Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

SHAPE Project *Cybeletech - CINES Partnership:*

Parallelization and optimisation for plant selection with Cybeletech

B. Cirou[a*], D. Fernandez[b], G. Hautreux[a], D. Wouters[b]

[a]*Centre Informatique National de l'Enseignament Supérieur (CINES), 950 rue St Priest, 34097 Montpellier, France*
[b]*Cybeletech, 2 rue de la Piquetterie, 91680 Bruyères le Châtel, France*

Abstract

Breeding a new variety is a long process that requires a decade and thousands of experimental trials in fields so as to select the most robust and efficient traits. Some steps in the process of plant selection could be conducted in-silico to reduce the duration and development cost of a new variety.
The plant growth model used in numerical simulations must be calibrated with plant phenotypes data. To define the optimal experimental protocol to be followed for calibrating the model, the model is run with a genetic algorithm.
Optimisation of the plant growth model enabled to reduce its computing time by a factor five. Performances of the whole application were highly improved by implementing a master-slave approach to the optimisation of the evolutionary algorithm. These performances are function of the number of protocols and realisations considered.

1. Introduction

Agriculture is facing big challenges. On the one hand, the agricultural production has to increase by 50% by 2050 in order to cope with the growing global population but on the other hand, the agricultural sector is under strong pressure of environmental cares, as it is responsible for 17 to 32% of green house gases emissions and consumption of 70% of water resources. One way to tackle these challenges is to breed new varieties that produce better yields with less inputs, water and nutrients. Major seed companies are constantly experimenting new varieties, following a process of selection that takes an order of ten years to breed a new variety. The principal aim of numerical technologies in this regard is to assist and optimize selection processes so as to reduce their costs and timescales. This project develops and implements numerical methods to identify the set of measures that should be taken on a tried variety in order to characterize it with good accuracy and with limited costs. To achieve this goal, a plant growth model is used that depends on a certain number of parameters that represent the genotype of a particular variety. The accuracy on the value of the plant model parameters deduced from a typical set of measures corresponding to a protocol is used to represent the characterization of the variety. A genetic optimisation algorithm [1] is then used to find the best protocol under the constraint of measurements costs.

*Corresponding author. *E-mail address*: cirou at. cines.fr

The plant growth model

A plant growth model is used in the project to simulate the construction of a plant phenotype in a given environment assuming a given genotype, represented by a set of parameters. The plant growth model describes generic state variables describing the plant such as the leaf area index or the biomass, but also its environment as for the soil (water or nitrogen content) or the atmosphere (culture temperature, evaporation...). These state variables are evolved on a daily basis taking into account the interaction with the environment and describing physical and empirical processes. For this reason, the model is mainly sequential but the computation is fast, of the order of 50 ms on one core. The seriality of the model is not an obstacle to parallelization of the whole application as the MPI parallelism is applied on the genetic algorithm for protocol selection, not on the plant growth model itself.

The fitness function

In order to find the protocol that yields the best accuracy on model parameters under a constraint of cost, a fitness function is built that associates to the a protocol P a value taking into account the accuracy obtained on model parameters A(P) and the costs of one realisation of the protocol C(P), that is to say the cost to take all the measurements in the protocol: $F(P) = A(P) + \lambda C(P)$ where λ is a constant that weights the relative influence of the accuracy (small λ) against the costs (high λ). In an experimental protocol, a limited number of observables are available and considered. These are here the leaf area index (LAI), the dry mass and the different phenological stages. For one trial of dry matter and leaf area index, three measurements are taken, each at one key moment of the development of the plant, in order to probe the evolution of these variables. The accuracy function on parameters $A(P) = \Sigma\ \xi i\ ai(P)$ is computed from the accuracy obtained on each labeled i. In order to estimate this accuracy, 100 realisations of the protocol P are produced, each one assuming a different set of parameters taken randomly. A calibrated set of parameters is then deduced for each one of the realisations. The accuracy ai(P) quantifies the discrepancy between the parameters set assumed to produce the realisation and the parameters set that is calibrated from the actual measurements. It is defined as the root mean square of the difference between assumed and calibrated parameter over the 100 realisations. A relative weight ξi is added. This weight stands for the sensitivity indexes obtained from a sensitivity analysis on the simulation of yield in the plant growth model in one given environment.

The cost function is computed by associating a fixed cost to each sample measurement and summing over all the measurements to take in the protocol. An arbitrary unit is used for the costs and λ is set so that for ten realisations of the protocol, the accuracy obtained is of the order of 10%.

Optimisation of protocols.

Based on the fitness function F(P), a genetic algorithm is implemented in order to identify the protocol P that minimizes F(P). For this purpose, the parameters of the fitness function, that is to say the number of measurements to take on each observable, are coded in gray binary representation ([2],[3]). This representation makes it convenient for the genetic algorithm to build crossovers between populations and to make random mutations. When generating a new population of protocols, cross-overs are made for each parameter between the binary gray codes of two parents selected randomly according to their fitnesses. In the cross-over, the point in the code corresponding to the transition from parent 1 to parent 2 is picked-up randomly so as to simulate biological meiosis. Random mutations (0->1 or 1->0) are made at a rate of 1%. This rate has been chosen as in [4,5,6] in order to achieve a good trade-off between exploration of the parameters space and exploitation around found solutions. From this new population, inheriting from the most fitted individuals of the previous population, the fitness is computed for each element.

Profiling and sequential optimisation

The CybeleTech code is fully written in C++. It has its own libraries and relies on the ROOT and AVRO packages. A costly instrumentation of all C++ methods was used. The graph of method invocations was extracted from the profiling output. Part of the call graph is shown in Figure 1. More than four millions of methods invocations appeared to concern memory allocation and for C++ objects and their erase. The INTEL sampling measurement tool Vtune was used to investigate the calling relationship between methods and the time spent in each of them. Vtune slowed down the execution time by a few percents only. The CPU consumption is shown in Table 1.

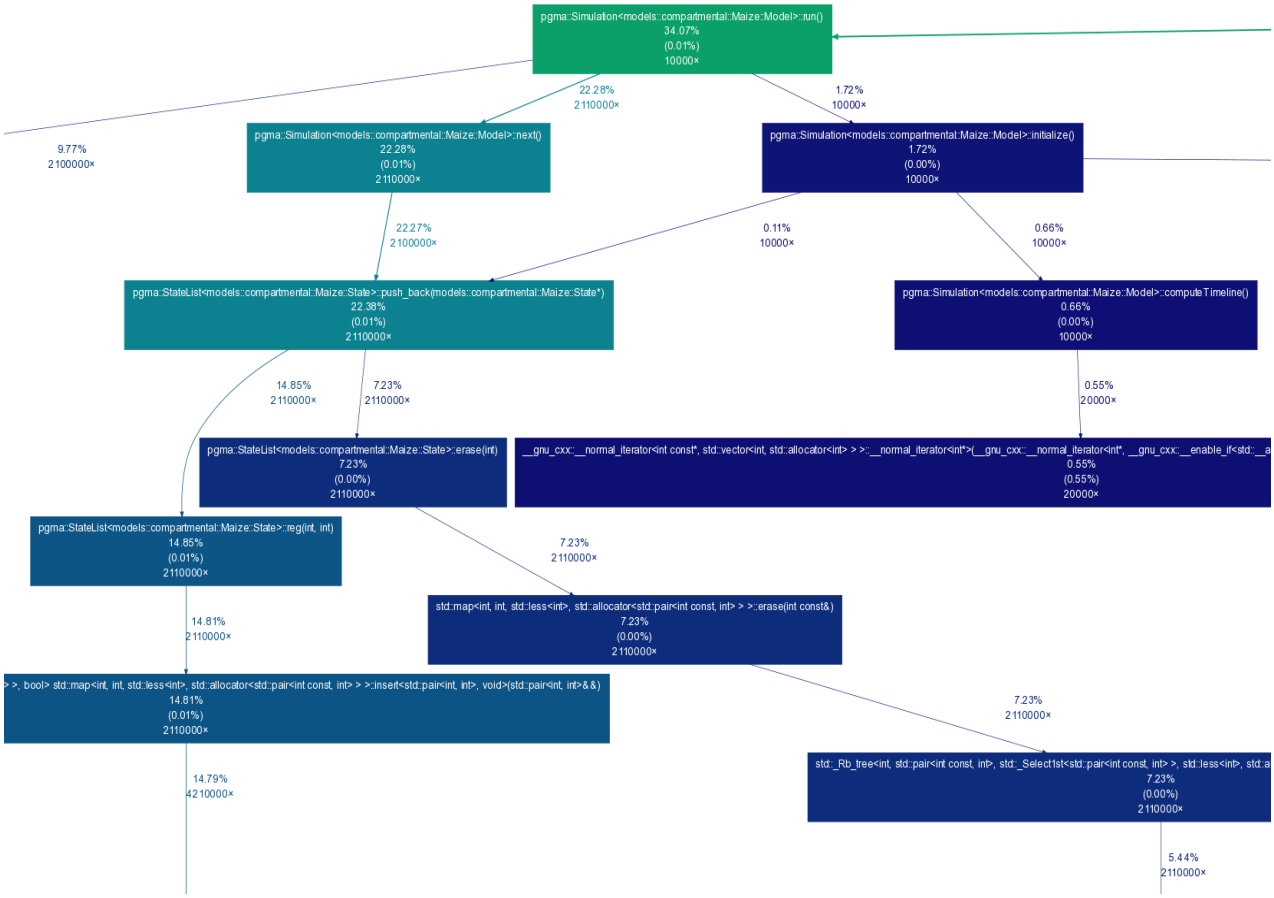

Figure 1: Part of call graph: memory allocation and erase.

The results obtained with Vtune confirmed those obtained from the call graph: a significant amount of computing time was lost in memory management by allocating memory for C++ objects and erasing them. Moreover about nine percent of the time was consumed in the mathematical functions `exp()` and `pow()`.

Optimization of memory management

A flag optimising memory allocation parameters (`-qopt-malloc-options=3`) was used when compiling with INTEL C++ compiler (`icpc`). The OpenMPI internal memory allocator was disabled through the usage of the environment variable `export OMPI_MCA_memory_linux_disable=1`. Then the benchmark was run anew and had an execution time of two minutes versus six.

Shell environment variables usage inside the C++ code were inserted to be able to change for each future simulation job the directory of climatic data and the pedologic file. Two lines of code below were needed:

```
climaticPDFdir = getenv("MC_PDFDIR");
pedologicFile = getenv("MC_PEDOLOGIC_FILE");
pedologicFile = getenv("MC_PEDOLOGIC_FILE");
```

Simulations are run on a daily basis and over a period of hundreds of days. Each day, a new state is created. To reduce the time spent in memory allocation and deallocation of this daily state vector a model state object buffer was created. As in most simulations only few states from the past are needed, the first states of the simulation could be easily overwritten. To decrease the number of calls to the C++ STL the buffer of state vectors "`std::vector<boost::shared_ptr<S> >_states`" was replaced by a static number of states pointers "`S*_states[]`", initialized at the beginning of the simulation. The array is filled in a circular way and a counter was implemented to identify the position of the current state. All the methods of the code were modified in consequences to suppress vector member functions like "`_states.resize()`". The definition of a static number of states have enabled to reduce the computing time by 20%.

Table 1: CPU time consumption for different functions of the model.

Function	CPU Time: Self
	Effective Time
opal_memory_ptmalloc2_int_malloc	12.4%
pgma::StateList<models::compartmental::Maize::State>::operator[]	9.2%
std::vector<double, std::allocator<double>>::vector	8.1%
opal_memory_ptmalloc2_malloc	5.8%
models::compartmental::library_water::modules::rootGrowthTrueDensity<models::compartmen	5.3%
__ieee754_exp	5.1%
models::compartmental::library_energy::modules::tSol<models::compartmental::Maize::State, r	4.5%
mutex_lock	4.2%
opal_memory_ptmalloc2_int_free	4.2%
models::compartmental::library_water::modules::distributionOfTranspiration<models::compartr	3.3%
models::compartmental::library_water::modules::distributionOfEvaporation<models::compartm	2.9%
__ieee754_pow	2.5%
opal_memory_ptmalloc2_free	1.9%
models::compartmental::library_plant::helpers::somSenLaiSen<models::compartmental::Maize:	1.7%
__exp1	1.7%
models::compartmental::library_water::modules::distributionOfIncomingWater<models::compar	1.6%
std::vector<double, std::allocator<double>>::operator[]	1.5%
std::vector<double, std::allocator<double>>::_M_emplace_back_aux<double>	1.3%
models::compartmental::library_water::helpers::WaterAvailableAndEffectiveRoots<models::com	1.1%
models::compartmental::library_plant::modules::laiGrowthForDeterminatePlants<models::comp	1.0%
malloc_consolidate	1.0%
models::compartmental::library_water::helpers::rootsHydricStress<models::compartmental::Ma	0.9%
__finite	0.9%
std::vector<double, std::allocator<double>>::operator=	0.8%
std::_Rb_tree_const_iterator<std::pair<int const, int>>::operator==	0.8%
models::compartmental::library_plant::helpers::dateFromBase	0.7%
malloc	0.6%
std::vector<double, std::allocator<double>>::emplace_back<double>	0.6%
operator new	0.5%
free	0.5%
models::compartmental::library_water::modules::waterBalanceAnnualCrop<models::compartme	0.5%
exp	0.4%
std::__uninitialized_copy<(bool)0>::__uninit_copy<std::move_iterator<double*>, double*>	0.4%
std::__uninitialized_copy<(bool)0>::__uninit_copy<std::move_iterator<double*>, double*>	0.4%
[Outside any known module]	0.4%
std::vector<boost::shared_ptr<models::compartmental::Maize::State>, std::allocator<boost::sh	0.3%

Similarly, the source code was modified to avoid numerous calls to the C++ STL for allocation and deletion of vectors. For instance the dynamic "`std::vector<std::vector<double> > lroot`" which was daily incremented, has been replaced by a static array of double. STL methods like `back(), push_back()` or `size()`, has been replaced by their equivalent for arrays. These changes enabled to reduce the computing time by 30%.

The previous state xl.last() is called many times to compute the new state of the simulation. This invariant C++ object could be called several times within one method which appeared to be highly time consuming (20%). A temporary object `MY_XL_LAST` was created at the beginning of each method to avoid several `xl.last()` calls within one method.

Lower precision but less time consuming approximations of functions exp() and pow() were implemented to reduce the computing time even further. These approximations were called in specific sections of the code only, in order not to loose too much precision in some computations.

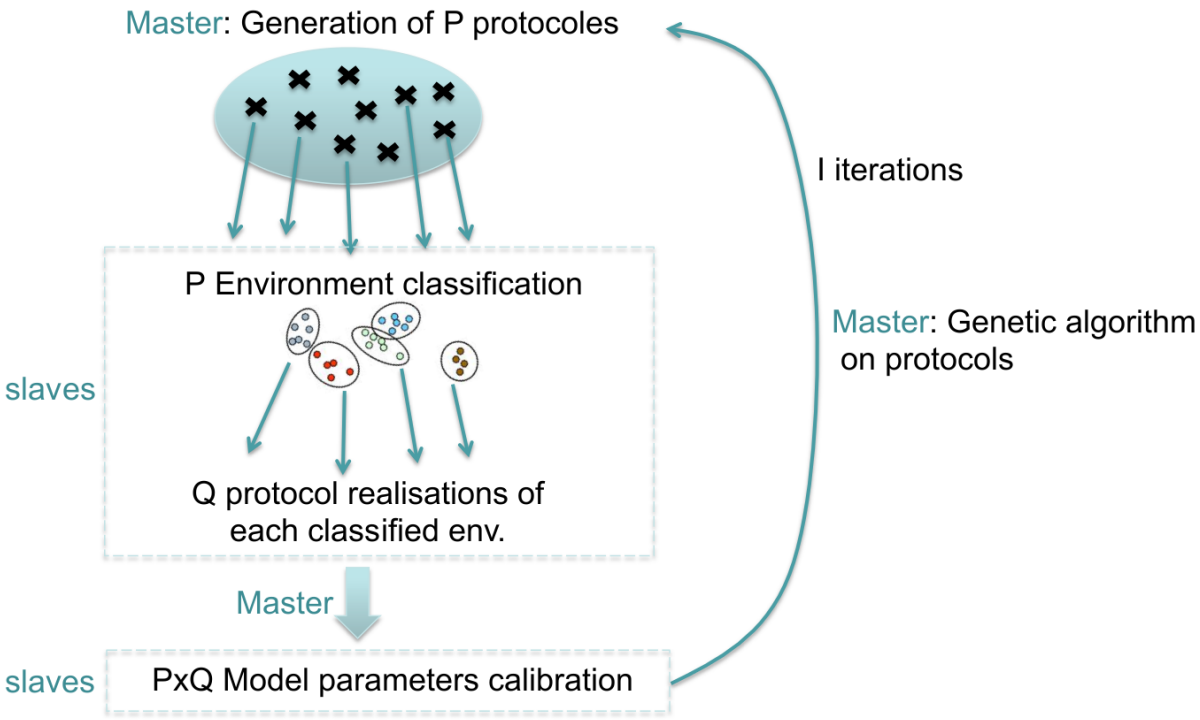

Figure 2: Schematic drawing of the master/slave parallelization of the implementation.

MPI Parallelism

The parallelization of the implementation is done with MPI following a master/slave rule according to the schematic drawing on Figure 2. A process is dedicated to the dynamical repartition of tasks between the other processes. In a first step, the master distributes the computation of the fitness and then runs the genetic algorithm to produce a new population. This last part is costless in regard to the time taken by the computation of the fitness of one protocol. With this level of parallelism, the number of parallel processes that can be deployed is limited by the number of elements in the population which is typically between 100 and 1000. Using a number of elements of 100, Figure 3 shows the scalability plot that can be achieved with this algorithm on the Bull machine Curie (2 Pflops of INTEL SandyBridge [4]). At low number of processes, the speed factor scales better than the ideal case of the diagonal because of the relative weight of the master process that decreases with higher number of processes. Going over 32 processes, the speed factor slows down as compared to the ideal case because of the heterogeneous time of computation of fitness functions causing processes to hang waiting for the last process to finalize.

A second level of parallelism is implement by calibrating the parameter sets for each realisation of each protocol in parallel, the distribution of calibration tasks still being handled dynamically by a master process. This implementation is represented on Figure 2. The maximum number of processes that can be deployed is limited by the number of elements in the population multiplied by the number of realisations of each protocol. Using 20 elements of population and 50 realisations for each element, corresponding to 1000 calibrations, Figure 3 shows the scalability that can be achieved by this parallelization. Above 200 processes, the speed factor slows down because of the relative importance of the sequential part in the second level of parallelism that consists in generating the realisations of the protocols before doing the actual calibration.

In production, this genetic algorithm runs with 100 elements of each population each one having 100 realisations of the protocols and can be deployed on up to 2000 cores.

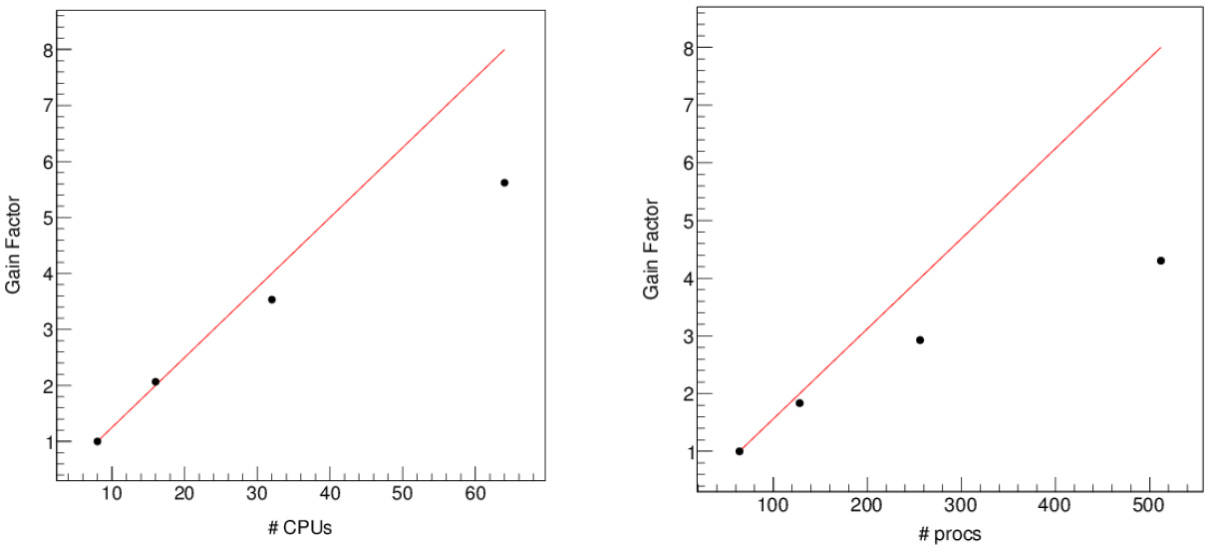

Figure 3: Scallability curves for a number of protocols P and Q realisation : PxQ=1x100 (left) and PxQ=50x20 (right).

Conclusion

An installation of the code and all its third party libraries were performed. An input dataset to be used as a benchmark for simulations was defined. Initial performances and correctness were first validated. The random number generator usage was modified in the sources to get repeatable results and timings. INTEL vtune was used to identify which lines of codes lead to excess time consumption. Those lines concerned C++ object memory management and mathematical functions. After the plant growth model optimisation, the parallelism was improved by adding a master-slave approach to distribute the work among hundreds of MPI ranks.

A face to face meeting with CybeleTech engineers during two days speeded up the understanding of the code and the implementation of optimisations. However not met security requirements to access Curie delayed the beginning of the project. Indeed the SME had internet through an ADSL box. Porting from one linux environment to an other linux environment was not straightforward for C++ codes. This should cover all aspects of the project from the viewpoint of both the SMEs and the PRACE coordinators/coaches e.g. application process, machine access, timescales ...

Thanks to this SHAPE initiative and work the SME Cybeletech was able to compute all the simulations planned and consumed the whole 400k hours allocated.

References

[1] Fraser, Alex (1957). "Simulation of genetic systems by automatic digital computers. I. Introduction". Aust. J. Biol. Sci. 10: 484–491.
[2] Kenneth Levenberg (1944). "A Method for the Solution of Certain Non-Linear Problems in Least Squares". Quarterly of Applied Mathematics 2: 164–168.
[3] Donald Marquardt (1963). "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". SIAM Journal on Applied Mathematics 11 (2): 431–441.
[4] J.D. Schaffer, R.A. Caruna, L.J. Eshelman, R. Das (1989), “A study of control parameters affecting on line performance of genetic algorithms for funtion optimization”, In *Proceedings of the Third International Conference on Genetic Algorithms,* Morgan Kaufmann.
[5] J. Hesser and R. Männer (1990), “Towards an optimal mutation probability”, In *Proceedings of the International Workshop Parallel Problem Solving from Nature*, Sprinter-verlag.
[6] R.N. Greenwell, J.E. Angus, M. Flinck (1995), “Optimal mutation probability for genetic algorithms”, Mathematical and Computer Modelling 21 (8): 1-11.
[7] http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.