TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTER'S THESIS

Distribution of IMUNES System - Graph Partitioning

Developed at Institute for
Distributed Systems
University of Vienna

Under the guidance of Ao. Univ. Prof. Dr. Helmut Hlavacs

Petra Schilhard

Date

Signature

Abstract

In my master's thesis, I have implemented a graph partitioning heuristic in IMUNES system. IMUNES is a program for network emulation/simulation. Since network emulation and simulation are computationally very intensive, a method for dividing the computational load between distributed processors, in a way that minimizes interprocessor communication, is required. The emulation in IMUNES is represented as graph nodes interconnected with links, so the problem of dividing the computational load between processors is reduced to dividing the nodes and edges of the graph, called *graph partitioning*.

After METIS graph partitioning algorithm, I implemented the graph partitioning which has three stages. In the first stage, coarsening, a hierarchy of approximations to the original problem is created. In the second stage an initial solution to the problem is found, which is then iteratively refined in the third stage.

In the main part of this thesis I describe the implementation of the three stages of the multilevel partitioning scheme in IMUNES. The main goal in my thesis was to implement a real-time job partitioning in IMUNES system. After implementation, I conducted some performance and quality evaluation. Experiment results, at the end of the thesis, show that implemented methods achieve the same quality level of partitions as when derived by the METIS method, but require more execution time because of a trade-off between speed and platform independence by using a Tcl/Tk script language instead of some compiled programming language.

Contents

1 Introduction

Network simulation and emulation is a valuable tool for testing, understanding and evaluating research prototypes of an existing or imaginary network system. Mostly, a simulation of the target network is also simpler and cost more effective: only a software model of the target system has to be created, which can be run on any hardware. On the other hand, it can be very hard or in some cases impossible to create an exact model of the target system and to simulate the real network conditions.

There are two major classes of experimental environments:

- Simulation environments – allow the user to predict the behaviours of network protocols and applications under different situations using an internal model specific to the simulator. Simulated networks can typically scale well in size, but simulated devices may have limited functionalities and their behaviour may be different than the real devices would have.
- Emulation environments – closely reproduce the features and behaviour of real hardware, while having implemented virtual network configurations atop of it: this means that experimenters can use real operating systems and other software, and run their applications unmodified. Differently from simulation systems, in an emulator the network undergoes the same state changes that would occur in the real word.

My thesis focuses on the fundamental concepts of emulation environments, and provides a brief introduction of IMUNES [2], an open source software emulator.

The primary challenge for emulation environments is the emulated network size scalability. Because emulated environments are supported by real hardware, the computational resources needed to run an emulated system may be higher than the available underlying physical system can provide. This requires the careful allocation and mapping (multiplexing) from the virtual resources to available physical resources. The simplest solution is to map virtual network nodes and links one-to-one onto dedicated physical resources. This is an inefficient use of shared hardware, because most of emulated resources use just a small part of the available resources. A more complex method is a controlled multiplexing of virtual onto physical resources. Inadequately mapped resources can degrade performance or introduce artefacts into an experiment [13].

As noted before, the emulated complex network topologies demand a lot of processing power. One of the effective ways of doing it is by dividing the work among distributed processors. This division is done through a process known as *graph partitioning*. The problem is to divide the vertices of a given graph into roughly equal and disjoint parts called *partitions*, with a minimal number of edges connecting the vertices in different partitions.

Graph partitioning is an NP-complete problem, for which many algorithms have been developed. Some of them produce very good partitions, but are very expensive in processing power, and some tend to be fast, but often produce bad partitions.

Currently there are a few packages with graph partitioning algorithms. One of the well-known is METIS [3]. METIS is developed by George Karypis and Vipin Kumar, at the University of Minnesota.

For my master's thesis I have implemented an algorithm that divides the workload in IMUNES after the METIS methods. First the algorithm analyses the graph and extracts required information, such as the number of nodes and links, their types, connections etc., in order to divide the graph better according to its characteristics. After the algorithm has collected all required pieces of information, the first stage begins. In the fist stage, it uses Heavy-edge algorithm (HEM) to generate smaller graphs; in the second stage it uses Graph growing algorithm (GGP) to divide the graph in partitions; and in the third stage it uses Fiduccia and Mattheyses (FM) improvement of the Kernighan-Lin algorithm (KL) to refine the partitions. Additionally, several other algorithms for each stage of the graph partitioning are also described.

2 An Overview of Emulation Environments

There are a lot of emulation products available, which differ in the type of the device they emulate, emulation technique, the size of virtual networks they can handle, distribution license, etc. In this section description of some of the best-known emulation environments are given: PlanetLab [13], Emulab [14] and ModelNet [15]. All three network emulators run experiments involving a set of geographically distributed nodes.

2.1 PlanetLab

PlanetLab is a network of computers geographically distributed at sites around the world, forming a global platform for supporting distributed systems and network research. This network of computers is called “PlanetLab nodes”. PlanetLab started 2002, and until the time of writing this thesis it spans over eight hundred nodes at almost four hundred sites [13]. To use the PlanetLab network one can either have an account, which are limited to persons affiliated with universities and corporations hosting PlanetLab nodes, or it can use one of the free public services. The members of PlanetLab develop the tools for the community, and as a result each user has a wide choice of tools to use.

Each research project, that uses the PlanetLab network, has a virtual machine access to a subset of the nodes. The set of virtual machines is called a “slice”. The slice is created through PlanetLab Central (PLC), and it represents a container in which owner's services are placed. After the PLC creates and assigns a slice to the user, the user may assign nodes to the slice. After assigning nodes to the slice, virtual servers for that slice on every assigned node are created. Slices expire after two months, and if not extended, are destroyed and all files in the slices are removed.

PlanetLab’s main characteristics are centralized trust authority and resource control.

PlanetLab Central (PLC) is a centralized trust design as a trusted intermediary between node owners and node users. PLC controls everything involving the public PlanetLab (permissions, resource allocation, etc.), and saves administrative information primarily about sites, users, and nodes, and information about slices running on the infrastructure. The administrators are limited to customize their systems and the owners of individual nodes have very limited control over their own resources. One of the advantages of centralization the control is a uniform view of the system.

A set of allocated resources on a single PlanetLab node is called “sliver”, and is implemented as Linux-VServer (Virtual Server). Network virtualized access of sliver on PlanetLab nodes is implemented using VNET (Virtualized Network Access). The goal of VNET is to be as transparent as possible, allowing PlanetLab users not to have to recompile any code.

2.2 Emulab

Emulab is a widely available, remotely accessible, network emulation laboratory (testbed). It is developed at the University of Utah. Emulab consists of both a facility and a software system. To start a “project” at Emulab for research or educational use almost any university, industrial research labs, US and non-US institutions are eligible. After the project is approved, the researcher is permitted to authorize additional project members and grants them access to the project.

Emulab facility is made of a cluster of interconnected PCs. Each PC is connected with four network interfaces to a switching fabric (a set of Cisco switches). This enables the users to build arbitrary network topologies without the need to physically move any wires. Emulab contains a wide range of experimental resources, including wired cluster PCs that can run operating systems like Linux, FreeBSD and Windows, physically distributed 802.11 wireless PCs, network processors, sensor boards, and software radio platforms.

A user can define an experiment by using a GUI tool or directly in NS simulation language. The definition includes description of the network topology, link characteristics, operating systems, and other software which runs on all machines. When a defined experiment is started, a set of machines is allocated for it, the network topology on the switches is configured, and the custom disk images are loaded. When a user starts an experiment, most of resources are allocated at a time only to that user. Thus, when Emulab is full, new users cannot enter the system. In Emulab the user of a started experiment has complete control over the allocated resources.

Emulab today has a portal to PlanetLab, allowing Emulab users to allocate slices on PlanetLab. Combination of both testbeds allows for much richer experimentation and deployment. This combination can currently be used only from the Emulab's side, and not from the PlanetLab's. More about the combining the two testbeds can be found under [35].

2.3 ModelNet

Modelnet is a software emulator developed at the University of California, San Diego. It allows building distributed network experiences running on realistic Internet-like environments. ModelNet enables researchers to deploy unmodified software prototypes in a configurable environment.

The architecture of the ModelNet network comprises of *Edge Nodes* and *Core Nodes*. User specified operating systems and application software runs on a *virtual node*. Virtual nodes are multiplexed across a set of physical machines that are called edge nodes. Each virtual node has a unique IP address. Edge nodes are configured to route their packets through a scalable cluster of core nodes. One core node consists of one or more physical machines, which are physically interconnected by gigabit links. The core nodes have modified FreeBSD kernels that enable them to emulate the behaviour of a target network under the offered traffic load. Each core node routes the packet through an emulated network of pipes and subject it to the bandwidth, loss and latency of a target network topology. The emulation runs in real time, so packets traverse the emulated network with the same rates, delays and losses as in the real network.

A typical cluster machine has mostly far more CPU power and network bandwidth than a single instance of the user application requires. Because of this, ModelNet creates, possibly, hundreds of virtual nodes on each application host.

ModelNet runs in four phases. The first phase, *Create*, generates a target network topology specified by a user. A topology is a graph whose nodes represent clients or intermediate nodes in the network, and whose edges represent network links. The current version of ModelNet supports few generation tools for topology generation. Next, the network topology is transformed to a pipe topology that models the target network. The second phase, *Assignment*, maps pieces of the pipes to core nodes, partitioning the pipe graph to distribute emulation load across the core nodes. The *Binding* phase assigns virtual nodes to the edge nodes and sets them up for executing the applications. Multiple VNs are multiplexed onto each edge node, and edge nodes are bound to a single core node. The final phase, *Execution*, executes target application code on the edge nodes.

3 IMUNES

IMUNES (Integrated Multiprotocol Network Emulator/Simulator) [2] is a lightweight network emulator developed at the University of Zagreb. Instead of running entire kernels as processes, it modifies a FreeBSD OS kernel to allow running multiple lightweight virtual nodes on a single operating system kernel. This kernel-level emulation allows efficient traffic manipulation, since it replaces frequent and expensive data coping and context switching by moving packets only by reference. The virtual nodes are interconnected via kernel-level links to form arbitrarily complex network topologies. In IMUNES, each virtual node offers equally rich set of capabilities as the standard kernel does, and can be efficiently implemented by reusing the existing OS kernel code. User-level applications are executed within virtual nodes with the illusion of running on a single workstation with its own network interfaces, thus they need not to be adapted for the new environment.

IMUNES is an open source project under GNU license.

3.1 The Architecture

The conventional organization of UNIX operating systems kernel and network stack is shown in Figure 1.

In UNIX operating systems the computer's memory is partitioned into two disjointed areas, one reserved for the kernel (*kernel space*) and the other for the applications (*user space*). All the user processes share a network facility and addressing space, which presents some limits to certain emerging applications, like for virtual hosting.

The idea of reusing and extending an existing OS network stack, as well as the concept of virtual machines is not new. Some of the different variations can be found in [27].

In IMUNES, a kernel is modified to allow multiple independent network stack instances to co-exist simultaneously within one OS kernel. The model of modified OS kernel in IMUNES is shown in Figure 2.

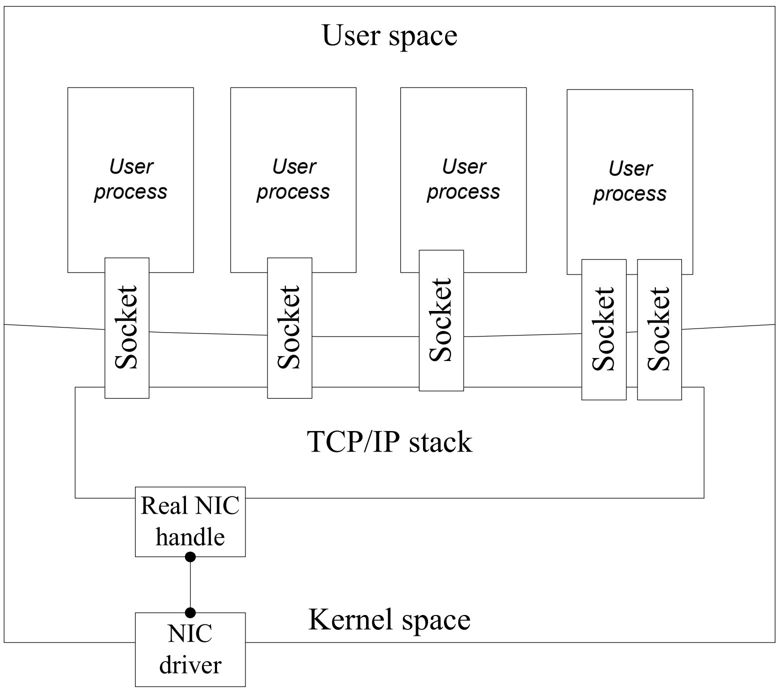


Figure 1: Common organization of UNIX OS kernel

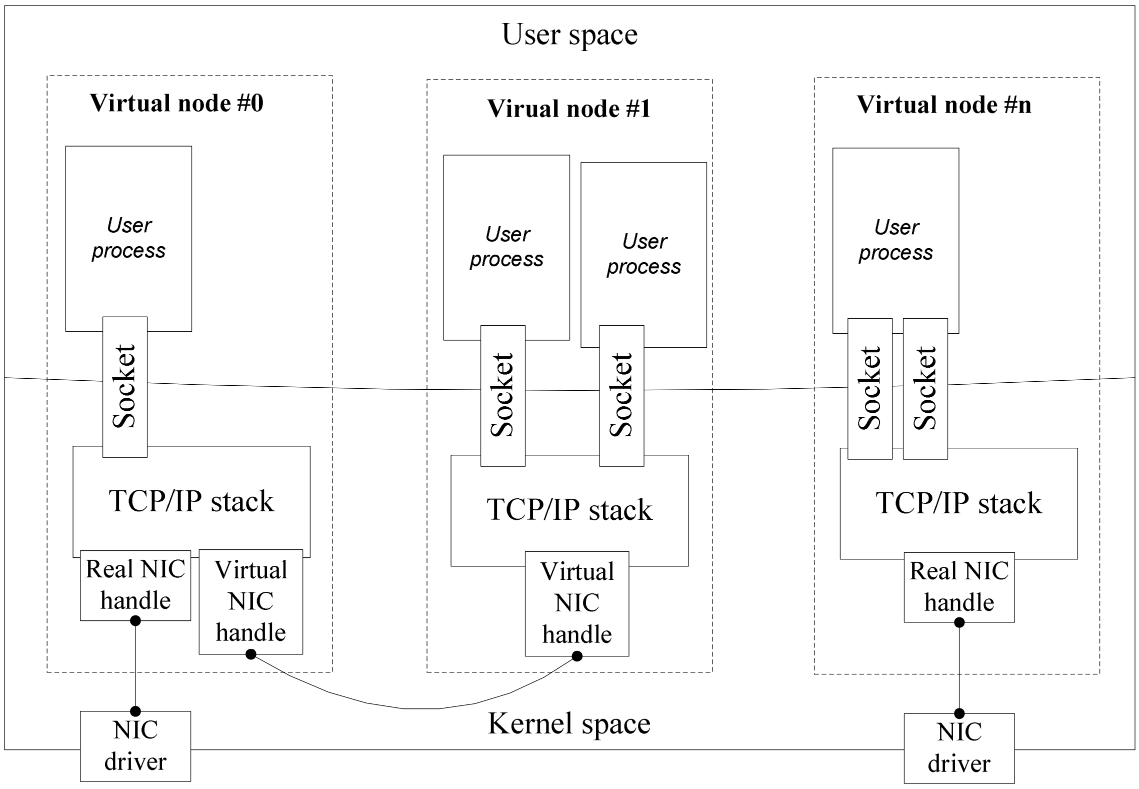


Figure 2: Modified OS kernel image in Imunes [37]

The key element of IMUNES emulation model is a virtual node. A virtual node is an isolated kernel entity, which consists of a group of associated user processes and a network stack instance. Each existing user process is associated with at most one network stack instance at a time. Each network stack instance is functionally independent, and maintains its private variables like routing tables, routing cache, list of network interfaces, communication sockets, as well as a set of optional networking facilities like packet filters, traffic shapers, etc. The standard user applications can be executed without any modifications, recompiling or runtime library replacement.

Upon the startup, the system creates one (default) virtual node, which contains the real network interface and all current user processes. The subsequent created processes are automatically associated with the default virtual node and real network interface. After startup the system administrator can dynamically create additional virtual nodes, associate them with real or pseudo network interfaces and create or reassign user processes into their environments. Different network stacks need explicit methods for establishing communication between them. To provide the methods for it IMUNES extends the standard netgraph framework in the FreeBSD kernel. User processes running in one virtual node are able to interact only with their own network stack instance and interfaces associated with its own virtual node. Communication between virtual nodes is accomplished either via bridged virtual or physical Ethernet interfaces, or through virtual point-to-point channels constructed from a pair of FreeBSD's standard netgraph interface nodes. The bridging method is entirely hidden inside of the kernel, allowing transparent connection to virtual nodes from the outer world via one or more physical Ethernet interfaces. Passing packets between two virtual nodes is done only by reference, thus providing zero coping.

Figure 3 illustrates the traditional virtualization model (a) and in IMUNES implemented virtualization model (b).

3.2 IMUNES Topology

IMUNES implements a simple Tcl/Tk based graphical topology editor and a topology compiler. Graphical topology editor in IMUNES allows for quick creation and viewing experiments consisting of two basic units: nodes and links. Nodes can exist independently, but for a link to exist, two different nodes must be available.

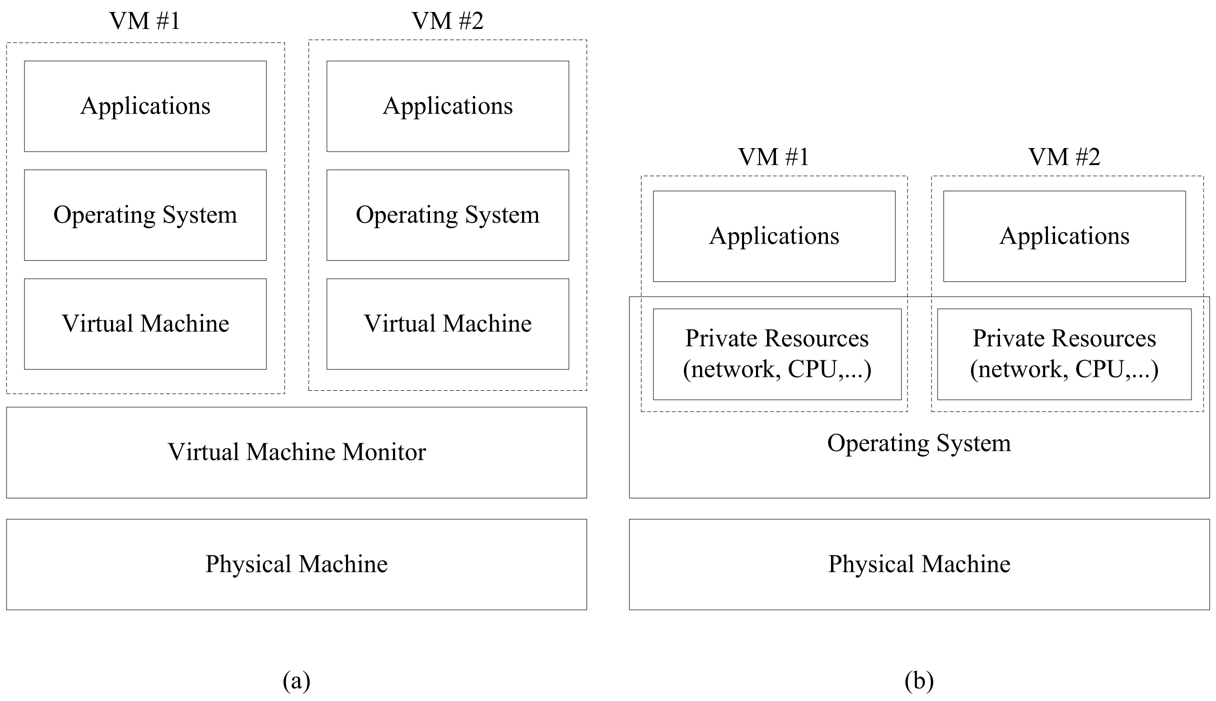
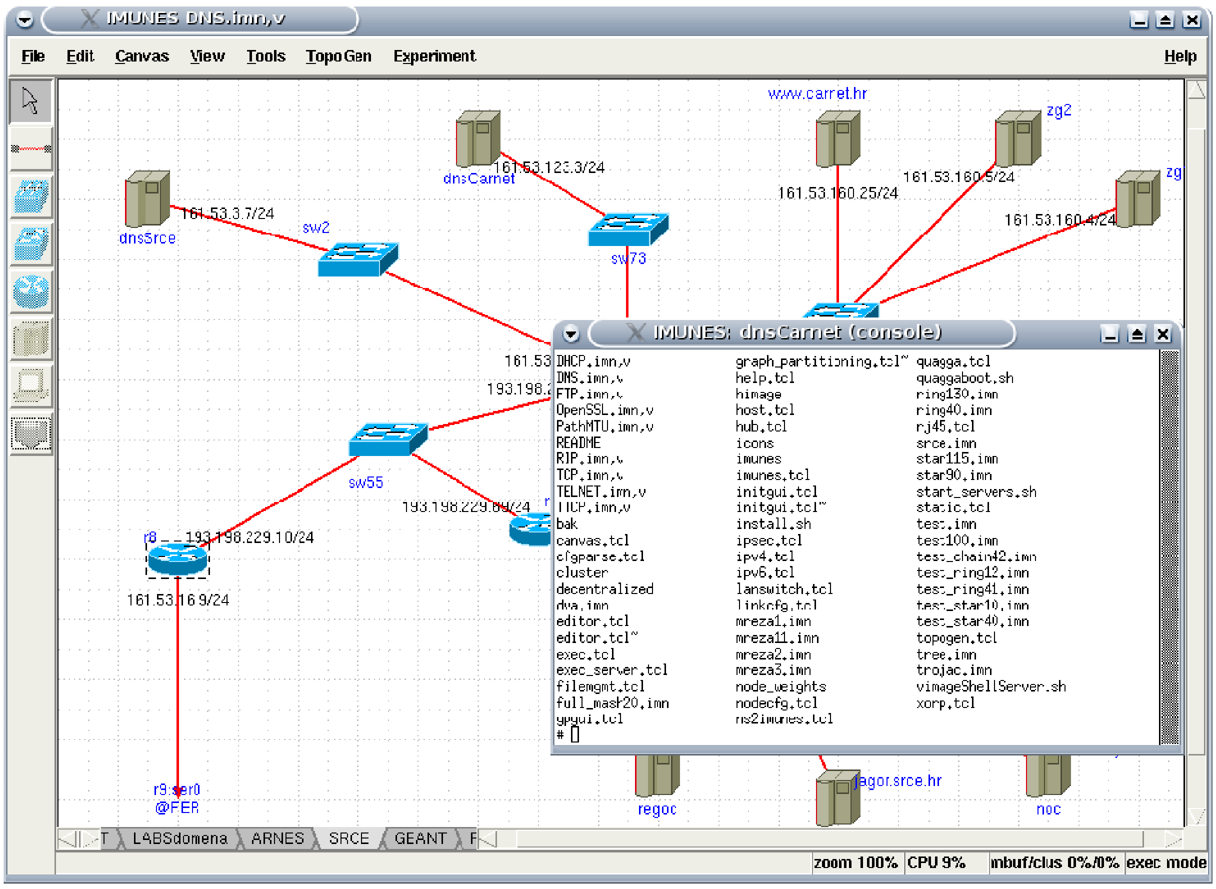


Figure 3: (a) Traditional virtualization model, (b) lightweight vm model -- packet going from vm1 to vm2 is copied ...) [27]

Nodes are made of network layer nodes: PCs, hosts, routers, and link layer nodes: switches, hubs and physical interfaces (RJ45). Link layer nodes provide just link layer functionality (data transfer between network entities), and are not implemented as virtual nodes but are just netgraph nodes. Network layer nodes provide network layer functionality (routing, fragmentation, reassembly, etc.), and are implemented as virtual nodes in a kernel. Each network node is a fully functional FreeBSD system with its own instance of a network stack. Host and PC types of nodes differ from router node in the type of routing. Router is capable of packet forwarding and uses dynamic routing; host and PC are not capable of packet forwarding and use static routing. Each router runs xorp [36] or quagga [24] open source routing software on a virtualized networking stack.

Links always connect two different nodes, and no two different links connect the same two nodes. Each link has its characteristics like link type, link bandwidth, BER, delay, etc. According to type, a link can be Ethernet (default) or serial, whereas all links are presumed to be full duplex. Links are also implemented as netgraph nodes.

Graphical editor simplifies creating, configuring and interconnecting the nodes in the target topology. During the emulation execution the services on different nodes can be started, terminated or changed through a command line interface of each node. An example of IMUNES graphical editor and an open shell on one node is shown in Figure 4.



```
DHCP.imn,v          graph_partitioning.tcl~ quagga.tcl
DNS.imn,v           help.tcl                quaggaboot.sh
FTP.imn,v           himage                  ring130.imn
OpenSSL.imn,v       host.tcl                ring40.imn
PathMTU.imn,v       hub.tcl                 rj45.tcl
README              icons                   srce.imn
RIP.imn,v           imunes                  star115.imn
TCP.imn,v           imunes.tcl              star90.imn
TELNET.imn,v        initgui.tcl             start_servers.sh
TTCP.imn,v          initgui.tcl~            static.tcl
bak                 install.sh              test.imn
canvas.tcl          ipsec.tcl               test100.imn
cfgparse.tcl        ipv4.tcl                test_chain42.imn
cluster             ipv6.tcl                test_ring12.imn
decentralized       lanswitch.tcl           test_ring41.imn
dva.imn             linkcfg.tcl             test_star10.imn
editor.tcl          mreza1.imn              test_star40.imn
editor.tcl~         mreza11.imn             topogen.tcl
exec.tcl            mreza2.imn              tree.imn
exec_server.tcl     mreza3.imn              trojac.imn
filemgmt.tcl        node_weights            vimageShellServer.sh
full_mask20.imn     nodecfg.tcl             xorp.tcl
gpgui.tcl           ns2imunes.tcl
# 
```

Figure 4: IMUNES GUI with a sample network topology and open shell on one node

4 Basic Notations and Definitions

The notation used in this and subsequent lectures is here described.

The graph $G = (V, E, W_v, W_e)$ is an undirected, weighted graph, without self edges *(e, e)* or multiple edges from one vertex to another. *G* is subject to the following conditions:

- *V* is a set of vertices in graph *G*, whose elements are called *nodes* or *vertices*
- *E* is a set of edges connecting the vertices $e = (v_1, v_2),\ where\ e \in E,\ v_1, v_2 \in V$. Elements of the set E are called *edges* or *lines.*
- W_v a set of weights of each vertex $v \in V$
- W_e a set of weights of each edge $e \in E$. The notation $|V|$ means the number of vertices in *V*, and $|E|$ the number of edges in *E*

Two vertices are adjacent if they are connected by an edge. Adjacent nodes are also often called neighbours.

If $e = (v_1, v_2),\ where\ e \in E \wedge v_1, v_2 \in V$, then v_1 and v_2 are adjacent and neighbours, and to an *e* incident.

A connected or complete graph is the one in which every vertex is through one or more edges connected with every other vertex in the graph, otherwise it is called disconnected. An isolated vertex is one which has no neighbours, i.e. it is not connected to any other vertex.

A *partition* of the graph *G* is the mapping of the set of vertices *V* into *P* disjoint sets V_p, so that $\bigcup V_p = V$. A partition is "good", if the vertices mapped to disjoint sets are highly similar, the weight between different sets is minimal and the weight within sets is maximal.

An *edge-cut* of *G* is a set of edges, whose incident vertices belong to different partitions, and whose removal leaves the graph disconnected.

5 Related Work

Many algorithms have been developed for the graph partition problem. Partition heuristics, described here, use graph bisection, i.e. they first bisect the graph *G* into two parts G_1 and G_2, ($G = G_1 \cup G_2$), and then in each further recursion subdivide each part again in two partitions.

There are two classes of heuristics, depending on the information available about the graph *G*. In the first case, the geometric information of the graph is used to find a good partition. This type of heuristic is called *geometric partitioning* algorithm. Geometric algorithms tends to be fast, but produce slightly worse partitions than spectral methods, and need to have coordinates for the vertices of the graph. In many problem areas (e.g. linear programming) there is no geometrical information about the graph.

The other class of heuristics doesn't have geometry associated with the graph. These kinds of graphs require combinatorial rather than geometric algorithms to partition them. *Spectral partitioning* algorithms produce very good partitions for a wide class of problems but are also very expensive. They are a class of techniques used to solve partial differential equations, often involving the use of the Fast Fourier Transform. Spectral method can be accelerated by using a *multilevel algorithm*. The idea behind the multilevel algorithms is to reduce the size of the graph by collapsing vertices, partitioning the smallest graph, and then extending the partition to construct the original graph. Multilevel algorithms are faster than spectral ones, but produce worse partitions.

First heuristics for minimum bisection were given by Kernighan and Lin [10], enhanced by Hendrickson and Leland [1] by applying vertex and edge weights, and later further improved by Fiduccia and Mattheyses [11].

Well known software packages such as Chaco [12] and METIS can be used for graph partitioning, which apply heuristics to generate approximate solutions. Algorithms in METIS are based on multilevel graph partitioning. In IMUNES system the implementation of graph partitioning is based on METIS partitioning algorithms.

5.1 CHACO

Chaco is a software package of various graph partitioning algorithms, available under license from Sandia National Laboratories. It can be used either as a stand-alone application or as a library of graph partitioning functions that can be linked from other applications. It addresses three types of problems: graph partitioning, embedding the partitions into different topologies, and the use of spectral methods to sequence the graphs.

Chaco implements five classes of partitioning algorithms: simple, inertial, spectral, Kernighan-Lin (KL), and multilevel-KL (see *Initial partitioning*). Those methods are categorized as global or local (Kernighan-Lin). A "local" method is also referred to as "improvement heuristic", because it takes a partitioned graph and attempts to improve the quality of the partitions. After [28] the combination of global and local methods leads to significant improvements in performance and robustness.

Most graph partitioning algorithms divide the graph recursively into two halves (Bisection) until the desired number of partitions has been reached. Chaco additionally offers to divide the graph recursively into four (Quadrisection) or eight (Octasection) parts.

To minimize the total communication time between sets Chaco uses the minimization of the total number of *cuts*, and the minimization of the total number of *hops*. The total number of hops takes into account the "architectural distance" between the sets, that is, the physical distance of the processors or regions of an integrated circuit, to which the partitions are to be mapped.

5.2 METIS

METIS is a set of programs for partitioning large irregular graphs, computing fill-reducing orderings of sparse matrices, and for partitioning large hypergraphs. The algorithms in METIS are based on the state-of-the-art multilevel paradigm (see 6.1).

METIS is a family of software packages consisting of three different packages: METIS, ParMETIS, and hMETIS. METIS is designed for partitioning large unstructured graphs on serial computers. ParMETIS (Parallel METIS) extends the functionality provided by METIS by parallelizing of METIS serial methods. hMETIS (Hypergraph METIS) is a set of programs for partitioning large unstructured hypergraphs and very large circuits. METIS and hMETIS are developed by George Karypis, and ParMETIS by George Karypis and Kirk Schloegel. METIS family of algorithms is freely available on the web [29].

In the following sections, the METIS algorithms, and the implementation of such algorithms in IMUNES system are described.

6 Graph Partitioning

The problem of partitioning a graph into *k* partitions is defined as follows: Given a graph $G = (V, E)$, with nodes *V* and edges *E* connecting them. The nodes represent data or tasks and edges represent communication between the nodes. The goal is to partition *V* into *k* equal sized subsets, $V_1, V_2, \ldots, V_k$ such that

i. $V_i \cap V_j = \emptyset, \forall i \neq j$

ii. $|V_i| \approx \frac{n}{k}$, where $|V| = n$

iii. $\bigcup V_i = V, \quad i \in \{1, \ldots, k\}$

while minimizing the capacity of the edge cut. The first condition (i) denotes that the two subsets can't have a shared node. The second condition (ii) denotes that each subset has approximately equal number of nodes. And the last condition (iii) denotes that the union of all subsets is a set *V*.

If the graph to be partitioned has weights associated with the nodes and the edges, the goal of partitioning is to have *k* disjoint subsets such that the sum of the node weights in each subset is the same, and the sum of the edge weights, whose incident nodes belong to different subsets, is minimized.

Graph partitioning is an important problem that has extensive applications in many areas including scientific computing, VLSI design, and task scheduling. It is NP-complete (Garey and Johnson, 1979), so in recent years much attention has been given to developing efficient heuristics.

6.1 Multilevel Graph Bisection

The idea behind the multilevel graph bisection is quite simple. There are three different stages to multilevel partitioning algorithm. First, in the initial level the nodes of the graph *G* are matched and in pairs coalesced to define a new coarse graph. Secondly, the smallest graph in the sequence of coarsed graphs is partitioned. And thirdly, the partition is propagated back towards the original graph (finer graph) through the sequence of larger and larger graphs. The refinement is successively applied on the graph in each sequence. Note that the structure of the graph progresses from a fine to a coarse one, and then back to a fine resolution. The algorithm uses edge and node weights to preserve in the coarse graphs as much structure of the original graph as possible.

The structure of multilevel graph partitioning algorithm is outlined in the following table.

```
(1) Until graph is small enough
        graph := coarsen(graph)
(2) Partition graph
(3) Until graph = original graph
        graph := uncoarsen (graph)
        partition := uncoarsen(partition)
        locally refine partition if desired
```

Table 1: Structure of multilevel graph partitioning [1]

Example of multilevel graph partitioning scheme:

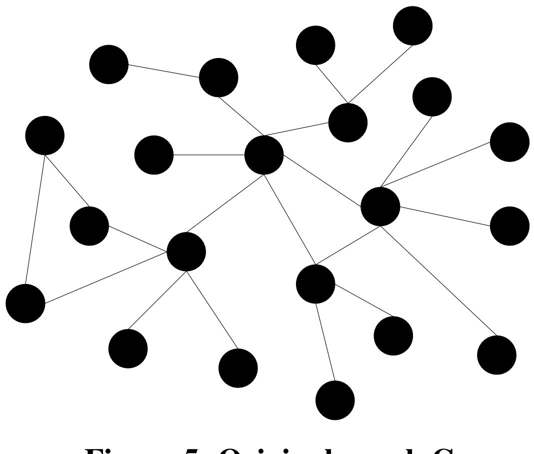


Figure 5: Original graph G_0

Figure 5 illustrates a sample graph G_0, which is to be partitioned in 3 parts. G_0 consists of 21 nodes, so each partition should contain 7 nodes.

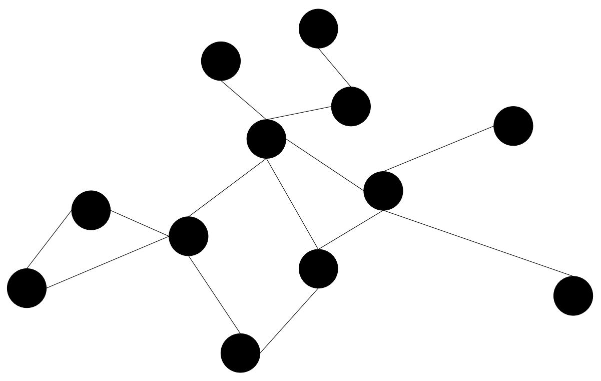


Figure 6: Coarsed graph G_1

Figure 6 illustrates a graph after the first pass of coarsening method. G_1 is coarsed to 12 nodes.

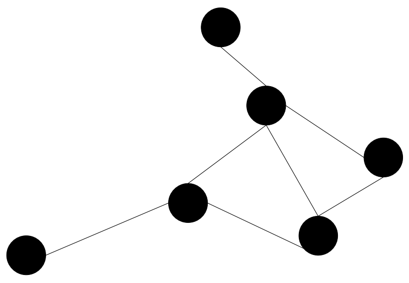


Figure 7: Coarsed graph G_2

Figure 7 illustrates a graph after the second pass of coarsening method. G_2 consists of 6 nodes.

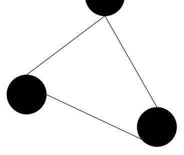


Figure 8: Coarsest graph G_3

In Figure 8 is shown the graph after the last pass of the coarsening method; the coarsest graph G_3. In the next step, the coarsest graph G_3 is partitioned into 3 partitions.

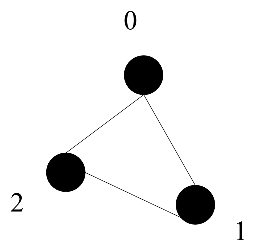


Figure 9: Partitioning the coarsest graph G_3

The partitioning is in this case trivial, because the graph consists of 3 nodes which need to be assigned to 3 partitions. Partitioning methods assign to each node in G_3 one of the three partition, as shown in Figure 9.

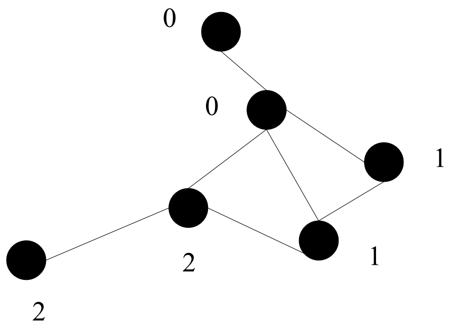


Figure 10: Uncoarsed graph G_2 with partitions

Next, the coarsest graph G_3 with partitions is uncoarsed to the graph G_2, and the partitions in G_3 are projected to G_2. Figure 10 illustrates the graph G_2 with projected partitions.

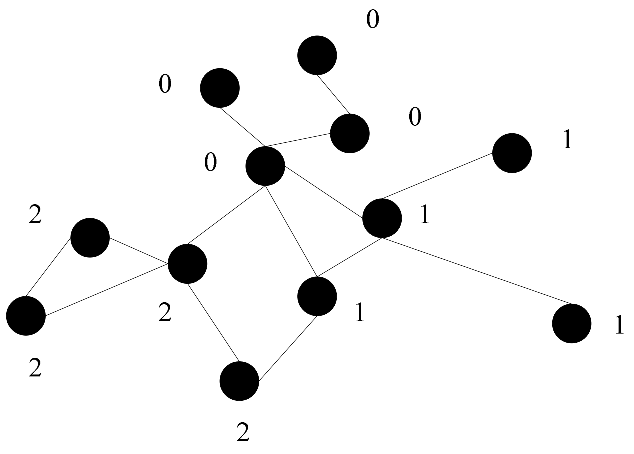


Figure 11: Uncoarsed graph G_1 with partitions

In the next uncoarsening step, the graph G_1 is uncoarsed from the graph G_2, and the partitions in G_2 are projected to the G_1, as it can be seen in Figure 11.

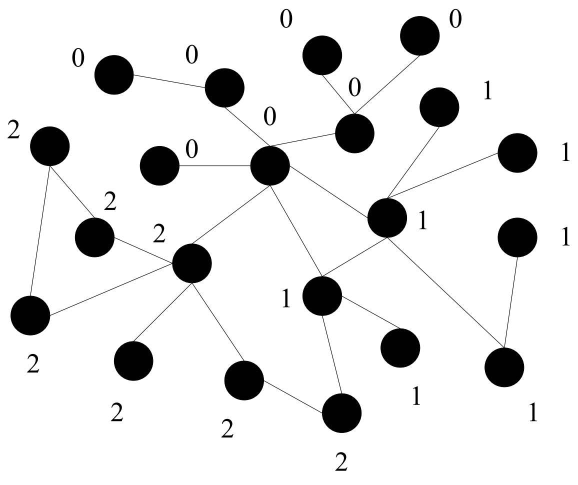


Figure 12: Original graph G_0 with partitions

In the last step of the algorithm (see Figure 12), the graph G_1 is uncoarsed to the original graph G_0, and its partitions are projected to the G_0. Each partition consists of 7 nodes.

6.2 Graph Partitioning Metrics

In this thesis several metrics for evaluating the quality of graph partitioning are used.

Edge-Cut

Edge-cut is the sum of weights of all edges whose incident vertices belong to different partitions. It represents the cost of communication between different processors.

Size of each partition

The size of partition measures the number of all vertices in a partition. In the unweighted graph the closer the sizes of all partitions are the more balanced the partitions are.

Weight of each partition

The weight of a partition is the sum of all vertices' weight in a partition. In the weighted graph the closer the weights of all partitions are the more balanced the partitions are.

Execution Time

Execution time is the amount of time required for graph partitioning.

6.3 Graph Partitioning in IMUNES

The algorithm for graph partitioning in IMUNES is integrated in the IMUNES system. Like the rest of the GUI, the algorithm is also written in Tcl/Tk script language, to make it platform independent.

Starting graph partitioning

Graph partitioning can be started through the IMUNES system, by choosing a "Graph partitioning" option under the "Tools" submenu in the main menu. This opens a new window *Graph partitioning* (see Figure 13) with the partitioning settings.

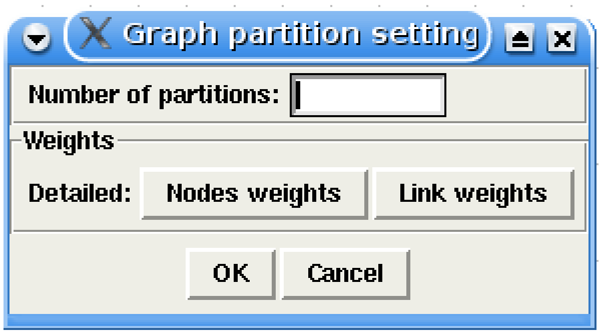
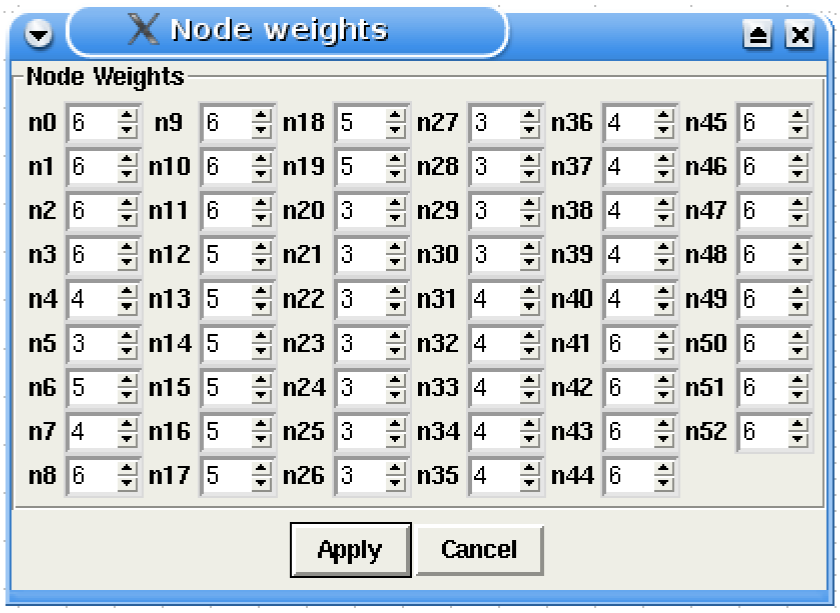


Figure 13: Graph partition settings dialog

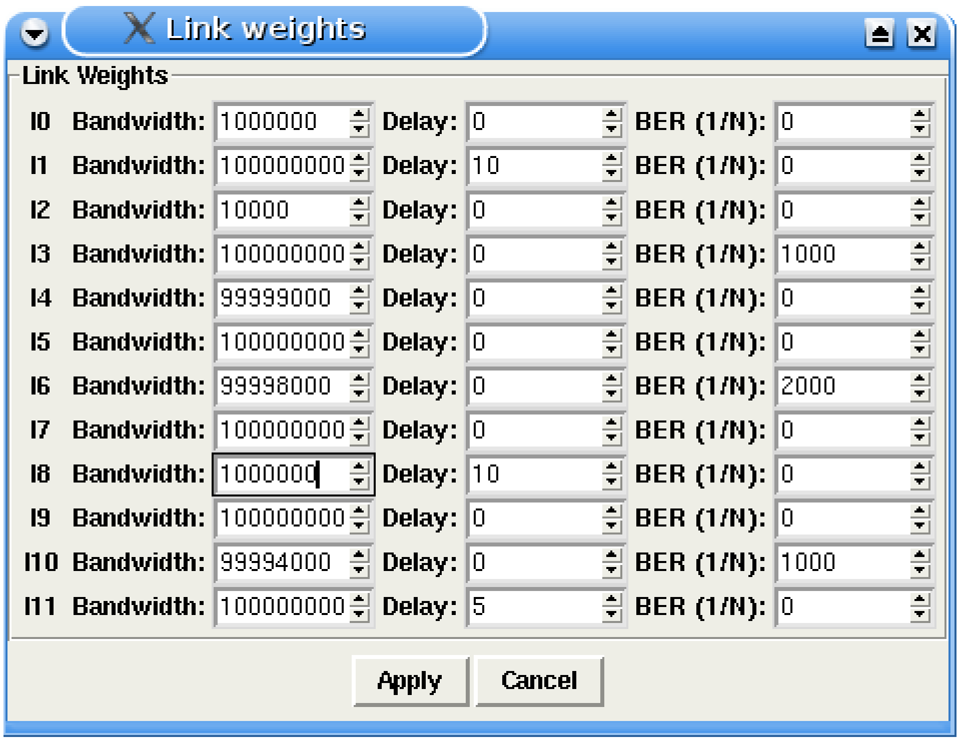
In the upper part of the *Graph partitioning* window, the number of parts, in which the topology is to be partitioned, must be entered. The number of partitions must be greater than 1 and less than the number of nodes in the topology. In the middle part of the window, two buttons "Node weights" (see Figure 14) and "Link weights" (see Figure 15) open two new dialogs for changing the weight of each individual node and link. The algorithm starts when the button *OK* is pushed.



Node Weights

n0	6	n9	6	n18	5	n27	3	n36	4	n45	6
n1	6	n10	6	n19	5	n28	3	n37	4	n46	6
n2	6	n11	6	n20	3	n29	3	n38	4	n47	6
n3	6	n12	5	n21	3	n30	3	n39	4	n48	6
n4	4	n13	5	n22	3	n31	4	n40	4	n49	6
n5	3	n14	5	n23	3	n32	4	n41	6	n50	6
n6	5	n15	5	n24	3	n33	4	n42	6	n51	6
n7	4	n16	5	n25	3	n34	4	n43	6	n52	6
n8	6	n17	5	n26	3	n35	4	n44	6		

Figure 14: Node weight settings dialog



Link Weights

l0	Bandwidth:	1000000	Delay:	0	BER (1/N):	0
l1	Bandwidth:	100000000	Delay:	10	BER (1/N):	0
l2	Bandwidth:	10000	Delay:	0	BER (1/N):	0
l3	Bandwidth:	100000000	Delay:	0	BER (1/N):	1000
l4	Bandwidth:	99999000	Delay:	0	BER (1/N):	0
l5	Bandwidth:	100000000	Delay:	0	BER (1/N):	0
l6	Bandwidth:	99998000	Delay:	0	BER (1/N):	2000
l7	Bandwidth:	100000000	Delay:	0	BER (1/N):	0
l8	Bandwidth:	1000000	Delay:	10	BER (1/N):	0
l9	Bandwidth:	100000000	Delay:	0	BER (1/N):	0
l10	Bandwidth:	99994000	Delay:	0	BER (1/N):	1000
l11	Bandwidth:	100000000	Delay:	5	BER (1/N):	0

Figure 15: Link weight settings dialog

6.4 UML Diagram

The following UML diagrams illustrate the multilevel partitioning scheme, as has it been implemented in IMUNES. The diagrams illustrate general steps involved in graph partitioning.

The first diagram (Figure 16) shows the first two phases of the multilevel scheme; coarsening and partitioning. Before the algorithm can start, the graph and all of its preferences are read, whereas all the useful information for partitioning is sorted out and saved. After that, the algorithm starts recursive by partitioning the graph. The first phase of the multilevel scheme is coarsening ("coarsen graph"). In this phase the graph is reduced in each iteration, until the desired size is reached ("Coarsen enough?" yields "Yes"). After coarsening, the smallest (coarsest) graph is sent to the partitioning phase ("init partitioning"). In the partitioning phase the graph is partitioned, new partitions are compared with the current best partitions, and if they are better then the current best, new partitions are saved as the best partitions. New partitions are created and compared with the best one, until "Enough partitions tried" yields "Yes". The coarsest graph with the best saved partitions is sent to the refinement.

The second diagram (Figure 17) shows the last phase of the multilevel scheme; uncoarsening and refining. The first step of refining is "balance", where it is verified whether the graph is partitioned in approximately equal partitions, or not (that is, the partitions are "unbalanced"). If the partitions are unbalanced, one node from the bigger partition is chosen and switched to the other, smaller partition. The nodes are moved from one partition to the other until the partitions become balanced, or enough swapping has been tried ("Are partitions unbalanced?" yields "No", or "Enough swapping tried?" yields "Yes"). Balanced partitions are then refined ("refine"). With refining the partition, the algorithm tries to reduce the edge-cut. If refining is needed, the node from the bigger partition, which brings the biggest decrease in edge-cut, is switched to the smaller partition ("swap node's partition"). The partitions are refined until there can't be any more node found to switch the partition, or enough swapping has been tried ("Is there any node to swap the partition?" yields "No", or "Enough swapping tried?" yields "Yes"). In the "Uncoarse", the refined partitions are projected from the coarser graph to the coarse graph one level higher. The partitions are uncoarsed until the original graph is reached ("Reached the original graph?" yields "Yes"). If the graph is to be partitioned in more then 2 partitions it is split in 2 subgraphs ("Split graph"), which are then again partitioned separately.

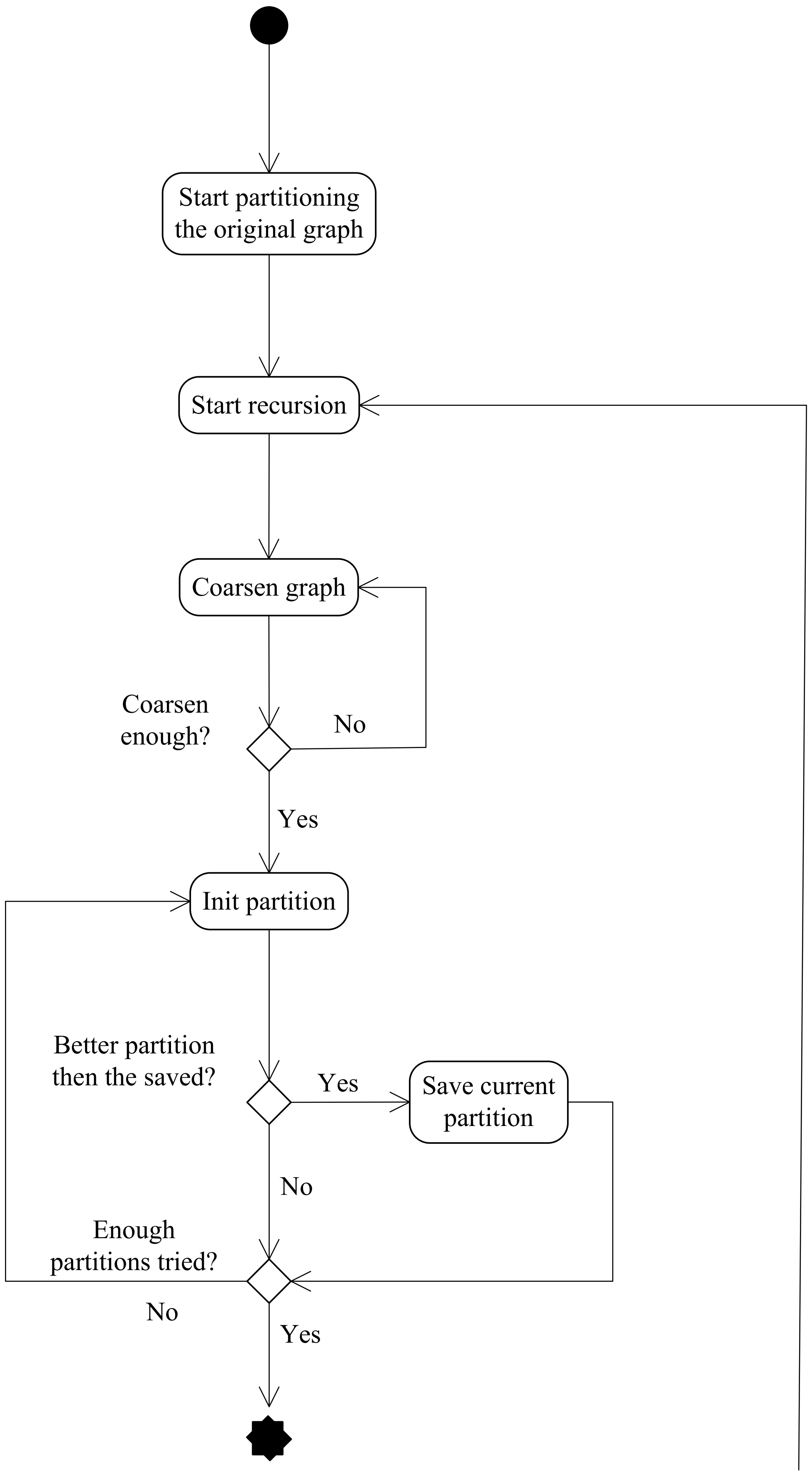


Figure 16: UML diagram for the first part of the multilevel scheme

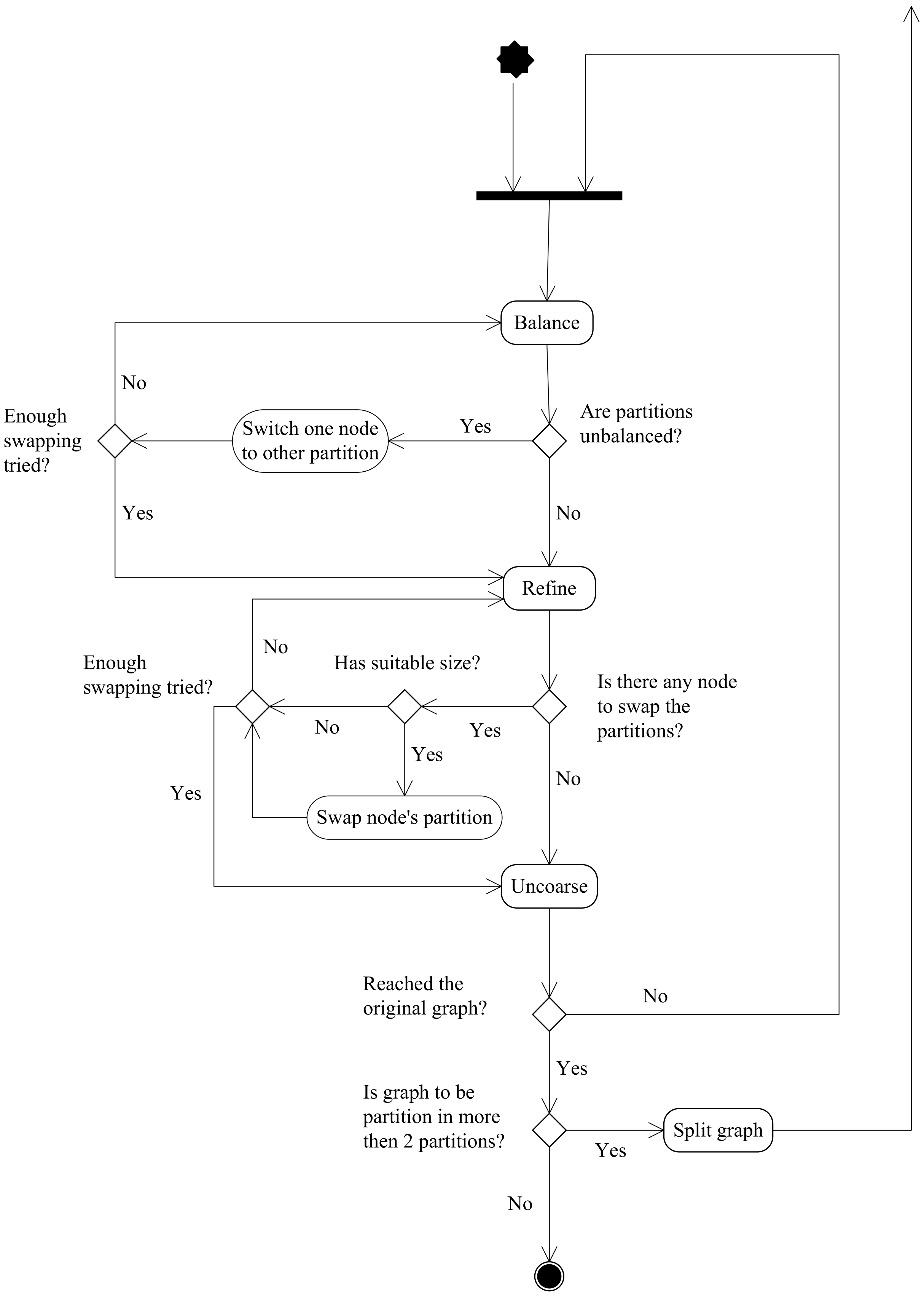

Balance
No
Enough swapping tried?
Switch one node to other partition
Yes
Are partitions unbalanced?
Yes
No
Refine
No
Enough swapping tried?
Has suitable size?
Is there any node to swap the partitions?
No
Yes
Yes
Yes
No
Swap node's partition
Uncoarse
Reached the original graph?
No
Yes
Is graph to be partition in more then 2 partitions?
Split graph
Yes
No

Figure 17: UML diagram for the uncoarsening and refining phases

6.5 Graph data structure

In IMUNES, a graph is represented via two lists: *node_list* and *link_list*. *node_list* contains information about each node like node id, node weight, node type, and etc., while *link_list* contains information about each link, such as link id, incident nodes to that link, link bandwidth, and etc. Nodes represent computational tasks, and edges represent data exchanges.

For each node $v \in V$, all interesting information for the graph partitioning algorithm, from those two lists, is saved in two arrays:

a) *node_weight* – the weight of v

b) *node_neighbour* – list of to v adjacent nodes

Interesting information about the edges in the graph is also saved in two arrays:

a) *edge_array* – the array of all edges in the graph, and their incident nodes

b) *edge_weight* – the weight of each $e \in E$

There is no need to hold all the nodes in an array in order to access them. Instead, the algorithm just needs to know the total number of nodes in the graph (*nvertices*). It uses the indexes, starting with index *0* and ending with index (*nvertices – 1*), to unambiguously address each node's information in each array.

The variable *nparts* specifies in how many partitions the graph is to be divided into. Its size is reduced by half in each recursion of the algorithm. The desired size of each partition is kept in an array *tpwgts*.

The original graph is regarded as a weighted graph, with a unit weight assigned to each edge and each node. Each node and each edge in IMUNES has its characteristics, which helps to better partition the work. For the graph partitioning algorithm the *node_weight* and *link_weight* can represent the load, the capacity, the network flow, or any other characteristic.

Currently, there are five types of nodes in IMUNES:

- Router
- Host
- PC
- RJ45
- Hub
- Switch

Weights can be assigned to the whole groups of nodes or to each node separately. At the time of writing this thesis, the weight of nodes is not yet calculated regarding its characteristics, but left to the users to set it according to their needs.

The weight of the link is calculated from the link's bandwidth and delay. Weights can only be changed for each link separately. To change the weight of a link, there are two possibilities: it can be changed by changing the bandwidth and/or delay in link's preferences (click once on the link) or through "Graph partition" window. If it is changed through the "Graph partition" window, changed characteristics also change in the whole program.

6.6 Non-sequential input data

For the graph partitioning algorithm it is necessary that the arrays that it uses always have sequentially ordered data. However, it is often the case that the graph that is to be partitioned has non-sequential nodes and/or links. A list with the non-sequential nodes and/or links is at the beginning of partitioning transformed into sequential arrays.

Figure 18 (a) illustrates a sample graph with non-sequentially ordered node ids and link ids, (b) illustrates this sample graph after it has been transformed into a graph with sequential ordered node ids and link ids. The ids in (b) are ordered in the same ascending sequence as the real ids in (a); in this example, the mapping for nodes is {n0->0, n3->1, n4->2, n7->3, n13->4, n14->5}, and for links {l0->0, l1->1, l3->2, l4->3, l7->4}.

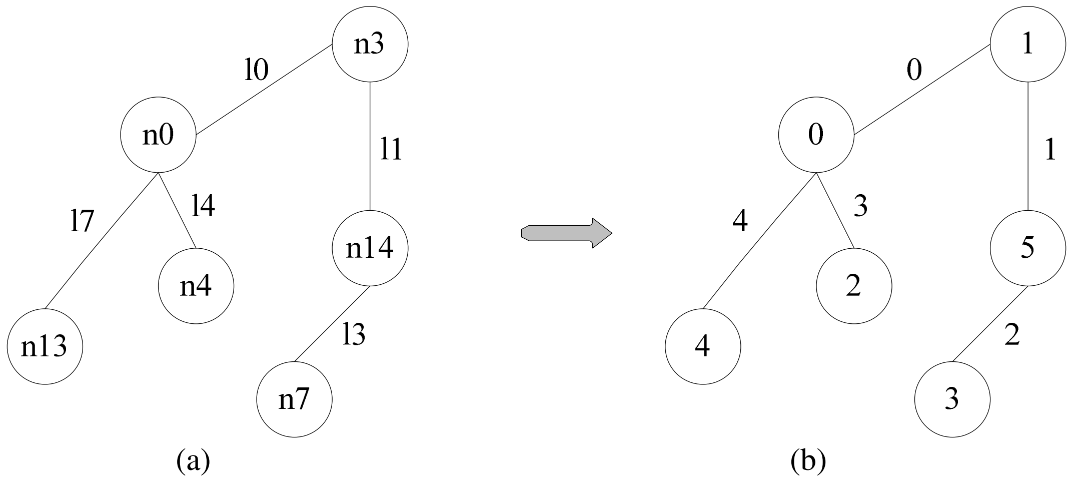


Figure 18: Example of transforming a non-sequential data into sequential data

6.7 Disconnected graphs

A graph is disconnected when it contains two or more components which are not connected by any edge (see Figure 19). The partitioning algorithm recognizes and can handle this type of graph.

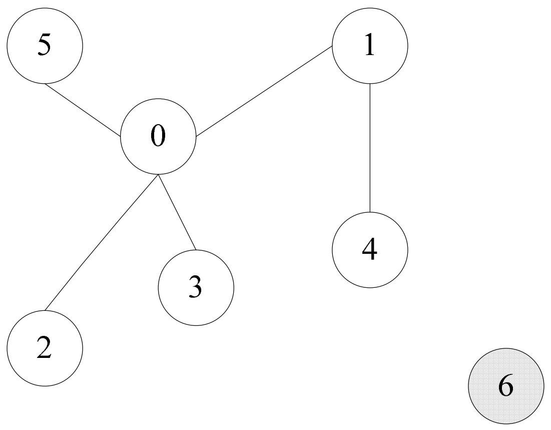


Figure 19: Example of a disconnected graph

7 The Algorithm

7.1 Coarsening

The goal of the coarsening phase is to create a sequence of smaller graphs, each with fewer vertices.

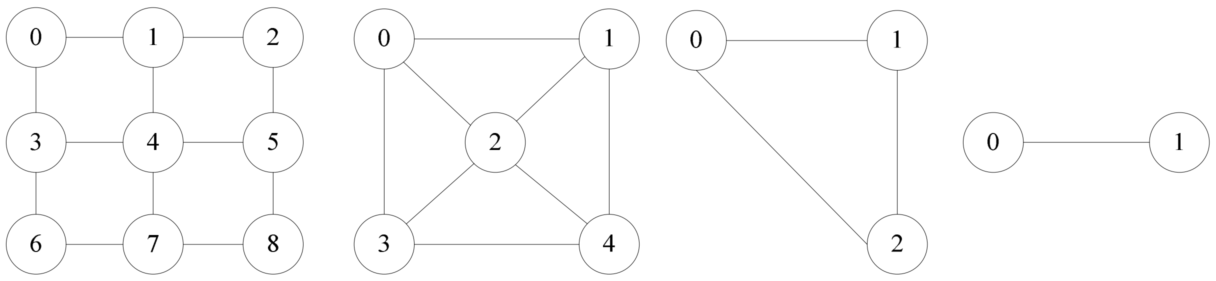


Figure 20: A sequence of coarsening

Consider a graph $G_0 = (V_0, E_0)$, with a sequence of smaller graphs $G_1, G_2, \ldots, G_k$, such that $|V_{i-1}| > |V_i|$. The smaller graph G_i is constructed from the finer graph G_{i-1} by finding a maximal matching M_{i-1}, and collapsing as many pairs of adjacent vertices of G_{i-1} as possible. Matching is defined as a set of edges, where no two edges share the same vertex. Maximal matching of a graph is the matching that contains the maximal possible number of vertices, to which no more vertices can be added while remaining matching. This is the most time consuming phase of the three (coarsening, partitioning, uncoarsening and refining).

There are two processes, through which the vertices in the coarsening phase traverse: matching and contraction (mapping and creating).

The first process is matching. In matching, the algorithm finds two adjacent vertices *v* and *u* to collapse them together. The collapsed pair (*v*, *u*) is called a *match*. Each vertex can be a match only once at each coarsening level. Pairs of vertices are matched until no more unmatched vertices remain. If a vertex *v* is unmatched, but all of its neighbours are matched, it is then matched with itself (*v*, *v*). The matching algorithm is sketched in Figure 21.

```
For Each matching edge (v,u)
    Contract edge to form new vertex n
    Vertex weight(n) := weight(u) + weight(v)
        If u and v are both adjacent to a vertex k Then
            Edge weight(n,k) := weight(u,k) + weight(v,k)
        End if
End for
```

Figure 21: Constructing a coarse approximation to a graph

There are a few ways that can be used to select matching for coarsening. They determine the order in which the vertices are added to the matching. Some of them are described here. In the graph partitioning algorithm for IMUNES implemented method, is a *heavy – edge matching* (HEM).

```
RM ()
    Unmark all vertices and set $M = Ø$
    While there are unmarked vertices left
        Randomly select unmarked vertex v
        Mark v
        If v has unmarked neighbours Then
            Randomly choose unmarked neighbour u of v          (*)
            Mark u
            Add edge $e = (v, u)$ to M
        End if
    End while
End
```

Figure 22: Random matching (RM)

- Random matching (RM)

Random matching (RM) is a method that uses a randomized algorithm to match the vertices. The vertices are visited in random order. If one visited vertex v is not already matched, then one of its unmatched adjacent vertices is randomly selected. If such a vertex u exists, the pair (v, u) is matched together. The complexity of the random matching is in worst-case $O(|E|)$ [3]. A sketch of RM algorithm is shown in Figure 22.

- Heavy - edge matching (HEM)

Heavy-edge matching (HEM) is a method that matches a vertex with the adjacent vertex with the heaviest edge weight. HEM matching, just like *random matching*, also uses a randomized algorithm, where the vertices are visited in random order. However, if some vertex v is not already matched, then instead of randomly selecting one of its unmatched adjacent vertices (line (*) of Figure 22), an unmatched adjacent vertex u with the heaviest edge weight among the neighbours of v is selected for the match (v, u). HEM reduces the weight of the edge-cut:

Consider a graph $G_i = (V_i, E_i)$ as a coarse and $G_{i+1} = (V_{i+1}, E_{i+1})$ as the coarsest graph of the original graph $G = (V, E)$. M_i is the matching used to coarsen G_i to G_{i+1}. Let $W(A)$ be the sum of the edge weights in A, where A is the set of edges, e.g. $W(E_i)$ is the sum of the edge weights of all the edges in E_i. After [3] HEM matching produces better results than RM:

$$W(E_{i+1}) = W(E_i) - W(M_i) \tag{1}$$

As it can be seen from the Equation 1, the sum of edge weights of the coarser graph is reduced by the weight of the matching. By matching the heaviest edges, the edge weight of the coarser graph is decreased by a greater amount. Since the coarser graph has smaller edge weight, it also has a smaller edge-cut.

- Light Edge Matching (LEM)

In some problems, the total edge weight of the coarser graph needs to be maximized instead of minimized (like in HEM). LEM algorithm is very similar to HEM algorithm. To compute a matching with minimal edge weight, instead of selecting to v an unmatched adjacent vertex u

with the largest edge weight among the neighbours, the algorithm selects an unmatched adjacent vertex *l* with the smallest edge weight among the neighbours (line (*) of Figure 22).

- Heavy Clique Matching (HCM)

In an unweighted graph $G = (V, E)$, a fully connected subgraph $G_c = (V_c, E_c)$ of *G* is called a *clique*. The HCM algorithm matches the vertices, whose clique has high edge density. The edge density is calculated from:

$$\frac{2|E_c|}{|V_c|(|V_c|-1)} \tag{2}$$

An example of the matching is shown in Figure 23.

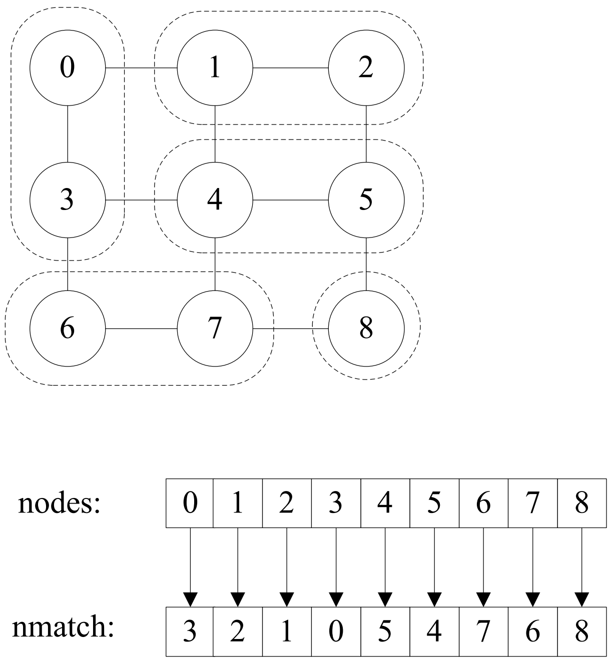


Figure 23: An example of matching

In the example shown in Figure 23, matching of a sample graph is given. Two arrays in the figure show the resulting matching between the nodes; node 0 is matched with node 3, node1 with node 2, node 8 with itself, and so on.

The second process is mapping. For every matched pair (*v*, *u*) of vertices in G_{i-1} a new vertex *n* is created to represent the pair in the next level graph G_i. In each iteration of creating a smaller graph, new vertices are labeled consecutive in range [*0*, *number of new vertices*]. This label uniquely identifies the vertex in the coarse graph to its parents in the finer graph. Information about mapping is saved in the array *nmap*. For each pair of matched vertices (v, u), it is true that nmap[v] = nmap[u]. In order to preserve the characteristics of the original graph, the weight of the new vertex n is set equal to the sum of the weights of the parent pair of vertices (v, u). The vertex n inherits all the neighbours of its parent vertices:

$$neighbor(n) = [nmap(x) \cup nmap(y)] \setminus [nmap(x) \cap nmap(y)], \;\; x \in neighbor(v), y \in neighbor(u)$$

All edges leading from the merged vertices are left unchanged, except for an edge connecting the merged vertices. This edge doesn't exist in the next iteration. Edge weights are also left unchanged unless both parent vertices are adjacent to the same neighbour. In that case, the new edge that represents the two original edges is given the weight of the sum of the weights of both original edges.

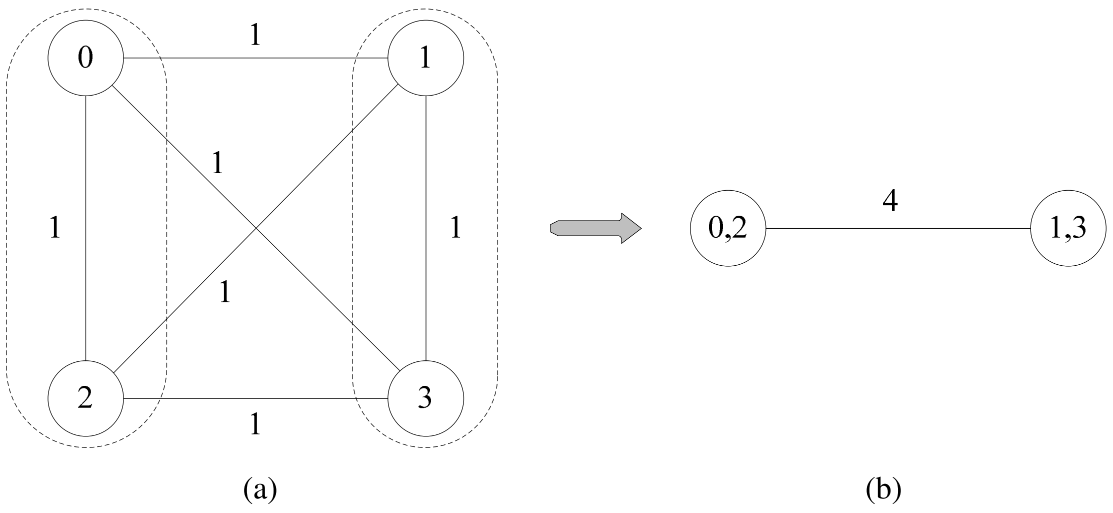


Figure 24: Example of updating the edge weights in coarsed graph

In Figure 24, the four vertices are coarsed, and two new vertices are created. The four edges connecting pairs of vertices in (a) are in the new, coarsed graph (b), combined into one edge with the weight of the sum of all four edges weight. Two edges between collapsed nodes (edge between node 0 and 2, and node 1 and 3) are removed.

An example of the mapping is shown in Figure 25.

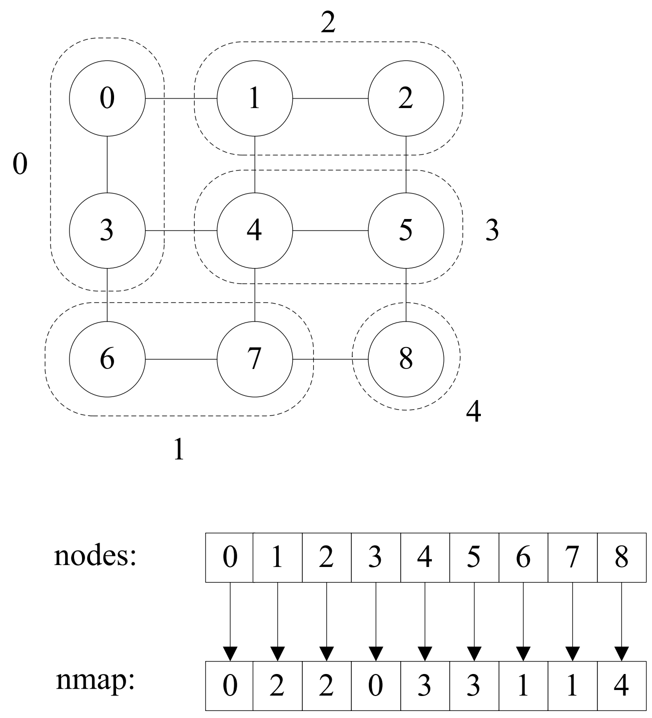


Figure 25: An example of mapping

In the example shown in Figure 25, the mapping of a sample graph is given. The graph in the figure is the coarsed graph from the example in Figure 23. For each pair of the coarsed vertices in the graph, a new vertex is created. Two arrays in the figure show the mapping from each coarsed vertex to its *mapped* vertex; nodes 0 and 3 are mapped to node 0, nodes 1 and 2 to the node 2, and so on. The number of vertices in the finer graph (nvertices) is reduced from 9 to 5 (cnvertices) in the coarsed graph.

The coarsening phase stops when the number of vertices in the coarsest graph is smaller than the threshold, or when the graph can't be coarsened any further. At its simplest, the contraction can be terminated when the number of vertices in the coarsest graph is the same as the number of subsets required. However, during the collapsing of vertices together, some new vertices may become much heavier than the others, so that collapsing the vertices until there is the same number of vertices as the number of subsets, could result in partitions with very different vertices' weights. Coarsening can in each recursion reduce the graph size by half.

7.2 Initial partitioning

Initial partitioning is the second phase of the multilevel graph partitioning where the initial partition for each vertex is computed. The initial partition is computed by recursively bisecting the coarse graph $G_c = (V_c, E_c)$ into two subgraphs $G_{c1} = (V_{c1}, E_{c1})$ and $G_{c2} = (V_{c2}, E_{c2})$, where $G_{c1} \cap G_{c2} = \varnothing$, $G_{c1} \cup G_{c2} = G_c$.

In the each run of the initial partitioning algorithm, two partitions are created. Those partitions are always seen as partitions 0 and 1. To compute the real partition of each node, at the end of each recursion of the partitioning, the number of already made partitions is added to each partition. The example in Figure 26 shows how real partitions for a sample graph are computed.

The Figure 26: An illustration of the multilevel recursive bisection algorithm illustrates on a sample graph how the multilevel recursive bisection algorithm works. The graph is to be partitioned in 7 partitions. (a) The original graph is in each recursion divided into 2 halves, first half gets the partition 0 and the second the partition 1. Graph division stops when the number of end nodes equals to the desired number of partitions, in this example when the number of end nodes is 7. (b) To calculate the real partitions; zero is added to the partitions 0 and 1 in the first subgraph, because there are no partitions before them; two is added to the partitions 0 and 1 in the second subgraph, because two partitions are already created; four is added to the partition 0 in the third subgraph, because there are already four partitions created, and so on. (c) Shows the real partitions.

There are a few different algorithms for partitioning the coarse graph. Some of them are described here.

- **Simple partitions methods**

Linear

In the Linear scheme, the location of the vertices in the original graph is used to determine the partitions. For an unweighted graph with *n* vertices and divided into *p* partitions, the first $\frac{n}{p}$ vertices of the graph are assigned to partition 0, the second $\frac{n}{p}$ to partition 1, and so on.

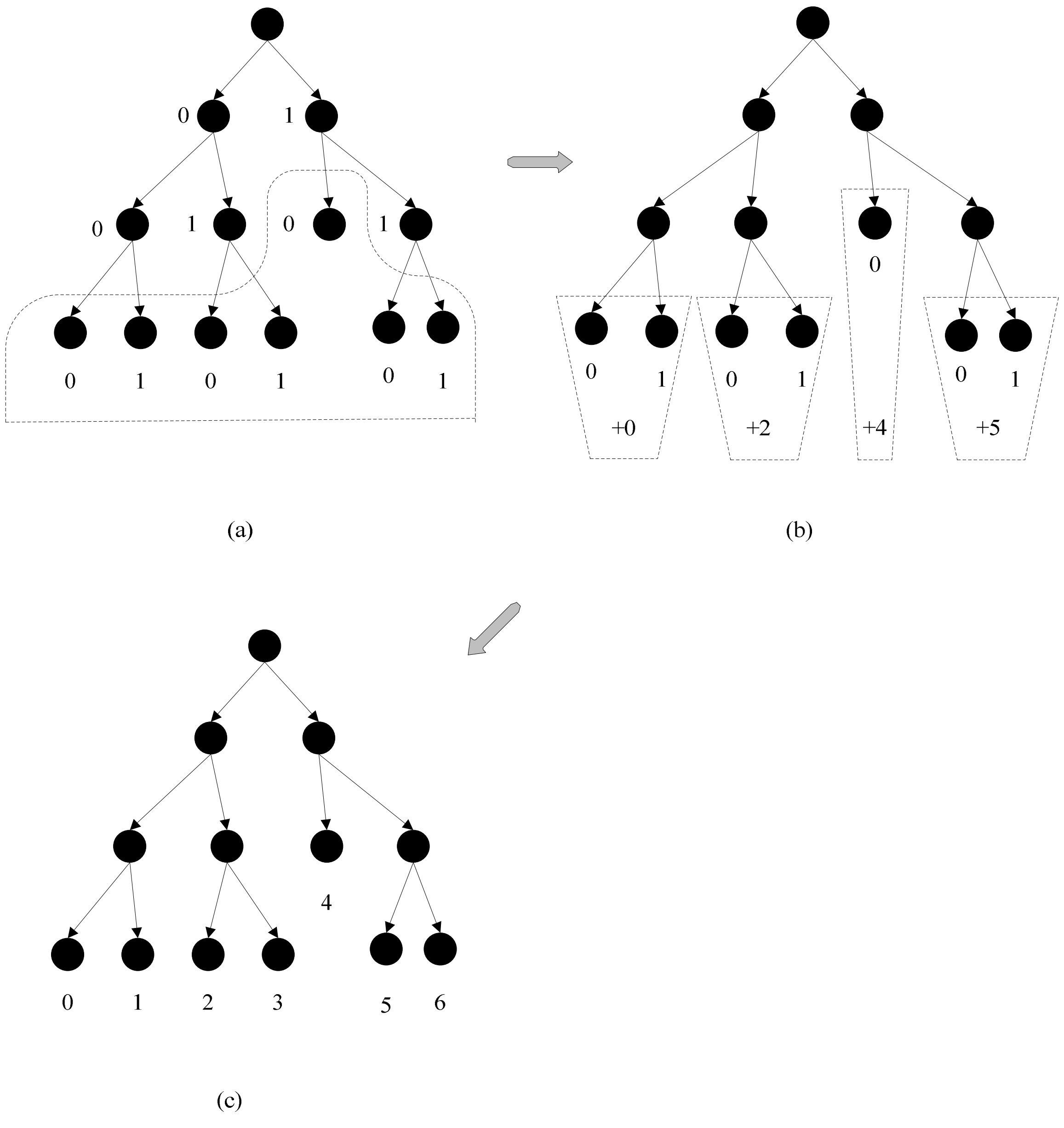


Figure 26: An illustration of the multilevel recursive bisection algorithm

Linear algorithm is extremely fast, on the order of $O(n)$. However, the edge-cut of its partitions is generally high.

Random

In the Random scheme, each vertex is randomly assigned to a partition which has less than $\frac{n}{p}$ vertices, in a way that preserves balance. Partitioning using Random scheme is very fast, on the order $O(n)$, but the edge-cut of its partitions is, like in Linear scheme, generally high.

Scattered

The Scattered scheme, like Linear scheme, also uses the location of the vertices in the original graph to determinate the partitions. Vertices are assigned to the partitions in an *n mod p* fashion, where *n* is the position of the vertex, and *p* the number of partitions. For example, the vertex at the position 0 is assigned to the partition 0, the vertex at the position 1, to the partition 1, etc. Scattered algorithm is also very fast, on the order $O(n)$, and produces partitions with generally high edge-cut.

- **Inertial partitions methods**

The Inertial method is a relatively simple and fairly quick graph heuristic. It uses geometric information to partition the graph. The user supplies geometric coordinates for each vertex in one, two or three dimensions. The algorithm first computes the principle axis of the graph, which is generally the direction in which the graph is elongated. The vertices are then projected onto the principle axis. The projection sorts the vertices according to their coordinate values for this axis. The vertices are then divided by plane(s) orthogonal into partitions of approximately equal sizes.

- **Spectral partitions methods**

Spectral methods [3][28][28] use eigenvectors of a matrix constructed from the graph to decide how to partition it. This method computes the eigenvector *y* corresponding to the second eigenvalue of the Laplacian matrix which is also called the Fiedler vector.

The Laplacian matrix $Q = D - A$, where A the adjacency matrix such that

$$a = a_{i,j} = \begin{cases} \text{edge\_weight}(v_i, v_j) & \text{if } (v_i, v_j) \in E_c \\ 0 & \text{otherwise} \end{cases}$$

Given *y*, the set of vertices *V* is partitioned into two parts. Let *r* be the ith element of the *y* vector, and the median of the values of y_i. *P[j] = 1* for all the vertices such that $y_j <= r$, and *P[j] = 2* for all the vertices such that $y_j > r$.
The eigenvector *y* can be computed using the Lanczos [30][31] or the multilevel method combining Rayleigh Quorient Iteration [32][33] and the linear solver Symmlq [34].

- **Multilevel-KL**

Multilevel-KL algorithm described in [3] uses "multilevel" heuristic which consists of three phases: coarsening, partitioning and uncoarsening (see Section 7). Kernighan-Lin algorithm is invoked at every level of uncoarsening to refine the partition.

- **Kernighan-Lin Algorithm (KL)**

Kernighan-Lin algorithm is a greedy, local optimization strategy. It starts with an initial bipartition of a graph, and searches for a subset of vertices in the graph, such that swapping them leads to a reduction of the number of edge-cuts.

Kernighan-Lin algorithm finds locally optimal partitions if it is given a good initial partition [28][3]. KL algorithm is described in more detail in Section 7.3.

- **Graph Growing Algorithm (GGP)**

The algorithm implemented in the IMUNES system is the g*raph growing* algorithm. To bisect the graph, the *graph growing* algorithm randomly chooses one "starting" node, and then uses Breadth - first search algorithm - to search the graph for other nodes.

At the beginning of the partitioning process, the algorithm assigns each node to the first partition (P_0). Then it randomly chooses the "starting" node, marks it, and puts it in the empty queue. Until the queue is not empty, the algorithm takes the first node from the queue, changes the partition of the node to the second partition (P_1), if its size suits, and searches for the node's unmarked neighbours. If such a neighbour exists, it marks it and puts it in the queue (FIFO queue). The search algorithm ends when the size of found nodes fills up the size of the second partition (P_1). The quality of this algorithm depends on the coarse phase and of the choice of the "starting" node. Different "starting" nodes produce different edge - cuts.

```
BFS(G)
    For each vertex u in V[G]
        partition[u] := $P_0$
        visited[u] := 0
    End for
    s := random node from V[G]
    visited[s] := 1
    ENQUEUE(Q, s)
    While (Q != Ø)
        u := DEQUEUE(Q)
        If (u.size not too big)
            partition[u] := $P_1$
        End if
        For each vertex v in Adj[u]
            If (visited[v] = 0)
                visited[v] := 1
                ENQUEUE(Q, v)
            End if
        End for
    End while
End;
```

Algorithm 1: Breadth – first algorithm (BFS)

If the algorithm comes to an empty queue, and the second partition (P_1) is not filled up either, it means that this graph is disconnected. In such case, the algorithm tries to find a new "starting" node, by iterating through the list of nodes, and searching for the first unmarked node. If the node is found, the algorithm continues, otherwise it ends.

The important fact about the Breadth - first algorithm is that all vertices at the same distance from the "starting" node are searched. After all the vertices on the same distance are exhausted, the distance is increased by one, and the vertices at the next distance are searched further. This feature of BFS algorithm allows better partitioning, by swapping a group of adjacent vertices to the same partition.

Figure 27 and Figure 28 illustrate the progress of Breadth - first search algorithm on a sample graph. As the vertices are explored by the algorithm, they are shown grey. Used paths between two distances are shown with full line:

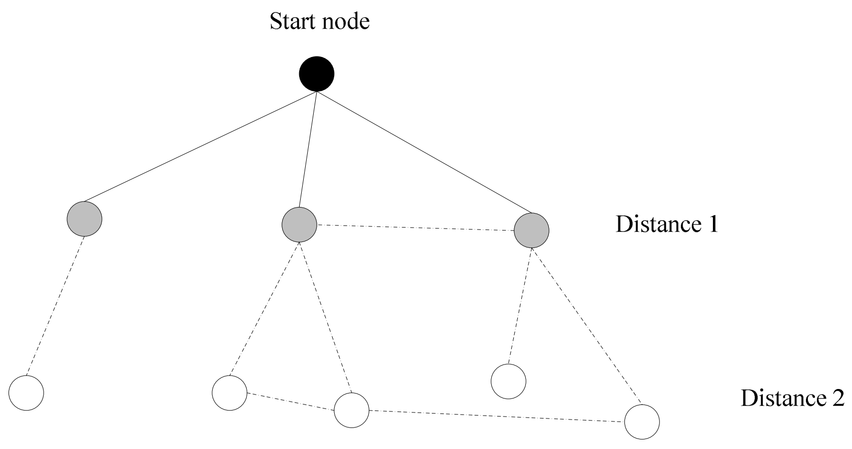


Figure 27: BFS search at the distance 1

BFS starts at a given vertex (*Start node*), which is at distance 0 (see Figure 27). In the first stage, all vertices at the distance 1 are searched.

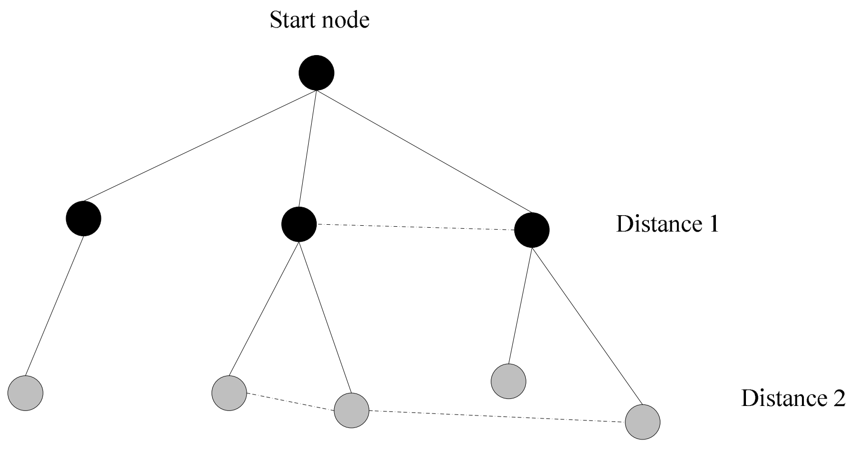


Figure 28: BFS search at the distance 2

When all the vertices at the distance 1 are exhausted, the vertices at the distance 2 are explored while the vertices at the distance 1 became black (see Figure 28). BFS traversal terminates when every vertex has been visited.

Analysis
In each iteration of Breadth - first algorithm, each node is enqueued and dequeued once at the most. In the worst case, when all the vertices of coarser graph $G_c\,(V_c)$ are enqueued and dequeued, the algorithm takes $O(|V_c|)$. Since the size of the coarser graph G_c is small ($|V_c| <$ 50), it takes only a few steps for the initial partition of the coarse graph G_c.

- **Greedy Graph Growing Algorithm (GGGP)**

Greedy graph growing algorithm makes, similar to Kernighan-Lin algorithm, use of the gain (see next section). Instead of growing the partition in a strict breadth-first fashion, GGGP obtains for each vertex *v* its gain in the edge-cut by inserting *v* into the growing region. The vertices with the smallest gain are inserted first. When a vertex is inserted into growing partition, to its adjacent vertices, which are not yet inserted, the gain is updated (if exists) or created. Thus, at each step, the vertex with the smallest increase (or largest decrease) in the edge-cut is added to the partition.

The quality of GGGP also depends on the choice of the initial vertex, but less then GGP.

7.3 Refining the initial partition

The initial partitions P_0 and P_1 computed for the coarse graph G_c are further refined. The refinement is based on the swapping of vertices between the two partitions to reduce the edge-cut.

The choice of swapping is based on Fiduccia and Mattheyses (FM) [11] improvement of the Kernighan - Lin algorithm [6]. The idea behind Kernighan - Lin algorithm is the concept of the *gain*. The gain g_v of a vertex *v* shows how big there a reduction in edge-cut could be, if the vertex *v* is moved to the other partition. More formally, if a vertex *v* is moved between two partitions, the corresponding gain g_v can be expressed as:

$$g_v = \sum_{P[v] \neq P[u]} w(v,u) - \sum_{P[v]=P[u]} w(v,u), \quad (v,u) \in E \tag{3}$$

```
Until No better partition is discovered
    Best Partition := Current Partition
    Compute all initial gains
    Until Termination criteria reached
        Select vertex to move
        Perform move
        Update gains of all neighbours of moved vertex
        If Current Partition balanced and better then Best Partition Then
            Best Partition := Current Partition
    End Until
    Current Partition := Best Partition
End Until
```

Figure 29: An algorithm for refining graph partitions [1]

where *P[v]* is a partition of vertex *v*, *P[u]* is a partition of vertex *u*, and $w(v,u)$ is the weight of the edge connecting vertices *v* and *u*. If g_v is positive, than swapping the vertex *v* will reduce the cost of the edge-cut; while if g_v is negative, than swapping the vertex *v* will increase the cost of the edge-cut; and if g_v equals zero, there will not be any reduction or incensement of the cost of the edge-cut.

In the program, the gain value is computed using two arrays, ID and ED, for each vertex *v:*

$$ID[v] = \sum_{P[v]=P[u]} w(v,u) \tag{4}$$

$$ED[v] = \sum_{P[v] \neq P[u]} w(v,u) \tag{5}$$

where *u* is a neighbour of *v*, *w(v, u)* is the edge-weight of the edge between *u* and *v.* ID stands for *internal degree*, and holds the sum of the edge-weights of the edges incident to the vertices in the same partition. ED stands for *external degree*, and holds the sum of the edge-weights of the edges incident to the vertices in different partitions. The gain is then given by:

$$g_v = ED[v] - ID[v] \tag{6}$$

Edge - cut is calculated from the ED array. It holds the total size of the edge weight of all the edges connecting two vertices in different partitions:

$$cut = 0.5 * \sum_{v \in V} ED[V] \tag{7}$$

The Fiduccia and Mattheyses (FM) algorithm starts with the initial partitions of the coarse graph. In each iteration it chooses a set of vertices whose swapping reduces the edge-cut. If such a set exist, the vertices in that set are swapped. The swapping repeats until no such set of vertices is found, which means that the partition is at a local minimum and no more refinement is possible.

In the implementation of FM in a program a few trials are performed to reduce the edge-cut of coarse graph G_c. All the vertices, which are connected to one or more vertices from the other partition ($ED[v] > 0$), are saved in a list *boundary*. At the beginning of each iteration the array *partitions* is initialized to the initial partition and the new swapping is tried. Vertices from the *boundary* list are put in two priority queues, one for the vertices from the first partition (P_0) and the other for the vertices from the second partition (P_1). Each vertex's priority is defined by its gain. The highest the gain, the highest is the vertex's priority. Set of vertices for swapping is chosen from the partition whose size is bigger. This step is needed to keep sets from becoming unbalanced. A vertex with the highest gain from the bigger partition is then moved to the smaller partition. Each vertex can only be moved once during each trial of the algorithm. The highest gain can also be negative, in which case the edge-cut is not reduced but increased. This is needed so that algorithm doesn't get stuck in local minima. Once a vertex is moved, the gain values of all its neighbours are updated.

The neighbours who where until that moment in the same partition as the moved vertex, are now in the other partition and the neighbours who were in the other partition are now in the same partition as the vertex:

For each neighbour *n*, now in the same partition as *v*:

$$ED[n] = ED[n] - w(v,n) \tag{8}$$

$$ID[n] = ID[n] + w(v,n) \tag{9}$$

For each neighbour *n*, now in the different partition than *v*:

$$ED[n] = ED[n] + w(v,n) \tag{10}$$

$$ID[n] = ID[n] - w(v.n) \tag{11}$$

At the end of each iteration the new edge-cut is compared with the current best solution. And if it is better than the saved one, it is saved as the best solution with the partition array.

Analysis

The algorithm consists of two nested loops. The outer loop tries a few times to refine the partitions. In each new iteration it initializes the variables to "starting" values. The cost of the initialization is $O(n)$, n is the number of vertices. In the inner loop each vertex is visited only once, and the gain of each vertex is computed, the cost of this being $O(n+m)$, where m is the number of edges. Reversion through all vertices and its neighbours costs $O(m^2)$.

7.4 Uncoarsening

In the uncoarsening phase, the goal is to project the partition from the coarser graph back to the original graph. In each iteration the partition of the coarse graph (starting with the coarsest graph up) is projected to a one level higher graph. Iteration stops when the original graph is reached.

The partition of the coarser graph G_i is projected all the way up through the coarse graphs $G_{i-1}, G_{i-2}, \ldots, G_0$, in order to reach the original graph G_0. Each vertex v of G_n contains a distinct subset of vertices of G_{n-1}. To project the partition of the graph G_n onto the graph

G_{n-1}, the partition of the vertex *v* is assigned to the *v* corresponding set of vertices of G_{n-1}; the vertices which are contained in the vertex *v* get the same partition as *v* has.

The projected partition may not be at a local minimum with respect to graph G_{n-1}, event though it is at the local minimum with respect to graph G_n. This is because graph G_{n-1} is finer than graph G_n. Finer information is then used to refine and improve the quality of the partitions in G_{n-1}. The refining scheme used to refine the partitions in uncoarsening phase, is FM refinement, used also in refining new partitions, described in Section 7.3.

In the implementation, the algorithm doesn't start with the coarsest graph G_n, but with a one level higher coarse graph G_{n-1}. Going through the vertices of G_{n-1}, to each vertex *v* from G_{n-1} the corresponding vertex *r* and *r*'s partition is found. This partition is than assigned to *v*. After all vertices from G_{n-1} have been assign one partition, ID and ED values are also computed and saved.

Analysis
The cost of the projection of the partition from the graph G_1 to the graph G_2 is *O*(*n*), where *n* is the number of vertices in the graph G_2. The arrays ID and ED are also passed on the graph G_2, with a cost of *O*(*n*m*), where *m* is the number of each vertex's $v \in G_2$ adjacent vertices.

7.5 Graph Splitting

The graph is split into two subgraphs after its vertices are divided into two partitions. The vertices in the first partition make the first subgraph and the nodes in the second partition the second subgraph. The edges connecting both subgraphs don't exist from this level of recursion downwards. Each subgraph contains about the half of the vertices' weight of the original graph. Figure 30 illustrates an example of graph splitting.

7.6 Preprocessing

Some preprocessing needs to be done at the initialization of the algorithm, and in each recursion, before the first step of graph partitioning algorithm, coarsening, begins.

Initialization

The processing at the initialization of the algorithm includes the transformation of IMUNES vertex and edge lists into for graph partitioning algorithm suitable arrays, calculation of weights and their saving into lists, and computation of the desired sizes of end partitions.

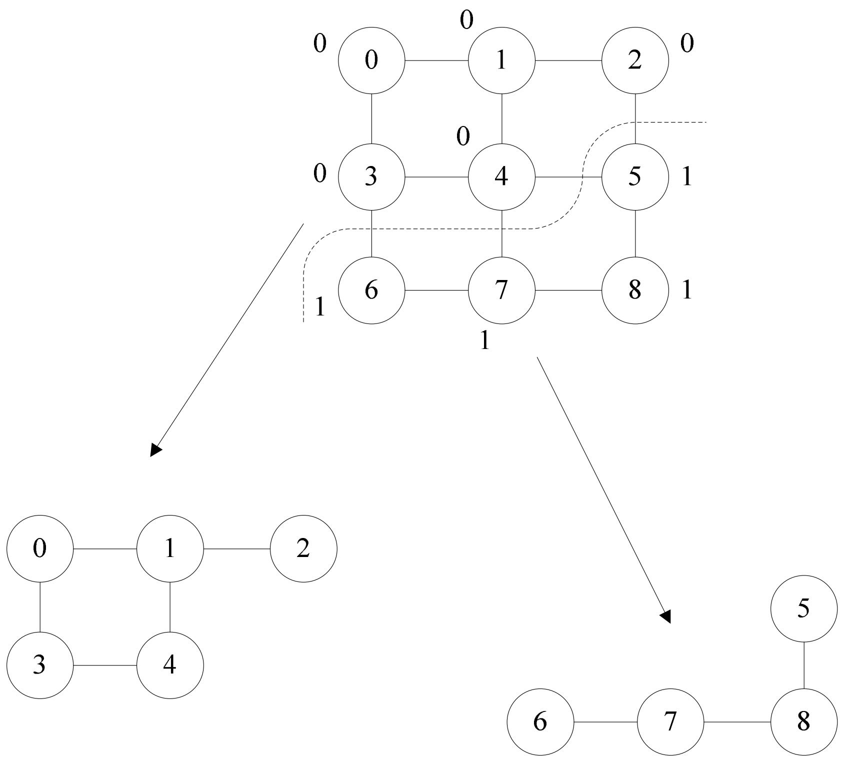


Figure 30: Sample graph is split in 2 parts

The transformation of IMUNES vertex and edge lists, as well as calculation of its weights was described in Section 6.5.

The number of partitions the graph is divided into (*nparts*) is obtained at the start of the partitioning by the user. At algorithm initialization, an array *tpwgts* is created. There are so many elements in an array as the number of parts in which the graph is going to be partition, each element holding the size of its ratio of the graph:

$$tpwgts(i) = \frac{1}{nparts}, \quad \text{where } i \in \{0, \ldots, nparts - 1\} \tag{12}$$

Recursion

- Calculating each partition's weight

To compute the desired size of each partition, the algorithm needs to know how many parts the graph is to be divided into, how many vertices are in the graph and how much do they weight. All those information are gathered at the initialization, and updated as the algorithm progress in its execution.

In each recursion of the algorithm, after splitting the graph into two halves, the weight of the next two partitions needs to be calculated. The calculation is done in a few steps:

1. Calculate the sum *S* of the weight of all vertices in the graph:

$$S = \sum_{i=0}^{nvertices-1} node\_weight(i) \tag{13}$$

2. Calculate the total ratio *T* of the first partition:

$$T = \sum_{i=0}^{\left\lfloor \frac{nparts}{2} \right\rfloor} tpwgts(i) \tag{14}$$

3. Calculate the size of the first partition (it is nearly half of the weight of the graph):

$$tpwgts2[0] = S * T \tag{15}$$

4. Calculate the size of the second partition (makes the rest of the weight of the graph, and is often heavier than the first partition):

$$tpwgts2[1] = S - tpwgts2[0] \tag{16}$$

- At the end of each recursion, depending on how many partitions the graph will additionally be divided into, the algorithm is either repeated or finished.

If the recursion is repeated, ex. *nparts > 2*, then the graph is divided into two separate parts (see Section 7.5), and for each separate part the values concerning the partition's size (*tpwgts* array) and number (*nparts*) are recalculated.

- Update the *tpwgts* array

1. The ratios for the first subgraph are recalculated by dividing the old ratios with the total ratio of the whole partition:

$$tpwgts[i] = \frac{tpwgts[i]}{T}, \quad i \in \left[0, \frac{nparts}{2}\right[\tag{17}$$

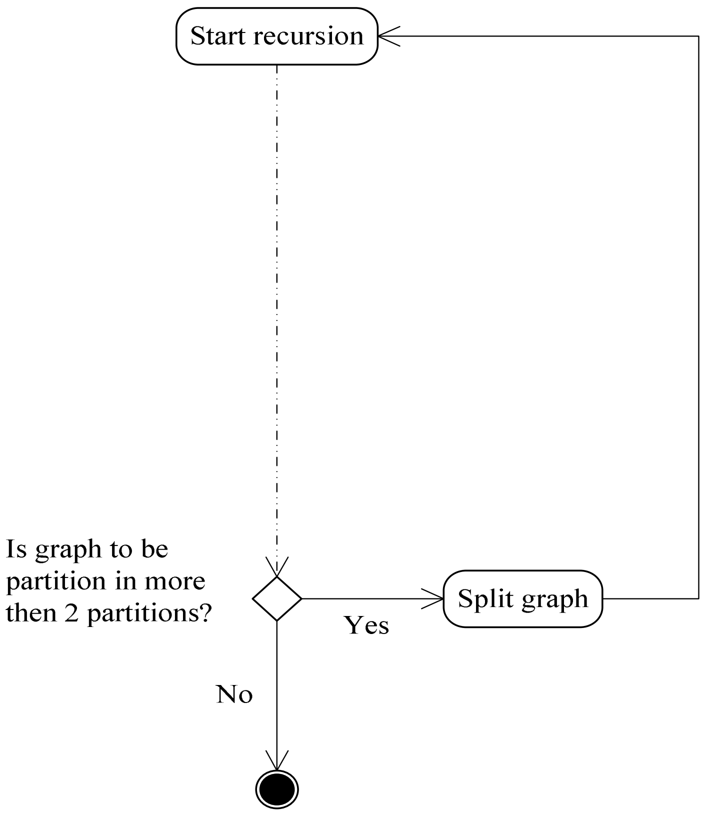


Figure 31: Recursion of the algorithm

2. And the new rations for the second subgraph are calculated by dividing the old ratio with the rest of the total ratio of the whole partition:

$$tpwgts[i] = \frac{tpwgts[i]}{1-T}, \quad i \in \left[\frac{nparts}{2}, nparts\right] \tag{18}$$

- Update the number of partitions

The number of parts to partition the graph is also recalculated. In each recursion the graph is partitioned in two new, additional partitions, therefore in each recursion the total number to partition the graph is reduced by 2. The number is also divided between two subgraphs, that way the first subgraph gets $\left\lfloor \frac{nparts}{2} \right\rfloor$ partitions, and the second subgraph the rest, $nparts - \left\lfloor \frac{nparts}{2} \right\rfloor$ partitions.

8 Experimental Results

In this section, the results from different experiments are presented. The first experiment focuses on the execution time of graphs for different network topologies. The second focuses on the quality of partitions, and the third one on the edge-cut.

All experiments for this thesis were performed on a personal computer with AMD Athlon64 and 512 MB RAM. The installed operating system is FreeBSD 4.11.

8.1 Timings for various network topologies

Graphs from various network topologies are used for testing the algorithm. All of those graphs are undirected and weighted. The network topologies used for testing are star, ring, full mesh and tree, the most used topologies for networks. For all the tests with network topologies, except for the tree test, the same weight for nodes and links is used in order to represent the results easier. In the tree test different node types with different node weights and link weights are used, to picture a real network.

The first three tests should show how the algorithm reacts in individual network topologies, and the last one should show how the algorithm with different interconnected topologies works.

8.2 Star

In star topology each node is directly connected to a central network hub, switch or router.

The algorithms in the coarsening phase (RM and HEM) are not well suited for star topology. This is because RM and HEM disregard the possibility of matching disconnected nodes, since those nodes are not neighbours. The coarsening phase in this case may produce a very unbalanced coarse graph. Moreover, RM and HEM match in each iteration only the central node with a node connected to it, which means that matching needs $O(|n-1|)$ steps to coarse the graph, where *n* is the number of vertices connected in star-like structure.

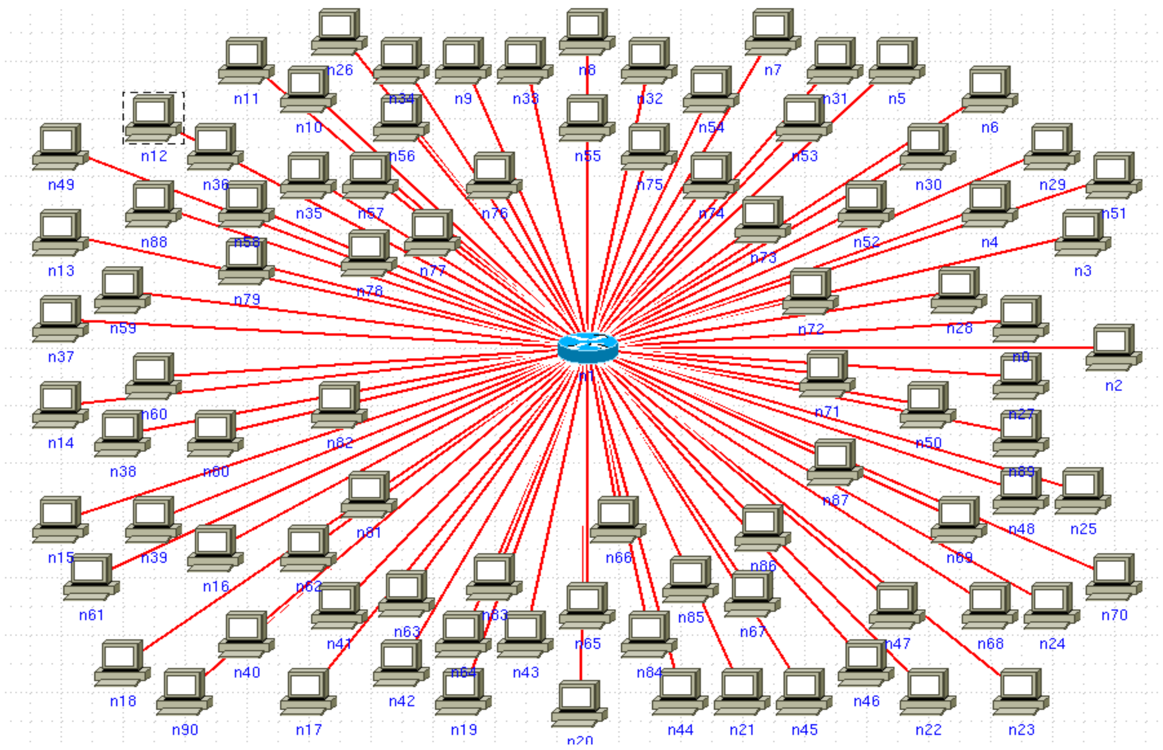


Figure 32: Star - graph with 90 nodes

A) Total timing for Star topology

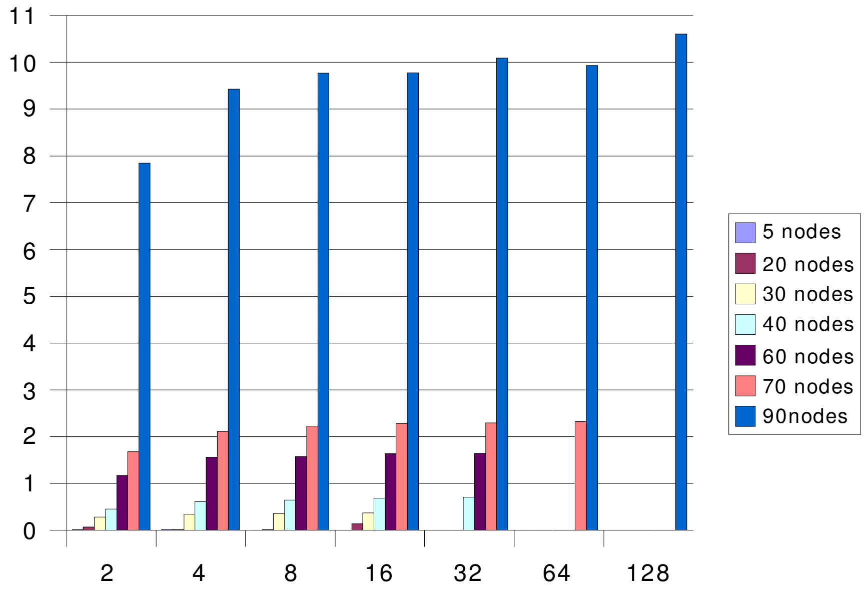


Figure 33: Star topology timings for test graphs

Table 1 gives the run times (*T(s)*) for different graphs with star-like structure partitioned in

2, 4, 8, 16, 32, 64, and 128 partitions. The chart in Figure 33 graphically illustrates the timings in the table.

Number of partitions	Number of nodes							
	5	20	30	40	60	70	90	130
2	0.013432	0.069398	0.2822253	0.448345	1.171291	1.680963	3.057247	7.841835
4	0.022237	0.101626	0.340814	0.608222	1.562072	2.105716	3.708379	9.428316
8	N/A	0.121671	0.355433	0.644864	1.573684	2.221258	3.853826	9.769447
16	N/A	0.137449	0.371197	0.685840	1.637396	2.281283	3.966196	9.774199
32	N/A	N/A	N/A	0.703748	1.64263	2.292259	4.020681	10.08976
64	N/A	N/A	N/A	N/A	N/A	2.316462	4.062516	9.93395
128	N/A	N/A	N/A	N/A	N/A	N/A	N/A	10.60498

Table 1: Run time comparison between graphs with different number of nodes

b) Quality of partitions

The chart in Figure 34 is based on results of a star-like graph *G* with 90 nodes. The percentages on the y-axis show the percentage of nodes from the whole. The x-axis denotes the number of partitions, in which the nodes are distributed. The chart shows a nearly equal distribution of nodes in different partitions.

c) Total edge-cut

In the star topology the edge-cut results are high. As mentioned earlier, edge-cut increases with each link between two nodes in different partitions. Since the algorithm distributes the nodes in the graph in nearly equal partitions (by size), and all the nodes in the graph are connected only to the node in the middle, putting the middle node in a partition leaves $\left\lceil n - \frac{n}{k} \right\rceil$ (see Section 6, ii) condition) other nodes in different partitions, and just as much increases the edge-cut (see Table 2).

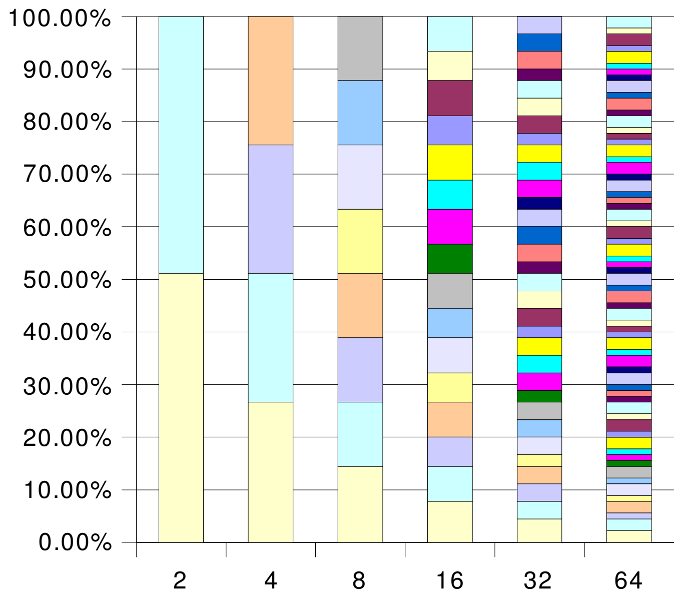


Figure 34: Distribution of nodes in different partitions

Number of partitions	Number of nodes							
	5	20	30	40	60	70	90	130
2	2	9	14	19	29	33	43	64
4	3	14	22	29	43	50	66	96
8	N/A	17	26	34	51	59	77	112
16	N/A	18	28	37	55	65	83	121
32	N/A	N/A	N/A	38	57	67	86	125
64	N/A	N/A	N/A	N/A	N/A	68	88	127
128	N/A	N/A	N/A	N/A	N/A	N/A	N/A	128

Table 2: Total of edge-cuts comparison between graphs with different number of nodes

8.3 Ring

Ring is a network topology in which nodes are connected in a closed loop. Every node is linked to two others.

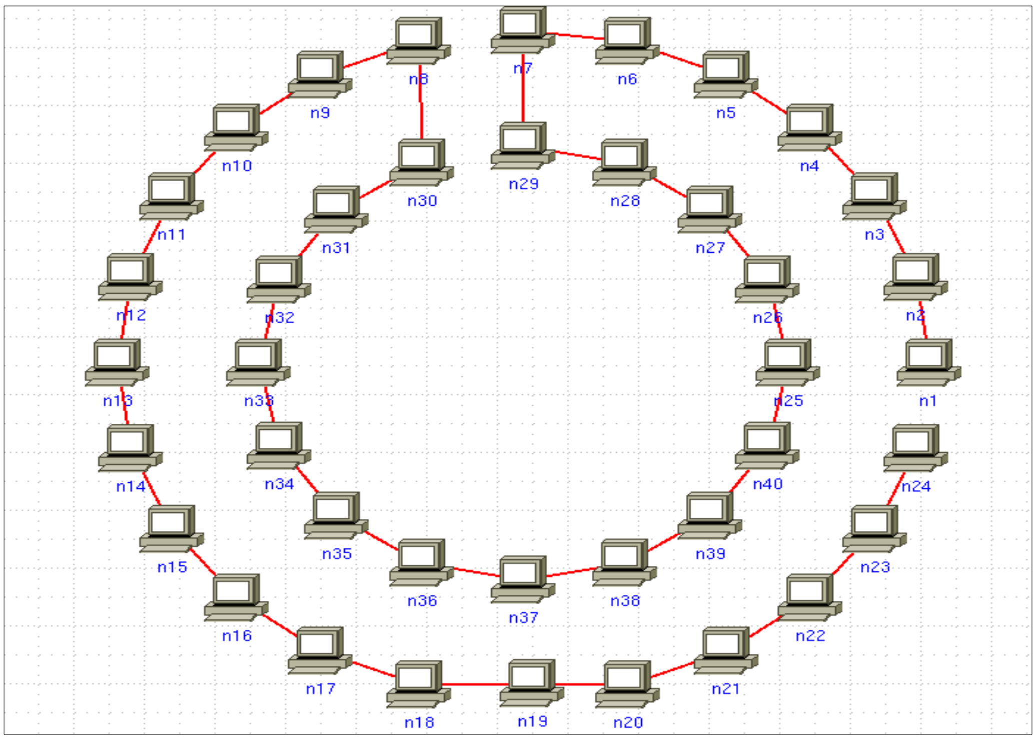


Figure 35: Ring – graph with 40 nodes

6.4.1 Total timing for Ring topology

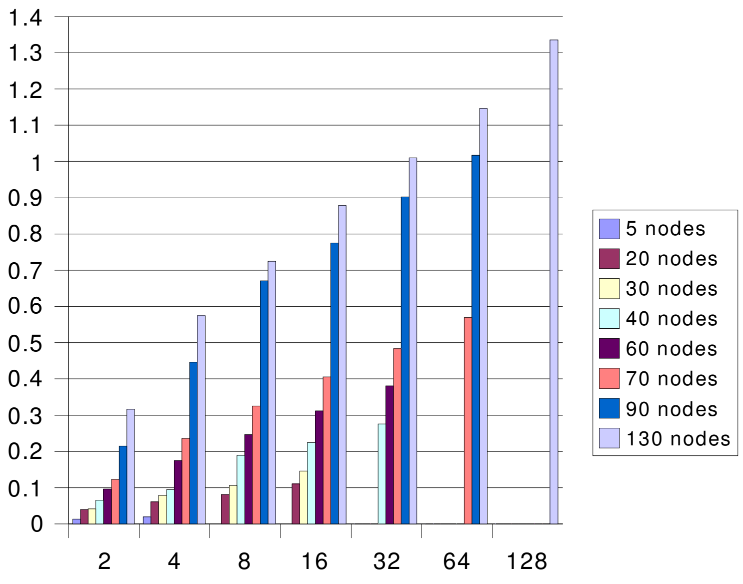


Figure 36: Ring topology timings for test graphs

Total timings for the ring topology are much lower then by star topology (see Table 3). The reason is the coarsening and matching phase, which in just a few steps produces better balanced coarse graphs, than by star topology, which significant improves performance.

Number of partitions	**Number of nodes**							
	5	20	30	40	60	70	90	130
2	0.018514	0.049718	0.05167	0.146507	0.115831	0.143352	0.221096	0.316519
4	0.030207	0.080555	0.10699	0.197555	0.209188	0.308619	0.516873	0.574916
8	N/A	0.119505	0.155408	0.258056	0.311466	0.402530	0.846527	0.724937
16	N/A	0.164115	0.215895	0.359187	0.418082	0.518079	1.024577	0.878811
32	N/A	N/A	N/A	0.475214	0.537911	0.634810	1.214013	1.010585
64	N/A	N/A	N/A	N/A	N/A	0.832473	1.398125	1.146311
128	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.335671

Table 3: Run time comparison between graphs with different number of nodes

6.4.2 Quality of partitions

The chart in Figure 37 is based on the ring-like graph with 90 nodes. As well as the chart for the ring topology, the y-axis shows the percentage of nodes in different partitions, and the x-axis denotes the number of partitions. The chart shows even more equally distributed nodes in partitions than the chart for the star-like graph.

6.4.3 Total edge – cut

The number of edge-cut in ring-like graphs is minimal. For each new partition of the graph, edge-cut increases only by one (see Table 4).

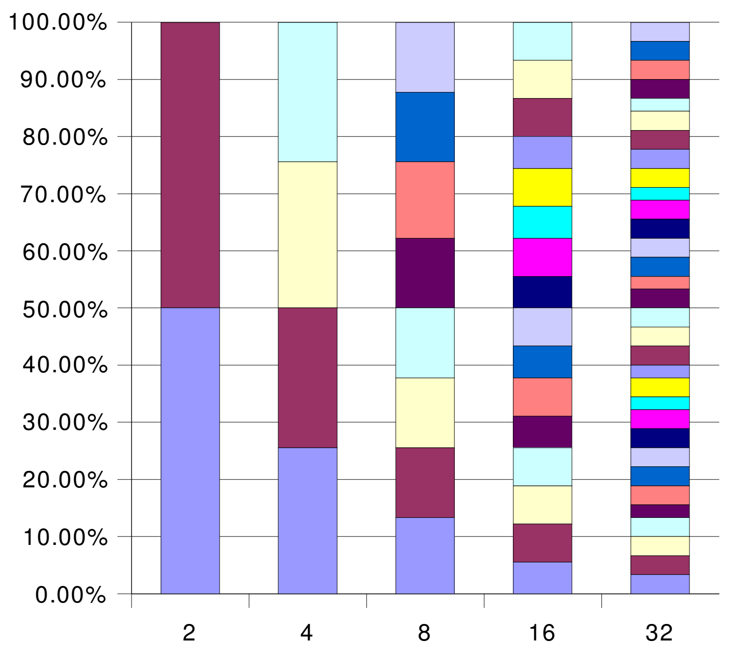


Figure 37: Distribution of nodes in different partitions

Number of partitions	Number of nodes							
	5	20	30	40	60	70	90	130
2	2	2	2	2	2	2	2	2
4	4	4	4	4	4	4	4	4
8	N/A	8	8	8	8	8	8	8
16	N/A	16	16	16	16	16	16	16
32	N/A	N/A	N/A	32	32	32	32	32
64	N/A	N/A	N/A	N/A	N/A	64	64	64
128	N/A	N/A	N/A	N/A	N/A	N/A	N/A	128

Table 4: Total of edge-cuts comparison between graphs with different number of nodes

8.4 Full Mesh

Full mesh topology is a network topology in which nodes are connected with every other node in the network.

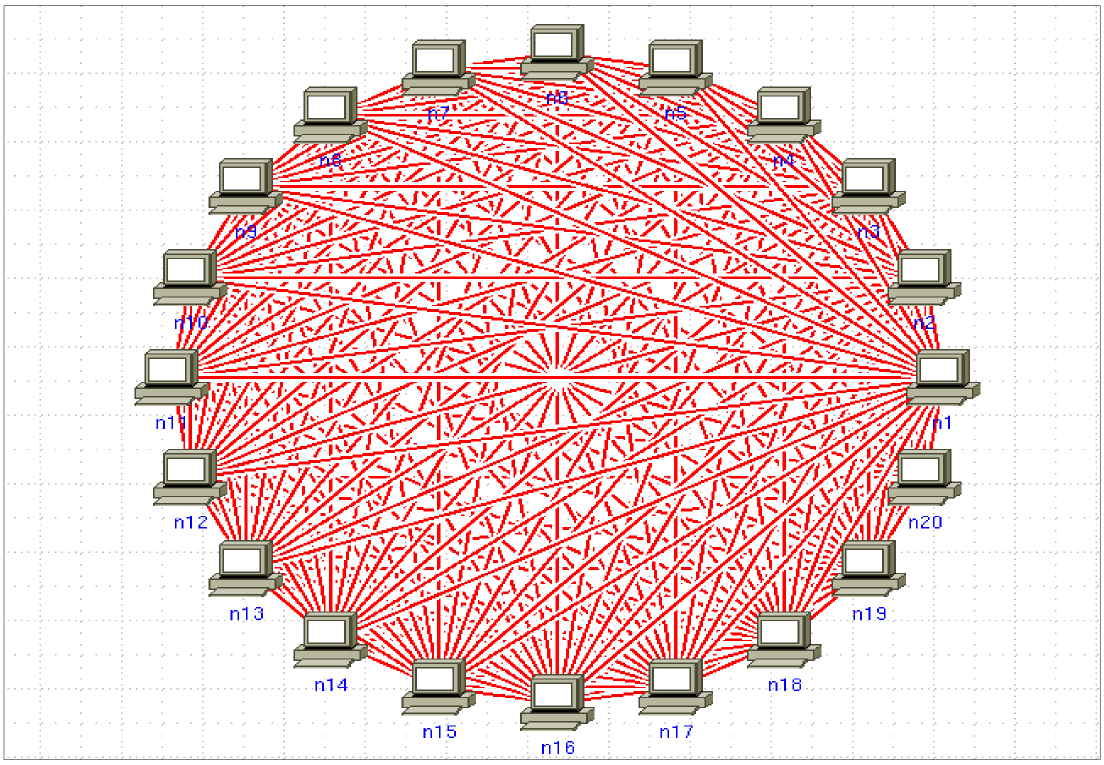


Figure 38: Full mesh graph with 20 nodes

6.5.1 Total timing for full mesh topology

Full mesh is very computationally expensive. As it can be seen from the chart Figure 39 and Table 5, the time needed to partition the graph grows (very fast) with the number of nodes in the graph. For this reason only the small full mesh - like graphs are tested.

6.5.2 Quality of partitions

The chart in Figure 40 shows that the nodes in full mesh graphs are slightly worst distributed than the fall is in other two network topologies.

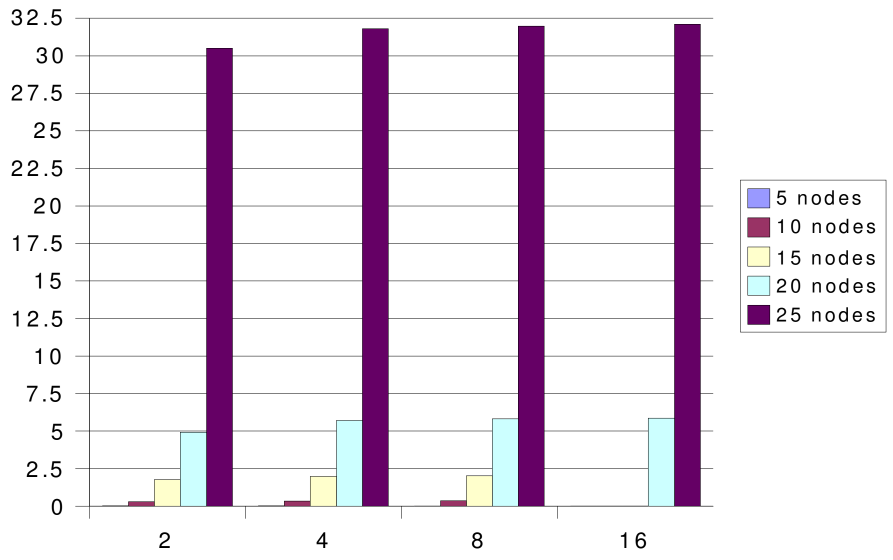


Figure 39: Full mesh topology timings for test graphs

Number of partitions	Number of nodes				
	5	10	15	20	25
2	0.023353	0.302575	1.772264	4.925654	30.503502
4	0.032155	0.350482	1.993322	5.712546	31.810411
8	N/A	0.370302	2.034807	5.823654	31.985632
16	N/A	N/A	N/A	5.856440	32.099377

Table 5: Run time comparison between full mesh graphs with different number of nodes

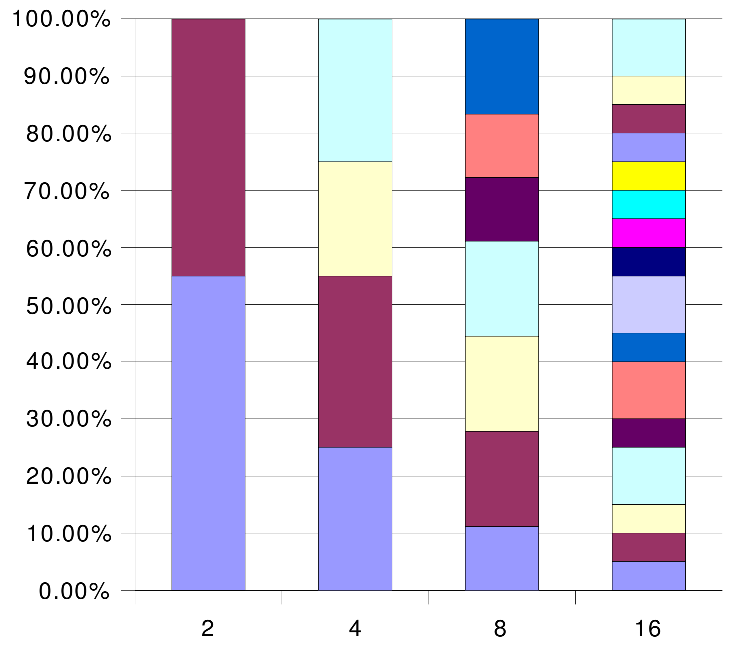


Figure 40: Distribution of nodes in different partitions

6.5.3 Total of edge-cut

Edge-cuts are very high in full mesh - like graphs (see Table 6). This is also understandable, because each node is connected to each other node, which means that at least once each partition is connected to a different one.

Number of partitions	**Number of nodes**				
	5	10	15	20	25
2	6	25	56	99	156
4	9	37	84	149	234
8	N/A	43	98	174	273
16	N/A	N/A	N/A	186	291

Table 6: Total of edge-cuts comparison between graphs with different number of nodes

8.5 Tree

In this section the example graph is taken from a real network map, to see how this algorithm works on a real network with different and interconnected topologies.

Nodes weight and links weight in this test are calculated from each node's and link's characteristics (see Section 6.5).

6.6.1 Total timing and edge-cut

Beside the timing and edge-cut result values, standard deviation for each result is also provided. The deviation of test results in previous tests was not discernible, where in this test it is - because of the differently sized nodes and links, and the algorithm which uses randomized methods to partition the graph.

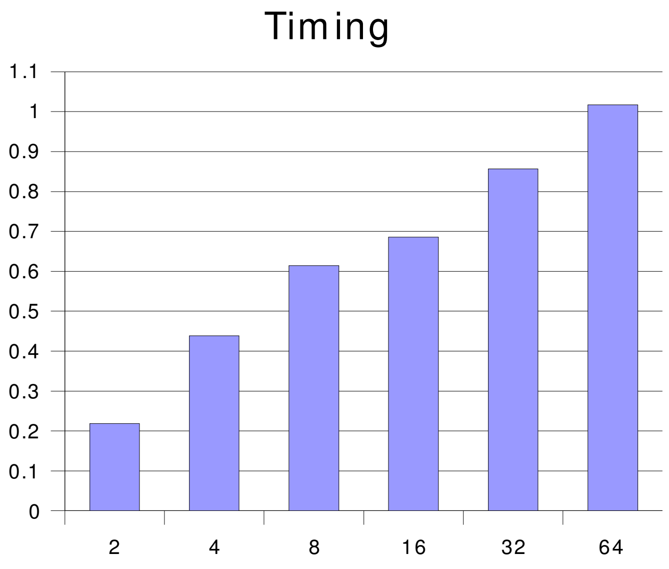


Figure 41: Timing for tree like sample graph

Figure 41 and Figure 42 illustrate the execution time and edge-cut between partitions for tree-like sample graph. Both graphs show a monotonically increase in time, btw. edge-cut. Table 7 contains the data from the example.

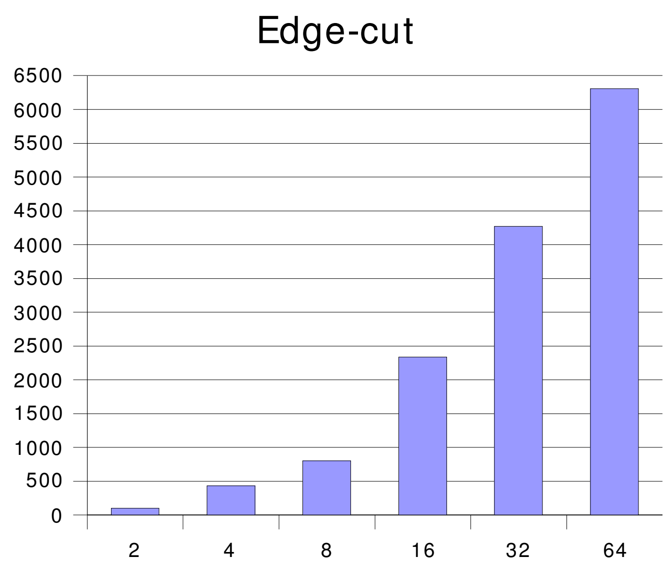


Figure 42: Edge-cut for tree like sample graph

Number of partitions	Timing		Edge-cut	
	Mean	**Std. Deviation**	**Mean**	**Std. Deviation**
2	0.218608	0.013098	101.333336	0.8888889
4	0.43860	0.016194666	435.33334	44.444443
8	0.61426866	0.036359113	802.0	133.33333
16	0.6860487	0.042385112	2336.6667	176.88889
32	0.856333	0.036103334	4272.6665	177.77777
64	1.0168877	0.0568331	6306.0	66.666664

Table 7: Timing and edge-cut comparison for tree like sample graph

6.6.2 Quality of partitions

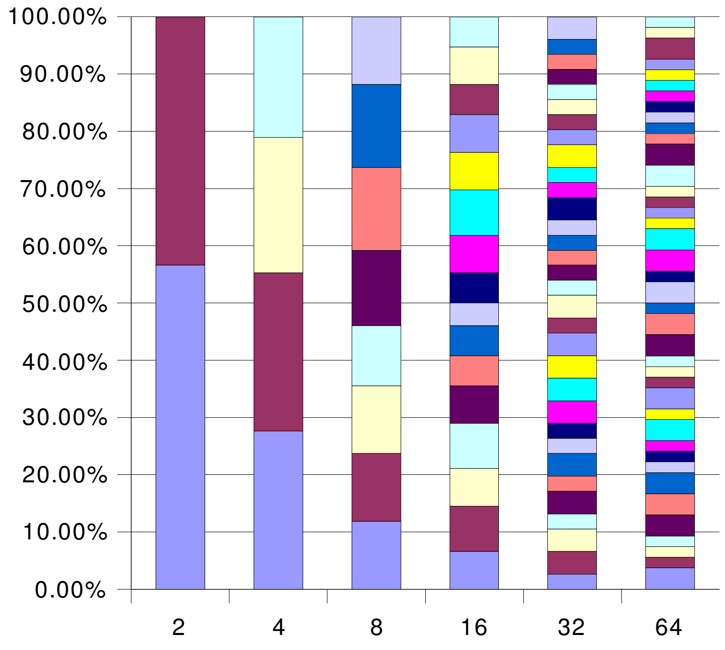


Figure 43: Distribution of nodes in different partitions

Despite the differently sized nodes and links, the partitions are of similar weight, with some partitions weighting a little bit more then the others (see Figure 43).

7 Conclusions

This thesis describes the implementation of a multilevel partitioning algorithm in IMUNES system after the METIS partition algorithm. Multilevel partitioning algorithm is a fast algorithm which is able to obtain partitionings with a very high quality, and utilize the concepts of vertex and edge weights to minimize the edge-cut. The implementation in IMUNES is in Tcl/Tk script language, which makes the algorithm slower than it would be in some procedural or object-oriented programming language.

7.1 Timings

As can be seen from the above results, the algorithm executes quite fast despite it is written in a script language.

The differences in time between the various topologies can be quite large. The full mesh topology takes about hundred times longer then the other topologies to execute. The fastest execution time has the ring topology.

7.2 Partition Quality

In terms of quality of partitioning, there is not much difference between different topologies. The greatest problems with the evenly distribution of the vertices has the full mesh topology. The ring topology can be partitioned in almost perfect partitions. In different interconnected topologies some unbalances can happen, which greatly depends on the complexity and the magnitude of the size difference in graph’s node’s weights.

7.3 Edge-cuts

Edge-cuts can negatively effect the execution time of the application. There are big differences between the various topologies. Most edge-cuts produce full mesh topology, while ring topology produces minimal edge-cuts, that is, just as many edge-cuts as there are partitions.

References

[1] B. Hendrickson, R. Leland, “A Multilevel Algorithm for Partitioning Graphs”, Samdia National Laboratories, http://www.chg.ru/SC95PROC/509_BHEN/SC95.HTM

[2] IMUNES homepage http://web.tel.fer.hr/imunes

[3] G. Karypis, V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”, http://www.cs.umn.edu/~metis

[4] A. Gupta, “Fast and effective algorithms for graph partitioning and sparse-matrix ordering”, IBM Journal of Research and Development, Volume 41, Numbers 1/2, 1997

[5] R. Ricci, C. Alfred, J. Lepreau, “A Solver for the Network Testbed Mapping Problem”, http://www.cs.utah.edu/flux/papers/assign-ccr03-base.html

[6] B. W. Kernighighan, S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs”, http://citeseer.comp.nus.edu.sg/608172.html

[7] G. Karypis, V. Kumar, “Unstructured Graph Partitioning and Sparse Matrix Ordering System”, http://glaros.dtc.umn.edu/gkhome/index.php

[8] J. Demmel, “Graph Partitioning”, CS 267, Applications of Parallel Computers, Spring 2005, http://www.cs.berkeley.edu/~demmel/

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, “Introduction to algorithms”, second edition, MIT Press, 2001

[10] B Kernighan, S Lin, "An efficient heuristic procedure for partitioning graphs", The Bell System Technical Journal, V-29, 1970

[11] C. M. Fiduccia, R. M. Mattheyses, "A linear-time heuristic for improving network partitions", Annual ACM IEEE Design Automation Conference archive, Proceedings of the 19th conference on Design automation table of contents, Pages: 175 - 181,1982

[12] Chaco http://www.cs.sandia.gov/~bahendr/chaco.html

[13] PlanetLab https://www.planet-lab.org/

[14] Emulab http://www.emulab.net/

[15] ModelNet http://modelnet.ucsd.edu/

[16] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, J. Lepreau, „Feedback-directed Virtualization Techniques for Scalable Network Experimentation“, http://www.cs.utah.edu/flux/papers

[17] M. Rimondini, “Emulation of Computer Network with Netkit”

[18] G. Apostolopoulos, C. Chasapis, “V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation”, http://www.ics.forth.gr/ftp/tech-reports/2006/2006.TR371_V-eM_Cluster_of_Virtual_Machines.pdf

[19] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, J. Lepreau, “Flexlab: A Realistic, Controlled, and Friendly Environment for Evaluating Networked Systems”, http://www.cs.utah.edu/flux/papers/flexlab-hotnets06.pdf

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar “An Integrated Experimental Environment for Distributed Systems and Networks”, http://www.cs.utah.edu/flux/papers/netbed-osdi02.pdf

[24] QUAGGA homepage, http://www.quagga.net/

[25] ZEBRA homepage, http://www.zebra.org/

[26] Z. Puljiz, M. Zec, M. Mikuc, “IMUNES Based Distributed Network Emulator”

[27] M. Zec, “Implementing a Clonable Network Stack in the FreeBSD Kernel”, In Proc. of the 2003 USENIX Annual Techical Conference, FreeNIX track, San Antonio, June 2003

[28] B. Hendrickson, R. Leland, “The Chaco User’s Guide”, Sandia National Laboratories, SAND93-2339, October 1993

[29] METIS homepage, http://glaros.dtc.umn.edu/gkhome/views/metis/

[30] B. Parlett, D. Scott, “The lanczos algorithm with selective orthogonalization”, Math. Com., 33:217-239, 1979.

[31] B. Parlett, “The Symmetric Eigenvalue Problem”, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[32] G. Golub, C. Van Loan, “Matrix Computations”, Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[33] B. Parlett, “The rayleigh quorient iteration and some generalizations ofr nonnormal matrices”, Math. Comp., 28(127):679-693, 1974.

[34] C. C. Paige, M. A. Saunders, “Solution of sparse indefinite systems of linear equations”, SIAM J. Numer. Anal., 12:617-629, 1975.

[35] GENI homepage, http://www.geni.net

[36] XORP homepage, http://www.xorp.org

[37] M. Zec, M. Mikuc, “Operating System Support for Integrated Network Emulation in IMUNES”, In Proc. of the 1st OASIS workshop, Boston, October 2004