9. Genetic Positioning of Fire Stations Utilizing Grid-computing Platform

Jerzy Balicki, Waldemar Korłub, Henryk Krawczyk

Gdansk University of Technology

Faculty of Electronics, Telecommunications and Informatics

Computer System Architecture Department

e-mail: balicki@eti.pg.gda.pl, stawrul@gmail.com, hkrawk@eti.pg.gda.pl

Abstract

A chapter presents a model for determining near-optimal locations of fire stations based on topography of a given area and location of forests, rivers, lakes and other elements of the site. The model is based on principals of genetic algorithms and utilizes the power of the grid to distribute and execute in parallel most performance-demanding computations involved in the algorithm.

Keywords: *grid computing, parallel programming, volunteer computing, genetic algorithm*

9.1. Fire spreading simulation

The starting point for choosing locations of fire stations is to understand and possibly simulate the way spreading fire behaves on a given terrain. Topography of the land, placement of forests, rivers and other elements of the site all need to be taken into account to perform an accurate simulation.

The model used for implementation is based on the spatially explicit representation where the whole terrain is divided into square-shaped fields [1]. Each field of the map is described by [2]:

- its location – e.g. a pair (x, y) representing coordinates in two-dimension space,
- a set of variables representing field's state – e.g. calorific value, fire intensity value, altitude,
- a finite set of neighborhood fields – the neighborhood relation is defined as Moore neighborhood (shown in Fig. 9.1 – gray fields constitute the neighborhood of black field) and comprises eight fields surrounding the central one,

- a transition function which calculates new state of the field as a function of its present state, state of its neighbors and time interval between present and to-be-calculated state.

The flow of time is simulated as a run of discrete time deltas. During each time step, state of a field can change according to the transition function. The simulation is in turn performed in subsequent iterations – with each iteration representing bygone time delta and dependent on the state from the previous step. During every iteration of the simulation the transition function is called for all fields to calculate their new state.

Each individual field can be in one of three possible states: inactive (before ignition), active (under fire) or burned out. Transition from inactive to active state requires a specific level of fire intensity spread from neighbor fields – its an ignition threshold. It helps to better approximate the actual behavior of fire spreading in reality.

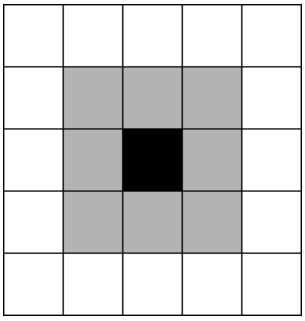


Fig. 9.1. Moore neighborhood [2]

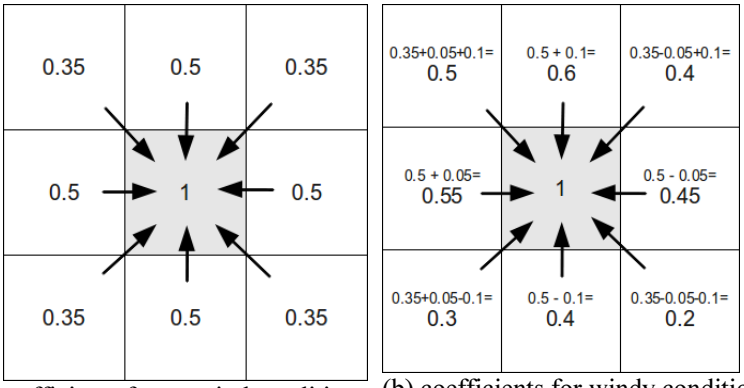
The parameter of calorific value plays an important role during the simulation. Firstly, it determines the ignition threshold. Fields with higher calorific value have lower threshold – they are easier to set on fire. Secondly, calorific value determines how long and how intensive given field will burn. A field is considered as burned out when its calorific value reaches zero, so the higher the value is, the longer field can burn.

Calorific value also influences the fire intensity value of a field under fire. Transition function uses saturation arithmetic when calculating the fire intensity, with maximum value dependent on the original calorific value of the field. Fields with higher original calorific value burn with greater intensity [1]. Existence of forest on a given field translates to higher calorific value, while fields occupied by rivers, lakes or seas have calorific value equal to zero – those fields cannot be set on fire.

The key part of the simulation is the transition function. Its purpose is to calculate new state of the field based on data from previous iteration of the simulation. New fire intensity is calculated by summing the fire amount that

spreads from all neighbors to the considered field. The fire intensity of the field itself from the previous iteration is also taken into account and the final value is truncated in accordance to the saturation arithmetic.

The amount of fire spread from neighbor fields is determined by coefficients of the simulation (possibly independent for each and every neighbor). Those need to be chosen experimentally to suite desired velocity of fire spreading. The value of fire spread from different neighbors is also influenced by the wind [1]. Fig. 9.2 shows two examples of coefficients values for neighbor fields.



(a) coefficients for no wind conditions

(b) coefficients for windy conditions (w_x=0.05, w_y=0.1)

Fig. 9.2. Example values of simulation coefficients

Fig. 9.2a shows an example of coefficients in no wind conditions. Coefficients for fields adjacent by edge to the central one are bigger than those for fields adjacent by corner by a factor of π (approximately). The motivation for such distinction is that the Euclidean distances between centers of respective fields and the middle of central field also differ by a factor of π. Fig. 9.2b shows coefficients influenced by wind. It is defined as a pair (w_x, w_y) representing horizontal and vertical components of the wind. Parameters w_x and w_y are dimensionless and are added (or subtracted) to the original coefficients of the fields depending on the direction of spreading.

Fire intensity of neighbor is multiplied by respective coefficient and added to the overall intensity of the field. New values are calculated based on a snapshot of values from previous iteration. In each iteration, calorific values of fields are decreased according to the current fire intensity.

9.2. Adding firefighters to the mix

So far, a model has been defined for simulation of fire spreading. To determine proper locations of fire stations, influence of firefighters has to be taken into account as well. Two aspects that require particular attention are velocity of

firefighters and the time required to suppress fire of different intensity on fields within the reach of firefighters. The number and individual behavior of firefighters in not considered in this model. It is assumed that there is always enough firefighters to extinguish the fire.

Velocity determines how long will it take before firefighters reach the fire. It is assumed that firefighters can move in any direction (excluding fields representing water reservoirs) and the maximum velocity is calculated based on terrain slant and the type of site. For example, firefighters move slower in forest due to existence of obstacle (in contrast to fire, which moves faster due to high calorific value).

When firefighters reach a field which is under fire, the influence of water (or other extinguishing means) in every following iteration is twofold:

- it reduces fire intensity, which is the direct effect of extinguishing,
- it reduces calorific value, so that fields visited by firefighters are less likely to be set on fire again (the longer water is applied to a field, the less likely re-ignition becomes).

Parameters of the simulation need to be fine-tuned to best reproduce real-world conditions.

To evaluate how suit a set of fire stations for extinguishing fire on a given terrain, a set of test cases has to be executed. Each test case consists of a set of locations pointing where fire is ignited around the terrain. Every set of fire stations is tested against multiple fires started in different place. It is needed, because in reality fire may be set up in any location. A set of fire stations that handles successfully the most of test cases, scores the best results. This way selected locations are not suited for only one scenario of fire spreading but covers a whole range of possible cases. The result of evaluation is a single number denoting the overall area that was burned during simulation of all test cases.

9.3. Choosing best locations

With a model for testing a single set of fire stations already defined, the remaining issue is how to choose sets of fire stations for evaluation to maximize the chance of finding optimal (or near-optimal) solution. The basic approach is to choose sets of stations randomly and test which of them are rated highest. The bigger input data is, the greater is the chance that one of randomly chosen sets scores good result. The problem is that increasing the size of input data also increases processing time significantly.

9.3.1. Grid-level parallelism

One solution, to the issue stated above, is to use a distributed, high-performance computing platform for the needs of simulations execution. Individual simulations can be run independently on parallel machines to reduce the overall time of tests [5]. The fire spreading model described above was implemented as a computational module for the Comcute grid-computing platform. The target platform makes use of processing power donated by volunteers. By distributing computational tasks – e.g. one set of fire stations tested by each participant – great overall performance can be achieved.

Computational module was implemented as a Java applet which runs in the browser of a volunteer. This way, joining the grid does not required installation of any additional software on volunteer's computer. Participants donate their processing power by a single click of a link on Comcute's project website.

Task distribution model of Comcute platform is especially useful in case of emergency situation. If there is a huge forest fire, described computational module can be used to determine locations where firefighter troops should be send. In such sudden situation a lot of participants can easily attach their machines to the grid ad-hoc by simply visiting project's website. Those can be both volunteers and people obligated to donate unused processing power of their machines in case of a crisis (e.g. employees of public institutions).

9.3.2. Volunteer's machine-level optimizations

Distributing computations among volunteers connected to the grid can decrease time needed to perform simulations for whole input set by a huge factor. However, the time of a single simulation run on volunteer's machine can be significant if the computational module is not designed with performance in mind.

Typical sizes of maps used for simulation range from 1024x1024 to 4096x4096 pixels. For the smallest map, usually about 1500 iterations of simulation are required to get to the point where fire is extinguished and there is no activity that could change situation on the map in any significant way. In a naive approach, computational module would iterate over whole map in every single iteration. Such approach would require 1 572 864 000 calls of the transition function in case of the smallest map. Time required for such simulation can be up to several or even over a dozen – in case of slower machines – minutes. Given the fact that a test of a single set of fire stations consist of several test cases, it can take over an hour to obtain a sole mere result.

The main problem of the described naive approach is that in each iteration of the simulation a lot of fields are processed unnecessarily. Those include fields that represent water reservoirs and places that fire did not reach yet. Depending on the map of the terrain, omitting rivers, lakes or seas can greatly improve the time of the simulation.

Avoidance of processing of fields that are not under fire (and do not have neighbors which are under fire) is especially important at the beginning and at the end of simulation. In the initial phase, only a small group of fields is under fire and it will take many iterations before fire covers larger area. At the end, most field are already burned or fire has been suppressed on them so any further iterations will not change the state of those fields. Processing whole map in those cases is a waste of processing power.

In a perfect situation, only fields under fire and their neighbors would be processed during each iteration. However, it would not be memory efficient to keep a list of individual fields involved in the simulation. Moreover, because of the size of such list, operations like looking for duplicates (a field can have two neighbors under fire, which would qualify it twice for processing in next iteration) or removing a field could take a significant time. Those issues could be diminished by the use of associative containers with guaranteed constant complexity of operations. Nonetheless, the issue of memory usage still remains. Additional problem is that handling each field individually does not facilitate efficient parallelization of work. Time of processing of a single field is so short that the cost of dispatching and handling of such task would surpass performance gain from simultaneous execution.

The solution to the problem is to divide the whole map into smaller blocks which contain a set of fields each. Fig. 9.3 shows an example of dividing a map of size 16×16 fields into blocks of size 4×4 fields.

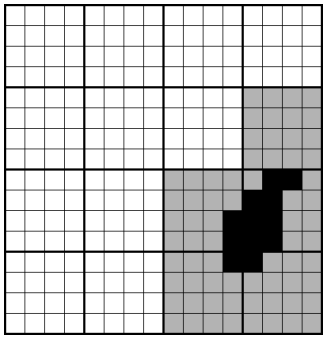


Fig. 9.3. Division of a map into blocks

Blocks are then used to determine which parts of the map require processing in the next iteration of the simulation. Black fields represent an area which is under fire after a given iteration (the actual amount of fire is not important – only the fact that there is some amount of fire that can spread is relevant in this example). Blocks with gray background are the ones which will be processed during the next iteration. The top-most block with gray background does not contain fields under fire but it has neighbor fields in that condition from another block. It means that there is a possibility that fire will spread between blocks in

the next iteration. With described optimization, the time of a single simulation was decreased from several minutes to under a minute.

Further optimization is based on the fact that processing of each field is independent from processing of all other fields. New state of a field is calculated based on the state of it neighbors from the previous iteration. It means that processing of fields within the same iteration can be performed in parallel without any locks or critical sections. To reduce the time of a single iteration, implemented Java applet creates a pool of worker threads and dispatches processing of blocks to them. Each block is processed sequentially by the worker thread. As was already stated, processing of individual fields in parallel would not provide any performance gain due to short time of calculations compared to the cost of dispatching tasks to the worker thread. By the use of block as a unit of work for worker threads, the time of processing of dispatched jobs is increased and overcomes the cost of dispatching and handling.

Multi-threaded execution decreases the time of simulation (down to below half a minute on dual core processor with Hyper-Threading) but at the same time, it increases the load generated on volunteer's machine. The number of worker threads created by the Java applet is one of its internal parameters and can be fine-tuned depending on the situation. In normal conditions only one worker thread can be used to not overload volunteer's machine. In case of emergency situation the size of worker threads pool can be increased to obtain needed results faster.

9.4. Improving the results – genetic approach

Results of a single run of simulations already bring some knowledge about optimal placement of fire stations. They are, however, only as good as the input data was. To improve received results, it is possible to increase the number of tested sets of fire station. The problem with such approach is that the overall time of simulations can increase significantly without any guarantee that final results are better than with smaller set of input data. It is, after all, just a random search in the - quite infinite - space of all possible solutions. The best approach would be to use results of one run of simulations and build upon them to produce another set of input data that could score better results.

The one solutions that comes up almost instantly is the use of genetic algorithms. They perfectly match the theme of the desired solution described above. What was previously called input data set, becomes a population according to the terminology of genetic algorithms. A single set of fire stations becomes an individual in the population. By using genetic operations like crossover and mutation one population can be transformed into another – with individuals expected to score better results – which would represent the next epoch. Fig. 9.4 shows an overview of a genetic algorithm applied to the problem of finding near-optimal locations for fire station.

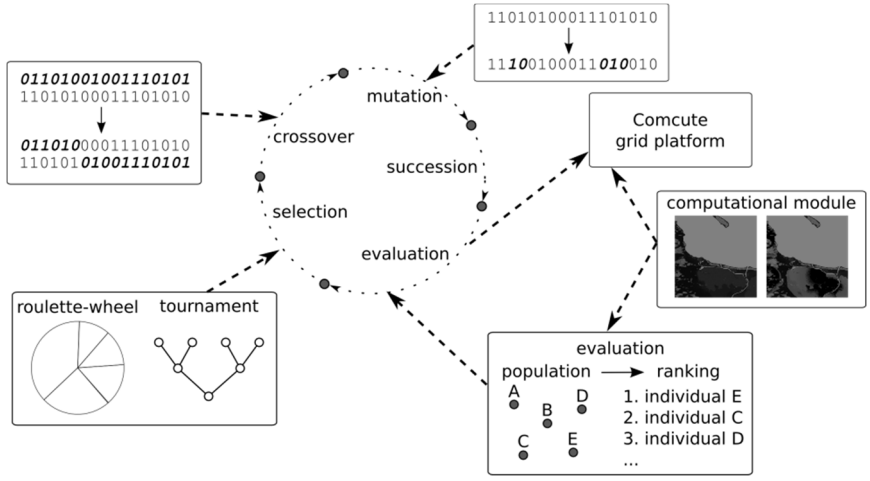


Fig. 9.4. Overview of genetic algorithm

Genetic algorithm typically consists of the following steps [4]:

1. Initial population generation.
2. Evaluation of individuals.
3. Selection
4. Crossover and mutation.
5. Succession.

Steps 2-5 are repeated in subsequent epoch.

In case of positioning of fire stations the most demanding step (in terms of processing power) is the evaluation of individuals. As was previously described, to obtain an accurate rating for a single set of fire stations, a handful of simulations need to be performed. By utilizing the power of the grid (as shown in Fig. 9.4), time required for a single epoch is greatly decreased.

9.4.1. Selection

Simulations performed on the grid result in a ranking of individuals ordered according to how well they handled fires. Results from that ranking can be used to calculate values of the fitness function. Fitness values are then used in the stage of selection of genetic algorithm. The basic approach is to use roulette-wheel selection, in case of which the probability of an individual being chosen for future procreation is proportional to the score it received during the evaluation stage. Fig. 9.5 shows an example of roulette-wheel build for five individuals.

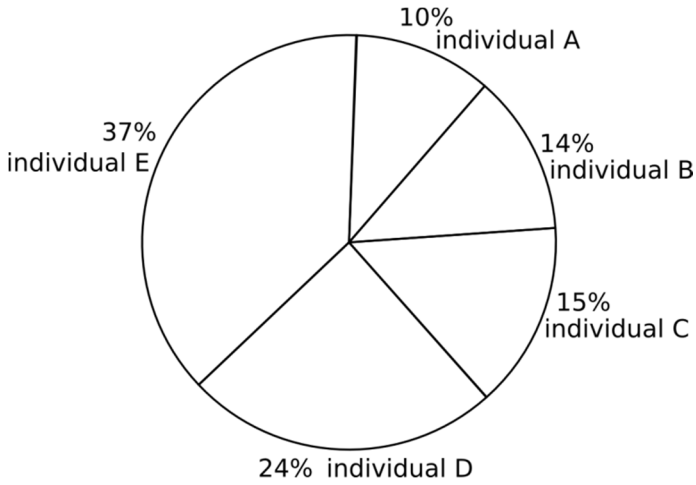


Fig. 9.5. Example of a roulette-wheel for five individuals [3]

Area occupied by each individual is proportional to its fitness value. Individual E got the highest rating during the evaluation stage and has the biggest chance to be selected for future procreation. Individual A got the lowest rating, so it is least likely that it is picked. It is not impossible, though. Presence of such lower-rated individuals can help to get out of local maximum of fitness function (where highest rated individuals may fall in instead of global maximum).

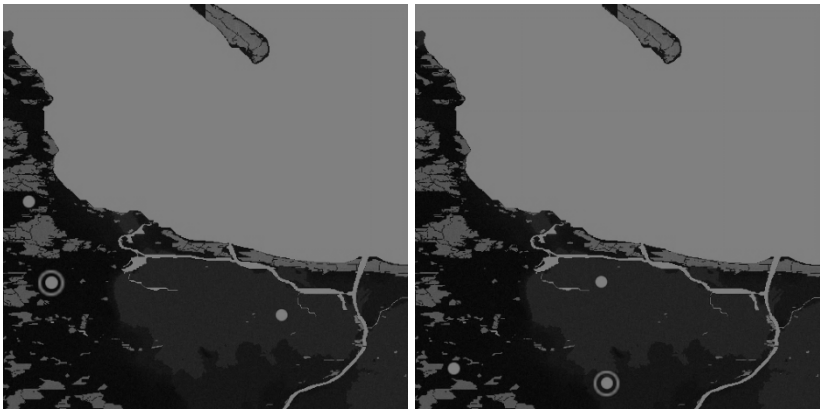
There is a risk, that when creating a new population through randomized operations of selection, crossover and mutation, the best individual from the previous population is lost. To prevent the disappearance of the best chromosome from the population, a method called elitism is used. It copies best individuals from previous population to the new one surpassing the procreation stage [3]. In the worst case the fitness value for the best individual in the new population is not improved compared to the previous population but also does not decline. It improves the performance of genetic algorithm by reducing the number of epoch required to obtain desirable results [3].

9.4.2. Crossover and mutation

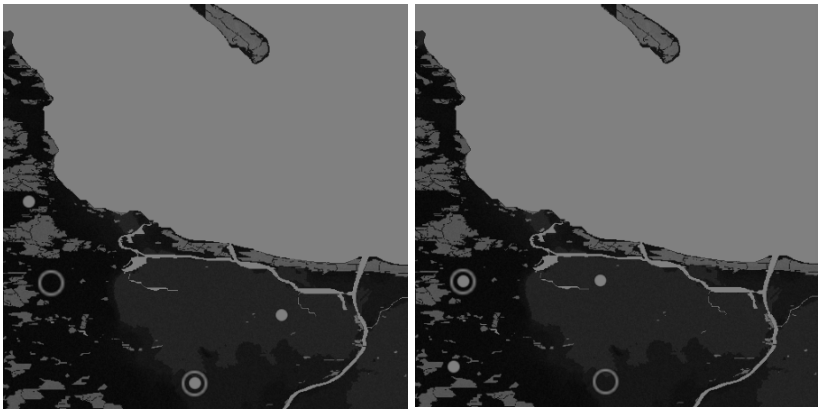
For the purpose of genetic algorithm two basic genetic operations need to be defined for considered individuals: crossover and mutation. The purpose of the crossover is to produce offspring by choosing genes from parent chromosomes [4]. In case of discussed individuals single gene corresponds to a single location of fire station. The crossover operation is to exchange a pair of fire stations between two individuals.

Fig. 9.6 shows an example of crossover. Positions of fire stations are marked with dots. In the Figure 6a parent individuals are presented. Stations (or genes) chosen for exchange are additionally marked with rings. Figure 6b shows two

offspring produced by crossover. Empty rings represent positions of original genes before crossover. The dots marked with rings are the exchanged stations.



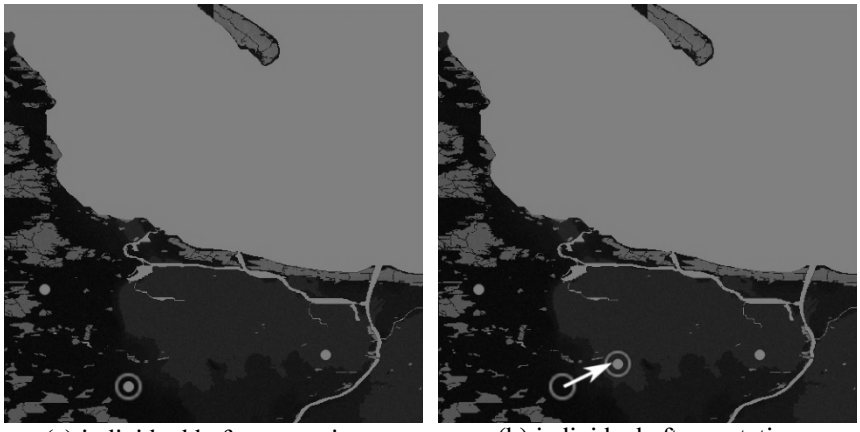
(a) parents chosen for crossover



(b) two offspring

Fig. 9.6. Example of crossover of two individuals

Mutation of a chromosome is a random change of some of its genes [3]. As genes correspond to fire stations, mutation is as simple as a random displacement of one or more stations. Fig. 9.7 shows an example of such mutation. In Fig. 9.7a an individual before mutation is presented. The fire station (the gene) which is about to undergo mutation is marked by a ring. Fig. 9.7b shows the individual after the mutation. Empty ring represents the original position of the station (the original gene) and the arrow shows the displacement of the station during mutation.



(a) individual before mutation (b) individual after mutation

Fig. 9.7. Example of mutation of an individual

9.4.3. Bringing the pieces together

By combining selection, crossover and mutation techniques described above with already-implemented Comcute computational module for evaluation of individuals, a complete genetic algorithm is obtained. It was implemented as a Java desktop application (shown in Fig. 9.8).

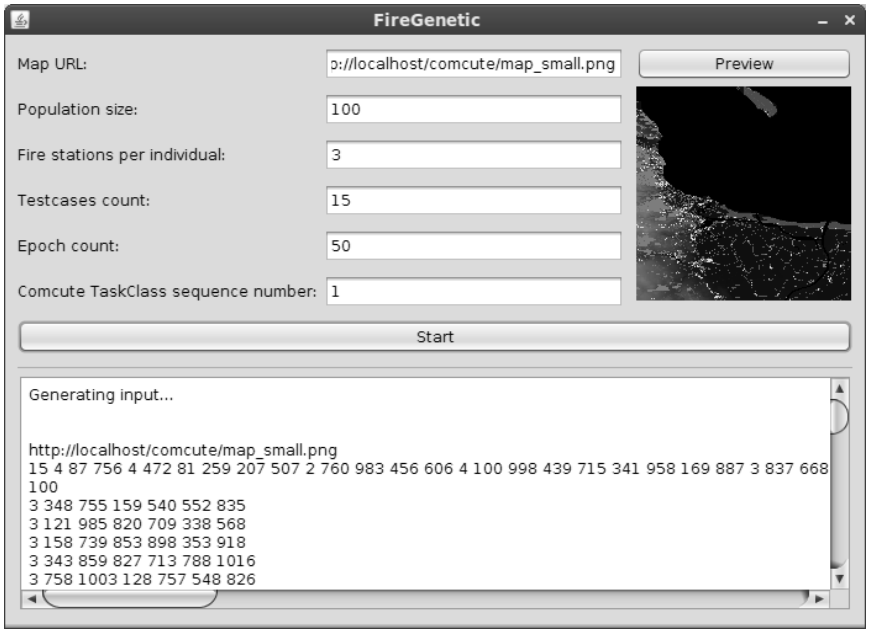


Fig. 9.8. Main window of Java application implementing genetic algorithm

The graphical user interface of the application provides fields for setting up parameters of the simulation (location of the map, number of fire stations in each set, number of test cases) and parameters of the genetic algorithm (population size, epoch count). Identifier of the Comcute computational module also needs to be provided to offload evaluation tasks to the grid.

Application uses Comcute platform's customer API to register tasks and input data, monitor the state of execution on the grid and to retrieve final results. Communication between Java application and Comcute platform takes place through web services. For the needs of authentication, a dedicated customer account (with adequate permissions) has been created on the Comcute platform for use by the Java application.

Java application performs subsequent steps of the genetic algorithm (as shown in Fig. 9.4). Evaluation stage is offloaded to the grid. Results of evaluation are retrieved and used as an input data for selection and further steps of the algorithm. At the end of iteration, a new population is obtained for the next epoch. Evaluation of individuals is again offloaded to the Comcute grid and the algorithm continues as above till reaching the desired number of epoch. Data from subsequent epoch is recorded for future analysis.

9.5. Tests and results

Presented solution was tested with the help of anonymous volunteers donating processing power of their computers to Comcute grid project. Fig. 9.9 shows the main project website with fire simulation in progress. To encourage users to participate in the project, a live visualization is presented on the website during the simulation. Fig. 9.10 shows four snapshots from a simulation performed on one of volunteers' machines.

During the simulation, an increased system load can be observed. Figure 11 shows history of CPU, memory and network usage during the start and execution of a simulation. The top-most chart shows CPU load history. It presents four series of data – one for each of processor's cores. The middle chart shows the history of RAM usage. It was truncated as all plotted values do not exceed 20%. The bottom-most chart shows network usage.

Point A in the Fig. 9.11 represents the moment when user activated a link on the Comcute website to participate in the computations (plots before that point represent load of an idle system with running web browse). Period A-B represents activity of Comcute loader program. Its purpose is to download computational module and other required resources (e.g. additional libraries) and pass the control flow to the module. High peek of the network usage shows when resources were downloaded.

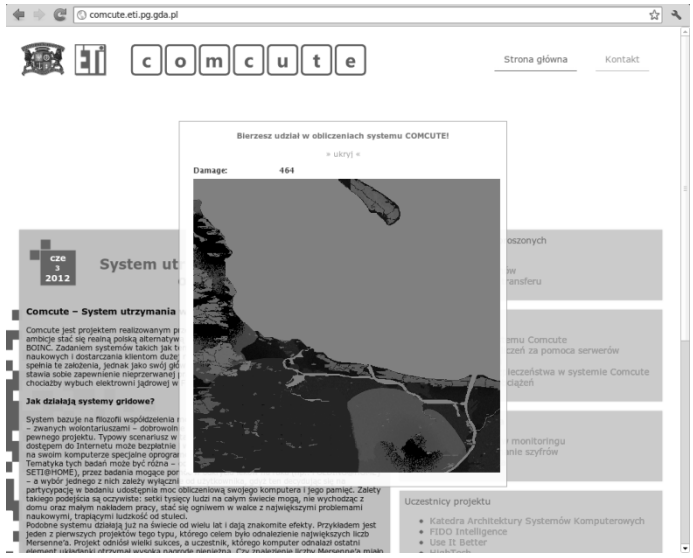


Fig. 9.9. Comcute grid website with fire simulation in progress

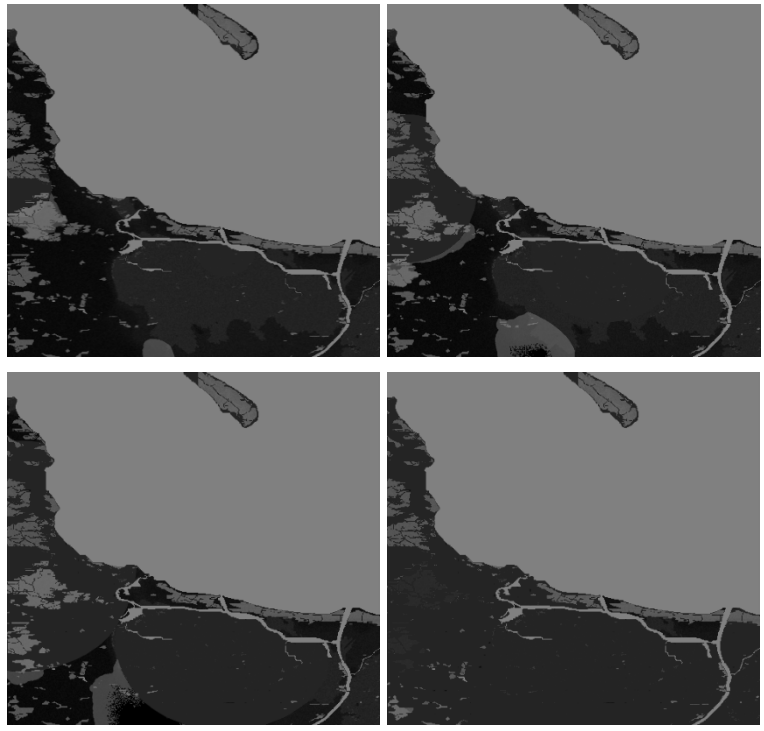


Fig. 9.10. Example of ongoing simulation

Point B marks the beginning of computational module execution. When the simulation starts, all cores of the processor are utilized (the number of worker threads can be tuned, as stated before). Points C and D show how system load drops between execution of subsequent test cases. Each test case is run using multiple worker threads but the switch to another test case is a single-threaded operation. According to the chart, execution of a single test case takes approximately 15-20 seconds.

Fig. 9.11. CPU, memory and network usage at the start and during the simulation

Values from several epoch of genetic algorithm were recorded to analyze if genetic operations produce individuals that score better results. The ultimate criterion is the score of the best individual in the population in subsequent epoch. A chart presenting area burned during simulations involving best individuals is shown in Fig. 9.12. The x-axis represents epoch and the y-axis shows burned area. As can be easily seen, scores improve over subsequent epoch.

How much can be improved during the execution of genetic algorithm, is highly dependent on the randomly chosen initial population. If one of initial individuals is already close to the optimal solution, improvements are slight at best (and there can even be no improvement at all). On the other hand, if none

of initial individuals is near the optimal solution, significant improvements are only seen after considerable amount of epoch.

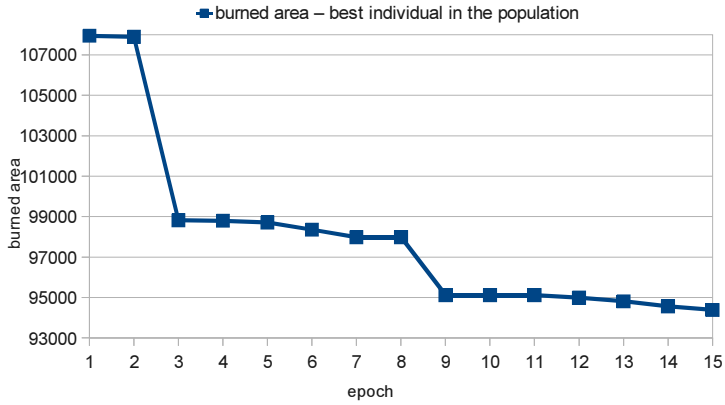


Fig. 9.12. Area burned during simulation involving best individual of the population in subsequent epoch.

Fig. 9.13shows a set of fire stations which corresponds to the best individual from epoch 15 of genetic algorithm execution presented by the chart in Fig. 9.12.

Fig. 9.13. Location of fire stations corresponding to the best individual from epoch 15 (see Fig. 9.12).

9.6. Conclusions

Performance of a single simulation performed on volunteer's computer could be further improved by the use of NVIDIA® CUDA technology. It would give possibility to offload multi-threaded computations to the graphic card, which it is best suited for. The performance gain would allow to execute more testcases (which increases the accuracy of evaluation) or to perform simulation on higher-resolution maps (which increases the accuracy of simulation) or possibly both. Described implementation uses Java applet as the technology for Comcute computational module and libraries like JCUDA [6] (a set of Java bindings for NVIDIA CUDA libraries) would provide means for utilizing the power of volunteers' graphic cards.

References

1. Almeida R. M., Macau E. E. N., *Percolation model for wildland fire spread dynamics*, Proceedings on International Conference on Chaos and Nonlinear Dynamics, São José dos Campos. South America, 2010.
2. Foster I., Kesselman C, Tuecke S., *The anatomy of the grid: Enabling scalable virtual organizations*, Int. J. High Perform. Comput. Appl., 15(3), August 2001, pp. 200-222.
3. Nedjah N., A. Abraham, Luiza de Macedo Mourelle, *Genetic Systems Programming: Theory and Experiences,* Springer Verlag, New York 2009.
4. Pyne S. J., P. L. Andrews R. D. Laven: *Introduction to wildland fire,* John Wiley and Sons, New York 1996.
5. Russell S. J., P. Norvig, *Artificial Intelligence a modern approach*, Prentice Hall, Upper Saddle River, 2nd edition, New York 2003.
6. JCUDA Project, http://www.jcuda.de/, May 2012.