Convergence of distributed symbolic regression.

Ben Cardoen
ben.cardoen@student.uantwerpen.be

August 2, 2017

1 Abstract

Symbolic regression (SR) fits symbolic expression to a dataset of expected values. Amongst its advantages over other techniques are the ability for a practitioner to interpret the resulting expression, determine important features by their usage in the expression, and an insight into the behavior of the resulting model (e.g. continuity, derivation, extrema). SR combines a discrete combinatoric problem (combining base functions) with a continuous optimization problem (selecting and mutating constants). One of the main algorithms used in SR is genetic programming. The convergence characteristics of SR using GP are still an open issue. In this work we will use a distributed GP-SR implementation to evaluate the effect of topologies on the convergence of the algorithm. We introduce and evaluate a topology with the aim of finding a new balance between diffusion and concentration. We introduce a variation of k-fold cross validation to estimate how accurate a generated solution is in predicting unknown datapoints. This validation technique is implemented in parallel in the algorithm combining both the advantages of the cross validation with the increase in covered search space for each instance. We validate our work on several test problems and a real world use case.

2 Introduction and design

Our tool is implemented in Python. The language offers portability, access to rich libraries and fast development cycles. The disadvantages are speed and memory usage compared with compiled languages (e.g. C++) or newer scripting languages (e.g Julia). Furthermore, Python's usage of a global interpreter lock makes shared memory parallelism not feasible. Distributed programming is possible using MPI.

2.1 Algorithm

The algorithm accepts a matrix X = n x k of input data, and a vector Y = 1 x k of expected data. It will evolve expressions that result, when evaluated on X, in an 1 x k vector Y' that approximates Y. N is the number of features, or parameters, the expression can use. We do not know in advance if all features are needed, which makes the problem statement even harder. The goal of the algorithm is to find f' such that

$$Y = f(X)$$

$$Y' = f'(X)$$

$$dist(Y, Y') = e$$

results in e minimal. F is the process we wish to model or approximate with f'.
Not all distance functions are equally well suited for this purpose. A simple root mean squared error (RMSE) function has the issue of scale, the range of this function is [0, +∞), which makes comparison problematic, especially if we want to combine it with other objective functions. A simple linear weighted sum requires that all terms use the same scale. Normalization of RMSE is an option, however there is no single recommended approach to obtain this NRMSE.
In this work we use a distance function based on the Pearson Correlation Coefficient. Specifically, we define

$$dist_p(Y, Y') = 1 - |r|$$

with

$$r = \frac{\sum_{i=0}^{n}(y_i - E[Y]) * (y'_i - E[Y'])}{\sqrt{\sum_{j=0}^{n}(y_j - E[Y])^2 * \sum_{k=0}^{n}(y'_k - E[Y'])^2}}$$

R has a range of [-1, 1] where 1, -1 indicate linear and negative linear correlation respectively, and 0 indicates no correlation. This function has a range [0,1] which facilitates comparison across domains and allows combining it with other objective functions. The function reflects the aim of the algorithm. We not only want to assign a good (i.e. minimal) fitness value to a model that has a minimal distance, we also want to consider linearity between Y an Y'. The use of the Pearson correlation coefficient as a fitness measure is not new, a variant of this approach is used in [9].

2.1.1 Genetic Programming Implementation

We use Genetic Programming (GP) [5] to find solution to the problem statement. The algorithm controls a population of expressions, represented as trees, that are initialized, evolved using operators, and selected to simulate evolution. The algorithm is subdivided in a set of phases, each phase initializes the population with a seed provided by an archive populated by previous phases or by the user. A phase is subdivided in runs, where each run selects a subset of the population, applies operators and if the application of the operator leads to fitness improvement replaces expressions. At the end of a phase the best expressions are stored in an archive to seed consecutive phases. At the end of a phases the best expressions are communicated to other processes executing the same algorithm with a differently seeded population. The next phase will then seed its population using the best of all aggregated expressions. We use a vanilla GP implementation, with 'full' initialization method. Expressions trees are generated with a specified minimal depth. The depth of the expressions during evolution is limited by a second maximal parameter. GP differs from most optimization algorithms in this variable length representation. We use 2 operators in sequence, mutation and crossover. Mutation replaces a randomly selected subtree with a randomly generated tree. Mutation introduces new information, and leads to exploration of the search space. Crossover selects 2 trees based on fitness and swaps randomly selected subtrees. Crossover tends to lead to exploitation of the search space. Selection for crossover is random biased by fitness value. A stochastic process decides if crossover is applied pairwise (between fitness ordered expressions in the population) or at random. The initialization of expression trees can lead to invalid expressions for the given domain. The probability of an invalid expression increases exponentially with the depth of the tree. A typical example of an invalid tree is division by zero. While some implementations opt for a guarded implementation, where division is altered in semantics to return a 'safe' value if the argument is zero, we feel that this alters the semantics of the results, and is somewhat opaque to the user. Our implementation will discard invalid expressions and replace them with a valid expression. Another approach is assigning a maximal fitness value to such an expression, but this can lead in corner cases to premature convergence when a few valid expressions dominate the population early on. We implement an efficient bottom up approach to construct valid trees where valid subtrees are merged. In contrast to a top down approach this detects invalid expressions early and avoids unnecessary evaluations of generated subtrees. Nevertheless, the initialization problem leads to a significant computational cost in the initialization stage of each phase and in the mutation operator.

2.2 Distributed algorithm

GP allows for both fine grained and coarse grained parallelism. Parallel execution of the fitness function can lead to a speedup in runtime, but will not alter the search process. Python's global interpreter lock and the cost of copying expressions for evaluation makes this approach unfeasible. A more interesting approach is coarse grained parallelism where we execute k instances of the algorithm in parallel and let them exchange their best expressions given a preset topology. The topology will determine both the convergence of the algorithm and the runtime.Exchanges of messages can introduce serialization and deadlock if the topology contains cycles. Our tool supports any user provided topology so must be able to deal with both issues effectively. After each phase a process looks up its targets given the topology. It then sends its best k expressions to the set of targets, either by copying all expressions to all targets or by spreading them over the target set. When the sending stage is complete, the process looks up its set of sources and waits until all source processes have sent their best expressions. To avoid deadlock a process sends its expressions asynchronously, not waiting until the receiving process has acknowledged receipt. The sent expressions are stored in a buffer for this purpose, together with an associated callable waiting object. After the sending stage the process synchronously collects messages from its sources, and executes the next phase of the algorithm. Before the next sending stage, the process will then check each callable to verify that all messages from the previous sending phase have been collected. Once this blocking call is complete, it can safely reuse the buffer and start the next sending phase. This also introduces a delay factor between processes. The phase runtime between processes will never be exactly identical, especially not given that the expressions have a variable length representation and differing evaluation cost. Without a delay factor processes would synchronize on each other, nullifying any runtime gains. With this delay factor a process is able to advance k phases ahead of a target process k steps distant in its topology. For hierarchical, non-cyclic topologies this can lead to a perfect scaling, where synchronization decreases as the number of processes increases.

2.3 Approximated k-fold cross validation

We divide the data over k processes with each process use a random sample of 4/5 of the total data. Each process operates on a further 4/5 split between training and validation data. The aggregate distributed process then approximates k-fold cross validation. Independent of the topology each randomly chosen pair of communicating processes will have the same probability of overlapping data. When this probability is too low, overfitting will be introduced and highly fit expressions from one process will have a high probability to be invalid for another process' training data. When the overlap is too great both processes will be searching the same subspace of the search space.

2.4 Topologies

A topology in this context is the set of communication links between the processes. The topology influences the convergence characteristics of the entire group. In a disconnected topology, there is no diffusion of information. If a process discovers a highly fit expression, that knowledge has to be rediscovered by the other processes in order to be used in the process. An edge case where this is an advantage is if we see the group of processes as a multiple restart version of the sequential algorithm. If the risk for premature convergence due to local optima is high, we can try to avoid those optima by executing the algorithm in k instances, without communication. Such an approach is sometimes referred to as partitioning, as we divide the search space in k distinct partitions.

Diffusion and concentration Our aim is for the processes to share information in order to accelerate the search process. With a topology we introduce two concepts : concentration and diffusion. Concentration refers to processes that share little or no information and focus on their own subset of the search space. Like exploitation concentration can lead to overfitting and premature convergence. It is warranted when the process is nearing the global optimum. Diffusion, in this work, is the spread of information over the topology. Diffusion accelerates the optimization process of the entire group. It is not without risk, however. If diffusion is instant, a single suboptimal solution can dominate other processes, leading to premature convergence. The distance between processes and connectivity will determine the effect diffusion has.

Grid The grid topology is 2 dimensional square of k processes with k a square of some natural number. Each process is connected with 4 other processes in north/south, east/west direction. The grid allows for delayed diffusion, to reach all processes an

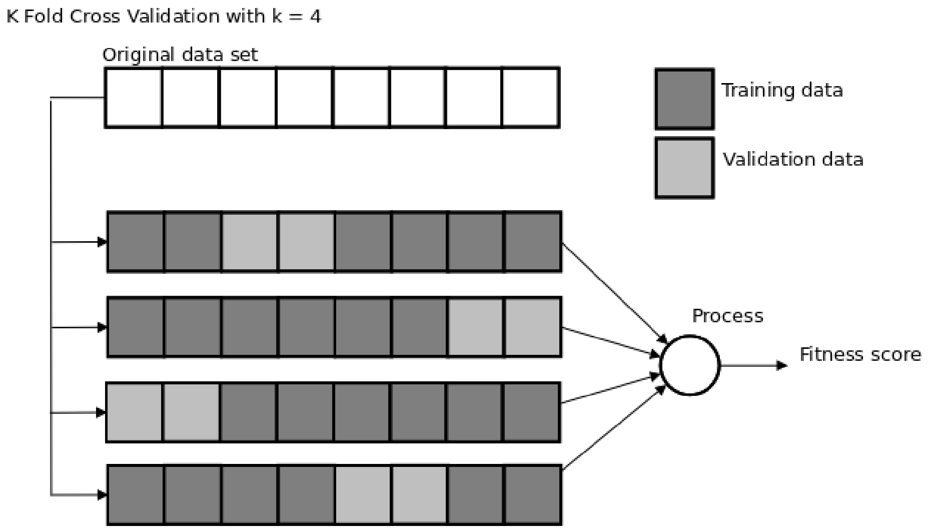
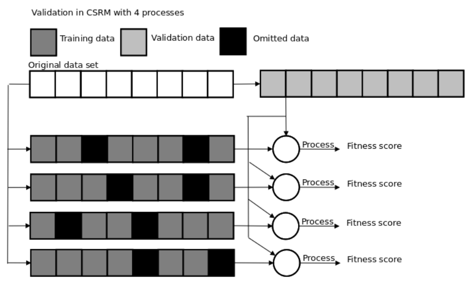


Figure 1: Visualization of k fold cross validation with k = 4.

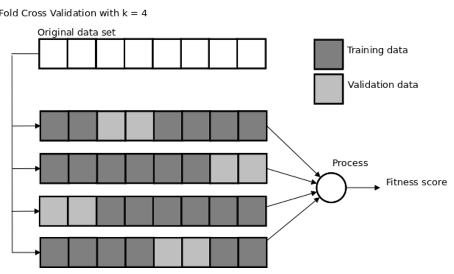
optimal expressions needs to traverse $\sqrt{k}$ links. This prevents premature convergence, with all processes still interconnected.

Tree A binary tree with with root as a source and leafs as sinks with unidirectional communication, the tree topology is an efficient example of a hierarchical topology. For k processes there are k-1 communication links, reducing the messaging and synchronization overhead significantly compared to the grid topology (4k). Diffusion can be hampered, each link is unidirectional so an optimal expression will not travel upwards to the root. On the other hand, with a spreading distribution policy where the optimal expressions are spread over the outgoing link, a partitioning effect will occur which can prevent premature convergence. There are no cycles, synchronization overhead is low.

Random In a random topology it is hard to predict how the convergence of the aggregated process will behave. It is possible that the topology contains cycles, but at the same time it can contain cliques. Diffusion is not guaranteed, and runtime performance will differ based on the actual instance. With grid and tree convergence and synchronization behavior is deterministic. The advantage of the random topology is that it can avoid certain patterns that occur in the other deterministic topologies. If the grid tends to lead to premature convergence for a given problem instance, it is possible that a random topology will avoid this.



(a) Approximation of k fold cross validation with parallel processes, k = 4, r = $\frac{3}{4}$.



(b) Visualization of k fold cross validation with k = 4.

Figure 2: Topologies in CSRM.

3 Experiments

3.1 Reproducability

All benchmarks were performed on an Intel Xeon E5 2697 processor with 64GB Ram, with Ubuntu 16.04 LTS, kernel 4.4.0 as operating system. CSRM is implemented using Python3, the test system uses Python 3.5. The experiments use a fixed seed in order to guarantee determinism. Where relevant the configuration is given. An open source repository holds the project's source code, benchmark scripts, analysis code and plots. The project dependencies are minimal making the project portable across any system that has a working Python3 implementation and pip as an installation manager. In order to run distributed the project requires an MPI implementation, which is available for most platforms.

3.2 Benchmark problems

Recent work on the convergence of GP-based SR [3, 4] featured a set of benchmark problems that pose convergence problems for SR implementations. These 15 problems use at most five features. CSRM does not know which features are used, making the problem harder. In other words it assumes each problem is a function of 5 features which may or may not influence the expected outcome. This is an extra test in robustness for the algorithm, while also testing the algorithm's capability as a classifier.

3.3 Experiment setup

- population : 20
- minimum depth : 4
- maximum depth : 8
- phases : 20

- generations per phase : 20
- datapoints : 20
- range : [1,5]
- features : 5
- archivesize : 20
- expressions to archive per phase : 4
- optimization strategy : none
- communication size m : 2, 4
- topology : Tree, Random, Grid
- spreadpolicy : distribute
- processes n : 25

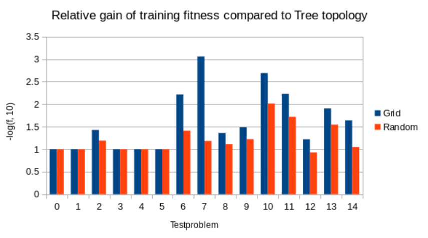
Discussion From section **??** we know that simply using m for all topologies will lead to unintended communication patterns. We test Tree and Random with m = 2, and Grid with m = 4. This results in the following number of messages per link :

- Grid : 1
- Tree : 1
- Random : 2

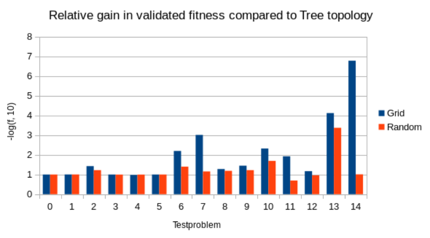
The total number of messages sent per phase is then :

- Grid : 4 * n = 200
- Tree : 1 * n = 25
- Random : 2 * n = 50

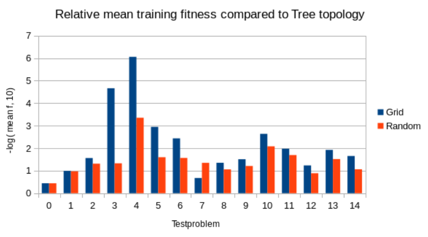
The value of m = 2 for Tree and m = 4 for Grid follows from our discussion in **??**. The Random topology in this configuration has a single outgoing link per process, resulting in 2 messages per link. This configuration forms a balance between the different strategies. For 25 processes the grid topology is a simple square of 5x5. The tree topology with 25 processes is a non full binary tree with depth 4. The random topology is highly dependent on the seed. In this instance the topology has become a disconnected graph of 2 cycles. This highlights the risk of using a random topology, without extra constraints the characteristics of the communication pattern are unknown and can be undesirable.



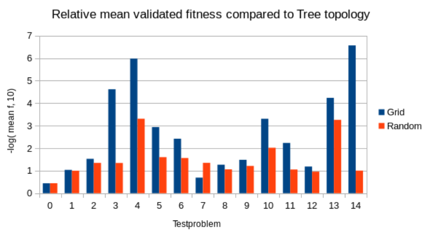
(a) Relative gain in best fitness of training data



(b) Relative gain in best fitness on full data.



(c) Relative gain in mean fitness on training data.



(d) Relative gain in mean fitness on full data.

Figure 3: Convergence differences between topologies.

3.3.1 Measures

When the experiment ends the best 20 expressions from all processes are collected and scored. We measure best fitness on the training data, and best fitness on the validated data. Finally we record the mean fitness values of the best 5 expressions, both on the training and validation data. The mean is restricted to the upper quarter of the population specifically to measure how the best expressions are distributed. This measure records the convergence more accurately as the fittest expressions drive the convergence rate.

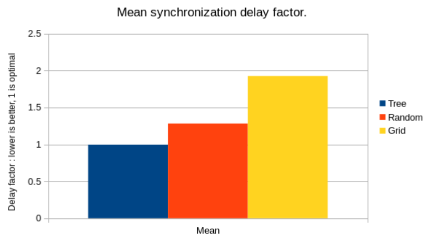
Calculation The fitness values fluctuate strongly between the test problems and even between topologies. We apply a negative logarithmic scale :

$$f_t = -\log_{10}(f)$$

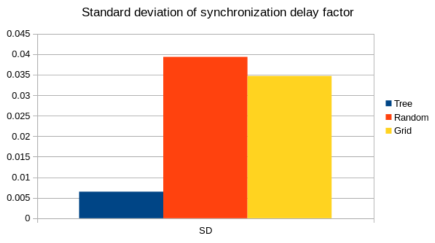
where f is either the best fitness value or the mean. Then we scale the results relative to the values obtained for the tree topology in order to measure relative gain or loss in orders of magnitude.

3.3.2 Results

Convergence In Figure 3 we see how the topology affects convergence. The first observation we make is that the first 5 testproblems, with exception of the second, all have identical values for best fitness on training and validation data. The processes converged to zero fitness values for these problems, hence the identical results. The



(a) Mean synchronization delay factor.



(b) Standard deviation of synchronization delay factor.

Figure 4: Synchronization overhead introduced by topologies.

training fitness results in Figure 3a indicate that the grid and random topology have superior convergence characteristics compared to the tree topology, with grid outperforming random on several testproblems. When we look at the fitness values on the validation data in Figure 3b we see a more nuanced result. The grid topology is overall still the best choice, but the random topology has worse results for problems 11 and 12. Next we look at the mean fitness values on training and validation data. Interestingly enough for problem 0 both random and grid score far worse than the tree topology. Overall the grid topology is still better for most problems, with the exception of problem 7. Note the similarity in pattern in the results here between training and validation data indicating that the predictive capability of the results is still good. If overfitting would have taken place we would see a reverse pattern in the results for the validation data.

Synchronization overhead From our discussion in **??** we know that cycles in the topology will lead to excessive synchronization and even serialization. We measure the mean execution time for testproblem 6. The convergence characteristics using the three topologies differed significantly making this a good testcase. The processes will communicate 25 times. If the runtime of a single phase is too long, the overhead of communication will become hard to measure. If it is too short the overhead dominates the entire runtime. This second case is one we should try to avoid, it will unfairly penalize topologies with cycles forcing them to serialize. The runtime of a phase is dependent on the generations, population and depth ranges of the expressions. Ideally we would like for a practitioner to choose these parameters based on the problem at hand and not be constrained by synchronization considerations. We compare the three topologies and use the disconnected or 'none' topology as a reference point, which has zero synchronization overhead. This last topology has an ideal speedup of n, where n is the processcount, compared to a sequential process. From the synchronization overhead we can then derive the speedup each of the topologies is able to offer. In practice even the 'none' topology will have some synchronization overhead, as the root process has to collect all results from the other processes. In Figure 4 we see that the tree topology has a near zero delay introduced by the synchronization. This is due to the delay tolerance we have built in in our implementation. The random topology has a mean delay factor of 1.3, the grid topology scores worst with a mean delay factor of nearly 2. This is easily translated in terms of speedup. A tree topology will have near linear speedup, a grid will have a speedup roughly half of that value and a random topology will have a speedup bracketed between those two. The standard deviation for the tree topology is significantly smaller indicating that a tree topology will have a far more predictable speedup.

Memory overhead Memory overhead is hard to measure in a language with a garbage collector. We can estimate the overhead by calculating the needed memory in function of depth and topology used. Let the depth be constrained by $[d_i, d_x]$, with n processes, m communicationsize and a distribution spreading policy. D_i is the minimum depth and d_x the maximum depth. If we let $d_a = \frac{d_i + d_x}{2}$ be the average depth, then the memory requirements on average for each topology are then given by

- Tree : $d_a \frac{m}{2}(n-1)$
- Random : $d_a \frac{m}{n}$
- Grid : $d_a \frac{m}{4} 4n$

Note that d_i is not necessarily equal to the initial depth. While rare, it is possible that CSRM evolves trees with a depth lower than the initial depth. Unless we constrain the operators from doing so trees will start with a depth of d_i but then vary between 1 and d_x. Here m $>=$ 4 for a grid if we use distributing spreading policy. This leads us to an important observation. The Tree topology can communicate expressions with an average depth that is 2 times greater than the one used by the grid with the same memory usage. This factor is important, an increase in depth has an exponential effect on the complexity of the entire algorithm but also allows for more complex solutions. In addition an increased depth tolerates more bloat without losing accuracy.

4 Related Work

Enrique Alba's book [1] on parallel metaheuristics is the reference work for the field. It provides a broad overview of the challenges and advantages of parallel metaheuristics. SR can be implemented using a parallel metaheuristic such as GP. A Python toolbox for evolutionary algorithms has been developed [2], not specifically directed at symbolic regression but as a repository of evolutionary algorithms in a distributed context. An interesting parallel GP SR implementation [8] introduces a random islands model where processes are allowed to ignore messages, contrary to our approach. The authors argue that this promotes niching, where 'contamination' of locally (per process) fit individuals could otherwise introduce premature convergence. The clear advantage of such a system is speedup, since no process ever waits on other processes. Another difference is the message exchange protocol. Whereas our tool exchanges messages after each phase, their tool uses a probability to decide per process if messages are sent or received interleaved with the generations. Such a setup allows for a heterogeneous set of processes. A different approach is shown in [7] where a master slave topology is used in combination with a load balancing algorithm in order to resolve the imbalance between the different slaves executing uneven workloads. The slaves do not form separate processes, they are assigned a subset of the population and execute only the fitness function. The selection and evolution steps are performed by the master process. This a a fine grained approach, and while it offers a speedup in comparison with a sequential GP SR implementation it does not increase the coverage of the search space. In Distributed Genetic Progamming (DGP) [6] a ring and torus topology are used. The two way torus topology is similar to our grid topology. The study finds that sharing of messages is essential to improve convergence but that the communication pattern is largely defined by the problem domain. It concludes that diffusion is a more powerful technique compared to partitioning. In partitioning no communication between subgroups is possible, which can protect against premature convergence.

5 Conclusion

6 Future work

Using our distributed architecture it is possible to give each process a different optimization algorithm, creating a heterogeneous set of communicating algorithms. This has two advantages. First, it allows for comparison of different optimization algorithms within the same framework. Second, it would make the SR tool more robust. We know from the NFL theorem that no optimization algorithm is optimal for all optimization problem instances. A cooperative set of optimizations algorithms could offer an optimal solution for all problem instances by balancing the disadvantages and advantages of each algorithm.

Topologies The inverted tree topology, where the root is a sink and leaves are sources, is an interesting alternative to the original tree. Future work will evaluate other communication strategies such as random sampling. A random tree topology could offer a balance between convergence and speedup. This approach would aim to combine the advantages of a stochastic approach with the speedup gains of the tree structure itself.

Incremental DOE Dividing the DOE generated input points into sections and using them in the regression tool can lead to issues regarding the structural properties of the design. While our results indicate that the model obtained from an incremental run has a higher quality compared to that of a unseeded run, this does not exclude the possibility that the seed we used corresponds with a biased coverage of the parameter space. An alternative approach would be to increment not datapoints but parameters. The Latin Hypercube Design maintains its structural properties when parameters collapse or are removed.

References

[1] ALBA, E. *Parallel Metaheuristics: A New Class of Algorithms.* Wiley-Interscience, 2005.

[2] FORTIN, F.-A., DE RAINVILLE, F.-M., GARDNER, M.-A. G., PARIZEAU, M., AND GAGNÉ, C. Deap: Evolutionary algorithms made easy. *J. Mach. Learn. Res. 13*, 1 (July 2012), 2171–2175.

[3] KORNS, M. F. *Accuracy in Symbolic Regression*. Springer New York, New York, NY, 2011, pp. 129–151.

[4] KORNS, M. F. *A Baseline Symbolic Regression Algorithm*. Springer New York, New York, NY, 2013, pp. 117–137.

[5] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[6] NIWA, T., AND IBA, H. Distributed genetic programming: Empirical study and analysis. In *Proceedings of the 1st Annual Conference on Genetic Programming* (Cambridge, MA, USA, 1996), MIT Press, pp. 339–344.

[7] OUSSAIDÈNE, M., CHOPARD, B., PICTET, O. V., AND TOMASSINI, M. Parallel genetic programming: An application to trading models evolution. In *Proceedings of the 1st Annual Conference on Genetic Programming* (Cambridge, MA, USA, 1996), MIT Press, pp. 357–362.

[8] SALHI, A., GLASER, H., AND DE ROURE, D. Parallel implementation of a genetic-programming based tool for symbolic regression. *Inf. Process. Lett. 66*, 6 (June 1998), 299–307.

[9] WILLEM, L., STIJVEN, S., VLADISLAVLEVA, E., BROECKHOVE, J., BEUTELS, P., AND HENS, N. Active learning to understand infectious disease models and improve policy making. *PLOS Computational Biology 10*, 4 (04 2014), 1–10.