VieM v1.00 – Vienna Mapping and Sparse Quadratic Assignment User Guide

Marcelo Fonseca Faraj, Konrad von Kirchbach, Markus Lehr, Christian Schulz and Jesper Larsson Träff

University of Vienna, *Vienna, Austria*
Email: {marcelo.fonseca-faraj,christian.schulz}@univie.ac.at

Technical University of Vienna,
Vienna, Austria
Email: {konrad.kirchbach,markus.lehr,traff}@tuwien.ac.at

Abstract

This paper severs as a user guide to the mapping framework VieM (Vienna Mapping and Sparse Quadratic Assignment). We give a rough overview of the techniques used within the framework and describe the user interface as well as the file formats used.

Contents

1 Introduction

Communication performance between processes in high-performance systems depends on many factors. For example, communication is typically faster if communicating processes are located on the same processor node compared to the cases where processes reside on different nodes. This becomes even more pronounced for large supercomputer systems where processors are hierarchically organized into, e.g. islands, racks, nodes, processors, cores with corresponding communication links of similar quality. Given the communication pattern between processes and a hardware topology description that reflects the quality of the communication links, one hence seeks to find a good mapping of processes onto processors such that pairs of processes exchanging large amounts of information are located closely.

Such a mapping can be computed by solving a corresponding quadratic assignment problem (QAP) which is a hard optimization problem. Sahni and Gonzalez [10] have shown QAP to be strongly NP-hard and, unless P=NP, admitting no constant factor approximation algorithm. In addition, there are no algorithms that can solve meaningful instances with $n > 20$ to optimality in a reasonable amount of time [3]. Hence, heuristic algorithms are necessary in order to solve large scale instances. Multiple heuristics have been proposed to tackle real world instances [2, 6, 9].

We make two important assumptions that are typically valid for modern supercomputers and the applications that run on those. First, communication patterns are almost always sparse since not all processes have to communicate with each other. This is especially true for large scale scientific simulations in which the underlying models of computation and communication are already sparse, see, e.g. [4, 5, 14]. To efficiently parallelize the simulation one normally employs graph partitioning techniques which then in turn yield a sparse communication pattern between the processes. Second, we assume that the hardware communication topology under consideration is hierarchical with communication links on the same level in the hierarchy featuring the same communication speed. This is typically observed in current high-performance systems, e.g. SuperMUC[1]. Using these assumptions, we derive algorithms that are able to create high quality mappings, as well as faster local search algorithms for improving assignments.

Problem Definition. The total communication requirement between the set of processes in an application can be modeled by a weighted communication graph. The underlying hardware topology can likewise be modeled by a weighted graph. Our abstract problem is to embed the communication graph onto the topology graph under optimization criteria that we explain below. We assume that the number of nodes in host and topology graphs are the same. Unless otherwise mentioned, a processing element (PE) typically represents a core of a machine.

Throughout the user guide, $\mathcal{C} \in \mathbb{R}^{n \times n}$ denotes the communication matrix and $\mathcal{D} \in \mathbb{R}^{n \times n}$ the topology matrix or distance matrix. More precisely, $\mathcal{C}_{i,j}$ describes the amount of communication that has to be done between process i and j and $\mathcal{D}_{i,j}$ represents the weighted distance between PE i and PE j. That is, the cost for communicating the amount $\mathcal{C}_{i,j}$ between processors i and j is $\mathcal{C}_{i,j}\mathcal{D}_{i,j}$. We follow Brandfass et al. [2] and others, and model the embedding problem as a quadratic assignment problem (QAP): Find a one-to-one mapping Π of processes to PEs which minimizes the overall communication cost. More precisely, we want to minimize $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{i,j} \mathcal{C}_{\Pi(i),\Pi(j)}\mathcal{D}_{i,j}$ where the sum is over all PE pairs and $k = \Pi(i)$ means that process k is assigned to PE i. Our framework assumes that $\mathcal{C}$ and $\mathcal{D}$ are symmetric – otherwise one can create equivalent QAP problems with symmetric inputs [2].

Graph partitioning is a key component in our algorithms to find initial solutions. The *graph partitioning problem* looks for *blocks* of nodes V_1,...,V_k that partition V, i.e. $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil c(V)/k \rceil$ for some parameter ϵ. In the *perfectly balanced case* the imbalance parameter ϵ is set to zero, i.e. no deviation from the average is allowed. One commonly used objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. A vertex $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a boundary vertex.

[1] Leibniz Supercomputing Centre, Gauss Centre for Supercomputing e.V.

2 Mapping Techniques within VieM

We now give a rough overview over the algorithms implemented in our framework. For details on the algorithms, we refer the interested reader to the corresponding paper [15].

2.1 Local Search

Heider [6] proposes a method to improve an already given permutation/mapping. The method repeatedly tries to perform swaps in the assignment. To do so, the author defines a pair-exchange neighborhood $N(\Pi)$ that contains all permutations that can be reached by swapping two elements in Π. Here, swapping two elements means that $\Pi(i)$ will be assigned to processor j and $\Pi(j)$ will be assigned to processor i after the swap is done. The algorithm then looks at the neighborhood in a cyclic manner. More precisely, in each step the current pair (i, j) is updated to $(i, j + 1)$ if $j < n$, to $(i + 1, i + 2)$ if $j = n$ and $i < n - 1$, and lastly to $(1, 2)$ if $j = n$ and $i = n - 1$. A swap is performed if it yields positive gain, i.e. the swap reduces the objective. The overall runtime of the algorithm is $O(n^3)$. We denote the search space with N^2. To reduce the runtime, Brandfass et al. [2] introduce a couple of modifications. Initially computing as well as recomputing the objective function after a swap is performed is an expensive step in the algorithm. In their work, both the communication pattern as well as the distances between the PEs are given as complete matrices. These matrices have a quadratic number of elements and hence the initial computation of the objective function costs $O(n^2)$ time. After a swap is performed, Brandfass et al. update the objective using the objective function value before the swap. Overall, an update step in their algorithm takes $O(n)$ time which is clearly a bottleneck for sparse communication patterns. We described methods how we speed up the initial computation as well as the update of the objective. This yields much faster local search algorithms.

In addition to that we defined swapping neighborhoods using the communication graph $G_{\mathcal{C}}$. In the simplest version, assignments are only allowed to be swapped if the processes are connected by an edge in the communication graph, i.e. the processes have to communicate with each other. We denote this neighborhood with $N_{\mathcal{C}}$. The size of the search space is $O(m)$ since it contains exactly m pairs that may be swapped. Swaps are performed in random order. Local search terminates after m unsuccessful swaps, i.e. all pairs have been tried and no swap resulted in a gain in the objective. Note that this approach assumes that swaps with positive gain are close in terms of graph theoretic distance in the communication graph. We also define augmented neighborhoods in which swaps are allowed if two processes have distance less than d in the communication graph. We denote this neighborhood by $N^d_{\mathcal{C}}$. Note that this creates a sequence of neighborhoods increasing in size $N_{\mathcal{C}} \subseteq N^2_{\mathcal{C}} \subseteq \ldots \subseteq N^n_{\mathcal{C}} = N^2$ where N^2 is the largest neighborhood used by Brandfass et al. [2].

2.2 Hierarchical Initial Solutions

Our framework also contains algorithms to initially create solutions. Throughout this section, we assume that the input communication matrix is already given as a graph $G_{\mathcal{C}}$, i.e. no conversion of the matrix into a graph is necessary. More precisely, the graph representation is defined as $G_{\mathcal{C}} := (\{1, \ldots, n\}, E[\mathcal{C}])$ where $E[\mathcal{C}] := \{(u, v) \mid \mathcal{C}_{u,v} \neq 0\}$. In other words, $E[\mathcal{C}]$ is the edge set of the processes that need to communicate with each other. Note that the set contains forward and backward edges, and that the weights of the edges in the graph correspond to the entries in the matrix $\mathcal{C}$.

Our most successful strategy is a top down approach. Intuitively, we want to identify subgraphs in the communication graph of processes that have to communicate much with each other and then place such processes closely, i.e. on the same node, same rack and so forth. In the following, we assume a homogeneous hierarchy of the supercomputer, but our algorithms can be extended to heterogeneous hierarchies in a straightforward way. Let $\mathcal{S} = a_1, a_2, ..., a_k$ be a sequence describing the hierarchy of the supercomputer. The sequence should be interpreted as each processor having a_1 cores, each node a_2 processors, each rack a_3 nodes, ...

The *top down approach* starts by computing a *perfectly balanced* partition of G_C into a_k blocks each having n/a_k vertices (processes). The partitioning task is done using the techniques provided by Sanders and Schulz [12] which provide high quality partitions and guarantee that each block of the output partition has the specified amount of vertices. In principle, the nodes of each block will be assigned completely to one of the a_k system entities. Each of the system entities provides precisely n/a_k PEs. We then proceed recursively and partition each subgraph induced by a block into a_{k-1} blocks and so forth. The recursion stops as soon as the subgraphs have only a_1 vertices left. In the base case, we assign processes to permutation ranks.

3 Graph Format

3.1 Input File Format

The graph format used by our programs is the same as used by Metis [8], Chaco [7] and the graph format that has been used during the 10th DIMACS Implementation Challenge on Graph Clustering and Partitioning [1]. The input graph has to be undirected, without self-loops and without parallel edges.

To give a description of the graph format, we follow the description of the Metis 4.0 user guide very closely. A graph $G = (V, E)$ with n vertices and m edges is stored in a plain text file that contains $n + 1$ lines (excluding comment lines). The first line contains information about the size and the type of the graph, while the remaining n lines contain information for each vertex of G. Any line that starts with % is a comment line and is skipped.

The first line in the file contains either two integers, $n\ m$, or three integers, $n\ m\ f$. The first two integers are the number of vertices n and the number of undirected edges of the graph, respectively. Note that in determining the number of edges m, an edge between any pair of vertices v and u is counted *only once* and not twice, i.e. we do not count the edge (v, u) from (u, v) separately. The third integer f is used to specify whether or not the graph has weights associated with its vertices, its edges or both. If the graph is unweighted then this parameter can be omitted. It should be set to 1 if the graph has edge weights, 10 if the graph has node weights and 11 if the graph has edge and node weights. However, note that since we compute one-to-one mappings, node weights are ignored.

The remaining n lines of the file store information about the actual structure of the graph. In particular, the ith line (again excluding comment lines) contains information about the ith vertex. Depending on the value of f, the information stored in each line is somewhat different. In the most general form (when $f = 11$, i.e. we have node and edge weights) each line has the following structure:

$$c\, v_1\, w_1\, v_2\, w_2 \ldots v_k\, w_k$$

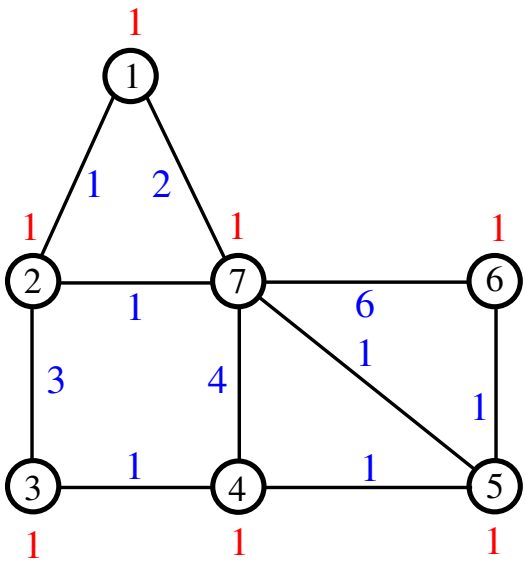
where c is the vertex weight associated with this vertex, $v_1, \ldots, v_k$ are the vertices adjacent to this vertex, and $w_1, \ldots, w_k$ are the weights of the edges. Note that the vertices are numbered starting from 1 (not from 0). Furthermore, the vertex-weights must be integers greater or equal to 0, whereas the edge-weights must be strictly greater than 0.

3.2 Output File Formats

The output format of a mapping is basically a text file named *permutation*. This file contains n lines. In each line the mapped ID of the corresponding vertex is given, i.e. line i contains the mapped processor ID of the vertex i (here the vertices are numbered from 0 to $n - 1$). The processor IDs are numbered consecutively from 0 to $n - 1$.

3.3 Troubleshooting

VieM should not crash! If VieM crashes it is mostly due to the following reasons: the provided graph contains self-loops or parallel edges, there exists a forward edge but the backward edge is missing or the forward and backward edges have different weights, or the number of vertices or edges specified does not match the number of vertices




```
7  10 11
1  2 1 7 2
1  1 1 3 3 7 1
1  2 3 4 1
1  3 1 7 4 5 1
1  4 1 7 1 6 1
1  5 1 7 6
1  1 2 2 1 4 4 5 1 6 6
```

Figure 1: An example graph and its representation in the graph format. The IDs of the vertices are drawn within the cycle, the vertex weight is shown next to the circle (red) and the edge weight is plotted next to the edge (blue).

or edges provided in the file. Please use the *graphcheck* tool provided in our package to verify whether your graph has the right input format. If our graphcheck tool tells you that the graph that you provided has the correct format and VieM crashes anyway, please write us an email.

4 User Interface

Our package contains the following programs: viem, generate_model, graphchecker, evaluator. To compile these programs you need to have Argtable, g++, and scons installed (we use argtable-2.10, g++-4.8.0, and scons-1.2). Once you have that you can execute *compile.sh* in the main folder of the release. When the process is finished the binaries can be found in the folder *deploy*. We now explain the parameters of each of the programs briefly.

4.1 VieM

Description: This is the mapping program. The default configuration of the program uses the top down approach as well as local search based on the communication graph with a neighborhood distances of 10. Note that the number of vertices in the model to be mapped has to be the same as the number of PEs specified using the hierarchy parameter string.

Usage:

viem [--help] file [--seed=<int>] [--preconfiguration_mapping] --hierarchy_parameter_string=<string> --distance_parameter_string=<string> [--construction_algorithm=<string>] [--distance_construction_algorithm=<string>] [--local_search_neighborhood=<string>] [--communication_neighborhood_dist=<int>]

Options:

file	Path to file (model).
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration_mapping=<string>	Use a preconfiguration for the partitioning tool within the mapping algorithm. One of strong, eco or fast. Default: eco.
--construction_algorithm=<string>	Initial construction algorithm to use. One of random, identity, growing, hierarchybottomup, hierarchytopdown. Default: hierarchytopdown.
--distance_construction_algorithm=<string>	Construction algorithm to use to initially construct the distance matrix. Use one of hierarchy or hierarchyonline which does not store distance matrix. Default: hierarchy.
--hierarchy_parameter_string=<string>	Specify hierarchy as 2:2:... for 2 cores per PE, 2 PEs per node, and so forth.
--distance_parameter_string=<string>	Specify distances between different levels as 1:10:... for 2 cores on the same PE have distance 1, and so forth
--local_search_neighborhood=<string>	Local search neighborhood to use nsquare, nsquarepruned, or communication. Default: communication
--communication_neighborhood_dist=<int>	set the communication neighborhood distance. Default: 10.
--output_filename=<string>	Specify the output filename (default permutation).

4.2 Generate Model of Computation and Communication

Description: This is program is for testing purposes. It takes a graph as input, partitions it using KaHIP [11, 13] and then creates a model of computation and communication. Here, blocks of the partition are vertices in the model and edge weights in the model are set to the number of edges that run between the respective blocks.

Usage:

```
generate_model  file --k=<int> [--help] [--seed=<int>] [--preconfiguration=variant]
                [--imbalance=<double>] [--output_filename=<string>]
```

Options:

file	Path to graph file that you want to partition and build the model from.
--k=<int>	Number of blocks to partition the graph into, i.e. number of vertices in the model.
--help	Print help.
--seed=<int>	Seed to use for the random number generator.
--preconfiguration=variant	Use a preconfiguration for partitioning. (Default: eco) [strong| eco | fast | fastsocial| ecosocial| strongsocial]. Strong should be used if quality is paramount, eco if you need a good tradeoff between partition quality and running time, and fast if partitioning speed is in your focus. Configurations with a social in their name should be used for social networks and web graphs.
--imbalance=<double>	Desired balance. Default: 3 (%).
--output_filename=<string>	Specify the output filename (default model.graph).

4.3 Graph Format Checker

Description: This program checks if the graph specified in a given file is valid.

Usage:
graphchecker file

Options:

file Path to the graph file.

4.4 Evaluator

Description: This program takes a model and a specification of the system hierarchy as well as a mapping of vertices of the model to processors in the system. It then computes the QAP objective.

Usage:
evaluator [--help] file --input_mapping=<string> --hierarchy_parameter_string=<string>
--distance_parameter_string=<string>

Options:

--help	Print help.
file	Path to file (graph/model).
--input_mapping=<string>	Input mapping to use.
--hierarchy_parameter_string=<string>	Specify hierarchy as 2:2:... for 2 cores per PE, 2 PEs per node, and so forth.
--distance_parameter_string=<string>	Specify distances between different levels as 1:10:... for 2 cores on the same PE have distance 1, and so forth

References

[1] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[2] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.

[3] R. E Burkard, E. Cela, P. M. Pardalos, and L. S. Pitsoulis. The quadratic assignment problem. In *Handbook of combinatorial optimization*, pages 1713–1809. Springer, 1998.

[4] Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proc. of the 3rd Intl. Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86. Springer, 1996.

[5] J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proc. of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.

[6] C. H. Heider. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Technical report, DTIC Document, 1972.

[7] B. Hendrickson. Chaco: Software for Partitioning Graphs. `http://www.cs.sandia.gov/~bahendr/chaco.html`.

[8] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[9] H. Müller-Merbach. *Optimale reihenfolgen*, volume 15 of *Ökonometrie und Unternehmensforschung*. Springer-Verlag, 1970.

[10] S. Sahni and T. F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976.

[11] P. Sanders and C. Schulz. KaHIP – Karlsruhe High Qualtity Partitioning Homepage. `http://algo2.iti.kit.edu/documents/kahip/index.html`.

[12] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th Intl. Sym. on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013.

[13] Peter Sanders and Christian Schulz. Kahip v0.53 - karlsruhe high quality partitioning - user guide. *CoRR*, abs/1311.1714, 2013.

[14] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

[15] Christian Schulz and Jesper Larsson Träff. Better process mapping and sparse quadratic assignment. *CoRR*, abs/1702.04164, 2017.