Predicting Parallelization of Sequential Programs Using Supervised Learning

Daniel Fried*, Zhen Li†‡, Ali Jannesari†‡, and Felix Wolf†‡
*University of Arizona, Tucson, Arizona, USA
†German Research School for Simulation Sciences, Aachen, Germany
‡RWTH Aachen University, Aachen, Germany
dfried@cs.arizona.edu,{z.li,a.jannesari,f.wolf}@grs-sim.de

***Abstract*—We investigate an automatic method for classifying which regions of sequential programs could be parallelized, using dynamic features of the code collected at runtime. We train a supervised learning algorithm on versions of the NAS Parallel Benchmark (NPB) [14] code hand-annotated with OpenMP parallelization directives in order to approximate the parallelization that might be produced by a human expert. A model comparison shows that support vector machines and decision trees have comparable performance on this classification problem, but boosting using AdaBoost is able to increase the performance of the decision trees. We further analyze the relative importance of the collected program features and demonstrate that within-loop instruction counts provide the greatest contribution to decision tree error reduction, with dependency graph features of secondary importance.**

I. Introduction

The growing prevalence of multi-core computing systems has led to an increased need for parallelized code capable of fully using the processing power of modern systems. Parallel programming models such as OpenMP make writing parallel code less difficult and error-prone by managing the low-level details of thread handling and synchronization. OpenMP in particular provides a way to incrementally parallelize existing sequential code by adding annotations to regions of code that specify the code should be run in separate threads and control how the work should be distributed and results synchronized. However, a fully-automatic system for auto-parallelization must choose which regions of sequential code to annotate with these parallelization directives and select the type of parallelization directive to apply to each region.

To select which regions of code to parallelize, static analysis (examination of the source code without running it) and dynamic analysis (running the source code and monitoring control and data flow) can be used. Static analysis can determine which regions of code are guaranteed to be parallelizable. However, static analysis can be overly conservative in identifying which loops are parallelizable since potential data dependencies identified by a static analysis of the program may actually never occur when the program is run. We investigate using dynamic analysis to determine whether to parallelize a given region of code. Our system extracts features from data dependency graphs generated dynamically as the program is run to capture the observed dependencies between instructions.

In this work we address the problem of identifying loops that are efficiently parallelizable, meaning that parallelization will result in a runtime speedup without a prohibitive amount of memory or thread synchronization overhead. Since the general agreement is that expert programmers are most capable of determining which loops are efficiently parallelizable, we use OpenMP parallelization annotations on standard benchmarks as groundtruth for classifier training. This allows us to train and evaluate a model against how a programmer might parallelize existing sequential code.

The paper is structured as follows. In Section II, we overview related work on using machine learning techniques to model program parallelization and performance using static and dynamic features. We outline our method of classification in Section III, describing features used in the learning algorithms, the construction of training and evaluation data sets, and experimental methodology. Section IV gives an overview of the classification models used. Parameter setting and classification results are discussed in Section V. Section VI analyzes the relative importance of each extracted feature in classification error minimization. We present the classification results in Section VII, and outline prospects for further investigation in Section VIII.

II. Related Work

Previous work has applied machine learning algorithms to the tasks of code optimization selection, identification of parallelizable regions of sequential code, and the automatic parallelization of such regions. These approaches include supervised methods, which train a model using information about whether parallelization is suitable for given regions of sequential code, as well as unsupervised methods, which attempt to infer parallelization effects without using labelled training data.

Tournavitis et al [16] perform supervised classification of loops in sequential code as parallelizable using a support vector machine and provide a method for automatically parallelizing loops with OpenMP pragma statements. The Centrifuge system of Demme et al [5] takes an unsupervised approach, building graphs of program control and data flow from static and dynamic analyses of the code, with graph vertices at the level of blocks of code. Programs are clustered into groups using similarity between these flow graphs. These clusters are evaluated based on the responsiveness of programs within each cluster to optimizations. Srivastava et al [15] use a neural network to predict how changing loop scheduling policy affects load balancing in an already parallelized application. Although not directly predicting parallelization, Park et al [11] use an SVM with a kernel function operating on the static control flow graph of a program to predict the improvements gained from compiler optimizations, and demonstrate that this graph-based classification improves performance.

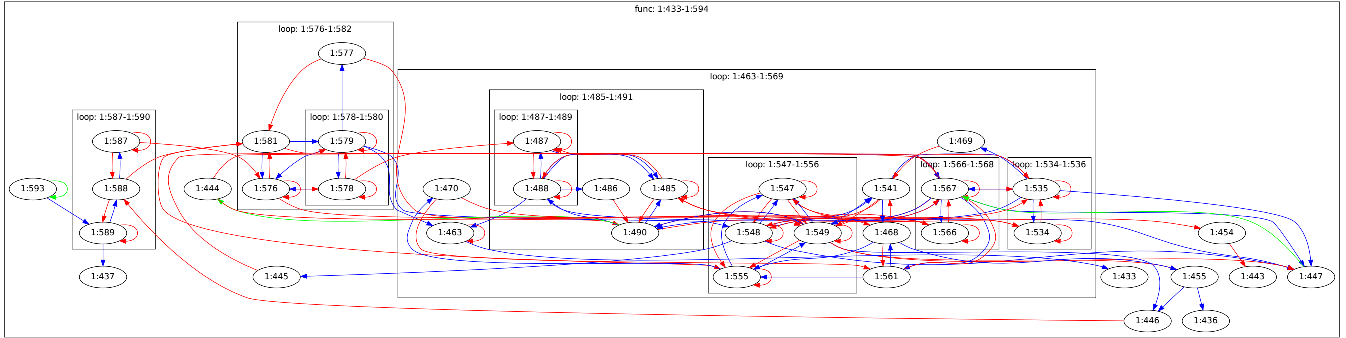


Fig. 1: Part of a data dependency graph for a function from the sequential *CG* (conjugate gradient) NAS Parallel Benchmark. Vertices represent the line numbers of individual instructions. RAW dependencies are blue, WAR dependencies are red, and WAW dependencies are green. Boxes correspond to loops within the function (possibly nested).

.

Our method, like Park's and Tournavitis's, is supervised and, like Demme's, uses data dependency graph features collected dynamically from running the program. Unlike these methods, our system is trained using a set of existing parallelizations of sequential code, creating a model of human parallelization decisions. Additionally, we extend the work of the supervised parallelization prediction methods through model comparison, the application of boosting to improve classifier performance, and analysis of relative feature importance in error minimization.

III. Loop Classification

A. Feature Extraction

We use the DiscoPoP (*Discovery of Potential Parallelism*) tool [10] to extract dynamic profiling and instruction dependency data from instrumented versions of sequential programs. First, sequential programs are compiled to the Low Level Virtual Machine (LLVM)'s intermediate representation (IR) [9]. Then, DiscoPoP instruments the target program and executes it. Control-flow information and data-dependence relationships are obtained in this phase. Next, analysis of potential parallelism is performed based on the information produced in the first phase, and the final result of the analysis is written into a file on the disk. DiscoPoP provides a number of features about the program execution for each block of code executed. A subset of these features (outlined below and listed in Table I) are extracted from the DiscoPoP output for each loop in the program and used as input to the classification algorithms.

`N_Inst` and `exec_times` provide a direct measurement of the execution cost of a loop. They are the number of IR instructions within the loop body and the number of times the loop is executed, respectively. Critical path length, `CFL`, measures the length of the maximal length sequence of dependent instructions within the loop [17]. Estimated speedup, `ESP`, is a heuristic calculated using the maximum breadth and critical path length of the dependency graph and Amdahl's Law [1]. This feature estimates the maximal decrease in run time for a given loop if it is parallelized.

To allow classification based on data dependencies between a loop and the code containing it, we extract features from each program's dependency graph. Fig. 1 shows a portion of a data dependency graph produced for a single function from *CG*, one program from the NAS Parallel Benchmark. Vertices represent individual instructions. Edges between instructions are the data dependencies detected by DiscoPoP. Our graph features are the counts of each type of dependency (RAW, WAR, and WAW) leaving each loop, entering each loop, and contained within each loop. These features extracted from the dependency graph for a loop (`outgoing_dep`, `incoming_dep`, and `internal_dep`, respectively) are further subdivided into counts for each of three different types of data dependency: Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW). A fourth type of data dependency, Read After Read, is not considered, since dependencies involving only reads do not affect parallelization.

TABLE I: Dynamic features used for loop parallelization classification.

feature name	description
N_Inst	Number of instructions within the loop
exec_times	Total number of times the loop is executed
CFL	Critical path length
ESP	Estimated speedup
incoming_dep	Dependency count of external instructions on loop instructions
internal_dep	Dependency count between loop instructions
outgoing_dep	Dependency count of loop instructions on external instructions

B. Comparing Sequential and Parallel Code

We use C implementations of the NAS Parallel Benchmarks (NPB) [2] provided in the SNU NPB Suite [14]. This test suite includes a set of programs implementing numerical algorithms that have been parallelized by hand using the OpenMP framework. However, the parallel and sequential versions of the benchmarks provided are not directly comparable either in source code or single-threaded performance due to the restructuring and refactoring of the parallel codes, similar to the restructuring reported by Tournavitis et al [16]. To facilitate direct comparison of the parallelized code against sequential versions, we produce sequential code from the OpenMP versions of the benchmarks by stripping out all OpenMP pragma statements and replacing calls to OpenMP's thread counting and thread identification functions with con-

stants appropriate for running in a single thread. This produces two versions of the benchmark code that are identical except for the presence of the OpenMP parallelization directives. Stripping out all pragma statements in this manner does not affect program correctness: all sequential programs produce the same numerical output as their parallelized counterparts when validated on the NPB tests.

C. Classification Methodology

In the supervised learning process, we divide the loops into training and testing sets. Each loop in the training set is labelled according to whether or not it is parallelized with an OpenMP pragma in the parallelized version of the benchmarks. All benchmarks are then instrumented and run with input size S [2], the smallest input size defined for the benchmarks. The dynamic features described above are recorded for each for loop in the program and used as input to one of the classification models (support vector machine, decision tree, or boosted ensemble) described below. These classification models learn an association between the features of each loop and whether or not it is designated as parallelized (using an OpenMP pragma) in the parallel version of the benchmarks.

To evaluate the ability of the classifier to predict whether unseen code can be parallelized, loops from the testing set are shown to the classifier, which predicts whether each individual loop should be parallelized using OpenMP. These predictions are then compared against the loops in the parallel version of the code.

We use a modified form of Leave-One-Out cross validation (LOOCV) to evaluate the performance of a classifier and estimate generalization error, or the tendency of the classifier to be biased toward the training data and lose accuracy on predictions of unseen data. Loops within a single function from the benchmark are withheld as the testing set and all other loops are used as the training set. Scores are recorded for this split of the data. This process is repeated with loops from each entire function in turn withheld as the testing set. Scores are then averaged across all partitions.

LOOCV's large training sets and high number of folds allow us to decrease variance in our estimates of the generalization error by running as many partitions as possible. We separate the loops based on their containing functions since loops can be nested inside of one another within the same function, and could therefore have identical features, such as number of incoming dependencies. Separating based on containing function therefore decreases the chance that a loop present in the testing set could exactly match one present in the training set.

IV. Classification Models

We compare the performance of three different classification models: support vector machine (SVM), decision tree, and an ensemble of decision trees boosted with AdaBoost. The `scikit-learn` [12] Python library is used for implementation of all classifiers.

A. Support Vector Machine

We use a soft margin SVM [4] with a Gaussian Radial Basis Function (RBF) kernel, which maps data non-linearly based on distance from the origin in the feature space. Since the RBF's performance is dependent on the mean and variance of its features, we normalize all data by centering each feature around 0 and scaling so that the standard deviation is 1.

The SVM has two hyperparameters: the soft margin multiplier, C, and a scaling constant, γ, in the RBF kernel function:

$$K(\mathbf{x_1}, \mathbf{x_2}) = \exp(-\gamma ||\mathbf{x_1} - \mathbf{x_2}||_2^2)$$

where $\mathbf{x_1}$ and $\mathbf{x_2}$ are feature vectors for two source code loops. We set C and γ using cross-validation on a training set, and evaluate performance on a held-out evaluation set (see Sec. V for details).

B. Decision Tree

SVMs learn a classification boundary in high dimensional space that can be difficult to visualize. Decision trees, on the other hand, produce an interpretable set of rules used for classification, and although often less accurate than an SVM, are less sensitive to error caused by irrelevant features [8]. This feature robustness motivates our comparison of decision trees to the SVM, since it is unknown if all the dynamic program features extracted are relevant to the classification problem.

We use the CART tree construction algorithm with Gini impurity function [3]. The hyperparameters for the tree classifier are the maximum depth of the tree and the maximum number of features to consider in any node when looking for the best split. Sec. V contains details of how the optimal values of these hyperparameters are determined.

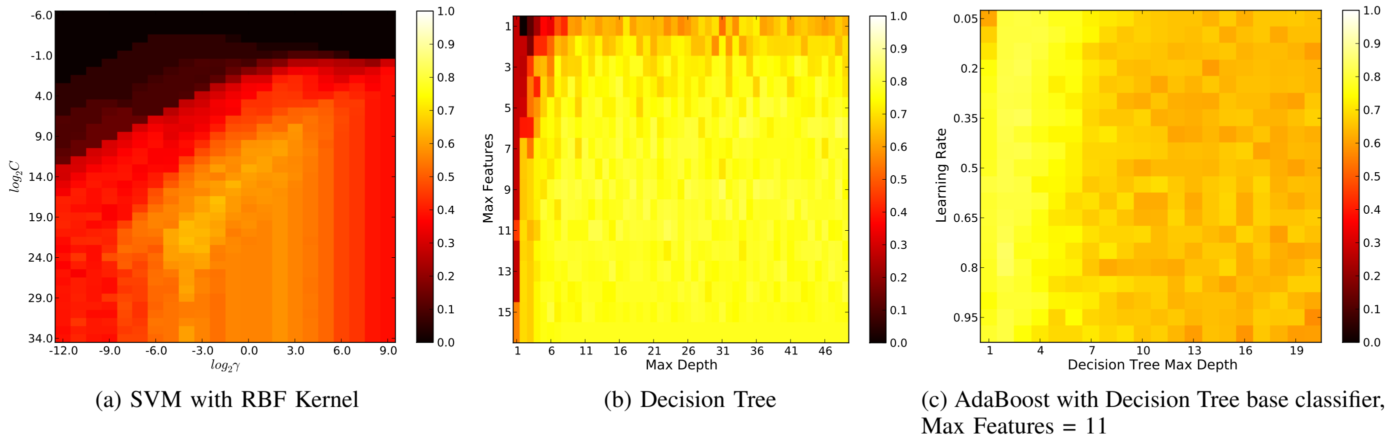
C. Boosted Decision Trees

Boosting attempts to increase classification performance by combining the decisions of many weak classifiers (classifiers not closely correlated with the target class). We use the AdaBoost algorithm [7] with a decision tree as the base classifier. AdaBoost iteratively trains T classifiers on the training data. At each stage, it weights the importance of each sample according to its misclassification by previous iterations: samples that are misclassified receive higher weights, biasing future classifier iterations toward classifying these harder examples correctly. Additionally, each iterated classifier is assigned a weight which is large if the classifier's error rate on the training set is small, and vice versa. The final result is an ensemble of instances of the weak classifier, each with its own weight. During testing, the ensemble classifies unseen instances using a weighted combination of the votes from each of these weak classifiers. We refer to [6] for a thorough description of the algorithm.

Hyperparameters for AdaBoost are the number of iterated classifiers, T; a learning rate $0 < r \leq 1$ which controls to what degree sample weights are updated at each iteration; and the hyperparameters of the base classifier (maximum depth and maximum number of features for our base decision tree classifier). We set these using cross-validation, see Sec V.

V. Parameter Setting and Evaluation

To maximize classifier performance, it is necessary to tune the hyperparameters for each of the classification models described above. We split the available data into a training set and a held-out evaluation set. Hyperparameter setting is



(a) SVM with RBF Kernel

(b) Decision Tree

(c) AdaBoost with Decision Tree base classifier, Max Features = 11

Fig. 2: Classifier F1 score as a function of hyperparameters, cross-validated using function-level LOOCV across the training data set.

performed through cross-validation on the training set, and classifiers are scored on the evaluation set. The training set consists of 630 loops, and the evaluation set of the remaining 160 loops, for a split of roughly 80% / 20%. 126 loops in the training set have pragma statements, compared to 21 in the evaluation set. Loops are divided so that no loop in the training set is in the same containing function as a loop in the evaluation set for the reasons described in Sec III-C.

To set the parameters for a given classifier, we perform a grid search through the Cartesian product of possible parameters for that classifier, within a given range and step size, delta (Table II). Parameters are chosen that maximize average F1 score averaged across all LOOCV folds within the training set. Once parameters are tuned, evaluation is performed by running the classifier on the held-out evaluation set. Using each scoring metric, labels produced by the classifier are compared with the presence or absence of a pragma on each loop in the parallel version of the NPB code.

Fig. 2 shows average F1 score as a function of parameter values for each classifier type. The heatmap in Fig. 2a shows that the SVM has a relatively narrow range of high performance and that increasing or decreasing C and γ will decrease F1 score. On the other hand, Fig. 2b shows that increasing the depth of the decision tree beyond about 4 and increasing the number of features considered at each split of the tree only increases the cross-validated F1 score slightly, which indicates that the decision tree is fitting the data completely for relatively small values of these parameters. In Fig. 2c, we see that F1 score does not vary much with the learning rate parameter, and is more greatly affected by the max depth limit of the base decision tree classifier. Weaker (depth-limited) decision trees produce higher F1 scores when using the AdaBoost method.

VI. Feature Analysis and Selection

We can compute feature importance in a decision tree by calculating a weighted sum of the reduction in the impurity criterion that each feature provides across all nodes for which it is the splitting point [8]. The weight on each node in this linear combination is the number of training instances present within the node. The importance for a feature l is calculated as

$$Importance(l) = \sum_{m=1}^{M} \left(w_m \sum_{t=1}^{M_J - 1} i_t^2 \cdot I(v(t) = l) \right)$$

where m indexes over the M trees in the ensemble, w_m is the weight of the tree within the ensemble, t indexes over the internal nodes inside each tree, i_t^2 is the square of the loss in Gini impurity for this node, and $I(v(t) = l)$ is an indicator function that is 1 if node t splits on feature l, and 0 otherwise.

Intuitively, features that receive higher importance scores were used to split larger number of training instances and resulted in larger impurity reductions in these splits. Importance for a single tree may not be informative if the tree is a weak classifier, but if we have an ensemble of trees (as we do in

TABLE II: Parameter ranges used in grid search for each classifier and optimal parameter value as determined by LOOCV on the training set.

classifier	parameter	range	delta	optimal value
SVM	C	$[2^{-6}, 2^{35}]$	2 (exp)	2^{22}
	γ	$[2^{-12}, 2^{10}]$	2 (exp)	2^{-4}
Decision Tree	Max Features	$[1, 16]$	1	11
	Max Depth	$[1, 50]$	1	13
AdaBoost DT	Learning Rate	$[0.05, 1]$	0.05	0.25
	DT Max Depth	$[1, 20]$	1	3

TABLE III: Feature importance in decision trees, calculated using weighted error reduction in an AdaBoost ensemble of trees.

feature	importance	feature	importance
N_Inst	0.12	internal_dep	0.06
internal_dep_RAW	0.09	incoming_dep_WAR	0.06
outgoing_dep_RAW	0.08	outgoing_dep_WAR	0.06
incoming_dep	0.08	CFL	0.05
incoming_dep_RAW	0.08	internal_dep_WAW	0.02
internal_dep_WAR	0.08	ESP	0.02
outgoing_dep	0.07	outgoing_dep_WAW	0.02
exec_times	0.07	incoming_dep_WAW	0.02

AdaBoost), we can average these feature importances across all the trees in the ensemble, producing more robust feature scores. Table III shows relative feature importance calculated in this manner on the loops in the training set. In Sec. VII we use these importance scores to perform feature selection, and evaluate the effect on classification performance on the held-out evaluation set.

VII. Results

Classifier results on the held-out evaluation set are shown in Table IV. Results are shown for two sets of features: all features listed in Table III, and those top features with an importance score of 0.08 or greater (calculated using the error reduction method of Sec VI). Also shown for comparison are the results achieved by a baseline classifier that simply predicts every loop as ”not parallelizable”.

We report accuracy as a basic measure of classifier effectiveness. However, since the majority of loops (643 out of 790) in our benchmarks do not have pragmas, a classifier could achieve a high accuracy by simply marking every loop it sees as not parallelizable, as the baseline classifier does. Therefore, we score classifiers according to three additional metrics, to more accurately portray performance despite class imbalance: precision, the proportion of the classifier's positive predictions that were correct; recall, the proportion of the parallelizable loops the classifier recognized; and F1-measure, the harmonic mean of precision and recall.

The scores under *identifying pragma presence* assess each classifier's ability to correctly identify loops with pragmas as parallelizable (i.e. a true positive is a loop with pragma that the classifier identified as parallelizable, while a false positive is a loop without pragma that the classifier identified as parallelizable). The scores in the *identifying pragma absence* set of columns assess each classifier's ability to correctly identify loops without pragmas (i.e. a true positive is a loop without a pragma that the classifier did not identify as parallelizable, while a false positive is a loop with a pragma that the classifier did not identify as parallelizable). Accuracy summarizes overall success for both classes as the number of correct identifications out of the total number of testing instances.

Because of the class imbalance, the baseline classifier is able to achieve a high accuracy simply by predicting every testing instance as *no pragma*. When using all features, the SVM and Decision Tree classifiers achieve nearly identical scores. Boosting (with AdaBoost) significantly improves the precision of the decision tree resulting in a higher F1 score as well as increased overall accuracy. When using only the most important features, as ranked by importance in the boosted ensemble of decision trees, performance of the SVM and decision tree both increase in accuracy and F1 score, although performance of the AdaBoost ensemble decreases slightly. The lower performance of the SVM and decision tree when using all features could indicate overfitting to less-relevant features. However, as the evaluation scores of AdaBoost are higher when using all features, overfitting seems to have less of an effect on the performance of the boosted trees.

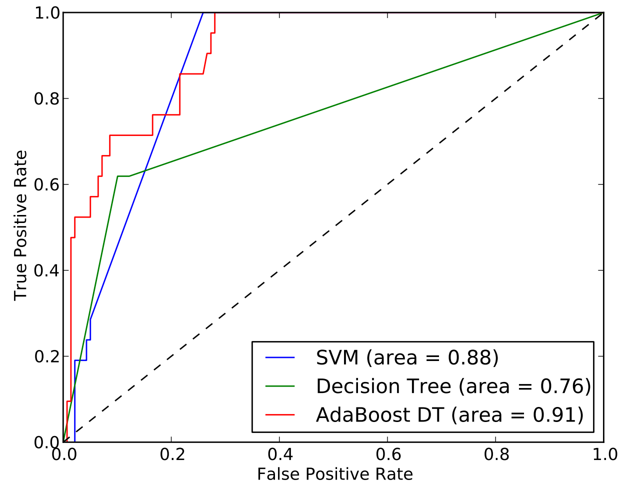


Fig. 3: Receiver operating characteristic curves showing trade-off between true positive rate (aka recall, $TP/(TP+FN)$) and false positive rate ($FP/(FP+TN)$) as a score cutoff threshold varies for each classifier.

To evaluate the trade-off between true positive and false positive rates in each type of classifier, we also evaluate prediction performance using thresholded score classification. Each type of classifier can be modified to output a score indicating the confidence that a given loop has a pragma, instead of simply functioning as a binary classifier. For SVM, this score is calculated using Platt scaling [13], with five-fold cross validation to prevent overfitting. For the decision tree, it is simply the proportion of training instances with pragmas in the decision tree leaf reached by the data point. The score in the boosted decision tree is the weighted average of the scores of all trees in the ensemble. Varying a cutoff threshold

TABLE IV: Classification scores on the held-out evaluation set, separated by loops with pragmas and loops without pragmas. All Features indicates classifier performance using all extracted features. Top Features indicates classifier performance using features with importance ≥ 0.08 (see Sec. VI).

	identifying pragma presence			identifying pragma absence			
classifier	**precision**	**recall**	**f1-measure**	**precision**	**recall**	**f1-measure**	**accuracy**
Baseline	0.00	0.00	0.00	**1.00**	**1.00**	**1.00**	0.81
SVM - All Features	0.46	0.62	0.53	0.94	0.89	0.92	0.85
Decision Tree - All Features	0.45	0.62	0.52	0.94	0.88	0.91	0.85
AdaBoost DT - All Features	**0.72**	0.62	**0.67**	0.94	0.96	0.95	**0.92**
SVM - Top Features	0.53	**0.81**	0.64	0.97	0.89	0.93	0.88
Decision Tree - Top Features	0.63	0.57	0.60	0.94	0.95	0.94	0.90
AdaBoost DT - Top Features	0.71	0.48	0.57	0.92	0.97	0.95	0.91

on these scores for each classifier gives a trade-off between true positive rate and false positive rate, producing receiver-operating characteristic (ROC) curves (Fig. 3). The area under these curves is another measure of the quality of the classifier: 1.0 is perfect, and 0.5 is no better than chance. The AdaBoost decision tree has the highest area under its ROC curve, at 0.91.

VIII. Conclusion and Future Work

This work investigates the use of dynamically-collected program features for the prediction of loop parallelization, modelled after human-produced parallelization decisions. We demonstrate an improvement in classifier performance using boosting of weak decision tree classifiers and analyze the importance of each program feature in the minimization of error in these decision trees. We observe that number of instructions within a loop is the feature that best minimizes classification error in decision trees. This may be due to a tendency for programmers to parallelize loops that perform a large amount of work. Extra computation required for thread management could make parallelization of simple loops not worth the extra overhead. Additionally, we find that RAW dependency counts have the highest relative feature importance of the three types of data dependencies. Furthermore, when evaluating the classifiers on only the most important features according to the decision-tree criterion, we see increased performance by both the SVM and decision tree, although they do not outperform the boosted tree classifiers when trained on all features.

We currently train and evaluate our classifier using a human-produced gold standard, in the form of the OpenMP-parallelized versions of the NAS Parallel Benchmarks. However, it may be the case that some loops that are not labelled with pragma statements in this benchmark are effectively parallelizable, or that some loops that have been parallelized do not produce an overall speedup in program execution. In future work, we could acquire labels using actual performance comparisons of sequential and parallelized code, rather than relying on the presence or absence of these OpenMP pragmas in the benchmarks as an indicator of whether the loop is parallelizable. It would also be possible to profile each individual loop in the program and calculate the speedup associated with each. This numeric data could be used as the target of a regression classifier, using for example support vector regression or (boosted) decision tree regression, with the goal of predicting the speed up resulting from parallelization of an individual loop. Such a regression algorithm could use the same dynamic runtime behavior and dependency features used in this classification problem.

Finally, there is the potential to include more features from profiling or statically analyzing the sequential code to improve detection of parallelism. We currently use numerical dependency counts extracted from the dynamic data dependency graph of the application, but it is possible that a full comparison of dependency graphs, using a graph kernel or graph similarity method, could capture more of the information present in these graphs. Control-flow graphs could also be produced and analyzed to contribute information about loop behavior, possibly increasing accuracy in the classification of potential parallelism.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.

[3] L. Breiman. *Classification and regression trees*. CRC press, 1993.

[4] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[5] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, Jan. 2012.

[6] Y. Freund, R. Schapire, and N. Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[7] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[8] T. Hastie, R. Tibshirani, and J. J. H. Friedman. *The elements of statistical learning*, volume 1. Springer New York, 2001.

[9] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[10] Z. Li, A. Jannesari, and F. Wolf. Discovery of Potential Parallelism in Sequential Programs. In *Proc. of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI), Lyon, France*, 2013.

[11] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 196–206. ACM, 2012.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*, pages 61–74. MIT Press, 1999.

[14] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148. IEEE, 2011.

[15] S. Srivastava, B. Malone, N. Sukhija, I. Banicescu, and F. M. Ciorba. Predicting the flexibility of dynamic loop scheduling using an artificial neural network. In *12th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2013.

[16] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM Sigplan Notices*, volume 44, pages 177–187. ACM, 2009.

[17] C.-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 366–373. IEEE, 1988.