J. Parallel Distrib. Comput. 122 (2018) 81–94

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Distributed computing by leveraging and rewarding idling user resources from P2P networks

Nunziato Cassavia [a,b], Sergio Flesca [b], Michele Ianni [b], Elio Masciari [a,*], Chiara Pulice [c]

[a] ICAR-CNR, Rende, Italy
[b] DIMES-UNICAL, Rende, Italy
[c] Dartmouth College, Hanover, NH, USA

ARTICLE INFO

Article history:
Received 15 September 2017
Received in revised form 15 March 2018
Accepted 19 July 2018
Available online 7 August 2018

ABSTRACT

Currently, many emerging computer science applications call for collaborative solutions to complex projects that require huge amounts of computing resources to be completed, e.g., physical science simulation, big data analysis. Many approaches have been proposed for high performance computing designing a task partitioning strategy able to assign pieces of execution to the appropriate workers in order to parallelize task execution. In this paper, we describe the Coremuniti^{TM} system, our peer to peer solution for solving complex works by using the idling computational resources of users connected to our network. More in detail, we designed a framework that allows users to share their CPU and memory in a secure and efficient way. By doing this, users help each other by asking the network computational resources when they face high computing demanding tasks. In this respect, as users provide their computational power without providing specific human skill, our approach can be considered as a hybrid crowdsourcing. Differently from many proposals available for volunteer computing, users providing their resources are rewarded with tangible credits, i.e., they can redeem their credits by asking for computational power to solve their own task and/or by exchanging them for money. We conducted a comprehensive experimental assessment in an interesting scenario as 3D rendering, which allowed us to validate the scalability and effectiveness of our solution and its profitability for end-users. As we do not require to power additional resources for solving tasks (we better exploit unused resources *already* powered instead), we hypothesize a remarkable side effect at steady state: energy consumption reduction compared with traditional server farms or cloud based executions.

1. Introduction

Computer science applications are deeply changing due to the cost of hardware components that is continuously decreasing [1] and their computing performances that are reaching unprecedented levels. Based on the availability of a plethora of powerful (even portable) computing devices, new high performance computing tasks have been designed and new data management paradigms emerged (e.g., the well known Big Data metaphor [2–5]). Moreover, the advances in computer science allow many problems to be solved in a quite effective way both in terms of computational time and resource usage. However, several problems still require an amount of computational resources that goes far beyond the power of a single device or a single user. In order to solve these kinds of problems, a new computing paradigm has born: *crowdsourcing*. This new paradigm is based on the idea of gathering the resources needed to complete a task from the crowd in order to parallelize its execution.

Many attempts have been made to properly define the key features of crowdsourcing systems; however, the answer to this apparently trivial question is not straightforward, since there exist many different crowdsourcing systems based on different models and assumptions. If we try to find the common features shared among all the successful crowdsourcing systems (e.g., Wikipedia, Yahoo! Answers, Amazon Mechanical Turk) we can clearly realize that they rely on some assumptions:

- They should be able to involve project contributors
- Each contributor should solve a specific task
- It is mandatory to effectively evaluate single contributions
- They should properly react to possible misconducts

As we can see from the examples above many systems are based on an explicit collaboration of many different people sharing

* Corresponding author.
E-mail addresses: nunziato.cassavia@icar.cnr.it (N. Cassavia), flesca@dimes.unical.it (S. Flesca), mianni@dimes.unical.it (M. Ianni), elio.masciari@icar.cnr.it (E. Masciari), chiara.pulice@dartmouth.edu (C. Pulice).

https://doi.org/10.1016/j.jpdc.2018.07.017
0743-7315/

the will to build a long-lasting product that can be used by the whole community. Aside from this human-centric view, there exist a plethora of systems that leverage implicit co-operation among users (e.g. multiplayer games) or tools that are not devoted to the production of a tangible object (e.g., Amazon Mechanical Turk).

A well known open source framework that is widely used (mainly) for scientific purposes is BOINC (Berkeley Open Infrastructure for Network Computing) [6]. It allows volunteers to contribute to a wide variety of projects. Their contribution is rewarded by credits used to climb a leaderboard. In recent years a new category of collaborative approaches for cryptovalue mining is emerging, such as Bitcoin [7] implemented by blockchain technology. Users aiming at mining new Bitcoins contribute in solving a decoding task and are rewarded with a portion of the gathered money, proportional to the effort put in the mining task. The latter approach is gaining a lot of attention from the end users just because it allows them to earn a tangible reward.

In this paper we present our system named *Coremuniti*™ (that stands for *Community of Cores*) that is inspired by the collaborative model used in BOINC while implementing an ad hoc rewarding strategy similar to Bitcoin mining. As we do not require in principle any specific user skills but they can join the network simply providing their under used computational resources, our approach can be seen as a hybrid crowd as tasks can be solved by computer-based resources [8]. Our proposal received a grant from Calabria region that fully funded the early start-up costs. Moreover, EU commission awarded our project with a seal of excellence. We are currently in beta testing phase for commercial use.

More in detail, the novelty of our innovative project is the design of a peer to peer framework able to provide services at much lower prices compared to centralized center farms, by exploiting idle computational resources from the users joining the network.

Our software can be used in several application scenarios, e.g. computer simulation and advanced data analysis and it is well suited for vertical implementation of computing intensive tasks, representing a trans-disciplinary opportunity. More specifically, our approach can be a valid alternative to traditional solutions, such as buying or renting expensive dedicated servers. Furthermore, in many cases, even using powerful dedicated servers, the time needed to solve a problem is still too high, because the subtasks composing the problem are not parallelized at all. Our approach is based on a high performance Peer to Peer (P2P) network composed of computational resources shared by the users of the network itself. Each node of the network (i.e. users in the crowd) can set the amount of their resources to share. When a peer needs to execute a high resource consuming task, she can ask the necessary computational power to the network. The process can be executed in few clicks thus making our software quite user friendly. In order to assess the effectiveness of our solution, we analyzed the 3D rendering scenario that turns to be a severe test bench for our technology. We developed a specialized plugin named *Mozaiko*™ (We chose this name as our approach splits a complex task in several sub-tasks that will be re-assembled like mosaic tiles) that allows to render Blender 3D models in our distributed network. The rendering process typically engages user's computers for a considerable amount of time. Our experimental analysis shows that existing solutions are slower and more expensive than our proposal. Moreover, our system does not require to frequently purchase new hardware, since users continuously provide (up to date) computational power. Finally, the better re-use of already powered resources could induce a beneficial systemic effect by reducing the overall energy consumption for complex tasks execution. We motivate our conjecture in the future work discussion as we are not able at this stage to generalize our early results.

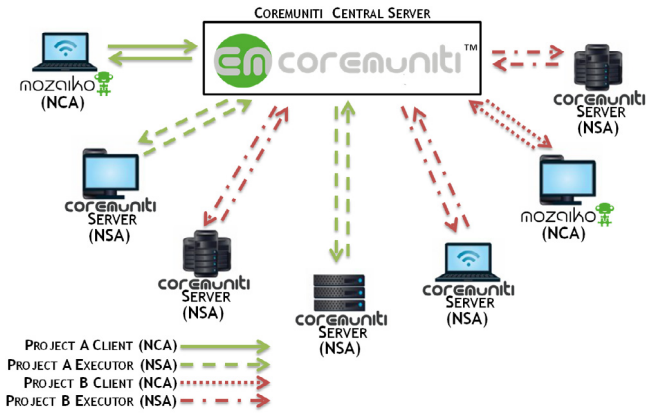

Fig. 1. Coremuniti system at work.

1.1. Coremuniti in a nutshell

P2P networks feature a common goal: the resources of many users and computers can be used in a collaborative way in order to significantly increase the computing power available for the users and parallelizing task executions. In "full" P2P networks, each computer communicates directly to each other thus allowing better bandwidth use. However, in many cases, there are some inherent drawbacks to P2P solutions and some functionality needs to be centralized. Those systems, that can be used both for data sharing [9] and distributed computation [10], are denoted as "hybrid" P2P. Coremuniti falls in the latter category and aims to build a P2P network where users can share their unexploited computing resources. In Fig. 1 we sketched a possible usage scenario for our platform when using Mozaiko plugin. However, we point out that our platform is general purpose, thus can be used for solving any complex problem that is parallelizable.

In order to join our network a user has to download the *Coremuniti Server* (platform independent) software. By running this software the user becomes a node server of our network, denoted in Fig. 1 as *NSA* (*Node Server Agent*). Our software does not interfere with other applications running on the computer and the user can easily set the amount of CPU that wants to share with the network, so that Coremuniti Server can be easily adapted to everyone's needs. On the opposite side, users who need additional computational power, in order to complete computing intensive tasks for example, can install the specific software that is denoted in Fig. 1 as *NCA* (*Node Client Agent*) (i.e., *Mozaiko* for our case study). To start a new task, they simply issue a request to the network in order to gather the required resources. The submission of a new task in the network will cost to the user a number of credits proportional to the complexity of the task itself (see 4). Since each node of the network can act both as a server and as a client, when submitting a new task two cases may occur:

1. the user has previously earned (a portion of) the required credits for running the task (e.g., because they acted as servers)
2. they bought the required credits

In order to guarantee a high level of service, the central server is responsible of performing the subtask assignment. More in detail, to fully take advantage from the capability of our network, we partition the (possibly huge) initial task in an adequate number of (much smaller) subtasks that can be quickly executed by the server peers. Moreover, we built an internal company network of 80 peers that can be used when the number of available public peers is not sufficient to guarantee proper execution of user tasks.

As soon as subtasks are completed, we check their correctness and we reward the participating peers. We will describe our model for subtask assignment that guarantees efficient execution for clients and gave to all peers (even if they have limited computational power) the possibility to be rewarded. Interesting enough, even users that are not going to ask for task execution have the chance to earn credits that can be redeemed by coins or gadgets. The latter feature makes Coremuniti a more convenient choice w.r.t. other collaborative systems such as cryptovalue mining (as shown in the experimental evaluation section). In the early stage of development of our system, we performed some preliminary analysis by asking 500 students at University of Calabria their interest in joining the network. After specifying our rewarding model all of them agreed to join the network and they are currently beta testing the software.

Our Contribution. To summarize, we make the following major contributions[1]:

- We design and implement a hybrid P2P infrastructure named Coremuniti that allows collaboration among users by sharing unexploited computational resources. In particular, by running our software, users can join the Coremuniti network and they can either provide or request computational resources;
- We define a robust model for task partitioning, assignment and rewarding to network users. More in detail, users that are available for task execution are assigned with a suitable set of (sub-)tasks. When the execution is completed, we reward users with credits that can be later redeemed for asking computational resources, coins or gifts;
- We discuss the 3D rendering case study to prove the effectiveness of our approach. We measured our performances against popular 3D rendering services. Moreover, we also compared our performances w.r.t. other systems that reward users for their effort in solving computational expensive tasks.

The rest of the paper is organized as follows. Section 2 reports on a comprehensive overview of existing crowdsourcing proposals. In Section 3 we present our system architecture. In Section 4, we describe our mathematical model for subtask assignment. Section 5 is devoted to the description of a challenging use case scenario for our framework. Section 6 discusses the outcome we get with our approach. Finally, in Section 7 we draw our conclusions and discuss future work.

2. Related work

The word Crowdsourcing was first introduced by Howe [12] in order to define the process of outsourcing some jobs to the crowd. It is used for a wide group of activities, as it allows companies to get substantial benefits by solving their problems in effective and efficient way. Indeed, crowd based solutions have been proposed for a wide range of applications as in image tagging [13,14], query optimization [15–17], data processing [18–21], sentiment analysis [22] to cite a few.

Each Crowdsourcing System (*CS*) faces several challenges, such as how to recruit users and combine their contributions, how to manage abuse and how to evaluate their skill [23]. Our system leverages the Web to tackle the above mentioned challenges. More in detail, among the CS listed in [24], our system belongs to the explicit systems category as our users are explicitly involved to solve a task. To this end, explicit CS systems working on the Web represent the focus of our discussion on related work we provide next.

To apply CS to a task, a crucial problem is to decompose the task in several parts and distribute them to the users, such that each user can contribute executing a single part. The contributions need then to be combined to solve the whole task. The problem of finding such parts and combining user contributions is usually task specific since it depends on the task structure (e.g., in Section 5 we show our solution for the rendering of an image). Distributing the parts to a crowd of users is, instead, quite general and many platforms have been developed to assist this process, such as Amazon's Mechanical Turk (AMT) [25], CrowdFlower, CloudCrowd, Witkey and so on. Such platforms support crowdsourced execution of subtasks, by requiring little or no domain expertise to the crowd, which may get paid from the requester on a per-task basis.

There are plenty of works addressing a wide variety of problems by leveraging the above mentioned frameworks [15,19,20,26–29]. There is also a broad area of research proposing their own solutions [14,30–36] which are more adequate for specific class of crowdsourcing applications.

In [34,35,37] the authors consider the task assignment problem for knowledge-intensive crowdsourcing applications, such as collaborative creation of Wikipedia articles. The spatial crowdsourcing problem is considered in [31,33], which requires workers to physically move towards some specific locations to finish tasks.

An important concern when assigning work to crowd workers is maintaining high-quality of the resulting output. In [30] the authors introduce a Quality-Aware Task Assignment System. The latter performs an online task assignment on top of a crowdsourcing platform (e.g., AMT), by estimating, through two widely used evaluation metrics, i.e., accuracy and f-score, the improvement in the quality of the answers if the subtask has been actually sent to a specific worker. An adaptive crowdsourcing framework, called iCrowd, has been proposed in [36]. This framework takes as input a set of subtasks, and assigns on-the-fly each subtask to the workers with the highest estimated accuracy among all online workers. The accuracy of the workers is computed by evaluating their performance on the completed tasks and is updated as they submit an answer to a task. In [32] the authors consider the problem of engaging the most diverse crowd, as diversity of opinions is essential to form a wise crowd and, consequently, guarantee the quality. A crowdsourcing data analytics system is described in [14]. Roughly speaking, the common goal of these solutions is to improve the task assignment with the intent of reducing the waiting time and increasing the output quality.

Differently from our proposal, the above mentioned works treat the crowd as a single processor available for solving the task. In other words, machines (i.e. CPUs) do not contribute to the answer. There are also some works which combine the machines and the crowd to improve the accuracy of the answer [38,39]. In our system, instead, the crowd is not directly involved in solving a task, as it has simply to set how much of its unused machine power is made available to the computing network.

Given the increasing availability of powerful phones, mobile cloud computing has been gaining lot of attention lately. The general idea is that computer intensive applications can be executed on low resource mobile devices by exploiting the computing and storage capabilities of an external infrastructure, known as *mobile cloud*. There are several ways to build a mobile cloud computing platform (see [40] for a recent survey). The approach that is closest in nature to ours has been proposed in [41], in which the author introduces a mobile peer-to-peer network consisting, at least partially, of smart phones. One of the main challenges faced by every mobile cloud computing framework and that limits the

[1] A preliminary abridged description of our system appeared in [11]. In this respect, we point out that: (1) the conference (short) paper contains a rough description of the system architecture without any detail of the actual system implementation and its distinguishing features; (2) The experimental discussion was not properly elaborated as in the journal paper; (3) No comparison with related work has been provided; (4) No case study has been discussed as in the journal paper.

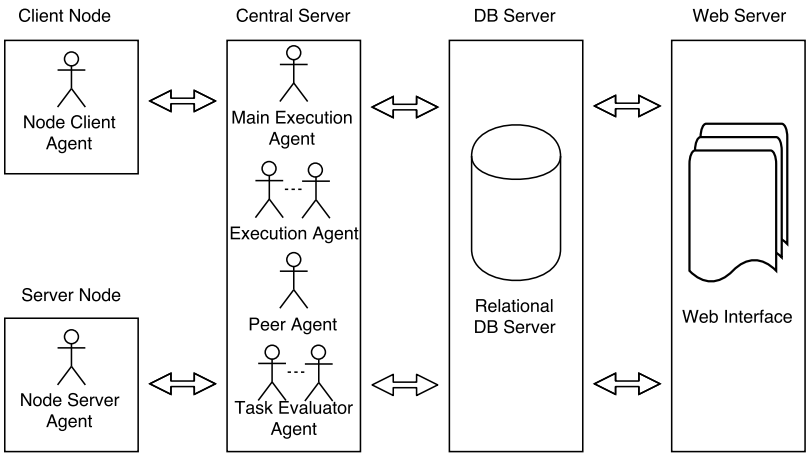

Fig. 2. Our system architecture.

benefits of using mobile devices is battery consumption. In order to overcome the energy-saving problem, several studies propose techniques to efficiently assign tasks to the resource providers and reduce the computation energy cost [42–44]. Another major issue concerns privacy and security of the resource being used. A common approach, which is also the one used in our framework, is to execute the tasks on a virtual machine. Sharing and transferring data in a secure way is another problem in distributed computing and it has been addressed in several works [45,46].

Other dynamic distributed computing environments have been proposed in many previous studies [47,48]. In [47], the author introduces BOINC (Berkeley Open Infrastructure for Network Computing) which is a collaborative platform used to solve a wide variety of projects (mainly) for scientific purposes. People who contribute in solving a task, are rewarded by credits which get them to a higher rank in a leaderboard. A similar crowdsourcing platform for researchers has been proposed in [48]. On the opposite side in recent years a new category of collaborative approaches is born for cryptovalue mining such as the well known Bitcoin, Litecoin and Ethereum. Users who aims at mining new Bitcoins contribute to a decoding task and they are rewarded with a portion of the gathered money that is proportional to the effort put in the mining task. We discuss in the detail this issue in Section 6 where we compare our rewarding strategy to them.

3. Coremuniti architecture

The goal of Coremuniti Network is to build a reliable infrastructure that allows to share computational resources in an easy and secure way. Our framework has to be robust against attacks from malicious users, analogously to every distributed computing [49] system [49] or distributed storage systems [50]. More in detail, we need to guarantee secure communication between clients and server, trusted software for remote execution and privacy for the intermediate computation. To this end, we implemented a hybrid peer to peer framework [15] where some functionality is still centralized, since we aim at guaranteeing continuous service availability.

Fig. 2 shows the interactions among system components. Herein: *Client Node* refers to those users who want to execute a high computing demanding *task* (also referred to as *project* in the following) using Coremuniti network. *Server Node* refers to those users who are willing to share with other Coremuniti network users the computational resources that are not fully employed on their devices. Finally, *Central Server* is the core of our system. It is in charge of the allocation of the nodes, the displacement of the messages and the scheduling of tasks issued from client nodes that have to be distributed among server nodes.

Our system is general purpose, every specific case study is implemented as a new dedicated plugin. Thanks to the use of software agents we are able to be independent from the low level processes and/or threads that are specific to the task being executed on the network. Fig. 2 depicts the overall system architecture and we can see how for each component we deploy several agents.

Aside from the Client and Server Nodes, two new components complete the architecture: the DB Server and the Web Server. The first consists of a classical relational DB which stores information about the network status, the users and the devices. The web server, instead, allows users to interact with the system via web interface.

In what follows, we describe main components of our architecture.

Definition 1 (*Client Node*)**.** The machines of all users who want to submit new projects to the Coremuniti Network run a Client Node. It allows the execution of the *NodeClientAgent* whose role is to create projects to be submitted.

More in detail, a project is submitted to an agent residing on Central Server component. Thanks to the *NodeClientAgent*, users, after authentication, can perform the following actions. *Create and send new project to Central Server* that allows the agent to create a new project by specifying the execution data and parameters. User can also *list all active projects*, *stop and resume the remote execution of a project* and *delete a project*. Finally, users can *download partial updates*, i.e., a user can download the results of the computation even if the main task is still not completed. Thus, if the output of the project is too big, it can be downloaded while it is still running. This feature is very useful for Rendering Type tasks where the output can be very large (typically in the order of Gigabytes).

Definition 2 (*Server Nodes*)**.** Users willing to share their computing resources making them available to other users act as *resource providers* by running a Server Node.

It allows the execution of the *NodeServerAgent*, whose job is to execute subtasks received from the Central Server. The agent is able to retrieve the technical features of the machine on which it is currently running and offers the opportunity to set the amount of computational resources to share with the network. Currently it is possible to share CPU or GPU power, RAM and disk space. The central server chooses the server nodes for a specific computation

taking into account several aspects related to the complexity of the task and the availability of computational resources.

The *Central Server* component implements a clustered architecture and manages the execution of tasks submitted to the Coremuniti Network. This is the component that centralizes some core functionality of the peer to peer network and allows the concurrent execution of different types of agents.

The *MainExecutionAgent* receives the remote execution requests performed by NodeClientAgent agents. For each request an *ExecutionAgent* is created, with the goal of handling the remote execution.

The primary role of MainExecutionAgent is to manage the load balancing of internal servers by properly assigning the ExecutionAgents to different machines.

The *ExecutionAgent* takes care of managing the remote execution of a single project. It performs the following jobs:

- *Subtask execution coordination for a single project.* This activity includes the initial assignment of subtasks to NodeServerAgents (Algorithm 1 reported in Section 4). During execution of the subtasks the ExecutionAgent receives updates about the execution status and once the computation ends, it receives the final subtask results from each NodeServerAgent. In addition to that, the ExecutionAgent periodically allocates new NodeServerAgents to subtasks if the assigned NodeServerAgents are slow (Algorithm 2 reported in Section 4).
- *Result Quality Control.* Specifically, a subtask is assigned to more than one resource provider. It is crucial for the ExecutionAgent to validate the obtained results. Since our protocol can be used in different scenarios by developing specialized plugins, the validation step is strongly tied to the process being executed. Generally, in order to perform the validation, different results are compared using a specific task evaluation metric. The comparison details can be implemented by using our *TaskValidation* API, that provides a fully customizable and easy way to manage different types of tasks in the Coremuniti network (as it will be shown for the rendering case study described in Section 5).
- *Credits Management.* When at least three NodeServerAgents complete the execution of the assigned subtask *st* the ExecutionAgent assigns credits to all NodeServerAgents working on *st* as described in Section 4.
- *Data Transfer Management (Upload/Download project/subtask data).* We use an internal file server that deals with data transfer among peers and central server.

The *PeerAgent* keeps track, using a database, of the information related to the execution of the tasks from the resource providers. It constantly monitors the availability of resource providers that can be assigned to new tasks and updates a pool of available resources that can be requested by the ExecutionAgent.

Since the primary goal of our framework is to minimize the overall completion time of a given problem, it is mandatory to estimate this total time in order to perform an effective division of the main task that guarantees good results in terms of execution time. The estimation of the overall completion time is performed by the *TaskEvaluatorAgent*. This component is also in charge of splitting the initial task into multiple subtasks with a similar expected duration.

Each TaskEvaluatorAgent runs on a single machine (denoted as reference machine). The reference machine adopted in the current implementation has the following technical specifications:

- CPU: Single core–Single thread 4.0 GHz Intel Processor
- RAM: 32 Gb DDR3 1600 Mhz
- HDD: SATA SSD 1 Tb 550 Mb/s peak

3.1. Communication protocols

This section describes the communication protocols between agents carrying out the main functions on the Coremuniti Network. Each project (task) *Pj* is a triple $\langle type, D, pms \rangle$, where *type* is the type of the task to be performed, *D* are the data on which the execution should be carried on and *pms* are the execution parameters. Each message *M* exchanged on the network has the form $\langle mtype, content \rangle$ where *mtype* is a string that defines the type of message and *content* is a generic object that specifies the content of the message.

In the following, we report the details of two communication protocols implemented in Coremuniti in order to enable the communication between the Central Server and Client/Server nodes: *Adding New Project Protocol* and *Task Assignment and Execution Protocol*. These protocols are open specification in order to allow the developer community to eventually design new Client/Server node implementations. (See Figs. 3 and 4.)

The main agents involved in the submission of a new task to the system are:

- ***NodeClientAgent***: It creates the project P_j by choosing its *type*, sending *D* and setting the values of one or more *pms* for the selected project *type*;
- ***MainExecutionAgent***: It listens for remote execution requests, creates a new ExecutionAgent and deploys it to a selected machine to balance the system load;
- ***ExecutionAgent***: it manages the project evaluation interacting with the TaskEvaluatorAgent, the PeerAgent and the NodeServerAgents;
- ***TaskEvaluatorAgent***: Receives projects from ExecutionAgent and estimates the *Pj* total cost splitting it in subtasks of uniform cost.

In order to add a new project, a NodeClientAgent sends to a MainExecutionAgent a message of type *RemoteExecutionRequest*. The MainExecutionAgent creates and deploys a new instance of ExecutionAgent trying to balance the system load. After the new ExecutionAgent is created, a new *ExecutionAgentAddress* message is sent to NodeClientAgent with a *content* containing the following information:

- Address and port of the ExecutionAgent
- Address and port of a data server
- Credentials for the data server

At this point the NodeClientAgent talks directly to the ExecutionAgent that will handle both the creation and the remote execution of the project. Each ExecutionAgent has an internal data Server in order to receive the data *D*. The user agent uploads *D* to the ExecutionAgent via data server using the credentials received with the previous message, (in our 3D rendering use case this data is the scene file for the project with *type*: *Rendering*). After the upload is completed, the NodeClientAgent can send a cost calculation request through a message with *mtype CalculateCostExecution* and with *content* embedding the values for the project parameters *pms*. In our example use case, the *pms* for the *Rendering* type projects are the numbers of the first and the last frame to render. Obviously, the value of the parameters can change the overall total execution time and therefore the cost of the project. For this reason, this message can be sent multiple times with different parameters in order to let the user choose the preferred parameters based on total project cost. When the first message with *mtype*: *CalculateCostExecution* comes is received by the ExecutionAgent, the *D* is uploaded to the data server and *pms* are sent via message to the TaskEvaluatorAgent to estimate the total cost of *Pj*. After the evaluation is completed a new message is

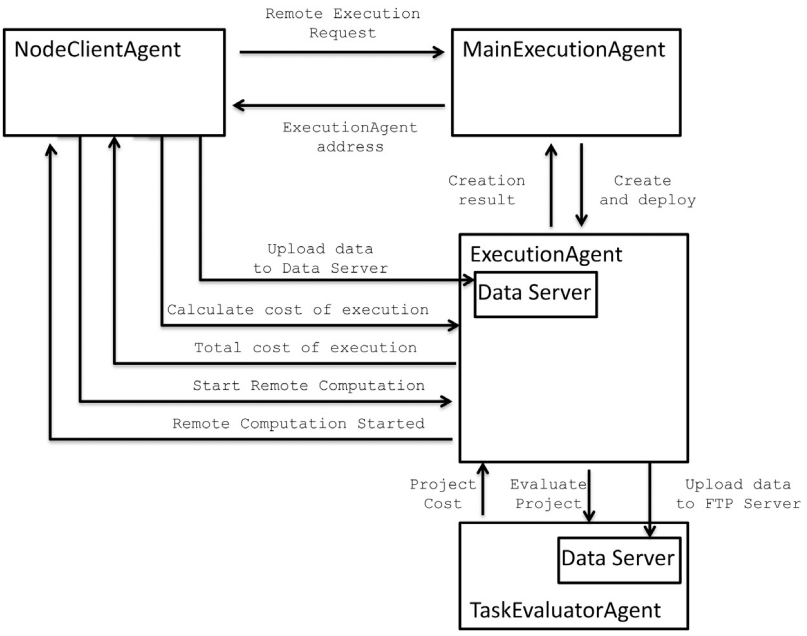

Fig. 3. Adding new project protocol.

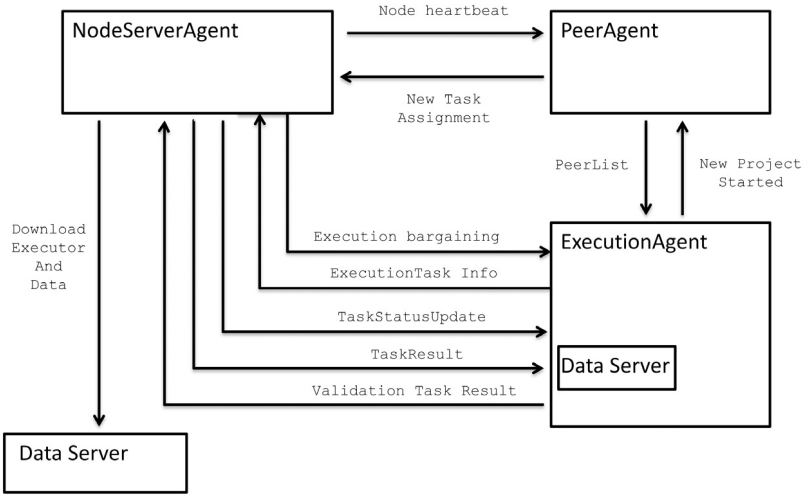

Fig. 4. Task assignment and execution protocol.

sent from the TaskEvaluatorAgent to the ExecutionAgent with the total cost and the number of subtasks of project. It is important to underline that an ExecutionAgent executes only one *Pj*, therefore a triple $\langle ExecutionAgent, Pj, pms\rangle$ uniquely identifies a request of *Pj* evaluation. As soon as the ExecutionAgent receives the message containing the total execution cost, it sends a new message to the NodeClientAgent notifying the total cost for the *Pj* with the selected *pms*. At this point the NodeClientAgent can start the remote execution sending a new message to the ExecutionAgent with *mtype StartRemoteComputation* and *content* the *pms* selected. If the user on NodeClientAgent has enough credits she can request the task execution: the ExecutionAgent writes on DB Server the project data and the remote computation can start; otherwise an error message is returned to the NodeClientAgent. In order to save resources on the machines where the ExecutionAgents are deployed a session timeout is set. If data *D* is not sent before the timeout the agent is closed and the resource released. Furthermore, after *D* is uploaded another timeout is used to allow users to select preferred *pms* (via *CalculateCostExecution* messages) and start remote computation. Again if this timeout expires the ExecutionAgent is closed.

Since each Coremuniti project *Pj* must be executed in a parallel way among the nodes of the network, *Pj* is split into several subtasks $(st_1, st_2, \ldots, st_n)$. In this section we will describe the interaction between the Server Nodes (*resource providers*) and the Central Server that occur when a subtask of a project *Pj* needs to be retrieved and executed. The main actors involved in this phase are:

- ***NodeServerAgent***: Receives the subtask to execute, downloads the data and uploads the results to the ExecutionAgent,
- ***PeerAgent***: Provides a pool of NodeServerAgent to the ExecutionAgent,
- ***ExecutionAgent***: Manages the remote execution of *Pj*.

When a NodeServerAgent is available, it sends a message to PeerAgent with *mtype NodeHeartBeat* to notify its presence and

Table 1
NodeServerAgent beat message content.

Field	Description
`Device name`	Name of the computational device
`Device type`	Type of the device. Currently supported types are *CPU* and *GPU*
`Core Number`	Number of cores of the device (optional)
`Available RAM (Mb)`	Total amount of RAM of the device
`Available HD (Mb)`	Total amount of disk space of the device
`Currently Used RAM (Mb)`	Currently used RAM of the device
`Currently Used Device Power (%)`	Percentage of device power currently used
`Currently Used HD (Mb)`	Amount of currently used disk space of the device
`Max Shared RAM (Mb)`	Maximum amount of RAM the user wants to share
`Max Shared Device Power (%)`	Maximum amount of device power the user wants to share
`Max Shared HD (Mb)`	Maximum amount of disk space the user wants to share
`Public signature key`	A public key used to sign the results sent to the central server

to ask for tasks to perform. The *content* of this message, shown in Table 1, is a list of information about the architecture and the status of the device on which the agent is running.

This message is cyclically sent to PeerAgent to notify the peer availability. A node is considered alive if

$$LTU < CST - \Delta t$$

where *LTU* is the timestamp of the *NodeHeartBeat* message, *CST* is the timestamp of the current server time and Δt is a fixed time interval (currently 60 s). In the actual state, in terms of GPUs, we only support Nvidia cards with CUDA-Enabled [51,52].

When an ExecutionAgent starts a remote project, it sends a message containing *mtype*: *NewProjectStarted* to the PeerAgent. This agent selects a pool of available NodeServerAgents that can be assigned to the project. Afterwards it sends a message to the ExecutionAgent with *mtype*: *PeerList* and *content* made of a list of nodes ids that are authorized to work on *Pj*. Subsequently, a message with *mtype*: *NewTaskAssignment* is sent by the ExecutionAgent to all the NodeServerAgents that have been chosen. This message contains the ExecutionAgent address and port. At this point, the NodeServerAgent contacts the ExecutionAgent using a message of the form *mtype*: *ExecutionBargaining* and, if authorized, it will receive an *ExecutionTaskInfo* message. This message contains all the information needed by the agent in order to execute the subtask and to send the results to the ExecutionAgent:

- ***Executable Links***: Links to download the executable file for the assigned task. This is a list of several links, each for a different system architecture and Operating System. Currently OS supported are Windows (x86 and x64), Mac Os (x64) and Linux (x86 and x64).
- ***Params***: Parameter to pass to the executable. For the Rendering type projects are the coordinates and the number of frames to be rendered.
- ***Upload Server Data***: These are the information needed to connect to the in internal ExecutionAgent data Server: address, port, username and password.

As soon as the *ExecutionTaskInfo* message has been received, the NodeServerAgent downloads the data *D* and starts the execution of the assigned subtask. Furthermore, cyclically it sends a message with *mtype*: *TaskStatusUpdate* to notify the local subtask execution progress. The *content* field of this message contains a couple $\langle ES, Perc \rangle$ where *ES* is the Execution Status and *Perc* is a percentage representing the current state of completion of the subtask. The *ES* variable can be one of the values: *DOWNLOAD*, *EXECUTION* or *UPLOAD*. When the execution of the subtask ends, the agent uploads the results (typically a file) to the Execution Agent data Server (e.g., in our 3D rendering case study, the result is an encrypted object containing the rendered tiles). A message with *mtype*: *TaskResult* is then sent by the NodeServerAgent in order to notify the ExecutionAgent about the completion of the task.

4. Subtask assignment and credit rewarding

In order to properly assign subtasks to resource providers we developed a mathematical model, described in this section, whose primary goals are the following:

1. it aims at minimizing the expected completion time for the overall task
2. it takes into account resource providers' revenue expectations

More in detail, as explained above, when a user submits a task to the Coremuniti network, this task is split in several subtasks, much easier to solve than the initial one. Every subtask can be completed using a reasonable amount of computational resources by the server nodes connected to our P2P network. The obtained results are then combined in order to produce the initial task solution. On the opposite side, users providing their computational power to the Coremuniti network wish to maximize their revenues by executing as much tasks as they can, in order to gain as many credits as possible. Obviously enough, the assignment should take into account the computing capability of each node in order to give to every node the chance to gain credits for their subtask executions. In the following we describe our assignment model.

The model assumes the presence of a set of available resource providers $\mathcal{RP} = \{rp_1, rp_2, \ldots, rp_n\}$, and of a function $\lambda_c : \mathcal{RP} \rightarrow N \times N$, that assigns to every resource provider *rp* a pair $\langle tmin, tmax \rangle$, where *tmin* (resp. *tmax*) is the minimum (resp. maximum) execution time required by the resource node to complete a subtask.

Furthermore, we leverage a credit assignment function which is based on the "usefulness" of the results yielded by the resource providers. Specifically, let *st* be a subtask whose execution was assigned *c* credits and which was assigned to the resource providers $rp_{i_1}, rp_{i_2}, \ldots, rp_{i_x}$. We sort the resource providers $rp_{i_1}, rp_{i_2}, \ldots, rp_{i_x}$ ascending w.r.t. their completion times and store them in a sequence RP_{st}. Resource providers which did not terminate their subtask are put at the end of the sequence RP_{st} ordered according to the percentage of the subtask they completed.

When at least three resource providers completed their work, we assign $\frac{3c}{10}$ of the total credits, paid by the client for running the subtask on the network, to the first three resource providers in RP_{st}, i.e., the resource providers which "step up on the podium" for returning *st* result. Moreover, we distribute the remaining $\frac{c}{10}$ credits among the other resource providers in RP_{st} as follows. For each $j \in [4 \ldots x]$ we assign to $RP_{st}[j]$ the credits given by the following formula

$$\frac{c \cdot Compl(RP_{st}[j])}{10 \cdot \sum_{k=4}^{x} Compl(RP_{st}[k])},$$

where $Compl(rp)$ is the percentage of the subtask st which was completed by rp.

During the task assignment phase we aim at finding an optimal distribution of the subtask among resource providers which minimize the expected completion time while guaranteeing that:

1. to every resource provider is assigned at most a subtask,
2. it is possible, for every resource provider with a subtask assigned, to be one of the first three resource providers completing the subtask.

Subtask assignment proceeds in two phases. First, an initial assignment of the various subtasks to resource is yielded (Algorithm 1). Next, at regular intervals, the systems check for the overall completion of the project and assign resource providers that are available to new subtasks (Algorithm 2).

Let st be a subtask and $RP = \{rp_1, rp_2, \ldots, rp_k\}$ be the set of resource providers assigned to st. The expected completion time of st given RP is denoted as $EC_{st,RP}$ and is computed as follows. Let $\vec{t_\downarrow}$ and $\vec{t_\uparrow}$ be two vectors reporting, respectively, the minimum and maximum completion times of the resource providers in RP in increasing order. Let t' denote the value $t_\downarrow[2]$ and t'' denote the value $t_\uparrow[2]$. Let $vect = [t_0 = t', \ldots, t_k = t'']$ be a vector containing all the values in $\vec{t_\downarrow}$ and $\vec{t_\uparrow}$ which lie in the interval $[t', t'']$. Hence, denoting with $F^3(x)$ the probability that at least three resource providers complete their job before time limit x, we have that:

$$EC_{st,RP} = AMZ@P \int_0^\infty (1 - F^3(x))dx = t' + \int_{t'}^{t''} (1 - F^3(x))dx = AMZ@Pt' + \sum_{i=0}^{k-1} \int_{t_i}^{t_{i+1}} (1 - F_i^3(x))dx = AMZ@Pt' + \sum_{i=0}^{k-1} FF_i^3(t_i, t_{i+1}),$$

where $F_i^3(x)$ is the probability that at least three resource providers complete their work before x time limit given that $t_i \leq x \leq t_{i+1}$ and $FF_i^3(t_i, t_{i+1})$ is equal to $\int_{t_i}^{t_{i+1}} (1 - F_i^3(x))dx$. It is easy to see that $FF_i^3(t_i, t_{i+1})$ can be easily derived from the minimum and maximum completion times associated to every resource provider by the function λ_c.

Example 1. Consider the case that a set of four resource providers $RP = \{rp_1, rp_2, rp_3, rp_4\}$ was assigned to st where $\lambda_c(rp_1) = \langle 1, 5\rangle$, $\lambda_c(rp_2) = \langle 2, 7\rangle$, $\lambda_c(rp_3) = \langle 6, 7\rangle$ and $\lambda_c(rp_4) = \langle 4, 8\rangle$. In this case, $t' = 4$ and $t'' = 7$ and

$$\begin{aligned} &EC_{st,RP} = \\ &4 + \int_4^5 (1 - F_0^3(x))dx + \int_5^6 (1 - F_1^3(x))dx + \int_6^7 (1 - F_2^3(x))dx = \\ &4 + \int_4^5 (1 - \tfrac{x-1}{5-1}\tfrac{x-2}{7-2}\tfrac{x-4}{8-4})dx + \int_5^6 (1 - \tfrac{x-2}{7-2}\tfrac{x-4}{8-4})dx + \\ &\int_6^7 \left(1 - \left(\tfrac{x-2}{7-2}\tfrac{x-6}{7-6} + \tfrac{x-2}{7-2}\left(1 - \tfrac{x-6}{7-6}\right)\tfrac{x-4}{8-4} + \left(1 - \tfrac{x-2}{7-2}\right)\tfrac{x-6}{7-6}\tfrac{x-4}{8-4}\right)\right) dx = \\ &4 + \int_4^5 (1 - \tfrac{x-1}{4}\tfrac{x-2}{5}\tfrac{x-4}{4})dx + \int_5^6 (1 - \tfrac{x-2}{5}\tfrac{x-4}{4})dx + \\ &\int_6^7 \left(1 - \left(\tfrac{x-2}{5}\tfrac{x-6}{1} + \tfrac{x-2}{5}\left(1 - \tfrac{x-6}{1}\right)\tfrac{x-4}{4} + \left(1 - \tfrac{x-2}{5}\right)\tfrac{x-6}{1}\tfrac{x-4}{4}\right)\right) dx = \\ &4 + \left[\tfrac{11x}{10} - \tfrac{7x^2}{80} + \tfrac{7x^3}{240} - \tfrac{x^4}{320}\right]_4^5 + \left[\tfrac{3x}{5} + \tfrac{3x^2}{20} - \tfrac{x^3}{60}\right]_5^6 \\ &+ \left[\tfrac{1}{10}(-126x + 44x^2 - \tfrac{17x^3}{3} + \tfrac{x^4}{4})\right]_6^7 = \\ &4 + \tfrac{901}{960} + \tfrac{11}{15} + \tfrac{31}{120} \approx 5.93 \end{aligned}$$

Before describing the details of our subtask assignment Algorithms, we define some additional notations. A subtask assignment $\mathcal{SA}$ is a set of pairs of the form $\langle st, rp\rangle$, where st is a subtask and rp is a resource provider. Moreover given a subtask assignment $\mathcal{SA}$ and a subtask st, we denote with $\mathcal{SA}(st)$ the set of the resource providers assigned to st in $\mathcal{SA}$, i.e., $\mathcal{SA}(st) = \{rp | \langle st, rp\rangle \in \mathcal{SA}\}$, and we denote as $\mathcal{ST}(\mathcal{SA})$ the set of subtasks mentioned in $\mathcal{SA}$, i.e., $\mathcal{ST}(\mathcal{SA}) = \{st | \langle st, rp\rangle \in \mathcal{SA}\}$. Furthermore, given a subtask assignment $\mathcal{SA}$ we define as $MaxECP(\mathcal{SA})$ the maximum expected completion time of a subtask in $\mathcal{SA}$, i.e., $MaxECP(\mathcal{SA}) = \max_{st \in \mathcal{ST}(\mathcal{SA})}(EC_{st,\mathcal{SA}(st)})$.

The initial subtask assignment is computed by running Algorithm 1. It works in two phases. First, an initial assignment of resource providers to tasks is yielded by assigning a resource provider at a time to the every subtask paying attention to minimize the maximum expected completion time of all the subtasks. This is done considering resource providers in ascending order of their expected completion time (lines 2–8). Next, the initial assignment is revised by swapping pairs of task between resource providers. The algorithm iteratively selects a pair of assignments $\langle st', rp'\rangle, \langle st'', rp''\rangle$ which once swapped provide the greatest decrement of the maximum expected completion time (lines 9–12). Finally, the subtask assignment is returned.

Input: Resource provider sequence $\mathcal{RP}$ of size n
Input: Subtask sequence $\mathcal{ST}$ of size $k < \frac{n}{3}$
Output: Tasks assignment $\mathcal{SA}$
1: $\mathcal{SA} = \emptyset$
2: $\mathcal{RP} = order(\mathcal{RP})$
3: **for** $i = 1$ to n **do**
4: $st = selectBestST(\mathcal{ST}, \mathcal{SA}, \mathcal{RP}[i])$
5: **if** $st \neq$ `null` **then**
6: $\mathcal{SA} = \mathcal{SA} \cup \langle st, \mathcal{RP}[i]\rangle$
7: **end if**
8: **end for**
9: **repeat**
10: $(\langle st', rp'\rangle, \langle st'', rp''\rangle) = selectBestCP(\mathcal{SA})$
11: $\mathcal{SA} = \mathcal{SA} - \{\langle st', rp'\rangle, \langle st'', rp''\rangle\} \cup \{\langle st', rp''\rangle, \langle st'', rp'\rangle\}$
12: **until** $existsCP(\mathcal{SA})$
13: **return** $\mathcal{SA}$

Algorithm 1: Initial Assignment

Function *order* receives as input a sequence of resource providers $\mathcal{RP}$ and returns $\mathcal{RP}$ sorted ascending w.r.t. the expected resource provider completion time. Function *selectBestST* returns, given a resource provider rp and a subtask assignment $\mathcal{SA}$, the subtask $st \in ST$ such that $MaxECP(\mathcal{SA} \cup \{\langle st, rp\rangle\})$ is the minimum, i.e., $st = \arg\min_{st \in \mathcal{ST}}(MaxECP(\mathcal{SA} \cup \{\langle st, rp\rangle\}))$, such that $MaxECP(\mathcal{SA} \cup \{\langle st, rp\rangle\}) < MaxECP(\mathcal{SA})$, `null` otherwise. Since in the initial step of the algorithm there could be some subtasks st in $\mathcal{ST}$ such that $|\mathcal{SA}(st)| = x < 3$, for each subtask st satisfying this constraint, when computing $MaxECT$ function *selectBestST* assumes that $3 - x$ fake resource providers have been assigned to st, where the minimum and maximum completion times of these fake resource providers are $max - 1$ and max, respectively, where max is a constant value (much) greater than the maximum completion time of every resource provider in $\mathcal{RP}$.

Function *selectBestCP* returns a pair of assignments $\langle st, rp\rangle$ and $\langle st', rp'\rangle$ in $\mathcal{SA}$ such that there do not exist an assignment $\langle st'', rp''\rangle$ and $\langle st^*, rp^*\rangle$ in $\mathcal{SA}$ such that $MaxECP(\mathcal{SA} - \{\langle st, rp\rangle, \langle st', rp'\rangle\} \cup \{\langle st, rp'\rangle, \langle st', rp\rangle\}) > MaxECP(\mathcal{SA} - \{\langle st'', rp''\rangle, \langle st^*, rp^*\rangle\} \cup \{\langle st'', rp^*\rangle, \langle st^*, rp''\rangle\})$. Function *existsCP* returns true if there is a pair of assignments $\langle st, rp\rangle$ and $\langle st', rp'\rangle$ in $\mathcal{SA}$ such that $MaxECP(\mathcal{SA} - \{\langle st, rp\rangle, \langle st', rp'\rangle\} \cup \{\langle st, rp'\rangle, \langle st', rp\rangle\}) < MaxECP(\mathcal{SA})$. Moreover, both functions *selectBestCP* and *existsCP* consider only pairs of assignments $\langle st, rp\rangle$ and $\langle st', rp'\rangle$ in $\mathcal{SA}$ such that swapping rp and rp' guarantees that the probability that rp (resp. rp') is among the first three resource providers assigned to st' (resp. st) completing st is greater than zero.

The following proposition describes the behavior of Algorithm 1.

Proposition 1. *Let $\mathcal{RP}$ be a sequence of resource providers of size n and $\mathcal{ST}$ be a sequence of tasks of size $k < \frac{n}{3}$. Algorithm 1 returns a subtask assignment $\mathcal{SA}$ in polynomial time w.r.t. $|\mathcal{RP}|$ and $|\mathcal{ST}|$ such that*

1. *for each* $st \in \mathcal{ST}$ $|\mathcal{SA}[st]| \geq 3$, *and*
2. *for each* $rp \in \mathcal{RP}$ *if* $\langle st, rp\rangle \in \mathcal{SA}$ *the probability that rp is among the first three resource providers that complete st is greater than zero.*

Proof (*Sketch*)**.** The first property follows from the hypothesis that $k < \frac{n}{3}$ and the fact that the initial assignment of resource providers to tasks (lines 2–8) is done by selecting the subtask *st* that provides the largest decrement of the maximum expected completion time. Therefore, it is easy to see that, as in the case that a subtask *st* has been assigned less than three resource providers, at least one fake resource provider is considered in the computation of *st* expected completion time and the minimum completion time of a fake resource provider is much larger than the maximum completion time of any (real) resource provider. Furthermore, it cannot happen that more than three resource providers are assigned to a subtask before that all the other tasks have been assigned at least three resource providers.

As regards the second property, it straightforwardly follows from the definition of functions *selectBestST* and *selectBestCP*.

Finally, it can be shown that the algorithm runs in polynomial time w.r.t. $|\mathcal{RP}|$ and $|\mathcal{ST}|$ by proving that in the case that a pair of assignments is returned by function *selectBestCP* at least one of the resource providers that will be swapped, will never be returned by subsequent invocation of function *selectBestCP*. □

Coremuniti central server runs Algorithm 2 periodically, in order to check for the overall completion of the project and to assign resource providers, that have completed the assigned tasks, to new tasks. The statistics of the resource providers involved in the computation of the subtasks are updated before running this algorithm, so that the system deals with up to date values for minimum and maximum completion times. Essentially Algorithm 2 first sorts the available resource providers ascending w.r.t. their expected completion times and next iteratively assigns each resource provider *rp* to the subtask *st* selected by invoking function *selectBest*. Function *selectBest* returns, given a resource provider *rp* and a subtask assignment $\mathcal{SA}$, the subtask $st \in ST$ such that $MaxECP(\mathcal{SA} \cup \langle st, rp\rangle)$ is the minimum, i.e., $st = \arg\min_{st \in \mathcal{ST}}(MaxECP(\mathcal{SA} \cup \langle st, rp\rangle))$ and is such that $MaxECP(\mathcal{SA} \cup \{\langle st, rp\rangle\}) < MaxECP(\mathcal{SA})$, `null` otherwise.

Input: Available resource providers sequence $\mathcal{RP}$ of size n
Input: Subtask sequence $\mathcal{ST}$
Input: Initial Subtasks assignment $\mathcal{SA}$
Input: Subtasks and Resource Providers constraints *STC*
Output: Revised Subtasks assignment $\mathcal{SA}$
1: $\mathcal{RP} = order(\mathcal{RP})$
2: **for** $i = 1$ to n **do**
3: $st = selectBest(\mathcal{ST}, \mathcal{SA}, \mathcal{RP}[i])$
4: **if** $st \neq$ `null` **then**
5: $\mathcal{SA} = \mathcal{SA} \cup \langle st, \mathcal{RP}[i]\rangle$
6: **end if**
7: **end for**
8: **return** $\mathcal{SA}$

Algorithm 2: Incremental Assignment

5. Case study: Efficient and effective 3D rendering

Many real life applications require huge computing resources in order to properly execute. As an example we mention here physical science simulation, mathematical simulation for insurance companies, biology simulation and cryptography. In this section we will describe our Coremuniti™ based solution for a quite interesting scenario, i.e., 3D Professional Rendering. More in detail, rendering is the process of converting a graphical model into a high quality image by means of a computer program. The input of the rendering process is the *model* (also called *scene*), composed of texture, lighting, shading, viewpoint and geometry information. Starting from these information, many rendering algorithms have been implemented in order to obtain the final image. The most important concept used by these algorithms is *light tracing*. A naive solution to the rendering problem could be obtained by tracing every particle of light from every source to every object in the scene. This solution is (obviously) impractical because it calls for massive amount of computational time. Moreover, it is useless because some portion of the scene being rendered will not be visible in the final image in the majority of real life rendering task. In particular, only a space region of the model may appear as a part of the rendered image, it is referred as *view frustum*. The exact shape of this region varies depending on many factors such as the viewing direction and the camera being used.

Based on this assumption a more efficient light transport modeling techniques have been defined, such as:

- *Ray casting*: it runs the rendering process starting from a point of view and using basic reflection laws and elementary geometry computes the rendering output. The obtained image quality could be improved by using *Monte Carlo* based correction techniques;
- *Ray tracing*: it works analogously to Ray Casting, but it is based on more sophisticated optical laws [53].

The use of the above mentioned approaches allows to limit the light tracing execution only to those pixels that will be visible in the final image, nevertheless, the rendering process still requires high performance resources for running to completion if the input scene is (slightly) complex.

Although many advances in computing device technology led to affordable computational power (thus ameliorating the curse of huge execution time for professional use), the always growing demand of photorealistic results and high resolution artistic products, make the rendering problem more and more challenging. Coremuniti provides a nice solution to this problem by taking advantage of the resources provided by users joining our P2P network. In Fig. 5, we report the Coremuniti Server status when executing a rendering task. In particular, resource providers can set the amount of resources they are willing to share both as a percentage of the total CPU usage (in Fig. 5 it has been set to 80%) and the maximum amount of memory (in Fig. 5 it has been set to 1 GB).

It is straightforward to note that the rendering task is highly parallelizable as frames and tiles (composing each frame) can be rendered independently. Parallel rendering drastically improves the speed of rendering operations. If we take into account ray tracing we can divide the whole view frustum into portions. Each piece of the frustum can be rendered on different nodes using the appropriate rays in order to obtain the corresponding portion of the rendered image. It is easy to see that single machine execution requires to send rays to all the pixels in the frustum slowing down the overall rendering time.

Mozaiko natively provides the opportunity to parallelize the rendering process on the Coremuniti network. In our implementation, we use *Blender* [54] rendering models. Blender is a professional computer graphic software available for many platforms that is actually considered one of the best tool for professional rendering and movie making. As a default add-on, Blender provides *Cycles* render engine that is based on ray-tracing. An interesting feature provided by Cycles is the *Render Border* function. Border rendering works on a smaller piece of the original render model. It is typically used to refine the rendering of an image portion that requires longer execution times w.r.t. the rest of the image or for

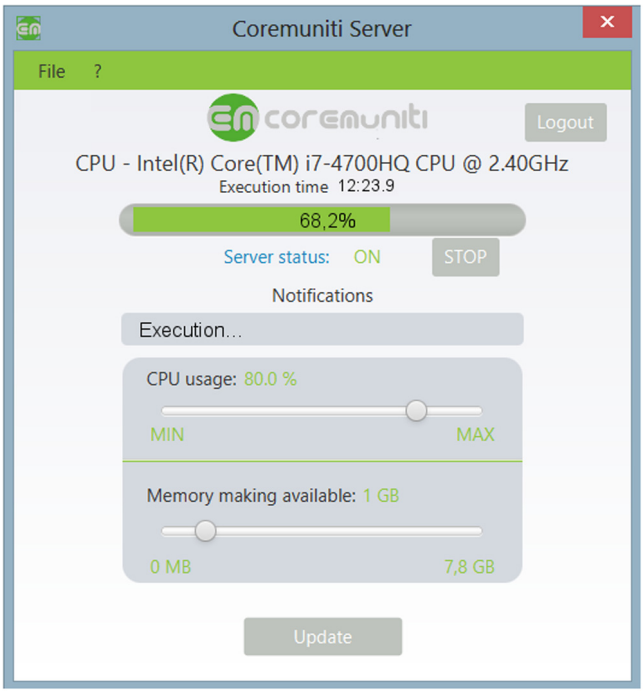

Fig. 5. Coremuniti server at work.

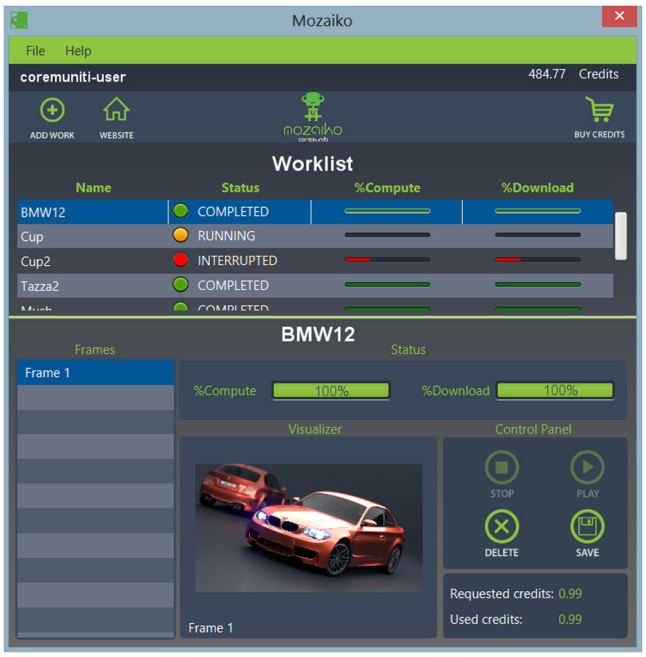

Fig. 6. Mozaiko rendering view.

previewing purposes. The images are then easily combined in order to obtain the final rendered image.

More in detail, the rendering process starts from the pixel that will be visible in the final image and sends a "ray" into the scene to identify the exact color for that pixel. Each ray will bounce around the scene from object to light source based on rendering settings set to calculate the final result. While the render border will reduce the pixels used as the starting point for each ray, it does not reduce the objects or lights in the scene that each ray may encounter on its way. Each ray crossing the scene will still "see" every visible object and light effects in the scene that can influence the final result for each pixel. Using border rendering, then, does not introduce any degradation on the quality of the resulting output image. Mozaiko uses border render function to split the original model to be rendered into rectangular regions that are sent to the computing nodes for parallel rendering. As the intermediate results from each node are gathered, the rendered regions are merged into the final image. Fig. 6 depicts the Mozaiko client. It allows to check the completion status of the tasks that have been launched and to see the preview of the rendering output. It is also possible to control the execution flow and to check the credits needed to perform the rendering.

Rendering Task Validation. A crucial requirement for our architecture is the detection of incorrect task execution. Since we render the models using distributed resources, it is mandatory to implement a checking strategy to determine if the rendering task solved by peers is correct. In particular, we need to output properly rendered files to requesting users and identify possible malicious actions (e.g., a user submitting a wrong image with the intent of damaging the network). As explained in Section 4, we ask several peers to solve a specific task and we wait for their answers. Since these results may slightly differ each other, we need to find a way to evaluate them in order to output the best one w.r.t. the requesting user viewpoint. As this process involves complex neurological mechanism that is beyond the scope of this paper, we simply mention here that we applied a method based on visual perception metrics [55] in order to determine the similarity threshold for the pixels that differs among rendered images. If the difference is higher than the obtained threshold, we randomly choose a small subset of pixels we render on our specialized backup server in order to determine the best result to be returned to the requesting client. As a final note, we point out that, it is mandatory to prevent the injection of malicious script in the model to be rendered (e.g., Blender allows to execute custom python script in a 3D model). In our software, we perform two security actions. We first disable the execution of unnecessary scripts, in the case that the script is claimed by the client to be crucial for the rendering completion, we run it in a secured sandbox [56].

6. Experimental evaluation

In this section, we experimentally evaluate Coremuniti performances from two standpoints: (1) we assess the efficiency and effectiveness of our approach by comparing our performance against the state of the art solutions available; (2) we perform a comparison of the revenue users can get by joining our network w.r.t. other collaborative approaches that reward users (e.g., Bitcoin mining).

We focus our attention on two different kinds of tasks: 3D rendering a matrix multiplication (implemented by the highly parallelizable Strassen algorithm described in [57]) since it is one of the basic operation behind most of computer simulations. For the latter we used the dataset of the University of Florida Sparse Matrix Collection, available at [58].

The experimental evaluation was carried out on our test network that involves 200 peers that have been classified according to the computational power of their devices. The network features are reported in Table 2.

As explained above, our software works in parallel with the user activities. This is a crucial assumption as traditional approaches for high performance computing rely on dedicated resources. We measured the running time of the rendering tasks by considering several usage scenarios for our resource providers:

- **Low-Use:** users who are performing activities that do not require huge amount of computing resources such as browsing simple pages and/or editing text (this is our best case);

Table 2
Computation Power of resource providers in our network.

Type	Equipment	Peer percentage
BASIC	Intel i5-6440HQ - 8 GB RAM - GPU Nvidia GeForce GTX 950M	60
INTERMEDIATE	Intel i7-4790K 4 GHz - 32 GB RAM - GPU Nvidia 980 Ti 6 Gb RAM	30
ADVANCED	Xeon E5-26700 2.60 GHz - 128 GB RAM - GPU NVIDIA GK110GL	10

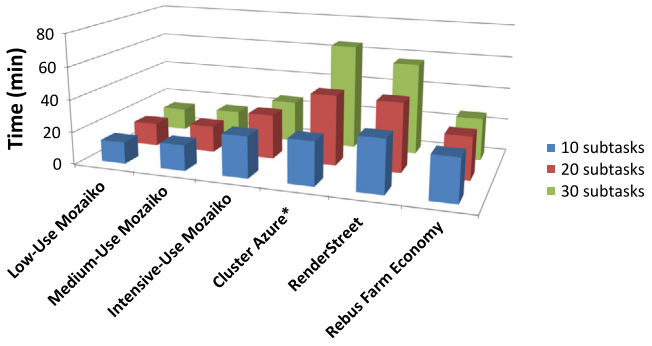

Fig. 7. Execution times comparison — 3D rendering.

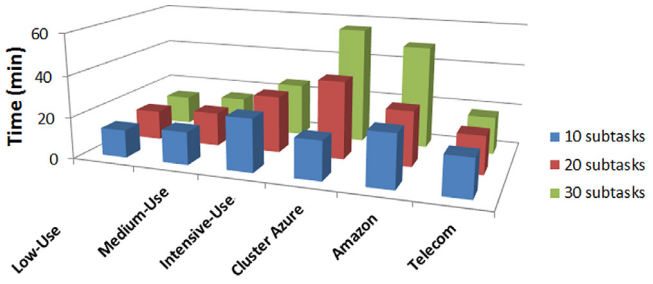

Fig. 8. Execution times comparison — matrix multiplication.

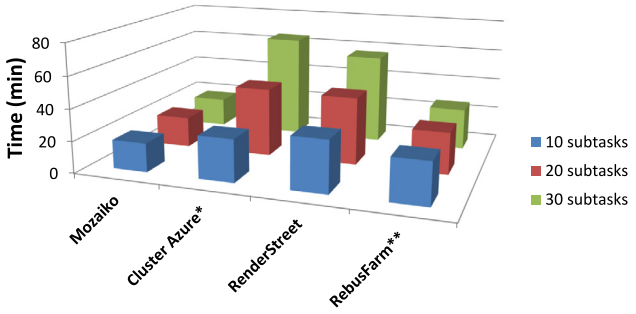

Fig. 9. Aggregate execution times comparison.

- **Medium-Use:** users who are playing multimedia files (this is our medium case);
- **Intensive-Use:** users who are playing video games (this is our worst case).

For each possible scenario, we run three task categories (each category being composed of 10 tasks). We choose for each category a set of models that can be partitioned respectively in 10, 20 and 30 subtasks but we point out that there is in principle no fixed upper bound for the number of subtasks.

6.1. System performances

In order to validate our framework we conduct a comparison against some of the most popular specialized rendering server farms in the case of 3D rendering (i.e., RenderStreet [59] and Rebus-Farm [60] and a well known cloud service platform, i.e., Microsoft Azure (8 Virtual Machines D5 V2 having 2.4 GHz Intel Xeon E5-2673 v3 - 16 Core (Virtualized) CPU, 56 Gb RAM, 800 GB HD) [61]) and some of the most popular cloud service platforms (i.e. Microsoft Azure, Amazon Cloud and Telecom) in the case of matrix multiplication.

6.1.1. Time performances

In Figs. 7 and 8, we report the execution times of our solution by running a first series of tests by asking our users to use their devices according to the above mentioned usage scenarios. In order to extremely stress our approach, we gave our competitors the advantage of discarding the (eventual) set-up time for the cloud and the server farm devices (marked in Fig. 7 as $*$ and $**$). More in detail, both Render Farm and Render Street require users to sign up and choose a profile type, if the user chooses the cheapest one, her tasks will be categorized as low priority. This significantly slows down the overall execution time. Moreover, among the proposed usage accounting model we chose for each competitor the topmost, i.e., the one that claims the better performances.

It is easy to see that, except for our worst case scenario (i.e., when our service providers are all intensively busy with other activities) our execution times are always better than our competitors. Even in the intensive use case, our times are slightly higher than RenderFarm. However, we point out that we discarded the setup time that, if summed up to the execution time, is likely to be higher than our execution in our worst case. Finally, it is worth noticing that the execution times of the cloud based solution grow almost linearly with the number of the tasks being executed, while they keep almost constant for our framework and the server farm based solutions. As regards the matrix multiplication scenarios, it is easy to see that Coremuniti performances are still better than the services we compare to.

Our compelling performances are better outlined when we consider a usage scenario where resources providers are uniformly distributed on our network as shown in Fig. 9.

6.1.2. Cost evaluation

Our strongest advantage w.r.t. our competitors is the money that users can save using our network. Due to our peculiar model we do not suffer from fixed cost deriving from continuous update of the hardware infrastructure. Indeed, users joining our network are mainly professionals and technical faculty university students that own up to date computing devices, thus our network do not suffer from technical obsolescence. This competitive advantage, reflects on our cost model, as we can use a fixed price for task (in our use-case scenario is 0.12 Euro) (see Fig. 10).

The total cost paid by users executing their tasks on our network is always lower than the cost paid if using our competitor solutions. Moreover, as for the time performances, we gave our competitor some advantage in the calculus, i.e., we do not account the cost for signing to their service, while users can join our Coremuniti network for free. Moreover, we do not account also the possibly accumulated credits by users that prior to ask for additional resources may had provided their own computational time to the network. Finally, we redistribute the 80% of the paid cost to reward the resource providers. Next paragraph is devoted to discuss this issue.

Table 3
Actual gain comparison.

Network	Max effort	Network Difficulty	Coin gain	Current rate	Daily gain (Euro)
Coremuniti	10 subtask/hour	–	47,36	0,12	5,68
Bitcoin	240 Mhash/s	3007,38 G	0,000000002	8698,71	0,000017
Litecoin	310 KHash/s	5038170	0,0000323	175,78	0,01
Ethereum	18,5 Mhash/s	3030,53 T	0,001465	713,7	1,05

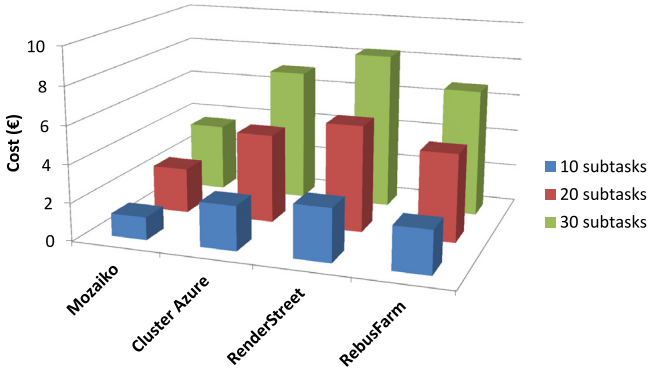

Fig. 10. Cost comparison.

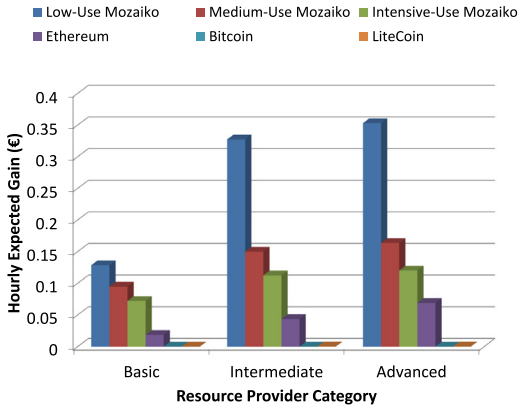

Fig. 11. Gain comparison.

6.2. User revenues evaluation

As mentioned above we redistribute to resource providers a great portion of the money paid by users requesting computational power to the Coremuniti network. In order to compare the possible outcome for joining Coremuniti network with respect to other collaborative systems that rewards their users we compare our performances to the most popular services currently available using their own metrics, i.e. *Bitcoin* mining [62], *Litecoin* mining [63] and *Ethereum* smart contracts [64]. First of all, we must clarify that those services require intensive-use advanced devices in order to give users a chance to be rewarded (according to publicly available statistics) as reported in Fig. 11 (we reported in figure the expected hourly gain obtainable by using different device categories).

For the sake of clarity, in Table 3 we provide more details on the key parameters for actual system comparison. The parameter *Network Difficulty* is peculiar to cryptovalue mining while it is not applicable to our network. We considered a resource provider holding a video card *Nvidia GTX 980 Ti (EVGA)* fully dedicated to task execution for the considered networks. We were forced to choose this device because our basic and intermediate settings are not powerful enough to get neither Bitcoin or Litecoin or Ethereum revenues. The device is able to complete our task in 5 min and 30 s. This means that it is able to complete 10 tasks per hour while for mining the other coins it can decode the number of hashes reported in Table 3. Considering that we redistribute to the network 0.12 Euro for each accumulated credit, by applying our mathematical task assignment model the user can get up to 5.68 Euros per day. While the maximum revenue with other networks is 1.05 Euros obtained by Ethereum (We considered the exchange rates available on 27st Feb 2018).

6.3. Remarks

In [65], the authors deal with the emergent Crowdsourced top-k problem. The latter consists in asking the crowd to take a decision over an objects comparison and in identifying the top-k best ones. In our case study we dealt with the problem of comparing rendered images, which is typically hard for machines. In order to further assess the quality of our proposal, we evaluated our results by an approach inspired by [65]. In particular, we asked users involved in our network to evaluate the obtained rendering (for each rendering we asked this question to *all* the peers). We obtained a quite encouraging result, i.e., 96% of our renderings have been validated by at least 90% of the crowd.

7. Conclusion and future work

In this paper, we proposed a hybrid peer to peer architecture for computational resource sharing. Users joining our network are able to share their unexploited computational resources and are rewarded by tangible credits. The computational power shared is used to solve difficult task submitted by other users. In order to guarantee the efficiency and effectiveness of the computation process, we designed a task partitioning and assignment algorithm that reduces the execution times while allowing satisfactory revenues for resource providers. We conducted extensive experiments in a real life scenario such as 3D rendering to assess our performances. As shown in our experimental evaluation our framework is able to improve the use of efficient digital solutions in “traditional" field like engineering and architectural design, by offering a new high speed and low cost tool for increasing their productivity.

7.1. Future work

We are currently developing specialized plugins for other challenging application fields such as medical and insurance simulation. Furthermore, we are in the process of evaluating the energy consumption achievable by leveraging our Coremuniti network. More in detail, the ICT industry currently accounts for about 2% of global emissions of carbon dioxide (CO2). Many international organizations recommend the sector to “show the way for the rest of the economy by already reducing its own carbon footprint by 20% by 2016". Coremuniti shows a possible way to reduce the impact of computing intensive processes by a P2P network that can induce a significant energy saving (as we do not need to power additional resources but better use *already* powered ones) compared to processes running on a single node or server farm.

This reduction in energy consumption will reduce the pollution emission, characterizing our solution as an environment friendly solution for high performance computing. In order to better understand this beneficial effect, consider that data center energy consumption is about 5% (1,5% in 2007, 3% in 2011, 5% today) of the total electricity consumed and (excluding in-house solutions) the annual EU consumption is about 1600 million tons of oil equivalent (toe). By implementing our network, we hypothesize a reduction of at least 25 million toe per year when reaching the 75% of steady state. We do not report here our early results as these are based on the actual measurements we performed in the controllable environment of our test network. We are working on obtaining more reliable results on a large scale network.

Acknowledgments

The authors thank the Calabria regional government that fully funded the project development and the EU commission that awarded our project with a seal of excellence.

References

[1] Trends in the cost of computing AI Impacts website, https://aiimpacts.org/trends-in-the-cost-of-computing/. (Accessed 22 February 2018).
[2] Nature. Big data. Nature, September 2008.
[3] The Economist. Data, data everywhere. The Economist, Feb 2010.
[4] D. Agrawal, et al., Challenges and opportunities with big data. A community white paper developed by leading researchers across the United States, 2012.
[5] Vinayak R. Borkar, Michael J. Carey, Chen Li, Inside "Big Data Management": Ogres, onions, or parfaits? in: International Conference on Extending Database Technology, 2012, pp. 3–14.
[6] BOINC website, https://boinc.berkeley.edu. (Accessed 22 February 2018).
[7] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, Freely available on the web, 2008.
[8] Dai Peng, Hybrid crowdsourcing platform, United States Patent Application 20150178134, 2015.
[9] Min Yang, Yuanyuan Yang, An efficient hybrid peer-to-peer system for distributed data sharing, IEEE Trans. Comput. 59 (9) (2010) 1158–1171.
[10] Beverly Yang, Hector Garcia-Molina, Comparing hybrid peer-to-peer systems, in: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 561–570.
[11] Nunziato Cassavia, Sergio Flesca, Michele Ianni, Elio Masciari, Giuseppe Papuzzo, Chiara Pulice, A peer to peer approach to efficient high performance computing, in: 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, St. Petersburg, Russia, March 6–8, 2017, 2017, pp. 539–542.
[12] Jeff Howe, Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business, first ed., Crown Publishing Group, New York, NY, USA, 2008.
[13] Tingxin Yan, Matt Marzilli, Ryan Holmes, Deepak Ganesan, Mark Corner, mCrowd: A platform for mobile crowdsourcing, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09, 2009.
[14] Xuan Liu, Meiyu Lu, Beng Chin Ooi, Yanyan Shen, Sai Wu, Meihui Zhang, CDAS: A crowdsourcing data analytics system, Proc. VLDB Endow. 5 (10) (2012) 1040–1051.
[15] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, Reynold Xin, CrowdDB: answering queries with crowdsourcing, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 61–72.
[16] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, Robert Miller, Human-powered sorts and joins, Proc. VLDB Endow. 5 (1) (2011).
[17] Jiannan Wang, Guoliang Li, Tim Kraska, Michael J. Franklin, Jianhua Feng, Leveraging transitive relations for crowdsourced joins, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2013, pp. 229–240.
[18] Jiannan Wang, Tim Kraska, Michael J. Franklin, Jianhua Feng, CrowdER: crowdsourcing entity resolution, Proc. VLDB Endow. 5 (11) (2012).
[19] Daniel Haas, Jason Ansel, Lydia Gu, Adam Marcus, Argonaut: Macrotask crowdsourcing for complex data processing, Proc. VLDB Endow. 8 (12) (2015) 1642–1653.
[20] Chen Jason Zhang, Lei Chen, Yongxin Tong, Zheng Liu, Cleaning uncertain data with a noisy crowd, in: 31st IEEE International Conference on Data Engineering, ICDE, 2015, pp. 6–17.
[21] Stephen Guo, Aditya Parameswaran, Hector Garcia-Molina, So who won?: Dynamic max discovery with the crowd, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, 2012, pp. 385–396.
[22] Anthony Brew, Derek Greene, Pádraig Cunningham, Using crowdsourcing and active learning to track sentiment in online media, in: Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence, 2010, pp. 145–150.
[23] Habibur Rahman, Saravanan Thirumuruganathan, Senjuti Basu Roy, Sihem Amer-Yahia, Gautam Das, Worker skill estimation in team-based tasks, Proc. VLDB Endow. 8 (11) (2015) 1142–1153.
[24] AnHai Doan, Raghu Ramakrishnan, Alon Y. Halevy, Crowdsourcing systems on the world-wide web, Communun. ACM 54 (4) (2011) 86–96.
[25] Aniket Kittur, Ed H. Chi, Bongwon Suh, Crowdsourcing user studies with mechanical turk, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '08, 2008.
[26] Jeffrey Heer, Michael Bostock, Crowdsourcing graphical perception: Using mechanical turk to assess visualization design, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, 2010, pp. 203–212.
[27] Adam Marcus, David Karger, Samuel Madden, Robert Miller, Sewoong Oh, Counting with the crowd, in: PVLDB, 2013, pp. 109–120.
[28] Winter Mason, Siddharth Suri, Conducting behavioral research on amazon's mechanical turk, Behav. Res. Methods 44 (1) (2012) 1–23.
[29] Barzan Mozafari, Purna Sarkar, Michael Franklin, Michael Jordan, Samuel Madden, Scaling up crowd-sourcing to very large datasets: A case for active learning, Proc. VLDB Endow. 8 (2) (2014) 125–136.
[30] Yudian Zheng, Jiannan Wang, Guoliang Li, Reynold Cheng, Jianhua Feng, Qasca: A quality-aware task assignment system for crowdsourcing applications, in: ACM SIGMOD, 2015, pp. 1031–1046.
[31] Peng Cheng, Xiang Lian, Zhao Chen, Rui Fu, Lei Chen, Jinsong Han, Jizhong Zhao, Reliable diversity-based spatial crowdsourcing by moving workers, PVLDB 8 (10) (2015) 1022–1033.
[32] Ting Wu, Lei Chen, Pan Hui, Chen Jason Zhang, Weikai Li, Hear the whole story: Towards the diversity of opinion in crowdsourcing markets, PVLDB 8 (5) (2015) 485–496.
[33] Zhao Chen, Rui Fu, Ziyuan Zhao, Zheng Liu, Leihao Xia, Lei Chen, Peng Cheng, Caleb Chen Cao, Yongxin Tong, Chen Jason Zhang, gMission: A general spatial crowdsourcing platform, PVLDB 7 (13) (2014) 1629–1632.
[34] Senjuti Basu Roy, Ioanna Lykourentzou, Saravanan Thirumuruganathan, Sihem Amer-Yahia, Gautam Das, Task assignment optimization in knowledge-intensive crowdsourcing, VLDB J. 24 (4) (2015) 467–491.
[35] Panagiotis Mavridis, David Gross-Amblard, Zoltán Miklós, Using hierarchical skills for optimized task assignment in knowledge-intensive crowdsourcing, in: Proceedings of the 25th International Conference on World Wide Web, WWW '16, 2016, pp. 843–853.
[36] Ju Fan, Guoliang Li, Beng Chin Ooi, Kian-Lee Tan, Jianhua Feng, iCrowd: An adaptive crowdsourcing framework, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1015–1030.
[37] Habibur Rahman, Senjuti Basu Roy, Saravanan Thirumuruganathan, Sihem Amer-Yahia, Gautam Das, Task assignment optimization in collaborative crowdsourcing, in: 2015 IEEE International Conference on Data Mining, ICDM, 2015, pp. 949–954.
[38] Chen Jason Zhang, Lei Chen, Yongxin Tong, MaC: A probabilistic framework for query answering with machine-crowd collaboration, in: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM '14, 2014, pp. 11–20.
[39] Liyue Zhao, Gita Sukthankar, Rahul Sukthankar, Robust active learning using crowdsourced annotations for activity recognition, in: Proceedings of the 11th AAAI Conference on Human Computation, AAAIWS'11-11, 2011, pp. 74–79.
[40] Niroshinie Fernando, Seng Wai Loke, J. Wenny Rahayu, Mobile cloud computing: A survey, Future Gener. Comput. Syst. 29 (1) (2013) 84–106.
[41] Eugene E. Marinelli, Hyrax: Cloud Computing on Mobile Devices Using MapReduce (Masters thesis), Carnegie Mellon University, 2009.
[42] Keke Gai, Meikang Qiu, Hui Zhao, Lixin Tao, Ziliang Zong, Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing, J. Netw. Comput. Appl. 59 (2016) 46–54.
[43] Keke Gai, Meikang Qiu, Hui Zhao, Energy-aware task assignment for mobile cyber-enabled applications in heterogeneous cloud computing, J. Parallel Distrib. Comput. 111 (2018) 126–135.
[44] Shuang Ding, Xin He, Jicheng Wang, Multiobjective optimization model for service node selection based on a tradeoff between quality of service and resource consumption in mobile crowd sensing, IEEE Internet Things J. 4 (1) (2017) 258–268.
[45] Keke Gai, Meikang Qiu, Zhong Ming, Hui Zhao, Longfei Qiu, Spoofing-jamming attack strategy using optimal power distributions in wireless smart grid networks, IEEE Trans. Smart Grid 8 (5) (2017) 2431–2439.
[46] Keke Gai, Longfei Qiu, Min Chen, Hui Zhao, Meikang Qiu, SA-EAST: Security-Aware Efficient Data Transmission for ITS in Mobile Heterogeneous Cloud Computing, ACM Trans. Embedded Comput. Syst. 16 (2) (2017) 60:1–60:22.
[47] David P. Anderson, Boinc: A system for public-resource computing and storage, in: 5th IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10.

[48] Atsuyuki Morishima, Sihem Amer-Yahia, Senjuti Basu Roy, Crowd4U: An initiative for constructing an open academic crowdsourcing network, in: Proceedings of the Seconf AAAI Conference on Human Computation and Crowdsourcing, HCOMP, 2014.
[49] Rashmi Bhatia, Grid computing and security issues, Int. J. Sci. Res. Publ. 3 (2013) August 2013 Edition.
[50] Mohamed Firdhous, Implementation of security in distributed systems - A comparative study, 2012, CoRR abs/1211.2032.
[51] John Nickolls, Ian Buck, Michael Garland, Kevin Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008).
[52] Mark Harris, Many-core GPU computing with NVIDIA CUDA, in: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS '08, 2008.
[53] Andrew S. Glassner, An Introduction to Ray Tracing, Elsevier, 1989.
[54] Blender website, https://www.blender.org. (Accessed 21 May 2016).
[55] R.F. Witzel, R.W. Burnham, J.W. Onley, Threshold and suprathreshold perceptual color differences, JOSA 63 (5) (1973) 615–625.
[56] M. Payer, T. Hartmann, T.R. Gross, Safe loading - a foundation for secure execution of untrusted programs, in: 2012 IEEE Symposium on Security and Privacy, May 2012, pp. 18–32.
[57] Junjie Li, Sanjay Ranka, Sartaj Sahni, Strassen's matrix multiplication on gpus, in: Parallel and Distributed Systems, ICPADS, 2011 IEEE 17th International Conference on, IEEE, 2011, pp. 157–164.
[58] University of florida sparse matrix collection, https://www.cise.ufl.edu/research/sparse/matrices/. (Accessed 22 February 2018).
[59] RenderStreet website, https://render.st/. (Accessed 22 February 2018).
[60] RebusFarm website, https://it.rebusfarm.net/en/. (Accessed 22 February 2018).
[61] Microsoft Azure website, https://azure.microsoft.com/en-us/. (Accessed 22 February 2018).
[62] Bitcoin-Calculator website, https://alloscomp.com/bitcoin/calculator. (Accessed 22 February 2018).
[63] Litecoin-Calculator website, https://www.litecoinpool.org/calc. (Accessed 22 February 2018).
[64] Ethereum-calculator website, https://badmofo.github.io/ethereum-mining-calculator. (Accessed 22 February 2018).
[65] Xiaohang Zhang, Guoliang Li, Jianhua Feng, Crowdsourced top-k algorithms: An experimental evaluation, PVLDB 9 (8) (2016) 612–623.

Nunziato Cassavia received his Master Degree in Computer Engineering from the University of Calabria in 2012. Currently he is a research fellow and Ph.D. student at University of Calabria in ICT. His research interest includes Big Data, Distributed Computing System, Data Warehouse and Relational and NoSQL Databases.

Sergio Flesca is full professor at University of Calabria. He received a Ph.D. degree in Computer Science from University of Calabria. His research interests include databases, web and semi-structured data management, information extraction, inconsistent data management, approximate query answering, and argumentation.

Michele Ianni received his Ph.D. in Information and Communication Technologies from the University of Calabria, Italy, in 2018. Currently, he is a research fellow at University of Calabria, Italy. His main research interests include cyber security, cryptography, software vulnerability exploitation, malware analysis and trusted computing.

Elio Masciari is currently senior researcher at the Institute for High Performance Computing and Networks (ICAR-CNR) of the National Research Council of Italy. His research interests include Database Management, Semistructured Data and Big Data. He has been advisor of several master theses at the University of Calabria and at University Magna Graecia in Catanzaro. He was advisor of PhD thesis in computer engineering at University of Calabria. He has served as a member of the program committee of several international conferences. He served as a reviewer for several scientific journals of international relevance. He is author of more than 100 publications on journals and both national and international conferences. He also holds "Abilitazione Scientifica Nazionale" for Full Professor role.

Chiara Pulice received her Ph.D. in Computer and Systems Engineering from the University of Calabria, Italy, in 2015. Currently, she is a research fellow at Dartmouth College, Hanover, NH, USA. Her main research interests include data integration, inconsistent databases, social network analysis as well as data mining and machine learning, particularly for counterterrorism.