Using the Grid to run population dynamics simulations

José R. Valverde
EMBnet/CNB, Centro Nacional de Biotecnología, CSIC.
C/Darwin, 3. 28049 Madrid. Spain

Abstract
Analysis of population evolutionary dynamics using realistic models is a challenging task requiring access to huge resources. Estimates for simple models of population growth under different mutation and selection conditions yield running times of Central Processing Unit (CPU) years. As mutations are stochastic events, experiments can be split into many separate jobs, reducing to a large Monte Carlo-like problem that is embarrassingly parallel and thus maps perfectly on the Grid.
We have been able to run simulations with realistic population sizes (up to 1,000,000 individuals) and growth cycles using the Grid with a ~190x efficiency gain, thus reducing execution time from years to a few days. This speed-up allows us to accelerate the simulation cycle, and work on data analysis and additional model refinements with minimal delays and effort.
We have taken measures at various steps in the process to study the efficiency gains obtained. While our simple approach may arguably be far from achieving optimum efficiency, we were able to achieve significant gains. Here, we analyse Grid efficiency and discuss which benefits can be realistically expected with the current technology; we also provide useful advice for future Grid developers.

All the tools described are available under GNU's Public License (GPL) from http://ahriman.cnb.csic.es/sbg/tiki-download_file.php?fileId=16

Introduction
Building realistic population simulations is a typical embarrassingly parallel large-scale computation. This kind of problem maps naturally to massively distributed architectures, like the EGEE Grid[1] (Enabling Grids for E-science in Europe). Solving this instance therefore provides solid ground both for solving other similar tasks and for testing the adequacy of current technology.

Our main interest was to study the selection processes taking place in bacteria with different mutation rates. The problem of itself is interesting for many reasons: from a theoretical point of view, it is a simplified model of the evolution of more complex organisms and ecosystems; but it also has relevant practical implications to further our understanding of population dynamics, evolution and mutation rates, and to understand the development of interesting traits, like bacterial resistance to antibiotics.

There is a relevant interest in solving, or at least understanding, the problem in detail; however, while growing a bacterial population in the laboratory is cheap routine work, analysing the evolution and selection of gene mutations experimentally is not so simple, as it would require genotyping of representative samples of bacterial populations and assessment of the impact of each selected genotype on the viability of its carrier (Sniegowski *et al.*, 1997).

Because experimental validation is inconvenient, it is desirable to model *in silico* what would happen in the test tube. The main problem now is being able to produce realistic simulations: as cell division is an exponential process, we soon find ourselves modelling large numbers of specimens, whose mutation events must be tracked, and we need to collect statistically significant data.

Running these simulations has largely been constrained by technological limitations, resulting in reductionist models that (despite their shortcomings) have harvested useful insights on the problem (Wilke *et al.*, 2001; Lenski *et al.*, 1999; Adami *et al.*, 2000; Taddei *et al.*, 1997; Johnson, 1999). Despite Moore's law, running a realistic simulation easily results in very long computation times, limiting its usefulness. More specifically, our estimates for the simulation we wanted to run were in the order of years of CPU time.

Our simulations use a Monte Carlo method: we repeat a basic experiment enough times to collect statistically sound results. Additionally, because each simulation experiment is independent from all others, by simply using a different seed, our approach may be generalised to any embarrassingly parallel system with a large number of non-communicating tasks.

Finally, because simulated population growth is affected by mutation rates and the effect of random mutations on viability, varying initial conditions have a large impact on population size during the simulation, resulting in large variability of simulation run times, posing additional challenges and making ours a problem of more generic interest.

1 www.eu-egee.org

This paper deals with the implementation details of these simulations on the Grid. Our population dynamics simulations are still being further refined, although preliminary results from the analysis involving various combinations of different mutator phenotypes, selection coefficients and mutation rates led to two main scenarios, demanding more extensive analysis; these were presented as part of the 2007 Workshops, Current Trends in Biomedicine series, "*Stress, stress responses and mechanisms of evolvability*" at the Universidad Internacional de Andalucia, Baeza, Spain, 2007, and will be fully discussed once the analysis and experimental verification have been completed in a separate publication.

Methods

Simulation code

The population dynamics simulation was based on in-house code written in Fortran95, requiring no additional libraries or dependencies. The long run-times required for a realistic simulation necessitated the problem to be split into sub-problems suitable for running on the EGEE Grid. All programs were compiled statically using the Gfortran compiler to avoid library dependencies on remote hosts.

Each experiment tests a set of constraints under a large variety of initial parameters (up to 1,000), executing a sensible number of simulated culture cycles (up to 100). The initial model simulated laboratory conditions, using in each culture cycle an inoculate of individuals with several genes, taken from a previous culture, that would undergo many replication, mutation, competition and selection events until a sensibly large colony size (usually of the order of a million individuals), or number of replication events, was reached.

Output of each simulation run was used to further refine and optimise the initial model, making it more meaningful. This refinement process is still an ongoing concern.

Owing to the large variation of constraints, run-times also show large variation, as may be expected: a population suffering more deleterious mutations grows less, its reduced number of individuals resulting in lesser simulation resource and time requirements.

Grid parallelisation

The simulation was conducted to mimic many *in vivo* experiments under controlled starting conditions. Because mutation is a stochastic process, we could split work into separate runs using different random seeds. To manage jobs, we developed tools that have been progressively refined to adapt to various issues and shortcomings.

The job-management scripts were developed as shell scripts, and can be coarsely classified into three categories: a set of scripts to generate the large number of jobs required; a set of generic scripts to launch jobs, monitor their status and collect results; and a set to process the results into manageable statistics.

Job management was designed as a set of generic scripts that can be used for any kind of non-specific job: the system expects all jobs for an experiment to be collected in a single directory, with each job being stored in a separate, self-contained sub-directory with all data and software needed for the computation. Submission works by traversing all job sub-directories, making links to generic Job Definition Language (JDL) and execution script files, and independently sending each job to an appropriate resource broker. Failure recovery involves traversal of the job sub-directories to search for aborted, failed or silently dead jobs and resubmitting them up

```
Type = "job";
JobType = "normal";
VirtualOrganisation = "biomed";
Executable = "job.sh";
StdOutput = "std.out";
StdError = "std.err";
InputSandbox = {"job.sh", "program", "input"};
OutputSandbox = {"std.out","std.err", "result.dat"};
```

Figure 1. A typical job.jdl file may be as simple or complex as needed.

```
#!/bin/bash
#
chmod 755 program
./program < input
```

Figure 2. A typical job.sh script.

to a maximum number of tries. Data collection checks job status for successful termination and retrieves the output from the Grid into the job directory. The whole process is managed from a higher-level script that controls the timing of submission, failure recovery and output retrieval until all jobs have successfully finished.

With generic job management in place, it is now easy to automate generation of the large numbers of jobs required: only a generic execution script and JDL file need to be written, and copied by the submission system to the job subdirectory; and a simple script or shell loop-command are also needed to create the job sub-directories, copy (or better, hard link to save space) any common files, and generate any specific files depending on job parameters (Figures 1, 2 and 3).

Data collection and analysis were similarly performed by a set of scripts or shell commands: all that was needed was a loop traversing every job sub-directory and parsing output to extract relevant information.

Execution of data collection

In order to assess the impact of Grid architecture on the efficiency gains obtained, we inserted in our code specific instructions to collect timing data at various key steps, so that we could measure the time invested at each step and investigate its influence on overall performance. The steps chosen were as follows: start and end of job submission (s_0, s_1); start and end of job execution (e_0, e_1) at the Working Node (WN); detection of job termination/start of result retrieval, and end of result retrieval (r_0, r_1).

Collecting times on the Grid requires additional care, as different steps will take place in different time zones. We took advantage of the fact that the Grid has a universal time and clock synchronisation, and measured time in Universal Coordinated Time (UTC) to avoid local offsets.

Another issue worth considering is the underlying WN architecture, as different machines may lead to different execution speeds. While this is intuitively true, we didn't consider it because it must be coupled to an unknown factor: a given WN may be simultaneously running more than one job at different priorities, hence, perhaps counter-intuitively, a loaded high-speed computer might perform worse than an old slower machine. Because there is no way to know which other tasks a given node is executing, at what priority, or for how long they overlap our job, this issue was not dealt with.

As our programs were compiled only for a 32-bit architecture, we also did not examine architecture-specific (64- vs. 32-bit) differences.

```
for i in {10..50..10}; do
    for j in {1..20}; do
            job=$i-`printf %02d $j` ;
            mkdir $job
            cd $job
            ln ../../exe/program .
            echo "$i $j" > input
            cd ..
        done
    done
done
```

Figure 3. A typical job-generation command.

Results

Choice of computing system

From preliminary measures, we expected full experiment simulations to need from one to several years of CPU time for each experiment. This prompted us to seek other alternatives. Our two main options were the *Marenostrum* massively parallel supercomputer and the EGEE Grid. We opted for the Grid owing to its simplicity and immediate availability.

The problem reduces to a very large Monte Carlo simulation of mutation events on a dynamically growing population. We could further simplify the simulation by dividing it into separate growth cycles, much like one would do in laboratory practice.

Running one simulation on the Grid

We first tried to shift the parallel/serial balance towards computation by trying to fit all growth cycles for a given parameter-set in one process. One experiment would therefore require as many jobs as different initial conditions (hundreds). Each job was submitted and monitored separately.

This results in many sleeping processes waiting on the system for their monitored jobs to terminate, to the detriment of other concurrent users. Moreover, we observed that a discouragingly high number of jobs (~40%) aborted on execution. Investigation showed that *many sites maintain short-lived batch queues with execution times of 72 hours or less*. Because our problem could be further split with little extra work, we therefore decided to generate a larger number of shorter jobs.

Running a large simulation on the Grid

Next, we selected a job size that would ensure all jobs would run within the minimum queue lengths. Thus, instead of simulating 100 independent cycles for each set of initial conditions, we ran 10 jobs of 10 cycles, each requiring between 8 minutes and 8 hours.

We then changed job management to launch all the jobs at once and use a daemon that would periodically check job status, retrieve results, if complete, or resubmit if aborted, looping for a reasonable time to ensure all jobs had a chance to terminate. With the new approach, we achieved success rates of 90% and analysed the rest to determine the reasons for failure.

The most concerning kinds of failure were *unspecified job failures*. As there is very limited information on these failures, and they are relatively infrequent, there is little else to be done besides re-starting them. A special kind of problem that appears about one in every 9,000 jobs is that *job submission hangs indefinitely*. A more worrisome anomaly is *immortal jobs*. These are jobs that remain in 'Running' status indefinitely, even after Grid-execution permissions have expired, probably because the job termination notification has been lost. Finally, we were made aware of a side-effect of our approach on other users: while we had reduced the load on our front-end (the User Interface or UI node), we were using and overloading our default Grid Resource Broker (RB), which takes care of matching jobs to available resources. As the RB is shared among several sites, our load was affecting many other users. Other failures identified involved successful jobs whose output was lost, unrecoverable or empty.

To solve submission problems, we extended our submission tool to use a time-out to detect stalled submissions, and to maintain a dynamic list of available RBs to load-balance submissions over them and avoid overloads. As for job failures, we added to the monitor script the ability to detect aborted or failed jobs and to resubmit them automatically. This simple device is useful for most problems except immortal jobs, which can only be detected if it is possible to impose an upper bound on execution times that may be used as a time-out or, if not, by submitting jobs more than once to collect the results of the first to finish, and kill the others.

Efficiency measures

Using the timings collected, we could measure for each job the time spent on submission (s_1 - s_0), time required by the Grid to allocate resources and start the job (e_0 - s_1), time taken by the job ($e_1 - e_0$), delay incurred to detect job termination ($r_0 - e_1$), and time needed to retrieve results ($r_1 - r_0$). In addition, by collating the individual statistics, it was easy to measure total times incurred at each step: *e.g.*, for submission, it would be $\max\{s_1\} - \min\{s_0\}$), accumulated CPU time ($\sum(e_1 - e_0)$), total execution wall-clock time ($\max\{r_1\} - \min\{s_0\}$), *etc.*

The mean **execution time** for our jobs varied slightly across experiments, about 8-10K seconds, yielding, in principle, a good balance between the serial and parallel parts. However, time variation ranged between ~500 and 115,000 seconds.

Our initial estimation of the benefit expected from the Grid was based on our perception that **job submission** was a quick process, which we further bound with a time-out. Indeed, our measures reveal that, for our problem (homogeneous jobs of ~800KB in size), submission times are in the range of 12-266 seconds, with a mean of 32 seconds. Thus, the contribution of the submission step is very low in relation to the average running time (0.3-0.4%). Something similar happens with the final **output retrieval** step, which ranges between 5 and 150 seconds.

There are other sources of overhead though: once a job is copied to the Grid, there is a delay owing to **internal Grid housekeeping**. Similarly, once a job is finished, there is a delay until the overall Grid self-monitoring structure gets notified and the status is updated.

From our measures, we conclude that this contribution is significant and poses a strong tax on the efficiency gains that can be achieved: the time taken for a job to start execution ranged between 30 seconds and 60K seconds, with an average of ~4-6K.

In order to put these measures in perspective, we need to know the **number of CPUs actually used:** we noted the host name of the WNs and counted the number of different machines accessed for each simulation experiment. Usual numbers were uniformly around 2,400 different machines for a simulation running 10,000 jobs.

Finally, by comparing the actual execution time of the job with the total wall-clock time taken, we can quantify efficiency gains: on average, jobs took ~9 times longer to run on the Grid, with the best case taking only 1.006 and the worst case 150 times more than local execution.

The massively parallel nature of the Grid, however, may compensate for these efficiency losses by allowing many jobs to run simultaneously. We added the total CPU time used for a 10,000 job experiment and divided it by the total time taken. This total time includes job resubmission and hence accounts for more than 10,000 actual jobs. For our problem, this consistently resulted in a speed-up of ~190-fold relative to a single computer.

To quantify these benefits, let us denote N_n the number of nodes used, N_j the number of jobs to be run, t_j the time per job, t_s the time to submit a job, t_b the time used in Grid house-keeping tasks, t_e the execution time, and t_r the time required for result retrieval.

(1) The average time needed to run a job would be $t_j = t_s + t_b + t_e + t_r$.

(2) The time needed for sequential execution of our jobs on a single node would be $t_l = t_e \times N_j$, whereas the time needed for sequential execution on the Grid (*e.g.*, using only one node) would be $t_g = t_j \times N_j$, which, as $t_j \rangle t_e$, means that Grid execution time is obviously longer for sequential jobs.

(3) The time required for parallel execution on the Grid is more difficult to evaluate, and depends on the number of nodes that can be used in parallel. Ideally, the Grid overhead times (t_s, t_b and t_r) should be close to zero, making the total time for parallel execution $\rightarrow t_l / N_n$. Ideally, one would expect nodes to be reconsidered as soon as they finish a job, hence $N_n \propto (t_e / t_s) + 1$. However, as the Grid is geographically spread, one may expect a significant delay between the time a node finishes execution and the time an RB notices it is free. This has an impact on resource allocation, which now takes longer, making $N_n \propto ((t_b + t_e) / t_s) + 1$. This means that we may expect to use up fewer nodes for short-running jobs than for long-running jobs. We may also derive estimations for the maximum number of nodes that can be reached by using the maximum values of t_b and te and the minimum value of t_s.

We have already seen that both $\overline{t_s}$, and $\overline{t_r}$ are relatively small (~30 seconds each), and thus, as $t_e \rangle\rangle t_s \wedge t_r$, their impact tends to zero (0.3 – 0.4% in our case). The scheduling overhead, however, is non-negligible. This delay becomes significant for small job numbers and for short jobs, hence reducing Grid speed-up[2]. On the other hand, as execution time decreases, the impact of the time required for sequential job submission increases. This can be ameliorated

2 We have been able to verify these results on other kinds of problem with different numbers of jobs and execution times (Carrera, G., Solano, A., Valverde, J. R. and Carazo, J.M., unpublished).

by partially parallelising job submission, but will still hit a sequential limit in data transfer from the submission node to the RB, and usually results in downgraded performance with respect to an ideal parallel execution.

Discussion

We needed to reproduce the behaviour of a population system whose experimental analysis would have been too cumbersome to simulate fully, being a stochastic process (mutations), which requires Monte Carlo-like methods. The dynamic behaviour of the system results in dramatic population size changes, depending on the initial parameters (as a higher impact on survival fitness means slower growth and smaller populations), which in turn results in a wide variation in running times (various orders of magnitude).

The Grid gives any researcher immediate access to huge computing power through a large number of geographically spread machines. For large parallel problems with reduced communication needs such as this, the Grid is an easy and powerful solution.

Optimising computation

Communications in the Grid have a larger latency and are slower than on a cluster; hence, it is desirable to keep them at a minimum in relation to parallel computation, according to Amdahl's law. The best trade-off can be achieved when computation may proceed for long times with a large number of jobs, but most sites impose run-time limits (usually 72h).

If the number of jobs to perform is not too high, users may aim for the smaller number of sites that accept longer jobs on their queues. On the other hand, if users prefer to get results more swiftly by splitting the work among many shorter jobs, the number of available machines increases considerably.

When execution times are fairly homogeneous, users may fine-tune jobs to fit on the allowed time-slot and optimise the communications/computation ratio; in our case, large run-time variability forced us to plan for the worst-case scenario (ensure longest jobs would fit), resulting in relevant efficiency penalties for the shortest jobs.

Job management

For running a single job, the EGEE Grid offers convenient commands for the user. However, when the number of jobs grows to the order of thousands, new problems arise that demand more sophisticated job-handling mechanisms: the incidence of aborted or failed jobs, for various reasons, may reach 10-15% of jobs, requiring the inclusion of additional job-management procedures. The most immediate approach, and the one we have used here, is to detect and re-start failed jobs up to a maximum number of times, but other approaches are possible: *e.g.*, launching various instances of the same job, taking the results of the first to finish and discarding all others, or waiting for various jobs to finish and comparing their output for additional resilience.

As the number of jobs increases into the tens of thousands, new issues need to be considered. First, we reduced overload over the RB by performing some load balancing over all available hosts. As RBs themselves may also fail, a dynamic detection and recovery mechanism for failing RBs was added too. Second, very rare events need to be considered and dealt with, either manually (if their incidence is low enough and circumstances allow) or automatically. The most relevant of these is probably jobs hanging on submission, as this may stop the whole experiment; stalled submission can be conveniently dealt with by implementing a simple time-out mechanism.

A different problem is posed by immortal jobs, which remain eternally in 'running' state. This may be easy to spot if upper-bound estimation of job run-time is possible, so that jobs exceeding it can be considered lost and re-started; but when there is high variability in run-times (as was our case), or there is no easy way to predict an upper bound, detection of these jobs becomes increasingly difficult, as the long run-time might be inherently correct. In such cases, possible solutions are:

- run single instances and after sensible time (we used $\sim$80 hours) detect, kill and re-start unfinished jobs;
- replicate all jobs and take results from the first to finish, killing all other copies.

Efficiency considerations

We have taken timing measures at the various steps *avoiding use of our local cluster and making sure jobs were freely allocated to any WNs by the Grid*, so that measures include real-world effects. Timing checkpoints were taken using UTC to enforce a common time frame.

Regarding Grid efficiency, we can see that the submission process is efficient. The same can be said of result retrieval. Consequently, their impact is almost negligible. This is demonstrated by our finding a minimum efficiency loss of 0.006 for a Grid job not executed on our local cluster. Once the job is submitted, jobs suffer a house-keeping delay until execution. In our experience, job scheduling took a significant amount of time (on average, 4-6K seconds) with large variability. Given our experiment design, we did not take accurate measures of Grid house-keeping after jobs finished: it is possible that there were large delays, which we didn't detect because our data were actually available when we performed the test. Nevertheless, our results suggest that this final step may be fairly quick, taking perhaps a few minutes, but this needs confirmation.

With these data at hand, we can already draw several conclusions, which can be used as advice for Grid usage. First, resource management on the Grid is undoubtedly the area where biggest efficiency gains can still be achieved. If efficiency is a concern, it may be worth considering using alternate scheduling mechanisms, such as those provided by GridWay (Huedo *et al.*, 2004), currently part of the Globus Toolkit (Foster and Kesselman, 1997) and planned for inclusion on gLite[3].

For single jobs, efficiency may reduce to as little as 1.006 or as much as 150 times; however, on average, it will be reduced by about one order of magnitude. Thus, if the single job to be run is a Message Passing Interface (MPI) parallel job to be launched against a big (more than 10-node) cluster, it may compensate for the Grid inefficiency. If the job takes too long and the system cannot be tied for that amount of time (*e.g.*, a shared desktop), or if the local system is already overloaded (*e.g.*, a time-sharing system with too many CPU-bound processes), then the Grid provides a convenient way to run jobs that otherwise would be impossible, difficult or very slow to complete locally.

3 http://glite.cern.ch

For large numbers of jobs, the Grid provides a way to speed up problems and deliver quicker responses, which may prove successful for most researchers. For instance, we were far from the maximum theoretical linear speed-up (10,000 times for 10,000 independent processes), and even from the practical speed-up (2,400 times for the 2,400 different CPUs we could harvest), but we still could accelerate our problem 190 times, which allowed us to run in 1½ days (1 day 14h 01m 42s) a project that otherwise would have taken almost one year (313 days 04h 39m 33s), or in 4 ½ days (4 days 19h 38m 37s) a project requiring 2 ½ years (930 days 02h 25m 20s) of CPU time.

It is worth noting that our low efficiency was partly the result of our unequal run-times, which prevented reaching a better parallel/serial ratio. Higher speed-ups should be possible for better-behaved problems, or with more refined job-management strategies.

Conclusion

We have been able to run large-scale population dynamics simulations on the Grid with relatively little effort: no changes were needed to the simulation software, work was split into suitably-sized chunks for execution, and job management was handled by relatively simple shell scripts. In the process, we had to deal with and solve a number of problems, developing generic tools that are available under the GNU public license[4] from the author.

Each experiment involved large numbers of jobs (usually 10,000), allowing us to collect statistical data to monitor Grid performance and efficiency gains. We have identified Grid house-keeping as a major contributor to reduced efficiency, although we could still achieve significant speed-ups (~190x) using thousands (>2,400) of CPUs, allowing us to solve in days a problem that would otherwise have taken years to complete. Our results are in line with observations on other applications by our group and others (Jacq *et al.*, 2007), and lay the basic foundation for understanding the main issues affecting Grid development for large embarrassingly parallel applications.

4 http://ahriman.cnb.csic.es/sbg/tiki-list_file_gallery.php?galleryId=1

Acknowledgements

The author wishes to acknowledge the invaluable scientific cooperation of A. Couce and J. Blázquez of CNB/CSIC.

Funding: I wish to thank the European Commission for its support to projects EGEE (INFSO-RI-031688) and EMBRACE (LHSG-CT-2004-512092), which made this work possible.

References

1. Adami C, Ofria C, Collier TC (2000) Evolution of biological complexity. *Proc Natl Acad Sci USA* **97**, 4463-4468.
2. Foster I, Kesselman C (1997) Globus: A metacomputing infrastructure toolkit *Int J Supercomput Appl* **11**(2), 115-128.
3. Huedo E, Montero RS, Llorente IM (2004) A framework for adaptive execution in Grids. *Softw Pract Exper* **34**(7), 631-651.
4. Jacq N, Salzemann J, Jacq F, Legré Y, Medernach E *et al.* (2007) Grid-enabled Virtual Screening Against Malaria. *J Grid Computing* **6**(1), 29-43.
5. Lenski RE, Ofria C, Collier TC, Adami C (1999) Genome complexity, robustness and genetic interactions in digital organisms. *Nature* **400**, 661-664.
6. Sniegowski PD, Gerrish PJ, Lenski RE (1997) Evolution of high mutation rates in experimental populations of E. coli. *Nature* **387**, 703-705.
7. Taddei F, Radman M, Maynard-Smith J, Toupance B, Gouyon PH, Godelle B (1997) Role of mutator alleles in adaptive evolution. *Nature* **387**, 700-702.
8. Wilke CO, Wang JL, Ofria C, Lenski RE, Adami C (2001) Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature* **412**, 331-333.