

# Parallel LEA

## A parallel software for Likelihood-based Estimation of Admixture in population genetics\*

AMBRA GIOVANNINI<sup>1</sup>      GAETANO ZANGHIRATI<sup>1</sup>

MARK A. BEAUMONT<sup>2</sup>      LOUNÈS CHIKHI<sup>3,4</sup>      GUIDO BARBUJANI<sup>5</sup>

<sup>1</sup> *Department of Mathematics, University of Ferrara, via Machiavelli, 35, I-44100 Ferrara, Italy.*

<sup>2</sup> *Philip Lyle Research Building, PO Box 68, University of Reading, Whiteknights, RG6 6BX Reading, U.K.*

<sup>3</sup> *Instituto Gulbenkian de Ciência, Rua da Quinta Grande, n. 6, P-2780-156 Oeiras, Portugal.*

<sup>4</sup> *Laboratoire Evolution et Diversité Biologique, UMR CNRS/UPS 5174, Université Paul Sabatier, 118 route de Narbonne, 31062 Toulouse cédex 09 France.*

<sup>5</sup> *Department of Biology, University of Ferrara, via Borsari, 46, I-44100 Ferrara, Italy.*

Version 1.0 – Revision: December 2008

### Overview

This software implements an innovative parallel approach to the admixture estimation process for increasingly large genomic data. It is aimed for strictly-coupled, distributed memory multiprocessor systems running Linux and equipped with MPI (Message Passing Interface, Forum (1994)).

Estimating population admixture has important implications in relevant research areas such as human evolutionary history, biogeography and conservation genetics.

Likelihood-based approaches are very appealing but their effectiveness depends on a sufficiently large amount of Monte Carlo simulations.

Some computational tools for admixture estimation are available for scalar environment but they usually require quite long running time.

For additional details we refer the interested reader to Giovannini *et al.* (2008); Choisy *et al.* (2004); Wang (2003) and the references therein. Further theoretical and model discussions can be found for example in Chikhi *et al.* (2001); Belle *et al.* (2006); Langella *et al.* (2001).

### Algorithm

The main program phases are the following:

1. initialization of the parallel environment;
2. the master processor reads the input files and distributes the input data to the other processors;
3. locally and asynchronously, each processor computes the data likelihood based on the allele frequencies in the populations involved;
4. the master processor collects the evolution results from each processor and computes the new parental population frequencies.

The parallel implementation is developed by distributing the structure storing the loci data in a uniform way among the available processors.

### Source Code

The program is an object-oriented code in C++ language and the sources are available for download at <http://dm.unife.it/parlea>. It is free for non-commercial users. At present the program has been tested on the following system:

- Intel-based IBM Linux cluster (Pentium IV)
- AMD-based IBM Linux cluster (Opteron)

---

\*The development of this software was partially supported by a joint initiative of the Computer Science Course of the University of Ferrara (Italy) and the CINECA Supercomputing Center (Bologna, Italy).

## How to build

To compile the sources you need a C++ compiler for your platform and a UNIX-like `make` command. Make the source directory your current directory, then edit the `Makefile` file and set the environment variable `$CXX` to the name of your C++ compiler command: for instance, type

```
$CXX = mpiCC (or $CC = mpicc to compile standard C sources in place of C++ sources)
```

if you use the Intel compiler. You have to use `pgCC` for the PGI compiler, instead. Then set the optional compiler command line arguments in the `$CXXFLAGS` environment variable. This is the right place where to put platform-specific code optimization options such as `-O3` or `-Ofast`. The variable `$LIB` is mainly intended for expert users and shouldn't need any change from the provided defaults. Type in this variable the list of additional libraries that you want to be linked with the executable, in the standard way `-llibname`. Furthermore, you should ensure that the MPI libraries be linked to the program: this issue is quite dependent upon the compiler/linker settings and in some cases there is no need to explicitly declare such a library. Otherwise, something like `-lmpi` or `-lmpich` should be added in `$LIB`.

All binaries will be created by default in the current directory, but this can be changed by adding the `BINDIR` environment variable to the `Makefile`, by setting it to the preferred path for binaries and finally by pre-pending `{ $BINDIR } \` to the name of the executable in the definition of the variable `EXE`. Please note that under UNIX-like systems you should previously check that you have writing access to the desired directory.

The compilation process should be quite fast. Warning messages could be issued by the compiler if strict adherence to the ANSI C/C++ language is requested (*i.e.*, if something equivalent to the `-pedantic` or `-ansi` options of `gcc` is given to the compiler).

At the system prompt, simply type

```
make
```

or

```
make parlea
```

to get the executable `parlea.x` in the current directory.

## How to use

The parallel code `parlea.x` must be called within an MPI-compliant environment. This is again strongly dependent on the system the program must run on. Usually, the command-line invocation requires as an option at least the number of processors that the system should allocate to the program.

For example, if you are using a Linux cluster equipped with OpenMPI library and supporting LSF (Load Sharing Facility), then to run MPI-based parallel programs you must use the command `mpirun.lsf`. To run `parlea.x` interactively on `Npes` processors:

```
bsub -Is -n Npes -a openmpi -W hh:mm -q queueName mpirun.lsf ./parlea.x ParLEA options
```

In the command line, the option `-Is` specifies the interactive mode, `Npes` is the number of required processors, the option `-a openmpi` selects the job submission system that will be used to run the program (it is strongly site dependent) and `-W` gives the wallclock limit. Optionally, one can also set the option `-q` with the name of the queue where one would like the task to be sent, but this is not a commitment for LSF, which always chooses the most appropriate one. The final part after the executable name `parlea.x` is intended to be replaced by the options for the `ParLEA` program only.

Similarly, on a cluster machine supporting POE (Parallel Operating Environment), the following line will run the program in interactive mode on `Npes` processors that can be allocated on at most `Nnod` nodes, each node having `Ntsk` tasks:

```
poe ./parlea.x -procs Npes -nodes Nnod -task_per_node Ntsk ParLEA options
```

Notice here that since `parlea.x` is not an OpenMP-based executable, each task is actually a single MPI process running on a single processor: thus, the number of tasks per processor is always 1, so that the value `Ntsk` specifies in this case the number of MPI processes to start on each node. It follows that the value given to the option `-task_per_node` is the number of processors that the system is required to allocate on each node.

Other parallel systems can have different command line syntaxes. However, on parallel machines shared by a number of different users the interactive mode is not the preferred choice for running jobs: in these cases, the administrators always set up a batch queuing system, with different levels of running times and resources allocation. Analogously to the parallel environments, different batch queuing systems are available and each one has its own syntax: there, to run a program you need to write a batch file and submit it to the queuing system. Given that both the scripting syntax and the job submission syntax system can be quite different on your specific platform, if you want to run `parlea.x` onto one of these systems we strongly suggest to carefully check the batch system documentation to see how to correctly write the batch file and how to submit it to the appropriate queue. An example is provided in the section "Testing the program".

A number of input files must be provided to the executable: they are described in the next subsections. For easier reading, in what follows we assume that the data of exactly three populations are available: two parental ( $P_1$  and  $P_2$ ) and one hybrid ( $P_h$ ) populations.

### infile

This file contains the absolute alleles frequencies of each locus. The following data are stored at the very beginning of the file:

value	type	meaning
<b>flag</b>	boolean (0 or 1)	0 if the data in each row belong to the same population, 1 if they belong to the same allele.
<b>npop</b>	integer	number of populations. Usually 3: two parental and one hybrid.
<b>nloc</b>	integer	number of loci to be analyzed ( $\geq 1$ ).

Then **nloc** groups of rows follow, one for each locus. Each group starts with the number **nall** of alleles in the locus, then the next rows in the group have a different meaning depending on the value **flag** in the first file row:

value	type	meaning
if <b>flag</b> = 0 then there are three additional rows in the group:		
$f_{1_1}, f_{1_2}, \dots, f_{1_{nall}}$	integers	absolute allelic frequencies in the first population.
$f_{2_1}, f_{2_2}, \dots, f_{2_{nall}}$	integers	absolute allelic frequencies in the second population.
$f_{h_1}, f_{h_2}, \dots, f_{h_{nall}}$	integers	absolute allelic frequencies in the hybrid population.
if <b>flag</b> = 1 then there are <b>nall</b> additional rows in the group:		
$f_{1_1}, f_{2_1}, f_{h_1}$	integers	absolute frequencies of the first allele in the three populations.
$f_{1_2}, f_{2_2}, f_{h_2}$	integers	absolute frequencies of the second allele in the three populations.
$\vdots$		
$f_{1_{nall}}, f_{2_{nall}}, f_{h_{nall}}$	integers	absolute frequencies of the last allele in the three populations.

An example of how the **infile** file looks like in the two cases is the following: there are three populations and two loci, the first one with five alleles and the second one with three alleles. In the first case **flag** = 0:

	<i>file content</i>	
	0	<b>flag</b>
	3	<i>number of populations</i>
	2	<i>number of loci = number of groups</i>
<i>first group</i> {	5	<i>number of alleles in the first locus (nall)</i>
	0 2 3 56 1	<i>absolute frequencies in the first population</i>
	2 0 61 0 0	<i>absolute frequencies in the second population</i>
	0 3 4 25 10	<i>absolute frequencies in the hybrid population</i>
<i>second group</i> {	3	<i>number of alleles in the second locus (nall)</i>
	0 2 5	
	9 1 3	
	0 10 0	

so that both the two groups have exactly three rows following the number of alleles (one row for each population), whose numbers are the absolute frequencies of the alleles in the first, second and hybrid populations, respectively.

In the second case **flag** = 1:

	<i>file content</i>	
	1	<b>flag</b>
	3	<i>number of populations</i>
	2	<i>number of loci = number of groups</i>
<i>first group</i> {	5	<i>number of alleles in the first locus (nall)</i>
	0 2 0	<i>absolute freq. of the first allele in the 1<sup>st</sup>, 2<sup>nd</sup> and hybrid popul., respect.</i>
	2 0 3	<i>absolute freq. of the second allele in the 1<sup>st</sup>, 2<sup>nd</sup> and hybrid popul., respect.</i>
	3 61 4	<i>absolute freq. of the third allele in the 1<sup>st</sup>, 2<sup>nd</sup> and hybrid popul., respect.</i>
	56 0 25	<i>absolute freq. of the fourth allele in the 1<sup>st</sup>, 2<sup>nd</sup> and hybrid popul., respect.</i>
<i>second group</i> {	1 0 10	<i>absolute freq. of the fifth allele in the 1<sup>st</sup>, 2<sup>nd</sup> and hybrid popul., respect.</i>
	3	<i>number of alleles in the second locus (nall)</i>
	0 9 0	
	2 1 10	
	5 3 0	

so each group has exactly the same number of rows next to **nall** as the number of alleles in the locus and each one of these rows has exactly three numbers, indicating the absolute frequencies of that allele in the first, second and hybrid populations, respectively. Mathematically speaking, the allelic frequencies matrix of each locus in the second case (**flag** = 1) is the transpose of the corresponding matrix in the first case (**flag** = 0).

## KEEP\_STATE

This file contains information on the Monte Carlo simulation status updated every 500 iterations. It allows to trace allelic frequencies during the run and is useful to start the analysis in the middle of the process in place of its beginning, using the collected results as a preliminary estimate of the likelihood.

## INTFILE

This is the file where the (integer) seeds to initialize the provided random number generator are stored. The generator is a *shift-register random number generator*. If this file is not available, then it is automatically created by the program: it does not need to be edited in any way and is automatically updated during the run. The only requirement is that it must be stored in the same directory as the data to be analyzed.

## mh\_tn.dat

This text file is updated every 5 steps: it stores the Markov chain status for the model parameters. An example of how it looks like is the following:

```
5395 -483.360645 0.302217 0.001282 0.008396 0.001241
5400 -481.171866 0.307863 0.001130 0.008388 0.001263
5405 -482.422665 0.348920 0.001296 0.008549 0.001183
```

The columns of each row contain the following data (left-to-right): number of steps, likelihood, allelic frequency of the first population ( $p_1$ ) and the three “scaled times” (*i.e.*, generations divided by the effective size) of the first population ( $t_1$ ), of the second population ( $t_2$ ) and of the hybrid population ( $t_h$ ), respectively.

This information can be used to compute the likelihood distribution of the model parameters through a spreadsheet.

## par1\_freqs.dat and par2\_freqs.dat

These two files are updated every 10 iterations. They contain information about the allelic frequencies of the parental populations during the Monte Carlo simulation. If there are more than two parental population, one file is created for each one. However, currently the method only deals with two parental populations.

## STATE

The current status of the Markov chain is stored here and the file must be put in the same directory of the files `infile` and `INTFILE`.

It is a very important file: it allows the user to stop the program at any moment and to restart the process later. This file is updated every 5 iterations, so it is possible that the program does not start exactly from the same point and that some of the last iterations (up to 4) be lost.

When the program runs for the first time (no restart) the content of this file should look like:

```
0 0 50000 0.0 0
```

In this case, the third value is the number of Monte Carlo iterations required and it's the only relevant value.

Notice that there's no rule to establish how long a Markov chain should be to get to the equilibrium. From a practical experience, there's a common agreement that 50000 steps are enough to get a single locus equilibrium: that's why this number is chosen as the default. Nevertheless, the user must remember that the number of Monte Carlo iterations required depends on the number of loci, on the number of alleles per locus and on other parameters. When the program ends, the file looks as follows:

```

      28134 50000 50000 -22.626588 0.848869
locus 0 { 2.131631e-01 7.868369e-01
          { 4.864052e-01 5.135948e-01
          :
locus N { allele0 ... alleleM
          { allele0 ... alleleM
          4.795702 0.165841 1.400176
```

The values in the first row are: the number of *successful Monte Carlo jumps* (that is how many times the new state generated by the Markov chain process is accepted as the new model state), the actual number of completed iterations, the number of iterations required, the log-likelihood and the contribution of the first parental population ( $p_1$ ). Then, for each locus there is a group of rows containing the parental allelic frequencies at that locus, one row for each parental population. In the example the locus 0 has two alleles, whilst locus  $N$  has  $M + 1$  alleles. The very last row reports the times of admixture scaled by the actual population size, that is  $t_1$ ,  $t_2$  and  $t_h$ , in the case of two parental populations.

## Testing the program

Here we provide an execution example to allow the user to check if the executable produced by the compilation gives reasonable results.

Instances of each one of the input files described in the previous section are provided in the subdirectory `test`. These files contain the data of two parental populations and one hybrid population: the test case is a very small one and is for checking purpose only.

To check the results we ran the scalar version of the program, `lea`, and we collected the corresponding results in the output file `restest.dat` in the `test` subdirectory.

### Parallel tests

We assume here to work on a Opteron-based Linux cluster, that the executable `parlea.x` is available in the current directory and that the OpenMPI library is installed on the system. If your machine supports LSF (Load Sharing Facility), then normally the only possibility to run MPI-based parallel programs is through the command `mpirun.lsf`, either in a batch interactive or in a batch background session.

In order to run `parlea.x` interactively on 4 processors for two hours in the queue named `parallel`, then you could use the following command:

```
bsub -Is -n 4 -a openmpi -W 02:00 -q parallel mpirun.lsf ./parlea.x > parlea_batch.out
```

The last part of the command line (from “>” up to the end of the command) will redirect the standard output to the file `parlea_batch.out` in the current directory for an easier inspection. If you prefer to see the output on your console simply omit that last part.

The same can be done in a non-interactive batch session: in this case, you should first save in a file, say `parlea_job.script`, a job script similar to the following (note that directory names and modules version are all site dependent):

```
1  #!/bin/bash
2
3  ## mpirun.lsf needs this option for Infiniband network
4  #BSUB -a openmpi
5
6  ## Number of processors (tasks)
7  #BSUB -n 4
8
9  ## Wall clock limit in the form hh:mm (hours:minutes)
10 #BSUB -W 02:00
11
12 ## Output file
13 #BSUB -oo lea_prog.%J.out
14
15 ## Error file
16 #BSUB -eo lea_prog.%J.err
17
18 ## Module loading
19 . /options/modules/init/bash
20
21 module purge
22 module load openmpi/1.2.5/intel/10.1
23
24 ROOTDIR=${PWD}
25 EXE=parlea.x
26
27 mpirun.lsf ${ROOTDIR}/${EXE} < lea_batch.in > lea_batch.out
```

and then submit it to the queuing system (often known as “load leveler program”) with the command

```
bsub < parlea_job.script
```

For those not familiar with the script syntax, a detailed description of the previous script file content follows. However, this is specific for the OpenPBS system. First of all, lines beginning with a double sharp sign (“##”) are comment lines, so they are ignored by both the shell and the load leveler program. Those lines beginning with “#BSUB” (with one sharp sign only) are ignored by the shell, but interpreted by the load leveler. The lines not beginning with a sharp sign are interpreted as shell commands, as usual, and ignored by the load leveler program. The first line of the script calls the shell interpreter, that in this case is the `bash` shell: it requires that all the shell commands appearing in the rest of the script are given with the appropriate syntax. The next lines (3–16) are grouped in couples: in each couple,

the first line is a comment briefly explaining what data have to be provided to the load leveler program through the next option. These options are the same (with the same meaning) as the command-line options to the `bsub` command that you can use in interactive mode, as shown before. At line 4, the option `-a` selects the connection network linking the cluster nodes: in our case the *Infiniband network* is selected. The option `-n` at line 7 provides the number of processors the user requires to be allocated for the execution of his/her program. The option `-W` at line 10 specifies the wallclock time limit required for completing the execution of the user's program, including I/O. Particular care must be given to this option: a small time limit can increase the chances that the job will run soon, but can be a too short time to complete it, whereas a too large time limit will ensure the program completion, but can easily enlarge quite a lot the waiting time before the system actually loads and executes the user's code. Moreover, site-specific constraints on time limits are usually set by the system managers. So, a general rule is to try to get an accurate estimate of the expected running time and provide a time limit 10-20% larger (and always check the load leveler documentation). The options `-oo` and `-eo` at lines 13 and 16, respectively, provide filenames that the load leveler will use to redirect its output and error messages, respectively. The `%J` in the filenames is a placeholder for the job number, that will be automatically assigned by the load leveler once the job will be submitted: this is a way to store the load leveler messages while preserving their association with the job generating them. If the console standard output and standard error messages are not explicitly redirected in the command invocation line (through the operator `>`), then they will be added to those two files, otherwise those files will only contains information from the load leveler (notice that in some cases they could not be generated at all, because of script syntax errors, execution errors or queuing system aborting conditions). Clearly, what is described above is just a subset of all available load leveler options and not all of them are strictly required for the script submission. Additional information about OpenPBS options, including easy-to-use quick reference guides, can be found in the official documentation and on internet resources.

The shell commands start at line 19. Lines 19–22 are site-specific commands to load software modules related to the particular MPI architecture the program will run on and the appropriate set of libraries. Lines 24 and 25 set two environment variables in the current shell: `$ROOTDIR` is intended to be the directory of the user's executable program, while `$EXE` stands for the executable filename. This is *not required* in any way, but it can help in managing different copies of the script file to run different executable versions, for instance (however, in the shown script they have limited usefulness). In this case `$ROOTDIR` will store the directory name from where the script file will be submitted to the load leveler (retrieved from the standard environment variable `$PWD`), while `parlea.x` is the executable program to be run. Finally, line 27 contains the actual program invocation through the OpenMPI standard command `mpirun.lsf`. Once the script is loaded for execution, the values stored in the two environment variables `$ROOTDIR` and `$EXE` are expanded, giving the full path to the executable program to be run. The second part of the command line uses both input and output redirection. First, the operator `<` redirects the standard input, telling the user's executable to load its options from the input file called `lea_batch.in`, that must be positioned in the same directory as both the executable and the script files. Second, the operator `>` redirects the standard output to the file `lea_batch.out`, which will be saved in the same directory as before. The program interacts with the input file as it would be the user entering the data from the keyboard and sends the messages to the output file as it would be the console screen (not a graphics one, of course). This is the usual way to make completely autonomous those executables requiring direct user interaction. When `parlea.x` starts it asks to enter few information before actually running the simulation, which are usually provided from the keyboard: hence, in batch mode they must put in the input redirection file (`lea_batch.in` in this case) exactly as they would be entered from the keyboard, one answer per line. For instance, to start from scratch 10000 iterations one should put the following in the file `lea_batch.in`:

```
1
y
10000
```

that are the data that would be typed in, in interactive mode. Obviously, the standard input file must be a text file (do not use any word processor to write it!) and every new line entered in the file is interpreted as such by the executable, that is it will act as the data line terminator exactly as when the `ENTER` key is pressed on the keyboard at the end of each single answer to the program. Hence, the first answer must be put exactly in the first line and will be read by the program as the first answer, the second line will be the second answer and so on and so forth, for as many lines as the number of input data the program would expects in interactive mode. Pay attention: blank lines in the standard input file are not ignored, but interpreted as empty answers. When the user's program ends, or when the execution is stopped for some reasons, the system automatically closes all files.

The previous description is clearly neither exhaustive nor general in any respect, but it is provided only as an example of how a batch system managing job queuing in a parallel architecture can be used to run the user's program. Please, refer to the specific documentation of your OpenPBS system for more detailed information, or ask the system manager for additional documentation. Other batch queuing system can have a completely different syntax, of course.

Table 1 summarizes the results obtained with 10000 iterations on the toy problem described by the data files available in the `test` subdirectory. It can be seen that the application scales quite well, as it is shown by the wallclock speedup. The data in the third column are the cumulated CPU time of all processors: that's why these numbers can increase even when the total wallclock time decreases. The data in the next column are the mean CPU time per processor, that is the ratio of the cumulative CPU time data in the same row and the corresponding number of processors: they are intended to give only a rough idea of how the workload is distributed to the available processors.

procs	wallclock time (sec.)	cumulative CPU time (sec.)	av. CPU time per proc. (sec.)	wallclock speedup	av. CPU speedup
1	1313	1306	1306		
2	573	1133	567	2.3	2.3
4	389	1502	376	3.4	3.4
8	216	1635	204	6.1	6.4

Table 1: performance results of some parallel runs on an Opteron-based IBM Linux cluster for 10000 iterations.

procs	likelihood	$p_1$	scaled times		
			$t_1$	$t_2$	$t_3$
1	-448.457041	0.666701	0.050013	0.056662	0.072376
2	-453.689567	0.443556	0.077580	0.063987	0.048624
4	-434.030562	0.641873	0.063295	0.046166	0.047533
8	-447.660627	0.385333	0.062971	0.057100	0.060644

Table 2: simulation state in file `mh.tn.dat` after 5 iterations.

These are not actual timings because they are just computed and not obtained by actually monitoring idle time, communications, critical sections and parallel computations. Nevertheless, it can be seen from the last column how the hypothetical speedup is always very close to the real one (wallclock speedup): this can be interpreted as the result of a good parallelization, even if it does not guarantees for that.

The last wallclock speedup is worse than the previous: this is because the test data set is too small for 8 processors. One can easily expect that larger data sets lead to very good performance also with a larger number of processors. Also, the superoptimal behaviour we can observe in the case of two processors is because the task distribution allow for an improved memory exploitation on each processor.

Moreover, notice that you could observe a variability in the execution time on repeated runs of the same command line, due to both the user policy and the machine workload.

As a final reference, we report in Table 2 the data written in file `mh.tn.dat` at the 5th iteration and in Table 3 the numbers at the end of the simulation (file `STATE`): the different figures are caused by the way the random numbers are selected in this version of the code, even if we always started the sequence from the same seed. In particular, the likelihood and the contribution of population  $p_1$  with 2 and 4 processors are quite different from those of the scalar run. On the other hand, the results with 8 processors are much closer to those of the scalar run. Moreover, the last line of the table shows how by increasing the simulation time both the likelihood and  $p_1$  tend to get closer to those of the scalar run. This behaviour is hardly traceable and controllable with the current version of the code (for debug purpose), but we clearly expect that the posterior distribution is independent on the number of processors used. This is one of the main code improvement we plan for the next release: it will use a completely different random number generation library, specifically designed for multiprocessor systems.

## References

- Belle, E. M., Landry, P.-A., and Barbujani, G. (2006). Origins and evolution of the Europeans' genome: evidence from multiple microsatellite loci. *Proceedings of the Royal Society B: Biological Sciences*, **273**, 1595–1602.
- Chikhi, L., Bruford, M. W., and Beaumont, M. A. (2001). Estimation of admixture proportions: A likelihood-based approach using Markov Chain Monte Carlo. *Genetics*, **158**(3), 1347–1362.
- Choisy, M., Frank, P., and Cornuet, J.-M. (2004). Estimating admixture proportions with microsatellites: comparison of methods based on simulated data. *Molecular Ecology*, **13**, 955–968.
- Forum, M. P. I. (1994). MPI: A Message-Passing Interface standard. Technical Report UT-CS-94-230.

procs	successful	completed	required	likelihood	$p_1$
	MC jumps	iterations	iterations		
1	3140	10000	10000	-415.608938	0.346530
2	2772	10000	10000	-378.859664	0.621306
4	3058	10000	10000	-391.607923	0.516936
8	3054	10000	10000	-386.369848	0.356807
8	8511	30000	30000	-403.396287	0.357435

Table 3: final simulation state in file `STATE` after 10000 iterations.

- Giovannini, A., Zanghirati, G., Chikhi, L., and Beaumont, M. A. (2008). A novel parallel approach to the Likelihood-based Estimation of Admixture in population genetics. (*submitted*).
- Langella, O., Chikhi, L., and Beaumont, M. A. (2001). LEA (Likelihood-based Estimation of Admixture): a program to estimate simultaneously admixture and time since the admixture event. *Molecular Ecology Notes*, **1**(2), 357–358.
- Wang, J. (2003). Maximum-likelihood estimation of admixture proportions from genetic data. *Genetics*, **164**, 747–765.