2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

COMPSs-Mobile: parallel programming for Mobile-Cloud Computing

Francesc Lordan
Barcelona Supercomputing Center, BSC-CNS
Universitat Politècnica de Catalunya, UPC
Barcelona, Spain
francesc.lordan@bsc.es

Rosa M. Badia
Barcelona Supercomputing Center, BSC-CNS
IIIA - CSIC
Barcelona, Spain
rosa.m.badia@bsc.es

***Abstract*—The advent of Cloud and the popularization of mobile devices have led us to a shift in computing access. Computing users will have an interaction display while the real computation will be performed remotely, in the Cloud. COMPSs-Mobile is a framework that aims to ease the development of energy-efficient and high-performing applications for this environment. The framework provides an infrastructure-unaware programming model that allows developers to code regular Android applications that, transparently, are parallelized, and partially offloaded to remote resources.**

This paper gives an overview of the programming model and describes the internal components of the toolkit which supports it focusing on the offloading and checkpointing mechanisms. It also presents the results of some tests conducted to evaluate the behavior of the solution and to measure the potential benefits in Android applications.

***Keywords*-Mobile Cloud Computing Framework, Parallel programming model, Android, Offloading, Checkpointing**

I. Introduction

In the recent years, we have assisted to a revolution in IT technologies. On the first mile, mobile devices permanently connect people to IT services; on the other end, Cloud technologies enable the access to computation as a utility. Mobile Cloud Computing (MCC) brings together the benefits of both: the immediacy of access of mobile devices and the infinite computing capacity of the Cloud.

Developing applications that fully exploit MCC is not straight-forward. Programmers face the concerns of parallelizing the application and distributing its components. Besides, developers deal with the rapid variability of the network conditions induced by the high mobility of the mobile device. To not harm the performance and the energy-efficiency, applications should rapidly adapt their behaviour to the environmental conditions. Facing these issues requires a high level of expertise, and dealing with them means to increase the development time of the application.

This paper presents COMPSs-Mobile: a framework that eases the coding of MCC applications by freeing the developer of all these concerns. Its main contributions are the redesign of the COMPSs runtime to target MCC environments; the checkpointing mechanism; temporal, energetic and economic models to decide whether to offload application computations, and the validation on real platforms.

II. COMPSs-Mobile Overview

The core of the COMPSs-Mobile framework is a programming model that abstracts application developers from the parallelization and distribution details. Developers code in a sequential fashion making no reference to the infrastructure nor any particular API. At compilation time, the code is modified to insert a set of invocations to a runtime system that manages the partitioning and deployment of the application on the underlying infrastructure.

A. Programming Model

The COMPSs programming model [1] aims to allow developers to write applications without being aware about the parallelism or infrastructure details. COMPSs applications are considered composites of methods that run parallelly; each component is called Core Element (CE); and the whole composition, Orchestration Element (OE).

To pick a method as a CE, programmers declare the method in the Core Element Interface (CEI) along with a *@Method* annotation indicating the implementing class and some *@Parameter* annotations to describe of how the CE operates on each data it accesses (parameters) specifying its type and directionality (in, out, in-out).

B. Application Modification and Packaging

Android applications are written in Java and bundled in Android packages for distribution. The building process starts with the creation of a Java class to access non-source code entities (Resource Manager) and all the proxy-stub classes required for interprocess communications (Pre Compiler). Java classes are compiled to generate Java bytecode (Java Builder) that is translated into Dalvik bytecode and bundled with the resources into the package file.

To parallelize and distribute the sequential code, CE invocations are replaced by asynchronous tasks whose executions are orchestrated by a runtime toolkit. Similarly, all accesses to remote values are instrumented so the runtime fetches them from the node. For this purpose, we have added a step to the building process after the Java Builder, the Parallelization, which replaces the original Java classes by instrumented versions of them and adds to the bundle all the components required by the runtime library. To instrument

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.16

IEEE computer society

the code, the framework leverages on Javassist [2], a library for Java class editing. After that, the packaging process progresses as done for any regular Java application.

C. Programming Model Runtime

When the final user launches the application, it executes the instrumented calls that invoke a runtime toolkit that orchestrates the CE executions aiming to fully exploit the application parallelism while guaranteeing sequential consistency.

The runtime library is two-fold. Its front-end registers accesses to private data and intercepts CE invocations; it is instantiated in every COMPSs-Mobile application, and its code is executed by the threads of the application. The back-end manages accesses to shared data, detects data dependencies between tasks and orchestrates their executions; a unique instance of it is deployed as a Android service in an independent process.

Finally, each remote worker node hosts a pool of threads that execute the tasks submitted by the runtime. Worker nodes are responsible for fetching all the input data required to run tasks, notifying the creation of their results and transferring these values to other nodes that need them.

D. Offloading Mechanism

To decide whether a task runs locally or on a surrogate, the library has an engine that leverages on three models to predict the energetic, economic and temporal cost of hosting a task execution on the mobile and offloading it. With accurate predictions, the engine evaluates the improvement/-worsening of each parameter and takes a decision according to a three-variable (representing the improvement for time, energy and cost) inequality. To feed the models with the variables described in Table I, task executions are profiled to make a statistical analysis of each CE.

Data transfers are one of the most influential factors on the economic, energetic and temporal cost of running an application. By providing workers with a scheduler that plans the execution of tasks, they can overlap the fetching of input values for a task with the computation of other tasks.

Variable	Description
W_i	Number of CE i tasks waiting to run locally
WR	Waiting time on offloaded resources
LT_i	Execution time for a CE i task on the phone
RT_i	Execution time for a CE i task on the surrogate platform
S_{IL}	Size of the input data to be transferred to the phone
S_{IR}	Size of the input data to be transferred to the surrogates
S_{OR}	Size of the output data to be transferred from the surrogates
C_{IN}	Price per byte to download data to the phone
C_{OUT}	Price per byte to upload data from the phone
E_{IN}	Energy per byte to download data to the phone
E_{OUT}	Energy per byte to upload data from the phone
E_i	Energy spent to run a CE i task on the phone

Table I
DESCRIPTION OF THE VARIABLES PROVIDED BY THE RUNTIME LIBRARY

A data-sharing platform across workers enables a reduction on the mobile energy consumption. Currently, the mobile node hosts a data directory. When a value is generated, the creating node notifies the data generation to the mobile device which registers the data availability in the directory. Whenever a node requires that value, it queries the locations of that data and requests its transfer to any hosting node.

Table II contains the equations used for each prediction when processing a CE i task t. To compute the timespan, the model considers two aspects: the waiting time before the computation starts and the actual computation time. Since data transfers overlap with the computation of other tasks in both cases, they are not considered in the model. For the local economical cost, it only considers the cost to transfer back the data values that are not on the phone yet; whereas for the remote case it considers the cost shipping the input data only available at the phone and transferring back the results. For the energy prediction, it uses the energetic cost of transmitting/receiving instead of the price. For the local case, it also considers the energy spent on the task computation.

	Local	**Remote**
Time	$\sum_{j=1}^{N}(W_j * LT_j) + LT_i$	$WR + RT_i$
Cost	$S_{IL} * C_{IN}$	$S_{IR} * C_{OUT} + S_{OR} * C_{IN}$
Energy	$S_{IL} * E_{IN} + E_i$	$S_{IR} * E_{OUT} + S_{OR} * E_{IN}$

Table II
MODELS EQUATIONS

E. Checkpointing Mechanism

When the mobile device reads a data value generated by an offloaded task, the main code execution waits for the runtime library to obtain the actual value. Due to the high mobility of the mobile device, connection losses are likely. To recover from them and continue with the execution, the mobile device runs locally the task that produces the value, which at its time may require other values. This mechanism results in a back-tracking process that only stops when all the input data required by a task exists in the local device.

To avoid a full re-execution of the application, the runtime transfers back some values to establish some checkpoints. These values are selected according to a graph-partioning into blocks. The runtime analyses each block, identifies its output values and transfers them back to the mobile as soon as they are available. To enable this backtracking procedure, a task can not be removed from the dependency graph until all its output values have been transferred back to the mobile, or the result of all its successors have been saved and the values can not be accessed by the application in the future.

III. Evaluation

To evaluate the behavior of the COMPSs-Mobile toolkit on different situations, we have ported a scientific application: HeatSweeper; and executed it considering two different scenarios where COMPSs-Mobile uses resources within a local network or geographically distributed resources.

A. Use Case: HeatSweeper

HeatSweeper is an intensive search algorithm that optimizes the location of 1-to-N heat source to minimize the time to warm-up the whole surface to certain temperature. For this purpose, the algorithm runs a set of heat transfer simulations, each one encapsulated in a *simulate* task that generates a report describing the result of the simulation. In a second phase, the algorithm selects the best performing configuration by comparing pairs of reports in *getBest* tasks.

We run two different configurations to optimize the placement of up to two heat sources: low-resolution representing short-lasting applications, with 9 possible locations and short simulations (up to 50 iterations each); and high-resolution emulating large computations, with 25 different spots on the surface and long simulations (up to 10,000 iterations each).

B. Testbed

HeatSweeper runs on a smartphone equipped with a quad-core processor at 2.5GHz and 3GB of memory. For the LAN case, it offloads tasks to a laptop equipped with a quad-core at 2.40GhZ and 8 GB of memory connected to the mobile via a 52 Mbps Wi-Fi. On the WAN scenario, the phone uses as surrogates a cluster of eight quad-core VMs on an OpenNebula cloud. Physical nodes are equipped with six-core at 2.67 GHz processors and 24 GB of memory interconnected by a Gigabit Ethernet network. The connection between the mobile device and the surrogates has a 85.5 ms RTT.

Table III contains energy and time measurements observed when benchmarking the testbed components. The base consumption of the mobile is 0.08 W; the additional cost of turning on the display depends on the screen brightness (from 0.3 to 1 W). Using the processor increases power consumption by 1.5 W when the mobile computes at full-capacity. When the display is off, the processor governor prioritizes battery lifetime over phone responsiveness and reduces the CPU frequency to a 5%. Despite this mechanism increases power consumption only by 0.1 W, computations last longer and the overall energy spent grows.

C. Performance evaluation

1) Low-resolution: HeatSweeper takes 71 s to run and consumes 135.52 J when the screen is on; 1,631 s and 251.72 J when it is off. We select the scenario where the screen stays on as the representative for the phone since it is better performing and less consuming that switching it off.

Figures 1(a) and 1(b) depict the relation between the amount of surrogates resources and the application timespan and energy consumption respectively. The isolated points show the obtained values for the mobile submitting tasks to the laptop; the continuous lines, when offloading to one, two, four and eight cloud instances; and the cross and the dotted line, the optimal values according to Table III values.

The best performing testbed for the low-resolution scenario is using the laptop as a surrogate. If the screen is kept on during the execution, the application achieves a speed up 32 times faster than the phone case (1,632 ms) and reduces the energy consumption to a 0.5% of the original (0.74 J).

Cloud scenarios behave better than using an isolated phone, but the execution time grows along with the number of surrogate nodes. The high latency on the network slows down the exchange of messages to notify data creations across different surrogates. When a data value is generated, all the tasks within the creating node can already access it while other surrogates need the data directory to notify them the existence and sources for that piece of data. Therefore, the best performing case is the one with a single surrogate since only the task description messages and the initial data transfers are affected by the high network latency.

2) High-resolution: Solving the high-resolution problem takes 99,641 seconds (more than 27 hours) on the phone with the screen on and the phone needs to keep plugged to an energy source. It is an example of the large set of applications whose executions are not viable in current mobile devices; however, COMPSs-Mobile provides them with an extra computing power that enables its execution by reducing the execution time and the energy consumption.

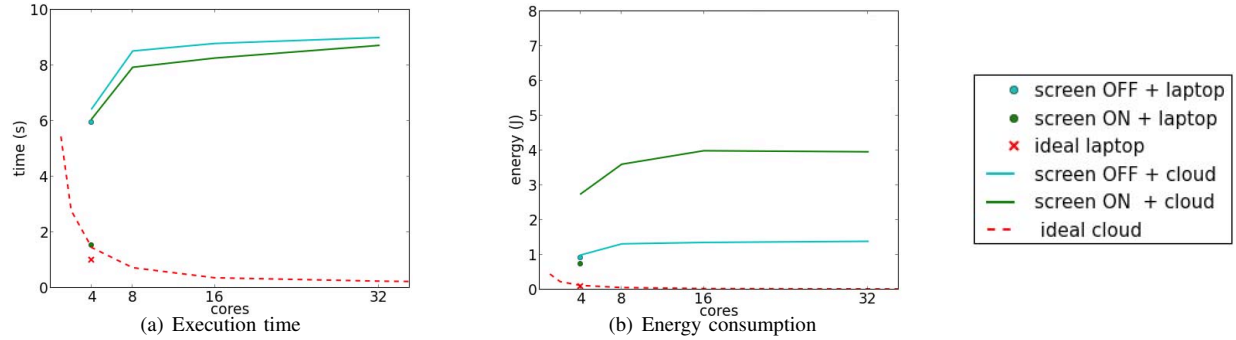
Again, figures 2(a) and 2(b) show the execution time and energy consumption according to the surrogate platform as done for the low-resolution case.

Since the Core Element execution time and the network latency are lower when the runtime uses a laptop as a surrogate than to a cloud platform when only 4 cores are available, the first behaves much better than the latter. Offloading to the laptop, it takes 1,368 seconds to solve the problem, achieving a 72.83x speed-up compared to running the application on the phone. This severe reduction on the timespan has a significant impact on the energy consumption that enables the application execution on the mobile device: 621.63 J when brightness is at 0%. Switching off the screen has a small impact on the application performance 1,401 s and gets a better energy consumption 216 J.

On the cloud scenario, when using only four cores, the execution time is significantly higher; and, therefore, the energy consumption too. In the respective best cases, the application lasts 2,318 s (42.99x), and the consumption is 363 J. However, the strong point of the cloud is the amount of resource available for the runtime to offload tasks. When the resource pool is increased up to 32 cores and the display is on, the application execution time is reduced to 320 seconds, and it consumes 146 J. This is 310 times faster than the isolated phone scenario and 4.26 times faster than

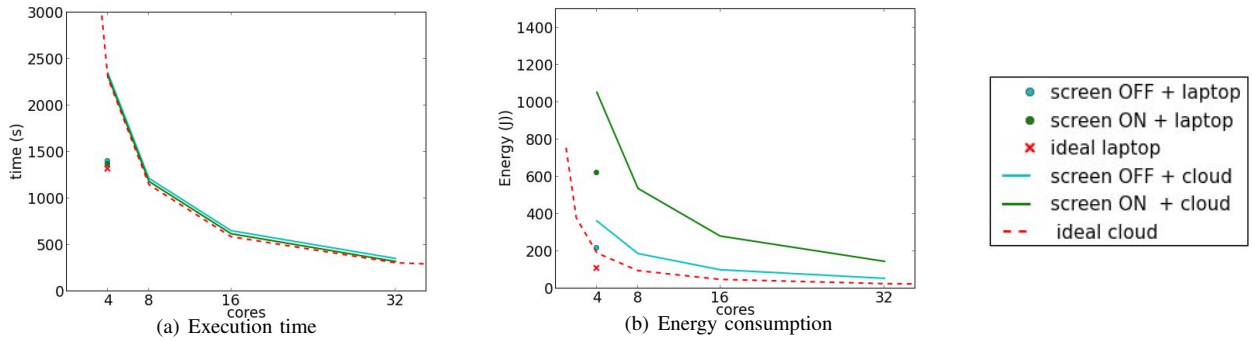
	Computing Capabilities			Core Element Analysis					
	Idle	Computing		50 iters. sim.		10k iters. sim.		Merge	
	Power (W)	MIPS	Power (W)	Time (ms)	Energy (J)	Time (ms)	Energy (J)	Time (ms)	Energy (J)
Mobile - off	0.08	15.92	0.20	35,549	6.72	6,794,135	1,350	negligible	negligible
Mobile - on 0%	0.37	376.96	1.87	1,483	2.85	288,667	561.61	negligible	negligible
Mobile - on 100%	1.07	376.30	2.55	1,468	3.95	288,816	797.03	negligible	negligible
Laptop	-	2,369.232	-	38	-	6,072	-	negligible	-
Cloud	-	1,326.08	-	57	-	27,979	-	negligible	-

Table III
RELATION BETWEEN EACH COMPUTING CONFIGURATION (MOBILE WITH SCREEN OFF, MOBILE WITH THE SCREEN ON AT 0% BRIGHTNESS AND 100%, LAPTOP OR CLOUD VM) WITH ITS COMPUTING CAPABILITIES AND THE ANALYSIS OF EACH CORE ELEMENT EXECUTION



(a) Execution time (b) Energy consumption

Figure 1. Low-resolution scenario results according to the surrogate platform



(a) Execution time (b) Energy consumption

Figure 2. High-resolution scenario results according to the surrogate platform

offloading tasks to a laptop. Switching off the screen allows the runtime to obtain a lower energy consumption 54.61 J.

IV. CONCLUSIONS AND FUTURE WORK

This paper presents the features of COMPSs-Mobile, a framework that automatically parallelizes MCC applications shorting their execution time while reducing the energy consumption as shown in Section III. To the best of our knowledge, it is the first framework to bring together Mobile Cloud Computing and automatic parallelization.

Organizing the infrastructure as a peer-to-peer network to distribute the data directory management would solve the observed latency issues and increase the independence of the workers during network breakdowns. Both ends of MCC have room for improvement: mobiles are equipped with GPUs and clouds allow dynamic resource provisioning; enhancing their exploitation would improve the performance.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Goverment (contracts TIN2012-34557, TIN2015-65316-P and BES-2013-067167), by Generalitat de Catalunya (contract 2014-SGR-1051) and by the European Commission (ASCETiC project, FP7-ICT-2013.1.2 contract 610874).

REFERENCES

[1] F. Lordan *et al.*, "Servicess: An interoperable programming framework for the cloud," *Journal of Grid Computing*, vol. 12, no. 1, pp. 67–91, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10723-013-9272-5

[2] "Java programming assistant (javassist)," http://www.javassist.org.