An Approach to Map Sequential Applications onto Multi-Processor Platforms

Giso Dal (0752975)

August 29, 2010

Contents

1 Introduction **3**

2 Background **3**

2.1 Kahn Process Networks 4
2.2 Polyhedrons 4
2.3 Tool Flow 8

3 Implementation **11**

3.1 Goals 11
3.2 Updated Tool Flow 12
3.2.1 Overview 12
3.3 Process Network Transformations 19
3.3.1 Plane Cut 23
3.3.2 Modulo Unfolding 25
3.3.3 Merge 28
3.4 Assumptions 30
3.5 Unimplemented 31
3.6 Implementation Problems 33
3.7 Usage 34
3.8 Partition File 34

4 Experiments **36**

5 Conclusion **38**

Abstract

Sequentially specified applications are not optimized to run on multi-processor platforms. Kahn Process Networks can be used to represent a sequential application and with it, we are able to compute a mapping onto a multi-processor platform. A Kahn Process Network is a graph representation in which the nodes represent the function calls (or processes) of the sequential application, and the edges are unbounded FIFO channels. Respecting the dependencies which are analyzed by the PN compiler, process partitioning transformations can be applied to optimize the network for a given platform. Three different transformations have been implemented and will be discussed, namely plane cut, modulo unfolding and merge.

1 Introduction

The partitioning process of sequential components is both time-consuming and difficult. The Kahn Process Network model of computation is used as a representation of sequential nested-loop programs: each node in the network corresponds to a function call statement in the sequential application. Compilers produce the control code for the synchronization of the partitioned processes, ensuring correct execution. In Section 2 some background is discussed which is be needed to understand the transformations. The implementation uses other tools, like `PN`[4] and `CLooG`[2]. In Section 3 the implementation is discussed, giving an overview of the tool flow using the tools and a detailed description of the inner workings of the transformation software. Also, three different transformation techniques are explained by example in this section. The transformation software has been tested, of which the results are discussed in Section 4.

2 Background

Before the implementation of the partitioning software is discussed, some knowledge is needed to understand how it works. Kahn Process Networks are explained and polyhedrons which describe iteration domains. Afterwards, the initial tool flow is given, along with a short description of each component.

2.1 Kahn Process Networks

Kahn Process Networks[3] is a distributed model of computation for sequential processes which communicate through unbounded FIFO channels deterministically. The network is not dependent on computation or communication delays, making it useful for modeling embedded systems. In our case, the nodes represent the function calls. An example of a Kahn Process Network is given in Figure 1.

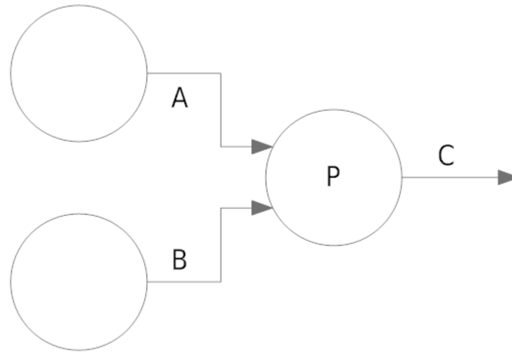

Figure 1: A Kahn process network of three processes without feedback communication. Edges A, B and C are communication channels. One of the processes is named process P.

2.2 Polyhedrons

A subset of sequentially specified applications can be represented by the Polyhedral model. We use this model to create a mapping to multi-processor platforms. Each function call in a sequential application has a domain. This domain can be translated into a polylib matrix consisting of integers, which we will refer to as a polyhedron. The syntax of a row in a polyhedron is consistent and can be viewed as follows, when using n iterators I and m parameters P:

$$\begin{pmatrix} \geq / = & I_1 & I_{..} & I_{n-1} & I_n & P_1 & P_{..} & P_{m-1} & P_m & constant \\ .. & a & .. & b & c & d & .. & e & f & g \\ .. & .. & .. & .. & .. & .. & .. & .. & .. & .. \end{pmatrix}$$

Where a-$g \in \mathbb{Z}$. The first and last column always represent the same. The first column contains either the number 1 which stands for $\geq$ or the number 0 which stands for $=$. After the first column, the iterators that are used in the program are enumerated. After the iterators, the parameters are enumerated which make the domain of a function call variable. The last column is a constant integer. The meaning of each row can be derived when taking the addition of every column. The above polyhedron translates to $a \cdot I_1 + .. + b \cdot I_{n-1} + c \cdot I_n + d \cdot P_1 + .. + e \cdot P_{m-1} + f\ cdot P_m + g >= 0$, given that the first column contains 1. With this in mind we can create the needed functions that represent the domain of a function call. Take for example the following C-style code:

```
void function_call();

int i;
for(i = 0; i < 100; i = i + 1){
    function_call();
}
```

When we look at the function $function_call()$, we can see that it is repeated 100 times, due to the for loop and its iterator i. The domain D of this statement can be written as $\{0 \leq i \ \wedge \ i \leq 99\}$. This can be translated into a polyhedron of two rows when we split our domain in to two inequalities which cover the entire domain. These inequalities are $\{0 \leq i\}$ and $\{i \leq 99\}$. One row must represent the lower bound of iterator i and the other row must represent the upper bound of iterator i.

$$\begin{pmatrix} \geq / = & i & constant \\ 1 & 1 & 0 \\ 1 & -1 & 99 \end{pmatrix}$$

We will now compute the meaning of each row by taking the addition of every column. The first row is translated into

$$\begin{aligned} & 1 \cdot i \ + \ 0 \geq 0 \\ \equiv \ & i \geq 0 \end{aligned}$$

The second row is translated into

$$
\begin{aligned}
& -1 \cdot i \ + \ 99 \geq 0 \\
\equiv \ & 0 \ + \ 1 \cdot i \leq 99 \\
\equiv \ & i \leq 99
\end{aligned}
$$

Thus, the polyhedron covers the domain of the function call. We will now look at a more complicated example, where iterators depend on variables. Using a pragma, we can define a constant to have an upper and lower bound, which makes it variable. In the next example the constants M and N have a value between 10 and 100.

```
void function_call1();
void function_call2();

#pragma parameter M 10 100
#pragma parameter N 10 100
int i, j;
for(i = M - 1; i < 100; i = i + 1){
    for(j = M; j < N - 1; j = j + 1){
        function_call1();
        function_call2();
    }
}
```

In this example both function calls have the same domain $D = \{M - 1 \leq i \leq 99 \ \wedge \ M \leq j \leq N - 2\}$. The polyhedron for this example consists of a lower and upper bound for iterator i and iterator j.

$$
\begin{pmatrix}
\geq / = & i & j & M & N & constant \\
1 & 1 & 0 & -1 & 0 & 1 \\
1 & -1 & 0 & 0 & 0 & 99 \\
1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & -1 & 0 & 1 & -2
\end{pmatrix}
$$

When we check this polyhedron, we can see that it covers the entire domain

of the first and second function call.

$$\begin{aligned} row\ 1 &\equiv 1 \cdot i \ + \ 0 \cdot j \ + \ -1 \cdot M \ + \ 0 \cdot N \ + \ 1 \geq 0 \\ &\equiv i \ + \ -M \ + \ 1 \geq 0 \\ &\equiv i \geq 0 + M \ - \ 1 \\ &\equiv i \geq M \ - \ 1 \end{aligned}$$

$$\begin{aligned} row\ 2 &\equiv -1 \cdot i \ + \ 0 \cdot j \ + \ 0 \cdot M \ + \ 0 \cdot N \ + \ 99 \geq 0 \\ &\equiv -i \ + \ 99 \geq 0 \\ &\equiv 0 \ + \ i \leq 99 \\ &\equiv i \leq 99 \end{aligned}$$

$$\begin{aligned} row\ 3 &\equiv 0 \cdot i \ + \ 1 \cdot j \ + \ -1 \cdot M \ + \ 0 \cdot N \ + \ 0 \geq 0 \\ &\equiv j \ + \ -M \ \geq 0 \\ &\equiv j \geq 0 \ + \ M \\ &\equiv j \geq M \end{aligned}$$

$$\begin{aligned} row\ 4 &\equiv 0 \cdot i \ + \ -1 \cdot j \ + \ 0 \cdot M \ + \ 1 \cdot N \ + \ -2 \geq 0 \\ &\equiv -j \ + \ N \ - 2 \geq 0 \\ &\equiv 0 + j \leq N \ - \ 2 \\ &\equiv j \leq N \ - \ 2 \end{aligned}$$

A polyhedron, however, does not depict the order in which the statements are executed. `CLooG` has solved this shortcoming by introducing scattering functions[1].
Scattering stands for scheduling, allocation, chunking functions. Scattering functions are needed when integral points of a domain have dependencies among one another. If a user wants to set precedence constraints between the statements of the following code, two scattering functions have to be introduced.

```
void function_call1(int x);
void function_call2(int x);

int i;
for (i = 1; i <= N; i++){
    function_call1(i);
}
for ( i = 1; i <= N; i++){
    functions_call2(i);
}
```

The scatting functions which depict the order of the function calls in the example above, are the following:

```
T_function_call1(i) = (1),
T_function_call2(i) = (2);
```

This scattering means that each integral point of the domain of *function_call1()* is scanned at logical date 1 while each integral point of the domain of *function_call2()* is scanned at logical date 2. As a result, all instances of *function_call1()* are executed before the instances of *function_call2()*.
The user can define any order of the function evaluations. The scattering functions are defined for every domain and can be multi-dimensional. When we look at the more complicated example code, that was given for which polyhedrons where created, the scattering functions are given by:

```
T_function_call1(i,j) = (0,i,0,j,0),
T_function_call2(i,j) = (0,i,0,j,1);
```

We will however not concern ourselves with scattering functions, because these handle dependencies that will be analyzed by one of the tools that is used when performing transformations on sequential applications.

2.3 Tool Flow

The process partitioning software uses other tools. The initial tool flow is given in Figure 2, after which the tools that have been used will be discussed.

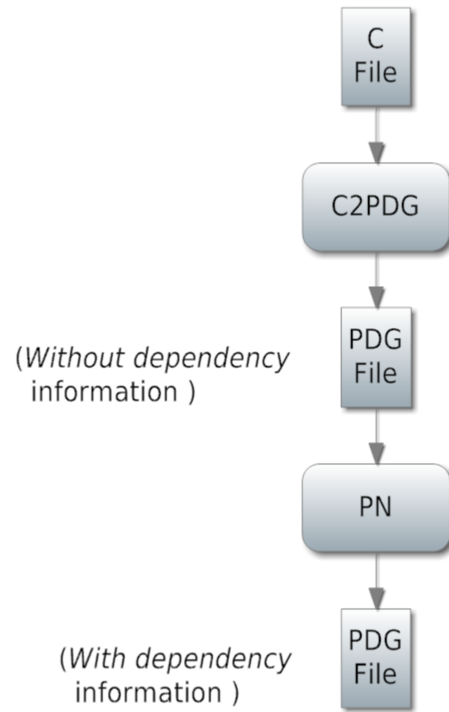

Figure 2: Initial tool flow

C2PDG

`C2PDG` takes as input a sequential program written in C. Only a subset of C is supported. There are a couple of restrictions, which will now be listed.

- Only function calls are valid statements
- Function calls have no return value
  - All functions are declared **void**, outputs are handled by function arguments
  - Function arguments which are declared as pointers to a type, are outputs
  - Function arguments which are not declared as pointers to a type, are inputs

The tool then takes this C input file and generates a Polyhedral Dependence Graph or PDG, without the dependency information. A PDG is a Kahn Process Network in which every node represents a function call from the input file. The data structure which represents the PDG includes the line number of each function call and their domain, which is a polyhedron. Also, a *prefix* is generated that indicates the position of each function call in relation to other function calls. With the prefix we can also reproduce if, for instance, the function call is located in an **if** statement within a **for** loop. The generated PDG is then written as a *YAML* formatted output file.

PN

The `PN`[4] compiler takes the output of `C2PDG` and adds the data dependency information to the PDG. It takes the *YAML* file and stores the information in a data structure similar to the data structure `C2PDG` uses. The dependencies are analyzed and inserted into the data structure. Afterwards, the now complete PDG is also outputted as a *YAML* file. Optionally, the output file generated by PN can be passed to `DOTTY`, which creates a visual representation of the PDG. The `ESPAM` tool can be used to map the PDG to a given platform.

CLooG

`CLooG`[2] stands for Chunky Loop Generator. Although not present in the tool flow presented, `CLooG` is used in the tool chain. We do not look at the extended tool chain, as it is not important for the the implementation of the partitioning software. `C2PDG` generates polyhedra for the iteration domains of C code. CLooG does the opposite. It takes the polyhedra and generates **for** loops, **if** statements and function calls, i.e. C code. We need this functionality when implementing the partitioning software. CLooG is specially designed to avoid control overhead and to produce very efficient code.

3 Implementation

The implementation of the process partitioning software uses other tools, which are written in the programming language C. Therefore, the partitioning software is also written in C. For convenience, the process partitioning software is referred to as `handle` from now on. In this section the tool flow and each tool used will be discussed. Also, each implemented transformation will be explained. Afterwards, we discuss the assumptions that have been made and possible improvements that can be made.

3.1 Goals

The initial goal was to make an implementation to automatically create a mapping of sequential applications onto multi-processor platforms using Kahn Process Networks. A list of steps describes all the tasks that need to be performed.

1. sequential application ⇒ PDG
2. loop until all partition combinations have been generated
   (a) partition PDG
   (b) evaluate partitioned PDG
   (c) remember best evaluated partitioning
3. best evaluated partitioning ⇒ partitioned application

Unfortunately, this proved to be too time consuming to implement, with the given time constraints. What `handle` has been programmed to do, is to partition sequential software by manual input and generate the partitioned sequential application, which results in the following tasks.

1. sequential application ⇒ PDG
2. analyze manually written transformation input
3. perform transformation on PDG
4. generate partitioned sequential application

3.2 Updated Tool Flow

This section discusses the updated tool flow. First, a flow chart is given. Afterwards, each component is discussed individually (except for the components of the original tool flow).

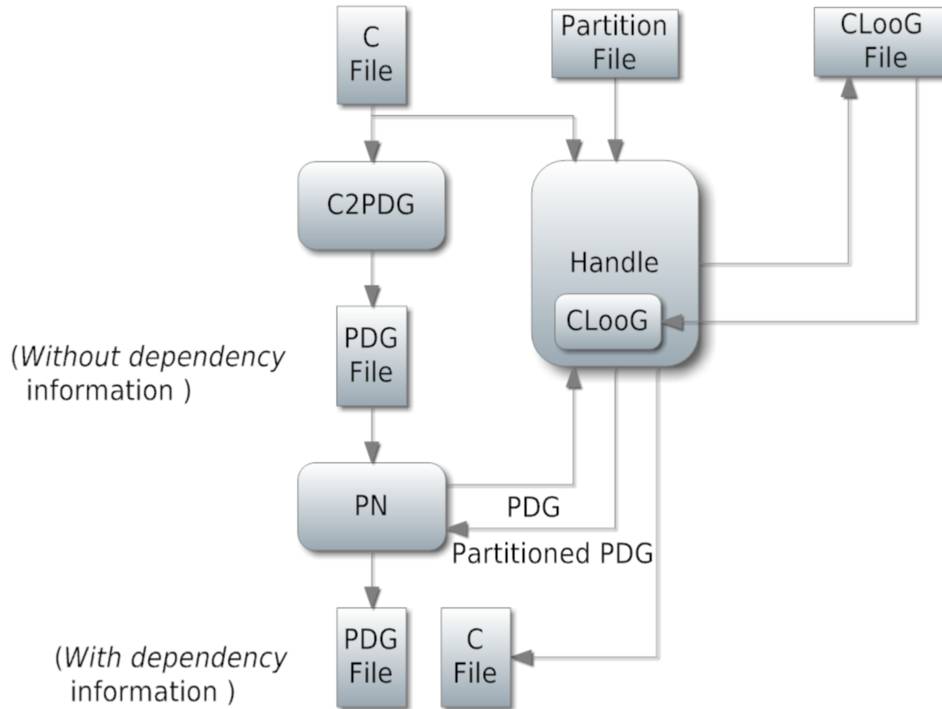

Figure 3: Tool Flow with partitioning software

3.2.1 Overview

The input *C File* is passed to `C2PDG`, which creates a *YAML* formatted output file with the information of the PDG, without the dependency information. `PN` parses the file generated by `C2PDG`, and transfers the information into a PDG data structure. Without having analyzed the dependencies, the PDG is passed to `Handle`. `Handle` parses the manually written *Partition File* and uses it to apply transformations on the PDG, when there are plane cuts or modulo unfoldings among the transformations. The optionally partitioned PDG is passed back to `PN` for dependency analysis. The now complete PDG is

passed back to `Handle`. Before writing the output file, merge transformations are handled by creating a mapping for the `CLooG` tool. This will be discussed later. `C2PDG` creates polyhedrons for every function call. These polyhedrons need to be changed back to valid C code. This is what `CLooG` does. `Handle` passes the polyhedrons of the PDG to `CLooG` with the *CLooG File*, which then generates **for** and **if** statements accordingly. `CLooG` however, does not have the actual function calls. It only knows the number of the function call. `Handle` takes the generated code and the numbers of the function calls that `CLooG` wants to print. It then opens the input *C File* and looks up the function call with the line number that is specified in the PDG, which is then printed instead of the number of the function call. This alone does not generate a valid C file, so all lines of the input *C File* before the main function are copied to the output *C File* accompanied by the declarations of all variables used by the program.

CLooG

`CLooG` takes an input file. The *CLooG File* consists of three main parts. A context section, a statement section and a section for the scattering functions. When the symbol # occurs, the right hand side is a comment. The context section contains the programming language and all parameters that are used. Also, an option is given to enter the parameter names manually. For future purposes, we will use the following example throughout the remained of this article.

```
void function_call1(int *x, int y);
void function_call2(int *x, int y);
void function_call3(int *x, int y);

int i, j, z;
for(i = 0; i < 10; i = i + 1){
    for(j = 0; j < 10; j = j + 1){
        function_call1(&z, z);
        function_call2(&z, z);
        function_call3(&z, z);
    }
}
```

In our case, no parameters are used, creating the following block.

```
# ----------- CONTEXT --------------
C # programing language

# context polyhedron:
1  2  # 1 row, 2 columns
1  0  # 0 >= 0, always true

0 # do not set parameter names manually
```

The statement section consists of all the function call polyhedrons. It is the first time the number of domains per statement pops up. This option is used for the **merge** transformation. When merging nodes, the polyhedrons will be printed as domains of one statement. For instance, when we merge the second and third function call of our example code, the *CLooG File* will contain the following.

```
# ---------- STATEMENTS -------------
2 # number of statements

1 # statement 1: 1 domains
4  4  # 4 rows, 4 columns
1  1  0  0
1 -1  0  9                 ( domain of statement 1         )
1  0  1  0                 ( polyhedron of function call 1 )
1  0 -1  9

2 # statement 2: 2 domains
4  4  # 4 rows, 4 columns
1  1  0  0
1 -1  0  9                 ( first domain of statement 2   )
1  0  1  0                 ( polyhedron of function call 2 )
1  0 -1  9

4  4  # 4 rows, 4 columns
1  1  0  0
1 -1  0  9                 ( second domain of statement 2  )
1  0  1  0                 ( polyhedron of function call 3 )
1  0 -1  9

1 # set iterator names manually
i  j
```

The scattering function section consists of all scattering functions, which are analyzed by `PN`. When a merge has been performed, only the scattering function of the first function call to be merged will be printed. `PN` generates these and are then printed by `Handle` to the *CLooG File*.

```
# ------------- SCATTERING -------------
2 # number of functions

# statement 1
2  6  # 2 rows, 6 column
0  1  0 -1  0  0
0  0  1  0 -1  0

# statement 2
2  6  # 2 rows, 6 columns
0  1  0 -1  0  0
0  0  1  0 -1  0

0  # do not set scattering names manually
```

`CLooG` generates for every statement **for** loops and **if** statements accordingly, with this input file. For more information on `CLooG` input files, please visit there website[1].

Handle

`Handle` is called two times. Once before dependency analysis of `PN`, handling the plane cut and modulo unfolding, and once after, for the merge transformations. When performing a plane cut or modulo unfolding, the PDG must be updated. The node we want to partition is multiplied by the partition factor, which is explained in Section 3.3. Polyhedrons are updated accordingly. Because the actual number of function calls does not increase, but the number of nodes in the PDG do, a mapping needs to be created from the nodes in the PDG to the actual function calls. This mapping is then later on used to print the correct function call to the output *C File*. The node numbering starts from 0 and the function calls starts from 1. For instance, if we have three function calls, Table 1 contains the original mapping. When we apply a transformation like a plane cut with factor 2 on node 1, the resulting mapping table is displayed in Table 2.

Node	Function Call
0	1
1	2
2	3

Table 1: Original Mapping

Node	Function Call
0	1
1	2
2	2
3	3

Table 2: Updated Mapping

For the plane cut, the *prefix* needs to updated as well. A prefix is defined for every function call and tells us about the structure of the program and more specifically, where a function call takes place. For instance, from the prefix of a function call we can derive that a given function call is situated as the second function call in the body of an **if** statement within the body of a **for** loop, and so on. A prefix is a series of numbers where three rules apply when creating one.

1. When a function call is encountered:
   (a) 1 is appended if there is no function call counter
   (b) else, 2 is added to the previous count
2. When a **for** loop is encountered:
   (a) 1 is appended if the preceding number is -1 or if there are no numbers
   (b) -1 is appended
3. When a **if** statement is encountered:
   (a) 1 is appended if the preceding number is -1 (**for** loop indicator) in all prefixes with the same domain

To give an example, prefixes are generated for the running example and the following code, on which a plane cut transformation has been performed. The plane cut transformation will be explained in the next section.

```
void function_call1(int *x, int y);
void function_call2(int *x, int y);
void function_call3(int *x, int y);

int i, j, z;
for(i = 0; i < 10; i++){
    function_call_1(&z, z);      // node 0

    if(i >= 0 && i <= 4)
      function_call_2(&z, z);    // node 1
    if(i >= 5 && i <= 9)
      function_call_2(&z, z);    // node 2

    function_call_3(&z, z)       // node 3
}
```

- Prefix of running example
  - node 0: [1, -1, 1]
  - node 1: [1, -1, 3]
  - node 2: [1, -1, 5]
  - node 3: [1, -1, 7]
- Prefix of partitioned example
  - node 0: [1, -1, 1, 1]
  - node 1: [1, -1, 1, 3, 1]
  - node 2: [1, -1, 1, 5, 1]
  - node 3: [1, -1, 1, 7]

3.3 Process Network Transformations

We now discuss how to perform transformations on a PDG with three different techniques, namely *plane cut*, *modulo unfolding* and *merge*. To decide which technique to use and to which function call to apply it, we must look at its domain. We will use the running example given earlier and apply every implemented transformation on it.
The domain for the first function call is the same as for the second and third function call. The domain is given by $D = \{0 \leq i \leq 9 \ \wedge \ 0 \leq j \leq 9\}$. This is translated into the graph in Figure 4, where the dependencies have not yet been analyzed.

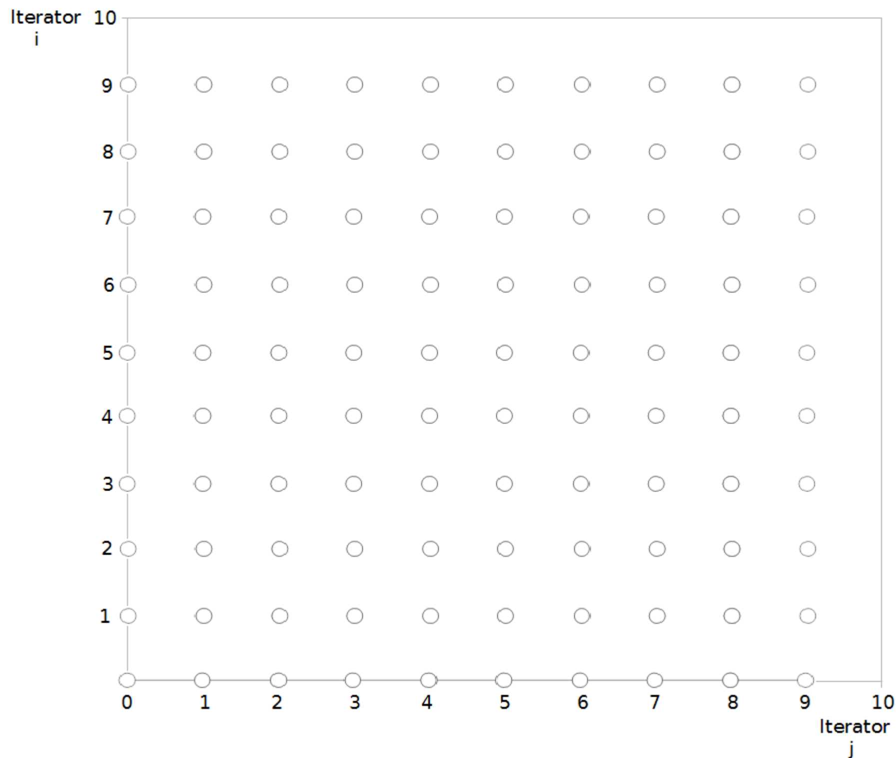

Figure 4: Graph representation of a function call in a double **for** loop, without dependency information

Every point in the graph in Figure 4 represents one iteration of the function call. When assuming there are no dependencies between the function calls, the domain can be partitioned in any way we see fit. Every partitioning created can be executed concurrently. If we need to map the sequential application onto a multi-processor platform which has two processors, we can divide the domain in two equal halves and run them concurrently without communication between the two processors. When we look at the PDG representation in Figure 5 of the given code, we want to create concurrent nodes.

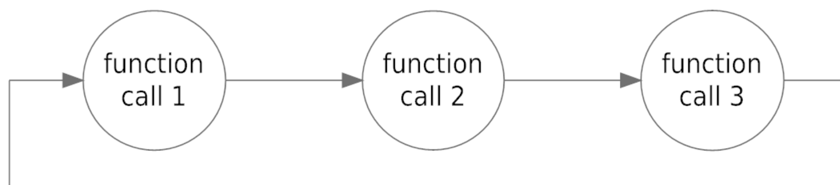

Figure 5: Kahn Process Network of example code

If a program does not contain concurrent elements, we assume it executes entirely on one processor. When having two processors, this is not optimal. One processor does all the work, while the other is idle. If we partition the second function call, we create the PDG in Figure 6, which now contains nodes (function calls) that can be executed concurrently. As a result, one partition can be executed on one processor, while the other partition can be executed on the other processor, optimizing workload distribution.

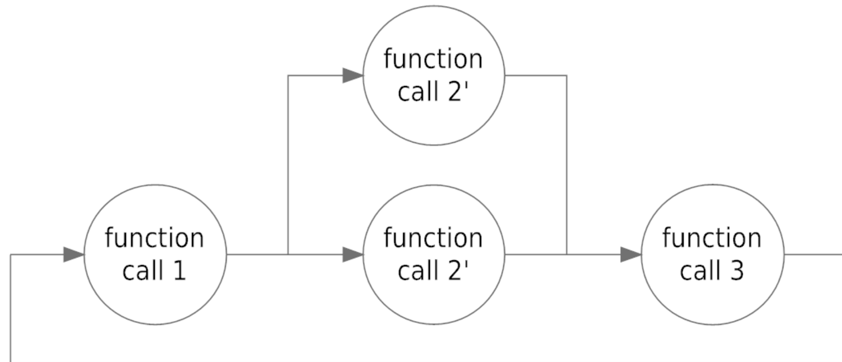

Figure 6: Partitioned Kahn Process Network of example code

When we analyze the dependencies, we have to be smart about partitioning the domain. The goal is to create partitions that are as independent as possible to ensure minimal communication between processors when having mapped the sequential application. Figure 7 holds the same domain *with* dependencies information.

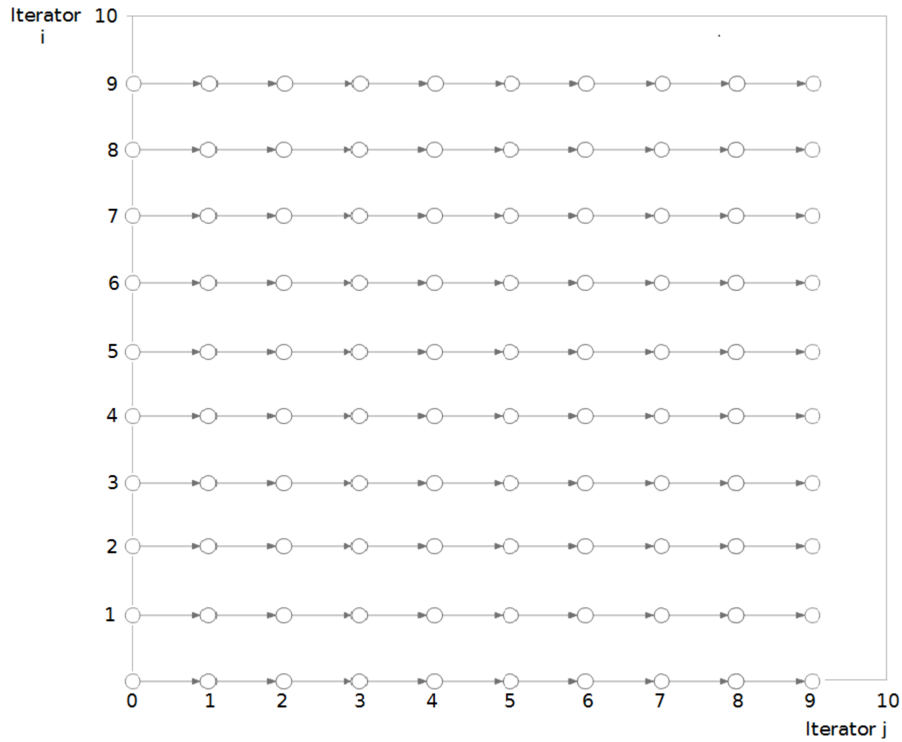

Figure 7: Graph representation of function call in double **for** loop, with dependencies

If we perform a plane cut on iterator j, we cut right through the dependencies (indicated in Figure 8 by cut A), meaning that when we apply this cut, there will be great overhead due to communication between the processors after the mapping. A better solution would be to cut on iterator i (indicated in Figure 8 by cut B), leaving two completely independent partitions of the domain.

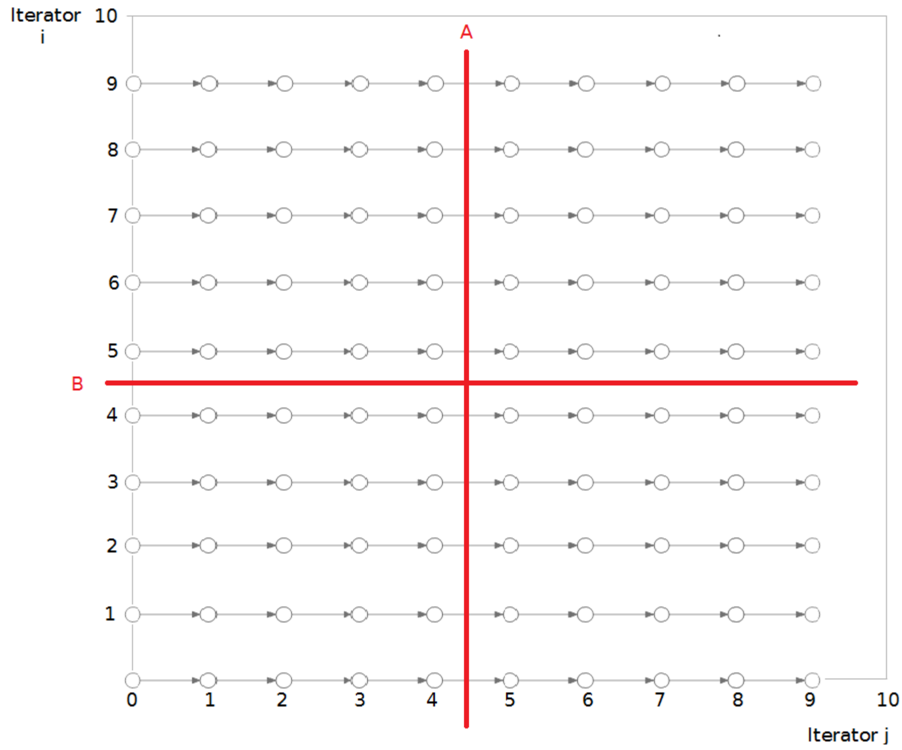

Figure 8: Cut on iterator i and iterator j

3.3.1 Plane Cut

We apply a plane cut on the second function call of the given example code. By applying a transformation to the polyhedron of that function call, the output code we want to obtain is the following:

```
void function_call1(int *x, int y);
void function_call2(int *x, int y);
void function_call3(int *x, int y);

int i, j, z;
for(i = 0; i < 10; i = i + 1){
    for(j = 0; j < 10; j = j + 1){
        function_call1(&z, z);
    if(i >= 0 && i <= 4)
            function_call2(&z, z);
    if(i >= 5 && i <= 9)
            function_call2(&z, z);
        function_call3(&z, z);
    }
}
```

We need to add a new node to the PDG, just like Figure 6. Afterwards, the polyhedrons of the added nodes and the polyhedron of the node we apply the transformation to, need to be changed accordingly. We start of with the polyhedron of the second function call (this is the same polyhedron as the polyhedrons of the other function calls).

$$\begin{pmatrix} \geq/= & i & j & constant \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 9 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 9 \end{pmatrix}$$

The domain of iterator i of the function call is $D = \{0 \leq i \leq 9\}$. `Handle` is implemented in a way that you have to give a factor with which you want to divide the domain. The domain is then divided in equal partitions. When applying a plane cut with factor two on iterator i (for instance, if you have two processors), we have to split its domain in two equal parts. Two polyhedrons have to be created, with the domain D of iterator i given by $D = D_1 \cup D_2$ and $D_1 \cap D_2 = \emptyset$, where $D_1 = \{0 \leq i \leq 4\}$ and $D_2 = \{5 \leq i \leq 9\}$. The

polyhedron with domain D_1 would be:

$$\begin{pmatrix} \geq / = & i & j & constant \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 4 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 9 \end{pmatrix}$$

The polyhedron with domain D_2 would be:

$$\begin{pmatrix} \geq / = & i & j & constant \\ 1 & 1 & 0 & 5 \\ 1 & -1 & 0 & 9 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 9 \end{pmatrix}$$

3.3.2 Modulo Unfolding

Again we use the second function call to partition. A modulo unfolding partitions the PDG, similar to a plane cut. The resulting PDG is given in Figure 6. The resulting code is shown below.

```
void function_call1(int *x, int y);
void function_call2(int *x, int y);
void function_call3(int *x, int y);

int i, j, z;
for(i = 0; i < 10; i = i + 1){
    for(j = 0; j < 10; j = j + 1){
        function_call1(&z, z);
        if(i % 2 == 0)
            function_call2(&z, z);
        if(i % 2 == 1)
            function_call2(&z, z);
        function_call3(&z, z);
    }
}
```

As the name suggests, modulo **if** statements are inserted according to the partition factor, which in our case is two. The polyhedrons that need to be created are not as obvious as the ones for a plane cut. An extra iterator needs to be added, which is used to express the modulo operator. The original polyhedron of the second function call is the following.

$$\begin{pmatrix} \geq / = & i & j & constant \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 9 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 9 \end{pmatrix}$$

When adding another iterator m to the polyhedron, a column needs to added in the right place. The description of m can be found in Section 3.4. Also a row needs to be added specifying the domain of this extra iterator, which includes an indication for the iterator we partition, creating the following polyhedron.

$$\begin{pmatrix} \geq / = & i & j & m & constant \\ 0 & -1 & 0 & 2 & x \\ 1 & 1 & 0 & 0 & y \\ 1 & -1 & 0 & 0 & z \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 9 \end{pmatrix}$$

The value in the column of the first row is the factor with which we partition the domain of the function call. Only the constants x, y and z need adjusting. The constants follow a pattern for each polyhedron representing a modulo **if** statement, which is represented by the following function.

```
int x, y, z;
modulo_polyhedron_constants(int n, int factor){
    x = n - 1;
    y = y - (factor - n);
    z = z - (n - 1);
}
```

Where $factor$ is the partition factor and n represents the nth polyhedron

that we print, with $\{1 \leq n \leq factor\}$. The resulting two polyhedrons will therefore be the following.

$$\begin{pmatrix} \geq / = & i & j & m & constant \\ 0 & -1 & 0 & 2 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 8 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 9 \end{pmatrix}$$

$$\begin{pmatrix} \geq / = & i & j & m & constant \\ 0 & -1 & 0 & 2 & 1 \\ 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 9 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 9 \end{pmatrix}$$

We work this out, to see if it is correct.

$$\begin{array}{cccccl} \geq / = & i & j & m & constant & \\ 0 & -1 & 0 & 2 & 0 & \rightarrow i = 2m \\ 1 & 1 & 0 & 0 & 0 & \rightarrow i \geq 0 \\ 1 & -1 & 0 & 0 & 8 & \rightarrow i \leq 8 \\ 1 & 0 & 1 & 0 & 0 & \rightarrow j \geq 0 \\ 1 & 0 & -1 & 0 & 9 & \rightarrow j \leq 9 \end{array}$$

$$\begin{array}{cccccl} \geq / = & i & j & m & constant & \\ 0 & -1 & 0 & 2 & 1 & \rightarrow i = 2m+1 \\ 1 & 1 & 0 & 0 & -1 & \rightarrow i \geq 1 \\ 1 & -1 & 0 & 0 & 9 & \rightarrow i \leq 9 \\ 1 & 0 & 1 & 0 & 0 & \rightarrow j \geq 0 \\ 1 & 0 & -1 & 0 & 9 & \rightarrow j \leq 9 \end{array}$$

The domain of iterator i of the first polyhedron consists of all even iterations of the original domain. Also, the domain of iterator i of the second polyhedron consists of all uneven iterations.

3.3.3 Merge

To merge function calls, one has to take the dependencies into account. When we look at the Figure 5 we can see that the function calls are executed in a predefined order due to dependencies. We cannot for instance execute *function_call3()* before *function_call2()*, as the semantics of the application will change. The merge transformations that can be preformed are denoted by the set $S = \{\{1,2\},\{2,3\},\{1,2,3\}\}$, where the numbers represent the function calls, respectively. When two or more nodes are merged, only one scattering function is needed to define the order of execution. The scattering function of the first function call is used as scattering function for the merged function calls.
Merge has to be handled differently than a plane cut or modulo unfolding. The dependencies need to be known. While in the other cases, they need not. When performing a merge transformation we do not make changes to the PDG but change what we write to the *CLooG File*. For this to work, a mapping is needed from the statements that are printed to the *CLooG File* to the nodes in the PDG. We start again with the original mapping (Table 3). When we apply the merge transformation to the second and third function call, we can see in Table 4 that statement two has two domains, which we covered earlier.

Statement	Node
1	0
2	1
3	2

Table 3: Original Mapping

Statement	Node
1	0
2	1,2

Table 4: Updated Mapping

The transformation is also passed to the mapping for the function calls. When function calls are merged, a new function is created. This new function will obviously be declared before the **main** function. The function names and arguments used by both functions are combined to do so. Within this newly defined function, the function calls that are merged are called upon in the order they occurred in the input *C File*. All function arguments are enumerated, followed by a unique number. Suppose we take the following two function calls:

```
function_call2(&x, x);
function_call3(&x, y);
```

When we merge these function calls and fill in the function call arguments, we can see that the previous function calls do no equal the following function calls.

```
void function_call2_function_call3(int *x_1, int x_2,
                                   int *x_3, int y_4){
    function_call2(&x_1, x_2);
    function_call3(&x_3, y_4);
}
```

Whenever we encounter a function call argument which equals another, we use the first definition of that argument. When outputs are involved, some additional statements need to be added, i.e. the value of the output arguments also need to be equal to the first definition of that argument. This creates the following function, which equals the function calls we began with.

```
void function_call2_function_call3(int *x_1, int x_2,
                                   int *x_3, int y_4){
    function_call2(&x_1, x_1);
    function_call3(&x_1, y_4);
    *x_3 = *x_1;
}
```

This approach also works with array arguments.

3.4 Assumptions

All assumptions that have been made during the implementation process have been listed below.

- input *C File*
  - There is only one statement per line
  - Each **for** loop has an integer iterator and only one iterator
  - All functions are declared **void**
  - All functions are declared within the same file as the **main** function
  - All function calls are located in the scope of a **for** loop
  - All variables used are accessible throughout the scope of the main function
  - Function arguments are of the following format:

    ```
    [const] type [*] variable_name
    ```

    (All items between brackets are optional)

- The opening bracket of a **for** loop is located on the same line and any closing bracket is the only character of its line

- *Handle*
  - The size of the domain of the iterator we want to apply the transformation to, must must be divisible by the partition factor (*When trying to manually apply the transformations, the output of the resulting program is incorrect when this assumption is violated*)

- *Partition File*
  - All line numbers obey a natural order
  - All transformations are disjoint, i.e. only one transformation can be applied per function call.

3.5 Unimplemented

Due to time constraints the transformation process is not performed automatically. A manual component is still present. This was discussed in Section 3.1. Another thing that is not implemented is the use of parameters. Each transformation uses constants to calculate the resulting polyhedra, which can not be done when using parameters, because they are variable. The domain of an iterator needs to be defined by these variables. Some calculations are needed for this. We look at how polyhedrons turn out when using parameters and apply a plane cut transformation. We will use the following code.

```
#pragma parameter M 10 100
#pragma parameter N 10 100
void function_call();

int i;
for(i = M + 10; i <= N - 10; i = i + 1){
    function_call();
}
```

The polyhedron for the function call is given by:

$$\begin{pmatrix} \geq / = & i & M & N & constant \\ 1 & 1 & -1 & 0 & -10 \\ 1 & -1 & 0 & 1 & -10 \end{pmatrix}$$

The general formulas for a plane cut are

$$\begin{aligned} lowerbound &= lowerbound + \alpha \frac{upperbound \; - \; lowerbound}{\beta} \\ upperbound &= (\; lowerbound + (\alpha + 1) \; \frac{upperbound \; - \; lowerbound}{\beta} \;) - 1 \end{aligned}$$

Where α is the number of the partitioned polyhedron and $\{0 \leq \alpha \leq factor - 1\}$, and β is the partition factor. We now need to expand this formula with the parameters and their offsets. We will make the offset variable as well, to be able to create a more general formula. The offset of M is M_0 and the offset of N is N_0.

$$\begin{aligned} lowerbound &= M \; + \; M_0 \\ upperbound &= N \; + \; N_0 \end{aligned}$$

We calculate the lower bound line of the polyhedron to be created. The upper bound line should be trivial after this. We start by filling in the lower and upper bound defined above, into the formula for lower bound.

$$lowerbound \;\; = \;\; (M \; + \; M_0) + \alpha \frac{(N \; + \; N_0) \; - \; (M \; + \; M_0)}{\beta}$$

We now rewrite this into a format of which we can easily create a polyhedron.

$$
\begin{array}{rcl}
i & \geq & (M + M_0) + \alpha\frac{(N + N_0) - (M + M_0)}{\beta} \\
\equiv \quad 0 & \geq & -i + (M + M_0) + \alpha\frac{(N + N_0) - (M + M_0)}{\beta} \\
\equiv \quad 0 & \geq & -\beta i + \beta(M + M_0) + \alpha((N + N_0) - (M + M_0)) \\
\equiv \quad 0 & \geq & -\beta i + \beta(M + M_0) + \alpha(N + N_0) - \alpha(M + M_0) \\
\equiv \quad 0 & \geq & -\beta i + \beta M + \beta M_0 + \alpha(N + N_0) - \alpha(M + M_0) \\
\equiv \quad 0 & \geq & -\beta i + \beta M + \beta M_0 + \alpha(N + N_0) - \alpha(M + M_0) \\
\equiv \quad 0 & \geq & -\beta i + \beta M + \beta M_0 + \alpha N + \alpha N_0 - \alpha M - \alpha M_0 \\
\equiv \quad 0 & \geq & -\beta i + (\beta - \alpha)M + \alpha N + (\beta - \alpha)M_0 + \alpha N_0
\end{array}
$$

This can directly be translated into the following polyhedron row, which represents the lower bound for every partitioned polyhedron. The upper bound row is analog to this row.

$$
\begin{pmatrix}
\geq / = & i & M & N & \mathit{constant} \\
1 & -\beta & (\beta - \alpha) & \alpha & ((\beta - \alpha)M_0 + \alpha N_0)
\end{pmatrix}
$$

3.6 Implementation Problems

The `PN` source code is not commented, which as a result, took quite some time to understand. Some undocumented problems occurred when working with it as well. There is a problem with giving an application to `PN` when it contains a modulo **if** statement. The scattering functions that are generated are incompatible with the official release of `CLooG`.
Another problem occurred when checking the dependencies `PN` has analyzed before performing a merge transformation. There are two pointers that point to the function call nodes that are dependent on each other. When a self dependency occurs, one of these pointers points to NULL, resulting in a segmentation fault when traversing the list of dependencies. Also, an oddity were the prefixes of the nodes. When performing a manual plane cut, the prefixes changed as expected, due to the insertion of **if** statements. When performing a manual modulo unfolding however, the prefixes did not change. `CLooG` also produced some faults, when the library they provide was used. This was bypassed by writing the *CLooG File*.

3.7 Usage

The partitioning process can be invoked by calling PN:

```
./c2pdg FILE.c
./pn -i FILE.yaml -o FILE_pn.yaml
```

Where FILE is the name of the input file. Input and output files of Handle are:

```
FILE.part     ( input : file containing manually written
                        transformations                  )
FILE_out.c    ( output: generated C file                 )
FILE.cloog    ( output: file meant for CLooG. Contains
                        function call polyhedrons and
                        scattering functions             )
```

3.8 Partition File

The manually written *Partition File* will be parsed by Handle. Therefore, the file has a predefined syntax beginning with the number of transformations. No comments in this file are tolerated, though to explain the file, all character on the right hand side of the # symbol are comments. The syntax for plane cutting and modulo unfolding is given by the following.

```
[plane | modulo]
[line number of function call]
[partition factor]
[iterator of transformation]
```

The line numbers of the function calls in our running example are 8, 9 and 10. The line numbering of a file starts from 1. To indicate on which iterator to apply the transformation on, we start counting from 1, which is a representation of the iterator of the outer most **for** loop. For every following **for** loop, the count goes up by 1. The file containing the plane cut transformation of the second function call on the first iterator would be the following file.

```
1      # 1 transformation
plane  # transformation type
9      # line of function call
2      # partition factor
1      # iterator of transformation
```

A merge is different, because it does not need a partition factor or iterator number to preform a transformation. The syntax is denoted by the following.

```
[merge]
[line number 1] [..] [line number n]
```

Where n is the nth node to merge. The line numbers are written on one row, separated by a space character. If we were to merge the first and second function call, the *Partition File* would look like the following.

```
1      # 1 transformation
merge  # transformation type
8 9    # lines of function calls
```

Multiple transformations are also supported. A merge of the first two function calls and a modulo unfolding on the third function call would look like the following.

```
2      # 2 partitions
merge  # partition type
8 9    # lines of function calls
plane  # partition type
10     # line of function call
2      # partition factor
1      # iterator of transformation
```

There is a limitation to what can be passed to `Handle`. The line numbers used in the *Partition File* must obey a natural order and all transformations must be disjoint, meaning no more than one transformation can be performed on each function call.

4 Experiments

Some experiments have been done with an application called `sobel`. `Sobel` is an image edge detecting application. It computes an approximation of the gradient of the image intensity function.
The experiments have been done on a four core machine (the *SILVER* server at the LIACS faculty of Leiden University). Results are shown in Table 5. The `Time` columns contain the results of the application called `Time`, which in our case has as argument the `PN` application. Table 5 also includes the number of rows of the code, which refer to the number of rows of the output file within the **main** function. As a reference, the input program contains 29 rows, not including variable declarations nor empty rows. Regarding the transformations, one transformation has been done at a time with different partition factors (indicated by the number on the right hand side of the transformation type).

Transformation	Time	Lines
None	0:00.56	14
Plane (2)	0:00.71	24
Plane (4)	0:00.82	36
Plane (10)	0:02.85	47
Plane (50)	0:09.27	166
Modulo (2)	0:00.77	26
Modulo (4)	0:00.96	36
Modulo (10)	0:02.29	66
Modulo (50)	0:24.33	266
Merge (2)	0:05.09	16 + 4
Merge (3)	0:06.36	15 + 5

Table 5: Experiment results of the usage of the *PN* application

We see that the more partitions we create, the more time it takes. This is trivial. Regarding the plane cut and modulo unfolding, the more partitions we need to create, the more nodes need to be added to the PDG and polyhedrons need to be generated. There is however a difference between these cuts: the time. This can be explained by two things. The first being the calculation for the modulo output polyhedron. It takes more calculations than the polyhedron created for a plane cut. Second, `CLooG` generates more lines for a modulo unfolding than it does for a plane cut. Writing output causes huge delay, hence the modulo unfolding takes more time.
In comparison, merge takes longer than a low partition factor of a plane cut or modulo unfolding. This is due to dependency evaluations. The graph representing the dependencies needs to be fully analyzed for every two nodes to be merged. There is a logic when we look at the line numbers of the merge transformation. When two function calls merge, it will become one function call. This newly created function call needs to be declared, taking up the row in which the actual function is declared, followed by the function calls that have been merged, in its body.

5 Conclusion

We have discussed a compile-time mapping technique using Kahn Process Networks. The techniques used in this article are a step forward for the parallelisation of sequential applications. It is up to the user which transformation to use, to what degree to use it and to which instruction. By using evaluation metrics, this process can be automated.

References

[1] Defining a scanning order: Scattering functions, August 2010. `http://bastoul.net/cloog/manual.php`.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] Gilles Kahn. The semantic of a simple language for parallel programmming. 1974.

[4] Hristo Nikolov Sven Verdoolaege and Todor Stefanov. pn: A tool for improved derivation of process networks. 2007.