University of Lleida

Polytechnic School

Degree in Computer Engineering

Final project report

DGGA
Distributed Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms

Author: Josep Pon Farreny

Director: Carlos Ansótegui
Co-director: Kevin Tierney

Lleida - September 8, 2014

Acknowledgements

I would like to thank all the people that have helped me during this project. First of all, I want to thank my director, Carlos Ansótegui, for guiding me during this project, his constant support and the time he has spent resolving my doubts, even on weekends and vacations. Secondly, to my co-director, Kevin Tierney, for the interest he has shown and for clarifying me any obscure detail of the implementation of *GGA*. Thirdly, I would like to offer my gratitude to Fernando Cores, who provided valuable and constructive comments.

I also want to thank those close to me, my family who has had to bear with my irritable mood, my girlfriend who, in addition, has given me her support and reminded me that there was more in life than this project, and an special mention to my uncle for his support, even when he was living a difficult situation.

Finally, even though these lines do not make sense for him, I would like to thank Puc, who, no matter how I felt, has constantly put a smile on my face making each day a little bit better.

Puc

Contents

Chapter 1

Introduction

Combinatorial optimization problems arise in many domains: scheduling and planning, software and hardware verification, knowledge compilation, probabilistic modelling, bioinformatics, energy systems, smart cities, social networks, computational sustainability, etc. From a computational point of view, many optimization problems are NP-hard meaning that is unlikely that they admit a polynomial-time algorithm. The good news is that some real problems are already efficiently solved by state-of-the-art Constraint Programming algorithms [1] and many others are only slightly beyond the reach of these algorithms.

There are different algorithms with complementary strengths and weaknesses, which additionally expose parameters that need to be tuned for peak performance. Not considering these issues leads to lost performance in industrial and academic applications. For example, from the Satisfiability (SAT) competitions that take place every year since 2002, we have learnt that no solver dominates over all the instances. Therefore, it seems reasonable to have a pool of SAT solvers, and given a SAT instance try to predict their expected running time in order to choose the best candidate. This is known as the algorithm selection problem, which consists of choosing the best algorithm from a predefined set, to run on a problem instance [2]. Algorithm portfolios tackle this problem. Portfolios have been shown to be very successful in SAT [3] and CP [4]. The first successful algorithm portfolio for SAT was exploited by SATzilla 2007 [3].

A related topic to the algorithm selection problem, tackled by portfolios, is the automatic configuration problem, which consist of choosing the best configuration of the parameters of a given algorithm to run on a problem instance. We will refer also to this process as the tuning of a solver.

Automatic configuration has been shown to be a key piece of the puzzle to come up with efficient solvers. For example, it has been observed that SAT solvers can exhibit a quite different performance depending on how they are tuned (parametrized) [5,6]. Indeed, we can see every possible configuration of a solver as a new solver. Recently, the techniques developed for solving the automatic configuration problem have been integrated to create instance-specific tuners. For example, Hydra [4] the parameter tuner ParamILS [5] and ISAC [7] uses the parameter tuner *GGA* [6].

From a practical point of view, algorithm configuration has already achieved dramatic speed-ups of state-of-the-art algorithms for formal verification and mixed integer programming on various benchmarks of industrial relevance (e.g. hardware verification and process planning & optimization). For example, it has been successfully applied to improve the performance of IBM CPLEX, a commercial solver used widely in industry.

It is natural to think about the algorithm configuration problem as a search problem over the space of possible configurations of the solver. Therefore, the search algorithm we use to solve this problem, as many others algorithms, can be also parallelized. "Parallelism is the wave of the future.. and always will be" is a famous quote in the parallel computing community cited into the introduction of [8]. Thanks to the rapid growth of multi-core architectures, clusters of computation, p2p networks, etc., that future is becoming closer. There has been already significant work in parallelizing, for example, SAT solvers [9], and some attempts on MaxSAT solvers [10]. Also, both portfolio and automatic configuration tools can also greatly benefit from parallel computing.

1.1 Objectives

The main goal of this project is to provide a distributed automatic configuration tool. In particular, we will extend the state-of-the-art automatic configuration tool *GGA*. In order to reach our goal, we will work on the following objectives:

- Identify strengths and weaknesses of the current *GGA* design.
- Redesign the architecture of *GGA* to make it more modular and extensible.
- Identify the most suitable existing technologies for supporting a distributed version of *GGA* on massive computing facilities: clusters, grids, etc.
- Design and implement a distributed version of *GGA* (*DGGA*).

- Evaluate the performance of *DGGA* on algorithms for hard combinatorial problems by experimenting with highly parametrizable state-of-the-art SAT algorithms.

1.2 Document structure

This document is structured into 9 chapters.

Chapter 1 corresponds to the introduction of this project.

Chapter 2 reviews the state of the art in automatic configuration algorithms and distributed architectures for high performance architectures.

Chapter 3 presents the architecture of the automatic configuration tool *GGA*. We present the class diagram of the main classes and the execution diagram of the main process.

Chapter 4 presents the architecture of *DGGA*. This is the distributed version of *GGA* and constitutes the original contribution of this project. We present the class diagram of the main classes and the details of the communication infrastructure to hold the distributed architecture.

Chapter 5 describes the details of the implementation of *DGGA* tool. Additionally, it presents a list of fixed bugs in the original *GGA* tool.

Chapter 6 presents the experimental evaluation we have conducted to evaluate the soundness, robustness and performance of the approach. The target algorithms and instances to be configured which come from Constraint Programming applications and Operations Research applications.

Chapter 7 corresponds to the installation guide of the *DGGA* tool. We also include a fully detailed execution example.

Chapter 8 details the chronology of the projects summarizing and dating all the tasks conducted.

Finally, *Chapter 9*, presents the conclusions of this project and the future research lines.

Chapter 2

State of the art

In this chapter, we will present an overview of the of automatic configuration algorithms and we will particularly focus on a Gender based Genetic Algorithm (*GGA*) which is the starting point of this project. We make clear that a significant portion of the description comes from [6].

Once we have described *GGA*, since the goal of the project is to parallelize this tool, we will also briefly review the distributed architectures for high performance computing on which the distributed version *DGGA* will be deployed.

2.1 Automatic configuration algorithms

Several approaches exist in the literature for the automatic tuning of algorithms. The first methods were created for tuning specific algorithms for a certain task. [11] devised a modular algorithm for solving constraint satisfaction problems (CSPs) and used a combination of exhaustive enumeration of all possible configurations and a parallel hill-climbing technique to automatically configure the system for a given CSP with an associated set of training instances. [12] classified local search (LS) approaches for SAT by means of context-free grammars and devised a genetic programming approach to select a good LS algorithm for a given set of SAT problems. [13] embedded a sequential parameter optimization approach in a wider framework for the design of evolutionary algorithms.

To tune the continuous parameters of general algorithms, [14] suggested an approach that determines good parameters for individual training instances. These parameters are found by trying configurations where parameters are at their extreme values and then fitting a regression function to the parameter/value tuples obtained in this way. The minimization of the resulting function yields a set of parameters for the given instance. A parameter set for the entire collection of instances was then obtained by averaging the parameter tuples for the individual instances.

Tuning problems with small sets of parameter configurations were considered in [15], a setting which is closely related to that in algorithm portfolios [16, 17]. In this case, it is possible to race the different algorithms against each other, whereby a statistical test is used to eliminate inferior algorithms before the remaining algorithms are run on the next training instance.

In [18] Oltean used evolutionary algorithms by means of linear genetic programming. The genome of an individual is an encoding of an actual C-program for the problem to be solved, and crossover and mutation operators are problem dependent. The linear genetic program generates new individuals which replace the current worst individual in the population.

The CALIBRA system, proposed by [19], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine is started from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments. The only system we know of that can configure arbitrary algorithms with very large numbers of parameters was proposed by [20]. Their system, called ParamILS, conducts an iterated local search, whereby a special technique is used to limit the number of training instances that need to be run for each parameter set by focusing the test runs on promising parameter sets. In particular, a new set of parameters is not considered better than the current best until it has been evaluated on at least as many training instances as the current best. If a very large set of training instances is available, this approach allows quick movement through the search space while still avoiding an "over-tuning" effect which would be caused by considering few training instances only.

2.1.1 Gender based Genetic Algorithm (GGA)

In [6] it is proposed a genetic algorithm for the problem of configuring solvers. There are two main reasons for this choice of approach. First of all, genetic algorithms are known to be very robust with respect to optimization problems that have undesirable objective landscapes [21]. Note that, in ordinary optimization, we usually have the freedom to adjust the objective in such a way that it is better suited for sequential local search which often yields good solutions faster than population-based approaches. In contrast, in our setting, where the target algorithm is given and the effect of changing parameters is a priori unknown, we must be able to cope with what-ever objective landscape we encounter. The other reason is that genetic algorithms are inherently parallel. When trying to assess which individuals are competitive (the most time-intensive step in solver configuration), genetic algorithms allow us to race them against each other. Therefore, the time spent for the evaluation is determined by the good parameter sets, and this saves a lot of time in practice. In order to really exploit this last aspect, in [6] it is introduced the concept of gender in the genetic algorithm. It is proposed to apply different selection pressure on the two gender populations. In particular, it is applied intra-specific competition only in one part of the population (competitive individuals or genomes). Individuals in this group must compete for the right of mating, and only the fittest in each generation win the right to mate with some of the individuals of the opposite gender (non-competitive individuals or genomes). The individuals in this other group are not subjected to intra-specific selection.

In the following, we describe the specific *GGA* that follows the structure of a typical Genetic Algorithm.

GGA distinguish three types of target algorithm parameters: continuous and integer parameters, both associated with an upper and lower bound, and categorical values that come with an explicit list of feasible values. In addition *GGA* has the input parameters (*X, P, M, A, S*) which are used in the following way:

- **Initialization:** first, we randomly initialize the population and assign a gender C (for competitive) or N (for non-competitive) and an "age" of 1 to A years uniformly at random to each individual. In our experiments, we set A to 3.

- **Mating Rules:** among the individuals with gender C, we select the top $X\%$ (in our experiments we set X to 10%). These have gained the right to mate in this season. $200/A\%$ of individuals of gender N are assigned uniformly at random to one of the mating individuals of gender C. The individuals of gender C then mate with all individuals of gender N which have been assigned to them.

- **Crossover:** each mating of a couple of genomes results in one new genome with age 0 and random gender. The genome of the offspring is determined by traversing the parameter tree top-down.

  The parameter tree is the representation of the target algorithm parameters as an And/Or tree (see e.g [22]), where:

  - Each node is labelled with a parameter or the additional label "&", and each algorithm parameter is associated with at least one node.
  - Nodes associated with continuous or integer parameters have at most one child, and And-nodes have at least two child-nodes.

  - The children of categorical nodes partition the set of values that their parent parameter can take. Branches leading to the children are labelled by the respective value(s) of the categorical parameter.

- **Mutation:** as a final step to determine the offspring's genome, with probability $M\%$ we mutate the value of each parameter (in our experiments, we set M to 10%). If we mutate a categorical parameter, we choose a new value in its domain uniformly at random. For continuous and integer parameters, we choose a new value according to a Gaussian distribution where the current value marks the expected value and the variance is set as $S\%$ of the parameter's domain. In our experiments, we set S to 10%.

- **Death:** after the new offspring is created, all individuals' ages are increased by 1. Those with age greater than A are removed from the population. In combination with the mating rules that only $200/A\%$ of individuals of gender N mate in every season, this stabilizes the total population size.

2.2 Distributed architectures for high performance computing

A distributed computer system consists of multiple software components that are on multiple computers, but run as a single one. There is no clear distinction between "distributed computing" and "parallel computing", in fact the same system may be characterized as "parallel" and "distributed". A distributed system composed by multi-core computers is inherently running concurrently in parallel.

We can consider that parallel computing is composed by tightly coupled components, while the components of a distributed computing system are loosely coupled. Therefore we can classify concurrent systems using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange data.
- In distributed computing, the processors do not have access to shared memory and data is exchanged by passing messages between them.

Current clusters for high performance computing allow to have both types of concurrent systems since as we will see in the next section, they are composed by nodes with their own private memory that can cooperate. Moreover, each node usually corresponds to a multi-core computer where all jobs running on it can access to shared memory.

In this project, since we will follow a Master-Worker pattern where Workers run on independent computers, we will use the term “distributed” to reference the parallelization of *GGA*.

2.2.1 Computer cluster

A computer cluster is a set of loosely or tightly connected independent computers, with common hardware components, that work together as a single system. The computers of a cluster are usually connected to each other through fast Local Area Networks (LAN) and share the same file system.

Within a computer cluster we can identify three different types of nodes:

- **Master node**: it is in charge of the cluster administration and has the scheduler and several parallel libraries.
- **Submit/Interactive nodes**: they are the users entry points and are mainly used to launch jobs.
- **Computational nodes**: these are the nodes where the jobs are executed.

2.2.2 Job scheduler

A job scheduler, also known as distributed resource management system (DRMS), is a computer application for controlling unattended background program execution. It is the responsible of the properly distribution of the resources of a distributed system among the requested jobs.

A global view of a job scheduler has two main modules:

- A **Distributed resource management** (DRM) module, which is in charge of managing the resources of the system.
- A **Scheduler** module, which is the responsible to instruct to the DRM what to do with the available resources. It is also in charge of monitoring the jobs and managing them at run time.

The scheduler module uses different policies to grant that all the jobs can be executed avoiding resource conflicts between them. Some examples are:

- **First In First Out** (FIFO): it is the simplest algorithm. It simply queues jobs in the order that they arrive.
- **Round-Robin** (RR): the scheduler assigns a time unit per job and cycles through them.
- **Shortest Job First** (SJF): it requires an estimation of the required execution time of each jobs. Then, the jobs are arranged based on that estimation.
- **Multilevel queue**: this policy is used when the jobs can be classified into groups based on properties like CPU time or IO access. Each queue has a preassigned priority and its own scheduling algorithm.

There are several implementations of job schedulers, but in this project we are focused in the batch-queue schedulers, which are the most widely used in computer clusters.

A batch-queue scheduler is an specific implementation of a job scheduler. In this type of schedulers the available resources are grouped in one or several queues. By grouping resources, a queue is automatically imposing some resource limitations to the jobs. Besides this physical limitations, a queue can also be configured to force additional logical limitations or requirements to the jobs.

SGE

The Sun Grid Engine (SGE), originally developed by Sun Microsystems and continued by Oracle under the name Oracle Grid Engine, is a batch-queue system used in high-performance computing (HPC) clusters running on UNIX-like operating systems or Windows with the SFU or SUA extensions. The system is responsible of accepting, scheduling, dispatching and managing, standalone, parallel or interactive user jobs and the resources requested for those jobs.

Chapter 3

GGA architecture

This chapter aims to provide a general vision of the original *GGA*. In particular, we will focus on those parts which had been substantially modified to create the distributed version of *GGA* (*DGGA*).

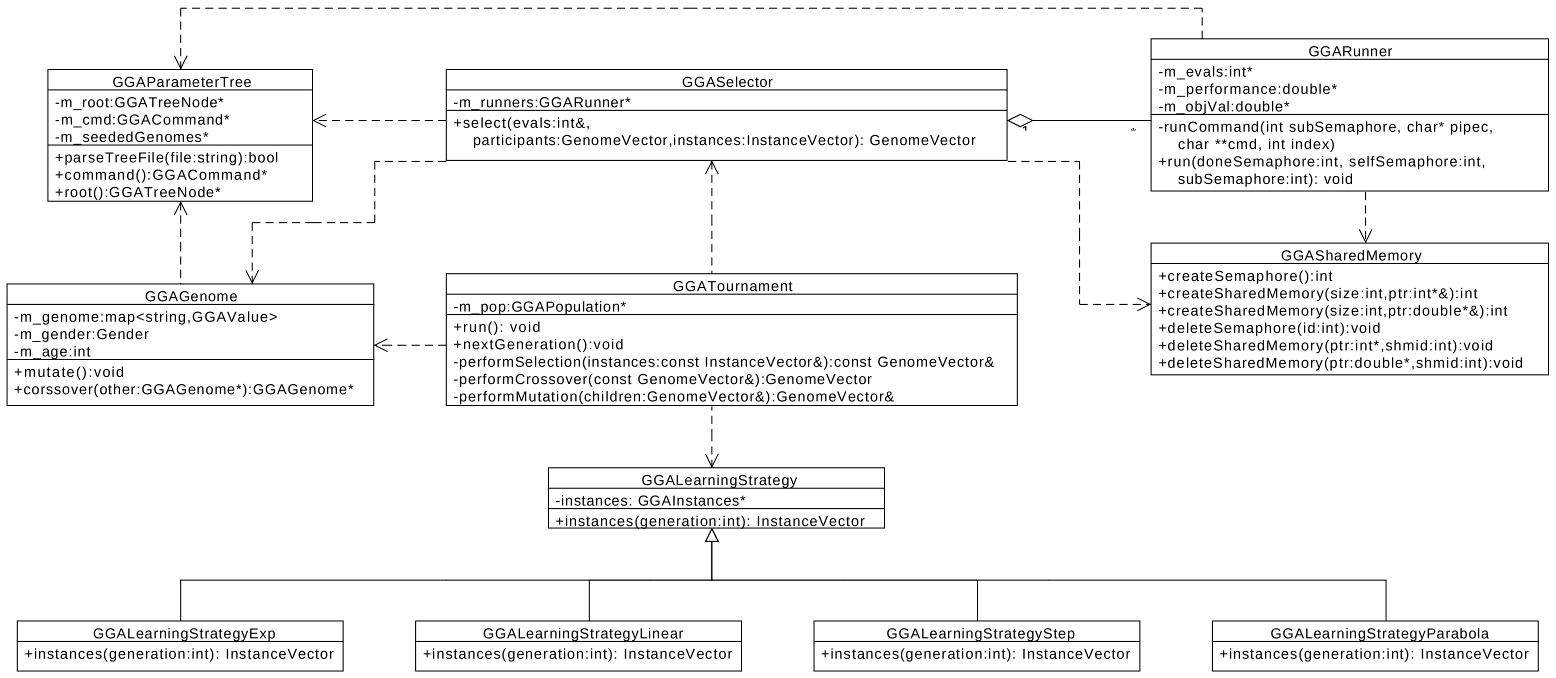
First, we will present the simplified class diagram of *GGA* that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. Then, we will discuss the main execution sequence of *GGA*.

3.1 Class diagram

The current *GGA* system involves up to 16 classes. For the sake of clarity, we present in figure 3.1 only the critical classes needed for describing the general functionality of the system. These classes are: *GGATournament, GGALearningStrategy, GGASelector, GGARunner, GGASharedMemory, GGAGenome* and *GGAParameterTree.*

Class GGATournament

The class GGATournament manages the overall tuning process. First of all, from the current population of genomes, divided into competitive and non-competitive, it sends to GGASelector the competitive genomes. From GGASelector it retrieves the set of winners. Then, it applies the crossover operation of individuals from the non-competitive genomes and the set of winners to get new genomes. The new genomes are mutated and labelled as non-competitive or competitive. Finally, those genomes



GGAParameterTree
-m_root:GGATreeNode*
-m_cmd:GGACommand*
-m_seededGenomes*
+parseTreeFile(file:string):bool
+command():GGACommand*
+root():GGATreeNode*
GGASelector
-m_runners:GGARunner*
+select(evals:int&, participants:GenomeVector,instances:InstanceVector): GenomeVector
GGARunner
-m_evals:int*
-m_performance:double*
-m_objVal:double*
-runCommand(int subSemaphore, char* pipec, char **cmd, int index)
+run(doneSemaphore:int, selfSemaphore:int, subSemaphore:int): void
GGASharedMemory
+createSemaphore():int
+createSharedMemory(size:int,ptr:int*&):int
+createSharedMemory(size:int,ptr:double*&):int
+deleteSemaphore(id:int):void
+deleteSharedMemory(ptr:int*,shmid:int):void
+deleteSharedMemory(ptr:double*,shmid:int):void
GGAGenome
-m_genome:map<string,GGAValue>
-m_gender:Gender
-m_age:int
+mutate():void
+corssover(other:GGAGenome*):GGAGenome*
GGATournament
-m_pop:GGAPopulation*
+run(): void
+nextGeneration():void
-performSelection(instances:const InstanceVector&):const GenomeVector&
-performCrossover(const GenomeVector&):GenomeVector
-performMutation(children:GenomeVector&):GenomeVector&
GGALearningStrategy
-instances: GGAInstances*
+instances(generation:int): InstanceVector
GGALearningStrategyExp
+instances(generation:int): InstanceVector
GGALearningStrategyLinear
+instances(generation:int): InstanceVector
GGALearningStrategyStep
+instances(generation:int): InstanceVector
GGALearningStrategyParabola
+instances(generation:int): InstanceVector

Figure 3.1: GGA simplified class diagram

too old for the generation are eliminated. This process is repeated until a predefined stop criterion is achieved.

Main members:

- `m_pop`: an object of class GGAPopulation that contains the tournament's population divided in two vectors: one for the competitive genomes and another for the non-competitive ones.

Main methods:

-
  ```
  void run()
  ```

  This method starts the routine that manages the overall tuning process.

-
  ```
  void nextGeneration()
  ```

  Perform all the required steps to create the individuals of the next generation.

-
  ```
  const GenomeVector performSelection(
                          const InstanceVector& instances)
  ```

  Selects the best individuals of the competitive population.

  Parameters

  – *instances* [in]
    A vector with the instances to evaluate the competitive individuals.

  Return value

  A vector with references to the best individuals of the population.

-
  ```
  GenomeVector performCrossover(const GenomeVector&
      winners)
  ```

  Performs the crossover between the individuals returned by `performSelection()` and the non-competitive population. To generate new individuals.

  Parameters

  – *winners* [in]
    A vector of references to competitive GGAGenomes.

  Return value

  A vector of new GGAGenomes. Result of combining the *winners* with the non-competitive population.

- `GenomeVector& performMutation(GenomeVector& children)`

  Mutates the genome of all the provided individuals.

  Parameters

  - *children* [in, out]
    A vector with references to the individuals to mutate.

  Return value

  The reference to the vector *children* provided as parameter.

Dependencies: GGAGenome, GGALearningStrategy and GGASelector

Class GGALearningStrategy

The class GGALearningStrategy is in charge of selecting the instances used by GGA-Tournament, at each generation, to select the most fit genomes of the current generation. It has 4 different implementations, with a different selection policy each.

Main members:

- `m_instances`: a pointer to the list of selectable instances.

Main methods:

- `virtual InstanceVector instances(int generation)`

  Selects the instances to test the specified generation of individuals. The default implementation is return all the instances.

  Parameters

  - *generation* [in]
    The generation for which select the instances.

  Return value

  A vector with references to the selected instances.

Class GGALearningStrategyLinear

Specific implementation of GGALearningStrategy that returns randomly selected instances. The amount of selected instances increases linearly respect to the number of generations.

Class GGALearningStrategyStep

Specific implementation of GGALearningStrategy that returns randomly selected instances. The amount of selected instances increases discretely respect to the number of generations.

Class GGALearningStrategyParabola

Specific implementation of GGALearningStrategy that returns randomly selected instances. The amount of selected instances increases in a parabola-like way, with respect to the number of generations.

Class GGALearningStrategyExp

Specific implementation of GGALearningStrategy that returns randomly selected instances. The amount of selected instances increases exponentially respect to the number of generations

Class GGASelector

The class GGASelector manages the assignment of the competitive genomes to the available computational resources, and retrieves the winners from the set of the competitive genomes at the current generation.

Main members:

- `m_runners`: a vector of runners used to compute the performance of each genome.

Main methods:

-
  ```
  virtual GenomeVector select(
                      const GenomeVector& participants,
                      const InstanceVector& instances,
                      int& evals)
  ```

  Performs several tasks. First assigns the available computation resources to the set of competitive individuals. Then executes a bunch of runners to gather the performance of those individuals, and finally selects the winners.

  Parameters

  - *participants* [in]
    The list of individuals to evaluate.
  - *instances* [in]
    The list of instances to evaluate the participants.
  - *evals* [out]
    The number of evaluations performed during the selection process.

  Return value

  A vector with references to the best individuals after the evaluation.

Dependencies: GGARunner

Class GGARunner

The GGARunner class executes the target algorithm, using the parametrization of a given genome on the instances provided by the GGALearningStrategy, to compute the performance of the genome.

Main members:

- `m_evals`: the number of evaluations performed to compute the genome performance.
- `m_performance`: a vector that contains the performance of the genome for each tested instance.
- `m_objVal`: the sum of all the values in m_performance.

Main methods:

-
  ```
  void run(int doneSemaphore, int selfSemaphore,
           int subSemaphore)
  ```

  Executes the evaluation routine for each instance.

  Parameters

  - *doneSemaphore* [in]
    The identifier of the semaphore used to notify that some runner has finished.
  - *selfSemaphore* [in]
    The identifier of the semaphore used to notify that this specific runner has finished.
  - *subSemaphore* [in]
    Not Used.

-
  ```
  void runCommand(int subSemaphore, char* pipec,
                  char** cmd, int index)
  ```

  Executes the command to test the target algorithm with the genome parametrization, on one of the instances, and recover execution statistics.

  Parameters

  - *subSemaphore* [in]
    Not Used.
  - *pipec* [in]
    The name of the named pipe, used to communicate the *GGA* process with the child process running the target algorithm.
  - *cmd* [in]
    The command to execute the target algorithm, with an specific parametrization on one of the test instances.
  - *index* [in]
    The index of the current test instance to use.

Class GGAGenome

The GGAGenome class, represents an individual of the tournament population.

Main members:

- `m_genome`: it is a map that contains the parametrization of the target algorithm.
- `m_gender`: stores information about the gender of the individual.

- `m_age`: the age of the individual.

Main methods:

- `void mutate()`

  Modifies the genomic information of this individual.

- `GGAGenome crossover(GGAGenome* other)`

  Creates a new GGAGenome by mixing the genomic information of this genome an the given one.

  Parameters

  - *other* [in]
    Another individual of different gender.

  Return value

  A new individual result of crossing the information of this and the given one.

Class GGASharedMemory

The class GGASharedMemory is an utility class composed entirely by static methods. It provides an abstraction layer over some operating system's IPC mechanisms, and can also generate random names to create named pipes.

Main methods:

- `int createSharedMemory(int size, int*& ptr)`

  Creates a block of integers in shared memory.

  Parameters

  - *size* [in]
    The number of integers to allocate in shared memory.
  - *ptr* [out]
    Reference to the pointer used to access the block of memory.

  Return value

  The identifier of the allocated block of memory.

- `int createSharedMemory(int size, double*& ptr)`

  Creates a block of doubles in shared memory.

  Parameters

  - *size* [in]
    The number of doubles to allocate in shared memory.
  - *ptr* [out]
    Reference to the pointer used to access the block of memory

  Return value

  The identifier of the allocated block of memory.

- `void deleteSharedMemory(int* ptr, int shmid)`

  Deletes the specified block of integers.

  Parameters

  - *ptr* [in]
    Pointer to a previously allocated block of memory.
  - *shmid* [in]
    The identifier of the block of memory pointed by *ptr*.

- `void deleteSharedMemory(double* ptr, int shmid)`

  Deletes the specified block of doubles.

  Parameters

  - *ptr* [in]
    Pointer to a previously allocated block of memory.
  - *shmid* [in]
    The identifier of the block of memory pointed by *ptr*.

- `int createSemaphore()`

  Creates a semaphore initialized to 0.

  Return value

  The identifier of the recently created semaphore.

- `void  deleteSemaphore(int id)`

  Deletes the specified semaphore.

  Parameters

  - *id* [in]
    The identifier of a semaphore.

Class GGAParameterTree

The class GGAParameterTree contains the parameter tree of the target algorithm along with the command to test it, and optional user provided genomes. It is also the responsible of parsing and verifying the algorithms configuration file.

Main members:

- `m_root`: the root of the parameter tree.
- `m_cmd`: the command to execute the target algorithm.
- `m_seededGenomes`: the list of user provided genomes.

Main methods:

-
  ```
  bool parseTreeFile(std::string file)
  ```

  Parses the XML file with the target algorithm configuration.

  Parameters

  - *file* [in]
    The path to the XML file with the tree configuration.

  Return value

  True if there is no error parsing the file, false otherwise.

-
  ```
  GGATreeNode* root()
  ```

  Return value

  A reference to the root of the parameter tree, or NULL if there is no tree.

-
  ```
  GGACommand* command()
  ```

  Return value

  A reference to the command to execute the target algorithm, or NULL if there is no command.

3.2 Execution sequence

This section explains the sequence of steps performed by *GGA* (see figure 3.2) to try to find the best possible configuration of an algorithm , i.e., the genome with the best performance, respect to a given test set of instances, and a restricted time and memory resources.

Initially, *GGA* loads the data provided by the user. This data corresponds to: the solver to tune and its parameters, the test set of instances and some execution parameters of the *GGA* tool itself (see Chapter 7). With this data *GGA* creates a randomly generated population, i.e., set of genomes, that corresponds to the first generation.

From now on, class GGATournament leads the main configuration process. This configuration process lasts to a user defined number of generations. Within each generation, class GGATournament performs three main steps: the selection of the instances to evaluate the competitive genomes of the current population, the actual evaluation of these genomes, and the update of the population for the next generation.

In order to select the instances for the evaluation of the genomes, class GGA-Tournament interacts with class GGALearningStrategy. The latter, returns a subset of instances depending on the selection strategy defined by the user (see previous section for the details of the strategies).

The competitive genomes and the selected subset of instances are then sent to GGASelector to evaluate their performance and select a percentage of winners.

Class GGASelector starts the selection process by distributing the competitive genomes in several balanced mini-tournaments. After the counterbalance, for each mini-tournament, it creates a GGARunner instance per genome. Then, each GGARunner is executed, into a separated process, to evaluate the genomes of the current mini-tournament asynchronously. Notice that the size of a mini-tournament is at most the number of cores of the machine where GGSelector is running.

When all the GGARunners for a given mini-tournament have finished, class GGASelector selects a percentage of the winners from the current mini-tournament.

Once all the mini-tournaments have finished, class GGASelector returns the set of all the winners to class GGATournament, which creates the next individuals of the generation by crossing the winners with the non-competitive genomes of the current generation. These new genomes are then mutated to introduce more diversity to the population.

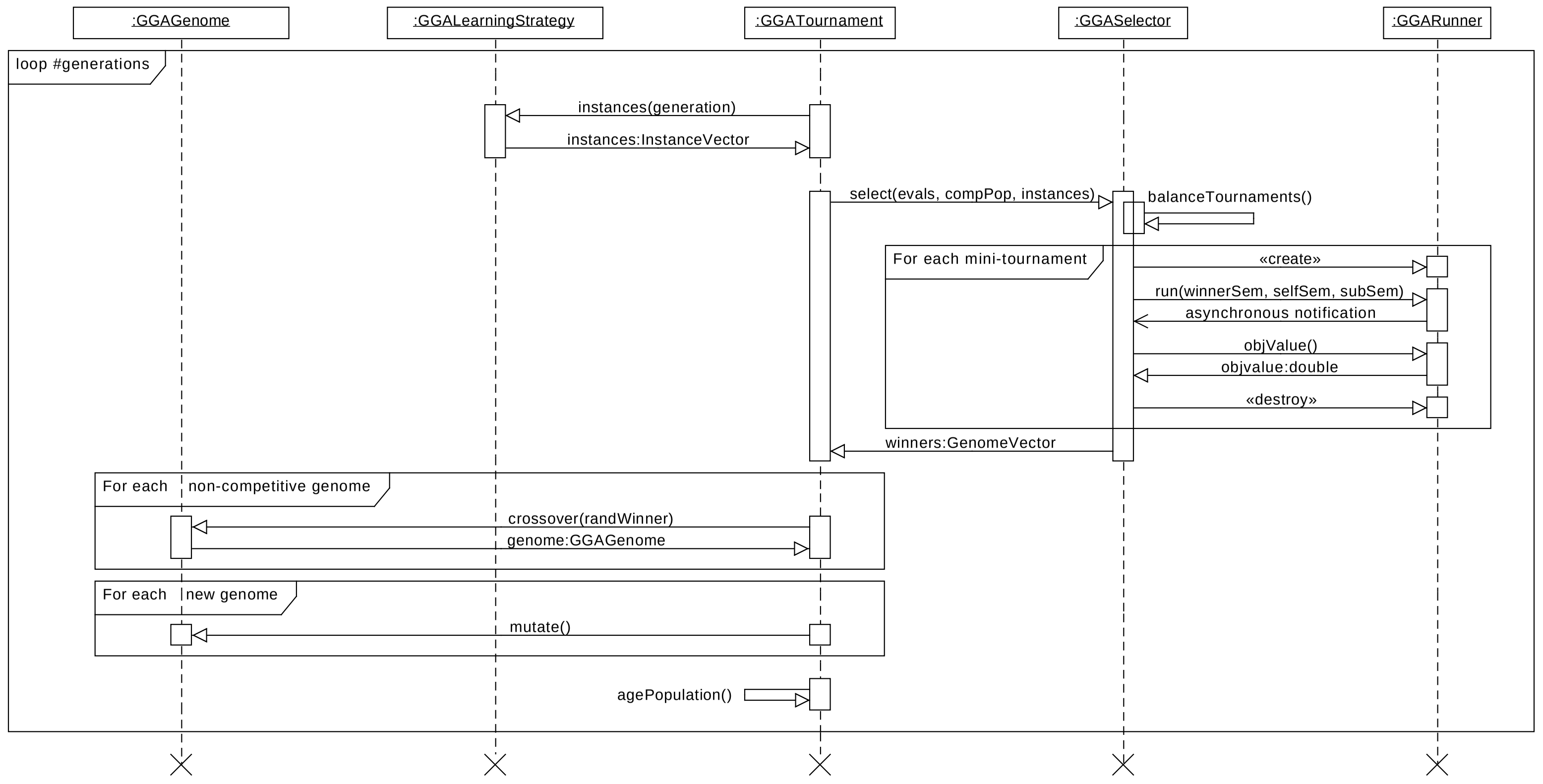


Figure 3.2: GGA simplified sequence diagram

Finally, before starting with the next generation, the age of the individuals of the current population is increased and the older ones are discarded and removed from the population of genomes.

Chapter 4

DGGA architecture

In this chapter, we present the distributed architecture of *DGGA* (see figure 4.1). *DGGA* has been designed trying to reuse as much as possible the original *GGA* extended with a lightweight communication component.

In the following sections, we present the class diagram showing only the main classes, and the communication protocol employed in the distributed environment.

4.1 Class diagram

DGGA has been designed so that it can be run transparently on a single machine as the original GGA, or on any cluster with a shared file system. Moreover, although it is not covered in this project, the design allows to deploy DGGA on any distributed architecture (that guarantees uniform computing resources) without applying any substantial modification.

In figure 4.1, we can see that *DGGA* follows the typical Master-Worker pattern. Basically, the Master sends to each Worker a set of genomes to be tested on a set of instances. The Workers are responsible on evaluating the genomes on the set of instances and send the results to the Master.

The Master basically corresponds to the class GGATournament in *GGA*. Class DGGARemoteSelectorMaster replaces class GGASelector in *GGA* to make transparent the communication with the Workers. The Worker corresponds to the classes GGASelector and GGARunner in *GGA*. Class DGGARemoteSelectorWorker wraps class GGALocalSelector, GGASelector in *GGA*, making the communication trans-

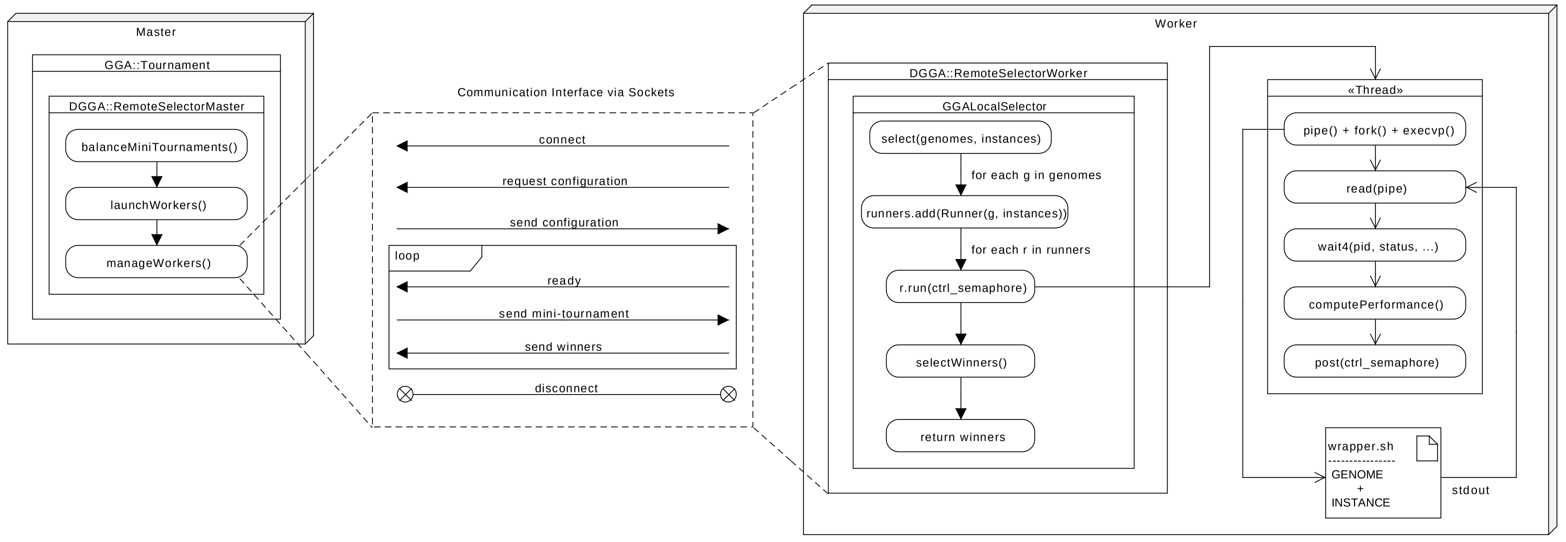


Figure 4.1: DGGA general architecture overview

parent for the *GGA* system.

Going into more detail, in figure 4.2 we can consult the class diagram of the main classes in *DGGA*. We basically distinguish two groups, the new set of classes, and the set of classes inherited from *GGA* that have been modified.

4.1.1 New classes

Below, we describe the new four classes we have added respect to GGA: *DGGARemoteSelectorMaster, DGGARemoteSelectorWorker, DGGATcpAcceptor* and *DGGATcpConnection*. Finally, we present the GGA classes that have been modified.

Class DGGARemoteSelectorMaster

The DGGARemoteSelectorMaster has three main tasks. First it assigns the competitive genomes to the available computational resources, creating one or several mini-tournaments. Then it starts several Workers, and finally manages the communication with the Workers and recovers the results.

Main members:

- `m_start_worker_cmd`: the command that must be executed to start *DGGA* Workers.
- `m_acceptor`: an object of the class DGGATcpAcceptor responsible of managing the acceptance of new DGGATcpConnections.
- `m_connections`: a vector of DGGATcpConnections with all the active Workers.

Main methods:

-
  ```
  virtual GGAGenomeVector select(int& evals,
            const GGAGenomeVector& participants,
            const GGAInstanceVector& instances)
  ```

  Distributes the available computational resources among the set of competitive individuals and, starts some Workers. Then enters an event based loop, driven by the communication protocol, until the selection has finished.

  Parameters

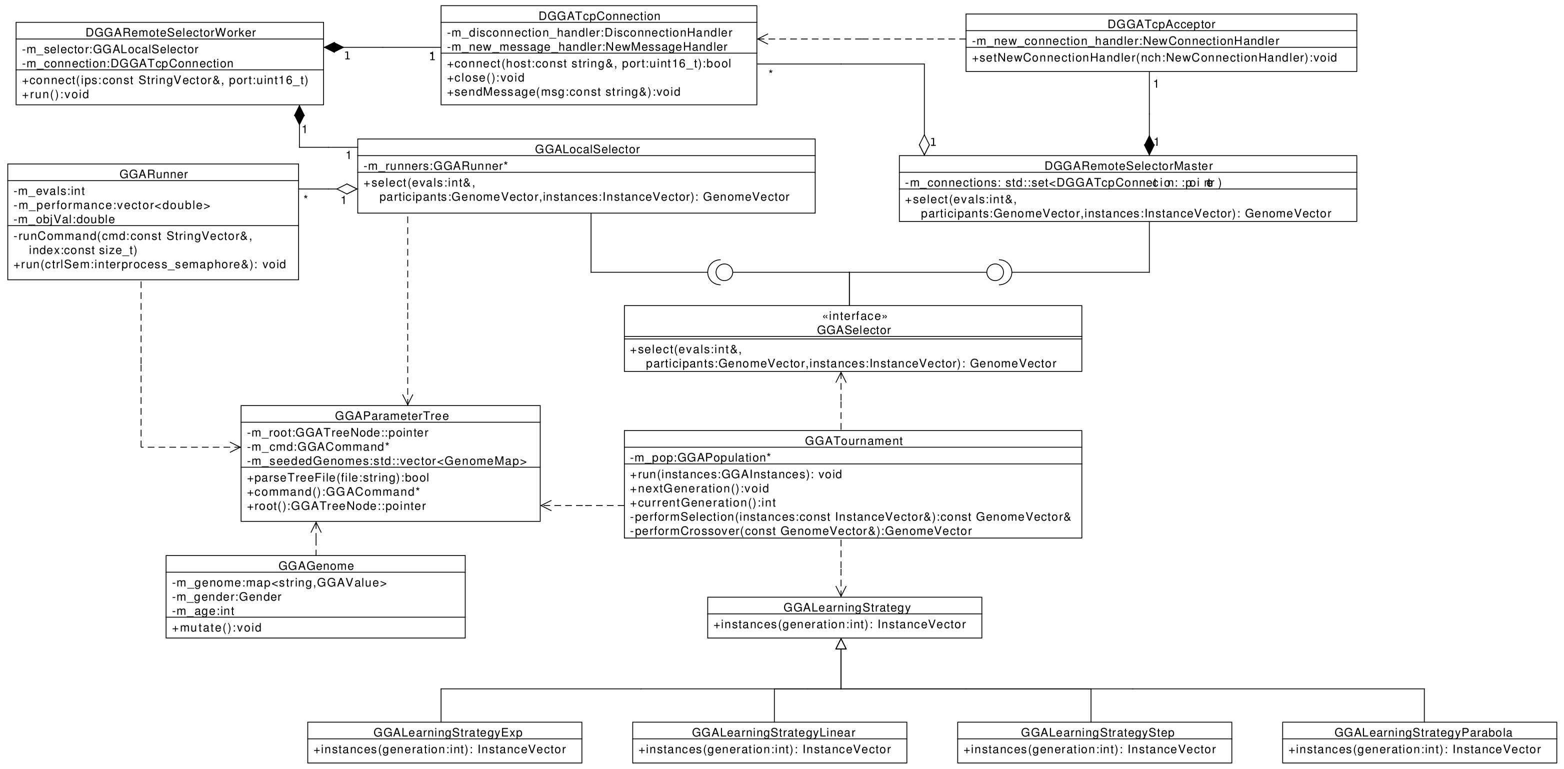

DGGARemoteSelectorWorker
-m_selector:GGALocalSelector
-m_connection:DGGATcpConnection
+connect(ips:const StringVector&, port:uint16_t)
+run():void
DGGATcpConnection
-m_disconnection_handler:DisconnectionHandler
-m_new_message_handler:NewMessageHandler
+connect(host:const string&, port:uint16_t):bool
+close():void
+sendMessage(msg:const string&):void
DGGATcpAcceptor
-m_new_connection_handler:NewConnectionHandler
+setNewConnectionHandler(nch:NewConnectionHandler):void
GGALocalSelector
-m_runners:GGARunner*
+select(evals:int&,
participants:GenomeVector,instances:InstanceVector): GenomeVector
DGGARemoteSelectorMaster
-m_connections: std::set<DGGATcpConnection::pointer>
+select(evals:int&,
participants:GenomeVector,instances:InstanceVector): GenomeVector
GGARunner
-m_evals:int
-m_performance:vector<double>
-m_objVal:double
-runCommand(cmd:const StringVector&,
index:const size_t)
+run(ctrlSem:interprocess_semaphore&): void
«interface»
GGASelector
+select(evals:int&,
participants:GenomeVector,instances:InstanceVector): GenomeVector
GGAParameterTree
-m_root:GGATreeNode::pointer
-m_cmd:GGACommand*
-m_seededGenomes:std::vector<GenomeMap>
+parseTreeFile(file:string):bool
+command():GGACommand*
+root():GGATreeNode::pointer
GGATournament
-m_pop:GGAPopulation*
+run(instances:GGAInstances): void
+nextGeneration():void
+currentGeneration():int
-performSelection(instances:const InstanceVector&):const GenomeVector&
-performCrossover(const GenomeVector&):GenomeVector
GGAGenome
-m_genome:map<string,GGAValue>
-m_gender:Gender
-m_age:int
+mutate():void
GGALearningStrategy
+instances(generation:int): InstanceVector
GGALearningStrategyExp
+instances(generation:int): InstanceVector
GGALearningStrategyLinear
+instances(generation:int): InstanceVector
GGALearningStrategyStep
+instances(generation:int): InstanceVector
GGALearningStrategyParabola
+instances(generation:int): InstanceVector
1
1
1
1
*
1
1
1
*
1

Figure 4.2: DGGA simplified class diagram.

  - *evals* [out]
    The number of evaluations performed during the selection process.
  - *participants* [in]
    The list of individuals to evaluate.
  - *instances* [in]
    The list of instances to evaluate the participants.

  Return value

  A vector with the best individuals selected by the Workers.

- `void startWorkers()`

  Starts some Workers using the user provided command.

- `bool sendTourney(DGGATcpConnection::pointer con)`

  Sends a mini-tournament to the specified Worker (connection).

  Parameters

  - *con* [in]
    The connection to which send the tournament.

  Return value

  True if a mini-tournament was sent, false otherwise.

- `void recoverResults(const std::string& str)`

  Extracts the results from the given string.

  Parameters

  - *str* [in]
    A constant reference to a string with the results of a mini-tournament.

Class DGGARemoteSelectorWorker

The DGGARemoteSelectorWorker has only one responsibility, manage the communication with the Master. All the other components are inherited from *GGA* with slight modifications.

Main members:

- `m_selector`: an instance of the original *GGA* selector used to select the winners of a mini-tournament.

- `m_connection`: an instance of DGGATcpConnection that handles the connection with the Master.

Main methods:

- `bool connect(const StringVector& ips, uint16_t port)`

  Tries to connect to any of the provided IPs at the specified port.

  Parameters

  - *ips* [in]
    A list of IPs to connect to.
  - *port* [in]
    The connection port.

  Return value

  True if a connection is established, false otherwise.

- `void run()`

  Runs the Worker main loop until either the tuning process finishes or there is a communication error.

Class DGGATcpAcceptor

The class DGGATcpAcceptor is an event based acceptor for TCP connections. It listens to an specific address and port, and calls a hook function whenever a new connection is accepted.

Main members:

- `m_new_connection_handler`: an instance of the type NewConnectionHandler that will be called when a new connection is accepted.
- `m_acceptor`: an instance of the underlying acceptor object.

Main methods:

- `void setNewConnectionHandler(NewConnectionHandler nch)`

Sets the new connection event hook function.

Parameters

- *nch* [in]
  It can be either a C-like function or a boost::function, with the restriction that should return void and accept a DGGATcpAconnection::pointer as the unique parameter.

Class DGGATcpConnection

The class DGGATcpConnection is a wrapper over a TCP connection that hides all the communication issues. It sends and receives a predefined format of messages, and uses events to notify any communication change.

Main members:

- `m_disconnection_handler`: an instance of the type DisconnectionHandler that will be called when the connection is finished.
- `m_new_message_handler`: an instance of the type NewMessageHandler that will be called when a new message is received.

Main methods:

-
  ```
  bool connect(const std::string& host, uint16_t port)
  ```

  Tries to connect to the specified address (*host:port*). If the return value is true, this connection can be used to send and receive data.

  Parameters

  - *host* [in]
    An IP or host name to connect to.
  - *port* [in]
    The connection port.

  Return value

  True if a connection is established with the given address, false otherwise.

-
  ```
  void close()
  ```

  Closes a previously established connection. After this method, this connection can not be used to send or receive data.

- `void sendMessage(const std::string& msg)`

  Sends the given message to the other end of the connection. This function fails if a connection have not been already established.

  Parameters

  - *msg* [in]
    The message to send.

4.1.2 Modified classes

During the development of *DGGA*, we have modified some of the classes inherited from *GGA*, in order to make more easy to implement the modifications required to distribute the original *GGA* tool. Below, we list the modifications we have made to the main classes of *GGA* already introduced in Chapter 3.

Class GGASelector

This class was renamed as GGALocalSelector. We reused the name GGASelector for an interface, which defines the methods required by GGATournament, to perform the selection of the best genomes. This interface is the one that allows the distributed architecture to be transparent for the GGATournament class.

Class GGARunner

Basically we have shaved this class by erasing unrequired attributes and simplifying methods.

Class GGAGenome

The `crossover()` method has been refactored as a non-friend, non-member function. This change was made following the principles explained in *Effective C++* [23](Chapter 4, Item 23), to make the code more flexible, clean and with better encapsulation. These are important points to cover when modifying any software.

Class GGASharedMemory

The functionality provided by this class was replaced by a better and cross-platform implementation provided by the *Boost* project. While GGASharedMemory only covers the creation of semaphores, and shared memory blocks of integers and doubles, using the Unix IPC mechanisms. *Boost* offers a more mature and cross-platform implementation, of threads, shared memory, semaphores, and other interprocess communication mechanisms.

Class GGAParameterTree

Originally, this class had a significant amount of code devoted to deal with memory management. We changed all the member raw-pointers to smart-pointers and removed almost all the memory management code.

4.2 Communication protocol

The communication protocol of *DGGA* has been designed bearing in mind, that it must be cross-platform and easy to implement and debug. For this reason, we decided to use a text-based protocol over TCP. Any other transmission protocol with less guarantees than TCP is not reliable with the actual design.

In this project the classes that use this protocol are: DGGARemoteSelectorMaster and DGGARemoteSelectorWorker.

The whole protocol can be split in 4 major steps: connection, configuration, execution and disconnection.

Connection

This is the first step and involves different actions in both ends of the communication.

On one side, the Master opens a TCP connection to listen to an specified port on all network interfaces. After that, it is ready to accept incoming connections.

On the other side, the Worker opens a TCP connection and connects to one of the Master's computer network interfaces at the port that the Master is listening to.

Once a connection between the Master and a Worker is established. The connection is in the *INIT* state, and the Worker starts the greeting sequence composed by the following messages:

1. Worker → Master: `DGGA_HELLO`
2. Master → Worker: `DGGA_ACK`

That sequence guarantees that the other end of the communication is aware of the protocol, and the connection switches to the *CONFIG* state .

Configuration

This step follows the greeting sequence, when the connection is in the *CONFIG* state. Its purpose is to guarantee that the Workers share the same configuration than the Master, and that the mini-tournaments will be properly executed.

The sequence of messages of this step, involves 4 fixed messages and one that is the textual serialisation of the Master's configuration. After the sequence, the connection is in the *IDLE* state.

1. Worker → Master: `DGGA_GET_CONFIGURATION`
2. Master → Worker: `DGGA_CONFIGURATION_BEG`
3. Master → Worker: *<configuration data [serialised]>*
4. Master → Worker: `DGGA_CONFIGURATION_END`
5. Worker → Master: `DGGA_ACK`

Notice that the second and the fourth messages are used to enclose the serialised data. Therefore, it is not necessary to know the size of the serialised data in advance.

Execution

This step is performed every time the Master has to evaluate the competitive genomes. When Master requires the execution of the tests, it can only redirect this request to the Workers that are in the *IDLE* state.

The sequence starts with two messages, the objective of these messages is to ensure that the other end is online and that there are no communication errors. After this check, the Master sends the mini-tournament to the Worker. The connection is now in the *WORKING* state. Finally, when the Worker has finished the tests, it sends the results back to the Master and the connection switches again to the *IDLE* state.

1. Master → Worker: `DGGA_POLL`
2. Worker → Master: `DGGA_READY`
3. Master → Worker: `DGGA_TOURNEY_BEG`
4. Master → Worker: *<mini-tournament data [serialised]>*
5. Master → Worker: `DGGA_TOURNEY_END`
6. Worker → Master: `DGGA_RESULTS_BEG`
7. Worker → Master: *<results data [serialised]>*
8. Worker → Master: `DGGA_RESULTS_END`

Notice that the third, the fifth, the sixth and the eighth messages are used to enclose the serialised data like in the **Configuration** step.

Disconnection

The disconnection step takes advantage of the TCP protocol, there is no specific message or sequence to notify to the other end an imminent disconnection. The idea is that any TCP level disconnection is treated as a common event in the communication protocol, which reduces the complexity of writing specific disconnection sequences.

On one side, the Master is responsible of closing all the connections when the configuration process has finished. Hence any disconnection is treated as a communication error. When that happens, the Master rolls-back any mini-tournament assigned to the disconnected Worker. Then, it polls all the Workers whose connection is in the *IDLE* state, to start the mini-tournament again. Finally, if there are more mini-tournaments, it tries to start a new Worker.

On the other side, the Worker expects a disconnection when it is in the *IDLE* state, but it does not expect such event when it is in any other state. In any case, the Worker finishes its execution after it is disconnected from the Master. In addition, if the disconnection happens when the Worker is in the *WORKING* state, it also has to finish the execution of the child processes that are executing the target algorithm.

Chapter 5

DGGA implementation

In this chapter, we will explain the implementation of *DGGA*. First, we will review the libraries and the development tools. Secondly, we will describe how the project is structured in modules and their main implementation details. Finally, we will list the bugs we found in the original *GGA* and how we have fixed them.

Since this project is based on and extends *GGA*, it inherits some of its characteristics. For example, the programming language (C++) and libraries. Moreover during the development of *DGGA*, some of the original modules of *GGA* were modified, removed or replaced, to serve better the purpose of this project.

5.1 Libraries

Initially, *GGA* used external libraries from two projects: *GNOME* [24] and *Boost* [25], but since the *Boost* project itself covers all the requirements and it is implemented in the same programming language as the project, being the integration simpler, we decided to replace the *GNOME* XML library with its *Boost* equivalent.

5.1.1 C++

As mentioned before, C++ is the language used in the original implementation of *GGA*. This language was initially designed by Bjarne Stroustrup [26] to mid-80s, as an extension of the C language [27].

C++ supports three different programming paradigms: structured programming, template programming and object oriented programming. In addition its standard library offers a set of data structures, iterators and algorithms, which release the developer from the low-level details. However its C inheritance, still allow the developers to work at low-level when they require it.

This project takes advantage of the standard library of C++, while it uses its low-level capabilities to communicate with the operating system API.

5.1.2 Boost

The *Boost* project is widely used in C++ projects. Its main goals are: offer generic cross-platform solutions to daily C++ problems and work along with the C++ standard library. For this reason the *C++ Standards Committee* has included some of its libraries in the C++11 release. In addition its licence allows use all its libraries in any kind of project.

We have used the following *Boost* libraries: *System*, *Filesystem*, *Regex*, *Timer*, *Program Options*, *Serialization*, *Chrono*, *Iostreams*, *Thread.* To perform file system, network, multi-threading and serialization tasks.

5.2 Tools

In this section, we briefly explain the different tools used during the development of this project. All of them are open source and cross-platform.

5.2.1 Git

Git [28] is a distributed revision control and source management system. Initially designed and developed by Linus Torvalds for the Linux kernel in 2005, and has since become the most widely adopted version control system for software development.

Some known projects and businesses that use *Git* are: *Google*, *Facebook*, *Microsoft Twitter*, *Linkedin*, *Netflix*, *Android*, *X-org*, *Eclipse*, *GNOME*, *KDE*, *Linux Kernel.*

We used *Git* to control the changes performed to the project's source code and as a backup system, with 3 computers having an entire copy of the repository.

5.2.2 GNU C++ Compiler

GNU Compiler Collection or *GCC* [29] is a collection of front ends for different programming languages. One of them is the *g++*, which is the C++ compiler, and the main compiler of this project.

5.2.3 LLVM C++ Compiler

LLVM [30] is a collection of modular and reusable compiler and toolchain technologies. Like *GCC* it has different front ends, including one for the C++ language called *clang++*.

This compiler was used during the development, since it is the default compiler for the Mac OS X Mavericks operating system [31].

5.2.4 GNU Emacs

GNU Emacs [32] is the GNU implementation of the *Emacs* text editors family, the original version was released by Richard Stallman in 1985. Nowadays is one of the most powerful editors with over 2000 built-in commands that can be combined into macros for automate work.

We used this editor to edit *DGGA* files and to write this document.

5.2.5 GNU Make

GNU Make [33] is the GNU implementation of the *Make* utility, which automatically builds executable programs and libraries form source code by reading the build rules specified in files called *makefiles*.

We used this tool to build the *DGGA* executable without repeating the build sequence each time.

5.2.6 Eclipse

Eclipse [34] is an integrated development environment (IDE) written in Java. It was originally conceived as a Java editor, but thanks to its plug-in system it can be adapted to work with C++ projects in different platforms, and also interact with the *Git* system.

We used it at the beginning of the project, when the several utilities included in the system helped us identify the different parts of the original *GGA*.

5.2.7 GDB

GDB [35] or the GNU Debugger is a program that offers extensive facilities for tracing and altering the execution of a computer program. Initially written by Richard Stallman in 1986 as part of the GNU system is now included as a basic utility in some Linux distribution and other Unix-like Operating Systems.

We used it in very few cases to find the source of some estrange *DGGA* errors

5.2.8 Valgrind

Valgrind [36] is a programming tool for memory debugging, memory leak detection, and profiling. Its original author was Julian Seward and nowadays is a community project with multiple authors.

Its usage helped us to identify and fix some memory bugs and leaks on the original *GGA*.

5.3 Implementation details

This section provides a more detailed vision of the implementation of the different modules of *DGGA*.

The source code is distributed in two main directories *include/* and *src/* (see Chapter 7 for more details on the installation structure). The first one contains the header files, with extension *.hpp*, and the latter contains the source files, with extension *.cc*. Inside those directories, the new *DGGA* modules are located under the directory *dgga/*.

5.3.1 Master module

The Master module was designed in a way that it reuses the original *GGA* implementation supplanting only the GGASelector class.

Initially, the GGATournament class had an instance of GGASelector as a member. Since we only needed to modify the selection process, we replaced this member with an interface and created two implementations of that interface. One with the same behaviour as the old GGASelector but called GGALocalSelector, and another one that executes the selection process in distributed computers, transparently to GGATournament, called GGARemoteSelectorMaster. Therefore, when *DGGA* has to run as the original *GGA*, the main routine simply instantiates a GGATournament with the local selector, and likewise, if it has to run as *DGGA*, the main routine instantiates a GGATournament with the remote master selector.

Involved files: DGGARemoteSelectorMaster.hpp, DGGARemoteSelectorMaster.cc, GGASelector.hpp, GGALocalSelector.hpp, GGALocalSelector.cc, GGATournament.hpp, GGATournament.cc

5.3.2 Worker module

The Worker module was designed as a wrapper around the original GGASelector class.

When *DGGA* is executed to run as a Worker, it creates an instance of GGARemoteSelectorWorker (this is the wrapper), which has an instance of GGALocalSelector as a member, and starts the communication process with the Master through class DGGATcpConnection. Each time the wrapper receives the instruction to execute a mini-tournament, it translates the information received for the GGALocalSelector, which executes the tournament as if it was the original *GGA*. After the execution, the wrapper recovers the results and sends them back to the Master.

Involved files: DGGARemoteSelectorWorker.hpp, DGGARemoteSelector-Worker.cc, GGALocalSelector.hpp, GGALocalSelector.cc, GGATournament.hpp, GGATournament.cc

5.3.3 Parameters parsing

DGGA accepts several command line parameters as *GGA* (see Chapter 7.2 for a detailed list).

The *DGGA* parameter parsing differs substantially respect to the one in the original *GGA*. The original parsing has been reimplemented with *Boost program options*, which in addition of automatically parse the parameters, performs some type and range tests.

The set of parameters of *GGA* has also been extended with 6 new parameters to configure the distributed system:

- *master*: boolean flag to indicate that the program must act as the Master of *DGGA*.
- *worker*: boolean flag to indicate that the program must act as a Worker of *DGGA*.
- *ip <x.x.x.x>*: specifies an IP to connect to the Master. It can be used several times.
- *port <integer>*: specifies the port to connect to the Master.
- *nodes <integer>*: specifies the desired number of Workers.
- *start-workers-wrapper <file_path>*: specifies the wrapper used to start the Workers. The script specification is explained in Chapter 7.

Involved files: GGAOptions.hpp and GGAOptions.cc

5.3.4 Genome evaluation

This is one of the main steps of the whole configuration process. In *DGGA* we have introduced several changes respect to the original *GGA*.

Initially, *GGA* used a complicated structure of children and grandchildren processes in order to gather the performance of each parametrization. This also involved using shared memory and several interprocess synchronization mechanisms.

We could not afford that in *DGGA*, which in addition has to deal with network management, serialization and the synchronization of the distributed components. For this reason we reimplemented this part from scratch.

The new approach uses one thread and one child process per genome evaluation, removing the shared memory, reducing the interprocess synchronization part, and, since a thread is lighter than a process, reducing the load of the whole system.

Each time *DGGA* has to execute a performance test, it creates a new thread and then forks a child process that runs the target algorithm. Once the child has finished the performance is recovered through the system call `wait4()`, which provides access to the statistics of the process.

This part was implemented using the *Boost thread* library, because it offers a portable C++ like way to create and manipulate threads, and the *POSIX* functions to create and manage child processes: `fork()` and `wait4()`.

Files involved: GGARunner.hpp and GGARunner.cc

5.3.5 Specific Linux child process management

An important point, not covered in the original *GGA*, was the properly management of the child processes, used to evaluate the genomes, when their parent dies.

After some research, we found that there is no easy cross-platform solution to solve this problem. The most commented solution was based on periodically querying the parent process identifier. We can not implement this solution in *DGGA*, because we can not control the target binary behaviour.

Our approach is based on a Linux specific system call: `prctl()`. Using that function we specify to the Linux kernel that, whenever the parent of the child process dies, the process must automatically receive the signal *SIGKILL*, which can not be captured and its default behaviour is to terminate the process immediately.

Files involved: GGARunner.cc

5.3.6 Serialization

In order to interchange the necessary data between the different distributed components of *DGGA* we needed a way to transform the information in one computer's main memory, and recover it in another computer.

This task is performed by the *Boost serialization* library, which offers several options and formats to serialize data. In our implementation we used the default text-based format, that automatically transforms each C++ type into a predefined text representation and separates different data with white spaces.

The chosen format is not as easy to debug as an XML or JSON format and requires the data to be recovered in the same order as it is serialized but, is lighter than other text-based formats.

As a class is a user defined type, the library does not know how it must be serialized. To achieve that, the library offers an interface that must be implemented by each class that we want to serialize.

The code below, shows an example of a class extended to be serializable with *Boost serialization.*

```
class gps_position
{
private:
    // This grants the library access to the private data members
    friend class boost::serialization::access;

    // When the class Archive corresponds to an output archive, the
    // & operator is defined similar to <<.  Likewise, when the
    // class Archive is an input archive the & operator is
    // defined similar to >>.
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & degrees;
        ar & minutes;
        ar & seconds;
    }

    int degrees;
    int minutes;
    float seconds;

    // ... amazing code beyond this comment ...
};
```

Involved files: GGAInstance.hpp, GGAInstance.cc, GGAGenome.hpp, GGAGenome.cc, GGAParameter.hpp, GGAParameter.cc, GGAParameterTree.hpp, GGAParameterTree.cc, GGAValue.hpp, GGAValue.cc,

5.3.7 Communication

When testing different network communication systems, we found that most of them were too complicated to be combined with an already working system, or that they required a complex infrastructure. Finally, we decided to use the *Boost Asio* library, which offers a portable sockets API and a single-threaded asynchronous system based on events.

Another option was to use *MPI*, but we wanted the user to use *DGGA* as the original *GGA*. Therefore, the user must be able to use it without installing additional middleware. In addition, we wanted a system that uses the avaliable resources as soon as possible and recovers from possible faliures in some Workers, things that *MPI* does not allow.

The designed system detects 3 events: connection, message received and disconnection. All those events are controlled by an IO service object provided by *Boost Asio*, responsible of invoking the appropriated handler when an event occurs.

Above that infrastructure we have created different handlers in each side of the communication that match each state of the communication protocol. When a message is received, the current active handler processes the information and acts accordingly by: sending a response, executing another routine, storing information or changing the handler.

Involved files: DGGARemoteSelectorMaster.hpp, DGGARemoteSelectorMaster.cc, DGGARemoteSelectorWorker.hpp, DGGARemoteSelectorWorker.cc, DGGATcpAcceptor.hpp, DGGATcpAcceptor.cc, DGGATcpConnection.hpp, DGGATcpConnection.cc

5.3.8 Signal handling

One of the main obstacles of this project was the properly management of the system signals. The fact that a signal can interrupt the execution at any time, can produce unexpected behaviour if we ignore them. In addition, as the system runs several child processes to evaluate the performance of the competitive genomes, an unexpected signal can leave those processes as zombies wasting resources.

GGA used the simple signal handling mechanism provided by the C standard library: `signal()`, which does not offer enough guarantees for this project. The main problem of this approach is that multiple signals can interrupt each other leaving the handling routine in an unstable state.

Our solution was to use the *POSIX* replacement `sigaction()`, which offers a reliable signal management system. Allowing the developer to specify several restrictions before, during and after the signal handling routine.

Involved files: main.cpp

5.4 Fixed bugs

During the development of *DGGA*, since we tried to reuse as much as possible of the original *GGA*, we found some bugs that we had to fix. In what follows, we present the most representative bugs and how they have been fixed.

5.4.1 GGASharedMemory destructor

Fixed an error triggered by a map out of range access. The destructor routine has not taken into account that a map iterator is in an unstable state after removing it from the map.

5.4.2 GGAParameterTree destructor

Fixed a double memory free when deleting the parameter tree nodes using shared pointers. A shared pointers keeps a tree nodes alive while there is an active reference to it, and automatically deletes it when the last reference goes out of scope.

5.4.3 Target algorithm command parsing

Fixed several memory leaks when parsing the command line parameters for the target algorithm. The previous routine used C standard library functions like `strtok()` to parse each individual parameter. The new version based on regular expressions uses C++ objects, that automatically releases the memory when the parsing function goes out of scope.

5.4.4 GGAParameterTree XML parsing

Fixed several memory leaks when parsing the XML file with the parameter tree information. The leaks were automatically fixed when changing the library used to parse the XML, from the old C like solution, to the new C++ like version, which automatically releases all the allocated memory when the parsing function goes out of scope.

5.4.5 Evaluation results propagation

When a genome has been tested on an instance, the result is stored so the next time it is not necessary to run the test again. The error was that this propagation was performed without the penalty factor.

5.4.6 CPU timeout per evaluation

When an evaluation is executed, it is supposed that the user has configured the target algorithm to finish its execution after an specified amount of seconds.

In order to avoid running the evaluation indefinitely, due to a user miss configuration, we have integrated a CPU timeout in *DGGA*, with a grace period of 30 seconds, that kills the execution process when it has exhausted the available time.

Chapter 6

Experimental evaluation

In this chapter, we present an intensive experimental investigation on the soundness, robustness and performance of *DGGA*. Our target algorithm, in order to carry this experimental evaluation, is a SAT solver.

We run our experiments on a cluster featured with Intel Xeon CPU E5-2620 @ 2.00GHz processors and a memory limit of 3.5 GB. Each machine runs an instance of Linux 2.6.32 and the compiler used to build *DGGA* was *gcc 4.4.6*, and we used 11 of the 12 CPU cores to evaluate the genomes and 1 to carry the communication.

6.1 Stress tests

In this section, we present the tests we conducted to evaluate the robustness of the design and implementation of *DGGA*.

We have performed the following stress tests:

- Run *DGGA* while no one is using the cluster, without forcing any error.
    - **Expected behaviour**: the Master process starts correctly and spawns the specified amount of Workers. As the Workers are executed by the scheduler, the tuning process starts and finishes a few hours later without problems.
- Run *DGGA* while other users are using the cluster, without forcing any error.

  - **Expected behaviour**: the Master process starts when the cluster has enough resources. After this, it proceeds requesting the specified amount of Workers. However, since there are not enough resources for all of them, only some of the Workers start their execution. The tuning process finishes. i.e., the Master has the results. However, notice that only a fraction of the Workers have conducted the whole process while the rest have remained in the queue. At some point these remaining Workers will be accepted for execution in the cluster. Then, they will try to connect to the Master that has already finished its execution. At this point, they detect the Master is no longer alive and they finish safely.

- Run *DGGA* while no one is using the cluster and some Workers are killed randomly.

  - **Expected behaviour**: the Master proceeds as in the first test. After some time, we kill some of the Workers using the *qdel* command. The Master detects the disconnections, rolls-back the mini-tournaments and requests new Workers. The rolled-back data is then sent to another Worker.

    Finally, after having repeated this sequence 3 times, the configuration process finishes as in the first test.

- Run *DGGA* while other users are using the cluster and some Workers are killed randomly.

  - **Expected behaviour**: The Master process starts correctly and requests the specified amount of Workers. Initially, the cluster is busy and all the Workers are pending to be executed. After a while, the jobs of the other users finish and the Workers start their execution. Then, the other users enqueue more jobs and we kill some Workers to let them execute their jobs. After killing the Workers the Master proceeds as in the second test, requesting more Workers. Once again, when the jobs of the other users finish, the last requested Workers start their execution.

    Finally, the system finishes the execution properly.

- Run *DGGA* and after having the system working, kill the Master.

  - **Expected behaviour**: The Master process starts and spawns the requested amount of Workers. After some time and before the configuration process has finished, we kill the Master. Then, each Worker detects that the Master is no longer alive and they finish safely.

In this case we can not get the results, but the system releases all the resources as soon as the anomaly is detected.

After performing these test we can conclude that *DGGA* is able to recover from different errors as long as the Master is alive.

6.2 Automatic configuration of a SAT Solver

In this section, we analyse the soundness and performance of *DGGA* configuring the SAT solver *lingeling* [37]. The data sets used to conduct this experiments are part of the Algorithm Configuration Library (ACLIB) [38].

In order to compare different aspects of the configuration process, we have configured *lingeling* with two different evaluation timeouts, leaving all the other parameters with their default values (see Chapter 7.2.1). In particular we have configured *lingeling* for 30 and 300 seconds. Tables 6.1 and Table 6.2 show the results achieved by the default parametrization of *lingeling* ("default *lingeling*" at tables) versus the parametrizations found for each set of instances after tuning *lingeling* with *DGGA* (tuned *lingeling*).

In Table 6.1 we can see that the configured version with a timeout of 30 seconds, outperforms the default one in 13 of the 16 data sets within the same timeout. In 7 sets of instances the tuned *lingeling* is able to solve more instances, being this particularly dramatic for sets: circuit fuzz, K3 v300-c1279 and K3 v275-c11172. In many of the sets where the default *lingeling* already solves all the instances, the tuned *lingeling* is able to improve the average run time, being this particularly dramatic on sets: SWV calysto GZIP v1.2.4 and SWV calysto XINETD v2.3.14, where we can see speed-ups of more than one order of magnitude.

Despite of the good results for the majority of the data sets, there are three sets that have not been improved. After reviewing the logs of the configuration process, we suspect that this performance degradation is related to the instance selection module. The issue is that not all the instances in a test set are used during the tuning process and some may be repeated too often among generations. This is certainly something to be improved and has to do with a better balance of the instances selection.

In Table 6.2 we can observe that the version configured with a timeout of 300 seconds outperforms clearly the default *lingeling* as in the experiment with 30 seconds timeout. The achieved results are similar to the ones in the 30 seconds test, the tuned *lingeling* has solved more instances and when both versions have solved all the instances for a particular set, the tuned *lingeling* is faster.

Instance set		# Instances	tuned *lingeling*	default *lingeling*
Circuit fuzz		884	**4,94(684)**	7,61(649)
CSSC regression tests		171	**0,01(171)**	0,02(169)
LABS		701	2,20(441)	**3,13(444)**
UNSAT unif K5		600	**0,94(600)**	1,81(600)
UF250		100	3,58(100)	**2,49(100)**
K3	v200-c853	100	**0,34(100)**	0.53(100)
	v225-c960	100	**0,96(100)**	1,41(100)
	v250-c1066	100	**3,10(100)**	5,57(100)
	v275-c1172	100	**8,66(98)**	11,67(80)
	v300-c1279	100	**13,09(70)**	10,41(42)
	v325-c1385	100	13,00(18)	**12,56(61)**
SWV calysto	DSPAM v3.6.5	100	**0,14(100)**	1,14(100)
	GZIP v1.2.4	90	**0,05(90)**	13,32(89)
	HSAT	279	**0,19(279)**	1,92(274)
	WINE v0.9.27	80	**0,30(80)**	1,81(80)
	XINETD v2.3.14	55	**0,38(55)**	10,75(54)
Total		3660	**3086**	3042

Table 6.1: Solved instances in 30s. Average time in seconds and (amount of solved instances)

The main improvement in this experiment is that *DGGA* has been able to find a configuration for the *K3* data set, that is better than the default one in all the *K3* subsets. That leads us to think, that the specific sub set that is improved in contrast with the 30 seconds results, has instances that *lingeling* is not able to solve in 30 seconds, no matter which parametrization we use. Hence *DGGA* is not able to find any good configuration for that subset with a timeout of 30 seconds.

Even though we have set a timeout ten times bigger for this experiment, there are two sets of instances that *DGGA* is still not able to configure effectively, clearly showing that there are some points where *DGGA* needs to be improved since it is seeded initially with the default configuration fro *lingeling*

Instance set		# Instances	tuned *lingeling*	default *lingeling*
Circuit fuzz		884	**18,78(783)**	21,15(776)
CSSC regression tests		171	**0,03(171)**	0,01(169)
LABS		701	**14,64(501)**	14.84(499)
UNSAT unif K5		600	**0,94(600)**	1,76(600)
UF250		100	**1,48(100)**	2.49(100)
K3	v200-c853	100	**0,41(100)**	0,50(100)
	v225-c960	100	**1,07(100)**	1,41(100)
	v250-c1066	100	**3,52(100)**	5,64(100)
	v275-c1172	100	**8,90(100)**	18,31(100)
	v300-c1279	100	**23,84(100)**	50,10(100)
	v325-c1385	100	**85,31(100)**	107,53(61)
SWV calysto	DSPAM v3.6.5	100	**0,12(100)**	1,14(100)
	GZIP v1.2.4	90	**0,04(90)**	13,32(89)
	HSAT	279	**0,17(279)**	2,90(279)
	WINE v0.9.27	80	**0,27(80)**	1,99(80)
	XINETD v2.3.14	55	**0,37(55)**	12,25(55)
Total		3660	**3359**	3308

Table 6.2: Solved instances in 300s. Average time in seconds and (amount of solved instances)

Chapter 7

DGGA installation and execution guide

This chapter describes how to perform the installation of *DGGA* and shows an execution example of the tool on the SAT solver *lingeling* [37].

7.1 Installation guide

In this section, we show how to perform a local installation of *DGGA*. First of all, we list the hardware and software prerequisites needed for the installation. Then, we proceed on how to obtain *DGGA*'s source code, install the necessary libraries and finally, build *DGGA*.

7.1.1 Prerequisites

To install the software you need a computer with at least 1 GB of free disk space running a Unix-like operating system, a bash-compatible shell, and the following standard command-line development tools:

- A **C++ compiler**, preferably the GNU g++ [29] compiler or any other that already delivers C++11 STL features in C++03 mode. This tool is the responsible of transforming source code written in C++ into an executable. All the source code of *DGGA* is written in C++ and no other compiler is needed.

- A **Make** tool, with the same extensions than GNU make [33] in its version 3.8 or later. **Make** is a utility that automatically builds executable programs and libraries from source code.

- A command line version of the **Git** client [28] in its version 0.99 or later. **Git** is a distributed revision control and source code management system. This is the system we have used to develop *DGGA*.

7.1.2 DGGA distribution

DGGA is publicly available through a Git repository. To get a copy of the last version, open a shell and type:

```
$ git clone https://jponf@bitbucket.org/jponf/dgga.git
```

The previous command will download the last version of *DGGA* into a new directory called *dgga* into the working directory. This is the directory structure of *dgga*:

```
dgga
├── include
│   └── dgga
│       └── net
├── src
│   └── dgga
│       └── net
└── examples
    └── clasp
```

The source code of *DGGA* is located in sub-directories *include/* and *src/*. Directory *include/* contains all the *.hpp* files with the declaration of the classes, methods and functions. Directory *src/* contains the declarations in *.cc* files. See section ... for further information.

Directory *examples/* contains several XML files which describe solvers to tune and their parameters (see Section 7.2.2). In particular, we can find the following solvers: *satenstein* [39] and *clasp* [40]

7.1.3 Third party libraries

The DGGA distribution makes use of several third party libraries of the *Boost* project. Here we show how to perform a local installation of the *Boost* libraries, in case they are not already in your system. These libraries will be installed inside the DGGA directory (see previous Section).

DGGA has been developed with *Boost* 1.54, and it is the one covered in this guide. However, any version above 1.45, meets the requirements for this project.

Before starting the installation you must create a new directory called *lib* inside *DGGA*'s main directory. This is where we will install the *Boost* libraries after building them.

```
[/path/to/dgga]$ mkdir lib
```

7.1.4 How to get Boost

The most reliable way to get a copy of *Boost* is to download a distribution from SourceForge [41]. In particular, we have to follow these steps:

1. Select any version 1.XX of *Boost* above 1.45 from `http://www.boost.org/users/history`.
2. Download the file *boost_1_XX_X.tar.bz2*.
3. Move into the *DGGA* directory and execute:
   - `$ cd dgga`
   - `$ tar -xjvf /path/to/boost_1_XX_X.tar.bz2`

How to build Boost

At this point we have all the required elements to build the set of *Boost* libraries. This guide only covers the installation of the static version of the libraries. If you choose to build the shared version, you may need to modify the environment variable: `LD_LIBRARY_PATH`, to let the final *DGGA* binary find dynamically the libraries at run time.

To build the libraries simply execute the following commands in the specified order:

- `$ cd` *`/path/to/`*`dgga/boost_1_XX_X`
- `[/path/to/dgga/boost_1_XX_X]$ ./bootstrap.sh --prefix=‘pwd‘/build`
- `[/path/to/dgga/boost_1_XX_X]$ ./b2 link=static`
- `[/path/to/dgga/boost_1_XX_X]$ ./b2 install`
- `[/path/to/dgga/boost_1_XX_X]$ mv build/include/boost ../include`
- `[/path/to/dgga/boost_1_XX_X]$ mv build/lib/libboost*.a ../lib`

7.1.5 How to build DGGA

The installation is quite straightforward since all the steps needed to compile and install *DGGA* are specified into a *Makefile*.

If you also have followed the previous section or your system has the *Boost* libraries in the default compiler paths, there is no need to change anything. Otherwise, you may need to modify lines 41 to 47 of the Makefile, to instruct the compiler where the *Boost* libraries are located.

The last step before building *DGGA* is to choose which version we want. There are two versions: debug and release. As the name suggests, the debug version is used to test and develop the tool. For production purposes, use the release version since it can be about 10 times smaller.

Finally, the building process consists in moving to the *DGGA*'s directory and execute make.

- `$ cd` *`/path/to/`*`dgga`
- For the debug version: `[/path/to/dgga]$ make debug`
- For the release version: `[/path/to/dgga]$ make release`

After executing the previous commands you will find a new directory, *build/*, into *DGGA*'s directory. The structure of the directory may be different depending on the selected version but a generic view should be this one:

```
dgga
├── build
│   ├── debug
│   │   ├── bin
│   │   └── obj
│   └── release
│       ├── bin
│       └── obj
└── ...
```

The *DGGA* binary for the specified version is located into the corresponding bin directory (*<version>/bin/*).

7.2 Execution guide

This section contains a simple explanation of how to execute *DGGA*. First of all, we discuss the execution parameters of DGGA. Then, we present the configuration files needed to describe the configuration or execution parameters of the target algorithm and the set of instances used to evaluate the performance of a particular genome or configuration.

Next, we describe both how to adapt *DGGA* and the target algorithm on a particular distributed execution environment. This task is in charge of what we call execution environment wrappers.

Finally, we introduce an example of how to execute *DGGA* to tune the SAT solver *lingeling* [37] in a Rocks Cluster distribution.

7.2.1 DGGA execution parameters

DGGA inherits all the execution parameters of *GGA* extended with six new parameters. In what follows, we will show how to execute *DGGA*, and the list of parameters with a brief explanation of each one.

```
$ ./dgga <param_tree_file> <instances_seed_file> [parameters]
```

The first two parameters are mandatory and must be specified in the given order:

- `<param_tree_file>`: the path to the XML configuration file defining the parameter tree. See next section for a detailed explanation.

- `<instances_seed_file>`: the path to the instances seed file with the instances to tune the algorithm. See an example in next section.

The rest of the parameters are optional:

- `-g/--generations` *arg*: maximum number of generations (default value: 100).
- `-p/--population_size` *arg*: size of the initial population. This can vary over the course of the algorithm, but generally doesn't stray more than 25% (default value: 100).
- `-t/--num_threads` *arg*: mini-tournament size. Number of members to run simultaneously (default value: 4).
- `-w/--pct_winners` *arg*: percentage of winners per mini-tournament [0.0, 1.0] (default value: 0.125).
- `--is` *arg*: number of instances at the start of the tuning (default value: 5).
- `--ie` *arg*: number of instances at the end of the tuning (default value: 100).
- `--gf` *arg*: generation at which to reach the amount of instances specified with the parameter `--ie` (default value: -1).
- `--seed` *arg*: seed for the internal random engine (default value: current time in seconds).
- `--seeded_genomes` *arg*: number of "seeded genomes" to create from some set of default parameters (default value: 0).
- `--pe` *arg*: penalty applied to the evaluation of those genomes that reach the timeout time per instance (default value: 1.0).
- `--max_evals` *arg*: maximum number of evaluations allowed to find the best configuration. This is not strict, but a best effort will be made to come close to this number (default value: 2147483647).
- `-m/--mutation_rate` *arg*: the probability that a parameter is mutated when generating new individuals (default value: 0.1).
- `--st` *arg*: sub-tree split probability [0.0, 1.0] (default value: 0.1).
- `--sp` *arg*: sigma percentage. Determines sigma for the Gaussian distribution of the random engine (default value: 1.0).
- `-v/--verbosity` *arg*: verbosity level [0, 5] (default value: 2).

- `--ga` *arg*: maximum age of a genome (default value: 3).
- `--rt` *arg*: if false, *DGGA* tunes for output, rather than runtime. The last line of the algorithm output must be the objective value to minimize [true, false] (default value: true).
- `--nc` *arg*: if true, *DGGA* normalizes all the continuous variables (default value: false).
- `--su1` *arg*: if true, sends SIGUSR1 before sending SIGTERM to kill the evaluation processes (default value: false).
- `--ls` *arg*: specifies the learning strategy (default value: 1). {0 = TESTING, 1 = Linear, 2 = Step, 3 = Parabola, 4 = Exponential}
- `--lsd` *arg*: generation at which start the learning strategy. Until then `--is` instances will be used (default value: 0).
- `--lss` *arg*: number of generations per step for the step strategy (default value: 5).
- `--tacl` *arg*: target algorithm CPU timeout in seconds (default value: 30).
- `--tc` *arg*: tuner CPU timeout in seconds (default value: 2147483647).
- `--twc` *arg*: tuner wall-clock timeout in seconds (default value: 2147483647).
- `--conf_file` *arg*: specifies a configuration file. Command line options are preferred (default value: "").
- `--traj_file` *arg*: path of the trajectory file (default value: "").
- `--scen_file` *arg*: specifies a scenario file as used by ParamILS/SMAC. Note: this file overrides certain command line options (default value: "").
- `--master`: the presence of this flag indicates that the process must act as the Master of *DGGA*.
- `--worker`: the presence of this flag indicates that the process must act as a Worker of *DGGA*.
- `--ip` *arg*: specifies an IP to connect to the Master. It can be used several times to specify more than one IP.
- `--port` *arg*: specifies the port to connect to the Master (default value: 6789).
- `--nodes` *arg*: specifies the desired number Workers. It is mandatory when "--master" is specified.
- `--start-worker-wrapper` *arg*: specifies the path to the wrapper that the Master will use to start the Workers (default value: "").
- `-h/--help`: prints the help message and exits.

7.2.2 Tuning configuration files

DGGA depends on two main configuration files. A XML configuration file, with the configuration of the target algorithm, and an instances seed file, with the list of instances to tune the algorithm.

XML configuration file

This file provides to *DGGA* all the necessary information about the target algorithm and its configuration or execution parameters. It also provides user defined parametrizations and the command to run the algorithm. The format of this file, as mentioned before, is XML

Below, we show and describe the structure of the XML file:

```xml
<algtune>

  <cmd>/path/to/algorithm_binary $instance $seed $cutoff
                                  $param1 $param2 $param3 $param4
  </cmd>

  <seedgenome>
    <variable name="root" value="0" />
    <variable name="param1" value="3" />
    <variable name="param2" value="b" />
    <variable name="param3" value="5.654" />
    <variable name="param4" value="12" />
  </seedgenome>

  <node type="and" name="root" start="0" end="0">
    <node type="and" name="param1" prefix="--param1=" start="0" end="4"/>
    <node type="or" name="param2" prefix="--param2=" categories="a,b">
      <node type="and" name="param3" prefix="--param3=" start="5.5" end="7.5"/>
      <node type="and" name="param4" prefix="--param4=" start="10" end="20"/>
    </node>
  </node>

  <forbidden>
    <forbid>
      <setting name="param1" value="0" />
      <setting name="param2" value="a" />
    </forbid>
    <forbid>
      <setting name="param3" value="6.0" />
      <setting name="param4" value="15" />
    </forbid>
  </forbidden>
</algtune>
```

All the specification of the target binary is enclosed within the *<algtune></algtune>* tags.

The second pair of tags of the file, *<cmd></cmd>*, contain the command that DGGA must use to execute the target algorithm. The variables *$instance, $seed* and *$cutoff* are special variables that tell DGGA where to put: the path to the instance, the seed and the target binary cutoff. If you do not need one of them for your algorithm, you can simply omit it. The rest of the parameters ($param1 - $param4) correspond to parameters to be tuned.

Next, the *<seedgenome></seedgenome>* tag specifies settings of the parameters to insert into the initial population. Each *<variable />* tag, specifies the name of a parameter and the value for it to take in the genome.

The next portion of the file specifies the parameter tree itself. The tree is specified by a single node corresponding to the root of the tree. All node tags that are not a child of this root node will be ignored.

Each <node> tag my have any number of nodes underneath it. <node>s may have one of two types: "and" or "or", which will be described in a moment. Each node is either categorical, discrete or continuous. Categorical nodes have the *"categories"* attribute specified (see *param2*). Discrete parameters have "start" and "end" specified with integer values (see *param1* and *param4*). Floating point parameters are specified like discrete parameters, except with floating point numbers in start and end (see *param3*). Prefix indicates what text to prefix to the parameter in the command line, and is optional.

As mentioned before a node can be either an "and" node or an "or" node. An "and" node indicates that the child nodes should all be present whenever the parent node is present in the parameter settings of a genome. An "or" node means that only the node corresponding to the selected branch should be included in the settings. In other words, or nodes allow users to select between categorical parameters, and associate a branch of the parameter tree with the parameter. This relationship is used within the optimization to find good parameters. In the example above, *param2* has two settings: 'a' and 'b'. Setting 'a' is associated with the branch of *param3* and setting 'b' with *param4*.

The final part of the XML file specifies which parameter combinations are forbidden to be used together. Each *<forbid></forbid>* block specifies a set of parameters that may not be found together. In the example above, *param1* may not be 0 at the same time that *param2* is 'a' (and vice-versa); and *param3* may not be 6.0 when *param4* is 15. Notice that as floating point values are not represented exactly, it is unlikely that the constraint will ever be hit.

Instances seed file

This file contains lines with pairs of values, the first element is the seed of the instance and the second is the path to the instance file. These values correspond to the variables *$seed* and *$instance* mentioned in the previous point.

Below, we show the structure of this file with three instances:

```
1425435 /path/to/one_instance
68764 /path/to/another_instance
35634 /path/to/yet_another_one
```

Notice that if the path to the instance is a relative path, then, it must be relative to the *DGGA* binary rather than from the directory where this file is stored. Our recommendation is to always use absolute paths.

7.2.3 Execution environment wrappers

DGGA can not modify the behaviour of the target algorithm and does not know how to start a Worker in an specific architecture. For these reasons the user has to create wrappers to adapt *DGGA* and the target algorithm.

The language used to create the wrappers does not matter as long as it is executable like a native binary. Since the wrappers do not have to perform resource consuming tasks, we recommend to use a scripting language such as: Bash [42] or Python [43].

Worker execution wrapper

This wrapper is in charge of the execution of the Workers, and for this reason it is mandatory for the proper execution of *DGGA*. Whenever the Master needs to request a Worker, it will execute this wrapper.

The wrapper must accept as parameters the number of CPU cores required by the Worker and the command line used to start *DGGA* as a Worker. In addition, the Master expects that the wrapper returns 0 to indicate that there has been no error, and any other value on error.

Target algorithm wrapper

This wrapper is used to adapt the target algorithm to some restrictions imposed by both the user and *DGGA*. It is optional, only those algorithms that do not meet the necessary requirements will need a wrapper.

Below, we show an example of a Bash wrapper that filters the input, limits the CPU time and the virtual memory, and filters the output for an out of memory error.

```bash
#!/usr/bin/env bash

# The first and second parameters are the CPU timeout and the
# instance. We specified that using the $cutoff and the $instance
# variables in the XML configuration file.
cpulimit=$(($1+5))
problem=$2
shift 2       # Removes $1 and $2 from $@.

cmd="/path/to/binary --input=$problem $@"

ulimit -v 2097152    # Limits memory to 2GB
ulimit -t $cpulimit # Limits the CPU time

bad_alloc=$($cmd 2>&1 | grep "std::bad_alloc")

if [ -n "$bad_alloc" ]; then
  # This line tells DGGA that the evaluation has failed. When printed,
  # DGGA will treat the evaluation as if it has not finished in time.
  printf "Result for: crashed"
fi
```

7.2.4 Example: automatic configuration of Lingeling

In this example, we show the different steps required to tune the SAT solver *lingeling* [37] using *DGGA*. First, we will present a reduced version of the tuning configuration files. Then, we will present the execution environment wrapper for the solver, and finally the execution script to launch *DGGA* on high performance computing cluster as a Sun Grid Engine job (see Chapter 2).

We assume that the user has a directory `/home/user/dgga` where we will put the following files:

- *dgga*: the *DGGA* binary.
- *qsub_script.sh*: we will use this file to launch the Master job into the cluster.

- *start_worker_wrapper.sh*: the wrapper responsible of starting the Workers.
- *lingeling_wrapper.sh*: the wrapper responsible to set lingeling time and memory restrictions.
- *config/lingeling.xml*: the target algorithm XML configuration file.
- *config/instances.txt*: the instances seed file.
- *worker_logs/*: directory where we will put the output of the Workers.
- *dgga_tuning_logs/*: directory where we will put the output of the Master.

XML configuration file

For the sake of readability, we have reduced the number of parameters that we will tune in this example to 10. Note: *lingeling* can be configured with 240 parameters.

config/lingeling.xml:

```
<algtune>
  <cmd>/home/user/dgga/lingeling\_wrapper.sh $cutoff
       -f $elmreleff $lkhdmisifelmrtc $elim
       $move $rstinoutinc $unhdextstamp $deco
       $cardocclim $liftwait $rephaseinc $instance
  </cmd>

  <!-- default parameters values -->
  <seedgenome>
    <variable name="__dummy__root__" value="0" />
    <variable name="elmreleff" value="200" />
    <variable name="lkhdmisifelmrtc" hint="categorical" value="0" />
    <variable name="elim" hint="categorical" value="1" />
    <variable name="move" hint="categorical" value="2" />
    <variable name="rstinoutinc" value="110" />
    <variable name="unhdextstamp" hint="categorical" value="1" />
    <variable name="deco" hint="categorical" value="2" />
    <variable name="cardocclim" value="100" />
    <variable name="liftwait" hint="categorical" value="2" />
    <variable name="rephaseinc" value="10000" />
  </seedgenome>

  <node type="and" name="__dummy__root__" start="0" end="0" >
    <node type="and" name="elmreleff" prefix="--elmreleff=" start="0" end="10000" />
    <node type="and" name="lkhdmisifelmrtc" prefix="--lkhdmisifelmrtc="
         categories="0,1" />
    <node type="and" name="elim" prefix="--elim=" categories="0,1" />
    <node type="and" name="move" prefix="--move=" categories="0,1,2,3" />
    <node type="and" name="rstinoutinc" prefix="--rstinoutinc=" start="1" end="1000"
         />
    <node type="and" name="unhdextstamp" prefix="--unhdextstamp=" categories="0,1" />
    <node type="and" name="deco" prefix="--deco=" categories="0,1,2" />
    <node type="and" name="cardocclim" prefix="--cardocclim=" start="0" end="2147483647" />
    <node type="and" name="liftwait" prefix="--liftwait=" categories="0,1,2" />
    <node type="and" name="rephaseinc" prefix="--rephaseinc=" start="1" end="2147483647" />
  </node>
</algtune>
```

Notice that in the *<cmd></cmd>* part, we are executing the file `lingeling_wrapper.sh`, which we will present later, instead of the *lingeling* binary.

Instances seed file

For this example, we have used the K3 set of instances of the ACLIB [38]. We have assumed that these instances are located in `/home/user/dgga/seeds`. Then, the instances seed file should look like:

config/instances.txt:

```
...
1258 /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1172-797-S58452150.cnf
5689 /home/user/dgga/seeds/K3/K3-inst/k3-v250-c1066/unif-v250-c1066-895-S168722514.cnf
941231 /home/user/dgga/seeds/K3/K3-inst/k3-v250-c1066/unif-v250-c1066-912-S1767016736.cnf
32165 /home/user/dgga/seeds/K3/K3-inst/k3-v300-c1279/unif-v300-c1279-365-S2037425360.cnf
985475 /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-13-S666309010.cnf
86484 /home/user/dgga/seeds/K3/K3-inst/k3-v225-c960/unif-v225-c960-51-S1204466261.cnf
849435 /home/user/dgga/seeds/K3/K3-inst/k3-v325-c1385/unif-v325-c1385-606-S441587708.cnf
45612 /home/user/dgga/seeds/K3/K3-inst/k3-v300-c1279/unif-v300-c1279-35-S1850770749.cnf
48921 /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1172-51-S826466208.cnf
1356 /home/user/dgga/seeds/K3/K3-inst/k3-v225-c960/unif-v225-c960-445-S1858222715.cnf
98465 /home/user/dgga/seeds/K3/K3-inst/k3-v225-c960/unif-v225-c960-320-S2002322624.cnf
5321 /home/user/dgga/seeds/K3/K3-inst/k3-v250-c1066/unif-v250-c1066-790-S458306906.cnf
549445 /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-777-S900373011.cnf
54312/home/user/dgga/seeds/K3/K3-inst/k3-v325-c1385/unif-v325-c1385-215-S153041018.cnf
...
```

Worker execution wrapper

Below, we show the wrapper that we used to start the Workers in our cluster with Sun Grid Engine. The wrapper sets a timeout of 4 days and puts the information printed by the Worker into: `/home/user/dgga/worker_logs`.

start_worker_wrapper.sh:

```
#!/usr/bin/env bash

# Expected input: <ncores> <binary_path> <dgga_parameters>
ncores=$1
```

```
binary=$2
shift 2

qsub_cmd="qsub -pe smp $ncores  -l h_cpu=345600 -V -cwd -q medium.q
                    -N dgga_worker -o /home/user/dgga/worker_logs
                    -e /home/user/dgga/worker_logs"

echo "$binary $@" | $qsub_cmd
```

Target algorithm wrapper

The target algorithm wrapper, that we used in this example, filters the received parameters and sets the time and memory limits before executing the target algorithm. This is the file `lingeling_wrapper.sh` that we used in the *<cmd></cmd>* part of the XML configuration file, also notice, that the received parameters will change depending on what we write on that part of the XML configuration file.

lingeling_wrapper.sh

```
#!/usr/bin/env bash

# See the XML configuration file to check the order of the parameters
binary=$1
# We add a grace period of 15 seconds to compensate the time consumed
# by the script
cutoff=$(($2+15))
shift 2 # Removes $1 and $2 from the list of parameters

ulimit -v 2097152 # 2 GB
ulimit -t $cutoff

$binary $@   # Executes the binary with the rest of the parameters
```

DGGA execution in Sun Grid Engine

Finally, the last step is to launch *DGGA* and wait for the results. This step may vary depending on the cluster's Job Scheduler. In the particular case of Sun Grid Engine, a user must use the *qsub* command to launch a job. We have used the following script to launch *DGGA* in our cluster, using the command:

```
$ qsub qsub_script.sh
```

Where *qsub_script.sh* is:

```
#!/bin/sh

## Command interpreter for this job
#$ -S /bin/bash

## Amount of threads/jobs in the same node
#$ -pe smp 1

## This line specifies the job queue. It is strongly related to the
## cluster's configuration, in our case long.q does not impose any
## timeout to the jobs
#$ -q long.q

## Directories where to put the output of the program. In this example
## we have created a directory called dgga_tuninglogs.
#$ -o /home/user/dgga/dgga_tuninglogs
#$ -e /home/user/dgga/dgga_tuninglogs

## Finally, the command line that executes the job.

/home/user/dgga/dgga /home/user/dgga/lingeling.xml /home/user/dgga/k3.
    txt --master --port 12345 --nodes 2 --start-worker-wrapper /home/
    user/dgga/start_worker_wrapper.sh -p 100 -g 100 -t 11 --gf 75 --tac
     300  -v 5 > /home/user/dgga_tuninglogs/lingeling_k3_300.txt
```

After the *qsub* execution, we will have to wait until the scheduler executes the job. Then, if we look at the output of the DGGA, which in the case of this example is in the file *dgga/dgga_tuninglogs/lingeling_k3_300.txt*, we will see the following:

```
[0.000000] runDGGA
[0.000000] DGGA MASTER
[0.000000] PROGRAM OPTIONS:
[0.000000] Program name: /home/user/dgga/dgga
[0.000000] Parameter Tree file: /home/user/dgga/lingeling.xml
[0.000000] Instance Seed file: /home/user/dgga/instances.txt
[0.000000] ---
[0.000000] Generations: 100
[0.000000] Population size: 100
[0.000000] Num. Threads: 11
[0.000000] Pct winners: 0.125
[0.000000] # Instances start: 5
[0.000000] # Instances end: 100
[0.000000] Gen inst finish: 75
[0.000000] Seed: 1409240695
[0.000000] # Seeded Members: 0
[0.000000] Cutoff penalty multiplier: 1
[0.000000] Max obj evals: 2147483647
...
[0.230000] (GGAParameter)[Name: __dummy__root; Type: Discrete; Start/End: 0/0]
[0.230000] (GGAParameter)[Name: wait; Type: Categorical; Domain: {0, 1}]
[0.230000] (GGAParameter)[Name: unhidewait; Type: Categorical; Domain: {0, 1, 2}]
[0.230000] (GGAParameter)[Name: unhide; Type: Categorical; Domain: {0, 1}]
[0.230000] (GGAParameter)[Name: unhdroundlim; Type: Discrete; Start/End: 0/100]
[0.230000] (GGAParameter)[Name: unhdreleff; Type: Discrete; Start/End: 0/10000]
[0.230000] (GGAParameter)[Name: unhdmineff; Type: Discrete; Start/End: 0/2147483647]
[0.230000] (GGAParameter)[Name: unhdmaxeff; Type: Discrete; Start/End: -1/2147483645]
[0.230000] (GGAParameter)[Name: unhdlnpr; Type: Discrete; Start/End: 0/2147483647]
[0.230000] (GGAParameter)[Name: unhdhbr; Type: Categorical; Domain: {0, 1}]
[0.230000] (GGAParameter)[Name: unhdextstamp; Type: Categorical; Domain: {0, 1}]
[0.230000] (GGAParameter)[Name: trnrmineff; Type: Discrete; Start/End: 0/2147483647]
[0.230000] (GGAParameter)[Name: trnrmaxeff; Type: Discrete; Start/End: -1/2147483645]
[0.230000] (GGAParameter)[Name: trnreleff; Type: Discrete; Start/End: 0/1000]
...
[0.480000] TOURNAMENT START
[0.480000] Start generation 1
```

```
[0.560000] Generation 1 population: [GGAPopulation (101)]
[Competitive (58)]
[GGAGenome: 0x28fbb70; Gender: C; Age: 2; Genome: {__dummy__root: 0; elmreleff: 123; lkhdmisifelmrtc...
...
0.560000] [GGALearningStrategyLinear] Using 6 instances.
[0.560000] nextGeneration(): 1 | 22242 | /home/user/dgga/seeds/K3/K3-inst/k3-v300-c1279/unif-v300-c1...
[0.560000] nextGeneration(): 1 | 21289 | /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c85...
[0.560000] nextGeneration(): 1 | 24341 | /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c85...
[0.560000] nextGeneration(): 1 | 27334 | /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1...
[0.560000] nextGeneration(): 1 | 31069 | /home/user/dgga/seeds/K3/K3-inst/k3-v325-c1385/unif-v325-c1...
[0.560000] nextGeneration(): 1 | 28376 | /home/user/dgga/seeds/K3/K3-inst/k3-v300-c1279/unif-v300-c1...
[0.570000] Tournament sizes for this generation: [9, 10, 10, 10, 10, 9]
...
[53.060000] Recovering tournament results
[53.080000] Recovering tournament results
[53.090000] Recovering tournament results
[53.110000] Generation 53 most fit Obj: 471.566
[53.110000] Generation 53 most fit parameters: [GGAGenome: 0x1539770; Gender: C; Age: 1; Genome: {__d...
[53.120000] Generation 53 number of evaluations so far: 85416
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1172-676-S...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-15-S595...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v325-c1385/unif-v325-c1385-407-S...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1172-112-S...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-825-S16...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v275-c1172/unif-v275-c1172-777-S...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-594-S11...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v250-c1066/unif-v250-c1066-972-S...
[53.120000] (Winner Performance) /home/user/dgga/seeds/K3/K3-inst/k3-v200-c853/unif-v200-c853-108-S59...
...
```

The previous log file is composed by different extracts of the one we get after executing *DGGA* in our cluster. The first part, is the list of *DGGA* parameters, with their associated values. Next, we can see the list of parameters that we have specified in the XML configuration file. After the list of parameters, we can se the beggining of the tournament and the population of the first generation. Then, we can see the instances selected to evaluate the genomes and the sizes of the mini-tournaments. The last part of this extract shows that the Master is recovering the results of the evalutions, the winner of the current generation (53) and its performance.

Finally, when the execution of *DGGA* finishes, in our case at generation 80 because the improvement between generation 79 and 80 is lower than a fixed threshold, we can see the best configuration that *DGGA* has found at the end of the output file. The line "Final most fit command: ...", contains the command generated using the information of the genome that has achieved the best performance.

```
...
84.460000] End generation 80
[84.470000] Stopping because: below improvement threshold
[84.470000] Final most fit Obj: 961.58
[84.470000] Final most fit parameters: [GGAGenome: 0x7fffccb607b8; Gender: C; Age: 2; Genome: {
    __dummy__root: 0; elmreleff: 3790; lkhdmisifelmrtc: 1; elim: 0; move: 3; ...
...
[84.470000] Final most fit command: /home/user/dgga/lingeling_wrapper.sh -f --elmreleff=3790
    --lkhdmisifelmrtc=1 --elim=0 --move=2 --rstinoutinc=763 ... instance_here
```

Chapter 8

Project chronology

In this chapter, we list the chronological evolution of the project. We reported the progress of the project every *Monday, Wednesday and Friday* of each week, to have a detailed view of the project development. The project has been concluded in 87 days (about 3 months), within the period of 4 months estimated for graduate thesis projects at the Escola Politècnica Superior of the University of Lleida.

Week 1

- *Day 1*
  - Identified *GGA* missing dependencies in a Rocks Cluster distribution.
  - Installed *GGA* dependencies in a Rocks Cluster distribution.
  - Successfully executed *GGA* in the cluster front-end.
- *Day 3*
  - Installed *GGA* from scratch in a fresh Linux installation on a VM, without using any package of the distribution.
  - Created mini-manual with all the steps to build and execute GGA and its dependencies.

Week 2

- *Day 6*
  - Studied the option of using MPI [44] and OpenMP [45] to create the distributed version of *GGA*.

  - Identified and issue with the *GGA*'s command line in the configuration file: it does only accept white spaces as the command parameters separator.

- *Day 8*
  - Tested the behaviour of the shared pipes with the cluster shared file system.
  - Created class diagrams of the original *GGA* to find potential issues.

- *Day 10*
  - Identified original *GGA* classes responsibilities.
  - Detected some bugs in the shared memory module.

Week 3

- *Day 13*
  - Performed *GGA* stress tests in the cluster's front-end. Everything OK.
  - Performed *GGA* stress tests in the cluster's medium queue. Everything OK after changing all the paths in the configuration files from relative to absolute.

- *Day 15* Nothing due to exams.

- *Day 17* Nothing due to exams.

Week 4

- *Day 20*
  - Performed some socket connectivity tests within cluster nodes. Detected requirements:
    - Retrieve absolute path to the executable, not viable with argv[0].
    - Retrieve all the IPs of the different interfaces.
    - Create an script to start new jobs using *qsub*.
  - Confronted some problems when integrating the raw C sockets API with the already existing code. Exploring two solutions: creating a wrapper for the C sockets API or using *Boost Asio*.

- *Day 22*
  - Designed a connection protocol for DGGA using sockets.

        * Textual protocol, ASCII only.
        * Best effort policy: request resources in small chunks and use it as soon as possible.
        * The Master must be able to recover from Worker and network failures.
        * Greeting sequence initialised by a Worker when successfully connected to the Master.

- *Day 24*
    - Implemented a first naive version of the communication protocol. It only prints the messages and works with fake results.
    - Implemented dummy target algorithm executions to simulate *GGA* results and test the communication protocol.
    - Working on a job recovery system for the Master.

Week 5

- *Day 27*
    - Added support for IPv4 and IPv6. Using an IPv6 socket with IPV6_ONLY deactivated.
    - Children of the "qsubed" processes remain alive after the parent death.
    - Tested the job recovery system without problems.
    - Communication protocol successfully tested on the cluster's medium queue.
- *Day 29*
    - Tested the *Boost* serialization library.
    - Further investigation of the *Boost* Asio library.
        * Higher level of abstraction.
        * Uniform API.
        * Cross platform.
        * Uses O.S specific APIs (more efficient).
    - Reported and fixed a non critical bug in the original *GGA*.
- *Day 31*
    - Fixed some of the circular dependencies in the original *GGA* code.
    - Moved some parts that used pointers to use constant references whenever it was possible.
    - Incorporated the *Boost* serialization functions to some of the fixed classes.

Week 6

- *Day 34*
  - Moved code from pass by value to pass by reference and constant reference.
  - Moved code that used pointers or references to use copies to remove data dependencies between code that will run in different computers.
  - Fixed a segmentation fault error, triggered by the deletion of GGAParameterTree.
  - Fixed GGAValue retrieve operations.
  - Added '\n' as a valid character to separate parameters on the cmd tag in the XML configuration file.
- *Day 36*
  - Fixed several memory leaks on the XML parsing routine.
  - Minor changes in GGATournament to isolate GGAGenome from GGASelector.

Week 7

- *Day 41*
  - Completely isolated GGASelector and GGARunner from GGATournament.
    - Discovered possible improvements after isolating the code.
  - Started DGGA Master and Worker implementation.
- *Day 43*
  - Implemented first Master selector version.
  - Moved communication from raw sockets to *Boost* Asio.
  - Tested Master behaviour with Netcat.
  - Started Remote selector first version.
- *Day 45*
  - Tested the Master and Worker first versions in localhost.
  - Cleaned up some portions of code.

Week 8

- *Day 48*
  - Fixed "End of message" mark conflict with the serialized data.
  - Moved memory management of GGAParameterTree, GGATreeNode and GGAParameter from manual new and delete to boost::shared_ptr.
  - Identified a bug when compiling *Boost* 1.55 with clang++ in OS X. The threads library is not compiled on that platform.
  - Discussion with Kevin, the author of the original tool, about the actual design, what should be improved and which bugs are necessary to solve.
- *Day 50*
  - Fixed error at GGASharedMemory destructor routine.
  - Moved mkfifo to pipe, avoiding to leave named pipes in the file system.
  - Discovered potential problems when wrapping the binaries to tune. Scripts in bash/sh/ksh/... hold signals while a non built-in command is running.
  - Tested that SGE sends signals to terminate a job to the whole process group.
- *Day 52*
  - Implemented a new routine to notify when a GGARunner has finished its execution, but not tested yet.
  - Added a mechanism to start Workers, based on wrapper files, in other Job Scheduler systems different from SGE.

Week 9

- *Day 55*
  - Moved GGASharedMemory to *Boost* equivalent, which is more integrated with C++ and multi-platform.
  - Added Linux special instruction to kill children processes when the parent dies.
  - Studied the option of changing the actual architecture of double fork() + waitpid() to gather the execution times to thread() + fork() + wait4().
- *Day 57*
  - Timeout check integrated into *GGA*.

- Changed the old architecture of fork() + waitpid() to thread() + fork() + wait4().
- Fixed GGAGenome-GGAInstance performance propagation to next generations. It was propagated without the penalty factor.

- *Day 59*
  - Fixed some memory leaks in the XML parameters parsing routine.
  - Moved C signal() to POSIX sigaction(), which offers more control.
  - Added the necessary code to execute the selection of Genomes asynchronously in the remote selector.

Week 10

- *Day 62*
  - Fixed some warnings when compiling *DGGA* in Rocks cluster with an ancient version of *GCC*.
  - Started *DGGA* stress tests.
- *Day 64*
  - Started the project report.

Week 11

- *Day 69*
  - Writing the project report.
  - Started the tuning of a MaxSAT solver with *DGGA*. The solver is being partially tuned the objective is to find errors in the tool.
- *Day 73*
  - Writing the project report.
  - Recovered the results of tuning the MaxSAT solver, the execution of *DGGA* has finished properly.
  - Relaunched the tuning of the MaxSAT solver with more parameters.

Week 12

- *Day 78*
  - Writing the project report.
  - *DGGA* stress tests simulating a busy cluster.
  - Recovered the results of tuning the MaxSAT solver, something went really wrong.
- *Day 80*
  - Found some problems with the MaxSAT solver wrapper.
  - Writing the project report.
  - More *DGGA* stress tests.

Week 13

- *Day 83*
  - Writing the project report.
  - Started tuning the *lingeling* SAT solver for 30 seconds.
- *Day 85*
  - Writing the project report.
  - Recovered the results of tuning *lingeling* for 30 seconds, without major problems.
- *Day 87*
  - Writing the project report.
  - Started the tuning of *lingeling* for 300 seconds.
- *Day 89* (Extra report)
  - Started the tests with the *lingeling* configurations although some of the 300 seconds configurations have not finished.
- *Day 90* (Extra report)
  - Finished the project report.

Chapter 9

Conclusions and future work

Through the development of this project, we have seen that automatic configuration is crucial to exploit all the capabilities of an algorithm or solver. For example, many of the solvers that are available today have parameters which cannot be set by the end user. These parameters have been fixed by the developers to values that they have found beneficial, without knowing the particular needs of the end user. Automatic configuration allows solvers to adapt to the final environment in which they need to perform. After the installation in the users environment, an algorithm should be automatically configured for the common tasks it is actually used for, and without requiring the user to learn about the algorithm parameters.

Another key aspect of this project is to realise that some applications or tools can greatly benefit from parallelization. The global computing architecture trend is to potentiate parallel or distributed applications. Therefore, we conclude that developers should have always in mind the potential parallelization of their applications and design accordingly to this.

This project has also allowed me to work with different technologies what has broaden my skills as a software engineer. In particular, I improved my skills as a C++ programmer, after learning how the developers of the *Boost* project have solved some common problems. I also learned how to exploit the resources of a high performance computing cluster, and, at lower level, I have experimented with different operating system APIs, to solve specific process management issues.

As future work, we have several working avenues, which is a sign of the potential of this project:

- Improve the instance selection strategy, to improve the results when configuring an algorithm using instances with different characteristics, i.e., ensure that all the *representative* instances have been uniformly tested.

- Make public some internal parameters, allowing the end user (or an automatic configuration tool) improve the results of *DGGA* for specific problems.
- Modify the design of the communication protocol for systems without a shared file system.
- Implement a new performance evaluation system to exploit the resources of heterogeneous systems like cloud computing or grids.
- Extend *DGGA* to allow interaction at running time in order to modify some parameters, e.g.: request more nodes or stop the configuration process at any moment.

Finally, this project has encouraged me to continue working in this field. I found all that I have done within this work somehow interesting, but as any project, this one also had its pros and cons. Writing this report has been tedious and difficult because I am not used to write this kind of documents. However, my final impression is positive because I have been working in what I like and I think that the final result is a useful project.

Bibliography

[1] F. Rossi, P. van Beek, and T. Walsh, eds., *Handbook of Constraint Programming.* Elsevier, 2006.

[2] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[3] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for sat," *J. Artif. Intell. Res. (JAIR)*, vol. 32, pp. 565–606, 2008.

[4] L. Xu, H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *AAAI*, 2010.

[5] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Intell. Res. (JAIR)*, vol. 36, pp. 267–306, 2009.

[6] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, (Berlin, Heidelberg), pp. 142–157, Springer-Verlag, 2009.

[7] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "Isac - instance-specific algorithm configuration," in *ECAI*, pp. 751–756, 2010.

[8] Y. Hamadi and M. Schoenauer, "Guest editorial: special issue - revised selected papers of the lion 6 conference," *Ann. Math. Artif. Intell.*, vol. 69, no. 2, pp. 149–150, 2013.

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[10] R. Martins, V. M. Manquinho, and I. Lynce, "Parallel search for maximum satisfiability," *AI Commun.*, vol. 25, no. 2, pp. 75–95, 2012.

[11] S. Minton, "Automatically configuring constraint satisfaction programs: A case study," *Constraints*, vol. 1, no. 1-2, pp. 7–43, 1996.

[12] A. S. Fukunaga, "Automated discovery of local search heuristics for satisfiability testing," *Evol. Comput.*, vol. 16, pp. 31–61, Mar. 2008.

[13] M. Preuss and T. Bartz-Beielstein, "Sequential parameter optimization applied to self-adaptation for binary-coded evolutionary algorithms," in *Parameter Setting in Evolutionary Algorithms* (F. G. Lobo, C. F. Lima, and Z. Michalewicz, eds.), vol. 54 of *Studies in Computational Intelligence*, pp. 91–119, Springer, 2007.

[14] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil, "Using experimental design to find effective parameter settings for heuristics," *Journal of Heuristics*, vol. 7, pp. 77–97, Jan. 2001.

[15] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, (San Francisco, CA, USA), pp. 11–18, Morgan Kaufmann Publishers Inc., 2002.

[16] C. P. Gomes and B. Selman, "Algorithm portfolio design: Theory vs. practice," *CoRR*, vol. abs/1302.1541, 2013.

[17] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275, no. 5296, pp. 51–54, 1997.

[18] M. Oltean, "Evolving evolutionary algorithms using linear genetic programming," *Evol. Comput.*, vol. 13, pp. 387–410, Sept. 2005.

[19] B. Adenso-Diaz and M. Laguna, "Fine-tuning of algorithms using fractional experimental designs and local search," *Oper. Res.*, vol. 54, pp. 99–114, Jan. 2006.

[20] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pp. 1152–1157, AAAI Press, 2007.

[21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[22] R. Marinescu and R. Dechter, "And/or branch-and-bound search for combinatorial optimization in graphical models," *Artificial Intelligence*, vol. 173, no. 16–17, pp. 1457 – 1491, 2009.

[23] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[24] "Gnome foundation." http://www.gnome.org, 1997-2014.

[25] "Boost c++ libraries." http://www.boost.org, 2001-2014.

[26] B. Stroustrup, *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.

[27] B. Stroustrup, "Adding classes to the c language: An exercise in language evolution," *Softw., Pract. Exper.*, vol. 13, no. 2, pp. 139–161, 1983.

[28] L. Torvalds, "Git." http://git-scm.com, 2014.

[29] Free Software Foundation, Inc, "GNU Compiler Colection." https://gcc.gnu.org, 1987-2014.

[30] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[31] Apple, Inc, "Mac OS X operating system." http://www.apple.com/osx/, 2001-2014.

[32] Free Software Foundation, Inc, "GNU Emacs." http://www.gnu.org/software/emacs/, 2013.

[33] Free Software Foundation, Inc, "GNU Make." www.gnu.org/software/make, 2014.

[34] Eclipse Foundation, "Eclipse." https://www.eclipse.org, 2014.

[35] Free Software Foundation, Inc, "The gnu project debugger." http://www.gnu.org/software/gdb/, 2014.

[36] J. Seward, "Valgrind." http://valgrind.org/, 2013.

[37] A. Biere, "Lingeling sat solver." http://fmv.jku.at/lingeling/, 2010-2014.

[38] F. Hutter, M. López-Ibáñez, C. Fawcett, M. T. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "AClib: a benchmark library for algorithm configuration," in *Learning and Intelligent Optimization, 8th International Conference, LION 8* (P. M. Pardalos, M. G. C. Resende, C. Vogiatzis, and J. L. Walteros, eds.), vol. 8426 of *Lecture Notes in Computer Science*, pp. 36–40, Springer, 2014.

[39] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Satenstein: Automatically building local search sat solvers from components.," in *IJCAI* (C. Boutilier, ed.), pp. 517–524, 2009.

[40] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "clasp: A conflict-driven answer set solver," in *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)* (C. Baral, G. Brewka, and J. Schlipf, eds.), vol. 4483 of *Lecture Notes in Artificial Intelligence*, pp. 260–265, Springer-Verlag, 2007.

[41] Dice Hodings, Inc, “Download, develop and publish free open source software.” http://www.sourceforge.net, 1999-2014. Accessed 13-August-2014.

[42] B. Fox, “Bash (unix shell).” http://www.gnu.org/software/bash/, 1989-2014.

[43] G. van Rossum, “Python programming language.” https://www.python.org/, 1991-2014.

[44] M. P. I. Forum, “MPI: A Message-Passing Interface Standard Version 3.0.” http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[45] OpenMP Architecture Review Board, “OpenMP application program interface version 4.0.” http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, Jul 2013.