Self-Improving Sparse Matrix Partitioning and Bulk-Synchronous Pseudo-Streaming

MSc Thesis

Jan-Willem Buurlage

Scientific Computing Group
Mathematical Institute
Utrecht University

supervised by
Prof. Rob Bisseling

February 23, 2016

CONTENTS

PREFACE

This thesis is the result of work done at Utrecht University during the better part of one year in order to obtain a MSc degree in Mathematical Sciences, or more specifically in Scientific Computing. It describes a new method to minimize the runtime of parallel iterative solvers, and a generalization of the BSP model to a specific type of chip called a many-core coprocessor.

I would like to thank my supervisor prof. dr. Rob Bisseling for all his effort and support, and in particular for the enjoyable discussions we have had about this project. I would also like to thank my fellow students, in particular Abe, Erik, Peter, Tom and Lois, for all the equally enjoyable coffee breaks.

INTRODUCTION

One of the backbone operations in numerical mathematics is multiplying a sparse matrix with a vector. In computations involving very large linear systems, the sparse matrix-vector multiplication (SpMV) can be sped up tremendously by performing it in parallel. Effective parallelization of a SpMV requires an efficient distribution of the matrix A over processing elements. To this end the matrix A is partitioned. Common techniques for this partitioning rely on the so-called multi-level method.

For many applications the matrix A is reused a number of times, for example in an iterative solver. Current partitioners compute an efficient distribution of the matrix completely before it is used in an application. In particular, a partitioning is commonly computed before the first SpMV operation involving A is performed, and this partitioning is reused indefinitely. For some applications this may not be efficient, as the computational time spent partitioning A should always be less than the time saved by the parallelization of the SpMVs that rely on this partitioning. A more flexible approach would be to instead look at *iterative* partitioning methods, so that partitionings are refined over many iterations. This would allow for an efficient partitioning method for matrices that are used an indeterminate number of times.

In this research we will explore *self-improving partitioning*. We will modify current techniques to allow for a dynamic partitioning that self-balances the computational effort with the gain in efficiency by an improved partitioning. The major goal is to develop a flexible, iterative partitioning technique.

This thesis is organized as follows. In Chapter 1 we discuss numerical methods to solve linear systems, and some mathematical background of these *solvers* is given in Appendix A. In Chapter 2 we will discuss how to parallelize the SpMV operation, and in Chapter 3 and Chapter 4 we will discuss how to optimize the resulting algorithm by finding a good *data distribution*.

In the second part, consisting only of Chapter 5, we will discuss an extension of the BSP model to many-core coprocessors. This chapter can in principle be read separately from the other chapters, but if the reader is unfamiliar with the BSP model we suggest to first study Chapter 2.

As part of this research we have developed a *sparse matrix partitioning framework* called *Zee*. In Appendix B we provide details on how to use this framework, discuss its inner workings, and see how it compares to other available partitioning software.

Part I

SPARSE MATRIX PARTITIONING

1

SOLVING SYSTEMS OF LINEAR EQUATIONS

Solving a system of linear equations plays an important role in many fields of science. Mathematically these systems can be formulated as solving the system $Ax = b$ for a certain matrix A and output vector b. For many applications the system, and thus the corresponding matrix, is sparse. For example this is commonly the case for systems deriving from the discretization of differential equations. In the upcoming section we will give a short overview of common methods and also give a number of examples of problem areas which this research targets. The methods we discuss will serve as a motivation for our treatment of the optimization of parallel algorithms for sparse matrix vector (SpMV) multiplication, since this operation dominates the cost of approximating the solution of linear systems.

1.1 OPTIMIZING LINEAR SOLVERS

An important class of linear solvers are Krylov subspace methods. We introduce these methods, as well as two specific algorithms that are included in this class in Appendix A. For Krylov subspace methods such as CG and GMRES, we see that sparse matrix-vector multiplication is a crucial component of the algorithm. Indeed, the three important kernels of iterative methods are *vector updates, inner products, and matrix-vector products*, see also the discussion on page 181 of [22]. If we manage to keep the dimension of the Krylov subspace low, then the matrix-vector product will dominate the total cost of the algorithms. Optimizing the running time of these solvers then amounts to optimizing the running time of these products.

Much can then be gained by speeding up this operation, or in particular by parallelizing this operation. Furthermore, we note that in these algorithms the same matrix A is multiplied with many different vectors. The number of times the matrix A is used in this manner before convergence is hard to predict. This serves as the main motivation of our research. By spending a fixed amount of time preprocessing the matrix A, we can speed up the SpMV operations. Optimizing the time spent between speeding up the operations and actually performing them requires careful thought and

balance in the algorithm. Here we focus on iterative partitioning methods that accommodate this requirement.

Of course there are many other (iterative) linear solvers, e.g. those that apply to non-invertible and rectangular matrices. Furthermore, there are many variations and improvements of CG and GMRES that make the algorithms faster, more stable, or more general. Indeed, the two methods we have described only work for square matrices with specific characteristics. To solve systems with rectangular matrices A we can instead solve the equivalent system $A^TA\vec{x} = A^T\vec{b}$. CGLS is an example of a method that solves this equivalent system without actually forming the matrix product. Instead, it requires two matrix-vector products per iteration, one with A and one with A^T. All of these algorithms have one thing in common: they revolve around repeated (sparse) matrix vector multiplication, and this algorithm is thus an important *kernel operation* in these solvers.

Balancing the cumulative runtime of solvers and optimization techniques

Optimizing operations such as matrix-vector multiplication takes some initial effort, in order to obtain a long term gain in the running speed of the solver. The main question we treat in this part of the thesis is: "How can we balance the effort that is put into the *optimization* of the solver, with the actual gain these optimizations realize?". After introducing these optimization techniques themselves, we will discuss a possible answer to this question of balance in Chapter 4.

1.2 EXAMPLE: COMPUTED TOMOGRAPHY

As a real-world example of such a sparse linear system of equations, we consider Computerized Discrete Tomography. This technique is used in CT-scans to make detailed three-dimensional images of for example the human brain. In our (somewhat simplified) discussion here, we consider an object in some pre-determined region of space, and a number of rays that are sent through this region. For a more detailed exposition see e.g. [32].

We discretize the region into a $w \times l \times h$ grid of *voxels*, the three-dimensional equivalent of pixels. Mathematically this space can be viewed as $X = \mathbb{Z}_w \times \mathbb{Z}_l \times \mathbb{Z}_h$. A total number of m rays are sent through this space either parallel to each other (parallel beam tomography), or originating from a single point source (fan beam tomography). The intensity of the rays is measured on the other side of the space using a detector. This results in an m-dimensional vector $\vec{b}$, with one component for each ray, that stores the so-called *weighted line sum* of the rays through the space. This vector $\vec{b}$ is called the *projection vector*.

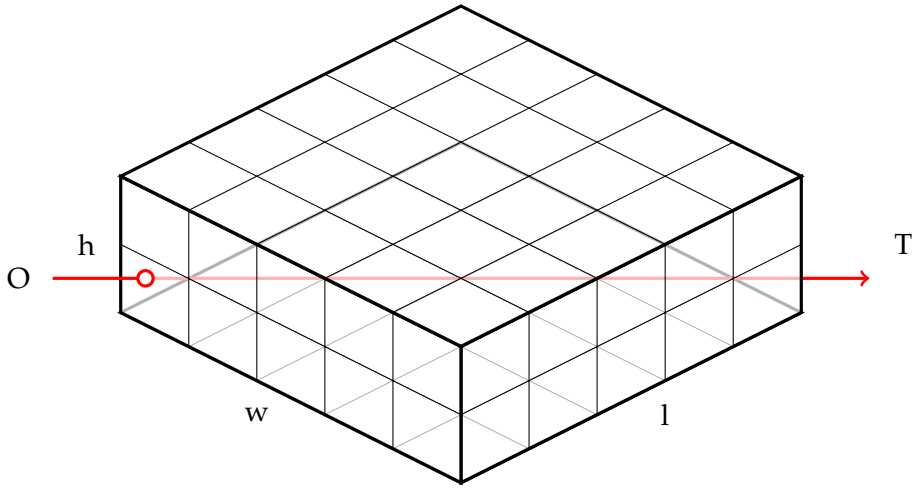


Figure 1.1.: Typical tomography setup. A ray from the origin (O) to a target (T) is sent through a 3D grid of voxels of size $w \times l \times h$. Between entering (marked with a circle) and leaving the grid the ray passses through a number of voxels.

To reconstruct the object within the space we have to solve a system of equations. Indeed, a ray passes through a fixed number of voxels, and if this voxel contains material that reduces the intensity of the ray, then the resulting component in $\vec{b}$ should reflect this. We can model this as a matrix A of size $m \times whl$, called the *projection matrix*, where we have a row for each ray, and a non-zero for each voxel that a ray passes through. We find the problem of the type $A\vec{x} = \vec{b}$, where $\vec{x}$ is the density profile of the object as seen by the rays within the space X – which contains exactly the information we require. See also Figure 1.1 for a schematic depiction of this setup.

Finally we note that this matrix will necessarily be very sparse. If we consider for example a cubic region of size $k \times k \times k$, then any straight line will go through at most $\mathcal{O}(k)$ voxels, which implies that the majority of the matrix elements will be zero. In applications this matrix is almost never stored explicitly, but is instead generated on demand [23].

1.3 EXAMPLE: GOOGLE'S PAGERANK ALGORITHM

The PageRank algorithm was introduced in 1998 as a way to measure the interest in webpages [38]. Here we discuss a simplified version of this idea. This will serve as a different kind of example that motivates the techniques we develop, because it does not take the form of a linear system of equations directly. It does however, revolve around the repeated application of a matrix-vector multiplication.

We can model the world wide web as a collection of webpages p_i which can reference (or *link to*) other webpages. The collection of all webpages can be considered as a vector $\vec{p}$. We define the matrix A with elements a_{ij} as:

$$a_{ij} = \begin{cases} 1 & \text{if } p_j \text{ links to } p_i \\ 0 & \text{otherwise.} \end{cases}$$

We can repeatedly apply A to the vector $e = (1,1,1,\ldots,1)^T$, i.e. we consider the vectors:

$$\vec{P}_n = A^n e.$$

These vectors can be considered as encoding the *importance* or *popularity* for each webpage. In the first iteration, webpages $\vec{p}_i$ that have many incoming references, will see their corresponding component $(\vec{P}_n)_i$ increased more than webpages with fewer incoming references. In subsequent iterations, pages that were deemed more important than others will count more strongly towards the importance of websites that they link to compared to less popular webpages that link to the same pages.

With appropriate modifications such as normalizing after iterations, and introducing damping terms, we can let this process converge to a unique vector $\vec{P}_n \to \vec{P}$, the components of which will each encode the *PageRank* of a webpage p_i.

SUMMARY

Solving systems of linear equations has many important applications in scientific computing. Therefore, much can be gained by optimizing the techniques that are used in this process. Instead of finding new methods and iterative techniques to solve these systems, we can also try to optimize existing solvers by looking at parallel algorithms. However, efficiently parallelizing the operations involved is not trivial and it takes some initial investment before they can be employed. The efficient parallelization of iterative solvers is exactly the subject of the first part of this thesis.

2

OPERATIONS WITH SPARSE MATRICES

In this chapter we will describe some basic operations with sparse matrices, and introduce some notation which we will use throughout this thesis. A sparse matrix $A \in \mathcal{M}_{m\times n}$ is a matrix with a low *density*. The density of a matrix is defined as its number of nonzeros divided by its size:

$$\rho = \frac{\mathrm{nz}(A)}{mn},$$

so that for a sparse matrix we have that $\rho \ll 1$. In this work we will focus primarily on multiplying a *sparse matrix* with a *dense vector*, or sparse matrix-vector multiplication (SpMV for short). The dense *in-vector* is denoted with $\vec{v} \in \mathbb{R}^n$, and the *out-vector* with $\vec{u} \in \mathbb{R}^m$. We are then interested in computing:

$$\vec{u} = A\vec{v}, \tag{2.1}$$

as efficiently as possible. For us, this will mean that we use a parallel algorithm to evaluate Equation 2.1. Before we discuss this algorithm we will first recall some facts and notions from parallel algorithms.

2.1 PARALLEL COMPUTING

Large-scale computations can be sped up greatly by using multiple processing units. *Parallel algorithms* have been studied and used for many years. Recent developments in technology have enabled both the industry as well as the scientific community to more easily scale up simulations, data processing etc. by targeting large clusters of computers. These methods are popularly referred to as *Big Data* technology, and are gaining widespread adoption.

2.1.1 *The BSP model*

A popular class of parallel algorithms are algorithms that are either directly or indirectly based on the bulk-synchronous paradigm. In the bulk-synchronous processing (BSP) model, introduced by Valiant in 1990 [47], the computer is assumed to have a collection of p identical processing units (or

processors for short) and a communication network so that these units can communicate. A BSP algorithm is structured in a number of supersteps. Each superstep consists of a *computation phase* and a *communication phase*. At the end of each step a barrier synchronization is performed between the cooperating processing units, so that the next superstep is initiated only after each unit has finished communication completely.

Each processor runs the same program, but on different data. This is referred to as SPMD for **S**ingle **P**rogram **M**ultiple **D**ata. Therefore an important aspect of a parallel algorithm is the data distribution. This distribution needs to be optimized in two different ways. First of all, we want to divide the complete set of data equally so that each processor roughly performs the same amount of work; this keeps them from wasteful idling. Secondly, we want to minimize the amount of communication between processors, since in realistic systems communication can be very time-consuming.

Example 1 (BSP algorithm for the inner product)**.** As an example of a BSP algorithm we will compute the inner product between two vectors:

$$\alpha = \vec{v} \cdot \vec{w} = \sum_{i=0}^{n-1} \vec{v}_i \vec{w}_i,$$

so that it is computed in parallel using the BSP model. First we must decide how we want to divide the data (i.e. the n components of both vectors) of the processors. As an example we will distribute them *cyclically*, so that the ith component of each vector will be sent to processor $s = i \mod p$.

The algorithm then consists of two supersteps:

1. Each processor s computes the partial sum:

$$\alpha_s = \sum_{\mathtt{i}} \vec{v}_{\mathtt{i}} \vec{w}_{\mathtt{i}},$$

   where `i` are the *local indices* of the vector as seen by the processor, i.e. the `i`th element that is stored on processor s. It then sends this partial sum to processor $s = 0$.

2. Processor $s = 0$ then sums over all the received partial sums to obtain the final value for the dot product:

$$\alpha = \sum_s \alpha_s.$$

To predict the gain of using a parallel algorithm, we need specific information on the actual hardware on which the program runs. We can however approximate the total running time of a parallel algorithm by introducing a number of parameters for an abstract *BSP computer*. This leads to the so-called BSP cost model.

Our basic unit of time, as is standard in the field of numerical computing, is the FLOP for **F**loating **P**oint **OP**eration. For an actual system this can be measured in seconds as the time necessary to carry out a single FLOP. This is captured in the BSP computing model as a parameter r, the FLOP rate, which is defined as the number of FLOPs that can be carried out every second.

Next we want to relate this *computation speed* to the cost of communication. We assume that there is a fixed cost of starting up communication and performing a barrier synchronization. This can be viewed as the *latency* of the system, and the number of equivalent FLOPs that could be performed in this time is called l. Finally we introduce the *communication-to-computation ratio* g, which is the number of FLOPs that could be performed while sending a single data word (in this discussion we will assume one floating point number is one word).

Another important concept in the BSP model is that of an h-relation. The deciding factor in the running speed of a parallel algorithm is not the total amount of communication or work done. Indeed, every operation that gets performed in a sequential algorithm must also be performed in the parallel version. What is important is the *maximum* amount of work and/or communication done by *any processor*. This motivates the following definition:

Definition 1. An h-relation is a superstep in which each processors sends and/or receives no more than h data words, and at least one processor sends and receives h data words. Thus we can write:

$$h = \max_{0 \leq s < p} \left(\max\{h_{s,\text{send}}, h_{s,\text{receive}}\} \right),$$

where $h_{\text{s, send/receive}}$ is the number of data words sent and received respectively by a processor s in a given superstep.

We can define a similar quantity for the maximum amount of work done in a superstep:

Definition 2. We define w as the *workload*, the maximum number of flops done by a processor in any given superstep.

The total cost T of the ith superstep can then be written as:

$$T(i) = w_i + gh_i + l$$

For the dot product we see that the running time in terms of flops is $2n$ in the sequential algorithm. Ignoring the cost of the initial distribution, and using the parameters introduced above we can show that in the parallel version this gets reduced to:

$$2\left\lceil \frac{n}{p} \right\rceil + 2p + pg + 2l.$$

Depending on the specific system, i.e. values for g, l and p, this can be a significant speed-up.

For an extensive introduction on the subject of parallel computing, and the BSP model in particular we refer to [6].

2.2 PARALLEL SPARSE MATRIX VECTOR MULTIPLICATION

In this section we will describe a parallel algorithm to evaluate the right-hand side of Equation 2.1. We assume that $A, \vec{v}$ and $\vec{u}$ are distributed. This means in particular that each of their nonzero entries are assigned to some processor. In the next chapter we will focus on developing techniques that compute and optimize this distribution. We assume we have p processors, and we capture the distribution using the following *assignment maps*:

$$\begin{aligned} P_A &: \mathbb{Z}_m \times \mathbb{Z}_n \to \mathbb{Z}_p, \\ P_{\vec{v}} &: \mathbb{Z}_n \to \mathbb{Z}_p, \\ P_{\vec{u}} &: \mathbb{Z}_m \to \mathbb{Z}_p. \end{aligned}$$

These maps are completely defined by the data distribution. We can view P_A and $P_{\{\vec{u},\vec{v}\}}$ as sending a pair of indices (i, j) or an index i respectively, to some processor $0 \leq s < p$. Note that we are only interested in the behaviour of P_A on the subset of $(i, j) \in \mathbb{Z}_m \times \mathbb{Z}_n$ such that $a_{ij} \neq 0$.

As we will see, the parallel SpMV algorithm consists of 4 phases. Let us identify these phases by working back from the end result. We want the correct value of u_i to be stored by $P_{\vec{u}}(i)$. This means we need to compute the value:

$$u_i = \sum_j A_{ij} v_j$$

on this processor. However, for general partitionings not all the entries A_{ij} and components v_j will be *local* to the processor $P_{\vec{u}}(i)$ for all j's. We therefore decompose this sum:

$$u_i = \sum_{s=0}^{p-1} (\vec{u}_i)_s,$$

where $(\vec{u}_i)_s$ is the *contribution* of processor s to u_i. If we define for a set A the function **accumulate**$(\oplus, A)$ as $a_1 \oplus \ldots \oplus a_n$ where $a_i \in A$, then this partial sum can be written as:

$$(\vec{u}_i)_s = \textbf{accumulate}(+, \{A_{ij} v_j \mid P(i, j) = s\}).$$

Here again, we note that v_j is not necessarily local to s. We solve this by requesting the assigned processor $P_{\vec{v}}(j)$ to send the value to the processor s. We are now ready to describe the algorithm in detail. We describe it in SPMD style. In particular, the algorithm is described as it should be executed

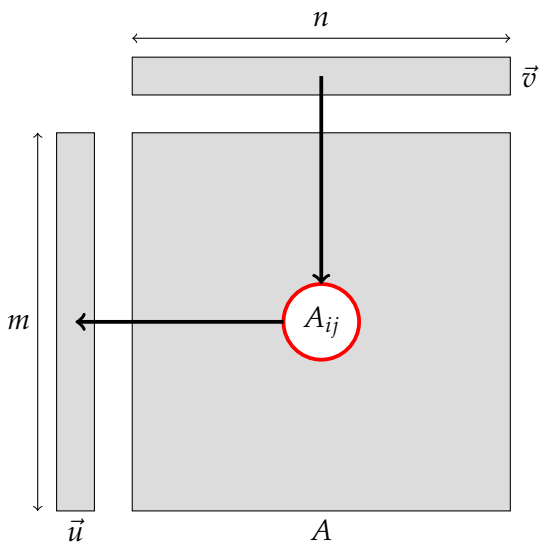


Figure 2.1.: Visualization of (parallel) SpMV. The location of a nonzero element A_{ij} is shown with a red circle. Vertical and horizontal communication are shown with a thick black arrow. For every non-zero element A_{ij} we require the vector component $\vec{v}_j$, which induces vertical communication, and we contribute to $\vec{u}_i$, inducing horizontal communication.

by an arbitrary processor s. See Figure 2.1 for a schematic depiction of the communication involved.

Phase I: Vertical communication. First we obtain v_j for each nonzero A_{ij} assigned to us:

for all j such that $P_A(i,j) = s$ for some i **do**
get v_j from $P_{\vec{v}}(j)$.

Afterwards we perform a barrier synchronization to ensure that all the data is available for the next phase. This phase is also called the *fan-out* phase.

Phase II: Compute contributions. We then compute the contributions $(u_i)_s$ on every processor.

for all i such that $P_A(i,j) = s$ for some j **do**
$u_i \leftarrow 0$
for all (i,j) such that $P_A(i,j) = s$ **do**
$u_i \leftarrow u_i + A_{ij}\vec{v}_j$

Phase III: Horizontal communication. Next we communicate the contributions $(u_i)_s$ to the assigned processor $P_{\vec{u}}(i)$.

for all i such that $P_A(i,j) = s$ for some j **do**
send u_i to $P_{\vec{u}}(i)$

We again perform a barrier synchronization before proceeding to the next phase. This is also called the *fan-in phase*.

Phase IV: Adding partial sums. Finally we compute the components u_i:

for all i such that $P_{\vec{u}}(i) = s$ **do**
$u_i \leftarrow 0$
for all t such that $P_A(i,j) = t$ for some j **do**
$u_i \leftarrow u_i + (u_i)_t$

such that the result $\vec{u}$ has been computed and is available at the processors according to the distribution $P_{\vec{u}}$.

We have skipped over specific implementation details here, which can prove to be hard to do right. Here we list some of the details that we have assumed or ignored:

- We assume that $P_{\vec{u}}$ and $P_{\vec{v}}$ is available at each processor. Note that storing the necessary information for these functions takes $\mathcal{O}(n)$ storage, which is something we want to avoid. In general, for parallel algorithms, we want both computation and communication to scale as $\mathcal{O}\left(\frac{n^k}{p}\right)$ for some k, such that the algorithm *scales* well with the number of processors.

- Efficient storage is required for the matrix elements, since we want to exploit the sparsity of the matrix. The most straightforward approach is to store only the triplets (i, j, a_{ij}), but more memory-efficient schemes have been developed

- We need to relate local indices `i` used for efficient computation to global indices i used for communication. Alternatively we need to relate local indices of processor s to indices of processors t that are the targets of communication.

- We have to find an efficient distribution $P_A, P_{\vec{v}}, P_{\vec{u}}$ that minimizes the total communication needed.

The first three of these points will not be discussed further in this thesis. Finding an efficient distribution will be the subject of Chapter 3.

for all j such that $P_A(i,j) = s$ for some i **do** ▷ (I)
  get v_j from $P_{\vec{v}}(j)$.

for all i such that $P_A(i,j) = s$ for some j **do** ▷ (II)
  $u_i \leftarrow 0$
for all (i,j) such that $P_A(i,j) = s$ **do**
  $u_i \leftarrow u_i + A_{ij}v_j$

for all i such that $P_A(i,j) = s$ for some j **do** ▷ (III)
  send u_i to $P_{\vec{u}}(i)$

for all i such that $P_{\vec{u}}(i) = s$ **do** ▷ (IV)
  $u_i \leftarrow 0$
  for all t such that $P_A(i,j) = t$ for some j **do**
    $u_i \leftarrow u_i + (u_i)_t$

Algorithm 2.1: A parallel algorithm that computes the SpMV between a matrix A and a vector $\vec{v}$ resulting in a vector $\vec{u}$. These objects are assumed to be distributed according to the assignment maps $P_{\{A,\vec{u},\vec{v}\}}$.

For reference we give the full SpMV algorithm that we developed in this chapter in Algorithm 2.1.

2.3 PREDICTING THE PERFORMANCE OF PARALLEL SPMV

In this section we analyze the (expected) performance of the algorithm described in Algorithm 2.1. The running time of the SpMV kernel is highly dependent on the distribution $P = P_{\{A,\vec{u},\vec{v}\}}$, because this directly influences the number of floating-point numbers that have to be communicated between processors.

Efficient parallelization of the SpMV operation therefore requires a good distribution of the matrix A over the processors. There is a delicate balance between the efficiency gain because of a lower computational load per processor (i.e. a lower number of FLOPs to be performed on a single processor), and the added cost because of the necessary communication and coordination between processors. We can more easily reason about this by

introducing some quantities that specifically relate to the distribution of a sparse matrix. This is the content of this section. Finally we will relate them to the BSP cost function introduced at the beginning of the chapter.

2.3.1 *Sparse matrix distributions*

We distinguish between one-dimensional and two-dimensional distributions P_A. A one-dimensional distribution depends on only one of the indices, e.g. if we can write $P_A(i,j) = \phi(i)$ we call the distribution a *row distribution*. A *column distribution* is defined similarly. An important class of distributions are *Cartesian distributions*, where we think about the processors as being laid out in a grid of size $M \times N$ such that $MN = p$. For a Cartesian distribution we have $P_A(i,j) = (\phi(i), \psi(j))$. Note that this limits the number of processors that have a nonzero in a row or column by M and N respectively.

In the standard SpMV algorithm, there are two phases in which communication happens, namely phase I and III. The total communication needed in the first phase (fan-out) for any processor s is the number of non-local vector components v_i that are required because processor s owns a nonzero a_{ki} for some k. The total communication needed in the second phase (fan-in) for any processor s is equal to the number of contributions from remote processors to every vector component u_j assigned to the processor s.

The vector distributions $P_{\{\vec{v},\vec{u}\}}$ do not influence the total number of communications done, as long as the vector components are assigned to a processor owning at least one nonzero in the corresponding row or column. We assume here that we can distribute $P_{\vec{v}}$ and $P_{\vec{u}}$ independently, which as we will see may not always be the case. Note also that under this assumptions we have that for row distributions the fan-out phase is free of communication, while for column distributions the fan-in phase is free of communication.

2.3.2 *Quality of a distribution*

As we have seen in our discussion on the BSP model, we distinguish between *sending*, and *receiving* data. Ultimately, in our model, the deciding factor for the communication in a parallel algorithm is the value for h in any superstep. Ideally we want to minimize the value of h for our two communication phases. However, we can also introduce the *total amount* of communication done, which is also an important measure of the quality of a distribution. As we will see, it will be easier to optimize distributions with respect to this measure.

Definition 3 (Communication Volume)**.** Given a partitioning $P = (P_A, P_{\vec{u}}, P_{\vec{v}})$, the communication volume is defined as the total number of floating point numbers to be communicated during a single SpMV operation.

For an arbitrary distribution we can write for the communication volume:

$$V = \sum_{i=0}^{m-1} (p_i - 1) + \sum_{j=0}^{n-1} (q_j - 1), \tag{2.2}$$

where we define p_i as the number of processors holding a nonzero in row i, and q_j as the number of processors holding a nonzero in column j. We subtract one from p_i and q_i, because we assume that the vector partitionings are such that the assigned processor for some vector component holds at least one nonzero in the corresponding row or column. Note that this volume, like all the other quantities we introduce in this section, only depends on the sparsity pattern of a matrix, and not the actual values.

The first term corresponds to the total communication done in the fan-out phase, while the second term corresponds to the total communication done in the fan-in phase. Note that we only count each number that gets communicated once (i.e. we consider sending and receiving independently).

The actual *communication cost* is the sum of the two values for h for the fan-out and fan-in phases. This value depends on both the vector distributions and the matrix distribution. We introduce I_s and J_s as the collection of rows and columns respectively, in which s holds at least one nonzero.

Using these sets we can write for the h values for each processor in the two phases:

$$\begin{aligned}
h_{\text{fan-out, } s\text{, send}} &= \sum_{j \text{ s.t. } P_{\vec{v}}(j)=s} \left(q_j - 1\right), \\
h_{\text{fan-out, } s\text{, receive}} &= |\{j \in J_s \mid P_{\vec{v}}(j) \neq s\}|, \\
h_{\text{fan-in, } s\text{, receive}} &= \sum_{i \text{ s.t. } P_{\vec{u}}(i)=s} \left(p_i - 1\right), \\
h_{\text{fan-in, } s\text{, send}} &= |\{i \in I_s \mid P_{\vec{u}}(i) \neq s\}|.
\end{aligned}$$

Taking the maximum value over the processors s then gives us the value for h of both communication phases.

Let us now consider the workload of any processor performing the SpMV algorithm. This is completely decided by the maximum number of nonzeros assigned to a processor s, and the number of different processors owning nonzeros in the same row.

Definition 4 (Workload)**.** Recall that the workload w is the maximum amount of computational work done by a processor. For the SpMV algorithm it is the sum over the maximum number of flops over all processors $0 \leq s < p$ of the two computational phases. This amounts to:

$$\begin{aligned}
W = &\max_{0 \leq s < p} |\{(i,j) \mid P_A(i,j) = s\}| \\
&+ \max_{0 \leq t < p} \sum_{i \text{ s.t. } P_{\vec{u}}(i)=t} |\{s \mid \exists_j P_A(i,j) = s\}|
\end{aligned}$$

Here the first term corresponds to the computation of the partial sums, and the second term corresponds to the accumulation of the partial sums. Note that the second term is usually much smaller than the first. Furthermore it is bounded by the communication volume, since each contribution has to be communicated over the network. Since communication is usually much more expensive than addition, i.e. $g \gg 1$ for common computer systems, it is commonly ignored [51]. Therefore we will simply write for the workload:

$$W = \max_{0 \leq s < p} |\{(i,j) \mid P_A(i,j) = s\}| \tag{2.3}$$

Note that our working definition of the workload, like the communication volume, does not depend on the specific vector distribution. For this reason we will focus mostly on optimizing the matrix distribution P_A.

2.4 SUMMARY

In this chapter we have introduced the standard SpMV algorithm, as well as the BSP framework in which we consider this algorithm. To optimize the running time of this parallel algorithm, we need to optimize the distribution of the matrix and the relevant vectors. To make this more concrete we have introduced concepts such as the *communication volume* and the *workload*, which will play an important role in our discussion on partitioning algorithms that will lead to good distributions.

3

PARTITIONING TECHNIQUES FOR SPARSE MATRICES

In this chapter we discuss finding a good distribution for a given sparse matrix A to optimize the running speed of the parallel SpMV algorithm. Various methods have been developed to tackle this particular problem. In general, finding the optimal partitioning is unfeasible, although attempts have been made to compute optimal distributions for small sparse matrices [40]. Therefore heuristic methods are applied instead that try to find good partitionings within reasonable time constraints. Many of them rely either directly or indirectly on underlying (hyper)graph models of the sparse matrix A. We will introduce some of these models and methods, and compare them in terms of partitioning cost and expected quality.

3.1 THEORY AND NOTIONS

We will first make concrete what problem we are trying to solve. We let A be some sparse matrix, which in general can be very large. The methods we treat in this research try to efficiently *minimize* the total communication volume, while assigning the nonzeros equally over the processors.

Instead of considering the distribution maps directly we can view the problem as partitioning A into p mutually disjoint submatrices:

$$A = A_0 \cup A_1 \cup \ldots \cup A_{p-1}, \qquad A_i \cap A_j = \emptyset.$$

From this viewpoint, the requirement of equal distribution can be made concrete by letting the matrix partitioning $A = \bigcup_s A_s$ satisfy a *load balance constraint*:

$$\mathrm{nz}(A_s) \leq (1+\varepsilon)\frac{\mathrm{nz}(A)}{p}, \qquad 1 \leq s \leq p, \tag{3.1}$$

where ε is the tolerance level of the *load imbalance*, which decides how strictly we want the nonzero assignment to happen equally over the processors. We have defined the *communication volume* as the total number of words that need to be communicated between two different processors. The problem we will treat in various forms in the upcoming chapters, the *sparse matrix partitioning problem*, is to minimize the communication volume while satisfying

Equation 3.1 for some fixed load imbalance ε. Under reasonable assumptions about the computer that performs the SpMV operation, a distribution with a low communication volume with a roughly equal distribution will lead to good running times of the SpMV algorithm.

3.1.1 *Partitioning a graph*

The sparse matrix partitioning problem is closely related to the partitioning problem on graphs. This is because a sparse matrix admits various graph models that can represent the structure of the sparse matrix. In this section we will introduce this similar problem, and in the upcoming sections we will relate these problems to sparse matrices.

Before we talk about graph representations of our sparse matrix, we will introduce the partitioning problem in the context of graphs. We will see later how we can apply the techniques that have been developed for graph partitioning to sparse matrix partitioning. To fix notation we first recall some basic concepts from graph theory, starting with the definition of a graph.

Definition 5 (Graphs)**.** A graph $G = (V, E)$ is a collection of *vertices* $V = \{v_i\}$ and a collection of ordered pairs called *edges* $E = \{(v_i, v_j)\}$ such that $v_i, v_j \in V$. If for $u, v \in V$ we have $(u, v) \in E \iff (v, u) \in E$ we call the graph *undirected*, otherwise we call the graph *directed*.

A graph is called *bipartite* if there exists a decomposition $V = L \cup R$ with $L \cap R = \emptyset$ such that every edge $e \in E$ is of the form $e = (l, r)$ with $l \in L$ and $r \in R$. Two vertices u and v are said to be *neighbours* if there exists an edge $E \ni e = (u, v)$. The *degree* of a vertex v is equal to the number of neighbours of v. We will also require the notion of a *weighted* graph.

Definition 6 (Weighted graph)**.** A weighted graph $G = (V, E, w)$ is a graph $G = (V, E)$ together with an *edge weight function* $w : E \to \mathbb{R}$. Sometimes we will also talk about *vertex weights*, in the form of a function $\omega : V \to \mathbb{R}$, in which case we will write $G = (V, E, w, \omega)$.

Because we want to apply graph partitioning methods to sparse matrix partitioning, we introduce graph partitionings.

Definition 7 (Graph partitioning)**.** A k-way partitioning of a graph $G = (V, E)$ is a partitioning of the vertices V into $k \geq 1$ disjoint subsets $V = V_0 \cup V_1 \cup \ldots \cup V_{k-1}$. Equivalently, we can represent this with a surjective function $\pi : V \to \{1, \ldots, k\}$. If $k = 2$ we call the partitioning a bipartitioning.

The corresponding load balance constraint for a weighted graph requires that the weighted sum over each of the subsets are roughly equal:

$$\sum_{v \in V_i} \omega(v) \leq (1 + \varepsilon) \frac{\sum_{u \in V} \omega(u)}{k} \qquad 0 \leq i < k. \tag{3.2}$$

We can also think of a partitioning as assigning a *label* (often represented with a colour) to each vertex. Indeed, every partitioning takes the form of a surjective function, which can be seen as assigning a label to each vertex. If we talk about a labeling of vertices we will use a function $C_V : V \to \mathbb{Z}_k$ and call it a k-labeling on the vertices of the graph. Similarly $C_E : E \to \mathbb{Z}_k$ denotes a k-labeling on the edges of the graph. This will prove to be a convenient way to think about graph partitionings.

We are now ready to state the partitioning problem for graphs. A quality metric Θ assigns a number to a partitioning representing the quality of the partitioning. For example, in the sparse matrix partitioning problem our specific metric was the communication volume. In the case of graphs we want to minimize $\Theta(\pi)$ for an arbitrary quality metric. Here π is some surjective function $\pi : V \to \mathbb{Z}_k$ that defines a partitioning. The graph partitioning problem can then be stated as minimizing $\Theta(\pi)$ while satisfying the graph load balancing constraint Equation 3.2. As an example of a metric, we will introduce the *edge cut metric* for graphs.

Definition 8 (Edge-cut)**.** Let $G = (V, E, w)$ be a graph, and Θ a quality metric. The edge cut of a partitioning π of G is the weighted sum over all edges between different parts:

$$\mathrm{EC}(\pi) = \sum_{\substack{e=(u,v) \\ \pi(u) \neq \pi(v)}} w(e).$$

We could now ask ourselves the question just how hard it is to solve the *graph partitioning problem* if we use the edge cut as a metric. It turns out that it is very unlikely that an optimal solution to this problem can be tackled efficiently, in fact it has been shown that for an arbitrary load imbalance the partitioning problem for unweighted graphs with $k = 2$ is NP-complete. The following theorem summarizes this:

Theorem 1 (Bui-Jones)**.** *Let $G = (V, E)$ be a graph. Let $\varepsilon \in \mathbb{Q}_{\geq 0}$ be a tolerance level for the load imbalance. Let $0 \leq b < |E|$ be some maximum number of edges cut. It is NP-complete to decide whether there exists a bipartitioning π of G that satisfies Equation 3.2, and such that $EC(\pi) = b$.*

Proof. Theorem 3.1 of Bui and Jones [11]. □

Hypergraphs

A hypergraph is the generalization of an (undirected) graph. As it will turn out, a hypergraph is able to model completely the communication volume resulting from a sparse matrix partitioning. Here we introduce these objects, and carry over our notions for graphs to these more general structures.

Definition 9 ((Weighted) hypergraphs)**.** A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is a collection of vertices $\mathcal{V}$, along with a set of *nets* (or *hyperedges*) $\mathcal{N}$ such that $n_i \in \mathcal{N}$ is a subset of $\mathcal{V}$. Two vertices $u \neq v \in \mathcal{V}$ are said to be *connected* or *neighbours*, if there exists a net n_i such that $u \in n_i$ and $v \in n_i$. Similarly to graphs, we often make use of weight functions in the form of *vertex weights* $\omega : \mathcal{V} \to \mathbb{R}$ and *net weights* $w : \mathcal{N} \to \mathbb{R}$.

Definition 10 (Hypergraph partitioning)**.** A partitioning of a hypergraph has the same definition as the partitioning of a graph, with G substituted by $\mathcal{H}$.

The hypergraph partitioning problem is also stated completely analogous to the graph partitioning problem. Instead of the edge cut metric, we define a similar cut-net metric:

Definition 11. The cut-net metric for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N}, w)$ with a k-way partitioning π, is the weighted sum over all nets which contain vertices of different parts:

$$\mathrm{CN}(\pi) = \sum_{\substack{n \in \mathcal{N} \\ \exists_{u,v \in n} \pi(u) \neq \pi(v)}} w(n).$$

An alternative metric that is often used in the context of sparse matrix partitioning is the $(\lambda - 1)$-metric.

Definition 12. We define $\lambda(n)$ to be the number of parts in which a net $n \in \mathcal{N}$ is divided: $\lambda(n) = |\{j \mid \exists_{v \in n} \pi(v) = j\}|$. The $(\lambda - 1)$-metric is then given by the following expression:

$$\mathrm{LV}(\pi) = \sum_{n \in \mathcal{N}} w(n)(\lambda(n) - 1).$$

For bipartitionings the LV and CN metrics are identical. Furthermore, they reduce to the edge cut metric in the case of graphs – such that we can immediately conclude that the hypergraph partitioning for both metrics with $k = 2$ is also NP-complete.

Coarsenings of (hyper)graphs

Since the (hyper)graph partitioning problems are NP-complete, it is often infeasible to find optimal partitionings for large graphs. We therefore employ two strategies to tackle this problem. The first one is to find good heuristics that run in polynomial time, and produce good partitionings. The other strategy is to *reduce the problem size* by coarsening the graph.

Definition 13 (Graph coarsening)**.** Let $G = (V, E)$ be a graph. A coarsening of the graph is a pair $(\chi, G' = (V', E'))$ such that $|V'| < |V|$ and:

- $\chi : V \to V'$ is a surjective function. This implies that V' is not larger than necessary.
- $E' = \{(\chi(u), \chi(v)) \mid (u, v) \in E\}$. This means the graph structure of G is maintained.

If G is edge- or vertex-weighted, then the corresponding weights of edges and vertices G' are equal to the sum over their pre-images.

Hypergraph coarsenings are defined similarly. The main difference is that the edges are replaced by nets. If the coarsening results in self-edges $(v, v) \in E'$ (or trivial nets $|n'| = 1$ in the case of hypergraphs) then we will remove them from the graph G'.

An important tool used to find good graph coarsenings χ, is *graph matching*.

Definition 14. A matching $M \subseteq E$ is a set of edges that are disjoint (i.e. each vertex is part of at most a single edge in M), and that contains no self-edges. A matching is *maximal* if there exists no matching M' of G such that $M \subsetneq M'$. It is a *perfect* matching if $|M| = \frac{|V|}{2}$. The weight of a matching is the weighted sum over its edges.

We will define a matching for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ as a matching for the graph G defined by $V = \mathcal{V}$ and $e = (u, v) \in E \iff \exists_{n \in \mathcal{N}}$ s.t. $u, v \in n$. We will revisit these ideas when we introduce the multi-level hypergraph partitioning approach.

3.1.2 *Modeling a sparse matrix as a (hyper)graph*

In this section we will describe the various graph structures that are used in the sparse matrix partitioning problem (in particular for optimizing SpMV). A major advantage of reducing the sparse matrix partitioning problem to that of a graph partitioning problem, is that partitioning methods (as well as developed software) can be reused for either application.

We will first look at natural (undirected) graph representations for a sparse matrix A. Recall that every graph G has an associated *adjacency matrix* of size $V \times V$, for which $a_{ij} \neq 0 \iff (v_i, v_j) \in E$. For symmetric sparse matrices $A = A^T$, we can then consider the unique graph that has the matrix A as its adjacency matrix. We will call this the *symmetric graph representation*.

One graph structure of particular interest to us is what we will call the *bipartite graph representation* of a sparse matrix.

Definition 15. Given a sparse matrix $A \in \mathcal{M}_{m \times n}$, we define the *bipartite graph representation* as the bipartite graph with vertices $\mathcal{V} = R \cup C$ where $r_i \in R$ represent the rows of A, and $c_j \in C$ represent the columns of A.

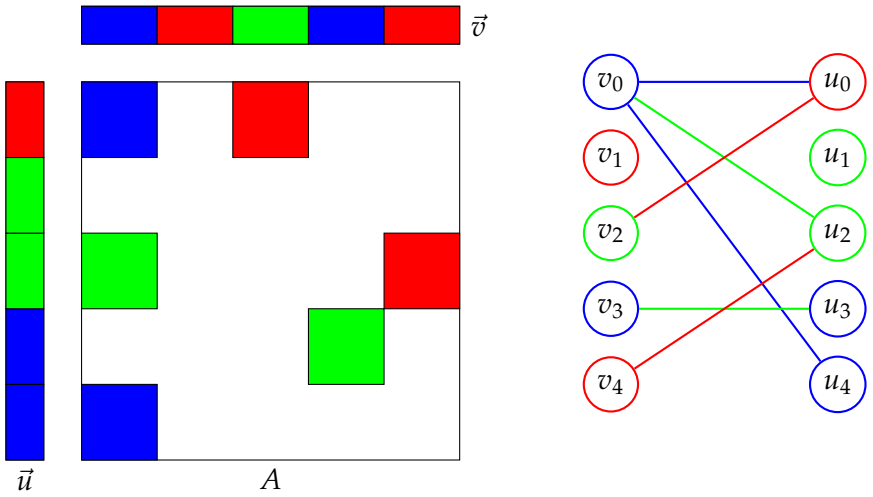


Figure 3.1.: The distribution of a sparse 5×5 matrix and the two vectors $\vec{v}, \vec{u}$ is represented by a colouring of the elements and components. The bipartite graph representation of this matrix is shown on the right, with the associated colouring. Each edge represents a contribution of a nonzero a_{ij} together with the component v_j, to the component u_i.

Alternatively these can be viewed as the components of the vectors $\vec{u}$ and $\vec{v}$ respectively. The edges of the graph are $E = \{(r_i, c_j) \mid i \in \mathcal{I}, j \in \mathcal{J}, a_{ij} \neq 0\}$.

For an example of a distributed sparse matrix and its associated bipartite graph representation, see Figure 3.1.

A labeling $\mathcal{C}_V : V \to \mathbb{Z}_p$ (i.e. partitioning) of the vertices of the bipartite graph representation corresponds to the vector distributions $P_{\vec{v}}$ and $P_{\vec{u}}$ by letting for example $P_{\vec{v}}(j) = \mathcal{C}_V(c_j)$. An edge labeling $\mathcal{C}_E : E \to \mathbb{Z}_p$ of this graph corresponds to a a distribution P_A of our sparse matrix. With this identification we can make the following observations:

- The number of processors that require the vector component $\vec{v}_i$ is equal to the number of distinct labels of the incident edges of the corresponding vertex.
- Similarly, the number of processors contributing to the vector component $\vec{u}_j$ is equal to the number of distinct labels of the incident edges of its corresponding vertex.
- If we label each vertex with the label of one of the incident edges, then the communication volume of the corresponding matrix distribution is equal to a $(\mu - 1)$-metric, where $\mu(v) : V \to \mathbb{Z}_p$ is equal to the number of distinct labels incident to a vertex v, i.e.:

$$\mu(v) = |\{j \mid \exists_{(v,u)\in E}\, \mathcal{C}_E((v,u)) = j\}|$$

(c.f. also the $(\lambda - 1)$ metric introduced earlier)

We conclude that we can define a metric on the bipartite graph, such that the graph *edge* partitioning problem with a corresponding balance constraint, reduces to the sparse matrix partitioning problem.

Hypergraph models

We have found a way to reduce our sparse matrix partitioning problem to a partitioning problem on the edges of a specific graph, but we ultimately want to reduce it to an ordinary graph partitioning problem. It turns out that hypergraphs are required in order to find such a reduction. Below we introduce three common hypergraph models of sparse matrices.

1. The *row-net model* was introduced by U. Catalyurek and C. Aykanat in 1996 [14]. In this model each column j is represented with a vertex $v_j \in \mathcal{V}$. For each row i we have a net n_i that is defined with:

   $$n_i = \{v_j \mid A \ni a_{ij} \neq 0\},$$

   i.e. a column is in the ith net if the corresponding matrix entry is nonzero.

2. The *column-net model* introduced in the same article is defined identically to the row-net model with the roles of the columns and rows reversed.

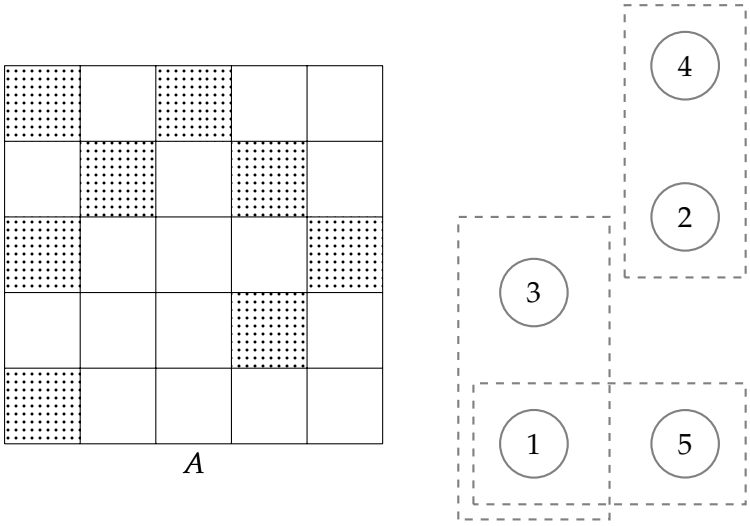
3. In the *fine-grain model* [16] each *nonzero matrix entry* is represented with a vertex $v_{ij} \in \mathcal{V}$. For each row i and each column j there is a net of all the nonzero entries in that row/column, i.e.:

   $$r_k = \{v_{ij} \in \mathcal{V} \mid i = k\},$$
   $$c_q = \{v_{ij} \in \mathcal{V} \mid j = q\}.$$

The various graph representations of a sparse matrix are summarized in Table 3.1.

The row net and column net models together with the $(\lambda - 1)$-metric for hypergraphs, model the communication volume of one-dimensional partitionings exactly, under the assumption that the vector distribution is such that the vector components are needed at least once by the assigned processor.

Indeed, consider without loss of generality a column-distribution, such that we can write $P_A(i,j) = P_A(j)$. Then the communication between processors is equal to the number of partial contributions $(\vec{u}_i)_s$ minus the number of rows, since for each i one of the partial contributions will already be at the



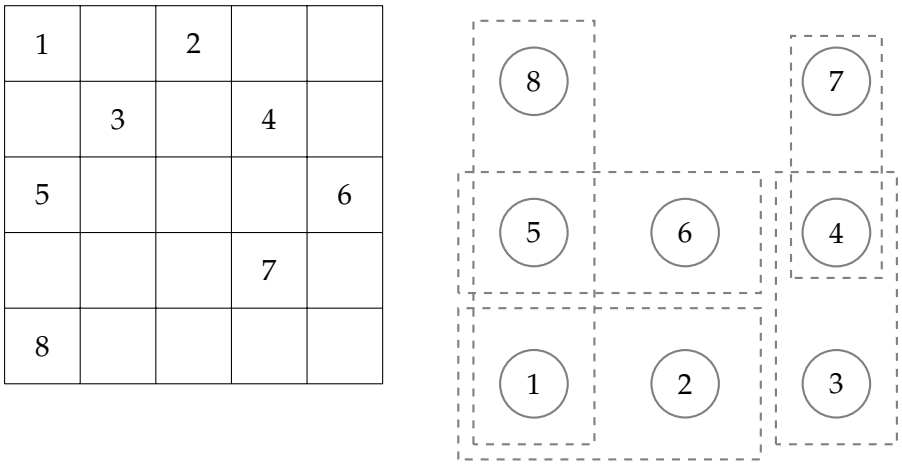


Figure 3.2.: A sparse 5×5 matrix and the corresponding hypergraphs using the row-net model (top) and the fine-grain model (bottom). Here we number the columns from 1 to 5, and the vertices in row-major order from 1 to 8.

name	*vertices* $\mathcal{V}$	*nets* $\mathcal{N}$
symmetric	$\{0 \leq i < m\}$	$\{(i,j) \mid a_{ij} \neq 0\}$
bipartite	$\{r_i\} \cup \{c_j\}$	$\{(i,j) \mid a_{ij} \neq 0\}$
row-net	$\{0 \leq j < n\}$	$\{n_i = \{j \mid a_{ij} = 0\}\}$
column-net	$\{0 \leq i < m\}$	$\{n_j = \{i \mid a_{ij} = 0\}\}$
fine-grain	$(v_{ij} \mid a_{ij} \neq 0)$	$\{\{a_{kj} \neq 0 \mid i = k\} \mid i\} \cup$ $\{\{a_{ik} \neq 0 \mid k = j\} \mid j\}$

Table 3.1.: The graph representations we consider of a sparse matrix A of size $m \times n$. See also [7].

correct location. Recall that the number of partial contributions to a given component of $\vec{u}$ is exactly equal to the number of distinct processors in each row, which we have called p_i. The communication volume then reduces to only the first term in Equation 2.2.:

$$V = \left(\sum_{i=0}^{m-1} |\{s \mid \exists_j \, a_{ij} \neq 0 \text{ and } P_A(j) = s\}| \right) - m \equiv \sum_{i=0}^{m-1} (p_i - 1) \,.$$

Let us now consider this system as a hypergraph using the row-net model, and assign each vertex corresponding to a column j to a part $P_A(j)$. The $(\lambda - 1)$-metric for this partitioning will yield a sum over

$$\begin{aligned} \mathrm{LV}(\pi) &= \sum_{i=0}^{m-1} \left(|\{k \mid \exists_{j \in n_i} \text{ s.t. } \pi(j) = k\}| - 1 \right) \\ &= \left(\sum_{i=0}^{m-1} |\{k \mid \exists_j a_{ij} \neq 0 \text{ and } \pi(j) = k\}| \right) - m, \end{aligned}$$

which is identical to the expression above by our choice of π.

For two-dimensional partitionings we have to resort to the fine-grain method, which will yield a similar correspondence. The communication volume for a general 2D distribution is given by Equation 2.2. Using the fine grain method, together with the vertex partitioning π that is defined by assigning $\pi(v_{ij}) = P_A(a_{ij})$ we see that the $(\lambda - 1)$-metric reduces exactly to the communication volume:

$$\begin{aligned} \mathrm{LV}(\pi) &= \sum_{n \in \mathcal{N}} w(n)(\lambda(n) - 1) \\ &= \sum_{i=0}^{m-1} (\lambda(r_i) - 1) + \sum_{j=0}^{n-1} (\lambda(c_j) - 1) \\ &= \sum_{i=0}^{m-1} (p_i - 1) + \sum_{j=0}^{n-1} (q_j - 1). \end{aligned}$$

We conclude that our sparse matrix partitioning problem can, by a good choice for a model, be reduced to a hypergraph partitioning problem.

3.1.3 *k-way matrix partitionings*

Before we end our theoretical discussion on graph partitioning, we will prove a convenient theorem which ensures that we only need to focus on bipartitioning methods. These can then be applied recursively to find general k-way partitionings. This theorem states essentially that *when splitting a part of the matrix, we need only consider the affected submatrix.*

Theorem 2 (Theorem 2.2 in [51])**.** *Let A be a sparse matrix, and suppose we have some k-way partitioning of $A = A_0 \cup A_1 \cup \ldots \cup A_{k-1}$. We denote the corresponding communication volume with:*

$$V(A, \{A_0, \ldots, A_{k-1}\}) = \sum_{i=0}^{n-1} (p_i(A, \{A_0, \ldots, A_{k-1}\}) - 1) + \sum_{j=0}^{n-1} (q_j(A, \{A_0, \ldots, A_{k-1}\}) - 1).$$

Let $A_{k-1} = A'_{k-1} \cup A_k$, such that $A'_{k-1} \cap A_k = \emptyset$ (i.e. a bipartitioning of the k-th part). Then we have the following equality:

$$V(A, \{A_0, \ldots, A'_{k-1}, A_k\}) = V(A, \{A_0, \ldots, A_{k-1}\}) + V(A_{k-1}, \{A'_{k-1}, A_k\})$$

Proof. It is sufficient to prove that this holds for each term separately, i.e. that the values for p_i and q_j in the left- and right-hand side are identical. Without loss of generality we treat the case of p_i.

We split into two cases: either the ith row in A_{k-1} is non-empty (i.e. it contributes to p_i) or it is empty. If it is empty, the equality immediately follows from:

$$p_i(A, \{A_0, \ldots, A_{k-1}\}) = p_i(A, \{A_0, \ldots, A_{k-2}\})$$

and

$$p_i(A_{k-1}, \{A'_{k-1}, A_k\}) = 0.$$

If it is non-empty then we know that the ith row in A'_{k-1} and A_k is non-empty for at least one of the two. If both parts contain an element in the ith row then we see:

$$\begin{aligned} p_i(A, \{A_0, \ldots, A'_{k-1}, A_k\}) &= p_i(A, \{A_0, \ldots, A_{k-2}\}) + 2 \\ &= p_i(A, \{A_0, \ldots, A_{k-1}\}) + 1 \\ &= p_i(A, \{A_0, \ldots, A_{k-1}\}) + p_i(A_{k-1}, \{A'_{k-1}, A_k\}) \end{aligned}$$

If only one of the two has a non-empty row i then we find:

$$\begin{aligned} p_i(A, \{A_0, \ldots, A'_{k-1}, A_k\}) &= p_i(A, \{A_0, \ldots, A_{k-2}\}) + 1 \\ &= p_i(A, \{A_0, \ldots, A_{k-1}\}) + 0 \\ &= p_i(A, \{A_0, \ldots, A_{k-1}\}) + p_i(A_{k-1}, \{A'_{k-1}, A_k\}) \end{aligned}$$

Thus we can conclude that the equality holds in each case. □

In light of this theorem, we will only discuss *bipartitioning* methods. These can be applied recursively to obtain general k-way partitionings by setting appropriate load balance constraints in the recursion steps.

3.1.4 *Vector partitioning*

Before we discuss specific partitioning methods, we need to consider the *vector partitionings*. In our derivation of the communication volume we have assumed that a vector component gets sent to a processor that holds at least one nonzero in the corresponding row or column. This requires in particular that we can distribute $\vec{u}$ and $\vec{v}$ independently.

However, in practice, we often require $P_{\vec{u}} = P_{\vec{v}}$. In linear solvers for example, the output vector is often the input vector in the following iteration. If these distributions are completely independent then we require a large amount of communication between different iteration steps to prepare the resulting vector $\vec{u}_i$ as the next input vector $\vec{v}_{i+1}$. This requirement complicates the vector partitioning. Indeed, sometimes the communication volumes we find can not be realized under these constraints.

We define $P(i, *)$ as the collection of processors holding a nonzero in row i and $P(*, j)$ as the collection of processors holding a nonzero in column j. If the diagonal element a_{kk} is nonzero, then we know that $P(k, *) \cap P(*, k) \neq \emptyset$ such that we can assign $P_{\vec{v}}(k) = P_{\vec{u}}(k) = P_A(k, k)$, but in general the intersection might as well be empty. If the collection of processors in the row or column k are mutually disjoint, then we can not realize the communication volume under this constraint.

In this research, where in our experiments we require that the input and output distributions are identical, we will apply the following heuristic to decide $P_{\vec{v}}(k)$. If $a_{kk} \neq 0$ it is natural to assign the vector component to $P_A(k, k)$. Otherwise we assign it to some some processor in the intersection of $P(k, *)$ and $P(*, k)$, while trying to balance the number of vector components assigned to the processors. Otherwise we assign it to some processor in $P(k, *) \cup P(*, k)$, also while greedily balancing the vector components assignment. Computing good matrix and vector distributions specifically under the constraint $P_{\vec{v}} = P_{\vec{u}}$ is left for further research.

3.2 METHODS

3.2.1 *Kernighan-Lin*

The Kernighan-Lin heuristic (KL) is a procedure for finding a locally optimal partitioning of a graph [34]. It is used extensively in methods that target the sparse matrix partitioning problem. Here we recite the basic ideas of the algorithm. The KL algorithm finds a partitioning of a graph G, while attempting to minimize the total edge cut, with constraints on the maximum size of subsets. We first specify more precisely the problem that we want to solve.

Given a weighted, directive graph $G = (V, E, c)$, we want to find p subsets $V = \bigcup_{i=0}^{p-1} V_i$ (i.e. a partitioning π of V) such that $|V_i| \leq M$ for some given $M > 0$, that minimizes the total edge cut. For a fixed M, we can find an appropriate load imbalance ε, and vice versa, so we can view this problem as the graph partitioning problem introduced before.

Considering all possible p-way partitioning of V is not an option, even for a relatively small number of vertices. Indeed, if $M = \frac{n}{p}$, we have $\binom{n}{n/p}$ options for V_0, $\binom{n-n/p}{n/p}$ options for V_1 etc, so that there are:

$$\frac{1}{p!} \prod_{k=0}^{p-1} \binom{n - kn/p}{n/p}$$

options in total, which grows very fast as n increases.

As we have seen, this problem is in NP, so that we consider heuristics instead. One may attempt to use λ-Opting techniques. In this context, a λ-change is the (best possible) exchange of λ vertices between different sets V_i for any current state of a graph partitioning. A partitioning is said to be λ-opt(imal) if there is no λ-change possible that reduces the edge cut. However, we note that for $\lambda = 1$ and $p = 2$ we need to consider every possible swap, and this already leads to an $\mathcal{O}(n^3)$ algorithm, so that we need to look for even more efficient heuristics.

The KL heuristic avoids having to try each possible swap, instead it associates to every vertex an internal and external cost. First we consider 2-way partitioning of V into $V = A \cup B$. Furthermore we assume $|V| = 2n$ and put $M = |A| = |B| = n$.

As we will show afterwards, the technique we develop can be extended to p-way partitionings, and to arbitrary values for M. For an $a \in A$ we define the external cost as the total weight of edges going from a to B:

$$E_a = \sum_{b \in B} c_{ab},$$

and we define E_b similarly. We also introduce the internal cost as the sum of the weights of edges in A:

$$I_a = \sum_{a' \in A} c_{aa'},$$

and denote their difference with $D_a = E_a - I_a$. If we then interchange $\tilde{a} \in A$ and $\tilde{b} \in B$, i.e. our new partitioning becomes $\tilde{A} = A - \{\tilde{a}\} + \{\tilde{b}\}$ and $\tilde{B} = V \setminus \tilde{A}$ then the reduction from the old edge cut T to the new edge cut $\tilde{T}$ is equal to:

$$\begin{aligned} g_{\tilde{a}\tilde{b}} \equiv T - \tilde{T} &= \sum_{a \in A} E_a + \sum_{b \in B} E_b - \sum_{a \in \tilde{A}} E_a - \sum_{b \in \tilde{B}} E_b \\ &= D_{\tilde{a}} + D_{\tilde{b}} - 2c_{\tilde{a}\tilde{b}}. \end{aligned}$$

With these definitions in place, we are now ready to describe the algorithm. In the first phase we compute all the value D_v for each $v \in A \cup B = V$. We then find the $a_1 \in A, b_1 \in B.$ such that g_{ab} is maximal:

$$g_1 = \max_{a \in A, b \in B} g_{ab} = \max_{a \in A, b \in B} D_a + D_b - 2c_{ab}.$$

We will call the pair that maximizes the gain (a_1, b_1). We then remove these from A, B and repeat the procedure for $V = A \setminus \{a_1\} \cup B \setminus \{b_1\}$, finding pairs (a_2, b_2). We continue until we have found n pairs of vertices which we can exchange. We then choose the value of $1 \leq k \leq n$ that maximizes the total gain:

$$G = \sum_{i=0}^{k} g_i,$$

and interchange $\{a_1, \ldots, a_k\}$ and $\{b_1, \ldots, b_k\}$ between A and B. If $G = 0$ we have found a locally optimal partitioning.

Partitioning with general values for M

We want to relax the constraint $M = |A| = |B| = n$. We consider graphs of arbitrary size $|V|$, and let $M > 0$ be arbitrary. We can generalize the procedure described above for this problem by adding additional vertices d to the graph that have no connections whatsoever, i.e. $c_{dv} = 0$ for all $v \in V$ until the graph has $2M$ elements. Note that $2M - |V|$ elements have to be added. These vertices are referred to as *dummy vertices*. If we apply the procedure above to this system, then these dummy vertices will have no influence on the different costs. Both A and B will have at most M elements. After removing the dummy vertices we will have found a solution to our original graph partitioning problem.

p-way partitionings

We conclude our discussion on the KL algorithm for the graph partitioning problem by discussing p-way partitionings for some $p > 0$. Again we will assume $|V| = pn$, and start with some initial partitioning $V = \cup_i A_i$ where $|A_i| = n$. Now we apply the 2-way partitioning algorithm described above to *pairs* of subsets. There are $\binom{k}{2}$ pairs of sets A_i, A_j to consider, such that we have to run our 2-way algorithm at least as many times as that. Of course, when we first run the algorithm for a pair (A_i, A_j) and next for a pair (A_j, A_k), then the pair (A_i, A_j) need no longer be relatively optimal. It turns out that in practice only a few passes are required for complete pairwise optimality.

KLFM for hypergraphs

To apply the KL heuristic to the SpMV partitioning problem we need to generalize the procedure to hypergraphs. For a hypergraph $\mathcal{H}$ we can rephrase the partitioning problem as finding a partitioning $\mathcal{V} = A \cup B$ such that as few as possible nets $n \in \mathcal{N}$ have elements in both A and B.

Definition 16 (Hypergraph partitioning problem)**.** Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ (for simplicity we consider here unit weights $w \equiv 1$), find a p-way partitioning $\mathcal{V} = \bigcup_{i=0}^{p-1} \mathcal{V}_i$ such that $|\mathcal{V}_i| \leq M$ for some given $M > 0$ that minimizes the $(\lambda - 1)$-metric LV.

Vertices of the hypergraph that share a net are called neighbours. A net is *cut* if it contains elements belonging to different parts V_i and V_j and is called *uncut* otherwise. In general a net has some *distribution* of elements (i.e. number of elements from A and B resp.). The size s_i of a vertex i is the number of nets it sits in, and the size of some set X is $|X| = \sum_{i \in X} s_i$.

Note that we can rewrite our constraint $|\mathcal{V}_i| \leq M$ as $|\mathcal{V}_i| \leq (1 + \epsilon)\frac{|V|}{p}$, where ϵ is some *maximum load-imbalance*. To keep our discussion clear, we consider here $p = 2$ only. The method we discuss here is due to Fiduccia and Matthyses [21], and is often referred to as the KLFM heuristic. As we will see it greatly improves the efficiency of the algorithm by choosing an appropriate data structure, a technique that can also be applied to the graph partitioning problem.

In order to extend KL for this hypergraph problem, we need to introduce the concept of *vertex gain*. This gain $g(v)$ for $v \in \mathcal{V}$ is defined as the change in the LV metric if we move the vertex v from the set V_1 to the set V_2. If we denote with $\mathcal{N}_v = \{n \in \mathcal{N} | v \in n\}$ the collection of nets of v, then we have $-|\mathcal{N}_v| \leq g(v) \leq |\mathcal{N}_v|$, since the move of v can only impact the nets in which it is contained. If we denote $\mathcal{N}_{\max} = \max_{v' \in \mathcal{V}} |\mathcal{N}_{v'}|$ then we have for all v:

$$-\mathcal{N}_{\max} \leq g(v) \leq \mathcal{N}_{\max} \tag{3.3}$$

The KLFM procedure simply chooses at iteration i the vertex v_i among all v which maximizes the gain, given that the move does not violate the balance criterion. It can happen that this gain is positive, however we will still allow the vertex to be moved in order to get out of local minima.

If we find the v_i naively, then this would take $\mathcal{O}(|\mathcal{V}|)$ time in each iteration, which would lead to a slow execution time. However the KLFM algorithm uses a convenient data structure called a *bucket list* which can find this maximum, and update the data structure after a move in $\mathcal{O}(1)$ time. Because of the observation (3.3), this data structure only takes takes $\mathcal{O}(\mathcal{N}_{\max} n)$ storage. In short, the idea is to group the vertices by their respective gains, using an array of maximum size $2\mathcal{N}_{\max} + 1$ whose kth element is a doubly-linked list of vertices that have gain $-\mathcal{N}_{\max} + k$.

This algorithm is usually run a fixed number of iterations, and we keep track of the best partitioning found so far (since the best vertex gain is not guaranteed to be negative). This leads to a heuristic for hypergraph partitioning that is *linear* in the input.

3.2.2 *Multi-Level methods*

For very large matrices, the corresponding hypergraphs are too large to apply KLFM to directly. Instead, they are first reduced in size (or coarsened), before a partitioning heuristic is applied. These methods are called multi-level methods, and were first developed by Bui and Jones [10]. They have been introduced as a heuristic for reducing the fill-in of sparse matrix factorization. It is similar in nature to the multigrid method used to solve (discretized) differential equations which was introduced by Brandt [9].

The multi-level method consist of three phases. In the first phase the (hyper)graph is *coarsened* by repeatedly merging vertices. In the second phase an *initial partitioning* is applied to the coarse graph. Since this graph is relatively small, more expensive methods may be used. In the third and final phase, the graph and the partition are uncoarsened, or *refined*.

In this section we will describe each of these phases specifically for the graph partitioning problem. These ideas apply equally well to hypergraph partitioning problems.

Phase 1: Coarsening

In order to coarsen the graph we need to identify collections of vertices that are 'close' in a certain sense, for example if they are neighbours in the graph. As we have discussed in Section 3.1.1, matchings are often employed for this task.

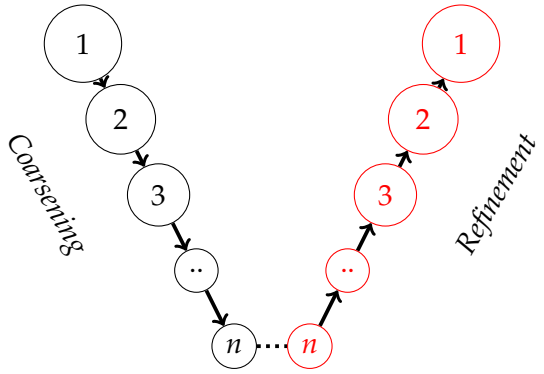


Figure 3.3.: An overview of the multi-level method. On the left, the graph is gradually reduced in size. Then an initial partitioning is applied. The graph is then refined to its original size, while the partition is relaxed at each step.

Phase 2: Initial partitioning

On the smaller graph a partitioning algorithm such as the KLFM heuristic is run to obtain a good initial partitioning. Because in this graph the number of vertices is limited, it is feasible to use more computationally expensive partitioners than usual. For example, the Mondriaan partitioner [51] runs the KLFM heuristic 8 times, and chooses the best result among these runs.

Phase 3: Refinement

After a good initial partitioning is obtained, the coarsening is undone step-for-step. After each *uncoarsening* the partitioning is relaxed using e.g. another run of KL. This process is called refinement.

Multi-level methods rely ultimately on the quality of the coarsening and initial partitioning.

3.2.3 *Medium-grain method*

The medium grain method (MG) [39] is a partitioning method that first tries to capture the structure of the matrix in a larger matrix B, and then partitions this matrix using a 1D partitioner on a coarse-grain model leading to a very efficient partitioning method. Because the method is 1-dimensional, but the resulting partitioning is 2D in nature, we can consider the method as being pseudo-2D.

In the MG method the matrix A is first split into two mutually disjoint parts A_r and A_c, i.e. $A = A_c \cup A_r$ and $A_c \cap A_r = \emptyset$ using some heuristic.

The part to which a vertex is assigned is decided by a cost function. An example of a successful cost function is assigning a nonzero to A_r if there are less elements in the row of the matrix containing the nonzero, and to A_c otherwise.

Once the parts A_r and A_c have been constructed, a matrix B is formed which has the form:

$$B = \left(\begin{array}{c|c} I_n & A_r^T \\ \hline A_c & I_m \end{array}\right),$$

where I_k is the k-dimensional identity matrix. To this matrix a relatively cheap column-based 1D partitioning method is applied; for example the KLFM method can be applied to the row-net model of the matrix B. The resulting partitioning of $B = B_1 \cup B_2$ is then projected back to a corresponding partitioning of the matrix $A = A_1 \cup A_2$ in the following fashion: if a nonzero (i, j) has been assigned to A_c then we simply assign the nonzero to A_1 if the column j is assigned to B_1, and to A_2 otherwise. If a nonzero (i, j) has been assigned to A_r then we assign it to A_1 if the column $n + i$ is assigned to B_1, and we assign it to A_2 otherwise.

This method is very intuitive, in that in general we should always prefer to keep groups of nonzeros in the same row or column together. However since every nonzero is both in a row and column we can not always satisfy this constraint since we can only assign a limited number of nonzeros to the same processor. Therefore we try to keep e.g. the smallest of the two (in terms of the number of nonzeros) together.

The MG method can be seen as a preprocessing step. In this context it is convenient to look at it as simply applying a *pre-coarsening* to the fine-graph model, which makes the model *almost* one-dimensional. The particular pre-coarsening that is applied is a direct result of the choice for the cost-function.

3.2.4 *PuLP*

So far the methods we have discussed rely on the KL method to do the actual partitioning of the (hyper)graph. Here we discuss a different method for graph-partitioning. This alternative approach to the graph partitioning problem is the PuLP method, which stands for **P**artitioning **u**sing **L**abel **P**ropagation [44]. As the name states, it is based on the idea of *label propagation* for graphs. Label propagation is commonly used for cluster detection in graphs.

Let $G = (V, E)$ be a graph, and let L be a *label function* $L : V \to \mathbb{Z}_k$. Label propagation consists of the following steps:

(i) We begin with a random label function L, for some fixed value of k. Intuitively this can be seen as assigning *labels* uniformly at random to each vertex from a set of k distinct labels.

(ii) We begin with an arbitrary vertex $v \in V$, and update its label to the most common label among its neighbours, that is to say we set:

$$L(v) = \text{argmax}_{0 \leq j < k} |\{u \in V \mid (v, u) \in E \text{ and } L(u) = j\}|$$

(iii) We repeat step (ii) while iterating over all vertices $v \in V$ until some stopping criteria is met. For example after a fixed number of iterations, or when no updates happen after one complete cycle over the vertices in V.

Assuming we stop after $c|V|$ iterations, we see that (after preprocessing the initial label counts of the neighbours of V), the running time of the algorithm is linear in $|V|$, which makes it very efficient.

This algorithm can be adapted to find a partitioning of G that attempts to minimize the edge cut, which leads to the PuLP method. This method is capable of minimizing multiple objectives, for example: both total edge cut, and maximal per-part edge cut (i.e. the maximum number of edges cut for a single part of the partitioning π), under multiple constraints, for example balance constraints on the size of the parts. It consists of three stages:

1. Initialize data structures, and perform an initial partitioning into parts using label propagation.

2. Label propagation based *balancing step*, that minimizes one of the objectives. For example, as explained below, the number of vertices in each part can be balanced using weighted label propagation.

3. An optional refinement step, that further improves upon a given objective. For example the KL method can be used to further improve upon the edge cut.

The second and third phases are repeated cycling through different techniques that focus on different objectives, for example balancing the number of edges per part while minimizing the per-part edge cut, or constrained refinement using the KL method. Furthermore, by using weights with our label propagation we can make 'bulky parts', i.e. parts that have already been assigned a lot of vertices, less likely to be chosen.

The PuLP algorithm will be the starting point from which we will build a self-improving partitioning method for sparse matrices. Therefore we will spend some time in the next chapter to look at each of these phases in detail, in preparation for generalizing the method to hypergraphs, and turning it into a self-improving method for the sparse matrix partitioning problem.

3.2.5 *Hypergraph partitioning software*

name	sequential / parallel	ref.
hMETIS	sequential	[33]
ML-Part	sequential	[12]
PaToH	sequential	[15]
Mondriaan	sequential	[51]
Zoltan	parallel	[20]

Table 3.2.: Here we give a selection of different software packages available that target the hypergraph partitioning problem. For a complete overview we refer to [7].

There exist many software packages that target the graph partitioning and hypergraph partitioning problems. We give an overview in Table 3.2, which is taken from [7]. These existing partitioners are pieces of software that are separate from applications, and may be viewed by users as black boxes that provide good partitionings for hypergraphs that may arise in their models.

As part of this thesis we have written a unified software package *Zee* that targets applications (linear algebra, iterative solvers) and the partitioning problem together. We discuss this software further in Section 4.7, and an extensive overview is given in Appendix B.

4

SELF-IMPROVING SPARSE MATRIX PARTITIONINGS

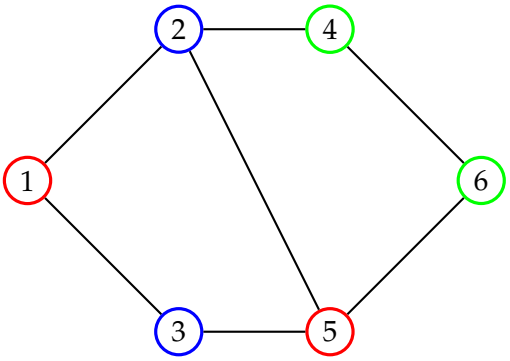
In the previous chapter we have introduced a number of methods that optimize matrix partitionings for the parallel SpMV algorithm. For some applications, the more involved methods can become prohibitively expensive. In many applications, matrices A arise that are specific to the problem, and are used as input to linear solvers an indefinite number of times. However, sometimes the matrices that show up only have a limited lifetime, or are generated ad-hoc and therefore a completely different system has to be considered by the solver every time. How much time we want to spend *partitioning the matrix*, which is only used throughout the lifetime of a particular solver run, should never be more than the performance we gain by using a parallel algorithm, since this would be counter-productive. Ideally we would stop improving the partitioning of a sparse matrix when the gain of any additional partitioning effort does not outweigh the added cost of partitioning further. We introduce the concept of *self-improving* partitioning methods, which automatically balance the computational effort put in partitioning with the actual application, in our case SpMV operations that arise in linear solvers.

4.1 A DETAILED LOOK AT THE PULP ALGORITHM

The PuLP algorithm was discussed in the previous chapter. Here we will generalize this algorithm to hypergraphs and discuss its parallelization so that it can target distributed systems. To this end we will first discuss each of the phases of the original algorithm in detail.

4.1.1 *Label propagation*

Label propagation is used for finding communities (groups of vertices that are in some sense related) in networks. An advantage of this method over other methods is that it is very cheap, in that it is a near-linear algorithm [37]. It also does not require any prior knowledge about the size of the



	red	green	blue
1	0	0	2
2	2	1	0
3	2	0	0
4	0	1	1
5	0	1	2
6	1	1	0

Figure 4.1.: An example of a labeled graph. The label of each vertex is one of {red, green, blue}. On the right we show a table with the label count of the neighbors for each vertex.

communities. In the following we denote with $L : V \to \mathbb{Z}$ the labeling. The label propagation algorithm can be separated into a number of steps [41]:

1. Each vertex is initially given a unique label. If we number the vertices $v_i \in V$ with $i \in \{1, \ldots, n\}$ we can write:

$$L(v_i) = i.$$

2. We apply a random permutation π to $\{1, \ldots, n\}$, and call the resulting index set X.

3. We visit each $x \in X$ in order and update the label:

$$L(v_x) = \operatorname{argmax}_{1 \leq i \leq n} C_i(N(v_x)).$$

   where $N(v)$ is the set of neighbours of v, and $C_i(X)$ is the number of elements in the set X with label i. This is to say we set the label of a vertex to the most common label among its neighbours. *Ties are broken in a random manner*. An example of a labeling after a number of iterations is given in Figure 4.1.

4. If each of the vertices v have label $L(v)$ equal to the most common label among its neighbours then we terminate the procedure. Otherwise we repeat step 3 until some other stopping criterion has been met (e.g. maximum number of iterations).

Intuitively this algorithm slowly grows groups of vertices of the same color. At first there are many different groups of vertices, but usually a number of dominating groups with the same label will start to grow increasingly large clusters within the graph.

4.2 GRAPH PARTITIONING USING LABEL PROPAGATION

When applying the ideas of label propagation to the *graph partitioning problem* (note that we treat here the general case of directed graphs), we identify a number of challenges:

- We need a way to force the formation of k communities, where here and in the following k denotes the number of parts we want to partition the graph in.
- These k resulting parts have to be of roughly equal size, such that we satisfy our load-balance constraints.
- We have multiple metrics that we want to optimize for (e.g. edge-cut, per-part edge cut, etc.), and our method should be able to incorporate all of these.

The PuLP algorithm [44] was introduced as a modification of the label propagation algorithm that is able to overcome these issues. We distinguish again a number of steps in the PuLP algorithm, which we will discuss one-by-one. The version of PuLP we present here is a somewhat simplified version of the original algorithm.

1. Initialization of the labels. Initial partitioning using *minimal-constraint label propagation*.
2. Iterative improvement
   a. Balancing by propagation of the labels.
   b. Refinement using an external method (such as KL)

The first *initialization* step differs from the standard label propagation algorithm in that it does not assign to each vertex a unique label, but instead chooses a label uniformly at random from the set $\{1, \ldots, k\}$, such that we start with k random parts that are each initially scattered throughout the graph. Furthermore, in this step we initialize the total counts $C : \mathbb{Z}_k \to \mathbb{N}$, and the degree-weighted neighbor counts for each vertex $N_v : \mathbb{Z}_k \to \mathbb{N}$ as an array of size k and $|V|$ arrays of size k respectively. See also Algorithm 4.1.

In the *minimal-constraint label propagation* phase the labels are updated, and in the process clusters are created of vertices that lie adjacent. The only constraint we put during this phase is that we make sure that each of the k parts is of a certain minimum size $s_{\min}$. This will make it easier in later stages to let the resulting partitioning satisfy the *load-balance constraints*. In the label propagation algorithm the most common label among the neighbours of a vertex is chosen as the new label, in PuLP the degree of a neighbour is taken

for all $v \in V$ **do**
    $L(v) \leftarrow \text{rand}(1, k)$.
    $C(L(v)) \leftarrow C(L(v)) + 1$.

for all $e \leftarrow (v, u) \in E$ **do**
    $N_u(L(v)) \leftarrow N_u(L(v)) + \text{degree}(v)$

Algorithm 4.1: Initializing the labels and their counts. Here $\text{rand}(a, b)$ is a function that returns an integer in the interval $[a, b]$ for $a, b \in \mathbb{Z}$. Note that since we consider directed graphs, we only update one of the vertices.

into account in the propagation phase. The idea behind this *degree-based weighted propagation* is that clusters will form around elements of high degree, such that the boundary of a part consists of vertices that are of relatively low degree which will be beneficial for the edge-cut metrics. In general the vertices are considered in a random order by shuffling the list $[1, ..., |V|]$, but we will not explicitly write this here. See also Algorithm 4.2.

After this initial partitioning phase, the partitioning π of G that we have obtained is *iteratively refined* in order to satisfy the load-balance constraint and to minimize our choice of metric(s), for example the total edge-cut, or the maximum per-part edge-cut. To satisfy the load-balance constraint $|\pi_i| \leq (1 + \varepsilon)(|V|/k)$ for all labels i, we put two constraints on changes during the iterative process. First, we set a maximum part size $s_{\max}$. If the size of a part exceeds this maximum size then it will not receive any additional vertices. Secondly, we introduce a *weight function* $W : \mathbb{N} \to \mathbb{R}^+$ which takes the size of a part π_i, and assigns to it a positive real number. This weight function satisfies $W(x) = 0$ for all $x \geq s_{\max}$, and W tends to infinity as x goes to zero. For example we can choose:

$$W_1 = \max(s_{\max}/x - 1, 0),$$

or if we want exponential behaviour we can choose

$$W_2(x) = \begin{cases} -\log\left(\frac{x}{s_{\max}}\right) & \text{if } x < s_{\max} \\ 0 & \text{otherwise} \end{cases}.$$

Otherwise this phase is completely analogous to the initial partitioning phase. Note that we still implicitly minimize the edge-cut by using degree-based propagation. This is also summarized in Algorithm 4.3.

After this balancing phase, the partitioning is explicitly refined to minimize the edge-cut further. We consider vertices that lie adjacent to another

$i \leftarrow 0,\ r \leftarrow 1$
while $i \leq I_1$ and $r \neq 0$ **do**
  $r \leftarrow 0$
  for all $v \in V$ **do**
    $p \leftarrow \text{argmax}_{1 \leq i \leq k} N_v(i)$
    if $p \neq L(v)$ and $C(L(v)) > s_{\min}$ **then**
      for all (v, u) and $(u, v) \in E$ **do**
        $N_u(L(v)) \leftarrow N_u(L(v)) - \text{degree}(v)$
        $N_u(p) \leftarrow N_u(p) + \text{degree}(v)$
        $C(L(v)) \leftarrow C(L(v)) - 1$
        $C(p) \leftarrow C(p) + 1$
      $L(v) \leftarrow p$
      $r \leftarrow r + 1$
  $i \leftarrow i + 1$

Algorithm 4.2: Building an initial partitioning using degree-based label propagation while keeping the partitionings larger than some minimum size $s_{\min}$. Here I_1 is the maximum number of iteration cycles, and r is the number of vertices that have had their label updated in the current cycle.

$i \leftarrow 0,\ r \leftarrow 1$
while $i \leq I_2$ and $r \neq 0$ **do**
  $r \leftarrow 0$
  for all $v \in V$ **do**
    for all $1 \leq i \leq k$ **do**
      if $C(i) + 1 \leq s_{\max}$ **then**
        $N'_v(i) \leftarrow N_v(i) \cdot W(C(i))$
      else
        $N'_v(i) \leftarrow 0$
    $p \leftarrow \text{argmax}_{1 \leq i \leq k} N'_v(i)$
    if $p \neq L(v)$ and $C(L(v)) > s_{\min}$ **then**
      for all $(v, u) \in E$ **do**
        $N_u(L(v)) \leftarrow N_u(L(v)) - \text{degree}(v)$
        $N_u(p) \leftarrow N_u(p) + \text{degree}(v)$
        $C(L(v)) \leftarrow C(L(v)) - 1$
        $C(p) \leftarrow C(p) + 1$
      $L(v) \leftarrow p$
      $r \leftarrow r + 1$
  $i \leftarrow i + 1$

Algorithm 4.3: Weighted degree-based label propagation in order to satisfy the load-balance constraint while implicitly minimizing the edge-cut. Here all the parameters have the same purpose as those of the previous phase, and I_2 is the maximum number of iterations in this phase.

part, and call them boundary vertices. In this refinement phase we consider moving these boundary vertices to adjacent parts, and see if this reduces the edge-cut. If it does, and adding a vertex to the adjacent part in question would not violate our load balance constraints, then we move the vertex from its current part to its adjacent part. See also Algorithm 4.4.

```
i ← 0, r ← 1
Initialize C without degree-weights
while i ≤ I_3 and r ≠ 0 do
    r ← 0
    for all v ∈ V do
        for all 1 ≤ i ≤ k do
            D(i) ← 0
        for all (v, u) ∈ E do
            D(L(u)) ← D(L(u)) + 1
        x ← argmax_{1≤i≤k} D(i)
        if x ≠ L(v) and C(x) + 1 ≤ s_max then
            C(L(v)) ← C(L(v)) − 1
            L(v) ← x
            C(x) ← C(x) + 1
            r ← r + 1
    i ← i + 1
```

Algorithm 4.4: Refining the partitioning to explicitly minimize the edge-cut by considering moves of boundary vertices to adjacent parts. Note that here we only consider the number of edges cut, and ignore the degree of a vertex.

Together these phases form a single-constraint single-objective partitioning method, which has been shown to be very efficient and gives good results on a large number of small-world graphs [44].

PuLP can be extended to optimize for multiple objectives under multiple constraints. We will not explore this extended method in more detail here, but instead focus on generalizing these ideas to the partitioning problem for hypergraphs.

4.3 LABEL PROPAGATION BASED PARTITIONING FOR HYPERGRAPHS

Because PuLP is a relatively inexpensive partitioning method, it makes for an attractive method when we are interested in obtaining a reasonable good partitioning that is not necessarily of the highest quality. In the context of

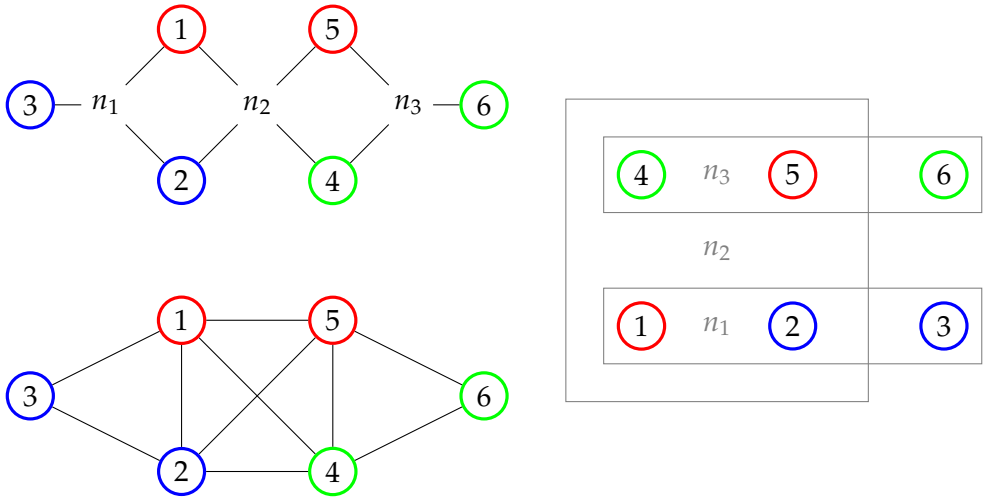


Figure 4.2.: An example of a hypergraph and its associated clique and star graph structure.

solving sparse matrix systems (in particular distributing for the SpMV operation), we have seen that the distribution quality is best modeled when using a hypergraph instead of a graph. In this section we will discuss possible generalizations of the PuLP method to hypergraphs.

4.3.1 *Indirect methods, graph representations*

A straightforward way to apply the PuLP method to hypergraphs is by representing our hypergraph as a graph, and run the PuLP partitioner on the resulting graph. Here we discuss two graph representations, the *star graph* G_* and the *clique graph* $G_\leftrightarrow$ associated to a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$.

To construct the star graph, we start with the vertex set $V = \mathcal{V}$, and add a vertex for each net in $\mathcal{H}$. Then we connect the vertex of each net, with all the vertices that are in that net. So we write $G_* = (V_*, E_*)$ where:

$$V_* = \mathcal{V} \cup \{v_{n_i} \mid n_i \in \mathcal{N}\}$$
$$E_* = \bigcup_{n_i \in \mathcal{N}} \{(v_{n_i}, v) \mid v \in n_i\}.$$

We let B be the matrix with a row for every vertex of our hypergraph, and a column for every net of the hypergraph, and with entries:

$$b_{ij} = \mathbb{1}(v_i \in n_j),$$

where $\mathbb{1}$ the indicator function. Then the star graph is precisely the bipartite graph associated to the matrix B.

To construct the clique graph $G_{\leftrightarrow} = (V_{\leftrightarrow}, E_{\leftrightarrow})$ we use the vertices of our hypergraphs, and connect all the vertices that share a net to each other, i.e. we add a clique for each net in our hypergraph:

$$V_{\leftrightarrow} = \mathcal{V}$$
$$E_{\leftrightarrow} = \{(u, v) \mid \exists_{n \in \mathcal{N}} \text{ s.t. } u, v \in n\}.$$

In terms of the matrix B, this clique graph is exactly the bipartite graph corresponding to BB^T. See also Figure 4.2 for an example of these associated graphs.

Note that the star graph is much sparser than the clique graph. However, we expect that the clique graph will yield better results, since vertices that share a net are direct neighbours in the clique graph, instead of sharing only a common neighbour (the vertex corresponding to their mutual net) in the star graph. In particular, for the star graph, this means that when applying label-propagation, all the information for the distribution of a net has to pass through the single vertex corresponding to that net. This means that label propagation will be delayed when compared to the finer structure of the clique graph. In particular, vertices are only influenced by the majority color of a net, while in the clique graph they are influenced directly by all vertices with which they share a net.

4.3.2 *Direct methods*

In general when applying PuLP to the graph models of the hypergraph that we introduced, the (hyper)graph partitioning that we obtain minimizes an edge-cut metric in the graph, which does lead ultimately to a reduction in the cut-net and $(\lambda - 1)$-metric of the hypergraph partitioning. For example, in the clique graph, vertices that share a net will be neighbours, such that when an edge is not cut, this means that two vertices in a net have obtained the same label. In the star graph, minimizing the edge cut means that vertices prefer to have the same label as the majority label in their nets, which should also end up decreasing the cut-net metric.

However, when optimizing for the cut-net metric (or more generally the $(\lambda - 1)$-metric), we are not interested in obtaining a partitioning where the majority of vertices in a net share the same label, but where the *number of different labels* in a net is as small as possible. Therefore we want to bias the process towards eliminating a label from a net completely, by reassigning the vertices to parts that are already well represented in the net. We conclude that if we want to minimize for the $(\lambda - 1)$-metric directly, applying PuLP to a graph representation of the hypergraph will not suffice. Here we will describe a possible generalization to hypergraphs which we call *Hyper-PuLP*.

As in the case of weighted label-propagation we introduce a total weight function. Let us write $Q(v, i)$ as a total weight function that denotes how

preferable it is to (re)assign $v \in \mathcal{V}$ to part π_i. For example, with label propagation (LP) on a graph $G = (V, E)$ we chose:

$$Q_{\text{LP}}(v, i) = \sum_{(v,u)\in E} \mathbb{1}_{L(u)=i},$$

where $\mathbb{1}$ is the indicator function. For weighted label propagation (WLP) we chose:

$$Q_{\text{WLP}}(v, i) = \sum_{(v,u)\in E} \mathbb{1}_{L(u)=i} w(u),$$

where in the case of degree-weighted propagation we had $w(v) = \text{degree}(v)$. In the case of minimizing for the $(\lambda - 1)$-metric of hypergraphs, the function Q will be given as a sum over the *nets* in which a given vertex resides. We want to encode two simple but key ideas in this function:

- We strongly prefer to not introduce new labels to a net that are currently not represented in the net. Furthermore when relatively few vertices in a net have a given label, we prefer not to give this label to more vertices in the hope that we can eliminate the label from the net entirely at some later point.
- When a label is already represented a lot of times in a net, we welcome vertices to take on that label, since we have little hope that we can eliminate this label from the net entirely.

This means that the contribution of a net n to the total weight function $Q(v, i)$ should be strongly negative as the number of vertices in n that belong to part i approaches zero, and should be strongly positive if this number of vertices approaches the size of the net $|n|$. Furthermore, we want this behaviour to be very steep, if a label is almost eliminated, or almost corresponds to the entire set, we want an exponentially low or high value respectively.

Our initial choice for hypergraph label propagation (HLP) will be:

$$Q_{\text{HLP}}(v, i) = \sum_{n\in\mathcal{N}, v\in n} w_{\text{HLP}}(i, n).$$

Here the weight function will be taken as the inverse hyperbolic tangent function:

$$w_{\text{HLP}}(i, n) = \log\left(\frac{1+x}{1-x}\right).$$

Here,

$$x = \alpha\left(\frac{2}{|n|}|\{u \in n \mid \pi(u) = i\}| - 1\right),$$

which corresponds to the remapping of the interval $[0, |n|]$ to $[-\alpha, \alpha]$. See also Figure 4.3. Because the inverse hyperbolic function tends to $-\infty$ and ∞ at the boundary of its domain $(-1, 1)$, we introduce the control parameter α.

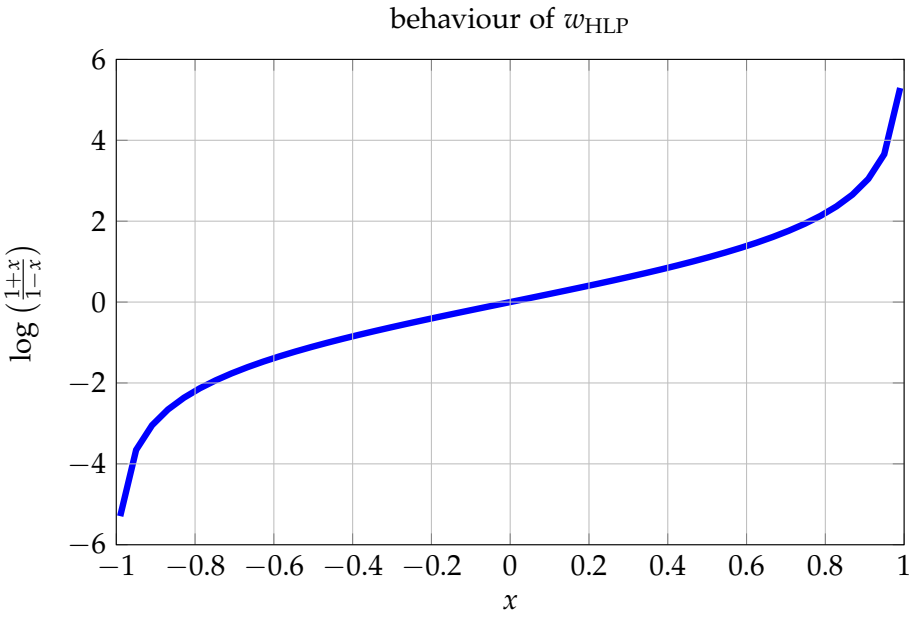


Figure 4.3.: The behaviour of the contribution of a net n to the preference function

In our experiments we found that the precise value of α is not critical for the behaviour of the partitioner, as long as it is chosen roughly in the interval $[\frac{2}{3}, 1)$, which corresponds to the range where the exponential behaviour of the function becomes apparent. We do not allow for label assignments that would make a part too large (i.e. moves that would violate the load balance constraint).

Initialization

In the previous section we have proposed a method to generalize the PuLP partitioner, in particular the balancing phase, to hypergraphs. Here we will introduce a method to initialize the hypergraph partitioning.

For graphs it is desired that parts form around vertices of high degree, so that vertices at the boundary of a part have small degree, which ultimately will lead to a low total edge-cut. For this reason the first phase of the PuLP algorithm uses *degree-weighted* propagation. For hypergraphs such an initial heuristic to form a reasonably good initial partitioning could be to build a partitioning with parts that keep relatively small nets together. The reasoning behind this is that we can view the minimization problem of the $(\lambda - 1)$-metric as attempting not to *break* nets in two. If a net has a low number of vertices, then this should be easier to accomplish. Furthermore, the largest nets are most likely to be broken in the final stage, so it may be a good strategy to disregard these largest nets at first.

We consider two options with which we can bias the process towards keeping small nets together. The first method we consider is to construct a set of *active nets* $\mathcal{A}$ in the *initialization* phase. When computing $Q_{\text{HLP}}^{(\text{initial})}$ we only consider the nets in $\mathcal{A}$:

$$Q_{\text{HLP}}^{(\text{initial})}(v,i) = \sum_{n \in \mathcal{A}, v \in n} w_{\text{HLP}}(i,n).$$

We add nets to $\mathcal{A}$ after every K cycles over the vertices, where K is a parameter we can choose freely. We can do this using different schemes. The main one we consider will be to sort the nets by size, and after the (Kj)th cycle in the initial phase the family $\mathcal{A}$ will contain the 2^j smallest nets of our hypergraph. When at least half of all the nets have been added to $\mathcal{A}$ we finish the initialization phase. Another way to view this is that we initially propagate labels on subhypergraphs of $\mathcal{H}$ that contain only nets that have small size.

Another method we consider is to scale the function w_{HLP} directly, by multiplying it with a factor $f(|n|)$ that depends on the size of the net. Possible options for such scaling functions are $f_1(|n|) = \exp(1/|n|)$ or $f_2(|n|) = (\max_i |n_i|) - |n|$. The idea is to make small nets count more towards the preference function Q, such that these are more likely to be kept together throughout the algorithm.

Implementation

Here we will give some details of sequential algorithms for the Hyper-PuLP method. In the initial (partitioning) phase we take the matrix in question A, and construct the hypergraph $\mathcal{H}$ according to a model of choice, e.g. the row-net model. We then choose in a uniformly random manner for each vertex one of k labels, corresponding to their initial part. Next we construct a number of data structures that keep track of the current state of the partitioning. The most important one keeps track of the *part distribution* for each net. We do this by constructing a list D of $|\mathcal{N}|$ distinct k-tuples. Each of these tuples encodes how many of the vertices $v \in n \in \mathcal{N}$ are in a given part, i.e.:

$$D_n(i) = |\{v \in n \mid L(v) = i\}|.$$

We also keep track of the total number of vertices in each part, such that we can satisfy any balance constraints:

$$C_i = |\{v \in \mathcal{V} \mid L(v) = i\}|.$$

```
H ← INITIALIZEHYPERGRAPH(A)

for all v ∈ V do
    L(v) ← rand(1, k)

for all n ∈ N do
    INITIALIZENETDISTRIBUTION(D_n)

for all 1 ≤ i ≤ k do
    INITIALIZEPARTSIZE(C_i)
```

Algorithm 4.5: Initializing the labels and data structures. Here again the $\text{rand}(1,k)$ function returns an integer in the range $[1,k]$ uniformly randomly. The initialization functions are trivial and therefore omitted.

The initialization of the method is summarized in Algorithm 4.5.

```
procedure PROPAGATELABEL(v, E, L)
    Q ← [0]
    for all n ∈ E_v do
        for 1 ≤ i ≤ k do
            Q[i] ← Q[i] + w(i, n)
    p ← argmax_{1≤i≤k} Q[i]
    if moving v to part p does not violate constraints then
        L(v) ← p
```

Algorithm 4.6: Pseudo-code for the label propagation function, on the sub-hypergraph defined by $(\mathcal{V}, \mathcal{E})$. We denote with $\mathcal{E}_v$ the set of nets in $\mathcal{E}$ that contain the vertex v. Here we compute the *preference function* Q for each part, and update the label to the part that maximizes this value as long as the move does not violate the constraints we wish to put. In general these constraints can even depend on the phase in which we use this procedure.

The label propagation procedure presented in Algorithm 4.6 is used for both phases. Note that we have not explicitly mentioned what constraints we put on our partitioning during each phase. As mentioned before, the PuLP method initially allows for oversized or moderately undersized parts

during the initial phase. We can employ a similar scheme here, although in our experiments we will simply disallow moves that lead to oversized parts, i.e. parts that violate the load-imbalance constraint. This is summarized in Algorithm 4.6.

```
A ← ∅
j ← 0
while |A| < ½|N| do
    A ← {n_i ∈ N | i ≤ 2^j}
    for 1 ≤ j ≤ K do
        for all v ∈ V do
            PropagateLabel(v, A, L)
```

Algorithm 4.7: Initial partitioning phase in which we consider subhypergraphs $\mathcal{A}$ that gradually increase in size. We assume that the nets are sorted by size.

In the initial phase we maintain the set of active nets, and gradually increase this in size after K cycles, in each of which we update every label once. Let us assume for simplicity that we have numbered the nets n_i with $i \in \{1, \ldots, |\mathcal{N}|\}$ such that $|n_i| \leq |n_{i+1}|$ for all $1 \leq i < |\mathcal{N}|$. This leads to the scheme of Algorithm 4.7.

```
i ← 0
while i ≤ I and not converged do
    for all v ∈ V do
        PropagateLabel(v, N, L)
    i ← i + 1
```

Algorithm 4.8: The balancing phase, where our goal is to obtain a good partitioning for the entire hypergraph $\mathcal{H}$. Here I is the (maximum) number of iterations we perform. The hypergraph is converged if after a complete cycle of updating the labels, we have not obtained an improved partitioning.

Finally the *balancing phase* is shown in Algorithm 4.8.

We have left out a number of details, some of which we will briefly mention here:

- We can put different constraints for each phase, and it may be worthwhile, like PuLP, to allow for oversized parts in the initial partitioning phase on subhypergraphs.

- In general the vertices are weighted, which should be taken into account when we check against the constraints we put.

- It may be beneficial to visit vertices in a random order each cycle, however in our experiments we have not seen large improvements when randomizing the index set.

The propagate label procedure has a time complexity of $\mathcal{O}(\text{degree}(v)k)$ where $v \in \mathcal{V}$, the degree of v is the number of nets containing v, and k is the number of parts. If k is large, we can also consider only a constant number of parts that are represented the most, as well as a constant number of parts that are represented the least, in any given net. To keep track of these *typical parts* for a net we can use an appropriate data structure from which we can extract such information, e.g. a Fibonacci heap (for each extreme) which can be updated efficiently to reflect the new part count for each net after a label update. This will further reduce this complexity.

We apply this function to $\mathcal{O}(|\mathcal{V}|)$ vertices during our entire algorithm, and need to sort the nets to construct $\mathcal{A}$. This takes $|\mathcal{N}| \log |\mathcal{N}|$ time. In total we see that the complexity of our straightforward implementation of this method is $\mathcal{O}(|\mathcal{V}|\overline{v}k + |\mathcal{N}| \log |\mathcal{N}|)$, where $\overline{v} = \max_{v \in \mathcal{V}} \text{degree}(v)$.

4.4 PARALLELIZING (HYPER-)PULP

Because we are interested in mixing partitioning and application steps, we want to parallelize the partitioning method. For the PuLP algorithm that targets graphs, it is not hard to parallelize the method for shared memory systems. Indeed, we can simply divide the vertices among the processors and let processors do the label propagation for the vertices assigned to them. This is also described in the original article [44].

In this section we will discuss the challenges when running label propagation algorithms on systems with distributed memory, and propose possible ways of handling these issues. We will only discuss the Hyper-PuLP algorithm, although most of the discussion will also apply to the graph partitioning method.

4.4.1 *Label propagation with distributed memory*

We let $\mathcal{H}$ be a hypergraph that has been given some initial partitioning, i.e. the vertices have been distributed over p processors. Within a partitioner iteration, we let each processor propagate labels for each of the vertices that has been assigned to it. If processor s wants to update the label for $v \in \mathcal{V}$, it requires the following information:

- The (part) distribution of all the nets $n \in \mathcal{N}_v$. Note that a part corresponds to a processor t, such that in accordance to the notation introduced before, we write $D_n(t)$ for the number of vertices in net n that are assigned to processor t.

- The current (total) part sizes, i.e. the number of vertices assigned to each other processor, which we have called C_t.

If we attempt to keep all this information locally on a processor, we identify two issues. First, the size of storing $D_n(i)$ for all relevant nets n, grows as $\mathcal{O}(|\mathcal{N}^{(s)}|p)$, where $\mathcal{N}^{(s)}$ denotes the collection of nets containing at least one vertex owned by processor s. For e.g. the fine-grain model corresponding to a matrix A, we have that $|\mathcal{N}^{(s)}|$ is at most twice the number of local vertices, but in general this collection can be prohibitively large, such that this can not be stored locally. Second, if we update the label of a vertex, it will change the distribution of all nets containing this vertex, such that information on remote processors will become outdated very quickly. Because latency is generally high, it is infeasible to synchronize all processors after every label update.

The first issue can be overcome by assigning each net to a processor, which is then responsible for keeping track of the part distribution within that net. This way, there is no redundant storage required. However, we want to assign each net to the processor that benefits from knowing this part distribution most. A possible way of doing is is by assigning each net to the part (resembling a processor) that owns the most vertices within this net. This way, processors will have to obtain part distributions stored remotely less often. The part sizes C_t can still be stored locally, and updated when required.

However, during a complete cycle over the vertices in a partitioner iteration, a processor will end up requiring data that has the size $\mathcal{O}(|\mathcal{N}^{(s)}|p)$. Because it may be infeasible to store all this data simultaneously, we have to make groups of vertices that have many common nets so that we only have to consider a subset of this data at any given time. This can be done for example by *matching* the local vertices.

The second issue can be overcome by allowing the part distributions of nets and the part sizes to be outdated, and only update remote counters after a fixed number of iterations. This significantly reduces the need to synchronize, but may come at a cost of overall partition quality.

4.4.2 *Migration costs*

When we reassign a vertex to another part, we require a number of updates to our data.

1. The vertex is no longer owned by a processor, which means we have to *migrate* the vertex to the new processor it has been assigned to.
2. The part distribution for the nets of this vertex has changed.
3. Also, the majority part of the nets of the vertex could be changed by this reassignment, which means that the part distribution of that net too has to be migrated.
4. The part sizes should be updated to reflect the new owner of v.

As hinted at in the previous section, we can *cache* these changes locally, and after a fixed amount of iterations update remote data structures only after a fixed number of iterations. In this way we make sure that the method has a relatively low *migration cost*, since we only synchronize when there are enough changes to warrant the synchronization latency.

4.5 APPLICATION TO SPMV PARTITIONING

We can apply the Hyper-PuLP partitioning method to the sparse matrix partitioning problem by choosing an appropriate hypergraph model to input to the problem. However, choosing the best model is not easy. In our initial experiments we have seen that using the fine-grain model with a random initial distribution does not lead to good partitionings. Therefore we focus mostly on choosing between the *row-net* and the *column-net* models.

The question of finding a *recipe* for choosing the best model has been posed before [18]. If the matrix is non-square, then we could for example choose the model that minimizes the number of nets. Alternatives may be to see how *dense* the nets for each model are, and take as a criterion whether the median net size is larger for the column-net or row-net model.

Although this might not be feasible in practice, the recipe we will use in our experiments is to take the model that has the lowest communication volume when distributed cyclically.

4.6 AUTO-BALANCING PARTITIONING AND APPLICATION

In this chapter we have proposed and developed a novel hypergraph partitioning method inspired by label propagation partitioning for graphs, that has low computational complexity. One of the targets of this partitioning method could be to obtain reasonably good partitionings in cases where the matrix to be partitioned is generated ad-hoc, and will only see a single use. The application we focus on in this work is to find a solution to sparse linear systems of equations.

If one would have perfect information on the number of times the matrix will be used, as well as the specifics of the hardware the algorithm will

run on, then one could in principle predict exactly how the running time is influenced by the specifics of the matrix distribution (such as load imbalance, communication volume and per-part communication volume). With this information one could ultimately precisely predict how to balance the *partitioning effort* and the actual application to lead to the smallest total cost.

However, the convergence behaviour of a linear solver is highly dependent on the matrix in question. Furthermore, it becomes increasingly hard to lower the communication volume through more partitioning effort. In this section we propose a method to automatically balance the partitioning and application effort with the goal of minimizing the total (cumulative) running time.

Partitioning method

The partitioning method we use in this context should be of an iterative nature. In particular, we should be able start from an initial partitioning (this could even correspond to the cyclic distribution) which we *iteratively refine* (from now on we will refer to these steps as *partitioning iterations*) depending on information that comes out of our application . For example, by systematically inspecting the convergence behaviour e.g. by looking at the residual $||\vec{b} - A\vec{x}_i||$ after the ith *iteration* of our linear *solver* (from now on called *solver iterations*), and comparing this to the required precision ρ, we can try to predict how many more solver iterations we require, and translate this into the optimal number of *partitioning iterations*.

Epochs

To make this more precise we adopt a similar naming scheme to the adaptive method introduced in [17]. We begin with some initial partitioning $\pi^{(0)}$ of (the hypergraph model of) A. We define an epoch to be a number of applications of A (i.e. solver iterations) before *repartitioning*, i.e. before the next partitioning iteration. We will say epoch j has α_j solver iterations, after which the partitioning $\pi^{(j)}$ will be refined to the partitioning $\pi^{(j+1)}$.

After each solver iteration we predict with a later specified method the number of solver iterations left, and we call this number $\hat{N}$, where the hat indicates that it is an estimation. Furthermore, we denote with $\Delta\hat{V}$ the (per-part) communication volume we gained during the previous partitioning iteration, and with $\hat{t}$ the (wall) time the iteration took. Let T be the BSP cost of a single solver iteration, and $T_{\text{SpMV}}(V)$ be the BSP cost of the SpMV algorithm which depends crucially on the communication volume V per

part. We assume $T \approx T_{\text{SpMV}}$. If we would not refine our partitioning the expected running time that is left is:

$$\hat{N} T_{\text{SpMV}}(V),$$

while if we refine our partitioning, which takes time $\hat{t}$, the running time that is left is approximately:

$$\hat{N} T_{\text{SpMV}}(V - \Delta\hat{V}).$$

We therefore take as a criterion to decide whether the next iteration should be a *refinement* or a *solver* iteration respectively, whether or not the following holds:

$$\hat{N} T_{\text{SpMV}}(V) > \hat{t} + \hat{N} T_{\text{SpMV}}(V - \Delta\hat{V}).$$

Predicting convergence and per-iteration gain

Our criterion requires three estimates to be made; $\hat{N}$, $\Delta\hat{V}$ and $\hat{t}$. For the latter two quantities, our initial experiments indicate that we can take the communication volume reduction and running time obtained in the previous partitioner iteration since they remain for more or less constant throughout the method, until convergence. However, in a parallel setting this may be very different, and maybe a more sophisticated method will turn out to be required.

We will thus focus on finding a good estimate $\hat{N}$, and analyze the cost of doing so. Good iterative solvers have superlinear convergence behaviour. We will mean with convergence behaviour the norm of the residual, as a function of the number of iterations. After k solver iterations, we have a collection of k pairs: $R = \{(x, ||r_x||) \mid 0 \leq x < k\}$. Because iterative solvers generally have superlinear convergence behaviour, we can try to model this using a function:

$$\hat{f}(x) = ||r_0|| - ax - bx^2.$$

Here, the parameters $a, b \in \mathbb{R}_{\geq 0}$ are taken as non-negative reals. We can find optimal parameters, in the least-squares sense, by fitting against the data points in R. Finally we take as $\hat{N}$ the first $y \in \mathbb{N}$ such that $\hat{f}(y) < \rho$, where ρ is the tolerance level we require.

The quality of our fit depends on the linear system itself, as well as the iterative solver that is used. Minimum residual methods such as GMRES have the property that the norm of the residual decreases monotonically every iteration. This is not necessarily the case for solvers that use another sense of optimality, in Appendix A we give an overview of the types of Krylov subspace methods that exist. Linear solvers with more stable convergence behaviour such as BiCGSTAB [49] may be more suitable for this method than solvers with erratic convergence behaviour.

Note also that we have some freedom in choosing when to update $\hat{N}$, since it is probably not necessary to do so after every iteration.

4.7 ZEE

The workflow of separating partitioning and application is no longer feasible when we want to apply the methods set out in the chapter, so that we have to incorporate the partitioning methods within our application. Since self-improving partitionings depend crucially on the application, we need software that mixes these two operations. Therefore we have developed a partitioning framework, named Zee, that does exactly this. We give some more background and a short introduction to the Zee partitioning framework in Appendix B.

4.8 RESULTS

In this section we present the results of some initial experiments that have been done for the proposed partitioning and balancing methods. For these experiments a sequential implementation of Hyper-PuLP on top of the Zee library was used. The programs were compiled with the GNU Compiler Collection (GCC) 5.2.0. It was executed on a computer running Arch Linux with Linux kernel 4.2.5. The computer was equipped with an Intel Core i5 − 4670 CPU running at 3.4 GHz, and 4 GB of RAM.

Hyper-PuLP

We have run the Hyper-PuLP algorithms for a variety of matrices found in the University of Florida collection of sparse matrices [19]. We will focus exclusively on bipartitionings.

We developed our partitioning method to be a cheap alternative to existing methods, that is still able to improve the communication volume beyond the value corresponding to e.g. the cyclic distribution. In light of this, an important metric will be the relative communication volume compared to a *zero-cost baseline*, i.e. the communication volume that could be obtained with no partitioning effort. For each a matrix A, we let V_C be the communication volume of a cyclic bipartitioning of A. If we let V_{HP} be the communication volume obtained by the Hyper-PuLP algorithm then we are interested in the *gain* G, which we will define as:

$$G = \left(1 - \frac{V_{\text{HP}}}{V_C}\right) \times 100\%.$$

If we average over multiple runs with random initial partitionings, then we will take for V_{HP} the average communication volume found. If this gain is large enough, then we have enough room in the communication volume to apply the method we proposed for balancing the solver- and partitioner iterations.

matrix	m	n	N	V_{HP}	$V_{\text{HP}}^{\text{min}}$	V_C	G	model
08blocks	300	300	592	8.0 ± 0.0	8.0	8	0.0%	row-net
GD02_b	80	80	232	35.8 ± 5.5	21.0	49	27.0%	row-net
GD95_c	62	62	287	34.3 ± 9.2	9.0	50	31.4%	column-net
GD97_a	84	84	332	54.1 ± 6.6	40.0	77	29.7%	column-net
IG5-7	62	150	549	49.8 ± 3.4	42.0	58	14.1%	row-net
ash608	608	188	1216	84.4 ± 12.9	50.0	186	54.6%	column-net
ash85	85	85	523	45.2 ± 8.2	21.0	85	46.9%	column-net
cage5	37	37	233	31.6 ± 2.9	25.0	37	14.6%	column-net
cage6	93	93	785	63.3 ± 7.0	49.0	93	32.0%	column-net
ch4-4-b2	96	72	288	51.4 ± 4.8	39.0	65	20.9%	column-net
chesapeake	39	39	340	36.5 ± 1.8	31.0	37	1.4%	column-net
curtis54	54	54	291	31.4 ± 5.4	16.0	54	41.9%	column-net
dwt_162	162	162	1182	73.0 ± 17.9	30.0	162	55.0%	column-net
dwt_87	87	87	541	44.2 ± 9.2	23.0	85	48.0%	column-net
flower_5_1	211	201	602	86.6 ± 8.3	63.0	51	-69.7%	row-net
football	115	115	1226	91.1 ± 7.9	78.0	115	20.8%	column-net
fs_183_3	183	183	1069	126.8 ± 16.3	98.0	143	11.3%	row-net
gams10am	114	171	407	55.1 ± 3.0	45.0	54	-2.1%	column-net
impcol_a	207	207	572	88.7 ± 10.8	62.0	127	30.2%	row-net
lp_scagr7	129	185	465	73.0 ± 8.4	43.0	72	-1.4%	column-net
lp_share1b	117	253	1179	46.7 ± 12.5	22.0	102	54.2%	row-net
lpi_mondou2	312	604	1208	88.3 ± 16.1	61.0	281	68.6%	row-net
mesh1e1	48	48	306	39.5 ± 7.3	24.0	48	17.6%	column-net
odepa400	400	400	1201	135.1 ± 13.5	103.0	400	66.2%	column-net
pde225	225	225	1065	103.2 ± 18.0	65.0	225	54.1%	column-net
rajat05	301	301	1384	125.0 ± 18.3	86.0	298	58.0%	column-net
rdb200	200	200	1120	99.8 ± 19.6	51.0	200	50.1%	column-net
rel5	340	35	656	20.9 ± 3.1	13.0	33	36.5%	column-net
saylr1	238	238	1128	107.5 ± 19.5	70.0	238	54.8%	column-net
steam3	80	80	928	36.4 ± 14.1	8.0	80	54.5%	column-net
west0067	67	67	294	40.8 ± 7.1	16.0	50	18.5%	column-net
wheel_6_1	83	85	254	41.0 ± 4.9	28.0	58	29.4%	row-net
will57	57	57	281	26.1 ± 8.6	9.0	56	53.4%	column-net
ww_36_pmec_36	66	66	1194	38.3 ± 1.1	38.0	56	31.6%	row-net
zed	116	142	666	31.3 ± 3.4	24.0	37	15.3%	column-net

Table 4.1.: Communication volumes obtained by applying Hyper-PuLP to a selection of small matrices over 100 runs with $\epsilon = 0.03$. The columns show respectively the name of the matrix, the number of rows and columns, the number of nonzeros, the average communication volume obtained by Hyper-PuLP and the standard deviation, the minimum communication volume obtained, the cyclic communication volume, the gain, and the model that was used.

matrix	m	n	N	V_{HP}	$V_{\mathrm{HP}}^{\mathrm{min}}$	V_{C}	G	model
Franz1	2240	768	5120	1220.1 ± 26.2	1161.0	320	-281.3%	row-net
GD99_b	64	64	252	46.6 ± 6.0	27.0	64	27.1%	column-net
M40PI_n1	2028	2028	5007	980.8 ± 45.2	871.0	948	-3.5%	column-net
Roget	1022	1022	5075	579.0 ± 21.6	539.0	746	22.4%	column-net
SmaGri	1059	1059	4919	289.7 ± 15.2	254.0	375	22.8%	row-net
TF10	99	107	622	71.1 ± 6.0	51.0	95	25.2%	row-net
bcspwr06	1454	1454	5300	565.6 ± 52.8	457.0	1242	54.5%	column-net
bcsstk03	112	112	640	42.8 ± 14.3	0.0	112	61.7%	column-net
bp_1000	822	822	4661	371.8 ± 52.9	262.0	595	37.5%	column-net
bp_1400	822	822	4790	281.8 ± 51.5	179.0	587	52.0%	row-net
bp_1600	822	822	4841	282.1 ± 44.8	195.0	586	51.9%	row-net
bp_800	822	822	4534	366.4 ± 48.5	258.0	582	37.0%	column-net
can_73	73	73	377	46.0 ± 6.3	32.0	73	37.0%	column-net
cdde1	961	961	4681	387.0 ± 48.3	286.0	961	59.7%	column-net
de063157	936	1908	5119	362.0 ± 28.0	296.0	700	48.3%	row-net
de080285	936	1908	5082	354.0 ± 22.7	299.0	700	49.4%	row-net
dwt_607	607	607	5131	251.1 ± 38.1	153.0	607	58.6%	column-net
dynamicSoaringProblem_1	647	647	5367	371.8 ± 32.5	263.0	602	38.2%	column-net
gent113	113	113	655	60.6 ± 8.8	26.0	98	38.1%	row-net
illc1033	1033	320	4732	67.2 ± 19.7	27.0	274	75.5%	column-net
l9	244	1483	4659	143.0 ± 16.0	102.0	244	41.4%	row-net
lp_agg2	516	758	4740	227.9 ± 32.7	131.0	292	22.0%	column-net
lp_agg3	516	758	4756	228.7 ± 30.8	129.0	272	15.9%	column-net
lp_recipe	91	204	687	37.7 ± 10.4	12.0	81	53.5%	row-net
lpi_klein2	477	531	5062	54.0 ± 0.0	54.0	54	0.0%	column-net
lpi_pilot4i	410	1123	5264	176.9 ± 27.4	110.0	393	55.0%	row-net
n2c6-b3	1365	455	5460	343.5 ± 21.9	281.0	455	24.5%	column-net
n3c5-b7	30	120	240	22.9 ± 2.2	17.0	30	23.6%	row-net
n3c6-b3	1365	455	5460	339.5 ± 20.7	281.0	455	25.4%	column-net
n4c5-b3	1350	455	5400	340.6 ± 17.1	295.0	455	25.1%	column-net
netscience	1589	1589	5484	356.6 ± 68.3	237.0	993	64.1%	column-net
nos5	468	468	5172	254.9 ± 51.4	145.0	400	36.3%	column-net
nos7	729	729	4617	376.2 ± 46.6	289.0	729	48.4%	column-net
rel6	2340	157	5101	106.0 ± 6.6	91.0	155	31.6%	column-net
spaceStation_3	467	467	5103	239.2 ± 58.7	96.0	407	41.2%	column-net
well1033	1033	320	4732	69.5 ± 21.2	30.0	274	74.6%	column-net
west0067	67	67	294	42.0 ± 6.2	15.0	50	16.1%	column-net

Table 4.2.: Communication volumes obtained by applying Hyper-PuLP to a selection of matrices with around 5000 nonzeros over 100 runs with $\epsilon = 0.03$. For a description of the columns see Table 4.1.

matrix	m	n	N	V_{HP}	$V_{\text{HP}}^{\text{min}}$	V_{C}	G	model
CAG_mat364	364	364	13585	172.4 ± 42.8	64.0	354	51.3%	column-net
Chebyshev2	2053	2053	18447	410.5 ± 45.5	305.0	4	−10161.5%	row-net
D_8	1132	1271	14966	824.8 ± 30.6	773.0	1123	26.6%	row-net
USpowerGrid	4941	4941	13188	1811.1 ± 90.2	1624.0	2605	30.5%	column-net
adder_trans_02	1814	1814	14579	1234.5 ± 133.4	948.0	1733	28.8%	column-net
bcsstm37	25503	25503	15525	378.4 ± 18.6	335.0	440	14.0%	column-net
ch7-6-b2	4200	630	12600	494.6 ± 24.2	430.0	630	21.5%	column-net
flower_4_4	1837	5529	16466	1187.3 ± 48.7	1030.0	1630	27.2%	row-net
orsreg_1	2205	2205	14133	995.6 ± 123.2	672.0	2205	54.8%	column-net
photogrammetry	1388	390	11816	246.2 ± 33.0	132.0	378	34.9%	column-net
rajat12	1879	1879	12926	1317.7 ± 264.6	640.0	1618	18.6%	column-net
rdb2048	2048	2048	12032	844.9 ± 87.1	572.0	2048	58.7%	column-net
robot24c1_mat5_J	302	404	15118	225.1 ± 9.2	208.0	296	23.9%	row-net
rosen8	520	1544	16058	201.1 ± 46.7	87.0	520	61.3%	row-net
spaceStation_10	1272	1272	17478	623.0 ± 127.3	317.0	1213	48.6%	column-net
spaceStation_5	1019	1019	15219	463.4 ± 102.4	254.0	899	48.5%	column-net
spiral	1434	1434	18228	418.1 ± 96.5	177.0	1433	70.8%	column-net

Table 4.3.: Communication volumes obtained by applying Hyper-PuLP to a selection of matrices with around 15000 nonzeros over 100 runs with $\epsilon = 0.03$. For a description of the columns see Table 4.1.

matrix	m	n	N	V_{HP}	$V_{\text{HP}}^{\text{min}}$	V_{C}	G	model
TS	2142	2142	45262	1020.0 ± 153.8	672.0	1960	48.0%	row-net
Zewail	6752	6752	54233	2038.1 ± 260.8	1276.0	3288	38.0%	row-net
abtaha1	14596	209	51307	143.4 ± 6.6	127.0	209	31.4%	column-net
bloweybq	10001	10001	69991	7556.4 ± 770.1	6680.0	10001	24.4%	column-net
ca-HepTh	9877	9877	51971	4057.4 ± 251.2	3563.0	6236	34.9%	column-net
complex	1023	1408	46463	384.5 ± 3.8	348.0	255	−50.8%	column-net
eurqsa	7245	7245	46142	4258.9 ± 160.2	3836.0	6405	33.5%	column-net
ex24	2283	2283	48737	892.9 ± 135.0	633.0	2283	60.9%	column-net
l30	2701	16281	52070	1493.8 ± 73.9	1311.0	2701	44.7%	row-net
mark3jac020	9129	9129	56175	4413.1 ± 230.4	3934.0	7511	41.2%	column-net
mimo46x46_system	13250	13250	48735	4417.2 ± 270.6	3648.0	9851	55.2%	row-net
mimo8x8_system	13309	13309	48872	4477.0 ± 294.8	3693.0	9882	54.7%	row-net
rajat13	7598	7598	48922	5144.6 ± 739.8	3375.0	7536	31.7%	column-net

Table 4.4.: Communication volumes obtained by applying Hyper-PuLP to a selection of matrices with around 50000 nonzeros over 100 runs with $\epsilon = 0.03$. For a description of the columns see Table 4.1.

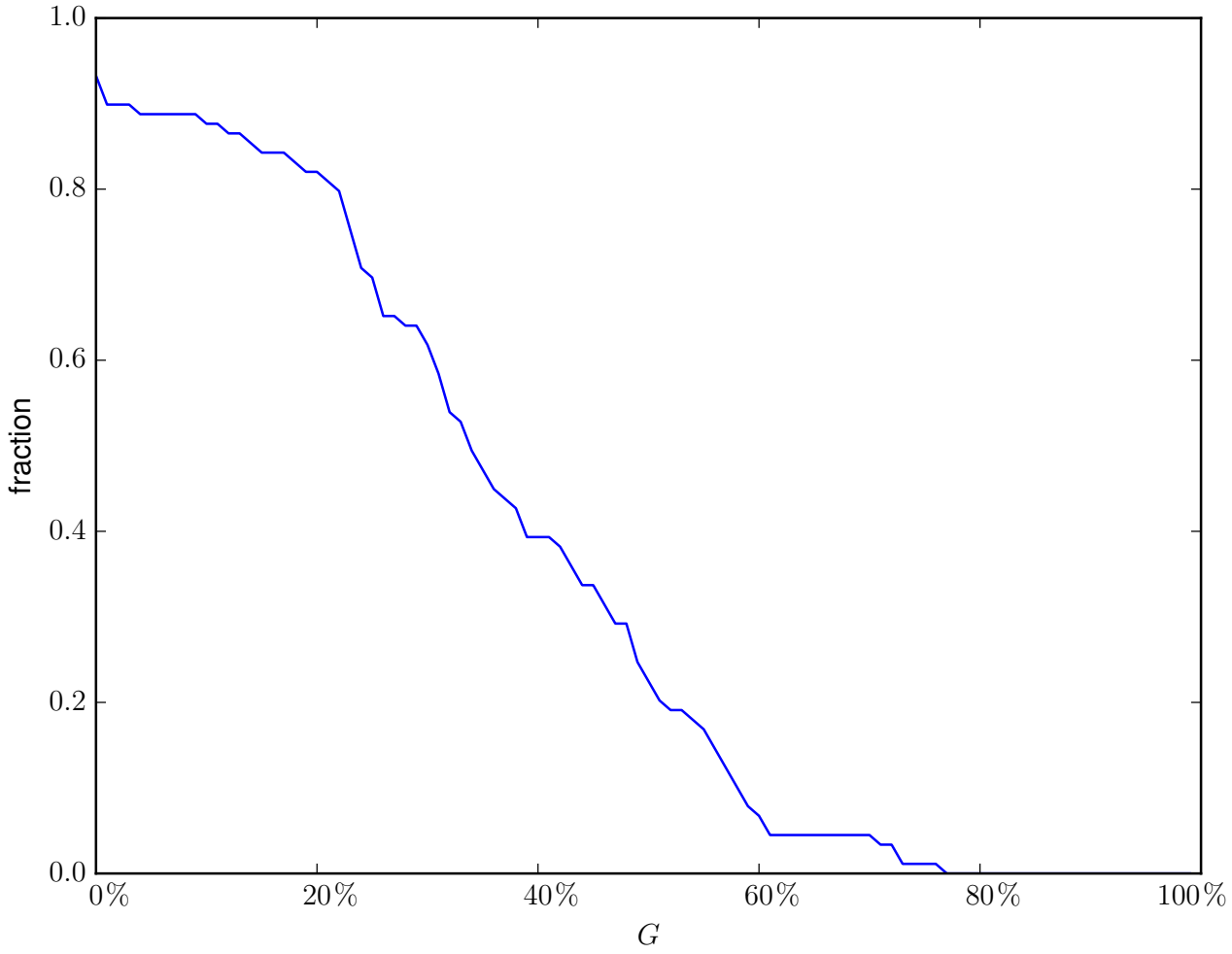


Figure 4.4.: In this figure we plot for all the matrices that were considered, i.e. that were shown in Tables 4.1 through 4.4, the fraction (vertical axis) for which we were able to obtain at least an average gain of G (horizontal axis) over 10 runs.

In Table 4.1 we show the results of running the Hyper-PuLP algorithm on a collection consisting of a selection of relatively small matrices. In all of our results we exclude matrices that had a cyclic communication volume of 0 from analysis. Also, we disregard matrices for which we were unable to find an initial partitioning that satisfied the load-imbalance constraint. Note that for the majority of the matrices, the Hyper-PuLP algorithm is able to greatly improve upon the cyclic partitioning. In Table 4.2 we show the results for a selection of matrices that have around 5000 nonzeros. In Table 4.3 the matrices considered have around 15000 nonzeros. Finally, in Table 4.4 we consider matrices with around 50000 nonzeros.

In Figure 4.4 we show the fraction of matrices for which a certain gain over the cyclic distribution was minimally obtained.

We see that there are a limited number of matrices for which the method performs poorly, and for a small selection we even end up with worse partitionings than the cyclic partitioning. Looking at e.g. the structure of the `flower_5_1` and `Franz1` matrices, we see that they have a non-structured non-zero pattern for which the cyclic partitioning does particularly well. Inspecting the behaviour of Hyper-PuLP on these matrices more closely may lead to ideas for improving the method, so that it also performs well for this category of matrices.

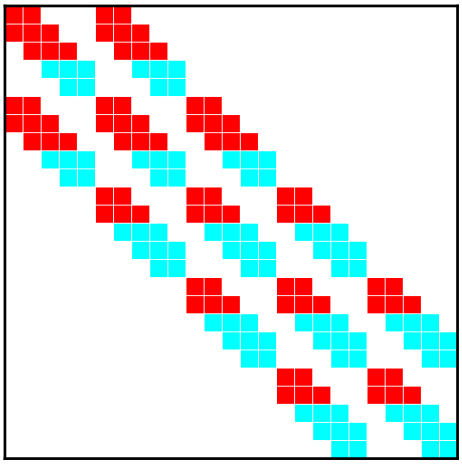
For these two matrices, as well as the large matrix `complex`, the model chosen actually yields worse results than the alternative model, but the cyclic distributions indicate otherwise, resulting in a negative gain. When the algorithm is run with the other model, these matrices actually do obtain an improvement over the cyclic communication volume of the other model. This indicates that a good recipe for choosing a model is a very important component of the method discussed in this chapter.

All in all, the numbers we obtained indicate that the Hyper-PuLP method is certainly capable of improving the communication well beyond communication volumes that are obtained with no partitioning effort. This warrants further investigation of partitioning techniques based on label propagation for hypergraphs.

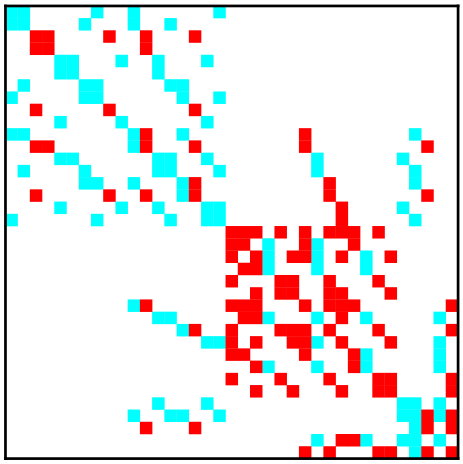
In Figure 4.5 we show the best bipartitionings obtained for a selection of the small matrices we considered. In Figure 4.6 we show the best bipartitioning found for the `cage7` matrix. In Figure 4.7 we show the progress the algorithm makes while partitioning the `steam3` matrix. Note that the rows, which in this case correspond to the nets, slowly eliminate the non-majority label throughout the run, which is what we expect to see.

4.9 SUMMARY

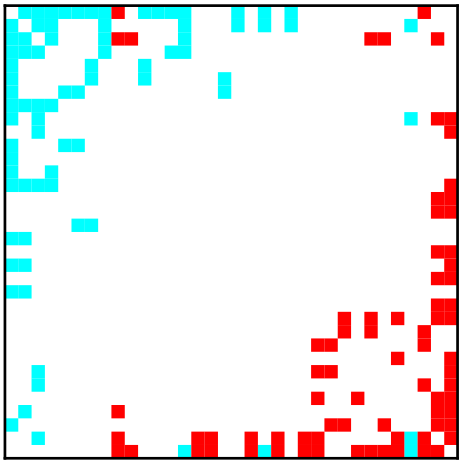
We proposed a modification to PuLP, a partitioning technique that uses label propagation, to target the partitioning problem for hypergraphs, which we



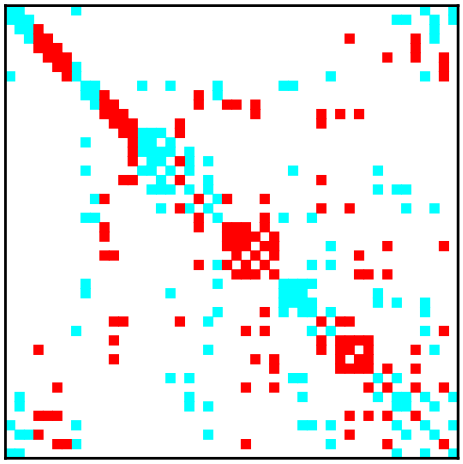
(a) `lap_25`. $V = 12$



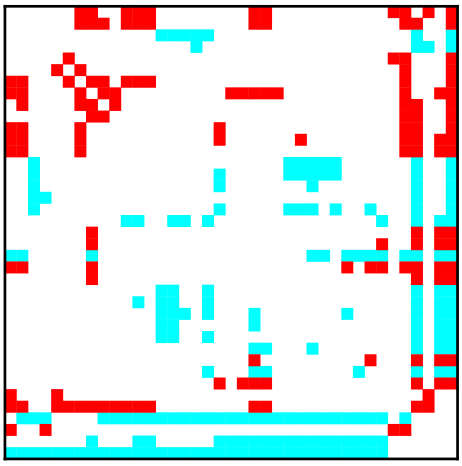
(b) `cage5`. $V = 19$



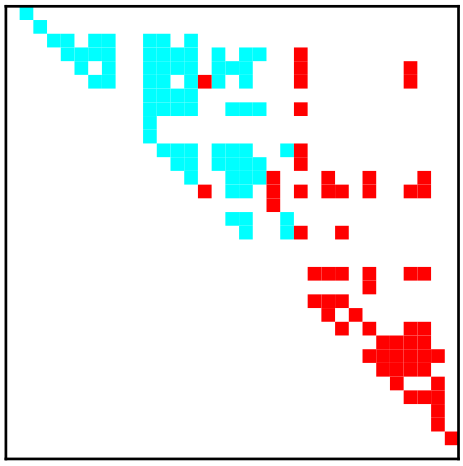
(c) `karate`. $V = 12$



(d) `mesh1e1`. $V = 22$



(e) `chesapeake`. $V = 27$



(f) `GD01_c`. $V = 9$

Figure 4.5.: Here we show spy plots of the best bipartitionings obtained for six small matrices. One-dimensional models were used for the partitioning. The matrix name and the communication volume are given for each of the matrices.

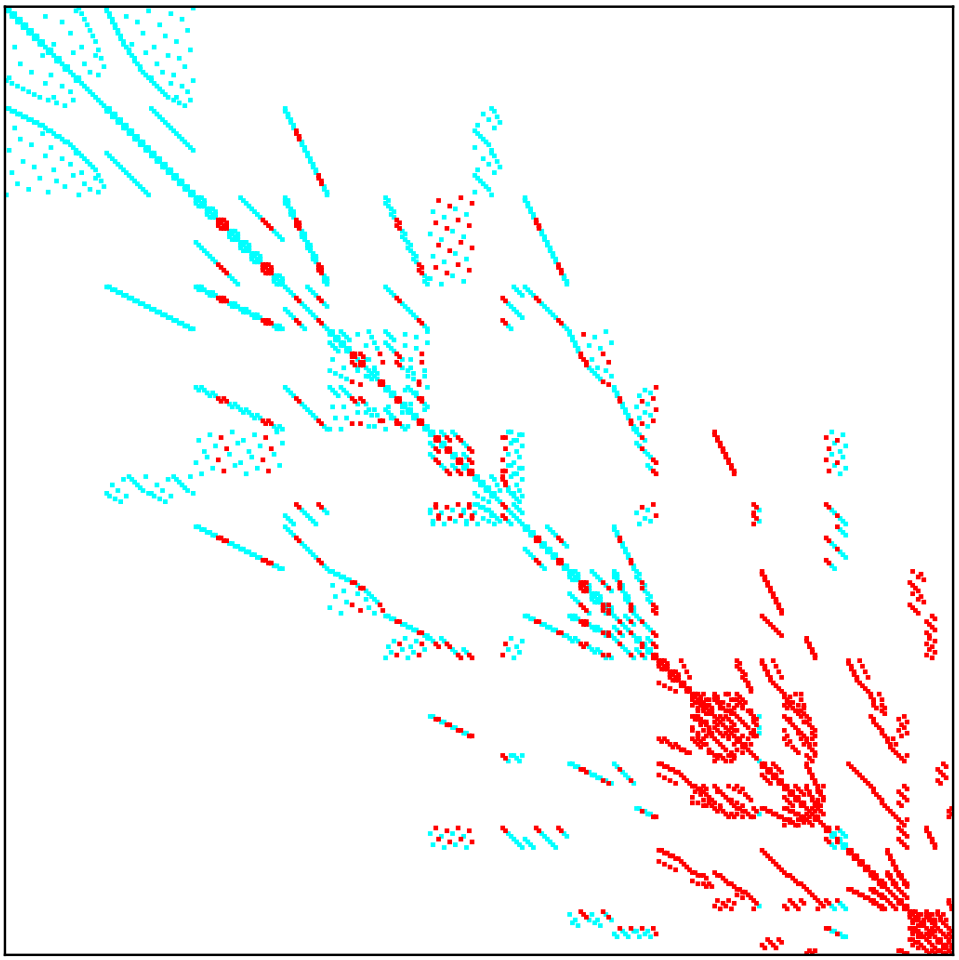
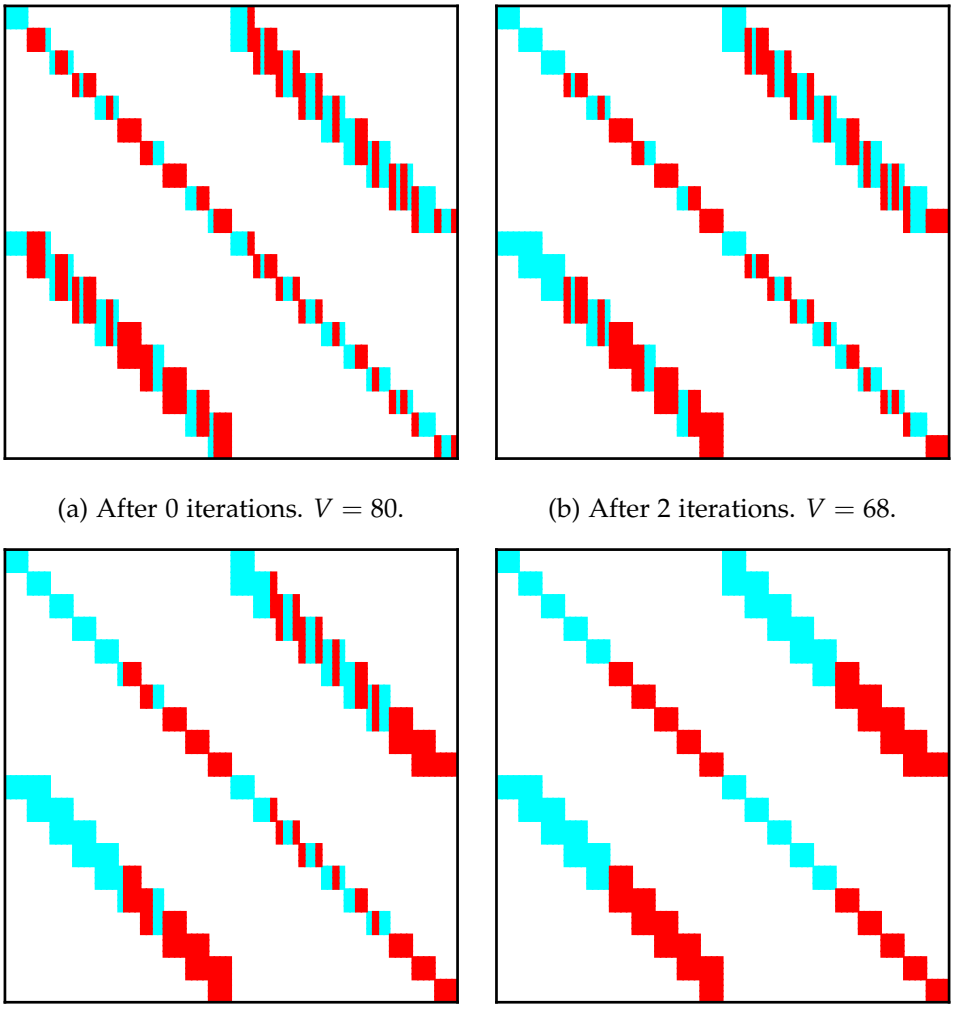


Figure 4.6.: Here we show spy plots of the best bipartitionings obtained for `cage7`. The row-net model was used for the partitioning. The communication volume that was obtained is $V = 111$.



(a) After 0 iterations. $V = 80$.

(b) After 2 iterations. $V = 68$.

(c) After 6 iterations. $V = 52$.

(d) After 7 iterations. $V = 8$.

Figure 4.7.: Here we show snapshots of the intermediate partitionings obtained when applying the Hyper-PuLP method to the `steam3` matrix. The number of iterations and the communication volumes are indicated below the figures.

called Hyper-PuLP. We also propose a new method in the form of a balancing scheme, for minimizing the cumulative runtime of a hypergraph partitioner, and a linear solver. Although this method is formulated such that it can be applied to any combination of solver and partitioner, the Hyper-PuLP method is particularly well suited for this balancing scheme because of its low complexity both in implementation and in runtime. We show that the Hyper-PuLP algorithm is capable of improving the communication volume of a distributed sparse matrix well beyond the baseline of a cyclic distribution.

4.10 RELATED WORK

The Hyper-PuLP partitioning method is based on the PuLP method by Slota, Madduri, and Rajamanickam [44]. Henne et al. have recently proposed a method based on label propagation on the associated clique- and star graphs of a hypergraph, to improve the coarsening phase in the multi-level method [27].

4.11 FUTURE WORK

The Hyper-PuLP partitioning method is a general and particularly flexible method to which many modifications could be made. For example, one may consider other options for the weight function and the way it should scale with respect to the size of a net. Also the weight function could be considered as an energy function in a Monte Carlo method, turning it into a randomized local search algorithm.

Currently, when the algorithm encounters a local minimum the algorithm terminates. An alternative would be to make random perturbations when this happens and let the algorithm continue by refining this perturbed partitioning.

Many optimizations could be made in the implementation of the algorithm. For example, since we only consider a finite set of points as input for the weight function, we could precompute the hyperbolic functions in these points to save computational costs. Also a queuing system could be used for updating the labels similar to the implementation described in [45].

A good recipe should be found for choosing the hypergraph model used. Here we chose the model that minimizes the cyclic communication volume, but this may not be feasible in practice.

We could also consider other hypergraph models beyond the one-dimensional models used in the results shown in this chapter. For example, a pseudo-2D hypergraph based on the medium-grain partitioning method could be used as input to the Hyper-PuLP method.

Optimizing the parallelization of this algorithm for distributed memory systems should make for particularly interesting future work.

The method that was discussed for balancing the time spent between partitioning and solving a linear system could benefit greatly from more stable iterative methods for solving linear systems. Furthermore, improvements could be made in the methods used to estimate $\hat{N}$, $\Delta\hat{V}$ and $\hat{t}$.

Part II

MATRIX ALGORITHMS FOR MANY-CORE ACCELERATORS

5

BULK SYNCHRONOUS STREAMING AND ALGORITHMS

Heterogeneous computing plays an important role in modern HPC (high performance computing). Many supercomputers that dominate the list of top-supercomputers in recent years, such as the Chinese Tianhe-I and Tianhe-2 supercomputers, as well as the Titan supercomputer at the Oak Ridge National Laboratory, use GPGPU (**g**eneral **p**urpose computing on GPUs) to obtain their high FLOP-rates. But not only GPUs are used, also for example FPGAs (field-programmable gate arrays) and many-core coprocessors are used to accelerate computations. In this section we will develop methods that target these *many-core coprocessors*, and propose a streaming framework within the BSP model which allows BSP algorithms to be generalized to these systems.

In Section 5.1 we discuss the Parallella, which is a small parallel computer that will serve as an hardware example to which we can apply the theory that we develop in this chapter. In Section 5.2 we discuss the implementation of the BSP model on the Parallella. This is based on joint work with Abe Wits who is currently a student at Utrecht University, and Tom Bannink who is a PhD candidate at CWI. Next we introduce pseudo-streams and BSP accelerators which lead to the BSPS model. We will then proceed by discussing a number of examples of algorithms that fit into the framework we introduced, and propose a new algorithm for performing a SpMV operation on an accelerator such as the Epiphany processor. We end this chapter with a discussion on the Epiphany processor as a BSP accelerator, and we introduce a micro-library for linear algebra on the Parallella that we developed based on the streaming extension to BSP we will discuss, and the Zee partitioning framework introduced in the previous chapter.

5.1 PARALLELLA AND EPIPHANY BSP

The Parallella is a small 'credit card-sized computer' which features two processors. It was launched after a successful crowd-funding campaign on

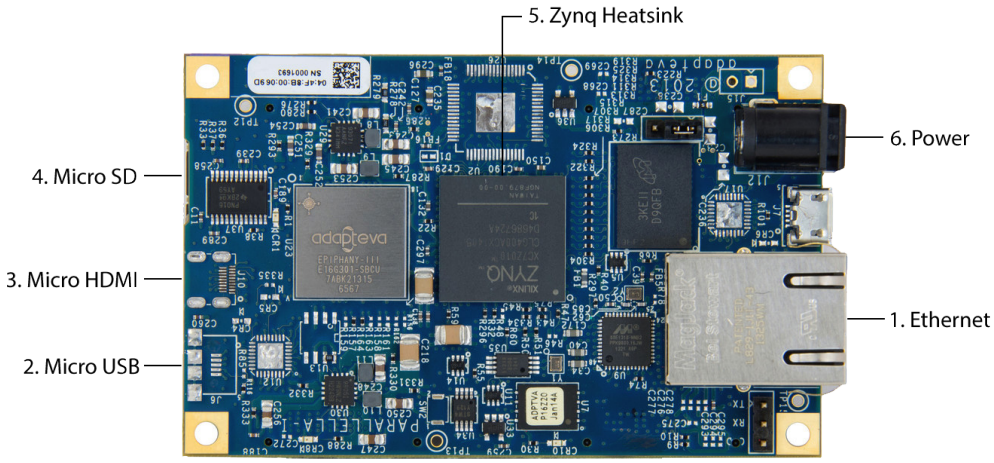


Figure 5.1.: An overview of the features of the Parallella board taken from the projects' website [31]. Here only the external connections are mentioned. Left of the center resides the Adapteva Epiphany processor, while in the center we have the Zynq ARM processor.

Kickstarter [1]. It is inspired by other small-form computing platforms such as the Raspberry Pi [2] and the Arduino [3], and the development boards are intended to make parallel programming accessible and open to a large community. Here we will introduce the specifics of the Parallella board in some detail, and the platform will serve as a motivation for the theory we develop in the remainder of this chapter.

The original Parallella board is a small computer that has basic network capabilities and also support for a number of peripherals, see Figure 5.1. There are two different processors available. The *host* processor, which runs the (Linux) operating system, is a dual-core ARM processor. The second *(co-)processor* is what makes the board special; it is a processor based on the Epiphany architecture which has 16 RISC (reduced instruction set computer) cores[4].

The Epiphany processor architecture [30] defines a square grid of cores of size $N \times N$. On the processor there is also a network-on-chip (NOC) present. There is support for single-precision floating point operations. The

1 Parallella: a supercomputer for everyone. `https://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone`

2 "The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing ..." `https://www.raspberrypi.org/`

3 "Arduino is an open-source electronics platform based on easy-to-use hardware and software." `https://www.arduino.cc/`

4 There has also been a limited production of Parallella boards with the Epiphany IV processor which has 64 cores. `http://www.adapteva.com/epiphanyiv/`

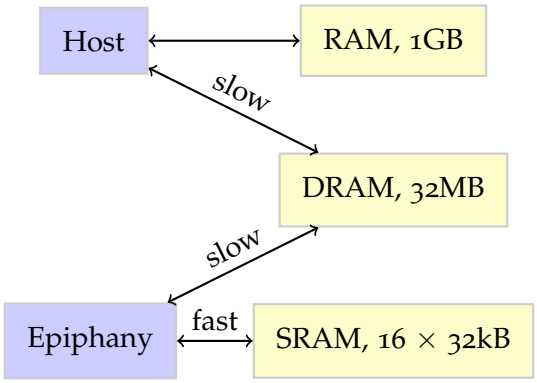


Figure 5.2.: Overview of the Parallella memory. We also give an indication of the speed of the different memory lanes available

chip supports core-to-core communication on the processor with very low latency (in the order of nanoseconds) and zero start-up costs.

We distinguish three layers of memory on the Parallella board. There is 1GB of RAM available, which is split into two parts; The largest part is exclusive to the host processor, and we will simply refer to it as RAM, and a relatively small section which is shared between the host and the Epiphany called the DRAM (or *dynamic* memory). Finally there is 32 kB of local memory present at each core, which we will refer to as the SRAM (or *static* memory). See also Figure 5.2. An important feature is the availability of two DMA (direct memory access) engines at each Epiphany core. These allow for asynchronous reading and writing among Epiphany cores, and between the Epiphany cores and the dynamic memory, and will play an important role in our implementation of pseudo-streaming algorithms.

5.1.1 *Epiphany BSP*

Epiphany BSP [5] (EBSP) is an implementation of the BSPlib standard [28] on top of the Epiphany SDK provided for the Parallella. It is released under the GNU lesser general public license, and was developed by Tom Bannink, Abe Wits, and Jan-Willem Buurlage. It provides a number of extensions to the BSPlib standard to accommodate for the dual-processor layout of the computer.

A typical Epiphany BSP application consists of two separate programs. The *host program* configures the application and prepares the data to be processed by the Epiphany coprocessor, and the *kernel* is a program that runs on each of the Epiphany cores in a SPMD manner. All the communication between Epiphany cores, and between the two processors can be done using the conventional BSP methods and syntax (e.g. buffered and unbuffered writes or through message passing mechanisms). A major goal

of the development of EBSP is to allow current BSP programs to be run on dual-processor hardware such as the Parallella with minimal modifications.

The Epiphany BSP library also provides many utilities to ease the development of BSP applications for the Parallella board, such as timers, dynamic memory management and debugging capabilities. Finally, EBSP also provides an extension to BSP to support streaming algorithms. We will introduce and formalize this extension in the next section.

5.2 STREAMING EXTENSION TO THE BSP MODEL

In this section we discuss streaming algorithms. This class of algorithms can be seen as processing methods for sequential data under typically two constraints:

Constraint 1. The computer executing the algorithm (or if you prefer, the algorithm itself) has limited (local) memory L available – typically much less than the total size S of the input, i.e. $L \ll S$. Here the *size* is taken as the number of floating-point numbers that can be stored, or of which the stream consists respectively.

Constraint 2. For each part of the input there is only a limited amount of processing time available.

An initial description of streaming algorithms has been given already in 1991 [26], but the first formal discussion was given in a 1999 article by Noga Alon, Yossi Matias, and Mario Szegedy [1]. Many streaming algorithms, in particular because of the second constraint, are massively parallel and often employ randomized methods to provide an approximation (typically called a *sketch*) of the answer. In the remainder of this section we will give a formal description of what we mean by a stream and a streaming algorithm, and will discuss how we can apply this to dual-processor systems such as the Parallella.

Definition 17. A *stream* is an ordered and finite collection of n tokens $\Sigma = (\sigma_1, \ldots, \sigma_n)$. Each token fits in the predetermined local memory, i.e. the size satisfies $|\sigma_i| < L$.

Many additional constraints can be set on streaming algorithms. For example, the data stream can be unbounded in size, or is not guaranteed to be presented in any predetermined order, or each token should be discarded or archived after a single pass; see e.g. [4, 35].

In particular, streaming algorithms usually refer to algorithms which only use the input a constant number of times, in many applications even only a single time. Here we will be much more lenient in the constraints we put on the algorithms, in particular we will *only* enforce the first constraint; that the

amount of local memory is severely limited $L \ll |\Sigma|$. Therefore some of the algorithms we will describe in the context of streams, for example Cannon's algorithm for dense-dense matrix multiplication, will not fit into the general streaming algorithm framework. The reason is that we are interested in streaming algorithms not because we want an approximate answer efficiently, but simply because the amount of local memory for the coprocessor is only sufficient to act on a small collection of tokens at once.

We will call algorithms *bulk-synchronous pseudo-streaming* (BSPS) algorithms if they have the following characteristics:

- The input is presented as (possibly multiple) *stream(s)* of tokens, and every processor core can only act on a single token at once (for each stream).
- The input is split into at least p streams Σ_s, one for each processor s. All these processors obtain their next tokens from their designated stream. This leads to algorithms that are naturally parallel.
- The processing of tokens occurs in a *bulk-synchronous manner*. The algorithms we describe here will be written in a SPMD manner, and we assert completion of the current pass over a token for each processing core before moving on to the next.
- Contrary to conventional streaming algorithms there is usually a strict order in which we process the tokens, and we are allowed to revisit tokens any number of times.

Many of the typical constraints that are put on streaming-algorithms are not enforced in what we consider *pseudo*-streaming. In particular we are free to reuse tokens any number of times, and furthermore we assume that we have random access to the tokens within the stream.

5.2.1 *BSP accelerators and hypersteps*

The BSP cost function provides a convenient method to predict the running time of a BSP program. We would like to be able to also predict the running time of a BSPS algorithm, but the architecture of the computers that we intend to run our algorithms on differ greatly from conventional BSP computers. Therefore we introduce a new kind of BSP computer which we will call a *BSP accelerator*. For a BSP accelerator we still define the two parameters g and l, the communication cost per word and the latency, and we also have a flop-rate local to each core of the accelerator. In addition we assume limited local memory L, but we have asynchronous external connections to an external memory pool of size $E \gg L$. We capture the bandwidth to this pool

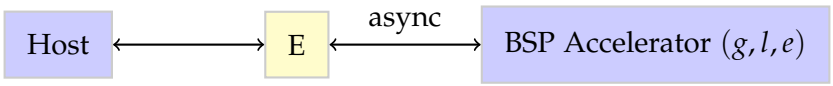


Figure 5.3.: Schematic overview of a BSP accelator.

with an additional parameter e, which is defined in flops per word similar to g.

A program for a BSP accelerator consists of a number of *hypersteps*, each of which consists of a number of ordinary BSP supersteps. In one hyperstep the sth processor only acts on a single token of each of the local streams Σ_s^i. During a hyperstep the code run can be seen as an ordinary BSP algorithm. After such a step, there is a global bulk-synchronization before every processor moves on to the next tokens of its local streams. The streams are prepared by an external processing unit which we will call *the host*.

Furthermore, we assume that there is an asynchronous communication mechanism with the external memory pool. During a hyperstep, the next token can (optionally) be requested, which we call *prefetching*, and is then written to a local buffer so that after a hyperstep we can immediately move on to the next token. This minimizes the down-time within hypersteps, and is reminiscent of cache prefetching techniques. Indeed one can view our discussion here as describing a software caching technique in a parallel environment. Note that prefetching data halves the maximum size of a single token, since we need to reserve storage for the buffer that holds the next token.

The main distinction between the classic BSP model and the model we consider here, is the asynchronous fetching from a memory bank of size E, that is prepared by a black box, which we have called the host. Extensions to the BSP model that specify parameters for the memory size have already been studied before, see e.g. [36, 46]. A recent development is Multi-BSP [48], introduced as a model for writing general and portable parallel algorithms for general multi-core systems. The Multi-BSP model is intended to be general and can be used to obtain a notion of parameter-independent optimality for parallel algorithms. For our application, the most distinguishing feature is prefetching data, and this is not incorporated in Multi-BSP. Contrary to e.g. the Multi-BSP model, our goal here is specifically to provide a framework for developing and implementing algorithms for a specific type of system (with a dual-processor layout) such as the hardware found on the Parallella board, and consequently we will keep our discussion to a relatively high level.

We are now ready to discuss the BSPS cost function. Consider a BSPS program consisting of H hypersteps. We view each hyperstep $1 \leq h \leq H$ as a separate BSP program, with an associated BSP cost function T_h. In every hyperstep (except for the first) we will prefetch the next token out of external

memory E with a certain bandwidth. We will denote the inverse of this bandwidth with e, which is measured in FLOPs per number of words. In this chapter we will define a data word to be equal to one *floating point number*, float for short. For simplicity we assume that every token has size C, and furthermore we assume that the first token is available for each core of the accelerator at the start of the program. The BSPS cost of a single hyperstep then corresponds to the maximum between the time spent processing the current token, which is equal to T_h, and the time taken to fetch the next token, which is equal to eC. We conclude that the cost function for a BSPS program should be:

$$\tilde{T} = \sum_{h=1}^{H} \max(T_h, eC). \tag{5.1}$$

If fetching the next token takes more time than processing the current token, we may say that the hyperstep is *bandwidth heavy*. Otherwise we say that the hyperstep is *computation heavy*.

5.3 EXAMPLES OF BULK-SYNCHRONOUS STREAMING ALGORITHMS

In this section we will discuss a number of algorithms that fit into the framework we described. First we will discuss algorithms for the inner-product and general dense-dense matrix multiplication. Finally we will propose a novel BSPS algorithm for dual-processor hardware for the SpMV operation.

5.3.1 *Inner-product*

As a simple example we will first consider two vectors $\vec{v}, \vec{u} \in \mathbb{R}^n$ of size n, and construct a BSPS algorithm to compute their inner product $\alpha = \vec{v} \cdot \vec{u} = \sum_{i=1}^{n} v_i u_i$. Here we assume that the total number of components $\vec{v}_i$ that can be stored at a single core is much smaller than the local memory L.

We begin by implicitly distributing the vectors over the processing cores of our BSP accelerator. In this discussion we will use a cyclic distribution so that $P_{\vec{u}}(i) = P_{\vec{v}}(i) = i \bmod p$. Next we need to partition the resulting data for the sth core, which we will take as the streams $\Sigma_s^{\vec{v}}$ and $\Sigma_s^{\vec{u}}$, into a number of tokens, each of which will fit in a designated chunk of local memory with a certain *chunk size* C, see also Figure 5.4.

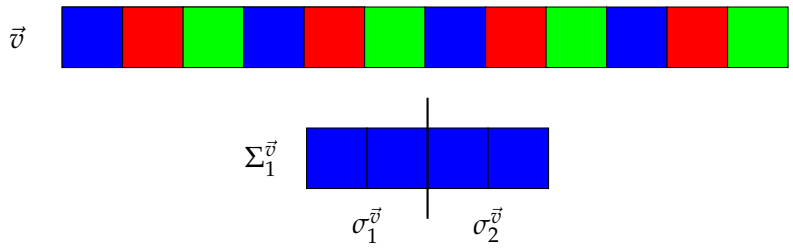


Figure 5.4.: Here we depict the construction of the streams $\Sigma_1^{\vec{v}}$ for $p = 3$ processors. Each token consists of $C = 2$ vector components, and the total stream size is $|\Sigma_1^{\vec{v}}| = 4$.

for all $1 \leq s \leq p$ **do** ▷ On host
  Prepare streams $\Sigma_s^{\vec{v}}$ and $\Sigma_s^{\vec{u}}$.

$\alpha_s = 0$ ▷ On accelerator core s
for all Token pairs $\sigma_i^{\vec{v}} \in \Sigma_s^{\vec{v}}$ and $\sigma_i^{\vec{u}} \in \Sigma_s^{\vec{u}}$ **do**
  $\alpha_s = \alpha_s + \sigma_i^{\vec{v}} \cdot \sigma_i^{\vec{u}}$

Send α_s to all (other) cores
$\alpha = \sum_{t=1}^{p} \alpha_t$

Algorithm 5.1: Summary of the BSPS algorithm for computing the inner product. After the completion of the algorithm every core of the accelerator will have computed the value $\alpha = \vec{v} \cdot \vec{u}$. This value can then be communicated back to the host.

Every core maintains a partial sum α_s throughout the algorithm. We consider each pair of tokens (both consisting of C vector components) and compute locally the inner product of this subvector and add it to α_s. After every token has been considered, the combined partial sums of all the processors will be equal to the desired value for the inner product α. Note that we can identify a token with a subvector, and we construct the streams for the two vectors in a completely identical manner. We summarize the algorithm in Algorithm 5.1.

Let us consider the BSPS cost of this algorithm. The total number of hypersteps is equal to $\frac{n}{pC}$, and after all the hypersteps have been performed an ordinary superstep is performed in which the sum of partial sums is computed. In each of these hypersteps we compute an inner product between two vectors of size C, taking $2C$ time, and this requires no communication.

We see that if $e > 2$ then the hypersteps are bandwidth heavy, otherwise they are computation heavy.

5.3.2 *Multi-level Cannon's algorithm*

Next we will consider a more elaborate example. One of the important operations in many applications is multiplying a matrix with another matrix. Here we will consider the product of two dense matrices, which are too large to fit completely in the local memory of the accelerator. There are a number of different parallel algorithms for general matrix-matrix multiplication, and many rely on the recursive nature of matrix multiplication. As an example we will first consider Strassen's algorithm. We can divide two matrices A and B of size $n \times n$ in four blocks of size $n/2 \times n/2$, so that the matrix multiplication $AB = C$ can be written as:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right).$$

This divides the problem of matrix multiplication into the multiplication of 8 pairs of smaller matrices. These products are completely independent and can thus be computed in parallel. We can subdivide these problems even further in the exact same manner if we want to use finer-grained parallelization.

We have divided the resulting matrix C into four blocks which we can write as C_{kl} with $1 \leq k, l \leq 2$ resulting in the system of equations:

$$C_{kl} = \sum_{i=1}^{n} A_{ki} B_{il}.$$

Strassen's algorithm relies on the observation that instead of performing these eight multiplications, we can find an equivalent system of equations that only requires seven multiplications of smaller matrices, for the cost of doing more additions.

The resulting (sequential) algorithm has a complexity of $\mathcal{O}(n^{\log_2 7})$, which is better than the straightforward implementation which has a complexity of $\mathcal{O}(n^3)$. Because of its recursive nature it is straightforward to parallelize Strassen's algorithm, however it not immediately clear how to generalize this algorithm to a BSPS algorithm, because the products that have to be computed in the recursion depend on higher levels making it hard to prepare streams without doing the actual computation. Therefore we will focus on an alternative algorithm called Cannon's algorithm [13], which we will describe in detail. As we will show, this algorithm is easily generalized to a streaming algorithm.

Cannon's algorithm

We will first describe the regular version of Cannon's algorithm which uses only a single level. We want to compute $AB = C$ for two matrices A and B. We assume we have $N \times N$ processors (for example, the Parallella board has $4^2 = 16$ processors). We index each processor core with a pair (s, t). We then split the matrices A and B in $N \times N$ blocks of equal size, so we write for example:

$$A = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \dots & A_{1N} \\ \hline A_{21} & A_{22} & \dots & A_{2N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{N1} & A_{N2} & \dots & A_{NN} \end{array}\right),$$

so that we can write for the resulting blocks of C:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \qquad 1 \leq i, j \leq N.$$

We see that the resulting block C_{ij} is the result of adding N terms, in each of which a block of A and a block of B are multiplied. Since there are exactly $N \times N$ blocks C_{ij}, and $N \times N$ processors, it would be very natural to let each processor compute exactly one of these sums in N steps. However there is one immediate problem: many blocks of A and B are needed simultaneously in the same step k, and we do not want to copy our blocks to every single processor since we may assume that there is finite storage, and therefore limited room for duplication. It turns out that we can rearrange the sum above so that in step k each processor requires a unique block of A and B, so that we never have any data redundancies. After computing a term, the matrix blocks that were used can be moved around to the processor that needs the block next.

Let us derive this reordering here. Without rearranging the processor indexed with (s, t) will compute the product $A_{sk} B_{kt}$ in the kth step. If we consider the sth *row* of processors, we see that they all require the block A_{sk} in the kth step. Similarly the tth *column* of processors require the block B_{kt} in the kth step. We might get the idea that we could let the processor index with (s, t) compute the $(s + t)$th term first (where we identify the $N + 1$th term with the first one), looping around all the way periodically until we compute the $(s + t - 1)$th term. In summary we could let the processor (s, t) compute the product:

$$A_{s,1+(t+s+k-3) \bmod N} B_{1+(s+t+k-3) \bmod N,t}$$

in the kth step[5]. After careful inspection we see that indeed each processor works on different blocks each step, because this scheme solves simultane-

5 Note that the indices would be much more straightforward had we used 0-based indices, but we will stick with 1-based indices in this discussion for consistency.

ously the similar issues of processors in the same row or column requiring the same block. The next step to consider is to see which processor needs the blocks of the current step after the processor is done with it. In the $(k+1)$th step, the processor (s,t) needs the A block that was previously owned by processor $(s, 1+(s+t+k-3 \bmod N))$ in the previous step, while the B block was previously owned by $(1+(s+t+k-3 \bmod N), t)$. We then come up with the following scheme:

1. Perform an initial distribution of the matrix blocks over the $N \times N$ processors, sending $A_{i,j} \mapsto (i, 1+((i+j-2) \mod N))$ and $B_{i,j} \mapsto (i, 1+((i+j-2) \mod N))$.

2. (Repeat N times:) Let each processor compute the product of the two local matrix blocks of A and B, adding the result to C_{st}.

3. Next each processors sends the matrix block of A to the right, i.e. to processor $(s, 1+(t \bmod N))$, and each matrix block of B down to processor $(1+(s \bmod N), t)$.

The resulting matrix product C will then be available distributed over the processors.

Multi-level Cannon's algorithm

We will now generalize this algorithm to a BSPS variant. The method we discuss here is similar to the one described in e.g. [42]. The distribution scheme that we derived in the previous section will not suffice in general for BSP accelerators, since for reasonably large matrices, dividing them into $N \times N$ blocks will not make them small enough so that they can be stored in the local memory of the cores. Thus, we need to reduce these sub-problems in size even further. We do this by subdividing the matrix in two levels. The first level will no longer consist of $N \times N$ blocks, but of $M \times M$ blocks, where M is taken suitably large. Each of these blocks will be divided further in $N \times N$ blocks, which will be distributed over the cores in the method described above. For example A will now look like this:

$$A = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \dots & A_{1M} \\ \hline A_{21} & A_{22} & \dots & A_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{M1} & A_{M2} & \dots & A_{MM} \end{array}\right), \; A_{ij} = \left(\begin{array}{c|c|c|c} (A_{ij})_{11} & (A_{ij})_{12} & \dots & (A_{ij})_{1N} \\ \hline (A_{ij})_{21} & (A_{ij})_{22} & \dots & (A_{ij})_{2N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline (A_{ij})_{N1} & (A_{ij})_{N2} & \dots & (A_{ij})_{NN} \end{array}\right).$$

In total we then have $MN \times MN$ blocks. We can choose our value of M such that the resulting smaller blocks $(A_{ij})_{kl}$ are small enough to fit on the local memory for the processors.

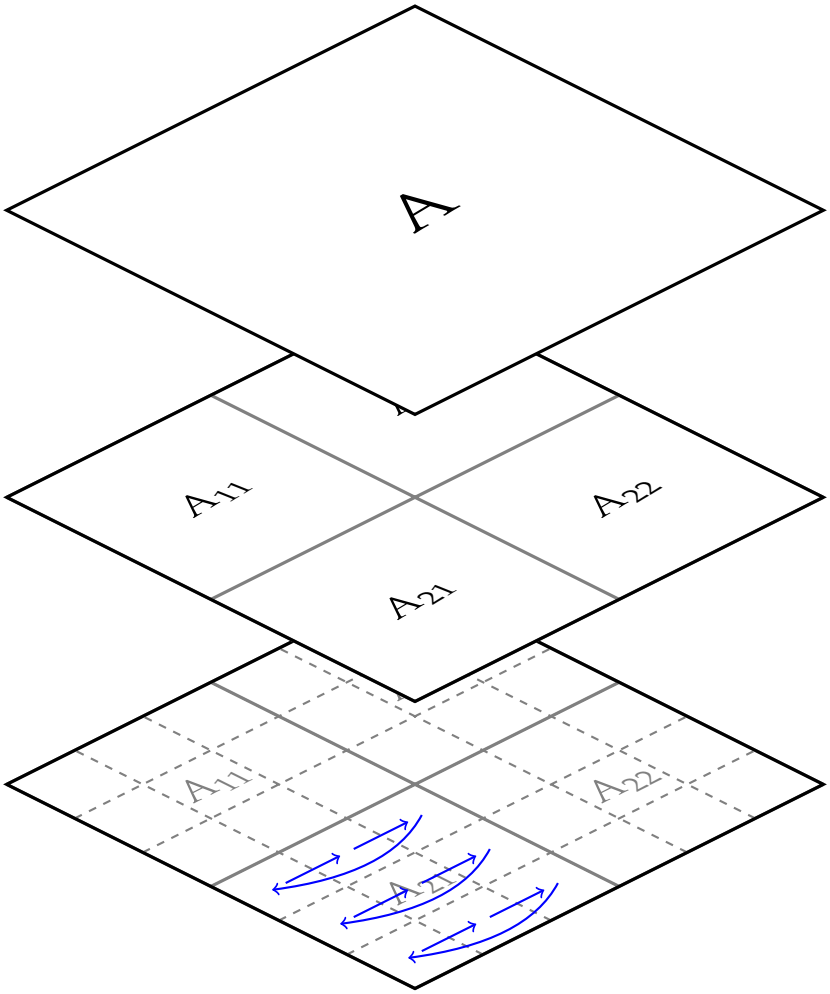


Figure 5.5.: The three layers of our modified Cannon's algorithm. The top layer is the matrix A, the second layer is the matrix A sliced in $M \times M$ parts, here we have $M = 2$. Finally at the bottom layer we have that each of the M^2 blocks has been divided into $N \times N$ parts with $N = 3$. We indicate the communication direction of the inner blocks for a single outer block in blue, which is *horizontal* for the matrix A, and would be *vertical* for the matrix B.

Let us now turn our attention to constructing the streams. We will consider the M^2 blocks of A in row-major order, and the M^2 blocks of B in column-major order. The blocks will form the tokens, and will all be considered M times. In every hyperstep we will compute the product of two blocks using Cannon's algorithm introduced above. To construct the stream, we will denote with e.g. $(A_{ij})^0_{st}$ (where $1 \leq i, j \leq M$ denote the *outer blocks* and $1 \leq s, t \leq N$ denote the *inner blocks*) the first inner block that the pro-

cessor (s,t) receives when considering the token A_{ij}. We define $(B_{ij})^0_{st}$ in a similar manner. We are now ready to define the streams:

$$\Sigma^A_{st} = \underbrace{(A_{11})^0_{st}(A_{12})^0_{st}\dots(A_{1M})^0_{st}}_{\circlearrowleft M \text{ times}}\underbrace{(A_{21})^0_{st}(A_{22})^0_{st}\dots(A_{2M})^0_{st}}_{\circlearrowleft M \text{ times}}\dots\underbrace{(A_{M1})^0_{st}(A_{M2})^0_{st}\dots(A_{MM})^0_{st}}_{\circlearrowleft M \text{ times}},$$

and

$$\Sigma^B_{st} = \underbrace{(B_{11})^0_{st}(B_{21})^0_{st}\dots(B_{M1})^0_{st}(B_{12})^0_{st}(B_{22})^0_{st}\dots(B_{M2})^0_{st}(B_{13})^0_{st}\dots(B_{1M})^0_{st}(B_{2M})^0_{st}\dots(B_{MM})^0_{st}}_{\circlearrowleft M \text{ times}}.$$

Here, we denote with $\circlearrowleft$ the order in which we consider the tokens, so that $\circlearrowleft M$ means that we will repeat looping over that particular section of blocks M times before moving on to the next section of blocks. Note that each block is only stored in the stream once. We will loop over groups of M blocks of A a number of M times before moving to the next, while we simply loop over the M^2 blocks of B a total number of M times.

After constructing these streams, from the perspective of an accelerator we have to multiply the two tokens, corresponding to the outer matrix blocks, given to us in each of the M^3 hypersteps. This is done by computing the product of the two outer blocks with the ordinary Cannon's algorithm, which can now be applied since we have chosen the outer blocks to be of small enough size, the result of which is added to the block C_{ij} that is currently being computed. After every M hypersteps we have completely computed one of the M^2 blocks of C, and we store the result in the (large) external memory E.

Let us consider the BSPS cost of this algorithm. First we will derive the BSP cost of Cannon's algorithm. There are N supersteps in which we compute the product of two inner blocks of size $k \times k \equiv \left(\frac{n}{NM}\right) \times \left(\frac{n}{NM}\right)$, which takes $2k^3$ flops. Next we send and receive such an inner block consisting of k^2 words. In fact, we do not send or receive such a block in the final superstep, but for simplicity we will ignore this. Then the BSP cost equals:

$$T_{\text{cannon}} = N(2k^3 + k^2 g + l).$$

The number of values in a token, the chunk size C, is given by the number of values in an inner block which is equal to k^2. For simplicity, we will ignore the costs of storing the resulting blocks. There are M^3 hypersteps, so that we have arrived at the following BSPS cost for this algorithm:

$$\tilde{T}_{\text{cannon}} = M^3(\max(N(2k^3 + k^2 g + l), ek^2)).$$

5.3.3 *Streaming implementation of SpMV*

The final BSPS algorithm we consider is a localized version of the parallel SpMV algorithm we have discussed in Chapter 2. Multiplying a sparse ma-

trix with a vector is a particularly bandwidth dependent operation. What we mean with this is not only that its running time is linear in the size of the input (the nonzeros of the matrix), but also that the constant factor is very low. Indeed, each element is only involved in two FLOPs; one multiplication with a component of $\vec{v}$, and the addition of the result to a component of $\vec{u}$. This can be made concrete with the FLOPs/byte metric, which is 0.25 for SpMV, i.e. two flops per floating-point number, which is a very low number. Because of this, improvements in the asynchronous reading speed from the external memory pool can provide large benefits for this algorithm.

Specific algorithms have been developed for treating sparse matrices in heterogeneous environments or to make the algorithm cache-friendly. In particular many specialized algorithms have been developed for treating the SpMV problem. A number of these modify the matrix to create e.g. sparse blocks. Here we present an alternative method, that does not require the explicit modification of the original matrix.

Strips and windows

Again we assume that the working memory at any given time is much smaller than the total storage requirements for the entire matrix. In other words, we assume that we do not have random access in the entire matrix and input/output vectors, but rather at small *chunks* of the total input at a time. In particular we want to focus on a given part of the input and output vectors at a time. Say we are focusing on a limited number of vector components v_i. After (implicit) permutation of the input vector, we see that this limits the corresponding current portion of interest in the matrix to a single *strip* which can be seen as a consecutive multi-column of the matrix. Similarly, when we require in addition that only a small part of the output vector is being written to at a time the current section of interest in the matrix is limited to a *window* within the previously mentioned strip.

To prepare the matrix for processing by the accelerator, we thus first decompose the matrix in a number of multi-columns. After reordering the columns of the matrix, we may assume that these multi-columns consist of consecutive columns and form a number of strips. Each of these strips are split (independently) in a number of windows, depending on the size of the matrix and the amount of local storage available. See also Figure 5.6.

A good partitioning then reduces to finding a good selection of strips and windows within these strips. Furthermore, we require good balance within these windows and strips. The vector components $\vec{v}$ are distributed as well, and we want to minimize the amount of communication within a strip. Furthermore we want to reduce the number of vector components u_j that a given core writes to. This leads to an entirely new partitioning problem that is related to the problems we have treated so far. Here we will focus mainly

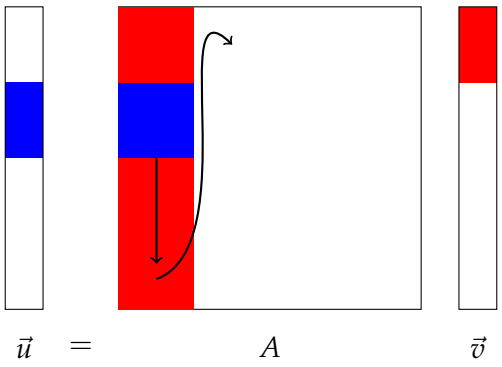


Figure 5.6.: Strips and windows in our streaming SpMV algorithm. We split the matrix A into a number of vertical strips. Here we depict the first strip and the corresponding part of $\vec{v}$ in red. Within this strip we consider windows; these windows and the corresponding part of $\vec{u}$ are depicted in blue. Here we choose the windows to start at the top, and move down until we have considered the entire strip.

on the method and its implementation itself. A straightforward partitioning method would be to partition the matrix in $M \times M$ blocks, much like we did for Cannon's algorithm. We would then end up with M strips, and M windows within these strips. From now on we assume that a partitioning of A into strips and windows, and a partitioning of $\vec{v}$ has been given. We ignore the addition of the partial sums, and will assume that this is done by the black box host after the conclusion of the algorithm.

Constructing the stream

Again we turn our attention on defining the streams and their tokens. There is one big difference with the algorithms that were treated before, and that is that the tokens are inhomogeneous. Not only can they be of different sizes, but we also require completely different information when e.g. we start considering a new strip or a new window. We will therefore make a distinction between *header tokens* and *content tokens*. The header tokens will contain information for windows, strips or even the entire algorithm. The content tokens will represent the components of the input vector and the entries of the matrix that should be considered next. In the discussion below we mix the strip header and the vector content token to reduce the number of different tokens that have to be considered. The remainder of this section will be devoted to defining these tokens, and we will go into detail how we define them.

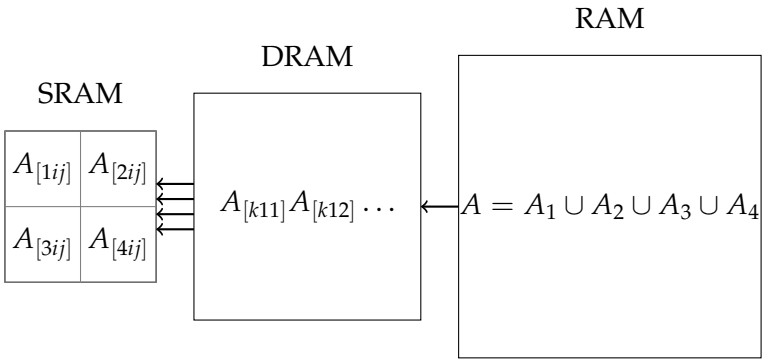


Figure 5.7.: Here we give an overview of the constructed streams for the matrix A, for $p = 4$ processors. The streams are constructed in the shared DRAM, while each of the windows are sent separately to the cores and put in the local SRAM.

We construct a single stream, in which the information for both the matrix Σ_s^A and the input vector $\Sigma_s^{\vec{v}}$ is stored. We will have three kind of *header tokens*, one *global* header, a number of *strip* headers, and a larger number of *window headers*. We will have a single kind of content token, a *window content token* containing the matrix elements inside the current window. We will discuss each of these in turn:

1. The **global header** will contain information about the maximum number of vector components of $\vec{v}$ the processor will be considering at any given time over all the strips. Similarly, it will contain the maximum number of $\vec{u}$ components the processor will write to. It will also contain the maximum number of nonzeros assigned to this processor with a window, the maximum strip width, and the number of strips. This way the accelerator core can preallocate the required memory once for the entire algorithm.

2. The **strip header** will contain the number of windows within this strip, the number of local vector components $\vec{v}$ for the strip, and the values of these components.

3. The **window header** contains the number of nonlocal vector components of $\vec{v}$ that are required for this window, the owners and remote indices of these components, and the number of nonzeros within this window.

4. The **window content token** contains the nonzeros of the current window using a user-defined storage mechanism.

An important part of implementing these streams efficiently is using appropriate *local indices* for the values $A_{\mathtt{ij}}$, $v_{\mathtt{i}}$ and $u_{\mathtt{j}}$. We will number the local

input vector components for each processor within a strip with numbers $\mathtt{i} \in \{1, \ldots, k\}$. The nonlocal components v_i vary from window to window, and will be given the indices $\mathtt{i} \in \{k+1, \ldots, l\}$, so that the vector components that we have to obtain can be stored consecutively. For each window we will number each column that is non-empty for any given processor with $\mathtt{j} \in \{1, \ldots, m\}$. We will therefore preprocess each window to store the triplets with indices corresponding to this choice of local indices. Note that this does not depend on the actual values of $\vec{v}$ such that this only has to be done once, and can then be reused for different input vectors. Every time we finish processing a window content token, we store the resulting partial subvector of $\vec{u}$ in the external memory E.

Note that it is possible to choose the windows independently not only for each strip, but also for each processor, since all the non-local information stays the same while processing the current strip.

Horizontal strips

In the discussion above we have taken our strips vertically. This leads to a lot of communication for the partial results of the output vector $\vec{u}$. Indeed, there can be a large number of strips, and for each strip there may be a contribution to any vector component u_j. This leads to a large storage requirement unless we spend time adding partial results, which would require an additional read and write from/to the external memory E. However, this choice does mean that we only consider any given vector component in a single strip, which means that communication for $\vec{v}$ will be relatively efficient. An alternative choice is to use horizontal strips. This way, we no longer consider vector components only once. However, after we are done with a strip we know the complete result for a part of $\vec{u}$, eliminating the need to add the partial results during or at the end of the algorithm. What method is the fastest depends ultimately on the parameters of the accelerator, as well as the sparsity pattern of the matrix.

Partitioning for the BSPS SpMV algorithm

Here we will mention some considerations for finding a good partitioning of the matrix for the BSPS SpMV algorithm discussed above. A good partitioning is crucial for the running speed of the algorithm. The main constraints of the partitioning are that:

- The strips can not be too wide, since we only have room for a limited part of the input vector $\vec{v}$.
- Windows cannot be too high such that we limit the number of vector components $\vec{u}$ that are written to.

Instead of optimizing for the total communication volume, as is the case in the general parallel SpMV algorithm, here we should optimize for the combined communication that occurs because of our choice of windows. There are at least two approaches one can take. The first is to first find a good global partitioning that reduces the communication volume for the general algorithm, such as those discussed in Chapter 3; and finding a good selection of strips and windows while keeping the vector/matrix distribution fixed. A better approach would be to simultaneously partition the matrix and select the strips and windows. This is an area that is still to be explored, and would make for exciting future work.

5.4 THE EPIPHANY PROCESSOR AS A BSP ACCELERATOR

As a concrete example of a BSP accelerator we will consider the Epiphany-III chip that is found on the original Parallella board. This chip has a grid of 4×4 cores which each run with a default clock rate of 600 MHz. As we mentioned when we introduced the Parallella in Section 5.1, the Epiphany chip is connected to a portion of memory called the DRAM which we will take as our external memory E, and each core comes equipped with a DMA engine which gives us an asynchronous connection to this memory pool.

There are many possible communication paths between the host, the Epiphany and the various kinds of memory. We are interested in estimating as accurately as possible the inter-core communication speed g, the latency l, and the read/write speed e from an Epiphany core to the external memory using the DMA-engine.

Actor	**Network state**	**Read**	**Write**
Core	busy	8.3 MB/s	14.1 MB/s
	non-busy	8.9 MB/s	270 MB/s
DMA	busy	11.0 MB/s	12.1 MB/s
	non-busy	80.0 MB/s	230 MB/s

Table 5.1.: These communication speeds were obtained from measurements done during the development of Epiphany BSP. In the *network state* column we indicate if a single core is reading/writing (non-busy) or if all cores are reading/writing simultaneously (busy). All the speeds are given *per core*.

We summarize the results of a number of measurements that were done during the development of Epiphany BSP in Table 5.1. From this we can estimate e. Note that there is a significant difference between the read and write speeds when multiple cores are communicating with the external memory at the same time. We will choose to use the most pessimistic number, the read speed using the DMA engine from the external memory with a busy

network state, since we expect that all cores will simultaneously be reading from the external memory during a hyperstep. We arrive at an external inverse bandwidth of:

$$e \approx (11 \text{ MB/s})^{-1} \approx 217 \text{ FLOP/float},$$

where we used that an Epiphany core runs at a default frequency of 600 MHz. Also we use single-precision floats which have a size of 4 bytes. Note that this value for e is rather high, which means that we need to do a large number of FLOPs with every floating point number we obtain or the time of a hyperstep will have this bandwidth as a bottleneck. This is an obvious limitation of the Parallella board, and is specific to this computer. We note that this high value for e is not a general property of the Epiphany chip (nor any other BSP accelerator).

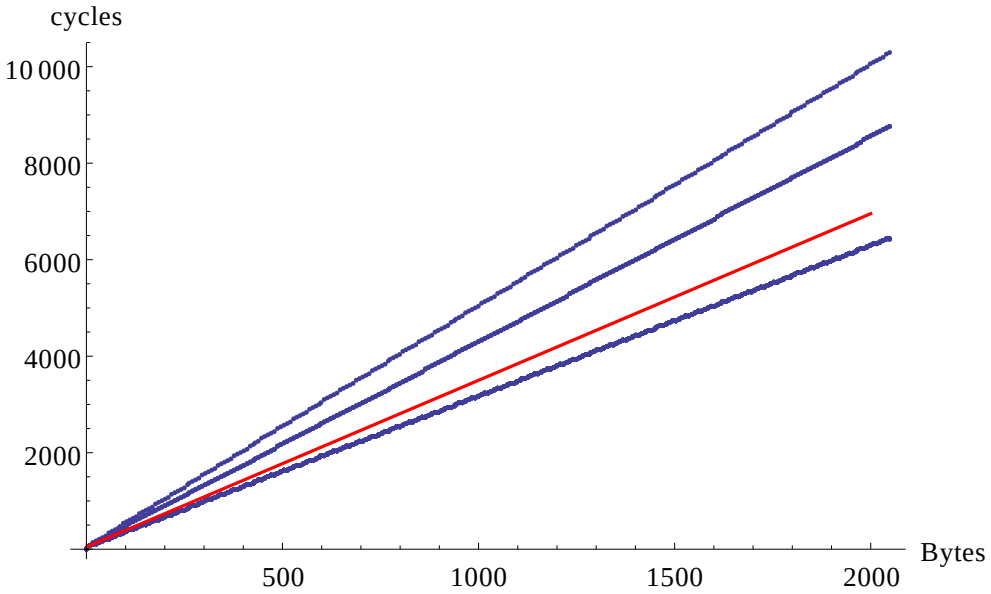


Figure 5.8.: In this figure we depict the number of clock cycles on the vertical axis as a function of the number of bytes sent on the horizontal axis. These numbers are for reading directly from the local memory of the next logical Epiphany core by a given Epiphany core. In blue the raw measurements are shown. There are three blue lines corresponding to different physical distances between Epiphany cores that are logical neighbours. The red line denotes a linear fit against all this data. Note that there are more data points in the lowest line, causing the fit to be closer to this line. From this fit we obtain the values for g and l. Alternatively to the method we used, one could use all-pair communication to determine these parameters.

For g and l we fit a linear function against the raw measurements that were obtained for core-to-core reads for a varying number of bytes. It turns out that the Epiphany hardware is such that this specific type of communication does not suffer from the large discrepancies between simultaneous and non-simultaneous communication of multiple cores. The result of this fit is depicted in Figure 5.8. After compensating for overhead because of the hardware clock that was used to perform the measurements, we obtain $l_r \approx 1.6$ FLOP for the latency, and $g_r \approx 13.8$ FLOP/float for the inter-core communication speed for reading[6].

Analogous measurements for writes yield: $l_w \approx 5.6$ FLOP for the write latency, and $g_w \approx 5.0$ FLOP/float for the write speed.

We will assume that a BSP program spends an equal amount of time reading and writing (although on this platform one should prefer to use writes to obtain the best possible speeds), so that the final values for g and l will be taken as the average of the values we obtained for reading and writing:

$$l \equiv \frac{l_r + l_w}{2} \approx 3.6 \text{ FLOP},$$
$$g \equiv \frac{g_r + g_w}{2} \approx 9.4 \text{ FLOP/float}.$$

We have implemented the algorithms discussed in this chapter in Zephany[7], a micro-library for the Parallella based on Zee and Epiphany BSP. In the future this library could be modified to use the BSPS cost function to decide whether it is worthwhile to accelerate a given operation using the Epiphany, without requiring user-intervention to make this choice.

5.5 SUMMARY

We have introduced a modification to the BSP model which is more suitable for many-core coprocessors. The specifics of these processors lead us to propose the definition of a BSP accelerator, and made us consider what we called bulk-synchronous pseudo-streaming algorithms which are able to run on these accelerators. We gave three examples of these algorithms. Finally we considered specifically the Epiphany chip as an example of a BSP accelerator.

6 Starting and stopping the hardware clock takes a specific, fixed number of clock cycles. We also used that the Epiphany is able to perform one FLOP per clock cycle. We do note however that there is support on this platform to perform the equivalent of two FLOPs per clock cycle when performing multiplications and additions in succession, but we will not make use of this to preserve generality, such that the values are valid for any BSPS algorithm.

7 `http://www.github.com/jwbuurlage/Zephany`

5.6 FUTURE WORK

There is still a wide range of algorithms and applications that we have not considered in this context. As an example, we can also imagine the BSPS cost function to apply to the processing of a video feed, where a frame is analyzed in each hyperstep. Here we could require the hypersteps to be bandwidth heavy to ensure that we are able to process the entire video feed in real-time.

We can also consider other kinds of accelerators, beside many-core co-processors, such as GPUs or FPGAs, and see if they would fit within this framework after the necessary modifications. Finally, we can also consider models in which we have different types of processing units, and develop a model that uses the BSP and BSPS costs to distribute the work of a single algorithm in this heterogeneous environment.

5.7 ACKNOWLEDGMENTS

I would like to thank Tom Bannink and Abe Wits for their work on Epiphany BSP, and their contribution to the early stages of a streaming extension to the BSP model. I would also like to thank Adapteva for providing part of the hardware that was used for the development of the relevant software, and so that we were able to perform the measurements presented in this chapter.

Part III

APPENDIX

A

KRYLOV SUBSPACE METHODS

One popular category of linear solvers are so-called Krylov subspace methods [50]. Here we will describe the basic theory, and focus in particular on a minimum residual method called GMRES. We will see that the important operations are SpMV and the dot-product, both of which can be parallelized.

First let us specify what we mean by a *linear solver*. Say we are given a matrix $A \in \mathcal{M}_{m\times n}$ and a vector $\vec{b} \in \mathbb{R}^m$ (throughout this research we focus on problems with real coefficients). We want to find a vector $\vec{x} \in \mathbb{R}^n$ such that

$$A\vec{x} = \vec{b}. \tag{A.1}$$

For now we will assume $m = n$, but the methods we develop also generalize to general rectangular matrices. Finding an exact solution to this can be done by using algorithms like Gaussian elimination or specialized algorithms such as Cholesky Decomposition, which run in $\mathcal{O}(n^3)$ time (see e.g. [24]). However, for large matrices finding the exact solution is infeasible. Furthermore the Gaussian elimination process does not exploit the sparseness of a matrix, and instead causes fill-in in the matrix – which is something we want to avoid.

To accommodate solving the linear problem for large and/or sparse matrices, heuristic methods were developed that approximate the vector $\vec{x}$ in $\mathcal{O}(n^2)$ time. Krylov subspace methods rely on the Cayley-Hamilton theorem.

Theorem 3 (Cayley-Hamilton)**.** *Let $A \in M_{n\times n}$, and let $p(\lambda) = \det(\lambda Id - A)$ be the characteristic polynomial of this matrix. Then A is a zero of the corresponding polynomial $p : M_{n\times n} \to M_{n\times n}$.*

A direct corollary of this theorem is that the inverse of a matrix A is given by a polynomial p in A of degree $k \leq n$:

$$A^{-1} = p(A) = c_0\mathrm{Id} + c_1A + c_2A^2 + ... + c_kA^k.$$

Indeed, if $p(A) = 0$ then, we have $d_1A + d_2A^2 + ... + d_kA^k = -d_0\mathrm{Id}$. We then factor out A in the left-hand side and divide by $-d_0$ to find $Aq(A) = \mathrm{Id}$ for some polynomial q, such that indeed $A^{-1} = q(A)$.

Note that the exact solution to Equation A.1 is given by $\vec{x} = A^{-1}\vec{b}$. This means in particular that

$$\vec{x} \in \text{span}\{\vec{b}, A\vec{b}, ..., A^k\vec{b}\}.$$

If the matrix A is such that higher powers of A become negligible, i.e. $||A^k\vec{b}||$ is small for high values of k, then we can try to approximate $\vec{x}$ by using only small powers of A. This observation shows the usefulness of the definition of Krylov subspaces.

Definition 18 (Krylov subspace)**.** The kth Krylov subspace corresponding to a matrix $A \in M_{n\times n}$ and a vector $\vec{y} \in \mathbb{R}^n$ is the space:

$$\mathcal{K}_k(A, \vec{y}) = \text{span}\{\vec{y}, A\vec{y}, \dots A^{k-1}\vec{y}\}$$

These subspaces give rise to (iterative) Krylov subspace methods. These methods all follow (roughly) the following scheme:

1. First an initial vector $\vec{x}_0$ is chosen.
2. The following procedure is repeated for each vector $\vec{x}_i$: compute the *residual* $\vec{r}_i = \vec{b} - A\vec{x}_i$.
3. Find our next guess $\vec{x}_{i+1}$ in the Krylov subspace $\mathcal{K}_i(A, \vec{r}_0)$ according to some notion of optimality, e.g. minimizing the residual.

Besides the residual we can also define the *error* ϵ_i as $\epsilon_i = \vec{x} - \vec{x}_i$. We can view the norm of the error as a measure for convergence in the domain of A, and the norm of the residual as a measure of convergence in the range of A. With this notation we can write

$$A(\vec{x}_i + \epsilon_i) = \vec{b}, \qquad \text{or equivalently} \qquad A\vec{x}_i = \vec{b} - \vec{r}_i.$$

As an example of an iterative scheme we will discuss the simplest example here, which is known as Richardson iteration. Note that we can rewrite the first equation above as:

$$A\epsilon_i = \vec{b} - A\vec{x}_i.$$

Assuming that A is close to the identity matrix $A \approx \text{Id}$, i.e. for example $||A - \text{Id}|| < 1$, it is natural to guess $\epsilon_i = \vec{b} - A\vec{x}_i$. This leads to the iterative scheme:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{b} - A\vec{x}_i = \vec{x}_i + \vec{r}_i$$

Let us see how the residual $\vec{r}_i$ behaves as i increases by rewriting it in terms of the initial residual $\vec{r}_0$

$$\begin{aligned}\vec{r}_i &= \vec{b} - A\vec{x}_i = \vec{b} - A(\vec{x}_{i-1} + \vec{r}_{i-1}) \\ &= (\vec{b} - Ax_{i-1}) - A\vec{r}_{i-1} = (\text{Id} - A)\vec{r}_{i-1} \\ &= (\text{Id} - A)^i\vec{r}_0,\end{aligned}$$

such that for the norm of $\vec{r}_i$ we obtain

$$||\vec{r}_i|| = ||(\mathrm{Id} - A)^i \vec{r}_0|| \leq ||(\mathrm{Id} - A)^i|| \cdot ||\vec{r}_0|| \leq ||\mathrm{Id} - A||^i \cdot ||\vec{r}_0||.$$

As before, if A is close to the identity we have $||\mathrm{Id} - A|| < 1$ such that $\vec{r}_i$ indeed converges to zero, as we desire.

If A is not close to the identity there is a very convenient method we can apply such that we can still use a similar scheme. This relies on a matrix M, a so-called *preconditioner*. The requirement on M is that it is a matrix that is easily invertible, relative to A, which approximates A in the sense that $M^{-1}A$ is close to the identity. In particular we require that $||\mathrm{Id} - M^{-1}A|| < 1$, because then we can apply M^{-1} to both sides of Equation A.1, to see that it converges with the previous argument.

A.1 CHOOSING OPTIMAL VECTORS FROM THE SUBSPACE

For the vectors found using Richardson iteration we have

$$\vec{x}_i = \vec{x}_{i-1} + \vec{r}_i = \vec{x}_0 + \sum_{k=0}^{i-1} \vec{r}_k = \vec{x}_0 + \sum_{k=0}^{i-1} (\mathrm{Id} - A)^k \vec{r}_0,$$

so that in particular if we choose $\vec{x}_0 = \vec{0}$ we have:

$$\vec{x} \in \mathrm{span}\{\vec{r}_0, A\vec{r}_0, \ldots, A^{k-1}\vec{r}_0\} = \mathcal{K}_k(A, \vec{r}_0).$$

However, it may not be the *best* vector to approximate the answer in this subspace. Therefore, algorithms have been invented that lead to better choices of vectors $\vec{x}_i$, that are optimal in one of the following senses:

1. Minimum residual. The residual has minimal Euclidean norm:

$$\vec{x}_k = \mathrm{argmin}_{\vec{y} \in \mathcal{K}_k(A, \vec{r}_0)} ||\vec{b} - A\vec{y}||_2. \tag{A.2}$$

2. Minimum error. The kth vector is taken from a different Krylov subspace, $\vec{x}_k \in A^T \mathcal{K}_{k-1}(A^T, \vec{r}_0)$, such that the error

$$||\vec{x} - \vec{x}_k||_2$$

is minimal.

3. Ritz-Galerkin. The residual $\vec{r}_k$ corresponding to $\vec{x}_k$ is orthogonal to the current Krylov subspace:

$$\vec{r}_k \perp \mathcal{K}_k(A, \vec{r}_0).$$

4. Petrov-Galerkin. Let $B \subset \mathbb{R}^n$ be some other suitably chosen k-dimensional subspace. The residual should be orthogonal to this space:

$$\vec{r}_k \perp B.$$

In the remainder of this section we will focus on a method that gives optimal answers in the sense of Equation A.2 called GMRES, but many other methods have been developed for each of these categories.

A.2 GMRES

An important Krylov subspace method is GMRES [43], for **G**eneralized **M**inimum **RES**idual. It requires A to be non-singular, non-symmetric and square – but it can be generalized to include other systems too. It works by generating a well-conditioned basis for the relevant Krylov subspaces using an orthonormalization phase called the Arnoldi process. In this process an upper Hessenberg matrix H is formed to which we can apply an implicit QR decomposition. Using a least-squares approach we obtain a vector that satisfies Equation A.2. Here we will introduce this method, and analyze the basic linear algebra operations that the resulting algorithm requires.

A.2.1 *Arnoldi process*

The Krylov subspace $\mathcal{K}_i(A, \vec{r}_0)$, in which we find our intermediate vector $\vec{x}_i$, is spanned by the vectors $\vec{r}_0, A\vec{r}_0 \dots A^{i-1}\vec{r}_0$. It might then seem natural to use these vectors as a basis for our Krylov subspace. However, for high powers of A these vectors will increasingly point in the direction of the dominant eigenvector, making this basis increasingly ill-conditioned. This will eventually lead to large numerical errors.

The Arnoldi process provides a way to compute a well-conditioned orthonormal basis of the Krylov subspace while retaining useful information about the original basis. Here we will use a modified Gram-Schmidt algorithm to implement this process. We define the first basis vector as $\vec{v}_1 \equiv \vec{r}_0/||\vec{r}_0||$ – the normalized initial residual. If we want to extend the basis of the ith Krylov subspace to the $(i+1)$th one, we orthogonalize $A\vec{v}_i$ against our previous basis to obtain $\vec{v}_{i+1}$ using Gram-Schmidt with two important alterations.

Recall that the Gram-Schmidt procedure works as follows: let $\{\vec{v}_i\} \equiv \{\vec{v}_0, \dots, \vec{v}_k\}$ be an orthogonal basis, and let $\vec{w} \notin \text{span}(\{\vec{v}_i\})$. Then we can form an orthogonal basis of $\text{span}\{\vec{v}_i\} \cup \vec{w}$ in an inductive manner by defining $\vec{v}_{i+1} \equiv \vec{w} - \sum_{k=0}^{i} \langle \vec{w}, \vec{v}_k \rangle \vec{v}_k$. In the modified Gram-Schmidt procedure we subtract the sum term-by-term, and use the intermediate results in the inner product instead of $\vec{w}$. This makes the algorithm less susceptible to numerical errors, but since the vectors $\vec{v}_i$ and $\vec{v}_j$ for $i \neq j$ are orthogonal it is mathematically equivalent.

In the Arnoldi process, we store the results of the inner products encountered in an upper Hessenberg matrix H. Let $\{\vec{v}_i\}$ be an orthonormal basis

of $\mathcal{K}_i(A, \vec{r}_0)$. We can find an orthonormal basis of $\mathcal{K}_{i+1}(A, \vec{r}_0)$ by orthogonalizing $\vec{w}_{i+1} \equiv A^i \vec{r}_0$ against $\{\vec{v}_i\}$ using modified Gram-Schmidt, by which we obtain a vector $\underline{\vec{w}}_{i+1}$, which we then normalize to obtain $\vec{v}_{i+1}$. We store the results of the inner products $h_{ji} = \langle \vec{w}_{i+1}, \vec{v}_j \rangle$ in the matrix $H^{(i+1,i)}$ of size $(i+1) \times i$, and store the norm of $\underline{\vec{w}}_{i+1}$ below the diagonal such that $h_{i+1,i} = ||\underline{\vec{w}}_{i+1}||$, the other elements of the matrix are defined to be zero. We then have the following relation

Lemma 1 (Arnoldi Relation)**.** *Let $H^{(i+1,i)}$ be defined as above. We define $V^{(k)}$ as the matrix defined by using the first k vectors $\vec{v}_i$ as columns. We then have:*

$$AV^{(k-1)} = V^{(k)} H^{(k+1,k)} \tag{A.3}$$

Proof. It suffices to show that the relation holds for individual matrix columns $\vec{v}_j$ for $j \leq k-1$. The relation then follows from a direct computation:

$$\vec{v}_j = \frac{\underline{\vec{w}}_j}{||\underline{\vec{w}}_j||} = \frac{A\vec{v}_{j-1} - \sum_{i=1}^{j-1} h_{i,j-1}\vec{v}_i}{h_{j,j-1}}$$

from which we see that

$$A\vec{v}_{j-1} = \vec{v}_j h_{j,j-1} + \sum_{i=0}^{j-1} h_{i,j-1}\vec{v}_i = \sum_{k=0}^{j} h_{k,j-1}\vec{v}_k$$

which leads to

$$A\vec{v}_j = \sum_{k=0}^{j+1} h_{k,j}\vec{v}_k$$

as required. □

A.2.2 *Least-squares approach*

Recall that we are interested in finding $\vec{x}_k \in \mathcal{K}_k(A, \vec{r}_0)$ such that $||\vec{b} - A\vec{x}_k||$ is minimal. Since $\vec{V}^{(k)}$ represents a basis for this space we can write $\vec{x}_k = \vec{V}^{(k)}\vec{y}$ for some $\vec{y} \in \mathbb{R}^k$. With the choice $\vec{x}_0 = 0$, we can rewrite the norm as follows:

$$\begin{aligned} ||\vec{b} - A\vec{x}_k|| &= ||\vec{b} - AV^{(k)}\vec{y}|| = ||\ ||\vec{r}_0||\vec{v}_1 - AV^{(k)}\vec{y}|| \\ &= ||\ ||\vec{r}_0||V^{(k+1)}\vec{e}_1 - V^{(k+1)}H^{(k+1,k)}\vec{y}|| = ||\ ||\vec{r}_0||\vec{e}_1 - H^{(k+1,k)}\vec{y}||. \end{aligned}$$

Here, we used that $V^{(k+1)}$ is orthogonal and thus norm preserving. We have thus transformed our problem into a least-squares problem: we want to find $\vec{y} \in \mathbb{R}^k$ such that:

$$\vec{y} = \text{argmin}_{\vec{z} \in \mathbb{R}^k} ||\ ||\vec{r}_0||\vec{e}_1 - H^{(k+1,k)}\vec{z}||, \tag{A.4}$$

which can be solved using QR decomposition. GMRES makes use of the fact that QR decompositions can be done efficiently for upper Hessenberg matrices using a sequence of Givens rotations.

A.2.3 *Givens rotations*

Definition 19. A Givens rotation is a transformation of the form

$$G = \begin{pmatrix} a & -b \\ b & a \end{pmatrix} \tag{A.5}$$

such that $a^2 + b^2 = 1$.

If we are interested in annihilating the second component of a vector $\begin{pmatrix} x \\ y \end{pmatrix}$, we use Givens rotations by requiring $G \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. For a vector $\begin{pmatrix} x \\ y \end{pmatrix}$ we see that if we define

$$a = \frac{x}{\sqrt{x^2 + y^2}}; \qquad b = \frac{-y}{\sqrt{x^2 + y^2}} \tag{A.6}$$

then indeed $G \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. In general, a vector in $\mathbb{R}^n$ can be transformed into a vector parallel to e_1 by a sequence of $n - 1$ Givens rotations.

A.2.4 *QR decomposition*

Since $H^{(k+1,k)}$ is an upper Hessenberg matrix, we only need to annihilate a single element per column, namely those on the subdiagonal, to perform a QR decomposition. We do this by using a series of Givens rotations. We denote the product of the necessary Givens rotations in the QR decomposition phase of step k, in which we find the vector $\vec{v}_{k+1}$, by $Q^{(k+1,k)}$, and the resulting upper triangular matrix by $R^{(k,k)}$. Thus we find the QR decomposition:

$$H^{(k+1,k)} = Q^{(k+1,k)} R^{(k,k)}.$$

It is easy to verify that we only have to add single columns to Q and R every step, and can therefore reuse the decompositions of the prior steps.

Using this QR decomposition we can rewrite the problem Equation A.4:

$$\begin{aligned} ||\vec{r}_0||\vec{e}_1 = H^{(k+1,k)}\vec{y} = Q^{(k+1,k)} R^{(k,k)}\vec{y}, \\ (Q^{(k+1,k)})^T ||\vec{r}_0||\vec{e}_1 = R^{(k,k)}\vec{y}, \\ (R^{(k,k)})^{-1}(Q^{(k+1,k)})^T ||\vec{r}_0||\vec{e}_1 = \vec{y}. \end{aligned}$$

Here, we can use that R, as an upper triangular matrix, can be easily inverted. Finally we find the required vector by $\vec{x}_k = V^{(k)}\vec{y}$.

A.2.5 *The GMRES algorithm*

In summary we conclude each ith iteration of the GMRES algorithm consists of 3 phases:

- Compute the vector $\vec{w}_{i+1} = A\vec{v}_i$. This requires one matrix-vector multiplication taking up to $\mathcal{O}(n^2)$ time if the matrix is dense.
- Orthogonalize the vector $\vec{w}_{i+1}$ against the basis $\{\vec{v}_j \mid j \leq i\}$ and normalize it, forming the matrix H in the process. This takes a total of i inner products which take $\mathcal{O}(n)$ time each.
- Finally we QR decompose the matrix H, and solve the least squares problem directly.

Because of the way the algorithm is constructed, we see that the final phase only takes constant time, since we can reuse the results of the previous iterations. We should also mention that certain optimizations can be applied. For example, we can restart the search after m iterations, using $\vec{x}_m$ as our initial guess. This keeps the Krylov subspaces of low-dimension, which limits the number of inner products necessary in the second phase. We can also apply a preconditioner to the system to further reduce the total running time.

A.3 CONJUGATE GRADIENT

For symmetric square matrices A an equally elegant method has been developed. In this section we will follow closely the discussion in Chapter 5 of [50]. We start with the Arnoldi relation Equation A.3, and apply $(V^{(k-1)})^T$ to both sides of the equation. We then find

$$(V^{(k-1)})^T A V^{(k-1)} = (V^{(k-1)})^T V^{(k)} H^{(i+1,i)} = H^{(i,i)}$$

where $H^{(i,i)}$ is the $i \times i$ upper block of the upper Hessenberg matrix $H^{(i+1,i)}$. Since A is symmetric the entire left-hand side is symmetric, and thus $H^{(i,i)}$ is symmetric. This implies in particular that the matrix $H^{(i+1,i)}$ is tridiagonal, which we will use to derive a method called the **C**onjugate **G**radient (CG) method. In this discussion we will denote this matrix with $T^{(i+1,i)}$ to remind us of the fact that it is tridiagonal.

The Lanczos method

The general method we will derive here is called the Lanczos method, of which CG is a clever variation. It is of the Ritz-Galerkin category, which means that each residual $\vec{r}_k$ is orthogonal to the previous Krylov subspace $\mathcal{K}_k(A, \vec{r}_0)$. In this approach we can extend our basis by letting the vector $\vec{v}_k$ coincide with the residual $\vec{r}_k$ automatically leading to an orthogonal basis

for the next Krylov subspace. Recall also that we have a relation between $\vec{r}_k$ and $\vec{r}_0$:

$$\vec{r}_k = (I - Aq(A))\vec{r}_0,$$

where $q(A)$ is some polynomial with constant term equal to the identity matrix. This follows directly from $\vec{r}_k = \vec{b} - A\vec{x}_k$ and $\vec{x}_k \in \mathcal{K}_k(A, \vec{r}_0)$. Using the Arnoldi relation, Equation A.3, we see that for vectors $\vec{v}_k \equiv \vec{r}_k$ we have:

$$A\vec{v}_k = t_{k+1,k}\vec{v}_{k+1} + t_{kk}\vec{v}_k + t_{k-1,k}\vec{v}_{k-1},$$

where t_{ij} denotes the coefficients of the tridiagonal matrix $T^{(i,i)}$. Combining these two equations we find a recurrence relation

$$t_{i+1,i} + t_{ii} + t_{i-1,i} = 0.$$

Since we let our basis vectors coincide with our residuals we have that $V^{(k)} = R^{(k)}$, where $R^{(k)}$ is the matrix defined by using the first k residuals as columns. We conclude that for CG, the Arnoldi relation reduces to

$$AR^{(k)} = R^{(k+1)}T^{(k+1,k)}. \tag{A.7}$$

Because $R^{(k)}$ forms a basis for the kth Krylov subspace, we can use it to write our vector $\vec{x}_k$ in terms of a vector inside this subspace. That is, there exists a vector $y \in \mathcal{R}^k$ such that:

$$\vec{x}_k = R^{(k)}\vec{y}.$$

Because $\vec{r}_k \perp \mathcal{K}_k(A, \vec{r}_0)$, we have:

$$0 = (R^{(k)})^T(A\vec{x}_k - \vec{b}) = (R^{(k)})^T AR^{(k)}\vec{y} - (R^{(k)})^T\vec{b}.$$

We use Equation A.7 to write:

$$\begin{aligned} & (R^{(k)})^T AR^{(k)}\vec{y} = (R^{(k)})^T\vec{b} \\ \implies \quad & (R^{(k)})^T R^{(k+1)}T^{(k+1,k)}\vec{y} = (R^{(k)})^T\vec{b} \\ \implies \quad & (R^{(k)})^T R^{(k)}T^{(k,k)}\vec{y} = ||\vec{r}_0||\vec{e}_1 \end{aligned}$$

because the residuals are mutually orthogonal. Scaling inversely with $||\vec{r}_0||$, we see that we have reduced our original problem to solving $T^{(k,k)}\vec{y} = \vec{e}_1$, from which $\vec{x}_k$ will follow.

Saving storage

The CG method is a way to achieve the above without storing the matrix $R^{(k)}$ explicitly. We assume that the matrix A is, in addition to being symmetric, also positive definite, such that in particular it has strictly positive elements

on the diagonal. Multiplying both sides of Equation A.7 with $(R^{(k)})^T$ on the left, we find

$$(R^{(k)})^T A R^{(k)} = (R^{(k)})^T R^{(k)} T^{(k,k)}.$$

Note that the left-hand side is positive definite, and that $(R^{(k)})^T R^{(k)}$ is a diagonal matrix. This implies in particular that the diagonal of $T^{(k,k)}$ is non-zero, such that we can perform an LU decomposition without pivoting. We will write

$$T^{(k,k)} = L^{(k)} U^{(k)}.$$

Starting from $\vec{x}_k = R^{(k)}\vec{y} = R^{(k)}(T^{(k,k)})^{-1}\vec{e}_1$, we see that we can write $\vec{x}_k$ as:

$$\vec{x}_k = R^{(k)}(U^{(k)})^{-1}(L^{(k)})^{-1}\vec{e}_1.$$

We will define $P_k \equiv R^{(k)}(U^{(k)})^{-1}$ and $q_k \equiv (L^{(k)})^{-1}\vec{e}_1$, such that we have $\vec{x}_k = P_k q_k$. Note that $L^{(k)}$ only has elements on the subdiagonal and the diagonal, and $U^{(k)}$ has unit diagonal, and elements on the superdiagonal. From $L^{(k)} q = \vec{e}_1$ we see that:

$$(q_k)_0 = \frac{1}{L^{(k)}_{00}}, \qquad L^{(k)}_{i+1,i}(q_k)_i + L^{(k)}_{i+1,i+1}(q_k)_{i+1} = 0.$$

In particular, each iteration we only have to extend the vector q with one component, which can be computed recursively. We can find a similar recursive relation for the columns of $P^{(k)}$, which we denote by $\vec{p}_0, \ldots, \vec{p}_{k-1}$. In particular we find

$$\vec{p}_k = \vec{r}_k - U^{(k)}_{k-1,k}\vec{p}_{k-1}.$$

Using $\vec{x}_k = P_k q_k$ we see that we can compute the next vector recursively:

$$\vec{x}_k = \vec{x}_{k-1} + (q_k)_{k-1}\vec{p}_{k-1}.$$

The CG algorithm

The name for the CG method can be explained by observing that the vectors $\vec{p}_k$ are conjugate, i.e. orthogonal with respect to the inner product defined by the symmetric matrix A. Furthermore they are the gradients of the function $f(z) = ||\vec{x} - \vec{z}||_A$, where $||\cdot||_A$ is the norm defined by the inner product induced by A.

We can simplify the notation greatly by observing that we can reuse most of the information from the previous iteration. We will drop the index from L, U, P and q, since in each iteration we only need to add an additional column or element respectively. We will write $\alpha_i \equiv \vec{q}_i$, $\beta_i \equiv U_{i-1,i}$. In summary, we have the following recursion relations:

$$\begin{aligned}
\vec{p}_k &= \vec{r}_k + \beta_{k-1}\vec{p}_{k-1} \\
\vec{x}_k &= \vec{x}_{k-1} + \alpha_{k-1}\vec{p}_{k-1} \\
\vec{r}_k &= \vec{r}_{k-1} - \alpha_{k-1}\vec{p}_{k-1}.
\end{aligned}$$

With a straightforward calculation we can recast these equations to obtain neat expressions for α_k and β_k:

$$\alpha_k = \frac{||\vec{r}_k||^2}{\vec{p}_k^T A \vec{p}_k}$$
$$\beta_k = \frac{||\vec{r}_k||}{||\vec{r}_{k-1}||}.$$

We see that for the kth iteration in the CG method we have the following phases:

- Compute $||r_{k-1}||$, taking $\mathcal{O}(n)$ time.
- We compute β_{k-1} and $\vec{p}_k$, taking one vector update of $\mathcal{O}(n)$ time. Next we compute the product $A\vec{p}_k$ taking $\mathcal{O}(n^2)$ time if the matrix is dense.
- Compute α_k, $\vec{x}_k$ and $\vec{r}_k$. This requires one inner product and two vector updates.

B

ZEE; A DISTRIBUTED MATRIX LIBRARY AND PARTITIONING FRAMEWORK

In this chapter we introduce a new partitioning framework called **Zee**. This framework is written in modern C++, and can be viewed as a unified software library for applications (initially linear algebra and in particular linear solvers) that also automatically and autonomously optimizes the running time of the operations involved, by means of balancing and partitioning.

B.1 INTRODUCTION

There are many software packages available that focus on hypergraph partitioning problems, and these packages can all be used to partition sparse matrices to optimize the running time of the SpMV operations. Examples of such packages can be found in Table 3.2.

Because there are already so many libraries available, the question arises "Why write another software library for partitioning?". For us, the answer is twofold. First and foremost, we are ultimately interested in mixing (linear) solver iterations and partitioner iterations, and this fact alone forces us to rethink the way we develop software for both of these problems. The existing solution, where the partitioner exists as a completely separate entity, simply does not suffice. Second, we felt there was room for a partitioning framework that was built more generally than the existing options. In particular, Zee was written with a number of design goals in mind:

- *Fully distributed*. Many partitioners are sequential, and are therefore not only unable to handle matrices that must be distributed from the start because of their size, they do not scale well either. A major goal of the framework is to be able to eventually do all computations in

a completely distributed way, without needing centralized or shared memory.

- *Modular and extensible*. The library is built generally enough to support new partitioning methods as they are invented, without requiring to make major changes to the software. Furthermore, it is easy to extend its capabilities by adding additional partitioning methods or functionality, also for third-party users.
- *Maintainable and future-proof*. C++ has been a very successful language for performance-critical applications. The latest revisions of C++, C++11 and C++14 add many features to the language that make it particularly well-suited for the development of this framework.
- *Cross-platform*. It is designed to work on embedded systems, personal computers, and we plan to support distributed computing using e.g. MPI in a future release. In particular the library has been successfully tested on multi-core computers as well as the Parallella.
- *Integration* in applications. In this thesis we have focused on developing methods for iterative refinement of partitionings. The Zee framework is meant to be integrated in software employing these techniques. While first we target sparse matrix applications, in the future we could support more general applications using the same techniques.

The syntax and overall structure of Zee is based on Eigen [25], which is a 'C++ template library for linear algebra', such that it should feel familiar to use our new framework for users of this popular linear algebra library. Software development for scientific computing requires careful design and practices (see e.g. [3] for an overview), and we have taken care to ensure that every component of the software is of high quality. Zee is available as free software under the lesser GNU public license (LGPL) at `http://git.codu.in/jwbuurlage/Zee`.

B.2 FEATURES

In this section we will talk about some of the features of the library. We will also mention some high-level details that users ought to know if they want to use the library effectively. Specific implementation details and a number of usage examples will be given in later sections.

B.2.1 *Linear algebra; types and operations*

Zee supports both dense and sparse matrix types, and supports many operations on these matrices. All of these types are distributed on the lowest level.

This means in particular that a matrix A consist of a number of *images*, which all form mutually disjoint submatrices of A. Although, besides cache-effects, this should have little implications on the behaviour of the library on shared memory architectures, it does mean the library is easily extended to systems with distributed memory. Storage of the matrices is logically separated from other components of the matrix type. Compatible types (including scalars and vectors) can be added, subtracted and/or multiplied with each other.

In the initial release of the library only dense vectors are supported. Just like matrices, vectors are necessarily distributed over multiple *images*. Many operations such as taking the dot product of two vectors, and computing the norm of a vector are supported.

Given a matrix A and a vector $\vec{b}$, solvers can be used to find a vector $\vec{x}$ such that $A\vec{x} \approx \vec{b}$. The solvers in Zee are completely general; they can support any matrix type to ensure cross-platform portability.

B.2.2 *Partitioning*

Partitioners are capable of (re)distributing a matrix into a number of images. The specific partitioners that come with the library (stock partitioners) optimize the distribution for the SpMV operation. New partitioners can be added by implementing only a few functions, and can be used instead of the stock partitioners for all operations that are supported by Zee.

We also support partitioning a (dense) vector. This is always done with respect to a matrix partitioning, and can be seen as a post processing step. Conversely, the *indices* of a matrix can be localized with respect to a vector partitioning. Operations such as SpMV are performed with local indices, such that Zee can be used on distributed systems.

B.2.3 *Utilities*

There is also an accompanying utility library with, among others, the following features:

- Loading dense and sparse matrices from the matrix market format [8].
- Elaborate benchmarks can be created using high-precision timers.
- There is also support for making plots using intermediate formats. The plots are made using `matplotlib` library [29] for the Python programming language. In particular we support *spy plots* of matrices, and *line plots* for e.g. inspecting convergence behaviour.
- Convenience methods for outputting tabular data can be used to make *reports* of the performance of applications that use Zee. We support

multiple output formats such as comma-separated values and LaTeX tables.

- The *logger* can be used to track information of running programs, and to see errors and warnings. All major types that come with the library can be output using the logger.

B.3 OVERVIEW OF INTERNAL STRUCTURE

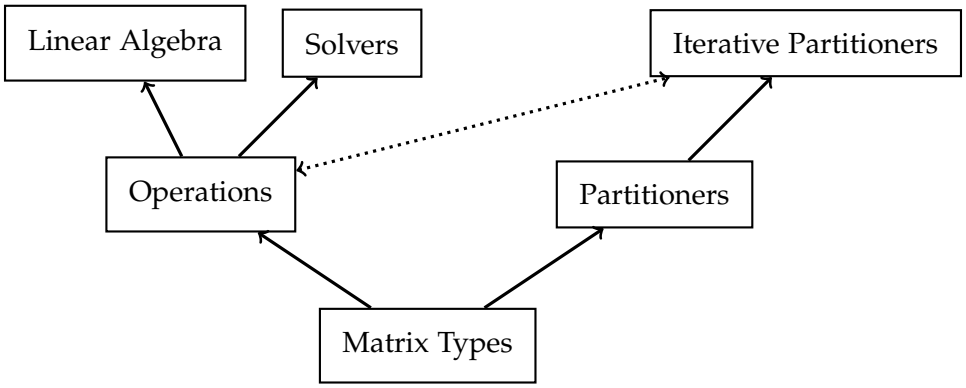


Figure B.1.: Here we give a schematic overview of the different modules of which Zee consists. Relations between modules are indicated with an arrow. One of the unique features of the library is that *iterative partitioners* and *operations* work together to minimize their cumulative run time, this is indicated with a dotted line.

An overview of the modules that are present in the pre-alpha release Zee are depicted in Figure B.1. The library is written in an object oriented (OO) fashion. The most fundamental types are (distributed) matrix objects. These matrix objects are used or manipulated by the other components of the library, such as partitioners and solvers.

Here we will give a detailed overview of the implementation of every component, which is relevant for users who want to use the library to its full capability, as well as for developers who want to extend the library. We will denote `Objects` and other code with a monospace font on a gray background. Our discussion will be such that readers that are familiar with OO programming, but unfamiliar with C++ syntax should still be able to grasp the necessary concepts.

MATRIX TYPES

Every matrix class in Zee derives from `Zee::DMatrixBase`, which is a base class for distributed matrices. This base class has a number of template

arguments. Arguably the most important arguments decide the *types* that are used for the indices, which should always be unsigned integers, and the values – which can in principle be any type that supports addition and multiplication operations. These are denoted throughout the library with `TIdx` and `TVal` respectively. The third and final template argument is `Derived`. This is done so that the parent (here the base matrix class) knows the type of its children, which is useful for optimizing certain operations (see also the section on linear algebra operations). This construction is known as the *curiously recurring template* (CRT) pattern. This base class also keeps track of common traits of matrix types such as the size, and the number of processors over which it is distributed.

Dense matrices all derive from `Zee::DDenseMatrixBase`. For multi-core processors we provide a type `Zee::DMatrix`, which is the default dense matrix type on these platforms. This type supports element access via the `at(i, j)` member function. It also supports other operations such as transposing.

Much thought has been put into the specific implementation of *sparse matrices*. The base class for sparse matrices is `Zee::DSparseMatrixBase`. In addition to the template arguments of the main base class discussed above, we also have freedom in choosing the *image type*.

The sparse matrix object can be used directly or via operations. Since this object is distributed, when we want to compute e.g. the communication volume we need to combine the information that is stored over multiple processors. This means that the elements of the matrix are not stored by the matrix object itself, but instead it holds references to its images (by default of the `Zee::DSparseMatrixImage` type) on other processors, which in turn hold references to an object that actually stores the matrices.

Sparse matrices can be stored in a great number of ways. In fact, in some applications such as computerized tomography they need not be stored explicitly at all. The Zee library recognizes this fact, and makes very little assumptions on the specific underlying storage mechanisms – which can even be supplied by the user. The storage objects need only support *iterating* over the triplets that make up the image (by implementing a type that conforms to the methods defined in `Zee::StorageIteratorTriplets`). These triplets can be computed on the fly, or can be stored explicitly. This makes it simple to implement general algorithms that work just as well with compressed or implicit storage mechanisms, as well as in applications where the triplets are stored explicitly.

Vector objects derive from `Zee::DVectorBase`, and are viewed as dense matrices of size $n \times 1$. Specific information about a vector such as its norm or size can be requested using the appropriate functions.

OPERATIONS ON VECTORS AND MATRICES

Compatible objects can be multiplied, added, subtracted, scaled etc. We support basic linear algebra operations such as the matrix-vector product, the matrix-matrix product, the dot product etc. A future goal is to support all BLAS operations, and possible even leverage LAPACK [2] routines.

To prevent the unnecessary creation of temporary objects every expression involving linear algebra types is first combined into a recursive type corresponding to a tree of the expression (similar to an abstract syntax tree). This construction is known as *expression templates*, and can be used to optimize operations *at compile time*. It also makes it much easier to implement new operations that can be used together with existing operations.

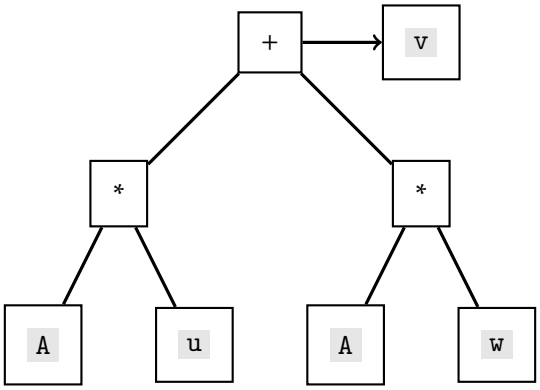


Figure B.2.: Here we consider the expression `v = A * u + A * w`. First a type tree of multiple objects of type `Zee::BinaryOperation` is constructed. Then we put `v` equal to this type, in effect calling the equal-sign operator. This operator recursively calls the appropriate functions which perform the operations in an efficient manner, for example the results can be written directly to `v`.

Consider for example the expression `v = A * u + A * w`, where `u` and `w` are of `Zee::DVector` type, and `A` is a dense matrix of type `Zee::DMatrix`. Every operation such as `A * u` returns an object that encodes the operation. In this case it will be a binary operation, with a dense matrix in the left hand side, and a vector in the right hand side. In particular it will be of type `Zee::BinaryOperation<Zee::DMatrix, Zee::DVector>`. These operation types can be combined. Indeed, in our main example the operation `+` has another binary operation on the left- and right hand side. These operations are not performed until the entire expression formed is assigned to an object, in this case the vector `v`. In this way we can directly use the storage allocated for this resulting vector without introducing any temporary objects for the intermediate results. We also have very fine control over the way the operations are performed. See also Figure B.2.

Operations are performed by functions with the signature `perform_operation<TOperation>`. New operations can be added by implementing this function for appropriate operation types. We can also use this function to manipulate the tree while the program is compiling. For example, we may detect that:

```
v = A u + A w = A(u + w)
```

and note that the right-most expression is more efficient to compute than the original expression. By rearranging the tree, and recursively calling the function that performs the computation on the new type we can compute `v` more efficiently.

Partitioning matrices

Partitioners are objects that modify the distribution of a matrix object. In particular it can create, modify or destroy images corresponding to a matrix. The base partitioner type is `Zee::Partitioner`. Every partitioner must implement a method for initialization, and a method for partitioning. A number of partitioners are supplied with Zee. Examples are basic partitioners corresponding to the cyclic, block, or random distributions, and more advanced partitioners such as multi-level partitioners based on the KLFM heuristic, and flavours of these partitioners such as the medium grain method.

There is a special type of matrix partitioner called an *iterative partitioner* which are partitioners that derive from `Zee::IterativePartitioner`. In addition to a partition method, these partitioners must also implement a *refine* method. This method should be able to (iteratively) refine the partitioning of a matrix.

Vector partitioner

After a matrix has been partitioned it is ready to perform parallel operations such as the SpMV, $\vec{u} = A\vec{v}$. However, to realize the communication volume that a partitioner obtains we need to have an appropriate distribution for the vectors that are involved in this operation. In principle this can be done greedily, by assigning a vector component of $\vec{u}$ and $\vec{v}$ to any processor in the corresponding row or column respectively. For iterative solvers we may require that the distribution of a vector $\vec{v}$ and the resulting vector $\vec{u}$ are identical, since the resulting vector will be multiplied with A in the next iteration. This is an example where we want a more involved vector partitioner.

Vector partitioners derive from `Zee::VectorPartitioner`. Zee comes with a vector partitioner `Zee::GreedyVectorPartitioner` that finds a good

distribution for $\vec{v}$ and $\vec{u}$ when the application requires that they should be identical.

Iterative solvers

The final module we will discuss in detail is the iterative solver. In particular we will discuss how they combine with iterative partitioners to minimize the runtime of an application. Solvers implement a single function, e.g. `Zee::GMRES::solve`. This method takes as arguments at least a matrix A, a vector $\vec{b}$, an initial guess $\vec{x}_0$ and various parameters specific to the solver, such as the number of iterations and the tolerance level.

Solvers that combine with partitioners take in addition an iterative partitioner as its argument. By expecting the convergence behaviour it estimates the number of iterations left, and it combines this information with the communication volume and the iterative partitioner to decide whether to *refine* the matrix A using the supplied iterative partitioner. The net result is that the total runtime of the solver is optimized.

B.4 EXAMPLES

To give an idea of what the library looks like in practice, and how to write programs using Zee, we give a number of basic example C++ programs here. We begin with initializing matrix and vector objects.

Listing B.1: Initializing vectors and matrices

```
1  // matrices are initialized with a single image containing
2  // all zeros by default
3  auto A = DSparseMatrix<unsigned int, float>(rows, cols);
4
5  // we can construct sparse matrices by supplying triplets
6  std::vector<Triplet<unsigned int, float>> triplets;
7  A.setFromTriplets(triplets.begin(), triplets.end());
8
9  // they can also be loaded from a matrix market file
10 // by default the matrices are distributed cyclically over the
11 // ‘procs‘ processors
12 auto A = DSparseMatrix<unsigned int, float>("matrix.mtx", procs);
13
14 // dense matrices can be loaded in similarly
15 auto B = DMatrix<unsigned int, float>(rows, cols);
16 B.at(0, 0) = 1.0f;
17 auto B = DMatrix<unsigned int, float>("dense_matrix.mtx", procs);
18
```

```
// vectors are zero initialized by default, and can be
// viewed as (n x 1) dense matrices
auto v = DVector<unsigned int, float>(size);

// they can also be filled with a supplied value
auto value = 1.0f;
auto v = DVector<unsigned int, float>(size, value);
```

Now that we have seen a number of ways to initialize matrices and vectors, we are ready to show how we can perform operations with these objects.

Listing B.2: Basic linear algebra

```

// we initialize some objects
DVector<> v, w;
DSparseMatrix<> A;
DMatrix<> B;
DMatrix<> C;

// we can add and subtract vectors
DVector<> u1 = v + w;
DVector<> u2 = v - w;

// we can multiply matrices with vectors
DVector<> u1 = A * v;

// we can multiply matrices with matrices
DMatrix<> D = B * C;

// we can combine any number of operations as we see fit
DVector<> u1 = 2.0f * A * (v + w) + v;

// or even reuse existing vectors
w = A * v;
```

Next we show how to partition matrices and vectors explicitly. Note that there is a difference in how ordinary partitioners, and how iterative partitioners are used.

Listing B.3: Partitioning matrices and vectors

```
// we initialize some objects
DVector<> v, w;
DSparseMatrix<> A, B;

```

```
5  // we can (bi-)partition the matrix A using e.g. medium-grain
6  MGPartitioner<decltype(A)> mediumGrain(epsilon);
7  mediumGrain.partition(A);
8
9  // we can refine the partitioner with MG-IR
10 while (!mediumGrain.locallyOptimal()) {
11     mediumGrain.refine(A);
12 }
13
14 // we can also use Hyper-PuLP to partition
15 PulpPartitioner<decltype(B)> pulp(B, procs, epsilon);
16 pulp.initialize(HGModel::row_net);
17
18 // we obtain an initial partitioning
19 pulp.initialPartitioning(iterations);
20
21 // and now we can perform refinement iterations using
22 pulp.refine();
23
24 // if we want to perform a SpMV we need to partition the vector
25 GreedyVectorPartitioner<decltype(A), decltype(b)> greedy(A, v, w);
26 greedy.partition();
27
28 // we localize the indices of the matrix for efficiency
29 greedy.localizeMatrix();
30
31 // now we can perform the SpMV
32 v = A * w;
```

The final example we will discuss here is how to use solvers. The solver that is supplied with Zee is (modified) GMRES. We plan to add more solvers after the initial release.

Listing B.4: Solving linear systems

```
1  auto A = DSparseMatrix<TVal, TIdx>("matrix.mtx", procs);
2  auto b = DVector<TVal, TIdx>(A.getRows(), 1.0);
3
4  // initial x is the zero vector
5  auto x = DVector<TVal, TIdx>(A.getCols());
6
7  // Start GMRES
8  GMRES::solve<TVal, TIdx>(A, // Matrix
9          b,                  // RHS vector
10         x,                  // initial guess for x
```

```
11          outer,              // outer iterations
12          inner,              // inner iterations
13          epsilon,            // tolerance level
14          true);              // plot residuals
```

We have only shown the basic usage of Zee here. The user has a lot of control over the inner workings of the algorithm, but the library has been built with sensible defaults such that it can also be used right out-of-the-box.

B.5 EXTENDING ZEE

In this section we describe how to extend Zee, either by implementing new features or by supplying platform specific types.

We note that direct contributions to the library are also welcomed. The source code of the initial release is hosted on GitHub[1] and further details on how to contribute to the library can be found there. More information on how to use and extend the library for specific applications can also be found on GitHub.

Extending Zee is typically done by implementing subclasses of the appropriate built-in types. As an example of this, we mention that we have successfully ported the library to the Parallella platform by providing special implementations of components of the sparse matrix, dense matrix and vector classes. The only other thing that is necessary is to write platform specific code for the implementation of basic linear algebra functions. For this specific platform, we have implemented the BSPS algorithms described in Chapter 5.

The implementation of Zee on the Parallella is released as a separate micro-library which we have called Zephany[2] and depends on Epiphany BSP and Zee itself.

Custom types and partitioners can be created by providing subclasses of the appropriate classes. These user-defined classes can be used as input for the partitioners and solvers that come with Zee, and can leverage features of Zee such as reusing the implementation of built-in classes, custom storage for sparse matrices, and the utility library.

1 http://www.github.com/jwbuurlage/Zee
2 http://www.github.com/jwbuurlage/Zephany

BIBLIOGRAPHY

[1] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137 – 147, 1999.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Katy Huff, Ian Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, Greg Wilson, and Paul Wilson. Best practices for scientific computing. *CoRR*, abs/1210.0530, 2012.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *IN PODS*, pages 1–16, 2002.

[5] Tom Bannink, Jan-Willem Buurlage, and Abe Wits. Epiphany BSP 1.0. `http://www.codu.in/ebsp/docs/`, 2015.

[6] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[7] Rob H. Bisseling, Bas O. Fagginger Auer, A. N. Yzelman, Tristan van Leeuwen, and Ümit V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*, Computational Science Series, pages 321–349. CRC Press, Taylor & Francis Group, Boca Raton, FL, 2012.

[8] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, London, UK, UK, 1997. Chapman & Hall, Ltd.

[9] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[10] T. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452. SIAM, Philadelphia, PA, 1993.

[11] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.*, 42(3):153–159, May 1992.

[12] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proceedings Asia and South Pacific Design Automation Conference*, pages 661–666. ACM Press, New York, 2000.

[13] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[14] Ümit V. Çatalyürek and Cevdet Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplications. In A. Ferreira, J. Rolim, Y. Saad, and T. Yang, editors, *Proceedings Third International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 1996)*, volume 1117 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, Berlin, 1996.

[15] Ümit V. Çatalyürek and Cevdet Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical report, Department of Computer Engineering, Bilkent University, Ankara, Turkey, 1999.

[16] Ümit V. Çatalyürek and Cevdet Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, page 118. IEEE Press, Los Alamitos, CA, 2001.

[17] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007. Best Algorithms Paper Award.

[18] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, February 2010.

[19] Timothy A. Davis. University of Florida sparse matrix collection. Online collection, http://www.cise.ufl.edu/research/sparse/matrices, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1994-2004.

[20] K. D. Devine, E. G. Boman, R.T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings IEEE International Parallel and Distributed Processing Symposium 2006*, page 102. IEEE Press, Los Alamitos, CA, 2006.

[21] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[22] A. E. Fincham and B. Ford, editors. *Parallel Computation (The Institute of Mathematics and its Applications Conference Series, New Series)*. Oxford University Press, 1994.

[23] W. Fortes and K.J. Batenburg. Quality bounds for binary tomography with arbitrary projection matrices. *Discrete Applied Mathematics*, 183:42–58, mar 2015.

[24] Gene Golub and Charles van Loan. *Matrix computations*. The Johns Hopkins University Press, Baltimore, 2013.

[25] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[26] David Heath, Simon Kasif, S. Rao Kosaraju, Steven Salzberg, and Gregory Sullivan. Learning nested concept classes with limited storage. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'91, pages 777–782, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[27] Vitali Henne, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, and Christian Schulz. n-level hypergraph partitioning. *CoRR*, abs/1505.00693, 2015.

[28] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[29] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[30] Adapteva Inc. Epiphany architecture reference: `http://www.adapteva.com/docs/epiphany_arch_ref.pdf`, 2012.

[31] Adapteva Inc. Parallella project website: `http://www.parallella.org`, 2012.

[32] A. C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[33] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings 36th ACM/IEEE Conference on Design Automation*, pages 343–348. ACM Press, New York, 1999.

[34] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, feb 1970.

[35] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 346–357. VLDB Endowment, 2002.

[36] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24(3-4):287–297, 1999.

[37] M. E. J. Newman. Detecting community structure in networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):321–330, 2004.

[38] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web, 1999.

[39] Daniël M. Pelt and Rob H. Bisseling. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *Proceedings IEEE International Parallel and Distributed Processing Symposium 2014*, pages 529–539. IEEE Press, 2014.

[40] Daniël M. Pelt and Rob H. Bisseling. An exact algorithm for sparse matrix bipartitioning. *Journal of Parallel and Distributed Computing*, 85:79 – 90, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[41] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[42] James A. Ross, David A. Richie, Song J. Park, and Dale R. Shires. Parallel programming model for the Epiphany many-core coprocessor using threaded MPI. In *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, MES '15, pages 41–47, New York, NY, USA, 2015. ACM.

[43] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[44] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *2014 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical & Electronics Engineers (IEEE), oct 2014.

[45] G.M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 550–559, May 2014.

[46] Alexandre Tiskin. The bulk-synchronous parallel random access machine. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing. Vol. II*, volume 1124 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, Berlin, 1996.

[47] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[48] Leslie G. Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011. Celebrating Karp's Kyoto Prize.

[49] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[50] H. A. van der Vorst. *Iterative Krylov methods for large linear systems*. Cambridge University Press, Cambridge, 2009.

[51] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.