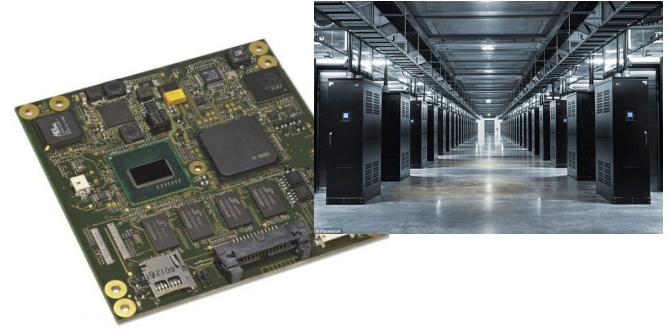


# Manual Software Parallelisation Assistant

*(machine learning based loop parallelisability predictor)*

# Problem

- Abundance of highly-parallel hardware
- Vast amounts of sequential legacy software



- Fully-automated parallelisation methods have to be conservative and mostly fail on general purpose applications



- Manual parallelisation requires expertise, knowledge and skills

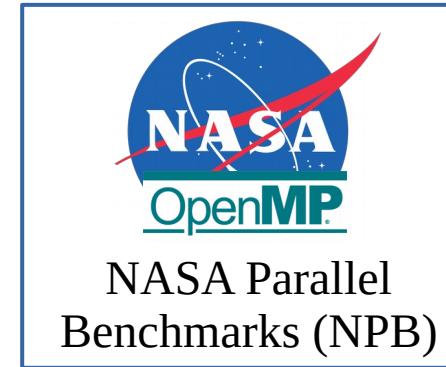
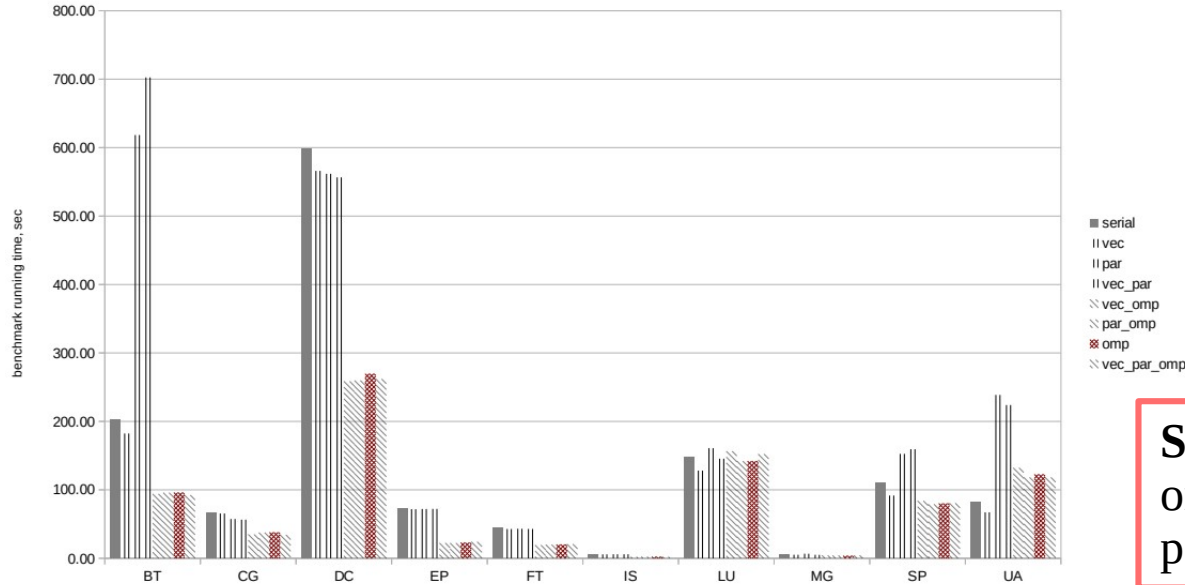


# Project Aims and Objectives

**Facilitate the task of manual software parallelisation by providing a programmer with a feedback and directions**

- These are not fully-automated methods, which rely on the final programmer approval
- Our methods are machine learning (ML) based and rely on structural properties of Program Dependence Graph (PDG) as ML features

# Project Motivation [1]



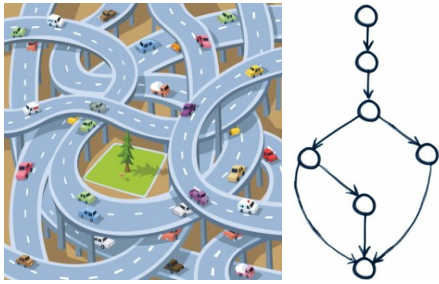
**SNU NPB:** Intel Compiler can discover only 82% of all available in SNU NPB parallelism

**SNU NPB:** Intel Compiler's automatic parallelisation fails to outperform manually parallelised developer OpenMP version

# Project Motivation [2]

## Software Source Code Quality Metrics

### Cyclomatic Complexity (CC)



- Lines Of Code (LOC)
- Bugs per line of code



### Halstead's Software Science

For a given problem, Let:

- $\eta_1$  = the number of distinct operators
- $\eta_2$  = the number of distinct operands
- $N_1$  = the total number of operators
- $N_2$  = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary:  $\eta = \eta_1 + \eta_2$
- Program length:  $N = N_1 + N_2$
- Calculated program length:  $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume:  $V = N \times \log_2 \eta$
- Difficulty:  $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort:  $E = D \times V$

## Software Parallelisability Metrics

$$speedup = \frac{\text{serial execution time}}{\text{parallel execution time}}$$

Different variations of speedup metric:

- **Relative speedup** (serial algorithm vs parallel algorithm)
- **Analytical speedup**
- **Asymptotic speedup** ( $O_{\text{serial}}(n)/O_{\text{parallel}}(n)$ )

Provide a software engineer with a feedback and assistance in the task of **manual software parallelisation**

# **Loop Parallelisability Metrics**

Quantitative Features for Machine Learning Application

# Program Dependence Graph and Iterator Recognition [1]

- **Program Dependence Graph (PDG)** is an IR graph, representing both data and control dependencies between instructions
  - **Generalized Loop Iterator** is a Strongly Connected Component on a PDG with no incoming dependence edges in the Component Graph
- 
- **The Program Dependence Graph and Its Use in Optimization.** Jeanne Ferrante (IBM T. J. Watson Research Center), Karl J. Ottenstein (Michigan Technological University), Joe D. Warren (Rice University). ACM Transactions on Programming Languages and Systems, 1987.
  - **Generalized Profile-Guided Iterator Recognition.** Stanislav Manilov, Christos Vasiladiotis, and Björn Franke (The University of Edinburgh). Proceedings of 27th International Conference on Compiler Construction (CC'18).

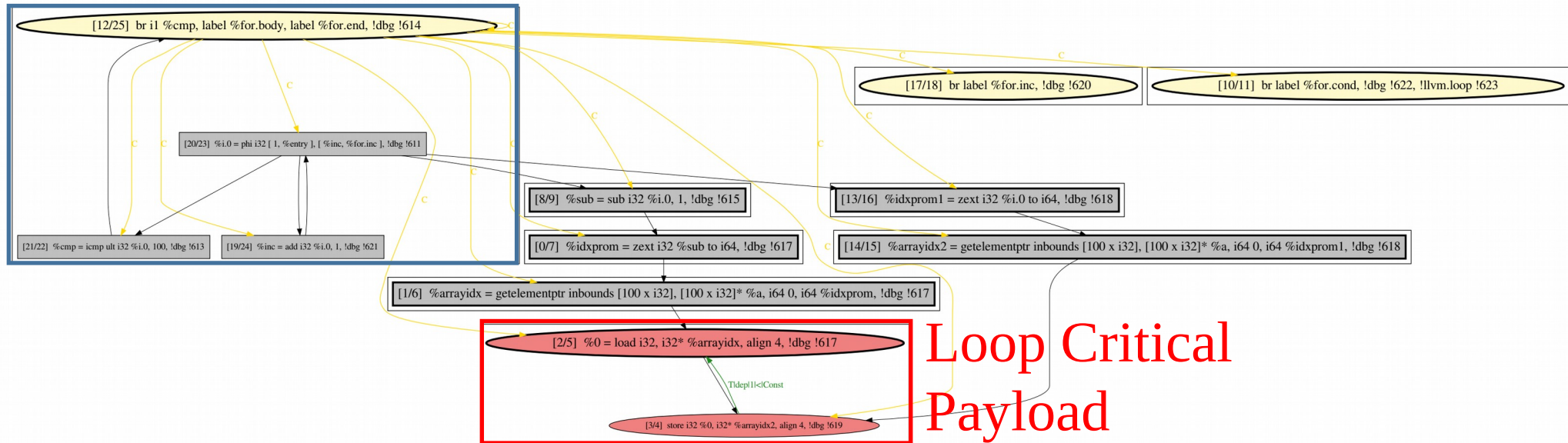
# Program Dependence Graph and Iterator Recognition [2]



```
unsigned int a[100];
```

```
for (unsigned int i = 1; i < 100; i++) {  
    a[i] = a[i-1];  
}
```

## Loop Iterator



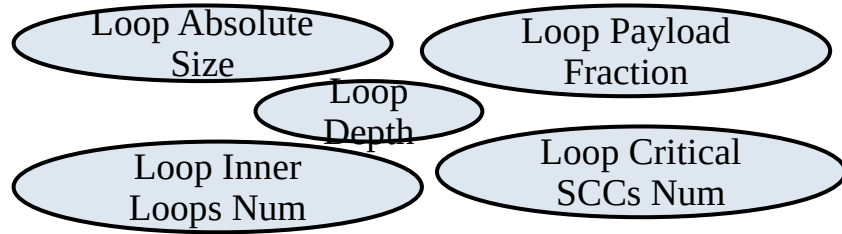


# Static PDG Based Parallelisability Metrics [75]

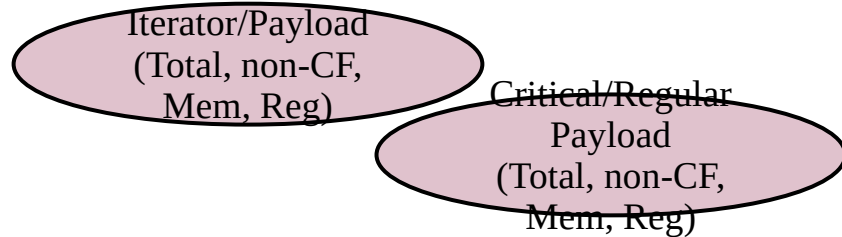
Metric Group	Metric	Metric Definition	Intuition
Loop Proportions	Absolute Size	<i>Number of LLVM IR instructions in a whole loop</i>	The bigger the loop, the harder it is to parallelize it
	Payload Fraction		The smaller the payload fraction (hence, the more complex iterator is), the harder it is to parallelize a loop
	Proper SCCs Number	<i>Number of SCCs with more than one LLVM IR instruction in a payload of a loop</i>	The more proper SCCs we have, the harder this loop is for parallelization
	Critical Payload Fraction		The bigger the critical part of a loop, the harder it is to parallelize a loop
Loop Dependencies Number	Payload Dependencies Number	<i>Number of PDG edges in a payload (<b>True</b>, <b>Anti</b>, <b>Output</b> and <b>Total</b>)</i>	The more dependencies we have, especially in the critical part of a loop payload, the harder it is to parallelize a loop
	Critical Payload Dependencies Number	<i>Number of PDG edges in a critical payload (<b>True</b>, <b>Anti</b>, <b>Output</b> and <b>Total</b>)</i>	
Loop Cohesion	Iterator/Payload Cohesion		No apparent intuition
	Critical/Regular Payload Cohesion		The tighter a regular payload is coupled with payload's critical part (more edges in between, bigger cohesion value), the harder it is to parallelize a loop
Loop Instruction Nature	Call instructions count	<i>Number of call instructions in iterator, payload and critical payload of a loop</i>	Uninlined calls prevent loop parallelisation
	Branch instructions count	<i>Number of branch instructions in iterator, payload and critical payload of a loop</i>	Branch instructions implement breaks, returns and a complicated CF in a loop body, what effectively hinders loop parallelisation

# Static PDG Based Parallelisability Metrics [75]

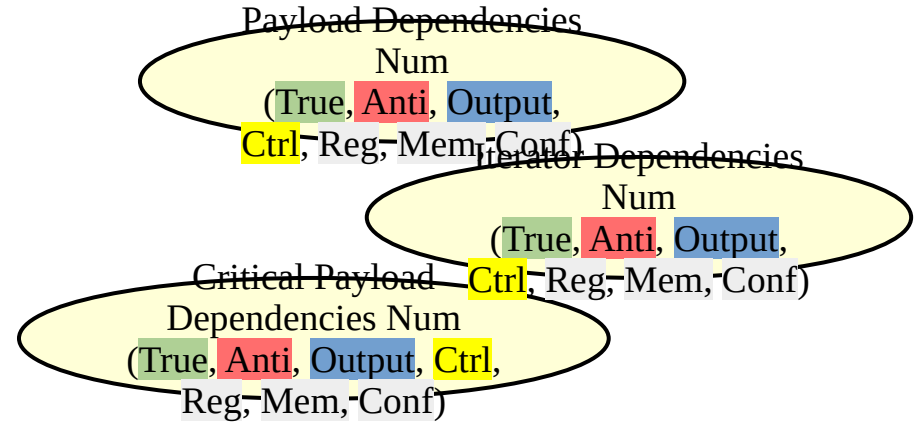
## Loop Proportions [10]



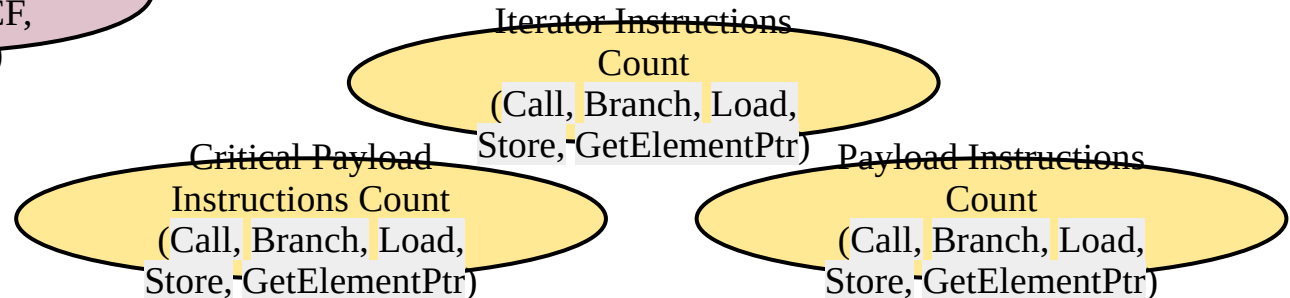
## Loop Cohesion [8]



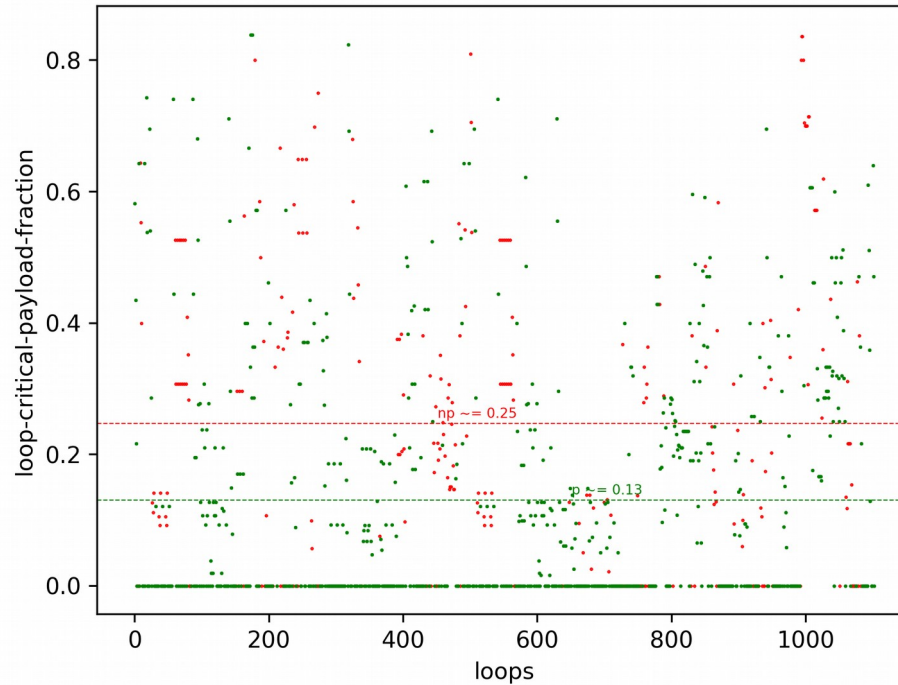
## Loop Dependencies Number [27]



## Loop Instruction Nature [30]



# Single Metric Parallelisability Correlation

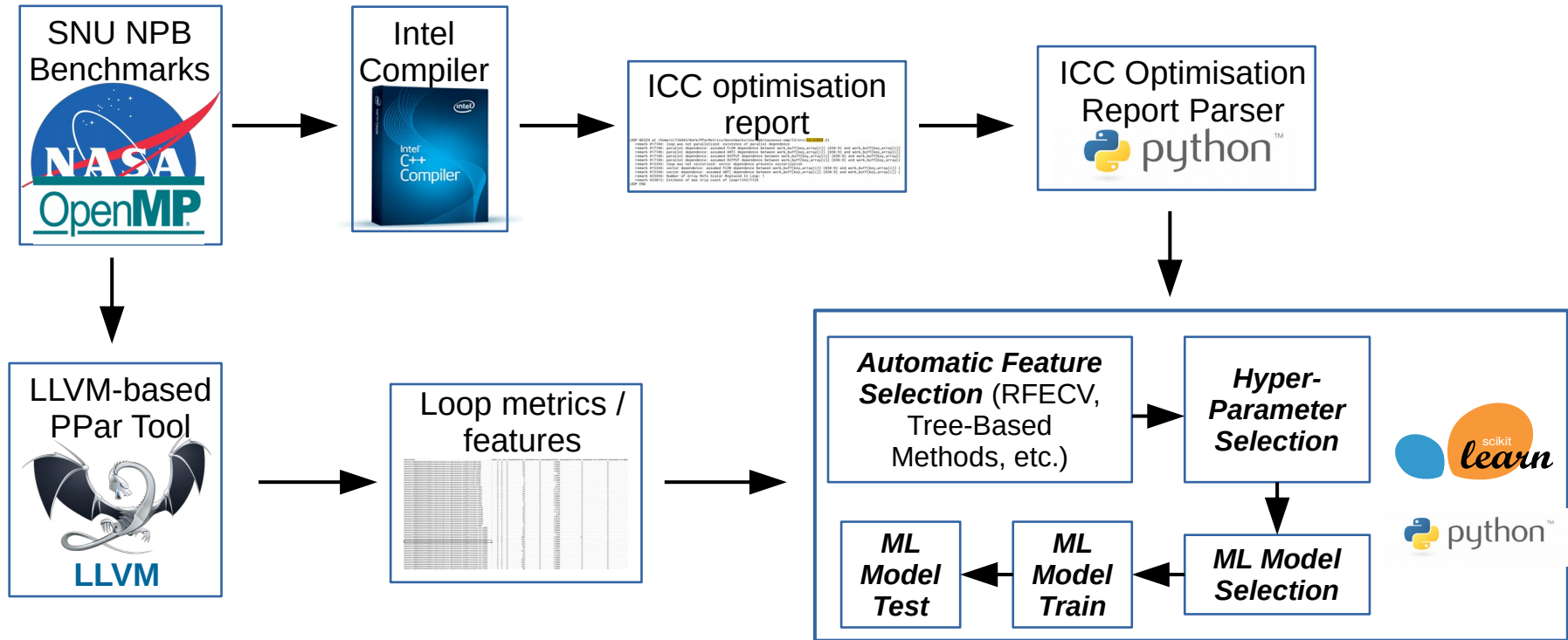


There is no single metric that can be used to precisely separate the sets of parallelisable and non-parallelisable loops

# **Machine Learning Part**

Supervised Classification Problem:  
*Learning Loop Parallelisability Property*

# Machine Learning Methodology: Framework



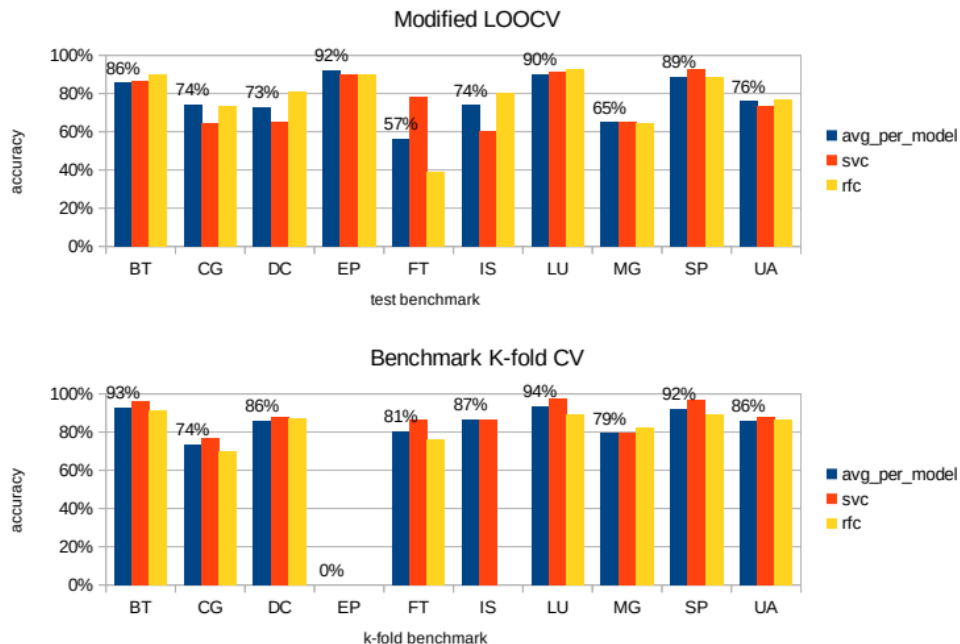
Supervised Classification Problem: classify program loops as parallelisable or not

# Machine Learning Methodology: K-fold CV

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

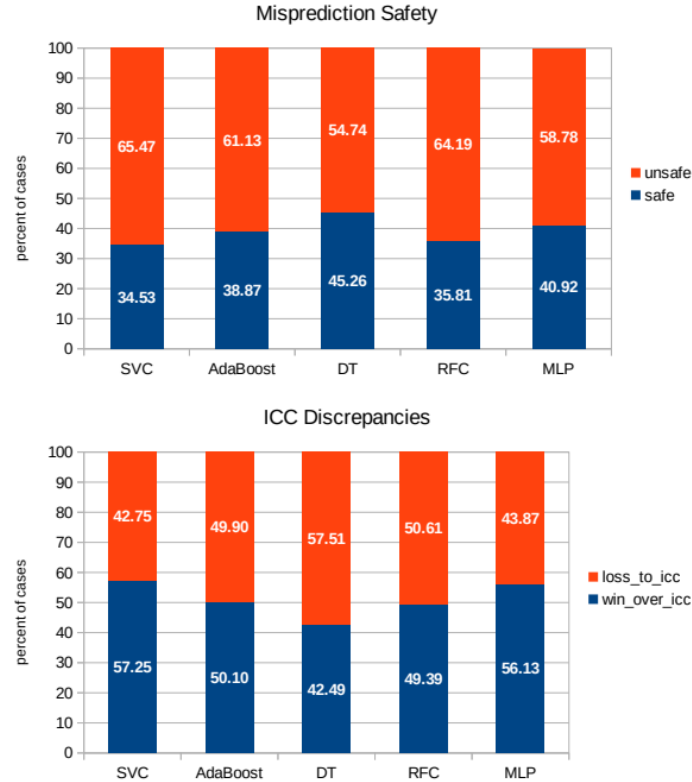
Support Vector Classifier (SVC) and Neural Network based MLP slightly outperform tree based learning algorithms

# Machine Learning Methodology: LOOCV



Lower LOOCV predictive performance can be explained by incomplete training data set and does not reflect the difficulty of learning parallelisability property on a chosen benchmark

# Machine Learning Methodology



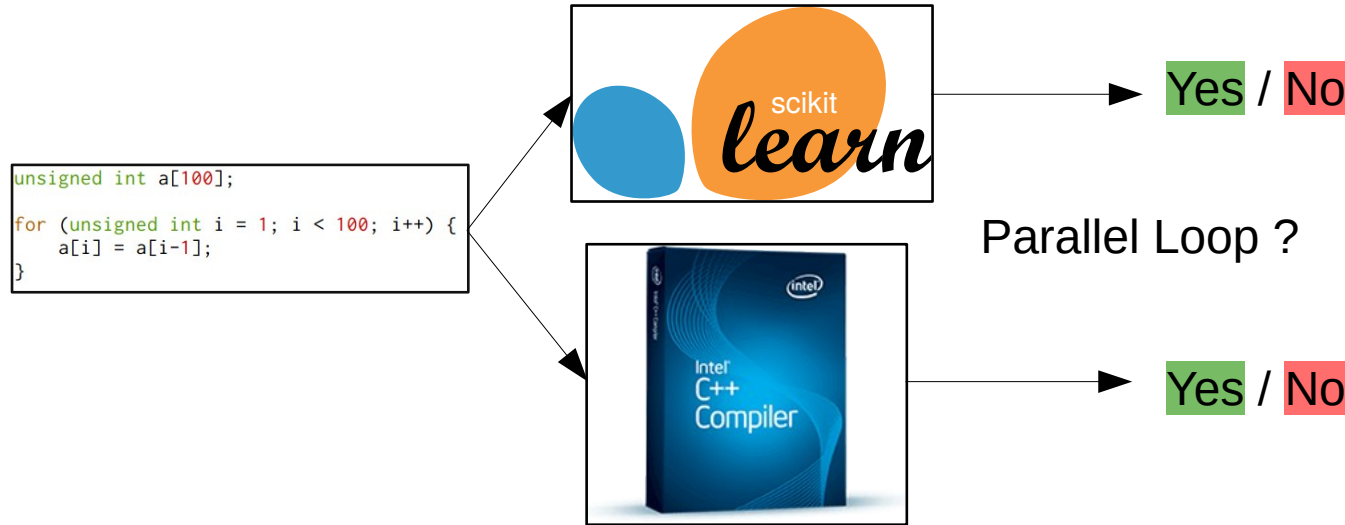
The distribution of misprediction (safe vs. unsafe) as well as Intel Compiler/Predictor disagreement (win vs. lose) cases is roughly 50/50



# **Practical Applications**

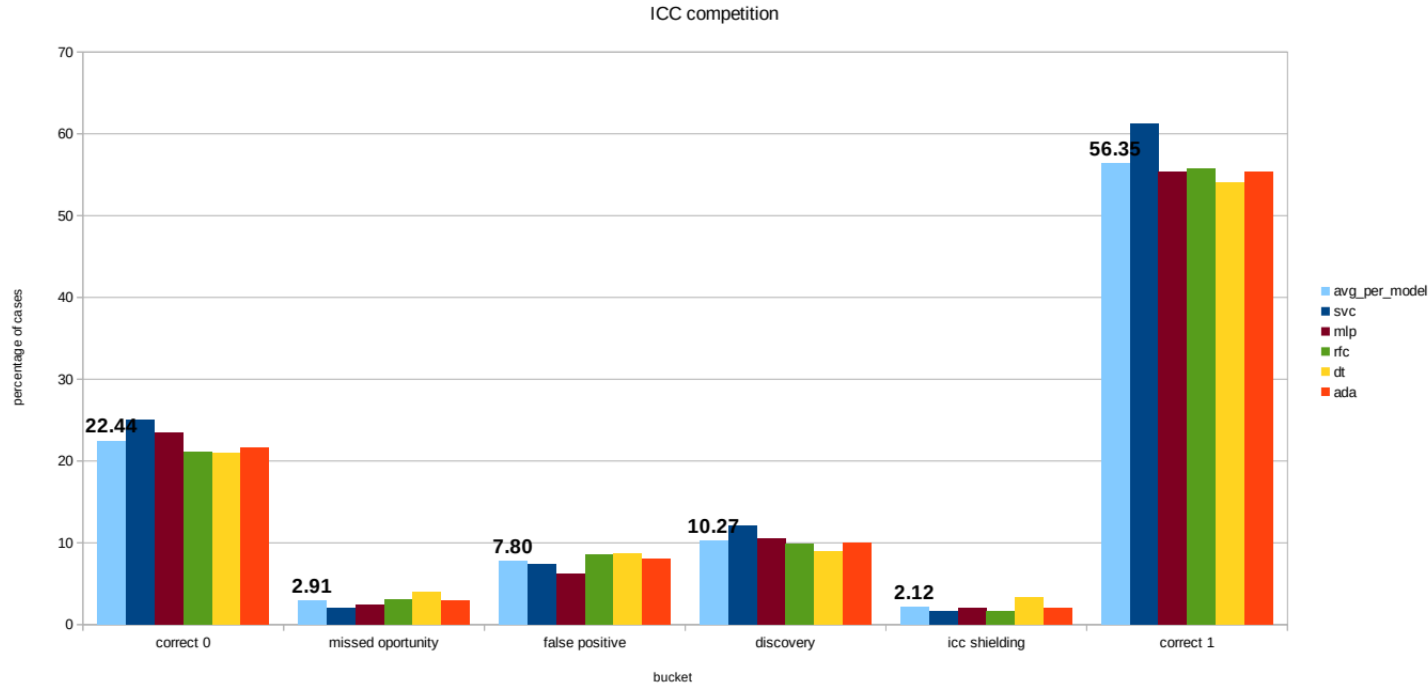
Trained Loop Parallelisability Predictor Use Schemes

# Practical Application [1]: Intel Compiler + Predictor



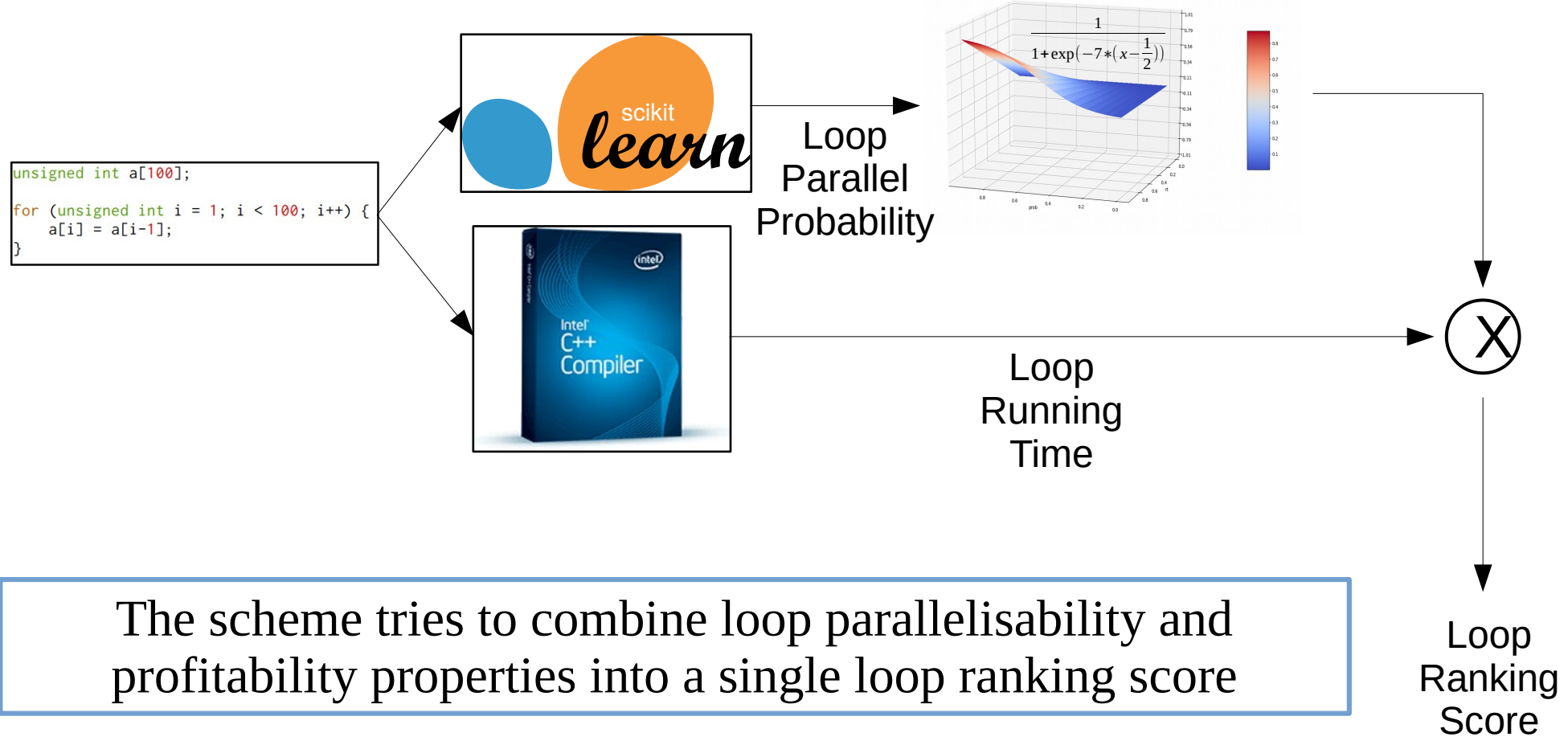
The scheme tries to increase Intel Compiler's parallelism discovery capabilities by augmenting it with a trained parallelisability predictor

# Practical Application [1]: Intel Compiler + Predictor

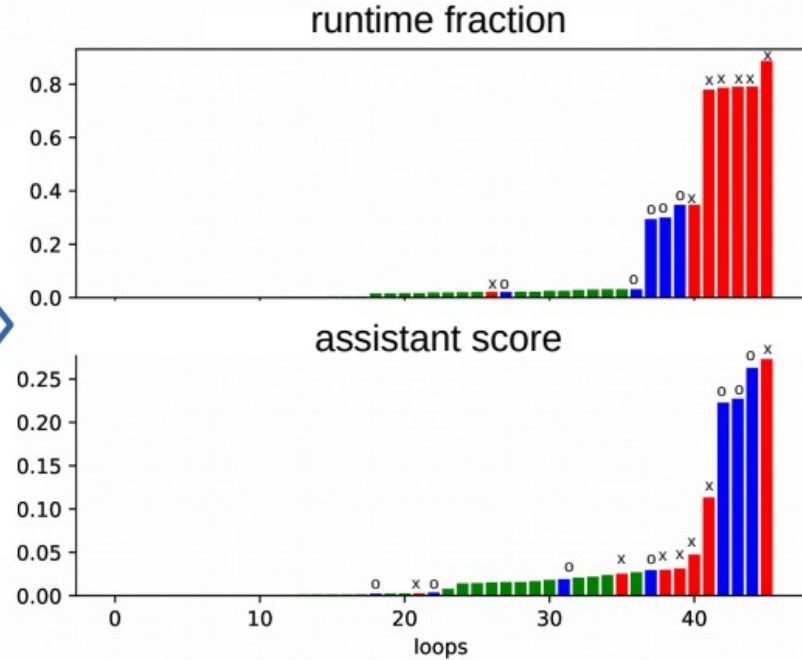
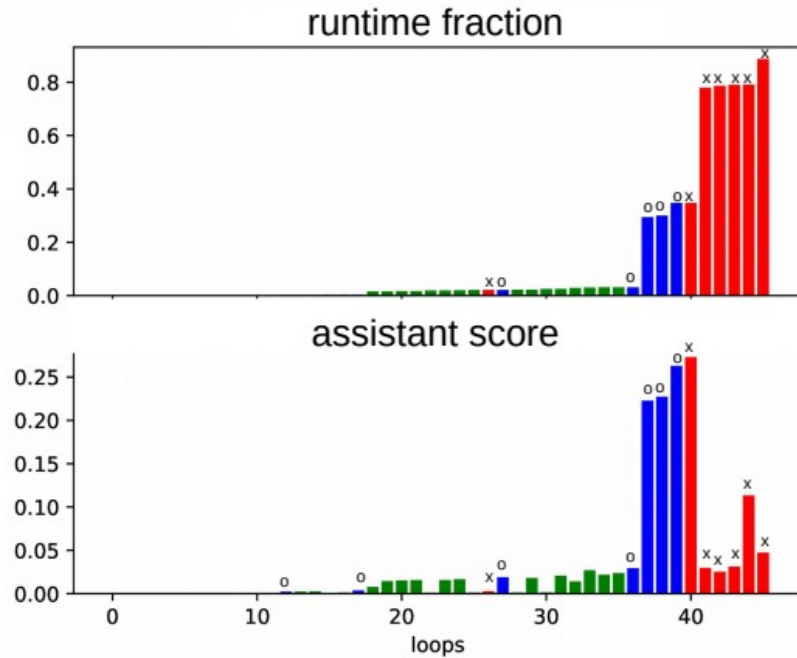


The scheme increases the amount of parallelism discovered in SNU NPB benchmarks from 81,6% to 95,9% (of all available there)

# Practical Application [2]: Loop Parallelisation Assistant

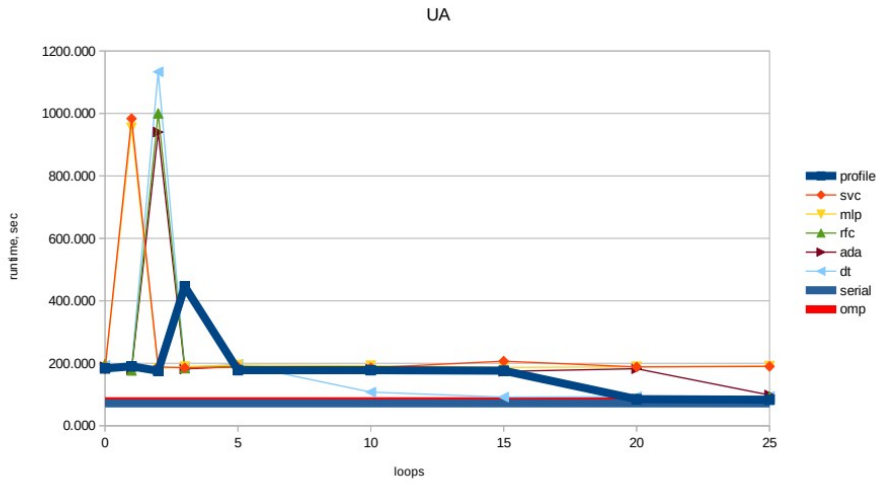
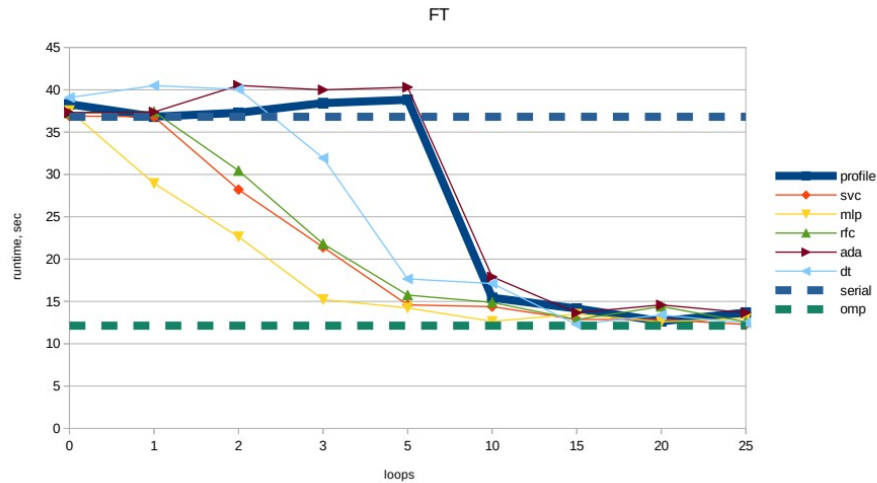
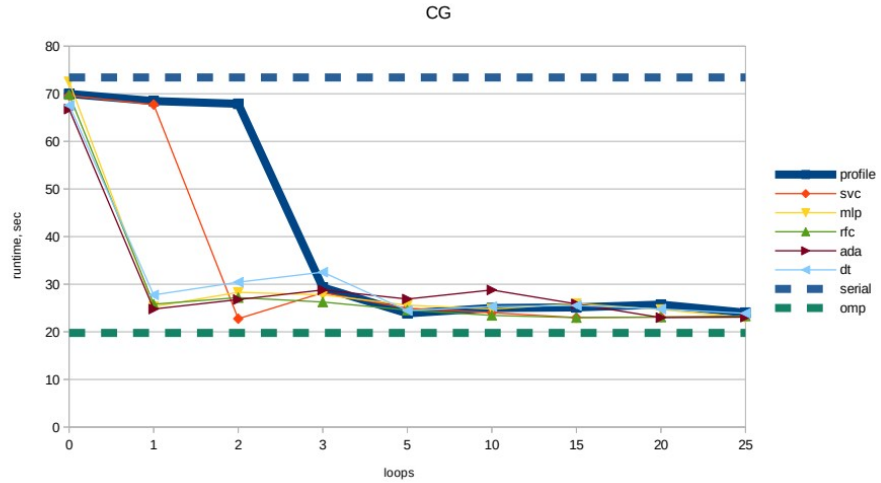
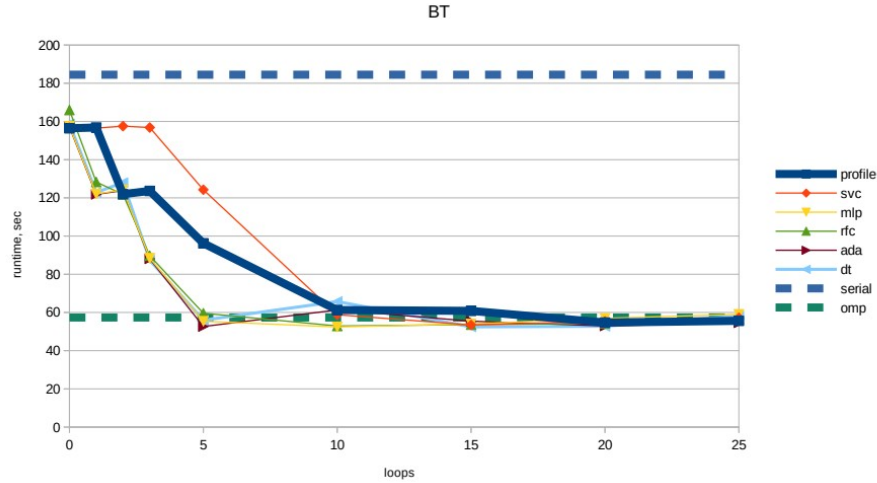


# Practical Application [2]: Loop Parallelisation Assistant



Loop parallelisation assistant reorders a runtime-ordered list of program loops and moves the best loops (parallelisable + profitable) to the beginning of the ordered list

# Practical Application [2]: Loop Parallelisation Assistant



# Summary

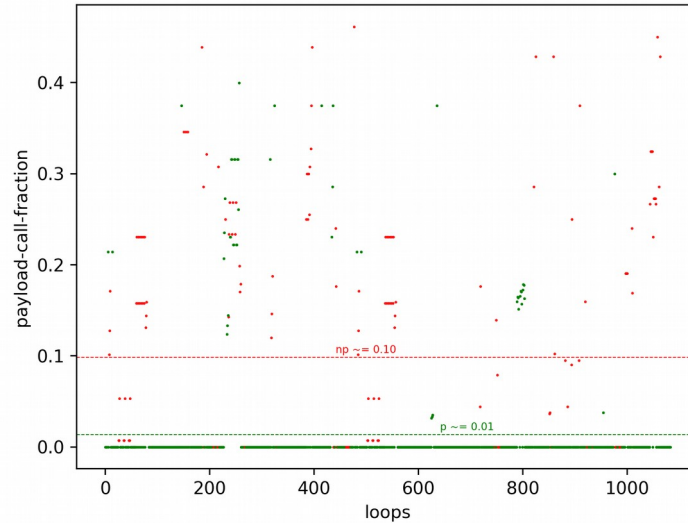
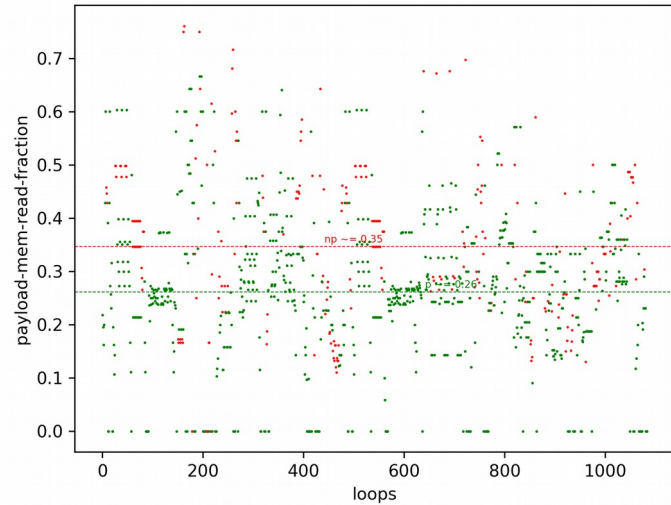
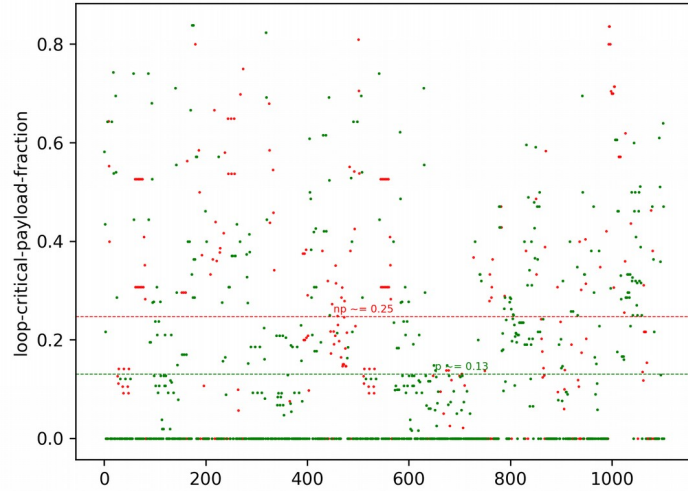
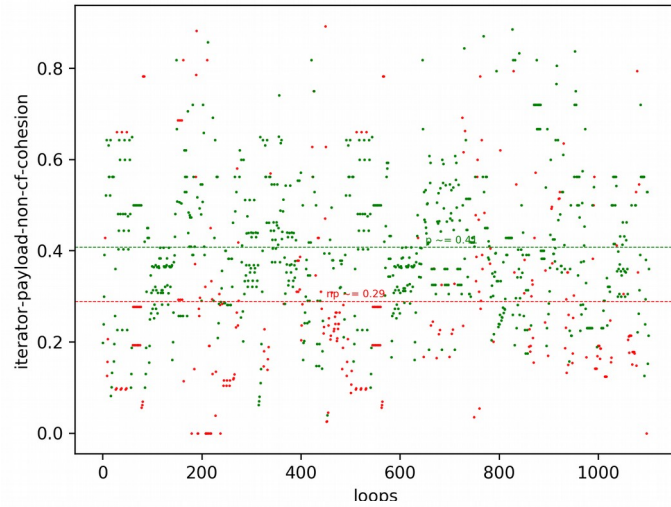
- Implemented LLVM-based C/C++ tool for the extraction of quantitative loop features
  - Implemented a Python parser for the extraction of loop parallelisability labels from Intel Compiler optimisation reports
  - Implemented a SciKit-learn based scripting framework for training and testing ML model of loop parallelisability prediction
- 
- Different Machine Learning algorithms have been successfully applied to the problem of Loop Parallelisability prediction reaching an average of 90% predictive accuracy (against 40-70% baselines)
  - Proposed and assessed 2 predictor utilisation schemes, which provide a programmer with a feedback and direct his/her efforts in the task of manual software parallelisation

**Thank you!**

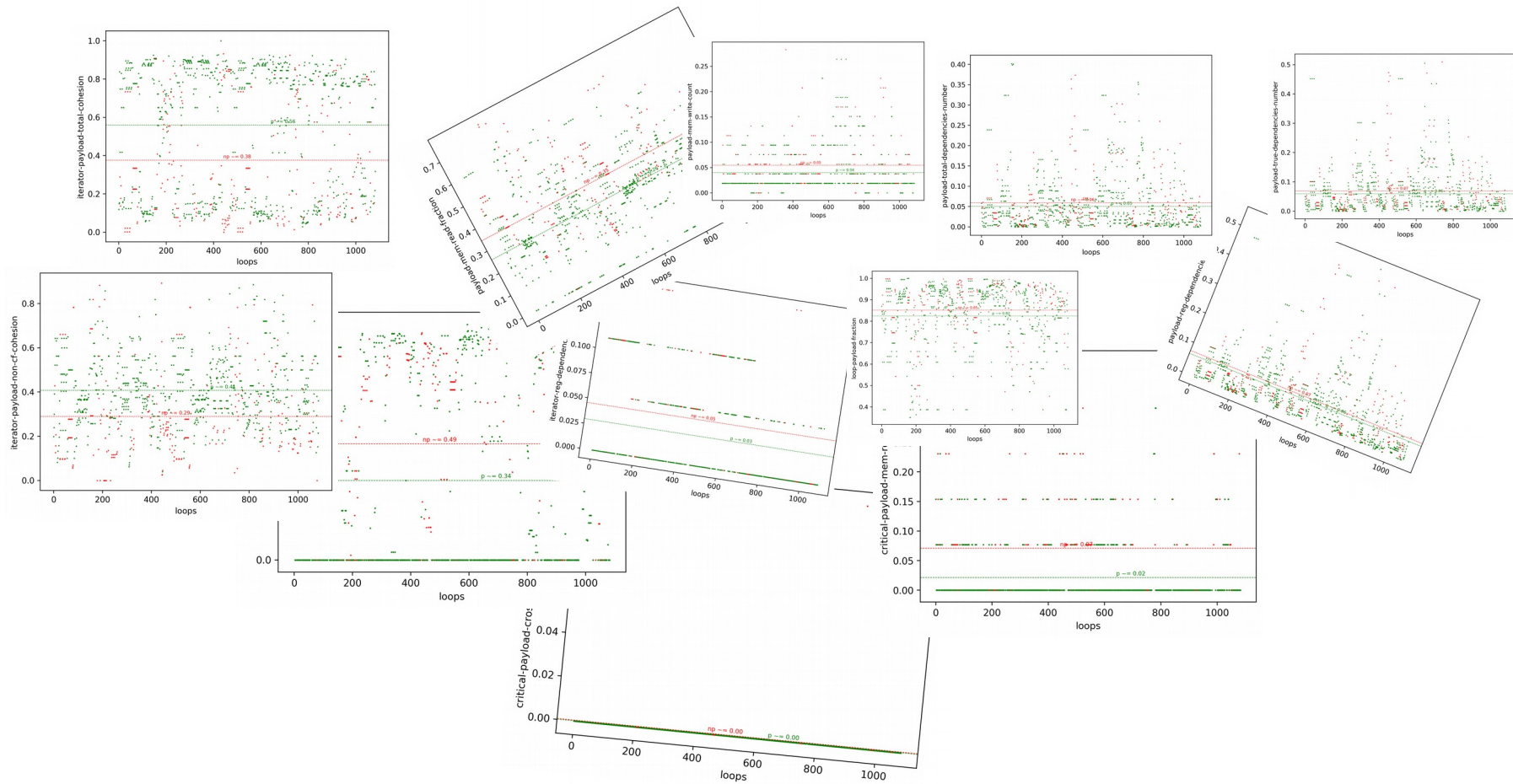


**Backup**

# Software Metrics for Parallelism

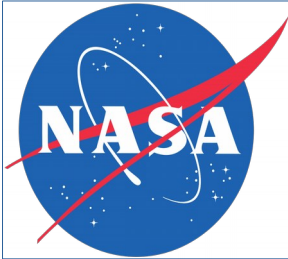


# Software Metrics for Parallelism



# Seoul National University

## NAS Parallel Benchmarks

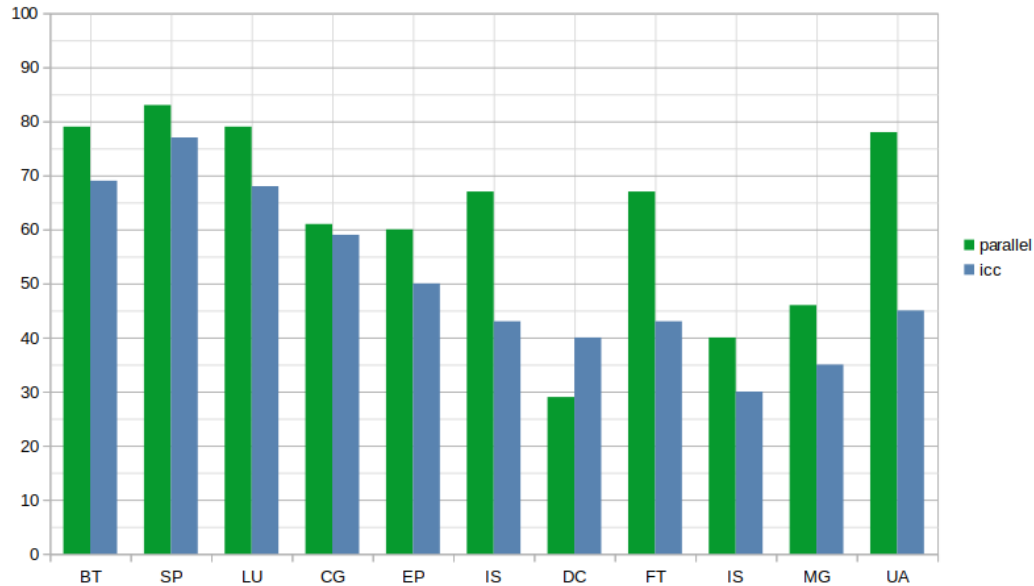
	Seoul National University (SNU) NASA Parallel Benchmarks (NPB)		
	Benchmarks Total Number	Loops Total Number	OpenMP Pragmas Number
	11	1575	211

- NAS Parallel Benchmarks (NPB) are a set of benchmarks targeting performance evaluation of highly parallel supercomputers
- SNU NPB version of benchmarks developed at Seoul National University
- **SNU NPB has sequential as well as manually parallelised OpenMP version**

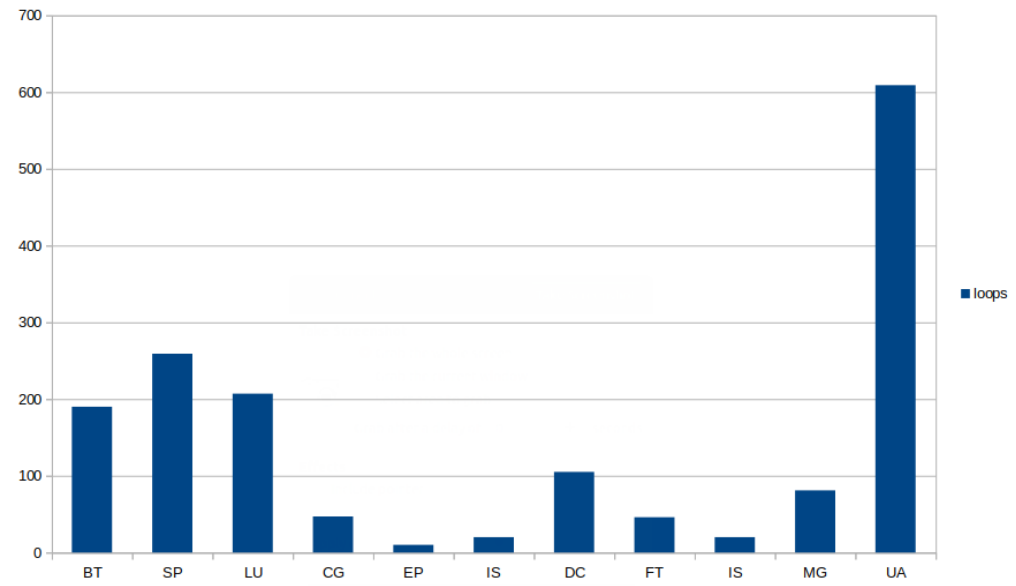
# Seoul National University

## NAS Parallel Benchmarks

### SNU NPB Parallelisability %



### SNU NPB Total Loops Number



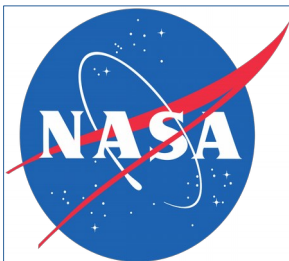
# Intel C/C++ Compiler Parallelisation Limitations (171)

## Potentially solvable limitations

<i>SNU NASA Parallel Benchmarks Intel C/C++ Compiler Limitations (89)</i>					
Unrecognised Reduction	Array Privatization Need	Alias Analysis	Statically unknown number of iterations	Static dependencies	Other
18	3	49	7	11	3

**Difficult to overcome: requires human source code comprehension and benchmark runtime information**

<i>SNU NASA Parallel Benchmarks Intel C/C++ Compiler Limitations (82)</i>					
Static dependencies: runtime information need	Array Privatization Need	Too complex (cross-iteration dependencies, entangled CF, indirect array referencing, etc.)	Alias analysis conservativeness	Uninlined function calls	Other
35	4	22	11	4	1



## Seoul National University (SNU) NASA Parallel Benchmarks (NPB)

Benchmarks Total Number	Loops Total Number	OpenMP Pragmas Number
11	1575	211

### SNU NPB ICC optimisations

Loop Distributions	Loop Fusions	Loop Collapses	Loop Tilings
34	214	58	27

Parallelised	Memset Generated	Vectorised	Parallelisation Dependencies	Not a Parallelisation candidate	Vectorisation Dependencies
653	35	737	535	103	266

### Machine Learning Parallelisability Labels (Parallelisable/Non-Parallelisable)

SNU NPB DEVELOPER KNOWLEDGE	ICC MISSED OPPORTUNITIES	ICC
1145 / 430	1051 / 524	962 / 613

# Decision Tree Model Learning Performance

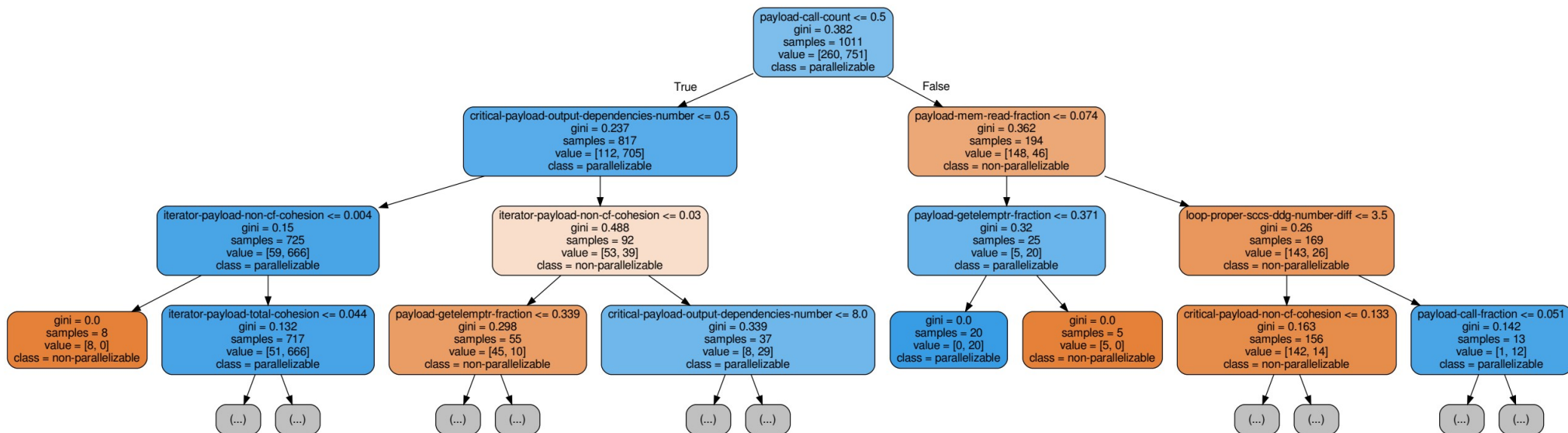
## Machine Learning Accuracy

*(outliers beyond 3 standard deviations have been filtered out)*

Loop Classification Labels	SNU NPB developer knowledge	ICC missed opportunities	ICC
Loops	1420 loops 995 / 425	1420 loops 901 / 519	1420 loops 812 / 608
Baseline (random) predictor accuracy	70%	63.5%	57%
DT accuracy	89-91.6%	90-93%	89-92.5%
Error types (safe/false negatives) (unsafe/false positives)	43-52% 48-57%	42-50% 50-58%	42-49% 51-58%



# Decision Tree Model Learning Performance



# ML based Parallelisation Assistance Tool

*Tool performance with ICC missed opportunities loop classification labels*

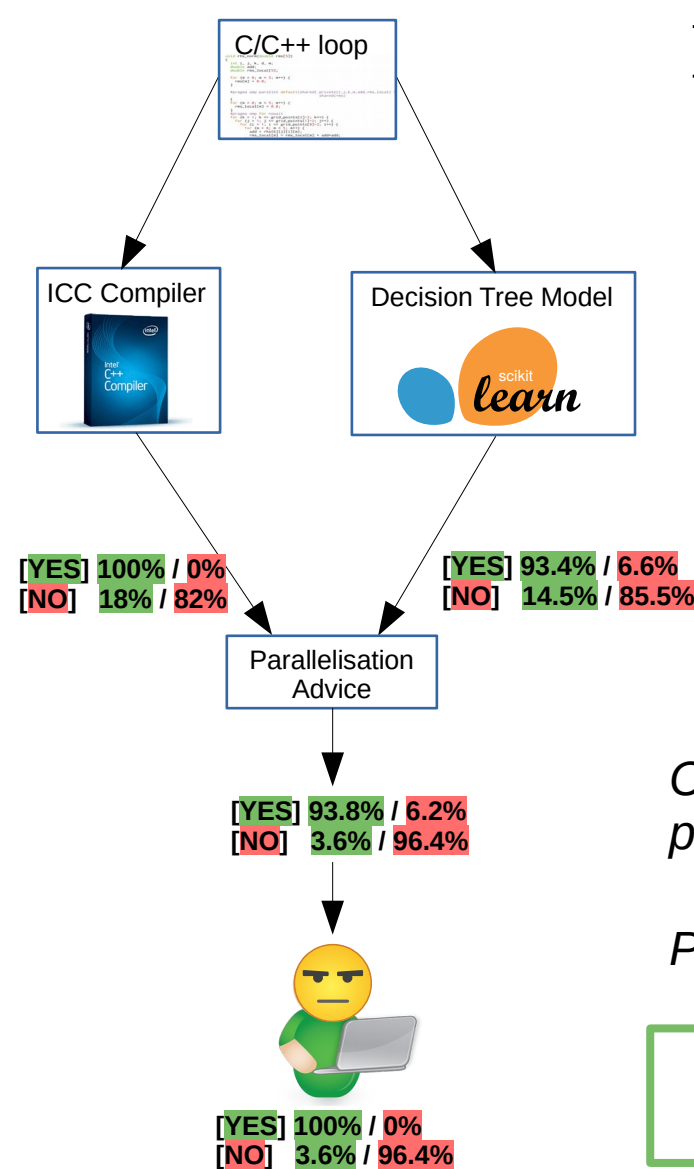
Training / Testing Set Sizes	ICC / ML	Cases Number	Real Parallel	Non-Parallel	Performance 91.6% → 98.9%	
880 / 220 161.6 / 59	0 / 0	49.9	1.8	48.1	ICC	148.1
	0 / 1	22.3	11.4	10.9	ICC+ML	159.8
	1 / 0	6	6.28	0		
	1 / 1	142.1	142.1	0	Real	161.6

\* Results are relatively stable across training/testing sets size variation

*Combined output utilises almost all SNU NPB available parallelism, but introduces unsafe false positives*

*Programmer eliminates unsafe false positives*

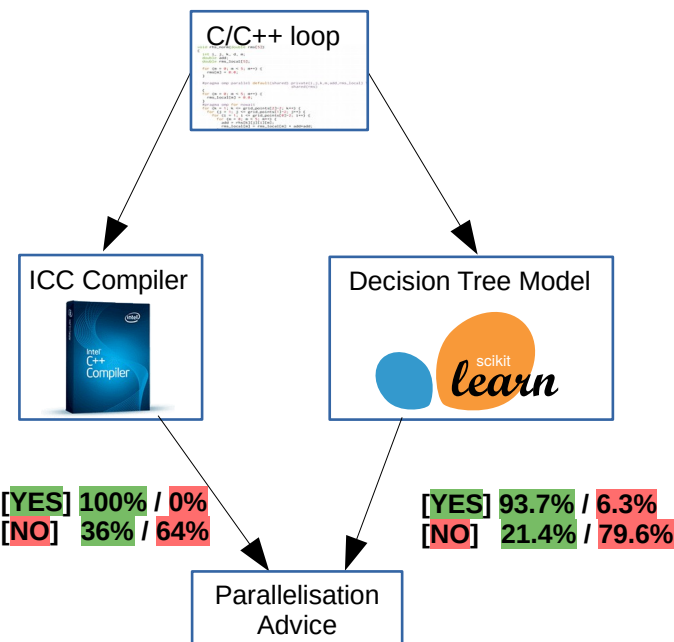
In a perfect use scenario SNU NPB available parallelism utilisation increases on average from 91.5% to 99%



# ML based Parallelisation Assistance Tool

*Tool performance with SNU NPB  
developer loop classification labels*

Training / Testing Set Sizes	ICC / ML	Cases Number	Real Parallel	Non- Parallel	Performance 85.7% → 98%	
867 / 217 173.3 / 43.6	0 / 0	36	3.4	32.6	ICC	148.5
	0 / 1	32.4	21.4	11	ICC+ML	169.9
	1 / 0	5.5	5.5	0		
	1 / 1	143	143	0	Real	173.3



[YES] 100% / 0%  
[NO] 36% / 64%

[YES] 93.7% / 6.3%  
[NO] 21.4% / 79.6%

[YES] 94% / 6%  
[NO] 9.5% / 91.5%



[YES] 100% / 0%  
[NO] 9.5% / 91.5%

*Combined output utilises almost all SNU NPB available  
parallelism, but introduces unsafe false positives*

*Programmer eliminates  
unsafe false positives*

In a perfect use scenario SNU NPB available parallelism  
utilisation increases on average from 86% to 98%



# Backup

- Source code examples
- Additional Tables

# BT, SP, LU benchmarks: array privatization need

```
void error_norm(double rms[5])
{
    int i, j, k, m, d;
    double xi, eta, zeta, u_exact[5], add;
    double rms_local[5];

    for (m = 0; m < 5; m++) {
        rms[m] = 0.0;
    }

    #pragma omp parallel default(shared) \
        private(i,j,k,m,zeta,eta,xi,add,u_exact,rms_local) shared(rms)
    {
        for (m = 0; m < 5; m++) {
            rms_local[m] = 0.0;
        }
        #pragma omp for nowait
        for (k = 0; k <= grid_points[2]-1; k++) {
            zeta = (double)(k) * dnzm1;
            for (j = 0; j <= grid_points[1]-1; j++) {
                eta = (double)(j) * dnym1;
                for (i = 0; i <= grid_points[0]-1; i++) {
                    xi = (double)(i) * dnxm1;
                    exact_solution(xi, eta, zeta, u_exact);

                    for (m = 0; m < 5; m++) {
                        add = u[k][j][i][m] - u_exact[m];
                        rms_local[m] = rms_local[m] + add*add;
                    }
                }
            }
        }
    }
    for (m = 0; m < 5; m++) {
        #pragma omp atomic
        rms[m] += rms_local[m];
    }
} //end parallel
```

```
void exact_solution(double xi, double eta, double zeta, double dtemp[5])
{
    int m;

    for (m = 0; m < 5; m++) {
        dtemp[m] = ce[m][0] +
            xi*(ce[m][1] + xi*(ce[m][4] + xi*(ce[m][7] + xi*ce[m][10]))) +
            eta*(ce[m][2] + eta*(ce[m][5] + eta*(ce[m][8] + eta*ce[m][11]))) +
            zeta*(ce[m][3] + zeta*(ce[m][6] + zeta*(ce[m][9] +
            zeta*ce[m][12]))));
    }
}
```

- Loop nest (k,j,i) accumulates the total error (difference between solution u and the exact solution u\_exact) via a reduction
- Every iteration (k,j,i) uses temporary rms\_local[] memory to compute and store one little portion of the reduction
- This rms\_local[] introduces cross-iteration dependency, since it is being reused on different loop nest iterations
- Programmer replicates this temporary rms\_local[] memory with OpenMP semantics and parallelises the loop nest
- Intel Compiler does not automatically do such a parallelising transformation

# UA benchmark: static dependence (runtime information need)

```
// mt_to_id[miel] takes as argument the morton index and returns the actual
// element index
// id_to_mt(iel) takes as argument the actual element index and returns the
// morton index
#pragma omp parallel for default(shared) private(miel,iel)
for (miel = 0; miel < nelt; miel++) {
    iel = mt_to_id[miel];
    id_to_mt[iel] = miel;
}
```

Output

```
LOOP BEGIN at /home/s1736883/Work/PParMetrics/benchmarks/snu-npb/nauseous-omp/UA/src/adapt.c(149,3)
remark #17104: loop was not parallelized: existence of parallel dependence
remark #17106: parallel dependence: assumed OUTPUT dependence between id_to_mt[mt_to_id[miel]] (151:5) and id_to_mt[mt_to_id[miel]] (151:5)
remark #17106: parallel dependence: assumed OUTPUT dependence between id_to_mt[mt_to_id[miel]] (151:5) and id_to_mt[mt_to_id[miel]] (151:5)
remark #15388: vectorization support: reference mt_to_id[miel] has aligned access [ /home/s1736883/Work/PParMetrics/benchmarks/snu-npb/nauseous-omp/
remark #15329: vectorization support: irregularly indexed store was emulated for the variable <id_to_mt[mt_to_id[miel]]>, part of index is read from m
remark #15305: vectorization support: vector length 8
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15463: unmasked indexed (or scatter) stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 4
remark #15477: vector cost: 20.750
remark #15478: estimated potential speedup: 0.190
remark #15488: --- end vector cost summary ---
remark #25015: Estimate of max trip count of loop=1100
LOOP END
```

- Loop has an output dependency and Intel compiler correctly detects it
- Intel compiler does not parallelise the loop, but can vectorise it
- Programmer knows, that dependence does not materialise and preceeds the loop with an OpenMP pragma

# IS benchmark: static dependence (runtime information need)

```
/* Determine the number of keys in each bucket */  
#pragma omp for schedule(static)  
for( i=0; i<NUM_KEYS; i++ )  
    work_buff[key_array[i] >> shift]++;
```

**True Anti Output**

```
LOOP BEGIN at /home/s1736883/Work/PParMetrics/benchmarks/snu-npb/nauseous-omp/IS/src/is.c(649,5)  
remark #17104: loop was not parallelized: existence of parallel dependence  
remark #17106: parallel dependence: assumed FLOW dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]]  
remark #17106: parallel dependence: assumed ANTI dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]]  
remark #17106: parallel dependence: assumed OUTPUT dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]]  
remark #17106: parallel dependence: assumed OUTPUT dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]]  
remark #15344: loop was not vectorized: vector dependence prevents vectorization  
remark #15346: vector dependence: assumed FLOW dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]] (  
remark #15346: vector dependence: assumed ANTI dependence between work_buff[key_array[i]] (650:9) and work_buff[key_array[i]] (  
remark #25456: Number of Array Refs Scalar Replaced In Loop: 1  
remark #25015: Estimate of max trip count of loop=134217728  
LOOP END
```

- Intel Compiler assumes all static dependencies to materialise and does not proceed with the parallelisation of the loop
- Programmer knows that



# Intel C/C++ Compiler optimisation report parsing and analysis problems

- Intel Compiler uses cost model to decide on how to process parallelisable loops and loop nests (parallelise, vectorise, generate memset/memcpy, leave sequential)
- Has been vectorised != absence of parallelisation restricting dependencies
- A set of transformations applied in order to enable loop parallelisation (loop distribution, fusion, tiling, collapsing, interchange, etc.)
- Loop eliminations (unrolling, inlining,

Machine Learning Parallelisability Labels (Parallelisable/Non-Parallelisable)		
OMP+MANUAL+ICC	MANUAL+ICC	ICC
1145 / 430	1051 / 524	962 / 613

# Seoul National University NAS Parallel Benchmarks

Benchmarks	Brief Description	Benchmark Parallelisability ( <b>ICC+OMP</b> / <b>ICC</b> )	Loops Number	Loops Nature	
				OMP+ ICC	ICC
BT	Mostly all loops are parallelisable by Intel Compiler. Multidimensional arrays with parallel access patterns.	<b>79%</b> / <b>69%</b>	190	<b>150</b> / <b>30</b>	<b>131</b> / <b>59</b>
SP		<b>83%</b> / <b>77%</b>	259	<b>217</b> / <b>42</b>	<b>200</b> / <b>59</b>
LU		<b>79%</b> / <b>68%</b>	207	<b>164</b> / <b>43</b>	<b>141</b> / <b>64</b>
CG	Half of the benchmark are parallelisable reductions (even without OpenMP compiler hints), the other half are	<b>61%</b> / <b>59%</b>	47	<b>29</b> / <b>18</b>	<b>28</b> / <b>19</b>
EP	Small, nothing interesting	<b>60%</b> / <b>50%</b>	10	<b>6</b> / <b>4</b>	<b>5</b> / <b>5</b>
IS	Materialisation of loop cross-iteration dependencies depends on a dynamic program properties. Programmer knows application's runtime behavior and	<b>67%</b> / <b>43%</b>	20	<b>8</b> / <b>12</b>	<b>6</b> / <b>14</b>

# Loops challenging ICC

- Loops with cross-iteration dependencies
- Loops with a statically unknown number of iterations
- Loops with unidentified control variable
- Loops with function calls inside the body
- Loops with multiple exits (break, return statements, etc.)

```

void exact_solution(double xi, double eta, double zeta, double dtemp[5])
{
    int m;

    for (m = 0; m < 5; m++) {
        dtemp[m] = ce[m][0] +
            xi*(ce[m][1] + xi*(ce[m][4] + xi*(ce[m][7] + xi*ce[m][10]))) +
            eta*(ce[m][2] + eta*(ce[m][5] + eta*(ce[m][8] + eta*ce[m][11]))) +
            zeta*(ce[m][3] + zeta*(ce[m][6] + zeta*(ce[m][9] +
            zeta*ce[m][12]))));
    }
}

```

```

LOOP BEGIN at /home/s1736883/Work/PParMetrics/benchmarks/snu-npb/nauseous-omp/BT/src/exact_solution.c(44,3)
remark #17104: loop was not parallelized: existence of parallel dependence
remark #17106: parallel dependence: assumed FLOW dependence between dtemp[m] (45:5) and ce[m][0] (45:5)
remark #17106: parallel dependence: assumed ANTI dependence between ce[m][0] (45:5) and dtemp[m] (45:5)
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between dtemp[m] (45:5) and ce[m][0] (45:5)
remark #15346: vector dependence: assumed ANTI dependence between ce[m][0] (45:5) and dtemp[m] (45:5)
LOOP END

```

# Loop classification problem [1]: what is parallel and what is not?

```
void error_norm(double rms[5])
{
    int i, j, k, m, d;
    double xi, eta, zeta, u_exact[5], add;
    double rms_local[5];

    for (m = 0; m < 5; m++) {
        rms[m] = 0.0;
    }

    #pragma omp parallel default(shared) \
        private(i,j,k,m,zeta,eta,xi,add,u_exact,rms_local) shared(rms)
    {
        for (m = 0; m < 5; m++) {
            rms_local[m] = 0.0;
        }
        #pragma omp for nowait
        for (k = 0; k <= grid_points[2]-1; k++) {
            zeta = (double)(k) * dnzm1;
            for (j = 0; j <= grid_points[1]-1; j++) {
                eta = (double)(j) * dnym1;
                for (i = 0; i <= grid_points[0]-1; i++) {
                    xi = (double)(i) * dnxm1;
                    exact_solution(xi, eta, zeta, u_exact);

                    for (m = 0; m < 5; m++) {
                        add = u[k][j][i][m]-u_exact[m];
                        rms_local[m] = rms_local[m] + add*add;
                    }
                }
            }
        }
    }
    for (m = 0; m < 5; m++) {
        #pragma omp atomic
        rms[m] += rms_local[m];
    }
} //end parallel
```

```
void exact_solution(double xi, double eta, double zeta, double dtemp[5])
{
    int m;

    for (m = 0; m < 5; m++) {
        dtemp[m] = ce[m][0] +
            xi*(ce[m][1] + xi*(ce[m][4] + xi*(ce[m][7] + xi*ce[m][10]))) +
            eta*(ce[m][2] + eta*(ce[m][5] + eta*(ce[m][8] + eta*ce[m][11]))) +
            zeta*(ce[m][3] + zeta*(ce[m][6] + zeta*(ce[m][9] +
            zeta*ce[m][12]))));
    }
}
```

- How to label these loops for a ML-based tool?
- OpenMP semantics is supposed to replicate `rms_local[]` and ensure atomic mutually exclusive access to a shared `rhs[]`
- OpenMP puts in an additional semantics, without which there is a memory dependence
- No dependence in OpenMP version, but memory cross-iteration dependence in the sequential version
- Need to provide memory in `rhs_local[]` for every parallel thread, before we parallelise it. Compiler does not do it automatically, without programmer's instruction
- Function calls scare conservative compiler. Should we classify the code as it is or only after assumed inlining is done?

# BT, SP, LU benchmarks:

```
double dtemp[5], xi, eta, zeta, dtmp;
int m, i, j, k, ip1, im1, jp1, jm1, km1, kp1;
```

```
#pragma omp parallel default(shared) private(i,j,k,m,zeta,eta,xi,\
dtmp,im1,ip1,jm1,jp1,km1,kp1,dtemp)
```

```
{
#pragma omp for schedule(static) nowait
for (k = 1; k <= grid_points[2]-2; k++) {
    zeta = (double)(k) * dnxm1;
    for (j = 1; j <= grid_points[1]-2; j++) {
        eta = (double)(j) * dnym1;
        for (i = 0; i <= grid_points[0]-1; i++) {
            xi = (double)(i) * dnxm1;
```

```
        exact_solution(xi, eta, zeta, dtemp);
```

```
        for (m = 0; m < 5; m++) {
```

```
            ue[i][m] = dtemp[m];
```

```
        }
```

```
        ...
        q[i] = 0.5*(buf[i][1]*ue[i][1] + buf[i][2]*ue[i][2] +
            buf[i][3]*ue[i][3]);
    }
```

```
for (i = 1; i <= grid_points[0]-2; i++) {
```

```
    im1 = i-1;
```

```
    ip1 = i+1;
```

```
    forcing[k][j][i][0] = forcing[k][j][i][0] -
        tx2*( ue[ip1][1]-ue[im1][1] )+
        dx1tx1*(ue[ip1][0]-2.0*ue[i][0]+ue[im1][0]);
    ...
```

```
    ...
```

```
}
```

```
for (m = 0; m < 5; m++) {
```

```
    i = grid_points[0]-3;
```

```
    forcing[k][j][i][m] = forcing[k][j][i][m] - dssp *
        (ue[i-2][m] - 4.0*ue[i-1][m] +
```

```
        6.0*ue[i][m] - 4.0*ue[i+1][m]);
```

```
    i = grid_points[0]-2;
```

```
    forcing[k][j][i][m] = forcing[k][j][i][m] - dssp *
        (ue[i-2][m] - 4.0*ue[i-1][m] + 5.0*ue[i][m]);
```

```
    ...
```

```
}
```

```
}
```

```
}
```

```
void exact_solution(double xi, double eta, double zeta, double dtemp[5])
```

```
{
    int m;
```

```
    for (m = 0; m < 5; m++) {
```

```
        dtemp[m] = ce[m][0] +
```

```
            xi*(ce[m][1] + xi*(ce[m][4] + xi*(ce[m][7] + xi*ce[m][10]))) +
```

```
            eta*(ce[m][2] + eta*(ce[m][5] + eta*(ce[m][8] + eta*ce[m][11]))) +
```

```
            zeta*(ce[m][3] + zeta*(ce[m][6] + zeta*(ce[m][9] +
```

```
            zeta*ce[m][12])));
```

```
    }
```

```
}
```

- ue[PROBLEM\_SIZE+1][5] is a global 2D array, which contains different values as a function of 2 loop nest iteration (k,j)
- Programmer does not replicate ue's memory per every thread, but seemingly relies on the non-overlapping timing of loop iterations
- Intel Compiler correctly detects all dependencies and refuses to parallelise this code

# OpenMP pragma [3]: UA

```
for (i = 0; i < LX1-1; i++) {  
    tmmor[idmo[iel][iface][0][0][j][i]] = 0.0; Output  
}
```

- Loop has an output dependency and Intel compiler correctly detects it
- Intel compiler does not parallelise the loop, due to an output dependency
- Same-value output dependency

# Getting ML classification labels for loops: methods and challenges

Methods and sources of loop classification labels extraction:

- **Seoul National University (SNU) NAS Parallel Benchmarks (NPB)** with added **OpenMP** pragmas
- **Intel C/C++ Compiler (ICC)** optimisation reports
- Manual SNU NPB source code study to check and verify parallelisation labels

Task challenges and complications:

- **Seoul National University (SNU) NAS Parallel Benchmarks (NPB)** with added **OpenMP** pragmas
- **Intel C/C++ Compiler (ICC)** optimisation reports
- Manual SNU NPB source code study to check and verify parallelisation labels