POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

COMPUTER SCIENCE AND ENGINEERING
MASTER'S THESIS

BEER: AN UNIFIED PROGRAMMING APPROACH FOR DISTRIBUTED EMBEDDED PLATFORMS

Author:
dott. Domenico Iezzi

Student ID (Matricola):
850623

Supervisor (Relatore):
Prof. William Fornaciari

Co-Supervisor (Correlatore):
Ph.D. Giuseppe Massari

A.Y. 2017/2018

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thanks my supervisor *prof. William Fornaciari* and co-supervisor *Giuseppe Massari* for their time spent guiding me from the first steps of the project to writing this thesis. I would also like to thank *Michele Zanella* and *Federico Reghenzani* for their valuable advice and help during the development of this thesis, and also all the *HeapLab* people for the great time and knowledge they shared with me.

A special thank goes to my parents *Guido* and *Grazia*, without whom I could not start this journey, and which supported me during all this time.

And last but not least, a special thank to my love *Silvia* which has endured me during these years of study, providing a great encouragement and helping me overcome moments of discomfort.

Sommario

Con l'introduzione di processori multicore, i sistemi hanno cominciato a sfruttare questo parallelismo per raggiungere livelli di prestazione sempre più elevati. Inoltre la creazione di dispositivi sempre più potenti ma allo stesso tempo più piccoli ha portato alla proliferazione di dispositivi embedded in molti contesti. Nuovi paradigmi di computazione hanno permesso di sfruttare le risorse inutilizzate di questi dispositivi interconnessi in una rete per poter eseguire piccole parti di computazioni lunghe e complesse, in modo da dividere il carico di lavoro su più dispositivi invece che una singola macchina. Questo approccio ha portato con se nuove sfide nel campo della gestione delle risorse e soprattutto il bisogno di un modello di programmazione adatto per il calcolo distribuito.

Questa tesi propone un framework per la computazione distribuita su sistemi embedded chiamato *BeeR*, che va ad integrarsi con il progetto MANGO, il quale fornisce un framework per applicazioni in ambito High Performance Computing. Sfruttando il modello di programmazione di MANGO, *BeeR* permette ad una applicazione client in esecuzione su un nodo General Purpose di un sistema basato su microprocessore di affidare parte dei task che compongono un applicazione e dei buffer di dati a dispositivi remoti, ed effettuare operazioni quali leggere e scrivere buffer, rimanere in ascolto di particolari eventi e di avviare la computazione. Inoltre, è stato possibile valutare sperimentalmente il progetto implementando un test da una suite di benchmark paralleli, e facendo prove sperimentali con differenti configurazioni di dispositivi e livelli di parallelismo.

Abstract

With the introduction of multicore CPUs, systems began to exploit the parallelism to achieve never seen before performance and concurrency. Moreover, the miniaturization of increasingly powerful and less energy systems led to the proliferation of embedded devices in many contexts. New computing paradigms began exploring the possibility of exploiting idle resources of mobile devices and smart objects connected through a network, by offloading to them parts of a complex computation. This approach introduced new challenges in terms of resource management of heterogeneous systems, and the need of a common programming model suitable for distributed computing.

This thesis presents a framework for distributed computation on embedded systems called *BeeR*, which integrates into the MANGO project providing a framework for High Performance Computing applications. By leveraging the MANGO programming model, *BeeR* allows a client application running on a General Purpose CPU-based node to offload part of its tasks and data buffers to a remote device, where an instance of the daemon is running, perform operation like reading and writing buffers, waiting for specific events and running the computation. Moreover, experimental evaluation was carried out by implementing a test application from a suite of parallel benchmarks, and running it considering different configurations of devices and levels of parallelism.

CHAPTER 1

Introduction

1.1 Trend

When the modern computer era began around 1985, computers were bulky and expensive. These computers operated independently from one another, without networks to connect them. With the development of increasingly powerful and smaller microprocessors, we assisted also to a miniaturization of computing machines, leading to the possibility of integrating complex digital systems into battery-powered devices, with performance that in some cases can be comparable to full-fledged computing systems. The most impressive outcome of this trend is the so-called *mobile device*, defined as highly portable battery powered system, including a screen, some sensors, wireless adapters and a software stack. Mobile devices are becoming more and more popular, reaching levels of pervasiveness never seen, to the point of influencing the lifestyle of an individual. Moreover, this evolution of computing systems has led to the definition of new computing paradigms, capable of exploiting the huge number of heterogeneous devices connected to the internet. In the following chapters, this new technologies are described, with an overview of the challenges they introduced and how this thesis contributes to the overall field.

1.2 Emerging Technologies

Thanks to their decreasing size, price and energy, electronic components are being increasingly integrated into everyday objects. Through sensors, these objects are able to perceive their context, and network capabilities allow them to interact with each other and access the Internet. The fact that devices may be equipped with network equipment allows to "observe" the surrounding world with a good level of detail but with a negligible cost. The increasing pervasiveness of mobile devices has led to the definition of new computing paradigms leveraging this opportunity

The concept of a networked smart object appeared for the first time in 1982, where a coke machine at Carnegie Mellon University was modified to be able to report the inventory and information about the state of products. The term "Internet of Things" (IoT) was coined for the first time by Peter. T. Lewis in 1985, in a speech given at a U.S. Federal Communications Commission (FCC), indicating the need to extend the connectivity beyond standard devices such as personal computers or mainframe, to "dumb" or non-internet-enabled physical devices and everyday objects. In the following years it was popularized by many, such as the Kevin Ashton in a presentation made at Procter & Gamble (P&G) in 1999 [1], as reported in a 2002 article by C.R. Schoenberger [2].

In practice, the Internet of Things must not be considered as a single technology, but the result of several technologies enabling different capabilities, such as:

- *Communication*: devices are able to connect to the Internet, or talk to each other in a network. Technologies such as WiFi, Bluetooth, GSM, UMTS are particularly useful in this field.
- *Addressability*: objects can be located though discovery, look-up services, thus remotely configured.
- *Identification*: objects are uniquely identifiable, technologies like RFID or NFC allow even passive object with no power source to be identified.
- *Sensing*: object can extract information from their surroundings through sensors, but also record information or simply react to specific events.
- *Actuation*: devices use actuators to manipulate the environment. Devices can be used to control real world elements through the Internet.
- *Information processing*: even without exceptional computation power and

storage, devices can compute information coming for example from sensors, or they may store information about state so that they can restore it.

- *Localization*: with technologies such as GPS or the mobile phone network it is possible to get a precise location of the device.
- *User Interfaces*: user can interact with smart objects through an appropriate interface. Also innovative technologies such as image or voice recognition could be used.

These capabilities opens up fascinating prospects and interesting application possibilities; but they are also come with substantial requirements relating to the underlying technology. Indeed the infrastructure for an Internet of Things must not only be efficient, scalable, reliable, secure and trustworthy, but it must also be widely applicable and must take economic considerations into account.

Today, the increasing pervasiveness of IoT and mobile devices has led to an huge increase in the data produced. In 2017, there were 8.4 billions of connected IoT devices according to [3], and this number is going to grow again, with a forecast of 20.4 billions devices connected by 2020. Moreover, the spread of service based software has led to the definition of cloud-assisted internet of things (CoT), a paradigm that defines self-configuring smart devices connecting to cloud based services through the Internet. The amount of data collected by these device cannot be processed by devices themselves, because of their limited physical resources, and the offloading to the cloud becomes necessary. However, this influx of data is pushing network bandwidth requirements to the limit, thanks also to the high transfer rates made possible by improving technologies such as WiFi. Since moving this data from devices to data centers is becoming more and more costly, the industry began exploring different alternatives to cloud computing. Most of them are based on the idea of shifting towards a distributed approach where devices on the edge of the network, such as smartphones or smart gateways will offer cloud services for a subset of the devices.

The paradigm of *Fog Computing* [4] proposes a highly virtualized platform that provides computing, storage and networking services between devices and the cloud computing data centers, located mainly at the edge of the network logically close to devices. The name "fog" comes from the fact that fog can be seen as a cloud closer to the ground. There are a number of characteristics that a fog platform should take into account. First, the geographical distribution plays a key role in the deployment of fog solutions, since application and services will benefit from highly distributed content; for example, it could be possible to provide high quality streaming to moving vehicles through access points and proxies

positioned along its route. Moreover, since these platform should communicate with mobile devices, it should support mobility techniques enabling the decoupling of host identity from location identity. To this aim, several solution were proposed. One of this is the Follow Me Edge [5] advanced from the Follow Me Cloud [6], which proposes an architecture integrating different modules whose goal is to manage nodes mobility and thus also the service migration. Originally designed for user's mobility only in cellular networks, it exploits the Locator/ID Separation Protocol (LISP) virtualized according to the Network Functions Virtualization (NFV) principle. Furthermore, services that need seamless support require also the cooperation between resource providers, hence the need for interoperability and federation between fog instances.

In a similar way, the *Edge Computing* paradigm tries to move part of the computation on the edge of the network, shifting the focus on the devices rather than infrastructure. To be more precise, the Edge Computing [7] paradigm refers to the enabling technologies allowing the computation to be performed at the edge of the network. This means that the computing should happen in proximity of data sources, using devices and gateways that are logically close to the edge of the network. As we can see in Figure 1.1, devices shift from data producers, to data producers and consumers: they not only request service and content from the cloud, but also perform computing tasks for the cloud. Edge provides feature such as computing offload, data caching and processing, feature typically performed by cloud systems in a centralized architecture, but also distribute requests and delivery service from cloud to user in a transparent way. This changes in the paradigm introduces also additional requirements in the reliability, security and privacy fields, as well as resources management [8].

Some key enabling technologies and research topics that will define the future of edge cloud systems has been proposed: to build a small-scale cloud platform at the edge and enable future IoT applications, Network Function Virtualization (NFV) and Software Defined Networking are the two key solutions. NFV is a network architecture concept that utilizes virtualization technologies to manage core networking functions via software rather than hardware; it is possible to create full-scale networking communication services by combining building blocks of Virtualized Network Functions (VNF). The approach of Software Defined Networking is to decouple network control and forwarding functions, enabling network control to become directly programmable and the underlying infrastructure to be abstracted from applications and network service. The separation of control from the data forwarding and the usage of centralized network control and configuration could greatly increase the flexibility of VNF and reduce the

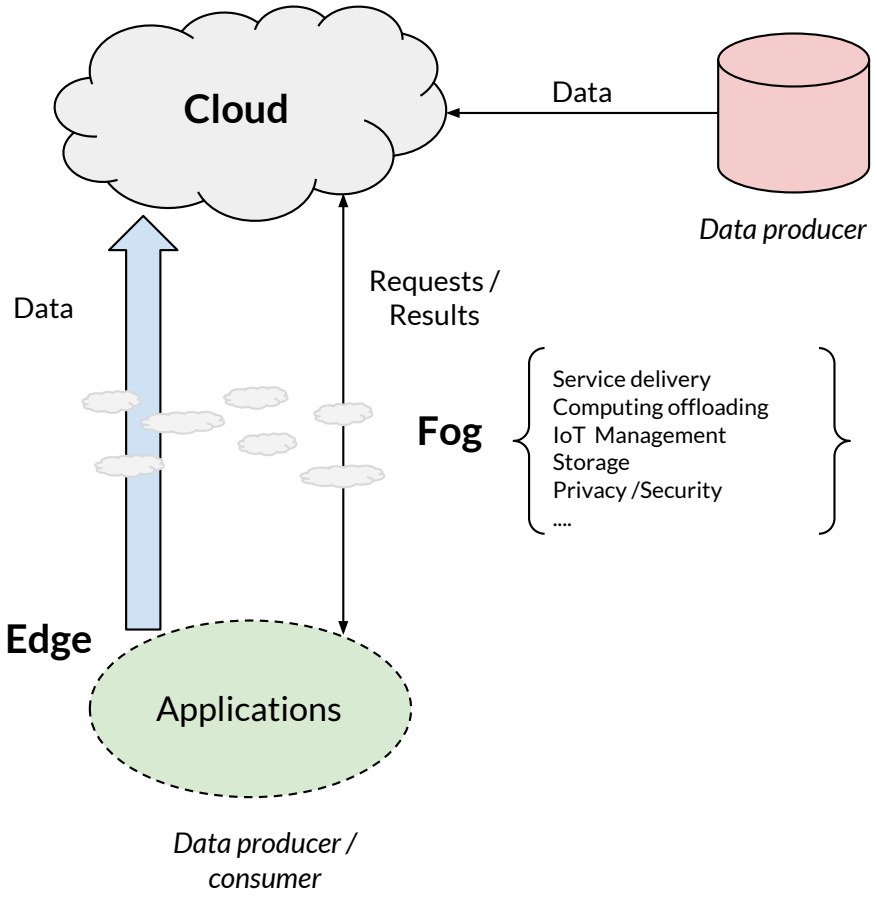


Figure 1.1: *Edge computing paradigm*

costs. Another important effort is to bring the cloud orchestration to the edge cloud. Orchestration is defined as a set of procedures and operations that the cloud providers and application owners undertake to manually or automatically select, deploy, monitor, and control the configuration of hardware and software resources for application delivery [9]. Existing cloud solution are not applicable to edge cloud technologies, since usually they are application-specific and highly customized for a specific type of application, therefore there is a demand for automated tools and abstraction to reproduce orchestration schemes on the edge.

1.3 Emerging Needs

The introduction of parallel and distributed computing paradigms allowed to tackle the increasing computing demand and exploit the pervasiveness of embedded systems. Devices themselves began to leverage multi/many core archi-

tectures, enabling the concurrent execution of different tasks at the same time. Platform complexity is growing, to the point where processing elements are grouped into clusters, memories are organized into several layers, and component communication made possible through Network-on-Chip (NoC) and buses. This introduces a number of resource management and application scheduling challenges.

With *resource allocation* or *scheduling* we refer to the problem of making an efficient assignment of resources to applications. It usually follows a *scheduling model*, based on a *system model* which is an abstract representation of the underlying resource. Especially in a High Performance Computing context, where applications have tight constraints, the allocation of resources became a crucial part of a computing framework. Moreover, having the hardware distributed across the network introduces additional complexity, because of the dynamicity of a smart object environment and the volatility of those devices.

Finally, to provide an easy to use interface, but at the same time allow system engineers to exploit all the available components, a solid programming model suitable for distributed computation should be adopted.

1.4 Thesis Contribution

The work described in this thesis has the goal of exploring the feasibility and performance of computation offloading in a context of distributed embedded systems, like the aforementioned Edge/Fog platform based scenarios. In particular, the key concept is to seamlessly exploit the availability of computing devices, located at different levels and exhibiting heterogeneous capabilities, to distribute the workload according to the application requirements. All this, without explicitly managing the task offloading at application code level, but by using an already proposed programming approach for multi-tasking HPC applications (MANGO), integrated with a run-time resource management solution (the BarbequeRTRM). In this thesis, this approach has been properly extended to this work on distributed embedded systems, and experimentally evaluated on real devices.

The thesis is structured as follows. In Chapter 2 the state-of-the-art related to distributed computing and novel edge/fog technologies are highlighted. Chapter 3 introduces the MANGO project and its software stack, with the programming model and the underlying run-time resource manager, the BarbequeRTRM. The proposed solution, extending the MANGO software infrastructure, is carefully explained in Chapter 4, with an in depth description of its design and implementation, while Chapter 5 and Chapter 6 describe the benchmark application

implemented and the experimental tests conducted. Finally, in Chapter 7 future improvements and extension of the solution are summarized, with a focus on crucial missing features such as authentication and communication encryption, as well as a tighter integration with the BarbequeRTRM.

CHAPTER 2

State Of The Art

This chapter introduces the state of the art in the field of distributed computation, considering many types of systems and architectures. Section 2.1 introduces the most important paradigms in the distributed computing field. In Section 2.2 an overview of the different solutions from the industry for the mobile area is presented, with a description of the most popular implementations. Finally in Section 2.3 a brief look at the new edge/fog alternatives proposed in recent years.

2.1 Distributed Computing paradigms

Distributed Computing is a paradigm according to which a single application can be developed to execute on multiple system nodes, connected though a shared network. For many years, Distributed Computing has been the exclusive approach to address the scalability requirements of scientific computing. This concept has been recently evolved into several paradigms. Here below, we can list a set of worth to consider ones:

- Peer-to-Peer Computing
- Cluster Computing
- Utility Computing

  - Cloud Computing
  - Grid Computing

- Jungle Computing

2.1.1 Peer-to-Peer Computing

The Peer-to-Peer (P2P) networking was introduced to address the scalability problem inherent in the distribution of resources among a high number of processes. In particular, there is no client-server structure in a P2P network, because each peer is both a client and a server, providing resource to all other nodes and asking other nodes for resources. There is no central structure keeping a representation of the entire system: this means that peers do not have a global view of the entire system, but only of a subset of it needed to acquire a specific set of resources.

2.1.2 Cluster Computing

Cluster computing consists of a set of independent and stand-alone computers, interconnected through a network, working together as a single integrated computing resource. They are connected through fast local area network, and all the nodes are usually supervised within a single administrative domain. When building a cluster computing network, different approaches may be adopted: *high-availability clusters* (also called *failover* clusters) try to improve the availability by having redundant nodes, which are used when some components fail, thus eliminating single points of failure. Such types of cluster are able to detect software and hardware fails, immediately restarting the service on another system without requiring administrative intervention.

In *load-balancing clusters* configurations, cluster nodes share and computational work loads to achieve better overall performance, like a web server cluster which may distribute queries to every node in order to optimize response times. *Software-based load balancing* cluster consists of a software installed on servers, accepting requests and dispatching them to different nodes. Node is chosen based on a specific algorithm, like a simple round-robin, or a complex one considering server affinity. On the other hand, it is possible to find *hardware-based load balancing* clusters , where the balancing logic is handled by a specialized router or switch.

Clusters are also used in the *high-performance computing* world, where the large-scale cluster can be seen as a compute farm: instead of running task locally, it is possible to submit it to the cluster, which is able to manage tasks through

a work queue, and execute them as soon as resources are available. Users who need to run many similar jobs with different parameters or data sets find clusters the ideal solution, to speed up the computation in case of big tasks that would require a great amount of time running in a local workstation, like for example complex scientific calculations.

2.1.3 Utility Computing

Utility Computing takes is name from an analogy derived from the real world, where service providers maintain supply utility services like electrical power, water and gas are delivered to consumers. This paradigm is based on a service provisioning model, because users pay providers for using computational power only when needed. Eventually, utility computing evolved into two main paradigms which became popular in the IT field: *cloud computing* and *grid computing*.

Grid Computing aim is to enable coordinated and distributed resource sharing and problem solving in dynamic virtual organizations. Like electrical power grid, a computing grid offers an infrastructure that couples computers, middleware software, connected across LAN, WAN or the Internet forming networks at regional, national or global scale. One of the most important solution in this field is the Berkeley Open Infrastructure for Network Computing (BOINC) [10] which is an open-source middleware system for grid computing, on a volunteer basis. It aims to exploit the processing power of personal computers around the world, by allowing users to select the project to contribute to. In particular, users can install the provided application, register for a project, and receive tasks from the project's scheduling server. The executable and input data are downloaded, the client software runs the program producing output files, which are then uploaded back to the project's server. The server keeps track of how much work each computer has done, with a credit-based system: each task is sent to two or more clients; when client uploads results, it claims a certain amount of credit to the server. The server then check if the results agree and assign to both clients a number of credit equal to the lowest reported claim. This approach, described in figure 2.1, has also the advantage to protect against false result reports.

Cloud Computing consists of shared pools of configurable computer system resources and higher-level services that can be provisioned in an on-demand way, with little or no up-front IT infrastructure investment costs. Cloud applications and services offer high scalability which can be achieved by cloning tasks onto multiple virtual machines orchestrated at run-time to meet changing work demand, with the help of load balancers distributing the work over the set of virtual

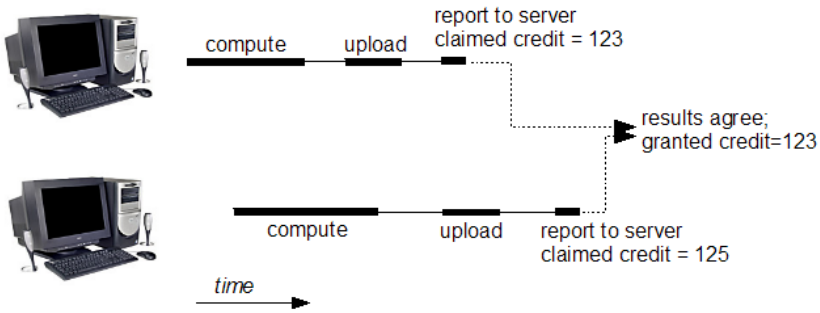


Figure 2.1: *Credit based system, in which each client reports results and claims a credit, while server checks for their validity and assigns credits*

machines. This process is transparent to the cloud user, who sees only a single access-point, and provides benefits for the cloud providers, which can scale the service based on its demand. Services offered in cloud computing follow different models:

- *Software as a Service* where users access an application and database, usually through a thin client like a web interface or a program interface.
- *Platform as a Service* where users gain access to a computing platform where they can deploy their product.
- *Infrastructure as a Service* where users are able to deploy and run arbitrary software, which can include operating systems and applications. Users have full control over operating systems, storage, and deployed applications thanks to virtualization software, which allows the separation between the service and the underlying infrastructure.

2.2 Mobile distributed computing

The increasing computing resources of mobile devices is allowing them to perform performance-hungry computation and running many multi-tasking applications without problems. These capabilities aren't usually exploited by the device software, leading to a waste of computing power. Moreover, the planned obsolescence policy imposed by the market gives an average life for a mobile device of 20 months, leading to huge numbers of devices being replaced even if they are perfectly working. This pervasiveness of mobile devices has led to the definition of new solutions leveraging mobile devices as a distributed computation network. Thanks to wireless communication improvements, researchers defined

new computing paradigms for distributed systems of mobile devices, which can be divided in the following categories:

- Transparent Computing
- Flexible Computing
- Voluntary Computing
- Enterprise Computing
- High-Performance Computing

2.2.1 Transparent Computing

The idea of *Transparent Computing* is that all data and software, including the operating system and user information, should be stored on servers, while data is computed on terminals. A small number of centralized servers act as warehouses storing software resources, and bare clients interacts with users, where they can chose heterogeneous OSes and applications in an on-demand fashion [11]. The advantage of this approach is that users should only worry if they can get the service or not, without worrying of the underlying details.

It can be regarded as a distributed paradigm since it is possible to have the same service available for different terminals. However this is not properly what we are looking for since the computation is not distributed but locally performed on the current device, once the service or the application is loaded.

2.2.2 Elastic Personal Computing

Starting from the concept of *Flexible Computing* [12], this paradigm relies on the fact that processing data in-pace and exchanging them directly between device can overcome bandwidth limitations, with respect to offloading the entire computation to a remote server. Many solutions exist, such as the Light Weight Map Reduce framework [13] that allows the submitting of jobs to any device or group of devices, collecting results and reacting to events such as network, battery or location changes. This leverages the Elastic Computing concept by providing a mobile version of the Hadoop MapReduce framework. Other solutions like GEMCloud [14] aim at exploiting mobile devices for computationally intensive and parallel tasks while keeping an high degree of energy efficiency. It is built with a client-server architecture, where the server is in charge of discovering device through the network, registering them in a database, and choosing the device where to offload the computation. On the other hand the client application can decide to become visible based on the device status, represented by

resources usage, battery level, and running applications. This does not represent a complete solution, since it lacks scheduling and task placement policy, needed in order to maximize performance and minimizing energy consumption.

2.2.3 Volunteer Computing

The *volunteer computing* is paradigm that relies on users making their devices available for hosting external computationally intensive tasks. Many popular projects such as SETI@home [15] or Folding@home [16] are exploiting this paradigm for scientific research purposes.

An important project is the previously mentioned BOINC framework, which provides also binaries for Android, so that it is possible to exploit mobile devices but with a slightly different approach: client application starts computing only when devices is plugged into a power source. Moreover, thanks to Android application sandboxing, tasks are not able to access other apps data.

Other alternatives include an attempt to implement volunteer computing on mobile systems [17], where a device with internet capabilities can elect itself as a local task distribution point, inviting other users to join the computation via existing Device to Device (D2D) communication methods.

Furthermore, the Reliable and Efficient Participatory Computing (REPC) [18] is a generic randomized task assignment framework for the participatory computing paradigm: a centralized server in fact, hosts the execution of a Task Manager, in charge of assigning tasks to subscribed devices. The overall goal is to guarantee the completion of a given minimal number of tasks, minimizing the number of tasks assigned per device. The central server is involved also in the estimation of the run-time statistics regarding the tasks execution.

2.2.4 Enterprise Computing

The idea of mobile devices for distributing computation has found its way also in the enterprise world. Arslan *et al.* [19] proposed a distributed computing infrastructure using smartphones in enterprise context. The main idea is to help company's server by using mobile devices for the computation while recharging, to enable energy and cost saving for enterprise. This solution is quite complex, taking into account device computation capabilities and status, and it shows two main drawbacks: first of all, the client-server architecture represents a limitation in terms of scalability and flexibility. Second, there is no local resource manager on client devices, exposing local resources to the server and executing local optimization considering the workload, because all the evaluation is performed server-side.

2.2.5 High-Performance Computing

Even in the *High Performance Computing* (HPC) area some solutions were proposed, which can be considered closer to the solution proposed in this work. One of them is the DroidCluster study [20] considering mobile devices as nodes of a parallel cluster, but without modifying or replacing the Android systems. This solution leverages the Message Passing Interface (MPI) as the standard for message exchange in parallel computations in distributed systems, and his target workload is made of parallel HPC benchmark applications, so that resource management is not needed.

2.2.6 Considerations

Most of the solutions presented so far, relies on the client-server architecture, with a single server managing multiple devices, thus introducing a problem of flexibility and scalability. A different approach to explore would be to keep a client-server infrastructure, but with different server and different clients: a client may be registered on more than one server, meaning that it will compute tasks of different complex applications, and each server acts on his own, based on different requirements without the need of a central authority.

2.3 Edge/Fog Computing

As discussed in Chapter 1, Edge and Fog Computing aim at moving the computation from the data centers to the edge of the network, closer to the local devices. Even if the they are relatively recent paradigms, there are an important number of solutions proposed. One of them is based on the concept of *cloudlets* proposed by Satyanarayanan *et al.* [21]. A cloudlet is a new architectural element arising from the convergence between mobile computing/IoT and cloud computing. It represents the middle tier in a 3-tier hierarchy made of devices, cloudlets and cloud: in practice it can be viewed as a small data center, whose goal is to bring the cloud closer to the device. In particular, it has the advantage of keeping cached state from the cloud, but also to buffer data going from the device to the cloud, so that the transfer will happen reliably and securely. This also means that the cloudlet is a self-managing entity, cause it does not have any kind of hard state.

Moreover, it has a sufficient compute power to offload resource-intensive tasks from one or more mobile devices, excellent connectivity to the cloud (usually wired Internet connection), plugged to power source to avoid relying on battery. Cloudlets instances are logically close to associated mobile devices to

ensure low latency and high bandwidth between the two. This often implies that devices and cloudlets are also physically close. Finally, it is build using the same technologies we may find in the cloud, mainly virtual machines (VMs).

FemtoCloud [22], a project from the Georgia Institute of Technology, is an evolution of the cloudlet concept: instead of relying on a extra piece of infrastructure, a group of mobile devices can be grouped to function as a cluster, in a scenario such as a theatre. The key point is to use idle resources of these device to run tasks managed by a controller, which allows better scalability and avoid relying on additional infrastructures. This implementation expects client service running on the devices, which estimates the computational capability of the mobile device, and using this along with user defined settings, to report resource available for sharing. For example, user may be able to define the maximum percentage of device capabilities the service may be able to use, or specify that the computation may happen only when the battery is charging. Moreover, the client application contains a module in charge of gathering data about user preferences and behaviour used for determining his presence time, while joining FemtoCloud. In this case, the scheduling algorithm is critical for the correct performance of the system. It differs from standard algorithms because of the great volatility of the devices, as in the case where a device departs prior to completing the task assigned, which needs to be rescheduled and started from scratch on another device. This approach has some disadvantages, mostly caused by the high volatility and mobility of devices, to allow them to fulfill the offloading with other devices in the environment.

Another worth to mention solution is *Nebula* [23]. This framework offers a location/context-aware distributed edge cloud infrastructure. The Nebula cloud, following a similar pattern seen with the BOINC framework, is made of volunteer nodes, donating their computation and storage resources, and a set of services hosted on dedicated, stable nodes. These services are:

- *Nebula Central* is the front-end, providing a web interface allowing volunteers to join and tools to manage and monitor the execution of applications.
- *DataStore* is a per-application storage service supporting location-aware data processing. Each DataStore instance is linked to a volunteer node storing the actual data, while the DataStore Master maintain metadata and makes data placement decisions.
- *ComputePool* provides per application computation resources through a set of volunteer nodes. Nodes in a ComputePool are scheduled by a ComputePool Master that coordinates the execution. Moreover, ComputeNodes

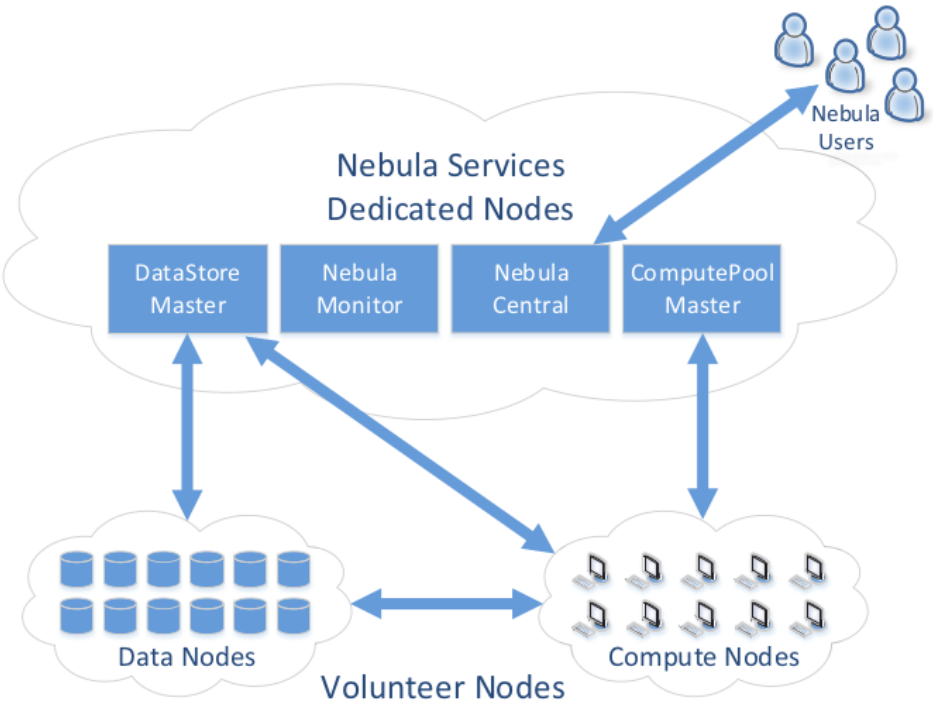


Figure 2.2: *Structure of Nebula Cloud services*

have access to Data Nodes for retrieving data needed, and they are assigned tasks based on application requirements and data location.

- *Nebula Monitor* is the service in charge of monitoring volunteer nodes and network characteristics. The parameters monitored are mainly computations speeds, memory and storage capacity, network bandwidth and health information about nodes.

A typical nebula application is made of a number of *jobs*. A job contains code to carry out a specific computation, it has a specific input dataset made of multiple data objects, and produces an output. Jobs may have dependencies between each other, meaning that a job's output may be needed for another job as input data. Moreover each job consists of multiple tasks, which may run in parallel and work on a partition of the input data.

CHAPTER 3

MANGO Project Background

In this chapter we will introduce the MANGO project, which provides a framework for High Performance Computing applications on heterogeneous architectures and represents the starting point of the work proposed in this thesis. In Section 3.1 an in-depth overview of the MANGO framework is presented, including its main components and functionality. In addition, Section 3.2 describes the run-time resource manager used by the framework, called *BarbequeRTRM*. Finally Section 3.3 explains the programming model adopted by the framework along with an example from an use-case.

3.1 The MANGO Approach

The main challenge that has to be faced in HPC is the performance/power efficiency. The gap between the application's requirements and the underlying architecture needs to be taken into account in order to achieve the maximum exploitation of computing technologies and power efficiency. The MANGO project goal is to investigate architectural implications of the emerging requirements in HPC applications, aiming at the definition of new generation high-performance, power-efficient, deeply heterogeneous architecture with isolation and QoS compliance in mind. Since QoS and time predictability are often ignored in HPC,

the traditional optimization space is extended from power/performance to *power, performance* and *predictability*.

One of the challenges that MANGO tries to solve is the optimization of resource allocation, which is not trivial since applications may be composed of multiple tasks, a task may be executed on different computing units of an heterogeneous architecture. Moreover, the framework needs to comply with applications' requirements while also addressing system-wide requirements. To provide applications with a resource-agnostic view of the available resources, MANGO provides a *hierarchical resource management* strategy, made of a global resource manager (GRM) in charge of workload balancing and thermal control of the system, and a local resource manager (LRM), in charge of the allocation of node resources, allowing multiple applications to share resources located on a single node.

This has led the project to develop a software stack including a novel programming model, for heterogeneous multi-processor systems, integrated with a run-time resource management solution [24]. The outcome has inspired this thesis work, with the idea of continue the development of this programming model, in order to experiment its adoption in distributed systems, including also embedded and mobile devices, like the ones emerging in this Fog/Edge computing era.

3.1.1 Local Resource Manager

The Local Resource Manager is in charge of a single node: a node is a subpart of the system containing a multiprocessor CPU-based system, the General Purpose node, to which are connected boards containing a set of heterogeneous processing units and memory nodes interconnected. From the point of view of the resource management, applications are made up of different tasks compiled for one or more architectures, and buffer allocation requests, with the possibility of inter-task data transfer. Starting from this assumption, the *resource assignment* process is made up of three sub-tasks: a) map for each task the processing units to assign, b) allocate buffers onto the memory nodes, taking into account access latency, memory bandwidth used and contention on memory controllers, c) reserve a minimum amount of bandwidth to support efficient data transfer. These decisions are enforced by user-defined policies, and leverage the *isolation* and *reservation* mechanisms of the underlying hardware platform support, increasing *predictability* of the execution.

Another important aspect of an HPC infrastructure is the power management: indeed the resource manager can a) switch off processing units, b) change op-

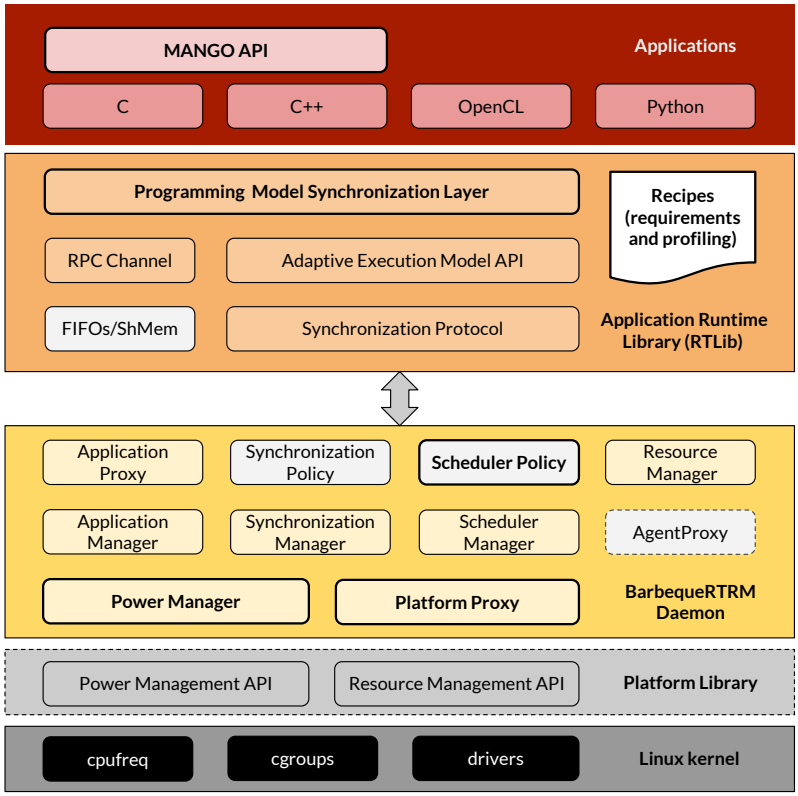


Figure 3.1: *BarbequeRTRM integrated into the MANGO framework*

erating point of the processing unit (e.g. DVFS [1]), c) select the processing unit with the lowest power consumption profile for the same architecture.

3.1.2 Global Resource Manager

The entry-point for all the MANGO applications is the Global Resource Manager, which decides at run-time the node where to execute new applications, and interacts with each Local Resource Manager instances. The main components of a GRM are: a) workload scheduler, in charge of receiving workloads and assigning them to nodes, b) power/thermal agent which applies defined policies in order to improve efficiency of the system, c) a thermal simulator used to predict the thermal behaviour of the system.

3.2 The BarbequeRTRM

The MANGO framework leverages Barbeque Run-Time Resource Manager, a project developed at *Politecnico di Milano* by the BarbequeRTRM OpenSource Project (BOSP). It is a portable and extensible framework for adaptive run-time

[1] Dynamic Voltage and Frequency Scaling

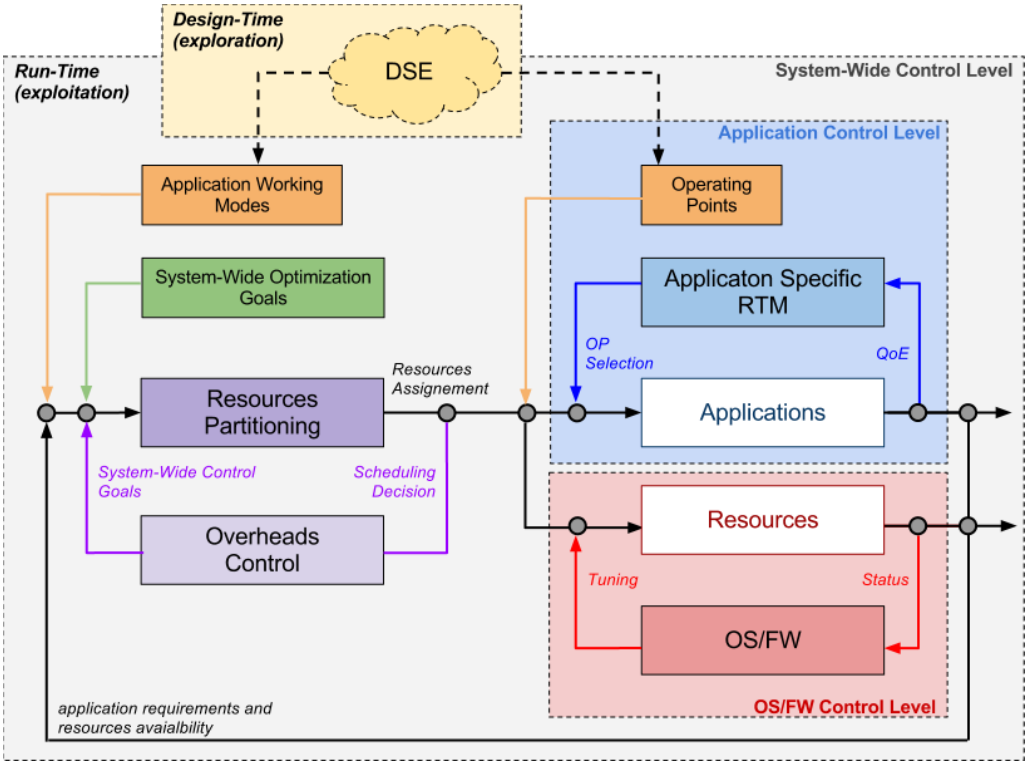


Figure 3.2: *Distributed control scheme*

resource management of many-core architectures, offering optimal resource partitioning and adaptive run-time scheduling of the different re-configurable architectures. It has been designed to be highly modular, providing support for easy integration and extension, and support for both homogeneous and heterogeneous architecture is achieved thanks to the *Platform Proxy* module which handles the low-level communication.

To effectively use computational resources, the application must be able to run according to different configurations, which means running with different resource usage levels. At design time, a suitable Design Space Exploration (DSE) activity identifies all the possible configurations by profiling the specific application. Each configuration is defined by a set of parameters, some of which impacts only on the applications behaviour, while others have a direct impact on the amount of required computational resources. This distinction allows to define two different granularity levels: *Operating Point* (OP) is a set of application specific parameters, corresponding to an expected QoS for the end user. While *Application Working Mode* (AWM) is a collection of resource requirements corresponding to an expected QoS for the application. So while OPs are tied to application specific properties, AWMs described system-wide properties which affect system resources and other entities competing in their usage.

For an optimal usage of these parameters, BarbequeRTRM offers a distributed and hierarchical control scheme, where each controller is in charge of a specific subsystem, allowing to spread the complexity and to scale better with system complexity and number of subsystem.

As shown in Figure 3.2, the hierarchical approach allows a fine-grained control at different abstraction levels. There are three main classes of subsystems, each one related to a different control level: the *application control level*, which is defined for every run-time tunable application, it's in charge of evaluating run-time behaviour of an application and tune its configuration based on the analysis. At this level, the *Application Specific Run-Time Manager* (AS-RTM) reads the amount of resources assigned to the applications (AWM) and its run-time QoE [2], using these data as a feed-forward signal by defining a suitable policy to select an OP.

At the same level lies the *OS/FW control level* which is a platform-specific modules where DVFS and thermal control take place. At the *system-wide control level* we have the *System-Wide Run-Time Resource Manager* targeting a set of optimization goals, which will be pursued thanks to a proper assignment of the available resources.

3.2.1 Adaptive Execution Model API

The *Application Run-Time Library* (RTlib) provides a rich set of features and APIs to interact with the framework and to support run-time management activities. This library provides the *Adaptive Execution Model*, a callback-based API similar to the Android programming model: developer has to implement a C++ class derived from the base class defined in the RTLib. At run-time, the instance of this class will spawn a control thread which will then call the methods as shown in Picture 3.3. These method have the following semantics:

- **onSetup**: initialization code
- **onConfigure**: reacts on resource allocation changes, so that the application can re-configure itself
- **onRun**: part of the application performing computation
- **onMonitor**: called after onRun to check current performance and notify the resource manager
- **onRelease**: de-allocation and release

[2] Quality Of Experience

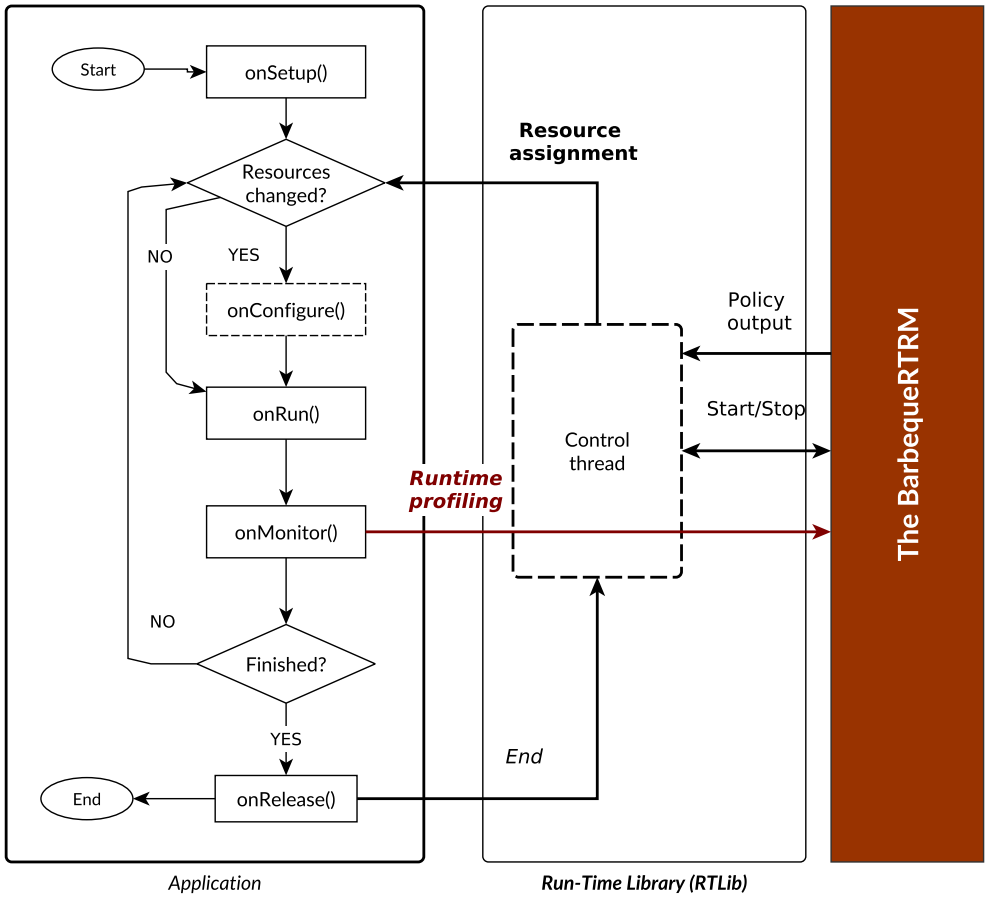


Figure 3.3: *Abstract Execution Model (AEM)*

Workflow starts with the *onSetup* initialization, and then waiting for resources to be ready to be assigned. After the assignment, the *onConfigure* is called so that applications may adapt to the availability of resources, and once configured the control thread starts a loop where *onRun* and *onMonitor* are called. *onConfigure* will be called again only if there's some change in the allocation of resources, event that is notified to the control thread by the daemon. Finally, after the application terminates its execution, the *onRelease* is called before exiting.

3.3 MANGO Programming Model

3.3.1 Application structure

As explained in section 3.1, an instance of the MANGO platform typically is made of a general-purpose node, based on specific processors, connected through a PCI Express link to a cluster of FPGA boards, containing a set of heterogeneous units. MANGO will run multi-tasking applications, which may spawn different

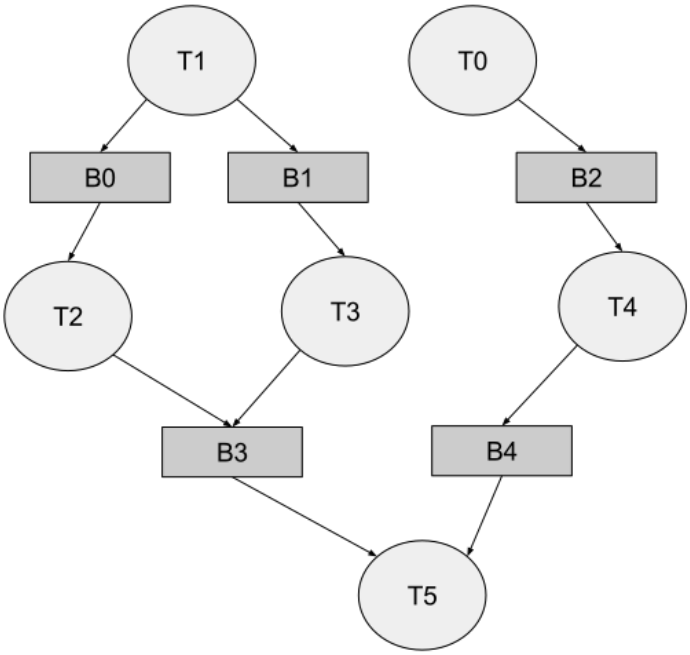


Figure 3.4: *An example of a Task Graph for a complex application. A buffer can be both output for some kernels and input for others*

threads and run on heterogeneous processing units, characterized by different architecture and specifics. These tasks may exchange data thanks to a shared memory context, where data buffers may be read or written by different units.

A MANGO application is usually made up of different kernels built for different architectures exchanging data though some buffer that will be allocated on a physical memory shared between all the nodes. To keep a representation of the structure of the application that the resource manager can understand in order to find the optimal resource allocation, applications build a Task Graph, a graph representing inter-dependencies and the hierarchy of the various elements of the applications, like shown in picture 3.4.

3.3.2 API

Since application may be developed by domain experts with limited knowledge in computing, the programming model adopted by MANGO needs to be simple to understand even to non-specialist, and hide all the complexities under an easy to use interface. The programming model run-time support, *MANGO application library*, is made of two layers one at the host-side and the other at the device-side. MANGO API will be presented by showing the *GIF_FIFO* sample application as a practical overview of the usage of the API.

The *host-side low-level runtime* (HLR) contains a set of functions used to access the functionality of the accelerators from the main application code running on a CPU-based general purpose node (GN). The initialization routine is provided by the *BBQContext* which instantiates the Application Controller and initializes the communication with the HN library. For each application task, it is possible to initialize a *KernelFunction* object that allows to load an executable for a target architecture. In this case application is made of only one task, as shown in the following listing.

Listing 3.1: *"Initialization of the BBQContext and KernelFunction"*

```
1 context = new mango::BBQContext("gif_animation","gif_animation");
2 auto kf_scale = new mango::KernelFunction();
3 kf_scale->load(kernel_binary_path_cpu,
4     mango::UnitType::GN, mango::FileType::BINARY) ;
5 kf_scale->load( kernel_binary_path_peak ,
6     mango::UnitType::PEAK, mango::FileType::BINARY) ;
```

To be able to create a Task Graph, buffers and kernel should be registered through the context. To register the kernel, required parameters are an unique id, the kernel object initialized before, and the list of input and output buffers reference by their ids. As for buffers, they need to be registered providing an id, list of kernel object reading and writing to it referenced by id and the type of buffer. With these object, it is possible to create a Task Graph and pass it to the the resource allocation command. When the function returns, the Task Graph is filled with information about the allocation, and the execution can start.

Listing 3.2: *"Resource allocation"*

```
1 auto kscale = context->register_kernel(KSCALE, kf_scale, {B1}, {B2
      });
2 auto b1 = context->register_buffer(B1,
3     SX*SY*3*sizeof(Byte), {}, {KSCALE}, mango::BufferType::FIFO );
4 auto b2 = context->register_buffer(B2,
5     SX*2*SY*2*3*sizeof(Byte), {KSCALE}, {}, mango::BufferType::FIFO
          );
6 tg = new mango::TaskGraph({ kscale }, { b1, b2 });
7 context->resource_allocation(*tg);
```

Before proceeding, the list of arguments needs to be defined, which allow us to define extra parameters for simple scalar values or events, followed by the initialization of buffer data. The function *start_kernel* is used to trigger the execution of the task, and it return a kernel completion event that can be used to wait for the end of the execution.

Listing 3.3: *"Kernel execution and synchronization"*

```
auto argB1 = new mango::BufferArg( b1 );
auto argB2 = new mango::BufferArg( b2 );
auto argSX = new mango::ScalarArg<int>( SX );
auto argSY = new mango::ScalarArg<int>( SY );
auto argE1 = new mango::EventArg( b1->get_event() );
auto argE2 = new mango::EventArg( b2->get_event() );
argsKSCALE = new mango::KernelArguments(
    { argB2, argB1, argSX, argSY, argE1, argE2 },
    kscale
);
b1->write(in, 4*SX*SY*3*sizeof(Byte));
b2->read(out, 4*SX*2*SY*2*3*sizeof(Byte));
auto e3=mango_rt->start_kernel(kscale, *argsKSCALE);
e3->wait();
```

On the device-side, the *device-side low-level run-time* (DLR) provides synchronization mechanisms at the accelerator level, and allows to map physical addresses of buffers allocated in the shared memory to virtual addresses. In particular, the *mango_wait* functions allows the kernel to wait for the input buffer to be ready for reading and the output buffer to be ready for writing. After the computation takes place, it is possible to notify the host that buffers are ready and accessible for reading and writing, through the *mango_write_synchronization* function

Listing 3.4: *"Device-side execution"*

```
void kernel_function(uint8_t *out, uint8_t *in, int X, int Y,
    mango_event_t e1, mango_event_t e2){
    for(int i=0; i<4; i++) {
        mango_wait(&e1, READ);
        mango_wait(&e2, WRITE);
        printf("KERNEL: mango_wait\n");
        scale_frame(out, in, X, Y);
        mango_write_synchronization(&e1, WRITE);
        mango_write_synchronization(&e2, READ);
        printf("KERNEL: mango_write_synchronization\n");
    }
}
void scale_frame(uint8_t *out, uint8_t *in, int X, int Y){
        int X2=X*2;
        int Y2=Y*2;
        for(int x=0; x<X2; x++)
            for(int y=0; y<Y2; y++)
                for(int c=0; c<3; c++){
                    out[y*X2*3+x*3+c]=in[y/2*X*3+x/2*3+c];
                }
}
```

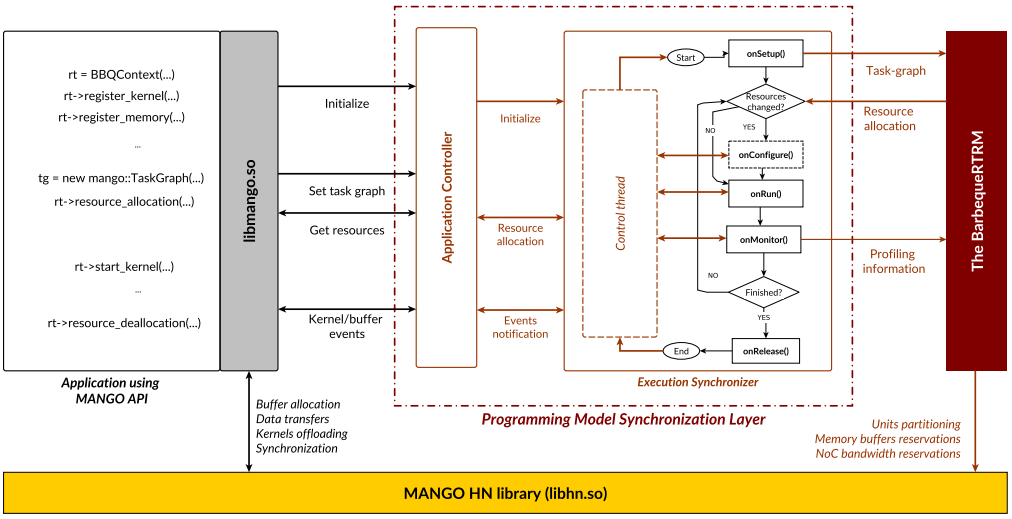


Figure 3.5: *Programming Model Synchronization Layer as a bridge between MANGO programming model and Barbeque*

3.3.3 Integration with BarbequeRTRM

To integrate the BarbequeRTRM daemon into the MANGO programming model, the *Programming Model Synchronization Layer* was built. This layer provides an abstraction of the resource assignment process, allows the synchronization between the task execution and the local resource management functions and profile the task execution. As shown in figure 3.5, the PMSL interface called in a sample MANGO application resides in the Application Controller, which initializes an execution context of the AEM. This controller is initialized by the *BBQContext* class, and the previously described API leverages directly this layer to provide a bridge between the two frameworks.

To share the Task Graph between MANGO and BarbequeRTRM instances, the *Task Graph Library* comes in help, providing functions for the management of those objects. The execution synchronizer will wait on the *onSetup* until all these object are instantiated and the task graph is provided from the MANGO application through the resource allocation call, so that the PMSL can forward it to the daemon. Here BarbequeRTRM executes the allocation policy and then return the Task Graph object to the PMSL, filled with all information needed for allocation. The resource allocation call in the sample application returns, the execution synchronizer enters the *onConfigure* phase waiting for a kernel to execute. Now the application can initialize buffers and load the kernel, since it has all the mapping information of the resource manager, and it is able to start the kernel with the *start_kernel* function. The *onConfigure* starts a thread for each kernel executed to monitor its execution time and throughput before entering the

onRun phase. The *onMonitor* will collect data retrieved from the control thread spawned in the *onConfigure* and send it to the resource manager, so that it can profile the application execution. Finally the *onRelease* function is in charge of releasing all the assigned resources, when all kernels terminate.

CHAPTER 4

The BeeR Framework

This chapter introduces the design and implementation of the BeeR framework, aiming at seamlessly managing the execution of multi-tasking (multi-kernel) applications, in a distributed fashion. This has the goal of extending the capabilities of the software support, previously developed for the heterogeneous computing platforms explored in the MANGO project. In Section 4.1, the first phase of the process is described, which corresponds to the definition of the requirements that the framework must satisfy and its general structure. In Section 4.2 we describe the server-side part of the framework, as well as the client-side, consisting of the extension of the *libmango* library. Section 4.3 provides details on the current implementation, with some code samples of the key components, and an overview of the tools and libraries used.

4.1 Requirements

The purpose of BeeR is to extend the capabilities of the MANGO framework to allow distributed embedded devices, connected together through the network, to participate in the execution of an application. With this approach, an application can choose to execute its tasks leveraging CPU-based general purpose nodes scattered across the network, or reserve only a subset of the tasks for remote computation and launching the remaining ones on heterogeneous nodes

connected through the PCI express link. This means that client applications are able to communicate with remote entities, where an instance of the BeeR daemon is in execution. This has to be done transparently by the MANGO library, without modifying its existing interface.

It is clear that in order to achieve this goal, a client-server architecture should be adopted. Client needs a way to discern if any of the tasks is running on a *local* node, or is going to be run in a *remote* one. In this way during the resource allocation stage it is possible to differentiate between local and remote resources, and act consequently. Furthermore, client should be able to replicate most of the data structure on the remote device, by sending and receiving data wrapped into messages, which of course require the definition of a protocol. On the other hand, server must be able to replicate the work-flow of a typical mango application, such as allocate buffers and kernels, execute the kernel, wait for events, read and write buffers. Client application should take into account the architecture of the server and correctly provide a pre-compiled binary executable. Since we could have multiple MANGO application running at the same time on different hosts, and different devices, BeeR follows a Many-To-Many structure, meaning that each instance of the server could receive tasks from multiple client applications.

Given the Programming Model described in Section 3.3, messages exchanged between client and servers must follow this interface. In particular, we need a mechanism for the following set of operations:

- Registering buffer and kernels using the id, size and other parameters provided to the MANGO library (*libmango*) by the application code
- Initialize a kernel object by providing the executable binaries compiled for the target architectures
- Read and write buffers remotely allocated
- Start the remote execution of the kernels
- Managing the occurrence of remote events (e.g., kernel termination)
- Retrieve statistics about the kernel execution

One of the techniques that can be used to transfer back and forth data structures is object serialization, which consists in the conversion of an object instance into a sequence of bytes. This can the be saved into a file or transfer through the network. In this way if both client and server have the same class defined, instances of that class can be wrapped into a simple message and transferred through a TCP connection.

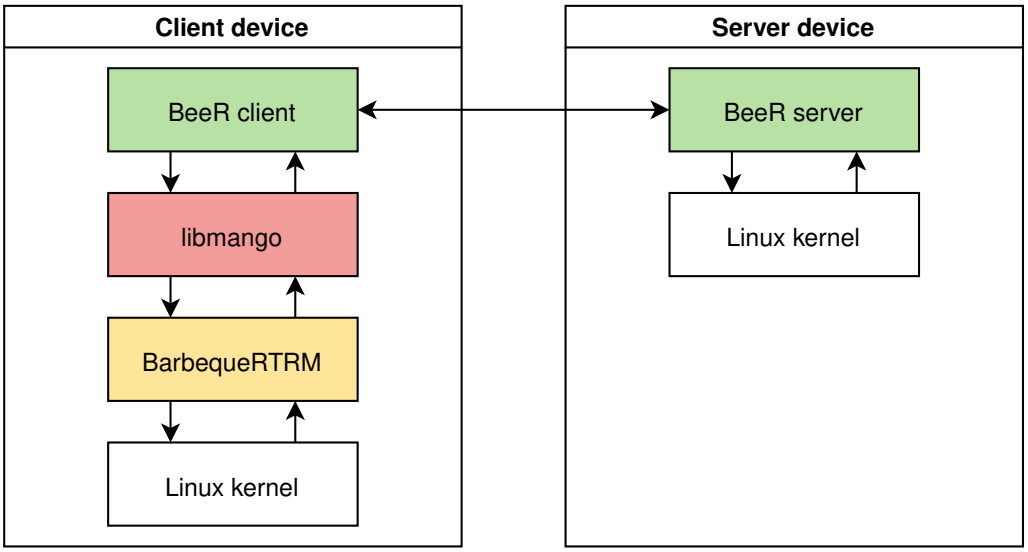


Figure 4.1: *Clients use the extended libmango, while the server is a standalone tool*

4.2 Design

The BeeR client is a typical MANGO application, leveraging the set of libraries provided by the framework, as shown in Figure 4.1. To provide the capability of offloading tasks and buffers to a remote BeeR server instance, the underlying *libmango* was extended.

The BeeR server is the component of the framework in charge of managing the execution of a kernel on a device, as remotely required from a "host" machine. The server is executed as a daemon and it does not need to link the MANGO library. This approach saves us from having to build the MANGO framework for each device architecture, a task which is time-consuming and may not be easy to port to different architectures, because of the long list of dependencies. Moreover, having a simple codebase written with portability in mind, allows us to easily build this implementation for different architecture without much effort.

4.2.1 Server: Requests management

As soon as the server receives a new connection, it instantiates a new *Task* object. This object will manage the incoming requests in a separate thread, so that the main server loop can immediately start listening for a new connection to handle. This concurrent approach allows the server to handle different tasks from different clients simultaneously using a *Thread Pool*. By default the pool configuration will spawn a maximum number of threads equal to the machine concurrency level (number of cores of the processor), otherwise it's possible to

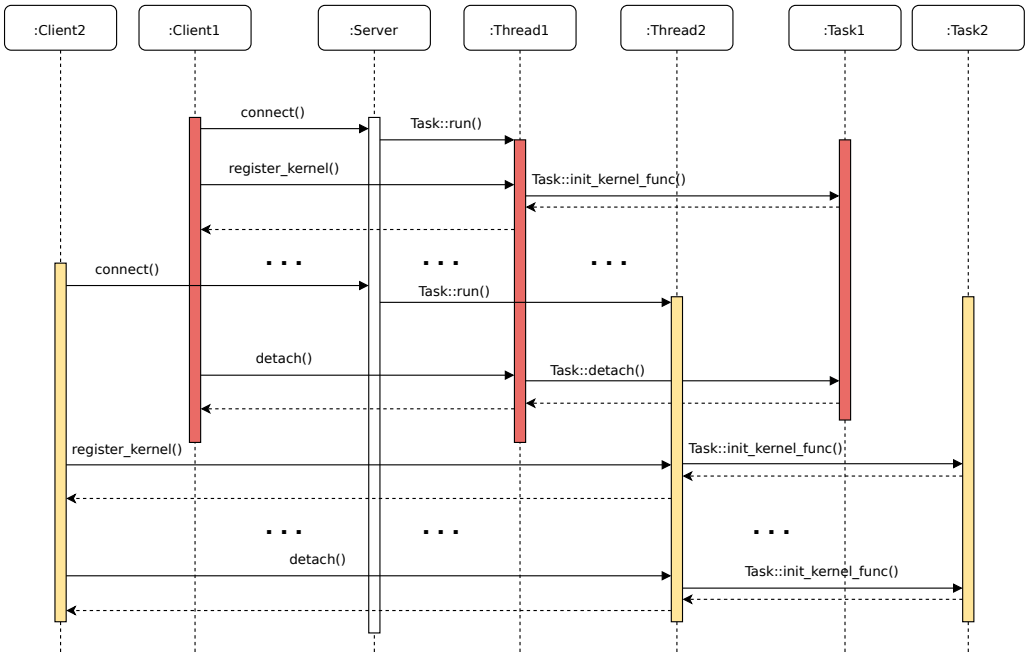


Figure 4.2: *UML Sequence diagram showing a typical flow in the execution of an application with kernels remotely spawned.*

specify a value for that number. A sample case is shown in figure 4.2: server receives a connect request from client1 and spawns a new thread *thread1* which initializes a new Task *task1* by calling the function *run()*. From now on, *thread1* will handle the TCP connection from *client1* and the server can return to listening for new connections. *client1* sends a *register_kernel()* command, and the corresponding thread will call the function *init_kernel_func()* to parse kernel data from the payload. During this time, a new client *client2* connects to the server, spawning a new thread *thread2* which initializes a new Task *task2*. These two tasks will run concurrently without any possibility of conflict: any data associated with a thread will have its own unique identifiers, so that each kernel can execute independently. After a *detach* call from *client1*, *thread1* will terminate and *client2* will continue with the same work-flow.

4.2.2 Server: Remote kernel execution management

To manage kernel execution, the Subprocess class was introduced taking care of the binary executable sent from the client. This class has two methods: *run* will execute the kernel with a given list of arguments sent from the client. Binary will be run as a separate process using the fork()+exec() syscalls sequence, where the fork call creates a new process by cloning the current one, and the exec replaces the content of the address space by executing the binary file provided as

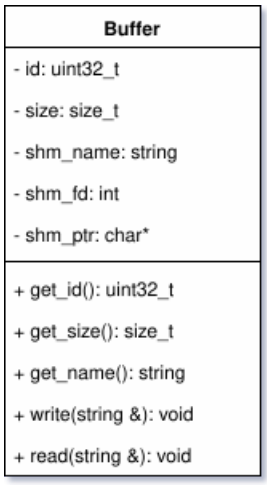


Figure 4.3: *The buffer class*

an argument.

This class will store the PID (process identification number) of the child process, allowing the server to wait for its completion. This is done by the second function *wait* which returns as soon as the child process exits. This works by calling the *waitpid* function defined by the C library, which waits for the child process identified by the supplied PID to terminate.

Each thread spawned by a new connection will contain only one Subprocess object: in this way different kernels are handled by different connections, so that they can be executed simultaneously by the server.

4.2.3 Server: Remote buffer management

As we seen in Chapter 3, the MANGO programming model expects data exchange between different kernels running on different nodes through buffers allocated on a shared memory. Following this structure, kernels launched by the BeeR daemon will be able to access buffers thanks to the POSIX shared memory API. The Buffer class manages manages a shared memory reference for a specific buffer identified by an *id* and with a specific *size* measured in bytes. To avoid conflicts with other tasks running on the same server instance that may allocate buffers with the same *id*, the shared memory object is initialized with a unique name. Member functions *write()* and *read()* are used to respectively write data from the buffer into the shared memory and read data from the shared memory into the buffer. To keep data buffers, BeeR uses the *string* C++ class, since it is basically a wrapper around an array of char, suitable for storing a sequence

Figure 4.4: *Structure of a request message*

of bytes, but also because it can be easily serialized and transferred.

4.2.4 Message types

Client and server are connected via TCP, while the exchange of messages is managed through a simple protocol: client sends an message to the server and receives a response containing the result of the operation, and optional messages regarding the execution of the command. The message structure is shown in Figure 4.4: first 8 bits contain the "opcode" identifying the type of request, followed by the "payload", containing the data characterizing the request. In this first version of the framework, the set of possible opcodes is the following:

- *COMMAND_KERNEL_FUNCTION*: initialize kernel object on the server, payload stores a serialized descriptor class containing binary executable and parameters such as file type and unit type.
- *COMMAND_BUFFER*: initialize a buffer, payload stores a serialized descriptor class containing id, size, list of kernels writing and reading to it referenced by their ids.
- *COMMAND_START_KERNEL*: triggers the execution of the kernel.
- *COMMAND_KERNEL_WAIT*: wait for kernel completion. This is a synchronous request, which returns as soon as the kernel terminates.
- *COMMAND_BUFFER_READ*: return buffer referenced by an id, payload stores buffer information.
- *COMMAND_BUFFER_WRITE*: write buffer referenced by an id, payload stores information and data that needs to be written.
- *COMMAND_DETACH*: closes the connection.
- *COMMAND_SCALAR_INT*: register an integer as scalar argument.
- *COMMAND_SCALAR_FLOAT*: register a float as scalar argument.
- *COMMAND_SCALAR_CHAR*: registers a character as scalar argument.

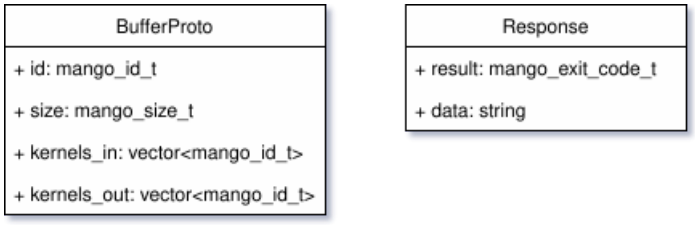


Figure 4.5: *Descriptors used for buffer and server's response*

The payload, as described in the previous list, can be a serialized object, a sequence of bytes representing some values, or it can be empty for tasks that does not need data, such as the kernel execution trigger. By serializing information, we are able to build a simple protocol, by moving most of the information into serialized descriptors, and leveraging external libraries for the implementation.

4.2.5 Remote client integration

To represent a remote instance of the BeeR daemon, libmango was extended with a new *Device* class. This class will provide functions for sending requests and receive data, and it is initialized by providing the host and port of the remote endpoint.

As anticipated in Section 4.1, to be able to send and receive serialized objects, client and server must share the same descriptors. Figure 4.5 shows an example: the *BufferProto* descriptor storing the id, size and the kernel ID for which it acts like an input or output buffer. *Response* is a simple descriptor containing the return code of a remote command and a possible result stored in the data string. For trivial operations, *data* will contain a simple message, while it may store buffers for read/write operations. Other requests such as *send_scalar* or *write_buffer* don't have descriptors, because data is serialized and sent right away, without the need of additional information.

4.3 Implementation

Most of the described functionality is achieved with the help of the BOOST libraries [25], a set of portable peer-reviewed C++ source libraries providing support for a wide of tasks and structures. It is composed of different libraries, which can be compiled individually if developers need only a subset of the functions provided. BOOST libraries provide (1) an easy to use API for object serialization/de-serialization, (2) APIs for implementing TCP servers and clients, both in a synchronous and asynchronous way, (3) facilities to wrap TCP connec-

tions into input/output streams, so that writing data to or from a server becomes as easy as writing to the standard output or reading from the standard input. The BOOST version used is the same as the one compiled in BarbequeRTRM, to avoid compatibility problems between clients and servers, but also to ease the integration of the daemon into the MANGO framework, when this will occur.

BeeR server implementation is quite small, averaging 1k SLOC. It consists of a main loop which receives a new connection and starts a new thread handling it, as shown in the following listing:

Listing 4.1: *"Daemon main loop and thread dispatching"*

```
1  void Server::start()
2  {
3      ThreadPool pool;
4      m_server.listen();
5      std::cout << "server listening on port " << m_port << std::endl
           ;
6      while (true) {
7          std::shared_ptr<tcp::iostream> stream = std::make_shared<
               tcp::iostream>();
8          Task t(stream, m_mutex);
9
10         m_server.accept(*(stream->rdbuf()));
11         // std::bind will copy Task variable
12         std::function<void()> f = std::bind(&Task::run, t);
13         pool.push_task(f);
14     }
15 }
```

First, thread pool is initialized and *m_server* object starts listening for new connections. In each iteration of the following loop, a new *Task* object is created, passing to its constructor a pointer to a new *iostream* object and a pointer to the mutex variable: the stream object will handle the connection after starting the Task, while the mutex will be used mainly for thread synchronization when logging to the standard output. In line 10 the *iostream* underlying buffer is passed to the *accept* method of the *m_server* variable, which is an istance of the BOOTS *tcp::acceptor*: by doing so, we instruct the acceptor to use iostream's buffer as the socket for the connection. Finally, using *std::bind* template it is possible to create a call wrapper for function *Task::run*, by binding together the method with the instance of the *Task* object, and add it to the thread pool queue.

Once the thread starts, it executes the wrapped function which starts reading the TCP stream for messages. The following listing shows how messages are read: in lines 3-5 we check if there's any trailing newline character in the stream and discard it. This is needed since sometimes after de-serializing an object, a

new-line character could be left behind by the parser causing an error with the next message. The real parsing starts by reading 1 byte from the stream, storing its value in a *RemoteCommand* variable.

Listing 4.2: *"OpCode parsing"*

```
1  void Task::run() {
2      std::atomic<bool> quit(false);
3      while(!quit) {
4          if (stream->peek() == '\n') {
5              stream->ignore(1);
6          }
7          try {
8              char type_frame;
9              stream->read(&type_frame, 1);
10             RemoteCommand cmd = (RemoteCommand) type_frame;
11             switch (cmd) {
12                 //
13                 // ...
14                 //
15             }
16         }
17     }
18 }
```

The value of the *cmd* is checked in a switch statement, where based on its value the correct method of the class is executed. Typically, the applications will follow the following workflow:

1. register a kernel
2. register some buffers or scalar values
3. send arguments for the kernels
4. write input buffers
5. start the kernel(s)
6. wait for kernel completion
7. read output buffers.

Concerning point (1), the function *init_kernel_func* will basically parse a serialized string from the TCP stream. This string stores the buffer containing the binary file to be executed which will be saved on server *tmp* directory with a unique name. A new *Subprocess* object is initialized with the path to the temporary file: this object will be used to run the executable and wait for its completion in the last phases.

Listing 4.3: *"Function for parsing kernel executable"*

```
1  void Task::init_kernel_func() {
2      std::string temp;
3      {
4          boost::archive::text_iarchive ia(*stream);
5          ia >> temp;
6      }
7      {
8          std::stringstream ss;
9          ss << "received kernel, size=" << temp.size();
10         log(ss.str());
11     }
12     std::ofstream ofs(binary_name, std::ios::binary|std::ios::out);
13     ofs.write(temp.c_str(), temp.size());
14     ofs.close();
15     chmod(binary_name.c_str(), S_IRWXU);
16     proc = std::make_shared<Subprocess>(binary_name.c_str());
17     {
18         std::stringstream ss;
19         ss << "initialized subprocess for " << binary_name;
20         log(ss.str());
21     }
22     Response r(ExitCode::SUCCESS, MSG_OK);
23     {
24         boost::archive::text_oarchive oa(*stream);
25         oa << r;
26     }
27 }
```

For point (2), registering a scalar is straightforward: function will parse it from the stream, and add it directly to the *Args* object. This object will keep track of all the arguments registered by the client, and generate a proper string when executing the binary. Function for registering buffers will at first create a new *Buffer* object, add it to the map containing all buffers referenced by their id, and add them to the *Args* instance. When generating the string for the executable, *Args* objects will write the shared memory name for buffers rather than their id, so that kernels have a direct reference to them.

Reading and writing a buffer is done by calling the appropriate method of the *Buffer* class. *read* and *write* copy the content of the shared memory into the string buffer, and *write* will copy the string buffer into the shared memory using the *memcpy* function.

Listing 4.4: *"Read and write functions of the Buffer class"*

```
1  void Buffer::write(const std::string & data) {
2      assert(data.size() != 0 && "empty buffer");
3      if (shm_fd == -1) {
```

```
4           throw std::runtime_error("invalid shm fd");
5       }
6       if (shm_ptr == MAP_FAILED) {
7           throw std::runtime_error("mmap failed");
8       }
9       ftruncate(shm_fd, size);
10      std::memcpy(shm_ptr, data.c_str(), size);
11  }
12
13  void Buffer::read(std::string & b) {
14      if (shm_fd == -1) {
15          throw std::runtime_error("invalid shm fd");
16      }
17      if (shm_ptr == MAP_FAILED) {
18          throw std::runtime_error("mmap failed");
19      }
20      b.insert(0, shm_ptr, size);
21  }
```

After setting up all the arguments and initializing input buffers, server will receive the *start_kernel* command which will generate the final argument string and run the executable, followed by the *wait_kernel* already discussed in Section 4.2.

4.3.1 Libmango remote extension

To keep things as close as the original API, a new *Device* object needs to be instantiated by the application and the registered within the *BBQContext*, like buffers and kernels objects. Then, in order to "mark" a kernel or a buffer as being remote, *register_kernel* and *register_buffer* calls will optionally receive an extra argument corresponding to the Device object initialized before. If this argument is not provided, it defaults to *nullptr* and the specific buffer/kernel is considered to be local. On the other hand if we specify a Device, the object is considered to be remote, and a reference to the provided Device is saved into the instance. In the *KernelFunction::load()* function, the code checks for a reference to the remote device and if it finds it, it will send the executable using the appropriate function.

Listing 4.5: *"Remote kernel management in load function"*

```
1  mango_exit_code_t KernelFunction::load(const std::string &
       kernel_file,
2           UnitType unit,
3           mango_file_type_t type,
4           const std::shared_ptr<Device> &dev) noexcept {
5       ...
```

```
6       if (dev != nullptr) {
7           res = dev->send_kernel_function( kernel_file );
8           mango_log->Info("syncing binary with remote server");
9       }
10      ...
```

This information is also used during the resource allocation phase:

Listing 4.6: *"Remote buffer management in resource allocation"*

```
1  mango_exit_code_t BBQContext::resource_allocation(TaskGraph &tg)
       noexcept {
2
3       for (auto & b : buffers) {
4           if (b.second->isRemote()) {
5               b.second->sync_with_remote();
6               mango_log->Info("buffer %d synchronized with server", b
                    .first);
7           }
8       }
9
10      ...
```

The *resource_allocation* function will iterate over the buffers to check their status. If a buffer is remote then *sync_with_server()* function will be called, which manages to send the request to the server. Notice that the code seen up to now basically is similar to the original, with the only addition of an if statement to execute remote instruction. The *start_kernel* function has a different behaviour:

Listing 4.7: *"Remote kernel offload"*

```
1  std::shared_ptr<Event> BBQContext::start_kernel(std::shared_ptr<
       Kernel> kernel,
2               KernelArguments &args, std::shared_ptr<Event> _e)
                    noexcept {
3
4       if (kernel->isRemote()) {
5           args.sync_with_server(kernel->get_remote_server());
6           kernel->get_remote_server()->start_remote_kernel();
7           return kernel->get_termination_event();
8       } else {
9           this->bbque_app_ctrl.NotifyTaskStart(kernel->get_id());
10          auto e = Context::start_kernel(kernel, args);
11          e->set_callback(
12              &bbque::ApplicationController::NotifyTaskStop,
13              this->bbque_app_ctrl,
14              kernel->get_id());
15
16          bbque_tg->Print();
17          print_debug(__FUNCTION__,__LINE__);
```

```
18            return e;
19        }
20    }
```

In this case we need to execute a different set of instructions for each case. If the kernel is remote, function will send arguments, start the remote kernel, and return the remote kernel completion event. While if it's local, we need to execute the normal set of instructions, so that non-remote application can keep working.

CHAPTER 5

Benchmark Porting

Once implemented the framework, the next step was to evaluate its performance and the overall behaviour of distributed tasks. To carry out this evaluation, an OpenCL benchmark from the Rodinia Benchmark Suite was taken as a reference, called *PathFinder*. Section 5.1 describes this benchmark in details, explaining the algorithm and how it is structured in its original implementaton. Then Section 5.2 describes how it was implemented from scratch with libmango, and in Section 5.3 we introduce the parallel version, required in order to make the most out of the BeeR framework.

5.1 Rodinia benchmarks: the PathFinder sample

The University of Virginia Rodinia Benchmark Suite is a collection of parallel programs which targets heterogeneous computing platforms with both multicore CPUs and GPUs. It contains various implementations of each program using CUDA, OpenCL and OpenMP. One of the programs contained in this suite is *PathFinder*, an alogrithm that finds a path on a two-dimensional matrix from the bottom to the top with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally. It iterates sequentially row by row, picking for each node a neighboring node on the previous row which has the smallest accumulated weight, and adding its own weight to the sum. The result

of the algorithm is a vector, containing a number of elements equal to matrix row length, containing the accumulated weight of the shortest path starting from the bottom of the matrix at that index. The matrix is generated randomly but starting from a fixed seed, so that values are always the same, to help the debugging process. As an example, starting from the following matrix

$$\begin{bmatrix} 5 & 4 & 5 & 7 & 0 & 3 \\ 0 & 8 & 2 & 2 & 6 & 3 \\ 8 & 9 & 7 & 5 & 9 & 0 \end{bmatrix} \tag{5.1}$$

the resulting array is equal to

$$\begin{bmatrix} 12 & 13 & 9 & 7 & 11 & 3 \end{bmatrix} \tag{5.2}$$

since starting from first element of the bottom row, the shortest path to the top is 8 -> 0 -> 4 with the sum equal to 12, then for the second element we have 9 -> 0 -> 4 with the sum equal to 13 and so on.

One of the first thing to notice is that we have the resulting weight, but we have no information about the path taken, which must be described manually. This missing information leads to the fact that it's not possible to run the code in parallel. To be precise, it's not possible to launch parallel executions of the algorithm dividing the matrix into sub-matrices by rows, and then merge result, since we have no information about the path taken, thus the program doesn't know how to merge those results.

Indeed, the algorithm does not implement parallel execution by rows, following instead a serial execution depicted in figure 5.1: each execution will compute the weight sum for each row of a submatrix and store the result in a temporary array. At the end of the execution, this array is passed as input to the next execution, which continues the sum of weights for the next submatrix. Kernel executions continue until all the matrix is computed, so that the array will contain the final result.

On the other hand, the Rodinia implementation offers a parallel execution by columns: indeed it is possible to divide matrix by columns, and run the kernel in parallel for each group. This in not trivial, since to compute result for the values on the edges of a group of columns, data from other groups is needed. To better understand the problem, lets take the previous example, divide the matrix in two

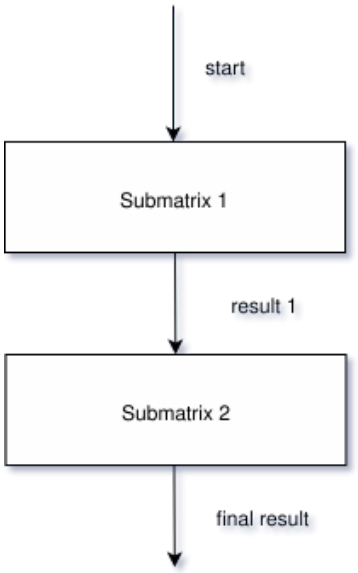


Figure 5.1: *Serial execution of the PathFinder algorithm*

groups of columns and execute the algorithm for each group

$$\begin{bmatrix} 5 & 4 & 5 \\ 0 & 8 & 2 \\ 8 & 9 & 7 \end{bmatrix} \begin{bmatrix} 7 & 0 & 3 \\ 2 & 6 & 3 \\ 5 & 9 & 0 \end{bmatrix} \tag{5.3}$$

the two results computed for each group are

$$\begin{bmatrix} 12 & 13 & 13 \end{bmatrix} \begin{bmatrix} 7 & 11 & 3 \end{bmatrix} \tag{5.4}$$

We can clearly see that the result for the third element of the first group is 13, while it should be 9 as shown in previous example, since in this case the last column of the first group can't reach values of the first column of the second group.

This problem can be solved by building overlapping groups, that is including extra columns to the left and the right of each group. Not all the data included is needed to perform the algorithm: the number of columns included is equal to the number of rows minus one, and for each row we can include one less element as we reach the bottom row, forming a sort of "ladder" as shown in example 5.5 In

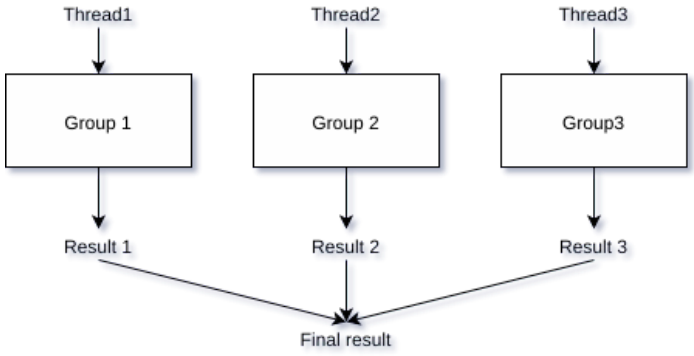


Figure 5.2: *Parallel execution of the PathFinder algorithm, dividing matrix by columns*

our example, since we have 3 rows we will introduce 2 columns for each group

$$\left[\begin{array}{ccc|cc} 5 & 4 & 5 & 7 & 0 \\ 0 & 8 & 2 & 2 & - \\ 8 & 9 & 7 & - & - \end{array}\right] \left[\begin{array}{cc|ccc} 4 & 5 & 7 & 0 & 3 \\ - & 2 & 2 & 6 & 3 \\ - & - & 5 & 9 & 0 \end{array}\right] \tag{5.5}$$

now we can compute correctly the result, which is again

$$\begin{bmatrix} 12 & 13 & 9 \end{bmatrix} \begin{bmatrix} 7 & 11 & 3 \end{bmatrix} \tag{5.6}$$

This is the approach defined in the Rodinia implementation when the number of columns provided is high, leveraging OpenCL to achieve parallel execution, as shown in figure 5.2. As we can see, each group of columns requires extra computation, since if we have n rows we need to include $n - 1$ extra columns, and this leads to an increasing computation time.

5.2 PathFinder: Porting to MANGO

PathFinder port as a benchmark for BeeR has a similar structure as the original implementation: there's a main executable whose goal is to generate matrix and execute kernels on the remote host, dividing the execution into many tasks, each one taking care of a piece of matrix. In particular, the executable will receive information about remote host, and other parameters such as the number of rows and columns of the generated matrix, and the number of rows to compute each iteration, which is called *pyramid_height* because of the shape of the matrix with extra columns resembling a reversed pyramid. The execution is organized like this: the total number of jobs to be executed it's equal to the number of rows divided by the *pyramid_height* which represents how many rows we have to

compute each iteration.

$$WorkJobs = \frac{height}{pyramid\_height} \tag{5.7}$$

The code tries to split the jobs equally between devices, so that each device has the same amount of jobs to compute.

$$JobsPerDev = \frac{WorkJobs}{Devices} \tag{5.8}$$

Moreover, these jobs are equally split among the threads available. In this implementation, threads are like work queues, handling one job and waiting for its completion before starting the next one. Of course the number of threads should always be less or equal of the number of WorkJobs, otherwise there would be some extra unused threads. The following listing shows the code used to dispatch jobs.

Listing 5.1: *"Job dispatching code"*

```
1  for(int i = 0; i < n_kernels; i++) {
2      int tid = i%n_threads;
3      std::function<void()> f = std::bind(
4              &KernelRunner::run,
5              kr,
6              dlist[i],
7              mtx,
8              vstart,
9              vend,
10             i,
11             pyramid_height,
12             pyramid_height,
13             cols);
14     std::cout << "launching kernel " << i << " on thread " << tid
           << std::endl;
15     if (tlist[tid].joinable())
16         tlist[tid].join();
17     tlist[tid] = std::thread(f);
18 }
```

Here *n_kernels* represents WorkJobs, while *n_threads* is given as a command line parameter to the benchmark. In line 2 a thread is selected in a round robin fashion, and then from line 3 it starts creating a call wrapper that thread will be able to execute. In line 15-16, code checks if the thread was already running, and if it was it waits for its completion. Finally in line 17 we start a new thread which will execute the kernel.

The *KernelRunner* class is in charge of executing the kernel leveraging *lib-*

mango and the underlying PMSL. Constructor takes the *BBQContext* instance, a pointer to the source matrix, and number of rows and columns as parameters, and it takes care of the initialization phase by registering kernel, registering buffers, creating a new task graph and perform the resource allocation.

Listing 5.2: *"Initialization code for buffer and kernels"*

```
KernelRunner::KernelRunner(
    const std::shared_ptr<mango::BBQContext> & ctx,
    int * src,
    int rows,
    int cols)
    : context(ctx.get())
    , matrix(src)
{
    kf = new mango::KernelFunction();
    kf->load(kernel_path, mango::UnitType::GN, mango::FileType::
        BINARY);
    auto kernel = context->register_kernel(KERNEL, kf, { B1 }, { B2
         });
    auto b1 = context->register_buffer(B1, sizeof(int)*rows*cols, {
        KERNEL}, {}, mango::BufferType::BUFFER);
    auto b2 = context->register_buffer(B2, sizeof(int)*cols, {}, {
        KERNEL}, mango::BufferType::BUFFER);

    tg = new mango::TaskGraph({ kernel }, { b1, b2 });
    context->resource_allocation(*tg);
}
```

This object will be initialized at the beginning of the main function, along with matrix and other data. Then during the dispatch code the *run* method is called with different sets of parameters.

Listing 5.3: *"Sub-matrix initialization"*

```
void KernelRunner::run(
        std::shared_ptr<mango::Device> dev,
        std::shared_ptr<std::mutex> mtx,
        std::shared_ptr<std::vector<double>> vstart,
        std::shared_ptr<std::vector<double>> vend,
        int current_step,
        int pyramid_height,
        int rows_to_compute,
        int cols_to_compute) {
    int array_size = pyramid_height*cols_to_compute;
    int submatrix_size = array_size*sizeof(int);

    int * submatrix = new int[array_size];
    std::memcpy(submatrix, matrix + (current_step*pyramid_height)*
        cols_to_compute, submatrix_size);
```

```
15      std::cout << "init submatrix of dimensions " << pyramid_height
            << "x" << cols_to_compute << std::endl;
16      std::cout << "starting from row " << (current_step*
            pyramid_height) << std::endl;
```

Besides the usual parameters like the *Device* object, the mutex for thread synchronization, there are two vectors of double values *vstart* and *vend*, which are used to store respectively the timestamp corresponding to the beginning of the kernel execution on the remote server, and the timestamp of the end. This values are used to calculate on the client-side kernel execution time and the overhead of the computation. Since source matrices can have very high dimensions, passing to the kernel the entire source matrix can be expensive in terms of bandwidth and time. To overcome this limit, the code sends to the remote matrix only the part of the matrix that it needs to perform the algorithm. *current_step* is the index of the current WorkJob, and it is used to compute the starting index of this submatrix:

$$StartRow = current\_step * pyramid\_height$$

$$StartIndex = StartRow * cols\_to\_compute$$

Submatrix is then copied in a temporary buffer in line 14. In the last part of the function, after setting up arguments, kernel is started and the wait is performed on the remote kernel completion event.

Listing 5.4: *"Kernel execution and synchronization"*

```
1     auto a1 = new mango::BufferArg(b1);
2     auto a2 = new mango::BufferArg(b2);
3     auto a3 = new mango::ScalarArg<int>(rows_to_compute);
4     auto a4 = new mango::ScalarArg<int>(cols_to_compute);
5     auto a5 = new mango::EventArg(b2->get_event());
6
7     auto args = new mango::KernelArguments({ a1, a2, a3, a4, a5 },
          kernel);
8
9     b1->write(submatrix, submatrix_size, dev);
10
11    auto ev = context->start_kernel(kernel, *args, nullptr, dev);
12    ev->wait();
13
14    delete[] submatrix;
15 }
```

The kernel implementation follows the same approach of the original, with some differences and simplifications. First of all, to be able to use buffers kernel needs to open shared memory objects referenced by the names passed as param-

eters. Once initialized, kernel can access this read/write memory using pointer returned by the *mmap* function.

Listing 5.5: *"Shared memory initialization in kernel"*

```
1     ...
2     int result_fd = open_shm(argv[2]);
3     if (result_fd == -1) {
4         printf("failed to open shm %s\n", argv[3]);
5         return 1;
6     }
7     int* result = (int*)get_mmap(result_fd, sizeof(int)*cols);
8     if (result == MAP_FAILED) {
9         printf("mmap failed for %s\n", argv[3]);
10        return 1;
11    }
12    ...
```

Before executing the algorithm, a *temp* buffer is initialized with the content of the first row. This buffer will store the temporary result for each iteration of the algorithm.

Listing 5.6: *"Pathfinder algorithm"*

```
1  for (int j = 0; j < rows-1; j++) {
2      for (int i = 0; i < cols; i++) {
3          W = i - 1;
4          E = i + 1;
5          U = i;
6          if (W < 0)
7              W = 0;
8          if (E >= cols)
9              E--;
10         left = temp[W];
11         right = temp[E];
12         up = temp[U];
13         int shortest = MIN(left, up);
14         shortest = MIN(shortest, right);
15         int wall_index = ( (j+1)*cols ) + i;
16         result[i] = shortest + wall[wall_index];
17     }
18     memcpy(temp, result, sizeof(int)*cols);
19 }
```

The algorithm will parse every row of the matrix starting from the second one, and for each value it will compute the shortest path when going straight ahead (the *up* value), diagonally to the left (the left value) or diagonally to the right (the *right*) value). Those values contains the accumulated weight for the previous rows, and the current value is added to the smallest one. Notice that if

we are dealing with the first columns, left and up values will be equal. Same goes for the last column, where up and right will reference the same value. At the end of each iteration, the *result* array contains the updated weights that needs to be copied into the *temp* buffer, so that next iteration can correctly continue. When the algorithm ends, no action needs to be performed since the shared memory object pointed by the *result* pointer will contain the final result for the submatrix, and the client will be able to read it.

5.3 PathFinder: Parallel Greedy Version

To better leverage the distributed nature of the BeeR daemon, a parallel version of PathFinder is needed. As discussed before, it is possible to divide the execution into groups of columns, execute the algorithm independently for each group and finally merge result, introducing more complexity. To keep things simple, and because the main goal of a benchmark is to simply stress the server in order to evaluate its performance, a simpler approach was adopted: divide the matrix by rows and apply the algorithm in parallel. Once all partial result are computed, they should be merged into the final result which of course can't be the optimal one, but it will represent a good approximation of it.

Searching for a way to merge result, a few options were evaluated: (1) add up each element of the partial results one by one, since those arrays have the same length, (2) re-apply the algorithm on the partial results, like they were rows of a matrix being computed. There's not much difference between the two methods, since both solutions won't produce exact results, but an estimate of the weight. Even is the first option is very simple to implement, the second option was finally implemented into the benchmark, since it is possible to reuse kernel code to merge results into the final array without much implementation effort.

Moreover, this estimate approach allows us to carry out another evaluation, namely the accuracy of the algorithm: after running the algorithm for each submatrix and merging temporary result into the final one, we will compare one by one the results of the estimate array against the results of the exact array computed directly with the original implementation.

The parallel implementation needed a small number of changes to the original code. The original serial implementation worked with a single result buffer and updated it on each WorkJob. Now we need to store a buffer for each WorkJob, thus having a list of buffer. This array is stored in the *KernelRunner* object, and initialized with a new method *init_result*.

Listing 5.7: *"Initialization of the results array"*

```
1  void KernelRunner::init_result(int steps, int cols) {
```

```
2       result = new int*[steps];
3       for (int n = 0; n < steps; n++) {
4           result[n] = new int[cols];
5       }
6   }
```

At the end of the computation, results from this array should be merged using the chosen approach: this is done with a new function called *merge_results*, which applies PathFinder on the intermediate results, and writes the final results in the supplied *final_res* buffer.

Listing 5.8: *"Code for merging different results into a single array"*

```
1   static void merge_results(int ** res, int * final_res, int n_res,
        int res_size) {
2       int temp[res_size];
3       int W, E, U, shortest;
4       for (int n = 0; n < res_size; n++) {
5           temp[n] = res[0][n];
6       }
7       for (int i = 1; i < n_res; i++) {
8           for (int j = 0; j < res_size; j++) {
9               W = j-1;
10              U = j;
11              E = j+1;
12              if (W < 0) W = 0;
13              if (E >= res_size) E = res_size-1;
14              shortest = std::min(temp[W], temp[U]);
15              shortest = std::min(shortest, temp[E]);
16              final_res[j] = shortest + res[i][j];
17          }
18          for (int j = 0; j < res_size; j++) {
19              temp[j] = final_res[j];
20          }
21      }
22  }
```

CHAPTER 6

Experimental Evaluation

Having both the BeeR framework and the benchmark ready, we tested the execution of the application on a network of embedded development platforms. The PathFinder algorithm turned out to be a good test bench, because it can stress the server with the computation but also evaluate the parallelism by launching different jobs on different devices. Section 6.1 will contain a brief description of the devices used for the test, and how the test was organized. Then Sections 6.2, 6.3 and 6.4 provide a report of the three main classes of tests performed, which are respectively *execution time*, *overhead* and *accuracy*.

6.1 Devices and methodology

Two devices were used to carry out the evaluation: and Odroid XU-3 and a Freescale i.MX 6 SABRE. The Odroid XU-3 is a board containing a Heterogeneous Multi Processing solution, leveraging ARM big.LITTLE technology: this technology allows the coupling of a slower and more battery-saving processor cores (LITTLE) with more powerful ones (big). In particular the ODROID XU-3 ships with a Samsung Exynos5422 Cortex-A15 quad-core big CPU and a Cortex-A7 quad core as the LITTLE one, equipped with 2 GB of LPDDR3 RAM. This architecture allows the board to use the less power-hungry CPU

Size					
2 Thread			4 Thread		8 Thread
2 Jobs	4 Jobs	8 Jobs	4 Jobs	8 Jobs	8 Jobs

Table 6.1: *Evaluation plan for a given matrix size*

when facing a low system load, and transitioning to the big one, when the load becomes higher. The Freescale Smart Application Blueprint for Rapid Engineering (SABRE) board allows developers to test the capabilities of different i.M6 processors based on the ARM Cortex A9 architecture. The test was organized in three different configurations: first evaluate only the ODROID board, second evaluate only the Freescale board, and third evaluate both device together, making sure to split equally the work between the two devices.

The evaluation is composed by different tests considering different combinations of the following parameters: (a) **Size** of the source matrix which inevitably affects the computation time of the algorithm, (b) number of **Jobs** to launch, which also tells how many sub-matrices should be obtained from the source and (c) number of **Threads** acting as work queues, which can be used to control the level of parallelism of the execution. As stated in chapter 5, the number of threads should be less or equal to the number of jobs, and to be sure that each threads gets the same amount of work, jobs should be equally split among them. Starting from these requirements we defined the following plan for the evaluation, shown in Table 6.1.

This plan expects a total of six different tests, for each size chosen: first tests will use two threads, meaning that at most two jobs are executing at the same time, regardless of the configuration considered. Inside this class, we will carry out the evaluation using two, four, or eight jobs, in order to consider different levels of parallelism limited by using only two queues. Second class of tests will consider four threads coupled with four and eight jobs, and last we consider the maximum level of parallelism achieved by using eight threads and eight jobs. This plan evaluated for different sizes and different device configuration allows us to experiment a wide number of scenarios.

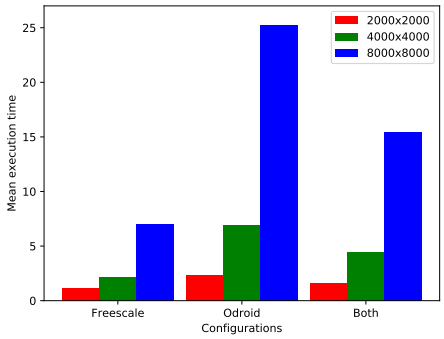
Moreover, the goal of these tests is to extract three classes of results, namely (a) **execution-time** to be intended as the overall time of the pathfinder execution, (b) **overhead** considered the time spent executing kernels compared to the overall time, (c) **accuracy** of the estimated results with respect to the correct results.

6.2 Execution Time

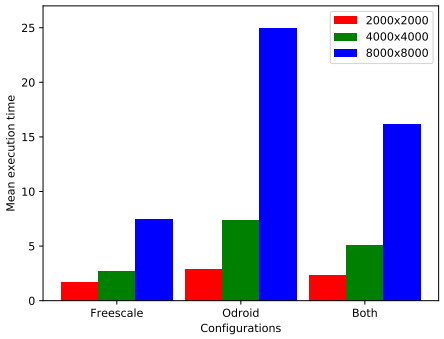
Evaluating the execution time is a rather simple task, since there are many tools available in GNU/Linux systems for this specific task. Since PathFinder executable allows to control many aspects of the benchmark through command line parameters, the evaluation can be carried out using a shell script, where it is possible to specify a command for each execution case. To measure time, command can be passed as parameter to the *time* program, not to be confused with the bash built-in command. This tool can execute a command and measure its resources, as well as some statistics regarding the execution. In particular, it can measure the total number of seconds the program spent in kernel mode and user mode, number of IO operations, number of socket messages sent and received, and the overall clock time elapsed from the start to the end of the execution. Shell script launches PathFinder for each execution case, measuring the execution and writing to the standard output the time statistics. This data is then collected in log files in a separate folder, where each file contains different runs for the same case.

The test was applied considering three matrix sizes: (1) 2000x2000, (2) 4000x4000 and (3) 8000x8000. For each size, we applied the evaluation plan described in Table 6.1, consisting of six runs with different values of *threads* and *jobs*. A first comparison can be made by considering different device configurations and sizes of the source matrix using the same plan, as shown in figure 6.1.

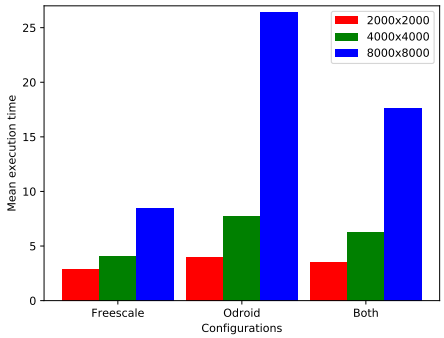
If we analyze the cases by plan, we can clearly see that the worst performance is given by selecting 2 threads and 8 jobs, as expected since the number of threads limits the number of concurrent tasks running at the same time. On the other hand, the plan using 4 threads and 4 jobs gives the best performance, since we are splitting the matrix into 4 pieces, and this will help both the computation time and the transfer of buffers from the host to the remote device. Moreover, the number of threads can handle simultaneously all the available jobs. Contrary to the expectation, the 8 threads and 8 jobs case gives slightly lower performance compared to the 4-4 case, mostly with the Freescale-only configuration: this happens because the server was configured to spawn a maximum number of threads equal to the number of available CPUs which is 4. So by receiving 8 requests, 4 are handled and 4 are queued. On the other hand the Odroid-only configuration gives slightly better performance on the 8-8 case with respect to the 4-4, since the board can handle all 8 requests simultaneously. The other two cases, namely 2-4 and 4-8 give very similar result but without any particular gain. It is also possible to notice that while overall Freescale gives better results and ODROID



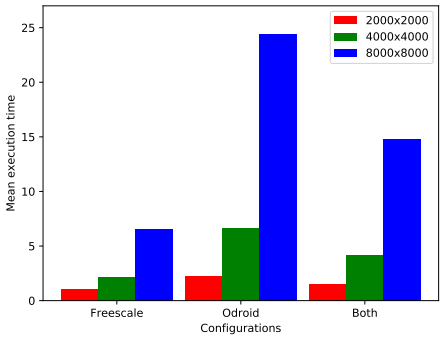
(a) *2 threads and 2 jobs*



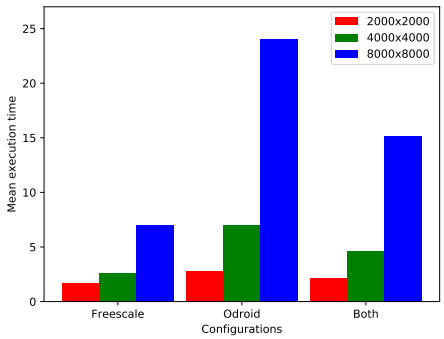
(b) *2 threads and 4 jobs*



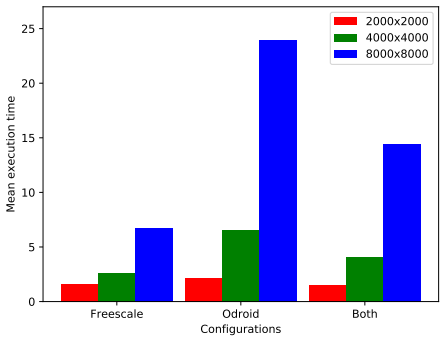
(c) *2 threads and 8 jobs*



(d) *4 threads and 4 jobs*



(e) *4 threads and 8 jobs*



(f) *8 threads and 8 jobs*

Figure 6.1: *Comparison between execution time organized by execution plan. Each graph reports results for each input data size, identified with red colour for the small, green for the medium and blue for the big, grouped by configuration on the x axis*

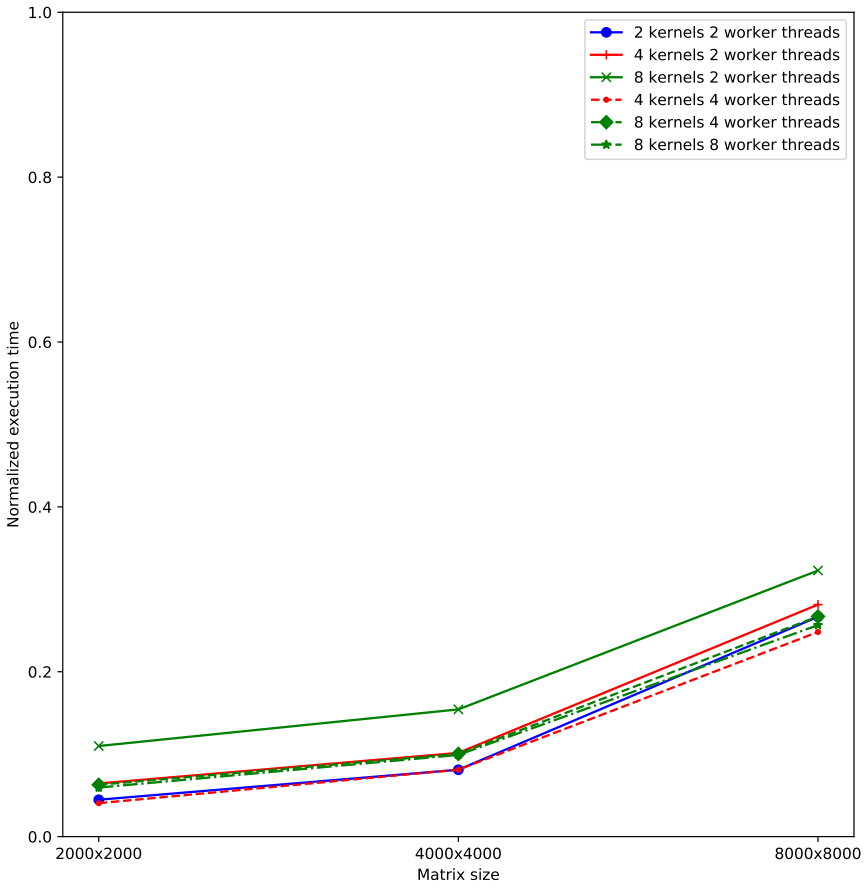


Figure 6.2: *Average execution time when using Freescale*

gives the worst, the split configuration provides an average performance between the two.

If we consider the overall results for the Freescale-only configuration, shown in the 6.2 graph with a normalized execution time, we can see that the variance of the results for each size is almost the same. Looking at the single plan in detail, an interesting thing to notice is the behaviour of the 2-2 plan: for small matrices it is slightly worse than the 4-4, it becomes on par with plan 4-4 for medium-sized matrices, but it slows down with big-sized matrices, becoming the third fastest plan. This happens because the cost of the data transfer becomes higher as the matrix size increasing, as it takes a considerable amount of time to write/read remote buffers (this result will be discussed in Section 6.3).

In the Odroid-only configuration, while having almost the same results for

small matrices, there's a huge increase in the time as the input data size becomes high, reaching its maximum value with the 8000x8000 matrix and the 2-8 plan. These high values can be explained considering that the ODROID board is not always exploiting the big processor, thus keeping the load on the LITTLE one. Whilst it is possible to force the execution on the high performance cores, tests were executed keeping the default behaviour. Another interesting thing is the fact that the 8-8 case, compared to the Freescale configuration, becomes the best performer. In this case, we have the execution of eight concurrent tasks, and the scheduler on the ODROID board can decide to migrate some of these to the high performance processors, leveraging the big.LITTLE architecture. The overall execution time for this plan remains slow compared to other configurations, since it is limited by the tasks running on the slower processor, but it gets a little speedup thanks to the tasks running on the faster one. Furthermore, the 2-2 plan follows the same behaviour found in the previous configuration, with the only difference that it becomes the second slowest plan with the 8000x8000 source matrix, where the data set for the single kernel execution becomes sizable, and the computation of the two Jobs is handled exclusively by the power saving CPU. Furthermore, values in each configuration are closer to each other with respect to other cases, showing that ODROID gives slightly more uniform results.

Finally the split configuration shown in Figure 6.4 shows an average behaviour between the Freescale and the Odroid one. The trend is the same as the previous ones, with the 2-8 and the 2-4 being the low performers, 4-4 and 8-8 competing for the best execution time and the 2-2 starting good and then slowing down.

6.3 Overhead

One of the things we may notice from previous tests is that the execution time depends on different factors, from the transfer of buffers to the execution of different concurrent instances of the kernel. Each factor can be more or less influential, based on the input data size, jobs and threads parameters of the current instance. In general the execution time of the PathFinder benchmark can be divided into three main categories:

- *Kernel Execution Time* which is the overall time spent by the server executing all the kernel, from the start of the first task to the end of the last one.
- *Transfer Time* is the time spent by reading buffers from the server and writing buffers to the server. It may be negligible for small matrices, but also

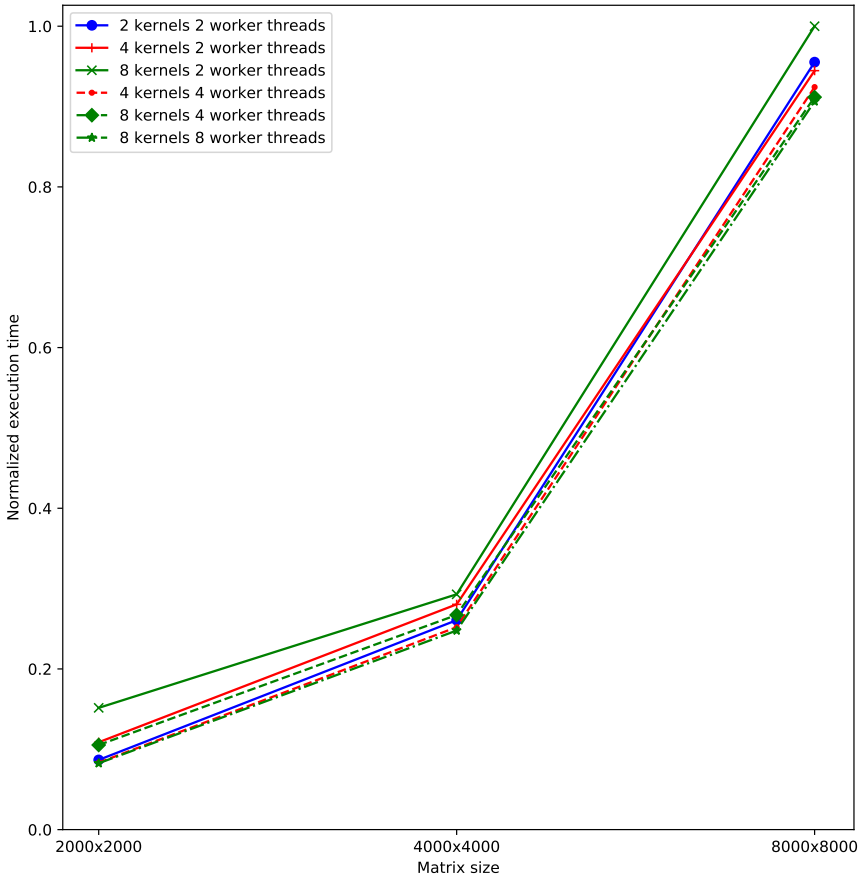


Figure 6.3: *Average execution time when using ODROID XU-3*

may take a considerable amount of time when working with big matrices.

- *Secondary Execution Time* consisting of all the other actions carried out by the benchmark, such as matrix initialization, results merging and basic server communication. It is usually negligible, but for huge input data it may constitute a considerable portion of the overall time.

Kernel Execution Time should be considered as the time that elapses between the start of the first kernel and the end of the last instance of the kernel, as shown in Figure 6.5. To avoid including extra time due from connection delays, it should be computed on the server-side. To enable BeeR to collect this additional data, some changes were required: first of all, the *Task* object must save the time point in which kernel starts running and the time point in which it ter-

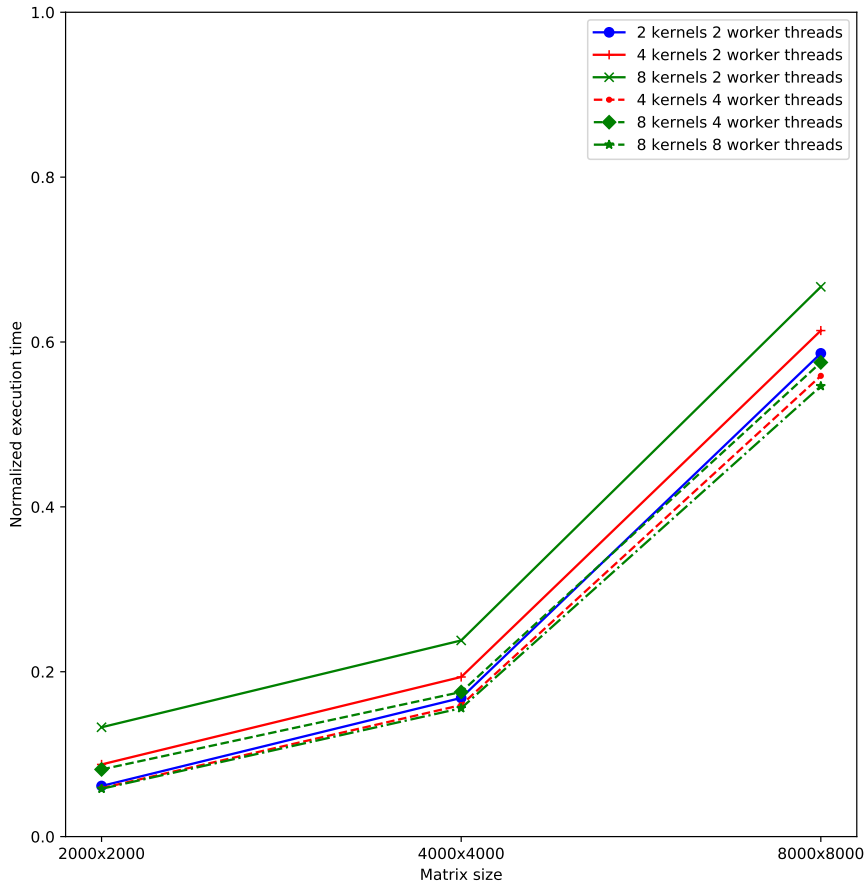


Figure 6.4: *Average execution time when using both devices*

minates, called respectively *kstart* and *kend*. The *kstart* time point is initialized right before running the sub-process, while the *kend* point after the wait call of the sub-process.

Listing 6.1: *"Time points being saved before and after the execution"*

```
1 ...
2 kstart = std::chrono::high_resolution_clock::now();
3 proc->run(arguments->generate());
4 ...
5 int status = proc->wait();
6 kend = std::chrono::high_resolution_clock::now();
7 ...
```

To allow client to fetch this data, two additional requests where added to the server, namely *get_time_start()* and *get_time_end()*, which basically converts

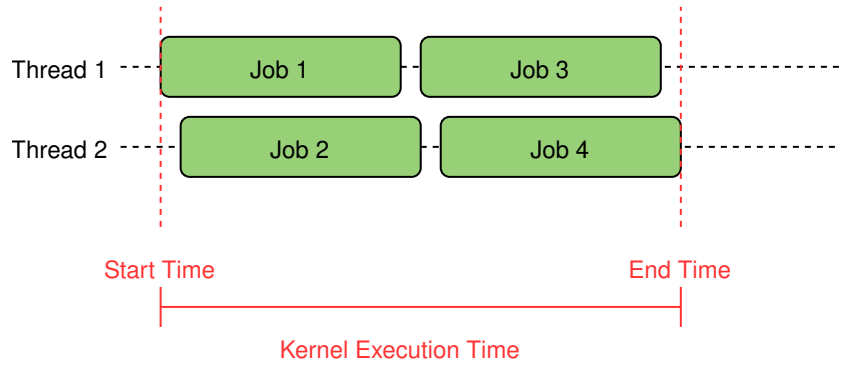


Figure 6.5: *Kernel execution time is the time from the start of the first job to the end of the last job*

start and end time points into timestamps, and transfers them back to the client. It is important to notice that these times refers to the current kernel, not the first one or the last one. Indeed the client must fetch the timestamp for the start of each kernel, the timestamp for the end of each kernel and save them into two separate lists. Finally, when computing the final kernel execution time, client will pick the lowest value of the start list and the highest value of the end list. The difference between those two values represents the *Kernel Execution Time*.

Listing 6.2: *"Computing kernel execution time from the timestamps"*

```
1 double min = (*vstart)[0];
2 double max = (*vend)[0];
3 for (double d : *vstart) {
4     if (d < min) min = d;
5 }
6 for (double d : *vend) {
7     if (d > max) max = d;
8 }
9 std::cout << "milliseconds overhead:  " << max-min << std::endl;
```

To measure how much time the benchmark spends executing kernels, it is now possible to compute the execution overhead, intended as the percentage of time the benchmark spends in executing kernels with respect to the overall execution time. A lower overhead does not mean that the benchmark was faster, but simply means that the kernel execution time is negligible with respect to the time spent doing other computation. Conversely a high value means that the kernel execution time occupies a large part of the overall execution time. Results are shown in Figure 6.6.

As expected, the plan 2-8 has the highest values in any case, since the benchmark spends most of the time executing kernels, and this execution is slowed down by the two threads having to handle four jobs each like a FIFO queue.

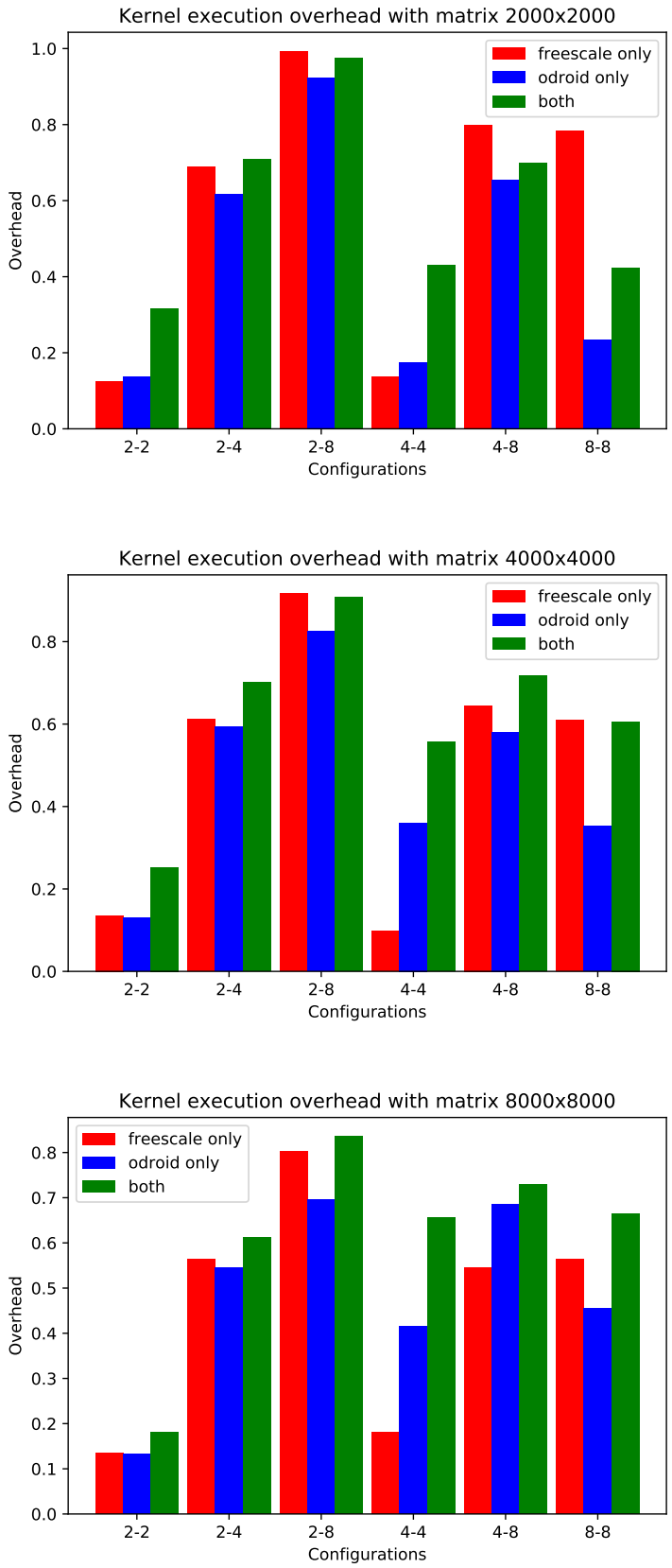


Figure 6.6: *Kernel execution overhead for different input data size*

Lowest values are represented by the 2-2 case since it needs to execute only two instances of the algorithm on two pieces of matrix, and these instances runs in parallel thanks to the two threads. An interesting case to analyze is the 4-4 one: using a small source matrix, as one would expect it gets almost the same values as the 2-2, since in both cases we have 4 tasks launched almost at the same time running in parallel. But as the size of the source matrix increases, we notice higher values for the Odroid-only and split configurations, while the Freescale one remains unchanged. The reason for this gain can be explained by the Odroid executing the tasks on the low power processor, causing a slowdown on the execution. Other plans 2-4, 4-8, 8-8 show almost the same values in every matrix size, ranging from 50% to 60%.

6.4 Accuracy

As explained in Section 5.3 the greedy implementation of the PathFinder algorithm provides an estimate results for the shortest path. In order for the benchmark to be relevant, it is necessary to demonstrate that these results represent a good estimate of the correct results. The latest part of the evaluation strategy included the assessment of the accuracy of the greedy algorithm, by comparing the two classes of result, one from the greedy version and one from the original version.

For each value of the result array, we compute the delta as the difference between the estimate result and the correct one

$$\Delta = |Estimate - Correct| \tag{6.1}$$

and calculate the accuracy as in equation 6.2.

$$Accuracy = 1 - \frac{\Delta}{Correct} \tag{6.2}$$

With this kind of approach, we may evaluate how much the greedy algorithm is reliable as the number of jobs increases, because by dividing the source matrix in different pieces, we lose more information as this number of pieces grows. Moreover, we may assess if the behaviour is the same for any matrix, or if the results varies as a function of the size of the input. The PathFinder benchmark generates a random matrix starting from a fixed seed value: the generated matrix will always be the same one, and the accuracy test can be carried out without any changes to the source code. Figure 6.7 shows the minimum value of accuracy with different number of Jobs, for the smallest input matrix: as expected the value slightly decreases as the number of jobs increases, because we are splitting

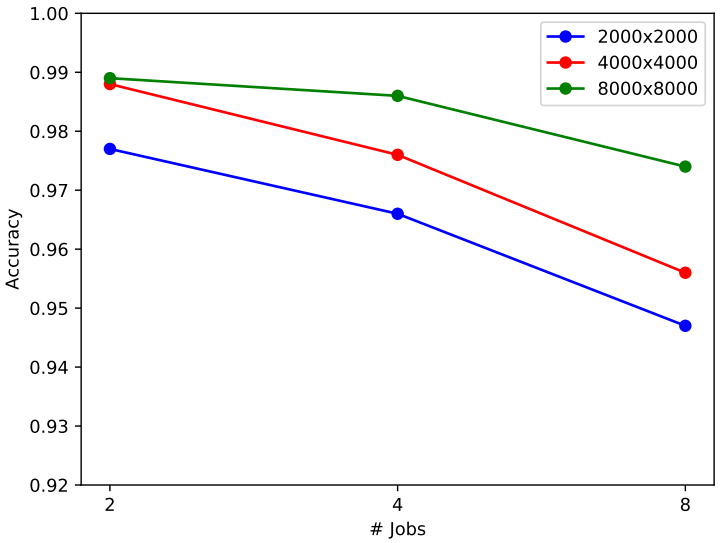


Figure 6.7: *Minimum accuracy value with different Jobs*

the input into more pieces, and since the estimate is calculated when merging results, the more result we have, the less accurate is the estimate. As we can see in the graph, the lowest value represented by 8 jobs for the small input matrix, is still a good result, since its value is 0.947. As the size of the matrix increases, we notice the same trend described before, but also an increase on the single values: with bigger input data sizes, regardless of the number of pieces produced by the split, the kernel is going to work with bigger data sets and it is able to produce more precise results.

CHAPTER 7

Conclusions and Future Work

In this chapter we discuss some final points about the solution proposed, as well as some improvements and additional features the framework needs to implement. In particular in Section 7.1 some general considerations about the work and the results obtained are presented, while Section 7.2 describes the features that this project lacks and that will be the subject of future works.

7.1 Conclusions

In this thesis we explored a distributed computing approach for High Performance Computing applications, integrated with the framework provided by the MANGO project. This approach consists in different instances of the *BeeR* daemon running on different devices across the network, which can be exploited by client applications in order to offload part of its tasks. Considering alternative solutions analyzed in the state-of-the-art, *BeeR* differs in the fact that it does not rely on a central authority managing resource allocation and job dispatching, instead each application running on a different General Purpose node can dynamically use any available server instance it needs.

The implementation of a benchmark application helped us evaluate the framework in terms of performance and reliability. The benchmark called *PathFinder*

was ported from the *Rodinia Benchmark Suite*, which is a collection of heavy parallel GPU benchmarks, making it a good candidate for our purpose. Thanks also to the similarities between MANGO and OpenCL programming model, the implementation proceeded without problems, and in the final implementation the algorithm was slightly changed in order to have a fully parallel tool. Evaluation of the execution time showed that the benchmark will benefit from a high parallel configuration in almost all cases, while low parallelism brings higher values of time. Furthermore this modification of the algorithm enabled us to evaluate also the accuracy of the results with respect to the original PathFinder, which showed to be quite high. Tests also demonstrated that with a good degree of parallelism the kernel overhead is quite small. To calculate this overhead, additional APIs were added to the BeeR server to be able to retrieve start and end time for each kernel execution.

7.2 Future Works

One of the most important feature for a distributed computation framework is the encryption of the communication between client and servers. Right now, client application will connect to running instances of the BeeR daemon without any kind of security and authentication, using a plain TCP connection. A crucial feature to be implemented is the communication encryption; this could be made possible by algorithms like the Diffie-Hellman key exchange method, where two entities, unknown to each other before, can securely establish a shared secret over a public, insecure channel. This secret can be used to encrypt subsequent communications using a symmetric key cipher. A third entity may exchange two different keys with the two district parties of the communication, masquerading as one to the other, therefore it should be used in conjunction with a signature algorithms such as RSA or DSA to verify the authenticity of the data source/destination.

Furthermore, in the current prototype remote devices have to be manually registered in the client application. In this case a service discovery protocol can help to automatically find all the available devices and connect to them. This introduces another advantage, because it is possible to remove the extra code introduced to register device, and manage external devices internally. Doing so, we could keep the original MANGO API interface without extra functions.

Another aspect to be improved is the resource management support of the daemon: in particular the daemon should be able to report its resources to the client application, which in turn can use this information during the resource allocation phase. In the current state, remote elements of the task graph are basi-

cally "placeholders", and when their functions are called they will send a request to the remote server, without taking into account device state and resources: in future works this behaviour should be tightly integrated within the *libmango* and the instance of BarbequeRTRM running on the GN node should be able to take into account remote resources.

Bibliography

[1] That 'internet of things' thing. https://www.rfidjournal.com/articles/view?4986.

[2] The internet of things. https://www.forbes.com/global/2002/0318/092.html.

[3] Gartner. https://www.gartner.com.

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[5] T. Taleb, S. Dutta, A. Ksentini, M. Iqbal, and H. Flinck. Mobile edge computing potential in making cities smarter. *IEEE Communications Magazine*, 55(3):38–43, March 2017.

[6] T. Taleb and A. Ksentini. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19, September 2013.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[8] Giuseppe Massari Michele Zanella and William Fornaciari. Back to the future: Resource management in post-cloud solutions (in publishing). In *Proceedings of ACM INTESA Workshop (INTESA'18)*. ACM, 2018.

[9] R. Ranjan, B. Benatallah, S. Dustdar, and M. P. Papazoglou. Cloud resource orchestration programming: Overview, issues, and directions. *IEEE Internet Computing*, 19(5):46–56, Sept 2015.

[10] Boinc. https://boinc.berkeley.edu/.

[11] Yaoxue Zhang and Yuezhi Zhou. Transparent computing: A new paradigm for pervasive computing. In Jianhua Ma, Hai Jin, Laurence T. Yang, and Jeffrey J.-P. Tsai, editors, *Ubiquitous Intelligence and Computing*, pages 1–11, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[12] D. F. Parkhill. *The challenge of the computer utility*. Addison-Wesley Pub. Co., 1966.

[13] D. Díaz-Sánchez, A. M. López, F. Almenares, R. Sánchez, and P. Arias. Flexible computing for personal electronic devices. In *2013 IEEE International Conference on Consumer Electronics (ICCE)*, pages 212–213, Jan 2013.

[14] H. Ba, W. Heinzelman, C. Janssen, and J. Shi. Mobile computing - a green computing resource. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 4451–4456, April 2013.

[15] Seti@home. `https://setiathome.berkeley.edu/`.

[16] Folding@home. `https://foldingathome.org/`.

[17] C. Funai, C. Tapparello, H. Ba, B. Karaoglu, and W. Heinzelman. Extending volunteer computing through mobile ad hoc networking. In *2014 IEEE Global Communications Conference*, pages 32–38, Dec 2014.

[18] Z. Dong, L. Kong, P. Cheng, L. He, Y. Gu, L. Fang, T. Zhu, and C. Liu. Repc: Reliable and efficient participatory computing for mobile devices. In *2014 Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 257–265, June 2014.

[19] Mustafa Y Arslan, Indrajeet Singh, Shailendra Singh, Harsha V Madhyastha, Karthikeyan Sundaresan, and Srikanth V Krishnamurthy. Computing while charging: Building a distributed computing infrastructure using smartphones. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 193–204. ACM, 2012.

[20] F. Büsching, S. Schildt, and L. Wolf. Droidcluster: Towards smartphone cluster computing – the streets are paved with potential computer clusters. In *2012 32nd International Conference on Distributed Computing Systems Workshops*, pages 114–117, June 2012.

[21] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[22] K. Habak, M. Ammar, K. A. Harras, and E. Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 9–16, June 2015.

[23] M. Ryden, K. Oh, A. Chandra, and J. Weissman. Nebula: Distributed edge cloud for data intensive computing. In *2014 IEEE International Conference on Cloud Engineering*, pages 57–66, March 2014.

[24] Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. Managing heterogeneous resources in hpc systems. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 7–12. ACM, 2018.

[25] Boost libraries. `https://www.boost.org/`.