UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Igor Butinar

Paralelizacija drevesnega preiskovanja Monte Carlo

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2013

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Igor Butinar

Paralelizacija drevesnega preiskovanja Monte Carlo

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Branko Šter

Ljubljana, 2013

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00530 / 2013
Datum: 14.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **IGOR BUTINAR**

Naslov: **PARALELIZACIJA DREVESNEGA PREISKOVANJA MONTE CARLO**
PARALLELIZATION OF MONTE CARLO TREE SEARCH

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Opišite komunikacijski protokol MPI (Message Passing Interface), ki se uporablja za paralelne implementacije časovno zahtevnih algoritmov. Z uporabo MPI paralelizirajte algoritem drevesnega preiskovanja Monte Carlo. Implementirajte, preizkusite in eksperimentalno ovrednotite dve različni paralelni izvedbi - na nivoju iger in na nivoju simulacij.

Mentor:

izr. prof. dr. Branko Šter

Dekan:

prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Igor Butinar, z vpisno številko **63060029**, sem avtor diplomskega dela z naslovom:

Paralelizacija drevesnega preiskovanja Monte Carlo

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Branka Štera
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 11. 12. 2013 **Podpis avtorja:**

ZAHVALA

Zahvaljujem se mentorju, profesorju dr. Branku Šteru, ter asistentu Tomu Vodopivcu za vso pomoč in nasvete pri izdelavi te diplomske naloge. Prav tako bi se rad zahvalil ženi Katji in svojim staršem za vso podporo in potrpljenje med izdelavo tega diplomskega dela.

Kazalo

Slike

Tabele

POVZETEK

Glavni cilj te diplomske naloge je paralelizacija programa, katerega jedro je algoritem UCT in je namenjen igranju različnih iger. Program je spisan v programskem jeziku C++, dobil pa sem ga v Laboratoriju za adaptivne sisteme in paralelno procesiranje (LASPP), katerega član je moj mentor pri tej diplomski nalogi.

Za paralelizacijo sem uporabil vmesnik MPI. Zaradi kompletnosti sem opisal njegove bistvene lastnosti, dele in pojme, ki jih srečujemo pri njegovi uporabi (kot npr. pošiljanje in sprejemanje sporočil, skupinska komunikacija, komunikator, skupina, proces, podatkovni tip, enostranska komunikacija, …), skupaj z večino njegovih funkcij ter navedel nekatere primere njihove uporabe v programskem jeziku C.

Predstavil sem družino algoritmov MCTS, katere del je algoritem UCT, skupaj z nekaterimi izboljšavami in razširitvami.

Na koncu sem izvedel tudi nekaj meritev, kjer sem opazoval in analiziral pohitritve, ugotavljal, katera paralelizacija prinese boljše rezultate igre glede na porabljeni čas, …

Ključne besede: MPI, paralelizacija programske kode, paralelno izvajanje programa, MCTS, UCT, algoritmi za igranje iger.

ABSTRACT

The main objective of this graduation thesis is the parallelization of the program, the core of which is the UCT algorithm and is designed for playing various games. The program is written in the C++ programming language. I got the original program from the Laboratory for Adaptive Systems and Parallel Processing (LASPP), where my mentor on this thesis is a member.

I used the MPI interface to parallelize the program. Because of completeness I described the essential features, components and concepts that are used by MPI (e.g. sending and receiving messages, collective communication, communicator, group, process, data type, one-sided communication, etc.). I also described most of MPI functions and provided some examples of their use in the C programming language.

I described the MCTS family of algorithms, of which the UCT algorithm is a part of, together with some improvements and extensions.

In the end I performed measurements, where I observed and analyzed the performance improvements and identified which parallel implementation yields better results regarding games with the processing time as the key metric.

Key words: MPI, parallelization of programming code, parallel implementation of program, MCTS, UCT, algorithms for playing games.

1 UVOD

Paralelizacija programske kode in njeno vzporedno izvajanje je gotovo eno izmed področij računalništva, ki veliko obetajo. Vendar je pisanje programske kode, ki bi se izvajala vzporedno, težavnejše, saj je pri tem treba upoštevati še dostop do morebitnih skupnih virov, hitrost izvajanja delov kode, ki tečejo vzporedno, je različna in nepredvidljiva, vpeljati je treba sinhronizacijo med paraleliziranimi deli kode, … Poleg tega je veliko problemov, ki se jih ne da paralelizirati. Glede na vse to ni čudno, da so do nedavnega proizvajalci procesorjev samo povečevali takt ure in s tem pridobivali na procesorski moči. Vendar pa zaradi čisto fizikalnih vzrokov takta ure ne moremo povečevati v nedogled in razvoj procesorjev se je obrnil v smer večjega števila jeder v enem procesorju. Če so bili še pred 10 leti večjedrni procesorji večinoma namenjeni dražjim in kompleksnejšim sistemom (npr. strežniki, zmogljiva omrežna stikala in usmerjevalniki, superračunalniki, …), jih dandanes srečamo že na skoraj vsakem koraku (v domačih računalnikih, igralnih konzolah, pametnih prenosnih telefonih, tabličnih računalnikih, avtomobilih, …).

Obstaja kar nekaj vmesnikov (knjižnic), ki omogočajo paralelizacijo programske kode (npr. MPI, OpenMP, POSIX Threads, OpenHMPP, …). Nekateri so namenjeni izključno paralelizaciji programske kode, ki teče vzporedno na enem večjedrnem procesorju, drugi (kot npr. MPI) pa omogočajo paralelizacijo na višjem nivoju. To pa ne pomeni, da deli programa MPI ne morejo istočasno teči na večjedrnem procesorju.

Področje algoritmov MCTS je, prav tako kot paralelno procesiranje, izredno hitro rastoče. Algoritmi MCTS se uporabljajo predvsem na področju umetne inteligence v povezavi z igranjem klasičnih namiznih iger in vsebujejo dele, ki jih je možno paralelizirati.

2 VMESNIK ZA POŠILJANJE SPOROČIL

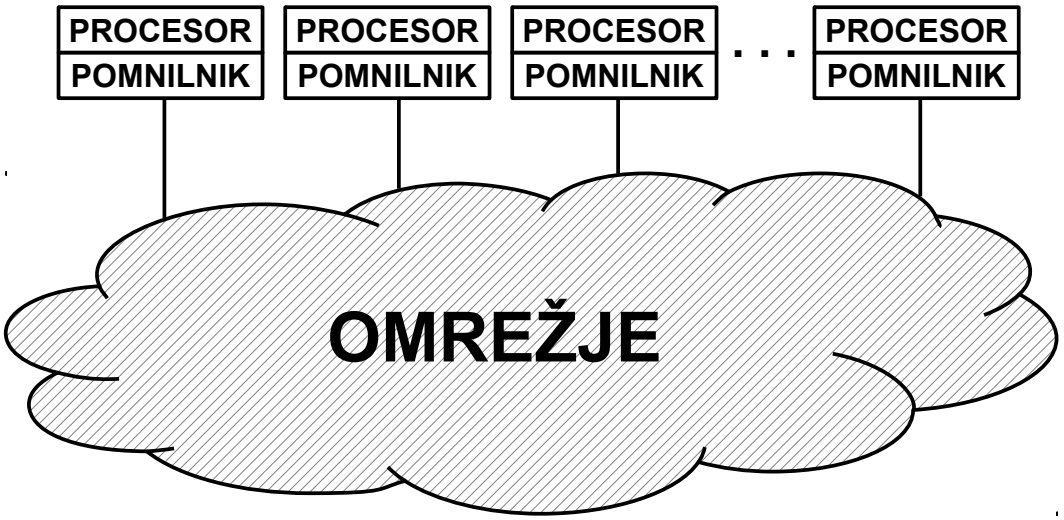
Vmesnik za pošiljanje sporočil (angl. *Message Passing Interface*, MPI) je v svojem bistvu specifikacija. Implementacija MPI v obliki knjižnice (angl. *library*) predstavlja razširitev funkcionalnosti programskega jezika. Obstaja več implementacij, tako odprtokodnih kot komercialnih, za različne programske jezike (C, C++, Fortran, Pyton, Java, …). Med samimi implementacijami so določene razlike (npr. neka implementacija za programski jezik C ima nekatere osnovne tipe MPI, ki jih druga implementacija za programski jezik Fortran nima, in obratno, nekateri parametri funkcije MPI v implementaciji za programski jezik C se lahko razlikujejo od parametrov iste funkcije MPI v implementaciji za programski jezik Fortran, …), vendar so te razlike pogojene s samo specifikacijo programskega jezika, za katerega je implementacija izvedena. Bistvene značilnosti MPI so med posameznimi implementacijami enake.

Obstajajo tri glavne specifikacije MPI: MPI-1, MPI-2 in MPI-3. MPI-2 ima v primerjavi z MPI-1 dodane določene koncepte, kot so npr. enostranska komunikacija, delo z vhodom in izhodom, dinamično dodajanje procesov, … MPI-3 podpira neblokirajočo skupinsko komunikacijo, ima v primerjavi z MPI-2 močno izboljšano enostransko komunikacijo, vezi C++ (angl. *C++ bindings*) so v celoti odstranjene, …

Za potrebe te diplomske naloge sem izbral prostodostopno implementacijo MPICH 2.2, ki vsebuje knjižnice za programske jezike C, C++ in Fortran. Vse opisane funkcije v tem diplomskem delu so del specifikacije MPI 2.2, ki se od osnovne specifikacije MPI-2 razlikuje le v nekaterih malenkostih. Primeri programov MPI v tem diplomskem delu so spisani v programskem jeziku C. Na koncu vsakega primera je tudi eden izmed možnih izpisov (posamezne vrstice se lahko izpišejo pred drugimi, odvisno v kakšnem vrstnem redu procesi dosežejo istoležne funkcije printf).

MPI lahko uporabimo za pisanje programov, katerih deli med seboj komunicirajo s sporočili. Deli programa MPI se izvajajo vzporedno na več rahlo sklopljenih računalnikih, pri čemer imajo lahko različni računalniki različno arhitekturo, lahko pa se vzporedno izvajajo tudi na istem računalniku, če gre za večprocesorski sistem ali večjedrni procesor. Rahlo sklopljen računalnik ima svoj procesor in svoj pomnilnik, z drugimi računalniki pa komunicira preko omrežja, kakor je prikazano na sliki 1. MPI lahko na enem računalniku generira več procesov, pri čemer se vsak proces obnaša tako, kot da bi tekel na svojem računalniku (vsak proces ima dodeljen svoj kos pomnilnika, ki ga drugi procesi ne vidijo). Zato je namesto o skupini

računalnikov pravilneje govoriti o skupini procesov, ki si s pomočjo vmesnika MPI med seboj izmenjujejo sporočila.



Slika 1: Rahlo sklopljeni računalniki

Vsak proces, ki je del programa MPI, je enakovreden drugim procesom. Programer sam določi potek pošiljanja in/ali sprejemanja sporočil, tako da ne moremo govoriti o strežniku (angl. *server*) in odjemalcu (angl. *client*), razen v primeru vzpostavitve povezave preko vrat (angl. *port*). Programerju ni treba skrbeti, kako bo sporočilo prispelo do cilja, saj za to poskrbi MPI.

2.1 STRUKTURA PROGRAMA MPI

Vsak program MPI v programskem jeziku C je videti zelo podobno kot navaden program za jezik C. Programer uporablja vmesnik MPI tako, da kliče njegove funkcije. Vsaka funkcija MPI se začne z `MPI_`.

Zgradba programa MPI je videti nekako takole:

```
#include "mpi.h"
// druge deklaracije

int main(int argc, char* argv[]) {
  // koda, ki jo izvajajo vsi procesi, ki jih je generiral MPI
  MPI_Init(&argc, &argv);

  /* tu programer s pomocjo funkcij MPI upravlja s posameznimi procesi,
     posilja in sprejema sporocila med procesi ... */

  MPI_Finalize();
```

```
  // koda, ki jo izvajajo vsi procesi, ki jih je generiral MPI

  return 0;
}
```

2.2 POGANJANJE PROGRAMOV MPI

Vsak program MPI moramo najprej prevesti s pomočjo prevajalnika za določen programski jezik, ki mu dodamo knjižnico MPI. Večina implementacij MPI vsebuje ovojni program (angl. *wrapper*), ki kliče prevajalnik z ustreznimi parametri (podane poti do knjižnic, stikala, …). Tako preveden program zaženemo s pomočjo ukaza `mpiexec` oz. `mpirun` v MPI-1. S pomočjo parametrov kreiramo želeno število procesov. Prav tako lahko določimo, na katerih računalnikih v omrežju se bo izvajal naš program.

Primer uporabe ukaza mpiexec, ki kreira 4 procese na istem računalniku:

```
mpiexec -n 4 program_MPI
```

2.3 POŠILJANJE IN SPREJEMANJE SPOROČIL

Sam prenos sporočila MPI izvede tako, da iz dela pomnilnika procesa, ki pošilja sporočilo, prevzame podatke in jih dostavi ciljnemu procesu. Pri pošiljanju in/ali sprejemanju sporočil MPI pozna dva načina, in sicer:

- blokirajoči (angl. *blocking*),
- neblokirajoči (angl. *non-blocking*).

Blokirajoči način je varnejši, saj proces, ki pošilja sporočilo, počaka, da MPI prevzame sporočilo. To pa ne zagotavlja, da je ciljni proces sporočilo sprejel. Če ciljni proces še ni pripravljen na sprejem, lahko sporočilo čaka v sprejemnem medpomnilniku (angl. *buffer*). V primeru sprejemanja sporočila ciljni proces v blokirajočem načinu čaka, dokler ne sprejme sporočila.

Neblokirajoči način je sicer hitrejši, saj proces po predaji zahteve vmesniku MPI in pošiljanju takoj nadaljuje z delom, vendar se pri tem lahko zgodi, da povozi/spremeni podatke, ki jih MPI še ni prevzel in poslal naprej.

2.4 OKOLJE MPI

Funkcije za delo z okoljem MPI se uporabljajo za inicializacijo in zaključitev dela z okoljem MPI, za pridobivanje informacij o okolju MPI, merjenje časa, rezervacijo in sproščanje pomnilnika, …

Naštetih je nekaj funkcij za delo z okoljem MPI:

- `int MPI_Init(int *argc, char ***argv);`
  Funkcija inicializira okolje MPI. Ta funkcija mora biti klicana pred katero koli drugo funkcijo MPI (izjema so funkcije `MPI_Initialized`, `MPI_Finalized` in `MPI_Get_version`).
- `int MPI_Initialized(int *flag);`
  Funkcija preveri, ali je bila funkcija `MPI_Init` že klicana.
- `int MPI_Finalize(void);`
  Funkcija zaključi delo z okoljem MPI. Za to funkcijo ne sme biti klicana nobena druga funkcija MPI (izjema so funkcije `MPI_Initialized`, `MPI_Finalized` in `MPI_Get_version`).
- `int MPI_Finalized(int *flag);`
  Funkcija preveri ali je bila funkcija `MPI_Finalize` že klicana.
- `int MPI_Abort(MPI_Comm comm, int errorcode);`
  Funkcija zaključi vse procese, ki so prisotni v komunikatorju *comm*, in vrne podano kodo napake.
- `int MPI_Get_version(int *version, int *subversion);`
  Funkcija vrne različico in podrazličico implementiranega standarda.
- `int MPI_Get_processor_name(char *name, int *resultlen);`
  Funkcija na naslov *name* zapiše ime procesorja, na katerem se izvaja trenutni proces.
- `int MPI_Get_address(void *location, MPI_Aint *address);`
  Funkcija na naslov *address* zapiše bajtni naslov, na katerega kaže parameter *location*.
- `int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr);`
  Funkcija rezervira *size* bajtov v pomnilniku.
- `int MPI_Free_mem(void *base);`
  Funkcija sprosti del pomnilnika, rezerviranega s funkcijo `MPI_Alloc_mem`.
- `double MPI_Wtick(void);`
  Funkcija vrne čas (v sekundah), ki preteče med dvema zaporednima pulzoma

časovnika (angl. *tick timer*).

- `double MPI_Wtime(void);`

  Funkcija vrne čas (v sekundah), ki je pretekel od naključno izbranega trenutka t_0 v preteklosti (t_0 se ne spreminja, dokler proces živi). Atribut *MPI_WTIME_IS_GLOBAL* nam pove, ali je čas t_0 enak za vse procese.

Primer uporabe nekaterih zgoraj opisanih funkcij:

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int ver, subver, flag, cnt;
  char name[MPI_MAX_PROCESSOR_NAME];
  double time;
  void *ptr;
  MPI_Aint addr;

  MPI_Get_version(&ver, &subver);
  printf("MPI verzija: %d.%d\n", ver, subver);

  MPI_Initialized(&flag);
  if (!flag)
    MPI_Init(&argc, &argv);

  time = MPI_Wtime();

  MPI_Alloc_mem((MPI_Aint)12, MPI_INFO_NULL, &ptr);
  MPI_Get_address(ptr, &addr);
  printf("Naslov, kamor kaze kazalec ptr: 0x%x\n", addr);

  MPI_Get_processor_name(&name[0], &cnt);
  printf("Ime procesorja: %s\n", name);

  printf("Pretecen cas: %f ms\n", (MPI_Wtime() - time)*1000);

  MPI_Free_mem(ptr);
  MPI_Finalized(&flag);
  if (!flag)
    MPI_Finalize();

  return 0;
}
```

Eden izmed možnih izpisov, če program poženemo na npr. 2 procesih:

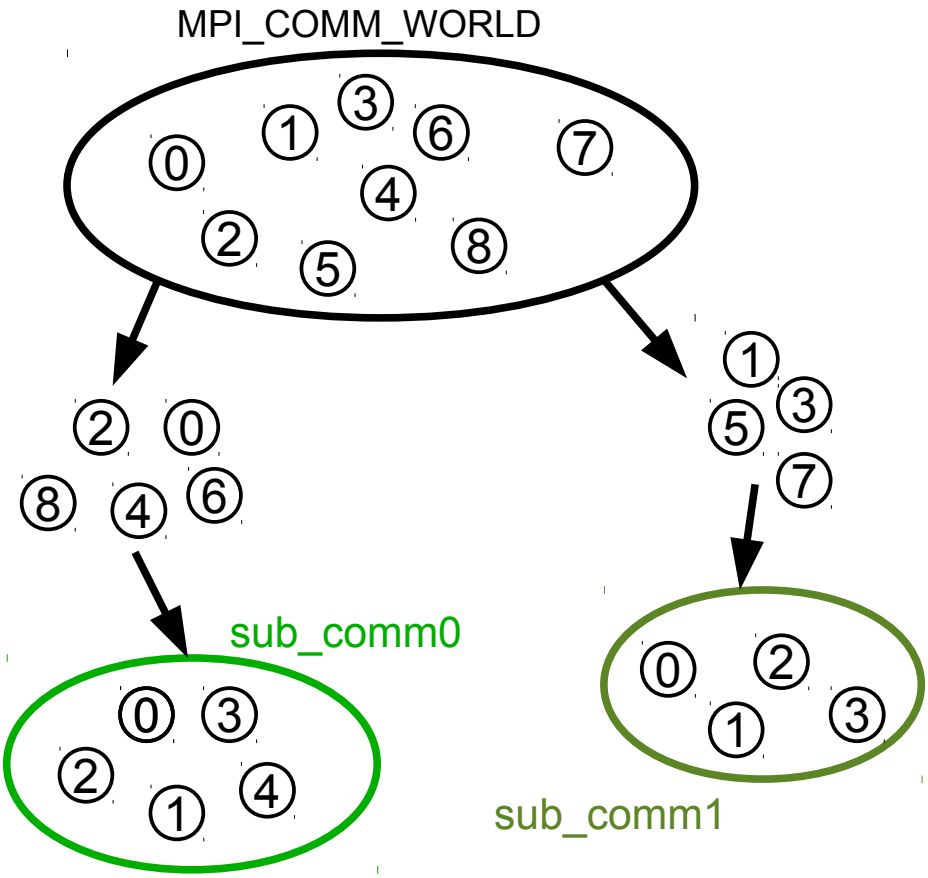
```
MPI verzija: 2.2
Naslov, kamor kaze kazalec ptr: 0x9d4848
Ime procesorja: toncy
Pretecen cas: 0.018997 ms
MPI verzija: 2.2
Naslov, kamor kaze kazalec ptr: 0x9d4888
Ime procesorja: toncy
Pretecen cas: 0.021232 ms
```

2.5 KOMUNIKATORJI IN SKUPINE

Komunikator (angl. *communicator*) je objekt, ki definira skupine procesov, ki lahko komunicirajo med seboj. MPI ima že nekaj vnaprej definiranih komunikatorjev:

- *MPI_COMM_WORLD* – vsebuje vse procese, kreirane ob zagonu programa MPI,
- *MPI_COMM_SELF* – vsebuje le trenutni proces,
- *MPI_COMM_NULL* – komunikator, ki ne vsebuje nobenega procesa (neveljaven komunikator).

Skupina (angl. *group*) je urejena množica procesov, znotraj katere ima vsak proces svojo edinstveno številko (0, 1, 2, … N-1, kjer je N število procesov v skupini), imenovano položaj (angl. *rank*). Funkcije za delo s skupinami se začnejo z `MPI_Group_`.



Slika 2: Primer kreiranja lastnih skupin in komunikatorjev

Komunikacija lahko poteka le med procesi znotraj istega komunikatorja (tak komunikator imenujemo tudi intrakomunikator) ali pa med procesi različnih komunikatorjev (pri čemer

moramo predhodno kreirati poseben komunikator, imenovan tudi interkomunikator), ne pa med skupinami. Vsako skupino moramo pred kakršno koli komunikacijo med procesi najprej pripojiti določenemu komunikatorju. Če v nadaljevanju besedila ni posebej navedeno, za kateri komunikator gre, se predpostavlja, da gre za intrakomunikator. Funkcije za delo s komunikatorji se tipično začnejo z `MPI_Comm_`.

MPI dopušča kreiranje lastnih skupin in komunikatorjev. Omogoča tudi kreiranje novih atributov komunikatorja. Nove atribute generiramo, brišemo in do njih dostopamo s pomočjo ključev (angl. *keyval*), ki so nekakšni kazalci na atribute. Obstajajo tudi vnaprej definirani ključi atributov, kot so npr.: *MPI_TAG_UB*, *MPI_WTIME_IS_GLOBAL*, *MPI_UNIVERSE_SIZE*, *MPI_WTIME_IS_GLOBAL*, *MPI_IO* …, ki jih ne moremo spreminjati ali brisati. Na sliki 2 je prikazan primer kreiranja lastnih skupin in komunikatorjev.

Nekaj funkcij za delo s komunikatorji in skupinami:

- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  Funkcija na naslov *`rank`* zapiše položaj procesa v komunikatorju *`comm`*.
- `int MPI_Comm_size(MPI_Comm comm, int *size);`
  Funkcija na naslov *`size`* zapiše število vseh procesov v komunikatorju *`comm`*.
- `int MPI_Comm_set_name(MPI_Comm comm, char *comm_name);`
  Funkcija komunikatorju *`comm`* dodeli ime (niz znakov).
- `int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen);`
  Funkcija na naslov *`comm_name`* zapiše ime komunikatorja *`comm`* in na naslov *`resultlen`* število znakov v njegovem imenu.
- `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);`
  Po izvedbi funkcije se na naslovu *`group`* nahaja skupina, ki je pripojena komunikatorju *`comm`*.
- `int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);`
  Funkcija na naslovu *`newcomm`* kreira komunikator in mu dodeli procese iz skupine *`group`*. Skupina *`group`* mora biti podmnožica skupine, ki pripada komunikatorju *`comm`*.
- `int MPI_Comm_free(MPI_Comm *comm);`
  Funkcija sprosti komunikator, ki se nahaja na naslovu *`comm`*, po tem ko se zaključijo vse njegove aktivnosti (vse operacije, ki se še izvajajo).

- `int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);`
  Funkcija naredi kopijo obstoječega komunikatorja *comm*.
- `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);`
  Funkcija za alternativni način kreiranja komunikatorjev. Procesi komunikatorja *comm* se razdelijo na disjunktne množice[1] glede na vrednost parametra *color*, pri čemer vsaka množica tvori novonastali komunikator. Parameter *color* je lahko samo nenegativno število ali konstanta *MPI_UNDEFINED* (v tem primeru je vrednost novonastalega komunikatorja *MPI_COMM_NULL*). Vrednost parametra *key* je položaj procesa v komunikatorju *comm*.
- `int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn, MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval, void *extra_state);`
  Funkcija generira nov ključ, pri čemer se nastavita dve funkciji: *comm_copy_attr_fn* in *comm_delete_attr_fn*. Kličeta se, če kopiramo oz. zbrišemo atribut.
- `int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val);`
  Funkcija nastavi vrednost atributu s pomočjo ključa *comm_keyval*.
- `int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val, int *flag);`
  Če ključ obstaja, funkcija na naslov *flag* zapiše logično vrednost *pravilno* (angl. *true*), na naslov *attribute_val* pa vrednost atributa, dodeljenega ključu *comm_keyval*. Če atribut s ključem *comm_keyval* ne obstaja, funkcija na naslov *flag* zapiše logično vrednost *napačno* (angl. *false*).
- `int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval);`
  Funkcija izbriše atribut, dodeljen ključu *comm_keyval*. Ob tem se kliče funkcija, ki je bila nastavljena ob kreiranju ključa *comm_keyval*.
- `int MPI_Comm_free_keyval(int *comm_keyval);`
  Funkcija sprosti veljaven ključ, ki se nahaja na naslovu *comm_keyval* in na isti naslov zapiše vrednost *MPI_KEYVAL_INVALID*.
- `int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm);`

1 Množice, katerih presečišče je prazna množica.

Funkcija na naslovu *newintercomm* kreira interkomunikator iz dveh intrakomunikatorjev. Parameter *local_leader* je položaj vodilnega procesa v intrakomunikatorju *local_comm*. Parameter *peer_comm* je komunikator, ki vsebuje oba vodilna procesa iz obeh intrakomunikatorjev. Parameter *remote_leader* je položaj vodilnega procesa iz drugega intrakomunikatorja, kot je viden v komunikatorju *peer_comm*. Značka *tag* je unikatna oznaka novonastalega interkomunikatorja.

- `int MPI_Comm_test_inter(MPI_Comm comm, int *flag);`
  S pomočjo te funkcije ugotovimo, ali je komunikator *comm* interkomunikator ali intrakomunikator.
- `int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[]);`
  Funkcija se uporablja za dinamično dodajane procesov. Generira *maxprocs* novih procesov programa *command*, ki mora biti preveden program MPI, z argumenti *argv*. Pri tem funkcija na naslovu *intercomm* kreira nov interkomunikator in za vsak novonastali proces v tabelo *array_of_errcodes* zapiše vrednost *MPI_ERR_SPAWN*, če je prišlo do napake pri zagonu procesa. Vsi novonastali procesi so otroci (angl. *children*) komunikatorja *comm*, ki postane starš (angl. *parent*) tem procesom.
- `int MPI_Comm_get_parent(MPI_Comm *parent);`
  Po izvedbi funkcije se na naslovu *parent* nahaja starševski komunikator. Če se je proces kreiral ob klicu funkcije `MPI_Comm_spawn`, je ime novonastalega interkomunikatorja *MPI_COMM_PARENT;* če je bil proces kreiran ob zagonu MPI programa, pa *MPI_COMM_NULL*.
- `int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup);`
  Funkcija na naslovu *newgroup* iz obstoječe skupine *group* kreira novo skupino, v kateri je *n* procesov. Kateri procesi so del nove skupine, je določeno s tabelo *ranks*, katere elementi so položaji v izvorni skupini *group*.
- `int MPI_Group_free(MPI_Group *group);`
  Ko se vse aktivnosti v skupini, ki se nahaja na naslovu *group*, zaključijo, funkcija sprosti skupino in ji priredi vrednost *MPI_GROUP_NULL*.
- `int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);`
  Funkcija na naslovu *newgroup* kreira novo skupino, ki je sestavljena iz unije

procesov skupin *group1* in *group2*. Položaj procesov je enak kot v prvi skupini, potem po vrsti sledijo še procesi iz skupine *group2*, ki niso del skupine *group1*.

- `int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);`
  Funkcija na naslovu *newgroup* kreira novo skupino, ki je sestavljena iz preseka procesov obeh vhodnih skupin.
- `int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);`
  Funkcija na naslovu *newgroup* kreira novo skupino, sestavljeno iz vseh procesov, ki so del skupine *group1* in niso del skupine *group2*.
- `int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result);`
  Po izvedbi funkcije se na naslovu *result* nahaja ena izmed spodnjih konstant:
  1. *MPI_IDENT*, če imata obe vhodni skupini iste procese in isto razporeditev procesov (položaj vsakega procesa v obeh skupinah je identičen);
  2. *MPI_SIMILAR*, če imata obe vhodni skupini iste procese z različno razporeditvijo procesov;
  3. *MPI_UNEQUAL*, če so procesi vhodnih skupin različni.
- `int MPI_Group_rank(MPI_Group group, int *rank);`
  Funkcija na naslov *rank* zapiše položaj procesa v podani skupini *group* oz. konstanto *MPI_UNDEFINED,* če proces ni del skupine.
- `int MPI_Group_size(MPI_Group group, int *size);`
  Funkcija na naslov *size* zapiše število procesov, ki so del skupine *group*.

Primer uporabe nekaterih funkcij za delo s komunikatorji in skupinami:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i, key, val, flag, err[3];
  int comm_world_id, sub_comm_id;
  int comm_size, sub_comm0_size, sub_comm1_size;
  int *sub_comm0_rank, *sub_comm1_rank;
  void *attr_val;
  char comm_name[MPI_MAX_OBJECT_NAME];
  MPI_Group comm_world_group, sub_group0, sub_group1;
  MPI_Comm sub_comm, sub_comm0, sub_comm1, inter_comm;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  sub_comm1_size = comm_size/2;
  sub_comm0_size = comm_size - sub_comm1_size;
  sub_comm0_rank = malloc(sub_comm0_size * sizeof(int));
```

```
sub_comm1_rank = malloc(sub_comm1_size * sizeof(int));
for (i = 0; i < comm_size; i++) {
  if (i & 1)
    sub_comm1_rank[i/2] = i;
  else
    sub_comm0_rank[i/2] = i;
}

MPI_Comm_rank(MPI_COMM_WORLD, &comm_world_id);
MPI_Comm_get_name(MPI_COMM_WORLD, &comm_name[0], &i);
if (!comm_world_id)
   printf("V komunikatorju %s je %d procesov\n", comm_name, comm_size);

MPI_Comm_group(MPI_COMM_WORLD, &comm_world_group);
MPI_Group_incl(comm_world_group, sub_comm0_size, &sub_comm0_rank[0],
    &sub_group0);
MPI_Group_incl(comm_world_group, sub_comm1_size, &sub_comm1_rank[0],
    &sub_group1);

MPI_Comm_create(MPI_COMM_WORLD, sub_group0, &sub_comm0);
MPI_Comm_create(MPI_COMM_WORLD, sub_group1, &sub_comm1);

MPI_Group_free(&comm_world_group);
MPI_Group_free(&sub_group0);
MPI_Group_free(&sub_group1);

sub_comm = (comm_world_id & 1) ? sub_comm1 : sub_comm0;

MPI_Comm_size(sub_comm, &comm_size);
if (!comm_world_id || (comm_world_id == 1))
   printf("V komunikatorju sub_comm%d je %d procesov\n", comm_world_id & 1,
       comm_size);

MPI_Comm_set_name(sub_comm, comm_world_id & 1 ? "sub_comm1" : "sub_comm0");
MPI_Comm_get_name(sub_comm, &comm_name[0], &i);
MPI_Comm_rank(sub_comm, &sub_comm_id);
printf("Polozaj v komunikatorju MPI_COMM_WORLD: %d; ", comm_world_id);
printf("polozaj v komunikatorju %s: %d\n", comm_name, sub_comm_id);

val = 916351;
MPI_Comm_create_keyval(MPI_NULL_COPY_FN, MPI_NULL_DELETE_FN, &key, (void *)0);
MPI_Comm_set_attr(sub_comm, key, &val);

if (!comm_world_id)
  MPI_Comm_delete_attr(sub_comm, key);

MPI_Comm_get_attr(sub_comm, key, &attr_val, &flag);
if (!comm_world_id || (comm_world_id == 1))
  printf("Kljuc 'key' %s definiran v komunikatorju %s\n", flag ? "je" : "ni",
      comm_name);
if (flag)
  if (!comm_world_id || (comm_world_id == 1))
    printf("Vrednost kljuca 'key' v komunikatorju %s: %d\n", comm_name,
        *((int *)attr_val));

MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &attr_val, &flag);
if (flag)
  if (!comm_world_id)
    printf("Vrednost kljuca 'MPI_TAG_UB' v komunikatorju MPI_COMM_WORLD: %d\n",
        *((int *)attr_val));

MPI_Comm_free_keyval(&key);

MPI_Comm_spawn("mpi_simple.exe", MPI_ARGV_NULL, 3, MPI_INFO_NULL, 1, sub_comm,
    &inter_comm, &err[0]);
```

```
  MPI_Comm_free(&sub_comm);
  MPI_Comm_free(&inter_comm);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  MPI_Comm_split(MPI_COMM_WORLD, (comm_world_id < comm_size/2) ? 0 : 1,
      comm_world_id, &sub_comm);
  MPI_Intercomm_create(sub_comm, 0, MPI_COMM_WORLD,
      comm_world_id ? 0 : comm_size/2, 73, &inter_comm);

  MPI_Comm_set_name(inter_comm, "inter_comm");
  MPI_Comm_set_name(sub_comm, (comm_world_id < comm_size/2) ?
      "new_sub_comm0" : "new_sub_comm1");
  MPI_Comm_get_name(sub_comm, &comm_name[0], &i);
  MPI_Comm_rank(sub_comm, &sub_comm_id);
  printf("Polozaj v komunikatorju MPI_COMM_WORLD: %d; ", comm_world_id);
  printf("polozaj v komunikatorju %s: %d\n", comm_name, sub_comm_id);
  if (!comm_world_id) {
    MPI_Comm comm_array[] = {sub_comm, inter_comm, MPI_COMM_WORLD};
    for (i = 0; i < 3; i++) {
      MPI_Comm_get_name(comm_array[i], &comm_name[0], &val);
      MPI_Comm_test_inter(comm_array[i], &flag);
      printf("Komunikator %s je %s komunikator\n", comm_name,
          flag ? "inter" : "intra");
    }
  }

  MPI_Comm_free(&sub_comm);
  MPI_Comm_free(&inter_comm);

  free(sub_comm0_rank);
  free(sub_comm1_rank);

  MPI_Finalize();

  return 0;
}
```

Če program prevedemo in poženemo na 7 procesih, lahko dobimo izpis:

```
Proces 1/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
Proces 2/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
Proces 0/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
Proces 2/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
Proces 0/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
Proces 1/3 v komunikatorju MPI_COMM_WORLD je potomec komunikatorja MPI_COMM_PARENT
V komunikatorju MPI_COMM_WORLD je 7 procesov
V komunikatorju sub_comm0 je 4 procesov
Polozaj v komunikatorju MPI_COMM_WORLD: 0; polozaj v komunikatorju sub_comm0: 0
Kljuc 'key' ni definiran v komunikatorju sub_comm0
Vrednost kljuca 'MPI_TAG_UB' v komunikatorju MPI_COMM_WORLD: 2147483647
Polozaj v komunikatorju MPI_COMM_WORLD: 0; polozaj v komunikatorju new_sub_comm0: 0
Komunikator new_sub_comm0 je intra komunikator
Komunikator inter_comm je inter komunikator
Komunikator MPI_COMM_WORLD je intra komunikator
Polozaj v komunikatorju MPI_COMM_WORLD: 5; polozaj v komunikatorju sub_comm1: 2
Polozaj v komunikatorju MPI_COMM_WORLD: 5; polozaj v komunikatorju new_sub_comm1: 2
V komunikatorju sub_comm1 je 3 procesov
Polozaj v komunikatorju MPI_COMM_WORLD: 1; polozaj v komunikatorju sub_comm1: 0
Kljuc 'key' je definiran v komunikatorju sub_comm1
Vrednost kljuca 'key' v komunikatorju sub_comm1: 916351
Polozaj v komunikatorju MPI_COMM_WORLD: 1; polozaj v komunikatorju new_sub_comm0: 1
Polozaj v komunikatorju MPI_COMM_WORLD: 6; polozaj v komunikatorju sub_comm0: 3
Polozaj v komunikatorju MPI_COMM_WORLD: 6; polozaj v komunikatorju new_sub_comm1: 3
Polozaj v komunikatorju MPI_COMM_WORLD: 4; polozaj v komunikatorju sub_comm0: 2
```

```
Polozaj v komunikatorju MPI_COMM_WORLD: 4; polozaj v komunikatorju new_sub_comm1: 1
Polozaj v komunikatorju MPI_COMM_WORLD: 2; polozaj v komunikatorju sub_comm0: 1
Polozaj v komunikatorju MPI_COMM_WORLD: 2; polozaj v komunikatorju new_sub_comm0: 2
Polozaj v komunikatorju MPI_COMM_WORLD: 3; polozaj v komunikatorju sub_comm1: 1
Polozaj v komunikatorju MPI_COMM_WORLD: 3; polozaj v komunikatorju new_sub_comm1: 0
```

Program mpi_simple.exe je preveden iz naslednje kode:

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  int id, size, i;
  char str[100] = {'\0'};
  char comm_name[MPI_MAX_OBJECT_NAME];
  MPI_Comm parent_comm;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_get_parent(&parent_comm);
  if (parent_comm != MPI_COMM_NULL) {
    strcpy(&str[0], "je potomec komunikatorja ");
    MPI_Comm_get_name(parent_comm, &comm_name[0], &i);
    strcat(&str[0], &comm_name[0]);
  } else
    strcpy(&str[0], "nima prednikov");
  MPI_Comm_get_name(MPI_COMM_WORLD, &comm_name[0], &i);
  printf("Proces %d/%d v komunikatorju %s %s\n", id, size, comm_name, str);

  MPI_Finalize();

  return 0;
}
```

Če bi pognali program mpi_simple.exe samostojno na npr. 4 procesih, je eden izmed možnih izpisov:

```
Proces 2/4 v komunikatorju MPI_COMM_WORLD nima prednikov
Proces 3/4 v komunikatorju MPI_COMM_WORLD nima prednikov
Proces 0/4 v komunikatorju MPI_COMM_WORLD nima prednikov
Proces 1/4 v komunikatorju MPI_COMM_WORLD nima prednikov
```

2.6 VIRTUALNA (NAVIDEZNA) TOPOLOGIJA

Virtualna (navidezna) topologija nam omogoča, da procese znotraj komunikatorja, ki je lahko le intrakomunikator, uredimo v neko geometrijsko obliko. Tako lahko procese porazdelimo v takšno obliko, da se čim bolj prilegajo dejanski strojni opremi, in s tem pripomoremo k hitrejšemu izmenjevanju sporočil med posameznimi procesi. MPI pozna dva osnovna tipa topologij: graf (angl. *graph*) in *n*-dimenzionalni prostor (angl. *cartesian*). Vsako vozlišče v grafu oz. koordinata *n*-dimenzionalnega prostora predstavlja en proces. Funkcije za delo z virtualno topologijo v obliki grafa se začnejo z `MPI_Graph`, funkcije za delo z virtualno

topologijo v obliki *n*-dimenzionalnega prostora pa z `MPI_Cart`.

Nekaj funkcij za delo z virtualno topologijo v obliki grafa:

- `int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph);`

  Funkcija na naslovu *`comm_graph`* kreira nov komunikator, v katerem so procesi razvrščeni v obliki grafa z *`nnodes`* vozlišči. Parameter *`index`* je tabela, ki vsebuje informacije o številu sosedov, parameter *`edges`* pa vsebuje informacije o povezavah do sosednih vozlišč. Če je število elementov novooblikovanega grafa manjše od števila vseh procesov v komunikatorju *`comm_old`*, se novonastalemu komunikatorju v procesih, ki niso del grafa, priredi vrednost *MPI_COMM_NULL*.

- `int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges);`

  Funkcija popiše dve tabeli. V tabeli *`nnodes`* se nahajajo informacije o številu sosednih vozlišč, v tabeli *`edges`* pa informacije o povezavah do sosednih vozlišč.

- `int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges);`

  S pomočjo te funkcije pridobimo informacije o številu sosednih vozlišč in povezavah do njih. Parameter *`maxindex`* predstavlja velikost tabele *`index`*, parameter *`maxedges`* pa velikost tabele *`edges`*.

- `int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors);`

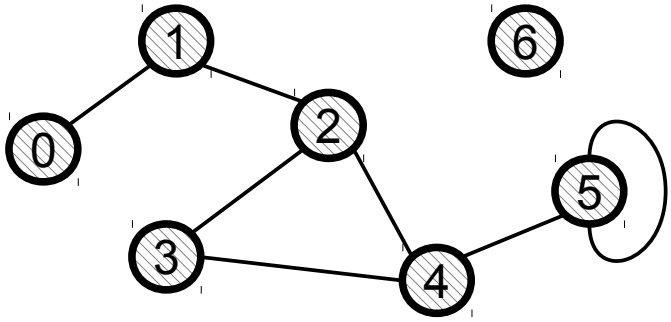
  Funkcija na naslov *`nneighbors`* zapiše število sosedov vozlišča *`rank`*.

- `int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors);`

  Funkcija popiše *`maxneighbors`* elementov tabele *`neighbors`* z informacijami o sosednih vozliščih.

- `int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges, int *newrank);`

  Funkcija preslika procese komunikatorja *`comm`* v virtualno topologijo grafa z *`nnodes`* vozlišči. Informacije o sosednih vozliščih in povezavami z njimi dobi v tabelah *`index`* in *`edges`*. Funkcija na naslov *`newrank`* zapiše nov položaj procesa, ki je lahko enak položaju v komunikatorju *`comm`*. Če proces ne postane del grafa, funkcija na naslov *`newrank`* zapiše vrednost *MPI_UNDEFINED*.



Slika 3: Primer razvrstitve procesov v virtualno topologijo grafa

Primer uporabe nekaterih funkcij za delo z virtualno topologijo v obliki grafa (procesi so razvrščeni v virtualno topologijo grafa, kot je prikazano na sliki 3):

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int id, size;
  int in_index[] = {1, 3, 6, 8, 11, 13, 13};
  int in_edges[] = {1, 0, 2, 1, 3, 4, 2, 4, 2, 3, 5, 4, 5};
  MPI_Comm comm_graph;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (size < 7)
    MPI_Abort(MPI_COMM_WORLD, 1);

  MPI_Graph_create(MPI_COMM_WORLD, 7, &in_index[0], &in_edges[0], 0, &comm_graph);
  if (comm_graph == MPI_COMM_NULL) {
    printf("Proces s polozajem %d ", id);
    printf("ni del novega komunikatorja s topologijo grafa\n");
  } else {
    int nnodes, nedges, i, nneighbors;
    int *index, *edges;

    MPI_Graphdims_get(comm_graph, &nnodes, &nedges);
    index = (int *)malloc(nnodes * sizeof(int));
    edges = (int *)malloc(nedges * sizeof(int));

    if (!id)
      printf("Graf ima %d vozlisc\n", nnodes);

    MPI_Graph_get(comm_graph, nnodes, nedges, &index[0], &edges[0]);
    nneighbors = index[id] - (id ? index[id-1] : 0);
    printf("Vozlisce %d ima %d sosedov%s", id, nneighbors,
        !nneighbors ? "\n" : ": ");

    if (id & 1) // podatki o sosednih vozliscih pridobljeni iz tabel index & edges
      for (i = 0; i < nneighbors; i++)
        printf("%d%s", edges[i+(id ? index[id-1] : 0)],
           (i == nneighbors-1) ? "\n" : ", ");
    else { // podatki o sosednih vozliscih pridobljeni z MPI funkcijami
      int *neighbors;
      MPI_Graph_neighbors_count(comm_graph, id, &nneighbors);
```

```
      neighbors = (int *)malloc(nneighbors * sizeof(int));
      MPI_Graph_neighbors(comm_graph, id, nneighbors, &neighbors[0]);
      for (i = 0; i < nneighbors; i++)
        printf("%d%s", neighbors[i], (i == nneighbors-1) ? "\n" : ", ");
      free(neighbors);
    }

    MPI_Comm_free(&comm_graph);
    free(index);
    free(edges);
  }

  MPI_Finalize();

  return 0;
}
```

Če zgornji program prevedemo in poženemo na npr. 8 procesih, lahko dobimo izpis:

```
Graf ima 7 vozlisc
Vozlisce 0 ima 1 sosedov: 1
Vozlisce 1 ima 2 sosedov: 0, 2
Vozlisce 4 ima 3 sosedov: 2, 3, 5
Vozlisce 2 ima 3 sosedov: 1, 3, 4
Vozlisce 3 ima 2 sosedov: 2, 4
Proces s polozajem 7 ni del novega komunikatorja s topologijo grafa
Vozlisce 6 ima 0 sosedov
Vozlisce 5 ima 2 sosedov: 4, 5
```

Nekaj funkcij za delo z virtualno topologijo v obliki *n*-dimenzionalnega prostora:

- `int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);`
  Funkcija na naslovu *comm_cart* kreira nov komunikator, v katerem so procesi razvrščeni v obliki *n*-dimenzionalnega prostora. Parameter *ndims* poda število dimenzij. Tabeli *dims* (velikost posamezne dimenzije) in *periods* (periodičnost posamezne dimenzije) morata vsebovati *ndims* elementov. Če kot parameter *reorder* podamo logično vrednost *pravilno*, nam funkcija lahko prerazporedi položaje procesov. Če je število elementov novega *ndims*-dimenzionalnega prostora manjše od števila vseh procesov v komunikatorju *comm_old*, se novonastalemu komunikatorju *comm_cart* v procesih, ki niso del novega *ndims*-dimenzionalnega prostora, priredi vrednost *MPI_COMM_NULL*.
- `int MPI_Cartdim_get(MPI_Comm comm, int *ndims);`
  Funkcija na naslov *ndims* zapiše število dimenzij *n*-dimenzionalnega prostora.
- `int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords);`
  S pomočjo te funkcije pridobimo podatke o virtualni topologiji *n*-dimenzionalnega prostora. Kot vhodni parameter *maxdims* podamo število dimenzij, ki predstavlja število elementov tabel *dims* (velikosti posameznih dimenzij), *periods* (logične

vrednosti, ki nam povedo, ali je dimenzija periodična ali ne) in *coords* (koordinate procesa, ki je klical funkcijo).

- `int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);`
  Funkcija na naslov *rank* zapiše položaj procesa na koordinatah *coords*.
- `int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords);`
  S pomočjo te funkcije pridobimo koordinate procesa na položaju *rank*.
- `int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *comm_new);`
  Funkcija razdeli *n*-dimenzionalni prostor v podprostore. Elementi tabele *remain_dims* določajo, ali dimenzija ostane v podprostoru (logična vrednost *pravilno*) ali ne (logična vrednost *napačno*).
- `int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank);`
  Funkcija razporedi procese komunikatorja *comm* v virtualno topologijo *ndims*-dimenzionalnega prostora, pri čemer so velikosti posameznega prostora podane v tabeli *dims*. Funkcija na naslov *newrank* zapiše nov položaj procesa, ki je lahko enak položaju v komunikatorju *comm*. Če proces ne postane del nove virtualne topologije *ndims*-dimenzionalnega prostora, funkcija na naslov *newrank* zapiše vrednost *MPI_UNDEFINED*.

Primer uporabe zgornjih funkcij:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int id, cart_id, size;
  int dims[2] = {3, 2};
  int periods[2] = {0, 1};
  MPI_Comm comm_cart;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (size < 3*2)
    MPI_Abort(MPI_COMM_WORLD, 1);

  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_cart);

  if (comm_cart == MPI_COMM_NULL) {
    printf("Proces s polozajem %d ", id);
    printf("ni del novega komunikatorja s topologijo N d prostora\n");
  } else {
    int ndims, i;
    int remain_dims[2] = {1, 0};
    int *dim, *period, *coord;
```

```
    MPI_Comm comm_cart_sub;

    MPI_Cartdim_get(comm_cart, &ndims);
    dim = (int *)malloc(ndims * sizeof(int));
    period = (int *)malloc(ndims * sizeof(int));
    coord = (int *)malloc(ndims * sizeof(int));

    MPI_Cart_get(comm_cart, ndims, &dim[0], &period[0], &coord[0]);
    MPI_Cart_rank(comm_cart, &coord[0], &cart_id);
    if (!cart_id) {
      printf("Topologija ima %d dimenzij - ", ndims);
      for (i = 0; i < ndims; i++)
        printf("%d%s", dim[i], (i == ndims-1) ? "\n" : "x");
    } else
      MPI_Cart_coords(comm_cart, cart_id, ndims, &coord[0]);

    printf("Proces s polozajem %d v MPI_COMM_WORLD ", id);
    printf("ima nov polozaj %d na poziciji (", cart_id);
    for (i = 0; i < ndims; i++)
      printf("%d%s", coord[i], (i == ndims-1) ?
          ") v novem komunikatorju s topologijo N d prostora\n" : ", ");

    MPI_Cart_sub(comm_cart, &remain_dims[0], &comm_cart_sub);

    MPI_Comm_free(&comm_cart);
    free(dim);
    free(period);
    free(coord);

    MPI_Cartdim_get(comm_cart_sub, &ndims);
    dim = (int *)malloc(ndims * sizeof(int));
    period = (int *)malloc(ndims * sizeof(int));
    coord = (int *)malloc(ndims * sizeof(int));

    MPI_Cart_get(comm_cart_sub, ndims, &dim[0], &period[0], &coord[0]);
    MPI_Cart_rank(comm_cart_sub, &coord[0], &cart_id);
    printf("Proces s polozajem %d v MPI_COMM_WORLD ", id);
    printf("ima nov polozaj %d na poziciji (", cart_id);
    for (i = 0; i < ndims; i++)
      printf("%d%s", coord[i], (i == ndims-1) ?
          ") v novem podkomunikatorju s topologijo N d prostora\n" : ", ");

    MPI_Comm_free(&comm_cart_sub);
    free(dim);
    free(period);
    free(coord);
  }

  dims[0] = 1;
  dims[1] = 3;
  MPI_Cart_map(MPI_COMM_WORLD, 2, dims, periods, &cart_id);

  if (cart_id == MPI_UNDEFINED)
    printf("Proces s polozajem %d v MPI_COMM_WORLD ni del nove topologije\n", id);
  else {
    printf("Proces s polozajem %d v MPI_COMM_WORLD ", id);
    printf("ima nov polozaj %d v novi topologiji\n", cart_id);
  }

  MPI_Finalize();

  return 0;
}
```

Če poženemo zgornji program na npr. 9 procesih, je eden izmed možnih izpisov:

```
Topologija ima 2 dimenzij - 3x2
Proces s polozajem 0 v MPI_COMM_WORLD ima nov polozaj 0 na poziciji (0, 0) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 0 v MPI_COMM_WORLD ima nov polozaj 0 na poziciji (0) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 0 v MPI_COMM_WORLD ima nov polozaj 0 v novi topologiji
Proces s polozajem 7 ni del novega komunikatorja s topologijo N d prostora
Proces s polozajem 7 v MPI_COMM_WORLD ni del nove topologije
Proces s polozajem 3 v MPI_COMM_WORLD ima nov polozaj 3 na poziciji (1, 1) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 3 v MPI_COMM_WORLD ima nov polozaj 1 na poziciji (1) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 3 v MPI_COMM_WORLD ni del nove topologije
Proces s polozajem 1 v MPI_COMM_WORLD ima nov polozaj 1 na poziciji (0, 1) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 1 v MPI_COMM_WORLD ima nov polozaj 0 na poziciji (0) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 1 v MPI_COMM_WORLD ima nov polozaj 1 v novi topologiji
Proces s polozajem 5 v MPI_COMM_WORLD ima nov polozaj 5 na poziciji (2, 1) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 5 v MPI_COMM_WORLD ima nov polozaj 2 na poziciji (2) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 5 v MPI_COMM_WORLD ni del nove topologije
Proces s polozajem 8 ni del novega komunikatorja s topologijo N d prostora
Proces s polozajem 8 v MPI_COMM_WORLD ni del nove topologije
Proces s polozajem 2 v MPI_COMM_WORLD ima nov polozaj 2 na poziciji (1, 0) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 2 v MPI_COMM_WORLD ima nov polozaj 1 na poziciji (1) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 2 v MPI_COMM_WORLD ima nov polozaj 2 v novi topologiji
Proces s polozajem 4 v MPI_COMM_WORLD ima nov polozaj 4 na poziciji (2, 0) v novem
komunikatorju s topologijo N d prostora
Proces s polozajem 4 v MPI_COMM_WORLD ima nov polozaj 2 na poziciji (2) v novem
podkomunikatorju s topologijo N d prostora
Proces s polozajem 4 v MPI_COMM_WORLD ni del nove topologije
Proces s polozajem 6 ni del novega komunikatorja s topologijo N d prostora
Proces s polozajem 6 v MPI_COMM_WORLD ni del nove topologije
```

2.7 FUNKCIJE ZA DVOTOČKOVNO KOMUNIKACIJO

Funkcije za dvotočkovno komunikacijo se uporabljajo za pošiljanje in sprejemanje sporočil (podatkov) med dvema procesoma znotraj komunikatorja.

Ker se pri večini funkcij za dvotočkovno komunikacijo parametri ponavljajo, so opisani posebej:

- *buf* – začetni naslov podatkov za pošiljanje/sprejemanje,
- *count* – število elementov, ki jih želimo poslati/sprejeti,
- *datatype* – tip elementov, ki jih želimo poslati/sprejeti,
- *dest* – položaj procesa, kateremu je sporočilo namenjeno,
- *source* – položaj procesa, od katerega želimo sprejeti podatke,
- *tag* – značka, ki enolično označuje sporočilo (nenegativno celo število, ki ga sami

določimo). Značka se mora ujemati na pošiljateljevi in sprejemnikovi strani,

- `comm` – komunikator, znotraj katerega poteka dvotočkovna komunikacija,
- `status` – struktura, ki hrani izvor in označbo sporočila.

Nekaj funkcij MPI za dvotočkovno komunikacijo:

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
  Funkcija za pošiljanje podatkov, ki blokira nadaljnje izvajanje procesa, dokler MPI ne odda vseh podatkov.
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
  Funkcija za sprejemanje podatkov, ki blokira nadaljnje izvajanje procesa, dokler MPI ne dostavi sporočila, ki se ujema po parametrih *`source`* in *`tag`*.
- `int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
  Funkcija za pošiljanje podatkov, ki blokira nadaljnje izvajanje procesa, dokler ciljni proces ne začne sprejemati podatkov.
- `int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
  Neblokirajoča funkcija za pošiljanje podatkov. Funkcija preko parametra *`request`* vrne ročico (angl. *handler*), ki jo lahko uporabimo v funkciji `MPI_Wait`.
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`
  Neblokirajoča funkcija za sprejemanje podatkov.
- `int MPI_Wait(MPI_Request *request, MPI_Status *status);`
  Funkcija blokira nadaljnje izvajanje procesa, dokler se funkcija, določena s parametrom *`request`*, ne zaključi.

Primer uporabe nekaterih funkcij za dvotočkovno komunikacijo:

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
  int id, size, i;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (size == 2) {
    if (!id) {
      i = 100;
      MPI_Send(&i, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
```

```
    } else {
      MPI_Status status;
      MPI_Recv(&i, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
      printf("Od procesa %d sem prejel stevilo %d\n", status.MPI_SOURCE, i);
    }
  }

  MPI_Finalize();

  return 0;
}
```

Če poženemo program z 2 procesoma, dobimo naslednji izpis:

```
Od procesa 0 sem prejel stevilo 100
```

2.8 FUNKCIJE ZA KOLEKTIVNO (SKUPINSKO) KOMUNIKACIJO

Funkcije za kolektivno (skupinsko) komunikacijo se uporabljajo za pošiljanje/sprejemanje sporočil (podatkov) med vsemi procesi znotraj komunikatorja, zato ne potrebujejo značk. V standardih MPI-1 in MPI-2 so vse funkcije za skupinsko komunikacijo blokirajoče, zato jih morajo klicati vsi procesi znotraj komunikatorja. Lahko jih razdelimo v tri skupine:

1. sinhronizacija (angl. *synchronization*) – procesi čakajo na sinhronizacijski točki (zapori), dokler je ne dosežejo vsi procesi komunikatorja;
2. premik podatkov (angl. *data movement*) – procesi v komunikatorju si po nekem pravilu izmenjajo podatke;
3. kolektivno računanje (angl. *collective computation*) – en proces od drugih pridobi potrebne podatke in nad njimi opravi neko (aritmetično ali logično) operacijo.

Pri kolektivnem računanju se lahko uporabljajo naslednje operacije: *MPI_MAX* (maksimum), *MPI_MIN* (minimum), *MPI_SUM* (vsota), *MPI_PROD* (produkt), *MPI_LAND* (logični in), *MPI_BAND* (bitni in), *MPI_LOR* (logični ali), *MPI_BOR* (bitni ali), *MPI_LXOR* (logični ekskluzivni ali), *MPI_BXOR* (bitni ekskluzivni ali), *MPI_MAXLOC* (maksimum in lokacija) ter *MPI_MINLOC* (minimum in lokacija). Poleg vnaprej definiranih operacij lahko kreiramo tudi svoje operacije, ki pa morajo biti asociativne.

Za delo z lastnimi operacijami imamo na voljo funkciji:

- `int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);`

  Funkcija na naslovu *`op`* kreira novo operacijo s pomočjo parametra *`function`*, katerega tip `MPI_User_function` je definiran kot:

  ```
  typedef void MPI_User_function(void *invec,
  void *inoutvec, int *len, MPI_Datatype *datatype);
  ```

- `int MPI_Op_free(MPI_Op *op);`
  Sprosti operacijo, ki se nahaja na naslovu *op* in nastavi njeno vrednost na *MPI_OP_NULL*.

Ker se pri večini funkcij za skupinsko komunikacijo parametri ponavljajo, so opisani posebej:

- *sendbuf* – začetni naslov podatkov za pošiljanje,
- *sendcount* – število elementov, ki jih želimo poslati,
- *sendtype* – tip podatkov, ki jih želimo poslati,
- *recvbuf* – začetni naslov, kamor želimo shraniti prejete podatke,
- *recvcount* – število elementov, ki jih želimo prejeti,
- *recvtype* – tip prejetih podatkov,
- *root* – položaj procesa, ki pošilja/sprejema podatke,
- *comm* – komunikator, znotraj katerega poteka kolektivna komunikacija.

Nekaj funkcij MPI za kolektivno komunikacijo:

- `int MPI_Barrier(MPI_Comm comm);`
  Funkcija postavi sinhronizacijsko točko, ki jo morajo doseči vsi procesi znotraj komunikatorja *comm*, preden lahko nadaljujejo izvajanje.
- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
  Funkcija razpošlje kopijo podatkov vsem drugim procesom v komunikatorju *comm*. Pri pošiljanju podatkov sodelujejo vsi procesi, ki so podatke že sprejeli. Če imamo npr. 4 procese, bo v prvi fazi proces *root* (recimo, da je to proces 0) podatke poslal najprej procesu 1, v drugi fazi pa bo proces 0 poslal podatke procesu 2, istočasno pa bo proces 1 poslal podatke še procesu 3.
- `int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount MPI_Datatype recvtype, int root, MPI_Comm comm);`
  Ob klicu funkcije vsi procesi v komunikatorju *comm* pošljejo podatke procesu *root*, ta pa jih pri sebi združi v tabelo.
- `int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);`
  Funkcija je identična funkciji `MPI_Gather`, le da tu vsak proces procesu *root* pošlje različno število elementov.
- `int MPI_Allgather(void *sendbuf, int sendcount,`

```
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm);
```

Funkcija opravlja isto nalogo kot funkcija `MPI_Gather`, le da tu združeno tabelo prejmejo vsi procesi komunikatorja *comm*.

- ```
  int MPI_Allgatherv(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int *recvcount,
  int *displs, MPI_Datatype recvtype, MPI_Comm comm);
  ```
  Funkcija je nadgradnja funkcije `MPI_Allgather`, le da tu različni procesi pošljejo različno število elementov.
- ```
  int MPI_Alltoall(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, MPI_Comm comm);
  ```
  S pomočjo te funkcije si vsi procesi med sabo razpošljejo svojih *sendcount* elementov istega tipa.
- ```
  int MPI_Alltoallw(void *sendbuf, int *sendcounts, int *sdispls,
  MPI_Datatype *sendtypes, void *recvbuf, int *recvcounts,
  int *rdispls, MPI_Datatype *recvtypes, MPI_Comm comm);
  ```
  Funkcija je nadgradnja funkcije `MPI_Alltoall`, le da si tu procesi med sabo razpošljejo različno število svojih elementov, ki so lahko različnih tipov.
- ```
  int MPI_Scatter(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm);
  ```
  Funkcija razdeli tabelo podatkov, ki se nahaja pri procesu *root*, na enake dele in jih razpošlje vsem procesom (tudi procesu *root*).
- ```
  int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
  ```
  Ob klicu funkcije vsak proces v komunikatorju *comm* pošlje podatke procesu *root*, ta pa nad njimi izvede operacijo *op*.
- ```
  int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
  ```
  Funkcija izvede operacijo *op* nad vsemi podatki in nato razpošlje rezultat vsem procesom v komunikatorju *comm*.
- ```
  int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
  int *recvcounts, MPI_Datatype datatype, MPI_Op op,
  MPI_Comm comm);
  ```
  Funkcija je nadgradnja funkcije `MPI_Allreduce`, le da tu za vsak proces določimo, koliko elementov končnega rezultata bo na koncu prejel.
  ```

Primer uporabe nekaterih zgoraj opisanih funkcij:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
#define MAX(X,Y) ((X) < (Y) ? (Y) : (X))

typedef struct {
  int a, b;
} int2;

void minmax(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype) {
  int i, min[2], max[2];
  int2 r;

  for (i = 0; i < *len; i++) {
    min[0] = MIN(((int2 *)invec)->a, ((int2 *)inoutvec)->a);
    min[1] = MIN(((int2 *)invec)->b, ((int2 *)inoutvec)->b);
    max[0] = MAX(((int2 *)invec)->a, ((int2 *)inoutvec)->a);
    max[1] = MAX(((int2 *)invec)->b, ((int2 *)inoutvec)->b);

    r.a = MIN(min[0], min[1]);
    r.b = MAX(max[0], max[1]);

    *((int2 *)inoutvec) = r;

    invec++;
    inoutvec++;
  }
}

int main(int argc, char *argv[]) {
  int2 in, out;
  MPI_Datatype int2_type;
  MPI_Op minmax_op;
  time_t t;
  int id;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  srand(((unsigned)time(&t)) + id*1000);

  in.a = (rand() % 1000) * (rand() & 1 ? 1 : -1);
  in.b = (rand() % 1000) * (rand() & 1 ? 1 : -1);

  MPI_Type_contiguous(2, MPI_INT, &int2_type);
  MPI_Type_commit(&int2_type);
  MPI_Op_create(minmax, 1, &minmax_op);
  MPI_Reduce(&in.a, &out.a, 1, int2_type, minmax_op, 0, MPI_COMM_WORLD);

  printf("vhod %d)%+4d, %+4d\n", id, in.a, in.b);
  if (!id)
    printf("min: %+4d; max: %+4d\n", out.a, out.b);

  MPI_Op_free(&minmax_op);

  MPI_Finalize();

  return 0;
}
```

Če poženemo zgornji program na npr. 5 procesih je eden izmed možnih izpisov:

```
vhod 4)+602, +837
vhod 2)+839, +382
vhod 1)-573, +771
vhod 3)+336, +226
vhod 0)-308, -159
min: -573; max: +839
```

2.9 PODATKOVNI TIPI

Podatkovni tip je formalni opis, ki nam pove, kako so podatki predstavljeni. MPI pozna dve vrsti podatkovnih tipov: osnovne (primitivne) podatkovne tipe (angl. *elementary data types*) in izpeljane (sestavljene) podatkovne tipe (angl. *derived data types*). Razlog, da MPI uporablja svoje podatkovne tipe, se skriva v dejstvu, da lahko skupino računalnikov, na katerih izvajamo procese MPI, sestavlja več heterogenih računalnikov (tako lahko npr. na računalniku A spremenljivka tipa int zaseda 8 bajtov v pomnilniku in je shranjena po pravilu tankega konca (angl. *little endian*), medtem ko na računalniku B spremenljivka tipa int zaseda 4 bajte v pomnilniku in je shranjena po pravilu debelega konca (angl. *big endian*)). Podatki osnovnih tipov se vedno držijo skupaj (angl. *contiguous*).

Naštetih je nekaj primerov osnovnih podatkovnih tipov:

- *MPI_CHAR*
- *MPI_UNSIGNED_SHORT*
- *MPI_INT*
- *MPI_UNSIGNED_INT*
- *MPI_LONG*
- *MPI_LONG_LONG_INT*
- *MPI_FLOAT*
- *MPI_DOUBLE*

Izpeljane podatkovne tipe sestavimo iz osnovnih podatkovnih tipov in/ali iz že izpeljanih podatkovnih tipov. Izpeljane podatkovne tipe lahko sestavimo na 4 načine, in sicer kot:

- tabelo (angl. *contiguous*), kjer med posameznimi elementi istega podatkovnega tipa ni razmika (vrzeli),
- vektor (angl. *vector*), kjer je (lahko) med posameznimi elementi istega podatkovnega tipa vedno enako velik razmik,
- indeksiran razmik (angl. *indexed*), kjer so (lahko) med posameznimi elementi istega podatkovnega tipa poljubno veliki razmiki, in

- strukturo (angl. *struct*), kjer so (lahko) med posameznimi elementi različnega podatkovnega tipa poljubno veliki razmiki.

Vsak podatkovni tip ima spodnjo in zgornjo mejo. Spodnja meja nam poda odmik od začetka prvega podatka podatkovnega tipa, informacija o zgornji meji pa je odmik od konca zadnjega podatka v podatkovnem tipu. S pomočjo spodnje in zgornje meje lahko izračunamo obseg (angl. *extent*) podatkovnega tipa, kot razliko med zgornjo mejo in spodnjo mejo.

Nekatere funkcije za delo s podatkovnimi tipi:

- `int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip, ki je enak podatkovnemu tipu *`oldtype`*, le da ima spremenjeno spodnjo in zgornjo mejo.
- `int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent);`
  Funkcija na naslov *`lb`* zapiše spodnjo mejo in na naslov *`extent`* obseg podatkovnega tipa *`datatype`*.
- `int MPI_Type_set_name(MPI_Datatype type, char *type_name);`
  Funkcija nastavi novo ime podatkovnega tipa *`type`*.
- `int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen);`
  Funkcija na naslov *`type_name`* zapiše ime podatkovnega tipa *`type`* in na naslov *`resultlen`* dolžino njegovega imena (samo število znakov, brez znaka '\0').
- `int MPI_Type_size(MPI_Datatype datatype, int *size);`
  Funkcija na naslov *`size`* zapiše velikost (v bajtih) enega elementa podanega podatkovnega tipa *`datatype`*.
- `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip oblike tabela, ki vsebuje *`count`* elementov podatkovnega tipa *`oldtype`*.
- `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip, ki vsebuje *`count`* blokov. Vsak blok sestoji iz *`blocklength`* elementov podatkovnega tipa *`oldtype`*. Parameter *`stride`* poda razmik (v številu elementov podatkovnega tipa *`oldtype`*) med začetkom vsakega bloka.

- `int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija je identična funkciji `MPI_Type_vector`, le da je tu razmik *`stride`* podan v številu bajtov,
- `int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip, ki vsebuje *`count`* blokov. Vsi bloki so veliki *`blocklength`* elementov podatkovnega tipa *`oldtype`*. Tabela *`array_of_displacements`* mora imeti *`count`* elementov in vsebuje odmike (v številu elementov *`oldtype`*) za vsak blok posebej.
- `int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip, ki vsebuje *`count`* blokov. Velikosti posameznih blokov so navedene kot elementi tabele *`array_of_blocklengths`*, ki mora vsebovati *`count`* elementov. Blok i je velik *`array_of_blocklengths[i]`* elementov podatkovnega tipa *`oldtype`*. Tabela *`array_of_displacements`* mora vsebovati *`count`* elementov, ki predstavljajo odmike (v številu elementov *`oldtype`*) za vsak blok posebej.
- `int MPI_Type_create_hindexed(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype);`
  Funkcija je identična funkciji `MPI_Type_indexed`, le da so tu razmiki v tabeli *`array_of_displacements`* podani v številu bajtov, in ne v številu elementov podatkovnega tipa *`oldtype`*.
- `int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype);`
  Funkcija na naslovu *`newtype`* kreira nov podatkovni tip s *`count`* bloki. Vsaka od treh tabel mora vsebovati *`count`* elementov. Tip elementa v posameznem bloku je naveden v tabeli *`array_of_types`*. Velikost bloka i (v številu elementov *`array_of_types[i]`*) je *`array_of_blocklengths[i]`*. Odmik posameznega bloka (v bajtih) je podan v tabeli *`array_of_displacements`*.
- `int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype);`
  Funkcija naredi kopijo podatkovnega tipa *`type`*.

- `int MPI_Type_commit(MPI_Datatype *datatype);`

  Funkcija preda v uporabo podatkovni tip, ki se nahaja na naslovu *`datatype`*. Podatkovni tip mora biti predan v uporabo s klicem te funkcije, preden ga začnemo uporabljati v komunikaciji med procesi.
- `int MPI_Type_free(MPI_Datatype *datatype);`

  Funkcija sprosti podatkovni tip, ki se nahaja na naslovu *`datatype`* in nastavi njegovo vrednost na *MPI_DATATYPE_NULL*. Sprostitev določenega podatkovnega tipa nima vpliva na druge podatkovne tipe, tudi če so bili izpeljani iz pravkar sproščenega podatkovnega tipa.

Primer programa, ki uporablja funkcije za delo s podatkovnimi tipi:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define DERIVED_DATATYPES_NUM 5

int main(int argc, char *argv[]) {
  int id, comm_size, size, i, j;
  int *block_len, *displacements;
  char *tab;
  char *derived_datatype_name[DERIVED_DATATYPES_NUM] = { "tip_tabela",
      "tip_vektor", "tip_indeksiran_razmik", "tip_indeksiran_razmik2",
      "tip_struktura" };
  MPI_Aint lower_bound, extent;
  MPI_Datatype *array_of_types;
  MPI_Datatype type_array, type_vector, type_indexed, type_indexed2, type_struct;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

  if (comm_size < 2)
    MPI_Abort(MPI_COMM_WORLD, 1);

  tab = calloc(10*comm_size, sizeof(char));
  block_len = malloc(comm_size * sizeof(int));
  displacements = malloc(comm_size * sizeof(int));
  array_of_types = malloc(comm_size * sizeof(MPI_Datatype));

  if (!id)
    for (i = 0; i < 10*comm_size; i++)
      tab[i] = i + 1;

  for (i = 0; i < comm_size; i++) {
    block_len[i] = i + 1;
    displacements[i] = i*comm_size;
    array_of_types[i] = !i ? MPI_INT : MPI_CHAR;
  }

  MPI_Type_contiguous(10, MPI_CHAR, &type_array);
  MPI_Type_vector(10, 1, comm_size, MPI_CHAR, &type_vector);
  MPI_Type_create_indexed_block(comm_size, 2, &displacements[0], MPI_CHAR,
      &type_indexed);
  MPI_Type_indexed(comm_size, &block_len[0], &displacements[0], MPI_CHAR,
      &type_indexed2);
```

```
  MPI_Type_create_struct(comm_size, &block_len[0], (MPI_Aint*)(&displacements[0]),
      &array_of_types[0], &type_struct);

  MPI_Datatype derived_datatype[DERIVED_DATATYPES_NUM] = { type_array, type_vector,
      type_indexed, type_indexed2, type_struct };
  for (i = 0; i < DERIVED_DATATYPES_NUM; i++) {
    MPI_Type_commit(&derived_datatype[i]);
    MPI_Type_set_name(derived_datatype[i], derived_datatype_name[i]);
    MPI_Type_get_extent(derived_datatype[i], &lower_bound, &extent);
    MPI_Type_size(derived_datatype[i], &size);
    char datatype_name[100] = { 0 };
    int datatype_name_len;
    MPI_Type_get_name(derived_datatype[i], &datatype_name[0], &datatype_name_len);
    if (id == 1)
      printf("Velikost %s v bajtih:%d; dejansko stevilo zasedenih bajtov:%d\n",
          datatype_name, extent, size);
  }

  for (i = 0; i < DERIVED_DATATYPES_NUM; i++) {
    MPI_Bcast(&tab[0], 1, derived_datatype[i], 0, MPI_COMM_WORLD);
    if (id == 1) {
      printf("%-22s:", derived_datatype_name[i]);
      for (j = 0; j < 10*comm_size; j++) {
        printf("%2d%s", tab[j], (j != 10*comm_size - 1) ? ", " : "\n");
        tab[j] = 0;
      }
    }
    MPI_Type_free(&derived_datatype[i]);
  }

  MPI_Finalize();

  free(tab);
  free(block_len);
  free(displacements);
  free(array_of_types);

  return 0;
}
```

Izpis programa, če ga poženemo na 3 procesih:

```
Velikost tip_tabela v bajtih:10; dejansko stevilo zasedenih bajtov:10
Velikost tip_vektor v bajtih:28; dejansko stevilo zasedenih bajtov:10
Velikost tip_indeksiran_razmik v bajtih:8; dejansko stevilo zasedenih bajtov:6
Velikost tip_indeksiran_razmik2 v bajtih:9; dejansko stevilo zasedenih bajtov:6
Velikost tip_struktura v bajtih:12; dejansko stevilo zasedenih bajtov:9
tip_tabela           : 1,  2,  3,  4,  5,  6,  7,  8,  9, 10,  0,  0,  0,  0,  0,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
tip_vektor           : 1,  0,  0,  4,  0,  0,  7,  0,  0, 10,  0,  0, 13,  0,  0,
16,  0,  0, 19,  0,  0, 22,  0,  0, 25,  0,  0, 28,  0,  0
tip_indeksiran_razmik : 1,  2,  0,  4,  5,  0,  7,  8,  0,  0,  0,  0,  0,  0,  0,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
tip_indeksiran_razmik2: 1,  0,  0,  4,  5,  0,  7,  8,  9,  0,  0,  0,  0,  0,  0,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
tip_struktura        : 1,  2,  3,  4,  5,  0,  7,  8,  9,  0,  0,  0,  0,  0,  0,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
```

2.10 OBJEKT INFO

MPI_Info je objekt, ki vsebuje poljubno število neurejenih dvojic (ključ (angl. *key*), vrednost

(angl. *value*)). Tako ključ kot vrednost sta tipa niz znakov. Objekt tipa MPI_Info se uporablja v mnogih funkcijah MPI kot parameter, s katerim sporočamo določene informacije, ki niso ključne za pravilno izvedbo funkcije. Preko objekta tipa MPI_Info lahko od funkcije MPI dobimo podrobne informacije o njeni izvedbi. Če objekta tipa MPI_Info ne želimo uporabljati, funkcijam MPI, ki ga zahtevajo, podamo vrednost *MPI_INFO_NULL*. Funkcije MPI za delo z objekti tipa MPI_Info se začnejo z `MPI_Info_`.

Funkcije za delo z objekti tipa MPI_Info:

- `int MPI_Info_create(MPI_Info *info);`
  Funkcija kreira nov objekt tipa MPI_Info.
- `int MPI_Info_free(MPI_Info *info);`
  Funkcija sprosti objekt tipa MPI_Info, ki se nahaja na naslovu *`info`* in mu priredi vrednost *MPI_INFO_NULL*.
- `int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo);`
  Funkcija naredi kopijo objekta *`info`*.
- `int MPI_Info_set(MPI_Info info, char *key, char *value);`
  Funkcija objektu *`info`* doda novo dvojico (ključ, vrednost). Maksimalna dolžina niza *`key`* je *MPI_MAX_INFO_KEY*, maksimalna dolžina niza *`value`* pa *MPI_MAX_INFO_VAL*.
- `int MPI_Info_get(MPI_Info info, char *key, int valuelen,`
  `char *value, int *flag);`
  S pomočjo te funkcije dobimo vrednost, dodeljeno ključu *`key`*. Logična vrednost na naslovu *`flag`* nam pove, ali ključ sploh obstaja. Če ključ obstaja, funkcija na naslov *`value`* zapiše *`valuelen`* znakov.
- `int MPI_Info_get_valuelen(MPI_Info info, char *key,`
  `int *valuelen, int *flag);`
  Če je na naslovu *`flag`* logična vrednost *pravilno*, funkcija na naslov *`valuelen`* zapiše dolžino (brez znaka '\0') podatka, dodeljenega ključu *`key`*.
- `int MPI_Info_get_nkeys(MPI_Info info, int *nkeys);`
  Funkcija na naslov *`nkeys`* zapiše število ključev v objektu *`info`*.
- `int MPI_Info_get_nthkey(MPI_Info info, int n, char *key);`
  Funkcija na naslov *`key`* zapiše *`n`*-ti ključ objekta *`info`*.

Program, ki uporablja funkcije za delo z objektom MPI_Info:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[]) {
  int id, num, len, flag;
  char buff[10];
  MPI_Info info, info_dup;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  MPI_Info_create(&info);
  MPI_Info_set(info, "kljuc1", "kivi");
  MPI_Info_set(info, "kljuc1", "mango");
  MPI_Info_set(info, "kljuc2", "banana");
  MPI_Info_set(info, "kljuc3", "pomaranca");
  MPI_Info_dup(info, &info_dup);
  MPI_Info_get_nkeys(info, &num);
  if (!id)
    printf("Objekt 'info' ima %d parov (kljuc, vrednost)\n", num);
  MPI_Info_get_valuelen(info, "kljuc1", &len, &flag);
  if (flag) {
    MPI_Info_get(info, "kljuc1", len+1, &buff[0], &flag);
    if (!id)
      printf("Kljuc 'kljuc1' ima vrednost '%s' (velikost: %d)\n", buff, len);
  }
  MPI_Info_get_nthkey(info, 1, &buff[0]);
  if (!id)
    printf("2. kljuc je '%s'\n", buff);
  MPI_Info_get_valuelen(info, "kljuc3", &len, &flag);
  if (flag)
    MPI_Info_delete(info, "kljuc3");
  MPI_Info_get_nkeys(info, &num);
  if (!id)
    printf("Objekt 'info' ima %d parov (kljuc, vrednost)\n", num);
  MPI_Info_get(info, "kljuc7", 10, &buff[0], &flag);
  if (flag) {
    if (!id)
      printf("Kljuc 'kljuc7' ima vrednost '%s'\n", buff);
  } else {
    if (!id)
      printf("Kljuc 'kljuc7' ni definiran v objektu 'info'\n");
  }
  MPI_Info_get_nkeys(info_dup, &num);
  if (!id)
    printf("Objekt 'info_dup' ima %d parov (kljuc, vrednost)\n", num);
  MPI_Info_free(&info);
  MPI_Info_free(&info_dup);

  MPI_Finalize();

  return 0;
}
```

Če program poženemo, dobimo izpis:

```
Objekt 'info' ima 3 parov (kljuc, vrednost)
Kljuc 'kljuc1' ima vrednost 'mango' (velikost: 5)
2. kljuc je 'kljuc2'
Objekt 'info' ima 2 parov (kljuc, vrednost)
Kljuc 'kljuc7' ni definiran v objektu 'info'
Objekt 'info_dup' ima 3 parov (kljuc, vrednost)
```

2.11 VHOD/IZHOD

MPI omogoča delo z vhodom in izhodom. Ker so v sistemih UNIX/Linux vhodno-izhodne naprave s strani operacijskega sistema vidne kot datoteke, v nadaljevanju namesto izraza vhod/izhod, uporabljam izraz datoteka. Vse funkcije MPI za delo z datotekami se začnejo z MPI_File. Pred kakršno koli operacijo nad datoteko moramo datoteko najprej povezati z datotečno ročico (angl. *file handler*). Vsem funkcijam za delo z datotekami moramo kot argument podati bodisi datotečno ročico bodisi ime datoteke.

Nekatere funkcije za delo z datotekami:

- `int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);`
  Funkcija odpre datoteko z imenom *`filename`* in ji priredi datotečno ročico *`fh`*. Datoteko lahko odpremo v več načinih:
  1. *MPI_MODE_RDONLY* – odpre datoteko samo za branje;
  2. *MPI_MODE_WRONLY* – odpre datoteko samo za pisanje;
  3. *MPI_MODE_RDWR* – odpre datoteko za branje in pisanje;
  4. *MPI_MODE_APPEND* – ob odprtju postavi datotečni kazalec na konec datoteke;
  5. *MPI_MODE_CREATE* – če datoteka ne obstaja, jo naredi;
  6. *MPI_MODE_EXCL* – če datoteka obstaja, funkcija vrne napako;
  7. *MPI_MODE_SEQUENTIAL* – dostop do datoteke bo zaporeden, skakanje po datoteki je prepovedano. Ta način je primeren za npr. tračne enote ali tokove podatkov, saj je optimiziran za zaporedni dostop;
  8. *MPI_MODE_DELETE_ON_CLOSE* – ob zaprtju datoteke jo tudi izbriše;
  9. *MPI_MODE_UNIQUE_OPEN* – poskrbi, da datoteka ni dostopna od drugod, dokler je MPI ne zapre. S tem načinom zmanjšamo odvečno komunikacijo, potrebno za zaklepanje odprte datoteke. Ta način lahko uporabimo, le če smo prepričani, da v tem času do datoteke ne bo dostopal noben drug proces.

  Načine lahko kombiniramo s pomočjo bitnega operatorja ali (|). Kombinacija nekaterih načinov vrne napako (npr. kombinacija *MPI_MODE_CREATE* in *MPI_MODE_RDONLY*, kombinacija *MPI_MODE_SEQUENTIAL* in *MPI_MODE_RDWR*, …). Pri odpiranju datoteke moramo obvezno navesti natančno enega od načinov: *MPI_MODE_RDONLY*, *MPI_MODE_WRONLY* in *MPI_MODE_RDWR*. Objekt *`info`* lahko uporabimo na 3 načine:
  1. podajamo določene informacije (namige, angl. *hints*) o datoteki (npr. *shared_file_timeout* – podamo čas, ki ga imajo vsi procesi na voljo za dostop

do datotečne ročice);
2. beremo določene informacije o datoteki (npr. *filename* – dobimo ime datoteke);
3. ne uporabljamo objekta `info` – na njegovo mesto vstavimo *MPI_INFO_NULL*.

- `int MPI_File_close(MPI_File *fh);`
  Funkcija zapre datoteko, pripojeno datotečni ročici `fh`.
- `int MPI_File_delete(char *filename, MPI_Info info);`
  Funkcija zbriše datoteko z imenom `filename`. Datoteke ne sme imeti odprte noben drug proces.
- `int MPI_File_set_atomicity(MPI_File fh, int flag);`
  Funkcija zagotovi, da se bodo nad datoteko izvajale le atomske operacije[2].
- `int MPI_File_set_info(MPI_File fh, MPI_Info info);`
  Funkcija datoteki pripne objekt `info`,
- `int MPI_File_set_size(MPI_File fh, MPI_Offset size);`
  Funkcija nastavi novo velikost datoteke. Datoteke ne moremo zmanjšati, dejanska nova velikost je maksimum trenutne velikosti datoteke in vrednosti parametra `size`.
- `int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info);`
  Funkcija spremeni način, kako posamezen proces vidi razporeditev podatkov v datoteki. Z odmikom `disp` se določi odmik od začetka datoteke. Parameter `etype` predstavlja podatkovni tip podatkov v datoteki, `filetype` pa nam pove, kako posamezen proces vidi podatke (določen vzorec uporabnih bajtov in vrzeli). Parameter `datarep` ima lahko naslednje vrednosti:
  1. `native` – podatki v datoteki so shranjeni enako kot v spominu. Hiter način, vendar ni prenosljiv. Po navadi se uporablja, če program poganjamo na istem računalniku (oz. na drugem računalniku z enako arhitekturo), ali pa za začasne datoteke;
  2. `internal` – prenosljiv format, ki nam zagotavlja, da zapisano datoteko lahko identično preberemo na drugem računalniku z drugačno arhitekturo, če le uporabljamo isto implementacijo MPI;
  3. `external32` – prenosljiv format, ne le med različnimi arhitekturami, ampak tudi med različnimi implementacijami MPI. Pisanje/branje v tem primeru poteka le v 32-bitnem načinu, kar ima lahko za posledico manjšo hitrost dostopa do datotek.

2 Operacije, ki se bodo vedno neprekinjeno izvedle od začetka do konca.

- `int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);`
  Funkcija ponastavi individualni datotečni kazalec na odmik *`offset`*. Parameter *`whence`* določa začetno točko odmika in ima lahko vrednost:
  1. *MPI_SEEK_SET* – začetek datoteke; nova vrednost datotečnega kazalca = 0 + odmik;
  2. *MPI_SEEK_CUR* – trenutni položaj datotečnega kazalca; nova vrednost datotečnega kazalca = trenutni položaj datotečnega kazalca + odmik;
  3. *MPI_SEEK_END* – konec datoteke; nova vrednost datotečnega kazalca = konec datoteke + odmik.
- `int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence);`
  Funkcija premakne skupen datotečni kazalec. Parametri so isti kot pri funkciji `MPI_File_seek`.
- `int MPI_File_get_atomicity(MPI_File fh, int *flag);`
  Funkcija preveri, ali je izbran atomični način dela z datoteko.
- `int MPI_File_get_info(MPI_File fh, MPI_Info *info_used);`
  Funkcija na naslov *`info_used`* vrne objekt tipa MPI_Info.
- `int MPI_File_get_amode(MPI_File fh, int *amode);`
  Funkcija na naslov *`amode`* zapiše način, v katerem je bila datoteka odprta.
- `int MPI_File_get_size(MPI_File fh, MPI_Offset *size);`
  Funkcija na naslov *`size`* zapiše trenutno velikost datoteke.
- `int MPI_File_get_position(MPI_File fh, MPI_Offset *offset);`
  Funkcija na naslov *`offset`* zapiše trenutni odmik individualnega datotečnega kazalca.
- `int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset);`
  Funkcija na naslov *`offset`* zapiše trenutni odmik skupnega datotečnega kazalca.

V Tabeli 1 so zbrane funkcije za branje iz datoteke. Če v imenih funkcij za branje podatkov iz datoteke besedo *read* zamenjamo z besedo *write*, dobimo funkcije za pisanje v datoteko. Obnašajo se popolnoma enako, le da namesto branja iz datoteke sprožijo operacijo pisanja v datoteko. Tudi parametri so isti in se enako uporabljajo.

Način premika po datoteki	Kolektivnost funkcije	Način komunikacije	Ime funkcije
Funkcija uporablja skupni datotečni kazalec	Nekolektivna funkcija	Neblokirajoča funkcija	`MPI_File_iread_shared`
		Blokirajoča funkcija	`MPI_File_read_shared`
	Kolektivna funkcija	Neblokirajoča funkcija	`MPI_File_read_ordered_begin`[3]
		Blokirajoča funkcija	`MPI_File_read_ordered`
Funkcija uporablja individualni datotečni kazalec	Nekolektivna funkcija	Neblokirajoča funkcija	`MPI_File_iread`
		Blokirajoča funkcija	`MPI_File_read`
	Kolektivna funkcija	Neblokirajoča funkcija	`MPI_File_read_all_begin`[4]
		Blokirajoča funkcija	`MPI_File_read_all`
Funkcija uporablja odmik	Nekolektivna funkcija	Neblokirajoča funkcija	`MPI_File_iread_at`
		Blokirajoča funkcija	`MPI_File_read_at`
	Kolektivna funkcija	Neblokirajoča funkcija	`MPI_File_read_at_all_begin`[5]
		Blokirajoča funkcija	`MPI_File_read_at_all`

Tabela 1: Pregled funkcij MPI za branje iz datoteke

Primer uporabe nekaterih funkcij za delo z datotekami:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  int id, i, size;
  char str_w[7] = { 0 };
  char str_r[7] = { 0 };
  time_t t;
  MPI_File fh;
  MPI_Status status;
  MPI_Datatype type_vector;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(((unsigned)time(&t)) + id*1000);

  MPI_File_open(MPI_COMM_WORLD, "test.txt", MPI_MODE_CREATE | MPI_MODE_WRONLY,
      MPI_INFO_NULL, &fh);

  MPI_File_seek(fh, (MPI_Offset)id, MPI_SEEK_SET);
  for (i = 0; i < 6; i++) {
    str_w[i] = (i < 3 ? 'A' : 'a') + rand() % 26;
    MPI_File_write_all(fh, (void *)(&str_w[i]), 1, MPI_CHAR, &status);
```

3 V kombinaciji s kolektivno blokirajočo funkcijo MPI_File_read_ordered_end.
4 V kombinaciji s kolektivno blokirajočo funkcijo MPI_File_read_all_end.
5 V kombinaciji s kolektivno blokirajočo funkcijo MPI_File_read_at_all_end.

```
    MPI_File_seek(fh, (MPI_Offset)(size-1), MPI_SEEK_CUR);
  }
  MPI_File_close(&fh);

  MPI_Type_vector(6, 1, size, MPI_CHAR, &type_vector);
  MPI_Type_commit(&type_vector);

  MPI_File_open(MPI_COMM_WORLD, "test.txt", MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, (MPI_Offset)(size-(id+1)), MPI_CHAR, type_vector,
      "native", MPI_INFO_NULL);

  MPI_File_read(fh, (void *)(&str_r[0]), 6, MPI_CHAR, &status);

  printf("%d) W:'%s' R:'%s'\n", id, str_w, str_r);

  MPI_Type_free(&type_vector);
  MPI_File_close(&fh);

  MPI_Finalize();

  return 0;
}
```

Če program poženemo na npr. 5 procesih, je eden izmed možnih izpisov:

```
1) W:'ZHSwer' R:'EYJzdz'
3) W:'EYJzdz' R:'ZHSwer'
0) W:'RZEqsn' R:'UGEfpc'
4) W:'UGEfpc' R:'RZEqsn'
2) W:'PPNbrv' R:'PPNbrv'
```

2.12 ENOSTRANSKA KOMUNIKACIJA

MPI omogoča oddaljen dostop do pomnilnika (angl. *Remote Memory Access*, RMA). To pomeni, da lahko en proces dostopa do pomnilnika drugih procesov, tako da sam poda vse potrebne parametre komunikacije – izvor (angl. *origin*), cilj (angl. *target*), količino in tip prenesenih podatkov, drugi procesi pa mu le omogočijo dostop do določenega dela svojega pomnilnika. Del pomnilnika, ki ga proces določi, da ga lahko drugi procesi poljubno berejo in spreminjajo, sam pa pri tem ne sodeluje, imenujemo okno (angl. *window*).

MPI pozna tri funkcije za komunikacijski del enostranske komunikacije, in sicer `MPI_Get`, `MPI_Put` ter `MPI_Accumulate`. Funkcije, ki se začnejo z `MPI_Win`, so sinhronizacijske funkcije enostranske komunikacije.

Komunikacijske funkcije za delo z enostransko komunikacijo imajo naslednje skupne parametre:

- *`origin_addr`* – začetni naslov na izvornem procesu, na/s katerega želimo prejeti/poslati podatke,
- *`origin_count`* – število elementov na izvornem procesu, ki jih želimo

prejeti/poslati,

- *origin_datatype* – podatkovni tip elementov na izvornem procesu, ki jih želimo prejeti/poslati,
- *targent_rank* – položaj ciljnega procesa od katerega/kateremu želimo prejeti/poslati podatke,
- *target_disp* – odmik od začetnega naslova, ki ga je ciljni proces določil ob kreiranju svojega okna povečan za odmik *target_disp*,
- *target_count* – število elementov na ciljnem procesu, ki jih želimo prejeti/poslati,
- *target_datatype* – podatkovni tip elementov na ciljnem procesu, ki jih želimo prejeti/poslati,
- *win* – objekt tipa MPI_Win preko katerega poteka enostranska komunikacija.

Komunikacijske funkcije za delo z enostransko komunikacijo:

- ```
  int MPI_Get(void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win);
  ```
  Funkcija prepiše podatke iz okna ciljnega procesa v pomnilnik izvornega procesa. Podatki v oknu ciljnega procesa ostanejo nespremenjeni.
- ```
  int MPI_Put(void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Win win);
  ```
  Funkcija prepiše podatke v okno ciljnega procesa iz pomnilnika izvornega procesa. Podatki izvornega procesa ostanejo nespremenjeni.
- ```
  int MPI_Accumulate(void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank,
  MPI_Aint target_disp, int target_count,
  MPI_Datatype target_datatype, MPI_Op op, MPI_Win win);
  ```
  Funkcija nad podatki v oknu ciljnega procesa in nad podatki izvornega procesa izvede operacijo *op*, katere rezultat se shrani v okno ciljnega procesa. Podatki izvornega procesa ostanejo nespremenjeni.

Pred prvim in po zadnjem dostopu do okna moramo izvesti sinhronizacijo. To lahko izvedemo na tri načine (z uporabo treh skupin sinhronizacijskih funkcij MPI):

1. funkciji `MPI_Win_lock` in `MPI_Win_unlock`,
2. funkcija `MPI_Fence`,
  ```

3. funkcije `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post` in `MPI_Win_wait`.

Skupino dostopov do okna med dvema sinhronizacijama imenujemo doba (angl. *epoch*).

Oddaljeni dostop do pomnilnika je s stališča ciljnega okna lahko aktiven ali pasiven. Aktiven pomeni, da ciljni proces deloma sam izvaja dostop (piše/bere) do okna, deloma pa dostop do okna prepusti izvornemu procesu. Pasiven pa pomeni, da do okna ciljnega procesa dostopata dva izvorna procesa (pri čemer sinhronizacija poteka z zaklepanjem in sproščanjem ciljnega okna), ciljni proces pa sploh ne dostopa do svojega okna.

Nekatere sinhronizacijske funkcije za delo z enostransko komunikacijo:

- `int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win);`
  Funkcija določi okno v pomnilniku, ki se začne na naslovu *`base`*, z velikostjo *`size`* bajtov, pri čemer lahko vsak proces določi svoj začetni naslov in velikost, ki je lahko tudi 0. Parameter *`disp_init`* določi, v kakšnih enotah (koliko bajtov) se bo štel odmik v komunikacijskih funkcijah.
- `int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win);`
  Funkcija drugim procesom zaklene dostop do okna procesa na poziciji *`rank`*. Vrednost parametra *`lock_type`* je lahko *MPI_LOCK_EXCLUSIVE* ali *MPI_LOCK_SHARED*.
- `int MPI_Win_unlock(int rank, MPI_Win win);`
  Funkcija sprosti dostop do okna procesa na poziciji *`rank`*.
- `int MPI_Win_fence(int assert, MPI_Win win);`
  Zaključi morebitno predhodno dobo (če je bila predhodno klicana ista funkcija) in začne novo dobo (če je za tem klicem še kakšen klic istoimenske funkcije). Parameter *`assert`* določa dodatne sinhronizacijske opcije (*MPI_MODE_NOSTORE*, *MPI_MODE_NOPUT*, *MPI_MODE_NOPRECEDE* in *MPI_MODE_NOSUCCEED*), ki se lahko uporabijo za dodatno optimizacijo, lahko pa se uporabi konstanta 0.
- `int MPI_Win_start(MPI_Group group, int assert, MPI_Win win);`
  Funkcija začne novo dobo. Vsi pripadniki skupine *`group`* morajo poklicati sinhronizacijsko funkcijo `MPI_Win_Post` z istim parametrom *`assert`*. Parameter *`assert`* ima lahko vrednost 0 ali *MPI_MODE_NOCHECK* (v tem primeru se ne preveri, ali so pripadniki skupine group poklicali `MPI_Win_post`).
- `int MPI_Win_post(MPI_Group group, int assert, MPI_Win win);`
  Funkcija začne novo dobo. Skupina, podana s parametrom *`group`*, mora vsebovati vse

procese, ki so klicali funkcijo `MPI_Win_start`.

- `int MPI_Win_complete(MPI_Win win);`
  Funkcija zaključi dobo, ki se je začela s klicem funkcije `MPI_Win_start`,
- `int MPI_Win_wait(MPI_Win win);`
  Funkcija zaključi dobo, ki se je začela s klicem funkcije `MPI_Win_post`.
- `int MPI_Win_test(MPI_Win win, int *flag);`
  Funkcija je neblokirajoča različica funkcije `MPI_Win_wait`, ki na naslov *`flag`* zapiše logično vrednost *napačno*, če se doba (še) ni zaključila, in logično vrednost *pravilno*, če se je. Vstavimo jo lahko le na mesto, kjer bi drugače morala biti funkcija `MPI_Win_wait`.
- `int MPI_Win_set_name(MPI_Win win, char *win_name);`
  Funkcija nastavi ime objektu *`win`*.
- `int MPI_Win_get_name(MPI_Win win, char *win_name,`
  `int *resultlen);`
  Funkcija na naslov *`win_name`* zapiše ime, na naslov *`resultlen`* pa dolžino imena objekta *`win`*.
- `int MPI_Win_get_attr(MPI_Win win, int win_keyval,`
  `void *attribute_val, int *flag);`
  Če ključ *`win_keyval`* obstaja, funkcija na naslov *`flag`* zapiše logično vrednost *pravilno* in na naslov *`attribute_val`* vrednost atributa.

Primer uporabe nekaterih funkcij za enostransko komunikacijo:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define N 20

int main(int argc, char *argv[]) {
  int id, i, size, buff_size, flag, new_id;
  void *val, *buff = NULL;
  char win_name[MPI_MAX_OBJECT_NAME];
  MPI_Win win;
  MPI_Group comm_group, new_group;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  buff_size = id ? N : 2*N;

  MPI_Alloc_mem((MPI_Aint)(buff_size * sizeof(char)), MPI_INFO_NULL, &buff);

  MPI_Win_create(buff, (MPI_Aint)buff_size, sizeof(int), MPI_INFO_NULL,
      MPI_COMM_WORLD, &win);
  MPI_Win_set_name(win, "test");
  MPI_Win_get_name(win, &win_name[0], &i);
```

```
  MPI_Win_get_attr(win, MPI_WIN_SIZE, &val, &flag);
  if (flag)
    printf("%d)%s atribut 'MPI_WIN_SIZE': %d\n", id, win_name, *((int *)val));

  for (i = 0; i < buff_size; i++)
    *((char *)buff + i) = id + i;

  switch (id) {
    case 0:
      new_id = 1;
    break;
    case 1:
      new_id = 0;
    break;
    default:
      new_id = id;
  }

  MPI_Comm_group(MPI_COMM_WORLD, &comm_group);
  MPI_Group_incl(comm_group, 1, &new_id, &new_group);

  if (!id) {
    MPI_Win_start(new_group, 0, win);
    MPI_Put(buff, 2, MPI_INT, 1, (MPI_Aint)0, 2, MPI_INT, win);
    MPI_Get(buff, 2, MPI_INT, 1, (MPI_Aint)3, 2, MPI_INT, win);
    MPI_Win_complete(win);
  } else if (id == 1) {
    MPI_Win_post(new_group, 0, win);
    MPI_Win_wait(win);
  }
  MPI_Group_free(&new_group);
  MPI_Group_free(&comm_group);

  if (!id) {
    MPI_Put(buff, 2, MPI_SHORT, 1, (MPI_Aint)2, 2, MPI_SHORT, win);
    MPI_Get(buff + 24, 1, MPI_INT, 1, (MPI_Aint)1, 1, MPI_INT, win);
    MPI_Put(buff + 1, 3, MPI_CHAR, 2, (MPI_Aint)1, 3, MPI_CHAR, win);
    MPI_Get(buff + 30, 2, MPI_INT, 2, (MPI_Aint)0, 2, MPI_INT, win);
  }
  MPI_Win_fence(0, win);

  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 2, 0, win);
  if (id == 2) {
    *((char *)buff) = -id;
    MPI_Put(buff, 1, MPI_CHAR, 0, (MPI_Aint)5, 1, MPI_CHAR, win);
  }
  MPI_Win_unlock(2, win);

  if (!id)
    MPI_Accumulate(buff, 3, MPI_CHAR, 3, (MPI_Aint)2, 3, MPI_CHAR, MPI_SUM, win);
  MPI_Win_fence(0, win);

  printf("  medpomnilnik:");
  for (i = 0; i < buff_size; i++)
    printf("%2d%s", *((char *)buff + i), (i == buff_size-1) ? "\n" : ", ");

  MPI_Win_free(&win);
  MPI_Free_mem(buff);

  MPI_Finalize();

  return 0;
}
```

Če program poženemo na npr. 5 procesih lahko dobimo izpis:

```
4)test atribut 'MPI_WIN_SIZE': 20
  medpomnilnik: 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23
0)test atribut 'MPI_WIN_SIZE': 40
  medpomnilnik:13, 14, 15, 16, 17, 18, 19, 20,  8,  9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, -2, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39
2)test atribut 'MPI_WIN_SIZE': 20
  medpomnilnik:-2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21
1)test atribut 'MPI_WIN_SIZE': 20
  medpomnilnik: 0,  1,  2,  3,  4,  5,  6,  7,  9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20
3)test atribut 'MPI_WIN_SIZE': 20
  medpomnilnik: 3,  4,  5,  6,  7,  8,  9, 10, 24, 26, 28, 14, 15, 16, 17, 18, 19,
20, 21, 22
```

2.13 KOMUNIKACIJA PREKO VRAT

MPI omogoča, da se dva neodvisno zagnana programa MPI povežeta in si izmenjujeta sporočila preko vrat. To poteka tako, da proces na eni strani odpre vrata in se poveže z njimi, na drugi strani pa drug proces vzpostavi povezavo z istimi vrati. Vzpostavljena povezava kreira komunikator tipa interkomunikator, preko katerega poteka komunikacija. Proces, ki vrata odpre, imenujemo strežnik, proces, ki se poveže z že odprtimi vrati, pa odjemalec. Nekatere izmed spodaj naštetih funkcij bi lahko uvrstili tudi v skupino za delo s komunikatorji.

Funkcije za delo s komunikacijo preko vrat:

- `int MPI_Open_port(MPI_Info info, char *port_name);`
  Funkcija vzpostavi povezavo z vrati. MPI samodejno izbere številko vrat in jo skupaj z naslovom IP zapiše v parameter *port_name*. Funkcija se uporablja na strani strežnika.
- `int MPI_Close_port(char *port_name);`
  Strežnik s to funkcijo zapre vrata, podana s parametrom *port_name*.
- `int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm);`
  Funkcija na naslovu *newcomm* kreira interkomunikator in ga poveže z vrati, katerih opis je podan s parametrom *port_name*. Uporablja se na strani strežnika.
- `int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm);`
  Funkcija se uporablja na strani odjemalca. S pomočjo parametra *port_name*, ki med drugim vsebuje tudi številko vrat in IP-naslov strežnika, vzpostavi povezavo s

strežnikom.

- `int MPI_Comm_disconnect(MPI_Comm *comm);`
  Funkcija počaka, da se zaključi prenos vseh sporočil, nakar sprosti interkomunikator na naslovu *comm* in postavi njegovo vrednost na *MPI_COMM_NULL*. Funkcijo uporabljata tako strežnik kot odjemalec.

Primer uporabe zgornjih funkcij:

```
mpi_streznik.c:
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int id, msg;
  char port[MPI_MAX_PORT_NAME];
  MPI_Comm inter_comm;
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (!id) {
    MPI_Open_port(MPI_INFO_NULL, &port[0]);
    printf("%s\n", port);
    fflush(stdout);

    MPI_Comm_accept(port, MPI_INFO_NULL, 0, MPI_COMM_SELF, &inter_comm);
    do {
      MPI_Recv(&msg, 1, MPI_INT, 0, 573, inter_comm, &status);
      printf("prejel: %d\n", msg);
    } while (msg);

    MPI_Comm_disconnect(&inter_comm);
    MPI_Close_port(port);
  }

  MPI_Finalize();

  return 0;
}

mpi_odjemalec.c:
#include "mpi.h"

int main(int argc, char *argv[]) {
  int msg = 6;
  MPI_Comm inter_comm;

  if (argc < 2)
    return -1;

  MPI_Init(&argc, &argv);

  MPI_Comm_connect(&argv[1][0], MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_comm);
  MPI_Send(&msg, 1, MPI_INT, 0, 573, inter_comm);
  msg = 0;
  MPI_Send(&msg, 1, MPI_INT, 0, 573, inter_comm);
  MPI_Comm_disconnect(&inter_comm);

  MPI_Finalize();
```

```
  return 0;
}
```

Če najprej poženemo program mpi_streznik, dobimo izpis z informacijo o vratih:

```
tag=0 description=toncy port=22797 ifname=192.168.1.101
```

potem poženemo še program mpi_odjemalec z zgornjim nizom kot prvim atributom in na strežnikovi strani dobimo dodaten izpis:

```
prejel: 6
prejel: 0
```

2.14 DELO Z NAPAKAMI

Vse funkcije MPI, razen funkcij `MPI_Wtime` in `MPI_Wtick`, vračajo številko, ki predstavlja kodo napake. Če se je funkcija izvedla brez napake, je to koda napake *MPI_SUCCESS* z vrednostjo 0. Če med izvajanjem pride do napake, se lahko sproži posebna funkcija, ki upravlja z napakami (angl. *error handler*). Ali bo do klica takšne funkcije prišlo, je odvisno od same implementacije vmesnika MPI. Uporabnik lahko napiše svoje funkcije za delo z napakami, na voljo pa ima že vnaprej definirane:

- *MPI_ERRORS_RETURN* – funkcija ne naredi nič drugega, le vrne številko napake;
- *MPI_ERRORS_ARE_FATAL* – v primeru napake se takoj sproži prekinitev dela na vseh procesih.

Funkcije za delo z napakami lahko pripojimo trem vrstam objektov: komunikatorjem (objektom tipa MPI_Comm), datotečnim ročicam (objektom tipa MPI_File) in objektom za delo z enostransko komunikacijo (objektom tipa MPI_Win). Ko funkcije, ki niso povezane z nobenim izmed zgoraj naštetih objektov, sprožijo napako, se pokliče funkcija za delo z napakami, pripojena komunikatorju *MPI_COMM_WORLD*.

MPI poleg kod napak pozna tudi kode razredov napak. Kode razredov napak so podmnožica kod napak in so prav tako veljavne kode napak.

Funkcije za delo z napakami:

- ```
  int MPI_Comm_create_errhandler(
  MPI_Comm_errhandler_function *function,
  MPI_Errhandler *errhandler);
  ```
  Funkcija kreira funkcijo za delo z napakami, ki jo lahko pripojimo komunikatorju. `MPI_Comm_errhandler_function` je definirana kot:
  ```

```
typedef void MPI_Comm_errhandler_function(
MPI_Comm *, int *, ...);
```

- `int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler);`
  Funkcija pripoji funkcijo za delo z napakami komunikatorju *comm*,
- `int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler);`
  Funkcija na naslov *errhandler* zapiše naslov funkcije za delo z napakami, ki je pripojena komunikatorju *comm*,
- `int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode);`
  Funkcija pokliče funkcijo za delo z napakami, pripojeno komunikatorju *comm*, s številko napake *errorcode*.
- `int MPI_File_create_errhandler( MPI_File_errhandler_function *function, MPI_Errhandler *errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_create_errhandler`, le da tu parameter *errhandler* lahko pripojimo datotečni ročici.
- `int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_set_errhandler`, le da tu parameter *errhandler* pripoji datotečni ročici,
- `int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_get_errhandler`, le da tu na naslov *errhandler* zapiše naslov funkcije za delo z napakami, pripojene datotečni ročici.
- `int MPI_File_call_errhandler(MPI_File fh, int errorcode);`
  Funkcija je enaka funkciji `MPI_Comm_call_errhandler`, le da tu pokliče funkcijo za delo z napakami, pripojeno datotečni ročici.
- `int MPI_Win_create_errhandler( MPI_Win_errhandler_function *function, MPI_Errhandler *errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_create_errhandler`, le da tu parameter *errhandler* lahko pripojimo objektu za delo z enostransko komunikacijo.
- `int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_set_errhandler`, le da tu parameter

errhandler pripoji objektu za delo z enostransko komunikacijo.

- `int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler);`
  Funkcija je enaka funkciji `MPI_Comm_get_errhandler`, le da tu na naslov *errhandler* zapiše naslov funkcije za delo z napakami, pripojene objektu za delo z enostransko komunikacijo.
- `int MPI_Win_call_errhandler(MPI_Win win, int errorcode);`
  Funkcija je enaka funkciji `MPI_Comm_call_errhandler`, le da tu pokliče funkcijo za delo z napakami, pripojeno objektu za delo z enostransko komunikacijo.
- `int MPI_Errhandler_free(MPI_Errhandler *errhandler);`
  Funkcija nastavi naslov funkcije za delo z napakami na *MPI_ERRHANDLER_NULL*. Funkcija za delo z napakami *errhandler* se bo zares sprostila, ko se bo sprostil objekt (komunikator, datotečna ročica ali objekt za delo z enostransko komunikacijo), s katerim je povezana.
- `int MPI_Error_class(int errorcode, int *errorclass);`
  Funkcija pretvori kodo napake v kodo razreda napake.
- `int MPI_Error_string(int errorcode, char *string, int *resultlen);`
  Funkcija na naslov *string* prepiše opis, ki je povezan s kodo napake. Opis je lahko dolg maksimalno *MPI_MAX_ERROR_STRING* znakov.
- `int MPI_Add_error_class(int *errorclass);`
  Funkcija kreira nov razred napak, katerega kodo zapiše na naslov *errorclass*.
- `int MPI_Add_error_code(int errorclass, int *errorcode);`
  Funkcija kreira novo kodo napake in jo doda v razred napak *errorclass*,
- `int MPI_Add_error_string(int errorcode, char *string);`
  Funkcija poveže opis napake s podano kodo napake *errorcode*.

Primer uporabe nekaterih funkcij za delo z napakami:

```
#include "mpi.h"
#include <stdio.h>

void err_handler(MPI_Comm *comm, int *err, ...) {
  if (*err == MPI_SUCCESS)
    return;

  int resultlen, err_class, len;
  char comm_name[MPI_MAX_OBJECT_NAME];
  char err_string[MPI_MAX_ERROR_STRING];

  MPI_Comm_get_name(*comm, &comm_name[0], &resultlen);
  MPI_Error_class(*err, &err_class);
  MPI_Error_string(*err, err_string, &len);
```

```
  printf("Napaka stevilka %d/%d z opisom '%s' ", *err, err_class, err_string);
  printf("se je zgodila v komunikatorju %s\n\n", comm_name);
}

int main(int argc, char *argv[]) {
  int ret, err_code, err_class;
  MPI_Errhandler err;

  MPI_Init(&argc, &argv);

  MPI_Add_error_class(&err_class);
  MPI_Add_error_code(err_class, &err_code);
  MPI_Add_error_string(err_code, "to je opis napake");

  MPI_Comm_create_errhandler(err_handler, &err);
  MPI_Comm_set_errhandler(MPI_COMM_WORLD, err);
  ret = MPI_Comm_delete_attr(MPI_COMM_WORLD, MPI_TAG_UB);
  MPI_Comm_call_errhandler(MPI_COMM_WORLD, ret);
  MPI_Comm_call_errhandler(MPI_COMM_WORLD, MPI_ERR_COMM);
  ret = MPI_Comm_call_errhandler(MPI_COMM_WORLD, MPI_ERR_GROUP);
  MPI_Comm_call_errhandler(MPI_COMM_WORLD, ret);
  MPI_Bcast(NULL, 0, MPI_INT, -1, MPI_COMM_WORLD);
  MPI_Comm_call_errhandler(MPI_COMM_WORLD, err_code);
  MPI_Errhandler_free(&err);

  MPI_Finalize();

  return 0;
}
```

Ob klicu programa z npr. 1 procesom dobimo izpis:

```
Napaka stevilka 537475120/48 z opisom 'Invalid keyval, error stack:
PMPI_Comm_delete_attr(169): MPI_Comm_delete_attr(MPI_COMM_WORLD,
comm_keyval=1681915905) failed
PMPI_Comm_delete_attr(125): Cannot set permanent attribute' se je zgodila v
komunikatorju MPI_COMM_WORLD

Napaka stevilka 537475120/48 z opisom 'Invalid keyval, error stack:
PMPI_Comm_delete_attr(169): MPI_Comm_delete_attr(MPI_COMM_WORLD,
comm_keyval=1681915905) failed
PMPI_Comm_delete_attr(125): Cannot set permanent attribute' se je zgodila v
komunikatorju MPI_COMM_WORLD

Napaka stevilka 5/5 z opisom 'Invalid communicator' se je zgodila v komunikatorju
MPI_COMM_WORLD

Napaka stevilka 8/8 z opisom 'Invalid group' se je zgodila v komunikatorju
MPI_COMM_WORLD

Napaka stevilka 1008280071/7 z opisom 'Invalid root, error stack:
PMPI_Bcast(1478): MPI_Bcast(buf=00000000, count=0, MPI_INT, root=-1,
MPI_COMM_WORLD) failed
PMPI_Bcast(1440): Invalid root (value given was -1)' se je zgodila v komunikatorju
MPI_COMM_WORLD

Napaka stevilka 1073742080/1073741824 z opisom 'to je opis napake' se je zgodila v
komunikatorju MPI_COMM_WORLD
```

3 DREVESNO PREISKOVANJE MONTE CARLO

Drevesno preiskovanje Monte Carlo (angl. *Monte Carlo Tree Search*, MCTS) je družina algoritmov, ki gradi asimetrična drevesa in išče suboptimalne rešitve, saj bi zaradi velikosti iskalnega prostora iskanje optimalne rešitve trajalo predolgo ali pa optimalne rešitve zaradi narave problema sploh ni mogoče najti.

To je eno izmed izjemno hitro rastočih področij raziskovanja, saj je bilo samo v letih 2007–2012 objavljenih skoraj 250 člankov na temo MCTS, kar v povprečju pomeni skoraj 1 članek na teden. Algoritmi MCTS se uporabljajo predvsem na področju umetne inteligence v povezavi z igranjem iger, pri čemer prednjači klasična namizna igra go. Šah, ki je bil včasih sinonim za testiranje umetne inteligence, je po zmagi IBM-ovega računalnika DeepBlue nad velemojstrom Garyjem Kasparovom nadomestil go, saj računalniki šele zdaj dosegajo stopnjo, kjer se lahko kosajo z najboljšimi igralci go-ja (gre predvsem za igralce iz Koreje, z Japonske in Kitajske). Algoritmi MCTS se uporabljajo tudi za umetno inteligenco drugih iger, kot so npr. Gomoku, ConnectFour, TicTacToe, Mancala, Hex, Havannah, Amazons, Ms. Pac-Man, Tron, Magic: The Gathering, Skat, Scotland Yard …, in reševanje drugih problemov (npr. problem trgovskega potnika, fizikalne simulacije, razvrščevalni problemi, …).

Velika prednost algoritmov MCTS je, da jih lahko enostavno apliciramo na vsak problem, kjer lahko izvajamo simulacije, pri tem pa ne potrebujemo specifičnega znanja o področju problema. Seveda nam to znanje lahko koristi in še dodatno izboljša ter pohitri iskanje rešitve. Še ena prednost algoritmov MCTS je, da lahko iskanje rešitve kadar koli prekinemo, saj se rezultati sproti posodabljajo in imamo tako vedno na voljo trenutno najboljšo rešitev. Ker algoritmi MCTS gradijo asimetrična drevesa, se lahko posvetijo delom iskalnega prostora, ki trenutno obetajo boljše rezultate, kljub temu pa še vedno lahko del iskanja namenijo slabo raziskanemu delu prostora, ki lahko prinese boljši rezultat v prihodnosti.

3.1 PREDHODNIKI ALGORITMOV MCTS

Algoritmi MCTS imajo podlago v več teorijah in metodah, med drugim tudi v odločitveni teoriji (angl. *decision theory*) in teoriji iger (angl. *game theory*) ter metodah Monte Carlo in metodah na osnovi igralnega avtomata (angl. *bandit-based methods*).

Odločitvena teorija je kombinacija verjetnostne teorije (angl. *probability theory*) in

uporabnostne teorije (angl. *utility theory*). Preučuje prehode med stanji, pri čemer so ti prehodi pogojeni z verjetnostjo. Primer takšne teorije je Markovski odločitveni proces (angl. *Markov Decision Process*, MDP).

Teorija iger je nadgradnja odločitvene teorije. Vključuje dva ali več igralcev, ki medsebojno vplivajo drug na drugega (natančneje na njihove odločitve). Vsako igro lahko klasificiramo glede na njene lastnosti, ki jih igra ima ali nima:

- ničta vsota (angl. *zero-sum*) – definira, ali boljši rezultat enega igralca avtomatično pomeni slabši rezultat drugih igralcev. Z drugimi besedami ta lastnost pomeni, da je seštevek vseh pridobljenih točk vseh igralcev enak seštevku vseh izgubljenih točk vseh igralcev (če eno vsoto odštejemo od druge, dobimo 0);
- popolna informacija – trenutno stanje igre je v celoti razkrito. Pri igrah z nepopolno informacijo je trenutno stanje igre le deloma razkrito;
- determinizem – pri deterministčnih igrah naključja (verjetnost) ne igrajo nobene vloge. Nasprotje deterministične igre je stohastična[6] igra;
- zaporednost – poteze se izvajajo zaporedno, za razliko od iger, kjer se poteze lahko igrajo sočasno;
- diskretnost – če je igra diskretna, pomeni, da je prehod med dvema stanjema igre časovno neodvisen (čas ne vpliva na potek igre).

Igre za dva igralca, ki imajo vse zgoraj naštete lastnosti (ničta vsota, popolna informacija, determinizem, zaporednost in diskretnost), so deklarirane kot kombinatorične igre.

Metode Monte Carlo so stohastične metode, ki s pomočjo velikega števila ponovitev pridejo do nekih hevrističnih[7] rezultatov, ki se med sabo le malo razlikujejo.

Metode na osnovi igralnega avtomata so metode, pri katerih je treba zaporedoma izbirati med *n* možnostmi (avtomati), tako da se končni rezultat (oz. nagrada) maksimizira. Ker ne vemo, kakšna je porazdelitev rezultatov na posameznem avtomatu, se lahko zanašamo le na predhodne rezultate. To nas pripelje do vprašanja, kako najbolj optimalno porazdeliti izkoriščanje (angl. *exploitation*) avtomata, ki trenutno obeta najboljši rezultat, z raziskovanjem (angl. *exploration*) drugih avtomatov, pri katerih je rezultat na dolgi rok lahko še boljši. Formula za izračun zgornje meje samozavesti (angl. *Upper Confidence Bound*, UCB1) uravnoteži izkoriščanje in raziskovanje, kar je prikazano v enačbi št. 1:

6 Odvisna od naključja.
7 Temelječ na izkušnjah.

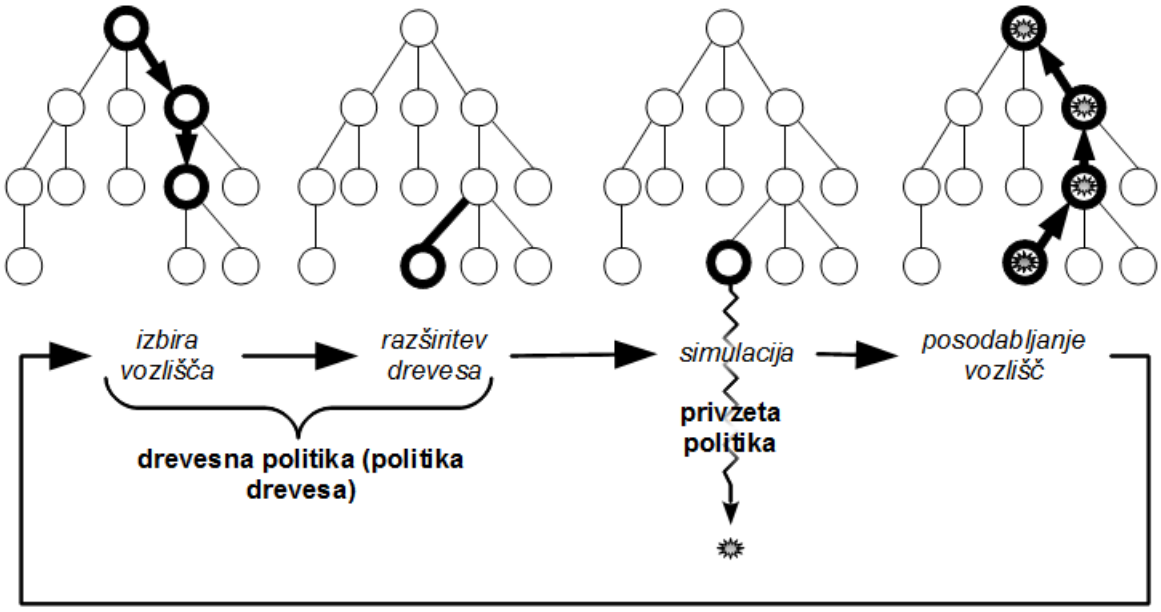
$$UCB1_i = \overline{X}_i + \sqrt{\frac{2\ln(n)}{n_i}};\ 0 \le \overline{X}_i \le 1 \quad (1)$$

$\overline{X}_i$ je povprečen rezultat avtomata i (vrednosti med 0 in 1), n_i število odigranih potez na avtomatu i in n število vseh odigranih potez na vseh avtomatih. Avtomat i, ki ima med vsemi avtomati najvišjo vrednost UCB1 (enačba 1), je najboljša izbira za igranje.

3.2 DELOVANJE ALGORITMOV MCTS

Osnovni element vsakega algoritma MCTS imenujemo iteracija (angl. *iteration*). Vsaka iteracija je sestavljena iz 4 korakov, kar je prikazano na sliki 4:

1. izbira vozlišča (angl. *node selection*),
2. razširitev drevesa (angl. *tree expansion*),
3. simulacija (angl. *simulation*),
4. posodabljanje vozlišč (angl. *backpropagation*).



Slika 4: Osnovni koraki iteracije

Prvima dvema korakoma skupaj pravimo tudi drevesna politika oz. politika drevesa (angl. *tree policy*), tretjemu koraku pa privzeta politika (angl. *default policy*).

Podobno kot pri metodah na osnovi igralnega avtomata moramo tudi pri izbiri vozlišča

uravnotežiti izkoriščanje že znanega prostora in raziskovanje neznanega prostora. V vsakem primeru začnemo s korenskim vozliščem (angl. *root node*) in glede na politiko drevesa iščemo najprimernejše razširljivo (angl. *expandable*) vozlišče. To pomeni, da to ni vozlišče, ki predstavlja končno stanje, in da ima še neraziskane potomce. Ko izberemo primerno vozlišče, ga razširimo – dodamo potomca. Sledi izvajanje simulacije, katere začetna točka je pravkar dodano vozlišče. Simulacija je lahko zelo preprosta (naključno izbrane poteze) ali pa zelo kompleksna (časovno in prostorsko zelo zahteven algoritem), odvisno od privzete politike. Simulacija se zaključi, ko doseže končno stanje igre. V končnem stanju se nato izračuna rezultat, s katerim posodobimo novododano vozlišče in vse njegove predhodnike, vključno s korenskim vozliščem.

V družino algoritmov MCTS je uvrščenih več algoritmov, med drugim tudi:

- Zgornja meja samozavesti za drevesa (angl. *Upper Confidence bounds for Trees,* UCT). UCT uporablja formulo UCB1 (enačba 1) kot osnovo za politiko drevesa. Pri odločitvi, katerega potomca j vozlišča i izbrati za prehod na nižji nivo drevesa, algoritem uporablja enačbo št. 2:

$$UCT = \overline{X_j} + 2C_p\sqrt{\frac{2\ln(n)}{n_j}};\ C_p>0,\ 0\le\overline{X_j}\le 1 \qquad (2)$$

  n_j je število, ki nam pove, kolikokrat je bilo obiskano vozlišče j, n pa število iteracij. C_p je konstanta, ki mora biti večja od 0, tipično je njena vrednost $1/\sqrt{2}$. Z večanjem vrednosti konstante C_p se pomikamo v smer raziskovanja malo obiskanega dela drevesa v škodo premikov v del drevesa, kjer so rezultati trenutno boljši. $\overline{X_j}$ je povprečna ocena vozlišča j, ki mora biti iz intervala [0, 1]. Algoritem izračuna UCT (enačba 2) za vse potomce vozlišča i in izbere vozlišče z največjo vrednost UCT (enačba 2). Če katero vozlišče j, ki je neposredni potomec vozlišča i, ni bilo še nikoli obiskano (njegov $n_j = 0$), bo njegov UCT = ∞; to nam zagotovi, da bodo vsi potomci vozlišča i dodani v drevo, preden se bo kateri izmed njegovih neposrednih potomcev razširil naprej.
- MCTS za enega igralca (angl. *Single-Player MCTS*, SP-MCTS) je, kot že samo ime pove, algoritem, primeren za igre z enim igralcem. V tem algoritmu formuli UCT dodamo (prištejemo) še en člen, tako da dobimo enačbo št. 3:

$$UCT_{SP-MCTS} = \overline{X_j} + 2C_p\sqrt{\frac{2\ln(n)}{n_j}} + \sqrt{\sigma^2+\frac{D}{n_j}};\ C_p>0,\ 0\le\overline{X_j}\le 1 \qquad (3)$$

  σ^2 je varianca rezultatov simulacij vozlišča, n_j je število obiskov vozlišča in D konstanta. Malo obiskanim vozliščem se umetno (s parametrom D/n_j) povečuje standardni odklon, kar zmanjšuje zanesljivost njihovih rezultatov. Takšni algoritmi uporabljajo hevristične simulacije.

- MCTS za več igralcev (angl. *Multi-Player MCTS*) je algoritem, ki v vozlišču ne hrani samo enega rezultata, ampak hrani tabelo rezultatov. To je potrebno zato, ker bi v nasprotnem primeru lahko izračunali le vsoto vseh rezultatov drugih igralcev. Ker je v več igrah mogoče sklepanje koalicij, mora algoritem paziti, da s svojo potezo čim manj škoduje igralcem iz iste koalicije.
- MCTS v realnem času (angl. *Real-Time MCTS*) je primeren za hitre arkadne igre. V primeru klasičnih namiznih iger si lahko algoritem privošči veliko časa za iskanje najboljše poteze, pri hitrih arkadnih igrah pa to ne velja. Ker MCTS sodi v skupino algoritmov, ki imajo rezultate vedno na voljo, je primeren tudi za tak tip iger. Zaradi zahteve po hitrem produciranju rezultatov si je pogosto treba pomagati s približki.
- Nedeterminističen MCTS (angl. *Nondetrministic MCTS*) je algoritem, ki mora upoštevati verjetnost in/ali nepopolno informacijo. V resnici gre tu za poddružino algoritmov MCTS, v katero spadajo npr. HOP (angl. *Hindsight OPtimisation*), UCT+, MCαβ (Monte Carlo α-β), MCCFR (angl. *Monte Carlo CunterFactual Regret*), …
- Rekurzivni MCTS (angl. *Recursive MCTS*) je poddružina algoritmov MCTS, ki gradi drevo s pomočjo rekurzije. V to poddružino med drugim sodijo algoritmi NMCS (angl. *Nested Monte Carlo Search*), NRPA (angl. *Nested Rollout Policy Adaptation*), Meta-MCTS, HGSTS (angl. *Heuristcally Guided Swarm Tree Search*), …
- Načrtovanje na podlagi vzorcev (angl. *Sample-Based Planners*) je še ena poddružina algoritmov MCTS, pri katerih vzamemo dovolj dobre enostavne rešitve, jih prilagodimo in uporabimo pri kompleksnejšem problemu. Sem med drugim sodijo algoritmi FSSS (angl. *Forward Search Sparse Sampling*), RRTs (angl. *Rapidly-exploring Random Trees*), ρUCT, MCRW (angl. *Monte Carlo Random Walks*), …

3.3 RAZŠIRITVE ALGORITMOV MCTS

Razširitve drevesne politike algoritmov MCTS lahko razdelimo na dve skupini:

1. Razširitve, odvisne od področja (domene) – tipično gre za boljše rešitve, saj lahko izkoristimo predhodno znanje z dotičnega področja. Njihova slabost je neprenosljivost (ali delna neprenosljivost) na druga področja.
2. Razširitve, neodvisne od področja – za njih velja, da so prenosljive med področji, saj ne zahtevajo predhodnega znanja o specifičnem področju. V tem primeru izboljšave po navadi prinesejo manjše izboljšanje kot izboljšave, vezane na točno določeno področje, oz. je izboljšanje v nekaterih domenah boljše kot v drugih.

Nekaj razširitev drevesne politike:

- Bayesov UCT (angl. *Bayesian UCT*) je razširitev, ki lahko potencialno pridobi natančnejšo oceno vrednosti vozlišča (v primerjavi z navadnim algoritmom UCT) iz omejenega števila simulacijskih poskusov. Razširitev uporablja 2 drevesni politiki, ki ju prikazujeta enačbi št. 4 in 5:

$$\max B_i = \mu_i + \sqrt{\frac{2\ln(N)}{n_i}} \tag{4}$$

$$\max B_i = \mu_i + \sqrt{\frac{2\ln(N)}{n_i}}\,\sigma_i \tag{5}$$

  μ_i je povprečje ekstremov porazdelitve P_i in σ_i je standardni odklon porazdelitve P_i. Izkaže se, da enačba 5 prinese boljše rezultate kot enačba 4, obe pa dajeta boljše rezultate kot navaden UCT (enačba 2), vendar je navaden UCT bistveno hitrejši.
- Urgentna prva igra (angl. *First Play Urgency, FPU*) je razširitev, ki dodeli vsem neobiskanim vozliščem neko fiksno vrednost (rezultat). Z uravnavanjem te vrednosti lahko omogočimo hitrejše izkoriščanje na škodo raziskovanja.
- Premik skupin (angl. *Move Groups*) je uporabna razširitev pri igrah z veliko možnimi potezami, kjer so si nekatere med seboj zelo podobne. Podobne poteze lahko združimo v skupine. Nato z UCB1 (enačba 1) izberemo skupino in igramo potezo iz te skupine.
- Transpozicija (angl. *Transpositions*) je razširitev, ki povezuje drevesa MCTS in usmerjene aciklične grafe (angl. *Directed Acyclic Graph*, DAG). V veliko primerih lahko igro, katere poteze išče MCTS, predstavimo kot usmerjen aciklični graf, saj lahko neko stanje igre dosežemo z različnimi potezami. Ker je drevo MCTS praviloma precej večje kot aciklično usmerjen graf, lahko dve popolnoma različni poti v drevesu MCTS pripeljeta v isto vozlišče aciklično usmerjenega grafa. Če informacije o vozliščih aciklično usmerjenega grafa shranimo, si lahko z njimi pomagamo pri izbiri potez. Če se v drevesu MCTS pojavi identičen par (stanje, poteza), tako že poznamo končni rezultat.
- Postopna pristranskost (angl. *Progressive Bias*) je razširitev, ki uporablja hevristično znanje o domeni, ki ima večjo težo, če je vozlišče malokrat obiskano, in obratno – z večanjem števila obiskov vozlišča pomembnost hevurističnega znanja pada. Matematično lahko to zapišemo kot enačbo št. 6:

$$f(n_i) = \frac{H_i}{n_i + 1} \tag{6}$$

  H_i je hevristična vrednost vozlišča i, n_i pa število obiskov vozlišča i.
- Vse poteze kot prve (angl. *All Moves As First*, AMAF), je razširitev, ki si med simulacijo zapomni vsako potezo. Nato s končnim rezultatom simulacije popravi vsa

vozlišča v drevesu MCTS, ki predstavljajo eno izmed potez simulacije, tudi če niso del poti od korenskega vozlišča do pravkar dodanega vozlišča. Obstajajo tudi izpeljanke algoritma AMAF, kot so npr. α-AMAF, Some-First AMAF, Cutoff AMAF, …

- Hitra ocena vrednosti akcije (angl. *Rapid Action Value Estimation*, RAVE) je razširitev algoritma AMAF (natančneje razširitev algoritma α-AMAF). Vrednost RAVE se izračuna po enačbi št. 7:

$$RAVE = max\left\{0, \frac{V - v(n)}{V}\right\} AMAF + \left(1 - max\left\{0, \frac{V - v(n)}{V}\right\}\right) UCT; \quad V > 0 \qquad (7)$$

  V je neničelna celoštevilska konstanta, $v(n)$ pa število obiskov vozlišča v. Ko število obiskov nekega vozlišča doseže konstanto V, se vrednost RAVE (enačba 7) ne upošteva več. Vsako vozlišče mora hraniti dva rezultata, enega pridobljenega z algoritmom UCT (enačba 2) in drugega pridobljenega z algoritmom RAVE (enačba 7). Tako kot osnovni algoritem AMAF ima tudi algoritem RAVE izpeljanke, kot so npr. Killer RAVE, RAVE-max in PoolRAVE.
- Iskanje dokazanih števil (angl. *Proof Number Search*, PNS) je razširitev, ki se uporablja v igrah, katerih končni rezultat je lahko zmaga, poraz ali neodločen izid. PNS je algoritem, katerega končna stanja so lahko dokazano zmagovalna (angl. *proven win*) z rezultatom +∞ ali pa dokazano poražena (angl. *proven loss*) z rezultatom -∞. Za vsa druga stanja velja, da je stanje dokazano zmagovalno, če je vsaj en njegov potomec dokazano zmagovalen, in da je stanje dokazano poraženo, če so vsi njegovi potomci dokazano poraženi. V drevesu MCTS nadomestimo vse rezultate na končnih vozliščih z dokazano zmagovalnimi ali dokazano poraženimi in posodobimo njihove predhodnike. Tako pridemo do delov drevesa, ki precej zanesljivo vodijo v zmago oziroma poraz.
- Rezanje premikov (angl. *Move Pruning*) je razširitev, s pomočjo katere lahko začasno (angl. *soft pruning*) ali trajno (angl. *hard pruning*) odstranimo dele drevesa, ki prinašajo očitno slabe rezultate.

Poleg razširitev drevesne politike lahko algoritme MCTS izboljšamo še z boljšim simulacijskim delom, izboljšanim posodabljanjem vozlišč in paralelizacijo algoritma.

Osnovna simulacija je preprosto naključno izbiranje med posameznimi možnimi potezami. Bolj ko poznamo naravo problema, bolj realistične so lahko simulacije. Simulacije lahko uporabljajo tudi katerega izmed učnih algoritmov. Izvajanje simulacij lahko nadzorujemo s pravili, ki se jih mora simulacija držati. Tipično takšne simulacije niso dosti počasnejše od naključno izbranih potez, vendar so bolj ali manj vezane na posamezno domeno. Primer izboljšave simulacij, ki je neodvisen od domene, je povezovanje simulacij, ki se odvijajo v

istem območju (relativno blizu glede na velikost prostora). Z uporabo statistike že končanih simulacij lahko usmerjamo nove simulacije. Takšna razširitev se imenuje kontekstno iskanje Monte Carlo (angl. *Contextual Monte Carlo Search*). Algoritem z imenom napolni ploščo (angl. *Fill the Board*) je bil razvit za izboljšanje simulacij v igri go. Algoritem izbere *n* različnih presečišč in v primeru praznega prostora presečišča in njegovih sosedov igra na tem mestu, drugače pa izbere naključno veljavno potezo. Zanimiva razširitev za simulacije je algoritem zadnji dobri odgovor (angl. *Last Good Reply*, LGR), kjer je vsaka poteza obravnavana kot odgovor na prejšnjo potezo. Za vsako potezo se shrani zadnji uspešen odgovor nanjo, ki se uporabi v primeru ponovitve iste poteze. Vzorci (angl. *patterns*) so še ena razširitev osnovne simulacije. To so v primeru namiznih iger neprazni, majhni koščki igralne površine. Če simulacija v igri odkrije takšen vzorec, izvrši točno predvideno potezo.

Pri izboljšavah posodabljanja vozlišč velja omeniti obtežitev rezultatov (angl. *Weighting Simulation Results*). Rezultati simulacij, ki so bile izvršene pozneje in so krajše, veljajo za boljše, zato je tudi obtežitev njihovih rezultatov večja. Algoritem z imenom meje rezultata (angl. *Score Bounds*) loči med močnimi in šibkimi zmagami (in porazi). Namesto vrednosti 0 za poraz in 1 za zmago je vrednost za poraz izbrana iz intervala $[0, \gamma]$, vrednost za zmago pa z intervala $[\gamma, 1]$. Razpadajoča nagrada (angl. *Decaying Reward*) je algoritem, ki predpostavlja, da so zmage na začetku vredne več kot tiste na koncu. Vsak rezultat pred posodobitvijo vozlišč pomnoži z vrednostjo γ iz intervala $(0, 1]$, glede na njegovo pomembnost.

Paralelizacija lahko znatno prispeva k pohitritvi izvajanja algoritma MCTS oz. k njegovi učinkovitosti. Dejansko ločimo 4 načine paralelizacije algoritmov MCTS:

1. Paralelizacija listov/simulacij (angl. *leaf parallelization*) – z več simulacijami pridobimo bolj zanesljive rezultate. Slaba stran te paralelizacije je, da moramo vedno čakati na najpočasnejšo nit/proces, da zaključi s simuliranjem, preden dobimo rezultat.
2. Korenska paralelizacija (angl. *root parallelization*) – v tem primeru gradimo več medsebojno neodvisnih dreves, zato takšno paralelizacijo včasih poimenujemo tudi večdrevesni MCTS (angl. *multi-tree* MCTS). Prednost takšne paralelizacije je, da lahko za vsako drevo posebej določimo, koliko časa se bo gradilo.
3. Paralelizacija drevesa z globalno ključavnico (angl. *tree parallelization with global mutex*) – ena nit/proces gradi drevo, medtem ko druge izvajajo simulacije. Ta pristop je smiseln, če so simulacije časovno potratne v primerjavi z gradnjo drevesa.
4. Paralelizacija drevesa z lokalnimi ključavnicami (angl. *tree parallelization with local mutexes)* – tukaj uporabljamo lokalne ključavnice, s katerimi zaklenemo vozlišče, ko ga nit/proces obišče, in ga odklenemo, ko nit/proces vozlišče zapusti.

4 PARALELIZACIJA KODE Z MPI

Koda je paralelizirana na dva načina:

1. paralelizacija iger,
2. paralelizacija simulacij.

Uporabnik pred zagonom programa določi, kateri način paralelizacije bo uporabil. Namen paralelizacije kode je predvsem pohitritev izvajanja in/ali doseganje boljših rezultatov v istem času v primerjavi s kodo, ki se izvaja sekvenčno[8].

4.1 PARALELIZACIJA IGER

Ideja je, da se glavno zanko, ki se odvrti *n*-krat (pri čemer je *n* število različnih iger, ki jih želimo paralelizirati), razdeli na *p* procesov, ki se izvajajo vzporedno. Ko vsi procesi zaključijo delo (tj. odigrajo vse igre, ki so jim bile dodeljene), se rezultati prenesejo na izbrani proces, ki jih obdela in izračuna statistiko (tj. število zmag, porazov in neodločenih izidov, standardni odklon rezultatov, …).

Pri paralelizaciji iger moramo vedeti, koliko procesov imamo na voljo. To informacijo pridobimo s pomočjo funkcije `Get_size`, katere rezultat shranimo v spremenljivko `process_total`. Nato izračunamo, koliko različnih iger bo moral posamezen proces odigrati, kar predstavlja vrednost spremenljivke `num_games_per_process`. Če je število vseh procesov nedeljivo s številom vseh iger, bo moral vsaj en proces odigrati eno igro več kot drugi. Spremenljivka `num_games_per_process` je tako v takem primeru pri določenih procesih za 1 večja kot pri drugih. Spremenljivka `num_games_tmp` nam pove, kolikokrat se bo glavna zanka zavrtela, in je na vseh procesih enaka, medtem ko spremenljivka `num_games` hrani število vseh iger. Koda, ki izračuna zgoraj naštete vrednosti:

```
int num_games_per_process = num_games / process_total;
int num_games_tmp = num_games_per_process + ((num_games % process_total) ? 1 : 0);
if (process_id < (num_games % process_total))
  num_games_per_process++;
```

Procesi si nato rezervirajo prostor za tabelo, v katere si bodo začasno shranjevali rezultate. Rezultate bodo na koncu predali procesu s položajem 0 (v nadaljevanju glavni proces). Glavni proces si mora zato rezervirati dodatno tabelo, v kateri bo na koncu zbral vse rezultate:

8 Zaporedno.

```
process_score = new double [num_games_tmp*number_players];
if (!process_id)
  gather_score = new double[num_games_tmp*process_total*number_players];
```

Nato v zanki, ki se odvrti `num_games_tmp`-krat, igramo igre. Preden poženemo igranje, preverimo, če je posamezen proces že odigral zahtevano število iger, torej če je igranje sploh potrebno. Če igra ni igrana, je lahko na mestu v tabeli, kjer bi moral biti rezultat trenutne igre, kar koli. Posamezna igra se igra tako, da algoritem dobi trenutno stanje igre in poizkuša s pomočjo algoritma UCT (enačba 2) predvideti najboljšo potezo ter jo odigra. To se ponavlja, dokler se igra ne zaključi. Po koncu igre se izračunajo končni rezultati za vse igralce, ki se shranijo v tabelo `process_score`.

Ko proces odigra vse igre, se ustavi pri funkciji `Gather`, kjer počaka, da tudi drugi procesi do konca odigrajo svoje igre. Sledi prenos tabele `process_score` vseh procesov h glavnemu procesu, ki jih združi v tabeli `gather_score`:

```
MPI::COMM_WORLD.Gather(&process_score[0], num_games_tmp*number_players,
MPI::DOUBLE, &gather_score[0], num_games_tmp*number_players, MPI::DOUBLE, 0);
```

Na koncu se glavni proces sprehodi čez celo tabelo `gather_score`, pri čemer izpusti podatke o rezultatih neigranih iger. Igra je bila igrana, če je številka igre (šteti se začne pri 0) manjša od števila vseh iger:

```
for (int g = 0; g < num_games_tmp*process_total; g++) {
  int game_number = g%num_games_tmp*process_total + g/num_games_tmp;
  bool game_skipped = (game_number < num_games) ? false : true;
#if 0 /* izpis */
  printf("\nrepeate %d) process %d %s game %3d", r, g/num_games_tmp,
    game_skipped ? "skipped" : "played ", game_number + 1);
  if (!game_skipped)
    printf(", player %d get score %f", return_score_player_num + 1,
      gather_score[g*number_players + return_score_player_num]);
#endif
  if (game_skipped)
    continue;

  // shranjevanje rezultatov
  // . . .
}
```

Rezultat *i*-tega igralca igre številka `game_number` je tako na mestu `gather_score[game_number*number_players + i]`, pri čemer mora veljati, da je vrednost spremenljivke `game_number` manjša od vrednosti spremenljivke `num_games`.

4.2 PARALELIZACIJA SIMULACIJ

Ker se po navadi izvaja le 1 simulacija na iteracijo, na prvi pogled paralelizacija simulacij v časovnem smislu ne prinese nobene pohitritve. Še več, iteracije so lahko celo počasnejše, saj lahko ena simulacija traja dlje kot druge, nato pa je treba rezultate vseh simulacij še zbrati in preračunati oz. z rezultatom vsake simulacije posodobiti vozlišča. Vendar nam več simulacij na iteracijo pomaga, da dosežemo občutno boljši rezultat. To pa lahko izkoristimo tako, da zmanjšamo število iteracij na potezo in pridobimo čas. Pri tem se moramo zavedati, da z zmanjševanjem števila iteracij pada tudi uspešnost algoritma.

V funkcijah, ki izvajajo simulacije, nastopajo izbire naključnih vrednosti (klic funkcije `rand`). Ker si želimo, da se simulacije v različnih igrah odvrtijo različno, samo nastavitev semena na podlagi časa ne zadošča, zato vanjo vključimo tudi položaj posameznega procesa:

```
srand((unsigned int)time(NULL) + process_id*10000);
```

V primeru paralelizacije simulacij drevo MCTS gradi samo glavni proces, drugi pa se v aktivno izvajanje algoritma vključijo le, ko je na vrsti simulacija. To je lepo vidno v spodnjem delu kode:

```
for (MCTS_current_iterNum = 0;
     MCTS_current_iterNum < UCT_param_iterNum;
     MCTS_current_iterNum++) {
  if (parallelize_simulations < 1) {
    selected_leaf = UCT_Tree_Policy(UCTroot);
  } else {
    if (!process_id)
      selected_leaf = UCT_Tree_Policy(UCTroot);
  }
  final_rewards = UCT_Default_Policy();
  if (parallelize_simulations < 1) {
    UCT_Backup(selected_leaf, final_rewards);
  } else {
    if (!process_id)
      UCT_Backup(selected_leaf, final_rewards);
  }
}
```

Parameter `parallelize_simulation` nam pove, ali paraleliziramo simulacije (v tem primeru je njegova vrednost večja ali enaka 1) ali igre (vrednost manjša od 1), pri čemer vrednost 0 pomeni, da izvajamo 1 simulacijo na iteracijo, absolutna vrednost neničelnih vrednosti pa število simulacij na iteracijo.

V funkciji `UCT_Default_Policy` moramo najprej glede na parameter `parallelize_simulation` določiti, koliko simulacij na iteracijo bomo izvedli in število

procesov, ki jih imamo za to na voljo:

```
const int num_sim = (!parallelize_simulations) ? 1 : abs(parallelize_simulations);
const int process_total = (parallelize_simulations < 1) ?
    1 : MPI::COMM_WORLD.Get_size();
```

Sledi uporaba podobne računice, kot je uporabljena pri paralelizaciji iger, s pomočjo katere izračunamo, koliko simulacij mora opraviti določen proces:

```
int num_sim_per_process = num_sim / process_total;
int num_sim_tmp = num_sim_per_process + ((num_sim % process_total) ? 1 : 0);
if (parallelize_simulations > 0) // samo ob paralelizaciji simulacij
  if (process_id < (num_sim % process_total))
     num_sim_per_process++;
```

Če je število simulacij večje od 1, mora glavni proces drugim razposlati informacije o trenutnem stanju igre. Vsak proces si nato naredi lokalno kopijo trenutne igre, saj jo bo potreboval, če bo izvajal več simulacij (vsaka simulacija mora začeti na isti točki). Procesi si pripravijo tabelo, kamor bodo shranjevali rezultate. Sledi zanka, ki se na vsakem procesu zavrti `num_sim_tmp`-krat. V vsaki neprvi ponovitvi zanke se stanje igre ponastavi. Nato je na vrsti izvajanje simulacij. Če je število procesov nedeljivo s številom zahtevanih simulacij na iteracijo, nekateri procesi izvedejo eno simulacijo več kot drugi. Vsakič ko se simulacija zaključi, se njen rezultat prišteje v lokalno spremenljivko. Ko vsi procesi končajo izvajanje simulacij, svoje rezultate pošljejo glavnemu procesu, ta pa jih sešteje z uporabo funkcije `Reduce`. Glavni proces nato pravkar seštete rezultate vseh procesov (tj. rezultate vseh simulacij) samo še deli s številom vseh simulacij (tj. z vrednostjo spremenljivke `num_sim`). Koda, ki opravlja zgoraj opisani postopek, je naslednja:

```
if (num_sim > 1) {
  for (int i = 0; i < number_players; i++) {
    process_score[i] = 0.0;
    total_score[i] = 0.0;
  }
  if (parallelize_simulations > 0) {
    MPI::COMM_WORLD.Bcast(&simulatedGame->board_state[0],
        simulatedGame->board_size, MPI::CHAR, 0);
    MPI::COMM_WORLD.Bcast(&simulatedGame->current_number_moves[0],
        number_players, MPI::INT, 0);
    for (int i = 0; i < number_players; i++)
      MPI::COMM_WORLD.Bcast(&simulatedGame->current_moves[i][0],
          simulatedGame>maximum_allowed_moves, MPI::BOOL, 0);
    MPI::COMM_WORLD.Bcast(&simulatedGame->score[0],
        number_players, MPI::DOUBLE, 0);
    MPI::COMM_WORLD.Bcast(&simulatedGame->current_player, 1, MPI::INT, 0);
    MPI::COMM_WORLD.Bcast(&simulatedGame->game_ended, 1, MPI::BOOL, 0);
    MPI::COMM_WORLD.Bcast(&simulatedGame->current_plys, 1, MPI::INT, 0);
    MPI::COMM_WORLD.Bcast(&internalGame->current_plys, 1, MPI::INT, 0);
  }
  if (num_sim_tmp > 1)
    simulatedGameStartPosition->Copy_Game_State_From(simulatedGame, false);
```

```
}

for (int s = 0; s < num_sim_tmp; s++) {
  if (s) {
    simulatedGame->Copy_Game_State_From(simulatedGameStartPosition, false);
  }
  bool simulation_skipped;
  if (s < num_sim_per_process) {
    simulation_skipped = false;
    // izvajaj simulacijo do konca igre
    // . . .
  } else {
    simulation_skipped = true;
  }
  if (num_sim == 1) { // ena simulacija na iteracijo
    return simulatedGame->score;
  } else { // več kot 1 simulacija na iteracijo
    if (!simulation_skipped) {
      for (int i = 0; i < number_players; i++) {
        process_score[i] += simulatedGame->score[i];
      }
    }
  }
}

if (parallelize_simulations > 0) {
  MPI::COMM_WORLD.Reduce(&process_score[0], &total_score[0],
      number_players, MPI::DOUBLE, MPI::SUM, 0);
  if (!process_id) {
    for (int i = 0; i < number_players; i++) {
      simulatedGame->score[i] = total_score[i] / num_sim;
    }
  }
} else {
  for (int i = 0; i < number_players; i++) {
    simulatedGame->score[i] = process_score[i] / num_sim;
  }
}

return simulatedGame->score;
```

5 MERITVE

Vse meritve so bile izvedene na dveh različnih računalnikih. Prvi računalnik je prenosnik Acer Aspire V3-771G s štirijedrnim procesorjem Intel Core i7-3610QM s taktom ure 2,3 GHz in 6 MB predpomnilnika. Procesor podpira dve niti na enem fizičnem jedru, kar so pri Intelu poimenovali hipernitna tehnologija (angl. *Hyper-Threading Technology*), tako da je s strani operacijskega sistema (v tem primeru Windows 7) vidnih 8 procesorjev. Drugi računalnik je prenosnik Dell Latitude E5420 z dvojedrnim procesorjem Intel Core i5-2410M s taktom ure 2,3 GHz in 3 MB predpomnilnika. Operacijski sistem, ki teče na tem računalniku, je Ubuntu 13.10. Ker tudi tu procesor podpira hipernitno tehnologijo, ga operacijski sistem vidi kot 4 procesorje.

Vsaka meritev je bila izvedena nad tremi različnimi klasičnimi igrami za dva igralca: Gomoku, ConnectFour in TicTacToe. Pri vseh igrah velja, da prvo potezo izvede nasprotni algoritem (igralec), ki je navaden algoritem UCT. Paraleliziran algoritem UCT, katerega uspešnost merimo, izvede drugo potezo, nakar nasprotni igralec izvede tretjo potezo. Paraleliziran algoritem UCT nanjo odgovori in tako se igra igrá do konca (tj. dokler eden od igralcev ne zmaga oz. dokler ne zmanjka veljavnih potez, kar pomeni, da je končni rezultat neodločen). Ob koncu vsake igre igralec pridobi 1 točko za zmago, 0,5 točke za neodločen rezultat in 0 točk za poraz. Posamezna meritev se vedno izvede nad 10.000 igrami, nakar se izračuna uspešnost igralca. Največje možno število zbranih točk v eni meritvi je 10.000. Uspešnost algoritma (tj. odstotek pridobljenih točk v meritvi) se povečuje z večanjem števila iteracij na potezo in z večanjem števila simulacij na iteracijo.

Igra Gomoku se tipično igra na igralni plošči velikosti 19 x 19, kjer igralca izmenično polagata svoje kamenčke (žetone) na prosto polje na igralni površini. Zmaga tisti igralec, ki prvi navpično, vodoravno ali po diagonali poveže 5 svojih kamenčkov. Program uporablja igralno površino velikosti 7 x 7, saj bi drugače za igranje igre porabili precej več časa. Kljub temu, da je igralna površina manjša, traja igranje te igre najdlje.

Pri igri ConnectFour igralca spuščata svoje kamenčke z vrha posebne igralne plošče, ki meri 7 stolpcev krat 6 vrstic. Kamenčki se tako nabirajo drug na drugem, zmaga pa tisti igralec, ki prvi navpično, vodoravno ali po diagonali poveže 4 svoje kamenčke.

Igra TicTacToe je nekakšna pomanjšana različica igre Gomoku. Od vseh treh iger zanjo potrebujemo najmanj časa. Tipično se igra na listu papirja, kjer en igralec v prosta polja

igralne površine 3 x 3 riše križce, drugi pa krožce. Zmaga igralec, ki prvi navpično, vodoravno ali po diagonali poveže svoje tri znake.

5.1 PARALELIZACIJA IGER

V prvi seriji meritev nas zanima, kako se spreminja čas izvajanja posamezne meritve (čas igranja serije 10.000 iger), če povečujemo število procesov, med katere se porazdelijo igre ene meritve. Tako pri enem procesu ta proces odigra vseh 10.000 iger, pri dveh procesih odigrata vsak 5.000 iger, pri treh procesih odigra prvi proces 3.334 iger druga dva pa vsak po 3.333 iger, … Pri vseh meritvah opazovani algoritem in njegov nasprotnik opravita 400 iteracij na potezo, pri čemer se v vsaki iteraciji izvede samo 1 simulacija.

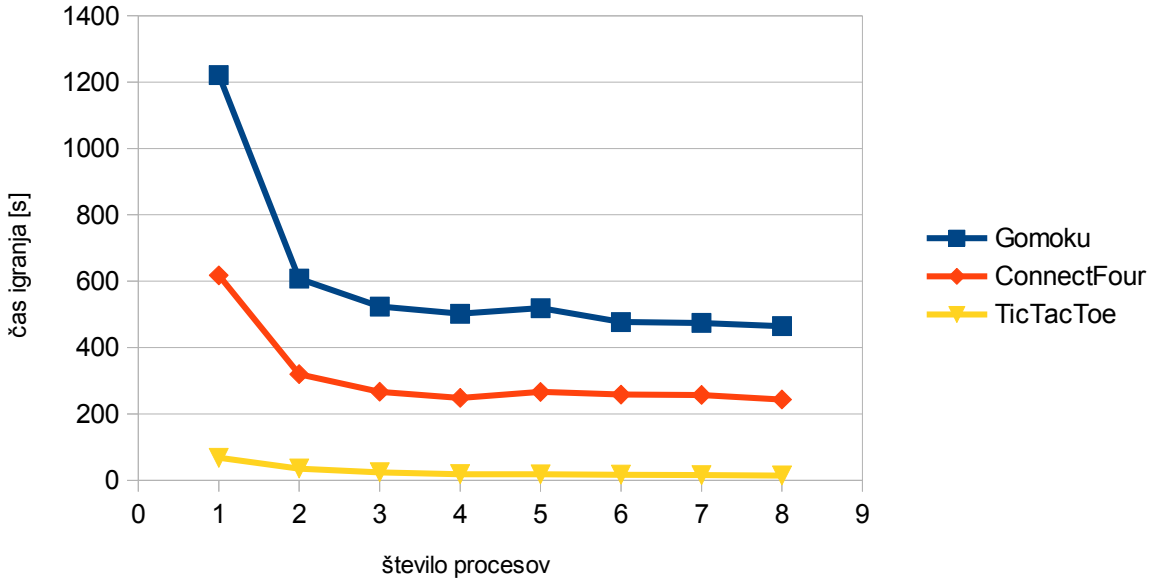
Uspešnost algoritma se med posameznimi meritvami ne spreminja in se vrti okoli 45 %. To je tudi pričakovano, saj se število iteracij na potezo in število simulacij na iteracijo pri obeh igralcih med posameznimi meritvami ne spreminja. Spreminja se le čas izvajanja posamezne meritve.

Na prvem računalniku čas igranja pada s povečevanjem števila procesov do števila 4. Pri petih procesih se čas igranja poveča, nato pa spet pade. Izjema je igra TicTacToe. Pohitritev pri prehodu iz dveh na štiri procese ni tako očitna kot pri prehodu iz enega na dva procesa. Tukaj moramo upoštevati, da računalnik poleg našega programa izvaja tudi druge procese in mora operacijski sistem vsaj nekaj procesorskega časa nameniti tudi njim. Če opazujemo prehod s štirih na pet procesov, se v primeru iger Gomoku in ConnectFour čas celo poveča. To lahko razložimo s tem, da pri štirih procesih zasedemo vsa fizična jedra, nakar mora operacijski sistem pri procesih, ki trajajo dalj časa, večkrat izvesti zamenjavo. Poleg tega je pri prostorsko potratnejših procesih večja verjetnost zgrešitve podatka v predpomnilniku, kar ima za posledico dodaten čas zaradi prenosa podatkov med pomnilnikom in predpomnilnikom. Z večanjem števila procesov se povečuje komunikacija (količina prenesenih podatkov) med samimi procesi, kar dodatno zmanjša hitrost izvajanja programa. Na drugem računalniku, ki ima le dve jedri, se podobno zgodi pri prehodu z dveh na tri procese.

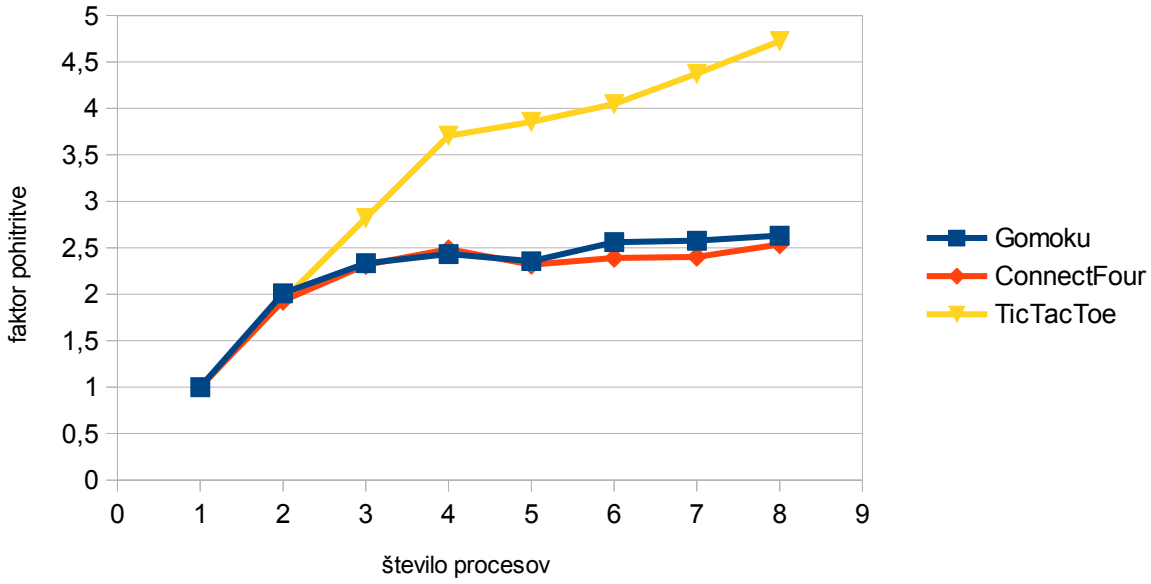
Iz meritev lahko sklepamo, da povečevanje števila vzporedno izvajajočih se procesov nad fizično število procesorjev/jeder, ki jih imamo na voljo, ni smiselno.

Slaba stran takšne paralelizacije je vzporedna gradnja več dreves, ki si med sabo ne izmenjujejo nobenih informacij. To pomeni, da so lahko (predvsem v manjših preiskovalnih

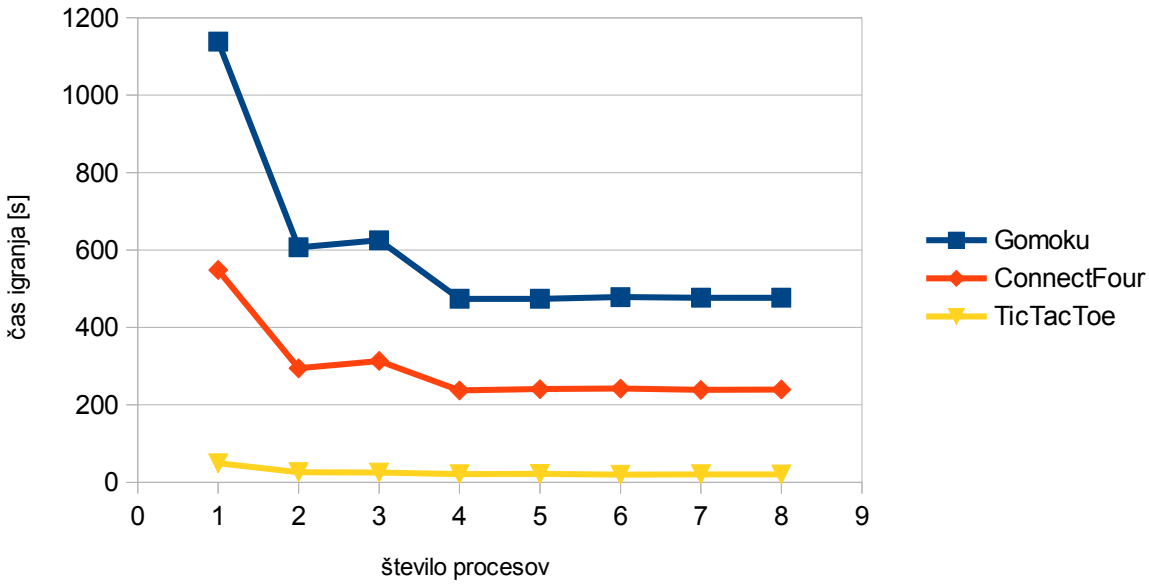
prostorih) deli posameznih dreves identični in nam zato ne dajo nobenih novih informacij. Rezultati te serije meritev so prikazani na slikah 5 in 6 za meritve, izvedene na prvem računalniku, in na slikah 7 in 8 za meritve, izvedene na drugem računalniku.



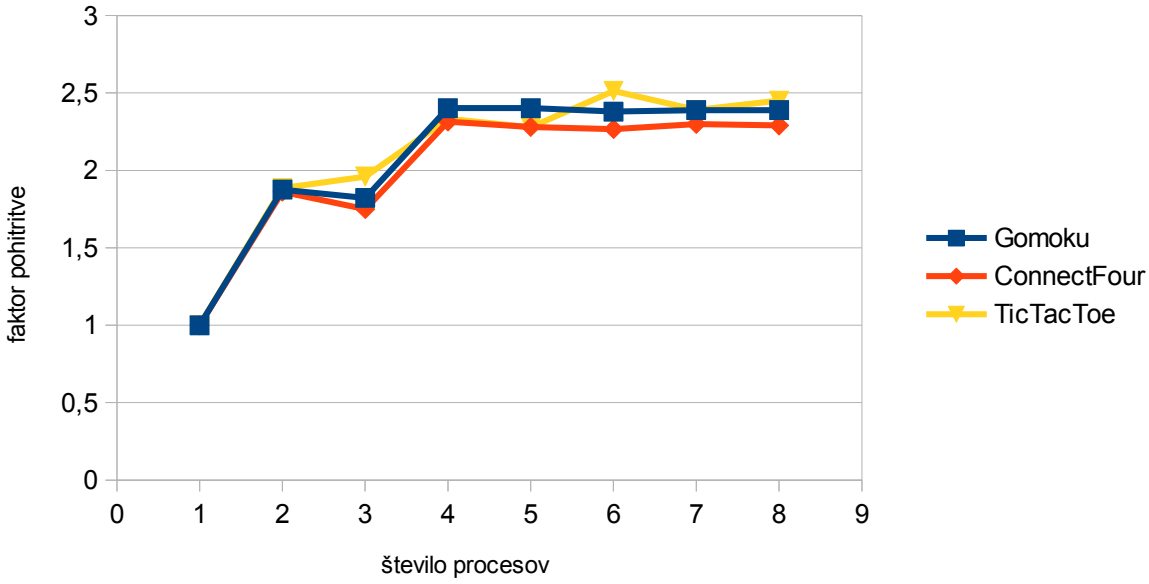
Slika 5: Čas igranja serije 10.000 paraleliziranih iger v odvisnosti od števila procesov na prvem računalniku



Slika 6: Pohitritev igranja serije 10.000 paraleliziranih iger v odvisnosti od števila procesov na prvem računalniku



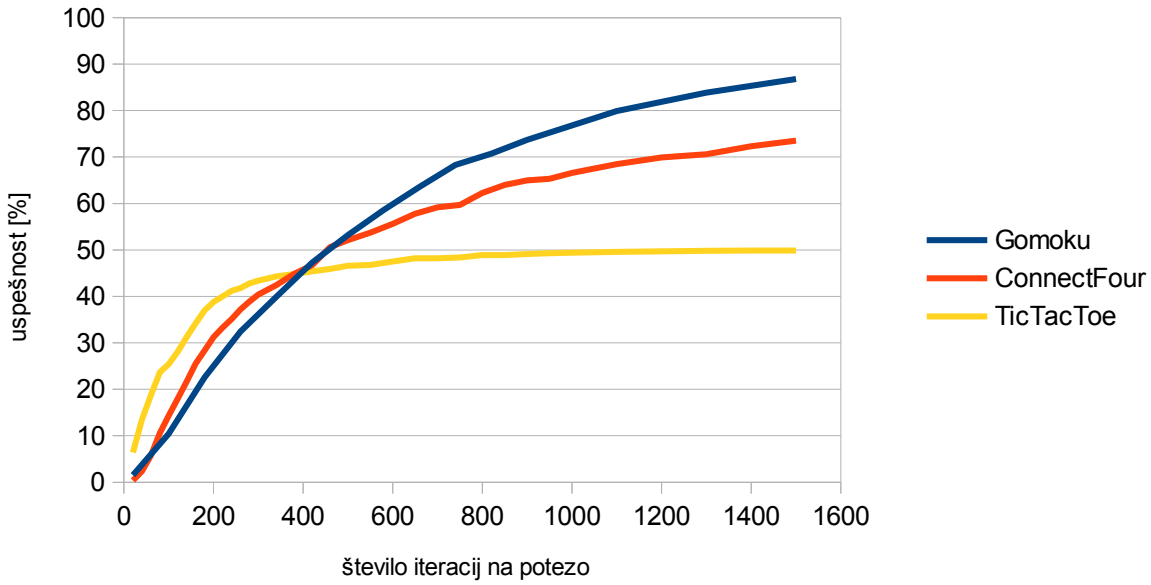
Slika 7: Čas igranja serije 10.000 paraleliziranih iger v odvisnosti od števila procesov na drugem računalniku



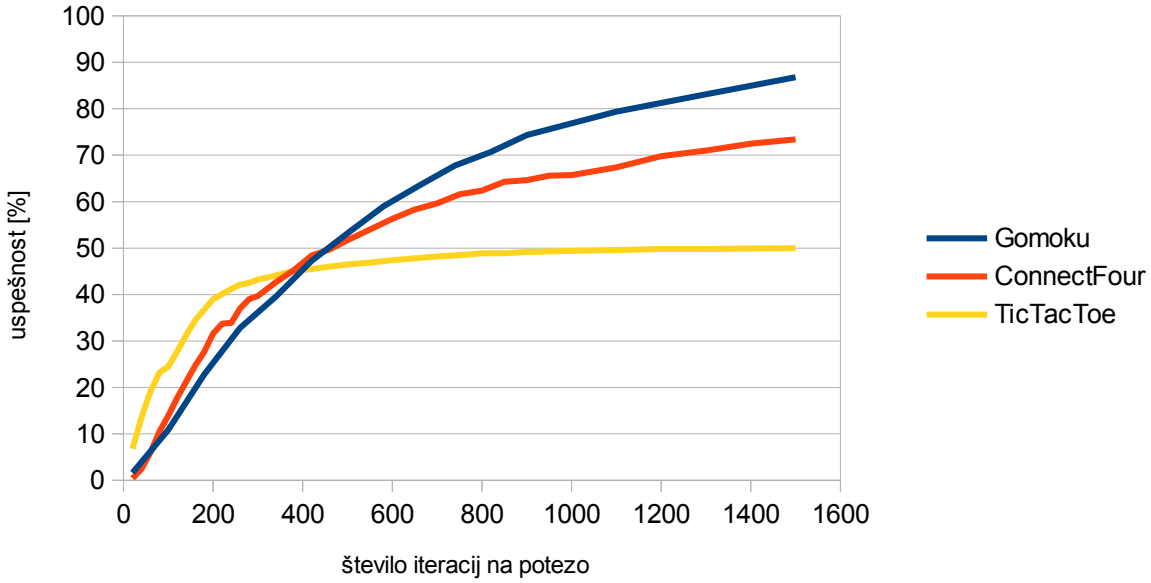
Slika 8: Pohitritev igranja serije 10.000 paraleliziranih iger v odvisnosti od števila procesov na drugem računalniku

5.1.1 POVEČEVANJE ŠTEVILA ITERACIJ NA POTEZO

Pri tej seriji meritev opazujemo, kako se s povečevanjem števila iteracij na potezo povečuje uspešnost igralca v primerjavi z nasprotnikom, ki vedno opravi konstantno število iteracij na potezo. Vse meritve so bile najprej izvedene na enem procesu, kjer je ta proces odigral vseh 10.000 iger, in nato še na štirih procesih, pri čemer je vsak proces odigral 2.500 iger. Število simulacij na iteracijo je bilo pri obeh igralcih vedno 1. Nasprotni igralec je vedno izvedel 400 iteracij na posamezno potezo.



Slika 9: Uspešnost igralca pri posamezni igri v odvisnosti od števila iteracij na potezo na prvem računalniku



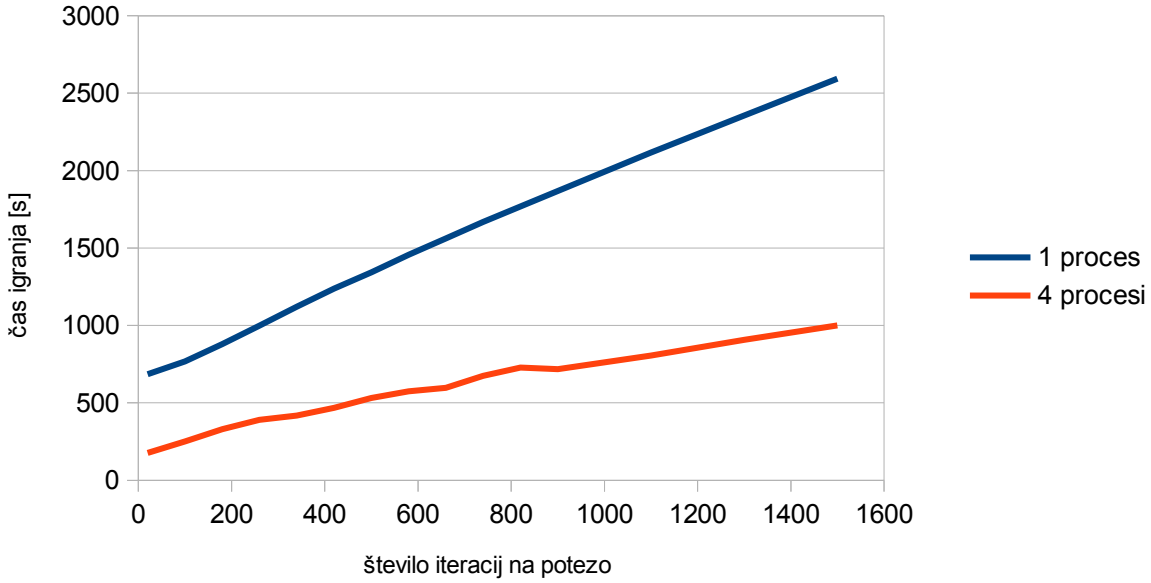
Slika 10: Uspešnost igralca pri posamezni igri v odvisnosti od števila iteracij na potezo na drugem računalniku

Na slikah 9 (serija meritev, opravljena na prvem računalniku) in 10 (serija meritev, opravljena na drugem računalniku) je prikazana uspešnost igralca v odvisnosti od števila iteracij na potezo.

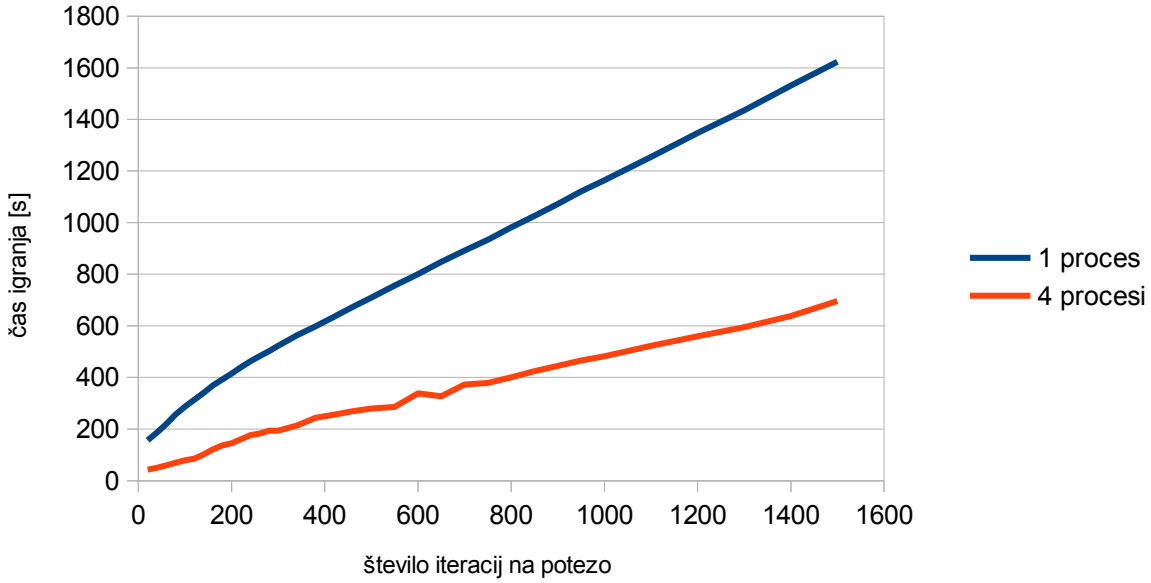
Pri igri Gomoku opazimo, da ima povečevanje števila iteracij na potezo tudi pri faktorju 3 in več (glede na število nasprotnikovih iteracij na potezo) še vedno za posledico vedno boljšo uspešnost glede na nasprotnega igralca. Isto velja za igro ConnectFour. V primeru igre TicTacToe pa opazimo, da uspešnost igralca kmalu po tem, ko presežemo dvakratno število nasprotnikovih iteracij na potezo, začne limitirati proti vrednosti 50 %. To se zgodi v obeh serijah meritev, opravljenih tako na prvem kot na drugem računalniku. Iz tega lahko sklepamo, da pri enostavnejših igrah, kot je na primer TicTacToe, tudi veliko večje število iteracij na potezo, kot jih ima nasprotnik, ne prinese boljšega rezultata, kot ga dosežemo, če je število iteracij na potezo dvakratnik nasprotnikovega števila iteracij na potezo.

Pri izvajanju enake serije meritev na štirih procesih opazimo, da je na prvem računalniku za igro Gomoku faktor pohitritve na začetku skoraj 4, nato pa pade na okoli 2,5. Pri drugih dveh igrah je padec viden nekoliko pozneje (pri meritvah, kjer je število iteracij na potezo večje glede na igro Gomoku). To lahko pripišemo dejstvu, da se pri časovno daljših igrah zgodi več zamenjav procesov, kar podaljša izvajanje. Pri drugem računalniku se faktor pohitritve giblje med 2,27 in 2,52. Takšno spremembo obnašanja lahko pripišemo različnim operacijskim

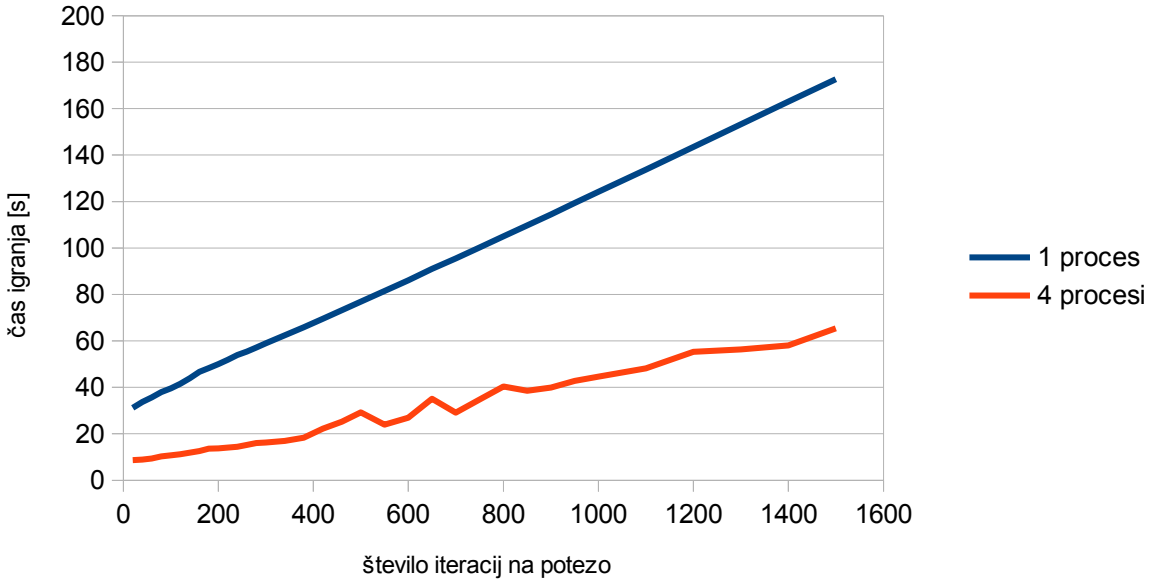
sistemom. Na slikah 11–13 so predstavljeni časi igranja glede na število iteracij na potezo za meritve, izvedene na prvem računalniku. Na slikah 14–16 so predstavljeni časi igranja glede na število iteracij na potezo za meritve, izvedene na drugem računalniku.



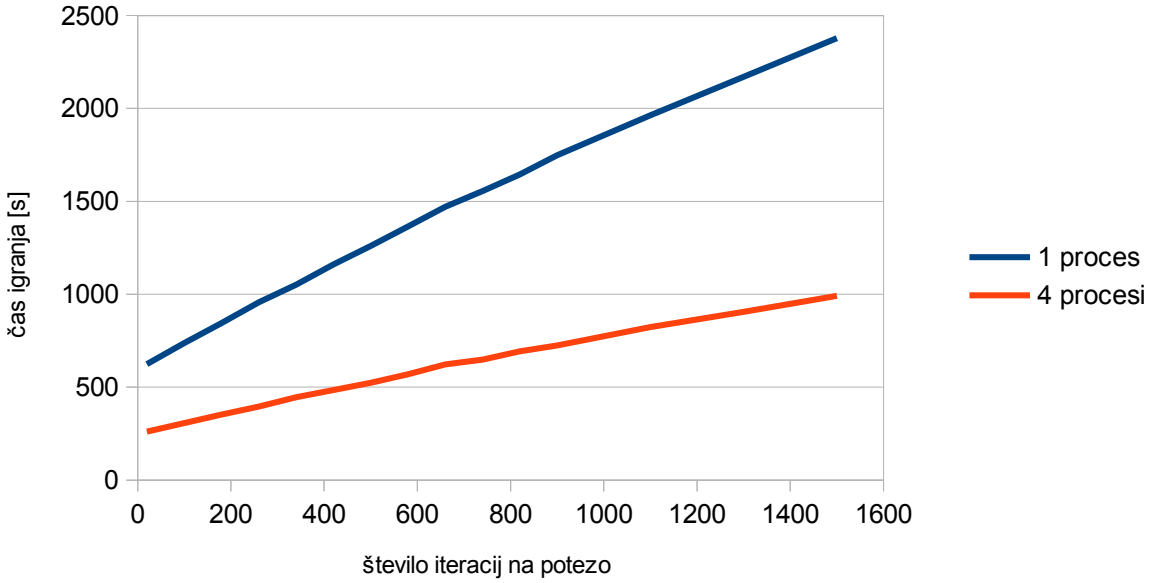
Slika 11: Čas igranja serije 10.000 paraleliziranih iger Gomoku v odvisnosti od števila iteracij na potezo na prvem računalniku



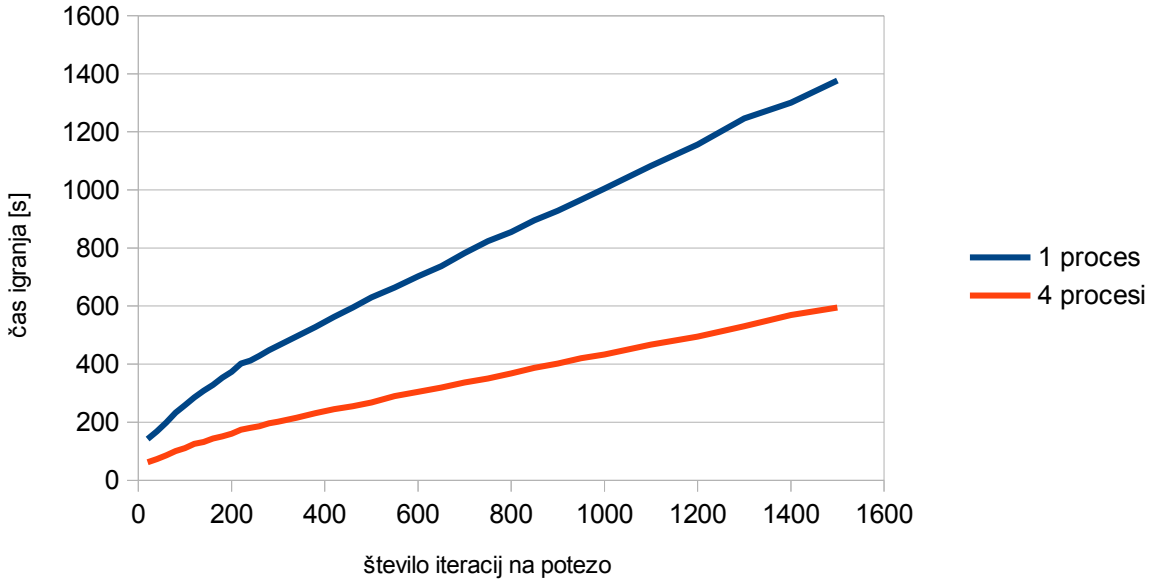
Slika 12: Čas igranja serije 10.000 paraleliziranih iger ConnectFour v odvisnosti od števila iteracij na potezo na prvem računalniku



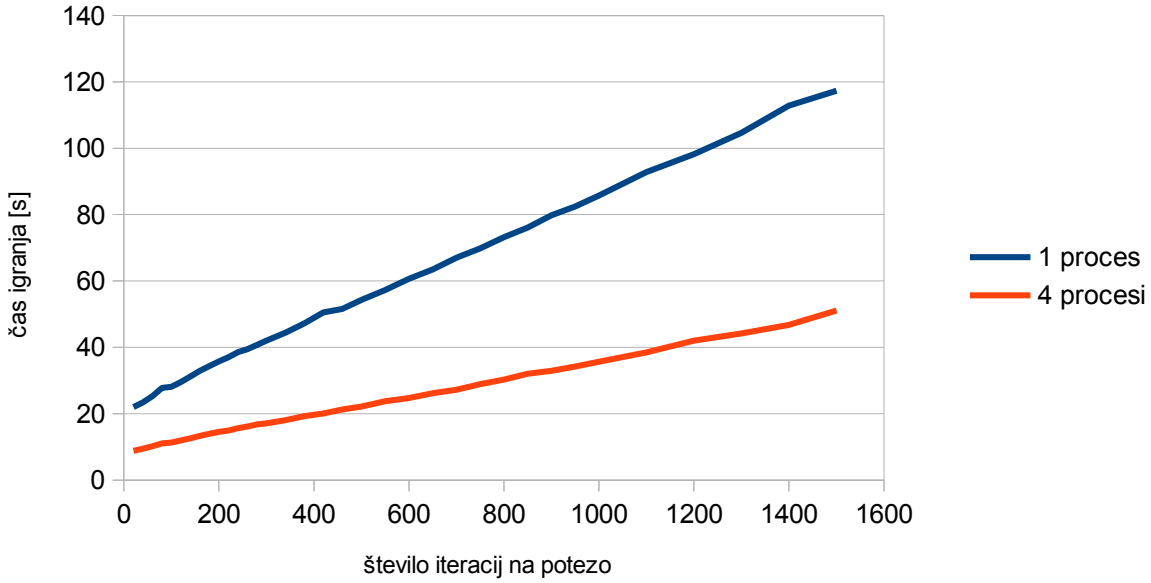
Slika 13: Čas igranja serije 10.000 paraleliziranih iger TicTacToe v odvisnosti od števila iteracij na potezo na prvem računalniku



Slika 14: Čas igranja serije 10.000 paraleliziranih iger Gomoku v odvisnosti od števila iteracij na potezo na drugem računalniku



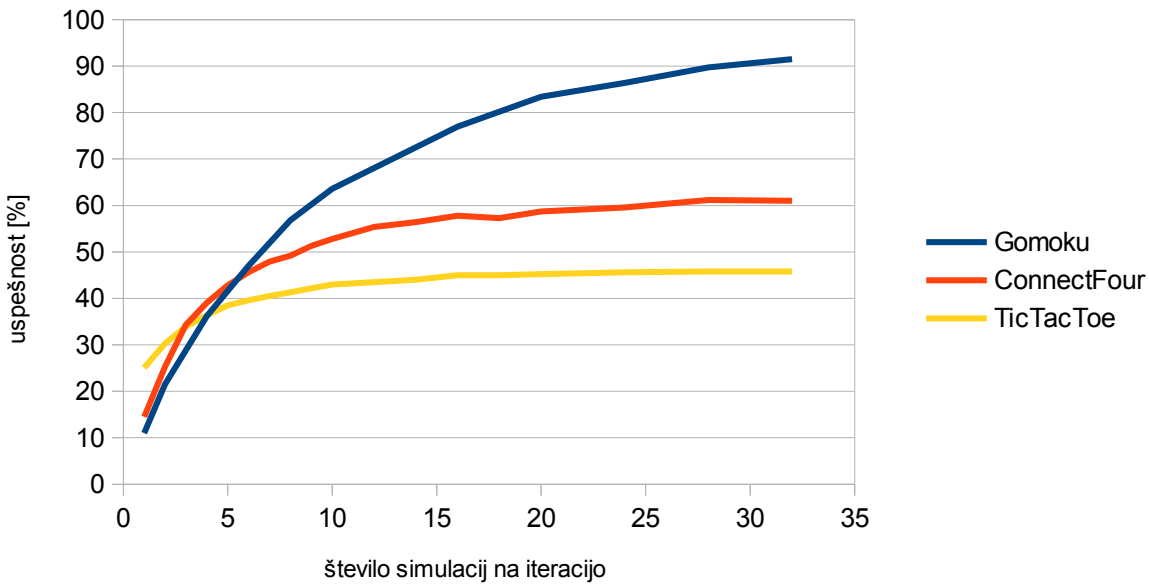
Slika 15: Čas igranja serije 10.000 paraleliziranih iger ConnectFour v odvisnosti od števila iteracij na potezo na drugem računalniku



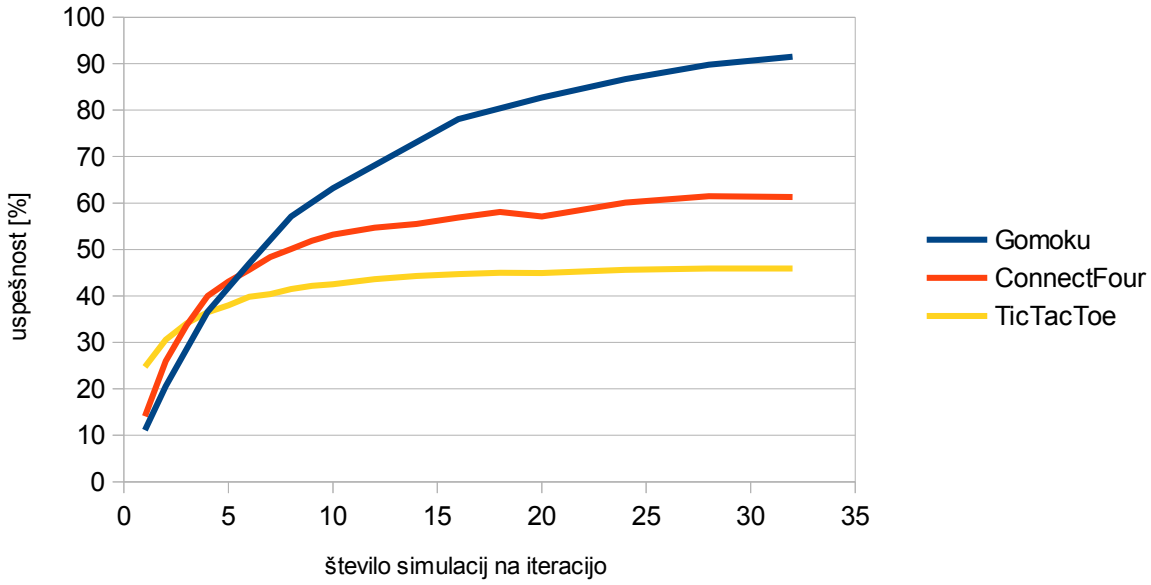
Slika 16: Čas igranja serije 10.000 paraleliziranih iger TicTacToe v odvisnosti od števila iteracij na potezo na drugem računalniku

5.1.2 POVEČEVANJE ŠTEVILA SIMULACIJ NA ITERACIJO

Tu opazujemo, kako se povečuje uspešnost igralca, če povečujemo število simulacij na iteracijo. V tem primeru opazovani igralec vedno izvede 100 iteracij na potezo, pri čemer v vsaki meritvi poveča število simulacij na iteracijo. Nasprotni igralec vedno izvede 400 iteracij na potezo, pri vsaki iteraciji pa vedno izvede le 1 simulacijo. Tudi tu so bile meritve najprej izvedene na enem procesu, kjer je ta proces odigral vseh 10.000 iger, in nato še na štirih procesih, pri čemer je vsak proces odigral 2.500 iger.



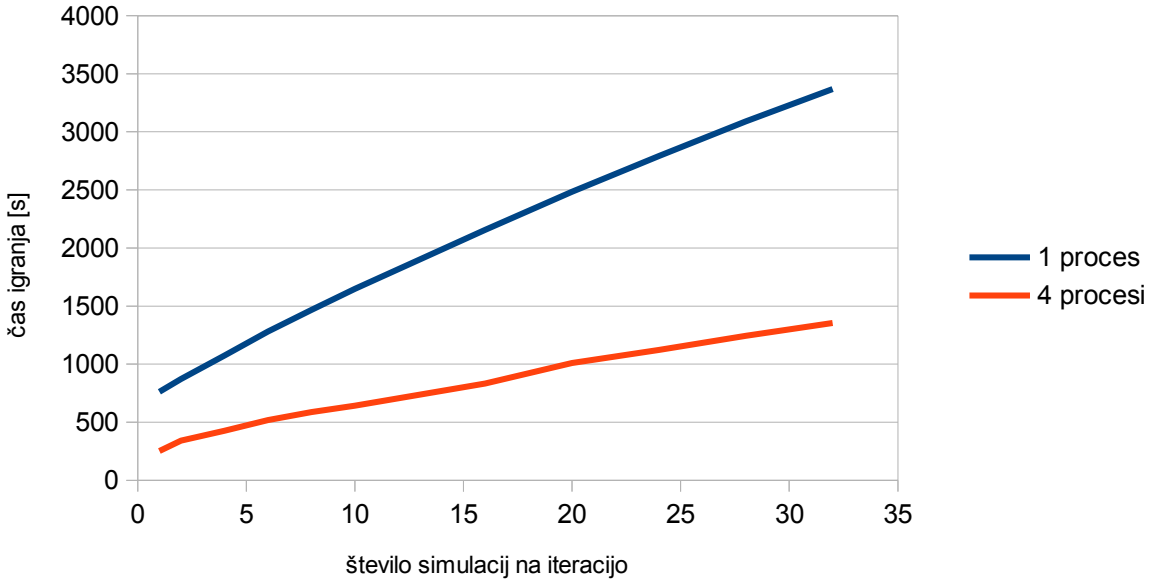
Slika 17: Uspešnost igralca pri posamezni igri v odvisnosti od števila simulacij na iteracijo na prvem računalniku



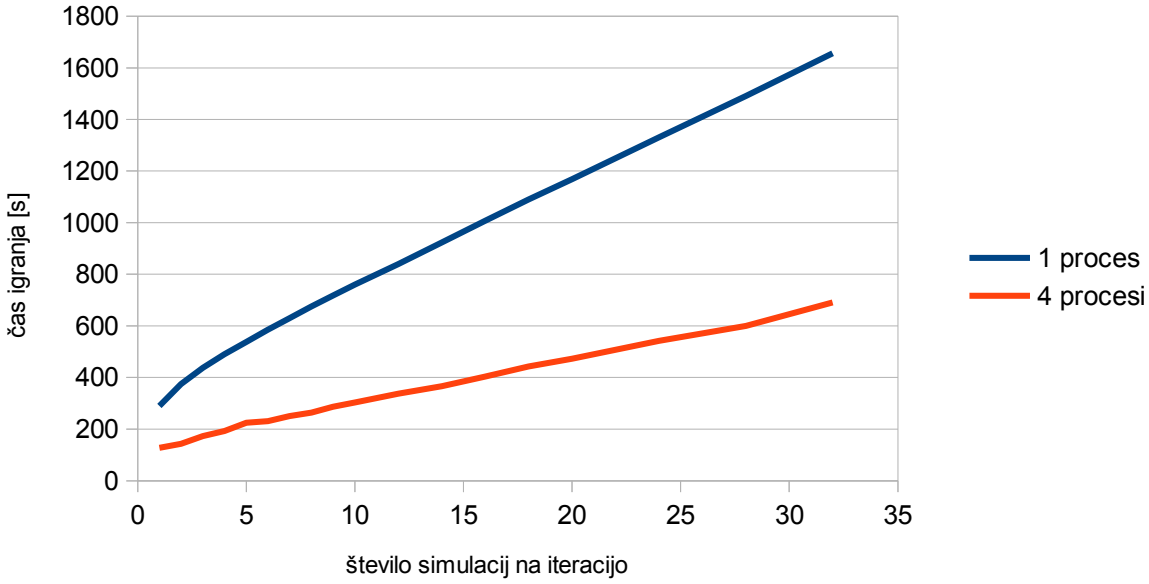
Slika 18: Uspešnost igralca pri posamezni igri v odvisnosti od števila simulacij na iteracijo na drugem računalniku

Tako kot pri prejšnji seriji meritev, ko smo povečevali število iteracij na potezo, je tudi tukaj (na slikah 17 in 18) vidno, da pri igri TicTacToe dokaj hitro pride do limitiranja uspešnosti, vendar je tu limita še malo manjša (okoli 46 %). Opazimo, da tudi igra ConnectFour pri vrednosti 28 simulacij na iteracijo začne limitirati proti uspešnosti 61 %, vendar bi morali za dokončno potrditev te opazke pognati še nekaj meritev pri višjem številu simulacij na iteracijo. Pri igri Gomoku glede na zadnji dve meritvi (še) ne moremo reči, da bi uspešnost limitirala proti določeni vrednosti.

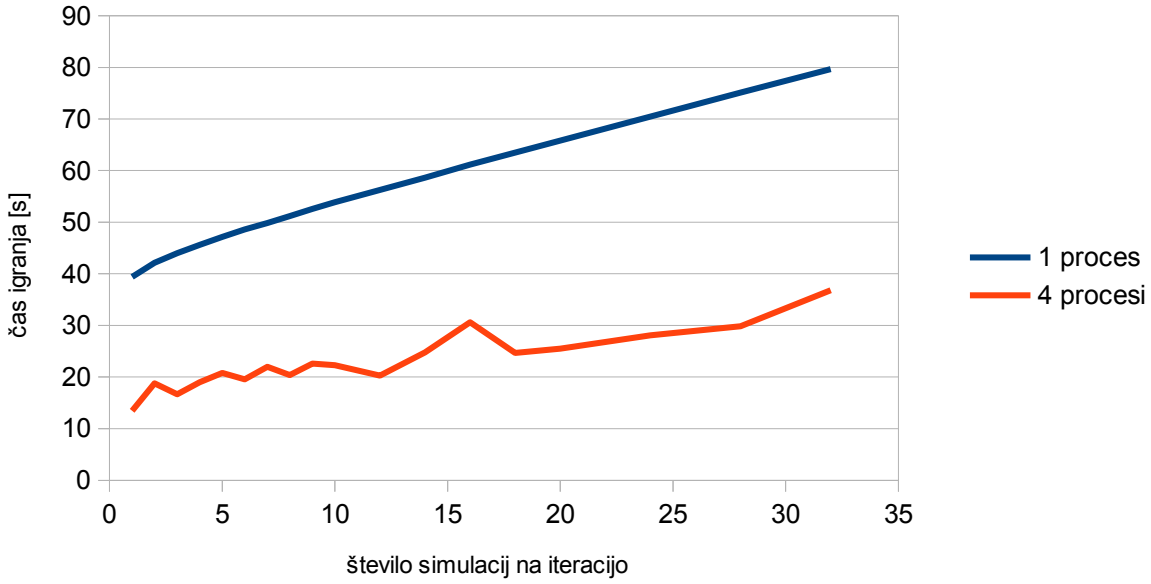
Pri primerjavi pohitritve izvajanja programa na štirih procesih v primerjavi z enim procesom opazimo, da je pohitritev na prvem računalniku na začetku približno 3 pri igrah Gomoku in TicTacToe, nato pa pade in se ustavi pri približno 2,5 pri igri Gomoku, pri 2,4 pri igri ConnectFour in pri 2,2 pri igri TicTacToe. Na drugem računalniku se pohitritve pri vseh meritvah gibljejo okoli 2,3. Rezultati serije meritev so prikazani na slikah 19–21 za meritve, izvedene na prvem računalniku, in na slikah 22–24 za meritve, izvedene na drugem računalniku.



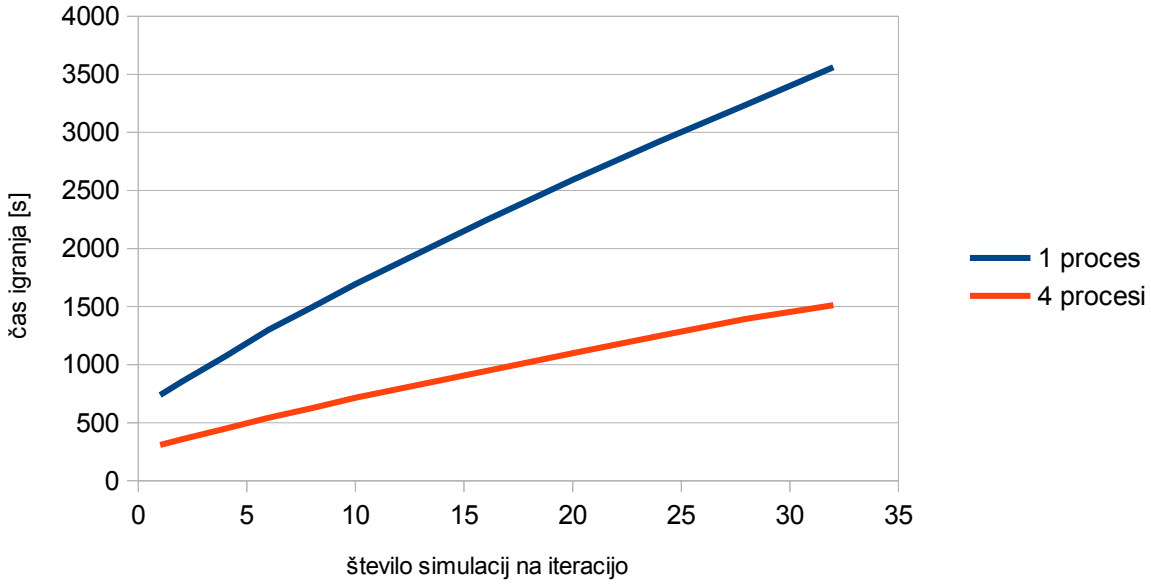
Slika 19: Čas igranja serije 10.000 paraleliziranih iger Gomoku v odvisnosti od števila simulacij na iteracijo na prvem računalniku



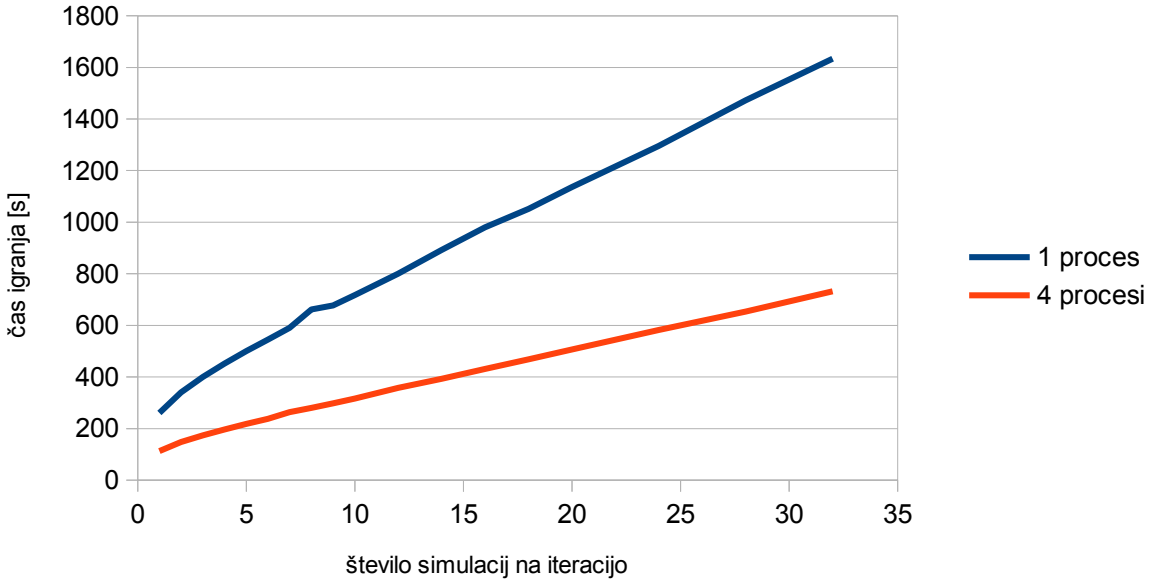
Slika 20: Čas igranja serije 10.000 paraleliziranih iger ConnectFour v odvisnosti od števila simulacij na iteracijo na prvem računalniku



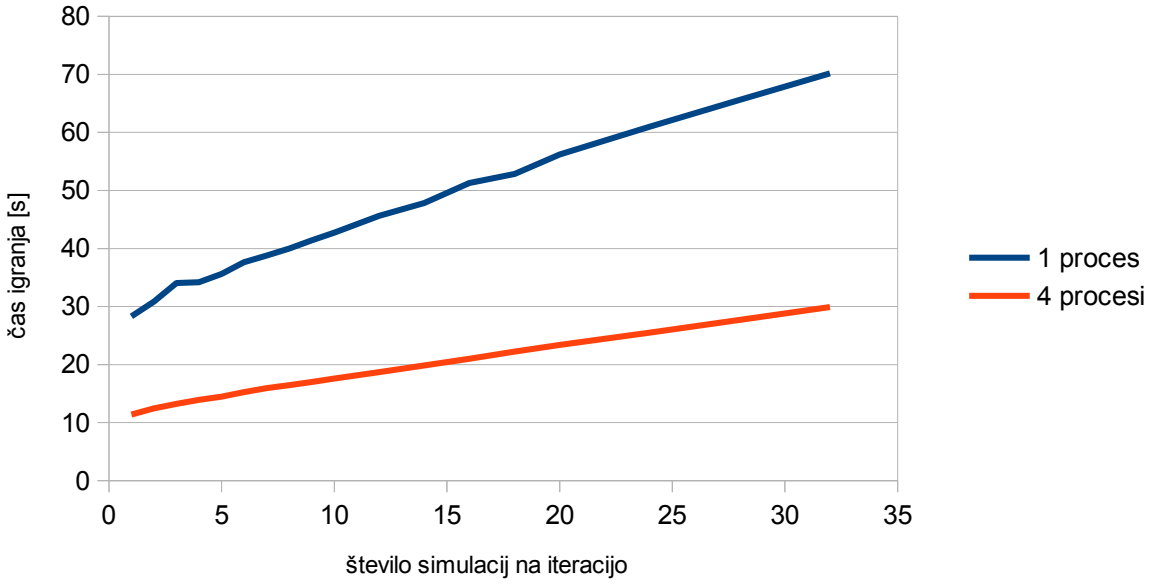
Slika 21: Čas igranja serije 10.000 paraleliziranih iger TicTacToe v odvisnosti od števila simulacij na iteracijo na prvem računalniku



Slika 22: Čas igranja serije 10.000 paraleliziranih iger Gomoku v odvisnosti od števila simulacij na iteracijo na drugem računalniku



Slika 23: Čas igranja serije 10.000 paraleliziranih iger ConnectFour v odvisnosti od števila simulacij na iteracijo na drugem računalniku

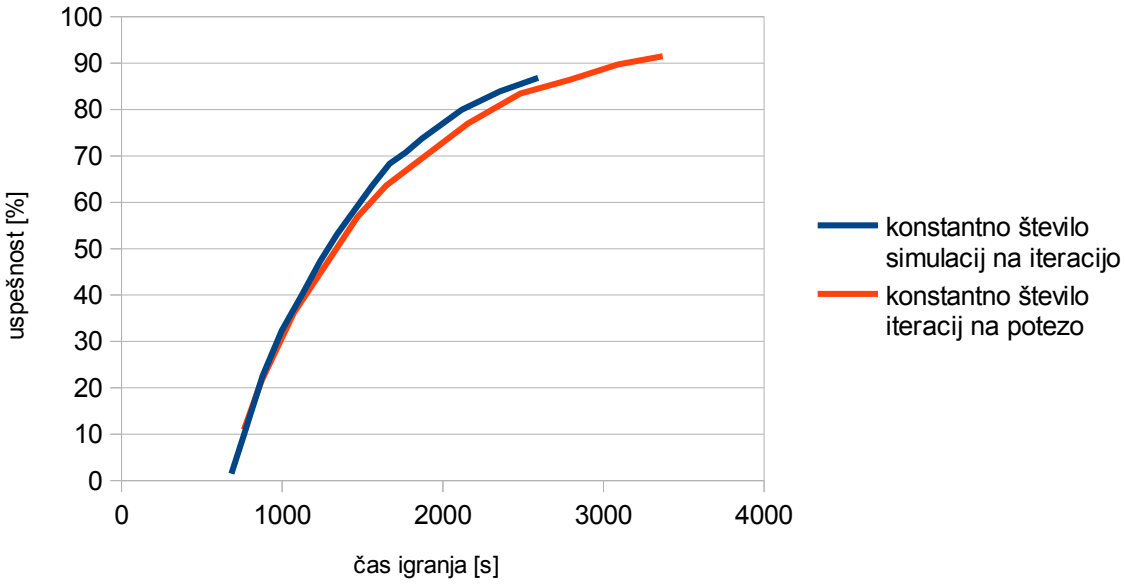


Slika 24: Čas igranja serije 10.000 paraleliziranih iger TicTacToe v odvisnosti od števila simulacij na iteracijo na drugem računalniku

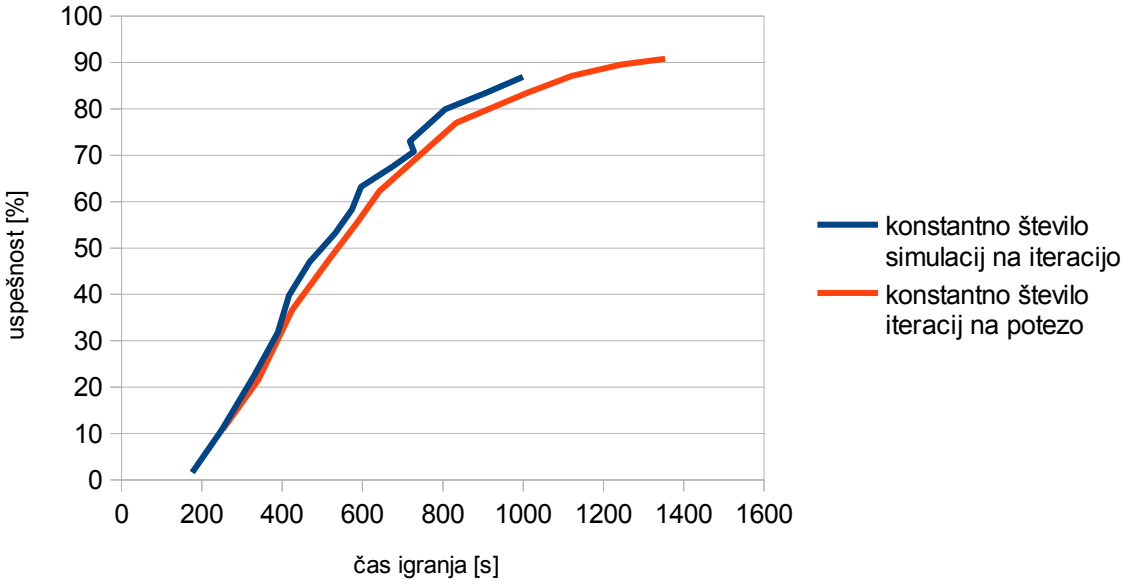
Sliki 25 in 26 (za seriji meritev, izvedene na prvem računalniku) ter sliki 27 in 28 (za seriji meritev, izvedene na drugem računalniku) predstavljata uspešnost igralca v odvisnosti od časa igranja pri igri Gomoku. Pri tej igri vidimo, da s povečevanjem števila iteracij na potezo na prvem računalniku v istem času dosežemo malenkostno boljši rezultat kot s povečevanjem števila simulacij na iteracijo. Na drugem računalniku je razlika med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo že bolj očitna.

Pri igri ConnectFour (sliki 29 in 30 za seriji meritev, izvedene na prvem računalniku) ter sliki 31 in 32 (za seriji meritev, izvedene na drugem računalniku) sta povečevanje števila iteracij na potezo in povečevanje števila simulacij na iteracijo nekako izenačena do uspešnosti 55 % na prvem računalniku in do uspešnosti 50 % na drugem računalniku. Nato nam povečevanje števila iteracij na potezo v istem času prinese boljše rezultate (večji odstotek uspešnosti). Na sliki 30 sicer vidimo, da so začetne meritve, kjer smo povečevali število simulacij na iteracijo, počasnejše. Možno je, da se je prav v času meritev izvajal še kakšen dodaten proces, ki je odvzel nekaj procesorskega časa (ta meritev je tekla na štirih procesih, kar pomeni, da so bila vsa fizična jedra procesorja zasedena), zato lahko sklepamo, da to ni nek konstanten pojav, kar dokazuje tudi slika 32, kjer takšnih odstopanj ni videti.

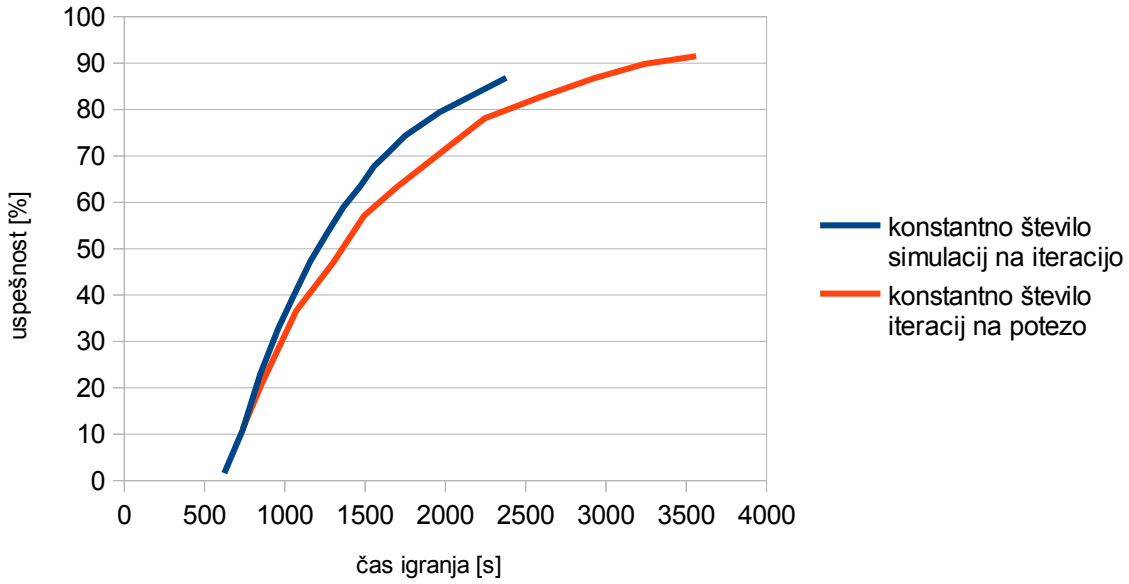
Pri igri TicTacToe vidimo (sliki 33 in 34 za seriji meritev, izvedene na prvem računalniku) ter sliki 35 in 36 (za seriji meritev, izvedene na drugem računalniku), da sta povečevanje števila iteracij na potezo in povečevanje števila simulacij na iteracijo nekako izenačena do uspešnosti 45 %, kar se lepše vidi na slikah 35 in 36. Pri sliki 34 opazimo zelo velika nihanja, kar je opazno tudi na sliki 21. Tako kot pri igri ConnectFour (slika 30) lahko tudi tu predpostavimo, da je posledica takšnih anomalij zasedba procesorskega časa s strani kakšnega časovno zahtevnega procesa (najverjetneje kakšnega sistemskega opravila).



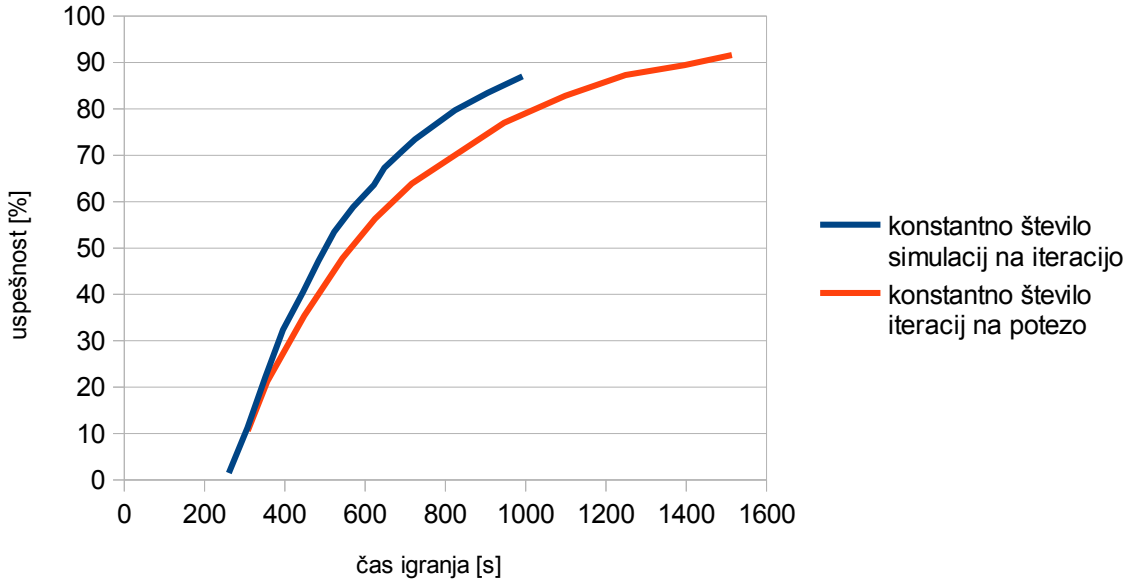
Slika 25: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger Gomoku na enem procesu na prvem računalniku



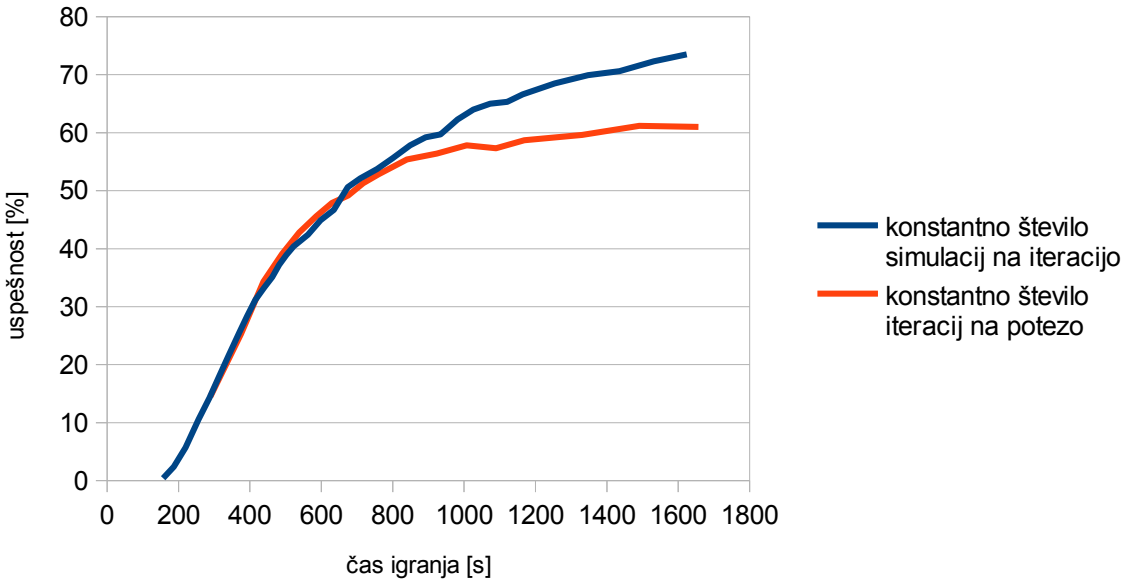
Slika 26: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger Gomoku na štirih procesih na prvem računalniku



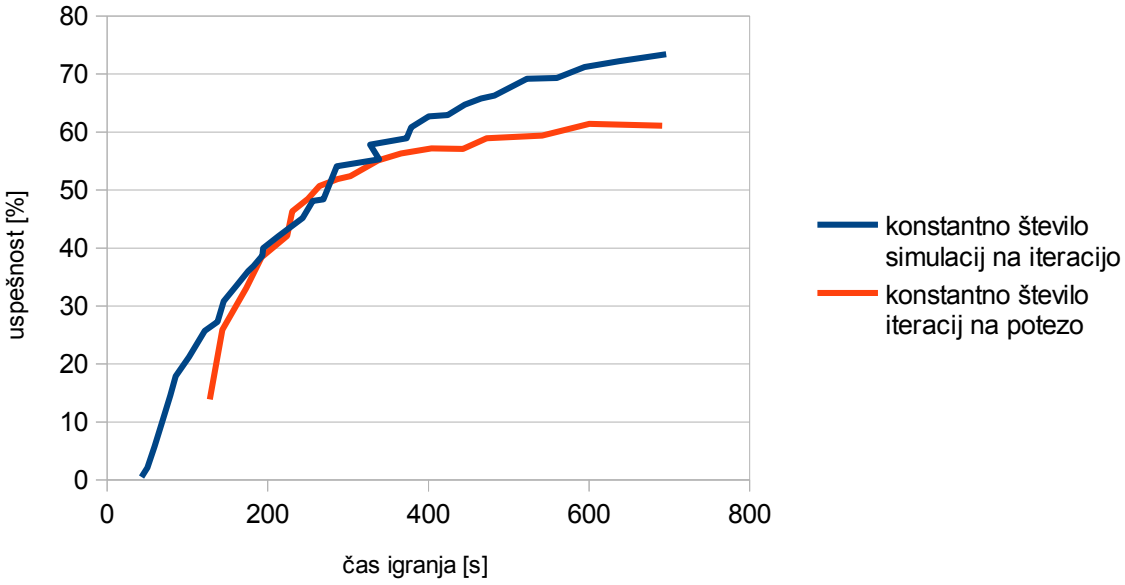
Slika 27: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger Gomoku na enem procesu na drugem računalniku



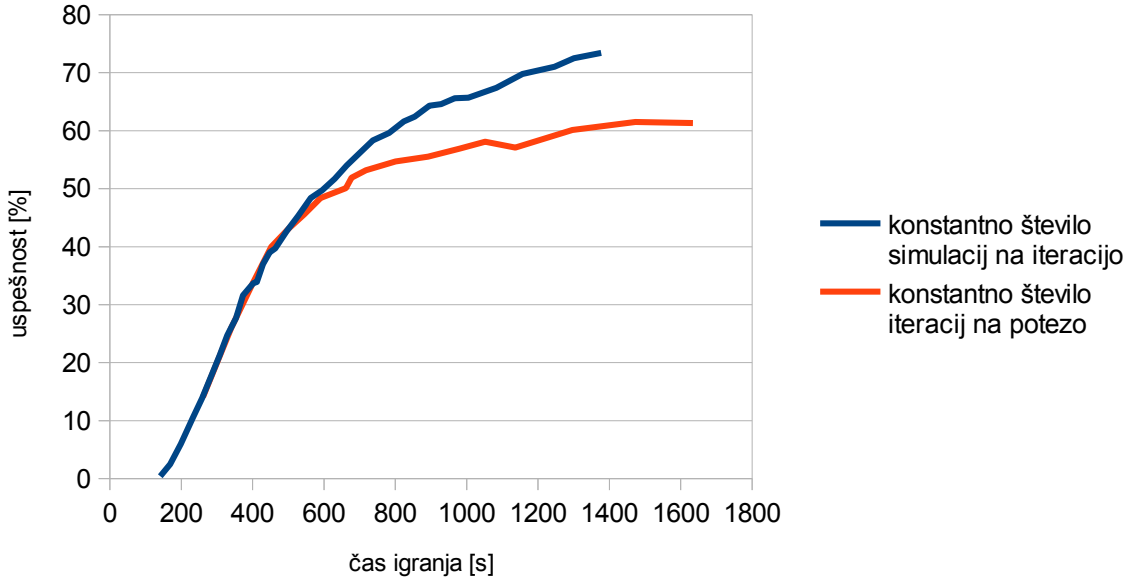
Slika 28: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger Gomoku na štirih procesih na drugem računalniku



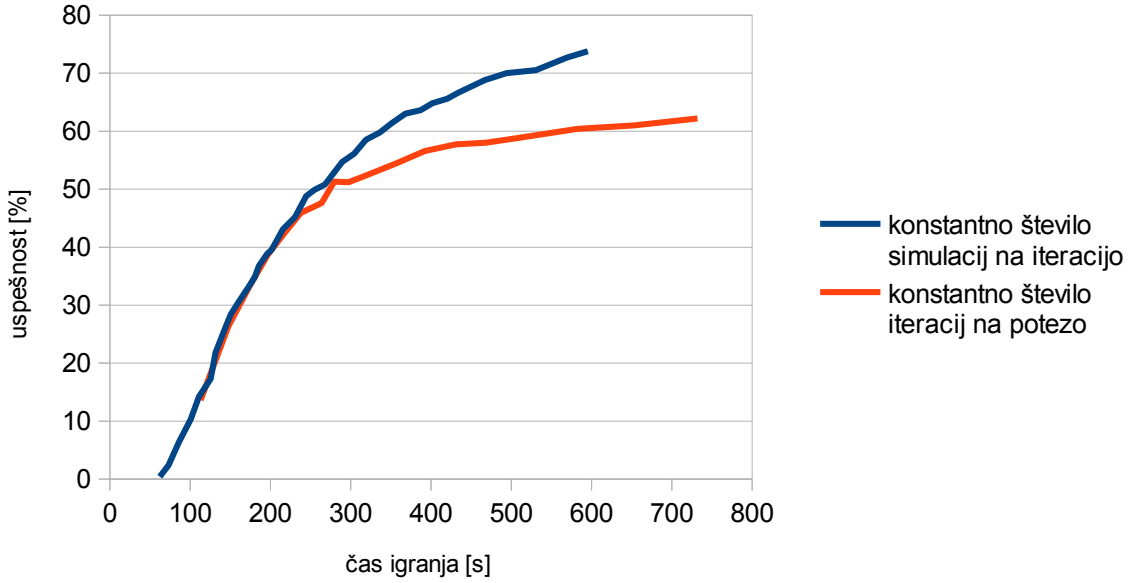
Slika 29: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger ConnectFour na enem procesu na prvem računalniku



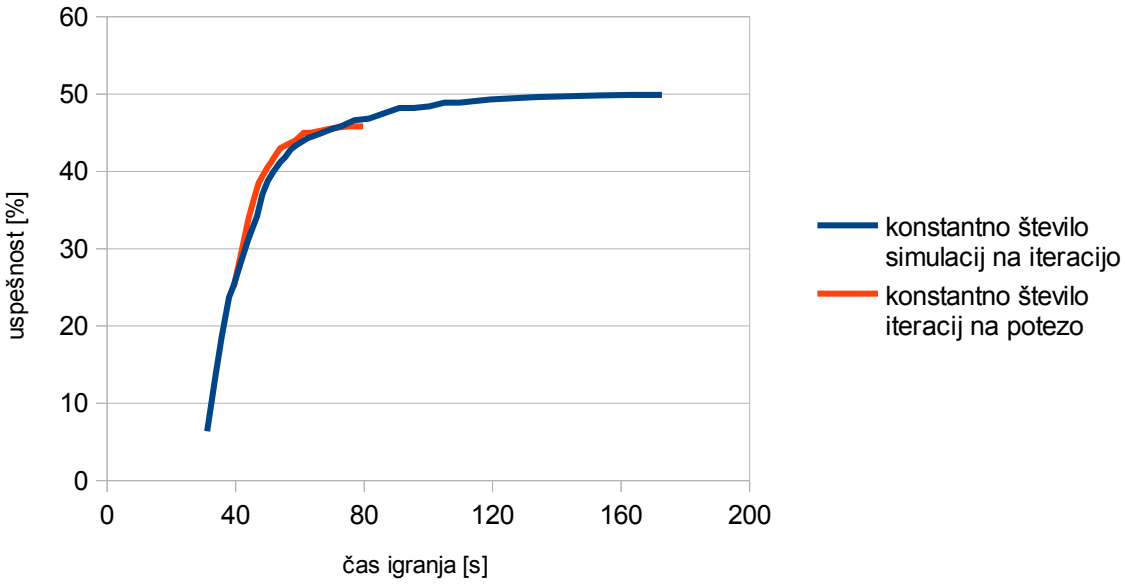
Slika 30: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger ConnectFour na štirih procesih na prvem računalniku



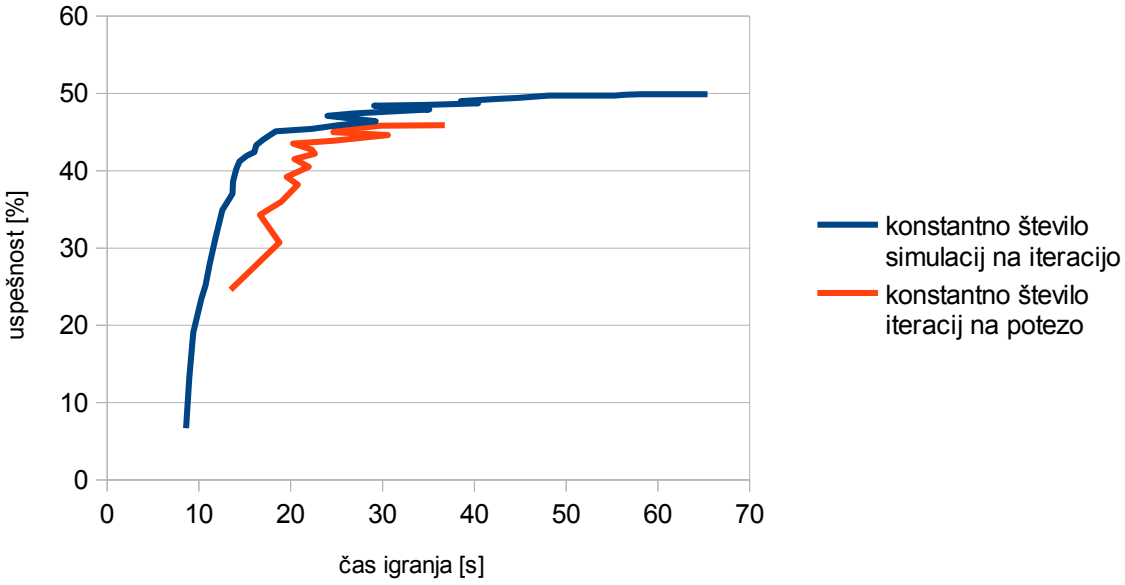
Slika 31: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger ConnectFour na enem procesu na drugem računalniku



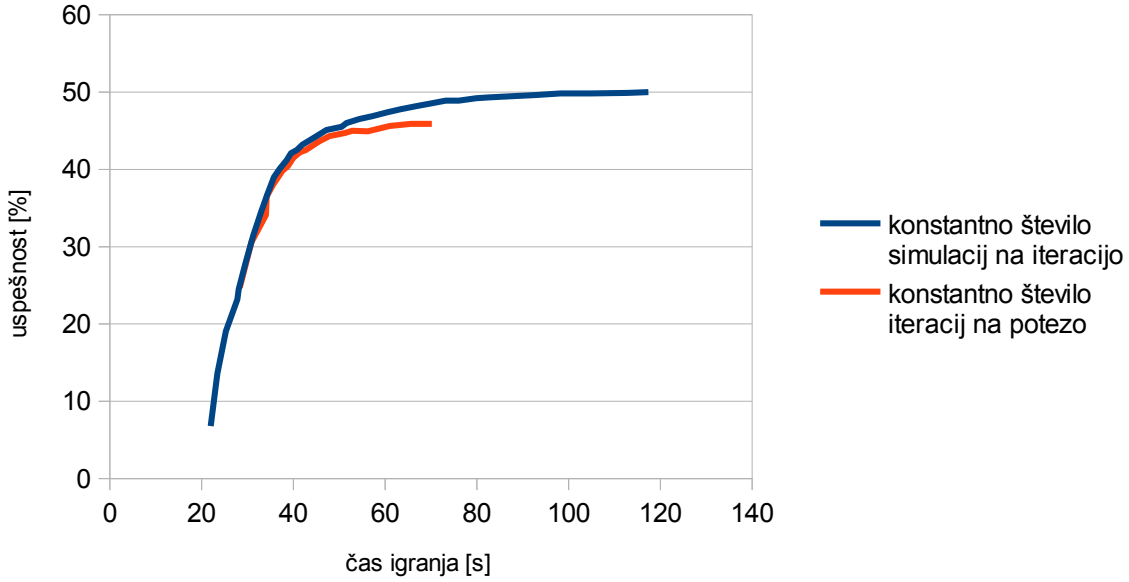
Slika 32: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger ConnectFour na štirih procesih na drugem računalniku



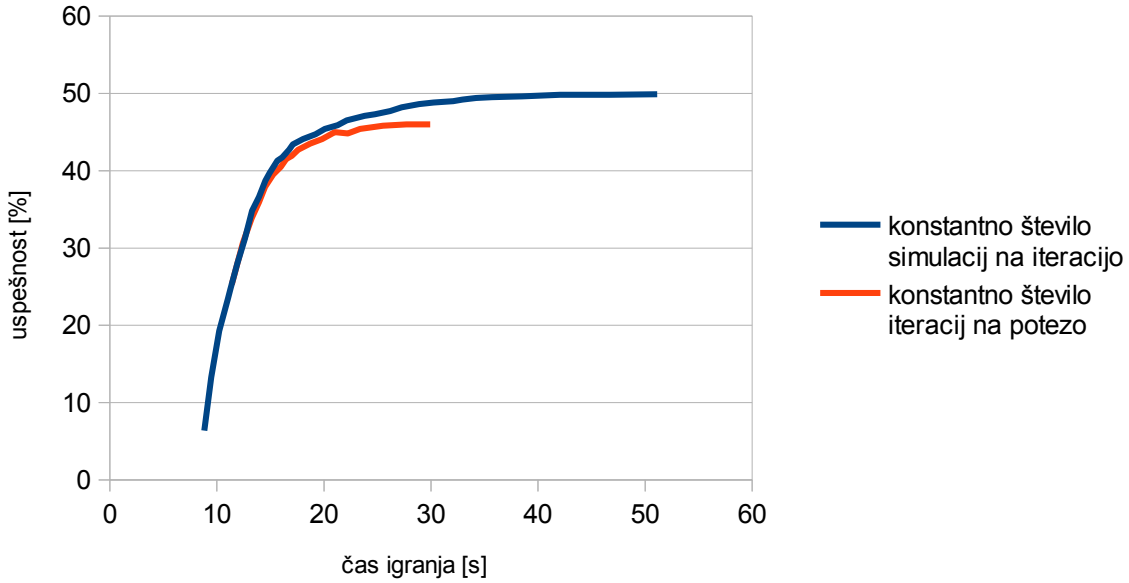
Slika 33: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger TicTacToe na enem procesu na prvem računalniku



Slika 34: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger TicTacToe na štirih procesih na prvem računalniku



Slika 35: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger TicTacToe na enem procesu na drugem računalniku



Slika 36: Primerjava uspešnosti igralca med povečevanjem števila iteracij na potezo in povečevanjem števila simulacij na iteracijo glede na čas igranja serije 10.000 paraleliziranih iger TicTacToe na štirih procesih na drugem računalniku

5.2 PARALELIZACIJA SIMULACIJ

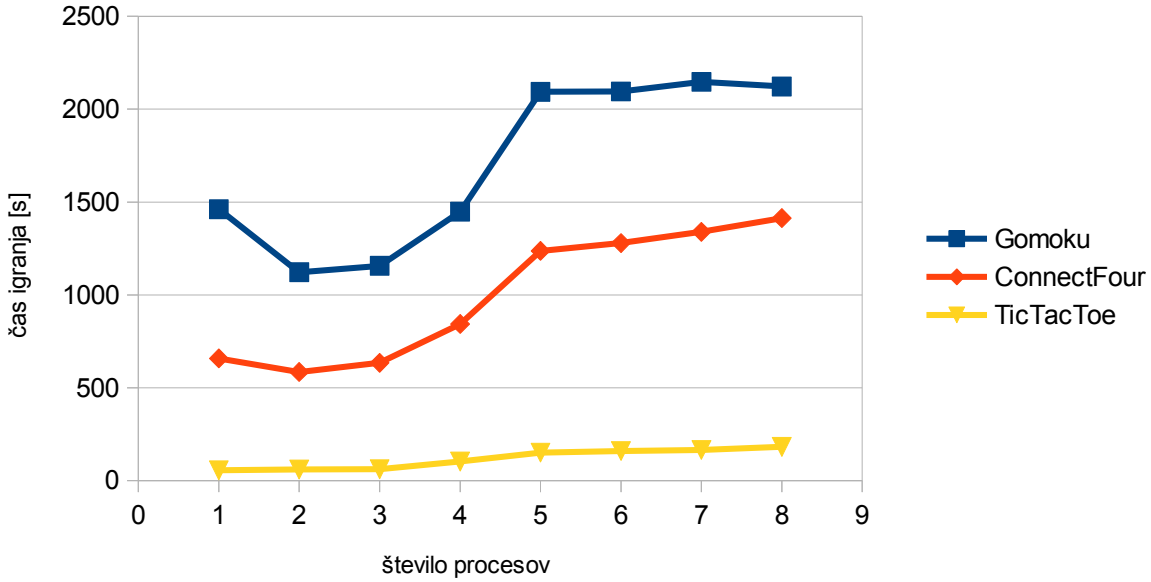
V sklop teh meritev sodi serija meritev, pri kateri povečujemo število procesov od 1 do 8 in izvajamo 8 simulacij na iteracijo. V prvi meritvi en proces opravi vse simulacije, v drugi se število simulacij porazdeli med dva procesa, tako da vsak opravi 4 simulacije, … Nasprotni igralec vedno izvede 1 simulacijo na iteracijo in 400 iteracij na potezo. Opazovani igralec vedno izvede 100 iteracij na potezo. Tudi tu ena meritev obsega 10.000 iger.

Ker se med posameznimi meritvami število odigranih simulacij na iteracijo in število iteracij na potezo ne spreminja, so tudi rezultati med meritvami zelo podobni. Pri igri Gomoku uspešnost opazovanega igralca v povprečju dosega 56,3 %, pri igri ConnectFour 50,1 % in pri igri TicTacToe 41,4 %.

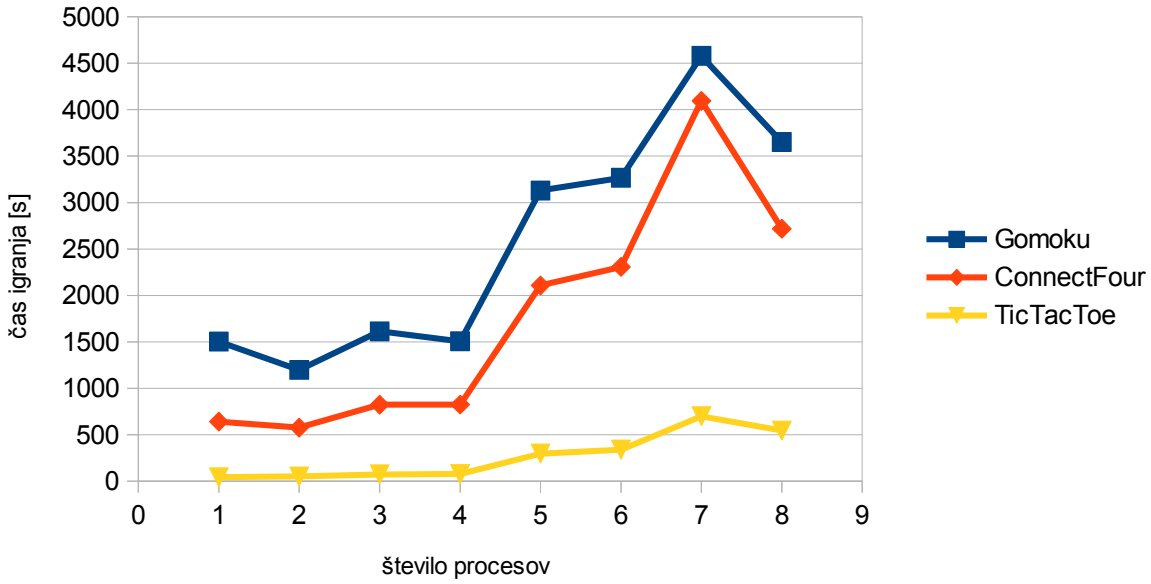
Pri igrah Gomoku in ConnectFour opazimo, da se pri prehodu z enega procesa na dva čas izvajanja programa zmanjša, vendar ne za 2 x, kot bi sprva pričakovali, pri igri TicTacToe pa se čas celo poveča. Pri tem je treba upoštevati, da je v primeru več procesov stanje igre iz glavnega procesa treba prenesti na vse druge procese, kar terja svoj čas in v določenih primerih celo upočasni izvajanje. Pri igri ConnectFour opazimo, da je razmerje med časom trajanja simulacije in časom, porabljenim za komunikacijo med procesi, bistveno slabše kot pri igri Gomoku, saj je pri štirih procesih čas igranja igre ConnectFour daljši, kot če igra teče le na enem procesu, medtem ko je čas igranja igre Gomoku na štirih procesih skoraj enak času igranja na enem procesu. Pri obeh igrah opazimo občutno podaljšanje trajanja serije iger pri prehodu s štirih na pet procesov. Tukaj se poveča število prekinitev izvajanja programa, saj operacijski sistem procesorski čas dodeljuje tudi drugim procesom. Če pa ima proces še kakšno nedokončano simulacijo, ga morajo drugi procesi čakati. Pri drugem računalniku opazimo znaten padec izvajalnega časa iger pri prehodu s sedmih na osem procesov. Če vsak proces izvede 1 simulacijo (ali nobene), mu ni treba obnavljati stanja igre na stanje pred prvo simulacijo, kar se pozna na času igranja.

Ker prenos stanja igre pred začetkom izvajanja simulacij in prenos rezultatov simulacij med procesi zahteva določen čas, je smiselno na enem procesu izvesti več simulacij. Če tako povečamo število simulacij na 16 oziroma 32, se to pozna tudi pri uspešnosti igralca. Pri 16 simulacijah na iteracijo se pri igri Gomoku uspešnost v povprečju poveča na 77,3 %, pri igri ConnectFour na 57,5 % in pri igri TicTacToe na 44,7 %. Pri 32 simulacijah na iteracijo so ti odstotki še višji: 91,5 %, 61,4 % in 46,0 %. Časi izvajanja programa se sicer povečajo, vendar je (predvsem pri zahtevnejših igrah) uspešnost igralca občutno višja.

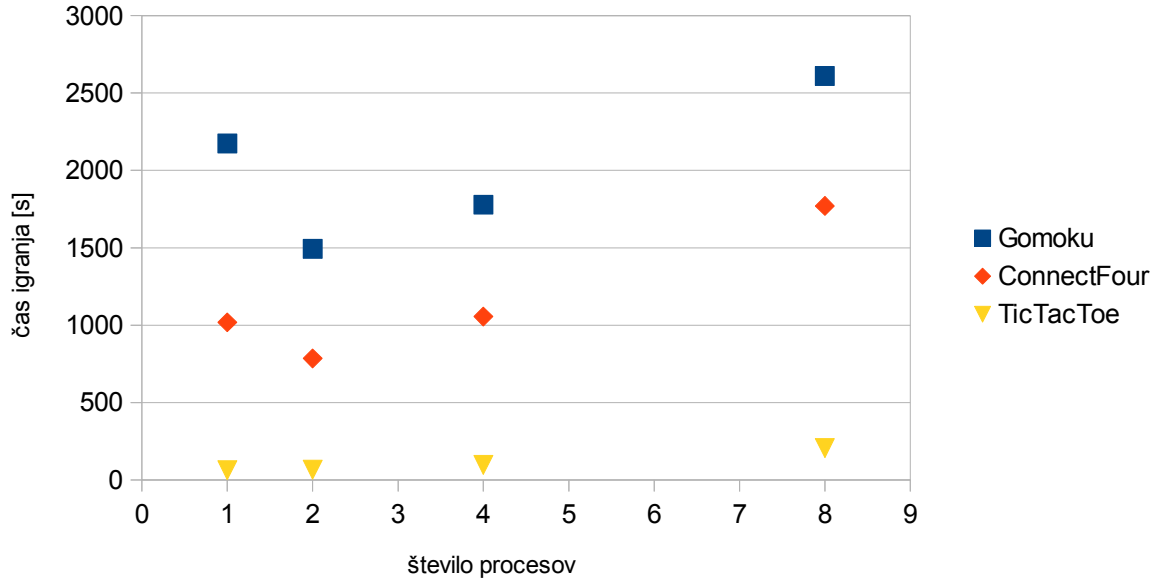
Rezultati paralelizacije 8 simulacij so prikazani na slikah 37 in 38, paralelizacije 16 simulacij na slikah 39 in 40 ter paralelizacije 32 simulacij na slikah 41 in 42.



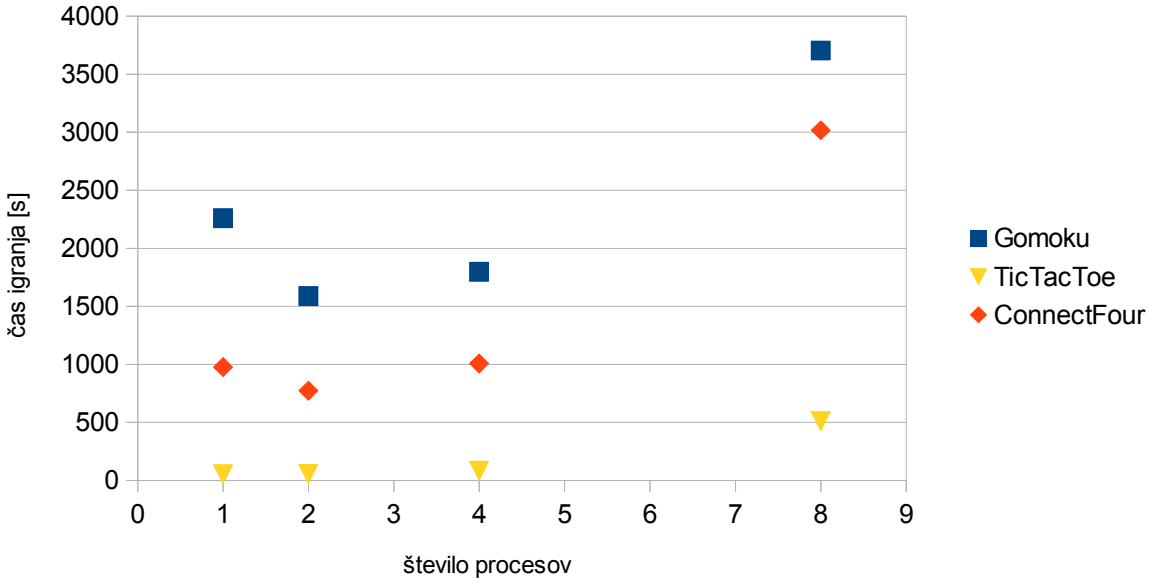
Slika 37: Čas igranja serije 10.000 iger s paraleliziranimi 8 simulacijami na iteracijo v odvisnosti od števila procesov na prvem računalniku



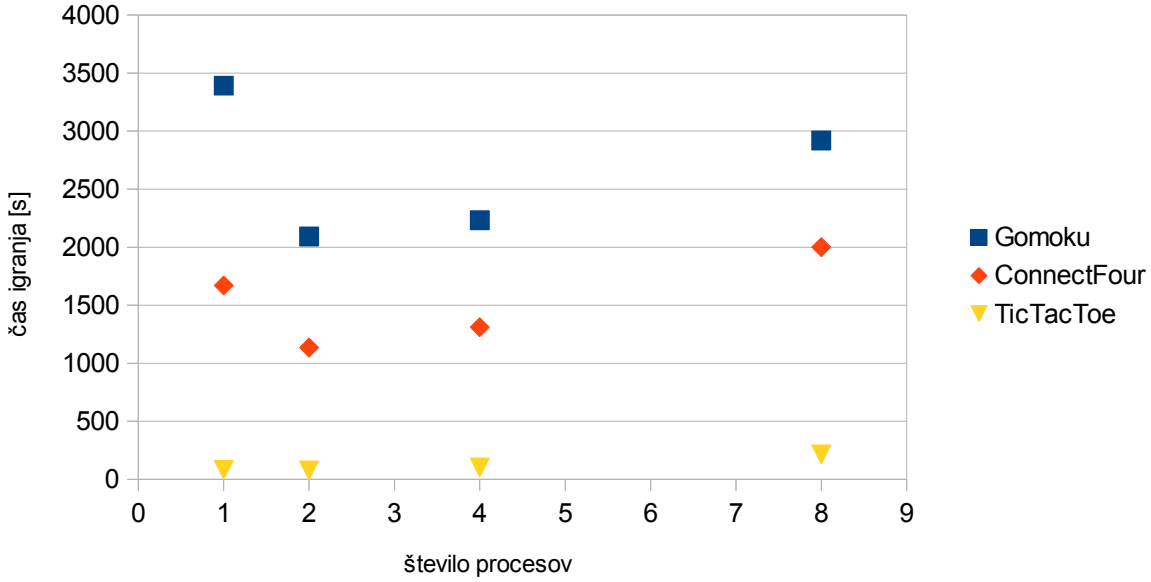
Slika 38: Čas igranja serije 10.000 iger s paraleliziranimi 8 simulacijami na iteracijo v odvisnosti od števila procesov na drugem računalniku



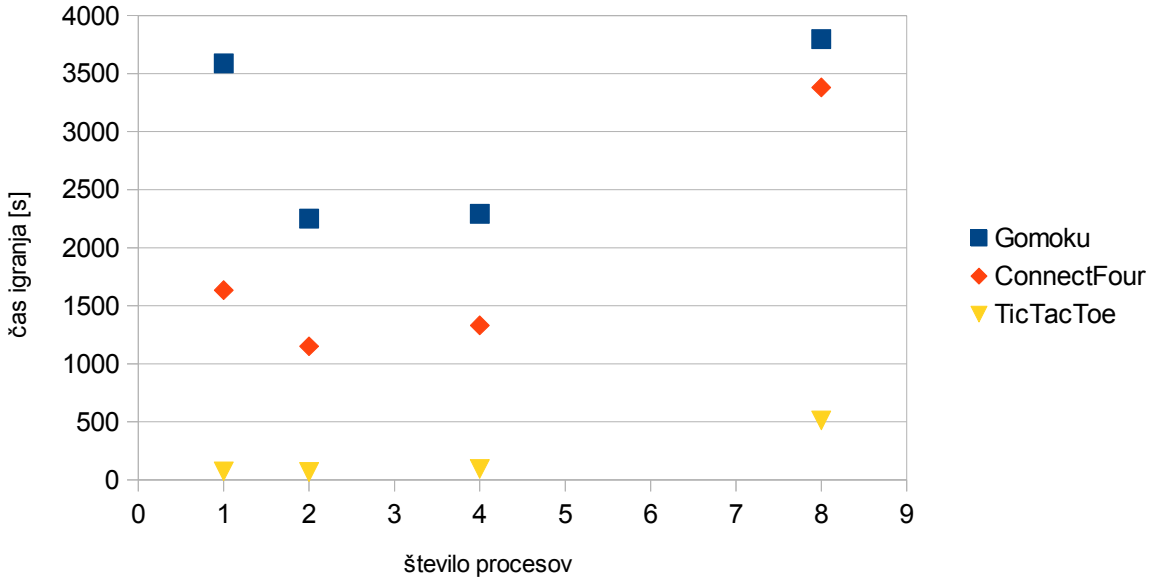
Slika 39: Čas igranja serije 10.000 iger s paraleliziranimi 16 simulacijami na iteracijo v odvisnosti od števila procesov na prvem računalniku



Slika 40: Čas igranja serije 10.000 iger s paraleliziranimi 16 simulacijami na iteracijo v odvisnosti od števila procesov na drugem računalniku



Slika 41: Čas igranja serije 10.000 iger s paraleliziranimi 32 simulacijami na iteracijo v odvisnosti od števila procesov na prvem računalniku



Slika 42: Čas igranja serije 10.000 iger s paraleliziranimi 32 simulacijami na iteracijo v odvisnosti od števila procesov na drugem računalniku

6 KONČNE UGOTOVITVE

Cilj vsakega igralca je v čim krajšem času doseči čim večjo uspešnost. Obstajata dve možnosti za izboljšanje uspešnosti: povečevanje števila iteracij na potezo in povečevanje števila simulacij na iteracijo. Čeprav se povečevanje števila iteracij na potezo obnese nekoliko boljše kot povečevanje števila simulacij na iteracijo, je simulacije mnogo lažje paralelizirati. Če pri igri Gomoku vzamemo najboljši čas izvajanja paraleliziranih 8, 16 in 32 simulacij in na sliki 25 oz. 27 pri približno enakem času odčitamo odstotek uspešnosti igralca, opazimo, da nam paralelizacija simulacij prinese boljši rezultat. Pri igri ConnectFour v primeru paralelizacije 8 in 16 simulacij dobimo boljši rezultat, v primeru paralelizacije 32 simulacij pa je v istem času uspešnejši neparaleliziran algoritem. Pri igri TicTacToe oba algoritma v istem času dosežeta približno isti rezultat. V tabelah 2–7 so za vse tri igre na obeh računalnikih prikazani najboljši rezultati, ki jih dosežemo v približno istem času igranja, skupaj s podatkom o številu iteracij na potezo, času igranja serije 10.000 iger in številu simulacij na iteracijo ter številu procesov, na katerih je tekel paraleliziran algoritem UCT.

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	340	1	39,9	1.120
	1	100	6	47,0	1.281
	2	100	8	56,0	1.122
ConnectFour	1	380	1	44,9	598
	1	100	6	45,6	586
	2	100	8	50,1	585
TicTacToe	1	280	1	42,8	57
	1	100	12	43,5	56
	1	100	8	41,4	56

Tabela 2: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 8 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na prvem računalniku

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	500	1	53,2	1.260
	1	100	6	47,0	1.301
	2	100	8	56,5	1.198
ConnectFour	1	460	1	49,7	595
	1	100	7	48,4	590
	2	100	8	50,0	576
TicTacToe	1	300	1	43,2	42
	1	100	10	42,5	43
	1	100	8	41,4	42

Tabela 3: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 8 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na drugem računalniku

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	580	1	58,6	1.456
	1	100	8	56,8	1.467
	2	100	16	77,5	1.493
ConnectFour	1	600	1	55,6	800
	1	100	12	55,4	840
	2	100	16	57,5	785
TicTacToe	1	340	1	44,3	62
	1	100	18	45,0	63
	1	100	16	44,7	63

Tabela 4: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 16 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na prvem računalniku

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	740	1	67,8	1.556
	1	100	10	63,2	1.695
	2	100	16	77,2	1.589
ConnectFour	1	700	1	59,6	783
	1	100	12	54,7	801
	2	100	16	57,5	774
TicTacToe	1	500	1	46,5	54
	1	100	20	44,9	56
	1	100	16	44,7	54

Tabela 5: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 16 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na drugem računalniku

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	1.100	1	79,9	2.117
	1	100	16	77,0	2.156
	2	100	32	91,4	2.091
ConnectFour	1	950	1	65,3	1.121
	1	100	20	58,7	1.169
	2	100	32	61,1	1.136
TicTacToe	1	500	1	46,6	77
	1	100	32	45,8	80
	2	100	32	45,8	76

Tabela 6: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 32 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na prvem računalniku

Igra	Število procesov	Število iteracij na potezo	Število simulacij na iteracijo	Uspešnost [%]	Čas igranja [s]
Gomoku	1	1.500	1	86,8	2.379
	1	100	16	78,1	2.243
	2	100	32	91,5	2.251
ConnectFour	1	1.200	1	69,8	1.156
	1	100	20	57,1	1.136
	2	100	32	61,6	1.151
TicTacToe	1	700	1	48,2	67
	1	100	32	45,9	70
	2	100	32	46,1	67

Tabela 7: Primerjava najboljših rezultatov, ki jih dobimo pri paralelizaciji 32 simulacij, z rezultati, ki jih dobimo, če izvajamo program na enem procesu v približno enakem času na drugem računalniku

7 ZAKLJUČEK

V tej diplomski nalogi sem s pomočjo knjižnice MPI paraleliziral izvajanje kode MCTS algoritma UCT. Tako lahko v istem času odigramo več iger oz. izboljšamo uspešnost igralca, saj se v istem času lahko izvede več simulacij, kar občutno poveča igralčevo uspešnost. Odprtih je še nekaj možnosti, s katerimi bi lahko še dodatno izboljšali obstoječi program. Namesto uporabe vmesnika MPI se lahko odločimo za uporabo drugih vmesnikov (npr. OpenMP). Druga možnost je kombinacija dveh različnih vmesnikov za paralelizacijo ali pa implementacija paralelizacije iteracij. Še ena možnost paralelizacije je pristop, kjer en proces skrbi za drevo MCTS, drugi pa izvajajo simulacije. To bi bilo smiselno pri igrah, kjer so simulacije časovno potratne v primerjavi z drugimi koraki iteracije. Tudi implementacija zahtevnejših iger (tukaj je mišljena predvsem igra go) je ena izmed možnih nadgradenj obstoječega programa.

8 VIRI IN LITERATURA

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, "A Survey of Monte Carlo Tree Search Methods", *IEEE Transactions on Computational Intelligence and AI in Games*, št. 1, zv. 4, str. 1-43, 2012.

[2] (2013) Message Passing Interface (MPI). Dostopno na: https://computing.llnl.gov/tutorials/mpi/

[3] (2013) Monte-Carlo tree search. Dostopno na: http://en.wikipedia.org/wiki/Monte-Carlo_tree_search

[4] (2013) Monte Carlo Tree Search. Dostopno na: http://www.mcts.ai/index.html

[5] (2013) Message Passing Interface. Dostopno na: http://en.wikipedia.org/wiki/Message_Passing_Interface

[6] (2009) MPI: A Message-Passing Interface Standard Version 2.2. Dostopno na: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[7] (2012) MPI: A Message-Passing Interface Standard Version 3.0. Dostopno na: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf