# **PARTITA** parallelization and code generation

Laurent HASCOET
Simulog Sophia-Antipolis

## 1 Introduction

As computer technology evolves, languages evolve too. One important evolution was the arrival of supercomputers for numerical computation, with vectorial, then parallel technology. This evolution mainly affected the *FORTRAN* language, since it is widely used in the world of intensive computing.

In one sense, it is a pity that the language must change when the machine changes. These changes should be endorsed by the compiler. However, in another sense, it is good to change the language when changes improve the readability of programs (this was the case when structured programming appeared), or their conciseness (this is the case with vectorial notation). Moreover, it is unavoidable to modify the language to express new ideas that were not expressed before, such as the parallelism opportunities contained in an algorithm. In the current state of the art, it is not possible to find out automatically all the parallelism contained in a program.

So a new work appeared, that is to transform a *FORTRAN* program towards a new dialect of *FORTRAN*, with new constructs expressing parallelism. In this process, one must take into account some characteristics of the target supercomputer. Opportunities to simplify the program and to promote new structured constructs should not be neglected.

The **PARTITA** system is dedicated to help the user in this difficult task. It is based on the **PARTITA** analyses. It takes as input a parametrable description of the target machine. It provides the user with two main results:

- On the one hand, it shows on the original program the opportunities for vectorization or parallelization. The user may ask for explanations, and understand why the system eventually didn't find what he expected.

- On the other hand, it shows a new generated parallelized program. This program is automatically generated, using the parallelization opportunities, and is adapted to the *FORTRAN* target dialect.

In the next chapters, we shall present the algorithms running in this process. They are separated in two main phases, one called **detection**, since it detects parallelization opportunities, and the other called **generation**, since it generates a new piece of *FORTRAN* code. Another aspect is the user interface, whose main task is to present and explain the results of **detection** in a convenient manner.

For readability, we shall abbreviate the **op-ref:** type prefix to a single colon, so that **op-ref:group:loop** becomes **:group:loop**, for example.

## 2 Parallelism detection

The fundamental tool for parallelism **detection** is the data dependence graph. In **PARTITA**, this graph is just included into the data flow graph (*dfg*). The basic rule is that cycles in the *dfg* correspond exactly to non-parallelizable parts.

The reason is intuitively clear. Cycles in the *dfg* come from loops. If the program control flow could only go downwards, there could not be a cycle in the data dependencies, since there could not be dependencies going upwards in the control flow. If there is a *dfg* cycle between two instructions in a loop, it means that one cannot perform all the first instructions, then all the second instructions, or conversely, since this would violate at least one dependence arrow in the *dfg*. But this is what might happen if the loop was done in parallel mode.

The classical algorithm to find cycles is called the *Tarjan* algorithm, and is very cheap and efficient. The main technical difficulties come with what is around cycle detection:

- Nested loops must be taken into account. Their parallelization status is not necessarily the same at all levels. So there must be *levels* in the data dependence graph, so that there may be a cycle at some level and not at another.

- In some cases, a non-parallel (or *sequential*) part, *i.e.* a cycle in the *dfg*, can be considered as a parallel one. For instance induction variables are inherently sequential, even if they are a linear function of the loop counter, which is itself an induction variable. But every parallel or vectorial machine has a built-in way to count from 0 up to $n$ in a non-sequential manner (For example with the vectorial notation $0 : n : 1$). This is also applicable to *reduction* instructions.

- Some sequential parts can be automatically transformed to remove some data dependencies, for example by *scalar expansion.* After that, the cycle may disappear, and a new parallel component may be detected.

- *Static code*, code that is exactly the same at each iteration of a loop, must be detected to be later factored out of the loop.

- *Unused dead code*, dead because it computes a value that ultimately nobody uses, must be detected and removed from the unit.

- Two consecutive sequential sections may and must be merged into a single one. The same applies to vectorial sections and, with some restrictions, to parallel sections.

## 2.1 *Unused dead code*

Let us first present the way we detect the *unused dead code.* By definition, the result of a *FORTRAN* **unit** is composed of the exit values of:

- The return value of the function name, for a `function`.

- The formal parameters.

- The variables in `common`, even if these `common`'s are used only in called subroutines.

- All the input and output streams.

All these values have received a *dfg* dependence towards the special unique **:atomic:exit** of the unit. Therefore the *unused dead code* corresponds to the **:atomic**'s that do not influence by *use* data dependence the **:atomic:exit**.

To find out the *unused dead code*, we simply start from the **:atomic:exit**, and go upwards the *dfg* arrows that express the use of a value. Those are the *direct*, *value*, and *control* dependencies. All visited **:atomic**'s are marked as used. At the end, the *unused dead code* consists of the **:atomic**'s not marked.

*Unused dead code* is not removed from the *dfg*. It is simply marked as such. Later algorithms will simply avoid it, so that it will not appear in the final parallel informations, or in the generated code.

All the *unused dead code* is available in a special field of the main data structure (**flow-graph**).

## 2.2 *The nested levels of groups*

Let us now present the way we build the *levels* in the *dfg*. Since other tools may use it, we must not modify the *dfg*. The levels will be built on top of it. The elements of these levels are called groups, and are of type **:group**.

For code generation, or simply when one thinks of the execution order of all **:atomic**'s stored in the *dfg*, the main characteristic of the **:group** is the following: When one starts executing a **:group** $G$, no other brother **:group** may be started until $G$ (and all its sub-groups) is completely executed.

Moreover, the execution order of the sub-groups of a given **:group** is determined, first by its kind (sequential, parallel, etc...), and second by the order implied by the data-flow dependencies between the sub-groups. Therefore, we will need a simple and efficient way to precompute what dependencies between sub-groups are implied by the original *dfg* dependencies, known as **d-arrows**.

Here follows the sub-type hierarchy of the **:group**'s.

- **:group:loop**'s are groups whose contents must be executed repeatedly, according to a loop control. They usually come from instructions that were in a loop in the original code, so that they must still be in some kind of a loop in the generated code There are subtypes, according to the kind of loop control one may use.

  - **:group:loop:sequential** denotes loops that must be implemented as classical, sequential loops. The order of the iterations must follow the order they had in the original, sequential, loop.
  - **:group:loop:parallel** denotes loops whose iterations may be executed in any order, and for instance in parallel.
  - **:group:loop:vectorial** denotes loops that may be replaced by (a sequence of) vectorial instructions. This execution order is slightly more restrictive than in parallel, since it guarantees that one vectorial instruction is completely finished before the next one starts.
  - **:group:loop:reduction** denotes those sequential loops which can be replaced by a special operation (*reduction operation*) which is implemented efficiently on the target architecture. Classical *reduction operations* are sum, product, extremum element of an array.
  - **:group:loop:induction** denotes the groups that compute an induction variable. These groups are sequential, but they will simply not be present in the generated code when all uses of the

induction variables are directly replaced by the induction expression.

  - **:group:loop:external** is the special case of the loop instructions that are constant, whatever the iteration. They may therefore be executed only once, and this saves time. In some sense, this is another kind of *dead code.*

- **:group:once**'s group **:atomic**'s together, just to indicate that they must be done together, in a single instruction. But this group does not require that this instruction be repeated in any manner.

We may now sketch how groups are created. First, every **:atomic** that is not dead, is put as a direct son of some **:group:once**. When two **:atomic**'s are sons of the same **:group:once**, it means that they must be done during the same instruction.

Then every **:group:once** may be inside a **:group:loop**, when the original instruction was in a loop, and so on for each enclosing loop. Thus the nesting of the **:group:loop**'s reflects the nesting of the loops.

Finally, all **:group:loop** of a top-level loop, and every **:group:once** of instruction not in a loop, are sons of a single **:group**, called the *top* group.

The algorithm to detect parallelism is to build an original tree of groups, above the *dfg*. This tree will reflect the behavior of the original program, with one **:group:once** for each instruction, and all **:group:loop**'s above of the **sequential** kind. Then, by splitting and regrouping the groups, this tree will be transformed into the final tree of groups, with their proposed parallel status.

## 2.3 Projection of the d-arrow's

To distinguish sequential groups from parallel ones, we need to analyze the cycles (*strongly connected components* or *scc*) of dependencies between groups. The nesting of **:group**'s allows us to say it more precisely. The natural objects to study cycles are the elements of groups, *i.e.* the direct sub-groups of a group. There is no question of cycles between the **:atomic**'s under an **:group:once**, since all dependencies there are inside a single instance of the instruction, and therefore show no cycle. A cycle may appear only between elements of an **:group:loop**. So we need to project the *dfg* arrows to obtain the dependencies between sub-groups of a given group. The type of the resulting dependencies will be called **s-arrow**.

- The first rule is that an **s-arrow** links two groups with the same father group. It makes no sense to put a **s-arrow** between two groups with different fathers, since their relative order is completely specified by the relative order of their fathers.

- The second rule is that a **d-arrow** projects to **s-arrow**'s at some loop levels, but not at every loop level. A **d-arrow** will have no projection at a level if is has projections of distance not 0 at the levels above.
- The third rule is that a **d-arrow** of distance 0 from a group to itself makes no sense, and therefore will not be set. This is because there cannot be a dependence from an iteration of an instruction to the same iteration of the same instruction.

We illustrate those rules with a simple case. Suppose we have two **:atomic**'s, A and B, both inside loop j, itself inside loop i. Suppose the **d-arrow** we want to project is of distance 0 with respect to i, and 1 with respect to j. Figure 1 shows the **s-arrow**'s obtained by projection on group trees of different shapes.

When A and B are in the same **:group:once** (case 1), then the projected distance indicates that in the same iteration of i, within iterations of loop j, the iteration $n$ must be done *before* the iteration $n + 1$. To illustrate, this forbids parallelization with respect to j, but vectorization is allowed, if all A's are done before all B's. When A and B are in different instructions, in the same loop with respect to j, the projected distance indicates just the same as above (case 2)

When only the loop with respect to i is common (case 3), the projected distance is 0, since the dependence occurs for the same value of i. In the loop with respect to i, for a given iteration, the instruction "loop j A" must be done before "loop j B".

When there are two "i" loops (case 4), the loop dealing with A must be done before the loop dealing with B. The projected distance must always be 0, since we are under the "Top" group, which is done only once, so a distance different from 0 makes no sense.

Of course, since projection highly depends on the current structure of the groups tree, each time this tree is modified by splitting a group or merging two groups, some **s-arrow**'s must be erased and a (partial) projection must be done again.

## 2.4 Strongly Connected Components

We are now in our beginning state, where there is one **:group:once** per **instr-node**, and one **:group:loop:sequential** for each loop. All **d-arrow**'s

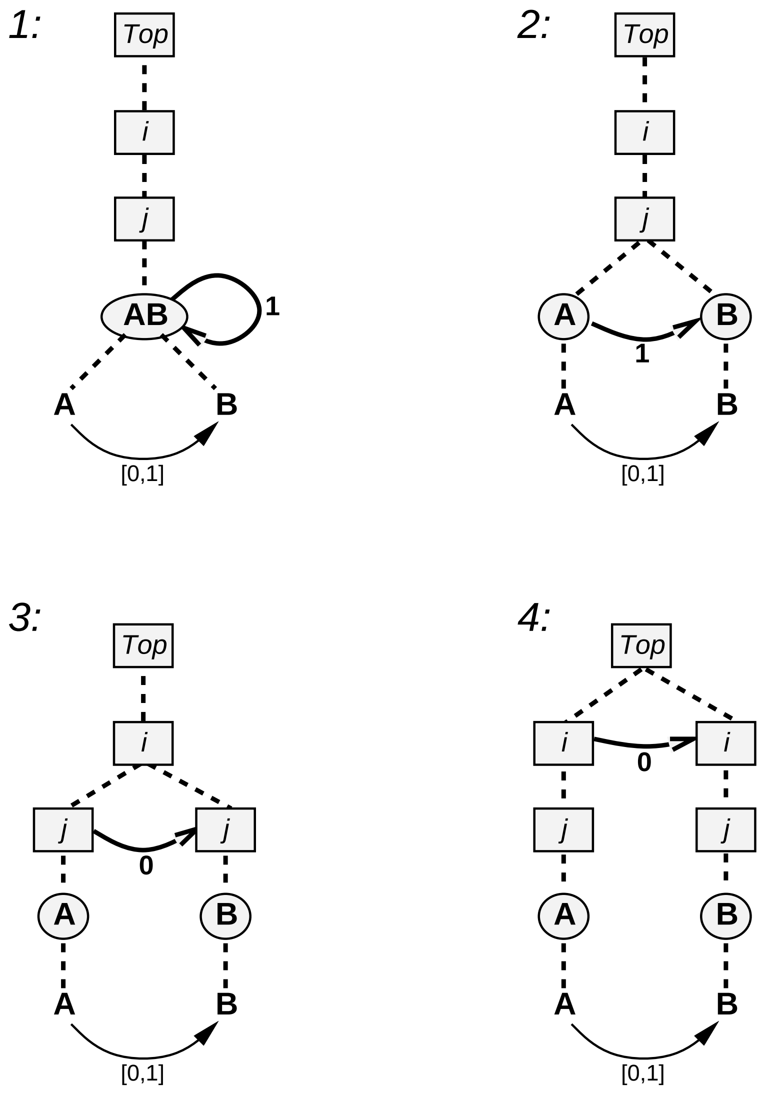

Figure 1: projection of a **d-arrow**

have been projected. We are going to split every group as much as possible, defining one new group for each *scc*.

First we split every **:group:once**, creating one for each **:atomic**. The only case where some **:atomic**'s still stay in a common **:group:once** is when *FORTRAN* syntax cannot separate them into different instructions. This is the case between a function `call` and its side-effects, for instance.

Then for every **:group:loop:sequential**, from the bottom up in the groups tree, we find *scc*'s. We then replace this **:group:loop:sequential** with as many groups as *scc*'s, sequential or parallel depending if the *scc* is a cycle or not.

## 2.5 Special cases

When we find a sequential **:group:loop**, we still have hope to improve things.

First if the sequential group defines an induction variable, then we give it subtype **:group:loop:induction**. The uses of the induction variable are disconnected from this group by removing the anti-dependencies from them to the write of the induction variable.

Else if the sequential group defines a reduction expression, then we give it subtype **:group:loop:reduction**.

Else if the sequential group can be improved (e.g. turned into parallel) with scalar expansion, then we perform this scalar expansion, and the *scc* analysis is run again. Note that we do not modify the source file nor the *dfg*. We only set some annotations, so that the generated code will feature the expanded arrays instead of the scalar.

In all other cases, the group is definitely sequential.

## 2.6 External code

We are able to detect code which is executed exactly in the same manner, with the same arguments at each iteration of a loop. The **:group:loop**'s that contain such code are retyped as **:group:loop:external**, and the corresponding generated code will be executed only once.

## 2.7 Regroup phase

This leaves us with a huge number of groups. Moreover, there are lots of parallel groups with a single **:atomic** inside. This would lead to an inefficient code, since we would generate loops performing a single operation, preceded by loads of temporary variables and followed by stores of temporary variables. For example for the expression `i+2` we would have one loop nest computing all i's and storing them into an intermediate array, then another loop nest loading this array, adding 2 to every element, and storing the result into another array. This is impraticable. So we need to regroup groups to reduce this load/store overhead to a minimum.

There is a trade-off since on the one hand, regrouping saves load/store operations, plus intermediate memory space, but regrouping a parallel group with a sequential group gives a sequential group, which is unfortunate for a parallelization tool. We have a cost/benefit mechanism to evaluate this trade-off. This is parametrized by a user-given description of the target machine characteristics.

Note that this regrouping is needed only between **:atomic**'s from a same **instr-node**, since there is no value communicated through the stack between two different instructions.

So the algorithm is

```
For each instr-node
   nest = the complete nest of op-ref's of the instruction
   For each pair of op-refs O1 and O2 in the nest, with
             a "value" dependence between them,
      For D = 1 upto the loop nesting depth of O1 and O2
        G1 = group of depth D above O1
        G2 = group of depth D above O2
        If G1 = G2
           Then do nothing
        ElseIf the group above G1 differs from the the group above G2
           Then do nothing
        ElseIf the merging of G1 and G2 gives a benefit
           Then merge G1 and G2
        Endif
      EndFor
   EndFor
EndFor
```

## 2.8 Vectorial status

During the above phase, each time a new parallel group shows up, we test if it is vectorial or not, i.e. can it be expressed in vectorial notation.

There are some subtleties since intermediate values are stored into new arrays, and we have the choice of the order of the array's indexes. But this order directly affects the vectorial status. So we try and choose the best order to improve vectorial status.

The vectorial status influences the above grouping phase in another manner. It may prevent the regroup of a parallel group and a vectorial group, since the resulting group would be "only" parallel.

Figure 2 shows the groups nesting for a program fragment, at the end of the Regroup phase. For readability, we didn't show all the **d-arrows**, but only those explaining the **scc**'s. The **:group:once** are shown by circles, the other groups are shown with rectangles. Thick arrows are the **s-arrows** obtained by projection.

## 3 Parallelized code generation

The detection phase has left us with a tree of groups, hopefully with a reduced number of **:group:loop**'s. This tree of groups reflects the structure of the future generated parallel code.

Each **:group:once** corresponds to a single instruction. If it has incoming (*resp.* outcoming) "value" **s-arrow**'s, this means that the original instruction has been split. Then we must generate the read (*resp.* write) of a temporary variable to implement this value dependence in the generated code.

Each **:group:loop** corresponds to an iterative control structure, to be put around the code generated for all its sub-groups. As we said before, once we start generating the code for a **:group**, we must generate all the code before we start another brother **:group**.

The subtype of the **:group:loop** indicates the kind of iterative control structure to be used. A classical `do` loop for a sequential group, the target language's parallel `do` for a parallel group, a triplet notation for vectorial groups. The reduction groups are written using the target machine's reduction intrinsic functions, and the external groups are put into no iterative control at all.

The isolated groups containing the loop counters, when they are parallel (most frequent case) correspond to no specific generated code. They correspond to the loop control put around each generated loop. When these groups are sequential, it means that the number of iterations must be determined dynamically, by a sequential loop. This is the case when there is a loop exit at some unknown iteration. Then this sequential group is generated just like normal ones, but it may store the number of iterations, for use by other groups.

## 3.1 Topological sorting

The only question remaining is the relative order of the sub-groups in their father's group code. This order may be any total order compatible with the partial order of the **s-arrow**'s. In other words, we must respect the constraints of the **s-arrow**'s between the sub-groups.

Therefore, there is a topological sorting phase, which tries to satisfy the following heuristics, from the most to the less important.

- Respect the **s-arrows**. This is compulsory. Exception: for sequential loops, **s-arrows** cycle. Then just follow the order of the original code.
- Try and put consecutively, vectorial (or reduction) instructions with the same `where` control , to reduce the `where` overhead.
- Try and put consecutively, vectorial instructions with one `where` control then vectorial instructions with the opposite `where` control. This to promote the `elsewhere`.
- Try and put consecutively, parallel instructions, especially when there is no dependence of distance not 0 between them, and regroup them in a single parallel `do` loop. This to reduce the parallel overhead, and to reduce the number of synchronizations inside a parallel loop.
- Try and put consecutively, sequential loops, and regroup them. This to reduce the `do` loop overhead.
- Try and reduce the amount of temporary variables used at the same moment.
- Put external code first or last, when possible. This is just for readability.
- Lastly, when we have the choice, it is best to keep the order of the original file. This is just for readability.

## 3.2 Control-Flow Distribution

Now we are able to sketch the code generation algorithm, for a given **:group:loop**. This algorithm generates a new **flow-graph** object. The generation of the actual **VTP** abstract syntax tree is done by the **FOREST** tool of **FORESYS**. Then from the abstract syntax tree, **FORESYS** easily produces the parallelized *FORTRAN* code.

We call our algorithm **Control-Flow Distribution**, because it is an extension of the method known as **Loop Distribution** to arbitrary control-flow-graphs.

We start considering the top **:group** and the list of all **block-nodes** of loop level 0. The following recursive function generates the new control-flow-graph. At the first call, `<header>` is empty.

```
 Function GENERATE-LOOP-BODY (<group>, <block-nodes>, <header>)
   <tail-block> = new block-node
   annotate "cycling" c-arrows of <header> with <tail-block>
   For each <block-node> in <block-nodes>, in reverse dfst order.
      If <block-node> is a loop header
         different from <header>
       Then
         LOOP-DISTRIBUTION (<block-node>, <group>)
       Else
         <proj-block> = new block-node containing the instructions
              generated for "once" sub-groups of <group>, when these
              instructions come from <block-node>.
         <destinations> = all block-nodes found on c-arrows
                          exiting <block-node>
         If <proj-block> is empty
            AND <destinations> has 1 element.
          Then
            <proj-block> = the element of <destinations>.
          Else
            When <destinations> has more than 1 element
               add a test at the end of <proj-block>.
            Set new c-arrows from <proj-block> to <destinations>.
         Endif
         annotate c-arrows arriving on <block-node> with <proj-block>.
      EndIf
   EndFor
   <first-block> = the block-node found on the arrow from
         <header> to the loop body.
   The result is the control-flow-graph fragement going from
      <first-block> to <tail-block>
End
```

The above algorithm generates the code for all the **:group:once** direct sons of `<group>`, and the code for the remaining groups, of type **:group:loop** is done by the call to procedure LOOP-DISTRIBUTION. LOOP-DISTRIBUTION recursively calls GENERATE-LOOP-BODY.

```
     Procedure LOOP-DISTRIBUTION (<header>, <father-group>)
       <sub-groups> = all :group:loop's in <father-group>,
that are groups with respect to <header>.
       sort <sub-groups> in the manner described above.
       For each <group> in <sub-groups>
          According to the type of <group>:
          * sequential: Put a sequential do directed by <header> around
             GENERATE-LOOP-BODY(<group>, block-nodes in <header>, <header>)
          * parallel: Put a parallel do directed by <header> around
             GENERATE-LOOP-BODY(<group>, block-nodes in <header>, <header>)
          * vectorial or reduction:
             generate the vectorial or reduction instruction.
          * induction:
             generate nothing
          * external:
             GENERATE-LOOP-BODY(<group>, block-nodes in <header>, <header>)
          End
       EndFor
     End
```

Again on figure 2, we can see the **s-arrows** between the groups. They are shown by the thick arrows at levels 1, 2 and 3. Those arrows must be respected by the order of the generated code. After **Control-Flow Distribution**, the generated code may look like:

```
...
sum = sum + SUM(A(1:100))
Tmp1(0:99) = B(1:100)*C(1:100)
Do i=1,100
  A(i) = A(i-1) + Tmp1(i-1)
  T(i,0:100:2) = T(i-2,0:100:2) + A(H(i))
  ...
EndDo
...
```

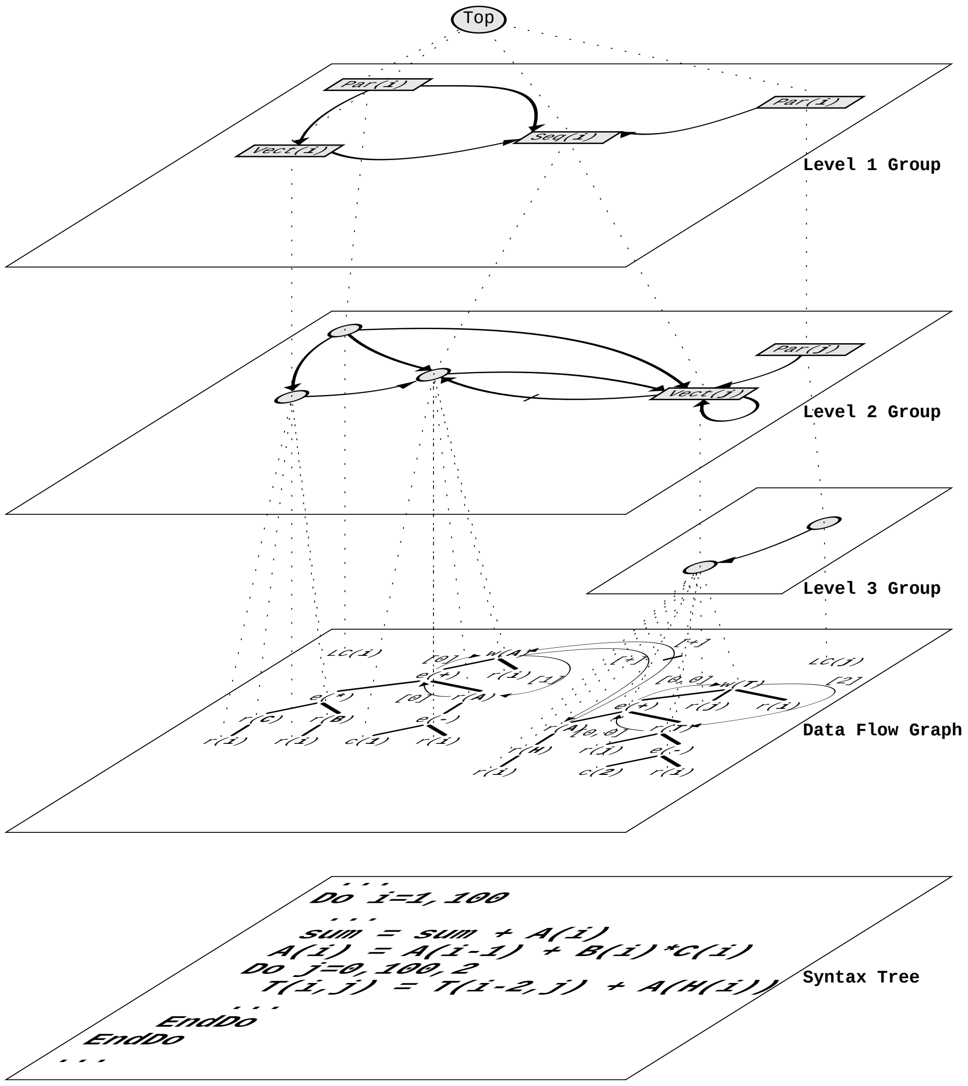

Figure 2: The groups structure

## 4 User Interface aspects

The result of **Parallelism Detection** is shown and explained with three levels of precision. The first level is the most concise, but the less precise.

The first level shows on the source abstract syntax tree, for a selected loops, the parts that are sequential, parallel, vectorial, etc. It just reaches for the levels of the groups tree that deal with this selected loop, then for each of them find the corresponding source text and then colors the background according to this parallellism status.

The second level shows the dependence graph itself. More precisely, it displays the **:group:loop**'s with respect to the selected loop, with the projected **s-arrow**'s between them. Since this graph may be too large for a good display, we provide ways to shrink it, replacing a number of nodes by a single small circle.

The third level displays, in a separate window, all the **d-arrows** between selected **:atomic**'s, with the type and the distance of all data dependencies shown.

For the **Code Generation** part, the only interface tool is a display of the generated code. The source-code correspondance is provided, using the following mechanism: From a point in the syntax tree, there is an annotation leading to the **:atomic**'s of this tree. Then each **:atomic** points to its enclosing **:group:once**. If there is none, it means this **:atomic** is dead, and there in no corresponding code. From the **:group:once**, we find all **:group:once**'s in source and generated codes. And then we go to their abstract syntax trees.

## A PARTITA Data Structures

Structure **flow-graph**:

| | |
|---|---|
| • **unit:** | The abstract syntax tree |
| • **symtab:** | Symbol table |
| • **entry:** | The **block-node:entry** |
| • **declarations:** | Non-instructions |
| • **exit:** | The **block-node:exit** |
| • **top-blocks:** | The **block-node**'s of level 0 |
| • **blocks:** | All the **block-node**'s |
| • **dead-code:** | All the dead **block-node**'s |
| • **pseudo-vars:** | New variables needed by analysis |
| • **global-dirs:** | Global directives |
| • **group:** | The **Top** group |
| • **dead-op-refs:** | The *unused* dead code |
| • **codes:** | The generated codes |

Structure **block-node**:

| | |
|---|---|
| • **rank:** | Rank of the **block-node** in the *dfst* |
| • **instructions:** | List of the **instr-nodes** |
| • **flow:** | **c-arrows** flowing from **block-node** |
| • **back-flow:** | **c-arrows** flowing to **block-node** |
| • **enclosing-loop:** | **block-node:header** of enclosing loop |
| • **context:** | Context of control |
| • **label:** | Original `goto` label of the **block-node**, if any |
| • **exit-induc:** | Induction variables information upon **block-node** exit |
| • **flow-graph:** | The **flow-graph** above |
| • **dominator:** | Dominator **block-node** |

Structure **block-node:header**:

| | |
|---|---|
| • **counter-ref:** | **op-ref:atomic:loop-counter** |
| • **counter-var-rank:** | Pseudo-variable for the loop counter |
| • **index-name:** | Index variable name, if any |
| • **izone:** | Index variable zone, if any |
| • **inside:** | List of the **block-node**'s directly in the loop |
| • **depth:** | Depth of the loop in the loop nesting |
| • **context:** | Context of control of the loop |
| • **push-pops:** | **c-arrows** that enter or exit the loop. |
| • **exit-induc:** | Induction variables information upon loop exit |
| • **killed-zones:** | zones completely overwritten by the loop |
| • **index-bounds:** | Bounds of the loop index values |
| • **max-cycles:** | A maximum of the number of cycles |
| • **scalar-expansions:** | Scalars expanded with respect to the loop |

Structure **block-node:entry**:

| | |
|---|---|
| • **in-out:** | in-out information through the unit |
| • **entry-bounds:** | Bounds information upon unit beginning |
| • **nest0:** | Nest of **op-refs** done before first call |
| • **nest+:** | Nest of **op-refs** done before next calls |
| • **nest*:** | Nest of **op-refs** done before all calls |
| • **nest-decls:** | Nest of all **op-refs** done during declarations processing |

Structure **block-node:exit**:

| | |
|---|---|
| • **exit-bounds:** | Bounds upon exit from the unit |
| • **visible-zones-nest:** | Nest of **op-refs** done before unit exit |
| • **all-zones-nest:** | Special exit nest where all zones are exported |

Structure **c-arrow**:

| | |
|---|---|
| • **origin:** | The **block-node** at the origin of the arrow |
| • **destination:** | The **block-node** at the end of the arrow |
| • **change:** | The value to propagate along the arrow |
| • **iter:** | Push'ed and Pop'ed loops |
| • **type:** | Type of the arrow |

Structure **instr-node**:

| | |
|---|---|
| • **vtp:** | Abstract syntax tree of the instruction |
| • **Rpath:** | Path to the instruction |
| • **block:** | The **block-node** containing this **instr-node** |
| • **nest:** | The nest of all **op-ref:atomic**'s of the instruction |
| • **induc-info:** | Induction variables information, just before the instruction |
| • **bounds-in:** | Bounds information, just before the instruction |
| • **dir-in:** | Directives valid just before the instruction |

Structure **op-ref**:

| | |
|---|---|
| • **in-deps:** | The arriving **d-arrows** or **s-arrows** |
| • **scc:** | The **op-ref** just above in the groups tree |

Structure **op-ref:atomic**:

| | |
|---|---|
| • **vtp:** | The corresponding abstract syntax tree |
| • **Rpath:** | The path to the abstract syntax tree |

Structure **op-ref:atomic:rw**:

| | |
|---|---|
| • **varinfo:** | Information about the variable name |
| • **syslin:** | Polynomes corresponding to the array indexes |
| • **total:** | Are all zones cells accessed? |
| • **mask-from-start:** | The *mask* from the beginning of the **block-node** |
| • **mask-till-end:** | The *mask* till the end of the **block-node** |
| • **vect-status:** | Vectorial status of the **op-ref** |

Structure **op-ref:atomic:loop-counter**:

| | |
|---|---|
| • **loop-header:** | The **block-node:header** of the loop |

Structure **op-ref:atomic:call**:

| | |
|---|---|
| • **params-sig:** | Read-Write signature of the formal parameters |
| • **read-commons-sig:** | Read signature of the commons |
| • **write-commons-sig:** | Write signature of the commons |
| • **external-common-sig:** | Signature for variables not declared in the unit |
| • **mask-from-start:** | The *mask* from the beginning of the **block-node** |
| • **mask-till-end:** | The *mask* till the end of the **block-node** |
| • **func-name:** | Name of the subroutine |

Structure **op-ref:group**:

| | |
|---|---|
| • **min:** | Used by *Tarjan algorithm* |
| • **out-deps:** | The exiting **s-arrows** |

Structure **op-ref:group:once**:

| | |
|---|---|
| • **nest:** | sub-nest containing the same **op-ref**'s |
| • **instr-node:** | The **instr-node** above |
| • **cost:** | The total estimated cost of all operations in the group |
| • **vect-status:** | The vectorial status of the group |
| • **codes:** | List of the corresponding **op-ref**'s in the generated codes |

Structure **op-ref:group:loop**:

| | |
|---|---|
| • **elements:** | List of sub-groups |
| • **header:** | Corresponding loop **block-node:header** |
| • **infos:** | Optional additional infos |

Structure **op-ref:group:loop:parallel**

Structure **op-ref:group:loop:vectorial**

Structure **op-ref:group:loop:sequential**

Structure **op-ref:group:loop:reduction**

Structure **op-ref:group:loop:induction**

Structure **op-ref:group:loop:external**

Structure **d-arrow**:

| | |
|---|---|
| • **origin:** | **op-ref** at the origin of the arrow |
| • **type-list:** | List of the types and distances of dependences |

Structure **s-arrow**:

| | |
|---|---|
| • **origin:** | Origin of the projected dependence |
| • **destination:** | Destination of the projected dependence |
| • **distance:** | Projected types and distances |