FlexGP 2.0: Multiple Levels of Parallelism in Distributed Machine Learning via Genetic Programming

by
Dylan J. Sherry

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2013

Author ....
Department of Electrical Engineering
and Computer Science
September 12, 2013

Certified by ....
Kalyan Veeramachaneni
Research Scientist
Thesis Supervisor

Certified by ....
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Accepted by ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

FlexGP 2.0: Multiple Levels of Parallelism in Distributed Machine Learning via Genetic Programming

by

Dylan J. Sherry

Submitted to the Department of Electrical Engineering
and Computer Science
on September 9, 2013[1], in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents FlexGP 2.0, a distributed cloud-backed machine learning system. FlexGP 2.0 features multiple levels of parallelism which provide a significant boost in speed and accuracy. The amount of computational resources used by FlexGP 2.0 can be scaled along several dimensions to support large and complex datasets. FlexGP 2.0's core genetic programming (GP) learner includes multithreaded C++ model evaluation and a multi-objective optimization algorithm which is extensible to pursue any number of objectives simultaneously in parallel. FlexGP 2.0 parallelizes the entire learner to obtain a large distributed population size and leverages communication between learners to increase performance via the sharing of search progress. FlexGP 2.0 features parallelized subsampling of training data and other parameters to boost performance and to enable support for increased data size and complexity.

We provide rigorous experimental verification of the efficacy of FlexGP 2.0's multi-level parallelism. Experiments are on a large dataset from a real-world regression problem. The results FlexGP 2.0 provides an increase in the speed of computation and an overall increase in accuracy, and illustrate the value of FlexGP 2.0 as a platform for machine learning.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Research Scientist

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

[1]This thesis is a revised version of the physical copy originally submitted to MIT EECS. Revisions were made with the full approval and consent of the thesis supervisors.

Acknowledgments

I have spent almost three years benefiting from the guidance of and collaboration with Kalyan Veeramachaneni & Una-May O'Reilly. You're both top-notch researchers and fantastic people; I've deeply enjoyed working with you and thank you for all you've done for me.

Owen Derby was my primary collaborator on FlexGP. His work on FlexGP and on factorization is a valuable cornerstone of this thesis. His ability to get at the crux of a problem is second to none. Owen is solid proof Mainers are the best. Thanks Owen.

I must thank Ignacio Arnaldo for his keen advice and for his crucial contributions on FlexGP's C++ fitness evaluation.

Thank you to all my friends from 32-D433 and to the rest of the ALFA group for the company, the many tea times, and for providing me with a constant reminder of how incredible the people at MIT are.

To all the folks in the course VI undergrad office, and in particular to Anne Hunter, I send my thanks for five years of friendship and guidance. I'd also like to thank the entire MIT community for granting me my first true intellectual home.

To Maryam Yoon and to my family for their love and support over many years: I am an unfathomably lucky person to have you in my life.

Last but not least, I am deeply grateful to GE Global Research for their support of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen much growth and change in the size and availability of data. Technological proliferation has allowed data to be more easily collected, organized and stored. The diversity of data size, structure and topic has seen similar growth. Data has become available in a variety of domains which previously were sparsely populated, including the documentation of social interaction, semantic content, medical instrumentation and measurements, and educational information. At present it appears the growth in data size, availability and diversity will continue indefinitely.

The increased availability of data in each domain represents a greater opportunity to learn about the world, given the ability to leverage the data effectively.

However, many existing learning algorithms cannot provide high accuracy in a short amount of time, relative to the size and complexity of data under consideration. A larger data size translates to increased runtime which can easily be super-linear in relation to the data size and complexity. Datasets which represent an expression of more complex relationships require increasingly more complex models to provide desirable results, which further contributes to the computational slow-down. From the perspective of search, a greater size and complexity of data increases the likelihood the search will eventually find itself stuck in undesirable local minima or maxima.

The FlexGP Distributed Machine Learning System (FlexGP) provides swift and accurate machine learning on large and complex data sets. FlexGP seeks to be able to scale with data size and complexity in a graceful and tractable manner. FlexGP's goals are two-fold:

Faster Learning: decrease the time required to obtain accurate results.

Better Learning: provide an accuracy which was previously unobtainable.

Modern cloud frameworks provide a vast wealth of computational resources; FlexGP was designed to run in a cloud environment where the system is able to scale the amount of computational resources to fit the difficulty of the problem at hand. FlexGP's primary focus is on the goals stated above; measuring the efficiency of the system relative to the amount of computational resources under utilization is not the focus of this thesis.

FlexGP uses genetic programming (GP) as its underlying learning algorithm. A form of evolutionary algorithm (EA), GP is a sophisticated technique for search and optimization which is easily adapted to solve traditional machine learning problems of regression, classification and clustering[11]. For addressing these challenges it is particularly effective for GP to use Genetic Programming for Symbolic Regression (GPSR), where solutions are modeled as evaluable trees of operators and terminals representing variables from the training data [21]. Several commercial systems have demonstrated similar applications of GP to machine learning and data mining, notably Eureqa[12] and DataModeler[1].

1.1 FlexGP 1.0

Version one of FlexGP (FlexGP 1.0) represented a giant leap forward from an original prototype for cloud-backed machine learning via GP [13]. FlexGP 1.0 introduced the concept of factorization, whereby data and other select parameters may be randomly subsampled or permuted to provide an unique configuration for a set of parallelized learners. FlexGP 1.0 demonstrated a marked improvement of performance with the addition of the factorization of training data. FlexGP 1.0 was focused on addressing

[1]`http://evolved-analytics.com/?q=datamodeler`

regression challenges; it incorporated a means of fusing regression models to form ensembles which provided a noticeable performance boost[19][6].

As a framework FlexGP 1.0 was designed to support the usage of any learning algorithm. As mentioned earlier, the learner used in this thesis and in previous work on FlexGP is a GP learner implemented with our custom Java library for GP called evogpj. At the time of its inclusion in FlexGP 1.0 the evogpj library performed model evaluation in Java via a simple recursive evaluation procedure. The core algorithm used Silva's operator equalization for controlling the bloating of models[15]. A detailed, rigorous description of the design and performance of FlexGP 1.0 can be found in a previous publication[6].

1.2 FlexGP 2.0

This thesis presents FlexGP 2.0, which introduces several key improvements over FlexGP 1.0. FlexGP 2.0 includes four levels of parallelism which each provide distinct benefits:

Search-Level Parallelism: allows GP to pursue multiple user-specified objectives. This improves the flexibility of evogpj's GP learner and enhances the clarity of expression of machine learning problems.

Evaluation-Level Parallelism: translates model evaluation to batch-processing in C++ and distributes model evaluations across multiple CPUs, resulting in a dramatically faster runtime.

Population-Level Parallelism: distributes the population across multiple GP learners on separate machines. This results in potentially super-linear gain in accuracy and further reducing the runtime required to obtain an accurate solution.

Factor-Level Parallelism: boosts accuracy through factorization of the data and problem parameters. Factor-level parallelism also reduces the overall runtime by providing each GP learner with a smaller training dataset. Factor-level-parallelism is included from FlexGP 1.0; a detailed description and analysis of factorization can be found in a previous publication on FlexGP 1.0 [6].

A more in-depth discussion of each level of parallelism is provided in the subsequent chapters.

1.3 Evaluating FlexGP 2.0

To provide a thorough evaluation of the performance of FlexGP 2.0 we introduce a type of assessment and comparison which is used throughout this thesis.

***Time*:** a comparison of experiments by the elapsed wall-clock time since FlexGP was initiated. The x axis displays the elapsed time since the beginning of the experiment, and the y axis displays either the MSE_{test} or the MAE_{test} of the best model from the most recent generation. Points are shown at regularly spaced intervals in time.

This comparison provides insight into the performance of FlexGP by clearly showing any improvement in accuracy v.s. elapsed time.

1.4 Preview of Results

Figure 1-1 provides a preview comparison of the performance of FlexGP 2.0 v.s. FlexGP 1.0 on the MSD dataset described in Appendix A. With all four levels of parallelism, FlexGP 2.0 outperforms FlexGP 1.0 by a wide margin. The body of this thesis details how this performance gain was achieved.

1.5 Index

Chapter 2 covers search-level parallelism, Chapter 3 describes evaluation-level parallelism and Chapter 4 details population-level parallelism. A comprehensive discussion of the combination of the four levels of parallelism is provided in Chapter 5, as well as

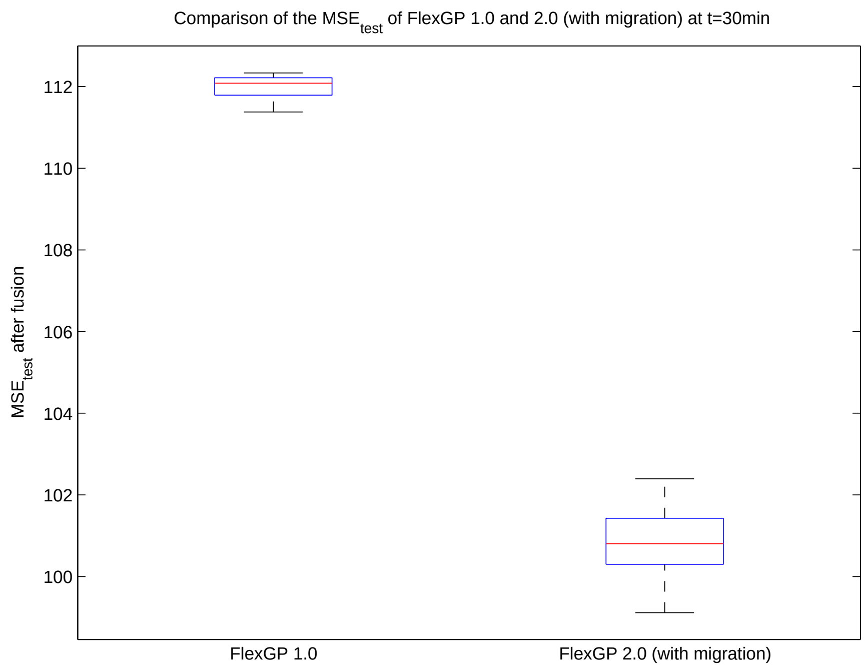


Figure 1-1: A comparison of the performance of FlexGP 2.0 v.s. FlexGP 1.0 after 30 minutes of training. The boxplot represents the distribution of MSE_{test} values obtained after the fusion process. The specifics of the two experiments can be found in Section 5.2.1.

an experimental comparison of FlexGP 2.0 with FlexGP 1.0 and with another prominent machine learning system called *vowpal wabbit*. Chapter 6 presents thoughts on future work and Chapter 7 summarizes the findings of this thesis.

Appendix A presents the details of the problem addressed in this thesis' experiments and explains how the data was partitioned. Appendix B describes in detail the parameters of each experiment and how the results were gathered and analyzed. Appendix C describes the fusion process used in some experiments to boost performance by combining predictions from multiple regression models. Appendix D provides the specific parameters and design notes regarding the configuration of evogpj's GP learner in the experiments. Finally, Appendix E includes information about the cloud platform on which the experiments were conducted.

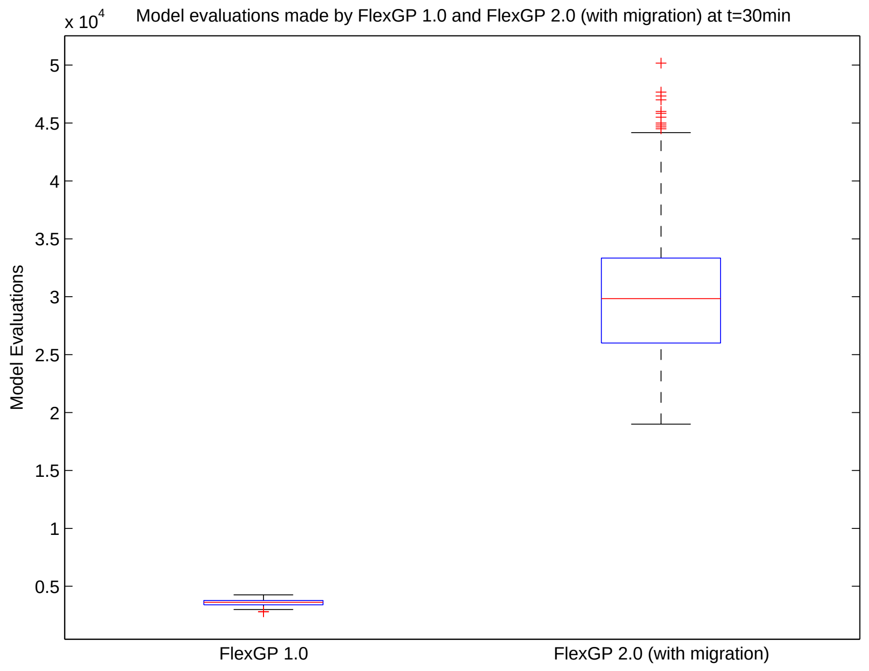


Figure 1-2: A comparison of the model evaluations made by FlexGP 2.0 v.s. FlexGP 1.0 after 30 minutes of training. The boxplot represents the distribution of the number of model evaluations made by each learner. The specifics of the two experiments can be found in Section 5.2.1.

Chapter 2

Search-Level Parallelism

Search-level parallelism allows the simultaneous pursuit of multiple user-specified search objectives. This form of parallelism represents an improvement in the flexibility of FlexGP, and allows a more clear expression for many optimization challenges, including classification and regression.

2.1 Motivation and Goals

Most practical search and optimization problems require a balance between multiple objectives. This is particularly true for problems in the domain of machine learning. Regression problems seek solutions with both low predictive error and low model complexity. Classification problems seek solutions with low rates of both false positives and false negatives in addition to low model complexity.

Traditional optimization techniques which address these problems develop a means of converting multiple objectives to a single objective. One example is the inclusion of model complexity in the fitness score of a candidate solution. The efficacy of such objective-dimensionality reduction metrics varies with the type of problem being solved. More dangerously, objective-dimensionality reduction methods which are *a priori* custom-tailored for a particular problem or domain could introduce bias [20].

However, it is possible to design an algorithm which can reduce the potential for bias while pursuing a variety of objectives simultaneously. This form of optimization process is known as multi-objective optimization. A multi-objective optimization algorithm directly considers multiple objectives and thus eliminates the need for an objective-dimensionality reduction metric.

Unlike single-objective optimization challenges, multi-objective optimization problems may have a range of solutions which are considered optimal. The multi-objective optimization search algorithm maintains and updates a set of optimal solutions rather than focusing on obtaining one best solution. The set of optimal solutions is known as the Pareto-optimal front, or Pareto front. An example of a Pareto front is shown in figure 2-1. This can prove advantageous when contrasted with single-objective optimization techniques which typically produce one candidate solution after a search is complete. For example, multi-objective optimization applied to a regression problem would return a set of models ranging from higher error and lower complexity to lower error and higher complexity. This provides a method to produce models which yield desirable accuracy but are simpler than those obtained from a single-objective optimization [5].

The inclusion of model complexity as an objective raises the possibility of using multi-objective optimization to gauge variable importance. If a variable appears in less complex solutions which still yield high accuracy, the variable is likely more important than others. Similarly, this approach could be used to perform feature selection. If a subtree appears frequently in trees which obtain high accuracy, that subtree is a good candidate for feature extraction[17].

Multi-objective optimization allows the search process to pursue multiple optima at once. Each member of the Pareto front represents a unique optimal solution to the multi-objective optimization problem. Each member can only be replaced if another model dominates it in fitness-space. Section 2.2.2 provides a discussion of dominance and the non-dominated sorting algorithm.

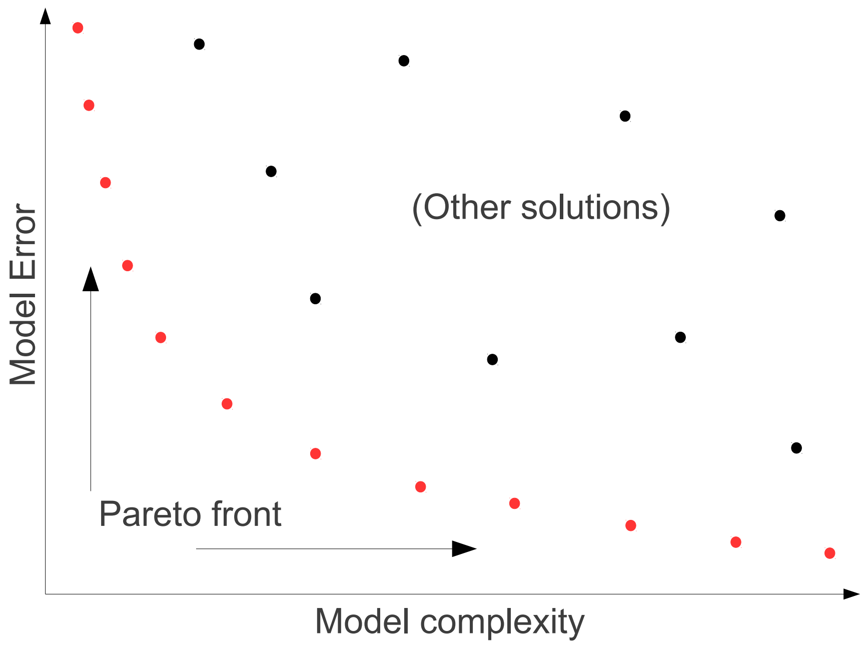


Figure 2-1: An example of a Pareto front. The red dots represent models which are members of the Pareto front, and the black dots represent other models under consideration. The two objectives under minimization are shown on the two axes.

2.1.1 Multi-Objective Optimization v.s. Operator Equalization

FlexGP 1.0 used Silva's operator equalization technique to minimize the bloating of models [14] [6]. FlexGP 2.0 uses multi-objective optimization to achieve a similar goal by including minimization of model complexity as an additional objective.

Multi-objective optimization provides several features which operator equalization does not. Using multi-objective optimization reduces the risk of bias by avoiding the need for GP to rely on a method of objective-dimensionality reduction. Multi-objective optimization maintains a set of optimal solutions rather than a single model.

Multi-objective optimization provides another advantage over operator equalization: a fixed number of model evaluations made per generation. The *NSGA-II* multi-objective optimization algorithm, discussed in Section 2.2.2, requires n model evaluations per generation. Operator equalization will always make at least n model evaluations per generation but may require many multiples more to satisfy the equalizer. As shown in Section 2.3.2 we found this to be particularly prevalent in the first generation. A fixed number of model evaluations per generation means the progress and time consumption of the algorithm are more predictable.

FlexGP 1.0's operator equalization algorithm placed a constraint on the independence of model evaluation. An equalizer was used to maintain a distribution over model complexity. The equalizer would either accept a model into the new population or discard the model and move on to the next. The equalizer could only consider a single model at once, which meant new offspring must be generated, evaluated and considered by the equalizer one at a time. This constraint presented a complication in the parallelization of model evaluation.[1]

With the introduction of multi-objective optimization as discussed in Section 2, no equalizer is required to maintain a distribution over model complexity, removing the sequential evaluation constraint imposed by operator equalization's equalizer. Further, the number of offspring to be bred per generation is fixed, which simplifies parallelization and enables the use of techniques which rely on batch processing.

2.1.2 Advantages of Multi-Objective Optimization

In summary, the important advantages of multi-objective optimization are as follows:

Multiple objective functions: multi-objective optimization allows the inclusion of any number of search objectives, which lends itself to increased flexibility.

Multiple optima: multi-objective optimization maintains a Pareto front consisting of multiple solutions which are deemed as optimal.

Diversity of Complexity: if model complexity is included as an objective, the Pareto front will represent models with a range of complexities. This demonstrates a potential for multi-objective optimization to be an effective vehicle for gauging variable performance and for performing feature selection.

Batch Model Evaluation: multi-objective optimization removes the sequential constraint imposed in FlexGP by operator equalization's equalizer, enabling the

[1]It is certainly possible to design an equalizer which parallelizes more naturally; that is not the focus of this thesis.

optimization of model evaluation through techniques like the C++ model evaluation discussed in Chapter 3.

2.1.3 Goals of Search-Level Parallelism

The goals of search-level parallelism are to accomplish the following:

1. Enable the simultaneous pursuit of multiple objectives.
2. Yield a spectrum of optimal solutions rather than a single solution.
3. Improve the predictability of FlexGP by fixing the number of model evaluations made per generation.
4. Show that multi-objective optimization provides performance comparable to that of operator equalization.

2.2 Integration of Search-Level Parallelism

Multi-objective optimization and operator equalization require different implementations of each component of the evolutionary loop. This section will describe the implementation of multi-objective optimization used in FlexGP's core library, evogpj.

2.2.1 Objective Functions and Evaluation

The evogpj library features a simple model fitness function class hierarchy. Any class which seeks to become an objective function by inheriting from `fitness.FitnessFunction` must implement an `evalPop` method which assigns a fitness score to each model in the input population. In principle any type of model evaluation can be included.

The following steps must be taken to define a custom objective function in evogpj:

1. Add an entry to the `algorithm.Parameters` class' `Operators` static class with the new objective function's name.

2. Define a new class which subclasses the abstract class `fitness.FitnessFunction`, and which is defined in the `fitness` package.

3. Define the `FITNESS_KEY` field to point to the new objective function's name in `algorithm.Parameters.Operators`.

4. Implement the `isMaximizingFunction` method inside the new class, which returns `true` if this objective should be maximized and `false` if it should be minimized.

5. Implement the `evalPop(pop)` method inside the new class, which takes a population as input and assigns a fitness score to each individual.

6. Add support for the new objective function to the `algorithm.AlgorithmBase` class' `create_operators` method by calling the new objective function's constructor and adding it to the linked hash map of fitness functions called `fitnessFunction`. The `create_operators` method initializes the fitness functions and operators used in evolution.

7. Add the new objective function's name to the comma-separated list of objective function names under the parameter `fitness_op` in an evogpj properties file which defines the problem at hand. Multi-objective optimization will be used if more than one name is specified; otherwise operator equalization will be used.

This thesis focused on addressing regression challenges and therefore utilized two fitness functions: a symbolic regression fitness function which seeks to minimize error on a dataset, and a complexity minimizer.

The symbolic regression fitness function assigns higher fitness to models with lower predictive error on the specified data. The mean squared error is calculated using the l^2 norm after scaling the output variable to be between 0 and 1. The scaling allows GP to focus on replicating the correct form of equation rather than the specific coefficients [21].

The complexity minimizer uses a subtree counting function to assess the complexity of a model. Keijzer & Foster introduce this as "visitation length", remarking the

Algorithm 1 SubtreeComplexity(*tree*)

```
tree: a tree with unfixed arity
complexity ← 1
for subtree in CHILDREN(tree) do
    complexity ← complexity + SubtreeComplexity(subtree)
return complexity
```

method directly measures the degree of balance or skew in a tree [10] [21]. We refer to this method as "subtree complexity." The calculation of the subtree complexity of a tree as described in Algorithm 1 is a simple recursive sum of the number of nodes in every subtree of a tree.

Example Objective Function Definition: Subtree Complexity

We provide an example of defining an objective function in FlexGP 2.0 by considering the definition of subtree complexity. The steps for doing so as follows:

Step 1: Add a line to the `Operators` static class in `algorithm.Parameters`:

```
public static final String SUBTREE_COMPLEXITY_FITNESS =
                                    "fitness.SubtreeComplexity";
```

Steps 2 - 5: The `fitness.SubtreeComplexityFitness` class appears in its entirety as follows:

```
package fitness;


import algorithm.Parameters;
import genotype.Tree;
import gp.Individual;
import gp.Population;


/**
 * Evaluates an individual's subtree complexity
 * @author Dylan Sherry
 */
public class SubtreeComplexityFitness extends FitnessFunction {
        public static final String FITNESS_KEY =
            Parameters.Operators.SUBTREE_COMPLEXITY_FITNESS;

        public Boolean isMaximizingFunction() {
                return false;
        }

        @Override
        public void eval(Individual ind) {
                Tree t = (Tree) ind.getGenotype();
                Integer complexity = t.getSubtreeComplexity();
                ind.setFitness(SubtreeComplexityFitness.FITNESS_KEY,
                                                  (double) complexity);
        }

        @Override
        public void evalPop(Population pop) {
                for (Individual individual : pop) {
                        this.eval(individual);
                }
        }
}
```

This example includes the definition of an `eval` method which operates on sole models. This is an optional; only the `evalPop` method is invoked elsewhere in evogpj. The `getSubtreeComplexity` method is defined elsewhere in evogpj, and provides an implementation of Algorithm 1.

Step 6: Import the new `fitness.SubtreeComplexity` class in the `algorithm.AlgorithmBase` class. Include a section in the `algorithm.AlgorithmBase` class' `create_operators` method similar to:

```
if (fitnessOperatorName.equals(
        Parameters.Operators.SUBTREE_COMPLEXITY_FITNESS)) {
        fitnessFunctions.put(fitnessOperatorName,
                new SubtreeComplexityFitness());
```

Refer to the `algorithm.AlgorithmBase` class' `create_operators` method for the full context and to see other examples.

Step 7: The fitness functions specified for this problem were: `fitness_op = fitness.SRFitness.ExternalData,fitness.SubtreeComplexityFitness` where `fitness.SRFitness.ExternalData` references the C++ model evaluation objective discussed further in Chapter 3.

2.2.2 Learning Algorithm Modifications

We implemented *NSGA-II*, an elitist multi-objective optimization algorithm which includes the previous generation's population when selecting the next generation. *NSGA-II* uses the non-dominated sorting algorithm to select the next generation's population, and uses a crowded tournament selection operator to choose which models to breed during the generation of children [5].

Algorithm 2 provides a formal description of our implementation. The key parameters of the algorithm are the population size N, the list of fitness functions f, a method B by which to select the best model and a stopping criterion $STOP$.

The specification of objective functions is detailed in Section 2.2.1. The options for the method B for selecting the best model are described in this section. The stopping criterion may be a desired number of generations to run, a desired maximum elapsed runtime or a desired fitness threshold at which to stop if a model is found which exceeds the threshold.

Initialization

- First an initial population of size N is generated using Koza's ramped-half-and-half [11].
- Each model in the initial population is then evaluated by all fitness functions in f and assigned a *non-domination rank* and *crowding distance*.
- The initial population is sorted so that the best models are at the beginning of the population.
- A single best model is chosen to be reported as the primary model obtained from the initial population using selection method B. The model is scaled to fit the output variable via linear regression and is exported to the user.

Each Generation

While the stopping criterion is not met, the algorithm is allowed to advance another generation. During each generation:

- A set of N children are generated from the current population, using crowded tournament selection to pick which parents are allowed to reproduce.
- The children are evaluated by each fitness function and combined with the previous generation's population to form a population of size $2N$. The mixing of parents and children is a form of archiving.
- The models in the $2N$ population are each given a *non-domination rank* and a *crowding distance*, and the population is sorted.
- The next generation's population is then obtained by keeping the top N of the $2N$ models.
- A single best model is chosen to be reported as the primary model obtained from the current generation using selection method B. The model is scaled to fit the output variable via linear regression and is exported to the user.

Algorithm 2 *NSGA-II* MULTI-OBJECTIVE OPTIMIZATION (N, f, B, $STOP$)

```
$N$: size of population, $f$: list of $M$ fitness functions
$B$: a method by which to select the best model, $STOP$: a stopping criterion
$pop \leftarrow$ INITIALIZE(n)
for $function$ in $f$ do
    EVAL($pop$,$function$)
CalculateNonDominationCount($pop$)
CalculateCrowdingDistances($pop$)
Sort($pop$)
$best \leftarrow$ SelectBest($pop$,$B$)
$scaledBest \leftarrow$ ScaleModel($best$)
while not $STOP$ do
    $children \leftarrow$ BreedChildren($pop$)
    for $function$ in $f$ do
        EVAL($children$,$function$)
    $total \leftarrow pop + children$
    CalculateNonDominationCount($total$)
    CalculateCrowdingDistances($total$)
    Sort($total$)
    $pop \leftarrow total[0, N-1]$
    $best \leftarrow$ SelectBest($pop$,$B$)
    $scaledBest \leftarrow$ ScaleModel($best$)
```

FlexGP scales the output variable to span the range $[0, 1]$ so GP can focus on learning the shape of the relation described in the data [20], as discussed in Appendix D. Models are represented internally in their unscaled form for learning. To obtain an external representation of a model FlexGP performs simple linear regression to find a best fit between the models' scaled predictions and the original unscaled output variable. FlexGP provides fast model scaling which is compatible with both the Java and C++ model evaluation schemes.

The user can elect to save all models from the Pareto front for each generation. In this case all models are scaled before they are exported from FlexGP.

Non-Dominated Sort

Single-objective optimization algorithms rank models by their performance on the sole objective under consideration. A multi-objective optimization algorithm must consider all objectives when determining a ranking of models. Further, as there is no implicit absolute ordering of models in a multi-objective optimization problem, multiple models may be determined to represent optimal solutions. Multi-objective

Algorithm 3 CalculateNonDominationCount(pop, f)

```
$pop$: a population of size $N$
$f$: a list of $M$ fitness functions
for $a$ in 1 to $N$ do
    $model_A \leftarrow pop[a]$
    for $b$ in 1 to $N$ s.t. $a \neq b$ do
        $I_B \leftarrow pop[b]$
        if Domination($model_A$,$model_B$,$f$) then
            IncrementDominationCount($model_B$)
        else if Domination($model_B$,$model_A$,$f$) then
            IncrementDominationCount($model_A$)
        else if Identical($model_A$,$model_B$) then
            if $a < b$ then
                IncrementDominationCount($model_B$)
            else
                IncrementDominationCount($model_A$)
```

optimization requires a substantially more complex method of sorting to identify the set of optimal models.

The non-dominated sorting algorithm is the heart of multi-objective optimization. Non-dominated sorting identifies the set of optimal models (which is referred to as the Pareto front) and subsequent fronts by calculating the domination count of each model, as shown in Algorithm 3. The non-dominated sorting algorithm specified here is not, strictly speaking, a sorting algorithm, but rather calculates the non-domination count of each model to be used in the model sorting and selection described later in this section. Our non-dominated sorting algorithm is an adaptation of Deb's $O(MN^2)$ algorithm [5]. Our algorithm removes the additional step of calculating each model's front number, as the non-domination count represents the minimum information needed for ranking a population.

The calculation relies on a dominance relation between two models. As shown in Algorithm 4, the dominance relation stipulates model $model_A$ dominates $model_B$ if and only if $model_A$ performs equivalent to or better than $model_B$ for all objectives, and strictly better than $model_B$ for at least one objective. The references to GetFitness($model$,$function$) access the memoized fitness computed prior to the non-dominated sorting.

Algorithm 4 DOMINATION($model_A$, $model_B$, f)

```
$model_A$ and $model_B$: models to compare, $f$: a list of $M$ fitness functions
for $function$ in $f$ do
    $fitness_A \leftarrow$ GETFITNESS($model_A$,$function$)
    $fitness_A \leftarrow$ GETFITNESS($model_A$,$function$)
    if $fitness_A > fitness_B$ then
        return $false$;
```

Crowded Tournament Selection

Tournament selection methods are those used to select a parent or parents from a population in order to breed children to be considered for inclusion in the next generation's population [7]. A simple implementation of a binary tournament selection procedure is as follows, parameterized by the tournament size:

1. Randomly select a model from the population to be the initial winner of the tournament.

2. Randomly select a challenger from the remaining population. If the challenger's fitness is better than the current tournament winner's fitness, the challenger is now the winner of the tournament.

3. Repeat step 2 until the number of challengers which have been considered is equal to the tournament size parameter, after which return the winning model.

Traditional tournament selection compare fitness values obtained from a single objective. We must adopt an adaptation of tournament selection which can handle multiple objectives.

A simple tournament selection method which can function in multi-objective optimization can be designed by using the non-domination count previously calculated by non-dominated sorting. In step 2 of the above single-objective tournament selection process we compared fitnesses to determine the winner. We introduce the following modification to step 2: first compare the models' non-domination rank. If the non-domination ranks are different, select the model with the more favorable rank. Otherwise if the non-domination ranks are equivalent, the non-dominated sort has deemed the two models as equivalently optimal, so we select the best model at random.

There is an important nuance to consider. Selection naturally reduces diversity of models being considered, relying on variation, the subsequent step in an evolutionary algorithm, to increase the diversity. In multi-objective optimization, when the models which compose the Pareto front are densely packed in one region of the fitness space, it is likely the models are also closely situated in the search space. However, to fully benefit from search-level parallelism the search process should consider a set of areas of the search space which are as diverse as possible. For this reason it is desirable to maintain a Pareto front whose models are as evenly spaced as possible, thus maximizing the diversity of activity in the search space.

Crowded tournament selection provides a method to favor models in a population which are the most distant or isolated and therefore represent the most diverse and desirable search points to pursue. To use the crowded tournament selection operator during the selection step of GP, each model is first assigned a crowding distance which indicates the density of population of the region surrounding that member. The crowded tournament selection then favors models with a better non-domination count and a better crowding distance [5].

Algorithm 5 describes the calculation of crowding distances. The underlying principle of the crowding distance calculation is that a model's crowding distance is equal to the product of the distances between its neighbors along each of the M objectives under consideration. This describes the volume of the largest possible M-dimensional hypercube in fitness-space which touches at least one neighbor per face, where M indicates the number of objectives under consideration. A larger crowding distance indicates a greater distance between the model in question and its neighbors, and is therefore more desirable.

The subroutine SortByFitnessFunction(*pop*,$function$) takes as input a population of models and an objective function, and returns the indices of the population sorted from best to worst along the objective indicated by the objective function. No model evaluations are made in the crowding distance algorithm; the memoized fitness

Algorithm 5 CALCULATECROWDINGDISTANCE(*pop*, f)

```
$pop$: the population of size $N$ on which to operate, $f$: a list of $M$ fitness functions
for $i$ in $[0, 1, ..., (N-1)]$ do
    $model.crowdingDistance \leftarrow 0$
for $function$ in $f$ do
    $sortedIndices \leftarrow$ SORTBYFITNESSFUNCTION($pop$,$function$)
    for $i$ in $[0, 1, ..., (N-1)]$ do
        $index \leftarrow sortedIndices[i]$
        $model \leftarrow pop[index]$
        if $(index = 0)$ || $(index = N-1)$ then
            $distance_{function} \leftarrow MAX\_DIST$
        else
            $prevModel \leftarrow pop[index-1]$
            $nextModel \leftarrow pop[index+1]$
            $distance_{function} \leftarrow \|$GETFITNESS($nextModel$,$function$)$-$
                                        GETFITNESS($prevModel$,$function$)$\|$
        $model.crowdingDistance \leftarrow model.crowdingDistance + distance_{function}$
```

values are accessed by the procedure GETFITNESS(*model*,*function*).

Once the crowding distances have been calculated for each model in the population, the crowded tournament selection method is as follows:

1. Randomly select a model from the population to be the initial winner of the tournament.

2. Randomly select a challenger from the remaining population. If the challenger's non-domination rank is better (smaller) than the current tournament winner's non-domination rank, the challenger is now the winner of the tournament. Else if the non-domination ranks are equivalent, the model with the better (larger) crowding distance is the new tournament winner.

3. Repeat step 2 until the number of challengers which have been considered is equal to the tournament size parameter, after which return the winning model.

Model Selection Criterion and Population Sorting

After each model has been assigned a non-domination rank and a crowding distance, the combined population of size $2N$ is sorted in order to choose the best N models to become the next generation's population. The best model is found at the top of the population after sorting. Sorting the population also makes it easy to identify the best b models to send for migration, which will occupy the top b positions in the population after sorting.

Once the previous sort has been performed FlexGP must choose a model to record as the best of that generation. In the single-objective case the best model is simply the model with the best fitness; multi-objective optimization yields multiple optimal solutions which means a new strategy is needed. Multiple migrants must be selected if FlexGP has been asked to perform migration, so FlexGP must also provide a means of selecting a variable number of best models from the population.

FlexGP includes the option of recording the entire Pareto front if running multi-objective optimization, but a best model must still be identified.

FlexGP handles this by sorting the population according to a user-specified strategy. The three strategies are defined below.

A. Original Sort

The simplest solution is the *original sort* method, which preserves the sort originally made as described above, where models are sorted by non-domination count and, within each front, by crowding distance.

Preserving an ordering by non-domination rank is desirable for this selection, since multi-objective optimization deems the Pareto front and subsequent fronts to contain the most optimal models. However, secondary sort by crowding distance may not provide the best outcome. Crowding distance favors models who are the most distant and therefore in conjunction represent the most diverse solutions. An alternative selection metric would address the tradeoff between objectives naturally made in multi-objective optimization.

B. Best Fitness

The *best fitness* selection method sorts models within each front by fitness rather than crowding distance. This selection method ignores any objectives other than the one which was listed first and therefore identified as the primary objective for

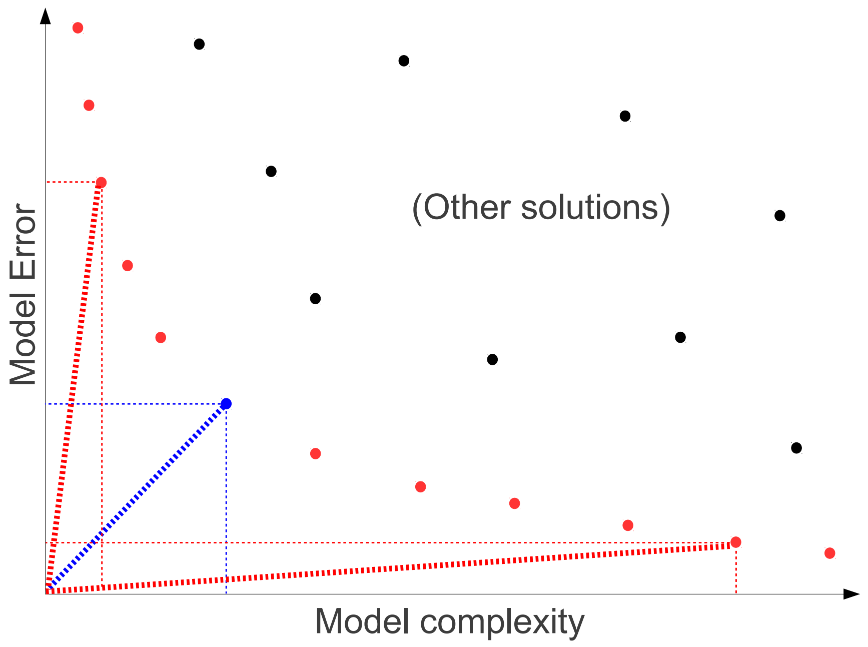


Figure 2-2: An example of using the euclidean distance to find the knee of a Pareto front. The red dots represent models which are members of the Pareto front, and the black dots represent other models under consideration. The two objectives under minimization are shown on the two axes. The euclidean distance of several models is shown by the diagonal dotted lines, where the blue dot shows the model with the lowest euclidean distance (the "knee").

best fitness selection. To select the best model the *best fitness* method picks the model from the Pareto front which has the highest fitness. The best model from the Pareto front must be the best in the population along the first objective because it represents one extreme of the Pareto front. In the case of the two-objective regression problem discussed in this thesis, this corresponds to selecting the model with the lowest MSE_{train} in the population.

C. Euclidean Distance

FlexGP provides a third option for a selection criterion called *euclidean distance*. The euclidean distance selection metric sorts models primarily by non-domination rank, as in the original sort method, and uses the model's distance from the origin in fitness-space as the secondary sort. Euclidean distance favors models which are near the "knee" of the front, as depicted in figure 2-2 [3].

Algorithm 6 CALCULATEEUCLIDEANDISTANCE(*pop*, f)

```
$pop$: the population of size $N$ on which to operate, $f$: a list of $M$ fitness functions
for $i$ in $[0, 1, ..., (N-1)]$ do
    $model.euclideanDistance \leftarrow 0$
for $function$ in $f$ do
    $(min_{function}, max_{function}) \leftarrow$ FINDMINANDMAX($pop$,$f$)
    $range_{function} \leftarrow max_{function} - min_{function}$
    for $i$ in $[0, 1, ..., (N-1)]$ do
        $model \leftarrow pop[index]$
        $fitnessScore_{function} \leftarrow$ GETFITNESS($model$,$function$)
        $distance_{function} \leftarrow (\frac{fitnessScore_{function} - min_{function}}{range_{function}})^2$
        $model.euclideanDistance \leftarrow model.euclideanDistance + distance_{function}$
```

Algorithm 6 describes the calculation of the euclidean distance of each model within a population. The first step is to identify the minimum and maximum values of the population's models for each objective. Then a model's euclidean distance is given by the sum of the squares of the models' distances from the origin for each objective, after normalization to place each distance between 0 and 1 according to the objective's minimum and maximum values within the population.

2.3 Experimental Evaluation

This section presents experiments conducted to determine if FlexGP 2.0's search-level parallelism has met the goals outlined in Section 2.1.3.

2.3.1 Experimental Setup

Four experiments were conducted with FlexGP operating with the following configurations:

1. ***OE-SIMP***: Operator equalization with a simple function set

2. ***MO-SIMP***: Multi-objective optimization with a simple function set

3. ***OE-COMP***: Operator equalization with a complex function set

4. ***MO-COMP***: Multi-objective optimization with a complex function set

Simple	$+ - * /$
Complex	$+ - * /$ *exp log sqrt square sin cos*

Table 2.1: The two function sets used in the four experiments described in this section. The log function was defined to return 0 if the input would otherwise produce an undefined output.

Table 2.1 shows the two function sets used in these experiments.

To provide a fair comparison between the four experiments, all were conducted with Java-based model evaluation. C++ model evaluation was not used because evogpj's implementation of operator equalization was not modified to support it. The focus of this thesis at an algorithmic level was on increasing the speed of multi-objective optimization; further, operator equalization imposes constraints on model evaluation which make batch processing unfavorable, as discussed at the end of Section 2.1.1. To compensate for the slow speed of Java-based model evaluation In each experiment FlexGP was allowed to train for 48 hours and all measurements were collected during that interval.

For multi-objective optimization the *best fitness* metric discussed in Section 2.2.2 was used to determine the best model. The *best fitness* metric when applied to multi-objective optimization ignores the second objective, subtree complexity, and selects the model with the best fitness along the primary objective. This means both multi-objective optimization and operator equalization identified their best model per generation as that with the highest fitness and therefore the lowest MSE_{train}.

In each repetition of each experiment the learner given access to a different 70% training data split. No population-level or factorization-level parallelism was used.

Experiments *OE-SIMP* and *MO-SIMP* both used a simple function set, and experiments *OE-COMP* and *MO-COMP* both used a complex function set. This was done to investigate what effect the function set might have on the comparison of multi-objective optimization and operator equalization.

All experiments were conducted on the Million Song Dataset year prediction challenge, which is described in Appendix A. Each experiment was repeated 10 times to demonstrate statistical validity. The salient information collected from each generation of each trial includes the elapsed time, the number of model evaluations made and the MSE_{test} and MAE_{test} of the best model. s

2.3.2 Results

Table 2.2 shows the mean and standard deviation of the number of model evaluations made in the first generation of each experiment.

Table 2.3 shows the mean number of evaluations and the mean and standard deviation of the number of generations per model evaluation. The table also shows the mean and standard deviation of the number of model evaluations made per generation. Therefore a lower mean($\frac{evals}{gens}$) indicates more evaluations were made per generation. This is because operator equalization's requires a variable number of model evaluations each generation to equalize the child population by model size, and is discussed further in Section 2.1.2.

Figure 2-3 shows the mean MSE_{test} and MAE_{test} v.s. time for experiments *OE-SIMP* and *MO-SIMP*. Figures 2-4 show the mean MSE_{test} and MAE_{test} v.s. time for experiments *OE-COMP* and *MO-COMP*. Table 2.4 shows the same results in tabular form. This represents an *Time* comparison as defined in Section 1.3.

figure 2-5 shows the mean MSE_{test} and MAE_{test} v.s. the elapsed number of model evaluations for experiments *OE-SIMP* and *MO-SIMP*. Figures 2-6 show the mean MSE_{test} and MAE_{test} v.s. time for experiments *OE-COMP* and *MO-COMP*. Table 2.5 shows the same results in tabular form. This represents an *Model Evaluations* comparison as defined in Section 1.3.

Experiment	mean($evals$)	stddev($evals$)
OE-SIMP	3223.3	246
MO-SIMP	1000	0.0
OE-COMP	73175.6	36665.9
MO-COMP	1000	0.0

Table 2.2: The average and the standard deviation of the number of model evaluations made during the first generation for 10 trials of the four experiments described in Section 2.3.1.

Experiment	mean($gens$)	stddev($gens$)	mean($\frac{evals}{gens}$)	stddev($\frac{evals}{gens}$)
OE-SIMP	36.6000	7.6768	3725.2	526.8
MO-SIMP	145.9000	30.1347	1000	0
OE-COMP	29.1429	8.9336	5937.3	2517.3
MO-COMP	138.4444	32.9777	1000	0

Table 2.3: Generation statistics are shown for the four experiments from Section 2.3.1. The 2nd and 3rd columns show the mean and standard deviation of the number of generations elapsed during the model evaluations indicated in the second column. The 4th and 5th column show the mean and standard deviation of the number of generations per model evaluation.

	Metric	8hrs	16hrs	24hrs	32hrs	40hrs	48hrs
OE-SIMP	mean(MSE_{test})	117.456	116.420	115.249	114.753	114.506	113.876
	stddev(MSE_{test})	3.216	3.120	3.610	3.748	3.811	4.025
	mean(MAE_{test})	8.095	8.050	7.992	7.969	7.959	7.931
	stddev(MAE_{test})	0.092	0.103	0.156	0.166	0.172	0.173
MO-SIMP	mean(MSE_{test})	117.046	115.113	114.487	113.835	113.515	113.375
	stddev(MSE_{test})	3.615	4.145	4.556	4.762	4.901	4.984
	mean(MAE_{test})	8.087	7.989	7.961	7.940	7.927	7.922
	stddev(MAE_{test})	0.129	0.217	0.221	0.230	0.238	0.239
OE-COMP	mean(MSE_{test})	118.107	117.186	116.079	113.600	112.316	110.740
	stddev(MSE_{test})	2.232	3.427	4.936	5.128	5.530	5.497
	mean(MAE_{test})	8.168	8.100	8.036	7.896	7.814	7.733
	stddev(MAE_{test})	0.104	0.175	0.265	0.248	0.291	0.309
MO-COMP	mean(MSE_{test})	117.237	114.592	113.609	112.782	112.045	111.561
	stddev(MSE_{test})	3.690	6.168	6.840	7.357	7.614	7.851
	mean(MAE_{test})	8.101	7.968	7.901	7.864	7.824	7.806
	stddev(MAE_{test})	0.130	0.266	0.304	0.337	0.352	0.359

Table 2.4: The average and the standard deviation of the MSE_{test} and MAE_{test} v.s. the elapsed time (in hours) for 10 trials of the four experiments described in Section 2.3.1.

2.3.3 Analysis

The results show multi-objective optimization gives a slight benefit in performance relative to operator equalization when the simple function set is used, as shown in Figure 2-3. For experiments *OE-COMP* and *MO-COMP* which were conducted with the complex function set multi-objective optimization outperforms operator equalization for most of the time, but operator equalization eventually begins to yield higher accuracy than multi-objective optimization, as shown in Figure 2-4.

The model evaluation plots from figure 2-5 and figure 2-6 show no dramatic differ-

	Metric	3000	6000	9000	12000	15000	18000
OE-SIMP	mean(MSE_{test})	117.362	116.164	115.004	114.501	113.876	113.876
	stddev(MSE_{test})	3.309	3.163	3.741	3.842	4.025	4.025
	mean(MAE_{test})	8.091	8.036	7.978	7.958	7.931	7.931
	stddev(MAE_{test})	0.095	0.107	0.162	0.173	0.173	0.173
MO-SIMP	mean(MSE_{test})	116.404	114.986	114.278	113.673	113.387	113.375
	stddev(MSE_{test})	3.656	4.126	4.730	4.861	4.991	4.984
	mean(MAE_{test})	8.054	7.977	7.953	7.933	7.922	7.922
	stddev(MAE_{test})	0.162	0.215	0.232	0.239	0.239	0.239
OE-COMP	mean(MSE_{test})	118.301	117.011	116.054	113.473	112.144	110.911
	stddev(MSE_{test})	2.031	4.104	4.988	6.102	6.235	5.718
	mean(MAE_{test})	8.189	8.083	8.036	7.902	7.807	7.741
	stddev(MAE_{test})	0.100	0.224	0.265	0.331	0.321	0.321
MO-COMP	mean(MSE_{test})	116.600	114.148	113.096	112.403	111.806	111.561
	stddev(MSE_{test})	3.831	5.964	6.800	7.180	7.649	7.851
	mean(MAE_{test})	8.066	7.935	7.877	7.847	7.815	7.806
	stddev(MAE_{test})	0.173	0.253	0.310	0.333	0.349	0.359

Table 2.5: The average and the standard deviation of the MSE_{test} and MAE_{test} v.s. the elapsed number of model evaluations for 10 trials of the four experiments described in Section 2.3.1.

ence to the time-series plots, which suggests operator equalization and multi-objective optimization were roughly matched in terms of the overall number of fitness evaluations made in 48 hours. This is corroborated by Table 2.3.

The final MSE_{test} was lower for both operator equalization and multi-objective optimization when operating with the complex function set. This indicates a more complex function set provides a better fit on the MSD year recognition challenge.

These results indicate multi-objective optimization will generally yield better performance than operator equalization in the same time. No further statement can be made about the performance of the two algorithms' accuracy with respect to time, as the reported performance likely has a high dependence on the specific characteristics of the data. Yet the fact that multi-objective optimization was able to compare with operator equalization on an experiment of this size and complexity suggests the other advantages of multi-objective optimization over operator equalization indicate multi-objective optimization is generally a better option when time is valued.

Table 2.2 shows an important characteristic. For the simple function set operator equalization consumed on average over three times as many fitness evaluations in the first generation as did multi-objective optimization. For the complex function set operator equalization consumed on average over 73 times as many fitness evaluations in the first generation as did multi-objective optimization. Further, the standard deviation was quite high. This can also be seen in the time and model evaluation plots. The natural conclusion is that operator equalization will always take a long time to set up a distribution over model size, which is undesirable. Multi-objective optimization, on the other hand, will always make 1000 evaluations per generation. Therefore multi-objective optimization is a better choice when results are desired quickly.

These results suggest a strategy for benefiting from both algorithms: run the first few generations with multi-objective optimization, then switch to operator equalization once the population reflects a distribution over model complexity. That way MOO sets up a complexity distribution so operator equalization won't consume tens of thousands of model evaluations in the first generation. Once multi-objective optimization has set up a desirable complexity distribution, operator equalization can take over.

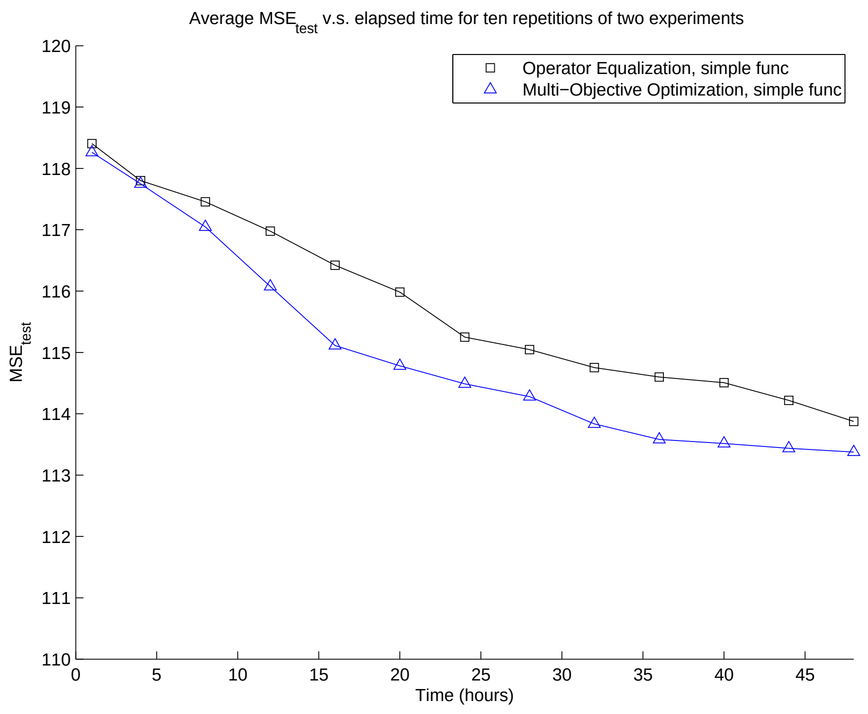
To summarize, the results in this section in combination with the description of the design in the previous section demonstrate multi-objective optimization has met all the goals originally stipulated in Section 2.1.3:

1. Enable the simultaneous pursuit of multiple objectives.

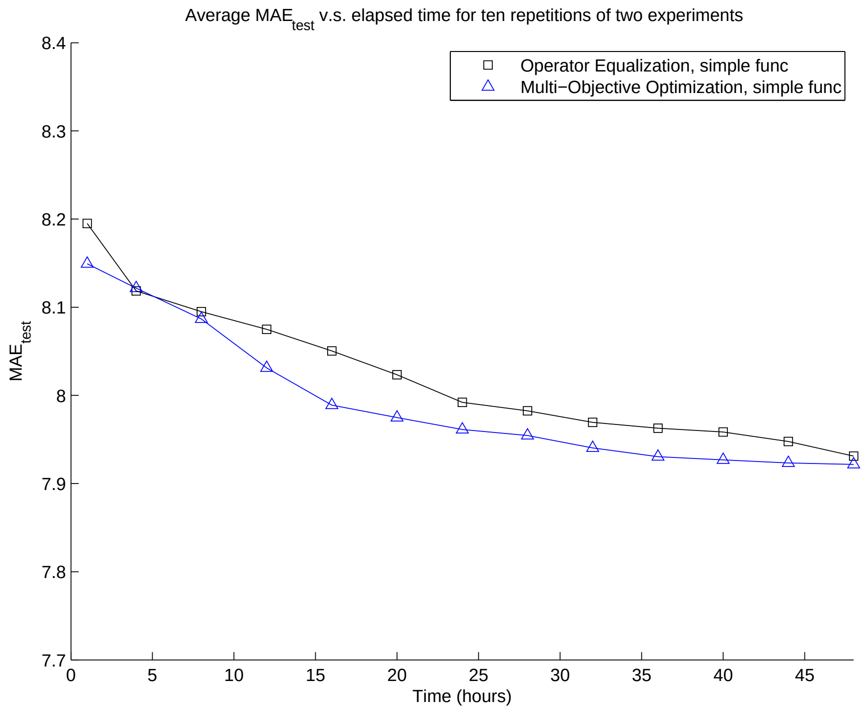
2. Yield a spectrum of optimal solutions rather than a single solution.

3. Improve the predictability of FlexGP by fixing the number of model evaluations made per generation.

4. Show that multi-objective optimization provides performance comparable to that of operator equalization.

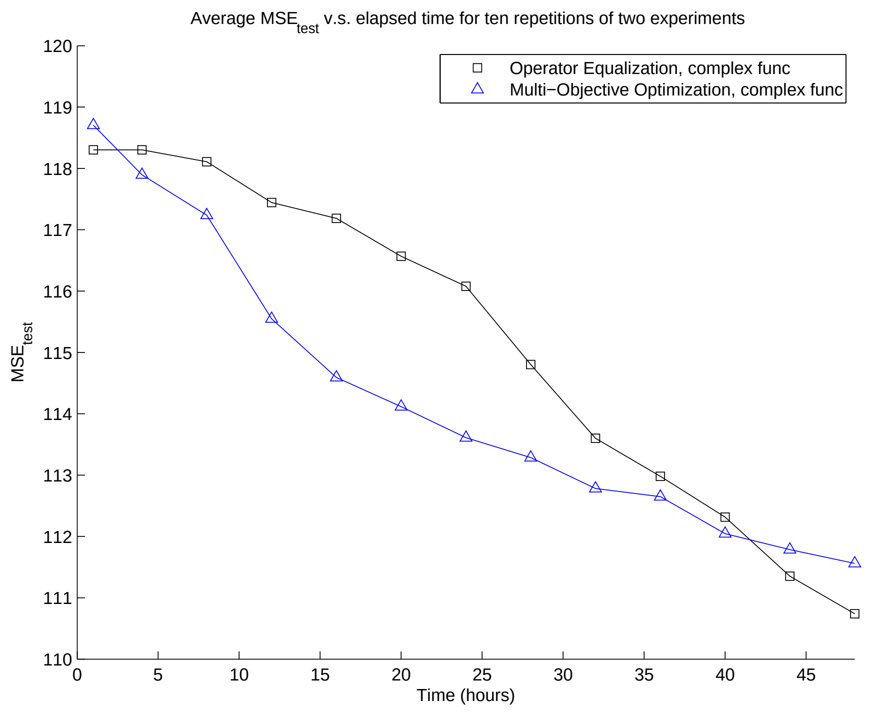


(a) mean(MSE_{test}) v.s. elapsed time for search-parallel experiments *OE-SIMP* and *MO-SIMP*

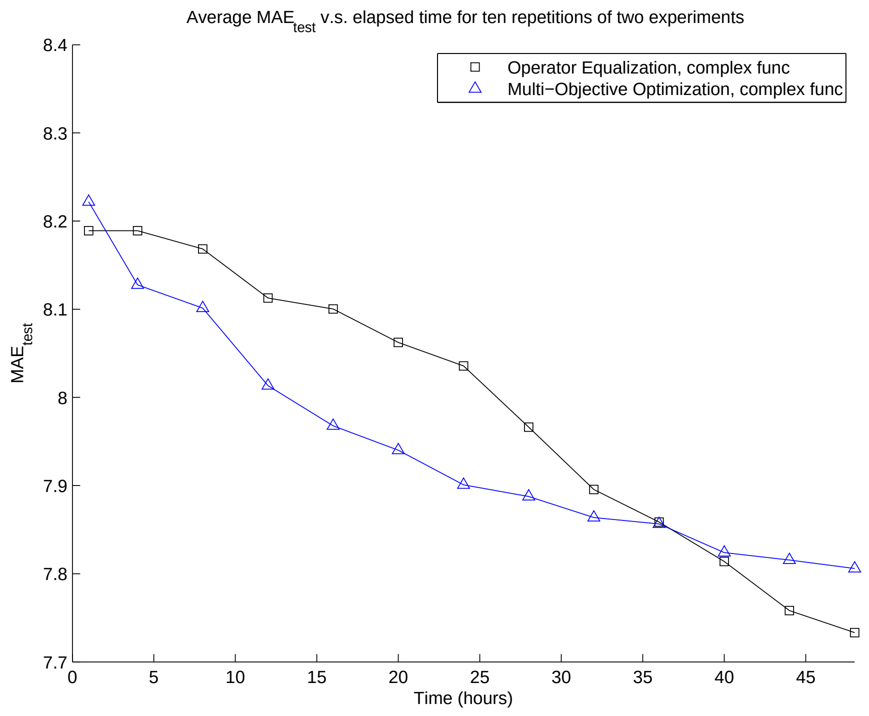


(b) mean(MAE_{test}) v.s. elapsed time for search-parallel experiments *OE-COMP* and *MO-COMP*

Figure 2-3: A comparison of the mean(MSE_{test}) and mean(MAE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-SIMP* and *MO-SIMP* described in Section 2.3.1, which ran with a simple function set.

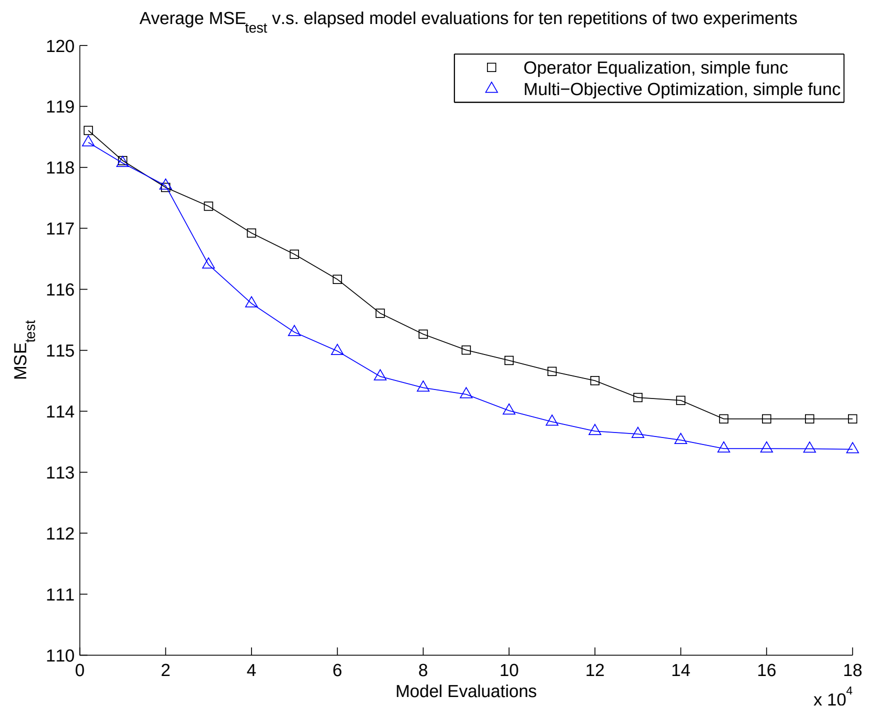


(a) mean(MSE_{test}) v.s. elapsed time for search-parallel experiments *OE-COMP* and *MO-COMP*

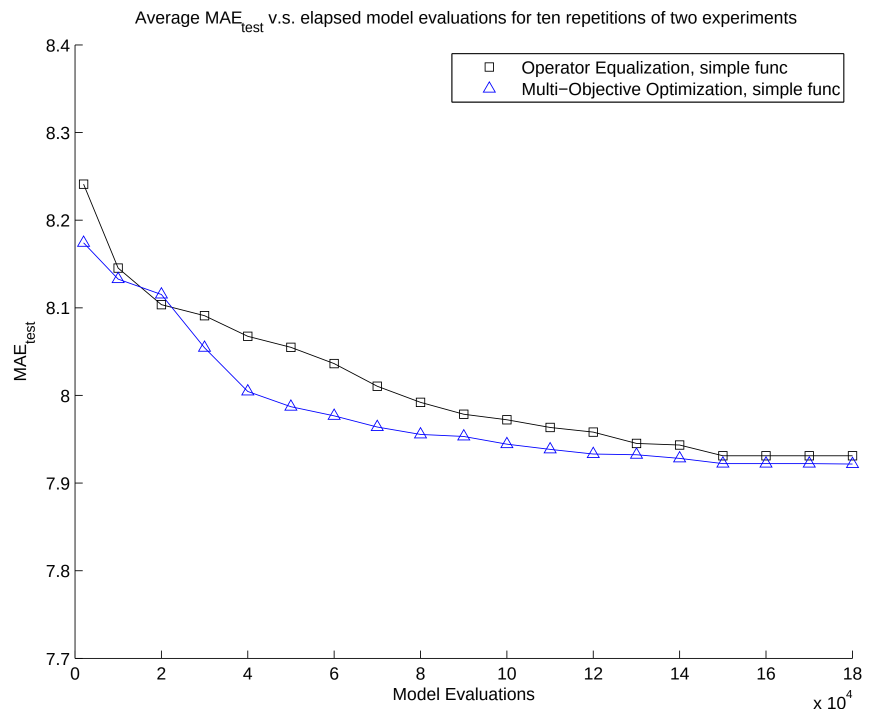


(b) mean(MAE_{test}) v.s. elapsed time for search-parallel experiments *OE-COMP* and *MO-COMP*

Figure 2-4: A comparison of the mean(MSE_{test}) and mean(MAE_{test}) v.s. elapsed time for search-parallel experiments *OE-COMP* and *MO-COMP* described in Section 2.3.1, which ran with a complex function set.

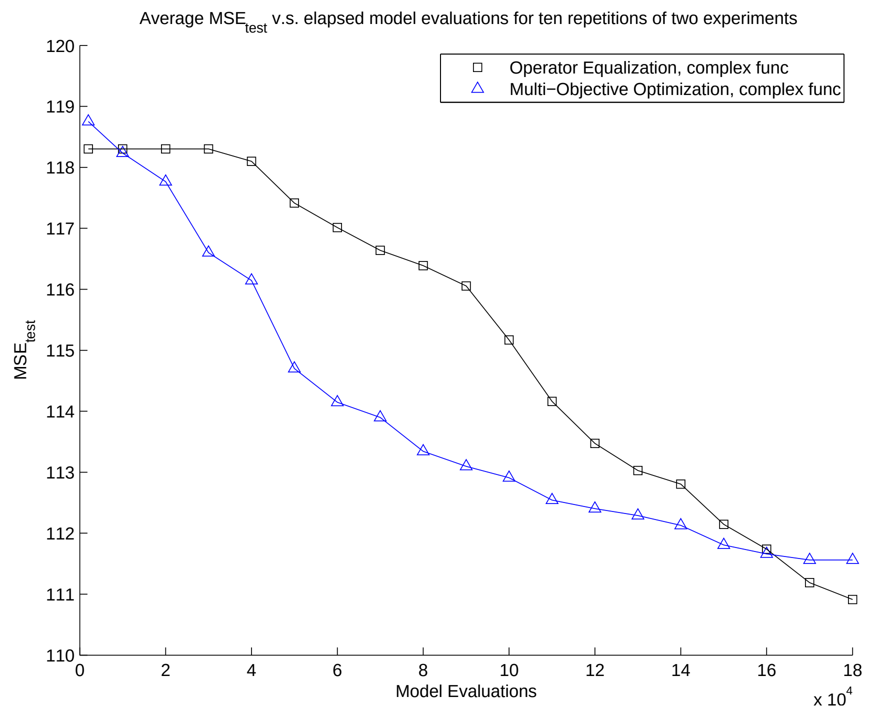


(a) mean(MSE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-SIMP* and *MO-SIMP*

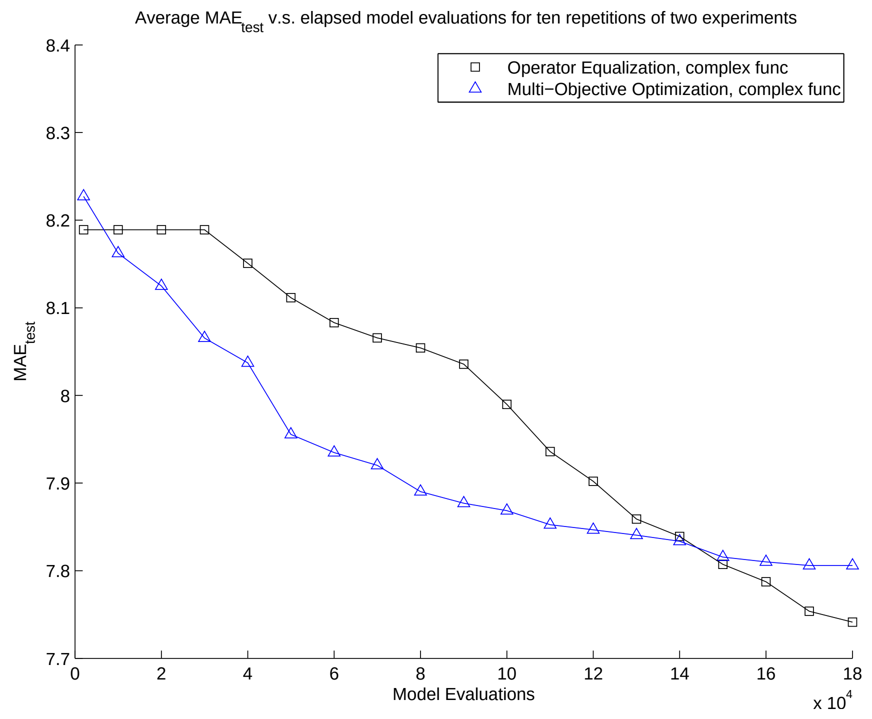


(b) mean(MAE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-SIMP* and *MO-SIMP*

Figure 2-5: A comparison of the mean(MSE_{test}) and mean(MAE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-SIMP* and *MO-SIMP* described in Section 2.3.1, which ran with a simple function set.



(a) mean(MSE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-COMP* and *MO-COMP*



(b) mean(MAE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-COMP* and *MO-COMP*

Figure 2-6: A comparison of the mean(MSE_{test}) and mean(MAE_{test}) v.s. elapsed model evaluations for search-parallel experiments *OE-COMP* and *MO-COMP* described in Section 2.3.1, which ran with a complex function set.

Chapter 3

Evaluation-Level Parallelism

Evaluation-level parallelism distributes the computation of models' fitnesses to multiple CPUs. This distribution dramatically reduces the runtime of FlexGP's core GP learner, allowing learning to occur with data of a size and complexity which was previously intractable.

Chapter 2 saw FlexGP 1.0's Java-based model evaluation make no more than 180k model evaluations on average in 48 hours. The results from this section show the same number of model evaluations made after only 3 hours of computation when evaluation-level parallelism is activated. This represents a speedup of approximately 20x.

3.1 Motivation and Goals

Model evaluation is typically the fundamental bottleneck in GP, and in EAs in general [8]. Figure 3-1 shows a standard evolutionary loop which was used in FlexGP 1.0. Model evaluation represents the most computationally expensive component of the loop.

There are four primary reasons why model evaluation is the most time-consuming step of GP. They are enumerated as follows:

Data Length: A model's predictions must be evaluated for all training cases in

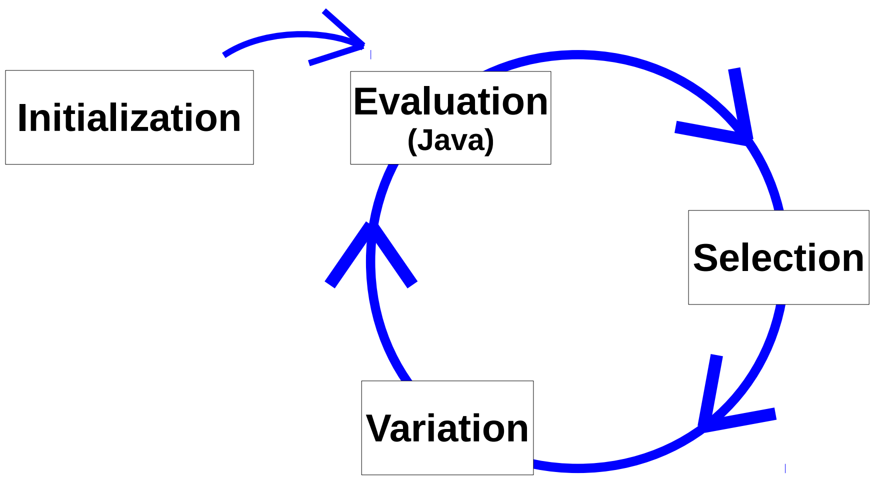


Figure 3-1: This figure visualizes the standard evolutionary loop used by FlexGP 1.0. Each labeled box represents a step in the loop. FlexGP 1.0 performs model evaluation in Java. The blue path shows the flow of each population of models through the algorithm during each generation. In FlexGP the beginning of each generation occurs before the variation step.

order to assign the model a fitness score. As a result the number of points in the training data contributes linearly to the time spent on model evaluation.

Data Width's Impact on Tree Size: If we can assume a sizable majority of the features in the data set are uncorrelated and possess a similar level of information content, it is highly likely the trees produced by GP will include references to most features. The trees will become exponentially more complex as the number of features in the training dataset increases. Therefore the resulting trees will require an exponentially greater amount of time to evaluate.

Data Complexity and Operator Set: Any dataset exhibiting a complex relationship between input and output variables will require models of still greater complexity to represent accurate solutions. More nonlinear operators like the logarithm, trigonometric functions and exponentials can be included to address more complex datasets, but at the expense of more time spent on evaluation.

Population Size: Each new model generated must be evaluated before selection can consider it for addition to the next generation's population. Traditional GP algorithms generate a number of children equal to the population size.

Therefore the number of model evaluations per generation must be on the order of the population size.

The four dimensions of scale listed above represent important opportunities to boost GP's search capabilities. Setting aside the expense of time, the performance of GP generally benefits from access to richer data of greater length, width and complexity. GP's performance also improves with larger population size, as we demonstrate in Chapter 4. Careful attention to making model evaluation fast and efficient can enable GP to tackle problems which have increased in scale along any of these four dimensions. For these reasons it is important to increase the speed of model evaluation as much as possible when constructing a system to perform GP.

FlexGP 2.0 addresses model evaluation in two parts:

C++ Model Evaluation: FlexGP 2.0 translates a batch of models to be evaluated into a C++ program where each model's calculation of fitness is represented as a function declaration. The C++ program is compiled and run to obtain the models' fitness values. We demonstrate the export of model evaluation to C++ provides a significant increase in the rate of model evaluation relative to the rate obtained in Java.

C++ Shared Memory Multithreading: The previous C++ program is modified to divide the evaluations amongst multiple threads. To benefit from multithreading, execution occurs on a machine with multiple CPUs. Multi-core CPUs are now standard in most modern computers. Parallelization is a natural means of speeding up model evaluation. GP is more easily parallelizable than many search and optimization techniques due to high degree of independence of the steps in the evolutionary loop. In particular, a GP algorithm's population is composed of a large set of models which each represent a point of current pursuit in the search space. The models are fully independent from one another and thus can be evaluated independently. Figure 3-2 depicts the evolutionary loop with the inclusion of evaluation-level parallelism.

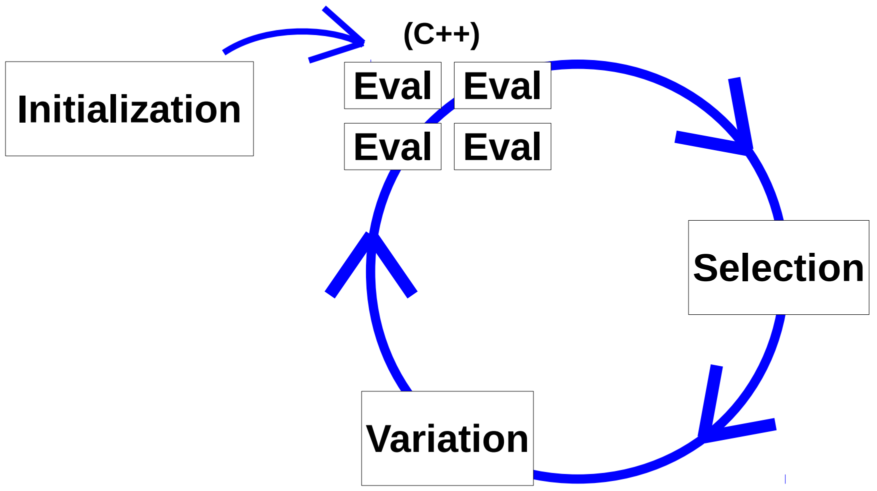


Figure 3-2: This figure visualizes FlexGP 2.0's new evolutionary loop after the introduction of evaluation-level parallelism. Each labeled box represents a step in the loop. The four "Eval" boxes represent FlexGP 2.0's parallelized C++ model evaluation. The blue path shows the flow of each population of models through the algorithm during each generation. In FlexGP the beginning of each generation occurs before the variation step.

In summary, FlexGP's evaluation-level parallelism addresses model evaluation via transferral of evaluation to C++ combined with simple multithreading. The goal of model evaluation parallelization is to increase the speed of the algorithm. This will decrease the time spend to obtain results and in doing so expand the range of data sizes, data complexities and population sizes which may be addressed by GP.

3.2 Integration of Evaluation-Level Parallelism

FlexGP's parallelization of model evaluation consists of two components. The first transfers evaluation from FlexGP's native language of Java to C++. The second component is the addition of multithreading to the C++ model evaluation. The changes made to support evaluation-level parallelism are encapsulated in FlexGP's custom Java-based GP library, evogpj.

3.2.1 C++ Model Evaluation

FlexGP's C++ model evaluation introduces a new evaluation workflow. From the perspective of the evolutionary loop, C++ model evaluation processes models in batches rather than in sequence. In FlexGP 2.0 the batch under evaluation is the set of children bred from the previous generation's population.

To support C++ model evaluation FlexGP must load training data into a shared memory region. This makes the data accessible by the C++ evaluation program by preserving the data in memory between generations to avoid repeatedly incurring the time overhead associated with loading the data.

The procedure for performing the model evaluation calculation in C++ is as follows:

1. Translate each model into a series of sequential operations (example shown in figure 3-3).

2. Compose a single C++ program which:

   A. Defines each model evaluation as a separate function consisting of each model's sequential operations determined in step 1.

   B. Invokes each model's function to evaluate the model on all training data points, storing the obtained fitness in a global array.

   C. Writes the fitness of each model from a global array to a file entitled "results.txt" in the order the models were received.

3. Write the C++ code to file.

4. Compile the C++ code to a native binary.

5. Run the resulting C++ binary.

6. Read the resulting fitness scores from "results.txt" back into Java. [1]

[1]This could have been done via shared memory instead of on disk. But the time spent reading and writing results was measured to be insignificant compared to the time consumed by evaluation.

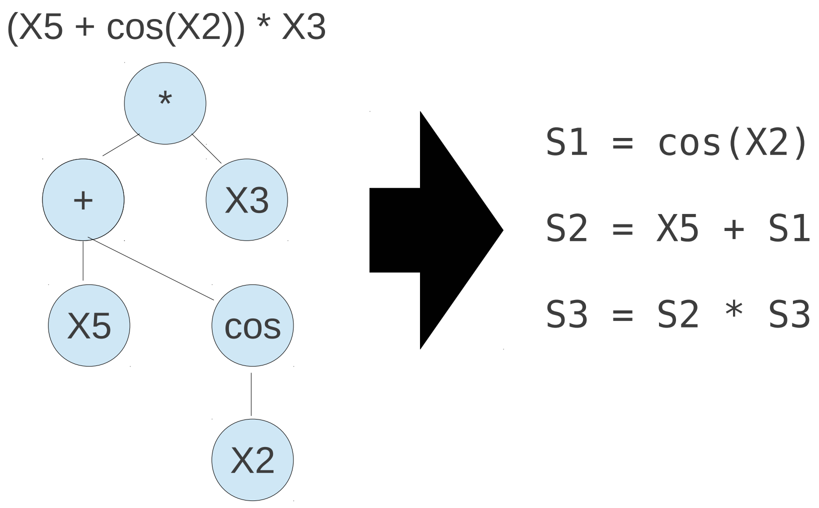


Figure 3-3: An example of the translation of models to a series of sequential steps. On the left is GP's tree representation of the model $(X5 + cos(X2)) * X3)$. The sequential translation of that model is shown at right, where the ultimate value returned by the model is *S3*.

Of the steps described above, the only two which require a non-negligible quantity of time to complete are the compilation of the C++ source and running the C++ binary. FlexGP must compile the C++ evaluation code each generation. The compilation time depends primarily on the number of models. Each FlexGP 2.0 learner is typically operated with a fixed population size of 1,000 models, which means compilation time can be viewed as a constant rather than variable cost in time. The average duration of the compilation time across the 10 repetitions of experiment *C++4* from Section 3.3, which was run with a population of 1,000 and non-multithreaded C++ model evaluation, was 5.9 seconds.[2]

To reduce the amount of memory and time consumed by model evaluation, data is stored in shared memory using the `float` type rather than the `double` type. Representing the data with `float` results in a slight decrease in the precision of the fitness calculation. This is acceptable since the purpose of model evaluation is to provide a method of comparison between models. A minute decrease in precision can only affect the comparison of models who are already closely situated in fitness-space. A

[2]The duration of compilation presents an inherent dependence of the efficacy of C++ model evaluation relative to Java model evaluation which is discussed in Chapter 6 and is not the focus of this thesis.

mistake in comparison of such models will not have an explicitly positive or negative effect on the search.

3.2.2 Adding Shared Memory Multithreading

The procedure described in the previous section requires little modification to support multithreaded model evaluation. All steps are identical to the sequence outlined in Section 3.2.1 except for step 2, the modifications to which are shown below:

2. Compose a single C++ program which

    A. Defines c `pthread`s, and assign as equal a number of models as possible to each `pthread`.

    B. Passes each `pthread` a pointer to a global array in which to save the resulting fitness scores.

    C. Passes each `pthread` a pointer to the training data, located in shared memory.

    D. Within each `pthread`, defines each of the `pthread`'s models as a separate function consisting of sequential operations determined in step 1.

    E. Defines the main method of each `pthread` to invoke each of that `pthread`'s models' function, saving the resulting fitness scores to the global array.

    F. Start each `pthread`.

    G. Once all `pthread`s have been joined with the main thread, writes the global array containing each models' computed fitness to a file entitled "results.txt".

It is safe to assume each `pthread` receives models which have on average the same complexity, since the population sort method from Section 2.2.2 sorts the population first by domination count and then by crowding distance. Balancing the load of each `pthread` ensures all `pthread`s will complete their designated evaluations at about the same time.

We don't explicitly balance the computational load of each `pthread`. However the population sort method from Section 2.2.2 will implicitly do so by sorting individuals primarily by non-domination rank. This means each front resulting from the sort will contain a range of model complexities, where the fronts are then spread sequentially throughout the population.

Multithreading will only result in a noticeable performance benefit if the computation is performed on a machine with multiple cores. Otherwise the multithreaded computation will evaluate the entire population in sequence rather than in parallel.

Apart from the execution duration of the compiled C++ binary, none of the above steps consume significantly more time when multithreading is enabled. The compile time was averages across the 10 repetitions of experiment 3 in Section 3.3, which was run with a population of 1,000 and multithreaded C++ model evaluation. The average compile time with multithreading was found to be only several tenths of a second greater than the average compile time observed without multithreading in Section 3.2.1.

3.3 Experimental Evaluation

In this section we experimentally establish the value of evaluation-level parallelism by demonstrating the speedup offered by C++ model evaluation and by the addition of multithreading.

3.3.1 Experimental Setup

We compare the performance of FlexGP operating under the following conditions:

1. *Java1*: Java-based model evaluation running on a single-core machine.

2. *C++1*: C++ model evaluation running on a single-core machine.

3 *C++4*: Multithreaded C++ model evaluation running on a four-core machine with four `pthreads`.

All experiments were conducted on the Million Song Dataset year prediction challenge, a large regression problem which is described in Appendix A. Each experiment was repeated 10 times to demonstrate statistical validity. During each repetition FlexGP was allowed to train for three hours. The information collected from each generation of each trial includes the elapsed time, the number of fitness evaluations made and the MSE_{test} and MAE_{test} of the best model. The euclidean distance metric discussed in Section 2.2.2 was used to determine the best model.

3.3.2 Results

Results are presented in two sections:

Speedup: results which demonstrate the speedup achieved by C++ model evaluation.

Performance Improvement: results which demonstrate the improvement in accuracy afforded by an increase in evaluation speed.

To best illustrate the contrast between the experiments' results, the y axis of our plots show the gain in MSE_{test} and gain in MAE_{test} rather than the absolute MSE_{test} and MAE_{test}. This allows all curves to originate for the origin, which makes the change in MSE_{test} and in MAE_{test} readily apparent. We define the gain in MSE_{test} or MAE_{test} at time i as:

$$MSE_{test}[0] - MSE_{test}[i]$$

The same definition is made for the MAE_{test}. The starting MSE_{test} and MAE_{test} of all three experiments was determined to be equivalent on average, which legitimizes this comparison.

Speedup

Figure 3-4 shows the average number of model evaluations made over time. Each point was calculated by finding the number of elapsed model evaluations made before

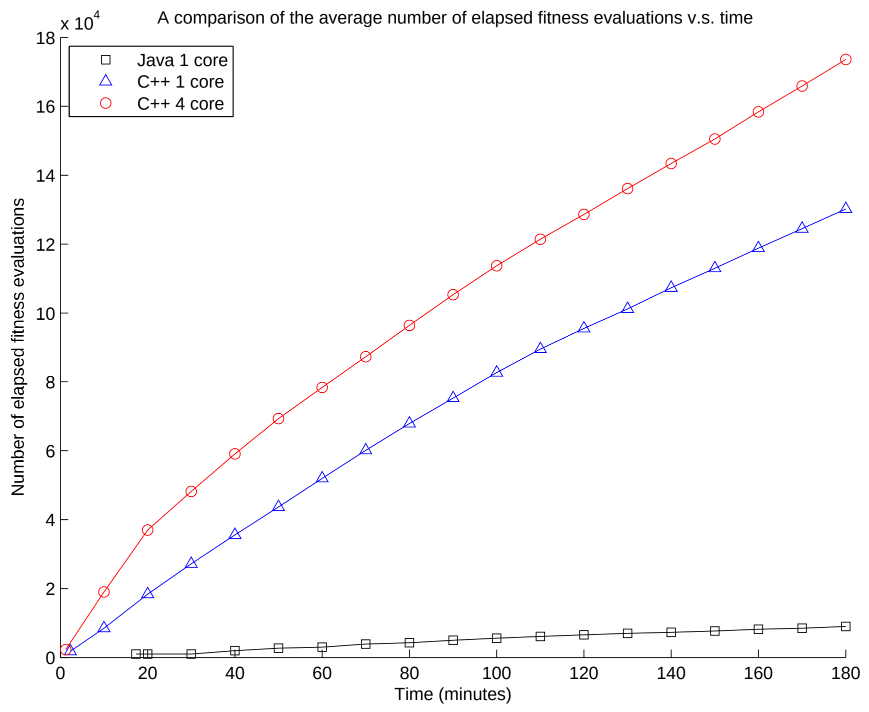


Figure 3-4: Average number of model evaluations made v.s. time for each of the 3 experiments

a given time for each learner, and calculating the average of the 10 resulting values. The standard deviation values included in table 3.1 were obtained by calculating the standard deviation of the 10 resulting values.

	Metric	30min	60min	90min	120min	150min	180min
Java1	mean($\#fitevals$)	1000	3000	5000	6600	7700	9000
	stddev($\#fitevals$)	0	0	0	516.4	675.0	666.7
C++1	mean($\#fitevals$)	27200	52000	75300	95500	113000	130200
	stddev($\#fitevals$)	4614	8353	12047	15686	19788	24485
C++4	mean($\#fitevals$)	48200	78400	105300	128600	150500	173600
	stddev($\#fitevals$)	8548	17083	24829	31655	37533	43043

Table 3.1: The average and the standard deviation of the number of fitness evals v.s. time for 10 trials of the three experiments in this chapter. All values are given in units of millions of fitness evaluations.

Performance Improvement

Figure 3-5 shows the average MSE_{test} and MAE_{test} of the best model from each trial. Each point was obtained by identifying which model scored the highest MSE_{test} or

MAE_{test} on the test data for each learner, and calculating the average of the 10 resulting values. The standard deviation values included in table 3.2 are the standard deviation of the 10 resulting values.

	Metric	30min	60min	90min	120min	150min	180min
Java1	mean(MSE_{test})	0.0000	0.0370	0.2820	0.2890	0.3030	0.3490
	stddev(MSE_{test})	0.0000	0.1067	0.4384	0.4339	0.4328	0.4214
	mean(MAE_{test})	0.0000	0.0145	0.0494	0.0557	0.0568	0.0699
	stddev(MAE_{test})	0.0000	0.0454	0.0668	0.0646	0.0652	0.0658
C++1	mean(MSE_{test})	0.1310	1.1710	2.2100	2.6730	3.0020	3.5540
	stddev(MSE_{test})	0.4073	2.0266	2.6097	3.3430	3.3986	3.5623
	mean(MAE_{test})	0.0095	0.0444	0.0890	0.1147	0.1272	0.1598
	stddev(MAE_{test})	0.0299	0.0921	0.1285	0.1709	0.1716	0.1776
C++4	mean(MSE_{test})	1.0330	5.0410	6.4930	7.6570	8.0340	8.8470
	stddev(MSE_{test})	1.6018	2.3680	2.6770	2.6156	2.6821	2.7008
	mean(MAE_{test})	0.0473	0.2167	0.2881	0.3535	0.3781	0.4246
	stddev(MAE_{test})	0.0818	0.1366	0.1666	0.1697	0.1813	0.1877

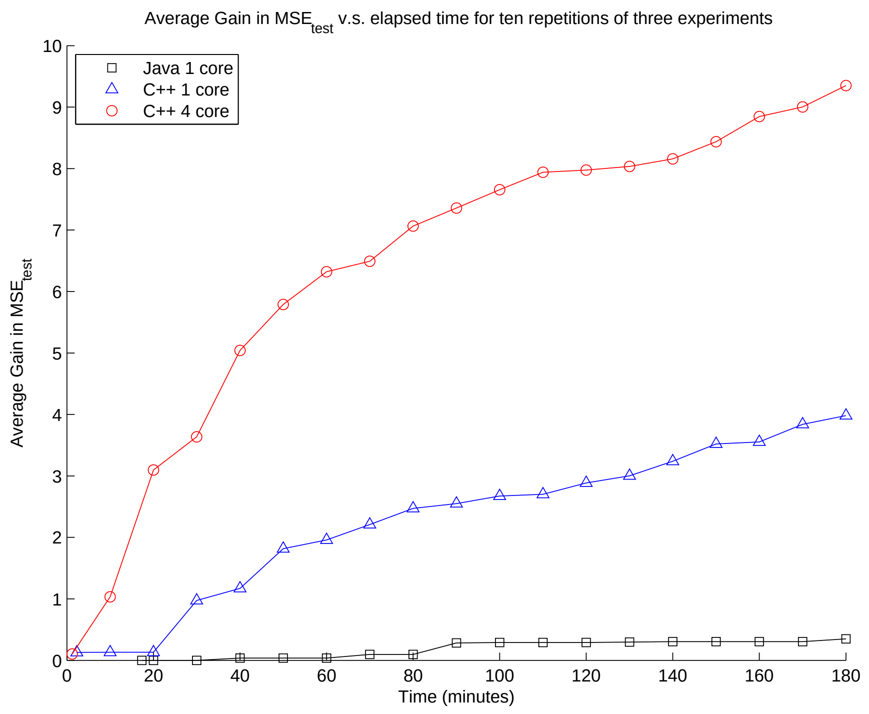
Table 3.2: The average and the standard deviation of MSE_{test} and MAE_{test} v.s. time for 10 trials of the three evaluation-parallel experiments.

3.3.3 Analysis

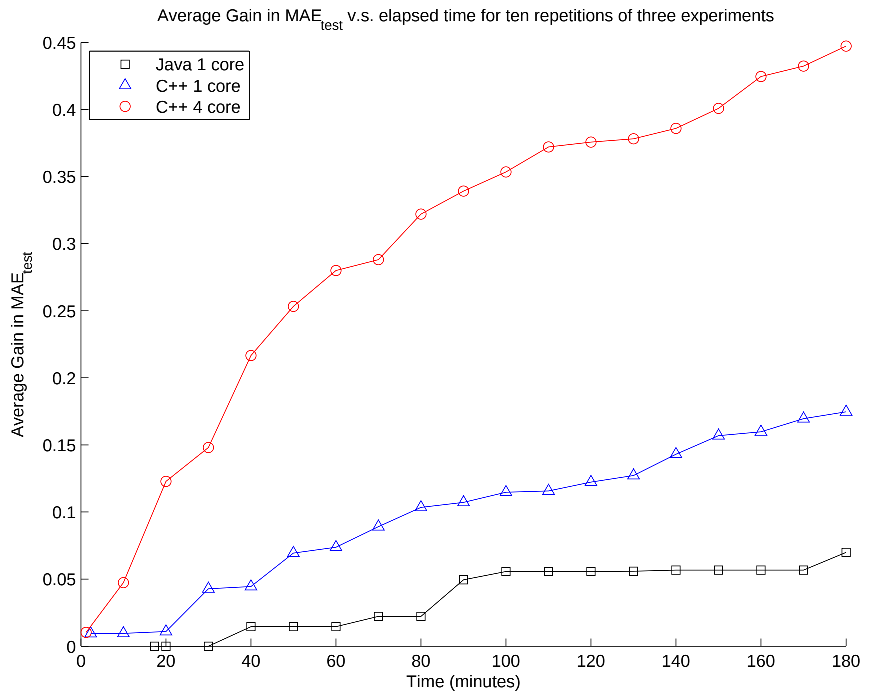
Figure 3-4 shows both experiments *C++1* and *C++4* perform model evaluations at a faster rate than *Java1*. This confirms that C++ model evaluation runs significantly faster than Java-based evaluation, and that multithreaded C++ model evaluation runs about twice as fast as C++ model evaluation with no multithreading. A comparison of the mean($\#fitevals$) for *C++4* and *Java1* from table 3.1 shows *C++4* provides 19.29 times more model evaluations on average than *Java1* in the same amount of time.

Figure 3-5 shows *C++1* and *C++4* outperform *Java1* in time, which confirms that the boost in speed from C++ model evaluation ultimately results in more accurate results, and that multithreaeded C++ model evaluation performs noticeably better than C++ model evaluation with no multithreading.

These facts in conjunction mean that multithreaded C++ model evaluation has has satisfied the sole goal of evaluation-level parallelism outlined in Section 3.1, and



(a) Average gain in MSE_{test} v.s. time of evaluation-parallel experiments



(b) Average gain in MAE_{test} v.s. time of evaluation-parallel experiments

Figure 3-5: A comparison of the gain in MSE_{test} and of the gain in MAE_{test} of the three evaluation-parallel experiments. The gain in performance is due to the increased speed of model evaluations.

that multithreaded C++ model evaluation represents a significant augmentation of FlexGP's capabilities.

Chapter 4

Population-Level Parallelism

Population-level parallelism adds a second tier of computational resource exploitation to FlexGP by replacing each factor-parallelized GP learner with a sub-network of learners given the same factorized data and parameters. Below the factor-parallel layer, each learner is given a different factorization of the data and other parameters to train on; FlexGP's population-level parallelism replaces that sole learner with a sub-network of learners each running on an independent machine. This arrangement is shown in figure 4-1. These population-parallelized learners periodically communicate their best models to other learners in the same sub-network. The transmission of best models is referred to as *migration*.

By allocating more computational power to each factorization, population parallelization results in a swifter and more powerful search and grants FlexGP a second tier of scalability with respect to data size.

4.1 Motivation and Goals

Increased population sizes are beneficial to GP. Each model in the search algorithm's population represents a candidate solution to the problem at hand. An increased population size allows the algorithm to simultaneously explore more points in the search space.

For performing machine learning with GP, this benefit is particularly important

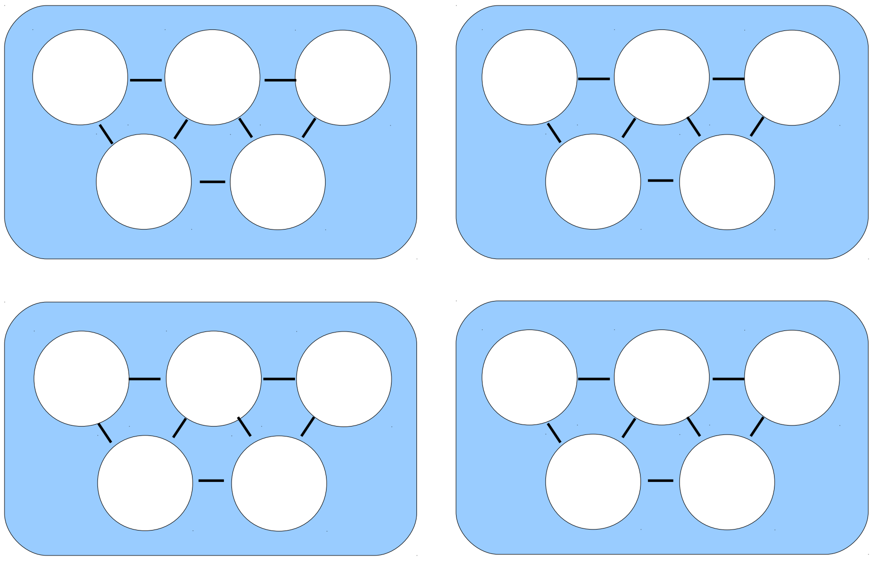


Figure 4-1: This figure depicts the combination of factor-level parallelism from FlexGP 1.0 with the population-level parallelism described in this chapter. Each blue box represents a unique factorization group with its own training and validation data split. Each white circle shows an independent population-parallelized node which sustains its own GP process. The lines between the population-parallelized nodes indicate a randomized topology for the communication of best models within each factorization group.

when the number of variables in a dataset increases and the search space expands in size and complexity.

However, the time consumed by GP scales linearly with population size, as the fitness of each model must be evaluated in each generation. This means a ten-fold increase in population size results in an algorithm which is ten times slower. An alternative strategy is needed to address increasing population size while maintaining a fast runtime.

A natural first solution to parallelize the computation is to split a large population into smaller chunks which are each supported by different machines. This essentially amounts to an emulation of what would happen on one machine, but introduces several new difficulties. The distributed GP algorithm must be able to access all models across the entire population in a centralized manner in order to perform tournament selection and possibly other steps in the evolutionary loop. To support this emulation it is necessary to provide a means of passing information about the models over the network. Further, there must be one machine which, at some point in the emulation, must make a centralized decision to complete a step like tournament selection. This means a machine or machines could represent a central point in the distributed network of machines, which is undesirable from the perspective of fault tolerance. More importantly, it is unlikely this scheme would increase the rate of computation by a large amount, since the need for centralization implies all machines must wait for a centralized step to complete before proceeding. Finally, this rate is further limited by network latency and the possibility of network bottlenecking due to heavy traffic required to support the emulation.

Fortunately a better solution exists. It is not necessary to preserve the continuity of the population across the different machines. Splitting a large population into smaller chunks which are each sustained by independent GP learners running on a different machines is an effective means of preserving speed while increasing overall population size. Doing so preserves the benefit of supporting more points in the search space while simultaneously guaranteeing the aggregate population will have the same runtime as any of the constituent learners.

This configuration may provide further benefits; it is an example of a commonly used technique in evolutionary computation known as the island model. Under the island model each island exists as an independent learner which supports a small population. As the algorithm proceeds the islands' populations will grow relatively more diverse as they naturally come to focus on different regions of the search space due to the inherent stochasticity of evolution.

Each island periodically copies a portion of its population, typically that island's best models, to another island. This transfer is known as migration, and enables a mixing of genetic information throughout the network, thus allowing the search progress of each island to be communicated to the others. The inclusion of migration will even further reduce the time to an accurate solution. In addition to allowing the most fit models' genes to permeate the network, migration also regulates the diversity between islands' populations, preventing extreme divergence in population which could lead to overfitting.

Not only does migration decrease the time to a good solution, but the diversity between populations enables the discovery of solutions which outperform those which can be found without migration in a reasonable amount of time. This means migration can produce a better than linear speedup of model performance.

Much work has been conducted regarding the characteristics of migration topologies and parameters for optimal flow of genetic material through the network. With careful attention to these aspects, migration can have significantly beneficial effects on the search process[13][1][4][16].

However, FlexGP stands out from most work on the island model by granting each island a different cloud-backed machine on which to run. In fact many implementations of the island model have sustained all islands on the same machine. The focus of other implementations has been on investigating the effects of controlling diversity, rather than on increasing the wall-clock speed or on handling large and complex datasets.

We demonstrate in this chapter the island model is particularly advantageous from this perspective when each island is supported by an independent GP learner running on its own machine with periodic migration. This means the rate of the overall algorithm is governed only by the average speed of each of the small-population learners, while preserving the potentially super-linear performance gain associated with migration.

To summarize, the goals of FlexGP's population-level parallelism are to achieve the following:

1. Allow solutions to reach the same degree of accuracy as can be attained without population-level parallelism, but in significantly less time.

2. Discover solutions which are more accurate than any which could be obtained without population-level parallelism.

3. Provide FlexGP with another, highly effective dimension of scalability.

To achieve those goals, our contributions in this chapter are two-fold:

1. Design a system which can perform population-level parallelism with migration.

2. Demonstrate the efficacy of population-level parallelism with migration in both time and accuracy.

These techniques are not trivial to implement. Population-level parallelism with migration requires support at several levels. At the resource level, support is needed for obtaining the required computational resources for parallelization. At the network level support is required for discovering and recording the existence of other nodes in the network and for sending and receiving migrants and other inter-node messages. Finally, support is required at the algorithmic level for selecting emigrants and ingesting immigrants. The next section will discuss the design FlexGP uses to meet these challenges.

4.2 Integration of Population-Level Parallelism

Under population-level parallelism each factor-tier[1] node is replaced with a sub-network (sub-network) of population-tier[2] nodes. Each sub-network consists of multiple constituent nodes which are configured with the same parameters and data as the factor-tier node would have been. Each node periodically communicates its best models to another randomly chosen node in the sub-network.

Population-level parallelism preserves FlexGP's learner-agnosticism. While the concept of migration may not be portable to other learning algorithms, FlexGP can sustain parallel computation of any learning algorithm in each sub-network through population-level parallelism.

[1]Factor-tier nodes are nodes which have received a distinct factorization of the data and problem parameters.

[2]Population-tier nodes are nodes which maintain independent GP learners and may periodically exchange best models as part of a sub-network. Population-tier nodes are given the same factorization as others in a sub-network.

4.2.1 Initial Resource Acquisition

The first step in starting FlexGP is to acquire resources from the cloud infrastructure. FlexGP uses a recursive "distributed startup" procedure for easy scalability in network size. Here we augment the recursive startup procedure to enable the creation of population-level sub-networks as follows:

1. The user specifies a desired number of factored sub-networks F, nodes per sub-network S and max number of children k during startup.

2. The gateway node is started, which acts as a connection between the client and the FlexGP system running on the cloud. The gateway node also serves as a factor-tier node.

3. The gateway starts k children, passing forward to them the number of nodes which still need to be started. The children each define a new sub-network.

4. Subsequent nodes start more factor-tier children as needed to fulfill the desired sub-network quantity F. All children define a new sub-network. The resulting F sub-networks named $subnet_f$ are numbered 1 through F.

5. Step 4 is repeated in a recursive manner until n_F nodes have been started. Each node is referred to as $node_{fi}$, where f indicates the sub-network and i refers to i-th node in the sub-network.

6. After $node_{fi}$ is started in steps 2 through 5, $node_{fi}$ starts $(S - 1)$ children in the population-tier. $node_{fi}$ is also a population-tier node.

In the end the full network will be of size $F \cdot S$.

4.2.2 Establishing a Sub-Network

To support population-level parallelism at the network level, each node holds a sub-network identifier s_i which indicates the node belongs to sub-network i. The sub-network identifiers are generated during the initial distributed startup procedure.

Each node also maintains a list of other nodes which have identified themselves as belonging to s_c.

When a FlexGP node desires to send a network-level message like a ping, that node obtains information about its destination by querying its neighbor list, which contains the IP and the TCP port of all other nodes discovered via the gossip protocol. When the evolutionary process desires to send a packet of migrants, the FlexGP node randomly selects a destination from its list of sub-network-neighbors and then queries the neighbor list to obtain the IP and port of that destination.

Simple modifications are required to FlexGP 1.0's *neighborlist* datastructure which each FlexGP node uses to keep track of all other known nodes in the network. The *neighborlist* from FlexGP 1.0 contained a map which associates each node's unique ID with an IP and port at which to contact that node, as well as a timestamp indicating when last successful contact was made.

FlexGP 2.0 introduces two additions to this datastructure: an additional field in the map structure which indicates each node's sub-network ID, and a list of IDs of nodes known to belong to the same sub-network. Now when a FlexGP learner wishes to perform migration it randomly selects an ID from the list of other nodes in the sub-network, and uses that ID to obtain the IP and port from the *neighborlist* map.

4.2.3 Modifications to the GP Learner

To support migration the algorithm must be able to send and receive models. Sending models to another node is known as *emigration*, and receiving models from another node is known as *immigration*.

Emigration

There are five parameters which govern emigration:

1. Enable or disable migration.

2. m, the number of migrants to be sent in each migration message

3. g_{start}, the number of the first generation when migration should occur

4. g_{modulo}, the number of generations to wait before migration happens again

5. The criterion used to select the best m models from the population

In the architecture of FlexGP the Evolve thread is responsible for running the GP learner. Further details are available in previous work on FlexGP [6].

At the completion of each generation the Evolve thread checks the emigration parameters to decide whether to send emigrants. When the Evolve thread does elect to send migrants, it chooses the best m models from the population according to the specified selection criterion, and sends those models to a randomly selected destination node which belongs to the same sub-network. The datastructure used to do this is described in Section 4.2.2.

The criterion used for selecting the best model may also be used to choose the best m models for emigration. The criterion supported in FlexGP are discussed in Section 2.2.2.

Immigration

Support for immigration is provided by the FlexGP mailbox, which maintains a queue of received messages which contain immigrant models. The Evolve thread checks the immigrant queue at the beginning of each generation, adds any new immigrants to the population and removes the corresponding message from the mailbox queue.

4.3 Experimental Evaluation

In this section we experimentally establish the value of population-level parallelism by addressing the following questions:

1. Is larger population size beneficial to the performance of GPSR?

2. Will a larger distributed population with migration yield the same accuracy as a smaller centralized population, but in significantly less time?

3. Will a distributed population with migration yield better accuracy as a centralized population of the same overall size, and in significantly less time?

Note that define the word "distributed" to imply an increase in the overall amount of computational resources consumed, with the ultimate goal of providing more accurate results faster.

4.3.1 Experimental Setup

To address the above questions, three experiments were conducted.

***1K-SOLE*: Non-parallelized GP Learner.** A single GP learner with a population of 1,000 models and 3 hours of training

***10K-SOLE*: Large non-parallelized GP Learner.** A single GP learner with a population of 10,000 models and with 24 hours of training

***1K-PAR10*: Population-parallelized GP Learner.** A sub-network of 10 GP learners with a population of 1,000 models per node and with 3 hours of training

The GP learners in experiments *1K-SOLE* and *10K-SOLE* were each given an unique 70% segment of the MSD dataset for training as described in Section A. For each iteration of experiment *1K-PAR10* the 10 GP learners which composed the sub-network all trained on the same unique 70% segment of the MSD dataset. The training segment was changed for each iteration. Therefore all GP learners in all experiments had access to the same amount of data.

Experiment *1K-PAR10* was additionally configured with a migrant size of 50 best models sent to one random neighbor per generation. To allow an even comparison across the different population sizes the large non-parallelized experiment was allowed to train for 24 hours which is nearly 10 times as long as the non-parallelized and population-parallelized experiments. Each experiment was replicated 10 times to demonstrate statistical validity.

The *euclidean distance* metric described in Section 2.2.2 was used to determine the best model for each generation. The same metric was used to select the best 50 models as migrants in the population-parallelization experiment.

4.3.2 Results

Each iteration of each experiment was processed by performing fusion on the 50 best models obtained over time. Each learner produces one model per generation which is judged to be the best. The best 50 models since $t = 0$ were identified for 20 points in time between 0 and the expected durations of each experiment. For each point, an ARM meta-model was trained on 80% of the MSD data, and was evaluated on the remaining 20% to calculate the MSE_{test}. The 80% training data used for fusion consisted of the original 35% used as training data on each node plus the remaining 45% not reserved for testing data.

The results are shown in figure 4-2. Each of the 10 iterations for the three experiments is shown as a separate line in the plot. The results are presented in tabular form in table 5.2.

Table 4.1 shows the number of generations normalized by the duration of each experiment. This can be used to gauge the average complexity of the models under evaluation. Experiment *10K-SOLE* shows a lower mean number of generations per unit time than the other experiments, implying a larger model complexity.

Expt.	mean($gens$)	stddev($gens$)	mean($time$)	std($time$)	mean($\frac{gens}{time}$)	stddev($\frac{gens}{time}$)
1K-SOLE	339.9	117.6	5.7	0.15	61.9	19.2
10K-SOLE	105.9	16.5	23.0	0.47	4.6	0.69
1K-PAR10	140.3	27.2	2.8	0.07	50.0	9.7

Table 4.1: The average and the standard deviation of the number of generations, total time in hours, and generations per hour for 10 trials of experiments *1K-SOLE*, *10K-SOLE* and *1K-PAR10*.

	Metric	1.5hrs	3hrs	4.5hrs	6hrs	12hrs	24hrs
1K-SOLE	mean(MSE_{test})	119.416	117.370	116.768	115.733	-	-
	stddev(MSE_{test})	4.843	5.672	5.850	5.807	-	-
	mean(MAE_{test})	8.738	8.640	8.611	8.553	-	-
	stddev(MAE_{test})	0.222	0.278	0.296	0.286	-	-
10K-SOLE	mean(MSE_{test})	123.270	121.117	120.092	119.379	113.930	111.535
	stddev(MSE_{test})	3.742	2.728	3.073	3.631	3.954	2.924
	mean(MAE_{test})	8.939	8.842	8.798	8.757	8.441	8.319
	stddev(MAE_{test})	0.148	0.144	0.159	0.194	0.227	0.154
1K-PAR10	mean(MSE_{test})	113.507	110.989	-	-	-	-
	stddev(MSE_{test})	5.337	3.913	-	-	-	-
	mean(MAE_{test})	8.442	8.278	-	-	-	-
	stddev(MAE_{test})	0.278	0.222	-	-	-	-

Table 4.2: The average and the standard deviation of MSE_{test} and MAE_{test} v.s. time for 10 trials of experiments *1K-SOLE*, *10K-SOLE* and *1K-PAR10*.

4.3.3 Analysis

From 4.1, the mean number of generations per unit time is lower for experiment *10K-SOLE* than for experiments *1K-SOLE* and *1K-PAR10*. This provides verification a GP learner with a larger non-parallelized population (10k) will run approximately 10 times slower than the learner from experiment *1K-SOLE* with a proportionally smaller population (1k), at the cost of 10 times more computational resources consumed by *1K-SOLE*.

Interestingly, the mean number of generations per unit time is slightly lower for experiment *1K-PAR10* than for experiment *1K-SOLE*. This indicates the models under evaluation by the GP learners in experiment *1K-PAR10* were on average of greater complexity than those on the learners from experiment *1K-SOLE*. This provides corroborating evidence for the increased performance of experiment *1K-PAR10* relative to experiment *1K-SOLE*.

We will now use the results shown in figure 4-2 to answer the questions posed in Section 4.3 and to show that population-level parallelism satisfies the goals outlined in Section 4.1.

1. Is large population size beneficial to GPSR?

To answer this question we will compare experiment *1K-SOLE* and experiment *10K-SOLE*, which have a population size of 1k and 10k respectively. The final performance of experiment *10K-SOLE* greatly exceeds that obtained by experiment *1K-SOLE*. Therefore the answer to this question is that large population size is in fact beneficial to GPSR.

2. Will a larger distributed population with migration yield the same accuracy as a smaller centralized population, but in significantly less time?

To answer this question we will compare experiment *1K-SOLE* and experiment *1K-PAR10*. The performance of experiment *1K-PAR10* greatly exceeds that of experiment *1K-SOLE*, and does so in half the time available to experiment *1K-SOLE*. Therefore the answer to question 2 is yes.

3. Will a distributed population with migration yield better accuracy as a centralized population of the same overall size, and in significantly less time?

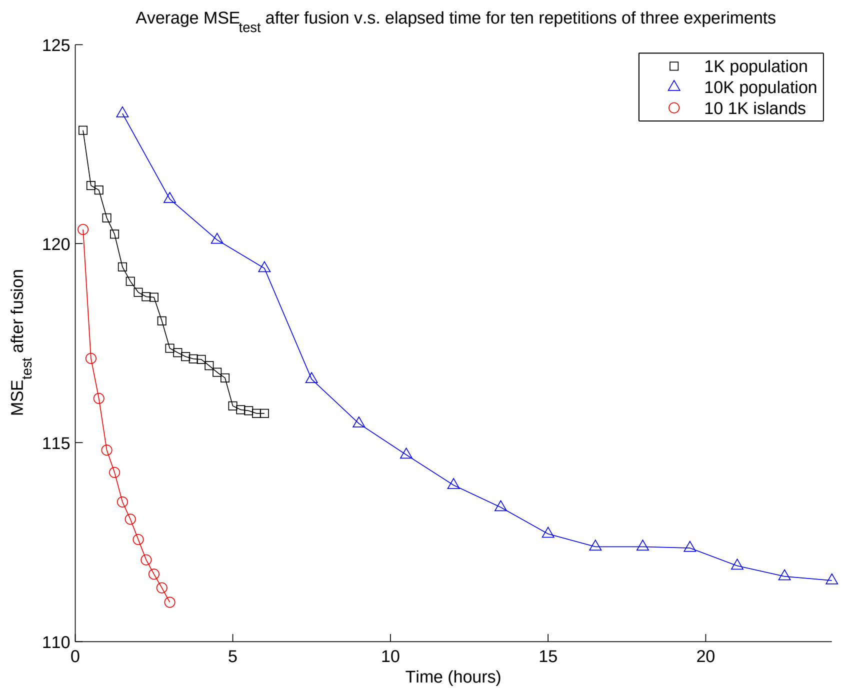
To answer this question we will compare experiment *10K-SOLE* and experiment *1K-PAR10*. The final performance of experiment *1K-PAR10* is close to but exceeds the final performance of experiment *10K-SOLE*. Therefore the answer to this question is yes.

To summarize, the results presented in this section have provided answers to the questions asked in 4.3, and therefore have provided evidence FlexGP 2.0 has satisfied the three goals outlined in the introduction of this chapter:

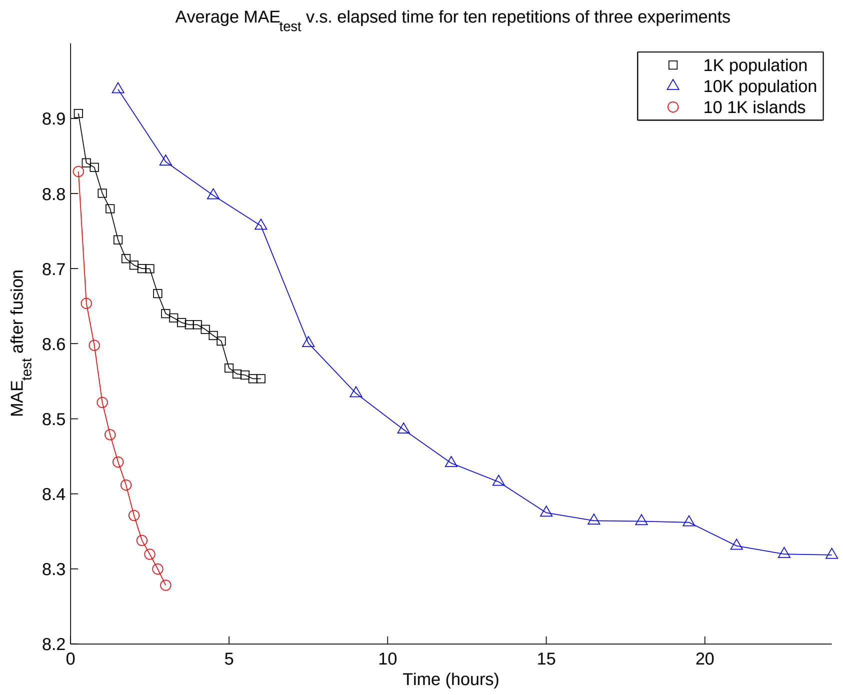
1. Allow solutions to reach the same degree of accuracy as can be attained without population-level parallelism, but in significantly less time.

2. Discover solutions which are more accurate than any which could be obtained without population-level parallelism.

3. Provide FlexGP with another, highly effective dimension of scalability.



(a) MSE_{test} of population-parallel experiments



(b) MAE_{test} of population-parallel experiments

Figure 4-2: A comparison of the time-series results of the three population-parallel experiments described in this chapter. Each point on a curve represents the MSE_{test} or MAE_{test} of the fusion meta-model generated with the best 50 models available up to that point in time. The fusion process is described in Appendix C. The starting time is different for experiment *10K-SOLE* because the evaluation of the initial generation took more than one hour on average.

Chapter 5

FlexGP 2.0: Multi-Level Parallelism

This section reiterates the goals of FlexGP 2.0, presents the design required to achieve those goals and provides experimental evidence which demonstrates FlexGP 2.0 meets those goals. FlexGP 2.0 integrates all levels of parallelism discussed in this thesis:

Evaluation-Level Parallelism (Chapter 3) increases the speed of FlexGP via multithreaded C++ model evaluation.

Search-Level Parallelism (Chapter 2) introduces a multi-objective optimization algorithm which improves the search characteristics and flexibility of FlexGP.

Population-Level Parallelism (Chapter 4) increases the size of FlexGP's population of candidate models by distributing the population among multiple independent learners, and introduces migration as a means for distributed learners to share search progress.

Factor-Level Parallelism (Section 1.1) increases accuracy by giving each learner a subset of the data or function set and by providing a method for fusing models from the different environments to create a highly accurate meta-model. Originally included in FlexGP 1.0.

We seek here to demonstrate the full power of FlexGP 2.0 by combining the three levels of parallelism discussed in this thesis with FlexGP's remaining layer, factor-level parallelism. This combination represents a leverage of all the advantages and optimizations offered by FlexGP 2.0.

The goals of FlexGP 2.0 are to accomplish the following:

1. Improve the ability of FlexGP to learn swiftly and effectively when operating on large data sets.

2. Produce solutions with the same accuracy as were obtained from FlexGP 1.0, but in significantly less time.

3. Produce solutions with better accuracy than those obtained from FlexGP 1.0.

4. Yield a system which produced performance comparable to other machine learning methods.

5.1 Integration of Multi-Level Parallelism

This section discusses the design which integrates FlexGP 2.0's four levels of parallelism. Detailed descriptions of the designs for each layer are outlined in each layer's chapter in this thesis.

Each layer may be activated or deactivated when the user starts FlexGP. The parameters which govern each layer are discussed in the subsequent sections.

To launch FlexGP a user will initiate the distributed startup procedure described in Section 4.2.1. Several groups of parameters are required to initiate distributed startup.

Distributed Startup Parameters: To initiate distributed startup the user must specify a set of FlexGP parameters which will be used to configure the entire network of FlexGP nodes. Some of these parameters inform the distributed startup procedure on how to interface with the cloud infrastructure.

FlexGP Node Parameters: Some of the parameters specified before distributed startup control the behavior of each FlexGP node and are passed to the FlexGP Java executable when it is initiated by the distributed startup procedure.

evogpj Parameters: The user must provide an evogpj parameters file containing the parameters which will configure each GP learner.

Note that this section only discusses the details of parameters which were modified from FlexGP 1.0 and are important to the design of FlexGP 2.0. FlexGP includes many parameters which govern the behavior of evogpj, FlexGP's underlying GP library, as well as several parameters which control the networking and other parts of FlexGP and are not directly related to the learning process. More detailed descriptions of FlexGP's parameters can be found in Appendix B, Appendix D and in previous publications documenting FlexGP 1.0 [6].

5.1.1 Integration of Evaluation-Level Parallelism

Evaluation-level parallelism is governed by the following parameters:

Objective function: C++ model evaluation will be enabled if the user specifies the *ExternalFitness* objective function as one of the functions used in the multi-objective optimization algorithm.

Number of threads: Multithreaded C++ model evaluation will be enabled if this parameter is set to a value greater than 1.

Evaluation-level parallelism is a feature of FlexGP's underlying evogpj library. Therefore the above parameters must be set in the evogpj properties file which the user passes to FlexGP at runtime.

5.1.2 Integration of Search-Level Parallelism

The key parameters for controlling multi-objective optimization are:

Objective functions: any number of objective functions. Multi-objective optimization is activated if more than one is listed.

Tournament Selection Operator: crowded tournament selection.

Best Model Selection Method: euclidean or hands-off. Section 2.2.2.

Like C++ model evaluation, multi-objective optimization is part of the evogpj library and is configured via the evogpj properties file passed to FlexGP at runtime.

5.1.3 Integration of Population-Level Parallelism

The parameters which affect population-level parallelism are:

Sub-network size: the number of learners started in each sub-network.

Migration activation: enables or disables migration within sub-networks.

Migration size: the number of models to be included in each group of emigrants.

Migration start generation: the generation in which to begin sending emigrants.

Migration rate: the number of generations to wait before sending emigrants again.

The sub-network size parameter is specified as an optional command-line argument of the program used to initiate the distributed startup procedure described in Section 4.2.1. The remaining four parameters governing migration are passed to the FlexGP Java executable as optional command-line arguments by the distributed startup procedure.

5.1.4 Integration of Factor-Level Parallelism

The factorization parameters are:

Data row factorization percentage: specifies what percentage of the training data to subsample for factorization.

Data column factorization: specifies which columns of the data GP will be allowed to train on.

GP operator set factorization: specifies a series of lists of function sets to be selected randomly by each factored learner or sub-network.

Each of these parameters is specified as an optional command-line argument of the program used to initiate the distributed startup procedure. More details on factor-level parallelism are available in prior work on FlexGP [6].

5.2 Experimental Evaluation

5.2.1 FlexGP 2.0 v.s. FlexGP 1.0

Experimental Setup

To demonstrate FlexGP's fulfillment of the goals presented at the beginning of this chapter we conducted the following experiments:

***1.0*:** 8 nodes with data-factorized FlexGP 1.0 and Java-based fitness evaluation.

***2.0-NO-POP*:** 8 nodes with data-factorized, multi-objective FlexGP 2.0 and multithreaded C++ model evaluation but without population-level parallelism.

***2.0-POP*:** 8 sub-networks with data-factorized, multi-objective FlexGP 2.0 and multithreaded C++ model evaluation, with 10 nodes per sub-network.

Experiment *1.0* demonstrates the performance of FlexGP with factorization but before the addition of the evaluation, search and population parallelization layers. Experiment *2.0-NO-POP* demonstrates FlexGP's performance with all but population parallelization. Experiment *2.0-POP* demonstrates the operation of FlexGP with all four parallelization techniques. A visual summary of this information is shown in table 5.1.

Note that experiment *2.0-POP* requires 10 times as many machines as the other two experiments. This increase in computational resource load is provided in order to obtain results of increased accuracy at a swifter speed.

	Evaluation	Search	Population	Factorization
1.0				✓
2.0-NO-POP	✓	✓		✓
2.0-POP	✓	✓	✓	✓

Table 5.1: The configuration for each of the three FlexGP 2.0 experiments in this chapter. Each column corresponds to one of the four levels of parallelism in FlexGP 2.0. Each row corresponds to one of the three experiments. A check mark indicates an experiment includes the level of parallelism which belongs to the column.

The data in these experiments was factored so that each node in experiments *1.0* and *2.0-NO-POP* and was given a different 35% factored training split. Correspondingly, each sub-network of nodes in experiment *2.0-POP* was given a different 35% split of the training data, where each node in the sub-network received the same 35% split as the others.

All experiments were conducted on the Million Song Dataset year prediction challenge, which is described in Appendix A. Each of the three experiments was repeated 10 times for statistical validity.

The comparison of experiments *1.0* and *2.0-POP* shows the difference in performance between FlexGP 1.0 and FlexGP 2.0. The comparison of experiments *1.0* and *2.0-NO-POP* shows the difference in performance between FlexGP 1.0 and FlexGP 2.0 stemming solely from the addition of the evaluation and search parallelization layers, but not from population parallelization. Finally, the comparison of experiments *2.0-NO-POP* and *2.0-POP* shows the difference in performance of FlexGP 2.0 from only the addition of population parallelization. Experiment *2.0-NO-POP* was included to highlight the performance gain derived solely from population-level parallelization with migration.

Results

Figures 5-1a and 5-1b respectively show the MSE_{test} v.s. time and the MAE_{test} v.s. time for each of the ten iterations of the three experiments. Each experiment consists of 8 factorization groups, where each group contains a node or nodes which were trained with the same factorization of the training data.

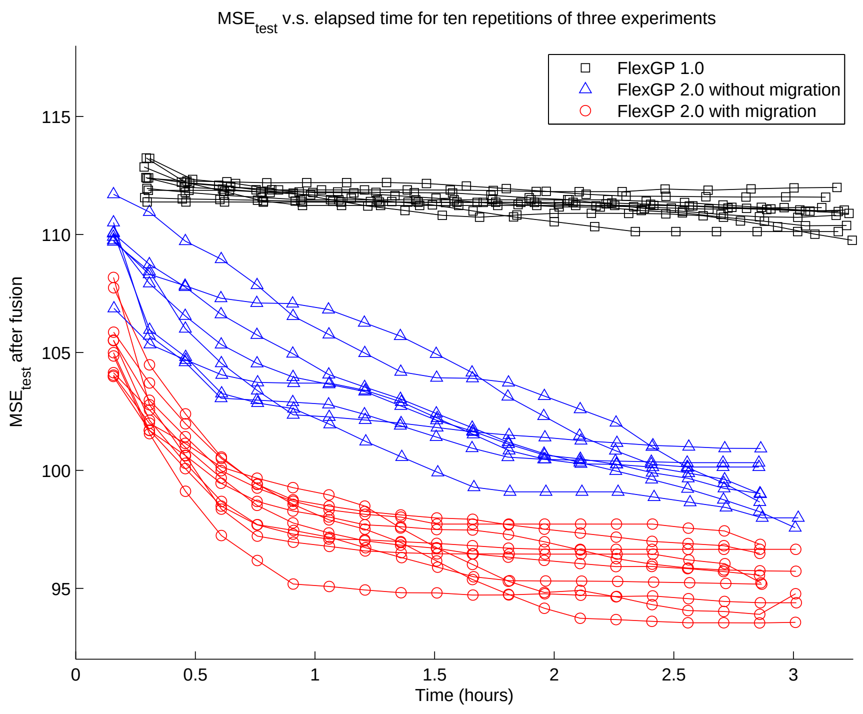
The performance of a fused meta-model trained on the current 50 best models was calculated for 20 points in time evenly spaced over the 3 hour interval of the experiment. The details of fusion and the calculation of the MSE_{test} and MAE_{test} for fusion are discussed in Appendix C.

Table 5.2 shows the MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} after fusion for various points in time. The $\sqrt{MSE_{test}}$ was included to provide further meaning to the results, as the units of $\sqrt{MSE_{test}}$ for this problem are in *years* whereas the units of the MSE_{test} are in $years^2$.

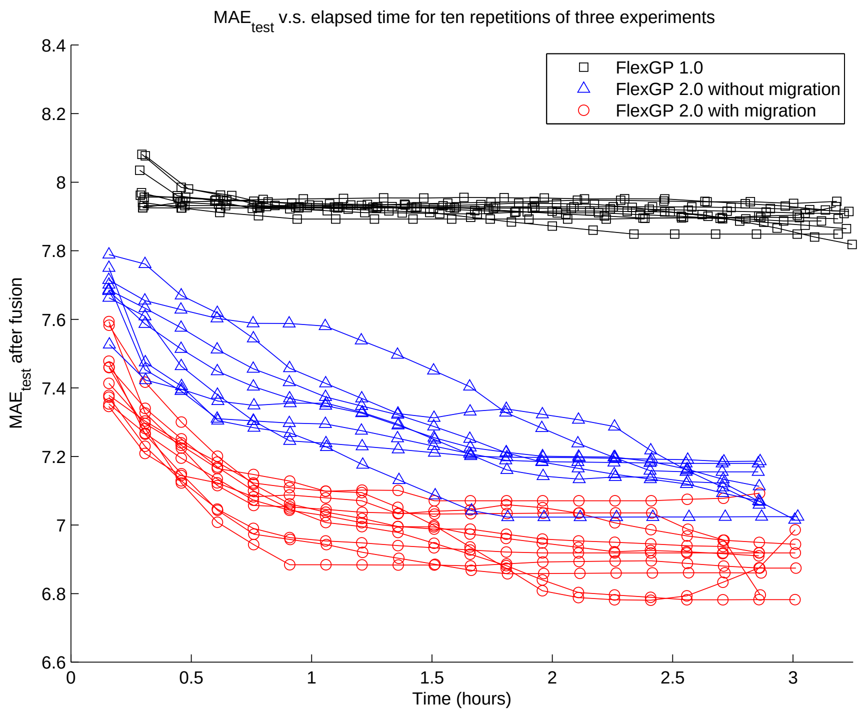
Table 5.2 contains values calculated for $\sqrt{MSE_{test}}$. However, a plot for $\sqrt{MSE_{test}}$ is not shown because both the shape and the numerical values can be inferred from the plot for MSE_{test}.

	Metric	30min	60min	90min	120min	150min	180min
1.0	mean(MSE_{test})	111.974	111.657	111.517	111.336	111.143	110.952
	stddev(MSE_{test})	0.331	0.287	0.346	0.376	0.471	0.528
	mean($\sqrt{MSE_{test}}$)	10.582	10.567	10.560	10.552	10.542	10.533
	stddev($\sqrt{MSE_{test}}$)	0.016	0.014	0.016	0.018	0.022	0.025
	mean(MAE_{test})	7.937	7.926	7.920	7.917	7.906	7.897
	stddev(MAE_{test})	0.014	0.013	0.015	0.015	0.025	0.025
2.0-NO-POP	mean(MSE_{test})	106.439	104.091	102.714	100.975	100.171	99.396
	stddev(MSE_{test})	1.733	1.741	1.524	1.179	0.673	1.039
	mean($\sqrt{MSE_{test}}$)	10.317	10.202	10.135	10.048	10.008	9.970
	stddev($\sqrt{MSE_{test}}$)	0.084	0.085	0.075	0.059	0.034	0.052
	mean(MAE_{test})	7.483	7.357	7.300	7.196	7.159	7.117
	stddev(MAE_{test})	0.144	0.125	0.117	0.094	0.059	0.063
2.0-POP	mean(MSE_{test})	100.867	97.822	96.969	96.052	95.760	95.355
	stddev(MSE_{test})	0.948	1.184	0.971	1.235	1.290	1.148
	mean($\sqrt{MSE_{test}}$)	10.043	9.890	9.847	9.800	9.785	9.765
	stddev($\sqrt{MSE_{test}}$)	0.047	0.060	0.049	0.063	0.066	0.059
	mean(MAE_{test})	7.132	7.018	6.978	6.916	6.909	6.879
	stddev(MAE_{test})	0.085	0.068	0.058	0.087	0.092	0.088

Table 5.2: The average and the standard deviation of MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} v.s. time for 10 trials of the three experiments conducted in this chapter.



(a) MSE_{test} of total experiments



(b) MAE_{test} of total experiments

Figure 5-1: A comparison of the time-series results of all ten iterations of the three experiments conducted in this chapter.

Analysis

This section provides a discussion of the results obtained from the experiments comparing the performance of FlexGP 1.0 and FlexGP 2.0.

The results shown in figure 5-1 clearly show FlexGP 2.0 with all factorizations enabled outperforms FlexGP 1.0. This satisfies goal 1 from the beginning of this chapter.

It takes four times longer for experiment *1.0* to reach an MSE_{test} of under 101 that it does for experiment *2.0-POP* to do so. This satisfies goal 2 from the beginning of this chapter.

Both experiment *2.0-NO-POP* and experiment *2.0-POP* outperform experiment *1.0* on average. This satisfies goal 3 from the beginning of this chapter.

5.2.2 Comparison of FlexGP 2.0 with *vowpal wabbit*

Experimental Setup

The *vowpal wabbit* project is a machine learning system which uses gradient descent to perform linear regression. Previous work has used *vowpal wabbit* to address the Million Song Dataset year prediction challenge [2].

We trained and evaluated *vowpal wabbit* with the same training and test data as were given to FlexGP in the experiments from Section 5.2.1. We used the same parameters[1] to run *vowpal wabbit* as were cited in previous work[2].

FlexGP 2.0 trained 80 learners for 3 hours where each learner executed on a separate machine. As *vowpal wabbit* performs simple stochastic gradient descent for linear regression it only took a few minutes to train and evaluate *vowpal wabbit* on the MSD executing on one multicore workstation. This comparison is discussed further in Section 5.2.2.

[1]The parameters used are `--passes 100 --loss_function squared -l 100 --initial_t 100000 --decay_learning_rate 0.707106781187`

Results

Table 5.3 contains the MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} obtained by running *vowpal wabbit* on each of the 8 training factorizations. Table 5.4 contains the MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} averaged across the 8 factorizations.

Factorization	0	1	2	3	4	5	6	7
MSE_{test}	88.168	88.302	88.614	87.923	88.537	87.929	88.107	88.379
$\sqrt{MSE_{test}}$	9.390	9.397	9.414	9.378	9.409	9.377	9.387	9.401
MAE_{test}	6.794	6.800	6.812	6.785	6.807	6.786	6.796	6.799

Table 5.3: The MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} of *vowpal wabbit* on the MSD year prediction problem, when trained on each of 8 different data factorizations. Each training factorization consisted of 35% of the valid data. 20% of the remaining data was used for testing.

	FlexGP 2.0	*vowpal wabbit*
mean(MSE_{test})	95.355	88.245
mean($\sqrt{MSE_{test}}$)	9.765	9.394
mean(MAE_{test})	6.879	6.797

Table 5.4: The MSE_{test}, $\sqrt{MSE_{test}}$ and MAE_{test} of *vowpal wabbit* and FlexGP 2.0 on the MSD year prediction problem. The *vowpal wabbit* values were obtained by averaging across all 8 training data factorizations shown in table 5.3. Each training factorization consisted of 35% of the valid data. 20% of the remaining data was used for testing.

Analysis

Experiment 3 from Section 5.2.1 (FlexGP 2.0 with migration) yields a $\sqrt{MSE_{test}}$ and MAE_{test} of 9.765 and 6.879 respectively, which is comparable to the values of 9.394 and 6.797 given by *vowpal wabbit* as shown in table 5.4.

The performance of *vowpal wabbit* as measured here is not as good as the performance obtained in prior work[2] with the same *vowpal wabbit* parameters. The reason for this is the amount of data used to train *vowpal wabbit* in this experiment is lower than the amount used in prior work. For a fair comparison with the experiments from Section 5.2.1 we gave *vowpal wabbit* the same amount of data which was given to FlexGP 1.0 and FlexGP 2.0. Prior work on applying *vowpal wabbit* to the

MSD[2] used a published train-test split of the MSD which allocates 90% to training.[2] The fact that linear regression via *vowpal wabbit* gave slightly more accurate results than FlexGP may indicate the input variables have a fairly linear relationship to the output in the MSD year prediction problem.

Further, it is possible the partitioning of the data by author produced a problem which is highly linear, allowing *vowpal wabbit* to perform with higher accuracy than FlexGP. A different method of data partitioning would possibly see the nonlinear models produced by FlexGP outperform a linear approach.

The primary advantage FlexGP provides over *vowpal wabbit* is a more complex model. GP will outperform linear regression on problems which are highly nonlinear. Feature extraction can be performed prior to linear regression to handle nonlinearity, but to do so efficiently requires *a priori* knowledge; GP discovers the most meaningful subtrees automatically.

[2]http://labrosa.ee.columbia.edu/millionsong/pages/tasks-demos is the location of the published train-test split.

Chapter 6

Future Work

The following are potential augmentations to FlexGP:

Factor-level and Population-level Generalization: The factor-level and population-level parallelization layers of FlexGP are generalizable beyond GP. An investigation of how to recombine these techniques with other machine learning methods would be valuable. The migration included in the population-parallel layer would generalize as the transfer and exchange of intermediate solutions amongst independent learners. Factorization would preserve the diversity of models across all learners.

Model Evaluation with Subsampled Data: The fitness of each model is intended to be a relative measure, not an absolute one. As such it may be worth sacrificing some accuracy in the fitness calculation in exchange for speed. One could achieve such a tradeoff by evaluating each model not on the entire training data but on a subsample of that data, while aiming to preserve the ultimate ranking of individuals within a population.

Automatic Model Evaluation Thresholding: C++ model evaluationis not always faster than Java-based model evaluation, which will outperform C++ model evaluationon datasets of sufficiently small size. If this tradeoff were better understood FlexGP could automatically infer which evaluation method is best.

Model Evaluation on the GPU: GPU-based model evaluation is an area of active research and could provide FlexGP with a significant evaluation speedup [18].

Subtree Caching: Much work has been conducted regarding the caching of common subtrees has been used to speed up model evaluation [22][9][17]. Adding subtree caching to FlexGP would result in faster model evaluation and may aid in feature selection.

Chapter 7

Conclusion

This thesis has demonstrated FlexGP 2.0 provides a significant improvement over FlexGP 1.0 through the conjunction of multiple levels of parallelism. It addressed all of the goals first outlined in the introduction as well as a set of goals included with each level of parallelism.

- Chapter 2 presented a comparison of multi-objective optimization and operator equalization as two GP algorithms which aim to prevent overfitting by the bloating of models. We were able to obtain better accuracy v.s. time with multi-objective optimization, which also allowed the specification of multiple objective functions.

- Chapter 3 showed a significant increase in model evaluations v.s. time of C++ model evaluation over Java-based model evaluation. We demonstrated an almost 20x speedup over Java-based model evaluation when using multithreaded C++ model evaluation.

- Chapter 4 showed that population-level parallelism with migration provides both faster arrival at solutions and yields better solutions than otherwise. We demonstrate population-level parallelisation with migration provided the same MSE_{test} on a large regression problem in about an eighth of the time as was consumed without population-parallelism. Further, we demonstrate the addition of randomized migration provided a 5-point increase in MSE_{test} on a large regression problem in half the time as was consumed without migration.

- Chapter 5 provided a comprehensive demonstration of all four of FlexGP 2.0's levels of parallelism and highlighed the dramatic improvements made by FlexGP 2.0, including a 15-point increase in MSE_{test} on a large regression problem. The chapter presented evidence to establish FlexGP 2.0 as a competitive system for performing machine learning.

All four chapters have demonstrated the ability of FlexGP 2.0 to produce results of competitive accuracy on a dataset of significant size and complexity by current standards, and to do so in a reasonably short amount of time. This evidence showcases FlexGP's commitment to scaling elegantly and effectively with data size and complexity.

It is our hope FlexGP will come to play a valuable role in the application of machine learning to problems of diverse domains. We hope the contributions established in this thesis will prove useful to future research in GP, machine learning and other fields. We also hope the contributions of this thesis will spur researchers of GP to make further contributions on swift performance relative to elapsed training time, and on algorithm scalability with respect to data size and complexity. Finally, we envision the application of machine learning and data mining via FlexGP towards problems of societal relevance will make the world a better place.

Appendix A

Dataset Organization and Partitioning

This appendix describes the details of the regression problem used in the experiments for chapters 3 through 5. It also describes the manner in which data was partitioned for training and testing. It builds on Derby's description of data partitioning[6] to include modifications made for FlexGP 2.0.

A.1 Background and Problem Definition

All experiments in this thesis make use of the Million Song Dataset (MSD) [2] from the *music information retrieval* (MIR) community. The MSD contains information about one million songs, ranging from meta-data to harmonic content. In particular, experiments in this thesis are aimed at the year recognition challenge [1], where the year a song was released must be predicted from a set of 90 features extracted from the song's audio [6].

We model the year recognition challenge as a regression problem. Because the output of the models produced from GP is continuous rather than discrete, models' predictions are rounded to the nearest integer before calculation of a fitness score or a MSE[6].

[1] http://labrosa.ee.columbia.edu/millionsong/pages/tasks-demos#yearrecognition

A.2 Dataset Partitioning

There are 566,564 entries in the million song database which contain the 90 features required to be used for the year prediction challenge. The usable data is partitioned into three sections: 20% is designated as test data, 10% as training data for fusion and 70% as general training data.

Figure A-1 shows the scheme used to partition the MSD. The factor-parallel layer, which includes data factorization, is included from FlexGP 1.0. Experiments which use data factorization define a number of factorization groups, where each group can consist of one or more FlexGP nodes. When multiple nodes are included in a factorization group, they all recieve the same 35% factorized training data.

All data partitions were generated in advance of computation. This was done for simplicity, to carefully avoid the author problem described in the next section and to avoid including dataset-specific code in FlexGP. 10 sets of 70%-10%-20% data splits were generated for the experiments discussed in this thesis. Each 70%-10%-20% training split was numbered 0 through 9. From these, 10 35% factorized splits were generated for use in experiments which require data-factorization. Each 35% factorized training split was numbered 0 through 9. Validation data was not used for validation in any of the experiments.

In experiments which used 70% of the MSD as training data, each FlexGP node was given a distinct 70% training split. Similarly, in data-factorized experiments which used 35% of the MSD as training data, each FlexGP node was given a distinct 35% factorized training split.

Appendix C provides a discussion of how the fusion training data was used in conjunction with the rest of the data. To obtain the MSE_{test} and MAE_{test} used in this thesis, all models were evaluated on the 20% test split which complements the training data the model was given.

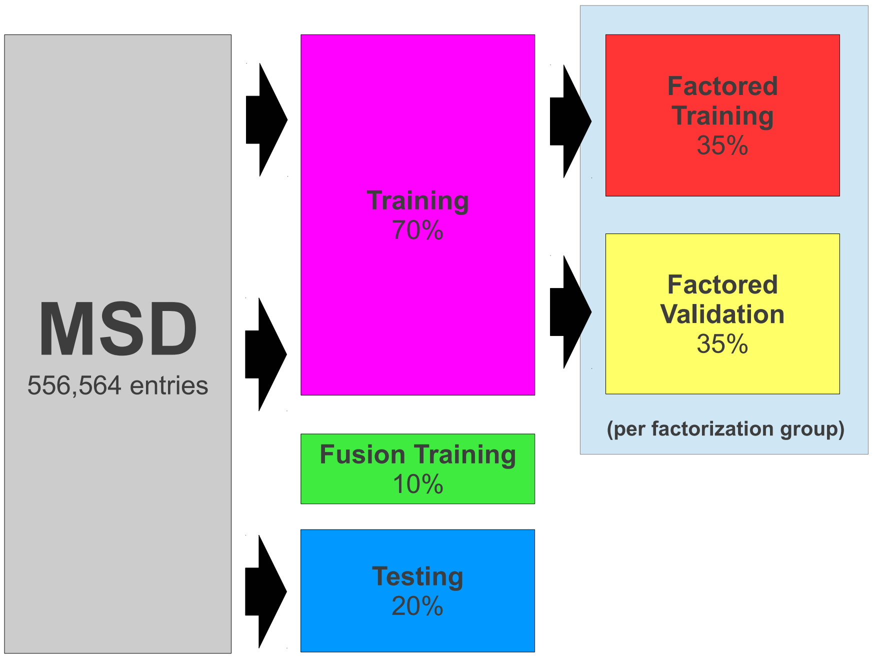


Figure A-1: A colorful depiction of the data partitioning hierarchy used in FlexGP. The original MSD is shown at left. The center column shows the training, fusion training and test datasets. The red and yellow boxes represent the factored training and validation data. If the user specifies a data factorization percentage, FlexGP will train on the factored training data in red. Otherwise, FlexGP will train on the magenta training data. The light blue box on the right indicates factorization occurs for each factorization group in FlexGP. The percentages shown are particular to the MSD problem, but the scheme is generalizable to other applications of FlexGP

A.3 Avoiding the Producer Effect in the MSD

The MSD has a complication: when splitting the data, all songs by one artist must be entirely included in one of the splits. Otherwise trained models may learn the hidden artist variable rather than the actual year prediction problem. This is known as the "producer effect" [2]. The data partitioning used here circumvent this by grouping artists' songs and randomly selecting groups until the correct percentage of the split has been approximately satisfied [6].

Appendix B

Collection and Organization of Results

This appendix details the organization of data collection for this thesis and provides a discussion of the methods used to calculate the statistics and figures in this document.

B.1 Experimental Organization

The highest level of organization in this thesis are the experiments, each of which involve a distinct mixture of computational elements and configurations. For example, chapter 4 contained a comparison of three experiments: running FlexGP on 10 nodes in isolation, running FlexGP on 10 nodes with an increased population size, and running FlexGP on 100 nodes partitioned into 10 sub-networks, each with 10 constituent islands. The list below provides a comprehensive summary of experiments:

***OE-SIMP* (Ch. 2):** Operator equalization running with a simple function set and 70% training data for 48 hours.

***MO-SIMP* (Ch. 2):** Multi-objective optimization running with a simple function set and 70% training data for 48 hours.

***OE-COMP* (Ch. 2):** Operator equalization running with a complex function set and 70% training data for 48 hours.

***MO-COMP* (Ch. 2):** Multi-objective optimization running with a complex function set and 70% training data for 48 hours.

***Java1* (Ch. 3):** Java model evaluation on 70% training data for 3 hours.

***C++1* (Ch. 3):** 1-thread C++ model evaluation on 70% training data for 3 hours.

***C++4* (Ch. 3):** C++ 4-thread model evaluation on 70% training data for 3 hours.

***1K-SOLE* (Ch. 4):** 1 node running with 70% training data for 6hrs.

***10K-SOLE* (Ch. 4):** 1 node running with a 10k population and 70% training data for 24hrs.

***1K-PAR10* (Ch. 4):** a sub-network of 10 nodes with migration, running with 70% training data for 3hrs.

***1.0* (Ch. 5):** 8 independent nodes running operator equalization with Java model evaluation, each given a different 35% factorization of the training data, for 24 hours.

***2.0-NO-POP* (Ch. 5):** 8 independent nodes running multi-objective optimization with C++ 4-thread model evaluation, each given a different 35% factorization of the training data, for 6 hours.

***2.0-POP* (Ch. 5):** 8 sub-networks with 10 nodes per sub-network, where each sub-network of 10 nodes is given a different 35% factorization of the training data, running multi-objective optimization for 3 hours.

For all experiments, the default configuration is assumed for any values which were not stated above. The defaults are:

Evaluation: C++ 4-thread

Algorithm: multi-objective optimization

Function set: simple

Population: 1k models

Duration: 3 hours

Sub-network: disabled, no migration

Data: 70% training data.

The 3rd experiment for Chapter 3 and the 1st experiment for Chapter 4 are both subsets of the 2nd experiment for Chapter 2. This reduces the amount of data which needed to be collected by a small amount.

Each experiment is repeated 10 times for statistical validity. Each repetition is referred to here as an "iteration."

B.2 Organization of Results

Each node saves results locally in two files. The first file, named "models.txt," simply contains the best model for each generation, with each model occupying one line of the file. The second file, "evogpj-log.json," contains one JSON structure per line for each generation. The information contained in the JSON structures is detailed in table B.1.

Key	Description
timestamp	The elapsed time since FlexGP started.
generation	Generation number, starting at 0.
stats	The min/max/mean/stddev of fitness in the population.
fitnessEvaluations	The number of model evaluations made.
bestModel	The model chosen as best. Includes which method was used.
paretoFront	The entire Pareto front if running multi-objective optimization.
paretoFrontSize	The number of models in the Pareto front.
numImmigrants	The number of models incorporated into the population.
numEmigrants	The number of models emigrated to other nodes.
equalizer	If running operator equalization, current equalizer parameters.

Table B.1: The contents of each JSON log.

When an experiment is complete, evogpj-log.json and models.txt files are retrieved from each node and stored together in one folder. The naming convention for both files is "¡filename¿-¡nodeID¿.¡extension¿". For example, the folder containing results from a single 10-island experiment would contain 10 JSON and 10 text files, labeled "evogpj-log-0.json" through "evogpj-log-9.json" and "models-0.txt" through "models-9.txt", where the numbers appended to the name are used to distinguish between nodes.

The distinct parameters of each experiment are described by the folder name. The parameters of each experiment are described here.

Number of nodes or islands (*numNodes*): the character "n" or "i" followed by three digits which indicate the number of nodes or islands in the experiment.

Algorithm used (*alg*): three characters indicating the algorithm used, with "moo" indicating multi-objective optimization and "deq" indicating operator equalization.

Model evaluation method (*eval*): the character "j" for Java or "c" for C++, followed by a digit indicating the number of threads used (always 1 for Java).

Training split (*train*:) the character "d" followed by two digits indicating which of the 10 splits was used. The first character is an "A" if this experiment is part of 10 experiments which covered all training splits.

Factorization (*factor*): the character "f" followed by three digits. The first indicates the function set used. The last two indicate which training factorization was used. If no factorization was used the last two digits will appear as "XX", but the first will still indicate the function set used.

Migration size (*m*): the character "m" followed by four digits indicating the migration size. If no migration was used this will appear as "mXXXX".

Population size (*n*): the character "p" followed by five digits indicating the population size.

Duration (*t*): the character "t" followed by two digits indicating the duration of the experiment in hours.

Repetition (rep): the character "r" followed by four digits which serve as a unique identifier used to distinguish amongst repetitions of an experiment.

Combining the parameters in the order they are listed, the full format of the folder names is as follows:

$$numNodes\_alg\_eval\_train\_factor\_m\_n\_t\_rep$$

Consider the following example:

$$n001\_deq\_j1\_dA3\_f0XX\_mXXXX\_p01000\_t06\_r0003$$

This indicates the contained results are from a 1-node run with operator equalization and Java model evaluation, operating on a distinct 70% training split (#3), using function set 0 with no data factorization and no migration, a population of 1000 and a duration of 6 hours (fourth iteration).

B.3 Analysis of Results

Each JSON log has alongside a file named "performance-train-X.csv", where the X indicates the node ID for uniqueness. This file contains six columns: elapsed time, elapsed model evaluations, MSE_train, MAE_train, MSE_test and MAE_test.

Directories beginning with "c" instead of "n" or "i" contain results from performing fusion. Each such directory contains a file *mse_test_fusion_vs_thresholdMetric_N.csv*, where $thresholdMetric$ can either be "time" or "fiteval" and N is the number of models considered for fusion during each frame. The file contains three columns: thresholding metric, MSE_test and MAE_test. See Appendix C for more information on model fusion and thresholding metrics.

Appendix C

Calculation of Fused Performance

This section defines the fusion process used by FlexGP. This fusion process is used to generate each point in the figures presented in Chapters 4 and 5.

C.1 Motivation

A fully parallelized FlexGP with all levels of parallelism enabled can churn out thousands of models in a matter of minutes. FlexGP reduces this massive influx of information to a fused regression ensemble or "meta-model" which combines the predictions of the models deemed the most useful and boosts accuracy beyond that which is possible from sole models.

C.2 Background

A regression ensemble is a grouping of regression models such as those returned from FlexGP. Regression Ensemble Fusion (referred to here as simply "fusion") is the act of learning a method of combining the predictions of multiple regression models in the ensemble. The result of fusion is a meta-model which uses its constituent models to make predictions of greater accuracy than would be possible by a single model.

FlexGP uses a regression ensemble fusion technique known as Adaptive Regression Mixing (ARM) [23]. Previous work comparing a variety of regression ensemble fusion has found ARM to be a particularly effective fusion method[19].

C.3 Fusion Process

The fusion process used in the experiments for Chapters 4 and 5 has three primary steps:

1. **Best N Selection:** the best N models are identified from a larger group. The experiments in this thesis used an N of 50.

2. **Regression Ensemble Fusion:** the best N models are fused via ARM to form the meta-model which is used to make predictions.

3. **Ensemble Evaluation:** the meta-model is evaluated on the test data to obtain a post-fusion MSE_{test} and MAE_{test}.

In this thesis the training data used to perform fusion was all non-test data. This means the data each GP learner was trained on was included in the data used for fusion training. We argue the MSD is large enough to make this permissible without fear of overfitting. To be explicit: for learners which were trained with 70% of the MSD, the ultimate fusion training set consisted of the 70% training segment plus the 10% set aside for fusion training. For learners which were trained with a 35% factorization of the MSD, the ultimate fusion training set consisted of the 35% factorized training segment, the additional 35% set aside as validation data and unused in this thesis, and the final 10% set aside for fusion training.

C.4 Thresholding Metrics

Two metrics were used for determining the thresholds by which to filter models before best N selection:

Time: Any models which appeared before time t are considered.

Model Evaluations: Any models which have a number of elapsed fitness evaluations less than N are considered.

Plots whose x-axis is labeled "time" used time thresholding. Similarly, plots whose x-axis is labeled "model evaluations" used model evaluation thresholding.

Appendix D

GP Parameters

This appendix describes the parameters and implementation used to configure GP in FlexGP 2.0. Slight modifications are applied to Derby's specification of the parameters used in FlexGP 1.0 [6].

All experiments discussed in this thesis were configured with the same parameters as described by Koza [11], with the following differences. The population size is set to 1000 models. Initialization was performed using Koza's ramped-half-and-half algorithm. The mutation rate is 0.5 and the crossover rate is 0.5. Nodes were selected uniformly at random for crossover. The max depth is set to 5 for initialization, afterwards tree depth is limited to 32. For both operator equalization and multi-objective optimization, tournament selection was configured with a tournament size of 10. For operator equalization experiments the equalizer[15] was configured with a bin width of 5. During model evaluation, models' predictions are transformed with Vladislavleva's approximate linear scaling [21], where the output variable is scaled to span the range $[0, 1]$ so GP can focus on learning the shape of the relation described by the data [6].

Appendix E

Cloud Infrastructure

FlexGP is designed to be a cloud-backed system. All experiments discussed in this thesis were conducted on a private cloud maintained by MIT CSAIL. The cloud uses the Openstack cloud management framework [1], a free and open source software for maintenance of public and private clouds. The Openstack interface is highly similar to the API used by Amazon's Elastic Compute Cloud (EC2)[2] service [6].

All experiments ran on 64-bit virtual machines with 4 virtualized CPUs, 4GB of RAM and 20GB of disk storage. The same virtual machine configuration was used for all experiments to eliminate the parameters of the virtual machine used for each experiment as a consideration when analyzing results. Each virtual machine was configured with Ubuntu 12.04. The JVM included on each virtual machine was OpenJDK 6.[3]

[1]http://www.openstack.org
[2]https://aws.amazon.com/ec2
[3]http://openjdk.java.net/

Bibliography

[1] Ignacio Arnaldo, Ivn Contreras, David Milln-Ruiz, J.Ignacio Hidalgo, and Natalio Krasnogor. Matching island topologies to problem structure in parallel evolutionary algorithms. *Soft Computing*, 17(7):1209–1225, 2013.

[2] T. Bertin-Mahieux, D.P.W. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference, October 24-28, 2011, Miami, Florida*, pages 591–596. University of Miami, 2011.

[3] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. Finding knees in multi-objective optimization. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 722–731. Springer, 2004.

[4] Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Springer, 2000.

[5] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley Interscience Series in Systems and Optimization. Wiley, 2001.

[6] Owen C Derby. FlexGP: a scalable system for factored learning in the cloud. Master's thesis, Masachusetts Institute of Technology, 2013.

[7] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[8] Yaochu Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing*, 9(1):3–12, 2005.

[9] Maarten Keijzer. Alternatives in subtree caching for genetic programming. In *Genetic Programming*, pages 328–337. Springer, 2004.

[10] Maarten Keijzer and James Foster. Crossover bias in genetic programming. In *Genetic Programming*, pages 33–44. Springer, 2007.

[11] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[12] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[13] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May OReilly. Flex-GP: genetic programming on the cloud. In Cecilia Di Chio, Alexandros Agapitos, Stefano Cagnoni, Carlos Cotta, Francisco Fernndez de Vega, Gianni A. Di Caro, Rolf Drechsler, Anik Ekrt, Anna I. Esparcia-Alczar, Muddassar Farooq, William B. Langdon, Juan J. Merelo-Guervs, Mike Preuss, Hendrik Richter, Sara Silva, Anabela Simes, Giovanni Squillero, Ernesto Tarantino, Andrea G. B. Tettamanzi, Julian Togelius, Neil Urquhart, A. ima Uyar, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation*, number 7248 in Lecture Notes in Computer Science, pages 477–486. Springer Berlin Heidelberg, January 2012.

[14] S. Silva and S. Dignum. Extending operator equalisation: Fitness based self adaptive length distribution for bloat free GP. *Genetic Programming*, pages 159–170, 2009.

[15] Sara Silva. Handling bloat in GP. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 1481–1508, New York, NY, USA, 2011. ACM.

[16] Zbigniew Skolicki and Kenneth De Jong. The influence of migration sizes and intervals on island models. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1295–1302. ACM, 2005.

[17] Guido Smits, Arthur Kordon, Katherine Vladislavleva, Elsa Jordaan, and Mark Kotanchek. Variable selection in industrial datasets using pareto genetic programming. In *Genetic Programming Theory and Practice III*, pages 79–92. Springer US, 2006.

[18] JazzAlyxzander Turner-Baggs and Malcolm I. Heywood. On gpu based fitness evaluation with decoupled training partition cardinality. In AnnaI. Esparcia-Alczar, editor, *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 489–498. Springer Berlin Heidelberg, 2013.

[19] Kalyan Veeramachaneni, Owen Derby, Dylan Sherry, and Una-May OReilly. Learning regression ensembles with genetic programming at scale. in press, 2013.

[20] C. Vladislavleva and G. Smits. Symbolic regression via genetic programming. *Final Thesis for Dow Benelux BV*, 2005.

[21] E.Y. Vladislavleva. *Model-based problem solving through symbolic regression via pareto genetic programming*. PhD thesis, CentER, Tilburg University, 2008.

[22] Phillip Wong and Mengjie Zhang. Scheme: Caching subtrees in genetic programming. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 2678–2685. IEEE, 2008.

[23] Y. Yang. Adaptive regression by mixing. *Journal of the American Statistical Association*, 96(454):574–588, 2001.