Ciurul lui Eratosthenes

Implementare paralela in C folosind MPI

Programare paralela si concurenta, anul I, semestrul I
Master Sisteme Distribuite

Student: Melemciuc Marius-Constantin

Cuprins

1. Introducere

Aceast document prezinta analiza variantelor paralele ale algoritmului de generare al Ciurului lui Eratosthenes, care se refera la identificarea numerelor prime mai mici decat o valoare (data) n.

Varianta secventiala a algoritmului este eficienta si usor de implementat, avand complexitatea O(n log log n). Desi complexitatea este buna, timpul de executie poate fi totusi prea ridicat atunci cand n este foarte mare.

O solutie ce va fi tratata si in acest raport o reprezinta varianta paralela a algoritmului, astfel ducand complexitatea la

X^((n log log n) / p + (sqrt(n) / log (sqrt(n))) * a * log

unde

X = timpul necesar pentru a marca un element ca fiind multiplu de numar prim

n = numarul dat la input

p = numarul de procese

a = timpul de latenta

Pe parcursul acestui document se vor trei prezenta variante de imbunatatire a algoritmului, care vor duce la un timp mediu de executie mult mai bun:

- eliminarea numerelor pare
- eliminarea broadcastului
- rearanjarea loop-urilor.

2. Varianta secventiala

Acest capitol prezinta varianta secventiala a problemei propuse.

2.1 Descriere

Ideea de baza este urmatoarea:

Se tine evidenta tuturor numerelor de la 2 pana la n intr-un array.

Se itereaza de la valoarea 2 si se urmareste daca numarul curent este marcat sau nu:

- daca nu este marcat semnifica faptul ca numarul curent este prim, si se marcheaza toti multiplii acelui numa
- daca este marcat, inseamna ca numarul curent nu este prim si se continua procesul cu urmatoarea valoare.

Pseudocod:

```
ciur(n)
|       count = 0;
|       for i = 2, n, i = i + 1
|       |_      a[i] = true;
|       for i = 2, n, 1
|       |       if (a[i] == true)
|       |       |       count = count + 1;
|       |       |       for j = i * i, n, j = j + i
|       |_      |_      |_      a[j] = false;
|_      return count;
```

In varianta prezentata mai sus, se returneaza numarul elementelor prime mai mici sau egale cu valoarea n. Pozitiile din array-ul a a caror in cadrul carora elementele au valoarea true sunt numere prime.

2.2 Complexitate

Complexitatea medie a algoritmului este O(n log log n).

3. Varianta paralela

3.1 Descriere

In implementarea paralela impartim array-ul in n - 1 elemente si asociem un task fiecarui grup de elemente din array.

Vom folosi impartirea array-ului in blocuri de elemente - se imparte vectorul in p blocuri de elemente de lungime cat mai apropiata.

exemplu: primul element controlat de procesul i este i * n / p

ultimul element controlat de procesul i este (i + 1) * n / p - 1

In implementarea algoritmului, avem acces la elementele vectorului controlat de un anumit proces folosind urmatoarele functii macro, folosite prin directivele de preprocesare (directive *define*):

```
#define BLOCK_LOW(id, p, n) ((id) * (n) / (p) / BLOCK_STEP)
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id) + 1, p, n) - 1)
#define BLOCK_SIZE(id, p, n) (BLOCK_LOW((id) + 1, p, n) - BLOCK_LOW((id), p, n))
#define BLOCK_OWNER(index, p, n) (((p) * ((index) + 1) - 1) / (n))
```

Trecerea de la algoritmul secvential la cel paralel:

Se creaza un array dintre care nicio pozitie nu este marcata

In cazul variantei paralele, fiecare proces creaza partea lui de vector - fiecare parte contine practic [n / p] valori.

Fiecare proces va trebui sa stie valoarea k pentru a putea marca multiplii de k din portiunea curenta.

MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);

Prin aceasta functie din biblioteca MPI se trimite valoarea k la toate procesele. Vom folosi MPI_Reduce() pentru a aduna toate sumele intermediare de la fiecare proces intr-o valoare ce va reprezenta totalul global.

3.2 Implementare

La rularea programului, utilizatorul trebuie sa specifice, pe langa numarul de procese, si valoarea n. In cazul in care valoarea n lipseste din lista parametrilor din linia de comanda, se opreste executia programului, caz in care se apeleaza si functia MPI_Finalize(). Daca este oferit si numarul n, realizam convertirea acestuia la tipul de date *int*.

```
if (argc != 2)
{
   if (id == 0) /* parent process */
       printf("Command line: %s <m>\n", argv[0]);
   MPI_Finalize();
   exit(1);
} /* if (argc != 2) */

n = atoi(argv[1]);
```

Determinam minimul si maximul valorilor pentru care fiecare proces este responsabil, alaturi de numarul total de numere pe care le verifica, utilizand functiile macro prezentate mai sus.

```
low_value  = BLOCK_FIRST + BLOCK_LOW(id, p, n - 1)  * BLOCK_STEP;
high_value = BLOCK_FIRST + BLOCK_HIGH(id, p, n - 1) * BLOCK_STEP;
size       = BLOCK_SIZE(id, p, n - 1);
```

Urmatorul pas este alocarea de memorie pentru array-ul ce va reprezenta daca o valoare este marcata sau nu - s-a folosit un array de char.

```
marked = (char*)malloc(size * sizeof(char));
if (marked == NULL)
{
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit(1);
} /* if */
```

Tinem elementele array-ului ca fiind nemarcate:

```
for (i = 0; i < size; i++)
    marked[i] = 0;
```

Determinam indexul primului element din fiecare proces ce trebuie marcat.

```
if (prime * prime > low_value)
{
      first = prime * prime - low_value;
}
else
{
      if (!(low_value % prime))
            first = 0;
      else
            first = prime - (low_value % prime);
}
```

Apoi fiecare proces marcheaza multiplii numarului curent de la indexul calculat pana la finalul portiunii ce i-a fost alocata.

Procesul 0 cauta apoi urmatorul numar prim, gasind urmatorul numar nemarcat.

```
if (id == 0) /* parent process */
{
    while (marked[++index])
        ;
    prime = index + 2;
}
```

Procesul 0 transmite tuturor celorlalte procese valoarea urmatorului numar prim folosind functia urmatoare:

```
MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Procesele continua sa marcheze in array pana cand patratul numarului prim depaseste valoarea lui n.

```
while (prime * prime <= n);
```

Fiecare proces numara cate numere prime sunt in portiunea sa de vector.

```
for (i = 0; i < size; i++)
    if (!marked[i])
        count++;
```

Toate rezultate sunt inglobate in variabila *global_count*. Prin apelul functiei *MPI_Reduce()*, se strang valorile calculate de fiecare proces (*count*) si se face suma lor, care se va salva la adresa variabilei *global_count*.

```
MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Procesul 0 afiseaza raspunsul cautat, alaturi de durata de executie a algoritmului.

```
if (id == 0) /* parent process */
{
    printf("%d primes are less than or equal to %d\n",
           global_count,
           n);
    printf("Total elapsed time: %10.6f\n",
           elapsed_time);
} /* if */
```

3.3 Complexitate

Consideram X ca fiind timpul necesar pentru a marca un element ca fiind multiplu de numar prim. Valoarea trimisa fiecarui proces se realizeaza prin broadcast, asadar costul acestei operatii este a * [log p], a = timpul de latenta al operatiei.

Stim ca numarul de numere prime marginit de intervalul 2 si n este [n / log n].

Numarul de iteratii este (sqrt(n) / log(sqrt(n)))

Asadar, timpul de executie este

X^((n log log n) / p + (sqrt(n) * log(sqrt(n)) * a * log p))

3.4 Optimizare

In continuare, prezentam trei tehnici pentru a imbunatati varianta paralela a algoritmului.

3.4.1 Eliminarea numerelor pare

Datorita faptului ca toate numerele pare, cu exceptia numarului 2, nu sunt numere prime, stocarea si analizarea numerelor pare reprezinta o risipa ce o putem fructifica. Este suficienta stocarea si analizarea numerelor impare.

Cu aceasta optimizare, timpul de executie devine

X^((n log log n) / 2 * p + (sqrt(n) log(sqrt(n)) * a * log p))

3.4.2 Eliminarea broadcast-ului

In varianta actuala a algoritmului, procesul parinte (master) 0 comunica tuturor celorlalte procese valoarea variabilei k, reprezentand cel mai mic numar nemarcat.

Daca pentru fiecare proces s-a cunoaste de la inceput acesta valoare, s-ar obtine un timp mai bun, astfel sa nu se mai foloseasca apelul functiei *MPI_Bcast()*.

Cu aceasta observatie se ajunge la:

X^((n log log n) / 2 * p + X * sqrt(n) * log log (sqrt(n))

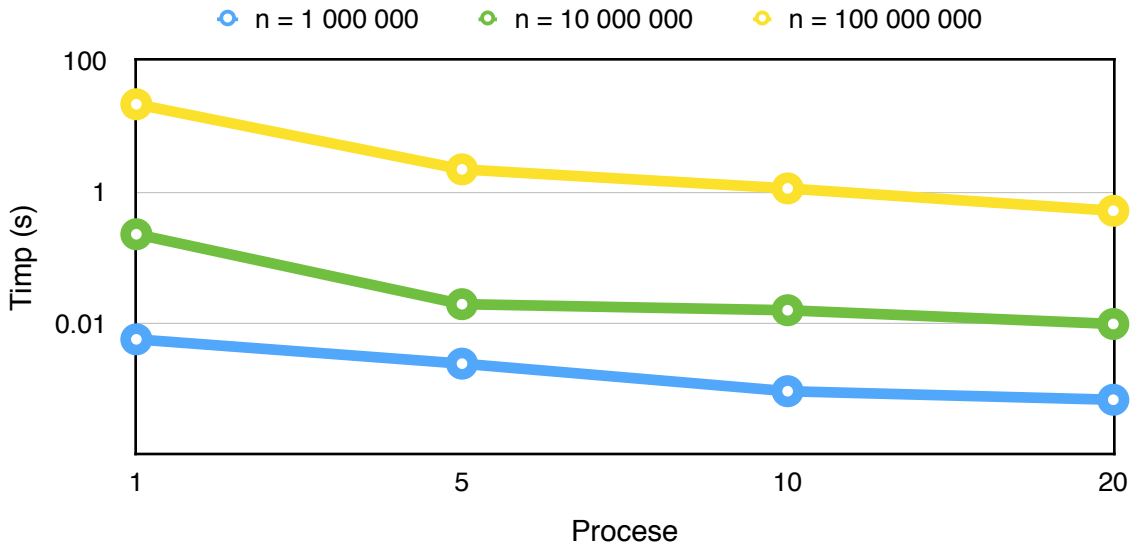
3.4.3 Reorganizarea loop-urilor

In implementarea actuala a algoritmului avem doua loop-uri imbricate. Cand *k* este mare, marcarea elementelor i + k, i + 2 * k, i + 3 * k, … se poate intinde pe o distanta mai mare in memorie, asadar se foloseste memoria cache foarte putin, sau chiar deloc.

Pentru a imbunatati rata de succes a memoriei cache, o optimizare care se poate face este aceea de a schimba ordinea buclelor imbricate, astfel facilitand utilizarea cache-ului.

4. Analiza performantei

n	numere prime <= n	nr procese	timp executie (*s*)
1 000 000	78 498	1	0.005740
		5	0.002456
		10	0.000938
		20	0.000696
10 000 000	664579	1	0.225300
		5	0.019596
		10	0.015854
		20	0.009812
100 000 000	5 761 455	1	21.071835
		5	2.170743
		10	1.114905
		20	0.510633



Bibliografie

Michael J. Quinn, Parallel Programming in C and OpenMP, International Edition 2003.