mCluster: A Software Framework for Portable Device-based Volunteer Computing

Dimitris Theodoropoulos, Grigorios Chrysos, Iosif Koidis, George Charitopoulos,
Emmanouil Pissadakis, Antonis Varikos, Dionisios Pnevmatikatos
Microprocessor and Hardware Laboratory, Department of Electronic and Computer Engineering,
Technical University of Crete Chania, Greece
Email: {dtheodoropoulos, chrysos, koidis, gcharitopoulos, pnevmati}@mhl.tuc.gr {epissadakis,avarikos}@isc.tuc.gr

Georgios Smaragdos
Christos Strydis
Dept. of Neuroscience
Erasmus Medical Center
Rotterdam, The Netherlands
Email: {c.strydis,g.smaragdos}@erasmusmc.nl

Nikos Zervos
Algosystems S.A.
Athens, Greece
Email: nzervos@algosystems.gr

***Abstract*—Recent market forecasts predict that the portable computing trend will vastly spread, as by 2020 there will be more than 3 billion LTE device users worldwide. Motivated by this fact, many companies and research institutes have already launched research projects that utilize portable devices, voluntarily provided by users, to perform the required computations. Many such projects employ Berkeley's BOINC middleware, since it can support a large variety of stationary and mobile devices. However, currently available BOINC high-level APIs, either do not support portable devices or lack advanced processing capabilities (such as inter-node task dependencies) and/or easiness of use. To resolve these issues, we propose the mCluster software framework for application execution powered by the BOINC middleware on portable devices. mCluster adopts a task-based programming model that requires simple, pragma-based annotations of the application software, in order to dynamically resolve task dependencies. To evaluate our framework, we have have mapped a scientific application from the neuroscience domain on an small-scaled network of portable devices. mCluster significantly reduces the required programming effort and complexity to efficiently map BOINC-powered applications with task dependencies on portable devices compared to previous approaches.**

***Keywords*-Task-based programming, internet of things, volunteer computing**

I. Introduction

Volunteer computing is a form of distributed computing. It pertains to computer or mobile device owners (volunteers), who donate their processing power and / or storage resources to scientific projects that need to execute computationally intensive software routines (tasks). Volunteer computing was first applied by the "Great Internet Mersenne Prime Search" project[1] in 1996. However, it was mainly established in the early 2000's, when the University of California, Berkeley released the BOINC middleware [1] to the public. To date, BOINC is still the most widely used middleware that scientists and companies utilize to launch research projects on distributed networks of desktop computers and mobile devices (smartphones and tablets)[2].

Although many companies and universities have acknowledged the power of volunteer computing, and used BOINC to launch a variety of research projects, it is still applied with limitations mainly due to the following reasons:

- Currently, BOINC does not support inter-node task dependencies, hence limiting the nature of projects and applications that are applicable for volunteer computing. Task dependencies are only supported at intra-node level, where developers manually divide an application into sets of dependent tasks, and assign each set to a virtual machine hosted on a client node [2].
- Setting up and launching research projects, requires (a) skills and experience in computer-science aspects, and (b) significant effort and time (the very same experience that this team had while carrying out this work). As a result, scientists from other research domains (e.g. astrophysics, seismology, biology, biomedical, financial) find very hard starting such projects, which also impacts volunteer computing.

To alleviate these issues, in this paper we propose the mCluster software framework for portable device-based volunteer computing powered by the BOINC middleware. With our work we strive to facilitate the deployment of applications on distributed networks consisting of portable devices, and minimize the enormous effort and knowledge currently required to successfully launch research projects powered by BOINC. Towards this, mCluster adopts a task-based programming model that requires simple pragma-based annotations of the application software, in order to dynamically resolve task dependencies. In other words, scientists and developers need only to insert certain key-words in their original software to describe task dependencies, which mCluster automatically handles at runtime. Overall,

[1] http://www.mersenne.org/

[2] https://boinc.berkeley.edu/

the paper contributions are the following:

- We propose the mCluster software framework for application execution powered by BOINC on portable devices. Our framework extends the BOINC infrastructure with a task-based programming model that requires simple, pragma-based task annotations.
- The mCluster implementation is bundled with the BOINC middleware; it resolves all *inter-node task dependencies at runtime*, a feature that -to the best of our knowledge- was not previously supported.
- We provide a first mCluster-compatible implementation of a biologically accurate brain simulation (Inferior Olive), and execute it on an experimental network of smartphones and tablets.

The rest of the paper is organized as follows: Section II provides references to related work and various projects based on volunteer computing. Section III describes the mCluster front-end, while Section IV its integration with the BOINC middleware. Section V presents the Inferior-Olive application implementation. Section VI describes our experimental setup and evaluates the mCluster framework. Section VII discusses our future work and concludes the paper.

II. Related Work

Distributed computing has already been utilized in various projects, such as the BOINC middleware [1] and Wisconsin Condor [3], however they target desktop computers as processing nodes. A few years ago, BOINC started supporting mobile devices via client applications that users can download from the Google Play app store[3] to perform volunteering processing for specified scientific problems.

A significant amount of work can be found on adding virtualization to BOINC [4], [5], [2], [6]. Virtualization offers many benefits for the host machine, such as application porting, security and system-level checkpointing, at the expense of increased resource utilization and slightly degraded performance. Moreover, [7] presents DC-API, a simple API specifically targeted for desktop grid systems that overcomes the lack of Java-application support in BOINC-powered desktop systems. Although, DC-API hides many low-level details of the BOINC infrastructure similarly to our approach, it only supports desktop computers as clients.

Major mobile-market companies (e.g. HTC, Samsung, IBM, Sony) offer BOINC-powered applications that users can download from app stores to actively contribute to specified scientific problems (cure for cancer, Alzheimer's disease and AIDS)[4, 5, 6, 7]. However, to date, volunteer computing has been applied with limitations, since launching portable devices-based projects requires significant effort. Moreover, efforts on supporting task dependencies are limited at host-level [2]; a virtual machine running on a host, can resolve dependencies only among tasks that are executed locally.

[3]https://play.google.com/store/apps/details?id=edu.berkeley.boinc
[4]https://play.google.com/store/apps/details?id=com.htc.ptg
[5]https://play.google.com/store/apps/details?id=at.samsung.powersleep
[6]https://www-03.ibm.com/press/us/en/pressrelease/45594.wss
[7]https://play.google.com/store/apps/details?id=com.sonymobile.androidapp.gridcomputing

```
//vectors addition task
void v_add(int *vA,int *vB, int *vC) {
   //vectors addition code
   ..
}
//vectors subtraction task
void v_sub(int *vA,int *vB, int *vC) {
   //vectors subtraction code
   ..
}
//vectors multiplication task
void v_mult(int *vA,int *vB, int *vC) {
   //vectors multiplication code
   ..
}
int main (){
   //vectors declaration
   ..
   //vectors initialization
   ..
#pragma mcluster input (int &v1[0],VECTOR_DIM,int &v2[0],
VECTOR_DIM) output (int &v3[0],VECTOR_DIM)
   v_add(&v1[0],&v2[0],&v3[0]);
#pragma mcluster input (int &v3[0],VECTOR_DIM,int &v5[0],
VECTOR_DIM) output (int &v4[0],VECTOR_DIM)
   v_sub(&v3[0],&v5[0],&v4[0]);
#pragma mcluster input (int &v4[0],VECTOR_DIM,int &v6[0],
VECTOR_DIM) output (int &v3[0],VECTOR_DIM)
   v_mult(&v4[0],&v6[0],&v3[0]);
   return 0;
}
```

Figure 1: Application-task annotation with the mCluster pragma keywords.

III. The mCluster Framework

Programming Model: mCluster adopts a task-based programming model that requires developers to simply pragma-annotate the application code to dynamically resolve task dependencies when executed on a BOINC-powered environment. Figure 1 illustrates an example of a simple code that processes 1-d vectors. There are three tasks, namely "v_add", "v_sub" and "v_mult".

To define application tasks, developers need to insert above each task the *#pragma mcluster* string, followed by the *input* keyword. This defines all task inputs, where developers insert within parenthesis each input type, starting address and size, all separated by commas. Similarly, developers can define all task outputs with the keyword *output*, followed within parenthesis by each output type, starting address and size.

Our framework supports all input/output types, i.e. char, short, int, float, double and structs. Furthermore, it also provides manual task synchronization with the *mClusterSync()* function (discussed in Section V). The latter essentially works as a sync barrier that suspends further application execution, until all currently dispatched tasks have returned their outputs. It should be noted also that the mCluster API

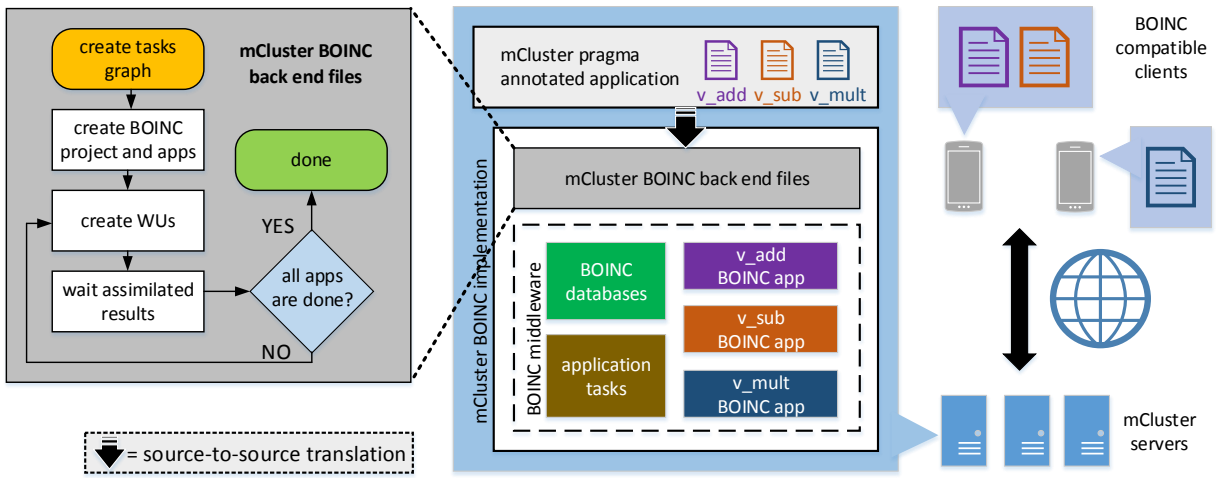

Figure 2: The mCluster BOINC implementation, where the original pragma-annotated code is source-to-source translated to generate all BOINC-oriented back-end files.

imposes certain limitations, such as nested task annotation. In other words, developers are not allowed to annotate a task within an already annotated task.

mCluster Front End: The mCluster front end is responsible for translating the annotated code to source files compatible with BOINC-specific API calls for task generation / management. As observed, the original code includes Write-After-Write (WAW) hazards between *v_add* and *v_mult* tasks, and Write-After-Read (WAR) hazards between *v_sub* and *v_mult* tasks. We have implemented an in-house source-to-source translator that works in two phases: First, all WAR and WAW hazards are resolved using the task-output renaming technique described above. At the second phase, the generated code (a) keeps all task metadata (i.e. unique id, input/output parameters and their sizes) in private structures, and (b) replaces all #pragma-based annotations with middleware-specific API calls for task generation/management. We should note though that task-output renaming introduces a storage overhead, since it increases the total number of arguments used during the application execution. In the next section, we describe our mCluster implementation that utilizes the BOINC middleware for tasks dispatching to portable devices.

IV. mCluster Implementation

Figure 2 illustrates the mCluster-BOINC implementation. Considering as example application the source code depicted in Figure 1, the workflow proceeds as follows:

Remove data hazards and generate BOINC-compatible back end files: As described in Section III, the original annotated code is source-to-source translated, in order to (a) remove any WAW and WAR hazards among tasks, and (b) generate the required BOINC-compatible back end files for task generation / management. In order to tackle inter-node tasks dependencies and, at the same time, follow the strict project/application BOINC structure, we associate each task with a BOINC application [1].

Thus, the mCluster back end BOINC files (a) parse the original code to keep all task metadata (i.e. unique id, inputs/outputs parameters and their sizes) in private structures, (b) run the required BOINC scripts to generate a new full BOINC project (including its internal databases and other structures), and (c) automatically generate a unique BOINC application associated to each annotated task in the original code, excluding its Work Generator (WG) [1], thus suspending execution. As soon as all applications are created, every client device that is registered to the BOINC project will receive all task executables and be ready to accept task instances for execution.

Start ready BOINC applications: mCluster traverses the generated task graph and starts all BOINC applications with ready inputs by automatically generating its WG. Note that issues like fault tolerance and task starvation are handled by BOINC, as each task is issued to more than one portable device, and if a task gets timed out, it will be assigned to another device.

Wait assimilated results: mCluster halts further execution of the back end files, until at least an application reports new results in its corresponding BOINC assimilator [1]. Based on the updated ready data, mCluster traverses further into the task graph to identify which BOINC applications have their inputs now ready, in order to proceed to step 2. mCluster also deletes all BOINC applications whose outputs are no longer needed, keeping at minimum resource utilization on the server side. If all BOINC applications are done, it exits.

It should be noted that the number of concurrently ”active” BOINC applications depends on the server resources capacity (e.g. CPU power, available memory, etc). For this reason, mCluster has the option to constrain them up to a predefined limit, in order not to overload the server machine. Such a case can occur for example within for-loops iterated hundreds or thousand times that call independent task(s).

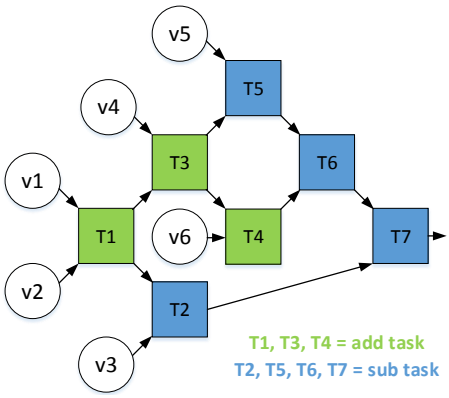

Figure 3: The synthetic micro-benchmark used for our experiments.

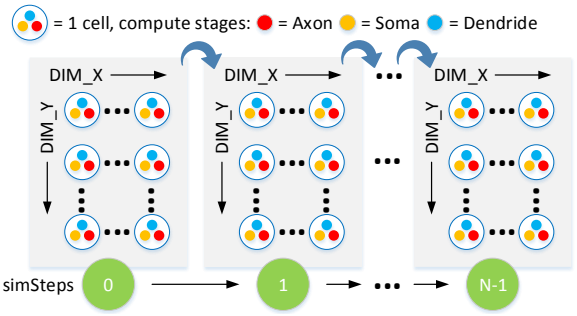

Figure 4: Illustration of the Inferior-Olive application execution procedure.

V. Case-Study Applications

To verify the correctness and evaluate the feasibility of our approach, we have performed tests with a synthetic micro-benchmark and a real, computationally intensive application from the biomedical domain [8].

Vector-processing micro-benchmark: Figure 3 illustrates our synthetic micro-benchmark that imposes various dependencies among tasks. All task inputs and outputs are 1-d vectors. More specifically, v1 to v6 are input vectors to tasks, T1, T3 and T4 tasks perform vector addition, and T2, T5, T6 and T7 perform vector subtraction. Within the original micro-benchmark, we annotated each task with the mCluster keywords as described in Section III.

The Inferior-Olive application: The main goal of accurate brain simulation is the accelerated brain research, by the creation of more advance research platforms. There are two distinctive threads in experimentation using accurate computational neuron models: Either (a) real-time or close to real-time, high-performance modeling experiments, or (b) large-scale network experimentation, in much lower than real-time performance, where simulation speed is traded for experiments in network sizes respective to the actual biological network (the human cerebellum consists of 50 billion cells).

mCluster can effectively support such demanding research domains, by distributing neuron-cell computations in the virtual cluster of portable devices (many of them based on commercial GPGPUs). Even though there would be a significant data-transfer penalty due to the wireless communication involved, data-transfer demands in the case of accurate-model simulations are low compared to the computational demands. By taking into account the fact that almost 200 million devices are sold to the public every quarter year, an mCluster implementation of this application shall give the ability to perform brain simulations (a) with biologically accurate neuron models, (b) in the realistic sizes of several millions of neurons, and (c) in a small fraction of the cost compared to investing on a CPU or GPGPU cluster.

In its current version, the application models, in time steps of 0.05 msec each, the state of a 2-d neurons grid (cells), all interconnected with their 8 neighbours. Each cell consists of three main computational stages (Dendride, Soma and Axon). Figure 4 illustrates this procedure for N simulation steps (simSteps). At every simStep, a task calculates a cell's state within the 2-d cell grid with dimensions DIM_X x DIM_Y, while all tasks can be executed concurrently.

VI. Experimental Results and Evaluation

Experimental System: To verify our BOINC-compatible implementation we deployed an experimental system consisting of one BOINC server machine and six client devices. The BOINC server is hosted on a 64-bit Ubuntu 12.04 LTS virtual machine (VM) with a single-core CPU at 2.1 GHz, 4GB RAM and 20GB storage. The client machines are two Samsung Galaxy S4 smartphones and four Asus Nexus 7 (2013) tablets under different Android versions.

Micro-benchmark evaluation: We annotated the task graph source code (Figure 3) with the mCluster #pragma keywords. mCluster automatically source-to-source translated the original code, generating the corresponding BOINC project and all ready applications. As can be observed, only two tasks can be executed concurrently, hence up to two client devices can work simultaneously. Table I shows the overall micro-benchmark execution times using 1-d vectors with 1,000 and 10,000 elements, respectively, divided into the following parts: (a) input-data uploading to client devices, (b) vector processing, (c) output-result transmission to server, and (d) work on server for task management. In this case, mCluster introduces a small time overhead on the server work of approximately 3.2% and 3.6% when using 1,000- and 10,000-element vectors, respectively. Finally, the actual execution time attributes to 32% and 48% of the overall elapsed time.

Inferior-Olive application evaluation: Figure 5 provides the Inferior-Olive annotated pseudocode. Within each iteration of the simSteps-bounded for-loop, the code traverses the 2-d grid and calculates concurrently each neuron's computational state, by calling the *cellTask* task. For each neuron, *cellTask* requires the following arguments: an input struct (60 bytes) that provides the current state of its neighbours, and three output floats (12 bytes in total)

```
//cell task
void cellTask (cellCompParams *cellCompParams,
float *newVDent, float *newVSoma, float *newVAxon) {
   //dendrite calculations function
   CompDend (cellCompParams,newVDent);
   //soma calculations function
   CompSoma (cellCompParams,newVSoma);
   //axon calculations function
   CompAxon (cellCompParams,newVAxon);
}
int main () {
   //variables declaration
   ..
   //variables initialization
   ..
   for (i=0;i<simSteps;i++){
      //Compute one sim step for all cells
      for (j=0;j<CELL_NETWORK_DIM_X;j++) {
         for (k=0;k<CELL_NETWORK_DIM_Y;k++) {
            idx = i*CELL_NETWORK_DIM_X*CELL_NETWORK_DIM_Y
            + (j*CELL_NETWORK_DIM_X + k);
            //get neighbors' voltage influence
            ..
#pragma mcluster input (struct &cellCompParamsPtr[idx],1)
output (float &vDent[idx],1,float &vSoma[idx],1,float &vAxon[idx],1)
            cellTask(&cellCompParamsPtr[idx],&vDent[idx],
            &vSoma[idx],&vAxon[idx];
         }
      }
      //wait until all cells for this task
      //are calculated
      mClusterSync();
   }
   return 0;
}
```

Figure 5: Annotation of the Inferior-Olive application with the mCluster pragma keywords.

Table I: Execution time of the synthetic micro-benchmark.

task processing vectors with 1,000 elements				
upload	**download**	**processing**	**server work**	**TOTAL**
1 sec	1 sec	1 sec	0.1 sec	3.1 sec
task processing vectors with 10,000 elements				
upload	**download**	**processing**	**server work**	**TOTAL**
1 sec	1 sec	2 sec	0.15 sec	4.15 sec

that designate its next dendrite, soma and axon states. Consequently, the original code requires only the #pragma annotations of the *cellTask* task shown in Figure 5.

To suspend further application execution until all neuron states have been calculated for a particular simulation step, we insert the *mClusterSync()* function. The latter implements a barrier that checks if all currently dispatched tasks have finished their execution. mCluster pauses further task generation until all active ones return their results, thus ensuring a correct application behavior.

To evaluate the application on a real environment, we have executed it on the experimental setup described above. Figure 6 provides the average simStep execution time for 96, 192, 288 and 384 cell grid sizes, divided into the following segments: server work, download task data, task execution, and upload task results.

The server work segment represents an average 13% overhead introduced by (a) the mCluster framework that creates / deletes the required BOINC applications associated

Table II: Platform comparison table

platform	max. cell #	simStep exec. time	cost
FPGA Virtex-7 [9]	$\leq$ 14,400	6.4 msec	$3,500
Maxeler Maia [10]	$\leq$ 7,840	0.04 msec	>$25,000
mCluster (10k nodes)	1,000,000*	[7.26 sec – 2,578.80 hr]	$0**

* No real upper limit exists for mCluster; human-sized Inferior Olive selected here for comparison purposes.
** A small cost may be applied, in case mCluster is hosted to a dedicated server machine instead of a VM.

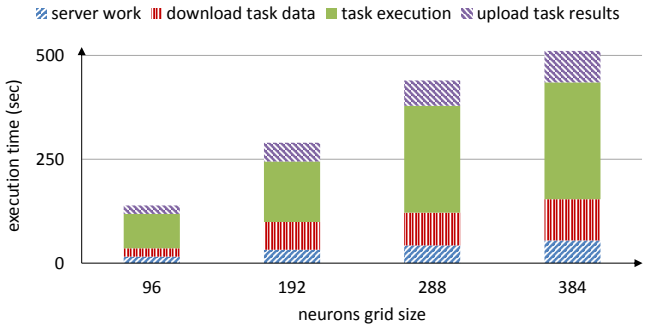

Figure 6: Average simStep execution time of the Inferior-Olive application on all client devices.

with the original tasks, and (b) run the BOINC middleware. Compared to the previous micro-benchmark, the source-to-source translated application code in this case is much larger, hence writing all output files creates an increased overhead. On average, a 67% of the overall time is spent on the tasks execution, thus indicating the benefit and need of having more processing resources available, in order to reduce the execution time. Finally, approximately 20% of the time is spent on exchanging data (upload and download) between the clients and the server.

Discussion: Taking our analysis one step further, we estimate the processing time needed to model real biological brains. According to our experiments, an Inferior-Olive task requires on average 1.46 sec, 0.93 sec and 4.87 sec for data transfers, server work (create and manage the corresponding BOINC application) and actual execution on a client device, respectively. Figure 7 shows the projected processing time for evaluating a single simStep when simulating the aforementioned different brain complexities (mouse, cat and human).

In our analysis, we assume that data transfers and task processing are equally distributed to all available devices. As expected, the currently implemented system is bounded by the capabilities of server machine which hosts the mCluster framework and orchestrates the overall application execution. However, this limitation can be easily addressed by (a) utilizing more powerful server(s), and (b) employing additional servers to host the mCluster framework. The latter method is feasible, since the BOINC middleware server architecture is *scalable* across different machines [1].

Comparison to other platforms: Volunteer computing-based processing platforms, work on a "best-effort" approach. Processing nodes are unreliable in terms of availability and connectivity, thus execute tasks only when

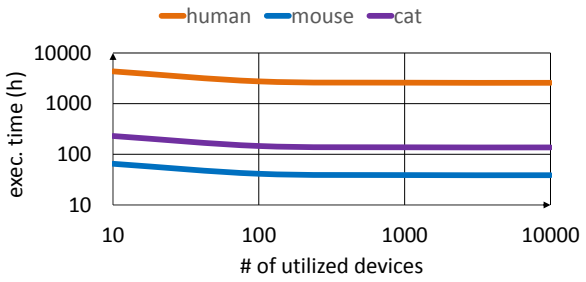

Figure 7: Projected processing time of a single simStep when simulating different brain types using up to 10,000 portable devices; execution time is lower bounded by the server work.

certain parameters and circumstances apply, thus can not compete against high-performance (and dedicated) computing platforms.

Table II lists two works that have implemented the Inferior-Olive application in a Xilinx Virtex-7 FPGA platform [9] and a Maxeler Maia Data-Flow-Engine (DFE) platform [10]. The table reveals that both implementations can execute a single simStep at least an order of magnitude faster compared to our system (we include the server-work and data-transfer overheads). However, [9] and [10] can process up to 14,400 and 7,840 neurons respectively due to their limited resources. In contrast, an mCluster-powered system is only limited by the available portable devices in terms of processing resources.

In addition, commercial platforms introduce significant acquisition and maintenance costs. In contrast, mCluster-powered systems are solidly based on portable devices owned by external users. Furthermore, as described in the beginning of this section, mCluster projects can be deployed even on VMs hosted on already available server machines, hence not imposing extra setup and maintenance costs.

Integration with related work: As described in Section II, the setup overhead of currently available middleware and their lack of supporting inter-node task dependencies, still constrain the applicability of volunteer computing. mCluster targets to bridge this gap, and provide a simple and powerful programming environment, that will allow scientists from various research domains to discover and utilize the power of volunteer computing to solve computationally intensive problems.

The current mCuster implementation utilizes the original BOINC middleware, however with certain modifications, it may be bundled with BOINC's extensions, such as the VBOINC and vboxwrapper. Such an approach would combine the benefits of both works, like inter / intra-node task dependencies support, easy application porting to different hosting platforms, increased security and system-level checkpointing. A drawback, though, is that virtualization layers introduce performance overheads, and allocate a significant amount of storage and memory resources, hence cannot be executed on low-end hosting platforms (such as smartphones and tablets), which are the "processing heart" of volunteer computing.

VII. Conclusions and Future Work

In this paper we presented the mCluster framework that offers a simple -yet powerful- framework for BOINC-powered application execution on clusters of portable devices. As future work, we would like to focus on (a) integrate advanced scheduling policies for improved tasks scheduling on such unreliable (in terms of connectivity, and availability) distributed processing networks, (b) support the virtualized extensions of BOINC in order to improve application portability, security and support system-level checkpointing, and (c) make available to the public our Inferior-Olive application project, so users worldwide can actively contribute to biologically accurate brain simulation.

Acknowledgment

This work has been supported by the 2007 - 2013 National Strategic Reference Framework (NSRF) of the General Secretariat for Investments and Development / Greek Ministry of Economy and Finance, under the thematic session "Competitiveness and Entrepreneurship and Regional Operational Programme of the Regions in transitional support" (project #1592).

References

[1] David P. Anderson, et. al., "High-Performance Task Distribution for Volunteer Computing," in *International Conference on e-Science and Grid Computing*, 2005, pp. 196–203.

[2] Gary A. McGilvary, et. al., "V-BOINC: The Virtualization of BOINC," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2013, pp. 285–293.

[3] Douglas Thain, et. al., "Distributed Computing in Practice: The Condor Experience," in *Concurrency and Computation: Practice and Experience*, vol. 17, February 2005, pp. 323–356.

[4] Diogo Ferreira, et. al., "ibboincexec: A generic virtualization approach for the BOINC middleware," in *International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, May 2011, p. 1903–1908.

[5] Daniel Lombrana Gonzalez, et. al., "Interpreted applications within BOINC infrastructure," in *Iberian Grid Infrastructure Conference Proceedings*, May 2008, p. 261–272.

[6] Ben Segal, et. al., "LHC cloud computing with CernVM," in *International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, February 2010.

[7] Attila Csaba Marosi, et. al., "Enabling Java applications for BOINC with DC-API," in *Distributed and Parallel Systems, In Focus: Desktop Grid Computing ISBN: 978-0-387-79447-1, Springer Science+Business Media, LLC*, 2008, pp. 3–12.

[8] Jornt R. De Gruijl, et. al., "Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting," in *PLoS Computational Biology*, vol. 8, December 2012.

[9] Georgios Smaragdos, et. al., "FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons," in *22nd ACM/SIGDA Int. Symposium on FPGAs (FPGA)*, February 2014, pp. 89–98.

[10] ——, "Real-Time Olivary Neuron Simulations on Dataflow Computing Machines," in *Int'l Supercomputing Conference (ISC)*, June 2014, pp. 487–497.