Dinu Gabriela-Loredana

IS2

Paralelismul in bucla: Parallel One-Dimensional Iterative Averaging

Contents

Paralelismul in bucla

Paralelismul in bucla este o forma de paralelism in programarea software care se preocupa de extragerea sarcinilor care se pot executa in paralel din bucle. Oportunitatea paralelismului la nivel de bucla apare adesea in programele de calcul in care datele sunt stocate in structuri de date cu acces aleatoriu. Acolo unde un program secvential va itera peste structura de date si va opera pe indici pe rand, un program care exploateaza paralelismul la nivel de bucla va folosi mai multe fire sau procese care opereaza pe unii sau pe toti indicii in acelasi timp. Un astfel de paralelism asigura o accelerare a timpului de executie general al programului.

Pentru buclele in care fiecare iteratie este independenta de celelalte, paralelizarea necesita doar alocarea unui proces care sa gestioneze fiecare iteratie. Cu toate acestea, multi algoritmi sunt proiectati sa ruleze secvential si esueaza in paralelizare din cauza dependintelor din cod. Algoritmii secventiali sunt uneori aplicabili in contexte paralele cu usoare modificari. De obicei, totusi, acestea necesita sincronizarea. Sincronizarea poate fi fie implicita, prin transmiterea mesajelor, fie explicita, prin intermediul metodelor de sincronizare precum semaforele.

Tipuri de dependinte la nivel de cod

Se considera S1si S2 ca fiind pasi din bucla. Pe baza acestora avem urmatoarea clasificare a depentintelor:

Nume	Notatie	Descriere
True (Flow) Dependence/ dependinta directa	S1 ->T S2	S1 scrie intr-o locatie din care S2 va citi mai tarziu.
Antidependinta	S1 ->A S2	S1 citeste dintr-o locatie pe care S2 o va scrie mai tarziu.
Dependinta de iesire	S1 ->O S2	S1 si S2 scriu in aceeasi locatie.

Dependinta de intrare	S1 ->I S2	S1 si S2 citesc din aceeasi locatie.

Pentru a pastra comportamentul secvential al unei bucle atunci cand este rulata in paralel, trebuie pastrata dependinta directa. Antidependinta si dependinta de iesire pot fi tratate oferind fiecarui proces propria copie a variabilelor (cunoscuta sub numele de privatizare).

Exemplu de dependinta directa

S1: int a, b;

S2: a = 2;

S3: b = a + 40;

S2 ->T S3

este o dependinta directa pentru ca S2 scrie variabila a, care este citita de S3.

Exemplu de antidependinta

S1: int a, b = 40;

S2: a = b - 38;

S3: b = -1;

S2 ->A S3

este anti-dependinta pentru ca S2 citescte variabila b inainte ca aceasta sa fie scrisa de S3.

Exemplu de dependinta de iesire

S1: int a, b = 40;

S2: a = b - 38;

S3: a = 2;

S2 ->O S3

este un exemplu de dependinta de iesire pentru ca atat S2 cat si S3 scriu aceeasi variabila a.

Exemplu de dependinta de intrare

S1: int a, b, c = 2;

S2: a = c - 1;

S3: b = c + 1;

S2 ->I S3

este un exemplu de dependinta de intrare pentru ca atat S2 cat si S3 citesc aceeasi variabila c.

Tipuri de dependinte la nivel de bucla

Dependinta intre iteratiile buclei (loop-carried)

In dependinta intre iteratiile buclei, instructiunile dintr-o iteratie a unei bucle depind de instructiunile dintr-o alta iteratie a buclei.

Un exemplu de acest fel de dependinta este S1[i] ->T S1[i + 1], unde i este indicele iteratiei curente, iar i - 1 este indicele iteratiei anterioare.

```
for (int i = 1; i < n; i ++) {
    S1: a[i] = a[i - 1] + 1;
}
```

Se poate crea un graf care sa exemplifice dependintele intre iteratiile buclei. Fiecare iteratie este reprezentata ca fiind un nod al grafului iar muchiile dorecte reprezinta dependintele (directe, antidependinte, de iesire sau de intrare) intre fiecare iteratie.

Dependinta independenta de bucla

In dependinta independenta de bucla, buclele au dependinta in interiorul aceleiasi iteratii, dar nu au dependinte intre iteratii. Fiecare iteratie poate fi tratata ca un bloc si efectuata in paralel fara alte eforturi de sincronizare.

```
for (int i = 1; i < n; i ++) {
   S1: tmp = a[i];
   S2: a[i] = b[i];
   S3: b[i] = tmp;
}
```

In exemplul de mai sus, exista o dependenta directa independenta de bucla a S1 ->T S3.

Metode de paralelizare a buclelor

Exista mai multe metode de paralelizare a buclelor:

- Distribuirea
- Paralelism DOALL
- Paralelism DOACROSS
- Paralelism DOPIPE

Procesul de paralelizare poate fi spart in cativa pasi si anume:

Pas	Descriere
Descompunere	Programul este impartit in mai multe sarcini.
Asignare	Sarcinile sunt asignate proceselor.
Orchestrare	Reprezina sincronizarea, comunicarea si accesul la resurse al proceselor.
Mapare	Procesele sunt mapate pe procesoarele fizice.

Distribuirea

Cand o bucla are o dependinta intre iteratiile buclei, o modalitate de a o paraleliza este de a distribui bucla in mai multe bucle diferite. Instructiunile care nu sunt dependente unele de altele sunt separate astfel incat aceste bucle distribuite sa poata fi executate in paralel.

De exemplu,

```
for (int i = 1; i < n; i ++) {
   S1: a[i] = a[i -1] + b[i];
   S2: c[i] = c[i] + d[i];
}
```

Bucla are o dependinta intre iteratiile i si i + 1: S1[i] ->T S1[i + 1] dar S1 si S2 nu sunt dependente asa ca exemplul se poate sparge in doua bucle:

```
loop1: for (int i = 1; i < n; i ++) {
   S1: a[i] = a[i -1] + b[i];
}
loop2: for (int i = 1; i < n; i ++) {
   S2: c[i] = c[i] + d[i];
}
```

Cele doua bucle pot fi executate in paralel.

Paralelism DOALL

Paralelismul de tip DOALL este posibil atunci cand doua secvente de cod din iteratia buclei pot fi executate independent. De exemplu, codul de mai jos nu are nicio dependinta intre iteratii, variabila a[i] nu e niciodata citita, iar b[i] si c[i] nu sunt niciodata scrise:

```
for (int i = 0; i < n; i++) {
   S1: a[i] = b[i] + c[i];
}
```

Asa ca se poate paraleliza executand fiecare iteratie independent in paralel.

```
begin_parallelism();
for (int i = 0; i < n; i++) {
   S1: a[i] = b[i] + c[i];
   end_parallelism();
}
block();
```

Paralelism DOACROSS

Paralelismul DOACROSS este posibil cand iteratiile unei bucle sunt paralelizate extragand calculele ce pot fi realizate independent si apoi folosind metode de sincronizare pentru a satisface dependintele intre iteratii.

Putem considera exemplul de mai jos cu dependinta S1[i] ->T S1[i + 1].

```
for (int i = 1; i < n; i++) {
   a[i] = a[i - 1] + b[i] + 1;
}
```

Fiecare iteratie din bucla contine doua:

- Calculeaza a[i - 1] + b[i] + 1
- Asigneaza valoarea lui a[i]

Cele doua calculi pot fi executate in doua instructiuni separate:

S1: int tmp = b[i] + 1;

S2: a[i] = a[i - 1] + tmp;

Prima secventa nu contine dependinte intre iteratii, asa ca bucla poate fi paralelizata prin a calcula valoarea lui b[i]+1 in paralel si apoi a sincroniza valoarea ce va fi asignata lui a[i].

```
post(0);
for (int i = 1; i < n; i++) {
   S1: int tmp = b[i] + 1;
   wait(i - 1);

   S2: a[i] = a[i - 1] + tmp;
   post(i);
}
```

Paralelism DOPIPE

Cand o bucla are o dependenta intre iteratii, o modalitate de a o paraleliza este de a distribui bucla in mai multe bucle diferite. Scopul DOPIPE este sa actioneze ca o linie de asamblare, in care o etapa este pornita de indata ce exista suficiente date disponibile pentru aceasta din etapa anterioara.

In exemplul de mai jos cu dependinta S1[i] ->T S1[i + 1]

```
for (int i = 1; i < n; i++) {
   S1: a[i] = a[i - 1] + b[i];
   S2: c[i] = c[i] + a[i];
}
```

Se poate observa ca S1 trebuie executata secvential, dar S2 poate fi executata in paralel,

aplicand metodologia DOALL, imediat ce S1 a calculat elementul curent din a.

```
for (int i = 1; i < n; i++) {
   S1: a[i] = a[i - 1] + b[i];
       post(i);
}
for (int i = 1; i < n; i++) {
       wait(i);
   S2: c[i] = c[i] + a[i];
```

}

Problema Propusa

Se da un array unidimensional. La fiecare iteratie, fiecare element din array este inlocuit cu media aritmetica a vecinilor sai.

Doua array-uri separate sunt folosite la fiecare iteratie, unul pentru valorile vechi si unul pentru cele calculate.

Phasers

Un phaser este o constructie foarte asemanatoare cu CountDownLatch care ne permite sa coordonam executia firelor. In comparatie cu CountDownLatch, are unele functionalitati suplimentare.

Phaserul este o bariera la care numarul dinamic de fire trebuie sa astepte inainte de a continua executia. In CountDownLatch, acel numar nu poate fi configurat dinamic si trebuie sa fie furnizat atunci cand cream instanta.

Putem coordona mai multe faze de executie, reutilizand o instanta Phaser pentru fiecare faza a programului. Fiecare faza poate avea un numar diferit de fire care asteapta trecerea la o alta faza. Faza de dupa initializare este egala cu zero.

Pentru a participa la coordonare, firul de executie trebuie sa se inregistreze cu instanta Phaser folosind metode register(). Acest lucru creste doar numarul de parti inregistrate si nu putem verifica daca firul de executie curent este inregistrat.

Firul semnaleaza ca a ajuns la bariera apeland arriveAndAwaitAdvance(), care este o metoda de blocare. Cand numarul de participanti sositi este egal cu numarul de parti inregistrate, executia programului va continua, iar numarul de faze va creste. Putem obtine numarul fazei curente apeland metoda getPhase().

Cand firul isi termina treaba, ar trebui sa apelam metoda arriveAndDeregister() pentru a semnala ca firul de executie curent nu mai trebuie luat in considerare in aceasta faza speciala.

Versiunea secventiala:

```
for (int iteration = 0; iteration < nrOfIterations; iteration++)
{
```

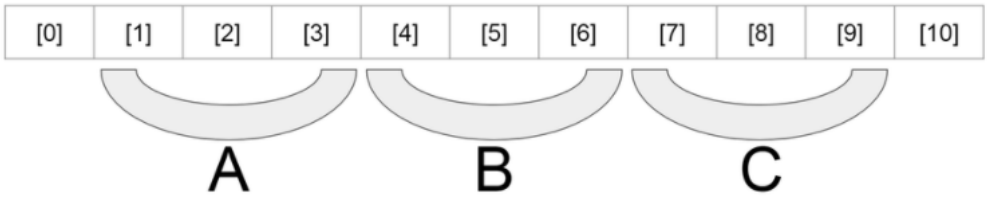
```
        for (int element = 1; element <  nrOfElements-1; element++)
        {
                result[element] = ((array[element - 1] + array[element +
1]) / 2.0);
        }
                tmp = result;
                result = array;
                array = tmp;
}
```

Paralelizare cu phaser

Fiecare thread poate lucra paralel pe o bucata de array deoarece avem doua arrayuri, unul din care doar citim si unul in care doar scriem, deci nu vom avea dependinte la nivelul aceleiasi iteratii. Dar exista dependinte intre iteratii diferite din bucla deoarece la fiecare iteratie rezultatul unui element depinde de rezultatul calculului din iteratia trecuta.

Deci putem aloca fiecarui task o bucata de array pe care sa execute calculele, iar apo isa astepte folosind phaserul pana cand ce celelalte fire de executie si-au terminat treaba in iteratia curenta.

If you are to create 3 slices an array of length 11:



the IndexedRanges should have these properties:

	sliceIndexID	minInclusive	maxExclusive
sliceA	0	1	4
sliceB	1	4	7
sliceC	2	7	10

```
create phaser
register phaser for each task
parallel loop
    sequential loop
```

```
        work
        arrive and await advance on phaser
```

Paralelizare cu fuzzy phaser

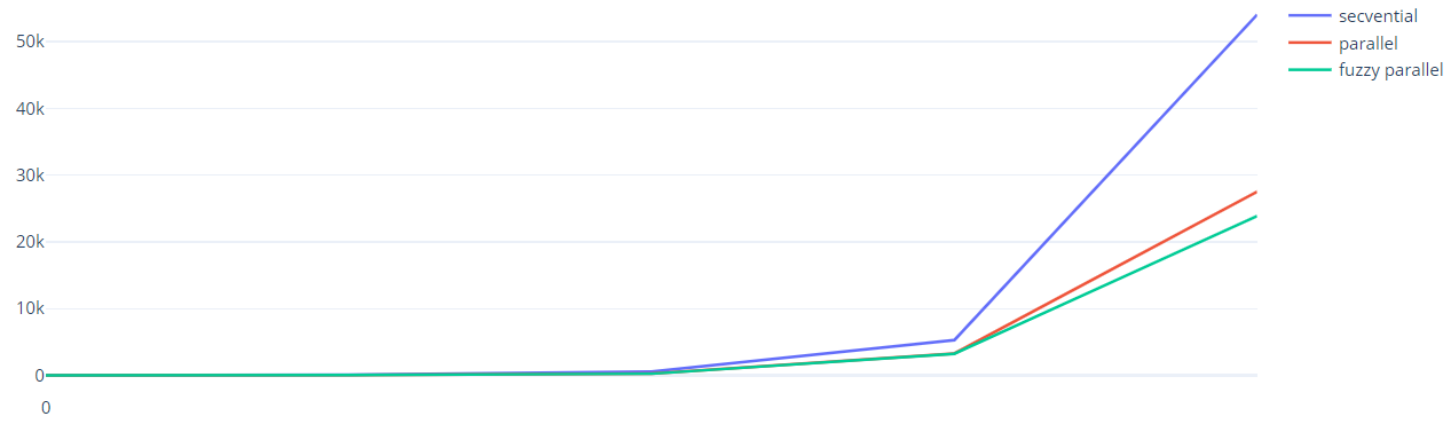
Se poate observa o posibila imbunatatire a algoritmului si anume ca nu toate elementele fiecarui thread sunt dependente de munca celorlalte threaduri din iteratia anterioara ci doar primul si ultimul element din grupul calculat de thread. Deci putem calcula initial primul si ultimul element din grup si apoi sa semnalam utilizand phaserul ca se poate avansa. Apoi vom continua calculele pentru restul elementelor si apoi vom astepta ca si celelalte threaduri sa termine calculele din iteratia curenta.

Rezultate

Am rulat experimentul pe seturi de date de dimensiuni exponential crescatoare. Pe fiecare set de date cele 3 experimente au fost executate de cate 20 de ori pentru a elimina posibilele interferente externe sistemului, iar rezultatele sunt prezentate in tabelul de mai jos.

set de date	secvential	parallel	fuzzy parallel
1000	17ms	9ms	8ms
10000	82ms	43ms	41ms
100000	569ms	259ms	244ms
1000000	5289ms	3283ms	3225ms
10000000	54027ms	27528ms	23883ms

Pentru o mai buna vizualizare a datelor, tabelul de mai sus este transpus intr-un grafic:



Concluzii:

Paralelizarea algoritmului propus a redus semificativ timpii de executie, iar aces lucru este vizibil in special pe seturi mari de date. De asemenea, imbunatatirile adduse prin algoritmului paralel prin cel fuzzy paralel aduce o scadere a timpului.

Proiectul poate fi gasit la:

https://github.com/DinuGabrielaLoredana/Parralel-Iterative-Averaging/

Bibliografie

https://en.wikipedia.org/wiki/Loop-level_parallelism

https://www.sciencedirect.com/topics/computer-science/loop-carried-dependence

https://www.baeldung.com/java-phaser

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html

https://www.cs.rice.edu/~vs3/PDF/SPSS08-phasers.pdf

https://www.coursera.org/lecture/parallel-programming-in-java/parallel-one-dimensional-iterative-averaging-demo-hGVzS

https://github.com/kakshay21/Parallel-Programming-Coursera/blob/master/miniproject_4/src/main/java/edu/coursera/parallel/OneDimAveragingPhaser.java