Parallel Genetic Programming: an application to Trading Models Evolution

Mouloud Oussaidène  
Bastien Chopard  
Olivier V. Pictet  
Marco Tomassini

ABSTRACT
We present a parallel implementation of genetic programming on distributed memory machines. To overcome the time overhead due to uneven load associated with program evaluation, we propose and evaluate a non-preemptive dynamic scheduling algorithm for load balancing. The system is applied to the evolution of trading model strategies which is a compute-intensive application. Our results show that reasonable trading models can be inferred and that the system can produce a nearly linear speed-up for that application.

1. Introduction
Artificial evolutionary processes, such as genetic algorithms (GA) (Holland 1975), are based on reproduction, recombination and selection of the fittest members in an evolving population of candidate solutions. (Koza 1992) extended this genetic model of learning into the space of programs and thus introduced the concept of genetic programming. Each solution in the search space is represented by a genetic program (GP), traditionally using the Lisp syntax. Genetic programming is now widely recognized as an effective search paradigm in artificial intelligence, databases, classification, robotics and many other areas.

We present here a parallel genetic programming system (PGPS) which speeds up the searching process. In complex applications, such as trading model optimization, the evaluation of each GP is very long and thus the time spent in the selection and reproduction phases is practically negligible compared with the population evaluation time. Parallelizing the evaluation phase on a parallel coarse-grain machine such as the IBM SP-2, can be done easily. After the reproduction phase, each GP is simply sent to a processing node for evaluation, independently of operations in other processing nodes (Oussaidène, Chopard and Tomassini 1995). However, the run time GP complexity with programs of widely differing sizes makes the design of a parallel algorithm and its implementation to obtain large speed-ups a nontrivial task. This irregularity causes some processing nodes to be idle while others are active.

We describe a parallel scheme in which the evaluation phase is separated from the rest of GA calculations. We first consider a static scheduling algorithm to distribute the genetic programs upon the processing nodes at the evaluation phase. Next, we improve the processor utilization by a dynamic load balancing algorithm based on run time GP complexity. Our implementation shows that, for problems of large enough size, the parallelization of genetic programming on distributed memory machines can deliver a nearly linear speed-up. The code was written in C++ using the PVM3 message passing library (Geist et al. 1994) and can be run on other parallel machines such as the Cray T3D. The sequential version of the code is based on (Fraser 1994).

The organization of this paper is as follows. The evolution of genetic programs is outlined in section 2. Section 3 discusses the PGPS architecture. Section 4 highlights the load balancing problem and analyzes two different techniques for solving the problem. The parallel GP system has been applied to a difficult problem of trading model search described in section 5. We discuss our experimental results in section 6 and concluding remarks are made in section 7.

2. Evolution of genetic programs
A GP can be regarded as a Lisp function. It is usually represented by a parse tree.
The number of nodes (including the terminal nodes) in the parse tree gives a measure of the complexity of that GP. Thus, the GP $(\ast (\ast (x \ y) \ t \ b))$ is of complexity 7. The terminal (non-terminal) nodes are randomly taken from some well defined TerminalSet (FunctionSet). In the creation phase, each parse tree is built in a recursive way, starting from the root node. To ensure the syntax validity and control the complexity expansion of GPs during this creation process, some rules must be observed. If the current node in the parse tree is a function taking $r$ arguments then choose $r$ nodes, from the Terminal/FunctionSet, to be child nodes. In the other case, the current node is a terminal so return to the parent node. A maximum depth, $d$, is fixed before the execution so that when a branch in the parse tree reaches this level then a terminal must be chosen. The depth level corresponding to the root node is $d = 0$.

Let $r$ be the maximum function arity in the FunctionSet. At run time, the complexity, $C$, of a GP is therefore bounded by the formula (2.1).

$$1 \leq C \leq \frac{d + 1}{r - 1} \quad \text{if} \quad (r \neq 1)$$
$$1 \leq C \leq d + 1 \quad \text{if} \quad (r = 1)$$

(2.1)

Two genetic operators are used in the reproduction phase; crossover and mutation. The crossover operator selects two GPs from the population and chooses one node (crosspoint) on each. Each node is, by definition, the root of some complete subtree. The two subtrees are extracted and each is swapped with the other. Note that the resulting GPs are syntactically valid and only the maximum depth constraint is checked for non-violation.

For illustration, if we choose the nodes 4 and 2 (numbering from the left) respectively on the GPs $(\ast (\ast (x \ y) \ t \ b))$ and $(\ast (\ast a b))$ then the new GPs will be $(\ast (\ast a b) \ t \ y \ z)$ and $(\ast (\ast (x \ y) z))$. The mutation genetic operator is unary. A GP is modified according to some rules. Any terminal node can be replaced by any other terminal taken from the TerminalSet, a function node can only be replaced by another function (from the FunctionSet) which has the same number of arguments. This operator does not modify the complexity of the GP. The evaluation phase consists of assigning a fitness value to each GP in the population. This fitness calculation requires evaluating each GP as many times as there are fitness cases related to the problem to solve. Particularly, in learning trading models, each GP is evaluated on various price time series where each series element represents a fitness case. In this analysis, the problem size is given by $n = p \times g \times e$ where $p$, $g$, $e$ indicate respectively the population size, number of generations and number of fitness cases.

3. PGPS scheme

Figure 1 shows the parallel scheme and the i/o communication interface between the genetic programming management system and the user defined problem. The architecture of PGPS follows the master-slave model. The master process is the genetic programming management system and the slave process is the user problem evaluation program. The genetic programming management system creates the initial population, applies the genetic operators (crossover and mutation) and performs the selection of genetic programs which will be the candidates for the reproduction phase. At the evaluation phase, the master process distributes the work load among all the processing nodes in the virtual machine, including the processor on which the master process runs. This computational load distribution can be accomplished by different strategies. In the next section we analyse a static scheduling algorithm and a dynamic one. The tasks assigned to the master process require no communication, they are executed locally. The communications between the master and slave processes are achieved using communication interface routines. The genetic programming management system packs each parse tree from its memory representation into a buffer and sends it to the slave processes using PVM routines. Each slave process receives the character string into a buffer and performs the unpack operation to build the equivalent parse tree in memory. After the fitness calculation of a genetic program, each slave process sends back the calculated fitness value to the master process. Note that GPs can be evaluated independently and therefore there is no communication between the slaves. The load distribution cost and the communication overhead are negligible compared with the computational cost associated with fitness calculation. Each message is sent individually and the messages belonging to the same destination are not clustered as a single message. This technique avoids delaying the fitness computations and allows the slave processes to be further kept busy. Thereby, fitness computations and work distribution overlap. The
evaluation phase ends when the whole population is evaluated (each processing node has performed its assigned tasks). The termination signal of parallel execution is given when a preassigned maximum number of generations has been attained. The main steps of the parallel algorithm are shown in Figure 2.

Master process:
0: Load the slave processes.
1: Create the initial population.
2: Distribute the work load to the processing nodes.
3: Execute the tasks assigned to the Master process.
4: Receive the fitness values sent by the slave processes.
5: Perform the selection phase.
6: Perform the reproduction phase.
7: Repeat Steps 2 - 6 up to the maximum number of generations.
8: Terminate the slave processes.
9: End.

Slave process:
0: Receive a genetic program.
1: Calculate the fitness of the received program.
2: Send the fitness value to the Master process.
3: Repeat Steps 0 - 2 until reception of termination signal.
4: End.

fig 2: A parallel algorithm for Genetic Programming.

4. Load balancing

Load balancing refers to equal distribution of the computational load among the processing nodes. In the terminology used here, task is the fitness computation and task size refers to the GP complexity. A task is the smallest unit that can be scheduled on a processing node. Load balancing can be either static or dynamic.

4.1. A static scheduling algorithm

This algorithm distributes the work load in an ordered way defined at compilation time. The Round-Robin policy is used to assign tasks to the processing nodes. The criterion of load balancing upon the processors is the number of tasks assigned to each processor. Task \( i \), in the task pool to be distributed over \( m \) processors, is assigned to the processing node given by \( \lfloor i/m \rfloor \). This algorithm handles only static parameters, fixed at compilation time, and does not reflect dynamic factors which may change during the system evolution at run time. Even though the algorithm regulates the number of tasks assigned to each processor, irregularity in computational load may occur. This overhead is due to the fact that tasks are not of the same size and the work load on a processing node depends not only on the number of tasks to be performed, but also on the size of these tasks. This irregularity at the complexity level induces differences in the task evaluation time which may unbalance the work load on the processing nodes.

4.2. A dynamic scheduling algorithm

In the dynamic load balancing scheme, the decision of which processor should run a specified task is made at run time. The processor allocation is a function of population complexity at each generation of the evolutionary process. The work load on a processing node is considered as the sum of each task size assigned to that node. This problem can be formulated as follows.

Given \( m \) processing nodes \( M_i \) \((i = 1, \ldots, m)\) and \( p \) tasks \( T_j \) \((j = 1, \ldots, p)\), one wishes to find a schedule of minimum length. A schedule is optimal if the maximum task completion time is minimum. Note that this problem is known to be NP-hard. Let \( l_i \) be the work load of processor \( i \). The algorithm is shown in figure 3. It is based on a “greedy” heuristic giving, on average, a well approximated solution to the problem.

Sort \( C_j \) in Downward order

\[ l_i \leftarrow 0 \quad \text{(} i \in \{1 \ldots m\}\text{)} \]

\[ \text{For } j \leftarrow 1 \text{ to } p \text{ do} \]

\[ \text{Assign } T_j \text{ to } M_{i^*} \]

\[ l_{i^*} \leftarrow l_{i^*} + C_j \]

End

fig 3: A dynamic scheduling algorithm.

The tasks are sorted by descending size. The task distribution starts from the largest task. At each iteration, the algorithm assigns the current task to the least loaded processing node. When a processor is selected to perform a task its work load is increased by the size of that task.

5. Evolving trading models

Besides alleviating the restrictions of fixed-length representation of genetic structures, genetic programming provides a natural way to represent and evolve decision trees (Allen and Karjalainen 1993). The application presented here describes a way to use genetic programming to learn technical trading models for foreign exchange (FX) market. The recommendations given by a trading model are purely based on past prices of the exchange rate being analyzed. The price history is summarized in the form
of variables called indicators. A trading model is a system of rules catching the movement of the market and providing explicit trading recommendations for financial assets. A simple form of a trading rule could be

\[
\text{IF } |I| > K \text{ THEN } G := \text{SIGN}(I) \quad (5.1)
\]

ELSE \( G := 0 \)

Where \( I \) is an indicator whose sign and value model the current trend and \( K \) is a threshold constant called breaklevel. The gearing, \( G \), is the recommended position of the model. The value \( G = +1 \) corresponds to a ‘buy signal’, \( G = -1 \) corresponds to ‘sell signal’ and \( G = 0 \) corresponds to the neutral position (stay out of the market). A trading rule with a more complex strategy may use more than one indicator: typically, an indicator giving the current trend can be used in conjunction with an indicator reflecting the volatility of prices. Since no single indicator can ever be expected to signal all trend reversals, it is essential to combine a set of indicators so that an overall picture of the market can be built up. The trading models used in this study are decision trees seen as logical combinations of rules having the form (5.1). The decision trees are evolved using genetic programming. An indicator is a function of time series and, in particular, all the indicators used here are functions of time and price history. The indicator construction is based on the concept of momentum. The price momentum is essentially the deviation between the current price and the mean of multiple exponential moving averages (EMA) used for smoothing the price time series \( x(t) \). An EMA is characterized by its range (in days), \( r \), and its order (the number of successive applications of EMA to the initial price time series). A full description of momentum computation is given in (Pictet et al. 1992). We use the notation \( I_{s,r,j,a} \) to refer to the normalized price momentum based on \( r \) days EMAs of order in \([J, n]\).

The major problem in optimizing trading models is to avoid overfitting caused by the presence of noise. Overfitting means building trading models that fit a price history very well but generalize badly. In order to avoid this phenomena, the idea is to build genetic programs based on pre-optimized building blocks. These building blocks (robust indicators) were optimized using a niching genetic algorithm based on a fitness sharing scheme (Pictet et al. 1995). The fitness measure of a trading model quantifies not only the return but also the risk involved by the model. The total return of a model is obtained by continuously accumulating the return of each transaction (which is null for transactions starting from the neutral position). A transaction holds each time a new signal is generated.

If \( p_1 \) and \( p_2 \) are the prices of two consecutive transactions, then the return of the second transaction is given by the ratio \( (p_2 - p_1) / p_1 \).

The fitness function, termed \( X_{\text{eff}} \), is defined as:

\[
X_{\text{eff}} = \langle R \rangle - \frac{C \sigma^2}{2}\quad \text{where } \langle R \rangle \text{ is the annualized average total return, } C = 0.1 \text{ is a risk aversion constant and } \sigma^2 \text{ is the variance of the total return over time. Since the variance is a measure of the stability of the return, then high effective return means also high stable return. The notion of robustness is directly related to the ability of generalizing the results beyond the training sample. For this purpose, each trading model is tested on more than one exchange rate time series. The fitness measure is then extended as follows:}

\[
X_{\text{eff}} = \langle X_{\text{eff}} \rangle - \frac{\sigma^2_{\text{eff}}}{3}
\]

where the first term is the average fitness value obtained for each exchange rate and the second term is proportional to the variance of these values.

The \( \text{FunctionSet} \) used in the construction of the trading strategies is composed of the logical operators \{ AND, OR, NOT, IF \} and the \( \text{TerminalSet} \) is composed of the pre-optimized indicators and the values \{ +1, -1 \}. Because of the presence of three possible values (-1, 0, +1) for the deal signal, we have redefined these logical operators. The \( \text{OR} \) operator returns the sign of the sum of its arguments, the \( \text{NOT} \) function returns the opposite decision of the argument, the \( \text{AND} \) function returns the neutral signal when one of its arguments is zero; otherwise returns the \( \text{OR} \) value. The \( \text{IF} \) function takes three arguments. It returns the second argument if the first one is true; otherwise it returns the third argument.

The indicators are revalued at each price event. Once updated, the normalized value \( I_{s,r,j,a}(t) \) is used according to rule (5.1) to obtain the indicator signal \( G_{s,r,j,a} = G(I_{s,r,j,a}(t)) \). The different signals returned by the indicators are then embedded in the logical \( S \)-expression corresponding to the trading model. The value obtained by the evaluation of the \( S \)-expression represents the trading model recommendation.

6. Experimental results

This section discusses both parallel programming and trading model performances. Given the large size of our data set, the speed-up measurements were obtained by limiting the number of fitness cases to 1000 when fitting the logistic function (Oussaidène, Chopard and Tomassini 1995) and are consistent with the speed-ups obtained for the trading model application.
6.1. Parallel programming performance

The parallel genetic programming system was implemented on an IBM SP-2 machine. SP-2 is a distributed memory machine using a network based computing model, providing two types of processing nodes: Wide and Thin nodes. The POWER2 processor in the Wide processing node runs at 66.7 MHz, giving a peak performance of 266 MFLOPS. The POWER processor in the Thin processing node runs at 62.5 MHz, giving a peak performance of 125 MFLOPS. The processing nodes share data via message passing over the High Performance Switch multistage packet switched Omega network (IBM 1995). The switch chip provides 40 MBytes/s peak channel bandwidth and 500 nsec hardware latency. We used from 1 to 10 Thin processing nodes. All the speed-up measurements reported here are calculated in the traditional form. For this, we have fixed not only the problem size but also the seed for the random number generator so that the output of a parallel execution is exactly the output of its corresponding sequential execution. Other authors suggested fixing only the problem size and to average over many runs (Koza and Andre 1995) but their model is based on evolving several loosely-connected subpopulations and is different from ours. The speed-ups of both static and dynamic load balancing schemes are shown in Figure 4. The elapsed time was measured in dedicated mode. By setting the problem size as $n = 100 \times 100 \times 1000$ and the maximum depth as $d = 6$, the sequential execution takes 234.97 seconds on a single node of the SP-2. Using 10 SP-2 nodes, the static load balancing scheme completes in 34.05 seconds whereas the dynamic load balancing algorithm takes 29.57 seconds.

When increasing the run time maximum depth up to $d = 12$ and setting problem size as $n = 100 \times 10 \times 100$, the sequential execution time takes 743.77 seconds on a single node. On 10 nodes, the static scheduling algorithm completes in 140.2 seconds while the dynamic scheduling algorithm takes 105.68 seconds. We can remark that for relatively small depths, both static and dynamic load balancing schemes give approximately similar results.

According to formula (2.1), the maximum task size grows exponentially with the maximum depth parameter. The crossover genetic operator introduces some variance in the task sizes within the task pool to be distributed and this variance becomes significant for high values of the maximum depth. The presence of this task size variability unbalances the work load distribution based on the static load balancing scheme which delivers a speed-up of 5.3 on 10 processors for a maximum depth of 12 while the dynamic load balancing scheme gives a speed-up of 7.

6.2. Trading model performance

The optimization of the trading models is performed on seven exchange rates (GBP/USD, USD/DEM, USD/ITL, USD/JPY, USD/CHF, USD/FRF, USD/
GENetic programming has been used to infer robust trading models. Although profitable, the average returns provided by the models present some variability, partly due to overfitting, during the test period. Our preliminary results show that genetic programming indicates a promising direction for trading model optimization which is not a well understood domain. Also, when addressing such highly complex problems it becomes necessary to decompose the problem representation so that an overall solution can be viewed as a combination of small modules.

**Acknowledgments**

The authors would like to acknowledge the Swiss National Science Foundation for its financial support. We would like to thank Olsen & associates research institute for providing the price data and making valuable contributions in understanding financial trading models.

**Bibliography**


Fraser, A. P. 1994. Genetic programming in C++, public domain genetic programming system.


