Párhuzamos genetikus algoritmus

Szimuláció

Készítette:
Eperjesi Alfréd
epaeaat.elte

1. Bevezetés

A GRID rendszerek megjelenésével lehetővé vált a személyi számítógépek, a szuperszámítógépek, a számítógépes laborok, a helyi hálózatok vagy akár a klaszterek összefogása. Egyéni erejüket kihasználva, reális időn belül végezhetőek bonyolult, végtelennek tűnő számítások. A technológia ereje a párhuzamosságban rejlik. A rendszer szimulálására talán a legalkalmasabb egy párhuzamos környezet, egy klaszter. A rendszer erejének bemutatására, egy erre alkalmas algoritmust, a *genetikus algoritmust* választottam.

2. Célkitűzés

A cél, a mesterséges intelligencia területén gyakran alkalmazott *evolúciós,* vagy *genetikus algoritmus* párhuzamos megvalósíthatóságának vizsgálata és egy alkalmas módszerrel való elkészítése és elemzése. A dokumentum további részeiben bemutatásra kerül az algoritmus általánosan, majd a párhuzamosítási lehetőségei. Ezután az algoritmust implementációja következik, C nyelven a PVM könyvtár segítségével. Végül véletlenszerűsített populációkkal az algoritmus tesztelésére kerül sor.

3. Genetikus algoritmus

A genetikus algoritmusok a nem módosítható szélsőérték keresési módszerek közé tartoznak. Nem adnak pontos eredményt, mert valószínűségi alapokon nyugszanak, de jól alkalmazhatóak nagy terekben, függvények globális szélsőértékeinek keresésére. Egy adott pillanatban nem egyetlen választ, hanem lehetséges válaszok (egyedek) halmazát (populáció) tartják nyílván, amely elemeit minden lépésben megpróbálják egyre „jobbakra” cserélni.

Az alapalgoritmus:

```
Genetikus_Algoritmus()
     p <- kezdeti populáció
     while terminálási feltétel nem igaz loop
          p1 <- szelekció(p)
          p2 <- rekombináció( p1 )
          p3 <- mutáció(p2)
          p  <- visszahelyezés(p, p’’’)
     endloop
end Genetikus_Algoritmus;
```

Az algoritmus működése:

Kezdetben egy véletlen populációt állít elő, majd lépésenként a következő operátorokat hajtja végre a populáción:

- szelekció: a szülő egyedek kiválasztása
- rekombináció: a szülők keresztezéséből utódok előállítása
- mutáció: az utódok kismértékű változtatása
- visszahelyezés: az utódokat tartalmazó új populáció létrehozása

Kódolás

Fontos az egyedek megfelelő reprezentációja, az úgynevezett *kódolás*. A *kódolás* során egy-egyedet jelsorozattal írunk le. A jelekkel (gének) vagy jelek csoportjával az egyednek különböző tulajdonságai adhatók meg (pl. az első pozíció a szem szín, a második és a harmadik a magasság…). A jelnek a jelsorozatban elfoglalt pozíciója határozza meg, mely tulajdonságnak felel meg. A gének alkotják a kromoszómákat, amik egy-egyedet reprezentálnak Az eljárás kromoszómák egy halmazát használja, amit populációnak nevezünk.

Fittnesz vagy rátermettségi függvény

Ez a módszer egyetlen nem újrahasznosítható része. Speciálisan az adott problémára vonatkozik. Ez az optimalizálási függvény, amely az adott kromoszóma „jóságát" határozza meg. A fittnesz függvényt minden egyes kromoszómára, minden egyes iterációban ki kell számolni.

Szelekció

A szülő egyedek kiválasztása. A kiválasztott kromoszómákból jön létre az utód, így a szülő jelen lesz a következő populációban is. A kiválasztás a *fittnesz függvény* alapján történik.

Módszerek:

- rulett kerék algoritmus: a rátermettségi függvényre vagy annak skálázására épül
- rangsorolásos: rátermettség alapján a kisebb értékűeknek nagyobb esélyt ad
- versengő: véletlenül választott egyedcsoportok legjobb egyedét választja
- csonkolásos: a rátermettség szerint legjobbak közül választ néhányat

Rekombináció

Szülő egyedekből utódok létrehozása. A szülők tulajdonságait öröklik az utódok. A leggyakoribb rekombinációs módszer a *keresztezés.* A keresztezés csak szerkezeti átrendezést jelent. Sokféle keresztezési eljárás ismert.

Mutáció

Az utódokon kismértékű változtatás végrehajtása. A mutációra kiválasztott bitet (gént) negálja. Az operátor valószínűségét ezrelék nagyságrendűre ajánlott venni.

4. Párhuzamos genetikus algoritmus

Mivel az algoritmus egyedek halmazán dolgozik, ezért lehetséges a halmazok részhalmazokra való felosztása és párhuzamos feldolgozása, továbbá a párhuzamosság lehetőséget ad esetleges különböző *rátermettségi függvények* párhuzamos alkalmazására is a populáción, felgyorsítva így már a kísérleti fázist is.

A párhuzamosításra többféle modell állítható fel. Ezek közül az úgynevezett *Sziget modell (Island model)* kerül megvalósításra.

Sziget modell

Több populáció fejlődik egyszerre párhuzamosan, elszigetelve egymástól. Egy előre meghatározott szabályrendszer szerint, minden egyes populációból a legrátermettebb egyedek vándorolnak a szigetek között. Így akár lehetőség van különböző operátorok alkalmazására az egyes szigeteken.

Lehetséges operátor stratégiák:

Szelekció:
- Rulett-kerék szabály: vegyük a szülő populáció összrátermettségét, majd osszuk el vele az egyes kromoszómák egyéni fittnesz értékét, ekkor 0 és 1 közé eső számokat kapunk, majd véve egy véletlen számot 0 és 1 között, a hozzá legközelebb eső egyed nyeri a kiválasztást
- A populáció összes tagja közül véletlenszerűen kiválasztott egyedet választja
- Az első n legrátermettebb közül választ véletlenszerűen

Rekombináció:
- A keresztezési valószínűség különböző az egyes szigeteken
- Szomszédos egyedek keresztezése
- Több mint kettő egyedet keresztez

Mutáció:
- A mutációs valószínűség különböző az egyes szigeteken
- Különböző eloszlású véletlenítés

Lehetséges egyedvándorlási stratégiák
- Minden kiválasztott emigráns minden szigetre eljut
- Mester-szolga: egy kitüntetett szerepű sziget megkapja az összes szigettől az emigránsokat, majd egy rangsorolás után (másodlagos szelekció) a kiválasztottakat eljuttatja az összes populációba
- A populációkat egy topológiába rendezzük és csak a szomszédok, cserélik el egymással a kiválasztott egyedeket (pl. lánc, rács topológia)

Lehetséges egyed elhelyezési stratégiák:
- Előre lefoglalt helyre kerülnek
- Az n leggyengébb egyedet cseréljük le
- Hasonló egyedek helyére kerülnek
- Véletlenszerű helyre kerülnek

Lehetséges leállási feltételek:
- Korlátot megadása az iteráció számra
- Rátermettségi arány átlagos változását figyelembe véve, ha kevésbé változott, akkor leállás

A modell lehetséges megvalósítása
1. Egyedreprezentáció, kezdeti populáció előállítása és N részre partícionálása, szomszédsági kapcsolatok definiálása
2. minden egyes populációra a 3. 4. lépés végrehajtása
3. genetikus operátorok végrehajtása
4. szomszédoknak a legrátermettebb egyed elküldése, a szomszédoktól kapott egyedek elhelyezése a populációban
5. ha nem teljesül a leállási feltétel, akkor ugrás a 2. – re

5. Implementáció

Az implementáció a *sziget modell* alapján készül C nyelven a PVM könyvtár alkalmazásával. A program futtatása a Nyelvi labor (nylab) gépein történik, ahol lehetőség van a klaszter lehetőséget kihasználva párhuzamos programok futtatására.

Az implementációban a következő szabályok lettek alkalmazva:
Kódolás: megadott hosszúságú kromoszómák, amelyek minden egyes génje egész típusú, 0 és 9 közötti szám
Kezdeti populáció: véletlen 0 és kilenc közötti egész szám generálás
Fittnesz függvény: gén értékek átlagolása
Szelekció: rulett-kerék algoritmus
Keresztezés: a gének második felének a megcserélése
Mutáció: nem lett alkalmazva
Leállási feltétel: felső korlát az iteráció számra
Egyedvándorlás: a legrátermettebb egyed átküldése a következő szigetre
Egyed elhelyezés: a legrosszabb fittnesz értékű egyed lecserélése az újonnan érkezett egyedre

A program működése

A program egy n+1 elemű hálózatot épít fel, ahol n db elem egy gyűrűt alkotva valósítja meg a sziget modellt. A szomszédos elemek egymással kommunikálnak és van egy központi szerepet betöltő n+1. elem, aki felépíti a hálózatot és monitorozza az eredményeket.

A központ működése (1 db)

A központ a program indításakor jön létre. Először megkísérli létrehozni az n db szigetet, ha ez sikerült, akkor elküldi minden szigetnek a megelőző és a következő sziget azonosítóját, továbbá létrehozza a szigethez tartozó kezdeti populációt és elküldi azt. Így létrejön a gyűrű topológia. Ha az inicializálás sikeres volt, akkor egy start üzenettel elindítja a szigeteken a „munkát”, majd várja az eredményeket, és azt kiírja a képernyőre, megvalósítva így a monitorozást.

A szigetek működése (n+1 db)

Minden egyes szigeten önállóan fut *a genetikus algoritmus*. Az algoritmus egyszeri lefutása után a legrátermettebb egyedet a következő szigetre átküldi, majd az előző szigetről vár egy emigráns egyedet, amit a legrosszabb fittnesz értékkel rendelkező egyed helyére helyez. Az átlagos fittnesz értéket minden iteráció végén eltárolja. Ez mindaddig ismétlődik, amíg a leállási feltétel nem teljesül, azaz ebben az esetben, amíg az iterációk száma el nem éri a határt. Végül a tárolt részeredményeket és a végső populációban a legrátermettebb egyedek egyikét elküldi a központnak.

A forráskód

A program két állományból áll:

A központ implementációja: **genetic_algorithm_main.c**

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

```
#include <time.h>
#include <math.h>
#include <sys/timeb.h>
#include "pvm3.h"

#define msg_init  1
#define msg_start  2
#define msg_finish 3

int main(int argc, char* argv[])
{
    // argumentumok vizsgalata
    if( argc == 5 )
    {
        //inicializálás hálózat felépítése
        // valtozo deklaraciok
        int i, j, msec,k;
        int populationNumber = atoi(argv[1]); //populációk száma     int
        islandNumber = atoi(argv[2]);       //szigetek száma
        int chromosomeLength = atoi(argv[3]); //kromoszóma hossza
        int iterationNumber = atoi(argv[4]);  //iterációk száma
        int *islandIds=malloc(sizeof(int)*populationNumber*
        chromosomeLength);

        // valtozok az idomeresehez
        struct timeb start;
        struct timeb stop;
        double time, *results;
        (double*)results = malloc(sizeof(double) * iterationNumber);

        //az n db taszk elinditasa
        char *args[] = {argv[1], argv[2], argv[3], argv[4],  (char*)0};
        if( pvm_spawn("genetic_algorithm_child", args, PvmTaskDefault,
        NULL, islandNumber, islandIds)<islandNumber )
        {
            pvm_perror("spawn : Nem sikerult az n db taszk indítása.");
            return 1;
        }
        // szomszéd taszk ID-k és a kezdeti populáció elküldése
        for(i=0;i<islandNumber;++i)
        {
            pvm_initsend(PvmDataDefault);
            if( i == 0 ) pvm_pkint(&islandIds[islandNumber-1],1 ,1);
            else pvm_pkint(&islandIds[i-1],1 ,1);
            if( i == islandNumber-1 ) pvm_pkint(&islandIds[0],1 ,1);
            else pvm_pkint(&islandIds[i+1],1 ,1);
            for( j=0;j<populationNumber;++j )
            {
                  for(k=0;k<chromosomeLength;++k)
                  {
                        int l = (int)((10.0*rand())/(RAND_MAX*1.0));
                        pvm_pkint(&l,1 ,1);
                  }
            }
            pvm_send(islandIds[i], msg_init);
      }
      //inicializálás vége
      ftime(&start);

      // várakozás az eredményre a csatornából
      k=0;
```

```
      for(j=0;j<islandNumber;j++)
      {
            pvm_recv(islandIds[j], msg_finish);
            pvm_upkdouble(results, iterationNumber, 1);
            printf("\nEz innen ( %d ):\n ",islandIds[j]);
            for( i = 0;i<iterationNumber;++i)
            {
                  printf(" %d: %.3f\t",i,results[i]);
            }
            k++;
      }
      // befejezés
      ftime(&stop);

      msec=stop.time-start.time==0?stop.millitm-start.millitm:1000-
      stop.millitm + start.millitm;
      time=(double)msec/1000+stop.time-start.time;
      printf("\n\n\tSzamolasi ido : %.3f sec\n\n\n",time);
      free(islandIds);
      free(results);
    }
    else
    {
      printf("\n\n\n\tNem megfelelo argumentum szam!!!");
      help(argv[0]);
    }
    printf("\nMain: finished.");
    pvm_exit();
    return 0;
}
```

A sziget implementációja: **genetic_algorithm_child.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "pvm3.h"

#define msg_init  1
#define msg_start  2
#define msg_finish 3
#define msg_data   4
```

A fittnesz függvény kiszámítása minden egyedre:

```
void fittness(int* population, int* populationNumber, int*
chromosomeLength, double *fitnessValues, double* sumFittness)
{
      int i,j;
      sumFittness[0] = 0.0;
      for(i = 0; i<*populationNumber; ++i )
      {
            int tmp = 0;
            for(j = 0; j < *chromosomeLength; ++j )
            {
                  tmp += population[i*(*chromosomeLength)+j];
            }
            fitnessValues[i] = (1.0*tmp)/((*chromosomeLength)*1.0);
```

```
            sumFittness[0] +=  fitnessValues[i];
      }
}
```

Segédfüggvény a kiválasztáshoz.

```
void check(int* selected, int* k, double* r, double* fittnessValues)
{
      int i, maxIndex = 0;
      double d = fabs(fittnessValues[*k]-(*r));
      double maxValue = 0.0;
      int l = 0;
      double e;
      for(i = 0; i<8; ++i)
      {
            e = fabs(fittnessValues[selected[i]]-(*r));
            if( ( l == 0 ) &&
                  ( d < e ) )
            {
                  l = 1;
            }
            if( maxValue < e )
            {
                  maxValue = e;
                  maxIndex = i;
            }
      }
      if( l )
      {
            selected[maxIndex] = *k;
      }
}
```

Kiválasztás. Rulett-kerék algoritmussal nyolc egyed kiválasztása.

```
void selection(int* selected, int* populationNumber, double*
fittnessValues, double* sumFittness)
{
      int i;
      int k = 0;
      double t = (1.0*rand()/(RAND_MAX*1.0));
      for( i = 0; i<*populationNumber; ++i)
      {
            fittnessValues[i] = fittnessValues[i]/(sumFittness[0]);
            if( k < 8 )
            {
                  selected[k] = i;
                  k++;
            }
            else check(selected, &i, &t, fittnessValues);
      }
}
```

A keresztezés. A kiválsaztott egyedek második felének megcserélése.

```
void crossover(int* population, int* selected, int* chromosomeLength)
{
      int i,j;
      int* tmp;
      int half = (*chromosomeLength)/2;
      for( i = 0; i < 4; ++i )
```

```
	{
		for( j = 0; j < half; ++j )
		{
			tmp = population[selected[i*2]+half+j];
			population[selected[i*2]+half+j]=
			population[selected[i*2+1]+half+j];
			population[selected[i*2+1]+half+j] = tmp;
		}
	}
}
```

A legrátermettebb egyed megkeresése.

```
int findMostFittness(double* fittnessValues, int* populationNumber)
{
	double maxValue = fittnessValues[0];
	int maxIndex = 0;
	int i;
	for( i = 1; i<*populationNumber; ++i)
	{
		if( maxValue < fittnessValues[i] )
		{
			maxValue = fittnessValues[i];
			maxIndex = i;
		}
	}
	return maxIndex;
}
```

A leggyengébb fittnesz függvénnyel rendelkező egyed kiválasztása.

```
int findSmallestFittness(double* fittnessValues, int* populationNumber)
{
	double minValue = fittnessValues[0];
	int minIndex = 0;
	int i;
	for( i = 1; i<*populationNumber; ++i)
	{
		if( minValue > fittnessValues[i] )
		{
			minValue = fittnessValues[i];
			minIndex = i;
		}
	}
	return minIndex;
}
```

A főprogram.

```
int main(int argc, char* argv[])
{
    if( argc == 5 )
    {
        // valtozo deklaraciok
	int parentTid, nextIslandId, previousIslandId, minIndex, i, j;
	int populationNumber = atoi(argv[1]); //populációk száma
	int chromosomeLength = atoi(argv[3]); //kromoszóma hossza
	int iterationNumber = atoi(argv[4]);
	int myTid = pvm_mytid();
	int *population, *selected, maxIndex;
```

```
double *fittnessValues, *results, *sumFittness;

// memoria lefoglalasa az input adatok fuggvenyeben

//a populáció
(int*)population = malloc(sizeof(int) * populationNumber *
chromosomeLength);
//a kiválasztott egyedek tárolója
(int*)selected = malloc(sizeof(int) * populationNumber);
//a fittnesz értékek
(double*)fittnessValues = malloc(sizeof(double) * populationNumber);
//részeredmények tárolója
(double*)results = malloc(sizeof(double) * iterationNumber);
//összfittnesz értékek
(double*)sumFittness = malloc(sizeof(double) * 1);

parentTid = pvm_parent();

pvm_recv(parentTid, msg_init);
//a központtól kapott adatok feldolgozásasa
pvm_upkint(&previousIslandId, 1, 1);
pvm_upkint(&nextIslandId, 1, 1);

//kezdeti populáció
for( i=0;i<populationNumber;++i )
{
      for( j = 0; j < chromosomeLength; ++j )
      {
            int p;
            pvm_upkint(&p,1 ,1);
            population[i*(chromosomeLength)+j] = p;
      }
}

//a genetikus algoritmus
int counter = 0;
while( counter < iterationNumber )
{
      fittness(population, &populationNumber, &chromosomeLength,
      fittnessValues, sumFittness);
      selection(selected, &populationNumber, fittnessValues,
      sumFittness);
      crossover(population, selected, &chromosomeLength);

      //kiválasztott kromoszóma elküldése a következő szigetre
      maxIndex =
      findMostFittness(fittnessValues,&populationNumber);
      pvm_initsend(PvmDataDefault);
      for(i = 0;i < chromosomeLength; ++i)
      {
      pvm_pkint(&population[maxIndex*chromosomeLength+i],1,1);
      }
      pvm_send(nextIslandId, msg_data);
      //a kromoszóma fogadása az előző szigetről
      minIndex = findSmallestFittness(fittnessValues,
      &populationNumber);
      int a;
      pvm_recv(previousIslandId, msg_data);
      for(i = 0;i < chromosomeLength; ++i)
      {
            pvm_upkint(&a, 1, 1);
```

```
                    population[minIndex*chromosomeLength+i] = a;
                }
                double result;
                result = (sumFittness[0]*(1.0))/(1.0*populationNumber);
                results[counter] = result;
                counter++;
        }
        //Eredmények elküldése a központba
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(results, iterationNumber, 1);
        pvm_send(parentTid, msg_finish);

        free(population);
        free(fittnessValues);
        free(results);
    }
    else
    {
        printf("\n\n\n\tNem megfelelo argumentum szam!!!");
        help(argv[0]);
    }
    pvm_exit();
    return 0;
}
```

A program futtatása

A programot a pvm környezetből a következő parancs kiadásával lehet elindítani:

```
spawn -> genetic_algorithm_main popNum islandNum chrLength itNum [fileName]
```

, ahol:
popNum: a populációk száma
islandNum: a szigetek száma
chrLength: kromoszómák hossza
itNum: iterációk száma
fileName: opcionális, az eredményeket tartalmazó file neve

A program kimenete

A program a képernyőre és egy file – ba is kiírja a részeredményeket és a végeredményeket is.

Kiadott parancs:

```
spawn -> genetic_algorith_main 32 1 10 50 file_32_1_10_50
```

Az output file tartalma:

0 4.731
1 4.856
2 4.962
3 5.062
4 5.162
5 5.228
6 5.225
7 5.319

8 5.409
9 5.491
10 5.569
…

Ahol az első oszlop a populáció számát, a második oszlop pedig az adott sorszámú populáció átlagos rátermettségét jelenti. A fenti példa egy sziget első 10 populációjának a részeredményeit mutatja.

A képernyőre kerül az egyes szigetek nyertesei és a számolási idő:

Legjobb: 9599989864
…
Szamolasi ido: 12.451 sec

Itt az látható, hogy az adott szigeten a 9599989864 kromoszóma nyert és a program futási ideje 12.451 másodpercig tartott(ez az összes sziget idejét jelenti).

6. Tesztelés

A tesztelési eredmények vizualizálásához a *gnuplot* programot használtam. Annak érdekében, hogy a tesztelés látványos legyen egy lassító függvény került a *genetic_algorithm_child.c* forrásba:

```
void delay()
{
    int i, j, k = 2;
      for(i=0;i<10000;i++)
      for(j=0;j<10000;j++)
      {
            k = k*1;
      }
}
```

A *delay()* függvény minden egyes iteráció végén meghívásra kerül, lassítva ezzel a működést.

A teszteléskor 20 számítógép állt rendelkezésre a klaszterben. Ezért a maximális szigetszámot 16 – nak vettem.

Futási idő

Először az kerül vizsgálatra, hogy a futási idő konstans-e, ha elegendő processzor áll a rendelkezésre, azaz ha minden egyes sziget egyszerre tud dolgozni külön-külön.

Bemenő adatok:
populációk száma: 32
kromoszóma hossza: 10
iteráció szám: 50

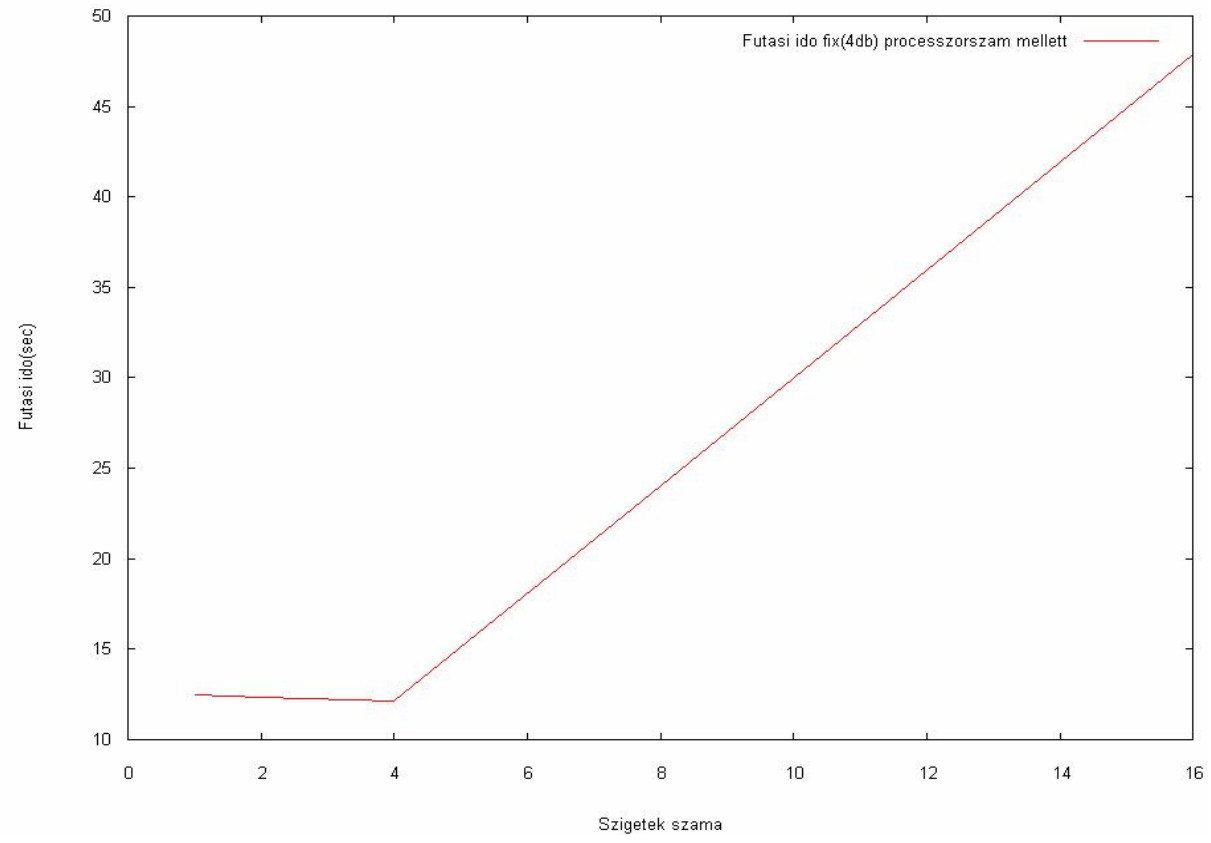
Az eredmények a következők lettek:

Szigetek száma	Futási eredmény
1	12.451
2	12.328
4	12.112
8	11.810
16	12.027

Tehát ha rendelkezésre áll a megfelelő számú processzor, akkor a futási idő konstans, ha figyelmen kívül hagyjuk az esetleges kommunikációs költségekből fakadó némi eltérést. Ebben az esetben az mondható, hogy a bemenő adatok 12 másodperc alatt lettek feldolgozva.

Következő vizsgálat célja, hogy hogyan változik a futási idő ha nincs elegendő processzor.
Bemenő adatok:
populációk száma: 32
kromoszóma hossza: 10
iteráció szám: 50
processzor szám: 4

Szigetek száma	Futási idő
1	12.451
2	12.328
4	12.112
8	24.034
16	47.879



Megfigyelhető, hogy a futási idő lineárisan nő. Az is látszik az ábrán, hogy ha rendelkezésre áll megfelelő számú (szigetek száma) processzor, akkor a futási idő közel konstans (1-4).

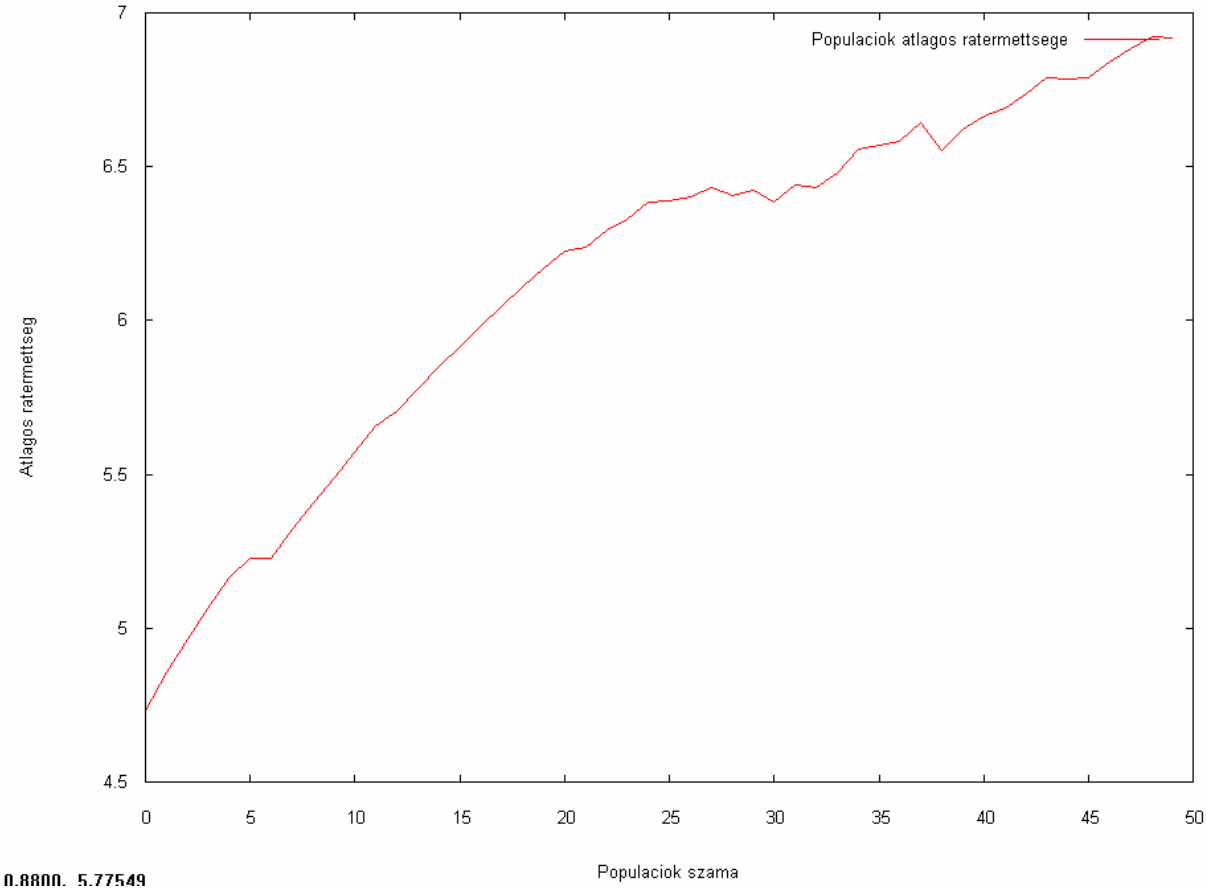
Eredmény

Az algoritmus működésének, eredményességének a vizsgálata. A továbbiakban a szigetszámnak megfelelő processzor szám rendelkezésre áll. Az ábrákon minden egyes piros vonal egy szigeten történő változást szemléltet.

Szekvenciális eset:

Bemenő adatok:
populációk száma: 32
sziget szám: 1
kromoszóma hossza: 10
iteráció szám: 50

Eredmény: 9599989864

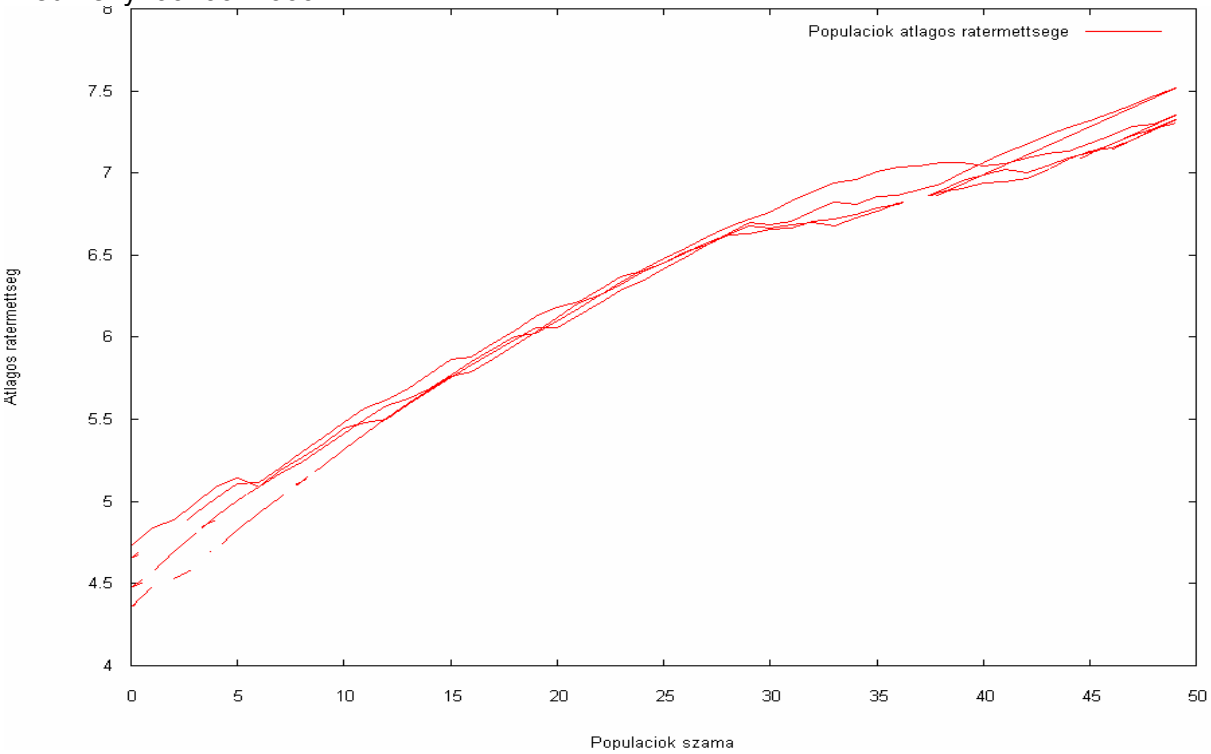


0.8800, 5.77549

Nem szekvenciális esetek:

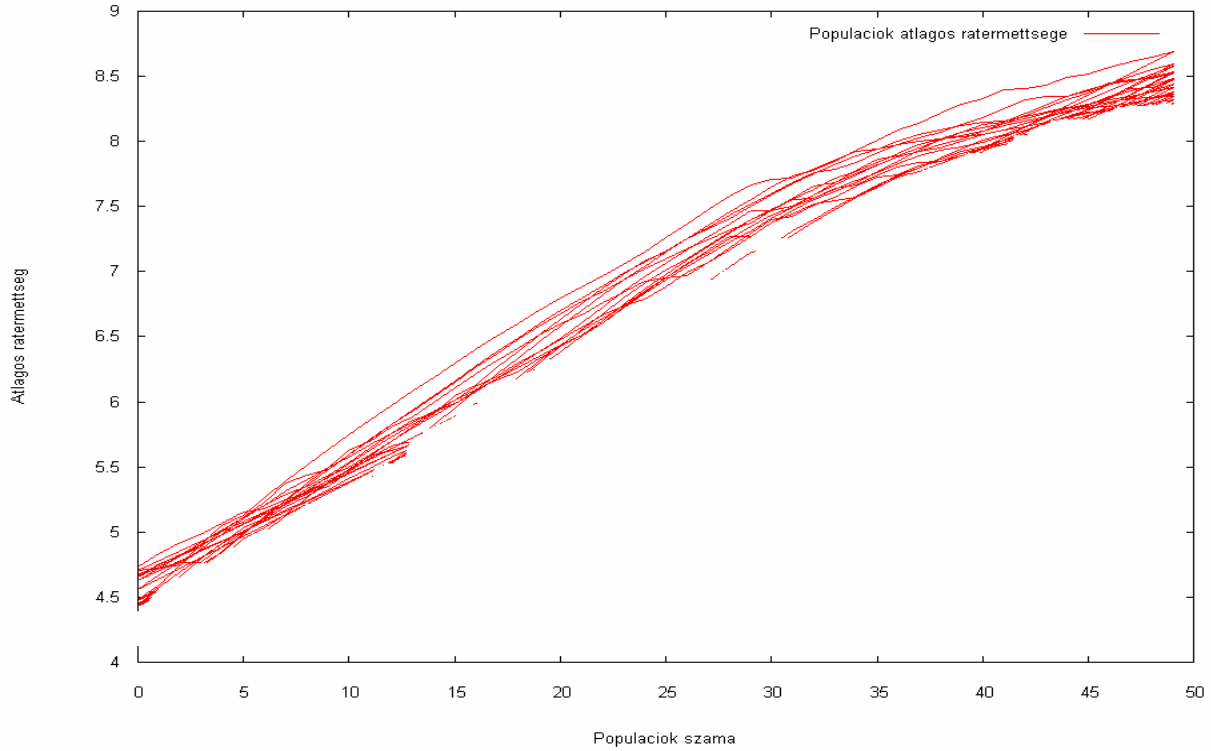
Bemenő adatok:
populációk száma: 32
sziget szám: 4
kromoszóma hossza: 10
iteráció szám: 50

Eredmény: 9979977999



Bemenő adatok:
populációk száma: 32
sziget szám: 16
kromoszóma hossza: 10
iteráció szám: 50

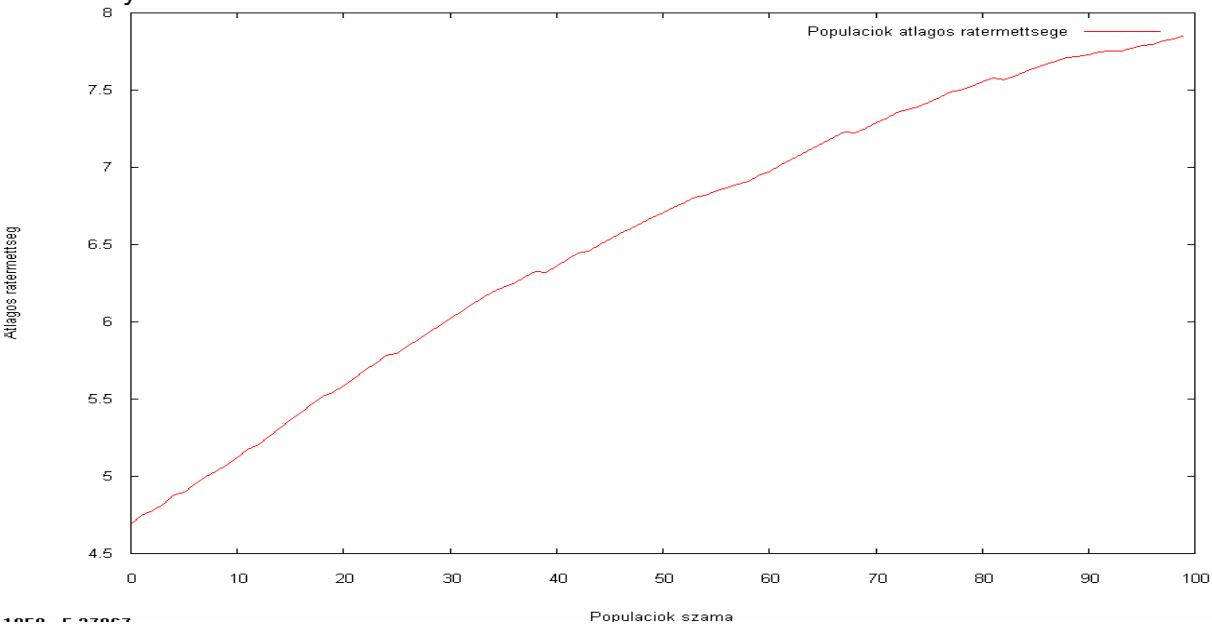
Eredmény: 9999999999



Növelve a populáció számát és az iteráció számot:

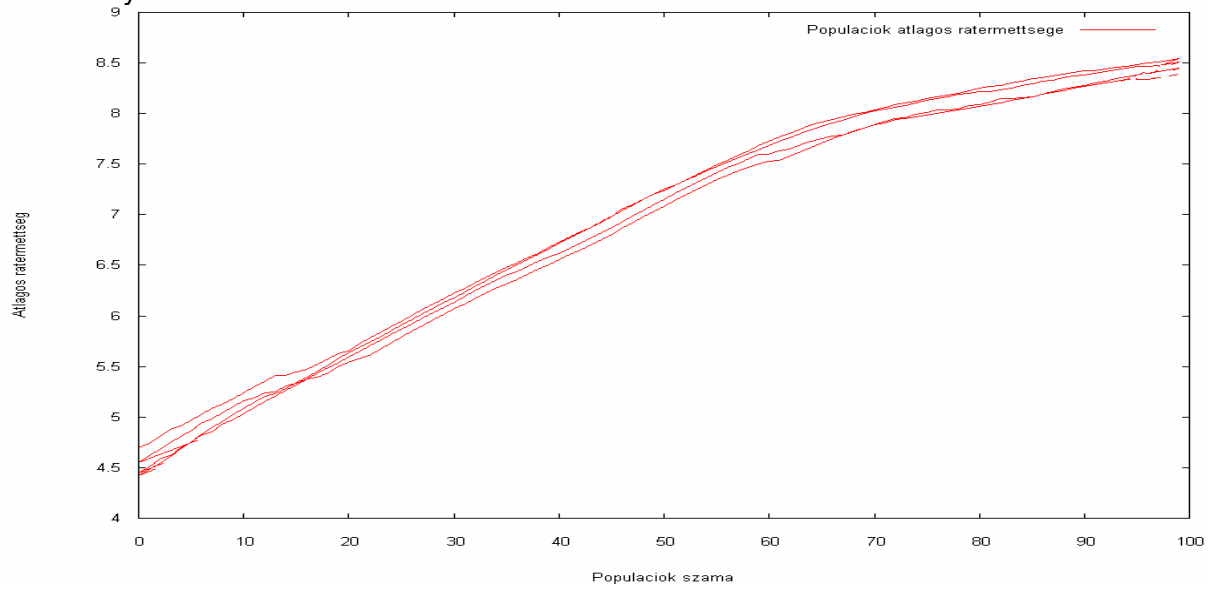
Szekvenciális eset:
Bemenő adatok:
populációk száma: 64
sziget szám: 1
kromoszóma hossza: 10
iteráció szám: 100

Eredmény: 5989999699



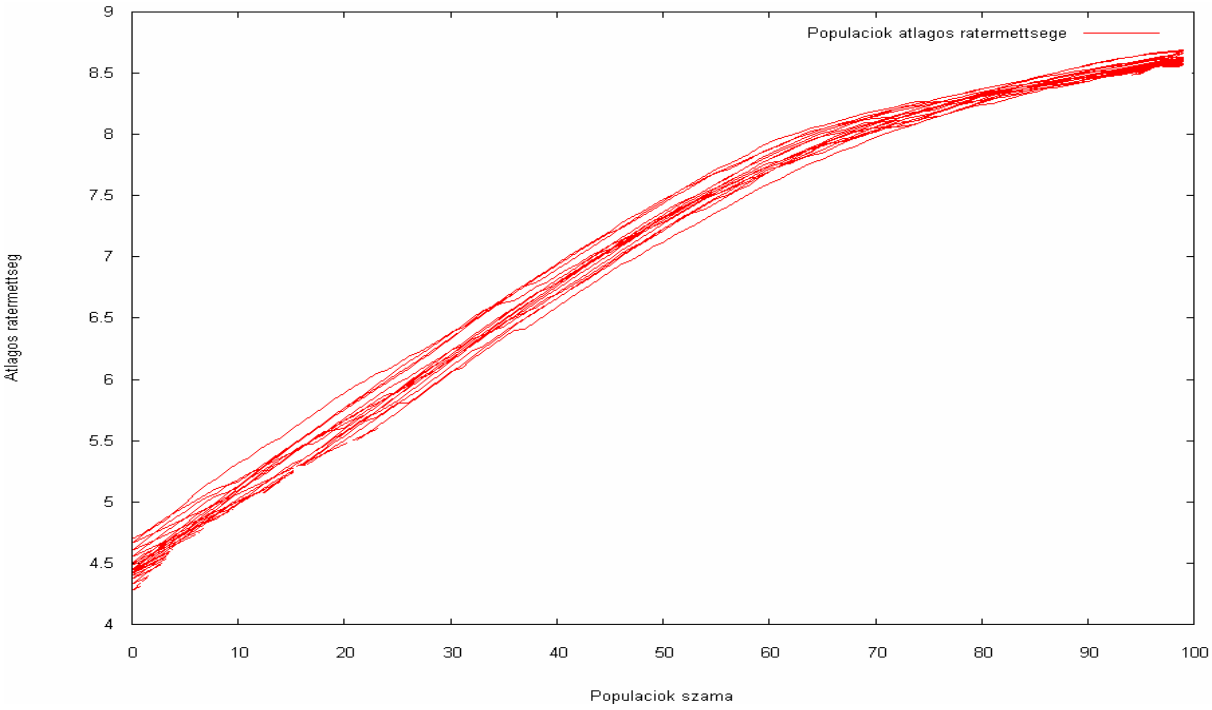
Nem szekvenciális eset:
Bemenő adatok:
populációk száma: 64
sziget szám: 4
kromoszóma hossza: 10
iteráció szám: 100

Eredmény: 9899999999



Bemenő adatok:
populációk száma: 64
sziget szám: 16
kromoszóma hossza: 10
iteráció szám: 100

Eredmény: 9999999999



Az ábrákon jól látszik a párhuzamosságból fakadó nem determinisztikusság. Látható az iteráció szám és a sziget szám növelésével egyre pontosabb eredményt kapunk, ez az átlagos rátermettség végső értékéből következik. Az is jól látható, hogy a párhuzamosság hatására sokkal pontosabb eredmény kapható ugyanannyi idő alatt (a futási időre vonatkozó eredmények miatt). A bemenő adatokra mindkét esetben a 16 szigeten történő számolás már pontos eredményt adott, ugyanis itt a legnagyobb fittnesz érték a 9999999999 – es kromoszómához tartozik.

7. Befejezés

Összességében elmondható, hogy az algoritmust érdemes párhuzamosítani. A GRID technológia lehetővé teszi különböző architektúrák, operációs rendszerek, nézőpontok együttműködését, kiterjesztve így a párhuzamosságot egy magasabb dimenzióba. A bemutatott algoritmus is bizonyítja, hogy az eddig emberi mértékben véve végtelennek tűnő számítások is egy olyan szintre redukálhatók, amik már korántsem tűnnek megoldhatatlannak.

A program elérhető a következő címről:

http://people.inf.elte.hu/fred/elte/ga/genetic_algorithm.zip