“It Looks Like You’re Writing a Parallel Loop”

A Machine Learning Based Parallelization Assistant

Aleksandr Maramzin
The University of Edinburgh
United Kingdom
s1736883@sms.ed.ac.uk

Christos Vasiladiotis
The University of Edinburgh
United Kingdom
c.vasiladiotis@sms.ed.ac.uk

Roberto Castañeda Lozano
The University of Edinburgh
United Kingdom
roberto.castaneda@ed.ac.uk

Murray Cole
The University of Edinburgh
United Kingdom
mic@ed.ac.uk

Björn Franke
The University of Edinburgh
United Kingdom
bfranke@ed.ac.uk

Abstract

Despite decades of research into parallelizing compiler technology, software parallelization remains a largely manual task where the key resource is expert time. In this paper we focus on the time-consuming task of identifying those loops in a program, which are both worthwhile and feasible to parallelize. We present a methodology and tool which make better use of expert time by guiding their effort directly towards those loops, where the largest performance gains can be expected while keeping analysis and transformation effort at a minimum.

We have developed a novel parallelization assistant that provides programmers with a ranking of all loops in a program based on their overall merit. For each loop this metric combines its potential contribution to speedup and an estimated probability for its successful parallelization. This probability is predicted using a machine learning model, which has been trained, validated, and tested on 1415 labelled loops, achieving a prediction accuracy greater than 90%.

We have evaluated our parallelization assistant against sequential C applications from the SNU NAS benchmark suite. We show that our novel methodology achieves parallel performance levels comparable to those from expert programmers while requiring less expert time. On average, our assistant reduces the number of lines of code that have to be inspected manually before reaching expert-level parallel speedup by 20%.

CCS Concepts • **Computing methodologies → Machine learning**; • **Software and its engineering → Compilers**.

Keywords Compilers, loop parallelization, machine learning, assistant tool

ACM Reference Format:
Aleksandr Maramzin, Christos Vasiladiotis, Roberto Castañeda Lozano, Murray Cole, and Björn Franke. 2019. “It Looks Like You’re Writing a Parallel Loop”: A Machine Learning Based Parallelization Assistant. In *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems (AI-SEPS ’19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358500.3361567

1 Introduction

Parallel hardware is ubiquitous through the entire spectrum of computing systems, from low-end embedded devices to high-end supercomputers. Yet, most of the existing software is written in a sequential fashion. Despite decades of intensive research in automatic software parallelization [16], fully exploiting the potential of modern multi- and many-core hardware still requires a significant manual effort. Given the difficulty of the obstacles faced by automatic parallelization today, we do not expect that programmers will be liberated from performing manual parallelization in the near future [12].

This paper introduces a novel parallelization assistant that aids a programmer in the process of parallelizing a program in the frequent case where automatic approaches fail to do so. The assistant reduces the manual effort in this process by presenting a programmer with a ranking of program loops that are most likely to 1) require little or no effort for successful parallelization and 2) improve the program’s performance when parallelized. Thus, it improves over the traditional, profile-guided process by also taking into account the *probability* of potential parallelization for each of the profiled loops.

AI-SEPS ’19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6983-1/19/10…$15.00
https://doi.org/10.1145/3358500.3361567

At the core of our parallelization assistant resides a novel machine-learning (ML) model of loop parallelizability. Loops are compelling candidates for parallelization, as they are naturally decomposable and tend to capture most of the execution time in a program. Furthermore, focusing on loops allows the model to leverage a large amount of specific analyses available in modern compilers, such as generalized iterator recognition [14] and loop dependence analysis [10]. The model encodes the results of these analyses together with basic properties of the loops as machine learning *features*.

The loop parallelizability model is trained, validated, and tested on 1415 loops from the SNU NAS Parallel Benchmarks (SNU NPB) [19]. The loops are labelled using a combination of expert OpenMP [5] annotations and optimization reports from Intel C++ Compiler (ICC), a production-quality parallelizing compiler. The model is evaluated on multiple machine learning algorithms, including tree-based methods, support vector machines, and neural networks. The evaluation shows that – despite the limited size of the data set – using support vector machines allows the model to achieve a prediction accuracy higher than 90%. The model improves over the ICC Compiler across the sequential C version of the SNU NPB suite by detecting 13% more parallel loops. Albeit this improvement comes at the cost of introducing *false positives*, where non-parallelizable loops are misclassified as parallelizable. However, the false positive rate in our evaluation is as low as 6.5%. We feel this is acceptable, as our parallelization assistant does not automatically restructure code but leaves the parallelization decision in the hands of the programmers.

The parallelization assistant combines inference on the parallelizability model with traditional profiling to rank higher those loops with a high probability of being parallelizable and impacting the program performance. An evaluation on eight programs from the SNU NPB suite shows that the program performance tends to improve faster as loops are parallelized in the ranking order suggested by our parallelization assistant compared to a traditional order based on profiling only. On average, following the order suggested by the assistant reduces by approximately 20% the number of lines of code a programmer has to examine manually to parallelize SNU NPB to its expert-level speedup. Given the high level of effort involved in manual analysis, such a reduction can translate into substantial development cost savings.

1.1 Motivating Example

Consider the sequential C implementation of the *Conjugate Gradient (CG)* benchmark from the SNU NPB suite. Table 1 shows the top three CG loops as ranked by the Intel Profiler (based on their execution time) and by our parallelization assistant (additionally taking into account their parallelizability). Both rankings include the same loops, but, crucially, the loops are **ranked in a different order**.

Table 1. Comparison of the profiler and assistant rankings for the CG benchmark loops (limited to the top three loops).

Ranking	Profiler	Assistant	
	loop	loop	parallelizability
1	cg.c:326	**cg.c:509**	85%
2	cg.c:484	cg.c:326	29%
3	**cg.c:509**	cg.c:484	8%

```
for (it = 1; it <= NITER; it++) {
  ...
  if (timeron) timer_start(T_conj_grad);
  conj_grad(colidx,rowstr,x,z,a,p,q,r,&rnorm);
  if (timeron) timer_stop(T_conj_grad);
  ...
  printf("    %5d       %20.14E%20.13f\n", it,
    rnorm, zeta);
  ...
}
```

Listing 1. cg.c:326. Longest running loop in CG. The loop cannot be parallelized due to inter-iteration dependences and side effects caused by system calls.

```
for (j = 0; j < lastrow-firstrow+1; j++) {
  suml = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++)
    suml = suml + a[k]*p[colidx[k]];
  q[j] = suml;
}
```

Listing 2. cg.c:509. Longest running loop in CG among those *that can be parallelized*.

Following the profiler ranking (second column in Table 1), a parallelization expert would concentrate on analyzing the loop cg.c:326 first. Analyzing this loop turns out to be costly (it consists of 100+ lines of code) and unfruitful (it is actually not parallelizable due to inter-iteration dependencies and side effects, see Listing 1). This analysis would be followed by an equally unfruitful analysis of loop cg.c:484.

In contrast, our parallelization assistant (two last columns in Table 1) ranks the loop **cg.c:509** before loops cg.c:326 and cg.c:484, as it finds that the former has a significantly higher probability of being parallelizable (see last column in Table 1). Following the assistant's ranking, a parallelization expert would thus concentrate on analyzing the loop **cg.c:509** first. Analyzing this loop is inexpensive (it consists of only six lines of code, see Listing 2) and fruitful: its parallelization speeds up CG by a factor of 2.8, which is 70% of the speedup obtained by parallelizing the entire benchmark. Hence, following the ranking proposed by our assistant, a parallelization expert can achieve most of the available speedup in CG in a fraction of the time required by a traditional profile-guided parallelization process.

1.2 Contributions

In summary, this paper makes the following contributions:

- ▷ We introduce a machine learning model, which can be used to predict the probability with which sequential C loops can be parallelized (Sections 2 and 3);
- ▷ we integrate profiling of execution time with our novel ML model into a parallelization assistant, which guides the user through a ranked list of loops for parallelization (Section 4); and
- ▷ we demonstrate that our tool and methodology increase programmer productivity by identifying parallel loop candidates better than existing state-of-the-art approaches (Section 5).

2 Predicting Parallel Loops

We approach the prediction of parallel loops as a *supervised probabilistic machine learning classification problem*. Based on sequential reference applications and their manually parallelized counterparts as well as Intel's parallelizing C/C++ compiler, we create a data set of parallelizable and non-parallelizable loops. We extract loop features and use the data set to train a machine learning model, which links feature vectors describing the loops with their observed parallelizability. We then use the trained model as a probabilistic predictor: for each new loop we determine its feature vector and then predict the probability of the loop being parallelizable [17]. For naturally probabilistic models like trees we directly use the computed classification probabilities (fraction of parallelizable training samples in the leaf node). For the support vector machines classifier, we use Platt scaling to derive the probabilities.

In this section we introduce the parallelizability model, whereas Section 3 presents a standard ML performance assessment including accuracy, precision and recall scores. Descriptions and definitions of the machine learning techniques we use can be found in [9]. We used the *scikit-learn* library [18] for all ML related tasks.

2.1 Loop Analysis & Feature Extraction

For the purpose of machine learning, program loops are represented by numerical *feature vectors*. We derive these features using standard compiler analyses operating on the Program Dependence Graph (PDG) [6] of a loop. The PDG is a representation that captures both data and control information and is constructed using dependence analysis [11]. In addition we use *generalized loop iterator recognition* [14] to separate *loop iterators* from *loop payloads*. This enables us to define and extract features relating to each of those loop components. In total, we extract a set of 74 static loop features which are based on structural properties of the PDG and the types of instructions constituting it. Table 2 summarizes these features.

Our features have simple and intuitive motivations behind them. Loop proportion features are backed up by the fact that larger loops tend to be harder to parallelize. Complex iterators include non-trivial cross-iteration transitions (e.g linked-list update), unknown iteration numbers, etc. Payload SCCs (Strongly Connected Components) introduce cross-iteration dependencies. Cohesion features characterize how tightly components of loops are coupled together in terms of the number of edges between them. Loop dependence features count the number of edges in different loop parts as well as their types. Loop instruction features characterize the loop's instruction mix, assigning more importance to memory reads/writes, calls and branches. Non-inlined function calls usually prevent loop parallelization. Intensive memory work (memory read/write fraction features) complicates parallelization as well.

2.2 Feature Selection

To avoid overfitting, we discard irrelevant or redundant features using a pipeline of feature selection methods from the scikit-learn library. First we eliminate features with a low variance score, then we fit a decision tree-based model and select features with importance score above a given threshold. After that we repeatedly run *Recursive Feature Elimination, Cross-Validated (RFECV)* to improve accuracy, precision and recall scores. This yields the final set of features. Table 3 presents the 10 highest-ranked features in this set.

2.3 Model & Hyper-Parameter Selection

We evaluate several machine learning classification algorithms in our parallelization assistant, including tree-based methods like decision trees (DT), random forests (RFC) and boosted decision trees (AdaBoost); support vector machines classifiers (SVC) and multi-layer perceptron neural networks (MLP). Section 3.1 shows that these models perform similarly with SVC and MLP performing slightly better.

For each ML model we use exhaustive hyper-parameter grid search and pick the grid node with the best cross-validation score on the validation set. The details of all ML pipeline stages are available in our repository [15].

2.4 Training Data & ML Model Training

For training our ML model we use a total of 1415 loops from the SNU NPB benchmark suite. Out of those loops, 210 have been annotated by (external) human experts with parallel OpenMP *pragmas*. We use these annotations as labelled data to indicate parallelizable loops. However, the data is not complete. Human programmers strive to capture only coarse-grain parallelism and do not annotate every parallelizable loop. Hence, we augment the training data with the help of the ICC Compiler, which finds additional parallelizable loops. We combine the results into our training set comprising a total 1415 loops, of which 995 are labelled as parallelizable. We then use K-fold and Leave-One-Out Cross-Validation (LOOCV) methodologies to train and test our ML models.

Table 2. Static features used for the characterization of loops.

Feature groups	Features	Description
loop proportion	absolute size	number of LLVM IR instructions
	payload fraction	payload instructions / total loop instructions
	proper SCCs number	number of payload SCCs with more than 1 instruction
loop dependencies	number of PDG edges for different dependence classes: read/write order (*true*, *anti*, *output*), dependency type (*register*, *memory*, *control*), other (*cross-iteration*, etc.)	
loop cohesion	iterator/payload	$\frac{\text{edges between iterator/payload}}{\text{total loop edges}}$
	critical/regular payload	$\frac{\text{edges between critical/regular payload}}{\text{total loop payload edges}}$
loop instruction nature	numbers and fractions of different parallelization critical instructions (memory loads and stores, branches, calls, etc.)	

Table 3. Relative importance of static loop features, ranked by fitting a tree-based ML model.

Feature	Importance
payload call fraction	23.5
iter/payload non-cf cohesion	18.5
payload mem write fraction	6.1
loop absolute size	5.7
critical payload pointer access count	5.3
payload memory dependence count	4.0
critical payload non-cf cohesion	2.9
payload pointer access fraction	2.7
critical payload total cohesion	2.6

3 ML Predictive Performance

In our work we employ two cross-validation (CV) techniques. We evaluate the overall predictive performance our trained ML model is capable of achieving on SNU NPB benchmarks using K-fold CV. To deploy our assistant against single benchmarks of the suite and assess its effectiveness (Section 5) we have to use a modified Leave-One-Out CV.

3.1 Overall Model Performance

Table 4 shows the overall predictive performance of different ML models measured with K-fold CV on the whole SNU NPB data set. Training and testing have been done for different values of K (5, 10, 15, 20, 25, 30) and the accuracy remains stable across the entire range. The same is true of recall and precision scores. We used the baselines (constant "parallelizable" prediction and uniform) available in scikit-learn to compare our models against.

The SVC model has the highest average accuracy and successfully manages to recall 95.24% of all parallel loops. The ICC Compiler succeeds in parallelizing 812 out of 995 parallelizable loops available in SNU NPB. Thus, on average SVC extends the ICC Compiler's parallelization capabilities to 948 loops. Figure 1 shows that out of the 10% of mispredictions that SVC makes, 65% are false positives. Hence, the average unsafe error rate is 6.5%.

Table 4. Average predictive performance of different ML models measured with K-fold CV on the whole set of 1415 SNU NPB loops.

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

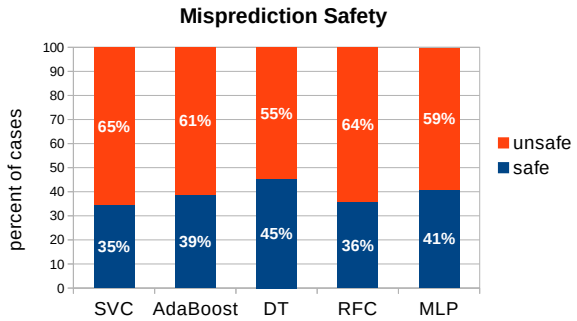

Figure 1. Breakdown of misclassification errors.

3.2 Model Performance Within Assistant

Our proposed assistant (Section 4) is trained and tested using LOOCV rather than K-fold CV. Instead of treating the entire set of loops from all SNU NPB benchmarks as a single data set, in this context we train the model on nine benchmarks and test it on the remaining one. Doing so completely excludes the loops of the target benchmark out of a training set, but allows us to get predictions for all benchmark loops, parallelize them if advised so, and test the effectiveness of our assistant. The drawback of this scheme is that it might potentially reduce the accuracy if the nature of loops in the target benchmark dramatically differs from that of loops seen in the training set. Figure 2 compares LOOCV accuracy against that of K-Fold CV for all SNU NPB benchmarks, where K-Fold CV is conducted on a data set consisting of

loops from a single benchmark only. The comparison proves that lower LOOCV accuracies are attributed to a reduced training data set and not to our ML model.

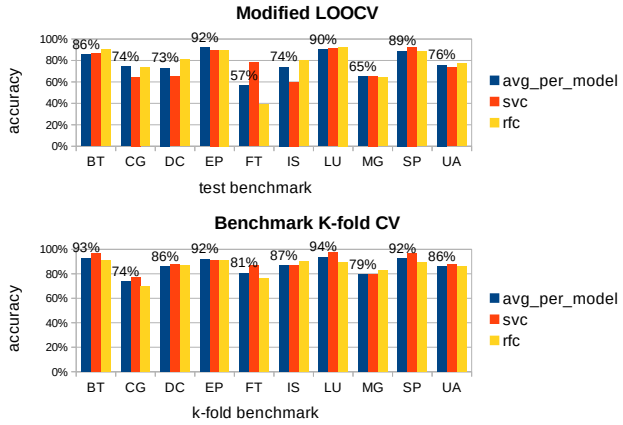

Figure 2. Prediction accuracy measured with modified LOOCV and compared against that of K-fold CV for single benchmarks.

4 Parallelization Assistant

The ML-based predictor developed and assessed in the previous two sections is a core component of our novel parallelization assistant. This assistant incorporates prediction results on whether a loop can be parallelized and combines this with profiling information. It then produces a ranking of all loops in an application to guide a programmer towards the most beneficial loop candidates for their *manual* parallelization effort. We do not seek to replace the programmers from the process, but aim to assist and increase their productivity.

Loop Ranking. The loop ranking computed by our parallelization assistant combines a loop's contribution to the overall program execution time with its predicted probability of being parallelizable. In particular, we obtain the ranking by applying a shifted sigmoid function to the predicted parallelizability probability multiplied by the application runtime fraction a loop takes to run as shown in Figure 3.

The intuition for using this function to combine the two metrics is that it prioritizes parallelizable long-running loops and scales down the weight of non-parallelizable loops irrespective of their contribution to execution time. The effect of this can be seen in Figure 4 for the loops of the FT benchmark.

If programmers attempt to parallelize loops in the order prescribed by their execution time, they will inevitably waste their efforts trying to parallelize loops which may be long-running, but offer little or no opportunity for extracting parallelism. Instead, by taking into account predicted parallelizability our ranking directly guides the programmer towards loops that significantly contribute to overall execution time **and** offer a realistic prospect of parallelization.

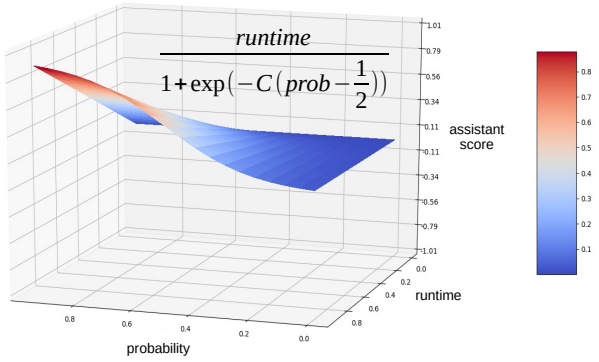

Figure 3. For each loop the ranking function combines its contribution to the application's execution time and its predicted probability of being parallelizable.

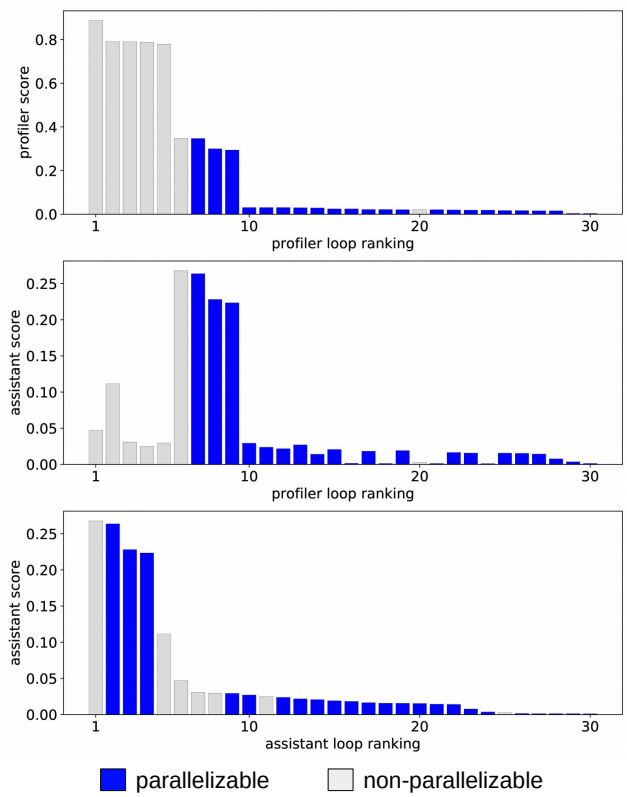

Figure 4. Change of loop rankings with the application of the assistant ranking function for the 46 loops of the FT benchmark. Ranking based on loop execution time alone (top figure) results in some high-ranked, but non-parallelizable loops. Combining profiled execution time and parallelizability in a single score (middle figure) results in a ranking that prioritizes parallelizable loops (bottom figure).

Table 5. Possible loop classification combinations for the side-by-side setup of our ML predictor and the ICC Compiler.

ICC	Predictor	True Parallel	Classification
0	0	0	icc/predictor agreement
0	0	1	missed opportunity
0	1	0	false positive
0	1	1	discovery
1	0	0	impossible
1	0	1	icc shielding
1	1	0	impossible
1	1	1	icc/predictor agreement

4.1 Comparison to Static Analysis

We have compared the generated loop parallelizability classifications of our assistant against that of the ICC Compiler, which due to its use of static analysis is conservative and occasionally misses some parallelization opportunities. The ML approach to parallelization with a human programmer responsible for final code transformation allows our parallelization assistant to be more aggressive than the ICC Compiler. In other words, our model can predict more loops as parallelizable.

We have set up an experiment where we apply our ML predictor side-by-side with the ICC Compiler. Both aim at independently classifying loops as parallelizable or not. There is a total of six possible classification combinations that our scheme can produce (summarized in Table 5). The first and last table rows show the agreement cases, where the ICC Compiler and the ML predictor identically classify truly (non-)parallelizable loops as (non-)parallelizable. The "missed opportunity" cases where both ICC Compiler and ML predictor miss parallelizable loops also represent agreement and are not interesting. The most interesting cases are those where the ML predictor and the ICC Compiler disagree. While ICC Compiler is conservative and will never classify a non-parallelizible loop as parallelizible, the statistical ML predictor can make a "false positive" error. That works in the opposite direction as well. The ML predictor can discover truly parallelizible loops which escape compiler analysis. These cases are classified as "discovery" and have been manually checked in the source code of SNU NPB. The results are summarized in Table 6, which reports on the reasons behind ICC conservativeness. False negative mispredictions make the ML predictor miss some real parallelization opportunities, but in the fraction of these cases the ICC Compiler can catch them and "shield" the ML predictor.

Figure 5 shows the relative frequency of different loop classification combinations from Table 5. We repeatedly ran K-fold CV on the whole set of SNU NPB loops and sorted the outcomes into separate classification buckets from Table 5. In 80% of the cases, our parallelization assistant agrees with the ICC Compiler and classifies truly (non-)parallel loops as (non-)parallel. This is an expected result as we have used ICC (along with OpenMP annotated loops) to train our ML model. However, sometimes our parallelization assistant and ICC reach different conclusions. While there is agreement in the majority of the cases, our tool discovers an additional 10% of genuine parallelism inaccessible to the ICC Compiler, while allowing 8% of false positives.

Table 6. Classification of parallelizable loops rejected for parallelization by the ICC Compiler.

Reason	Num	Reason	Num	Reason	Num
missed reduction	18	array privatization	7	conservative analysis	60
unknown iteration number	7	static dependencies	46	too complex	22
non-inlined calls	4	other	4	total	168

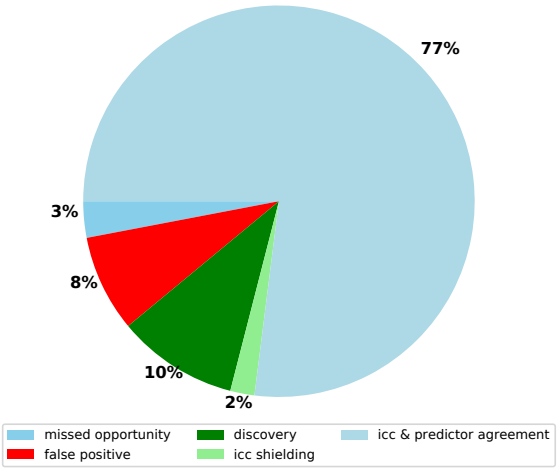

Figure 5. Distribution of loop classification by the ICC Compiler and our predictor.

5 Assistant Evaluation

In this section we evaluate the effectiveness of our parallelization assistant. In particular, we are interested in the potential programmer productivity gains delivered by our tool and savings on human expert time.

Our study assumes that the human expert starts with a sequential version of the SNU NPB benchmarks. The goal is to parallelize these applications to a performance level matching that of their existing parallel versions. By using our assistant we expect the human expert to consider fewer loops than by using a profiling-based approach, i.e. considering loops in decreasing execution time. We also compare against the state-of-the-art fully automated parallelization approach implemented in the ICC Compiler. The ICC Compiler not only fails to achieve noticeable speedups, but actually slows the performance down on most of the benchmarks.

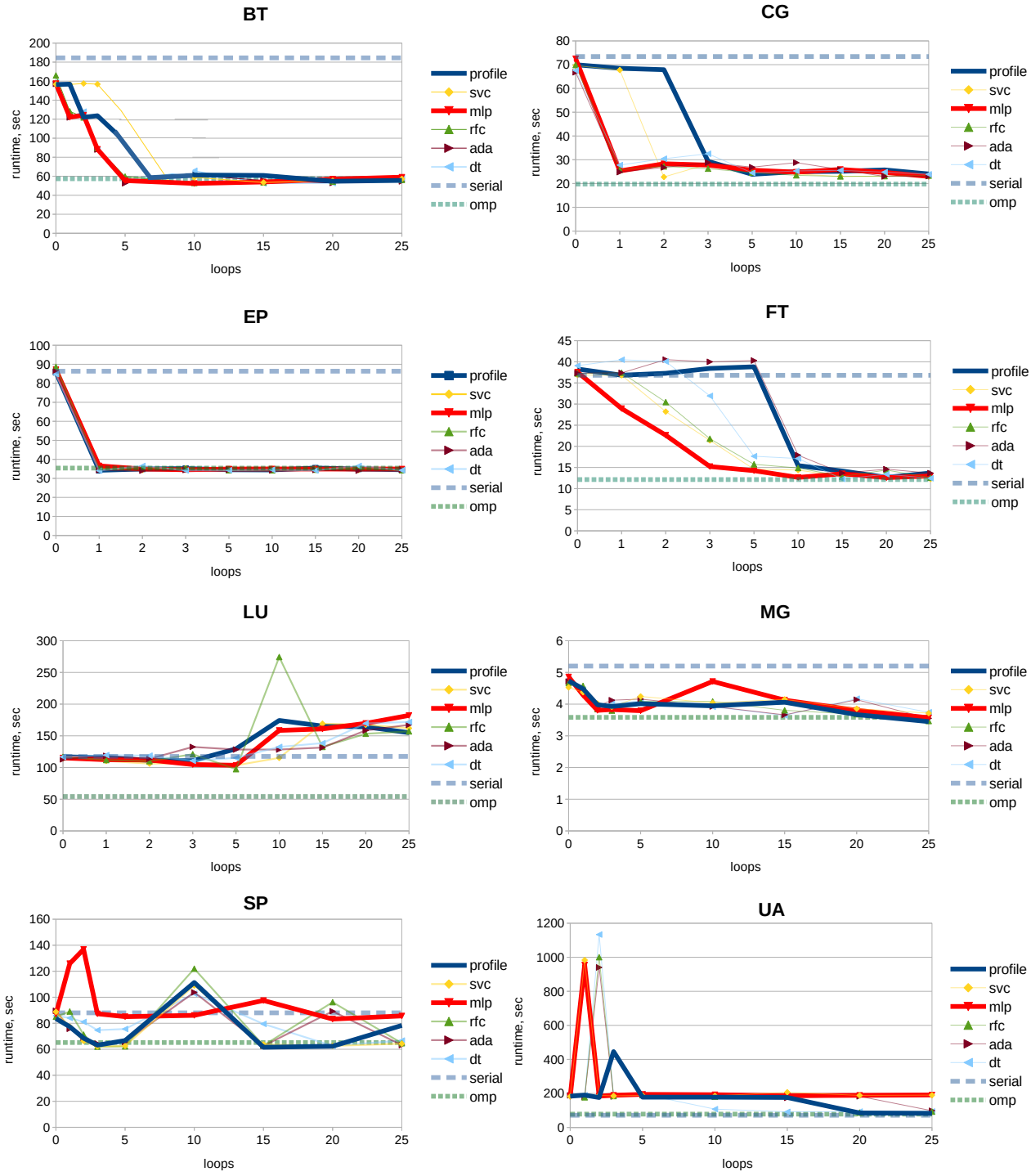

Figure 6. From left to right more loops are parallelized for each benchmark. As we parallelize more loops, program execution times improve over the initial sequential performance and reach the performance level of the reference OpenMP implementations. Our ML based parallelization assistant requires the user to parallelize fewer loops than a purely profile-guided approach to reach the maximum parallel speedup.

For the BT benchmark the slowdown reaches 3.5 times. In contrast, the OpenMP reference implementation results in a (geo-)mean speedup of 2.19 across the benchmark suite.

Figure 6 summarizes our results as performance convergence curves. For each benchmark the curves plot its execution time (y-axis) as a function of the number of analyzed and possible parallelized loops (x-axis). The runtime is bounded within the runtime of the serial execution (top dashed line) and the time of the reference parallel version (bottom dashed line). Our goal is to reach the performance of the reference parallel versions of each program by parallelizing them one loop at a time following the rankings offered by our assistant and the profiler. The neural network based MLP model of our assistant provides the fastest overall performance convergence. While there is some variation depending on the ML model used for the parallelization prediction, in general ML-assisted parallelization outperforms or equals the profile-guided schemes across all benchmarks.

Following the rankings of our assistant in parallelizing the BT, CG and FT benchmarks, we reach their maximum potential performance faster. For BT, maximum parallel performance can be reached after the user has parallelized the first three loops (3061 LOC in Table 7) suggested by our assistant, while profile-guided parallelization requires 6 loops (6122 LOC) to be parallelized first before reaching the same performance level. For CG, if we follow the suggestion of our assistant to first parallelize a small loop (6 LOC), we are able to achieve 70% of the maximum potential speedup (see Section 1.1). On the other hand, using the profiler ranking requires examining three loops, totalling in 330 LOC, to yield the same performance gains. Moreover, for the SP and UA benchmarks, some of the assistant rankings require the programmer to examine more loops than the profiler. However, the loops proposed by the assistant in the UA benchmark are actually simpler, since they consist of fewer LOC – 508 LOC for MLP and 579 for DT versus the 882 LOC offered by the profiler. By following our assistant's suggestions, a programmer would be required to examine 20% less LOC on average across all models.

In some cases, partial benchmark parallelization might result in a slowdown. In the case of LU, after having parallelized the first 25 loops we do not converge to the best achievable parallel version performance. There are a total of 40 OpenMP pragmas in the benchmark and we need to parallelize all the respective loops to reach the best performance level. In the case of UA, all rankings suggest analyzing a long-running innermost loop first. Its parallelization actually increases running time due to a synchronization barrier being introduced at a wrong program point. It takes 30 loops for the MLP model to achieve the parallel version performance.

Finally, we observe that neither our parallelization assistant nor the profiler reach the performance of the reference OpenMP versions on the DC and IS benchmarks. Manual inspection reveals that these benchmarks have been parallized using OpenMP parallel sections, but do not contain any OpenMP parallel loops. Both our parallelization assistant and the profiler incorrectly suggest to parallelize some of the benchmark loops, though. Table 7 summarizes the results of applying our parallelization assistant to the SNU NPB suite.

6 Related Work

Profitability Analysis. The SUIF [22] parallelizing compiler uses a simple heuristic based on the product of statements per loop iteration and number of loop iterations to decide whether a parallelizable loop should be scheduled to be executed in parallel. In contrast, Tournavitis *et al.* [21] use a machine learning based heuristic, which incorporates *dynamic* program features collected in a separate profiling stage, to decide if and how a potentially parallel loop should be scheduled across multiple processors.

ML in Compiler Optimization. Machine learning has been used to solve a wide range of problems, from the early successful work of selecting compiler flags for sequential programs, to recent works on scheduling and optimizing parallel programs on heterogeneous multi-cores. Some works for machine learning in compilers look at how, or if, a compiler optimization should be applied to a sequential program. Some of the previous studies build supervised classifiers to predict the optimal loop unroll factor [13, 20] or to determine whether a function should be inlined [3, 23]. These works target a fixed set of compiler options, by representing the optimization problem as a multi-class classification problem where each compiler option is a class. Evolutionary algorithms like generic search are often used to explore a large design space. Prior works [1, 2, 4] have used evolutionary algorithms to solve the phase ordering problem (i.e. in which order a set of compiler transformations should be applied).

Machine Learning and Parallelization. Most relevant to the work presented in this paper is the approach of Fried *et al.* [7]. Similar to our approach, Fried *et al.* train a supervised learning algorithm on code hand-annotated with OpenMP parallelization directives in order to approximate the parallelization that might be produced by a human expert. However, we do not rely solely on OpenMP annotations, but we complement our training set with substantially richer data obtained from an aggressively configured parallelizing compiler. While Fried *et al.* focus on the comparative performance of different ML algorithms, we contribute a practical parallelization assistant capable of ranking loop candidates in their order of merit. Through this we directly enhance programmer productivity in an ML-assisted environment. Similarly to our approach, Hayashi *et al.* [8] extracts various program features during compilation for use in a supervised learning prediction model. However, its aim is the optimal runtime selection of CPU vs. GPU execution and it is limited

Table 7. SNU NPB benchmark parallelization reports. The left part of the table shows execution times of serial, OpenMP and partially parallelized (critical) versions. The partially parallelized versions have only several critical (top ranked) loops parallelized. The right hand part of the table shows the number of top-ranked loops one needs to parallelize in order to reach the critical performance. The Profile column gives the reference number a profiler requires. The total lines of code (LOC) in the loops are written down as underscript. In most cases, ML based models converge to the critical performance faster than a profiler based approach (highlighted with green). Red cells show the cases where a profiler outperforms our assistants.

Bench.	Bench. Runtime, *sec*			Speedup, *times*		*Loops Number*$_{LOC}$					
	Serial	OpenMP	Critical	OpenMP	Critical	Profile	SVC	MLP	RFC	AdaBoost	DT
BT	158.76	57.36	56.57	2.77	2.81	6_{6122}	8_{6392}	4_{4088}	5_{5105}	3_{3061}	5_{5105}
CG	69.38	19.77	25.06	3.51	2.77	3_{330}	2_{118}	1_{6}	1_{6}	1_{6}	1_{6}
DC	698.82	254.29	698.82	2.75	1.00	∞	∞	∞	∞	∞	∞
EP	86.35	35.40	35.07	2.44	2.46	1_{45}	1_{45}	1_{45}	1_{45}	1_{45}	1_{45}
FT	36.81	12.13	14.69	3.03	2.51	9_{338}	4_{187}	3_{140}	4_{187}	9_{338}	5_{193}
IS	4.75	1.35	4.63	3.53	1.03	∞	∞	∞	∞	∞	∞
LU	115.46	55.00	140.53	2.10	0.82	∞	∞	∞	∞	∞	∞
MG	5.20	3.58	3.94	1.45	1.32	3_{43}	3_{43}	3_{43}	3_{43}	3_{43}	3_{43}
SP	86.65	65.19	62.90	1.33	1.38	3_{801}	3_{801}	∞	3_{801}	3_{801}	20_{1257}
UA	71.82	78.56	189.66	0.91	0.38	19_{882}	30_{918}	30_{508}	19_{861}	22_{883}	10_{579}

to programs written in Java using the parallel stream APIs that was introduced in version 8.

7 Summary & Conclusions

In this paper we contribute a methodology and tool for parallelization assistance. We acknowledge that parallelization is a complex process where the human expert still has a major role to play. We aim to assist the human experts by guiding them directly towards the most interesting loops, thus delivering savings for this costly human resource. We have developed a novel machine learning based approach to predicting whether or not a loop is parallelizable. We combine this prediction with traditional profiling information and develop a ranking function that prioritizes low-risk, high-gain loop candidates, which are presented to the user.

We have evaluated our parallelization assistant against the sequential C implementations of the SNU NPB suite. We show that our assistant recognizes parallelizable loops more aggressively than conservative parallelizing compilers. We also show that our parallelization assistant can increase programmer productivity. Our experiments confirm, that equipped with our assistant, a programmer is required to examine and parallelize substantially fewer loops to achieve performance levels comparable to those of the reference OpenMP implementations of the benchmarks.

Our work has demonstrated that there is scope for machine learning based tool support in parallelization despite its inherent lack of safety. By assisting human programmers rather than replacing them, machine learning techniques have the potential to deliver productivity gains beyond what is possible by relying on traditional parallelization approaches alone.

Acknowledgements

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism. We would like to thank Artemiy Margaritov and Kuba Kaszyk for their valuable feedback on earlier drafts of this paper. Finally, we would like to thank the anonymous reviewers for their comments and suggestions.

References

[1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding Effective Compilation Sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*. ACM, New York, NY, USA, 231–239. https://doi.org/10.1145/997163.997196 8

[2] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* 14, 3, Article 29 (Sept. 2017), 28 pages. https://doi.org/10.1145/3124452 8

[3] J. Cavazos and M. F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 14–14. https://doi.org/10.1109/SC.2005.14 8

[4] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2005. ACME: Adaptive Compilation Made Efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*. ACM, New York, NY, USA, 69–77. https://doi.org/10.1145/1065910.1065921 8

[5] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313 2

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041 3

[7] Daniel Fried, Zhen Li, Ali Jannesari, and Felix Wolf. 2013. 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*. IEEE Computer Society, Miami, FL, USA, 72–77. https://doi.org/10.1109/ICMLA.2013.108 8

[8] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. 2015. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/2807426.2807429 8

[9] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning : with applications in R*. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-1-4614-7138-7 3

[10] Nicklas Bo Jensen and Sven Karlsson. 2017. Improving Loop Dependence Analysis. *ACM Trans. Archit. Code Optim.* 14, 3, Article 22 (Aug. 2017), 24 pages. https://doi.org/10.1145/3095754 2

[11] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 3

[12] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. 2012. Parallelizing More Loops with Compiler Guided Refactoring. In *Proceedings of the 2012 41st International Conference on Parallel Processing (ICPP '12)*. IEEE Computer Society, Washington, DC, USA, 410–419. https://doi.org/10.1109/ICPP.2012.48 1

[13] H. Leather, E. Bonilla, and M. O'Boyle. 2009. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *2009 International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 81–91. https://doi.org/10.1109/CGO.2009.21 8

[14] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. 2018. Generalized Profile-guided Iterator Recognition. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 185–195. https://doi.org/10.1145/3178372.3179511 2, 3

[15] Aleksandr Maramzin. 2019. *Machine Learning Based Parallelization Assistant*. The University of Edinburgh. https://github.com/av-maramzin/PParMetrics 3

[16] S. Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool, San Rafael, CA, USA. https://ieeexplore.ieee.org/document/6813266 1

[17] Alexandru Niculescu-Mizil and Rich Caruana. 2005. Predicting Good Probabilities with Supervised Learning. In *Proceedings of the 22nd International Conference on Machine Learning (ICML '05)*. ACM, New York, NY, USA, 625–632. https://doi.org/10.1145/1102351.1102430 3

[18] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. 3

[19] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC '11)*. IEEE Computer Society, Washington, DC, USA, 137–148. https://doi.org/10.1109/IISWC.2011.6114174 2

[20] M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 123–134. https://doi.org/10.1109/CGO.2005.29 8

[21] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. 2009. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 177–187. https://doi.org/10.1145/1542476.1542496 8

[22] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.* 29, 12 (Dec. 1994), 31–37. https://doi.org/10.1145/193209.193217 8

[23] Peng Zhao and José Nelson Amaral. 2004. To Inline or Not to Inline? Enhanced Inlining Decisions. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer, Berlin, Heidelberg, 405–419. https://doi.org/10.1007/978-3-540-24644-2_26 8