Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jure Grabnar

Drevesno preiskovanje Monte Carlo v porazdeljenem okolju

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Branko Šter
Somentor: izr. prof. dr. Uroš Lotrič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Drevesno preiskovanje Monte Carlo v porazdeljenem okolju (Monte Carlo Tree Search in a distributed environment)

Tematika naloge:

Na osnovi knjižnice MPI (Message Passing Interface) paralelizirajte algoritem drevesnega preiskovanja Monte Carlo (Monte Carlo Tree Search, MCTS) na primeru igre Gomoku. Algoritem poganjajte na omrežju grid, za katerega napišite tudi navodila, ki bodo novim uporabnikom omogočala uspešno vključitev v omrežje. Izmerite pohitritev paralelne implementacije algoritma MCTS pri različnem številu procesorjev in različnem številu simuliranih iger.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jure Grabnar, z vpisno številko **63110167**, sem avtor diplomskega dela z naslovom:

Drevesno preiskovanje Monte Carlo v porazdeljenem okolju

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Branka Štera in somentorstvom izr. prof. dr. Uroša Lotriča,
- 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 na svetovnem spletu prek univerzitetnega spletnega arhiva.

V Ljubljani, dne 3. septembra 2015 Podpis avtorja:

Zahvaljujem se svojemu mentorju prof. dr. Branku Šteru in somentorju izr. prof. dr. Urošu Lotriču za strokovno pomoč med izdelavo diplomske naloge. Zahvala gre tudi asistentu Tomu Vodopivcu, ki je bil vedno pripravljen svetovati, kako še izboljšati meritve in rezultate. Hvala tudi Matevžu Markoviču za pomoč pri izdelavi navodila za omrežje grid. Posebna zahvala gre tudi mojim staršem in vsem domačim za potrpljenje in spodbudo med študijem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
MCTS	Monte Carlo Tree Search	drevesno preiskovanje Monte Carlo
SLING	Slovenian Initiative for National Grid	Slovenska iniciativa za nacionalni grid
EGI	European Grid Initiative	Evropska iniciativa za grid
MPI	Message Passing Interface	vmesnik za izmenjevanje sporočil
UCB	Upper Confidence Bounds	zgornja meja zaupanja
UCT	UCB applied to Trees	zgornja meja zaupanja za drevesa

Povzetek

Algoritem drevesnega preiskovanja Monte Carlo (MCTS) je računsko precej zahteven, poleg tega pa čas računanja vpliva na kakovost rezultatov. Namen dela je zato paralelizacija metode MCTS. S paralelizacijo se poveča število iger in drugih parametrov - rezultati so boljši in bolj zanesljivi. Paralelni algoritem smo napisali s pomočjo knjižnice MPI, ki omogoča izvajanje na več računalnikih. Čas izvajanja algoritma smo merili na različnih velikostih problema. Rezultati paralelizacije so bili zadovoljivi, saj je bila pohitritev večinoma linearna. Algoritem smo izvajali na omrežju grid, za katerega skrbi Slovenska iniciativa za nacionalni grid. V okviru dela so nastala tudi navodila za uporabo omrežja grid.

Ključne besede: drevesno preiskovanje Monte Carlo, porazdeljeni sistemi, SLING, umetna inteligenca, MPI.

Abstract

Monte Carlo Tree Search algorithm (MCTS) is a computationally expensive algorithm. The time needed for computation correlates with the quality of the results. The purpose of this work is to parallelize MCTS method. With parallelization we gain an ability to increase the number of simulated games per turn and other parameters and still be able to receive results in sufficient time. Quality of results has been improved significantly. Parallel algorithm was written in MPI library which enables the program to run on multiple computers. Algorithm was evaluated on different problem sizes. With big enough problem, the speedup was approximately linear. Algorithm was run on a grid network which is administered by Slovenian Initiative for National Grid (SLING). As a part of this work, instructions for usage of grid network were created.

Keywords: Monte Carlo Tree Search, distributed systems, SLING, artificial intelligence, MPI .

Poglavje 1

Uvod

Pred desetimi leti so bili najboljši računalniški igralci Goja mačji kašelj za profesionalne igralce. Igra Go je tako kompleksna, da je preiskovanje s klasičnimi algoritmi (minimaks, alfa-beta rezanje) prineslo slabe rezultate. Treba je bilo uporabljati hevristiko in omejevati globino preiskovanja. Čeprav so bili algoritmi še tako dobri in premeteni, niso bili kos človeškim igralcem. Leta 2006 je raziskovalec R. Coulom prvi uporabil metodo drevesnega preiskovanja Monte Carlo [22] pri igri Go. Algoritem je poimenoval CrazyStone. Na računalniški olimpijadi, ki je potekala tisto leto, je premagal vso konkurenco in osvojil zlato medaljo. Metoda drevesnega preiskovanja Monte Carlo je takrat dobila svoje mesto v umetni inteligenci.

Naša sekvenčna implementacija drevesnega preiskovanja Monte Carlo (MCTS) je za izračun rezultatov potrebovala več dni. Odločili smo se paralelizirati algoritem v okolju MPI ter tako povečati število simuliranih iger. V delu smo predstavili čase te paralelne implementacije in jih primerjali s časi sekvenčne kode. Algoritem smo poganjali na omrežju grid, za katerega skrbi Slovenska iniciativa za nacionalni grid (SLING). Želeli smo testirati infrastrukturo in ugotoviti, ali je primerna za naš problem. V okviru tega dela so nastala tudi navodila za uporabo omrežja grid.

Sprva smo na kratko predstavili metodo drevesnega preiskovanja Monte Carlo in omrežje grid. V Poglavju 4 smo omenili navodila za uporabo omrežja grid. Nato smo opisali naše implementacije paralelizacije algoritma. V Poglavju 5 smo opisali okolje, kjer se je algoritem izvajal, ter predstavili in evalvirali rezultate. Delo smo končali z diskusijo o svojih opažanjih in zaključkom, kjer smo predstavili smernice za nadaljnje delo.

Poglavje 2

Drevesno preiskovanje Monte Carlo

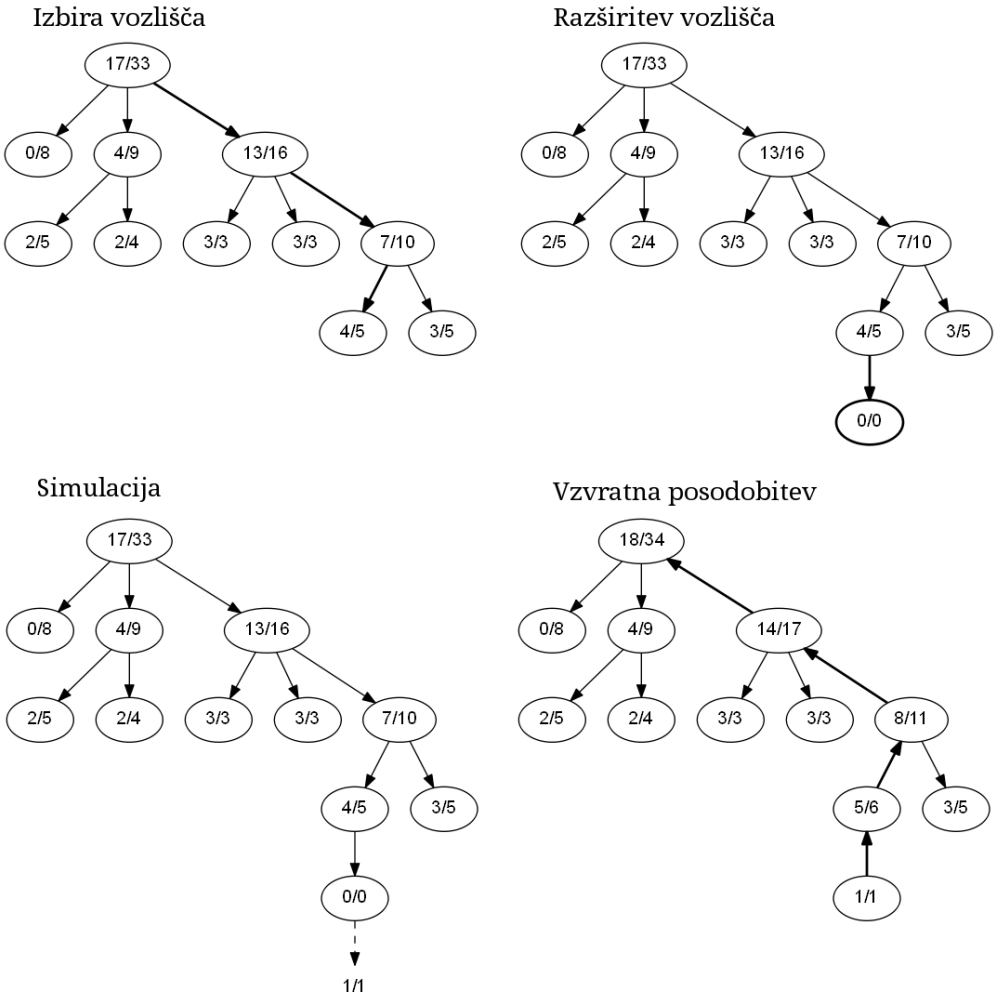
Drevesno preiskovanje Monte Carlo (ang. MCTS - Monte Carlo Tree Search) je preiskovalni algoritem, ki na podlagi simulacije z metodo Monte Carlo gradi drevo, na podlagi drevesa pa izbere najboljšo potezo. Uporablja se predvsem v umetni inteligenci pri igranju iger. Za nekatere igre so klasični algoritmi (kot sta npr. minimaks in alfa-beta rezanje) zadovoljivi, pri nekaterih igrah pa je vejitveni faktor prevelik za iskanje optimalne rešitve. Raziskovalci so se z igro Go, ki ima velik vejitveni faktor, ukvarjali že vrsto let. Leta 2006 je R. Coulom prvi predstavil svoj algoritem Crazy Stone na manjši plošči Go, veliki 9 x 9 polj, kjer je uporabil metodo MCTS [22]. Izkazal se je kot zelo uspešen in današnji najboljši algoritmi že konsistentno premagujejo dobre igralce Goja (vendar še niso kos najboljšim igralcem). Odtlej raziskovalci metode MCTS uporabljajo tudi za druge igre in pri tem so pogosto uspešni [17].

Metoda MCTS je v osnovi razdeljena na 4 korake [20]:

- **Izbira vozlišča** - na podlagi trenutnih podatkov v drevesu se izbere naslednje vozlišče, iz katerega bomo začeli simulacije. Treba je najti kompromis med najbolj obetavnim trenutnim vozliščem in raziskovanjem manj raziskanih vozlišč.
- **Razširitev vozlišča** - na izbranem vozlišču se doda novo vozlišče, ki predstavlja novo akcijo.

- **Simulacija** - začetna točka simulacije je na novo dodano vozlišče. Poteze se od tukaj naprej izbirajo naključno.

- **Vzvratna posodobitev** - ob končani simulaciji se posodobijo podatki na vseh vozliščih na poti med dodanim vozliščem in korenom drevesa.



Slika 2.1: Štirje osnovni koraki metode MCTS

Slika 2.1 predstavlja štiri korake v eni potezi MCTS. Številke v vozliščih predstavljajo "število zmag/število simuliranih iger" iz danega vozlišča. Najprej si izberemo najbolj obetaven list v drevesu, ga razširimo z novim vozliščem "0/0" in iz tega vozlišča simuliramo igro. Po koncu simulacije povečamo (v našem primeru 1 zmaga) rezultate v vseh vozliščih, ki vodijo do novega vozlišča.

Pri koraku izbiranja vozlišča je treba najti kompromis med izkoriščanjem znanja (izbira najbolj obetavnega vozlišča) in raziskovanjem (izbira manj raziskanih vozlišč). Za ta namen uporabljamo formulo UCT (Upper Confidence Bounds applied to trees), ki sta jo predstavila Kocsis in Szepesvári [23]:

$$\frac{r_i}{n_i} + C \cdot \sqrt{\frac{\ln n_p}{n_i}} \tag{2.1}$$

kjer je:

- r_i - število zmag v vozlišču i,
- n_i - število obiskov vozlišča i,
- n_p - število obiskov vozlišča p, ki je starš vozlišča i,
- C - empirično določena konstanta, ki pove ravnotežje med izkoriščanjem znanja in raziskovanjem manj raziskanih vozlišč.

Prvi del enačbe ($\frac{r_i}{n_i}$) meri trenutno oceno vozlišča, preostali del enačbe raziskovanje drevesa. S konstanto C, ki jo določimo sami, izberemo, kako pristranski želimo biti do raziskovanja - večji C pomeni, da želimo več poudarka na raziskovanju, manjši C pa daje več poudarka na izkoriščanje znanja. Enačba UCT (2.1) izhaja iz enačbe UCB (Upper Confidence Bounds). V koraku izbire vozlišča se sprehodimo po drevesu. Za vse otroke trenutnega vozlišča izračunamo oceno UCT in izmed njih izberemo vozlišče z največjo oceno. Ko pridemo do lista, preidemo v korak razširitve vozlišča.

2.1 Paralelizacija MCTS

2.1.1 Paralelizacija listov

Paralelizacija listov je eden najlažjih načinov paralelizacije metode MCTS. Prva sta jo opisala Cazenave in Jouandeau [18]. Glavna nit izbere vozlišče in ga razširi. Iz tega vozlišča nato vsaka nit simulira svoje igre neodvisno od drugih niti. Po koncu simulacije glavna nit zbere rezultate in posodobi vozlišča (4. korak pri MCTS). Največja težava te metode je prav v tem, da niti delujejo neodvisno. Če glavna nit izbere vozlišče s slabo potezo, potem lahko sklepamo, da se bo večina simuliranih iger končala s porazom. To je potrata računske moči, saj bi lahko isti izid napovedala ena nit, druge niti pa bi medtem simulirale igre iz drugega vozlišča in tako pripomogle k večji razvejanosti drevesa - ta metoda se imenuje paralelizacija korena.

2.1.2 Paralelizacija korena

Podobno kot pri paralelizaciji listov niti delujejo neodvisno. Vsaka nit gradi svoje drevo (lahko iz različnega vozlišča). Ob koncu simulacije se rezultati teh dreves za vsako posamezno vozlišče seštejejo in dodajo v glavno drevo.

2.1.3 Drugi načini paralelizacije

Raziskovalci iščejo načine, ki bi izboljšali osnovna dva načina paralelizacije. Chaslot [21] je predstavil paralelizacijo drevesa. Gre za paralelizacijo korena, kjer lahko vsaka nit dostopa do glavnega drevesa in ga spreminja. Zato so potrebne ključavnice in t. i. "navidezni poraz". Kadar več niti začne izbiro vozlišča iz istega korena, se lahko zgodi, da vse izberejo isto vozlišče. Drevo, ki ima lahko več milijonov vozlišč, se tako raziskuje iz enega samega vozlišča. Chaslot je uvedel metodo "navideznega poraza" [19]. Kadar nit obišče vozlišče, se temu vozlišču dodeli poraz in tako se njegova vrednost zniža. Naslednja nit bo izbrala to vozlišče le, če bo to še vedno najobetavnejše vozlišče. Na tak način se algoritem prisili, da se drevo bolj raziskuje. Za zdaj je paralelizacija drevesa po rezultatih še vedno na ravni paralelizacije korena.

Poglavje 3

Omrežje grid

Gruča je skupina računalnikov. Grid je več gruč skupaj. Gruče so lahko na različnih, geografsko ločenih lokacijah. Gruče se med seboj povezujejo prek akademskega omrežja in vmesne programske opreme [8]. Grid deluje kot superračunalnik, ki je razpršen po različnih lokacijah. Najbolj se izkaže pri izvrševanju nalog, kjer je potrebna velika računska moč in kjer želimo nalogo poganjati na veliki količini podatkov.

Leta 2009 je Arnes ustanovil Slovensko inicativo za nacionalni grid (SLING) [12]. Cilj je bil zagotoviti infrastrukturo in enake možnosti slovenskim raziskovalcem v evropski znanosti. SLING je vstopil med člane Evropske iniciative za grid (EGI). Pomagali so pri vzpostavitvi osrednje organizacije EGI.eu [4], ki danes podpira omrežje več kot 350 centrov (300 tisoč jeder in 200 PB prostora [3]).

Osnovna infrastruktura v Sloveniji je postavljena na Inštitutu Jožef Stefan in Arnes. Uporablja jo lahko vsak raziskovalec, ki izkaže svojo identiteto s certifikatom, ki ga za Slovenijo izdaja SiGNET [15]. S certifikatom se izkaže identiteta uporabnika, vendar ne zadostuje za pridobitev pravic uporabe grida. Uporabnik se mora včlaniti v virtualno organizacijo, ki lahko združuje več različnih fizičnih organizacij. Virtualna organizacija dovoljuje oz. omejuje dostop do sredstev grida. S certifikatom se torej uporabnik predstavi virtualni organizaciji, ta pa mu dovoli uporabo omrežja grid. Za uporabnike, ki nimajo virtualne organizacije, je namenjena Slovenska splošna virtualna organizacija.

Za oddajo poslov potrebujemo vmesno programsko opremo za grid. Podprta sta NorduGrid ARC [9] in gLite [6], vendar so gLite že nehali razvijati. V navodilih

za uporabo (Poglavje 4) smo predstavili uporabo programske opreme Nordugrid ARC.

Za shranjevanje podatkov skrbi dCache [2], ta podatke, ki so razpršeni po več različnih heterogenih strežnikih, združi v en navidezen datotečni sistem.

CREAM (Computing Resource Execution And Management) [1] posle upravlja nizkonivojsko (pošiljanje, nadzor, prekinitev poslov).

Tehnologij, ki skrbijo za pravilno delovanje omrežja grid, je še veliko, nas pa zanima predvsem iz uporabniškega vidika. Za ta namen smo napisali navodila za uporabo omrežja grid, da uporabniku olajšamo prve korake.

Poglavje 4

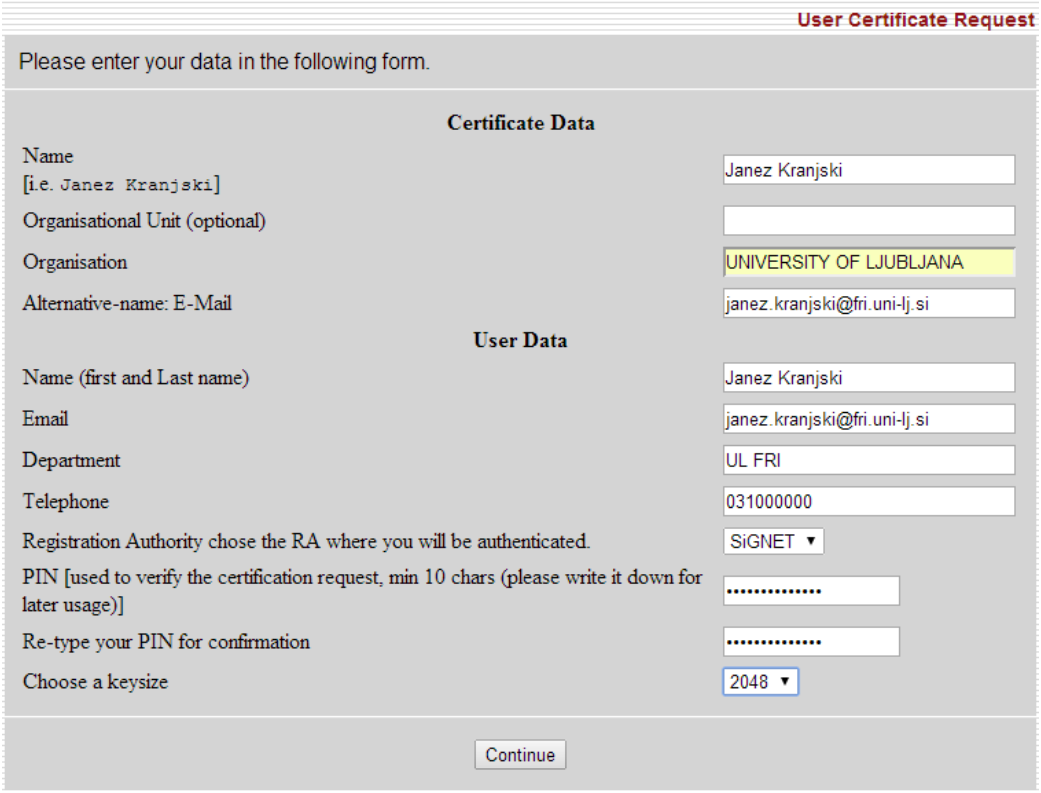
Navodila za uporabo omrežja grid

Večstopenjska namestitev delovnega okolja in pomanjkanje navodil otežijo uporabnikove prve korake pri dostopu do omrežja grid. V nadaljevanju smo zato predstavili strnjena navodila od zaprositve za uporabo omrežja do zagona prvega programa na omrežju. Priporočeno je, da za spletne strani uporabljamo brskalnik Mozila Firefox, sicer se pojavijo težave s prikazom vsebine.

4.1 Pridobitev certifikata

Postopek za zaprositev certifikata je opisan na spletni strani [15]. Po zaprositvi za certifikat po elektronski pošti dobimo obvestilo, da lahko certifikat prevzamemo na povezavi [11]. Kliknemo na zavihek "User", nato na gumb "Request a User Certificate" in nazadnje na gumb "Request a certificate with automatic browser detection". Pričaka nas obrazec, prikazan na sliki 4.1.

Izpolnimo obrazec in s tem pridobimo certifikat SiGNET. Certifikat uvozimo v brskalnik (Mozila Firefox je priporočen brskalnik). Zdaj se je treba včlaniti v VOMS, kjer pridobimo pravico do dostopa omrežja. Obiščemo spletno stran [16]. Če se pojavi napaka, povezana s SSL, pomeni, da certifikata nismo uvozili - uporabiti moramo brskalnik, ki ima uvožen certifikat, pridobljen v prejšnjem koraku. Po uspešni identifikaciji izpolnimo obrazec za novega uporabnika. Počakamo, da



Slika 4.1: Obrazec za pridobitev certifikata

administrator odobri članstvo - obvestilo dobimo po elektronski pošti. Zdaj imamo dostop do omrežja grid. V naslednjem koraku bomo namestili delovno okolje.

4.2 Namestitev delovnega okolja

Programsko opremo lahko namestimo na katerikoli operacijski sistem, ki podpira standard POSIX. Obstaja več možnosti namestitve [14]. V nadaljevanju smo opisali postopek namestitve vmesne programske opreme NorduGrid ARC na operacijski sistem Ubuntu 14.04.

4.2.1 Namestitev NorduGrid ARC

Potrebovali bomo certifikat SiGNET, ki smo ga pridobili v prejšnjem koraku. Za druge operacijske sisteme si postopek lahko preberemo na [10]. Odpremo terminal (bližnjica Ctrl+Shift+T). Namestiti je treba ključe GPG za NorduGrid ARC, ki se uporabljajo za avtentikacijo:

```
wget -q http://download.nordugrid.org/DEB-GPG-KEY-nordugrid.asc -O- | sudo apt-key add -}
```

Pridobimo podatke o repozitoriju:

```
wget http://download.nordugrid.org/packages/nordugrid-release/releases/13.11/ubuntu/14.04/amd64/nordugrid-release_13.11~trusty1_all.deb
```

In jih vnesemo v sistem:

```
sudo dpkg -i nordugrid-release_13.11\~trusty1_all.deb
```

Zdaj lahko namestimo NorduGrid ARC iz repozitorija:

```
sudo apt-get install nordugrid-arc-client nordugrid-arc-plugins-globus
```

4.2.2 Namestitev certifikatnih agencij

Namestimo paket, ki vsebuje zaupanja vredne certifikatne agencije. Odobril jih je IGTF (Interoperable Global Trust Federation). Namestimo ključe GPG:

```
wget -q -O - https://dist.eugridpma.info/distribution/igtf/
    current/GPG-KEY-EUGridPMA-RPM-3 | sudo apt-key add -
```

Vnesemo podatke o repozitoriju v datoteko */etc/apt/sources.list*:

```
echo -e "\n#### EGI Trust Anchor Distribution ####\ndeb http
    ://repository.egi.eu/sw/production/cas/1/current egi-igtf
    core" | sudo tee -a /etc/apt/sources.list > /dev/null
```

Osvežimo metapodatke in namestimo metapaket:

```
apt-get update
apt-get install ca-policy-egi-core
```

4.2.3 Nameščanje certifikata

Certifikat SiGNET, ki smo ga pridobili, je treba spremeniti v pravo obliko. Ustvarimo mapo *~/.arc*, ki je privzeta mapa za podatke programa NorduGrid ARC

```
mkdir ~/.arc
```

Certifikat SiGNET (datoteka `signet.p12`) pretvorimo v zahtevani format:

```
openssl pkcs12 -in signet.p12 -clcerts -nokeys -out usercert.
    pem
openssl pkcs12 -in signet.p12 -nocerts -nodes -out userkey.pem
```

Spremenimo dovoljenja dobljenih certifikatov:

```
chmod 400 userkey.pem
chmod 644 usercert.pem
```

in jih premaknemo v mapo ~/.arc:

```
mv userkey.pem ~/.arc/
mv usercert.pem ~/.arc/
```

4.2.4 Dostop do strežnika VOMS

Strežnik VOMS skrbi za avtorizacijo uporabnikov, zato je najprej treba vnesti podatke o strežniku. Ustvarimo mapo *~/.arc/vomsdir*, ki bo vsebovala podatke.

```
mkdir ~/.arc/vomsdir
```

Podatke vnesemo v datoteki *~/.arc/vomsdir/voms.sling.si.lsc* in *~/.arc/vomses*.

```
echo -e "/C=SI/O=SiGNET/O=SLING/CN=voms.sling.si\n/C=SI/O=
    SiGNET/CN=SiGNET CA" > ~/.arc/vomsdir/voms.sling.si.lsc
echo -e '"gen.vo.sling.si" "voms.sling.si" "15001" "/C=SI/O=
    SiGNET/O=SLING/CN=voms.sling.si""gen.vo.sling.si"' > ~/.
    arc/vomses
```

Za generiranje proksi certifikata poženemo

```
arcproxy -S gen.vo.sling.si
```

Če je vse pravilno, dobimo izpis:

```
Your identity: /C=SI/O=SiGNET/O=FRI Uni-Lj/CN=Jure Grabnar
Contacting VOMS server (named gen.vo.sling.si): voms.sling.si
    on port: 15001
Proxy generation succeeded
Your proxy is valid until: 2014-07-29 01:23:59
```

Zdaj lahko začnemo uporabljati omrežje grid. Kadar dobimo izpis

```
The VOMS server with the information:
  "gen.vo.sling.si" "voms.sling.si" "15001" "/C=SI/O=SiGNET/O=
    SLING/CN=voms.sling.si" "gen.vo.sling.si" "
can not be reached, please make sure it is available
Proxy generation succeeded
Your proxy is valid until: 2014-07-29 01:26:16
```

pomeni, da so se certifikati za strežnik iztekli in da je treba pridobiti nove. Za posodobitev certifikatov poženemo

```
sudo apt-get update
sudo fetch-crl
```

Zdaj lahko poženemo zgornji ukaz `arc-proxy`.

V nadaljevanju želimo na grid poslati testni program `mpi_test.c`. Predložimo skripte, ki jih potrebujemo, in navedemo običajno zaporedje ukazov za uporabo omrežja.

4.3 Skripte

Za uporabo omrežja moramo napisati dve skripti:

1. skripto, ki opisuje vire, ki jih želimo od omrežja - uporabljali bomo format xRSL (Extended Resource Specification Language),

2. zagonsko skripto, ki se prva zažene na gruči - z njo prevedemo naš program in ga zaženemo, uporabljali bomo bash.

Za obe skripti podamo primer, ki bo zadostoval za večino programov.

4.3.1 xRSL skripta

Osnovna sintaksa za xRSL je par atribut-vrednost,

```
(atribut="vrednost")
(atribut="vrednost1" "vrednost2") (za atribute z vecimi
    vrednostmi)
```

ki priredi vrednost atributu. Komentarji se začnejo z "(*" in končajo z "*)":

```
(*atribut="vrednost"*)
```

Spodnja skripta xRSL se začne z znakom &, ki označuje konjunkcijo med vsemi atributi.

```
&(atribut1=vrednost1)(atribut2="vrednost2")...
```

To pomeni, da velja atribut1 *IN* atribut2 *IN* ... vsi atributi do konca datoteke. Celotno specifikacijo si lahko ogledamo na [5].

Naslednji primer skripte xRSL zadošča za večino primerov. Posamezni atributi so razloženi v komentarjih.

```
&
(* Dolocimo zagonsko skripto, ki se bo zagnala prva *)
(executable = "mpi_test.sh")

(* Dolocimo datoteke, ki se naj kopirajo na grid,
poti so relativne glede na to, kje zazenemo ukaz
za posiljanje posla arcsub *)
(inputfiles =
```

```
  ("mpi_test.sh" "")
  ("mpi_test.c" "")
)

(* Ime programa; uporabno, ko zaganjamo vec poslov
za lazjo orientacijo, izpise se pri klicu arcstat *)
(jobname=mpi_test)

(* Datoteki za standardni izhod in standardni izhod za
napake *)
(stdout=mpi_test.out)
(stderr=mpi_test.err)

(* Argumenti, ki jih podamo programu (ki jih dobimo
preko argv tabele v C-ju *)
(arguments= "1 100 30 20")

(* Stevilo zahtevanih procesorjev. Privzeto 64. *)
(count=8)

(* Koliko casa naj bodo rezultati hranjeni, ce jih ne poberem?
    *)
(lifeTime="2 days")

(* Okolje v katerem se program izvaja:
APPS/ARNES/MPI-1.7 uporablja MPI 1.7.3
APPS/ARNES/MPI-1.6 uporablja MPI 1.6.4 *)
(runTimeEnvironment = "APPS/ARNES/MPI-1.7")

(* Ime direktorija, kamor se shranijo grid log datoteke.
Prenesejo se ob klicu ukaza arcget. Uporabno, kadar se kaj
zalomi pri sprejemanju posla. *)
(gmlog=gridlog)
```

Datoteki za `stderr` in `stdin` se bosta prenesli, ko prenesemo izpeljan posel z ukazom `arcget`.

Velja omeniti, da je lahko postopek, preden bo na vrsto prišel naš posel, če zahtevamo veliko število procesorjev, dolgotrajen (za privzetih 64 procesorjev lahko

traja tudi več dni).

4.3.2 Zagonska skripta

Zagonska skripa je preprosta skripta bash, ki prevede program in ga požene.

```
#!/bin/bash

PROGRAM="mpi_test"
ARGUMENTI="$@" # Argumenti, ki jih dobimo iz xRSL (atribut "
    arguments")

# 1. Prevajanje
mpicc $PROGRAM.c -o $PROGRAM

#---------------------------------------
# Izpis spremenljivk, da vidimo, kaj predstavljajo

# $MPIRUN se izbere glede na katero MPI razlicico
# smo izbrali v xRSL runTimeEnvironment atributu
echo "MPIRUN:""$MPIRUN"

# $MPIARGS doloca stevilo procesov preko -np parametrov,
# dolocen je na podlagi count atributa v xRSL
echo "MPIARGS:""$MPIARGS"
#---------------------------------------

# 2. Zagon programa
$MPIRUN $MPIARGS $PROGRAM $ARGUMENTI

# 3. Koncno stanje programa
exitcode=$?
echo Program je koncal s kodo $exitcode.

exit $exitcode
```

Prevedemo ga z `mpicc` in poženemo z `mpirun`. Grid podpira več okolij, v katerem se program lahko izvaja. Lokacija skripte `mpirun` se zato skriva v okoljski spremenljivki `$MPIRUN`.

Spremenljivka `$MPIARGS` pove, na koliko procesorjih naj se program izvede glede na atribut `count` iz skripte xRSL. Kadar želimo program izvajati na enem procesorju, se lahko pojavi hrošč, kjer `$MPIARGS` ne dobi pravilne vrednosti in sistem zavrne posel. V takem primeru je najlažje zamenjati vrstico za zagon programa v:

```
$MPIRUN -np 1 $PROGRAM $ARGUMENTI
```

Ne smemo pa pozabiti niti spremeniti atributa `count` v skripti xRSL na 1.

4.3.3 Uporaba omrežja grid

Skripte imamo pripravljene, zdaj je treba program poslati na grid, počakati, da se izvede, in prenesti rezultate. Preden začnemo uporabljati grid, bomo po vsej verjetnosti morali osvežiti proksi certifikat z ukazom:

```
arcproxy --voms=gen.vo.sling.si:/gen.vo.sling.si
```

Veljavnost takega certifikata je privzeto 12 ur. Veljavnost lahko preverimo z ukazom:

```
arcproxy -I
```

Z ukazom

```
arcproxy -c validityPeriod=24h
```

pa lahko spremenimo privzeti čas veljavnosti. Posel lahko pošljemo v izvajanje z ukazom `arcsub`

```
arcsub -c jost.arnes.si -o ids mpi_test.xrsl
```

Stikalo `-c` pove, na kateri gruči želimo posel izvajati, s stikalom `-o` pa določimo datoteko, kamor naj se shrani ID posla. Na podlagi te datoteke lahko potem bolj organizirano pridobivamo rezultate, npr. vse posle s štirimi procesorji shranimo v eno datoteko, posle z osmimi procesorji v drugo itd. To ni nujno, ampak zgolj omogoča lažje rokovanje z večjim številom poslov.
Posel potrebuje nekaj minut, da pride v sistem. Ko pride v sistem, lahko preverimo njegov status z ukazom `arcstat`:

```
arcstat id_posla
```

npr.

```
arcstat gsiftp://jost.arnes.si:2811/jobs/
    zA3NDmZGRDlnmmR0Xox1SiGmABFKDmABFKDm7rKKDmABFKDm0jK6Mo
```

ID posla se izpiše po končanem ukazu `arcsub`. Če nas zanimajo vsi posli, lahko uporabimo stikalo `-a`:

```
arcstat -a
```

Pomen statusa naj bi bil dokaj samoumeven, za podrobnejšo sliko si lahko preberemo 4. poglavje na [13]. Če nas zanima, kaj posel trenutno izvaja, lahko uporabimo ukaz `arccat`:

```
arccat id_posla
arccat -e id_posla
```

ki izpiše standardni izhod ali standardni izhod za napake (s stikalom `-e`). Ko se posel konča bodisi s statusom *Finished* bodisi s statusom *Failed*, se lahko odločimo, ali bomo rezultate prenesli, lahko pa jih tudi izbrišemo. Rezultate prenesemo z ukazom `arcget`:

```
arcget id_posla
```

Ukaz `arcget` ustvari nov direktorij in privzeto prekopira datoteki s standardnim izhodom in standardnim izhodom za napake ter direktorij z dnevniškimi datotekami omrežja grid. Prav tako lahko s stikalom `-a` prenesemo vse končane posle. Ob prenosu posla se ta izbriše iz sistema in ne obstaja več. Torej, kadar izbrišemo lokalni direktorij tega posla, ne moremo več dobiti njegovih rezultatov. Kadar pa ne želimo prenesti rezultatov končanega posla, lahko to storimo z ukazom `arcclean`

```
arcclean id_posla
```

ki posel in njegove pripadajoče datoteke pobriše iz sistema. Izvajanje posla lahko tudi predčasno končamo z ukazom `arckill`:

```
arckill id_posla
```

Za pregled naprednejših možnosti lahko uporabimo

```
man <ukaz>
```

ki nam pokaže seznam vseh stikal za izbrani ukaz. Ta množica ukazov je zadostna za normalno uporabo omrežja grid.

4.3.4 Dodatno

Zasedenost omrežja lahko nadzorujemo prek Grid Monitorja, dostopnega na [7]. Tu lahko vidimo aktivne uporabnike, kateri posli se izvajajo itd.

Kadar se nam pri uporabi omrežja kaj zalomi in ne vemo več, kako naprej, lahko kontaktiramo podporo za uporabnike na elektronsko pošto support@sling.si.

Poglavje 5

Implementacija

V tem poglavju smo predstavili implementacijo programa s knjižnico MPI (ang. Message Passing Interface). Sekvenčno implementacijo metode MCTS je spisal asistent Tom Vodopivec pri izdelavi svojega doktorata.

Program MCTS v osnovi ni bil pisan za večprocesorske sisteme. Del programa, ki bi ga želeli paralelizirati, se kliče iz več mest v programu, poleg tega je nek klic odvisen od prejšnjih klicev in naključnega generatorja. Zato smo želeli, da glavni proces skrbi za potek programa, medtem ko drugi procesi izključno računajo. S tem se izognemo sinhronizaciji vseh vmesnih rezultatov in po potrebi sinhroniziramo samo končno vrednost, ki jo izračuna glavni proces in je zadostna za nadaljnje računanje. Tako prihranimo komunikacijo med procesi. Uporabili smo paralelizacijo listov (poglavje 2.1.1) - vsi procesi začnejo pri praznem drevesu, rezultati za korensko vozlišče pa se združijo po končani paralelizaciji.

5.1 Gomoku

Gomoku ali 5 v vrsto se igra na plošči Go, veliki 19 x 19 polj, s črnimi in belimi figurami. Igralca izmenično postavljata svoje figure na ploščo. Zmaga tisti, ki mu prvemu uspe postaviti pet figur v vrsto (horizontalno, vertikalno ali diagonalno). Naš algoritem MCTS igra na plošči, veliki 7 x 7 polj.

5.2 Zgradba ukazov

Ker lahko iz glavnega procesa do delovnih procesov pošiljamo več števil, ki lahko imajo različen pomen, smo razvili preprost sistem komunikacije. Poleg poslanega podatka zraven pošljemo tudi ukaz, ki pove procesu, kaj naj stori s podatkom. Ukaz in podatek sta lahko različnega tipa, zato ju ne moremo poslati kot eno polje z več elementi. V MPI lahko za ta namen definiramo lasten tip.

```
1  struct s_update_params {
2      int command;
3      int selected_action;
4      double dw;
5  };
6
7  void init_struct {
8      int num_items = 3;
9      int blocklengths[3] = {1, 1, 1};
10     MPI_Datatype types[3] = {MPI_INT, MPI_INT, MPI_DOUBLE};
11     MPI_Aint offsets[3];
12
13     offsets[0] = offsetof(s_update_params, command);
14     offsets[1] = offsetof(s_update_params, selected_action);
15     offsets[2] = offsetof(s_update_params, dw);
16
17     MPI_Type_create_struct(num_items, blocklengths, offsets,
       types, &mpi_update_params_type);
18     MPI_Type_commit(&mpi_update_params_type);
19 }
```

Pomen posameznih spremenljivk:

- `num_items` - število blokov (med seboj neodvisnih spremenljivk) v tipu,
- `blocklengths` - število elementov v vsakem bloku,
- `types` - podatkovni tip vsakega bloka,
- `offsets` - odmiki posameznih blokov v naši strukturi `s_update_params`, ki jih izračunamo v vrsticah 13-15.

V 17. vrstici ustvarimo MPI podatkovni tip. Nato ga moramo registrirati v 18. vrstici, preden ga lahko uporabimo v komunikaciji med procesi. Komunikacijo med procesi lahko zdaj izvajamo s pošiljanjem strukture `s_update_params`. Element `command` je ukaz, ki lahko zavzame eno izmed sledečih vrednosti:

- `BCAST_EXIT`: ob prejemu tega ukaza se proces konča. Preostala dva elementa v strukturi sta nedefinirana.
- `BCAST_NUM_GAMES`: ob prejemu tega ukaza dobi proces število iger v elementu `selected_action` in nato začne izvajati paralelno funkcijo z danim številom iger. Element `dw` je nedefiniran.
- `BCAST_WEIGHTS`: ukaz služi posodobitvi vmesnih parametrov. Potrebne parametre dobi v elementih `selected_action` in `dw`.

5.3 Izvajanje

Z glavnim procesom označujemo proces, ki polega tega, da računa, skrbi za potek programa. Kadar potrebuje druge procese, jim sporoči parametre s klicem (blokirajoč) `MPI_BCast()`, prek katerega v svoji zanki čakajo tudi delovni procesi. Ko se prenehajo izvajati, se ponovno vrnejo v čakanje prek `MPI_BCast()`, vse dokler ne dobijo ukaza, ki vsebuje kodo `BCAST_EXIT`. Ta način je zelo podoben paralelizaciji z nitmi.

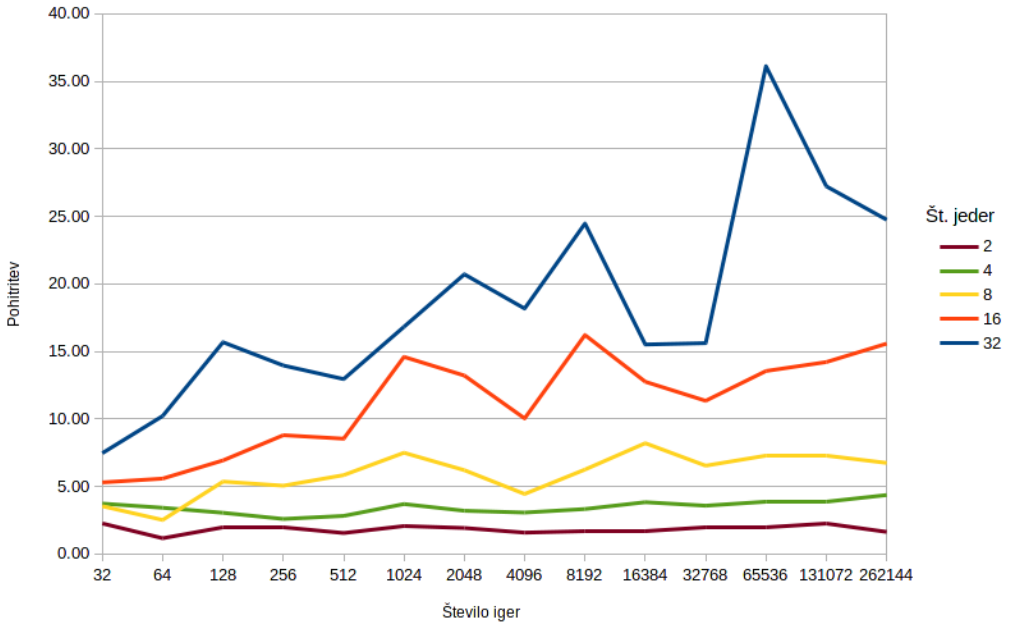
Metoda `Evaluate_Players()`, ki smo jo želeli paralelizirati, vrača rezultat v tabeli, ki je dolga toliko, kot je število igralcev. Vsak indeks predstavlja izid posameznega igralca, ki je odstoten (0 % pomeni, da je igralec zgubil v vseh ponovitvah, 100 %, da je zmagal v vseh). Vsak proces vsebuje delne rezultate in ker so izidi aditivni, jih lahko seštejemo s klicem `MPI_Reduce()`. Dobljeni rezultat nato delimo s številom procesov, da dobimo končni izid. Test, ki smo ga poganjali, začne s praznim drevesom in se izvede enkrat, zato sinhronizacija drevesa ni potrebna. Zanima nas le, kako se algoritem obnaša z različnimi parametri.

Poglavje 6

Meritve in rezultati

Program smo poganjali na različnem številu jeder za različno število iger in merili čas izvajanja. Parameter UCT C je bil fiksiran na 0,2. Za vsako potezo smo izvedli tisoč ponovitev. Algoritem MCTS smo testirali na igri Gomoku (plošča je bila velika 7 x 7 polj). Vsak test smo ponovili desetkrat in nato povprečili rezultat. Vsakega izmed testov smo pognali večkrat, da smo zagotovili, da se izvaja na isti oz. podobni arhitekturi procesorjev kot drugi testi. V nasprotnem primeru se lahko zgodi, da je čas testa z večjim številom jeder počasnejši od testa z manjšim številom jeder. Izvajalne parametre na gruči jost.arnes.si smo pustili privzete (uporabljali smo skripte, prikazane v poglavju 4.3), spreminjali smo le število jeder. Vsi posli so se izvajali na isti gruči.

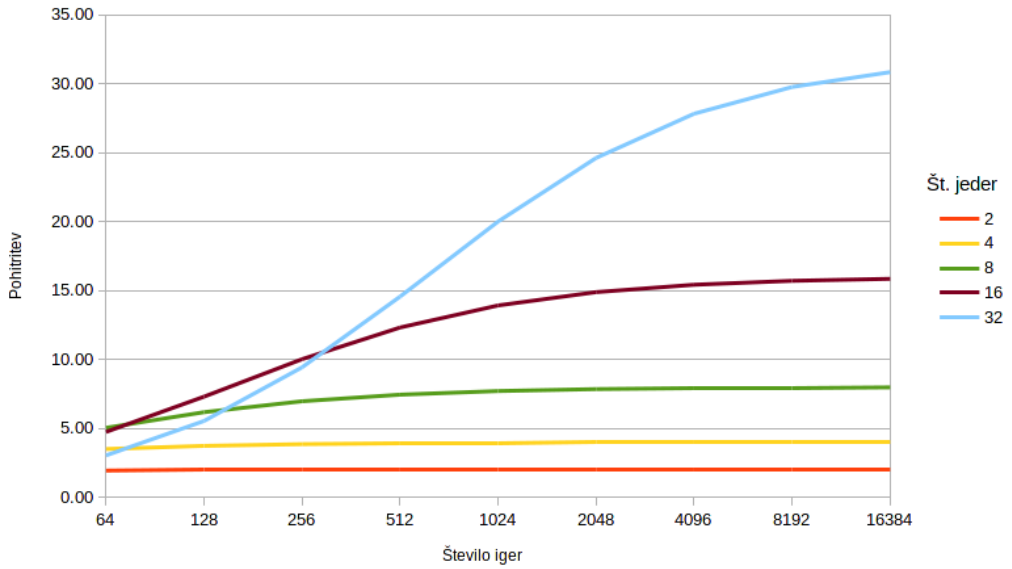
Slika 6.1 prikazuje graf pohitritve v odvisnosti od števila iger za različno število jeder. Število iger je vedno deljivo s številom jeder, zato vsako jedro dobi enako število iger. Pohitritev smo izračunali na podlagi časov, pridobljenih za eno jedro. Idealna pohitritev je enaka številu jeder, npr. za osem jeder bi bila idealna pohitritev osemkratna. Vidimo, da je pohitritev pri majhnem številu iger (32, 64) majhna. Računskega dela za vsako posamezno jedro je pri tem malo in večino časa program porabi za svojo inicializacijo in komunikacijo med jedri. Ko postaja program večji, se s tem daljša čas, potreben za vsako jedro, in čas, ki se porablja za komunikacijo, postane zanemarljiv. Večje število jeder pomeni tudi daljši čas, potreben za komunikacijo. 32 jeder pri 32 igrah nam prinese le sedemkratno pohitritev (idealna bi bila 32), medtem ko pri dveh jedrih dobimo skoraj dvakratno pohitritev.



Slika 6.1: Izmerjeni graf pohitritve v odvisnosti od števila iger za različno število jeder

Zanimalo nas je, pri kolikšnem številu iger postane komunikacija zanemarljiva. Z grafa lahko sklepamo, da se ta meja giba pri okoli 8.192 igrah. Pri tem številu iger ima 16 jeder že 16-kratno pohitritev in 32 jeder okoli 25-kratno. Pri majhnem številu jeder se idealna pohitritev doseže že mnogo prej (za dve jedri je že dovolj 32 iger). Velja omeniti, da je treba pohitritve jemati z rezervo. Težava je namreč v tem, da se program za 32 jeder izvaja na drugačnih procesorjih kot tisti s 16 jedri. Poleg tega se lahko program z istim številom jeder izvaja na drugačnih procesorjih vsakič, ko ga poženemo. Ta anomalija je vidna skoraj pri vsaki krivulji, še najbolj pa se opazi pri 32 jedrih. Pri 8.192 igrah je pohitritev 25-kratna, pri 16.348 in 32.768 igrah pa pade na 16-kratno pohitritev. Sklepamo lahko, da se je program pri 16.384 in 32.768 igrah izvajal na šibkejših procesorjih v primerjavi s tistimi pri 8.192 igrah. Lahko se je zgodilo, da se je program izvajal na dveh procesorjih po 16 jeder, namesto na enem z 32 jedri (kar lahko še dodatno poveča čas komunikacije, poleg tega so jedra lahko še šibkejša). Na drugi strani pa lahko

vidimo skoraj 36-kratno pohitritev pri 65.536 igrah. Ta superlinearna pohitritev je verjetno posledica močnejšega posameznega jedra pri 32 jedrih v primerjavi s tistim pri enem jedru (od katerega računamo pohitritev). Sklepamo lahko, da so se programi z 32 jedri večinoma izvajali na taki arhitekturi, ki omogoča največ 25-kratno pohitritev (na podlagi števila iger pri 8.192 in 262.144). Obrnili smo se na podporo SLING, da bi izvedeli, ali se da vsak program izvajati na isti arhitekturi. Dobili smo skop odgovor, da je mogoče, vendar je bil postopek nejasen. V upanju, da bi vse programe izvajali na isti arhitekturi, smo jih pognali večkrat, vendar nam pri nekaterih parametrih to ni uspelo. Da bi se izognili temu, bi teoretično lahko naredili en sam program, ki bi rezerviral 32 jeder, in naredili vse meritve v tem programu. Zaradi predolgega izvajanja bi sistem za upravljanje poslov tak program predčasno končal (izvajanje programov za $2 * 262144 = 524288$ iger že preseže dovoljeni čas).



Slika 6.2: Teoretičen graf pohitritve v odvisnosti od števila iger za različno število jeder

Na Sliki 6.2 smo izdelali teoretičen graf za meritve na Sliki 6.1, ki pove, kakšne rezultate smo pričakovali. Generirali smo ga na podlagi enačbe 6.1,

$$t = C \cdot t_{init} + \frac{N \cdot t_{game}}{C} + N \cdot C \cdot t_{overhead} \tag{6.1}$$

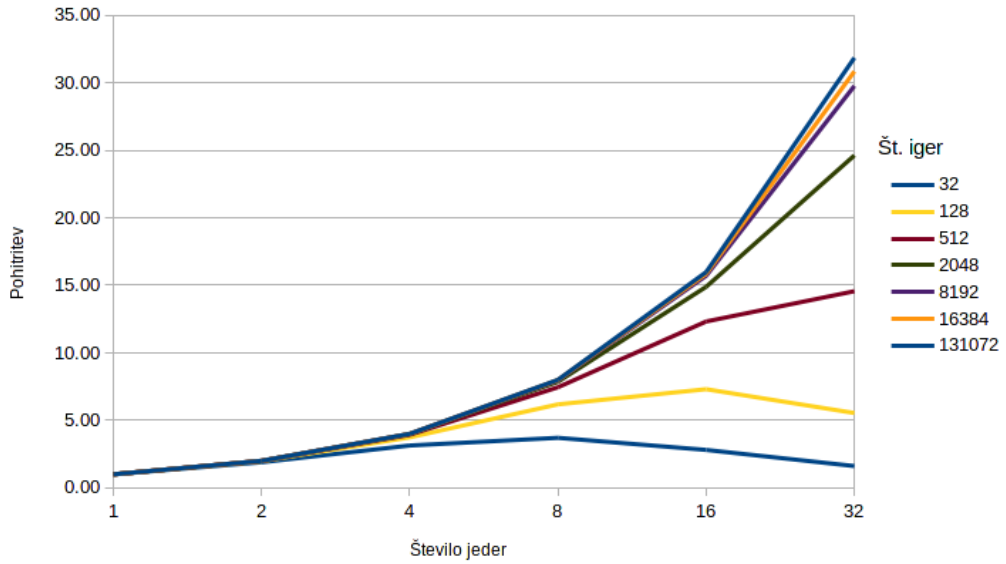
kjer je:

- C - število jeder,
- N - število iger,
- t_{init} - čas, potreben za inicializacijo enega jedra (v našem primeru 5),
- t_{game} - čas, potreben za simulacijo ene igre (v našem primeru 10),
- $t_{overhead}$ - čas, potreben za inicializacijo ene igre na enem jedru (v našem primeru 1).

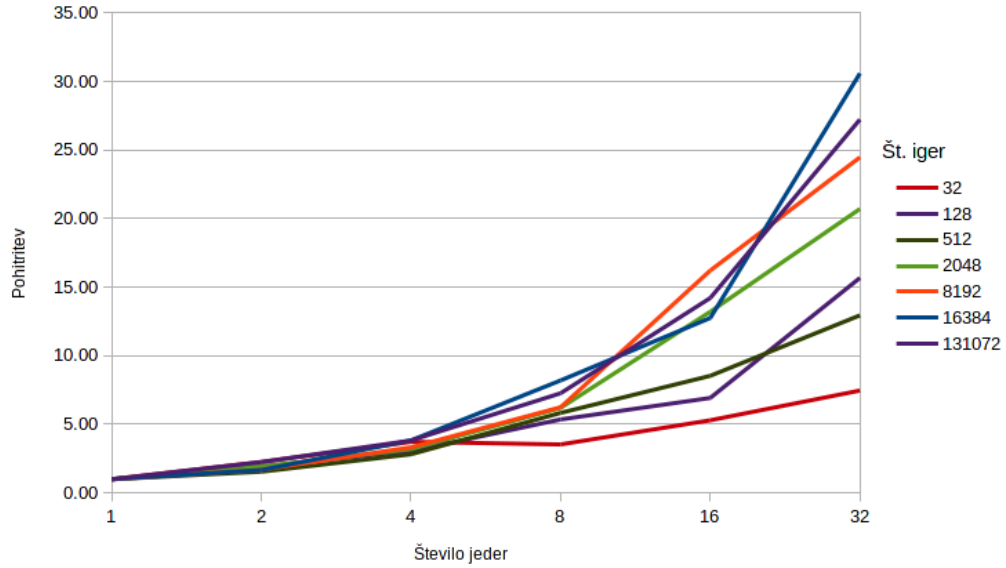
Parametre t_{init}, t_{game}, $t_{overhead}$ smo določili empirično, glede na naša pričakovanja. Če izvzamemo nihanja zaradi različnih arhitektur, sta si grafa podobna. Na teoretičnem grafu smo bili malo bolj pesimistični pri majhnem številu iger in velikem številu jeder - pričakovali smo, da se bo 32 jeder obneslo slabše kot osem jeder zaradi komunikacije. Krivulje v neki točki doesežejo skoraj linearno pohitritev in od takrat naprej je konstantna - komunikacija postane zanemarljiva.

To dodatno potrdi Slika 6.3, ki prikazuje obrnjen teoretičen graf. Nekatere krivulje so izpuščene zaradi preglednosti. Tu lahko vidimo optimalno število jeder, ki jih potebujemo za neko število iger. Če bi vzeli 16 procesorjev za 32 iger, bi večina jeder ostala neizkoriščenih. V takem primeru bi bilo bolje vzeti dve jedri ali štiri jedra, kjer bi bila vsa izkoriščena.

Slika 6.4, ki prikazuje obrnjen graf prvega izmerjenega grafa, tudi potrdi trditev, da komunikacija postane zanemarljiva pri okoli 8.192 igrah. Pri večjem številu iger krivulje že sovpadajo. Zaradi težav z različnimi arhitekturami, bi lahko bila meja že pri 2.048 ali celo 1.024 igrah.



Slika 6.3: Teoretičen graf pohitritve v odvisnosti od števila jeder za različno število iger



Slika 6.4: Izmerjeni graf pohitritve v odvisnosti od števila jeder za različno število iger

6.1 Diskusija

V začetku merjenja smo želeli program testirati tudi na več kot 32 jedrih. SLING sprva ni podpiral več kot 64 jeder, vendar so nam po kontaktiranju podpore omogočili neomejeno število jeder. Izkazalo se je, da je poganjanje na takem številu jeder nestabilno: včasih je delovalo, pogosto ne. Poleg tega smo lahko čakali po več dni, da je tak posel prišel na vrsto (tudi že pri 64 jedrih). V takem primeru se poraja vprašanje uporabnosti: zakaj bi čakali več dni na 64 jeder, če vemo, da bo manjše število jeder prišlo na vrsto prej in opravilo isto delo, še preden posel s 64 jedri pride na vrsto? Da bi 64 jeder ali več prišlo do izraza, bi potrebovali veliko večji problem - za primerjavo, eno jedro potrebuje približno 35 ur za 262.144 iger. Za optimalno razmerje med časom, potrebnim, da posel pride na vrsto, in časom izvajanja bi lahko naš problem razdelili na manjše podprobleme in vsakega izmed teh podproblemov izvajali na enem jedru. Rezultate bi nato združili sami in paralelizacija algoritma z MPI sploh ne bi bila potrebna. Če s pomočjo Gridmonitorja (4.3.4) sledimo navadam drugih uporabnikov omrežja, ugotovimo, da večina uporabnikov uporablja ravno to metodo. Posel z enim jedrom pride na vrsto zelo hitro (ponavadi že nekaj minut po tem, ko pride v sistem), zato je ta metoda zelo učinkovita.

Algoritem smo želeli testirati tudi na pet tisoč ponovitvah na potezo (potrebno je veliko več pomnilnika), vendar so bili rezultati slabi pri večjem številu iger. Tisoč ponovitev na potezo je dalo zadovoljive rezultate.

Poglavje 7

Sklep

V delu smo se spoznali z drevesno metodo Monte Carlo. Imeli smo sekvenčni algoritem, ki smo ga paralelizirali z MPI-knjižnico. Paralelni algoritem smo poganjali na omrežju grid, ki v Sloveniji dela v okviru SLING-a. Ta različne organizacije v Sloveniji združuje v razpršen superračunalnik. Predstavili smo našo implementacijo paralelnega algoritma in rezultate. Problem ima veliko stopnjo paralelnosti. Večinoma smo dosegli linearno pohitritev, nekje malo manjšo, ponekod pa celo superlinearno pohitritev. Seznanili smo se s slabostmi omrežja grid za merjenje rezultatov - različne arhitekture procesorjev. Za rešitev tega problema smo predlagali en velik program, ki bi ga pognali na 32 jedrih in izmerili rezultate za vse mogoče kombinacije parametrov (različno število iger, različno število jeder). Tako bi si zagotovili isto arhitekturo pri vsakem merjenju časa. V okviru dela so nastala tudi navodila za uporabo omrežja grid.

Paralelni algoritem omogoča računanje na večjem številu iger. Večje število iger pomeni večjo zanesljivost in natančnost algoritma, saj je metoda Monte Carlo, ki je del MCTS, v osnovi stohastična metoda. Paralelizirali smo osnovno metodo, ki ne potrebuje informacije od drugih procesov. Obstaja tudi naprednejša metoda, kjer je treba sinhronizirati stanja med posameznimi potezami. V tem primeru postane komunikacija veliko pomembnejši faktor kot pri osnovni metodi. Treba bi bilo paralelizirati napredno metodo in ugotoviti, kako učinkovita je paralelizacija. Če bi se metoda MCTS uporabljala v tekmovalne namene (računalnik proti računalniku), bi bilo treba razviti tudi sistem, ki bi prekinil izvajanje MCTS na vseh jedrih po pretečenem času - pri tekmovanjih smo ponavadi omejeni s časom.

Metoda MCTS je zanimiva nova metoda na področju umetne inteligence in upamo, da bo z novimi dognanji raziskovalcev postala še bolj uporabna tudi na drugih področjih.

Literatura

[1] Cream. Dosegljivo na naslovu: http://grid.pd.infn.it/cream/. [Dostop 31.3.2015].

[2] dcache. Dosegljivo na naslovu: http://www.dcache.org/. [Dostop 31.3.2015].

[3] Egi in numbers. Dosegljivo na naslovu: http://www.egi.eu/infrastructure/operations/egi_in_numbers/index.html. [Dostop 31.3.2015].

[4] European grid infrastructure. Dosegljivo na naslovu: http://www.egi.eu/. [Dostop 31.3.2015].

[5] Extended Resource Specification Language. Dosegljivo na naslovu: http://www.nordugrid.org/documents/xrsl.pdf. [Dostop 31.3.2015].

[6] glite - lightweight middleware for grid computing. Dosegljivo na naslovu: http://grid-deployment.web.cern.ch/grid-deployment/glite-web/. [Dostop 31.3.2015].

[7] Grid monitor. Dosegljivo na naslovu: http://www.sling.si/gridmonitor/loadmon.php. [Dostop 31.3.2015].

[8] Grid v Sloveniji. Dosegljivo na naslovu: http://www.arnes.si/storitve/omrezne-storitve/grid.html. [Dostop 31.3.2015].

[9] NorduGrid. Dosegljivo na naslovu: http://www.nordugrid.org/. [Dostop 31.3.2015].

[10] NorduGrid Downloads - Repository Information for 13.11. Dosegljivo na naslovu: `http://download.nordugrid.org/repos-13.11.html`. [Dostop 31.3.2015].

[11] SiGNET. Dosegljivo na naslovu: `https://signet-ca.ijs.si:443`. [Dostop 31.3.2015].

[12] Slovenska iniciativa za grid. Dosegljivo na naslovu: `http://www.sling.si/sling/`. [Dostop 31.3.2015].

[13] The NorduGrid grid manager and GridFTP server. Dosegljivo na naslovu: `http://www.nordugrid.org/documents/GM.pdf`. [Dostop 31.3.2015].

[14] Uporabniški vmesniki do omrežja grid. Dosegljivo na naslovu: `http://www.sling.si/sling/uporabniki/uporabniski-vmesniki/`. [Dostop 31.3.2015].

[15] User's guide to signet ca certificates. Dosegljivo na naslovu: `http://signet-ca.ijs.si/guide.html`. [Dostop 31.3.2015].

[16] VOMS SLING. Dosegljivo na naslovu: `https://voms.sling.si:8443/voms/gen.vo.sling.si/`. [Dostop 31.3.2015].

[17] B. Arneson, R.B. Hayward, and P. Henderson. Monte carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):251–258, Dec 2010.

[18] T. Cazenave and N. Jouandeau. On the parallelization of uct. *Proceedings of CGW07*, pages 93–101, 2007.

[19] G. Chaslot. *Monte-carlo tree search*. PhD thesis, PhD thesis, Maastricht University, 2010.

[20] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*. The AAAI Press, 2008.

[21] G. Chaslot, M. Winands, and H.J. van den Herik. Parallel monte-carlo tree search. In H.J. van den Herik, X. Xu, Z. Ma, and M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg, 2008.

[22] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, pages 72–83, 2006.

[23] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.