"*Smart*" Manual Software Parallelisation Assistant

Aleksandr Maramzin*[1], Björn Franke*[2],
Murray Cole*[2]

* *Institute For Computing Systems Architecture, Informatics Forum, The University of Edinburgh, 10 Crichton Street, Edinburgh, UK*

ABSTRACT

Since automatically parallelizing compilers have failed to deliver significant performance improvements, programmers are still forced to parallelize legacy software manually for all but some niche domains. Rather than hoping for an elegant silver bullet, we acknowledge the role of a human expert in the parallelization process and develop a *smart* parallelization assistant.

In its essence our assistant is yet another application of machine learning techniques to the field of optimizing compilers, which tries to predict the parallelisability property of program loops. We use Seoul National University version of NAS Parallel Benchmarks (NPB) [Div12], [Uni12] hand-annotated with OpenMP parallelisation pragmas to train our ML model. We show that the loop parallelisability classification problem can be successfully tackled with machine learning techniques (using only static code features) achieveing accuracy of around 90% and outperforming all available baseline random predictors working at an accuracy ranging between 40% and 70%.

To get a real practical application of our techniques, we integrate our trained ML model into an assistant scheme, designed to mitigate the effects of ineradicable statistical errors and make them less critical. Taking application profile our assistant directs a programmer's efforts by pointing the loops, which are highly likely to be parallelisible and profitable as well. Thus, decreasing the efforts and time it takes to parallelize a program manually. As a side effect our assistant extends the capabilities of Intel C/C++ compiler in the task of parallelism discovery by increasing the amount of parallelism found in SNU NPB benchmarks from 81% to 96%.

KEYWORDS: ACACES; poster session; software engineering; parallel programming; compilers; static program dependence analysis; loop iterator recognition; machine learning; programmer feedback;

1 Introduction

Since automatically parallelizing compilers have failed to deliver significant performance improvements, programmers are still forced to parallelize legacy software manually for all but some niche domains. Indeed, as our preliminary exploratory experiments showed the effects that Intel C/C++ Compiler (ICC) had brought with its vectorizing and parallelizing transformations to SNU NPB benchmarks ranged from no change to a significant slowdown.

[1]E-mail: s1736883@sms.ed.ac.uk
[2]E-mail: {bfranke,mic}@inf.ed.ac.uk

Rarely ICC vectorization was able to achieve a tiny speedup, but the former faded in comparison with the speedup of an expertly parallelised SNU NPB OpenMP version. Given these observations we decided to acknowledge the role of a human expert in the parallelization process and develop a *smart* manual software parallelization assistant.

Our assistant is based on a ML model, trained to classify program loops as parallelisible or not. We use Seoul National University version of NAS Parallel Benchmarks (NPB) [Div12], [Uni12] hand-annotated with OpenMP parallelisation pragmas to train our ML model. We show that the loop parallelisability classification problem can be successfully tackled with machine learning techniques (using only static code features) achieveing accuracy of around 90% and outperforming all available baseline random predictors working at an accuracy ranging between 40% and 70%.

The application of ML techniques to the field of parallelising compilers is not a new endeavour. As the survey of machine learning in optimising compilers [WO18] shows there have been numerous successful application attempts. Machine learning can be used for problems ranging from selecting the best compiler flags to determining how to map parallelism to processors. Uneradicable statistical errors inherent to all ML techniques might result in the performance losses here, but do not compromise the ultimate functional correctness of the program being compiled. In this work we step into a potentially dangerous area. False positives (non-parallel loops predicted as parallelisable) can break the program. There is an already published work [FLJW13] studying the possibility of learning loop parallleisability property. The answer is "yes" and the work reports on a predictive performance (accuracy, recall, precision, etc.) we can achieve. But apart from studying the possibility that work does not step any further to actually find any practical utilisation of loop parallelisability predictor. Moreover, the dataset is highly unbalanced (), which sets the baseline accuracy very high. The authors use dynamic program features.

In our work we use static program features for the same loop parallelisability classification problem. We work with a more balanced dataset with richer sources of information our classification labels have been derived from. At the end we harness our trained ML model into a practical application scheme. The scheme has been designed to mitigate the effects of ineradicable statistical errors and make them less critical. Taking application profile our assistant directs a programmer's efforts by pointing the loops, which are highly likely to be parallelisible and profitable as well. Thus, decreasing the efforts and time it takes to parallelize a program manually. As a side effect our assistant extends the capabilities of Intel C/C++ compiler in the task of parallelism discovery by increasing the amount of parallelism found in SNU NPB benchmarks from 81% to 96%.

2 Predicting Parallel Loops

We pose a supervised ML classification problem: create a machine learning (ML) based model and train it to classify loops of Seoul National University implementation [Uni12] of NAS Parallel Benchmarks [Div12] as parallelizable or not.

Our work encompasses two technical aspects we need to resolve. First, we need to find a representative set of quantifiable features, which will accurately reflect the parallelisability property of program loops. Once the problem of so-called feature engineering has been solved we need to select the exact ML training/testing methodology we are going to employ for our task. By ML methodology we mean

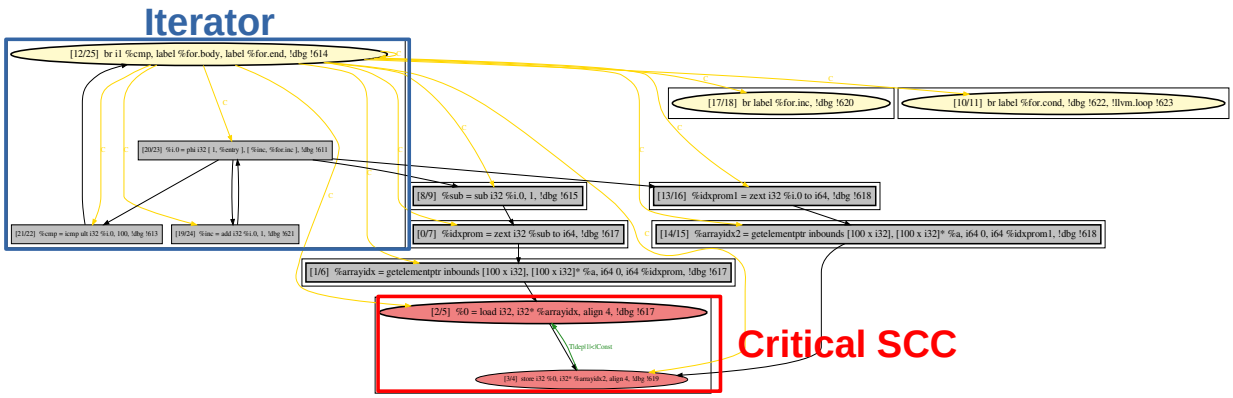

Figure 1: Illustration of a faster convergence to the maximum achievable parallel software performance.

This section describes the exact way we applied machine learning (ML) techniques to the problem of loop parallelisability prediction. Here we describe the features we chose to represent program loops, as well as the whole we used.

Our features and the exact parameters of methodology (automatic feature selection methods, ML models and their hyper-parameter spaces, etc.) have been iteratively tuned and refined with the help of K-fold CV. Predictive accuracy, recall and precision scores were used as the main selection criteria. To get the most accurate and honest assessment of our ML models we kept the testing data hidden and used only the training data during all ML pipeline stages. Following subsections present the final results rather than the path towards them.

For any machine learning methodology to work and make accurate predictions we need to choose and set all its stages correctly. Apart from ML model selection (Support Vector Machines, Decision Trees, Neural Networks, etc.) we need to provide training and testing data sets in the right format and decide on the exact training/testing approach we are going to employ.

We used facilities of *scikit-learn* [PVG+11] Python library for all machine learning related tasks. We developed a scripting framework based on this library taking an input data (training and testing) along with a ML pipeline configuration INI file. Configuration file allows for a flexible change in the settings of an experiment (ML model to use, its hyper-parameters, the exact automatic feature selection methods, etc.). The following sections give a detailed description of all ML pipeline stages.

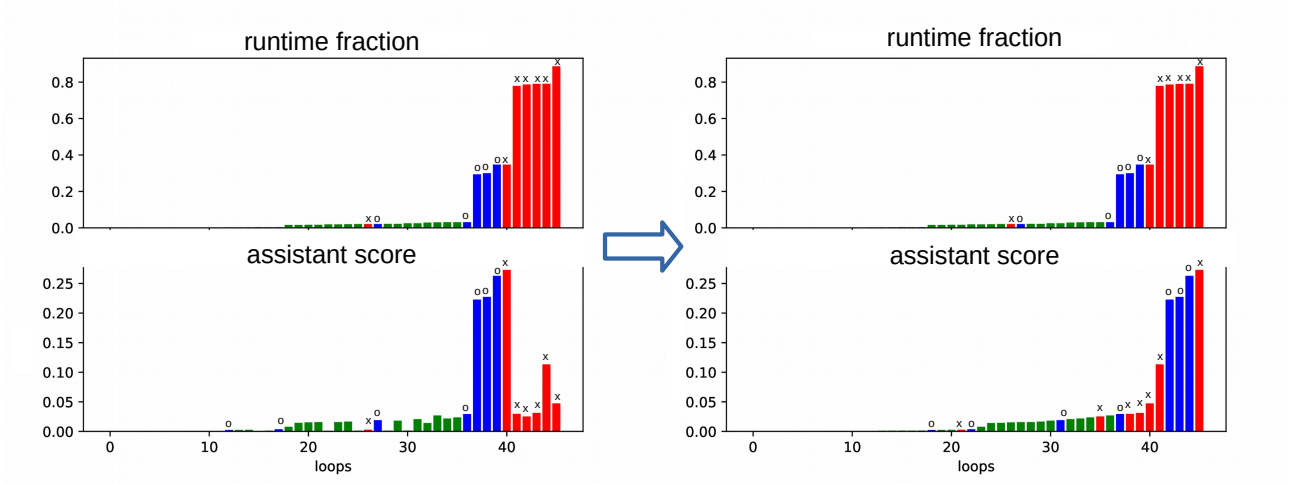

Figure 2: Illustration of loop ranking transformation.

2.1 Feature Engineering

2.2 Machine Learning Methodology

3 Manual Software Parallelisation Assistant

4 Evaluation

4.1 ML Model Predictive Performance

4.2 Assistant Scheme

References

[Div12] NASA Advanced Supercomputing (NAS) Division. Nas parallel benchmarks, August 2012.

[FLJW13] Daniel Fried, Zhen Li, Ali Jannesari, and Felix Wolf. Predicting parallelization of sequential programs using supervised learning. In *Proc. of the 12th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA*, pages 72–77. IEEE Computer Society, December 2013.

[PVG+11] 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.

[Uni12] Seoul National University. Snu nas parallel benchmarks, August 2012.

[WO18] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 11 2018.

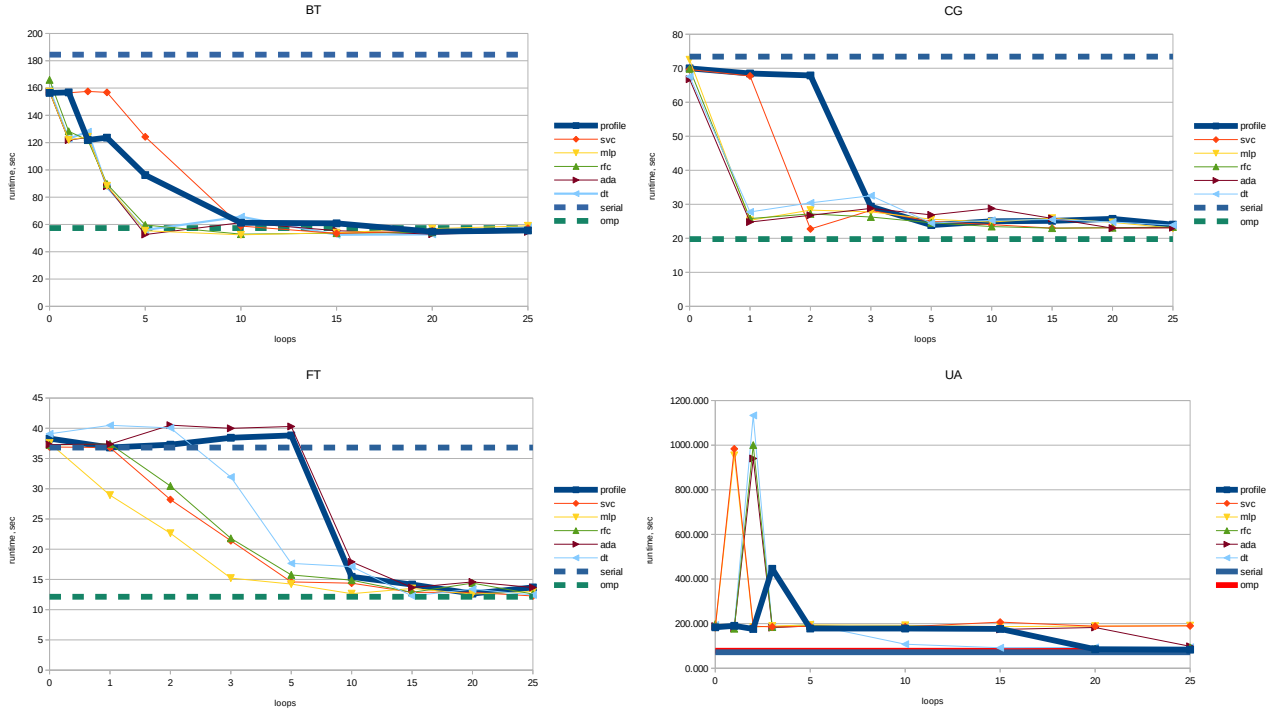

Figure 3: Illustration of a faster convergence to the maximum achievable parallel software performance.