EMBnet.news

Volume 14 Nr. 2
July 2008

- ChemGPS-NP web
- ASPicDB: A database resource for alternative splicing analysis
- Grid tutorials

and more ...

Editorial

Here is another issue of EMBnet.news with plenty to read. We start with a report on the third course in Kenya, followed with a sum-up of the activities of the ASBCB, that shows their activity in Africa and their connections with the rest of the world. Then we have an article on a very interesting tool that allows one to navigate in biologically relevant chemical space. A report on the new FP7 funded LUPA project reveals how interesting it can be to study human genetic disorders using the dog as a model system. The ASPicDB, a database of alternative splice patterns in human and mouse genes, with monthly updates. A series of papers is actually a tutorial on GRID usage for newcomers, topped by an article on effective large scale multitasking on a GRID environment. The next issue is already in preparation but will only be out after our AGM and 20th anniversary celebrations in Bari, Italy, in September. As usual we invite our readers to contribute to EMBnet.news and to encourage an even greater number of colleagues to do so.

The editorial board: Erik Bongcam-Rudloff, Domenica D'Elia, Pedro Fernandes, Kimmo Mattila and Lubos Klucar.

Protein Spotlight (ISSN 1424-4721) is a periodical electronic review from the SWISS-PROT group of the Swiss Institute of Bioinformatics (SIB). It is published on a monthly basis and consists of articles focused on particular proteins of interest. Each issue is available, free of charge, in HTML or PDF format at

`http://www.expasy.org/spotlight`

We provide the EMBnet community with a printed version of issue 94. Please let us know if you like this inclusion.

Cover picture: *Kniphofia uvari*, 'Red hot poker'. Nairobi, 2008 [© Erik Bongcam-Rudloff]

Contents

Editorial Board:

Erik Bongcam-Rudloff, The Linnaeus Centre for Bioinformatics, SLU/UU. SE
Email: erik.bongcam@bmc.uu.se
Tel: +46-18-4716696
Fax: +46-18-4714525

Domenica D'Elia, Institute for Biomedical Technologies - CNR, Bari, IT
Email: domenica.delia@ba.itb.cnr.it
Tel: +39-80-5929674
Fax: +39-80-5929690

Pedro Fernandes, Instituto Gulbenkian. PT
Email: pfern@igc.gulbenkian.pt
Tel: +315-214407912
Fax: +315-214407970

Lubos Klucar, Institute of Molecular Biology, SAS Bratislava, SK
Email: klucar@embnet.sk
Tel: +421-2-59307413
Fax: +421-2-59307416

Kimmo Mattila, CSC, Espoo, FI
Email: kimmo.mattila@csc.fi
Tel: +358-9-4572708
Fax: +358-9-4572302

New tools for bioinformatics teaching

Maria Wilbe and Erik Bongcam-Rudloff

Department of Animal Breeding and Genetics, SLU, Sweden

The course team: Erik Lagercrantz, Maria Wilbe, Erik Bongcam-Rudloff, Alvaro Martinez Barrio, Etienne de Villiers and Saidimu Apale (not all in the picture)

For the third year the ILRI/BecA EMBNet node organized an introductory course in Bioinformatics at the International Livestock Research Institute (ILRI) campus in Nairobi, Kenya. The objective was to introduce young scientists from east and central Africa to use bioinformatics/computational biology in their research and to present some of the biological resources available on the ILRI-BECA bioinformatics platform.
The course is organized in collaboration with the Swedish University of Agricultural Sciences (SLU), Uppsala University (UU) and Linnaeus Centre for Bioinformatics (LCB) funded by SIDA.

The course was 9 days long, from May 5 to 15, 2008. 24 participants from Uganda, Sudan, Tanzania, Burundi, Somalia, Cameroon, Ethiopia and Kenya attended the course. Futhermore

The course participants at ILRI, Kenya, May 9, 2008.

the lectures were recorded on DVD and will be used at the University of Buea, Cameroon and Maseno University in Kenya.

Topics covered during the course were: sequence analysis and alignments, EMBOSS/wEMBOSS, Staden package, Unix/Perl basics, genomics and comparative genomics, Artemis and Artemis Comparison Tool and Ensembl.

Many course participants are learning new tools and techniques that will be useful for them when returning to their own laboratories. Last year each student received a Bioinformatics Live-CD with all programs used during the course, but it was not possible for them to easily save their work. The course team came therefore up with a new idea that resulted on an "Bioinformatics workbench on a USB memory stick", **eBioUSB**. Each student received one USB-stick to use during the course and bring back home for continuous work. By us-

Saidimu Apale filming the lectures

ing an USB the students were also able to save all their work directly on the stick.

The eBioUSB stick is based on a Linux environment: the African UBUNTU system. The USB solution has the advantage that it can be used in any computer without changing any settings. It has a complete Desktop environment (e.g. office package, text editor and mail client).

The programs that were necessary for the course and installed on eBioUSB were:

- ClustalW
- ClustalX
- NCBI BLAST
- NCBI Tools
- EMBOSS
- wEMBOSS
- Staden
- PHYLIP
- Artemis
- Artemis comparison Tool
- JalView
- BioPerl

We have created 2 different USB sticks: one that can be ran on old computers and the one used for the course can be used on newer machines resulting on a higher speed system.

Future improvements of the eBioUSB include incorporating the SwissProt database for running most common analysis locally.

The "**BioMacKit**" a Bioinformatics Portable Teaching Kit that was created last year was used this time to access a complete mirror of various programs and databases used during the course (More information: EMBnet.news, Vol 13 Nr 2:7, 2007).

The course evaluation resulted on an average 4.9 of maximum 5 points encouraging the teachers to organize a new course next year.

Student comments of the course:

"Interesting and highly applicable to our work though not so easy for beginners"

The eBioUSB used for the course.

"I enjoy working with the emboss software because it has many programs and each has a manual, thus quite user friendly"

"I look forward to applying the knowledge I have acquired in the training in my research work and to assist others"

"The USB stick enables me to reproduce exactly what I have learned"

"I learned a lot about databases that were available to me and programs I could use to analyze protein and DNA sequences"

Introducing the African Society for Bioinformatics and Computational Biology (ASBCB; http://www.asbcb.org)

Daniel Masiga

(icipe, Kenya)

The ASBCB was established during a workshop at the South African National Bioinformatics Institute (SANBI) in 2004. The society has a vision to promote the exchange of ideas, infrastructure and resources in the fields of bioinformatics and computational biology and facilitate the interaction and collaboration among scientists and educators and to measurably advance the awareness and understanding of the science of bioinformatics and computational biology in Africa. The society represents the bioinformatics and computational biology community in Africa and aims to be the most respected and reliable international non-profit organization representing this community.

The mission of the society is to be a scholarly body dedicated to advancing, developing and promoting bioinformatics and computational biology in Africa, while serving a global membership, by impacting government and scientific policies, providing high quality publications and meetings, and through distribution of valuable information about training, education, employment and relevant news from related fields.

In seeking to have an impact in more than 50 countries with at least 4 major international languages and thousands of others, with a considerably varied educational and development landscape, we are confronted with a huge task. Yet in order to develop the application of bioinformatics in Africa we must work in collaboration with others who have walked this road before, or are on the same path. As of May 16, 2008, ASBCB had 268 members from 36 countries (Argentina, Burkina Faso, Cameroon, Canada, Cote d'Ivoire, Cuba, Democratic Republic of Congo, Egypt, Ethiopia, Finland, France, Gabon, Gambia, Germany, Ghana, India, Iran, Kenya, Malaysia, Mali, Mauritania, Mauritius, Nepal, Nigeria, Rwanda, Scotland, Singapore, South Africa, Sudan, Switzerland, Tanzania, Tunisia, Uganda, United Kingdom, United States).

As a society, we know that building educational and research capacity among our members is crucial to meeting our objectives. We therefore invite collaboration in addressing our objectives, which are:

1. identify, establish and promote opportunities for networking;
2. facilitate access to bioinformatics and computational biology infrastructure;
3. encourage and develop bioinformatics and computational biology nodes;
4. increase awareness and promote the use of bioinformatics and computational biology;
5. promote bioinformatics and computational biology education.

In May 2007, the ASBCB held its first conference, "Bioinformatics of pathogens and disease vectors" in Nairobi, co-hosted by the International Centre of Insect Physiology and Ecology (icipe) together with the International Livestock Research Institute (ILRI) and the Centre National de la Recherche Scientifique (CNRS, LIRMM Montpellier, France). One of the outcomes of this conference has been the growth of a very vibrant students community, the regional students group of the ISCB student council. From among these, we hope will emerge a core of African scientists who will help to drive the translation of genome resources using bioinformatics applications to improve the wellbeing of Africa's people in various spheres.

The society was formed during a bioinformatics workshop supported by the WHO Special Programme for Research and Training in Tropical Diseases (TDR). They have since held other

courses covering pathogens and their vectors (principally malaria and sleeping sickness) in Mali (MRTC) and South Africa (SANBI). We appreciate this support and look forward to continuing to partner with them. We also understand that we need strategies for long-term and sustained generation of scientists with competence in bioinformatics in Africa, through education and research. To achieve this, strong partnerships at different levels will need to be established. We look forward to future collaborations.







The Governing Council of ASBCB (since June 2007): President: Daniel Masiga (icipe, Kenya); Vice-President: Ezekiel Adebiyi (Covenant University, Nigeria); Secretary: Nicky Mulder (University of Cape Town, South Africa); Treasurer: Alia Benkahla (Pasteur Institute of Tunis, Tunisia); Committee Members: Jaco de Ridder (University of Pretoria, South Africa), Seydou Doumbia (University of Bamako, Mali); and the Newsletter Editor: Beatrice Kilel (Washington DC, USA).

ChemGPS-NP$_{web}$

A tool tuned for navigation in biologically relevant chemical space




Josefin Rosén[1], Anders Lövgren[2], Johan Gottfries[3], Anders Backlund[4]

[1] Div. of Pharmacognosy, Dept. of Medicinal Chemistry, BMC Box 574, S-751 23 Uppsala, Sweden
[2] The IT-/Computing Dept., BMC Box 570, S-751 23 Uppsala, Sweden
[3] Pharmnovo Inc., Sahlgrenska Science Park, S-413 46 Gothenburg, Sweden
[4] Div. of Pharmacognosy, Dept. of Medicinal Chemistry, BMC Box 574, S-751 23 Uppsala, Sweden

Introduction

The World Wide Web has become a central source for information, education, tools, and services that make life easier for medicinal chemists and drug discoverers. Internet technology offers an exceptional possibility to develop public tools. We have developed a web-based public tool ChemGPS-NP$_{web}$,(`http://chemgps.bmc.uu.se/`), for comprehensive chemical space navigation and exploration in terms of global mapping on to a consistent 8-dimensional map of structural characteristics. ChemGPS-NP [1, 2] is a principal component analysis (PCA) based global space map or a chemical global positioning system [3]. Compounds of interest or under study are positioned onto this map using interpolation in terms of PCA score prediction. The properties of the compounds together with trends and groupings can easily be interpreted from the resulting projections. In this article we review design, features, and proposed fields of application of ChemGPS-NP$_{web}$.

Technical details

General

ChemGPS-NP$_{web}$ includes a number of different programs and libraries that interact with each other according to the traditional UNIX-model. Each element performs a well defined task and together they solve a more advanced problem.

The system includes three main elements: DragonX [4], for calculation of molecular descriptors, Simca-QP [5], for multivariate predictions, and the web interface (Batchelor). Further more a batch queue manager is used. This allows jobs with long run times to be submitted to the web server and scheduled for later execution by its batch queue. The programs exchange information with the web interface by storing information in the file system, which acts as the database

Job flow

When the job queue starts a job, the following things occur: the uploaded SMILES [6] strings are processed by DragonX, and the obtained data are then transformed by a Perl script that organizes the values of the 35 descriptors used by the model. These transformed results are used as indata to cgpsclt (client) that connects to cgpsd (server) to start the multivariate prediction. Subsequently Simca-QP performs the prediction via libchemgps and cgpsd sends the result back to cgpsclt, which stores the result in the database. If cgpsd (the server) is not available the prediction will instead be performed locally by cgpsstd (standalone program). Figure 1 describes how the different elements interact.

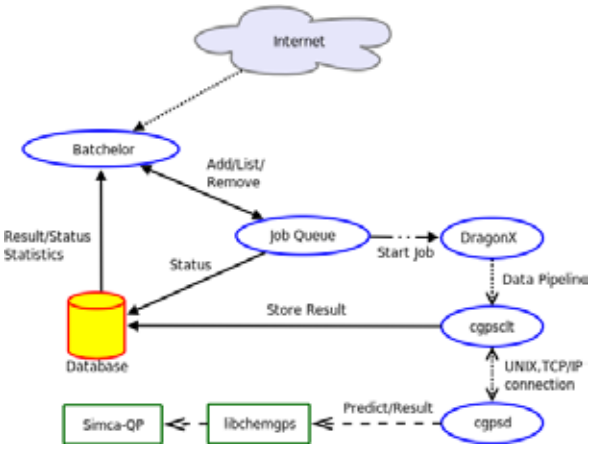

Figure 1. Flowchart describing the interaction between the different elements of ChemGPS-NP$_{web}$.

Optimizations

The extra step with client/server (cgpsclt/cgpsd) was incorporated to avoid having to load the project (reference set) for each job. As an additional benefit it also enables predictions to be performed by one or more computers on the network.

All elements (DragonX, Simca-QP, and cgpsd) are multithreaded, which becomes more and more important as the number of cores (CPUs) will increase in the future.

The web interface

The web interface (Batchelor) enables the upload of data (SMILES) and to obtain the results from the runs. The job queue can be filtered and sorted according to different criteria. Uploaded data and results are personal and can only be reached from the same computer as the job was initiated from. The information presented to the user is in part obtained from the database (results and statistics), or directly from the job queue (job status).

System information

The entire process runs at present on one single computer, a 64 bit 2 x Quad Core Xeon operating at 1.6 GHz with 4 GB RAM, and featuring a GNU/Linux operating system.

How to use ChemGPS-NP$_{web}$

The simple instruction for using ChemGPS-NP$_{web}$ is as follows:

a correct SMILES-file [6] with a maximum of 500 compounds (or 1024 kB) is uploaded and submitted using the buttons 'Browse'… and 'Send File' (figure 2). Any IDs in the file should be placed after (to the right of) the SMILES string. Alternatively the SMILES can be pasted in the 'Process data' drop box and submitted by clicking 'Send Data'.

The resulting ChemGPS-NP 8D coordinates are obtained through 'View results' in the left menu. Users (submitters) can monitor the state of their submitted jobs (pending, running or finished) and later download the result from the queue view (figure 3). The coordinates (figure 4) can then be plotted using preferred software. Here we have used Grapher 2.0 distributed together

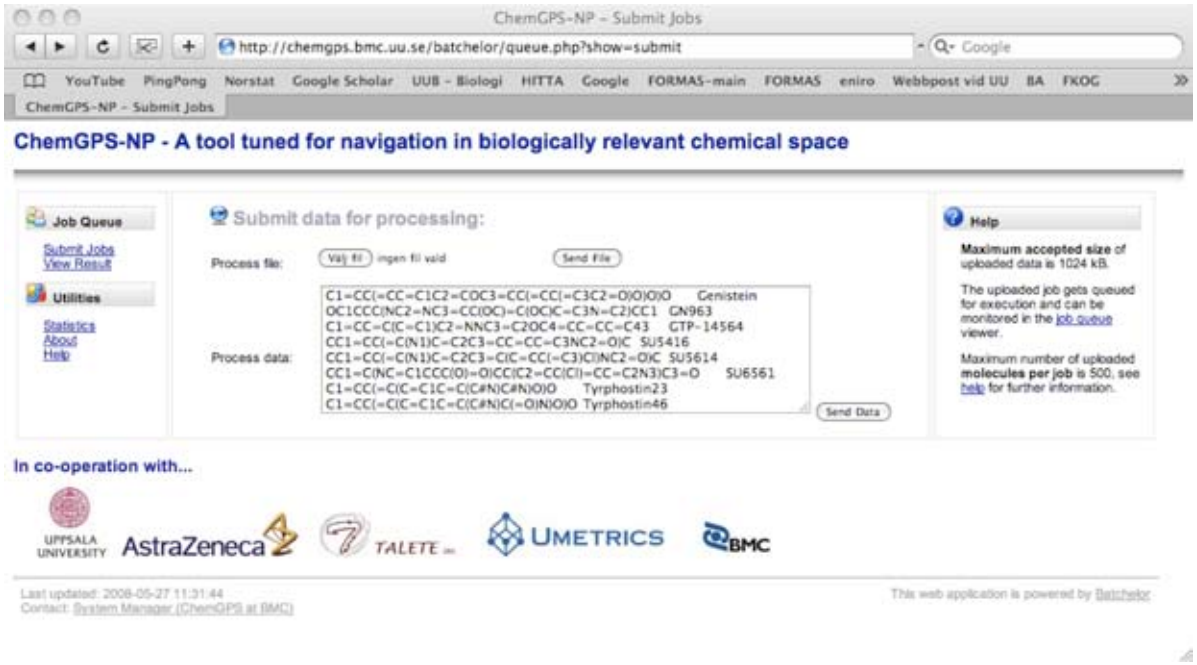

Figure 2. User upload interface of ChemGPS-NP$_{web}$.

with MacOS X (figure 5). From the plot it is evident that the two compounds indicated by the circle have very similar physical chemical properties as can be confirmed by the chemical structures displayed in figure 6.

Additionally post computational statistics are prepared based on results from each of the successive computational steps, and can be viewed by clicking 'Statistics' in the left menu (figure 2).

Final remarkss

The drug discovery process is today held back by increasing costs and high attrition-rates, with an overall decrease in the number of annually registered new chemical entities. Considering the immensity of chemical space, which is estimated to exceed 1060 possible compounds when only small carbon-based compounds are considered [7], it is obvious that the process of compound selection and prioritization is crucial. An efficient selection process would give a higher probability of obtaining a lead compound. ChemGPS-NP provides a framework for making compound comparison and selection more efficient, thereby increasing probability of

Figure 3. Queue feedback interface, allowing simple error tracing features as well as access to retrieved data both directly and as a downloadable file.

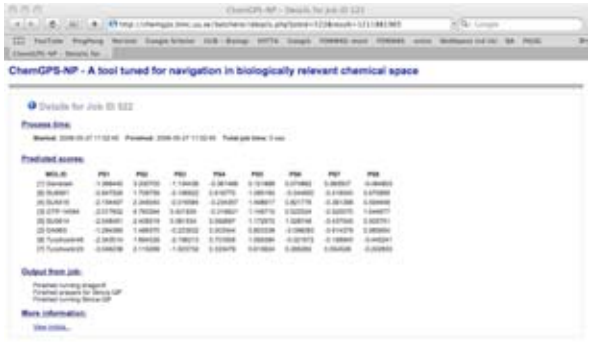

Figure 4. Resulting coordinates for submitted compounds retrieved through direct access.

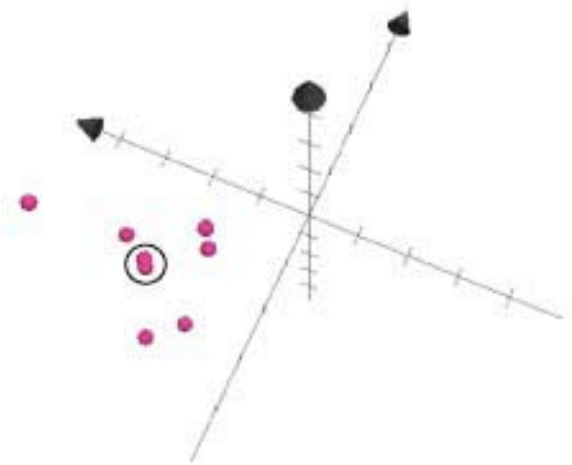

Figure 5. Retrieved data plotted using Grapher 2.0, showing first three of eight dimensions of chemical space as defined by ChemGPS-NP. From the plot it is obvious that two compounds indicated by the circle have very similar physical chemical properties.

hit generation in the search for novel bioactive molecules. The benefits of ChemGPS-NP are, in one way, comparable to the possibilities opened in molecular biology by rigorous application of the BLAST algorithms [8]. These allow, for example through web-interfaces, the research community to easily compare sections of nucleotide or amino-acid sequences for homology searching, identifying genes, or preparing datasets for phylogenetic analyses, all in huge datasets.

In summary we have developed an internet tool for chemical space navigation.

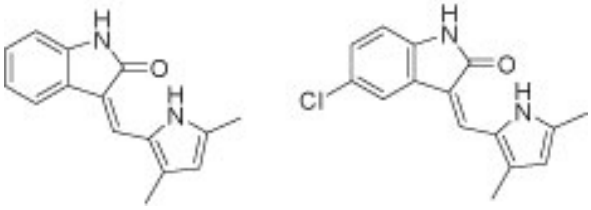

Figure 6. Molecular structures (drawn with ChemDraw Ultra 11.0.1) of the two apparently similar compounds encircled in figure 5.

ChemGPS-NP$_{web}$ can assist in for instance compound selection and prioritization; property description and interpretation; clustering overviews; as well as comparison and characterization of large datasets. ChemGPS-NP$_{web}$ so far includes several pieces of commercial software that connect via scripts handling input, queue and output of data files. The output files can be piped into other software for post-processing, plotting and visualization.

Acknowledgements

Instrumental at initial stages in implementing the ChemGPS-NP$_{web}$ were Thierry Kogej at AstraZeneca R&D, Mölndal, and Gustavo Gonzales-Wall and Nils-Einar Eriksson at the IT-/Computing Dept. at BMC. The authors are grateful for software support from UMETRICS and TALETE.

References

1. Larsson J, Gottfries J, Bohlin L & Backlund A (2005) Expanding the ChemGPS chemical space with natural products. J Nat Prod 68: 985-991.
2. Larsson J, Gottfries J, Muresan S & Backlund A (2007) ChemGPS-NP: tuned for navigation in biologically relevant chemical space. J Nat Prod 70: 789-794.
3. Oprea T I & Gottfries J (2001) Chemography: the art of navigating in chemical space. J Comb Chem 3: 157-166.
4. Talete srl, DragonX (Software for Molecular Descriptor Calculations). Linux version - 2007 - http://www.talete.mi.it/. Accessed May 28, 2008.
5. SIMCA-QP software, Umetrics AB, Umeå, Sweden. http://www.umetrics.com/. Accessed May 28, 2008.
6. Weininger D (1988) SMILES, a chemical language and infromations system. 1. Introduction to methodology and encoding rules. J Chem Inf Comput Sci 28: 31-36.
7. Bohacek R S, McMartin C & Guida W C (1996) The art and practice of structure-based drug design: a molecular modeling perspective. Med Res Rev 16: 3-50.
8. Altschul S F, Gish W, Miller W, Myers E W & Lipman D J (1990) Basic local alignment search tool. J Mol Biol 215: 403-410.

Bovine Trace
To the traceability of beef

Claudia P. Mendoza[1] and Allan Orozco[2]

[1] Agricultural Engineering, Bioinformatics. EARTH University, Las Mercedes de Guácimo, Limón, Costa Rica. www.earth.ac.cr
[2] Visiting Professor, EMBnet Costa Rica.

Accessibility to US and EU markets is invaluable for any beef industry, however, in order to enter these markets, bovine producers must provide a mechanism to guarantee traceability of their products. We have analysed the problem to propose a method and accompanying technology for the traceability of beef.

Building on standard technologies of the bovine industry we have developed a system which generates special codes that allow consumers to trace the origin of bovine products through all the steps back to the producer using either bar code technology (EAN.UCC-13/128) or manual input.

Introduction

The relevance of US and EU markets for the economy of bovine industry can not be understated. However, access to these markets requires a guarantee of traceability of bovine products through all the chain from production through delivery to consumer, a requirement that is a major obstacle for many producers in the world. Recognizing that this poses a serious economical disadvantage for many producers we set out to develop a tracing system that can address their needs.

In order to build our system we have had to analyse not only the technology available but also the impact of legal regulations on the process

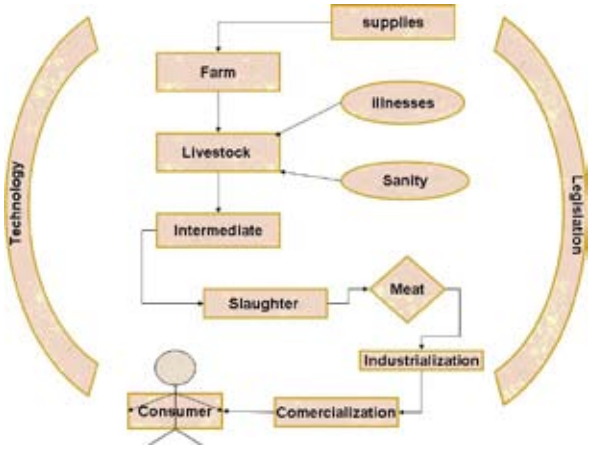

Figure 1. UML Conceptual Model.

and the complex interactions across the long bovine delivery chain. Building on theoretical basis and field experiences we used the Unified Software Development Process to build an UML conceptual model (see Fig. 1) leading to an object oriented computational model that was iteratively refined running simulations with sample data until it closely matched the real biological system.

Features

The requested features of a beef traceability system involve the ability to use user supplied information to trace (investigate and verify) such varied issues as cattle origins (and theft), sanitary control, verifiable cattle inventory or follow up of all steps in the supply chain. This additional information provides better guarantees for consumers, producers and food processors, delivering significant value to all participants in the food chain.

Figure 2. Animal registration.

Figure 4. Commercial Sticker.

In order to support this traceability we have designed a set of specialized interfaces, each aimed at a different player in the chain, who must provide the relevant data for the system at his/her own stage. These include interfaces for animal identification using genetic, genomic and nanotechnology (Scanning Tunneling Microscopy/Atomic Force Microscopy) markers, slaughter registration, sanitary control, etc. down to the final generation of sticker labels for sellers and trace queries by consumers (Figs. 2, 3, 4 and 5).

The interfaces allow producers and intermediate players to enter useful and legally required information, as well as to use this for their own

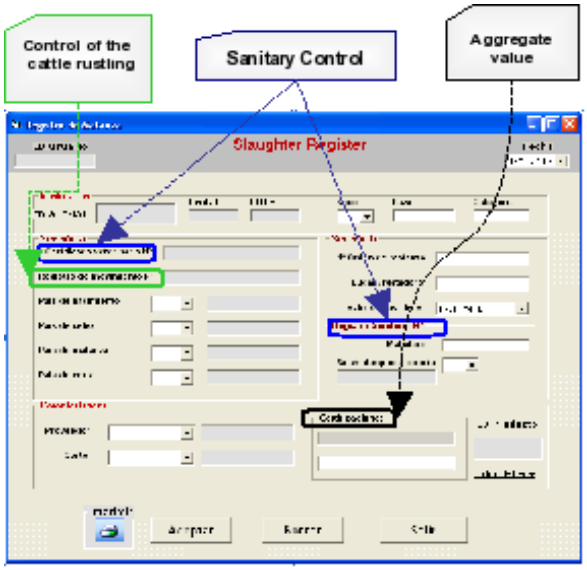

Figure 3. Registration of Slaughter.

Figure 5. Molecular mark and genomic identification.

records, and allow consumers to use the bar and numeric codes in the product label to access information about the quality and characteristics of a product tracing it through any step in the production-delivery-commercialization chain.

Conclusion

Guaranteeing beef traceability is a must have for the bovine industry worldwide. Providing this service requires a complex infrastructure that involves all players in the process, from the original producer to the end consumer and has deep legal and technological implications. We have developed a system to support modern tracing capabilities for the bovine industry and consumers. This system has been developed with relevant involvement from industry stakeholders and though closed source, it is the result of academic research to establish the basis for traceability of meat products at an industrial level.

Acknowledgements

This work is the result of a 2007 graduation project in Agricultural Engineering (Bioinformatics) at EARTH University that has received a honours degree and counted with assistance from two Nobel Prize Winners. (Dr. Yunus, Dr. Arias).

References

1. Grupo europeo de expertos en productos cárnicos (2002). EMEG. GS1. The Global language of Business. Aplicación de estándares EAN/UCC. 2002: 27-30.
2. Sandoval, Alejandro. Trazabilidad en Estados Unidos y Europa. FRIMA, SA.
3. Jacobson, I. (2000). El proceso unificado de desarrollo software. Booch, G. Madrid. Pearson Educación. 207-253.

LUPA: Unravelling the molecular basis of common complex human disorders using the dog as a model system

Anne-Sophie Lequarré

Coordinator of the LUPA project, Animal Genomics Unit, GIGA-Research, Université de Liège, Belgium

Understanding the pathogenic mechanisms of common human diseases - including cancer, cardiovascular, inflammatory and allergic disorders - is an important objective of ongoing genomics initiatives. Gaining molecular insights into the cellular processes disturbed in these pathologies is a direct path towards improved prevention, diagnosis and treatment. As genetic predisposition is a major risk factor for most common diseases, identifying predisposing gene variants is a promising strategy to achieve these objectives.

Despite major efforts and a few successes, identifying susceptibility genes for common diseases in human has so far proven to be difficult. This is likely due to the complexity of the underlying causes, including genetic and environmental heterogeneity, gene-by-gene and gene-by-environment interactions. Very large disease cohorts genotyped for hundreds of thousands of SNP markers are likely to be required to achieve satisfactory power. In order to accelerate the discovery pace, studies targeting human populations are advantageously complemented by studies of more tractable animal models of human disease. Until recently, the only widely used model organisms for that purpose were the mouse and rat but the dog has a number of unique features that have the potential to make it a superior genetic system to study the molecular basis of disease.

The dog: an ideal model system to unravel the molecular basis of common diseases

The dog population is composed of ~ 400 purebred breeds. Each of these is a genetic isolate with unique characteristics resulting either from persistent selection for desired attributes (e.g. size, morphology, coat colour and behaviour) or from genetic drift/inbreeding (e.g. susceptibility to specific diseases). Not a single other organism approaches the level of phenotypic variation that is observed amongst dog breeds [1, 2, 3]. This diversity includes breed-specific susceptibility to disease, many of which are the equivalents of common human disorders. Exposed to the same "westernized" environment as their owners, dogs suffer from the same range of diseases that plague our developed societies [4]. Over 200 genetic diseases have been reported including cancer, diabetes, inflammatory diseases, heart disease and epilepsy.

In addition, it has been recently demonstrated that the demographic history of dog breeds results in a number of features that make them uniquely suited for the detection of susceptibility genes [5]. These include (i) the fact that the genetic complexity of inherited diseases is bound to be considerably reduced within dog breeds when compared to humans, and (ii) the fact that – as a result of long-range linkage disequilibrium – the number of SNP markers needed to effectively perform whole genome scans is reduced by an order of magnitude from hundreds to tens of thousands of markers. A two-stage approach, combining within-breed and subsequent between-breed analyses, has recently been proposed for very precise localization of genetic risk factors [6].

As a result of these features, the dog has attracted a lot of attention amongst the human genetics community. The dog genome was the 4th mammalian genome to be completely sequenced by the NIH. Millions of SNP markers have been discovered [5] and utilized to assemble highly informative genome wide SNP panels that can be genotyped cost-effectively using high throughput platforms.

Proof of principle

To demonstrate the value of the dog system, Dr. Lindblad-Toh and Dr. Andersson [7] have performed pilot experiments targeting two mendelian traits, white spotting (sw), inherited in a semi-dominant manner in boxers and bull ter-

LUPA - logo

riers, and ridging, a characteristic dorsal band of abnormally oriented hair follicles from which the Rhodesian ridgeback takes its name. In a very short time and using a quite limited number of dogs they identified the gene for white coat color (MITF) then using the two-staged "between-breed" approach they positioned the mutation in a region immediately upstream of the transcriptional start site of the melanocyte-specific (M) promoter of MITF. They also identified the precise chromosomal localization and a candidate mutation for dermoid sinus. The genetic analysis of the defect revealed a 133-kb duplication that includes genes coding for three fibroblast growth factors (FGFs), suggesting that an increased gene dosage of one or more of these paracrine signalling molecules causes the dorsal hair ridge.

The LUPA project

Named after the legendary wolf who nourished the founders of Rome, the LUPA Consortium is a coordinated European effort taking advantage of this extraordinary genetic model system. The project funded by the 7th FP of the European Commission started last January. The Consortium blends highly qualified clinicians with all European research teams specialized in dog genetics. Twenty veterinary clinics in 12 European countries are working together to collect DNA samples from large cohorts of dogs suffering from a range of thoroughly defined diseases of relevance to human health, including cancers, inflammatory disorders, heart diseases and epilepsies. The European dimension and the establishment of protocols for standardized, high quality clinical characterization will allow the network to establish sample collections unique in the world. Once the cohorts are built, DNA samples are sent to a centralized, high-throughput SNP genotyping facility. The SNP genotypes stored in central database are made available to participating collaborating centres, who analyse the data with the support of dedicated statistical genetics platforms. Following genome wide association and fine-mapping the candidate genes will be followed up at the molecular level by expert animal and human genomics centres.

LUPA is expected to be very successful in providing insights into the pathogenesis of common human diseases – its primary goal. In addition, the project has the potential to have a major impact on the future of veterinary medicine in Europe. It will give a new impulse to veterinary clinicians across Europe towards the potentials offered by genomics in diagnosis and treatment.

Web site: `http://www.eurolupa.org/`

Information: `as.lequarre@ulg.ac.be`

References:

1. Neff MW, Rine J. A fetching model organism. Cell. 2006.124(2):229-31. Review.
2. Ostrander EA, Wayne RK. The canine genome. Genome Res. 2005. (12):1706-16. Review.
3. Pennisi E. News Focus: genetics. The geneticist's best Friend. Science. 2007. 317 (5845):1168
4. Khanna C, Lindblad-Toh K, Vail D, London C, Bergman P, Barber L, Breen M, Kitchell B, McNeil E, Modiano JF, Niemi S, Comstock KE, Ostrander E, Westmoreland S, Withrow S. The dog as a cancer model. Nat Biotechnol. 2006. 24(9):1065-6.
5. Lindblad-Toh K et al. Genome sequence, comparative analysis and haplotype structure of the domestic dog. Nature. 2005 438(7069):803-19.
6. Sutter NB, Bustamante CD, Chase K, Gray MM, Zhao K, Zhu L, Padhukasahasram B, Karlins E, Davis S, Jones PG, Quignon P, Johnson GS, Parker HG, Fretwell N, Mosher DS, Lawler DF, Satyaraj E, Nordborg M, Lark KG, Wayne RK, Ostrander EA. A single IGF1 allele is a major determinant of small size in dogs. Science. 2007. 316(5821):112-5.
7. Karlsson EK et al. Efficient mapping of Mendelian traits in dogs through genome-wide association. Nat Genet. 2007. 39(11):1321-8.

ASPicDB:
A database resource for alternative splicing analysis



Tiziana Castrignano`[1], Mattia D'Antonio[1] and Graziano Pesole[2,3]

[1] Consorzio Interuniversitario per le Applicazioni di Supercalcolo per Università e Ricerca, CASPUR, Rome, Italy,
[2] University of Bari, Dipartimento di Biochimica e Biologia Molecolare, via Orabona, 4, Bari 70126, Italy
[3] Istituto Tecnologie Biomediche del Consiglio Nazionale delle Ricerche, via Amendola 122/D, Bari 70126, Italy

Introduction

ASPicDB is a database designed to provide access to reliable annotations of the alternative splicing (AS) pattern of human and mouse genes and the functional annotation of predicted splicing isoforms. Splice site detection and full-length transcript modelling have been carried out by a genome-wide application of the ASPic algorithm [1-3], based on the multiple alignment of gene-related transcripts (typically a Unigene cluster) to the genomic sequences, a strategy that greatly improves prediction accuracy compared to methods based on independent and progressive alignments. Only human and mouse genes for which at least a RefSeq NM curated transcript and a Unigene cluster were available were included in the database.

ASPicDB, which is regularly updated on a monthly basis, also includes information on tissue-specific splicing patterns of normal and cancer cells, based on available EST sequences and their library source annotation.

Data production

A high-throughput software platform, *HTC for ASPic*, has been developed to produce large-scale alternative splicing analysis and transcript isoform data. It integrates computational intensive algorithms developed previously [1-3] with suitable web services and databases. The system has been optimized programming multi-threaded powerful Java client for data preprocessing and several distributed application servers for intensive computation. *HTC for ASPic* divides the input (lists of pairs of genomic sequence and its related Unigene cluster) into parallel tasks and it scales linearly with the number of dedicated processors. The system is also fault-tolerant. The web resource is available free of charge for academic and non-profit institutions at the site `http://www.caspur.it/HTC4aspic`. After each analysis a detailed file of logs can be accessed to manage the results. Furthermore each result can be visualized with the same graphics used in ASPicDB to provide maps for genes, transcripts, introns or exons.

A set of scripts allows us to insert new entries, update the existing one (in case of modification in genomic sequence or Unigene cluster) or delete obsolete entries. The same scripts validate the *HTC for Aspic* results before inserting them into ASPicDB; when the predicted RefSeq does not match with that provided by NCBI the entry is discarded.

Database Content

A first version of ASPicDB, containing only human data, has been already published [4]. Since last publication many improvements have been implemented both for database content and web interface.

New features of the database include:

- AS data for all mouse genes;
- crosslinks between human and mouse AS data for orthologous genes;
- new retrieval and download facilities at the exon level;
- Blast searches against transcript and protein isoforms collected into the database.

Table 1 reports some statistics on the data contained in the current version of ASPicDB (v1.2, May 2008) which currently contains splicing predictions for 18,599 human genes and 16,849 mouse genes.

	Human	Mouse
Genes	18,599	16,849
Transcript	278,515	103,167
Proteins	183,775	65,801
Exons	366,263	248,646
Introns	350,552	207,721
U2	294,848	192,763
U12	1,789	900
Splicing events	212,750	92,879

Table 1. ASPisDB statistics (v1.2, May 2008) for human and mouse genes.

We estimated that over 91% of human multi-exon genes may generate alternative isoforms and that each gene - on average - may generate about 12 different transcripts and 11 different proteins, most of them translated in frame with the RefSeq annotated protein. The resulting 10% of "untranslated" isoforms includes those transcripts for which a reliable ORF could not be annotated automatically. ASPicDB also contains information about cancer vs normal tissue specificity for 17 tissue types at both gene and splice site level. After an *HTC for Aspic* execution, a set of scripts evaluates the statistics of the gene expression patterns for each tissue in both normal and cancer conditions.

Database web interface

The database has a new web interface enabling a friendly and through exploration of the AS data. In particular, the ASPic data can be accessed through simple or advanced query interfaces. The simple query form allows the user to obtain the ASPic output for one or more genes selected according to one of their HGNC, Unigene, RefSeq, Entrez or MIM IDs, or according to a keyword term, or to their associated Gene Ontology (GO) IDs or textual terms belonging to the "biological process", "molecular function" or "cellular component" categories. The simple search form is shown in Figure 1.

The advanced query form allows the user to search for: 1) genes; 2) transcripts; 3) exons or 4) splice sites, fulfilling different criteria (e.g. type of splicing event, type of donor/acceptor splice

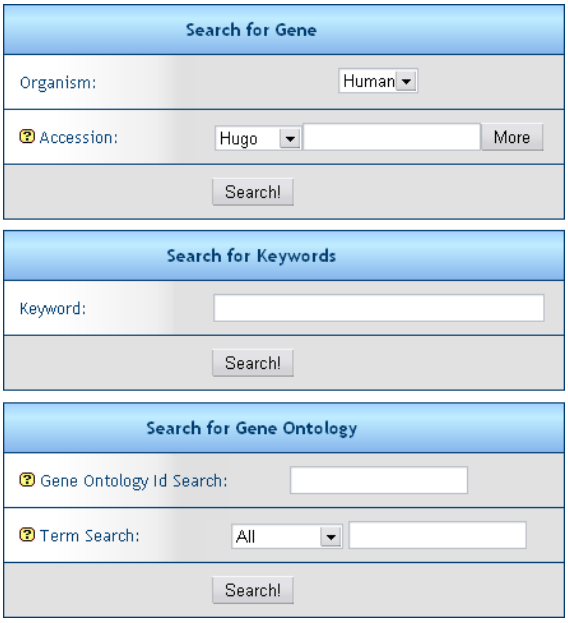

Figure 1: Simple query form.

site, etc.). Depending on this choice four separate query forms appear.

An exon search can be performed, for example, selecting a specific exon class (e.g. initial, internal, terminal and/or within a length range) or specific features of the flanking splice sites. The advanced query form for exons is shown in Figure 2.

Enhanced query and download facilities allow the users to select and extract specific sets of data related to genes, transcripts, exons and introns fulfilling a combination of user-defined criteria. Several tabular and graphical views of the results are presented, providing a compre-

Figure 2: Advanced search form for exons.

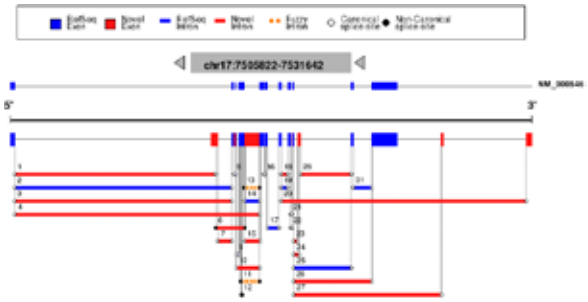

Figure 3A: Predicted gene structure.

hensive assessment of the functional implication of alternative splicing in the gene set under investigation. In the Gene Information panel a link between orthologs genes is also provided. A sample output for gene TP53 is shown in Fig. 3A and 3B.

It is also possible to retrieve set of genes or splice sites differentially expressed in the normal or cancer condition in 17 tissue types using different significance thresholds.

Finally a Blast facility allows the comparison between a user submitted sequence against the full collection of alternative transcripts and proteins collected in ASPicDB.

The Blast search can be carried out through a:

- a web form interface
- a dedicated web service

The web interface can be accessed by clicking on the "Blast" button in the left panel in the ASPidDB home page. The Blast web service enables user-developers to access to the remote services while using their own application written in any language, without having to know details of the architecture and implementation of the service. On public WSDL page the developer can read everything on messaging in XML for the service of interest.

A developer can blast ASPicDB, retrieve the results and use them in a workflow for further analysis. The WSDL descriptor for blast-ASPicDB web service is available at (`http://www.caspur.it:8080/webservices/Blast2Aspic.jws?wsdl`)

We are going to develop a set of Blast2Aspic Web Service API to allow the user to blast in a very simple way a huge list of query sequences.

Figure 3B: Predicted transcripts for gene TP53.

Conclusion and future work

ASPicDB is an ongoing project and we plan to further develop it in the next releases. The annotation of predicted isoforms will be further enriched by including information on specific regulatory elements in alternative mRNA untranslated regions and the functional features of the predicted protein isoforms (e.g. occurrence of PFAM domains, signal peptides, transmembrane helices, etc.). We also plan to extend the database to other organisms for which the genome sequence and a suitable amount of expressed sequences is available (e.g. rat, cow, zebrafish, etc.).

References

1. Bonizzoni P, Rizzi R, Pesole G. "ASPIC: a novel method to predict the exon-intron structure of a gene that is optimally compatible to a set of transcript sequences." BMC Bioinformatics. 2005 Oct 5;6:244.
2. Bonizzoni P, Rizzi R, Pesole G. Computational methods for alternative splicing prediction. Brief Funct Genomic Proteomic. 2006 Mar;5(1):46-51.
3. Castrignanò T, Rizzi R, Talamo IG, De Meo PD, Anselmo A, Bonizzoni P, Pesole G. ASPIC: a web resource for alternative splicing prediction and transcript isoforms characterization. Nucleic Acids Res. 2006 Jul 1;34(Web Server issue):W440-3.
4. Castrignanò T, D'Antonio M, Anselmo A, Carrabino D, D'Onorio DM, D'Erchia A, Licciulli F, Mangiulli M, Mignone F, Pavesi G, Picardi E, Riva A, Rizzi R, Bonizzoni P, Pesole G. ASPicDB: A database resource for alternative splicing analysis. Bioinformatics. 2008 Apr 3;

The Grid in practice

José R. Valverde

EMBnet/CNB, CNB/CSIC,
C/Darwin, 3, Madrid 28049

Introduction

"To ask may bring a moment's shame,
but not to ask is to remain in ignorance,
and so condemn oneself to lifelong shame."
(The Code of Scolarship)

Distributed programming taken to the extreme results in Grid computing, where we transcend the local computer center (or computer farm) to access shared resources in other locations all over the World.

Grid computing must solve a number of additional challenges resulting from the wide geographic distribution of the resources used and from its shared nature (they belong to different owners, unknown to us, whose interests may potentially conflict with ours). In this tutorial we will see some of the basic problems and understand how we work currently on the Grid.

Nowadays most of the work performed on the Grid is managed by batch processing, a technology that is overly familiar in High Performance Computing centres, as it has been used since the very first times of computing. We will have a chance to see how this work procedures have been taken to the Grid and the practical consequences derived.

Given the characteristics of batch processing, we do not need to change our programs to make use of the Grid: we simply use the same executables we already have. As a consequence, in this tutorial you will not be required to know any programming language, and as a result, we can say that to use the Grid it is not necessary to know about programming (although it helps if you want to make complex tasks); it is enough to have the executables. We are going to use **EGEE** for out examples. EGEE is the European production Grid. To use it we need special permissions, as we have already described. In the next sections we will describe the basic working procedures with very simple examples. Now, if you are curious to learn how you can use thousands or tens of thousands of computers to run your programs easily, just follow on.

The first steps

A ducks legs, though short, cannot be lengthened without discomfort to the duck;
A crane's legs, though long, cannot be shortened without discomfort to the crane.
(Chuang-Tzu)

Connecting to the UI

Our first step is obvious: we need to connect with a "front end" (also known as User Interface node, UI) that will allow us to harness and control the Grid. We will normally use ssh to connect so that all our subsequent work is securely protected, and will access the UI using our username and password as provided by the UI system administrator.

It is relevant to note here that the UI is a full UNIX (Linux actually) system, which *in addition*, is connected to the Grid. In other words, we can carry out any usual task we normally perform on LINUX/UNIX machines (like compiling our program, read e-mail, edit files or play 3D team games under X).

Transferring our certificate

The UI is just a gateway to the Grid, a machine that knows how to issue commands to the Grid, and that can be set up by anybody. If we do not want the Grid abused, we simply cannot trust any UI (or anybody with access to an UI) totally. We might implement verification mechanisms for the UI, but that will not solve the problem of identifying the people that uses the UI (and the Grid through it) as its owner may grant access to the UI to anybody he or she wishes.

In order to actually use the Grid we need a Grid identity, separate from the one we have on the UI. The identity on the UI is our username as provided by the UI owner, but our identity on the Grid is a personal "token", like an ID or credit card or passport, issued by a trusted authority and accepted by the Grid administrators. This personal token is not a physical card but an electronic "certificate".

As we have said, we must validate this certificate with a trusted authority, known as **Certificate Authority, CA**. There are many CAs out there (like Verisign or many national authorities): in order to learn which CA should we use our first step will usually be to go to the web pages of the Grid and find out which CA is trusted in our area. Once we know, we will generate a **personal certificate** including our personal identification details. Obviously we could be liars, that is why we are forced to validate it with the CA: the process will usually involve sending our personal certificate to the CA asking them to validate it; someone from the CA will get in contact personally with us to verify our claimed identity and after verifying that we really are the same person described in our personal certificate, the CA will sign it to stamp its validation certificate. It is this validated, signed certificate we got from the CA (and not the original CA we generated ourselves) that we must then send to the Grid administrator asking them to grant us access to the Grid. Once the administrator adds our signed certificate to the list of valid users, we can start using it to access the Grid.

To better understand the process, we can draw a parallel with accessing a foreign country: to do this we need a visa, and to get it we will usually start by finding the location of the offices were we can request a passport. To get the passport we first fill in a form with our data and submit it to a local authority which is trusted by the foreign country we want to visit, usually the local police. The local police/authority will take our form but will not issue a passport just on the data we provided: they will first verify the validity of the identification details we provided, and that they really belong to us. Once an approved officer has verified we really match the details in the form, we can get the passport. A passport is just a copy of the data we ourselves filled in in the form, in a nice booklet, and stamped with the local authority stamp and signature, just like the signed certificate we get from a CA is a copy of the one we generated by filling in our details signed by the CA; as such it is good enough to identify ourselves to anybody who trusts the issuer, but in many instances is not enough to grant us access to any country, it is just a valid proof of identity, nothing more. Once we have our valid passport issued by a trusted authority, we still need to get a visa: we simply produce our passport to the appropriate authority for the country we want to visit (or to the administrator of the Grid VO we want to use) and explain what do we want the visa for (work, tourism, fun, ...) and if they see it fit, they will grant access to visit the country (use the Grid VO) for the owner of the passport (of the personal certificate).

So, here we are, we have a valid certificate, signed by an approved CA, and accepted by the Grid administrator of the VO we want to use. Separately we got access to a UI node with independent user name and password. We must identify ourselves to the Grid from the UI, hence we must copy our valid, signed certificate to the UI. *We only need to do this once.* Once we have copied our certificate to the UI we can leave it there for future use (until it expires and we need to copy a new one). Obviously, if we leave it on the UI, it should be protected somehow as the UI administrator could otherwise impersonate us, that is why the certificate should be protected by a good, long pass phrase.

We can use any standard method to copy the certificate to the UI (scp, ftp...), we just need to remember that

1. first we must generate a special directory named "`.globus`" in our home

2. the certificate must be stored within this directory "`~/.globus`"

3. the certificate is composed of two files, and we must transfer both of them:
    1. `usercert.pem`
    2. `userkey.pem`

We can verify that the certificate has been correctly transferred using the command `grid-`

`cert-info`. If the certificate is correctly installed in the appropriate subdirectory `~/.globus` then we will see something similar to this:

```
# grid-cert-info
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 795 (0x31b)
        Signature Algorithm:
             md5WithRSAEncryption
        Issuer: C=ES, O=DATAGRID-ES,
             CN=DATAGRID-ES CA
        Validity
            Not Before: Mar 16 12:31:36
             2006 GMT
            Not After : Mar 16 12:31:36
             2007 GMT
        Subject: C=ES, O=DATAGRID-ES,
             O=CNB, CN=Jose Ramon
             Valverde Carrillo
        Subject Public Key Info:
            Public Key Algorithm:
rsaEncryption
...   ...   ...
```

As you can see the certificate has validity start and termination dates (just like a passport) and therefore must be renewed periodically.

Once more, let us not forget that the UI is a full LINUX system, with multiuser support and so on... this is to say, if we do not take care, other users might abuse our certificate: we should protect our certificate using a pass phrase that is long enough and difficult to remember. It won't be a serious hassle when we later want to use the Grid and will dramatically increase our security.

Activate the Grid

The correct term is creating a "*proxy* certificate" to work on the Grid (i. e. actually a new temporary certificate is created from ours, with a validity much, much shorter). In practice, we may think of what we are about to do next simply as a way to start a session with a limited duration.

The command to use is **voms-proxy-init**.

```
# voms-proxy-init
Your identity: /C=ES/O=DATAGRID-ES/
       O=CNB/CN=Jose Ramon Valverde
       Carrillo
Enter GRID pass phrase for this
       identity: enter pass phrase (not
       shown)
Creating proxy .................
.................. Done
Your proxy is valid until: Sat May  6
06:37:52 2006
```

This command will look into the directory named '`.globus`' in our home directory for a valid certificate, and if found will assume this is the one we want to use by default to open a new working session on the Grid.

As we can see this command displays our identity (which is stored within our certificate), and hence we do not need to specify a username to open our session on the Grid: we will work under the identity stored in the certificate. Just after that it prompts us for a pass phrase. This is the long sentence we have used to protect our certificate, and as any other password, it will not be shown while we type it.

Additionally, we can see that once we have typed in a valid pass phrase, the command states a validity period for our "proxy" (and hence for our work session). Usually this period will be 12 hours, but we may request longer or shorter periods when invoking the command (see `man voms-proxy-init`).

The validity period of the "proxy" is relevant: all commands we issue to the Grid afterwards will be associated with it (to verify we are authorized to issue them), thus the Grid will only accept our commands as long as the "proxy" associated to them is valid. When the proxy expires, our work session will expire as well, and it will not be possible to work on the Grid any longer. Our works will have run out of oxygen and die if they have not finished yet. Therefore, we want the validity of our proxy to last long enough. On the other hand, once we have activated a session (a proxy) it will be tied to our account on the UI for all its duration, and anybody logged in on our account will be able to use the Grid. Normally that will be only us, but since the security of a UNIX/LINUX password is usually lower than that of the pass phrase needed to access the Grid (if we chose a good, long one), the chances of someone breaking in on our account (even if low) make it desirable to have a proxy that does not last longer than nec-

essary (so that should a intruder gain access to our account on the UI, s/he can not use the Grid -possibly for malicious purposes- disguised as us).

So, in summary, how long should **a proxy be valid? A basic rule of thumb, a proxy should be valid for as long as we estimate our work will need to complete**. This would imply we need to know in advance how long it will take, and as we all know, most times there is no way to know in advance (actually, in general *there is no way to know in advance when a job will complete on the Grid*). This is not that bad as it seems: we can issue new proxies to extend the session (or rather overlap sessions) as needed, hence it is enough to submit a job, and then monitor it periodically before the current proxy/session expires to see if it has finished, and if it has not, then start a new session with the same `voms-proxy-init` command.

Once we have activated the work session on the Grid we can submit as many jobs as we need on any VO where we have been accepted.

Verifying the Grid status

This is normally not needed (unless something seems to be going wrong), but it is useful and fun to have an idea of in which shape is the Grid currently: the command `lcg-infosites` allows us to gather information on the status of the machines that belong to (support) a VO. For example, to see the status of all BIOMED components we could use

```
# lcg-infosites --vo biomed all
****************************************
These are the related data for biomed:
(in terms of queues and CPUs)
****************************************
#CPU    Free    Total Jobs      Running
Waiting ComputingElement
----------------------------------------
  93      53       9
9                   0    ce04.pic.
es:2119/jobmanager-lcgpbs-biomed
  12       8       0              0
0    polgrid1.in2p3.fr:2119/jobmanager-
lcgpbs-sdj
   7       7       0              0
0    ce00.inta.es:2119/jobmanager-
lcgpbs-biomed
...    ...    ...
```

Please, note that we use two dashes (-) before the "`vo`" parameter. The listing is usually very long, and we have cut it short here for the sake of brevity.

We may as well find out which resources are appropriate to run our job given a description of its needs, and therefore can be used to actually run it. To do this we need a file describing our job (more details below) and the command

```
edg-job-list-match -vo=biomed myjob.jdl
```

Here "`myjob.jdl`" is a file that contains a description of the requirements for running our job. The command will query the Grid and tell us which resources (work nodes, storage elements, etc..) are available to run it.

Creating a job

If you want to realize such a thing, you must be such a person.
Once you are such a person, why worry about such a thing?
(Yunju)

Once we have activated the session we can start submitting jobs to the Grid. As many as we want. From one to millions as long as our proxy is still valid.

To submit jobs we use a method very similar to the one that has been traditionally used to submit jobs to a supercomputer: we start by creating a file where we describe what we expect or want the Grid to do and the characteristics of our work, and then we send this file to the Grid for processing.

The file that contains a description of our work is written in a special language known as JDL (Job Description Language).

JDL the Job Description Language

The description of our work must include at the very least certain number of basic elements, and possibly several additional optative elements. To illustrate this we will work over an example:

```
Type            = "job";
JobType         = "normal";
VirtualOrganisation = "biomed";
Executable      = "/bin/sh";
StdOutput       = "output.txt";
StdError        = "error.txt";
InputSandbox    = { "job.sh",
      "executable", "data",
      "configuration" } ;
OutputSandbox   = { "output.txt",
      "error.txt", "results" };
RetryCount      = 7;
Arguments       = "job.sh -i data -o
      results";
Environments    = { "PATH=.:$PATH",
      "INSTALLDIR=." };
Requirements    = RegExp("cnb.uam.es",
      other.GlueCEUniqueId);
Rank            = 1000 * (other.
      GlueCEInfoTotalCPUs -
       other.GlueCEStateWaitingJobs)
       / other.GlueCEInfoTotalCPUs;
```

While not exhaustive, this example allows us to see the most common options (and some that are not so common):

- The first two directives (`Type` and `JobType`) identify this as a normal job

- `VirtualOrganisation` (note the 's') is optional and states which VO will be used to process the job. If none is stated on the file we can specify one on the command line.

- `Executable` (needed) specifies the name of the program to be executed. This program must be available on the destination systems where is will be run. It is possible that our programs are already installed (e.g. system commands or common commands to all systems in a given VO) or that they are not, in which case the executable must be copied to the remote machine in order to be executed (see below)

- `StdOutput` (optional) and `StdError` (optional). Our program will be run on a remote, unknown machine somewhere in the World, totally independent from us, and so we will not be able to interact with it. These parameters allow us to redirect its standard output and error messages to the specified files in the remote machine (or to discard them if not specified).

- `InputSandbox` (optional): the concept of *sandbox* is not exclusive of the Grid. A sandbox (like the ones used by kids playing on the kindergarten) is a safe space where we can merrily do anything we want in the confidence that we can do no harm (neither to ourselves nor to the system). Our job will be run on the remote machine in a "secured" environment to avoid users harming the remote systems of others. In order to run our job, this secured environment must contain all the tools and toys needed for it, and this is what we specify here: `InputSandbox` refers to all the files needed by our job to be run, and hence to build a complete sandbox for it on the remote executing system. When we submit the job description, the Grid will parse this and copy all the files specified here to the remote system before running the command requested. Hence, if we want to run a program that is not installed by default in the remote machine, we can include the name of the executable file here and it will be copied before being run, so that when the Grid next tries to run whatever we specified in "`Executable`" it will already be there. Obviously if the program is already installed, we do not need to copy it. Other files included here are all other files needed to run our program (data, input options, etc..).

- `OutputSandbox` (optional) states which files we want to recover from the Grid once the work is finished. Our program may generate any number of files while run, and here we state which among all of them we want to copy back after the jobs has completed. It should include (if we want them) `StdOutput` and `StdError` as well, otherwise they will be generated but not recovered. Please, note that there are limits to the sizes of the input and output sandboxes and that if need to exceed them, then we must resort to other facilities of the Grid to circumvent them.

- `RetryCount` (optional) states the number of times that our job should be retried if there is any problem while executing it on the Grid. If zero, then it will be attempted to be run only once.

- `Arguments` (optional) allows us to state the arguments that we usually add to the command on the command line.

- `Environments` (optional) is used to set environment variables needed to be defined before running our program.

- `Requirements` (optional) forces the Grid to choose the resources that satisfy our conditions. Usually we will simply allow the Grid to assign them freely, but in this example we have requested that our job will run only on machines dependent from CNB.

- `Rank` (optional) is a way to indicate how to measure the "goodness" of a computer center or farm: the job will be submitted to the CE with best score. Normally we will not use it.

Additional examples

Let us see a few more examples to better understand this:

```
Type     = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "hostname";
StdOutput = "where";
OutputSandbox = "where";
```

This example runs the system command "`hostname`" (which tells us the name of the computer). In practice this will tell us the name of the computer where it is run (which will not be ours). Since `hostname` display the name of the computer on its standard output, we need to redirect its standard output to a file ("`where`") so that we can recover it afterwards (by specifying it in the `OutputSandbox`). If we submit this job, once completed, we can recover its output sandbox consisting of its standard output stored in a file named "where" and whose contents will be the name of the computer where it was run.

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "/bin/ls";
StdOutput = "listing";
OutputSandbox = {"listing"};
Arguments = "-l";
```

This one is similar, but this time we run the system command '`ls`' with the command line argument '`-l`' to obtain a complete listing of the directory where it is run. We will save its output on a file named "`listing`" and recover it through the output sandbox.

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "clustalw";
StdOutput = "output";
StdError = "error";
InputSandbox = { "clustalw",
       "secuences", "input" };
Arguments = "< input";
OutputSandbox = { "output", "error",
       "secuences.dnd", "secuences.aln"
       }
```

This JDL would run the command "`clustalw`", but as we do not expect it to be already installed in any remote system, we must copy it to the remote executing node before running. In addition, as "`clustalw`" needs to read its parameters from its standard input (normally interactively) we also copy a file containing these parameters (as we would have typed them) and ask it to read them from a file. Thus we must copy (`InputSandbox`) the program executable file, the input file and the file with the sequence data. When it finishes we get (`OutputSandbox`) its output, any errors and the main files generated by Clustal (the dendrogram and the multiple alignment).

Note: as of recent versions of the middleware the command line is passed *"as is"* to the program and is not processed by a shell, hence I/O redirection and pipes specified as "`Arguments`" do not work. Not to worry, as there are other ways to achieve this, and coincidentally these are a lot better (see below).

The miracle trick

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "job.sh";
StdOutput = "out";
StdErr = "err";
InputSandbox = { "job.sh", "job.tgz" };
OutputSandbox = { "output.tgz" };
```

This is a trivial JDL, but with a generic application, where we use several tricks that make it advisable as a generic JDL to simplify job submission: this JDL copies a shell script and a compressed 'tar' package to the remote system, then runs the script redirecting its standard output and error, and finally specifies as output to be recovered another compressed package.

The trick consists of packing -and compressing to save bandwidth- everything that is needed for our job in a single file (executables, libraries, data, etc...) which we call '`job.tgz`' and then creating a script that extracts the contents of the package, runs our programs with any needed command arguments, environment variables, etc... and finally packs the results we want into 'output.tgz'.

This is also an elegant example of a general solution for the case where we want to run a sequence more or less complex of commands instead of just a single one: we would copy all executables and then run the commands, using pipes, I/O redirection and all the utilities of a shell just as we would locally. A sample script might look like:

```
#!/bin/sh
tar -zxvf job.tgz
       # extract all files needed from
       the package
# set up the environment to find shared
       libraries and executables
export LD_LIBRARY_PATH=./tinker/
       lib:$LD_LIBRARY_PATH
export PATH=./tinker/bin:$PATH
# do the job
pdbxyz coordinates.pdb
minimize coordinates.xyz < ./min.in
anneal coordinates.xyz_2 < ./ann.in
analyze coordinates.xyz_3 < ./ana.in
xyzpdb coordinates.xyz_3
# pack only the results we want
tar -zcvf coordinates.pdb_3 analyze.out
anneal.out minimize.out
```

Before submitting this script we will need to make a package containing all executables, shared libraries, parameter files for TINKER, files with program options for the programs we want to run (`*.in`) and the original data file (`coordinates.pdb`). The script will extract the package, set up the environment, run a series of commands that generate a large number of files and save for later recovery in a package only those that we will finally be really interested in. When we recover the results we only need to extract the contents from this package.

Once the job has been described we only need to submit it. This is what we are going to learn next.

Executing jobs

Once we have prepared the executables and data files and written a job description using JDL, we can finally run our job on the Grid.

Job submission

All we need to do to submit a job is use the command edg-job-submit stating the 'JDL' file to be processed and any needed extra options (e. g. if we have not used the `VirtualOrganisation` directive we can indicate now which `VO` to use with the option "`-vo=VONAME`")

```
# edg-job-submit -vo=biomed job.jdl
Selected Virtual Organisation name
       (from --vo option): biomed
Connecting to host egee-rb-07.cnaf.
       infn.it, port 7772
Logging to host egee-rb-07.cnaf.infn.
       it, port 9002
**************************************
                     JOB SUBMIT OUTCOME
 The job has been successfully
       submitted to the Network Server.
 Use edg-job-status command to check
       job current status. Your job
       identifier (edg_jobId) is:
 - https://egee-rb-07.cnaf.infn.
       it:9000/h7nQIl1ql5-oQsMJwABP8Q
**************************************
```

This command submits the Job to the Grid and displays information about it, most importantly the unique "`identity`" of our job in the Grid. It is very important to note down this identity as it is our only means to refer to our job when we want to ask the Grid about its status or we want to recover its output after it terminates. The identity is -as you can appreciate- a URL and it is really difficult to remember. For this reason, it is better to issue the command `edg-job-submit` followed by the command line option "`-o=file`" to ask that this identity be added to the file specified

and so that we can later use this file to refer to our job.

```
# edg-job-submit --vo biomed --output
      job.id job.jdl
Selected Virtual Organisation name
      (from JDL): biomed
Connecting to host egee-rb-07.cnaf.
      infn.it, port 7772
Logging to host egee-rb-07.cnaf.infn.
      it, port 9002
======= edg-job-submit Success ========
 The job has been successfully
      submitted to the Network Server.
 Use edg-job-status command to check
      job current status. Your job
      identifier (edg_jobId) is:
 - https://egee-rb-07.cnaf.infn.
      it:9000/4yh-ujUotqgxe9Gv13S5hw
 The edg_jobId has been saved in the
      following file:
 /home/jr/sample.id
=======================================
```

As you can appreciate, the difference is that now we are told that the identity of our job has been *appended* to the contents of the specified file.

Verifying the status of a job

Once submitted, we can verify the status of our job in the Grid using the command `edg-job-status` and the identity of our job. If you were wise enough to follow our advice from the previous section and saved it on a file, it will be easier to use this file to indicate the identity (instead of typing -or copy-paste- a long and random URL).

```
# edg-job-status --input sample.id
*****************************
*****************************
BOOKKEEPING INFORMATION:
Status info for the Job : https://
      egee-rb-07.cnaf.infn.it:9000/
      4yh-ujUotqgxe9Gv13S5hw
Current Status:     Scheduled
Status Reason:      Job successfully
      submitted to Globus
Destination:        mallarme.cnb.uam.
      es:2119/jobmanager-pbs-biomed
reached on:         Sat May  6 16:01:54
      2006
*****************************************
```

As you can also see we do not need to specify the VO any more as we are using a unique job ID.

Recovering job results

After our job has completed (`Current Status: Done`) we need to recover the output results from the Grid using `edg-job-get-output`. By default the results recovered will be saved on a directory under `/tmp` with out username and the job ID. This is inconvenient and most times we will want to save the results on a local subdirectory or our current work directory, which we can do using the argument "`--dir`":

```
# edg-job-get-output --dir results --
      input sample.id
Retrieving files from host lxshare0219.
      cern.ch
*****************************************
                 JOB GET OUTPUT OUTCOME
 Output sandbox files for the job:
 - https://egee-rb-07.cnaf.infn.
      it:9000/4yh-ujUotqgxe9Gv13S5hw
 have been successfully retrieved and
      stored in the directory:
/home/jr/resultados/4yh-
      ujUotqgxe9Gv13S5hw
*****************************************
```

All that remains is to get into the newly created directory and see the results.

Summary

As we have seen till now, using the Grid is not that difficult. Basically, once we have done the initial registration steps, all that is needed from us whenever we want to use the Grid is that we follow a number of easy steps:

1. Once every work session start it with `voms-proxy-init`
2. For each job
    1. prepare a description on a JDL file
    2. submit it with `edg-job-submit`
    3. verify its status periodically with `edg-job-status`
    4. when it has finished recover its output with `edg-job-get-output`

This is the easiest and simplest way to work. We have described only the basic mechanisms to il-

lustrate the philosophy of work, but we have a lot more versatility at our reach than what we have demonstrated: for instance we can specify that a job is to run a parallel program using MPI over a minimum number of nodes, or state that a job must be interactive (and then the Grid will open control connections between the remote node and a local terminal so we can interact directly with it) instead of a batch process, state a large number of requisites, require that it be allocated to queues with a given priority...

And there's more: we also have storage elements on the Grid where we can store and access data distributed all over the world, so that we can access large disk spaces and use huge datasets. We can modify existing programs and develop new ones using specific Grid APIs so that our program may include specific Grid support (similar to what we do with MPI), use high level routine libraries to access Grid resources through high level abstractions of the available distributed resources (like GlobalArrays which mimics distributed shared memory on clusters, NOWs or the Grid, DRMAA which provides a standard API to the Grid queue manager, etc..).

In any case we must keep in mind that Grid technologies are currently undergoing a quick and extensive development and that there is still a lot to be done and discovered regarding the best ways to exploit them. If you are interested in learning more, you are mostly welcome to participate in this most exciting field.

Acknowledgements

We want to thank EMBnet[1] for making publicly available its education web site [2], and the EU for its support to projects EGEE[3] (INFSO-RI-031688) and EMBRACE[4] (LHSG-CT-2004-512092) which have allowed us to do this work.

1. http://www.embnet.org
2. http://edu.embnet.org
3. http://www.eu-egee.org
4. http://www.embracegrid.org

Simplifying job management on the Grid

José R. Valverde

EMBnet/CNB, CNB/CSIC,
C/Darwin, 3, Madrid 28049

In previous articles we got a general vision of the Grid and started dealing with jobs and their management on the Grid. In this article we will try to systematize these experiences while performing phylogenetic analysis (ClustalW), molecular dynamics simulations (TINKER) and quantum mechanics computations (PSI3), and we will see how most of the process can be automatized.

What's in a job?

धर्मो रक्षति रक्षितः

Order protects those who protect order

In our previous article we saw a sample JDL to run **clustalw**. Let us have a look into it again:

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "clustalw";
StdInput = "input";
StdOutput = "output";
StdError = "error";
InputSandbox = { "clustalw",
       "secuences", "input" };
OutputSandbox = { "output", "error",
       "secuences.dnd", "secuences.aln"
       };
```

We needed to send the sequences to align (obviously) and, since **clustalw** is an interactive program, we needed to supply its input on a separate file (which also needs to be copied in the `InputSandbox`) and tell the Grid to feed it to

clustalw. As we do not expect **clustalw** to be a standard UNIX program installed on any remote machine, we also had to send a copy of it in the `InputSandbox`.

More generally, whenever we want to run a job in the Grid we need to provide everything that may be needed, and what is that?

- A COMMAND TO EXECUTE. Only standard commands can be expected to be available on remote nodes (actually, the Grid is rather homogeneous, running on Scientific Linux, but as local administrators may decide to install different subsets or all of the SL distribution, we can only rely on the standard UNIX and Grid tools being available). Any non-standard command must therefore be copied to the remote node for execution in the `InputSandbox` (there are ways around this, but we are not reviewing them now).

- OPTIONAL PARAMETERS. Most UNIX commands accept parameters on the command line. We can specify them as *Arguments* in the JDL. Most often these will serve to specify the input and output files to be used (but not necessarily as the clustalw example shows). The same can be said of *Environment* variables, which are a special way to provide additional arguments or modification notices to our programs.

- INPUT DATA. All data to be processed is on our local system. The Grid does not know -nor has it way to know- which input files will be needed for remote execution in advance, and so we must add them to the `InputSandbox`.

- AUXILIARY DATA. Besides the files we want to analyze, our program may need additional data files with auxiliary information (parameter files, databases, shared libraries, etc...). These must also be made available at the remote node, usually copying them as parts of the `InputSandbox`. Again, there are ways around this but we are not going to deal with them now.

- OUTPUT DATA. We want to perform a computation to achieve some results in the form of resulting data generated by the program. If it is produced on its standard output we can collect it to a file with the `StdOutput` and `StdError` directives, otherwise it will be saved on a file somehow. In any of these cases, we need to retrieve the results from the remote node by specifying the data on the `OutputSandbox`.

- SPECIAL REQUIREMENTS. Most often our job will be pretty standard and straightforward and the only special requirements we will need to specify are that ours is a normal job to be run on our default V.O. Some times, we will want to add some additional requests (like an MPI cluster, some minimum run time, etc...

All this is specified in the JDL as we have seen and may lead to a rather complicated JDL. It also requires us to change the JDL for each different command we want to run: if for instance, we wanted to run **t-coffee** instead of **clustalw** we would need to draft an altogether different JDL.

clustalw revisited

There is a different way in which we can run clustalw though: instead of crafting so much information in the JDL we can simplify the JDL by executing a script and moving as much logic into the script as possible. For example, we could use

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "clustal.sh";
StdOutput = "output";
StdError = "error";
InputSandbox = { "clustal.sh",
       "clustalw", "sequences", "input"
       };
OutputSandbox = { "output", "error",
       "sequences.dnd", "sequences.aln"
       };
```

Notice that in this JDL we have substituted the **clustalw** command by the name of a shell script (`clustal.sh`) and that we have added the script to the input sandbox as well. We do not need now to specify the standard input as we can take care of it in the script. A sample script might look like:

```
#!/bin/bash
ls -l
chmod 755 clustalw
./clustalw < input
ls -l
```

There are several things worth noting about this script. First is that we need to set execute permissions for **clustalw**: when we copy the input sandbox to the remote working node, the Grid knows that there is one executable file (the one labeled as such, `clustal.sh`) and sets the permissions for it, but all other files (including **clustalw** itself) are set to default read-write permissions.

So that you can verify by yourself, we have added two '`ls`' commands, one at the very beginning (so you can check that **clustalw** is indeed copied as a data file not as an executable) and one at the end (so you can see the final status of files). The output of all these commands will be in the standard output file "`output`" that we collect in the output sandbox.

Finally, notice that a script gives us more freedom: now *we can specify I/O redirection* (which can not be specified on the JDL), producing as complex pipelines as we need, and *we can run more than one command* on the same job. Using pipelines with more commands would entail adding the files to the input sandbox and setting the execute permissions before running the pipeline, but that's all.

The ability to execute more than one command is also *handy to debug and track job execution remotely, and to gather other relevant info* (like execution statistics). Usually we can only get job results if the executable finishes successfully, and hence, if we are running **clustalw** and it fails for one reason or another, we will never get anything back (and will not be able to know what went wrong). Using a script we can intercept **clustalw** failure and collect additional data, make the script store any needed data (as the '`ls -l`' output above) and retrieve it for later inspection. Note that you may need to trap abnormal exit conditions to avoid the shell aborting in the middle of execution (which would result in a failure status and all output being lost); check `bash set -e` and `trap` directives for more information.

Using a generic job

एकोहमसहायोहं कृश्नोहमपरिच्छदः ञ

स्वन्पेप्येववंविदा चिंता मृगेंग्रस्य न जायते ञञ

'I am alone, helpless, weak when not accompanied'
Even in its dreams the lion does not think like this.

There are compulsory reasons to resort to scripts when using the Grid: we can run more commands, win on versatility (may run complex pipelines), on JDL simplicity (we can move environment, argument, I/O directives and so on into the script), and on safety (we can ensure graceful job termination and additional monitoring). We may take this to the extreme by simplifying the JDL and moving everything else to the script. This leaves us with a minimalistic JDL file that can be used for any generic job:

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "job.sh";
StdOutput = "out";
StdError = "err";
InputSandbox = { "job.sh", "job.tgz" };
OutputSandbox = { "output.tgz", "out",
       "err" };
```

Note that in this JDL we have not specified any specific job name, executable, input or output file. It is all generic. If we save it as "`job.jdl`" it will work for almost any general kind of job. So, how do we use it?

The executable is a shell script called "`job.sh`". We need to create a shell script with that name for each job we want to run and put inside any directives needed to do our work on the remote node. We then copy in the input sandbox only two files, our script and a '`tar.gz`' compressed package. No executables, no special data files, no nothing.... where is it? You guessed right! It is all in the `job.tgz` package. In other words, we should pack inside it anything that we need to do our work: the executables, auxiliary data, input data files, etc... everything should be packed before submitting the job to the Grid. What about the output? It certainly will be different for each kind of job, but it does not matter either: we only need to retrieve another '`tar.gz`' compressed package file, requiring that any relevant output data be stored in it by our `job.sh` script before termination. Note that we need to keep standard output and error out of the output package:

that is because they won't be ready until our shell script is done, and then it is too late to pack them.

An skeleton script to run any job on the Grid would then look like:

```
#!bin/bash
tar -zxvf job.tgz

# set any needed environment variables,
#      e.g.
export PATH=$PATH:.

# run as many commands as desired, e.g.
step1 | step2 | step3 > result 2>
       errors

# pack any needed output data e.g.
tar -zcvf output.tgz result errors
       otherfiles
```

There are a few interesting points: *we need to start unpacking* the `job.tgz` package in order to create all the necessary input files and executables, and *we do not need to set permissions* this time (`tar` will do that for us, restoring the original permissions they had on our local node) so the script logic is simpler. *We are free to do anything we want now inside the script*, with the full power of the shell at our fingertips, including complex setups with error recovery, monitoring, etc... *Our only concern must be to remember storing at the end any data we need to recover* in the `output.tgz` archive.

Running a complex TINKER molecular dynamics simulation

Let us have a look at how it would look if we were running a molecular dynamics simulation using **TINKER**: our shell script might look like

```
#!/bin/bash
# run an MD simulaton using TINKER
# (C) José R. Valverde, 2006
# extract contents of job with
# appropriate perms
tar -zxvf job.tgz

# set up the environment to use shipped
# shared libraries
export LD_LIBRARY_PATH=/lib:/usr/lib:./
       tinker/lib:$LD_LIBRARY_PATH
export PATH=./tinker/bin:$PATH
```

```
#!/bin/bash
# run an MD simulaton using TINKER
# (C) José R. Valverde, 2006
# extract contents of job with
# appropriate perms
tar -zxvf job.tgz

# set up the environment to use shipped
# shared libraries
export LD_LIBRARY_PATH=/lib:/usr/lib:./
       tinker/lib:$LD_LIBRARY_PATH
export PATH=./tinker/bin:$PATH

# do the work
pdbxyz coordinates.pdb
minimize coordinates.xyz < ./min.in
anneal coordinates.xyz_2 < ./ann.in
analyze coordinates.xyz_3 < ./ana.in
xyzpdb coordinates.xyz_3

# save remote host name for monitoring
/bin/hostname > host

# pack only interesting results
tar -zcvf output.tgz coordinates.pdb_3
       analyze.out anneal.out minimize.
       out host
```

As you can see, we have launched a complex execution that runs a simulation composed of various steps within a single Grid job, using I/O redirection, setting environment variables, etc.. as needed and only worried about extracting the input file and generating the output package. A side benefit of this approach is that we pack and compress all required data prior to transfer and hence *we reduce the bandwidth (and time) needed to transfer data* to and from the remote nodes.

We now turn to the issue of building the original `job.tgz` package. For simple commands like **clustalw** above it might be trivial: e.g.

```
tar -zcvf clustalw sequences input
```

But for more complex environments it may easily turn difficult quickly. To see it, let us consider the problem of running various **TINKER** molecular dynamics simulations on the Grid again.

Our first problem is that if we want to run various simulations, files for the various jobs will become mixed unless we are careful. The easiest solution

is to *assign a separate directory for each different job*. This way files from one job won't mix with files from another, specially commonly named files (`job.jdl`, `job.sh`, `job.tgz`...) will not clash into each other.

We need to copy all the executable files into the `job.tgz` input package, and then specify the execution path in the shell script to find them. In addition **TINKER** uses various auxiliary parameter files, and we need to include any of them we will use as well. We will also need to tell **TINKER** where to find those auxiliary files in the remote node.

Our advice is that you *try to keep things as clean as possible*. In this case it would make sense to create a subdirectory within the job directory to hold all tinker files and copy them in it. Then we simply pack our input data and the directory and that's it. For example:

```
mkdir tinker
mkdir tinker/bin tinker/param tinker/
      lib
cp /somewhere/tinker/bin/pdbxyz tinker/
      bin
cp /somewhere/tinker/bin/minimze
      tinker/bin
....
cp /somewhere/tinker/param/amber.prm
      tinker/param
...
tar --exclude 'job*' -zcvf job.tgz *
```

Here we simply pack all files in our job directory except the `job.jdl`, `job.sh` and `job.tgz` files which will be copied by the Grid and need to be in the archive package. The shell script will only need to specify the paths accordingly (see above).

Finally, although **TINKER** is normally linked statically, many programs are not, and *we need to spot all shared libraries required by our executable and include them as well*. This we can do using the command '`ldd`', and it can be automatized as well: for instance we could use the following script

```
#!/bin/sh
# Save an executable under ./bin and
# all its dependence libraries
# under ./lib
# Use as get_exec path_to_executable
# (C) José R. Valverde, 2006
if [ ! -e ./bin ] ; then
    mkdir ./bin
fi
cp $1 bin
if [ ! -e ./lib ] ; then
    mkdir ./lib
fi
ldd $1 | cut -d' ' -f3 | grep -v ^/lib
       | grep -v ^/usr/lib | \
while read line ; do
    if [ -e $line ] ; then echo cp
       $line ./lib ; fi
done
```

This is a very simple script that will analyse an executable file and copy it to a subdirectory called '`./bin`', and all its depended shared libraries into a subdirectory named '`./lib`' both in the current directory. It can be greatly enhanced, e. g. by adding some error detection mechanisms and other features.

Running a PSI3 Quantum mechanics simulation

As another example, let us run a quantum mechanics simulation using the package **PSI3**. **PSI3** is a publicly accessible QM environment that relies on dynamic libraries and also uses several auxiliary data files. Let us start by creating a job directory and seeding it with our template `job.jdl` file from a master copy held somewhere else (`/somewhere/grid-skel`):

```
mkdir psi3job
cd psi3job
cp /somewhere/grid-skel/job.jdl .
```

We will now create our input data file (`input.dat`), which will be

```
% This is a sample PSI3 input file.
% Anything after a percent sign is
% treated as a comment.
psi: (
    label = "cc-pVDZ SCF H2O"
    jobtype = sp
    wfn = scf
    reference = rhf
    basis = "cc-pVDZ"
    zmat = (
       O
       H 1 0.957
       H 1 0.957 2 104.5
    )
)
```

Running **psi3** on the command line shows that it needs the programs `input cints` and `cscf` to run. It will also need two auxiliary files with internal data. So, we now create the directories and copy those files inside using the script we saw above to carry over any needed shared library as well:

```
sh /somewhere/grid-skel/get_exec.sh
       /somewhere/psi3/bin/psi3
sh /somewhere/grid-skel/get_exec.sh
       /somewhere/psi3/bin/input
sh /somewhere/grid-skel/get_exec.sh
       /somewhere/psi3/bin/cints
sh /somewhere/grid-skel/get_exec.sh
       /somewhere/psi3/bin/cscf
mkdir share
cp /somewhere/psi3/share/psi.dat share
cp /somewhere/psi3/share/pbasis.dat
       share
```

Next we create our `job.sh` script:

```
#!/bin/bash
tar -zxvf job.tgz

# find auxiliary data
export PSIDATADIR=share
# find executables
export PATH=$PATH:bin
# find shared libraries
export LD_LIBRARY_PATH=$LD_LIBRARY_
       PATH:./lib

# run the QM simulation
psi3 input.dat output.dat
/bin/hostname > host

tar -zcvf output.tgz output.dat psi.32
       host
```

In the script we have simply extracted the package, set up the environment so both auxiliary data, executables and libraries are found, we run the driver program (**psi3**) which will in turn run the others, and finally save the results produced by the programs into the `ouput.tgz` file.

To create the input package we can use a tar command like the one we used before:

```
tar --exclude 'job*' -zcvf job.tgz .
```

And we are ready to run our job. At this point we will have on our job directory all the files needed to run the job (`job.jdl`, `job.sh` and `job.tgz`) as well as the original contents of `job.tgz` (which you could delete if you so wish). All that we need to do now is run the job using the grid commands.

Wrapping it all together

अपि पौरुषमादेयं शास्त्रं चेद्युक्तिबोधकम्।

अन्यत्त्वार्षमपि त्याज्यं भाव्यं न्याय्यैकसेविना॥

One who ever stands for reason must accept a science, though man made, if it stands to reason; and he must reject the other (the unreasonable) though it may be propounded by the sages (of yore).

From previous articles we know what should come next: we start by initiating a Grid session with `voms-proxy-init`, then submit the job with `edg-job-submit`, monitor it with `edg-job-status` until it finishes and finally retrieve results using `edg-job-get-output`.

In more recent versions of the EGEE middleware you can substitute all `edg-*` commands by their new equivalents `glite-*` (`glite-job-submit`, `glite-job-status`, `glite-job-get-output`..).

In order to simplify our users' life we can also automate this process so that a user needs not know much about the Grid: s/he may only need to set up his/her simulation or job and then use a script to take care of all the magic needed to deal with the Grid. Let us recapitulate: we have decided to

- keep each job in a separate directory
- copy a standard job.jdl inside
- create a job.sh file
- pack everything except the job.* files
- start a Grid session
- submit the job
- monitor the job
- grab job output

We can simplify things if we keep a copy of our needed utilities somewhere handy (e. g. in a `grid-skel` directory) and tell users to start the Grid session themselves and build their own `job.sh` file. Then, we could provide a driver script like the following one:

```
#!/bin/bash
# (C) José R. Valverde, 2006

# copy the master JDL file
cp /somewhere/grid-skel/job.jdl .

# build input package
tar --exclude 'job.*' -zcvf job.tgz *

# submit the job and save its ID
edg-job-submit -o job.id job.jdl

# Wait for job to finish -Done (Success
# or Fail)-
/bin/echo -n "Waiting"
edg-job-status -i job.id | grep -q
       "Current Status:     Done"

while [ $? -eq 1 ] ; do
    /bin/echo -n "."
    sleep 15
    # note that we do not test for
    # abnormal termination!
    edg-job-status -i job.id | grep -q
       "Current Status:     Done"
done

/bin/echo ""

# Get job output into local dir and
# cleanup Job details
edg-job-get-output -i job.id --dir .
# clean up
rm -f job.id
rm -f edglog.log
rm -f job.sh
rm -f job.jdl

# Move job output to local dir and
# cleanup output directory
mv -i ${USER}_*/* .
rmdir ${USER}_*

# Extract output package and clean it
tar -zxvf output.tgz
rm output.tgz
```

There are a few considerations worth noting:

- First, this is a very simple script, and can be greatly enhanced with additional error detection and recovery mechanisms, but as it is intended to be run interactively by the user, we can leave that to the user.

- We copy the `job.jdl` file from a master directory into the current one. If the master directory were in the same file system we could save some space by hard linking to it.

- We retrieve all output into the current directory and then enter some seemingly 'magic' file handling: remember that `edg-job-get-output` stores all output not directly into the current directory but within a newly created subdirectory inside the current directory; this subdirectory is named after the user name (`${USER}`) and the job ID. Since we use a separate directory for each single job, we need not worry to find out the full name of this subdirectory: there will be only one and we can use a wild card to access it.

One might conceivably go one step further and remove the need for the user to start the Grid (it's a trivial addition) and to create his own `job.sh` script: instead of telling the user about it, we can create a generic one.

```
tar -zxvf job.tgz
rm job.tgz
sh run.sh     # user provided
rm run.sh
# save everything as we do not know
# what's needed
tar -exclude output.tgz -zcvf output.
       tgz *
```

and tell the user to write only the `run.sh` script. The problem with this approach is that normally we cannot foresee which executables the user will want to run, whether they are statically linked,

if any auxiliary files or environment variables are required, etc... If we are going to tell the user about all this, we may as well ask him/her to write his/her own `job.tgz` file using our template.

Of course, *if we know in advance what is going to be used then it becomes trivial to set up a master template job* with all required files and simply ask the user to create a new directory with his or her input data: then our `grid-run` script would copy the full master job directory (including JDL, scripts, executables, shared libraries, data files, etc...) and run the job.

Summary

In this article we have seen a generic approach to simplify job management. We have used this basic approach with different variations in many of our projects, reproducing the same basic pattern in different languages (shell, PHP, Python, Perl...) and adapting it to specific needs always with success. In the next article we will deal with a seemingly trivial problem: launching many jobs to the Grid; we will see how the management of large job numbers triggers a number of issues inherent to a massively distributed infrastructure, and will present successful approaches to circumvent them.

Acknowledgements

We want to thank EMBnet[1] for making publicly available its education web site [2], and the EU for its support to projects EGEE[3] (INFSO-RI-031688) and EMBRACE[4] (LHSG-CT-2004-512092) which have allowed us to do this work.

1. http://www.embnet.org
2. http://edu.embnet.org
3. http://www.eu-egee.org
4. http://www.embracegrid.org

Grid computing (4): Wuthering heights

José R. Valverde

EMBnet/CNB, CNB/CSIC, C/Darwin, 3, Madrid 28049

Yesterday afternoon set in misty and cold. I had half a mind to spend it by my study fire, instead of wading through heath and mud to Wuthering Heights.

Emily Brönte. Wuthering Heights

This is the fourth instalment of a series of articles on Grid computing. Up to now we have learned to submit a job and monitor it, we have seen how this can be applied to various example Bioinformatics tasks and finally we saw how all the process could be automated. In this article we will go one step further, looking at better ways to automate job management for large numbers of jobs; this will require us to face seriously for the first time the possibility of job failures.

Doing many things at a time

A person who has not done one half his day's work by ten o'clock, runs a chance of leaving the other half undone.

If there's something typical of modern computing it is the ability to multitask (e. g. execute more than one task simultaneously). Similarly, Life Sciences have evolved towards a new paradigm: whereas formerly we would be analyzing one gene or protein at a time, we are now aiming to analyse and understand full genomes or proteomes (with tens of thousands of products) at once.

We have seen in previous instalments how to automate the execution of a single job. Most tasks we can perform nowadays consist of running programs developed in the "one-at-a-time" days, and are well suited for performing one single analysis, but there are still too few tools to perform genome- or proteome-wide analysis with one

single command. Further to it, many tasks are time consuming, and if we would to implement them serially (i. e. processing one dataset after the other) full analysis would take considerable time. As most datasets are analysed independently of each other, it makes sense to exploit the Grid for these *high-throughput* (HT) tasks, devoting one machine to each analysis and performing all them simultaneously (thus reducing the wall clock time needed to reach our goal).

A similar argument may be applied to other highly parallel tasks that can be performed more or less independently of each other: some illustrative examples are doing phylogenetic bootstrap analysis, performing large Montecarlo simulations (where values are generated independently starting from different random seeds) or running other high-throughput analysis (like independent docking over separate PDB entries). There are many more situations in which we can benefit from the Grid and the possibilities are only bound by your imagination.

So, the question is: how do we go about running many simultaneous jobs on the Grid?

An example: bootstrap with PHYLIP

The most trivial approach would be to run each different job one after another. This would provide little advantage over working on a single local machine (although it still may make sense for various reasons). We could do this, for instance with a very simple shell loop, possibly typed in on the command line. In this loop all we need to do is go over each job submitting it and waiting for its completion to retrieve results: as we have already seen an example shell script to launch, monitor a job and retrieve its results in the former article, we can use it now.

If all our jobs are of the same kind, only differing in the initial data set (e. g. in an HT, bootstrap or Montecarlo analysis) we may create a master job template directory containing all common required files and a driver script (as shown in preceding articles). Then, generating all the required actual job directories is a trivial task using the shell: all we need to do is either copy the master template or populate directories with hard links to the master template (to save space).

For instance, let us suppose that we want to run a phylogenetic bootstrap on a large set of long sequences using Maximum Likelihood. ML is computationally costly and if the input dataset is large enough, DNAML may take days to run for each single dataset: it makes sense to split the bootstrap into many small jobs, each with a different initial dataset instead of performing it sequentially. Now, let us assume we have a directory where we hold our aligned sequences in a file named '`input _ sequences`':

```
# First create a master job template
mkdir master_job
# Populate master job template
cp /path/to/phylip/dnaml master_job/.
# Generate driver script (run DNAML
# with default options)
cat > master_job/job.sh <<END
tar -zxvf job.tgz
dnaml <<IN
y
IN
tar -zcvf output.tgz outfile outtree
END
# We now have a template job directory
# with everything
# but the infile data.
# Let's create a thousand replicas
mkdir bootstrap
cd bootstrap
# We will need random numbers for
# SEQBOOT
# this sets a trivial seed using our
# PID
RANDOM=$$
for ((i=0; i<1000; i++)) ; do
    # we might 'cp -Rua ../master_job
       $i' to copy everything too
    mkdir $i
    ln ../master_job/* $i/.
    # create 'infile' with only 1
replicate using
    # SEQBOOT and a random seed
    seqboot <<ENDSB
../../input_sequences
r
1
y
$RANDOM
ENDSB
    # move 'outfile' with the
    # replicate(s) to 'infile'
    # in the job directory
    mv outfile $i/infile
done
```

A short explanation is due here: the above instructions have created a master template directory (`master_job`) and populated it with the program we want to run (DNAML) and an appropriate driver script (which runs DNAML feeding it the basic input to analyse a single dataset using default options). Then we simply enter a loop that generates one thousand jobs, naming them by job number. Each directory contains a copy of the master files for the job (in our case we have chosen to use hard links to save space) and its own, different input file (which we generate using SEQBOOT with the appropriate input). If you want to compute more replicates per job then you simply change the number (1) given to SEQBOOT by whatever you want and modify the input to DNAML (or whatever program you intend to run) accordingly. Note that if you want to save space, hard links should be used: this is so because when we later use '`tar`' symlinks will be packed as such and the actual file contents will not be included in the job package, and may be a little bit more savvy than using symlinks.

Now, running the bootstrap executing each job sequentially becomes a trivial exercise:

```
for i in * ; do
    cd $i
    sh grid-execute.sh
    cd ..
done
```

This simply visits all subdirectories and runs their jobs, one after another. The script '`grid-execute.sh`' would be similar the last script we described in our previous article, which submits the job, monitors it and retrieves the output.

When this is finished we will have 1.000 directories, each with its own '`outfile`' and '`outtree`' computed for a different input dataset (generated by SEQBOOT), and all we need is to concatenate the trees together and use CONSENSE.

Exploiting Grid parallelism

In the previous section we took care of the most complex thing: generating the job directories. It looks complex on paper, but if you actually try it, you will see that it is very easy to do. However in the trivial case we have shown, we simply run jobs sequentially, one after the other, not unlike running DNAML with multiple datasets. We can do better than that: we can have each job run independently on a different CPU on the Grid, all of them simultaneously.

The easiest way to do this is to simply spawn each job control script as a background process:

```
for i in * ; do
    cd $i
    sh grid-execute.sh &
    cd ..
done
```

Now we will have a thousand copies of the monitor script, one for each job directory, submitting and following each job in parallel.

This is all too easy and certainly works. It has some drawbacks though: first it requires a separate monitoring process for each remote job in our local computer. For 1.000 jobs this is already a heavy penalty which slows down work on our machine (affecting other users), but if we want to run even more jobs (say 10.000 or 100.000) it may saturate our local machine capacity. Worst, these processes eat resources (at least some amount of memory) even though most of the time they are sleeping waiting for their remote jobs to finish, thus wasting valuable resources. And even worst yet, these processes will be spawned very quickly and compete for access to Grid resources (submission, monitoring and retrieval) all at the same time, saturating communications with the Grid.

This approach will usually work well for small numbers of jobs, but for large numbers of jobs it is wasteful, inefficient and disturbing to other users. In that case, it is better to use a different approach: it makes more sense to serialize each different step in the monitoring process. In other words: we use first a loop to submit all jobs, then we use a second loop to monitor them and retrieve results if finished:

```
#!/bin/bash
#
for i in * ; do
       cd $i
       ln ../../master_job/job.jdl .
       ln ../../master_job/job.sh .
       edg-job-submit -o job.id job.jdl
       cd ..
done
```

```
#!/bin/bash
#
for i in * ; do
       cd $i
       ln ../../master_job/job.jdl .
       ln ../../master_job/job.sh .
       edg-job-submit -o job.id job.jdl
       cd ..
done
# now monitor all jobs (check 24 times
# at one hour intervals)
for ((hour=0; hour<24; hour++)) ; do
    for i in * ; do
        edg-job-status -i job.id | grep
       -q "Current Status:     Done"
        if [ $? -eq 0 ] ; then
            edg-job-get-output -i job.
             id --dir .
            mv -i ${USER}_*/* .
        fi
    done
    sleep 3600 # wait one hour
done
```

The trick here is to define a maximum amount of time for all processes to exit and test their status periodically (e. g. at one hour intervals) instead of waiting for all to finish before retrieving results. There are compelling reasons for this as we will see later.

Looks simple enough, doesn't it? Well, it is, and we have used it, sometimes even issuing the commands by hand to the shell to control jobs. It is inefficient, though, at various steps, most notably in the retrieval step, where if a job is finished we may try to recover output more than once. But we will not worry more about it now.

Facing the storm

Having levelled my palace, don't erect a hovel and complacently admire your own charity in giving me that for a home.

Up to now, everything seems OK, we have done that many a time, and our experience has led us to conclude that, while initially correct, this approach requires many further improvements. Why so? Because we want to run many, many jobs, that's why.

See, when you run a single job, either manually or using a driver script like the one described in our previous article, there is not much need to worry about problems: in the unlikely event that something goes wrong you will notice and all you need to do is run the job again. But now we are talking many jobs. This implies two things: first is that however unlikely that something goes wrong, the odds now are higher (multiply by the number of jobs) and second that manual recovery is a real hassle because of the sheer number of jobs.

Still, we have been able to use this approach by simply running the jobs and only afterwards looking for failed jobs and restarting them. With minor modifications the same procedure used to launch many jobs can be used to launch aborted ones. However, it would be better if we could deal with most of these problems automatically.

```
# An improvised command line for
# 'ad hoc' job recovery

for i in * ; do cd $i ; grep -q Done
status ; if [ $? -ne 0 ] ; then rm job.
id ; edg-job-submit -o job.id job.jdl ;
fi ; cd .. ; done
```

At this point you may wonder if there really are problems to worry about. Well, yes, the Grid is a distributed infrastructure and things may go wrong for a number of reasons:

- Job submission may hang sporadically due to network problems: in this case, job submission hangs indefinitely (due to the way the submission protocol is implemented) and then all your job submission process stalls. This usually happens about 1 in every 10.000 jobs.
- Resource allocation to your jobs requires global Grid awareness on the resource broker (RB) node, and this awareness is network costly, taking a decision is computationally expensive as well. When you submit too many jobs to a single RB it saturates easily and gets inefficient. Further, what if your network connection to the RB fails for any reason or it is not available? All your jobs would stall until the RB recovered.
- Job execution at the working nodes may fail too: perhaps the WN is faulty, or may be it has not enough memory, or it was shutdown midway in the execution, or simply your job failed by itself (e. g. a program bug), or it exceeded some queue or resource limit. Your job will end with an Aborted status. Job failure rate my sometimes grow to as much as 10% of your jobs (even more in special cases).

- Sometimes jobs get lost in the nether world. This may happen because of a number of reasons (like network failures or Grid inconsistencies) and will result in your job never reporting a finish status. In practice it will be as if your job had never terminated, was eternally running. This typically happens about 1 in every 10.000 jobs.
- Occasionally some jobs end with a success status but their output is lost somewhere in the Grid and cannot be recovered.
- Other various kinds of failures may happen, but most often they will result in symptoms similar to those above.

Now you see why we called this article as we did: even if whatever may go wrong is unlikely, when we submit many jobs it will sooner or later affect us. Wouldn't it be nice if we dealt with this automatically?

Uttering the magic spells

If he loved you with all the power of his soul for a whole lifetime, he couldn't love you as much as I do in a single day.

In what follows we will explain the basic tricks that we use to deal with the above problems. We will not be providing full listings as this would make this article unsuitably large. If you want to get the full, tested, production scripts and save yourself the work, you are welcome to download them from our web site[1].

Dealing with submission problems

The tyrant grinds down his slaves and they don't turn against him, they crush those beneath them.

We have already mentioned that sometimes submission hangs indefinitely due to network problems. This is very easy to spot and fix: usually submission is a very quick process, and while it depends on the amount of data needed to be transferred for a given job it usually falls in the range of 10 seconds to 1 minute. The simplest approach is to define a timeout and consider that any submission taking more than the timeout time has stalled and needs to be stopped and retried again. Using the shell we would do it as follows:

```
# Maximum time in seconds to wait for
# a submission to occur
timeout=180
# Maximum number of attempts to submit
# a job
maxtries=5
for ((i = 0; i < $maxtries; i++)) ; do
       # $! is the PID of the most
       # recently spawned background
       # command
       (   submitter=$!
              # myself (I've just been
              # spawned)
           # start timeout watchdog
                 (  sleep $timeout
              pkill -P $submitter
           )&
           edg-job-submit -o job.id
              job.jdl
       ) &
       wait $!
done
```

What we are doing here is a bit convoluted because of the way the shell processes PIDs, but basically it is simple to understand: we want to kill a submission that takes too long, for which we create a separate watchdog process. The watchdog is simply a subshell that waits for some time and sends a KILL signal. Since the watchdog is a separate process and needs to be started before the submission takes place, it cannot know the ID of the `edg-job-submit` process, so the only way it can stop it is if both, the watchdog and the `edg-job-submit` command are siblings, sons of the same father, and we kill all our parent's offspring (pkill -P). Thus we have for the watchdog and `edg-job-submit` the simple code

```
(sleep $timeout; pkill -P $submitter) &
edg-job-submit -o job.id job.jdl
```

However, if both of these processes were executed directly, we might also kill our submission script, and so both of them must be executed within another subshell, so that when the watchdog kills its (intervening) parent it stops itself and `edg-job-submit`, but not the whole script. That is why we must use

```
(   submitter=$! # find my ID
    ( sleep $timeout ; pkill -
       P $submitter) & # kill the
       parent subshell and kids
   edg-job-submit -o job.id job.jdl
) &
wait $!
```

The construct '`( ... )& wait $!`' allows us to generate an independent subshell (`&`) and wait for it to complete. This is needed because if we omit the `&` bash will not create a subshell and instead execute everything directly.. leading the watchdog to kill the whole script, and if we don't wait, all submissions will be attempted simultaneously (which is what we want to avoid).

Dealing with resource management

Treachery and violence are spears pointed at both ends; they wound those who resort to them worse than their enemies.

The next issue to consider is how to deal with a broken RB, and being nice, how to avoid overloading any single RB obtaining as a by product possibly better resource management and less hassle to other users. The basic approach to deal with this is to spread the load of job submission over different RBs, as many as possible. In order to do this, we need to know which RBs are available, which we can find out using the `lcg-infosites` command:

```
# get list of available RBs into
# an array variable
lcg-infosites --vo biomed rb | sed -e
       's/:7772//g' > $base/rb.list
rblist=( $(cat "$base/rb.list") )
rbno=${#rblist[@]}
```

With this code we have selected all available RBs and stored them in a shell array. We may now use this array to drive submission. For this we need to massage two special configuration files that will direct `edg-job-submit` to a given RB chosen at random from the available list. The trick now is to first generate these two files from a template using '`sed`' and then submit the job. If job submission fails we assume that the RB we used is broken and remove it from the list. As an additional bonus, we may check if the list is empty and if so, reset the array of valid RBs to the initial full list:

```
# Re-read the list of valid/available
# RBs
# it may have changed if any one failed
rblist=( $(cat "$base/rb.list") )
rbno=${#rblist[@]}
# select a random number in the range
# 0 - $rbno
#      To avoid flooding and
# overloading a single RB
no=$RANDOM
let "no %= $rbno"

# set the target RB to ${rblist[$rbno]}
sed -e "s/%RB%/${rblist[$no]}/g" $base/
       rb.conf > $job/myrb.conf
sed -e "s/%RB%/${rblist[$no]}/g" $base/
       rb.vo.conf > $job/myrb.vo.conf

# actually submit the job
cd $job
edg-job-submit -c myrb.conf --config-vo
myrb.vo.conf -o job.id job.jdl
# save exit status
status=$?
cd ..
# edg-job-submit exits with 0 on
# success, >0 on error
#      check exit status and if failed,
# remove RB from list
if [ $status -eq 0 ] ; then
    exit $status
else
    mv $base/rb.list $base/rb.list.old
    # remove failing RB
    grep -v ${rblist[$no]} $base/
       rb.list.old > $base/rb.list
    # check if list is empty
    rbno=`cat $base/rb.list | wc -l`
    # if so, reset and hope for the
    # best
    if [ $rbno -eq 0 ] ; then
          #echo "resetting list of RBs"
          lcg-infosites --vo biomed rb
              | sed -e 's/:7772//g' >
              $base/rb.list
    fi
    exit $status
fi
```

We can combine the two tricks (a watchdog timeout and RB rotation and recovery) to produce a safer submission script.

Recovering failed jobs

You said I killed you - haunt me, then! The murdered do haunt their murderers, I believe.

This is initially easy: we just check the job exit status and if it is "Aborted" then resubmit it. As we are going to deal with many jobs, we would like to make it more efficient to avoid repeating useless queries to the Grid for already finished jobs. All we need to add is some kind of persistence recording the status, like saving it on a file or shell array.

To make recovery more efficient and reduce Grid load, we simply add a few tricks: first, we check if a file recording the status of a job exists and if it does, we read the contents to avoid queries on successfully terminated jobs. In all other cases we check the status and act accordingly. We also need a policy on resubmissions: we might keep on resubmitting failed jobs until we succeed, but if the problem lies in our job (e. g. a software bug) then we will never notice as it will be retried forever. Thus it is best to define a maximum amount of retries so that anomalous cases can be investigated further. As each job is independent and may fail (or not) at any time, we need to keep track of retries on a per job basis, for which having some persistence is again convenient.

Our recovery check for aborted jobs might thus look like

```
for i in * ; do
       if [ ! -d $i ] ; then continue ;
fi
       cd $i

       # check if we have any status
       # report
       if [ ! -f status ] ; then
                 edg-job-status -i job.
                     id > status
              fi
       # locate aborted jobs and
       # restart them
       grep -q Aborted status
if [ $? -eq 0 ] ; then
              edg-job-submit -o job.id
              job.jdl
              # increase retry count on
              # file
              try=`cat retry.cnt`
              let try++
              echo $try > retry.cnt
       fi
# .....
done
```

Restarting mute jobs

Any relic of the dead is precious, if they were valued living.

Another problem that can happen is that output recovery fails. Since we are saving the Grid termination status on a file, we may check for successfully finished jobs that have generated no output. For this we need some way to check output, usually the existence of a file that we know for sure should have been generated by the execution of our command. In our PHYLIP example, it might be '`outfile`': if it does not exist and the job executed OK, we know the output has been lost. Checking for this case would be similar to the above, but we now check for successful jobs instead of aborted ones and verify the results:

```
grep -q Done status
       if [ $? -eq 0 ] ; then
           if [ ! -s 'outfile' ] ; then
              # resubmit job
            edg-job-submit -o job.id
              job.jdl

              # increase retry count
              try=`cat retry.cnt`
              let try++
              echo $try > retry.cnt
           fi
       fi
```

Detecting immortal jobs

This last case is the most difficult to manage. These are jobs that were submitted correctly and started execution (their status reached the "Running" stage) but which otherwise disappeared from the Grid, being left forever in a "Running" state. They may have died, aborted or succeeded but their Grid status has not been updated and will never be.

Why is this such a difficult problem? In the general case it may be too difficult to predict when a given job should finish: in some cases job duration may vary widely from one instance to another. But even if all jobs were more or less homogeneous (like in our bootstrap example) there is still a large variability component stemming from differences at the working nodes: job duration will depend on WN specs (CPU type and speed, memory available, etc...), queue priority at the WN and workload (one WN may support various queues at different priorities for various kinds of jobs or VOs and simultaneous execution of several jobs). And even so, we are not guaranteed when a job will start executing (if the Grid is overloaded, it might be delayed waiting on queue for an indefinite amount of time). The bottom line is that in general it is difficult to predict exactly how much time a job will take on the Grid.

Not all is lost however: most times we will have an estimate of how much we can expect a job to take, or at least how much we are willing to wait for a job to finish. The simplest solution, and probably the most advisable one in general (as this is something that happens only about 1 in every 10.000 jobs) is to just ignore the problem and let the user know when after some reasonable time there are still unfinished jobs, providing the user with a method to cancel those and resubmit them.

If we want full automation, and can define an upper bound, then it is a trivial thing to automate following our previous examples: submit all jobs and monitor them until the time limit is reached and after time has completed, kill and restart any pending jobs.

The problem with a kill and restart strategy is that some times it may really happen that a job actually takes that long to finish... or that we cannot be sure and must wait. In these cases, another successful approach is to resubmit those jobs without cancelling the original one and wait for the first to finish. If the job was immortalized, the replica will terminate, if not, either of them will, depending on luck..

An extreme approach would be to replicate all jobs from the onset one or more times, waiting for the first to finish and killing all others once the output of one instance has been successfully retrieved. This is most interesting not to prevent immortal jobs but when we have not that many jobs but want to ensure quickest termination: it may be wasteful of resources but ensures you always get the fastest response from the quickest replica to terminate.

Wrap up

We have seen a number of problems and solutions that will prove useful if you want to run large numbers of jobs on the Grid. All of us have started with simple tasks and moved on to bigger problems, and all of us have had to face these same problems. Knowing them in advance and having a tool set of solutions to deal with them will enable you to understand what is going on when things get awry and quickly apply the fix for them.

Although the examples we have shown use the `bash` shell to automate all the work, the basic concepts laid out work exactly the same on any other setting: should you be using other scripting language, or a compiled or interpreted one, the approach is the same (you may need to use threads instead of processes for example but conversion should be easy). The main reason we have used the shell is one of generality: almost any language sports a `system()` call that executes a command under the `sh/bash` shell so that if your chosen development language falls short on any need you can always resort to use these tricks directly by invoking the shell from `system()`.

Eventually, the middleware -which is in continuous evolution- will solve these problems, or even provide automatic tools to deal with large job numbers safely, but you should not wait till them to make your dreams come true: now you know enough to go forward. In our case we have not finished our example, it is not a complete, working program and if you want to implement it you will need to complete it yourself. Most notably, we have ignored the final step to collect all separate output files (a simple '`cat */outtree > treesfile`') and to run CONSENSE. As a final advice, and as you can see this is not fit for the average John Doe, and so, whenever you decide to develop large projects, our counsel is that you also invest some time in making it easy to use for wet lab users.

From here it is only a matter of creativity and imagination to find problems requiring large numbers of jobs that can be adapted to the Grid. But, if there is something we can be sure, is that human ingenuity is difficult to drain.

Acknowledgements

We want to thank EMBnet[2] for making publicly available its education web site [3], and the EU for its support to projects EGEE[4] (INFSO-RI-031688) and EMBRACE[5] (LHSG-CT-2004-512092) which have allowed us to do this work.

1. http://www.es.embnet.org/~jr/download/ht-grid.tgz
2. http://www.embnet.org/
3. http://edu.embnet.org/
4. http://www.eu-egee.org/
5. http://www.embracegrid.org/

The selfish smell

Vivienne Baillie Gerritsen

We are surrounded by smells. Pleasant ones and not so pleasant ones, hard to distinguish ones, mild ones and strong ones. Smells are not part of our everyday life for the simple sake of pleasure. They are there for a purpose. The perfume of a flower can be used as an attractant for a potential pollinator, for instance. The scent given off by a poisonous mushroom is a way of warding off a predator and, by the same token, can be instantly recognised as toxic by an animal, thereby saving both species. Special scents are also given off by males and females when mating is in the air, and no wine grower will ever argue that a wine's fragrance is not for the sole purpose of seduction. But what is a smell? More often than not, a scent is made up of a mixture of odorant molecules which, together, will trigger off a complex olfactory system that will ultimately let us perceive it and, if we wish to, put words to it. The very first step in such a system involves an odorant receptor to which an odorant molecule binds. Recently, a new human odorant receptor – OR7D4 – was discovered. OR7D4 is special in that it is the first receptor known to respond to a specific odorant molecule.

'Woman smelling coffee' by Gizem Saka

Courtesy of the artist

Discovering an odorant receptor is nothing new. Major discoveries of the like were made almost twenty years ago. What is singular, though, is that for the first time scientists are able to link an odorant receptor to a specific ligand. This is a breakthrough because, unlike our visual and hearing senses which are relatively straightforward, our olfactory senses are part of a highly complex system. We know what light frequency will give which colour. And what sound pitch will give which sound. But no one can say 'this molecular structure will give that smell' – which would be the perfume maker's Holy Grail. What is more, knowing a molecule's structure is not sufficient. Any smell is the combination of more than one odorant molecule.

There are thousands of different smells and we are capable of discerning each one of them. Yet the human capacity to sniff out stuff has slackened over the millennia because we have put both our hearing facilities and our visual capacities to a greater use. As a result, our olfactory system is now one third of what it probably was in our ancestors. A rat, for instance, expresses about one thousand olfactory receptor genes, while we only express about three hundred. These olfactory receptors are found in ciliary membranes immersed in a film of mucus which lines the inner side of our nose. The cilia are the tips of sensory neurons which relay a smell along their axons to the brain. The brain will either recognise the smell it has just received – and consequently so will we – or it will discover a new smell and memorise it for the next time.

To cut a long story short, a specific scent is made up of a certain number of odorant molecules. Each molecule will bind to a specific receptor found on the surface of a sensory neuron. One molecule can bind to more than one species of receptor. Likewise, one receptor is able to bind more than one molecule. Consequently, one smell can set a whole network of neurones shivering and relaying messages to the brain. The brain sums them all

up and can either come up with an immediate answer such as 'coffee' or, if the smell is unknown, whatever it is we have just smelled will be remembered and a direct link will be made to it the next time we get a whiff of it.

OR7D4 is an olfactory G-coupled receptor similar to all those known to date. It is a transmembrane protein, with an extracellular loop, which acts as the binding pocket for its ligand, and a cytoplasmic domain which reacts with the G-protein. Androstenone, which is an odorous steroid derived from testosterone, lodges in the OR7D4 binding pocket thereby changing the receptor's conformation. This triggers off the formation of cAMP thanks to the G-protein, which in turn opens a channel protein. The opening of the channel lets in cations which change the neurone's membrane potential, and this is the very beginning of an electric signal which is relayed, along the sensory neurons' axons, all the way to the brain that will read the message as being part of a smell.

Discovering a ligand's receptor is one thing. Attributing an actual smell to a specific ligand is another. It seems to be the case with androstenone though. This is a smell which is usually perceived as unpleasant (urine- or sweat-smelling), pleasant (sweet- or floral-smelling) or without a smell. That would make three smells, you're thinking. Not really. It is one smell perceived differently. What is it that makes people smell a smell differently? Is it innate? Or does it have something to do with something far less tangible, such as personality, past experience, or even the subconscious? Many will answer that, though there is no doubt a genetic basis, much must be due to something which is not inherited. In the case of androstenone, however, there may be a case of an inherited difference in perception. Indeed, there is a common variant of the OR7D4 receptor and it seems that individuals that carry the same variant tend to smell androstenone in the same way…which would point to the genetic inheritance of a smell…

A lot of time and money is spent cracking the scent code. Besides the perfume industry, there are plenty of other industries that need smells to sell. Food, cosmetics, washing-up powders and beverages – to name a few – all make use of the power of fragrance. The industry has even learned how to trick our smelling senses. Take synthetic lemon juice, for instance, which only uses a few of the chemicals found in natural lemon scent and yet we can identify it as lemon. A handful of scientists have suggested that a crude quality of a smell, such as its pleasantness or unpleasantness, is simply a case of molecule compactness. So the more compact an odorant molecule is, the nicer it might smell. However, it is also known that one same odorant molecule can smell nice when there isn't too much of it, and awful when there is a lot. Take indole for example which is an odour molecule found abundantly in faeces and yet it is used in very small amounts in perfumes… Inherited or not, purely for our survival or not, what a smell will never be able to take away from us are the beautiful memories – and no doubt the bad – they are able to stir in us.

Cross-references to Swiss-Prot

Olfactory receptor 7D4, *Homo sapiens* (Human) : Q8NG98

References

1. Keller A., Zhuang H., Chi Q., Vosshall L.B., Matsunami H.
Genetic variation in a human odorant receptor alters odour perception
Nature 449:468-473(2007)
PMID: 17873857

2. Trivedi B.
Smells rank: Solving a stinker of a problem
The New Scientist Magazine, 17 November 2007

3. Hatt H.
Molecular and cellular basis of human olfaction
Chemistry and Biodiversity 1:1857-1869(2004)
PMID: 17191824

National Nodes

Argentina
IBBM, Facultad de Cs. Exactas, Universidad Nacional de La Plata

Australia
RMC Gunn Building B19, University of Sydney, Sydney

Austria
Vienna Bio Center, University of Vienna, Vienna

Belgium
BEN ULB Campus Plaine CP 257, Brussels

Brazil
Lab. Nacional de Computação Científica, Lab. de Bioinformática, Petrópolis, Rio de Janeiro

Chile
Centre for Biochemical Engineering and Biotechnology (CIByB). University of Chile, Santiago

China
Centre of Bioinformatics, Peking University, Beijing

Colombia
Instituto de Biotecnología, Universidad Nacional de Colombia, Edificio Manuel Ancizar, Bogota

Costa Rica
University of Costa Rica (UCR), School of Medicine, Department of Pharmacology and ClinicToxicology, San Jose

Cuba
Centro de Ingeniería Genética y Biotecnolgía, La Habana

Finland
CSC, Espoo

France
ReNaBi, French bioinformatics platforms network

Greece
Biomedical Research Foundation of the Academy of Athens, Athens

Hungary
Agricultural Biotechnology Center, Godollo

India
Centre for DNA Fingerprinting and Diagnostics (CDFD), Hyderabad

Israel
Weizmann Institute of Science, Department of Biological Services, Rehovot

Italy
CNR - Institute for Biomedical Technologies, Bioinformatics and Genomic Group, Bari

Mexico
Nodo Nacional EMBnet, Centro de Investigación sobre Fijación de Nitrógeno, Cuernavaca, Morelos

The Netherlands
Dept. of Genome Informatics, Wageningen UR

Norway
The Norwegian EMBnet Node, The Biotechnology Centre of Oslo

Pakistan
COMSATS Institute of Information Technology, Chak Shahzaad, Islamabad

Poland
Institute of Biochemistry and Biophysics, Polish Academy of Sciences, Warszawa

Portugal
Instituto Gulbenkian de Ciencia, Centro Portugues de Bioinformatica, Oeiras

Russia
Biocomputing Group, Belozersky Institute, Moscow

Slovakia
Institute of Molecular Biology, Slovak Academy of Science, Bratislava

South Africa
SANBI, University of the Western Cape, Bellville

Spain
EMBnet/CNB, Centro Nacional de Biotecnología, Madrid

Sri Lanka
Institute of Biochemistry, Molecular Biology and Biotechnology, University of Colombo, Colombo

Sweden
Uppsala Biomedical Centre, Computing Department, Uppsala

Switzerland
Swiss Institute of Bioinformatics, Lausanne

Specialist Nodes

EBI
EBI Embl Outstation, Hinxton, Cambridge, UK

ETI
Amsterdam, The Netherlands

ICGEB
International Centre for Genetic Engineering and Biotechnology, Trieste, Italy

IHCP
Institute of Health and Consumer Protection, Ispra. Italy

ILRI/BECA
International Livestock Research Institute, Nairobi, Kenya

LION Bioscience
LION Bioscience AG, Heidelberg, Germany

MIPS
Muenchen, Germany

UMBER
School of Biological Sciences, The University of Manchester,, UK

for more information visit our Web site

www.embnet.org

EMBnet.news
ISSN 1023-4144

Dear reader,

If you have any comments or suggestions regarding this newsletter we would be very glad to hear from you. If you have a tip you feel we can print then please let us know. Please send your contributions to one of the editors. You may also submit material by e-mail.

Past issues of EMBnet.news are available as PostScript or PDF files. You can get them from the EMBnet organization Web site:
http://www.embnet.org/download/embnetnews

Publisher:
EMBnet Executive Board
c/o Erik Bongcam-Rudloff
Uppsala Biomedical Centre
The Linnaeus Centre for Bioinformatics, SLU/UU
Box 570 S-751 23 Uppsala, Sweden
Email: erik.bongcam@bmc.uu.se
Tel: +46-18-4716696

Submission deadline for the next issue:
August 20, 2008