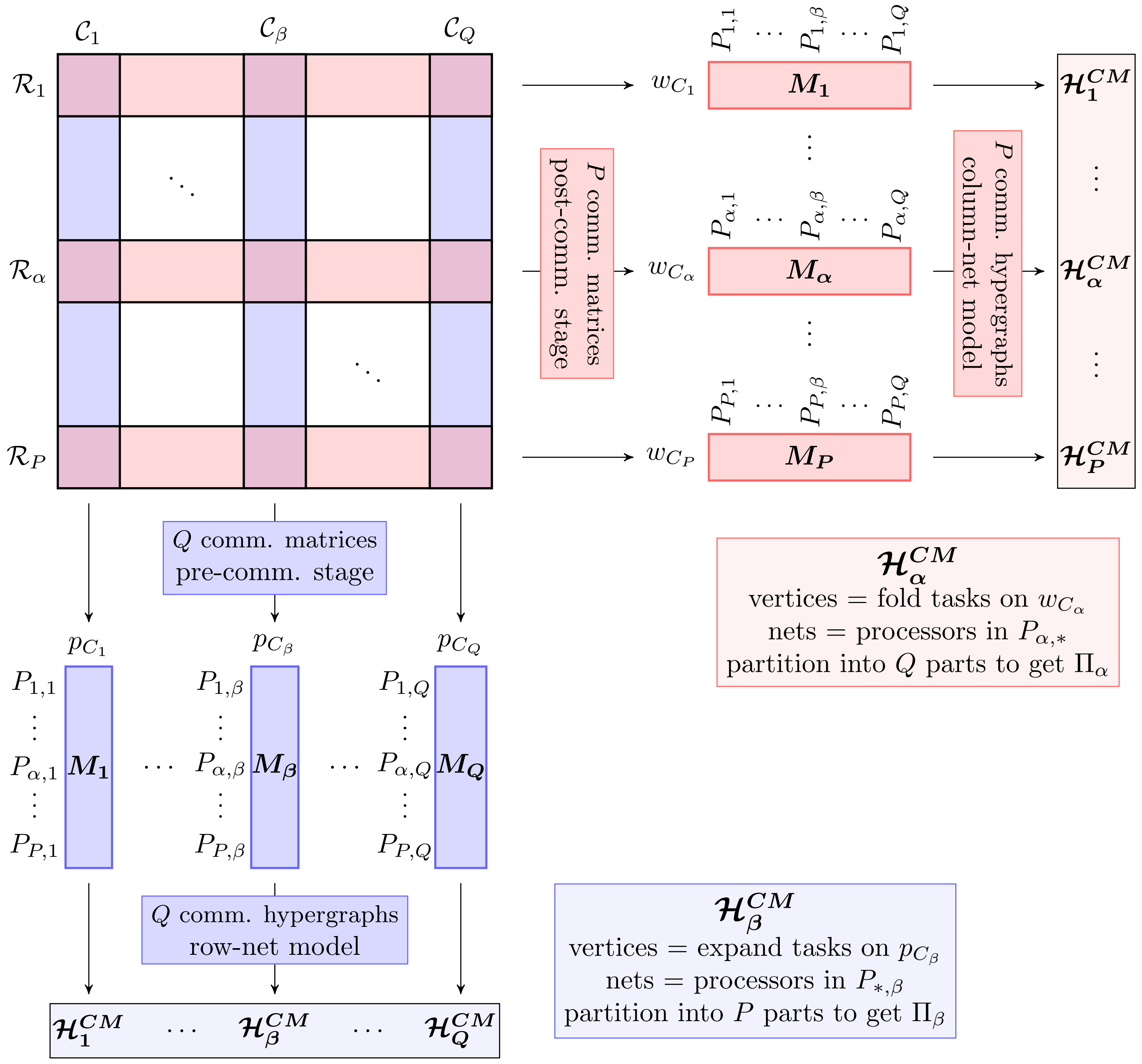
Figure 3.7: Formation of the communication matrix for the third row of the processor mesh ($\alpha = 3$) to summarize fold operations in the post-communication stage.

3.4.1.2 Formation of communication hypergraphs

We form Q hypergraphs from Q communication matrices for the pre-communication stage. For each M_β, a communication hypergraph $\mathcal{H}_\beta^{CM}$ is formed using the row-net hypergraph model, for $1 \leq \beta \leq Q$. The net set of $\mathcal{H}_\beta^{CM}$ represents the processors in column β ($P_{*,\beta}$) of the processor mesh and the vertex set of $\mathcal{H}_\beta^{CM}$ represents the expand tasks on p_{C_β}. Hence, there are P nets and $|p_{C_\beta}|$ vertices in $\mathcal{H}_\beta^{CM}$. A net $n_{\alpha,\beta}$ in $\mathcal{H}_\beta^{CM}$ represents the set of vertices that corresponds to expand communication tasks processor $P_{\alpha,\beta}$ takes part in. A vertex v_j in $\mathcal{H}_\beta^{CM}$ is connected by the set of nets corresponding to processors that communicate the respective vector element $p_{C_\beta}[j]$. In all Q communication hypergraphs, total number of vertices is equal to $|p_C| = \sum_{\beta=1}^{Q} |p_{C_\beta}|$ and total number of nets is equal to $Q \times P = K$.

In a similar manner, we form P hypergraphs from P communication matrices for the post-communication stage. For each M_α, a communication hypergraph $\mathcal{H}_\alpha^{CM}$ is formed using the column-net hypergraph model, for $1 \leq \alpha \leq P$. The net set of $\mathcal{H}_\alpha^{CM}$ represents the processors in row α ($P_{\alpha,*}$) of the processor mesh and the vertex set of $\mathcal{H}_\alpha^{CM}$ represents the fold tasks on w_{C_α}. Hence, there are Q nets and $|w_{C_\alpha}|$ vertices in $\mathcal{H}_\alpha^{CM}$. A vertex v_i in $\mathcal{H}_\alpha^{CM}$ is connected by the set of nets corresponding to the processors that communicate the respective vector element $w_{C_\alpha}[i]$. In all P communication hypergraphs, the total number of vertices is equal to $|w_C| = \sum_{\alpha=1}^{P} |w_{C_\alpha}|$ and the total number of nets is equal to $P \times Q = K$.

In total, we form $P+Q$ communication hypergraphs from $P+Q$ communication matrices. This process is illustrated in Figure 3.8.



$\mathcal{C}_1$
$\mathcal{C}_\beta$
$\mathcal{C}_Q$
$\mathcal{R}_1$
$\mathcal{R}_\alpha$
$\mathcal{R}_P$
$P_{1,1}$ … $P_{1,\beta}$ … $P_{1,Q}$
w_{C_1}
$\boldsymbol{M_1}$
$P_{\alpha,1}$ … $P_{\alpha,\beta}$ … $P_{\alpha,Q}$
w_{C_α}
$\boldsymbol{M_\alpha}$
$P_{P,1}$ … $P_{P,\beta}$ … $P_{P,Q}$
w_{C_P}
$\boldsymbol{M_P}$
P comm. matrices
post-comm. stage
P comm. hypergraphs
column-net model
$\boldsymbol{\mathcal{H}_1^{CM}}$
$\boldsymbol{\mathcal{H}_\alpha^{CM}}$
$\boldsymbol{\mathcal{H}_P^{CM}}$
Q comm. matrices
pre-comm. stage
p_{C_1}
p_{C_β}
p_{C_Q}
$P_{1,1}$
$P_{\alpha,1}$
$P_{P,1}$
$\boldsymbol{M_1}$
$P_{1,\beta}$
$P_{\alpha,\beta}$
$P_{P,\beta}$
$\boldsymbol{M_\beta}$
$P_{1,Q}$
$P_{\alpha,Q}$
$P_{P,Q}$
$\boldsymbol{M_Q}$
Q comm. hypergraphs
row-net model
$\boldsymbol{\mathcal{H}_1^{CM}}$ … $\boldsymbol{\mathcal{H}_\beta^{CM}}$ … $\boldsymbol{\mathcal{H}_Q^{CM}}$
$\boldsymbol{\mathcal{H}_\alpha^{CM}}$
vertices = fold tasks on w_{C_α}
nets = processors in $P_{\alpha,*}$
partition into Q parts to get Π_α
$\boldsymbol{\mathcal{H}_\beta^{CM}}$
vertices = expand tasks on p_{C_β}
nets = processors in $P_{*,\beta}$
partition into P parts to get Π_β

Figure 3.8: Minimizing latency cost in checkerboard partitioning model.

3.4.1.3 Partitioning of the communication hypergraphs

We partition $\mathcal{H}_\beta^{CM}$ to get a P-way partition $\Pi_\beta = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_P\}$ and obtain a distribution of expand tasks among P processors in column β of the processor mesh for the pre-communication stage, for $1 \leq \beta \leq Q$. The responsibility of the expand tasks represented by the vertices in $\mathcal{V}_\alpha \in \Pi_\beta$ is assigned to processor $P_{\alpha,\beta}$. Consider a net $n_{\alpha,\beta}$ in $\mathcal{H}_\beta^{CM}$ that represents $P_{\alpha,\beta}$. The connectivity set of this net contains the parts that correspond to the processors each of which send a message to $P_{\alpha,\beta}$. The size of this set can be at most P since $\mathcal{H}_\beta^{CM}$ is partitioned into P, bounding the number of messages sent/received by a single processor by $P-1$ in the pre-communication stage. Hence, this feature of original checkerboard partitioning is still respected. In partitioning $\mathcal{H}_\beta^{CM}$, the partitioning objective of minimizing cutsize corresponds to *minimizing the number of messages* communicated in column β of the processor mesh in the pre-communication stage, and the partitioning constraint of maintaining balance among part weights corresponds to obtaining a balance on the communication volume loads of these processors.

Similarly, we partition $\mathcal{H}_\alpha^{CM}$ to get a Q-way partition $\Pi_\alpha = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_Q\}$ and obtain a distribution of fold tasks among Q processors in row α of the processor mesh for the post-communication stage, for $1 \leq \alpha \leq P$. The responsibility of the fold tasks represented by the vertices in $\mathcal{V}_\beta \in \Pi_\alpha$ is assigned to processor $P_{\alpha,\beta}$. Consider a net $n_{\alpha,\beta}$ in $\mathcal{H}_\alpha^{CM}$ that represents $P_{\alpha,\beta}$. The connectivity set of this net contains the parts that correspond to the processors each of which receive a message from $P_{\alpha,\beta}$. The size of this set can be at most Q since $\mathcal{H}_\alpha^{CM}$ is partitioned into Q, bounding the number of messages sent/received by a single processor by $Q-1$ in the post-communication stage. Hence, this feature of original checkerboard partitioning is also respected. In partitioning $\mathcal{H}_\alpha^{CM}$, the partitioning objective of minimizing cutsize corresponds to *minimizing number of messages* communicated in row α of the processor mesh in the post-communication stage, and the partitioning constraint of maintaining balance among part weights corresponds to obtaining a balance on the communication volume loads of these processors.

The formed $P+Q$ hypergraphs can be independently partitioned since they do not depend on each other in any way. The maximum number of messages handled by a single processor is still $P+Q-2$ as in the original checkerboard partitioning. As a result, we improve latency costs in each row/column of the processor mesh independently while respecting basic characteristics of checkerboard partitioning.

3.4.2 Jagged partitioning

Jagged partitioning consists of two phases. The first phase consists of a single 1D partitioning model, whereas the second phase consists of multiple, independent and same type of 1D partitioning models. The second phase depends on the first phase by using the partitioning information obtained in the first phase to determine vertex sets and vertex weights for the models formed in the second phase. For the rest of the section, we assume a 1D rowwise partition in the first phase and a 1D columnwise partition in the second phase. For the arguments made in this section, an analogous discussion holds for the dual scheme as well.

Assume a $K = P \times Q$ processor mesh and an $n \times n$ square matrix A. The first phase of jagged partitioning is exactly the same as the first phase of checkerboard partitioning: a column-net hypergraph model $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ is used to obtain a P-way partition $\Pi_{\mathcal{R}} = \{\mathcal{V}_1, \ldots, \mathcal{V}_P\}$, which induces a rowwise partition $\{\mathcal{R}_1, \ldots, \mathcal{R}_P\}$ of A, where $\mathcal{R}_\alpha$ denotes the set of rows corresponding to vertices in $\mathcal{V}_\alpha$. At the end of this phase, the rows in row stripe A_α are associated with the Q processors in row α of the processor mesh, $P_{\alpha,*}$.

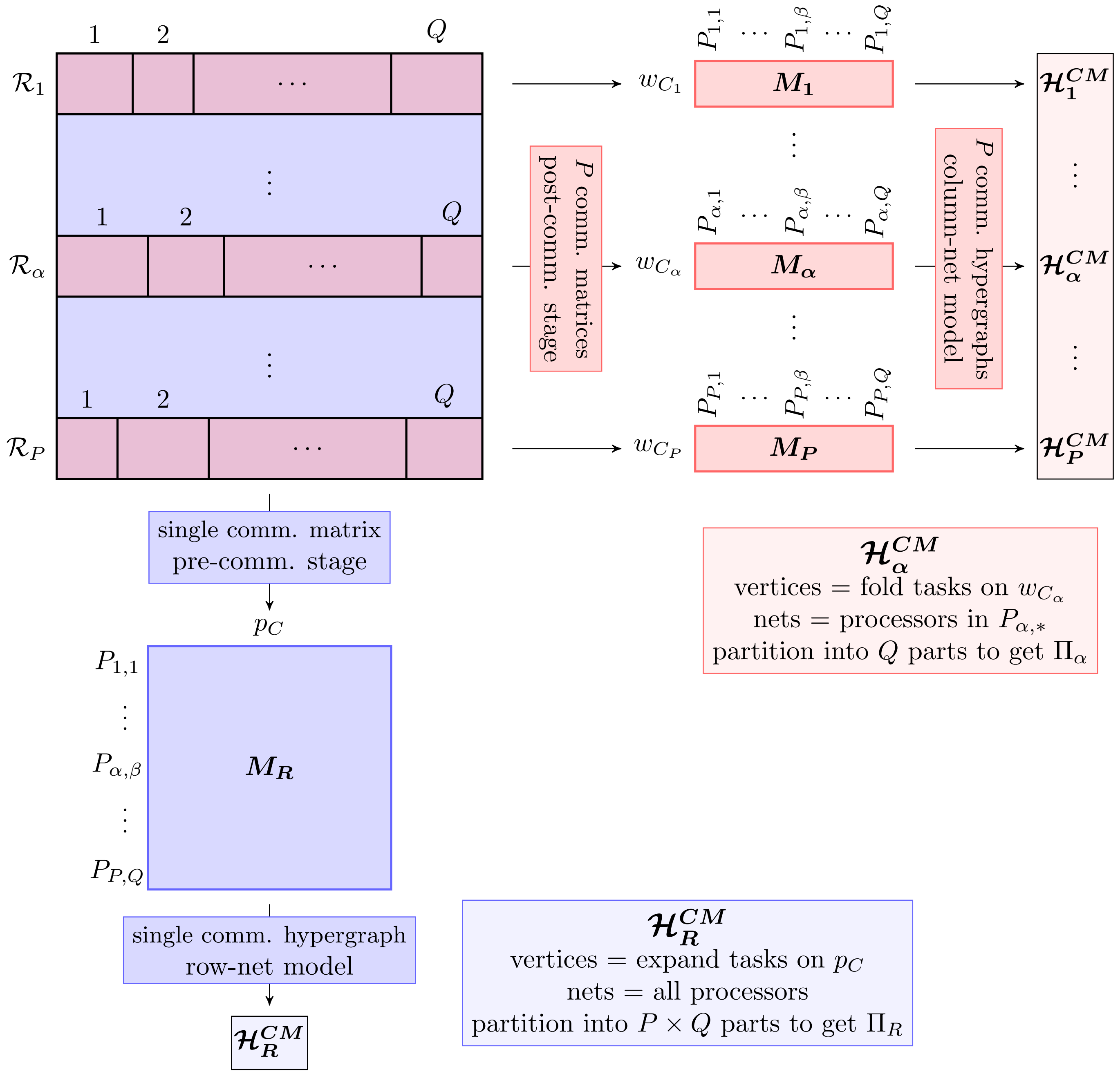
In the second phase, we form a hypergraph $\mathcal{H}_\alpha$ for each row submatrix A_α obtained in the former phase using the row-net hypergraph model, for $1 \leq \alpha \leq P$. In total, P hypergraphs are formed. In this aspect, jagged partitioning differs from checkerboard partitioning – which forms a single hypergraph in the second phase. The net set of $\mathcal{H}_\alpha$ represents rows of A_α and the vertex set of $\mathcal{H}_\alpha$ represents columns of A that have a nonzero column segment in A_α. Hence, the same vertex can appear in multiple hypergraphs since the corresponding column may have nonzero column segments in more than one row stripes. These P hypergraphs are independently partitioned into Q parts to obtain a Q-way partition of each row stripe. At the end of the second phase, for each row stripe A_α, we obtain Q submatrix blocks $A_{\alpha,1}, \ldots, A_{\alpha,Q}$ by partitioning vertices corresponding to columns of A_α.

The first and the second phase aim to minimize the volume of communication in pre- and post-communication stages, respectively, while maintaining computational load balance [57, 59]. In contrast to checkerboard partitioning, the objective in the second phase of the jagged partitioning is addressed independently by partitioning row stripes separately in distinct hypergraphs. The jagged model also differs from the checkerboard model as it does not lead to a Cartesian distribution of the matrix. Jagged partitioning is more flexible in this sense since it allows nonzeros of a column to be distributed among *any* processor that is in distinct rows of the processor mesh – not just among the processors that are in the *same* column of the mesh, as is the case for checkerboard partitioning. Hence, the column coherency is not preserved in the assignment of columns. This leads to improved communication volume for expand tasks in the pre-communication stage at the expense of increasing the upper bound on the number of messages handled by a single processor. In other words, the jagged model sacrifices the coarse level coherency of columns and causes the number of messages handled by a single processor in the pre-communication stage to be at most $P \times Q - Q = K - Q$. In this stage, a processor may communicate with any other processor except the processors that are in the same row of the processor mesh as this processor. On the other hand, the coherency of rows owned by a processor is respected in a coarse level as in checkerboard partitioning. This is because nonzeros of these rows are distributed among processors in the same row of processor mesh, $P_{\alpha,*}$, if the respective processor is in row α. Hence, the number of messages handled by a single processor for fold tasks in the post-communication stage is bounded by $Q - 1$ as there are Q processors in a single row of the processor mesh. Consequently, the maximum number of messages handled by a processor can be at most $(K - Q) + (Q - 1) = K - 1$ in jagged partitioning.

3.4.2.1 Communication matrices

The expand communication tasks in jagged model are not bound to distinct columns of the processor mesh. For this reason, we form a single communication matrix to summarize the communication requirements of expand tasks in the pre-communication stage. Let p_C denote the vector elements that necessitate communication. We summarize the communication operations with the $(K = P{\times}Q){\times}|p_C|$ communication matrix M_R. Rows of M_R correspond to *all* processors and columns of M_R correspond to expand tasks on p_C. Consider two vector elements owned by the same processor. Although these two elements can be communicated by at most P processors (each of which belongs a distinct row of the processor mesh), they do not necessarily need to be confined to the same column of the processor mesh. For this reason, we include all processors in M_R since a processor may be communicating with any other processor, except the processors that are in the same row of the processor mesh with this processor. The formation of M_R essentially resembles to that of 1D row-parallel $w = Ap$ (Section 3.3.5.1): $m_{kj} \neq 0$ if and only if column c_j has a nonzero column segment in k^{th} row stripe of A. Nonzeros in column $c_j \in M_R$ signify the set of processors that participate in communicating $p_C[j]$ and nonzeros in row $r_k \in M_R$ signify the expand tasks P_k participates in. The difference is, however, as a consequence of jagged partitioning, each column in M_R can have at most P nonzeros instead of K. As usual, vector elements corresponding to internal columns are not included in M_R since they do not necessitate communication.

In contrast to expand tasks, the fold communication tasks are bound to distinct rows of the processor mesh. The communication requirements of fold tasks in the post-communication stage are thus summarized by P distinct communication matrices. The formation of these matrices are the same as the formation of matrices for summarizing communication requirements of fold tasks in checkerboard partitioning. If we let w_{C_α} denote the vector elements that necessitate communication in row α of the processor mesh, then, at most Q processors participate in communicating w_{C_α}. Hence, communication operations in row α of the processor mesh are summarized with the $|w_{C_\alpha}| \times Q$ communication matrix M_α, for $1 \leq \alpha \leq Q$, where rows of M_α correspond to communication tasks on w_{C_α} and columns of M_α correspond to processors in $P_{\alpha,*}$. The semantics of nonzeros in rows and columns of M_α are identical to those of M_α in checkerboard partitioning.



$\mathcal{R}_1$
$\mathcal{R}_\alpha$
$\mathcal{R}_P$
1
2
Q
w_{C_1}
w_{C_α}
w_{C_P}
$P_{1,1}$ $\cdots$ $P_{1,\beta}$ $\cdots$ $P_{1,Q}$
$P_{\alpha,1}$ $\cdots$ $P_{\alpha,\beta}$ $\cdots$ $P_{\alpha,Q}$
$P_{P,1}$ $\cdots$ $P_{P,\beta}$ $\cdots$ $P_{P,Q}$
$\boldsymbol{M_1}$
$\boldsymbol{M_\alpha}$
$\boldsymbol{M_P}$
P comm. matrices post-comm. stage
P comm. hypergraphs column-net model
$\boldsymbol{\mathcal{H}_1^{CM}}$
$\boldsymbol{\mathcal{H}_\alpha^{CM}}$
$\boldsymbol{\mathcal{H}_P^{CM}}$
$\boldsymbol{\mathcal{H}_\alpha^{CM}}$
vertices = fold tasks on w_{C_α}
nets = processors in $P_{\alpha,*}$
partition into Q parts to get Π_α
single comm. matrix pre-comm. stage
p_C
$P_{1,1}$
$P_{\alpha,\beta}$
$P_{P,Q}$
$\boldsymbol{M_R}$
single comm. hypergraph row-net model
$\boldsymbol{\mathcal{H}_R^{CM}}$
$\boldsymbol{\mathcal{H}_R^{CM}}$
vertices = expand tasks on p_C
nets = all processors
partition into $P \times Q$ parts to get Π_R

Figure 3.9: Minimizing latency cost in jagged partitioning model.

We form a total of $P + 1$ communication matrices to summarize communication requirements of jagged partitioning. We can address the communication requirements of the post-communication stage independently using P matrices.

However, since communication operations in the pre-communication stage are not bound to the processors in the same column of the processor mesh, the expand tasks are represented in a single matrix with all K processors. Formation of these communication matrices is illustrated in Figure 3.9.

3.4.2.2 Formation of the communication hypergraphs

For the pre-communication stage, we form a single communication hypergraph $\mathcal{H}_R^{CM}$ from communication matrix M_R using the row-net hypergraph model. The net set of $\mathcal{H}_R^{CM}$ corresponds to K processors and the vertex set of $\mathcal{H}_R^{CM}$ corresponds to expand tasks on p_C. Hence, there are K nets and $|p_C|$ vertices in $\mathcal{H}_R^{CM}$. A vertex v_j in $\mathcal{H}_R^{CM}$ is connected by the set of nets corresponding to processors that communicate the respective vector element $p_C[j]$. Note that v_j can be connected by at most P nets, i.e., $d_j \leq P$.

For the post-communication stage, we form P communication hypergraphs from P communication matrices. The formation of these communication hypergraphs is actually the same as for checkerboard partitioning. A communication hypergraph $\mathcal{H}_\alpha^{CM}$ is formed using the column-net hypergraph model for matrix M_α, for $1 \leq \alpha \leq P$. The semantics of these hypergraphs are also the same: the net set of $\mathcal{H}_\alpha^{CM}$ represents the processors in row α of the processor mesh, $P_{\alpha,*}$, and the vertex set of $\mathcal{H}_\alpha^{CM}$ represents the fold tasks on w_{C_α}. Similarly, there are $|w_C|$ vertices and K nets in all communication hypergraphs.

In total, we form $P+1$ communication hypergraphs from $P+1$ communication matrices. This process is illustrated in Figure 3.9.

3.4.2.3 Partitioning of the communication hypergraphs

Communication hypergraph $\mathcal{H}_R^{CM}$ is partitioned to obtain a K-way partition $\Pi_R = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ to induce a distribution of expand tasks in the pre-communication stage. The responsibility of the expand tasks represented by the vertices in $\mathcal{V}_k$ is assigned to processor P_k. Consider a net n_k in $\mathcal{H}_R^{CM}$ that represents P_k. The connectivity set of this net corresponds to processors each of which send a message to P_k. Since $\mathcal{H}_R^{CM}$ is partitioned into K parts, the size of this set can be at most K. Thus, the maximum number of messages sent/received by a single processor is $K-1$ in the pre-communication stage (note that it is $K-Q$ in the original jagged partitioning). In partitioning $\mathcal{H}_R^{CM}$, the partitioning objective of minimizing cutsize corresponds to *minimizing the number of messages* communicated in the pre-communication stage, and the partitioning constraint of maintaining balance among part weights corresponds to obtaining a balance on the communication volume loads of K processors.

Communication hypergraph $\mathcal{H}_\alpha^{CM}$ is partitioned to obtain a Q-way partition $\Pi_\alpha = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_Q\}$, for $1 \leq \alpha \leq P$, to induce a distribution of fold tasks in the post-communication stage among Q processors in row α of the processor mesh. The partitioning of these communication hypergraphs are similar to those of checkerboard partitioning: the responsibility of fold tasks represented by the vertices in $\mathcal{V}_\beta$ is assigned to processor $P_{\alpha,\beta}$, and since we partition each hypergraph into Q parts, number of messages sent/received by a single processor is bounded by $Q-1$ in the post-communication stage. The partitioning objective and the balancing constraint are also the same with those of checkerboard partitioning.

The formed $P+1$ hypergraphs can be partitioned independently, since they do not depend on each other in any way. Note that the maximum number of messages handled by a single processor is slightly increased from $K-1$ to $K+Q-2$, which is caused by the partitioning for distributing tasks in the pre-communication stage. However, the cases beyond this are expected to be rare as a good partitioning tool will avoid them. As a result, we improve latency costs while respecting most characteristics of the original jagged partitioning.

3.4.3 Fine-grain (nonzero-based) partitioning

We briefly review fundamental properties of fine-grain partitioning. This model is first proposed in [56] and its communication costs are improved in a later work using the CHG model [58]. Both of these models are included in our experiments.

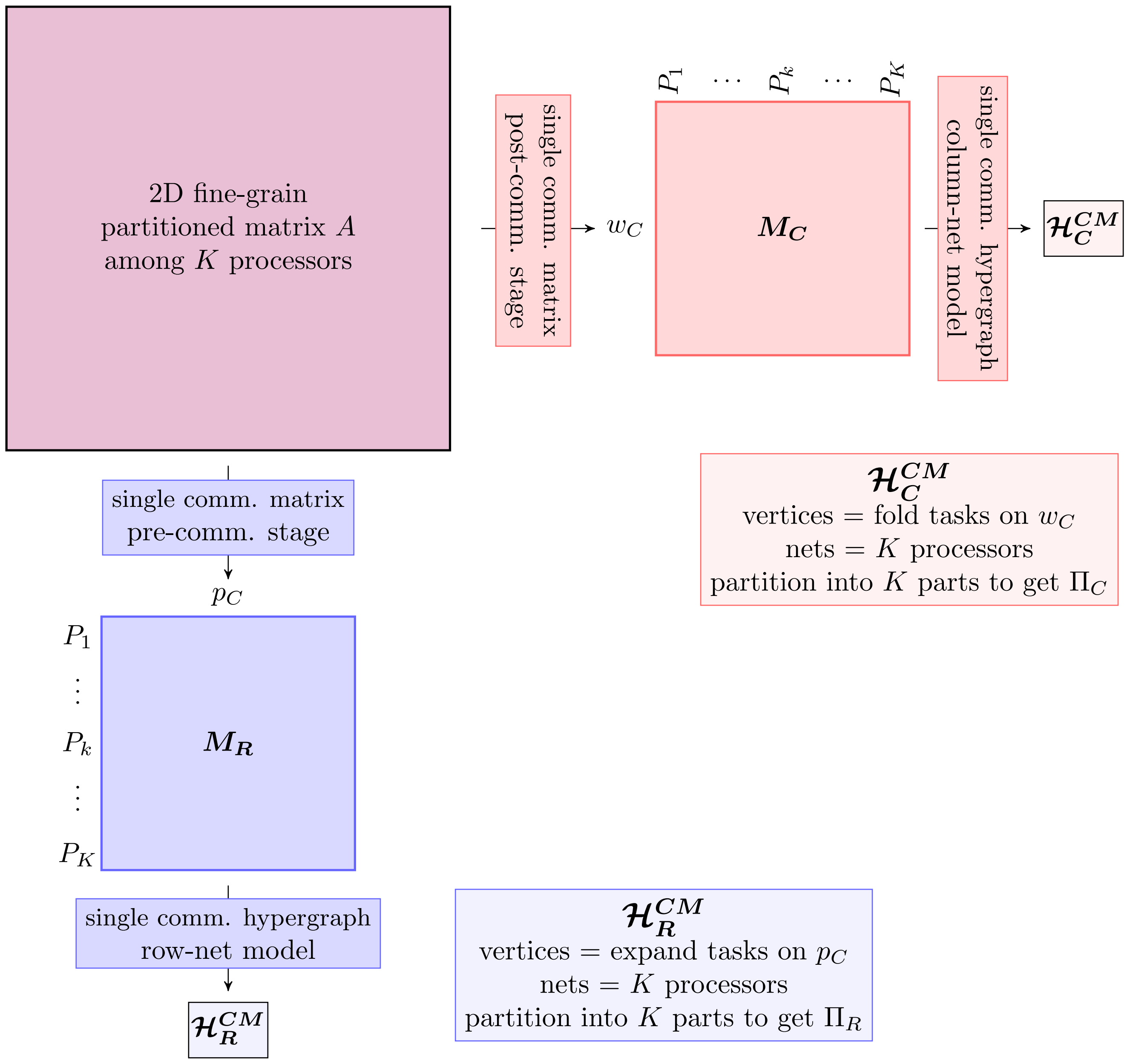

2D fine-grain partitioned matrix A among K processors
single comm. matrix post-comm. stage
w_C
P_1 … P_k … P_K
$\boldsymbol{M_C}$
single comm. hypergraph column-net model
$\boldsymbol{\mathcal{H}_C^{CM}}$
$\boldsymbol{\mathcal{H}_C^{CM}}$
vertices = fold tasks on w_C
nets = K processors
partition into K parts to get Π_C
single comm. matrix pre-comm. stage
p_C
P_1
P_k
P_K
$\boldsymbol{M_R}$
single comm. hypergraph row-net model
$\boldsymbol{\mathcal{H}_R^{CM}}$
$\boldsymbol{\mathcal{H}_R^{CM}}$
vertices = expand tasks on p_C
nets = K processors
partition into K parts to get Π_R

Figure 3.10: Minimizing latency cost in fine-grain partitioning model.

The fine-grain model forms a hypergraph in which vertices represent nonzeros of A and nets represent rows and columns of A. Nets corresponding to columns of A capture the communication volume incurred in the pre-communication stage, while nets corresponding to rows of A capture the communication volume incurred in the post-communication stage. Partitioning this hypergraph into K parts induces a distribution of nonzeros of the matrix among K processors. This leads to a completely arbitrary distribution of fine-grain computational tasks on a nonzero basis, where each vertex signifies a scalar multiplication with a single nonzero. Therefore, the fine-grain model respects neither row nor column coherence. In this aspect, it accommodates the highest level of flexibility by not restraining the computational tasks to coarser levels (i.e., nonzeros of a row and/or column) compared to checkerboard, jagged and 1D models. As a result, a processor may communicate with any other processor. Thus, the maximum number of messages handled by a single processor is $K-1$ in the pre-communication stage and is also $K-1$ in the post-communication stage, summing up to a total of $2(K-1)$ messages. The fine-grain model correctly minimizes the total volume of communication volume while maintaining computational load balance. Fore more details, see [56, 59].

The approach to improve communication requirements of the fine-grain model [58] consists of forming two communication matrices: one matrix for summarizing communication operations in the pre-communication stage and one matrix for summarizing communication operations in the post-communication stage. Compare this to the formation of communication matrices in the checkerboard and jagged models. We address the communication requirements in the checkerboard model by separately forming a total of $P+Q$ communication matrices since they are confined to distinct columns and rows of the processor mesh. Also in the jagged model, we form P communication matrices for the post-communication stage. These are not valid for the fine-grain model since the resulting partition is arbitrarily defined on the nonzeros of the matrix and any of the K processors may communicate with any other processor in both pre- and post-communication stages. For this reason, the whole set of processors and all vector elements that necessitate communication are included in two communication matrices. The process for reducing the communication costs of the fine-grain model is illustrated in Figure 3.10.

3.5 Comparison of partitioning models

We compare the basic properties of the investigated partitioning models to aid the discussions of the results in experiments. It is assumed that $P = Q = \sqrt{K}$ in $P \times Q$ processor mesh for the ease of presentation.

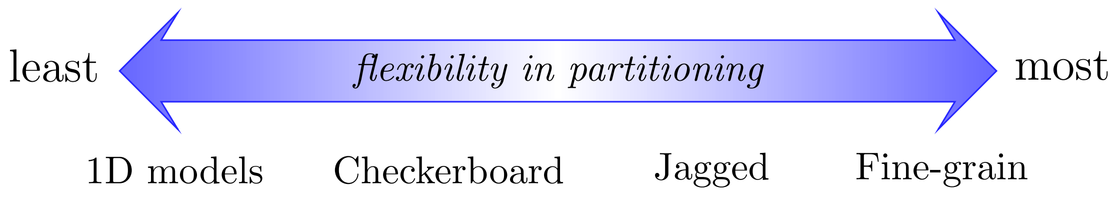


Figure 3.11: Comparison of models in terms of partitioning flexibility.

Figure 3.11 compares the partitioning models in terms of flexibility they provide during partitioning. 1D models lie at the left extreme of the spectrum since they represent each row/column of the coefficient matrix with a single distinct vertex as an atomic task. This leads to the assignment of all nonzeros of a row/column to an individual processor as a whole. Hence, 1D models respect row/column coherency at the individual processor level. The fine-grain model lies at the right extreme of the spectrum since in this model each vertex represents an atomic task corresponding to a single nonzero of the matrix. This is the most flexible and the finest level of partitioning granularity available, where neither row nor column coherency is preserved. So, in theory, nonzeros of a row/column can be distributed among K processors. Between these two extremes, the checkerboard and jagged models strive to distribute nonzeros of a row/column among a subset of processors. By doing so, they obtain a coarse-level row/column coherency at the processor mesh's row/column level. The checkerboard model leverages a coarse-level coherency in both partitioning phases whereas the jagged model leverages it in a single partitioning phase.

Among these partitioning models, 2D models are expected to achieve lower bandwidth costs compared to 1D models since they offer more flexibility in optimizing the objective of minimizing total communication volume. Among 2D models, fine-grain is expected to obtain the best results in terms of bandwidth costs, whereas checkerboard is likely to obtain the worst. The metrics related

Table 3.1: Comparison of partitioning models in terms of latency overhead and partitioning granularity.

number of messages	comm. stage	1D	2D partitioning models		
			Checkerboard	Jagged	Fine-grain
max	pre	$K-1$	$\sqrt{K}-1$	$K-\sqrt{K}$	$K-1$
	post	N/A	$\sqrt{K}-1$	$\sqrt{K}-1$	$K-1$
	overall	$K-1$	$2(\sqrt{K}-1)$	$K-1$	$2(K-1)$
total	**overall**	$K(K-1)$	$2K(\sqrt{K}-1)$	$K(K-1)$	$2K(K-1)$
Row/column coherency (Partitioning granularity)		Either entire row or Entire column	Coarse-level on both Rows and columns	Coarse-level on either Rows or columns	None

to latency costs (as upper bounds on the maximum number of messages) are presented in Table 3.1. Checkerboard has the lowest overhead with $2(\sqrt{K}-1)$ maximum messages per processor, whereas fine-grain has the highest overhead with $2(K-1)$. Although 1D and jagged models have the same upper bound $K-1$, in practice jagged partitioning is more likely to achieve better results in this metric since it restricts the number of messages in both stages of communication.

The discussions made so far in this section reflect the characteristics of the original partitioning models in which the communication hypergraph model is not used. The original models completely focus on minimizing bandwidth costs, disregarding latency-related objectives. Using the communication hypergraph model as a further step reduces latency costs at the expense of increasing bandwidth costs while respecting certain characteristics of the original models as much as possible. Our experimental evaluation shows that latency should definitely be on the table to achieve scalable performance.

3.6 Experiments

We evaluate two 1D-based and six 2D-based models, that is, a total of eight partitioning models. The evaluated models are based on 1D rowwise partitioning (`1D`), checkerboard partitioning (`CKBD`), jagged partitioning (`JGD`) and fine-grain partitioning (`FG`). Four of the evaluated models are the baseline models in which the communication tasks are assigned to processors using a simple heuristic that aims at balancing the communication volume loads while respecting total volume attained in the initial partitioning. This heuristic is also utilized in [63] and is an adaptation of the best-fit-decreasing heuristic used in solving the NP-hard K-feasible Bin Packing (`BP`) problem [78]. These `BP`-enhanced baseline models are referred to as `1D+BP`, `CKBD+BP`, `JGD+BP` and `FG+BP`. These four baseline models aim to reduce two important volume-related communication cost metrics total volume and maximum volume. The remaining four evaluated models are the `CHG`-enhanced versions in which the communication tasks are assigned to processors using communication hypergraph model. These `CHG`-enhanced models are referred to as `1D+CHG`, `CKBD+CHG`, `JGD+CHG` and `FG+CHG`. These aim to reduce total message count and maximum volume. Hence, we evaluate the merit of reducing a latency-related cost metric in partitioning. We use PaToH [42, 77] to partition the computational hypergraphs formed in the first phase of all models and the communication hypergraphs formed in the second phase of the `CHG`-enhanced models.

We implemented CGNE and CGNR solvers via the PETSc toolkit [69] and utilized the mentioned models for partitioning the coefficient matrix and vectors in these solvers. Since obtained runtime results for both solvers are similar, we only present speedup results corresponding to CGNR. Note that the metrics regarding partitioning models (Section 3.6.1) such as total volume, message count, etc. are the same for both solvers as they contain the same type of communication operations.

Table 3.2: Test matrices and their properties.

	Number of		Nonzeros				
				per row		per column	
Matrix	rows/cols	nonzeros	avg	min	max	min	max
venkat01	62,424	1,717,792	27.52	16	44	16	44
mc2depi	525,825	2,100,225	3.99	2	4	2	4
poisson3Db	85,623	2,374,949	27.74	6	145	6	145
thermomech_dK	204,316	2,846,228	13.93	7	20	7	20
stomach	213,360	3,021,648	14.16	7	19	6	22
FEM_3D_thermal2	147,900	3,489,300	23.59	12	27	12	27
laminar_duct3D	67,173	3,833,077	57.06	1	89	3	89
xenon2	157,464	3,866,688	24.56	1	27	1	27
iChem_Jacobian	274,087	4,137,369	15.10	5	17	5	17
torso3	259,156	4,429,042	17.09	7	22	6	21
tmt_unsym	917,825	4,584,801	5.00	3	5	3	5
t2em	921,632	4,590,832	4.98	1	5	1	5
Hamrle3	1,447,360	5,514,242	3.81	2	6	2	9
largebasis	440,020	5,560,100	12.64	4	14	4	14
Chevron4	711,450	6,376,412	8.96	2	9	2	9
cage13	445,315	7,479,343	16.80	3	39	3	39
PR02R	161,070	8,185,136	50.82	1	92	5	88
atmosmodl	1,489,752	10,319,760	6.93	4	7	4	7
kim2	456,976	11,330,020	24.79	4	25	5	25
memchip	2,707,524	14,810,202	5.47	2	27	1	27
Freescale1	3,428,755	18,920,347	5.52	1	27	1	25
circuit5M_dc	3,523,317	19,194,193	5.45	1	27	1	25
fem_hifreq_circuit	491,100	20,239,237	41.21	12	110	12	110
rajat31	4,690,002	20,316,253	4.33	1	1252	1	1252
CoupCons3D	416,800	22,322,336	53.56	20	76	20	76
Transport	1,602,111	23,500,731	14.67	5	15	5	15
ML_Laplace	377,002	27,689,972	73.45	26	74	26	74
RM07R	381,689	37,464,962	98.16	1	295	1	245

All models are tested with 28 matrices chosen from the UFL matrix collection [79]. The properties of these matrices are presented in Table 3.2. The evaluated models are tested on a BlueGene/Q system with varying number of processors $K \in \{256, 512, 1024, 2048, 4096, 8192\}$. A node on this system consists of 16 cores (single PowerPC A2 processor) with 1.6 GHz clock frequency and 16 GB memory. The nodes are connected by a 5D torus chip-to-chip network. We only consider the case of strong scaling.

For the pre-communication stages of `CKBD+CHG` and `JGD+CHG`, we opted not to apply the communication hypergraph model since the partitioning corresponding to this stage leads to a number of very small communication hypergraphs in which the number of communication tasks (that is, vertices) and the number of messages per processor are very low. Hence, utilizing a dedicated tool to partition these hypergraphs often does not pay off. Instead, the simple aforementioned heuristic is able to obtain comparable partition qualities in a shorter amount of time. Hence, the pre-communication stages (expand tasks) of `CKBD+BP` and `CKBD+CHG` models, and `JGD+BP` and `JGD+CHG` models have the same quantities for the statistics presented in Section 3.6.1. However, for the post-communication stage, the respective quantities drastically differ in these models as the benefits of using the communication hypergraph model are more apparent.

3.6.1 Bandwidth and latency costs of partitioning models

Tables 3.3 and 3.4 display the metrics related to latency and bandwidth costs for the evaluated models. The metrics related to latency are highlighted under "Number of messages" columns and the metrics related to bandwidth are highlighted under "Communication volume" columns. The statistics for both total and maximum metrics are presented. The columns "Expand" and "Fold" in the table indicate the results obtained in the pre- and post-communication stages of 2D models, respectively. Recall that the 1D models (`1D+BP` and `1D+CHG`) have a single communication stage, which is the pre-communication stage in our case. The values are averaged over 20 test matrices separately for each K. The communication volume statistics are in terms of words. Note that the total/maximum number of messages and maximum volume of communication of `CKBD+BP` and

Table 3.3: Average communication requirements and speedups (Part 1: $K \in \{256, 512, 1024, 2048\}$).

		Number of messages						Communication volume						
		Total			Maximum			Total			Maximum			
K	Model	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum	Speedup
	1D+BP	2464	-	2464	17.9	-	17.9	164470	-	164470	728	-	**728**	143
	1D+CHG	1640	-	**1640**	13.5	-	13.5	245514	-	245514	1234	-	1234	148
	CKBD+BP	414	1728	2141	4.8	6.6	11.4	38476	145554	184029	367	759	1125	142
256	CKBD+CHG	414	1423	1836	4.8	5.8	**10.5**	38476	214657	253133	367	1176	1542	138
	JGD+BP	850	1266	2117	10.0	4.9	14.9	38476	112712	**151188**	311	599	910	**158**
	JGD+CHG	850	1131	1982	10.0	5.0	15.0	38476	165081	203557	311	959	1270	153
	FG+BP	1959	1751	3710	15.5	6.8	22.3	107993	48359	156352	546	275	821	150
	FG+CHG	1513	1257	2770	12.5	4.6	17.1	173449	74672	248121	847	377	1223	150
	1D+BP	5683	-	5683	21.6	-	21.6	226127	-	226127	513	-	**513**	236
	1D+CHG	3496	-	**3496**	15.5	-	15.5	327581	-	327581	829	-	829	231
	CKBD+BP	1019	3590	4608	6.4	6.8	13.1	57434	195114	252548	264	530	794	225
512	CKBD+CHG	1019	2885	3904	6.4	6.6	**13.0**	57434	280548	337981	264	804	1067	220
	JGD+BP	2088	2604	4692	11.7	5.6	17.3	57434	147488	**204922**	220	405	626	**247**
	JGD+CHG	2088	2267	4355	11.7	4.5	16.2	57434	212002	269436	220	609	829	238
	FG+BP	4261	3784	8045	17.7	8.0	25.7	144968	70492	215460	374	205	580	228
	FG+CHG	3165	2622	5787	13.8	4.8	18.6	228987	106423	335410	566	273	839	229
	1D+BP	13201	-	13201	27.0	-	27.0	314109	-	314109	359	-	**359**	294
	1D+CHG	7451	-	**7451**	16.9	-	16.9	441340	-	441340	568	-	568	343
	CKBD+BP	1587	9624	11211	5.9	9.1	15.0	57434	288909	346343	167	399	567	320
1024	CKBD+CHG	1587	6897	8484	5.9	7.1	**13.0**	57434	404550	461984	167	534	701	320
	JGD+BP	3571	6775	10346	11.7	7.4	19.1	57434	223665	**281099**	138	314	452	**354**
	JGD+CHG	3571	5266	8837	11.7	5.2	16.9	57434	314303	371737	138	430	568	341
	FG+BP	9286	8141	17427	20.6	8.5	29.1	192165	103683	295848	252	153	405	318
	FG+CHG	6537	5420	11957	15.1	6.1	21.3	297236	152370	449606	376	201	577	327
	1D+BP	30651	-	30651	30.9	-	30.9	437009	-	437009	256	-	**256**	355
	1D+CHG	16028	-	**16028**	19.0	-	19.0	591207	-	591207	389	-	389	450
	CKBD+BP	3885	21008	24892	7.3	9.4	16.7	82863	395862	478725	116	277	393	421
2048	CKBD+CHG	3885	14275	18159	7.3	8.5	**15.8**	82863	533056	615920	116	362	478	429
	JGD+BP	8301	14621	22921	14.1	7.4	21.5	82863	297989	**380852**	99	214	313	**468**
	JGD+CHG	8301	10833	19134	14.1	5.6	19.6	82863	404389	487252	99	284	383	457
	FG+BP	20050	17669	37719	22.3	10.3	32.6	251398	154339	405737	171	116	287	412
	FG+CHG	13494	11272	24766	16.3	8.2	24.5	379611	221040	600651	243	152	395	440

The bold values in five columns total/maximum number of messages (Sum), total/maximum communication volume (Sum) and Speedup indicate the best values obtained by the respective model at a specific K.

Table 3.4: Average communication requirements and speedups (Part 2: $K \in \{4096, 8192\}$).

		Number of messages						Communication volume						
		Total			Maximum			Total			Maximum			
K	Model	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum	Speedup
4096	1D+BP	71313	-	71313	37.0	-	37.0	615662	-	615662	185	-	**185**	372
	1D+CHG	36563	-	**36563**	21.8	-	21.8	810012	-	810012	267	-	267	530
	CKBD+BP	6122	55186	61308	6.7	13.5	20.3	82863	579257	662121	72	209	280	511
	CKBD+CHG	6122	32230	38352	6.7	8.7	**15.4**	82863	747663	830526	72	252	324	556
	JGD+BP	13440	38102	51543	12.9	10.4	23.2	82863	449614	**532477**	62	163	225	584
	JGD+CHG	13440	24711	38151	12.9	6.7	19.5	82863	584442	667306	62	203	264	**586**
	FG+BP	43084	38239	81323	24.8	11.0	35.7	326646	228666	555312	114	87	201	516
	FG+CHG	28683	24606	53289	18.0	8.8	26.7	486581	323623	810204	160	117	278	554
8192	1D+BP	160507	-	160507	42.9	-	42.9	871090	-	871090	137	-	**137**	392
	1D+CHG	81592	-	**81592**	26.0	-	26.0	1040719	-	1040719	175	-	175	560
	CKBD+BP	14589	125660	140249	7.9	15.7	23.6	115948	814437	930384	49	149	198	593
	CKBD+CHG	14589	70942	85531	7.9	10.7	**18.6**	115948	1003817	1119764	49	171	220	667
	JGD+BP	28838	85171	114009	13.9	11.4	25.2	115948	618294	**734242**	43	114	157	683
	JGD+CHG	28838	52882	81720	13.9	6.5	20.3	115948	769539	885486	43	137	180	**701**
	FG+BP	90483	81319	171801	26.3	13.3	39.6	430725	327017	757741	77	65	142	591
	FG+CHG	59692	52380	112072	19.8	10.1	29.9	572597	432085	1004682	99	84	183	663

The bold values in five columns total/maximum number of messages (Sum), total/maximum communication volume (Sum) and Speedup indicate the best values obtained by the respective model at a specific K.

CKBD+CHG, and JGD+BP and JGD+CHG are the same since they use the same heuristic to distribute expand tasks. Total communication volume of two successive K values (512 and 1024, 2048 and 4096, etc.) are the same for the models that are based on checkerboard and jagged partitioning since there exists same number of processor columns in the corresponding processor mesh. For instance, at $K = 512$ and $K = 1024$, the processor meshes are of sizes 32×16 and 32×32, respectively. We also present the average speedup values obtained by models to give an idea about the efficiency. A more detailed and accurate discussion with performance profiles and speedup curves can be found in the next section.

The major factors that determine overall latency and bandwidth costs are the maximum number of messages and the maximum volume of communication handled by a single processor, respectively. As seen in Tables 3.3 and 3.4, with increasing number of processors, the maximum number of messages increases sharply, whereas maximum volume decreases despite the increase in total volume. For instance, for `1D+BP`, when K increases from 256 to 8192 processors, the maximum number of messages increases from 17.9 to 42.9, whereas maximum volume decreases from 728 to 137 words, on average. Hence, the latency overhead on average increases by a factor of 2.4, whereas the bandwidth overhead on average decreases by a factor of 5.3. Moreover, the total message count increases more sharply compared to total volume: 65.1 times versus 5.3 times. These figures imply that with increasing number of processors, latency costs steadily become more important than bandwidth costs in determining overall communication cost of parallel SpMV operations. Hence, reducing latency costs should pay off with improved scalability, as will be seen in the following section. Observe that similar arguments hold for other partitioning models as well.

If we compare the partitioning models that do not use the communication hypergraph model among themselves (i.e., `1D+BP`, `CKBD+BP`, `JGD+BP` and `FG+BP`) in terms of total communication volume, we see from Tables 3.3 and 3.4 `JGD+BP` obtains the best results, whereas `CKBD+BP` obtains the worst results. `FG+BP` is expected to achieve the best results in this metric since it offers the highest flexibility by performing the partitioning on a nonzero basis – the finest granularity available. However, the reason why `JGD+BP` achieves slightly better results than `FG+BP` in this metric is related not to models themselves but to the shortcomings of recursive bisectioning used in partitioning. The shortcomings of recursive bipartitioning are well known for high partitioning values [80, 81]. For example at $K = 4096$, `FG+BP` directly partitions the input matrix into 4096 parts whereas `JGD+BP` first partitions it into 64 parts, and for each of these parts, it partitions them into 64 parts again to obtain a 4096-way partition. Hence, by using smaller partition values compared to `FG+BP`, `JGD+BP` is relatively able to mitigate the drawbacks of recursive bisection. The poor performance of `CKBD+BP` in this metric is due to the use of multi-constraint partitioning. This limits the search space drastically, where the higher the number of constraints, the harder it is to get good quality partitions as the search space narrows down with increasing number of constraints. However, this is a tradeoff for `CKBD+BP` as it often achieves good results in total message count, which are comparable to those of `JGD+BP` at lower processor counts. At higher processor counts, like 4096 and 8192, `JGD+BP` achieves better results in total message count. This is again because the high number of constraints at high values of K leads to poor total volume in `CKBD+BP`, which in turn affects the total message count as a side effect by causing an increase. As expected, the smallest maximum number of messages is obtained by `CKBD+BP` as it bounds the communication to specific rows and columns of the processor mesh in both stages of communication. `FG+BP` is often the worst in terms of total and maximum number of messages because it causes increases in these metrics in order to reduce total volume.

To aid the assessment of benefits of the communication hypergraph, we present Table 3.5. In this table, each latency-improved (`CHG`-enhanced) model's performance metrics are normalized with respect to those of their baselines. In other words, the results of `1D+CHG` are normalized with respect to those of `1D+BP`, the results of `CKBD+CHG` are normalized with respect to those of `CKBD+BP`, etc. The normalization is performed on a matrix basis and the averages of these normalized values over 28 matrices are given separately for each K. As seen from the table, `CHG`-enhanced models improve the total message count drastically, as minimizing this metric is one the main objectives in these models. For example at 2048 processors, `1D+CHG` achieves 36% improvement over `1D+BP`, `CKBD+CHG` achieves 23% improvement over `CKBD+BP`, `JGD+CHG` achieves 13% improvement over `JGD+BP`, and `FG+CHG` achieves 23% improvement over `FG+BP`. However, this comes at the cost of increased total volume. Again at 2048 processors, `1D+CHG` increases the total volume by 40%, `CKBD+CHG` by 33%, `JGD+CHG` by 30%, and `FG+CHG` by 49%. The crucial observation, however, is that the message count improvements of partitioning models that rely on the communication hypergraph model tend to increase with increasing number of processors. For example, `1D+CHG` achieves a 24% improvement over `1D+BP` at $K = 256$ in total message count and this improvement becomes 37% at $K = 8192$. This improvement increases from 11% to 33%, 2% to 24%, and 14% to 24% for `CKBD+CHG`, `JGD+CHG` and `FG+CHG`, respectively. This implies that the benefits obtained using the communication hypergraph model are more prominent at higher processor counts. Note that for `CKBD+CHG` and `JGD+CHG`, normalized total message average is closer to fold message average rather than expand message average, which is also the case for normalized average volume. This is because the message count and communication volume in the post-communication stage are much higher than the pre-communication stage in these models. This is also where the communication hypergraph model is expected to perform well.

Table 3.5: Comparison of partitioning models with communication hypergraphs normalized with respect to their baseline counterparts averaged over all matrices for each K.

		Number of messages						Communication volume					
		Total			Maximum			Total			Maximum		
K	Model	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum	Expand	Fold	Sum
256	`1D+CHG`	0.76	-	0.76	0.85	-	0.85	1.51	-	1.51	1.66	-	1.66
	`CKBD+CHG`	1.00	0.86	0.89	1.00	1.01	0.98	1.00	1.49	1.41	1.00	1.54	1.38
	`JGD+CHG`	1.00	0.95	0.98	1.00	1.21	1.09	1.00	1.45	1.35	1.00	1.53	1.36
	`FG+CHG`	0.87	0.85	0.86	0.96	0.81	0.90	1.56	1.51	1.56	1.53	1.26	1.47
512	`1D+CHG`	0.72	-	0.72	0.84	-	0.84	1.48	-	1.48	1.62	-	1.62
	`CKBD+CHG`	1.00	0.85	0.88	1.00	1.41	1.08	1.00	1.46	1.37	1.00	1.52	1.35
	`JGD+CHG`	1.00	0.92	0.96	1.00	0.97	0.97	1.00	1.42	1.32	1.00	1.50	1.33
	`FG+CHG`	0.85	0.84	0.84	0.94	0.80	0.88	1.55	1.50	1.54	1.52	1.23	1.44
1024	`1D+CHG`	0.67	-	0.67	0.77	-	0.77	1.44	-	1.44	1.61	-	1.61
	`CKBD+CHG`	1.00	0.78	0.81	1.00	1.03	0.97	1.00	1.44	1.38	1.00	1.37	1.27
	`JGD+CHG`	1.00	0.86	0.91	1.00	0.88	0.94	1.00	1.41	1.34	1.00	1.42	1.29
	`FG+CHG`	0.81	0.81	0.80	0.86	1.37	0.87	1.53	1.48	1.51	1.51	1.24	1.43
2048	`1D+CHG`	0.64	-	0.64	0.74	-	0.74	1.40	-	1.40	1.57	-	1.57
	`CKBD+CHG`	1.00	0.74	0.77	1.00	1.02	0.99	1.00	1.39	1.33	1.00	1.33	1.23
	`JGD+CHG`	1.00	0.82	0.87	1.00	0.94	0.95	1.00	1.37	1.30	1.00	1.37	1.25
	`FG+CHG`	0.77	0.77	0.77	0.82	0.96	0.84	1.51	1.45	1.49	1.46	1.32	1.41
4096	`1D+CHG`	0.65	-	0.65	0.68	-	0.68	1.39	-	1.39	1.54	-	1.54
	`CKBD+CHG`	1.00	0.66	0.69	1.00	0.77	0.84	1.00	1.35	1.31	1.00	1.27	1.19
	`JGD+CHG`	1.00	0.74	0.80	1.00	0.82	0.90	1.00	1.33	1.29	1.00	1.33	1.23
	`FG+CHG`	0.77	0.78	0.77	0.82	0.85	0.81	1.50	1.46	1.49	1.45	1.38	1.42
8192	`1D+CHG`	0.63	-	0.63	0.69	-	0.69	1.35	-	1.35	1.47	-	1.47
	`CKBD+CHG`	1.00	0.64	0.67	1.00	0.79	0.84	1.00	1.30	1.26	1.00	1.23	1.16
	`JGD+CHG`	1.00	0.69	0.76	1.00	0.70	0.85	1.00	1.29	1.25	1.00	1.30	1.21
	`FG+CHG`	0.76	0.76	0.76	0.82	0.75	0.78	1.47	1.42	1.45	1.44	1.40	1.42

The models that use the communication hypergraph model improve the maximum number of messages as well. This is a consequence of the reduction in total message count. In Tables 3.3 and 3.4, if we compare partitioning models in this metric, it can be seen that `CKBD+CHG` obtains the best results which is usually followed by `JGD+CHG`. For example, at 8192 processors on average, the maximum number of messages handled by a single processor for `CKBD+CHG` is only 18.6 and for `JGD+CHG` it is only 20.3. These values are followed by `CKBD+BP` with 23.6 and `JGD+BP` with 25.2. When we examine the other important metric the maximum volume in Table 3.5, it is seen that the models that rely on communication hypergraph model close the gap with their baseline counterparts with increasing K. For instance, when K increases from 256 to 8192 processors, the increase in maximum volume incurred by the use of the communication hypergraph model decreases from 66%, 38%, 36% and 47% to 47%, 16%, 21% and 42% in `1D+CHG`, `CKBD+CHG`, `JGD+CHG` and `FG+CHG`, respectively, compared to their baseline counterparts. This is an important benefit of the communication hypergraph model since it strives for balancing volume.

Table 3.6: Average partitioning times (sequential, in seconds).

	1D+BP	CKBD+BP	JGD+BP	FG+BP
	32.43	33.79	21.68	114.22
CHG	1.68	0.98	1.13	1.98

The sequential partitioning times of the evaluated models are given in Table 3.6 averaged over all matrices and K values. The CHG times (indicated via `+CHG` row include only the partitioning times of communication hypergraphs formed for the respective model. As expected, the fine-grain model has the highest partitioning time as the hypergraphs formed in this model are typically larger. The partitioning times of the communication hypergraphs are quite low compared to the respective original partitionings since they are small as they contain only the vertices that correspond to the vector elements that necessitate communication. Note that `CKBD+CHG`, `JGD+CHG` and `FG+CHG` form a number of communication hypergraphs that can independently be partitioned, hence the partitioning of them can easily be parallelized. A more healthy comparison of partitioning overhead for 2D models can be found in [59].

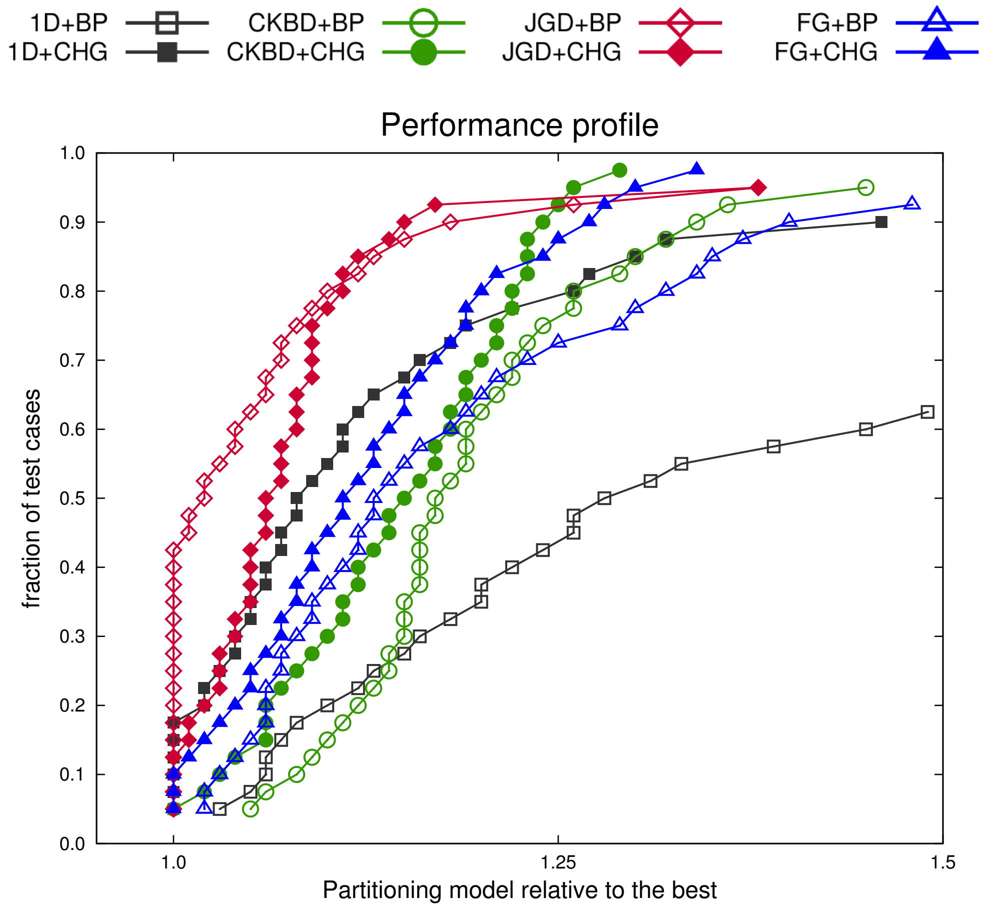


Figure 3.12: Performance profiles of eight partitioning models for all K.

3.6.2 Speedup analysis

For a detailed comparison of the partitioning models in terms of parallel solver running times/speedups, we present the performance profiles in Figures 3.12 and 3.13. Performance profiles provide a better understanding of the characteristics of the compared models as they capture the relative performance of the compared models more accurately [82]. A point x, y in a profile reads as the respective model is within the x factor of the best result in y fraction of the test instances. In other words, the closer the performance profile of a scheme to the y-axis, the better it is. A test instance in our case is the parallel solver running time obtained for a specific matrix and K. We compare the performances of partitioning models for all K values in Figure 3.12 and for $K \in 4096, 8192$

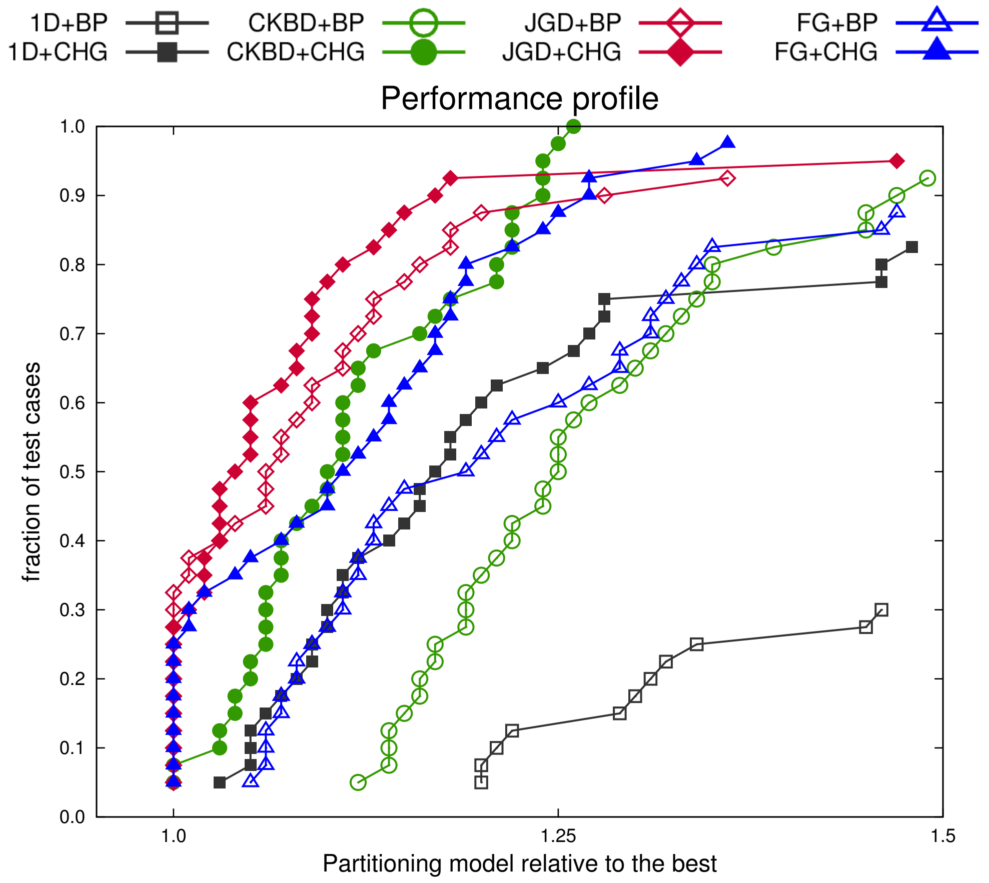


Figure 3.13: Performance profiles of eight partitioning models for $K \in \{4096, 8192\}$.

in Figure 3.13. The former contains 168 instances and the latter contains 56 instances.

When we compare the models considering all K values in Figure 3.12, `JGD+BP` is clearly the best performing model followed by `JGD+CHG`. `JGD+BP` obtains the best results for more than 40% of the test cases and exhibits very good performance for a very large fraction of the test cases. These two models are followed by two models that use communication hypergraph: `1D+CHG` and `FG+CHG`. Except jagged model, applying the communication hypergraph seems to improve performance of the partitioning models as `1D+CHG`, `CKBD+CHG` and `FG+CHG` perform better than `1D+BP`, `CKBD+BP` and `FG+BP`, respectively. `1D+BP` obtains the worst results, proving itself to be not a viable partitioning model compared to the 2D models as long as communication hypergraph is not used for it.

Figure 3.13 is presented to better assess the benefits of using the communication hypergraph model. As discussed, latency gets more important with increasing K and it is expected that the models using the communication hypergraph model should be performing better as K increases. If we consider the performances of parittioning models at only 4096 and 8192 processors, it can be seen from the figure that the models that use communication hypergraph improve the performance much more compared to the case when all K values are considered. In other words, for example, if we compare `CKBD+BP` and `CKBD+CHG` in Figure 3.12 and Figure 3.13 the performance difference between them increases in favor of `CKBD+CHG` in Figure 3.13. This can be observed for all parittioning models, i.e., by comparing `1D+BP` and `1D+CHG`, `CKBD+BP` and `CKBD+CHG`, `JGD+BP` and `JGD+CHG`, `FG+BP` and `FG+CHG` in Figure 3.12 and Figure 3.13. This is also validated as `JGD+CHG` can be said to be the best performing model in Figure 3.13 followed by `JGD+BP`. These two models are again followed by two models that use the communication hypergraph: `FG+CHG` and `CKBD+CHG`. These figures show that the communication hypergraph proves to a valuable method for achieving scalability.

We present the obtained speedup values of eight evaluated models in Figures 3.14 , 3.15 and 3.16. Among 28 matrices, we present the speedups of 27 matrices. The number of processors varies from 256 to 8192. The experiments are performed with the CGNR solver which is implemented via the PETSc toolkit.

As seen from the speedup curves, the models that adopt the communication hypergraph model often exhibit better scalability compared to their baseline counterparts. Moreover, the difference gets more prominent with increasing number of processors as latency becomes the determining factor for performance. When we compare `1D+BP` with `1D+CHG`, `1D+CHG` achieves superior scalability in all matrices. In 2D models, applying the communication hypergraph model usually improves scalability. For example, in matrices `circuit5M_dc`, `CoupCons3D`, `fem_hifreq_circuit`, `Freescale1`, `memchip`, `ML_Laplace`, `rajat31` and `RM07R`, the partitioning models `CKBD+CHG`, `JGD+CHG` and `FG+CHG` improve performance of their baseline counterparts CKBD+BP, JGD+BP and FG+BP, respectively. In the remaining matrices, latency-improved versions of 2D models either achieve close or slightly worse performance. This is mainly due to the fact that in these matrices, the latency costs obtained in the initial partitionings are already very low due to the characteristics of these matrices and further trying to improve them does not pay off since, at the other hand, the bandwidth costs are increased.

In terms of speedup values, it can be said that 2D models generally exhibit better scalability than 1D models. In most of the matrices, the best of 2D models exhibits better scalability than the best of 1D models, by obtaining lower runtime results. With increasing number of processors, this difference becomes more obvious. This can be attributed to the fact that 2D models have more flexibility in partitioning, which leads them to optimize communication objectives better.

Among all models, the performance of JGD+BP and JGD+CHG is especially worth to note. On average, these two models achieve quite good performance in terms of speedup. In matrices atmosmodl, Chevron4 and kim2, JGD+BP obtains better speedup values. The communication costs of these matrices are largely determined by bandwidth costs rather than by latency costs. From this point of view, FG+BP might be expected to achieve the best results. However, FG+BP usually causes high latency costs, even in the case of these matrices which are not latency bound. JGD+BP, on the other hand, obtains slightly worse bandwidth costs while drastically improving latency costs compared to FG+BP, finding a balance between FG+BP and CKBD+BP, hence performing best in these matrices.

In the remaining nine matrices, the latency-improved versions of 2D models obtain better scalability. The JGD+CHG model is almost always among the two best performing models. When comparing CKBD+CHG, JGD+CHG and FG+CHG, although CKBD+CHG has the lowest maximum number of messages, it has the highest maximum volume, whereas JGD+CHG obtains slightly worse maximum number of messages compared to CKBD+CHG and has the lowest maximum volume among these three models. Hence it is able to strike a good balance between minimizing latency and bandwidth costs, which leads to better scalability. Although

`FG+CHG` has low bandwidth costs, its high latency costs cause it to perform relatively poorly among these models (for example for `rajat31` matrix, the maximum number of messages of `FG+CHG` at 4096 processors is around hundreds).

A noteworthy case is seen for the `cage13` matrix. This matrix is characterized with its very high latency cost. For example, at $K = 8192$ the maximum number of messages is 345 for `1D+BP`. In such matrices, bounding *and* reducing the message count works better than by solely reducing it. As seen in speedup figures, the two models that do so, `CKBD+BP` and `CKBD+CHG`, achieve better scalability.

Judging from performance profiles and speedup curves, we can safely recommend the use of `JGD+CHG` model when latency costs prove vital in performance, and `JGD+BP` model when latency and bandwidth costs are comparable. Although other models may perform better for specific matrices in some cases, it can be said that `JGD+BP` and `JGD+CHG` will not perform too inferior even in these cases (for example, `circuit5M_dc` matrix).

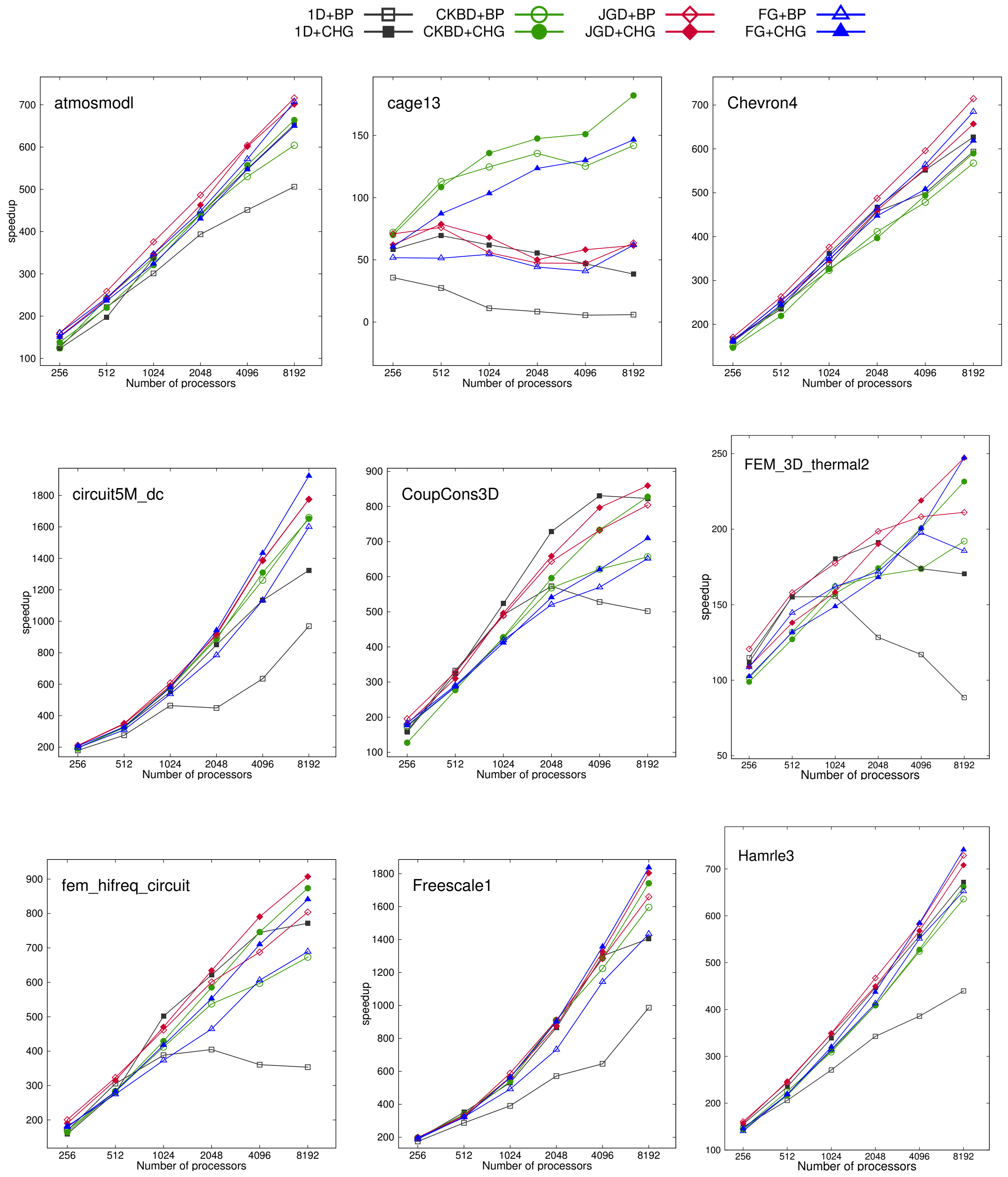


Figure 3.14: Speedup curves for 9 matrices (Part 1 of 3).

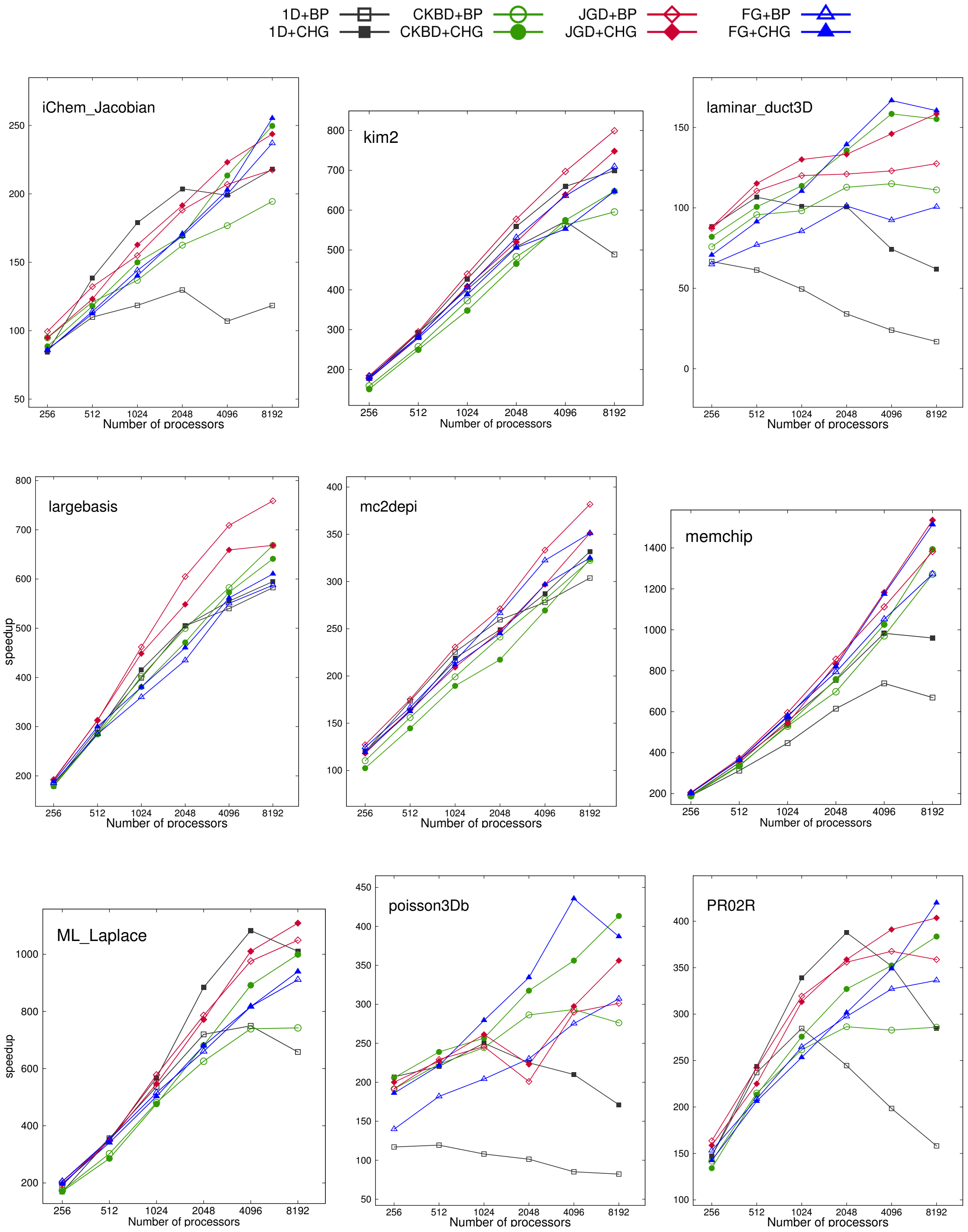


Figure 3.15: Speedup curves for 9 matrices (Part 2 of 3).

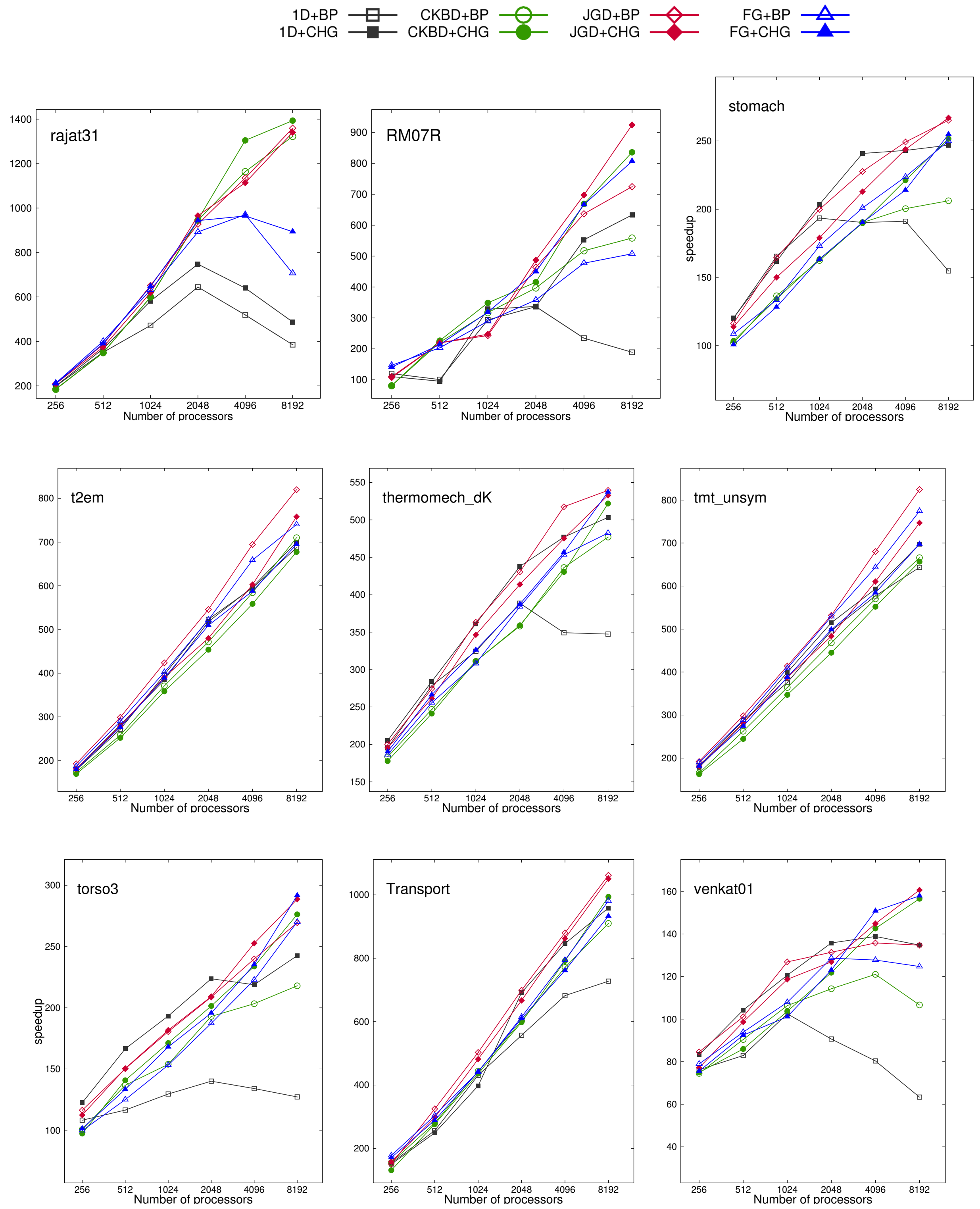


Figure 3.16: Speedup curves for 9 matrices (Part 3 of 3).

Chapter 4

A Recursive Hypergraph Bipartitioning Framework for Reducing Bandwidth and Latency Costs Simultaneously

For irregular applications in the scientific computing domain and several other domains, the intelligent partitioning methods are commonly employed to reduce the communication overhead for efficient parallelization in a distributed setting. Graph and hypergraph partitioning models are ubiquitously utilized in this regard.

4.1 Motivation and Related Work

A common cost model for representing the communication requirements of parallel applications consists of the bandwidth and latency components. The bandwidth component is proportional to the amount (volume) of data transferred and

see [83] for the original work

the latency component is proportional to the number of messages communicated. In order to capture the communication requirements of parallel applications more accurately, both components should be taken into account in the partitioning models.

Although graph/hypergraph partitioning models that address the bandwidth component are abundant in the literature [42, 84, 45, 85, 15, 86, 44, 57, 72, 87, 88], there exist only a few works that also address the latency component. A relatively early work by Uçar and Aykanat [63] proposes a two-phase approach in which the bandwidth and latency components are respectively addressed in the first and second phases by reducing total communication volume in the former and total message count in the latter. They propose the communication hypergraph model for the second phase to capture the messages and the processors involved. Their method is used for partitioning sparse matrices in the context of iterative solvers for nonsymmetric linear systems and exploits the flexibility of using nonconformal partitions for the vectors in the solver. A recent study by Deveci et al. [89] addresses multiple communication cost metrics via hypergraph partitioning in a single phase. These metrics involve the bandwidth-related metrics such as total volume, maximum send/receive volume, etc. as well as the latency-related metrics such as total message count and maximum send message count. All metrics are addressed in the refinement stage of the partitioning. Their approach introduces an additional cost of $O(VK^2)$ to each refinement pass for handling multiple metrics, where V and K denote the number of tasks in the application and the number of processors, respectively. Another work that is reported to reduce the latency cost in an indirect manner uses the $\lambda(\lambda-1)$ metric in order to correctly encapsulate the total communication volume in the target application [90].

There are studies that address the latency overhead via providing an upper bound on the number of messages communicated [52, 91, 51, 50, 92, 53, 54, 57, 59, 60, 5, 16]. These works usually assume that K processors are organized as a $\sqrt{K} \times \sqrt{K}$ mesh and restrict the communication along the rows and the columns of the processor mesh, which results in $O(\sqrt{K})$ messages for each processor. Most of the works bounding the latency component do not explicitly reduce the bandwidth component. The target applications in these works are usually centered around parallelizing sparse matrix computations. In an another work, Kuhlemann and Vassilevski [93] propose a vertex disaggregation scheme tailored for scale-free graphs in order to reduce the message count.

There are a few studies that also aim at reducing volume besides bounding the message count. Çatalyürek and Aykanat [57] propose a two-phase method that makes use of hypergraph partitioning to achieve a Cartesian distribution of sparse matrices, namely 2D checkerboard partitioning. In the first phase, they obtain a rowwise $\sqrt{K}$-way partition and in the second phase, they use multiple vertex weights determined from the partition information of the first phase and obtain a columnwise $\sqrt{K}$-way partition. In both phases, the objective is to minimize the total volume. Boman et al. [60] achieves a similar feat with a faster method for scale-free graphs, again in two phases. In the first phase, their approach can make use of any available graph/hypergraph partitioner to obtain a 1D vertex partition. In the second phase, they use an effective algorithm to redistribute the nonzeros in the off-diagonal blocks to guarantee the $O(\sqrt{K})$ upper bound. These two methods are proposed for efficient parallelization of sparse matrix vector multiplication. Kayaaslan et. al [88] recently proposed a semi-two-dimensional partitioning model that aims to exploit better aspects of 1D and 2D partitioning.

4.2 Contributions

Most of the existing graph/hypergraph partitioning models in the literature address only the bandwidth component while ignoring the latency component. In this work, we propose an augmentation to the existing models in order to minimize the bandwidth and the latency components simultaneously in a *single* phase. Our approach relies on the commonly adopted recursive bipartitioning (RB) framework [42, 74, 15, 43, 94]. The RB framework recursively partitions a given domain of computational tasks and data items into two until desired number subdomains is obtained. Consider a subdomain to be bipartitioned and the set of data items in this subdomain that are required by the tasks in some other subdomain. Keeping these items together in the bipartitioning ensures only one of the new subdomains to send a message to that other subdomain, avoiding an increase in the total number of messages. In order to encourage keeping these items together, we introduce *message nets* to the standard hypergraph model so that dividing these items is penalized with a cost equal to startup latency. The nets of the standard hypergraph model are referred to as the *volume nets* and with the addition of the message nets, this augmented hypergraph now contains both the volume and message nets. Partitioning this hypergraph presents a more accurate picture of the communication cost model as the objective of minimizing the cutsize in the partitioning encapsulates the reduction of both the total volume and the total message count.

Our approach is tailored for the parallel applications in which there exists a single communication phase, that is either preceded or succeeded by a computational phase. The parallel application is also assumed to be performed iteratively and a conformal partition on input and output data is required, where the input of the next iteration is obtained from the output of the current iteration. These common assumptions are suited well to the needs of several applications from various domains. Compared to the standard hypergraph partitioning model in which only the bandwidth component is minimized [42], our approach introduces an additional cost of $O(p \lg_2 K)$ due to the addition of the message nets, where p is the number of pins in the hypergraph. The proposed model does not depend on a specific hypergraph partitioning tool implementation, hence it can make use of any hypergraph partitioner such as PaToH [42, 95], hMetis [74] or Mondriaan [44]. In our experiments, we consider 1D parallel sparse matrix vector multiplication (SpMV) as an example application. Our approach is shown to be effective at reducing the latency component as it attains an 18%-52% reduction in the total number of messages at the expense of an 8%-70% increase in the total volume compared to the standard model. The experiments validate the necessity of addressing both communication components as the proposed model reduces the parallel running time of SpMV up to 29% for 2048 processors on the average.

The rest of this chapter is organized as follows. Section 4.3 describes the properties of the target applications and how to model them with hypergraphs for parallelization. The proposed hypergraph partitioning model and its extensions are given in Section 4.4. Section 4.7 evaluates the proposed model in terms of both the communication statistics and the parallel running time of SpMV.

4.3 Background

Our model heavily relies on the hypergraph partitioning. See Section 3.3.1 for the relevant description and notation.

4.3.1 Recursive Hypergraph Bipartitioning

There are two basic approaches to obtain a K-way partition of a given hypergraph $\mathcal{H}$: direct K-way partitioning and recursive bipartitioning (RB). In the direct approach, $\mathcal{H}$ is directly partitioned into K parts. Our work relies on recursive bipartitioning (RB), hence we give the relevant notation. In RB, a given hypergraph $\mathcal{H}$ is recursively partitioned into two parts until K parts are obtained. Obtaining a K-way partition of $\mathcal{H}$ through RB induces a binary tree with $\lceil \log_2 K \rceil$ levels, which is referred to as an RB tree. For the sake of simplicity, we assume K is a power of two. The ℓth level of the RB tree contains 2^ℓ hypergraphs, denoted with $\mathcal{H}_0^\ell, \ldots, \mathcal{H}_{2^\ell-1}^\ell$ from left to right, $0 \leq \ell \leq \log_2 K$. A bipartition $\Pi = \{\mathcal{V}_L, \mathcal{V}_R\}$ on hypergraph $\mathcal{H}_k^\ell$ in the ℓth level forms two new vertex-induced hypergraphs $\mathcal{H}_{2k}^{\ell+1} = (\mathcal{V}_L, \mathcal{N}_L)$ and $\mathcal{H}_{2k+1}^{\ell+1} = (\mathcal{V}_R, \mathcal{N}_R)$, both in level $\ell + 1$. Here, $\mathcal{V}_L$ and $\mathcal{V}_R$ are respectively used to refer to the left and right part of the bipartition without loss of generality. A single bipartitioning is also referred to as an RB step.

The net sets of the newly formed hypergraphs in an RB step are constructed via cut-net splitting method [42] in order to capture the cutsize (3.1). In this method, a cut-net n_j in $\Pi = \{\mathcal{V}_L, \mathcal{V}_R\}$ is split into two new nets $n_j^L \in \mathcal{N}_L$ and $n_j^R \in \mathcal{N}_R$, where $Pins(n_j^L) = Pins(n_j) \cap \mathcal{V}_L$ and $Pins(n_j^R) = Pins(n_j) \cap \mathcal{V}_R$. Internal

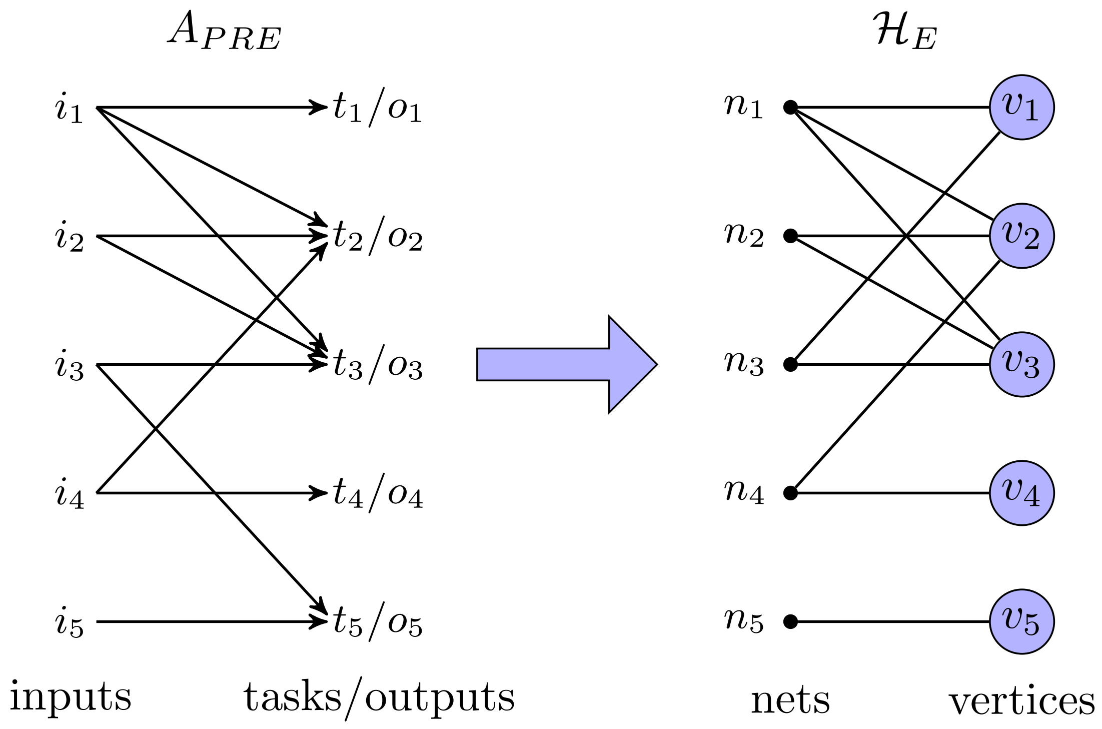


Figure 4.1: (a) An example for A_{PRE}. (b) The hypergraph $\mathcal{H}_E$ that represents A_{PRE}.

nets of $\mathcal{V}_L$ and $\mathcal{V}_R$ are respectively included in $\mathcal{N}_L$ and $\mathcal{N}_R$.

4.3.2 Parallelizing Applications

4.3.2.1 Target Application Properties

Consider an application $\mathcal{A}=(\mathcal{I},\mathcal{T},\mathcal{O})$ to be parallelized, where $\mathcal{I}=\{i_1,\ldots,i_I\}$ is the set of input data items, $\mathcal{T}=\{t_1,\ldots,t_T\}$ is the set of tasks and $\mathcal{O}=\{o_1,\ldots,o_O\}$ is the set of output data items. I, T and O respectively denote the sizes of $\mathcal{I}$, $\mathcal{T}$ and $\mathcal{O}$. The items and tasks of this application constitute a domain to be partitioned for parallelization. The tasks operate on input items and produce output items. There is no dependency among tasks, however there is interaction among the tasks that need the same input as well as the tasks that contribute to the same output. Input $i_j \in \mathcal{I}$ is required by a subset of tasks, denoted by

$tasks(i_j) \subseteq \mathcal{T}$. A subset of tasks produce intermediate results for output $o_j \in \mathcal{O}$, again denoted by $tasks(o_j) \subseteq \mathcal{T}$. $size(t_i)$ denotes the amount of time required to complete task t_i and $size(i_j)$ ($size(o_j)$) denotes the storage size of item i_j (o_j). The tasks are atomic, i.e., each task is processed exactly by one processor. In a parallel setting, tasks and items are distributed among a number of processors.

We focus on applications in which either the intermediate results for o_j are produced by a single task for each $o_j \in \mathcal{O}$ ($|tasks(o_j)| = 1$) or i_j is required by a single task for each $i_j \in \mathcal{I}$ ($|tasks(i_j)| = 1$). In a distributed setting, there is only a single communication phase in both cases, in which either only the inputs or only the intermediate results of the outputs are communicated. The applications that exhibit the properties in the former and the latter cases are respectively denoted with A_{PRE} and A_{POST}. In A_{PRE}, the communications are performed in a so-called *pre-communication* phase, whereas in A_{POST}, the communications are performed in a so-called *post-communication* phase.

In A_{PRE}, the processor responsible for task t_j is also held responsible for storing output o_j. First, for each input i_k, the processor that stores i_k sends it to each processor which is responsible for at least one task in $tasks(i_k)$. This communication operation on i_k is referred to as an *expand* operation. Then, the processor responsible for t_j executes it by operating on inputs $\{i_k : t_j \in tasks(i_k)\}$ to compute the result for o_j in a communication-free manner. An example for A_{PRE} is illustrated in Fig. 4.1(a).

In A_{POST}, the processor responsible for task t_j is also held responsible for storing input i_j. First, the processor responsible for t_j executes it on i_j in a communication-free manner and produces intermediate results for the outputs $\{o_k : t_j \in tasks(o_k)\}$. Then, the processor responsible for o_k receives corresponding intermediate results from each processor which is responsible for at least one task in $tasks(o_k)$ and reduces them through an associative and/or commutative operator. This communication operation on o_k is referred to as a *fold* operation. An example for A_{POST} is illustrated in Fig. 4.2(a).

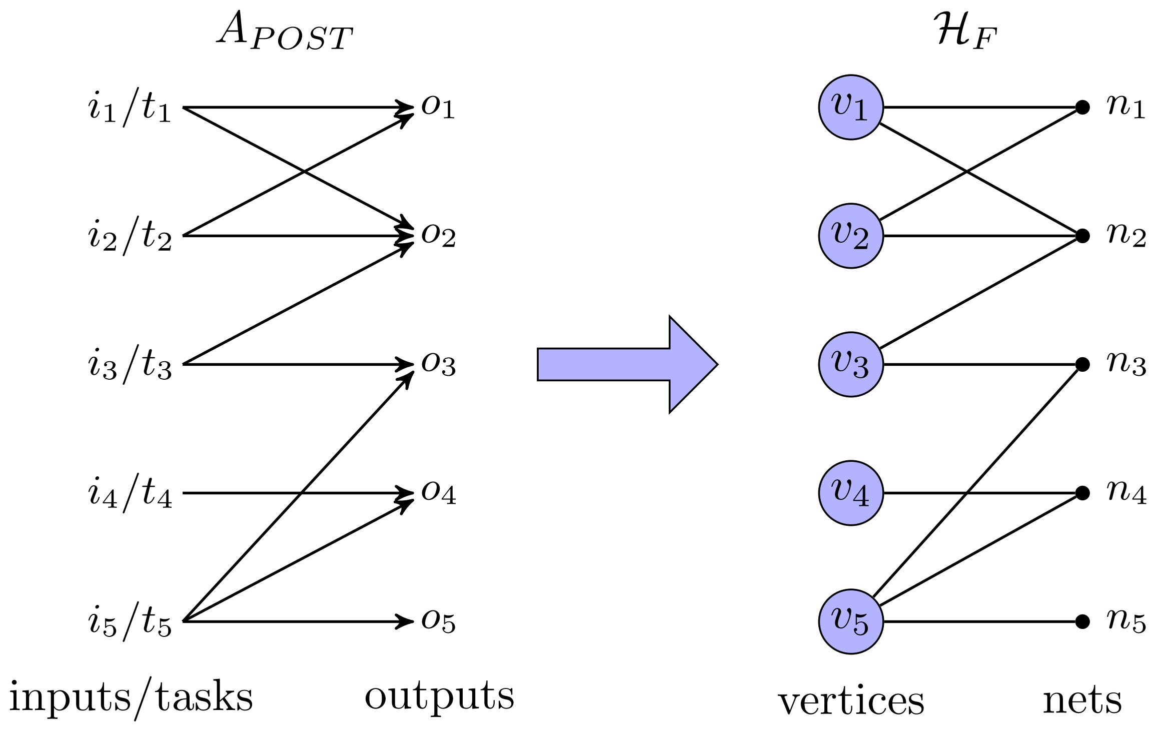


Figure 4.2: (a) An example for A_{POST}. (b) The hypergraph $\mathcal{H}_F$ that represents A_{POST}.

We assume that A_{PRE} and A_{POST} accommodate the following common properties: (i) they are performed repeatedly, (ii) the number of input and output items are equal, and (iii) there exists a one-to-one dependency between input and output items through successive iterations, i.e., output o_j of the current iteration is used to obtain input i_j of the next iteration. Note that if this one-to-one dependency is not respected in assigning items to processors, redundant communication is incurred. For this reason, a *conformal partition* on input and output items should be adopted in which i_j and o_j are assigned to the same processor.

4.3.2.2 Hypergraph Models for A_{PRE} and A_{POST}

We use hypergraphs $\mathcal{H}_E = (\mathcal{V}_E, \mathcal{N}_E)$ and $\mathcal{H}_F = (\mathcal{V}_F, \mathcal{N}_F)$ to represent A_{PRE} and A_{POST}, respectively. Subscripts "E" and "F" are used to denote the fact that A_{PRE} and A_{POST} contain "Expand" and "Fold" operations, respectively. In both

$\mathcal{H}_E$ and $\mathcal{H}_F$, the vertices represent tasks and items, i.e., $\mathcal{V}_E = \mathcal{V}_F = \{v_1, \ldots, v_T\}$, where v_i represents task t_i together with possibly multiple input-output pairs $(i_j,\ o_j)$ such that $tasks(o_j) = \{t_i\}$ for A_{PRE} and $tasks(i_j) = \{t_i\}$ for A_{POST}. The weight of a vertex $w(v_i)$ in both $\mathcal{H}_E$ and $\mathcal{H}_F$ is assigned the amount of time required to execute t_i, i.e., $w(v_i) = size(t_i)$. Both net sets $\mathcal{N}_E$ and $\mathcal{N}_F$ consist of $I = O$ nets, $\mathcal{N}_E = \mathcal{N}_F = \{n_1, \ldots, n_{I=O}\}$. The nets in $\mathcal{N}_E$ capture the interactions between tasks and inputs: For each input i_j, there exists an expand net n_j to represent the expand operation on i_j with $Pins(n_j) = \{v_i : t_i \in tasks(i_j)\}$. The nets in $\mathcal{N}_F$ capture the interactions between tasks and outputs: For each output o_j, there exists a fold net n_j to represent the fold operation on o_j with $Pins(n_j) = \{v_i : t_i \in tasks(o_j)\}$. The cost of an expand net $n_j \in \mathcal{N}_E$ is assigned the size of the respective input i_j multiplied with t_w, i.e., $c(n_j) = size(i_j)\ t_w$. In a similar manner, the cost of a fold net $n_j \in \mathcal{N}_F$ is assigned the size of the respective output o_j multiplied with t_w, i.e., $c(n_j) = size(o_j)\ t_w$. Here, t_w is the time required to transfer a single unit of data item. Figures 4.1(b) and 4.2(b) display the hypergraphs $\mathcal{H}_E$ and $\mathcal{H}_F$ that respectively represent the example applications in Figures 4.1(a) and 4.2(a).

A K-way partition of $\mathcal{H}_E/\mathcal{H}_F$ is decoded to obtain a distribution of tasks and data items among K processors. The responsibility of executing tasks and storing items in the subdomain represented by part $\mathcal{V}_k$ is, without loss of generality, assigned to processor P_k. A cut-net n_j in the partition of $\mathcal{H}_E$ necessitates an expand operation on input i_j from the processor that stores i_j to $\lambda(n_j) - 1$ processors, whereas a cut-net n_j in the partition of $\mathcal{H}_F$ necessitates a fold operation on the intermediate results for output o_j from $\lambda(n_j) - 1$ processors to the processor that stores o_j. These operations respectively amount to $size(i_j)(\lambda(n_j) - 1)$ and $size(o_j)(\lambda(n_j) - 1)$ volume of communication units. The partitioning constraint of maintaining balance on part weights (3.2) in both $\mathcal{H}_E$ and $\mathcal{H}_F$ corresponds to maintaining balance on the processors' expected execution time. The partitioning objective of minimizing cutsize (3.1) in both $\mathcal{H}_E$ and $\mathcal{H}_F$ corresponds to minimizing the total communication volume incurred in pre-/post-communication phases.

4.3.2.3 Examples for A_{PRE} and A_{POST}

We consider parallel sparse matrix vector multiplication (SpMV) $y \leftarrow Ax$ performed in a repeated manner (such as in iterative solvers) which is a common kernel in scientific computing. Here, A is an $n \times n$ matrix, and x and y are vectors of size n. In SpMV, the inputs are x-vector elements, i.e., $\mathcal{I} = \{x_1, \ldots, x_n\}$, and the outputs are y-vector elements, i.e., $\mathcal{O} = \{y_1, \ldots, y_n\}$, where x_j and y_i respectively denote the jth x-vector element and ith y-vector element. A conformal partition on input and output vectors is usually preferred in order to avoid redundant communication.

1D row-parallel SpMV and 1D column-parallel SpMV are examples for A_{PRE} and A_{POST}. In 1D row-parallel SpMV, task t_j stands for the inner product $\langle a_{j*}, x\rangle$, while in 1D column-parallel SpMV, t_j stands for the scalar multiplication $x_j a_{*j}$, where a_{j*} and a_{*j} respectively denote the jth row and jth column of A, for $1 \leq j \leq n$. The size of t_j is equal to the number of nonzeros in jth row and jth column of A, respectively in A_{PRE} and A_{POST}, i.e., the number of multiply-and-add operations in $\langle a_{j*}, x\rangle$ and $x_j a_{*j}$. In both, there exist a total of n inputs, n tasks and n outputs. In 1D row-parallel SpMV, input x_j is required by each inner product $\langle a_{i*}, x\rangle$ such that a_{i*} contains a nonzero in jth column, that is, $tasks(x_j) = \{t_i : a_{ij} \neq 0\}$. The intermediate results for each output y_j are produced only by the task $\langle a_{j*}, x\rangle$, i.e., $tasks(y_j) = \{t_j\}$. In 1D column-parallel SpMV, input x_j is required only by the task $x_j a_{*j}$, i.e., $tasks(x_j) = \{t_j\}$. Each task $x_i a_{*i}$ produces an intermediate result for output y_j such that a_{*i} contains a nonzero in its jth row, that is, $tasks(y_j) = \{t_i : a_{ji} \neq 0\}$. We represent 1D row-parallel and 1D column-parallel SpMVs respectively with $\mathcal{H}_E = (\mathcal{V}_E, \mathcal{N}_E)$ and $\mathcal{H}_F = (\mathcal{V}_F, \mathcal{N}_F)$. The cost of net n_j in both $\mathcal{H}_E$ and $\mathcal{H}_F$ is assigned t_w since it incurs the communication of a single item if it is cut. $\mathcal{H}_E$ and $\mathcal{H}_F$ are respectively called the column-net and row-net hypergraph models and they are proposed in [42].

The objective of partitioning $\mathcal{H}_E/\mathcal{H}_F$ correctly captures the total volume while disregarding the message count (Section 4.3.2.2). To illustrate the aspects of different partitions on these two metrics, we present a motivating example in

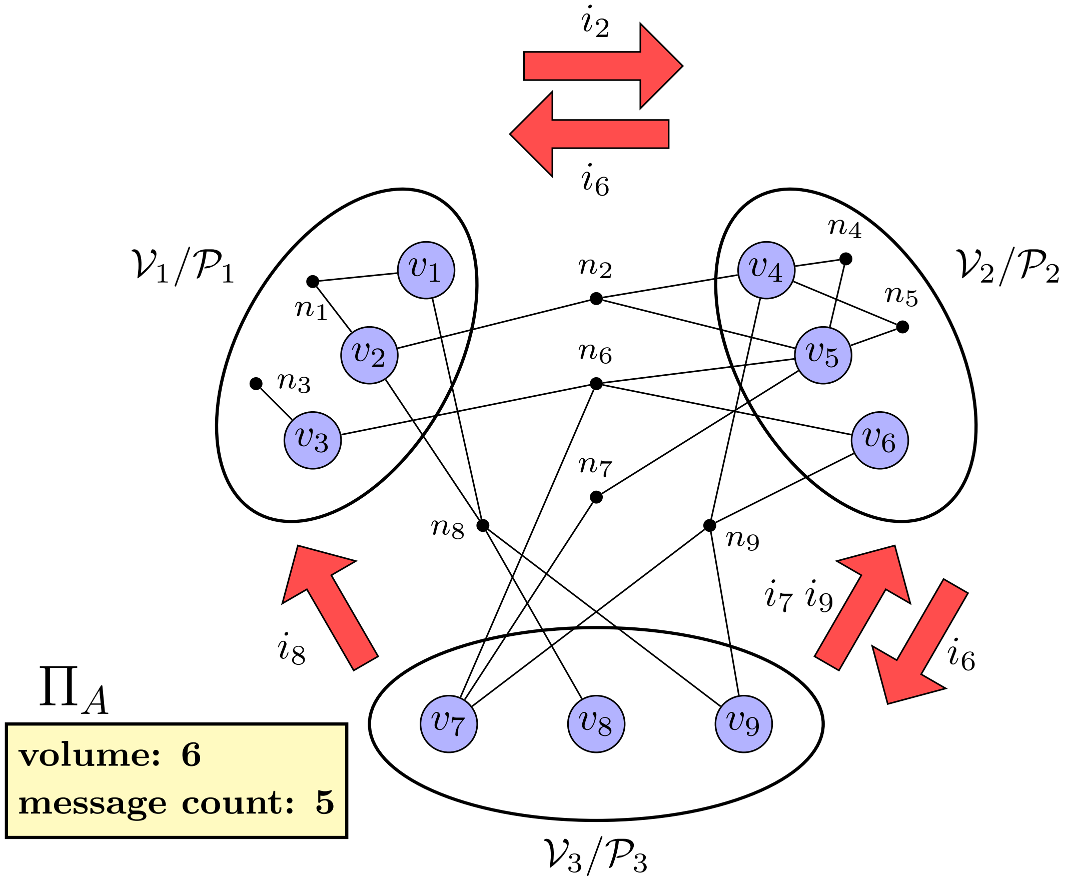


Figure 4.3: 3-way partitioning of hypergraph A_{PRE}. Note that v_j represents both task t_j and input i_j.

Figs. 4.3 and 4.4, where the same $\mathcal{H}_E$ is 3-way partitioned in two different ways. The arrows represent the messages between processors that are associated with the parts in the figure. For example, in Fig. 4.3, i_2 ($=v_2$) needs to be sent from $\mathcal{P}_1$ to $\mathcal{P}_2$ since it is stored by $\mathcal{P}_1$ and the tasks t_4 ($=v_4$) and t_5 ($=v_5$) in $\mathcal{P}_2$ need it (n_2 captures this dependency). The contents of the messages are indicated next to the arrows. In the first partition Π_A in Fig. 4.3, there are five messages and six communicated items, making up a total of $5t_s+6t_w$ communication cost. In the second partition Π_B in Fig. 4.4, there are three messages and seven communicated items, making up a total of $3t_s+7t_w$ communication cost. Partitioning $\mathcal{H}_E$ is more likely to produce Π_A since total volume is lower in Π_A as nets of $\mathcal{H}_E$ encode volume. However, Π_B is more desirable since $3t_s+7t_w$ is less than $5t_s+6t_w$ as t_s is usually much larger than t_w.

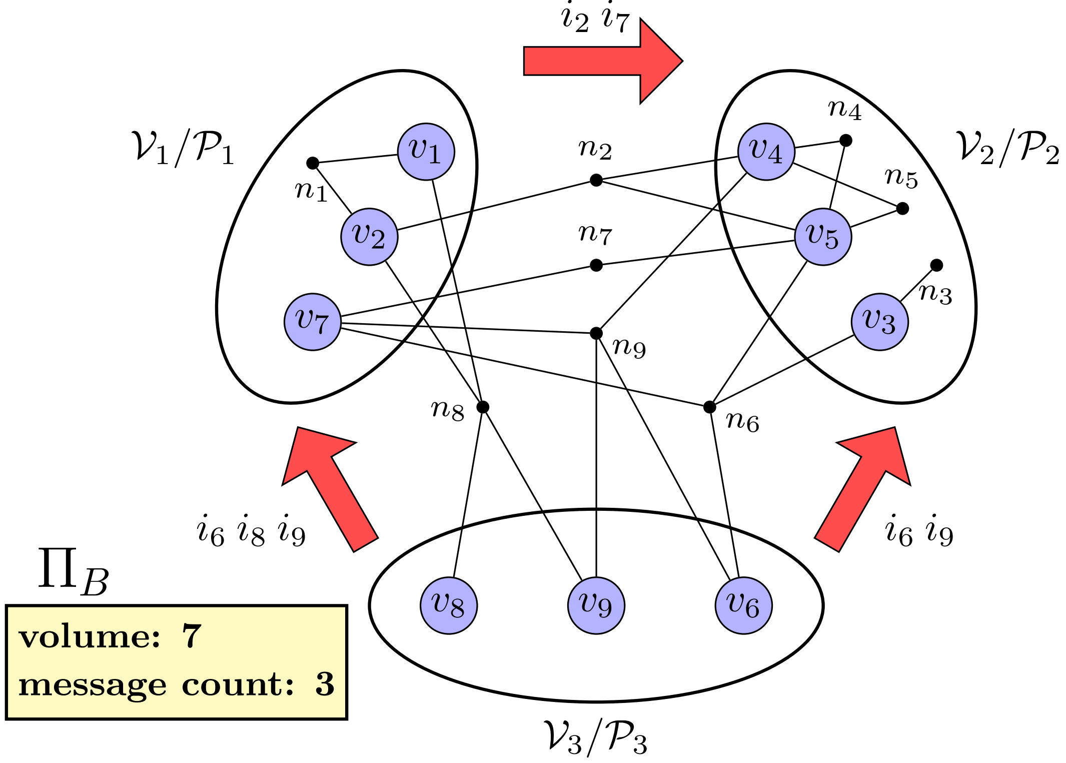

$i_2\ i_7$
$\mathcal{V}_1/\mathcal{P}_1$
$\mathcal{V}_2/\mathcal{P}_2$
$\mathcal{V}_3/\mathcal{P}_3$
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
n_1 n_2 n_3 n_4 n_5 n_6 n_7 n_8 n_9
$i_6\ i_8\ i_9$
$i_6\ i_9$
Π_B
volume: 7
message count: 3

Figure 4.4: Another 3-way partitioning of hypergraph A_{PRE}. Only the parts of v_3, v_6 and v_7 differ in Π_A (see Fig. 4.3) and Π_B. Π_A incurs less volume but more messages while Π_B incurs more volume but less messages.

4.4 Simultaneous Reduction of Bandwidth and Latency Costs

We consider parallelizations of A_{PRE} and A_{POST} via K-way partitions on $\mathcal{H}_E$ and $\mathcal{H}_F$. We describe our model first for A_{PRE} in detail (Section 4.4.1) and then show how to apply it to A_{POST} (Section 4.5.1), as they are dual of each other. Hereinafter, we refer to $\mathcal{H}_E$ as $\mathcal{H}$ and Π_E as Π. We first assume that $I = O = T$ and describe the model for this case and then extend it to the more general case $I = O \geq T$ (Section 4.5.2).

Latency cost: Processor P_k may communicate with P_ℓ due to possibly multiple input items, which we denote with the set $\mathcal{I}_{k\to\ell} \subseteq \mathcal{I}$. These items necessitate a single message from P_k to P_ℓ, whose size is equal to $\sum_{i_j \in \mathcal{I}_{k\to\ell}} size(i_j)$.

The latency cost is proportional to the number of messages (also referred to as message count), i.e.,

$$\sum_{P_k}\sum_{P_\ell}\begin{cases}0 & \mathcal{I}_{k\to\ell}=\emptyset\\ 1 & \mathcal{I}_{k\to\ell}\neq\emptyset\end{cases} \qquad \text{for } 1\le k<\ell\le K. \tag{4.1}$$

The latency cost of communicating one message is t_s.

4.4.1 Encoding Messages in Recursive Hypergraph Bipartitioning

Consider a recursive bipartitioning (RB) tree being produced in a breadth-first manner to obtain a K-way partition of $\mathcal{H}=(\mathcal{V},\mathcal{N})$. Let the RB process be currently at the ℓth level, prior to bipartitioning hypergraph $\mathcal{H}_i^\ell$ in this level. There are currently $2^\ell+i$ hypergraphs, enumerated from left to right, $\mathcal{H}_i^\ell,\ldots,\mathcal{H}_{2^\ell-1}^\ell,\mathcal{H}_0^{\ell+1},\ldots,\mathcal{H}_{2i-1}^{\ell+1}$, at the leaf nodes of the RB tree: $2^\ell-i$ of them at level ℓ and $2i$ of them at level $\ell+1$. The vertex sets of these hypergraphs induce a $(2^\ell+i)$-way vertex partition

$$\Pi_{\text{cur}}(\mathcal{H})=\{\mathcal{V}_i^\ell,\ldots,\mathcal{V}_{2^\ell-1}^\ell,\mathcal{V}_0^{\ell+1},\ldots,\mathcal{V}_{2i-1}^{\ell+1}\}.$$

This vertex partition is also assumed to induce a $(2^\ell+i)$-way processor partition $\mathbb{P}_{\text{cur}}=\{\mathcal{P}_i^\ell,\ldots,\mathcal{P}_{2^\ell-1}^\ell,\mathcal{P}_0^{\ell+1},\ldots,\mathcal{P}_{2i-1}^{\ell+1}\}$, where processor group $\mathcal{P}_i^\ell$ is held responsible for the items/tasks that are in the subdomain represented by $\mathcal{V}_i^\ell$. An example Π_{cur} is seen in the upper RB tree in Fig. 4.5.

We refer to the current hypergraph to be bipartitioned $\mathcal{H}_i^\ell$ as $\mathcal{H}_{\text{cur}}=(\mathcal{V}_{\text{cur}}=\mathcal{V}_i^\ell,\mathcal{N}_{\text{cur}}=\mathcal{N}_i^\ell)$. This bipartitioning generates $\Pi(\mathcal{H}_{\text{cur}})=\{\mathcal{V}_L,\mathcal{V}_R\}$ and forms two new hypergraphs $\mathcal{H}_L=(\mathcal{V}_L,\mathcal{N}_L)$ and $\mathcal{H}_R=(\mathcal{V}_R,\mathcal{N}_R)$. Note that $\mathcal{H}_L=\mathcal{H}_{2i}^{\ell+1}$ and $\mathcal{H}_R=\mathcal{H}_{2i+1}^{\ell+1}$. After bipartitioning, there now exist $2^\ell+i+1$ hypergraphs at the leaf nodes and their vertex sets induce a $(2^\ell+i+1)$-way partition:

$$\Pi_{\text{new}}(\mathcal{H})=\Pi_{\text{cur}}(\mathcal{H})-\{\mathcal{V}_{\text{cur}}\}\cup\{\mathcal{V}_L,\mathcal{V}_R\}.$$

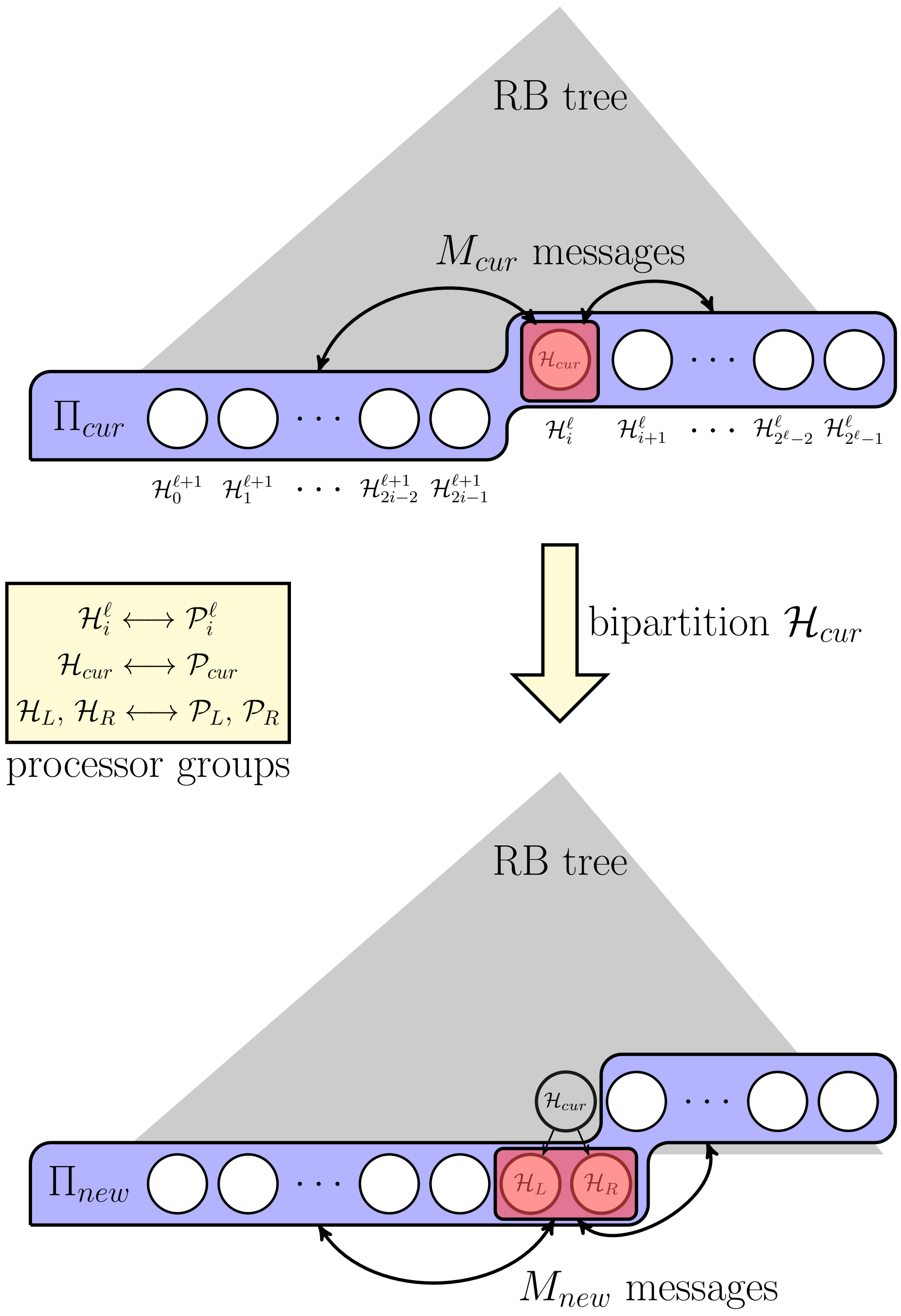

RB tree
M_{cur} messages
Π_{cur}
$\mathcal{H}_{cur}$
$\mathcal{H}_i^\ell$ $\mathcal{H}_{i+1}^\ell$ $\cdots$ $\mathcal{H}_{2^\ell-2}^\ell$ $\mathcal{H}_{2^\ell-1}^\ell$
$\mathcal{H}_0^{\ell+1}$ $\mathcal{H}_1^{\ell+1}$ $\cdots$ $\mathcal{H}_{2i-2}^{\ell+1}$ $\mathcal{H}_{2i-1}^{\ell+1}$
bipartition $\mathcal{H}_{cur}$
$\mathcal{H}_i^\ell \longleftrightarrow \mathcal{P}_i^\ell$
$\mathcal{H}_{cur} \longleftrightarrow \mathcal{P}_{cur}$
$\mathcal{H}_L,\, \mathcal{H}_R \longleftrightarrow \mathcal{P}_L,\, \mathcal{P}_R$
processor groups
RB tree
$\mathcal{H}_{cur}$
Π_{new}
$\mathcal{H}_L$ $\mathcal{H}_R$
M_{new} messages

Figure 4.5: The state of the RB tree and the number of messages from/to $\mathcal{P}_{\text{cur}}$ and $\{\mathcal{P}_L, \mathcal{P}_R\}$ to/from the other processor groups before and after bipartitioning $\mathcal{H}_{\text{cur}}$. The processor groups corresponding to the vertex sets of the hypergraphs are shown in the box.

Bipartitioning $\mathcal{V}_{\text{cur}}$ into $\mathcal{V}_L$ and $\mathcal{V}_R$ is assumed to also bipartition the processor group $\mathcal{P}_{\text{cur}}$ into two processor groups $\mathcal{P}_L$ and $\mathcal{P}_R$. In accordance, $\mathbb{P}_{\text{new}} = \mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\} \cup \{\mathcal{P}_L, \mathcal{P}_R\}$.

Fig. 4.5 displays the two states of the RB tree before and after bipartitioning $\mathcal{H}_{\text{cur}}$ and highlights the messages communicated. Let M_{cur} be the number of messages between $\mathcal{P}_{\text{cur}}$ and $\mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\}$ and $M_{\text{new}} \geq M_{\text{cur}}$ be the number of messages between $\{\mathcal{P}_L, \mathcal{P}_R\}$ and $\mathbb{P}_{\text{new}} - \{\mathcal{P}_L, \mathcal{P}_R\}$. M_{new} can be at most $2M_{\text{cur}}$ which occurs when both $\mathcal{P}_L$ and $\mathcal{P}_R$ communicate with every $\mathcal{P}_k$ that $\mathcal{P}_{\text{cur}}$ communicates with. A new message is incurred when items/tasks that necessitate a message between $\mathcal{P}_{\text{cur}}$ and $\mathcal{P}_k$ get scattered across $\mathcal{P}_L$ and $\mathcal{P}_R$. Consequently, after bipartitioning, both $\mathcal{P}_L$ and $\mathcal{P}_R$ communicate with $\mathcal{P}_k$. Here, the idea is to find a way for items/-tasks which as a whole necessitate a message between $\mathcal{P}_{\text{cur}}$ and $\mathcal{P}_k$ to be assigned together to either $\mathcal{P}_L$ or $\mathcal{P}_R$ so that only one of them communicates with $\mathcal{P}_k$. By doing so, the goal is to keep the number of messages between $\{\mathcal{P}_L, \mathcal{P}_R\}$ and the remaining processor groups in $\mathbb{P}_{\text{new}}$ as small as possible.

To this end, we define new nets, referred to as *message nets*, to keep the vertices corresponding to items/tasks that necessitate messages altogether. We extend $\mathcal{H}_{\text{cur}} = (\mathcal{V}_{\text{cur}}, \mathcal{N}_{\text{cur}})$ to $\mathcal{H}^M_{\text{cur}} = (\mathcal{V}_{\text{cur}}, \mathcal{N}^M_{\text{cur}})$ by adding message nets and keeping the expand nets as is, referred to as *volume nets*. We include both volume and message nets in $\mathcal{H}^M_{\text{cur}}$ in order to reduce the total volume and the total message count simultaneously. The following sections define message nets and present an algorithm for forming them.

4.4.2 Message Nets

Recall that a vertex v_j in $\mathcal{V}$ represents input i_j besides task t_j and output o_j, and the processor that stores i_j is also held responsible for the possible expand operation on i_j. Since net n_j represents this expand operation, for convenience, we define a function $src : \mathcal{N} \rightarrow \mathcal{V}$, that maps each original net $n_j \in \mathcal{N}$ to its *source* vertex $src(n_j) \in \mathcal{V}$, where $src(n_j) = v_j$.

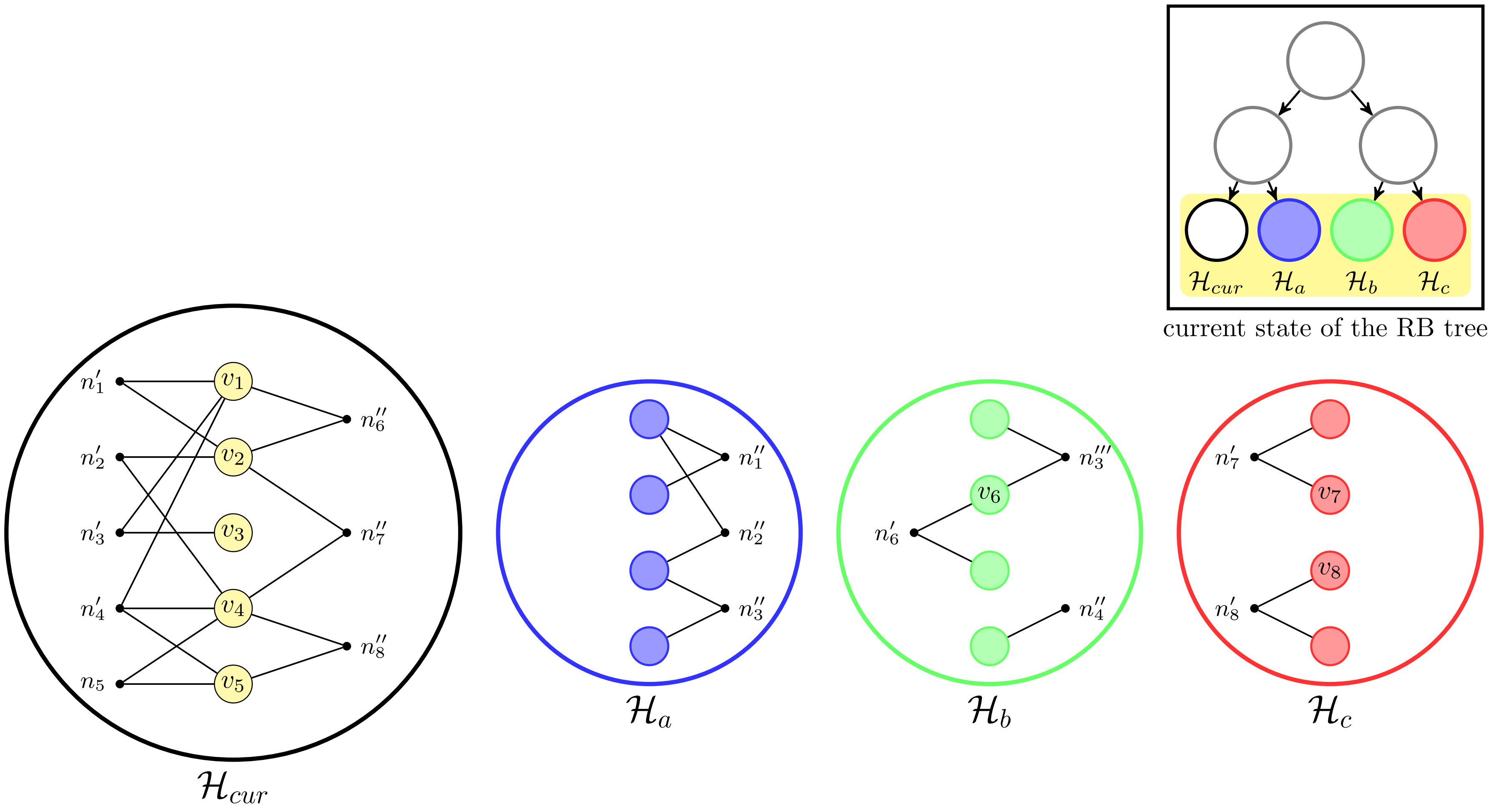


Figure 4.6: An example RB tree produced in the partitioning process. There are four hypergraphs in the leaf nodes of the RB tree: $\mathcal{H}_{\mathrm{cur}}$, $\mathcal{H}_a$, $\mathcal{H}_b$ and $\mathcal{H}_c$.

To aid the discussions in this section, we present an example RB tree in Fig. 4.6 that currently consists of four leaf hypergraphs $\mathcal{H}_{\text{cur}}$, $\mathcal{H}_a$, $\mathcal{H}_b$ and $\mathcal{H}_c$, whose vertex sets form 4-way partition Π_{cur}. We refer to the nets in given $\mathcal{H}=\mathcal{H}_0^0$ as *original nets* and use these nets in describing the algorithm for forming the message nets. An original net may split several times during RB or it may not split by being uncut in the bipartitionings it takes part in. For example in Fig. 4.6, the original net n_3 has split three times, producing n_3', n_3'' and n_3''' in $\mathcal{H}_{\text{cur}}$, $\mathcal{H}_a$ and $\mathcal{H}_b$, respectively. Observe that the vertices connected by n_3 in $\mathcal{H}$ are equal to the union of the vertices connected by n_3', n_3'' and n_3''' in the hypergraphs at the leaf nodes. On the other hand, the original net n_5 is never split and currently in $\mathcal{H}_{\text{cur}}$. In the figure, without loss of generality, a split net with a single prime in the superscript (e.g., n_3') connects the source vertex of the respective original net (n_3), while split nets with two or more primes (e.g., n_3'', n_3''') do not.

In the formation of the message nets, we make use of the most recent $(2^\ell+i)$-way partition information Π_{cur}. The message nets are categorized into two as send message nets and receive message nets, or simply *send nets* and *receive nets.*

We form a send net s_k for each $\mathcal{P}_k \neq \mathcal{P}_{\text{cur}}$ to which $\mathcal{P}_{\text{cur}}$ sends a message. s_k connects the vertices corresponding to the items sent to $\mathcal{P}_k$:

$$\begin{aligned} Pins(s_k) = \{v_j \in \mathcal{V}_{\text{cur}} :\ & src(n_j) = v_j \text{ and} \\ & Pins(n_j) \cap \mathcal{V}_{k\neq\text{cur}} \neq \emptyset\}. \end{aligned} \tag{4.2}$$

In other words, s_k connects source vertex v_j of each original net n_j that represents the expand operation which necessitates sending i_j to $\mathcal{P}_k$. Algorithm 2 shows the formation of the set of send nets $\mathcal{N}_{\text{SND}}$, which is initially empty (line 1). For each vertex $v_j \in \mathcal{V}_{\text{cur}}$, we first retrieve net n_j such that $src(n_j)=v_j$ (line 4). Then, the vertices which are connected by n_j and not in $\mathcal{V}_{\text{cur}}$ are traversed (line 5). Let v be such a vertex, currently in part $\mathcal{V}_k$. Since $\mathcal{P}_k$ needs i_j due to the task represented by v, i_j is sent from $\mathcal{P}_{\text{cur}}$ to $\mathcal{P}_k$. Hence, the vertices connected by send net s_k representing this message are updated (lines 7-12): If s_k is processed for the first time, $Pins(s_k)$ is initialized with $\{v_j\}$ and s_k is added to $\mathcal{N}_{\text{SND}}$ (lines 8 and 10), otherwise, $Pins(s_k)$ is updated to include v_j (line 12). Since $\mathcal{P}_{\text{cur}}$ can send at most one message to each of $2^\ell+i-1$ processor groups, the number of send nets

Algorithm 2: FORM-MESSAGE-NETS

```
Require: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), \mathcal{V}_{\text{cur}}, t_s$
1: $\mathcal{N}_{\text{SND}} = \emptyset$        ▷ The set of send nets
2: $\mathcal{N}_{\text{RCV}} = \emptyset$        ▷ The set of receive nets
3: for $v_j \in \mathcal{V}_{\text{cur}}$ do
4:     Let $src(n_j) = v_j$
       ▷ Add send nets
5:     for $v \in Pins(n_j)$ and $v \notin \mathcal{V}_{\text{cur}}$ do
6:         Let $\mathcal{V}_k$ be the part $v$ is currently in
7:         if $s_k \notin \mathcal{N}_{\text{SND}}$ then
8:             $Pins(s_k) = \{v_j\}$
9:             $c(s_k) = t_s$
10:            $\mathcal{N}_{\text{SND}} = \mathcal{N}_{\text{SND}} \cup \{s_k\}$
11:        else
12:            $Pins(s_k) = Pins(s_k) \cup \{v_j\}$
       ▷ Add receive nets
13:    for $n \in Nets(v_j)$ and $n \neq n_j$ and $src(n) \notin \mathcal{V}_{\text{cur}}$ do
14:        $v = src(n)$
15:        Let $\mathcal{V}_k$ be the part $v$ is currently in
16:        if $r_k \notin \mathcal{N}_{\text{RCV}}$ then
17:            $Pins(r_k) = \{v_j\}$
18:            $c(r_k) = t_s$
19:            $\mathcal{N}_{\text{RCV}} = \mathcal{N}_{\text{RCV}} \cup \{r_k\}$
20:        else
21:            $Pins(r_k) = Pins(r_k) \cup \{v_j\}$
22: return $\mathcal{N}_{\text{SND}}, \mathcal{N}_{\text{RCV}}$
```

included in $\mathcal{H}^M_{\text{cur}}$ is at most $2^\ell+i-1$, i.e., $0 \leq |\mathcal{N}_{\text{SND}}| \leq 2^\ell+i-1$. In Fig. 4.7, the send nets s_a and s_b are formed and included in $\mathcal{H}^M_{\text{cur}}$ to represent the messages from $\mathcal{P}_{\text{cur}}$ to $\mathcal{P}_a$ and $\mathcal{P}_b$. s_a connects the respective source vertices v_1, v_2 and v_3 of the original nets n_1, n_2 and n_3 since $\mathcal{H}_a$ contains vertices that are also connected by these nets (indicated by the split nets n''_1, n''_2 and n''_3). Similarly, s_b connects v_3 and v_4 due to the vertices connected by n_3 and n_4 in $\mathcal{H}_b$.

We form a receive net r_k for each $\mathcal{P}_k \neq \mathcal{P}_{\text{cur}}$ from which $\mathcal{P}_{\text{cur}}$ receives a message.

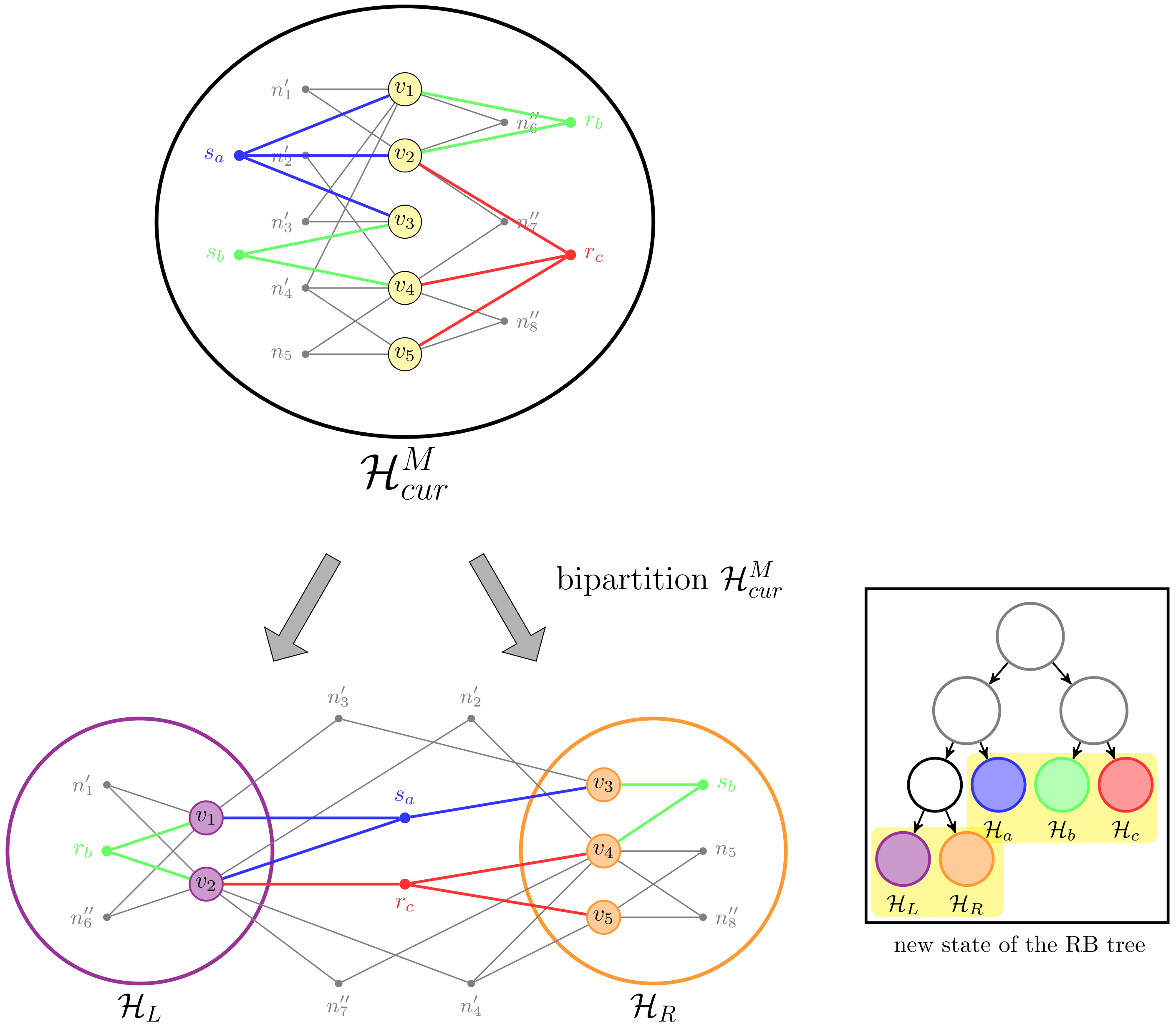


Figure 4.7: Addition of two send (s_a and s_b) and two receive (r_b and r_c) message nets to form $\mathcal{H}_{\text{cur}}^{M}$. Then, $\mathcal{H}_{\text{cur}}^{M}$ is bipartitioned to obtain $\mathcal{H}_L$ and $\mathcal{H}_R$. The colors of the message nets indicate the processor groups that the respective messages are sent to or received from. The volume nets in $\mathcal{H}_{\text{cur}}^{M}$, $\mathcal{H}_L$ and $\mathcal{H}_R$ are faded out to attract the focus on the message nets.

r_k connects the vertices corresponding to the tasks that need items from $\mathcal{P}_k$:

$$\begin{aligned} Pins(r_k) = \{v_j \in \mathcal{V}_{\text{cur}} :\ & v_j \in Pins(n) \text{ and} \\ & src(n) \in \mathcal{V}_{k \neq \text{cur}}\}. \end{aligned} \tag{4.3}$$

In other words, r_k connects vertex v_j which is connected by each original net n representing the expand operation that necessitates to receive the respective item from $\mathcal{P}_k$. Algorithm 2 shows the formation of the set of receive nets $\mathcal{N}_{\text{RCV}}$, which is initially empty (line 2). For each vertex $v_j \in \mathcal{V}_{\text{cur}}$, we traverse the nets that connect v_j other than n_j whose source vertices are not in $\mathcal{V}_{\text{cur}}$ (line 13). Let n be such a net and v be its source vertex, currently in $\mathcal{V}_k$ (lines 14-15). Since $\mathcal{P}_{\text{cur}}$ needs the item corresponding to v due to task t_j represented by v_j, this item is received by $\mathcal{P}_{\text{cur}}$ from $\mathcal{P}_k$. Hence, the vertices connected by receive net r_k representing this message are updated (lines 16-21): If r_k is processed for the first time, $Pins(r_k)$ is initialized with $\{v_j\}$ and r_k is added to $\mathcal{N}_{\text{RCV}}$ (lines 17 and 19), otherwise, $Pins(r_k)$ is updated to include v_j (line 21). Since $\mathcal{P}_{\text{cur}}$ can receive at most one message from each of the $2^\ell + i - 1$ processor groups, the number of receive nets included in $\mathcal{H}^M_{\text{cur}}$ is at most $2^\ell + i - 1$, i.e., $0 \leq |\mathcal{N}_{\text{RCV}}| \leq 2^\ell + i - 1$. In Fig. 4.7, the receive nets r_b and r_c are formed and included in $\mathcal{H}^M_{\text{cur}}$ to represent the messages from $\mathcal{P}_b$ and $\mathcal{P}_c$ to $\mathcal{P}_{\text{cur}}$. r_b connects v_1 and v_2, which are connected by n_6 (indicated by the split net n_6''), since $\mathcal{H}_b$ contains the source vertex of n_6. Similarly, r_c connects v_2, v_4 and v_5 due to the nets n_7 and n_8, both of which have their source vertices in $\mathcal{H}_c$.

Recall that the cost of a volume net in $\mathcal{H}_{\text{cur}}$ is $size(i_j)\ t_w$ and captures the bandwidth cost. To capture the latency cost via message nets, the costs of these nets are assigned the startup latency:

$$c(s_k) = c(r_k) = t_s, \quad \text{for } s_k \in \mathcal{N}_{\text{SND}} \text{ and } r_k \in \mathcal{N}_{\text{RCV}}, \tag{4.4}$$

in lines 9 and 18 of Algorithm 2. Note that the cost of a volume net is the size of the corresponding item in terms of t_w, whereas the cost of a message net is unit in terms of t_s since it encapsulates exactly one message. One might attempt to assign $t_s + mt_w$ as the cost of a message net, where m is the number of items in the message. However, the mt_w cost is already incurred once by the corresponding cut volume nets in the earlier bipartitionings. Hence, we assign the same cost of t_s to each message net independent of their size. The message nets have a higher cost than the volume nets since the startup time of a message is significantly higher than the time required to transmit a word. Finally, the message nets in $\mathcal{N}_{\text{SND}}$ and $\mathcal{N}_{\text{RCV}}$ are returned (line 22).

4.4.3 Partitioning and Correctness

The newly formed hypergraph $\mathcal{H}^M_{\text{cur}}$ is then bipartitioned to obtain $\Pi(\mathcal{H}^M_{\text{cur}}) = \{\mathcal{V}_L, \mathcal{V}_R\}$. Maintaining balance in partitioning $\mathcal{H}^M_{\text{cur}}$ is the same with that of $\mathcal{H}_{\text{cur}}$ since vertices and their weights are the same in both hypergraphs. With the newly introduced message nets, the cutsize of Π is given by

$$\begin{aligned} cut(\Pi) &= \sum_{n_j \in \mathcal{N}^{\text{cut}}_{\text{cur}}} c(n_j) + \sum_{s_k \in \mathcal{N}^{\text{cut}}_{\text{SND}}} c(s_k) + \sum_{r_k \in \mathcal{N}^{\text{cut}}_{\text{RCV}}} c(r_k) \\ &= \sum_{n_j \in \mathcal{N}^{\text{cut}}_{\text{cur}}} size(i_j)\ t_w + |\mathcal{N}^{\text{cut}}_{\text{SND}}|\ t_s + |\mathcal{N}^{\text{cut}}_{\text{RCV}}|\ t_s \end{aligned} \tag{4.5}$$

where $\mathcal{N}^{\text{cut}}_{\text{cur}}$, $\mathcal{N}^{\text{cut}}_{\text{SND}}$ and $\mathcal{N}^{\text{cut}}_{\text{RCV}}$ respectively denote the sets of cut volume nets, cut send nets and cut receive nets in $\Pi(\mathcal{H}^M_{\text{cur}})$.

Let $msg(\mathbb{P})$ denote the total number of messages communicated among the processor groups in $\mathbb{P}$.

Theorem 1. *Consider an RB tree prior to bipartitioning the ith hypergraph $\mathcal{H}^M_{\text{cur}} = (\mathcal{V}_{\text{cur}}, \mathcal{N}^M_{\text{cur}})$ in the ℓth level with message nets added. The vertex sets of the leaf hypergraphs of the RB tree are assumed to induce a $(2^\ell+i)$-way processor partition $\mathbb{P}_{\text{cur}}$. Suppose that the bipartition $\Pi(\mathcal{H}^M_{\text{cur}}) = \{\mathcal{V}_L, \mathcal{V}_R\}$ generates two new leaf hypergraphs $\mathcal{H}_L$ and $\mathcal{H}_R$, where processor groups $\mathcal{P}_L$ and $\mathcal{P}_R$ are associated with $\mathcal{V}_L$ and $\mathcal{V}_R$. After bipartitioning, the vertex sets of the hypergraphs are assumed to induce a $(2^\ell+i+1)$-way processor partition $\mathbb{P}_{\text{new}} = \mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\} \cup \{\mathcal{P}_L, \mathcal{P}_R\}$. Minimizing the number of cut message nets in $\Pi(\mathcal{H}^M_{\text{cur}})$ minimizes the increase ΔM in the number of messages between $\mathcal{P}_{\text{cur}}$ and $\mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\}$, which is given by*

$$\Delta M = msg(\mathbb{P}_{\text{new}}) - msg(\mathbb{P}_{\text{cur}}) - msg(\{\mathcal{P}_L, \mathcal{P}_R\}), \tag{4.6}$$

where $msg(\{\mathcal{P}_L, \mathcal{P}_R\}) \in \{0, 1, 2\}$.

Proof. A send net s_k in $\mathcal{H}^M_{\text{cur}}$ signifies a message from $\mathcal{P}_{\text{cur}}$ to $\mathcal{P}_k \in \mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\}$. If s_k is a cut-net in $\Pi(\mathcal{H}^M_{\text{cur}})$, then both $\mathcal{P}_L$ and $\mathcal{P}_R$ send a message to $\mathcal{P}_k$. The message from $\mathcal{P}_L$ to $\mathcal{P}_k$ and the message from P_R to P_k respectively contain the items corresponding to the vertices in $Pins(s_k) \cap \mathcal{V}_L$ and $Pins(s_k) \cap \mathcal{V}_R$. Hence, a cut send net contributes one to ΔM. If s_k is uncut being in either $\mathcal{V}_L$ or $\mathcal{V}_R$, then only the respective processor group sends a message to $\mathcal{P}_k$, whose content is exactly the same with the message from $\mathcal{P}_{\text{cur}}$ to $\mathcal{P}_k$. Hence, an uncut send net does not contribute to ΔM.

In a dual manner, a receive net r_k in $\mathcal{H}^M_{\text{cur}}$ signifies a message from $\mathcal{P}_k \in \mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\}$ to $\mathcal{P}_{\text{cur}}$. If r_k is a cut-net in $\Pi(\mathcal{H}^M_{\text{cur}})$, then both $\mathcal{P}_L$ and $\mathcal{P}_R$ receive a message from $\mathcal{P}_k$. The message from $\mathcal{P}_k$ to $\mathcal{P}_L$ and the message from $\mathcal{P}_k$ to $\mathcal{P}_R$ respectively contain the items required by the tasks corresponding to the vertices in $Pins(r_k) \cap \mathcal{V}_L$ and $Pins(r_k) \cap \mathcal{V}_R$. Hence, a cut receive net also contributes one to ΔM. If r_k is uncut being in either $\mathcal{V}_L$ or $\mathcal{V}_R$, then only the respective processor group receives a message from $\mathcal{P}_k$, whose content is exactly the same with the message from $\mathcal{P}_k$ to $\mathcal{P}_{\text{cur}}$. Hence, an uncut receive net does not contribute to ΔM.

The message nets are oblivious to the messages between $\mathcal{P}_L$ and $\mathcal{P}_R$ since our approach introduces these message nets for the processor groups in $\mathbb{P}_{\text{cur}}$. For this reason, $msg(\{\mathcal{P}_L, \mathcal{P}_R\})$ is not taken into account. Therefore, ΔM is equal to the number of cut message nets. □

By Theorem 1, the number of cut message nets in $\Pi(\mathcal{H}^M_{\text{cur}})$, $|\mathcal{N}^{\text{cut}}_{\text{SND}}| + |\mathcal{N}^{\text{cut}}_{\text{RCV}}|$, is equal to the increase in the message count, where the new messages between $\mathcal{P}_L$ and $\mathcal{P}_R$ are excluded. In other words, the number of cut message nets corresponds to the increase in the number of messages between $\mathcal{P}_{\text{cur}}$ and $\mathbb{P}_{\text{cur}} - \{\mathcal{P}_{\text{cur}}\}$ with $\mathcal{P}_{\text{cur}}$ bipartitioned into $\{\mathcal{P}_L, \mathcal{P}_R\}$ in $\mathbb{P}_{\text{new}}$. Observe that each cut message net contributes its associated cost $c(s_k)$ or $c(r_k)$ to the cutsize (4.5). For this reason as well as because of the presence of volume nets in (4.5), minimizing the cutsize does not exactly correspond to but relates to minimizing the number of cut message nets. Considering both the volume and the message nets, minimizing the cutsize corresponds to reducing both the total volume and the total message count.

Partitioning $\mathcal{H}^M_{\text{cur}}$ is oblivious to the messages between $\mathcal{P}_L$ and $\mathcal{P}_R$. However, $msg(\{\mathcal{P}_L, \mathcal{P}_R\})$ is negligible compared to $msg(\mathbb{P}_{\text{new}}) - msg(\mathbb{P}_{\text{cur}})$ since it is upper bounded by two. Moreover, $msg(\{\mathcal{P}_L, \mathcal{P}_R\})$ is empirically found to be almost constant as 2, being 0 or 1 in only 0.1% of 1M bipartitions. Note that $0 \leq \Delta M \leq 2(2^\ell + i - 1)$. The worst case for ΔM occurs when $\mathcal{H}^M_{\text{cur}}$ contains a send and a receive net for each other processor group in $\mathbb{P}_{\text{cur}}$ and they all become cut in $\Pi(\mathcal{H}^M_{\text{cur}})$.

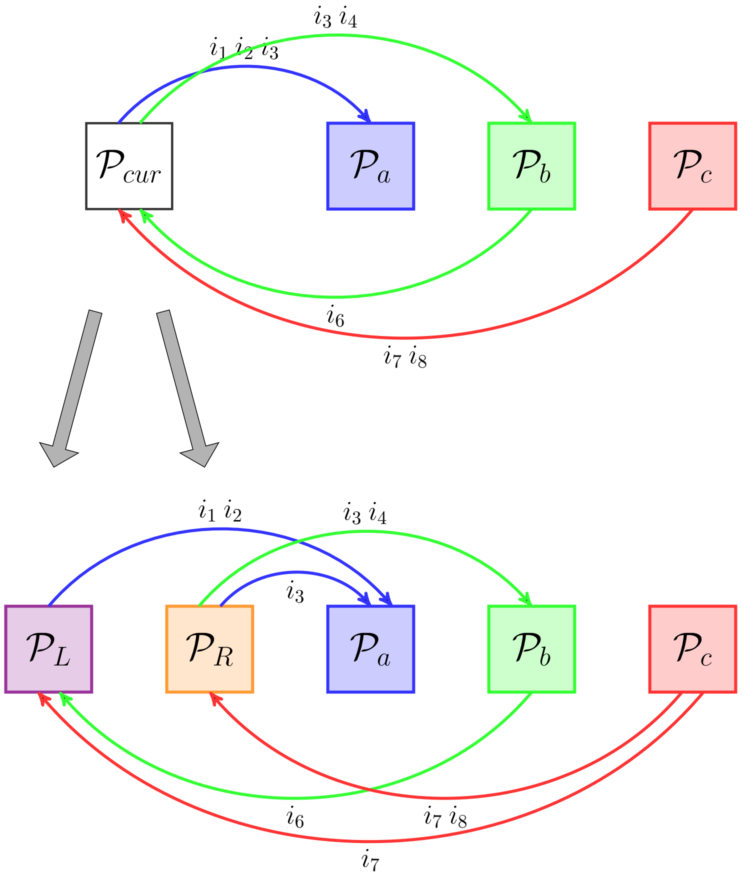


Figure 4.8: The messages communicated among the respective processor groups. $\mathcal{P}_{\text{cur}}$, $\mathcal{P}_a$, $\mathcal{P}_b$ and $\mathcal{P}_c$ are respectively associated with $\mathcal{H}_{\text{cur}}$, $\mathcal{H}_a$, $\mathcal{H}_b$ and $\mathcal{H}_c$ (see Figs. 4.5 and 4.7.). The colors of the message nets indicate the processor groups that the respective messages are sent to or received from.

In Fig. 4.7, there are two send nets s_a and s_b and two receive nets r_b and r_c in $\mathcal{H}^M_{\text{cur}}$. Among the send nets, s_a is a cut-net whereas s_b is an uncut net in $\mathcal{H}_R$ after bipartitioning. As seen in Fig. 4.8, s_a necessitates both $\mathcal{P}_L$ and $\mathcal{P}_R$ to send a message to $\mathcal{P}_a$ due to the items $\{i_1, i_2\}$ and $\{i_3\}$, respectively. Since s_a is a cut-net, it increases the total message count by one as $\mathcal{P}_{\text{cur}}$ was sending a message to $\mathcal{P}_a$. s_b necessitates only $\mathcal{P}_R$ to send a message to $\mathcal{P}_b$ due to the items $\{i_3, i_4\}$. Since s_b is an uncut net, it does not change the total message count as $\mathcal{P}_{\text{cur}}$ was already sending a message to $\mathcal{P}_b$. Among the receive nets, r_c is a cut-net whereas r_b is an uncut net in $\mathcal{H}_L$ after bipartitioning. As seen in Fig. 4.8, r_c necessitates both $\mathcal{P}_L$ and $\mathcal{P}_R$ to receive a message from $\mathcal{P}_c$ due to the tasks $\{t_2\}$ and $\{t_4, t_5\}$, respectively. Since r_c is a cut-net, it increases the total message count by one as $\mathcal{P}_{\text{cur}}$ was receiving a message from $\mathcal{P}_c$. r_b necessitates only $\mathcal{P}_L$ to receive a message from $\mathcal{P}_b$ due to the tasks $\{t_1, t_2\}$. Since r_b is an uncut net, it does not change the total message count as $\mathcal{P}_{\text{cur}}$ was already receiving a message from $\mathcal{P}_b$. Hence, two cut message nets cause an increase of two in the number of messages between $\mathcal{P}_{\text{cur}}$ and the other processor groups.

Algorithm 3 displays the overall RB process in which both the bandwidth and the latency costs are reduced. As inputs, the algorithm takes a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, K (the number of parts to be obtained), and t_s as the cost of the message nets. The partitioning proceeds in a breadth-first manner (lines 2-3). Each hypergraph $\mathcal{H}_{\text{cur}}$ to be bipartitioned does not contain message nets initially (line 4). The sets of send and receive nets $\mathcal{N}_{\text{SND}}$ and $\mathcal{N}_{\text{RCV}}$ are formed via FORM-MESSAGE-NETS (Algorithm 2) (line 6). Then, these message nets are added to $\mathcal{N}_{\text{cur}}$ to obtain $\mathcal{N}^M_{\text{cur}}$ (line 7) and consequently $\mathcal{H}^M_{\text{cur}}$ (line 8). Note that if $\mathcal{H}_{\text{cur}}$ is the root hypergraph $\mathcal{H}^0_0$ of the RB tree, no message nets can be added since there is only a single processor group at this point (line 10). The current hypergraph is bipartitioned with the BIPARTITION function to obtain the vertex parts $\mathcal{V}_L$ and $\mathcal{V}_R$ (lines 9 and 11). The call to BIPARTITION can be realized with any hypergraph partitioning tool; it is a call to obtain only a two-way partition. New hypergraphs $\mathcal{H}_L = \mathcal{H}^{\ell+1}_{2i}$ and $\mathcal{H}_R = \mathcal{H}^{\ell+1}_{2i+1}$ are formed as described in Section 4.3.1 (lines 12-13). $\mathcal{N}_L$ and $\mathcal{N}_R$ do not contain any message nets since these nets rely on the most recent partitioning information and thus need to be introduced from

Algorithm 3: RB-BANDWIDTH-LATENCY

```
Require: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), K, t_s$
 1: $\mathcal{H}_0^0 = \mathcal{H}$
    ▷ RB in breadth-first order
 2: for $\ell = 0$ to $\log_2 K - 1$ do
 3:    for $i = 0$ to $2^\ell - 1$ do
 4:       Let $\mathcal{H}_{\text{cur}} = (\mathcal{V}_{\text{cur}}, \mathcal{N}_{\text{cur}})$ denote $\mathcal{H}_i^\ell = (\mathcal{V}_i^\ell, \mathcal{N}_i^\ell)$
 5:       if $\ell > 0$ then
 6:          $\mathcal{N}_{\text{SND}}, \mathcal{N}_{\text{RCV}}$ = FORM-MESSAGE-NETS($\mathcal{H}, \mathcal{V}_{\text{cur}}, t_s$)
 7:          $\mathcal{N}_{\text{cur}}^M = \mathcal{N}_{\text{cur}} \cup \mathcal{N}_{\text{SND}} \cup \mathcal{N}_{\text{RCV}}$
 8:          $\mathcal{H}_{\text{cur}}^M = (\mathcal{V}_{\text{cur}}, \mathcal{N}_{\text{cur}}^M)$
 9:          $\Pi$ = BIPARTITION($\mathcal{H}_{\text{cur}}^M$)        ▷ $\Pi = \{\mathcal{V}_L, \mathcal{V}_R\}$
10:       else
11:          $\Pi$ = BIPARTITION($\mathcal{H}_{\text{cur}}$)        ▷ $\Pi = \{\mathcal{V}_L, \mathcal{V}_R\}$
          ▷ Subhypergraphs $\mathcal{H}_L$ and $\mathcal{H}_R$ contain only volume nets
12:       Form $\mathcal{H}_L = \mathcal{H}_{2i}^{\ell+1} = (\mathcal{V}_L, \mathcal{N}_L)$ of $\mathcal{H}_{\text{cur}}$ induced by $\mathcal{V}_L$
13:       Form $\mathcal{H}_R = \mathcal{H}_{2i+1}^{\ell+1} = (\mathcal{V}_R, \mathcal{N}_R)$ of $\mathcal{H}_{\text{cur}}$ induced by $\mathcal{V}_R$
```

scratch just prior to bipartitioning $\mathcal{H}_{2i}^{\ell+1}$ and $\mathcal{H}_{2i+1}^{\ell+1}$. Notice that at any step of the RB, among all the leaf hypergraphs, only the current $\mathcal{H}_{\text{cur}}$ is subject to the addition of the message nets, whereas other hypergraphs remain intact.

4.4.4 Running Time Analysis

We consider the cost of adding message nets in the ℓth level of the RB tree produced in partitioning $\mathcal{H}$ into K parts, $0<\ell<\log_2 K$. Recall that Algorithm 2 utilizes $Pins(\cdot)$ and $Nets(\cdot)$ functions on the *original hypergraph* $\mathcal{H}=(\mathcal{V}, \mathcal{N})$.

In the addition of the send nets, for each vertex v_j in the ℓth level, $Pins(n_j)$ is visited once, where $src(n_j)=v_j$ (line 5 in Algorithm 2). Observe that each net in $\mathcal{N}$ is visited only once since n_j is retrieved only for v_j. Updates related to a send net (lines 7-12) can be performed in $O(1)$ time. Hence, each pin of $\mathcal{H}$ is processed exactly once, making the cost of formation and addition of send nets $O(p)$ in the ℓth level, where p is the number of pins in $\mathcal{H}$.

In the addition of the receive nets, for each vertex v_j in the ℓth level, $Nets(v_j)$ is visited once (line 13 in Algorithm 2). Updates related to a receive net (lines 16-21) can also be performed in $O(1)$ time. Hence, each pin of $\mathcal{H}$ is again processed exactly once, making the cost of formation and addition of receive nets $O(p)$ in the ℓth level.

Therefore, the cost of adding message nets in a single level of RB is $O(p)$, which results in the overall cost of $O(p \log_2 K)$ for adding message nets. The solution of the partitioning problem with the addition of message nets is likely to be more expensive compared to partitioning of the original hypergraph.

4.5 Extensions

4.5.1 Encoding Messages for A_{POST}

We now describe how to apply the proposed model to A_{POST}. In $\mathcal{H}_F$, we define a function $dest : \mathcal{N} \rightarrow \mathcal{V}$ to determine the responsibility of the fold operation on each o_j, similar to the definition of src for expand operations in $\mathcal{H}_E$. In partitioning $\mathcal{H}_F$, a send net s_k connects the vertices corresponding to the tasks that produce intermediate results to be sent to $\mathcal{P}_k$:

$$\begin{aligned} Pins(s_k) = \{v_j \in \mathcal{V}_{\text{cur}} :\ & v_j \in Pins(n) \text{ and} \\ & dest(n) \in \mathcal{V}_{k \neq \text{cur}}\}. \end{aligned} \tag{4.7}$$

A receive net r_k connects the vertices corresponding to the output items for which the intermediate results need to be received from $\mathcal{P}_k$:

$$\begin{aligned} Pins(r_k) = \{v_j \in \mathcal{V}_{\text{cur}} :\ & dest(n_j) = v_j \text{ and} \\ & Pins(n_j) \cap \mathcal{V}_{k \neq \text{cur}} \neq \emptyset\}. \end{aligned} \tag{4.8}$$

Observe that the formation of a send net for $\mathcal{H}_F$ (4.7) is the same as that of a receive net for $\mathcal{H}_E$ (4.3) and the formation of a receive net for $\mathcal{H}_F$ (4.8) is the same as that of a send net for $\mathcal{H}_E$ (4.2). So, the message nets in $\mathcal{H}_E$ are the dual of the message nets in $\mathcal{H}_F$. Therefore, the correctness and complexity analysis carried out for A_{PRE} are also valid for A_{POST}.

4.5.2 Encoding Messages for $I = O \geq T$

To extend the proposed model to the case $I = O \geq T$ for A_{PRE}, we need a minor change in the formation of the message nets. In this case, a net n_j might be held responsible for multiple expand operations (see *src* definition). To reflect this change, line 4 of Algorithm 2 needs to be executed for each net n_j such that $src(n_j) = v_j$. The meaning of a message net does not change. The complexity of adding message nets is still $O(p \log_2 K)$ since each net n_j is again retrieved exactly once, uniquely by v_j. A similar discussion holds for A_{POST} by extending the definition of *dest*.

4.6 Adjusting Message Net Costs

As described in the previous section, the cost of a volume net is expressed in terms of t_w, while the cost of a message net is assigned to t_s. In practice t_s is hundreds of multiple of t_w. For this reason, the message nets have more significance than the volume nets in reducing the cutsize. In this situation, the number of cut volume nets is likely to be higher compared to a method in which only the volume nets exist. Apart from increasing bandwidth costs, this also has the side effect of causing an increase in latency costs. This is because the volume nets that remain in the cut in a specific bipartitioning signify the messages between the processor groups in the RB-subtree rooted at this specific bipartitioning. Especially in the early levels of the RB tree, this effect becomes more crucial since the subtrees rooted at these levels are bigger and contain more processors than their latter counterparts. In order to reduce the number of cut volume nets accordingly, we increase their significance in early levels of RB tree with a simple net cost assignment scheme.

Instead of utilizing a fixed cost of t_s in all levels of the RB tree, we propose a level-based cost assignment scheme for the message nets. In this scheme we reduce the costs of message nets in early levels of the RB tree so that the number of cut volume nets reduces. We utilize a linear function to determine these costs so that they increase from $t_s/(\log_2 K - 1)$ to t_s, from the first level to the $(\log_2 K - 1)$th level. Then the cost of a send net s_k or receive net r_k added in the ℓth level is given by

$$c(s_k) = c(r_k) = \ell \frac{t_s}{\log_2 K - 1}. \tag{4.9}$$

Note that t_s is divided by $\log_2 K - 1$ since message nets can not be added in the 0th and the $\log_2 K$th levels.

4.7 Experiments

4.7.1 Setup

For evaluation, we target the parallelization of an A_{PRE} application: 1D row-parallel SpMV. We model this application with hypergraph $\mathcal{H}_E$ as described in Section 4.3.2. We compare two schemes for the partitioning of $\mathcal{H}_E$:

- HP: The standard hypergraph partitioning model in which only the bandwidth cost is minimized (Section 4.3.2). In this scheme, $\mathcal{H}_E$ contains only the volume nets.
- HP-L: The proposed hypergraph partitioning model in which the bandwidth and the latency costs are reduced simultaneously (Section 4.4). In this scheme, $\mathcal{H}_E$ contains both the volume and the message nets.

Both HP and HP-L utilize recursive bipartitioning. We tested these schemes for $K \in \{128, 256, 512, 1024, 2048\}$ processors.

The two schemes are evaluated on 30 square matrices from the UFL Sparse Matrix Collection [79]. Table 4.1 displays the properties of these matrices. We

Table 4.1: Properties of test matrices.

name	kind	#rows/cols	#nonzeros
d_pretok	2D/3D	182,730	1,641,672
turon_m	2D/3D	189,924	1,690,876
cop20k_A	2D/3D	121,192	2,624,331
torso3	2D/3D	259,156	4,429,042
mono_500Hz	acoustics	169,410	5,036,288
memchip	circuit simulation	2,707,524	14,810,202
Freescale1	circuit simulation	3,428,755	18,920,347
circuit5M_dc	circuit simulation	3,523,317	19,194,193
rajat31	circuit simulation	4,690,002	20,316,253
laminar_duct3D	computational fluid dynamics	67,173	3,833,077
StocF-1465	computational fluid dynamics	1,465,137	21,005,389
web-Google	directed graph	916,428	5,105,039
in-2004	directed graph	1,382,908	16,917,053
eu-2005	directed graph	862,664	19,235,140
cage14	directed graph	1,505,785	27,130,349
mac_econ_fwd500	economic	206,500	1,273,389
gsm_106857	electromagnetics	589,446	21,758,924
pre2	frequency-domain circuit simulation	659,033	5,959,282
kkt_power	optimization	2,063,494	14,612,663
bcsstk31	structural	35,588	1,181,416
engine	structural	143,571	4,706,073
shipsec8	structural	114,919	6,653,399
Transport	structural	1,602,111	23,500,731
CO	theoretical/quantum chemistry	221,119	7,666,057
598a	undirected graph	110,971	1,483,868
m14b	undirected graph	214,765	3,358,036
roadNet-CA	undirected graph	1,971,281	5,533,214
great-britain_osm	undirected graph	7,733,822	16,313,034
germany_osm	undirected graph	11,548,845	24,738,362
debr	undirected graph sequence	1,048,576	4,194,298

consider square matrices since the proposed scheme aims at obtaining a conformal partition on the input and output items. The numbers of nonzeros in the test matrices range from 1.2M to 27.1M. This dataset contains small matrices (e.g., `bcsstk31`, `mac_econ_fwd500`, etc.) for which the latency cost is expected to be more important.

The partitionings for both HP and HP-L are performed with the hypergraph partitioner PaToH [42]. Specifically, for each bipartitioning, we call `PaToH_Part` function of PaToH (lines 7 and 9 of Algorithm 3). The partitioning imbalance is set to 10%. Since PaToH contains randomized algorithms, we run each partitioning instance five times and report the average results of these runs.

We present the communication statistics for partitionings obtained via HP and HP-L and the corresponding parallel SpMV times. We used parallel SpMV of PETSc toolkit [69] on a Blue Gene/Q system. A node on this system consists of 16 PowerPC A2 processors with 1.6 GHz clock frequency and 16 GB memory. The nodes are connected by a 5D torus chip-to-chip network.

4.7.2 Message Net Costs

Recall that in our model, the volume nets are assigned the cost of t_w (transfer time of a single word) and the message nets are assigned the cost of t_s (startup time). Hereinafter, both the bandwidth and the latency costs are expressed in terms of t_w for the sake of presentation. Hence, the costs of volume nets are unit whereas the costs of message nets are t_s/t_w. The message net costs are denoted with *mnc* to simplify the notation. We conducted ping-pong experiments on BlueGene/Q with varying message sizes and the average t_s/t_w ratio was found to be around 200 for the matrices in our dataset.

Compared to HP, HP-L is expected to obtain a higher total volume since HP-L addresses two communication components simultaneously while HP solely optimizes a single component, which is determined by the total volume. We found out that the cost assignment of message nets in HP-L has a crucial impact on the parallel performance. The t_s/t_w ratio varies in practice for different message sizes and depends on the protocol used for transmitting messages as well as the characteristics of the target application which is likely to incur a higher t_w, hence a lower t_s/t_w, than that was found. For this reason, as well as to control the balance between the increase in the volume and the decrease in the message count compared to HP, we tried out different values for *mnc* in HP-L. The tested *mnc* values are 10, 50, 100 and 200. The reason for including smaller *mnc* values is that when the communication cost is dominated by the bandwidth component, utilizing a high *mnc* value has an adverse affect on the parallel performance compared to HP as the volume increase caused by HP-L is more apparent. Hence, small *mnc* values become more preferable in such cases.

4.7.3 Results

Table 4.2 presents the average communication statistics and the parallel SpMV running times of HP-L normalized with respect to those of HP for four *mnc* values and five K values. Each entry at a specific K value is the geometric mean of the normalized results obtained at that K value. The communication statistics are grouped under "volume" and "#messages". Under the "volume" grouping, the column "tot" denotes the total volume of communication and the column "max" denotes the maximum send volume of processors. Under the "#messages" grouping, the column "tot" denotes the total number of messages and the column "max" denotes the maximum number of messages that a processor sends. The PaToH partitioning times and parallel SpMV running times are respectively given under the columns "PaToH part. time" and "parallel SpMV time".

Table 4.2: Communication statistics, PaToH partitioning times and parallel SpMV running times for HP-L normalized with respect to those for HP averaged over 30 matrices.

message net cost	K	volume tot	volume max	#messages tot	#messages max	PaToH part. time	**parallel SpMV time**
10	128	1.08	1.11	0.82	0.87	1.07	**0.956**
	256	1.10	1.16	0.78	0.83	1.13	**0.904**
	512	1.12	1.22	0.75	0.83	1.13	**0.838**
	1024	1.16	1.29	0.73	0.84	1.25	**0.792**
	2048	1.20	1.37	0.71	0.88	1.28	**0.774**
50	128	1.17	1.25	0.65	0.76	1.08	**0.924**
	256	1.25	1.44	0.59	0.70	1.14	**0.846**
	512	1.33	1.57	0.56	0.69	1.21	**0.760**
	1024	1.41	1.69	0.57	0.74	1.24	**0.715**
	2048	1.48	1.85	0.59	0.80	1.33	**0.708**
100	128	1.24	1.43	0.59	0.73	1.09	**0.954**
	256	1.35	1.66	0.53	0.68	1.17	**0.858**
	512	1.45	1.86	0.51	0.68	1.19	**0.768**
	1024	1.54	1.92	0.53	0.71	1.31	**0.706**
	2048	1.61	2.06	0.57	0.80	1.41	**0.707**
200	128	1.33	1.60	0.54	0.72	1.15	**1.031**
	256	1.46	1.87	0.48	0.67	1.19	**0.872**
	512	1.57	2.02	0.49	0.67	1.25	**0.778**
	1024	1.65	2.09	0.52	0.72	1.37	**0.722**
	2048	1.70	2.17	0.57	0.79	1.48	**0.712**

As seen in Table 4.2, HP-L achieves a significant reduction in the latency overhead. HP-L reduces the total number of messages by 18%-29%, 35%-44%, 41%-49% and 43%-52% for *mnc* values of 10, 50, 100 and 200, respectively, compared to HP. This substantial improvement comes at the expense of increased volume as expected. Compared to HP, HP-L increases the total volume by 8%-20%, 17%-48%, 24%-61% and 33%-70% for *mnc* values of 10, 50, 100 and 200, respectively. In other words, HP-L achieves a factor of 1.22-1.41, 1.54-1.79, 1.69-1.96 and 1.75-2.08 reductions in the total number of messages while causing a factor of 1.08-1.20, 1.17-1.48, 1.24-1.61 and 1.33-1.70 increase in the total volume, over HP for *mnc* values of 10, 50, 100 and 200, respectively.

The proposed HP-L scheme achieves significantly lower parallel SpMV running times compared to the HP scheme. As seen in Table 4.2, HP-L achieves 4%-23%, 8%-29%, 5%-29% and −3%-29% lower running times for *mnc* values of 10, 50, 100 and 200, respectively, compared to HP. The lowest average running times for K values of 128, 256, 512, 1024 and 2048 are obtained with *mnc* values of 50, 50, 50, 100 and 100, respectively. Since using a low *mnc* (e.g., 10) does not attribute enough importance to the reduction of the latency cost, the parallel running times attained by this value are generally higher than those of other *mnc* values. Observe that for a specific K value, increasing the *mnc* leads to a decrease in the total message count and an increase in the total volume.

The performance improvement of HP-L over HP increases in terms of parallel SpMV time with increasing K for almost all *mnc* values. For example, for $mnc = 100$, HP-L only achieves a 5% improvement in running time over HP at $K = 128$, whereas at $K = 2048$ this improvement becomes 29%. The effect of reducing total message count becomes more apparent in parallel running time with increasing K since the latency component gets more important at high K values.

When we compare HP and HP-L in terms of partitioning times in Table 4.2, we see that HP-L has higher partitioning overhead as expected. HP-L incurs 7%-28%, 8%-33%, 9%-41% and 15%-48% slower partitionings for *mnc* values of 10, 50, 100 and 200, respectively. Although the formation of message nets is not expensive ($O(p \log_2 K)$), note that the partitioning with HP-L includes the message nets in addition to volume nets, which leads to increased bipartitioning times compared to HP. As K increases, HP-L's partitioning time also increases compared to HP since the number of message nets increases as well.

Table 4.3: Communication statistics and parallel SpMV times/speedups for HP and HP-L for $K = 512$ and $mnc = 50$. Running time is in microseconds.

	tot vol		max vol		tot msg		max msg		running time		speedup	
name	HP	HP-L	HP	HP-L	HP	HP-L	HP	HP-L	HP	HP-L	HP	HP-L
d_pretok	108k	141k	302	451	5.3k	2.8k	19	11	312	232	66	**88**
turon_m	106k	127k	296	434	4.8k	2.5k	16	9	268	210	79	**100**
cop20k_A	142k	167k	411	482	5.2k	3.6k	20	15	542	466	82	**95**
torso3	197k	227k	550	825	4.9k	3.2k	21	15	411	348	112	**132**
mono_500Hz	226k	255k	703	812	6.0k	4.7k	26	26	492	498	**158**	157
memchip	102k	149k	679	705	3.9k	2.2k	51	16	1224	714	188	**322**
Freescale1	85k	151k	473	972	5.4k	2.4k	68	28	1697	1105	176	**271**
circuit5M_dc	84k	145k	386	939	5.5k	2.3k	64	24	1554	1030	196	**295**
rajat31	139k	184k	403	786	2.9k	2.0k	23	20	1010	983	325	**334**
laminar_duct3D	186k	211k	538	694	7.4k	5.9k	26	21	445	387	74	**85**
StocF-1465	581k	613k	1650	2109	6.8k	5.5k	31	29	1062	1048	229	**233**
web-Google	126k	266k	872	2971	37.2k	11.9k	281	231	17385	3492	12	**61**
in-2004	122k	169k	1933	2641	12.8k	6.4k	137	181	15690	4164	17	**64**
eu-2005	338k	408k	4207	6934	18.9k	9.9k	305	351	8375	4849	26	**44**
cage14	3184k	3291k	10528	10922	27.4k	19.7k	125	100	5757	4587	57	**71**
mac_econ_fwd500	124k	160k	335	490	5.6k	3.1k	16	11	252	231	80	**88**
gsm_106857	338k	371k	1161	1400	3.8k	3.1k	16	16	710	748	**521**	495
pre2	245k	261k	1143	1332	6.5k	3.8k	33	26	787	646	100	**122**
kkt_power	662k	684k	2930	3459	7.2k	4.2k	50	29	2072	1554	197	**262**
bcsstk31	62k	76k	223	297	4.3k	3.2k	19	17	283	253	45	**50**
engine	117k	144k	477	671	4.6k	3.1k	32	27	525	496	96	**102**
shipsec8	139k	159k	471	606	4.2k	3.2k	16	15	353	337	170	**178**
Transport	582k	645k	1487	1700	5.3k	3.2k	16	10	782	1230	**313**	199
CO	1044k	1070k	2710	2792	13.4k	10.1k	45	35	906	783	85	**99**
598a	104k	131k	341	529	5.6k	3.6k	29	24	435	382	103	**117**
m14b	158k	190k	596	804	5.3k	3.5k	38	31	630	547	123	**142**
roadNet-CA	35k	67k	138	496	3.1k	1.3k	15	8	848	753	178	**200**
great-britain_osm	26k	59k	168	640	4.1k	1.4k	29	11	1202	1151	509	**532**
germany_osm	33k	74k	242	772	4.2k	1.5k	31	13	1759	1719	603	**617**
debr	505k	899k	1268	2793	68.3k	16.9k	185	74	6047	1747	11	**37**

In Table 4.3, we present the detailed communication statistics and the parallel SpMV running times and speedups of 30 matrices for $K = 512$ and $mnc = 50$. In this table, the actual results of HP and HP-L are presented. The unit of the total volume is one kilo-item whereas the unit of the total number of messages is one kilo-message. The columns under "running time" denote the parallel SpMV running time in microseconds and the columns under "speedup" denote the speedups.

In 27 out of 30 matrices, HP-L obtains better speedup values than HP. As also observed and discussed for Table 4.2, reducing both the total volume and total message count significantly improves the parallel performance. For example, for `memchip`, HP-L increases the speedup by 71% (from 188 to 322) by reducing the total message count by 45% (from 3.9k to 2.2k). On the other hand, for `gsm_106857`, having a reduction of 18% in the total message count (from 3.8k to 3.1k) by HP-L does not lead to an improved parallel running time.

The reduction in the total number of messages leads to a reduction in the maximum number messages, as observed in Tables 4.2 and 4.3. In a similar manner, the increase in the total volume leads to an increase in the maximum volume. The models that provide an upper bound on the maximum message count usually have two communication phases, in each of which the maximum message count is $\sqrt{K} - 1$. Compared to these models, apart from the scale-free matrices, although our model does not provide such an upper bound, it usually obtains values below this bound, which is approximately $2(\sqrt{512} - 1) \approx 43$ for $K = 512$.

As seen in Table 4.3, a significant reduction in the total message count generally leads to a better performance. There are a couple of basic factors that can be argued to determine whether improving latency cost at the expense of bandwidth cost will result in a better parallel performance. For a partitioning instance, in general, if the average message size is relatively high and/or the maximum message count is relatively low, then it can be said that the bandwidth component dominates the latency component. For example, for `gsm_106857`, the average message size is high and the maximum message count is low compared to the other matrices. Hence, reducing the total message count by 18% does not pay off as the latency component is not worth exploiting.

Figs. 4.9 and 4.10 display the speedup values of 16 matrices for parallel SpMV attained by two schemes for $K \in \{128, 256, 512, 1024, 2048\}$. We do not present any matrices of the directed graph kind in the figure since their efficiencies are very low. HP-L-10, HP-L-50, HP-L-100 and HP-L-200 respectively denote the HP-L scheme with *mnc* values of 10, 50, 100 and 200. For certain matrices, HP-L drastically changes the scalability by scaling up the parallel SpMV while HP scales down. This is observed for the matrices in the circuit simulation category, and the matrices `pre2` and `kkt_power`. For these matrices, reducing the latency overhead seems to be more important than reducing the bandwidth overhead. HP already exhibits good scalability for the matrices such as `m14b`, `great-britain_osm` and `Transport`. Reducing latency cost for these matrices pays off as HP-L further improves their scalability. HP and HP-L attain comparable scalability for `gsm_106857`, `StocF-1465` and `shipsec8`. The latency costs for these matrices are a minor component of their overall communication cost. Among the HP-L schemes, the scalability of HP-L-10 resembles that of HP the most since HP-L-10 attributes less importance to reducing the latency overhead compared to the other HP-L schemes. For `CO` and `mono_500Hz`, both HP and HP-L scale down after a certain number of processors. Nonetheless, HP-L still improves the parallel SpMV running time.

The HP-L schemes with the four different *mnc* values generally exhibit different parallel performance, as seen in Figs. 4.9 and 4.10. For any K value, increasing the *mnc* further decreases the message count and further increases the total volume (see Table 4.2). How this affects the parallel SpMV running time depends on the communication requirements of the respective partitioning instance. It may pay off to use a high *mnc* value to further reduce the latency overhead (as is the case for `rajat31`) or doing so may worsen the parallel running time (as is the case for `StocF-1465`).

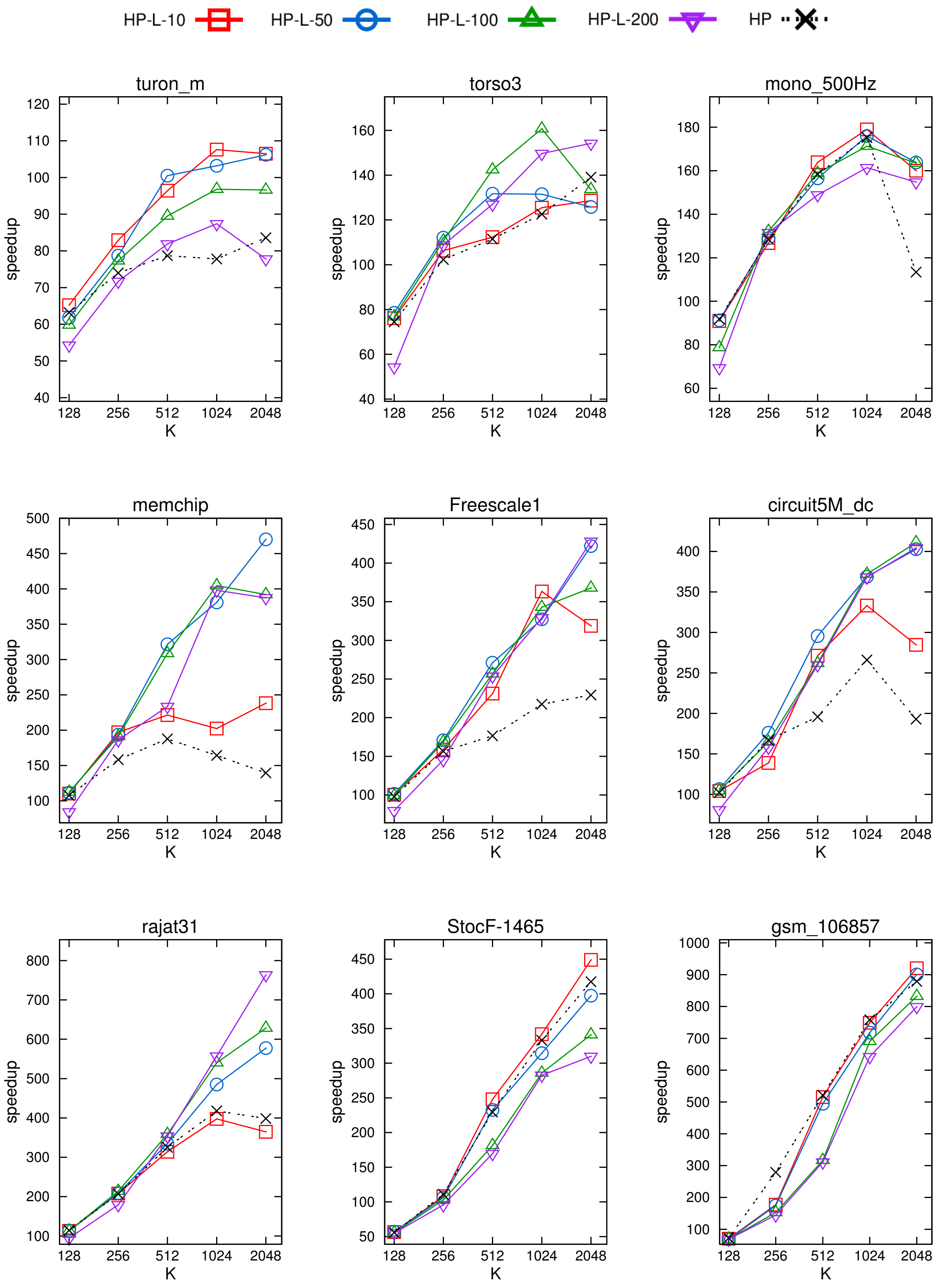


Figure 4.9: Speedup curves for 9 matrices (Part 1 of 2).

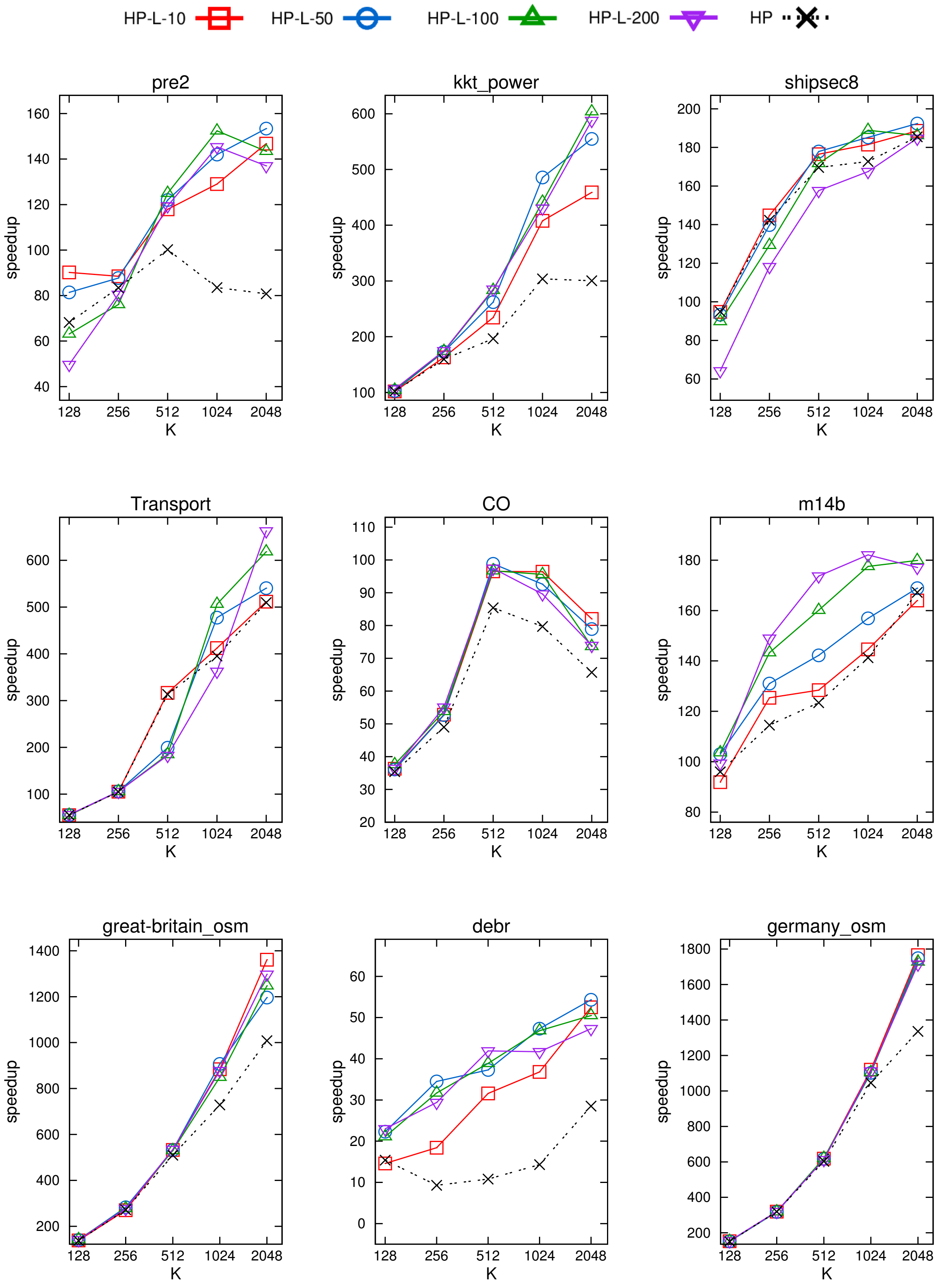


Figure 4.10: Speedup curves for 9 matrices (Part 2 of 2).

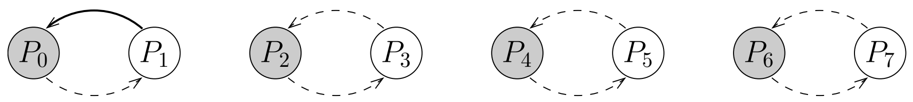
Chapter 5

Iterative improvement heuristics for reducing volume in the embedding scheme

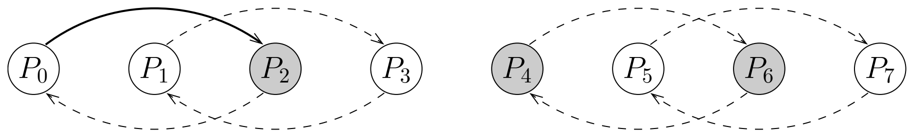
Another idea to reduce the latency overhead is to embed point-to-point communications into the pattern of the collective communication. This idea is proposed in [5]. Embedding has the disadvantage of increasing communication volume substantially. In this chapter, we propose heuristics to reduce this increased volume overhead. These heuristics respect the bounds provided by the embedding scheme. In other words, the mapping heuristics proposed in this section strive to reduce bandwidth costs while keeping the latency costs intact. An example of embedding is given in Fig. 5.1. $SendSet(P_k)$ function in the figure simply denotes the processors that P_k sends a message to.

There are two kinds of parallelization in the work [5]. The first is the conventional parallelization of the conjugate gradient solver and the second one is the one proposed by the authors that allow the idea of embedding point-to-point communications of SpMV into the collective communications of inner products in the solver. For details of these two kinds of parallelization, we refer the reader to [5].

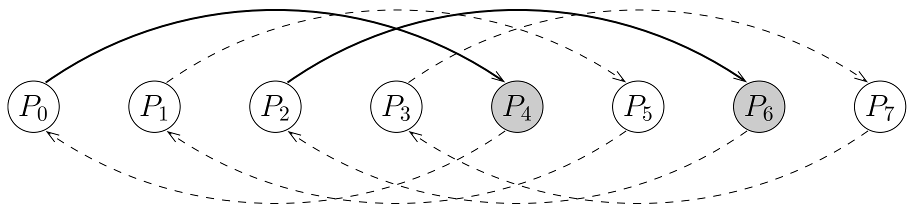
see [5] for the original work



Step 1: $SendSet(P_1) = \{P_0, P_2, P_4, P_6\}$



Step 2: $SendSet(P_1) = \{P_2, P_4, P_6\}$



Step 3: $SendSet(P_1) = \{P_4, P_6\}$

Figure 5.1: Embedding messages of P_1 into `ALL-REDUCE` for $SendSet(P_1) = \{P_0, P_2, P_4, P_6\}$.

As in [5], we assume a 1D rowwise partition of matrix A for the parallelization of the conjugate gradient solver to solve $q = Ap$.

5.1 Part to processor mapping

Consider a given row partition $\mathcal{R} = \{R_1, R_2, \ldots, R_K\}$ of matrix A and a set of processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$, where the number of row parts is equal to the number of processors. In row partition $\mathcal{R}$, a column c_i is said to be a coupling column if more than one row parts contain at least one nonzero in c_i. Observe that for parallelization of $q = Ap$ in the conjugate gradient solver, in the conventional parallel algorithm, only the input vector (i.e., p) entries associated with the coupling columns necessitate communication, whereas in the parallel algorithm with embedding, only the output vector (i.e., q) entries associated with such columns necessitate communication. Let $\Lambda(c_i)$ denote the set of row parts that contain at least one nonzero in c_i. Without loss of generality, let row r_i be assigned to row part $R_k \in \mathcal{R}$. Now consider an identity mapping function $M : \mathcal{R} \rightarrow \mathcal{P}$ where the row block R_k is mapped to processor $P_{M(k)=k}$, for $1 \leq k \leq K$. Then, due to the symmetric partitioning requirement, q_i is assigned to P_k. Besides, since all diagonal entries are nonzero, we have $R_k \in \Lambda(c_i)$. So, $\{P_l : R_l \in \Lambda(c_i) \text{ and } R_l \neq R_k\}$ denotes the set of processors to which q_i should be sent (multicast) by processor P_k. Thus, $|\Lambda(c_i)| - 1$ gives the volume of communication that is incurred by coupling column c_i. We define the set of processors that participate in the communication of q_i as $ProcSet(q_i) = \{P_l : R_l \in \Lambda(c_i)\}$, which includes the owner P_k of q_i as well, hence, $|ProcSet(q_i)| = |\Lambda(c_i)|$. For any arbitrary mapping, this definition becomes

$$ProcSet(q_i) = \{P_l : \exists R_m \in \Lambda(c_i) \text{ s.t. } M(m) = l\}. \tag{5.1}$$

For conventional parallelization, the total message volume is independent of the mapping, i.e., different part-to-processor mappings incur the same amount of message volume, which is:

$$ComVol(\mathcal{R}) = \sum_{q_i:\, c_i \in cc(\mathcal{R})} (|\Lambda(q_i)| - 1), \tag{5.2}$$

where $cc(\mathcal{R})$ denotes the set of coupling columns of the row partition $\mathcal{R}$. However, in the parallelization scheme with embedding, the total message volume depends on the mapping of parts to processors due to forwarding of vector elements in the embedding process.

As an example, in Fig. 5.1, assume that two parts R_a and R_b are mapped to processors P_1 and P_6, respectively, and P_1 needs to send vector elements to P_6. These vector elements need to be forwarded in two steps, increasing communication volume compared to a single P2P communication between these two processors. However, if R_b were mapped to P_0 (or P_3, or P_5), these vector elements would not be forwarded, and they would incur no extra communication volume at all.

Based on this observation, the objective of mapping should be to minimize the extra communication volume due to forwarding. In other words, we should try to keep the pairs of processors that communicate a large number of vector elements *close* to each other. The closeness here is defined in terms of the communication pattern of the `ALL-REDUCE` algorithm described in the previous section.

We now introduce assumptions and notations used to discuss the formulation adopted for computing total cost of a mapping M for a given row partition $\mathcal{R}$. We assume that the number of processors is an exact power of two (i.e., $K = 2^D$) and the processors are organized as a virtual D-dimensional hypercube topology H as the utilized `ALL-REDUCE` algorithm implies. In H, each processor is represented by a D-bit binary number. A dimension d is defined as the set of 2^{D-1} virtual bidirectional communication links connecting pairs of neighboring processors of which only differ in bit position d. Tearing along dimension d is defined as halving H_d into two disjoint $(d-1)$-dimensional subcubes, H_d^0 and H_d^1, such that their respective processors are connected along dimension d in a one-to-one manner. In this view, step d of the `ALL-REDUCE` algorithm can be considered as $K/2$ processors exchanging information along the $K/2$ virtual links of dimension d for $d = 0, 1, \ldots, D-1$.

For any coupling column c_i, the cost of communicating vector entry q_i is defined to be the number of `ALL-REDUCE` steps in which q_i is communicated. If q_i is communicated in step d of the `ALL-REDUCE` operation, we define the corresponding communication cost of q_i in this step as one, regardless of how many times q_i is communicated in this step because all communications of q_i in a single step are handled concurrently. Thus, in step d, q_i incurs a cost of one if the processors in $ProcSet(q_i)$ are scattered across different subcubes H_d^0 and H_d^1 of the tearing along dimension d. Otherwise, q_i does not incur any communication which corresponds to the case where all processors in $ProcSet(q_i)$ are confined to the same subcube of the tearing. Note that this latter case can be identified as all processors having the same value (either 0 or 1) at bit position d in their D-bit binary representations. Therefore, the communication cost of $q_i \in cc(\mathcal{R})$ is defined as:

$$cost(q_i) = \sum_{d=0}^{D-1} \Big(\bigwedge P_{k,d} \quad \otimes \quad \bigvee P_{k,d} \Big)_{P_k \in ProcSet(q_i)}. \tag{5.3}$$

In this equation, $P_{k,d}$ denotes the dth bit of P_k in its D-bit binary representation, and $\wedge$, $\otimes$, and $\vee$ denote the logical "AND", "XOR", and "OR" operators, respectively. Then, the total cost of mapping M is simply given by:

$$cost(M) = \sum_{q_i :\, c_i \in cc(\mathcal{R})} cost(q_i). \tag{5.4}$$

We should note here that the cost definition in (5.4) captures an objective that is in between the total and concurrent communication overheads. In fact, it represents the sum of the number of distinct q-vector entries communicated in each step of the ALL-REDUCE algorithm. In other words, (5.3) corresponds to the total concurrent cost associated with forwarding q_i to the processors that it should be sent. The total message volume could easily be captured by counting exactly how many times q_i is communicated in each step of ALL-REDUCE instead of counting it only once. We preferred this cost definition in order to capture some form of concurrency in the optimization objective.

In order to find a good mapping, we propose two Kernighan-Lin (KL) [96] based heuristics. As typical in KL-type algorithms, the proposed heuristics start from a given initial mapping and perform a number of moves in the search space to improve the given mapping. For both heuristics, the move operator is defined as the swapping of the processor mapping of two row blocks. The gain of a swap operation is given as the reduction in the total communication cost of the mapping, as defined in (5.4). Both heuristics perform a number of passes till their improvement rate drops below a predetermined threshold. In each iteration of a single pass, the swap operation with the highest gain is chosen, tentatively performed and the respective row blocks are locked to prevent any further operations on them in the same pass. Best swaps with negative gains are also allowed to be selected in order to enable hill-climbing. At the end of a pass, a prefix of the performed swap operations with the highest cumulative cost improvement is selected as the resultant mapping to be used in the following pass.

Although both heuristics utilize the same move operators, they differ in their move neighborhood definitions. The first heuristic, KLF, considers the full move neighborhood with all possible $K(K-1)/2$ swaps, whereas the second heuristic, KLR, restricts the neighborhood over the adjacent processors of the virtual hypercube topology. In other words, KLR allows swapping only the parts at the processors that directly communicate in the ALL-REDUCE algorithm. Restricting the swap neighborhood has the following advantages over searching the full neighborhood: (i) Initial number of swaps reduces from $K(K-1)/2$ to $K \lg K/2$, (ii) gain updates performed after a swap operation become confined to the swap operations that are in the same dimension as the performed swap, and (iii) gain updates performed after a swap operation can be done in *constant* time. The obvious disadvantage of `KLR` is the possible loss in the quality of the generated mappings compared to `KLF`. However, as we show in the experiments, this loss is very small, only around 10%. In this sense, there is a tradeoff between running time and mapping quality, where `KLR` favors time and `KLF` favors quality.

In this chapter, we only focus on describing the `KLR` heuristic because of its significantly better running time performance and algorithmic elegance.

5.2 Restricted Move Neighborhood Mapping Heuristic: KLR

5.2.1 Definitions

Before going into the details of the KL heuristic, we describe the notation used in Algorithms 4, 5 , 6 and 7:

- $\mathcal{P}=\{P_1, \ldots, P_K\}$: Set of K processors.
- $\mathcal{R}=\{R_1, \ldots, R_K\}$: K-way row partition of matrix A.
- H: Virtual hypercube topology in which the processors are considered to be organized for the communication pattern of `ALL-REDUCE`. H_d^0 and H_d^1 denote the tearing along dimension d.
- $M : \mathcal{R} \rightarrow \mathcal{P}$: Initial mapping describing which parts are assigned to which processors. Note that the KL heuristic improves a *given* mapping. We also use IM notation to indicate the inverse mapping.

- q: Output vector of sparse-matrix vector multiply.
- $ProcSet(q_i)$: The set of processors that participate in communication of q_i.
- $ComSet(P_k)$: The set of vector elements that P_k sends/receives.
- $P_{k,d}$: The dth bit of P_k (corresponding bit value in dth step of `ALL-REDUCE`). $P_{k,d} \in \{0, 1\}$. $\overline{P_{k,d}}$ denotes the complement of it.
- δ: $\delta(q_i, H_d^0)$ denotes the number of processors in H_d^0 that participate in communicating q_i and is equal to $|\Delta(q_i, H_d^0)|$ where

  $$\Delta(q_i, H_d^0) = \{IM(P_k) : P_k \in H_d^0 \text{ and } P_k \in ComSet(q_i)\}.$$

  Similarly, $\delta(q_i, H_d^1)$ denotes the number of processors in H_d^1 that participate in communicating q_i and is equal to $|\Delta(q_i, H_d^1)|$ where

  $$\Delta(q_i, H_d^1) = \{IM(P_k) : P_k \in H_d^1 \text{ and } P_k \in ComSet(q_i)\}.$$

  Note that if $P_{k,d} = 0$, then $P_k \in H_d^0$, and if $P_{k,d} = 1$, then $P_k \in H_d^1$. The δ values will come in handy in explaining criticality conditions and analyzing complexity.
- gain(R_a, R_b): The gain (the reduction in the cost function) of swapping $R_a = IM(P_k)$ and $R_b = IM(P_l)$.

5.2.2 Motivation

For the processors that participate in the communication of q_i, $\delta(q_i, H_d^0) = |ProcSet(q_i) \cap H_d^0|$ and $\delta(q_i, H_d^1) = |ProcSet(q_i) \cap H_d^1|$ denote the number of processors in the two subcubes of the tearing along dimension d of the virtual hypercube topology. Obviously, $\delta(q_i, H_d^0) + \delta(q_i, H_d^1) = |ProcSet(q_i)|$ for $1 \leq d \leq \lg K$. Observe that q_i does not incur any communication along dimension d only if all the processors that participate in the communication of q_i are confined to the same subcube, i.e., either $\delta(q_i, H_d^0) = 0$ or $\delta(q_i, H_d^1) = 0$. This implies that the virtual links between the two subcubes are not utilized in step d of `ALL-REDUCE`. Therefore, in dimension d, q_i is said to be critical

$$\begin{aligned}
&\text{to } H_d^0 \text{ if } \delta(q_i, H_d^0) = 1 \text{ or } \delta(q_i, H_d^1) = 0,\\
&\text{to } H_d^1 \text{ if } \delta(q_i, H_d^1) = 1 \text{ or } \delta(q_i, H_d^0) = 0.
\end{aligned} \tag{5.5}$$

The criticality of q_i to H_d^0 (H_d^1) signifies that swapping a row block R_a mapped to $M(R_a) = P_k \in ProcSet(q_i)$ at subcube H_d^0 (H_d^1) with row block R_b mapped to P_k's neighbor processor $M(R_b) = P_l \in ProcSet(q_i)$ at subcube H_d^1 (H_d^0) along dimension d will either save q_i incurring communication overhead or cause q_i to be communicated in that particular dimension. Note that since only the coupling columns are considered in the mapping process, we have $\delta(q_i, H_d^0) + \delta(q_i, H_d^1) \geq 2$ for any dimension d.

Consider the swap of two row blocks R_a and R_b that are currently mapped to two processors P_k and P_l that are neighbors over dimension d. This swap can be interpreted as two reassignment operations: the reassignment of R_a from P_k to P_l, and the reassignment of R_b from P_l to P_k. For a single vector entry $q_i \in ComSet(P_k)$, where $ComSet(P_k)$ denotes the vector entries that P_k sends/receives, the gain of reassigning R_a from P_k to P_l is computed as:

$$g(R_a : P_k \xrightarrow{d} P_l, q_i) = \begin{cases} 1, & \text{if } \delta(q_i, H_d^{\overline{P_{k,d}}}) = 1, \\ -1, & \text{if } \delta(q_i, H_d^{P_{k,d}}) = 0, \\ 0, & \text{otherwise.} \end{cases} \tag{5.6}$$

Here, for the tearing along dimension d, $H_d^{P_{k,d}}$ denotes the subcube that P_k belongs to, whereas $H_d^{\overline{P_{k,d}}}$ denotes the other subcube. So, the reassignment in the first case avoids communication of q_i along dimension d by confining the set of processors in $ProcSet(q_i)$ to the same subcube $H_d^{\overline{P_{k,d}}}$, which were previously scattered between subcubes $H_d^{P_{k,d}}$ and $H_d^{\overline{P_{k,d}}}$. On the other hand, the reassignment in the second case incurs communication of q_i along dimension d by scattering the set of processors in $ProcSet(q_i)$ to both subcubes $H_d^{P_{k,d}}$ and $H_d^{\overline{P_{k,d}}}$, which were previously confined to subcube $H_d^{P_{k,d}}$. The gain of reassigning R_b from P_l to P_k can be computed in a similar manner by replacing P_k with P_l and vice versa.

Hence, the total gain of swapping the row parts R_a and R_b along dimension d becomes:

$$
\begin{aligned}
g(R_a \longleftrightarrow R_b : P_k \overset{d}{\longleftrightarrow} P_l) = \\
& \sum_{q_i \in ComSet(P_k)} g(R_a : P_k \overset{d}{\rightarrow} P_l, q_i) \; + \\
& \sum_{q_j \in ComSet(P_l)} g(R_b : P_l \overset{d}{\rightarrow} P_k, q_j) \; - \\
& 2\, |\{q_h : q_h \in ComSet(P_k) \cap ComSet(P_l) \text{ and} \\
& \qquad \delta(q_h, H_d^0) = \delta(q_h, H_d^1) = 1\}| \qquad (5.7)
\end{aligned}
$$

The last term of (5.7) handles the corner case for q_h entries that are shared by both processors P_k and P_l where P_k and P_l belong to different subcubes in the dth dimension, and they are the sole processors communicating q_h in d.

After performing a swap operation, it may be necessary to update gain values of certain swaps. Whenever a transition to/from one of the criticality conditions in (5.5) occur for any q_i, a gain update procedure is triggered. Observe that a swap operation in step d only affects the subcubes in this dimension by changing $\delta(q_i, H_d^0)$ and $\delta(q_i, H_d^1)$ values of q_i. Thus, since the subcubes in other dimensions remain intact, there will be no transitions to/from the criticality conditions of any vector entry in any dimension other than d. This enables the gain update operations after the swap to be confined to the swaps that are in the same dimension as the performed swap operation. More importantly, the number of criticality transitions for q_i in a single dimension becomes constant, and this enables gain updates after a swap operation to be performed in *constant* time. With them being constant, the complexity of a single pass of `KLR` heuristic becomes $O(V(\lg K)^2)$, where $V = ComVol(\mathcal{R}) + |cc(\mathcal{R})|$.

5.2.3 Base `KLR`

Algorithm 4 presents the proposed KL-type heuristic with restricted move neighborhood, `KLR`. The input to this algorithm is the processor set $\mathcal{P}$, the part set $\mathcal{R}$,

Algorithm 4: KLR

```
Input: $\mathcal{P}=\{P_1,\ldots,P_K\}, \mathcal{R}=\{R_1,\ldots,R_K\}, H, q,$
$M : \mathcal{R} \rightarrow \mathcal{P}$
1  for each $q_i$ do
2      $ProcSet(q_i)$ = Set of processors that will send/recv $q_i$
3  for each $P_k$ do
4      $ComSet(P_k)$ = The $q$ entries that $P_k$ will send/recv
5  Initialize $\delta$ fields for each $q_i$ at each dimension $d$
6  $cost = best\text{-}cost =$ INITIAL-COST($M$)
7  repeat
8      $G =$ INIT-GAINS($\lg K, IM, \delta$)
9      $Q =$ BUILD-HEAP($G$)
10     while $Q \neq \emptyset$ do
11         gain($R_a, R_b$) = EXTRACT-MAX($Q$)
12         Remove all gains related with $R_a$ and $R_b$ from $Q$
13         Let $R_a = IM(P_k)$ and $R_b = IM(P_l)$
14         SWAP($P_k, P_l, H, IM, \delta$)
15         $cost = cost -$gain($R_a, R_b$)
16         if $cost < best\text{-}cost$ then
17             $best\text{-}cost = cost$
18     Rollback to the state where best-cost is encountered
19 until there are no more passes
```

virtual hypercube topology H, vector q, and the initial mapping M. The inverse mapping IM is also used in the algorithm. As in typical KL-based heuristics, the algorithm initializes gains and maintains an efficient structure to keep them (i.e., heap), then performs a number of successive swaps, and rolls back to the state where the best cost is encountered when there are no more swaps to perform. This is repeated for a number predetermined passes or till the solution cannot be improved anymore. Algorithm 4 starts by initializing necessary structures in lines 1–5: $ProcSet$, $ComSet$ and δ. Then it computes the initial cost of the given mapping M in line 6 according to the objective in (5.8). In lines 8–9, gains are computed and stored in a priority queue. We maintain swap operations keyed according to their gain values and use a binary heap as the priority queue implementation. In each iteration, the algorithm extracts the swap with the highest gain from heap and removes all related swaps from the heap accordingly in lines 11–12 (i.e., locks the extracted parts in order to avoid swap operations on them in

Algorithm 5: `INIT-GAINS`

```
Input: $D, H, IM : \mathcal{P} \rightarrow \mathcal{R}, \delta$
1  for $d = 0 \rightarrow D - 1$ do
2      for each $(P_k, P_l)$ pair in $d$ do
3          $R_a = IM(P_k)$, $R_b = IM(P_l)$
           ▷ Process vector entries that $P_k$ sends/receives
4          for $q_i \in ComSet(P_k)$ and $q_i \notin ComSet(P_l)$ do
5              if $\delta(q_i, H_d^{P_{k,d}}) = 1$ then
6                  gain$(R_a, R_b)$ = gain$(R_a, R_b) + 1$
7              if $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 0$ then
8                  gain$(R_a, R_b)$ = gain$(R_a, R_b) - 1$
           ▷ Process vector entries that $P_l$ sends/receives
9          for $q_i \in ComSet(P_l)$ and $q_i \notin ComSet(P_k)$ do
10             if $\delta(q_i, H_d^{P_{l,d}}) = 1$ then
11                 gain$(R_a, R_b)$ = gain$(R_a, R_b) + 1$
12             if $\delta(q_i, H_d^{\overline{P_{l,d}}}) = 0$ then
13                 gain$(R_a, R_b)$ = gain$(R_a, R_b) - 1$
```

the same pass). Then, these two parts are swapped and the cost of the mapping is updated in lines 13–17. This process is repeated till there remain no swaps in the heap. After all swaps are exhausted, in line 18, a rollback operation is performed to the state where the best cost is encountered.

5.2.4 Gain Initialization

Crucial to every KL-type heuristic are the gain initialization and swap routines which are given in Algorithm 5 (`INIT-GAINS`) and Algorithms 6 and 6 (`SWAP`), respectively. The `INIT-GAINS` algorithm in Algorithm 5 computes gain of swapping each pair of parts that are mapped to processors which directly communicate in `ALL-REDUCE` steps. Consider a pair of processors P_k and P_l that communicate over dimension d, and assume parts R_a and R_b are mapped to these processors, respectively. Recall that a swap operation actually consists of two move operations. Lines 4–8 in Algorithm 5 compute the gain due to moving R_a from P_k to P_l while lines 9–13 compute the gain due to moving R_b from P_l to P_k, which as a whole correspond to computing gain of swapping R_a and R_b.

In the algorithm, the vector entries that P_k sends/receives are checked to see whether moving R_a from P_k to P_l incurs communication or not. For each $q_i \in ComSet(P_k)$, if the communication of q_i is only due to mapping of R_a to P_k in this step ($\delta(q_i, H_d^{P_{k,d}}) = 1$), then moving R_a to P_l saves us from communicating this vector entry, thus the gain is incremented (lines 5–6). On the other hand, if q_i does not incur any communication while R_a is mapped to P_k but it will incur when it is moved to P_l ($\delta(q_i, \overline{H_d^{P_{k,d}}}) = 0$), then the gain is decremented (lines 7–8) (see criticality conditions in Equation (5)). In a similar manner, the vector entries that P_l sends/receives are checked and gain of swapping R_a and R_b is updated accordingly. Note that if P_k and P_l participate in communication of same vector element q_i, swapping corresponding parts assigned to them will not change anything in terms of communication volume, i.e., this will lead to no gain since this is a symmetric operation and both processors will still communicate q_i in this dimension due to bidirectional characteristic of `ALL-REDUCE` algorithm. Such cases are excluded in lines 4 and 9 with statements $q_i \notin ComSet(P_l)$ and $q_i \notin ComSet(P_k)$, respectively.

5.2.5 Gain Updates

Swapping two parts may necessitate gain updates for certain swaps. Given a swap on two parts $R_a = IM(P_k)$ and $R_b = IM(P_l)$, the `SWAP` in Algorithms 6 and 7 performs necessary gain and data structure updates to realize it. Recall that each swap operation belongs to a certain dimension of the `ALL-REDUCE` algorithm (line 1). Note that due to restricted neighborhood definition, we can associate each swap operation with a certain step of the `ALL-REDUCE` algorithm (line 1) since a part at a processor can only be swapped with the parts at the processor that processor communicates with. Such a restriction confines the scope of gain updates due to a swap over dimension d entirely to the swaps that belong to the same dimension.

In Algorithm 6, lines 2–19 update the swap gains and data structures due to

Algorithm 6: SWAP: Update gains due to moving $R_a = IM(P_k)$.

```
   Input: $P_k, P_l, H, IM : \mathcal{P} \rightarrow \mathcal{R}, \delta$
1  Let dimension of this swap operation be $d$
2  for $q_i \in ComSet(P_k)$ and $q_i \notin ComSet(P_l)$ do
3      if $\delta(q_i, H_d^{P_{k,d}}) = 2$ or $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 0$ then $\triangleright$ $q_i$ is critical to the
       subcube of $\mathbf{R_a} = \mathbf{IM}(\mathbf{P_k})$
4          for $P_m \in ProcSet(q_i)$ and $P_{m,d} = P_{k,d}$ do
5              $R_c = IM(P_m)$
6              Let $P_n$ be the pair of $P_m$ in dimension $d$ with $P_n \notin ProcSet(q_i)$
               and $R_d = IM(P_n)$
7              if $\delta(q_i, H_d^{P_{k,d}}) = 2$ then gain$(R_c, R_d)$ = gain$(R_c, R_d) + 1$
8              if $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 0$ then gain$(R_c, R_d)$ = gain$(R_c, R_d) + 1$
9      if $\delta(q_i, H_d^{P_{k,d}}) = 1$ or $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 1$ then $\triangleright$ $q_i$ is critical to the
       complementary subcube of $\mathbf{R_a} = \mathbf{IM}(\mathbf{P_k})$
10         for $P_m \in ProcSet(q_i)$ and $P_{m,d} = \overline{P_{k,d}}$ do
11             $R_c = IM(P_m)$
12             Let $P_n$ be the pair of $P_m$ in dimension $d$ with $P_n \notin ProcSet(q_i)$
               and $R_d = IM(P_n)$
13             if $\delta(q_i, H_d^{P_{k,d}}) = 1$ then gain$(R_c, R_d)$ = gain$(R_c, R_d) - 1$
14             if $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 1$ then gain$(R_c, R_d)$ = gain$(R_c, R_d) - 1$
15     $\delta(q_i, H_d^{P_{k,d}}) = \delta(q_i, H_d^{P_{k,d}}) - 1$
16     $\delta(q_i, H_d^{\overline{P_{k,d}}}) = \delta(q_i, H_d^{\overline{P_{k,d}}}) + 1$
17     $ProcSet(q_i) = ProcSet(q_i) \cup \{P_l\} - \{P_k\}$
18     $ComSet(P_k) = ComSet(P_k) - \{q_i\}$
19     $ComSet(P_l) = ComSet(P_l) \cup \{q_i\}$
```

moving R_a from P_k to P_l while lines 2–19 in Algorithm 6 update the swap gains and data structures due to moving R_b from P_l to P_k, which as a whole correspond to updating swap gains and data structures due to swapping R_a and R_b. Each swap gain update operation requires either INCREASE-KEY or a DECREASE-KEY operation on the priority queue. To realize movement of $R_a = IM(P_k)$, we may need to update the swap gains of the parts mapped to processors that are involved in communicating a vector entry with P_k, i.e., for all $R_c = IM(P_m)$ such that $ComSet(P_k) \cap ComSet(P_m) \neq \emptyset$. Only the vector entries in $ComSet(P_k)$ that become critical cause gain updates. If a vector entry q_i becomes critical after moving R_a, it implies that the further swap operations that will be performed on

Algorithm 7: `SWAP`: Update gains due to moving $R_b = IM(P_l)$.

```
    Input: $P_k, P_l, H, IM : \mathcal{P} \to \mathcal{R}, \delta$
1   Let dimension of this swap operation be $d$
2   for $q_i \in ComSet(P_l)$ and $q_i \notin ComSet(P_k)$ do
3       if $\delta(q_i, H_d^{P_{l,d}}) = 2$ or $\delta(q_i, H_d^{\overline{P_{l,d}}}) = 0$ then ▷ $q_i$ is critical to the subcube of $\mathbf{R_b = IM(P_l)}$
4           for $P_m \in ProcSet(q_i)$ and $P_{m,d} = P_{l,d}$ do
5               $R_c = IM(P_m)$
6               Let $P_n$ be the pair of $P_m$ in dimension $d$ with $P_n \notin ProcSet(q_i)$ and $R_d = IM(P_n)$
7               if $\delta(q_i, H_d^{P_{l,d}}) = 2$ then gain$(R_c, R_d) =$ gain$(R_c, R_d) + 1$
8               if $\delta(q_i, H_d^{\overline{P_{l,d}}}) = 0$ then gain$(R_c, R_d) =$ gain$(R_c, R_d) + 1$
9       if $\delta(q_i, H_d^{P_{l,d}}) = 1$ or $\delta(q_i, H_d^{\overline{P_{l,d}}}) = 1$ then ▷ $q_i$ is critical to the complementary subcube of $\mathbf{R_b = IM(P_l)}$
10          for $P_m \in ProcSet(q_i)$ and $P_{m,d} = \overline{P_{l,d}}$ do
11              $R_c = IM(P_m)$
12              Let $P_n$ be the pair of $P_m$ in dimension $d$ with $P_n \notin ProcSet(q_i)$ and $R_d = IM(P_n)$
13              if $\delta(q_i, H_d^{P_{l,d}}) = 1$ then gain$(R_c, R_d) =$ gain$(R_c, R_d) - 1$
14              if $\delta(q_i, H_d^{\overline{P_{l,d}}}) = 1$ then gain$(R_c, R_d) =$ gain$(R_c, R_d) - 1$
15      $\delta(q_i, H_d^{P_{l,d}}) = \delta(q_i, H_d^{P_{l,d}}) - 1$
16      $\delta(q_i, H_d^{\overline{P_{l,d}}}) = \delta(q_i, H_d^{\overline{P_{l,d}}}) + 1$
17      $ProcSet(q_i) = ProcSet(q_i) \cup \{P_k\} - \{P_l\}$
18      $ComSet(P_l) = ComSet(P_l) - \{q_i\}$
19      $ComSet(P_k) = ComSet(P_k) \cup \{q_i\}$
```

the parts mapped to processors in $ProcSet(q_i)$ may increase or decrease the communication volume. The crucial observation about criticality is that, in a specific dimension d, if the processors in $ProcSet(q_i)$ are all in $\Delta(q_i, H_d^0)$ $(\delta(q_i, H_d^0) = 0)$ or are all in $\Delta(q_i, H_d^1)$ $(\delta(q_i, H_d^1) = 0)$, then q_i does incur no communication overhead in this step; otherwise it incurs communication overhead. The transitions between these two states are described by the criticality of vector elements and occur only at most for a fixed number of times in a single pass. If movement of R_a makes q_i critical, the swap gains in dimension d that contain the parts at processors in $ProcSet(q_i)$ need to be updated. Regarding this, four criticality cases are grouped into two conditions and checked in lines 3 and 9.

For Algorithm 6, in the former condition where $\delta(q_i, H_d^{P_{k,d}}) = 2$ or $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 0$, the movement of R_a from P_k to P_l provides an opportunity to reduce the communication volume for q_i with the further swap of parts mapped to the processors in the same subcube as P_k ($P_m \in ProcSet(q_i)$ and $P_{m,d} = P_{k,d}$). Thus, the gain of swapping $R_c = IM(P_m)$ and $R_d = IM(P_n)$, where P_m and P_n are a pair of processors communicating over dimension d, is incremented in lines 7–8.

In the latter condition where $\delta(q_i, H_d^{P_{k,d}}) = 1$ or $\delta(q_i, H_d^{\overline{P_{k,d}}}) = 1$, the movement of R_a from P_k to P_l may cause an increase in the communication volume for q_i with the further swap of parts mapped to the processors in the complementary subcube of P_k ($P_m \in ProcSet(q_i)$ and $P_{m,d} = \overline{P_{k,d}}$). Thus, the gain of swapping $R_c = IM(P_m)$ and $R_d = IM(P_n)$, where P_m and P_n are a pair of processors communicating over dimension d, is decremented in lines 13–14.

Note that because of the bidirectional characteristic of `ALL-REDUCE` algorithm, we again need to exclude the cases where a swap contains the parts mapped to processors that communicate the same vector element (lines 2, 6, and 12). After updating gains due to movement of R_a for the current vector element q_i, necessary fields and structures are updated to reflect the effects of this operation in lines 15–19. The realization of movement of $R_b = IM(P_l)$ is performed in a similar manner in lines 1–19 of Algorithm 7.

5.2.6 Complexity Analysis

Recall that the number of processors is equal to the number of parts K. The total number of processors that participate in communication of all q_i entries is given by

$$\begin{aligned} V &= \sum_{q_i} |ProcSet(q_i)| = \sum_{P_k} |ComSet(P_k)| \\ &= ComVol(\mathcal{R}) + |cc(\mathcal{R})|. \end{aligned} \tag{5.8}$$

We now analyze the running time of a single pass of `KLR` heuristic (Algorithm 4). The initial cost computation in line 8, although its details are not given here, simply traverses the $ProcSet(q_i)$ for each q_i to compute the communication cost of q_i and sums them up to compute the total cost, which takes $O(V)$ time. The gain initialization algorithm `INIT-GAINS` in line 9 (given in Algorithm 5), for dimension d, traverses $ComSet(P_k)$ for each $P_k \in \mathcal{P}$, which is $O(V)$. There are $\lg K$ steps, thus the complexity of gain initialization is $O(V \lg K)$. As mentioned, there are $K \lg K/2$ swaps. The `BUILD-HEAP` in line 10 takes linear time in terms of the number elements the heap is built on, hence it takes $O(K \lg K)$ time. The upper bound on the number of `EXTRACT-MAX` operations in line 12 is $O(K \lg K)$, each of which takes $O(\lg(K \lg K))$ time in worst case, making complexity of all `EXTRACT-MAX` operations $O(K \lg K \times \lg(K \lg K)) = O(K(\lg K)^2)$. The removal of all elements from heap in a single pass in line 13 takes $O(K(\lg K)^2)$ time with a similar argument.

The complexity analysis of gain updates due to swap operations is somewhat complicated because the swap gain updates depend on the criticality of vector elements. The restriction of search space in `KLR` enables the gain update costs to be cheaper compared to the `KLF` that utilizes full search space. To be more specific, in `KLR`, a vector entry q_i becomes critical for a fixed number of times in a single dimension throughout a pass and this allows us to provide an upper bound on the number of gain updates. Assume that we are given two parts R_a and R_b to swap over dimension d that are mapped to processors P_k and P_l, respectively.

In Algorithm 6, if $q_i \in ComSet(P_k)$ and $q_i \notin ComSet(P_l)$, then the for loop in line 2 is executed whereas the for loop in line 2 of Algorithm 7 is not executed. On the other hand, if $q_i \in ComSet(P_l)$ and $q_i \notin ComSet(P_k)$, then the for loop in line 2 of Algorithm 7 is executed whereas the for loop in line 2 of Algorithm 6 is not executed. Note that if q_i is in both $ComSet(P_k)$ and $ComSet(P_l)$, then no gain updates are necessary since nothing will change after performing this swap, i.e., neither of the for loops in line 2 of Algorithms 6 and 7 are executed. Only the criticality conditions regarding movement of R_a are analyzed in the discussion below but a similar discussion also applies for the movement of R_b. The criticality conditions (Algorithms 6 and 7) for q_i at dimension d due to movement of R_a are given as follows:

- If movement of $R_a = IM(P_k)$ is from $\Delta(q_i, H_d^0)$ to $\Delta(q_i, H_d^1)$, q_i is critical if $\delta(q_i, H_d^0) = 2$, $\delta(q_i, H_d^1) = 0$, $\delta(q_i, H_d^0) = 1$, or $\delta(q_i, H_d^1) = 1$.

- If movement of $R_a = IM(P_k)$ is from $\Delta(q_i, H_d^1)$ to $\Delta(q_i, H_d^0)$, q_i is critical if $\delta(q_i, H_d^1) = 2$, $\delta(q_i, H_d^0) = 0$, $\delta(q_i, H_d^1) = 1$, or $\delta(q_i, H_d^0) = 1$.

Hence, it can be said that whenever $\delta(q_i, H_d^0) \leq 2$ or $\delta(q_i, H_d^1) \leq 2$, q_i is critical and its *ProcSet* needs to be processed for updating gains. Note that the movement of R_a decreases $\delta(q_i, H_d^{P_{k,d}})$ by one and increases $\delta(q_i, H_d^{\overline{P_{k,d}}})$ by one. Moreover, we cannot perform a swap operation on a part that is already swapped in that pass (locking). A series of swaps that affect q_i in dimension d can make it critical at most four times. The movements from/to $\Delta(q_i, H_d^0)$ that makes q_i critical are given by the transitions: $\delta(q_i, H_d^0) = 2 \rightarrow 1$, $\delta(q_i, H_d^0) = 1 \rightarrow 0$, $\delta(q_i, H_d^0) = 0 \rightarrow 1$, or $\delta(q_i, H_d^0) = 1 \rightarrow 2$. A similar argument also applies for $\delta(q_i, H_d^1)$, which makes q_i critical four more times, thus making q_i critical in dimension d a total of eight times. In fact, a more detailed analysis reveals that q_i can be critical at most four times due to overlapping of some conditions and incrementing of $\delta(q_i, H_d^{\overline{P_{k,d}}})$ besides decrementing of $\delta(q_i, H_d^{P_{k,d}})$. If q_i becomes critical, its *ProcSet* needs to be processed, which for all vector entries account for V number of elements in the worst case. Therefore, in a single pass for a single dimension d, there can be $O(V)$ gain updates each of which incurs an `INCREASE-KEY` or a `DECREASE-KEY` operation on the heap, making the complexity of a single step $O(V \lg (K \lg K)) = O(V \lg K)$. There exist $\lg K$ dimensions, thus the total complexity due to gain updates is $O(V (\lg K)^2)$.

To sum up, the initial cost computation is $O(V)$, the gain initialization is $V \lg K$, building heap is $O(K \lg K)$, the heap operations as a whole take $O(K(\lg K)^2)$, and the cost of updating gains is $O(V(\lg K)^2)$. Thus, the complexity of a single pass is $O(V + V \lg K + K \lg K + K(\lg K)^2 + V(\lg K)^2) = O(V(\lg K)^2 + K(\lg K)^2)$. And since generally $V > K$, $O(V(\lg K)^2 + K(\lg K)^2) = O(V(\lg K)^2)$.

5.3 Experiments

5.3.1 Experimental Framework

Four schemes are tested in the experiments: CONV, EMB, EMB-KLF and EMB-KLR. CONV refers to the conventional parallelization scheme described in [5]. EMB refers to the embedded communication scheme described in [5]. EMB-KLF and EMB-KLR refer to the parallelization schemes that uses embedding and part-to-processor mapping. Hereafter, we will use notation EMB* to refer to these three embedded schemes. In all four schemes, row-parallel SpMV algorithm is utilized, where the row partitions are obtained using the hypergraph partitioning tool PaToH on the column-net model [42] with default parameters. This model aims at minimizing total communication volume under the computational load balancing constraint. The load imbalance for all schemes is set to 10%. CONV and EMB rely on random row-part-to-processor mapping. EMB-KLF and EMB-KLR utilize the `KLF` and `KLR` row-part-to-processor mapping heuristics described in Section 5.1.

The number of passes for `KLF` and `KLR` is set to 10 and 20, respectively. Although lower number of passes could be used for these heuristics, we opted to keep them high to improve the mapping quality to a greater extent. In fact, a few number of passes would have been sufficient for `KLF` as it searches the full move neighborhood, whereas $\lg K$ passes would have been sufficient for `KLR` as it restricts the move neighborhood to the particular steps of the `ALL-REDUCE`.

Table 5.1 displays the properties of 16 structurally symmetric matrices collected from University of Florida Sparse Matrix Collection [79]. Matrices are sorted with respect to their nonzero counts.

The tested schemes are developed using the parallel SpMxV library developed by our group [97]. To obtain a partition of matrix A, we use the column-net hypergraph partitioning model [42]. This model aims to minimize the total communication volume under the constraint of maintaining computational loads of

Table 5.1: Test matrices and their properties.

	Number of		Nonzeros per row/col		
Matrix	rows/cols	nonzeros	avg	min	max
bcsstk25	15,439	252,241	16.34	2	59
ncvxbqp1	50,000	349,968	7.00	2	9
tandem-dual	94,069	460,493	4.90	2	5
finan512	74,752	596,992	7.99	3	55
cbuckle	13,681	676,515	49.45	26	600
cyl6	13,681	714,241	52.21	36	721
copter2	55,476	759,952	13.70	4	45
Andrews	60,000	760,154	12.67	9	36
pli	22,695	1,350,309	59.50	11	108
pcrystk03	24,696	1,751,178	70.91	24	81
598a	110,971	1,483,868	13.37	5	26
opt1	15,449	1,930,655	124.97	44	243
wave	156,317	2,118,662	13.55	3	44
pkuskt07	16,860	2,418,804	143.46	39	267
kkt-power	2,063,494	12,771,361	7.28	2	96
crankseg-2	63,838	14,148,858	221.64	48	3423

the processors. The number of iterations in the parallel CG is set to 10,000, and the average iteration time is reported in the results. Each scheme is run three times and their average running times are used to report the speedup curves.

5.3.2 Mapping Performance Analysis

Table 5.2 compares the `KLF` and `KLR` heuristics in terms of preprocessing time and mapping cost (computed according to (5.4)). The mapping times in the table are normalized with respect to the partitioning times of PaToH. For each instance, first, a row partition of the input matrix is computed using PaToH, and a random part-to-processor mapping is generated. Then, the `KLF` and `KLR` heuristics are applied separately on this initial solution to obtain two different mapping results. The improvement rates obtained using these heuristics are reported separately as

Table 5.2: Performance comparison of mapping heuristics `KLF` and `KLR` averaged over 16 matrices.

	mapping time normalized wrt partitioning time		% improvement in mapping cost wrt random map.	
K	KLF	KLR	KLF	KLR
16	0.02	0.05	39.4	32.7
32	0.11	0.10	44.1	39.2
64	0.42	0.14	46.5	41.9
128	1.45	0.24	45.9	41.0
256	4.71	0.44	47.8	42.9
512	13.24	0.61	46.5	41.1
1024	42.35	0.67	45.1	40.1
2048	129.64	1.21	41.4	37.9

average over all 16 test matrices.

As Table 5.2 illustrates, `KLF` obtains better mappings than `KLR` because it uses a broader search space. The mappings obtained by `KLR` are marginally worse, only 8%–12% on average. There is a trade-off between running time and mapping quality. The trade-off here actually favors `KLR` since it is orders of magnitude faster than `KLF`, but it generates only slightly worse mappings.

5.3.3 Communication Requirements Assessment

Table 5.3 compares the performance of four parallel schemes in terms of their communication requirements averaged over 16 test matrices. Message counts of CONV include both P2P and collective communication phases. For CONV, the maximum message volume value refers to the maximum volume of communication handled during P2P operations, whereas for EMB* schemes, it refers to the sum of the communication volume values of the processors that handle maximum amount of communication in each step of `ALL-REDUCE`. Since each processor sends/receives a single message in each step of `ALL-REDUCE`, the maximum message

Table 5.3: Communication statistics averaged over 16 matrices.

	message count			message volume							
	CONV		EMB*	total				max			
K	avg	max	max (=avg)	CONV	EMB	EMB-KLF	EMB-KLR	CONV	EMB	EMB-KLF	EMB-KLR
16	9.8	12.8	4	0.522	0.780	0.651	0.719	0.102	0.115	0.103	0.108
32	13.2	19.3	5	0.839	1.509	1.186	1.277	0.083	0.115	0.099	0.108
64	16.1	27.4	6	1.304	2.595	1.951	2.158	0.070	0.114	0.096	0.099
128	19.3	34.5	7	1.986	4.550	3.170	3.453	0.055	0.111	0.086	0.092
256	22.2	43.5	8	2.989	7.901	5.012	5.497	0.047	0.112	0.076	0.078
512	25.2	53.8	9	4.522	14.053	7.981	8.673	0.036	0.114	0.064	0.068
1024	28.2	71.1	10	6.831	25.631	13.118	13.650	0.029	0.116	0.058	0.061
2048	31.3	85.0	11	10.669	49.821	25.211	25.130	0.025	0.122	0.054	0.052

In "message count" column, avg and max denote the average and maximum number of messages, respectively, sent by a single processor. In the "message volume" column, max denotes maximum message volume handled (sent and received) by a single processor. Message volume values are given in terms of number of floating points words and they are scaled by the number of rows/columns of the respective matrices.

volume effectively represents the concurrent communication volume as well.

As seen in Table 5.3, for CONV, maximum message counts are significantly larger than average message counts for each K. This is due to the irregular sparsity patterns of the matrices which incur irregular P2P communications in parallel SpMxV computations. On the other hand, in EMB* schemes, average and maximum message counts are both equal to $\lg K$ for K processors independent of the sparsity pattern of the matrix.

In a parallel algorithm, the message latency overhead is actually determined by the processor that handles *maximum* number of messages. In that sense, as seen in Table 5.3, EMB* schemes perform significantly better than CONV for all K values. For example, for `pkustk07` test matrix, the maximum message counts are $16, 25, 34, 34, 47, 60, 90, 96$ in CONV, while they are only $4, 5, 6, 7, 8, 9, 10, 11$ in embedded schemes, for $K = 16, 32, \ldots, 2048$ processors, respectively. This performance gap between CONV and EMB* schemes increases with increasing number of processors in favor of embedded schemes. For example, with K increasing from 16 to 2048 processors, the maximum message count increases 7.08 times for CONV whereas it only increase 2.75 times for EMB*, on the average.

As expected, EMB* schemes increase both total and maximum communication volumes compared to CONV. Even so, this increase remains rather low, especially for EMB-KLF and EMB-KLR schemes that utilize intelligent mapping heuristics. Besides, this increase also remains considerably low compared to the increase in the message latency overhead of CONV. The message latency overhead of CONV compared to those of EMB* schemes is greater than the communication volume overhead of EMB* schemes compared to that of CONV. For example, at $K = 2048$, CONV incurs 7.73 times the message latency overhead of EMB* while EMB-KLR incurs only 2.34 times the total message volume overhead and 2.63 times the maximum message volume overhead of CONV.

The mapping quality improvement rates of `KLF` and `KLR` (utilized in EMB-KLF and EMB-KLR) are roughly reflected in their reduction of message volume in the actual runs compared to the random mapping (utilized in EMB), especially for $K \geq 256$. For instance, as seen in Table 5.2, for $K = 1024$, the `KLF` and `KLR` improve the cost of the random mapping on the average by 45.1% and 40.1%, respectively. In the actual runs, although not presented explicitly, compared to EMB, EMB-KLF obtains 46.5% less total message volume and 36.7% less maximum message volume, and EMB-KLR obtains 42.0% less total message volume and 32.8% less maximum message volume on the average for $K = 1024$. In that sense, it can be said that the objective used for mapping heuristics serves the purpose of reducing both total and maximum message volume successfully in the actual runs.

The communication cost of parallel SpMV operations mainly depends on the communication cost of the bottleneck processor, which is by large determined by the maximum message count and maximum message volume requirements. As seen in Table 5.3, for all schemes, the maximum message volume requirements tend to decrease with increasing K. On the other hand, for CONV, maximum message counts tend to increase sharply with increasing K, whereas for EMB* schemes, maximum message counts increase very slowly (logarithmic growth) with increasing K. This implies that as the number processors increases, the message latency becomes more and more dominant in the overall communication cost. This fact enables embedded schemes to scale better, which is confirmed by the speedup curves reported in the next section.

Recall that EMB* schemes perform redundant computation due to computational rearrangement. On average, the EMB* schemes perform $0.1\%, 0.2\%, 0.3\%, 0.4\%, 0.6\%, 0.8\%, 1.2\%, 1.7\%$ more computation than CONV per processor for $K = 16$, 32, $\ldots$, 2048, respectively. This computational increase is very low and thus negligible.

5.3.4 Speedup Results

Figs. 5.2 and 5.3 present the speedup curves of four tested schemes. The results obtained on XE6 and BG/Q supercomputers are illustrated with white and gray plots, respectively. These plots are grouped by test matrices for the ease of readability.

In both architectures, all schemes similarly scale up to $K = 64$ or $K = 128$ and then their distinctive characteristics begin to establish themselves with increasing number of processors. On XE6, all embedded schemes scale better than CONV, and EMB-KLF and EMB-KLR scale better than EMB by obtaining roughly the same speedup values. On BG/Q, EMB-KLF and EMB-KLR usually scale better than CONV and EMB, while CONV and EMB can scale better with respect to each other depending on the test matrix. We can say that the effect of message latency is more dominant on XE6, which leads to embedded schemes having better speedup values despite the increased message volume in general. Moreover, the embedded schemes start scaling better at lower K values compared to BG/Q. On the other hand, on BG/Q, this impact is not as dramatic as on XE6 and the effect of increased message volume in embedded schemes on speedup values is more prominent. This is basically due to relatively slow communication on BG/Q, which overshadows the benefits of reducing maximum message counts by making the embedded schemes' performance more sensitive to the increases in message volume. As seen in the plots that belong to BG/Q, EMB-KLF and

EMB-KLR are usually able to obtain better speedup values at relatively higher K values where the message startup costs completely dominate the message volume costs.

Regarding the plots in Figs. 5.2 and 5.3, among 16 matrices, the lowest speedup values and poorest scalability characteristics belong to `Andrews`, `cbuckle` and `cyl6` matrices on both architectures for CONV scheme. They exhibit quite poor scaling performance where the speedup values start deteriorating very early at low K values compared to other test instances. For these matrices, the speedup values of CONV scheme are below 60 on XE6 with 1024 processors, and below 100 on BG/Q with 2048 processors. These three matrices have the highest communication requirements in terms of maximum message counts. The corresponding values are $83, 96, 108, 128$ for `Andrews` matrix, $36, 52, 82, 126$ for `cbuckle` matrix, and $36, 54, 78, 126$ for `cyl6` matrix for $K = 128, 256, 512, 1024$, respectively. This poor performance is basically because of the high latency overhead which becomes the decisive factor in communication and overall execution times with increasing number of processors. On the other hand, observe that the embedded schemes have much better scalability characteristics for these matrices due to their lower latency overheads. Note that sparsity patterns of the matrices, which depend on the application, along with the partitioning process as a whole, determine the communication requirements of the parallel solver.

Speedups on BG/Q are typically higher than XE6 since according to our running time analysis, the computation on XE6 is approximately 8 to 10 times faster than BG/Q. This enables computation to communication ratio to remain high and processors to be computationally intensive even at high K values for BG/Q, thus leading to higher speedups.

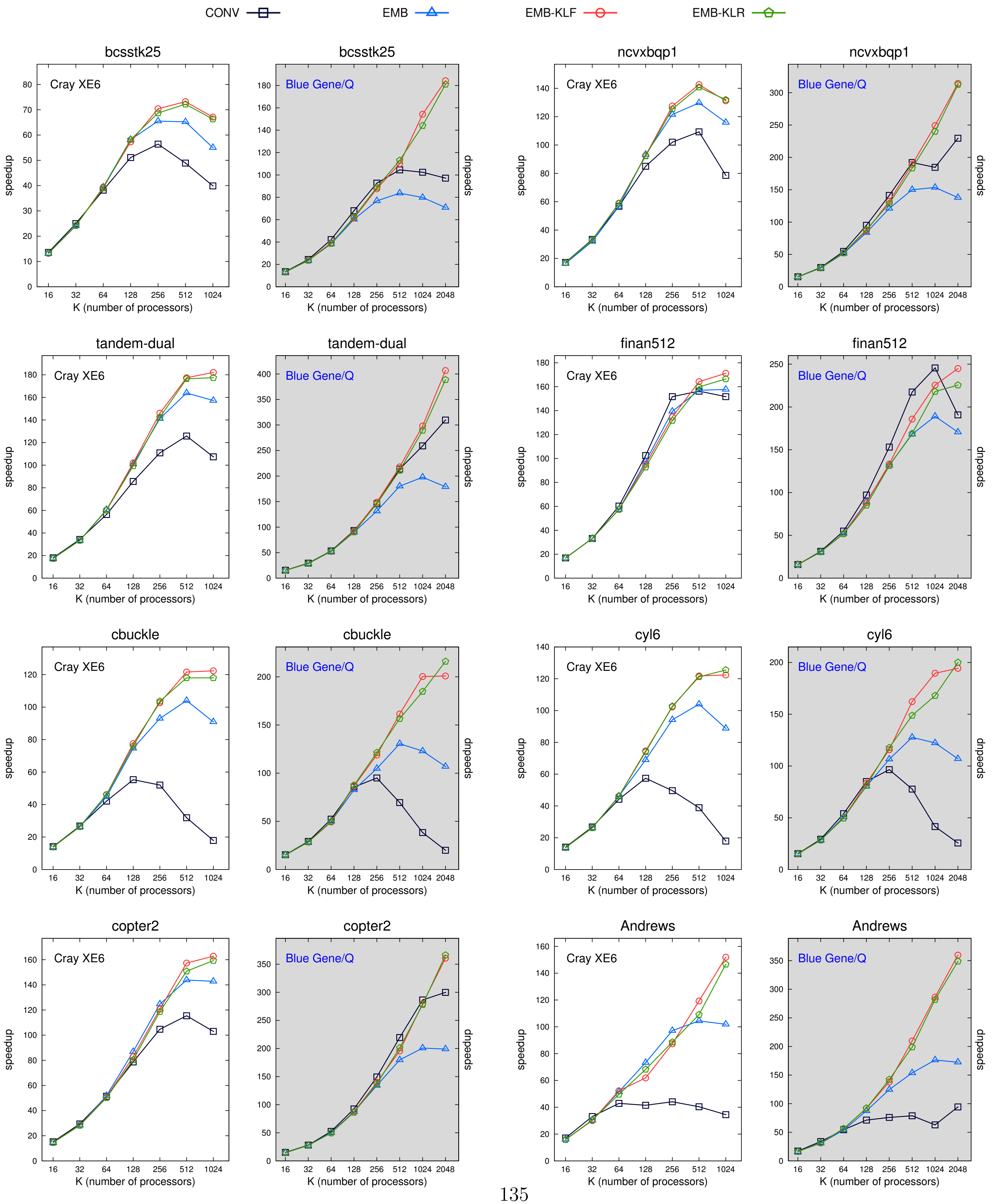


Figure 5.2: Speedup curves (Part 1 of 2).

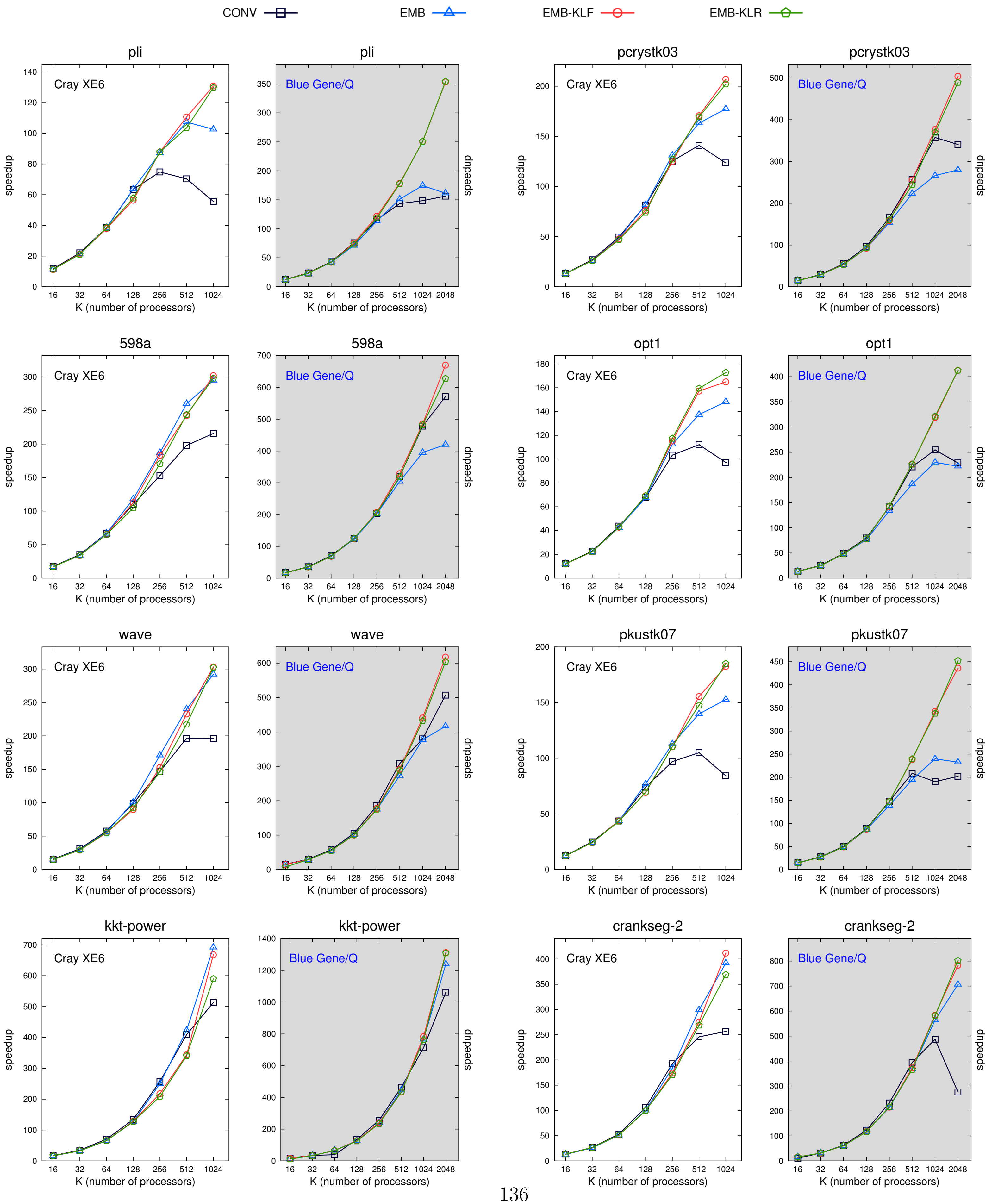


Figure 5.3: Speedup curves (Part 2 of 2).

Chapter 6

Conclusions

In this thesis we provided models and methods to reduce the latency overhead of sparse kernels and operations on distributed memory systems. Our approaches aim at reducing latency besides considering other widely accepted communication cost metrics. We showed the cases for which the latency cost is expected to be crucial for parallel performance and used the obtained results in order to make justifications about the findings of the experiments regarding proposed models and methods.

In Chapter 3, we focused on reducing latency costs of parallel sparse-matrix vector operations by proposing and utilizing several models based on 1D and 2D matrix partitioning. The latency costs are improved by using the communication hypergraph models, where the main motivation is to minimize the number of messages communicated in parallel operations. The described and tested models are realized with CGNE and CGNR solvers via PETSc toolkit on a modern HPC system. We compared a total of eight partitioning models, scaling them up to 8K processors. The results of extensive experiments indicate that along with the bandwidth costs, latency costs should certainly be considered in order to achieve scalable performance. Solely addressing a single of them hurts scalability and leads to poor performance. Our findings indicate that among the partitioning models, the 2D jagged model and its latency-improved version obtain the most promising results. This superior performance is the result of obtaining a good balance between minimizing latency and bandwidth costs.

In Chapter 4, we have proposed a hypergraph partitioning model in order to parallelize certain types of applications with the objective of reducing the total volume and the total number of messages simultaneously. Our model exploits the recursive bipartitioning paradigm and hence provides the flexibility of using any available hypergraph partitioner. The proposed approach provides a better way for capturing the communication requirements of the target parallel applications. The experimental results showed that the better representation of the communication costs in the proposed hypergraph partitioning model led to a reduction of up to 29% in the parallel running time on the average. As an extension to that work, we plan to develop heuristics to dynamically find the best message net cost for each bipartitioning by evaluating the relative importance of the components in the communication cost. We also wish to incorporate the other latency-based communication metrics such as the maximum number of messages communicated by a processor into our model.

In Chapter 5, we have improved the embedding scheme. Embedding allows to avoid all message startup costs of point-to-point communications at the cost of increasing message volume. We presented two iterative-improvement-based heuristics to address this increase in the volume. The results indicate that the message latencies become the determinant factor for the scalability of the solver with increasing number of processors. The results also show that our method, compared to conventional parallelization and the embedding scheme, yields better scalable performance by providing a low value on the number of messages communicated and reducing the increased communication volume. As an extension to that work, we plan to investigate applicability of the embedding and rearrangement scheme to preconditioned iterative solvers. We believe that the proposed embedding scheme is directly applicable to the explicit preconditioning techniques such as approximate inverses or factored approximate inverses [98, 76]. Such preconditioners introduce one or two more sparse matrix vector multiplications into the iterative solver. Since each sparse matrix vector multiplication (either with the coefficient matrix or the preconditioner matrices) is often preced-ed/followed by global reduction operation(s), embedding of P2P communications of sparse matrix vector multiplication operations into collective communication primitives should be viable. However, the computational rearrangement scheme may need modification according to the utilized preconditioning technique and the respective partitioning method used for it.

This thesis validated the claim that the latency cost proves to be as important as the bandwidth cost and should definitely be considered for achieving scalable performance. By proposing models and methods that also consider the latency cost, we showed that the scalability of sparse kernels and operations can be im-proved on modern large-scale systms.

Bibliography

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, “A view of the parallel computing landscape,” *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[2] D. A. Patterson, “Latency lags bandwith,” *Commun. ACM*, vol. 47, pp. 71–75, Oct. 2004.

[3] S. L. Graham, M. Snir, and C. A. Patterson, eds., *Getting Up to Speed, The Future of Supercomputing.* The National Academies Press, 2006.

[4] J. Dongarra and M. A. Heroux, “Toward a New Metric for Ranking High Performance Computing Systems,” Tech. Rep. SAND2013-4744, Sandia National Laboratories, 2013.

[5] 2015 IEEE. Reprinted, with permission, from R. O. Selvitopi and 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*, vol. 26, no. 3, pp. 632-645, 2014, March.

[6] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “Logp: Towards a realistic model of

parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, (New York, NY, USA), pp. 1–12, ACM, 1993.

[8] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation," tech. rep., Santa Barbara, CA, USA, 1995.

[9] M. I. Frank, A. Agarwal, and M. K. Vernon, "Lopc: Modeling contention in parallel algorithms," *SIGPLAN Not.*, vol. 32, pp. 276–287, June 1997.

[10] C. A. Moritz and M. I. Frank, "Logpc: Modeling network contention in message-passing programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 404–415, Apr. 2001.

[11] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Computing*, vol. 20, no. 3, pp. 389 – 398, 1994.

[12] V. Kumar, *Introduction to Parallel Computing.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

[13] C. L. Seitz, "The cosmic cube," *Commun. ACM*, vol. 28, pp. 22–33, Jan. 1985.

[14] A. N. Laboratory, "Mpi chameleon," 1992.

[15] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.

[16] Reprinted from *Parallel Computing*, Vol 57., O. Selvitopi and C. Aykanat, "Reducing latency cost in 2D sparse matrix partitioning models", pp. 1-24, Copyright (2016), with permission from Elsevier.

[17] H. M. Bücker and M. Sauren, "A parallel version of the unsymmetric Lanczos algorithm and its application to QMR," 1996.

[18] T.-X. Gu, X.-Y. Zuo, X.-P. Liu, and P.-L. Li, “An improved parallel hybrid bi-conjugate gradient method suitable for distributed parallel computing,” *J. Comput. Appl. Math.*, vol. 226, pp. 55–65, Apr. 2009.

[19] L. Yang and R. P. Brent, “The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures,” in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pp. 324–328, Oct 2002.

[20] T. P. Collignon and M. B. van Gijzen, “Minimizing synchronization in IDR(s),” *Numerical Linear Algebra with Applications*, vol. 18, no. 5, pp. 805–825, 2011.

[21] A. Chronopoulos and C. Gear, “s-step iterative methods for symmetric linear systems,” *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153 – 168, 1989.

[22] G. Meurant, “Multitasking the conjugate gradient method on the {CRAY} x-mp/48,” *Parallel Computing*, vol. 5, no. 3, pp. 267 – 280, 1987.

[23] E. F. D’Azevedo, V. L. Eijkhout, and C. H. Romine, “Conjugate Gradient Algorithms With Reduced Synchronization Overheads on Distributed Memory Processors,” Tech. Rep. 56, Lapack Working Note, 1993.

[24] Y. Saad, “Practical use of polynomial preconditionings for the conjugate gradient method,” *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 4, pp. 865–881, 1985.

[25] L. T. Yang and R. P. Brent, “The improved BiCG method for large and sparse linear systems on parallel distributed memory architectures,” in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS ’02, (Washington, DC, USA), pp. 315–, IEEE Computer Society, 2002.

[26] Z. Bai, D. Hu, and L. Reichel, “A newton basis GMRES implementation,” *IMA Journal of Numerical Analysis*, vol. 14, no. 4, pp. 563–581, 1994.

[27] A. T. Chronopoulos, "S-step iterative methods for (non)symmetric (in)definite linear systems," *SIAM J. Numer. Anal.*, vol. 28, pp. 1776–1789, Dec. 1991.

[28] A. Chronopoulos and C. Swanson, "Parallel iterative S-step methods for unsymmetric linear systems," *Parallel Computing*, vol. 22, no. 5, pp. 623 – 641, 1996.

[29] M. Hoemmen, *Communication-avoiding Krylov Subspace Methods*. PhD thesis, Berkeley, CA, USA, 2010. AAI3413388.

[30] W. D. Joubert and G. F. Carey, "Parallelizable restarted iterative methods for nonsymmetric linear systems. part I: Theory," *International Journal of Computer Mathematics*, vol. 44, no. 1-4, pp. 243–267, 1992.

[31] E. Carson, N. Knight, and J. Demmel, "Avoiding communication in two-sided krylov subspace methods," Tech. Rep. UCB/EECS-2011-93, EECS Department, University of California, Berkeley, Aug 2011.

[32] L. Grigori and S. Moufawad, "Communication Avoiding ILU0 Preconditioner," Rapport de recherche RR-8266, INRIA, Mar. 2013.

[33] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, "Parallel numerical linear algebra," *Acta Numerica*, vol. 2, pp. 111–197, 1 1993.

[34] 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," *Appl. Numer. Math.*, vol. 18, pp. 441–459, Oct. 1995.

[35] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a conjugate gradient solver with non-blocking collective operations," *Parallel Comput.*, vol. 33, pp. 624–633, Sept. 2007.

[36] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose, "Hiding global communication latency in the GMRES algorithm on massively parallel machines," *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C48–C71, 2013.

[37] P. Ghysels and W. Vanroose, “Hiding global synchronization latency in the preconditioned conjugate gradient algorithm,” *Parallel Computing*, vol. 40, no. 7, pp. 224 – 238, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.

[38] L. C. Mcinnes, B. Smith, H. Zhang, and R. T. Mills, “Hierarchical Krylov and nested Krylov methods for extreme-scale computing,” *Parallel Comput.*, vol. 40, pp. 17–31, Jan. 2014.

[39] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, “Minimizing communication in numerical linear algebra,” *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[40] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, “Avoiding communication in sparse matrix computations,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, April 2008.

[41] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, “Minimizing communication in sparse matrix solvers,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, (New York, NY, USA), pp. 36:1–36:12, ACM, 2009.

[42] U. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, pp. 673–693, Jul 1999.

[43] F. Pellegrini and J. Roman, “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs,” in *High-Performance Computing and Networking* (H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, eds.), vol. 1067 of *Lecture Notes in Computer Science*, pp. 493–498, Springer Berlin Heidelberg, 1996.

[44] B. Vastenhouw and R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication,” *SIAM Rev.*, vol. 47, pp. 67–95, January 2005.

[45] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, pp. 1519–1534, Nov. 2000.

[46] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland, "Massively parallel methods for engineering and science problems," *Commun. ACM*, vol. 37, pp. 30–41, Apr. 1994.

[47] O. C. Martin and S. W. Otto, "Partitioning of unstructured meshes for load balancing," *Concurrency: Practice and Experience*, vol. 7, no. 4, pp. 303–314, 1995.

[48] U. Çatalyurek and C. Aykanat, "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication," in *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '96, (London, UK), pp. 75–86, Springer-Verlag, 1996.

[49] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, pp. 884–896, Aug 1997.

[50] A. Ogielski and W. Aiello, "Sparse matrix computations on parallel processor arrays," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 519–530, 1993.

[51] J. G. Lewis and R. A. van de Geijn, "Distributed memory matrix-vector multiplication and conjugate gradient algorithms," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), pp. 484–492, ACM, 1993.

[52] B. Hendrickson, R. Leland, and S. Plimpton, "An efficient parallel algorithm for matrix-vector multiplication," *International Journal of High Speed Computing*, vol. 7, pp. 73–88, 1995.

[53] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 65:1–65:12, ACM, 2011.

[54] A. Yoo, A. H. Baker, R. Pearce, and V. E. Henson, "A scalable eigensolver for large scale-free graphs using 2D graph partitioning," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 63:1–63:11, ACM, 2011.

[55] U. V. Çatalyürek, *Hypergraph Models for Sparse Matrix Partitioning and Reordering*. PhD thesis, 1999.

[56] U. Çatalyürek and C. Aykanat, "A fine-grain hypergraph model for 2D decomposition of sparse matrices," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, (Washington, DC, USA), pp. 118–, IEEE Computer Society, 2001.

[57] U. Çatalyürek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, (New York, NY, USA), pp. 28–28, ACM, 2001.

[58] B. Uçar and C. Aykanat, "Minimizing communication cost in fine-grain partitioning of sparse matrices," in *Computer and Information Sciences - ISCIS 2003* (A. Yazıcı and C. Şener, eds.), vol. 2869 of *Lecture Notes in Computer Science*, pp. 926–933, Springer Berlin Heidelberg, 2003.

[59] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM J. Sci. Comput.*, vol. 32, pp. 656–683, Feb. 2010.

[60] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 50:1–50:12, ACM, 2013.

[61] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiply," *Electronic Transactions on Numerical Analysis*, vol. 21, pp. 47–65, 2005.

[62] U. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, "UMPA: A Multi-objective, multi-level partitioner for communication minimization," in *Graph Partitioning and Graph Clustering 2012* (D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds.), vol. 588 of *Contemporary Mathematics*, pp. 53–66, AMS, 2013.

[63] B. Uçar and C. Aykanat, "Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies," *SIAM J. Sci. Comput.*, vol. 25, no. 6, pp. 1837–1859, 2004.

[64] R. W. Freund, G. H. Golub, and N. M. Nachtigal, "Iterative solution of linear systems," *Acta Numerica*, vol. 1, pp. 57–100, 1 1992.

[65] N. Nachtigal, S. Reddy, and L. Trefethen, "How fast are nonsymmetric matrix iterations?," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 3, pp. 778–795, 1992.

[66] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[67] H. Elman, "Iterative methods for large, sparse, nonsymmetric systems of linear equations.," *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, vol. 43, pp. 1982 – 1982, 1982/// 1982.

[68] R. Freund and N. Nachtigal, "QMR: a quasi-minimal residual method for non-Hermitian linear systems," *Numerische Mathematik*, vol. 60, no. 1, pp. 315–339, 1991.

[69] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc users manual," Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[70] C. Berge, *Graphs and Hypergraphs.* Elsevier Science Ltd, 1985.

[71] C. Aykanat, A. Pinar, and U. V. Çatalyürek, "Permuting sparse rectangular matrices into block-diagonal form," *SIAM J. Sci. Comput.*, vol. 25, pp. 1860–1879, June 2004.

[72] B. Uçar and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM Rev.*, vol. 49, pp. 595–603, November 2007.

[73] T. Lengauer, *Combinatorial algorithms for integrated circuit layout.* New York, NY, USA: John Wiley & Sons, Inc., 1990.

[74] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 7, pp. 69–79, March 1999.

[75] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint hypergraph partitioning," Tech. Rep. 99-034, University of Minnesota, Dept. Computer Science/Army HPC Research Center, Minneapolis, MN 55455, 1998.

[76] B. Uçar and C. Aykanat, "Partitioning sparse matrices for parallel preconditioned iterative methods," *SIAM J. Sci. Comput.*, vol. 29, pp. 1683–1709, June 2007.

[77] C. Aykanat, B. B. Cambazoglu, and B. Uçar, "Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices," *J. Parallel Distrib. Comput.*, vol. 68, pp. 609–625, May 2008.

[78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* Rockville, MD, USA: Computer Science Press, 1978.

[79] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[80] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, (New York, NY, USA), pp. 343–348, ACM, 1999.

[81] H. D. Simon and S.-H. Teng, "How good is recursive bisection?," *SIAM J. Sci. Comput.*, vol. 18, pp. 1436–1445, Sept. 1997.

[82] E. D. Dolan and J. J. Mor, “Benchmarking optimization software with performance profiles,” *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

[83] 2016 IEEE. Reprinted, with permission, from O. Selvitopi and S. Acer and C. Aykanat, ”A Recursive Hypergraph Bipartitioning Framework for Reducing Bandwidth and Latency Costs Simultaneously”, *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1-1, 2016.

[84] B. Hendrickson, “Graph partitioning and parallel solvers: Has the emperor no clothes? (extended abstract),” in *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, IRREGULAR ’98, (London, UK, UK), pp. 218–225, Springer-Verlag, 1998.

[85] B. Hendrickson and T. G. Kolda, “Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing,” *SIAM J. Sci. Comput.*, vol. 21, pp. 2048–2072, Dec. 1999.

[86] K. Schloegel, G. Karypis, and V. Kumar, “Parallel multilevel algorithms for multi-constraint graph partitioning,” in *Euro-Par 2000 Parallel Processing* (A. Bode, T. Ludwig, W. Karl, and R. Wismller, eds.), vol. 1900 of *Lecture Notes in Computer Science*, pp. 296–310, Springer Berlin Heidelberg, 2000.

[87] D. Pelt and R. Bisseling, “A medium-grain method for fast 2d bipartitioning of sparse matrices,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 529–539, May 2014.

[88] E. Kayaaslan, B. Ucar, and C. Aykanat, “Semi-two-dimensional partitioning for parallel sparse matrix-vector multiplication,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 1125–1134, May 2015.

[89] M. Deveci, K. Kaya, B. Uçar, and Ümit Çatalyürek, “Hypergraph partitioning for multiple communication cost metrics: Model and methods,” *Journal of Parallel and Distributed Computing*, vol. 77, no. 0, pp. 69 – 83, 2015.

[90] O. Fortmeier, H. Bcker, B. F. Auer, and R. Bisseling, "A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications," *Parallel Computing*, vol. 39, no. 8, pp. 319 – 335, 2013.

[91] J. Lewis, D. Payne, and R. van de Geijn, "Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pp. 542–550, May 1994.

[92] A. Buluç and J. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pp. 503–510, Sept 2008.

[93] V. Kuhlemann and P. S. Vassilevski, "Improving the communication pattern in matrix-vector operations for large scale-free graphs by disaggregation," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. S465–S486, 2013.

[94] M. Wang, S. Lim, J. Cong, and M. Sarrafzadeh, "Multi-way partitioning using bi-partition heuristics," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, (New York, NY, USA), pp. 667–, ACM, 2000.

[95] D. A. Padua, ed., *Encyclopedia of Parallel Computing.* Springer, 2011.

[96] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Tech. J.*, vol. 49, pp. 291–307, 1970.

[97] B. Ucar and C. Aykanat, "A library for parallel sparse matrix vector multiplies," Tech. Rep. BU-CE-0506, Bilkent University, 2005.

[98] M. Benzi, J. K. Cullum, and M. Tuma, "Robust approximate inverse preconditioning for the conjugate gradient method," *SIAM J. Sci. Comput.*, vol. 22, pp. 1318–1332, Apr. 2000.