EPSRC
Engineering and Physical Sciences Research Council

THE UNIVERSITY of EDINBURGH
informatics

icsa
Institute for Computing Systems Architecture

A Machine Learning Based Parallelization Assistant

Aleksandr Maramzin, Christos Vasiladiotis, Roberto Castañeda Lozano,
Björn Franke, Murray Cole

The University of Edinburgh
United Kingdom

EPSRC
Engineering and Physical Sciences Research Council

THE UNIVERSITY of EDINBURGH informatics

icsa Institute for Computing Systems Architecture

A Machine Learning Based Parallelization Assistant

Aleksandr Maramzin, Christos Vasiladiotis, Roberto Castañeda Lozano,
Björn Franke, Murray Cole

"It Looks Like You're Writing a Parallel Loop"

AI-SEPS 2019 workshop

SPLASH ATHENS 2019

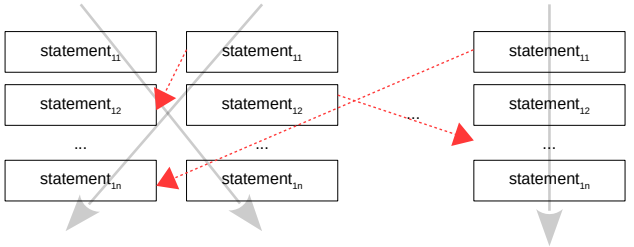
Presentation Structure

- Problem statement
- How to use the assistant tool
- Assistant internals
- Parallelization of SNU NPB benchmarks

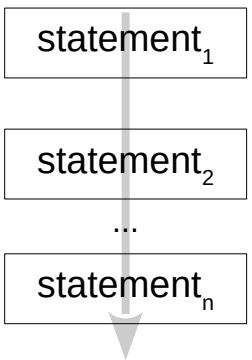
Problem Statement

Parallel Hardware is Ubiqutous

Automatic Parallelization is Limited



Software is Sequential



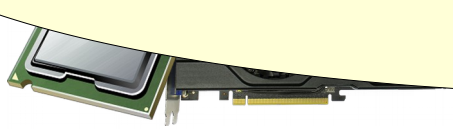
Manual Parallelization is Hard

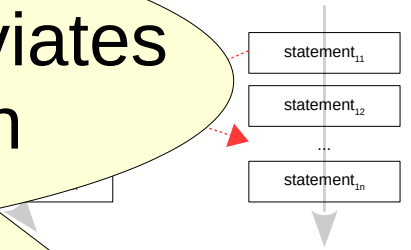
Problem Statement

Parallel Hardware is Ubiqut

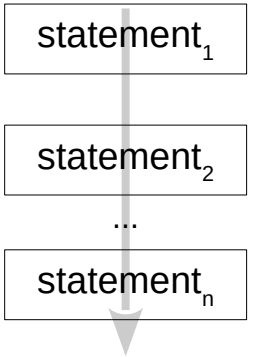
Automatic Parallelization is Limited

The **assistant solution** we propose alleviates the process of manual parallelization





Software is Sequential



Manual Parallelization is Hard

Problem Statement

NAS Parallel Benchmarks (NPB)

Parallelization Speedup (Gmean)

Manual: **1.73x**

Automatic: **0.79x** - **1.10x**

Programmer still plays the major role in the hard task of software parallelization

Manual Parallelization: Where Should I Start?

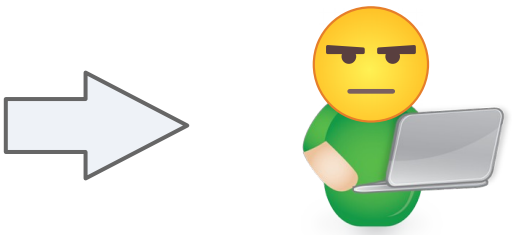
Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```



Manual Parallelization: Where Should I Start?

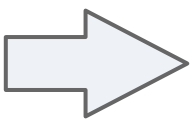
Source Code

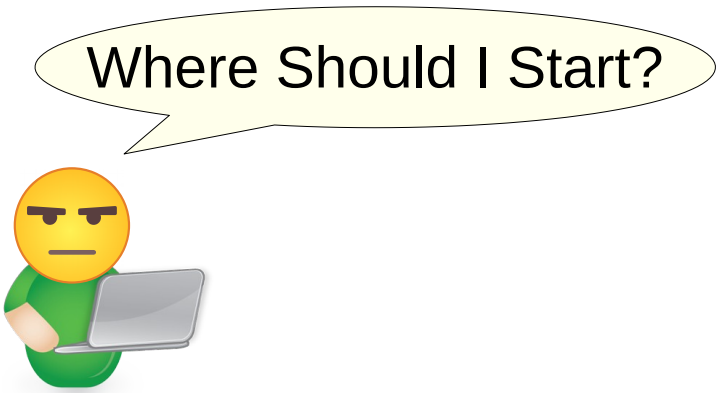
L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```





Manual Parallelization: Where Should I Start?

Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

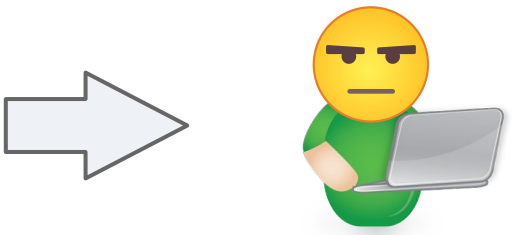
L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```

Program Loop Rankings

Source Order
L_0, L_1, … , L_n



Manual Parallelization: Where Should I Start?

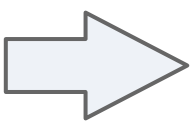
Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```



Program Loop Rankings

Source Order
$L_0, L_1, \ldots, L_n$

Manual Parallelization: Where Should I Start?

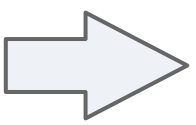
Source Code

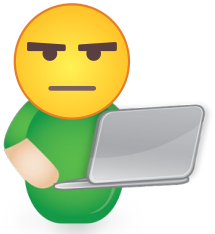
L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```





Program Loop Rankings

Source Order
$L_0, L_1, \ldots, L_n$

Manual Parallelization: Where Should I Start?

Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

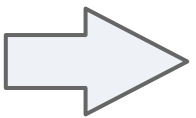
L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```

Program Loop Rankings

Source Order
L_0, L_1, … , L_n



Profile Order
L_{17}, L_{12}, … , L_1

Manual Parallelization: Where Should I Start?

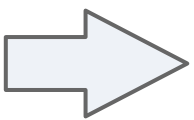
Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```



Program Loop Rankings

Source Order
$L_0, L_1, \ldots, L_n$

Profile Order
$L_{17}, L_{12}, \ldots, L_1$

Manual Parallelization: Where Should I Start?

Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

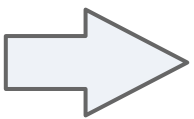
L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```

Program Loop Rankings

Source Order
$L_0, L_1, \ldots, L_n$



Traditional Empirical and Experimental Method

Profile Order
$L_{17}, L_{12}, \ldots, L_1$

AI-inspired Method

Profitability Order*
$L_{12}, L_7, \ldots, L_3$

$$^{*}Profitability = F\left(\frac{Performance\,Gain}{Programmer\ Effort}\right)$$

Manual Parallelization: Where Should I Start?

Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```

Program Loop Rankings

Source Order
$L_0, L_1, \ldots, L_n$

Profile Order
$L_{17}, L_{12}, \ldots, L_1$

AI-inspired Method

Profitability Order*
$L_{12}, L_7, \ldots, L_3$

$$^{*}Profitability = F\left(\frac{Performance\,Gain}{Programmer\ Effort}\right)$$

Manual Parallelization: Where Should I Start?

Source Code

L_0:
```
for (int i=0; … ) {
        …
}
```

L_1:
```
for (int j=0; j<n;… ) {
        …
}
```

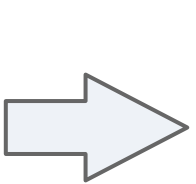
…

L_n:
```
for (int m=0;… ) {
        for (int k=m; k<n;… ) {
        …
        }
}
```

Program Loop Rankings

Final Programmer Decision

Source Order
$L_0, L_1, \ldots, L_n$



Traditional Empirical and Experimental Method

Profile Order
$L_{17}, L_{12}, \ldots, L_1$

AI-inspired Method

Profitability Order*
$L_{12}, L_7, \ldots, L_3$

$$^{*}Profitability = F\left(\frac{Performance\,Gain}{Programmer\ Effort}\right)$$

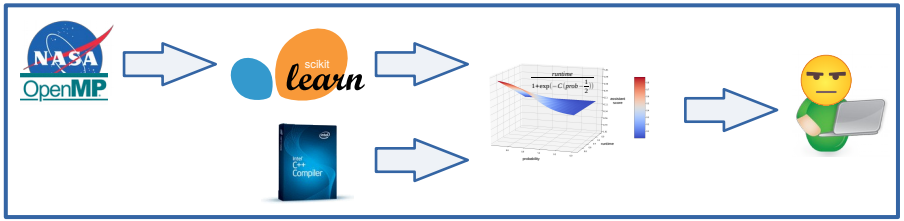
Proposed Solution

(ML based software parallelization assistant)

Tool

&

Methodology

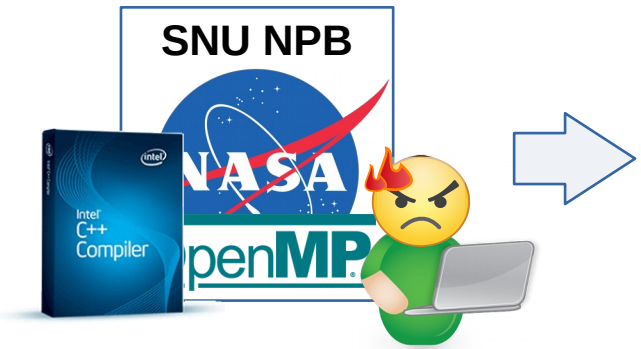


Assistant Training

Assistant Training

1415 loops

Assistant Training



1415 loops

Training Data Set

Parallelizable loops

for (int i=...) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int j=...) { … } [$f_1,f_2,\ldots,f_n$]

995 loops *****

Non-parallelizable loops

while (true) { … } [$f_1,f_2,\ldots,f_n$]

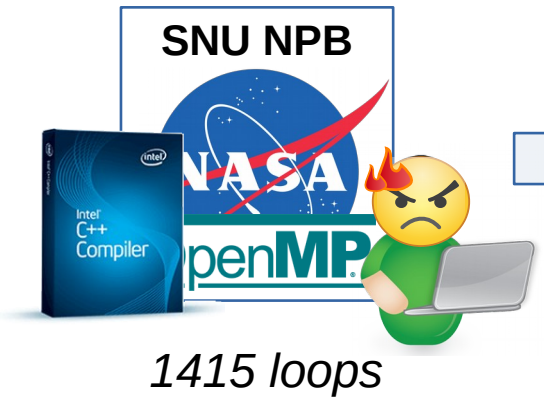
…

for (int k=...) { … } [$f_1,f_2,\ldots,f_n$]

420 loops

***** *Intel Compiler succeeds in parallelizing 812 loops*

Assistant Training



1415 loops

Training Data Set

Parallelizable loops

for (int i=...) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int j=...) { … } [$f_1,f_2,\ldots,f_n$]

995 loops *****

Non-parallelizable loops

while (true) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int k=...) { … } [$f_1,f_2,\ldots,f_n$]

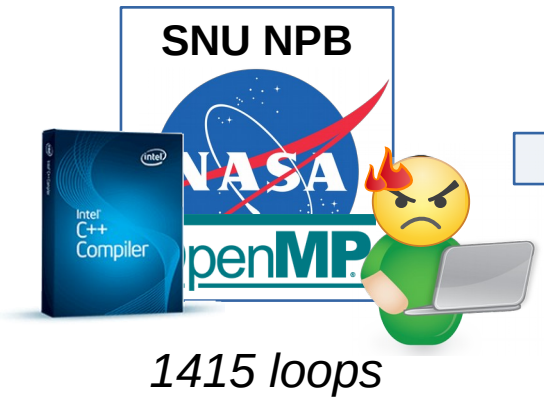
420 loops

***** *Intel Compiler succeeds in parallelizing 812 loops*

ML Based Loop Parallelizability Model

scikit learn

Assistant Training



1415 loops

Training Data Set

Parallelizable loops

for (int i=...) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int j=...) { … } [$f_1,f_2,\ldots,f_n$]

995 loops *****

Non-parallelizable loops

while (true) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int k=...) { … } [$f_1,f_2,\ldots,f_n$]

420 loops

***** *Intel Compiler succeeds in parallelizing 812 loops*

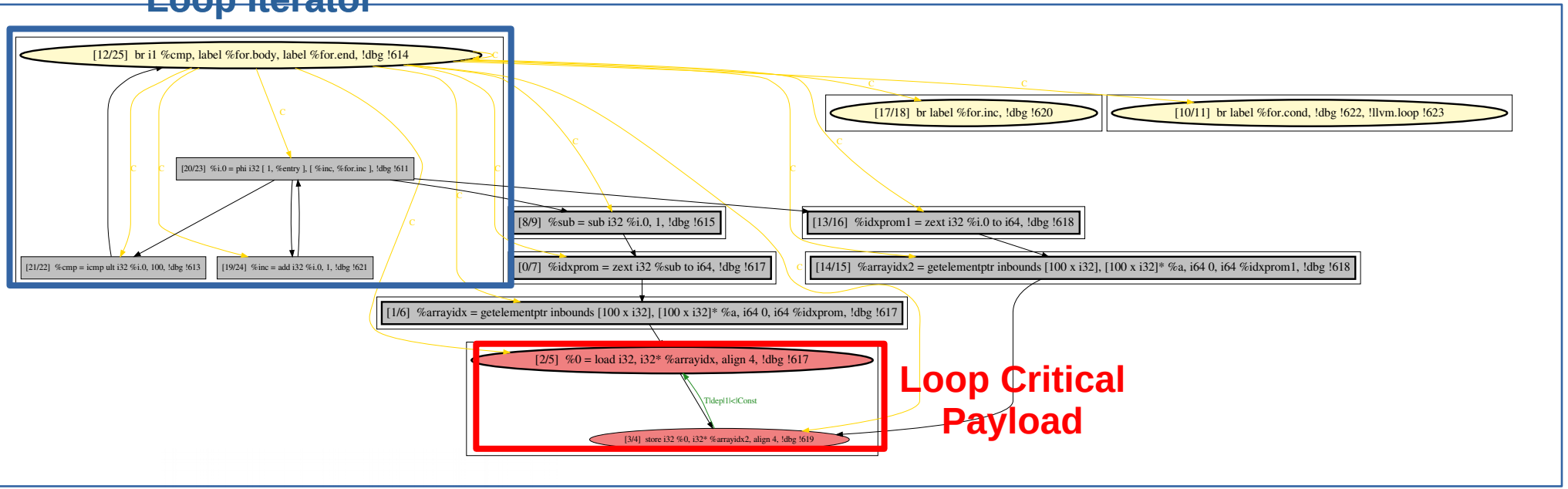
ML Based Loop Parallelizability Model

scikit learn

Loop Parallelizability Probability

Loop Features [74]

Loop Iterator

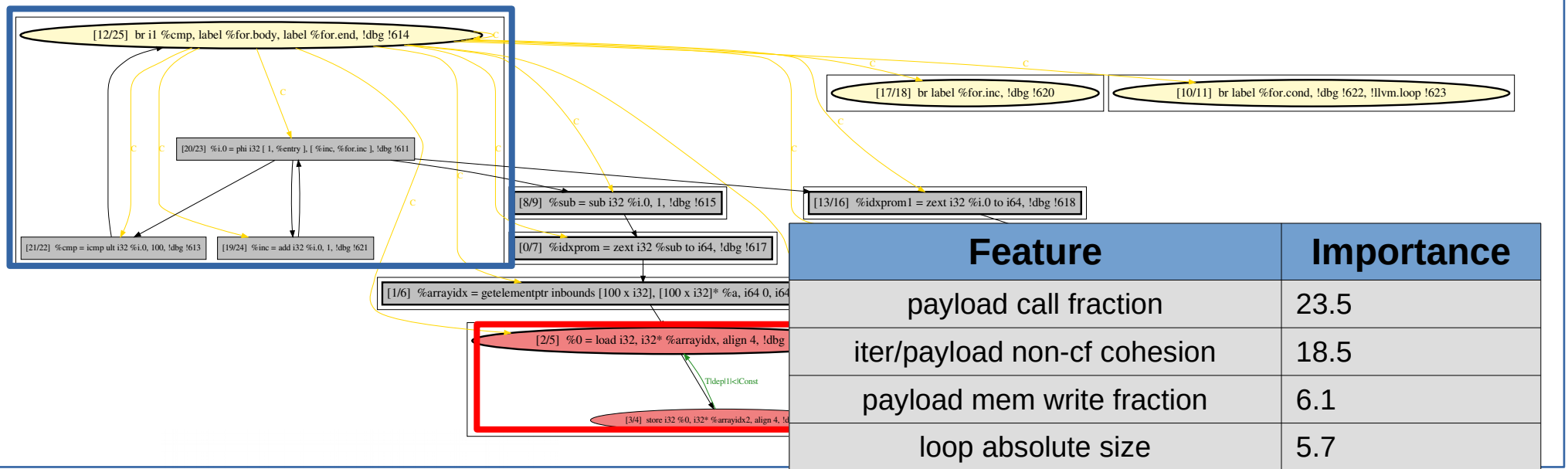


Loop features are based on static structural properties of loop program depencence graphs (PDGs):

- Absolute size
- Loop Iterator/Payload cohesion
- Number of dependence edges
- Instruction types (calls, loads/stores, etc.)
- etc.

Loop Features [74]

Loop Iterator



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 ptr access count	5.3
payload memory dependence count	4.0
critical payload non-cf cohesion	2.9
payload ptr access fraction	2.7

Loop features
of loop progra

- Absolute
- Loop Itera
- Number of dependence edges
- Instruction types (calls, loads/stores, etc.)
- etc.

Loop Ranking Score

ML Based
Loop Parallelizability
Model

Loop Ranking Score

ML Based
Loop Parallelizability
Model

Not Aware of Loop Running Time

Loop Ranking Score

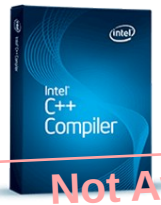
Profiler

ML Based
Loop Parallelizability
Model

Not Aware of Loop Running Time

Loop Ranking Score

Profiler



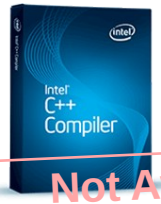
Not Aware of Loop Parallelizability

ML Based Loop Parallelizability Model

Not Aware of Loop Running Time

Loop Ranking Score

Profiler



Not Aware of Loop Parallelizability

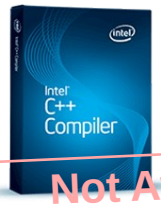
ML Based Loop Parallelizability Model

Programmer Effort

Not Aware of Loop Running Time

Loop Ranking Score

Profiler



Performance Gain

Not Aware of Loop Parallelizability

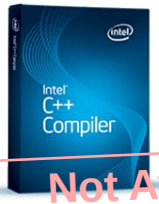
ML Based Loop Parallelizability Model

Programmer Effort

Not Aware of Loop Running Time

Loop Ranking Score

Profiler



Performance Gain

Not Aware of Loop Parallelizability

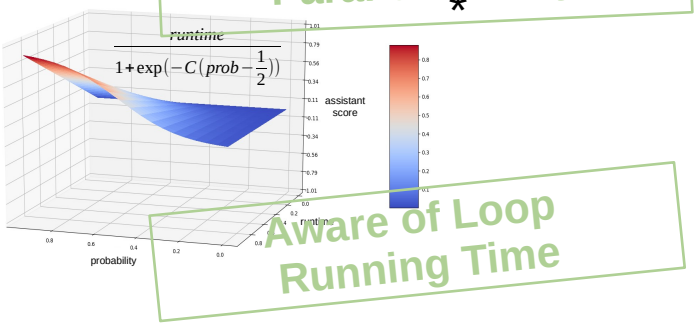
ML Based Loop Parallelizability Model

Programmer Effort

Not Aware of Loop Running Time

Aware of Loop Parallelizability

*

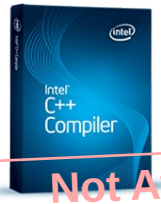


Aware of Loop Running Time

$$^{*}Profitability = F\left(\frac{Performance\,Gain}{Programmer\ Effort}\right)$$

Loop Ranking Score

Profiler



Performance Gain

Not Aware of Loop Parallelizability

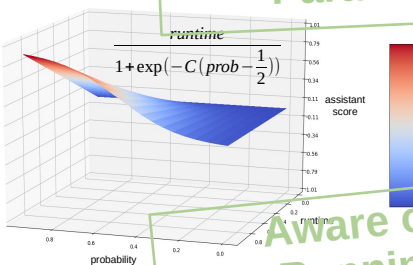
ML Based Loop Parallelizability Model

Programmer Effort

Not Aware of Loop Running Time

Aware of Loop Parallelizability

*



Aware of Loop Running Time

Improved Loop Ranking

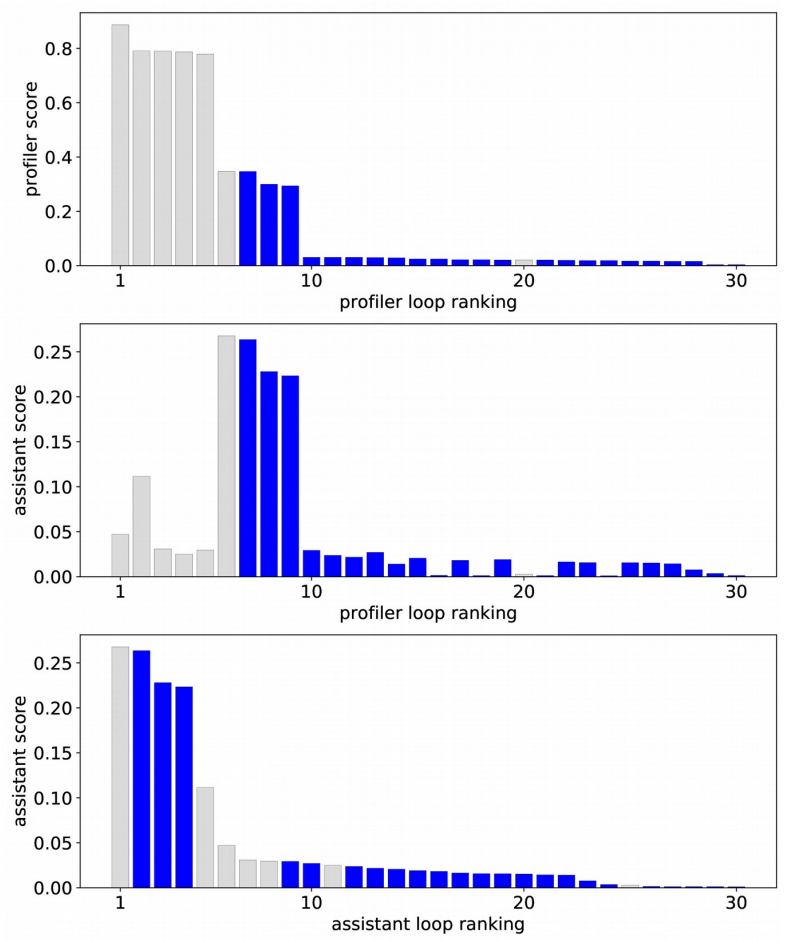
$$^{*}Profitability = F\left(\frac{Performance\,Gain}{Programmer\ Effort}\right)$$

Results

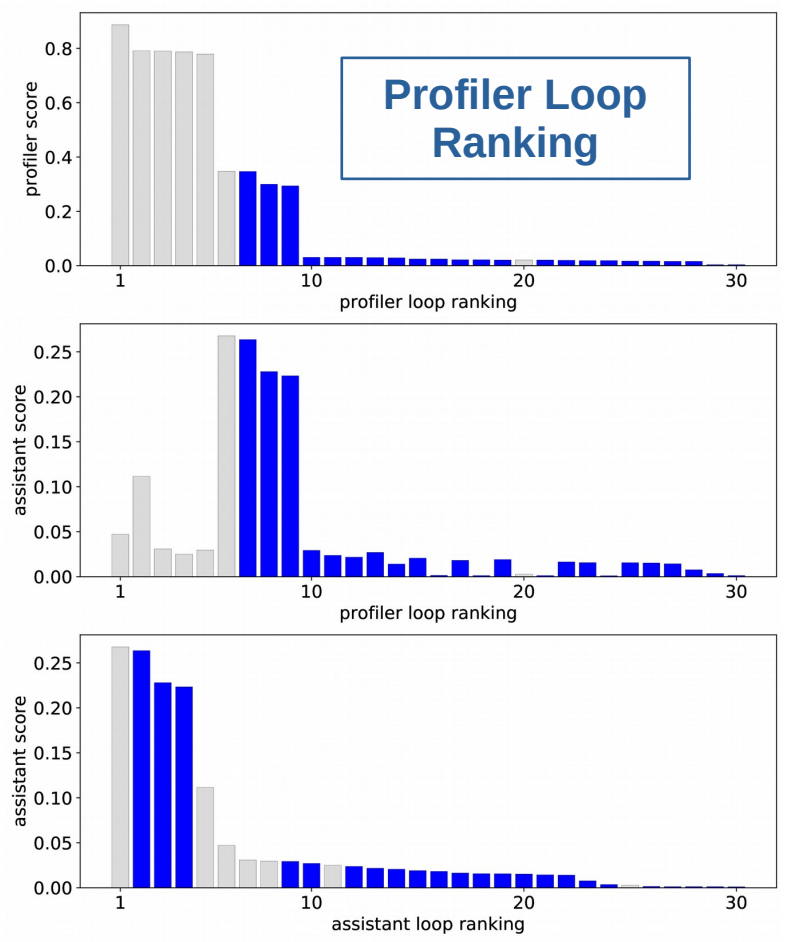
Deployment of our assistant on SNU NPB benchmarks

Predictive performance of our ML based loop parallelizability model

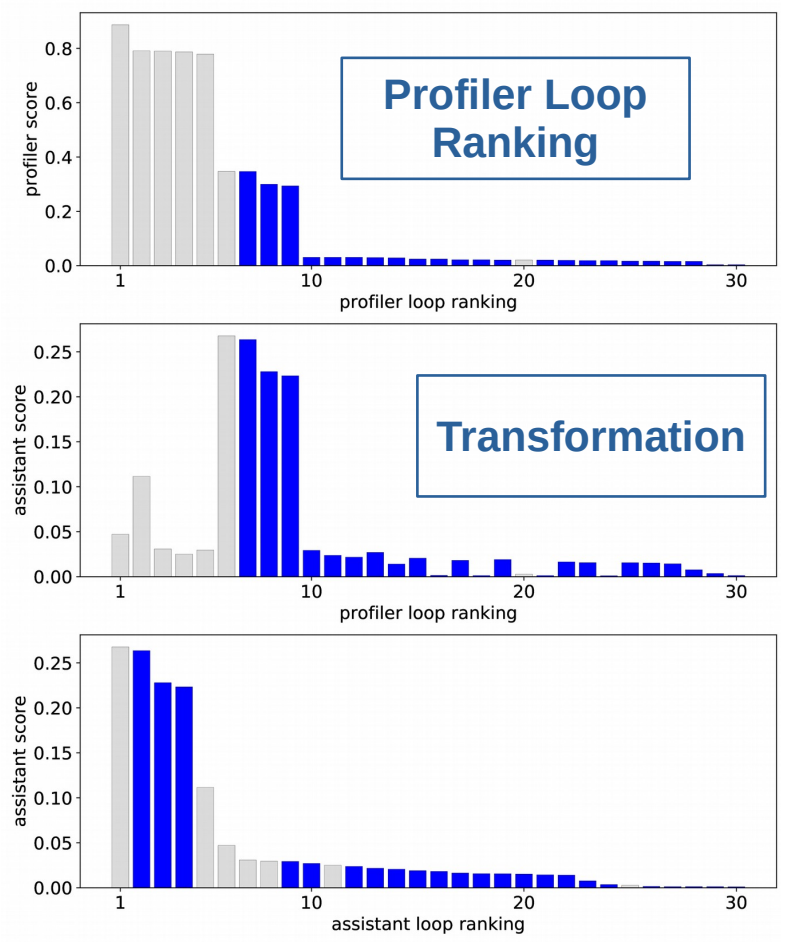
Loop Rankings: Profiler vs. Assistant



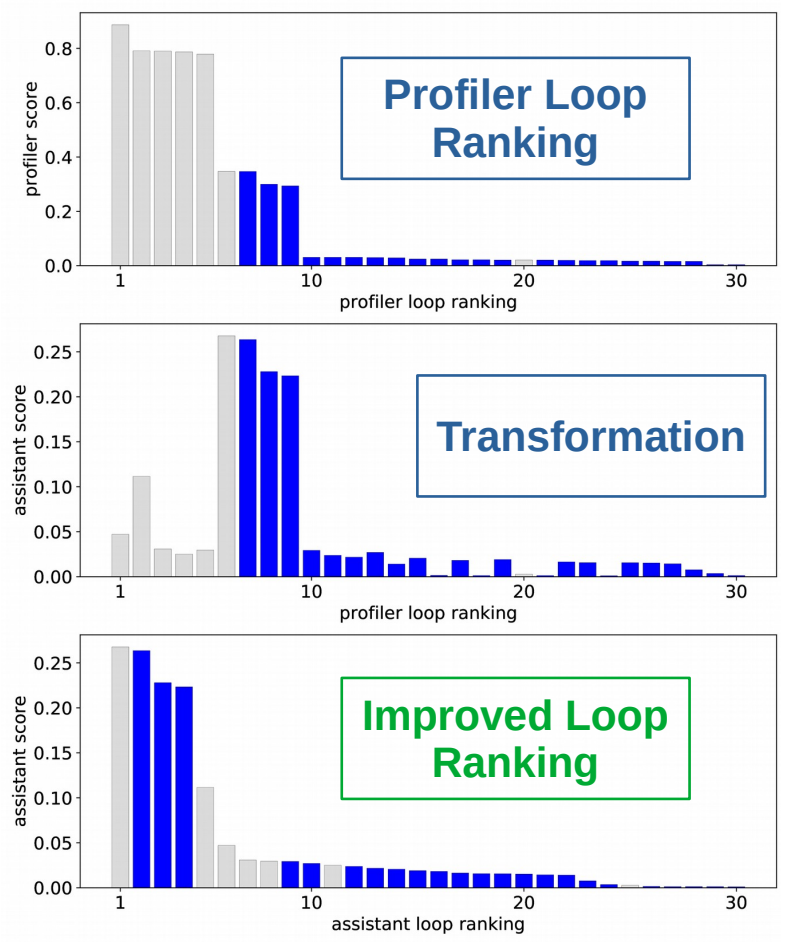
Loop Rankings: Profiler vs. Assistant



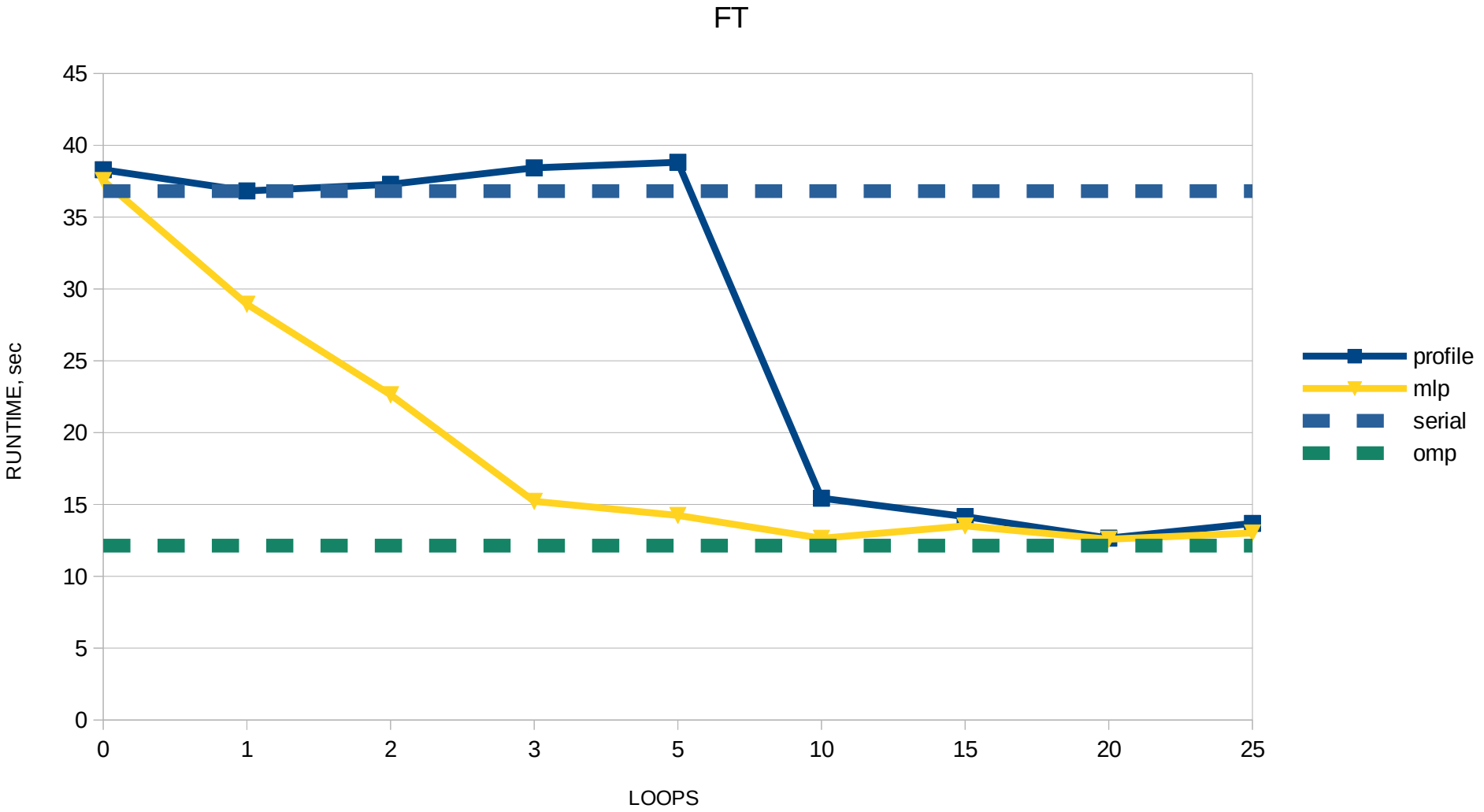
Loop Rankings: Profiler vs. Assistant



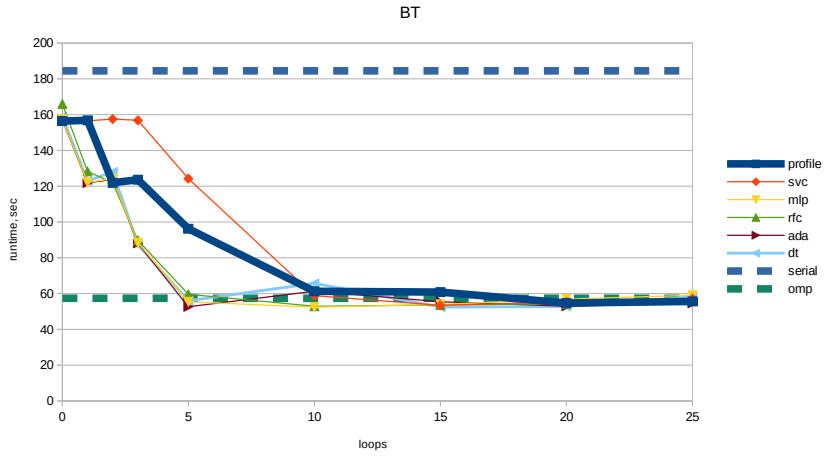
Loop Rankings: Profiler vs. Assistant

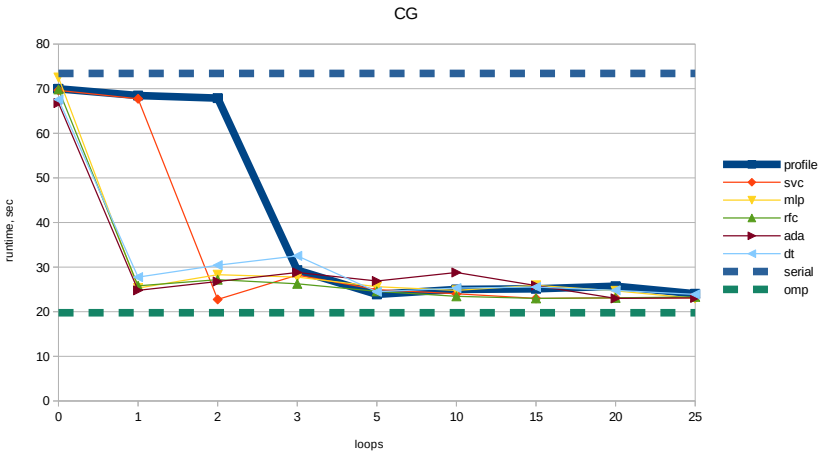


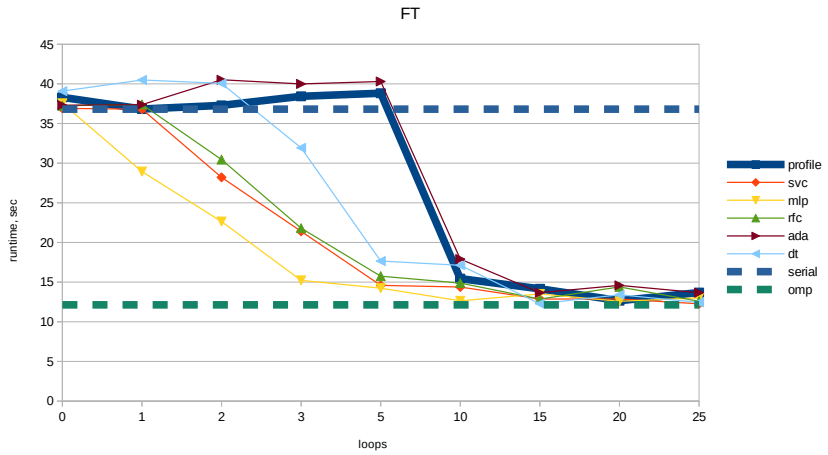
How well do we do?

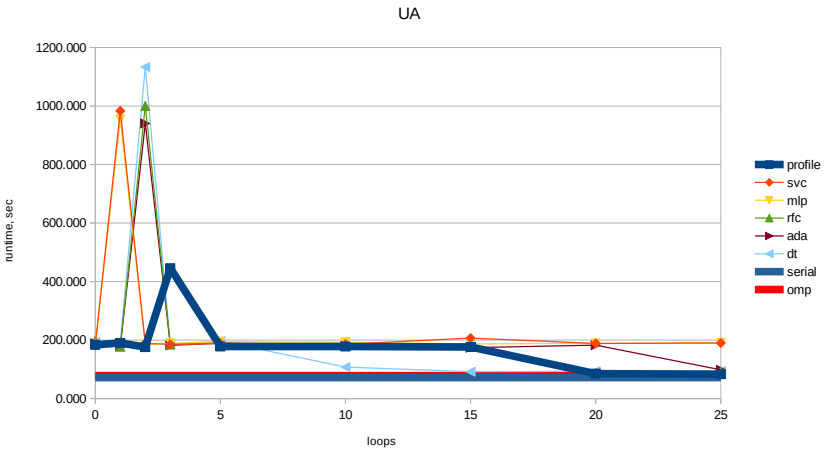


How well do we do?

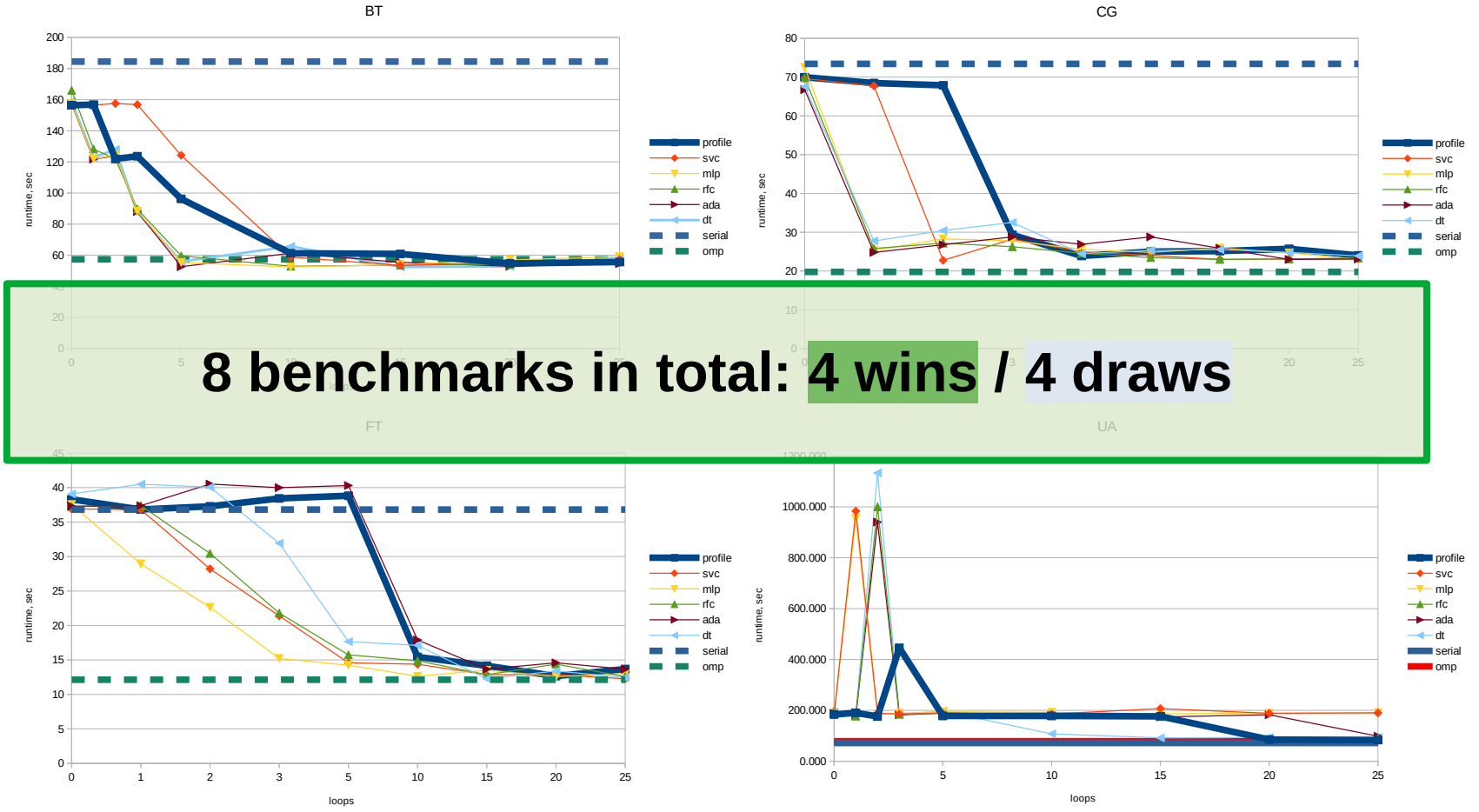






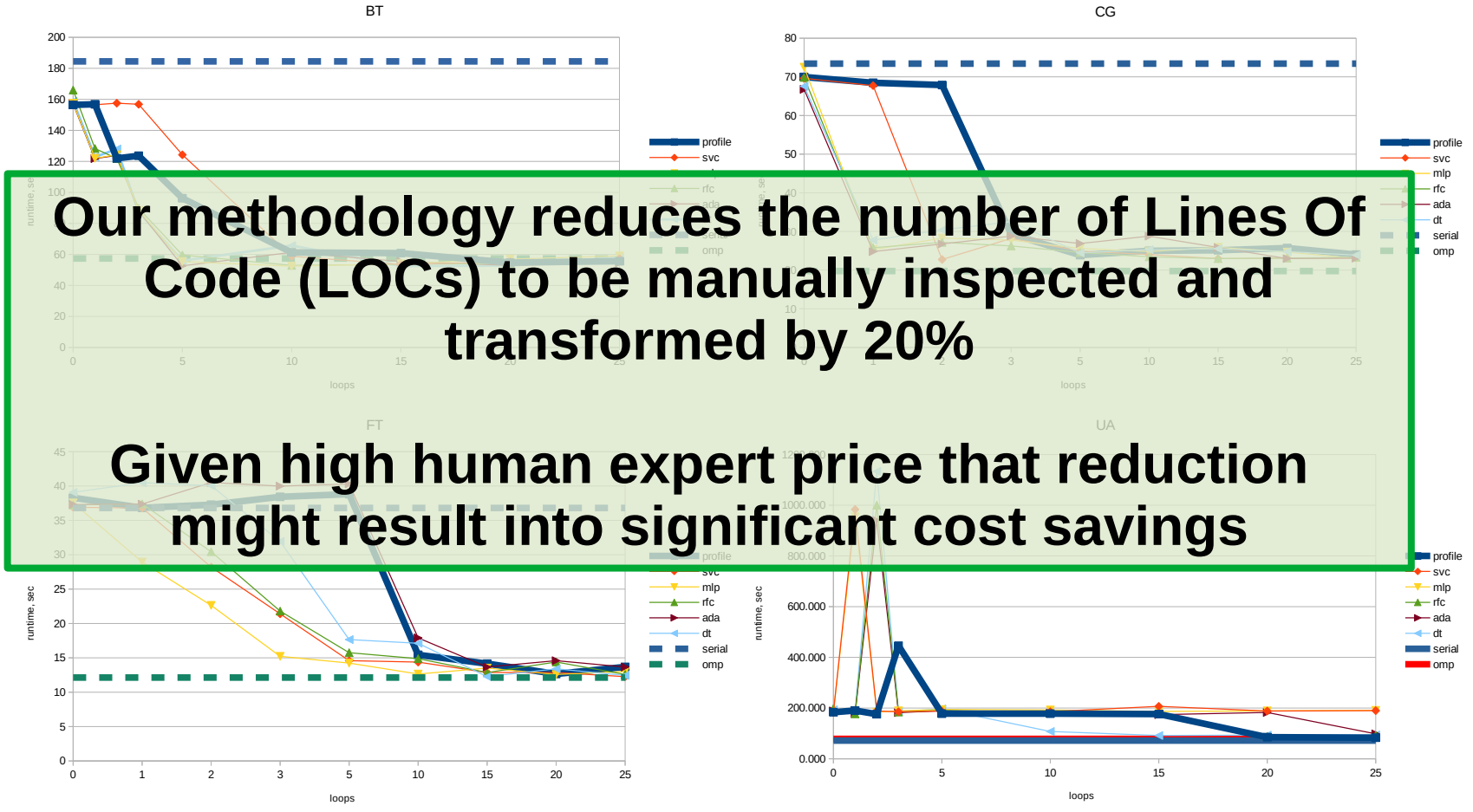


How well do we do?



8 benchmarks in total: 4 wins / 4 draws

How well do we do?



Our methodology reduces the number of Lines Of Code (LOCs) to be manually inspected and transformed by 20%

Given high human expert price that reduction might result into significant cost savings

ML Model Predictive Performance

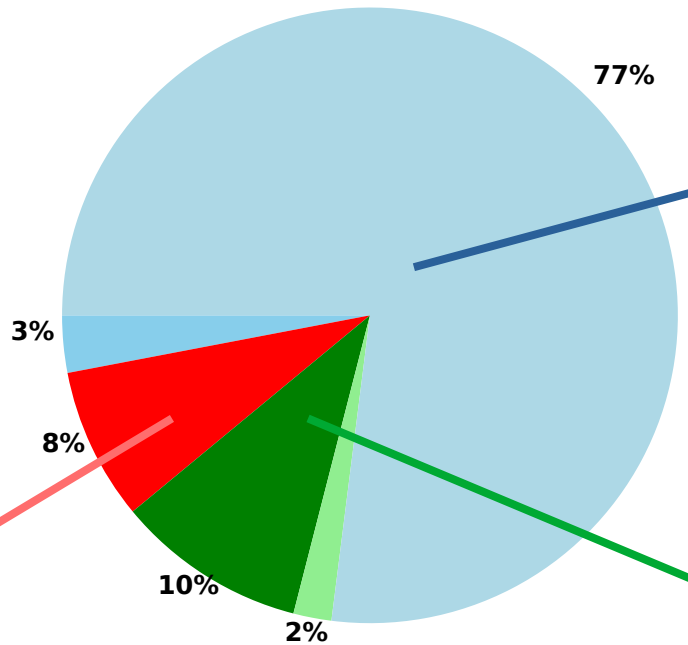
Reference

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

Around 90% predictive accuracy

ML Model Predictive Performance

ML Based Loop Parallelizability Model



Agreement rate of 80%

False positive rate of 8% is not critical

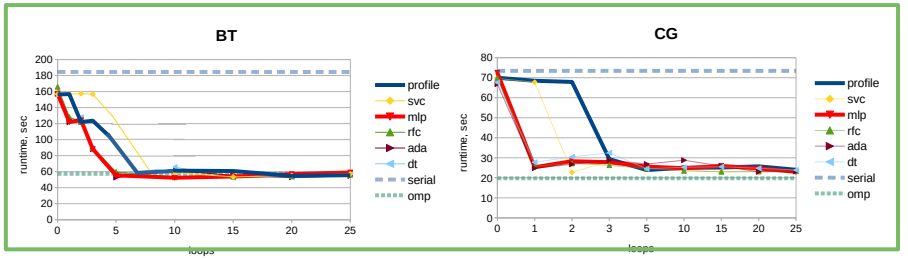
Discovery rate of 10%

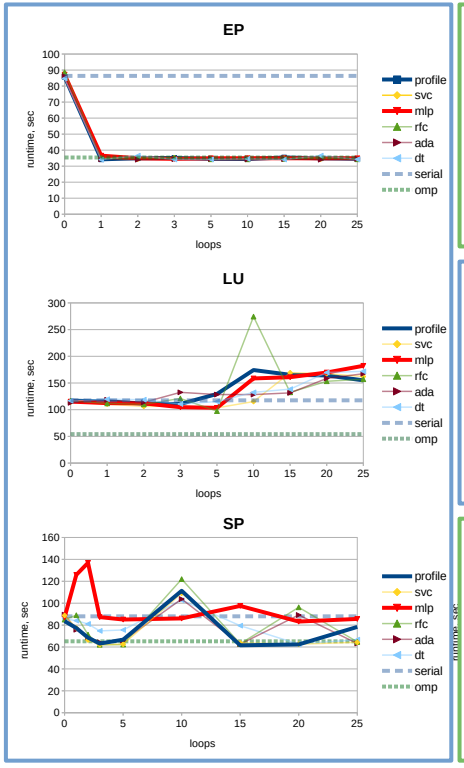
Summary & Conclusions

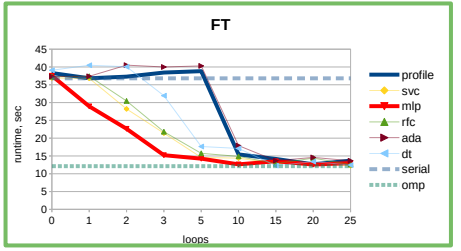
AI-inspired tool and methodology for manual software parallelization have been designed and implemented as a prototype:

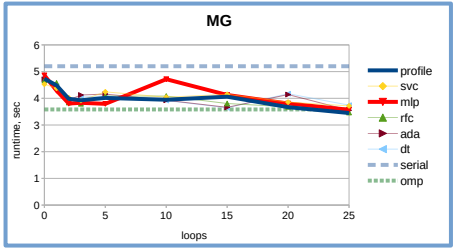
- Loop parallelizability is learnable property (we introduce a machine learning based model and train it to work with the accuracy of above 90%)
- ML model of loop parallelizability has been integrated into parallelization assistant
- Deployed against SNU NPB benchmarks our assistant showed a faster parallelization process over traditional profile-guided method: a programmer is required to inspect and transform 20% less Lines Of Code (LOC)

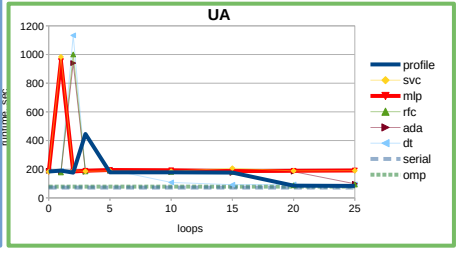
How well do we do?











Motivating example

SNU NPB Conjugate Gradient (CG) benchmark

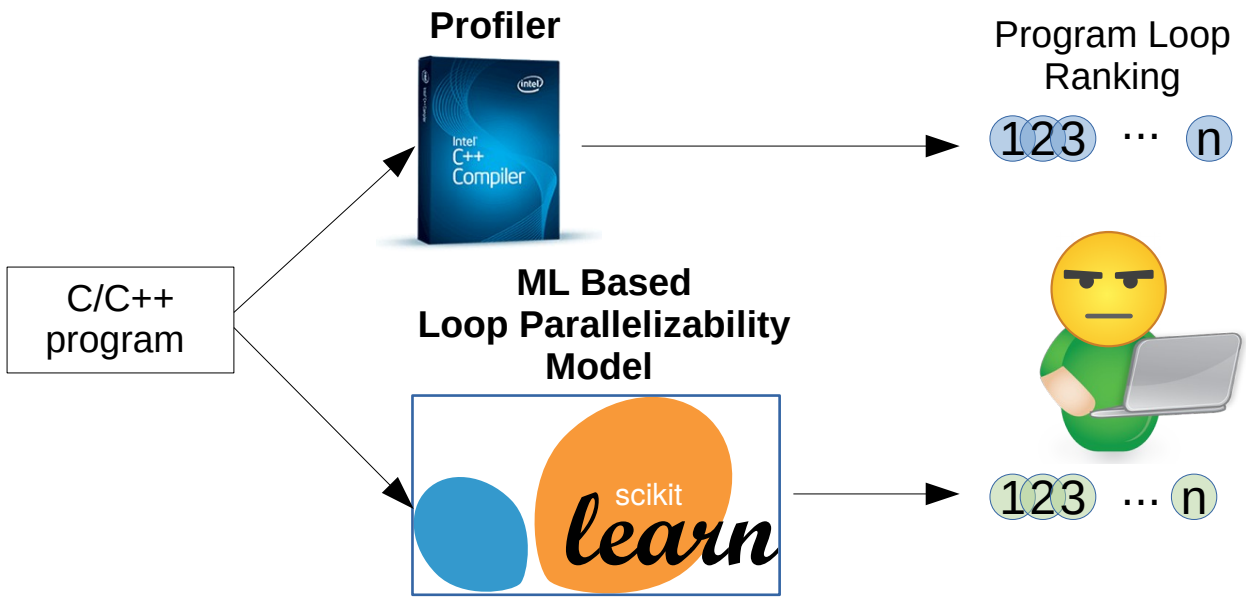
Profiler's Loop Ranking		Assistant's Loop Ranking	Assistant's Parallel Loop Probability
1 cg.c:326	*vs.*	1 **cg.c:509**	**[85%]**
2 cg.c:484		2 cg.c:326	[29%]
3 **cg.c:509**		3 cg.c:484	[8%]

```
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;
}
```

cg.c:509

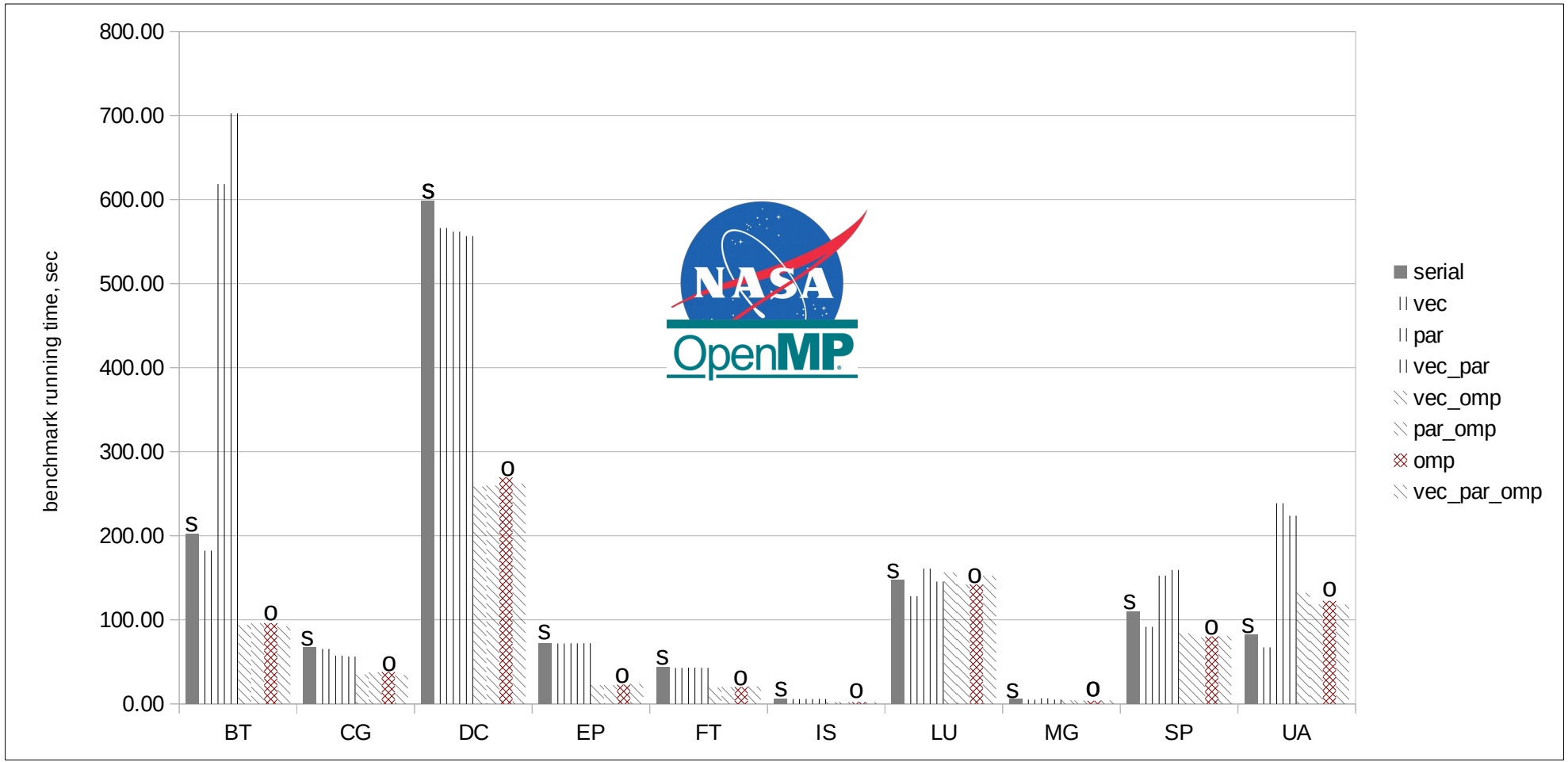
```
for (it = 1; it <= NITER; it++) {
  ...
  if (timeron) timer_start(T_conj_grad);
  conj_grad(colidx,rowstr,x,z,a,p,q,r,&rnorm);
  if (timeron) timer_stop(T_conj_grad);
  ...
  printf("    %5d       %20.14E%20.13f\n", it,
    rnorm, zeta);
  ...
}
```

cg.c:326



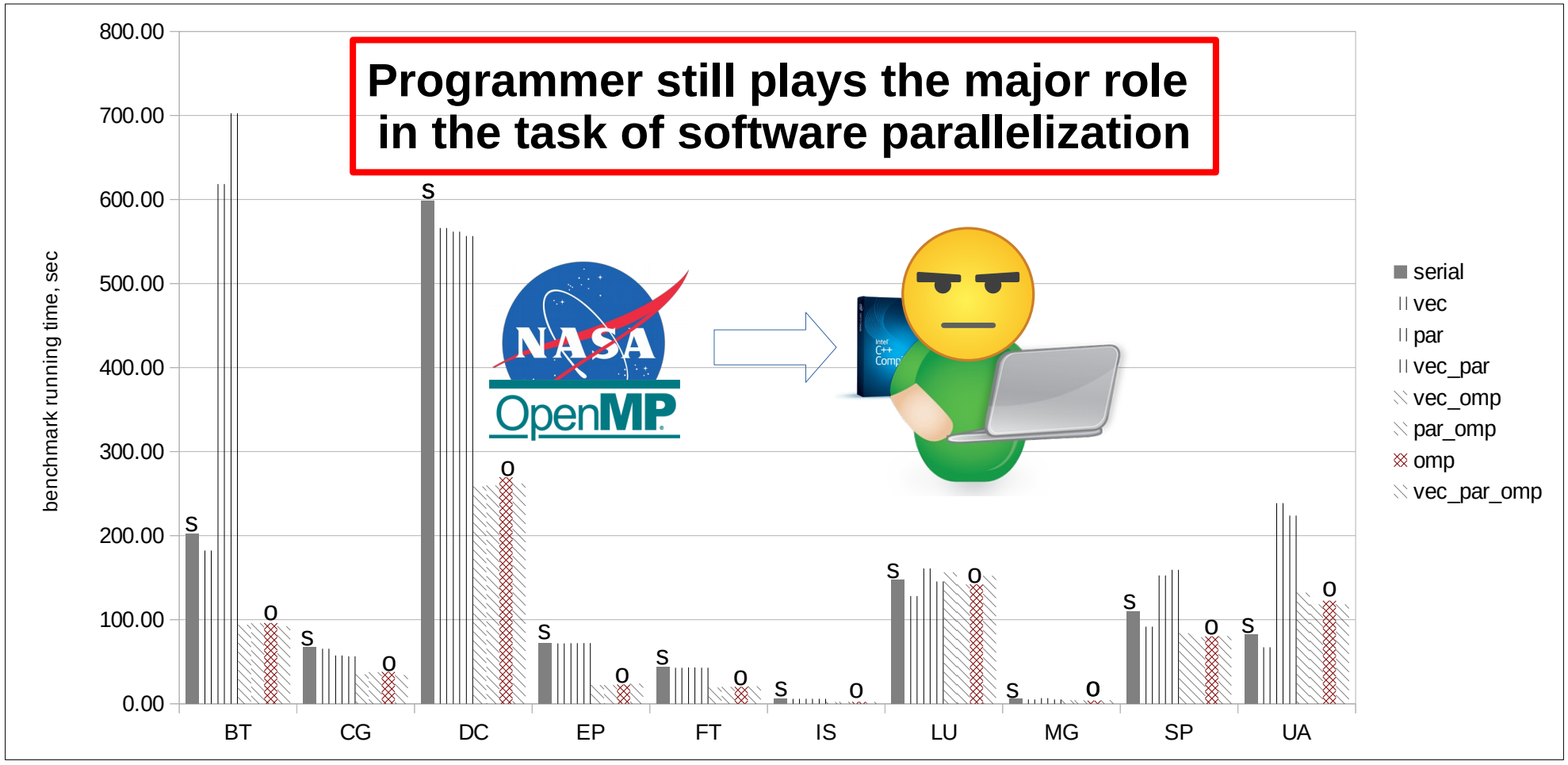
Problem Statement 2/2

NAS Parallel Benchmarks (NPB)



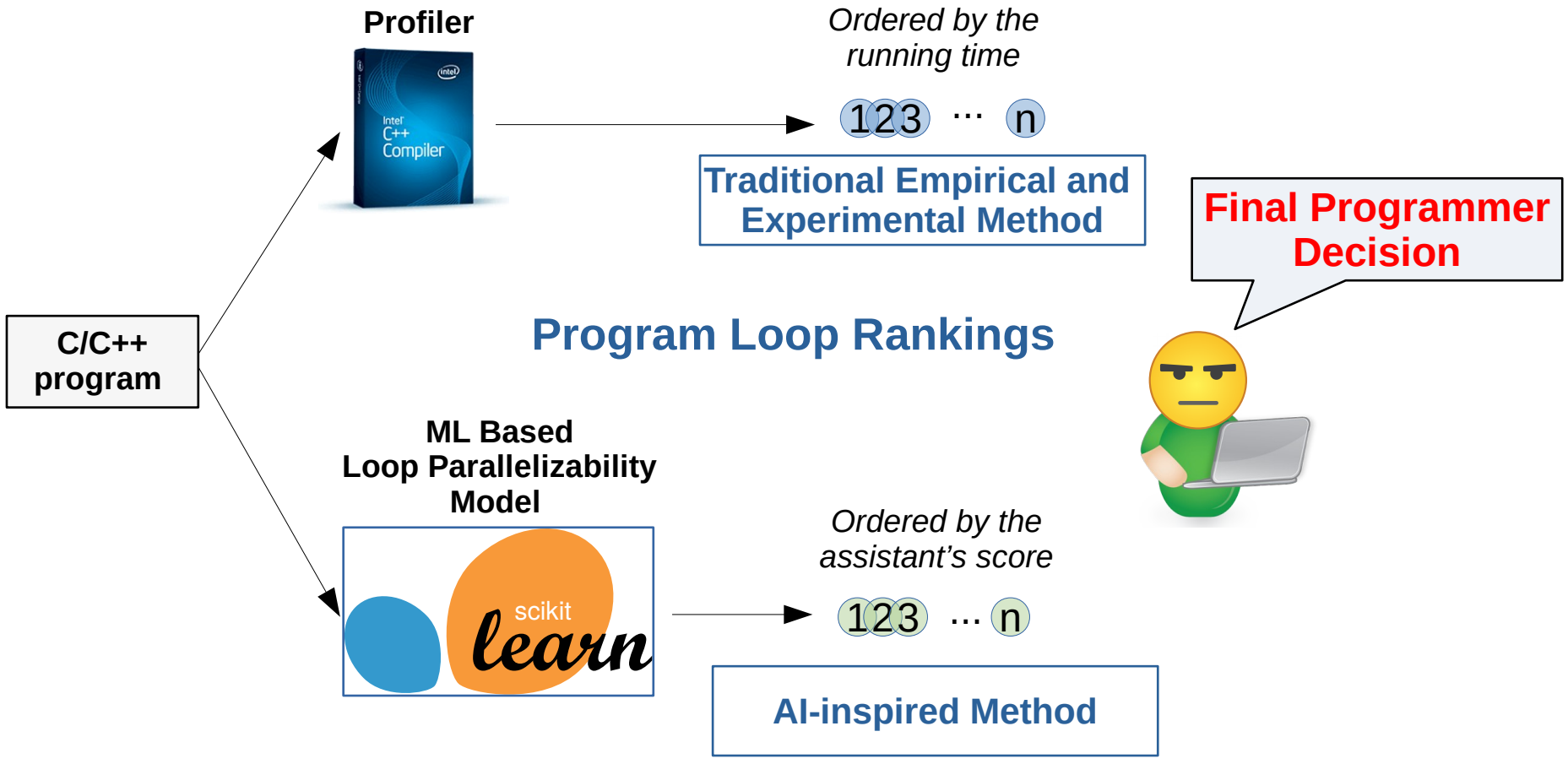
Problem Statement 2/2

NAS Parallel Benchmarks (NPB)



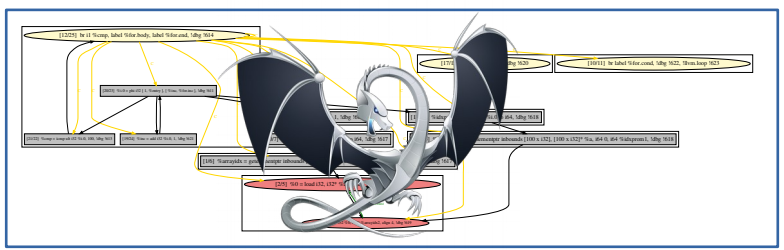
How to use the assistant

(from the programmers perspective)



Solution Scheme [1/2]

1415 loops



Loop Labels

Loop Features

Training Data Set

Parallelizable loops

for (int i=...) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int j=...) { … } [$f_1,f_2,\ldots,f_n$]

995 loops *

Non-parallelizable loops

while (true) { … } [$f_1,f_2,\ldots,f_n$]

…

for (int k=...) { … } [$f_1,f_2,\ldots,f_n$]

420 loops

ML Based Loop Parallelizability Model

scikit learn

* *Intel Compiler succeeds in parallelizing 812 loops*

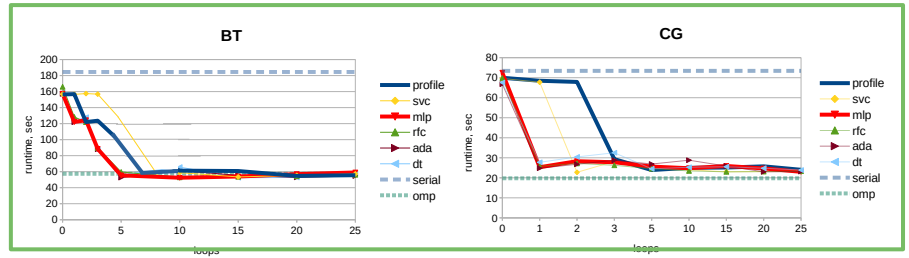
Loop features are based on static structural properties of loop program depencence graphs (PDGs):

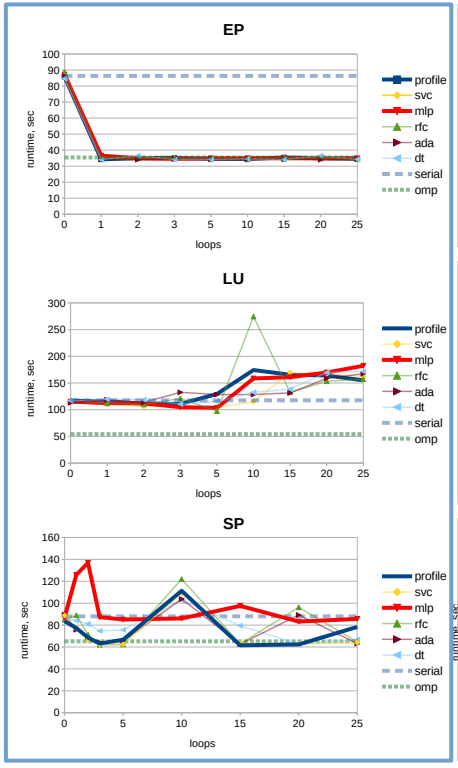
- Absolute size
- Loop Iterator/Payload cohesion
- Number of dependence edges
- Instruction types (calls, loads/stores, etc.)

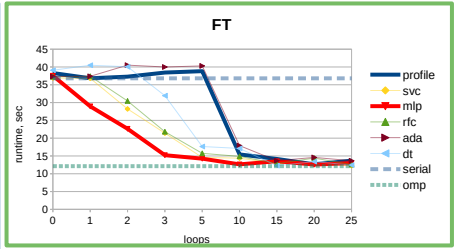
Loop labels are derived out of OpenMP pragmas present in parallelized SNU NPB versions as well as from the Intel Compiler's parallelization/vectorization reports.

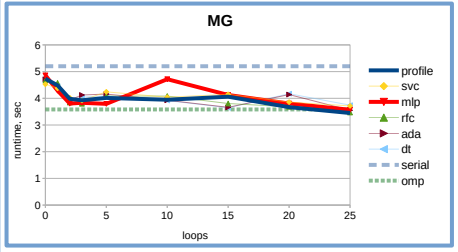
How well do we do?

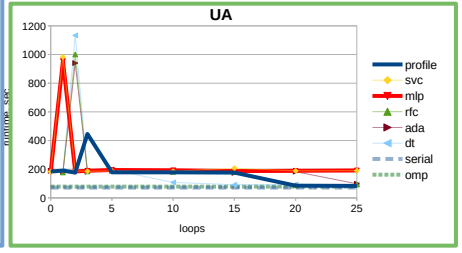
Improvement in 4 benchmarks











No change in 4 other benchmarks

Problem Statement