Programming a Many-Core platform: experiences with the Kalray MPPA

Sebastian Bacanu, Kamil Deja, Max Thonagel

A thesis submitted in partial fulfillment for the
R&D project
IST Semester

INSA Lyon

Supervisor: Assoc. Prof. Lionel Morel (CITI, INSA Lyon)

January 2015

Abstract

Modern processors now comprise from a few to several hundreds cores on the same chip. This multi/many core trend that will continue to go in the forseable future, as chip manufacturers rely on it to continue the historic increase in overall computer performance. Yet, no lingua universalis for programming them has been invented. It is actually probable that the future of programmers will be paved with a variety of different languages relying on different programming models, that will aleviate the parallel nature of processors in different ways. During this project, we have worked with a particular many-core processor, the MPPA from Kalray. We have studied different techniques to program it, using different programming models. The present document reports our experiments.

Contents

Chapter 1

Introduction

These days chip manufacturers advertise new products with computing power ever greater than previous generations, trying to prove right the famous law formulated by Gordon E. Moore in his 1965 paper[**?**]. This has been the case for decades, but in 2005 the trend of constantly increasing clock speed could no longer be kept up. Due to heat dissipation difficulties, manufactures decided to instead create chips with multiple cores without increasing clock rates.

Processors are thus potentially more powerful, and running multiple processes *concurrently* became much more efficient. However, not every program is able to make use of all the available resources. There is an important distinction between so-called *sequential* and *parallel* programs. There are tasks which can addressed in parallel—giving two distinct parts of the work to two different processors so they can be executed at the same time. There are some tasks that can only be executed as a whole and cannot be parallelised.

Almost every program consists at least partly of sequential code or code that would not be worth running in parallel. For problems that *are* parallelisable, programmers use different models to achieve better performance. The parallelism is realised on the hardware by splitting the program into separate workers which represent independent tasks when they are executed concurrently. This can be done by the programmer in either a more traditional imperative programming model, or by using a dataflow programming model.

In imperative programming languages such as C, there are a variety of ways to achieve parallelism. The most common is by splitting work up into threads and using a shared pool of memory to communicate and contribute data, but this creates issues with managing the integrity of this shared memory[**?**], as well as scaling the number of threads with the number of processors. There are alternative methods of communication available to solve the former problem, like the Message Passing Interface library which provides developers with an API for passing messages between workers so that programmers don't need to worry about manually synchronising data.

The dataflow programming model focuses on identifying unique tasks and encapsulating them inside of *actors*, sometimes also called *agents*. Actors are isolated from one another, communicating only via FIFO queues of data that they pass between one another. This programming model is very well suited to parallelisation, where multiple instances of each actor can being easily run to saturate the number of processors available.

We had available to us a development workstation with a Kalray MPPA 256-core processor on which

we experimented with parallel programming. Our goals were to 1) explore what programming using pthreads is like on this platform, and 2) try out the open-source Orcc and the Kalray-made ΣC dataflow languages on the MPPA.

The report is organized as follows. Chapter 2 will present background knowledge necessary for understanding the rest of the report. Chapter 3 will introduce the MPPA processor, on which our experiments have been done. We will present the architecture of the chip first along with available programming solutions, based either on a threading model or on a dataflow approach. Chapter 4 contains the core of our work during this project. It describes our example programs, how we wrote them in either C and thread or dataflow and reports of our experiment with executing these programs on the MPPA. Finally, section 5 concludes.

Chapter 2

Background

2.1 Multi/Many-Core Processors

Since 2005 the number of devices equipped with Processors called Multi-Cores has been increasing, with chip manufacturer like Intel and AMD releasing more and more into the market [**?**] [**?**] as a way to compensate for the impossibility to further increase processors' frequency. However, the development of these processors started much earlier. As an example we could cite the R65C29 released by Rockwell International in 1984 [**?**]. It consisted of two 6502 8-bit CPUs cadenced at 4 MHz and had an on-chip shared RAM of 128 bytes.

Later in the beginning of the 2000s big chip manufacturers like Intel and AMD started to concentrate on architectures with more than one complete computation unit on the same chip. From then on Moore's Law seemed to be satisfiable, because the former problems that came with increasing the clock rate like unbearable power consumption and heat seemed to be disappearing. Just by adding one or more extra cores, the potential performance of the chip (at least the maximum number of instructions it could perform per second) was increased.

A multi-core-architecture has three main advantages. At first, the processor time can be used more efficiently, because several active programs do not have to be switch by the scheduler so often. Secondly, programs can gain a speedup when programs internally have parallel tasks. This is the incentive of Many-Core-architectures. At last, multi-cores can offer a better energy management, because unused cores can be disabled on-demand.

In the panorama of multi-core processors, we can make an important distinction between homogeneous and heterogeneous processors.

An example of an heterogeneous architecture is the architecture familly developed by ARM since 2007, and know as "big.LITTLE". "big.LITTLE" stands for a chip design with usually a package of a number of small and low-power cores alongside higher performance cores. Depending on the architecture the chip can only switch between performance and energy saving cores or he can command them independent. Today the latest models comprise 4 big cores alongside 4 little ones, and all 8 cores can be activated simultaneously if need be. This approach is used a lot in the mobile sector.

For personal computers and multipurpose performance-oriented systems like workstations, homogeneous architectures are more common. Reasons are easier and cheaper layout of the chip in the case of personal computers. For workstations it is simpler to develop performance tools.

The distinction between multi- and many-core processors is not well defined. Some see the many-core as an evolution of multiprocessor systems[1], where 2 or 4 processors were put on one mainboard, and the new generation puts these onto one chip. Others make the difference simply in number of cores, not regarding a special form of architecture. For Shekhar Borkar from Intel, the difference lies in the way the architecture gains its computing power [**?**]. He points out that single cores within many-core chips are very weak compared to their multi-core equivalent (typically smaller frequencies to accommodate lower power). But the low consumption of these small cores allows high scalability which leads to higher overall-performance.

Up to now, many-cores—defined as processors with a meshed network as interconnection—are only meant for special purposes like signal and image processing, cloud computing, and cyber-security. For home users there are no ready-to-use solution on the market. A first reason is of course the experimental aspect of this kind of hardware. A second reason is that today's operating systems are not designed to take advantage of these many-core chips. Dedicated OSs are still experimental and research-focused (see for example Corey [**?**] or Barrelfish[**?**]).

Examples for Many-Cores are the Kalray MPPA 256 with 256 Cores which will be presented in more details in chapter 3, TILERA's TILE-Gx family[2] with up to 72 cores and Intel's Xeon Phi Coprocessor 5110P[3] with 60 cores, but versions with more cores are planned.

2.2 Programming Models for Parallel Architectures

In order to exploit multi/many-core architectures, software needs to be parallelised. There are two approaches for this. The first is for programmers to manually parallelise their software. Of course, this presents many challenges and can be extremely time-consuming. The second approach is autoparallelisation, where compilers and runtimes can figure out how to parallelise given portions of the code on their own.

While autoparallelisation can be used to great effect on simpler tasks, the sheer complexity involved in parallelising sequentially-executed software means that this approach has its limits. With the manual approach however, software is ultimately specifically developed to exploit parallelism as much as possible and therefore perform better [**?**].

Parallel models can be categorised as using either shared memory or distributed memory. With the former, a common memory space is shared between different processes and they all have complete access to it, therefore making each process able to communicate with the others by manipulating memory. In contrast to this, distributed memory models give each process its own private address space, and message passing is instead the communication method used.

What follows is an overview of a select group of commonly used programming models that employ a variety of combinations of the approaches described above.

[1] `http://goparallel.sourceforge.net/ask-james-reinders-multicore-vs-manycore/`, as of January the 21th,2015.

[2] `http://www.tilera.com/products/processors/TILE-Gx_Family`, as of January the 20th,2015.

[3] `http://ark.intel.com/de/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core`, as of January the 17th,2015.

2.2.1 POSIX Threads

Threads are a very flexible approach to parallelising software. POSIX threads [**?**] are implemented as a library, and any POSIX-compliant operating system can make use of them. Threads are lightweight processes with their own execution stacks that access a shared pool of memory to interact with one another.

Because of their use of shared memory, programming with this model can be problematic. Developers need to be aware of race conditions and deadlocks and they must be able to identify and deal with critical sections in their code. The pthreads library provides utilities for maintaining locks such as mutexes and semaphores.

Pthreads are not made for the specific use of parallelising software, but rather as a means of achieving concurrent programming. This leads to programs that are unstructured and difficult to scale with the number of processors available. These factors, including the memory complications, make pthreads a model that is difficult and problematic to implement.

2.2.2 Message Passing Interface

One way of getting around this shared memory mutual exclusion problem is to use another style of inter-process communication. The Message Passing Interface [**?**] is a library that provides developers with an API to facilitate all of this while taking care of the synchronisation all by itself. It can be added into, for example, programs which use pthreads as a method of parallelisation, helping to alleviate one of the issues mentioned above.

Unlike the shared-memory approaches, message-passing models (which MPI is almost synonymous with [**?**]) involve each parallel process owning its own memory and instead communicating with one another by purposely sending messages between themselves.

Messages are passed by copying sections of memory from one process to another. This can only occur when both the sender and the intended receiver declare that they want to send/receive a message, leaving the library to deal with the synchronisation required in order to enforce mutual exclusion. There is also a method for achieving one-sided communication where the sender and receiver don't need to first be matched.

2.2.3 OpenMP

OpenMP allows developers to automate the parellelisation of software. Blocks of code are defined as being "PARALLEL" with pragmas, and then the OpenMP compiler and runtime are able to figure out how to achieve this using threads and shared memory without developers needing to implement all of it themselves.

This style of parallelisation excels at effortlessly splitting up tasks such as iterating over the index of a loop across multiple processors. It can even handle parallelising dynamically-generated work such as while loops, meaning it can be used with graph algorithms and dynamic data structures.

The OpenMP pragmas are simply annotations that go alongside code for existing programming languages, such as C. If run through a traditional compiler, OpenMP programs would still compile as regular sequential programs as the compiler would simply ignore the pragmas.

2.2.4 Dataflow Programming

Dataflow programming is one of the primary models we investigated, and as such it has an entire section dedicated to it. To give a brief overview in the current context, Dataflow programming splits up its computations into ”actors” which communicate with one another by passing messages through FIFO queues. Once again, this model fits the theme of parallel computing very well, as 1) multiple instances of each actor can be run in an effort to do more at once and 2) all actors in a graph can be considered as independent tasks and can thus be safely executed in parallel.

Unlike the previous examples which were either libraries that were used with existing programming languages or compiler annotations added on to existing languages, Dataflow implementations (of which there are several) involve their own languages as well as specific compilers and, oftentimes runtimes.

2.3 Dataflow

Dataflow programming is an old, but underestimated programming paradigm. The present need for multi-core programming can be a great opportunity for the (re-)development of this idea. Indeed, using Dataflow models for parallel programming brings a lot of benefits.

The concept of dataflow started in the 60s, but the first contributions worth mentioning are dated 1974, when Gilles Kahn [**?**], and Jack B. Dennis [**?**] proposed a new dataflow programming languages. Kahn suggested more formal and mathematical solutions with schemas which we are still being used nowadays, even if they have changed slightly, especially from a syntax point of view. On the other hand Dennis proposed a language considering the semantics of Dataflow, the structure of actors and tokens which can be send among them. He also described the parallelism issue in dataflow programming.

The first big advantage is connected to the structure of the dataflow-based programs. In general, we can describe them as a set of nodes, also called actors or agents, connected to each other 2.1. All of the computation is run within each node and communication between them is solely a consequence of decisions taken by actors themselves.

This solution provides full separation between all actors. Hence, it creates the possibility for assigning, for example, one actor to each processor core, since all of them are asynchronous, and their working is based on the availability of data on their incoming communication channels.

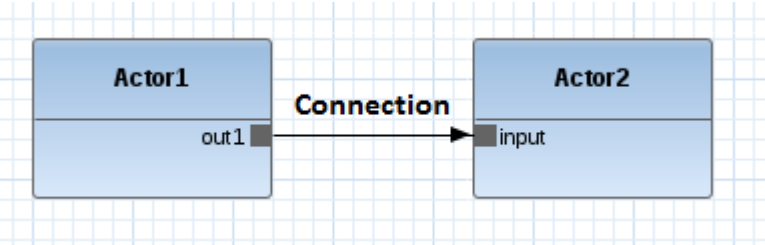

Figure 2.1: The most simple dataflow program with two nodes/actors, and one connection between them, which enables data to flow from Actor1 to Actor2

Considering dataflow programming, we should notice the distinction between static and dynamic manners.

The main difference between these two models can be found in the connections (channels) between the actors. Basically static dataflow [**?**] requires usage of fixed size links between every two actors. Dynamic

dataflow programming [**?**] allows to send/read as much data as we want. This difference results in each method having its own advantages and disadvantages, which will be described later.

In more accessible way, we can describe dataflow programming as boxes that are connected by arrows. We can imagine that those boxes (nodes) are workers, units that can work independently from the others. The only context between those workers is the data stream, represented by the arrow. This arrow means that there is no data exchange between two workers, except a one-way directed transport, like a conveyor in a factory. The difference between static and dynamic model can be described as a difference in this conveyor belt.

When considering static dataflow we can imagine that one worker is passing to the other a fixed size package of data, which has, for instance, always the same weight. However, if the workers work in the "dynamic dataflow company", they don't have to follow this restriction. They are more free, and can pass as much data as they want to the second worker, on a rhythm depending on their capacity to produce new packages. If it's more that the next worker needs, it should be stored on a specially prepared queue.

The easiest example of a dataflow parallel program is connected to graphics. We will consider a dataflow program whose intent is to process a flow of images. Let's assume that the 4-colored box on the left part of figures 2.2 and 2.3 is the image we're trying to work on. This data is really easy to split—we can just cut the image into, for example, four pieces, and then work on each part in parallel. Here we've got to take into account which model of Dataflow programming we should use.

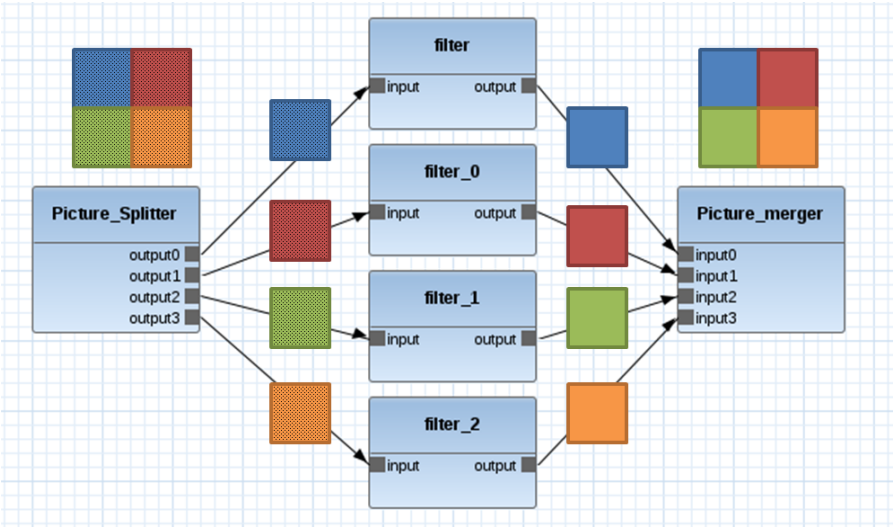

Figure 2.2: Example of Static Dataflow program schema. On the left side we can see the image with noise, which is splited to four pieces. Righ part of the image shows the results after filtering. It's essential that the data size didn't change, and should be equal in any case.

According to the type of work we'll do in the parallel nodes, we should use either static or dynamic dataflow model.

2.3.1 Static Dataflow

Let's start with the example where we want to remove some simple uniformly distributed noise from the bitmap pictures, without changing their resolution (see figure 2.2). In this case the size of the filter output is always the same—$1/4$ the size of the original picture—so we can very easily employ static dataflow

programming and learn about its advantages. First, we don't have to bother buffering the data method (and its potential overflow). Since the size of the transferred tokens is always equal, the receiver (printer) can be prepared to obtain and work with them. Second, a compiler for static dataflow benefits from this information and produces very efficient code. Generally writing static dataflow programs is much easier. However, this model is not always sufficient when one wants to program real-life applications.

2.3.2 Dynamic Dataflow

For instance let's now consider that we don't want to filter the images, but compress them using the simplified JPEG method. Normally we would use blocks of 8×8 pixels on which we'd run some computations. We can consider a really easy example where we want to compress the fixed size images as 16×16 pixels, so we can run all the blocks computations in parallel using a schema which is really similar to the previous one in figure 2.2.

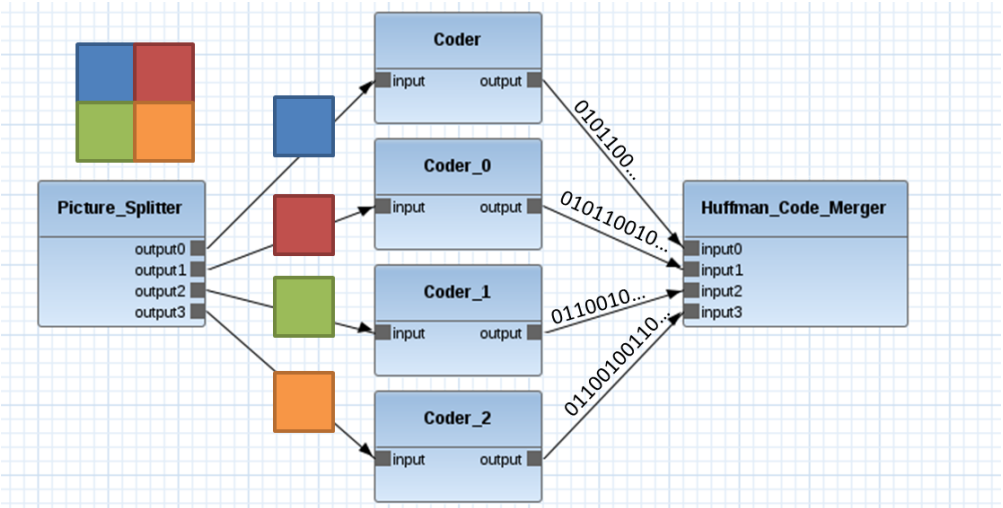

Figure 2.3: Example of Dynamic Dataflow program schema. We can see that on the left side we'll always get the fixed size piece of data, which is $1/4$ part of image. However, on the right side the output is the Huffman code, with unpredictable length

I don't want to dwell here on the JPEG compression method. I'd rather briefly say that what's going inside the nodes is just preparing and coding the content of the block using Huffman coding, which is the most significant characteristic of this application for us. The Huffman coding method tries to encode the given set of data (in our case block of pixels) by assigning to each pixel one binary number, with length proportional to its probability density function index. It means that the more frequent the element is, the shorter number it will have. It's now brightly seen that the final length of the code is not predictable.

According to this we cannot, as you might already suspect, use the model from the previous example. This time we should choose a dynamic dataflow programming language. Thanks to this solution we'd be able to send the original length of the Huffman coded image block, which is a great benefit. However it also has a few drawbacks. At first we've got to take care of the size of the packets, which the receiving actor will have to deal with. We can imagine a situation when there are a few images we want to encode one after another, and the computing nodes sends a continuous stream of data. It can be that, by coincidence, we'll read the content of the next image.

In this example we didn't get to see one of the worst behaviours of dataflow dynamic programming. Let's assume that instead of compressing we want to decompress the image (from example from one JPEG compression rate to another). In this case we'd also not know what the size of the output data sent by the nodes in the middle will be. This time we wouldn't have the upper bound (or at least it'd be really large). We can imagine that with huge resolution images it can be a significant value, which can have a great impact on the memory we'd have to provide, or otherwise we can easily end up with a memory overflow situation.

However, Dataflow programming [**?**] is one of the easiest method to create parallel multi-core programs. What's more, visual dataflow programming languages, such as the Orcc RVC-Cal tool described below, ease this process even more.

2.3.3 RVC-Cal

Orcc [4] is an Integrated Development Environment dedicated to dataflow programming. It is built on top of the Eclipse platform. Since we already introduced this distinction, it's good to mention that Orcc RVC-Cal[**?**] is a dynamic dataflow language, with all the benefits and drawbacks included. What's more, it also provides a graphic tool to create dataflow graphs with actors and connections.

The Graphic IDE for Orcc RVC-Cal is really convenient. It supports all the techniques we're used to. We can simply connect actors with just one mouse move, or create a new node with chosen actor code via simple drag and drop. It also allows us to define the size of the FIFOs which are responsible for the connections between nodes. However the graph itself is created in the special .xdf file. If we open it, we can notice that the code within is basically an XML dialect.

Now, let's move to inside of the actors. Each of them can have a number of outputs and inputs which connect them to others, as well as plenty of functions (actions) which can be written using the RVC-Cal language.

Example program We will use the example in figure 2.1 to demonstrate the syntax used in RVC-Cal to describe the internal behavior of an actor. We will also explain what tools are provided.

At first each actor should be placed somewhere within the project hierarchy (line 1), probably in the package which contains all the project components: actors code in .cal files, data and network schema in .xdf files. We can import data to the actor using import keyword, as can be seen in lines 2 and 3 of figure 2.4.

The next line of the code is the header of the actor. Here we specify its name, which should be the same as the name of the .cal file, and all the inputs and outputs we'll need. Orcc requires us to specify the types of streams on which the actors will receive or transmit data. However, as a dynamic dataflow language, it doesn't obligate us to establish the number of tokens read (resp. written) on each input (resp. output) port.

After the actor's header we can start to write its body, which can look a bit like any imperative program. For example we can start with the declaration of global variables (lines 7 and 8) or functions, which in this case are called actions (lines 11 and later 22).

[4] `http://orcc.sourceforge.net/`

```
1  package org.ietr.addorcc;           // this is the package name
2  import org.ietr.addorcc.Data.SRC1;  // import the value of "SRC1"
3  import org.ietr.addorcc.Data.max;   // that is in Data.cal to this actor
4
5  // This line declare the input/output ports of this actor
6  actor Actor_name () int(size=8) Input==> int(size=8) source1, int source2:
7      int i := 0;
8      int f := 0;
9
10     // first action
11     init: action ==> source1: [Out1]
12     guard
13                 f < 1
14         var
15         uint(size=8) Out1
16     do
17                 Out1 := 1;
18                  f := f+1;
19     end
20
21     // second action
22     run: action Input [In] ==> source2: [Out2]
23         guard
24                 i < 5
25         var
26          uint(size=8) Out2,
27          do
28     foreach int j in 0 .. In
29         do
30                         Out2 := SRC1[i];
31                         i := i+1;
32                 end
33     end
34 end
```

Figure 2.4: Simple example of RVC-Cal code

An action's header really recalls that of an actor. We can specify the name of an action, and we are obligated to declare which inputs and outputs it will need. Also notice that we don't have to use all of the actor's inputs/output for a particular action.

Line 12 in our code is the guard of action `init`, which is a useful tool. Basically if the conditions in the *guard* are not satisfied, the action won't be fired. It allows us to control the input, or some external conditions, like for example size of included data or global variables.

Since we're up to use the output from within the `init` action, we have to declare the local variable which will be consider as this output, Out1 in this case (see line 11 and 17). Then we finally can write the body of our function. RVC-Cal supports all of the useful programing tools like for/while loops, if conditions, arrays and so on. We can see in the run action the use of a simple `for` loop. It is very interesting, because it shows us the benefit of dynamic dataflow programming language. In this example, we can relate the number of forwarded tokens to the value of received input. It would not be possible in a static dataflow model.

The last thing to consider, is how our actions will be scheduled. To manage it, We can use some included support. For example, We can simply prioritise some actions, or even indicate the whole order of them. Since we didn't specify it, they will be fired in the non-deterministic manner. Speaking precisely, there is an infinite loop in each actor, which all the times tries to find the action which is next in the schedule and according to the guards, ready to fire.

Chapter 3

MPPA - Architecture and Programming Models

3.1 The MPPA

The MPPA 256 designed and commercialized by Kalray[1] is a massively parallel processor array (MPPA) or Multi-Purpose Processing Array technology[2]. The processor and everything related with it is often abbreviated with the term *K1*. Its overall architecture is described on figure 3.2. It consists of 16 clusters with 16 cores each and 4 IO subsystems around them. Those clusters are connected to each other and to the IO subsystems via a network on a chip (NoC), which is divided into a data network, *D-NoC* and a control network, *C-NoC*. It allows for high bandwidth and different ways of communication. This communication is managed by routers, that are on every cluster. Moreover all cores on one cluster have a shared memory and their own address space in the DDR RAM, which is accessed by the NoC. Figure 3.1 gives an overview of a cluster architecture.

3.2 MPPA development environment

Around the MPPA is a host system, on which the MPPA is mounted via PCIe. The host system is powered by a normal x86_64 processor, which comes with the ACCESSCORE development tool for the MPPA and debugging software, like GDB for the MPPA and a tracing software based on *LTTng*. The eclipse-based ACCESSCORE supports different parallel programming models like a POSIX-environment which is C with some extensions, and ΣC—a cyclo-static dataflow language. Both are based on C and can make use of already written C code, without big changes. These are discussed in the following sections 3.3 and 3.4.

3.3 POSIX-like Programming

Kalray provides developers with the *process management and inter-process communication application programming interface* (MPPAIPC)—an API that is based on the POSIX IPC. It combines communication of Host and IO via PCIe, IOs and Cluster via the NoC and cores on their Cluster.

[1] `http://www.kalray.eu/`, as of January the 18th, 2015.

[2] `https://indico.cern.ch/event/272037/material/slides/0.pdf`, as of January the 21th,2015.

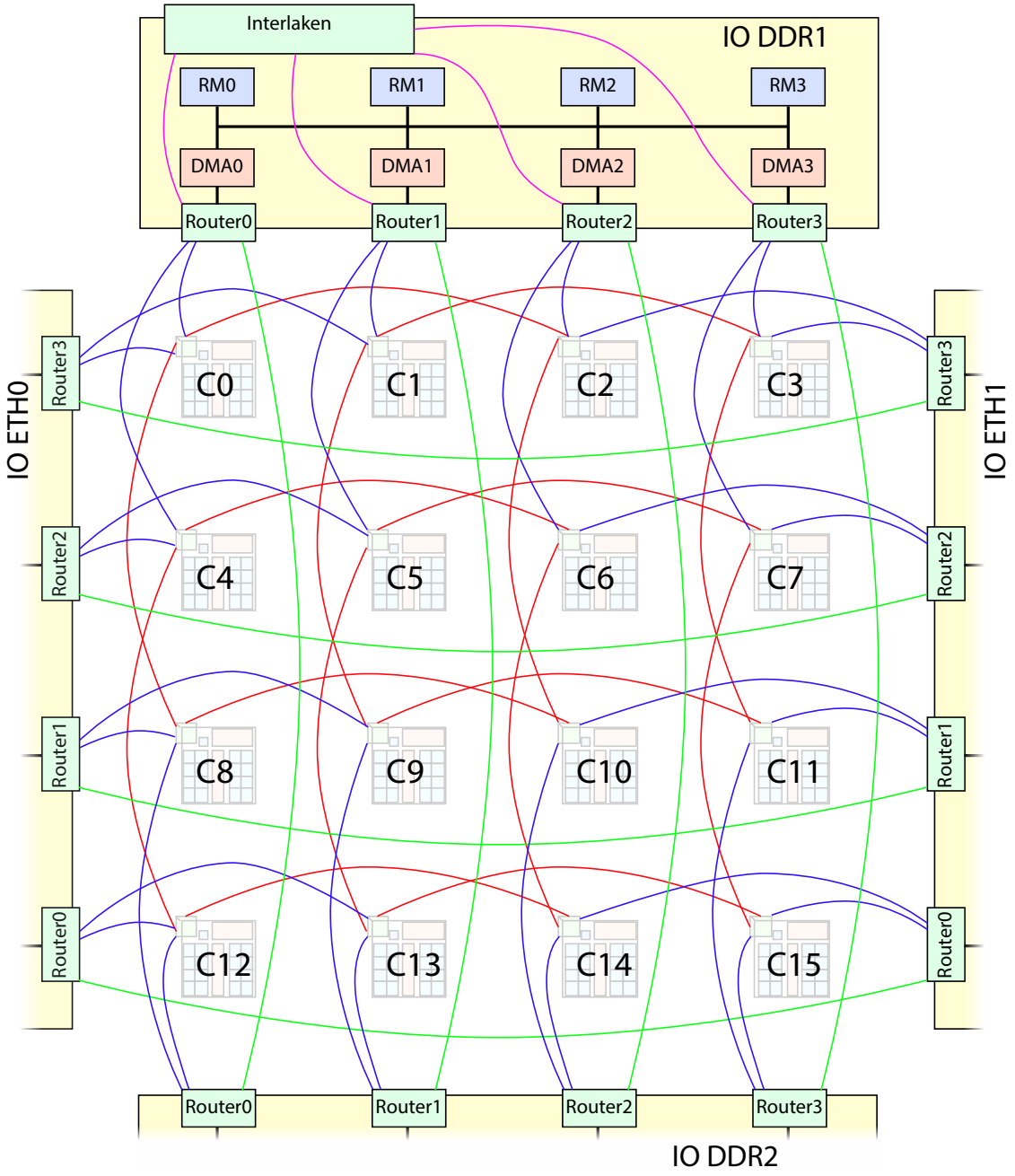

Figure 3.1: The components of the MPPA. The connections are bidirectional and symbolise the NoC (D-NoC and C-NoC cover the same routes). Red connections stand for cluster-to-cluster connections, blue for IO-to-cluster, green for IO-to-IO and pink for Interlaken-to-IO. All IO subsystems have a similar structure. The IO, completely drawn in the figure, is the one connected to the RAM and is responsible for handling clusters C0 to C7.

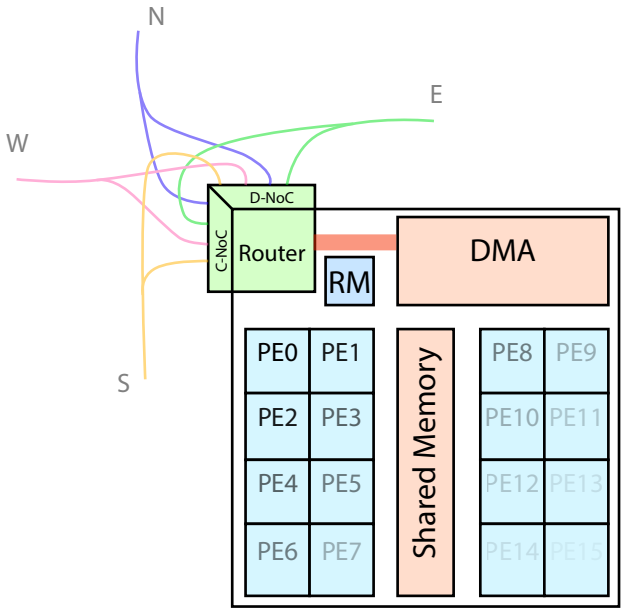

Figure 3.2: Structure of a single cluster. *DMA* - Direct Memory Access; *RM* - System Core; *PEs* - Compute Cores; *Shared Memory* - shared between all PEs; Tx-Processes form other clusters can directly write into the DMA, where memory of the Rx-connectors is allocated (see 3.3.3).

3.3.1 Constituents of a C program for the MPPA

In order to write our own programs we studied an example provided by Kalray which shows one of many ways to develop an application. This example adds two vectors invoking all 16 clusters. It shows the basic structure of a program for the MPPA which is based on three C source files, each containing a main procedure. Each of these C source files is compiled to a separate binary file, namely CLUSTER-binary, IO-binary and HOST-binary. These are described below.

The most basic is the CLUSTER-binary. It usually sets up the communication to the IO-system of the MPPA and implements the actual code for the task to work on. On this level we can start threads which run in a non-interruptible and non-preemptive mode[3], use C-libraries as we wish and communicate with the IO as well as to other clusters. The binary is received by the NoC router and the main() function is loaded onto core PE0, the first core of the 16, by the hidden RM core (see fig 3.2)

On the second level is the IO-binary file. It handles the connection to the clusters and to the host. The binary is loaded by PCIe. Moreover we have control over the whole periphery of the MPPA within the IO.

At last, the HOST-binary—It represents the host-sided interface between the host and the MPPA. Recall, at the moment the MPPA operates as a PCIe-device it depends on the host system, which is a Linux running on an x86 processor. The job of the HOST-C-source file is the initialisation of the MPPA and the communication between those two.

3.3.2 Connectors between Host and MPPA

Thus the MPPA is not a stand-alone machine, it needs to communicate with the Host system. Therefore two connectors are in use.

[3]this is a feature of one available toolchain (NodeOS) and is not valid for others (like RTEMS)

- The *Buffer* is a reserved address space for the Tx-process in the Rx-process's RAM. For the reader, this memory is blocked until the writing has finished.

- A message based connector named *MQueue*. It, as well, is one-directional and represents a queue that has a set size and can hold messages of variable size.

3.3.3 Connectors on the MPPA

The communication between IO and clusters, as well as clusters to clusters, can be performed in different ways using the following connectors:

- The *Sync* communication channel helps keep clusters and IO synchronous by blocking the IO until all clusters run over a certain point in their own code.

- A *Portal* is a one directional connection for data with inhomogeneous size. Writers can write to the portal with an arbitrary offset. For the reader, the Portal is blocked until a set trigger value is reached which counts the accesses of the writer.

- The *RQueue* is also one directional and many processes can write to it, but in this case the messages have to always have the same size and there is no offset. Instead, they are queued.

- For synchronisation and data transfer the *Channel* connector comes with a connection from writer to reader via the Data-NoC and an other connection in the opposite direction via the Control-NoC.

These connectors can be used with synchronous and asynchronous read-functions from the Rx-process.

Writing processes write directly into the memory (DMA see fig 3.1) of the reading process. If the same data has to be transmitted from the IOs to many clusters the *Sync* and *Portal* can make use of a multicast, or if all nodes are invoked, a broadcast, which is more efficient in terms of programming than series of write-calls, and under some circumstances, also in execution. The latter can only be indirectly influenced.

3.3.4 Program organization

As explained in section 3.3.1, a normal program for the MPPA consists at least of three source files, each representing a layer. The following list provides a short overview of what is usually done on the different layers.

Host

- sets up the MPPA

- communication to MPPA over connectors described in section 3.3.2

- unmounts the MPPA

- postprocessing of the received data

IO

- prepares data for each cluster
- spawns processes on clusters
- communication to clusters respectivly IOs over connectors described in section 3.3.3
- forwards data to the Host

Cluster

- prepares data for the threads
- starts threads
- Thread
  - computation of the actual task and communication over connectors
- waits until all threads terminated
- communication with the IO and exit

3.3.5 Compilation

Kalray developed a versions of gcc called *k1-gcc*, *k1-nodeos-gcc* and *k1-rtems-gcc*, which allows compiling in a habitual manner. Unfortunately are not all standard libraries available. The three C-source files (Host, IO, Cluster) mentioned earlier are compiled with a makefile that has a structure proposed by Kalray. This structure includes their own K1-TOOLCHAIN-makefile. In the actual makefile all flags can be set for single source files and also those for all K1-binaries (IO, Cluster). The Host source file is compiled with the normal *gcc*, while IO and Cluster are compiled with one of the three compilers above, depending on the choosen toolchain. All K1-binaries will be put together into a multibinary. This has to be asked for explicitly by the programmer from the makefile as follows:

```
multibin-objs := ${K1_IO_APPLI_NAME} ${K1_CLUSTER_APPLI_NAME}
```

In the makefile comments it is written, that the first list entry defines the binary that will be booted on the IO system, but this might be obsolete and it works as well when we switch the entries. In execution explanation of the Mandelbrot program (3.3.6) we will see why. In the end the Host-binary is a separated file from the multibinary and behaves like a normal program.

3.3.6 Program Launch and Execution

For explanation we use the *mandelbrot*-program we developed 4.3.

The first step is the execution of the Host-binary on the Host system with two arguments. The first argument is the path of the multibinary and the second is the name of the IO binary. After opening an *buffer*-connection (3.3.2), the host side loads the multibinary on to the IO of the MPPA with *mppa_load()* and then starts the IO binary by calling *mppa_spawn()*. One of the arguments of the spawn-function is the name of the binary to spawn. This way, it doesn't matter in which order all K1-binaries have been added

to the multibinary (see 3.3.5 for this problem). Moreover the Host opens connectors to read data from the IO.

When the IO is up, it loads and starts processes on the clusters as well with *mppa_spawn()* and the name of the cluster-binary as argument. With *mppa_open()* it establishes connections to Host as writer and to clusters as reader, by reading asynchronous using *mppa_aio_read()*.

Thus the clusters begin with their computation and communication to the IO and to each other. By the time the clusters finished their work, they contribute their result to the IO via the connectors and terminate by calling *mppa_exit()*. The last cluster writing data to the IO pulls a trigger which notifies the IO of an finished asynchronous read. Before the IO blocked on the function *mppa_aio_wait()*. This is one way of synchronisation, but we implemented a second one that makes use of the sync connector. After the IO collected all the data, it forwards it to the Host who is waiting for it by a synchronous read. The IO terminates calling *mppa_exit()*, while the host waited for this event with *mppa_waitpid()*. Now the last MPPA related function is *mppa_unload()*, that purges the multibinary from the MPPA.

3.4 Dataflow Programming with ΣC

ΣC [**?**] is a dataflow language build as an extension of C, and specially designed for the MPPA machine. This basis indicates a great opportunity to write C-environmental programs in a dataflow paradigm.

As in all dataflow languages, in ΣC we can use actors (here called *agents*) and connections between them—so-called channels. The most noticeable thing is that the program written in ΣC is composed of two elements: computation and connections. All functions inside the first agent section can be written fully in a subset of C. That is a great advantage as it allows for the reuse a lot of code of already existing programs. We can even imagine all the nodes as an isolated C program with inputs and outputs.

We can understand how ΣC works on the easy example of figure 3.3 which reverses the string it receives on its input channel. Let's go through the most important parts of the code. We can see at line 3 that an agent can take parameters, such as size in our case. However this parameter has to be known during compilation, which brightly shows that ΣC is an Static Dataflow Programming language. The rest of the agent is really easy to understand, since it's just C language code.

In the *subgraph* section at lines 16 and beyond, we can see the map function where we can specify which actors we want to use, and what the connection between them is. In our example, we are using the `reverse` agent which was specified on lines 3 to 15, and standard input and output agents from the `SigmaCstdio` library. Please notice that the variable `bsize` which identifies the size of transmitted data packets is constant, and defined during compilation. Lines 23 and 24 describe the connections between our agents.

The ΣC compilation process is rather complex. It is run in four separated phases. First one basically consists of syntax and grammar analyses. The second part compiles all the pure C components and creates the dataflow diagram from the subgraph section. Here it is possible to run a ΣC simulation, or move to the third phase, which tries to map the prepared program onto the MPPA machine. Finally if all the compilation processes are done, in the fourth part we can create one or a few binary executable files that will be downloaded on the desired components of the MPPA processor.

```
1  #include <SigmaCstdio.sch>
2  #define bsize 20
3  agent reverse (int size) {
4  interface {
5      in<char> input_port;
6      out<char> output_port;
7      spec {input_port[size]; output_port[size]};
8  }
9  void start (void) exchange (input_port input[size],
10 output_port output[size]){
11     int i;
12     for (i = 0; i < size; ++i)
13         output[i] = input[size-i];
14     }
15 }
16 subgraph root() {
17   interface {
18     spec{}; }
19 map {
20     agent reader = new StreamReader<char>(0x1, 11780, bsize);
21     agent rev = new reverse(bsize);
22     agent writer = new StreamWriter<char>(0x4e20, 11780, bsize);
23     connect(reader.output, rev.input_port);
24     connect(rev.output_port, writer.input); }
25 }
```

Figure 3.3: Simple example of Sigma C program [**?**]

Chapter 4

Experiments

This chapter groups a description of the experiments we set up for programming the MPPA. Section 4.1 describes the implementation of a Perlin Noise generator. Section 4.2 describes our attempt to implement and parallelize the Monte-Carlo algorithm in C. Section 4.3 describes our implementation of the Mandelbrot set algorithm both with POSIX-threads and Σ-C. Finally, section 4.4 describes our attempt to port the orcc compiler to the MPPA and related questions.

4.1 Perlin Noise

Perlin noise was developed by Ken Perlin in 1983 [1] in an attempt to create an algorithm that can give computer graphics more natural-looking textures. It is a type of coherent noise generator which gives the impression of a randomly-generated pattern.

This experiment (see figure 4.1) focuses on 2-dimensional Perlin noise. Because of its nature, a third dimension is hidden within the actual values of each pixel, therefore giving images a "height" element. Because of this, Perlin noise is being used to make other patten-based randomly-generated digital objects like procedurally-generated maps for video games. The popular game Minecraft[2] uses a modified version of the Perlin noise algorithm to create its terrain.

Algorithm overview

At the heart of the Perlin noise generation algorithm is a grid of nodes where the nodes are at a fixed distance away from one another. Each node is home to a vector that points in a random direction of random length. These vectors together influence and "push" on the underlying image to create the noise. Figure 4.2 shows what this grid might look like.

In order to generate the noise for a raster image one must sample each of the pixels individually. When sampling, you find which of the grid vectors are closest to the point you are sampling. Afterwards, you create vectors that points from the point being sampled to each of the grid vectors. We'll refer to these as the "relative vectors". Figure 4.3 illustrates the various vectors involved in this process.

[1] Some information at `http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html`, last checked January the 18th, 2015.

[2] `https://minecraft.net/`, last checked January the 18th, 2015

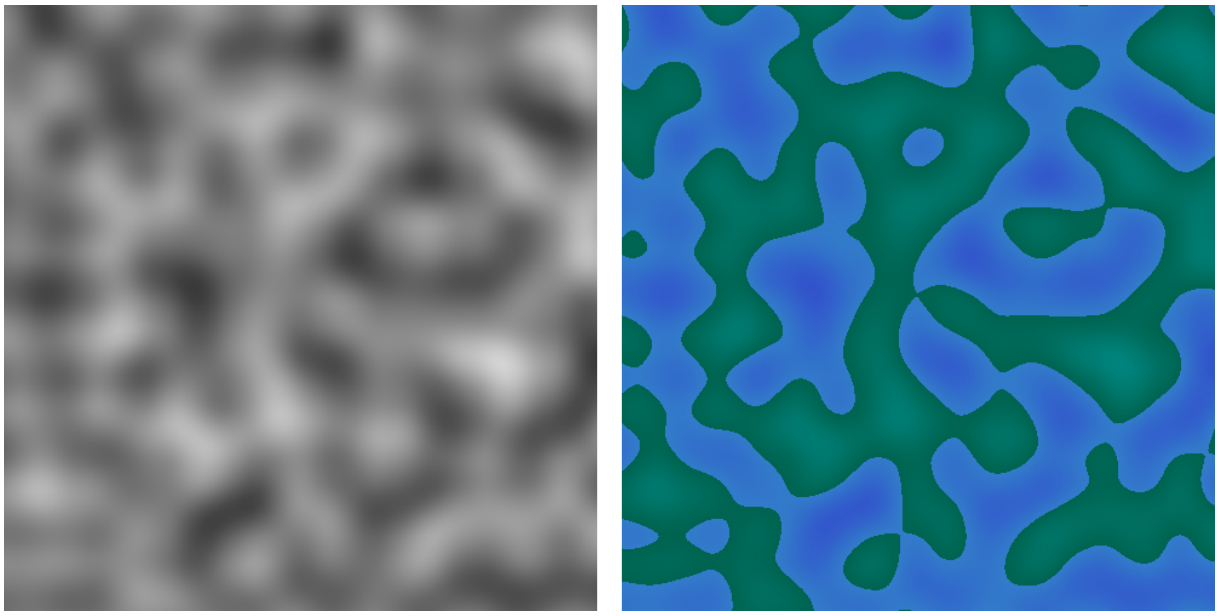

Figure 4.1: The same Perlin noise data, generated as a greyscale image on the left and as a map with on the right

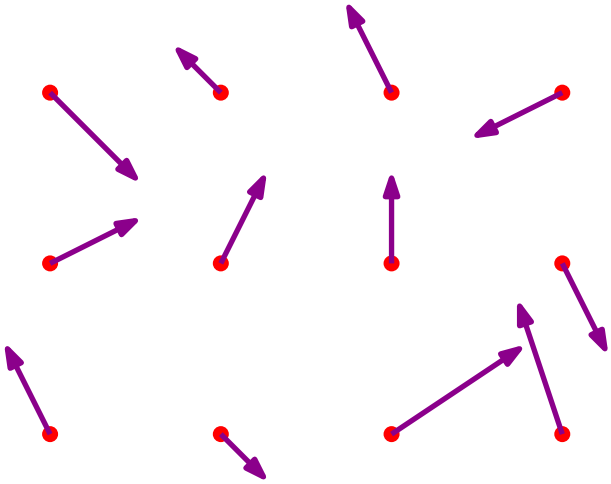

Figure 4.2: An example of the Perlin noise vector grid

Each of the grid vectors creates a gradient. By getting the dot product of each grid vector with the relative vector that points to it, we are told what value our sample point has on each of the grid vectors' gradients. We interpolate all of these values together to get the overall value for the combined gradient, which is a number in the range [-1 ... 1]. This is what represents the "third dimension" of the pixel.

Implementation

The first step in implementing Perlin noise is to randomly generate the grid that everything relies on. While this is very easily doable, parallelising it is difficult on the MPPA machine because of the lack of a thread-safe random number generator in Kalray's implementation of C. Rather than parallelizing this and rely on locks or more predictable RNGs, this step was left single-threaded. Because it's such a small part of the process in terms of computing time, this was considered a reasonable decision.

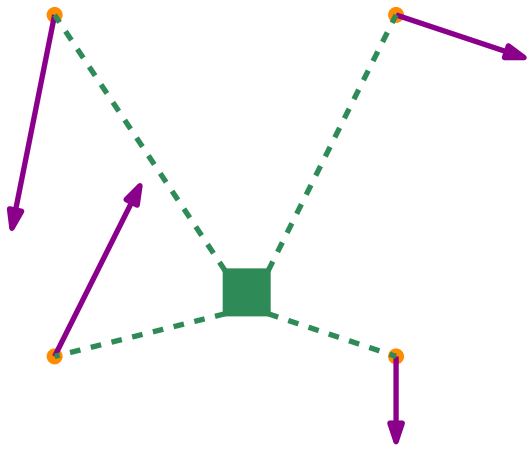

Figure 4.3: Example of a pixel sampling. The dotted lines are the relative vectors.

Once we have the random grid, a function $sample(x, y)$ was created. This function is fed the (x, y) coordinates of the pixel that is being sampled and returns the interpolated final value. The bulk of the work is done here, with the calculations described in the previous section being performed.

Parallelizing the sampling is a cinch. Since each pixel can be sampled on its own without affecting any of its neighbours, each thread can do its own work without writing over shared memory. The only common data is the vector grid (which is only read from) and the array of final values (where each thread has ownership over its own specific range).

Below is a table showing how the implementation scales with the amount of cores being used. The times are for creating a 50000×50000 pixel image with the vector nodes 2500 pixels apart. 3 seconds were subtracted from each of the results to account for overhead of getting into the MPPA environment. This benchmark shows that, at least on a single cluster, the program scales exceptionally well.

Cores	Time taken (min)
1	3:13
5	0:39
10	0:19
15	0:13

4.2 Monte-Carlo

4.2.1 A parallel Monte-Carlo-Algorithm

We searched for simple algorithms that, by their very nature, are perfect for parallelism. We found a special kind of algorithms called Monte-Carlo algorithms. The characteristic is that they rely on a random generator and that their result is not necessarily correct. But with a repeated execution of the algorithm, the possibility of failure can decrease. Furthermore the efficiency, depending on the problem, can be much more higher in comparison to other approaches.

We chose an algorithm that calculates π [**?**]. This well known and important number can be determined by a simple consideration. When we take a circle with the radius r and a square with the length of $2r$, the area of the circle is $\pi r^2$and the area of the square is r^2. Now the circle fits in the square and we have a setup for computing π using the following approach.

The actual algorithm is now to put random generated points into the square (see fig 4.4) and count whether they are in the circle or not. This is the part to be worked on in parallel, because the points do not depend on each other. Under the assumption that the number of the hits in the circle and the square are proportional to their area, the approximation 4.1 should be legit. Recalling the properties of a Monte-Carlo Algorithm, our result should be more precise, the more points we generate, for example by decomposing this algorithm into seperate threads, one thread being in charge of generating one point.

$$\frac{area\ of\ the\ circle}{area\ of\ the\ square} = \frac{r^2 * \pi}{(2*r)^2} = \frac{\pi}{4} \approx \frac{circle\ hits}{square\ hits} \tag{4.1}$$

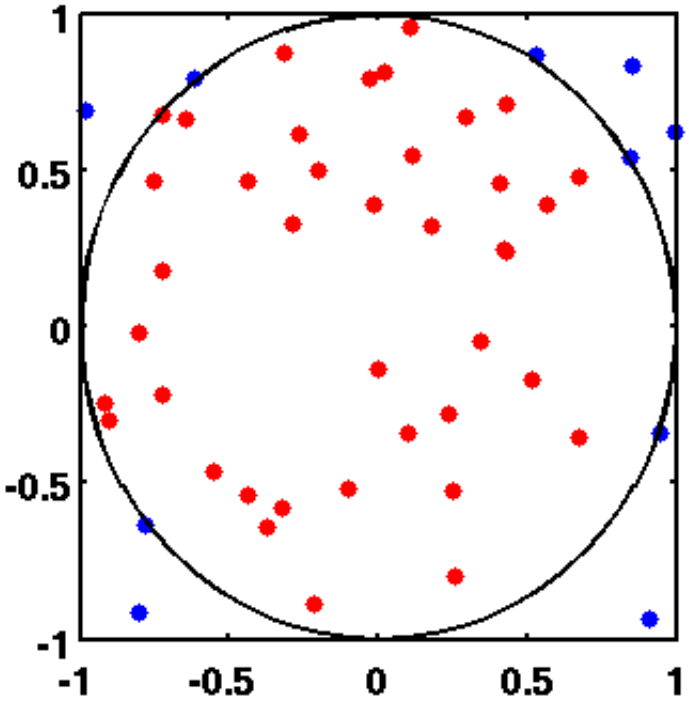

Figure 4.4: Clarification of the Monte-Carlo-Algorithm for calculation of π. The ratio of a square to a circle can be approximated by ratio of the count of random placed hits in the square or both. [source: wikipedia.org]

The C implementation is a bit simplified 4.5. Here we only treat one quarter of the circle and the rectangle. For the parallelisation, we generate threads produce random points and calculate their distance to the point $(0, 0)$. Then the distance is compared to the radius. Depending on this it is counted as a point in the circle or not. On exit the threads return the number of hits in the circle area to the main thread.

In the end, the main thread estimates π by the formula:

$$\frac{circle\ hits}{total\ hits} * 4 \tag{4.2}$$

To test the functionality of the program, we implemented it first for a x86 processor. We used the thread-safe drand48-functions that delivers a random double between 0 and 1 to get our random point coordinates. After everything worked right, we started to port the program to the Kalray MPPA. It uses only a single cluster and behaves almost like a normal x86 processor. Moreover, it takes care of everything

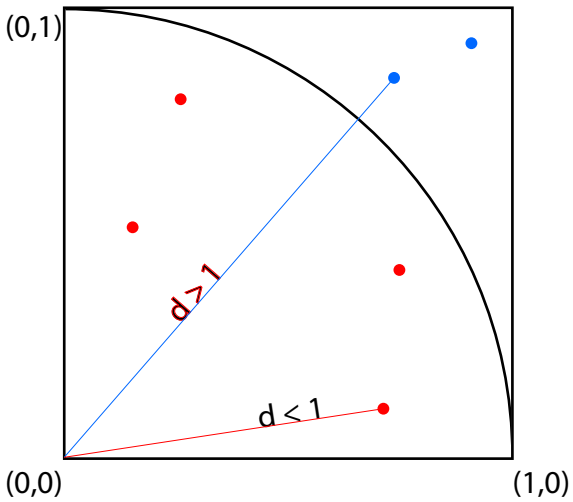

Figure 4.5: Clarification of the simplified π-algorithm. If the calculated distance is smaller or equal the radius (=1), the point is counted as an circle- and rectangle-hit, otherwise just as rectangle-hit.

that is higher in the hierarchy, like the NoC, the IO clusters and the host itself, not passing this burden to the programmer: the toolchain hides all the initialisation of those components. There are only some minor changes on the x86 code to make it run on the MPPA following this approach. It is essential to include `mppa/osconfig.h` header and to set the `CONFIGURE_MAXIMUM_POSIX_THREADS` macro.

However, a problem occurs concerning the drand48-functions. Potentially like other functions traditionally included in gcc toolchains, these are not available on the MPPA at the moment. For this reason, we had to find another randomise function. Unfortunately it is not that easy to write a function that distributes random values almost equal over the whole given range. Still, we picked code from the web and modified it a bit to return double values instead of integers and to use a time-function that returns milliseconds. This way we have a first snippet of code that runs on the MPPA, but the result is not really perfect. In fact, it gets worse if we increase the number of iterations, because of the self written randomise function.

4.3 Mandelbrot

4.3.1 Mandelbrot set with POSIX-Threads

Calculating π ran on 16 cores. Unfortunately, it did not work out so well. So far we still have not used all the computing power available on the MPPA at once. To do so, we considered an image processing exercise for the processor. A benefit that comes with image processing is that mistakes one makes in the function implementing the needed computation have a visible impact on the resulting image, at least in most of the cases. This helped a lot when it comes to debugging the code.

Basis of the Mandelbrot set

The Mandelbrot set was discovered by Benoit Mandelbrot. It is a pattern that contains itself recursively, a so called fractal, a fraction of itself. The formula below shows the simplicity the Mandelbrot set is based

on.

$$z_{n+1} = z_n^2 + c \tag{4.3}$$

All variables are in the complex plane. The c represents in our case the coordinates of a pixel, z_0 a fixed starting value and n the number of iterations have to be set. When the values are set we obtain a sequence of z-values, from which we can derive if it is bounded or not. Here are two examples[3]:

$$\begin{aligned} c = 1,\ z_0 = 0,\ n = 5 \quad &\rightarrow \quad 0,\ 1,\ 2,\ 5,\ 26 \quad \rightarrow \quad \textit{unbounded} \\ c = -1,\ z_0 = 0,\ n = 5 \quad &\rightarrow \quad 0,\ -1,\ 0,\ -1,\ 0 \quad \rightarrow \quad \textit{bounded} \end{aligned} \tag{4.4}$$

After the determination, bounded cs (pixels) belong to the Mandelbrot set and are coloured black. All other pixels are coloured in a colour corresponding to their degree of divergence. Every pixel can be calculated independently, this makes it perfect for parallelization.

Implementation

The bulk of our work here was to split the calculation of a picture of the Mandelbrot set, so that a multi-core processor can work on different parts in parallel. Moreover, we had to determine a limit for speedup, where the number n of threads the program does not gain speed up anymore compared to executing it with $n - 1$ threads. A benchmark is provided in 4.3.1. A set of of C-source and -header files were already provided. Those files give all the needed functionality, in other words the colour management, the calculation of the Mandelbrot fractal, and the function to write the bitmap-header. The most important function is given in figure 4.6.

```
/*
 * Writes a visual representation of the mandelbrot set into the supplied buffer.
 * Each pixel is represented by a pixel_data_t element containing the red, green
 * and blue components.
 *
 * It is possible to limit the calculation to a smaller part of the whole picture
 * with the parameters clip_x & y and clip_width and height.
 *
 * data:        Pointer a clip_width * clip_height large memory area the data is
 *              written to. The data is written row-wise, all pixel of the first
 *              row [x - (x + clip_width), y], followed by pixel of the second
 *              row [x - (x + clip_width), y + 1], and so on
 *              till [x - (x + clip_width), y + clip_height].
 * full_width:  Width of the complete picture
 * full_height: Height of the complete picture
 * clip_x:      x offset of the area to calculate
 * clip_y:      y offset of the area to calculate
 * clip_width:  Width of the area to calculate
 * clip_height: Height of the area to calculate
 */
void calc_mandelbrot(pixel_data_t *data, int full_width, int full_height,
                int clip_x, int clip_y, int clip_width, int clip_height);
```

Figure 4.6: header of the `calc_mandelbrot` function.

With this function it is possible to calculate only a certain area of the whole image. This is quite useful for our purpose, since a huge part we have to care about is that every cluster of the MPPA and every thread works on something different.

[3] from `http://en.wikipedia.org/wiki/Mandelbrot_set`

In this program the task of the Clusters is to generate threads, who generate each a different slice of the Mandelbrot set. A first merging is done by the main-thread of the Cluster. He assures that all thread-slices are transmitted to the IO. On IO-level all cluster-slices are concatenated to a whole picture. Finally the Host writes everything in to a Bitmap.

Benchmarking

To test the scalability we computed the Mandelbrot set with different number of clusters and threads per cluster. The resolution of 1024×512 pixels for one picture stayed the same over the whole benchmark. This results in a size of ca. 1.6 MB. We started to keep the time when the multibinary is loaded onto the IO and stopped when the IO called mppa_exit().[4]

Clusters	Threads per Cluster	Threads total	Time taken (ms)	speedup factor (wrt first line)
1	1	1	14223	1
1	2	2	7445	1.9
2	1	2	7427	1.9
2	2	4	5235	2.7
2	4	8	3118	4.5
4	2	8	3117	4.5
4	4	16	1793	7.9
4	8	32	1087	13.1
8	4	32	1083	13.1
8	8	64	736	19
8	15	120	684	20.8
15	8	120	631	22.5
16	15	240	560	25.4

From the table we can conclude that the speedup decreases with the number of threads. In the beginning the execution time is almost divided by 2, when we double the threads (compare first row with the two following). By redoubling 120 threads to 240 the speedup is less than a sixth. This is caused by the overhead and the synchronisation needed by the MPPAIPC and the part of the program, that is not parallelizable. Furthermore it is obvious that the difference between many threads on one core and the same amount of thread spread on the clusters do not really matter. The second variant is always slightly faster, probably because the combination of IO and NoC is faster than the PE0 running the main-thread on each cluster. Figure 4.7 is a visualisation of benchmark with more configurations. We can see once more that the MPPA scales its computation power approximately the same over clusters and threads.

4.3.2 Mandelbrot set in ΣC

The ΣC-programming language gives us the opportunity to write dataflow programs in a C-like syntax. In addition to that, we can include C-code that is already written. This lead us to try if it was possible to do the same image processing exercise in a different programming manner.

[4] In the speedup factor column, 1 means no speed. 2 means execution time is divided by 2, etc.

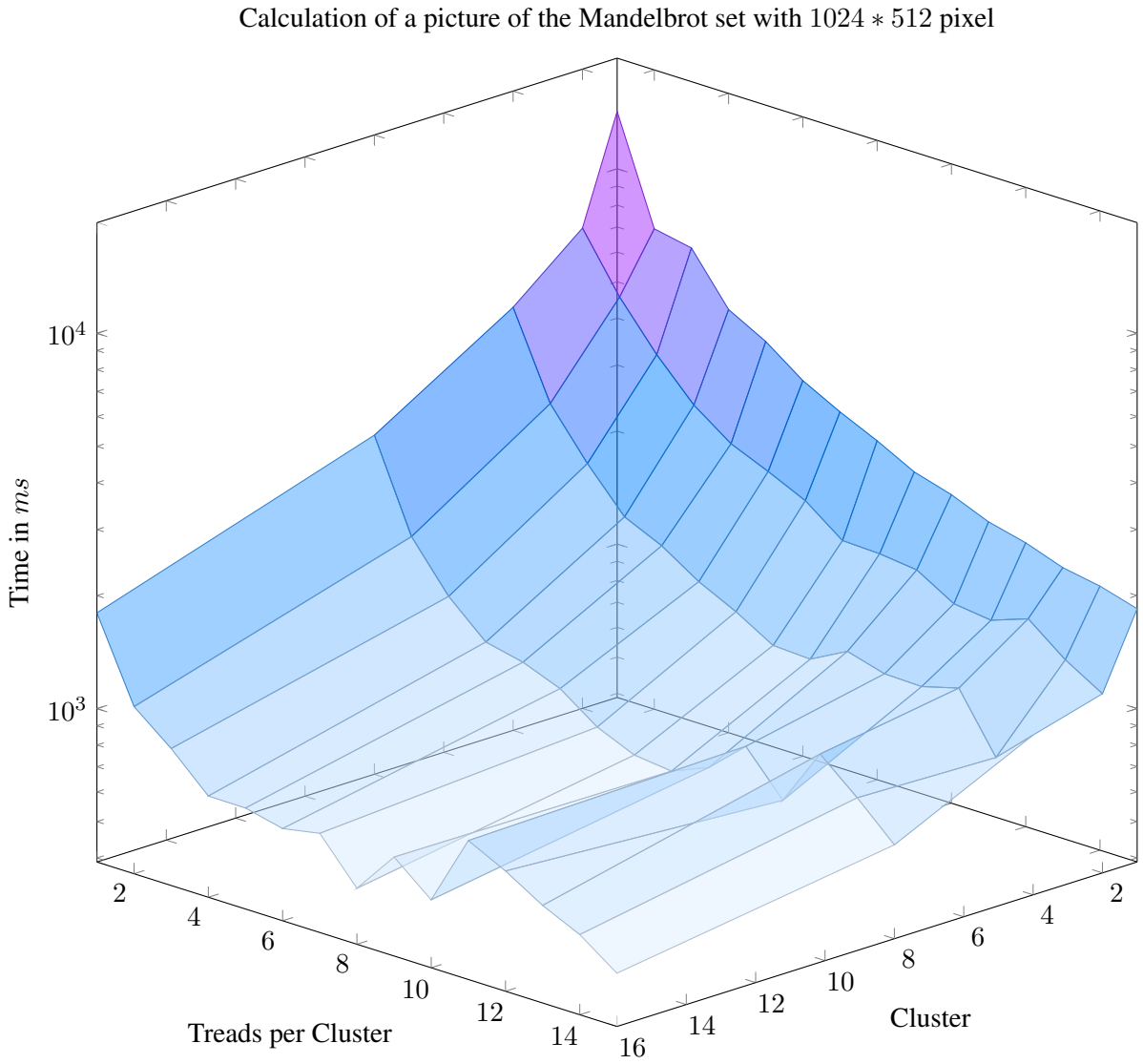

Figure 4.7: This plot shows the impact on the computation time when different a number of cluster and threads per cluster are used to calculate the Mandelbrot set.

Structure of the Program

For this task we took inspiration from the HelloWorld-example that comes with the MPPA Accesscore SDK. In our program we will use 4 kinds of agents. The first one calculates a part of the image and is called *mandelCalculator*, the second one is a build-in agent from a library that works as a data-stream-*joiner*, the third actor is also a build-in type who works as a *guide* and the last one merges all data to an image and is called *file composer*. It has similarities with the POSIX approach. We can compare the cluster with the first two agents, the IO with the *guide* and the Host with the *file composer*. Eventually this corresponds to where each agent is actuallly executed in the end.

The programming of the agents is straightforward. Besides ports and code of the agents, source files can describe subgraphs of the agents they hold or they included from headers. At least one source file has to implement the subgraph *root* which functions as a *main()*-agent. In our case *mandelCalculator* and *joiner* form one subgraph *imageProcessor* (see fig 4.8) because the number of inputs at the collector depends on the number of *mandelCalculator*. Further this source file comes with the *root* subgraph

(see figure 4.9). Two *joiners* and two *guides*(Dup, actually a duplicator; easies way for a necessary Cluster-Host connection) are used, each one to carry the picture data and the other one for the offset information.

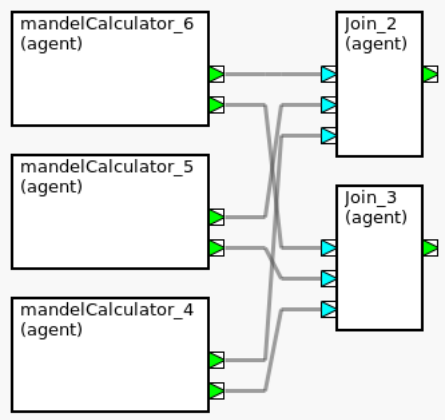

Figure 4.8: The *imageProcessor* subgraph with two different agents and two outputs.

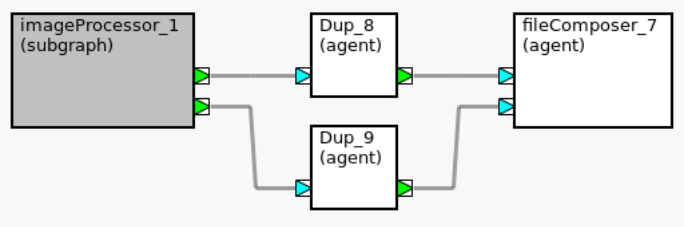

Figure 4.9: The *root* subgraph who has neither inputs nor outputs.

The listing in figure 4.10 shows the code of the *root* subgraph. It consists of an interface part, where inputs and outputs are declared, and a mapping part, where entities are generated and connected. Obviously the subgraph *imageProcessor* is handled like an agent. We can see that the declaration of agents like the *guide* and the *file composer* are followed by a command that sets the execution location for the mapping. The task of the *guide* is only to establish a connection between cluster and host. Theoretically the program could run completely on the Host with just a few modifications.

Problems

During the programming, the SDK often changed whitespaces with tabs and vice versa when it came to saving or loading the source code. This is not really a failure in ΣC but it happened only within its development environment and this is rather frustrating when the code is long and no auto-format is available.

If C-headers are included into ΣC-source files, the library-inclusions of the corresponding source files are marked as e.g. "Unresolved inclusion: <stdlib.h>". In the end it does not make a difference, the program compiles without error. Anyway, we have not found a way to work around this.

When it comes to the execution, we tried many different configurations of height and width of the picture as well certainly as varying the number of *mandelCalculators*. Starting with small resolutions like 256×256 on 16 *mandelCalculators* we increased to 2048×1024 with 128. Everything went fine and there

```
...
subgraph root()
{
        interface
        {
                spec{};
        }

        map
  {
                subgraph img_proc = new imageProcessor(NB_CALCS);
                agent fc = new fileComposer ();
                SigmaC_agent_setUnitType(fc, "native");
                agent guide = new Dup<pixel_data_t> (1, WIDTH*HEIGHT/NB_CALCS);
                SigmaC_agent_setUnitType(guide, "k1-I/O");
                agent guide2 = new Dup<unsigned int> (1, 1);
                SigmaC_agent_setUnitType(guide2, "k1-I/O");

                connect(img_proc.output,guide.input);
                connect(img_proc.output2,guide2.input);
                connect(guide.output[0], fc.input);
                connect(guide2.output[0], fc.input2);

        }
}
```

Figure 4.10: Graph creating within the root Σ-C agent.

seemed no limits. But after logging in again to the machine and recompiling the compilation exits with an mapping error. We tried it several times and this would not work, not before we reset resolution and agent amount to a minimum. From then on it was possible to increase all values again.

But the probably biggest problem is happens when the program on runs on the MPPA itself. Everything we executed ran in an MPPA-simulator on the host machine as expected. When it comes to the actual hardware, the terminal—linked to the MPPA—says the program has terminated, but nothing is printed in the terminal. An output file is not written either. Like in the preceding problem, the reasons were unclear.

4.4 Trying to compile RVC-Cal programs for the MPPA

4.4.1 Posix and Dataflow number factorisation

As mentioned before, we initially wanted to prepare and run some simple parallel programs on the MPPA with two different programming paradigms: POSIX multi-threaded programs, and a corresponding dataflow version. This could show the differences in programming complexity and performance.

The problem of number factorisation was chose. The idea was that we've got a set of numbers which we want to factorise. It's a good example of something that is easy to parallelise, where there is a possibility to split the complex problem thanks to the multiple approximately similar type of input. In those cases we can simply run a few parallel processes in which we'll run the computation for one tuple of data.

In this particular case, we wanted to run 14 threads/nodes (see simplified version on figure 4.11) so that each of them could factorise one number after another. For the purpose of experimentation, we didn't want to focus on complex and more efficient algorithms, but rather we've chosen the "brute force" method

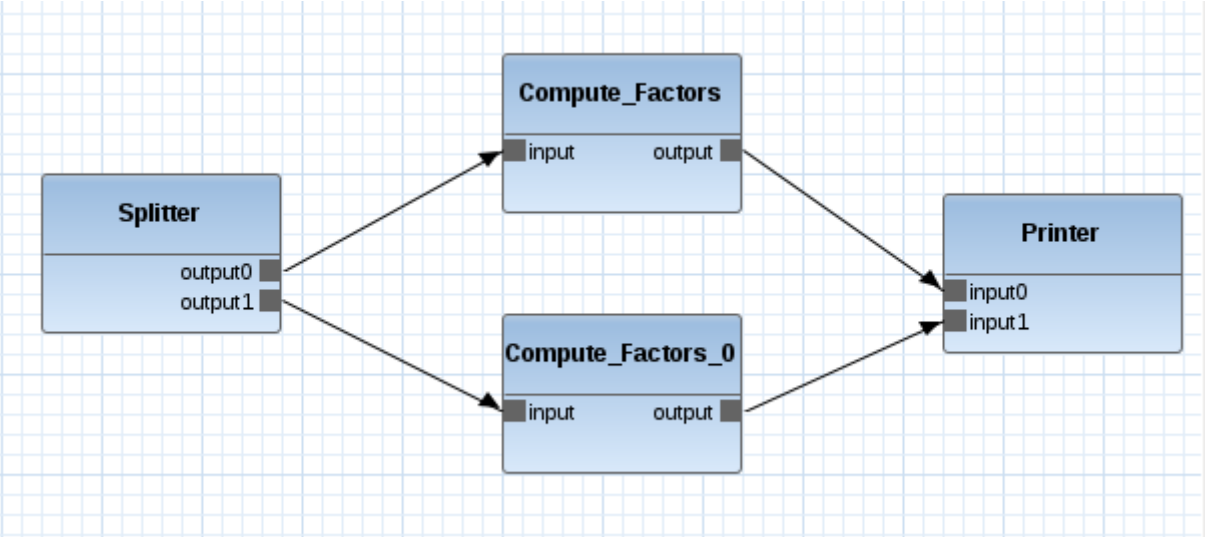

Figure 4.11: Simplified prime factors dataflow graph.

which consists of looping over the range $[2...N/2]$, and seeing if the index number can be divided against N. This loop should run until the number we are working on reaches zero.

Although it wasn't very complicated to write a POSIX version in C, the dataflow one caused a few problems. It's clear that because of the fact that we don't know the number of factors that each number has, we should use a dynamic dataflow model as described in section 2.3.

4.4.2 Orcc splitting data problem.

The Orcc RVC-Cal tool was used, where we faced a problem which taken to the easiest quantum, can be described as a data splitting problem.

Let's assume that there is a dataset which we want to divide and send to the two different actors. To provide it, we need one Splitter actor and two receivers (see figure 4.12). In this case we decided to set the size of the FIFO between splitter and other actors to 1 so there was no possibility for one node to get a few pieces of data at the same time. An important assumption we take for describing the problem we encountered is that Actors `Act1` and `Act2` don't work with equal efficiency considering their respective input data. Like for the example of the number factorisation problem, our goal is to have the ability to run a few less complex data computation on one actor, while the second one is working on a larger task.

Initially Orcc doesn't support this solution. It's related to the way how Orcc schedule the actions.

There are a few ways to manage it. We can, for instance, specify the order in which our actions should be fired, or simply prioritise all the actions. However, none of these solutions can help with our issue. That's why we decided to get a bit deeper, and try to force Orcc in some way to schedule the actions in the way we wanted it to. That's why we didn't use any of the solutions provided by Orcc, so actions should be fired non-deterministically.

The first idea, was to add an additional output from `Act1`, and `Act2` which could control the data flow from `splitter` to `Act1` and `Act2` by running the actions when the previous one have finished (see figure 4.13). In this case, `Act1` and `Act2` would produce the tokens at the end of computation, and Splitter would send the new data only when it would get a new token on the corresponding input.

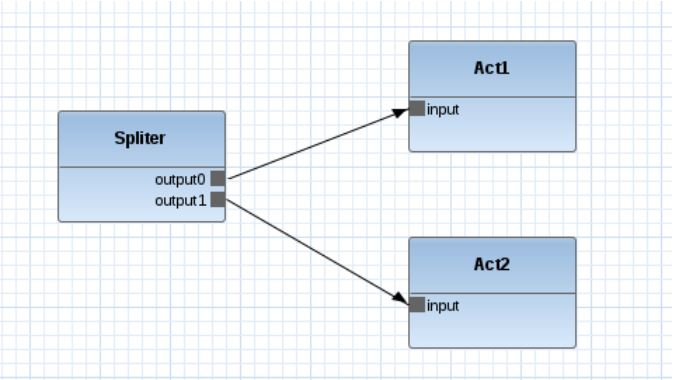

Figure 4.12: Instance of a data splitting graph.

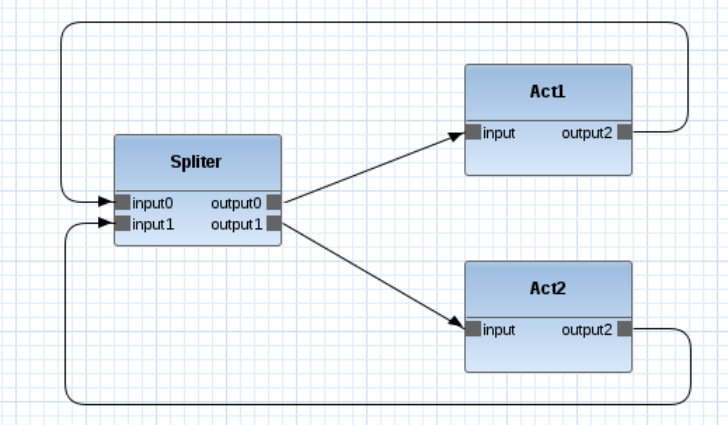

Figure 4.13: First attempt to solve the data splitter problem.

Theoretically, this idea should have solved the problem. However in practice, it changes nothing. Orcc makes the splitter wait for both of its inputs to be filled, and then sends data to both actors at the same time, so we tried to look into the C source files compiled by Orcc.

To realise the difficulty of the problem, we've got to understand the basic organisation of Orcc build C files.

In our particular case there are 3 files corresponding to the actors, and plenty of others providing standard operations like runtime data structures and scheduling functionalities. The splitter has 2 actions which are scheduled in one infinite loop.Actions basically consist on reading the new line from the dataset, and sending it to the appropriate actor. In this case the only guarding condition is initially checking if there is still something to send. However, we can consider waiting for the input as the second one.

Now, let's check step by step how the program runs. When Splitter starts to work, at first in the loop it looks for the action which should be scheduled next. According to the fact that all the actions are scheduled non-deterministically, theoretically all of them should be prepared to run. When the scheduler finds the ready action, it checks its guarding conditions, if they are satisfied the action is set otherwise the loop is quit without running the action.

The problem in the initial solution was that orcc doesn't check if there is a space in the FIFO between splitter and receiver. See figure 4.14.

This policy disables sending the data in the way we wish to do it. It might seem that moving the checking of the guarding conditions up to the place where we check if the action is next in the schedule,

```
While(1){
        If schedulable then
                If guards then
                        Run action1
                        goto finished
                Else goto finished
        If schedulable then
                If guards then
                        Run action2
                        goto finished
                Else goto finished
}
Finished:
Write_output_0;
Write_output_1;
```

Figure 4.14: Pseudo code, demonstrating C file representing Splitter

and moving Write_Output_0 to the inside of the if condition could solve the problem.

However, this solution is wrong. The policy of Orcc compiler is that all the outputs are sent in the same time. Changes done only in splitter file are not sufficient. The complexity of the other files in the Orcc libraries makes the adaptation of Orcc for using this kind of splitting, at least really hard, and probably requires rewriting the compiler.

Unfortunately it's still not the end of the problems with Orcc. We tried to adjust the Orcc-generated C-files, so they could be run on MPPA. However, due to complexity of Orcc libraries, we didn't manage it.

Chapter 5

Conclusion

We experimented on a new kind of processor architecture and tried out different parallel programming models on it, specifically pthread-based and dataflow-based approaches.

In our attempts to program using pthreads, we achieved very good scaling when the work was spread out across only one cluster, with experiments such as the Perlin Noise program scaling extremely well. However using the entire capacity of the MPPA processor was more difficult, as we needed to use platform-specific APIs to achieve inter-cluster communication. This also makes it difficult to easily port existing pthreads-parallelised programs to the MPPA platform while taking full advantage of the hardware.

Also due to this cluster-based architecture we achieved less-than-satisfactory scaling when trying to spread the Mandelbrot program's execution across multiple clusters, which we believe is due to the overhead involved in inter-cluster communication.

Orcc dataflow programs compile to *C with pthreads* programs. We were unsuccessful in getting these programs to compile in the MPPA environment because of certain libraries that Orcc uses as well as the difficulty in using its build environment. Even if we would have been successful, great effort would have had to go into editing these programs manually if we expected to run on multiple clusters, as mentioned above.

The ΣC dataflow approach available on the MPPA seemed to be a promising way to access its entire computing power. The underlying architecture is not completely hidden however, and we needed to manually map some of the different layers of the topology. Beyond that, writing the programs was easy. Unfortunately we were not able to run a program on the MPPA and get results from it which made it impossible to compare the two programming models in term of performance.

Overall, the MPPA architecture is very promising and what Kalray have been able to achieve is impressive. If it were to have an improved pthread implementation that takes care of the inter-cluster communication by itself, it would be a great platform to work on and it would allow for a lot more possibilities. As well as that, the ΣC environment is interesting to use but ultimately its implementation needs more work to achieve the stability required of it.