*Miloš Savić*

# PARPER – paralelizovani simulator procesa perkolacije na kvadratnim rešetkama

*Realizovana je paralelizovana verzija računarske simulacije procesa perkolacije na kvadratnim rešetkama u mrežnom okruženju. Kao model korišćena je probabilistička verzija modela šumskog požara. Osnovna ideja je da se rešetka podeli na manje podrešetke metodom prostorne dekompozicije. Simulator je realizovan kao klijent/server program u koncepciji koja je poznata kao distributivni server: klijenti procesiraju podrešetke pravilima modela, dok server raspoređuje podrešetke na procesiranje klijentima i vrši distribuciju stanja ćelija koje se nalaze na ivicama podrešetki. Koncept prikazan u ovom radu omogućio je decentralizaciju perkolacije na kvadratnim rešetkama. Suprotno očekivanjima, pristup se pokazao podjednako efikasan u pogledu vremena potrebnog za izvršavanje simulacije kao i sekvencijalna varijanta. Dobijeni rezultati simulacije su u skladu sa rezultatima iz literature.*

## Uvod

Osnovna ideja perkolacije je da se utvrdi postojanje naglog prelaza pri kojima povezanost elemenata sistema na velikim distancama nestaje ili se pojavljuje. Jedan od prvih modela perkolacije bio je Bernulijev model perkolacije kroz rešetke. Kasnije su razvijeni i drugi modeli od kojih su najpoznatiji model epidemije, model šumskog požara, model formiranja galaksija i klastera galaksija. Perkolaciona teorija je takođe omogućila istraživanje mnogo komplikovanijih fizičkih fenomena, kao što su fazni prelazi kod magnetnih materijala, i uopšte u termodinamici. Perkolacija se može simulirati kao Monte Carlo simulacija kod probabilističkih modela perkolacije (model šumskog požara i epidemije) (Kossi 1998), ili kao problem određivanja velikih perkolacionih klastera kod modela gde je perkolacija čisto topološki problem (određivanje klastera galaksija) (Tiggemman 2002; Hoshen and Kopelman 1970). Algoritmi koji simuliraju perkolaciju na kvadratnim rešetkama dizajnirani su prvenstveno za tradicionalne računare koji su zasnovani na sekvencijalnoj Von Neumann-

*Miloš Savić (1984), Valjevo, Maljenska bb, učenik 4. razreda Valjevske gimnazije*

ovskoj arhitekturi (Tiggemman 2002) i značajno su komplikovaniji od algoritama namenjenih za izvođenje na paralelnim arhitekturama (višeprocesorski računari, računarski klasteri).

Proces perkolacije pogodan je za implementaciju na višeprocesorskim arhitekturama jer se ceo sistem sastoji iz konačnog broja ćelija koji se mogu nezavisno procesirati. Jedan od načina da se paralelizuje proces perkolacije jeste prostorna dekompozicija (Tiggemman 2002). Ideja prostorne dekompozicije jeste da se rešetka podeli na onoliko podrešetki koliko ima procesorskih jedinica i da se svakoj procesorskoj jedinici unapred dodeli po jedna podrešetka na procesiranje.

Cilj rada je da se razvije algoritam koji će omogućiti decentralizaciju perkolacije na kvadratnim rešetkama i dobiti što efikasniju paralelizovanu implementaciju. Osnovna ideja kod klastera jeste mogućnost da umreženi računari rade kao jedinstven sistem. Na taj način se procesiranje informacija može podeliti, pri čemu svaki računar u tom klasteru dobija jedan deo tih informacija za procesiranje. Tako neke sekvencijalne algoritme možemo redizajnirati i napraviti paralelizovane verzije tih algoritama koje rade brže, jer se teži da se procesiranje ravnomerno deli među svim računarima u okviru klastera.

Ideja je da se proces perkolacije implementira koristeći koncepciju distributivnog servera. Kao model korišćen je probabilistički model šumskog požara (engl. *fire-forest model*) (Kossi 1998), koji je značajno komplikovaniji od topoloških problema i kao takav pogodan za testiranje efikasnosti koncepcije distributivnog servera koji sadrži *load balancer*.

Simulator se realizuje implementacijom dva programa: servera, koji se izvršava na samo jednom računaru u okviru klastera, i klijenta, koji se izvršava na ostalim računarima. Server podeli rešetku na podrešetke koje su oblika pravougaonika. Klijent program simulira proces perkolacije za datu podrešetku. Zadaci servera su da, u zavisnosti od opterećenja klijent računara, raspoređuje podrešetke na klijent računare i da se stanja na ivicama podrešetki sa jednog klijent računara preko njega distribuiraju odgovarajućem drugom klijent računaru gde se simulira perkolacija susedne podrešetke. Ostala stanja ćelija podrešetke se ne distribuiraju jer ne utiču na stanja ćelija u ostalim podrešetkama.

Za razliku od rešenja u literaturi (Tiggemman 2002), u novom pristupu server deli rešetku na više podrešetki od broja procesorkih jedinica i, u zavisnosti od opterećenja računara u klasteru i trenutka kada počne proces perkolacije podrešetke, server najneopterećenijem klijentu dodeljuje podrešetku na procesiranje. Tako je omogućeno da se simulacije podrešetki vrše paralelno, da nema "praznog hoda" nekih računara u klasteru, da opterećenje svih računara u klasteru bude približno isto, i da komunikacija među računarima u klasteru bude svedena na minimum.

## Model sistema

Problem perkolacije kvadratne rešetke posmatramo na modelu šumskog požara (engl. *fire-forest model*). Šuma je predstavljena kao kvadratna rešetka veličine n, gde je n promenljiv parametar. U svakoj ćeliji rešetke može biti jedno od tri stanja: nema drvo (stanje A, koje ima vrednost 0), ima drvo koje ne gori (stanje B, koje ima vrednost 1) i ima drvo koje gori (stanje C, koje ima vrednost 2). Na sledećoj šemi je dat model sistema zapisan u pseudo kodu koji je sličan programskom jeziku C.

```
int perkolacija(int n, double gustina)
{
    int br_iteracija = 0;

    inicijalizuj_resetku(gustina);
    zapali(0, 0, n/4, n);
    while (!perkolacioni_uslov) {
        br_iteracija++;
        for (drvo = svako_drvo_koje_gori) {
            if (vremenski_limit(drvo))
          ugasi_drvo(drvo);
      else {
          zapali(drvo-okolina);
                povecaj_vreme_gorenja(drvo);
            }
        }
    }
    return(br_iteracija);
}
```

U početnom stanju sistem se inicijalizuje tako što se svakoj ćeliji dodeli jedno od stanja A (ne postoji drvo) ili B (postoji drvo) sa određenom verovatnoćom za stanje B, koja je ulazni parametar za simulaciju. Potom se zapali svo drveće u jednoj četvrtini rešetke, tj. za sve ćelije rešetke koje se nalaze u polju pravougaonika čija su temena koordinate (0, 0) i (n/4, n), a koje imaju stanje B, to stanje se promeni u stanje C. Tada se sistem pusti da evoluira sa sledećim pravilima:

1. U toku jedne iteracije se u okolini svakog zapaljenog drveta bira jedno polje, i ukoliko na tom polju postoji drvo ono se zapali, tj. bira se jedna ćelija iz Murovog susedstva ćelije sa stanjem C i ukoliko je njeno stanje B ono se menja u C.
2. Svaka ćelija rešetke može goreti, tj. biti u stanju C, najviše sedam iteracija. Posle toga stanje te ćelije je A (ne postoji drvo).

Kažemo da rešetka (šuma) perkolira kada se požar ugasio, to jest kada ne postoji ni jedna ćelija rešetke koja ima stanje C. Zadatak računar-

ske simulacije je dobiti zavisnost vremena perkolacije od gustine šume, tj. verovatnoće stanja B u početnoj kvadratnoj rešetki, i odrediti kritičnu gustinu za koju bi šuma koja ima dimenzije koje teže beskonačnosti perkolirala beskonačno.

## Implementacija

Simulator koji realizuje ideje izložene u ovom radu napisan je u programskom jeziku C i zove se PARPER. Kao elementi međuprocesne komunikacije korišćeni su *socket*i. Simulator je tako dizajniran da nema potrebe za instalacijom dodatka LINUX operativnom sistemu koji omogućava *cluster computing*. Za pokretanje simulacije potrebna je mreža LINUX računara koja se može ostvariti i preko Interneta. Pored toga, ideja distributivnog servera omogućila je da broj računara u okviru klastera ne mora biti fiksan, to jest simulator se automatski prilagođava broju računara u klasteru i maksimalno koristi resurse svakog od njih, što doprinosi brzini simulacije. Simulator je implementiran tako da se može portovati na druge platforme, tako da je simulaciju moguće izvršavati i na heterogenim klasterima. Ulazni parametri za simulator su veličina rešetke i datoteka u kojoj se nalazi spisak svih računara u mreži.

## Koncepcije klijenata i servera

Server program generiše i inicijalizuje rešetku u kojoj je jedna četvrtina zapaljena. Potom se zapaljena oblast podeli na $k$ podrešetki, gde je $k$ broj računara u mreži. Ostatak rešetke se podeli na $3k$ podrešetki. Server rasporedi po jednu od zapaljenih $k$ podrešetki svakom računaru u mreži. Kako se požar širi, tako se nove zapaljene podrešetke dodeljuju na procesiranje najneuposlenijim računarima, tako da su resursi mreže maksimalno i skoro podjednako iskorišćeni. Informacije o svakoj podrešetki server čuva u specijalnoj strukturi podataka koji se zove niz zona. Niz zona ima onoliko elemenata koliko ima i podrešetki. Za svaku podrešetku se čuvaju dužina, širina, stanje podrešetke, (0, 0) koordinata podrešetke u rešetki, *socket* koji služi za komunikaciju sa klijentom i specijalni baferi (implementirani kao redovi (engl. *queue*)) u kojima se čuvaju koordinate onih ćelija koje će biti prosleđene klijentu, a na čija stanja su uticale susedne podrešetke. Svaka podrešetka može imati jedno od dva stanja: gori ili ne gori.

Server raspoređuje podrešetke na klijent računare. Zato on za svaki računar u mreži vodi evidenciju o tome koliko podrešetki se procesira na njemu u strukturi podataka koja se zove lista opterećenosti. Na jednom klijent računaru je moguće procesiranje više podrešetki jer se simulacija perkolacije jedne podrešetke izvršava u jednoj niti (engl. *thread*) klijent procesa.

Klijent deo simulacije čeka asinhronu poruku za prijem konekcije i kada je dobije kreira novi *thread* koji preuzima podrešetku i počinje sa njenim procesiranjem. Procesiranje se vrši tako što se podrešetka obilazi ćeliju po ćeliju. Ukoliko ćelija ima stanje B (postoji drvo) ona može promeniti jedno stanje iz njene okoline iz B u C (drvo koje gori). Posle sedam iteracija u kojem je posedovala stanje C, stanje ćelije se menja u stanje A (ne postoji drvo). Ukoliko se desi da neko stanje ćelije podrešetke na ivici prouzrokuje menjanje stanja ćelije koja ne pripada datoj podrešetki, koordinate tog polja se stavljaju u bafer. Posle svake iteracije taj bafer se šalje server delu simulatora koji preračuna date koordinate i šalje ih odgovarajućim klijentima sa njihovim novim stanjima. To znači da posle svake iteracije svaki *thread* u okviru klijenta pošalje jedan bafer sa koordinatama ćelija susednih podrešetki i primi jedan bafer od servera u kome se nalaze koordinate onih ćelija podrešetke čija su stanja promenjena uticajem neke druge podrešetke. Na taj način iteracije svih podrešetki su "usklađene", tj. ne može početi nova iteracija jedne podrešetke ukoliko nije kompletirana iteracija neke druge podrešetke. Tako se na veoma lak način može izračunati broj iteracija cele rešetke.

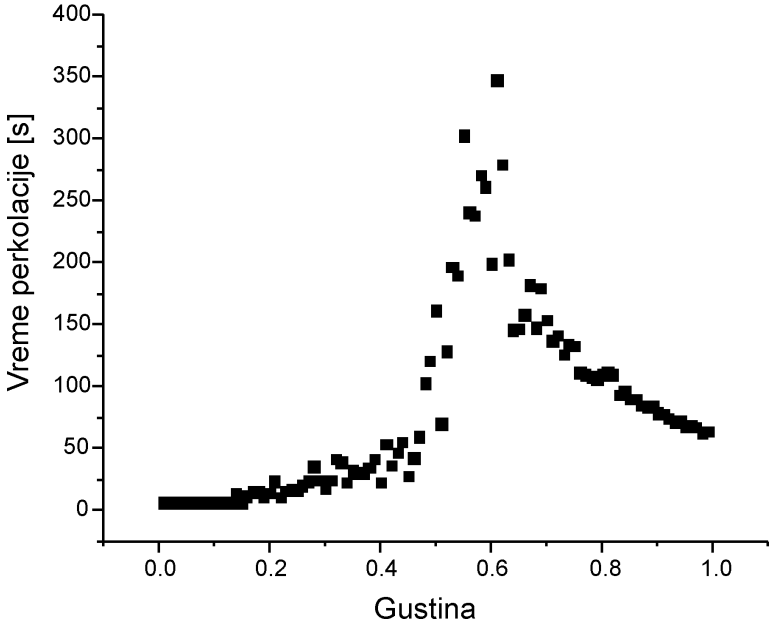
Kada se desi da je stanje jedne od ćelija na ivici neke podrešetke uzrokovalo da se druga podrešetka zapali, server na osnovu informacija iz liste opterećenosti bira klijenta koji je najmanje opterećen i njemu prosleđuje podrešetku na procesiranje. U toku procesiranja podrešetke u jednom trenutku će se dogoditi perkolacija iste. Tada, umesto bafera sa koordinatama, *thread* šalje takozvani stop kod i vraća celu podrešetku kao rezultat, jer je moguće da će se ta podrešetka u daljem toku simulacije opet zapaliti. *Thread* se tada uništava, a server smanjuje vrednost koja karakteriše opterećenost tog klijenta.

Server se posle slanja početnih podrešetki vrti u beskonačnoj petlji koja se može prekinuti (while (1)). U jednoj iteraciji petlje on prima bafere sa koordinatama polja od svih *thread*-ova i kreira za svaki *thread*, odnosno za svaku zonu, novi bafer koji je rezultat usklađivanja ivica podrešetki. Usklađivanje se vrši tako što se preračunavaju koordinate ćelija. Ukoliko se desi da se ćelija koja menja stanje nalazi u podrešetki koja ne gori, kao što je već rečeno server uspostavlja komunikaciju sa klijentom koji je najmanje opterećen i njemu se prosleđuje podrešetka, a ukoliko se desi da je umesto bafera primio stop kod, server prima podrešetku i vrši restauraciju originalne rešetke. U svakoj iteraciji te beskonačne petlje server uvećava brojač, koji je inicijalizovan na nulu, za jedan. Kada se desi da ne postoji ni jedna podrešetka koja se procesira, ta beskonačna petlja se prekida, a vreme perkolacije predstavljeno je vrednošću brojača, čime smo dobili rezultat simulacije.

U prilogu 1. nalazi se izvorni kod server dela simulatora PARPER, dok se u prilogu 2. nalazi izvorni kod klijent dela simulatora PARPER.

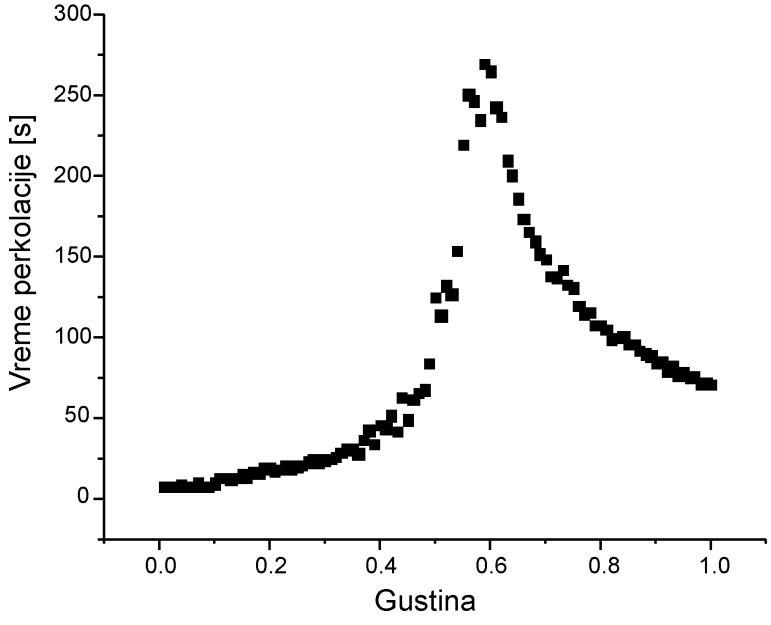
## Rezultati simulacije

Da bismo ispitali korektnost i efikasnost paralelizovane simulacije implementirali smo jednoprocesorku verziju iste. Na slici 1 data je zavisnost vremena perkolacije od gustine rešetke veličine 100×100 koja je dobijena jednoprocesorkom simulacijom, dok je na slici 2 ista zavisnost za istu rešetku koja je dobijena korišćenjem simulatora PARPER.




Slika 1.
Zavisnost vremena perkolacije od gustine za rešetku 100×100 dobijena jednoprocesorskom simulacijom

Figure 1.
Dependence of percolation time on 100×100 lattice density, determined using the one-procesor simulation




Slika 2.
Zavisnost vremena perkolacije od gustine za rešetku 100×100 dobijena simulatorom PARPER

Figure 2.
Dependence of percolation time on 100×100 lattice density, determined using the PARPER simulation

Sa grafika se vidi da vreme perkolacije kvadratne rešetke raste do određene gustine, a da posle te gustine opada. Vrednost kritične gustine dobijene jednoprocesorskom verzijom simulacije je $p_c = 0.61$, dok je vrednost kritične gustine dobijene simulatorom PARPER $p_c = 0.59$. Vrednosti za $p_c$ uzeti iz literature (Newman and Ziff 2000) iznose $p_c = 0.5927464$, čime se još jednom potvrđuje korektnost simulatora PARPER. Zanimljivo je da se na osnovu grafika može uočiti fazni prelaz, što ostavlja prostor za dalja istraživanja.

Kada dizajniramo i implementiramo algoritam na paralelnim arhitekturama, potrebno je odrediti i efikasnost algoritma (konkretne simulacije). Kod prostorne dekompozicije, "količina" potrebne komunikacije je grubo proporcionalna količini informacija koja je potrebna da se razmene stanja na ivicama podrešetki, tj. $N \cdot L^{d-1}$ (Tiggemman 2002), gde je $N$ broj procesorskih jedinica, $L$ veličina rešetke, a $d$ dimenzija rešetke (u našem slučaju $d = 2$). Komunikacija je osnovna razlika između sekvencijalne i paralelne implementacije i umanjuje efikasnost paralelizovane verzije.

Simulator PARPER je testiran na Petničkoj mreži, na jednom i na dva računara. Mereno je vreme izvršavanja programa za rešetke veličine 100, 200 i 500. Rezultati su prikazani u tabeli 1.

Tabela 1. Rezultati efikasnosti paralelizovane simulacije

| Veličina rešetke | Vreme na jednom računaru [s] | Vreme na dva računara [s] |
|---|---|---|
| 100 | 10±2 | 8.2±2.2 |
| 200 | 22±3 | 17.8±2.8 |
| 500 | 40.2±4.8 | 43.2±4.8 |

Svakako da na osnovu vremena izvršavanja simulacije na dva računara ne možemo doneti konačan zaključak o efikasnosti simulacije, i da simulacija ima svoj smisao jedino ako se izvršava na mreži od više računara. Na osnovu dobijene tabele vidi se da paralelizovana verzija simulacije perkolacije na kvadratnoj rešetki u dve dimenzije nije puno efikasnija od sekvencijalne verzije, tj. puno procesorskog vremena se troši na komunikaciju. Sličan rezultat dobio je i Tiggemman u svojoj paralelizovanoj implementaciji na super-računaru, koji je pokazao da se drastična ubrzanja dostižu tek u kubnoj rešetki i rešetki u četiri dimenzije. Moguće je da je za određene veličine rešetke paralelizovana verzija sporija do neke granice, a da je posle te granice brža, jer brzina klijenta zavisi od veličine podrešetke, koja se obrađuje. Ako je podrešetka dovoljne veličine da je procesorsko vreme potrebno za samo instanciranje iteracija veće od vremena koje se utroši na komunikaciju, trebalo bi očekivati da je vreme

potrebno za izvršavanje paralelizovane simulacije manje od vremena izvršavanja sekvencijalne. Pored toga treba imati u vidu da su dodatna usporenja uslovljena time što sve iteracije na klijentima moraju biti sinhronizovane.

## Zaključak

Dobijeni rezultati za kritičnu gustinu su u skladu sa rezultatima iz literature (Kossi 1998), što pokazuje da model iznet u ovom radu korektno simulira *fire-forest* model. Vreme potrebno za izvršavanje simulacije na dva računara je grubo približno isto vremenu koje je potrebno za sekvencijalnu simulaciju, što je logično jer se veliki deo procesorskog vremena troši na komunikaciju. Ovaj metod ima smisla jedino ako se primeni na velikom broju računara kada se očekuje da se vreme izvršavanja simulacije znatno smanji.

Algoritam razvijen u ovom radu omogućio je decentralizaciju cele simulacije. Uz minorne modifikacije PARPER-a i dovoljan broj računara (reda veličine 1000) moguće je dobiti kritičnu gustinu za kvadratne rešetke čija veličina prevazilazi vrednost aktuelnog svetskog rekorda, koji sada iznosi L = 4000256. Mrežu ove veličine moguće je napraviti uz pomoć Interneta. Server deo PARPER-a u slučaju veličina rešetki reda L nije u stanju da na kućnim računarima alocira toliko memorije da pamti stanja svih ćelija rešetke. U sadašnjoj verziji PARPER-a, rešetka se memoriše na serveru iz razloga što je neke podrešetke moguće ponovo "zapaliti". Problem se može rešiti tako što server neće uopšte čuvati rešetku, nego će se podrešetke inicijalizovati i čuvati na klijent računarima, a server će u specijalnoj strukturi podataka čuvati informacije o tome gde se koja podrešetka nalazi i koja je njena pozicija u rešetki. Kada server odluči da je za neku rešetku bolje da "migrira", on bi samo inicirao tu operaciju i promenio informacije u specijalnoj strukturi podataka o tome gde se ta rešetka sada nalazi, dok bi klijenti obavljali transfer podataka.

**Zahvalnost**. Zahvaljujem se Marku Petkoviću, Vladimiru Gligorovu, Milošu Božoviću i Srđanu Verbiću, koji su svojim predlozima i sugestijama znatno doprineli izradi celog projekta. Posebnu pomoć u finalizaciji ovog projekta pružila mi je vođa seminara fizike Jelena Grujić.

## Literatura

Hoshen J. and Kopelman R. 1976. *Phys. Rev.*, B14: 3438.

Kossi 1998. *Percolation and Forest Fires*. Dostupno na http://kossi.physics.hmc.edu/courses/p117/Project/1998/Fires/ForestFires.htm

Newman M. E. J., Ziff R. M. 2000. *Phys. Rev. Lett.*, 85: 4104, cond-mat/0101295

Schroeder M. 1991. *Chaos, Fractals and Power Laws*. New York: Freeman

Stevens R. 1998. *Unix Network Programming*. Prentice-Hall

Tiggemman D. 2002. Simulation of Percolation on Massively-parallel Computer

---

*Miloš Savić*

## PARPER – Parallelized Simulator of Percolation on Square Lattices

Percolation is a thoroughly-studied model in statistical physics. Algorithms, which simulate percolation, were invented for traditional sequential computers, and the implementation on modern parallel architectures is far more complicated. Parallelizing an algorithm that works on regular data structures is done with domain decomposition. In this case, the lattice is cut into sub-lattices and each processor unit is assigned on each strip for investigation.

The primary aim of this project was to create a decentralized and efficient parallel algorithm that simulates percolation on square lattices in computer network environment. The basic idea is to implement the process of percolation using domain decomposition with the concept of a distributive server that contains a load balancer.

We considered the problem of percolation on square lattices on the fire-forest model. The forest is represented as a matrix. The matrix is populated with a specific density of trees and the square area with coordinates (0, 0), (n/4, n) is set to fire. The fire is spread by the following rules: fire can only be spread from a burning tree to only one other tree (a single flaming tree cannot ignite two neighbors at once) and for each tree a burning lifetime of seven iterations is allowed. The task of the simulation is to determine the percolating threshold (the critical density).

The simulator implemented in this project is called PARPER. In order to run the simulation, we need a network that is made up by LINUX computers. That network can be realized on the internet. The whole implementation was done in the C programming language.

The simulator is realized by implementing two programs: a server that executes on one, and a client that executes on all the nodes in the network environment. The server divides the lattice into sub-lattices in dependence on the number of computers in the network. The client program

simulates the process of percolation for a sub-lattice, which is sent by the server. The main tasks of the server is to distribute the state of cells from borders for all clients to appropriate ones, and to schedule sub-lattices to clients depending on the resource usage. The client is realized in a model known as TCP concurrent server, so that one thread is allocated for every sub-lattice. The server keeps a special table in order to bind threads to appropriate clients. When the sub-lattice starts to percolate, the server, using the information in the table, determines to which client the sub-lattice will be processed and connects to it. After every iteration, the thread sends cells from borders whose states have been changed to the server. The server receives the cells, recalculates their coordinates, and sends new cells to appropriate clients. If the process of coordinate recalculation results in a cell whose sub-lattice has not been processed, the server starts to percolate that sub-lattice. If the server receives a stop code instead of cells from the client, then the sub-lattice has percolated (its thread is destroyed), and the server refreshes the thread per client table.

Using the PARPER simulator, we determined the value of the critical threshold and we got a graph that shows the dependence of percolation time on lattice density. The value of the critical threshold $p_c$ is 0.59 and is in agreement with the expected value (the value for $p_c$ taken from literature is 0.5927464).

When implementing an algorithm on a parallel architecture, we want to know how efficient our implementation is. We tested the PARPER in a two computer cluster and we found that the parallel implementation is roughly as fast as the sequential one.

The advantage of the concept implemented in this work is that it enables decentralization of the simulation, so with enough computers in the cluster we can beat the world record for lattices up to size 4000256. In the future, we should run PARPER in a network made up by more computers in order to determine the real efficiency of the algorithm and to check if this concept enables reduced network traffic and a better usage of cluster resources.

# Prilog 1. Izvorni kod server dela simulatora PARPER

```
#include  <stdio.h>
#include  <stdlib.h>h
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <netdb.h>
#include <string..h>
#include  <unistd.h>
#include  <time.h>

#define TRUE 1
#define FALSE 0
#define TCP_PORT 5983
#define N 100
#define MAX_COMPUTERS 50
#define ZAPALJENO 2
#define TIME_LIMIT 7

int suma[N][N];
int v[N][N];
struct resurs_t {  char ime_racunara[256];
  int broj_threadova;
} resurs[MAX_COMPUTERS];
int broj_racunara;
int gori_zona = 0;
struct kord_t {
  int x, y;
};
struct zona_t {
  int startx, starty;
  int duzina, sirina;
  int socket, resurs;
  int gori;
  int qs;
  struct kord_t queue[4 * N];
};
struct zona_t zona[MAX_COMPUTERS * 4];
int broj_zona;

int inet_addr(char *name, unsigned long *ip_addr)
{
  struct hostent *host;
  host = gethostbyaddr(name, strlen(name), AF_INET);
  if (!host) {
    if (!(host = gethostbyname(name))) {  perror("gethostbyname");
      return(-1);    }
  }
  bcopy(host-h_addr, (void *) ip_addr, host-h_length);
  return 1;
```

```
}
int connect_to_client(char *comp_name)
{
  struct sockaddr_in server_addr;
  int sock;
  unsigned long ip_addr;

  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock  0) {
    perror("socket");
    exit(-1);
  }
  if (!(inet_addr(comp_name, &ip_addr))) {
    printf("Ne znam za host\n");    exit(-1);
  }
  bzero((char *) &server_addr, sizeof(server_addr));
  server_addr.sin_port = htons(TCP_PORT);
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = ip_addr;
  if (connect(sock, (struct sockaddr *) &server_addr,
      sizeof(server_addr))  0)
  {
    perror("connect");    exit(-1);
  }  return(sock);
}
void disconnect_from_client(int socket)
{
  close(socket);
}

int generisi_drvo(double verovatnoca)
{
  int ver, hlp;
  ver = verovatnoca * 100;
  hlp = (int) ((double) 100.0 * rand()/(RAND_MAX + 1.0));
  if (hlp  ver)    return(1);
  else    return(0);
}

void generisi_sumu(double verovatnoca)
{
  int i, j;
  srand(time(NULL));
  for (i = 0; i  N; i++)
    for (j = 0; j  N; j++)
      suma[i][j] = generisi_drvo(verovatnoca);
}

int najmanje_opterecen()
{
  int i, min_thread, node = 0;
```

```
      min_thread = resurs[0].broj_threadova;
      for (i = 1; i  broj_racunara; i++) {
        if (min_thread  resurs[i].broj_threadova) {
          node = i;
          min_thread = resurs[i].broj_threadova;    }
      }
      return(node);
    }

    int u_sumi(int x, int y)
    {
      return(x = 0 && x  N && y = 0 && y  N);
    }

    void zapali_zonu(int id_zone)
    {
      int k, s, i, j, x;


      k = najmanje_opterecen();
      zona[id_zone].socket = connect_to_client(resurs[k].ime_racu-
nara);
      zona[id_zone].gori = TRUE;
      resurs[k].broj_threadova++;
      zona[id_zone].resurs = k;  s = send(zona[id_zone].socket, (void
*) &zona[id_zone].duzina,
               sizeof(int), 0);
      s = send(zona[id_zone].socket, (void *) &zona[id_zone].sirina,
               sizeof(int), 0);
      for (i = 0; i  zona[id_zone].duzina; i++) {
        for (j = 0; j  zona[id_zone].sirina; j++) {
         x = suma[zona[id_zone].startx + i][zona[id_zone].starty + j];

          s = send(zona[id_zone].socket, &x, sizeof(int), 0);    }
      }
      gori_zona++;
    }

    void ugasi_zonu(int id_zone)
    {
      int i, j, r, x, m, n;
      r = recv(zona[id_zone].socket, (void *) &m, sizeof(int), 0);
      r = recv(zona[id_zone].socket, (void *) &n, sizeof(int), 0);
      for (i = 0; i  m; i++)
        for (j = 0; j  n; j++) {
          r = recv(zona[id_zone].socket, (void *) &x, sizeof(int), 0);
         suma[zona[id_zone].startx + i][zona[id_zone].starty + j] = x;
        }
      disconnect_from_client(zona[id_zone].socket);
      zona[id_zone].gori = FALSE;
      resurs[zona[id_zone].resurs].broj_threadova--;
      gori_zona--;
```

```
    }

    void sinhronizuj_kordinate(int id_zone, int x, int y)
    {
      int stvarno_x, stvarno_y, i, j, id;
      /* zona je izgorela */
      if (x == 12 && y == 12) {
        ugasi_zonu(id_zone);    return;
      }
      stvarno_x = zona[id_zone].startx + x;
      stvarno_y = zona[id_zone].starty + y;
      if (!u_sumi(stvarno_x, stvarno_y))
        return;
      i = stvarno_x / (N / 4);
      j = stvarno_y / (N / broj_racunara);
      id = i * broj_racunara + j;
      /* zona ne gori */
      if (zona[id].gori == FALSE) {
        if (suma[stvarno_x][stvarno_y] == 1) {
          suma[stvarno_x][stvarno_y] = ZAPALJENO;
          zapali_zonu(id);
        }    return;
      }
      zona[id].queue[zona[id].qs].x = stvarno_x % (N / 4);
      zona[id].queue[zona[id].qs].y = stvarno_y % (N / broj_racunara);
      zona[id].qs++;
    }

    void simulacija_perkolacije(double gustina, int *vreme)
    {
      int vre = 0;
      int i, j, r, qs, x, y;
      FILE *fp;

      fp = fopen("input.txt", "r");
      fscanf(fp, "%d", &broj_racunara);
      for (i = 0; i  broj_racunara; i++) {
        fscanf(fp, "%s", resurs[i].ime_racunara); resurs[i].broj_thre-
adova = 0;
      }
      fclose(fp);
      generisi_sumu(gustina);
      for (i = 0; i  N/4; i++)
        for (j = 0; j  N; j++)
          if (suma[i][j] == 1)
            suma[i][j] = ZAPALJENO;
      broj_zona = 4 * broj_racunara;
      for (i = 0; i  4; i++)
        for (j = 0; j  broj_racunara; j++) {
          zona[i * broj_racunara + j].startx = i * N / 4;
          zona[i * broj_racunara + j].starty = j * N / broj_racunara;
          zona[i * broj_racunara + j].duzina = N / 4;
```

```
          zona[i * broj_racunara + j].sirina = N / broj_racunara;
          zona[i * broj_racunara + j].gori = FALSE;
           zona[i * broj_racunara + j].socket = 0;       zona[i *
broj_racunara + j].resurs = 0;
        }
      for (i = 1; i  4; i++)
        zona[i * broj_racunara - 1].sirina += N % broj_racunara;
      for (i = 0; i  broj_racunara; i++)
        zapali_zonu(i);  /* distributivni server */  while (1) {
        if (gori_zona == 0)
          break;
        ++vre;
        for (i = 0; i  broj_zona; i++)
          zona[i].qs = 0;
        for (i = 0; i  broj_zona; i++) {
          if (zona[i].gori == TRUE) {
            r = recv(zona[i].socket, (void *) &qs, sizeof(int), 0);
            for (j = 0; j  qs; j++) {
              r = recv(zona[i].socket, (void *) &x, sizeof(int), 0);
               r = recv(zona[i].socket, (void *) &y, sizeof(int), 0);
sinhronizuj_kordinate(i, x, y);
            }
          }    }
        for (i = 0; i  broj_zona; i++) {
          if (zona[i].gori == TRUE) {
           r = send(zona[i].socket, (void *) &zona[i].qs, sizeof(int),
0);
            for (j = 0; j  zona[i].qs; j++) {
              r = send(zona[i].socket, (void *) &zona[i].queue[j].x,
                       sizeof(int), 0);
              r = send(zona[i].socket, (void *) &zona[i].queue[j].y,
                       sizeof(int), 0);
            }
          }
        }  }
      *vreme = vre;
    }
```

# Prilog 2. Izvorni kod klijent dela simulatora PARPER

```
#include  <stdio.h>
#include  <stdlib.h>h
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <unistd.h>
#include <string..h>
#include  <signal.h>
#include  <pthread.h>
#include <fcntl..h>
#include  <linux/unistd.h>
#include  <time.h>

#define MAX_CLI 30
#define TCP_PORT 5983#define N 100
#define ZAPALJENO 2
#define TIME_LIMIT 7

int sockfd;
void *perkolacija(void *arg);

void server(void)
{
  int accept_sock, clilen;
  struct sockaddr_in cli_addr, serv_addr;
  pthread_t child_thread;

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd  0) {
    perror("socket");
    exit(-1);
  }
  memset((char *) &serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port = htons(TCP_PORT);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
           sizeof(serv_addr))  0)
  {
    perror("bind");
    exit(-1);
  }
  listen(sockfd, MAX_CLI);
  while (1) {
    clilen = sizeof(cli_addr);
    accept_sock = accept(sockfd, (struct sockaddr *) &cli_addr,
                         &clilen);
    if (accept_sock  0) {
      perror("accept");
```

```
      exit(-1);
    }
    pthread_create(&child_thread, NULL, &perkolacija,
                   (void *) accept_sock);
  }
}

void *perkolacija(void *arg)
{
  int socket = (int) arg;
  int suma[N][N], v[N][N], vre = 0;
  int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};
  int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};
  int broj_stabala = 0;
  struct kord_t {
    int x, y;
  } queue[4 * N];
  int qs, x, y, s, r, i, j, pos, m, n;

  r = recv(socket, (void *) &m, sizeof(int), 0);
  r = recv(socket, (void *) &n, sizeof(int), 0);
  for (i = 0; i  m; i++)
    for (j = 0; j  n; j++)
      r = recv(socket, (void *) &suma[i][j], sizeof(int), 0);
  for (i = 0; i  m; i++)
    for (j = 0; j  n; j++) {
      if (suma[i][j] == ZAPALJENO) {
        v[i][j] = 1;
        broj_stabala++;      }
      else
        v[i][j] = 0;
    }
  srand(time(NULL));
  /* evolucija sistema */  while (1) {
    vre++;
    if (broj_stabala == 0)
      break;    qs = 0;    for (i = 0; i  m; i++)
      for (j = 0; j  n; j++) {
        if (suma[i][j] == ZAPALJENO) {
          if (v[i][j] == TIME_LIMIT) {
            suma[i][j] = 0;
            v[i][j] = -1;
            broj_stabala--;
          }
          else {
            pos = (int) (8.0 * rand() / (RAND_MAX + 1.0));
            if (0  i + dx[pos] && i + dx[pos]  m &&
                0  j + dy[pos] && j + dy[pos]  n)
            {
              if (suma[i + dx[pos]][j + dy[pos]] == 1 &&
                  v[i + dx[pos]][j + dy[pos]] == 0)              {
                suma[i + dx[pos]][j + dy[pos]] = ZAPALJENO;
```

```
                    v[i + dx[pos]][j + dy[pos]] = 1;
                    broj_stabala++;
                  }
                }
                else {
                  queue[qs].x = i + dx[pos];
                  queue[qs].y = j + dy[pos];
                  qs++;
                }
                v[i][j]++;          }
            }
          }
        s = send(socket, (void *) &qs, sizeof(int), 0);
        for (i = 0; i  qs; i++) {
          s = send(socket, (void *) &queue[i].x, sizeof(int), 0);
          s = send(socket, (void *) &queue[i].y, sizeof(int), 0);
        }
        r = recv(socket, (void *) &qs, sizeof(int), 0);    for (i = 0;
i  qs; i++) {
          r = recv(socket, (void *) &x, sizeof(int), 0);
          r = recv(socket, (void *) &y, sizeof(int), 0);
          if (suma[x][y] == 1) {
            suma[x][y] = ZAPALJENO;
            v[i][j] = 1;
            broj_stabala++;
      }
        }
      }  qs = 1;
      s = send(socket, (void *) &qs, sizeof(int), 0);
      qs = 12;
      s = send(socket, (void *) &qs, sizeof(int), 0);  s = send(socket,
(void *) &qs, sizeof(int), 0);
      s = send(socket, (void *) &m, sizeof(int), 0);
      s = send(socket, (void *) &n, sizeof(int), 0);
      for (i = 0; i  m; i++)
        for (j = 0; j  n; j++)
          s = send(socket, (void *) &suma[i][j], sizeof(int), 0);
      close(socket);
      pthread_exit(NULL);
    }

    int main(void)
    {
      server();
      return(0);
    }
```