Alma Mater Studiorum
Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA – DIPARTIMENTO DEIS

SCUOLA DI DOTTORATO IN SCIENZE E INGEGNERIA DELL'INFORMAZIONE

Dottorato di ricerca in
Ingegneria Elettronica, Informatica e delle Telecomunicazioni

XX Ciclo

Settore scientifico disciplinare:
ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Enabling computationally intensive bioinformatics applications on the Grid platform

Ph.D. Dissertation

Autore
Dott. Ing. Gabriele Antonio Trombetti

Coordinatore	**Relatore**	**Correlatore**
Chiar.mo Prof. Ing. Paolo Bassi **(DEIS)**	Chiar.mo Prof. Ing. Maurelio Boari **(DEIS)**	Dr. Luciano Milanesi, Ph.D. **(CNR-ITB)**

Esame Finale anno 2008

1 Table of Contents

2 Abstract

Bioinformatics is a recent and emerging discipline which aims at studying biological problems through computational approaches. Most branches of bioinformatics such as Genomics, Proteomics and Molecular Dynamics are particularly computationally intensive, requiring huge amount of computational resources for running algorithms of ever-increasing complexity over data of ever-increasing size.

In the search for computational power, the EGEE Grid platform, world's largest community of interconnected clusters load balanced as a whole, seems particularly promising and is considered the new hope for satisfying the ever-increasing computational requirements of bioinformatics, as well as physics and other computational sciences.

The EGEE platform, however, is rather new and not yet free of problems. In addition, specific requirements of bioinformatics need to be addressed in order to use this new platform effectively for bioinformatics tasks.

In my three years' Ph.D. work I addressed numerous aspects of this Grid platform, with particular attention to those needed by the bioinformatics domain.

I hence created three major frameworks, Vnas, GridDBManager and SETest, plus an additional smaller standalone solution, to enhance the support for bioinformatics applications in the Grid environment and to reduce the effort needed to create new applications, additionally addressing numerous existing Grid issues and performing a series of optimizations.

The Vnas framework is an advanced system for the submission and monitoring of Grid jobs that provides an abstraction with reliability over the Grid platform. In addition, Vnas greatly simplifies the development of new Grid applications by providing a callback system to simplify the creation of arbitrarily complex multi-stage computational pipelines and provides an abstracted virtual sandbox which bypasses Grid limitations. Vnas also reduces the usage of Grid bandwidth and storage resources by transparently detecting equality of virtual sandbox files based on content, across different submissions, even when performed by different users.

BGBlast, evolution of the earlier project GridBlast, now provides a Grid

Database Manager (GridDBManager) component for managing and automatically updating biological flat-file databases in the Grid environment. GridDBManager sports very novel features such as an adaptive replication algorithm that constantly optimizes the number of replicas of the managed databases in the Grid environment, balancing between response times (performances) and storage costs according to a programmed cost formula. GridDBManager also provides a very optimized automated management for older versions of the databases based on reverse delta files, which reduces the storage costs required to keep such older versions available in the Grid environment by two orders of magnitude.

The SETest framework provides a way to the user to test and regression-test Python applications completely scattered with side effects (this is a common case with Grid computational pipelines), which could not easily be tested using the more standard methods of unit testing or test cases. The technique is based on a new concept of datasets containing invocations and results of filtered calls. The framework hence significantly accelerates the development of new applications and computational pipelines for the Grid environment, and the efforts required for maintenance.

An analysis of the impact of these solutions will be provided in this thesis.

This Ph.D. work originated various publications in journals and conference proceedings as reported in the Appendix. Also, I orally presented my work at numerous international conferences related to Grid and bioinformatics.

3 Introduction

This Ph.D. thesis reports my research work addressing numerous aspects of the Grid platform, with particular attention to those needed by the bioinformatics domain.

Bioinformatics is a recent and emerging discipline which aims at studying biological problems through computational approaches. Most branches of bioinformatics are particularly computationally intensive, requiring huge amount of computational resources for running highly complex large-scale algorithms over data of ever-increasing size.

In the search for computational power, the EGEE Grid platform, world's largest community of interconnected clusters load balanced as a whole, seems particularly promising and is considered the new hope for satisfying the ever-increasing computational requirements of bioinformatics, as well as physics and other computational sciences.

In the subsections of this chapter 3, firstly a brief introduction will be given to the bioinformatics discipline, explaining its history, purposes and peculiar needs, then the EGEE Grid platform will be introduced in greater detail, with an accent on discussing the major issues of this platform in section 3.2.2, which have been addressed by my research work.

The subsequent Chapters are devoted to the description of my research work on this subject.

In Chapter 4, four solutions will be presented in detail (respectively in chapters 4.1, 4.2, 4.3 4.4) for solving the main issues of the EGEE Grid platform. These solutions tackle the problems sinergically from many aspects. The solutions:

- create abstractions with reliability over the unreliable Grid environment
- increase the effective grid uptime
- bypass some Grid limitations and automate some procedures
- ensure proper cleanup of common Grid resources automatically
- increase the Grid performances

- reduce Grid storage costs
- reduce Grid bandwidth usage at multiple locations
- reduce the time and effort needed to *write* applications in the Grid environment
- reduce the time and effort needed to *test* applications in the Grid environment
- automatically maintain and manage specific data commonly needed by bioinformaticians, dynamically balancing between various types of costs, depending on the amount of usage
- provide a mechanism to help programmers write arbitrarily complex multi-stage pipelines or workflows for the Grid environment

In chapter 5, attempts will be made at evaluating the exact impact of these solutions in the Grid environment. Firstly, in 5.1 an evaluation of the increase in Grid performances will be given through monte-carlo simulations based on experimentally probed data and simulated submission algorithms. Then, in 5.2, the reduction in time and effort due to the solutions in chapter 4 will be examined. In 5.3 some more benefits will be mentioned. Finally, in 5.4 a case study will be presented showing how one of the frameworks was used bring to the grid a complex multi-stage computation pipeline.

In chapter 6, the conclusions of my Ph.D.'s work will be discussed together with the direction of the imminent future work.

In chapter 7 - *Appendices*, some low-level implementation details will be given about the solutions presented. Additionally, in 7.3 a tutorial section giving hints on how to develop effective applications in the Grid environments will be given, based on my three years' Ph.D. experience in the Grid environment and being part of the BioinfoGRID [BIOINFOGRID] project. Finally, in 7.4 the list of publications and oral presentations in international conferences related to my Ph.D. work will be listed.

3.1 Bioinformatics

Bioinformatics is an emerging discipline that leverages computer science means and techniques for addressing biological problems. The discipline is very new, first appearing in scientific journals at the end of 1980s.

Bioinformatics was first born with the aim of managing the enormous amount of data coming from the Human Genome Project [HGP] and that of developing algorithms for mining through such large amount of data to gather meaningful information. That specific branch of bioinformatics is now more specifically called *Genomics*.

Bioinformatics has since then evolved to cover a number of other fields of study such as

- **Proteomics:** large scale study of proteins, particularly aimed at studying their structures and sub-parts (domains) in relation with the structure of other known proteins. The mechanism of action of some proteins can hence be guessed on the basis of similes with other proteins or domains for which the action is known or partly known.
- **Molecular Dynamics:** studies the folding of protein atoms in the 3D space and interactions between proteins and other proteins or proteins and other molecules. This is performed through computer simulation of the laws of physics. Each protein contains thousands of atoms.
- **Phylogenetics:** studies the amount of evolutionary relationship among various groups of organisms usually through comparative analysis of the genomes associated with complex statistical simulations (e.g. Hidden Markov Model [HMM1], [HMM2]).
- **Systems Biology:** probably the newest branch of bioinformatics, Systems Biology studies the interaction between the various components of biological systems in order to understand the functions and behaviour of the system. This is achieved through mathematical equations and computational modelling.

...and an uncountable number of other often interconnected sub-disciplines.

Bioinformatics as well as physics and other mathematics-based experimental sciences is constantly in great need of always new computational powers which can be used to solve algorithms of ever-increasing complexity over data of ever-increasing size.

Genomics for example, has recently seen great increases in the amount of data being generated and which has to be analyzed, due to pyrosequencing and other novel DNA sequencing techniques increasing sequencing speeds by an order of magnitude. Considering the current trend and the enormous amount of money being spent each year for new sequencing technologies, it is foreseen that a complete human genome (that's 3 billion nucleotide bases) could probably be sequenced for just $1000 in 10 years from now. This would allow most people on the planet to have their complete genome sequenced, and would simultaneously create amazing challenges for the computing hardware and bioinformatics algorithms needing to mine through such amount of information for finding meaningful results.

In addition, new approaches usually referred to as "genotyping" aimed at sampling a large number of highly variable and probably meaningful points of the human genome called SNPs, are already a reality today. I will cite for example the Illumina beadstation [BEADSTATION] which can now sample 1 million SNPs out of a human DNA for about $500 in one hour. Even though the amount of information in this case is smaller than in the case of raw genome sequencing, the density of meaningful information which has to be analyzed by informatics means in this case is very high.

In this scenario, the Grid computing (see next chapter) appears as a hope for bioinformatics, a dream which might be able to provide enough computational power for bioinformaticians' needs already *today and on existing hardware*, for solving some of the most complex computational needs.

3.2 The EGEE Grid

3.2.1 Definition and history of the EGEE Grid

The term Grid Computing refers to a distributed infrastructure to allow the usage of computational and storage resources coming from an indistinct number of computers (each of these being of not particularly high power) interconnected via a network which is usually, but not necessarily, the Internet.

The term "grid" comes from the first devisers of Grid Computing which foresaw the advent of a time in which people would be using computational resources in the same way as power resources, that is, just connecting a plug to the power grid. The simile is actually quite unfortunate as one of the main problems of Grid Computing is its high complexity, which makes usage most hard and complicated, in contrast to just connecting a plug.

The idea of Grid Computing stemmed from realizing that, on average, the usage of computational resources in an organization is around 5% of maximum. With the Grid, the resources would be shared among various entities so to create a virtual organization with a much better infrastructure that the single entity could afford.

A grid is able to give users access to the computing power and storage of a distributed system, while the users can still be a virtual group not belonging to any single geographic location. The Grid guarantees a coordinate and checked access to shared resources and offers to the user the visibility of a single logic computation system where jobs can be submitted.

"Computing Grids" are mostly used to solve large-scale computational problems in the scientific and engineering subjects. Originarily evolved from High Energy Physics (HEP), their role is now extended to biology / bioinformatics, astronomy, and other subjects. Biggest commercial IT players have shown significant interest to the phenomenon, and have been collaborating in the main worldwide Grid projects as sponsors and/or developing their own Grid projects, expecting a future widespread use also in the commercial fields.

Another significant phenomenon to mention is the birth of numerous small-scale implementations of grids on local or metropolitan networks. These maintain the carachteristics of a grid and are usually referred to with the terms Local Area Grid (LAG) and Metropolitan Area Grid (MAG). These are in a sense similar to intranets, and might constitute an infrastructure for distributed computing in a company environment, while nation-wide grids might eventually (hopefully) join together and constitute a global scale World-Wide Grid.

EGEE (Enabling Grids for E-Science) [EGEE] is the current EU-funded project for a common research european grid and is the world's largest grid as of today. Started in 2004 within the fifth framework programme, went on in the sixth framework programme as EGEE-II and will be followed up by an EGEE-III in the seventh. EGEE leverages the existing infrastructure of the LCG (LHC Computing Grid) intended to perform the data analysis for the

CERN's LHC (Large Hadron Collider) when this is ready. As middleware, the EGEE project started out by using the earlier LCG-2 middleware, originated from the earlier EU grid project EDG (EU Data Grid) then started developing the next generation middleware gLite. All middleware versions are layered one on top of the other so that earlier versions (such as LCG-2) can still be used on the newer EU grids without problems, and in fact, for some middleware commands and libraries newer versions are no more than wrappers to the older versions.

3.2.1.1 Structure of the EGEE Grid

The EGEE Grid [EGEE] is a widely distributed and not centrally administered structure, composed of the following main elements (see figure 1 below):

- **CE (Computing Element):** this computer is the master node for a cluster of **WN (worker nodes)** located behind it and acts as gatekeeper and workqueue manager for such cluster. The Worker Nodes provide the real computational power of the Grid. The queueing system being used is usually PBS [PBS] or equivalent system, but is not accessible as it is abstracted to the user by the Grid environment.
- **SE (Storage Element):** a computer sporting large disk arrays which provides the storage capacity to the Grid environment. There should be one of these located near (within a Local Area Network) to every CE of the Grid so to provide large bandwidth to the Worker Nodes. Big amounts of data can be uploaded to a Storage Element before submitting computation jobs to the Grid. Jobs can be locked to go executing on a CE near a SE which holds a replica of the data they need. There can be multiple replicas on different locations for a user-uploaded file, up to one replica for every SE of the Grid.
- **RB (Resource Broker):** a computer whose main task is to receive jobs submitted through the User Interfaces. The RB routes such jobs to a CE capable of fulfilling the requirements for the execution of the jobs, as specified by the user in a so-called **JDL file** (from the name of its syntax: Job Description Language) at the time of submission. From the list of the CEs capable of fulfilling the request, the RB chooses the CE which is estimated to have the shorter execution queue. This evaluation is

performed through the information received by the BDII (see).

- **BDII (Berkeley Database Information Index):** this computer gathers a significant amount of statistical information from mostly any other element of the Grid, in particular the CEs and the SEs. From the CEs it receives the number of CPUs and their power, their busy/free state, the number of jobs waiting in the queue, the number of jobs executing and their provenance for accounting purposes (mainly, the VO). From the SE it receives the amount of disk space allocated and free. This information is feeded to the RB for helping it perform queue times estimation, in addition these are used for monitoring and accounting purposes.
- **UI (User Interface):** This is a user owned computer and strictly speaking not part of the Grid. This computer is what the Grid-user uses for interfacing himself/herself to the Grid, i.e. for submitting jobs, receiving results and doing file operations on the Storage Elements. The User Interface communicates prevalently with the RB for job management, but can also communicate the the SEs and the Replica Catalog. The UI runs a subset of the gLite Grid middleware in order to be able to communicate with the other elements of the Grid.
- **Replica Metadata Catalog (RMC), Local Replica Catalog (LRC) and LCG File Catalog (LFC):** These computers manage a virtual hierarchical filesystem for **Logical File Names (LFN)**. LFN are human readable names which can be associated to the cryptic names automatically generated when a file is uploaded to a Storage Element. Actually the automatically generated names are in **SFN (Storage File Name)** or analogous **(SRM)** form, which are then mapped to a **GUID** (Globally Unique IDentifier) by the RMC/LRC/LFC, and while there will be different SFN/SRM for each replica of the same file, there will be only one GUID. There can be multiple LFN (human readable names) for each GUID. The LRC/LFC allow bidirectional navigation between the three mappings (SFN or SRM / GUID / LFN).
- **VO (Virtual Organization):** this (not shown in figure) is not a computer but still an important constituting part of the Grid. The users which decide to join the Grid will be part of a Virtual Organization, which is not localized to a physical address like a normal organization but rather distributed, scattered across the planet. The amount of computation

consumed by each user is accounted firstly to the Virtual Organization and then possibly to the specific user. Grid site owners and VO managers might set a fee for the amount of computation being used (though this probably never happened up to now) or more likely ask the most active grid users to share some of their computational resources to compensate. Virtual Organizations also have a central administration for distributing Grid certificates for granting access, and a Responsible Person which monitors the good behaviour of the Grid users belonging to the VO, possibly denying further access to vile users in extreme cases.

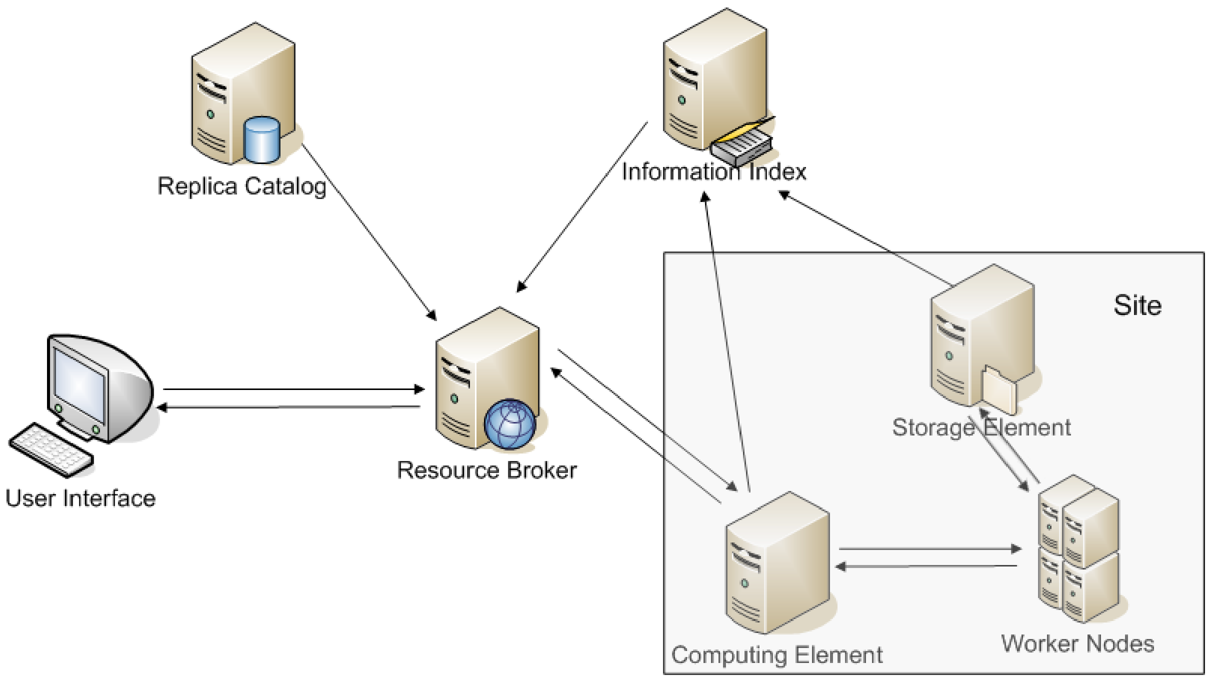


Illustration 1: Elements of the Grid and intercommunications

The Site in the figure 1 shows a group of computational and storage resources administered by an Organization which decided to actively join the Grid also sharing their computational resources. The said Organization will be part of a larger VO, and their computational resources will be usable by at least the VO where they belong, but can be used by more VOs, depending on their specific agreements of collaboration with the other VOs.

3.2.2 Existing issues

Notwithstanding the great efforts of many entities which collaborated for long years to the development of the EGEE Grid, due to the extremely distributed nature of such platform developing a flexible and stable grid interface is not easy and many issues remain. In the next subsections some of the main issues of the EGEE Grid will be summarized.

In my work I have addressed many of those issues, providing frameworks that either work completely around the limitations or significantly simplify such operations for the end user (in addition to other features which will be described later in this thesis).

3.2.2.1 Jobs failures

Not all the jobs which are submitted to the Grid execute successfully. In fact, excluding programming errors due to the Grid user, the EGEE Grid has still an unfortunately high percentage of job failures. (A quantitative measurement of the failure rate is in section 5.1.2.1 of this thesis)

Job failures happen in many forms, the following are the main ones:

1. A Job (J) is submitted by the user (U) and end up in a queue on a CE which appears very slow, and does not process jobs fast enough for J to reach execution before expiration of the Grid proxy (usually 48 hours). J dies in the queue.
2. A Job (J) submitted by the user (U) is assigned to a CE (C). The Job is reported to immediately enter execution after a very brief interval in the queue, but then, almost immediately, aborts.
3. A Job (J) submitted by the user (U) is assigned to a CE (C). The Job stays in the queue for some time then enters execution. The job executes for some time but as soon as it tries to contact the closest SE (S) for downloading a file or for uploading results, it fails.

The first two cases are most likely due to some misconfiguration of C but there are other responsibilities.

The first case is also partial responsibility of the RB or BDII which have not detected the extremely long queue and decided to schedule the job in the wrong place.

The second case also shows how C's failure rate does not influence in any way the ranking made by the RB for deciding where to route incoming jobs

(C should have ranked low and hence jobs should have not been scheduled there), and this is, in my humble opinion, a lack.

The third case can be a misconfiguration of C or S, or could be some error in the authentication subsystem of the middleware at the time of J attempting access to S.

Summing up, while for most job failures there is a human responsibility, it is still true that the Grid middleware does not do much to detect and provide automatic countermeasures to this kind of errors, which are so frequent that the Grid user simply cannot ignore them and really needs to handle them.

The simplest way for the user to handle job failures is just to resubmit failed jobs, and this is what is usually done, more or less manually. In extreme cases, the user might want to ban some misbehaving CEs by adding an apposite line in the JDL file when submitting the job, and/or contacting only trusted SEs.

3.2.2.2 Sandbox (size)

Jobs submitted by the Grid user allow for up to 10 (usually, depends on the VO) megabytes of data submitted together with the job.

Ten megabytes are insufficient for the majority of bioinformatics task, and in addition, only a few single files are allowed, not arbitrarily complex directory structures such as those of installed applications or libraries. Hence the user always needs to package directory structures in archives manually, and write code in the job so to unpack these.

When the 10 MB sandbox size is insufficient, Storage Elements have to be used for storing the needed files. This adds another layer of complexity, since tar or similar archives are to be made, files are to be manually uploaded to the SEs, code has to be written in the job for connecting to a SE and downloading the files, and for unpacking. In addition, the user **must** remember to manually delete the files from the SEs when these are not needed anymore, so to free disk space for the other Grid users.

The whole procedure is highly uncomfortable to non-computer-scientists and very error prone. In addition, the SEs risk to remain polluted due to forgetfulness of the users (also see next section).

3.2.2.3 Files maintenance on SEs and RMC

Manual management of SE allocation (as described in previous section) is highly uncomfortable for non-computer-scientists, and very error prone. The main problems which consistently happen with this procedure are:

- Users forget to delete files belonging to them from the SEs after finishing using the files. SE are wildly over-allocated due to this reason and space is now lacking. Browsing the LFN hierarchies you can find files which are years old and probably not used since short after their upload.
- Wild allocation of LFN (human readable) file names by users, most of which don't bother to make one subdirectory relative to their (physical) organization and simply pollute the root directory of the LFN filesystem. I remember having seen more than a thousand files in the root of the biomed LFN hierarchy (then the user responsible for this was banned, in this case).
- RMC is still very unreliable as of today, and often loses LFN names. This is unfortunate because most people rely on LFN names for finding their files, as the automatically generated ones are very long and cryptic, and impossible to remember (one would need to write a software that remembers the mapping, but this is exactly what the RMC was supposed to be doing). People relying on the LFN name only (most do), will not be able to access their uploaded files, or even delete them from the Storage Elements once the RMC loses the LFN reference. Storage Elements are overallocated also for this reason

3.2.2.4 Downtimes

Every element of the Grid has some downtimes, and this is understandable since this is a distributed architecture, not centrally administered, and each part is without official guarantees of uptime.

Downtimes cause some problems though, especially those related to the Storage Elements and Resource Brokers. While the former ones are relatively rare, the latter are particularly common. This might be because of the high load the RBs experience during large job submissions. In fact I have seen downtimes of around 5% for Resource Brokers, usually lasting one or two days.

RB downtimes cause significant problems because User Interfaces are programmed to use only one Resource Broker. When the configured resource broker is down, two things happen:

1. The User Interface cannot submit any new job to the Grid
2. It will not be possible to get the status or the results for all jobs submitted previously, which were submitted through the RB which is now down.

While the first point could be fixed by changing the configured resource broker (this requires intervention of the system administrator and also it is not a totally straightforward procedure) there is no possible fix for the second point.

In addition to these downtimes, Worker Nodes and RMC/LRC/LFC are sometimes very slow. WN slowness cause the problems already described in the 3.2.2.1 - *Job Failures* section above, while RMC/LRC/LFC slowness have the effect of significantly slowing down and annoying the users when they need to perform manual interactive operations on the SE / LFN filesystems (such as e.g. uploading of files which don't fit into the 10MB sandbox and/or removing old files).

3.2.2.5 Waste of bandwidth and execution time

In case of computations which need to access large files such as a multi-GB biological database (this is common in bioinformatics), the Grid is not optimal in the way bandwidth and computation time are used.

Biological databases are files which are commonly in the range 500MB-5GB, created and maintained by reputable bioinformatics institutes such as NCBI, EMBL and EBI. These kind of files are commonly uploaded once (manually) and then left on the Grid forever, since it is known that they will be reused sooner or later.

However, with this kind of manual management, the number of replicas for every one of these files is static, and is usually one.

Jobs on the Grid can be constrained to go executing near (within the Local Area Network) to a SE where a replica of the file they need is located.

However this cannot reasonably be done when only one replica is available, as only one CE would be usable in this case, and this would mean not exploiting the power of the Grid.

On the other hand with a higher number of replicas for files of this size the storage costs increase wildly. Leaving a multi-GB file with multiple replicas uploaded on the Grid for long times while it is not being used is considered unpolite and against the policies of the EGEE Grid.

It also appears too bothersome for the Grid user to upload the biological database to multiple locations only when needed (this also takes a significant user time if done interactively), and practically nobody does this as far as I know.

What is usually done is that the database is uploaded once and then left on the Grid with a single replica, and jobs are *not* constrained to go executing near the replica.

This is also not optimal because every job then needs to download the database to the WN prior to starting the computation. This is a remote download of a multi-GB file from the Internet, and this wastes a significant bandwidth. In addition, in case of many jobs launched in parallel (very common case), these will all start downloading at similar times so the impact will be multiplied and will easily saturate all the bandwith of the SE where the database was uploaded for hours. This in turn renders their own download slow, and also that of other users' jobs relying on the same SE.

In addition to this, there is another problem: the CEs have a scheduler and queuing system that do not let other jobs execute on the CPU that is already occupied by one job, whatever it is doing, computing or downloading, until the job exits or its proxy certificate times out. Hence, if the job wastes its time for downloading a multi-GB file from a remote location, the job is wasting CPU time for everybody, in addition to the waste of bandwidth.

Clearly the Grid needs a better solution.

In addition to this I will mention that there is no automatic mechanism on the Grid for keeping such kinds of databases updated, e.g. reflecting updates from the FTP sites of the maintainers. This means that most bioinformaticians update the databases they use more or less manually by replacing it with the new version every some time. This is really not a Grid's fault, since this requirement is specific only to bioinformaticians, however it needed to be fixed for our case. Hence, I also worked in this direction as you can see in section 4.2 – *BGBlast and the Grid Database Manager*.

3.2.2.6 RB suboptimal CE ranking

The Resource Broker (RB) CE ranking algorithm is based on estimated queue times of computing elements and incoming jobs are normally routed by the RB to the CE ranked highest among the Computing Elements being compatible with the job requirements (which are specified at submission time). A number of problems exist in this procedure which will be described here.

The ranking trusts the grid BDII service, which is not updated in real time, it has a certain propagation time. When submitting a sequence of two jobs using a resource broker, in the short time gap between the two submissions the LB service won't be able to notify the RB itself of the new estimated queue time for the CE where the first job has just been assigned. The RB itself doesn't have the required information to compute this new queue estimation by itself, hence, such CE has the likelyhood to remain first ranked with regard to the queue time at the time of the second job submission. In a long sequence of jobs submissions using the same or even different RBs, many jobs might be assigned to the same CE, thus not really performing a load balancing.

However, it is not easy to fix such a problem. Considering that multiple RBs exist for each VO, and these are relatively pluggable and can be added and removed from the Grid, most design changes would fail either in effectiveness or in scalability.

Recently, the probabilistic ranking has been introduced, which should allow a better load balancing, with regard to the previously mentioned problem. In this ranking mode, the estimated queue time for a CE is used to compute the probability that the job is delivered to a certain CE, so the job will not be necessarily submitted to the CE which is at the top of the list.

Currently, the deterministic ranking still remains the default for the EGEE Grid and most Grid users don't override this setting. This is partly because users are not aware of the option or do not care, and partly because the users hope to get a faster response time with the deterministic ranking, notwithstanding the problem mentioned above.

The fact that most Grid users use the deterministic ranking worsens the RB ranking reliability itself somewhat, because CEs that are declared by the LB service as having a short queue at a certain point in time, will be flooded by jobs by most RBs and most users until the next LB information update. This creates queues on the flooded CEs, that result in "unexpected" wait times for the jobs themselves.

Another serious problem, orthogonal to the ones mentioned above, is that the queue time estimation is based on insufficient data. Basically, only the number of CPUs in a CE, the declared power of these, and the number of jobs in the queue can be used for the queue time estimation. These data are clearly insufficient if the expected running lenght for the jobs in the queue is not known, and leads to gross mistakes in the queue time estimation. Unfortunately, the EGEE Grid does not have a mechanism for declaring the expected running length for a job, in contrast to e.g. PBS [PBS] (PBS enforces the termination of a job which goes beyond its declared running time, and this leads to a nice predictability of the queuing system not available on the Grid).

Unfortunately there are not many fixes available for all the problems mentioned above. Other than using the probabilistic ranking when this seems appropriate, my suggestion still remains that of monitoring the queue times of submitted jobs and killing-resubmitting jobs when these appear to be stuck in the queues. These actions are performed automatically by the Vnas framework I developed in my Ph.D. and which is reported in this thesis, which relieves the user from the burden to write automated code for this task.

An effort by the EGEE developers is in the direction of providing a pull-mode in the middleware. In this mode the CEs would pull jobs from the Resource Brokers, which would handle the queues of jobs in place of the CEs. Unfortunately, the fact that in this mode every RB has a queue which is separate from that of the other RBs might just move the problem of the queue estimation from the RBs to the UIs, and from the automatic matchmaking/ranking system to the hands of the users. This design decision was needed in order to share the great load of the queues handling among the various RBs, considering that this creates a burden an order of magnitude higher than with the push-mode. Notwithstanding this problem, many issues of the Grid might be resolved by this pull mode when it will be released. The pull mode is currently in alpha testing stage within the EGEE project.

4 Improving Grid support for bioinformatics applications

As was mentioned in the previous chapter, the EGEE Grid appears as very promising platform which can provide an unprecedented computational power for the scientists already *today* for performing computationally intensive researches in various scientific fields such as bioinformatics, physics and other mathematics-based sciences.

However, as was mentioned in section 3.2.2 – *Existing Issues*, the Grid is a very new architecture, not yet free from problems. In addition, work was needed for better supporting specific requirements of the bioinformatics domain in the Grid environment.

In my three year's Ph.D. course I hence created three major frameworks, Vnas, GridDBManager and SETest, plus a smaller standalone solution for addressing RB downtimes.

The presented solutions help in addressing a number of issues, either related generically to the Grid or specific to the bioinformatics domain. The four solutions work sinergically in the direction of optimizing Grid performances, increasing the user-perceived platform reliability, optimizing Grid storage and bandwidth usage, reducing the time and effort needed for the development of new Grid applications and reducing application testing times.

These solutions will be described in detail in the following subsections.

4.1 Vnas

This project started as a support for the the high performance cDNA analysis pipeline project, reported separately as a case study in section 5.4 .

The pipeline in question was a complex achievement for the Grid because it was multi-stage, and couldn't easily be supported in the Grid environment due to the lack of a callback system which could launch a second computation step when all the jobs belonging to the first step completed.

In addition, I was strongly feeling the lack of a job management system that could monitor Grid jobs and resubmit them in case of failure until they finally completed, otherwise the pipeline computation steps would have failed too easily. Also please note that the management and resubmission should have happened at a *lower* level than the callback system, so that the resubmission would occur transparently to the callback system, and the computation stage wouldn't fail until the maximum number of resubmissions for a single failing job had been exceeded!

The Virtual sandbox feature was also used in its very early implementation, as the files needed for the computation in the pipeline were exceeding 10MB by far.

Then the Vnas system evolved to refine and optimize such embrional features, and others were added, finally resulting in the Vnas as it is described in the following sections. Vnas has since then been used as a submission system in countless other projects by the Bioinformatics group at CNR-ITB for reliably launching computational challenges in the Grid environment.

The following subchapters are laid out as follows: first the motivation for Vnas will be described, then the features, then an overview of the internal functioning, and lastly I will describe the problems that still persist when developing Grid applications notwithstanding the use of Vnas.

4.1.1 Motivation

The idea of Vnas stems from the need to overcome the many Grid limitations described in section 3.2.2 – *Existing Issues* in particular those regarding jobs submission and monitoring, the sandbox management and the Storage Elements management.

In particular, the major problems related to grid jobs submission which I addressed with Vnas are the following:

- Limited sandbox: Users of the Grid have got a limited space to store data and executables they need for their job. Current limit is 1 MB for most users. All files not fitting into 1 MB need to be uploaded on Storage Elements (SEs) separately by hand, and need to be downloaded on the worker node prior to execution. Files stored on

SEs need to be deleted by hand when no longer needed or they would waste shared resources.

- Flat sandbox and Storage Elements: The Grid sandbox and the SEs can only bear raw files. If a job needs a nested directory structure this has to be created with a custom code on the job. Tar archives are also to be packed and unpacked manually.
- Needless reuploading of the same files: If many users need the same files, every user will need to upload the files on SEs unbeknownst to the others, wasting bandwidth and storage space.
- Slices management: Commonly, in order to improve performances on the Grid, conceptually monolithic jobs are first divided into computation steps by creating a pipeline, and then each pipeline step is split in smaller “slices” which can execute in parallel. Slices are then submitted to the Grid separately. Unfortunately, the code to poll and wait for all slices to complete execution, fetch results, and launch the next computation stage is to be created as custom code for each application, which is considerably time consuming for the developer.

In addition to these objective problems of the Grid, the quantity of details that the user has to keep in mind for producing effective and "nice" (wrt. sane resource usage and cleanup) grid jobs is simply too great not to overlook some, in addition this fact discourages the casual bioinformatics user from approaching the Grid. Automating these tasks, Vnas reduces the risk of mistakes which might result in unwise resource exploiting by the non-grid-proficient, while simultaneously helping prospective users approach the grid.

4.1.2 Vnas features

In this chapter an overview of Vnas's features will be given, while the details of internal Vnas functioning can be found in the next chapter.

- Virtual Sandbox: sandbox emulation providing a sandbox of unlimited[1]

[1]Please note that while there is no limitation imposed by Vnas on the size of the Virtual Sandbox, the Virtual Sandbox is only an abstraction substituting a manual upload of the files onto the Storage Elements. The Virtual

size also allowing nested subdirectories. This prevents (1) the need to manually upload big files to storage elements (2) the need to pack and upload archives for recreating directory structures needed for the job to run and (3) the need to write code in the job for downloading the files described at points -1- and -2- and for unpacking the archives at point -2- prior to starting the computation.

- Job completion callback: this allows a custom command to be executed by Vnas when a certain set of slices completes execution (Vnas constantly monitors the submitted jobs). This can be used to trigger the following step in a computational pipeline and releases the pipeline programmer from needing to write such code.
- Automatic deletion of Grid uploaded files that have not been used recently. This prevents the waste of Grid storage resources due to users forgetting to remove files they no longer need.
- Grid bandwidth optimization: the Vnas-uploaded files are left on the Grid Storage Elements for a certain time frame before deletion: this prevents a needless reupload of the same files in case these are needed by the same or even by another Grid user within the time frame. This prevents a waste of Grid bandwidth.
- Sharing of files uploaded by different Grid users: files uploaded by Vnas for Virtual Sandboxes are never uploaded by Vnas more than once: if a second job submission, even by a different Grid user, needs a file that has already been uploaded its presence is detected on the Grid and the file is not uploaded again. This saves users' time and Grid storage space.
- The files uploaded to the Grid by Vnas are identified by their md5 hash. This allows Vnas to identify files with certainty based on their content and regardless of the name the users locally assigns to the files (which could be different for the same file for different Grid users). The md5 hash also prevents false positive matches that can arise due to random name equality among files or due to a file existing in different versions with the same name.

Sandbox does not create any new storage space on the Grid and the user is still responsible for the files uploaded by Vnas in this way. The user who has uploaded a file to the Grid is always traceable by the Grid administrators.

4.1.3 Vnas functioning

Vnas architecture (see figure 2 below) is decentralized, with the exception of a central database containing hashes of files submitted for Vnas virtual sandbox functionality. The central database should (but is not required to) be shared among a high number of Grid users to better take advantage of the sharing of files uploaded by different users. The Vnas architecture ensures that the per-user load on the central database is very low hence the system can still scale linearly up to a large number of users (in the thousands).

Vnas job submission is instead performed from standard Grid User Interface (UI) nodes. A local database located on the same host stores Vnas information about job submissions and users' callback requests.

Vnas has the following three main operating modes:

- Job submit (direct user invocation)
- Job run (triggered by the grid infrastructure)
- Set-callback request (direct user invocation)

4.1.3.1 Outline of job submit:

- The user creates a sample job_home directory containing all the files needed for the job to run, i.e. both executables and data, then s/he invokes Vnas.
- Vnas scans the job_home directory, packs files and subdirectories of job_home separating them in sub-archives and single files according to a customizable algorithm.
- Vnas then computes the md5 hashes of all such files and archived contents, and uploads them onto the grid with a filename generated from their hash (some might exist already: the upload is skipped). Then Vnas contacts the central database for inserting the entries regarding the newly uploaded files, or updating the “last access” timestamp for files already existing (hash matched). A small set of files up to a certain (user definable) amount of KB are packed into a physical (traditional) sandbox.

- Vnas then submits the job to the grid, and records the job information such as the grid job identifier on the local database.

4.1.3.2 Outline of job run:

- A “jobprepare” script (belonging to the Vnas distribution, and automatically inserted into the job submission by Vnas) is run first. The jobprepare scans some bundled data to find the configuration of the job_home to be recreated.
- Jobprepare downloads all the needed archives and single-files to recreate the content of the job_home directory tree. The downloading is performed in a write-through fashion: if the file’s nearest replica is still geographically distant, it gets replicated from there onto the Storage Element closest to the worker node, then gets downloaded on the worker node. The latter happens on a local network, and is basically immediate. This kind of replication ensures that subsequent downloads of the same files are faster and faster, and that the more a file is used, the higher the number of replicas. Space does not risk to get wasted for long, as when a file remains unused for a number of days, all replicas get deleted by a Vnas instance polling information from the central Vnas's database.
- Vnas invokes the user specified executable and waits for its termination
- Vnas packs the user-requested result files and uploads them onto a storage element, using a filename automatically generated at the time of job submission.

4.1.3.3 Outline of set-callback request:

- The user invokes Vnas specifying a set of job identifiers that s/he wishes to wait for, and the command to be invoked by Vnas when the condition arises. Vnas records such information into the local database.

4.1.3.4 Vnas polling loops

In addition to the three above described main working modes, two additional slow paced Vnas polling loops are needed in order to provide the virtual sandbox file expiration and the callback functionalities:

- A very slow paced Vnas loop on the central database node, which monitors the “last access” timestamps of files uploaded by UI Vnas instances, for virtual sandbox functionality. Vnas clears the Grid from all replicas of files which were last used too long ago. The exact number of days for expiration can be configured but values between between 7 and 15 days are typical. Longer times save Grid bandwidth at the expense of storage space and vice versa.
- A medium paced Vnas loop on each UI interface, which polls the local database and the Grid information system to fetch the state of the last jobs submitted by the users of the node. When all the jobs required for a certain callback request complete their execution, the user command is invoked with the credentials of the user who requested it (leveraging sudo -b). This is typically used to trigger a further step in a computational pipeline

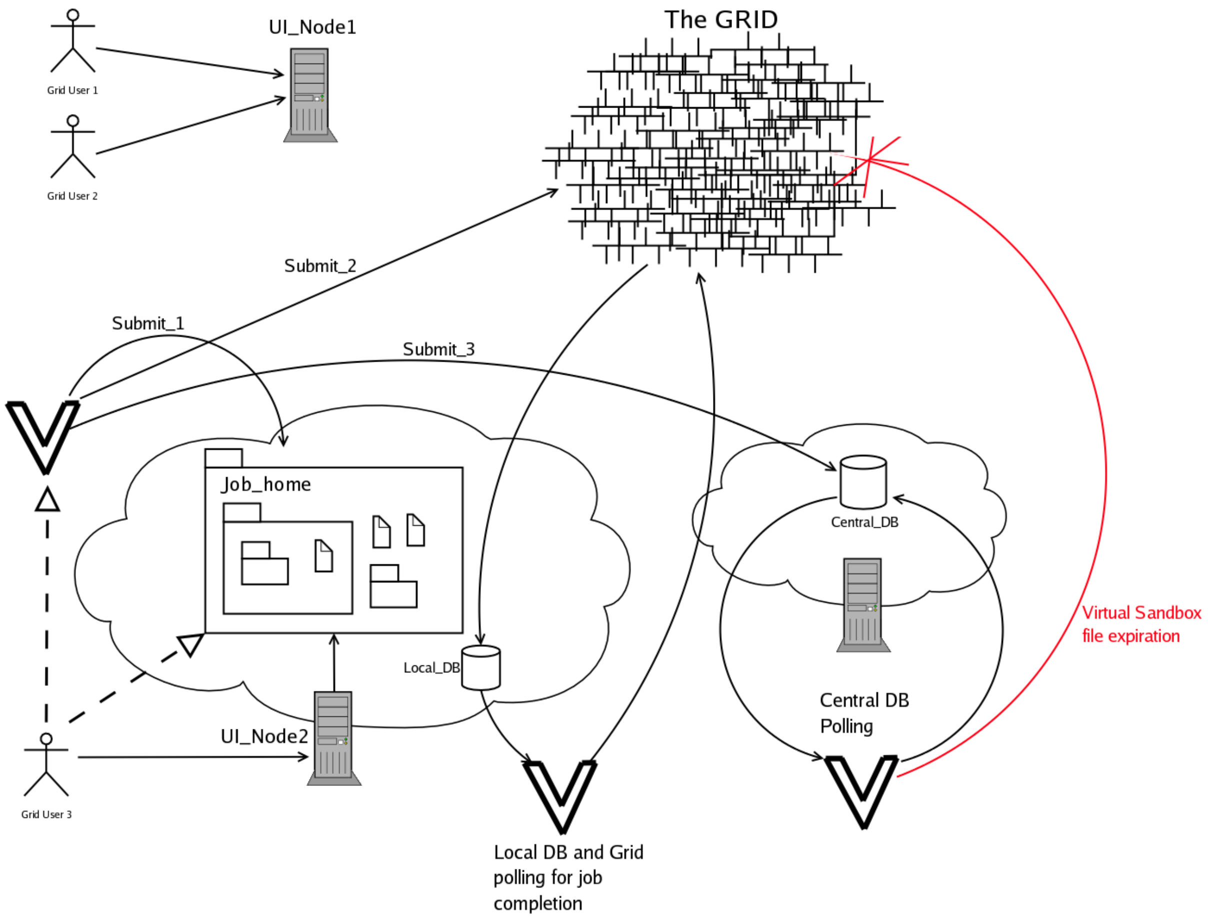


Illustration 2: Vnas distributed architechture - The figure shows, from right to left: A) A Vnas loop on the central database, detecting expired virtual sandbox files and removing them from the Grid. B) A Vnas loop on the local UI database and the Grid, polling for the status of the most recently submitted jobs. C) A job submission on UI2 by GridUser3, and more in detail: C1) User creates a job_home. C2) User invokes Vnas. C3) Vnas scans job_home (Submit_1), uploads the files to the grid (Submit_2) and creates the corresponding entries or updates the "last usage" timestamps for the uploaded files, on the Central Database (Submit_3).

4.1.4 Persisting obstacles in porting applications to the Grid with Vnas

One of the obstacles found when porting an application to the Grid which still persists notwithstanding the use of Vnas and that could deter a more widespread usage of the Grid is the burdensome splitting of input data to create multiple slices of a conceptually single job (even though for the implementation of the "wait all results" join the Vnas callback can be leveraged), and the consequential parsing of the results from multiple slices at the end, for recreating a single result set.

Unfortunately, it is not easy to conceive a tool which can help with this in the general case, since the way to perform splitting of data and merging back of results depends heavily on the type of data and the specific problem.

Of course, when comparing the Grid to a privately owned cluster, it has to be noticed that the problem would arise identically.

4.2 BGBlast and the Grid Database Manager

Blast [Altschul'90], [blast] is probably the most famous bioinformatics application. In my Ph.D. I continued an existing project by Merelli and Milanesi for evolving an existing porting of Blast on the Grid platform (GridBlast [Merelli'05]) adding features related to the data management that appear to be very innovative, unprecedented for a Grid application.

Here follows an introduction to the topic and discussion of existing solutions, then my approach and implementation will be described. Additional details can be found in the Appendix. This is partly taken and adapted from a publication of mine [Trombetti'07] with permission from the publisher.

4.2.1 Introduction

BLAST [Altschul'90], [blast] is a well known and widely used bioinformatics application for comparing (usually unknown) “query” biological sequences, either genomic or amino-acidic, against a set of known “reference” sequences (“Blast Reference Database” or BRD in these chapters). BLAST is a variation and approximation of the exhaustive dynamic-programming Smith-Waterman [Smith'81] algorithm for local sequence-alignment, resulting in a speed increase of 10-100x, at the expense of some sensitivity [alignmentscore].

While BLAST sensitivity is generally regarded as still adequate for most circumstances, the speed of BLAST can still be scarce for certain massive computations, which are in fact performed rather commonly by many bioinformatics research groups.

The problem of BLAST speed can be addressed in various ways, the solutions usually belonging to the following groups (a) faster alternatives to BLAST, (b) BLAST execution on clusters and (c) BLAST execution in Grid.

Pros and cons for (a) and (b) will be mentioned in the next section 4.2.2 - *Existing Solutions*. The solution I present here belongs to (c). As far as (c) is concerned, the main problems usually arising from BLAST execution in Grid are:

1. Defining and enforcing a policy for replication of the BRDs over the Grid. BRDs are large files needed during BLAST execution over the Grid. Due to their size they require allocation on Grid Storage Elements (SEs). Rising the number of replicas for a BRD reduces the Grid queue times for BGBlast runs using that BRD, but also rises the associated storage costs (see section 4.2.4.2.1 - *Motivation for ARM* for details). Due to their significant size, it is not reasonable to replicate every BRD on a large number of SEs. In this scenario, policies should be defined and enforced for allowing an optimal usage of Grid resources.
2. Keeping the replicated BRDs up-to-date.
3. Optionally it might be profitable to store older versions of the BRDs so that BLAST users can reproduce and verify results obtained in the past. The problem in providing this feature is that keeping older versions of BRDs available normally has a very high storage cost.

In section 4.2.4 - *Implementation* more details are given regarding the above issues and on how I was able to address them.

4.2.2 Existing solutions

4.2.2.1 Faster alternatives

Various alternatives to BLAST which are faster and similar in scope are available such as MegaBLAST [Zhang'00], [megablast], BLAT [Kent'02] and PatternHunter [Ma'02] . These alternatives usually are different enough to be not suitable for exactly the same situations as BLAST is, or sometimes can have different drawbacks. As far as the examples are concerned, MegaBLAST and BLAT, albeit much faster, have a lower sensitivity than BLAST. PatternHunter on the other hand claims a similar sensitivity but is a commercial closed source product, and the algorithm is not known exactly.

Such drawbacks might or might not be acceptable for the researcher, depending on the specific circumstances. In addition, researchers aiming at publishing their results might want to use specifically BLAST simply because its reliability is well established and cannot be object of discussion.

4.2.2.2 Cluster execution

Various solutions [Qi'05], [Mathog'03], [Darling'03] have been developed to parallelize the BLAST algorithm for execution on computing clusters and supercomputers. These solutions have been used for quite some time now and are regarded as reliable. The main drawback of cluster execution for BLAST is the initial cost for purchasing the dedicated cluster, which is high, and might be unreasonably high -relatively speaking- in case the cluster is not going to be used full-time (uneven workloads).

4.2.2.3 Grid execution

A number of implementations of BLAST for the Grid environments already exist [Bayer'04], [Konishi'03], [Merelli'05] but in general suffer from the problems already mentioned in section 4.2.1 - *Introduction*. I hereby present BGBlast, another Grid implementation for BLAST which I developed evolving the earlier Merelli–Milanesi's project GridBlast [Merelli'05]. In BGBlast I successfully addressed the issues mentioned in section 4.2.1 - *Introduction*.

4.2.3 BGBlast's approach

BGBlast (BioinfoGridBlast) has some unique advantages over the existing solutions. BGBlast is an innovative porting of BLAST onto the Grid providing the following capabilities (1) automatic update of the biological databases handled by BGBlast (2) adaptive replication of databases on the Storage Element Grid nodes (3) version regression for the biological databases. BGBlast is the evolution of the earlier project GridBlast [Merelli'05] on top of which I added the following features:

1. Automatic Database Updater (ADU): ensures the users always work with the latest version of every Blast Reference Database (BRD), and

this without needing human staff to manually monitor the release of newer versions of BRDs or manually performing database updates over the Grid.

2. Adaptive Replication (AR) for the BLAST Reference Databases: ensures that the most used BLAST databases are replicated more times than less used databases. The optimal number of replicas for each BRD is calculated dynamically based on the relative usage of such database in recent times. This keeps a constant optimization of Grid queue times vs Grid storage costs.
3. Version Regression for BLAST Reference Databases: allows the user of BGBlast to specify an older version of a certain BRD to be used for the computation. This allows the user to exactly reproduce computations obtained in the past, something that might be needed to confirm results that were obtained. The storage of older version of BRDs is performed with a delta-encoding efficient in both space (storage costs) and time (a short download time and a short time to patch a BRD for regressing it to an earlier version).

4.2.4 Implementation

4.2.4.1 GridBlast core

GridBlast [Merelli'05] is still the core for BGBlast, providing the following capabilities:

1. Factor J parallelization of large BLAST executions. This is done by splitting the user input into J even subset, each taking 1/J of the original time to execute. This is followed by the submission of J smaller BLAST jobs (1/J of query sequences against the target BRD) on the EGEE [EGEE] Grid platform. J is chosen so to create jobs of reasonable length, neither too small (Grid overhead would be comparatively large) nor too big (insufficient parallelization).
2. A rate limiting feature triggered on very large BLAST executions. This limits the rate at which the jobs are submitted to the Grid so to avoid a sudden massive Grid exploit.

3. Monitoring of the launched jobs and automatic resubmission in case of failure for any of those. This is still important nowdays, as the Grid platform is still new and reliability is not excellent.
4. Fetch of the results back after the completion of the Grid jobs. Merge of such results into a single BLAST results file.
5. A recent improvement of the core provides measurements of the queue times and cpu hours consumed by the J Grid jobs for each run of BGBlast. These measurements are passed to the Adaptive Replication Manager and are essential for the correct functioning of the AR functionality (see).

On top of the GridBlast core, I implemented the following functionalities:

4.2.4.2 Adaptive Replication Manager (ARM)

The Adaptive Replication of BLAST Reference Databases is a BGBlast feature for optimizing the number of replicas for each BLAST database dynamically and adaptively.

4.2.4.2.1 Motivation for ARM

BLAST Reference Databases (BRDs) are large files, usually in the range 500MB-5GB, and are needed during the run of BLAST on the Grid CEs for each of the J BGBlast-generated jobs. Due to their size, it is not reasonable to download a BRD from a remote location. It is hence necessary to constrain the J jobs to execute on CEs having a replica of the user-requested BRD on a near (local network) SE.

Due to this constraint, the number of CEs to choose from for the BGBlast generated Grid jobs is limited. This impacts the queue times negatively and this is particularly true if the replicas of the requested BRD are few. A massive replication of every BRD on all the SEs of the Grid is not feasible either, because of their size which would make the storage costs unbearable.

Clearly, it is more useful to have additional replicas for BRDs used often, so that the queue times are reasonably small for the most common BGBlast runs, while it is better to have fewer or possibly only one replica for the

BRDs used less frequently, and this is in order to reduce the Grid storage costs.

Since the amount of usage of for each of the BRDs cannot be known in advance, I have implemented a dynamic, adaptive replication mechanism to balance between queue times and storage costs.

4.2.4.2.2 Implementation for ARM

The ARM performs a D days moving average (usually D=10) of the cpu hours and queue times used for each reference database. This statistical measurement is used to compute the optimal number of replicas for each of the BLAST reference databases. This algorithm balances between the additional storage costs incurred in increasing the number of replicas and the benefit of the reduced queue times.

Additionally when evaluating the addition of a replica the ARM engine also evaluates which of ths SEs would be the most advantageous for a replica addition. Similarly, when evaluating the benefit of removing a replica the ARM engine also evaluates the least advantageous of the currently existing replicas, that is, best for removal. See the Appendix for details.

Initially I also thought about featuring a hysteresis for the variation on the number of replicas but I haven't yet implemented that as the D days moving average seems to already provide an acceptable behaviour. Also it was not clear to me what hysteresis-like algorithm would have been optimal in this situation (this is again a bandwidth vs performance balance) and in particular what algorithm would have allowed a reduction to exactly one replica for the least used BRDs. I might anyway implement this feature in the future.

The measurements of used cpu-hours and queue times experienced for each BGBlast run, and implicitly for each BLAST database, are provided to the ARM by the GridBlast core (see). The dynamic variation of the number of replicas is evaluated, and possibly performed, at each BGBlast run and at the end of each day.

Additional algorithm and implementation details regarding the ARM can be found in the Appendix section 7.1 – *BGBlast's Grid Database Manager algorithm details*.

4.2.4.3 Automatic Database Updater (ADU)

BGBlast's ADU engine constantly monitors FTP sites for newer versions of the BRDs registered to be handled by BGBlast. If a newer version of a BRD is detected, the ADU automatically updates all the replicas of such BRD over the Grid. This is not the only action performed by the ADU: the ADU also computes an xdelta (xdelta3 is used in the last version of the framework) patch for regressing the newer version of the BRD to the earlier version of the BRD now being replaced, and uploads such xdelta patch on a predefined SE. The xdelta patch computed by the ADU, together with the xdelta patches computed during previous database updates, is needed for the DVR functionality (see).

Such xdelta3 patches are at least one order of magnitude smaller than any version of the BRD they refer to, and this makes the storage costs reasonable. In order to further reduce the storage costs, I decided to keep only one replica for the xdelta patches. Also see the next section on this topic.

4.2.4.4 Database Version Regression (DVR)

BGBlast provides an option for specifying a version (in terms of date) of the BRD to be used for the BLAST computation, along with the name of the BRD. The requested version of the BRD is obtained from the latest version of the BRD by applying the ADU-generated xdelta patches in sequence, from the newest to the oldest. Each xdelta patch regresses the BRD by one version, and this action is performed until the requested version is reached.

The version regression operation is performed on the Computing Element (CE) after the download of the BRD from the near (local network) SE and prior of starting the computation.

The download of the xdelta patches is generally remote, as the patches are only replicated once on the Grid (see section 4.2.4.3 - *Automatic Database Updater (ADU)*), and this is in contrast with the download of the BRD (latest version) which is over a local network (see section 4.2.4.2.2 – *Implementation for ARM*). However, due to the small size of the patches, the patches' download time rarely surpasses that of the BRD (latest version). Since the DVR is also a relatively uncommon request by users, I considered the patches download time an acceptable overhead.

Additional implementation details for BGBlast can be found in the Appendix section 7.1 – *BGBlast's Grid Database Manager algorithm details*.

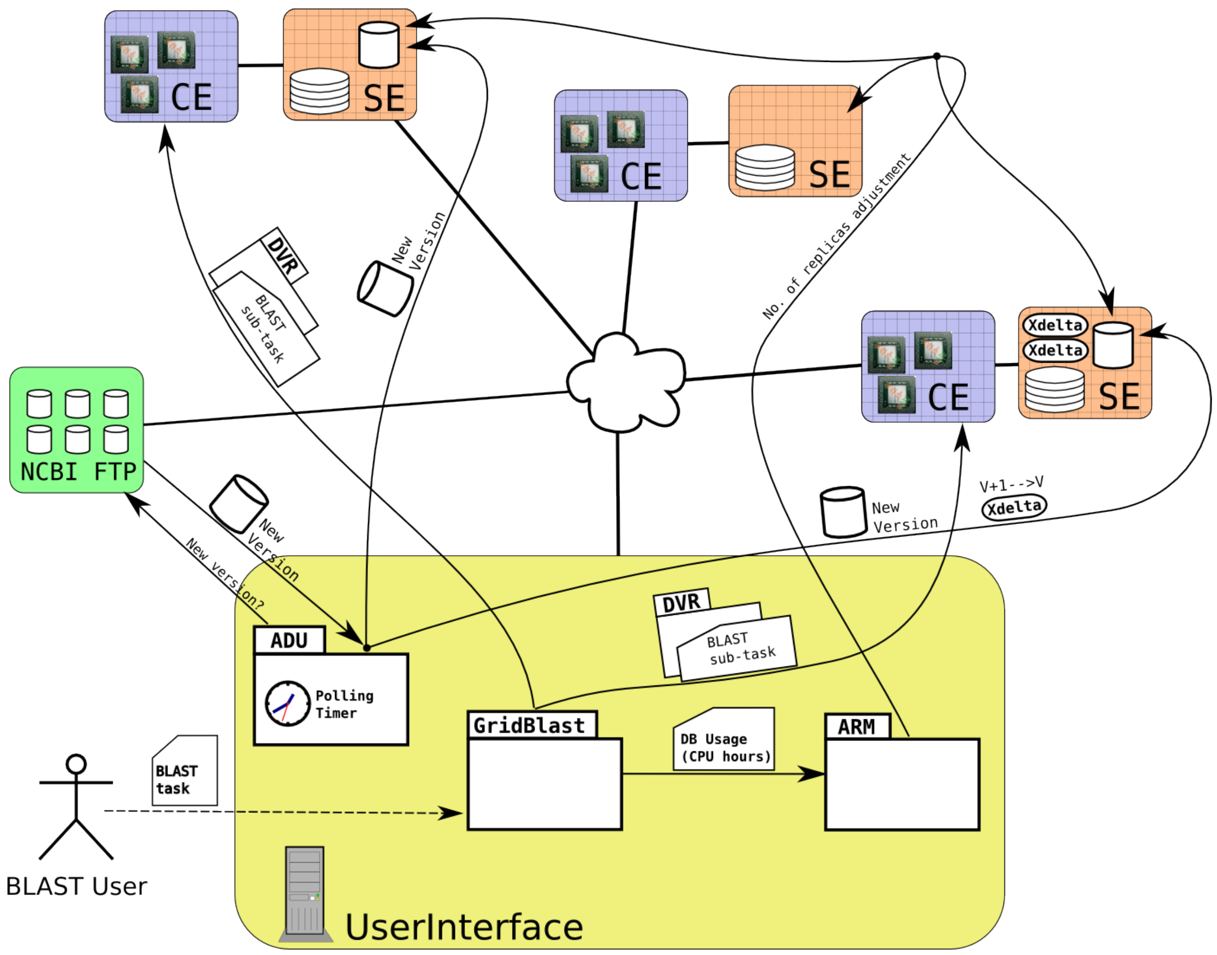


Illustration 3: BGBlast architecture - ADU polls for new version of handled databases from the FTP site of origin, downloads a new version then updates the version on the Storage Elements. ADU also uploads the corresponding Xdelta file for allowing version regression (not replicated). The GridBlast core processes Blast user requests and sends Blast sub-tasks to the Grid for computation, together with DVR agent code. ARM updates the number of replicas for the handled database based on usage information from GridBlast.

4.3 SETest testing framework

In addition to the very grid-specific frameworks described above, I also developed a Python testing framework for testing Python applications scattered with side effects. This framework proved very beneficial for accelerating the development of Grid computational applications such as those for the bioinformatics domain.

4.3.1 Motivation – Side effects

Most bioinformatics applications developed for the Grid are actually computational pipelines created joining together a number of existing programs (standalone executables) developed by leading bioinformatics institutes. The specific computation to be performed is achieved through the exact choice of these numerous standard pieces, their exact joining, the parameters for launching every executable, and the parsers, the converters and filters between every executable and the next.

Being every step of a pipeline a standalone executable (and in most cases with more than one input and output) their inputs and outputs are in most cases stored in files. The inputs have to be prepared by the pipeline and feeded to the executable, then the executable is run and the pipeline's logic again is used to parse the results reorganize the information for the next executable.

Operations on disk are called "side effects" in programming terms, since they are not completely contained inside an application's runtime memory dump. Other side effects are network communications and operations on external databases.

All these three types of side effects are very present in Grid computational pipelines, expecially in the bioinformatics domain. In my experience, in a bioinformatics Grid pipeline on average one line of code every *three* has got a side effect!

This fact makes testing bioinformatics pipelines really challenging.

It is a widespread belief that testing of applications should be performed through unit-testing and test-cases (see [UNITTESTING], [TESTCASE], [XP], [DIPTESTING], [TESTDRIVEN]).

However, unit tests and test cases cannot be used in regions of code containing side effects. The usual common practice is to encapsulate side effects in a small number of classes, and then test the rest of the application using unit testing and test cases. However, this is feasible when the application is big and mostly self-contained, and side effects are relatively few, while the technique would be at the very least highly unpractical to do when side effects are scattered every few lines of code and when side effects are actually the *purpose* of the programming logic (as is in our cases: the purpose of the pipeline is preparing correct side effects for use by external executables, and correctly parse side effects after the executable has run).

In this circumstance, a novel testing technique had to be developed, and this is what I did within the course of my Ph.D. with the SETest framework.

This framework could not be programmed in a statically linked language without introspection capabilities such as C++, while other dynamic languages where not optimal for other reasons written in the Motivation-Python subsection below.

4.3.2 Motivation – Python

Python is a modern object oriented fully dynamic and strict-typed language, fully supporting advanced exception handling, RAII and Guards programming techniques. And it is the ONLY language having all these features together. The reliable exception handling with deterministic destruction available in Python (and in C++ but not in other modern languages such as Java and Ruby) allows to create resilient program flows within complex algorithms with relative ease and very readable coding. The deterministic destruction (stack rollback) available in Python upon exceptions allows automatic cleanup to be performed, which allows the programmer to use the same RAII and Guards programming paradigms which were once available only for C++ programmers. These characteristics were of extreme help when developing programs for such an unreliable platform as the Grid, where every single command has a significant probability of failing and lots of cleanup actions are to be performed in these cases.

For these reasons Python was my language of choice for developing Grid frameworks and bioinformatics applications during the course of my Ph.D. The dynamic nature and introspection capabilities of Python also allowed to create this testing framework, which wouldn't have been possible with a compiled language without introspection capabilities such as C++.

4.3.3 Main features

4.3.3.1 Wrapping function calls

The SETest framework allows testing most Python applications heavily scattered with side-effects, such as most Grid computational pipelines, in particular bioinformatics ones.

The language of the application to be tested is mandatorily Python: Python introspection was used heavily for the internals of this framework.

The framework can be used to wrap almost any function call having side effects. Once function calls are wrapped, the framework is able to record the exact function call and its return values and replay the call in the future without executing it again.

The wrapping syntax looks like this:

This function call

```
results = obj.fun(param1,"foo",param5=6.51)
```

becomes

```
st = SETest.SETest() #Only once per file, after import
results = st.do(obj.fun, param1,"foo",param5=6.51) #Actual wrapping line
```

So as you can see, wrapping a function almost does not increment the number of lines (just one more per file) and the programmer only needs to type about 5 more carachters for each wrapped call. In addition, library function calls which *might* provoke side-effects (such as the Python `os.system` call and process opening) can be wrapped directly in the libraries.

4.3.3.2 The four operating modes

The framework has four operating modes:

- Bypass
- Recording
- Playback
- Playback with recording fallback

The framework can be set in any of these modes through a configuration file unique to each wrapped application.

In **bypass** mode, the call is executed exactly like if the SETest framework was not present. The overhead is almost zero.

In **recording** mode, the call is executed then results are stored into a *"dataset"* file together with the exact call which has generated them. With exact call I mean that also the value of parameters for the function call are recorded in the file, and if these are objects, their state is serialized. Results are also serialized if the type is complex. Finally, results are returned to the caller as it would have happened with an unwrapped call.

In **playback** mode, the call is *not* performed. An equal call (including parameters equality) is searched into the *dataset* file. If an equal call is found, the results are returned to the caller. If an equal call is not found we have two possibilities, depending on how SETest was configured: either a `DatasetLookupFailure` exception is raised (kind of "testing has failed" paradigm), or the programmer is dumped to a Python Interactive Shell ("let's investigate why calls are different this time" paradigm). From the interactive shell the programmer can do a series of things:

- inspect the call, the value of all parameters, the stack trace (including all variables in scope in every function of the call chain) and the dataset
- can perform the call manually
- can insert some results in the dataset, either hand-made or those coming from the manual execution of the call
- can raise a `DatasetLookupFailure` exception, or can return to the caller some results (again either hand-made or those coming from the manual execution of the call)

In **playback with recording fallback** mode, the framework behaves like in playback mode, except when the dataset lookup fails, in which case the call is transparently executed and results are stored into the dataset (like in recording mode) without returning any error.

The apparently innocent-looking "playback with recording fallback" mode is probably the most powerful for our scopes. This mode allows to greatly speed-up the initial testing of newly created computationally intensive pipelines in Python, relieving from Python almost the only disadvantage it has: slow initial testing due to the lack of a compiler.

The "playback with recording fallback" mode in fact allows to restart the execution of a computationally complex pipeline from the beginning, but skipping all computationally complex steps which have already been executed the first time (the steps are skipped because they are usually external applications and hence are wrapped, but even if they were internal side-effect-free function calls, it is still possible to also wrap these). This means e.g. that all syntax errors, undeclared variables or any other kind of interpreter-detected error the programmer might have written, which in python cause a stop of the execution, now do not anymore cause the re-execution of the computationally complex steps upon restart. This in turn means that the restarted execution will immediately get again to the point where last error was, so the programmer almost has the illusion that s/he can fix errors on the fly and continue execution sequentially (which is nothing short of a *dream* in Python expecially during the first executions, in which usually newly-created code filled with trivial errors needs to be continuously restarted, and wait times are longer and longer once the first errors are fixed and those in line 250something start to emerge).

4.3.3.3 Regression testing

Other than significantly accelerating initial debug times for newly created Python applications completely scattered with side-effects, this framework also allows an easy regression testing [REGRESSIONTESTING] of such applications, which would have been extremely difficult otherwise. A regression testing is defined as testing again something that has been modified, and it was known to work properly before modifications.

The way to proceed in this case is the following: with the stable version of the application, prior to modifications, the programmer sets the SETest framework in Recording mode and launches the application multiple times with multiple inputs, possibly covering all execution flows and critical types of input, producing in this way a number of dataset files with known-good function call invocations and results.

After performing the modifications to the application, the programmer sets the SETest framework to Playback mode (*without* recording fallback in this case!) and launches the application again using the dataset files recorded previously.

If the behaviour of the application is still equivalent to before, the application should execute very quickly (most computationally heavy steps are probably skipped) and exit without errors returning correct results to the user. Instead, if the behaviour of the application is not equivalent to before, some function call to an external component will be different than before, hence will cause a `DatasetLookupFailure` exception (or dump to the interactive shell, depending on the configuration). Syntax errors and other language errors are of course also detected in this way, and for all possible execution flows, providing that the programmer had filled enough datasets to test all execution flows.

This framework is hence very effective in providing a means for performing regression testing for applications heavily scattered with side effects.

4.3.3.4 Opaque object states

In the presence of functions or executables which store an opaque internal state, or use a state from some system component (such as the system clock --> `time` / `date` functions), or have an implicit random input or internal random number generation (such as a random number generator --> `random()` and similar library calls), which we will call opaque functions or opaque objects, one is actually forced to also wrap the calls to those functions treating them in the same way as if they were side-effect functions.

This is because in case the random or systemstate-related value is used as a parameter in a subsequent call, during a *second* execution of the pipeline the said subsequent call wouldn't match anymore with the corresponding call recorded in the dataset file during the *first* execution of the pipeline. The call would be detected as different because the input parameter would be different (random each time).

Not only, but if the random number generator function is called multiple times in the pipeline there is another problem: if more than one call to the same random function is wrapped (thing which is usually *needed*, as said in the above paragraph), as the random generator function takes zero parameters, all calls are detected to be equal by SETest. This means that during *Recording* modes all the calls would overwrite the same entry in the dataset, while in *Playback* modes all calls would return the same value. The same problem exists with most functions or executables which store an opaque internal state, or use a state from some system component (such as the system clock --> `time` / `date` functions), or have an implicit random input or internal random number generation (such as a random number generator --> `random()` and similar library calls), which we will call here "*opaque objects*".

For this purpose, the framework also has a `state_advance()` function, which can be called when a new separate results set should be used (stored/retrieved to/from the same dataset file but on a separate section of it). This is needed to wrap all calls to *opaque objects* which return a result which is different for multiple calls even if the input parameters or data are the same.

The SETest `state_advance()` function should then be called just before or just after such calls to the *opaque object* and the effect is practically that of moving to a separate partition of the *dataset* file, for each time `state_advance()` is called.

This surprisingly simple trick allows the SETest framework to be used in cases in which the real state of the pipeline is partially *opaque* and cannot be detected by the framework, such as in presence of the state of a random number generator, and it actually works so reliably that I cannot consider it a completely inelegant solution. The partitions created by `state_advance()` are however completely separated from one another, without any kind of smart logic behind. This means that if two truly identical calls are performed one before and one after the `state_advance()` call, these too will be treated as different. This is a lack of optimization but it should not be a semantic problem as far as I can see, for the way SETest is meant to be used.

The `state_advance()` can also be called with a `-n` parameter as in `state_advance(-n)` and this has the effect of regressing the state by `n`, going back to an older section of the dataset. As I said, this was probably not needed, semantically speaking, however being able to regress the state can be a helpful optimization when the code exits from a zone filled with random generators and similar opaque objects, returning to a region where all calls and external object invocations are truly described by their input parameters.

Additional implementation details about this frameworks can be found in the Appendix section 7.2 – *SETest framework implementation details*.

4.4 Resource Brokers Round-Robin (RBRR)

The most critical element of the Grid is probably the Resource Broker, which has the task of routing newly submitted jobs to the optimal CE based on a series of constraints, heuristic evaluations and a load balancing algorithm.

The RB is also the Grid element which the User Interface contacts when it wants information on the status of jobs. In particular, the UI automatically contacts the same RB that was used to submit a job.

As I mentioned in section 3.2.2.4 – *Downtimes*, RB failures are unfortunately common and this causes a series of issues. The uptime I experienced is around 95% for the resource brokers of the Biomed VO. During downtimes it is not possible to submit any job to the Grid, and in addition, it is not possible to retrieve the status of submitted jobs or the results of completed ones. The RB downtimes are hence serious failures.

During my Ph.D. course the problem was appearing very evident to us so I implemented a simple solution in the User Interface at CNR-ITB: resource brokers round-robin. I replaced glite_job_submit with a small proxy application that first delegates the call to the real glite_job_submit and then atomically switches the gLite configuration files so that a new RB will be chosen for the next job submission. Every job is now submitted to a different RB, looping through the list of available resource brokers for our VO.

This simple solution solved both mentioned problems and two additional ones:

- The downtime of a resource broker doesn't prevent us from sending jobs. Please note that while single submission failures are still possible (one RB down), the single submission failure is immediately handled by VNAS, which is programmed for resilience and retries failed Grid commands multiple (configurable, usually 3) times. The second submission attempt will be over a new resource broker, and we have yet to see a failed submission using 3 attempts (3 brokers).

- The downtime of a resource broker only prevents us fetching a small percent of job statuses and results (jobs that were launched with an RB that has now become unavailable), and we treat that small number of jobs through the upper level frameworks (VNAS) in the same way as Grid job failures: resubmitting the job
- We evenly distribute our workloads among the many RB of the Biomed VO so we don't overload a single RB which can be used by other people
- Load balancing of CEs jobs appears more effective now, that is, a long sequence of submitted jobs is spread across the available CEs more evenly. I am unsure of why this happens, it might be because different brokers update their BDII statistics at different times, so they have different rankings, or it might be because some configuration parameters affecting the ranking might have been set differently in the various RBs by the different system administrators.

5 Impact of solutions & case studies

In this chapter I will try to analyze the impact of the four solutions I developed in my Ph.D. work and which were described in detail in chapter 4.

Firstly, the most quantifiable enhancements produced by the solutions will be presented: in subsection 5.1 the impact of the solutions in increasing the Grid performances will be examined, and following that, in 5.2 an attempt will be made at evaluating the impact of the solutions in reducing the time and effort needed for the Grid user to create and launch computational applications in the Grid environment.

Then, other benefits which could not easily be quantified will be presented in subsection 5.3.

Lastly, in 5.4 a case study will be presented showing how Vnas was used in practice for supporting a multi-stage computational pipeline in the Grid environment.

5.1 Increase in the Grid performances

This section studies the increase in performances in the Grid environment which can be achieved through the usage of Vnas and the Grid Database Manager (from BGBlast package).

Vnas's impact on performances is due to the advanced management of Grid jobs (Vnas also causes bandwidth and storage space optimization not covered in this section). Vnas performs a frequent polling of the status of jobs and can resubmit them upon failure or when exceeding a configured maximum queue-time threshold (jobs stuck in the queue as described in 3.2.2.1 – *Job failures*), reliably and without delays.

GridDBManager on the other hand reduces the execution time of jobs by providing a replica locally to where the job is executing (i.e. within a LAN), and hence the job setup time (i.e. the time to be spent downloading files prior to computation) is greatly reduced. The GridDBManager is considered to be in fully running state as a simplifying assumption, i.e. all the required databases are already considered fully replicated for the purposes of the following sections.

The approach used to quantitatively obtain and then show these results is the following: in chapter 5.1.3 – *Grid Performance simulation results* you can find the Grid performance graphs computed with the Grid Performance Simulator (GridPerfSim), simulating the applications' performances in Grid under various Grid conditions, size of the overall task and submission algorithms, both for the manual case (without tools) and for the case of a user using Vnas and GridDBManager. The GridPerfSim simulation is based on the probed Grid performance data reported in chapter 5.1.2 – *Experimental Data*. Exact methods to obtain these data and the graphs are precisely explained in the next subsection. Each of the graphs implies millions of simulated jobs in the various conditions, and could not realistically be obtained without the help of a simulator.

5.1.1 Methods

For measuring the Grid platform's intrinsic performances

1 I sent probes to evaluate the job queue times during normal Grid load and during artificially elevated load (250-jobs artificial load).

2 I sent probes to evaluate the Grid download speed from a SE to a WN, both in case of local download (downloading from the closeSE, that should be LAN local network) and remote download. I computed log-normal μ and σ statistical distribution parameters.

For obtaining the performance graphs

1 I built a monte carlo simulator (GridPerfSim: see below) for quickly simulating the performances of massive Grid submissions following certain submission algorithms.

2 I selected a few representative use-cases for the Grid (“quick-response”, “small-challenge”, “big-challenge”), and coded the algorithm for those use cases in the simulator

3 I coded the algorithms for handling the various submission parameters that were different in the case of manual submission

versus the case of using the tools Vnas and GridDBManager

4 I plotted the graphs for all combinations of use cases and tools usage

5.1.1.1 GridPerfSim simulator algorithm

GridPerfSim is a valuable tool to examine the Grid performances through simulation, once certain experimental data are available.

GridPerfSim simulates a Grid submission of an arbitrary size (number of jobs, duration of each job) with a certain submission algorithm. GridPerfSim simulates this submission many times, computes average and standard deviation values for all the results obtained, and finally traces precise graphs showing these results.

For each job it has to launch, GridPerfSim takes the Grid queue time from one of my probes (experimental data). For each job then the time for downloading the database is simulated using the log-normal distribution with the parameters μ and σ that I probed from the Grid, or can be a fixed time.

The submission algorithm which can be set, includes the following parameters:

- task_size: size of submission (number of jobs)
- duration of each job
- size of database and download speed statistical parameters, or a fixed job-setup time (the job-setup time on the Worker Node is identified with the database download time in this thesis)
- rate_limiter: number of parallel jobs that can be launched. If this parameter is set to e.g. 100, after launching 100 jobs the simulator will wait for one job to complete before launching the next job.
- queue_time_ceiling: if the job stays in the queue for too long, GridPerfSim can kill the job (like a edg-job-cancel) after such preset threshold time has expired, and launch another job. This is a common practice on the Grid, because some jobs sometimes happen to stay stuck for extremely long times in broken queues.
- polling period: GridPerfSim is able to simulate the case in which the

user does not poll the status of the job immediately at the job's completion, but only after some time. The polling period can hence be configured and in our graphs will be different for the hand-made case and for the case with Vnas automated polling.

Two types of graph can be traced with GridPerfSim: the completion-time graph and the speedup graph. The first shows the total time to complete for a task, and the second shows what is the overall speedup that was obtained, that is, (completion time with 1CPU)/(completion time in Grid).

In this chapter I report the results of my experimental measurements for the Grid performances.

I measured both Grid queue times and failure rates, and the local (closeSE) and across-the-Grid download speeds using probes.

The experimental data obtained, reported in the next section, were used in the GridPerfSim simulator to compute the performance and scalability graphs which I will show in the subsequent sections.

5.1.2 Experimental Data

5.1.2.1 Grid queue times and reliability

Of first and utmost importance for evaluating the Grid's performances in various cases is to measure the Grid queue times and the reliability of the nodes during execution. I sent hundreds of probe jobs on the grid for this evaluation.

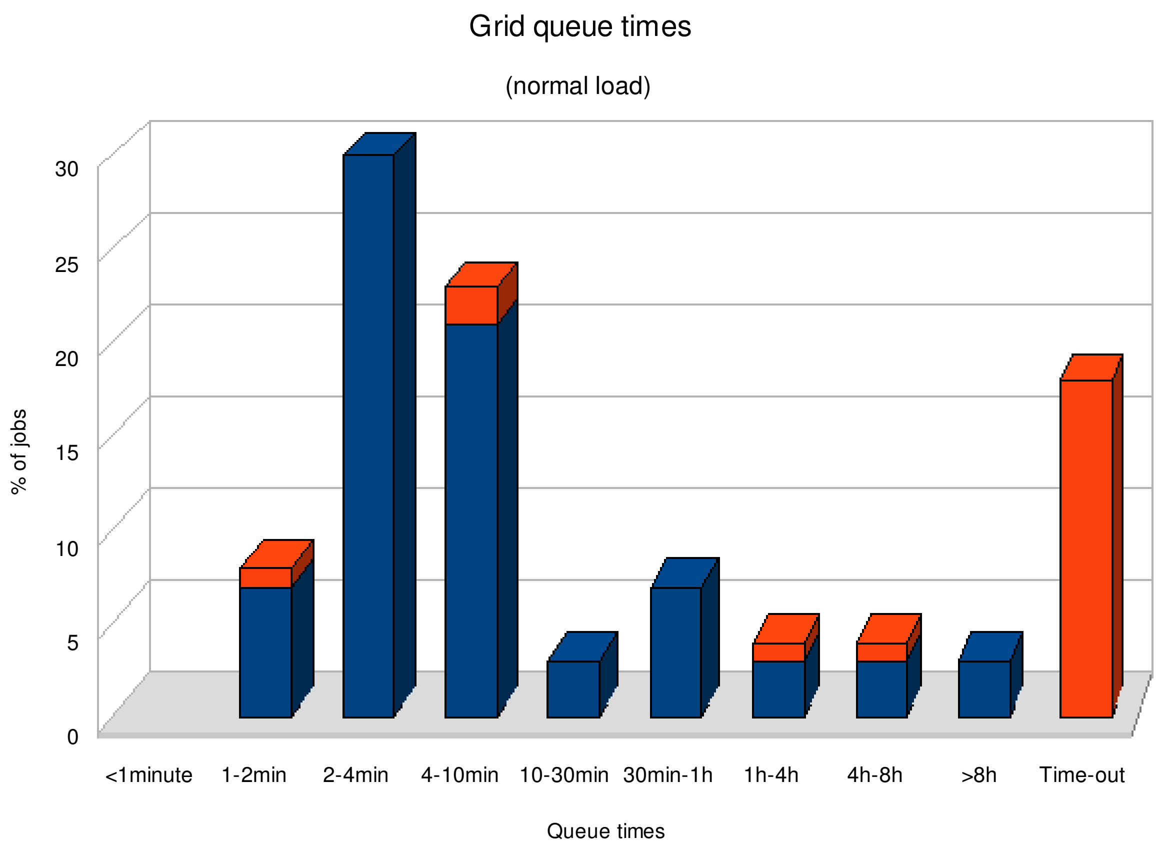
The probes were sent both in a situation of normal Grid load, and in a situation of artificially elevated Grid load. The results are shown in the following subsections.

The graphs represent aggregate (hence approximate) data to facilitate reading, however, the simulator uses the probed experimental data directly.

5.1.2.1.1 Normal Grid Load measurements

I sent three distinct groups of 50 probe jobs in separate moments for evaluating queue times and nodes reliability during execution. I waited for the jobs of one group to complete or time-out before sending another group of jobs in order not to artificially elevate the Grid load during this measurement. The jobs had 1-hour duration, enough to prevent immediate completion of all jobs by a single worker node.

The results are summed up in the following figure



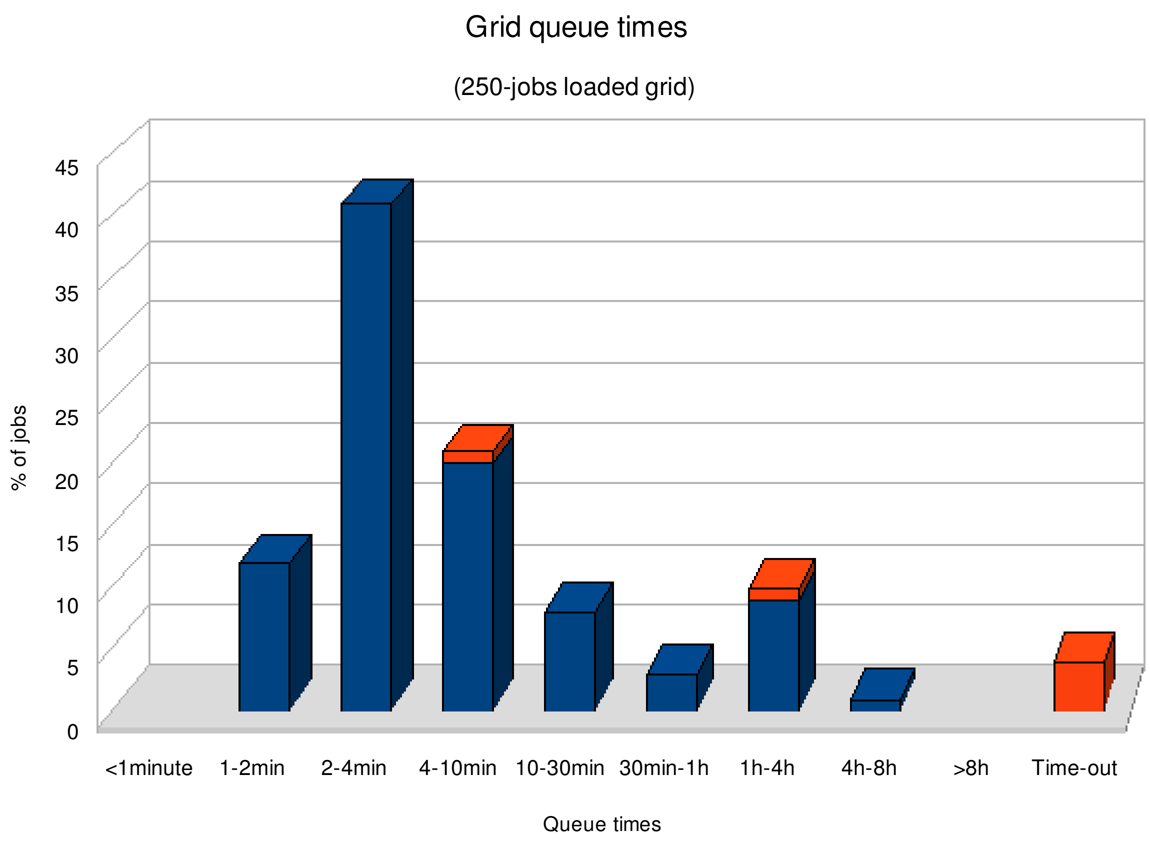
The red part of the bars represents the jobs that initiated execution but then failed, or in the last column those jobs which never reached the execution phase.

In the simulator, the jobs that initiated execution and then failed are resubmitted at the moment of the failure of the probe, or at the end of the simulated execution if this is set to be shorter. The jobs that never reached the computation phase will be resubmitted when the preset maximum-queue-time threshold is exceeded.

5.1.2.1.2 Elevated Grid Load measurements

For this measurement I locked 200 jobs on Grid nodes for 10 hours, then I sent 100 8-hours long probe jobs. In this way the average Grid load seen by the 100 probe jobs is about 250 (first probe launched sees 200-jobs load, second sees 201... last probe sees 299 jobs).

The results are the following



As it can be seen the outcome was even better than with the probes sent without artificial load.

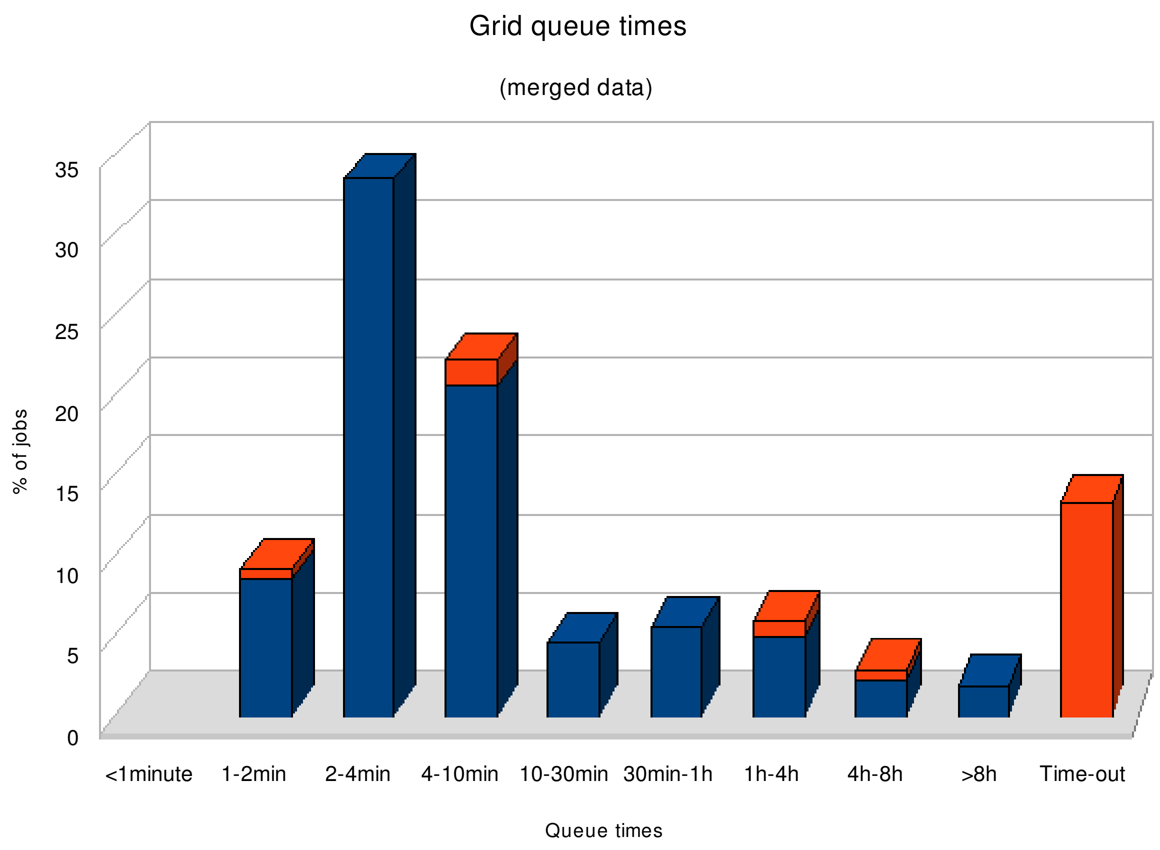
Presumably this is because the variability of the EGEE Grid load across different days is much higher than 250 jobs. Apart from this, the trend of the two graphs is similar.

5.1.2.1.3 Using the probed data

In my simulations I assume that the user does not want to launch more than 100-500 jobs in parallel (100 or 500 depending on the specific use case and submission algorithm, as doing this could be unrespectful of the other Grid users. In addition, I did not want to artificially load the Grid with e.g. 1000 jobs or more for many hours, in order to avoid a significant waste of Grid resources.

On the other hand, since a load of 250 jobs cannot be distinguished on the Grid (it is overwhelmed by the variability of the EGEE Grid, as it seems) I found more sensible to use all the four sets of probed data merged together. In this way I gathered a set of 250 probe jobs (50+50+50+100 probes) obtained at four different times, and this makes a fairly good dataset for our simulator.

This is the graph for the merged data:



5.1.2.2 Grid download speed

The Grid download speed from worker nodes is also important when simulating the performances of database-oriented bioinformatics applications on the Grid, because the database sizes can be significant and this would affect the “setup-time” of the application on a worker node, i.e. the time prior to starting the computation.

I sent 200 probes over the grid for estimating the download speed from various storage elements to the worker nodes.

The download speed I experienced varied significantly and I found it to be best expressed with a log-normally distributed download speed measured in bytes/sec (1MB = 1024^2 bytes) with the following μ and σ parameters:

distance	μ	σ	Average download speed
near (closeSE)	16.143	0.607	11.755 MB/sec
distant (a remote SE)	13.932	0.582	1.269 MB/sec

Hence the download from a near location practically never causes relevant delays, but the download from a distant location of a database larger than 1GB might incur in a significant delay and bandwidth usage and should be avoided, in particular if the job is going to run for relatively short time after the download.

In the next section I report the simulated performance graphs for:

- 1.5GB of remote download with stochastic download speed (case of the manual Grid submission). Average expected download time: 20m 10s
- 1.5GB of local download (from the closeSE) with stochastic download speed (case of the Grid submission using GridDBManager). Average expected download time: 2m 10s

5.1.3 Grid Performance simulation results

5.1.3.1 Introduction

In this chapter I report the GridPerfSim's Grid performance graphs computed using the probed performance data described in section 5.1.2 – *Experimental Data* and different submission parameters and algorithms depending on whether the submission was "manual" or through Vnas and GridDBManager. This was done as described in section 5.1.1 – *Methods*.

The whole simulation to obtain the following graphs with good precision implied about 10 million simulated grid job launches. This was clearly not feasible without a simulator.

Two types of graphs will be presented in this chapter: the completion-time graph and the speedup graph.

- The completion time graph shows the overall completion time for a certain submission size
- The speedup graph shows (completion time with 1 CPU) / (total grid completion time) for a certain submission type

Mean values are marked by dots and standard deviations by vertical segments. The submission size (number of jobs launched) is the X axis for both graphs.

Three submission modes (algorithms) are simulated:

1. Fast response: use case for a user who quickly wants a small number of results from the grid.
   - the overall number of launched jobs is small (1 to 100). More precisely, the following submission sizes (number of jobs) are simulated: 1, 5, 10, 20, 30, 40, 50, 75, 100.
   - the rate_limiter is not set
   - the duration of the computation for the jobs is short (1 hour)
   - the polling time for checking the status of jobs and resubmitting failed jobs will be 3 minutes in case of Vnas (**BLUE COLOR**), while it will be half an hour in case of manual grid submission (**RED COLOR**)
   - the resubmission algorithm towards long queue times (queue_time_ceiling) is aggressive for Vnas case (queue_time_ceiling timeout set at 20 minutes, **BLUE COLOR**) while it is up to the next polling moment (i.e. every 30 minutes) for the manual case (**RED COLOR**)
2. Small challenge: use case for a small sized challenge submitted to the Grid.

- the overall number of jobs launched is medium (50 to 10000). More precisely, the following submission sizes are simulated: 50, 225, 500, 1000, 2500, 5000, 10000.
- the rate_limiter is set at maximum 500 jobs running simultaneously, so not to impact the Grid too much for the duration of the submission, and be fair with other grid users.
- the duration of the computation for each jobs is medium (6 hours)
- the polling time for checking the status of jobs and resubmitting failed jobs will be 3 minutes in case of Vnas (**BLUE COLOR**), while it will be half an day (12 hours) in case of manual grid submission (**RED COLOR**)
- the resubmission algorithm towards long queue times (queue_time_ceiling) is 2 hours for the Vnas case (**BLUE COLOR**) while it is up to the next polling moment (i.e. every 12 hours) for the manual case (**RED COLOR**)

3. Big challenge: use case for a relatively big sized challenge submitted to the Grid.
   - The overall number of jobs launched is large (1000 to 100000). More precisely, the following submission sizes are simulated: 1000, 2500, 5000, 10000, 50000, 100000.
   - the rate_limiter is set to maximum 100 jobs running simultaneously, so to impact the Grid as little as possible for the duration of the submission. The rate limiter is set to a value smaller than for the “Small challenge” in order to be fair with the other Grid users, considering that the “Big challenge” is expected to run on the Grid for a very long time.
   - the duration of the jobs is long (10 hours)
   - the polling time for checking the status of jobs and resubmitting failed jobs will be 3 minutes in case of Vnas (**BLUE COLOR**), while it will be one day (24 hours) in case of manual grid submission (**RED COLOR**)
   - the resubmission algorithm towards long queue times (queue_time_ceiling) is 6 hours for the Vnas case (**BLUE COLOR**)

while it is up to the next polling moment (i.e. every 24 hours) for the manual case (**RED COLOR**)

Download times are also stochastically simulated with GridPerfSim. The size of the database to be dowloaded for the computation is considered to be 1.5GB in all use cases. The download will be considered local (within the LAN) for the GridDBManager case (**BLUE COLOR**) while it will be remote for the manual Grid submission (**RED COLOR**)

The simulated performance graphs follow:

5.1.3.2 Fast Response submission type

I will recall here the parameters used for the simulation of the Fast-response submission (details in section 5.1.3.1 – *Introduction*):

Meanining of colors	Manual Grid submission Submission through Vnas and GridDBManager
Submission sizes	1, 5, 10, 20, 30, 40, 50, 75 and 100 jobs
Rate limiter	disabled
Queue_time_ceiling	30 minutes 20 minutes
Polling period	30 minutes 3 minutes
Duration of the computation	1 hour
Line colors to setup time	1.5GB download from remote location 1.5GB download time close location

5.1.3.2.1 Completion time:

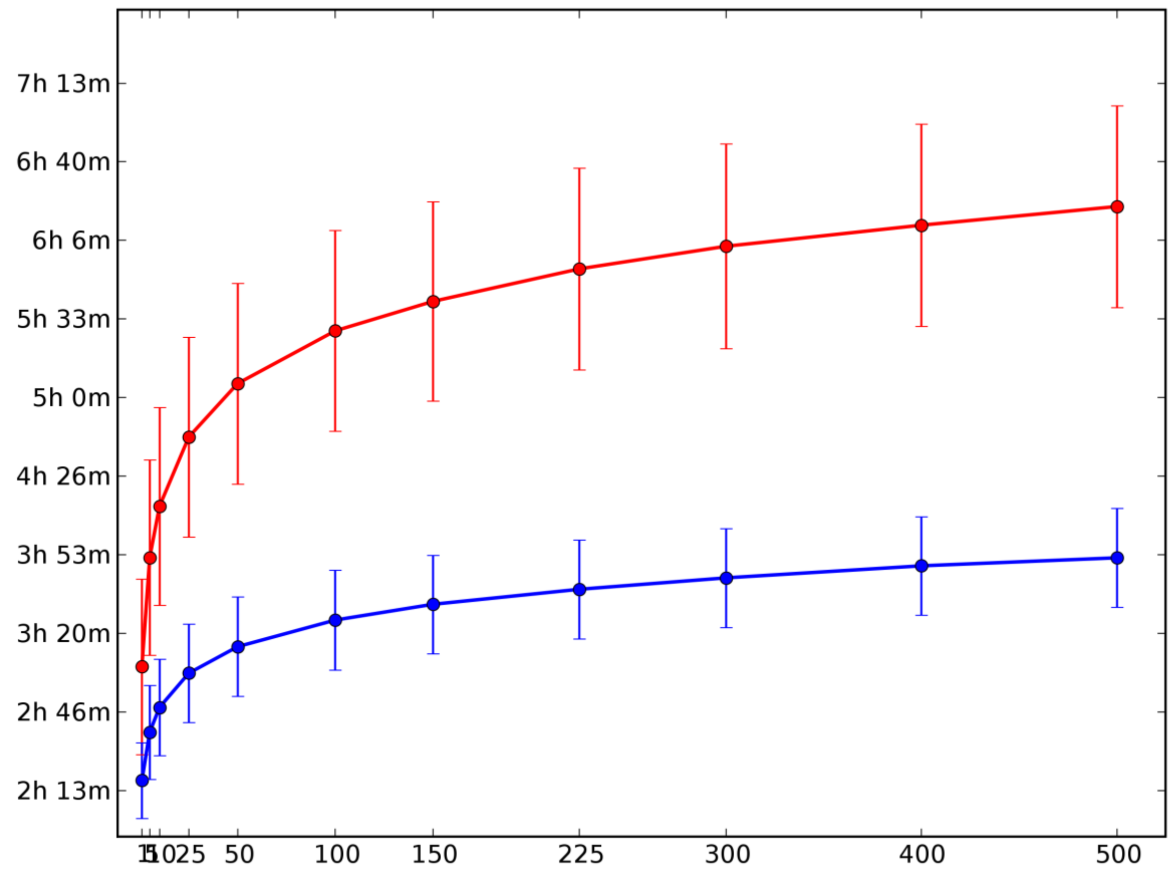


Illustration 4: Fast-response completion time

The completion time graph has an asymptotic trend when the number of jobs is increased. Keep in mind that the jobs are all submitted simultaneously at the beginning of the simulation since there is no rate_limiting. The reason for which the completion time increases when the number of jobs is raised is because with more jobs there is a greater likelihood of some unlucky jobs which happen to go in bad queues and stay stuck until they time-out (time-out set at 20 minutes or 30 minutes as described above) and eventually need to be resubmitted, maybe more than once.

Notwithstanding the aggressive resubmission timeout in the Vnas case (blue line), the standard deviation for the completion time is still significant in this type of submission. Anyway please note that the y-axis does not start from zero in this graph, so standard deviation bars might give the impression to be longer than they actually are.

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) is significant in this graph, showing the blue line providing an e.g. ~40% decreased completion time in the case of 500 jobs batch. The standard deviation is also reduced.

5.1.3.2.2 Speedup

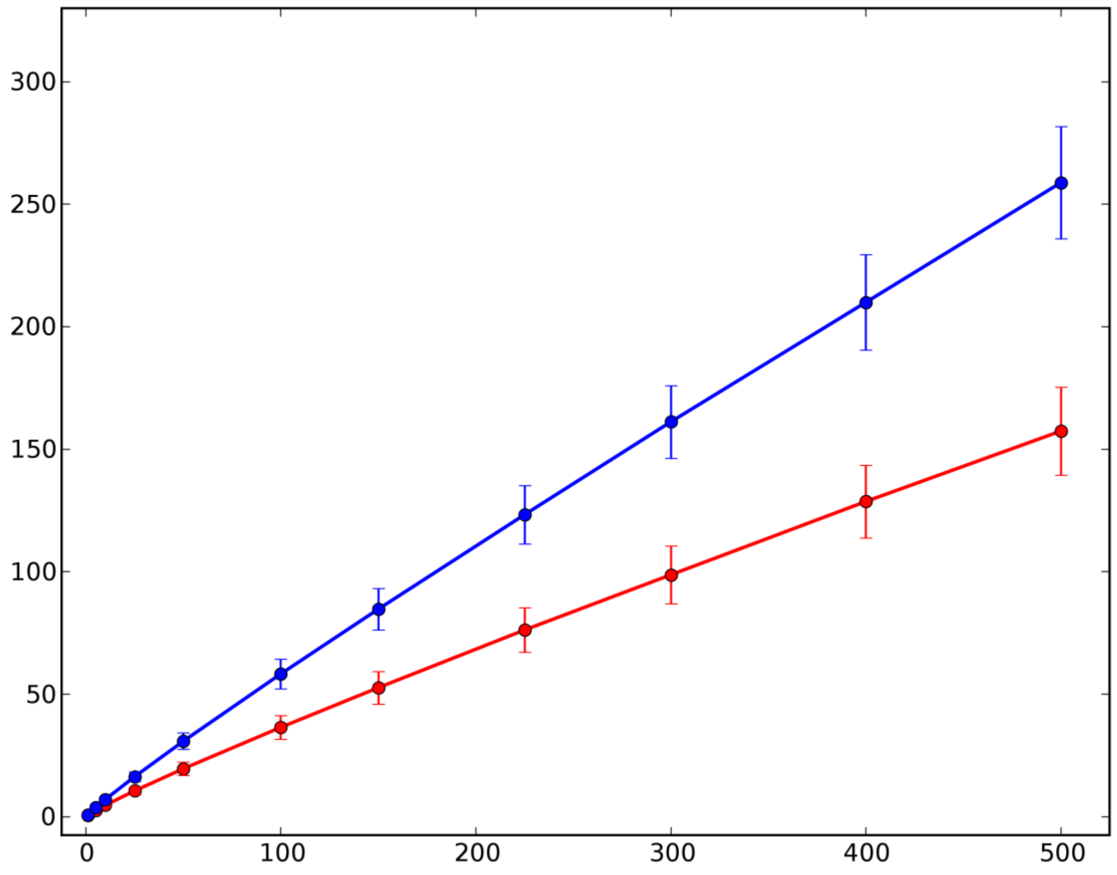


Illustration 5: Fast-response speedup

The speedup is roughly linear to the number of jobs since all jobs are submitted simultaneously at the beginning of the simulation (rate_limiter disabled). More submitted jobs mean higher parallelism in this scenario.

Also here it can be seen that the standard deviation is significant, though not extreme.

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) can be seen clearly also in this speedup graph, and it is mantained to a constant ~60% higher than the speedup obtained with manual execution, regardless of the number of jobs.

5.1.3.3 Small Challenge submission type

I will recall here the parameters used for the simulation of the Small-challenge submission (details in section 5.1.3.1 – *Introduction*):

Meanining of colors	Manual Grid submissiond Submission through Vnas and GridDBManager
Submission sizes	50, 225, 500, 1000, 2500, 5000 and 10000 jobs
Rate limiter	500 simultaneous jobs
Queue_time_ceiling	12 hours 2 hours
Polling period	12 hours 3 minutes
Duration of the computation	6 hours
Line colors to setup time	1.5GB download from remote location 1.5GB download time close location

5.1.3.3.1 Completion time

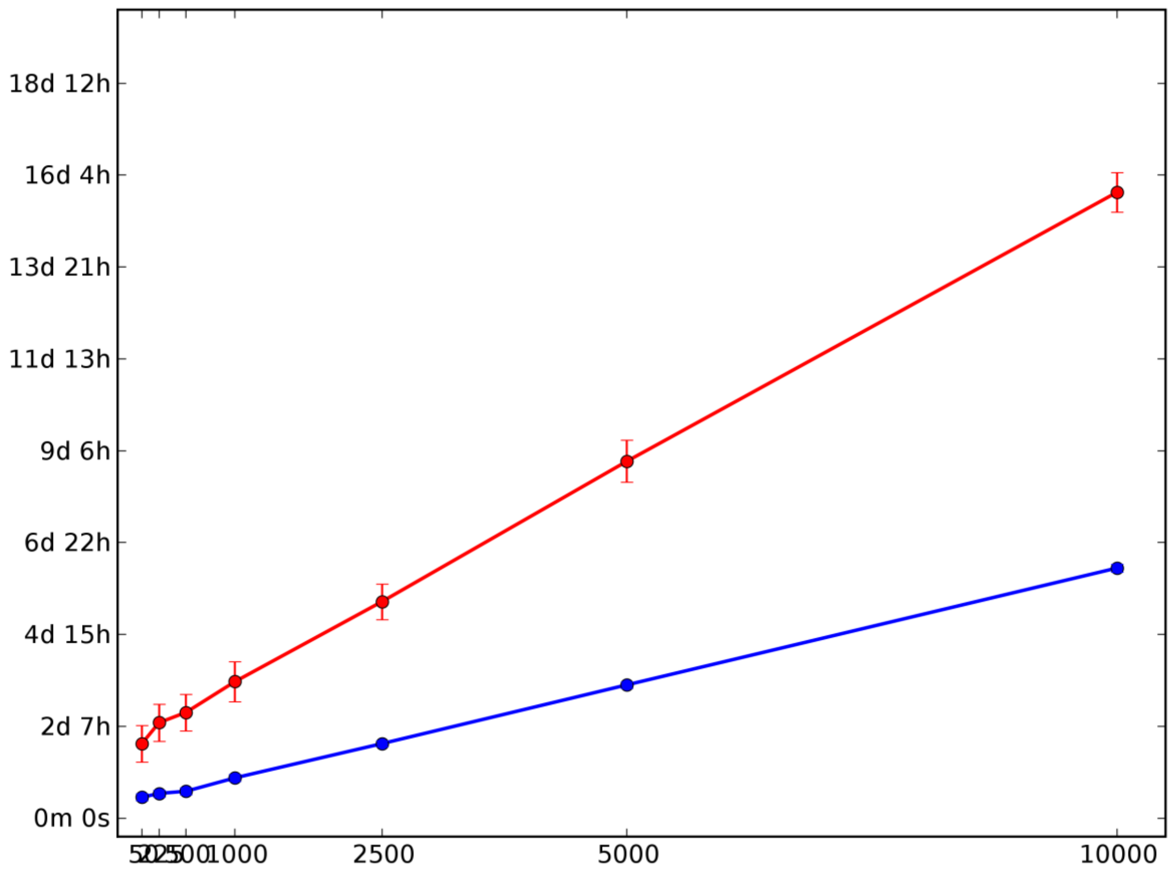


Illustration 6: Small-challenge completion time

Keep in mind that the rate_limiter is set at 500 for this type of submission.

Before the rate limiter is in action, as it is for the first three submission sizes (50, 225 and 500), the trend is asymptotic, like for the fast-response submission type. For larger submissions (1000 to 10000) where the rate_limiter is actively working, the trend is linear. The standard deviation is small for these rate-limited submissions, and almost non-existent in the Vnas submission (blue line).

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) appears even more evident in this small-challenge use case, showing the blue line providing a 60% to 70% decreased completion time across all batch sizes and reduced standard deviation.

5.1.3.3.2 Speedup

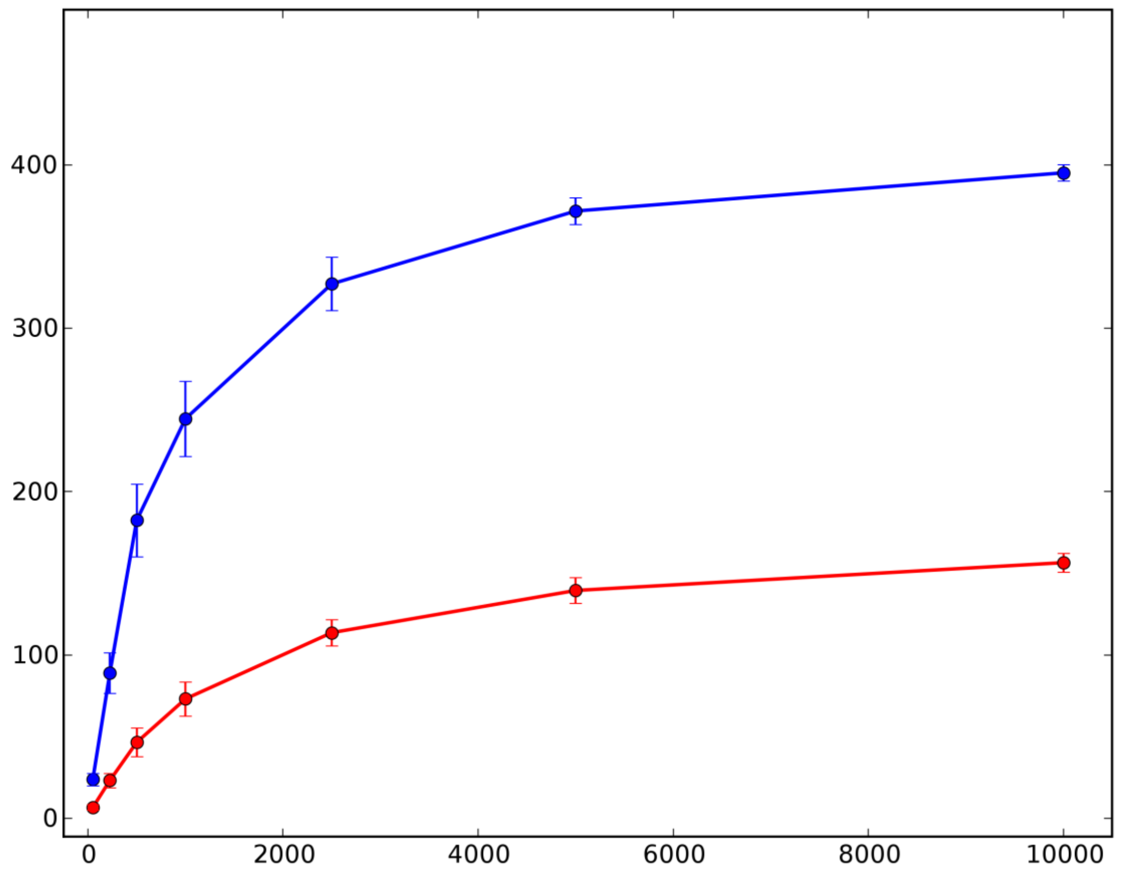


Illustration 7: Small-challenge speedup

For the first three submission sizes, where the rate_limiter is not effective, the trend is linear as it was for the fast-response submission type. For higher values, for which the rate_limiter is effective, the trend is asymptotic and, when using Vnas and GridDBManager, not too distant from 500.

Here the standard deviations are more visible than in the completion-time graph before because the scale is more favourable. Also here it can be noted that the rate limiter reduces the standard deviation significantly, though not completely, of course at the expense of not allowing the speedup increase beyond the value of the rate_limiter (500).

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) is very evident also from the speedup graph.

5.1.3.4 Big Challenge submission type

I will recall here the parameters used for the simulation of the Big-challenge submission (details in section 5.1.3.1 – *Introduction*):

Meanining of colors	Manual Grid submissiond Submission through Vnas and GridDBManager
Submission sizes	1000, 2500, 5000, 10000, 50000 and 100000 jobs
Rate limiter	100 simultaneous jobs
Queue_time_ceiling	24 hours 6 hours
Polling period	24 hours 3 minutes
Duration of the computation	10 hours
Line colors to setup time	1.5GB download from remote location 1.5GB download time close location

5.1.3.4.1 Completion time

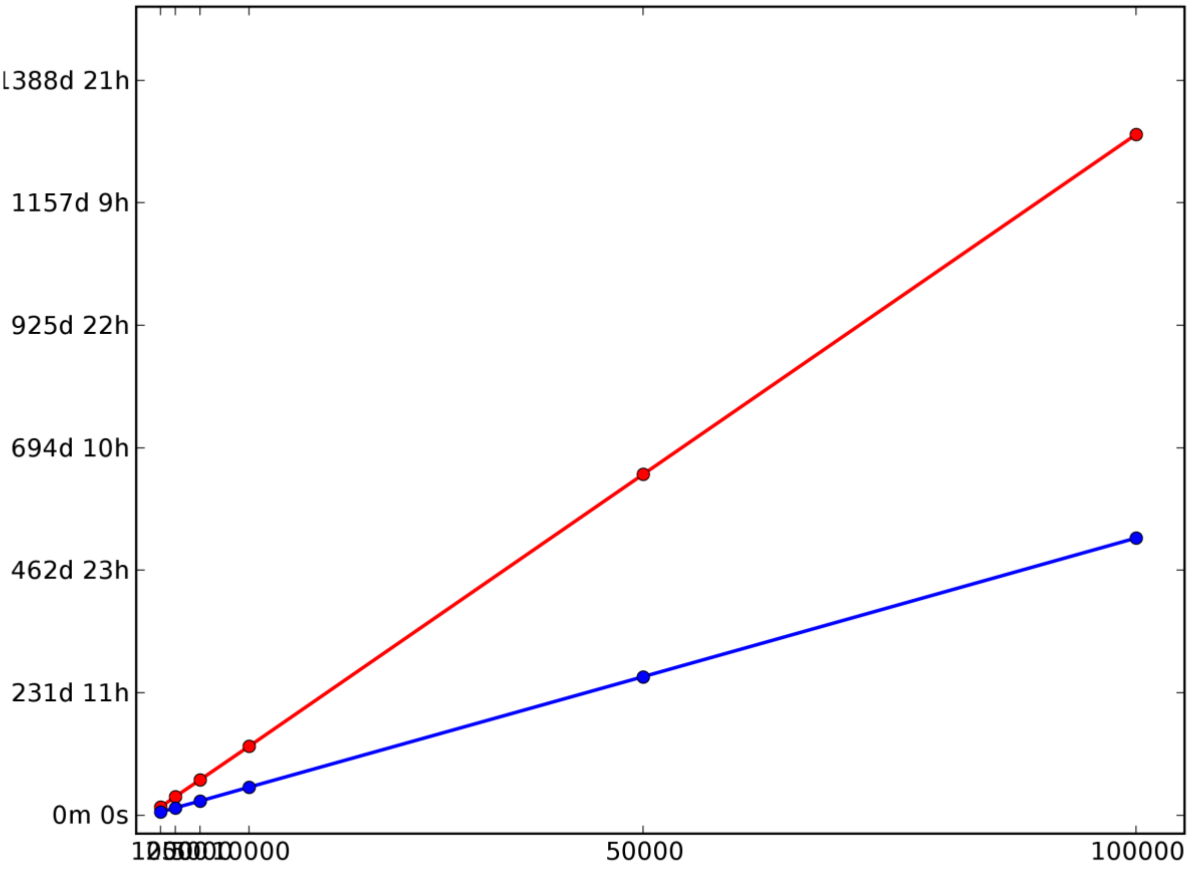


Illustration 8: Big-challenge completion time

Having the rate limiter set at 100 and the submission sizes starting from 1000, the rate limiter is always effective and we can see a perfectly linear graph for the completion time.

The relative standard deviations are almost zero for this submission type and this shows the almost complete predictability of the Grid for big rate-limited submissions such as large scale challenges.

Clearly, having the rate limiter set as low as 100 jobs prevents the speedup (shown in the next subsection) to raise above 100, and this makes such a big challenge take a very significant amount of time to complete, as is shown in the graph. I don't assert that 100 is the right value for such a large sized challenge: 100 is certainly on the safe side and should pass unnoticed, however, submissions of this size generally imply agreements with the members and the Responsible Person for the Virtual Organization the user belongs to, who will recommend the amount of Grid resources that can be exploited simultaneously, and this might be well over 100. The trend for a rate limiter set at 500 can be seen from the Small-challenge graphs in section 5.1.3.3 – *Small challenge submission type*.

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) in this big-challenge use case ranges from 45% to 50% reduced completion time across the various batch sizes. In this use case the standard deviation was not significant even for the manual submission.

It is also to be said that, contrary to the other use cases, nobody would really face such a big challenge without an automated tool for performing the submission and resubmission of jobs upon failure such as Vnas. Hence, in this and the following speedup graph, the red line is very theoretical.

5.1.3.4.2 Speedup

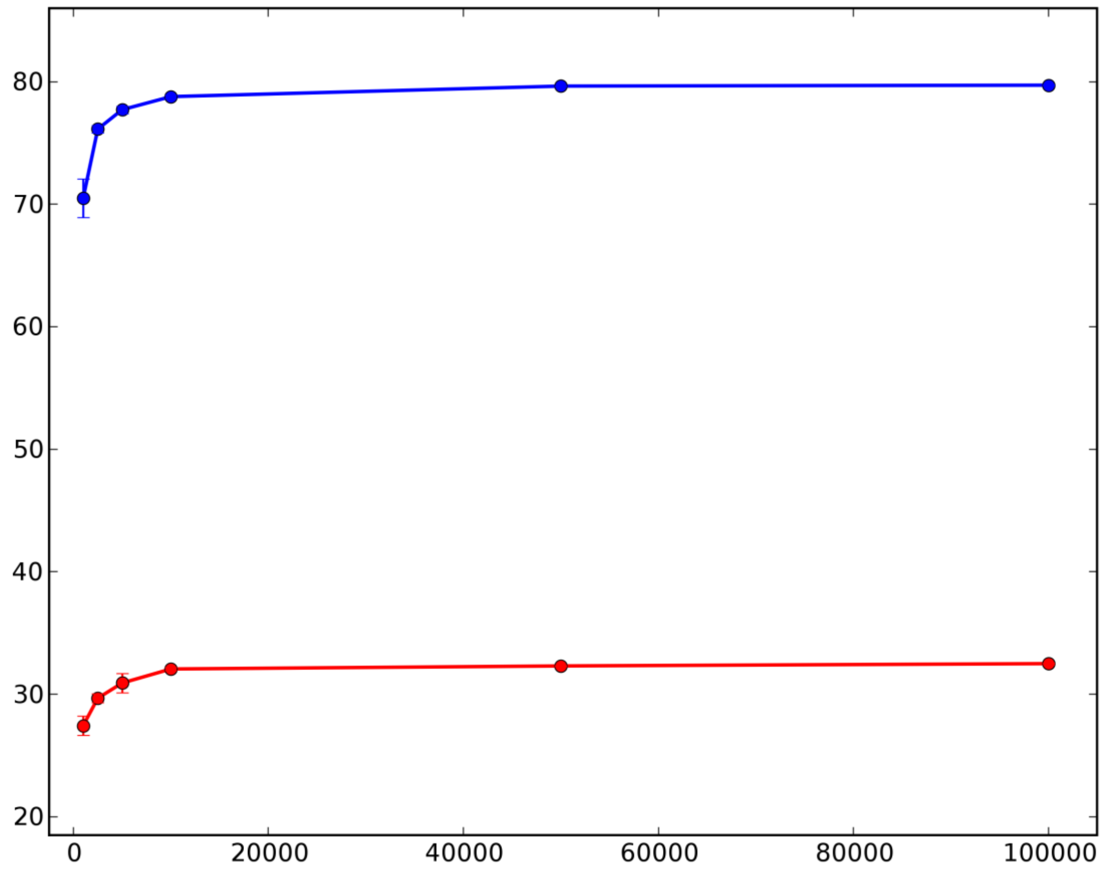


Illustration 9: Big-challenge speedup

As it was for small-challenge the trend is asymptotic but starts earlier and the asymptote is lower because the rate limiter is set lower than for small-challenge.

Standard deviation is almost non-existent for large rate-limited submissions. These kinds of submissions are very predictable on the Grid.

The benefit of the Vnas and GridDBManager tools (blue line) versus the manual submission (red line) appears very evident in this graph and ranges from factor 2 to 2.5 over the manual speedup across the various batch sizes. However, as said for the completion time graph, the red line for this use case is very theoretical.

5.2 Reduction of time and effort

In addition to the increments in the Grid performances as described in the previous section 5.1, the presented solutions can provide additional benefits for the Grid user in terms of reduction of the time and effort needed to create and launch computational applications in the Grid environment.

A precise evaluation of these additional benefits is particularly difficult as it depends on a number of factors such as: how the bioinformaticians exactly work, what types of "challenges" (computational pipelines / applications launched to the Grid as multiple parallel small jobs) they submit to the Grid, what is the submission algorithm being used for this challenge, whether they have or want to create automated procedures similar to those in Vnas or instead they prefer to handle the Grid completely manually, and so on.

Keeping this in mind, I will anyway attempt a rough estimate at quantifying the majority of the benefits. A few more unquantifiable benefits will then be presented in the section 5.3.

5.2.1 Reduction due to Vnas, GridDBManager and RBRR

In this section the reduction of time and effort due to Vnas, GridDBManager and the RBRR will be evaluated. The next section will attempt at evaluating the impact from SETest.

Please note that when evaluating the number of lines of code which need to be written due to the absence of the tools, I will refer to the number of lines of code for a "reasonably stable" approach, not for a "barely workable" one. For a barely workable one the number of lines of code might be half of that, but then significantly more human time would have to be accounted for bringing challenges to completion.

1. Vnas virtual sandbox management reduces the burden to create software packages with the applications needed to run the pipeline on the Worker Nodes **(20 minutes)** and upload these to a Storage Element **(20 minutes)**. In addition most computations need an amount of data which cannot fit within the normal 10MB sandbox and also changes from job to job within the same challenge. Hence, such data cannot be uploaded manually and has to really be packed and uploaded from within the submission loop **(6 lines of code)**. Then the SRM names resulting from the upload is to be somehow inserted into the job's configuration and read by the job **(4 lines of code)**
2. The SRM names from the point above also have to be logged to a file (2 lines of code) so to allow manual deletion of files from the Storage Elements at the end of the challenge **(10 minutes)** (Often forgotten, though.)
3. Vnas job submission relieves the need to programmatically create JDL files **(8 lines of code)**
4. Vnas's worker-node agent relieves the need to create code in the job for downloading and unpacking all needed files from Storage Elements, prior to starting the computation. **(20 lines of code)** (this includes some resilient code and some error reporting code to ease the very complex debugging in the Grid environment, already present in Vnas's worker-node agent)
5. Vnas job submission together with Resource Broker Round Robin (RBRR) relieves the need to check for returnvalues from glite-job-submit, and in case of error stop submission loop (the submission would fail again if attempted again on the same RB) reporting the error message from edg-job-submit to the user **(3 lines of code)**. In case of error, the code should also clear up temporary files and other side effects created on the UI which were needed for the job submission that failed, and bring everything to a consistent state **(10**

lines of code). Also the code should be checkpointable so to allow resume of the submission after human intervention, without needing to relaunch already launched jobs **(20 lines of code)**. None of this is needed with Vnas + RBRR: we have never seen glite-job-submit fail or hang anymore using 3 automatic attempts by Vnas (note: Vnas also has timeout implementation for Grid operations!) on rotating RBs.

6. RB problems happen in 5% of days. In these days the user should either wait, supposing s/he can work on other things, or call the administrator to change the RB configuration settings for the UI. The first choice is the most common in my experience, but let's account for the second which appears the most reasonable. Considering the debugging time to first ensure that the error is due to the RB, the time for contacting the system administrator and waiting the fix, we will account for 30 minutes of work by the personnel. This problem cannot happen when using Vnas + RBRR. **(30 man-minutes lost per working month)**.
7. Vnas job management relieves the need to manually poll for the status of jobs and manually resubmit in case of failure, or fetching results in case of completion. **(20 minutes twice per day until the challenge is completed)** It also relieves the need to keep the "source" files for all jobs sent, so to be able to resubmit single jobs if needed **(6 lines of code)**.
8. Vnas's callback system allows the user to be notified at the completion of the challenge. In case of a multi-stage pipeline, it will automatically submit the next stage. (Time included in the manual polling time above. Vnas solution is more comfortable, though.)
9. GridDBManager relieves the need to manually update biological databases in the Grid environment, including burdensome download, preparation of the databases (for BLAST databases, database formatting for BLAST databases is automatic with GridDBManager), and upload. **(3 man-hours / Month)**

Summing up the numbers and averaging them over the working days, and considering 20 working days /month:

77 lines of code + 50 minutes, once + 50m:30sec per day

These estimates are *per-challenge*, except for an average of 1m:30sec per working day coming from point 6 which is per-UI (there is usually one UI for each bioinformatics team).

Depending on how big the challenge is, this can be a very significant impact or not. Most bioinformatics computational pipelines are relatively small in terms of lines of code: some small computational pipeline can be made of just 30 lines of effective computational code (glue code joining external applications). Also, often bioinformatics computational pipelines are very short-lived in terms of maintenance: they run for a few days to a few weeks so to produce results, and then are replaced with something completely different.

It is then clear that on a such small and short-lived pipeline, the above estimates have a very strong impact, with the tools saving 72% of the lines of code plus 50 man-minutes per day. It should also be noted that the saved lines of code are also probably among the most difficult to write, since these are lines of code interfacing with the Grid environment, and have to be written as resilient as possible and capable to handle a wide variety of errors.

On the other hand, if the pipeline is much larger than that, the impact of the effort needed to interface it to the Grid would be proportionally much smaller. In addition, bioinformaticians which often submit to the Grid at a certain point will probably create code to automate the most common procedures, hence investing some development effort once for all in order to get a reduced impact from the Grid overhead code in the long run. But *exactly how much* time and effort a bioinformatician is willing to invest for reducing the Grid overhead code in the long run, is difficult to estimate. This is one of the reasons for which it is difficult to estimate the exact impact of the solutions presented.

5.2.2 Reduction due to SETest

The reduction of time and effort due to the usage of SETest is more difficult to quantify than the one coming from the other tools. This is because the usage of SETest does not cause a reduction in the number of lines of code of an application, nor a reduction of other measurable and demonstrable entities.

I will hence attempt this analysis with a different approach. I will simulate a debugging session of completely written but never tested Python code for execution of bioinformatics computations in the Grid environment.

In my experience, Grid bioinformatics pipelines contain between 30 and 200 lines of code (supposing the developer is using Vnas), 1/3 of which (10 to 66) contain side effects (though not all are computationally intensive).

The testing is performed locally, i.e. on the UI through a bash login, not on Grid WNs, and this is because otherwise the Grid queue times would increase the debug times by many orders of magnitude.

During a testing session, the Python interpreter will stop at every time it finds a syntax error (we will simplify our reasoning by assuming all errors are syntax errors and hence are detected by the interpreter). The user should then fix the error and restart the execution.

However, the real problem is (and this is one of the key reasons for which SETest was developed) that side-effect-based invocations such as invocations of Linux filesystem commands or external executables in the great majority of cases don't like files left over from previous executions. In other words, if the execution stops due to a syntax error and the user just corrects the Python error and then launches the pipeline again, the pipeline is going to immediately exit due to another error caused by the existence of a file which was not expected to exist on the filesystem at that point, and which in fact comes from the previous execution of the pipeline.

The net effect of this is that at *every* error found, the user needs to delete *all* intermediate files from the pipeline, recreate original tar archives and so on. Also, it is not possible to simply replace the whole directory of the job with a pristine version, because unfortunately this would also delete the fixes that one has done on the Python code up to that moment.

In addition to this, every time the pipeline is launched again, the execution is repeated. While for the errors in the first lines of code the execution takes milliseconds to reach there, when those are fixed and the errors which start to appear are in the middle of the pipeline, it can take half an hour to reach that point again at every execution. The user can indeed choose a simplified input for the debugging session so to shorten the execution times, but in many cases a valid input which has zero execution time does not exist, and the best that can be done is an input 15 minutes long or so.

If the pipeline has e.g. 115 lines of code (median case 30--200) and 1/5 (23) of these lines contain an error (remember there is no compiler to catch trivial errors), and as a simplification we assume that the execution time is equally spread across all the lines, the outline of the debugging session would be the following:

The execution will need to be restarted 23 times to fix all errors.

The first execution starts at line 0 and stops at line 5 (first error at the fifth line, 1/23 of the length) and takes 39 seconds (1/23 of 5 minutes). When the error is found the user needs to fix it (let's suppose 30 seconds, if the code is already open on the editor and just needs to be modified and saved) and then find and wipe all intermediate files by hand (let's suppose 20 seconds), then restart the execution.

The second execution starts at line 0 and stops at line 10 (first error at the 10th line, 2/23 of the length) and takes 1m:18sec, then the user needs to fix it (30 seconds), wipe intermediate files (20 seconds) and restart... and so on.

The whole debugging session would then take about 3 and a half hours.

The same debugging on a larger pipeline of 200 lines of code would take about 5h:45min, while on a shorter pipeline of 30 lines of code this would take about one hour.

By comparison, with SETest the times required for the 115-lines pipeline are roughly the following:

1. Wrap every side effect call: time required is about 10 seconds per call for a total of about 6m:23sec
2. Run 23 rounds of execution. SETest will skip every repeat of the same computation in successive runs, so, the 23 rounds of execution will take exactly 15 minutes to complete.
3. Fix 23 errors: this takes 23 x 30 seconds = 11.5 minutes

for a total of about 33 minutes (84% of time saved). In addition, the debug procedure becomes much less annoying with almost no dead times and, last but not least, the likelyhood that further errors may be introduced during the debugging process due to e.g. the user modifying the code to try to skip steps, is greatly reduced.

The same debugging on the 200 lines pipeline would take about 46 minutes (86% of time saved), and on the 30 lines pipeline it would take about 19 minutes (68% time saved). The benefit of SETest is hence significant.

In addition, SETest also has the added benefit of allowing regression testing, which would be otherwise very hard to perform for these side-effects based applications. When performed, regression testing should cut down debug times by at least 95% compared to a standard testing session performed from scratch, and give a *much* higher reliability due to the wide variety of input data that is possible to feed to the application being tested, in a short time and skipping all external computational steps.

Even if it is true that in the bioinformatics field probably not many pipelines are so long-lived to benefit from regression testing, still SETest can be used outside the bioinformatics and Grid domains.

5.3 Additional unquantifiable benefits

Besides the benefits presented in the previous sections 5.1 and 5.2, the solutions presented can provide additional benefits to the Grid infrastructure (in turn positively affecting other Grid users), and to the Grid VO managers and responsible personnel.

It is extremely difficult to quantify these benefits.

1. Vnas automated garbage collection system for virtual sandbox files ensures that no unused old file (garbage) remains forgotten on Storage Elements. This reduces SE occupation in the medium term, and relieves the VO Responsible Personnel and/or SE administrators from the burden of hunting misbehaving users and asking them to clear up their garbage from the Storage Elements.
2. The usage of Grid bandwidth is usually reduced when using Vnas's virtual sandbox functionality, compared to both the case in which the

overall size of programs and data is < 10MB and the user is using the Grid's native sandbox directly, and the case of overall size > 10MB and the upload of required files to storage elements is made with custom code inside the submission loop. Unless of course the user is very careful or implements an equality detection based on file contents such as that of Vnas.

3. The usage of internet bandwidth is greatly reduced by GridDBManager (as the download is local, versus remote download not using GridDBManager in the most common naive implementation with one replica). This reduction in bandwidth usage can have a series of effects and in particular it is likely to speed up network transfers by the other jobs located in the same CE. This would in turn reduce their total execution time (similar to: performance increase).

4. GridDBManager allows to effortlessly maintain older versions of the managed databases on the Grid environment, with an insignificant impact on the storage costs due to the novel approach based on reverse-delta files. Older versions of the databases might be of help to bioinformaticians who occasionally need to reproduce and verify results obtained in the past, or compare their results to others which were obtained in the past by other research groups.

5.4 High performance cDNA analysis - A Case Study

Some excerpts from my publication [TROMBETTI'06] follow. These are reprinted from Journal of Parallel and Distributed Computing, Vol 66, Issue 12, Trombetti G.A., Merelli I. and Milanesi L., "High performance cDNA sequence analysis using grid technology", p. 1482--1488, Copyright 2006, with permission from Elsevier.

The work described in the following subparagraphs was performed leveraging an early implementation of my Vnas framework (please see section 4.1 – *Vnas* for a discussion), in particular the completion-callback feature was used for triggering the next step in the computation. The framework didn't have a name at that time.

5.4.1 Abstract and Introduction

Innovative DNA sequencers, relying on pyrosequencing, are now being produced, which cut down costs and speed up sequencing by an order of magnitude. Hence the capability of handling high throughput sequencing is becoming increasingly important for bioinformatics.

This study concerns the development of a high performance pipeline for analyzing cDNA sequences produced by a high throughput pyrosequencer. Mainly, this analysis system has been developed by us to map the sequenced cDNA strands against a cDNA database for studying different mutations that can influence the genes functionality. The pipeline supports heterozygous organisms.

[...]

The results of this high performance pipeline are stored into an output database directly from the grid sites using the Web Services technology. By querying this database it is possible to inspect the analysis results to detect different mutations in the cDNA sequences, as well as other meaningful biological parameters and information.

[...]

This pipeline was designed to assemble the cDNA sequences starting from the sequencer output data, and using the human cDNA database as a reference, in order to identify punctual mutations in the expressed sequences.

Our main purpose is to detect punctual variations of the sequenced cDNAs in heterozygous organisms, in order to find either punctual mutations (genomic mutations [Garcia'00] present in isolate biological samples) or SNPs (Single Nucleotide Polymorphism - genomic variations present in a statistically relevant percentage of the population [Marsh'05]). These analyses are important to establish a relationship between mutations in the coding zone of DNA and genomic diseases.

5.4.2 Motivation

Problems in genomics and proteomics tend to have a quadratic or higher computational complexity [Karlin'93]. For example, global / local genome pairwise alignment with general or affine gap penalty functions, genome assembly, inversion distance computation, genome rearrangement analysis and molecular dynamics have all got a quadratic or higher complexity: small increases in the input data, due to the advancement in knowledge or improvement in machines providing the input, greatly increase the computation time. CPU speed increases also have been nonlinear (in facts exponential) for a long time, providing approximately a doubling in speed every two years; however, this is faulting lately, as the speed increases have almost stopped in recent years.

As far as genomic problems are concerned, also has to be taken into account that the development of sequencers has been far from linear in the last years, recently leading to high throughput pyrosequencers having a tenfold increase in throughput [Ronaghi'98], and similar decrease in operating costs, compared to the previous technology. Such high throughput pyrosequencer technologies create an enormous flow of genomic sequences that must be elaborated in minimum time to best exploit the sequencer capabilities.

In our case, for detecting punctual mutations, comparing each of the sequences output of the sequencer, called reads, against the whole cDNA database was necessary. Having as reference a large database of over 39,000 cDNAs [Wiemann'01], and the output rate of our pyrosequencer as high as 10,000 reads per hour it was not possible to keep up reliably with a single machine. In addition, we wanted to allow repeatability of past calculations after variation of the algorithm or parameters, which meant a potentially very large dataset to be recomputed in a reasonable time.

Hence, for the implementation of this pipeline a high performance system needed to be designed to coordinate a large number of computation resources and to manage the flow of such amount of information. This has been possible by leveraging a massively distributed environment such as the grid platform [Foster'01].

5.4.3 Implementation

First make it work, then make it fast. Our first approach was with a nondistributed pipeline (Fig. 10 below – *The analysis pipeline*). The first stage of our pipeline leverages Blast [ALTSCHUL'90] to match the reads against the 39000 cDNA reference database. From the Blast results groups are then made, gathering together the reads which best match the same reference sequence. These associations are written onto a database. In case of alternative splicing, two or more reference cDNAs will have common parts, hence, when the Blast matching is attempted, a read matching a common part will match all such reference cDNAs. Since at this first stage it is not possible to determine, or even guess, which reference cDNA this read belongs to, our policy is to accept all. During the next (second) stage, it will be possible to heuristically filter out some reference cDNAs (and their associated groups) which were not in facts expressed, based on the coverage of the cDNA which would be covered by reads only on the common parts of the alternative splicing. In the meanwhile, during this first stage we already distinguish the cases in which the second (or further) Blast match of a read against another cDNA is caused by alternative splicing (for which our policy at this stage is to accept it) from the cases in which the match is determined by random similarity (which we clearly want to reject). An heuristic and partly adaptive algorithm solves this.

Once groups are made, a second computation stage clusters the reads together, using the cDNA as a reference [PARSONS'92]. For this second stage again we leveraged Blast, this time for anchoring the sequences (as Blast ``subjects'') against the reference cDNA (as Blast ``query''), to obtain a multiple alignment of the reads referring to each cDNA [BELSHAW'05].

A third computation stage analyzes each multi-alignment obtained at the previous stage looking for punctual mutations. We remark that we are working with gene expressions from heterozygous organisms, which can have any punctual mutations present on either homologous chromosome, or both of them. In the same way, gene expressions also will.

Hence, when looking at multiple alignments, it is to be kept in mind that, on average (even if with a significant variance), half of the aligned reads should

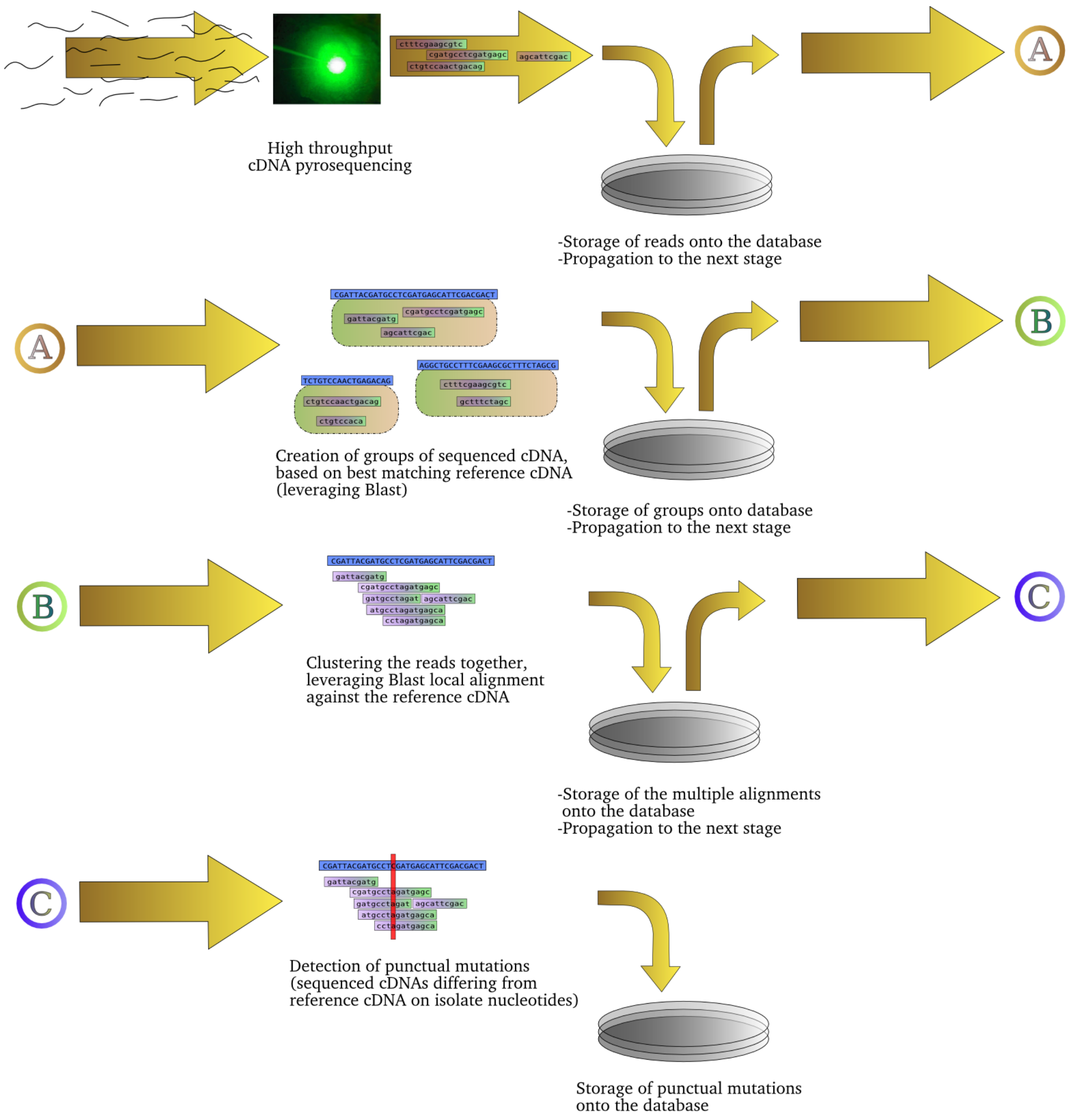
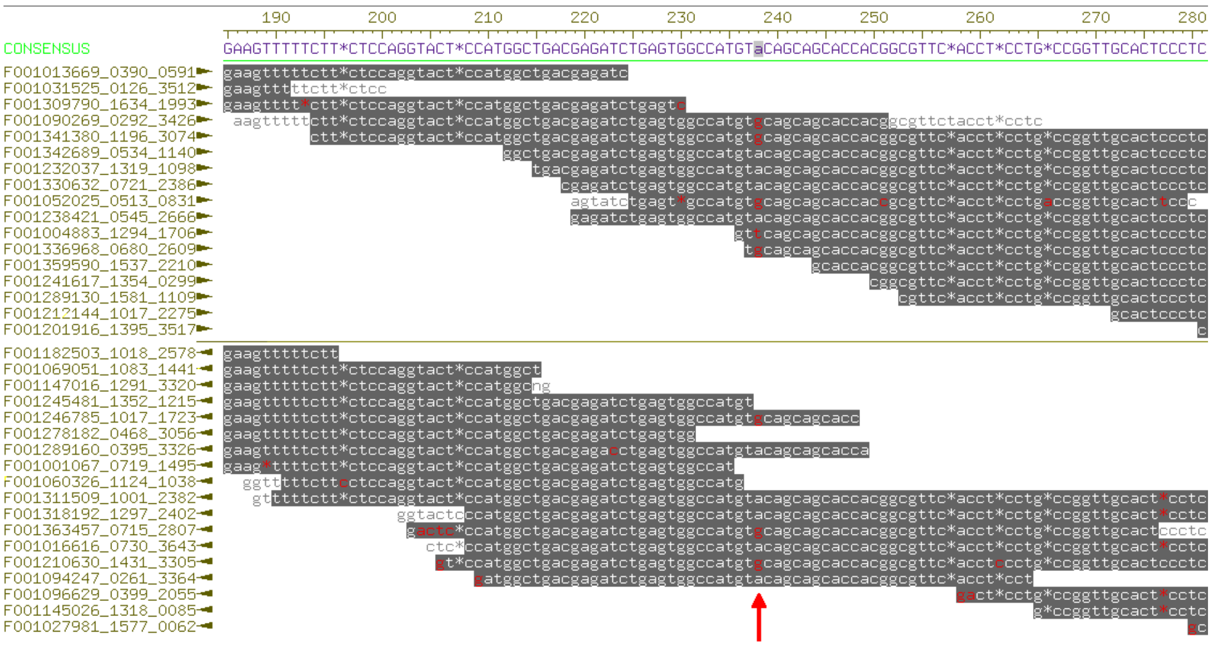

A
ctttcgaagcgtc
cgatgcctcgatgagc
agcattcgac
ctgtccaactgacag
High throughput
cDNA pyrosequencing
-Storage of reads onto the database
-Propagation to the next stage
A
B
CGATTACGATGCCTCGATGAGCATTCGACGACT
gattacgatg
cgatgcctcgatgagc
agcattcgac
TCTGTCCAACTGAGACAG
ctgtccaactgacag
ctgtccaca
AGGCTGCCTTTCGAAGCGCTTTCTAGCG
ctttcgaagcgtc
gctttctagc
Creation of groups of sequenced cDNA,
based on best matching reference cDNA
(leveraging Blast)
-Storage of groups onto database
-Propagation to the next stage
B
C
CGATTACGATGCCTCGATGAGCATTCGACGACT
gattacgatg
cgatgcctagatgagc
gatgcctagat
agcattcgac
atgcctagatgagca
cctagatgagca
Clustering the reads together,
leveraging Blast local alignment
against the reference cDNA
-Storage of the multiple alignments
onto the database
-Propagation to the next stage
C
CGATTACGATGCCTCGATGAGCATTCGACGACT
gattacgatg
cgatgcctagatgagc
gatgcctagat
agcattcgac
atgcctagatgagca
cctagatgagca
Detection of punctual mutations
(sequenced cDNAs differing from
reference cDNA on isolate nucleotides)
Storage of punctual mutations
onto the database

Illustration 10: The analysis pipeline - The three pipeline steps are shown in row 2, 3 and 4. Row 1 is the pyrosequencing step. Between any two steps, computation results of the previous stage are stored into the database (located on the central server in the grid implementation). Outgoing arrows at the right side of the picture connect to incoming arrows at the left side (A-A, B-B, C-C).

come from a chromosome and the other half from the other homologous. In a column of aligned reads where more than one nucleotide is present, the most present nucleotide will most certainly be that of (at least) one chromosome, but the second most present nucleotide can either be a sequencing error or the expression of the gene on the homologous chromosome (Fig. 11 – *Clustered reads belonging to a cDNA*)



```
                          190        200        210       220       230       240       250        260         270       280
CONSENSUS             GAAGTTTTTCTT*CTCCAGGTACT*CCATGGCTGACGAGATCTGAGTGGCCATGTaCAGCAGCACCACGGCGTTC*ACCT*CCTG*CCGGTTGCACTCCCTC
F001013669_0390_0591► gaagtttttctt*ctccaggtact*ccatggctgacgagatc
F001031525_0126_3512► gaagtttttctt*ctcc
F001309790_1634_1993► gaagtttt*ctt*ctccaggtact*ccatggctgacgagatctgagtc
F001090269_0292_3426►  aagtttttctt*ctccaggtact*ccatggctgacgagatctgagtggccatgtgcagcagcaccacggcgttctacct*cctc
F001341380_1196_3074►          ctt*ctccaggtact*ccatggctgacgagatctgagtggccatgtgcagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001342689_0534_1140►                              ggctgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001232037_1319_1098►                                 tgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001330632_0721_2386►                                    cgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001052025_0513_0831►                                     agtatctgagt*gccatgtgcagcagcaccaccgcgttc*acct*cctgaccggttgcacttccc
F001238421_0545_2666►                                     gagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001004883_1294_1706►                                                      gttcagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001336968_0680_2609►                                                       tgcagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001359590_1537_2210►                                                              gcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001241617_1354_0299►                                                                    cggcgttc*acct*cctg*ccggttgcactccctc
F001289130_1581_1109►                                                                       cgttc*acct*cctg*ccggttgcactccctc
F001212144_1017_2275►                                                                                             gcactccctc
F001201916_1395_3517►                                                                                                      c
F001182503_1018_2578◄ gaagtttttctt
F001069051_1083_1441◄ gaagtttttctt*ctccaggtact*ccatggct
F001147016_1291_3320◄ gaagtttttctt*ctccaggtact*ccatggcng
F001245481_1352_1215◄ gaagtttttctt*ctccaggtact*ccatggctgacgagatctgagtggccatgt
F001246785_1017_1723◄ gaagtttttctt*ctccaggtact*ccatggctgacgagatctgagtggccatgtgcagcagcacc
F001278182_0468_3056◄ gaagtttttctt*ctccaggtact*ccatggctgacgagatctgagtgg
F001289160_0395_3326◄ gaagtttttctt*ctccaggtact*ccatggctgacgagacctgagtggccatgtacagcagcacca
F001001067_0719_1495◄ gaag*ttttctt*ctccaggtact*ccatggctgacgagatctgagtggccat
F001060326_1124_1038◄   ggtttttcttcctccaggtact*ccatggctgacgagatctgagtggccatg
F001311509_1001_2382◄    gtttttctt*ctccaggtact*ccatggctgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcact*cctc
F001318192_1297_2402◄                   ggtactcccatggctgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcact*cctc
F001363457_0715_2807◄                    gactc*ccatggctgacgagatctgagtggccatgtgcagcagcaccacggcgttc*acct*cctg*ccggttgcactccctc
F001016616_0730_3643◄                      ctc*ccatggctgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cctg*ccggttgcact*cctc
F001210630_1431_3305◄                       gt*ccatggctgacgagatctgagtggccatgtgcagcagcaccacggcgttc*acctccctg*ccggttgcactccctc
F001094247_0261_3364◄                           gatggctgacgagatctgagtggccatgtacagcagcaccacggcgttc*acct*cct
F001096629_0399_2055◄                                                                             gact*cctg*ccggttgcact*cctc
F001145026_1318_0085◄                                                                                     g*ccggttgcact*cctc
F001027981_1577_0062◄                                                                                                     gc
```

Illustration 11: Clustered reads belonging to a cDNA – An A−−>G punctual mutation in one chromosome only (heterozygosis) is shown.

After a statistical analysis we decided that a coverage of 10x (at least ten overlapping strands over each and every point of the whole multi-alignment) should be used to distinguish reliably between the two, and we put the threshold of 30% (of the coverage over that specific point, i.e. the column height) for calling the expression of the second allele.

5.4.4 Distributed Implementation

Our distributed implementation for this pipeline shares the computational load over the grid nodes of EGEE grid. The whole pipeline is coordinated by a single central server, on which the grid *User Interface* software is installed. This creates a high performance and relatively scalable system according to the grid performance and the power of the central server (the central server can act as a bottleneck in the general case depending on the amount of work assigned to it, but such work was small enough in our case). The central server is both involved in the coordination of the parallel execution of the grid jobs through the *User Interface* and in the pre and post elaboration of sequences.

To obtain a high performance implementation of this analysis pipeline the most time-consuming steps have been implemented in a distributed way. The first step implemented on the grid platform is the blast that groups sequences according to cDNA similarity. The input data of this step are the raw sequences produced by the sequencer while the output is used as input of the second distributed step, anchoring each read to the related cDNA for creating a complete coverage.

Both of these steps are based on blast and their implementation is quite similar. Bioinformatics application that relies on the comparison of an input sequence against a database are usually implemented on a distributed platform by subdividing the input dataset in small groups [Merelli'05]. To manage the distributed implementation of these pipeline steps an efficient system has been implemented to coordinate the execution of the jobs, to control the completion status and to retrieve the output in case of a successful termination.

For each job a JDL script is generated with the information about the input sequence, the job requirements and the information about the databases that have to be accessed. The jobs are routed by the Resource Broker to the best *Computing Element* that is available at the moment. From the *User Interface* the execution of the pipeline analysis is automatically monitored by our software and, in case of failure, is re-submitted to the grid infrastructure.

Porting onto the grid platform bioinformatics applications relying on databases implies dealing with distributed database management. In the first analysis step of this pipeline, input sequences are clustered according to a database of cDNA. This is a flat file database of significant size which needs to be transferred to the used *Computing Elements* prior of invoking Blast. In order to minimize the transfer time we replicated the cDNA flat file database to various *Storage Elements*. In this way it is possible to use a high number of *Computing Elements* (each located near one of such replicas, i.e. local network) while keeping the database transfer overhead for the execution of step one minimal.

Grid technology does not support distributed RDBMS. This is a problem when the output data has to be collected in a database. Although it would be possible to retrieve data into the User Interface and parse them before storing results in the database, this solution would remarkably slow down the system performance. To obviate this problem a solution based on the technology of the Web Services has been implemented [FOSTER'02].

For each grid job, the blast output is parsed directly on the Computing *Element* on which it has been executed. In this way a temporary result set is created and, eventually, through a small Web Service client carried into the grid together with the input sequence, it is entirely stored into the results database on the *User Interface*. This is performed in one pass, hence minimizing the SOAP communication overhead [MERELLI'05-II]: the Web Service receives the incoming SOAP message from the client containing information about the blast results, and performs the SQL insert onto the results database.

5.4.5 Results Database

The working data are passed from a computation stage of the pipeline to the next through the help of a database (Fig. 12). In the non-distributed version of the pipeline, the previous stage would store the result on a database which the next stage would read. This ensured complete separation of the stages, and made the repeatability of computations very easy, implying a simple deletion of some result rows of a certain stage from the database (or cloning of some input rows of the same stage) and re-run of the computation stage.

In the current, distributed version of the pipeline, at the end of a stage computation a Web Service is used to dump the results onto the database on the *User Interface*, which is then responsible for passing such data to the next stage. This does add some overhead, however it ensures that:

- the stages are fully separated
- the computation is repeatable in part or whole
- the intermediate results are held on a single and local database.

Results of intermediate calculations may hold important information for biological analyses which might not even be fully foreseen at this point. In order to allow inspired searches by the biologists, we kept all such database information meaningfully ordered and easily searchable with SQL queries and scripts.

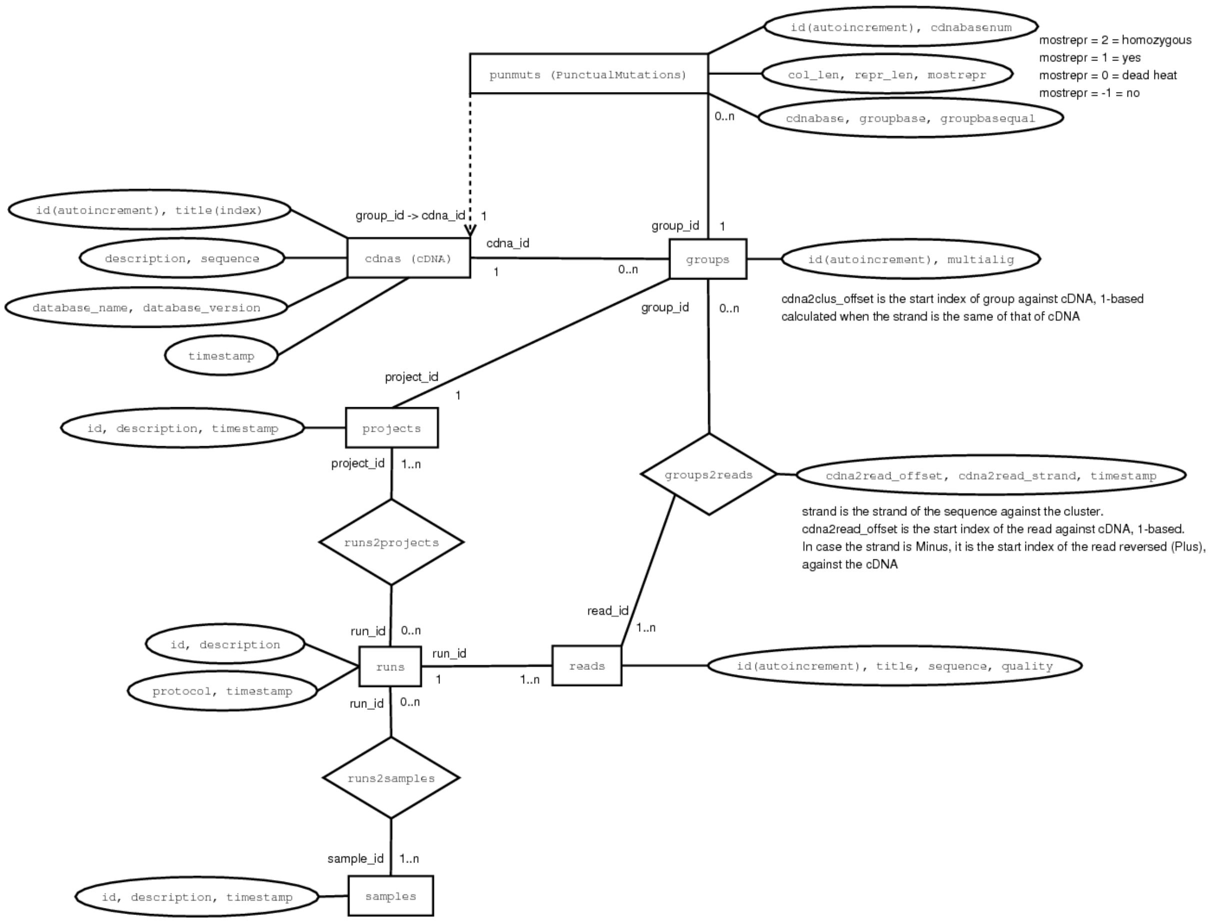

id(autoincrement), cdnabasenum
mostrepr = 2 = homozygous
mostrepr = 1 = yes
mostrepr = 0 = dead heat
mostrepr = -1 = no
punmuts (PunctualMutations)
col_len, repr_len, mostrepr
cdnabase, groupbase, groupbasequal
0..n
id(autoincrement), title(index)
group_id -> cdna_id
1
group_id
1
cdna_id
description, sequence
cdnas (cDNA)
1
0..n
groups
id(autoincrement), multialig
database_name, database_version
group_id
0..n
cdna2clus_offset is the start index of group against cDNA, 1-based
calculated when the strand is the same of that of cDNA
timestamp
project_id
1
id, description, timestamp
projects
project_id
1..n
groups2reads
cdna2read_offset, cdna2read_strand, timestamp
strand is the strand of the sequence against the cluster.
cdna2read_offset is the start index of the read against cDNA, 1-based.
In case the strand is Minus, it is the start index of the read reversed (Plus), against the cDNA
runs2projects
read_id
1..n
run_id
0..n
id, description
run_id
runs
1
1..n
reads
id(autoincrement), title, sequence, quality
protocol, timestamp
run_id
0..n
runs2samples
sample_id
1..n
id, description, timestamp
samples

Illustration 12: Entity-Relationship diagram for our current database - 10 tables. Biological samples are sequenced in pyrosequencer runs producing reads. Computation analysis projects are initiated on the reads of one or more runs, producing groups of reads according to Blast-detected most similar reference (cdnas) in pipeline step 1. These groups are clustered in multialignments (multialig) in pipeline step 2 and their punctual mutations (punmuts), either homozygous or heterozygous, are finally detected in pipeline step 3.

Results of intermediate calculations include:

- grouping of the reads mapping onto any cDNAs, for each biological sample, potentially giving information for the relative amounts of gene expression for each cDNAs
- multi-alignments within the groups
- punctual mutations found (homozygous / heterozygous), whose frequency could be investigated to find new SNPs, or correlation between the presence of different SNPs
- reads not matching reliably any known cDNA, which can be a clue for

a new unknown cDNA that could be investigated further via multiple alignments among such reads, and could then be tested for ``amount of expression'' comparing it against genomic data collected in the past (e.g. unknown multiple splicing of a known cDNA).

5.4.6 Distributed Implementation Performance

Even though the performance of the grid pipeline is more than adequate for our situation, a numeric estimate of such performance is difficult due to the great variance in queue times for jobs sent on the Grid. This depends mainly on the workload which is assigned to the Grid throughout Europe at the specific moment of submission. In addition, the highest the number of jobs which are submitted together to the Grid, the most unfavorable (in terms of queue times) the computing resource the last of those will get. Needing to wait the execution of all jobs makes the pipeline wait for the worst queue time of the set of jobs, hence, the pipeline-perceived queue wait time for 40 jobs is significantly worse than the pipeline-perceived queue wait time for 10 jobs.

It is also obvious that, given a certain amount of computation to be performed, splitting such computation into a high number of grid jobs will reduce the size for each of them, and since the jobs are executed independently on the grid, raising the number of jobs increases the execution parallelism and reduces the computation time as it is perceived by the pipeline. Hence it is clear that splitting the workload in a number of grid jobs too small would also hinder the pipeline performance. In facts, in our executions we try to keep the length of every job comparable to the (estimated) queue wait time of the set with a lower limit of 20 minutes (in order to avoid excessive grid overhead).

Our performances are best described by the following: the nondistributed version of our pipeline needs 19 computation hours on a Xeon class CPU for the 100,000 reads that an high throughput pyrosequencer is ideally able to produce in 1 hour. These are roughly 10 hours for the first step, 6 for the second step and 3 for the third step. With the grid distributed version of our pipeline we can parallelize the three steps as much as we want by raising the number of grid jobs we submit. A good compromise for the number of jobs for obtaining the completion of such a computation in the shortest time considering the queue wait time is around: 20 jobs for the first step, 12 for the second and 6 for the last step, for a total of 36 half-an-hour-long jobs. A typical queue wait time is 30 minutes for the first step (a 20-job set) 15 minutes for the second step (a 12-job set) and 10 minutes for the third step (a 6-job set) making an total wait+execution time of two and half hour on average (but the variance can be significant, as we said, depending on the EGEE Grid load). In addition, our computation resources remain still substantially free and capable of submitting and handling more grid computation if this is needed (e.g. for a recomputation of older genomic data with altered pipeline parameters).

The benefit of the Grid is hence very evident, at least for heavy computational loads, and the costs of joining the Grid are minimal (sometimes zero, depending on the load you intend to submit to the Grid) compared to those of a dedicated cluster.

6 Conclusion and future work

Bioinformatics, physics and other mathematics-based sciences using computational approaches for research purposes are always in need of more and more computational power for solving algorithms of ever-increasing complexity over data of ever-increasing size. In this scenario, computational grids and in particular the European and world's largest EGEE Grid platform show great promises, delivering to the hands of the scientists an enormous computational power which can be used already today to perform innovative research activities.

Unfortunately this Grid architecture is still new and not free of issues. These issues, which were discussed in this thesis, greatly complicate the work of the scientists who need to create and launch computational pipelines and applications in the Grid environment.

In this Ph.D. thesis three major frameworks and an additional smaller standalone solution have been presented, which can solve most of the main Grid issues by creating abstractions adding reliability over the most common stability and availability problems, bypassing some Grid limitations, increasing the Grid performances, reducing bandwith usage, and abstracting away much of the effort that is associated with writing and testing a computational application for the Grid environment.

More specifically, in chapter 5 the impact of the solutions was examined. An approximately two-fold increment in the perceived Grid performances was demonstrated through monte-carlo simulations based on experimentally probed data and simulated submission algorithms. In addition, a reduction of up to 72% of the lines of code needed to develop and manage a computational pipeline in the Grid environment was shown. Additionally, the SETest testing framework allows a significant reduction of the debug times of up to 86% for Python computational pipelines scattered with side effects, such as those commonly used for the execution of bioinformatics pipelines in the Grid environment.

The solutions presented include innovative concepts, and were published in three articles in scientific journals and four conference proceedings, and orally presented in five internationational conferences during the three years' Ph.D. course.

Future work related to these solutions will be in the direction of handling the failures and information losses by the Replica Metadata Catalog, which is the single most problematic component remaining after what is already handled by the current version of the frameworks.

7 Appendices

7.1 BGBlast's Grid Database Manager algorithm details

BGBlast's ARM optimizes the number of replicas for each BRD separately, by minimizing the sum of the storage cost and user wait time cost. The algorithm is an iterative algorithm which converges on the optimal number of replicas and the optimal location for them, simultaneously.

The ARM optimization algorithm at each cycle evaluates the benefit of the addition of one replica and the benefit of the removal of one replica.

During the evaluation of the addition of one replica, the ARM takes into account the specificity of each Grid location suitable for replication (i.e. every SE not yet holding a replica), hence finding the best location for an added replica. The ARM then evaluates whether the addition of a replica in that specific place is profitable or not, using the costs formula.

The best location for adding a replica is ideally a SE having a large amount of free disk space (so to cause a proportionally little impact when adding the replica) near a CE being large in the number of nodes (a larger computing power means that the job queue is generally consumed more quickly).

The costs formula for evaluating variations in replicas numbers considers the Grid queue times to be inversely proportional to the number of nodes useable by BGBlast, i.e. those having a replica nearby (see section 4.2.4.2.1 *"Motivation for ARM"*). The correctness of this assumption can in fact be demonstrated under some simplifying assumptions. The cost of a minute of user wait time is to be specified in the BGBlast configuration file.

The cost of Grid storage is the other cost to be specified in the BGBlast configuration file. The cost of storage is to be expressed in terms of cost per percent of free storage space occupied per day on a SE. This approach was chosen for reflecting the intuitively higher impact on other Grid users that a GB-sized file has when uploaded on a small or already full SE compared to the impact it has when uploaded on a SE with plenty of free space.

The ARM engine hence works by minimizing the sum of the storage cost and user wait time cost, for each BRD separately.

The process for evaluating the benefit of the removal of one replica is analogous. The worst existing replica is chosen using the same kind of analysis as described above. The cost formula is then recomputed while simulating the removal of the "worst" replica, and the result obtained in this way is compared to the cost associated to the current situation. If the cost after the removal of the replica appears lower, the replica is removed.

This algorithm converges quickly.

7.2 SETest framework implementation details

The framework can be used from the application to be tested by simply importing it as `SETest` then instantiating the main singleton class like

```
  st = SETest.SETest()
```

then

```
  retval = st.do(fun, args, kwargs)
```

will be the wrapper call for

```
  retval = fun(args, kwargs)
```

The SETest framework is to be configured before launching the application using it, so to set the correct functioning mode (Bypass, Recording, etc.) for the framework as described in section 4.3.3.2 . The framework is not intended to be normally configured through the application using it, but externally, this choice was made in order to preserve the transparency of the framework to the application. Were it not so, one would need to add the parsing of options for configuring the testing framework in the code of *every* application using it, and I believed this was not the intended behaviour for this kind of framework.

It is hence possible to configure the framework by launching it as if it was an application itself:

```
  ./SETest.py <options here>
```

this will store the configuration options in a configuration file. Among the most important options are

– the operating mode (Bypass, Recording etc.)

- policy on dataset lookup failure (exception or interactive)
- the name and path of the dataset file to be used
- log file name (for precisely logging all activity by SETest)
- log file verbosity level

whose location is specified within this launch itself (one of the options, mandatory). This location should preferably be in the same directory as the tested application, so that, at the beginning of the execution of the application, when the SETest singleton is instantiated, the SETest configuration is loaded from the SETest.config file located in the same working directory. It is also possible to specify a different location for the configuration file, and the first call to the singleton (instantiation of the singleton) would then become

```
  st = SETest.SETest(configfilelocation)
```

In this case multiple preconfigured configuration files can coexist for the same tested application, however, in this case the programmer of the application (e.g. computational pipeline) needs to add the parser for parsing the configfilelocation option, so this solution is more flexible but a bit less transparent to the application being tested.

The implementation of this framework heavily relies on introspection and serialization features of Python. Parameters are stored to the dataset as a hash of their serialized value, while results are fully serialized. Python iterators are unpacked to lists in order to perform equality tests and serialization. Clearly this has some overhead on memory usage, however there didn't seem to be a better strategy in this case. The unpacking is not performed in *Bypass* mode, which simply relays calls.

Non-serializable objects such as network sockets cannot currently be used with SETest (their value would have no meaning at the next execution), and this is the biggest limitation for this framework. However network communication can usually be wrapped at a higher level: you can still wrap the call which performs the connection + download / upload.

Database connections and cursors are supported as a special case in SETest (these also are non-serializable objects), due to their high importance in computational pipelines, however, their behaviour is simplified in the current version of the framework: connections to the database are not stored in the dataset and the connection is created again at every query. Commits are performed at every query (transactions not possible, except of course in bypass mode). Cursors can be stored to the dataset, and retrieved in playback mode (an emulated cursor is provided by the framework), however, as it happens for iterators, all rows will be loaded into memory. In bypass mode everything works normally and there is no overhead.

Last minor feature is the break_next() call which sets a breakpoint in the SETest framework so that the user will be dumped to the interactive shell at the next wrapped call, no matter what mode SETest was in or whether the call was present in the dataset or not. The purpose is the same as that of breakpoints in debuggers.

7.3 Developing applications for the Grid Environment

Notwithstanding the fact that within the frame of the BioinfoGRID [BIOINFOGRID] project, other researchers and I ported to the Grid a large number of the most common bioinformatics applications, thinking that no Grid user, or even Grid-bioinformatician, would ever need more than such standard applications is unrealistic.

Bioinformaticians and/or researchers in general will probably come to a point where they find that a computational pipeline for the specific task they need to accomplish does not exist yet, or maybe is not specific enough and should be improved, fixed, optimized or totally rewritten for such specific task.

In this case the Grid user will need to become a Grid application developer. Developing applications for the Grid is relatively difficult, as the platform is significantly different from almost anything else. The architecture of the Grid will have to be learned (and for this specific documentation exists), but after this, experience will still be lacking.

Hence, in this section I will write some recommendations for building applications for the Grid environment. These recommendations are mainly aimed at scientists approaching the Grid, and are based on the experiences gathered in my 3-years Ph.D. course in collaboration with the ITB-CNR and being involved into the BioinfoGRID project.

This section also helps better understanding the difficult work associated to the creation of Grid applications starting from scratch, i.e. without the help of frameworks or other software not part of the standard gLite installation. In fact in this section I will by purpose not refer to existing frameworks for easing the Grid development, not even those made by me and mentioned in the rest of this thesis. This section can in fact also act as a motivation for my Ph.D. work, and as a hint on the impact of solutions.

This section is novel and the recommendations were never published elsewhere up to now. The next sections are written down as an informal guide conceived for the standard Grid user, henceforth addressed as "you".

7.3.1 Introduction

The following sections presume knowledge of programming topics, at least in the localized environment, and some knowledge of network programming.

Reading the official documentation at [GRIDIT] is essential for getting started on the architecture and commands for the Grid and it is a prerequisite for understanding the following sections and the technical Grid terms which will be used. All command-line commands available from a User Interface node also have a man page, which covers the usage options in good detail.

7.3.2 General recommendations

As programming languages for developing Grid frameworks and applications, we were using mainly Python and Perl and this is what I would suggest, though most general purpose programming languages should work fine. Whatever the language you use, you should make sure that your application executes properly in a machine running Scientific Linux 3.0.4 + Glite framework, because this is what the great majority of EGEE Grid machines is running. You should test your application as well as possible before submitting it to the Grid, because receiving errors back from the grid is very frustrating. A “remote” debug would be difficult and extremely time-consuming. Some more helpful information can be found in the “Developing resilient program flows” section below.

Your application, together with the data files it needs for computation should fit within the sandbox that your VO has set for you, which is usually 10MB. If you need more space than that, you should upload the needed files by hand to a Storage Element and code your job so to download these files prior to starting the computation. Remember to delete those files from the SE when these are not needed anymore. Here there is room for various types of optimizations; some techniques will be described in the “Space, bandwidth and time optimizations” section below.

The execution time of your application should be thought of carefully. The application should not run over the time limit imposed by your certificate, which usually means making computation jobs shorter than 24 hours, computations longer than that will have to be split into multiple shorter jobs. The job also should not be too short, or the overhead for the RB and the queuing system of the CE would be comparably too high and you would be wasting common resources. Grid time is a precious resource and should be used wisely and sparely.

Grid jobs can fail: distributed environments are always unreliable. The Grid is very distributed, hence it is very unreliable. You should monitor the jobs you submitted and you should resubmit such jobs if you see them fail (see command glite-job-status). In order to be able to do this, you will need to keep the submission files until you are sure that your job has succeeded. Jobs also can remain stuck in the queue for a very long time (days... these problems are mostly due to misconfigurations in some CE of the Grid) so you should also decide a queue-time-ceiling threshold after which you programmatically kill and resubmit a job which is still declared as queued. This topic is covered in greater detail in the “Grid submission algorithms” section below.

7.3.3 Developing resilient program flows

Programming for a distributed environment is significantly different from programming for a local environment because of the intrinsic unreliability of distributed environments. This is particularly true for the Grid, where the various parts of the system are very distant and maintained by different people.

If in the localized environment the commands succeed practically every time, on the Grid for every distributed command you use there is a significant likelyhood of failure. This means that if in your job you use 10 gLite commands and you never check return values, your job is going to fail often. “How often” is difficult to say, as it widely depends on which glite commands you use, but connecting to SE, downloading files, determining the nearest SE, uploading files, replicating files, are all risky operations.

When you program for the Grid you should always check the returnvalue for every glite command you use (the “$?” variable in bash: nonzero means error). When programming the code for a job, you should think if alternative actions exist for a glite command that can fail. If there is a possible alternative action which can allow the job to go ahead (e.g. Can you download the needed file from another source? Can you upload your results to another destination? Can you skip the action this time, was it really needed?) you should code that alternative action in your job. If there is no possible alternative action, you should still identify the error and quit the job with failure (nonzero return code) and this should be done (if possible) **before** wasting hours of computation on incomplete/wrong data. Upon failure of a command, the job should also print some helpful (to you) message to stderr which you can fetch with glite-job-output and you can use to understand what exactly has gone wrong. If this happens often, at a certain point you might want to change your code using a more reliable technique.

Languages using exceptions can help in this a great deal and languages providing deterministic destruction upon exceptions (C++, Python) can help even further. Using these features you can easily create program flows which provide alternative execution paths upon failure of some commands without scattering your code with *if* statements. A code scattered with *if* statements is difficult to read, and this reduces code maintainability.

7.3.4 Space, bandwidth and time optimizations

This section is dedicated to Grid users who need to develop a Grid application for which the amount of data to be processed is significant, beyond the capability of the sandbox, in particular if the download time for such amount of data is not negligible.

The simplest approach, as already described, is to upload the input data files (data files which your job needs for the computation) to a Storage Element of the Grid. In a first unoptimized implementation, the SE that you choose for holding these files is not even relevant.

It is very important that you remember to delete the uploaded data once they are not needed anymore. Many Grid users unfortunately forget to do so, and the Grid SEs in the years have become loaded by leftovers of ancient Grid computations. In order to take care of this and avoid unneeded headaches, it is suggested that you write or find some kind of automatic cleanup manager, which you preset with a timeout (number of days) approximating your project length (plus some margin). When the timeout expires, the cleanup manager would delete such files for you.

Another approach is to upload the input data files to a machine of yours located outside of the Grid and make these files accessible via FTP or HTTP protocols. Since wget is installed on every WN machine, this can be used for the download. This approach has pros and cons. The pro is that you don't need to take care of removing leftover files from Storage Elements, the cons are that your “external” (to the Grid) storage server will never be located near the computation site (remote download means slow download speed) and that you cannot do data replication (more on this, below). With this approach you are directly responsible for your storage server to be online and reachable at the time your jobs enter execution. This is both a pro and a con: you don't rely on the system administrator of the SE for your jobs to execute properly, but you rely on yourself for the same task.

In either approach, in case you are going to submit a batch of jobs together (this is the common use case for the Grid) and not just an isolated one, it is wise to upload the input data which changes between the different jobs and the input data that is constant for all the batch as two separate entities. If you don't do this, e.g. you make a single tar file of all the input data, you will need to upload again the whole tar file for each job, instead of the only changing part, and this would at least waste time and Grid bandwidth during upload, probably would waste storage space, and makes replication (see below) practically infeasible. This might not be an issue if the input data is not very large anyway, e.g. less than 50MB, or if the changing part is much larger than the constant part.

As far as the download time for input data within your job is concerned, I recommend that attention is paid so that the download time (an overhead) be *at least* an order of magnitude smaller than the computation time (the useful part). You should not create jobs that download for half an hour then compute for 20 minutes. Keep in mind that when your job is running it is occupying a *”slot”* of the Grid, whatever it is doing, downloading or computing, without distinction. No other job can start in your slot while your job is running, they will be waiting in the queue until your job exits, so please try to optimize your jobs so that the overhead time (time for downloading files and other preparations your job might need) is no more than 10% of the productive time. In the mentioned example, you should either reduce (optimize) the download time, or increase the computing time, possibly both.

A technique for dramatically reducing download times is to constrain the job go executing near the location of the input files. This requires that you used a SE of the Grid for your input files, and not an owned external storage server. The location of the files (name of Storage Element) should be known from the upload operation, or can be reversed from the LFN name using *lcg-lr*. The execution of the job can be constrained by using the *--resource* option on *edg-job-submit* or through the JDL file (see official documentation). Both constraining techniques actually need the name of a CE while you only know the name of the SE where your files are. The following command gives you the proximity map of CEs to SEs, that is, it tells you for each CE what SEs are connected in local area network to it:

```
lcg-infosites --vo <yourvoname> -f closeSE
```

Having the input file in local area network means a download speed which is one order of magnitude faster than remote download (which is over the internet), hence, the download time for these input files during the execution of your job will decrease 10-fold.

However, constraining the execution to a single CE might be too much, as the queue times on that CE might be unfavourable at the time you submit your job. If this is the case, you have another option: replication. You can replicate (command *lcg-rep*) the same input files on multiple SEs then you can use the JDL file to constrain execution to the related group of CEs. Replication is very bandwidth consuming for the Grid and can be time consuming for the initiator (you) so is only meaningful if 1) the constant part of your input data is much larger than the variable part 2) you are replicating only the constant part, and 3) you are going to use that constant part for a significant number of jobs (i.e. the batch is large). At the end of your batch it is imperative that you remember to remove all the replicas of your input files.

Lastly, I will disclose a neat optimization that you can do with replication. Starting with only one replica, you would start sending the jobs without constraining the location of execution. At the moment of downloading the input files, you would not download these files directly, instead you would first replicate the remote files to the SE which is nearest to your current location (which will be in local area network). You can determine the closeSE name from the system variable VO_VONAME_DEFAULT_SE (replace VONAME with the actual VO name, all uppercase) if existing. After the replication is complete, you would download the files from the closeSE. This technique takes 10% more time than the simple download from remote location, but will also create an additional replica as a side effect. Essentially, you get an additional replica almost without overhead. After some job runs, the most used CEs will already have a replica near to them, and you would start saving download time. At a certain point you might also start constraining to execute near an SE where a replica already exists. Also with this technique, it is imperative that you remember to remove all the replicas that were generated for all your input files at the end of your submission task.

7.3.5 Grid Submission algorithms

Depending on the task you have to accomplish (or the “problem” you have to “solve”), the approach for submitting it to the Grid can vary widely.

Depending on the task, a few important parameters related to your Grid submission are intrinsically determined; I will call these the “problem parameters”. The problem parameters typically are:

1 the total number of computation hours (and this also affects the batch submission size i.e. the total number of jobs)

2 the size of the input files, both the variable part and constant part

A certain number of parameters for your submission can still be set by you, and these will be called the “submission parameters”. The submission parameters typically are, at least:

3. the computation length for each job (affects the batch submission size)
4. queue-time-ceiling threshold (affects the response wait time)
5. location of the input files (remote / closeSE) (balances the set-up overhead towards the bandwidth overhead)
6. rate-limiter setting (more on this below here)

Here are some words of explanation for every point mentioned:

The total number of computation hours is clearly a parameter intrinsic to the problem you have to solve, at least in our approximation (if you can optimize that, go for it!).

The total number of computation hours contributes to determine the number of jobs that have to be submitted to the grid, which I call the *submission size*. The other parameter that determines the submission size is the computation length for each job. The submission size (overall number of jobs) is obviously TotalCpuHours/JobComputationLengh .

The size of the input files (variable and constant parts across the batch) is the other parameter intrinsic to the problem you have to solve. Depending on how big the constant and variable parts are, you might be able to perform some kind of optimizations, which will result in having the files locally or remotely at job execution time (a submission parameter). Please see detailed discussion in *“Space, bandwidth and time optimization”*.

As I already mentioned, Grid jobs can remain stuck in the queue for a very long time, and these jobs won't be able to enter execution before the expiration of the certificate. These problems need to be addressed with a queue-time-ceiling threshold: the job should be killed and resubmitted when its time in the queue reaches the preset threshold. By setting this threshold you will set the aggressiveness of the resubmission algorithm, which has dramatic consequences for small submissions which have fast-response (quick turnover) requirements, while it can be left much more relaxed for large submissions not having fast-response requirements. Even for large submissions of the latter case, it is anyway important that a threshold exists, and a reasonable setting is it at 8 hours, while for fast-response requirements it might be set much lower such as at 20 minutes. Also see the next section.

The rate-limiter parameter determines the maximum number of jobs that you will have simultaneously running on the Grid. Rate-limiting your submission at 100 jobs means that you will not submit the 101th job until one of your jobs has returned. Rate limiting your submission becomes imperative if the submission size is larger than a few thousand jobs: if you submit more than a few thousands jobs simultaneously you could monopolize the Grid resources effectively causing a denial-of-service to the other Grid users. This is absolutely to be avoided.

As you probably have guessed by now, for large sized submissions you will need to use an automatic submitter and job monitor, which can be either a third party one or you can create your own. It is not extremely difficult to create an automatic submitter, and while making it you will learn a great deal about the functioning of the Grid. If you develop your own submitter you can also adapt it better to your requirements.

The automatic submitter should take care of:

1.launching jobs and monitoring execution

2.resubmit jobs if they fail during execution

3.kill and resubmit job if they reach the queue-time-ceiling threshold (jobs stuck in the queue)

4.if the batch submission is larger than about 1000 jobs, I recommend that a rate-limiter is also implemented, so that not all jobs are submitted to the Grid together at the beginning.

5.can be bundled with a timed garbage collection system for input files that might have been uploaded to SEs (see *"Space, bandwidth and time optimizations"*)

After having set up your automatic submitter, you will need to start tweaking the submission parameters for optimizing your Grid submissions.

While setting your submission parameters please keep in mind that you should keep a fair behaviour to the other Grid users. Do not submit massive amounts of jobs together and do not submit more than 500 parallel jobs for weeks without first asking authorization to the Responsible Person for your VO.

7.4 Publications and oral presentations

My 3-years Ph.D. work described in this thesis has originated various publications in journals and conferences proceedings. In addition, I personally presented my work as a speaker in numerous international conferences.

The details are in the following subsections.

7.4.1 Publications in journals and conferences proceedings

My Ph.D. work originated the following publications in journals and conferences proceedings:

Journal publication (first author)	Gabriele A Trombetti, Ivan Merelli, Luciano Milanesi – **High Performance cDNA Sequence Analysis Using Grid Technology** – *Journal of Parallel and Distributed Computing 2006*, **66**(12):1482-8

Journal publication (first author)	Trombetti GA, Bonnal RJP, Rizzi E, De Bellis G, Milanesi L – **Data handling strategies for high throughput pyrosequencers** – *BMC Bioinformatics BMC Bioinformatics 2007*, **8**(Suppl 1):S22 (Impact Factor 4.96 at the time of acceptance)
Journal publication (first author)	Trombetti, GA and Merelli, I. and Orro, A. and Milanesi, L. – **BGBlast: A BLAST Grid Implementation with Database Self-Updating and Adaptive Replication** – *Stud Health Technol Inform 2007*, **126:**23-30
Proceedings	L. Milanesi et al. – **BioinfoGRID: Bioinformatics GRID based applications overview** – *NETTAB2006 proceedings*
Proceedings	Milanesi L, Andreas G, Arlandini C, Beltrame F, Bishop C, Breton V, Ernest P, Jacq N, Legre Y, Liò P, Liuni S, Mazzuccato M, Maggi G, Meloni G, Merelli I, Morra G, Orro A, Porro I, Sanger M, Shuai S, Trombetti G – **BioinfoGRID: Bioinformatics Grid Application for life science** – *BITS2006 proceedings*
Proceedings	Alessandro Orro, Ivan Merelli, Gabriele Trombetti, Luciano Milanesi – **Enabling Post Processing Data Extraction in Grid Environment** – *NETTAB2006 proceedings*
Proceedings (first author)	Gabriele Trombetti, Alessandro Orro, Ivan Merelli, Luciano Milanesi – **FreshBlast: A Blast Grid Implementation with Database self-Updating and Adaptive Replication** – *NETTAB2006 proceedings*

7.4.2 Oral presentations in conferences

I personally presented my work in the following international (except INFN workshop which is national) conferences related to Grid Computing and/or Bioinformatics:

December 18-20th, 2006	INFN Workshop, Padova, Italy
April 24-27th, 2007	HealthGrid Conference, Geneva, Switzerland
May 9-11th, 2007	EGEE User Forum / 20th Open Grid Forum, Manchester, UK
May 14-19th, 2007	Biomed Grid School, Varenna, Italy
October 1-5th, 2007	EGEE'07 Conference, Budapest, Hungary
December 10-13th, 2007	BioinfoGRID Symposium 2007

8 Acknowledgement

A large part of the research work carried on during my My Ph.D. course in collaboration with CNR-ITB, Milan was financially supported by the Italian FIRB-MIUR projects “Italian Laboratory for Bioinformatics Technologies – LITBIO” [LITBIO] and ITALBIONET and by the European BioinfoGRID Specific Support Action [BIOINFOGRID] and the EGEE [EGEE] projects.

9 Bibliography - References

[ALIGNMENTSCORE]	http://searchlauncher.bcm.tmc.edu/help/AlignmentScore.html
[ALTSCHUL'90]	Altschul S. F. , Gish W., Miller W., Myers E. W. and Lipman D. J.: **Basic Local Alignment Search Tool** – *J. Mol. Biol. 1990*, **215:**403–410
[BAYER'04]	Micha Bayer, Aileen Campbell, Davy Virdee: **A GT3 based BLAST grid service for biomedical research** – *Proceedings of the UK e–Science All Hands Meeting 2004*
[BEADSTATION]	http://icom.illumina.com/products/systems/beadstation500g.ilmn
[BELSHAW'05]	R. Belshaw, A. Katzourakis: **BlastAlign: a program that uses blast to align problematic nucleotide sequences** – *Bioinformatics 2005*, **21:**122–3
[BIOINFOGRID]	http://www.bioinfogrid.eu
[BLAST]	http://www.ncbi.nlm.nih.gov/BLAST/
[DARLING'03]	A. Darling, L. Carey, and W. Feng: **The Design, Implementation, and Evaluation of mpiBLAST** – *4th International Conference on Linux Clusters; San Jose, CA, June 2003*
[DIPTESTING]	http://www.diveintopython.org/unit_testing/
[EGEE]	EGEE – Enabling Grids for E–SciencE: http://public.eu–egee.org/
[FOSTER'01]	I. Foster, C. Kesselman, S. Tuecke: **The Anatomy of the Grid: Enabling Scalable Virtual Organizations** – *International J. Supercomputer Applications 2001*, **15**(3)

[FOSTER'02]	I. Foster, C. Kesselman, J. Nick, S. Tuecke: **The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration** – *Open Grid Service Infrastructure WG, Proc. Global Grid Forum 2002*
[GARCIA'00]	C.A. Garcia, A. Ahmadian, B. Gharizadeh, J. Lundeberg, M. Ronaghi, P. Nyren: **Mutation detection by pyrosequencing: sequencing of exons 5–8 of the p53 tumor suppressor gene** – *Gene 2000*, **253**(2):249–257
[GRIDIT]	http://grid–it.cnaf.infn.it/
[HGP]	http://www.genome.gov/10001772
[HMM1]	http://en.wikipedia.org/wiki/Hidden_Markov_model
[HMM2]	http://jedlik.phy.bme.hu/~gerjanos/HMM/node2.html
[KARLIN'93]	S. Karlin, S. Altschul: **Applications and Statistics for Multiple High–Scoring Segments in Molecular Sequences** – *Proc. Natl. Acad. Sci. USA 1993*, **90:**5873–7
[KENT'02]	Kent WJ: **BLAT - the BLAST-like alignment tool** – *Genome Res 2002*, **12:**656–64
[KONISHI'03]	Konishi, F. and Shiroto, Y. and Umetsu, R. and Konagaya, A.: **Scalable BLAST Service in OBIGrid Environment** – *Genome Informatics 2003*, **14:**535–6
[LITBIO]	http://www.litbio.org
[MA'02]	Ma, B. and Tromp, J. and Li, M.: **PatternHunter: faster and more sensitive homology search** – *Bioinformatics 2002*, **18**(3):440–5

[Marsh'05] S. Marsh, C.R. King, A.A. Garsa, H.L. McLeod: **Pyrosequencing of clinically relevant polymorphisms** – *Methods Mol Biol 2005*, **311:**97–114.

[Mathog'03] D.R. Mathog: **Parallel BLAST on split databases** – *Bioinformatics Applications Note 2003*; **19**(14):1865–6

[megablast] http://www.ncbi.nlm.nih.gov/blast/megablast.html

[Merelli'05] I. Merelli, L. Milanesi: **High performance implementation of BLAST using GRID technology** – *Proc. BITS 2005, p. 59*

[Merelli'05–II] I. Merelli, M. Landenna, L. Milanesi: **Biological database access and integration using Web Services in GRID technology** – *Proc. Bits 2005, p. 60*

[Parsons'92] J.D. Parsons, S. Brenner, M.J. Bishop: **Clustering cDNA sequences** – *Comput. Appl. Biosci. 1992*, **8:**461–466

[pbs] http://en.wikipedia.org/wiki/Portable_Batch_System

[Qi'05] Qi Y, Lin F.: **Parallelisation of the blast algorithm** – *Cell Mol Biol Lett. 2005*; **10**(2):281–5

[regressiontesting] http://en.wikipedia.org/wiki/Regression_testing

[Ronaghi'98] M. Ronaghi, M. Uhlen M, P. Nyren: **A sequencing method based on real–time pyrophosphate** – *Science 1998*, **281**(5375):363–365

[Smith'81] Temple F. Smith and Michael S. Waterman: **Identification of Common Molecular Subsequences** – *Journal of Molecular Biology 1981*; **147:**195–197

[testcase] http://en.wikipedia.org/wiki/Test_case

[TESTDRIVEN] http://www.ibm.com/developerworks/java/library/j–xp042203/

[TROMBETTI'06] Trombetti G.A., Merelli I. and Milanesi L.: **High performance cDNA sequence analysis using grid technology** – *Journal of Parallel and Distributed Computing 2006*, **66**(12):1482–8

[TROMBETTI'07] Trombetti, GA and Merelli, I. and Orro, A. and Milanesi, L.: **BGBlast: A BLAST Grid Implementation with Database Self-Updating and Adaptive Replication** – *Stud Health Technol Inform 2007*, **126:**23–30

[UNITTESTING] http://en.wikipedia.org/wiki/Unit_testing

[WIEMANN'01] S. Wiemann , B. Weil ,R. Wellenreuther, J. Gassenhuber, S. Glassl, W. Ansorge, M. Bocher, H. Blocker, S. Bauersachs, H. Blum, J. Lauber, A. Dusterhoft, A. Beyer, K. Kohrer, N. Strack, H.W. Mewes, B. Ottenwalder, B. Obermaier, J. Tampe, D. Heubner, R. Wambutt, B. Korn, M. Klein, A. Poustka: **Toward a catalog of human genes and proteins: sequencing and analysis of 500 novel complete protein coding human cDNAs** – *Genome Res. 2001*, **11**(3):422–435

[XP] http://ootips.org/xp.html

[ZHANG'00] Zhang, Z., Schwartz, S., Wagner, L. and Miller, W.: **A greedy algorithm for aligning DNA sequences** – *J. Comput. Biol. 2000*, **7:** 203–214