PARALLELLA COMPUTING: ANOTHER DISTRIBUTED SYSTEM STORY

Noémien Kocher

A semester project

Department of computer science
University of Applied Sciences of Western Switzerland
Fribourg, Switzerland

Supervisors: Baltensperger Richard, Kuonen Pierre, Roche Jean-François, Gonçalves Lourenço Marco José

Abstract

The last decade has seen a huge rise in processing power demand. Since single systems are limited and hardly scalable, we look at distributed systems to cope this rising need of high performance computing. Parallella is the first distributed system using the Epiphany architecture. It has been designed to be energy efficient and scalable to thousands of cores, responding to high processing needs. The Epiphany architecture consist of a 2D mesh network of cores, programmable in plain C/C++. Despite its interesting features, it has not yet gained a major attraction and it is still in a growing stage. This paper is cut into four chapters. The first one (*Exploration*) aims at discovering what the Parallella is capable of, by exploring its functionalities. In the second chapter (*In deep*), some examples will be discussed, and the third chapter (*A practical example*) will analyze some real-life applications using its potential. The last chapter (*Final thoughts*) stands for the conclusion and further work.

Keywords. Parallella, Epiphany, distributed system, benchmark

Glossary

DMA Direct memory access, piece of hardware that performs data transfer independently of the CPU. 8, 13, 19–21

eCore An Epiphany core. 4, 5, 7–14, 16–19, 21–27, 29, 30, 32–37, 39–43

Epiphany Name of a 16 coprocessors cores architecture developed by the Adapteva company. 1, 4–9, 11–14, 16, 24–27, 32–42, 46, 47

FPGA Field-Programmable Gate Array, configurable integrated circuit. 7

LDF Linker Description File, it is a file used to map data and portion of code to the memory. 10, 14, 16, 19, 23

SDK Software Development Kit, set of software development tools that allows the creation of specific applications. 2, 12, 13, 18–20, 30, 47

SREC Standard assembly file created by Motorola. 11, 13, 14

Contents

List of Figures

Chapter 1

Exploration

This chapter discusses the general architecture of the parallella board, it's memory handling, the development stages, and the tools provided by Adapteva. The aim of this chapter is to have a global overview of the parallella board and the necessary key concepts to start developing on the parallella.

1.1 Introduction

Parallella was launched in 2012 by the Adapteva company. It is the first product based on the Epiphany architecture, providing a high-performance, energy-efficient and manycore architecture for real-time embeded systems. The Parallella aims to meet 5 constraints [7] :

- Energy efficiency
- High raw performance
- Scalable to thousands of cores
- Emplementable by a team of 5 engineers

Facing a huge quest for more computation performance, we are now moving on integration of manycore architecture on a single chip, sometimes called multicore or manycore architecture. A manycore architecture, like the Parallella, consists of a high number of cores. What makes it special is the way those cores can interact together connected as a mesh network, and how they easily provide scalable architecture.

Parallella is a credit card size computer containing 16 Epiphany cores (eCores). The development of this board was financed in 2102 with a crowd founding website called Kickstarter. In less than 30 days, Adapteva had raised close to 1M USD, making them the first semiconductor company in history to successfully crowd-fund development[7]. Figure 1.1 is a close look at a parallella.

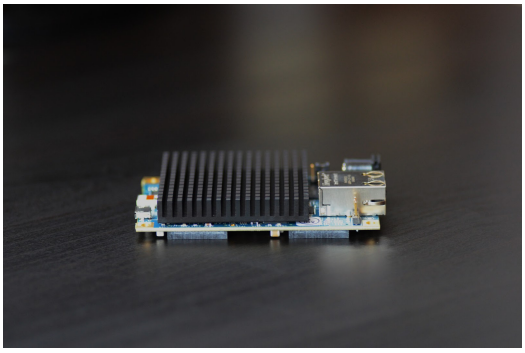

Figure 1.1: A close look at the parallela board

In this document, we are going to discover how we can exploit the potential of Parallella by studying it's architecture and developing some test-purposes programs.

1.2 Architecture

The parallella board includes a 866MHz dual-core ARM-A9 Zynq System-On-Chip and the Adapteva's Epiphany multicore coprocessor [6]. There are 3 different models of the parallella. The "Desktop" and the "Microserver" version with a Xilinx Zynq Dual-core ARM A9 XC7Z010 and an "Embeded" version with a Xilinx Zynq Dual-core ARM A9 XC7Z020. The main differences between them are the host processor versions and the FPGA specification.

The central processor is the Zynq 7000 AP SoC, which is a processor combining a standard ARM dual-core along with an FPGA. As shown in figure 1.2, the Zynq chip hosts an FPGA which assumes the communication between the Adapteva's Epiphany multicore side and the host side. The host side is seen as the Zynq board, which hosts the ARM processor, the memory as well as the peripherals.

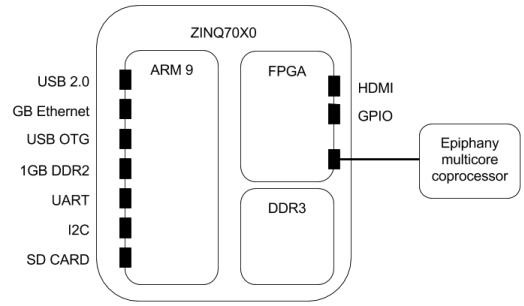

Figure 1.2: The architecture of a Parallella board

The Epiphany multicore coprocessor consists of a 2D squared grid composed by 16 Epiphany cores (eCores). Each eCore is linked with its neighbors to compose a networked set of eCores. Figure 1.3 shows how each eCore is assembled as a mesh network. Each node is an eCore that acts as a network router as well. Write transactions between eCores can work at an operating frequency of 1GHz and with a latency of 1.5 clock cycles per routing hop[1]. A transaction traversing from the left edge to right edge of a 16 eCores grid would thus take 6 clocks.

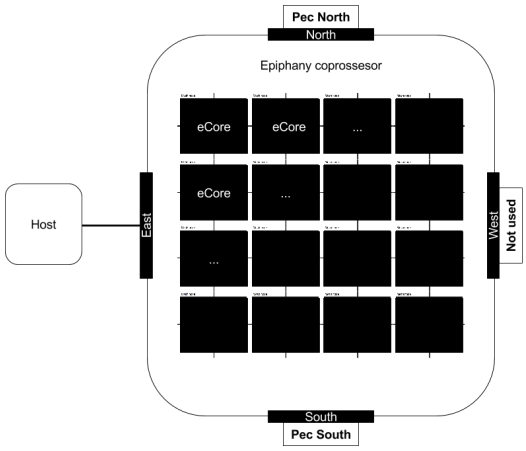

Figure 1.3: 16 cores assembled as a grid and composing the coprocessor of the parallella board

Figure 1.4 gives a closer look and shows what is contained in a single eCore. It is interesting to see that each eCore has it's own registers, DMA, local memory and everything to compute floating point operations. The local

memory is not directly situated on the eCore, but mapped from the host memory[6]. Data can be routed among eCores, meaning that each eCore has its own network interface that are capable of routing data.

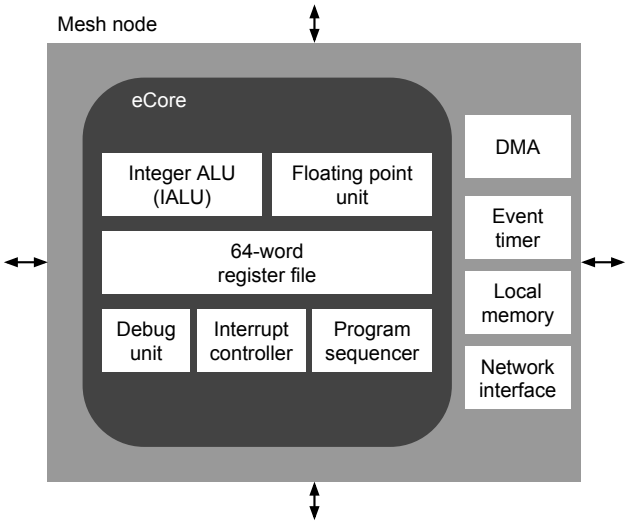

Figure 1.4: The architecture of a single eCore

1.3 Memory

The Zinq chip offers 1GB of DDR3 Memory to the Epiphany, mapping it from `0x8000'0000` to `0xBFFF'FFFF`, each addresses giving a 32-bit memory space. The equation 1.1 shows us how to obtain the 1GB space.

$$2^{7 \cdot 4+2} \cdot 32 = 1,073,741,824 \tag{1.1}$$

The Epiphany architecture uses 32-bit words to address memory, each of whose are word-aligned. It means that each addresses are divisible by 4, giving 4 bytes by words to address. Each eCore has its own local memory accessible from `0x0000` to `0x7FFF`. That memory is aliased from the global memory. Each eCore also have a globally addressable ID to ensure communication with all other eCores. The eCore ID is composed with 6-bits to address row and 6-bits to address column in the grid. Hence allowing a maximum of 4096 eCores in the grid (see equation 1.2). Those bits are situated

at the upper most-significant bits (MSB) of the address space.

$$2^{6+6} = 4096 \tag{1.2}$$

With space going from `0x0000` to `0x7FFF`, we see that the address range given for each eCore corresponds to 32KB (eq 1.3), which is splited in 4 bank of 8 Bytes.

$$2^{12+3} = 32,768 \tag{1.3}$$

That memory is used to store eCores' local information. However, the memory space for each eCore is bigger and table 1.1[1] summarizes the eCores' memory map.

Table 1.1: Local memory mapped for each eCore

Description	Start adress	End adress	Size [Bytes]
Interrupt vector bank	0x00	0x3F	64
Bank 0	0x40	0x1FFF	8K-64
Bank 1	0x2000	0x3FFF	8K
Bank 2	0x4000	0x5FFF	8K
Bank 3	0x6000	0x7FFF	8K
Reserved	0x8000	0xEFFFF	n/a
Memory Mapped Registers	0xF0000	0xF07FF	2048
Reserved	0xF0800	0xFFFFF	n/a

An interesting thing shown in table 1.1 is that each eCore's registers are accessible via memory mapped registers [6].

1.3.1 Real mapping

A linker description file (LDF) specifies where data and portion of code resides. That file is used at the linker stage to create a single executable [8]. The parallella is shipped with its specific LDF and, thanks to open hardware, freely open. All hardware sources are available on Github as a public repository [3].

The basic LDF that we will use maps the global memory from `0x8F00'0000` to `0x8FFF'FFFF`. That space range is the shared memory among all eCores. Its size, given by equation (1.4) is 16MB.

$$2^{4.6} = 16,777,216 \tag{1.4}$$

Another bank of shared memory if mapped from `0x8e00'0000` to `0x8eff'ffff`. That bank of memory is not currently used.

1.3.2 Data transfer

When an eCore in the top left sends data to an eCore in bottom right, data first travel through each intermediate eCores following the same row id. Then, when the column id matches the destination column id, data move along rows. As we saw in the previous section (1.3), each eCore has a global address ID composed with a 6-bit row id and a 6-bit column id. Data are routed using the column and row id. Figure 1.5 illustrates the path of a data sent from the top left eCore to the bottom right eCore. The dashed arrows represent what would be the reverse path, which is different than the coming path.

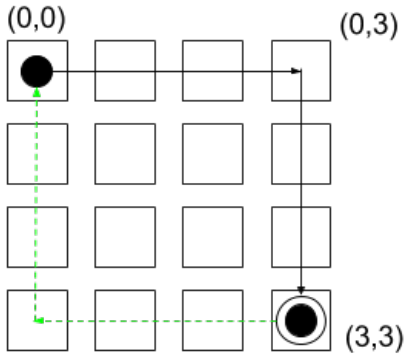

Figure 1.5: Data path

1.4 Workflow

Programming on the parallella is complex. Because the Epiphany coprocessor is not directly accessible to code, we must use the host part of the parallella to send SREC file to the Epiphany part. Then, on runtime, the compiled code is sent to the Epiphany cores. SREC files are the only file accepted by the Epiphany architecture. It a standard assembly file created by Motorola.

We, thus, have two distinct programs :

- the host one, which acts as the main entry point and sends the compiled code to the Epiphany
- the Epiphany program, which runs on each eCore and is sent by the main program

Figure 1.6 illustrates the principle, where the host part first fetches the compiled Epiphany code and then deploys it to the eCores.

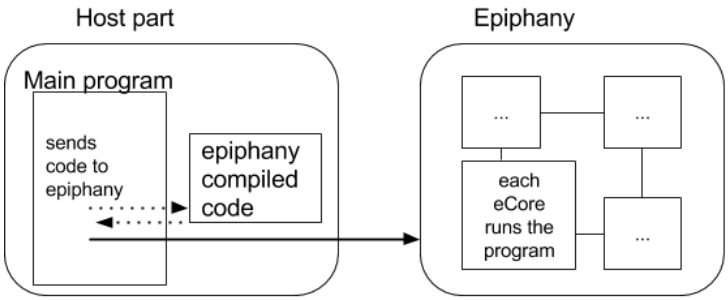

Figure 1.6: Development workflow

A major downside of this process is that eCores, once deployed, are totally isolated from the host part. Which means that it is impossible to print things from the eCores, thus making debug a bit harder. Communicating between the host and the Epiphany will only be possible with the aid of shared buffers memory or using local eCores' memory.

1.5 Workgroup

It is possible to treat blocs of eCores instead of all at the same time. When deploying source code to the eCores, one can create a workgroup of eCores and only target that workgroup. A workgroup consists of an eCores' rectangle out of the mesh network.

Workgroup has the advantage that a dedicated link is used to move data among eCores of the same workgroup[1], thus giving faster data transfers. We also use workgroups to manage memory and data transfers. Those points will be discovered in the rest of this document.

Using the provided SDK, workgroups are created using `e_open`. This method sets a variable of type `e_epiphany_t`, which represents a workgroup.

This variable will be used to send data to the eCores' workgroup and performing some manipulations on those. We also use workgroup to load Epiphany specific program (see section 1.4).

1.6 SDKs

Adapteva provides two SDKs, giving out-of-the-box ability to run any application written in ANSI-C. SDK's are :

- The Epiphany Hardware Abstract Library (HAL)
- The Epiphany Runtime Library (e-lib)

The Epiphany Hardware Abstraction Library (HAL) is meant to be used on the host. It provides methods to initialize the Epiphany system, create workgroups, start eCores and allocate global memory. That library is the software link between the host part and the Epiphany part.

The Epiphany Runtime Library (e-lib) runs on the eCores. It provides many methods to exploit the Epiphany-specific world like data transactions, interruptions handling, registers accesses, timer manipulation, DMA transfers, mutex and barriers functions and workgroup manipulations [2].

Those SDKs must be included in the compilation stage (see next chapter) and are freely available on Github[4]. Once done, they can be included with `#include <e-lib.h>` and `#include <e-hal.h>`.

Documentation for those SDKs is provided by Adateva and easy to follow. The use of those SDKs makes development for the Epiphany easier, it is a great part of the tools provided for the Epiphany.

1.7 Compiling

The Epiphany uses standard c programming environment based on the GNU toolchain[7]. So, happily, we will use standard tools like the `gcc` compiler.

Compiling a program to run on the Epiphany has several steps. We will first compile the host program with `gcc` tool, then compile the Epiphany specific program, then finally convert the Epiphany specific compiled code to SREC file.

In several examples, Adapteva provides build scripts examples to compile applications. We will comment the main phases.

Compiling host program :

```
${CROSS_PREFIX}gcc main.c -o main.elf  ${EINCS} ${ELIBS} -le-
    hal -le-loader -lpthread}
```

`-le-hal` loads the HAL SDK (see section 1.6). `-le-loader` loads the loader utility, responsible of loading programms onto the hardware platform[2]. In our case, that enables us to load our SREC file onto eCores. `-lpthread` gives access to the linker of the `pthread` library.

Compiling Epiphany program:

```
e-gcc -T ${ELDF} -O${OPT} emain.c -o emain.elf -le-lib
```

`e-gcc` is based on `gcc` and part of the Epiphany toolchain, it's job is to create a program that is suitable for the Epiphany architecture. `-T` provides a custom linker description file (LDF). `-O` sets the optimization level, tt can speeds up compilation stage but slow down the runtime.

Convert Epiphany binary to SREC :

```
e-objcopy --srec-forceS3 --output-target srec emain.elf emain.
    srec
```

`e-objcopy` is a provided utility to convert binary file to SREC file. Provided options sets correct format for the SREC file.

The compilation stage is specific and would be a pain without any examples. But fortunately, the parallella is shipped with many examples that help to understand and utilities provided gives a great toolchain for the Epiphany. The distributions available for the parallella includes by default the Epiphany toolchain.

1.8 Conclusion

Having an overview of the parallella functionalities took what we planned, about one third of the time allocated for this project. It wasn't always easy to find specific information with the documentation provided by Adapteva. For example, to know the memory mapping, looking at the documentation wasn't enough. We had to browse the hardware sources to find out what we were looking at. Documentation for the Epiphany architecture[1] is in general quite complete and done well. But the parallella documentation sometimes leaves a feeling of being hastily drawn. Finding books or articles speaking about the parallella didn't give much results, there is only few ones and it

is understandable for a rising system like the parallella. For somebody not familiar with the parallella ecosystem, it may take time to find its feet.

Chapter 2

In deep

The goal of this chapter is to practice what was learnt previously. It discusses some examples of memory transfers among the host and eCores as well as the implementation of the linpack benchmark. Examples will provide some pieces of code and schemes to help understanding the processes explained.

2.1 Example: Transfer between eCore and Host

Example available on GitHub[10]: *simple-epiphany/hello-from-epiphany*.

Figure 2.1 is a summary of the operations we are going to do so as to transfer data between the Epiphany and the host part. We are going to:

1. set a shared buffer from the host
2. set a pointer to the shared buffer from the Epiphany, then write to it
3. from the host, read data written from the epiphany to the shared buffer

Setting a shared buffer from the host is achieved with `e_alloc`. This method takes a variable's reference of type `e_mem_t` (representing the shared buffer) and allocs memory space on it. The following code declares our shared buffer then allocs a certain space.

```
e_mem_t emem; // shared memory buffer
e_alloc(&emem, BUFOFFSET, sizeof(unsigned)*ncores);
```

As we will see in the other examples, `e_alloc` allocates memory from address `0x8e00'0000`. Our linker description file (LDF) tells us that shared

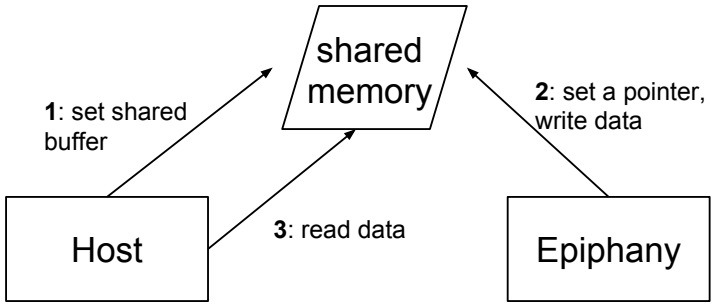

Figure 2.1: Simplistic scheme of transferring data from epiphany to host

memory is allocated from `0x8f00'0000`, thus we need an offset of `0x0100'0000` to start writing at the right address (`0x8e00'0000` + `0x8100'0000` = `0x8f00'0000`).

To be able to write to this shared buffer from the eCore, we need to set a pointer to the right address. In our case it will be `0x8f00'0000`. The following code, run by each eCores, sets a pointer beginning from `08f00'0000` plus an offset depending of each eCore's number. Then we write something to the buffer's address. Those data will be available from the host part. By doing so, we directly write to the memory shared buffer we set in the host part.

```
volatile unsigned *result; // Pointer to the shared buffer
result  = (volatile unsigned *) (0x8f000000
                                 + sizeof(unsigned)*num);
*result = num;
```

On the host, to finally read what eCores put, we use `e_read` to copy the buffer data to a local variable. The following code places the data from our shared buffer to a local array and then prints the result.

```
int result[ncores];
e_read(&emem, 0, 0, 0x0, result, ncores * sizeof(int));
for(i = 0; i < ncores; i++)
  printf("Result from core %02i is 0x%04x\n",i, result[i]);
```

2.2 Example: Transfer between eCore and eCore

Examples available on GitHub[10]: *simple-epiphany/hello-from-neighbor-ecore*.

In figure 2.2, we see that communicating from an eCore to another will be achieved by using eCores' local memory. The SDK gives us methods to manipulate data between each eCores. We are going to:

1. from one eCore, get the coordinates of it's neighbor
2. given the coordinates, writing to the eCore's neighbor local memory
3. from the neighbor eCore, read local memory so as to get the data

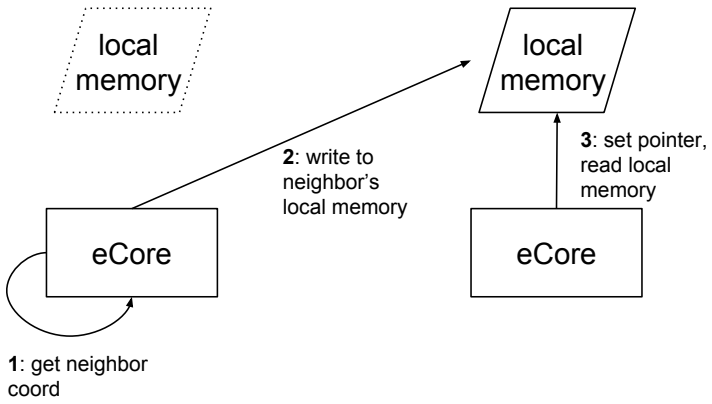

Figure 2.2: Simplistic scheme of transferring data from an eCore to another eCore

Reading and writing from an eCore to another one is achieved with `e_read` and `e_write`. Those methods take as parameter the row_id and the col_id of the eCore we want to read/write. The SDK gives us a method to get the coordinates of an eCore's neighbor. In the following code, written for an eCore, we retrieve the coordinates of the neighbor and then read a char at its locale memory.

```
unsigned neighbor_row, neighbor_col;
e_neighbor_id(E_NEXT_CORE, E_GROUP_WRAP,
              &neighbor_row, &neighbor_col);
e_read(&e_group_config,&neighbor_status,
       neighbor_row,neighbor_col,(char*)0x4000,1);
```

We note that we gave `0x4000` as the offset memory. That value corresponds to the relative offset memory of the local eCore's memory. As seen previously in this document, each eCore has 4 banks of memory, starting

from `0x0000` to `0x7fff`. Giving an offset of `0x4000` will read the data in the second bank of memory (see table 1.1).

From the target eCore, to read the value set from the other eCore, we must set a variable at the correct address. To make it easier, we declared a char array and specified its location to be on the second bank (hence starting at the address `0x4000`). We then used that array to manipulate the data. The following code shows us how we declared the array and how we read the data from the target eCore:

```
char swap[8] SECTION(".text_bank2");
swap[0] = core_num; // that data will be at 0x4000
swap[1] = ...; // that data will be at 0x4001
```

The section used in our code (text_bank2) is defined in the LDF.

2.2.1 Transfers times using different methods

We used `e_read` and `e_write` to communicate between eCores. But there is two other options. In the following example, we will set each eCore to read it's neighbor memory and measure the time spent for each options. The three options are:

- Using the function `e_read` provided by the SDK (as seen previously)
- Directly writing using the global address
- Using DMA

According to the SDK reference, the function `e_read` copy bytes of data from a remote source to a destination source [2]. In our case, the destination is given by the coordinates of the eCore. Our test uses the timer to count the number of clocks while we repeatedly perform the read operation:

```
e_ctimer_set(E_CTIMER_0, E_CTIMER_MAX);
e_ctimer_start(E_CTIMER_0, E_CTIMER_CLK);
unsigned time_e = e_ctimer_get(E_CTIMER_0);
for(i=0; i < 4000; i++) {
  // Using a dma transfer
  e_dma_copy(&neighbor_status, neighbor_status_pointer,
             sizeof(char));
}
unsigned time_s = e_ctimer_get(E_CTIMER_0);
e_ctimer_stop(E_CTIMER_0);
unsigned clocks = time_e - time_s;
```

Directly writing to the global address involves only getting the neighbor's buffer pointer and writing to that pointer. For the DMA, we will use a function provided by the SDK.

In our test, we will only transfer the smallest amount of data possible: an 8 bit char.

Figure 2.3 and 2.4 summarizes the result and shows that, for very small data, DMA transfer is not efficient, while direct addressing would be the most efficient method. The result are linear and constant over time because we always use the same data size, only changing the number of times we transfer it.

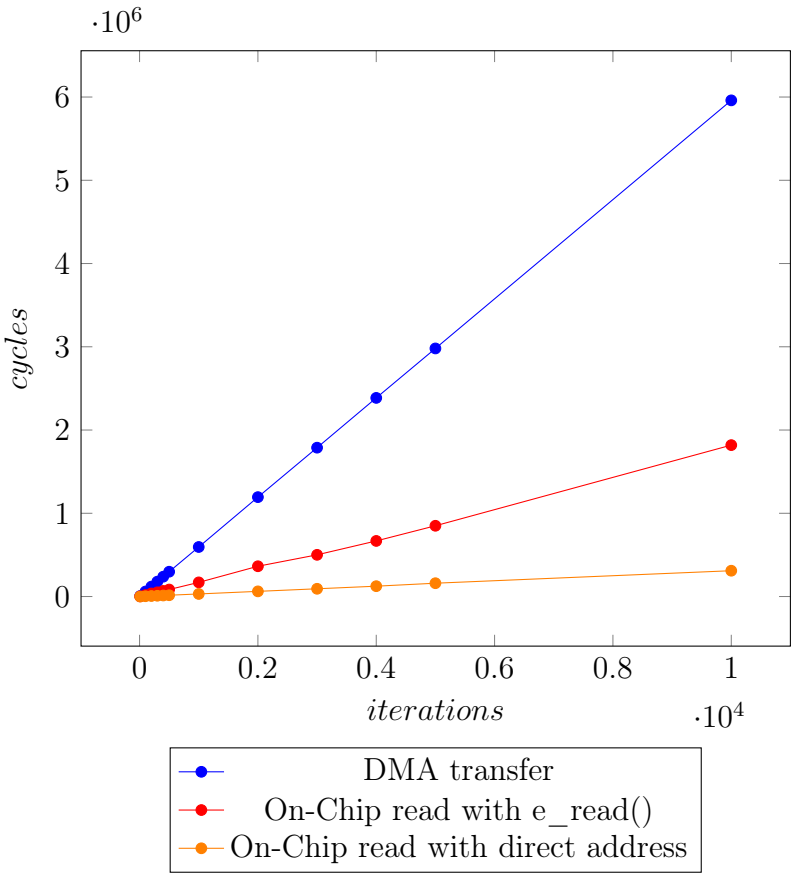

Figure 2.3: Performance measures based on number of cycles executed for a given number of iterations transferring 8 bits of data

It is important to note that we were only interested by measuring the

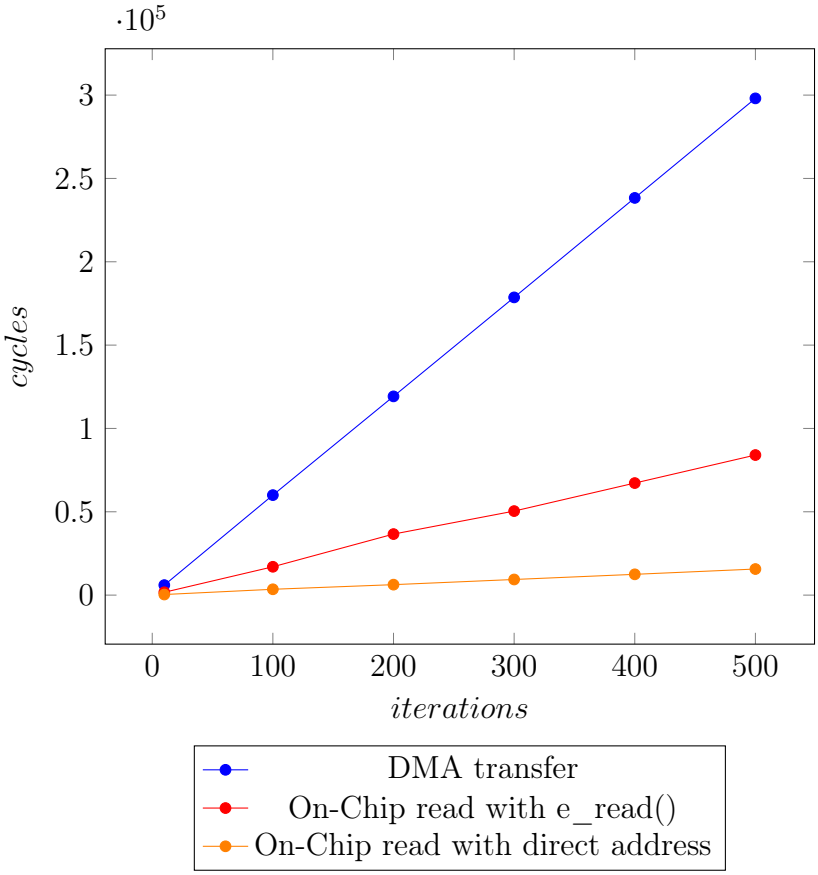

Figure 2.4: Performance measures based on number of cycles executed for a given number of iterations transferring 8 bits of data, zoomed from 0 to 500 iterations

transfer of a small amount of data. Using the DMA is faster for bigger data size [12].

2.3 Example: Transfer between host and eCores

Examples available on GitHub[10]: *simple-epiphany/data-from-host-to-epiphany*, *simple-epiphany/data-from-host-to-eCore*.

The goal we want to reach is creating a shared space from the host, then filling it with data to be used by eCores.

Figure 2.5 illustrates the process of how we will transfer data from the host to eCores (version a), or an individual eCore (version b). We see that we can either use a shared memory buffer to set a memory available for every eCores, or we can directly write to an eCore's local memory. For the version *a*, the steps are:

1a. allocating the shared space (`e_alloc`) from the host

2a. filling the shared space (`e_write`) from the host

3a. loading eCores and make them use the shared space

For the version *b* the steps are simpler because we avoid the need of a shared memory buffer by directly writing to an eCore's local memory (*1b*). Then, an eCore can use it's local memory to read data set from the host (2b).

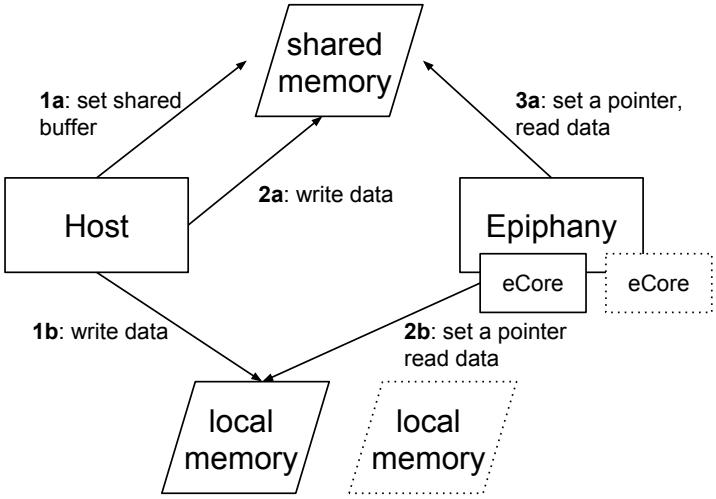

Figure 2.5: Simplistic scheme of transferring data the host to eCores, or an individual eCore

In this example, we will use 2 int arrays of 200 plus an array to store a result. Allocating memory and filling a shared buffer is not a tough task once we know how to use `e_alloc` and `e_write`. In the following code, we show the allocating and writing part:

```
e_mem_t shared_result;
e_mem_t shared_x;
```

```
e_mem_t shared_y;

e_alloc(&shared_result, BUFOFFSET, ncores*sizeof(float));
e_alloc(&shared_x, BUFOFFSET + ncores*sizeof(float),
                                 200*sizeof(float));
e_alloc(&shared_y, BUFOFFSET + ncores*sizeof(float)
                             + 200*sizeof(float),
                                 200*sizeof(float));

float y[200];
float x[200];
for(i=0; i<200; i++) y[i] = x[i] = i;
int status = e_write(&shared_x, 0, 0, 0x0, y,
                     200*sizeof(float));
printf("Status of shared_x writing: %i\n", status);
status = e_write(&shared_y, 0, 0, 0x0, x,
                   200*sizeof(float));
printf("Status of shared_y writing: %i\n", status);
```

To get those buffers from the eCore, one has just to set pointers to the right addresses. Getting the right addresses implies that we know exactly where memory is mapped. As seen in the first part of this document (section 1.3), the linker description file (LDF) gives us that information. In our case, shared memory starts at `0x8f00'0000`. Hence, we get our buffers starting with this base address plus an offset, corresponding each time to the cumulated size of the previous buffers. The following code shows the part on the eCores' source code where buffers are retrieved by setting pointers to corresponding addresses:

```
volatile float *result;
volatile float *shared_x;
volatile float *shared_y;

result   = (volatile float *) (0x8f000000 + 0x4*num);
shared_x = (volatile float *) (0x8f000000 + 16*sizeof(float)
   );
shared_y = (volatile float *) (0x8f000000 + 16*sizeof(float)
                                          + 200*sizeof(float
                                             ));
```

We can then use those buffers. In the following example, we read values from the x and y arrays then write a result to the result buffer. That result will tell us if the process successfully worked, because we will be able to read it from the host part:

```
        *result = shared_matrix1[num] + shared_matrix2[num]
                                      + num / 10.0;
```

Another way to write data from the host to the Epiphany is directly writing to an eCore's local memory. That method also uses the `e_write` method. But, instead of giving as first parameter a shared_buffer, we pass an `epiphany_t` object, which represents a workgroup of eCores (see section 1.5). We can then specify the eCore's coordinates, relative to the workgroup.

The following code shows how a data is directly sent to an eCore's local memory:

```
// Here we can select which eCore to unlock (eCore(0;1) there)
int ok = 1;
e_write(&dev, 0, 1, 0x4000, &ok, sizeof(int));
```

We note that we gave `0x4000` as the offset memory. That value corresponds to the relative offset memory of the local eCore's memory. As seen previously in this document, each eCore has 4 banks of memory, starting from `0x0000` to `0x7fff`. Giving an offset of `0x4000` will write the data to the second bank of memory (see table 1.1).

From the eCore, to read the values set from the host, we had to set a variable at the correct address. To make it easier, we declared a char array and specified it's location to be on the second bank of local memory (hence starting at the address `0x4000`). We then used that array to place or read the data. The following code shows how we declared the array and how we placed the data:

```
        volatile int instructions[8] SECTION(".text_bank2");
            // 0x4000
        // instruction[0] is at 0x4000
        // instruction[1] is at 0x4004
        int result = instructions[0]; // Data from the host
```

2.4 Benchmark

The point of this part is to implement the linpack benchmark test.Taking the existing c implementation, we will first simply run it only on the host part. We will then try to improve it by including the Epiphany in the process, using the eCores. We should be able to measure a better performance between the host only implementation and the one using the Epiphany.

The linpack works nearly out-of-the-box and only requires to adapt the `second` method, which is used to measure the performance. The result we obtained was approximately 120 MFlops, which is what we could expect from an ARM-A9 processor[11]. We will now look at how we could include the Epiphany in the process.

Using linpack directly on the Epiphany won't work due to memory capacity (see figue 2.6). What we will do is delegate jobs from the host to the Epiphany. To do so, we will set some data in the global memory, thus being available to the eCores and some status variables to know which eCores are already working or not. The host will have to poll each eCores' statuses, once it gets a free eCore set to it some specific data required to compute and then tel the eCore to do the work using global memory with it's specific data received from the host. Figure 2.7 describes how the host part will interact. As long as there is still some work to do (more iterations), the host will look for a free eCore, give to it some instructions and then release it.

```
... ld: clinpack.elf section `.bss' will not fit in region `
    INTERNAL_RAM'
... ld: region `INTERNAL_RAM' overflowed by 318544 bytes
```

Figure 2.6: Memory capacity exceeded if using c linpack on the Epiphany

On the Epiphany part (see figure 2.8), each eCore waits to be released by the host, compute what is needed using global data and specific ones (instructions), write the result and loops again. It is stopped only when application exits.

Using that process works for a small number of iterations. We implemented the concept inside a loop called once. But no relevant changes were observed since the section optimized did not represent a significant part of the benchmark. We had the same performance.

When profiling the linpack implementation (using `gprof`), we found out that approximately 88% of the total running time spent by the benchmark is in one function (see figure 2.9). That function (`daxpy`) makes the addition of an array plus an array multiplied by a constant. A more precise measure (using `gcov`), to see where the critical section is, showed us that the most used line within that function is called 69'679'500 times (see figure 2.10). So we will be focusing on optimizing this particular part of the code, which is the following code:

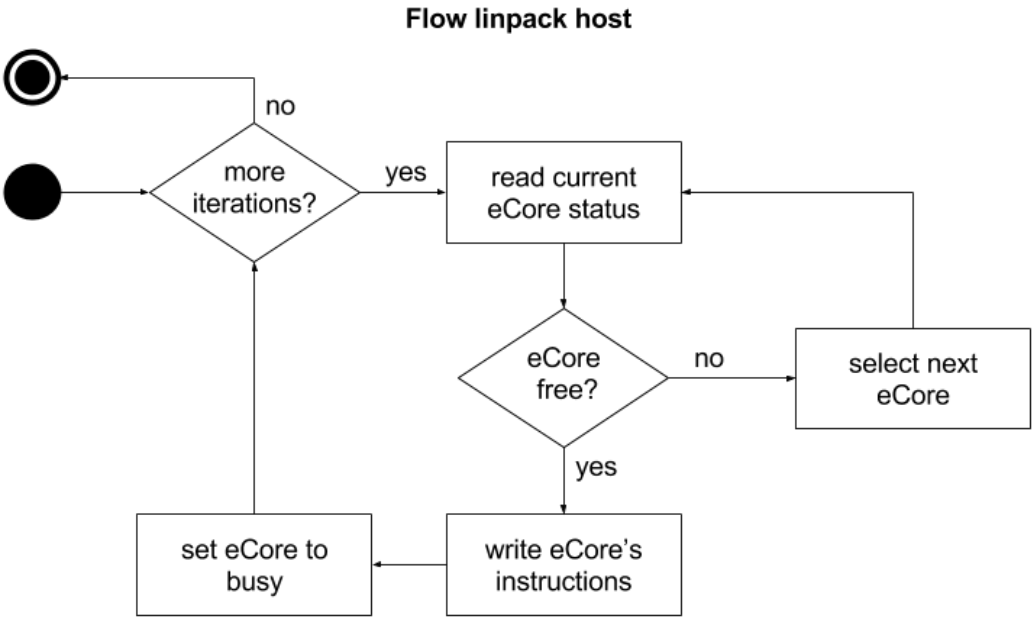

Figure 2.7: Activity diagram on the host side

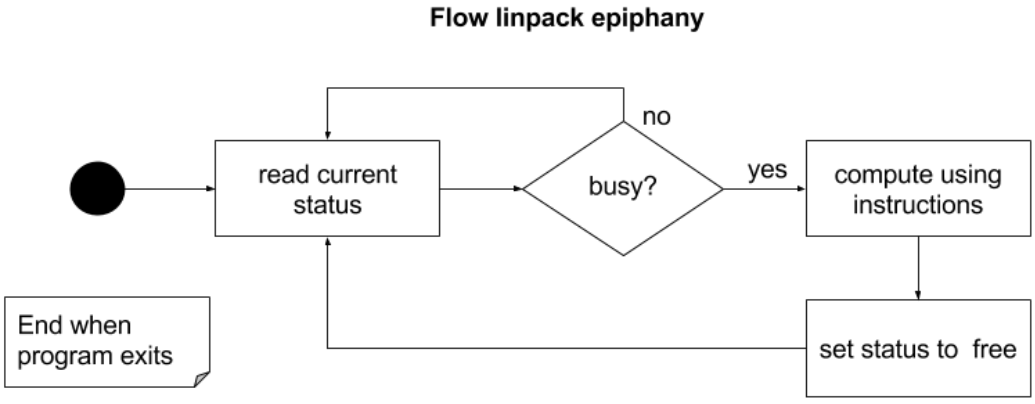

Figure 2.8: Activity diagram on the Epiphany side

```
for (i = 0;i < n; i++) {
  dy[i] = dy[i] + da*dx[i];
}
```

Using the process we described previously (fig 2.7 and fig 2.8), we faced a critical deadlock while executing the program. After a while, all eCores were

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 88.49      1.23     1.23  1060694     1.16     1.16  daxpy
  7.91      1.34     0.11                             matgen
  2.88      1.38     0.04                             dgefa
  0.72      1.39     0.01    20394     0.49     0.49  dscal
  0.00      1.39     0.00    20394     0.00     0.00  idamax
```

Figure 2.9: Results of the c linpack profilling using gprof

```
69679500:  572:        for (i = 0;i < n; i++) {
69679500:  573:                dy[i] = dy[i] + da*dx[i];
        -:  574:        }
```

Figure 2.10: c linpack Line profiling of the most used line using gcov

in a busy state and didn't change their state ever.

Trying to solve the problem, we saw that it never happened after the same number of iterations. To get a better idea of what could cause the deadlock, we made a merged activity diagram showing data access (figure 2.11). It seemed that status variable was read both at the same time by the Epiphany and the host side, it could have been the source of our deadlock.

From figure 2.11, we developed another activity diagram preventing the program accessing the same variable at the same time (figure 2.12). That way, we would use shared memory and local eCores' memory. It would prevent the host and Epiphany side polling the same variables at the same time.

2.4.1 When hardware meets software

Before implementing our second activity diagram (fig 2.12) and after some time testing some possibilities of what could cause our deadlock, we observed a curious fact: our program always took more time to come to a deadlock when it was used after a time of inactivity. It is shown in graph 2.13. Based on this observation, we suspected temperature to be involved in our problem. After more precise measures, it came that temperature was definitely the cause of our deadlock. As shown in graph 2.14, above 69°C, system became

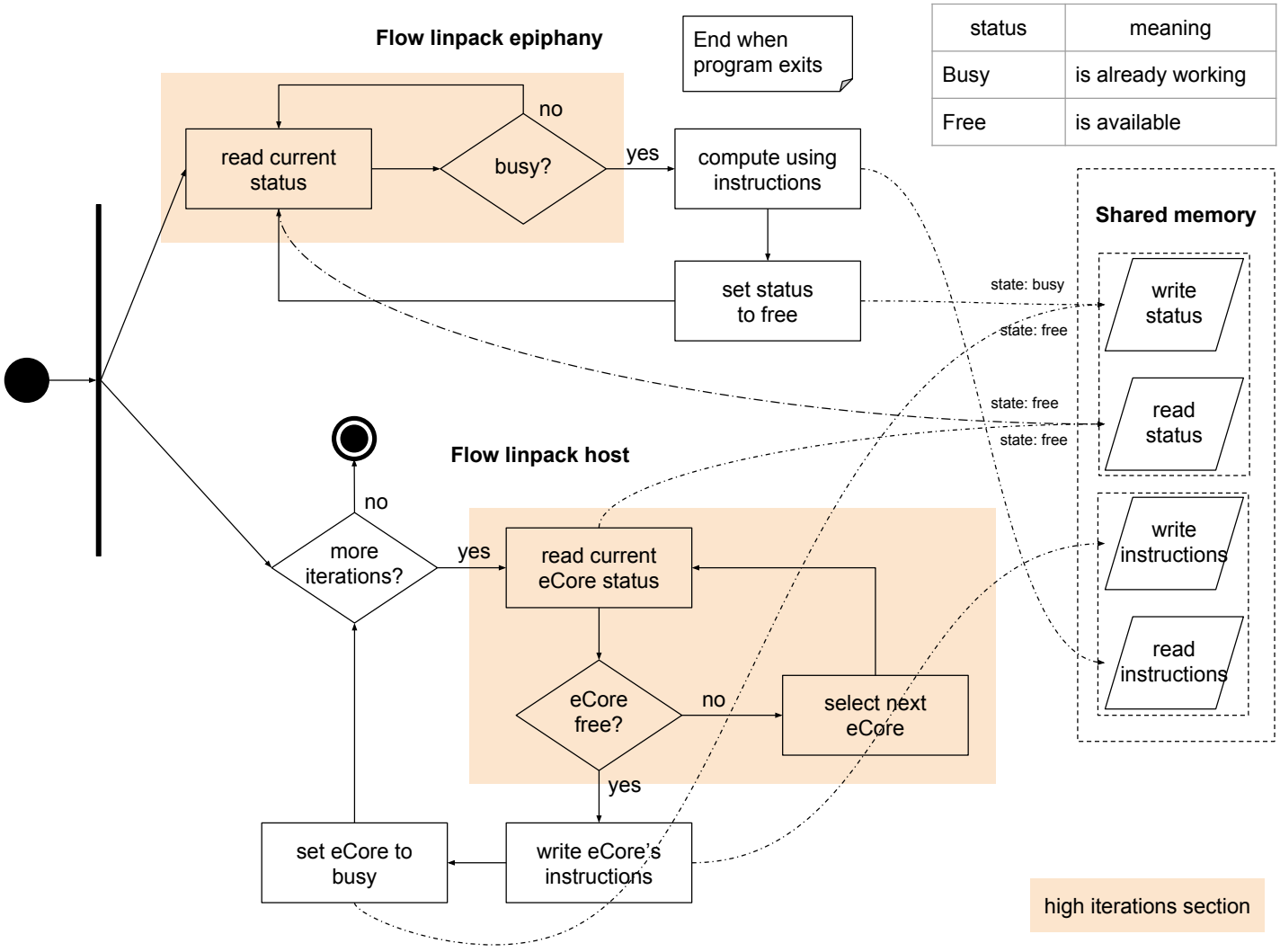

Figure 2.11: General activity diagram

instable and gave unpredictable results. Looking at adapteva's hardware github repos [3], we found out that an issue warns us about the parallella becoming unstable under unless cooled properly[5]. This new factor will then be watched in our future implementation. Graph 2.13 and 2.14 show number of iterations and temperature after and before a launch of our program. We measured temperature of the parallella, then we launched our program and measured temperature reached when deadlock occurred as well as number

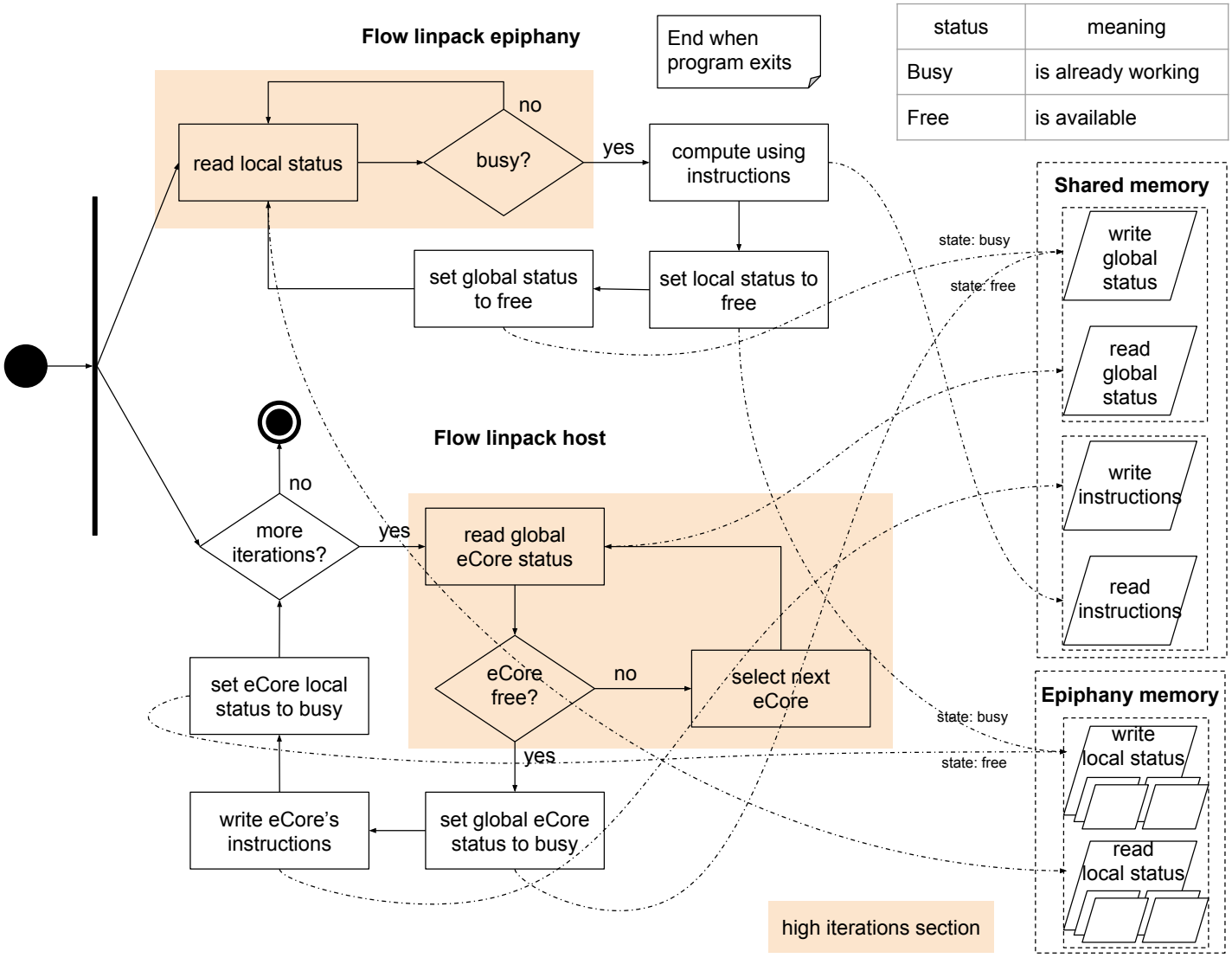

Figure 2.12: General activity diagram second version using eCores' local memory

of iterations executed. It is not surprising to observe that if the parallella is cooler, number of iteration before deadlock increases accordingly (graph 2.13). Deadlocks in our measures start from 69.2°C to 70.6°C (graph 2.14. Measuring a pertinent range of temperature was a tough task because, after

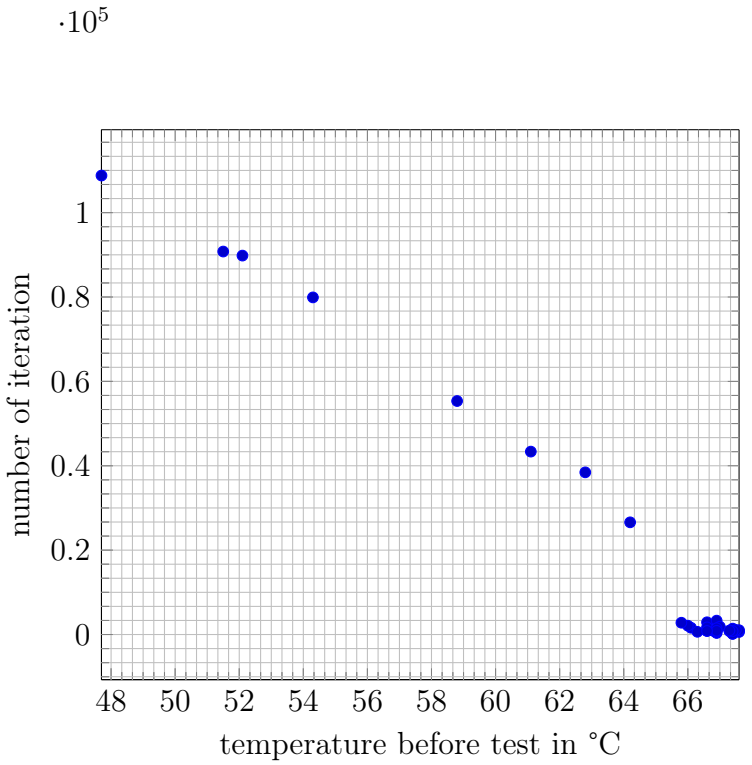

Figure 2.13: Temperature before test vs the number of iterations reached just before deadlock

each test, the parallella was at it's warmest. We had to cool the heatsink, wait, or use a fan. In some cases, the parallella could have been harder to warm because we would have cooled the heatsink, thus taking more time and giving more iterations before coming to a deadlock.

Leaving from this observation, we will now be more careful to properly cool the parallella by adding a fan or placing it in a cooler area. We leave the final implementation of the linpack for the next chapter.

2.5 Conclusion

We learnt in this chapter how to transfer data among the host and the eCores. If we understand well how memory is mapped and how the SDKs provided by Adapteva work, it is not a tough task. Knowing how data transfers

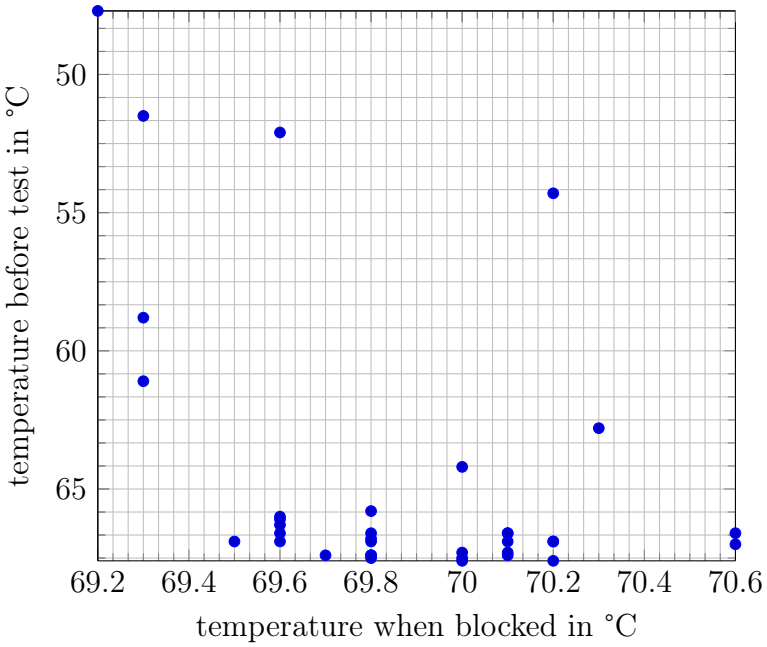

Figure 2.14: Temperature reached when deadlock occurred along with temperature when started

work is essential to start developing on the parallella, it will allow us to properly make eCores communicating among them and with the host. We also reached the physical limits of the parallella, either by trying to fit the linpack implementation on the Epiphany, reaching its memory limit, or by under cooling the parallella until it became instable. What we learnt from those examples and observations will be useful for the next chapter. Let's move to some practical examples.

Chapter 3

A practical example

Moving from all we learnt in the previous chapters, this one will discuss the implementation of the linpack benchmark and the implementation of a π approximation, using an easily parallelizable algorithm. It will be discussed the limits set by the granularity of a program and the speedup obtained with the eCores versus the host.

3.1 Linpack

The purpose of this section is to optimize the c linpack implementation with the help of the Epiphany side and its 16 cores. We will be optimizing a function computing a constant times a vector plus a vector :

```
for (i = 0;i < n; i++) {
  dy[i] = dy[i] + da*dx[i];
}
```

We will attempt to take the model (see fig 2.11) discussed in the previous chapter and improve it, especially the Epiphany part. The aim is to beat 120MFlop obtained by the host part only.

The first results obtained with the help of the Epiphany side are much worse than those obtained with the host part only.

To communicate with the Epiphany side, we need to use shared buffer as shown in section 2.3. In our case - constant times a vector plus a vector - it implies declaring and using 3 shared buffers :

- Vector[n], array of size n

- Vector[n], array of size n
- C, a constant

Those buffers are to be initialized each time the function is called :

```
status = e_write(&shareddy,  0, 0, 0x0, dy, n*sizeof(REAL));
printf("[info] Status of shareddy writing: %i\n", status);
status = e_write(&sharedda, 0, 0, 0x0, &da,   sizeof(REAL));
printf("[info] Status of sharedda writing: %i\n", status);
status = e_write(&shareddx, 0, 0, 0x0, dx, n*sizeof(REAL));
printf("[info] Status of shareddx writing: %i\n", status);
```

Unfortunately for us, we measured that the only fact of initializing those buffers decreases performances by more than 5 times comparing to the host only implementation. Meaning that doing better than the host implementation is impossible with shared buffers. Figure 3.1 shows metrics of KFlops measured with only the host side and with the buffer declaration.

As shown previously in figure 2.9, the method we are working on is called 1060694 times. Hence, calling so often the buffers initialization is not reasonable and will never make better results than the host implementation. In a more general way, our program offers high concurrency and should have a coarse granularity to avoid excessive concurrency [9]. But the way we sent small operations to the Epiphany offers too much granularity by it's executing time and the number of time it is called comparing to the total executing time. Meaning that, in this particular case, transfer times are too important. In our case, we will not raise granularity since it would require changing the way linpack uses `daxpy`.

Moving from this observation, we have three options :

- Finding a way not to to initialize buffers in the function
- Finding a way not to use shared buffers
- Trying to raise the granularity by changing the implementation

Trying to initialize buffers outside the `daxpy` method is not possible since each time the method is called, we use new data to be shared. We could bypass the need of using shared buffers by exploiting the eCores' local memory, but this option would make the implementation more complex and be limited by the eCores' local memory size.

Finally, as said before, we would need to change the architecture of the linpack implementation to find a way of raising the granularity.

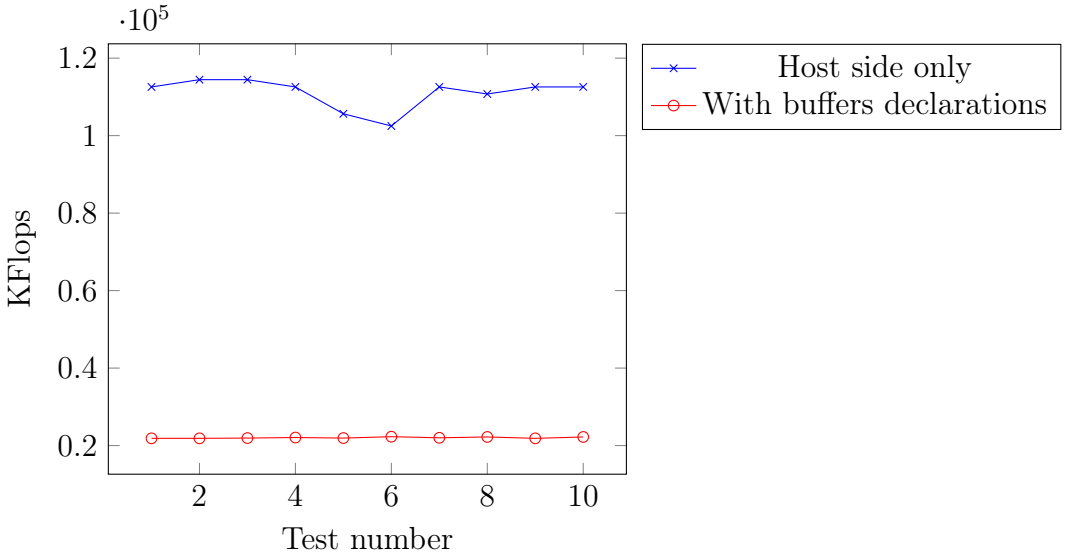

Figure 3.1: Measure of KFlops running c linpack on the host and with the shared buffers declaration. The host side is around 110MFlops, with shared buffer declaration arround 21MFlops.

3.2 π approximation

This section will be focused on implementing a concurrent program that will be approaching the PI value using machin-like formula (3.1). The advantage of this formula is to be highly parallelizable. Because it is an infinite sum, we can easily distribute part of the sum across cores.

$$\pi = 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^n}{2n+1} \tag{3.1}$$

We will be working on implementing an host only version of the PI approximation using machin-like formula, and an Epiphany version, using the eCores. The formula will be implemented as a loop calling a function which computes part of the sum. That function will then be distributed among eCores. The main loop size will represent the main number of iterations, and the part size of the sum the function will compute will be called the sub-iterations number.

Figure 3.2 is a representation of the sub and main iterations. The total number of iterations is given by the number of main iterations times

the number of sub iterations. The sub iterations will be distributed among eCores.

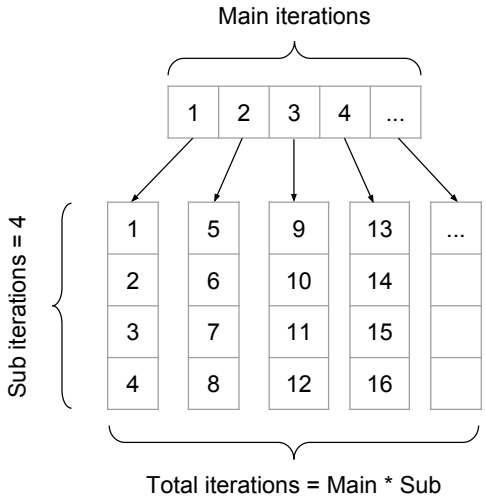

Figure 3.2: Representation of the machin-like implementation using sub and main iterations

The code remains simple, it is a loop calling N times a function which runs n iterations. Hence, it is computing the sum for $N \cdot n$ iterations. The code for the host only implementation is quite straightforward :

```
for(i = 0; i < main_iteration; i++) {
  res = res + f(i);
}

float f(unsigned x) {
  float res = 0;
  unsigned a = x * sub_iteration;
  unsigned b = a + sub_iteration;
  for(; a < b; a++) {
    res += pow(-1,a) / (2*a + 1);
  }
  return res;
}
```

To adapt our code for the Epiphany, we moved the function computing the sub iterations to the eCores' program. We then made the host managing eCores to compute the sub iterations, getting results and making the addition

of all eCores results. We used the same concept spoken in section 2.4, but using only local eCores' memory to set instruction and getting results.

The first measures we made was comparing execution times between the host implementation and the Epiphany one. We also wanted to know the impact of the number of main iterations with respect to the number of sub iterations. What we did was executing our program inside two loops modifying the number of main and sub iterations. We did the same operations for the host only implementation and the Epiphany one. We then plotted the results in a 3D graph showing the main number of iterations, the sub number of iterations and the execution time.

Figures 3.3 and 3.5 show the 3D graph obtained after plotting our results. What we can note is that the variance of the main number of iterations and sub iterations for the same total number of iterations does not affect the execution time. Figure 3.3 compares the two implementations, the host one is on top. We see that the Epiphany implementation is about two times faster. Figure 3.4 gives a better representation of the performance, giving, for the two implementations, the total number of iteration with the execution time. Figure 3.5 is the top view of the 3D graph for each implementations. We plotted on top of it curves showing values for $N * n$ (number of iterations times number of sub iterations is constant), showing that for the same number of iterations, the variance of N and n gives the same execution time.

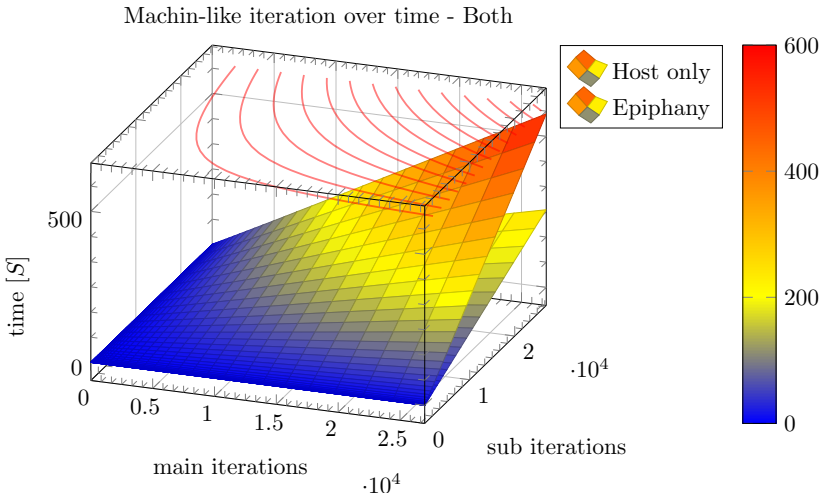

Figure 3.3: Comparing time execution of host only and Epiphany computation of pi using machin-like method

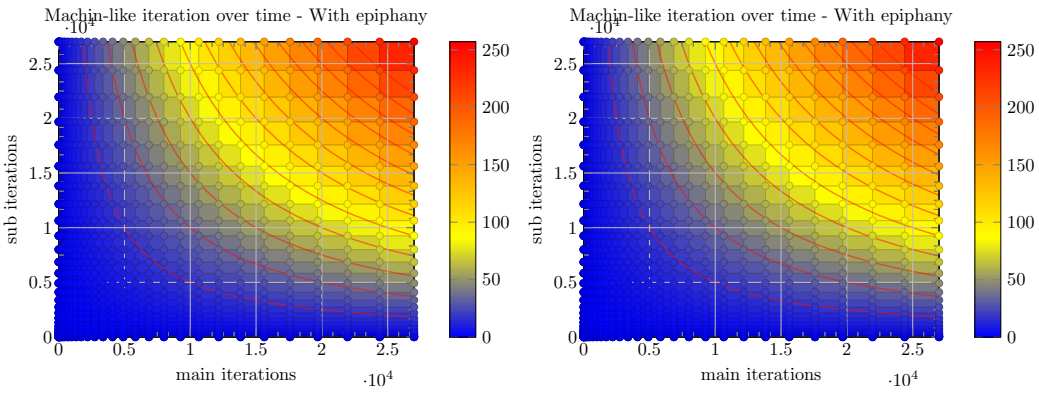

Figure 3.4: Top view of time execution with respect to iterations number

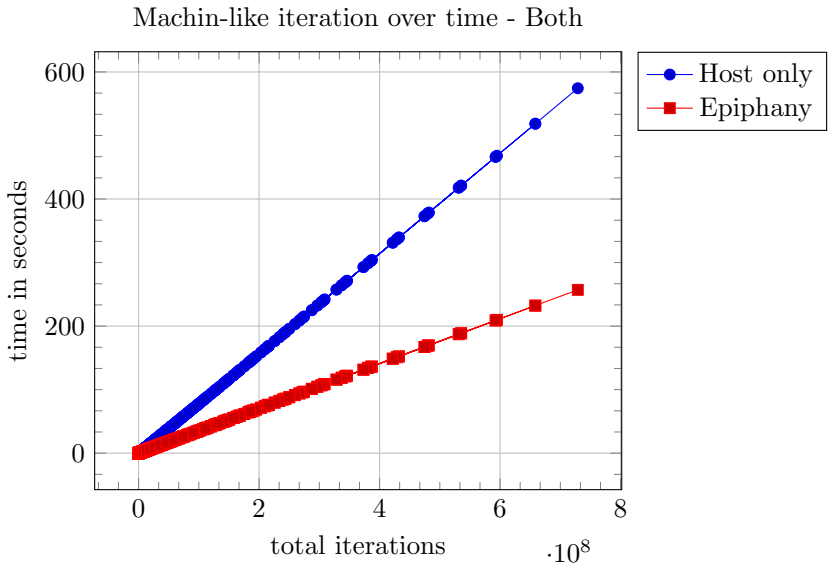

Figure 3.5: Comparing time execution of host only and Epiphany computation of pi using machin-like method

Our second measure will be the speedup obtained by using the Epiphany comparing to using only the host with its ARM-A9 processor. For the test, we will need a constant number of iterations, M, which corresponds to $M = N \cdot n$ (N is the number main of iterations, n the number of sub iterations). To compute the speedup, we will run our program with different number of cores. Then, we will compare the execution time when using one core with

the time obtained with one to sixteen cores (eCores).

To keep the same number of iterations, we will use those formulas :

$$M = N \cdot n$$
$$n = \frac{M}{k \cdot nb_{cores}}$$
$$N = k \cdot nb_{cores}$$

with M the total number of iterations, N the number of main iterations, n the number of sub iterations, nb_{cores} the number of cores and k a factor dividing the number of sub iterations. In our measures, we chose a constant M which is divisible by the total number of cores and a constant k. We then ran the program making the number of cores varying from 1 to 16 (the total number of eCores). With the time obtained, we used it to divide the reference time, taken as the time used by the host only (hence one core). This time is taken as the *time to beat*.

Figure 3.6 shows the speedup of the Epiphany according to the host. We see that to obtain the same performance of the host, we need to use seven eCores and the maximum speedup using all the eCores gives an execution time 2.24 times faster than the host only.

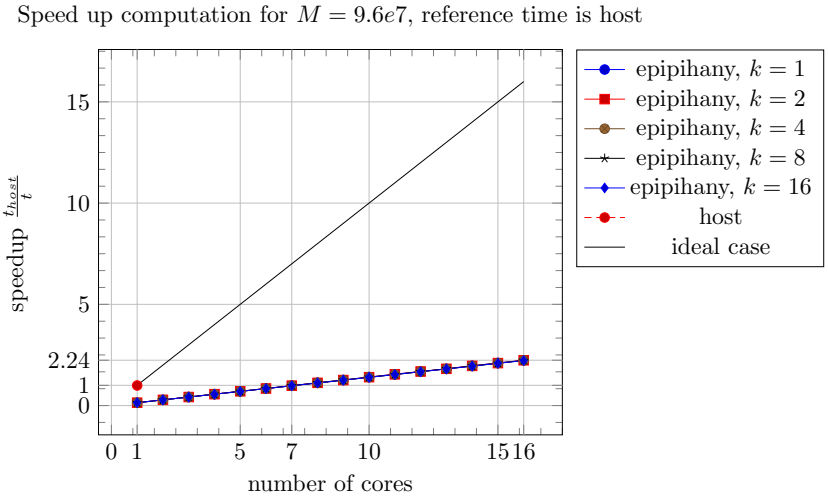

Figure 3.6: Speed up results, the reference time is the host

In section 1.2, we saw that the host side runs an ARM-A9 at 866MHz,

while each eCore runs at 1GHz. So even using a single eCore against the host should be faster, but our measures shows that a single eCore is seven times slower than the host. In fact, the documentation provided by Adapteva[1][6] tells us that the 1GHz stands for the operating frequency of the eCores' mesh network. But, in out test, we didn't use that mesh network at all. Because each jobs was dispatched from the host to each individual eCore, we didn't benefit from the mesh network provided by the Epiphany architecture. Hence, our results are reasonable given the fact that we didn't use all the capacities of the Epiphany architecture.

Taking a single eCore as the reference time gives a linear growing speedup following nearly the ideal case. The figure 3.7 shows that our problem is well distributed among eCores and that the parallel portion of our program is near 100%. In this case, transfer times are negligible.

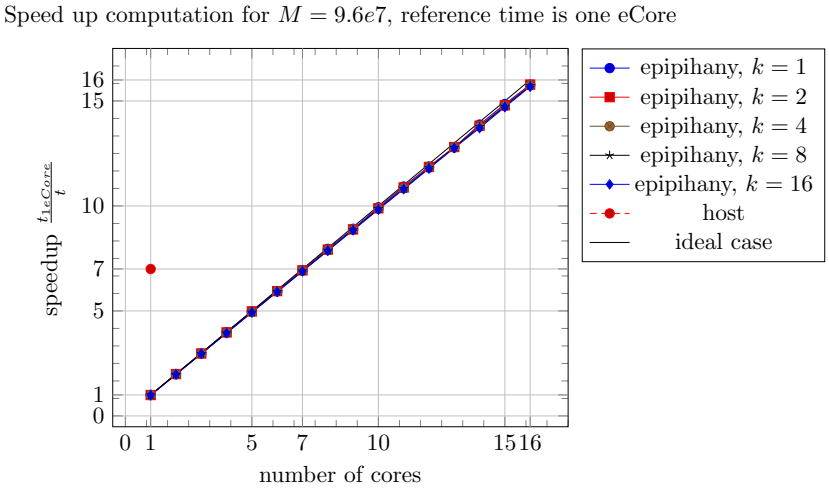

Figure 3.7: Speed up results, the reference time is one eCore

3.3 Conclusion

In this chapter, we tried to improve the linpack implementation without any success. What we learnt from that was an important fact: concurrent programming cannot improve a program at a too small granularity level depending on its transfer times. In our case, communication times was far over the execution time of the code we wanted to improve. We would need

to re-think the implementation of the linpack program so as to raise the granularity.

Moving from the linpack, we took this opportunity to implement an approximation of π that would offer an easy concurrent implementation. We found out that, for this problem, the host is seven times faster than a single eCore, and using sixteen eCore against the host is only 2.24 times faster. That result is far bellow what we could expect. One explanation would be the way jobs are distributed, without using the Epiphany mesh network at all. The speedup taking a single eCore as a reference time grows linearly, nearly following the ideal case. This show us that our problem is well distributed and communication times doesn't affect much the speed. It would have been interesting to test with more than sixteen cores, for example using many parallella boards connected together.

Chapter 4

Final thoughts

4.1 Conclusion

We have seen an overview of the parallella architecture along with its main features, some practical examples and made some measures so as to study its potential. Although we didn't cover everything the parallella has to offer, we still can draw some conclusions.

The parallella is still in a growing stage. Released in 2012, it has not much evolved since today and there is still work to be done. Documentation and tools provided by Adapteva suffer from a lack of updates. What we missed the most was a way to debug program hosted on the Epiphany. Even performing a simple output print wasn't possible, which made testing and debugging harder. Another missing feature would be an interrupt line between the host and the Epiphany side. Without it, we were forced to use polling so as to fetch information and states of the Epiphany from the host.

The parallella is good only on specific cases. With its very limited memory and the cost of using shared memory, we saw that we couldn't adapt every concurrent program to the Epiphany. We had to consider the program size as well as the shared memory needed, as we saw it wasn't possible to adapt the clinpack implementation due to those restrictions. With the π implementation, we saw that even with minimal transfers time, the program wasn't as fast as we would think. The reason was that the main advantage of the parallella is its mesh network of eCores providing high speed memory transfers, which we didn't use at all. Hence developing on the parallella asks sometimes to think differently so as to exploit all its potential.

With its low price and low electricity consumption, the parallella has certainly a place on the market of distributed system. But without a strong toolchain, easy workflow and serious documentation, it will be impossible for Adapteva to find its place and gain adoption in a larger scope than only experimented embed system users.

4.2 Further work

It would be interesting to adapt the π implementation using the mesh network of eCores. The idea would be to send from the host all the job once in a single eCore, and then make eCores capable of spreading jobs among them using the mesh network. It should be the best way to exploit the capacities of the parallella.

The parallella supports the Message Passing Interface (MPI), we didn't talk about it but it would certainly be worth exploring its potential on the parallella. For example by adapting the π implementation with MPI.

Last but not least, one of the main requirement of the parallella is to be scalable to thousands of cores. There is much to do with setting up and developing on a distributed architecture using many parallellas, either connected using ethernet or using GPIO (General Purpose Input Output). We could then be able to measure the speedup with more than sixteen cores.

Bibliography

[1] Adapteva. "Epiphany architecture". In: (2013).

[2] Adapteva. "Epiphany SDK reference". In: (2013).

[3] Adapteva. *epiphany-libs*. [Online; accessed 4-january-2016]. 2015. URL: `https://github.com/parallella/parallella-hw`.

[4] Adapteva. *epiphany-libs*. [Online; accessed 1-Decembre-2015]. 2015. URL: `https://github.com/adapteva/epiphany-libs/tree/master/bsps/parallella_E16G3_1GB`.

[5] Adapteva. *epiphany-libs*. [Online; accessed 4-january-2016]. 2015. URL: `https://github.com/parallella/parallella-hw/issues/2`.

[6] Adapteva. "Parallella manual". In: (2014).

[7] Tomas Nordström Andreas Olofsson and Zain Ul-Abdin. "Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany". In: *Cornell University Librairy* (2014), p. 1.

[8] Analog Devices. "Understanding and Using Linker Description Files on SHARC® Processors". In: (2007), pp. 2–3.

[9] Hidehiko Tanaka ed. *Parallel Inference Engine - PIE*. Ohmsha, 2000, p. 107.

[10] Noémien Kocher. *Git project*. [Online; accessed 4-january-2016]. 2016. URL: `https://github.com/nkcr/parallella-computing`.

[11] Roy Longbottom. *Linpack Benchmark Results On PCs*. [Online; accessed 27-january-2016]. 2016. URL: `http://www.roylongbottom.org.uk/linpack%20results.htm`.

[12] Anish Varghese et al. "Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor". In: (2014).

Declaration of Authorship

I hereby certify that the paper I am submitting is entirely my own original work except where otherwise indicated. I am aware of the regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Place date signature

Appendix A

Source code structure

Sources involved with this projet can all be found at:

`https://github.com/nkcr/parallella-computing`

It contains three folders:

Miscellaneous: contains some tests to check deadlock on basic C operations. It was used to debug the linpack implementation.

benchmark: contains the clinpack implementation in few versions. One is the host only and the others (containing *Epiphany* in their name) are the implementations for the Epiphany. There is also the π approximation using machin-like formula in two versions, one for the host and the other for the Epiphany. Inside of them there is scripts to run measures which generate cvs files, there is also the measures used for this paper.

simple-epiphany: contains all the examples developed for the second chapter *In Deep*. All the examples of memory transfers as well as a simple game of life reside in this folder.

Each folder contains a README file which describes the content of it. Please note that some code was used to produce simple technological test and may not be properly maintained, this it is not the case of the π implementation.

Appendix B

Versions used

The distribution used for this project was:

```
$ lsb_release -a
Distributor ID: Linaro
Description:    Linaro 14.04
Release:        14.04
Codename:       trusty
```

Kernel release:

```
$ uname -r
3.14.12-parallella-xilinx-g40a90c3
```

The parallella board was a desktop version with Zynq7010. All source code was plain ANSI C, compiled using gcc version 4.8.2. The Epiphany toolchain (SDKs, e-gcc utility, ...) was of version version 4.8.2 20130729 (prerelease) (2015.1).