Distributed system for Evolutionary Robotics Experiments

Pavel Petrovic, petrovic@idi.ntnu.no

Department of Computer and Information Science
Norwegian University of Science and Technology
Technical report 05/2004.
ISSN 1503-416X.

Abstract

Experimental research in the field of Evolutionary Robotics requires powerful CPU resources. They can be available in an unexpected and untraditional form – student computer laboratories. We have utilized the laboratories in order to perform the experimental runs. For this purpose, we have designed and implemented a distributed evolutionary system based on a unique combination of Linux live CD, our robotics simulator, GaLib, Mysql, and bash scripts. We describe the current distributed evolutionary systems, and the conceptual and implementation issues of our distributed system, as well as show how we run the system to obtain results for our evolutionary experiments.

Introduction

Thanks to the steady progress of wide spectrum of research disciplines and interdisciplinary efforts, many difficult research or applied problems can be solved using efficient methods of today. Still, for many other interesting challenges, we do not know an efficient method or an algorithm. And even if we do, such efficient methods sometimes imply constraints, which we might not be able or willing to comply with. However, we still demand a solution to these challenges, and it often appears to be the case that finding a solution (sometimes only approximating it) can be achieved only by employing extensive computational resources. An application that runs on a set of computers is a *distributed system*, and all such applications form a class of *distributed computing systems*. Many types of distributed computing systems of various scales exist. They are running on various operating systems and having different purpose. The next section gives a short overview of the relevant distributed systems.

The purpose of the simple distributed system developed under this work was a particular experiment in the field of Evolutionary Robotics. Evolutionary Algorithms belong to the class of problem solving (or search) methods that generate approximate (and possibly exact) solutions, and have extensive computational needs. Due to their parallel nature, they are very amenable to distributed computing. Later section reviews the related work on distributed evolutionary systems.

Many distributed systems aim at providing a general computational platform. As a consequence, they typically limit the user to the standard computational environment, often with restrictions or limitations. Our experimental setup required a Linux operating system and the application ought to be run in the super-user mode. A separate section describes the solution we used to employ idle computers in the computational student laboratories during summer holiday season.

The distributed solution we implemented and used to acquire experimental results consisted of two major technologies, the UNIX shell scripts with secure copying for submitting, monitoring, and managing tasks, and SQL database for evolutionary distributed application. Individual sections describe the SQL database used to distribute the data to the computational tasks, and the UNIX shell script system used to submit and monitor the tasks.

Final sections show example runs, which brought interesting results, and discuss further work and possible improvements for future research efforts.

Detailed database structure and the bash scripts are provided in the appendices.

Distributed computing systems and paradigms

Distributed systems can be classified from several different viewpoints.

One class of distributed systems comprises *large scale* distributed computing based on thousands to millions of voluntary CPU-time donators available *throughout the Internet*. The users download specialized clients for their particular software and hardware platforms, and let their computers work during the unused CPU time on a particular distributed computing project, thus making their CPUs consume maximum power possible. However, these systems are typically closed for the user, who usually has little control over the code or the data being processed on his or her machine. On one hand, this is required to guarantee validity of the submitted results; on the other hand, it is a potential threat to the user's security and trust. Such systems typically try to solve or prove some hard mathematics, bioinformatics, cryptographic or other search challenges. Examples include Folding@home, Find-a-Drug, or D^{2}OL, helping to find oral drugs which could fight Anthrax, Smallpox, Ebola, SARS, deadly diseases for which there is currently no cure, and Malaria, a life-threatening disease for which 40% of the world's population is at risk. An example of a multi-purpose platform of this kind is BOINC (Berkley Open Infrastructure for Network Computing) [2]. An overview of the ongoing, past, and planned distributed computing projects is available for example at [10].

Another class of distributed computing systems is utilizing specialized clusters. These are typically Linux-based network-installed and booted rack-mounted powerful PCs or other workstations running software that supports distributed computing either in form of message passing, threading, sockets-based, or batch submission system. Clusters of powerful computational nodes are available to users for submitting distributed applications. An example of such cluster is the Clustis at IDI, NTNU [4].

Distributed applications can be run on large parallel-architecture computers containing tens to thousands of CPUs. An example of such is the NTNU High-Performance Computing project [1].

Yet another class of distributed computing systems is utilizing idle CPU time of a particular institution for general purpose CPU-intensive computational tasks of its own authorized users. Universities, banks, and many other institutions are equipped with enormous unused CPU time, which they could benefit from, if they run such a system. A popular example of such a system is Condor [11], put in force for example at the University of Oslo, and an open-source Q^{2}ADPZ system developed at IDI/NTNU. Such systems typically run a background daemon, which activates the task only when the local user logs out of the workstation. The authorized user submitting the task can monitor and control the progress, while the tasks can optionally be moved to different nodes when the workstation becomes occupied again [5].

A distinguished class of distributed computing is *grid computing*. Grids aim to give answer to all possible needs. Their approach is to provide a general-purpose environment combining all possible platforms and uses. This term is sometimes used as synonym to distributed computing, when it refers to a particular site that integrates distributed computing resources of different types under single concept. An example of such a grid is UK's National Grid Service.

Evolutionary computation and distributed computing

Evolutionary algorithms (EA) are highly parallel stochastic search methods for finding approximate solutions useful when no deterministic algorithm generating good solutions is known. They are inspired by the Darwinian natural evolution principles, and work with a population (a set) of solutions that survive, mate and get mutated from generation to generation based on their performance (fitness). In each generation, all individuals in the population have to be evaluated independently. That is where it is natural to parallelize the execution of the evolutionary algorithms. Some flavors of EA work on multiple populations that evolve independently (island models) – and for them another natural place for parallelizing is allocating one (or several CPUs) for each sub-population. Alternately, evaluating a single individual can also be performed in parallel on several CPUs, if the objective function is suitable for parallelizing. Some existing EA packages support parallelization on various levels. One example is the ECJ package [9] running on the Java platform thus offering high portability across platforms. Among other features, it supports platform-independent checkpointing (which stands for automatic saving of the state of the computation for the purpose of restarting the computation from a given checkpoint) and logging, multithreading, multiple subpopulations and species, inter-subpopulation breeding, inter-process or inter-machine transfer of individuals (even across different platforms), asynchronous island models. Another popular package is the EO (evolutionary objects) implemented in C++ with templates [7]. It provides its own extension ParadisEO for parallelization supporting the cellular model, parallel evaluation functions, parallel evaluation, and island model. Another distributed evolutionary system DREAM [3] is based on JEO (Java brother of EO). Its distributed computation core, DRM is independent of the EA field and can run any distributed application. Instead of master-slave or client-server architecture, they base the distributed engine on an epidemic protocol, where each node keeps a database of some of the peers in the computational network. The user of DREAM can work on several different levels depending whether the standard set of features satisfies his needs, or whether a more low-level application interface is required. For instance the user can specify its evolutionary application using simple EASEA high-level description language that is compatible also with GaLib or EO.

When setting up a distributed EA, one typically uses a combination of a package for distributed computation and a package for EA. The distributed computation package will be responsible for delivering the inputs/outputs to/from the computational nodes, and submitting the tasks to the nodes automatically. The user has to configure which code and data have to be processed. Usually the user specifies at how many nodes he runs a particular application, or optionally what would be the topology of the parallel virtual computer. The user can implement the communication between the computational nodes either through the shared file system, message-passing, or sockets, or simply rely on the parallelization features provided by the chosen EA package. In our case, the evolutionary robotics experiment was based on the GaLib package, which does not support parallelization, and thus we needed a distributed computing package. We chose to implement our own for the reasons of the simplicity, modifiability, control, low maintenance and installation costs, and because of the other requirements that are described in the next section.

Experimental setup requirements

Our experiment is an EA attempting to evolve an arbitration mechanism for robot controller represented as a set of augmented finite state automatons. The fitness of an individual in the evolutionary population is the quality of the controller with respect to the task it is to perform. To evaluate an individual and obtain its fitness, the robot with that controller has to be started multiple times from several different locations. Even though the program is made for real robots based on the LEGO RCX platform, it would take infeasible long time and amount of work to test the controllers on the real robots, therefore they are started in a specialized simulator. And even though simulating the LEGO computing environment on the PC Workstation is relatively straight-forward task in the realistic time, i.e. real-to-simulated time ratio is 1-to-1. However, that would still be time-infeasible as the experiment requires hundreds of generations (300-600) with hundreds of individuals (100-200), started from many different locations (up to 12), and each lasts several minutes (1-4), 600*200*12*4 = 5760000 minutes = 4000 days of CPU time. We must therefore run the simulation in faster simulated time (200-300 –times faster than reality), which is possible thanks to a higher CPU frequency of the PC, but gets more intricate as the Linux kernel can schedule out any thread of the controller for a relatively long time and perform some system maintenance tasks unlike the computing environment on the RCX. To minimize these troubles, we ought to run the experiment in the real-time scheduling mode which requires super-user privileges (`sched_setscheduler(SCHED_RR)` system call). This however rules out using a public computing cluster environment (in our case clustis), or a background computational daemon running in a limited user space on the public student lab computers in their idle time (in our case Q^2ADPZ). The requirement for running our experiment in a distributed way implies a set of networking Linux workstations where the task can be run with super-user privileges. Our EA extension for distributed computing also requires a running Mysql server. In addition, we require a recent Linux kernel containing the new threading library (NPTL, [6]), which introduced the lightweight threading into Linux and improved the performance of thread context switching *about one hundred times*.

Solution using Linux live CD

Thanks to the support of the technical groups of the Department of Computer and Information Science, and Department of Electronics and Telecommunications at NTNU, we were able to make use of the student computer laboratories during the summer vacation period for running the experiments. To satisfy the requirements specified above, we found the easiest solution to prepare a specialized ISO image and produce tens of Linux live CDs – bootable Linux CDs containing the whole operating system. The system was based on the standard Knoppix 3.4 distribution [8], with several modifications in the booting start-up sequence, because the default auto configuration scripts made the system fail on our hardware. The initialization code included on the CD created all temporary files and large-enough virtual drive in the ramdisk. Then the startup script downloaded and started the distributed computing scripts and executables from our server during CD startup. In the case of computational nodes with the lack of memory, swap partitions were created on the local hard disks – in place of an unused FreeBSD partition. In this way, the code submitted to the node could not access the local file system, and the console was disabled so that the random local users could not interfere with the computation. However, they could reboot the computational nodes at any time and work in the Windows operating system to utilize the workstation for regular use. The solution with live Linux CD allows starting an arbitrary distributed application provided from the configured server (in fact it attempts to locate the startup scripts at three different locations: a particular server, and two backup URL addresses in case the server is not reachable). Earlier implementation relied on *nfs*, however it turned out to be too unstable to keep the connection active for several days for tens of nodes. The scripts controlling the distributed computational environment are described in the following section.

Batch system for submitting and monitoring the computational tasks

The batch system is based on a *scp* (secure copy) UNIX utility that securely retrieves files from and sends files back to the master. The live CD contains a private *dsa* key using which it connects to a specialized user on the master. The master communicates with the computational nodes through this user home directory. Even though gaining access to this low-privileged user (for example by taking a copy and tweaking the live CD) could harm the computational progress, we found this solution secure enough, given the workstations were booted to a console-locked mode. The Mysql database was configured so that only the authorized nodes could manipulate the database, a similar precaution could be implemented on the distributed master node. The script retrieved from the master (*cargo-main-script*) executes the following steps:

- determines the local IP-address (to be combined with the *pid* for unique computational node program id),
- determines if the UNIX swap partition is being created,
- prepares the system,
- starts the main loop script (*guard*).

The guard script is then periodically checking at the master:

- if there is a new task to be started,
- if the task is to be stopped (or sent some other signal),
- if there is a request to execute some service command, and return its output,
- it reboots the node on request, and
- it periodically informs the master that it is ready to accept tasks,
- it is also checking for the network connection: if it goes down, the script tries to reconnect, so that the node can continue receiving and sending data to the database, and communicate with the master.

Computational tasks submitted to nodes are started there using a separate script (*start_compute_task.launch*), which moves the retrieved task to its starting location, executes it while redirecting its output, and notifies the master when it terminates and after it has copied the output from the program back to the master. In our experiment, the computational tasks were themselves scripts that downloaded (again using *scp*) all the required data and binary files, and monitored the task running (for example *task-rep/seq82_ea_master*). When the program running at the node stopped before the computation was completed, it was automatically restarted.

Set of scripts at the master allowed the user to manage the distributed environment. The *submit* script places the computational task to a directory from where the *guard* fetches it, and starts it. The *remote* script submits and executes any shell command at a given node, and displays its output (even while some program is running there). In this way, additional data can be sent to the nodes, or partial results retrieved, and viewed, even though the computation is still in progress. The *slstatus* script prints a summary of all computational nodes, and displays the task that is being computed, if any, see Figure 1 for an example. The output from each node is saved into the *runs* subdirectory for later use and analysis. The user can collect list of computational nodes into files named *pool** and the *start* and *stop* scripts allow to submit the same script to all nodes specified. These files can be easily constructed by taking all running nodes (*ls really-on/** > *poolX*), and dividing into multiple parts, if several distributed experiments are running in parallel. In this way, the user has full control over the computational status and can easily take some of the nodes in or out of the pool of nodes, even though some more expert knowledge might be required (if the system shall be used in larger scale, this part could be made more advanced or user-friendly).

```
cargo@search:~$ ./slstatus
87 nodes are on...
129.241.103.142: 8831 ea project/cargo/cfg/sequential82_e.prj
129.241.105.114: 8583 ea project/cargo/cfg/sequential82_e.prj
…
129.241.105.59: 23253 ea project/cargo/cfg/sequential83_e.prj
129.241.103.147: 12596 ea project/cargo/cfg/sequential83_e.prj
129.241.105.68: MASTER: ea project/cargo/cfg/sequential83_e.prj
129.241.105.69: 24241 ea project/cargo/cfg/sequential83_e.prj
…
129.241.103.78: 5688 ea project/cargo/cfg/sequential82_e.prj
129.241.105.105: 5848 ea project/cargo/cfg/sequential82_e.prj
129.241.105.65: MASTER: ea project/cargo/cfg/sequential82_e.prj
129.241.103.195: 30266 ea project/cargo/cfg/sequential83_e.prj
…
129.241.105.189: 29849 ea project/cargo/cfg/sequential83_e.prj
129.241.105.98: 32345 ea project/cargo/cfg/sequential83_e.prj
129.241.105.70: 17363 ea project/cargo/cfg/sequential83_e.prj
no more nodes.
```

Figure 1. Example of the computation progress as seen by user from the master.

In many situations, it is necessary to submit the same task again and again, to collect enough statistical evidence about the performance of a computational experiment. For this purpose the *resubmitter2* script monitors the progress of the computation, and automatically restarts the job (that is one master node and many slave nodes as soon as they have finished). The *kill* script allows sending signal to the running program on a specified node, and the *killall* scripts sends this signal to all nodes in the pool. The current status of the nodes is available in the *really-on* directory. Each time a node is restarted, a note is made in the *startup* directory, and the log of the submissions is stored in the *submit.log* file. See the Appendix A for the details of the batch scripts.

Distributed populations in evolutionary algorithm

Our distributed settings (student lab) implied that any computational node could have been unexpectedly restarted at any time. The computational nodes were distributed along the whole campus, but all were connected with a high throughput at least 100MBit switched Ethernet. The experiment typically included 40 to 120 computational nodes. Individuals that were already evaluated once were not evaluated again in order to speed up the evolution process: all the computed fitness values were stored in a cache (database table). Each generation then contained about 25 – 200 individuals to be evaluated. The evolutionary algorithm that performed the evolutionary operators (crossover, mutation, selection) and maintained the population was running on one of the nodes that took a role of the master. All the other computational nodes ran the same program, but did not maintain the population. Instead, they published their unique identifier to the list of nodes table. The master saved all individuals that needed to be evaluated into a database table in the beginning of each generation. It also retrieved the current list of nodes that were ready to compute, and assigned each individual to a particular node. A node retrieved all of the assigned individuals and evaluated them the required number of times from all required starting locations, and computed the resulting fitness value. This value has been stored into the database, and the individual was marked as evaluated. When all individuals were evaluated, they were moved with a single SQL statement from the *TODO* table into the *CACHE* table. As soon as there were no individuals remaining, the master retrieved all fitness values from the *CACHE* table and proceeded to the next generation. Figure 2 depicts the whole distributed computation setup.

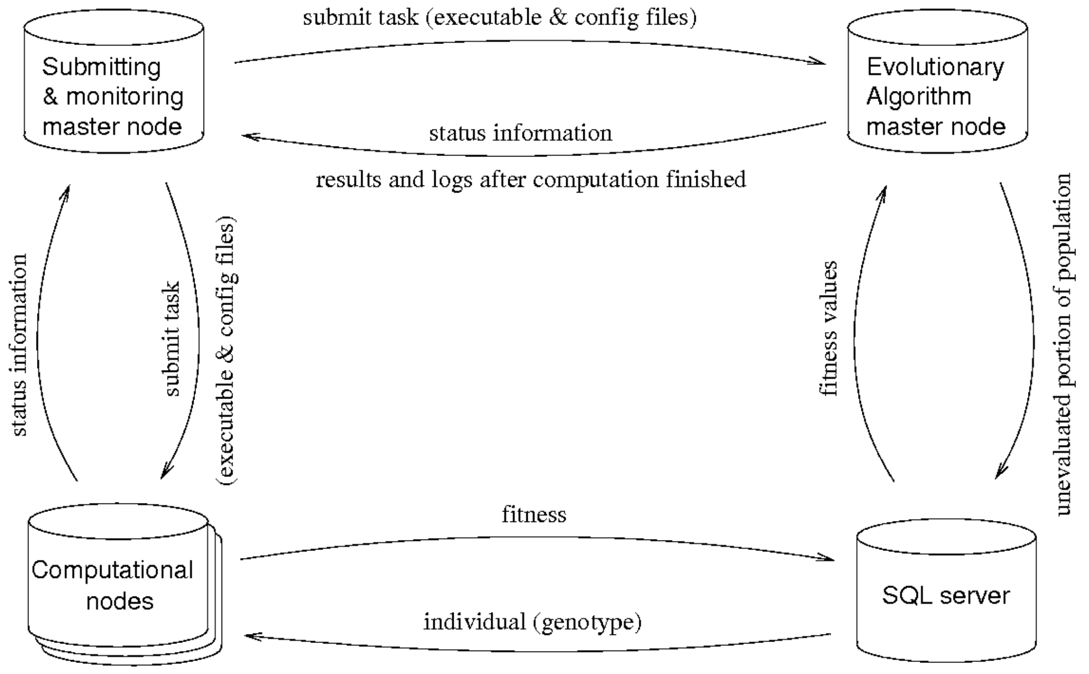
Submitting & monitoring master node
submit task (executable & config files)
status information
results and logs after computation finished
Evolutionary Algorithm master node
status information
submit task
(executable & config files)
fitness values
unevaluated portion of population
Computational nodes
fitness
individual (genotype)
SQL server

Figure 2. Overall architecture of the distributed evolutionary system.

Different events could influence the computational progress: a computational node could have been stopped (booted to Windows), or the program could have experienced a program crash (this was unavoidable during the development phase). In such situation, the master measured the average time of a result delivery. If some node was computing 5-times longer than was the average for all other nodes that already had completed the same generation (thus having a comparable complexity), the individual was re-assigned to another node that already completed (if available), or was still computing. When a program crashed, it was usually restarted automatically. When a newly restarted program detected a unique identifier of a crashed slave running on the same machine, it marked this node as crashed so that the master could reassign its individuals to another node immediately. (The newly started program could not simply adopt the individuals of the crashed one, because it needed to receive additional information from the master, which was distributed only on the start of each generation: it needed to proceed to the correct evolutionary step). Appendix B contains a commented structure of the database tables used to deliver the individuals and results, as well as SQL commands applied.

Later, we have modified the distributed scenario in order to test if a new algorithm would have a higher throughput. Instead of assigning the individuals to particular computational nodes, the master only publishes all unevaluated individuals through the database. The computational nodes individually take the individuals to be evaluated and save the fitness back to the database on the one-by-one basis. There is a little bit of communication overhead, and a little time wasted before each individual is evaluated as the slaves must make short breaks while querying the database perpetually and waiting for new individuals. On the other hand, and more importantly, slaves that proceed faster, and finish their package of individuals early, are not utilized again in the earlier version of the algorithm until all slaves deliver their results. However, there still may be some individuals for which the evaluation has not been started yet. Difference lies also in the handling of the faulty slave nodes – these could be simply ignored in the new version of algorithm without the resource-demanding detection process, because the initiative was coming from the nodes, not from the master. If the result has not been delivered back within a fixed time constant, the individual that the faulty node checked-out became considered unallocated after that time, and another computational node allocated it, that is, started its evaluation. We compared the utilization of CPU nodes of the two algorithms, as shown in the figure 3.

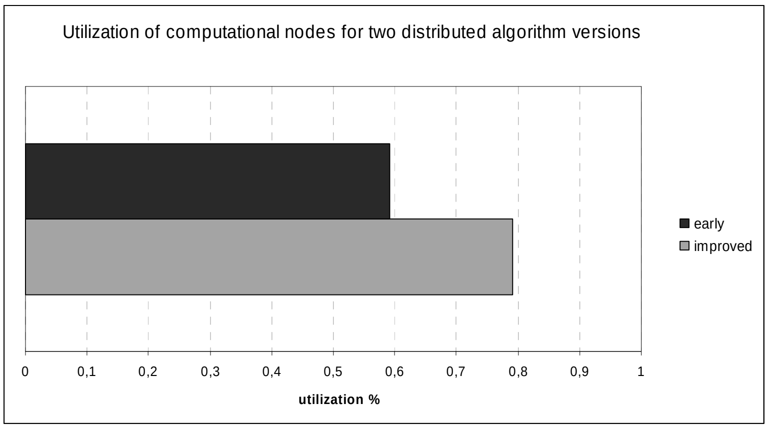

Figure 3. Utilization of two versions of distributed algorithm, evaluated on 53 computational nodes on real experimental run (average of total utilization throughout all incremental steps of sequential experiment). Standard deviation for early algorithm was 0.16%, whereas only 0.05% for the improved algorithm. The utilization of the nodes is influenced by several factors: in the early algorithm, the master assigns more individuals to the same faster computational nodes (when the number of nodes does not divide the population evenly), however the speed difference does not necessarily have to be so high, and slower nodes need to wait for the faster nodes to complete the extra individual; another extreme is when the population is evenly divided and the faster nodes need to wait for the slower nodes to complete; finally, in the second algorithm, the faster nodes are doing more work, but not on the centralized request of master, but naturally thanks to their higher speed; they are therefore not necessarily more utilized, as they take on more individuals only when there is still time for it.

Conclusions

We have designed, implemented and evaluated a distributed system for performing highly-specialized research experiments using very low extra cost solution utilizing computers in the student computer laboratories. The communication over TCP/IP network is based on a set of bash scripts which rely on the *scp* (secure copy, OpenSSL) protocol, and Mysql database server and client libraries. We use a modified live Linux CD distribution to easily boot the computational nodes into ready state based on demand and availability. A comparison of two applied distributed algorithms shows that a decentralized version had higher utilization and thus also delivered results in shorter time.

References

[1] Aerts P, Lüthi H P, Ynnerman A (2004) *Evaluation of NOTUR. NOTUR – A Norwegian High Performance Computational Infrastructure*. Norges Forskningsråd. ISBN 82-12-01991-8.

[2] Anderson P (2004). *BOINC: A System for Public-Resource Computing and Storage.* In 5th IEEE/ACM International Workshop on Grid Computing, November 8, 2004, Pittsburgh, USA.

[3] Arenas M G, Collet P, Eiben A E, Jelasity M, Merelo J J, Peachter B, Preuß M, Schoenauer M (2002) *A Framework for Distributed Evolutionary Algorithms*. In PPSN VII, LNCS 2439, pp. 665-675, Springer-Verlag.

[4] Cassens J, Constantinescu Z (2003) *It's Magic: SourceMage GNU/Linux as a High Performance Cluster OS*. (abstract). LinuxTag 2003 Conference, July 10-13, Karlsruhe.

[5] Constantinescu Z, Petrovic P (2002) Q2ADPZ, *An Open Source, Multi-platform System for Distributed Computing,* ACM Crossroads Student Magazine, 9, 2.

[6] Drepper U, Molnar I (2005) *The Native POSIX Thread Library for Linux*, RedHat, http://people.redhat.com/drepper/nptl-design.pdf.

[7] Keijzer M, Merelo J J, Romero G, and Schoenauer M (2001) *Evolving objects: a general purpose evolutionary computation library*. In P. Collet et al., editor, Proceedings of Evolution Artificielle'01, LNCS. Springer Verlag.

[8] Knopper K (2000) *Building a self-contained auto-configuring Linux system*. In Proceedings of the 4th Annual Linux Showcase & Conference, October 10-14 2000 Atlanta.

[9] Luke S et al (2005) *A Java-based Evolutionary Computation and Genetic Programming Research System*, George Mason University's ECLab Evolutionary Computation Laboratory, http://cs.gmu.edu/~eclab/projects/ecj/.

Merelo J J, Arenas M G, Carpio J, Castillo P, Rivas V M, Romero G, Schoenauer M (2000) *Evolving objects*. In M. Grana, editor, FEA (Frontiers of Evolutionary Algorithms) proceedings.

[10] Pearson K, editor (2005) *Distributed computing info*, http://distributedcomputing.info/**.**

[11] Thain D, Tannenbaum T, Livny M (2005) Distributed *computing in practice: the Condor experience*. Concurrency and Computation: Practice and Experience, Volume 17, Issue 2-4, pp. 323-356. John Wiley & Sons.

Appendix A – bash scripts

A1. cargo-main-script

```
#!/bin/bash

if [[ -e /cargo/boot_welcome_msg.txt ]]; then
  umount /cargo;
fi

ifconfig | sed -e "s/^.*inet addr:\([0-9]*\.[0-9]*\.[0-9]*\.[0-
9]*\).*$/\1/g" -e "s/^[^1].*$//g" |grep "[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*" |
grep -v "127\." > /etc/ip

rm /etc/inittab
scp cargo@search:inittab /etc
chown root:root /etc/inittab
echo cargo-main-script: hello `cat /etc/ip`
echo cargo-main-script: determining if swap partition is going to be
created on FreeBSD partition...

MEMORY=`free | head -2 | tail -1 | head -c 18 | tail -c 14`

echo cargo-main-script: free memory: $MEMORY

if [[ $MEMORY -gt 280000 ]]; then
  echo "cargo-main-script: there is more than 256MB free memory, not
creating swap.";
else
  echo "cargo-main-script: less than 256MB memory, looking for FreeBSD
partition..."
  SWAPPART=`fdisk -l /dev/hda | grep "FreeBSD" | head -c 9`
  if [[ -z $SWAPPART ]]; then
    echo "cargo-main-script: no FreeBSD partition at `cat /etc/ip` found"
  else
    echo "cargo-main-script: creating swap on FreeBSD partition..."
    mkswap $SWAPPART
    swapon $SWAPPART;
  fi;
fi

echo "cargo-main-script: downloading the scripts..."

scp cargo@search:checkenter /home
scp cargo@search:guard /home
scp cargo@search:start_compute_task.launch /home
scp cargo@search:guarding.what /home
scp cargo@search:cargo-init /home
scp cargo@search:boot_welcome_msg.txt /home
chmod a+x /home/guard /home/start_compute_task.launch /home/cargo-init

mkdir /home/task
mkdir /home/busy
mkdir /home/reboot
mkdir /home/out
mkdir /home/kill
mkdir /home/cmd

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/X11R6/lib

/home/cargo-init

if /home/checkenter "To stop the automatic startup from cargo server and
work in standard Knoppix Linux, PRESS ENTER NOW..."; then
  echo "Continuing startup of cargo...";
else
  echo "OK, automatic startup interrupted."
  exit;
fi;

cd /home
STARTUPFILE=/home/`cat /etc/ip`.`date +%F.%H-%M-%S.%N`
date > $STARTUPFILE
scp $STARTUPFILE cargo@search:startup
rm $STARTUPFILE
printf "cd reboot\nrm all\nquit\n" >/home/rmrebootall
sftp -b /home/rmrebootall cargo@search
rm /home/rmrebootall

# a bit of cleaning up...
printf "sleep 30\n/etc/init.d/xsession stop\nsleep 5\nrm -rf
/home/knoppix\n" >/etc/stopx
printf "telinit q\n" >>/etc/stopx
#printf "sleep 30\nrm /dev/tty[1-4]\n" >>/etc/stopx
#printf "ps -Fu root|grep login|sed \"s/^root *\\([0-9]*\\).*/kill -9 \\"
>>/etc/stopx
#printf "1/g\" >/etc/stopconsoles\nchmod a+x
/etc/stopconsoles\n/etc/stopconsoles\n" >>/etc/stopx
chmod a+x /etc/stopx
if /home/checkenter "X?"; then
  echo "Terminating X in a moment."
  /etc/stopx &
else
  echo "X will not be terminated.";
fi

cat /home/boot_welcome_msg.txt

# the main task-fetching loop
/home/guard `cat /etc/ip` `cat guarding.what` &
```

A2. guard

```
#!/bin/bash

cd /home
COUNTER=1
COUNTER2=1

while [[ -e /home/guard ]]; do
  sleep 10

  scp cargo@search:task/$1 /home/task >&/dev/null

  if [[ -e /home/task/$1 ]]; then
    echo "guard: starting new task at `date`"
    /home/start_compute_task.launch /home/task/$1 &
    printf "rm task/$1\nquit\n" > tmp.rmtask
    sftp -b tmp.rmtask cargo@search >&/dev/null
    rm tmp.rmtask;
  fi

  scp cargo@search:reboot/* /home/reboot >&/dev/null

  if [[ -e /home/reboot/all ]]; then
    echo "guard: received reboot all request, rebooting..."
    if ps -u root | grep $2; then  # see if the program is still running...
      # yes,yes -> send signal
      kill -9 `ps -u root | grep $2 | head -c 5` >& /home/out/$1.killed;
    fi
    sync
    reboot
    exit;
  fi

  if [[ -e /home/reboot/$1 ]]; then
    echo "guard: received reboot request, rebooting..."
    if ps -u root | grep $2; then  # see if the program is still running...
      # yes,yes -> send signal
      kill -9 `ps -u root | grep $2 | head -c 5` >& /home/out/$1.killed;
    fi
    printf "rm reboot/$1\nquit\n" > tmp.rmreboot
    sftp -b tmp.rmreboot cargo@search >&/dev/null
    sync
    reboot
    exit;
  fi

  let COUNTER=$COUNTER+1
  if [[ $COUNTER -eq 24 ]]; then
    COUNTER=1
    echo `date` > $1
    scp $1 cargo@search:on >&/dev/null;
  fi
  let COUNTER2=$COUNTER2+1
  if [[ $COUNTER2 -eq 120 ]]; then
    COUNTER2=1
    if [[ -z `grep eth0 /etc/network/interfaces` ]]; then
      printf "\niface eth0 inet dhcp\n" >> /etc/network/interfaces
      ifdown eth0
      ifup eth0;
    fi;
  fi

  scp cargo@search:kill/$1 /home/kill >&/dev/null

  if [[ -e kill/$1 ]]; then   # see if we got to send signal to program...
    echo "guard: signal request received, looking for $2..."
    if ps -u root | grep $2; then  # see if the program is still running...
      echo "guard: seeing $2, sending signal..."
      kill `cat kill/$1` `ps -u root | grep $2 | head -c 5`
      rm kill/$1
      printf "rm kill/$1\nquit\n" >tmp.rmkill
      sftp -b tmp.rmkill cargo@search >&/dev/null
      rm tmp.rmkill;
    else
      echo "Process $2 not found :("
      rm kill/$1
    fi;
  fi

  scp cargo@search:cmd/$1 /home/cmd >&/dev/null

  if [[ -e cmd/$1 ]]; then          # see if we got to copy a file
    cmd/$1 >& out/$1.remote
    rm cmd/$1;
    printf "rm cmd/$1\nput out/$1.remote out/$1.remote\nquit\n" >tmp.rmcmd
    sftp -b tmp.rmcmd cargo@search >&/dev/null
    rm tmp.rmcmd;
  fi;

done

echo "guard: script deleted, terminating."
```

A3. submit

```
#!/bin/bash

if [[ -z $1 ]]; then
  echo "submit IP task_file [VAR VALUE]"
  exit;
fi

if [[ ! -e task-rep/$2 ]]; then
  echo "ERROR: task file $2 not found in task-rep"
  exit;
fi

if [[ ! -z $3 ]]; then
  echo "replacing $3 for $4 in the task file..."
  sed "s/$3/$4/g" <task-rep/$2 >task/$1;
  chmod a+rx task/$1
else
  echo "not replacing any variables in the task file."
  cp task-rep/$2 task/$1;
fi

echo "Submitting task $2 to $1..."

CNTR=1
while [[ -e task/$1 ]]; do
  let CNTR=$CNTR+1
  if [[ $CNTR -eq 600 ]]; then
    echo "waiting too long, giving up for $1."
    exit;
  fi
  printf "."
  sleep 1;
done

echo "Task successfully submitted."
echo submit $1 $2 $3 @ `date`>> submit.log
```

A4. start, stop

```
#!/bin/bash

# usage: start master_ip task_id X
#  node IPs should be listed in file poolX

if [[ $1 -eq "x" ]] ; then
  echo "skipping master, assuming it is running already" ;
else
 ./submit $1 $2_master ;
fi

cat pool$3 | grep -v "#" | sed "s/^/.\/submit /g" | sed "s/$/ $2_slave
\&/g" > start$3.tmp
chmod a+x start$3.tmp
./start$3.tmp
```

```
#!/bin/bash

# usage: stop X
#  node IPs should be listed in file poolX

cat pool$1 | grep -v "#" | sed "s/^/.\/remote /g" | sed "s/$/ \"cp
\/home\/boot* \/home\/quit; kill -9 \\\\\`ps -u root|grep 129.241|head -c
5\\\\\`;killall ea;rm \/home\/current3\/log\/*\" \&/g" >stop$1.tmp

chmod a+x stop$1.tmp
./stop$1.tmp
```

A5. resubmitter2

```
#!/bin/bash

cd /cargo

if [[ -z $1 ]]; then
  printf "resubmitter2 pool# master_ip task_id\n"
  exit;
fi

if [[ -z $4 ]]; then
  printf "Resubmitter2 $1 $2 $3 `date`:\n\n starting...\n" >>
resubmitter2.log
  ./start $2 $3 $1
  sleep 600
fi

EASTATUS=`./remote $2 "ps -u root" |grep ea`
if [[ -z $EASTATUS ]]; then
  # try again...
  sleep 30
  EASTATUS=`./remote $2 "ps -u root" |grep ea`
  if [[ -z $EASTATUS ]]; then
    printf "\n--- resubmitter2 $1 $2 $3\n\n"
    printf "\n\nmaster not found at $2 on `date`, stopping... " >>
resubmitter2.log
    ./stop $1
    sleep 240
    printf "\n\n`date` resubmitting $1 $2 $3...\n" >> resubmitter2.log
    ./start $2 $3 $1
    sleep 240;
  fi;
fi

printf "...aaA.."
sleep 360
printf "Pffff..."
exec ./resubmitter2 $1 $2 $3 again
```

A6. remote

```
#!/bin/bash

if [[ -z $1 ]]; then
  echo "remote IP \"remote_command\" [-quiet]"
  exit;
fi

if [[ ! -z $3 ]]; then
  quiet=1;
else
  quiet=0;
fi

if [[ $quiet -eq 0 ]]; then
  echo "executing $2 at $1 in bash...";
fi

printf "#!/bin/bash\n\n$2" >temp.$1
chmod a+x temp.$1
mv temp.$1 cmd/$1

while [[ -e cmd/$1 ]]; do
  if [[ $quiet -eq 0 ]]; then
    printf ".";
  fi
  sleep 1;
done

if [[ $quiet -eq 0 ]]; then
  echo "remote command request accepted, waiting for result...";
fi

CNT=1
while [[ ! -e out/$1.remote ]]; do
  let CNT=$CNT+1
  if [[ $CNT -eq 600 ]]; then
    echo "waiting too long, giving up for $1."
    exit;
  fi
  if [[ $quiet -eq 0 ]]; then
    printf ".";
  fi
  sleep 1;
done
sleep 1;

cat out/$1.remote
rm -f out/$1.remote
```

A7. slstatus, onner

```
#!/bin/bash

# this script prints the status of runs on all [specified] nodes

cd really-on
ls $1 > ../status.tmp
cd ..
N=`wc -l status.tmp |sed "s/^\([0-9]*\) .*/\1/g"`
echo $N nodes are on...
G=`cat guarding.what`

for (( i=1; $i <= $N; i++ )); do
  IP=`head -$i status.tmp | tail -1`
  echo $IP: `./remote $IP "ps -Fu root" | grep $G | grep -v "guard\|scp"
|sed "s/root *\([0-9]*\).*bin\/\(ea.*\)-slave.*/\1 \2/g" |sed
"s/.*bin\/\(ea.*\)-master.*/MASTER: \1/g"` &
done

rm status.tmp

NT=`ps -F | grep remote`
S=0
while [[ ! -z $NT ]]; do
  NT=`ps -FA | grep remote | sed "s/^.*\(129.241.[0-9]*\.[0-9]*\).*$/\1/g"
| grep -v grep`
  sleep 1
  S=$S+1
  if [[ $S -gt 180 ]]; then
    echo "no response from:" $NT;
    killall remote
    killall slstatus
  fi;
done
echo "no more nodes."
```

```
#!/bin/bash

cd /cargo
rm -f really-on/* >& /dev/null
cp -f on/* really-on >& /dev/null
rm -f on/* >& /dev/null
sleep 2000
date >>/var/log/onner
exec ./onner
```

A8. Example of a submitted task: task-rep/seq82_ea_master

```
#!/bin/bash

cd /home/current3   # the current3 structure is created on node startup
scp cargo@search:current3/bin/* bin
scp cargo@search:current3/project/cargo/cfg/sequential82_e.prj
project/cargo/cfg
scp cargo@search:current3/project/cargo/cfg/sequential4.prj.?
project/cargo/cfg
scp cargo@search:current3/project/cargo/modules/* project/cargo/modules
scp cargo@search:current3/project/cargo/fsa-handmade/* project/cargo/fsa-
handmade
mkdir project/cargo/population/for_sequential
scp cargo@search:current3/project/cargo/population/for_sequential/1*
project/cargo/population/for_sequential
rm project/cargo/*skip*
rm -rf project/cargo/fsa/sequential/?
rm -rf ../out/*
rm -rf log/*
date
mkdir project/cargo/fsa
mkdir project/cargo/fsa/sequential
mkdir project/cargo/fsa/sequential/{0,1,2,3,4,5,6,7}
mkdir project/cargo/population
mkdir project/cargo/population/sequential
mkdir project/cargo/traj
mkdir project/cargo/traj/sequential
mkdir project/cargo/traj/sequential/{0,1,2,3,4,5,6,7}
bin/ea project/cargo/cfg/sequential82_e.prj -master >& ../out/ouput.txt
# --verbose
cd /home
echo "task: sending results back to server..."
ARCHIVEFILE=`cat /etc/ip`.`date +%F.%H-%M-%S.%N`.tgz
tar cvz out current3/log current3/project/cargo/fsa/sequential
current3/project/cargo/traj/sequential
current3/project/cargo/population/sequential current3/project/cargo/cfg
current3/project/cargo/modules >$ARCHIVEFILE
scp $ARCHIVEFILE cargo@search:runs
rm $ARCHIVEFILE
rm /home/current3/log/*
rm /home/current3/project/cargo/population/sequential/*
df
date
```

Appendix B – structure of the database tables used

The tables are created automatically by the EA master on startup, and deleted when computation completes. The table names are uniquely combined with the project name so that multiple projects can be running simulatenaously using the same database without interference (xxx stands for project name):

TODO_xxx	contains the list of currently computed individuals
SLAVES_xxx	contains the list of nodes that are ready to receive computational task
xxx_yy	contains all the fitness values of the individuals that were already evaluated (cache), and yy stands for incremental evolutionary step;
OBJINFO_xxx	contains detailed information from the runs in order to gain an understanding of how the individual achieved its score

Structure of these tables is shown in the following box:

```
mysql> describe TODO_sequential;
+----------+------------+-----+----------------+
| Field    | Type        | Key | Extra          |
+----------+------------+-----+----------------+
| ID       | int(11)     | PRI | auto_increment |
| GENOME   | blob        |     |                |
| VALUE    | double      |     |                |
| STATE    | int(11)     |     |                |
| SLAVE_ID | varchar(30) |     |                |
+----------+------------+-----+----------------+
5 rows in set (0.00 sec)


mysql> describe SLAVES_sequential;
+-------------------+-------------+-----+-------+
| Field             | Type        | Key | Extra |
+-------------------+-------------+-----+-------+
| SLAVE_ID          | varchar(30) | PRI |       |
| GENOMES_REMAINING | int(11)     |     |       |
| ISTEP             | int(11)     |     |       |
| STARTTIME         | int(11)     |     |       |
| SPEED             | float       |     |       |
| CMD               | text        |     |       |
+-------------------+-------------+-----+-------+
6 rows in set (0.00 sec)


mysql> describe cargo_sequential_0;
+-----------+---------+-----+----------------+
| Field     | Type    | Key | Extra          |
+-----------+---------+-----+----------------+
| id        | int(11) | PRI | auto_increment |
| cache_key | blob    | MUL |                |
| value     | double  |     |                |
+-----------+---------+-----+----------------+
3 rows in set (0.00 sec)
```

```
mysql> describe OBJINFO_sequential;
+----------+-------------+-----+----------------+
| Field    | Type        | Key | Extra          |
+----------+-------------+-----+----------------+
| ID       | int(11)     | PRI | auto_increment |
| VAL      | double      |     |                |
| OBJINFO  | text        |     |                |
| SLAVE_ID | varchar(30) |     |                |
+----------+-------------+-----+----------------+
4 rows in set (0.00 sec)
```

The following is a list of SQL queries performed by the EA master, followed by a list of SQL queries of the EA computational nodes.

Master preparing the OBJINFO table:

```
INSERT INTO OBJINFO_sequential (VAL,OBJINFO,SLAVE_ID)
     VALUES (0,'$x','legend')
```

```
$x="tot_time-obst_time-below_light-flw_line-line_below_lyit-total_dist-
moving_chngd-m_c_under_l-(active_count)-(script_count)"
```

Computational node checking if a crashed slave from the same machine isn't listed in the table:

```
SELECT * FROM SLAVES_sequential WHERE SLAVE_ID LIKE '$ip.%'
```

Computational node notifying master about a crashed slave:

```
UPDATE SLAVES_sequential
   SET GENOMES_REMAINING=SLAVE_TERMINATED,STARTTIME=STARTTIME-3600
 WHERE (SLAVE_ID='$brother.id')
```

where `SLAVE_TERMINATED` is a special constant indicating to master that the node crashed and `$brother.id` is the identifier of the crashed node.

Computational node notifying master that it is ready to evaluate individuals:

```
INSERT INTO SLAVES_sequential
           (SLAVE_ID, GENOMES_REMAINING, ISTEP, STARTTIME, SPEED, CMD) VALUES
           ('slave.id', 0, 0, 0, cpu_benchmark(), 'none')
```

where `cpu_benchmark()` is a function that computes relative speed of the node so that the master can prefer faster nodes when possible.

Master notifying the computational nodes that the job is completed and it is going to terminate. (the nodes will then change the value to 0 indicating that they have received the message and terminated, so that the master will cleanly drop the table afterwards):

```
UPDATE SLAVES_sequential SET GENOMES_REMAINING=-1
```

Master checking the number of nodes that still did not terminate:

```
SELECT COUNT(GENOMES_REMAINING) FROM SLAVES_sequential
       WHERE (GENOMES_REMAINING=-1)
```

Master checking before computation the slaves that are ready to receive the individuals, they are ordered so that the faster nodes will receive more individuals, if the distribution is not completely even:

```
SELECT SLAVE_ID, SPEED FROM SLAVE_sequential ORDER BY SPEED DESC
```

Master sending the individuals to be computed:

```
INSERT INTO TODO_sequential (genome, value, state, slave_id)
            VALUES ('$genome_representation', 0.0, 0, 'none')
```

where `$genome_representation` is a blueprint of the genotype into a numeric string.

Those that are already in the cache are removed from the `TODO` list:

```
DELETE FROM TODO_sequential USING TODO_sequential, sequential_0
       WHERE sequential_0.cache_key=TODO_sequential.genome
```

Assigning the work to the slaves:

```
UPDATE TODO_sequential SET slave_id='$node_id' WHERE slave_id='none'
       LIMIT $number_of_ind_assigned
```

And letting them know that the work is ready to be processed:

```
UPDATE SLAVES_sequential
   SET GENOMES_REMAINING=$number_of_ind_assigned,
       ISTEP=$istep,
       STARTTIME=$time_work_start
 WHERE SLAVE_ID='$node_id'
```

Checking if any node has crashed:

```
SELECT SLAVE_ID FROM SLAVES_sequential
       WHERE GENOMES_REMAINING=SLAVE_TERMINATED
```

Checking if any node completed its work package:

```
SELECT SLAVE_ID, STARTTIME, GENOMES_REMAINING FROM SLAVES_sequential
       WHERE (GENOMES_REMAINING > 0)
```

If some node did not complete in time, look for alternative slaves that can take its work:

```
SELECT SLAVE_ID FROM SLAVES_sequential WHERE (ISTEP=$istep)
   AND (SLAVE_ID!='$slow_slave') ORDER BY GENOMES_REMAINING, SPEED DESC
```

Delete the node that did not deliver the result in time:

```
DELETE FROM SLAVES_sequential WHERE SLAVE_ID='$slow_slave'
```

Reassign the work of one node to another node:

```
UPDATE TODO_sequential SET SLAVE_ID='$free_slave' WHERE
(SLAVE_ID='$slow_slave') AND (STATE=0)
```

And update its work package so that it knows that it has more to compute:

```
UPDATE SLAVES_sequential SET GENOMES_REMAINING=GENOMES_REMAINING +
$genomes_remaining, STARTTIME=$current_time WHERE SLAVE_ID='$free_slave'
```

Copy all computed fitness values to the cache:

```
INSERT INTO sequential_0 (cache_key, value) SELECT GENOME, VALUE FROM
TODO_sequential
```

If the cache is not used, load the computed values directly:

```
SELECT VALUE from TODO_sequential
```

Clean the `TODO` list for the next generation:

```
DELETE FROM TODO_sequential
```

Computational node checking if a new work is ready:

```
SELECT GENOMES_REMAINING, ISTEP, CMD FROM SLAVES_sequential
       WHERE SLAVE_ID='$slaveid'
```

Computational node notifies the master that it has received a message to terminate:

```
UPDATE SLAVES_sequential SET GENOMES_REMAINING=0 WHERE SLAVE_ID='$slaveid'
```

Computational node notifies the master that it has finished processing the requested command:

```
UPDATE SLAVES_sequential SET CMD='none' WHERE SLAVE_ID='$slaveid'
```

Computational node retrieving the work package to be computed:

```
SELECT ID, GENOME FROM TODO_sequential
       WHERE (SLAVE_ID='$slaveid') AND (STATE=0)
```

Computational node saving the fitness value back to the table:

```
UPDATE TODO_sequential SET VALUE=$val, STATE=1 WHERE (ID='$id')
```

Computational node saving detailed information about a particular run (only runs with best fitness so far are saved):

```
INSERT INTO OBJINFO_sequential (VAL,OBJINFO,SLAVE_ID)
       VALUES ($val,'$oi_description','$slaveid')
```

Computational node reporting to the master that it has completed its work package:

```
UPDATE SLAVES_sequential SET GENOMES_REMAINING=0 WHERE (SLAVE_ID='$slaveid')
```

Master sending a shell command to be executed at all the compuational nodes, this can optionally occur at the end of each incremental step:

```
UPDATE SLAVES_sequential SET CMD='$slavecmd[istep]'
```

Master waiting for the compuational nodes to complete the shell command:

```
SELECT CMD FROM SLAVES_sequential WHERE (CMD!='none')
```

Master counting the number of records in the cache for statistical purposes:

```
SELECT ID FROM cargo_sequential_$istep
```