# Load-Balancing and Task Mapping for Exascale Systems

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Mehmet Deveci B.S.,

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Prof. Ümit V. Çatalyürek, Advisor

Prof. P. Sadayappan

Prof. Radu Teodorescu

# Abstract

In many areas of science and engineering, both problem and dataset sizes are increasing rapidly due to technological advances that enable generation and storage of such data. Many of the datasets generated are irregular, in the sense that no immediately observable or predictable pattern exists among data elements. Lack of such patterns makes it difficult to find simple, effective and efficient mechanisms to partition data for scalable parallel processing. Moreover, the hierarchy of the parallel processing systems becomes more complex with the drastic increase in system sizes. Thus, a scalable partitioning and mapping data and tasks to processors has become critical to application performance. In the literature, this problem is referred as the mapping problem.

Computational tasks in scientific computing are modeled either by using spatial information (geometric model) or connectivity based topological information (such as the graph/hypergraph models). The mapping problem is usually solved with a two-phase approach; in the first phase, a load-balanced partitioning of the tasks (data) is found, while in the second phase, each part is mapped to a physical processor. In this work, two-phase mapping problems are investigated. The load-balancing problem is studied in the contexts of both spatial (geometric) partitioning problems and

connectivity-based (hypergraph) partitioning, and task mapping methods are proposed using spatial and connectivity-based (graph) models.

For the applications with spatial information, spatial partitioning is used for the load-balancing purposes. In this work, parallel spatial partitioning methods are investigated on distributed systems, and a parallel multi-jagged algorithm (MJ) is proposed and implemented. In contrast to the traditional recursive coordinate bisection algorithm (RCB), which recursively bisects subdomains, the proposed parallel algorithm does recursive multi-sections in each dimension. It concurrently computes multiple cuts in the dataset, therefore it reduces the recursion depth with respect to RCB. Moreover, it implements wise algorithms to predict the cost of the data movements, and avoids data movements when the gain is predicted not to amortize the data movement cost. This reduces the data movements throughout the execution, and significantly improves the scalability of the algorithm compared to the efficient implementations of RCB. Once the load-balance is achieved, further optimizations are exploited with task mapping using spatial models. Using the proposed geometric partitioner MJ, a geometric mapping algorithm is proposed. The proposed mapping algorithm judiciously assigns the interdependent tasks to the nearby cores, and hence, lowers the distances the messages need to travel and the congestion in the network.

The spatial models have a major drawback as they do not provide an exact model for the actual communication requirements. This becomes problematic especially for the applications that have irregular communication patterns. In such applications,

the load distribution to the processors with minimized the inter-processor communication overhead, which is crucial for parallel application performance, is achieved with connectivity-based models and partitioners adapting these models. In this work, a novel directed hypergraph model that allows the optimization of multiple communication metrics (such as maximum communication volume, maximum number of messages, total number of messages as well as the traditional total communication volume) is proposed. A multi-objective and k-way hypergraph partitioning algorithm that improves multiple communication metrics in a single phase is studied. The proposed model and algorithm are implemented under a software tool, UMPa. In order to improve the scalability, parallelization of hypergraph coarsening and matching methods on multicore architectures are examined. Moreover, since the matrices and hypergraphs are expected to be bigger and more complex in future, efficient compression and sparsification techniques are investigated to reduce processing time. Further optimizations for the communication overhead minimization are investigated by using graph models in task mapping problem for the applications with connectivity information. As the spatial models do not provide an exact model for the communication requirements of unstructured applications, graph mapping algorithms that accurately addresses various communication metrics are proposed and the effect of these communication metrics on the actual communication time is investigated.

To my family

# Acknowledgments

I would like to give my sincerest gratitude to my advisor, Ümit V. Çatalyürek, for his instructive comments, invaluable support, and encouragement during my doctoral study. I would also like to thank to him and his family for their hospitality.

I am grateful to all my advisory committee members, P Sadayappan and Radu Teodorescu, and Christopher Jekeli for spending their time and effort to read and comment on my dissertation.

I would like to thank my collaborators and mentors: Kamer Kaya, Bora Uçar, Abhinav Bhatele, Siva Rajamanickam, and Karen D. Devine for their help and contributions to my research. I would also like to thank my fellow lab members, past and present, for their assistance and support over the years.

Last, but not least, many thanks to my family and my friends Ahmet Erdem Sarıyüce, Emre Tepe and Emel Elvin Yıldız for their help and support during these years.

Thanks all.

# Vita

| | |
|---|---|
| 2010 | B.S. Computer Engineering,<br>Middle East Technical University. |
| 2010 – present | Graduate Research Associate,<br>Computer Science and Engineering,<br>The Ohio State University. |

# Publications

### Research Publications

M. Deveci, K. Kaya, B. Uçar, Ü. V. Çatalyürek, "Fast and High Quality Topology-Aware Task Mapping" 29th IEEE International Parallel and Distributed Processing Symposium, May, 2015

M. Deveci, S. Rajamanickam, K. D. Devine, Ü. V. Çatalyürek, "Multi-jagged: A Scalable Parallel Spatial Partitioning Algorithm" IEEE Transactions on Parallel and Distributed Systems, 2015 (to appear)

M. Deveci, K. Kaya, B. Uçar, Ü. V. Çatalyürek, "Hypergraph partitioning for multiple communication cost metrics: Model and methods" Journal of Parallel and Distributed Computing, vol. 77, pg. 69 - 83, 2015

M. Deveci, Ü. V. Çatalyürek, A. Toland, "mrSNP:Software to detect SNP effects on microRNA bindings", BMC Bioinformatics, vol. 15, pg. 73, 2014

M. Deveci, S. Rajamanickam, V. Leung, K. T. Pedretti, S. L. Olivier, D. P. Bunde, Ü. V. Çatalyürek, K. D. Devine, "Exploiting Geometric Partitioning in Task Mapping for Parallel Computers" 28th IEEE International Parallel and Distributed Processing Symposium, May 2014.

K. Eren, M. Deveci, O. Küçüktunç, Ü. V. Çatalyürek, "A Comparative Analysis of Biclustering Algorithms for Gene Expression Data", Briefings in Bioinformatics, vol. 14, no. 3, pg. 279-292, 2013

M. Deveci, K. Kaya, Ü. V. Çatalyürek, "Hypergraph Sparsification and its Application to Scalable Partitioning", Proc. of 42nd Int'l. Conf. on Parallel Processing, Oct 2013.

M. Deveci, K. Kaya, B. Uçar, Ü. V. Çatalyürek, "A Push-Relabel-based Maximum Cardinality Bipartite Matching Algorithm on GPUs", Proc. of 42nd Int'l. Conf. on Parallel Processing, Oct 2013.

M. Deveci, K. Kaya, B. Uçar, Ü. V. Çatalyürek, "GPU accelerated maximum cardinality matching algorithms for bipartite graphs", Proc. of 19th Int'l. Euro-Par Conf. on Parallel Processing, Aug 2013.

Ü. V. Çatalyürek, M. Deveci, K. Kaya, B. Uçar, "Multithreaded Clustering for Multilevel Hypergraph Partitioning", 26th IEEE International Parallel and Distributed Processing Symposium, Shanghai, China, 2012.

Ü. V. Çatalyürek, M. Deveci, K. Kaya, B. Uçar, "UMPa: A Multi-objective, multilevel partitioner for communication minimization", 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, Feb 2012, Published in Contemporary Mathematics, Vol. 588, Editors D.A. Bader, H. Meyerhenke, P. Sanders, D. Wagner, 2013.

## Fields of Study

Major Field: Computer Science and Engineering

# Table of Contents

# List of Tables

**Table** **Page**

# List of Figures

**Figure** **Page**

# Chapter 1: Introduction

Due to the technological advances in recent decades, the sizes of data in scientific computing is increasing exponentially. This is forcing scientific computing experts to execute their tasks in parallel not only to overcome the data storage limitations of single node computers, but also to achieve higher performance. However, the computational tasks in scientific computing usually have complex interaction patterns. A good distribution of the computational tasks to the processing units is crucially important for a better utilization of the computation and communication units, less energy usage and shorter execution times. In addition, as the number of processors in new supercomputers grows from $O(100K)$ to $O(1M)$ and beyond, the supercomputers get larger and hierarchical networks. Many users concurrently submit jobs with different sizes, and processor allocations become sparse and spread further in the network. This causes the communication messages to travel long routes, and network links to be congested due to the heavy traffic. Therefore not only a good partitioning of the computational tasks, but also a good assignment of them to the

supercomputers' cores is crucial to obtain a high performance. This problem is called *Mapping Problem* in the literature.

The computational tasks in scientific computing can be represented using different models. Graph/hypergraph models use the connectivity-based information,whereas the geometric models use the spatial information. In the literature, the mapping problem is solved using either model, usually with a two-phase approach [26,27,49,89]. In the first phase, a load balanced partition of the tasks is found, then, in the second phase the obtained parts are mapped to cores of a supercomputer. On the other hand, one-phase approaches [68, 115, 116, 143] solve the partitioning and mapping problems concurrently. In this work, our focus is on the two-phase approaches.

The applications using geometric models are those for which the geometric locality is important, such as particle simulations, crash simulations, and parallel volume rendering. For example, particle simulations simulate dynamic system of particles under the influence of the gravitational forces, which are calculated by using the locations of the particles [92, 122]. Crash simulations simulate the transformation of a vehicle's kinetic energy into deformation energy in order to evaluate the safety of the vehicle [32]. They use the geometric information of the elements for collision detection. Similarly, by using the geometry information, volume rendering applications calculate the projection of a 3D dataset on 2D display [98]. Such applications model their data using the geometric models, and most of the time there is no connectivity information on hand. When such applications are run in parallel, they require mapping methods to

preserve the data locality in order to reduce either the inter-processor communications or the data duplications. Therefore, mapping methods that are to be used by these applications need to adapt the geometric models.

On the other hand, the applications such as linear solvers, preconditioners, electrical circuit simulations, finite element methods have more irregular interactions between their tasks [40, 74, 77, 147]. They model their data using connectivity-based models, since their communication interdependencies are represented by those, rather than geometry. Effective parallelization of these applications requires mapping methods with connectivity-based models. Therefore, there is not single model and a mapping method that meets the needs of all scientific applications. For these reasons, in this work, the load-balancing problem and task mapping problems are studied using both connectivity-based and spatial (geometric) models and methods.

Table 1.1: Overview of this work. The load-balancing and task mapping problems, that are first and second phases of the mapping problem respectively, are studied using both geometric and connectivity-based models from Chapter 3 to Chapter 7.

| | | Mapping problem | |
|---|---|---|---|
| | | Load-Balancing | Task Mapping |
| Models | Geometric | Chapter 3 [62] | Chapter 6 [63] |
| | Connectivity-Based | Chapter 4, 5 [44, 45, 57, 61] | Chapter 7 [60] |

Table 1.1 gives the overview of the rest of this dissertation. Below we briefly describe contributions of this work in the context of the-state-of-the-art, where the

details of these are presented in Chapters 3 to 7. In Chapter 2, background information about the models as well as the load-balancing and task mapping problems is presented. In Chapter 3 and 4, the load-balancing problem has been studied in the contexts of geometric and connectivity-based (hypergraph) models. Chapter 5 presents parallelization (Section 5.1) and sparsification techniques (Section 5.2) to improve the running time of the hypergraph partitioning problem. Similarly, Chapter 6 and 7 study the task mapping problem using geometric and connectivity-based (graph) models. Finally, Chapter 8 presents the conclusion and plans for future work for the dissertation.

## 1.1 Load-Balancing

A good partitioning should divide the load among the processors as evenly as possible, while minimizing the inter-processor communications. In the literature, algorithms have been proposed using connectivity-based information [36, 37, 42, 84, 145]; and geometric information [10, 16, 109, 119, 126, 139] to solve the partitioning problem. Also, a hybrid algorithm that uses both geometric model and connectivity-based models have been studied in [133]. Yet the partitioning problem remains NP-Hard [101], making it challenging to develop effective and scalable solutions.

### 1.1.1 Partitioning using Geometric Models

Geometric (spatial) partitioning is fast and effective for load-balancing dynamic applications that require geometric locality of data. While there exist several works

that study the serial spatial partitioning methods [10, 16, 109, 119, 120, 126, 139] in the literature, there exist fewer efforts to parallelize the spatial partitioning process. For example, Zoltan package [65] provides a distributed parallel implementation of Recursive Coordinate Bisection (RCB) [16] algorithm, in which a cutting plane perpendicular to a coordinate axis is computed such that the total weight of data on each side of the cutting plane is equal; the resulting two subregions are then recursively divided until the desired number of parts is obtained. This algorithm recursively bisects the dataset, and migrate the data after each bisection so that a data locality in the processors is obtained and further partitionings can be performed by a subset of the processors on a subset of the data. Therefore, depending on the initial data distribution, it might cause significant data movement which is in the order of $O(k)$, where $k$ is the number of the parts. In Chapter 3, we present, to our knowledge, the first hybrid parallel (MPI+OpenMP) geometric partitioning algorithm. The algorithm computes multiple cut planes concurrently. This reduces the number of recursion levels, therefore the worst-case cost of the data movements. Moreover, smart heuristics to predict the cost of the data movement are studied, such that the algorithm decides whether to migrate or not during run time. The experiments that are run on variety of the datasets (e.g., synthetic and real datasets with different initial distributions) show that the proposed algorithm performs and scales better than the existing RCB method without degrading the load balance. The implementation computes $65,536$

parts of 24 billion points within a few seconds and exhibits near perfect weak scaling up to 6144 cores.

### 1.1.2 Partitioning using connectivity-based models

The connectivity-based partitioning problem, specifically the hypergraph models and hypergraph partitioning methods, is studied in Chapter 4. To solve this partitioning problem, several graph [84, 145] and hypergraph models have been proposed [36, 37, 42] in the literature. These models transform the partitioning problem such that the balance restriction on part weights corresponds to the load balance in a parallel environment, and the minimization objective relates to the minimization of the communication between the processing units. Graphs models [93, 117, 124] usually minimize the edge-cut which is an approximation to total communication volume, where as hypergraph models [38, 64, 95, 142] can accurately model the total communication volume metric. However, other communication metrics such as the total number of messages, the maximum amount of data transferred by a processor, or a combination of them are equally, if not more, important. Unlike most of the previous graph and hypergraph partitioners, the main objective in Chapter 4 is to minimize multiple communication cost metrics, while preserving the load-balance of the processing units. Although there exist studies addressing the minimization of multiple communication cost metrics (such as [25,136]), they work in separate phases where each phase is concerned with disjoint subsets of communication cost metrics,

possibly at the expense of other phases. Unlike these multi-phase studies, in one-phase approach the aim is to concurrently minimize multiple communication metrics in a single partitioning phase. Chapter 4 shows that the existing graph/hypergraph models are not suitable for the other communication metrics, and proposes *a directed hypergraph model* that allows the optimization of multiple communication metrics, such as maximum communication volume, maximum number of messages, total number of messages as well as the traditional total communication volume. Also, since the minimization of the per-processor communication cost metrics requires a global view of the partition, the use of direct $K$-way partitioning heuristics are suggested instead of traditional recursive bisection algorithm; and new multi-objective $K$-way refinement heuristics are proposed. The proposed model and the methods are implemented a under software tool, UMPa. We show that, on average, UMPa produces 24%, 17%, 20% better partitions (than the best state-of-part hypergraph partitioner) in terms of the maximum communication volume, maximum send message and total message, respectively.

### 1.1.3 Runtime improvements for Hypergraph Partitioning

Although high quality partitions can be obtained with the connectivity-based methods, their run times can be expensive due to their computational complexities. This problem is addressed in Chapter 5. The parallelization techniques to speed-up the hypergraph partitioning process are investigated in Section 5.1, whereas the sparsification techniques to reduce the size of the problems are studied in Section 5.2.

Most of the current-state-of-the art hypergraph partitioners are designed and implemented before the advancements in multi-core computing. Thus, utilization of such advancements have become important in such tools for reducing the cost of the partitioning process. Hypergraph partitioners follow the multi-level approach, which consists of three phases: coarsening, initial partitioning, and uncoarsening. The coarsening phase is the first phase and it corresponds to the clustering of the highly connected vertices in the hypergraph. It iteratively reduces the hypergraph size, and obtains smaller and similar hypergraphs to the original hypergraph. At the initial partitioning phase, an initial solution is found for the coarsest hypergraph, and this solution is iteratively refined and back-projected to the original hypergraph in uncoarsening phase. A parallelism can be exploited in all of these three phases. However, only the parallelization of the coarsening step is studied in Section 5.1, since the coarsening phase is the most time consuming phase of a traditional recursive bisection based multi-level partitioning, and its worst case time complexity is quadratic, while the other phases have linear time complexities in most of the implementations. Moreover, it is the most important phase of the partitioning. The quality and runtime of the following phases and the overall partitioning process highly depends on the quality obtained in the coarsening phase. Therefore, we present three efficient multi-threaded clustering algorithms for the hypergraph coarsening in Section 5.1. Their performance are compared with that of the ones currently used in today's hypergraph partitioners. Experiments on a large number of hypergraphs show that the

proposed parallel algorithms achieve good speedups without any harm on the quality of the partitioning.

The runtime of hypergraph partitioning process can be further improved by data compression and sparsification techniques regardless of the underlying architecture properties. There exist a few works that partially study the hypergraph compression techniques in the literature. For example, a compression technique that removes the identical nets has been used in [11, 38, 64], however, the effectiveness of the sparsification is not studied in detail. In addition, some of the state-of-art hypergraph Partitioners PaToH and Zoltan, have some implementations for hypergraph compression but not sparsification. The cost of the identical net removal methods does not amortize in PaToH, therefore it is disabled. On the other hand, Zoltan offers some sorting based compression techniques, which has room for further improvements. Novel hypergraph compression and sparsification techniques and the problem of handling the large data in partitioning problem are investigated in Section 5.2. A set of compression and sparsification techniques is presented and these techniques are implemented under UMPa [45, 57] for the hypergraph partitioning problem. Experiments on a large set of hypergraphs from various classes show that these techniques significantly reduce the cost of the partitioning process with a minimal harm on the partitioning quality.

## 1.2 Task mapping

The first phase of the mapping problem is completed once the desired number of parts are obtained via load-balancing. Mapping parts, and hence applications' tasks, to the cores of a parallel computer is performed in the second phase of the mapping problem. This becomes crucial for obtaining high performance, since the number of processors in new supercomputers is growing from $O(100K)$ to $O(1M)$. The supercomputers have larger and more complex networks, and more users are using them concurrently. Most of the supercomputers allow sparse processor allocations for the user jobs (e.g., Cray, clusters). Users concurrently submit jobs with various sizes, thus most of the time, processor allocations become sparse, and communicating tasks are potentially placed farther in the network. This causes communication messages to travel longer routes, furthermore, network links to be congested with heavy traffic. Thus communication stalls might occur more often on the communication links, which can be an important obstacle to obtain a better performance on large supercomputers. This increases the importance of the task mapping algorithms, that place the heavily communicating tasks to the nearby cores, so that the distances that messages must travel and the amount of congestion in the network are lowered. Chapter 6 and Chapter 7 address this task mapping problem.

### 1.2.1 Task mapping using geometric models

Most of the research on task mapping are carried out for the block-based allocations (e.g., IBM's BlueGene systems) [5, 20, 80, 148]. Unlike these works, the focus of the mapping algorithms presented in Chapter 6 are the non-contiguous sparse allocations. However, mapping strategies developed for general allocations can be used automatically for the more restricted case of block allocations. Regardless of the allocation type, the mapping problem can be modeled using either connectivity-based or spatial models. In the literature, most of the research carried out are based on the graph models [26, 28, 49, 50, 100]. Unlike most of the previous research, in this work the mapping problem is formulated using the geometric models as will be presented in Chapter 6. The proposed geometric mapping algorithm [63] is used to reorder the tasks and processors using their geometric information. This reordering is later used to map the tasks that are close to each other to the nearby processors. The experiments on two different regular applications show that the proposed mapping method reduced the total execution time up to 34% on average on 65,536 cores of a Cray XE6. The mapping algorithm is also compared to the existing mapping methods, and shown that it obtains better performance results. In addition, we evaluate mapping metrics using performance counter information from the Cray Gemini routers.

### 1.2.2 Task mapping using graph models

Geometric models to perform task mapping are not applicable when the applications have no geometry information, or the communication pattern is irregular.

Moreover, some of the new interconnection networks (e.g., DragonFly Network) cannot be accurately represented with the geometric models. In Chapter 7, we propose novel graph-based task mapping algorithms for sparse allocations.

Using the graph models, we show accurate representation of mapping the application onto the interconnection network so that various communication overhead metrics are captured correctly. Using this model, efficient mapping algorithms that obtain high quality mappings have been proposed. We show that these mapping methods accurately minimize various communication overhead metrics. On the experiments on two irregular application, we show improvements on the overall execution time of the applications up to 23% and 43% by just performing an accurate placement of the tasks on the allocated processors.

## 1.3 Summary of contributions of this dissertation

This dissertation has contributions on the two-phase mapping problem on both geometric models and connectivity-based models, which improve the state of art in terms of various aspects. This contributions are summarized below.

- **New algorithms for parallel geometric load-balancing**: To our knowledge, first hybrid (MPI+OpenMP) parallel geometric partitioning algorithm has been proposed. Techniques that significantly improve overall partitioning time by minimizing data movement has been studied.

- **New algorithms for geometric task mapping:** Using the proposed geometric partitioning algorithm and geometric models, a geometric task mapping algorithm has been proposed.

- **Novel directed hypergraph model:** The lack of the existing models on modeling the applications' communication requirement for multiple communication metrics has been shown, and a directed hypergraph model that accurately models various communication metrics has been proposed.

- **New algorithms and tools for connectivity-based load-balancing and mapping:** Partitioning and mapping algorithms that can handle multiple communication metrics have been proposed.

- **New hypergraph compression and sparsification techniques:** In order to speed up the methods adapting connectivity-based models, compression and sparsification techniques that are used to represent the large hypergraphs in smaller and similar ways have been proposed.

- **Novel parallel matching and clustering algorithms:** In order to speed up the hypergraph partitioning process, novel parallel algorithms have been proposed for hypergraph matching and clustering.

# Chapter 2: Background

The mapping problem can be defined as the combinatorial optimization problem that aims to reduce the overall execution time of a parallel application by assigning its tasks to a parallel computer's processors in which the computational load is distributed evenly to the processors with a good inter-processor communication pattern. Finding the optimal solution to mapping problem is NP-Complete [52, 107]. Therefore, various sub-optimal heuristics have been proposed in the literature.

A mapping method models the underlying computational structure of the tasks using two general models: geometric models, connectivity-based models. Regardless of the model that has been used, mapping methods both partition and map the tasks to the processors. They achieve the computational load-balance of the processors and inter-processor minimization by partitioning the tasks to the processes, and further improvements on the communication pattern (e.g. reducing the distance messages travel, or the physical link contentions) are addressed by mapping of the partitions to the physical processors. Existing mapping heuristics can be divided into two classes: single-phase and two-phase methods. The algorithms in the former class perform

simultaneous partitioning and mapping [68, 115, 143], whereas those in the latter partition the tasks in the first phase and map these partitions to the processors in the second one. In this work, we focus on the two-phase approaches, and below we explain the load-balancing and task mapping, which are the first and second phases of a two-phase mapping method, respectively.

## 2.1 Preliminaries

In scientific computing, the models that have been used to represent the computational tasks and their interactions can be broadly divided into two categories: geometric and connectivity-based. Below we define these models.

### 2.1.1 Geometric Data

Geometric models represent the computational tasks by their physical coordinates (e.g., (x,y,z) in 3D). Each task is associated with a weight representing the computation cost of the task. They leverage the assumption that computations which are close by in the space are more likely to share data than computations that are far in the space. Therefore, rather than an exact model for the communication requirement of the application, they provide an estimate model on the applications' communication. They are widely used for applications such as contact detection and particle methods for which geometric locality of data is important. Moreover, they can be used in applications where connectivity information is unavailable, such as particle methods or visualization.

### 2.1.2 Connectivity-Based Models

Connectivity-based methods model the tasks and their interaction as a graph or a hypergraph with computation weights on the nodes and communication volumes on the edges or hyper edges. They provide a more accurate model for the communication requirements, however, they are usually computationally expensive and require to build an accurate graph (or hypergraph) model of the computation. These models suit well for the applications in which the interactions between tasks are irregular.

#### 2.1.2.1 Graph Models

An undirected graph $G = (V, E)$ is defined as a set of vertices $V$, and a set of edges $E$ . Each edge $e \in E$ connects a two distinct vertices, and the number of edges adjacent to a vertex $v \in V$ is called the degree of $v$. The vertices and edges of the graph can be associated with weights and costs, respectively. Graph models model an application by representing its tasks with vertices, and their interactions with edges. The weights associated with the vertices correspond to the computational weights of the tasks, while the costs associated with the edges correspond to the communication cost between these tasks. Graph models are useful and accurate when modeling interactions that are exactly between two tasks, however, it has some deficiencies when modeling more complex interactions that could involve more than two tasks.

#### 2.1.2.2 Hypergraph Models

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. A net $n \in \mathcal{N}$ is a subset of vertices and the vertices in $n$ are called its *pins*. The number of pins of a net is called the *size* of it, and the *degree* of a vertex is equal to the number of nets it belongs to. We use $\mathsf{pins}[n]$ and $\mathsf{nets}[v]$ to represent the pin set of a net $n$, and the set of nets containing a vertex $v$, respectively. Vertices can be associated with weights, denoted with $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted with $\mathsf{c}[\cdot]$. Hypergraph models model an application by representing its tasks with vertices, and their interactions with nets. The weights associated with the vertices correspond to the computational weights of the tasks, while the costs associated with the nets correspond to the communication cost between these tasks. Hypergraphs can be considered as generalization of graph models which allows to represent complex interactions that could involve more than two tasks.

## 2.2 Load Balancing Problem and Models

The basic aim of the load-balancing is to distribute the workloads evenly across multiple processors. The methods adapting geometric models are called spatial partitioning methods, while those adopting graph and hypergraph connectivity-based models are called graph and hypergraph partitioning methods, respectively. Below we formally describe spatial partitioning, and hypergraph partitioning.

### 2.2.1 Spatial partitioning

Spatial partitioners use only geometric information from the application to compute partitions. They have several advantages over graph- or hypergraph-based methods. Spatial partitioners typically have lower runtimes to compute partitions. They assign physically close objects together within a processor; this feature is important for applications such as contact detection and particle methods for which geometric locality of data is important. And they can be used in applications where connectivity information is unavailable, such as particle methods or visualization.

The input to spatial partitioners includes a set of physical coordinates (e.g., $(x, y, z)$ in 3D) and, optionally, weights associated with each point. Additionally, desired part sizes can be provided; the algorithmic details of handling nonuniform part sizes are straightforward. The output of spatial partitioners is a list of part numbers to which the input points are assigned.

Formally, given the coordinates $C_{i,j}$ and weights $W_j$ of $N$ data points in $d$ dimensional space ($0 \leq i < d$ and $0 \leq j < N$), and the number of parts $\kappa$ and maximum allowed imbalance ratio $\varepsilon$, the spatial (coordinate) partitioning problem can be defined as finding $\kappa$ non-overlapping bounding boxes $B_k = \{b_{1-}, b_{1+}, b_{2-}, b_{2+}, \ldots, b_{d-}, b_{d+}\}$ for all parts $1 \leq k \leq \kappa$ that covers all the data points. A data point $j$ is said to be in part $k$, i.e., $\mu_j = k$, if and only if it is inside the bounding box $B_k$, i.e., $b_{i-} < C_{i,j} < b_{i+}$ for all $0 \leq i < d$. The goal is to achieve balanced parts defined as:

$$\sum_{j,0\leq j<N\wedge k=\mu_j} W_j \leq (1+\varepsilon)W_{avg}$$

for all parts $1 \leq k \leq \kappa$, where $W_{avg}$ is the average ideal weight defined as $W_{avg} = \frac{\sum_j W_j}{\kappa}$.

### 2.2.2 Hypergraph partitioning

A *K-way partition* of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is a partition of its vertex set, which is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$, where

- parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$,
- each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, i.e., $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$,
- the union of $K$ parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

Let $W_k$ denote the total vertex weight in $\mathcal{V}_k$, that is $W_k = \sum_{v\in\mathcal{V}_k} \mathsf{w}[v]$, and $W_{avg}$ denote the weight of each part when the total vertex weight is equally distributed, that is $W_{avg} = \sum_{v\in\mathcal{V}} \mathsf{w}[v]/K$. If each part $\mathcal{V}_k \in \Pi$ satisfies the *balance criterion*

$$W_k \leq W_{avg}(1+\varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{2.2.1}$$

we say that $\Pi$ is *$\varepsilon$-balanced* where $\varepsilon$ is called the maximum allowed imbalance ratio.

For a $K$-way partition $\Pi$, a net that has at least one pin (vertex) in a part is said to *connect* that part. The number of parts connected by a net $n$ is called its *connectivity* and denoted as $\lambda_n$. A net $n$ is said to be *uncut* (*internal*) if it connects exactly one part (i.e., $\lambda_n = 1$), and *cut* (*external*), otherwise (i.e., $\lambda_n > 1$). Given a

partition $\Pi$, if a vertex is in the pin set of at least one cut net, it is called a *boundary vertex*.

In the text, we use $\mathsf{part}[v]$ to denote the part of vertex $v$ and $\mathsf{prts}[n]$ to denote the set of parts net $n$ is connected to. Let $\Lambda(n,p) = |\mathsf{pins}[n] \cap \mathcal{V}_p|$ be the number of pins of net $n$ in part $p$. Hence, $\Lambda(n,p) > 0$ if and only if $p \in \mathsf{prts}[n]$.

There are various cutsize definitions [101] for hypergraph partitioning such as:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} \mathsf{c}[n] \tag{2.2.2}$$

$$conn_{\mathcal{H}}(\Pi) = \sum_{n \in \mathcal{N}} \mathsf{c}[\mathrm{n}](\lambda_n - 1)\ . \tag{2.2.3}$$

In (2.2.2) and (2.2.3), each cut net $n$ contributes $\mathsf{c}[n]$ and $\mathsf{c}[n](\lambda_n - 1)$ to the cutsize, respectively. The cutsize metric given in (2.2.2) will be referred to here as *cut-net* metric and the one in (2.2.3) will be referred as *connectivity* metric. The *connectivity* metric is widely used in the literature and shown to accurately model the total communication volume of parallel sparse matrix-vector multiplication [37]. Given $\varepsilon$ and an integer $K > 1$, the hypergraph partitioning problem can be defined as the task of finding a balanced partition $\Pi$ with $K$ parts such that $conn_{\mathcal{H}}(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [101].

The total amount of data transfer throughout the execution of the tasks is called the total communication volume ($\mathsf{TV}$). Given an application with interacting tasks, in the traditional hypergraph model, the tasks, interactions, and processors correspond to vertices, nets, and parts, respectively. In this model, when a net $n$ is connected to

multiple parts, a communication of size $\mathsf{c}[n]$ from one of those parts to others (i.e., to $\lambda_n - 1$ parts) is required, hence resulting in a communication cost of $\mathsf{c}[n](\lambda_n - 1)$. Hence, the connectivity-1 metric (2.2.3) corresponds exactly to the total communication volume [47]. Note that this metric is independent from the direction of the interactions.

The traditional undirected hypergraph model is used for circuit partitioning in VLSI layout design [101]. It is also widely used to model various scientific computations such as the sparse-matrix vector multiplication (SpMxV) [36,37,42,142]. There are three basic models for partitioning a matrix by using a hypergraph: *column-net*, *row-net*, and *fine-grain*. In the column-net model, the rows and the columns are represented with vertices and nets, respectively [37]. It is vice versa for the row-net model. In the fine grain model, the nonzeros of the matrix correspond to the vertices and the rows and columns correspond to the nets [39]. A one-dimensional row (column) partitioning is applied when the matrix is modeled with a column-net (row-net) hypergraph. For the fine-grain model, a 2D partitioning of the nonzeros is employed. We refer the reader to [42, 47] for further information and comparative evaluation of these models.

## 2.3 Task Mapping

Task mapping aims to improve the execution time of a parallel application with the assignment of a parallel application's tasks to the processors of a parallel computer.

### 2.3.1 Problem Definition and Mapping Metrics

Let $G_t = (V_t, E_t)$ be a directed MPI task graph, where $V_t$ is the task set, and $E_t$ is the edge set modeling task-to-task communications, i.e., $(t_1, t_2) \in E_t$ if and only if $t_1 \in V_t$ sends a message to $t_2 \in V_t$. Let $G_m = (V_m, E_m)$ be the network topology graph where $V_m$ is the set of computing nodes equipped with many processors/cores, and $E_m$ is the edge set modeling the physical communication links between the nodes, i.e., $(m_1, m_2) \in E_m$ if and only if $m_1 \in V_m$ and $m_2 \in V_m$ have a link in between. Let $V_a \subseteq V_m$ be the set of computing nodes reserved for the application. The topology-aware mapping problem can be defined as finding a mapping function $(\Gamma : V_t \rightarrow V_a)$ that minimizes the parallel execution time. For a task set $S \subseteq V_t$, we use $\Gamma[S]$ to denote the node set to which the tasks in $S$ are mapped to.

There are two well-received metrics to model the network communication overhead. The *total hop count* (TH), which is the total length of paths taken by communication packets, is a latency-based metric. The *maximum message congestion* (MMC), which is the maximum number of messages sent across a link. We assume that the messages are not split and sent through only a single path via static routing.

Let $\Gamma$ be a given mapping where $m_1 = \Gamma(t_1)$ if and only if $t_1$ is assigned to one of node $m_1$'s processors. Then

$$\text{TH}(\Gamma) = \sum_{(t_1,t_2)\in E_t} \text{dilation}(t_1, t_2),$$

where

$$\text{dilation}(t_1, t_2) = \text{SPL}(\Gamma(t_1), \Gamma(t_2), G_m)$$

which is the shortest-path length between $\Gamma(t_1) \in V_m$ and $\Gamma(t_2) \in V_m$.

Let the congestion on a network link $e \in E_m$ be

$$\text{Congestion}(e) = \sum_{(t_1,t_2)\in E_t} \text{inSP}(e, \Gamma(t_1), \Gamma(t_2), G_m), \tag{2.3.1}$$

where, inSP = 1 if and only if $e$ is on the shortest path between $\Gamma(t_1)$ and $\Gamma(t_2)$, and 0 otherwise. Therefore, a link's congestion is equal to the number of messages that passes through it. Hence, the maximum message congestion is

$$\text{MMC}(\Gamma) = \max_{e\in E_m}\{\text{Congestion}(e)\}.$$

The above metrics assume unit communication costs and link capacities. In order to handle heterogeneous costs and bandwidths, the task and topology graph models are extended as follows. Each edge $e' \in E_t$ in the task graph is associated with a cost $c(e')$ that corresponds to the communication volume sent/received between the tasks. Similarly, each link $e \in E_m$ in the topology graph is associated with a communication capacity $bw(e)$ that corresponds to the link bandwidth. Moreover, for further heterogeneity support, each node $m \in V_m$ is associated with a computation capacity $w(m)$ that corresponds to the number of available (allocated) processors in $m$. All the nodes that are not in $V_a$ have zero capacity. Based on these attributes on the edges, the *weighted hop* (WH) metric, i.e., the total number of hops taken by each packet, is

$$\text{WH}(\Gamma) = \sum_{(t_1,t_2)\in E_t} \text{dilation}(t_1, t_2)c(t_1, t_2).$$

We define the *volume congestion* (VC) of a link $e \in E_m$ as

$$\text{VC}(e) = \frac{\sum_{(t_1,t_2)\in E_t} \text{inSP}(e, \Gamma(t_1), \Gamma(t_2), G_m)c(t_1, t_2)}{bw(e)}$$

which is the ratio between the overall volume of the communication passing from $e$ and its bandwidth. Then, the *max volume congestion* (MC) metric is the congestion of the bottleneck link in the network with the maximum VC value.

A communication-bounded parallel application can be either latency-bounded or bandwidth-bounded depending on the communication pattern. For example, applications with frequent communication steps and small messages are likely to be latency-bounded, while those with larger messages are likely to be bandwidth-bounded. For latency-bounded applications, TH and WH metrics can better correlate with the communication overhead. On the contrary, MMC and MC are better choices for the bandwidth-bounded applications. Depending on the communication pattern, it might be better to minimize MC (MMC) in the expense of an increase on WH (TH), or vice versa. However, it is indeed difficult to find a consensus between these metrics. Therefore, here we define *average message congestion* (AMC) and *average congestion* (AC) metrics to account for both number of hops and congestion. Let $E_m^t$ be the set of the links that are used throughout the execution of the parallel application. The message and volume based average link congestions over the used links can be defined as

$$\text{AMC}(\Gamma) = \frac{\sum_{e \in E_m} \text{Congestion}(e)}{|E_m^t|},$$

$$\text{AC}(\Gamma) = \frac{\sum_{e \in E_m} \text{VC}(e)}{|E_m^t|}$$

Since $\text{TH} = \sum_{e \in E_m} \text{Congestion}(e)$, AMC is the ratio of TH to the total number of links used during the application. Similarly, AC is related to the ratio of WH to the number

of links used during the execution (they are equal only when the communication links have unit bandwidths).

### 2.3.2 Targeted Computing Environment

Six of the top ten supercomputers in the Top500 list (June 2014), have torus networks (one 3D, four 5D, one 6D). In this work, our target computing environment are Cray XE6's 3D torus supercomputers, specifically, Cielo and Hopper. In the Cray XE6's 3D torus, the network has Gemini routers directly connected to two computing nodes. Each router is associated with a coordinate on $x$, $y$, and $z$ dimensions, and connected to all six neighbor routers (see Figure 2.1). Messages' routes between nodes can be represented as a path of "hops" along network links in $x$, $y$ and $z$. The torus network provides wrap-arounds, and the messages between the nodes are statically routed following the shortest paths. Differences in bandwidth along the various dimensions can exist depending on the physical connections (e.g., backplane, mezzanine, cable) used in each dimension.

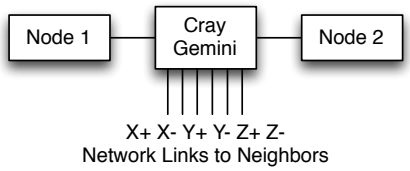



Figure 2.1: Layout of a Cray Gemini router

In mesh- and torus-based systems, "coordinates" of the routers within the network are often made available by calls to a system library. A router with 3D coordinates $(i, j, k)$ can communicate with a router with coordinates $(i + 1, j + 1, k + 1)$ via a three-hop path, with one hop in each of the $x$, $y$ and $z$ dimensions. The torus provides wrap-around, so that messages take the shortest path (i.e., proceed in the positive or negative direction) along each dimension. In the Cray XE6, these coordinates are available from Resiliency Communication Agent (RCA) tool through calls to `rca_get_meshcoord`. Each MPI process can obtain the coordinates of the router to which its compute node is attached. Moreover, the topology information of the whole machine, e.g., the routers' coordinates and connections, as well as link bandwidths, can be captured using system calls (`rtr`), and a static topology graph can be obtained. During runtime, each MPI process can obtain its node id, and the vertices in the topology graph can be associated with the computational units. We use the coordinate information of each processor is in our geometric task mapping method in Chapter 6, while the static graph information is used in Chapter 7.

Each router in a mesh/torus network is typically connected to one or more multi-core compute nodes. In the Cray XE6, for example, each router connects two nodes (hosts). The two Cray platforms we used (DOE's Cielo and NERSC's Hopper) have 16 and 24 cores per node, respectively. Parallel applications can use from one MPI process per node (with threading providing parallelism within the node) to one MPI process per core (with shared-memory message passing within the node). In the latter

case, co-locating interdependent MPI processes within a node reduces communication over the network, and, thus, reduces execution time.

One practical difference between Cray XE and IBM BlueGene systems is the way compute nodes are allocated to jobs. In IBM systems, jobs are given a contiguous "partition" or block of nodes within the network; each dimension of this block must be a power of two. In contrast, on Cray systems, jobs are given non-contiguous node allocations of any size requested by the user. Available nodes are selected according to a space-filling curve algorithm in the ALPS scheduler [4]. Thus, while the scheduler attempts to assign nearby nodes to jobs, no guarantees of locality are provided. As a result, task-mapping algorithms for Cray systems need to accommodate non-block, non-contiguous allocations. While our methods are designed for non-contiguous allocations, they could be applied to contiguous allocations as well.

In Hopper, the network latencies for the nearest and farthest node pairs are 1.27$\mu$s and 3.88$\mu$s, respectively. The link bandwidths vary from 4.68 to 9.38 GB/sec. Ideally, reducing the hop counts between the communicating tasks lowers the overhead. But when there are many communicating tasks, a link can be congested due to the communication pattern, which might cause communication stalls and harm the performance. Still, thanks to static routing, the congestion can be measured and optimized accurately.

### 2.3.3 Related Work

Existing mapping algorithms can be divided into two classes: *single-phase* and *two-phase* methods. The algorithms in the former class perform simultaneous partitioning and mapping using the task and topology graphs, whereas those in the latter partition the task graph in the first phase and map the parts to the processors in the second one.

Pellegrini and Roman [116] proposed a single-phase recursive-bipartitioning algorithm for various topologies. Wallshaw and Cross [143] proposed a multilevel partitioning algorithm which performs mapping in the initial partitioning and refinement phases.

The two-phase mapping methods make an abstraction of the partitioning phase and work with a pre-partitioned task set. These studies can be divided into two based on the task-dependency model and network topology. The first set adapts geometric task-interaction models, e.g., [17, 19] on IBM's BlueGene systems with block-based node allocations and on sparse allocations [63]. Still, most of the work focuses on the connectivity-based models, specifically graph models. This problem is shown to be NP-complete [26,89], and many heuristics exist in the literature, e.g., [18,28,49,50,60]. In this work, we investigate two-phase approaches using both graph and geometric models.

## 2.4 Multi-level framework and partitioning

The *multi-level approach* has been shown to be the most successful heuristic for various graph/hypergraph partitioning problems [11, 13, 34, 37, 85, 134]. Although, it has been first proposed for recursive-bisection based graph partitioning, it also works well for hypergraphs [37, 95]. In the multi-level approach, a given hypergraph is successively (i.e., level by level) coarsened to a much smaller one, a partition is obtained on the smallest hypergraph, and that partition is successively projected to the original hypergraph while being improved at each level. These three phases are called the *coarsening*, *initial partitioning*, and *uncoarsening* phases, respectively. In a coarsening level, similar vertices are merged into a single vertex, reducing the size of the hypergraph at each level. In the corresponding uncoarsening level, the merged vertices are split, and the partition of the coarser hypergraph is refined for the finer one using Kernighan-Lin (KL) [96] and Fiduccia-Mattheyses (FM) [71] based heuristics.

Most of the multi-level partitioning tools used in practice, such as the sequential partitioning tools like PaToH [38] and hypergraph partitioning implementation of Mondriaan [142], and parallel hypergraph partitioning implementation of Zoltan [64], are based on *recursive bisection*. In recursive bisection, the multi-level approach is used to partition a given hypergraph into two. Each of these parts is further partitioned into two recursively until $K$ parts are obtained in total. Hence, to partition a hypergraph into $K = 2^k$, the recursive bisection approach uses $K - 1$ coarsening,

initial partitioning, and uncoarsening phases. A direct $K$-way partitioning approach within the multi-level framework is also possible. Given the hypergraph, the partitioner gradually coarsens it in a single coarsening phase and then partitions the coarsest hypergraph directly into $K$ parts in the initial partitioning phase. Starting with this initial partition, at each level of the uncoarsening phase, the partitioner applies a $K$-way refinement heuristic after projecting the partition of the coarser hypergraph to the finer one. Hence, to partition a hypergraph, a $K$-way partitioner has only one coarsening, one initial partitioning, and one uncoarsening phase, where an RB-based partitioner has $K-1$ of them. On the other hand, the initial partitioning and uncoarsening phases are much cheaper for RB since a $K$-way partitioner needs a $K$-way partition in the initial partitioning scheme and a $K$-way refinement at each level of the uncoarsening phase. Several works show that, the direct $K$-way approach can be successfully used within a multi-level framework [11, 45, 134] and it can produce higher quality results. The only caveat is that the partitioning time of a direct $K$-way approach increases more rapidly with increasing $K$, in comparison to the RB-based partitioner.

# Chapter 3: Load-Balancing via Geometric Models

In many areas of science and engineering, both problem and dataset sizes are increasing rapidly due to technological advances that enable generation and storage of such data. This growth is fueled by our urge to better understand more complex phenomena. Many of the datasets generated are irregular, in the sense that no immediately observable or predictable pattern exists among data elements. Lack of such patterns makes it difficult to find simple, effective and efficient mechanisms to partition data for scalable parallel processing, as well as indexing and storage. Some data, like interaction networks (e.g., gene/protein interaction networks, social networks, task dependency networks) are defined only by interactions among entities. Another significant portion of the data is defined by spatial coordinates together with some governing rules about how the elements interact based on those spatial coordinates. This chapter deals with efficient partitioning of the latter datasets for parallel processing.

With the drastic increase in system sizes, scalable partitioning for distributing data and tasks to processors has become critical to application performance. Researchers

have been developing partitioning techniques for interaction-based datasets, in the form of graph and hypergraph partitioning ( [37,64,84,85,93,96,127]) and for spatial datasets ( [16, 82, 109, 120, 126]) for more than four decades. Yet the complexity of the problems (many are NP-hard) makes it challenging to develop both effective and scalable solutions. For many applications for which spatial information about data elements is available, spatial partitioning is preferred over interaction-based (also called connectivity-based) methods because of its speed, despite the fact that spatial partitioners do not provide an exact model for the communication cost of most applications. Scalability is crucial especially when growing dataset sizes and architectural changes require applications to use massively parallel machines to obtain effective solutions. The scalability of the partitioner is an important part of the applications' scalability, especially in applications with a need for dynamic load balancing. In the context of partitioning for load balancing, massively parallel machines require both the number of parts needed by the application and the number of processors available to the partitioner scale at the same rate.

Doing partitioning in parallel is a chicken-and-egg problem. We assume the data is already distributed in the memory of the parallel machine by the application and there is no replication. This initial data distribution directly affects at least the execution time of the partitioner, and in some implementation might affect the partition quality. The obvious example is that if the initial data distribution is not load balanced, some of the processors may have significantly more work than others, leading

to load imbalance while computing the partition. Some succesful parallel partitioners, such as recursive coordinate bisection (RCB) [16] implementation in Zoltan, migrate data while partitioning, which aims to improve the load imbalance during partitioning and also helps to create data locality. As its name implies, RCB obtains a partition of a dataset by recursively bisecting the domain into two parts with cutting planes orthogonal to a coordinate axis, so that the weight of data on each side of the cutting plane is equal. Migration happens after each bisection, and hence, depending on the initial data distribution, it could result in significant data movement during partitioning.

In this chapter, an efficient parallel geometric partitioning algorithm is proposed for multi-dimensional datasets. In simple terms, the algorithm achieves a final partition by recursively multi-sectioning the dataset. Hence, the proposed algorithm can be viewed as generalization of two-dimensional jagged partitions (also called Semi Generalized Block Distribution) [106, 119, 139] to multiple dimensions. There are many efficient heuristics and optimal, sequential algorithms [106, 119, 126] to compute the two-dimensional partitions, when the workload is given by a dense two-dimensional array, where each element represents load of its respective object in two-dimensional space. We aim to partition objects where their real-valued coordinate information is given in a multi-dimensional space. The algorithm can be also viewed as a generalization of RCB such that in each level of recursion, instead of doing bisection, we perform multi-sections where the number of sections can be given by the user

(or computed by us). Hence, one can use the proposed algorithm to perform RCB. However, as shown in [126], by considering multiple cut-lines concurrently, heuristic jagged partitioning generally yields better load balance, especially for workloads with non-uniform weights, than a greedy recursive bisection algorithm which assumes that a good bisection at each level yields a good final partition. As we will further show, in a parallel environment, the proposed algorithm also scales better than RCB due to reduced data movement.

We have implemented our algorithm in the Zoltan2 [30] framework, which is the revision of Zoltan library [75], that exploits recent advances in compiler technology and software design principles. Zoltan2 is also our platform for delivering current research in combinatorial scientific computing. We have experimentally evaluated our implementation of the new algorithm against Zoltan's robust RCB implementation, which has been successfully used in many large scale applications [53, 54, 76]. The experiments on real and synthetic datasets show that the proposed algorithm performs and scales better than the existing RCB method in terms of run-time without degrading the load balance. We also compare the communication patterns of the resulting partitions, and show that, although the multi-sectioning has a larger theoretical upper-bound, the two methods have similar communication patterns in practice.

The rest of the chapter is organized as follows. Background and related work are presented in Section 3.1. Section 3.2 includes the details of our proposed parallel multi-dimensional jagged algorithm, *multi-jagged*. Experimental evaluation is presented in Section 3.3. We conclude in Section 3.4. Refer to Section 2.2.1 for the problem formulation.

## 3.1 Background

The complexity of spatial partitioning increases with the number of dimensions in the data, the existence of weights for the data, and the data's density/sparsity. For dense one-dimensional (1D) partitioning (also known as *chains-on-chains* partitioning), Pınar and Aykanat [120] provide an extensive theoretical and experimental comparison of 1D algorithms. For dense two-dimensional partitioning (2D), Saule et al. [126] present a survey of existing methods and introduce new, more effective variants. For example, *rectilinear partitioning* (also known as *general block distribution* [10]) partitions 2D space into a $P \times Q$ mesh with non-uniform mesh spacing such that the load of each part is balanced. This problem is NP-hard [78], and, hence, efficient heuristics are proposed [109]. $P \times Q$ *jagged-partitioning* [119, 139] relaxes rectilinear partitioning such that space is first partitioned into $P$ row (or column) stripes, and then each stripe is partitioned independently into $Q$ parts. In another variant called *m-way Jagged partitioning* [126], the number of parts while partitioning $P$ stripes is decided based on the load of each stripe. Octrees [104, 108] and space-filling curves [112, 118, 146] have been used for two- and three-dimensional spatial

partitioning; both produce partitions with similar quality. Each point is assigned an octant or a space-filling curve key, respectively, based on its position in space; a 1D traversal of the octree or space-filling curve is then partitioned into equally weighted parts.

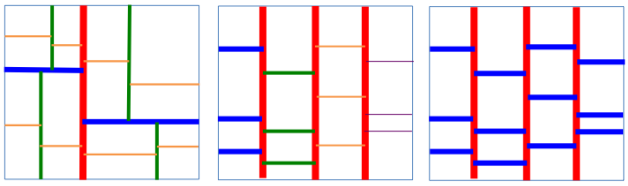


Figure 3.1: A partition of size 16 using (left) RCB, (middle) Multi-Jagged with no migration for $4 \times 4$ configuration, and (right) Multi-Jagged with migration. The order the cuts are computed are indicated by colors red first, then blue, green, orange, and purple and line thickness (from thickest to thinnest).

The most popular techniques for spatially partitioning dense or sparse data arguably are variants of *recursive bisection*. In Recursive Coordinate Bisection (RCB) [16], a cutting plane perpendicular to a coordinate axis is computed such that the total weight of data on each side of the cutting plane is equal; the resulting two subregions are then recursively divided until the desired number of parts is obtained. (When the number of desired parts is not a power of two, parts can be obtained by simply adjusting split ratios in each bisection.) Figure 3.1(left) shows an example using RCB to find 16 parts. The thick, red line represents the initial bisection; progressively thinner

blue, green, and orange lines represent subsequent cuts in the recursion. Several alternatives for selecting the dimension ($x$, $y$, or $z$) of each cutting plane [126] are possible, such as alternating dimension or longest dimension. The most common implementations are based on the longest dimension approach; as its name implies, the longest dimension of the bounding box including the data is chosen to be cut. The dimension is selected independently for each bisection, with the intent that subdomains' aspect ratios approach one. Principal axes of the geometric data can be used instead of the coordinate axes; this variant is called Recursive Inertial Bisection [129, 132].

The main kernel of RCB is finding the cutting plane. Zoltan's parallel implementation uses a binary-search approach to finding the median. The minimum and maximum data coordinates in the cut direction are found, and the initial cutting plane is taken as the average of the minimum and maximum. The weight of points to the left and right of the cut is computed via a parallel reduction operation, and the cut is moved left or right depending on whether the left side's load is heavier or lighter, respectively, than the right's. This process repeats until the left and right side are equally weighted. Information about points closest to the cut is included in the reduction operations to allow small amounts of weight to be shifted between parts and, potentially, avoid additional iterations by satisfying the balance criterion immediately.

Once the median is found, data is reorganized into left and right subgroups for recursion. The algorithm recursively partitions the subgroups each with its own new bounding box and a new cut dimension.

In serial implementation, points are not moved when the left and right subgroups are formed; instead, an index array points to the data in each subgroup. But in the parallel implementation, points are migrated into subgroups based on their positions relative to the cutting plane. The processors are divided into two sets; points to the left of the cut are sent to processors in one set while points to the right are sent to processors in the other. Only the coordinate information is sent, not all the application data associated with the points. The amount of data to be sent depends on the input distribution of the data. While doing dynamic partitioning to correct small imbalances or changes in locality, the migration cost is expected to be small. Static partitioning with arbitrarily distributed data, however, can result in significant migration costs.

Once the data is migrated, multiple cuts can be computed independently and in parallel by using different subcommunicators for the processors in different sets. Reduction operations to compute bounding boxes and accumulate weights on each side of a cutting plane during the median-finding routine are performed within the subcommunicators. When the subcommunicator size is one, the implementation reverts to the serial algorithm. There are some similarities between this implementation of

RCB and our implementation of multidimensional jagged algorithm. The nex section describes our algorithm and the implementation details.

## 3.2 Multi-jagged: A Multi-dimensional Jagged Algorithm

The Multi-dimensional Jagged algorithm (MJ) is a geometric partitioning method in which a $P_l$-way multi-sectioning is applied recursively to partition a domain into $\kappa = P_0 \times P_1 \times \ldots \times P_{d'-1}$ parts. Figure 3.1(middle) shows a 16-part decomposition of a 2D domain generated by MJ with $P_0 = P_1 = 4$. Thick red lines show the first four-way multi-section; each of the four subdomains is then divided into four, as shown by the thinner horizontal lines.

The goal for MJ is to improve upon parallel RCB to make a scalable algorithm for very large data sets and for large numbers of parts. MJ has several key differences from RCB.

First, MJ reduces the depth of recursion by doing multi-section instead of bisection. To multi-sect, say, the $x$-dimension into $P_l$ parts, the initial cut lines are spaced evenly between the minimum and maximum $x$-coordinates of the input data. The weight of points in the parts between pairs of adjacent cut lines is computed, and the cut lines are moved in order to adjust the weight in the parts. This process iterates until the parts all have the same weight.

Second, to improve scalability, MJ has several options that allow trade-offs between computation and data migration. MJ has the ability to avoid entirely the data migration costs that are inherent in parallel RCB. All processors cooperate to

compute cuts at all levels of the recursion; subcommunicators are not used as in RCB. This mode is illustrated in Figure 3.1(middle), where the horizontal cuts in each subdomain are computed in sequence by all processors. Alternatively, MJ can migrate data to intermediate subdomains as in RCB to allow concurrent computation of cuts during the recursion; in Figure 3.1(right), the horizontal cuts are computed concurrently in subcommunicators. Each mode has advantages, depending on the distribution of the input. In addition, MJ can perform "smart" migration, switching between modes based on the imbalance of the partitioner, predictions of the number of global reductions needed during partitioning and estimations on the future communication-computation bounds. We describe below the basic MJ algorithm, and then discuss variations to compute cuts lines of multiple parts and migrate data.

Third, unlike Zoltan's RCB which uses only MPI for parallelism, MJ is implemented using a hybrid MPI+OpenMP paradigm. This implementation allows MJ to fully exploit multicore architectures to provide dynamic load balancing to applications running with MPI+threads.

### 3.2.1 The Basic MJ Algorithm

Algorithm 1 gives the overall description of the MJ algorithm. In order to avoid data movement, MJ maintains a permutation array ($Permute$) of length $n$, where $n$ is the number of input coordinates on the local processor. Coordinate data is not rearranged during partitioning; instead, $Permute$ stores the indices of the coordinates,

and part assignments are made by reordering $Permute$. $xPerm$ maintains the beginning and end indices of each part in $Permute$. Initially, there is a single part with all coordinates. Then, $P$ (an array of number of parts required in each dimension) is traversed, and the dimension on which the partitioning will occur ($i$) is determined in a round-robin fashion. 1DPart is then called for all available parts, i.e., for $\kappa$ which is initially 1, and partitioning information is stored in $\mu$. updatePerm (not shown) updates the $Permute$ array, and stores the beginning indices of new parts in $newxPerm$ according to the part assignment information. Once all parts are partitioned along a dimension, the number of parts increases by factor of $P_l$. At this point, the algorithm estimates whether migrating the coordinates is good for the execution time or not by using CheckAndMigrate where, if needed, the actual migration operation is also performed and the sub communicators are created. After this point, the partitioning continues in the next dimension.

The pseudocode of 1DPart is given in algorithm 2. 1DPart finds $P_l$ part assignments with $\varepsilon$ imbalance for the given single dimensional coordinates associated with weights $W$. As the first step, getInitial (not shown) finds the minimum and maximum coordinate together with the total weight of the part $(C_i^{min}, C_i^{max}, W_{tot})$ by using a single parallel reduction reduceAll. Initial cut coordinates are assigned as uniform slices between $C_i^{min}$ and $C_i^{max}$. Moreover initial parts are assigned with following formula:

$$\mu_j = \left\lfloor (C_{i,j} - C_i^{min}) \times \frac{P_l}{C_i^{max} - C_i^{min}} \right\rfloor$$

**Algorithm 1:** Parallel MULTI-JAGGED Algorithm (MJ)

```
   Data: d, n, C_{d,n}, W_n, ε, d', P_{d'}
   for j from 0 to n − 1 do
       Permute_j ← j; // Initialize permutation
   κ ← 1; // All of C is in one part
   xPerm_0 ← 0;
   xPerm_1 ← n;
   for l from 0 to d' − 1 do
       i ← l mod d; // Dimension to partition
       // Compute P_l parts within each current part
1      for k from 0 to κ − 1 do
           pBegin ← xPerm_k;
           pEnd ← xPerm_{k+1};
           μ ← 1DPART(C_{i,*}, W, Permute, pBegin, pEnd, ε, P_l);
           newxPerm ← UPDATEPERM(Permute, pBegin, pEnd, μ);
       κ ← κ × P_l;
       CHECKANDMIGRATE (n, C_{*,*}, W, μ, newxPerm, κ);
       xPerm ← newxPerm;
```

1DPART maintains two arrays ($hiBounds, loBounds$) to store the upper and lower coordinates bounds of each cut. The information whether a cut position is finalized or not is saved in *done* array. After initialization, the iteration at line 1 continues until all cut positions are fixed. In every step, the part weights ($pWeights$) are calculated by calling GETPARTWEIGHTS; then according to this part weight information, GETNEWCUTS either fixes a position of a cut, or makes a new estimate for the new cut position.

The most computationally expensive portion of our algorithm is deciding the part to which all coordinates belong. This is a O($n$) operation in RCB. However, it is

**Algorithm 2:** 1DPart

```
Data: C_i, W, Permute, pBegin, pEnd, ε, P_l
// Get initial weights, min, max across all parts;
// requires one reduceAll operation
(cuts, μ, W_tot, C_i^min, C_i^max) ← getInitial(C_i, W, Permute, pBegin, pEnd, P_l);
cutCnt ← P_l − 1;
for k from 0 to cutCnt do
    hiBounds_k ← C_i^max;
    loBounds_k ← C_i^min;
    done_k ← False
leftCutCnt ← cutCnt;
// Find P_l − 1 cut positions
1 while leftCutCnt > 0 do
    pWeights ←getPartWeights(C_i, W, Permute,
                pBegin, pEnd, cuts, cutCnt, μ); (cuts, cutReduction) ←
    getNewCuts(W_tot, pWeights, ε, cuts, cutCnt,
          hiBounds, loBounds, done); leftCutCnt ← leftCutCnt − cutReduction;
return μ;
```

a $O(n \log(P_l))$ operation when using multi-section as each point has to be placed in one of the $P_l$ parts, which can be done in a binary-search-like fashion. Even though computation cost is slightly increased, finding multiple cuts at once in each iteration reduces the needed communication cost, hence yields faster execution. Algorithm 3 outlines getPartWeights. getPartWeights initializes the part weights of non-fixed parts to 0. It computes the part assignment for all coordinates by traversing all of the coordinates and comparing them to the cut lines. To reduce the computational cost even further, the part information from the previous iteration is used as an initial part estimate for the binary search. Once the correct part is found, the part weight and part assignment of the coordinate are updated accordingly. If the left and right

cut lines of the part in which the coordinate lies are fixed, no further calculation is done for the coordinate. During these iterations, we also keep the left and right closest coordinates to each cut line. This detail is omitted in the psuedocode for simplicity. This information is used later in GETNEWCUTS to be able to skip any huge holes in the input domain.

Once the part weights are known in each process, a prefix-sum operation is performed on the $pWeights$ and a single REDUCEALL operation is used to compute the global part weights. MJ can then evaluate each cut position independently of other cut positions; each cut can be evaluated by comparing the weight on its left and right.

Once the global part weights are known, the GETNEWCUTS function (Algorithm 4) determines whether the cut positions are final. If the cuts are to be moved, it makes a new guess for the new cut coordinates. GETNEWCUTS traverses all the cuts and skips the ones that are already finalized. For cuts that are not finalized it computes the expected weight ($ew$) and the imbalances for the left and right of the cut ($li, ri$). If they are both smaller than $\varepsilon$, the position of the cut is fixed. Otherwise, a better upper or lower bound is computed and the new cut position is estimated using them.

In order to compute a better upper or lower bound, let us consider the case where the weight on the left side of a cut is less than the expected weight; the operations are just flipped when the weight on the right side is less than its expected weight. When the weight on the left side of a cut is less than the expected weight, the lower

**Algorithm 3:** getPartWeights

```
Data: C_i, W, Permute, pBegin, pEnd, cuts,
      cutCnt, done, μ
for k from 0 to cutCnt do
    if not(done_k and done_{k-1}) then
        pWeights_k ← 0;
for j from pBegin to pEnd − 1 do
    j ← Permute_j;
    k ← μ_j;
    if (done_k and done_{k-1}) then
        continue;
    uc ← cutCnt − 1;
    lc ← 0;
    while uc ≥ lc do
        distance ← C_{i,j} − cuts_k;
        if distance ≤ 0 then
            onRightSide ← False;
            leftCutDistance ← C_{i,j} − cuts_{k-1};
            if leftCutDistance > 0 then
                break;
            uc ← k − 1;
        else
            onRightSide ← True;
            rightCutDistance ← C_{i,j} − cuts_{k+1};
            if rightCutDistance ≤ 0 then
                break;
            lc ← k + 1;
        k ← (uc + lc)/2
    if onRightSide then
        pWeights_{k+1} ← pWeights_{k+1} + W_j;
        μ_j ← k + 1;
    else
        pWeights_k ← pWeights_k + W_j;
        μ_j ← k;
for k from 1 to cutCnt do
    pWeights_k ← pWeights_{k-1} + pWeights_k;
reduceAll(pWeights);
return pWeights;
```

bound for where the cut could be is set to the cut's position, as that position is our best estimate so far. In order to compute a better position for the cut, the cuts on the right side of the current cut are traversed. If a right-side cut ($cuts_{k'}$) with an equal weight to the expected weight is found, the new cut coordinate is set to $cuts_{k'}$. If the weight of a right-side cut is greater than the expected weight while it is less than the current upper bound, the upper bound is tightened to this position. When the weight of a right-side cut is lower than the expected weight, it is a better lower bound than before.

After the new upper and lower bounds are determined, the function estimates a new position for the cut. While doing that, the weights of the upper and lower bounds are used (omitted in the algorithm because of the simplicity), and an estimation is done by assuming the uniform distribution of the weights between the upper and the lower bounds. If the new estimated cut position is same the previous cut position, the imbalance tolerance cannot be achieved; therefore, the cut coordinate is finalized. This situation occurs when there are coordinates with a total weight more than unit weight exactly on the cut line position. To achieve a balanced partition, some of the coordinates must be placed on the left side of the cut, while the rest of the coordinates must be placed on the right side.

**Algorithm 4:** GETNEWCUTS

```
Data: W_tot, pWeights, ε, cuts, cutCnt,
      hiBounds, loBounds, done
cutReduction ← 0;
for k from 0 to cutCnt − 1 do
    if done_k then
        continue;
    ew ← (k + 1) ∗ W_tot/(cutCnt + 1);
    (li, ri) ←IMBALANCEOF(pWeights_k, k, W_tot);
    if li < ε and ri < ε then
        done_k ← True cutReduction ← cutReduction + 1;
        continue;
    else
        if pWeights_k < ew then
            loBounds_k ← cuts_k;
            for k′ from k + 1 to cutCnt − 1 do
                if pWeights_k′ = ew then
                    newCuts_k ← cuts_k′;
                    break;
                else if pWeights_k′ > ew then
                    if hiBounds_k > cuts_k′ then
                        hiBounds_k ← cuts_k′;
                    break;
                else
                    loBounds_k ← cuts_k′;
        else
            hiBounds_k ← cuts_k;
            for k′ from cutCnt − 1 to k + 1 do
                if pWeights_k′ = ew then
                    newCuts_k ← cuts_k′;
                    break;
                else if pWeights_k′ < ew then
                    if loBounds_k > cuts_k′ then
                        loBounds_k ← cuts_k′;
                    break;
                else
                    hiBounds_k ← cuts_k′;
        newCuts_k ←GETNEWCUT(loBounds_k, hiBounds_k);
        if newCuts_k = cuts_k then
            done_k ← True cutReduction ← cutReduction + 1;
    return (newCuts, cutReduction);
```

### 3.2.2 Migration of the Coordinates

After a partition along a dimension is completed, the coordinates in each part can be localized in a smaller number of processors by performing migration. However, the volume of the migration operation is linear with the total number of coordinates; hence, the migration might become a bottleneck for the partitioning algorithm. Therefore, MJ tries to avoid migration. Note that when MJ does not migrate data, all MPI processes participate in computing the position of each cut. This requirement might reveal two problems.

**Load Imbalance:** MPI processes might have different numbers of coordinates in parts computed during the execution of MJ. Therefore, the MPI processes will have different loads in further partitioning of these parts. The worst case occurs when an MPI process does not have any points in a part. Even though the process does not have any points, it will participate in the computation and communication. Taking such considerations into account, CheckAndMigrate calculates the imbalance of the processors in each part. The function performs a single reduceAll to find the total number of coordinates in each part ($N_{*,*}$), and another reduceAll to calculate the imbalance which is defined as:

$$imbalance \ = \frac{\sum_{i=0}^{\rho} \sum_{j=0}^{\kappa} \frac{|N_{avg,j} - N_{i,j}|}{N_{avg,j}}}{\rho \times \kappa}$$

where $\rho$ is the number of processors, $N_{avg,j} = \frac{N_{*,j}}{\rho}$ is the average number of coordinates over the processors in part $j$, and $N_{i,j}$ is the number of local points of processor $i$ in part $j$. The minimum value for imbalance is 0, and as the imbalance of the

processors over parts increases this value also increases. Note that, if MJ does not perform migration, all subsequent partitions are computed one after the other, with each subsequent partition taking less time if the work is balanced. As the imbalance increases, the cost of subsequent partitioning will also increase and, hence, yield slower execution. On the other hand, as the imbalance increases, migration becomes cheaper, as it suggests that the data has already been localized, which reduces the volume of the migration. Therefore, MJ performs migration only when this imbalance value is greater than a specified threshold. We choose this threshold empirically as 30% as a result of our initial experiments.

**High Number of Global Messages:** In a single call of the 1DPART function, the number of global messages $m = 1 + it$, where $it$ is the iteration count of the loop in line 1 of Algorithm 2. Assuming that $it'$ is the average iteration count, and $t$ is the total number of calls to 1DPART, the total number of messages becomes $t \times (it' + 1)$, in which $t$ can be computed as follows:

$$t = 1 + \sum_{l=0}^{d'-2} \prod_{l'=0}^{l} P_{l'}$$

For example, while partitioning 3D space into $32 \times 32 \times 32 = 32,768$ parts, $t = 1057$. Although this is a small number, it might become problematic as the number processors and the number of partitioning dimensions $d'$ increases. Therefore, throughout its execution, MJ maintains the number of future REDUCEALL operations $red_{opt}$, and performs migration when $red_{opt} * \rho$ is greater than a specified value. We choose this threshold empirically as $1.5M$ as a result of our initial experiments.

**Communication Bounded Last Dimension Partitioning:** Even if the imbalances of the processors are very low, avoiding migration might cause problems during the last dimension partitionings, since last steps may become communication bounded. Note that there are $P^{d'-1} = \prod_{l=0}^{d'-2} P_l$ parts at the beginning of the last step. In the best case (when there is no imbalance between the processors), each processor owns $N' = \frac{N}{P^{d'-1}}$ in each part. Processors will perform a REDUCEALL operation after comparing $N'$ coordinate againsts $P_{d'-1} - 1$ cuts, and this will be repeated for all $P^{d'-1}$ parts. If $N'$ is a very small number, this will make the execution heavily communication bounded (For example, when $N = 500K$, and $K = 32 \times 32 \times 32$, each processor will own 500 coordinates in their parts at the beginning of last dimension (best case)). Therefore, MJ keeps track of $N'$ during its execution, and forces to migrate coordinates if it is smaller than $1K$.

The algorithm of the CHECKANDMIGRATE initially checks the above three conditions, and decide whether to do migration or not. When doing the migration, there are two cases that might occur. When $\rho < \kappa$, one processor can be given several parts. To minimize migration volume, a part is greedily assigned to the available processor with the largest number of coordinates in that part. When $\rho > \kappa$, several processors will be assigned to a single part. As before, the processors are greedily chosen according to their availability and the number of coordinates they have in that part. Since there are several processors assigned to a single part, the processors will perform communication operations in the further iterations. Therefore, another idea

is to assign a part to the set of processors that are close to each other. Although this might increase the volume of the migration, it might be beneficial since it decreases the cost of the future communication operations. However, the performances of these variants are architecture dependent. Although MJ supports both of these processor assignment heuristics, we only use the first one in the experiments of this chapter, since the cost of the data movement on the second one is more expensive and the reduced communication time for the further iterations does not amortize this cost on the architecture used in the experiments of this chapter.

After the processors determines the which processor it should send its data, the migration operation is performed, and the processors obtain their new local coordinates. A part might be assigned to a single processor, and the processors might own more than one part. In this case, further partitionings of these parts are executed sequentially by the owner processor. On the other hand, if a part is assigned to a set of processors, a smaller sub communicator is created and the further partitioning of that part is performed in this smaller subset of processors. Depending on the number of parts assigned to the processor the function updates $newxPerm$, $xPerm$, $\kappa$, and finishes its execution.

### 3.2.3 Partitioning into Arbitrary Part Numbers

For simplicity reasons, section 3.2.1 explains a limited partitioning into $\kappa = P_0 \times P_1 \times \ldots \times P_{d'-1}$ parts. However, MJ algorithm does not necessarily require

$\kappa$ to be multiple of integers, instead $\kappa$ can be any number, possibly a prime number. Therefore, MJ can be defined in a more general way: MJ partitions the set of coordinates into a desired number of parts ($\kappa$) in a given number of steps called the *recursion depth* ($d'$). For example, the leftmost partitioning in Figure 3.2 can be considered as MJ partitioning into $\kappa = 16$ in $d' = 4$ steps, while the other two figures can be considered partitioning into $\kappa = 16$ in $d' = 2$ steps.

When a possible prime $\kappa$ and $d'$ is given, MJ tries to choose multi-section counts in each dimension as close as possible. Firstly, MJ partitions the data into $P_0 = \lceil \kappa^{\frac{1}{d'}} \rceil$ along the first dimension. Then, for each of the parts that will be obtained at the first level, the future number of partitions is calculated, and this is assigned to part as its expected weight. For example, Figure 3.2 shows an example of MJ used to partition a dataset (with total area -weight- as 460) into $\kappa = 23$ parts in $d' = 2$ steps. Initially, MJ determines the number of sections that will be applied to partition the data along the first dimension, that is $partArray_0 = \lceil 23^{1/2} \rceil = 5$. Then, MJ determines how many future partitions will be obtained from each part. In this case, the first three part will be partitioned into 5, while the last two parts are found to be partioned into 4. The weights for the partitons are adjusted according to the future number of partitions, and MJ multisects the data that yields a partioning into 5 parts with areas (weights) 100, 100, 100, 80 and 80, respectively. After the first partitioning, $d'$ is decremented by 1, and the future number of partitions is used to determine the section count along

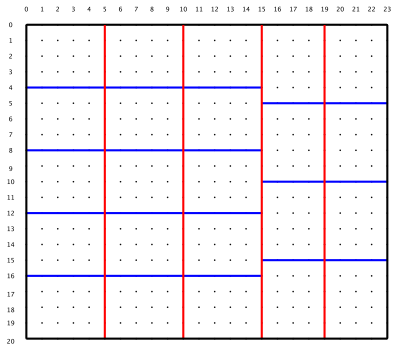


Figure 3.2: A partitioning using MJ with $\kappa = 23$ $d' = 2$.

the second dimension. The first three parts the last two are partitioned into 5 and 4 parts, respectively, yielding a partitioning into 23 with each part area is 20.

### 3.2.4 Properties of partitions from Multi-Jagged

In this subsection, we will reuse some definitions used to analyze the properties of RCB [16]. The definitions are restated here for clarity. Each partition line in Figure 3.1 is subdivided into a number of *segments* by the incidence of other partition lines. For example, the first (red) cut line using RCB in Figure 3.1(left) is subdivided into seven segments. The *graph of a partition* is the dual graph obtained by representing each part by a vertex, and adding an edge between two nodes if and only if the corresponding regions share a segment.

For the analysis of spatial partitions, assume that each part is assigned to different processors. As a result, there are two metrics to analyze the quality of the partition: the maximum degree of a vertex in the partition graph and the total number of edges

in the partition graph. The analysis in this section is restricted to partitioning in two dimensions $p \times q$, even though the multi-jagged algorithm works for arbitrary number of dimensions. To simplify the discussion, we will call the regions corresponding to parts in the first dimension as *stripes*. For example, Figure 3.1(middle) has four vertical stripes. Each stripe is further partitioned into $q$ parts with $q-1$ cut lines.

The upper bound for the maximum degree of a vertex in the partition graph is $2 \times q + 2$. This arises from the case when for any one part in stripe $i$ $(1 < i < p)$ shares its perimeter with all the $q$ parts of each of the two neighboring stripes $i-1$ and $i+1$. This has been observed previously in Manne [106]. If $p = q = 2^{k/2}$ for computing $2^k$ parts, the upper bound for the maximum degree of a vertex in the partitioning graph is $2 \times 2^{k/2} + 2$ which is worse than the upper bound for RCB which is $2^{k/2} + 2^{k/2-1} + 3$ (when $k$ is even and $k \geq 4$). The difference between the two upper bounds is $2^{k/2-1} - 1$; however, we will show that MJ and RCB behave similarly in practice. When $k$ is odd, we can reach the same upper bound by using $p = 2^{\lfloor k/2 \rfloor + 1}$ and $q = 2^{\lfloor k/2 \rfloor}$. Instead of using $p = q$ we can increase $p$ to obtain a better bound in practice.

The total number of edges in the partition graph can be derived using a simple constructive argument. Within any one stripe $i$, there are $q$ regions (or $q$ vertices in the partition graph) and $q-1$ cut lines (or $q-1$ edges in the partition graph). There are $p$ such stripes, resulting in $p \times (q-1)$ edges corresponding to the cut lines in the second dimension (A).

The number of edges corresponding to the cut lines in the first dimension is the number of segments that could be created by joining $p$ stripes. Consider the case with two stripes ($p = 2$). The least number of segments ($q$) on the cut line between the two stripes is obtained if for any $j, 1 \leq j \leq q$, the $j^{th}$ cut lines inside both stripes are collinear. Moving any one of the cut lines in the second dimension up/down increases the number of segments by at most two. There are $q - 1$ such cut lines, which gives us $2 \times (q - 1)$ as the maximum number of segments between two stripes. There are $p - 1$ such cut lines (or $p$ stripes) giving us $(p - 1) \times (2 \times q - 1)$ segments. Adding the number of segments from the second dimension(A), the upper bound for the total number of edges in the partition graph is $p \times (q - 1) + (p - 1) \times (2 \times q - 1)$. When $p = q = 2^{k/2}$, the upper bound is $3 \times 2^k - 2^{k/2+2} + 1$, the same as the upper bound for RCB (when $k$ is even).

## 3.3 Experimental results

We evaluated the performance of the proposed MJ algorithm on various real and synthetically generated datasets. We compared the run-time and the quality of the partitioning results with RCB algorithm.

All experiments were carried out on the Hopper supercomputer, a Cray XE6 at the National Energy Research Scientific Computing Center. Each compute node on Hopper has two twelve-core AMD "MagnyCours" processors (24 cores per node), running at 2.1GHz, with 32 GBs of memory (slightly more than 1 GB of memory per core). Hopper has a Cray "Gemini" interconnect for internode communication.

We used Zoltan and Zoltan2 [75] to test the RCB and MJ algorithms, respectively. Zoltan (hence, RCB) is implemented in C; Zoltan2 (hence, MJ) is in C++. We used the gcc compiler (version 4.7.1) with -O3 optimization flag. For the MPI library, we used Open MPI (version 1.4.5).

The experiments were run on four synthetic datasets and three real datasets. The properties of the datasets are given in Table 3.1. `Uniform` and `Normal` are 2D datasets whose coordinates are randomly distributed with uniform and normal distribution, respectively. `2D ANorm` is generated with absolute normal distribution, where the dataset contains a circle hole (Fig. 3.3). `3D Anorm` is obtained with the application of `2D ANorm` function in three-dimensional space. Two of the real datasets (`huge-bubbles` and `europe_osm`) are from the University of Florida sparse matrix collection [55]. The `tetraMesh` dataset is a real 3D dataset representing the artic ice sheet modeling of Greenland [91].

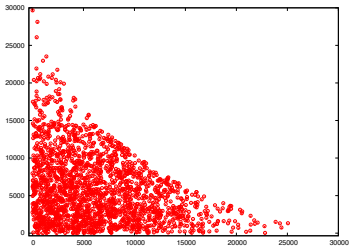



Figure 3.3: 2D Anorm dataset: Absolute normal distribution with a hole.

Table 3.1: Properties of datasets for the experiments.

| #Name | # Coordinates | $d$ | Sequential RCB Time (s) |
|---|---|---|---|
| Synthetic Datasets | | | |
| `Uniform` | 4M/proc or 50M | 2 | 95.958 |
| `Normal` | 4M/proc or 50M | 2 | 114.869 |
| `2D ANorm` | 4M/proc or 50M | 2 | 114.252 |
| `3D ANorm` | 4M/proc or 50M | 3 | 120.155 |
| Real Datasets | | | |
| `hugebubbles-0020` | 21,198,119 | 2 | 35.694 |
| `europe_osm` | 50,912,018 | 2 | 96.762 |
| `tetraMesh` | 94,732,680 | 3 | 102.005 |

### 3.3.1 Weak Scaling

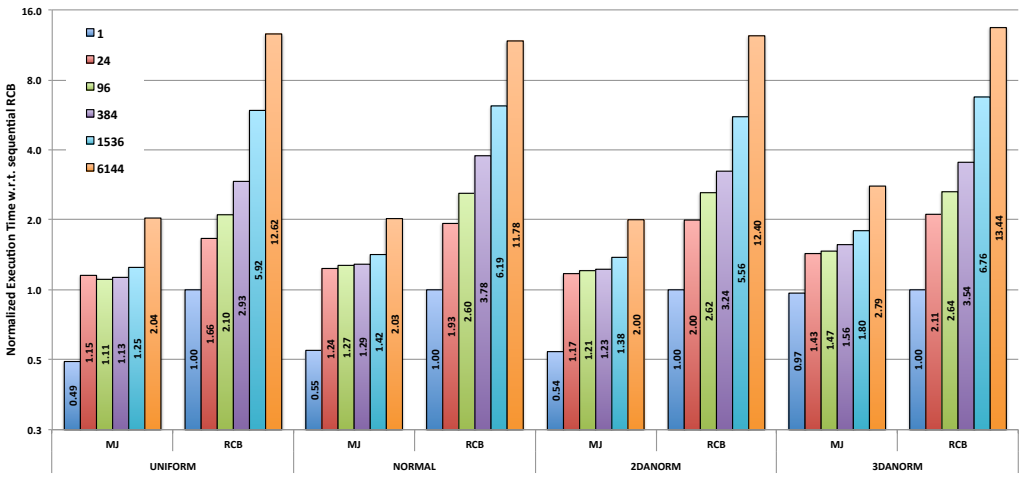



Figure 3.4: Weak scaling results on synthetic datasets with 4M points per process. The points are randomly generated and scattered all over the space for all processors.

Weak scaling experiments were performed using the synthetic datasets. Every processor was given 4M coordinates, and the number of processors was increased up to $6,144$. Since the number of points per process, and hence, the work, was kept constant, the ideal result would be a flat line. We expect some small increase in the execution time with the increase in the number of processes used. In this experiment, 2D datasets were partitioned into $65,536$ parts, whereas the 3D dataset was partitioned into $32,768$. MJ partitioned the space into $256 \times 256$ and $32 \times 32 \times 32$ parts in the 2D and 3D datasets, respectively.

The execution time of the partitioning algorithm highly depends on the initial distribution of the data. Thus, we have studied weak scaling on datasets with different distributions of the coordinates over the processors. For example, Figure 3.4 shows the weak scaling results with the execution time of the algorithms normalized with respect to the sequential run-time of RCB on the corresponding dataset. For these generated datasets, each processor generates points across the entire domain according to the specified distribution; thus, each processor has points from across the entire domain. Therefore, after partitioning along a dimension, if further partitionings are to be localized via migration, the migration volume becomes very large. On the other hand, if localization is avoided, and further partitionings are also done with all processors, the workloads of the processors become close to each other, since the processors have close number of coordinates along the parts.

As seen in the figure, execution time of RCB algorithm drastically increases with the increasing number of processors due to the cost of migration, since it performs migrations regardless of the data distribution. Since the points owned by a processor can be from any region of the domain, a significant portion of the data (roughly half) needs to be migrated between subsets of processors after each bisection. On the other hand, in most of these instances, MJ tries to avoid migration as it is predicted to be expensive. As seen in the figure, there is a difference between MJ's sequential performance and parallel performances, but the increase of MJ's parallel execution times is much lower than RCB. Note that, even though the migration is expensive in these datasets, MJ performs migration starting from 6144 (1536) processors in these experiments, in order to avoid $256 \times it'$ ($1024 \times it'$) REDUCEALL operation among large number of processors on 2D (3D) datasets. Although performing migration on such big set of processor in these datasets harms the performance, the cost of the migration is not as high as in RCB, since it is performed only once. Even in the 3D datasets, the migration is performed only after the first partitioning along the first dimension, which reduces the migration cost w.r.t. RCB.

Figure 3.5 shows weak scaling results with the execution time of the algorithms for the same datasets for which processors have localized coordinates. In this experiment, the processors generate their coordinates randomly, then, unlike the previous experiment, the RCB algorithm is used to partition the datasets into the given number of processors. All of the coordinates that are found to be in the same part by this initial

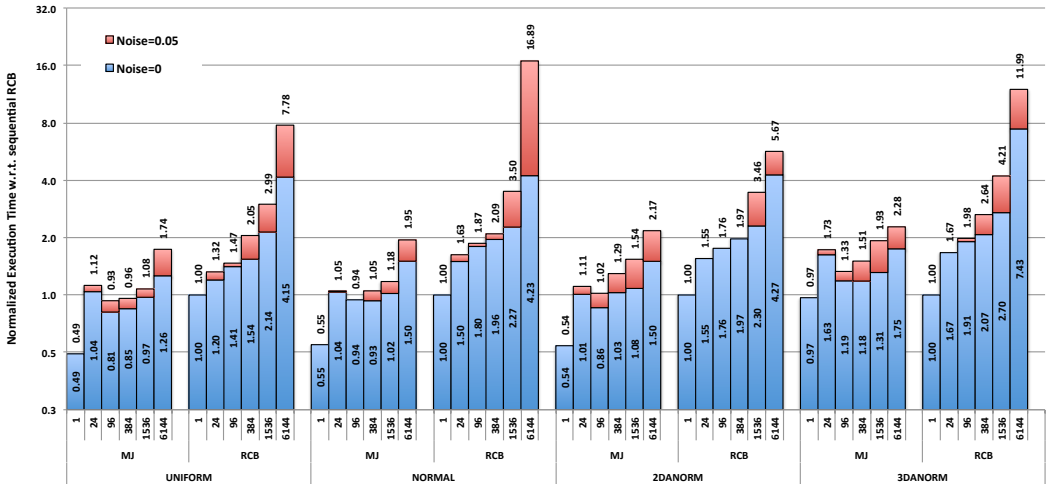



Figure 3.5: Weak scaling results on synthetic datasets with 4M points per process. The points are randomly generated then prepartitioned using RCB algorithm into given number of processors. Each processors owns a unique area in the space.

RCB partitioning are migrated to a single processor. After this pre-partitioning MJ and RCB is called to partition the dataset into the $32,768$ or $65,356$ parts, as in previous experiment. Therefore, at the beginning of the execution, each processor has localized coordinates that are in between discrete intervals. Note that, the algorithms may still need to migrate some of their coordinates. Basically, MJ might require migration of the some of the local coordinates, since pre-partitioning is done with RCB. On the other hand, RCB might require migration since the pre-partitioning and the actual partitioning have different number of target parts.

In this experiment, we also studied the effect of perturbation of the input after pre-partitioning step. In the Figure 3.5, the given noise value refers to the percentage of perturbed points in each processor. If the noise value is given as 0.05, this means that 5% of the local data points in each processor (after pre-partitioning) is regenerated using uniform distribution. Each processor chooses the intervals for its uniform distribution by doubling the minimum and maximum coordinate distance to the processor centroid. In Figure 3.5, $Noise = 0$ represents the execution time of the algorithms after the pre-partitioning, while $Noise = 0.05$ represents how much the execution time increases when the data perturbation is applied. Adding noise changes the total execution time, because it may either change the communication time (an increase in the migration volume), or the computation time (changes global coordinate intervals which results differences in the execution time to find the cut position).

In this experiment, the processors have localized data, and the migration volume is expected to be lower than the previous experiment. Since the processors have localized data, the workloads of the processors over the parts are uneven. Therefore MJ algorithm chooses to perform migration in all instances in this experiment, including the ones with noise. As seen in Figure 3.5, we observe that the execution times of algorithms increase with the increasing number of processors. MJ performs better than RCB in most of the instances, although the pre-partitioning is done with RCB

and the migration volume of MJ is expected to be larger than that of RCB algorithm. This is mainly because the smaller number of migration occurrences in MJ. For example, on 2D datasets, the migration is performed only once after partitoning the dataset into 256 parts, regardless of the number processors used. The migration cost of MJ increases with the increasing number of processors, because the migration volume and the processors participating to the migration increases. In addition to these, for RCB, the number of migration occurances also increases with the increasing number of processors in RCB. Therefore, a slight increase in the migration volume drastically increases the cost of the migration in RCB. We also observe this when a noise is added to the datasets. For example in Figure 3.5, when 5% noise is added to the 3D Anorm dataset on $6,144$ processors, the execution time of RCB increases to 92.08 seconds from 57.11 seconds, while the execution time of MJ increases to 17.50 seconds from 13.42 seconds. Therefore, RCB experience a drastic increase in the execution time on high number of processors with the addition of noise, while its effect is much lower for MJ algorithm.

A surprising result in this figure is that, there exists a drastic decrease in the MJ's execution time after 24 processors. Note that this is because of the imbalance of the workload among the processors. 2D (3D) datasets are partitioned into 256 (32) along the first dimension, and after the migration, 16 (8) of the processors are assigned to partition 11 (2) of the result parts, while 8 (16) of the processors are assigned to 10 (1) parts. Therefore, after a very early step, the processors have different workloads,

which causes the execution time to increase because of the heavier loaded processors. This is not a case as the processor number increases, because either the workloads are equal among the processors, or they become different at later steps.

In the weak scaling experiments, we observe that the execution time of both MJ and RCB are smaller when the data is pre-partitioned for almost all instances compared to their counterparts in Figure 3.4. This is mainly due to the fact that the localization via migration makes the algorithms faster for the further steps, and this localization can be done in lower costs due to the lower migration volume in the latter experiment. There are several instances for which the algorithms run slower on the experiment where the data is pre-partitioned with 5% noise. For example, MJ performs migration in both experiments on 2D Anorm Dataset with $6,144$ processors (or RCB on Normal Dataset). Even though the migration volume of the former experiment is smaller, the execution time for the latter one with noise is observed to be higher. Note that, these results are because of the increase in the computation times in the latter ones, since with the addition of noise, the global intervals changes, and the computation time to find the cut positions increases.

### 3.3.2 Strong scaling

Strong scaling experiments were carried out for all synthetic and real datasets in Table 3.1. The synthetic dataset size was chosen as 50M points, and the scaling experiments were done up to 1,536 processors. As with the weak scaling experiments, 2D datasets were partitioned into $65,536$ ($256 \times 256$ for MJ) parts and 3D datasets

were partitioned into $32,768$ ($32 \times 32 \times 32$ for MJ). Figure 3.6 shows the strong scaling results with the speedup with respect to sequential RCB for the datasets generated with the specified distribution, and each processor has points from across the entire domain as in the datasets presented in Figure 3.4 in weak scaling experiments. The execution times of sequential RCB for strong scaling experiments are given in Table 3.1.

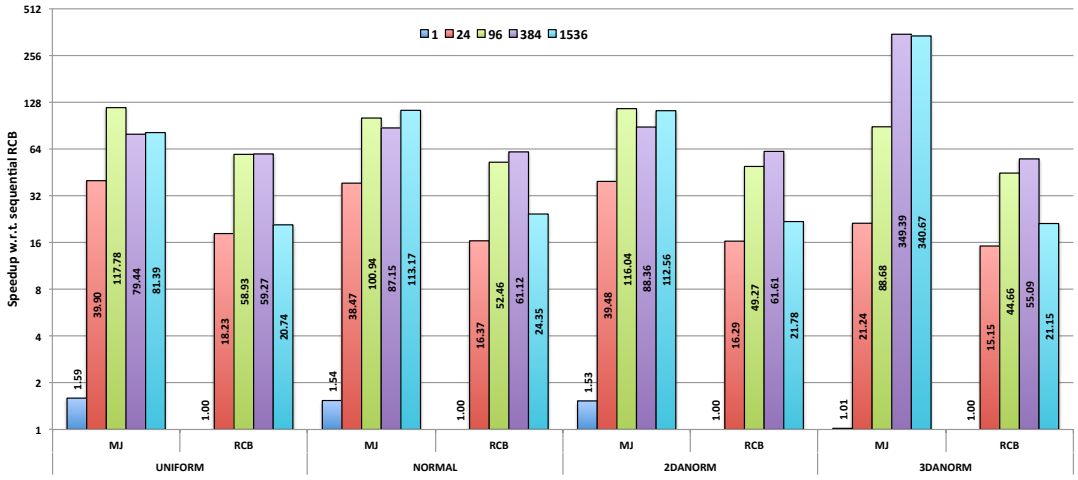



Figure 3.6: Strong scaling results. The points are randomly generated and scattered all over the space for all processors.

The first observation is that in all instances, MJ obtained higher speedups w.r.t. RCB. This is again due to the costs related to the migration of the coordinates as explained in weak scaling experiments. In all datasets, RCB scales well up to 384

processors, starting from $1,536$ processors the partioning of RCB become communication bounded. On the other hand, the strong scaling results of MJ shows a different behaviour on 2D datasets. MJ obtains increasing speedups upto 96 processors, then the speedup drops slightly for 384 processors, and it increases again for $1,536$ processors. Note that, because of the distribution of the data, MJ tries to avoid migration in these datasets. However, increasing number of processors reduces the work of each processor, and makes the partitioning more communication bounded. For example, when partitioning Uniform Datasets with 384 processors, each processors initially owns $130K$ coordinates. After partitoning into 256 along the first dimension, each part in each processor roughly has 500 coordinates. In the case of avoiding the migration, each processor will compare 500 coordinate against 256 parts, and a REDUCEALL operation will be performed all over the processors, which will be repeated for 256 parts found along the first dimension partitioning. This will make the partitioning communication bounded, and since 500 is lower than the cut-off ($1K$), MJ chooses to migrate the coordinates even though the migration volume is high. Therefore, the slight drop in the speedup at 384 processors is because MJ switches from the no-migration mode to migration mode. However, after 384 processor, the speedup increase continues for all 2D datasets. This switch between 2 modes of MJ occurs at 96 processors on 3D datasets, which is the reason for not observing this slight drop on the speedup. On the other hand, we observe higher speedups for 3D Anorm dataset on large number of processors. For example, on 384 and $1,536$ processors the

speedups of MJ on 3D Anorm dataset are much larger than the speedups obtained in 2D datasets. This is because less work is performed on 3D Anorm dataset, when the partitioning is communication bounded. On $1,536$ processors, each processor initially owns $32.5K$ coordinates, which makes the partitioning communication bounded. MJ spends more time on 2D datasets in this phase, as it performs more operations by partitioning 2D datasets into 256, while $3D$ datasets are partitioned to 32 in the first phase. Therefore, MJ obtains bigger speedup values on 3D Anorm dataset. We will explain this issue in more detail in the next set of experiments.

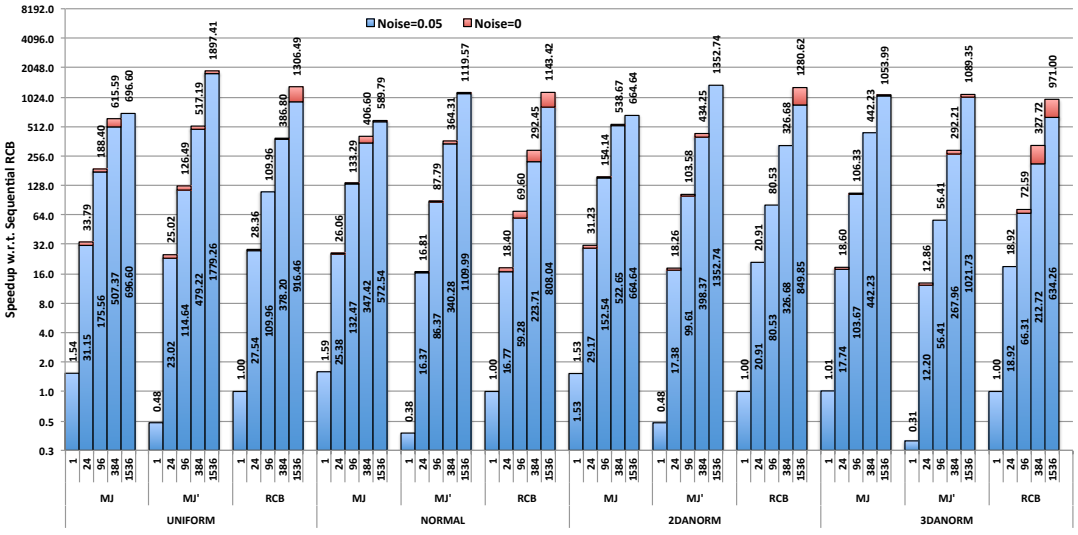



Figure 3.7: Strong scaling results. The points are randomly generated then prepartitioned using RCB algorithm into given number of processors. Each processors owns a unique area in the space.

Figure 3.7 shows the strong scaling results for the prepartitioned counterparts of the datasets. In these experiment, one more MJ variant (MJ') is added for the comparisons. The difference of MJ' is that, the recursion depth is doubled, that is MJ' partitions the 2D datasets into $16\times16\times16\times16$, and 3D datasets into $4\times4\times4\times8\times8\times8$ parts. Also, similar to the weak scaling experiments given in Figure 3.5, we also study how the speedups are affected by the addition of the 5% noise. In the charts, Noise=0.05 corresponds to the speedups obtained for the datasets perturbed with 5% noise. Noise=0 corresponds to the speedup decrease when the noise is added, and the total speedup for Noise=0 is found with the addition of the two values, which is printed on the top of each bar in the chart.

Initially, we observe that all algorithms scales upto $1,536$ processors and the speedups values are always higher than their counterparts in Figure 3.6, thanks to the pre-localized data. The addition of the noise is not as big problem as in the weak scaling experiments since the used number of coordinates is much lower ($50M$), and this limits the increase in the migration volume.

Secondly, we observe that the speedups obtained by MJ is larger than RCB and MJ' up to 384 processors in all instances. However, when the number of processors increases to $1,536$, MJ is outperformed by both RCB and MJ'. This is again because of high workload of MJ, when the execution is communication bounded. Like the previous experiment, MJ partitions the dataset into 256 parts, and when there are

$1,536$ processors with $130K$ coordinates, the execution is dominated by the communication. RCB and MJ' performs a small work in this phase by partitioning into 2 and 16, and then they create subcommunicators and further partitionings are performed by smaller processor groups. On the other hand, MJ partitons the dataset into 256 parts in this phase, therefore this phase, where the execution is dominated by the communication takes much longer for MJ. However, MJ' outperforms RCB in all instances with $1,536$ except Normal dataset where RCB is slightly faster. This suggests that when the workload of the processors are low, it is better to increase the recursion depth slightly.

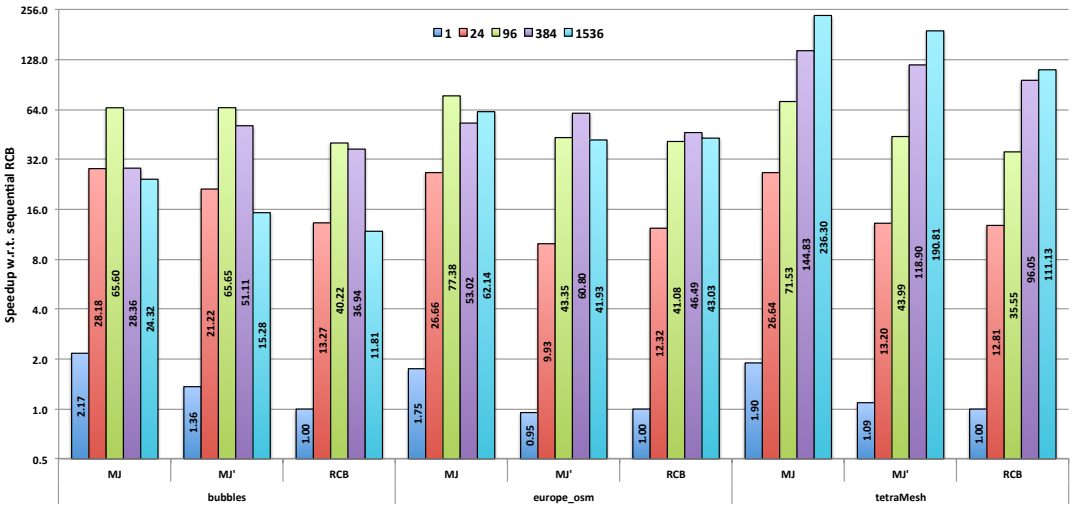



Figure 3.8: Strong scaling results on real datasets.

Figure 3.7 shows the strong scaling results for the real datasets for 2 MJ variant and RCB algorithm. Among the real datasets, the first two datasets are 2D, while tetraMesh is 3D. The algorithms are adjusted as in the syntehic experiments based on the dimension. Note that, these real datasets are read from the original input file in the way they are stored, and then, processors shares the coordinates by taking a portion of the consecutive coordinates. In this experiment, because of the way that the real datasets are stored, the distribution of the coordinates to the processors are close to localized, which is in between two extremes tested in the synthetic datasets. MJ and MJ' performs migration for all of these real datasets, in the first step. However, for the instances where the recursion depth is larger than 2, after the migration of the first partitioning, some processor groups performed migration while some avoid the migration. based on the imbalance calculations.

Huge-bubbles dataset is the smallest real dataset in this experiment. None of the algorithms scaled after 96 processors. In all instances of Huge-bubbles dataset, RCB is outperformed by MJ variants. Among the MJ variants MJ' outperforms MJ on 96 and 384 processors as explained before. However, MJ is faster than MJ' on $1,536$ unlike what we expect. We believe that this is because the gain of MJ' on the first phase does not amortize the repeatetive migration calls. This is similar to the reason why RCB is slower than MJ', that is, although RCB performs less work on the first phase, its recursion depth increases the communication cost. Similarly, for the europe dataset, RCB is outperformed in all instances by MJ variants. In this dataset, MJ'

is faster than MJ only on 384 processors. Similiarly on 1,536, the cost of repetative migrations becomes more expensive then the gain in hte first phase of partitoning, results in slower execution of MJ'. tetraMesh, on the other, hand is the biggest dataset that is used in the strong scaling experiments. All of the algorithms scaled upto 1,536 processors, in which MJ obtained the best performances in all instances.

### 3.3.3 Dynamic Partitioning Simulation

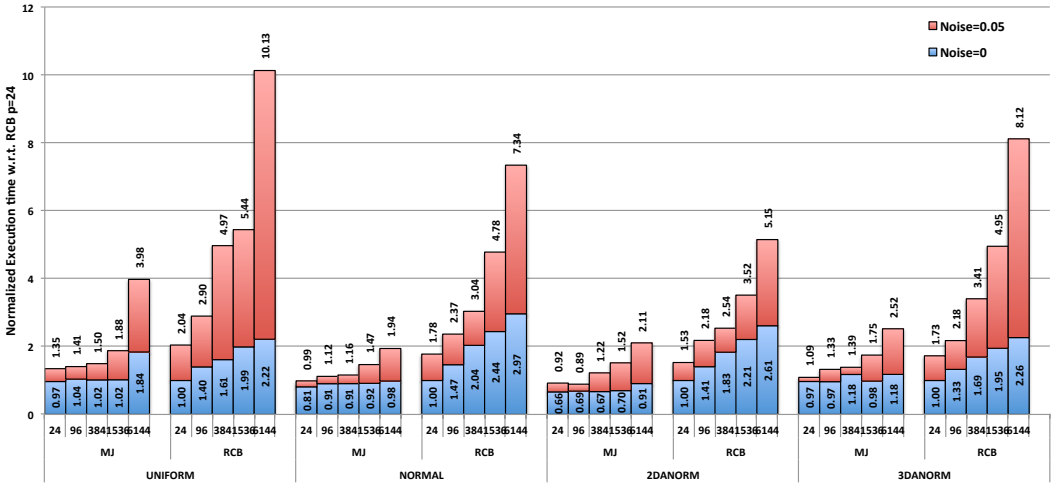



Figure 3.9: Simulation of dynamic partitioning.

In this experiments, we wanted to simulate the behaviours of the algorithms as dynamic partitioning algorithms. This experiment are carried out for all synthetic datasets. Every processor was given 4M coordinates, and the number of processors was increased up to 6,144. As different than the weak scaling experiments, the

target number of parts is set to the used number of processors in this experiment. Therefore, although the points per process is constant, the target number of parts, therefore the number of comparisons increases. For experiment, MJ is only provided the target number of parts, and it automatically adjusted the number of the cuts along the dimensions automatically, as mentioned in section 3.2.3. Since the data is already prepartitioned during the dynamic partitoning, we generate the data for each distribution then we prepartitioned the datasets, using MJ and RCB beforehand. Figure 3.9 gives the normalized execution time of the algorithms w.r.t. the execution time of RCB on 24 processors. In this experiment, for each of the algorithms, the data is localized with the prepartioning using the algorithms itself, hence migration volume becomes zero, when no noise is added. However, with the addition of the noise, the migration volume of the algorithms is increased.

Although the work of the processors increases with the increasing number of processors, we experience almost a flat line for MJ when there is no noise in Figure 3.9. We also observe that MJ is always faster than RCB for all instances. and it is affected by the addition of the noise much less than RCB.

Note that, the current RCB implementation provides a special mechanism for dynamic repartitioning. RCB keeps the cuts from the previous partitionings, and uses it as the guesses for the new cutlines at the next partitoning steps. To be fair, we do not provide this information to RCB in this experiment. However, even this information is provided to RCB, its behaviour would be similar to the one experienced

in Uniform datasets, since in Uniform dataset it takes very small time to find the cut positions.

### 3.3.4 Quality of partitioning

There are two metrics we can use to evaluate the quality of a partitioning: load-imbalance and communication costs for the application. As the data used in the experiments have unit weights, both MJ and RCB were able to find perfectly balanced partitions. Therefore, their partitioning quality in terms of load-imbalance is the same in these experiments. This is the goal of both algorithms and both do well in this metric.

In terms of communication overhead, we computed the maximum and total number of messages induced by computed partitions. Since each neighbor represents a communication message, we counted the number of neighbors of each part and reported the maximum number of messages and the total number of messages of the partitions found by RCB and MJ variants. Note that both RCB and MJ do not try to minimize maximum or overall number of messages metric directly. 2D datasets are partitioned into 256 ($16 \times 16$ and $4 \times 4 \times 4 \times 4$), 4,096 ($64 \times 64$ and $8 \times 8 \times 8 \times 8$) and 65,536 ($256 \times 256$ and $16 \times 16 \times 16 \times 16$), while 3D datasets are partitioned into 512 ($8 \times 8 \times 8$ and $2 \times 2 \times 2 \times 4 \times 4 \times 4$), 4,096 ($16 \times 16 \times 16$ and $\times 4 \times 4 \times 4 \times 4 \times 4 \times 4$) and 32,768 ($32 \times 32 \times 32$ and $\times 4 \times 4 \times 4 \times 8 \times 8 \times 8$). Table 3.2 and Table 3.3 give the results for 2D and 3D datasets, respectively. As seen in the tables, the overall number of messages of the partitions is similar for RCB and MJ variants. This is supported

Table 3.2: Maximum and Total number of Messages for 2D partitioning

| Name | 256 | | | | | | 4,096 | | | | | | 65,536 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Maximum | | | Total | | | Maximum | | | Total | | | Maximum | | | Total | | |
| | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB |
| `Uniform` | 8 | 8 | 8 | 1,410 | 1,410 | 1,410 | 8 | 8 | 8 | 24,066 | 24,066 | 24,066 | 8 | 8 | 8 | 391,166 | 391,170 | 391,166 |
| `Normal` | 8 | 8 | 8 | 1,410 | 1,410 | 1,410 | 8 | 8 | 8 | 24,066 | 24,066 | 24,066 | 8 | 8 | 8 | 391,170 | 391,168 | 391,168 |
| `2D ANorm` | 7 | 8 | 8 | 1,410 | 1,410 | 1,410 | 15 | 12 | 8 | 24,066 | 24,066 | 24,066 | 45 | 36 | 9 | 391,168 | 391,170 | 391,164 |
| `Huge-bubbles` | 13 | 10 | 11 | 1,410 | 1,410 | 1,410 | 20 | 19 | 17 | 24,016 | 23,952 | 23,746 | 30 | 74 | 50 | 385,966 | 377,702 | 362,456 |
| `europe_osm` | 13 | 12 | 9 | 1,410 | 1,410 | 1,410 | 25 | 29 | 15 | 24,066 | 24,066 | 24,066 | 109 | 90 | 47 | 391,110 | 391,134 | 391,136 |

Table 3.3: Maximum and Total number of Messages for 3D partitioning

| Name | 512 | | | | | | 4,096 | | | | | | 32,768 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Maximum | | | Total | | | Maximum | | | Total | | | Maximum | | | Total | | |
| | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB | MJ | MJ' | RCB |
| `3D ANorm` | 19 | 25 | 20 | 5,726 | 5,726 | 5,726 | 19 | 23 | 24 | 51,390 | 51,404 | 51,400 | 26 | 27 | 26 | 434,558 | 434,590 | 434,612 |
| `tetraMesh` | 20 | 21 | 20 | 5,726 | 5,714 | 5,734 | 44 | 28 | 23 | 51,328 | 51,070 | 50,606 | 112 | 68 | 44 | 433,044 | 428,592 | 415,100 |

by the theory in Section 3.2.4 as well. Even though the maximum number of messages of MJ is worse than RCB in some cases, it is much lower than the theoretical worst-case bounds. For example, for $65,536$ parts with $256 \times 256$ partitioning, the theoretical worst case maximum number of messages is $2 \times 256 + 2 = 514$, while the actual number is 109 or less in all our experiments. Moreover, in most of the cases, the maximum number of neighbors metric can be reduced by increasing the recursion depth, as MJ' has mostly smaller maximum number of neighbors than MJ.

## 3.4 Summary

We have proposed a parallel multi-jagged coordinate partitioning algorithm, MJ, that differs from RCB in its ability to partition into multiple parts at once and its handling of data during partitioning. We have presented experiments on various datasets, and compared the performance of MJ against Zoltan's RCB. Our weak scaling experiments demonstrated that our MJ algorithm performs better than Zoltan's RCB, and scales up to 6144 processors. MJ scaled well up to 384 processors in most of our strong scaling experiments. We have also evaluated the quality of the RCB and MJ partitions in terms of communication.

One of the most significant differences between MJ and Zoltan's RCB is that RCB migrates the coordinates and weights of the objects after each bisection. MJ intelligently decides whether to migrate or not. Migration allows RCB (and MJ) to work with fewer processors at each recursion and also allows execution of subsequent

bisections in parallel. Even though the amount of data moved decreases with increasing levels of recursion for each bisection, all processors perform migration at each recursion level. The worst case occurs when the data is randomly distributed (with uniform distribution) to all processors. In such a case, half of the data will be moved at each recursion. In the ideal case, the initial data is already partitioned, and RCB will not migrate any data. The effect of migration is visible from our experiments. In the weak scaling tests, RCB's performance can be attributed to communication costs due to migration. In the strong scaling results for real datasets, the data was already well-localized in processors, and hence, provided a more ideal case for RCB.

After each partitioning, MJ checks the distribution of the data to processors and estimates the cost of global reduction operations if data were not to migrate, and decides whether to migrate or not. We have shown that MJ does reasonably well to identify the best course of action; however, the interplay of the data size and system size to decide when migration will be most beneficial needs further investigation.

# Chapter 4: Load-Balancing via Hypergraph Models

Finding a good partition of communicating tasks among the available processing units is crucial for obtaining short execution times, using less power, and utilizing the computation and communication resources better. To solve this problem, several graph and hypergraph models have been proposed [36, 37, 42, 84, 145]. These models transform the problem at hand to a balanced partitioning problem. The balance restriction on part weights in conventional partitioning corresponds to the load balance in a parallel environment, and the minimization objective for a given metric relates to the minimization of the communication between the processing units.

The most widely used communication metric is the *total communication volume*. Other communication metrics, such as the *total number of messages* [136], the *maximum volume of messages sent* and/or *received* by a processor [25, 136], or the maximum number of messages sent by a processor have also been shown to be important. The latency-based metrics, which model the communication by using the number of messages sent/received throughout the execution, become more and more important as the number of processors increases. Ideal partitions yield perfect computational

load balance and minimize the communication requirements by minimizing all the mentioned metrics.

Given an application, our main objective is to partition the tasks evenly among processing units and to minimize the communication overhead by minimizing several communication cost metrics. Previous studies addressing different communication cost metrics (such as [25, 136]) work in two phases where the phases are concerned with disjoint subsets of communication cost metrics such that in each phase, they minimize a different metric, sometimes at the expense of others. We present a novel approach to treat the minimization of multiple communication metrics as a multi-objective minimization in a single phase. In order to achieve that, the use of directed hypergraph models is proposed. The proposed approaches are materialized in UMPa (pronounced as "Oompa"), which is a multi-level partitioner employing a directed hypergraph model and novel $K$-way refinement heuristics. UMPa not only takes the total communication metric into account, but it also treats the maximum communication volume, and the total and maximum number of messages in a generalized framework. It aims to minimize the primary metric and obtains improvements in the secondary communication metrics. Compared to the state of the art partitioning tools PaToH [38], Mondriaan [142], and Zoltan [64] using the standard hypergraph model, which minimize the total communication volume, a large number of problem instances show that UMPa produces much better partitions in terms of several communication metrics with 128, 256, 512, and 1024 processing units.

The organization of the chapter is as follows. The previous work on minimizing multiple communication cost metrics is summarized in Section 4.1. Section 4.2 presents the directed hypergraph model, and explains how the communication metrics are encoded by those hypergraphs. In Section 4.3, the multi-level, multi-objective partitioning tool UMPa and give its implementation details is presented in Section 4.3.5. Section 4.4 presents the experimental results, and Section 4.5 concludes the chapter. Refer to Section 2.2.2 for the hyperhraph partitioning problem formulation and background information on the multilevel hypergraph partitioning.

## 4.1 Related work

Although minimizing the total communication volume TV is important, it is sometimes preferable to reduce other communication metrics [84]. The previous studies on minimizing multiple communication cost metrics are based on two-phase approaches. Generally, the first phase tries to obtain a proper partition of data for which the total communication volume is reduced. Starting from the partition of the first phase, the second phase tries to optimize another communication metric. Please note that, even though such two-phase approaches could allow the use of state-of-the-art techniques in one or both of the phases, since solutions are sought in one phase are oblivious the metric used in the other phase, they could stuck in some local optima that it cannot be improved in the other phase.

Bisseling and Meesen [25] discuss how to reduce the maximum send and receive volume per processor in the second phase, while keeping the total volume of communication intact. This is achieved by a greedy assignment algorithm that assigns a data source to a processor that needs it and that has the smallest send/receive volume under current assignments. Bisseling and Meesen also discuss a greedy improvement algorithm applied after the assignments are done.

Uçar and Aykanat [136] discuss how to reduce the total number of messages and achieve balance on the maximum volume of messages sent by a processor as a hypergraph partitioning problem in the second phase. The balance on the volume of messages sent by a processor is achieved only approximately (the proposed model does not encode the send volume of a processor exactly). The metric of the maximum number of messages sent by a processor is somehow incorporated into the second phase as well. Both these second phase alternatives, however, can increase the total volume of communication found in the first phase. The amount of increase is bounded by the number of cut nets found in the first phase.

Both of the mentioned studies [25, 136] consider applications where some input data is combined to yield the output data, as in the sparse matrix-vector multiply operation $\mathbf{y} \leftarrow \mathbf{Ax}$. In such settings, sometimes it is advisable to align the partition on the input and output vectors, e.g., in $\mathbf{y} \leftarrow \mathbf{Ax}$ the processor that holds an $\mathbf{x}_i$ holds the corresponding entry $\mathbf{y}_i$. This latter requirement is called *symmetric vector*

*partitioning* and cannot be satisfied easily with the methods proposed in these two studies.

Uçar and Aykanat [135] discuss how to extend their earlier approach to address the symmetric partitioning requirement as well for the computations of the form $\mathbf{y} \leftarrow \mathbf{Ax}$. Their approach can only handle the cases where one has the liberty to partition the vector entries independent of the matrix partition. In their work, Uçar and Aykanat show how this liberty arises when $\mathbf{A}$ is partitioned on the nonzero basis. Again, the second phase can increase the total volume of communication found in the first phase where a non-trivial upper bound on the amount of increase is known. In some other cases, for example, when the matrix is to be partitioned rowwise and the owner computes rule has to be respected, then the method is not applicable. The entry $\mathbf{y}_i$ should be computed at the processor holding the $i$th row of $\mathbf{A}$ (this, in turn, determines the partition on $\mathbf{x}$), unless of course one is ready to pay for another communication round.

The main contribution in this work is to address multiple communication cost metrics in a single-phase. Addressing all the metrics in a single-phase would allow trading off the cost associated with one metric in favor of that associated with another one. The standard hypergraph model cannot see the communication metrics that are defined on a per-processor basis, therefore balance on communication loads of the processors cannot be formulated naturally. Furthermore, since all the state-of-the-art

partitioners use iterative-improvement-based heuristics for the refinement, a single-phase approach increases search space by avoiding to get stuck in a local optima for a single metric. In order to overcome these obstacles, we propose to use directed hyperedges and minimize a prioritized set of metrics all together. We associate each input and output data and computational tasks with a vertex as is done in the standard hypergraph models [137]. We then encode the dependencies among the data and computational tasks with directed hyperedges so that a unique source or a destination is defined for each hyperedge. This way, we are able to accurately model the total as well as per-processor communication cost metrics and reduce all metrics together. Furthermore, this allows the optimization of the communication cost metrics both in unsymmetric and symmetric input and output data partitions constraints, by incorporating the vertex amalgamation technique discussed in [137]. Directions on hyperedges necessitate revisiting some parts of the standard multi-level partitioning heuristics. We do so and realize our communication cost minimizing methods in a multi-objective, multi-level hypergraph partitioner called UMPa. As balancing the per-processor communication cost metrics requires a global view of the partition, we design a direct $K$-way refinement heuristic. One more positive side effect of using the proposed directed model, as we will demonstrate in the experimental evaluation section, is that one could further reduce a primary metric when additional secondary and tertiary metrics (related to the communication) are given to UMPa.

## 4.2 The directed hypergraph model

Let $\mathcal{A} = (\mathcal{T}, \mathcal{D})$ be an application where $\mathcal{T}$ is the set of tasks to be executed and $\mathcal{D}$ be the set of data elements processed during the application. The tasks may have different execution times; for a task $t \in \mathcal{T}$, we use $exec(t)$ to denote its execution time. The data elements may have different sizes; for a data element $d \in \mathcal{D}$, we use $size(d)$ to denote its size. Data elements can be input and output elements, hence, respectively, they may not have any producer or consumer task in the application, or they can be produced by some tasks and consumed/used by other tasks. Graphs, and its variants, such as the standard and bipartite graphs [84], directed acyclic graphs [99], and hypergraphs [42] have been used to model many such applications with different dependency constraints.

Here, we are interested in a set of applications that can be executed in parallel similar to the bulk synchronous parallel (BSP) model of execution [21, 140]. In other words, the set of tasks will be executed concurrently on a set of processors, consume and produce data elements, and then they will exchange data among each other. This process is usually repeated, and the same computation and communication patterns are realized in multiple iterations. To ensure that the data dependencies are met, one can either have an explicit barrier synchronization between each iteration, as in the original BSP model, or a more complex scheduling can be used in each processor that would delay the execution of the tasks for which input data elements have not been received yet. Such an execution model fits well to many scientific computations [21],

iterative solvers, and also to generalization of other execution models, such as parallel reductions [70], MapReduce [56], and its iterative [67] and pipelined [51] variants.

We assume the owner-computes rule: each task $t \in \mathcal{T}$ is executed by the processing unit to which $t$ is assigned. Consider an iterative solver for linear system of equations (such as the conjugate gradients) that repeatedly performs $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ and applies linear vector operations on $\mathbf{x}$ and $\mathbf{y}$. One way to parallelize such a solver is to use rowwise matrix partitioning and assign sets of matrix rows and corresponding $\mathbf{y}$ and $\mathbf{x}$ elements to the processing units [21, 37]. In such an assignment, the atomic task $t_i$ is defined as the computation of the inner product of the $i$-th row with the $\mathbf{x}$ vector, i.e., $\mathbf{y}_i \leftarrow \mathbf{A}_{i*}\mathbf{x}$, where data element $\mathbf{y}_i$ is produced by task $t_i$.

We assume that during the application execution, the producers of the data elements send data to their consumers. If they exist, the consumers of a data element-$\mathcal{D}$ are all in $\mathcal{T}$. In some applications, there is a direct one-to-one mapping of the data elements and tasks. That is, task $t_i$ produces data element $d_i$. However, in general, there can be data in $\mathcal{D}$ which are given as input and may not be produced by a task.

### 4.2.1 Modeling the application

We propose modeling the application with a *directed hypergraph* [73]. Given an application $\mathcal{A}$, we construct the directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ as follows. For each task $t_i \in \mathcal{T}$, we have a corresponding vertex $v_i \in \mathcal{V}$. For each data element $d_j \in \mathcal{D}$,

we have a corresponding vertex $v_j \in \mathcal{V}$ and a corresponding net $n_j \in \mathcal{N}$, where

$$\mathsf{w}[v] = \begin{cases} exec(t) & \text{if } v \text{ corresponds to a task } t, \\ 0 & \text{if } v \text{ corresponds to a data item.} \end{cases}$$

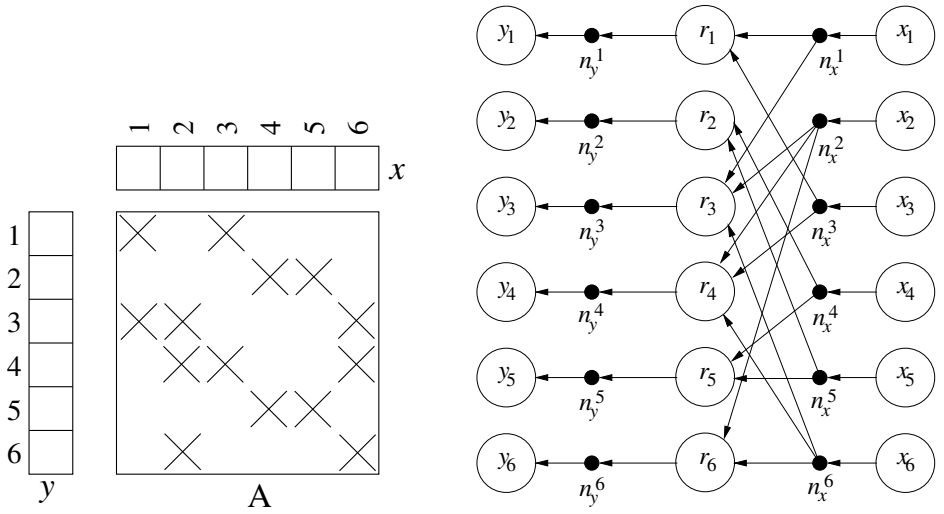
and $\mathsf{c}[n_i] = size(d_i)$. Since we are interested in balancing the computational load of the processors, the vertices corresponding to data items have zero weight. If one also wants to balance the storage, which may be necessary for memory-restricted processing units, a multiconstraint hypergraph partitioning scheme [40] can be used with extra positive weights, representing size of data elements, on these vertices and zero weights on the vertices corresponding to tasks.

The pins of a net $n_j$, $\mathsf{pins}[n_j]$, includes a producer (also called *source*), which will be denoted as $\mathsf{src}[n_j]$, and consumers of the corresponding data item $d_j$. In this directed hypergraph model, the communication represented by a net $n$ is flowing from its producer vertex to its consumer vertices $\mathsf{pins}[n] \setminus \{\mathsf{src}[n]\}$.

Figure 4.1a shows the sparse matrix and vectors of the SpMxV operation $\mathbf{y} \leftarrow \mathbf{Ax}$ in a sample of the iterative solver application mentioned above. Figure 4.1b shows the associated directed hypergraph model in the case where a rowwise partitioning is required. In the model, there are three types of vertices ($x_i$, $y_i$, and $r_i$ for $i = 1, \ldots, 6$), each modeling an $\mathbf{x}$- or a $\mathbf{y}$-vector entry, or a row of $\mathbf{A}$. The vertices are shown as labeled white circles. The nets are shown as filled, small circles. The directions of the pins are set from $x$-vertices to the corresponding nets, from those nets to the row vertices. The $y$-vertices are the destinations of the corresponding nets whose sources are the corresponding row vertices.

In the directed hypergraph model of Figure 4.1b, there is a single sender and multiple receivers for each net. This model fits well with many scientific applications where there is a single owner of the data, who is also responsible distributing the updated value to processors that needs it. There are applications in which there are multiple senders that send, usually, their partial updates to the owner of the data. Some other applications use both of these models iteratively, such as gathering of partial updates followed by distribution of the updated values (e.g. [123]). A directed hypergraph can model such applications. However, the current implementation of UMPa supports only the discussed applications with a single sender. We note that in SpMxV when the partitioning is columnwise or on nonzero basis (that is a two dimensional partitioning on the matrix), some nets of the directed hypergraph will have multiple senders, in which case UMPa cannot be used in its current form.

In our directed hypergraph model, a part $p$ corresponds to a processing unit $p$, and the balance restriction of the partitioning problem on the part weights necessitates a balanced distribution of the computational load among the processing units. In addition, the total communication volume corresponds to the connectivity-1 metric in (2.2.3).




(a) Sample matrix-vector multiplication

(b) Directed hypergraph model

Figure 4.1: Sample sparse matrix-vector multiplication and the corresponding directed hypergraph model.

### 4.2.2 Communication cost metrics using direction information

Given a $K$-way partition, let $\mathsf{SV}[p]$ and $\mathsf{RV}[p]$ be the data volume the processing unit $p$ sends and receives, respectively. That is

$$\mathsf{SV}[p] = \sum_{\mathsf{part}[\mathsf{src}[n]]=p} \mathsf{c}[n](\lambda_n - 1) \ , \tag{4.2.1}$$

$$\mathsf{RV}[p] = \sum_{\substack{\mathsf{part}[\mathsf{src}[n]]\neq p \\ p\in\mathsf{prts}[n]}} \mathsf{c}[n] \ . \tag{4.2.2}$$

Hence, $\mathsf{TV}$ equals to $\sum_{i=1}^{K} \mathsf{SV}[p_i] = \sum_{i=1}^{K} \mathsf{RV}[p_i]$. Let $\mathsf{SRV}[p] = \mathsf{SV}[p] + \mathsf{RV}[p]$ be the total volume of data sent/received by the processor $p$. The maximum data volume sent $\mathsf{MSV}$ and sent/received $\mathsf{MSRV}$ by a single processor are defined as

$$\mathsf{MSV} = \max_i \{\mathsf{SV}[p_i]\} \ , \tag{4.2.3}$$

$$\mathsf{MSRV} = \max_i \{\mathsf{SRV}[p_i]\} \ . \tag{4.2.4}$$

Let $\mathsf{SM}[p_i]$ be the number of messages the processing unit $p_i$ sends, that is

$$\mathsf{SM}[p_i] = |\{p_j : \exists n \ s.t. \ \mathsf{part}[\mathsf{src}[n]] = p_i \text{ and } p_j \in \mathsf{prts}[n] \setminus \{p_i\}\}| \ . \tag{4.2.5}$$

The total number of messages $\mathsf{TM}$, and the maximum number of messages sent by a single processor $\mathsf{MSM}$ are defined as

$$\mathsf{TM} = \sum_i \mathsf{SM}[p_i] \ , \tag{4.2.6}$$

$$\mathsf{MSM} = \max_i \{\mathsf{SM}[p_i]\} \ . \tag{4.2.7}$$

Consider again the sparse matrix-vector multiplication of Figure 4.1 in the context of an iterative solver where the vectors $\mathbf{x}$ and $\mathbf{y}$ undergo linear operations (such as $\mathbf{x}_i \leftarrow \mathbf{x}_i + \beta \mathbf{y}_i$ for a scalar $\beta$ to form the $\mathbf{x}$ of the next iteration). In this case, the hypergraph can be simplified by using a set of modifications which are useful to avoid some extra communications. First, the data items $\mathbf{x}_i$ and $\mathbf{y}_i$ should be in the same part upon partitioning—otherwise an extra data transfer is required during the vector operations. Second, $\mathbf{y}_i$ is produced by $r_i$; unless they are in the same part, an extra communication is required. It is therefore advisable to combine the vertices $x_i$, $y_i$, and

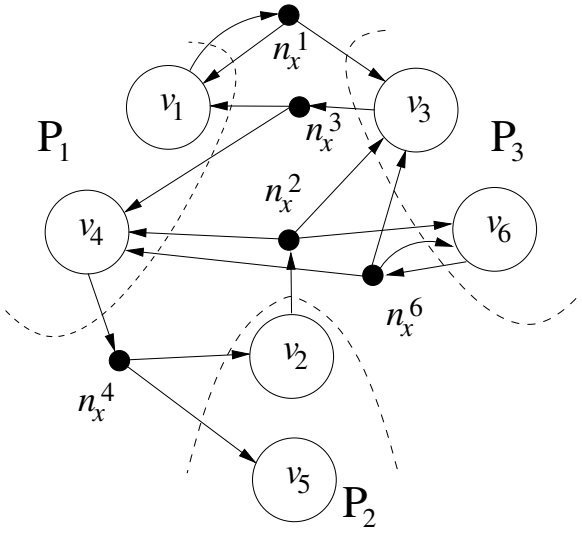



Figure 4.2: A 3-way partition of the simplified directed hypergraph. A vertex $v_i$ now represents $x_i$, $y_i$ and row $r_i$. Among the nets, the cut ones are shown.

$r_i$ for all $i$ (see the vertex amalgamation operation introduced in [136]; see also [137]). This simplified model corresponds to the column-net hypergraph model [37]. Consider the 3-way partition of the resulting hypergraph depicted in Figure 4.2 (internal nets are not shown for clarity). Under this partition, the processor $p_1$, corresponding to the part $P_1$, holds vector entries $x_1, x_4$, rows $r_1, r_4$ and is responsible for computing $y_1$ and $y_4$ at the end of multiplication. Similar data and computation assignments apply to the processors $p_2$ and $p_3$ corresponding to the parts $P_2$ and $P_3$. As seen with the directions on the pins of the cut nets, processor $p_1$ sends $x_1$ to $p_3$ and $x_4$ to $p_2$; processor $p_2$ sends $x_2$ to $p_1$ and $p_3$; and $p_3$ sends $x_3$ and $x_6$ to $p_1$. Combining these,

we see that $\mathsf{SV}[p_i] = 2$, for all $i$; $\mathsf{RV}[p_1] = 3$, $\mathsf{RV}[p_2] = 1$, and $\mathsf{RV}[p_3] = 2$. We further note that $\mathsf{SM}[p_1] = 2$, $\mathsf{SM}[p_2] = \mathsf{SM}[p_3] = 1$.

The directions on the hyperedges help to quantify the sends and receives of each processor. Without directions, one would not know, for example, if the information flow on $n_x^3$ should have been as described. If there were no vertex amalgamation, it would be possible to compute these without directions, but with a little bookkeeping and additional computation. However, vertex amalgamation type operations (e.g., the coarsening phase in a multi-level partitioner) always take place in the state-of-the-art partitioners. In this case, quantifying the sends and receives of each processor would require too much bookkeeping (the contents of a composite vertex and connections of the composing vertices to the nets) and too much computation to be useful in the multi-level partitioning framework. The directions in the nets avoid this difficulty, but necessitate the development of suitable partitioning tools.

## 4.3 UMPa: A multi-objective partitioning tool for communication minimization

The proposed partitioner, UMPa, aims to optimize a given volume- or latency-based primary metric and tries to reduce a set of secondary communication metrics. Although, the recursive bisection approach can work well for the total communication volume metric ($\mathsf{TV}$), it is not suitable for the maximum volume ($\mathsf{MSV}$), the total number of messages ($\mathsf{TM}$), and the maximum number of messages sent by a processor

(MSM). Since we aim to handle multiple communication metrics all together, UMPa follows the direct $K$-way partitioning approach.

### 4.3.1 Multi-level coarsening phase

In this phase, the original hypergraph is gradually coarsened in multiple levels by clustering subsets of vertices at each level. There are two types of clustering algorithms: matching-based and agglomerative. The matching-based algorithms put at most two similar vertices in a cluster, whereas the agglomerative ones allow any number of similar vertices. There are various similarity metrics—see for example [6, 38, 64]. All these metrics are defined only on two adjacent vertices (one of them can be a vertex cluster). Two vertices are adjacent if they share a net and they can be in the same cluster if they are adjacent.

We use an agglomerative algorithm and the absorption clustering metric using pins [6, 38]. For this metric, the similarity between two adjacent vertices $u$ and $v$ is

$$\sum_{n \in \mathsf{nets}[u] \cap \mathsf{nets}[v]} \frac{\mathsf{c}[n]}{|\mathsf{pins}[n]| - 1}$$

This is also the default metric in PaToH [38]. In each level $\ell$, we start with a finer hypergraph $\mathcal{H}^\ell$ and obtain a coarser one $\mathcal{H}^{\ell+1}$. If $\mathcal{V}_C \subset \mathcal{V}^\ell$ is a subset of vertices deemed to be clustered, we create the cluster vertex $u \in \mathcal{V}^{\ell+1}$ where $\mathsf{nets}[u] = \cup_{v \in \mathcal{V}_C} \mathsf{nets}[v]$. We also update the pin sets of the nets in $\mathsf{nets}[u]$ accordingly.

Since we need the direction, we always store the source vertex of a net $n \in \mathcal{N}$ as the first pin in $\mathsf{pins}[n]$. To maintain this information, when a cluster vertex $u$ is

formed in the coarsening phase, we put $u$ to the head of $\mathsf{pins}[n]$ for each net $n$ whose source vertex is in the cluster. We also discard the nets that become singleton at each step of the coarsening phase; as well as the initial singleton nets, since they do not cause any communication.

### 4.3.2 Initial partitioning phase

To obtain an initial partition for the coarsest hypergraph, we use PaToH [38], which is proved to produce high quality partitions with respect to the total communication volume [37]. We execute PaToH five times and get the best partition according to the given primary metric. We chose to use PaToH for three reasons. First, since we always take TV into account either as a primary or a secondary metric, it is better to start with an initial partition having a good total communication volume. Second, since TV is the sum of the send volumes of all parts, minimizing it should also be good for both MSV and MSRV and even for the latency-based metrics. We verified the second reason, although it sounds intuitive, in preliminary experiments [45]. The third reason is more esoteric. The coarsest hypergraph which has small net sizes and high vertex degrees lends itself gracefully to the recursive bisection and FM-based improvement heuristics (see also elsewhere [11]).

### 4.3.3 Uncoarsening phase and one-phase $K$-way refinement

The uncoarsening phase is realized in multiple levels corresponding to the coarsening levels where at the $\ell$th level, we project the partition $\Pi^{\ell+1}$ obtained for $\mathcal{H}^{\ell+1}$

to $\mathcal{H}^\ell$. Then, we refine it by using a novel $K$-way refinement heuristic which takes the primary and the secondary metrics into account. The proposed heuristic runs in multiple passes where in a pass it visits each boundary vertex $u$ and either leaves it in $\mathsf{part}[u]$, or moves it to another part according to some move selection policy.

UMPa provides refinement methods for four primary metrics: the total communication volume $\mathsf{TV}$; the maximum send volume $\mathsf{MSV}$; the total number of messages $\mathsf{TM}$; and the maximum number of messages a processing unit sends $\mathsf{MSM}$. We use the notation $\text{UMPa}_X$ to denote the partitioner with the primary objective function $X$ during the refinement. These methods take secondary and sometimes tertiary metrics into accounts. As an aid to the refinement process, the part weights can be used for tie-breaking purposes while selecting the best move.

Since each metric is different, the implementation details of the refinement heuristics are also different. However, their main logic is the same. The heuristics perform a number of passes on the boundary vertices. To be precise, UMPa uses at most $2^\ell$ passes for $\mathcal{H}^\ell$, the hypergraph at the $\ell$th coarsening level. We observed that most of the improvement on the metrics are coming from the refinement on the coarser hypergraphs. Furthermore, since these hypergraphs are smaller, the passes on them take much less time. That is, the impact of these passes are high and their overhead is low. Hence, we decided to perform more passes on the coarser hypergraphs. UMPa also stops the passes when the improvement on the primary metric is not significant during a pass.

**Algorithm 5:** A generic pass for $K$-way refinement

```
   Data: H = (V, N), (M1, M2, M3): the metrics
   for each u ∈ boundary do
       p_best ← part [u];
1      L_best ← (gain_M1, gain_M2, gain_M3) ← LEAVEVERTEX(H, u);
2      for each part p ≠ part[u] do
           if p has enough space for vertex u then
3              L = (loss_M1, loss_M2, loss_M3) ← PUTVERTEX (H, u, p);
4              (p_best, L_best) ← SELECTMOVE(p_best, L_best, p, L, W);
       if p_best ≠ part[u] then
5          move u to p_best and update the data structures accordingly;
```

The high-level structure of a pass is given in Algorithm 5. In a pass, the heuristic visits the boundary vertices in a random order and for each visited vertex $u$ and for all $p \neq \mathsf{part}[u]$, it computes how the communication metrics in $(M_1, M_2, M_3)$ are affected when $u$ is moved from $\mathsf{part}[u]$ to $p$. This computation is realized in two steps. First, $u$ is removed from $\mathsf{part}[u]$ and the *leave gains* for the communication metrics are computed with LEAVEVERTEX (line 1). Second, $u$ is tentatively put into a candidate part $p$ and the *arrival losses* are calculated with PUTVERTEX (line 3). We first set the processing unit for the best move as $p_{best} = \mathsf{part}[u]$. Since the leave gains and arrival losses are equal while removing a vertex from its part and putting it back (the total gain is zero), initially, the best arrival loss triplet $\mathcal{L}_{best}$ is set to $(gain_{M_1}, gain_{M_2}, gain_{M_3})$. Then, for each possible target processing unit $p$, the arrival losses on the metrics are computed. With the SELECTMOVE heuristic, which is given in Algorithm 6, these losses are compared with $\mathcal{L}_{best}$ to select the best move (line 4). After trying all target

processing units and computing the arrival losses of the corresponding moves, $u$ is moved to $p_{best}$. If $p_{best} \neq \mathsf{part}[u]$ the data structures, such as $\mathsf{boundary}, \mathsf{part}, \lambda$, and $\Lambda$, used throughout the partitioning process are updated accordingly.

**Algorithm 6:** The generic SELECTMOVE operation

**Data**: $p, \mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3), p', \mathcal{L}' = (\mathcal{L}'_1, \mathcal{L}'_2, \mathcal{L}'_3), W$
**if** $\mathcal{L}_1 < \mathcal{L}'_1$ **then**
  **return** $(p, \mathcal{L})$ ; ▷Primary metric
**else if** $\mathcal{L}_1 = \mathcal{L}'_1$ **and** $\mathcal{L}_2 < \mathcal{L}'_2$ **then**
  **return** $(p, \mathcal{L})$ ; ▷Secondary metric
**else if** $\mathcal{L}_1 = \mathcal{L}'_1$, $\mathcal{L}_2 = \mathcal{L}'_2$ **and** $\mathcal{L}_3 < \mathcal{L}'_3$ **then**
  **return** $(p, \mathcal{L})$ ; ▷Tertiary metric
**else if** $\mathcal{L}_1 = \mathcal{L}'_1$, $\mathcal{L}_2 = \mathcal{L}'_2$, $\mathcal{L}_3 = \mathcal{L}'_3$ **and** $W_p < W_{p'}$ **then**
  **return** $(p, \mathcal{L})$ ; ▷Tie-breaking
**return** $(p', \mathcal{L}')$

The SELECTMOVE heuristic given in Algorithm 6 compares two vectors $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, W_p)$ and $(\mathcal{L}'_1, \mathcal{L}'_2, \mathcal{L}'_3, W_{p'})$. Starting from the first ones, it compares the vector entries one by one. When a smaller entry is found, the corresponding processing unit is returned as the better candidate.

### 4.3.4 Multi-objective one-phase $K$-way Refinement

Although the structure of a pass and the execution logic is the same, the computation of the leave gains and arrival losses differ with respect to the communication metrics used. Here we describe the implementation of the functions LEAVEVERTEX and PUTVERTEX for UMPa$_{\mathsf{TV}}$, UMPa$_{\mathsf{MSV}}$ with secondary and tertiary metrics MSRV

and TV, and UMPa$_{\mathsf{TM}}$ with a secondary metric TV. We also implemented UMPa$_{\mathsf{MSM}}$ and used it in the experiments. But we do not give the details of its leave gain and arrival loss computations since the structure of these algorithms is similar to the ones described below.

#### 4.3.4.1 Total communication volume

In the directed hypergraph model, the total volume of the communication sent/received throughout the execution corresponds to the cutsize definition (2.2.3) as in the standard hypergraph model. In other words, the TV metric is global and hence, the sense of direction does not have any effect in its computation. Here we describe UMPa$_{\mathsf{TV}}$ which takes a single communication metric TV into account, i.e., it is a standard hypergraph partitioner with the traditional objective function (2.2.3).

**Algorithm 7:** LEAVEVERTEXTV

```
   Data: H = (V, N), u
   gain_TV ← 0;
   P_u ← ∅;
   for each n ∈ nets[u] do
       if Λ(n, part[u]) = 1 then
1          gain_TV ← gain_TV + c[n];
2      P_u ← P_u ∪ prts[n];
   return(gain_TV, P_u);
```

As described above, the refinement heuristic performs passes on the boundary vertices. Let $u$ be the boundary vertex visited during a pass. The LEAVEVERTEXTV method given in Algorithm 7 computes the gain on TV when $u$ leaves part$[u]$. After the move, there will be a gain if and only if the target processor $p$ contains at least one vertex which shares a net with $u$. Let $\mathcal{P}_u$ be the set of such parts. To be more

efficient, this set can be computed beforehand and the for loop in Algorithm 5 (line 2) can be iterated over $p \in \mathcal{P}_u$ instead of $p \in \{1, \ldots, K\} \setminus \mathsf{part}[u]$. For this reason, leaveVertexTV also builds this set (line 2) while computing $gain_{\mathsf{TV}}$.

**Algorithm 8:** putVertexTV

```
Data: H = (V, N), u, p (the candidate part)
loss_TV ← 0;
for each n ∈ nets[u] do
    if Λ(n, p) = 0 then
1       loss_TV ← loss_TV + c[n];
return loss_TV;
```

After $u$ leaves $\mathsf{part}[u]$ and the leave gain is computed, the refinement heuristic computes the arrival losses for all target parts $\mathcal{P}_u \setminus \mathsf{part}[u]$. Algorithm 8 gives the description of the putVertexTV function. Given $u$ and a part $p$, the function computes the increase on TV after putting $u$ to $p$. For each net $n \in \mathsf{nets}[u]$, TV increases by $\mathsf{c}[n]$ if $u$ will be the only pin of $n$ in $p$ (line 1).

#### 4.3.4.2 Total number of messages

Although the directionality is not important for the total volume, the flow of communication is crucial while minimizing other metrics. Hence for each net $n$ affected by a move, we need to use the source information $\mathsf{src}[n]$. Here we describe the leave gain and arrival loss computations of UMPa$_{\mathsf{TM}}$ which minimizes the total number of

messages throughout the execution by also taking the total volume of communication, TV, into account as the secondary metric.

**Algorithm 9:** LEAVEVERTEXTM

```
   Data: H = (V, N), u
   gain_TV ← 0, gain_TM ← 0;
   senders ← ∅, receivers ← ∅;
   p_u ← part[u];
   for each n ∈ nets[u] do
1      if Λ(n, p_u) = 1 then
           gain_TV ← gain_TV + c[n];
2      if src[n] = u then
           if Λ(n, p_u) > 1 then
               receivers ← receivers ∪ {p_u};
3          for each p ∈ prts[n] \{p_u} do
               receivers ← receivers ∪ {p};
               netComm(p_u, p) ← netComm(p_u, p) − 1;
               if netComm(p_u, p) = 0 then
                   gain_TM ← gain_TM + 1;
4      else
           p_sender ← part[src[n]];
           senders ← senders ∪ {p_sender};
           if Λ(n, p_u) = 1 then
               netComm(p_sender, p_u) ← netComm(p_sender, p_u) − 1;
               if netComm(p_sender, p_u) = 0 then
                   gain_TM ← gain_TM + 1;
   return (gain_TM, gain_TV, senders, receivers);
```

Let $netComm(p, p')$ be the number of nets $n$ such that $p, p' \in \mathsf{prts}[n]$ and $\mathsf{src}[n] = p$, i.e., the number of data to be sent from the processing unit $p$ to the processing unit $p'$. The refinement heuristic performs passes on the boundary vertices. Let $u$

be the boundary vertex visited during a pass. When $u$ leaves $\mathsf{part}[u]$, the gains on TM and TV are computed by LEAVEVERTEXTM given in Algorithm 9. The gain on TV (line 1) is computed as described in the previous subsection. For the gain on TM, there are two cases:

1. $u$ is the source pin of a net $n$ (line 2): For each part $p \in \mathsf{prts}[n]$, there is a gain if $netComm(\mathsf{part}[u], p) = 0$ after removing $u$ from $\mathsf{part}[u]$. Note that a vertex can be a source for multiple nets.

2. $u$ is a non-source pin of a net $n$ (line 4): Let $p_{sender}$ be the part having the source of $u$. There is a gain if $netComm(p_{sender}, \mathsf{part}[u]) = 0$ after removing $u$ from $\mathsf{part}[u]$.

In addition to gain computations, the set of the parts that receive data from $u$ (i.e., $u$ is the source of a net connected to the receiver part), $receivers$, and sends data to $u$ (i.e., $u$ is a non-source pin of a net whose source is in the sender part), $senders$, are obtained. These sets are used to efficiently compute the arrival losses for each target part candidate.

After the leave gains are computed with LEAVEVERTEXTM, the heuristic tentatively moves the boundary vertex to the candidate parts and computes the losses on the communication metrics with PUTVERTEXTM given in Algorithm 10. Let $p$ be the target part. The function first computes the arrival loss on TV as described in the previous subsection. Then for all $p_{receiver} \in receivers$, it checks if a new message from $p$ to $p_{receiver}$ is necessary. This implies a loss on TM. It repeats a similar process

**Algorithm 10:** PUTVERTEXTM

```
Data: H = (V, N), u, p (the candidate part), senders, receivers
loss_TM ← 0, loss_TV ← 0;
for each n ∈ nets[u] do
    if Λ(n, p) = 0 then
        loss_TV ← loss_TV + c[n];
1 for each p_receiver ∈ receivers do
    if netComm(p, p_receiver) = 0 then
        loss_TM ← loss_TM + 1;
2 for each p_sender ∈ senders do
    if netComm(p_sender, p) = 0 then
        loss_TM ← loss_TM + 1;
return (loss_TM, loss_TV);
```

for all $p_{sender} \in senders$ and check if a message from $p_{sender}$ to $p$, which was not necessary before the move, is now necessary.

#### 4.3.4.3 Maximum communication volume

Here we describe UMPa$_{\mathsf{MSV}}$ which minimizes MSV, the maximum volume sent by a processing unit, and also takes MSRV (the maximum send and receive volume), and TV into account as the secondary and tertiary metrics, respectively. This metric configuration is useful when one wants a partition which does not yield a bottleneck processor and does not use the network extensively.

The generic structure of the refinement heuristic given in Algorithm 5 also applies to the local communication metrics such as MSV and MSM. But for an efficient implementation, we slightly alter this structure for UMPa$_{\mathsf{MSV}}$. When a vertex is taken

from its part and put to another one during the refinement, instead of computing the exact gains/losses on the metrics, we compute a set of intermediate values which correspond to the changes in the send/receive volumes of a processing unit $p$, i.e., $\mathsf{SV}[p]$ and $\mathsf{RV}[p]$. After each move, these intermediate values are used to compute the exact changes on the metrics, and the best move is selected in a similar fashion.

**Algorithm 11:** LEAVEVERTEXMSV

```
   Data: H = (V, N), u
   toOldPart ← 0;
   gain_RV ← 0;
   for each processing unit p do
       gain_SV[p] ← 0;
   p_u ← part[u];
   for each n ∈ nets[u] do
       p_sender ← part[src[n]];
       if src[n] = u then
           gain_SV[p_u] ← gain_SV[p_u] + c[n](λ_n − 1);
1          if Λ(n, p_u) > 1 then
               gain_RV ← gain_RV − c[n];
               toOldPart ← toOldPart + c[n]
       else
           if Λ(n, p_u) = 1 then
2              gain_SV[p_sender] ← gain_SV[p_sender] + c[n];
3              gain_RV ← gain_RV + c[n];
   return (gain_SV, gain_RV, toOldPart);
```

Algorithm 11 describes the LEAVEVERTEXMSV function. To store the changes on the send volumes of the processing units after the leave operation, the function uses an array $gain_{\mathsf{SV}}[\cdot]$ of size $K$. It also stores the change on $\mathsf{RV}[\mathsf{part}[u]]$, i.e., the receive

volume of $\mathsf{part}[u]$, in a variable $gain_{RV}$. If $u$ is the source for a net $n$, removing $u$ yields $\mathsf{c}[n](\lambda_n - 1)$ gain on the send volume of $\mathsf{part}[u]$. But after the move, there will also be an increase on $\mathsf{RV}[\mathsf{part}[u]]$ if there are more pins of $n$ in $\mathsf{part}[u]$ (line 1). The potential increase due to the nets having $u$ as the source does not depend on the target part. It is stored in a variable *toOldPart* which later will be used by PUTVERTEXMSV function to compute the overall change on the send volume of each target part. When $u$ leaves $\mathsf{part}[u]$, $\mathsf{RV}[\mathsf{part}[u]]$ decreases by $\mathsf{c}[n]$ if $u$ is not the source of $n$ and if it is the only pin of $n$ in $\mathsf{part}[u]$ (line 2). The same decrease must also be observed on $\mathsf{SV}[p_{sender}]$, the send volume of the part containing $u$'s source pin (line 3).

If $u$ is the source pin of $n$, as described above, we take the gain on $\mathsf{SV}[\mathsf{part}[u]]$ into account. However, we do not do the same for the corresponding gains on the RV values of other parts in $\mathsf{prts}[n]$. That is, we do not compute all the reductions on RV, and hence, if we compute the metrics here the MSRV value will not be exact. Although exact RV values and hence the exact gains on the metrics can be computed, we chose to follow this approach to be more efficient. Because, after putting $u$ to the target part $p$, these reductions on the receive volumes of the processing units in $\mathsf{prts}[n] \setminus \{\mathsf{part}[u], p\}$ will not be effective anymore since these processing units will continue to receive $n$'s data from $p$. Hence, if one computes the exact RV values here most of the computation will be redundant.

The details of the PUTVERTEXMSV are given in Algorithm 12. Similar to LEAVEVERTEXMSV, the algorithm computes the increase on the send volumes of

all parts and the increase on the receive volume of the target part $p$. After the move, the target processing unit will send the data which was sent by $\mathsf{part}[u]$ before. Hence, the value $loss_{\mathsf{SV}}[p]$ must cover $gain_{\mathsf{SV}}[p_u]$. It also needs to contain the amount of the communication from $p$ to $\mathsf{part}[u]$ due to the nets having $u$ as the source and remain connected to $\mathsf{part}[u]$, i.e., $toOldPart$ (line 1). The rest of the loss updates are similar to the ones in the previous algorithms.

**Algorithm 12:** PUTVERTEXMSV

```
   Data: H = (V, N), u, p (the candidate part), gain_SV, toOldPart
   loss_RV ← 0;
   for each processing unit q do
       loss_SV[q] ← 0;
   p_u ← part[u];
1  loss_SV[p] ← gain_SV[p_u] + toOldPart;
   for each n ∈ nets[u] do
       p_sender ← part[src[n]];
       if src[n] = u then
           if Λ(n, p) > 0 then
2              loss_SV[p] ← loss_SV[p] − c[n];
               loss_RV ← loss_RV − c[n]
       else
           if Λ(n, p) = 0 then
               loss_SV[p_sender] ← loss_SV[p_sender] + c[n];
               loss_RV ← loss_RV + c[n];
   return (loss_SV, loss_RV);
```

Algorithm 13 shows a pass of the proposed refinement heuristic of $\text{UMPa}_{\mathsf{MSV}}$. Firstly, $u$ is removed from $\mathsf{part}[u]$, and the intermediate values, partial gains on the

send/receive volumes, are computed by LEAVEVERTEXMSV. Then, $u$ is tentatively put to each target candidate $p$, and the intermediate values, arrival losses on the send/receive volumes, are obtained by PUTVERTEXMSV. As described above, although these intermediate values are inexact, their sum gives the exact change on the send/receive volumes for each processor. After computing these changes and the send/receive volumes after the move, the metrics are computed and the best move is selected in a similar fashion. That is the move with the smallest value on the primary metric is preferred. If these values are equal for two target processors, then the secondary metric is considered. In case of equality for the secondary metric values, the tertiary metric is taken into account. As we will show in the experimental results

section, this move selection policy and tie-breaking scheme have positive impact on all the communication metrics.

**Algorithm 13:** A pass for $K$-way maximum send volume refinement

```
Data: H = (V, N)
for each u ∈ boundary do
    (gain_SV, gain_RV, toOldPart) ← LEAVEVERTEXMSV(H = (V, N), u);
    M_best ← (MSV, MSRV, TV);
    p_best ← part[u];
    for each part p ≠ part[u] do
        if p has enough space for vertex u then
            (loss_SV, loss_RV) ← PUTVERTEXMSV(u, p, toOldPart);
1           for each part q do
                Δ_S ← loss_SV[q] − gain_SV[q];
                SV'[q] ← SV[q] + Δ_S;
                SRV'[q] ← SV[q] + Δ_S + RV[q];
            SRV'[part[u]] ← SRV'[part[u]] − gain_RV;
            SRV'[p] ← SRV'[p] + loss_RV;
            M_p ← (max(SV'), max(SRV'), TV + loss_RV − gain_RV);
            (p_best, M_best) ← SELECTMOVEMSV(p_best, M_best, p, M, W);
    if p_best ≠ part[u] then
        move u to p_best and update the data structures accordingly;
```

### 4.3.5 Implementation Details

During the gain computations, the heuristic uses the connectivity information between the nets and parts stored in data structures $\lambda$, $\Lambda$, and $netComm$. These structures are constructed after the initial partitioning phase and maintained by the uncoarsening phase. When a vertex $u$ is moved, we revisit the nets of $u$ and update the data structures accordingly. When new boundary vertices are detected, they are inserted to the array boundary and visited in the same pass. We restrict the number

of moves for a vertex $u$ during a pass to 4, in order to limit the execution time. When this number is reached the vertex is locked and removed from the boundary.

Let $\rho = \sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|$ be the number of pins in a hypergraph. For UMPa$_{\mathsf{TV}}$, considering the restriction on the number of moves per vertex, the time complexity of the leave gain and arrival loss computations is $\mathcal{O}(\rho)$ and $\mathcal{O}(\rho K)$, respectively. Hence, the overall complexity of a refinement pass is $\mathcal{O}(\rho K)$. The factor $K$ comes from the number of candidate target parts for each move. For UMPa$_{\mathsf{TV}}$, we only visit the boundary vertices and try the parts of the visited vertex's nets as target candidates. In practice, these numbers are much smaller than $|\mathcal{V}|$ and $K$ as a side-effect of the minimization of the objective functions.

For UMPa$_{\mathsf{TM}}$, UMPa$_{\mathsf{MSV}}$, and UMPa$_{\mathsf{MSM}}$, the overall time complexity of a refinement pass is $\mathcal{O}(\rho K + |\mathcal{V}|K^2)$ since all these variants need an additional for loop which iterates $\mathcal{O}(K)$ times (lines 1 and 2 in Algorithm 10 and line 1 in Algorithm 13). However, since only the boundary vertices and parts of the nets are visited, the factors are much smaller in practice, and the worst-case analysis is very loose.

To store the numbers of pins per part for each net, $\Lambda$, we use a 2-dimensional array. Hence, the space complexity is $\mathcal{O}(K|\mathcal{N}|)$. This can be improved by using a sparse storage as shown in [11].

## 4.4 Experimental results

The experiments are conducted on a computer with 2.27GHz dual quad-core Intel Xeon CPUs and 48GB main memory.

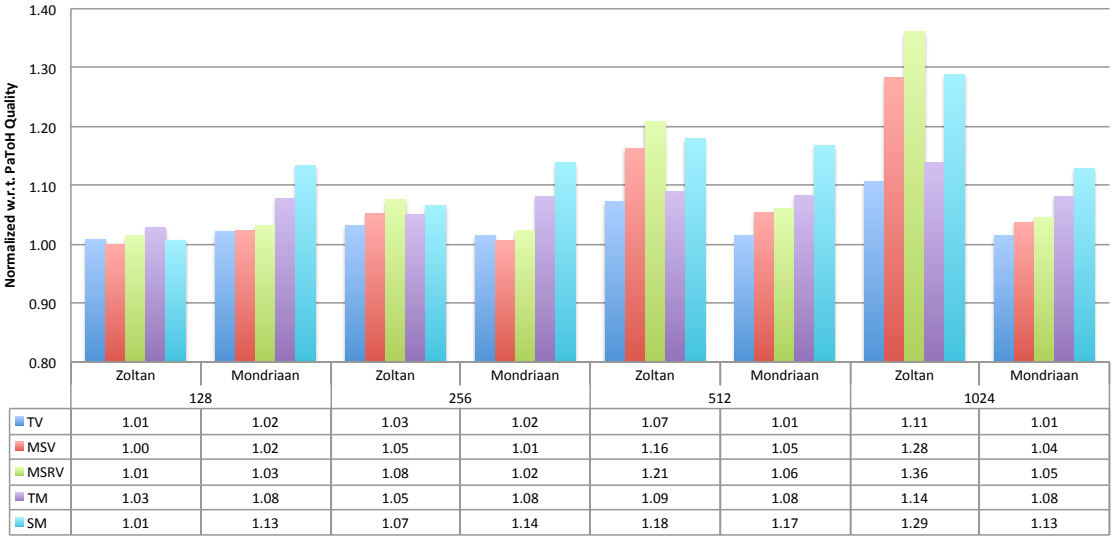
We compare the quality and efficiency of the partitioners UMPa, Zoltan [64], Mondriaan [142] (`version 4.0`) and PaToH [38] (in `default` setting)[1]. UMPa is implemented in C++, while the other partitioners are implemented in C. All implementations are compiled with the `gcc` suite version 4.4.4. To obtain our data set, we used several graphs with 100K–1,500K vertices and 409K–38,354K edges from the 10th DIMACS implementation challenge dataset [66] which contains 38 graphs from eight different classes. The names and the details of these graphs are given in Table 4.1. These graphs are translated into hypergraphs using column-net model to fit the problem definition of 10th DIMACS challenge[2]. In all of our experiments, the vertices have unit weights, while the nets may have non-uniform costs (as in the ComputationalTask graphs). Although Zoltan hypergraph partitioner is designed to work on distributed memory machines, all executions are run serially in our experiments. The reported results are the averages of five different executions. The performances of the partitioners are evaluated for five different communication metrics in addition to execution time. These are the maximum send volume (MSV), the maximum send/receive volume (MSRV), the total volume (TV), the maximum send message (MSM), and the total message (TM). The chosen hypergraphs are partitioned into 128, 256, 512, and 1024 parts with $\varepsilon$=0.03.

[1]The `default` setting of PaToH is chosen instead of `quality` setting, as the `quality` option obtains only 4% better cut sizes. On the other hand, the `default` option is 3.56 times faster.

[2] http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf

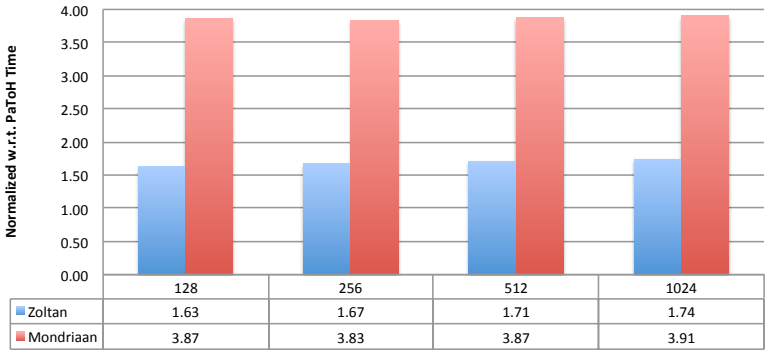
We first compare the execution time and the quality of the existing hypergraph partitioning tools. Although Mondriaan uses a hypergraph model, the partitioner is specifically designed for matrices, and it does not accept non-uniform net weights. For this reason, the experiment is run on 36 (out of 38) graphs excluding the ones in the ComputationalTask class. We adjust Mondriaan as described in [23] for the experiment. Figure 4.3 gives the communication metrics and execution times of Zoltan and Mondriaan normalized w.r.t. PaToH. As seen in the figure, Mondriaan obtains partitions whose TV values are similar to that of PaToH on TV. However, its quality on the other metrics is slightly worse. Furthermore, it is significantly slower than PaToH. Although the partitions found by Zoltan yield similar communication metrics when $K$ is small, Zoltan's partitioning quality gets worse as $K$ increases. We attribute this to simplified refinement implementations for the parallelization purposes. Since PaToH obtains the best quality and the best serial execution time, we compare the performance of UMPa against PaToH for the rest of the experiments.

Figures 4.4 and 4.5 show the average values of the metrics normalized with respect to the corresponding average metric value of PaToH partition. Each column of the table (and its visual representation as a group of five bars) corresponds to an UMPa variant with a different metric and tie-breaking combination, while each row of the table corresponds to the values with respect to a different communication metric. We experimented with four main variants UMPa$_{\mathsf{X}}$ with the primary metric




| | Zoltan | Mondriaan | Zoltan | Mondriaan | Zoltan | Mondriaan | Zoltan | Mondriaan |
|---|---|---|---|---|---|---|---|---|
| | 128 | | 256 | | 512 | | 1024 | |
| TV | 1.01 | 1.02 | 1.03 | 1.02 | 1.07 | 1.01 | 1.11 | 1.01 |
| MSV | 1.00 | 1.02 | 1.05 | 1.01 | 1.16 | 1.05 | 1.28 | 1.04 |
| MSRV | 1.01 | 1.03 | 1.08 | 1.02 | 1.21 | 1.06 | 1.36 | 1.05 |
| TM | 1.03 | 1.08 | 1.05 | 1.08 | 1.09 | 1.08 | 1.14 | 1.08 |
| SM | 1.01 | 1.13 | 1.07 | 1.14 | 1.18 | 1.17 | 1.29 | 1.13 |

(a) Normalized communication metrics




| | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|
| Zoltan | 1.63 | 1.67 | 1.71 | 1.74 |
| Mondriaan | 3.87 | 3.83 | 3.87 | 3.91 |

(b) Normalized execution times

Figure 4.3: The normalized communication metrics and execution times of Zoltan and Mondriaan w.r.t. PaToH for $K \in \{128, 256, 512, 1024\}$.

$\mathsf{X} \in \{\mathsf{TV}, \mathsf{MSV}, \mathsf{TM}, \mathsf{MSM}\}$. For each of these variants, we tried different combinations where in the first one only the primary metric is taken into account with no tie-breaking. Then, except for UMPa$_{\mathsf{TV}}$, we obtained a variant which use additional metrics (shown with a '+' sign in the figures). At last, we added tie-breaking and obtained the full variant (shown with "+ PW" in the figures). We executed each of these 11 variants and PaToH on all 38 hypergraphs 5 times and obtain 5 different partitions. For each variant/metric pair, the geometric mean of these 190 executions are computed, and the averages for the UMPa variants are normalized with respect to PaToH's averages. The figures show these normalized values.

As the first two columns in the tables (and the first two bar groups) of the figures show, for various $K$ values, UMPa$_{\mathsf{TV}}$ is as good as PaToH for all the communication metrics. Furthermore, its MSV and MSRV (2nd and 3rd rows/bars) values are 4-8% better than PaToH for all $K$ values. The tie-breaking scheme (the second column denoted with PW), which uses the part weight information, improves the performance of UMPa$_{\mathsf{TV}}$ (the first column) by around 2%. Since the direction information is not important while minimizing TV, the undirected hypergraph model and recursive bisection can be used as is. Therefore, the proposed directed hypergraph model and the $K$-way partitioning do not have an advantage for TV unlike for the other communication metrics as we discuss below.

We experimented with three UMPa$_{\mathsf{MSV}}$ variants to carefully evaluate the effects of the secondary and tertiary objectives and the tie-breaking mechanism. Although

minimizing only with the primary metric and no tie-breaking (column 3) obtains 7–11% better MSV value, with respect to the other metrics, UMPa$_{\mathsf{MSV}}$ is worse than PaToH. Using MSRV and TV as the secondary and tertiary objectives (column 4) has positive effects on the other communication metrics. Moreover, this approach further improves the primary communication metric MSV. This variant of UMPa$_{\mathsf{MSV}}$ doubles the improvement on the primary metric MSV and reduces it 17–22% on the average. Furthermore, it also obtains decent values for the other metrics with respect to PaToH. Extending the refinement with part weight tie-breaking (column 5) also reduces all the volume-based communication metrics around 2%. These observations are not only true on the average but also for most of the graphs in the dataset. For all partitionings with different $K$ values, UMPa$_{\mathsf{MSV}}$ with tie-breaking obtains more than 20% improvement in MSV for 12 graphs. The improvements is between 10% and 20% for 13 graphs, and between 2% to 10% for the remaining 13 graphs.

Only with a primary metric, UMPa$_{\mathsf{TM}}$ with no tie-breaking (column 6) improves TM about 7–12%. It also improves MSM with respect to PaToH, but its effect is almost always negative for the volume-based metrics. Using TV as the secondary objective to UMPa$_{\mathsf{TM}}$ (column 7) reduces the increase on these metrics. Furthermore, it also improves TM by 2–3% more. Similar to other cases, the communication metrics are improved around 3% more with the addition of tie-breaking (column 8).

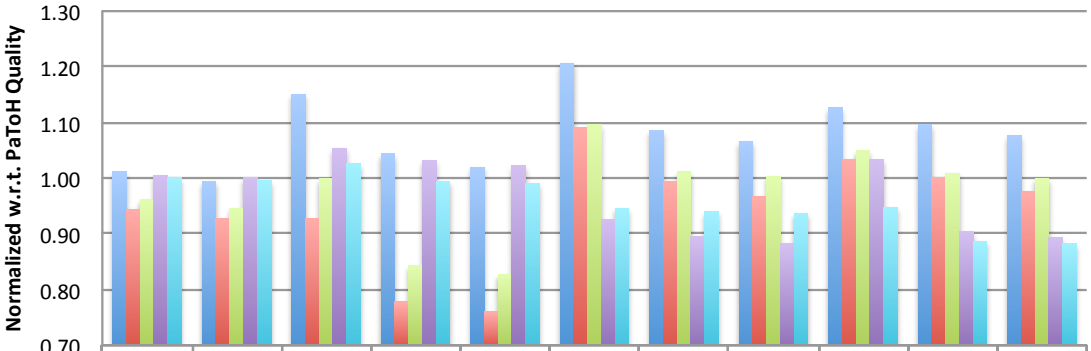
The results for UMPa$_{\mathsf{MSM}}$ are similar to that of other variants. Only with the primary objective function (column 9), UMPa$_{\mathsf{MSM}}$ obtains 5–7% better MSM values

than PaToH. Using TM and TV as the secondary and tertiary objectives (column 10) makes the difference around 7–19%. A further 2–3% improvement is obtained with the addition of tie-breaking (column 11).

Figure 4.6 shows the UMPa variants' runtimes normalized with respect to PaToH. As a $K$-way partitioner, the complexity of the refinement is linear with the number of parts for UMPa$_{\mathsf{TV}}$ and UMPa$_{\mathsf{TM}}$, and quadratic for UMPa$_{\mathsf{MSV}}$ and UMPa$_{\mathsf{MSM}}$. Therefore, as $K$ increases, the proposed direct $K$-way method is expected to be slower than a recursive-bisection-based partitioner. For 128 parts, except UMPa$_{\mathsf{MSV}}$, the execution times are closer to that of PaToH. However, when the number of parts get bigger, the slowdown increases.

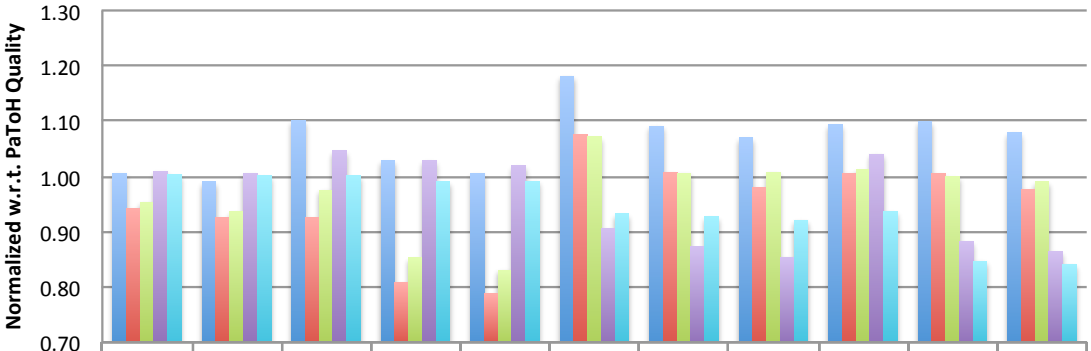
Table 4.1 gives the individual execution times and the volume metric results of PaToH for $K = 512$. The normalized execution times and normalized volume metrics of the UMPa variants are listed on the right side of the table. As seen in the table, the best TV improvement of the UMPa$_{\mathsf{TV}}$ is 18% improvement on G_n_pin_pout, while it obtains 5% TV degrade on eu-2005 graph. UMPa$_{\mathsf{MSV}}$ obtains its best MSV improvement on preferentialAttachment as 85%, while it obtains its worst improvement on thermal2 graph by 1%. UMPa$_{\mathsf{TM}}$ and UMPa$_{\mathsf{MSM}}$ have their best improvements on smallworld graph as 55% and 57%, respectively.

As Table 4.1 shows, the Citation and Clustering graphs have relatively large MSV, TM, and MSM values compared to other classes. For these graphs, the UMPa variants obtain decent improvements. However, when these metrics have small values, e.g.,




| | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | | UMPa$_{TM}$ | | | UMPa$_{MSM}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | +PW | | +MSRV +TV | +PW | | +TV | +PW | | +TM +TV | +PW |
| TV | 1.01 | 0.99 | 1.15 | 1.04 | 1.02 | 1.21 | 1.09 | 1.07 | 1.13 | 1.10 | 1.08 |
| MSV | 0.94 | 0.93 | 0.93 | 0.78 | 0.76 | 1.09 | 0.99 | 0.97 | 1.03 | 1.00 | 0.98 |
| MSRV | 0.96 | 0.95 | 1.00 | 0.84 | 0.83 | 1.10 | 1.01 | 1.00 | 1.05 | 1.01 | 1.00 |
| TM | 1.00 | 1.00 | 1.05 | 1.03 | 1.02 | 0.93 | 0.90 | 0.88 | 1.03 | 0.90 | 0.89 |
| MSM | 1.00 | 0.99 | 1.03 | 0.99 | 0.99 | 0.95 | 0.94 | 0.94 | 0.95 | 0.89 | 0.88 |

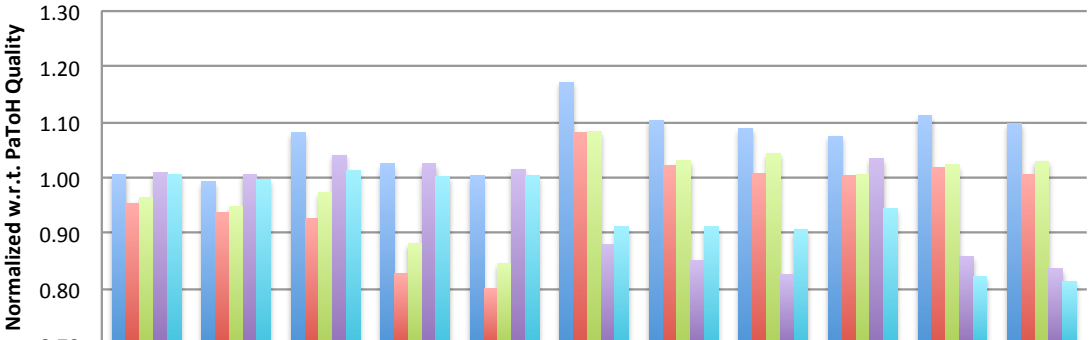
(a) $K = 128$




| | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | | UMPa$_{TM}$ | | | UMPa$_{MSM}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | +PW | | +MSRV +TV | +PW | | +TV | +PW | | +TM +TV | +PW |
| TV | 1.01 | 0.99 | 1.10 | 1.03 | 1.01 | 1.18 | 1.09 | 1.07 | 1.09 | 1.10 | 1.08 |
| MSV | 0.94 | 0.93 | 0.93 | 0.81 | 0.79 | 1.08 | 1.01 | 0.98 | 1.01 | 1.00 | 0.98 |
| MSRV | 0.95 | 0.94 | 0.98 | 0.85 | 0.83 | 1.07 | 1.01 | 1.01 | 1.01 | 1.00 | 0.99 |
| TM | 1.01 | 1.01 | 1.05 | 1.03 | 1.02 | 0.90 | 0.87 | 0.85 | 1.04 | 0.88 | 0.86 |
| MSM | 1.00 | 1.00 | 1.00 | 0.99 | 0.99 | 0.93 | 0.93 | 0.92 | 0.94 | 0.85 | 0.84 |

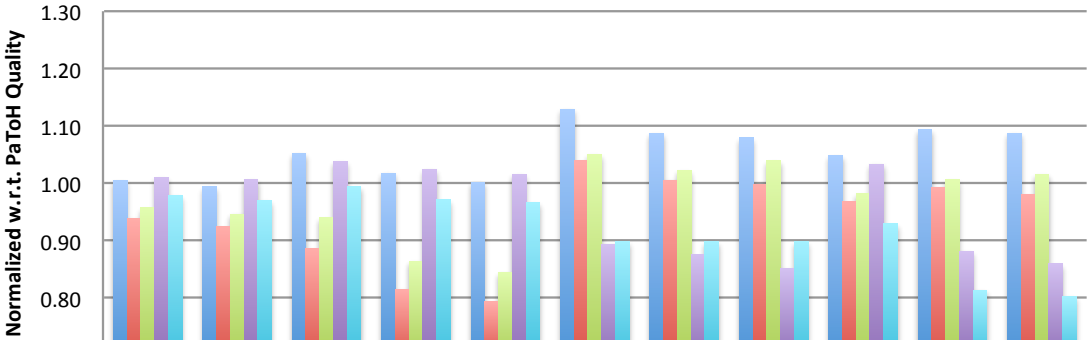
(b) $K = 256$

Figure 4.4: The normalized metrics of UMPa w.r.t. PaToH for $K = 128$ and $K = 256$.




| | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | | UMPa$_{TM}$ | | | UMPa$_{MSM}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | +PW | | +MSRV +TV | +PW | | +TV | +PW | | +TM +TV | +PW |
| TV | 1.01 | 0.99 | 1.08 | 1.02 | 1.00 | 1.17 | 1.10 | 1.09 | 1.07 | 1.11 | 1.10 |
| MSV | 0.95 | 0.94 | 0.93 | 0.83 | 0.80 | 1.08 | 1.02 | 1.01 | 1.00 | 1.02 | 1.01 |
| MSRV | 0.96 | 0.95 | 0.97 | 0.88 | 0.85 | 1.08 | 1.03 | 1.04 | 1.01 | 1.02 | 1.03 |
| TM | 1.01 | 1.01 | 1.04 | 1.02 | 1.02 | 0.88 | 0.85 | 0.83 | 1.03 | 0.86 | 0.84 |
| MSM | 1.01 | 1.00 | 1.01 | 1.00 | 1.00 | 0.91 | 0.91 | 0.91 | 0.94 | 0.82 | 0.81 |

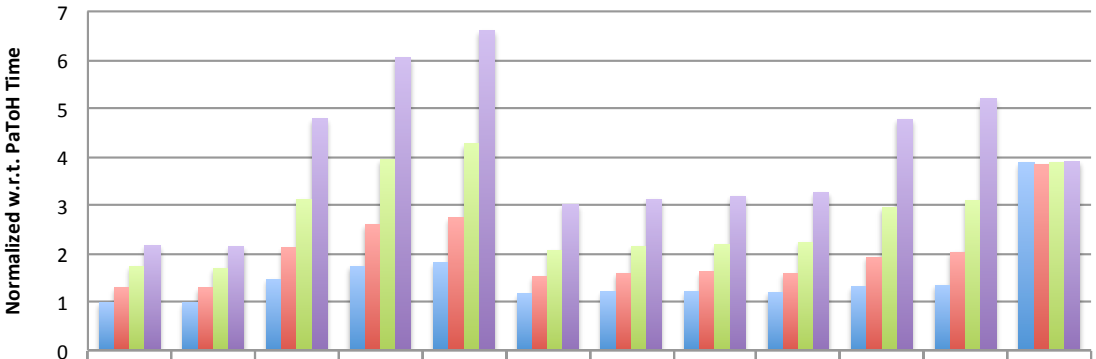
(a) $K = 512$




| | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | | UMPa$_{TM}$ | | | UMPa$_{MSM}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | +PW | | +MSRV +TV | +PW | | +TV | +PW | | +TM +TV | +PW |
| TV | 1.00 | 0.99 | 1.05 | 1.02 | 1.00 | 1.13 | 1.09 | 1.08 | 1.05 | 1.09 | 1.09 |
| MSV | 0.94 | 0.92 | 0.89 | 0.81 | 0.79 | 1.04 | 1.01 | 1.00 | 0.97 | 0.99 | 0.98 |
| MSRV | 0.96 | 0.95 | 0.94 | 0.86 | 0.84 | 1.05 | 1.02 | 1.04 | 0.98 | 1.01 | 1.01 |
| TM | 1.01 | 1.01 | 1.04 | 1.02 | 1.02 | 0.89 | 0.87 | 0.85 | 1.03 | 0.88 | 0.86 |
| MSM | 0.98 | 0.97 | 0.99 | 0.97 | 0.97 | 0.90 | 0.90 | 0.90 | 0.93 | 0.81 | 0.80 |

(b) $K = 1024$

Figure 4.5: The normalized metrics of UMPa w.r.t. PaToH for $K = 512$ and $K = 1024$.




| | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | | UMPa$_{TM}$ | | | UMPa$_{MSM}$ | | | Mond |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | +PW | | +MSRV +TV | +PW | | +TV | +PW | | +TM +TV | +PW | |
| 128 | 1.00 | 0.99 | 1.47 | 1.74 | 1.80 | 1.16 | 1.21 | 1.22 | 1.19 | 1.32 | 1.35 | 3.87 |
| 256 | 1.30 | 1.29 | 2.12 | 2.59 | 2.74 | 1.53 | 1.59 | 1.62 | 1.59 | 1.92 | 2.03 | 3.83 |
| 512 | 1.73 | 1.68 | 3.11 | 3.94 | 4.26 | 2.06 | 2.15 | 2.18 | 2.23 | 2.95 | 3.10 | 3.87 |
| 1024 | 2.17 | 2.14 | 4.79 | 6.05 | 6.60 | 3.00 | 3.12 | 3.18 | 3.26 | 4.78 | 5.19 | 3.91 |

Figure 4.6: The normalized execution times of UMPa w.r.t. PaToH.

PaToH's MSM values for the Sparse class, the improvements are not significant. Since PaToH minimizes only the total communication volume, a small metric value implies that minimizing TV also minimizes that metric. We believe that this is due to the structure of the graphs. For example, the Delaunay graphs are planar. Hence, when PaToH puts similar vertices in the same part, the nets connected to a part will have similar pin sets. Even if some of these nets are in the cut, the corresponding communications can be packed into a few messages. The same argument is also true for the Random Geometric graphs where only the vertices close to each other in the Euclidian space are connected.

Table 4.1: The experiments results for $K = 512$. The actual execution time (TT) and the volume metrics of PaToH are given on the left side of the table. The normalized time and normalized primary volume metrics of UMPa variants (with full tie-breaking scheme) are given on the right side of the table.

| | | | | PaToH | | | | | UMPa$_{TV}$ | | UMPa$_{MSV}$ | | UMPa$_{TM}$ | | UMPa$_{MSM}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Graph | Class | Vertices | Edges | TT | TV | MSV | TM | MSM | TT | TV | TT | MSV | TT | TM | TT | MSM |
| citationCiteseer | Citation | 268,495 | 1,156,647 | 24.17 | 449,324 | 3,230 | 53,824 | 252 | 3.30 | 0.95 | 6.64 | 0.62 | 4.11 | 0.64 | 6.50 | 0.58 |
| coAuthorsCiteseer | Citation | 227,320 | 814,134 | 7.23 | 128,873 | 1,147 | 41,486 | 190 | 2.35 | 0.97 | 7.17 | 0.60 | 3.91 | 0.63 | 6.64 | 0.63 |
| coAuthorsDBLP | Citation | 299,067 | 977,676 | 10.87 | 274,463 | 2,524 | 78,531 | 300 | 3.07 | 0.97 | 8.72 | 0.59 | 4.98 | 0.61 | 7.84 | 0.67 |
| coPapersCiteseer | Citation | 434,102 | 16,036,720 | 249.29 | 656,097 | 5,100 | 53,467 | 198 | 0.22 | 0.95 | 1.56 | 0.77 | 0.36 | 0.69 | 1.12 | 0.74 |
| coPapersDBLP | Citation | 540,486 | 15,245,729 | 157.64 | 1,592,718 | 9,388 | 106,073 | 334 | 0.70 | 0.90 | 3.40 | 0.77 | 1.03 | 0.71 | 2.42 | 0.81 |
| caidaRouterLevel | Clustering | 192,244 | 609,066 | 8.39 | 132,375 | 4,886 | 20,491 | 384 | 2.81 | 0.98 | 6.93 | 0.60 | 3.78 | 0.74 | 6.16 | 0.61 |
| cnr-2000 | Clustering | 325,557 | 2,738,969 | 28.14 | 288,308 | 4,241 | 6,411 | 154 | 3.26 | 0.95 | 6.93 | 0.73 | 3.90 | 0.75 | 5.73 | 0.78 |
| eu-2005 | Clustering | 862,664 | 16,138,468 | 215.40 | 1,568,558 | 11,704 | 22,600 | 251 | 2.81 | 1.05 | 6.68 | 0.73 | 3.09 | 0.68 | 5.22 | 0.70 |
| G_n_pin_pout | Clustering | 100,000 | 501,198 | 12.15 | 719,743 | 1,584 | 230,910 | 480 | 5.20 | 0.82 | 7.01 | 0.81 | 6.32 | 0.62 | 9.14 | 0.71 |
| in-2004 | Clustering | 1,382,908 | 13,591,473 | 160.83 | 449,886 | 7,297 | 10,898 | 182 | 1.08 | 1.01 | 2.83 | 0.68 | 1.46 | 0.59 | 2.74 | 0.66 |
| preferentialAttachment | Clustering | 100,000 | 499,985 | 16.91 | 755,796 | 26,256 | 190,519 | 511 | 5.77 | 0.97 | 8.48 | 0.15 | 6.34 | 0.67 | 6.29 | 1.00 |
| smallworld | Clustering | 100,000 | 499,998 | 5.29 | 235,807 | 558 | 137,196 | 302 | 5.02 | 0.96 | 7.84 | 0.89 | 6.73 | 0.55 | 11.82 | 0.57 |
| 1,280,000.00 | ComputationalTask | 1,279,253 | 1,285,172 | 14.72 | 2,194,852,000 | 69,956,800 | 3,033 | 47 | 2.59 | 1.01 | 2.57 | 0.90 | 2.65 | 0.57 | 2.59 | 0.90 |
| 320,000.00 | ComputationalTask | 320,142 | 322,473 | 9.30 | 1,066,920,000 | 50,649,680 | 2,481 | 51 | 1.58 | 1.02 | 2.15 | 0.56 | 2.01 | 0.72 | 2.09 | 0.57 |
| delaunay_n17 | Delaunay | 131,072 | 393,176 | 3.50 | 29,990 | 81 | 3,028 | 10 | 2.98 | 1.01 | 6.05 | 0.95 | 3.37 | 1.00 | 4.27 | 0.98 |
| delaunay_n18 | Delaunay | 262,144 | 786,396 | 6.28 | 41,748 | 112 | 3,032 | 12 | 2.89 | 1.01 | 5.15 | 0.96 | 3.42 | 1.00 | 4.14 | 0.89 |
| delaunay_n19 | Delaunay | 524,288 | 1,572,823 | 11.24 | 58,493 | 165 | 3,035 | 11 | 1.50 | 1.01 | 4.38 | 0.89 | 2.11 | 1.00 | 2.82 | 0.85 |
| delaunay_n20 | Delaunay | 1,048,576 | 3,145,686 | 20.60 | 82,067 | 213 | 3,026 | 10 | 1.36 | 1.01 | 3.49 | 0.98 | 1.99 | 1.00 | 2.50 | 0.94 |
| rgg_n_2_17_s0 | RandomGeometric | 131,072 | 728,753 | 4.26 | 29,206 | 96 | 2,864 | 12 | 2.08 | 1.00 | 5.70 | 0.89 | 2.70 | 0.91 | 3.21 | 0.83 |
| rgg_n_2_18_s0 | RandomGeometric | 262,144 | 1,547,283 | 8.37 | 43,052 | 132 | 2,885 | 11 | 1.70 | 0.99 | 4.06 | 0.93 | 2.26 | 0.92 | 2.65 | 0.91 |
| rgg_n_2_19_s0 | RandomGeometric | 524,288 | 3,269,766 | 16.95 | 63,291 | 193 | 2,898 | 11 | 1.56 | 1.00 | 3.51 | 0.89 | 1.85 | 0.96 | 2.31 | 0.89 |
| rgg_n_2_20_s0 | RandomGeometric | 1,048,576 | 6,891,620 | 35.78 | 94,118 | 288 | 2,896 | 9 | 0.69 | 1.00 | 2.31 | 0.88 | 1.04 | 0.96 | 1.38 | 0.98 |
| af_shell10 | Sparse | 1,508,065 | 25,582,130 | 103.09 | 252,404 | 647 | 2,886 | 8 | 0.32 | 1.02 | 1.13 | 0.96 | 0.40 | 1.00 | 0.73 | 1.00 |
| af_shell9 | Sparse | 504,855 | 8,542,010 | 35.56 | 144,713 | 370 | 2,804 | 8 | 0.51 | 1.00 | 1.61 | 0.95 | 0.60 | 1.00 | 1.03 | 1.00 |
| audikw1 | Sparse | 943,695 | 38,354,076 | 357.19 | 947,370 | 2,824 | 5,284 | 21 | 0.27 | 1.02 | 1.57 | 0.94 | 0.35 | 0.95 | 0.75 | 0.98 |
| ecology1 | Sparse | 1,000,000 | 1,998,000 | 13.50 | 79,633 | 207 | 2,895 | 9 | 1.85 | 1.04 | 4.58 | 0.98 | 2.76 | 0.99 | 3.27 | 0.87 |
| ecology2 | Sparse | 999,999 | 1,997,996 | 13.46 | 79,373 | 205 | 2,895 | 9 | 1.86 | 1.04 | 4.58 | 0.98 | 2.82 | 0.99 | 3.24 | 0.91 |
| G3_circuit | Sparse | 1,585,478 | 3,037,674 | 26.09 | 147,446 | 423 | 2,975 | 12 | 1.56 | 1.02 | 5.40 | 0.96 | 2.28 | 0.98 | 2.91 | 1.02 |
| ldoor | Sparse | 952,203 | 22,785,136 | 121.70 | 216,259 | 661 | 3,008 | 14 | 0.25 | 1.00 | 1.61 | 0.88 | 0.30 | 1.00 | 0.62 | 1.01 |
| thermal2 | Sparse | 1,227,087 | 3,676,134 | 25.88 | 95,963 | 250 | 2,831 | 9 | 0.77 | 1.03 | 3.12 | 0.99 | 1.34 | 1.00 | 1.79 | 1.00 |
| belgium.osm | Street | 1,441,295 | 1,549,970 | 10.01 | 12,526 | 75 | 3,000 | 14 | 1.29 | 1.02 | 2.06 | 0.87 | 1.59 | 0.82 | 1.85 | 0.68 |
| luxembourg.osm | Street | 114,599 | 119,666 | 0.85 | 3,397 | 20 | 2,144 | 9 | 3.17 | 1.01 | 4.83 | 0.85 | 3.82 | 0.81 | 4.29 | 0.74 |
| 144 | Walshaw | 144,649 | 1,074,393 | 11.75 | 128,880 | 442 | 5,665 | 33 | 2.81 | 0.99 | 6.64 | 0.91 | 3.20 | 0.93 | 4.02 | 0.88 |
| 598a | Walshaw | 110,971 | 741,934 | 8.52 | 103,693 | 372 | 5,514 | 31 | 3.27 | 0.99 | 6.93 | 0.88 | 3.67 | 0.93 | 4.32 | 0.83 |
| auto | Walshaw | 448,695 | 3,314,611 | 34.37 | 273,173 | 806 | 5,598 | 29 | 1.89 | 1.00 | 6.96 | 0.89 | 2.44 | 0.94 | 3.16 | 0.80 |
| fe_ocean | Walshaw | 143,437 | 409,593 | 5.51 | 83,761 | 247 | 5,284 | 19 | 3.48 | 1.04 | 7.36 | 0.94 | 4.18 | 0.90 | 4.92 | 0.91 |
| m14b | Walshaw | 214,765 | 1,679,018 | 16.82 | 158,215 | 563 | 5,347 | 37 | 2.90 | 1.00 | 6.94 | 0.89 | 3.39 | 0.94 | 4.52 | 0.85 |
| wave | Walshaw | 156,317 | 1,059,331 | 10.94 | 142,792 | 421 | 6,758 | 37 | 2.81 | 1.00 | 6.87 | 0.89 | 3.13 | 0.92 | 4.23 | 0.72 |
| | | | **GEOMEAN** | 19.20 | 234,643 | 1,167 | 7,954 | 38 | 1.68 | 0.99 | 4.26 | 0.80 | 2.18 | 0.83 | 3.10 | 0.81 |
| | | | | | | | | **BEST** | 0.22 | 0.82 | 1.13 | 0.15 | 0.30 | 0.55 | 0.62 | 0.57 |
| | | | | | | | | **WORST** | 5.77 | 1.05 | 8.72 | 0.99 | 6.73 | 1.00 | 11.82 | 1.02 |

## 4.5 Summary

A directed hypergraph model and a multi-level partitioner UMPa are proposed in this chapter. The partitioner uses a novel $K$-way refinement heuristic employing a tie-breaking scheme to handle multiple communication metrics. UMPa yields good communication patterns by reducing multiple communication metrics all together.

Although most of the previous research has mainly focused on the minimization of the total communication volume, there are studies on the minimization of multiple metrics. Existing hypergraph-based solutions on multiple metrics follow a two-phase approach where in each phase a different metric is minimized, sometimes at the expense of others. UMPa follows a one-phase approach where all the communication cost metrics are effectively minimized in a multi-objective setting where reductions in all metrics can be achieved at the same time, thanks to the proposed directed hypergraph model (undirected models were discussed to be inadequate). Experiments with large hypergraphs are conducted and up to 1024 parts in which the proposed approach yielded improvements over the state-of-the-art hypergraph partitioner PaToH. For example, for $K = 1024$, UMPa$_{\mathsf{MSM}}$ produces 20% and 14% better partitions in terms of MSM and TM, respectively.

A future work is to speed up UMPa and the proposed refinement approach by implementing them on modern parallel architectures. Investigating the hypergraph partitioning models and communication metrics for hierarchical memory systems,

such as cluster of multi-socket, multi-core machines with accelerators is another direction of research. Also, another future work is to extend UMPa to handle the directed hypergraphs with multiple senders. Such directed hypergraph models can be used to model computations which require communications for outputs or for both inputs and outputs. Columnwise or nonzero-based sparse matrix partitioning models yield such directed hypergraphs.

# Chapter 5: Hypergraph Partitioning Run time Improvements

## 5.1 Multithreaded Clustering algorithms for scalable Multi-level Hypergraph Partitioning

Hypergraph partitioning is an important problem widely encountered in parallelization of complex and irregular applications from various domains including VLSI design [101], parallel scientific computing [37, 42], sparse matrix reordering [41], static and dynamic load balancing [43], software engineering [22], cryptosystem analysis [24], and database design [103, 110, 128]. Being such an important problem, considerable effort has been put in providing tool support, see hMeTiS [95], MLpart [35], Mondriaan [142], Parkway [134], PaToH [38], and Zoltan [29]. However, most of the current-state-of-the art software libraries that provide tool support for the hypergraph partitioning problem are designed and implemented before the game-changing advancements in multi-core computing. Hence, analyzing the structure of those tools for designing multithreaded versions of the algorithms is a crucial task.

All the tools above follow the multi-level approach. This approach consists of three phases: coarsening, initial partitioning, and uncoarsening. In the coarsening phase, the original hypergraph is reduced to a much smaller hypergraph after a series of coarsening levels. At each level, vertices that are deemed to be similar are grouped to form vertex clusters, and a new hypergraph is formed by unifying a cluster as a single vertex. That is, the clusters become the vertices for the next level. In the initial partitioning phase, the coarsest hypergraph is partitioned. In the uncoarsening phase, the partition found in the second phase is projected back to the original hypergraph where the partition is locally refined on the hypergraphs associated with each coarsening level.

The coarsening phase is the most important phase of the multi-level approach. This is for the following three reasons. First, the worst-case running time complexity of this phase is higher than the other two phases (initial partitioning and uncoarsening phases have, in most common implementations, linear worst-case running time complexity). Second, as the uncoarsening level performs local improvements, the quality of a partition is highly affected by the quality of the coarsening phase. For example, given a hypergraph, a coarsening algorithm, a conventional initial partitioning algorithm and a refinement algorithm based on the most common ones, very slight variations on vertex similarity metrics can effect the performance quite significantly (see for example the start of Section 5.1 of [138]). Third, it is usually the case that the

better the coarsening, the faster the uncoarsening phase is. Therefore, the coarsening phase also affects the running time of the other phases.

The aim of this chapter is to efficiently parallelize the coarsening phase of PaToH, a well-known and commonly used hypergraph partitioning tool. The algorithmic kernel of this phase is a *clustering* algorithm that marks similar vertices to be coalesced. There are two classes of clustering algorithms in PaToH. The algorithms in the first class allow at most two vertices in a cluster. These algorithms are called *matching-based* or *matching* algorithms in short. The algorithms in the second class, called *agglomerative* algorithms, allow any number of vertices to become together to form a cluster. The most effective clustering algorithms in PaToH are agglomerative ones whereas the fastest ones are matching based. We propose efficient parallelization of these two classes of algorithms (Section 5.1.2). We report practical experiments with PaToH (and its coarsening phase alone) on a recent multicore architecture (Section 5.1.3). The proposed techniques are easily applicable to some other sequential hypergraph partitioners, since they use the same multilevel approach and have similar data structures. Refer to Section 2.2.2 for the hyperhraph partitioning problem formulation and background information on the multilevel hypergraph partitioning.

### 5.1.1 Background and related work

#### 5.1.1.1 Clustering algorithms for hypergraph partitioning

As said before, there are two classes of clustering algorithms: matching-based ones and agglomerative ones. The matching-based ones put at most two similar vertices

in a cluster whereas the agglomerative ones allow any number of similar vertices. There are various similarity metrics—see for example [6, 37, 95]. All these metrics are defined on two adjacent vertices (one of them can be a vertex cluster). Two vertices are adjacent if they share a net, i.e., the vertices $u$ and $v$ are matchable if $\mathcal{N}_{uv} = \mathsf{nets}[u] \cap \mathsf{nets}[v] \neq \emptyset$. In order to find a given vertex $u$'s adjacent vertices, one needs to visit each net $n \in \mathsf{nets}[u]$ and then visit each vertex $v \in \mathsf{pins}[n]$. Therefore, the computational complexity of the clustering algorithms is at least in the order of $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$. As mentioned in the introduction, the other two phases of the multi-level approach have a linear time worst case time complexity. As $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$ is most likely to be the bottleneck, an effective clustering algorithm's worst case running time should not pass this limit for the algorithm to be efficient as well.

The sequential implementations of the clustering algorithms in PaToH proceed in the following way to have a running time proportional to $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$. The vertices are visited in a given (possibly random) order and each vertex $u$, if not clustered yet, is tried to be clustered with the most similar vertex or cluster. In the matching-based ones, the current vertex $u$ if not matched yet, chooses one of its unmatched adjacent vertices according to a criterion. If such a vertex $v$ exists, the matched pair $u$ and $v$ are marked as a cluster of size two. If there is no unmatched adjacent vertex of $u$, then vertex $u$ remains as a singleton cluster. In the agglomerative ones, the current vertex $u$, if not marked to be in a cluster yet, can choose a cluster to join (thus forming a cluster of size at least three), or can create another cluster with one

of its unmatched adjacent vertices (thus forming a cluster of size two). Hence in agglomerative clustering, vertex $u$ never remains as a singleton, as long as it is not isolated (i.e., not connected to any net).

For the clustering algorithms in this chapter, there exists a representative vertex for each cluster. When a vertex $u \in \mathcal{V}$ is put into a cluster, we set $\mathsf{rep}[u]$ to the representative of this group. When a singleton vertex $u$ chooses another one $v$, we choose one as the representative and set $\mathsf{rep}[u]$ and $\mathsf{rep}[v]$ accordingly. For all the algorithms, we assume that $\mathsf{rep}[u]$ is initially **null** for all $u \in \mathcal{V}$. This will also be true if $u$ remains singleton at the end of the algorithm.

Algorithm 14 presents one of the matching-based clustering algorithms that are available in PaToH. In this algorithm, the vertex $u$ (if not matched yet) is matched with currently unmatched neighbor $v$ with the maximum connectivity, where the connectivity refers to the sum of the costs of the common nets. This matching algorithm is called as Heavy Connectivity Matching (HCM) in PaToH [37], and Inner Product Matching (IPM) in Zoltan [64] and Mondriaan [142]. One can have different variations of this algorithm by changing the vertex visit order (line 1) and/or using different scaling schemes while computing the contribution of each net to its pins (line 6). The array $\mathsf{conn}[\cdot]$ of size $|\mathcal{V}|$ is necessary to compute the connectivity of the vertex $u$ and all its adjacent vertices in time linearly proportional to the number of adjacent vertices. The operation at line 3 is again for efficiency. It removes the matched vertices from $\mathsf{pins}[n]$, hence the next searches on $\mathsf{pins}[n]$ will take less time.

**Algorithm 14:** Sequential greedy matching (HCM)

```
   Data: H = (V, N), rep
1  for each vertex u ∈ V in a given order do
       if rep[u] = null then
2          ensuremath adj_u ← {};
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if rep[v] ≠ null then
3                      pins[n] ← pins[n] \{v};
                   else
4                      if v ∉ ensuremath adj_u then
5                          ensuremath adj_u ← ensuremath adj_u ∪ {v};
6                      conn[v] ← conn[v] + c[n];
           v* ← u;
           conn* ← 0;
           for each vertex v ∈ ensuremath adj_u do
               if conn[v] > conn* and v ≠ u then
                   conn* ← conn[v];
                   v* ← v;
               conn[v] ← 0;
           if u ≠ v* then
               rep[v* ] ← u;
               rep[u] ← u;
```

Algorithm 15 presents one of the agglomerative clustering algorithms that are available in PaToH. Similar to the sequential HCM algorithm, vertices are again visited in a given order. If a vertex $u$ has already been clustered, it is skipped. However, an unclustered vertex $u$ can choose to join an existing cluster, can start a new cluster with a vertex, or stay as a singleton cluster. Therefore, compared to the previous algorithm, all adjacent vertices of the current vertex $u$ are considered for

**Algorithm 15:** Sequential agglomer. clustering (HCC)

```
Data: H = (V, N), maxW, rep
for each vertex u ∈ V in a given order do
    if rep[u] = null then
        ensuremath adj_u ← {};
        for each net n ∈ nets[u] do
            for each vertex v ∈ pins[n] do
                if v ∉ ensuremath adj_u then
                    ensuremath adj_u ← ensuremath adj_u ∪ {v}
            conn[v] ← conn[v] + c[n]
        v* ← u;
        conn* ← 0;
        for each vertex v ∈ ensuremath adj_u do
            if u = v then
                continue;
            v^r ←rep[v] ;
            if v^r = null then
                v^r ← v;
            if v^r ≠ v then
                conn[v^r] ← conn[v^r] + conn[v];
                conn[v] ← 0;
                ensuremath adj_u ← ensuremath adj_u ∪ {v^r}  \ {v};
1           totW ← w[u] + w[v^r];
            if conn[v^r] > conn* then
                if totW < maxW then
                    conn* ← conn[v^r];
                    v* ← v^r;
        for each vertex v ∈ ensuremath adj_u do
            conn[v] ← 0;
        if u ≠ v* then
            rep[v* ] ← v* ;
            rep[u] ← v* ;
            w[v* ] ← w[v* ] + w[u];
```

selection. In order to avoid building an extremely large cluster (which would cause load balance problem in initial partitioning and refinement phases), we also enforce that weight of a cluster must be smaller than a given value $maxW$. Our experience shows that such restriction is not needed in matching based clustering, since at each level only at most two vertices can be clustered together.

In Algorithm 15, we use the total shared net cost (heavy connectivity clustering) as the similarity metric. In practice (and in our experiments), we use the absorption clustering metric (implemented in PaToH) which divides the contribution of each net to the number of its pins. That is, a net $n$ contributes $\mathsf{c}[n]/|\mathsf{pins}[n]|$ to the similarity value instead of $\mathsf{c}[n]$. This metric favors clustering vertices connected via nets of small sizes. The sequential code in PaToH also divides the overall similarity score between two vertices by the weight of the cluster which will contain $u$ (the value $totW$ at line 1). Hence, to compare the performance of our multithreaded clustering algorithms with PaToH, we also use this modified similarity metric in our implementations. However, for simplicity, we will continue to use the heavy connectivity clustering metric in the chapter.

#### 5.1.1.2 Metrics

We define the metrics of *cardinality* and *quality* to compare different clustering methods. The cardinality is defined as the number of clustering decisions taken by an algorithm, i.e.,

$$\text{cardinality} = \sum_{u \in \mathcal{V}, \mathsf{rep}[u] \neq \mathbf{null}} (|\{v \in \mathcal{V} : \mathsf{rep}[v] = u\}| - 1) \ .$$

In the multi-level framework, this represents the reduction on the number of vertices between two consecutive coarsening levels. The quality of a clustering is defined as the sum of the similarities between each vertex pair which resides in the same cluster, i.e.,

$$\text{quality} = \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{C}_u} \sum_{n \in \mathcal{N}_{uv}} \frac{\mathsf{c}[n]}{2} \ ,$$

where $\mathcal{C}_u = \{v \in \mathcal{V} \setminus \{u\} : \mathsf{rep}[v] = \mathsf{rep}[u] \text{ and } \mathsf{rep}[v] \neq \mathbf{null}\}$ is the set of vertices which are in the same cluster with $u$, and $\mathcal{N}_{uv}$ is the set of nets shared by $u$ and $v$.

Although the definitions are generic to be used for both matching-based and agglomerative clustering, we do not use these criteria to compare a matching-based clustering heuristic with an agglomerative one, since the latter has an obvious advantage.

#### 5.1.1.3 Related work

For a given hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, let $A$ be the vertex-net incidence matrix, i.e., the rows of $A$ correspond to the vertices of $\mathcal{H}$, and the columns of $A$ correspond to the nets of $\mathcal{H}$ such that $a_{vn} = 1$ iff $v \in \mathsf{pins}[n]$. Consider now the symmetric matrix $M = AA^T - diag(AA^T)$. The matrix $M$ can be effectively represented by an undirected graph $\mathcal{G}(M)$ with $|\mathcal{V}|$ vertices and having an edge of weight $m_{uv}$ between two vertices $u$ and $v$ if $m_{uv} \neq 0$. That is, there is a one-to-one correspondence between

the vertices of $\mathcal{H}$ and $\mathcal{G}(M)$. As $m_{uv} \neq 0$ iff the vertices $u$ and $v$ of $\mathcal{H}$ are adjacent, the correspondence implies that a matching among the vertices of $\mathcal{H}$ corresponds to a matching on the vertices of $\mathcal{G}(M)$. Therefore, various matching algorithms and heuristics for graphs can be used on $\mathcal{G}(M)$ to find a matching among the vertices of $\mathcal{H}$.

Bisseling and Manne [105] propose a distributed memory, 1/2-approximate algorithm to find weighted matchings in graphs. Building on this work, Çatalyürek et al. [46] present efficient distributed-memory parallel algorithms and scalable implementations. Halappanavar et al. [81] present an efficient shared-memory implementations for computing 1/2-approximate weighted matchings. Fagginger Auer and Bisseling [69] study an implementation of a parallel greedy graph matching on GPUs. For maximum cardinality matching problem, in a recent work, Patwary et al. [113] propose a distributed memory, sub-optimal algorithm. There also exist various studies for the parallelization of the maximum cardinality matching algorithms in the context of the bipartite graphs [12, 58, 59, 141].

There are a number of reasons why we cannot use aforementioned algorithms. First and foremost, storing the graph $\mathcal{G}(M)$ requires a large memory. The time required to compute this graph is about as costly as computing a matching in $\mathcal{H}$ in a sequential execution. Second, it is our experience (with the coarsening algorithms within the multi-level partitioner PaToH) that one does not need a maximum weighted matching, nor a maximum cardinality one, nor an approximation guarantee to find

helpful coarsening schemes. Third, while matching the vertices of a hypergraph, we sometimes need to avoid matched vertices become too big, or favor vertex clusters with smaller weights (due to multi-level nature of the partitioning algorithm), and vertices that are mostly related via nets of smaller size. These modifications can be incorporated into the graph $\mathcal{G}(M)$ by adjusting the edge weights (or by leaving some edges out). This will help reduce the memory requirements of the graph matching based algorithms. However, the computational cost of constructing the graph remains the same. Almost all of the most effective sequential clustering algorithms implemented in PaToH for coarsening purposes has the same worst case time complexity but are much faster in practice. We, therefore, cannot afford building the graph $\mathcal{G}(M)$ or its modified versions and call existing graph matching algorithms. Furthermore, agglomerative clustering algorithms cannot be accomplished by using the aforementioned algorithms.

We highlight that the matching-based clustering algorithms considered in this work can be perceived as a graph matching algorithm adjusted to work on an implicit representation of the graph $\mathcal{G}(M)$ or its modified versions. However, to the best of our knowledge, there is no immediate parallel graph-matching based algorithm that is analogous to the agglomerative clustering algorithms considered in this work (although variants of agglomerative coarsening for graphs exist, see for example [1] and [48]).

As a sanity check, we implemented a modified version of the sequential 1/2-approximation algorithm of Halappanavar et al. [81] which works directly on hypergraphs. In other words, instead of explicitly constructing the graph $\mathcal{G}(M)$, adjacencies of vertices are constructed on the fly, like Algorithm 14. We compared the quality and cardinality of this algorithm with that of the greedy sequential matching HCM. The approximation algorithm obtained matchings with better quality by 14% while the cardinalities were the same. However, this good performance comes with a significant execution time overhead, yielding 6.5 times slower execution. When we integrated the 1/2-approximation algorithm into the coarsening phase of PaToH, we observed that better matching quality helps the partitioner to obtain better cutsize. For example, when the partitioner is executed 10 times with random seeds, the average cutsize of 1/2-approximation algorithm is 8% better than the one obtained by using HCM. However, when we compare the minimum of these cutsizes, HCM outperforms the approximation algorithm by 2%. Moreover, the difference between the minimum cut obtained by using HCM with the average cut obtained by using the approximation algorithm is 14% in favor of HCM. After these preliminary experiments, we decided to parallelize HCM and HCC, since they are much faster, and one can obtain better cutsize by using them.

### 5.1.2 Multithreaded Clustering for Coarsening

In this section, we will present three novel parallel greedy clustering algorithms. The first two are matching-based and the third one is a greedy agglomerative clustering method.

#### 5.1.2.1 Multithreaded matching algorithms

To adapt the greedy sequential matching algorithm for multithreaded architectures, we use two different approaches. In the first one, we employ a locking mechanism which prevents inconsistent matching decisions between the threads. In the second approach, we let the algorithm run almost without any modifications and then use a conflict resolution mechanism to create a consistent matching.

The lock-based algorithm is given in Algorithm 16. The structure of the algorithm is similar to the sequential one except the lines 2 and 5, where we use the atomic CHECKANDLOCK operation. To lock a vertex $u$, this operation first checks if $u$ is already locked or not. If not, it locks it and returns **true**. Otherwise, it returns **false**. Its atomicity guarantees that a locked vertex is never considered for a matching. That is, both the visited vertex $u$ (at line 1) and the adjacent vertex $v$ must not be locked to consider the matching of $u$ and $v$. If they are, and if the similarity of $v$ is bigger than the current best (at line 3), the algorithm keeps $v$ as the best candidate $v^*$. When a better candidate is found, the old one is UNLOCKed to make it available for other threads (line 6), and to construct better matchings in terms of cardinality and quality.

**Algorithm 16:** Parallel lock-based matching

```
   Data: H = (V, N), rep
1  for each vertex u ∈ V in parallel do
2      if CHECKANDLOCK(u) then
           ensuremath adj_u ← {};
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ ensuremath adj_u then
                       ensuremath adj_u ← ensuremath adj_u ∪ {v};
                   conn[v] ← conn[v] + c[n];
           v* ← u;
           conn* ← 0;
3          for each vertex v ∈ ensuremath adj_u do
4              if conn[v] > conn* then
5                  if CHECKANDLOCK(v) then
                       if u ≠ v* then
6                          UNLOCK(v*);
                       conn* ← conn[v];
                       v* ← v;
               conn[v] ← 0;
           if u ≠ v* then
               rep[u] ← u ;
               rep[v* ] ← u ;
```

As a different approach without a lock mechanism, we modify the sequential code slightly and execute it in a multithreaded environment. If the **for** loop at line 1 of Algorithm 14 is executed in parallel, different threads may set $\mathsf{rep}[u]$ to different values for a vertex $u$. Hence, the rep array will contain inconsistent decisions. To solve this issue, one can make each thread use a private rep array and store all of its matching decisions locally. Then, a consistent matching can be devised from this information

in another phase. Another idea is keeping the sequential code (almost) as is, letting the threads create conflicts, and resolving the conflicts later. Our preliminary experiments show that there is not much difference between the performances of these two approaches in terms of the cardinality and the quality. However, the first one requires more memory: one rep array per thread compared to a shared one. Hence, we followed the second idea and use a conflict resolution scheme with $\mathcal{O}(|\mathcal{V}|)$ complexity. Algorithm 17 shows the pseudocode of our parallel resolution-based algorithm.

Our conflict resolution scheme starts at line 5 of Algorithm 17. Note that instead of setting a fixed representative for the matched vertices $u$ and $v^*$, we set their rep values to each other. This bidirectional information is then used in our resolution scheme to check if the rep array contains inconsistent information. That is if $u \neq \mathsf{rep}[\mathsf{rep}[u]]$ for a vertex $u$, we know that at least two threads matched either $u$ or $v$ with different vertices. If this is the case, the resolution scheme acts greedily and aggressively sets $\mathsf{rep}[u]$ to **null** indicating $u$ will be unmatched. After the first loop at line 5, which is executed in parallel, the rep array will contain the matching decisions consistent with each other. Then, with the last parallel loop, we set the representatives for each matched pair.

The proposed resolution scheme is sufficient to obtain a valid matching in the multithreaded setting. However, we slightly modify Algorithm 14 to obtain better matchings. Since each conflict will probably cost a pair, and losing a pair reduces matching cardinality and hence, quality, we desire lesser conflicts. To avoid them, at

**Algorithm 17:** Parallel resolution-based matching

```
   Data: H = (V, N), rep
   for each vertex u ∈ V in parallel do
       if rep[u] = null then
           ensuremath adj_u ← {};
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ ensuremath adj_u then
                       ensuremath adj_u ← ensuremath adj_u ∪ {v}
                   conn[v] ← conn[v] + c[n]
           v* ← u;
           conn* ← 0;
           for each vertex v ∈ ensuremath adj_u do
1              if rep[v] = null then
                   if conn[v] > conn* then
                       if u ≠ v then
                           conn* ← conn[v];
                           v* ← v;
               conn[v] ← 0;
2          if rep[u] = rep[v] = null then
3              rep[u] ← v* ;
4              rep[v* ] ← u ;
5  for each vertex u ∈ V in parallel do
       v ← rep[u];
       if v ≠ null and u ≠ rep[v] then
           rep[u] ← null;
   for each vertex u ∈ V in parallel do
       v ← rep[u];
       if v ≠ null and u < v then
           rep[u] ← u;
```

line 1 of Algorithm 17, we check if a vertex $v$ adjacent to $u$ is already matched. If we detect an already matched candidate, we do not consider it as a possible mate for $u$. Furthermore, at line 2, we again verify if $u$ and $v$ are already matched right before matching them. This check is necessary since either of them could have been matched by another thread after the current one starts considering them.

#### 5.1.2.2 Multithreaded agglomerative clustering

To adapt the sequential agglomerative clustering algorithm to multithreaded setting, we use the same lock-based approach integrated to Algorithm 16. The pseudocode of the parallel agglomerative clustering algorithm is given in Algorithm 18. The algorithm visits the vertices in parallel, and when a thread visits a vertex $u$, it tries to lock $u$. If $u$ is already locked the thread skips $u$ and visits the next vertex. If $u$ is not locked but is already a member of a cluster, the thread unlocks $u$. Since a cluster cannot be the source of a new cluster, this is necessary. On the other hand, if $u$ is a singleton vertex, the thread continues by computing the similarity values for each adjacent vertex and then traverses the adjacency list ensuremath $\mathsf{adj}_u$ along the same lines as the sequential algorithm. The main difference here is the locking request for $v^r$ (line 1) which is either set to $v$ if $v$ is singleton, or to the representative of the cluster that $v$ resides in. Before considering $v^r$ as a matching candidate, this lock is necessary. However, if $v^r$ is already the best candidate, it is not so (since the thread has already grabbed $v^r$). When the lock is granted, the thread checks if the adjacent vertex $v$, which was singleton before, has been assigned to a cluster by another

thread. If this is the case, the thread unlocks the representative and continues with the next adjacent vertex. Otherwise, it recomputes the total weight of $u$ and $v^r$ (line 3) since new vertices might have been inserted to $v^r$'s cluster by other threads. Since insertions can only increase $\mathsf{w}[v^r]$ and $\mathsf{conn}[v^r]$, we do not need to compare $\mathsf{conn}[v^r]$ with $conn^*$ again. On the other hand, since we cannot construct clusters with large weights, we need to check if $totW$ is still smaller than $maxW$ (line 4). When the best candidate $v^*$ is found, we put $u$ in the cluster $v^*$ represents and update the rep and w arrays accordingly. Unlike the matching based algorithms, a cluster is allowed to be a candidate more than once throughout the execution. Hence, at the end of the iteration (lines 5 and 6) we unlock all the vertices that are locked during this iteration.

#### 5.1.2.3 Implementation Details

To obtain lock functionality for the multithreaded clustering algorithms described in the previous section, we use the compare and exchange CPU instruction which exists in x86 and Itanium architectures. We first allocate a lock array of length $|\mathcal{V}|$ and initialize all entries to 0. For each call of the corresponding function in C, `__sync_bool_compare_and_swap`, the entry related with the lock request is compared with 0. In case of equality, it is set to 1, and the function returns **true**. On the other hand, if the entry is not 0 then it returns **false**. To unlock a vertex, we simply set the related entry in the lock array to 0.

**Algorithm 18:** Parallel agglomerative clustering

```
Data: H = (V, N), maxW, rep
for each vertex u ∈ V in parallel do
    if checkAndLock(u) then
        if rep[u] ≠ null then
            Unlock(u);
            continue;
        ensuremath adj_u ← {};
        for each net n ∈ nets[u] do
            for each vertex v ∈ pins[n] do
                if v ∉ ensuremath adj_u then
                    ensuremath adj_u ← ensuremath adj_u ∪ {v}
            conn[v] ← conn[v] + c[n]
        v* ← u;
        conn* ← 0;
        for each vertex v ∈ ensuremath adj_u do
            if u = v then
                continue;
            v^r ←rep[v] ;
            if v^r = null then
                v^r ← v;
            if v^r ≠ v then
                conn[v^r] ← conn[v^r] + conn[v];
                #replace v with v^r ;
                conn[v] ← 0;
                ensuremath adj_u ← ensuremath adj_u ∪ {v^r}  \ {v};
            totW ← w[u] + w[v^r];
            if conn[v^r] > conn*and totW < maxW then
1               if v^r = v*or checkAndLock(v^r) then
2                   if rep[v] ≠ v^r and rep[v] ≠ null then
                        Unlock(v^r);
                        continue;
3                   totW ← w[u] + w[v^r];
4                   if totW < maxW then
                        conn* ← conn[v^r];
                        v* ← v^r;
                        if u ≠ v* then
                            Unlock(v*);
                    else
                        Unlock(v^r);
        for each vertex v ∈ ensuremath adj_u do
            conn[v] ← 0;
        if u ≠ v* then
            rep[v* ] ← v* ;
            rep[u] ← v* ;
            w[v* ] ← w[v* ] + w[u];
5           Unlock(v*);
6       Unlock(u);
```

**Algorithm 19:** Parallel lock-based matching: modified

```
   for each vertex u ∈ V in parallel do
1      if rep[v] = null then
2          if checkAndLock(u) then
               ··· ;
               for ··· do
                   ··· ;
                   if ··· then
3                      if rep[v] = null then
4                          if checkAndLock(v) then
                               ··· ;
           ··· ;
           ··· ;
```

Although this function provides great support and flexibility for concurrency, our preliminary experiments show that it can also reduce the efficiency of a multithreaded algorithm. To alleviate this, we try reduce the number of calls on this function by adding an if statement before each lock request which helps us to see if the lock is really necessary. We observe significant improvements on the execution times due to these additional **if** statements. For example, the parallel lock-based matching algorithm described in the previous section should be implemented as in Algorithm 19 to make it much faster. We stress that the **if** statements at lines 1 and 3 do not change anything in the execution flow. That is if a vertex is not locked, it cannot be also matched since a matched vertex always stays locked. Hence everything that can pass

the lock requests (lines 2 and 4) can also pass the previous if statements. However, the opposite is not true.

We used the same hypergraph data structure with PaToH. We store the ids of the vertices of each net $n$, that is its pins, consecutively in an array $\mathsf{ids}$ of size $\sum_{n\in\mathcal{N}} |\mathsf{pins}[n]|$. We also keep another array $\mathsf{xids}$ of size $|\mathcal{N}|+1$, which stores the start index of each net's pins. Hence, in our implementation, the pins of a net $n$, denoted by $\mathsf{pins}[n]$ in the pseudocodes, are stored in $\mathsf{ids}[\mathsf{xids}[n]]$ through $\mathsf{ids}[\mathsf{xids}[n+1]-1]$.

With the data structures above, the computational complexity of the clustering algorithms in this work are in the order of $\sum_{n\in\mathcal{N}} |\mathsf{pins}[n]|^2$, since all non-loop lines in their pseudocodes have $\mathcal{O}(1)$ complexity. For example, in Algorithm 14, to remove matched vertices from $\mathsf{pins}[n]$ (line 3), we keep a pointer array $\mathsf{netend}$ of size $\mathcal{N}$ where $\mathsf{netend}[n]$ initially points to the last vertex in $\mathsf{pins}[n]$ for all $n \in \mathcal{N}$. Then, to execute $\mathsf{pins}[n] \leftarrow \mathsf{pins}[n] \setminus \{v\}$, we only decrease $\mathsf{netend}[n]$ and swap $v$ with the vertex in the new location. In this way, we also keep the list of vertices connected to each net unchanged since we only reorder them.

In the actual implementation of Algorithm 14, the set ensuremath $\mathsf{adj}_u$ corresponds to an array of maximum size $|\mathcal{V}|$, and an integer which keeps the number of adjacent vertices in the array. With this pair, the vertex addition (line 5) and reset (line 2) operations take constant time. Furthermore, to find if a vertex $v$ is a member of ensuremath $\mathsf{adj}_u$ (line 4), we use $\mathsf{conn}[v]$ since the edge costs are positive, and $\mathsf{conn}[v]$

$> 0$ if and only if $v \in$ ensuremath $\mathsf{adj}_u$. The implementation of these lines are the same for other algorithms.

### 5.1.3 Experimental Results

The algorithms are tested on a computer with 2.27GHz dual quad-core Intel Xeon CPUs with 2-way hyper-threading enabled, and 48GB main memory. All of the algorithms are implemented in C and OpenMP. The compiler is icc version 11.1 and -O3 optimization flag is used.

To generate our hypergraphs, we used real life matrices from the University of Florida (UFL) Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices). We randomly choose 70 large, square matrices from the library and create corresponding hypergraphs using the column-net hypergraph model [37]. An overall summary of the properties of these hypergraphs is given in Table 5.1. The complete list of matrices is at http://bmi.osu.edu/~kamer/multi_coarse_matrices.txt.

Table 5.1: Properties of the hypergraphs used in the experiments

| | min | max | average |
|---|---|---|---|
| # vertices | 256,000 | 9,845,725 | 1,089,073 |
| # pins | 786,396 | 57,156,537 | 6,175,717 |
| $\frac{\#pins}{\#vertices}$ | 1.91 | 39.53 | 6.61 |

#### 5.1.3.1 Individual performance of the algorithms

We first compare the performance of the multithreaded clustering algorithms with respect to the cardinality, quality and speedup as standalone clustering algorithms.

##### 5.1.3.1.1 Performance on cardinality and quality

For matching based clustering, a matching with the best quality can be found by first constructing the matrix $M = AA^T - diag(AA^T)$ where $A$ is the vertex-net incidence matrix. Then a maximum weighted matching in $\mathcal{G}(M)$, the associated weighted graph of $M$, is also the maximum quality matching.

We use Gabow's maximum weighted matching algorithm [72]—implemented by Rothberg and available as a part of The First DIMACS Implementation Challenge (available at `ftp://dimacs.rutgers.edu/pub/netflow`). This algorithm has a time complexity of $\mathcal{O}(|\mathcal{V}|^3)$ for a graph on $|\mathcal{V}|$ vertices and finds a maximum weighted matching in the graph, not necessarily among the maximum cardinality ones.

Due to the running time complexity of the maximum quality matching algorithm, it is impractical to obtain the relative performance of the clustering algorithms on our original dataset. We therefore use an additional data set containing considerably small matrices. The new dataset contains all 289 square matrices in the UFL sparse matrix collection with at least $3,000$ and at most $10,000$ rows. We construct the hypergraphs for each of these matrices and find the maximum quality matching on them.

Table 5.2: Relative performance of the sequential and parallel matching based clustering algorithms w.r.t. to the maximum quality matchings (# of threads = 8).

| | Quality | | | Cardinality | | |
|---|---|---|---|---|---|---|
| | min | max | gmean | min | max | gmean |
| Sequential | 0.24 | 1.00 | 0.81 | 0.85 | 1.68 | 1.02 |
| Lock-based | 0.32 | 1.00 | 0.83 | 0.74 | 1.65 | 1.02 |
| Resolution-based | 0.25 | 0.99 | 0.74 | 0.77 | 1.78 | 0.99 |

The relative performance of an algorithm is computed by dividing its cardinality and quality scores to those of Gabow's quality matching algorithm. Table 5.2 shows the minimum, the maximum, and the geometric mean of all 289 relative performance for each algorithm. As the table shows, the sequential algorithm and its parallel lock-based variant are only 17–19% far from the optimal in terms of quality and almost equal in terms of cardinality. Considering the difference in computational complexities, we can argue that their relative performance is reasonably good. The lock-based algorithm performs slightly better than the sequential one. This demonstrates that while reducing the execution time, the proposed lock-based parallelization does not hamper the performance of the sequential matching algorithm in terms of both cardinality and quality. For this experiment, the parallel algorithms were executed with 8 threads.

Figure 5.1 shows the performance profiles generated to analyze the results in more detail. A point $(x, y)$ in the profile graph means that with $y$ probability, the quality of the matching found by an algorithm is more than $max/x$ where $max$ is

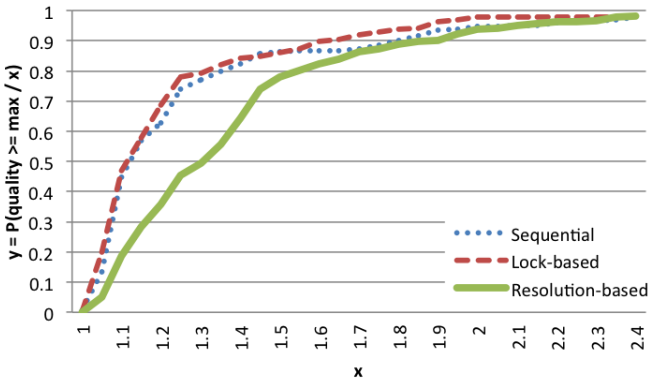



Figure 5.1: Performance profiles for sequential and multithreaded matching algorithms with respect to the maximum quality matchings. A point $(x, y)$ in the profile graph means that with $y$ probability, the quality of the matching found by an algorithm is larger than $max/x$ where $max$ is the maximum quality for that instance.

the maximum quality for that instance. The figure shows that for this data set, the resolution-based algorithm performs worse than the other algorithms. The lock-based algorithm obtains matchings which are at most 1.25 times less than $max$ with 74% probability. However, the resolution-based matching has this performance only for 45% probability. While obtaining matchings having at most 15% worse quality than the optimum, the probabilities are 56% and 28% for the lock- and resolution-based algorithms, respectively. Hence, the former performs two times better than the latter.

For the original data set with large hypergraphs, the relative performance of the multithreaded algorithms are given with respect to that of their sequential versions. Table 5.3 shows the statistics for this experiment.

Table 5.3 shows that the multithreaded versions of lock-based and agglomerative algorithms perform as good as their sequential versions in terms of cardinality and quality. Hence, once again, we can conclude that parallelization via locks does not diminish the performance of the multithreaded algorithms. On the other hand, the resolution-based algorithm is outperformed by other algorithms in terms of quality.

Table 5.4 shows the average numbers of matched vertices and conflicts for the proposed resolution-based algorithm with respect to the number of threads. To compute the averages, we execute the algorithm on each hypergraph ten times and report the geometric mean of these results. As expected, the number of the conflicts increases with the number of threads. However, when compared with the cardinality of the matching, the conflicts are at most 0.7% of the total match count for a single graph instance. This shows that the probability of a conflict is very low even with 8 threads.

Table 5.3: Relative performance of the parallel matching-based and agglomerative clustering w.r.t. to their sequential versions (# of threads = 8).

| | Quality | | | Cardinality | | |
|---|---|---|---|---|---|---|
| | min | max | gmean | min | max | gmean |
| Lock-based | 0.79 | 1.1 | 0.99 | 0.99 | 1.01 | 1.01 |
| Resolution-based | 0.63 | 1.1 | 0.91 | 0.78 | 1.01 | 0.97 |
| Agglomerative | 0.56 | 1.3 | 1.01 | 0.94 | 1.03 | 0.99 |

#### 5.1.3.1.2 Speedup

Figure 5.2 shows the speedups achieved by the multithreaded algorithms. On the average, the algorithms obtain decent speedups compared to their sequential versions up to 8 threads. In 8-thread experiments, the resolution-based algorithm has the highest speedup of 5.87, which is followed by the parallel agglomerative algorithm with a speedup of 5.82. The lock-based algorithm has the least speedup of 5.23 in this category. However, this is still decent especially when we consider the 5–7% overhead due to OpenMP and atomic operations.

To analyze the scalability of the algorithms from a closer point of view, we draw the speedup profiles in Figure 5.3. The resolution-based algorithm has better scalability in general. For example, with 4 threads, the probability that the resolution-based algorithm obtains a speedup of at least 3.2 is 83%. The same probabilities for the lock-based and parallel agglomerative algorithms are 54% and 65%, respectively. With 8 threads, the resolution-based version achieves at least 6.6 speedup for 33% the

Table 5.4: The average matching cardinality and the number of conflicts for the proposed resolution-based algorithm with respect to the number of threads.

| Thread # | $\#match$ | $\#conflict$ | $\frac{\#conflict}{\#match}$ |
|---|---|---|---|
| 1 | 290206.9 | 0.0 | 0.000000 |
| 2 | 290103.6 | 17.8 | 0.000061 |
| 4 | 290052.9 | 18.9 | 0.000065 |
| 8 | 289965.9 | 24.1 | 0.000083 |

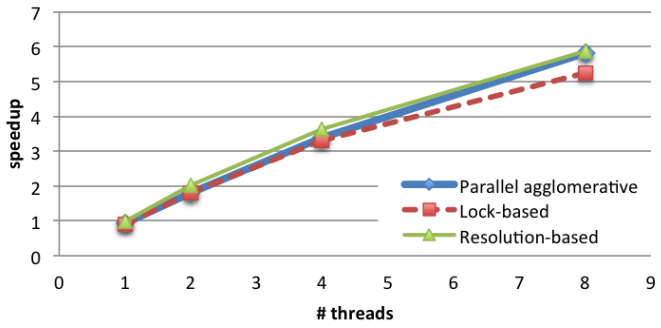
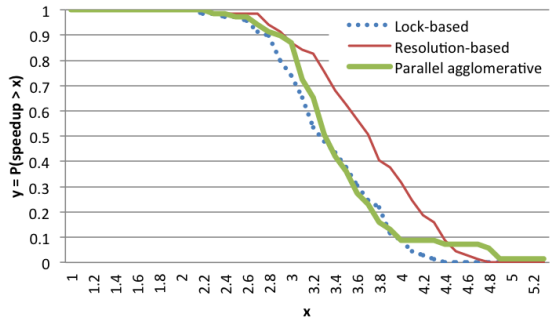



Figure 5.2: Speedups for matching and clustering: For the lock- and resolution-based algorithms, speedup is computed by using the execution times of sequential greedy matching algorithm. For the parallel agglomerative one, its sequential version is used.

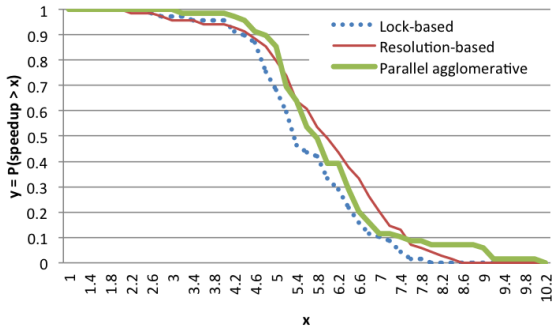
hypergraphs. However, the lock-based and parallel agglomerative algorithms achieve the same speedup only in for 16% and 20% of them. Hence, the resolution-based algorithm is the best among the ones proposed in this chapter in terms of scalability.

#### 5.1.3.2 Multi-level performance of the algorithms

As mentioned in the introduction, we integrated our algorithms into the coarsening phase of PaToH [38]. In this section, we first investigate how our algorithms scale for the clustering operations inside PaToH. The overall performance of a clustering algorithm in such a setting can be different from the standalone performance of the same algorithm, since in the multi-level framework, the hypergraphs are coarsened until the coarsest hypergraph is considerably small (for example, until the number vertices reduces below 100).




(a) $\#threads = 4$




(b) $\#threads = 8$

Figure 5.3: Speedup profiles for the multithreaded algorithms: A point $(x, y)$ in the profile graph means that with $y$ probability, the speedup obtained by the parallel algorithm will be more than $x$.

Figure 5.4 shows that the speedups on the multi-level clustering part are slightly worse than that of the standalone clustering. For example, the average speedups for the 8(4) threads case, are 5.25(3.22), 5.56(3.40), and 5.47(3.23) for the lock-based, resolution-based, and parallel agglomerative algorithms, respectively.

Since we only parallelize the clustering operations inside the partitioner, the speedups we obtain on the overall execution time cannot be equal to the number

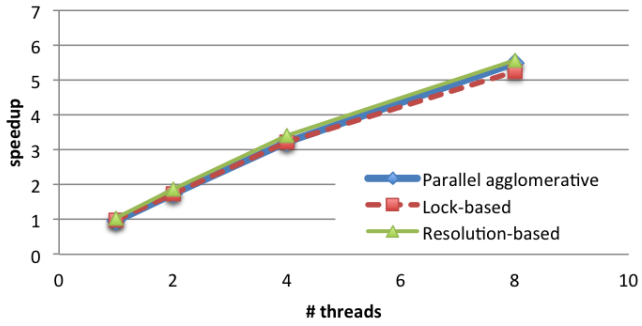
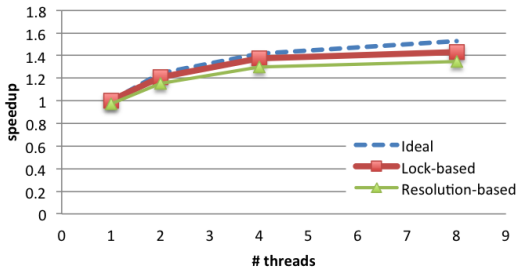
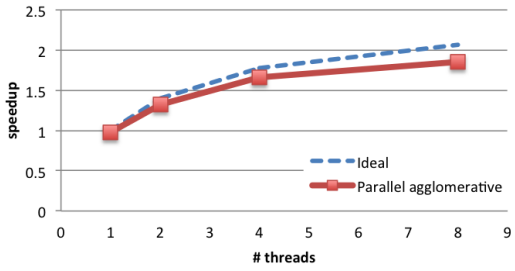



Figure 5.4: Speedups on the time spent by the clustering algorithms in the multi-level approach.




(a) Matching-based clustering




(b) Agglomerative clustering

Figure 5.5: Overall speedup on the total execution time of PaToH. The ideal speedup line is drawn by using Amdahl's law.

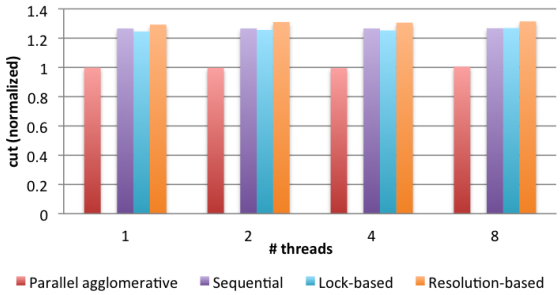
of threads, even in the ideal case. To find the ideal speedups, we use Amdahl's law [7]:

$$speedup_{ideal} = \frac{1}{(1-r) + \frac{r}{\#threads}} \tag{5.1.1}$$

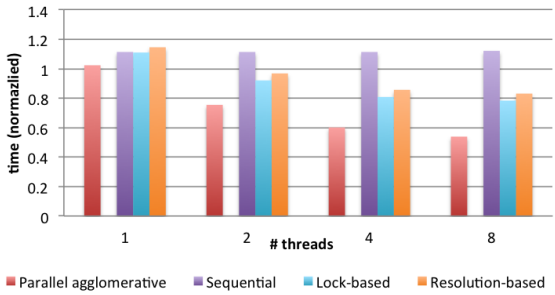
where $r$ is the ratio of the total clustering time to the total time of a sequential execution. To find the ideal speedup on the average, we compute (5.1.1) for each hypergraph and then take the geometric mean since we do the same for actual speedups.

Figure 5.5 shows the ideal and actual speedups of the multithreaded algorithms. Since the ideal speedup lines (in dashed style) are drawn by using different sequential algorithms for the matching-based and agglomerative clustering, we separated these two cases and draw two different charts. On the average, all the algorithms obtain speedups close to the ideal. Among the matching-based algorithms, the lock-based one is more efficient since its speedup is closer to the ideal. This is interesting since according to Figure 5.4, it has less speedup on the total clustering time. At first sight, this looks like an anomaly because this is the only part that has been parallelized. However, the since lock-based algorithm's quality is better (Table 5.3), there is probably less work remaining for the refinement heuristics in the uncoarsening phase. Hence, in total, one can achieve better speedup by using the lock-based algorithm rather than the resolution-based one.

We also obtain good speedups with the parallel agglomerative algorithm. For 2, 4, and 8 threads, the algorithm makes PaToH only 6%, 6% and 10% slower, respectively, than the best possible parallel execution time.




(a) Minimum cutsize found by PaToH




(b) Total execution time

Figure 5.6: The minimum cut and execution time of PaToH when equipped with the clustering algorithms in this chapter. The numbers are normalized with respect to that of agglomerative clustering algorithm.

Parallelization of both matching-based and agglomerative clustering algorithm reduces the total execution time significantly. As mentioned before, matching-based algorithms are faster than the agglomerative ones. However, Figure 5.6 shows that PaToH is 20–30% faster when an agglomerative algorithm is used in the coarsening phase rather than a matching based one. According to our experiments, the coarsening phase is indeed 25% slower with the agglomerative clustering algorithm. However, the total execution time is 13% less. The difference comes from the reduction on the

time of initial partitioning and uncoarsening/refinement phases. The initial partitioning takes less time because the coarsest hypergraph has fewer number of vertices with an agglomerative algorithm. In addition, the agglomerative clustering results in 25% less cutsize compared to the matching-based clustering. Hence, we can claim that it is more suitable for the cutsize definition given in (2.2.3).

When equipped with the multithreaded clustering algorithms, the cutsize of the partition found by PaToH is almost equal to the original cutsize obtained by using the sequential versions. For the agglomerative case, the cutsize changes only up to 1%. This is also true with the lock-based matching algorithm when compared with sequential greedy matching. For the resolution-based matching algorithm, there is at most 3% percent increase in the cutsize on average. On the other hand, as shown above the algorithms scale reasonably well.

### 5.1.4 Summary

Clustering algorithms are the most time consuming part of the current-state-of-the-art hypergraph partitioning tools that follow the multi-level framework. The matching-based and agglomerative clustering algorithms are investigated in this chapter. We have argued that the matching-based clustering algorithms can be perceived as a matching algorithm on an implicitly represented undirected, edge weighted graph, whereas there is no immediate equivalent algorithm for the agglomerative ones.

Two different multithreaded implementations of the matching-based clustering algorithms are proposed in this chapter. The first one uses atomic lock operations to

prevent inconsistent matching decisions made by two different threads. The second one lets the threads perform matchings as they would do in a sequential setting and then later on resolves the conflicts that would arise. A multithreaded agglomerative clustering algorithm has also been proposed in this chapter. This algorithm also uses locks to prevent conflicts.

Different sets of experiments have been presented on a large number of hypergraphs. The experiments have demonstrated that the multithreaded clustering algorithms perform almost as good as their sequential counter parts, sometimes even better in terms of clustering quality and cardinality. The experiments have also shown that our algorithms achieve decent speedups (the best was 5.87 with 8 threads).

The proposed algorithms are integrated to a well-known hypergraph partitioner PaToH. This integration makes PaToH 1.85 times faster where the ideal speedup is 2.07. In addition, it does not worsen the cutsizes obtained.

We observed that clusterings with better quality helps the partitioner to obtain better cuts. Fortunately, the multi-level framework may tolerate slower algorithms which generate better clusterings in terms of cardinality and quality. This is because of the fact that such clusterings will reduce the time required for the initial partitioning and uncoarsening phases. However, there is a limit with this tolerance. If the algorithm is too slow, one can execute the partitioner several times with a faster, parallelizable algorithm that generates clusterings with acceptable quality and achieve even better cutsizes.

## 5.2 Hypergraph Sparsification Methods for scalable Multi-level Hypergraph Partitioning

Using and analyzing large data for practical purposes has always been a fundamental problem. But today, the data one needs to cope with for a major scientific innovation or discovery is immense, distributed, and unstructured. Although recent advancements on computer hardware and storage technologies allow us to gather and store large-scale data, the data itself does not serve science and society. In addition to its size, utilizing this data and finding meaningful patterns inside it require complex algorithms and an immense amount of computing power. Hence, techniques to make the data smaller are always appreciated.

Hypergraphs emerged as a good alternative model for unstructured data for many applications such as parallelization of complex and irregular applications from various domains including parallel scientific computing [42], sparse matrix reordering [79], social network analysis [131], clustering and recommendation [130], and database design [103, 110, 128]. In many of these applications, once the problem is modeled as a hypergraph, the optimization problem in hand reduces to, sometimes with little variations, the hypergraph partitioning problem. Due to its wide area of applications, a considerable effort has been put into providing tool support for hypergraph partitioning (see hMeTiS [95], MLpart [35], Mondriaan [142], Parkway [134], PaToH [38], UMPa [45] and Zoltan [29]).

There are two main criteria to evaluate the performance of a tool: the quality of the partition with respect to a partitioning metric and the time required to partition the given hypergraph. It has been repeatedly shown that minimizing the partitioning metrics in the literature can significantly increase the performance for many computations. However, the relative importance of these criteria change with respect to the application. If the partitioning time is bigger than the gain in the computation time, it may be better to trade quality for partitioning cost. But one needs to be careful with this trade-off since worsening the quality can also worsen the performance. For example, in a recent study, Akbudak et al. studied different partitioning models to optimize the cache locality for the sparse-matrix vector multiplication kernel (SpMV) [2]. They reported that when the 1D hypergraph model in [149] is used, the partitioning overhead is amortized in 286 SpMV operations. For their 2D model, this number is 1110 but the SpMV performance is 13% better than the 1D case at the same time. Hence, the best model, or even the decision of using a partitioning depends on the subsequent computation. Clearly, if the partitioning process is made faster it will be much more useful for many applications.

In the past, powerful parallel machines were only accessible by a small set of researchers. Today, a High Performance Computing (HPC) system with thousands of processors/cores is now almost an ordinary commodity. However, when the number of processors increases to tens of thousands, which is the case today, the data and task distribution cost via partitioning also increases. Yet the communication between

processors and hence the quality of the partition become more important with the increasing size of the systems. Hence, reducing the partitioning cost, especially for a larger system, is a crucial task.

In this chapter, *hypergraph sparsification* to reduce the partitioning cost is investigated. At the high level, the sparsification problem is modeled as finding the sets of identical (or very similar) sets/nets/vertices in a hypergraph. Let $\mathcal{S}$ be such a set containing distinct elements with identical connectivity information. Hence, the same information is duplicated $|\mathcal{S}|$ times in the hypergraph. Duplicate information causes two main problems for a task: First, the same information is processed several times, and this redundancy usually worsens the efficiency of the task. Second, if the purpose of this task is optimization, the redundant (or almost redundant) information makes the search space larger and finding good (or optimal) solutions harder. The action we take to remove such information is using a single *representative* instead of all the elements in $\mathcal{S}$. The representative can be either an existing element of $\mathcal{S}$ or can be artificially created according to a criterion by using the properties of $\mathcal{S}$'s elements.

In particular, here cheap hypergraph sparsification heuristics: identical-net, identical-vertex, and similar-net removal are investigated. Although identical-net removal has been used in partitioning before [11, 38], to the best of our knowledge, there is no work which analyzes its effectiveness in detail. Even though, some implementations exist in widely-used partitioning tools such has PaToH and Zoltan, it is disabled in

PaToH, since on average, the existing code does not amortize itself in PaToH's recursive bisection framework, and we show that Zoltan's sort based approach can be improved. Hence, the algorithms designed and analyzed in this work can be used in other tools, including PaToH and Zoltan, for faster partitioning. Furthermore, there is no previous work which analyzes the effectiveness of the similarity-based sparsification techniques proposed in this work for hypergraph based data.

A set of experiments is carefully designed to analyze the effectiveness of the proposed sparsification techniques, their integration to the multi-level approach, and their efficient implementation. For the experiments, we use a multi-level $K$-way partitioning tool UMPa (pronounced as "Oompa") [45] and minimize the total communication volume which is the classical partitioning objective used in practice. Our experiments show that when the hypergraphs in the multi-level framework are sparsified, one can obtain partitions with similar quality in significantly less time.

The rest of the chapter is organized as follows: Section 5.2.1 gives some background information and the notation used in this chapter, and Section 5.2.2 describes the proposed sparsification heuristics. The experimental results are presented in Section 5.2.3. Section 5.2.4 concludes the chapter. Refer to Section 2.2.2 for the hyperhraph partitioning problem formulation and background information on the multilevel hypergraph partitioning.

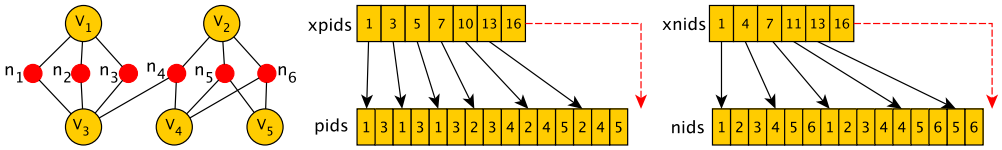



Figure 5.7: A simple hypergraph with 6 nets and 5 vertices with the data structures used in the implementation.

## 5.2.1 Background

### 5.2.1.1 Data Structures for Hypergraphs

For efficient access, hypergraphs are usually stored and utilized using two pairs of arrays which represent the hypergraph w.r.t. its nets (i.e., pins[]) and vertices (i.e., nets[]), respectively.

1. *xpids-pids* pair gives the net view. The sizes of *xpids* and *pids* are $|\mathcal{N}|+1$ and $\rho$, respectively. The array *pids* stores the pin ids, and $xpids[i]$ points to the start location in *pids* for each net $i$. The last value of *xpids* is equal to $\rho$. That is $pids[xpids[i], \ldots, xpids[i+1]-1]$ contains the pin ids for net $i$.

2. *xnids-nids* pair gives the vertex view. The sizes of *xnids* and *nids* are $|\mathcal{V}|+1$ and $\rho$, respectively. The array *nids* stores the net ids, and $xnids[i]$ points to the start location in *nids* for each vertex $i$. The last value of *xnids* is equal to $\rho$. That is $nids[xnids[i], \ldots, xnids[i+1]-1]$ contains the net ids for vertex $i$.

In multi-level partitioning context, these arrays are generated not only for the original hypergraph, but also for each coarse hypergraph too. Hence, in any part of the partitioner, accesses to pins[n] and nets[v] of a net $n$ and a vertex $v$ can be handled very efficiently. A toy hypergraph and the corresponding data structures are given in Figure 5.7.

### 5.2.2 Sparsification Techniques for Hypergraphs

In this section, we materialize this sparsification approach for the problem of scalable hypergraph partitioning. However, the heuristics and algorithms proposed here can be used for other problems and applications.

#### 5.2.2.1 Identical-Net Removal

Although we start with the description of the identical-net removal heuristic, the techniques described in this subsection are also valid for the other heuristics in this chapter. The identicality of the nets depends on the set of vertices they are connected to.

**Definition 1** *Two nets $n_i$ and $n_j$ are* identical *if* pins$[n_i]$ *and* pins$[n_j]$ *are the same.*

Given a hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, let $\{\mathcal{N}_1,\mathcal{N}_2,\ldots,\mathcal{N}_\kappa\}$ be the partition of the net set $\mathcal{N}$ such that two nets in $\mathcal{N}$ are in the same subset $\mathcal{N}_\ell$, $1 \leq \ell \leq \kappa$, if and only if they are identical. The identical-net removal (INR) process generates a smaller hypergraph $\mathcal{H}'=(\mathcal{V},\mathcal{N}')$ where $\mathcal{N}' = \{n'_1,n'_2,\ldots,n'_\kappa\}$ and each net $n'_\ell$ corresponds to $\mathcal{N}_\ell \subseteq \mathcal{N}$ for $1 \leq \ell \leq \kappa$. The net $n'_\ell$ is also called *representative* net of the nets in $\mathcal{N}_\ell$,

and its pin set is defined as

$$\mathsf{pins}'[n'_\ell] \leftarrow \mathsf{pins}[n] \text{ s.t. } n \in \mathcal{N}_\ell, \tag{5.2.1}$$

and the net set of each vertex $v \in \mathcal{H}'$ contains the representatives having $v$ in their pin set.

In hypergraph partitioning context, if two nets $n_i$ and $n_j$ are identical they are either both internal or both external with respect to a partition $\Pi$. Furthermore, $\lambda_{n_i} = \lambda_{n_j}$. Since each net's contribution to the connectivity-1 metric given in (2.2.3) is proportional to its cost, the cost of a representative $n'_\ell \in \mathcal{N}'$ is computed as

$$\mathsf{c}'[n'_\ell] \leftarrow \sum_{n \in \mathcal{N}_\ell} \mathsf{c}[n]. \tag{5.2.2}$$

Let $\Pi$ be a $K$-way partition of the vertex set $\mathcal{V}$. The contribution of net $n'_\ell \in \mathcal{N}'$ to the cutsize $conn_{\mathcal{H}'}(\Pi)$ is equal to the sum of contributions of the nets in $\mathcal{N}_\ell$ to $conn_{\mathcal{H}}(\Pi)$ since the connectivity $\lambda_{n'_\ell}$ of the representative is equal to the connectivity of identical nets. That is, the cutsize metrics, $conn_{\mathcal{H}}(\Pi) = conn_{\mathcal{H}'}(\Pi)$, are equal. Hence, an optimal partition $\Pi$ for the sparsified hypergraph $\mathcal{H}'$ is also optimal for the original $\mathcal{H}$.

Since INR is not expected to change the quality of the final partition, reducing the overhead of the net-removal process is of utmost importance. The naive algorithm for detecting and removing identical nets requires $\mathcal{O}(V^2)$ pairwise comparisons of the pin sets. The number of these comparisons can be greatly reduced by using a simple

checksum function,

$$\textsc{Cs1}(n) = \sum_{i \in \mathsf{pins}[n]} i, \tag{5.2.3}$$

as shown by Algorithm 20, InrSrt. According to Definition 1, if two nets are identical their checksums must be equal. Hence, the inequality of the checksums can be used as a *witness* for two nets being non-identical. On the other hand, such a witness is not sufficient to identify *false-positive* net pairs who have the same checksum value but are not identical.

InrSrt first computes the checksum values of all $n \in \mathcal{N}$. Second, it sorts the nets with respect to these values. And third, it examines only the nets with the same csum value at a time to see if they are identical. In InrSrt, during the examination for a net $n \in \mathcal{N}$, rep[n] is set to an existing representative id (line 4) which is added to $\mathcal{N}'$ after a previous examination. If such a representative does not exist, at line 5, rep[n] will be equal to a new id, and a new representative will be added to $\mathcal{H}'$. The pin set and cost of the representative are assigned according to (5.2.1) and (5.2.2), respectively. This algorithm has been used in the literature with similar motivations such as detecting identical vertices in graphs [9] and identifying supervariables in a nested-dissection based matrix reordering scheme [86]. It has also been used to remove identical nets [11].

We say that a *collision* exists when $\mathsf{csum}[n_i] = \mathsf{csum}[n_j]$ for two non-identical nets $n_i$ and $n_j$. If there are no collisions, i.e., no false positives, line 3 of InrSrt will be executed at most once for each net $n \in \mathcal{N}$. Since each identicality check for a net

**Algorithm 20:** InrSrt

**Data**: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), \mathsf{c}, \mathsf{pins}$
**Output**: $\mathcal{H}' = (\mathcal{V}, \mathcal{N}'), \mathsf{c}', \mathsf{pins}'$
$\mathcal{N}' \leftarrow \emptyset$;
**for each** $n \in \mathcal{N}$ **do**
1  $\mathsf{csum}[\mathrm{n}] \leftarrow \mathrm{Cs1}(n)$;
$\sigma \leftarrow$ the permutation of the nets in the increasing order of $\mathsf{csum}$ values;
$prev \leftarrow -1$;
$r \leftarrow 1$;
**for** $i = 1$ to $|\mathcal{N}|$ **do**
  $n \leftarrow \sigma[i]$;
  **if** $prev \neq \mathsf{csum}[\mathrm{n}]$ **then**
    $\mathcal{R} \leftarrow \{n'_r\}$;
    $r \leftarrow r + 1$;
    $prev \leftarrow \mathsf{csum}[\mathrm{n}]$;
  **else**
    $\mathsf{rep}[\mathrm{n}] \leftarrow r$;
2    **for each** $n'_\ell \in \mathcal{R}$ **do**
3      **if** $\mathsf{pins}[n'_\ell] = \mathsf{pins}[n]$ **then**
4        $\mathsf{rep}[\mathrm{n}] \leftarrow \ell$;
         $\mathsf{c}'[n'_r] \leftarrow \mathsf{c}'[n'_r] + \mathsf{c}[n]$;
         **break**;
    $c \leftarrow \mathsf{rep}[\mathrm{n}]$ ;
5    **if** $c = r$ **then**
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{n'_r\}$;
      $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{n'_r\}$;
      $\mathsf{c}'[n'_r] \leftarrow \mathsf{c}[n]$;
      $\mathsf{pins}'[n'_r] \leftarrow \mathsf{pins}[n]$;
      $r \leftarrow r + 1$;
$\mathcal{H}' \leftarrow (V, \mathcal{N}')$;

$n$ costs $\mathcal{O}(|\mathsf{pins}[n]|)$, the total cost due to 3 is $\mathcal{O}(\rho)$. Hence, with a good checksum function, which only creates a negligible amount of conflicts, the total complexity is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}| + \rho)$. We evaluate the effectiveness of a checksum function w.r.t. two criteria:

- *False-positive cost*: The number of pairwise comparisons (line 3) during the course of InrSrt for two non-identical nets.
- *Checksum occupancy*: The average number of distinct representatives having the same checksum value. When no false-positives exist, the occupancy is one. Otherwise, it is greater.

When all $\kappa$ representatives have the same checksum value, the occupancy is $\kappa$, and the false-positive cost is $\mathcal{O}(\kappa|\mathcal{N}|)$. Hence, the total worst case complexity is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}| + \kappa\rho)$. Furthermore, if $\kappa$ is also $\mathcal{O}(|\mathcal{N}|)$, i.e., all nets are distinct and have the same checksum value, the total cost of line 3 will be huge. Fortunately, hypergraphs from real life are random enough to avoid such pathological cases even with a simple checksum function such as Cs1. Still, it is better in practice to reduce the number of collisions to make the identical-net removal faster. Yet again, using a simple function is also a good practice since the cost of the checksum computation will be less which may be important for some applications.

As described above, InrSrt sorts the nets with respect to their checksums. This operation can be considered as using the witnesses to eliminate a huge amount of

**Algorithm 21:** InrMem

```
   Data: H = (V, N), c, pins
   Output: H' = (V, N'), c', pins'
   N' ← ∅;
   for i from 1 to q do
       first[i] = −1;
   for i from 1 to |N| do
       next[i] = −1;
1      csum[i] ← Cs1(n_i) mod q;
   r ← 1;
   for i from 1 to |N| do
       c ← first[csum];
       if c = −1 then
           first[csum[i]] = i;
           c ← i;
       while c ≠ i do
           ℓ ← rep[n_c] ;
2          if pins'[n'_ℓ] = pins[n_i] then
               c'[n'_ℓ] ← c'[n'_ℓ] + c[n_i];
               break;
           else
               if next[c] = −1 then
3                  next[c] = i;
4          c ← next[c];
       if c = i then
5          N' ← N' ∪ {n'_r};
           c'[n'_r] ← c[n_i];
6          pins'[n'_r] ← pins[n_i];
           rep[n_i] ← r ;
           r ← r + 1;
   H' ← (V, N');
```

pairwise comparisons. However, when a good checksum function with a small occupancy and false-positive cost is used, the sorting operation can be the dominant factor considering the complexity of the rest of the algorithm. In this work, we propose to use another, *hash-based*, approach given in Algorithm 21, InrMem that trades memory for performance. The data structure we use contains two arrays, $first$ and $next$, of sizes $q$ and $|\mathcal{N}|$, respectively, to store the representative information. Here, $q$ is the first prime number greater than $|\mathcal{N}|$. InrMem first initializes the entries in $first$ to $-1$ and starts to traverse the nets from $n_1$ to $n_{|\mathcal{N}|}$. For each net $n_i$, by using the $first$ array, it checks if there exist a net $n_c$ s.t. $c < i$ and $\text{Cs1}(n_c) \equiv \text{Cs1}(n_i) \pmod{q}$. If such a net exists the corresponding entry in $first$ will be $c$, and $-1$, otherwise. In the former case, starting from $n_c$, the algorithm initiates a set of consecutive checks (line 2) to eliminate false positives for $n_i$ by following the links in the $next$ array (line 4). Similar to InrSrt, if $n_i$ is identical to an existing representative $n'_\ell$ its contribution is added to $\mathsf{c}'[n'_\ell]$. If no such representative exists, a new one is created (line 5) and $next$ is updated accordingly (line 3). With a good checksum function, the total complexity of InrMem is $\mathcal{O}(\rho)$, since the check to distinguish false positives is executed only once for most of the nets.

Since the net information is stored in sets, the order of the set elements should not affect the csum value, i.e., the checksum function should be *order independent*. Having collision reduction in mind, in this work, we investigate two simple order-independent

| Partition | Checksum |
|---|---|
| $\mathcal{N}_1 = \{n_1, n_2, n_3\}$ | $\textsc{Cs1}(n'_1) \mod q = 4$ |
| $\mathcal{N}_2 = \{n_4\}$ | $\textsc{Cs1}(n'_2) \mod q = 2$ |
| $\mathcal{N}_3 = \{n_5, n_6\}$ | $\textsc{Cs1}(n'_3) \mod q = 4$ |

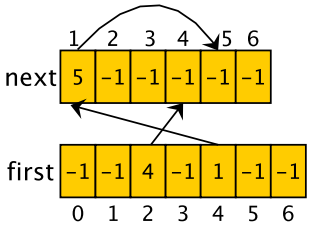



Figure 5.8: Identical-net removal via InrMem on a toy hypergraph in Figure 5.7 with 6 nets (i.e., $q = 7$). On the left, the 3 subsets of identical nets and the corresponding representatives' Cs1 mod $q$ values are given. On the right, the *first-next* data structure at the end of the process is shown.

checksum functions:

$$\textsc{Cs2}(n) = \sum_{i \in \mathsf{pins}[n]} i^2, \tag{5.2.4}$$

$$\textsc{Cs3}(n) = \sum_{i \in \mathsf{pins}[n]} i^3. \tag{5.2.5}$$

These functions can be integrated to the process by modifying only lines 1 of InrSrt and InrMem, respectively. Another checksum function we used is *MurmurHash*, a fast but order dependent hash function with good collision properties [8]. Since the function is order dependent, we need to sort the pin set of each net in the hypergraph as a prerequisite for InrSrt and InrMem.

#### 5.2.2.2 Identical-Vertex Removal

The connectivity information of hypergraphs can be viewed from two different perspectives: nets and vertices. Here we investigate a hypergraph sparsification technique from the vertices' point of view. The identicality definition of the vertices is similar to that of the nets:

**Definition 2** *Two vertices $v_i$ and $v_j$ are* identical *if* $\mathsf{nets}[v_i]$ *and* $\mathsf{nets}[v_j]$ *are the same.*

Given a hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, let $\{\mathcal{V}_1,\mathcal{V}_2,\ldots,\mathcal{V}_\kappa\}$ be the partition of the vertex set $\mathcal{V}$ such that two vertices in $\mathcal{V}$ are in the same subset $\mathcal{V}_\ell$, $1\leq\ell\leq\kappa$, if and only if they are identical. The identical-vertex removal process generates a smaller hypergraph $\mathcal{H}'=(\mathcal{V}',\mathcal{N})$ where $\mathcal{V}'=\{v'_1,v'_2,\ldots,v'_\kappa\}$ and each vertex $v'_\ell\in\mathcal{V}'$ corresponds to a $\mathcal{V}_\ell$ for $1\leq\ell\leq\kappa$. The vertex $v'_\ell$ is also called a *representative* vertex for vertices in $\mathcal{V}_\ell$. The net set and weight of a representative $v'_\ell\in\mathcal{V}'$ are defined as

$$\mathsf{nets}'[v'_\ell]\leftarrow\mathsf{nets}[v]\text{ s.t. }v\in\mathcal{V}_\ell,$$

$$\mathsf{w}'[v'_\ell]\leftarrow\sum_{v\in\mathcal{V}_\ell}\mathsf{w}[v].$$

After vertex removal, the pin set of each net $n\in\mathcal{H}'$ contains the representative vertices having $n$ in their net set.

Unlike the case for identical-nets, the optimality of the partitioning is not preserved for identical-vertex removal: Let $\Pi$ and $\Pi'$ be two optimal partitions for $\mathcal{H}$ and $\mathcal{H}'$, respectively. Then we have $conn_{\mathcal{H}}(\Pi)\leq conn_{\mathcal{H}'}(\Pi')$. The equality does not always hold due to the load balancing constraint since the original hypergraph $\mathcal{H}$ is

more relaxed in terms of vertex moves, and some partitioning configurations for $\mathcal{H}$ may not be feasible for $\mathcal{H}'$ due to its larger vertex weights. However, removing identical vertices also reduces the search space for the refinement heuristics employed in the uncoarsening phase. As the experiments will show, this may lead the partitioner to a slightly better final partition.

We use the same approach in InrMem to detect the identical vertices in $\mathcal{H}$ and to create $\mathcal{H}'$. Hence, the implementations of the identical-net and -vertex removal heuristics are very similar except that the former one uses *xpids-pids* whereas the latter uses *xnids-nids* for efficient checksum computations. Although identical-vertex removal can be used for any hypergraph, for partitioning, we only use it on the original, finer one, since the coarser hypergraphs are generated by a vertex matching process which is expected to match the identical vertices with high probability.

#### 5.2.2.3 Similar-Net Removal

The sparsification heuristics in the previous sections are solely based on the concept of identicality and they only aim to remove the redundancy from a large hypergraph. That is, they are only effective when there exists identical nets and/or vertices in the hypergraph. Here, we will describe the *similar-net removal* (SNR) heuristic, which can be used even when there is no redundancy. The heuristic can be considered as a lossy compression technique since it discards some information while sparsifying the hypergraph. Although discarding information usually worsen the quality of the final analysis/partition, the partitioning process will be faster since the hypergraph

will be smaller. Such a quality/time tradeoff will be very useful in practice, especially when the performance of the application is not very sensitive against small changes in partitioning quality. The effectiveness of such a tradeoff depends on the discovery of a large amount of information which can be more or less compensated by a relatively smaller representative. Considering each net is a set of pins, to attack the information discovery problem, we use the well-known Jaccard similarity metric [90]:

**Definition 3** *The* similarity *between two nets $n_i$ and $n_j$ is defined as*

$$J(n_i, n_j) = \frac{|\mathsf{pins}[n_i] \cap \mathsf{pins}[n_j]|}{|\mathsf{pins}[n_i] \cup \mathsf{pins}[n_j]|}.$$

Since the number of nets is large, it is infeasible to compute the similarity for each net pair. Instead, in this work, we propose computing a footprint of each net by using the minimum hash values, which is an efficient way to approximate $J(n_i, n_j)$ [31]. Let $\sigma$ be a random permutation of the integers from 1 to $|\mathcal{V}|$, and $min_\sigma(n)$ is the first vertex id of a net $n \in \mathcal{N}$ under the permutation $\sigma$. Then,

$$\Pr[min_\sigma(n_i) = min_\sigma(n_j)] = J(n_i, n_j).$$

In other words, if $x$ is a random variable s.t.

$$|x| = \begin{cases} 1, & \text{if } min_\sigma(n_i) = min_\sigma(n_j), \\ 0, & \text{otherwise.} \end{cases}$$

then $x$ is an unbiased estimator of $J(n_i, n_j)$. However, its variance is too high. In practice, one can use multiple independent permutations and get the average of the $x$ values to reduce the variance [31, 125]. However, this alone is not sufficient for

efficient hypergraph sparsification since it is still based on pairwise similarity. In this work, to obtain a more efficient solution, we use $t$ permutations $\sigma_1$ to $\sigma_t$ and first generate a *minwise footprint* of each net. The similarity definition is then modified as follows:

**Definition 4** *Two nets $n_i$ and $n_j$ are similar if their minwise footprints are the same, where the footprint of a net $n \in \mathcal{N}$ is defined as*

$$mf(n) = (min_{\sigma_1}(n), \ldots, min_{\sigma_t}(n)) .$$

Following the definition, given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, the similar-net heuristic constructs a partition $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_\kappa\}$ of the net set $\mathcal{N}$ where two nets $n_i$ and $n_j$ are in the same subset $\mathcal{N}_\ell$ if and only if their footprints are identical. It then generates a smaller hypergraph $\mathcal{H}' = (\mathcal{V}, \mathcal{N}')$ where $\mathcal{N}' = \{n'_1, n'_2, \ldots, n'_\kappa\}$ and each net $n'_\ell$ corresponds to $\mathcal{N}_\ell \subseteq \mathcal{N}$ for $1 \leq \ell \leq \kappa$. The net $n'_\ell$ is also called a *representative* net for nets in $\mathcal{N}_\ell$ and its cost is defined as

$$\mathsf{c}'[n'_\ell] \leftarrow \sum_{n \in \mathcal{N}_\ell} \mathsf{c}[n].$$

Since the connectivity of the nets in a subset $\mathcal{N}_\ell$ can be different, the representation of the connectivity information removed from $\mathcal{H}$ will be lossy, and the amount of discarded information depends on the assignment of $\mathsf{pins}'[n'_\ell]$. We investigated three options:

- *Large* (LRG): $\mathsf{pins}'[n'_\ell]$ is set to the pin set of the net with the largest pin set in $\mathcal{N}_\ell$.

- *Important* (IMP): $\mathsf{pins}'[n'_\ell]$ is set to the pin set of the net $n$ which maximizes
$$\sum_{v \in \mathsf{pins}[n]} \left( \sum_{n_i \in \mathsf{pins}[v] \cap N_\ell} \mathsf{c}[n_i] \right).$$
This metric prioritizes the pins which are connected to heavy nets with large $\mathsf{c}[.]$ values.

- *Union* (UNI): $\mathsf{pins}'[n'_\ell]$ is set to $\bigcup_{n \in \mathcal{N}_\ell} \mathsf{pins}[n]$.

After computing the footprints, we use the same approach in InrMem to detect similar nets in $\mathcal{H}$ and to create $\mathcal{H}'$. We changed line 2 since we are now checking footprints instead of pin sets. Among other minor modifications, we also changed line 6 w.r.t. the option we use while generating the pin set of the representatives.

#### 5.2.2.4 Parallelization of Sparsification Techniques

Although they are much cheaper than partitioning, the efficiency of the proposed heuristics may need to be improved by implementing them in parallel for other applications using hypergraphs. If this is the case, the first task, computation of the checksum values is pleasingly parallel since each thread can process a vertex/net without any concurrency problem. As the experiments show, the sorting phase of InrSrt dominates its execution time. Hence, it should be the first target for parallelization. After that, the false positive detection can be handled by multiple threads where each thread analyzes a different set of nets/vertices sharing the same hash value.

The proposed approach given in InrMem is also amenable for parallelization. Similar to InrSrt, the checksum computations can be naively parallelized. After that one can do only a single pass on the nets to build a variant of $first/next$ data structure by simply storing the id of every net in it without doing an equality check. Figure 5.9 shows this variant after adding all the elements in the toy hypergraph of Figure 5.9 when Cs1 is used. Then starting from the non-negative entries in the $first$

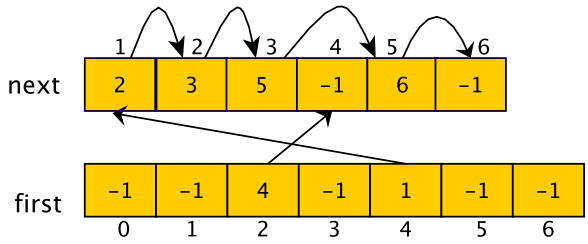



Figure 5.9: The $first/next$ structure when all the elements are added.

array, each thread can check the false positives throughout the path by following the $next$ pointers till a $null$ pointer $(-1)$ is reached. Starting from the one in Figure 5.9, after the false-positive checks, the same structure in Figure 5.8 will be obtained.

For some applications with huge and dynamic data, the heuristics may need to be executed several times and their efficiency can be of interest. Hence, we believe that the algorithmic details are important. However, in this work, we do not experiment with parallel implementations of the heuristics, since in the partitioning context, they are already cheap and their effectiveness in practice is much more important.

### 5.2.3 Experimental Results

For the experiments, we used a machine with a 2.27GHz dual quad-core Intel Xeon (Bloomfield) CPU and 48GB main memory. All the codes is implemented in C++ and compiled with g++ version 4.5.2. To create our test instances, we used a set of 28 matrices from the dataset of $10th$ DIMACS implementation challenge on graph partitioning and graph clustering [66]. We have used the row-net hypergraph representation of the matrices [37]. For all the matrices in our set, the number of vertices (and nets) is between $5 \times 10^5$ and $5 \times 10^6$, and the number of pins is roughly between $3 \times 10^6$ and $10 \times 10^7$. The properties of the hypergraphs we use are given in Table 5.5.

In our first experiment, we compare the performance of the algorithms and checksum functions described in Section 5.2.2. Figure 5.10 shows their execution times, false positive costs, and occupancies normalized w.r.t. those of InrSrt, when equipped with Cs1. As expected, the checksum functions are performing slightly better in terms of occupancy when used with InrSrt since the size of hash range is not limited, i.e., the same with the range of a 32-bit `int`. For InrMem, this number is $q$, the first prime number after $|\mathcal{N}|$ which is much smaller than the range of an `int` in practice. We observed that except Cs1, all other functions have an occupancy value close to one which is the optimal occupancy. Hence, an $\mathcal{O}(|\mathcal{N}|)$ hash range can be considered as good for practical performance. Yet, it is not enough and a good checksum function is necessary. As the figure shows, even a small increase on the occupancy leads

Table 5.5: Hypergraphs used in the Experiments

| Hypergraph | Class | #Vertices/ #Nets | #Identical Vertices | #Pins |
|---|---|---|---|---|
| *coPapersDBLP* | Citation | 540,486 | 285,113 | 30,491,458 |
| *eu-2005* | Cluster. | 862,664 | 29,979 | 32,276,936 |
| *in-2004* | Cluster. | 1,382,908 | 122,984 | 27,182,946 |
| *delaunay_n19* | Delan. | 524,288 | 0 | 3,145,646 |
| *delaunay_n20* | Delan. | 1,048,576 | 0 | 6,291,372 |
| *delaunay_n21* | Delan. | 2,097,152 | 0 | 12,582,816 |
| *hugetrace-00000* | Frames | 4,588,484 | 0 | 13,758,266 |
| *packing-500x100x100* | Numer. | 2,145,852 | 9 | 34,976,486 |
| *venturiLevel3* | Numer. | 4,026,819 | 0 | 16,108,474 |
| *channel-500x100x100* | Numer. | 4,802,000 | 0 | 85,362,744 |
| *rgg_n_2_19_s0* | Random | 524,288 | 24,433 | 6,539,532 |
| *rgg_n_2_20_s0* | Random | 1,048,576 | 46,228 | 13,783,240 |
| *rgg_n_2_21_s0* | Random | 2,097,152 | 88,681 | 28,975,990 |
| *rgg_n_2_22_s0* | Random | 4,194,304 | 169,311 | 60,718,396 |
| *ca2010* | Redistr. | 710,145 | 68,325 | 3,489,366 |
| *tx2010* | Redistr. | 914,231 | 116,811 | 4,456,272 |
| *af_shell9* | Sparse | 504,855 | 403,884 | 17,084,020 |
| *audikw1* | Sparse | 943,695 | 629,360 | 76,708,152 |
| *ldoor* | Sparse | 952,203 | 814,652 | 45,570,272 |
| *ecology2* | Sparse | 999,999 | 0 | 3,995,992 |
| *ecology1* | Sparse | 1,000,000 | 0 | 3,996,000 |
| *thermal2* | Sparse | 1,227,087 | 0 | 7,352,268 |
| *af_shell10* | Sparse | 1,508,065 | 1,206,452 | 51,164,260 |
| *G3_circuit* | Sparse | 1,585,478 | 0 | 6,075,348 |
| *kkt_power* | Sparse | 2,063,494 | 195,078 | 12,964,640 |
| *nlpkkt120* | Sparse | 3,542,400 | 0 | 93,303,392 |
| *belgium.osm* | Street | 1,441,295 | 103 | 3,099,940 |
| *netherlands.osm* | Street | 2,2166,88 | 61 | 4,882,476 |

to a high false positive cost. This is somehow expected because when there is more than one representative sharing the same checksum value, all the nets corresponding

to a representative need to be compared with other representatives. And when the cardinalities of identical/similar-net sets are large, there can be a large false-positive overhead. The experiments show that this overhead is usually much smaller than the overhead of the sorting phase of Inr-Sort. Even for InrMem equipped with Cs1, which has the highest false positive cost, the execution time is much smaller than all the InrSrt variants. But InrMem equipped with Cs2 is the best identical-net removal variant according to our experiments, since the checksum function Cs2 is as good as Cs3 and MurmurHash, and at the same time, computationally cheaper. Hence, we will continue to use it with Inr-Sort for all our removal experiments in the rest of this section.

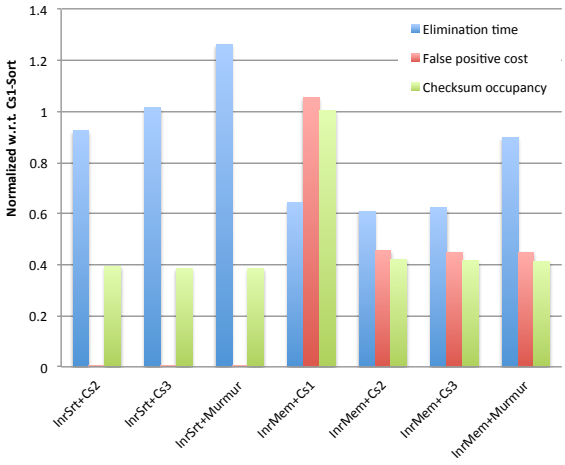



Figure 5.10: Performance of InrSrt and InrMem with various checksum functions. The results are normalized with that of InrSrt equipped with Cs1.

Table 5.6: Average partitioning times (in seconds) for base UMPa and its variants with identical-net (INR) and -vertex removal (IVR) heuristics. The last column shows the speedup on the partitioning time when compared with the base

| $K$ | Base UMPa | INR | INR+IVR | Speedup |
|---|---|---|---|---|
| 2 | 7.08 | 6.87 | 5.98 | 1.18 |
| 8 | 8.20 | 7.28 | 6.43 | 1.27 |
| 32 | 12.21 | 8.88 | 7.95 | 1.53 |
| 128 | 29.04 | 13.73 | 12.73 | 2.28 |
| 512 | 143.57 | 44.13 | 44.19 | 3.25 |
| 1,024 | 382.89 | 119.47 | 115.98 | 3.30 |

Table 5.6 shows the execution times of base UMPa and its variants equipped with identical-net and -vertex removal heuristics. Compared with the base, we obtain between 1.18–3.30 speedups while partitioning our hypergraphs to a different number of parts. The speedup values are increasing with $K$ and this makes the heuristics more promising and beneficial since the overhead of the partitioning problem is usually an issue for large $K$ values.

We observe that most of the improvement is obtained after removing identical nets. As Table 5.5 shows, not all of our hypergraphs contain identical vertices. But some have a lot: for the graph *ldoor*, 85% of the vertices have some number of identical copies in the graph. When this redundancy is removed, we obtain an additional speedup of more than 2, for $K = 512$, compared to UMPa equipped with INR. On the other hand, 14/28 of the matrices in the test set have less than 103 identical vertices hence, the overhead of executing a vertex removal heuristic will only be a

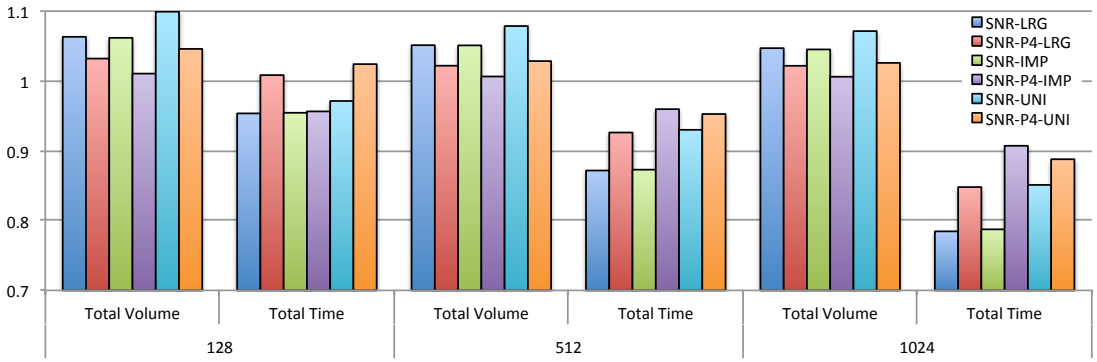
burden. Fortunately, the heuristic is efficient and on average it does not increase the overall time. When $K$ is small, it even helps to reduce the partitioning time.

Table 5.7: Average partitioning quality (cutsize) for base UMPa and its variants with identical-net (INR) and -vertex removal (IVR) heuristics. The last column shows the improvement on the quality metric when compared to the base

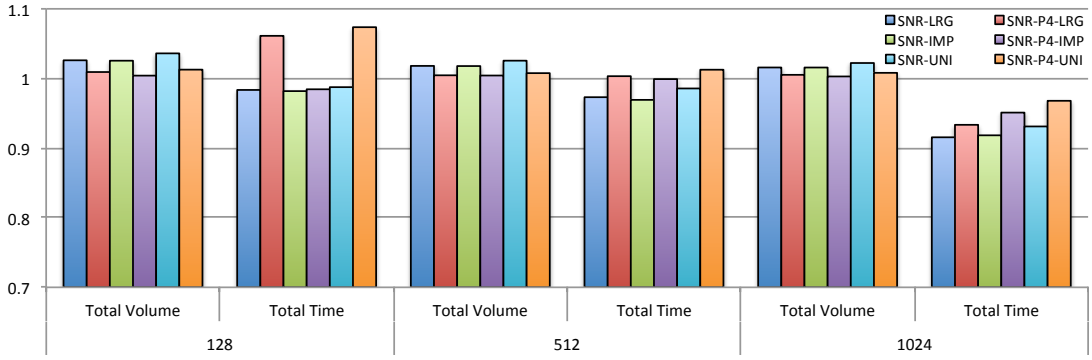
| $K$ | Base UMPa | INR | INR+IVR | Impr. |
|---|---|---|---|---|
| 2 | 2,826.78 | 2,762.04 | 2,760.49 | 2.4% |
| 8 | 15,183.28 | 15,026.15 | 14,915.84 | 1.8% |
| 32 | 41,656.66 | 41,833.82 | 41,541.03 | 0.3% |
| 128 | 102,209.71 | 101,865.58 | 101,690.31 | 0.5% |
| 512 | 223,187.20 | 223,126.15 | 222,582.41 | 0.3% |
| 1,024 | 321,706.01 | 321,728.75 | 319,865.46 | 0.6% |

As Table 5.7 shows, the quality of the final partition does not reduce when the proposed INR and IVR heuristics are used. On the contrary, as described in Section 5.2.2.2, removing identical vertices reduces the search space and hence, helps the refinement heuristics while minimizing the partitioning objectives. As the experiments show, when both heuristics are used, the partitioning quality is increased within a small margin, 0.3%–2.4%, on average.

To measure the performance of the similar-net removal heuristic, we compared the performance of INR+IVR with and without SNR. Figure 5.11 shows the average partitioning times and qualities of the version with SNR normalized w.r.t. INR+IVR for $K = \{128, 512, 1024\}$. To analyze various tradeoff configurations, we tried 4 different variants of SNR. In Figures 5.11.(a) and Figures 5.11.(b), we used $t = 4$ and




(a) $t = 4$, i.e., 4 permutations are used to generate minwise footprints.




(b) $t = 8$, i.e., 8 permutations are used to generate minwise footprints.

Figure 5.11: Partitioning times and qualities when the similar-net removal heuristic is used in additino to UMPa equipped with the identical-net and -vertex removal heuristics (INR+IVR). The results are normalized w.r.t. INR+IVR. In both figures, SNR-X is the proposed similar-net removal heuristic with the representative selection option X. The SNR-P4-X variant processes only the nets with 4 or more pins.

8 permutations to generate the footprints. Since the footprints are larger in the latter, the heuristic is more restricted, and a smaller number of similar nets are removed. For each $t$ value, in addition to SNR, we used another variant SNR-P4 which restricts the removal process to only the nets with 4 or more pins. In the figures, SNR-X and SNR-P4-X denote the similar-net removal heuristics used with the representative selection option X $\in$ {LRG,IMP,UNI}.

When $t = 8$, we obtain around 10% additional reduction on partitioning time with only 1%–2% reduction on quality. On the other hand, when $t = 4$, since the heuristic is less restricted, the improvements on the partitioning time are larger: 22% and 15%, respectively, for the variants SNR-LRG and SNR-P4-LRG. For these variants the reductions on the partitioning quality are only 5% and 2%, respectively. Overall, by using all the proposed heuristics, we obtained 4.2 speedup w.r.t. base UMPa with 4% reduction on the quality or a 3.9 speedup with only 2% quality reduction.

For the representative selection options, the experiments do not reveal any significant evidence to differentiate LRG and IMP. However, the results show that UNI is slightly worse than the other two for exploiting the time/quality tradeoff. This result is somehow expected since unifying pin sets creates a representative net with a large number of pins which can create a burden for the upcoming matching and refinement heuristics during the multilevel scheme.

### 5.2.4 Summary

A set of lossless and lossy hypergraph sparsification heuristics is investigated in the context of hypergraph partitioning, which is widely used in parallel computing for load balancing and ordering. As the experiments show, the heuristics are highly effective and efficient, and can be easily integrated to the tools used in practice. The effectiveness of the heuristics increases with the number of parts. This makes them more promising and effective since the partitioning overhead becomes an issue while parallelizing a computation with today's high performance machines which have a large number of processors.

In addition to the proposed heuristics, there exist some simple techniques which have already been employed in UMPa base version such as trivially handling the vertices connected to only one net and vice versa. Since these techniques are relatively straightforward, they are not reported. A future work is to exploit graph theoretical structures such as articulation points and bridges in the graph and hypergraph partitioning context.

# Chapter 6: Task Mapping using Geometric Partitioning for Regular Applications

An effective mapping of tasks to processors considers both the tasks' communication pattern and the physical network topology to reduce application communication cost. We propose a new task mapping strategy that uses geometric information to represent application tasks and compute resources. The goal is to assign tasks to cores so that interdependent tasks are performed by "nearby" cores, thus lowering the distance messages must travel, the amount of congestion in the network, and the overall cost of communication. The proposed method applies a geometric partitioning algorithm to both the tasks and the processors, and assigns task parts to the corresponding processor parts. We define metrics based on this geometric information to represent the cost of communication between tasks, and use these metrics to evaluate and select effective mappings.

Much research has focused on mapping tasks to block-based allocations, such as those on IBM's BlueGene systems (e.g., [5, 20, 80, 148]). Our focus is on non-contiguous (i.e., sparse) allocations, where nodes from any portion of the machine

can be assigned to a job without regard to the allocation's shape or locality. Such allocations are used in many parallel systems (e.g., Cray, clusters). Mapping strategies developed for general allocations can be used automatically for the more restricted case of block allocations.

Most previous non-contiguous approaches have represented tasks' communication patterns and network topologies as graphs; graph algorithms were then applied to find good mappings. Finding optimal topology mappings has been shown to be NP-Complete [52,89], so heuristics are often used to reduce complexity (e.g., [18,26,28,49, 50,100]). We, instead, use an inexpensive geometric partitioning algorithm to reorder tasks and processors based on their geometric locality, and use the reordering to map tasks that are "close" to each other geometrically to processors that are "close" to each other in the mesh or torus. Initial experimentation with geometric approaches proved promising [102]; our work improves the geometric strategies, compares them with more sophisticated graph-based methods, demonstrates them for additional applications, and provides software suitable for use in parallel applications.

General-purpose, open-source graph-based mapping algorithms are available. The LibTopoMap library [89] requires as input a task-communication graph describing the amount of communication between tasks, as well as static files describing the network topology. It uses the ParMETIS graph partitioner [94] to divide tasks into $n$ parts, where $n$ is the number of nodes in the allocation, and then applies a graph algorithm (Greedy, Recursive Bisection, Reverse Cuthill-McKee) to map the parts to nodes.

The JOSTLE [144] and Scotch [117] libraries combine mapping with load balancing by using recursive bisection of both network-topology and application-data graphs to partition data and map the resulting parts to processors. Like these libraries, our approach is designed for general-purpose use in applications and is available in the Zoltan2 [30] library.

The main contributions of this work follow.

• We present a new geometric algorithm for task mapping in non-contiguous processor allocations (Section 6.2).

• We present metrics for evaluating mappings in mesh- and torus-based networks (Sections 6.1 and 2.3.2), and validate these metrics using performance counter information from the Cray Gemini routers (Section 6.3).

• We demonstrate our algorithm in two proxy applications on up to 64K cores, and assess the quality of our mappings with respect to application communication cost and execution time (Section 6.3).

• We compare our geometric mappings to applications' default mappings, application-specific optimizations, and the LibTopoMap graph-based mapping library, showing that our geometric mappings reduce both communication time and communication metrics for the target applications relative to other methods (Section 6.3).

## 6.1 Mapping Metrics Revisited

In this chapter, we use two primary metrics to represent the network communication:

• **Average Hop Count:** Average hop count is defined as the average length of paths taken by messages. For the rest of this chapter, we measure the average hop count, and use the terms "hop count" and "average hop count" interchangeably.

$$AverageHopCount(\Gamma) = \text{TH}(\Gamma)/|E_t| \tag{6.1.1}$$

• **Maximum Message Congestion:** The maximum number of messages sent across communication links as given in Equation 2.3.1. In this work, we assume static routing of messages. Also, we assume that each message is transferred over a single path (i.e., messages are not split and sent through multiple paths), and that all links have the same capacity.

Maximum message congestion represents the maximum number of messages that go through any link. We refer to maximum message congestion as "congestion" in the rest of this chapter. Since communication is a real time process and is affected by many outside factors (e.g., network traffic and overhead from competing jobs), theoretical metrics can only approximate actual communication time. Experiments in Section 6.3 validate the efficacy of our two metrics.

## 6.2 Geometric Task Mapping

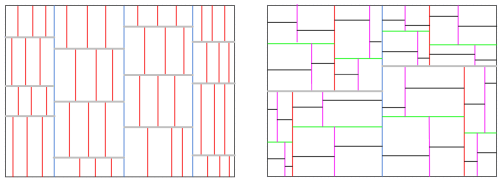
Our proposed topology-aware mapping algorithm uses the router coordinates to represent the network topology of the machine. The cost of communication between pairs of cores is approximated by the dilation (Eqn. 2.3.1) of their routers' coordinates. Thus, the machine topology is described only by the cores' coordinates, rather

than a topology graph in which bandwidth information between every pair of cores must be specified. Each of the application's MPI processes is also represented by a coordinate, corresponding to either the center of the process' application domain or the average coordinate of its application data. For example, in a structured grid-based finite difference application, the center of an MPI process' subgrid can be used as its coordinate. Our algorithm uses a geometric partitioning algorithm to consistently reorder both the MPI processes and the allocated cores; this reordering is used to construct the mapping. In this section, we provide details of our mapping algorithm. We use the term "machine coordinates" to refer to the router coordinates associated with each core, and "task coordinates" to refer to the centroid or averaged coordinates provided by the application's MPI processes.

### 6.2.1 Multi-Jagged (MJ) Algorithm for Geometric Partitioning

Our proposed task mapping algorithm uses a geometric partitioning algorithm, the Multi-dimensional Jagged algorithm (MJ) [62] of the Zoltan2 Toolkit [30], to partition task and machine coordinates. The MJ algorithm partitions a set of coordinates into a desired number of parts ($P$) in a given number of steps called the *recursion depth* ($RD$). During each recursion, one-dimensional partitioning is done along a dimension; the dimension is alternated at each recursion. Therefore, MJ is a generalization of the Recursive Coordinate Bisection (RCB) algorithm [16] in which the MJ algorithm has ability to do multi-sections instead of bisections. Although our implementation

of MJ can partition into any number of parts $P$, we simplify our explanation here by assuming $P$ can be written as $P = \prod_{i=1}^{RD} P_i$. In the first level, MJ partitions the domain into $P_1$ parts using $P_1 - 1$ cuts in one direction. In the next level, each of the $P_1$ parts is partitioned separately into $P_2$ parts using cuts in an orthogonal direction. This recursion continues in each level. Figure 6.1 shows two 64-way partitions using MJ with $RD = 3$, $P = 4 \times 4 \times 4$ (left), and $RD = 6$, $P = 2 \times 2 \times 2 \times 2 \times 2 \times 2$ (right). When $RD = \lceil \log_2 P \rceil$, MJ is equivalent to RCB (as in Figure 6.1b).



(a) $RD = 3$ (b) $RD = 6$

Figure 6.1: Partitioning into 64 parts using MJ with different recursion depths. Cutlines in the same level of recursion share the same color.

MJ's complexity depends on $P$, $RD$, the number of points $n$, and the average number of iterations $it$ needed to compute cutline locations. During partitioning on level $i$, each point is compared to $\log_2 P_i$ cut lines (using binary search). Thus, MJ's complexity is $O(n \times it \times \sum_{i=1}^{RD} \log_2 P_i)$. When MJ is used as RCB, its complexity is $O(n \times it \times \log_2 P)$.

### 6.2.2 Using MJ for Task Mapping

Although MJ is proposed as a parallel (MPI+OpenMP) algorithm [62], we use it as a sequential algorithm in this context. The size of the partitioning problem is proportional to the number of processors. Since current supercomputers typically have $O(100K)$ processors, the partitioning algorithm would be communication bound if done in parallel; little or no speedup would be obtained by parallelizing this process. Instead, each processor calculates the same mapping independently. A reduceAll operation is performed at the beginning of task mapping to provide all machine and task coordinates to every processor. Then every processor performs the sequential mapping operation and obtains the exact same mapping. We describe in Section 6.2.3 how we exploit parallelism to improve the quality of the mapping with minimal additional cost.

The proposed mapping algorithm is defined as follows: Given $tdim$-dimensional coordinates of the tasks ($tc$), and $pdim$-dimensional coordinates of the cores ($pc$), together with the number of tasks ($tn$) and cores ($pn$), the algorithm returns a mapping from cores to tasks ($p2t$) (and/or tasks to cores $t2p$). Algorithm 22 gives the description of the task mapping algorithm.

MJ's main purpose in Algorithm 22 is to consistently number the cores and tasks. Function MJ partitions the task and cores into $usedNumProcs$ parts, and assigns a part number to each core and task. Cores and tasks that share the same part number

**Algorithm 22:** Task Mapping Algorithm using MJ

```
Require: tc, tdim, tn, pc, pdim, pn, RD
  minDim ← min(tdim, pdim)
  usedNumProcs ← numParts ← min(tn, pn)
  if pn > tn then
    procPerm ← getClosestSubset(pc, pdim, pn, tn)
  else
    procPerm ← range(0, pn)
  taskPerm ← range(0, tn)
  taskParts ← MJ(tc, minDim, tn,
          taskPerm, numParts, RD)
  procParts ← MJ(pc, minDim, usedNumProcs,
          procPerm, numParts, RD)
  p2t, t2p ← getMappingArrays(taskParts, procParts,
          taskPerm, procPerm, tn, pn)
```

are then mapped to each other by GETMAPPINGARRAYS, and the resulting mappings are stored in $p2t$ and $t2p$.

Since the tasks and cores are partitioned separately, Algorithm 22 ensures consistent part numbering among both MJ calls. First, the minimum dimension is chosen between the tasks and cores. For example, if $pdim = 3$ while $tdim = 2$, one of the cores' coordinates is ignored to ensure that the geometric partitioner follows the same order in both of the partitioning operations. Next, the algorithm can follow different paths depending on the number of coordinates of tasks and cores. There are three possibilities at this step:

**1)** $tn = pn$**:** A one-to-one mapping between cores and tasks exists. For task $t$ assigned to core $p$, $t = p2t[p]$ and $p = t2p[t]$.

**2)** $tn > pn$**:** When there are more tasks than cores, a core is assigned multiple tasks. Both cores and tasks are partitioned into $pn$ parts, with multiple tasks in the each part. The mapping results will be $t \in p2t[p]$ and $p = t2p[t]$.

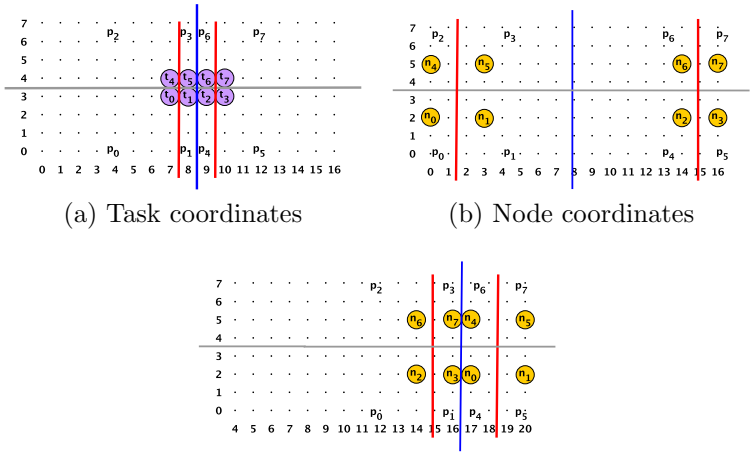
**3)** $tn < pn$**:** When there are more cores than tasks, the algorithm does not split a task among multiple cores. Instead, during a preprocessing step, it chooses a subset of $tn$ cores. Then, mapping is performed within this subset as if $tn = pn$. Some cores will be idle, as they are not assigned any tasks. Our implementation uses a modified K-means clustering algorithm [83] to choose the closest subset of cores within the allocation; in this chapter, however, this special case is not considered.

The complexity of Algorithm 22 is dominated by the calls to MJ, since $getMappingArrays$ runs in linear time with respect to $tn$ and $pn$. Thus, when MJ is used as RCB and $tn = pn$, the overall complexity of the mapping algorithm is $O(tn \times it \times \log_2(tn))$.

### 6.2.3 Improving the quality of the mapping

The ability of our mapping strategy to reduce communication costs depends on the results of the MJ partitioner. In this section, we describe several ways that we can improve the quality of the mapping by modifying the input to MJ. These improvements are computed with very little extra expense, as they are computed in parallel across sets of processors.

**Shifting the machine coordinates:** The first improvement involves considering the 3D torus interconnection present in many supercomputer networks. Torus networks provide wrap-around communication links in each network dimension that are



(a) Task coordinates

(b) Node coordinates

(c) Shifted node coordinates

Figure 6.2: An example showing the benefit of shifting node coordinates in torus networks. The numbers of nodes and tasks are equal. Tasks and nodes sharing the same number are mapped to each other (6.2a). Assuming nearest-neighbor communication, the unshifted mapping (6.2b) has average hop count of 3.66 and 3 in the $x$ and $y$ directions, respectively; note that messages between $n_1$ and $n_2$ and between $n_5$ and $n_6$ require six hops due to wrap-around links. With the node partition obtained after shifting around the wrap-around links (6.2c), the mapping has average hop count of 2 and 3 in $x$ and $y$.

not reflected in the machine coordinates. Thus, since MJ is not aware of connectivity information, MJ considers nodes at edges of the network coordinates to be far apart, even though there is a one-hop path between them.

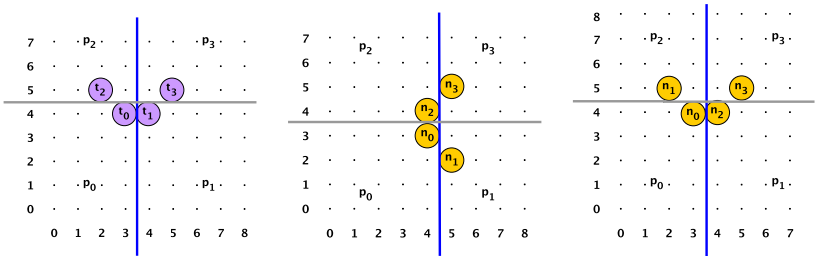
In our proposed task-mapping method, we transform the coordinates to account for wrap-around in each dimension. Our shifting strategy applies a one-dimensional operation to each dimension independently. First, we find the shift position – the largest gap in the node coordinates. Then, assuming the largest gap is greater than

one, we transform the machine coordinates on one side of the shift position by adding to them the maximum machine coordinate in that dimension. Ties in the largest gaps are broken using the number of nodes that are on either side of the shift positions. Even though the list of machine coordinates is not sorted for any dimension, gap detection can still be performed in O($pn$) time by using a counting sort algorithm.

Figure 6.2 shows an example of mapping eight tasks onto eight nodes in a 17x8 2D torus topology. In Figure 6.2b, the maximum gap along the $x$ dimension is found between $n_1$ ($x = 3$) and $n_2$ ($x = 14$) (also between $n_5$ and $n_6$). The heuristic shifts the coordinates of all nodes that have $x \leq 3$ by adding the maximum $x$ coordinate (17) to the $x$ coordinates of these nodes. Figure 6.2c shows the updated machine coordinates after this shift operation. Assuming nearest neighbor communication for the tasks in Figure 6.2a, the mapping in Figure 6.2b results in average hop counts of 3.66 and 3 in the $x$ and $y$ dimensions, respectively. After shifting, the mapping in Figure 6.2c achieves average hop counts of 2 and 3.

**Rotating the machine and task coordinates:** The quality of the mapping also depends on the order of the dimensions to which the partitioning is applied (e.g., first partition in $x$, then $y$, then $z$). For example, Figure 6.3 shows how the quality of the mapping can change by choosing a different order of dimensions in partitioning.

It is difficult to predict which dimension ordering for partitioning will provide the best mapping quality. One could choose a permutation of the dimensions based on the aspect ratios of the machine and task coordinates. The permutation that makes




(a) Task coordinates (b) Node coordinates (c) Rotated node coordinates

Figure 6.3: An example showing the benefit of rotating the node orientation. Assuming communication is required between only tasks 0 and 2, the unrotated mapping (6.3b) has average hop count of 1 and 1 in the $x$ and $y$ directions, respectively. In the rotated node-partition (6.3c), partitioning is performed in the $y$ dimension first, and then in the $x$ dimension. The mapping obtained after rotation has average hop count of 1 and 0 in $x$ and $y$.

the aspect ratios along dimensions closest can be chosen as the best permutation. However, as our experiments will show, this greedy method fails to find the best permutation in most of the mappings. To overcome this issue, we use a speculative method. Recall that, in Section 6.2.2, we described sequential task mapping in which every process computed the same mapping. Since there are $pn$ processes, we instead calculate different mappings with different rotations in each process. Then, given the communication pattern of the tasks, each mapping's hop count is computed, and the one with the lowest hop count is chosen. This comparison requires one extra reduceAll and broadcast operation. If the dimensions of the tasks and the machine are $tdim$ and $pdim$, there are $rp = (tdim)! \times (pdim)!$ different rotations. For a 3D

torus with 3D task coordinates, $rp = 3! \times 3! = 36$. We group processes into sets of size 36, in which each process calculates a mapping using a different rotation. Each process calculates the quality of its own mapping. Then within each group, the best quality mapping is determined, and is broadcast to the group. When the number of processes is not divisible by $rp$, the remaining processes are distributed among groups so that as many rotations as possible are calculated within each group.

Similarly, reflections of the node or task coordinates along coordinate axes could be done. The total number of different reflections is $2^{maxDim}$, where $maxDim = max(tdim, pdim)$. Again, the processes can be grouped such that each group has $2^{maxDim}$ processes that each calculate a different mapping. Combined with the rotation operation, the total number of different solutions becomes $2^{maxDim} \times tdim! \times pdim!$, which is 288 for the usual case of a 3D torus with 3D task coordinates. Our implementation includes the rotations described above, but does not yet include reflections.

## 6.3 Experiments

We tested our geometric mapping methods in two proxy applications: MiniGhost [15] and MiniMD [88]. For each application, we ran weak scaling experiments to evaluate the effect of mapping on communication and execution time. We compared our geometric method with the applications' default task layout and with the graph-based task mapping library LibTopoMap [89]. For MiniGhost, we also compare with an

| Method | Abbreviation | Description |
|---|---|---|
| No mapping | None | Task $i$ performed by core $i$. |
| Multicore Grouping | Group | 16-task blocks; 2x2x4 tasks per block. |
| Geometric | Geom | Geometric with one rotation w.r.t. aspect ratio |
| Geometric + Rotations | Geom+R | Geometric with 36 rotations |
| Geometric + Rotations + Coordinate Shift | Geom+R+S | Geometric with 36 rotations and torus-aware shifting |
| LibTopoMap | TopoMap | Graph-based mapping [89] |

Table 6.1: Mapping methods used in experiments

application-specific grouping of tasks for multicore nodes. These mapping methods are described below and summarized in Table 6.1.

• **None:** The application's default mapping of tasks to ranks: task $i$ is performed by rank $i$.

• **Multicore Grouping (Group):** Tasks reordered into 16-task blocks, with 2x2x4 tasks per block. A block is then assigned to cores within the same node, so that frequently communicating tasks are within the same node. However, this reordering does not account for inter-node communication, since it does not use any information about the position of the nodes in the network. Group exploits application-specific knowledge about the finite-difference grid; thus it is available only in MiniGhost.

• **Geometric (Geom):** Geometric mapping [102] with recursion depth $RD = \lceil \log_2 P \rceil$ (i.e., performing bisection at each level). A single rotation is determined at the beginning of the algorithm by using the aspect ratios of the task and machine coordinates. Ties among coordinates along a dimension are broken arbitrarily; if several $x$ coordinates lie along a cut, the choice of coordinates that go to the left of the cut or to the right is arbitrary.

• **Geometric with Rotations (Geom+R):** Geometric (MJ) mapping with recursion depth $RD = \lceil \log_2 P \rceil$; it calculates 36 different solutions according to 36 different rotations, and chooses the one with the lowest hop count metric. Ties among machine coordinates are broken first by coordinates in other dimensions, and then by MPI ranks so that coordinates with lower MPI ranks go to one side and those with higher ranks to the other.

• **Geometric with Rotations and Coordinate Shift (Geom+R+S):** Geom+R mapping with coordinate shifting done as preprocessing to account for torus networks.

• **LibTopoMap (TopoMap):** Graph-based mapping strategies available in the open-source library LibTopoMap [89]. For each experiment, we use the result with the lowest application execution time among LibTopoMap methods Greedy, Recursive Bisection, and Reverse Cuthill-McKee. If LibTopoMap does not find a mapping that is better than the input mapping, it returns the input mapping.

We ran all experiments on the DOE Cielo Cray XE6 at Los Alamos National Laboratory, and the Hopper Cray XE6 at NERSC. On both platforms, we used

gcc 4.7.2 compilers and Cray's MPICH2 implementation. Our geometric mapping techniques are implemented in the Zoltan2 library [30].

### 6.3.1 Mapping in a finite difference application

We compared the effect of our mapping method in MiniGhost [15], a finite-difference proxy application that implements a finite difference stencil and explicit time-stepping scheme across a three-dimensional uniform grid. Using a seven-point stencil, each task communicates with two neighbors along each dimension; tasks along geometry boundaries communicate with only their neighbors interior to the boundary (i.e., boundary conditions are non-periodic). Each task is assigned a subgrid of the 3D grid based on its task number. The numbers of tasks in each dimension $pn_x, pn_y, pn_z$ (with $pn_x \times pn_y \times pn_z = pn$) are specified by the user. Subgrids of the 3D grid are assigned to tasks by sweeping first in the $x$ direction, then the $y$ direction, then the $z$ direction. Thus, task $i$ shares subgrid boundaries (and, thus, requires communication) with tasks $i+1$ and $i-1$ to its east and west, respectively; with tasks $i+pn_x$ and $i-pn_x$ to its north and south; and with tasks $i+(pn_x)(pn_y)$ and $i-(pn_x)(pn_y)$ to its front and back. In the default MiniGhost configuration, task $i$ is performed by rank $i$.

As shown in [14], the execution time of MiniGhost with its default mapping does not scale well in weak scaling tests. Our goal is to improve scalability by mapping tasks onto processors so that tasks that share boundaries are placed "near" each other

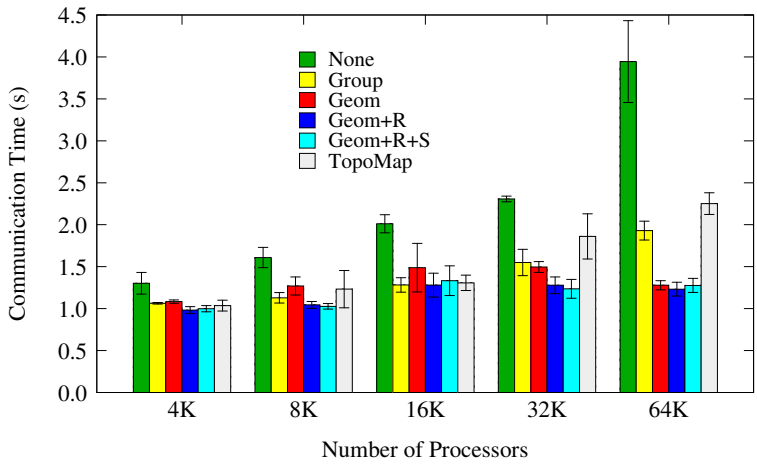
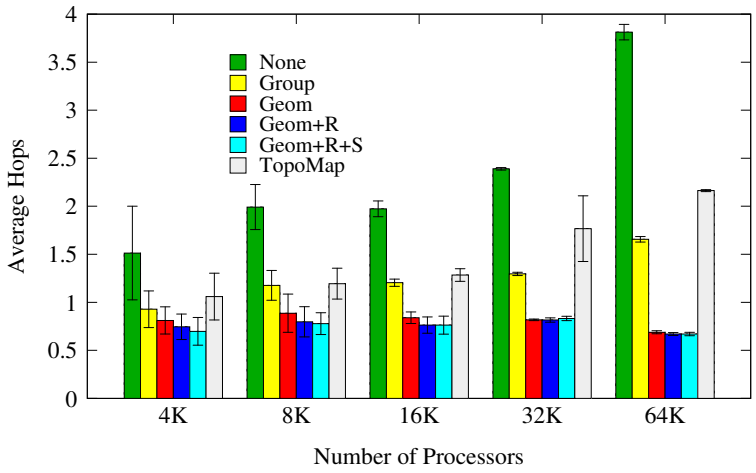


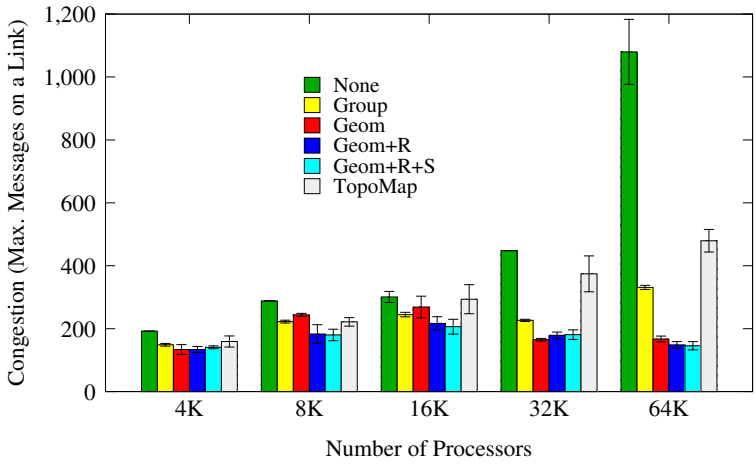

Figure 6.4: Maximum communication time in weak scaling experiments with MiniGhost

in the allocation. We ran weak-scaling experiments with 4096–65,536 processors (256–4096 nodes) of Cielo. Each task owned a 60×60×60-cell subgrid; we ran the simulation for 20 timesteps with 40 variables per grid point. For each experiment, we obtained a node allocation of the requested size, and ran all mapping methods within that allocation. We repeated each experiment five times with different allocations, and averaged the results across the five instances; error bars in figures show the standard deviation from the averages.

Figure 6.4 shows the maximum communication time (across processors) for weak scaling experiments with MiniGhost. With MiniGhost's default mapping (None), communication time increases dramatically as the number of processors is increased.




(a) Average Hop Count




(b) Maximum Congestion

Figure 6.5: Average Hop Count (a) and Maximum Congestion (b) for weak scaling experiments with MiniGhost

| | None | | Group | | Geom | | Geom+R | | Geom+R+S | | TopoMap | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | % Comm | Total | % Comm | Total | % Comm | Total | % Comm | Total | % Comm | Total | % Comm |
| 4K | 6.07 | 16.2% | 5.70 | 13.9% | 5.76 | 15.4% | 5.75 | 13.6% | 5.77 | 13.9% | 5.74 | 14.2% |
| 8K | 6.40 | 18.3% | 5.91 | 14.3% | 6.05 | 15.7% | 5.80 | 13.8% | 5.78 | 13.8% | 5.97 | 14.5% |
| 16K | 7.16 | 16.9% | 6.14 | 14.2% | 6.36 | 15.4% | 6.22 | 14.2% | 6.23 | 14.2% | 6.15 | 14.6% |
| 32K | 7.60 | 19.0% | 6.94 | 13.0% | 7.29 | 13.8% | 6.41 | 12.9% | 6.38 | 13.0% | 6.82 | 14.1% |
| 64K | 9.57 | 24.4% | 8.31 | 12.2% | 7.53 | 13.2% | 6.29 | 13.4% | 6.29 | 13.4% | 8.26 | 13.8% |

Table 6.2: Total execution time and percentage of that time spent in communication in weak scaling experiments with MiniGhost.

MiniGhost's Group method controls the growth in communication costs, but consistent with results in [14], costs increase at the highest processor counts. Geometric methods Geom+R and Geom+R+S provide the lowest communication costs, and, as desired for weak scaling, the communication costs remain nearly constant as the number of processors increases. Above 16K processors, denser node allocations ($> 20\%$ of the total machine) help the geometric methods further reduce average hop count and congestion. TopoMap also reduced communication relative to MiniGhost's default mapping, but was unable to reduce communication as much as Group, Geom+R and Geom+R+S. Table 6.2 shows the total execution time and the percentage of total execution time that was spent in communication. These results show the importance of topology-aware mapping in general, as all mapping strategies maintained scalable communication better than the default layout. Geom+R and Geom+R+S provided the most consistent performance, maintaining communication as 13–14% of total execution time.

Figure 6.5 shows the calculated quality metrics: average $x+y+z$ hops (Eqn. 6.1.1) and maximum congestion (Eqn. 2.3.1). As the number of processors increases, the average hop count, congestion and communication cost all follow the same upward trend for the default MiniGhost mapping. Since Group does not account for inter-node communication, its average hop count increases with the number of nodes. TopoMap's average hop count and max congestion also increase as we scale to larger number of processors. Average hop count for the geometric mappings Geom, Geom+R and Geom+R+S, however, remains nearly unchanged as we use more processors, suggesting greater scalability using the geometric mappings. Congestion is also low for the geometric mappings, resulting in lower communication cost.

Among the geometric methods, Geom+R and Geom+R+S provided the best mappings. The benefit of coordinate shifting in Geom+R+S is small for these experiments. Geom+R+S usually produces hop counts no greater than Geom+R, and when the allocation allows, can reduce the hop count relative to Geom+R. Thus, the averaged values are very similar. Geom+R+S has slightly greater variation in hop counts for individual experiments, since for some allocations, it can further reduce the hop count via shifting.

The effect of communication between two nodes connected by the same Gemini router is not accounted for in the average hop metric, since both nodes in this case have the same machine coordinate. But this communication can impact overall communication costs. In Figure 6.6, we show the percentage of messages that go between

the two nodes in a Gemini router (i.e., the percentage of communication that is intra-Gemini communication) for each method. Arbitrarily breaking ties among machine coordinates in Geom leads to high intra-Gemini communication; tasks with the same machine coordinates are placed in either node without regard to their positions. Since intra-Gemini communication is more expensive than intra-node communication, reducing the amount of intra-Gemini communication (through grouping as in Group or better tie breaking as in Geom+R and Geom+R+S) can reduce overall communication costs. Interestingly, the very low intra-Gemini communication for the default mapping ("None") on 64K processors is due to the experiment's $32 \times 64 \times 32$-task configuration. By default, tasks are ordered by first sweeping in the $x$ direction. Thus, with 32 tasks in the $x$ direction, the first 16 tasks are given to one node attached to a Gemini, and the next 16 are given to its other node. Thus, only two tasks share an intra-Gemini boundary (the two in the middle of each $x$ sweep), keeping intra-Gemini communication very low. Other configurations (e.g., $16 \times 64 \times 64$ tasks) would not have this happy benefit. These results show the importance of minimizing both intra-Gemini communication and hop count.

As a final step, we leveraged the Cray Gemini's performance counters to measure network congestion empirically [114]. Our model calculates network congestion assuming that all messages are transferred simultaneously. In reality, the message traffic is spread over time and messages interleave with one another. We measured

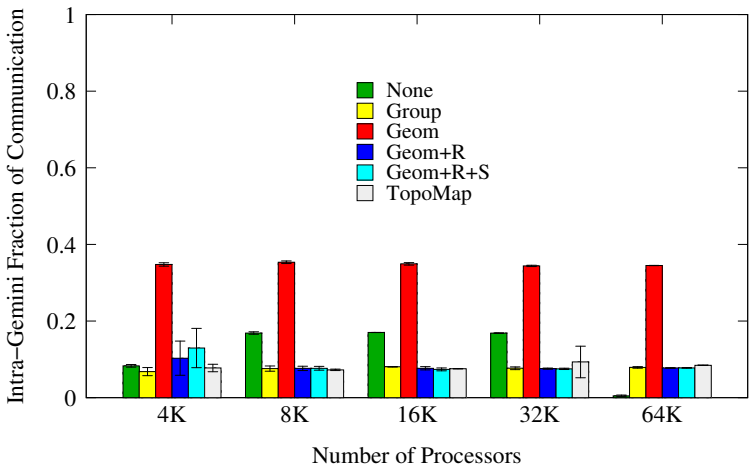



Figure 6.6: Intra-Gemini communication: the fraction of total communication between processors in different nodes sharing the same machine coordinate. This communication is not reflected in the hop count metric.

this real-time behavior using the Gemini's per-link stall cycle counters, which increment whenever a message can not move towards its destination due to network congestion. For each Gemini being used by an experiment, we captured the stall counters for each of the Gemini's seven network links (XYZ links plus host link). We then calculated summary statistics such as minimum/maximum/average number of stalls encountered over all links, over only host links, over only X links, etc. We omit a full analysis due to space, but in general the empirical measurements closely match the predictions of our model. Table 6.3 lists the Pearson correlation coefficients comparing modeled and measured values over all 605 experiments performed for this work. The column labeled "Measured Max Stalls" corresponds to the link with the

| | Modeled Avg Hop Count | Modeled Max Congestion | Measured Max Stalls |
|---|---|---|---|
| Max Comm Time | .835 | .915 | .936 |
| Total Time | .760 | .872 | .886 |

Table 6.3: Correlation coefficients comparing measured run times with computed metrics and network counter data. A coefficient of one indicates perfect linear correlation.

highest network stall count for each experiment (i.e., the link with the most congestion). This metric is found to have the best correlation to maximum communication time, and correlates well with our model's calculated maximum congestion metric. The modeled maximum congestion metric correlates slightly less well to the measured run times, possibly due to interference from other jobs running in the system and the heterogeneous link speeds in the network, which we do not account for. Finally, the average hop count metric is slightly less correlated to the measured run times than the congestion metrics. The empirical data suggest that our congestion metrics are accurate for MiniGhost and that the maximum congestion metric should be preferred over the average hop count metric.

Overall, our geometric mapping methods reduced the total execution time by 5-34% relative to the default MiniGhost mapping, and 0-17% relative to the application-specific Group mapping available in MiniGhost. The largest reductions were seen at

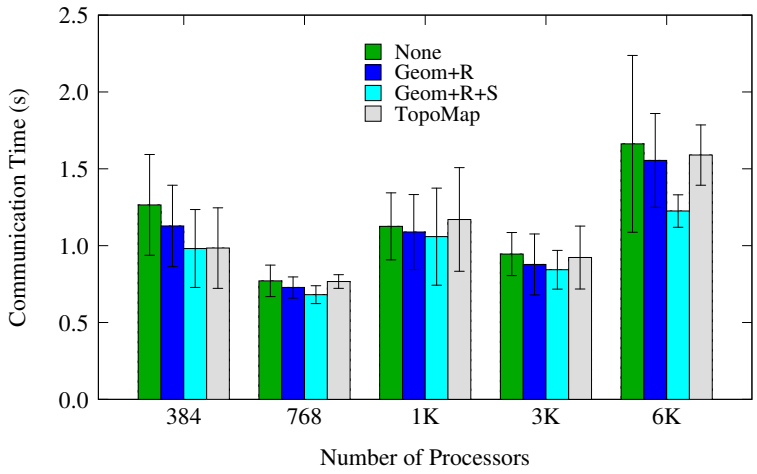



Figure 6.7: Maximum communication time in weak scaling experiments with MiniMD

the highest processor counts, reflecting the importance of mapping as the number of cores in parallel computers increases.

### 6.3.2 Mapping in a molecular dynamics application

MiniMD, part of Mantevo [88], is an application proxy for parallel molecular dynamics (MD) simulations. It implements several algorithms that are typical in MD codes and emulates the key performance characteristics of larger molecular dynamics codes like LAMMPS [121]. MiniMD essentially solves Newton's equation on $N$ particles. In our experiments, we used the MiniMD algorithm that uses the Lennard-Jones potential for force calculation. MiniMD assigns each processor a fixed spatial dimension in 3D by splitting the spatial dimension into small boxes. At each timestep, each processor computes forces and updates the positions of atoms within its box. The

processor communicates with its nearest neighbors in order to compute the forces of the atoms in its box. As atoms move they are reassigned to different processors. Compared to MiniGhost, an important difference in MiniMD's communication pattern is that, based on the cutoff distance used to calculate the force, a processor might need to communicate to more than one processor in a direction when the box sizes are smaller than the cut-off. However, no processor communicates to its corner neighbors or to its neighbors that are more than one hop away. Instead it communicates to its six nearest neighbors multiple times to achieve the same purpose.

MiniMD reorders the MPI ranks into 3D processor grid and assigns the boxes to the ranks based on the rank's position in the logical processor grid. It uses the Cartesian topology interfaces in MPI (MPI_Cart_Create, MPI_Cart_Shift and MPI_Cart_get) to do the reordering. While theoretically MPI can reorder the ranks based on the topology, optimizing for nearest-neighbor communication, we do not observe that in practice. The MPI implementation usually returns the ordering we call "None." We replace the mapping of the tasks to the processor grid by using two of our best algorithms (Geom+R, Geom+R+S) and TopoMap. Our scaling studies for MiniMD were run on Hopper, the Cray XE6 at NERSC. We do a weak scaling study from 384 cores up to 6144 cores. The number of atoms increases from 415K to 6.7M. We use three repetitions for our experiments. Figure 6.7 shows the maximum communication time of the MiniMD runs in our weak-scaling study. Our geometric algorithms reduce the communication time when compared with the mapping provided

by the MPI implementation on all processor counts. Compared to the default MPI mapping, Geom+R+S algorithm reduced the communication time of MiniMD by 6% to 27%. Unlike in MiniGhost, in MiniMD, Geom+R+S does better than Geom+R in larger core counts. At 384 cores, TopoMap ties Geom+R+S in terms of the average communication time. However, Geom+R+S mapping results in reduction in communication time over mappings from TopoMap at all core counts beyond 384. For MiniMD the reduction in communication time by using Geom+R+S over TopoMap range from 1% (384 cores) to 23% (6K cores). More notably, Geom+R+S mappings do better than TopoMap as the core counts increase. We do not present the metrics for the MiniMD runs here due to lack of space. However, as one would expect, we observed reductions in average hop count over no mapping.

## 6.4 Summary

We have proposed a new topology-aware task mapping method that uses multi-jagged geometric partitioning (MJ) to reorder the given task and processor coordinates in a way that assigns communicating tasks to "nearby" processors. This method is designed to be effective on mesh and torus-based networks with non-contiguous node allocations, such as the Cray XE6, but extends naturally to block allocations as in IBM BlueGene systems. We also have proposed several improvements (e.g., multiple rotations, coordinate shifting) that improve geometric mappings relative to a baseline geometric method. We compared our method with the applications' default mapping, as well as application-specific mappings and graph-based mappings from

LibTopoMap. to improve the quality of the mapping. We showed that our geometric mappings reduced application execution time up to 34% on 64K cores relative to the default mapping for the MiniGhost finite difference proxy application, and up to 23% on 6K cores for the MiniMD molecular dynamics proxy application. We correlated communication time in MiniGhost with computed metrics (average hops, congestion) and validated these metrics with congestion information obtained from the Gemini routers' counters.

As future work, our mapping methods will be extended to accommodate non-uniform bandwidths in the dimensions of the torus networks. We will also investigate the effect of geometric mapping on unstructured applications. And we will experiment with the processor-subset selection via k-means clustering mentioned in Section 6.2 to provide effective mappings when there are fewer tasks than processors. Our test application will be a multigrid-based linear algebra algorithm, in which coarse matrices may be too small to efficiently utilize all cores needed for the fine matrices; in such cases, we will use our modified k-means clustering algorithm to select a subset of cores to use.

# Chapter 7: Topology-Aware Task Mapping Using Graph Models

For parallel computing, task mapping has a significant impact on the performance, especially when supercomputers with hundreds of thousands or millions of processors shoulder the execution. It is usually the case that the communication pattern between the tasks has already been designed to minimize possible performance bottlenecks, such as high number of messages or communication volume, via tools such as graph and hypergraph partitioners, e.g., [38, 61, 93, 117, 124]. However, this effort alone is not sufficient, since the mapping-based metrics such as the maximum link congestion and the total number of hops the messages travel in the network can also be significant bottlenecks on the performance. This is especially true for today's supercomputers with large-diameter interconnection networks, and concurrent and non-uniform job submissions yielding sparse and wide-spread processor allocations for parallel applications. There exist various studies in the literature which analyze the impact of task-to-processor mappings on the parallel performance [3, 20, 33, 97] and report significant speedups, e.g., 1.64X [80], just with an improved mapping.

There are two main research directions for mapping. The first one focuses on *block-based processor allocations*, e.g., the ones on IBM BlueGene [5, 20, 80, 111, 148]. The second direction focuses on *sparse allocations* in which the allocated processors are not contiguous within the network [63, 89]. This direction is more general: sparse allocations have been used in various parallel systems, and the mapping algorithms based on this model can also be used for the block-based model. In this work, we follow the second direction.

The problem of finding an optimal task-to-processor mapping is NP-complete [89], and various heuristics have been proposed [18, 28, 49, 50]. Many of these heuristics use graphs and related combinatorial structures to model the task interactions as well as the network topologies. For example, the open-source mapping library LibTopoMap [89] uses a task graph and network topology information. The task graph is first partitioned into the number of allocated nodes, and various graph-based algorithms are used to map the tasks to the allocated processors. Other libraries such as JOSTLE [143] and Scotch [117] also exist. These two libraries apply simultaneous partitioning and mapping of the task and topology graphs.

A good mapping algorithm must be able to provide high-quality task-to-processor mappings. It also needs to be efficient in order not to intervene the supercomputer's performance. We follow these two important criteria and propose novel, very efficient, refinement-based mapping algorithms. We show that they can produce high-quality mappings w.r.t. the topology-related metrics such as the average link congestion and

the total hops the messages take. We compare the performance of the proposed algorithms with that of LibTopoMap and Scotch. The experiments on a supercomputer with a 3D torus network and 4096 processors show that the algorithms can improve the *weighted hop* and the *maximum congestion* (that will be described below) by 16% and 32% on the average, respectively, compared to the default mapping. These metric improvements yield a 43% performance improvement on one case for a synthetic, communication-only application (Figure 7.4b–PATOH) and a 23% improvement on the performance of a sparse-matrix vector multiplication (SpMV) kernel (Figure 7.5–METIS). Overall, with 4096 and 8192 processors, they improve the performance of a parallel SpMV kernel and a communication-only application by 9% and 14%, respectively (Table 7.2).

The organization of the chapter is as follows. In Section 7.1, we present three mapping algorithms minimizing various metrics. Section 7.2 presents the experimental results, and Section 7.3 concludes the paper. Note that, the background information on the topology-aware task mapping and the target architecture can be found at 2.3.

## 7.1 Fast and High Quality Mapping Algorithms

We propose three mapping algorithms to minimize WH and MC. Here we will describe these algorithms. Their adaptation for TH and MMC is trivial. Among them, the ones that minimize WH can be applied to various topologies, whereas those minimizing MC require static routing.

### 7.1.1 A Greedy mapping algorithm

The first algorithm *Greedy Mapping* given in Algorithm 23 finds a mapping $\Gamma : V_t \rightarrow V_m$ to minimize WH. It uses the task graph $G_t$ and the topology graph $G_m$. The algorithm is similar to *greedy graph growing*, and initially maps $N_{BFS}$ seed task vertices to the nodes. It assumes a symmetric $G_t$ while finding the neighbors of a given task since WH is an undirected metric, i.e., the number of hops between $m_1$ and $m_2$ is the same regardless of direction.

**Algorithm 23:** Greedy Mapping

**Data**: $G_t = (V_t, E_t), G_m = (V_m, E_m)$: task and topology graphs, $N_{BFS}$: # vertices to be initially mapped

$conn_t \leftarrow 0$ for each $t \in V_t$ ▶ *initialize the max-heap*
$\Gamma[t] \leftarrow -1$ for each $t \in V_t$ ▶ *initialize the mapping*
▶ *Find the task with MSRV*
$t_0 \leftarrow t_{MSRV}$
▶ *Map $t_0$ to an arbitrary node*
$\Gamma[t_0] \leftarrow m_0$
▶ *Update connectivity for the tasks in $nghbor(t_0)$*
**for** each $t_n$ in $nghbor(t_0)$ **do**
    $conn.update(t_n, c(t_0, t_n))$
**while** there is an unmapped $t$ **do**
    **if** number of mapped tasks $< N_{BFS}$ **then**
        $t_{best} \leftarrow$ the farthest unmapped task ▶ *found by BFS*
    **else**
        $t_{best} \leftarrow conn.pop()$ ▶ *the one with maximum conn.*
1   $m_{best} \leftarrow$ GETBESTNODE$(t_{best}, G_m, G_t, \Gamma, conn)$
    $\Gamma[t_{best}] \leftarrow m_{best}$
    **for** each $t_n$ in $nghbor(t_{best})$ **do**
2       $conn.update(t_n, c(t_{best}, t_n))$

Throughout the algorithm, the total connectivity of each task to the mapped ones are stored in a heap *conn*. The algorithm first maps the task $t_{MSRV}$ with the maximum send-receive communication volume to an arbitrary node. Until all tasks are mapped, the algorithm gets an unmapped task from *conn* after all the $N_{BFS}$ seeds are mapped. Otherwise, the farthest task to the set of mapped tasks is found by a breadth-first search (BFS) on $G_t$ where all the mapped tasks are assumed to be at level 0 of the BFS. Ties are broken in the favor of the task with a higher communication volume. If $G_t$ is disconnected, a task with the maximum communication volume from one of the disconnected components is chosen. Once $t_{best}$ is found, its best node is obtained by GETBESTNODE. If $t_{best}$ is connected to none of the mapped tasks, GETBESTNODE performs a BFS on $G_m$ to **return** one of the farthest allocated nodes to the set of the non-empty nodes, i.e., the ones with a mapped task. Otherwise, if $t_{best}$ is connected to at least one of the mapped tasks, a BFS on $G_m$ is performed from the nodes to whom one of the $nghbor(t_{best})$ is mapped (again assuming these nodes are at level 0). As an early exit mechanism, a BFS stops when the empty nodes (without a mapped task) are found at a BFS level. Then among these empty nodes, the one with the minimum WH overhead is returned. Therefore, the algorithm performs multiple BFS executions on $G_t$ and $G_m$.

For simplicity, the description above assumes one-to-one task-to-node mapping, i.e., $|V_t| = |V_a|$. In reality, each node has multiple processors, so multiple tasks can

be assigned to a single node. These cases can be addressed by using the computation loads and capacities, and modifying GETBESTNODE so that it returns only a node with some free capacity. Another common solution is using traditional graph partitioning as a preprocessing step to reduce the number of tasks to the number of the allocated nodes while minimizing the edge-cut [89]. We follow this approach and use METIS [93] to partition $G_t$ into $|V_a|$ nodes, where the target part weights are the number of available processors on each node in $V_a$. Since graph partitioning algorithms do not always obtain a perfect balance, as a post processing, we fix the balance with a small sacrifice on the edge-cut metric via a single Fiduccia-Mattheyses (FM) iteration [71]. When the number of processors in the nodes are not uniform, we map the groups of tasks with different weights at the beginning of the greedy mapping since their nodes are almost decided due their uniqueness.

In the algorithm, $N_{BFS}$ controls the number of initial seed mappings. A large $N_{BFS}$ distributes the loosely connected components of the task graph to the nodes that are farther from each other. However, this will not work well for the task graphs with a low diameter. In our implementation, we use $N_{BFS} \in \{0, 1\}$ to generate two different mappings and return the one with the lower WH.

The complexity of the algorithm is dominated by the operations at lines 1 and 2. Each update of the heap (line 2) takes $\mathcal{O}(\log |V_t|)$ time, and this line is executed at most $|E_t|$ times, yielding $\mathcal{O}(|E_t| \log |V_t|)$. The BFS operation in GETBESTNODE has $\mathcal{O}(|E_m|)$ cost, yielding an overall complexity of $\mathcal{O}(|V_t||E_m|)$. For a task $t_{best}$

and a candidate node at the last level of the BFS performed in GETBESTNODE, the cost of computing the change on WH is proportional to the number of edges of $t_{best}$ (the hop count between two arbitrary nodes can be found in $\mathcal{O}(1)$, since $G_m$'s are regular graphs). Since there are at most $|V_a| = |V_t|$ candidate nodes and $|E_t|$ edges, the complexity of this part throughout the algorithm is $\mathcal{O}(|V_t||E_t|)$. Therefore, the complexity of Algorithm 23 is $\mathcal{O}(|V_t|(|E_m| + |E_t|))$—in practice it runs faster thanks to the early exits in GETBESTNODE BFSs.

### 7.1.2 A Refinement algorithm for the weighted hop

Algorithm 23 is our main algorithm and the other two will refine its mapping. Even after its execution, it is possible to improve WH via further refinement. We have implemented a Kernighan-Lin [96] type algorithm which uses "task swaps" to refine WH (Algorithm 24). It gets a $\Gamma$, $G_t$, and $G_m$ as input and modifies $\Gamma$ to lower the WH metric. Similar to greedy mapping, for simplicity, Algorithm 24 assumes that $G_t$ is symmetric and $\Gamma$ is a one-to-one mapping between the tasks and nodes.

The algorithm selects a pair of task vertices and swaps them to improve WH. The first task $t_{wh}$ is chosen using a max-heap, $whHeap$, which initially organizes the tasks w.r.t. the WH amount they incur computed by a function TASKWHOPS function (line 1). Hence, $t_{wh}$ is the task individually responsible for the largest WH. Choosing the second task for the swap operation is more complicated; a naive approach that considers to swap $t_{wh}$ with all the other tasks requires $\mathcal{O}(|V_t|^2)$ comparisons. In order

**Algorithm 24:** WH Refinement

```
   Data: G_t = (V_t, E_t), G_m = (V_m, E_m), Γ, Δ
   ▶ compute the current WH for Γ
   WH ← CALCULATEWEIGHTEDHOPS(G_t, G_m, Γ)
   while WH is improved do
       ▶ compute WH incurred by each task
       ▶ place the tasks in a max-heap whHeap
1      for t in V_t do
           wh_t ← TASKWHOPS(t, G_t, G_m, Γ)
           whHeap.insert(t, wh_t)
2      while whHeap is not empty do
           t_wh ← whHeap.pop()
3          for the first Δ nodes m ∈ V_a visited in the order of the BFS from
           Γ[nghbor(t_wh)] do
               t ← the task mapped to m
4              if swapping t_wh and t improves WH then
                   Γ[t] ← Γ[t_wh]
                   Γ[t_wh] ← m
5                  Update whHeap for neighbors of t_wh
6                  Update whHeap for neighbors of t
                   break
```

to avoid this cost, we have implemented a BFS-based task-selection algorithm. A simple observation is that to reduce WH, $t_{wh}$ needs to move closer to its neighbor tasks. Therefore, we perform a BFS on $G_m$ starting from the nodes which have a neighbor of $t_{wh}$, i.e., the nodes in $\Gamma[nghbor(t_{wh})]$ (these are the level 0 nodes of BFS). Whenever a $V_a$ node with a the task $t$ is found, the WH value after the potential $\Gamma[t_{wh}] \leftrightarrow \Gamma[t]$ swap operation is computed. The actual swap is performed as soon as this computation reveals an improvement on WH. Since the likelihood of a WH improvement decreases when we go deeper on the BFS tree, we use an early exit mechanism to avoid a

full BFS traversal of $G_m$. Here we give an example to clarify the statement, assume $c(e) = 1\ \forall e \in E_t$. If the maximum hop count between $\Gamma[t_{wh}]$ and $\Gamma[nghbor(t_{wh})]$ is $d$ then when $t_{wh}$ is moved to a node after the BFS level $d$, WH incurred by $t_{wh}$ cannot be improved. Furthermore, when we go deeper in the BFS, $t_{wh}$'s incurred WH value will increase. Even in this case, the overall WH may still be improved due to the reduction of WH incurred by the second task $t$. However, this is less likely to happen considering $\Gamma[t]$ is handpicked for $t_{wh}$ but $\Gamma[t_{wh}]$ is only a random node for $t$. The early exit mechanism reduces the number of considered swap operations that are unlikely to improve WH. In Algorithm 24, a parameter $\Delta$ is used as an upper bound on this number. If $\Delta$ operations are checked for $t_{wh}$, the algorithm continues with the next $whHeap$ vertex. A refinement pass is completed when $whHeap$ is empty, and the next pass is performed only if there is an improvement in the previous pass.

The complexity of the loop at line 1 is $\mathcal{O}(|V_t| \log |V_t| + |E_t|)$. The loop at line 2 iterates $|V_t|$ times. The complexity of the BFS operation at line 3 is $\mathcal{O}(|E_m|)$ and it is also performed $|V_t|$ times. The complexity of the swap operation and the calculation of new WH at line 4 is proportional to the total number of edges of $t_{wh}$ and $t$ for each candidate node. Since there are at most $\Delta$ candidate nodes for each BFS, the complexity of line 4 for each pass becomes $\mathcal{O}(\Delta|E_t|)$. Lines 5 and 6 are executed at most once for each vertex and during a single pass, their total cost is $\mathcal{O}(|E_t| \log |V_t|)$. Therefore, the overall complexity of the algorithm becomes, $\mathcal{O}(|V_t| \log |V_t| + |E_t| + |E_m||V_t| + \Delta|E_t| + |E_t| \log |V_t|)$. The most dominant factor is the

complexity of the BFS operations which is $\mathcal{O}(|E_m||V_t|)$. Fortunately, the practical execution time is very low, since we stop after $\Delta$=8 swap candidates. We experimented with other exit mechanisms based on the maximum BFS level instead of the number of swap operations, and the preliminary experiments favored the approach described above. Furthermore, we observed that most of the improvement in WH is obtained after only a few passes. Hence, in order to be more efficient, we perform a pass only if WH is improved more than 0.5% in the previous one.

Similar to that of Algorithm 23, the description above assumes a one-to-one task-to-node mapping and performs the refinement on the node level, i.e., by swapping the vertices representing a group of tasks. With slight modifications, it can perform the refinement on the finer level task vertices or in a multilevel fashion from coarser to finer levels. In our experiments we choose to perform only on the coarser task graphs we obtained after METIS, since with WH-improving swap operations on the finer level, the total internode communication volume can also increase and the performance may decrease. Although this increase can also be tracked during the refinement, we do not want to sacrifice from the efficiency.

### 7.1.3 A Refinement algorithm for the maximum congestion

Although Algorithms 23 and 24 significantly improve WH, they can negatively affect MC or MMC, and this can degrade the performance especially for the bandwidth-bounded applications. Therefore, we propose another refinement algorithm (Algorithm 25), which improves the MC metric with minimal WH damage (adapting this

algorithm to refine MMC is trivial). The algorithm can accurately model and minimize MC for the interconnection networks with static routing. We will discuss the required enhancements for the dynamic-routing networks.

**Algorithm 25:** MC Refinement

```
   Data: G_t = (V_t, E_t), G_m = (V_m, E_m), Γ,  Δ
   ▶ calculate initial max and average congestions
   MC, AC ← calculateCongestion(G_t, G_m, Γ)
   ▶ initialize the link congestion heap
   ▶ store the tasks whose messages goes through links
   congHeap, commTasks ← INITCONG(G_t, G_m, Γ)
   while MC or AC is improved do
1      e_mc ← congHeap.pop()
2      for t_mc ∈ commTasks[e_mc] do
3          for the first Δ nodes m ∈ V_a visited in the order of the BFS from
           Γ[nghbor(t_mc)] do
               t ← the task mapped to m
               if swapping t_mc and t improves MC or AC then
                   Γ[t] ← Γ[t_wh]
                   Γ[t_wh] ← m
                   Update congHeap for t_mc and t edges
                   Update commTasks for t_mc and t edges
                   goto line 1
```

The algorithm gets a $\Gamma$, $G_m$, and $G_t$ and modifies $\Gamma$ to find mapping with a better congestion. First, it computes the initial congestion of $\Gamma$, and initializes the $congHeap$ using an INITCONG function. This max-heap stores the topology graph edges w.r.t. their congestion values. The algorithm also initializes $commTasks$, that is used to

query the tasks whose messages go through link $e$, i.e., $commTasks[e]$. Since, a message can go at most $D$ (network diameter) hops, the maximum size of $commTasks$ becomes $|E_t|D$, which is manageable since $D$ is not a large number.

After the initialization, the algorithm finds the most congested link $e_{mc}$. Then for each $t_{mc} \in commTasks[e_{mc}]$, the node to which $t_{mc}$ will be moved is sought via BFS traversals on $G_m$ starting from the nodes $\Gamma[nghbor(t_{mc})]$. The second task $t$ to swap is chosen from the tasks of the $V_a$ nodes traversed during the BFSs. This BFS order is important to have a minimal damage on WH. For each such candidate node, a virtual swap operation is performed to compute new MC and AC values. As soon as an improvement is detected, the actual swap operation is performed, and the execution continues with the next congested link. Whenever a vertex is moved, updates on the $congHeap$ and $commTasks$ are performed for all the incoming and outgoing edges of $t_{mc}$ and $t$ in $E_t$. If there is no improvement after $\Delta = 8$ trials, the early exit mechanism terminates the inner **for** loop. Then, the next task in $commTasks[e_{mc}]$ is chosen and the search restarts. If no improvement is found for the most congested link the algorithm stops. This algorithm can be applied both to coarser and finer task graphs. However, we only apply it on the coarser graph due to the reasons explained before.

With static routing, a message route can be found in $\mathcal{O}(D)$ time. The congestions of all the links can be calculated in $\mathcal{O}(D|E_t|)$ time, and the cost of initializing $congHeap$ is $\mathcal{O}(|E_m| \log |E_m|)$. A task insertion to a $commTasks$ set (implemented as a red-black binary tree using `std:set` in C++) can be done in $\mathcal{O}(\log |V_t|)$. Since

each message (an edge in $E_t$) can pass through at most $D$ links, the complexity of $commTasks$'s initialization is $\mathcal{O}(D|E_t|\log|V_t|)$. Therefore the initialization phase has a complexity of $\mathcal{O}(D|E_t|\log|V_t| + |E_m|\log|E_m|)$. A refinement pass starts at line 2. The main **for** loop iterates at most $|V_t|$ times and the complexity of a BFS at line 3 is $\mathcal{O}(|E_m|)$. Hence, the overall BFS complexity in a pass is $\mathcal{O}(|V_t||E_m|)$. For each candidate swap operation, we compute MC and AC by temporarily updating $congHeap$ where an update costs $\log|E_m|$ and a candidate swap requires at most $D$ updates for each of the $t_{mc}$ and $t$ edges. Since we consider at most $\Delta$ swap operations for a $t_{mc}$ which can be any vertex in $V_t$, the cost of MC and AC computation is $\mathcal{O}(E_t\ \Delta\ D\log|E_m|)$ for each $e_{mc}$. Once an improvement is found, the data structures $congHeap$ and $commTasks$ are updated and this happens only once per pass (a **while** loop iteration). Hence, the overall complexity of a pass is dominated by the BFS and MC–AC computations. Therefore the overall complexity of a pass becomes $\mathcal{O}((E_t\ \Delta\ D\log|E_m|) + (|V_t||E_m|))$.

Algorithm 25 accurately sees the maximum congestion on a static-routing network. For the networks with dynamic routing, an approximate refinement algorithm with a similar structure can be used. For example, the bandwidth on the Blue Gene/Q and Blue Gene/P can be maximized by placing the heavily communicating tasks to the diagonals of the torus [17, 19].

## 7.2 Experiments

In order to evaluate the quality of the mapping algorithms, we conducted various experiments on two irregular applications, an SpMV kernel and a synthetically generated application. The proposed methods are implemented in the UMPa framework. The $U_G$ and $U_{WH}$ variants minimize WH using Algorithms 23 and 24, and $U_{MC}$ and $U_{MMC}$ minimizes MC and MMC, respectively using Algorithm 25 (we do not give the results for TH variant as they are very close to those of $U_G$ and $U_{WH}$). These mapping methods are compared against the default MPI mapping (SMP-STYLE) in Hopper (DEF), the mapping provided by SCOTCH (SMAP, version 5.1.0 as the newer one does not support sparse allocations) [117], and the ones provided by LibTopoMap (TMAP) [89].

We selected 25 matrices from University of Florida (UFL) sparse matrix collection, belonging to 9 different classes. Table 7.1 gives the list of these matrices with their properties. We used 7 graph and hypergraph partitioners to partition these matrices: SCOTCH [117], KAFFPA (KAHIP) [124], METIS [93], PATOH [38], and UMPa [61]. A summary of the partitioning results are given in Section 7.2.1. MPI task communication graphs corresponding to these partitions are created and mapped to real processor allocations in Hopper. The analysis of the metrics and algorithm efficiency is presented in Section 7.2.2. Section 7.2.3 analyzes the impact of the mapping algorithms on the communication time, whereas Section 7.2.4 evaluates the performance

Table 7.1: The list of the matrices with their properties (e.g. matrix class and number of vertices and edges) used in the rest of the experiments. The matrix has ✓on the corresponding part number if partitioning algorithms are able to find a balanced partition whose imbalance ratio is below 0.05.

| **Matrix** | **Class** | **# vertices** | **# non zeros** | 1,024 | 2,048 | 4,096 | 8,192 | 16,384 |
|---|---|---|---|---|---|---|---|---|
| coAuthorsDBLP | Citation | 299,067 | 1,955,352 | ✓ | | | | |
| fl2010 | Redistricting | 484,481 | 2,346,294 | ✓ | ✓ | ✓ | | |
| delaunay_n19 | Delanuay | 524,288 | 3,145,646 | ✓ | ✓ | ✓ | ✓ | ✓ |
| ca2010 | Redistricting | 710,145 | 3,489,366 | ✓ | ✓ | ✓ | | |
| tx2010 | Redistricting | 914,231 | 4,456,272 | ✓ | ✓ | ✓ | | |
| netherlands.osm | Street | 2,216,688 | 4,882,476 | ✓ | ✓ | ✓ | ✓ | ✓ |
| G3_circuit | Sparse | 1,585,478 | 6,075,348 | ✓ | ✓ | ✓ | ✓ | ✓ |
| delaunay_n20 | Delanuay | 1,048,576 | 6,291,372 | ✓ | ✓ | ✓ | ✓ | ✓ |
| delaunay_n21 | Delanuay | 2,097,152 | 12,582,816 | ✓ | ✓ | ✓ | ✓ | ✓ |
| kkt_power | Sparse | 2,063,494 | 12,964,640 | ✓ | ✓ | ✓ | ✓ | ✓ |
| italy.osm | Street | 6,686,493 | 14,027,956 | ✓ | ✓ | ✓ | ✓ | ✓ |
| venturiLevel3 | NumericalSim | 4,026,819 | 16,108,474 | ✓ | ✓ | ✓ | ✓ | ✓ |
| great-britain.osm | Street | 7,733,822 | 16,313,034 | ✓ | ✓ | ✓ | ✓ | ✓ |
| hugetric-00020 | Frames | 7,122,792 | 21,361,554 | ✓ | ✓ | ✓ | ✓ | ✓ |
| rgg_n_2_21_s0 | RandomGeometric | 2,097,152 | 28,975,990 | ✓ | ✓ | ✓ | ✓ | ✓ |
| coPapersDBLP | Citation | 540,486 | 30,491,458 | ✓ | ✓ | | | |
| coPapersCiteseer | Citation | 434,102 | 32,073,440 | ✓ | ✓ | | | |
| road_central | Clustering | 14,081,816 | 33,866,826 | ✓ | ✓ | ✓ | ✓ | ✓ |
| packing-500x100x100-b050 | NumericalSim | 2,145,852 | 34,976,486 | ✓ | ✓ | ✓ | ✓ | ✓ |
| hugetrace-00020 | Frames | 16,002,413 | 47,997,626 | ✓ | ✓ | ✓ | ✓ | ✓ |
| hugebubbles-00010 | Frames | 19,458,087 | 58,359,528 | ✓ | ✓ | ✓ | ✓ | ✓ |
| rgg_n_2_22_s0 | RandomGeometric | 4,194,304 | 60,718,396 | ✓ | ✓ | ✓ | ✓ | ✓ |
| channel-500x100x100-b050 | NumericalSim | 4,802,000 | 85,362,744 | ✓ | ✓ | ✓ | ✓ | ✓ |
| cage15 | Sparse | 5,154,859 | 94,044,692 | ✓ | ✓ | ✓ | ✓ | ✓ |
| rgg_n_2_23_s0 | RandomGeometric | 8,388,608 | 127,002,786 | ✓ | ✓ | ✓ | ✓ | ✓ |

improvements for a Trilinos SpMV kernel [87]. We analyze the impact of the partitioning and mapping metrics on the parallel performance in Section 7.2.5.

### 7.2.1 Partitioning results

The matrices are first converted to a column-net hypergraph model, i.e., the rows represent the tasks with loads proportional to their number of non-zeros. The columns represent sets of data communications where each message has a unit communication

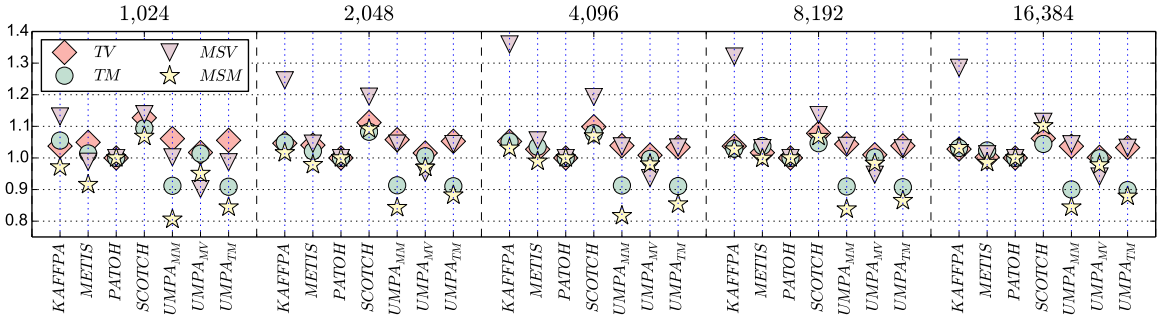



Figure 7.1: Geometric means of the partition metrics w.r.t PATOH for the corresponding part number.

costs. On these matrices we perform 1D row-wise partitioning for 1024, 2048, 4096, 8192 and 16384 parts (we only use 19 matrices for 16384 parts; balanced partitions were not feasible for the remaining 6). The graph partitioners, SCOTCH and KAFFPA, are run to minimize the edge-cut, and METIS and PATOH are run to minimize the total communication volume TV. Being a multi-objective partitioner, UMPa is used with different metrics: $\text{UMPA}_{\text{MV}}$ minimizing maximum send volume (MSV) and TV; $\text{UMPA}_{\text{MM}}$ minimizing maximum number of sent messages (MSM), total number of messages (TM) and TV; $\text{UMPA}_{\text{TM}}$ minimizing TM and TV; as their primary, secondary, and tertiary objectives, respectively [61]. All the partitioners are run with their default parameters.

Figure 7.1 shows the mean metric values normalized with that metric value of PATOH. Overall, all the tools obtain similar results, but edge-cut minimizing ones, SCOTCH and KAFFPA, obtain a slightly worse communication volume quality. For

the MSV metric, UMPA$_{MV}$ has the best results, e.g., it obtains a 5–10% better average MSV value w.r.t. PATOH which obtains the best results for TV. For the message metrics, UMPA$_{MM}$ obtained a 16–19% better MSM value, and UMPA$_{TM}$ obtained a 9–10% better TM value. These numbers are not given here to compare the partitioners since the experiment is not designed for that purpose. We want to better understand the impact of the partitioning and mapping metrics on the execution time.

### 7.2.2 Mappings on Hopper

Here we evaluate the mapping metric results on Hopper, which has a 3D Torus network and 24 processors per node. Even though the proposed algorithms do not have constraints on the number of processors, we tested them on numbers that are powers of two. Using all the processors in a node results in non-uniform processor allocations per node (since 24 does not divide 1024), in which case we experienced a few failing algorithms in LibTopoMap. Therefore, we used 16 processors per node (4 processors per NUMA domain).

We create directed task graphs by running all the partitioners on each matrix; for each graph and part number, we have 7 MPI task graphs. We will refer a task graph as $G_t^X$ when the part vector obtained from the partitioner X is used to create it. The mapping algorithms are then used to map these graphs to 5 different Hopper processor allocations. Figure 7.2 shows the average metric values of all mapping algorithms normalized to those of the default mapping on $G_t^{PATOH}$ graphs[3]. Almost all

[3]The results of the mapping algorithms on the results of other partitioners can be found at http://bmi.osu.edu/hpc/software/umpa/ipdps15/

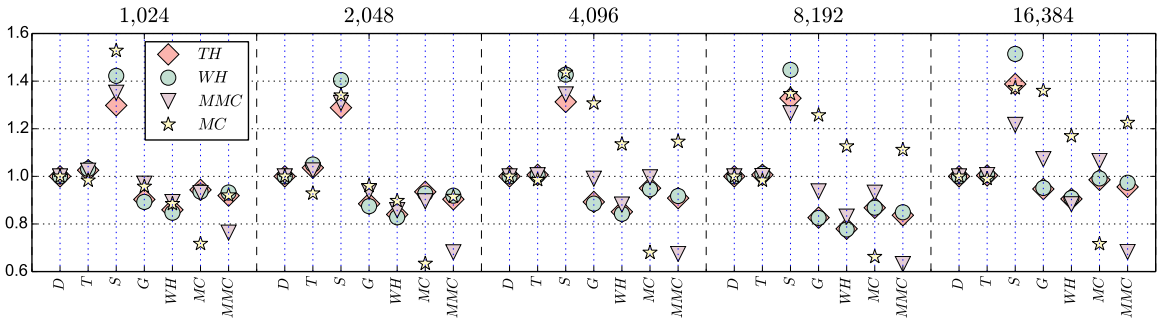



Figure 7.2: Mean metric values of the algorithms on $G_t^{\text{PATOH}}$ graphs normalized w.r.t. those of DEF. The numbers at the top denote the number of the processors, and the letters at the bottom correspond to the mapping algorithms DEF, TMAP, SMAP, $\text{U}_\text{G}$, $\text{U}_\text{WH}$, $\text{U}_\text{MC}$, $\text{U}_\text{MMC}$, respectively.

algorithms have their best WH and MC values on $G_t^{\text{PATOH}}$, and best TH and MMC values on $G_t^{\text{UMPA}_\text{TM}}$. The results are expected for WH and TH, since WH is closely related to the communication volume, and TH is related to total number of messages. On the other hand, it is expected to have better MC and MMC values on the task graphs with lower MSV and MSM values, respectively. However, in our experiments, we see a better correlation of these metrics with TV and TM.

In Fig. 7.2, the DEF mapping obtains already good results on WH and TH. This is due to the part ID assignment in recursive-bisection-based partitioners and the placement mechanism in Hopper: the partitioner puts highly communicating tasks to the parts with closer IDs. On the machine side, Hopper places the consecutive MPI ranks within a single node, then it moves to the closer nodes using space filling curves. Therefore, highly communicating consecutive MPI ranks are placed fairly

close to each other. However, there is still room for improvement when we exploit the actual task communication requirements. For example, $U_G$ obtains 5–18% and 5–17% better values on WH and TH, respectively.

Metric improvements on more sparse allocations (with less number of processors) are higher: $U_G$ significantly reduces WH and TH, and $U_{WH}$ improves them by another 4–5%. Also the variants that improve the WH metric also improve MC and MMC. For example, $U_G$ ($U_{WH}$) improves MC by 4% (10–12%) on 1024 and 2048 processors. However, when the number of processors is high, they increase the MC metric by 13–36% when the number of parts is high. Still, they reduce MMC, TH, and WH except $U_G$ on $16,384$ processors. Also, $U_{MC}$ significantly reduces (27–37%) the MC metric for all cases and have 1–13% improvement on WH and TH. Similarly, $U_{MMC}$ reduces MMC by 24–37% with small increases on TH and WH.

LibTopoMap provides six algorithms, and the best one in our experiments employs recursive graph bi-partitioning. Here we only present the best variant's performance (TMAP or T). The primary metric for LibTopoMap is MC. If TMAP's MC value is not smaller than the DEF mapping, it returns the DEF mapping. Overall, TMAP improves MC by 1–7% with 1–5% increase on the other metrics. On the other hand, SMAP's results are worse than DEF mappings for most of the cases.

Figure 7.3 presents the (geometric) mean mapping times of the algorithms. The times of the SMAP, $U_G$ and $U_{WH}$ are the lowest, and they are followed by $U_{MC}$ and

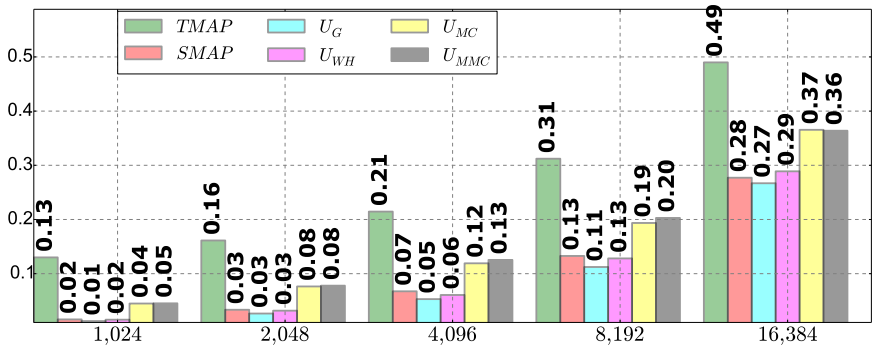
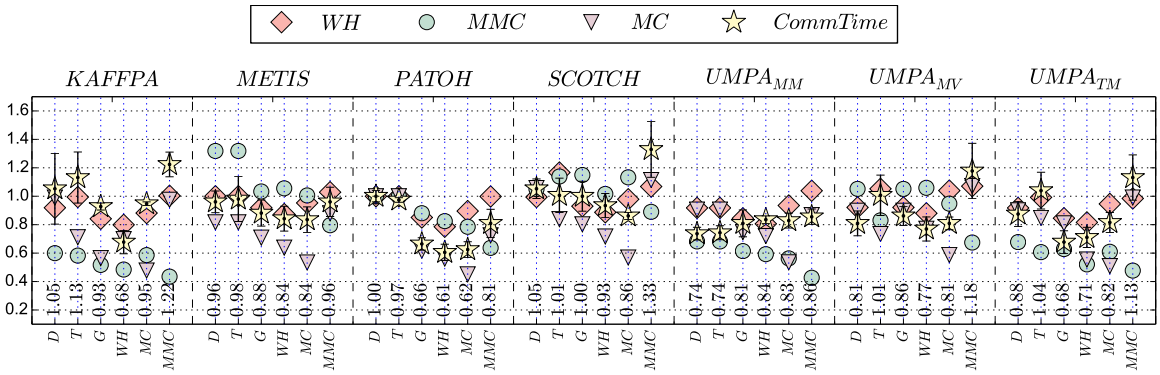



Figure 7.3: (Geometric) mean execution times of different mapping algorithms on PATOH partitions. The time of $U_{WH}$, $U_{MC}$, and $U_{MMC}$ includes $U_G$ time, as they run on top of it.

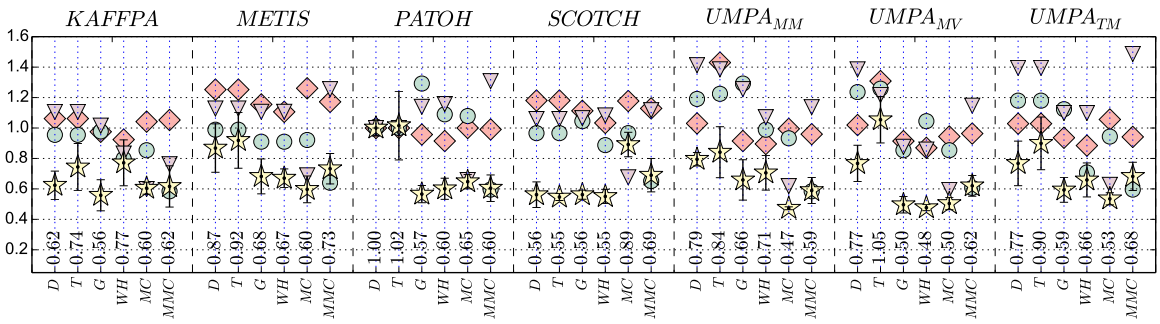
$U_{MMC}$. TMAP's execution time is more than the other methods, and it is 1.3–2.6 times slower than the slowest UMPa variant.

### 7.2.3 Communication-only experiments

In task mapping, the communication is usually modeled by assuming all the messages are transferred at once. However, this may not be the case in practice: load imbalance can delay some transfers, and applications might be using common techniques such as communication-computation overlap to hide the latency. Hence, improvements due to mapping may not be visible on an application's execution time. Here, to limit the impact of these factors, we generate irregular, communication-only applications based on the SpMV communication patterns of the two largest matrices in our dataset: `cage15` and `rgg_n_2_23_s0` (in short `rgg`). In this SpMV-like executions, no computation is performed, and all the transfers are initialized at the same time where




(a) `cage15`, #procs = 4096, scale = 4K




(b) `rgg`, #procs = 4096, scale = 256K

Figure 7.4: Average execution times and metrics for pure communication-based applications generated from `cage15` and `rgg`: the numbers at the bottom are the normalized execution times w.r.t. DEF mapping on $G_t^{\text{PATOH}}$. The partitioner names are given at the top, and the names at the bottom are the mapping algorithms, as given in Figure 7.2.

each processor follows the pattern in the corresponding communication graph. Therefore the total execution time of this application is equal to its communication time. To make the improvements more visible and reduce the noise, we scale the message sizes by using the factors $4K$ and $256K$ for `cage15` and `rgg`, respectively. The experiment is performed with 4096 processors. Each mapping algorithm is run with the 7 communication graphs (one per partitioner), and for each mapping, the execution is repeated 5 times to reduce the noise on the time. Figure 7.4 shows the normalized mean execution times with standard deviations and the metric values ( In real case, link congestions are also affected by the other running jobs. The reported congestion-metrics refer to those only incurred by the application.) normalized w.r.t. those of DEF mapping on $G_t^{\text{PATOH}}$. Although we run SMAP in this experiment, its communication time is worse than the others (we exclude SMAP from the figure for clarity). We do not report TH, as it is highly correlated with WH. Results with 8192 processors and a different sparse allocation can be found at http://bmi.osu.edu/hpc/software/umpa/ipdps15/.

Figure 7.4a shows the results for `cage15` communication graphs. The overall execution time correlates well with WH. In most cases, $\text{U}_\text{G}$ and $\text{U}_\text{WH}$ improve WH, MC, and the communication time w.r.t. DEF with a few exceptions. For example, on $G_t^{\text{UMPA}_\text{MM}}$, WH minimizing algorithms (Algorithms 23 and 24) improve all three metrics at the same time. However, the execution times with these mappings slightly increase. $\text{U}_\text{WH}$ obtains much better WH, MC, and execution times compared to $\text{U}_\text{G}$. On $G_t^{\text{KAFFPA}}$, it improves WH in the expense of increasing MC but the execution time

significantly reduces. Overall, $U_G$ and $U_{WH}$ improve the performance up to 34% and 39% w.r.t. the DEF. For all graphs, $U_{MC}$ obtains the best MC values and it usually improves the performance w.r.t. DEF. Moreover, it obtains the best execution time on $G_t^{SCOTCH}$, which has the highest TV w.r.t other partitioners on this graph. Among the UMPa algorithms, $U_{MMC}$ obtains the worst execution times although it always significantly reduces MMC. This is expected; since the message sizes are scaled, the executions have a high TV value and the volume-related metrics are likely to be the bottleneck rather than the message-related ones. TMAP can not improve the results of the DEF in some of cases, e.g., $G_t^{METIS}$, $G_t^{PATOH}$, and $G_t^{UMPA_{MM}}$, and returns the default mapping (the times vary 2–3% due to noise). Lastly, although DEF obtains the best mean execution time on UMPa graphs, overall, the best times are obtained on PATOH graphs with $U_{WH}$ and $U_{MC}$ with 39% and 38% improvement, respectively.

Figure 7.4b shows the results for `rgg` communication graphs. The proposed mapping algorithms improve the execution time for all the graphs except for $G_t^{SCOTCH}$. Similar to `cage15` experiments, the best performance is obtained by $U_G$, $U_{MC}$ and $U_{WH}$. The best execution time is achieved by $U_{MC}$ on $G_t^{UMPA_{MM}}$ with a 40% improvement w.r.t DEF mapping. TMAP obtains the same mappings with DEF on most of the graphs except $G_t^{UMPA_{MM}}$ and $G_t^{UMPA_{MV}}$. As the results for $G_t^{PATOH}$ show, the proposed algorithms improves the performance 35–43% for `rgg` experiments.

The execution time is improved better with the algorithms minimizing WH and then MC. The improvements achieved by $U_{MMC}$ is not as high as the others since for

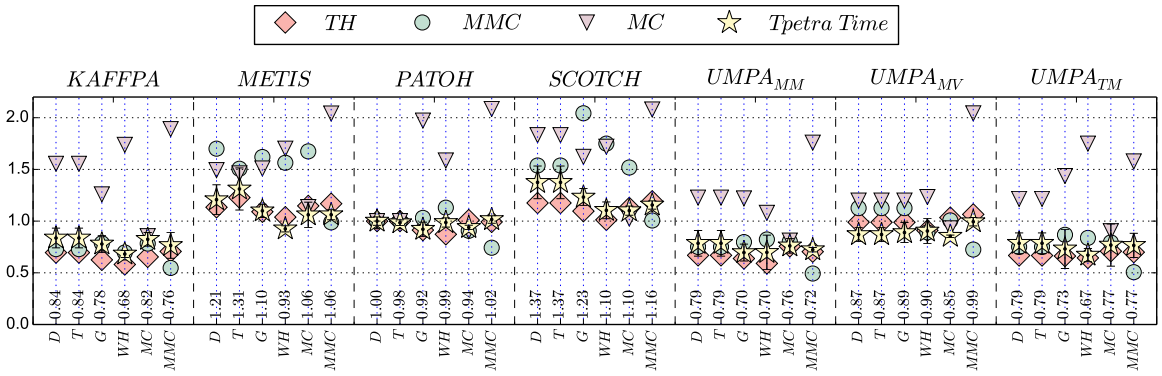



Figure 7.5: Trilinos SpMV results for `cage15` on 4096 processors. Each metric is normalized w.r.t that of DEF on $G_t^{\text{PATOH}}$.

these "*scaled*" applications, the volume metrics are likely to be the bottleneck. In Section 7.2.5, we perform a regression analysis to better analyze the relation between the metrics and the execution time.

### 7.2.4 SpMV experiments

In this section, we study the impact of the proposed algorithms on the SpMV performance. We use `cage15` and perform SpMV using the Tpetra package of Trilinos with 500 and 1000 iterations. Figure 7.5 shows the performance results, where each metric and the overall execution time is normalized w.r.t. that of DEF on $G_t^{\text{PATOH}}$. The experiment is run for 4096 and 8192 processors on two different allocations. Only the results of a single allocation on 4096 processors is shown while the rest of results can be found at `http://bmi.osu.edu/hpc/software/umpa/ipdps15/`. The SpMV

operation is repeated 5 times for each mapping and communication graph. We report the average of these 5 executions, and error bars represent the standard deviations. Unlike the previous experiment, TH is reported instead of WH, as its correlation with the total execution time is better.

In this setting, $U_{WH}$ obtains the best performance; it decreases the overall execution time almost always (except for $G_t^{UMPA_{MV}}$) and up to 23% (for $G_t^{METIS}$) w.r.t DEF. $U_G$ obtains a similar performance with slightly higher execution times. Although $U_{MC}$ improves the performance for many cases w.r.t DEF, its performance is not as competitive as in the previous experiment since the message sizes are much smaller. Similar to the communication-only experiment, $U_{MMC}$ obtains smaller improvements than the other UMPa variants. The overall performance of TMAP is very close to DEF, since it returns the DEF mapping for most of the cases.

Overall, TH highly correlates with the execution time. Moreover, this correlation also holds among different communication graphs. In Section 7.2.2, we already observed that TH is much lower on the graphs with a lower TM. Improving the TH metric via both partitioning (with the objective TM) and mapping significantly reduces the parallel SpMV time. The best TM values for `cage15` have been found by KAFFPA, $UMPA_{MM}$ and $UMPA_{TM}$ (see the supplementary page for a `cage15`-only version of Fig. 7.1) and as Fig. 7.5 shows, these are the best partitioners for the default mapping. $U_{WH}$ reduces the execution time by another 9–16% for the for these cases and obtains the best overall execution time for $G_t^{UMPA_{TM}}$. This is more than two times

faster than the slowest variant, which is obtained by the DEF on $G_t^{\text{SCOTCH}}$. It also has 34% lower TM and 44% lower TH value. This shows the importance both the partitioning and mapping on SpMV performance.

### 7.2.5 Regression analysis

To analyze the performance improvements obtained for the communication-only applications and SpMV kernel w.r.t. the partitioning and mapping metrics, we use a linear regression analysis technique and solve a nonnegative least squares problem (NNLS). In NNLS, given a variable matrix $\mathbf{V}$ and a vector $\mathbf{t}$, we want to find a dependency vector $\mathbf{d}$ which minimizes $\|\mathbf{Vd} - \mathbf{t}\|$ s.t. $\mathbf{d} \geq 0$. In our case, $\mathbf{V}$ has 14 columns: the partitioning metrics MSV, TV, MSM, TM; the mapping metrics WH, TH, MC, MMC, AC, AMC; inter-node communication volume (ICV), i.e., the total communication volume on the network excluding the intra-node communication (from TV); number of inter-node communication messages (ICM); the maximum receive volume of a node (MNRV); and the maximum number of messages received by a node (MNRM). A row $\mathbf{t}$ of $\mathbf{V}$ corresponds to an execution where the time is put to the corresponding entry of $\mathbf{t}$. To standardize each entry of $\mathbf{V}$ and make them equally important, each column of $\mathbf{V}$ is normalized by first subtracting the column mean from each column entry, and dividing them to the column standard deviation. We then use MATLABs *lsqnonneg* to solve NNLS. The coefficient $\mathbf{d}_i$ of the output shows the dependency of the execution time to the $i$th metric.

We perform linear regression on the communication-only experiments' results with `cage15` graphs, 4096 processors and two sparse Hopper allocations. The analysis distinguished three metrics with non-zero coefficients. The metric with the highest coefficient is WH, followed by MSV and MC (0.023, 0.020 and 0.20), whereas the message-based metrics are found not to highly correlate with the performance. This is expected since the communication is scaled and the volume metrics' importance are increased. The results show that from the mapping perspective, WH and MC are the most important metrics for the applications with a high communication volume, whereas from the partitioning perspective, it is likely to be MSV.

We used the same experimental setting (`cage15`, 4096 processors, two allocations) for the SpMV kernel which is more latency bounded than the communication-only counterpart since there is no scaling on the communication volume. The metrics with non-zero coefficients are found to be AMC, ICV, MMC, TH, and MNRV (0.109, 0.070, 0.051, 0.050, 0.040). Since AMC better correlates with the performance compared to TH, it can be a good practice to utilize the already used links while reducing TH. One weakness of the regression analysis is that when highly correlating metrics are given in $\mathbf{V}$, the analysis may return a positive coefficient for only one of them. In our case, the importance of MNRM, ICM, and TM is hidden by the regression analysis. We also computed pairwise Pearson correlation of the metrics and observed a high correlation ($\geq 0.92$) of these metrics with AMC.

Table 7.2: Average improvements of the mapping algorithms on communication-only applications and SpMV kernel that runs for 500 and 1000 iterations for the first and second allocations, respectively.

| | # procs | Rep. | DEF | TMAP | $U_G$ | $U_{WH}$ | $U_{MC}$ | $U_{MMC}$ |
|---|---|---|---|---|---|---|---|---|
| cage15 SpMV | 4096 | 1 | 1.44 sec. | 1.01 | 0.93 | 0.87 | 0.93 | 0.95 |
| | | 2 | 2.77 sec. | 1.00 | 0.91 | 0.89 | 0.94 | 0.95 |
| | 8192 | 1 | 1.25 sec. | 0.99 | 0.98 | 0.96 | 0.99 | 1.01 |
| | | 2 | 3.43 sec. | 1.01 | 0.99 | 0.94 | 1.01 | 1.04 |
| | | **Gmean** | **2.03 sec.** | **1.00** | **0.95** | **0.91** | **0.97** | **0.99** |
| cage15 Comm | 4096 | 1 | 0.28 sec. | 1.06 | 0.90 | 0.83 | 0.88 | 1.15 |
| | | 2 | 0.28 sec. | 1.06 | 0.88 | 0.82 | 0.88 | 1.18 |
| | 8192 | 1 | 0.19 sec. | 1.01 | 1.02 | 0.89 | 1.01 | 1.16 |
| | | 2 | 0.20 sec. | 1.00 | 0.95 | 0.89 | 0.99 | 1.18 |
| | | **Gmean** | **0.23 sec.** | **1.03** | **0.93** | **0.86** | **0.94** | **1.17** |
| rgg Comm | 4096 | 1 | 0.39 sec. | 1.11 | 0.77 | 0.83 | 0.79 | 0.85 |
| | | 2 | 0.33 sec. | 0.98 | 0.96 | 0.77 | 0.84 | 0.87 |
| | | **Gmean** | **0.36 sec.** | **1.05** | **0.86** | **0.80** | **0.81** | **0.86** |

### 7.2.6 Overall Improvements

Table 7.2 presents a summary of the improvements achieved by the mapping algorithms in our experiments. For each allocation and part number, we calculate the geometric mean of the execution times obtained with the mapping methods on all graphs. The table shows the geometric mean of the execution times for DEF, and the normalized time for the other algorithms. The average for all allocations and part numbers are given at the bottom of the table. Overall, $U_{WH}$ improves the `cage15` SpMV kernel time by 4–13%, whereas the improvements for the communication-only `cage15` and `rgg` applications are 14% and 20%, respectively, w.r.t. DEF.

## 7.3 Summary

We have proposed fast and high quality topology-aware task mapping methods that use graph models. We have compared the proposed methods with some other graph-based algorithms from the literature and with a default method used in Nersc's Hopper Supercomputer. The experiments showed that on a set of 25 matrices from the UFL collection, the proposed methods obtained high quality mappings in a very short time for the target system. The experiments with 4096 processors revealed significant improvements on the mapping metrics compared to the Hopper's default mapping. These improvements yield a 43% performance improvement on one case for a communication-only application and a 23% improvement on the SpMV performance. Overall, with 4096 and 8192 processors, the proposed algorithms improve the performance of the SpMV kernel and the communication-only applications by 9% and 14%, respectively. We also evaluated the metrics according to their correlation with the performance. For the applications with a large communication volume, our analysis revealed that the weighted hop metric is the most dominant one, and for those with smaller messages, the average message congestion is a good metric that correlates with the performance.

# Chapter 8: Conclusion, Future Plan and Open Problems

In this work, the two-phase mapping problem has been studied in the contexts of both geometric models and connectivity-based models (hypergraph models). First, the partitioning algorithms have been studied in the context of geometric models. A parallel multi-jagged geometric partitioning algorithm, MJ, which has the ability to concurrently partition the dataset into multiple parts along a single dimension, has been proposed. Moreover, intelligent decision heuristics for the data movement have been investigated and implemented in the proposed geometric partitioner. Various experiments on various real and synthetic datasets have been conducted, and the performance of the proposed algorithm has been compared with respect to Zoltan's RCB, a well used method in the literature. The experiments have shown that the proposed algorithm scaled fairly well up to $6,144$ processor. Moreover, thanks to the data movement decision heuristics, the proposed partitioning algorithm performed better than RCB in almost all experiments on the datasets with various initial distribution properties.

Secondly, the partitioning problem, the first phase of the mapping problem, has been investigated using hypergraph models. Unlike the previous studies that use general hypergraph models and minimizes the traditional communication volume, a new directed hypergraph model has been proposed, and algorithms for multi-objective hypergraph partitioning have been presented. The proposed model and the algorithms have been materialized under the software package UMPa. Experiments on several hypergraphs from various applications have been carried out and the improvements of the proposed model and algorithms have been shown up to 85%, 45%, 43% on maximum communication volume, total message and maximum message metrics, respectively. Then, methods to improve the running time of the hypergraph partitioning problem have been studied. First set of improvements has been performed by exploiting the multi-core architectures. A set of multi-core clustering algorithms have been proposed. The proposed algorithms have been implemented in PaToH, a state-of-art hypergraph partitioner. Experiments on large number of hypergraphs have shown that proposed methods improve the running time of PaToH by 1.85, where the ideal speedup was 2.05. Second set of improvements on hypergraph partitioning problem have been studied by performing data manipulations. A set of hypergraph sparsification algorithms have been investigated. The proposed sparsification techniques have been implemented under the software package UMPa, and the experiments have

shown the effectiveness of the sparsification methods on hypergraph partitioning problem. The proposed techniques have made UMPa up to 4.1 faster with a minimal 4% harm on the quality.

Thirdly, the second phase of the mapping problem has been investigated. A new topology-aware task mapping method that uses a geometric model has been proposed to assign communicating tasks to nearby processors. Furthermore, heuristics have been presented to further improve the quality of the mappings. The experiments have been carried out on 2 different regular applications, and they have shown that the proposed mapping method improved the total execution time up to 34% on 64K cores. Moreover, the correlation of the theoretical metrics with the actual performance have been analyzed, and a discussion of which metric is the most suitable has been carried out.

Fourthly, the task mapping problem has been extended using the graph models. Using the graph models, an accurate representation of the applications tasks mapping on the supercomputers processors has been presented. Using this model, 3 very efficient mapping algorithms that obtain high quality mappings have been proposed. The mapping methods have been shown to minimize the communication overhead metrics accurately. On the experiments on the irregular applications, the mapping methods have been shown to improve the total execution time of SpMV up to 23%, and the total execution time of the communication-only experiment upto 43%.

## 8.1 Future Plans

Although this work covers a significant portion of the mapping problem, further research is being planned to extend the existing work.

Firstly, a hybrid partitioning method that incorporates both geometric and connectivity information is planned to be studied. Although connectivity-based partitioning methods are expected to obtain better quality, it has been shown that this is not the case for regular datasets in a previous work [60]. Therefore, the hybrid partitioning methods are planned to be investigated in order to improve both speed and quality of the partitioning processes.

Secondly, the power/energy consumption is a major concern in current high performance systems and it appears to become more vital issue in near future. Therefore, as another future work, the current partitioning work is planned to be extended to account for the energy consumption of the parallel applications. That is, an energy-aware partitioning model and method are planned to be investigated. Currently, the load-balancing methods assume that the energy usage is reduced by improving execution times of parallel applications. However, this may not be accurate. For example, energy consumption varies during the communication and computation steps, or when processors become idle because of the imbalances in these steps. More accurate models and methods for energy-aware partitioning is planned to be studied.

Thirdly, the scalability of the connectivity-based partitioning methods will be investigated further. The parallelization of the coarsening and uncoarsening heuristics

are challenging, however those are crucial steps to improve the scalability. Hybrid parallel (MPI+OpenMP) connectivity-based partitioning heuristics are planned to be investigated to address this issue.

## 8.2 Open Problems

Apart from the future plans, there are some interesting open problems for researchers to extend the presented work. For example, current supercomputer architectures allow complex message routing algorithms (e.g., dynamic routing mechanisms). Some provides mechanisms to separate big messages into smaller chunks and send them through multiple routes. Further research can be performed on task mapping algorithms by taking such technologies into account. In addition, incremental algorithms have been a hot topic in high performance computing in the last decade. There exists some research performing incremental graph partitioning. However, as static graph partitioning methods, they do not accurately model the communication behavior, and only address a single metric. In order to overcome these issues, multi-objective methods for incremental hypergraph partitioning can significantly improve the current state of art.

# Bibliography


[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[2] K. Akbudak, E. Kayaaslan, and C. Aykanat. Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication. Technical report, Bilkent University, Dept. of Computer Engineering, 2012.

[3] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary. Topology-aware mappings for large-scale eigenvalue problems. In *Euro-Par 2012 Parallel Processing*, pages 830–842. Springer, 2012.

[4] C. Albing, N. Troullier, S. Whalen, R. Olson, and J. Glensk. Topology, bandwidth and performance: A new approach in linear orderings for application placement in a 3D torus. In *Proc Cray User Group (CUG)*, 2011.

[5] G. Almasi, S. Chatterjee, A. Gara, J. Gunnels, M. Gupta, A. Henning, J. Moreira, and B. Walkup. Unlocking the performance of the BlueGene/L supercomputer. In *Proc 2004 ACM/IEEE Conf Supercomputing*, page 57, 2004.

[6] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *VLSI Journal*, 19(1–2):1–81, 1995.

[7] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of AFIP'67*, pages 483–485. ACM, 1967.

[8] A. Appleby. SMHasher & MurmurHash, 2012. http://code.google.com/p/smhasher/.

[9] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Sci. Comput.*, 16(6):1404–1411, 1995.

[10] B. Aspvall, M. M. Halldórsson, and F. Manne. Approximations for the general block distribution of a matrix. *Theoretical Computer Science*, 262(1-2):145–160, 2001.

[11] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, May 2008.

[12] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen. Multithreaded algorithms for maximum matching in bipartite graphs. In *26th IPDPS*, pages 860–872. IEEE, 2012.

[13] S. T. Barnhard and H. D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):67–95, 1994.

[14] R. Barrett, C. Vaughan, S. Hammond, and D. Roweth. Reducing the Bulk of the Bulk Synchronous Parallel Model. *Parallel Process Lett*, 23(4), 2013.

[15] R. F. Barrett, C. T. Vaughan, and M. A. Heroux. MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical Report SAND2012-10431, Sandia National Laboratories, Albuquerque, NM, 2012.

[16] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Tran. on Computers*, C36(5):570–580, 1987.

[17] A. Bhatele, T. Gamblin, S. H. Langer, P. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still. Mapping applications with collectives over sub-communicators on torus networks. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012*. IEEE.

[18] A. Bhatele, G. Gupta, L. Kale, and I.-H. Chung. Automated mapping of regular communication graphs on mesh interconnects. In *Proc Intl Conf High Performance Computing (HiPC)*, 2010.

[19] A. Bhatele, N. Jain, K. E. Isaacs, R. Buch, T. Gamblin, S. H. Langer, and L. V. Kale. Optimizing the performance of parallel applications on a 5D torus via task mapping. In *IEEE International Conference on High Performance Computing*. IEEE Computer Society, Dec. 2014.

[20] A. Bhatele, L. V. Kale, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proc 23rd Intl Conf Supercomputing*, pages 110–116. ACM, 2009.

[21] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford, 2004.

[22] R. H. Bisseling, J. Byrka, S. Cerav-Erbas, N. Gvozdenović, M. Lorenz, R. Pendavingh, C. Reeves, M. Röger, and A. Verhoeven. Partitioning a call graph. In *Proceedings Study Group Mathematics with Industry 2005, Amsterdam*. CWI, Amsterdam, 2005.

[23] R. H. Bisseling and B. O. Fagginger Auer. Abusing a hypergraph partitioner for unweighted graph partitioning. In *10th DIMACS Implementation Challenge Workshop: Graph Partitioning and Graph Clustering*, pages 19–36, Feb 2012. Published in Contemporary Mathematics, Vol. 588, Editors D.A. Bader, H. Meyerhenke, P. Sanders, D. Wagner, 2013.

[24] R. H. Bisseling and I. Flesch. Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm–A case study. *Parallel Computing*, 32(7–8):551–567, 2006.

[25] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[26] S. H. Bokhari. On the mapping problem. *IEEE Trans Comput*, 100(3):207–214, 1981.

[27] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *ICPP (1)*, pages 1–7, 1988.

[28] S. W. Bollinger and S. F. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans Comput*, 40(3):325–333, 1991.

[29] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, Ü. V. Çatalyürek, D. Bozdağ, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W.

[30] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and U. Catalyurek. Zoltan2: Next-generation combinatorial toolkit. Technical Report SAND2012-9373C, Sandia National Laboratories, 2012.

[31] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proc. 13th ACM Symposium on Theory of Computing*, STOC '98, pages 327–336, 1998.

[32] K. Brown, S. Attaway, S. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Computer methods in applied mechanics and engineering*, 184(2):375–390, 2000.

[33] W. M. Brown, T. D. Nguyen, M. Fuentes-Cabrera, J. D. Fowlkes, P. D. Rack, M. Berger, and A. S. Bland. An evaluation of molecular dynamics performance on the hybrid Cray XK6 supercomputer. In *Proc Intl Conf Computational Science (ICCS)*, 2012.

[34] T. N. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[35] A. Caldwell, A. Kahng, and I. Markov. Improved algorithms for hypergraph bipartitioning. In *Proc. Design Automation Conference*, pages 661–666, june 2000.

[36] Ü. V. Çatalyürek and C. Aykanat. A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. In *Proc. International Conference on High Performance Computing*, Dec. 1995.

[37] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[38] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm, 1999.

[39] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings of 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, Apr 2001.

[40] Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *ACM/IEEE SC2001*, Denver, CO, Nov 2001.

[41] Ü. V. Çatalyürek, C. Aykanat, and E. Kayaaslan. Hypergraph partitioning-based fill-reducing ordering for symmetric matrices. *SIAM Journal on Scientific Computing*, 33(4):1996–2023, 2011.

[42] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 32(2):656–683, 2010.

[43] Ü. V. Çatalyürek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, Aug 2009.

[44] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. Multithreaded clustering for multi-level hypergraph partitioning. In *26th IPDPS*, pages 848–859. IEEE, 2012.

[45] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. UMPa: A multi-objective, multi-level partitioner for communication minimization. In *10th DIMACS Implementation Challenge Workshop: Graph Partitioning and Graph Clustering*, pages 53–66, Feb 2012. Published in Contemporary Mathematics, Vol. 588, Editors D.A. Bader, H. Meyerhenke, P. Sanders, D. Wagner, 2013.

[46] Ü. V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel algorithms for matching and coloring. In *2011 International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Parallel Computing and Optimization (PCO'11)*, pages 1966–1975, 2011.

[47] Ü. V. Çatalyürek, B. Uçar, and C. Aykanat. Hypergraph partitioning. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 871–881. Springer, 2011.

[48] C. Chevalier and I. Safro. Comparison of coarsening schemes for multilevel graph partitioning. In *Learning and Intelligent Optimization*, pages 191–205. Springer, 2009.

[49] T. Chockalingam and S. Arunkumar. Genetic algorithm based heuristics for the mapping problem. *Computers and Operations Research*, 22(1):55–64, 1995.

[50] I.-H. Chung, C.-R. Lee, J. Zhou, and Y.-C. Chung. Hierarchical mapping for HPC applications. In *Proc Workshop Large-Scale Parallel Processing*, pages 1810–1818, 2011.

[51] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[52] S. A. Cook. The complexity of theorem-proving procedures. In *Proc 3rd Annual ACM Symp Theory of Computing*, pages 151–158. ACM, 1971.

[53] N. Crane et al. Presto 4.16 user's guide. Technical Report SAND2010-3112, Sandia National Labs., 2010.

[54] N. Crane et al. SIERRA solid mechanics 4.22 user's guide. Technical Report SAND2011-7597, Sandia National Labs., 2011.

[55] T. A. Davis and Y. Hu. The Univerity of Florida collection. *ACM Trans. Math. Software*, 38(1), 2011.

[56] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of ACM*, 51(1):107–113, January 2008.

[57] M. Deveci, K. Kaya, and Ü. V. Çatalyürek. Hypergraph sparsification and its application to partitioning. In *Proc. of 42nd Int'l. Conf. on Parallel Processing*, Oct 2013.

[58] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. In *Proc. of 19th Int'l. Euro-Par Conf. on Parallel Processing*, August 2013.

[59] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek. A push-relabel-based maximum cardinality bipartite matching algorithm on gpus. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 21–29. IEEE, 2013.

[60] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek. Fast and high quality topology-aware task mapping. In *29th IEEE International Parallel and Distributed Processing Symposium*, May 2015. (to appear).

[61] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77:69 – 83, 2015.

[62] M. Deveci, S. Rajamanickam, K. D. Devine, and Ü. V. Çatalyürek. Multi-jagged: A scalable parallel spatial partitioning algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 2015. (to appear).

[63] M. Deveci, S. Rajamanickam, V. Leung, K. T. Pedretti, S. L. Olivier, D. P. Bunde, Ü. V. Çatalyürek, and K. D. Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *28th IEEE International Parallel and Distributed Processing Symposium*, May 2014.

[64] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[65] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[66] 10th DIMACS implementation challenge: Graph partitioning and graph clustering, 2011. http://www.cc.gatech.edu/dimacs10/.

[67] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *The First International Workshop on MapReduce and its Applications (MAPREDUCE'10) - HPDC2010*, pages 810–818, 2010.

[68] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, 10(1):35–44, 1990.

[69] B. Fagginger Auer and R. Bisseling. A GPU algorithm for greedy graph matching. *Facing the Multicore-Challenge II*, pages 108–119, 2012.

[70] R. Ferreira, T. Kurc, M. Beynon, C. Chang, A. Sussman, and J. Saltz. Object-relational queries into multi-dimensional databases with the Active Data Repository. *Parallel Processing Letters*, 9(2):173–195, 1999.

[71] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conference*, pages 175–181, 1982.

[72] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, April 1976.

[73] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. Directed hypergraphs and applications. *Discrete Appl. Math.*, 42(2–3):177–201, 1993.

[74] J. Garbers, H. J. Promel, and A. Steger. Finding clusters in vlsi circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 520–523. IEEE, 1990.

[75] E. G.Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[76] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Labs., 2006.

[77] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[78] M. Grigni and F. Manne. On the complexity of the generalized block distribution. In *IRREGULAR '96*, pages 319–326, 1996.

[79] L. Grigori, E. G. Boman, S. Donfack, and T. A. Davis. Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization. *SIAM J. Sci. Comput.*, 32(6):3426–3446, Nov. 2010.

[80] F. Gygi, E. W. Draeger, M. Schulz, B. de Supinski, J. Gunnels, V. Austel, J. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform. In *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.

[81] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *International Journal of High Performance Computing Applications*, 26(4):413–430, 2012.

[82] Y. Han, B. Narahari, and H.-A. Choi. Mapping a chain task to chained processors. *Information Processing Letter*, 44:141–148, 1992.

[83] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *J Roy Stat Soc C Appl Stat*, 28(1):100–108, 1979.

[84] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519 – 1534, 2000.

[85] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing*, page 28, New York, NY, USA, 1995. ACM.

[86] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, 1998.

[87] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[88] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, 2009.

[89] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc 25th Intl Conf Supercomputing*, pages 75–84. ACM, 2011.

[90] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[91] I. Kalashnikova, M. Perego, A. Salinger, R. Tuminaro, S. Price, and M. Hoffman. A new parallel, scalable and robust finite element first order stokes ice sheet dycore built for advanced analysis. *Geoscientific Model Development*, in preparation.

[92] H. Karimabadi, H. Vu, D. Krauss-Varban, and Y. Omelchenko. Global hybrid simulations of the earth's magnetosphere. In *Numerical Modeling of Space Plasma Flows*, volume 359, page 257, 2006.

[93] G. Karypis. *MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices version 5.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, 2011.

[94] G. Karypis and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, Dept. Computer Science, University of Minnesota, 1997.

[95] G. Karypis and V. Kumar. *hMeTiS: A hypergraph partitioning package*. Minneapolis, MN 55455, 1998.

[96] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb. 1970.

[97] H. Kikuchi, B. Karki, and S. Saini. Topology-aware parallel molecular dynamics simulation algorithm. In *Proc Intl Conf Parallel & Distributed Proc Tech & Applications*, 2006.

[98] H. Kutluca, C. Aykanat, et al. Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids. *The Journal of Supercomputing*, 15(1):51–93, 2000.

[99] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[100] S.-Y. Lee and J. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans Comput*, 100(4):433–442, 1987.

[101] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley–Teubner, Chichester, U.K., 1990.

[102] V. J. Leung, D. Bunde, J. Ebbers, S. Feer, N. Price, Z. Rhodes, and M. Swank. Task mapping stencil computations for non-contiguous allocations. In *Proc 19th Symp Principals & Practice of Parallel Prog (PPoPP)*. ACM SIGPLAN, 2014.

[103] D.-R. Liu and M.-Y. Wu. A hypergraph based approach to declustering problems. *Distributed and Parallel Databases*, 10:269–288, 2001.

[104] R. M. Loy. *Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws*. PhD thesis, Comp. Sci. Dept., Rensselaer Polytechnic Inst., 1998.

[105] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In R. Wyrzykowski, K. Karczewski, J. Dongarra, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag Berlin, 2008.

[106] F. Manne and T. Sørevik. Partitioning an array onto a mesh of processors. In *Proc of PARA '96*, pages 467–477, 1996.

[107] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[108] T. Minyard and Y. Kallinderis. Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations. *Int. J. Numer. Meth. Fluids*, 26:57–78, 1998.

[109] D. Nicol. Rectilinear partitioning of irregular data parallel computations. *J. of Parallel and Distributed Computing*, 23:119–134, 1994.

[110] M. M. Ozdal and C. Aykanat. Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery*, 9:29–57, 2004.

[111] J. A. Pascual, J. Miguel-Alonso, and J. A. Lozano. Optimization-based mapping framework for parallel applications. *Journal of Parallel and Distributed Computing*, 71(10):1377–1387, 2011.

[112] A. Patra and J. T. Oden. Problem decomposition for adaptive *hp* finite element methods. *J. Computing Systems in Engg.*, 6(2), 1995.

[113] M. M. A. Patwary, R. H. Bisseling, and F. Manne. Parallel greedy graph matching using an edge partitioning approach. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 45–54, New York, NY, USA, 2010. ACM.

[114] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and K. S. Hemmert. Using the Cray Gemini performance counters. In *Proc Cray User Group (CUG)*, 2013.

[115] F. Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 486–493. IEEE, 1994.

[116] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. In *TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I.* Citeseer, 1996.

[117] 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*, pages 493–498. Springer, 1996.

[118] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. CSE Technical Report CS94–349, Dept. Comp. Sci. & Eng., U. of California, 1994.

[119] A. Pınar and C. Aykanat. Sparse matrix decomposition with optimal load balancing. In *HiPC*, 1997.

[120] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *J. of Parallel and Distributed Computing*, 64:974–996, 2004.

[121] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comput Phys*, 117(1):1–19, 1995.

[122] S. J. Plimpton, D. B. Seidel, M. F. Pasik, R. S. Coats, and G. R. Montry. A load-balancing algorithm for a parallel electromagnetic particle-in-cell code. *Computer physics communications*, 152(3):227–241, 2003.

[123] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 72:1–72:11, 2012.

[124] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms–ESA 2011*, pages 469–480. Springer, 2011.

[125] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, pages 721–732, 2011.

[126] E. Ö. Baş, and Ü. V. Çatalyürek. Load-balancing spatially located computations using rectangular partitions. *J. of Parallel and Distributed Computing*, 72(10):1201 – 1214, 2012.

[127] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proc. 9th Design Automation Conference*, pages 57–62, 1972.

[128] S. Shekhar, C.-T. Lu, S. Chawla, and S. Ravada. Efficient join-index-based spatial-join processing: A clustering approach. *Knowledge and Data Engineering, IEEE Transactions on*, 14(6):1400–1421, nov/dec 2002.

[129] H. D. Simon. Partitioning of unstructured problems for parallel processing. In *Conf. on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergammon Press, 1991.

[130] A. Strehl and J. Ghosh. Cluster ensembles — a knowledge reuse framework for combining multiple partitions. *J. Mach. Learn. Res.*, 3:583–617, 2003.

[131] S. Tan, J. Bu, C. Chen, B. Xu, C. Wang, and X. He. Using rich social media information for music recommendation via hypergraph model. *ACM Trans. Multimedia Comput. Commun. Appl.*, 7S(1):22:1–22:22, 2011.

[132] V. E. Taylor and B. Nour-Omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *Int. J. Numer. Meth. Engng.*, 37:3809–3823, 1994.

[133] J. D. Teresco, J. Faik, and J. E. Flaherty. Hierarchical partitioning and dynamic load balancing for scientific computation. In J. W. Jack Dongarra, Kaj Madsen, editor, *Applied Parallel Computing. State of the Art in Scientific Computing: 7th International Conference, PARA 2004*, volume 3732 of *Lecture Notes in Computer Science*, pages 911–920, Lyngby, 2006. Springer. Previously published as Williams College Department of Computer Science Technical Report CS-04-04, 2004.

[134] A. Trifunovic and W. Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *Proc. ISCIS*, volume 3280 of *LNCS*, pages 789–800. Springer Berlin / Heidelberg, 2004.

[135] B. Uçar and C. Aykanat. Minimizing communication cost in fine-grain partitioning of sparse matrices. In A. Yazici and C. Şener, editors, *Computer and Information Sciences - ISCIS 2003*, volume 2869 of *Lecture Notes in Computer Science*, pages 926–933. Springer Berlin / Heidelberg, 2003.

[136] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Sci. Comput.*, 25:1837–1859, June 2004.

[137] B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review*, 49(4):595–603, 2007.

[138] B. Uçar, Ü. V. Çatalyürek, and C. Aykanat. A matrix partitioning interface to PaToH in Matlab. *Parallel Computing*, 36(5–6):254–272, 2010.

[139] M. Ujaldon, S. Sharma, E. Zapata, and J. Saltz. Experimental evaluation of efficient sparse matrix distributions. In *SuperComputing*, 1996.

[140] L. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, 1990.

[141] C. Vasconcelos and B. Rosenhahn. Bipartite graph matching computation on GPU. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 42–55. Springer, 2009.

[142] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[143] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future generation computer systems*, 17(5):601–623, 2001.

[144] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comp Syst*, 17(5):601–623, 2001.

[145] C. Walshaw, M. G. Everett, and M. Cross. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel Distributed Computing*, 47:102–108, December 1997.

[146] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing*, 1993.

[147] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar. On partitioning and reordering problems in a hierarchically parallel hybrid linear solver. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1391–1400. IEEE, 2013.

[148] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for Blue Gene/L supercomputer. In *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.

[149] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM J. Scientific Computing*, 31(4):3128–3154, 2009.