Extending ePython to support parallel Python interpreting across multiple interconnected Epiphany processors

Dongyu Liang

August 18, 2017

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2017
Abstract

The Epiphany is a RISC core based many-core architecture processor. In this project, an Epiphany cluster has been designed to achieve a larger-scale parallelism of the Epiphany RISC cores. An existing Python interpreter ePython has been extended to support the explicit parallel Python programming on the many-core architecture cluster. Besides, an evaluation of the extended ePython has been carried out to verified its correctness, stability and performance. The extended ePython in this project has been proven to be capable of making use of the larger amount of concurrent Epiphany cores within the cluster. The implementation details of the cluster and ePython as well as the evaluation results have been included in this report.
# Contents

1 Introduction .......................................................... 1

2 Background ............................................................. 3
   2.1 Parallella and Epiphany ........................................... 3
      2.1.1 Manycore architecture ....................................... 3
      2.1.2 Hardware platform ........................................... 3
      2.1.3 Softwares and development environment ..................... 5
   2.2 Heterogeneous cluster computing ............................... 6
      2.2.1 Heterogeneous system ......................................... 6
      2.2.2 Cluster and distributed system .............................. 7
      2.2.3 Message passing ............................................. 7
   2.3 ePython ............................................................. 8
      2.3.1 From Python to ePython ....................................... 8
      2.3.2 ePython structure ........................................... 9
      2.3.3 Local communication mechanism ............................ 11
   2.4 Challenges ......................................................... 14

3 Design and Implementation ............................................ 15
   3.1 Construction overview ........................................... 15
      3.1.1 Building a Parallella cluster ............................... 15
      3.1.2 General methodologies ...................................... 17
   3.2 Point-to-point communication .................................... 20
      3.2.1 Extending the send P2P communication call .................. 21
      3.2.2 Extending the receive P2P communication call ............... 24
      3.2.3 Extending the send-and-receive P2P communication call ... 25
   3.3 Collective communication ........................................ 27
      3.3.1 Extending the barrier synchronisation call .................. 28
      3.3.2 Extending the broadcast collective communication call .... 29
      3.3.3 Extending the reduce collective communication call ....... 30
   3.4 Miscellaneous Supports for Clusters ............................ 32
      3.4.1 Environmental native functions ............................. 32
      3.4.2 Running configurations ..................................... 32
   3.5 Trouble Shooting ................................................ 33
      3.5.1 Premature POSIX Thread termination ........................ 33
      3.5.2 Stack overflow on Epiphany ................................ 33
4 Results and Evaluation

4.1 Individual functions test .............................................. 35
  4.1.1 Environmental native functions ............................... 35
  4.1.2 Unidirectional send and receive .............................. 36
  4.1.3 Bidirectional send-and-receive ............................... 38
  4.1.4 Broadcast .......................................................... 39
  4.1.5 Reduction .......................................................... 40

4.2 Benchmark ............................................................. 41
  4.2.1 Strong scaling test ............................................... 41
  4.2.2 Weak scaling test ............................................... 43

5 Conclusions .................................................................... 45

5.1 Future work ............................................................... 46

A Python test code .......................................................... 47
  A.1 Cluster configuration test code ................................. 47
  A.2 Unidirectional send and receive test code ................. 47
  A.3 Bidirectional send-and-receive test code ................. 48
  A.4 Jacobi iteration benchmark code ............................. 48

B The prototype cluster ..................................................... 51
  B.1 Mapping of the host names ..................................... 51
  B.2 Setting the passwordless SSH ................................. 51

C Files changed in ePython source .................................... 53
List of Tables

4.1 Comparison of unidirectional P2P message passing runtime between
the local and remote communication ........................................... 38
4.2 Comparison of bidirectional P2P message passing runtime between the
local and remote communication .................................................. 39
List of Figures

2.2 The ePython architecture [21] .............................................. 9

3.1 Remote communication procedure ........................................ 18
3.2 Register for remote send ..................................................... 22
3.3 Monitor deadlock caused by mutual waiting ............................... 23
3.4 Register for remote receive .................................................. 25
3.5 Register for remote send-and-receive .................................... 26
3.6 Recursive barrier synchronisation ......................................... 28
3.7 Register for remote broadcast .............................................. 30
3.8 Register for global reduce .................................................... 31

4.1 Message passing pattern for send/recv test ............................... 37
4.2 Data swapping in send-and-recv test ..................................... 38
4.3 Collective communication runtime with varying number of e-cores ... 40
4.4 Strong scaling test on prototype cluster with Jacobi benchmark .... 41
4.5 Parallel efficiency for test on prototype cluster with Jacobi benchmark 42
4.6 Weak scaling test on prototype cluster with Jacobi benchmark .... 43
4.7 Execution time per iteration of Jacobi benchmark ...................... 44
Listings

3.1 MPI host file .................................................. 17
4.1 Environmental native functions test output .................. 35
4.2 Optional argument -d test output ............................ 36
4.3 Send and receive test output ................................. 37
4.4 Send and receive test output ................................. 38
A.1 cluster.py ..................................................... 47
A.2 Byte code representation of cluster.py ....................... 47
A.3 remote_send_recv.py ......................................... 47
A.4 remote_sendrecv.py .......................................... 48
A.5 jacobi.py ...................................................... 48
B.1 Address mapping of the Parallella nodes ..................... 51
Acknowledgements

I really appreciate the assistance provided by my supervisor, and the original writer of the ePython, Dr. Nick Brown in carrying out this fun project. Without his constructive guidance this project would not be so successful. And I am grateful to him for being tolerant of my fault and giving me enough freedom to implement my own ideas. He is the beacon of knowledge blazing out across a dark sea of my ignorance.

I would also like to thank the Adapteva Inc. for donating the Parallella boards to us, they have been used for building the prototype cluster. And I wish to thank the EPCC staffs and students for helping me learn the HPC knowledge. Finally, I wish to thank my friends, teachers and family members for their support and encouragement throughout my study.
Chapter 1

Introduction

The idea of parallel programming was introduced in the pursuit of higher performance computing. People found in the early stage of computer science that, if given enough time and memory, a computing machine can theoretically perform any tasks by serially executing instructions\[1\]. Since then, a lot of computing systems have been programmed to accomplish various kinds of tasks. Before long, the speed of serial program can no longer satisfy the demands for computation. To improve the computing efficiency, two general approaches have been taken by computer scientists. The first approach tackles execution time for a single instruction, which continues to improve the serial processing speed, whereas the second focus on the number of instructions that can be executed simultaneously.

This project is one of many efforts at carrying out the second approach, parallelisation. It is believed to be a natural and feasible solution to many computationally demanding problems for the computing itself is an artificial abstraction of the processes happen in the physical world and the real world is made up of numerous concurrent events.

In parallel programming, a task should be properly split into independent serial code sections and correctly performed by the parallel processing units. In practical terms, the code split can be explicitly instructed by many programming languages such as Fortran or C, and the computing devices that contain concurrent processing units have evolved into a huge variety of forms. The close cooperation of parallel software and computing device relies on the middleware such as compiler or interpreter to correctly translate the instructions from the programmers. This project will do some foundational work in supporting Python, which is new for parallel programming, on a self-designed device based on a newly emerged type of parallel computing architecture.

The innovative parallel computing device designed in this project is an interconnected many-core processor system. This system contains multiple Epiphany processors as the basic units. Each Epiphany processor contains 16 individual cores to concurrently execute users’ explicit parallel code. With many advanced features, the Epiphany and other many-core processors of its kind have shown a great potential in many scenarios. However, the Epiphany processor itself is using a completely new architecture and it
is notoriously difficult to program. Nevertheless, this project took up the challenge of developing on it. Furthermore, The primary goal is to make it reach a larger scale of parallelisation than the single Epiphany so as to support more complicated parallel programming patterns. The main uses of the system are fast prototyping and teaching, hence in its development the usability concern takes priority over the performance. In order to free the users from the low level programming with the natively supported C on the Epiphany, Python was chosen to be the language to program this multi Epiphany system and accordingly a parallel Python interpreter has been tailored for it. The Python interpreter implemented for this system is an extended version of ePython, which was originally a successful C based Python interpreter for programming the single Epiphany. Because of the programming difficulty, all the previous attempts to build the Internet-connected Epiphany cluster have failed. The technology developed in this project is the first successful attempt in the history for supporting parallel programming on multiple interconnected Epiphanies transparently.

The report is structured as follows.

In the first part, The background chapter explains the context of the project proposal and introduces the existing technologies used in the project. In the second part, the design and implementation chapter relates the building of the Epiphany cluster and the extending work on ePython to make it capable of managing all the Epiphany cores in the Epiphany network with message passing model. In the third part, the results and evaluation chapter presents the test results of several individual and comprehensive tests conducted for examining and evaluating the correctness, stability and performance of the system. Lastly, the conclusions chapter discusses the outcome of this project and points out the roadmap for its future development work.

The source code of the extended ePython can be found at the online git repository[2] under the dev branch, current commit header is c85a848.
Chapter 2

Background

2.1 Parallella and Epiphany

2.1.1 Manycore architecture

The manycore architecture originated from the multicore architecture. It was designed to bypass the cache coherency problem in dealing with the scaling of the multicore processor. The manycore architecture integrates complete processing units (cores) into a single computing chip. Each core has its own memory and connects to other cores through a fast network-on-chip (NoC). These cores are usually simple and identical (at current stage). Therefore the advantage of the manycore architecture is overall energy efficiency rather than single thread performance.

The manycore architecture has been widely endorsed in parallel computing systems, such as the SW26010 used in thus far the fastest supercomputer Sunway TaihuLight[3] which comprises 465-core processors, the GPUs which are essentially composed of many vector processing cores and the Intel Xeon Phi which integrates many multicore processors into a chip. The Epiphany manycore architecture, is one implementation of the various categories of manycore architectures. Different from other microcore architecture processors, it is the Reduced Instruction Set Computer (RISC)[4] cores that the Epiphany architecture is using as its basic component.

2.1.2 Hardware platform

The Epiphany processor which implements the Epiphany manycore architecture is an open-source system-on-chip designed by Adapteva Inc. Specifically, the Epiphany chip (E16G301) used in this project is a 16-core coprocessor hosted by a credit-card-sized computer Parallella (Microserver P1600). As shown in figure 2.1, the Parallella also has a dual-core ARM CPU as its main processor and provides several ports for extending.
The Parallella was launched in 2012\textsuperscript{[5]}. With more than 10,000 shipped worldwide it has gained popularity among HPC educators and parallel programming hobbyists due to its high level explicit parallelism and low price. Apart from its capability of fast parallel code prototyping, the outstanding energy efficiency also makes it promising in the application of the next generation exa-scale computing and embedded devices. The Epiphany manycore architecture used on it can achieve up to 50 GFlops/Watt in 28 nm technology\textsuperscript{[5]}. Merely for reference, various supporting facilities included, the most energy efficient supercomputer in the Green500 list Tsubame 3.0 can only reach 14.110 GFlops/Watt by using the NVIDIA Tesla P100 SXM2\textsuperscript{[3]}.

![The Parallella board and the Epiphany chip](image)

Figure 2.1: The Parallella board and the Epiphany chip \textsuperscript{[6]}

The central processor Zynq-7000 (equipped with dual-core ARM Cortex-A9 processor) in Parallella board operates on a 1GB 32-bit wide DDR3L SDRAM\textsuperscript{[7]}, 32 MB of which are shared memory with Epiphany. The Parallella uses a single SD card as its storage and supports 10/100/1000 Mb/s Ethernet.

As per figure 2.1, the 16 cores on Epiphany chip are arranged in a 4x4 array. Each Epiphany core (e-core) consists of

1. A superscalar, floating-point RISC CPU that is capable of execute two floating point operations and a 64-bit memory load operation on every clock cycle\textsuperscript{[8]}
2. 32kB of local memory that provides 32 Bytes/cycle of sustained bandwidth\textsuperscript{[8]}
3. Multicore communication infrastructure to communicate with host CPU and other Epiphany cores

All e-cores can directly access other e-cores’ local memory via a low-latency, zero startup overhead NoC eMesh\textsuperscript{[8]}. In comparison with the inter-e-core communication using the fast eMesh, it is much slower for the host CPU to access the local memory of the e-core and for the e-cores to access the shared memory with the host CPU. The Epiphany does not have any input and output (IO) ports to users, the only way for the e-cores to interact with the outside world is through the shared memory with the host ARM CPU.
2.1.3 Softwares and development environment

The operating system on the Parallella host is a special Linux distribution Parabuntu. It is a modified version of Ubuntu by the manufacturer Adapteva based on the Ubuntu 15.04\textsuperscript{[9]}. Therefore the software development on the ARM CPU is same as that on the other computers running Linux.

For the software development on the Epiphany, Adapteva also provides an official SDK\textsuperscript{[10]}, the Epiphany Software Development Kit (eSDK). The eSDK supports users to write ANSI-C code for the Epiphany without language extensions. Internal to the eSDK, the C/C++ code written for the Epiphany will be compiled by e-gcc to generate an e-core executable file with .srec extension. The e-gcc runs on a lightweight kernel that supports the Epiphany runtime library (e-lib), the standard C library (newlib) and the math library (libm). The compiled device-executable files will then be loaded by the host program using the Epiphany Hardware Abstraction Layer (eHal) API to every e-core and start executing. However, these libraries are low-level and users still need to deal with the hardware specifically, which has become one major reason for many never properly used Parallellas.

On the base of the eSDK, several third-party technologies for Epiphany programming have also been developed. For instance, the eBSP by Coduin\textsuperscript{[11]} for Bulk Synchronous Parallel (BSP), the OMPi\textsuperscript{[12]} for OpenMP and the CO-PRocessing THReads (COPRTHR) SDK\textsuperscript{[13]} for OpenCL and OpenSHMEM. Nevertheless, programming on the Epiphany are still difficult and time-consuming even for those experienced parallel programmers, because most of these newly developed technologies are still involving many low-level programming and are usually bug-prone. Moreover these technologies only support single Epiphany programming. Among these programming technologies, the ePython, based on which this project was carried out, was one that supports the explicit parallel Python programming with message passing model on the single Epiphany.

The Supercomputer.io project

In the exploration of the full potential of Parallella beyond its regular usage, Adapteva launched Supercomputer.io project in 2015\textsuperscript{[14]}. By trying to connect a large number of volunteering Parallellas via the Internet, it aimed to build a Parallella cluster that could handle the jobs only supercomputers can do. Some passionate users around the world connected their idle Parallellas with great interest and vitality but many of them got no activities processed on their Parallellas. The project lasted arduously for about one year but the cluster barely finished any job. Because of the unexpected programming difficulty of the interconnected Parallellas, the project had to be aborted in 2016.

The unsuccessful outcome aside, the Supercomputer.io project has certainly advanced a remarkable concept. Inspired by this, the project was initially attempted to realise a similar idea and turned out to be a successful small scale resurrection of the deceased Supercomputer.io project.
2.2 Heterogeneous cluster computing

The Parallella itself is a heterogeneous system, and the way that the Supercomputer.io project was trying to organise those distributed Parallella nodes makes the imagined Parallella supercomputer a effectively heterogeneous computer cluster. Although, as for this project, the Python programmers will treat the Epiphanies as a homogeneous system, the underlying interpreter ePython is actually a software operating on many dissimilar cores carrying out heterogeneous computing. Hence before carrying on this project and helping the Supercomputer.io project out of its predicament, we need to delve more into the support technologies.

2.2.1 Heterogeneous system

Heterogeneous computing systems are systems that uses multiple dissimilar cores or processors. All cores in a heterogeneous system cooperates to accomplish a single complex task. The cores are designed differently because they are supposed to handle different types of subtasks.

The heterogeneous system area has been under developing ever since the dawn of the multicore architecture. The reason why multicore and hence the heterogeneous system emerges is that the technological limit of the single core frequency forces the scientists and engineers to find a roundabout mean to achieve higher overall processing speed. However, the ever-increasing hardware complexity coming with the development of heterogeneous systems have also increased the programming difficulty. Because the heterogeneous systems can be constructed in many drastically different ways, there is not a unified standard of programming that fits all the heterogeneous systems in general.

For a common heterogeneous system that uses GPUs as accelerators, the typical style of programming is write a single piece of code for the entire heterogeneous system, mark the parts to be executed by the accelerator, and the interface libraries such as CUDA\textsuperscript{[15]} or OpenACC\textsuperscript{[16]} will do other things for the programmer to offload the marked part outside the main processor cores. By contrast, the official eSDK provides a different developing style from the GPU programming and it is lower-level, which means the software developing on the Parallella need explicit separate code writing and compiling for the host CPU and Epiphany cores.

Concerning this project, the developing of ePython is directly based on the eSDK and the Epiphany cores were dealt with in a heterogeneous manner. Furthermore, because the structure of a higher level (i.e the Epiphany cluster) is to be formed, inside the runtime, the The e-cores will be differentiated by whether they are local or not.
2.2.2 Cluster and distributed system

In terms of the hardware organisation manner, both the aforementioned manycore and multicore systems implement the hardware parallelisation in processing unit level within a computer, where as the cluster and the distributed system can be seen as the hardware parallelisation in computing node level.

A computer cluster is a group of interconnected computers that is capable of processing a single program. The computers which individually run their own operating systems are called nodes. Normally, all the nodes within a cluster communicate with each other via a dedicated high speed network. In practice, the phrase heterogeneous cluster has double meanings, it can refer to either a cluster comprising different type of nodes or a cluster comprising homogeneous nodes but each node itself is a heterogeneous system. In this project, the cluster using Parallella as its component node is a heterogeneous cluster. Different from the simple homogeneous clusters, the main computing power of a heterogeneous cluster comes from the cores that do not run the operating system.

By contrast, the distributed system refers to a group of loosely connected computers such as Parallellas in the aforementioned Supercomputer.io project. The computers of a distributed system might be physically located in different geographical areas. In terms of tasks being handled, a distributed system does not necessarily process a single program. Although the programs on different nodes may perform a certain task, they can still be treated as individual programs. This kind of programs are called distributed program. With the use of distributed program the configuration of distributed system could be more flexible, which means the underlying hardware of a distributed system might change during the runtime of a distributed program.

As a matter of fact, the distributed system and the cluster do not have to be as different as chalk and cheese. A system used for computing can be a combination of both. The remotely located computers can also have fast network connections that allows them to behave like a densely located cluster. Thus, what really matters for the "node level" parallel computing is how those processors using individual operating systems cooperate with each other.

2.2.3 Message passing

The Message passing is not only a widely used parallel programming model but also the fundamental operations performed by processors to cooperate with each other. In message passing, each set of instructions whose execution is independent of others is called a process. Their independency means each process has its own data in its own memory and its instructions can be performed by only interacting with its own data. Processes can be executed concurrently because of their independency. They can also change their own behaviour by certain executions based on other processors’ data. In this case, they interact with each other, and the data they obtained from other processor is usually not through direct memory accessing but through explicit memory transfer.
from other processes’ physical address to its own address. The data being transferred is called the message.

Processes pass the messages through specific library functions that are able to operate the underlying hardware. Because the hardware differs from vendor to vendor, these library functions were implemented independently on different computers. After a long period of development, the interface of these functions began to be unified through the cooperation of the hardware vendors. And the standard for the message passing library interface in C and Fortran bindings came into existence. The specification for the library interface is called the Message-Passing Interface (MPI). Despite the fact that the interface has been standardised, the implementation of MPI still have many versions inherited form their earlier message passing library developers. The Open MPI\cite{17} used in this project is one of the open source MPI implementations. Its foundation was based on the merging of some early industrial and academic MPI implementations and it is now applicable to many kinds of operation systems and hardware, including the Parallella.

The major advantage of using MPI is that it is efficient and stable. Also, it can save a lot of effort from programming at network communication layer because the MPI library will cope with the underlying hardware. However, the MPI will make the program run more like on a cluster rather than a distributed machine because the hardware alteration during the running of MPI programs would be fatal.

2.3 ePython

2.3.1 From Python to ePython

Python is a high level scripting language designed for general-purpose programming, emerged in the late 1980s and firstly released in 1991\cite{18}. It is free, open-sourced and has incorporated various programming paradigms such as the structured programming and the object-oriented programming. Up to the present, Python has become vastly popular among the programmers because it places the emphasis on the simplicity and coding productivity. However, the Python language is currently not widely used in the HPC world out of its performance consideration. In fact, from a long-term point of view, the performance issue can sooner or later be remedied by the ever-growing hardware capability. And in some cases, the time of programming is more valuable than the time of executing. Hence, Python still have a great potential in the high performance computing field of the future. Most importantly, Python can be extended to support the explicit parallel programming paradigm, moreover, the feature that the Python code has a better readability allows the parallel programing to become easier for some beginners, and hence it can be used not only for the fast parallel code prototyping but also for the HPC education.

ePython is a Python interpreter specially designed for executing explicit parallel Python
code on the Epiphany processor. It is an open source software under the BSD-II license, the source tree can be found at the online git repository\[^{19}\]. The current ePython is not a full Python interpreter. Only the core Python code and part of the libraries are supported.

### 2.3.2 ePython structure

Same as other Epiphany softwares using eSDK, the ePython has two executable parts for host ARM CPU (ePython host) and for Epiphany cores (ePython device).

![The ePython architecture \[^{21}\]](image)

As shown in figure 2.2, the ePython main program starts on the host CPU. It will first process the command line arguments through a configuration process on the host. If the configuration indicates a normal execution, the host side program will take the Python code in and assemble it into byte code which is a concise representation of the original Python code. Meanwhile, the ePython interpreter will start on every e-core waiting for the byte code to interpret. After passing the byte code to the interpreter, ePython will start the monitor on the host side. The monitor will then assist the interpreter to process specific function calls such as IO and math functions. The byte code will act as the incarnation of the original Python code and be executed concurrently on all active e-cores. The procedure described above is the standard running mode of ePython, a list of options altering its behaviour can be found by issuing `-help` in the shell when starting the ePython.

The host can also run interpreters as virtual Epiphany cores. The reason why it is the byte code rather than the original Python code that is passed to the Epiphany is that the local memory of e-core is too small to run a full Python interpreter. Thus the interpreter and the code itself need to be optimised to make most use of the limited e-core local memory.

The ePython code is composed of several main functional parts as follows.
Lexer

The first step of generating byte code from Python code is the lexical analysis of the Python code. This is performed by the lexer. The lexer will filter out the comments in the code, omit the characters for formatting such as some extra spaces and extract the key words and the functional characters as tokens. The processed code will then be handed over to the parser.

Parser

The second step of generating byte code from Python code is the syntax analysis of the token sequence provided by the lexer. This is also known as parsing and is performed by the parser. This is the step in which the machine comprehends the code. The tokens will be endowed with properties according to the Python grammar rules. They will be categorised into variables, values, delimiters, operators and functional keywords. And based on their properties, the tokens will be related by their dependencies. All the tokens will be linked into a parse tree. The parse tree will be used as the basis for the byte code assembling.

Byte code Generator

The final step of generating byte code from Python code is assembling the byte code with respect to the parse tree. All the variables, values, operators and functions will be stored as hexadecimal numbers in bytes. Among them, in the byte code, the variables will be regarded as symbols. The symbols are just the substitutions of the variables. A symbol table will also be created along with the byte code. The symbol table is a map between the symbols in the byte code and their physical memory address in the e-cores' heap. The byte code is composed of simple statements, these statements can be executed directly by the ePython interpreter.

For instance, the byte code representation of python code in listing A.1 can be found in listing A.2. To generate a byte code file from a Python code file without executing it, users can pass \(-o \) bytecodename to the ePython. As an optional running mode, users can pass \(-l \) bytecodename to the ePython to execute a byte code file.

Monitor

The monitor manages the running of ePython on the Epiphany and assists the e-cores to finish certain functions in the runtime such as IO functions and some math functions. The monitor runs on the host side, concurrently with the interpreter on the device, and communicate with the e-cores in runtime level. The monitor is the agent for the e-cores to interact with outside world. It cyclically inspects the data sent by the e-cores in the shared memory and send the data back to the e-cores by writing the shared memory.
Every time an e-core sends a request command to the monitor, the monitor will suspend the e-core and release it again until the host has finished the request and written the data to the shared memory.

In this project, the monitor will be extended to become a pivot for the messages passed among e-cores located on different Parallella boards.

**Interpreter**

The interpreter is the program that executes the byte code generated by the ePython host. Normally, it starts on the active e-cores but it is portable and can, if needed, be launched on the host as well, acting as a virtual e-core. The interpreter will start once it has received the byte code from the byte code generator or a byte code file. The interpreter processes the byte code statement by statement. When executing some native function calls, the interpreter will invoke runtime. The interpreter exits after having processed the end of the byte code, and the exit of the interpreter will deactivate its hosting e-core.

**Runtimes**

The runtime invoked by the interpreter performs low level memory operations such as memory allocation and deallocation, inter-e-core communication with e-mesh NoC and communication via shared memory. Because the interpreter are designed to be able to run on both the host side and the device side, ePython therefore has two runtimes, they are used in the host ARM CPU and the Epiphany processor respectively. The IO functions and some math functions on the e-core require the assistance of the host CPU, thus they are also a part of the Epiphany runtime. In addition, all the message passing function calls are implemented in the runtime.

The mechanisms of the local communication calls, based on which the remote message passing functions were implemented in this project, are described as follows.

**2.3.3 Local communication mechanism**

Prior to the extending work in this project, the ePython device runtime has already been equipped with a set of functions to accomplish the point-to-point communication calls as well as some collective communication calls. They are described as "local" because the communication happens when and only when passing messages among the e-cores within one single Epiphany chip. These runtime functions are designed to be invoked by the interpreter and can be executed individually by the e-cores without the assistance of the host CPU. Nevertheless, the single call for any of these functions will not work, the message passing operation as a whole still needs every cooperating e-core to correctly perform certain combinations of these functions.
Local send

The sending process is merely the copying of data from sender e-core’s private memory to target core’s receiving buffer. It is performed by the sender core. Every e-core has a section of memory used as receiving buffer in their local memory space. Each core’s receiving buffer is divided into 16 cells for receiving the messages coming from the 16 e-cores respectively.

The synchronisation in a unidirectional point-to-point message passing is guaranteed by each core in a send-receive pair comparing the other core’s synchronisation flag with the flag value attached with the message in the receiver’s receiving buffer. The synchronisation flag is stored in the shared memory, in a char array of length 16, which maps the 16 e-cores in an Epiphany. The two communication cores can only modify the flag value of the other core in a point-to-point communication. At the beginning of the unidirectional send and receive operation, the sender core updates the receiver core’s synchronisation flag to a transitional state and send this flag value with the message. After sending the message through remote memory access, the sender core will update the receiver core’s flag to a finish state. If the receiver core gets the message with a transitional flag appended to it, the receive function will update this state value in the message it just received, which currently locates in its receive buffer, to a finish state. The sender core will wait until eventually finding out that the state value in the receiver’s receiving buffer matches the finish state for the receiver’s synchronisation flag and exit the local send procedure.

Local receive

The receiving process is the retrieving of data from the receiver core’s receiving buffer. It is performed by the receiver core. The receive function choose which cell in the receiving buffer to copy based on its sender ID argument.

To ensure the synchronisation with the sender core, the receive function update the sender’s synchronisation flag to a transitional state at the beginning of the execution. If the message has arrived at the receiving buffer, the flag value of the message should match this transitional flag. If not, the receiver core will continue checking the flag value in the receiving cell where the message is supposed to be delivered and copying the message to its private memory during every inspection. Once the transitional flag value in the receiving message has been confirmed, the receive function will update the sender’s synchronisation flag to the finish state. As stated in the local send section, the sender core will be released after this operation. The message copy in the private memory will then be repacked in the format of internal data type and returned to the interpreter.
**Local send-and-receive**

The send-and-receive operation is accomplished by a pair of cores to pass messages between them in a bidirectional manner.

At the beginning of the bidirectional send-and-receive operation, each core updates the other core’s synchronisation flag to a transitional state and send this flag value with the message. The messages will arrive at target cores’ receiving buffer. Then for each core, a receive operation, as stated in the local receive section, will be performed but instead of sending the returned value of receive function directly to the interpreter, it will be kept by the send-receive-function temporarily. After the receiving, each core will update the synchronisation flag value in its own receiving message to a finish state. Next, they will be repeatedly checking the synchronisation flag value in each other’s receiving message. If it is the finish state, the returned value from the receive function will be passed to the interpreter.

**Local barrier**

When the ePython device starts to execute, with the call of the barrier initialisation, two arrays both of length 16 will be created for each e-core in their local memory. The first array A will be initialised to all zero. For the core 0 the \(i\)th element in the second array B will be initialised to the remote address of the first element of array A on the core \(i\). For other cores, the first element in the second array B will be initialised to the remote address of the \(i\)th element of array A on the core 0. Briefly, The array B is used to remotely access the elements in the array A of other e-cores.

If the barrier synchronisation function is invoked by all active cores, every core except for core 0 will modify the \(i\)th element of array A on core 0 to declare their arrival, where \(i\) is its core ID. Then they will wait for the first element of their own array A to be modified as a continue permit signal signed by core 0. Before continue, the core 0 will wait for all other cores to modify their corresponding values in core 0’s array A. Then the core 0 will reset its array A and modify the first element of array A for all other cores to release them. By the coordination of core 0, all the cores can continue executing the code behind the barrier synchronisation only if all the active cores have reached the barrier.

**Local broadcast**

The broadcast function is built on top of the previously implemented unidirectional point-to-point communication function. Every e-core excluding the root core itself will perform receive, whereas the root core will perform multiple send operations to pass its data to all other cores. At last, all the broadcast function calls on every active e-core will return the broadcast value to their respective interpreters in the format of internal data type.
Local reduction

The reduction function is built on top of the previously implemented bidirectional point-to-point communication function. Each e-core will exchange its own value with all other e-cores through performing multiple send-and-receive operations with the core ID from the lowest to the highest excluding itself. Rather than storing all other cores’ values and then perform a reduction with multiple arguments, every time a core receives a new data, it will perform a binary reduction operation, store the result and discard the value received. The reduction operation can be summation, multiplication, maximum or minimum. Eventually, all the reduction function calls on every active e-core will return the same reduced value to their respective interpreters in the format of internal data type.

2.4 Challenges

So far, we have introduced all the technologies that might be helpful in building the multi-Epiphany computing system. This work has never been accomplished by anyone before. Programming a single Epiphany is already a challenging problem because of the low level programming required. And programming a interconnected Epiphany is even harder. The Supercomputer.io might be a paradigm but it failed. The eBSP, OMPi, COPRTHR and ePython can work on single Epiphany but they all cannot scale in the Epiphany network. No precedent hardware to reference and no existing software technology to instantly employ, the extended ePython will be the only one of its kind to transparently support explicit parallel programming on remotely connected Epiphanies.
Chapter 3

Design and Implementation

3.1 Construction overview

From the Python programmer point of view, all the Epiphany cores within a Parallella can identically and simultaneously run their parallel Python programs. However, using the previously developed ePython, they had to limit the number of concurrent processes in their program to whatever the core count a single Epiphany can provide. What if the programmer wants more Epiphany cores? Inevitably the ePython need to be extended to run on a more powerful machine that contains enough number of Epiphanies. But the Parallella board does not contain any slots to insert additional Epiphany chips. Even if it could, the programmer might still run out of Epiphany cores very soon. Since the previously mentioned Supercomputer.io project has failed, do they have to wait until the Adapteva releases the next generation of Epiphany? What if they want an extremely large number of Epiphany cores instantly and even the greatly increased core number in the next generation of Epiphany is still insufficient?

A simple and feasible way to obtain a machine that meets a wide range of requirements of the number of e-cores without changing the hardware design of Parallella itself is:

3.1.1 Building a Parallella cluster

The cluster was designed with one simple principle. That is from the perspective of the Python programmer, this cluster will be a large distributed machine with a large number of Epiphany chips and hence enough number of e-cores to use.

As was mentioned above, Parallella is a small-sized computer that supports Ethernet technology. All versions of Parallella contain a RJ45 connector makes it possible to connect the Parallellas to a wired Local Area Network and to utilise MPI. And this is the way how the stand-alone Parallellas are united to comprise the prototype Parallella cluster for developing ePython and demonstration of interpreting Python code. The
configuration details can be found in the appendix B. Although only two Parallellas were used in the prototype cluster, the technology developed here can support clusters with any number of distributed Parallellas.

Using the Parallella node as the basic component and having all the Parallella installed in an equal position, this distributed Parallella cluster has twofold advantage. On the one hand, the flat structure of the cluster can decrease the developing difficulty of ePython on it; On the other hand, the cluster size can be flexible and the scaling of the cluster will be straightforward. Users could connect as many Parallellas as they want to make the cluster contain plenty of Epiphany cores.

To manage the cluster efficiently, especially when distributing Python code developed by the ePython user, a shared file system would be necessary. In this implementation of the cluster version ePython, every node within the cluster needs to get a copy the Python code before the ePython starts running. As far as the usability is concerned, Python programmer should just edit the Python code in one place and does not need to worry about copying the code across all Parallellas. Among all kinds of the shared file systems two of them were considered in building the Parallella cluster.

NFS or SSHFS?

Both the NFS(Network File System) and the SSHFS(Secure SHell File System) allow the users to access files in a remote machine with the same syntax as accessing local files. They both are distributed file system using client-server mode, which means not every node within the cluster possesses the shared directory equally even through their shells access it in the same way. The files in the directory mounted by these file systems are physically located in the server node and the client nodes need underlying network communication application to to help them perform the operation remotely. For NFS, it is based on its own protocol of the same title which is currently not installed on Parallella. Whereas for SSHFS, the SSH File Transfer Protocol it is using is shared with SSH and has been installed on the Parallella’s official operating systems already.

Theoretically, both file systems could serve the purpose of distributing Python code on the Parallella cluster very well. However, in the actual implementation, the Parallella runs a special Ubuntu distribution version in which two Linux kernels were compiled and the active one is not compatible with the NFS. The only plausible way to fix this was to modify the the Parabuntu OS, but this might cause further issues. Compared with the complicated configuration procedure of NFS, the installation of SSHFS is much easier and hence was adopted as a roundabout. Another advantage of using SSHFS is that SSHFS simplified the security settings and thus can be easily applied to the Internet connected Parallellas,

The deployment of SSHFS does not require any operation on the server-side. After installing the SSHFS program on all client nodes, users could mount the directory of a server node to all client Parallellas’ local directories with the same relative address by issuing:
$ sudo sshfs username@IP:/dir /dir -C -p 22 -o allow_other

at the start of each client Parallella. This SSHFS command also works for the Internet-connected Parallellas. In contrast, NFS setting could have much more complicated procedures for the same purpose due to security considerations.

Then the shared file system is ready to use. In using the distributed ePython, users only need to login to the server Parallella and edit their Python code in the shared directory.

### Configuring MPI on Parallella cluster

Running MPI on a cluster is essentially starting the MPI program on one master node and then the master node performing locally execution for its own MPI process and performing remote executions on slave nodes for other MPI process. Therefore, having MPI installed on all nodes is required. In the prototype cluster, the MPI implementation is OpenMPI 1.8.3 which comes preinstalled in the Parabuntu OS.

The remote execution performed by the MPI stands on the foundation of passwordless SSH. Before enabling this feature, the IP addresses of all Parallella nodes on the LAN need to be obtained first. And for the sake of convenience, usernames for logging in to Parallellas are set to be the same. The details about the passwordless setting can be found in appendix B.2.

Which process should be executed on which node is specified on the master node in a host file. The MPI program should be started at the master node with the host file passed as an argument into mpirun. As shown in listing 3.2, The host file records the host names(or IP addresses in LAN) and number of process for each node(specified by slots) in lines.

```plaintext
# The host file for OpenMPI
# Nodes are considered as single processor machines
# The master node
parallel0 slots=1
# The slave node(s) list
parallel1 slots=1
```

Listing 3.1: MPI host file

According to the topology we used for the prototype cluster, each node will run one MPI process, which has been specified by slots=1 in the MPI host file. And these MPI processes will be communicating with each other and executing the host-side monitor threads of the ePython which are used to manage their own Epiphany processes.

### 3.1.2 General methodologies

The principle of modifying ePython followed in the whole project is to make as few unnecessary adaptations as possible. Since the existing ePython has run very stably on the single Parallella already, the implementation of the functionalities that support the
distributed Epiphany should make most use of the code for the local communication whenever applicable. And this could have many benefits. One obvious benefit is that this could reduce the workload for the debugging as the debugging on the Epiphany would prove to be the most challenging and tiresome work in this project. Another benefit is that the extended ePython will not incur too many extra overheads when it is used back in a degenerate case in which the cluster only contains one node and which is exactly the previous and the most common usage of ePython.

The purpose of the extending work is to enable ePython to run on the cluster just as if it was running on a single Parallella board except with a larger virtual Epiphany that contains more cores. Specifically, the number of Epiphany cores that is available to the distributed ePython users is the summation of all the Epiphany cores on all Parallella boards within the cluster. The performance of the distributed ePython for this virtual Epiphany is not the first consideration because passing data between different remote e-cores without using the eMesh network will definitely increase the communication overhead. However the increased number of Epiphany cores can allow the parallel Python programmers to implement patterns of more complexities in their code. For the ePython users, the realisation of parallelisation in Python code will still be transparently carried out by the calling of communication functions using the same syntax.

As described in the background chapter, the programming model of parallel Python is message passing. Hence, what the distributed ePython will do is effectively passing data among all e-cores in the cluster. The passing of data among local e-cores has been implemented in the previously developed ePython, therefore, the extending work will be focusing on passing the data among remote e-cores.

![Remote communication procedure](image)

**Figure 3.1: Remote communication procedure**

In general, the completion of all kinds of remote communication includes three phases:

- **a)** device calling phase,
- **b)** host relaying phase and
- **c)** device completing phase.

As shown in figure 3.1, both the calling phase and the completing phase take place on the Epiphany cores, whereas the relaying phase takes place on the host ARM CPUs.
Thus both the host-side code and device-side code (c.f. section 2.3.2) of origin Epiphany were modified. Followed is a brief introduction of what the cluster ePython does on both sides to finish all three phases.

**Host-side activities**

The host-side program is executed at the entry of ePython. In terms of cluster version of ePython, each host-side program is a parallelised MPI process. Therefore in the makefile, the C compiler for epython-host should be changed to mpicc. Each Parallella node runs one of these MPI processes. Hence at the beginning of the main function of the host program is where the MPI initialisation routine is called. Because the host process will create a pthread on which the ePython monitor will run, the MPI process should be initialised with MPI thread environment. Instead of using MPI_Init, MPI_Init_thread is used. The MPI threading mode is set to be MPI_THREAD_SERIALIZED because only the monitor thread will make MPI calls. The bash script file for starting the ePython by executing epython in the terminal also need to be modified. Inside the master node’s bash file under /usr/bin/epython, instead of using ./ to start the epython-host program, mpirun should be used as a substitution:

```bash
mpirun -x LD_LIBRARY_PATH -x EPIPHANY_HDF=${EHDF} -x EPYTHONPATH --hostfile ./mpi_hostfile
```

Because the epython-host program on the slave node is started remotely by MPI and the bash script in regular starting of ePython will be bypassed, apart from the process number and the MPI hostfile, the environment variables in the ePython starting bash script also need to be passed to mpirun.

When starting the program, the ePython needs to know the cluster topology in order to perform correct message passing across different Parallellas. To allow the epython-host to find out on which Parallella it is running and how many other Parallella nodes there are in the cluster, two variables (the `nodeId` and `numNodes`) are defined. One naïve way to assign values to these two variables would be passing predefined environment variables that store the information about cluster configuration on each Parallella node to the compiler as compile-time variables. However, by doing this, every time a new Parallella node is added to the cluster, all the nodes have to recompile the ePython as the `numNodes` will be changed. There must be a more elegant and dynamic way to do this. Since the MPI has been adopted as the communication means among different Parallellas, these two variables can be assigned at runtime. The ePython hosts running on different Parallellas can get their node ID and total number of nodes by the call of `MPI_Comm_rank` and `MPI_Comm_size`. Benefited from this dynamic assignment mechanism, the user will not have to worry about setting different environment variables for different Parallella nodes when they install the cluster. In other words, the ePython can be preinstalled in Parallella for the ePythons running on different Parallellas within the cluster are essentially the same compiled version. The Epiphany will get to know the node ID and the total number of nodes through the Epiphany configuration process (as mentioned in section 2.3.2) during the initialisation of epython-host.
After successfully starting epython-host on all Parallella nodes, every host program will assemble the byte code from the Python code, load the byte code to their own Epiphany and start the monitor thread on the host CPU. When the monitor comes across a request from a e-core, it will carry out its part of work and then reply to the e-cores which sent the request. The remote communication request is a kind of requests sent by the e-cores. The aforementioned host relaying phase is the period when the monitor processing the remote communication requests.

The monitor returns after all the local e-cores have finished their job. The MPI finalises when all monitors have returned. The ePython exits once the MPI has finalised.

Device-side activities

The epython-device will start running on the e-cores at the same time the host begins to execute epython-host. The interpreter runs in the epython-device processing byte code sent from the host. When the interpreter comes across functions that need interaction with the host to finish (e.g. the remote communication functions), it will suspend the execution of the byte code and wait for the host to reply. This is called the standard monitor-device interaction and it always happens between two remote communication phases. During the standard monitor-device interaction an e-core executes the following steps in sequence:

1. Storing the core_busy flag to a temporary shared variable used as barrier reference
2. Updating the core_command flag which indicates the type of operation the core requests to be performed by the monitor
3. Updating the core_busy flag to 0 to allow the monitor operation to start
4. Waiting until the core_busy flag becomes larger than previously stored barrier reference (the core_busy flag is updated by the monitor after having finished its operation)

In normal cases, the device program finishes when it hits the end of the byte code or meets the commands that instruct it to exit explicitly. All the exits of the e-cores will have to be reported to the host as a special request.

Up till now, the basic structure and the operating principle of the distributed ePython have been given. Among the functions that make the distributed ePython work, the communication functions play a vital role. They are described as follows.

3.2 Point-to-point communication

Definition: The point-to-point communication refers to passing messages between two arbitrary Epiphany cores. The cores may be on different Parallellas.
Data transferring between two cores in the same Epiphany can be accomplished through via either the e-Mesh network or the shared memory of the host ARM CPU and the Epiphany. However, the fast e-Mesh network will be unavailable if the Epiphany cores are passing messages remotely. In the Parallella cluster, the two remote communication cores need to send or receive data to or from their local host processors through the shared memory. The data in local shared memories is transferred by the host MPI processes via LAN connecting these Parallellas.

To invoke one-way point-to-point communication, one needs to issue a send statement on the sending core and a pairing receive statement on the receiving core. Two-way point-to-point communication can also be accomplished having single send-receive statement issued on both sides.

For python programmers, the point-to-point message passing module on the Parallella cluster is implemented in the ePython as native functions declared in Python library parallel that comes with ePython. The syntax of Python code for these functions is:

\[
\text{send(value_to_be_sent, receiver_global_id, [value_count])}
\]

\[
\text{recv(sender_global_id, [value_count])}
\]

\[
\text{sendrecv(value_to_be_sent, receiver_global_id, [value_count])}
\]

The send does not have return value. The recv has the received value as its return value. The sendrecv has the received value from the receiver specified by its second argument receiver_global_id as its return value. They are still same as the previous version in parallel Python. But for the core ID arguments, the the Python programmer can now use any integers that are smaller than the total e-core counts of their Parallella cluster. Internal to this, the module interfaces with the runtime via a special inbuilt function, callNativeFunction.

### 3.2.1 Extending the send P2P communication call

**Definition:** The send refers to all the actions a core takes to pass its own data to the other core in a one-way point-to-point communication. The Epiphany core can be instructed to perform the send by issuing send() in parallel Python code.

The send internal function set obtains the data from sending core and pushes it into either another local Epiphany core’s private memory or the inter-Parallella network, depending on whether the target is local or remote. All the sends implemented on the current version of ePython, regardless of communication locality, are blocking. For the local send has been accounted in section 2.3.3, this section will focus on the newly added remote send. Remote send needs the assistance of host CPU to carry out. Hence the complete remote send consists of the monitor and the Epiphany runtime running on the host and device respectively. And the system performs the remote send by close cooperation of the two programs.

At the beginning of the send procedure, inside the epython-device, exists a sendData helper function that helps the e-cores to decide which kind of send should be performed.
Because the send is one of native functions implemented in the parallel Python, this function is invoked by the `callNativeFunction` which performs the role of interface between ePython’s interpreter and runtimes. With the call of this helper function, the parameters that contain useful information for sending, e.g. the target core identity number, are also passed to the remote send function. This marks the beginning of the calling phase for the remote send. The helper function will then resolve the global target core ID first and then decided if performing remote send is necessary. If the global ID is considered non-local, the remote send procedure will start working.

The first step of a remote send is to transfer the data from e-core’s private memory to the share memory where the data is visible to the ePython monitor running on the host. This step is accomplished by `sendDataToHostProcess` which is made up of two sub-procedures running on the Epiphany: the data preparation and the standard monitor-device interaction. The data preparation in remote send on the device is made up of a rearrangement of the bytes to make most use of memory and a copying of the data itself and the additional data (such as data type and target ID) to the send register, specifically, to the postbox memory area that is reserved for communication data between monitor and this Epiphany core performing remote send. The data is packed into a 16-byte register in the way shown in figure 3.2.

![Figure 3.2: Register for remote send](image)

The second step of a remote send is the local host sending the data in the shared memory to the remote host using MPI. This is the beginning of the host relaying phase where the local host process will cooperate with another remote host which is handling the remote receive request coupled with the current send. The ePython monitor running on the host CPU scans and processes Epiphany cores’ requests in a cyclic pattern. Ideally, the monitor will process the send request by issuing MPI send and wait for another host to issue the paired MPI receive. After the MPI send has returned the monitor can reply to the e-core and carry on scanning of requests coming from the next cores. If only two distant cores issue send and receive respectively, and the monitor have no other commands to deal with, this model will work perfectly. But what if the monitors need to deal with the multiple communication requests that might be able to lock and unlock each other? Consider the circumstance as illustrated in figure 3.3.

As in figure 3.3, for the monitor on Parallella A, the Epiphany core 2 precedes the Epiphany core 3 in the request processing sequence. If the core 2 wants to send a
message to core 20 using MPI blocking send and using the target core’s global ID as the MPI tag, it will have to wait for the monitor on Parallella B to finish the processing of request from core 19 and to issue MPI receive in processing the request of core 20. The previous implemented monitor can and only can handle one command for each core in one monitor cycle. This is because the monitor was originally designed to handle non-communicational commands such as input and output, and these commands can be completed by the monitor one by one without causing any troubles. Thus, if we were to use the existing monitor to handle the remote communication requests, the processing of the request from core 19 would never finish because the processing of core 3’s request never started on the Parallella A. The solution to this problem is:

**Two-step mechanism**

In order to avoid the deadlock of monitors, the remote send procedure has been split into two parts. The first part will be executed once the command flag has been updated by the e-cores and then the monitor will start to process the next e-core’s command. The second part will not be executed until the nonblocking MPI calls issued in the first part have been finished when the monitor turns its attention again to this sending e-core. Therefore the updated monitor will not necessarily have to finish a send in a single cycle of scan for all local e-cores, and all MPI calls can be issued. Furthermore, it can always know which part has yet to be executed even the whole send procedure spans several monitor cycles. This is called the two-step mechanism and is the key to avoid deadlocks in all kinds of inter-Parallella point-to-point communication functionalities.

When the monitor gets a send command from a device core, it will retrieve the send stage information by reading the current core’s commStatus flag. If the flag is 0, the monitor will start executing the first stage of remote send which contains nonblocking MPI statements, the first stage will leave the MPI request handlers of nonblocking MPI
calls to the monitor for later checking of MPI call status. The MPI request handlers are placed in a handler array 32 entries in length, which means the monitor can store up to 2 MPI request handlers for each local e-core. If the first stage has been indicated completed by commStatus flag equals to 1, the monitor will issue MPI_Test for the nonblocking calls made in the first stage. If the completion of the MPI communication calls is confirmed, it will start executing the second stage of remote send and reply to the Epiphany, otherwise continue processing other e-cores’ commands. In the first stage, the monitor packages the data in the shared memory, resolves the target Parallella’s MPI rank and sends it to the target Parallella by issuing the MPI_Isend statement. The MPI_Isend was used to instruct MPI explicitly to choose not to buffer the outgoing messages so as to save memory in the monitor. The monitor uses the request core’s global ID as the MPI tag in the send. Since the send buffer has been arranged by the e-core and it is now just a linear combination of memory in the 16 bytes long "postbox" memory window, also known as the "register", MPI_BYTE is used here as the send type regardless of the actual data type of the message. Instead of sending the whole send register, only 5 effective bytes start from the byte position 5 in the send register will be send via MPI. At the end of the first stage, the monitor updates the commStatus flag to 1 and retains the core_command flag. The second stage of remote send is executed after the monitor detects the nonblocking communication has been finished in a certain monitor cycle. This stage for the remote send is quite simple. The host resets the commStatus and the core_command flags to 0, increments the core_busy flag by one and the Epiphany core will continue the execution of its own program after noticing the updated barrier reference at the end of the standard monitor-device interaction.

3.2.2 Extending the receive P2P communication call

**Definition:** The receive refers to all the actions a core takes to obtain data from the other core in a one-way point-to-point communication. The Epiphany core can be instructed to perform the receive by issuing `recv()` in parallel Python code.

Similar to the send, at the beginning of the receive procedure, inside the runtime of Epiphany, also exists a `recvData` helper function that helps the e-cores to decide which kind of receive should be invoked. The local receive does not entail any changes from the previous implementation. The description of local receive can be found in section 2.3.3. In terms of extended core ID, the following adaptation applies to all local P2P runtimes even if all the local P2P functions themselves are essentially not changed in the extending work for the cluster ePython. Because the core ID argument passed in all helper functions that determine whether to perform remote communication is the target core’s global ID, and it is incompatible with the local communications implemented before, the core ID passed to `recvDataFromDeviceCore` and any other kinds of local communication functions has to be reverted to the local ID inside the helper functions before the local communication functions are invoked.

If the helper function decides to perform a remote receive call, the device remote receive runtime function `recvDataFromHostProcess` will start. The remote receive command
along with the expected sender ID will be sent to the host and the Epiphany core will enter the standard monitor-device interaction directly.

On the host-side, the monitor also completes this remote communication request with the two-step mechanism. After the monitor receives the request command from the receiving core, it will issue an unblocking MPI receive call immediately based on the rank resolved from the sender ID provided by the e-core and the tag that is exactly the global sender ID itself. This is the first stage of remote receive and it is performed through executing `remoteP2P_Recv`. This function will post a MPI_Irecv call and leave the monitor a MPI request handler. The receive buffer for MPI is a 5 bytes long memory area in the receive register as shown in figure 3.4. At the end of the first stage, core_command flag will be retained and commStatus flag will be set to 1 to indicate that the monitor has completed the first step of remote receive. Then it will test the nonblocking MPI receive call in every inspection cycle of the monitor by issuing the MPI_Test.

![Figure 3.4: Register for remote receive](image)

If the monitor finds that nonblocking receive has been completed in a certain follow-up monitor cycle, the commStatus flag and the core_command flag will then be reset to 0 and the core_busy flag of requesting core will be updated to release the e-core from suspension.

On the device-side, after leaving the standard monitor-device interaction, the e-core will translate the message in the receive register and return the received data to interpreter-runtime interface in the form of ePython-device internal data transfer type. Finally, the `callNativeFunction` will return the received data to the ePython interpreter.

### 3.2.3 Extending the send-and-receive P2P communication call

**Definition:** The send-and-receive refers to a combined operation for a core to pass its own data to the other core and to obtain the data form the other core in a two-way point-to-point communication. The Epiphany core can be instructed to perform the send-and-recv by issuing `sendrecv()` in parallel Python code.

As mentioned in section 2.3.3, the syntax of `sendrecv` in the current parallel Python only allows the caller to send and receive with the same target core. Which means only one sender global ID is passed to this function and it is also used as the receiver global Identifier. The two e-cores involved in a send-and-receive activity are called a send-and-receive pair.
The remote send-and-receive is essentially a combination of remote send and remote receive. For the remote send, the sender needs to pack the data into the register in the device calling phase, it will not have any operations on the data in the device completing phase. In contrast, for the remote receive, the receiver only transfers the sender ID without data in the device calling phase and the operations on the data per se happen in the device completing phase. The job of send-and-receive is, for each core in a send-and-receive pair, the runtime not only performs the outgoing data transfer in the device calling phase but also performs incoming data transfer in the device completing phase. In the host relaying phase, both cores in a send-and-receive pair employ the two-step mechanism.

Same as all other point-to-point communication mechanisms, the send-and-receive is led by a helper function `sendRecvData` to evaluate the necessity of performing the remote communication. Inside the device runtime, when the remote send-and-receive function `sendRecvDataWithHostProcess` is invoked, the e-core will do everything just as it will do in device calling phase of remote send. Except that the request it sends to host this time is reserved for send-and-receive and the two-way message passing demands more data in the register as shown in figure 3.5.

![Figure 3.5: Register for remote send-and-receive](image)

Then, during the standard monitor-device interaction the core will be suspended until the monitor have written the received data to the register in the shared memory.

On the host-side, the monitor will issue a MPI_Isend and MPI_Irecv at the first stage. The synchronise mode is explicitly used for MPI communication for the same reason as stated in section 3.2.1. MPI_Sendrecv is not used here because the two-step mechanism of the monitor requires the MPI calls in the first stage to be nonblocking. During the succeeding monitor cycles, it will test the MPI requests for the two nonblocking MPI calls to decided whether it is ready to perform the second stage which is to unpack the received data to the send-and-recv register for the e-core to use. A correct match of remote send-and-receive can make sure that two monitors will rendezvous at some point after all of them have completed the first send-and-received stage.

On the device-side, after leaving the standard monitor-device interaction, just as what the receive will do, the e-core will translate the message in the shared data and return the received data to interpreter-runtime interface. At the last step of send-and-receive, the `sendRecvDataWithHostProcess`will return the received data to the ePython interpreter in the format of ePython-device internal data transfer type via `callNativeFunction`. 

26
Note  All the technical details of the point-to-point communication mentioned above are runtime level mechanisms. The unidirectional send/recv pair can only transfer one single value at a time and the bi-directional send-and-receive pair can only be used to transfer one single value at a time for each direction. They are designed as such because the limited Epiphany memory requires the lightweight and simplicity for the ePython device runtime. In fact, users can still pass an entire array with a single point-to-point statement in their Python code and specify the value_count argument. The ePython will split such statements into a series of massage passing operations when assembling the byte code from users’ Python code.

3.3 Collective communication

Definition:  The collective communication refers to passing messages among a group of Epiphany cores. The cores involved in the collective communication are possibly on different Parallellas.

In the current version of parallel Python, the group of participating cores, which is defined as a communicator, for the collective calls is by default all the active cores initialised by the ePython. Users can not designate their own communicators during the interpreting time. The finish of a collective call in a core does not necessarily mean that the other cores in the communicator has finished their communication but if a core exits its collective communication call all other cores must have started this call.

To invoke collective communication, users need to issue the same collective statement for all Epiphany cores within the communicator. These statement will be matched by ePython in accordance with the order they appear in the Python code.

For python programmers, same as the point-to-point functionalities, all the collective message passing functionalities on the Parallella cluster are implemented in the ePython as native functions declared in Python library parallel that comes with ePython. The syntax of Python code for these functions is:

\[
\begin{align*}
\text{sync}() \\
\text{bcast(buffer, broadcast_root_id)} \\
\text{reduce(data, reduce_operation)}
\end{align*}
\]

The sync performs a global barrier, it does not take in any argument and does not have any return value. The second call bcast sends the value in buffer from the core with the core ID broadcast_root_id to all other active cores within the cluster. For all the cores involved in broadcast, they receive the broadcast message from the return value of bcast. The third call reduce performs reduce_operation for the data provided by all cores. Every core can obtain the reduced data from the return value of its own reduce call.

Although these remote collective calls could have been constructed from the previously implemented point-to-point calls directly, the distributed ePython implemented them in a better way that makes the execution of these collective calls more efficient.
3.3.1 Extending the barrier synchronisation call

**Definition:** The barrier refers to a pause point in the program before where the program part must have been finished by all the cores. The actions taken by the cores in the pause point is called barrier synchronisation. The Epiphany cores can be instructed to perform the barrier synchronisation by issuing `sync()` in parallel Python code.

The global barrier is used for waiting all active cores with in the cluster to reach at a checking point before any cores continue to execute the following code. The barrier call is the foundation of all collective communication function calls.

A barrier communication consists of a all-to-one operation and a one-to-all operation. A naïve way to realise this would be for all the active e-cores within the cluster to report to a master e-core by sending signal messages. And the master e-core would release them after all the report messages form other e-cores have been received. However, when the reporter cores are located on a different Parallella from the master core, every reporting message will become a remote message, this will greatly slow down the entire system especially when most reporting messages are passed remotely. In reality, the ePython will perform local barrier first and then the remote communication will be limited to the master host and all other local master cores. As shown in figure 3.6, this series of recursive operation can reduce the unnecessary remote message passings, thus the communication overhead will be significantly decreased.

![Figure 3.6: Recursive barrier synchronisation](image)

When the ePython interpreter identifies a sync token in the byte code, the interface function `callNativeFunction` will invoke `syncCores` helper function to start the global barrier synchronisation. This helper function takes the charge of deciding to what extent the recursive barrier should be performed based on the cluster configuration information stored in the shared memory. If the active cores are scattered on multiple Parallellas, the `performBarrier_remote` in the runtime will start.
The first part of the performBarrier_remote will perform exactly what the previously implemented performBarrier will do in the all-to-one operation stage (c.f. section 2.3.3). Next, instead of releasing all the local cores, the local master core will send a global barrier request command to the monitor and enter the standard monitor-device interaction.

On the host side, when the monitor receives a global barrier request from the local master core, it will call syncNodes function to perform global barrier. This function contains blocking MPI calls. The monitor is guaranteed not to raise deadlock because by the time the local master core sends the request all its local peers have been locally synchronised and they all have not been released by the local master e-core yet. The MPI_Barrier call could have been used for the global level barrier synchronisation but to reveal the internal working of a barrier, an alternative algorithm was implemented.

The monitor will decide if it is the master host based on the cluster configuration. If it has the lowest MPI rank, it will start a set of blocking MPI receive calls, after all these receive calls have been finished it will then start a set of blocking MPI_Ssend calls to release other hosts. For those non-master hosts they will issue a MPI_Ssend call first and MPI_Recv call next. After all the MPI communication has been completed, the monitor will set the requesting core’s core_command to 0 and release the core from the standard monitor-device interaction by incrementing the core_busy flag.

As soon as the local master core finishes the standard monitor-device interaction, it will release all the local cores just as what it dose in the local barrier one-to-all operation stage. The released local cores will then successively exit the barrier call.

### 3.3.2 Extending the broadcast collective communication call

**Definition:** The broadcast refers to all the actions all cores take to spread data from a single core to all cores. The Epiphany cores can be instructed to perform the broadcast by issuing bcast() in parallel Python code.

The broadcast is used to send a message conveniently from a root core to all active cores including itself within the cluster.

The extended broadcast in cluster uses recursive pattern mentioned in section 3.3.1. Different from the previously described communication runtimes, the broadcast does not contain a leading helper function that helps the communication functions switch to remote procedures. The remote broadcast has been integrated into the original local broadcast function bcastData itself and has a local barrier at the entry of the bcastData. Considering that the inter-Parallella communication in the remote broadcast will be blocking, a local barrier is used to avoid monitor deadlock in the recursive broadcast. For the receiver Parallella, the barrier is implicit inside the remote broadcast receiver runtime, where as for the sender Parallella, the barrier is a explicit statement at the beginning.

When the bcastData is invoked by the runtime interface function callNativeFunction,
the core will figure out if the root core is located on the same Parallella as it is in the first place and then the root core and its local peers will issue the local barrier. If the core finds that the root core is its local peer, it will start a local receive procedure and wait for the root core to send the message, the message will be returned in the format of ePython-device internal data transfer type. If the core finds itself being the root core, it will, if necessary, perform the remote broadcast first by invoking \texttt{bcastSender} and then perform the local broadcast. If the core finds that the root core is a remote core it will act as a \texttt{bcastReceiver}.

The \texttt{bcastSender} will pack the message into a the register shown in figure 3.7 and enter the standard monitor-device interaction with the broadcast request command sent for the monitor. The root core will then send the data to all other local cores and return the data in the format of ePython-device internal data transfer type.

The monitor will decide whether to perform send or receive for MPI call based on the Bcast role flag in the remote broadcast register.

The \texttt{bcastReceiver} will choose a local core with the lowest core ID to be a remote broadcast agent to receive the message from the remote root core. Other local cores will be reporting to this core, waiting the agent’s response and issuing local receive calls after the waiting. This agent core will wait all other cores to arrive and then request the monitor to issue a blocking receive. After the standard monitor-device interaction session, it will perform a sequence of local core release along with local send. The agent core will return the value from remote root core and other local cores will return the value received from the local agent core. All the return values for the \texttt{bcastReceiver} will be in the format of ePython-device internal data transfer type.

### 3.3.3 Extending the reduce collective communication call

**Definition:** The reduce refers to all the actions all cores take to produce new data from the data provided by all cores according to certain reduction operation. The e-cores can be instructed to perform the reduce by issuing \texttt{reduce()} in parallel Python code.

The reduce synthesises the values each provided by an active core to a reduced value based on a certain reduce operation. ePython supports four kinds of reduce operation, summation, multiplication, maximum, and minimum.

A reduce operation consists of a all-to-one communication, a set of centralised reduce calculation and a one-to-all communication. Hence the reduce operation has an implicit

![Figure 3.7: Register for remote broadcast](image)
root to perform the calculation on the data it gathered from all active cores within the cluster including itself. The recursive pattern introduced in section 3.3.1 also applies to the global reduce on Parallella cluster. Same as the broadcast runtime, the reduce does not contain a leading helper function that helps the communication calls switch to a remote procedure. The global recursive has been integrated into the original reduce function `reduceData` itself.

At the start of the `reduceData`, it will do just as what the previously implemented reduce will do for a local reduce. The same operation goes through the entire `reduceData` until just before the return statement. Before the reduce returns the locally reduced data, this data will be send to host to perform global reduce, the procedure is carried out by `reduceData_remote`.

When the `reduceData_remote` starts, for each Parallella node the local core with the lowest core ID will be designated as the local reduce coordinator. The local reduce coordinator will set an implicit barrier at the beginning `reduceData_remote` to make all other local cores waiting for it to get the globally reduced data. Then the coordinator core will prepare the global reduce register as shown in figure 3.8 and send the locally reduced data to monitor with the global reduce command.

![Figure 3.8: Register for global reduce](image)

On the host side, MPI_Reduce could have been used but in order to illustrate the very internal working of reduce, an alternative algorithm was implemented.

During the standard monitor-device interaction session, the monitor with the lowest MPI rank will be elected as global reduce master monitor. It will perform the final reduce calculation with respect to the reduce operation flag in its global reduce register. The master monitor will issue MPI blocking receive for each of other monitors carrying their own local reduce value. Every time the master receives a new local value from other Parallella node, its local reduce value will be updated. After having reduced all the local reduce values within the cluster, the monitor on the master Parallella board will broadcast the global reduce value to all other Parallellas processing reduce command. Next all the monitors within the cluster will send the global reduce value to their local coordinator core.

After leaving the standard monitor-device interaction, the local coordinate core will broadcast the global reduce value to their local peers and release them meanwhile. At the last stage of reduce, the `reduceData` will return the received global reduce data to
the ePython interpreter in the format of ePython-device internal data transfer type via callNativeFunction.

3.4 Miscellaneous Supports for Clusters

3.4.1 Environmental native functions

The parallel Python code cannot correctly perform message passing only by invoking the communication functions themselves.

When making communication calls in the parallel Python code, program needs to know on which Epiphany core it is currently running. The only way to achieve this is by the programmer explicitly issuing the coreid() statement in the Python code. Sometimes the program also needs to know the total number of cores in its communication group by executing the numcores() function. Other two functions might be useful particularly for the distributed ePython running on the cluster are nodeid() and numnodes(). These four functions are fully supported on the distributed version of ePython as declared native functions inside the parallel module packaged with the ePython.

The cluster configuration values are stored in the shared memory when ePython host starts. The configuration parameters are initialised with the readConfiguration in the host main function. These configuration will then be write into the shared memory along with the byte code by loadCodeOntoEpiphany when the host initialises the interpreter on the Epiphany. When the device interpreter executing these environmental native functions, the runtime interface function callNativeFunction will be invoked. Next, the runtime will read the required value in the shared memory and return it to the interpreter.

3.4.2 Running configurations

When the user issues the epython command in the shell without any extra optional arguments to run a Python program, the ePython will start running with the default configuration. In the default configuration, ePython interprets the Python code on all available Epiphany cores within the cluster.

The optional argument -d can be used to specify the Epiphany core number used to run the Python program. Some adaptations to the ePython has been made to allow this argument work on the Parallella cluster.

Inside the readConfiguration, lies the only MPI_Comm_size and MPI_Comm_size that are called throughout the entire ePython code. They are used to initialise all the variables relates to the Parallella nodes configurations. The nodes configurations can be modified later by the parseCommandLineArguments. This function is located at
the end of readConfiguration and will do nothing if no optional argument is provided by the ePython user. For the \texttt{-d} argument, the \texttt{parseCommandLineArguments} will modify the intentActive array that instructs a specific core to be started and marked active when the ePython interpreter is initialised on the Epiphany. Instead of using the value provided for the \texttt{-d} directly, which might exceed the total local core number, as the number of value 1 in the intentActive array, the ePython will recalculate the local intentActive array based on the number of nodes, node ID and required total active core. The core on lower node rank with lower core ID will be activated preferentially.

3.5 Trouble Shooting

3.5.1 Premature POSIX Thread termination

When the MPI\_Init and MPI\_Finalize were first added to the host main function, the ePython could finish its work successfully but did not exit.

ePython failed to exit because of the \texttt{pthread\_exit} it used for explicitly terminating the master thread. Because the MPI\_Finalize was stated after the pthread\_exit in the main function, the premature termination of the master thread will leave the MPI\_Finalize unexecuted and cause the deadlock of MPI.

To fix this problem, the \texttt{pthread\_exit} should be replaced by \texttt{pthread\_join} to instruct the master thread to perform MPI\_Finalize after the created thread for monitor has finished. And because of the thread environment MPI used, MPI\_Init was replaced by MPI\_Init\_thread.

3.5.2 Stack overflow on Epiphany

When the expression of the \texttt{print} statement in Python code is very long or contains complex calculation, the ePython will exit immediately after this statement without any error message.

The problem was found when the Epython interpreter executing the print command in a test code for sendrecv(). The test passed when the implementation for the sendrecv() has just finished. But the trouble emerged after all the implementation for communication function had been carried out. To examine the running state of monitor, several printf statements were added and the \texttt{-O3} optimisation for the GCC was turned off. It is worth mention that when Monitor executing functions that contain IO operations, using \texttt{-O3} option can cause BUS error on Parallella. In the monitor print results, it showed that the monitor exited normally because all e-cores had their core\_run flag to be 0. The only thing that is able to modify the core\_run flag except the \texttt{deactivateCore} function in the monitor is the main function running on the device. The main function on the device resets this flag at the final step after the interpreter returns. Therefore the cause for this
issue is the unexpected exit of the interpreter. Since the interpreter had not reached the end of the program, the reason must be corrupt memory. As mentioned in section 2.1.2, the Epiphany does not have IO, and there will be no error message output. This explains why the ePython exited smoothly but left the rest part of code unexecuted. This issue did not happen when implementing point-to-point functions must because the Epiphany still had abundant working memory when other remote communication functions had not been implemented by then.

To fix this problem, some unused runtime functions such as garbageCollect in the original code were temporarily commented out so as to release enough memory for the evaluation of the expression in print. However, this is not an elegant approach and this solution should be substituted in the future by a better memory optimisation plan for all other functions in the runtime.
Chapter 4

Results and Evaluation

4.1 Individual functions test

In this section, the correctness and stability for the implemented functions are tested. These functions are grouped by their kind with respect to the classification made in chapter 3. Some of the testing Python code inherits from the previous version of the ePython source under the directory `/examples`. All the previous message passing functions has been extended without any syntax change, hence most of the existing parallel Python code can be directly transplanted onto the cluster. Because some of the cluster environmental native functions are complete new to the distributed ePython, a new test program was written and used for their test.

4.1.1 Environmental native functions

In this test, every core within the cluster prints its global core ID and node ID, the core with highest rank will also print the configuration of the cluster. The test code can be found in appendix A.1. The terminal output was shown as in listing 4.1.

```
$ epython cluster.py
[device 1] My global coreid: 1, I am on node0.
[device 2] My global coreid: 2, I am on node0.
[device 3] My global coreid: 3, I am on node0.
[device 4] My global coreid: 4, I am on node0.
[device 5] My global coreid: 5, I am on node0.
[device 6] My global coreid: 6, I am on node0.
[device 7] My global coreid: 7, I am on node0.
[device 8] My global coreid: 8, I am on node0.
[device 9] My global coreid: 9, I am on node0.
[device 10] My global coreid: 10, I am on node0.
[device 11] My global coreid: 11, I am on node0.
[device 12] My global coreid: 12, I am on node0.
[device 13] My global coreid: 13, I am on node0.
[device 14] My global coreid: 14, I am on node0.
```
As mentioned in section 3.1.1, The prototype Epiphany cluster has two Parallella nodes, each node contains 16 cores. The result shows that core 0-15 are located on node 0, core 16-31 are located on node 1, which are in accordance with the topology designed in section 3.1.1. The display messages of core 0 and core 16 fall behind their peers because there is an implicit barrier at the start of interpreter and they are the local master core which are the last ones exit their local barrier call. The display messages of node 1 fall behind those of node 0 because the ePython was invoked on the node 0 and the node 1 is a remote host managed by MPI, therefore the print messages were slightly delayed in the inter-Parallella network. All four environmental native functions numcores(), numnodes(), coreid() and nodeid() passed this test.

The -d optional argument has been tested as well. In particular, when the argument -d 16 is specified, the cluster will degenerate into a single Parallella board as shown in listing 4.2.

```bash
$ epython -d 16 cluster.py
[device 0] My global coreid: 0, I am on node0.
[device 1] My global coreid: 1, I am on node0.
...  
[device 15] My global coreid: 15, I am on node0.
[device 15] The cluster has 16 cores in 1 node(s).
```

Listing 4.2: Optional argument -d test output

### 4.1.2 Unidirectional send and receive

In this section, unidirectional message passing between two pairs of remote Epiphany cores were tested. In order to verify the capability of the ePython monitor in performing
the two-step mechanism introduced in section 3.2.1, a cross pattern for remote send and receive was designed as shown in figure 4.1.

Figure 4.1: Message passing pattern for send/recv test

The core 1 was commanded to send a integer value to core 18 which is the second core on the second Parallella, meanwhile the core 17 was commanded to send a real value to core 2 which follows the core 1 in the monitor request processing sequence. The test code can be found in appendix A.2. The terminal output for performing the remote cross message passing 1 time \((n=1)\) was shown as in listing 4.3.

```
$ epython remote_send_recv.py
[device 1] [core 1] sending value 20 to core 18
[device 2] [core 2] Got value 30.000000 from core 17
[device 17] [core 17] sending value 30.000000 to core 2
[device 18] [core 18] Got value 20 from core 1
```

Listing 4.3: Send and receive test output

The result shows that the monitor can not only carry out the two-step mechanism but also pass the correct value between remote Epiphany cores with the unidirectional send/recv operations. The remote unidirectional message passing functions `send()` and `recv()` passed the correctness test.

To evaluate the performance of the remote send/receive mechanism, the aforementioned cross message passing was executed 1, 100 and 10,000 times. The same test was also performed on one node so as to compare the performance between the local and remote point-to-point communication. In the local message passing test, the core 17 and 18 are replaced by the core 4 and 3 respectively. The runtime of each core can be obtained by passing `-t` argument to ePython. The result is shown in table 4.1.

The runtime of the slowest core involved in the message passing is regarded as the final communication time. The runtime for the remote P2P communication is significantly longer than the communication performed locally. This is more obvious in multiple consecutive message passing communication operations because for those tests with small \(n\), start-up overhead is dominant. The remote communication is relatively more time consuming simply because the messages inside the remote passing mechanism undergo more procedures, they need to be sent to the host and then transferred via LAN, moreover, on the remote host they still need monitor operation to be finally delivered to the target e-core.
Table 4.1: Comparison of unidirectional P2P message passing runtime between the local and remote communication

<table>
<thead>
<tr>
<th>n (times)</th>
<th>Local (s)</th>
<th>Remote (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.003594</td>
<td>0.004257</td>
</tr>
<tr>
<td>100</td>
<td>0.006101</td>
<td>0.029123</td>
</tr>
<tr>
<td>10,000</td>
<td>0.277485</td>
<td>2.545559</td>
</tr>
</tbody>
</table>

4.1.3 Bidirectional send-and-receive

In this section, bidirectional message passing between two remote Epiphany cores were tested. As shown in figure 4.2, the core 0 was commanded to swap a value in an array with the remote core 17.

![Figure 4.2: Data swapping in send-and-recv test](image)

The bidirectional send-and-recv operation was used for swapping the second value stored in the data array. The test code can be found in appendix A.3. The terminal output for this test was shown as in listing 4.4.

```bash
$ epython remote_sendrecv.py
[device 0] Core0 has value: [0.000000,0.000000]
[device 0] Core0 has value: [0.000000,1.000000]
[device 17] Core17 has value: [1.000000,1.000000]
[device 17] Core17 has value: [1.000000,0.000000]
```

Listing 4.4: Send and receive test output

Before performing the send-and-receive operation, the data array on the core 0 has value (0, 0), and the data array on the core has value (1, 1). Next, the send-and-receive function swapped the second entry in each data array. Then the second prints from each Epiphany core showed the correct values after swapping. The bidirectional message passing function `sendrecv()` passed the correctness test.

To evaluate the performance of the remote send-and-receive mechanism, the above data swapping was executed 1, 100 and 10,000 times. The same test was also performed on one node so as to compare the performance between the local and remote point-to-point communication. In the local message passing test, the core 17 is replaced by the core 1. The result is shown in table 4.2.

The runtime for the remote bidirectional P2P communication is significantly longer than the communication performed locally. For the same reason stated in the unidirectional
Table 4.2: Comparison of bidirectional P2P message passing runtime between the local and remote communication

<table>
<thead>
<tr>
<th>n (times)</th>
<th>Local (s)</th>
<th>Remote (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.003760</td>
<td>0.004940</td>
</tr>
<tr>
<td>100</td>
<td>0.006912</td>
<td>0.028902</td>
</tr>
<tr>
<td>10,000</td>
<td>0.356076</td>
<td>2.343262</td>
</tr>
</tbody>
</table>

communication test, the remote communication overhead is more obvious in multiple consecutive message passing communication operations because for those tests with small n, start-up overhead is dominant. Notice that the time of performing local message passing is different even though the amount of data transferred is equal in both the unidirectional and bidirectional tests, this is because the serial operations for the local send-and-receive is longer than that of the local send/receive, and in the unidirectional test all the cores only perform one of the send or receive operation per core.

### 4.1.4 Broadcast

In this section, the collective message passing function broadcast are tested. The Python code used is one of the existing example code comes with the ePython. It can be found in the ePython source code\(^\text{19}\) under /example/broadcast.py.

The test code will generate a random integer between 0 and 100 for each active Epiphany core and broadcast the integer hold by core 0. Every other active e-core will then print the broadcasted value.

The test result showed that every core within the cluster can print the same value as the core 0. The value varies from time to time because it is generated randomly. Therefore, the collective message passing function bcast() passed the correctness test.

To evaluate the performance of the broadcast mechanism, the test code was executed by varying number of cores from 1 to 32. The result is shown in figure 4.3.

When the core count is smaller than 17, the broadcast only happens on one Parallella node, and the root core broadcast data via fast eMesh NoC. This explains why the runtime of the broadcast is overall smaller with core number under 17. Because the root core issues the send in a sequential manner, the local broadcast time increases with the number of cores. The broadcast involving remote communication via LAN caused a surge in runtime when core count exceeds 16. However the declining pattern of the remote broadcast runtime between 17 and 32 cores is still not completely understood.

As a speculation, this might have something to do with the underlying MPI when the unbalanced workload among nodes could potentially lead to excessive delay. The host processor might have scheduled more cycles on the blocking MPI call when the monitor is relatively idle and this might cause the monitor tasks did not get processed in time.
4.1.5 Reduction

In this section, the collective message passing function reduce were tested. Because the reduction mechanism contains explicit global barrier synchronisation, the functionality check for `sync()` is also included in this test. The Python code used for testing is one of the existing example code comes with the ePython. It can be found in the ePython source code\textsuperscript{[19]} at /example/reduction.py. The test code will generate a random integer between 0 and 100 for each Epiphany core and find the maximum among all Epiphany cores. Every Epiphany core will then print the reduced value.

The value varies from time to time because it is generated randomly. The test was conducted several times for different total active cores and the result showed that the probability of getting large number is higher when the active core count is bigger. The test code was also slightly modified to test other reduce operations, the reduced value mostly scattered around the region very close to their expected values. Therefore, the collective message passing function `reduce()` passed the correctness test.

To evaluate the performance of the reduction mechanism, the test code was executed by varying number of cores from 1 to 32. The result is shown in figure 4.3. The reduction costs a slightly longer time than the broadcast no matter how many e-cores was used. This happened because the reduction also performs reduction calculations in addition to the communication.Same as the pattern shown in the broadcast, the surge in the runtime when the core count exceeds 16 is caused by message passing without using the fastest eMesh, and the downward tendency of the remote reduction runtime has not been completely understood. As described in section 3.2.1, the monitor cyclically processes the e-cores’ requests, this the source of the uncertain behaviour of the remote communication especially when the communication pattern is complex. As a result, the runtime for programs involving remote communication might dramatically vary as proven by the reduction runtime fluctuation between 17 and 32 cores in the figure 4.3.
4.2 Benchmark

Although the performance is not the first consideration of this project, it is still worth carrying out some comprehensive performance tests on the prototype cluster to see how competent the distributed cluster is when using ePython to handle practical tasks.

The existing example code comes with the ePython for solving one dimensional Laplace’s equations\textsuperscript{[20]} was used here as performance benchmarks. The test Python code can also be found in appendix A. It contains both the point-to-point communication calls and the collective communication calls. Moreover, it is this program that was chosen for the performance test not only because the its diverse communication calls can demonstrate the robustness of the distributed ePython, but also because it can be a complement of the benchmark program used in the original introductory ePython paper\textsuperscript{[21]}, where the Gauss-Seidel method was used instead to solve the same problem.

4.2.1 Strong scaling test

The strong scaling describes how the running time varies with the number of cores when the total problem size in the testing program is fixed. The benchmark program for this test uses geometric decomposition to fulfil parallelisation in solving one dimensional diffusion equations. The numerical solving with Jacobi iteration algorithm\textsuperscript{[22]} was fixed at 128 and 256 equidistant discrete points within the 1D area. The number of cores used started from 1 and was incremented all the way up to 32. In each run, the residual converged below critical point 1e-4. The test results are shown in figure 4.4.

The solution for problem size of 128 takes 12521 iterations to converge and for the test
with problem size of 256, the number of iterations rise to 36616. With correct halo
swapping, as long as the granularity is fixed the iteration number will not change with
the varying core count. This is another indirect method to validate the correctness of
the communication functions. When the number of cores is fewer that 17, the program
is run by one Parallella node. For the test of problem size 128, with the increase of
core count, the run time decreases from its peak core count equals one to its global low
core count equals 10. After exceeding 10 cores, the program reached strong scaling
limit and could no longer benefit from parallelism, because the local problem size is too
small to counter the increase of the communication overhead. For the test of problem
size 256, the communication runtime decline can be found between 17 and 20 cores
even though the overall runtime has been increased due to the invoking of the second
Epiphany chip. The graph shows a predictable bumping up when the core count is
increased to 17. Because from this point the inter-Parallella communication appears,
and the messages start being passed between two remote monitor processes via LAN.
Based on the running time data, the parallel efficiency was calculated. The results are
shown in figure 4.5.

Figure 4.5: Parallel efficiency for test on prototype cluster with Jacobi benchmark

Notice that the pattern of the parallel efficiency curve are different between the sections
where the programs have not reached the strong scaling limit and where the programs
have reached the limit. As shown in figure 4.4 the strong scaling limit for program
with problem size of 128 is 10 where as for program with problem size of 256 that is
20. Before the strong scaling limit core count, the program can still benefit from the
parallelisation, the declining speed is slower than what function 1/x would be. And we
could even see a slightly upward convex in the beginning of the 256 problem size line.
After the strong scaling limit core count, the programs are no longer able to benefit from
the parallelisation and the declining speed of the parallel efficiency becomes close to
what function 1/x would be and even worse. Same as the runtime patter shown in figure
4.4, the sudden decrease of the parallel efficiency at core count equals 17 means the inter-node communication arose. This happened because the mechanism of collective communication inside the multinode Parallella cluster has been fundamentally altered from its one node version. The monitor relaying phase and the standard monitor-device interaction in the remote communication take significantly longer time than the local communication using eMesh.

### 4.2.2 Weak scaling test

The weak scaling describes how the running time varies with the number of cores when the problem size per core is fixed in the testing program. The benchmark program for this test uses the same one as in strong scaling test except for that the total problem size is changed in accordance with the total cores used. In this test, no matter how many cores are used in solving the problem, each core has a subdomain of 4 points in size, which means the calculation overhead will be same for all e-cores and the only thing influences the varying time of the test program is the communication runtime change caused by the varying number of cores. This test indicates exactly how the communication overhead impacts the overall performance of the program.

The the results for total solving time are shown in figure 4.6, and the the results for running time per iteration are shown in figure 4.7.

![Figure 4.6: Weak scaling test on prototype cluster with Jacobi benchmark](image)

As in figure 4.6, the running time increase with the number of cores and, as expected, there is a bumping up at core count equals 17 due to the remote communication. We should mention that, in this program the number of iterations to converge varies with the total problem size, they were recorded to obtain the running time per iteration. In order to filter out the influence posed by the calculation with different iterations and
retain purely the communication overhead, the time needs to be divided by the number of iterations respectively. Most communication happens inside the iterations, and since each core has same amount of calculation workload, the changes of time per iteration from different core counts indicate the varying time for communication.

The figure 4.7 shows that the communication overhead per iteration increased linearly in both sections. The surge at core count equals 17 is about 1 millisecond which is contributed by a remote send-and-receive operation and a remote reduction operation. But wait, the reduction runtime as shown in figure 4.3 does not increase linearly when the remote communication exists and no matter how the total number of cores are used each core only swaps halo with its two neighbours which means that the point-to-point communication seems not to be contributing to the communication, and how does the communication overhead per iteration grow in such a regular pattern? The answer is the send-and-receive used for the halo swapping in the benchmark program. As shown in listing A.5, in lines 38-39, each core send-and-receives with its left neighbour in the first step except for the leftmost core. Because their left neighbours have not posted the receive with the right neighbours yet, all cores except for the core 0 will wait. Then the core 0 will issue a rightward send-and-receive and unlock the core 1 from its first step, then core 1 can perform the following rightward send-and-receive to unlock core 2, and so on. The message passing is performed consecutively across the active cores rather than happen simultaneously on all cores, hence the span of the point-to-point communication is proportional the total cores used. This is the reason for the linearly growing communication runtime per iteration.
Chapter 5

Conclusions

The project has proven that multiple many-core architecture processors can cooperate with each other in interpreting parallel Python code. The ePython has been successfully extend to be the only technology that supports the programming on the interconnected Epiphany with Python. Moreover, the extended ePython is thus far the most simple-and-easy-to-use developing tool on the Parallella cluster.

In this project, the ePython has been equipped with a reliable interconnection module that enables stable underlying data transfer among remote Epiphany cores in various communication patterns, including the blocking uni/bidirectional point-to-point message passing, the barrier synchronisation and some basic collective communication such as broadcast and reduction. The extended ePython has an excellent scalability. It can be used to expand the number of concurrent cores available in parallel programming by assembling and configuring a Parallella cluster within few steps. The extended ePython also takes the usability into consideration. Now the original parallel Python can be interpreted by the extended ePython without any syntax changes or language extensions. The design of the distributed ePython makes it capable of operating on a massive scale Parallella clusters and managing a large number of Epiphany cores.

The communication latency among cores located in different Epiphany processors is significantly higher than that of the cores located within the same Epiphany processor. This has an impact on the overall performance of the ePython to some extent when running on an Epiphany cluster. Nevertheless, the performance decreasing is acceptable since the aim of this project is to extend the number of the cores which can run explicitly parallel programs from software engineers’ prospective.

The technology developed in this project has opened up a whole new prospect for the parallel programming on not only the Epiphany but also other many-core architecture processors such as Kalray MPPA[23] and XMOS xCore[24]. Last but not least, if the technology had been developed two years earlier, it could have become a cornerstone of the Supercomputer.io project and its ending might have been reversed.
5.1 Future work

The project used a two-node Parallella cluster as the platform for development work. Although the programming of ePython has fully supported any number of Parallella boards with 16-core Epiphany processors, the actual scaling of the cluster has never been carried out. It is worth to build a lager prototype cluster to test the limits of the distributed ePython.

The ePython currently can only support the Parallella cluster with homogeneous nodes. Considering that there are several kinds of more advanced Epiphany processors which contains more e-cores, without much work, the ePython could be extended further to support the Parallella cluster with heterogeneous nodes.

As mentioned in section 4.1.4 and 4.1.5, the underlying MPI might have caused the downward tendency of the communication runtime when the core count exceeds 16. This is still a mystery and worth investigating. Moreover, the indeterministic nature of monitor might cause many interesting phenomena during the remote communication. This could also be studied.

Due to the limited time, the taskfarm has not been well supported, despite the fact that it is essentially based on the point-to-point communication which has implemented in this project. This is because the support Python library for taskfarm is too large to fit into the e-cores’ small local memory, the previous version of ePython store the code in main memory but this has raised some issues that cause the taskfarm module does not work normally on the remote work allocation. This should be fixed in the future.

The last thing needs to mention is the memory overflow issue encountered in section 3.5.2. Obviously the Epiphany runtime function should be optimised to make more used of the local memory especially when there might be more functionalities to be added on the ePython.
Appendix A

Python test code

A.1 Cluster configuration test code

```python
from parallel import *

print "My global coreid: " + str(coreid()) + ", I am on node" + str(nodeid()) + "."

if coreid() == (numcores() - 1):
    print "The cluster has " + str(numcores()) + " cores in "+str(numnodes())+" node(s)."
```

Listing A.1: cluster.py

```python
0000000 00a9 0000 0001 0223 0001 0a0a 0a0a 4d11
0000010 2079 6c67 626f 6c61 6320 726f 6965 3a64
0000020 0020 a11b 0000 1100 202c 2049 6d61 6f20
0000030 206e 6f6e 6564 1b00 0091 0000 2e11 1300
0000040 1b04 00a1 0000 1b00 099 0000 0112 0000
0000050 3b00 2300 0102 0a00 0a0a 110a 6854 2065
0000060 6c63 7375 6574 2072 6168 2073 1b00 0099
0000070 0000 2011 6f63 6572 2073 6e69 002e 0001
0000080 1c00 1b23 0000 001a 1c00 1c23 0000 001a
0000090 1c00 0e23 0000 001a 1c00 0f23 0000 001a
```

Listing A.2: Byte code representation of cluster.py

A.2 Unidirectional send and receive test code

```python
from parallel import *

n=1
i=0
if coreid() == 1:
    val=20
    print "[core 1] sending value " + str(val) + " to core 18"
```

47
while i<n:
    send(val, 18)
    i+=1
if coreid()==18:
    val=0
    while i<n:
        val=recv(1)
        i+=1
    print "[core 18] Got value " + str(val) + " from core 1"
if coreid()==17:
    val=30.0
    print "[core 17] sending value " + str(val) + " to core 2"
    while i<n:
        send(val, 2)
        i+=1
if coreid()==2:
    val=0
    while i<n:
        val=recv(17)
        i+=1
    print "[core 2] Got value " + str(val) + " from core 17"

Listing A.3: remote_send_recv.py

A.3 Bidirectional send-and-receive test code

Listing A.4: remote_sendrecv.py

A.4 Jacobi iteration benchmark code
from math import sqrt

DATA_SIZE=128
MAX_ITS=100000

# Work out the amount of data to hold on this core
local_size=DATA_SIZE/numcores()
if local_size * numcores() != DATA_SIZE:
    if (coreid() < DATA_SIZE-local_size*numcores()):
        local_size=local_size+1

# Allocate the two arrays (two as this is Jacobi) we +2 to account
for halos/boundary conditions
data=[0] * (local_size+2)
data_pl=[0] * (local_size+2)

# Set the initial conditions
i=0
while i<=local_size+1:
    data[i]=0.0
    i+=1

if coreid()==0: data[0]=1.0
if coreid()==numcores()-1: data[local_size+1]=10.0

# Compute the initial absolute residual
tmpnorm=0.0
i=1
while i<=local_size:
    tmpnorm=tmpnorm+(data[i]*2-data[i-1]-data[i+1])^2
    i+=1
tmpnorm=reduce(tmpnorm, "sum")
bnorm=sqrt(tmpnorm)

norm=1.0
its=0
while norm >= 1e-4 and its < MAX_ITS:
    # Halo swap to my left and right neighbours if I have them
    if (coreid() > 0): data[0]=sendrecv(data[1], coreid()-1)
    if (coreid() < numcores()-1):
        data[local_size+1]=sendrecv(data[local_size], coreid()+1)

    # Calculate current residual
tmpnorm=0.0
    i=1
    while i<=local_size:
        tmpnorm=tmpnorm+(data[i]*2-data[i-1]-data[i+1])^2
        i+=1
tmpnorm=reduce(tmpnorm, "sum")
norm=sqrt(tmpnorm)/bnorm

    if coreid()==0 and its\%1000 == 0: print "RNorm is "+norm+" at "+its+" iterations"
# Performs the Jacobi iteration for Laplace
i=1
while i<=local_size:
    data_p1[i]=0.5* (data[i-1] + data[i+1])
i+=1

# Swap data around for next iteration
i=1
while i<=local_size:
    data[i]=data_p1[i]
i+=1
its+=1

if coreid()==0: print "Completed in "+str(its)+" iterations, RNorm="+str(norm)

Listing A.5: jacobi.py
Appendix B

The prototype cluster

In this project, two Parallellas are connected to the Ethernet switch of a large machine named Hydra, which is also a cluster at EPCC. The Hydra plays the role as the Inter-Parallella Network router and provides Internet access to both Parallellas. Operating the Parallella cluster requires remote access with Secure Shell (SSH). Since Hydra also runs Linux, the actual access procedure is not complicated, simply using SSH twice, first ssh to Hydra from anywhere in the Internet and then ssh to any one of the Parallellas from Hydra.

B.1 Mapping of the host names

In order to allow the users or MPI to referring to the host in the shell script command by just using their names rather than the confusing IP addresses, maps between host names and addresses can be made by inserting the following lines in listing A.1 to the file /etc/hosts.

```
#PARALLELLA CLUSTER SETUP
172.16.29.58 parallella0
172.16.29.56 parallella1
```

Listing B.1: Address mapping of the Parallella nodes

B.2 Setting the passwordless SSH

The passwordless SSH can be enabled by adding one user’s public key to another remote user’s SSH authorized_keys list. A RSA key pair was generated on master node parallella0 by issuing

```
$ ssh-keygen -t rsa
```
Then the pub-key need to be added to remote machines to enable passwordless ssh, to do this issue

$ ssh-copy-id parallellal

Then the master node will be able to ssh to the slave node with out password. This operation should also be carried out on the slave node to help the slave node perform passwordless ssh to master node. If the cluster contains more than one slave node, the passwordless ssh should be enabled on each master-slave pair.
Appendix C

Files changed in ePython source

Modified files

README.md
epython.sh
device/device-functions.c
host/byteassembler.c
host/byteassembler.h
host/configuration.c
host/configuration.h
host/device-support.c
host/lexer.c
host/main.c
host/makefile
host/memorymanager.c
host/misc.c
host/parser.c
interpreter/basictokens.h
interpreter/interpreter.h
makefile
modules/parallel.py
shared.h

Added files

testPyCode/cluster_env.py
testPyCode/cluster_jacobi.py
testPyCode/rm_p2p_Sendrecv.py
testPyCode/rm_p2p_msgPassing.py
.mpi_hostfile
Bibliography


[12] Parallel Processing Group, University of Ioannina 2015 OMPi compiler http://paragroup.cse.uoi.gr/wpsite/software/ompi/

[14] Adapteva Inc. 2015 Supercomputer.io Project Proposal
https://www.parallella.org/2015/05/25/help-us-build-the-worlds-most-efficient-supercomputer/


