UNIVERSITÀ DI PISA

Scuola Superiore
Sant'Anna
di Studi Universitari e di Perfezionamento

Università degli Studi di Pisa

Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Master Thesis

Data Center Resource Allocation: a Genetic Algorithm Approach

Candidate:

Giuseppe Portaluri

Supervisors:

Prof. Stefano Giordano

Prof. Dzmitry Kliazovich

Accademic year:
2012/2013

Ai miei genitori,
ai miei nonni,
ai miei zii,
ed ai miei cugini.

Acknowledgements

Il primo ringraziamento va senza dubbio alla mia famiglia, senza la quale non avrei potuto raggiungere nessuno dei traguardi che mi sono prefissato nel corso di questi anni; pertanto ringrazio i miei genitori, ai quali devo tutto, ed il resto della mia famiglia che mi è stata sempre vicina aiutandomi ad affrontare ogni difficoltà e facendomi crescere giorno dopo giorno.

Ringrazio anche i professori con i quali, nel corso di questi anni, ho potuto instaurare un proficuo rapporto nella trasmissione dei Saperi e della Conoscenza ma anche per la possibilità di interazione dialettica che spesso si è determinata e che ha contribuito a farmi crescere anche dal punto di vista umano. Tra tutti un ringraziamento particolare ai miei relatori, per gli stimoli e le possibilità che mi hanno dato nel lavoro per la realizzazione di questa tesi.

Vorrei quindi ringraziare tutti gli amici conosciuti durante questi cinque anni ed in modo particolare: Federico, Alessio e Benedetto con i quali ho condiviso il percorso della Laurea triennale ed anche tante belle e forti esperienze che hanno creato un legame molto saldo tra di noi, le loro famiglie che mi hanno fin dal primo momento accolto con molto affetto, Salvatore e Maria che nei miei due anni a Pisa sono stati a tutti gli effetti la mia seconda famiglia, supportandomi sempre con la loro grande bontà e disponibilità, Elio e Marta per la loro immensa simpatia, generosità e per l'enorme affetto che hanno nei miei confronti, Carmine, Stefano e Valeria per il bel rapporto di amicizia instaurato con ognuno di loro.

Infine, vorrei ringraziare il mio mancato fratello Cosimo, Olimpia, Alberto e Federico e tutti gli altri che hanno dato un loro contributo nella mia vita durante la mia permanenza a Pisa e nella mia breve esperienza lussemburghese ad Esch-sur-Alzette, che non elenco singolarmente esclusivamente per ragioni di bre-

vità.

Contents

List of Figures

Chapter 1

Introduction

In the recent years, data centers changed the way to provide hardware and software resources for high performance, scientific and business computing: with this facility is possible to reach good performances or virtually unlimited computational power without buying the whole needed infrastructure. Data center owner instead has to deal with different problematics respect to end user like for example hardware maintenance and redundancy (because with a large number of devices fault probability of some of them increases), energy and power consumption needed to keep turned on the whole infrastructure, hot air dissipation and cooling for servers and switches and also other internal organization problems regarding for example the proper design of a network topology without bottlenecks providing the best quality of service as possible.

Energy and power consumption in data centers definitely affects management costs. In the last years, different solutions where developed to improve energy efficiency directly on hardware and especially in processors but less improvements were done in the networking part also because percentage of energy cost is less crucial rather than processors and servers in general. Nevertheless, the right displacement of network connections could improve energy efficiency also for the networking part reducing the number of turned on devices.

In this thesis is introduced a *task allocation algorithm* for data centers aiming to find a reasonable trade off between task's completion time and devices power

consumption.

A new model for both data centers and tasks representation was developed. *Three-tier fat tree* topology for data center is considered. Tasks are characterized by a number of instructions to be executed and a bandwidth requirement; the originality in this model lies in this task differentiation: adding this two parameters, allocation problem turns an already existent and solved permutation problem in a more challenging one.

This algorithm is designed using Genetic heuristics that allow both to explore solutions space and to search for the optimal solution in an efficient manner, and it is implemented on a dedicated framework for multi-objective Genetic algorithms, called jMetal[2]. Using Genetic heuristics, is possible to solve allocation problem involving a big number of tasks with better performances, having a measured time complexity which is cubic respect to the number of tasks, rather than other methods like VMPlanner [13] which performs allocation of X Virtual Machines (VMs) in a network topology of Y switches in $O(X^4) + O(Y^{2.5})$ plus the complexity of the Quadratic Assignment Problem that is not practical to be executed with input size greater than 20; moreover VMPlanner doesn't consider at all any kind of computational requirement for VMs to be allocated.

Network flows are allocated and managed with the help of Software Defined Networking (SDN) architecture. SDN decouples control plane from data plane in switches; SDN control plane is centralized and every switch receives the proper forwarding rules according to the controller network view. Through this approach is possible to allocate perfectly connections in the network avoiding congestions and bottlenecks, as first step to realize energy saving also in the networking part. Moreover one of the greater potentiality of SDN is that controllers (which are external respect to the forwarding devices) could implement the developed algorithm and execute directly the task allocation according to the obtained results.

1.1 Thesis structure

This document is structured in several chapters:

- chapter 1 is the introduction of the thesis which presents the examined problem;
- chapter 2 describes data centers focusing especially on the internal network topology and on the power consumption problem;
- chapter 3 explains the genetic heuristics and describes the jMetal framework used to design, implement and execute the algorithm;
- chapter 4 introduces the SDN research field and some developed solutions;
- chapter 5 deals with the implemented algorithm explaining the developed model, and the obtained results;
- chapter 6 summarizes the whole thesis and describes future works.

Chapter 2

Data Center

2.1 Introduction

A data center is a facility that hosts servers and associated components such as telecommunication devices (switches, routers, ...), storage systems, coolers, uninterruptible power supplies, air filters; there is a certain degree of redundancy of the resources in order to avoid single points of failure and more in general to improve fault tolerance. A data center typically houses a large number of heterogeneous networked computer systems and usually it can occupy one room of a building, one or more floors, or an entire building.

There are some motivations behind the establishment of massive data centers both economic and technical: it is possible to achieve economies of scale in order to amortize total costs of bulk deployments and to receive benefits from the ability to dynamically reallocate resources among services reaching better performances through load balancing techniques.

This huge amount of devices requires an internal organization and a specific topology design in order to better exploit the whole set of resources: servers should be interconnected in the proper way to guarantee full accessibility and low communication latency to internal and external customers. Moreover, applications could be distributed on multiple servers therefore highly performant connection between servers of the same data center should be ensured.

Data centers are characterized by two types of network traffic:

- North-South traffic, coming from the external gateway and directed towards the servers (and vice versa);
- East-West traffic or intra-data center traffic, representing the traffic between servers of the same data center e.g. the traffic between distributed applications.

It's important to design an efficient network topology avoiding bottlenecks, single point of failure and reaching good performance and scalability. Last two parameters play a fundamental role when there is an agreement between customers and data center owner which explicitly defines the kind of guaranteed performance and availability.

2.2 Fat-tree topology

From the network bandwidth perspective, the availability of per-server bandwidth is one of the most fundamental requirements affecting the design of modern data centers. The most widely used topology in data centers is the *three-tier fat tree*[4][5]. In this topology, servers are grouped in *rack*s that are able to host from 20 up to 40-50 servers [3].

Three-tier fat tree topology is hierarchically organized in three layers (Fig.2.1):

- *Access* or *Edge*;
- *Aggregation*;
- *Core*.

Access layer provides connection to servers belonging to the same rack, all connected to the same switch called *Top of the rack*. Link between server and top of the rack switch usually has a capacity of 1 Gbps.

In the aggregation layer, top of the rack switches are connected to a pair of *Aggregation switches* through links of capacity 10 Gbps.

The *oversubscription ratio* is the ratio between the offered and the usable capacity. Considering 48 servers per rack, the oversubscription ratio for the access layer is 2.4 (because no more than 48 Gbps could be requested by servers and at most 20 Gbps are provided by the upper layer).

An aggregation switch typically offers 8-12 ports for access layer connections and is connected to all the core layer switches. The set of racks connected to the same couple of aggregation switches is called *pod*. Typical value of aggregation layer oversubscription ratio is 1.5.

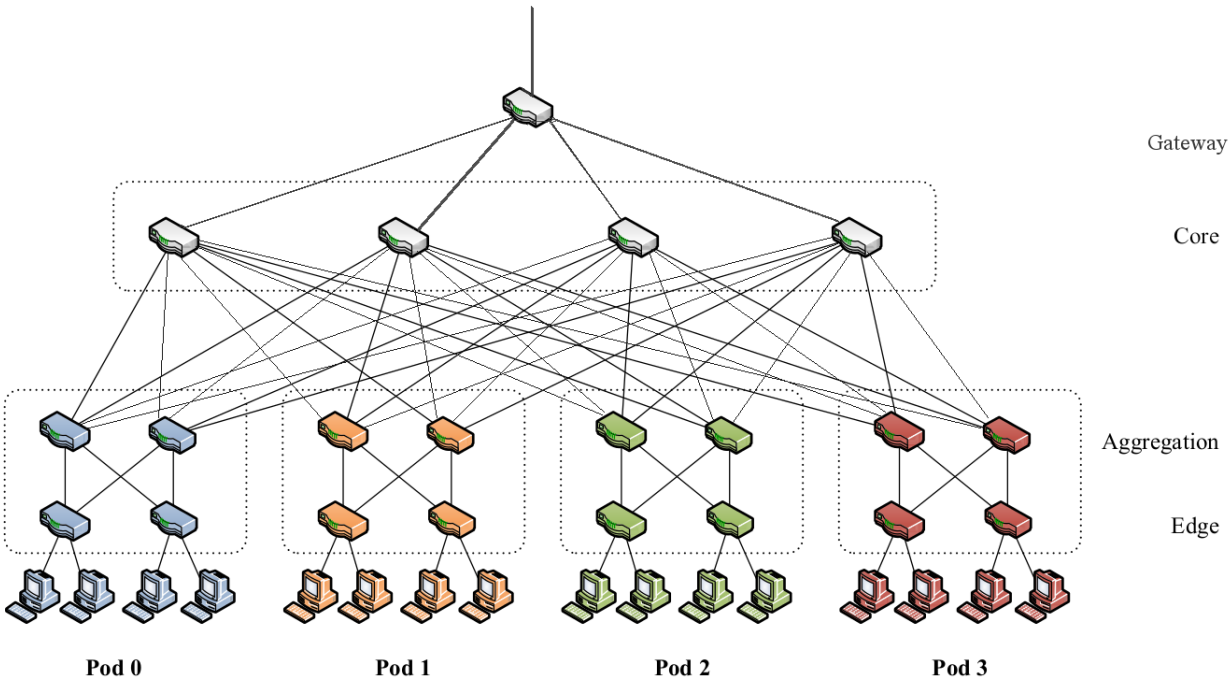

Figure 2.1: Fat-tree topology in data centers

The last layer is the core layer which hosts *core switches*. Usually the number of core switches is limited (in general 8, depending on the data center size). Each aggregation switch is connected to all the core switches allowing a connection to the external gateway and to the other aggregation switches. At the core layer, oversubscription ratio should be 1 or less.

2.3 Other topologies

Three-tier fat tree is not the only available topology: there are other network topologies that could be exploited to interconnect servers in data centers and new solutions are always developed because new ways to structure data center internal topology are currently under research to improve reliability, availability and performance of the network.

2.3.1 Portland

An approach which slightly modifies fat-tree topology is Portland, mainly based on a centralized manager called *fabric manager*. Fabric manager maintains a soft-state of the network, recording possible faults and creating multipaths exploiting some protocols that allows to keep informations about the network. This entity could be an application on a network host or an element of a separate control network. Thanks to this centralization element which stores informations about link status, it's possible to compute alternative paths for unicast and multicast/broadcast communications in case of fault. Keepalive messages are used to determine if a link either active or failed. If a link fails, fabric manager computes the new paths storing the informations about the actual network status and sending these informations to all the switches.

2.3.2 DCell

DCell approach doesn't exploit the tree topology; it uses instead a recursive definition for its construction and composition; in order to realize this kind of topology, each server must be equipped with multiple network ports. DCell is structured in levels and the basic one, named DCell_0, contains n servers connected to a single mini-switch. Typical value of n is less than or equal 8 therefore the mini-switch requires only a limited number of ports. The next level of DCell is named DCell_1 and it's created grouping and connecting all together $n + 1$ DCell_0. Within the same DCell_1, a DCell_0 is connected to all the others: each server of the same DCell_0 is connected to another server of a different DCell_0(Fig.2.2). With this approach,

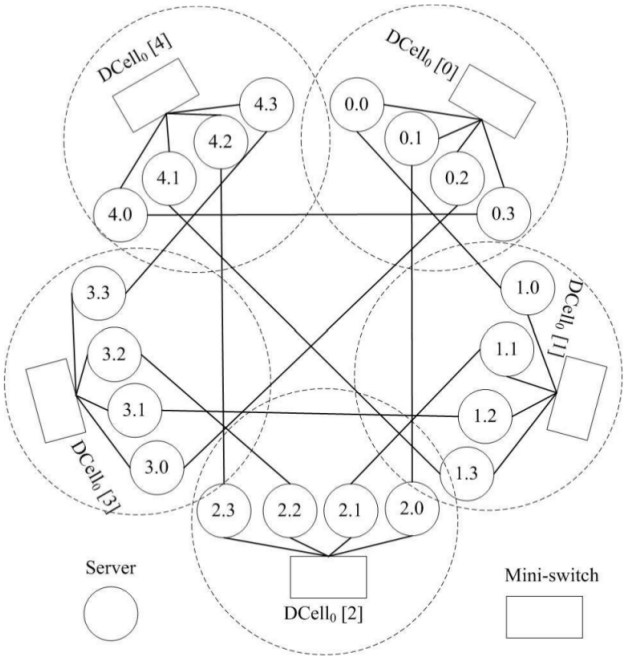

Figure 2.2: DCell_1 structure

every DCell_0 is treated like a virtual node and it is fully connected to the others. Generalizing this procedure, having DCell_{k-1} with t_{k-1} servers it is possible to create up to $(t_{k-1}+1)$ DCell_{k-1}. More in general, when all the necessary DCell_{k-1} are created, each one is fully connected to the others and a DCell_k is created; DCells are connected through servers (except for DCell_0 in which mini-switches are used) therefore to realize a DCell_k, $k+1$ ports per server are needed.

DCell approach compared with Fat-tree guarantees better bandwidth performance and implements fault tolerant routing avoiding single points of failure.

2.3.3 Comparison with Fat-tree

Portland introduces some features to the classical fat-tree view maintaining the former topology. However, using SDN switches in the data center is possible to obtain same or more improvements collecting more details about the network status (section 4).

Considering the second approach, fat-tree is less scalable respect to DCell because

top of the rack switches represent a single point of failure while in case of aggregation or core switch fault, network performance decreases dramatically.

Although these new approaches solve this kind of problem, fat-tree is actually one of the most diffused topology in data centers representing a consolidated structure with predictable and known performances.

2.4 Data center power consumption

One of the main obstacles to data center scale-economy problem is power consumption. Total power consumption due to data center, in a country such as US, doubled in 6 years (from 2000 to 2006) reaching nearly 61 billion kW h (1.5% of total US electricity consumption)[13]. Out of each watt delivered, about 59% goes to the IT equipment (represented by servers, consuming between 39% and 49% and by switches consuming between 10% and 20% of the total amount of power delivered), 8% goes to power distribution loss, and 33% goes to cooling.

Most of the current researches in data centers focus on saving power, proposing more energy and power efficient technologies. Servers and cooling systems represent the top two power consumers in current data centers.

On the other side, network infrastructure (such as routers, switches and links) contributes only with a relative small proportion in the total power consumption: only the 10%-20% of the energy budget of a data center is employed in these devices.

Power consumption in data center is expected to reach great values growing fast in the next years. Following this growth, large slices of power will be dissipated during the delivery phase and to cool network elements; for these reasons now it's the time to address the challenge for reducing the network power consumption without affecting network system performance in data centers.

During the last years, power management in computer has evolved implementing hardware support for power performance and idle sleep states. Two techniques are used: the former intends to save power during active times by lowering down operating frequency and/or voltage to different scales (Dynamic Voltage and Frequency Scaling or DVFS) while the latter tries to save power during idle times, by powering

down as much as possible all the sub-components. This kind of philosophy is also followed by researchers with respect to network power consumption: i.e. reducing the number of used path aggregating (*consolidating*) the traffic, a network operator can free some devices and put them to sleep reducing power consumption, but this kind of solution requires a good degree of coordination between every device present in the network. This approach is viable because in data center, network utilization has been found very low during most of the time (even a quota higher then 40% of link capacity is unused or idle), and the network capacity is generally far from being exceeded by the traffic load.

Chapter 3

Genetic Algorithm

3.1 Introduction

Genetic Algorithms (GAs) are stochastic search algorithms based on the mechanism of natural selection and natural genetics. This kind of algorithms differs from conventional search techniques because they start with an initial set of random solutions called *population* which satisfies some specified boundaries and/or system constraints characterising the problem and then, during the execution of the algorithm, solutions of the population evolve towards the optimal one. Each individual in the population is called a *chromosome*, representing a solution to the problem at hand. Chromosome is in general a string of symbols and usually, but not necessarily, a binary bit string. Each chromosome comprises a number of individual structures called *genes* describing part of the solution. Chromosomes evolve through successive iterations called *generations*. During each generation, the chromosomes are evaluated, using some measures of fitness through specific *fitness functions*. To create the next generation, new chromosomes, called *offspring*, are formed by either merging two chromosomes from current generation using a *crossover operator* or modifying a chromosome using a *mutation operator*. A new population is formed, using the *selection operator*, choosing the best chromosomes and rejecting the others keeping the population size constant. Fitter chromosomes have higher probabilities of being selected respect to the others. After several generations, the algorithms converge

to the best chromosome, which hopefully represents the optimum or sub-optimal solution to the problem.

3.2 Searching for Solutions

Search is the core phase for GA problem solving methods. There are two important issues in search strategies: *exploiting* the best solution and *exploring* the search space. GA is a class of general purpose search methods which is able to combine elements of directed and stochastic search producing a good balance between exploration and exploitation of the search space.[1]

3.3 Chromosome

Chromosomes represent specific solutions for the problem. Each chromosome comprises a number of individual structures called *genes* and each gene has an associated value called *allele*.
Genes can be represented or *encoded* in different ways like binary strings, integer or real numbers, arrays, array permutations or other defined data structures. Real number encoding performs better for function optimization problems while integer and permutation encoding are utilized for combinatorial optimization problem. For each kind of gene a proper set of operators should be defined.

3.4 GA Operators

3.4.1 Crossover

Crossover represents one of the most important GA operators. It works on two different chromosomes at a time, generating an offspring combining both chromosomes' features. As example, if we consider one chromosome encoded like a binary string, one crossover operator could take two chromosomes, cut them in the same position (cut the related encoded binary string) and take the first part from one parent and

the last from the other. In this way the generated chromosome includes parts of both parents. The *crossover probability* is the probability to apply the crossover operator to a couple of chromosomes. Denoting pop_Size the population size and P_c the crossover probability, the expected size of the offspring is $P_c * pop\_Size$.

Examples of crossover operator are:

- *one-cut point crossover*(Fig.3.1): two chromosomes choose the same random-cut point dividing them in two parts (left and right) and generate the offspring taking the left segment from one parent and the right segment from the other one and/or vice versa.

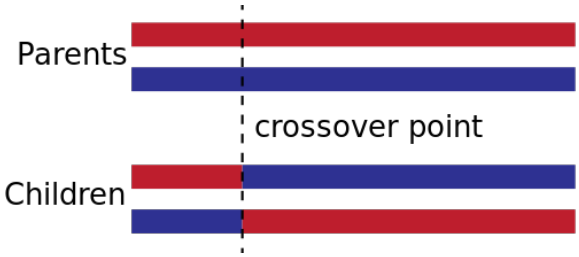

Figure 3.1: One-cut point crossover

- *two-cut points crossover* (Fig.3.2) and *multi-cut points crossover* which follow the same approach of before, applying two or more cut points instead of one;

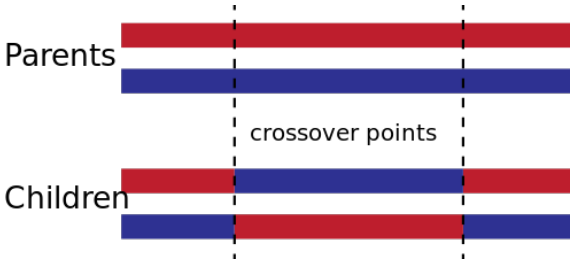

Figure 3.2: Two-cut point crossover

- *uniform crossover* (Fig.3.3): a fixed mixing ratio between parents is decided and the offspring is generated mixing the two parents according to that ratio in a probabilistic way; uniform crossover enables the parent chromosomes to contribute at gene level rather than at segment level.

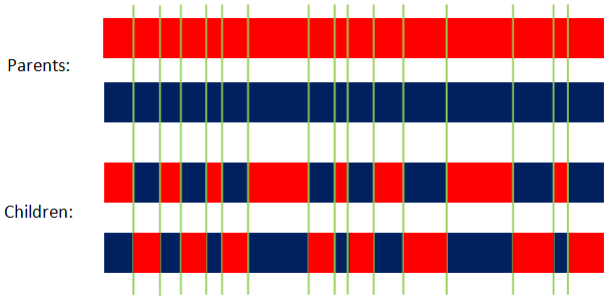

Figure 3.3: Uniform crossover with mixing ratio 0.5

The above described crossover operators are not able to destroy one gene: every time that one cut is executed, the cut falls always between two consecutive genes and never in the middle of a single gene.

3.4.2 Mutation

Mutation is an operator which produces random changes in various chromosomes and usually it accomplishes two important tasks:

1. replaces the lost genes from the population during the selection process in order to tried them again in a new context;

2. provides the genes that were not present in the initial population;

One of the most simple way to achieve a mutation is to randomly change one gene of a chromosome. The mutation probability (P_M) represents the probability of introducing new genes into the population. A good tuning of this probability is needed because if it is too high, offspring will be too much different respect to the parents losing the possibility to learn by the previous history while if it's too low, some important genes could never get in the solutions; For each gene of a given chromosome, the mutation probability is evaluated and the value of the gene is changed according to the adopted mutation strategy.

3.4.3 Selection

Selection operator is used to select the population for the next generation among the different chromosomes. This operators drives the algorithm in a certain region of the solution space; if the pressure of this operator is low, wider areas of the solution space are explored otherwise, with an high pressure, this operator will focus on narrower areas of the solution space. A good strategy is to implement a selection operator with a low pressure in the beginning and a high pressure at the end; in this way, a wider solution region is explored at the beginning, focusing on the best local solutions only at the end.

One of the most important selection operator is called *Tournament selection*. It runs a "tournament" between a certain number of individuals chosen at random from the population and selects the winner according to the fitness values; selection pressure can be easily adjusted by changing the tournament size indeed if the tournament size is larger, weak individuals have a smaller chance to be selected. When the tournament size is 2, this kind of tournament is called *Binary Tournament*.

3.5 Solutions

Generally, an algorithm for solving optimization problems is a sequence of computational steps which asymptotically converge to the optimal solution. Most classical optimization methods are based on gradient or higher order derivatives of objective function applied to a single point in the search space. This point-to-point approach embraces the danger of failing in local optima. GA performs a multi-directional search by maintaining a population of potential solutions. This approach has different advantages:

- *Adaptability*, GA does not have much mathematical requirement about the optimization problems and, due to the evolutionary nature, GA will search for solutions without regarding to the specific inner workings of the problem handling any kind of objective functions and any kind of constraints, i.e., linear or nonlinear, defined on discrete, continuous or mixed search spaces;

- *Robustness*, the use of evolutionary operators makes GA very effective in performing a global search (in probability); respect to most of the conventional heuristics that usually perform a local search, it has been proved by many studies that GA is more efficient and more robust in locating optimal solution, reducing computational effort more than other conventional heuristics[1];

- *Flexibility*, thanks to the possibility to hybridize GA with domain-dependent heuristics in order to make an efficient implementation for a specific problem.

3.5.1 Legal and feasible solutions

Using a GA approach it's necessary to deal with some issues: sometimes generating randomly the initial population or applying mutation or crossover operator, is possible to create unallowed chromosomes, mainly divided in *infeasible* and *illegal* chromosomes.

An infeasible chromosome represents a solution which lies outside the feasible region of a given problem (usually defined by some constraints) while an illegal chromosome doesn't represent a solution at all.

Infeasible chromosomes are usually considered within the algorithm because often in constrained optimization problems, the optimum typically occurs at the boundary between feasible and infeasible areas.

Illegal chromosomes usually are originated by the used encoding technique; in many combinatorial optimization problems, problem-specific encodings are used and such encodings usually yield illegal offspring applying a simple one-cut point crossover. In that situation different strategies could be adopted to deal with this kind of chromosomes:

- *Rejecting strategy*, discarding all infeasible chromosomes created;

- *Repairing strategy*, turning an illegal chromosome into a legal one;

- *Modifying Genetic operators strategy*, creating operators working only into the feasible region;

3.6 Multiobjective GA

Optimization methods could involve one or more parameter to be optimized. With only one parameter we are dealing with a *single objective* optimization problem, with more than one with a *multiobjective* optimization problem. A single objective optimization problem is expressed into the following form:

$$max\,\{z = f(\mathbf{x})\}$$

$$\mathbf{x} = \mathbf{x}(x_1, x_2, ..., x_j, ..., x_n);$$

$$\text{subject to } g_i(\mathbf{x}) \leq 0,\ i = 1, 2, ..., m$$

$$\text{and } x_j > 0,\ j = 1, 2, ..., n$$

with

- $\mathbf{x} \in R^n$ a vector of n decision variables;
- $z \in R$ the objective;
- $f : R^n \to R$ the fitness function;
- $g_i : R^n \to R$ a generic constraint functions.

In this scenario, is really simple to classify the solution: it is sufficient to compare the fitness value and create a descending ordered set respect to those values.

More in general with multiple objective we have:

$$max\,\{z_1 = f_1(\mathbf{x}), z_2 = f_2(\mathbf{x}), ..., z_q = f_q(\mathbf{x})\}$$

$$\mathbf{x} = \mathbf{x}(x_1, x_2, ..., x_j, ..., x_n);$$

$$\text{subject to } g_i(\mathbf{x}) \leq 0,\ i = 1, 2, ..., m$$

$$\text{and } x_j > 0,\ j = 1, 2, ..., n$$

In this case, there does not necessarily exist a solution that is the best with respect to all objectives because of incommensurability (it's not possible to compare vectors between each others) and conflict among objectives.

Consider two solutions $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{x}$ *dominates* $\mathbf{y}$ if exists a value k such that:

$$\begin{cases} f_k(\mathbf{x}) > f_k(\mathbf{y}) \\ f_l(\mathbf{x}) \geq f_l(\mathbf{y}) \quad \forall l \neq k \end{cases}$$

If that value of k doesn't exist, $\mathbf{y}$ is *nondominated* by $\mathbf{x}$.

The main task of a multiobjective GA is to find a set of *nondominated* solutions (or *Pareto optimal* solutions) optimizing the current problem.

3.7 JMetal Framework

3.7.1 Introduction

JMetal is a Java-based framework designed for multi-objective optimization. It is publicly available to the community of people interested in multi-objective optimization. It is licensed under the GNU Lesser General Public License 2 and it can be obtained freely. This framework is specifically oriented to multi-objective optimization rather than other frameworks that are mainly focused on single objective optimization implementing only a general support to multiobjective optimization. JMetal implements the following features:

- a number of modern multi-objective optimization metaheuristics like *Non dominating sorting algorithm* II (NSGA-II), MOCell and others;
- a rich set of test problems;
- different variable representations;

3.7.2 Java Implementation

Framework classes are named and built in order to make them general enough to be used in any context. The main classes of jMetal are:

- Algorithm;
- Solution;

- SolutionSet;
- Variable;
- Problem;
- Operator.

Class **Algorithm** represents the superclass for all the optimizers: whatever metaheuristic included in jMetal has to inherit from it. Application-specific parameters can be added and accessed through the methods *addParameter* and *getParameter*. Similarly, an algorithm may also use of some operators incorporated with the method *addOperator* and retrieved by the *getOperator* method.

In the context of evolutionary algorithms, populations and individuals correspond to **SolutionSet** and **Solution** jMetal classes: in this implementation class SolutionSet represents a set of Solution objects. Solution objects are typed by **SolutionType** objects and encapsulate an array of **Variable** objects. Variable is a superclass used to describe different kinds of representations for solutions. A Solution object non necessarily has to contain variables of the same representation indeed it can be composed by an array of mixed variable types. Furthermore, this design method provides a good extensibility because new solution representations can be easily incorporated into the framework just by inheriting from the class Variable.

In jMetal, all the problems have to inherit from class **Problem**. This class contains two basic methods: *evaluate* and *evaluateConstraints*. Both methods operate with Solution objects representing candidate solution to the problem; the former determines the evaluation of a solution in terms of its fitness functions while the latter determines the overall constraint violation of a solution. All the problems have to define the evaluate method, while only problems having side constraints need to define evaluateConstraints.

Operator is a superclass for all the operators. As Algorithm class, Operator contains the *getParameter* and *setParameter* methods, which are used to add and access to specific parameters like crossover probability for crossover operators.

NSGA-II

Nondominated Solution Genetic Algorithm II (NSGA-II) is the second version of one of the first search strategy heuristic able to find solutions for multi-objectives GA in a single run; in the past each objective should be considered one at time[6]. NSGA-II has a complexity of $O(MN^2)$ with M the number of objectives and N the population size and improves the former algorithm which was able to perform only $O(MN^3)$. This heuristic sorts a population into different nondomination levels with a procedure called *ranking*. Initially, a random parent population is created. The population is sorted according to the property of domination: if a solution p dominates a solution q, q belongs to the dominated set of p. This procedure is repeated for every solution creating different groups or *nondomination levels* (solutions of the same group are nondominating themselves); an integer value called *rank* is assigned to each nondomination level (1 is the best level, 2 is the next-best level, and so on).

When applying selection and sorting, NSGA-II is able to deal with constraints and unfeasible solutions. Using a binary tournament selection with a single objective the possible cases confronting two solutions are:

- both solutions are feasible and the one with the best fitness is chosen;
- only one solution is feasible and that is chosen;
- both solutions are unfeasible and the one with the smallest overall constraint violation is chosen.

Using multiobjective optimization, it's necessary to introduce the definition of *constraint domination*: a solution i is said to constrained-dominate a solution j, if any of the following conditions is true:

1. solution i is feasible and solution j is not;
2. solutions i and j are both infeasible, but solution i has a smaller overall constraint violation;
3. solutions i and j are feasible and solution i dominates solution j.

Applying a binary tournament operator, constraint-dominating solutions are preferred. All feasible solutions are ranked according to their nondomination level which is based on the objective function values. However, among two infeasible solutions, the solution with a smaller constraint violation has a better rank. If two or more solutions belong to the same nondomination level, another parameter called *crowding distance* is used to complete the ranking procedure. The crowding distance value of a solution provides an estimate of the density of solutions surrounding that solution. Crowding distance of point i is an estimate of the size of the largest cuboid enclosing i without including any other point(Fig. 3.4).

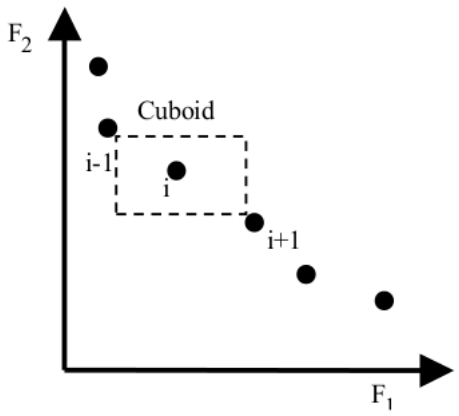

Figure 3.4: Crowding distance for i-th solution in a two objectives algorithm

Boundary solutions which have the lowest and highest objective function values are given an infinite crowding distance. Solution A is better ranked than B if and only if:
$nondominated\ level(A) < nondominated\ level(B)$
or

$$\begin{cases} nondominated\ level(A) = nondominated\ level(B) \\ crowding\ distance(A) > crowding\ distance(B) \end{cases}.$$

When the ranking procedure is done, a number of chromosome equal to the population size is taken from the best ranked solutions. After this selection, crossover

and mutation operator can be applied to the new population in order to generate another offspring.

Chapter 4

Software Defined Networking

4.1 Introduction

Software Defined Networking (SDN) [8] is an emerging dynamic and a flexible network architecture in which the network control plane logic is programmable and decoupled from the forwarding plane.

SDN provides a framework which enables the centralized control of data plane elements, completely independent respect to the used network technologies.

One of the advantages of this centralized control, relies on the possibility to maintain the network-wide view updated. With this functionality is possible to improve network management and configuration permitting also to software developers to rely upon network resources in the same way they do with computing resources. The intelligence present in SDN is localized in software-based controllers while network devices act simply like packet forwarding devices that can be programmed through open interfaces such as OpenFlow[9].

SDN research field is recent, it's actually growing fast and some important challenges have to be addressed: for example, the grain of SDN control could include not only packet forwarding at switching level but also link tuning at data link level optimizing communication in a cross-layer way.

4.2 Benefits

Software defined networking, with its inherent decoupling of control plane from data plane, offers a greater control of a network through programming.

The first benefit is the automatic configuration of the network: when a new device is attached to an existent net, it has to be configured and, specially in case of heterogeneous devices, manual intervention is subject to error. During this phase SDN really helps thanks to the single point of configuration of the control plane.

Another important benefit due to the centralization, is the possibility to create global view of networks instead of managing only local informations; with more informations is possible to optimize and improve performance of the network: actual optimization approaches, that are based on local information without cross-layer consideration, could lead to sub-optimal performance, or even worse to conflicting network operations; with SDN is possible to develop different kind of approaches able to reach optimal configurations.

Elasticity of SDN allows to implement simplified and full programmable testbeds useful to prepare experiment, to deploy and test new applications and functionalities; in this sense, SDN encourage innovation because it allows the possibility of testing application under specified network conditions and performances. Progressive deployment of new ideas can be performed through a seamless transition from an experimental phase to an operational phase also with the support of a SDN network simulator like Mininet[10]

4.3 SDN Reference Model

Reference model for SDN consists of three layers:

1. *Infrastructure layer*;
2. *Control layer*;
3. *Application layer*.

4.3.1 Infrastructure Layer

This layer consists of switching devices in the data plane. These elements have two main functions:

- collect network information, storing them in a local memory and flushing them to the centralized controller;
- forward packets according to the received rules.

Connections among switching devices could be realized through different transmission media, including copper wires, wireless radio, and also optical fibers.

Switches are completely depleted from their "intelligence": they are only able to apply received rules and to collect data; for this reason this architecture could require to produce a new kind of devices or to adapt the existent ones enabling software defined networking functionalities.

In order to be more efficient, SDN switches should improve memory utilization according to the network scale. In case of insufficient memory space, packets would be dropped or directed to the controller for further decisions on how to process them, resulting in a degraded network performance. Memory saving techniques already implemented in classical routers (like route aggregation) can reduce the memory usage by aggregating several routing records with a common routing prefix to a single new routing record with the common prefix.

Into the data plane, SDN switch first identifies the forwarding rule that matches with the packet and then forwards the packet to next hop accordingly. SDN packets could be forwarded not only (like in legacy routers) according to IP or MAC address but also to TCP/UDP connections, VLAN tags and other parameters. Using a long vector for forwarding decision would undoubtedly increases processing complexity, resulting a fundamental trade-off between cost and efficiency in SDN packet processing; implementing for example hardware classification for incoming packets based on header fields in the onboard Network Interface Controller (NIC), switch CPU is exempted from the lookup process and performance increases.

SDN switching devices are divided in three categories according to their implementation if it's done:

- on general PC Hardware, which is usually a software running upon a Linux operative system; software switches provide a port density limited to the number of NICs onboard and relatively slow packet processing speed using software processing but it's possible to implement also virtual switching for VMs;

- on open network hardware (e.g. NETFPGA), offering a vendor independent and programmable platform to build networks for research and experiments; this kind of switches are more flexible than vendor's switches and provide higher throughput than the one with software implementation;

- on vendor's switch; actually hardware vendors are releasing their SDN strategies and solutions, along with a vast variety of SDN-enabled switches; there are also other project to turn old switches in SDN switches enabling all the required features using firmware upgrades.

4.3.2 Control Layer

Control layer is a bridging layer between infrastructure and application layers having two defined interfaces; the one interacting with the infrastructure layer is called *south-bound interface* and it specifies interfaces to access functions provided by switching devices possibly including network status report and packet forwarding rules import.

Interface between this layer and application layer is called *north-bound interface*; it provides service access points in various forms, for example Application Programming Interfaces. Since multiple controllers could coexist for a large administrative network domain, it's possible to have another kind of communication at this layer, defining an *east-west communication*.

SDN controller deals with two types of objectives: *network controlling*, including policies imposed by the application layer and packet forwarding rules for the infrastructure layer and *network monitoring*, in the format of local and global network status. Therefore, because of these two objectives, the logical architecture has two counter-directional information flows from application to infrastructure layer

and viceversa. Leveraging these architectural principles, the logical design for SDN controllers can be decoupled into four building components:

- High Level Language: this function dictates a communication protocol between application and control layer in order to translate application requirements into packet forwarding rules. This language should embrace an expressive and comprehensive syntax for SDN applications to easily describe their requirements and network management strategies.

- Rules Update: control plane is in charge to update forwarding rules in switching devices; rules need to be updated because of configuration changes and dynamic control; in the presence of network dynamics, consistency is a basic feature that rule update should preserve to ensure proper network operations and preferred network properties, such as, loop free, no black hole, and security. There are two defined kinds of rules consistency:

  - Strict, ensuring that either the original rule set or the updated rule set is used;
  - Eventual, in which later packets use the updated rule set eventually at the end the update procedure while earlier packets of the same flow could use rules set before or during the update procedure;

  One possible solution to guarantee strict consistency is to stamp each packet with a versioning number indicating which rule set should be applied. Then, the packet will be processed depending on the version of the applied ruleset. Later packets will be stamped to take the updated rule set. Thus, no more packets will take the original rule set after a time long enough and the original rule set will be removed. Nevertheless, both the original and updated rulesets are kept in switching devices before the originale one expires and is removed.

  Eventual consistency could generate undesired behaviours but it allows memory saving because with this implementation it's not necessary to save both the old rules and the updated ones.

- Network Status Collection: SDN controllers collect network status to build a global view of the entire network and provide to the application layer the necessary informations like network topology graph, for network operation decisions; main network status is represented by traffic statistics such as duration time, packet count, byte count, and bandwidth share of a flow; local traffic statistics may be retrieved by controllers in the so called *pull mode* or proactively reported to controllers using the *push mode*.

- Network Status Synchronization: with a centralized controller some problems can occur like bottlenecks and single point of failures, typical of every centralized approach; common solution to overcome these problems is deploying multiple controllers acting peer, backup, or replicate controllers. Maintaining a consistent global view among all controllers is essential to ensure proper network operations because inconsistent or stale states may result in the application layer making incorrect decisions, leading to inappropriate or sub-optimal operations of the network. To maintain consistency between different controller instances there are several methods like *publish/subscribe* systems and protocols of communication among multiple controllers that are widely used to achieve a synchronized global network view.

4.3.3 Application Layer

Last layer contains SDN applications designed to fulfill user requirements. SDN applications are able to access and control switching devices at the infrastructure layer. Example of SDN applications could include dynamic access control, seamless mobility and migration, server load balancing, network virtualization. SDN applications are able to manipulate the underlying physical networks using the high level language provided by the control layer. An example of application in the routing domain is the one using a cross-layer approach which is a highly touted technique to enhance integration of entities at different layers in a layered architecture allowing entities at different layers to exchange information among each other. This is feasible with SDN because the offered platform for applications easily accesses

network status information and then cross-layer approaches can be developed. Applications could involve also network security indeed thanks to the centralized approach, if attacks are detected, SDN can install proper packet forwarding rules to all the switching devices at the same time blocking quickly the attacking traffic; another functionality at application level is *network virtualization* the allows multiple heterogeneous network architectures to cohabit on the same shared infrastructure; conventional virtualization methods using tunnels and VLAN or MPLS tags require tedious configurations on all the involved network devices; SDN instead offers a platform allowing configuration of all switching devices in a network from a single controller.

4.4 SDN and task allocation

SDN is a good support for task allocation within a data center: exploiting its functionalities, it's possible to efficiently reserve and manage network resources during task allocation phase. Running the resource allocation algorithm such as an application on the SDN controller, is possible to perfectly allocate tasks according to their bandwidth requirements making the real performances of the algorithm closer to the expected ones.

Chapter 5

Resource Allocation

5.1 Introduction

Due to the rapid growth of demands for computational power in data center for scientific highly performant, business and web related applications, data center size is increasing together with power and energy consumption. More energy efficient hardware was developed but the overall consumption is anyway growing year by year. This high power consumption contributes to increase carbon dioxide emission in substantial manner affecting the greenhouse effect.

Nowadays is fundamental to find the right balancing between performance and power saving inside data centers. A new model of data center and activities to be allocated within was designed and, based on this model, also a resource allocation algorithm was developed in order to to find a good trade off between power consumption and performance of tasks execution. This algorithm exploits GA heuristics which allow to find optimal o closely optimal solutions ensuring good execution times.

5.2 Model

Data center is modeled with an internal three-tier fat-tree topology; servers are grouped in racks and pods: one rack contains up to 24 servers and a single pod includes up to 8 racks.

The objective is to allocate in this data center, which is empty at the beginning, a set of independent *tasks* on the available servers.

A generic tasks is characterized by:

- a number of instructions to be executed;
- a constant amount of bandwidth required to perform the execution.

Being each task independent by the others, bandwidth required by tasks is only for North-South traffic avoiding East-West communications.

Server processor is modeled like a single core processor with a fixed computational power expressed in terms of instructions per second. Tasks allocated onto the same server share equally the processor. Server power consumption model is load (or frequency) proportional; in our case we supposed that a server containing one or more tasks is maximum loaded (works at the maximum frequency) in order to execute as fast as possible all the tasks; if no tasks are allocated, server is turned off having no power consumptions. All the servers into the data center have the same characteristics in terms of computational power and power consumption.

Switches power consumption model is linear respect to the amount of used bandwidth assuming all the values between idle and peak power respectively when traffic load is 0 or 1 but, also in this case, if a switch is idle (no traffic is carried on) it's turned off in order to save more power. Power models are more detailed described in subsection 5.3.2.

To better formalize the problem, let define:

N =tasks to be allocated
M =total number of servers
S =total number of switches
R =total number of racks
T_c =maximum completion time between all the tasks
B_i =i-th task bandwidth request
P_{SRj} =power consumption of server j-th
$P_{SR} = \sum_j P_{SRj}$ = total server power consumption
P_{SWk} =power consumption of k-th switch (top of the rack or aggregation)
$P_{SW} = \sum_k P_{SWk}$ = total switch power consumption
$P = P_{SR} + P_{SW}$ =global power consumption
P_{SRP} =peak power consumption for servers
P_{SWP} =peak power consumption for switches
P_{SWI} =idle power consumption for switches
V_{Lj} =bandwidth utilization on j-th server link
V_{Tk} =bandwidth utilization on k-th top of the rack switch
IPS =number of instruction per second executed by a single processor on a server
n_j =number of tasks allocated on j-th server

To solve this allocation problem a GA was used. A single solution is represented by a N-dimensional integer array ranging between 1 and M[1]; if y_i is the i-th integer component in the solution array, this indicates that i-th task was allocated to the y_i-th server.

Operators and heuristic used by the algorithm are:

- One-cut point crossover;
- Random mutation, one or more genes of the same chromosome are randomly changed i.e. one or more task are randomly re-allocated to another servers;

[1]In the software implementation, integer value ranges between 0 and M-1

- Binary Tournament selection;
- NSGA-II search heuristic.

With these operators, all the generated chromosomes are never illegal therefore is not necessary to implement a repairing or a rejecting strategy.

Two objectives are present:

1. Minimization of the maximum completion time between all the tasks also called *makespan*;
2. Minimization of the power consumption;

The two objectives are in contrast because when one is improved the other is penalized: what we want is to find a suitable trade-off between this two objectives.

The two fitness functions for this algorithm are denoted with:

$$\begin{cases} f_1 = T_c \\ f_2 = P \end{cases}$$

but in this case, fitness functions should be minimized and not maximized.

Our problem is subject to two different constraints: it's not possible to allocate on a single server more than the link capacity (1Gbps) and more than the 20Gbps per top of the rack switches in order to avoid link congestion to both access and aggregation levels.

The two considered constraints are in this way formalized:

$$\begin{cases} V_{Lj} \leq 1Gbps & 1 \leq j \leq M \\ V_{Tk} \leq 20Gbps & 1 \leq k \leq R \end{cases}$$

In the power consumption profile we do not consider core switches because usually those switches are always turned on in data centers and it's very rare to switch off one of them. For this reason and because of the linearity of the switch power model, core switch power consumption is not considered: this value is more or less constant in every solution (due to the linearity of switch power model) and could be neglected.

5.3 Fitness functions

5.3.1 Completion Time

The total completion time is the completion time of the last task in execution. Defining:

$$\begin{cases} x_{ij} = \begin{cases} 1 \text{ if the i-th task is allocated on the j-th server;} \\ 0 \text{ otherwise;} \end{cases} \\ \#ins_i = \text{total number of instruction of task i-th;} \\ I_j = \sum_{i=1}^{N} x_{ij} \ \#ins_i = \text{total number of instructions to be executed on server j-th.} \\ T_j = \dfrac{I_j}{IPS} = \text{Completion time of tasks allocated on j-th server} \end{cases}$$

We can define the total completion time as:

$$T_c = max_{\ 1 \leq j \leq M} \ T_j$$

This formula consider the maximum of all server completion time or equivalently the completion time of the slowest task. To compute this value, each task on the same server is supposed to be executed in parallel equally sharing processor computational power. It's the same to consider one big task per server containing the sum of all the instruction of the allocated task (on the same server) and to compute that completion time.

5.3.2 Power consumption

Server power consumption is modeled with a load proportional law. Supposing that the load value of each processor could assume only two values (0 if no tasks are assigned and 1 if one ore more task are running), power consumption law for server is binary:

$$P_{SRj} = \begin{cases} P_{SRP} & \text{if } n_j \geq 1 \\ 0 & \text{if } n_j = 0 \end{cases}$$

Switch power consumption is load proportional and linear respect to the traffic between two values P_{SWP} (when all all the link are full utilized) and P_{SRI} (when the switch is idle). In our model we suppose also that, in case of load equal to 0, switched are turned off.

Defining also:

$$\begin{cases} C_i = \text{throughput of the i-th link} \\ C_{MAX} = \text{sum of maximum capacity of all uplinks} \\ \\ \#link_k = \text{number of uplinks in switch k-th} \\ l_k = \sum_{i=1}^{\#link_k} \frac{C_i}{C_{MAX}} = \text{load factor of k-th switch} \end{cases}$$

Switch power consumption law is:

$$P_{SWk}(l_k) = \begin{cases} P_{SWI} + (P_{SWP} - P_{SWI})l_k & 0 < l_k \leq 1 \\ 0 & l_k = 0 \end{cases}$$

5.3.3 Constraints

Two constraints were defined regarding the allocated bandwidth per link between:

- server and top of the rack switches;
- top of the rack switches and aggregation switches.

Denoting:

$$\left[\begin{array}{l} x_{ijk} = \begin{cases} 1 \text{ if the i-th task is allocated on the j-th server of the k-th rack;} \\ 0 \text{ otherwise;} \end{cases} \\ \\ c_{j1} = \left(\sum_{i=1}^{N} x_{ij}\ B_i - C\right) sgn\left(\sum_{i=1}^{N} x_{ij}\ B_i - C\right) \\ \\ c_{k2} = \left(\sum_{j=1}^{M}\sum_{i=1}^{N} x_{ijk}\ B_i - 20\ Gbps\right) sgn\left(\sum_{j=1}^{M}\sum_{i=1}^{N} x_{ijk}\ B_i - 20\ Gbps\right) \end{array}\right.$$

the two constraints are:

$$\begin{cases} c_1 = \sum_{j=1}^{M} c_{j1} \\ \\ c_2 = \sum_{k=1}^{R} c_{k2} \end{cases}$$

having that:

$$sgn(x) = \begin{cases} 0\ \forall x \leq 0 \\ 1\ \forall x > 0 \end{cases}$$

Defining in this way the two constraints, if links are not congested no penalty will be applied otherwise penalty value will be exactly the difference between the total required capacity and the available one. Everytime that tasks allocated on the same server (or on the same rack) require more bandwidth than the one available on links directed to the upper layer, the exceeding bandwidth is added to the constraint violations. Is important to recall that with this search heuristic, solutions that are violating constraints are still considered during the search phase but they tend to disappear in the last generations because non violating solution are always preferred.

5.4 Execution

This algorithm was executed with a different set of input values but there are some parameters which are common to each execution.
For genetic algorithm execution we have:

Parameter	Value
Population size	100
Iterations number	25000
Crossover probability	0.9
Mutation probability	$\frac{1}{\text{task number}}$

Data center topology is represented in this way:

Parameter	Value
Server per rack	24
Rack per pod	8

Tasks parameters have these values:

Parameter	Value
Average instructions per task	$7.5 * 10^8$
Average required bandwidth	$5.01 * 10^8[bps]$
Instruction distribution	uniform between [5;10]$*10^8$ instruction
Bandwidth distribution	uniform between [1;1001]$*10^8$bps

Server parameters are:

Parameter	Value
IPS	$120 * 10^6[instr/sec]$
P_{SRP}	$300[W]$

Top of rack switches power consumption values:

Parameter	Value [W]
P_{SWP}	200
P_{SWI}	160

Aggregation switches power consumption values:

Parameter	Value [W]
P_{SWP}	2500
P_{SWI}	2000

5.4.1 Experiment 1: variable task number

First experiment was executed with these input values:

Parameter	Value [W]
Server number	1536
Task number	[500:15000] with steps of 500
Same experiment repetition	3

Figure [5.1] shows the best completion time solutions obtained running three times the experiment, choosing the best three different solutions and averaging those values. The completion time trend is almost linear respect to the number of tasks and this was expected because increasing the number of tasks to be allocated, computational resources are more and more shared and global completion time grows; when task number is three-four times the server number, completion time is no more linear: the algorithm tries to find the best solutions for both the objectives and power consumption could be improved only turning off servers and switches unbalancing server load; in this situation, it's still possible to find some solutions which sacrifice completion time objective to achieve a better value of power consumption. Increasing furthermore the number of tasks, it's really hard for the algorithm to find solutions with empty servers: there are too many tasks allocated per server on average and with crossover and mutation operator only a few number of solutions are able to reach some particular kind of configurations like the one with a lot of empty servers; for the big majority of solutions, only small variations are present in power consumption objective which tends to reach the maximum limit (the sum of the maximum values of power consumption for all the devices); in this situation, completion time becomes the main objective and it grows linearly again.

Figure [5.2] shows the best values of power consumption using the same technique as before to compute best completion times. Here the curve follows a different trend

because if the number of server and switches is constant, increasing the number of tasks the final effect is to reach the limit of the sum of the peak power consumption for every devices into the data center.

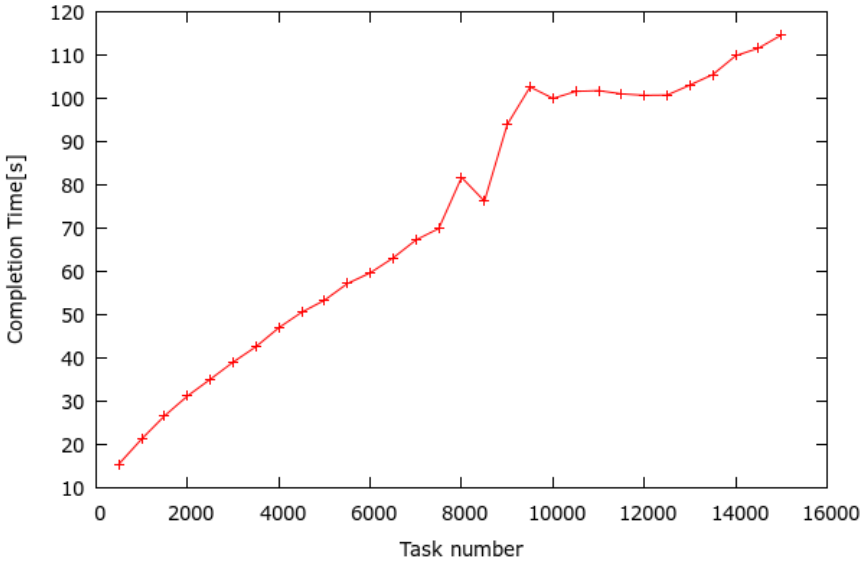

Figure 5.1: Best completion times

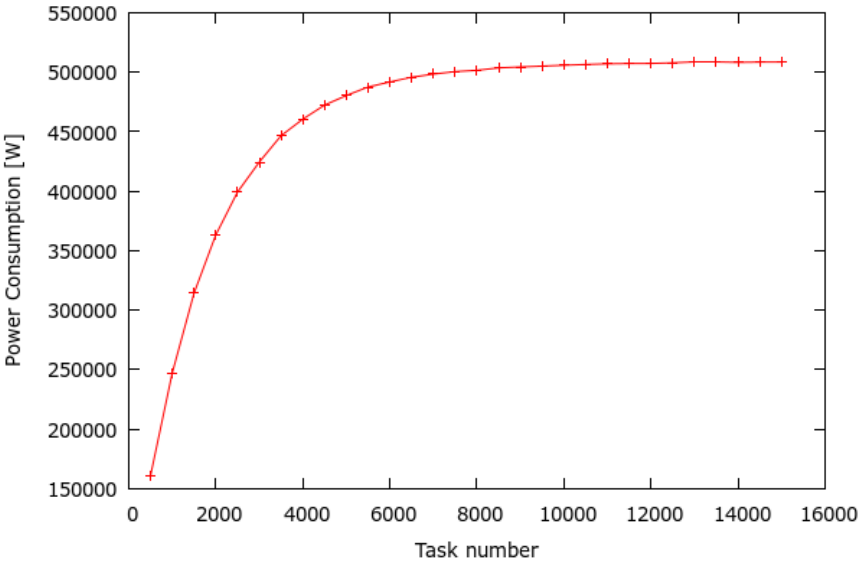

Figure 5.2: Best power consumption values

5.4.2 Experiment 2: variable server number

The second experiment was realized varying the number of server and fixing instead the number of tasks:

Parameter	Value W
Server number	[192:5760] with steps of 192 (adding always 8 racks)
Task number	3000
Same experiment repetition	3

In this situation, we would like to compare the best solutions for power consumption and completion time normalized with their average values.

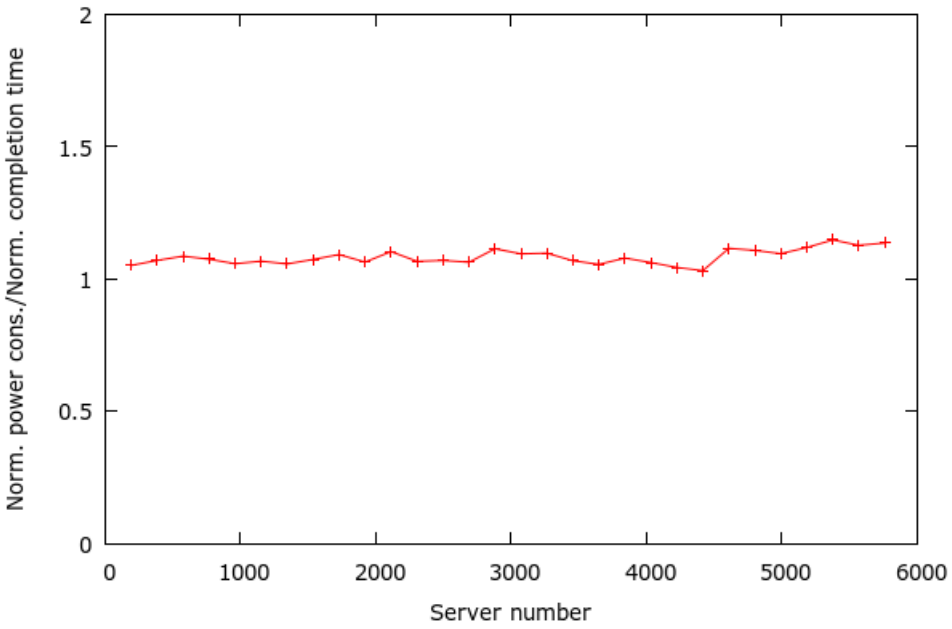

Figure 5.3: Normalized power consumption and completion time ratio

In figure [5.3] normalized values of the best solutions for power consumption and completion time are evaluated. Varying the number of servers, it is possible to note that the two objectives performs similarly because the ratio is close to 1 but increasing the number of servers, power consumption increases more than completion

time that should be more or less stable (if the number of servers grows becoming greater than the number of tasks, those could be allocated one per server minimizing the maximum completion time while some tasks can be allocated in servers belonging to different racks and/or different pods, turning on more switches and increasing the power consumption).

5.4.3 Experiment 3: Pareto-optimal solutions

This experiment shows the relation between Pareto-optimal solutions of different executions of the problem varying server numbers. In figure 5.4 it's shown one example of solution of a single instance of one specific execution obtained with these input values[2]:

Parameter	Value W
Server number	1536
Task number	3000
Same experiment repetition	3

In figure [5.4] the completion time range from around 40 seconds to 48 seconds (optimal solutions) while the power consumption ranges from 425 KW to 446 KW (again the best results). If power consumption and completion time could be considered equally relevant the vector with minimal module can be consider the best solution. On the contrary if power consumption (or vice versa completion time) would be more relevant the two metrics could be weighted to find the preferred solution.

Figure 5.5 shows the superimposition of solutions varying the number of servers; each execution is repeated three times. Some solution of the same execution could dominate the others only because they belongs to different runs of the same execution.

Increasing the number of servers, completion time decreases while power consumption increases as it was expected; between different runs of the same algorithm

[2]remember that we will not find a single solution but a set of suitable solutions corresponding to the search for the optimum

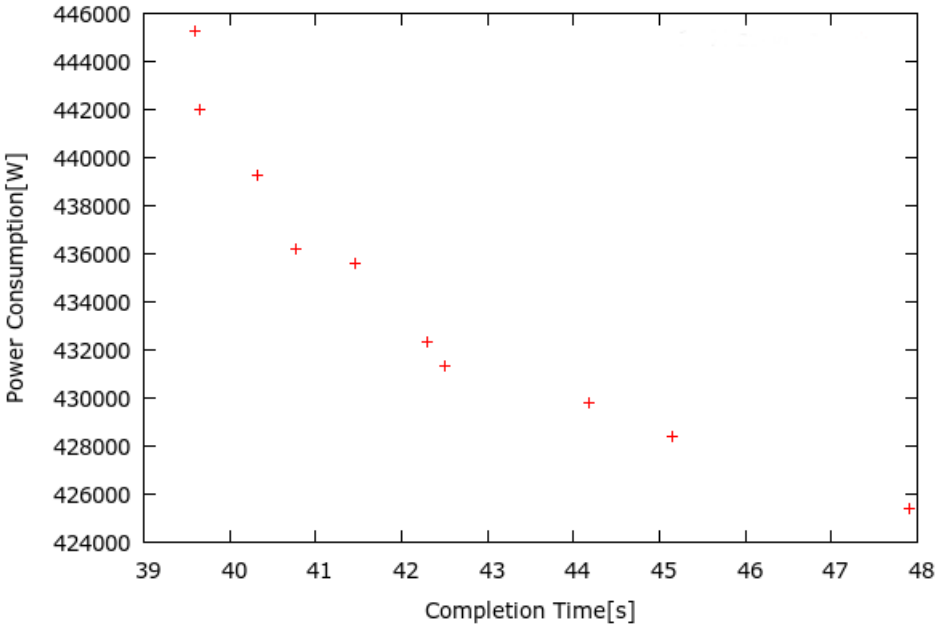

Figure 5.4: Pareto-front obtained by a single experiment

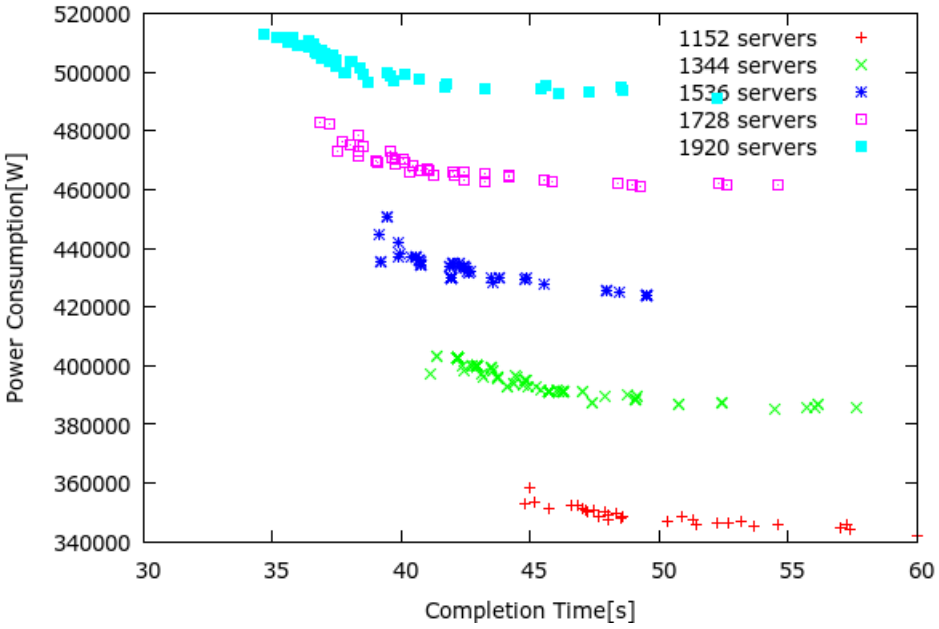

Figure 5.5: Solution with different number of servers

there are only slight differences, in this way different executions of the algorithm converge to the same solution set showing a certain degree of stability and reliability.

5.4.4 Experiment 4: algorithm execution time

Last experiment measures the performance of the framework to compute the solution with a fixed number of servers (1536) varying the task number. Figure 5.6 shows the results with a trend line. Points are close to the cubic trend function that fits

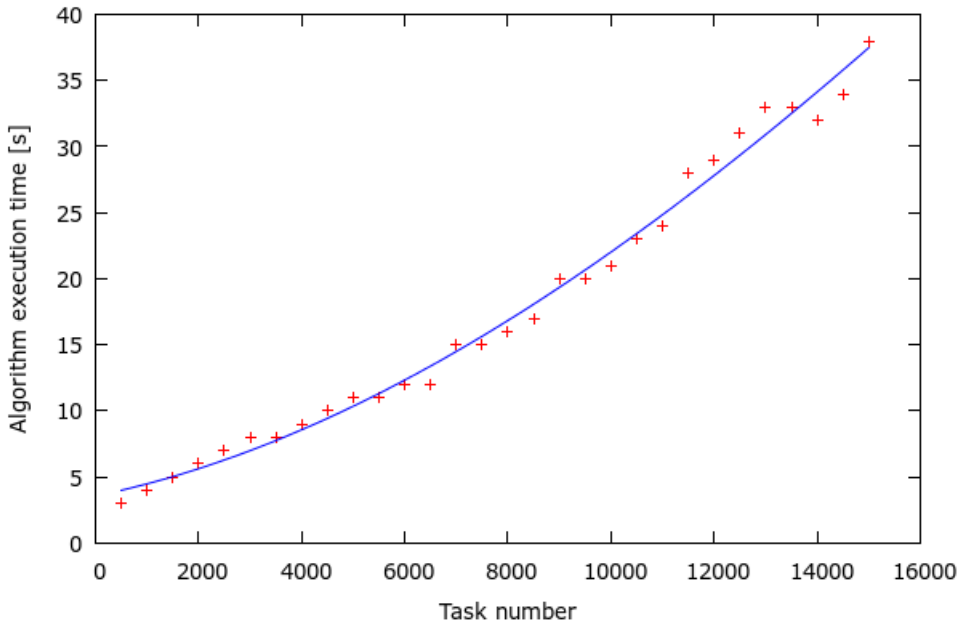

Figure 5.6: Execution time of the algorithm

well the measured values then the algorithm in the considered range of values has a cubic dependence from the number of input tasks.

5.4.5 Other approaches

5.4.6 VMPlanner

VMPlanner is a network-wide power manager that optimizes VM placement and traffic flow routing to reduce data center power costs by sleep scheduling of network

elements [13]. In this approach, data center network topology is modeled as a simple undirected graph with two types of vertices, servers and switches, and edge which represents a communication link between a server and a switch or between a pair of switches. VMPlanner solves the VM allocation problem with three different algorithms:

1. Traffic-aware VM grouping, which collects VMs into a set of VM-groups so that the overall inter-group traffic volume is minimized while the overall intra-group traffic volume is maximized; this algorithm is approximated and has a complexity $O(X^4)$ with X the VMs number.

2. Distance-aware VM-group to server-rack mapping, which optimally assigns VM-groups to servers and racks such that the mapping minimizes the total inter-rack traffic load in the network; this kind of algorithm falls into the class of Quadratic Assignment Problem (QAP) and exact algorithms are still not practical to solve QAP problems with rack number > 20;

3. Power-aware inter-VM traffic flow routing, that moves and aggregates network traffic onto a fewer number of paths so as to put the remaining network elements into dormant state for energy conservation; this algorithms scales up only for networks with small size having complexity $O(Y^{2.5})$ with Y number of switches.

This work solves this problem only with a combinatorial approach without any kind of information about VM computational requirements and mainly focusing only on the network part.

5.4.7 Energy Aware Scheduling

In this work, authors developed a particular kind of Service Level Agreement (SLA) called Green SLA [14]. Task scheduling is based on specified task execution time, CO2 emission and power consumption through energy aware schedulers. There are two kinds of scheduling in this approach:

- best effort, which minimizes task execution time;

- energy-performance trade off, based on the green SLA; users accepts a tolerable performance loss, for example, additional 10% of task execution time, to reduce more energy consumption and make their computing more green.

This approach implements some advanced power consumption strategies like Dynamic voltage and frequency scaling (DVFS) and supports parallel activities but doesn't consider the network part at all.

Chapter 6

Conclusions

Resource allocation in data centers will be necessarly related to the problem of finding the best performances reducing in the meanwhile the power consumption of the whole infrastructure. For this reason, an algorithm was developed to find the right allocation for groups of tasks in order to find the best compromise as possible. The developed algorithm finds a set of nondominated solution in this multi-objective computation. When the execution of the algorithm is completed and the Pareto Optimal solutions are retrieved, is possible to choose the required trade off between power consumption and execution time with fine tuning approaches that could be done upon the obtained solutions choosing the proper tasks allocation, privileging sometimes one objective or the other according to the policy that would be adopted.

Algorithm execution time reaches good values having a cubic dependency on the number of tasks to be allocated.

Genetic heuristics are able to find good solutions exploring the whole solution space focusing on the optimal or almost optimal ones while another approaches typical of operations research (adopted for example by VMPlanner) lead to higher complexities and to approximated solutions that for sure don't reach the optimum. This algorithm represents a first approach of resource allocation exploiting SDN functionalities which allow to jointly allocate computational resources and network resources for tasks or VMs in data centers. The main contribution of this thesis is that the developed algorithm considers computational and network requirements

(differentiating tasks betweem themselves with a more complete model respect for example to VMPlanner), power consumption of servers and network devices and exploits network resource allocation provided by the SDN architecture, with the possibility to be executed directly on SDN controller.

6.1 Future works

For future works, it's possible to adjust the model or implement new features:

- as first step, East - West traffic could be considered introducing dependencies between tasks;
- support for new data center topologies could be implemented;
- algorithm could be improved to achieve also inter-data center scheduling of tasks, considering more than one data center and adding new parameters like electricity cost and data center load factor;
- another important extention could be the *dynamic task allocation*: thanks to the informations provided by the SDN controller (which is able to retrieve informations about the allocated flows in each switch) it could possible to find strategies to reallocate tasks which are already present or to allocate new ones on-the-fly.

This work provides also the basis for a new kind of scheduling looking towards a cognitive perspective: tasks could be allocated using a cognitive scheduler following a kind of artificial intelligence able to choose the best allocation patterns based on the collected parameters on server and network status; Software Defined Networking could assist the cognitive scheduler thanks to its flexibility and the global network view available for all the network devices providing the needed informations, allocating the tasks to the right server and configuring switches and network flows in the proper way.

Appendix A

Source Code

In the following sections is included the most important part of the source code of the algorithm. Problem, Crossover, Mutation and Main classes were designed *ad hoc*. Algorithm and Selection classes belong to the jMetal framework already implemented classes.

A.1 Problem

This is the problem implementation: here fitness functions and constraints are defined.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import jmetal.core.Problem;
import jmetal.core.Solution;
import jmetal.core.Variable;
import jmetal.encodings.solutionType.IntSolutionType;
import jmetal.encodings.variable.Int;
import jmetal.util.JMException;

public class DynamicAllocationProblem extends Problem {

  private  int TASK_NUM; // Number of tasks present in
     the system
```

```
public static int SERV_NUM; // Number of servers
private static int SERV_ON_RACK; // Number of servers
   in a rack
private static int RACK_NUM; // Number of racks
private static int RACK_ON_POD; // Number of racks in a
    pod
private static int POD_NUM; // Number of pods
private static double CPU_CAPACITY; // CPU
   computational power [instruction/second]
private double P_S_IDLE = 0; // Fixed amount of power
   used by CPUs [W]
private double P_S_PEAK = 300; // Peak power of a
   generic CPU [W]
private double P_TS_PEAK=200; // Peak power of a top of
    rack switch [W]
private double P_TS_IDLE = 0.80* P_TS_PEAK; // Fixed
   amount of power used by top of rack switches [W]
private double P_AGG_PEAK= 2500; // Peak power of an
   aggregation switch [W]
private double P_AGG_IDLE=0.80*P_AGG_PEAK; // Idle
   aggregation switch power consumption [W]

private double SERVER_LINK_CAPACITY = 1E9; // Single
   link Server - ToR switch capacity [bits/sec]
private double RACK_LINK_CAPACITY = 20e9; // Single
   link ToR - Aggregation switch capacity [bits/sec]

private ArrayList<Task> task; // List of tasks to be
   allocated

private static final long serialVersionUID = 1L;

@SuppressWarnings("unchecked")
public DynamicAllocationProblem(HashMap<String, Object>
    parameters) {

  numberOfObjectives_ = 2;
  numberOfConstraints_ = 2;
  problemName_ = "Dynamic Allocation";
  solutionType_ = new IntSolutionType(this);
  SERV_NUM = (int) parameters.get("serverNumber");
  SERV_ON_RACK = (int) parameters.get("serverPerRack");
  RACK_ON_POD=(int)parameters.get("rackPerPod");
  RACK_NUM = (SERV_NUM / SERV_ON_RACK)
      + (SERV_NUM % SERV_ON_RACK == 0 ? 0 : 1);
```

```
    POD_NUM=(RACK_NUM / RACK_ON_POD) +(RACK_NUM %
       RACK_ON_POD==0?0:1);
    task = (ArrayList<Task>) parameters.get("task");
    TASK_NUM = task.size();
    numberOfVariables_ = TASK_NUM;
    upperLimit_ = new double[numberOfVariables_];
    lowerLimit_ = new double[numberOfVariables_];
    for (int i = 0; i < numberOfVariables_; ++i) {
      upperLimit_[i] = SERV_NUM - 1;
      lowerLimit_[i] = 0;
    }
  }

  public int sgn(double x) {
    return x > 0 ? 1 : 0;
  }

  @Override
  public void evaluate(Solution solution) throws
     JMException{
    int task_per_server[] = new int[SERV_NUM];
    int task_per_rack[]= new int[RACK_NUM];
    int task_per_pod[]=new int[POD_NUM];
    double instruction_per_server[] = new double[SERV_NUM
       ];
    double bandwidth_per_server[] = new double[SERV_NUM];
    double bandwidth_per_rack[] = new double[RACK_NUM];
    double bandwidth_per_pod[] = new double[POD_NUM];
    Variable[] var = solution.getDecisionVariables();

    for (int i = 0; i < numberOfVariables_; ++i) {
      Int server = (Int) var[i];
      int server_index = (int) server.getValue();
      task_per_server[server_index]++;
      task_per_rack[server_index/SERV_ON_RACK]++;
      task_per_pod[(server_index/SERV_ON_RACK)/
         RACK_ON_POD]++;
      instruction_per_server[server_index] += task.get(i)
          .getInstructionNumber();
      bandwidth_per_server[server_index] += task.get(i).
         getBandwidth();
    }
    // Fitness function 1;
    double total_completion_time = 0;
    for (int i = 0; i < SERV_NUM; ++i) {
```

```
    double server_completion_time =
       instruction_per_server[i]
        / (CPU_CAPACITY);
    total_completion_time = total_completion_time >
       server_completion_time ? total_completion_time
        : server_completion_time;

  }
  // Fitness function 2;
  double server_power_consumption = 0;
  double switch_power_consumption = 0;

  for (int i = 0; i < SERV_NUM; ++i) {
    server_power_consumption += (task_per_server[i] ==
       0 ? P_S_IDLE : P_S_PEAK);
    //Sums server allocated bandwidth if it less than
       the maximum link capacity
    //otherwise the last one is considered.
    bandwidth_per_rack[i / SERV_ON_RACK] += (
       bandwidth_per_server[i] <= SERVER_LINK_CAPACITY
       ? bandwidth_per_server[i]
        : SERVER_LINK_CAPACITY);
  }
  for (int i = 0; i < RACK_NUM; ++i) {
  //If the offered rack capacity is greater than the 20
      Gbps, than only 20 Gbps are provided to the upper
      layer
    bandwidth_per_pod[i / RACK_ON_POD] += (
       bandwidth_per_rack[i] <= RACK_LINK_CAPACITY ?
       bandwidth_per_rack[i]
        : RACK_LINK_CAPACITY);
  }

  double tor_switch_power_consumption = 0;
  for (int i = 0; i < RACK_NUM; ++i) {
    if(task_per_rack[i]!=0){
    tor_switch_power_consumption += P_TS_IDLE
        + (P_TS_PEAK - P_TS_IDLE)
        * (bandwidth_per_rack[i] / (SERV_ON_RACK *
           SERVER_LINK_CAPACITY));
    }
  }

  double agg_switch_power_consumption = 0;
  for (int i = 0; i < POD_NUM; ++i) {
```

```
        if(task_per_pod[i]!=0){
        agg_switch_power_consumption += 2*P_AGG_IDLE
            + (P_AGG_PEAK - P_AGG_IDLE)
            * (bandwidth_per_pod[i] / (RACK_ON_POD *
               RACK_LINK_CAPACITY));
        }
      }

      switch_power_consumption =
         tor_switch_power_consumption
          + agg_switch_power_consumption;

      solution.setObjective(0, total_completion_time);
      solution.setObjective(1, server_power_consumption
          + switch_power_consumption);

    }

    public void evaluateConstraints(Solution solution)
       throws JMException {
      double bandwidth_per_server[] = new double[SERV_NUM];
      double bandwidth_per_rack[] = new double[RACK_NUM];
      double bandwidth_per_pod[] = new double[POD_NUM];

      // Constraint evaluations;
      Variable[] var = solution.getDecisionVariables();
      // System.out.println(var.length);
      for (int i = 0; i < var.length; ++i) {
        Int server = (Int) var[i];
        int server_index = (int) server.getValue();
        bandwidth_per_server[server_index] += task.get(i).
           getBandwidth();
      }
      for (int i = 0; i < SERV_NUM; ++i) {

        bandwidth_per_rack[i / SERV_ON_RACK] +=
           bandwidth_per_server[i];
      }
      for (int i = 0; i < RACK_NUM; ++i) {
        bandwidth_per_pod[i / RACK_ON_POD] +=
           bandwidth_per_rack[i];
      }

      //First constraint
      double server_constraint = 0;
```

```
    for (int i = 0; i < SERV_NUM; ++i) {
      server_constraint += (bandwidth_per_server[i] -
         SERVER_LINK_CAPACITY)
          * sgn(bandwidth_per_server[i] -
             SERVER_LINK_CAPACITY);
    }

    //Second constraint
    double rack_constraint = 0;
    for (int i = 0; i < RACK_NUM; ++i) {
      rack_constraint += (bandwidth_per_rack[i] -
         RACK_LINK_CAPACITY)
          * sgn(bandwidth_per_rack[i] -
             RACK_LINK_CAPACITY);
    }
    int number_violated_constraint = 0;
    if (server_constraint > 0)
      number_violated_constraint++;
    if (rack_constraint > 0)
      number_violated_constraint++;

    solution.setOverallConstraintViolation(
       rack_constraint
        + server_constraint);
    solution.setNumberOfViolatedConstraint(
       number_violated_constraint);

  }
}
```

A.2 Operators

A.2.1 Crossover

```
package jmetal.operator.crossover;

import java.util.HashMap;
import jmetal.util.Configuration;
import jmetal.core.Variable;
import jmetal.core.Operator;
import jmetal.core.Solution;
import jmetal.util.JMException;
import jmetal.util.PseudoRandom;

/*
 * This class allows to apply a One Point crossover
    operator using two parent
 * solutions.
 *
 */
public class UniformCrossover extends Operator {

  private static final long serialVersionUID =
     7679206687912422515L;
  private double probability;

  public UniformCrossover(HashMap<String,Object>
     parameters)throws JMException{
    super(null);
    if(!parameters.containsKey("crossoverProbability"))
       throw new JMException("Missing
       crossoverProbability");
    probability=(double)parameters.get("
       crossoverProbability");
  } // OnePointUniformCrossover


  public Solution[] doCrossover( double probability ,
                                 Solution parent1   ,
                                 Solution parent2   )
                                    throws JMException
                                    {
    Solution [] offSpring = new Solution[2];
```

```
    offSpring[0] = new Solution( parent1 );
    offSpring[1] = new Solution( parent2 );

    try {
      if (PseudoRandom.randDouble() < probability) {
        int len = offSpring[0].numberOfVariables();

        Variable[] vars1 = offSpring[0].
           getDecisionVariables();
        Variable[] vars2 = offSpring[1].
           getDecisionVariables();

        int half     = len>>1;
        int crossPnt = PseudoRandom.randInt( 0 , half );

        Variable[] buffer = new Variable[ half ];
        System.arraycopy( vars1  , crossPnt , buffer , 0
                  , half );
        System.arraycopy( vars2  , crossPnt , vars1  ,
           crossPnt , half );
        System.arraycopy( buffer , 0        , vars2  ,
           crossPnt , half );

      } // if
    } // try
    catch (ClassCastException e1) {
      Configuration.logger_.severe("
         OnePointUniformCrossover.doCrossover: Cannot
         perfom " +
      "OnePointUniformCrossover");
      throw new JMException("Exception in
         OnePointUniformCrossover.doCrossover()") ;
    } // catch
    return offSpring;
  } // doCrossover


  public Object execute( Object object ) throws
     JMException {
    Solution[] parents = (Solution []) object;



    if (parents.length < 2) {
```

```
        Configuration.logger_.severe("
            OnePointUniformCrossover.execute: operator " +
        "needs two parents");
        throw new JMException("Exception in
            OnePointUniformCrossover.execute()") ;
      } // if

      Solution [] offSpring;
      offSpring = doCrossover( probability , parents[0],
         parents[1] );

      for (int i = 0; i < offSpring.length; i++) {
        offSpring[i].setCrowdingDistance(0.0);
        offSpring[i].setRank(0);
      } // for
      return offSpring;
    } // execute

  } // OnePointUniformCrossover
```

A.2.2 Mutation

```
import java.util.HashMap;
import java.util.Random;
import jmetal.core.Operator;
import jmetal.core.Solution;
import jmetal.core.Variable;
import jmetal.util.JMException;

public class MyRebalanceMutation extends Operator {

  private static final long serialVersionUID =
     -7403795178769425557L;
  private int numberOfTasks_ = 0;
  private int numberOfMachines_ = 0; // the server number
  private int rounds_ = 2;
  private double probability_;

  public MyRebalanceMutation(HashMap<String, Object>
     parameters) throws JMException{
    super(parameters);
    if(!parameters.containsKey("mutationProbability")){
      throw new JMException("Mutation Probability is
         missing");
    }
    probability_=(double)parameters.get("
       mutationProbability");
    if(!parameters.containsKey("serverNumber")){
      throw new JMException("Number of server is missing"
         );
    }
    numberOfMachines_=(int)parameters.get("serverNumber")
       ;

  }
  private void doMutation(Solution solution) throws
     JMException{
    Variable [] var = solution.getDecisionVariables();
    for(int i=0;i<var.length;++i){
      Random r=new Random();
      double prob=r.nextDouble();
      if(prob<=probability_){
        Random t=new Random();
        var[i].setValue(t.nextInt(numberOfMachines_));
```

```
        }
      }
      return;
    }

    @Override
    public Object execute(Object object) throws JMException
        {
      Solution solution = (Solution)object;
      double rand = Math.random();
      if ( rand <= probability_ )
        doMutation( solution );
      return null;
    }

  }
```

A.2.3 Selection

This is the Binary Tournament implementation already present in jMetal framework[2].

```
package jmetal.operators.selection;

import jmetal.core.Solution;
import jmetal.core.SolutionSet;
import jmetal.util.PseudoRandom;
import jmetal.util.comparators.DominanceComparator;

import java.util.Comparator;
import java.util.HashMap;

/**
 * This class implements an binary tournament selection
    operator
 */
public class BinaryTournament extends Selection {

  /**
   * Stores the <code>Comparator</code> used to compare
      two
   * solutions
   */
  private Comparator comparator_;

  /**
   * Constructor
   * Creates a new Binary tournament operator using a
      BinaryTournamentComparator
   */
  public BinaryTournament(HashMap<String, Object>
     parameters){
    super(parameters) ;
    if ((parameters != null) && (parameters.get("
       comparator") != null))
      comparator_ = (Comparator) parameters.get("
         comparator") ;
    else
      //comparator_ = new BinaryTournamentComparator();
      comparator_ = new DominanceComparator();
  } // BinaryTournament

  /**
```

```
   * Performs the operation
   * @param object Object representing a SolutionSet
   * @return the selected solution
   */
   public Object execute(Object object){
     SolutionSet solutionSet = (SolutionSet)object;
     Solution solution1, solution2;
     solution1 = solutionSet.get(PseudoRandom.randInt(0,
        solutionSet.size()-1));
     solution2 = solutionSet.get(PseudoRandom.randInt(0,
        solutionSet.size()-1));

     if (solutionSet.size() >= 2)
       while (solution1 == solution2)
         solution2 = solutionSet.get(PseudoRandom.randInt
            (0,solutionSet.size()-1));

     int flag = comparator_.compare(solution1,solution2);
     if (flag == -1)
       return solution1;
     else if (flag == 1)
       return solution2;
     else
       if (PseudoRandom.randDouble()<0.5)
         return solution1;
       else
         return solution2;
   } // execute
 } // BinaryTournament
```

A.3 Algorithm

This is the NSGA-II implementation present in jMetal Framework[2].

```
/  NSGAII.java
//
//  Author:
//       Antonio J. Nebro <antonio@lcc.uma.es>
//       Juan J. Durillo <durillo@lcc.uma.es>
//
//  Copyright (c) 2011 Antonio J. Nebro, Juan J. Durillo
//
//  This program is free software: you can redistribute
   it and/or modify
//  it under the terms of the GNU Lesser General Public
   License as published by
//  the Free Software Foundation, either version 3 of the
    License, or
//  (at your option) any later version.
//

package jmetal.metaheuristics.nsgaII;

import jmetal.core.*;
import jmetal.qualityIndicator.QualityIndicator;
import jmetal.util.Distance;
import jmetal.util.JMException;
import jmetal.util.Ranking;
import jmetal.util.comparators.CrowdingComparator;

/**
 *  Implementation of NSGA-II.
 */

public class NSGAII extends Algorithm {
  /**
   * Constructor
   * @param problem Problem to solve
   */
  public NSGAII(Problem problem) {
    super (problem) ;
  } // NSGAII

  /**
   * Runs the NSGA-II algorithm.
```

```
   * @return a <code>SolutionSet</code> that is a set of
      non dominated solutions
   * as a result of the algorithm execution
   * @throws JMException
   */
  public SolutionSet execute() throws JMException,
     ClassNotFoundException {
    int populationSize;
    int maxEvaluations;
    int evaluations;

    QualityIndicator indicators; // QualityIndicator
       object
    int requiredEvaluations; // Use in the example of use
        of the
    // indicators object (see below)

    SolutionSet population;
    SolutionSet offspringPopulation;
    SolutionSet union;

    Operator mutationOperator;
    Operator crossoverOperator;
    Operator selectionOperator;

    Distance distance = new Distance();

    //Read the parameters
    populationSize = ((Integer) getInputParameter("
       populationSize")).intValue();
    maxEvaluations = ((Integer) getInputParameter("
       maxEvaluations")).intValue();
    indicators = (QualityIndicator) getInputParameter("
       indicators");

    //Initialize the variables
    population = new SolutionSet(populationSize);
    evaluations = 0;

    requiredEvaluations = 0;

    //Read the operators
    mutationOperator = operators_.get("mutation");
    crossoverOperator = operators_.get("crossover");
    selectionOperator = operators_.get("selection");
```

```
// Create the initial solutionSet
Solution newSolution;
for (int i = 0; i < populationSize; i++) {
  newSolution = new Solution(problem_);
  problem_.evaluate(newSolution);
  problem_.evaluateConstraints(newSolution);
  evaluations++;
  population.add(newSolution);
} //for

// Generations
while (evaluations < maxEvaluations) {

  // Create the offSpring solutionSet
  offspringPopulation = new SolutionSet(
     populationSize);
  Solution[] parents = new Solution[2];
  for (int i = 0; i < (populationSize / 2); i++) {
    if (evaluations < maxEvaluations) {
      //obtain parents
      parents[0] = (Solution) selectionOperator.
         execute(population);
      parents[1] = (Solution) selectionOperator.
         execute(population);
      Solution[] offSpring = (Solution[])
         crossoverOperator.execute(parents);
      mutationOperator.execute(offSpring[0]);
      mutationOperator.execute(offSpring[1]);
      problem_.evaluate(offSpring[0]);
      problem_.evaluateConstraints(offSpring[0]);
      problem_.evaluate(offSpring[1]);
      problem_.evaluateConstraints(offSpring[1]);
      offspringPopulation.add(offSpring[0]);
      offspringPopulation.add(offSpring[1]);
      evaluations += 2;
    } // if
  } // for

  // Create the solutionSet union of solutionSet and
     offSpring
  union = ((SolutionSet) population).union(
     offspringPopulation);

  // Ranking the union
```

```
Ranking ranking = new Ranking(union);

int remain = populationSize;
int index = 0;
SolutionSet front = null;
population.clear();

// Obtain the next front
front = ranking.getSubfront(index);

while ((remain > 0) && (remain >= front.size())) {
  //Assign crowding distance to individuals
  distance.crowdingDistanceAssignment(front,
     problem_.getNumberOfObjectives());
  //Add the individuals of this front
  for (int k = 0; k < front.size(); k++) {
    population.add(front.get(k));
  } // for

  //Decrement remain
  remain = remain - front.size();

  //Obtain the next front
  index++;
  if (remain > 0) {
    front = ranking.getSubfront(index);
  } // if
} // while

// Remain is less than front(index).size, insert
   only the best one
if (remain > 0) {  // front contains individuals to
    insert
  distance.crowdingDistanceAssignment(front,
     problem_.getNumberOfObjectives());
  front.sort(new CrowdingComparator());
  for (int k = 0; k < remain; k++) {
    population.add(front.get(k));
  } // for

  remain = 0;
} // if

// This piece of code shows how to use the
   indicator object into the code
```

```
        // of NSGA-II. In particular, it finds the number
           of evaluations required
        // by the algorithm to obtain a Pareto front with a
            hypervolume higher
        // than the hypervolume of the true Pareto front.
        if ((indicators != null) &&
            (requiredEvaluations == 0)) {
          double HV = indicators.getHypervolume(population)
              ;
          if (HV >= (0.98 * indicators.
              getTrueParetoFrontHypervolume())) {
            requiredEvaluations = evaluations;
          } // if
        } // if
      } // while

      // Return as output parameter the required
         evaluations
      setOutputParameter("evaluations", requiredEvaluations
          );

      // Return the first non-dominated front
      Ranking ranking = new Ranking(population);
      ranking.getSubfront(0).printFeasibleFUN("FUN_NSGAII")
          ;

      return ranking.getSubfront(0);
    } // execute
  } // NSGA-II
```

A.4 Main

This is the main class of the algorithm: it instantiates the other classes, sets all the parameters and launches the framework execution.

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Random;
import java.util.logging.FileHandler;
import jmetal.core.Algorithm;
import jmetal.core.Operator;
import jmetal.core.Problem;
import jmetal.core.Solution;
import jmetal.core.SolutionSet;
import jmetal.core.Variable;
import jmetal.encodings.variable.Int;
import jmetal.metaheuristics.mocell.MOCell;
import jmetal.metaheuristics.nsgaII.NSGAII;
import jmetal.metaheuristics.nsgaII.NSGAII_main;
import jmetal.operator.crossover.UniformCrossover;
import jmetal.operators.selection.SelectionFactory;
import jmetal.util.Configuration;
import jmetal.util.JMException;
import jmetal.util.NonDominatedSolutionList;

public class NSGAII_main_new extends NSGAII_main {
  public static Algorithm setup(Problem problem){
    Algorithm algorithm;
      algorithm = new NSGAII(problem);
      algorithm.setInputParameter("populationSize", 1000)
         ;
      algorithm.setInputParameter("maxEvaluations",
         25000);
    return algorithm;
  }
  //Task generation
  public static ArrayList<Task> taskGenerator(int server,
      int task_num){

    ArrayList<Task> list = new ArrayList<Task>();
    for (int i = 0; i < task_num; ++i) {
      Random r=new Random();
```

```
      double instruction =r.nextDouble()*500E6+500E6;;
      double bandwidth=r.nextDouble()*1E9+1E6;
      list.add(new Task(instruction, bandwidth));
    }
    return list;

  }

  @SuppressWarnings("unchecked")
  public static void main(String args[]) throws
     JMException,
      SecurityException, IOException,
         ClassNotFoundException {
    //ARGS[0] TASK NUM
    //ARGS[1] SERVER NUM
    //ARGS[2] SERV_PER_RACK NUM
    //ARGS[3] SERV_PER_POD NUM
    Problem problem; // The problem to solve
    Algorithm algorithm; // The algorithm to use
    Operator crossover; // Crossover operator
    Operator mutation; // Mutation operator
    Operator selection; // Selection operator
    HashMap<String,Object> parameters; // Operator
       parameters
    // Logger object and file to store log messages
    logger_ = Configuration.logger_;
    fileHandler_ = new FileHandler("NSGAII_main.log");
    logger_.addHandler(fileHandler_);
    int task_num=Integer.parseInt(args[0]);

    int serverNumber=Integer.parseInt(args[1]);;
    int serverPerRack=Integer.parseInt(args[2]);
    int rackPerPod=Integer.parseInt(args[3]);
    ArrayList <Task>list=taskGenerator(serverNumber,
       task_num);

    // Problem
    parameters = new HashMap<String,Object>();
    parameters.put("serverNumber", serverNumber);
    parameters.put("task", list);
    parameters.put("serverPerRack",serverPerRack);
    parameters.put("rackPerPod", rackPerPod);
    problem = new DynamicAllocationProblem(parameters);

    //Algorithm
```

```
        algorithm=setup(problem);
        parameters.put("crossoverProbability", 0.9);
        //AllocationComparator is used to compare different
           solutions
        parameters.put("comparator", new AllocationComparator
           ());
        crossover = new UniformCrossover(parameters);
        //Mutation Probability
        parameters.put("mutationProbability", 1.0 / problem.
           getNumberOfVariables());
        mutation=new MyRebalanceMutation(parameters);

        // Selection Operator
        parameters = null;
        //This method creates the selection operator through
           the provided parameters
        selection = SelectionFactory.getSelectionOperator("
           BinaryTournament",
            parameters);

        // Add the operators to the algorithm
        algorithm.addOperator("crossover", crossover);
        algorithm.addOperator("mutation", mutation);
        algorithm.addOperator("selection", selection);

        // Execute the Algorithm
        long initTime = System.currentTimeMillis();
        SolutionSet population = algorithm.execute();
        long estimatedTime = System.currentTimeMillis() -
           initTime;
        // Result messages
        logger_.info("Total execution time: " + estimatedTime
            + "ms");
        logger_.info("Variables values have been writen to
           file VAR");
        population.printVariablesToFile("VAR");
        logger_.info("Objectives values have been writen to
           file FUN");
        population.printObjectivesToFile("FUN");

    }

}
```

Bibliography

[1] Gen, Cheng, Lin, "Network Models and Optimization, Multiobjective Genetic Algorithm Approach", Springer, 2008

[2] Durillo, Nebro, "JMetal: A Java framework for multi-objective optimization", Advances in Engineering Software, Elsevier, October 2011

[3] Kliazovich, Granelli et al, "Energy-Efficient Data Replication in Cloud Computing Datacenters", Globecom 2013 Workshop

[4] Guo, Wu et al, "DCell: "A Scalable and Fault-Tolerant Network Structure for Data Centers", Sigcomm 2008

[5] Mysore, Pamboris et al, "Portland: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric", Sigcomm 2009

[6] Deb, Pratap et al, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II", IEEE Transactions on evolutionary computation, April 2012

[7] Raquel, Naval, "An Effective Use of Crowding Distance in Multiobjective Particle Swarm Optimization", Genetic and Evolutionary Computation Conference 2005

[8] Open Networking Foundation, "Software-Defined Networking:The New Norm for Networks", April 2012

[9] McKeown, Anderson et al, "Openflow: enabling innovation in campus networks", ACM SIGCOMM Computer Communication Review, 2008.

[10] Mininet: An Instant Virtual Network on your Laptop (or other PC) www.mininet.org

[11] Mendonca et al, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks", May 2013

[12] Beloglazov, Buyya, "Energy Efficient Resource Management in Virtualized Cloud Data Centers", 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010

[13] Fang, Liang et al, "VMPlanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers", Computer Networks, January 2013

[14] Wang, Laszewski et al, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS", 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010

[15] Greenberg, Hamilton et al, "The Cost of a Cloud: Research Problems in Data Center Networks", January 2009