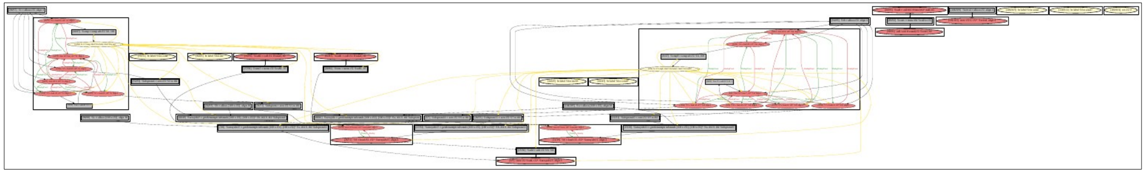
# Machine Learning Based
# Code Parallelisability Analyzer

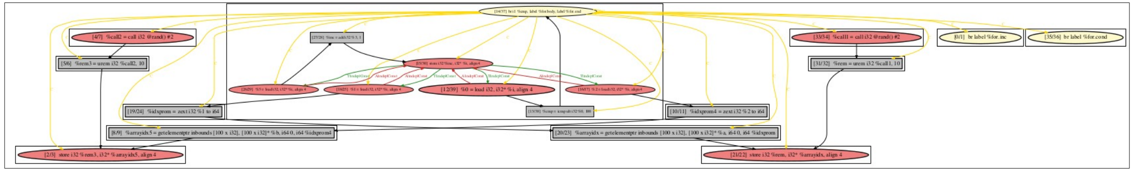
# Basic idea

```cpp
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

static const unsigned int size = 100;
static const unsigned int range = 10;

int main() {
    unsigned int a[size];
    unsigned int b[size];
    unsigned int c[size];

    std::srand(std::time(nullptr));

    // initialization loop
    for (unsigned int i = 0; i < size; i++) {
        a[i] = rand() % range;
        b[i] = rand() % range;
    }

    // vector sum computation
    for (unsigned int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

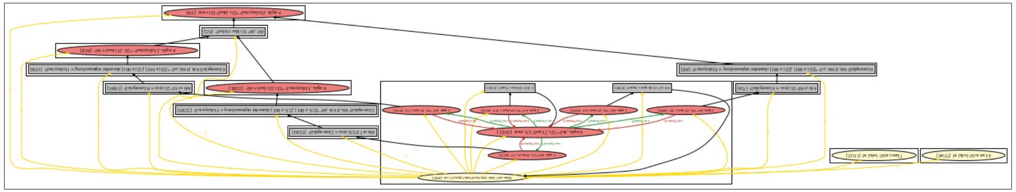
Function PDG



Loop0 PDG



Loop1 PDG



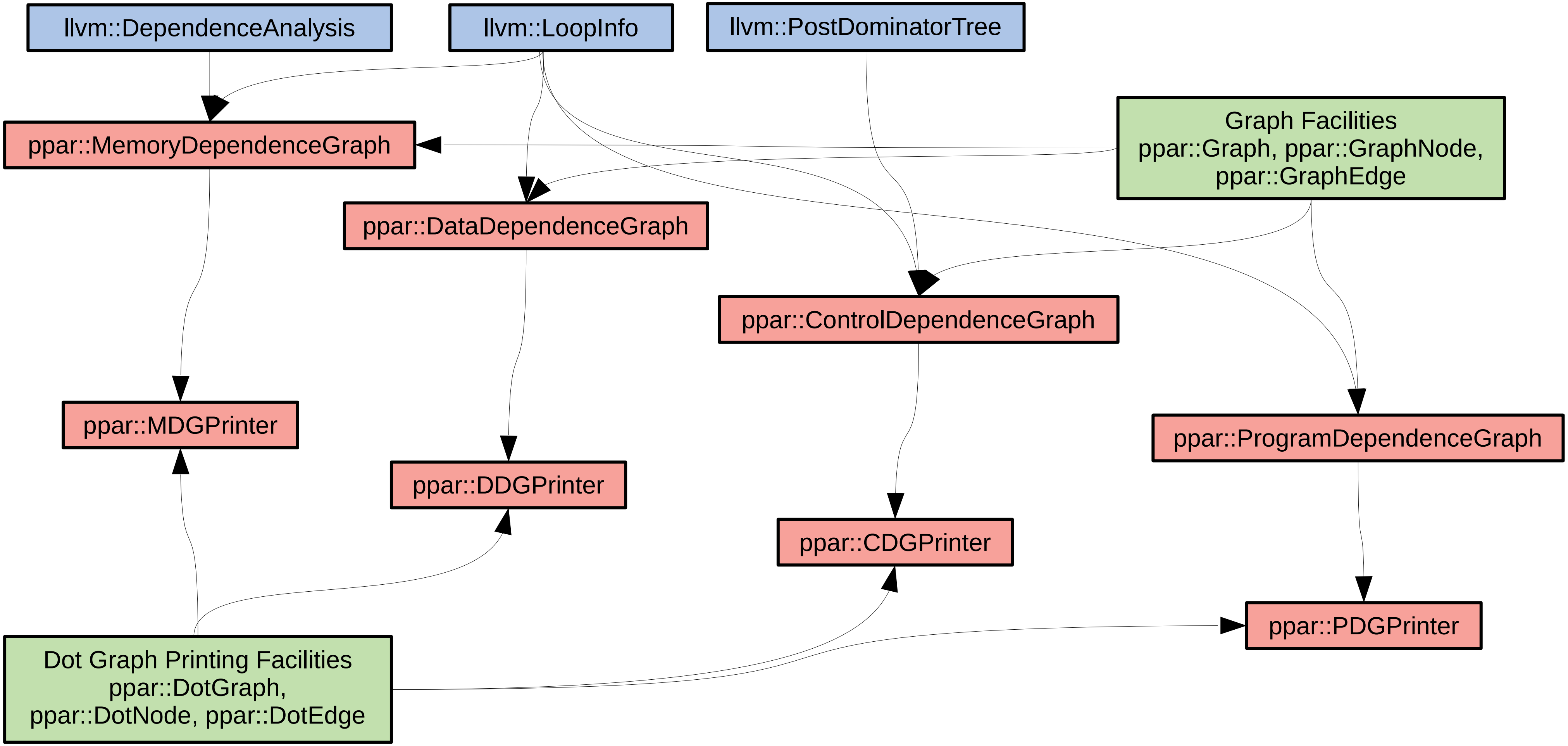
# Pervasive Parallelism (PPar) Code Parallelisability Analyzer

- A tool to be used for software parallelisability assessment
- Developed as a dynamic library (.so) to be plugged into LLVM 6.0
- Uses standard LLVM passes to build dependence graph of the source code
- Built graphs might be printed and studied
- Tool computes a set of metrics on the built dependence graphs

GitHub https://github.com/av-maramzin/PParMetrics

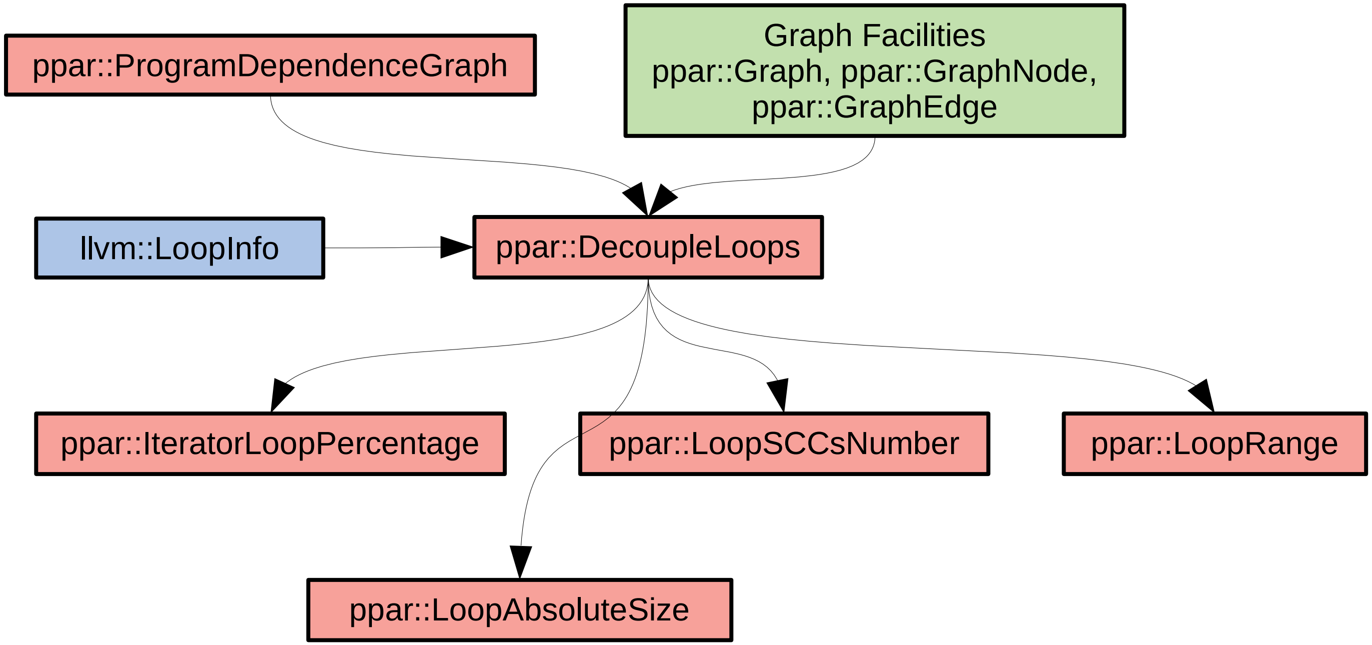
# Software Architecture of Graph Building and Printing Facilities

## A collection of LLVM passes



# Software Architecture of Metrics Collecting Facilities

## A collection of LLVM passes



# llvm::DependenceAnalysis

- LLVM pass which analyses dependencies between memory accesses. As of LLVM 6.0 it is an implementation of the approach described in [1].

- llvm::DependenceAnalysis pass exists to support llvm::DependenceGraph pass (*does not exist in LLVM yet*).

- There are two separate passes, because it is a useful separation of concerns. A dependence exists if two conditions are met:
  1) Two instructions reference the same memory location
  2) There is a flow of control, leading from one instruction to the other.

- llvm::DependenceAnalysis addresses the first condition, llvm::DependenceGraph (*not available as a standard LLVM pass yet, as of LLVM 6.0*) addresses the second.

- llvm::DependenceAnalysis is *Work In Progress* in LLVM

[1] Practical Dependence Testing. Gina Goff, Ken Kennedy, Chau-Wen Tseng. Department of Computer Science, Rice University, Houston, TX. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, June 26-28, 1991.

# llvm::DependenceAnalysis

- LLVM class Dependence represents a dependence between two memory references in a function.

```
class Dependence {
  Dependence(Instruction *Source,
             Instruction *Destination);

  bool isInput() const;
  bool isOutput() const;
  bool isFlow() const;
  bool isAnti() const;
  bool isOrdered() const { return isOutput() || isFlow() || isAnti(); }
  bool isUnordered() const { return isInput(); }

  /// isLoopIndependent - should be set by the caller if it appears that control flow can reach
  /// from Src to Dst without traversing a loop back edge.
  virtual bool isLoopIndependent() const { return true; }
  /// isConfused - Returns true if this dependence is confused
  /// (the compiler understands nothing and makes worst-case assumptions).
  virtual bool isConfused() const { return true; }
  /// isConsistent - Returns true if this dependence is consistent
  /// (occurs every time the source and destination are executed).
  virtual bool isConsistent() const { return false; }
  /// getLevels - Returns the number of common loops surrounding the
  /// source and destination of the dependence.
  virtual unsigned getLevels() const { return 0; }
  /// getDirection - Returns the direction associated with a particular level.
  virtual unsigned getDirection(unsigned Level) const { return DVEntry::ALL; }
  /// getDistance - Returns the distance (or NULL) associated with a particular level.
  virtual const SCEV *getDistance(unsigned Level) const { return nullptr; }
  /// isPeelFirst - Returns true if peeling the first iteration from this loop will break this dependence.
  virtual bool isPeelFirst(unsigned Level) const { return false; }
  /// isPeelLast - Returns true if peeling the last iteration from this loop will break this dependence.
  virtual bool isPeelLast(unsigned Level) const { return false; }
  /// isSplitable - Returns true if splitting this loop will break the dependence.
  virtual bool isSplitable(unsigned Level) const { return false; }
  /// isScalar - Returns true if a particular level is scalar; that is, if no subscript in the source or
  /// destination mention the induction variable associated with the loop at this level.
  virtual bool isScalar(unsigned Level) const;
};
```

# llvm::DependenceGraph

Not yet ready in LLVM!
Work In Progress!

- Generally, the dependence analyzer will be used to build a dependence graph for a function. Looking for cycles in the graph shows us loops, that cannot be trivially vectorized/parallelized

- We can try to improve the situation by examining all the dependences that make up the cycle, looking for ones we can break. Sometimes, peeling the first or last iteration of a loop will break dependences, and there are flags for those possibilities. Sometimes, splitting a loop at some other iteration will do the trick, and we've got a flag for that case. Rather than waste the space to record the exact iteration (since we rarely know), we provide a method that calculates the iteration. It's a drag that it must work from scratch, but wonderful in that it's possible.

  Here's an example:

  ```
  for (i = 0; i < 10; i++)
      A[i] = ...
      ... = A[11 – i]
  ```

  There's a loop-carried flow dependence from the store to the load, found by the weak-crossing SIV test. The dependence will have a flag, indicating that the dependence can be broken by splitting the loop. Calling getSplitIteration will return 5. Splitting the loop breaks the dependence, like so:

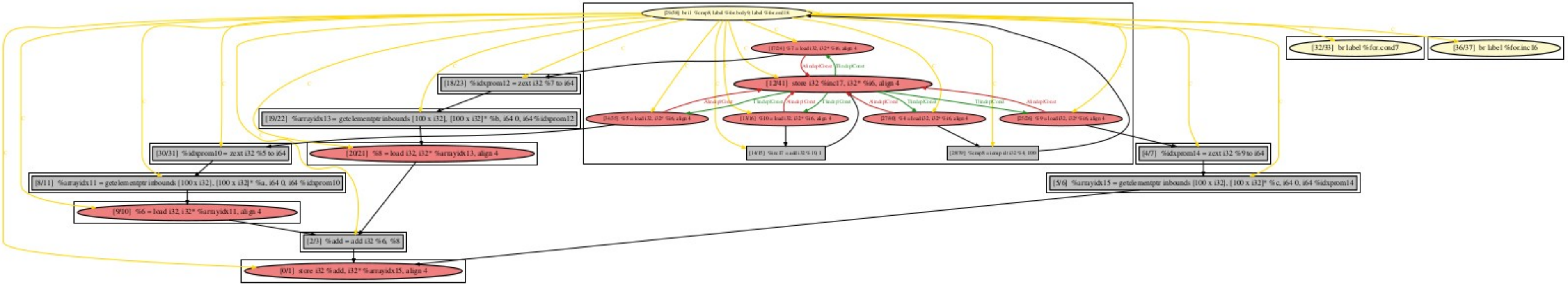
  ```
  for (i = 0; i <= 5; i++)
    A[i] = ...
    ... = A[11 - i]
  for (i = 6; i < 10; i++)
    A[i] = ...
    ... = A[11 - i]
  ```

  breaks the dependence and allows us to vectorize/parallelize both loops.

# Case study [1] : parallelisible loop

```c
// vector sum computation
for (unsigned int i = 0; i < size; i++) {
    c[i] = a[i] + b[i];
}
```

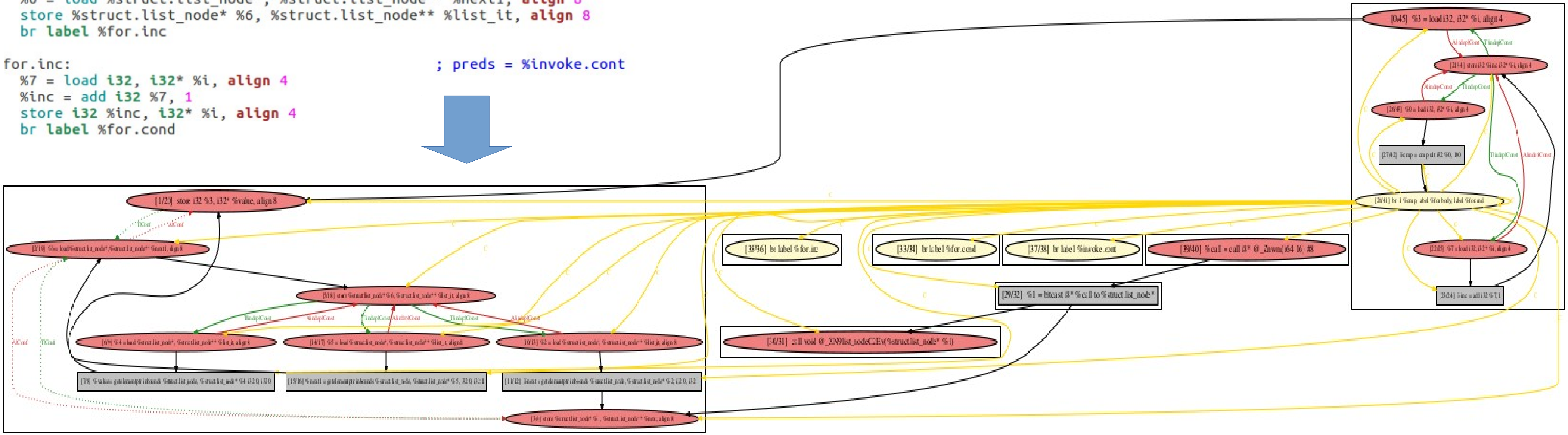
```llvm
for.cond7:                                        ; preds = %for.inc16, %for.end
  %4 = load i32, i32* %i6, align 4
  %cmp8 = icmp ult i32 %4, 100
  br i1 %cmp8, label %for.body9, label %for.end18

for.body9:                                        ; preds = %for.cond7
  %5 = load i32, i32* %i6, align 4
  %idxprom10 = zext i32 %5 to i64
  %arrayidx11 = getelementptr inbounds [100 x i32], [100 x i32]* %a, i64 0, i64 %idxprom10
  %6 = load i32, i32* %arrayidx11, align 4
  %7 = load i32, i32* %i6, align 4
  %idxprom12 = zext i32 %7 to i64
  %arrayidx13 = getelementptr inbounds [100 x i32], [100 x i32]* %b, i64 0, i64 %idxprom12
  %8 = load i32, i32* %arrayidx13, align 4
  %add = add i32 %6, %8
  %9 = load i32, i32* %i6, align 4
  %idxprom14 = zext i32 %9 to i64
  %arrayidx15 = getelementptr inbounds [100 x i32], [100 x i32]* %c, i64 0, i64 %idxprom14
  store i32 %add, i32* %arrayidx15, align 4
  br label %for.inc16

for.inc16:                                        ; preds = %for.body9
  %10 = load i32, i32* %i6, align 4
  %inc17 = add i32 %10, 1
  store i32 %inc17, i32* %i6, align 4
  br label %for.cond7
```



# Case study [2] : non-parallelisible loop

```
for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, i32* %i, align 4
  %cmp = icmp ult i32 %0, 100
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %call = call i8* @_Znwm(i64 16) #8
  %1 = bitcast i8* %call to %struct.list_node*
  call void @_ZN9list_nodeC2Ev(%struct.list_node* %1)
  br label %invoke.cont

invoke.cont:                                      ; preds = %for.body
  %2 = load %struct.list_node*, %struct.list_node** %list_it, align 8
  %next = getelementptr inbounds %struct.list_node, %struct.list_node* %2, i32 0, i32 1
  store %struct.list_node* %1, %struct.list_node** %next, align 8
  %3 = load i32, i32* %i, align 4
  %4 = load %struct.list_node*, %struct.list_node** %list_it, align 8
  %value = getelementptr inbounds %struct.list_node, %struct.list_node* %4, i32 0, i32 0
  store i32 %3, i32* %value, align 8
  %5 = load %struct.list_node*, %struct.list_node** %list_it, align 8
  %next1 = getelementptr inbounds %struct.list_node, %struct.list_node* %5, i32 0, i32 1
  %6 = load %struct.list_node*, %struct.list_node** %next1, align 8
  store %struct.list_node* %6, %struct.list_node** %list_it, align 8
  br label %for.inc

for.inc:                                          ; preds = %invoke.cont
  %7 = load i32, i32* %i, align 4
  %inc = add i32 %7, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond
```

```
list_node_t begin;
list_node_t* list_it;

list_it = &begin;
for (unsigned int i = 1; i < size; i++) {

    list_it->next = new list_node_t;


    list_it->value = i;


    list_it = list_it->next;
}
```



# Case study [2] : non-parallelisible loop

```
list_node_t begin;
list_node_t* list_it;

list_it = &begin;
```

Iterator

```
for (unsigned int i = 1; i < size; i++) {
```

```
    list_it->next = new list_node_t;
```

T

A

```
    list_it->value = i;
```

Write

Read

```
    list_it = list_it->next;
```

Read

Write

A

```
}
```

T

Payload

```
list_node_t begin;
list_node_t* list_it;

list_it = &begin;
for (unsigned int i = 1; i < size; i++) {
```

Read

```
    list_it->next = new list_node_t;
```

Write

Write

```
    list_it->value = i;
```

Read

```
    list_it = list_it->next;
}
```

# General work observation

- To judge about true program parallelisability, we have to disassemble the program (its program dependence graph) to the smallest finest-grain pieces

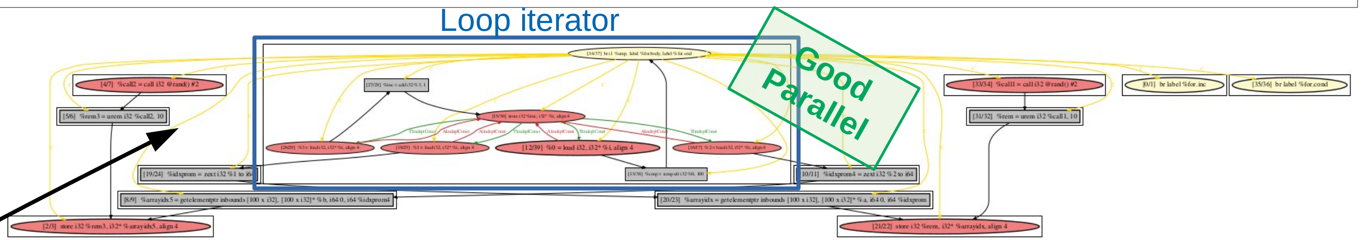
- As it appears at the moment, the true indicator of the loop parallelisability is the absence of strongly connected components (besides the iterator one) of the size, greater than 1 instruction: there are cycles present in such strongly connected components, which imply dependence between instructions tying up/entangling  loops and preventing them from parallelisation

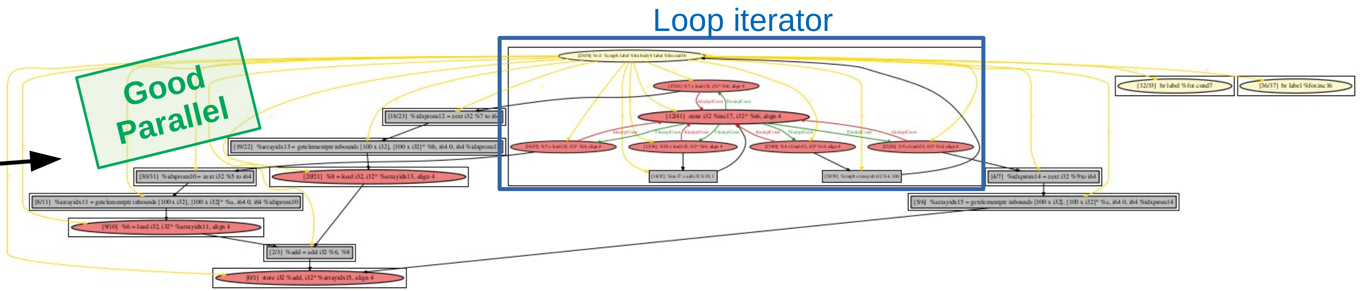
- According to code comments, there is a work going on in LLVM on building llvm::DependenceGraph and breaking its edges with loop splitting and peeling.

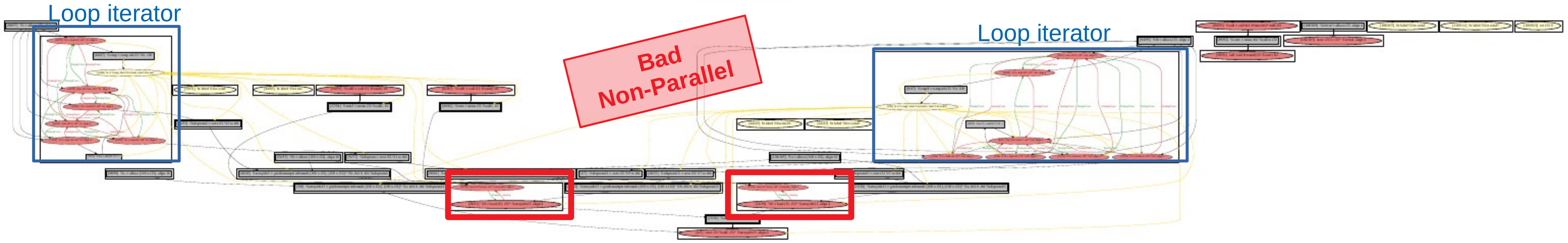
# General work observation : manifestation [1]

- To judge about true program parallelisability, we have to disassemble the program (its program dependence graph) to the smallest finest-grain pieces

```cpp
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace std;

static const unsigned int size = 100;
static const unsigned int range = 10;

int main() {
    unsigned int a[size];
    unsigned int b[size];
    unsigned int c[size];

    std::srand(std::time(nullptr));

    // initialization loop
    for (unsigned int i = 0; i < size; i++) {
        a[i] = rand() % range;
        b[i] = rand() % range;
    }

    // vector sum computation
    for (unsigned int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```



# General work observation : manifestation [1]