Paralelizacija računalniškega igralca v namizni igri Reversi

Porazdeljeni sistemi 2014/15

Silvester Jakša

63110185

22. februar 2015

Paralelizacija računalniškega igralca v namizni igri Reversi

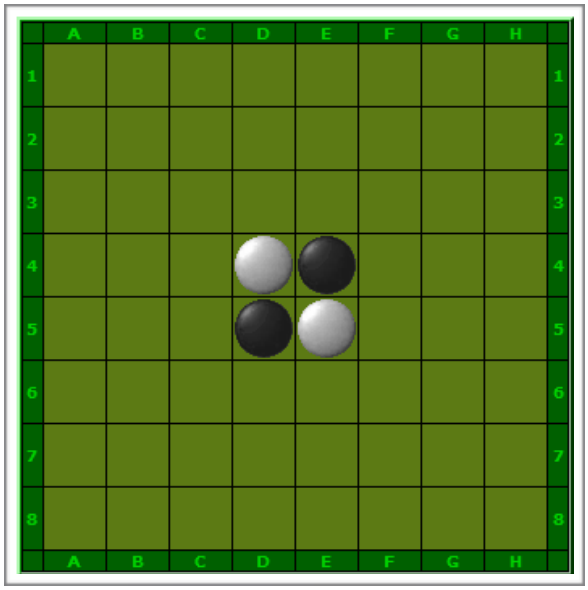
Porazdeljeni sistemi 2014/15

Reversi

Reversi ali Othello je abstraktna strateška igra za dva igralca na igralni deski z 8 x 8 polji. Igralca potrebujeta za igro črne in bele figure.

Pravila igre

Igralca se odločita, kdo bo beli in kdo črni, ta odločitev velja vso igro. Vsak igralec položi na ploščo dva žetona kot je prikazano na spodnji skici:



Črni začne. Pri vsaki potezi mora biti žeton postavljen poleg nasprotnikovega žetona: lahko vertikalno, horizontalno ali diagonalno. Položeni žeton mora ujeti enega ali več nasprotnikovih žetonov med enega ali več svojih žetonov, ki so položeni v kateri koli smeri od pravkar položenega. Ujete nasprotnikove žetone igralec obrne in jim spremeni barvo. Igralca nadaljujeta s postopkom igranja dokler ne zapolnita vseh polj, ali dokler imata možnost postavitve žetona. Včasih se igra konča že prej, v primeru da igralca nimata več možnosti za naslednjo potezo. Zmaga tisti igralec, ki ima na koncu več žetonov na igralni plošči.

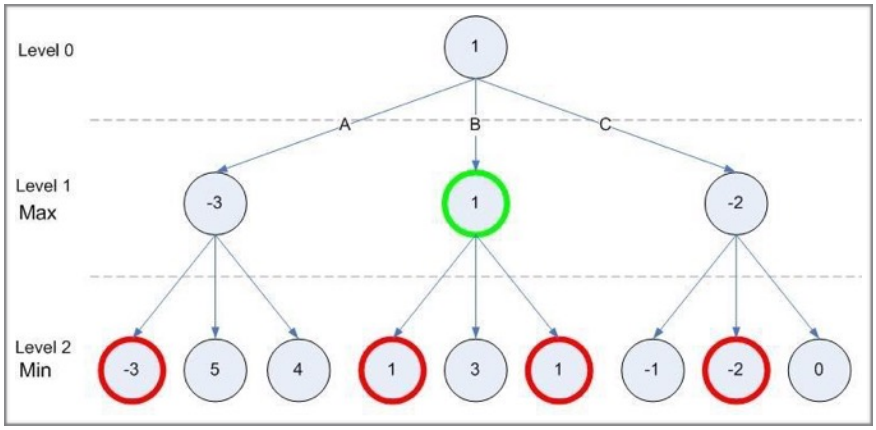
Računalniški igralec

Reversi spada med tako imenovane "zero-sum" igre, pri katerih dobiček za igralca pomeni enako veliko izgubo za nasprotnika. Število dovoljenih postavitev v igri reversi je ocenjena na okoli 10^{28}, pri čemer je velikost igralnega drevesa 10^{54} vozlišč. Optimalna igra je računsko zelo zahtevna. Obstajajo različni algoritmi za preiskovanje igralnega drevesa:

Minimax
Alfa-Beta prunning
MTD-f
NegaC*

Minimax

Minimax algoritem je optimalna strategija za igranje "zero-sum" iger. Temelji na ideji, da igralec, ki je na potezi poskuša maksimizirati dobiček, v naslednji potezi pa nasprotnik poskuša minimizirati igralčev dobiček. Implementacija je razmeroma preprosta, vendar je rešitev računsko potratna, saj bi bilo potrebno za optimalno igro preiskati celotno drevo, kar pa praktično ni izvedljivo.



Primer vejitve drevesa pri algoritmu Minimax

Pri preiskovanju se zato omejimo na določeno globino. V sami globini je tudi moč algoritma. Globina in računska zahtevnost sta sorazmerni. Globlji spust pomeni da je igralec pametnejši, vendar na ta račun potrebujemo boljši računalnik, ali pa problem rešujemo z uporabo paralelnih sistemov.

Paralelizacija algoritma Minimax

Naprej sekvenčno (brez uporabe tehnik paralelizacije) ustvarimo dovolj začetnih igralnih položajev. Pri reševanju visoko paralelnih problemov je najbolj pomembna učinkovita razdelitev dela med posamezne niti. Vsaka nit oz. proces dobi svoj začetni položaj in ga razišče v globino ter izračuna njegovo oceno. Na koncu rezultate glavna nit/proces združi in izbere ter izvede najboljšo potezo. Takšen pristop je splošen ne glede na izbrano arhitekturo oz. tehnologijo paralelizacije.

Paralelizacija z uporabo programskega jezika C in knjižnico Pthreads

Knjižnica Pthreads za paralelizacijo uporablja niti. Niti so sestavni deli procesa(programa v izvajanju), ki se izvajajo pod okriljem operacijskega sistema. Vsaki niti pripada lasten programski števec, lasten sklad in lasten kazalec na sklad, vendar si vse niti v procesu delijo isti kodni segment, isto kopico in statične podatke. Niti se izvajajo v istem naslovnem prostoru, zato je komunikacija med njimi preko pomnilnika precej bolj enostavna kot komunikacija med procesi. Moderni operacijski sistemi znajo razvrščati in preklapljati posamezne niti. Paralelizacija, ki se izvaja na takšni arhitekturi, pravimo paralelizacija z deljenim pomnilnikom.

Pthreads (POSIX Threads) je knjižnica funkcij jezika C, namenjena večnitnemu programiranju po standardu POSIX. Vsebuje več kot 60 funkcij za upravljanje z nitmi, delo s ključavnicami,…

Opis implementacije

Implementacija sledi splošnemu pristopu. Naprej sekvenčno generiramo vse možne poteze in jih shranimo v tabelo struktur. Vsak element tabele vsebuje x in y koordinato možnih potez. Glede na število možnih potez v vsaki rundi in glede na število niti razdelimo delo. Razdeljevanje je opisano v sledeči kodi:

```
#define NTHREADS 4          //število niti

…

int rem = n_moves%NTHREADS; //elementi ki so ostali po delitvi med niti
int sendcounts[NTHREADS];   //koliko elementov pošljemo vsaki niti
int displs[NTHREADS];       //odmiki za vsak segment
int sum = 0;

for (int i = 0; i < NTHREADS; i++) {
    sendcounts[i] = n_moves / NTHREADS;
    if (rem > 0) {
        sendcounts[i]++;
        rem--;
    }

    displs[i] = sum;
    sum += sendcounts[i];
}

int dynamic_n = 0;
if (n_moves < NTHREADS) dynamic_n = n_moves;
else dynamic_n = NTHREADS;
```

Razdeljevanje je enakomerno, če je število možnih potez v rundi deljivo s številom niti. V nasprotnem primeru program razdeli več dela nitim, ki jih kliče prej. Če je število niti večje kakor je število možnih potez, se število niti zmanjša na število možnih potez.

Primeri delitev:

Število niti	Število možnih potez v rundi	Razdelitev med niti
4	12	3,3,3,3
4	10	3,3,2,2
8	15	2,2,2,2,2,2,2,1
8	5	1,1,1,1,1

Vsaka nit dobi izračunano količino dela. Če je nit dobila številko 3, pomeni da bo preiskala 3 možne poteze. Delo razdeli glavna nit in v sledeči kodi izvede paralelizacijo s klici funkcije **pthread_create**(). Funkcija pthread_create ustvari novo nit in znotraj ustvarjene niti pokliče funkcijo func z argumenti arg.

pthread_create	
NAMEN	Ustvari novo nit
INCLUDE	`#include <pthread.h>`

UPORABA	`int pthread_create(pthread_t *thread,`
	`pthread_attr_t *attr,`
	`void *(*func) (void *)`
	`void *arg);`

ARGUMENTI	`thread` kazalec na spremenljivko tipa `pthread_t`
	`attr` kazalec na spremenljivko tipa `pthread_attr_t` ali `NULL`
	`func` funkcija, ki jo bo izvedla nit
	`arg` argumenti funkcije `func`

VRNE	0, če je klic uspel errcode, če klic ni uspel

```
for (int i = 0; i < dynamic_n; i++)
{
    //priprava potrebnih podatkov za posamezni izračun
    memcpy(args[i].board, board, BOARD_SIZE);
    args[i].player_color = player_color;
    args[i].max_depth = max_depth;
    args[i].depth = 1;
    args[i].count = sendcounts[i];
    args[i].displacement = displs[i];
    args[i].id = i;

    pthread_create(&threads[i], NULL, minimax_thread, (void *)&args[i]);
}
```

Znotraj funkcije **minimax_thread**() se pokliče rekurzivni algoritem Minimax tolikokrat kolikor dela je prejela nit in po vsakem klicu shrani rezultat v globalno tabelo rezultatov (ret_score[ ]).

Ko niti končajo z delom jih glavna nit združi s klicom funkcije **pthread_join**() in preveri v globalni tabeli rezultatov največji rezultat ter izvede najboljšo potezo.

pthread_join	
NAMEN	Čaka, da se nit zaključi
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_join(pthread_t *thread, void **retval);`
ARGUMENTI	`thread` kazalec na spremenljivko tipa `pthread_t` `retval` kazalec na spremenljivko, ki sprejme povratno vrednost iz niti
VRNE	0, če je klic uspel errcode, če klic ni uspel

```
for (int i = 0; i < dynamic_n; i++)
{
    pthread_join(threads[i], NULL);
}

for (int i = 0; i < n_moves; i++)
{
    //Najdemo najboljšo potezo
    if (ret_score[i] * player_color > score*player_color) {
        score = ret_score[i];
        row = moves[i].i;
        column = moves[i].j;
    }
}
```

Paralelizacija z uporabo programskega jezika C in specifikacijo MPI

Specifikacija MPI za paralelizacijo uporablja več-računalniške sisteme. Za razliko od sistemov z deljenim pomnilnikom, so tu pomnilniki nepovezani. Vsak procesor ima neposreden dostop samo do svojega pomnilnika. Ker nimajo skupnega pomnilnika, procesorji med seboj komunicirajo s pošiljanjem sporočil. Programer mora eksplicitno poskrbeti za oddajanje in sprejemanje sporočil.

MPI (Message Passing Interface) je specifikacija za razvijalce in uporabnike knjižnic. Je *de facto* standard za izmenjevanje sporočil. MPI-2 nam ponuja 152 funkcij za komunikacijo. Paralelizem po specifikaciji MPI je ekspliciten.

Programer je odgovoren za identifikacijo paralelnih delov algoritma in implementacijo z uporabo konstruktov MPI. Za prenos podatkov je potrebno sodelovanje vseh vpletenih procesov.

Opis implementacije

Pri MPI implementaciji že zelo zgodaj v programu kličemo inicializacijo večih

```
int rank, num_procs;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

procesov.
Funkcija **MPI_Init**() zahteva 2 argumenta, ki sta kar število argumentov programa in argumenti programa. Preko argumentov programa lahko funkcija določi število procesov na katerih se bo izvajala koda ter posebne nastavitve okolja. **MPI_Comm_size**() vrne število procesov na katerih se izvaja program, **MPI_Comm_rank**() pa vrne rank oziroma identifikator procesa. Rank je izredno pomemben v nadaljnjem programu, saj preko ranka ločimo kateri del kode se bo izvajal na posameznem procesu.

Takoj po inicializaciji procesov sledi if stavek, kjer se ločimo na dve vrsti procesov. Če je rank enak 0 pomeni da smo v glavnem procesu, ki skrbi za oskrbovanje svojih delavskih procesov, če pa rank ni enak nič pa pomeni da smo se znašli v delavskem procesu. Torej se pri MPI implementaciji delo razdeli samo na delavske procese in njihovo število je vedno za eno manjše od vseh procesov(če odštejemo glavni proces).

Delo se deli enako kot pri implementaciji s knjižnico Pthreads, vendar je razlika v tem, da se pri MPI ne da uporabljati globalnih spremenljivk, zato je treba vse podatke poslati na proces.

Priprava posameznih sporočil za procese in klic funkcije **MPI_Send**() je prikazan v sledeči kodi. **MPI_Send**() je blokirajoči klic, lahko je sinhron ali preko medpomnilnika. Klic funkcije se zaključi, ko je sporočilo varno na poti.

```
for (int i = 0; i < num_procs; i++)
{
     //Priprava sporočila
     int message[MESS_SIZE];
     message[0] = BOARD_SIZE;

     for (int j = 1; j<=BOARD_SIZE; j++) {
          message[j] = tempBoard[j-1];
     }

     message[BOARD_SIZE+1] = player_color;
     message[BOARD_SIZE+2] = max_depth;
     message[BOARD_SIZE+3] = 1;
     message[BOARD_SIZE+4] = sendcounts[i];

     int tj = displs[i];
     for (int j = BOARD_SIZE+5; j<(BOARD_SIZE+5+sendcounts[i]*2); j+=2) {
            message[j] = moves[tj].i;
            message[j+1] = moves[tj].j;
            tj++;
     }

     int length = BOARD_SIZE+5 + sendcounts[i]*2;

     //Pošiljanje sporočila
     MPI_Send(&message, length, MPI_INT, i+1, 0, MPI_COMM_WORLD);
}
```

V sporočilu pošljemo velikost igralne deske trenutno stanje na njej, kateri igralec je trenutno na vrsti, do kakšne globine se naj spustimo v rekurzivnem algoritmu, kakšna je trenutno globina, koliko "dela" naj opravi vsak proces (koliko potez mora obdelati) in pa tudi vse koordinate potez, ki nastopajo v zaporednih parih (message[i]=x1, message[i+1]=y1, message[i+2]=x2,…).

Med tem ko proces z rankom 0 računa delo za delavske procese, delavski procesi čakajo na sporočilo od glavnega procesa. To opravijo s funkcijo **MPI_Recv**().

```
while(1)
{
     int message[MESS_SIZE];
     MPI_Recv(&message, MESS_SIZE, MPI_INT, 0, MPI_ANY_TAG,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);

     …

     int r_scores[cnts];

     for (int i = 0; i < cnts; i++)
     {
         r_scores[i] = minimax(t_board, p_color, m_depth, dpth,
                               temp_moves[i].i, temp_moves[i].j);
     }

     MPI_Send(&r_scores, cnts, MPI_INT, 0, rank, MPI_COMM_WORLD);
}
```

Proces prejme sporočilo od glavnega procesa in ga v obratni smeri obdela in izlušči potrebne informacije za izračun rezultatov potez. Rezultate shranjuje v tabeli **r_scores[ ]** in jo pošlje nazaj glavnemu procesu.

```
for (int i = 0; i < num_procs; i++)
{
     int message[MESS_SIZE];
     MPI_Recv(&message, MESS_SIZE, MPI_INT, i+1, MPI_ANY_TAG,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);

     for (int j = 0; j<sendcounts[i]; j++) {
         ret_score[displs[i]+j] = message[j];
     }
}

for (int i = 0; i < n_moves; i++)
{
    //Najdemo najboljšo potezo
    if (ret_score[i] * player_color > score*player_color) {
        score = ret_score[i];
        row = moves[i].i;
        column = moves[i].j;
    }
}
```

Glavni proces med tem čaka na rezultate in ko jih prejme jih združi v tabelo **ret_scores** ter na enak način kakor pri tehnologiji Pthreads najde najboljšo potezo in jo izvede.

Rezultati

Pri testiranju programov sem uporabljal različne konfiguracije. Spreminjal sem globino preiskovanja pri različnem številu niti/procesov. Število potez na igro se je gibalo med 30 in 32. Paralelen algoritem je tekmoval proti naključnemu izbiranju potez. Rezultati so predstavljeni v sekundah na igro.

Računalnik: MacBook Air(2014) , 1.4 GHz Intel Core i5 , 4GB 1600MHz DDR3

Globina preiskovanja: 4

Sekvenčni algoritem	Zmag.	Pthreads št. niti 2	Zmag.	Pthreads št. niti 16	Zmag.	MPI 4 procesi	Zmag.
1.04	random	2.477	random	0.92	random	3.48	random
2.67	random	2.668	random	0.41	minimax	3.30	random
2.36	minimax	2.085	random	0.39	minimax	3.05	random
2.12	minimax	1.933	random	1.282	random	2.49	random
1.73	remi	2.28	random	0.85	random	3.34	random
0.44	minimax	2.525	minimax	1.579	random	3.17	random
3.05	random	2.657	random	1.401	random	4.15	random
2.06	minimax	1.892	random	1.678	minimax	3.21	random
2.45	minimax	1.89	minimax	1.554	random	2.41	random
2.26	minimax	2.443	random	1.63	minimax	3.19	random
2.02	60%	2.28	20%	1.17	40%	3.18	0%

Globina preiskovanja: 5

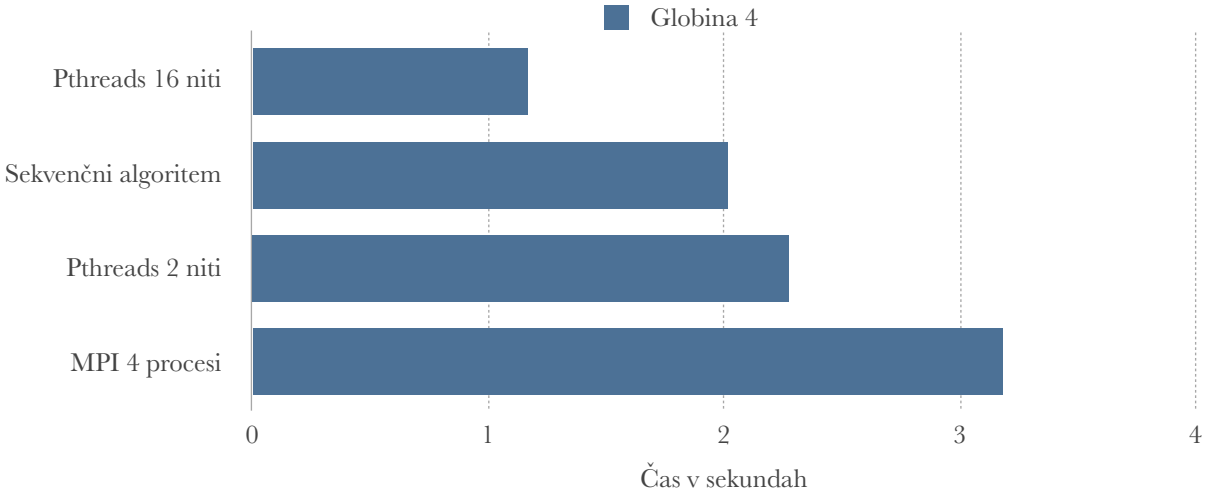
Sekvenčni algoritem	Zmag.	Pthreads št. niti 16	Zmag.	MPI 4 procesi	Zmag.	MPI 8 procesov	Zmag.
3.62	minimax	1.937	random	23.65	random	7.38	random
5.86	minimax	1.758	random	13.15	minimax	10.50	minimax
28.27	minimax	1.019	minimax	12.41	random	4.60	minimax
0.51	minimax	1.122	minimax	11.13	minimax	8.01	minimax
7.56	minimax	1.672	random	15.41	random	5.32	minimax
23.43	minimax	1.56	minimax	6.63	random		
9.53	random	1.752	random	9.78	minimax		
9.90	minimax	1.345	minimax	14.71	minimax		
9.70	minimax	1.556	remi	10.95	random		
8.60	minimax	2.337	random	5.41	minimax		
10.70	90%	1.61	40%	12.32	50%	7.16	80%

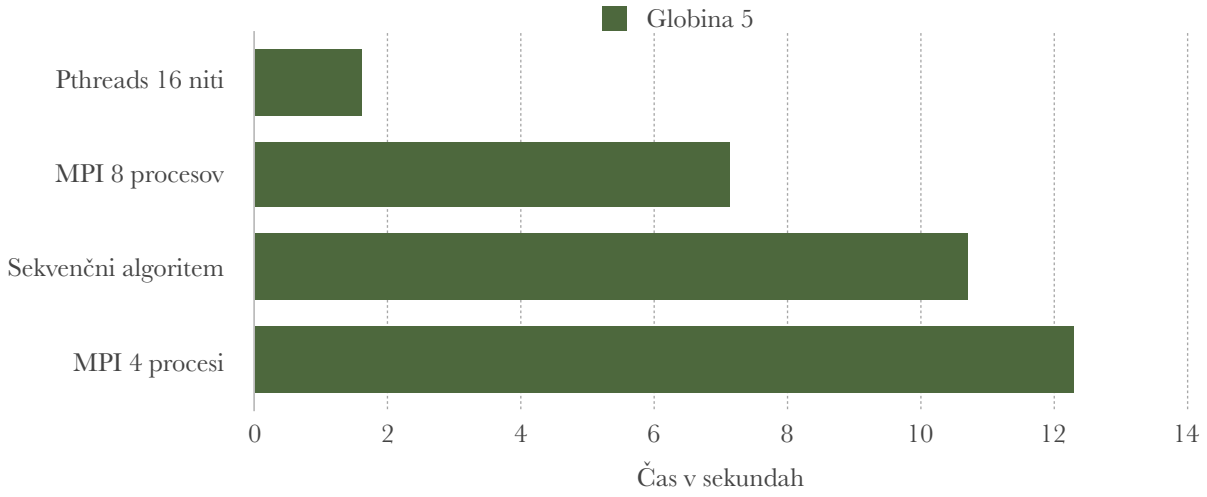
Globina preiskovanja: 6

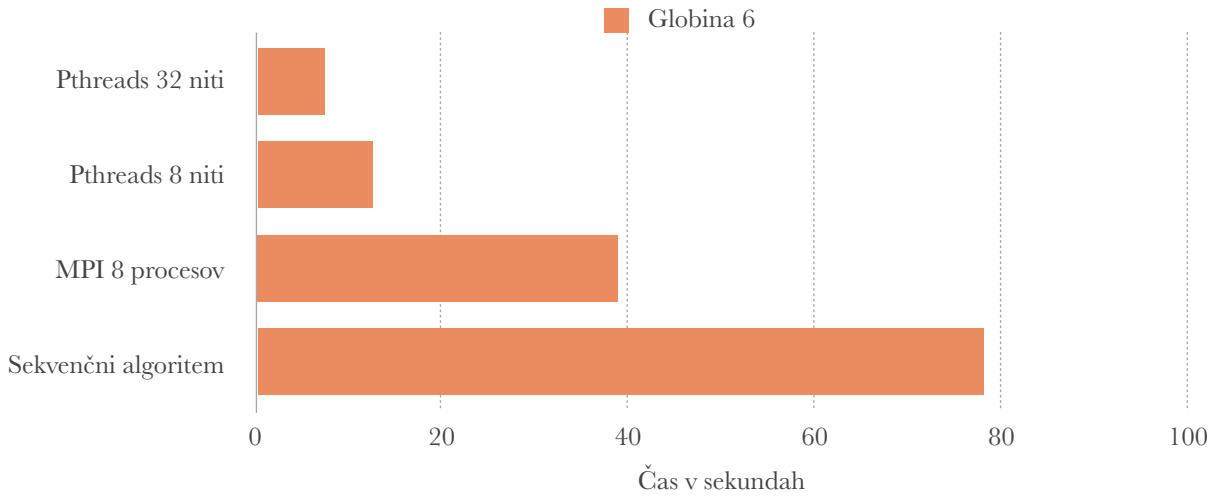
Sekvenčni algoritem	Zmag.	Pthreads št. niti 8	Zmag.	Pthreads št. niti 32	Zmag.	MPI 8 procesov	Zmag.
66.07	random	13.961	minimax	7.594	minimax	29.26	minimax
99.81	minimax	10.708	minimax	5.732	minimax	53.40	minimax
103.02	minimax	12.995	random	3.819	minimax	33.37	minimax
94.70	minimax	17.849	random	9.358	minimax	50.75	minimax
102.18	minimax	17.988	random	6.896	random	28.34	minimax
70.32	minimax	8.032	minimax	6.53	random		
50.93	minimax	14.687	minimax	4.279	minimax		
70.8	minimax	7.18	random	14.704	random		
89.48	minimax	7.568	minimax	9.708	random		
33.37	minimax	14.055	minimax	5.362	minimax		
78.07	90%	12.50	60%	7.40	60%	39.02	100%

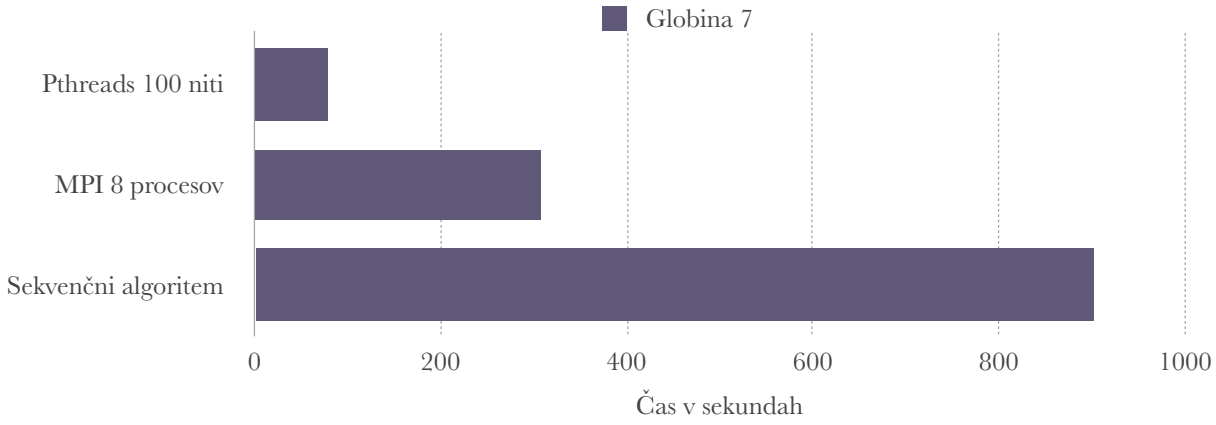
Globina preiskovanja: 7

Sekvenčni algoritem	Zmag.	Pthreads št. niti 100	Zmag.	MPI 8 procesov	Zmag.
850.51	minimax	50.35	minimax	347.34	minimax
953.50	minimax	108.85	minimax	267.64	minimax
902.01	100%	79.60	100%	307.49	100%









Komentar rezultatov

Paralelizacija je prišla popolnoma do izraza z globljim spustom po drevesu, saj je takrat računalnik potreboval veliko računske moči in posledično je algoritem postajal počasnejši.

Pri globini 4 nam je priprava na paralelizacijo in pošiljanje veliko podatkov pri MPI odvzelo kar nekaj časa in se je tako uvrstil na zadnje mesto. Pthreads tehnologija se je izkazala z visokim številom niti, ker je takrat vsaka nit dobila maksimalno eno ali dve potezi in je računanje potekalo bliskovito hitro. Pri samo dveh nitih pa je bila delitev dela za vsako nit očitno prezahtevna tudi za sekvenčni algoritem. Sekvenčni algoritem se je odrezal precej dobro, ker ni bilo deljenj dela in globina je bila še obvladljiva. Hitrost se je poznala na odstotkih zmag, precej pogosto se je dogajalo, da je naključni algoritem premagoval paralelni algoritem.

Faktor pohitritve za globino 4 pri najboljši tehnologiji (Pthreads s 16 nitmi) je 1,73. Skupen odstotek zmag je 30%;

Pri globini 5 se je zopet izkazalo 16 Pthreads niti, ki so v povprečju za igro potrebovale 1,61 sekunde, veliko manj od MPI in sekvenčnega algoritma. 4 MPI procesov je očitno še vedno premalo, da bi se kosali s sekvenčnim algoritmom. Je pa zadostovala konfiguracija osmih, ki je potrebovala za eno igro v povprečju približno tretjino manj časa kot sekvenčni algoritem. Maksimalni faktor pohitritve pri najboljši tehnologiji (Pthreads s 16 nitmi) je 6,64. Skupen odstotek zmag je narasel na 65%.

Globina 6 je brez paralelnega algoritma izredno zamudna. Pri MPI delitev dela in pošiljanje sporočil ne igra več velike vloge, saj je na 8 procesih pohitritev glede na sekvenčni algoritem dvakratna. Zopet sta najboljši performans dosegli konfiguraciji s Pthreads nitmi. Maksimalni faktor pohitritve pri najboljši tehnologiji (Pthreads z 32 nitmi) je 10,55. Odstotek zmag je narasel že na 77,5%.

Pri globini 7 lahko sekvenčni algoritem kar pozabimo. Čeprav je tudi za paralelne rešitve precej zamudno računanje je odstotek zmag 100%. Čeprav sem izvedel za vsako tehnologijo samo po dva testa, sem opazil, da so zmage precej bolj izrazite kot pri manjših globinah. Zato lahko skoraj z zagotovostjo trdim da proti naključnemu igralcu težko izgubimo. Maksimalni faktor pohitritve pri najboljši tehnologiji (Pthreads s 100 nitmi) je 11,33. Čeprav večinoma 100 niti algoritem ne potrebuje, sem želel pognati algoritem in sistem do skrajnosti.

Globina	Najboljša konfiguracija	Faktor pohitritve	Odstotek zmag
4	Pthreads 16 niti	1.73	30
5	Pthreads 16 niti	6.64	65
6	Pthreads 32 niti	10.55	77.5
7	Pthreads 100 niti	11.33	100

Zaključek

Algoritem je deloval. Čeprav naj bi ble teoretične pohitritve precej višje, je potrebno pri trenutni implementaciji upoštevati da je še precej kode sekvenčne, paralelen je le rekurziven minimax algoritem. Veliko časa pri MPI vzame pošiljanje sporočil in čakanje na odgovor. Najboljša bi bila implementacija MPI v kombinaciji s Pthreads, kjer bi lahko prvo delitev dela, na vsakem procesu še dodatno razdelili in bi potem vsaka nit računala samo eno potezo. Trenutna implementacija pa dopušča da vsaka nit/proces lahko dobi več kot eno potezo, če se znajde v takšni situaciji in s tem se logično podaljšuje čas izvajanja programa. Problematična je tudi neenakomerna razporeditev dela med niti oz. procese. Lahko se zgodi da zaradi narave problema, neka nit/proces dobi manjšo količino dela kot druga nit/proces in zaradi tega prej konča in je nato v stanju čakanja.

Če sklenemo kompromis med časom izvajanja in odstotkom zmag, je idealen izbor globine definitivno globina 6.