FAKULTETA ZA ELEKTROTEHNIKO RAČUNALNIŠTVO IN INFORMATIKO

Paralelno procesiranje

poročilo

Študent: Uroš Golob

Vpisna št.: 93578256

E-mail: golob.uros@gmail.com

GSM: 031494355

Mentor: Izr. prof. Janez Brest

Maribor, 20. april, 2010

KAZALO

1. Uvod
2. 1. parallel presort quick sort
3. 2. parallel recursive quick sort
4. 3. parallel merge quick sort
5. Strojna in programska oprema
6. Rezultati preizkusa
7. Zaključek

Uvod

Skupina smeri logike in sistemov je majhna zato smo predmet paralelno procesiranje izvajali projektno. Naloga projekta je zajemala implementacijo paralelnega algoritma za sortiranje podatkov, z uporabo MPI (message passing interface) programskega vmesnika.

Implementiral sem tri različne paralelne izvedbe algoritma quick sort, ter le te preizkusil z različno velikimi polji podatkov naključnih števil, ter tudi z različnim številom procesov.

Bistvo paralelnega procesiranja je v tem, da si niti, procesi, procesorji ali računalniki delijo delo ter ga tako izvajajo hkrati vzporedno. Na ta način je delo hitreje končano, kar lahko uporabniku prinese hitrejše rezultate.

En izmed pogosteje uporabljenih načinov deljenja dela je uporaba sistema za pošiljanje sporočil med procesi. Obstaja pa tudi metoda deljenega pomnilnika.

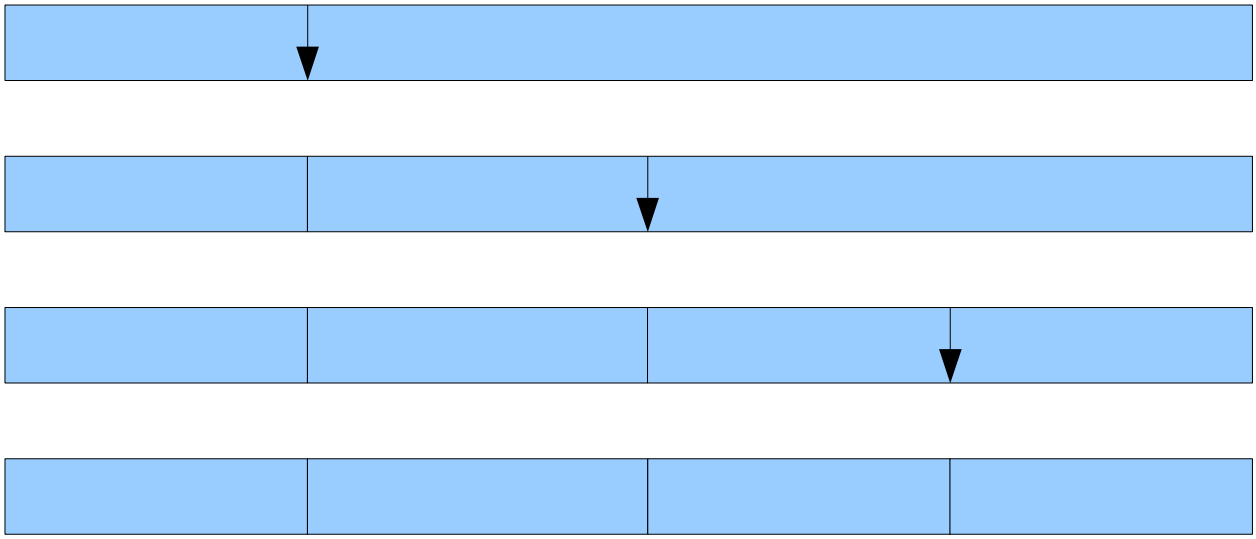
Ker sem želel pridobiti čim več ažurnega znanja sem se odločil uporabiti MPI-2 in ne le MPI-1 ali MPI-1.1. V času nastajanja projekta pa se je že začel razvoj protokolov MPI-2.1, MPI-2.2 ter MPI-3. Nekaj časa sem porabil za iskalne najboljše implementacije tega protokola. Preizkusil sem LAM/MPI, MPICH, OpenMPI ter se na koncu tudi odločil za slednjo implementacijo. Mogoče tudi zato, ker se je v času izbiranja implementacije(knjižnice) uporabljala na takrat najhitrejšem super računalniku. K izbiri je pripomoglo tudi dejstvo, da je openMPI odprtokoden projekt, sam pa sem podprornik odprte kode.

OpenMPI je sicer predviden, da teče na večih enakih serverjih povezanih z hitro infiniband ali iWARP povezavo. Ampak ker je do takih sistemov težko priti, predvsem zaradi njihove visoke cene se lahko alternativno uporabi

navadne delavne postaje, bolje poznane kot osebni računalnik. Čeprav je zelo priporočljivo, da so računalniki enaki jih je mogoče uporabiti tudi če le ti niso. V kolikor ni mogoče uporabiti kake ekstremno hitre optične povezave lahko računalniki komunicirajo preko omrežne elthernet povezave. Je pa priporočljivo, da je le ta 1000BASE-T. Ker žal dostopa do take opreme nisem imel, sem uporabil le tisto, ki sem jo imel na voljo. To pa sta bila 2 osebna računalnika povezana z 100BASE-TX omrežjem. V prihodnosti imam namen v preizkus implementiranih algoritmov vpeljati še več računalnikov.

Parallel presort quick sort

Pri tej izvedbi za vsak proces izberemo pivot in podatke predhodno preuredimo, tako imamo dejansko več različnih delno urejenih delov podatkov. Na ta način lahko podatke sortiramo in preprosto zlepimo skupaj. Slabost te izvedbe je veliko dela za proces z rangom 0 in to predvsem pred samim začetkom sortiranja na ostalih procesih.



Prikaz razdeljevanja podatkov med procese pri parallel presort quick sort algoritmu

- funkcija pred-urejanja, ki poskrbi, da procesi delavci dobijo le podatke manjše od pivota izbranega zanj

```
int presort(int a[], int l, int r, int p)
{
   int i = l, j = r, t;
   while (i <= j) {
      while (a[i] < p)
         i++;
      while (a[j] > p)
```

```
            j--;
        if (i <= j) {
            t = a[i];
            a[i] = a[j];
            a[j] = t;
            i++;
            j--;
        }
    }
    return j;
}
```

- Osnovni proces izvrži izbiro pivota za vsak proces posebej. To naredi tako, da izmed vseh podatkov, ki jih mora sortirati najde najmanjšega ter najvišjega. Potem pa najde vmesne vrednosti, število le teh pa je odvisno od števila procesov.

```
    for (i = 0; i < MAX_ELEMENTS; i++) {
        t = array[i];
        if (t > max) {
            max = t;
        }
        if (t < min) {
            min = t;
        }
    }

    int **pivoti;
    pivoti = malloc((np - 1) * sizeof(int*));
    for (i=0; i < np - 1; i++)
```

```
    pivoti[i] = malloc(4 * sizeof(int));

i = 0;
int stkosov, n;
float kvoc, povp, temp;
stkosov = MAX_ELEMENTS / c;
j = min;
povp=min+max;
for (n = 0 ; n < np-1 ; n++) {
    kvoc= ((float)n+1.0)/(float) np;
    temp= povp * kvoc;
    pivoti[n][0] = (int)temp+1;
}
```

- Delna predurediev sortiranih podatkov se kliče z naslednjo kodo, ki

  poskrbi, da procesi delavci dobijo le podatke v območju vrednosti, ki smo jih izbrali v prejšnem koraku.

```
pivoti[0][1] = presort(array, 0, MAX_ELEMENTS, pivoti[0][0]);
for (i = 1; i < np - 1; i++) {
    pivoti[i][1] = presort(array, pivoti[i-1][1]+1,
MAX_ELEMENTS,pivoti[i][0]);
}
```

- delavcem pošljemo njihove dele podatkov za urejanje,

```
for (i = 1; i < np; i++) {
    if (i==1) {
        pivoti[i-1][2]=pivoti[i-1][1]+1;
        MPI_Send((int *)array, pivoti[i-1][2], MPI_INT,i, 0,
```

```
MPI_COMM_WORLD);
        } else {
            pivoti[i-1][2]=pivoti[i-1][1]-pivoti[i-2][1];
            MPI_Send((int *)(array+pivoti[i-2][1]+1), pivoti[i-1][2],
MPI_INT,i, 0, MPI_COMM_WORLD);
        }
      }
```

- preostanek sortiramo na procesu z 0, za to uporabimo sistemsko funkcijo qsort

```
qsort((int*)(array+pivoti[np-2][1]+1), MAX_ELEMENTS-pivoti[np-2]
[1]-1, sizeof(int), (void*) comp);
```

- sestavimo podatke, ki so jih delavci uredili

```
for (i = 1; i < np; i++) {
    if (i==1) {
        MPI_Recv((int *)array, pivoti[i-1][2], MPI_INT, i,MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    } else {
    MPI_Recv((int *)(array+pivoti[i-2][1]+1), pivoti[i-1][2], MPI_INT,
i,MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
  }
```

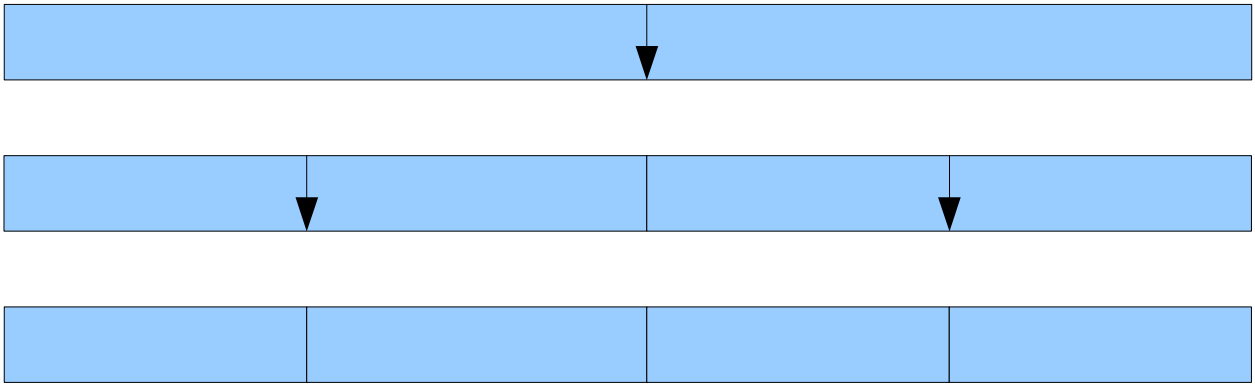
- delavci izvedejo slednjo kodo; z funkcijama mpi_probe ter mpi_get_count pridobimo velikost prihajajočega sporočila. Tako lahko pripravimo prostor za podatke brez, da bi osnovni proces najprej

pošiljal velikost z ločenim sporočilom.

```
MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_INT, &count);
int* rec = (int*) malloc(count * sizeof(int));
MPI_Recv(rec, MAX_ELEMENTS, MPI_INT,
MPI_ANY_SOURCE,MPI_ANY_TAG, MPI_COMM_WORLD, &status);
qsort(rec, count, sizeof(int), (void*) comp);
MPI_Send(rec, count, MPI_INT,0,1, MPI_COMM_WORLD);
free(rec);
```

Parallel recursive quick sort

Vzporedno rekurzivni quick sort poskuša ohraniti samo zasnovo algoritma tako, da ostane še naprej rekurziven in se izkaže za dobrega pri višjem številu procesov. Še posebno če se le ti procesi izvajajo na procesorjih, ki so na seznamu razpoložljivih procesorjev na višjih mestih. Omenjen seznam pa je urejen od zmoglivejših k manj zmoglivejšim.



Prikaz razdelitve podatkov med procese pri parallel recursive quick sort algoritmu

- Tudi pri tej izvedbi uporabljamo funkcijo, ki nam preduredi podatke na dva dela in jih na ta način pripravi na pošiljanje k ostalim procesom.

```
int presort(int a[], int l, int r, int p)
{
   int i = l, j = r, t;
   while (i <= j) {
      while (a[i] < p)
         i++;
      while (a[j] > p)
         j--;
      if (i <= j) {
         t = a[i];
```

```
            a[i] = a[j];
            a[j] = t;
            i++;
            j--;
        }
    }
    return j;
}
```

- Bistvo celotne implementacije tega programa se skriva v funkciji sortiraj rekurzivno

- Prvi del funkcije obsega preprost izračun z katerim ugotovimo ali moremo podatke urediti sami ali jih pošljemo naprej delavcu. Ker je funkcija rekurzivna slej kot prej zmanjka procesov in nekdo mora urediti podatke in jih poslati tistemu, ki jih je poslal njemu

```
int sort_recursive(int* arr, int size, int prank, int maxrank, int rindex)
{
    MPI_Status status;
    int share = prank + pow(2, rindex);
    rindex ++;
    if (share > maxrank) {
        qsort(arr, size, sizeof(int), (void*) comp);
        return 0;
    }
```

- v drugem delu izberemo pivot ter preduredimo podatke.

```
    int pmin,pmax,min = 2147483647, max = -2147483647,i,t;
```

```
for (i = 0; i < size; i++) {
    t = arr[i];
    if (t > max) {
        max = t;
        pmax=i;
    }
    if (t < min) {
        min = t;
        pmax=i;
    }
}

int ostanek = presort(arr, 0, size,(max+min)/2) + 1;
```

- V tretjem delu se odločimo, kateri del pošljemo naprej v urejanje drugim procesom, ter ostanek rekurzivno urejamo naprej, ter prejmemo nazaj urejene podatke.

```
if (ostanek > size - ostanek) {
    MPI_Send((arr + ostanek), size - ostanek, MPI_INT, share, ostanek,
MPI_COMM_WORLD);
    sort_recursive (arr, ostanek, prank, maxrank,    rindex);
    MPI_Recv((arr + ostanek), size - ostanek, MPI_INT,
share,MPI_ANY_TAG, MPI_COMM_WORLD, &status);
} else {
    MPI_Send(arr, ostanek, MPI_INT, share , ostanek,
 MPI_COMM_WORLD);
    sort_recursive ((arr + ostanek), size - ostanek, prank,    maxrank,
rindex);
    MPI_Recv(arr, ostanek, MPI_INT, share, MPI_ANY_TAG,
```

```
 MPI_COMM_WORLD, &status);
   }
}
```

- osnovni le kliče funkcijo rekurzivnega sortiranja

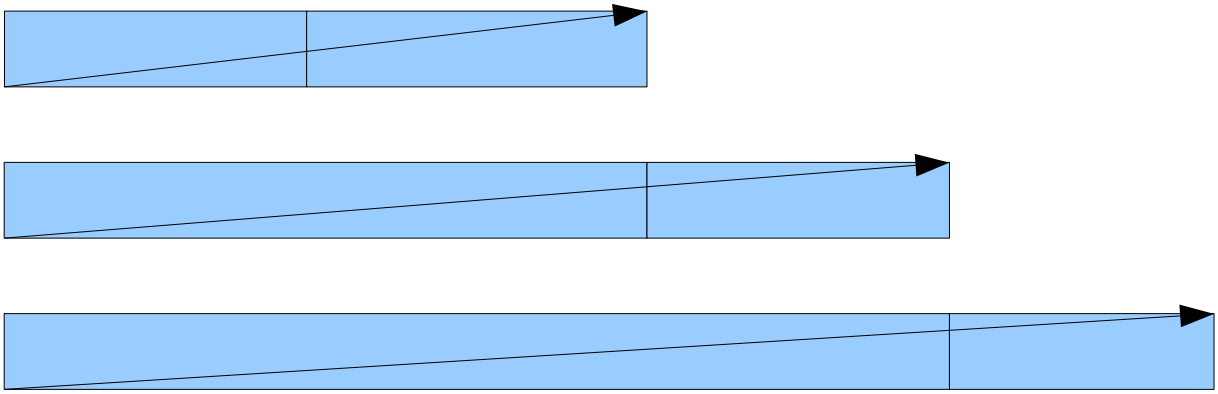
```
sort_recursive(array, MAX_ELEMENTS, myrank, np -1,0);
```

- delavci pa izvedejo slednjo kodo: zavzema izračun indeksa procesa, prejetje podatkov in potem klic funkcije sort_recursive z prijetimi podatki, ko se urejeni podatki vrnejo iz rekuzivne funkcije jih pošljemo nazaj k procesu, ki nam je poslal podatke

```
MPI_Status status;
int* rec = NULL;
int size = 0;
int index_count = 0;
int prsource = 0;
while (pow(2, index_count) <= myrank)index_count ++;
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD,&status);
MPI_Get_count(&status, MPI_INT, &size);
prsource = status.MPI_SOURCE;
rec = (int*)malloc(size * sizeof(int));
MPI_Recv(rec, size, MPI_INT, MPI_ANY_SOURCE,MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
sort_recursive(rec, size, myrank, np -1,index_count);
MPI_Send(rec, size, MPI_INT, prsource,0, MPI_COMM_WORLD);
free(rec);
```

Parallel merge quick sort

Ta izvedba je nekakšen preizkus prve izvedbe saj deluje ravno obratno, podatki se razdelijo med procese tam presortirajo, ob prejemu sortiranih podatkov nazaj pa proces z rangom 0 dokončno pravilno uredi podatke. Z implementacijo tega algoritma želim preveriti ali je bolje sortirati predhodno urejene podatke, ali je bolje združevati že urejene dele podatkov.



Risba prikazuje kako zlivamo podatke pri algoritmu paralle merge quick sort

- funkcija postsort zlije dve predurejeni polji v večje ter dokončno urejeno večje enotno polje

```
int* postsort (int* v1, int n1, int* v2, int n2)
{
    int i, j, k;
    int* result;
    result = (int *)malloc ((n1 + n2) * sizeof (int));
    i = 0;
    j = 0;
    k = 0;
    while (i < n1 && j < n2)
        if (v1[i] < v2[j]) {
            result[k] = v1[i];
```

```
            i++;
            k++;
        } else {
            result[k] = v2[j];
            j++;
            k++;
        }
    if (i == n1)
        while (j < n2) {
            result[k] = v2[j];
            j++;
            k++;
        }
    else
        while (i < n1) {
            result[k] = v1[i];
            i++;
            k++;
        }
    return result;
}
```

- Osnovni proces na začetku pošlje delavcem dele polja

```
int c= MAX_ELEMENTS/np;
for (i = 1; i < np; i++) {//prvi delavec je št 1 in ne 0!!!!!
MPI_Send((int *)(array+((i-1)*c)), c, MPI_INT,i, 0, MPI_COMM_WORLD);
```

- Preostanek števil uredi sam ter jih prestavi v polje z rezultati

```
s=MAX_ELEMENTS-(c*(np-1));
res = malloc(MAX_ELEMENTS * sizeof(int));
rec = malloc(c * sizeof(int));
qsort((int*) array+(c*(np-1)) , s, sizeof(int), (void*) comp);
res=memcpy(res, array+(c*(np-1)), s*sizeof(int));
```

- na koncu še zlije podatke z tistimi, ki so jih uredili delavci

```
for (i = 1; i < np; i++) {
MPI_Recv((int *)rec, c, MPI_INT, i,1, MPI_COMM_WORLD, &status);
res=postsort(res,s,rec,c);
s+=c;}
```

- Delavci izvedejo maslednjo kodo

```
        MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_INT, &count);
        int* rec = (int*) malloc(count * sizeof(int));
        MPI_Recv(rec, MAX_ELEMENTS, MPI_INT, MPI_ANY_SOURCE,0,
    MPI_COMM_WORLD, &status);
        qsort(rec, count, sizeof(int), (void*) comp);
        MPI_Send(rec, count, MPI_INT,0,1, MPI_COMM_WORLD);
        free(rec);
```

Strojna in programska oprema

Strojna oprema je sestavljena po topologiji zvezde. Vse skupaj sem preverjal na 2 računalnikih z nameščenim operacijskim sistemom openSUSE 11.2 z openMPI implementacijo 1.3.2.1-1 z prevajalnikom gcc 4.4.2-4. Povezana pa sta bila z omeržnim stikalom digitus DN5002C/A (100BASE-TX) izkazalo se je, da je to omrežno stikalo rahel ozki vrat v konfiguraciji:

1. z procesorjem intel core duo 2050 (2 ločeni jedri), 2gb polnilnika

   - podatki so bili izvoženi z nfs

2. z procesorjem intel pentium 4 2.8C z omogočeno HyperThreading tehnologijo (2 niti), 1gb polnilnika

   - podatki so bili pripeti z nfs

Posledično sem lahko programe preizkušal z do 4 procesi. Pri mojih računalnikih se je izkazalo, da je procesor intel P4 2.8C na žalost sposoben polne obremenitve le na enem procesesu, saj se je program upočasnil pri uporabi dveh niti.

Rezultati preizkusa

V tabeli se nahajajo časi izvedb programa z različnimi algoritmi, različnim bremenom, ter različno izbiro delavcev. V prvem stoplpcu so navedeni algoritmi v drugem in tretjem pa je navedena izbira algoritmov. V ostalih pa se nahajajo rezultati pognanih testov; V prvi vrstici je navedea velikost polja, ki ga urejamo, v ostalih pa čas samega sortiranja.

Vlogi procesorjev: #1=CD2050@1.6GHz 2#P4 2.8C@2.8GHz

	#1	#2	100000000	10000000	1000000	10000
PPQS	2	2	49.93s	4.73s	0.48s	0.00s
PRQS	2	2	51.02s	4.81s	0.50s	0.02s
PMQS	2	2	49.91s	4.99s	0.46s	0.07s
PPQS	2	1	43.16s	4.21s	0.39s	0.08s
PRQS	2	1	33.70s	3.14s	0.31s	0.00s
PMQS	2	1	43.69s	4.21s	0.40s	0.06s
PPQS	2	0	31.87s	3.01s	0.29s	0.03s
PRQS	2	0	31.63s	2.86s	0.32s	0.03s
PMQS	2	0	31.77s	2.94s	0.26s	0.02s
PPQS	1	1	63.67s	5.95s	0.58s	0.01s
PRQS	1	1	63.63s	6.02s	0.58s	0.01s
PMQS	1	1	63.00s	5.98s	0.56s	0.02s
QS	1	0	57.81s	5.23s	0.46s	0.00s
QS	0	1	39.75s	4.60s	0.41s	0.01s

Maribor, 20. april, 2010

Če pa zamenjamo vloge procesorjev: #1=P4 2.8C 2#CD2050

	#1	#2	100000000	10000000	1000000	10000
PPQS	2	2	56.47s	5.41s	0.51s	0.01s
PRQS	2	2	56.40s	5.33s	0.50s	0.00s
PMQS	2	2	57.99s	5.48s	0.57s	0.01s
PPQS	2	1	51.47s	4.87s	0.44s	0.01s
PRQS	2	1	40.33s	3.75s	0.35s	0.01s
PMQS	2	1	51.50s	4.79s	0.45s	0.01s
PPQS	2	0	42.89s	3.69s	0.34s	0.00s
PRQS	2	0	42.59s	3.75s	0.35s	0.00s
PMQS	2	0	42.83s	3.72s	0.34s	0.00s
PPQS	1	1	68.53s	6.39s	0.62s	0.01s
PRQS	1	1	65.79s	6.44s	0.60s	0.00s
PMQS	1	1	66.79s	6.35s	0.63s	0.01s
QS	1	0	39.66s	4.61s	0.42s	0.00s
QS	0	1	57.93s	5.22s	0.46s	0.00s

Zaključek

Uporaba, preučevanje ter koriščenje možnosti paralelnega procesirana je odličen ter poceni način pohitritve različnih aplikacij.

Z povezovanjem večih računalnikov, z hitrimi omrežnimi ali kakimi drugimi v ta namen vzpostavljenimi linijami in uporabo večih jeder na modernih večjedernih hitrih procesorjih, skrajšamo izvajalni čas.

Z nadalnjim raziskovanjem, bi lahko ugotovili kakšna je obtimalna zasnova za največjo izkoriščenje opreme, ki jo imamo na voljo. Pri dodatni optimizaciji programa pa bi moral upoštevati predvsem količino predpolnilnika vsakega jedra. Seveda pa bi pri optimizaciji morali upoštevati tudi količino drugih nivojev predpomnilnika, ter seveda pomnilnika samega, trgega diska, vodil... Ti podatki bi bili pomembni pri razdeljevanju delov podatkov, ki jih je potrebno sortirati med različne procese. Od hitrosti povezave računalnikov med sabo pa bila odvisna velikost in število sporočil, ki jih pošiljamo med procesi.

Že po krajšem razmisleku lahko ugotovimo, da je zaradi zakasnitev omrežja najoptimalneje, da se število sporočil čim bolj zmanjša. Velikost sporočil pa omeji na MTU v našem komunikacijskem mediju.

Jaz sem na žalost ugotovil, da je na moji opremi uporaba napisanega algoritma nesmiselna, saj nisem imel na voljo kake zelo hitre omrežne opreme. 100BASE-TX ni kos mojemu algoritmu, saj se je najhitreje izvedel na enem samem računalniku z uporabo 2 jeder. Zanimivo mi je bilo dejstvo, da so se podatki sekvenčno hitreje sortirali na starejšem procesorju tipa Pentium 4.

Mogoče pa bi bilo dobro tudi razmisliti o implementaciji sistemske funkcije qsort, ki bi sama ugotovila če je „pametneje“ izvajati klasično sekvenčno

funkcijo qsort ali v primeru prisotnosti večih procesorskih jeder ugotoviti kateri niso preveč zasedeni in jih uporabiti v paralelnem načinu, ki bi mogoče bil na nek način podoben moji drugi implementaciji algoritma. Taka inplementacija qsort funkcije bi vsekakor pohitrila izvajanje večine obstoječih aplikacij, ki uporabljajo to funkcijo.

Med samim razvojem sem neletel na premnoge težave, največja mi je bila razhroščevanja. To težavo sem kasneje rešil z naslednjo funkcijo

```
static void DebugWait(void)

{

    int i = 0;

    char hostname[256];

    gethostname(hostname, sizeof(hostname));

    printf("PID %d on %s ready for attach \n", getpid(), hostname);

    fflush(stdout);

    while (0 == i)

        sleep(5);

    printf("%d: on %s Starting now \n", getpid(), hostname);

}
```

Upam, da bom v prihodnosti še kdaj uspel uporabiti pridobljeno znanje. Dandanes se po večini iščejo le programerji višjih programskih jezikov pa še to po večini le za spletne aplikacije. Sistemskega programiranja ali kakršnegakoli razvoja paralelnih sistemov pa žal še nisem srečal.

Ker dvomim, da bo v bližnji prihodnosti tehnologija bistveno napredovala v smeri hitrosti procesorjev, upam na večje vlaganje in posledično razvoj paralelizacije.

Maribor, 20. april, 2010