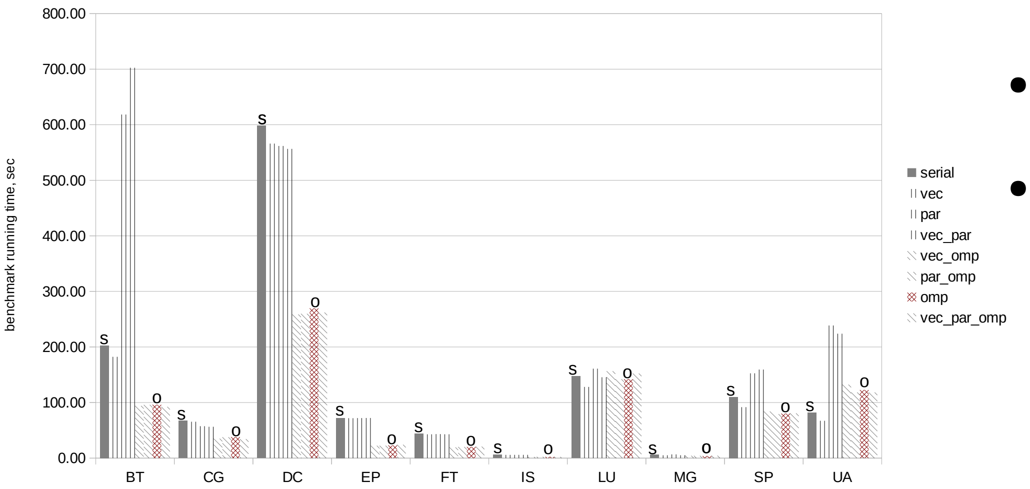
First Year Review

Software Parallelisation:
The Assistant Solution

Aleksandr Maramzin,
s1736883

The Problem



Seoul National University
NASA Parallel Benchmarks
(SNU NPB)

- Parallel hardware is ubiquitous
- Software needs to be parallelised
- The problem of software parallelisation is ***multifaceted*** (*loop nest dependencies, ambiguous pointers, unsuccessfully chosen data structures, etc.*)
- Fully automatic approaches largely **fail** to address the whole range of parallelisation constraining factors

The Solution

- We do not expect a "**silver bullet**" in the area of automatic parallelisation
- Instead, we acknowledge the role of a human programmer and propose ***an assistant solution***
- The solution is a set of manual program parallelisation assistance tools

PhD Thesis Structure Vision

(**Software Parallelisation Assistance Toolkit**)

- **1st Year:** Designed & Developed a 1st tool in a set (*Machine Learning Based Loop Parallelisability Adviser*)

  Submitted a paper to AI-SEPS 2019 workshop and **got accepted**

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

  A machine learning based parallelisation assistant

- **2nd Year:** Design & Develop a 2nd tool (*Data Structure Recogniser Tool)* aimed at identification of higher level data structures in GitHub hosted programs
- **3rd Year:** Design & Develop a *Data Structure Replacement Tool*

Tool [1]: A Machine Learning Based Loop Parallelisation Assistant

Motivating Example
(CG benchmark)

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%

Comparison of rankings provided by profiler and our assistant.

- Our loop parallelisability assistant provides a better ranking over profiler's one
- Profiler proposes to start with the longest running **non-parallelisable** loop
- Assistant points us at the longest running out of **parallelisable** loops straight away

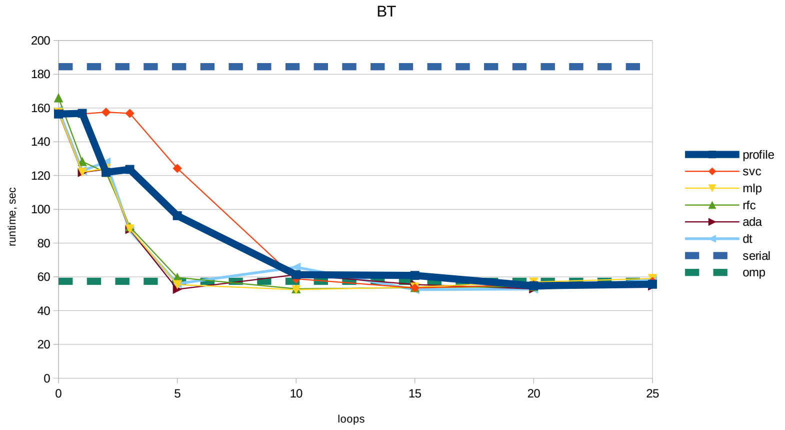
```
for (it = 1; it <= NITER; it++) {
  ...
  if (timeron) timer_start(T_conj_grad);
  conj_grad(colidx,rowstr,x,z,a,p,q,r,&rnorm);
  if (timeron) timer_stop(T_conj_grad);
  ...
  printf("    %5d       %20.14E%20.13f\n", it,
    rnorm, zeta);
  ...
}
```

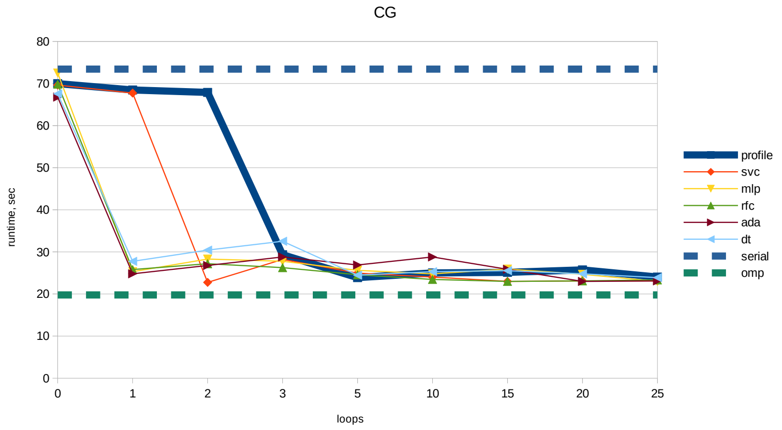
The longest running in the CG loop. The loop is *non-parallelisable*.

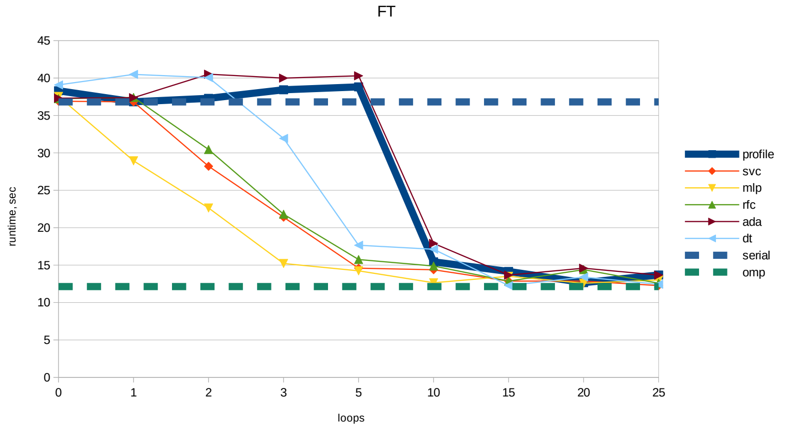
```
for (j = 0; j < lastrow-firstrow+1; j++) {
  suml = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++)
    suml = suml + a[k]*p[colidx[k]];
  q[j] = suml;
}
```

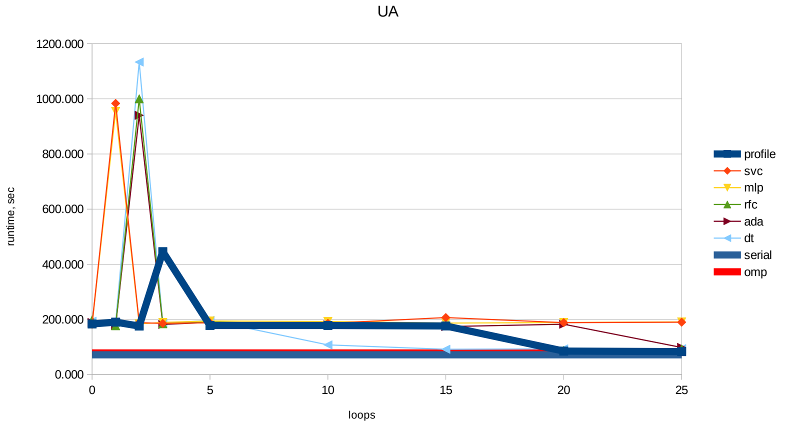
The longest running out of parallelisable loops in the CG benchmark.

SNU NPB Parallelisation





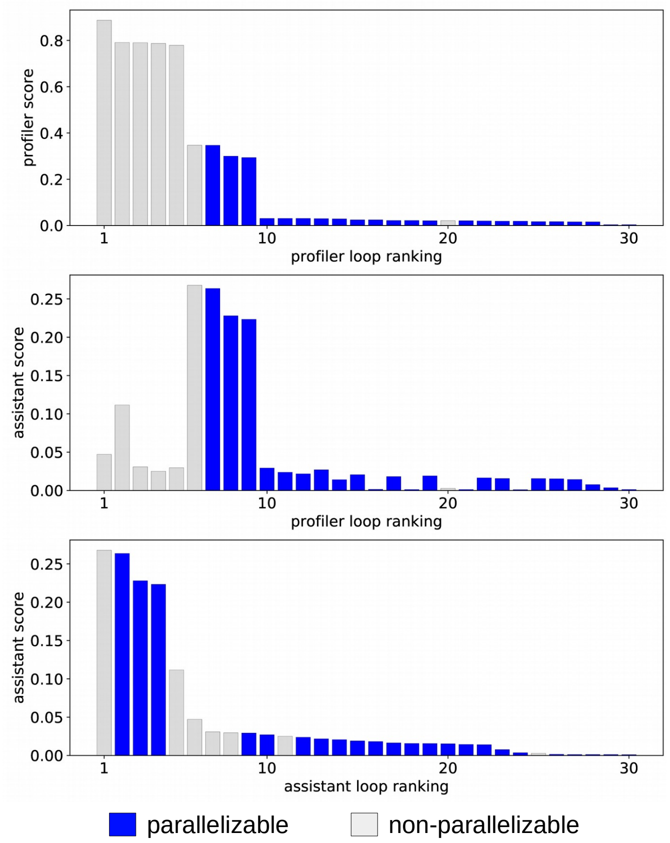




A new parallelisation methodology and a tool we propose in the paper lead to a faster convergence to the best possible benchmark performance

On average our tool decreases the number of LOC to inspect manually by 20%

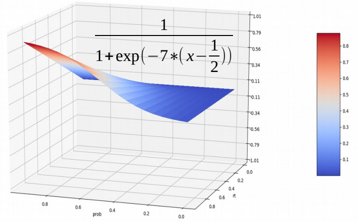
Internal Workings



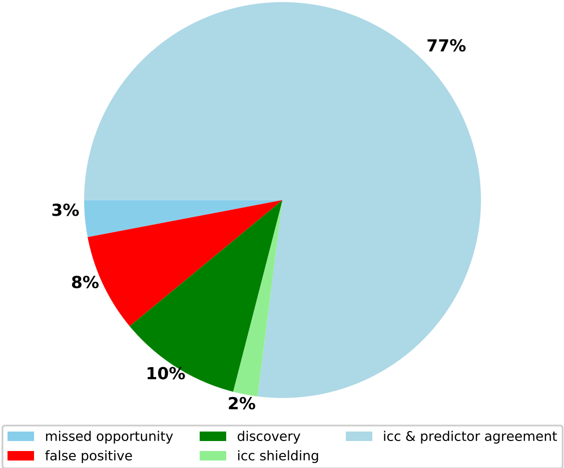
ML model	accuracy	recall	precision
constant	70.32	100	70.32
uniform	46.27	41.50	69.79
SVC	90.04	95.24	91.06
AdaBoost	86.96	92.92	89.06
DT	84.36	89.57	87.90
RFC	86.65	93.22	88.47
MLP	89.40	93.77	91.39

Loop Running Time

Loop Parallelisability Probability



Loop Assistant Ranking Score



Machine Learning Details

Loop Labels

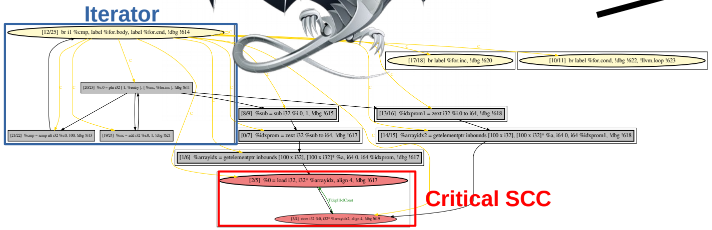
- *Intel C/C++ Compiler (ICC) Optimisation Reports*
- *SNU NPB OpenMP pragmas*
- *Manual SNU NPB study*

…

1415 loops

ML Labels

ML Features



Static Features [74]

- *Loop Iterator Size*
- *Loop Payload Dependencies Number*
- *Loop Call Instruction Number*

…

Tool [2]: Data Structure Recogniser

Tool [2]: Data Structure Recogniser (DSR)

- To be developed during the 2nd year of my PhD
- Data Structure Recogniser (DSR) ideally aims towards the following goal: *search GitHub (through its API) highly starred repositories for unsuccessfully chosen data structures like (linked-lists, pointer-based trees and graphs, etc.), identify them and ultimately replace with a more parallelisable alternatives (like arrays)*

2nd Year Plan

Approximately by the end of October (St. Andrew's meeting) I plan to finish the following steps

1. **[Write and play with toy examples]** Write a set of programs working with linked-lists, trees, graphs, compile them to LLVM IR, machine code and study the results. Transform the toy examples and estimate potential performance improvements we might get.

2. **[Benchmark search on the GitHub]** Since SPEC CPU2006 benchmarks have proved to be too complex for the task of data structure identification (automatic and even manual with an expect programmer appointed) I need to start with ***a search of suitable benchmarks***. Benchmarks must contain linked-lists and trees and be reflective of the real world code.

   - Andrew Lindsay's CodeGrep tool (*https://github.com/Andrew-lindsay/Code-Grep*) will be used as a starting point to pull the benchmarks from the GitHub. Possibly extensions will be required. GitHub use will allow us to make a statement that "benchmarks represent the real world code".

3. **[Manual transformation of software pulled from GitHub]** Once the benchmarks are at hand I will try to transform them manually and measure the potential performance improvement.

2nd Year Plan

Approximately by the end of October (St. Andrew meeting) I plan to finish the following steps

1. **[Benchmark search on the GitHub]** Since SPEC CPU2006 benchmarks have proved to be too complex for the task of data structure identification (automatic and even manual with an expect programmer appointed) I need to start with ***a search of suitable benchmarks***. Benchmarks must contain linked-lists and trees and be reflective of the real world code.
   - Andrew Lindsay's CodeGrep tool (*https://github.com/Andrew-lindsay/Code-Grep*) will be used as a starting point to pull the benchmarks from the GitHub. Possibly extensions will be required. GitHub use will allow us to make a statement that "benchmarks represent the real world code".

2. **[Manual transformation of software pulled from GitHub]** Once the benchmarks are at hand I will try to transform them manually and measure the potential performance improvement.

3. **[Static techniques effectiveness exploration on the GitHub code]** Return to Phillip's IDL tool and try to identify and replace the data structures in the GitHub programs.

2nd Year Plan

[1st scenario] _Starting from November 2019_

1. **[Switch to dynamic analysis techniques for abritrary code from the GitHub]**
   Having done a literature review in the domain of automatic data structure identification I conclude with the following:
   - Nowadays the most significant results in the field are achieved with the use of dynamic techniques, but the data structure identification is still mostly limited to standard DS libraries, textbook examples and single benchmarks.
   - Among static analysis techniques shape analysis is the most widely-known. But shape analysis is provably undecidable and necessarily conservative severely limiting its application.
   - I am planning to extend dynamic analysis techniques from DS libraries to GitHub hosted software.

2^{nd} Year Plan

[1st scenario] *Starting from November 2019*

2. **[Implementation of dynamic memory graph based DS identification tool]** The tool is going to search for patterns in the trace of memory graph update events and examine the call graph of the program.
   - [**DDT**] Uses dynamic memory graph in combination with a call graph to deduce the set of DS iface functions, detect their invariants and thus classify them (insert, delete, etc.)
   - [**dsOli**] Collects the trace of memory graph update events [E_1, E_2,…, E_n] and transforms that trace into the trace of feature vectors [F_1, F_2,…, F_n]. Uses Minimum Length Description (MLD) technique to compress the trace to a set of repetitive patterns corresponding to particular DS update operations
   - [**DSI**] On the contrary to the above mentioned tools DSI uses strands (linked-lists) as DS building primitives (not just single allocated nodes)

Rough Conceptual Scheme

Call Graph

void insert(int val)

void _insert_into_list(Node_t list, int val)*

void _insert_node(Node_t node)*

void delete(int val)

void delete_node(Node_t node)*

E_1 alloc M_{new}

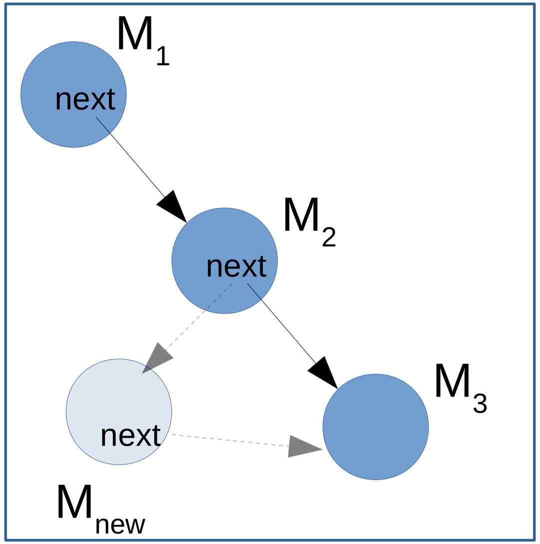
$\boldsymbol{E_2}$ *store* $M_{new} \rightarrow next = M_3$

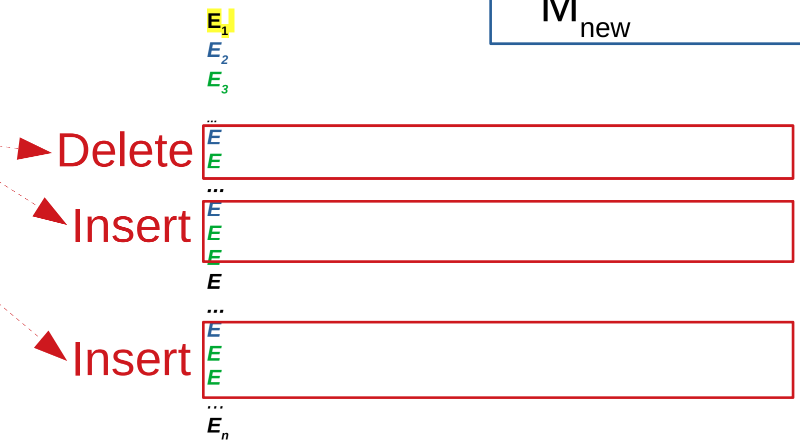
$\boldsymbol{E_3}$ *store* $M_2 \rightarrow next = M_{new}$

$\boldsymbol{E_4}$ ...

...

$\boldsymbol{E_n}$ ...





2nd Year Plan

[2nd scenario] *Before the end of October (St Andrew's meeting)*

1. **[Static techniques effectiveness exploration on the GitHub code]** Return to Phillip's Idiom Description Language (IDL) tool and try to identify and replace the data structures in the GitHub hosted programs.

[2nd scenario] *Starting from November 2019*

2. **[Continue with Phillip's IDL and static analysis on GitHub hosted code]** Try to statically recognise the patterns of linked lists, trees etc. and transform the code automatically.

Thank You!