"*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 machine learning (ML) classification problem***: create an ML based model and train it to classify loops of Seoul National University (SNU) implementation [Uni12] of NAS Parallel Benchmarks (NPB) [Div12] as parallelizable or not.

Our work encompasses two technical aspects. First, we need to find a representative set of quantifiable features, which would accurately reflect the parallelisability property of program loops, then we need to select the exact ML training/testing methodology to employ for our problem.

Our features and the exact parameters of methodology (automatic feature selection meth-

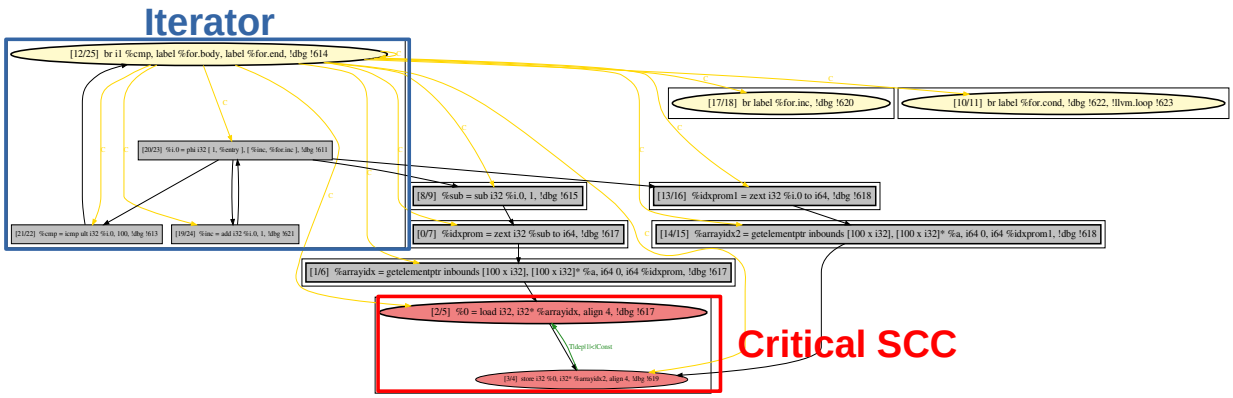

Figure 1: Program Dependence Graph of a simple loop with recognised iterator, regular and critical payload parts. Nodes are LLVM IR instructions, edges are different types of dependencies between them.

ods, ML models and their hyper-parameter spaces, etc.) have been iteratively tuned with the help of K-fold CV. Predictive accuracy, recall and precision scores were used as the main selection criteria. We used facilities of *scikit-learn* [PVG+11] Python library for all machine learning related tasks.

2.1 Feature Engineering

In the task of coming up with a set of loop features we are guided by general program dependence analysis theory [KA02], exact types of loops present in SNU NPB benchmarks and Intel C/C++ compiler optimisation reports.

There is a range of SNU NPB loops, which escape Intel compiler parallelisation for different reasons: indirect array references, unrecognised reductions on array locations, pointers with statically unknown memory locations, etc. But all that range of different reasons is going to ultimately materialise into data and control dependencies present between loop instructions, represented as edges on the Program Dependence Graph (PDG) [FOW87] of a loop. To refine our features we conduct a generalised loop iterator recognition [MVF18] analysis on the PDG graph. Figure 1 shows an example of a PDG built for a simple loop and visualised with our tool.

We base our loop features on a static structural properties of PDG. There are 74 different static loop features we use for the ML task. The features are simplistic and defined as different numbers and fractions calculated on a graphs: number of particular LLVM IR instructions (calls, loads, stores, branches, etc.), numbers of various dependence edges (true, anti, output, etc.), sizes of iterators and payloads, etc.

2.2 Machine Learning Methodology

The final predictive performance we are going to achieve depends not only on the successful choice of loop features, but on the exact training and testing methodology as well. The latter is composed of a number of stages lined up in a pipeline. The pipeline starts with data preprocessing and automatic feature selection (low variance feature elimination, tree

based methods, RFECV, etc.) in order to exclude redundant and irrelevant features, which lean to model over-fitting. Then we choose the exact ML model (SVC, DT, RFC, MLP, AdaBoost) to use and tune its hyper-parameters. Typical examples include *C*, *kernel* and γ for Support Vector Classifiers (SVC), exact architecture for neural network based models, etc. And finally, we conduct the actual training and testing with standard K-fold and modified Leave-One-Out Cross-Validation (LOOCV) techniques described in the literature [JWHT14]. These methods aim at different goals. While K-fold method averages accuracy scores on different data set splits and can be used for feature selection, hyper-parameter tuning and overall predicitve performance estomation, modified LOOCV can be applied against any single SNU NPB benchmark in order to assess our assistant scheme (see section 3). Table 1 provides the ultimate accuracy we managed to achieve.

3 Manual Software Parallelisation Assistant

Achieving high prediction accuracy is not enough. We need to find a way to utilise our work practically. Due to statistical nature inherent to all ML techniques, it is impossible to completely eliminate all prediction errors. While false negative mispredictions might just miss available parallelisation opportunities and lose some performance, false positive mispredictions can break the program and are the most critical in the context of our ML problem. For that reason we propose assistant schemes that leave the question of final parallelisation up to a programmer to decide.

ML model	accuracy	recall	precision
constant	70.32	100	70.32
uniform	46.27	41.50	69.79
SVC	90.04	95.24	91.06
AdaBoost	86.96	92.92	89.06
DT	84.36	89.57	87.90
RFC	86.65	93.22	88.47
MLP	89.40	93.77	91.39

Table 1: Average predictive performance for different ML models measured with a K-fold CV method on the whole set of 1415 SNU NPB loops.

Our predictor is capable of discovering additional parallelisation opportunities, which escape the conservative analysis of Intel compiler. For SNU NPB benchmarks we increse the amount of discovered parallelism from 81% to 96%. But not all parallel loops have to be parallelised. We are interested only in the loops, which take significant application runtime fraction and thus directly affect overall performance. Our assistant takes an application profile and orders an application loops according to their running time. But not all long running loops are parallelisable. We multiply loop running times on the function (shifted sigmoid) of loop parallelisability probability extracted out of ML model and get an improved ranking. Figure 2 illustrates the principle.

4 Evaluation

Parallelising the program by following the reorded list of program loops, a programmer can get to the best achievable performance faster. Figure 3 illustrates the process.

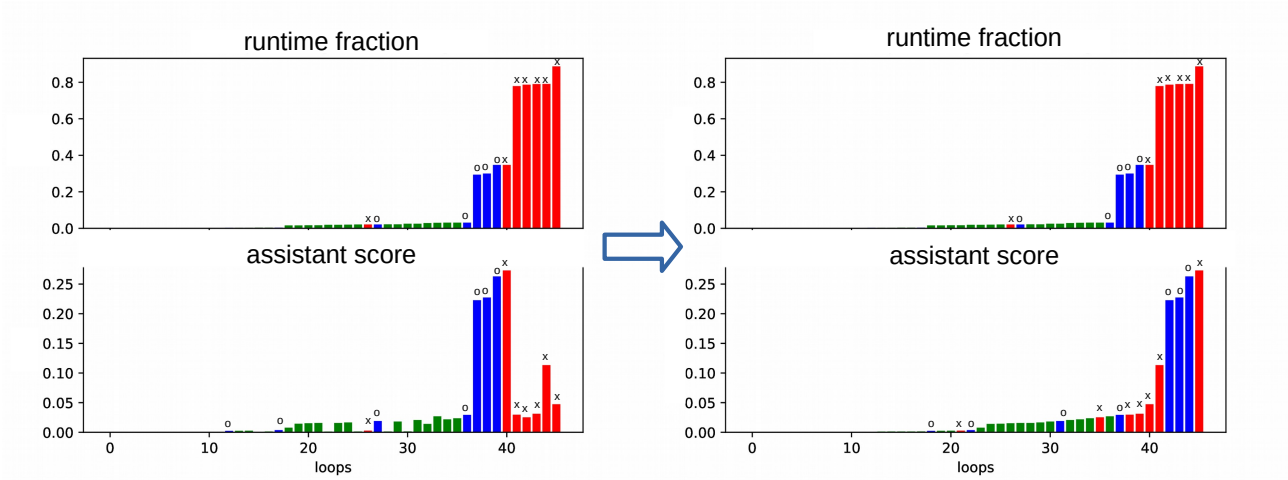

Figure 2: Illustration of loop ranking transformation.

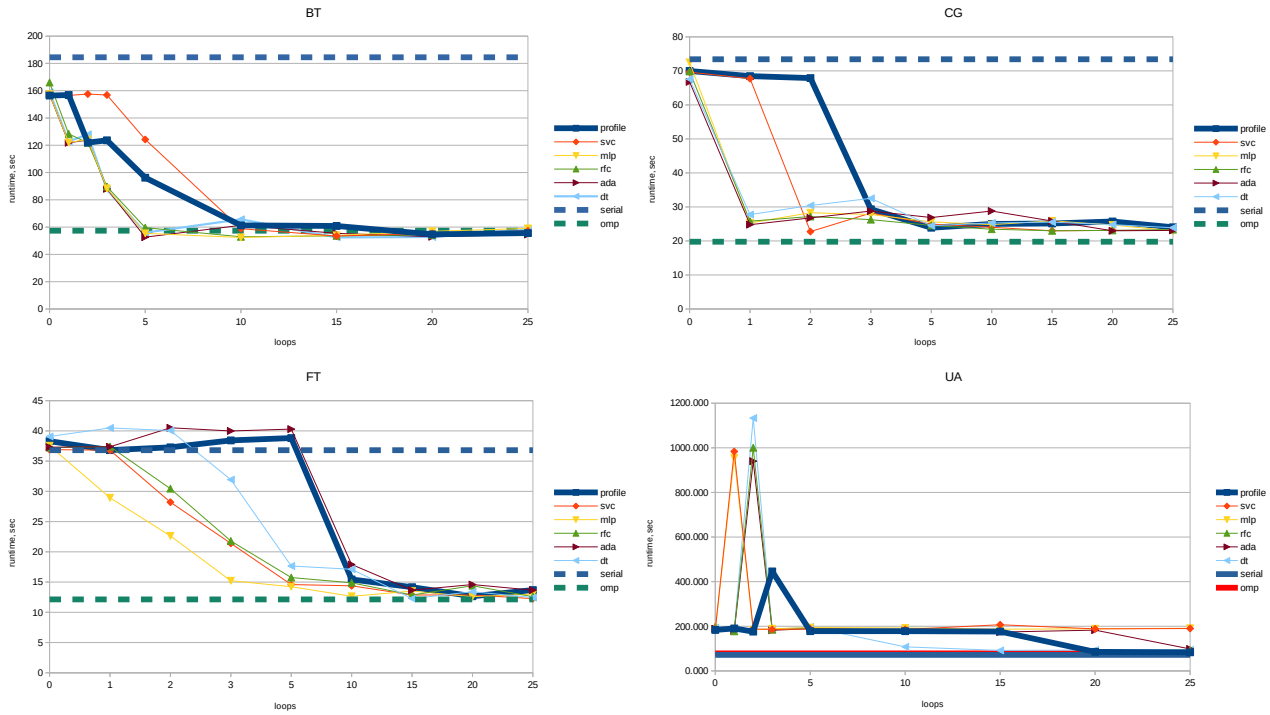

Figure 3: Illustration of a faster convergence to the maximum achievable parallel software performance.

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.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[JWHT14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[MVF18] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 185–195, New York, NY, USA, 2018. ACM.

[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.