# Solving Critical Simulation Problems Under Emergency Conditions Using Volunteer Computing

Darkhan Akhmed-Zaki[1], Bolatzhan Kumalakov[1] and Grzegorz Dobrowolski[2]
[1]*Faculty of Mechanics and Mathematics, al-Farabi Kazakh National University, Almaty, Kazakhstan*
[2]*Department of Computer Science, AGH University of Science and Technology, Cracow, Poland*

Keywords: Volunteer Computing, Multi-Agent Systems, Simulation.

Abstract: Paper presents a case study where a multi-agent system based volunteer computing platform is used to solve critical simulation problem under emergency conditions. As a result, we extend working platform functionality, better utilize agent paradigm and get valuable data on prototype performance. Main resulting novelty is an advancement in system architecture.

## 1 INTRODUCTION

Computer scientists developed numerous methods and technologies for solving large scale computational problems. They vary from highly dedicated pulls of servers, such as cloud or grid infrastructures (see (Amazon, 2013), (Google, 2013)), to volunteer resource sharing platforms for Internet users ((Ries, 2012), (Balicki et al., 2012)). In every individual case technology choice depends on computational features of the problem, user requirements, available hardware and networking resources.

In business and scientific research, for instance, it is common to perform resource intensive computing using scalable grid and cloud solutions. Industry level service providers, such as (Amazon, 2013) or (Google, 2013), guarantee system dependability and platform uptime higher than 99.9%. They are accessible from any location at any time via standard Internet connection, significantly reduce ownership costs and decrease need in IT staff. As a result, they perfectly satisfy everyday need in intensive computing due to comparatively cheep virtual solutions and good level of customer service.

Yet, consider an emergency when computing is ought to serve as a crisis management tool that provides necessary information to the stakeholders. Let one also assume that an access to the cloud or the grid is lost because of emergency cause (natural disaster). In this case the result might no longer be needed by the time restoring connection is complete. Thus, it is feasible to employ a software platform that makes use of personal devices provided by public servants, people who are being threatened by emergency risks or anyone else on site.

In (Ahmed-Zaki et al., 2013) we proposed an agent based volunteer MapReduce architecture for resource intensive computing. The final goal is to extend it into a platform, that is capable of pulling together a volunteer ad-hoc mobile cloud to solve critical problems under emergency conditions (such as described above). In order to achieve this target we solve several case studies by introducing necessary design extensions, implementing them in the prototype (building it further) and evaluating its performance. In this paper we present the first case, where the platform serves as a tool to solve a numeric simulation problem as part of the overall solution.

Reminder of the paper is structured as follows. Section 2 introduces the case study. Section 3 gives an overview of *volunteer computing platforms landscape* and positions our platform on it. Section 4 contains a detailed description of the extended platform model and architecture concept, followed by implementation details in Section 5. Finally, Sections 6 and 7 present computational experiment, its results and conclusions.

## 2 CASE DEFINITION

Let us consider the real world disaster event. Fukushima nuclear power station in Japan was damaged by an explosion following an earthquake and tsunami in March 2011. Explosion released radia-

Akhmed-Zaki D., Kumalakov B. and Dobrowolski G..
Solving Critical Simulation Problems Under Emergency Conditions Using Volunteer Computing.
DOI: 10.5220/0005040401700178
In *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2014), pages 170-178
ISBN: 978-989-758-038-3

tion that spread over heavily populated neighbor region and, as a result, military forces had to urgently evacuate people from the local area.

Technological disasters of that kind are characterized by uncontrolled and extreme environment where people lifes are threatened by dangerous pollution that is spreading over geographical area. The pollution may be of different physical and chemical nature (i.e. gas, fluids, radiation, etc.), but its convection dynamics is simulated using methods described in 6.1.

Never mind the polluting element, we are interested in the means of assisting specialized rescue teams that try to evacuate people from the disaster site. (Erickson, 1999) notes that "in the absence of data and information, emergency response is simply well-intended guesswork that will most likely result in significant loss of human life" . In this case information is a forecast of how the pollution spreads. Retrieving it requires the system that collects and processes relevant data at a real time and supplies output on site and in acceptable format (Carver and Turoff, 2007), (Turoff, 2002).

Designing such a solution is an ambitious task. It requires data collection and storage planing, communication management, etc. Moreover, defined disasters domain phenomena are usually described using differential equations (see (Adam, 2003)) and are solved by applying complex numeric methods. As a result, we assume that all the data is collected in prior and is available when needed. We particularly concentrate on the design of a simulation software platform that: (a) may be used in an absence of a connection to the industrial scale cluster system; (b) makes use of voluntarily provided personal computer devices of the people at sight; (c) solves numeric problems; (d) builds on the platform from (Ahmed-Zaki et al., 2013).

## 3 BACKGROUND AND RELATED WORK

Throughout following text an *agent* or a *multi-agent system* will imply using the intelligent agent concept, as defined in (Wooldridge and Jennings, 1995) and (Wooldridge, 2001), unless explicitly stated otherwise in the main body of the text.

### 3.1 Non-agent Based Volunteer Solutions

First stream of solutions is based on different Linux kernel extensions that boot machines in a volunteer cluster mode. Machines either join a running cluster as processing nodes, or initiate formation of a new cluster. Such infrastructures may be regarded as volunteer because machine owner voluntarily donates its resources by booting specially designed operating system.

Open source PelicanHPC GNU Linux, Kerrigehed and OpenSSI are instances of such systems. They make use of OpenMP and MPI technologies in different variations, and do not facilitate run time cluster reconfiguration. Their main disadvantage is the need to install specialized operating system on every node to launch the cluster, and the need to have a separate boot driver for every computer architecture to launch cluster nodes.

Second stream of solutions requires installing a software application that facilitates job execution. As a result, the ability to share resources does not depend on the operating system. Let us consider following two examples.

BOINC (Ries, 2012) is a working volunteer computing platform that functions over Internet. Central server divides a job into tasks and posts them to the server. Volunteers request tasks, execute them and return their results in exchange for a reward (estimated in credits). Every task is released to two volunteers. When server receives both results they are compared and, if answers match, both machines are awarded credits. To join the platform user has to install a lightweight application that: connects to the task server, downloads code and input data, performs processing, returns the result and claims credits.

Comcute system (Balicki et al., 2012) is another volunteer platform. It requires volunteers to install Java applet to their Internet browser in order to carry out job execution. The Java applet requests the job from *Data generator*, downloads executable code and input data from *distributor servers*, runs the code in browser window and returns the result back for verification. Unlike BOINC, Comcute does not make use of monetary reward to attract the resources, but is supported by volunteers who are interested in sharing the final result. For instance, groups of people who are in danger and their well being depends on the computational result.

Despite the fact that application based software overcomes the shortcomings of the first stream solutions, they rely on central server to initiate task execution and collect the results. It is a single point of possible failure, i.e. the bottleneck.

### 3.2 Agent Based Volunteer Solutions

Agent based platforms that solve large scale computational problems are collectively called "agent grids" (Manola and Thompson, 1999). In the grid an agent controls computer resources, manipulates data, manages code execution and handles its result; or it provides an access to its peripheral devices for collective use. A set of such agents constitutes a multi-agent system that solves problems using autonomy and collaboration principles (i.e. individual agents do not execute jobs by themselves, but do so in cooperation with others).

Agent grid typically has two levels: application and functional (Manola and Thompson, 1999). Application level is a set of requirements that defines platform characteristics, such as scalability and adaptability. Functional level, on the other hand, consists of requirements for the computing environment and its components (i.e. how available resources are linked to each other).

In (Gangeshwari et al., 2012) authors organize multiple agent supervised data centers into a hyper cubic grid structure. Every data center has preinstalled execution software, while agents optimize inter-data center workload distribution and communication channels load when distributing jobs. However, on the level of a single data center nodes are controlled by pre-defined algorithms, thus, are not autonomous.

Another platform of this kind is presented in (Marozzo et al., 2011). Here, every node is managed by an agent, who are assigned *master* or *slave* role. Master nodes cooperate to organize and manage job execution. One of them acts as user interface, whilst others monitor its performance and voluntarily take on control if it fails. Slaves, on the other hand, get commands from master nodes, execute them and return the result. Node autonomy, in this case, is utilized at the master level, whilst slaves are directly managed. In (Dang et al., 2012) authors present similar solution that extends Gnutella protocol to facilitate peer-to-peer execution of jobs. In particular software components called *super agents* organize themselves into groups that cooperate to facilitate the execution. Task initiator becomes master node and other peers become slaves.

AGrIP (Luo and Shi, 2007), on the other hand, is a FIPA compliant platform based on MAGE project. It satisfies two main requirements: creates and manages a pool of computing machines, and provides standardized build in grid services. In order to do so AGrIP creates agent roles that target particular functionality on both application and functional levels.

We develop a platform that is also FIPA compliant and extends Jade framework (Bellifemine et al., 2007). Following rules apply with respect to agent autonomy:

1. no node has direct control over others, but may indirectly influence execution flow. We refer to this mechanism as supervision and it includes *reducer-mapper* and *supervisor-reducer* relationships as part of system architecture.
2. agents store the data in a distributed fashion so, that there is no central storage that would create a bottle neck.
3. supervision includes state duplication on peer devices to allow restarting processes at different stages and not from the start if needed.
4. agents may independently change roles and/or take numerous roles (e.g. reducer and supervisor) at the same time.

Finally, we narrow down agent-grid definition by merging it with the notion of "scale out" solution (Lin and Dyer, 2010). Scale out is an architecture type that offers cluster computing with machines connected to the network. The difference is - an agent grid does not necessarily imply changes in execution efficiency when number of agents changes; whilst, scale out does, but lacks machine autonomy (freedom to self-organize at execution time). Thus, agent-grid should hold following property: an increase in number of agents should result in increase in computing speedup and visa versa, while all machines are autonomous at all levels.

## 4 PLATFORM MODEL AND ARCHITECTURE

First, we extend formal description of the volunteer job execution in the light of the case study requirements in 4.1. Then we introduce an algorithm for the ad hoc mobile cloud composition in 4.2.

### 4.1 Workload Distribution Function

In order to solve the problem we construct an iterative scheme that employs domain decomposition method from (Barry et al., 2004) (figure 1). General computing domain is divided into arbitary number of subdomains (three in the figure) to be dostributed between nodes and computed in parallel.

We denote entire computation by $J$ and its step by $k$, such as $J = \{k_1, k_2, \ldots, k_n\}$. Here, $k$ represents a sub-domain to be computed. All steps are performed

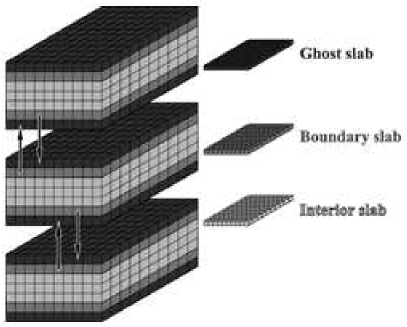

Figure 1: Figure visulizes how general compuing domain is divided into several sub-domains to be distributed between nodes for parallel execution.

by a set of computing nodes $A = \{a_1, a_2, \ldots, a_m\}$. If step $k_n$ may be performed by node $a_m$, we denote it as a mapping function $k_n \rightarrow a_m$.

Let us use *price* to reflect device workload and be computed as a derivative of available resources, device workload and basic price (equation 1). Every successful mapping $k_i \rightarrow a_j (1 \le i \le n, 1 \le j \le m)$ gets computing price $p_{ijk}$ assigned by an accepting computing node.

$$p_{ijk} = f(\omega_{ijk}, p_b, b_l) \quad (1)$$

Here, $p_b$ denotes basic resource price, which is set by device owner; $b_l$ denotes battery load; and $\omega_{ijk}$ denotes resources availability at the time, when step $k$ arrives. $\omega_{ijk}$ has following descrete values:

$$\omega_{ijk} = \begin{cases} 1 & \text{device free, can map and reduce} \\ 0.6 & \text{device buisy, can map and reduce} \\ 0.3 & \text{device can map only} \\ 0 & \text{otherwise} \end{cases}$$

Computed price for different mappings may not be the same $p_{ijk} \neq p_{ljk}$, where $i \neq l$ and $1 \le i, l \le m$. If they are equal, the conflict is resolved on the *first come first served* basis.

After price is computed tuple $((\rho(\omega_{ijk}), p_{ijk})$ is returned to initiator node, where $p_{ijk}$ is computer price and $\rho(\omega_{ijk})$ is determined as follows:

$$\rho(\omega_{ijk}) = \begin{cases} 1 & \omega_{ijk} > 0, \text{ want to supply services} \\ 0 & \text{otherwise} \end{cases}$$

Then, issuer returns result of function $\varphi(p_{ijk})$, which determines executor node. Only in this case it takes eligibility traces into account:

$$\varphi(p_{ijk}) = \begin{cases} 1 & \rho(\omega_{ijk}) = 1,\ p_{ijk} \rightarrow min,\ q_{nk} \rightarrow max \\ 0 & \text{otherwise} \end{cases}$$

Here, $q_{nk}$ is the locally stored value of the eligibility trace. Maximizing it in this case means choosing the highest available value, thus, minimize time spent to perform the task.

Client balance $cb$ represents the amount of money user can spend on services.

Using values stated above distribution function is formulated as follows:

$$\min_p \sum_{i=1}^{n} \rho(\omega_{ijk})\varphi(p_{ijk}) \quad (2)$$

Subject to:

$$\sum_{i=1}^{n} \rho(\omega_{ijk})\varphi(p_{ijk}) \le cb \quad (3)$$

$$\sum_{i=1}^{n} \rho(\omega_{ijk})\varphi(p_{ijk}) > 0 \quad (4)$$

Objective function (2) minimizes overall cost of performing the job by choosing lowest price at each step. Constraints ensure that overall solution cost is always lower than client balance (3) and at least one path of job execution exists (4).

## 4.2 Composite Infrastructure Model

Figure 2 pictures computer machines located on a disaster site that are geographically or technologically grouped into pulls. If one pull initiates a resource demanding job and can not satisfy the power demands, it seeks a resource sharing alliance with others. If such an alliance is achieved, resources donor pull becomes "service provider" and its nodes self-organize to execute part of the overall job.

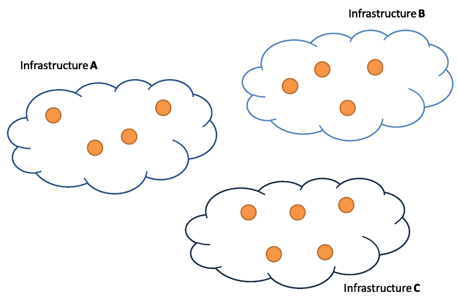

Figure 2: Several pulls of resources that are locatd at the disaster site.

From the alliance seeker point of view service provider choice is not independent, because it has to merge every time the simulation tasks is launched. As a result, agents must consider future implications and immediate consequences of each decision. Moreover, no agent or group of agents may have full information about potential service providers. Thus, they must learn about them through experiencing the consequences of cooperations.

Core of the solution algorithm is a combination of Sarsa learning algorithm and Boltzmann selection strategy (Sutton and Barto, 1998). It was argued to be optimal for solving small scale problems (Coggan, 2004) where learning system exploits best solution and explores new options at a reasonable rate. Reasonable rate in this case depends on the problem and should be determined in every separate case.

In order to solve our case, following changes were introduced into the basic algorithm and the model.

First, system goal is to minimize execution time and price. While node choice is based on an offer price, service providers' execution time ratings are to be learned through numerous iterations. Thus, agents should have two modes of learning: for cases when service providers meet time expectations, and for cases when they fail. Sarsa learning algorithm makes use of a parameter called the *learning rate*. The grater it is, the more agent tents to explore unknown options. In other words when service providers successfully meet expectation it should have lower value and visa versa.

Second, classic Sarsa learning algorithm updates only agent choice at time $t$, thus, leaving out decisions at time $t-1$ and earlier. In order to implement interdependencies of choices we apply the eligibility trace (Sutton and Barto, 1998). For details lets consider Fig. 3.

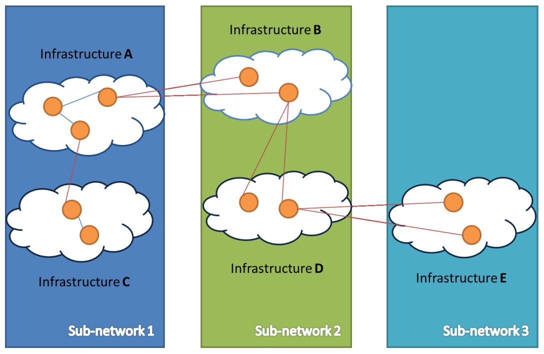

Figure 3: Domains of eligibility traces distribution. Platform learns through updating eligibility trace on every iteration.

Figure 3 presents a composite infrastructure that consists of five pulls that are located in different subnetworks. Accessability is limited and following are connectivity rules: C can connect only to A; A can connect to B and C; B is able to connect to A and D; D can connect to B and E; and E can only connect to D.

Selection strategy is executed at infrastructure connection points (agents that can reach members of other pulls) at a run time. As a result, it only takes into account service providers known to the connecting agent, but is not aware of neighbor options. For instance, if Infrastructure A chooses B, it does not know that B can not connect to the other pull. Potentially execution could take considerably longer time to finish and platform fails to meet the requirements.

Eligibility trace is used to update all the intermediate reward values with regards to peer infrastructure availability of the next service provider. Updated learning algorithm is presented in algorithm 1.

**Algorithm 1:** Modified Sarsa Learning Algorithm.

```
Initialize $Q(s,a)$ arbitrarily;
for each episode do
    Initialize $s$;
    Choose $a$ from $s$ using policy derived from $Q$;
    while $s$ is not terminated do
        if time exceeds expected then
            $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_{high}\phi_t e_t(s,a)$ ;
        else
            $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_{low}\phi_t e_t(s,a)$ ;
        end
        $s \leftarrow s'$, $a \leftarrow a'$;
    end
end
```

where,

$$\phi_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & : s = s_t, a = a_t \\ \gamma\lambda e_{t-1}(s,a) & : otherwise \end{cases}$$

and, $s$ represents environment state, $a$ is an agent action, $Q$ is a state-action value, $\alpha$ is the learning rate and $\gamma$ is the discount factor.

Boltzmann selection strategy determines how $Q(s,a)$ value is used to make a decision. It takes into account relative values of $Q(s,a)$ and applies probability that depends on how one state-action value is compared to the others. Probability of choosing action $a$ at the state $s$ is computed using equation 5. $T$ is referred to as *temperature* and determines exploration rate (the higher $T$ is, the higher is the exploration rate).

$$p = \frac{e^{\frac{Q(s,a) - max_b Q(s,b)}{T}}}{\sum_a e^{\frac{Q(s,a) - max_b Q(s,b}{T}}} \qquad (5)$$

## 4.3 Architectural Solution

Figure 4 presents the conceptual scheme of the system.

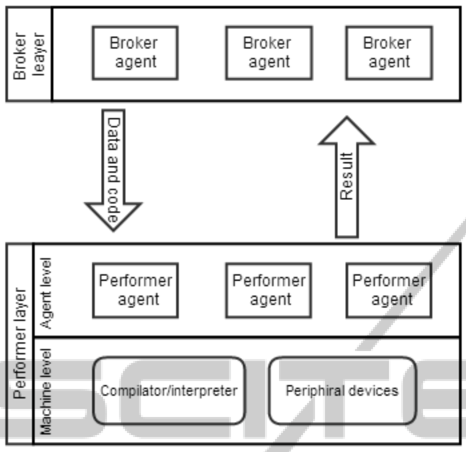

Figure 4: The system consists of broker and perofmer layers. When brokers pas the task to performer layer, performer agents selforganize and pass the instructions to their own compilator for execution, or to the pheriphiral device.

Broker layer is a collective term that represents all broker agents active at the given point in time. Performer layer, on the other hand, consists of two levels: agent level and machine level. Agent level refers to potential job performers, who are active at the same given point in time. Machine level is resources, available to performer agents.

Every agent has direct access only to its device resources. However, on the diagram they are imaged as a common pool because eventually they are accessible to all performers through negotiation.

All system participants are able to take on roles of broker or/and performer, depending on the situation. Figure 5 presents high level concept of the computing node architecture that includes three main components: broker agent, performer agent and interpreter/compilor.

Broker agent is created by interpreter/compiler when user wants to launch a computing task. It is responsible for broadcasting information messages, finding reducer nodes and handling organizational communication at execution time. Its life cycle is limited to job execution and when final result is received by user interface it terminates.

Performer agent runs all the time and listens to job offers. When execution offer is received, it evaluates node's current state and makes a decision whether to form an offer message or to do nothing. If an offer is issued to the requesting node, performer is responsible for handling upcoming operations. That is finding supervisors, passing executable code to the interpreter/compiler, retrieving execution results and sending them to the destination.

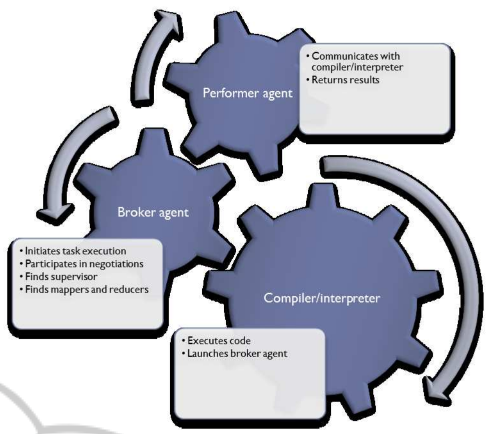
Performer agent
• Communicates with compiler/interpreter
• Returns results
Broker agent
• Initiates task execution
• Participates in negotiations
• Finds supervisor
• Finds mappers and reducers
Compiler/interpreter
• Executes code
• Launches broker agent

Figure 5: Performer agent listens to execution calls and communicates with the light weight compilator/interpriter interface. When machine launches the task compilator/interpriter interface launches the vroker agent that initiates the process.

Interpreter/compilor gets job logic (code) and is used to compile/run user applications that launch the broker agent.

Unlike in previous platform design, performer agents gets a Linked List type data structure that stores accessible infrastructures list and corresponding state-action values. Figure 6 presents formalized job execution and state-action values update process.

## 5 SYSTEM IMPLEMENTATION

We implemented system prototype using JADE. Executable code is encapsulated into ACLMessage object and passed between agents. Code is executed on Java Virtual Machine using Clojure-1.4 (PC and server machines) and JSceme-7.2 (mobile devices).

Agent initialization includes publishing two advertisements: first, supervision services, second, MapReduce services. There is no predefined role, because it depends on self-evaluation at execution time. Supervisor does not copy reducer state directly, but knows about changes by listening to duplicated messages, sent to the reducer. In other words it updates state record when receives mapper and leaf reducer message duplicates.

In order to describe job submission and failure recovery mechanisms we take example scenario that corresponds to the algorithm described in Figure 6.

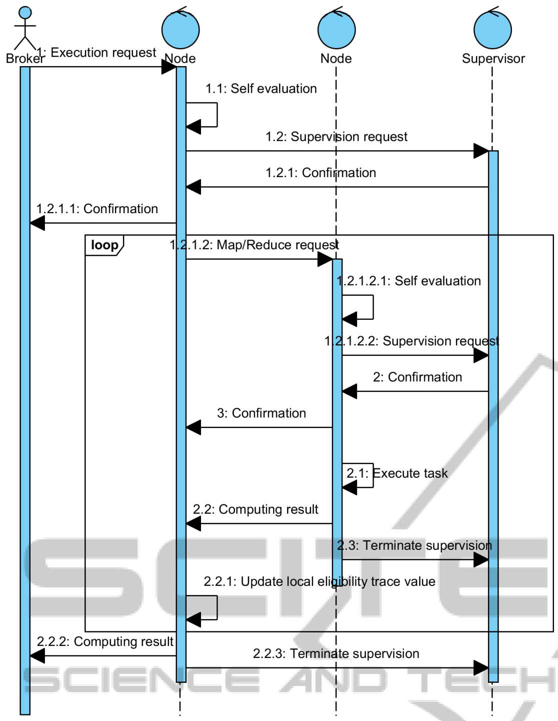

Figure 6: UML Sequence Diagram formalizes job execution and state-action values update sequence.

## 6 EXPERIMENT AND EVALUATION

### 6.1 Problem Definition

We adopt a large scale physical-technological phenomena that employs well known numeric methods (as would estimating radiation dissemination) to solve it.

Problem is specified as follows: consider equations of water (6) and oil (7) balance, as well as Darcy's law (8).

$$m\frac{\partial s_1}{\partial t} + div(\overrightarrow{u_1}) = q_1 \tag{6}$$

$$m\frac{\partial s_2}{\partial t} + div(\overrightarrow{u_2}) = q_2 \tag{7}$$

$$\overrightarrow{u_i} = -\frac{K_0 f_i}{\mu_i}\nabla p; (i = 1,2) \tag{8}$$

$$s_1 + s_2 = 1 \tag{9}$$

here, $m$ is porosity of media, $K_0$ is absolute permeability of the media, $\mu_i$ is liquid viscosity and $f_i$ is relative phase permeability. Index 1 corresponds to the water phase, and 2 to the oil phase.

Oil deposit is represented as a 3D finite domain $\Omega$ with smooth boundary $\partial\Omega$. We are required to compute pressure $p$ satisfying following equations and relations:

$$\frac{\partial}{\partial x}\left(K_x\frac{\partial P}{\partial x}\right) + \frac{\partial}{\partial y}\left(K_y\frac{\partial P}{\partial y}\right) + \frac{\partial}{\partial z}\left(K_z\frac{\partial P}{\partial z}\right) = -f(x,y,z) \tag{10}$$

where,

$$K_x = \frac{K_{0_x} f_1}{\mu_1} + \frac{K_{0_x} f_2}{\mu_2}; K_y = \frac{K_{0_y} f_1}{\mu_1} + \frac{K_{0_y} f_2}{\mu_2}; K_z = \frac{K_{0_z} f_1}{\mu_1} + \frac{K_{0_z} f_2}{\mu_2}; \tag{11}$$

$$div(\overrightarrow{u_1}) + div(\overrightarrow{u_2}) = q_1 + q_2 = f$$

Initial conditions:

$$s_1|_{t=0} = s^0(x) \tag{12}$$

Boundary conditions:

$$(p, s_1) = (p_0, s_{10}), (x,t) \in \Sigma^1 = \partial\Omega^1 \times [0,T] \tag{13}$$

Problem (10)-(13) is solved using Jacobi method.

### 6.2 Experiment Design

In order to evaluate platform performance we set up following infrastructure: 13 PCs with 3rd generation Intel Core i5 processors and 4 Gb of RAM; 3 HP server machines with Intel Xeon processors and 8, 16 and 16 Gb RAM; HP visualization cluster with 94 virtual machines; and 18 cell phones that run Android OS.

Machines are organized into three sub-networks and five architectures as illustrated in Fig. 2 (connectivity rules correspond to those in the figure). Infrastructure identity is build into the agent at initiation time in a form of a private variable. It can not be changed at execution time and determines agent communications provided that all sub-networks contain a distributed Jade platform to enable message passing and communications.

Every infrastructure consists of at least 2 PCS, 1 cell phone, 18 Virtual hosts and some of them have additional 1 HP Server machine. For installation and configuration purposes all nodes got the same software installed and tested by running sample tasks.

In order to perform the testing we build following modification: performer agents receives a resource call, accepts it and with probability of 0.3 does not proceed to execution. This way we simulate possible channel failures. If such failure accrues, supervisor re-launches execution, which results in a time loss.

### 6.3 Experiment Results

Testing consisted of numerous sets of the job runs with following input sizes: 120, 240, 360, 480 and 720 points (input size is the number of points per dimension of the hypercube structure). Every set of runs had different number of sub-infrastructures avaliable. Figure 7 presents prototype testing results.

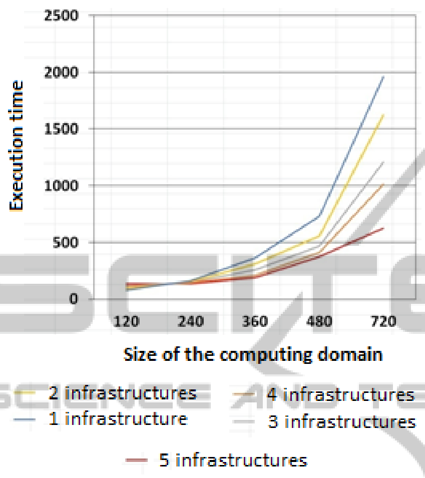

Figure 7: Figure presents prototype testing results for 1-6 sub-infrastructure configurations.

First we did get the expected increase in system performance with the increase in number of available sub-infrastructures. In particular increasing number of sub-infrastructures from one to five decreases computing time three times. Of course this coefficient depends on the task and the sub-infrastructures' connectivity because of the changing merger order and communication channels etc.

After this result was obtained, we compared platform performance to the MPI implementation of the same task. We run MPI code only using 720x720x720 hypercube using 8-core Intel Xeon Server machine with 24 Gb of RAM. We do realize that comparison of the 5 machine distributed system prototype to a 8-core server machine is not an appropriate academic experiment, but it does provide some impression on the architecture performance. So after comparing testing data we estimated that our platform gives on average 500-times slower performance on given input size. Nonetheless, it is subject to the scale (big data, number of iterations) and prototype evolution.

## 7 CONCLUSION

Paper presents our first expansion of the platform capabilities through solving critical simulation case study. In particular, we employ agent learning techniques (modified Sarsa algorithm and Boltzmann selection strategy) to form an ad hoc mobile cloud that solves a simple simulation problem. Experimental results do confirm execution speed up when number of available resources increase, but it is not adequately comparable to the classical MPI implementation yet. On the other hand, we do not claim to design a multi-agent system that could compete with well-established parallel execution technologies, but propose a tool that is capable of solving the defined task in an absence of an offsite cluster system. Thus, future work might involve some performance considerations, but not in the light of enchasing parallel execution efficiency.

Future work is going to concentrate on further development of the platform as a crisis management tool through utilizing agents' intellectual features. In particular, agents may assist in planning and execution of search and rescue operations at different stages.

## ACKNOWLEDGEMENTS

Research is funded under Kazakhstan government scientific grant Organizing distributed and cloud computation to solve resource demanding tasks.

## REFERENCES

Adam, J. (2003). *Mathematics in Nature: Modeling Patterns in the Natural World*. Princeton University Press.

Ahmed-Zaki, D., Dobrowolski, G., and Kumalakov, B. (2013). Peer-to-peer mapreduce platform. In *ICAART (2)*, pages 565–570.

Amazon (2013). Amazon elastic compute cloud (amazon ec2).

Balicki, J., Krawczyk, H., and Nawarecki, E., editors (2012). *Grid and Volunteer Computing*. Gdansk University of Technology.

Barry, F., Petter, E., and William, G. (2004). *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, NJ.

Carver, L. and Turoff, M. (2007). Human-computer interaction: the human and computer as a team in emergency management information systems. *Commun. ACM*, 50(3):33–38.

Coggan, M. (2004). Exploration and exploitation in reinforcement learning. Technical Report CRA-W DMP, McGill University, 845 Sherbrooke Street West, Montreal, Quebec, Canada H3A 0G4.

Dang, H. T., Tran, H. M., Vu, P. N., and Nguyen, A. T. (2012). Applying mapreduce framework to peer-to-peer computing applications. In Nguyen, N. T., Hoang, K., and Jedrzejowicz, P., editors, *ICCCI (2)*, volume 7654 of *Lecture Notes in Computer Science*, pages 69–78. Springer.

Erickson, P. A. (1999). *Emergency response planning for corporate and municipal managers*. Academic Press, San Diego.

Gangeshwari, R., Janani, S., Malathy, K., and Miriam, D. D. H. (2012). Hpcloud: A novel fault tolerant architectural model for hierarchical mapreduce. In *ICRTIT 2012*, pages 179–184. IEEE Computer Society.

Google (2013). Google cloud platform.

Lin, J. and Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Luo, J. and Shi, Z. (2007). Distributed system integration in agent grid collaborative environment. In *Integration Technology. ICIT 07. IEEE International Conference*, pages 373–378. IEEE Computer Society.

Manola, F. and Thompson, C. (1999). Characterizing the agent grid. Technical Report F30602-98-C-0159, Object Services and Consulting, Inc.

Marozzo, F., Talia, D., and Trunfio, P. (2011). A framework for managing mapreduce applications in dynamic distributed environments. In Cotronis, Y., Danelutto, M., and Papadopoulos, G. A., editors, *PDP*, pages 149–158. IEEE Computer Society.

Ries, C. (2012). *BOINC - Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing*. Springer-Verlag, Berlin Heidelberg.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA.

Turoff, M. (2002). Past and future emergency response information systems. *Commun. ACM*, 45(4):29–32.

Wooldridge, M. and Jennings, N. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152.

Wooldridge, M. J. (2001). *Multi-agent systems : an introduction*. Wiley, Chichester. GBA1-Z6596 Michael Woolridge.