(19) OFICIUL DE STAT
PENTRU INVENŢII ŞI MĂRCI
Bucureşti

ROMÂNIA

(11) **RO 135546 A2**
(51) **Int.Cl.**
G06F 40/40 (2020.01)

(12) # CERERE DE BREVET DE INVENŢIE

(21) Nr. cerere: **a 2020 00462**

(22) Data de depozit: **29/07/2020**

(41) Data publicării cererii:
28/02/2022 BOPI nr. **2/2022**

(71) Solicitant:
• **YALOS SOFTWARE LABS S.R.L.**,
BVD.REGINA ELISABETA, NR.28, CORP A, MANSARDA, BIROU A31, SECTOR 5, BUCUREŞTI, B, RO

(72) Inventatori:
• **IALOVOI LAURENŢIU RADU**,
CALEA VICTORIEI, NR.23, AP.9, SECTOR 3, BUCUREŞTI, B, RO;
• **DAVIDOIU ALEXANDRA**,
CALEA MOŞILOR, NR.221, BL.31A, SC.1, ET.2, AP.6, BUCUREŞTI, B, RO

(54) PROCEDEU DE PARALELIZARE A LIMBAJELOR SECVENŢIALE

(57) **Rezumat:**

Invenţia se referă la un procedeu de paralelizare a limbajelor secvenţiale cu aplicabilitate în domeniul seturilor masive de date, prin a cărui implementare se pot obţine parametri caracteristici supercalculatoarelor pe maşini de calcul obişnuite.

Revendicări: 2
Figuri: 4

RO 135546 A2

RO 135546 A2

Descriere

Procedeu de paralelizare a limbajelor secvențiale

Domeniul tehnic: la care se refera procedeul este: informatică aplicată

Prezentarea stadiului tehnicii:

Limbajele tradiționale de programare sunt concepute in general pentru mașini Von Neuman si ca atare generează cod mașina care este executat tipic pe un singur procesor. Pe de alta parte, sistemele de calcul in uz curent operează cu mai multe procesoare. Mitigarea intre capacitatea limbajului de a genera numai cod secvențial si capacitatea mașinii de a rula cod in paralel se face prin primitive puse la dispoziție de sistemul de operare cum ar fi firul de execuție („thread" in engleza).

Prin cererea de brevet „Partajarea Memoriei intre mașini discrete" autorii prezinta o tehnica prin care un proces rulând pe o mașina arbitrara poate accesa in scriere si citire memorie RAM de pe un număr arbitrar de mașini. Tehnica este prezentata succint in Figura 1. Partajarea memoriei intre mașini discrete. Un proces rulând pe mașina #1 poate prin aceasta tehnica sa acceseze memorie fizica e mașinile #2 .. #3. Un proces prealabil de sindicalizare înregistrează mașinile si zonele de memorie RAM sindicalizate

Prezentul procedeu propune o metoda prin care se realiza o paralelizare intrinseca a limbajului ce ascunde primitivele sistemului de operare. Mai mult, codul este rulat pe masa mașinilor sindicalizate, putând folosi atât RAM cat si procesoare aflate in toate mașinile.

Prezentarea problemei tehnice:

In Figura 2. Alocatoare sincronizate se generalizează sistemul de partajare prin aceea ca introduce cate un alocator de memorie pentru fiecare dintre mașinile sindicalizare. Din punct de vedere al procedeului este esențial ca o adresa virtuala in mașina #1 sa fie asociata cu in mod identic in toate celelalte mașini sindicalizate. Astfel, daca adresa virtuala P1 este asociata in mașina #1 la adresa fizica B1 din mașina #1, in mașina #2 aceeași adresa P1 va fi asociata la adresa fizica B1 aflata in mașina #1 si aceeași asociere va fi prezenta pe toata familia de mașini. In figura 2 aceasta este exemplificata prin multiplicarea tabelului de adrese virtuale in toate mașinile.

In acest fel masivul de memorie înregistrat este disponibil in citire/scriere in toate mașinile sindicalizate. Codul compilat prin acest procedeu va fi încărcat in spațiul de adrese al alocatorului si astfel va avea acces intrinsec la masivul de memorie.

In Figura 3. Distribuție cod compilat se prezinta instantaneea codului. Paralelizarea se obține prin rularea aceleași cod pe toata familia de mașini, fiecare dintre ele având procesoare locale. Din punct de vedere al memoriei, tabelul de alocare virtuala este comun tuturor mașinilor sindicalizate si a fost definit ca un tabel la rețelei.

Funcția principala a alocatorului nu este de a pune la dispoziția codului compilat acces la memorie ci de a abstractiza paralelismul. In Figura 4. Unitate de execuție pe o mașina arbitrara este prezentat procedeul in o mașina cu toate componentele sale:

- Alocatorul de memorie controlează accesul la memorie.
- Algoritmii paraleli abstractizează conceptele de paralelism si pun la dispoziție primitive atomice.

- Codul compilat utilizează algoritmii prin API
- Eventual, codul compilat poate accesa direct memoria.

Prin abstractizare noţiunii de paralelism se înţelege introducerea de primitive ale limbajului pe care se implementează procedeul. In funcţie de limbaj aceştia pot fi funcţii, functori sau predicate sau alte facilitaţi puse la dispoziţie de limbaj. Implementarea acestora conform procedeului este singura care face uz de elementele de paralelism ale sistemului de operare ţintit. La nivelul codului compilat se accesează atomi, prin un API pus la dispoziţie de implementare.

Procedeul are următoarele elemente:

- Element de sindicalizare a memoriei. Face obiectul unei cereri separate de brevet.
- Alocatoare de memorie pe sistemul sindicalizat.
- Algoritmi paraleli.

Modalitatea prin care codul compilat este transportata pe suma maşinilor nu face obiectul acestui procedeu. La minim acesta poate fi realizata prin copiere.

Invenţia se referă la o metoda prin care se realiza o paralelizare intrinseca a limbajului ce ascunde primitivele sistemului de operare. Mai mult, codul este rulat pe masa maşinilor sindicalizate, putând folosi atât RAM cat si procesoare aflate in toate maşinile

Descrierea figurilor:

Figura 1. Partajarea memoriei intre maşini discrete.

Figura 2. Alocatoare sincronizate

Figura 3. Distribuţie cod compilat

Figura 4. Unitate de execuţie pe o maşina arbitrara

Exemplificări:

Prin standard (ANSI) C++ introduce o biblioteca standard numita STL (standard template library). Aceasta biblioteca este un foarte mare pas înainte pentru ca introduce un mecanism foarte eficace de separare a datelor si algoritmilor. Foarte important pentru proiectul de fata, introduce doua concepte esenţiale: **container si algoritm**. Un container împachetează datele in timp ce un algoritm operează pe datele din container. Containere standard sunt implementate, cum ar fi: vector, lista, stiva, etc. La fel si algoritmi (căutare, sortare, etc).

Păstrând filozofia timpului conţinerii si algoritmii sunt gândiţi pentru operare secvenţiala. Aplicând tehnica descrisa se poate trece la **paralelizarea acestor concepte**. Într-adevăr, cel puţin in C++ se poate introduce un container care este identic in funcţionare cu un container tradiţional dar este alocat pe un spaţiu de maşini fizice prin schimbarea alocatorului. A se vedea definiţia vectorului, spre exemplu (http://www.cplusplus.com/reference/vector/vector/):

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Unde instantaneea tipului suporta un parametru adiţional care se ocupa cu alocarea/dealocarea memoriei. Se poate o defini biblioteca adiţionala care defineşte alocatori sofisticaţi care interacţionează cu un sistem de management al memoriei care transcende graniţa maşinii de calcul. In acest fel vectorul (spre exemplu) devine o structura cu 2 nivele: nivel de cache local, aflat in zona de

memorie a programului in execuţie si un nivel de date globale aflat in sistemul de management al memoriei. Încercarea de a accesa un element care nu este in cache (miss) va aduce o zona de date (pagina) din memoria globala.

Cele de mai sus definesc un container care poate transcende o maşina de calcul dar nu introduc elemente de paralelizare intrinseca. Paralelizarea intrinseca se introduce prin **algoritmi**. Aici se are in vedere introducerea de algoritmi cu operare pe sistemul de management al memoriei care sa substituie algoritmi STL tradiţionali. Spre exemplu, C++ introduce pentru sortare următorul algoritm (sort):

```
template <class RandomAccessIterator>
  void sort (RandomAccessIterator first, RandomAccessIterator last);
```

Se introduce in plus fata de acest algoritm un p_sort dupa cum urmează:

```
template <class RandomAccessIterator>
  void p_sort (RandomAccessIterator first, RandomAccessIterator last);
```

Când alocatorul literatorilor este standard acesta nu face decât sa mapeze pe vechiul algoritm. Când se detectează alocator pe sistemul de management al memoriei execuţia se deleagă către acesta.

Cele de mai sus definesc un model de paralelizare intrinseca a limbajului. Într-adevăr, se poate imagina cod de genul urmator:

```
1:    p_vector date;
2:    p_load<formator>( date, locatie);
3:    p_sort( date.begin(), date.end());
4:    cout << date.begin()->as_string();
```

Unde liniile 2 si 3 sunt executate pe suma maşinilor/procesoarelor existente (prin formator se are in vedere un functor care creaza un obiect dintr-un sir de caractere).

Se are in vedere si o paralelizare definita, ca opus al paralelizării intrinseci, prin introducerea de functori delocalizaţi. Un altfel de functor este executat pe sistemul de management a memoriei si nu ca parte a programului in execuţie. Spre exemplu, definim o ipotetica operaţiune de calcul al sumei tuturor elementelor unui vector după cum urmează:

```
1:    p_vector date;
2:    p_load( date, locatie, formator);
3:    p_summabile acumulator = 0;
4:    p_for_each( date.begin(), date.end(), acumulator );
5:    cout acumulator->as_string();
```

Este importat de menţionat ca acumulatorul este la rândul sau un tip de date **alocat pe sistemul de management al memoriei**. De data aceasta el este si **executat** acolo. Într-adevăr, pentru a fi utilizabil tipul `p_summable` trebuie sa definească o operaţie echivalenta cu:

```
void functor(const Type &a);
```

unde tipul `a` este un tip către care literatorii pot fi nepreferenţiaţi si convertiţi implicit. Atât timp cat `p_for_each` este un algoritm executat pe sistemul de management al memoriei si operaţia trebuie sa fie executabila in acelaşi context.

15

REVENDICĂRI

Procedeu de paralelizare a limbajelor secvenţiale caracterizat prin faptul că permite scrierea de cod secvenţial care este apoi paralelizat prin procedeu. Codul astfel scris rulează pe o familie de procesoare din maşini de calcul diferite si accesează memoria tuturor maşinilor.

Revendicări: 2

1. Execuţie cod compilat in mod simetric pe o familie de procesoare, in mai multe maşini de calcul.
2. Distribuire a datelor pe mai multe maşini de calcul.

DESENE

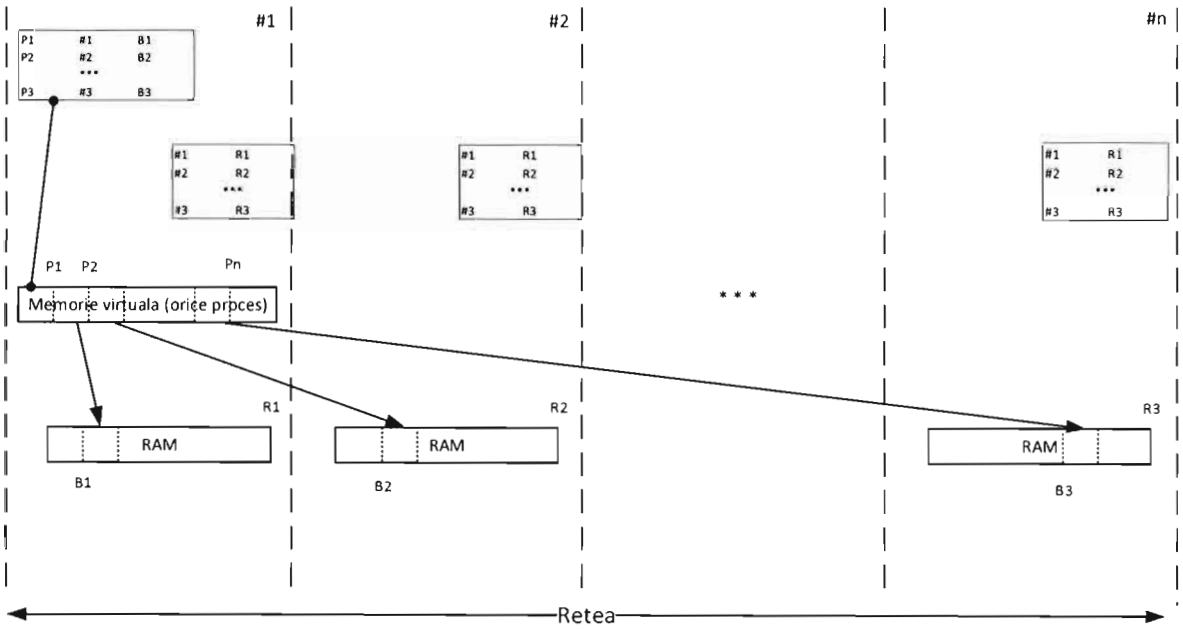

Figura 1. Partajarea memoriei intre mașini discrete.

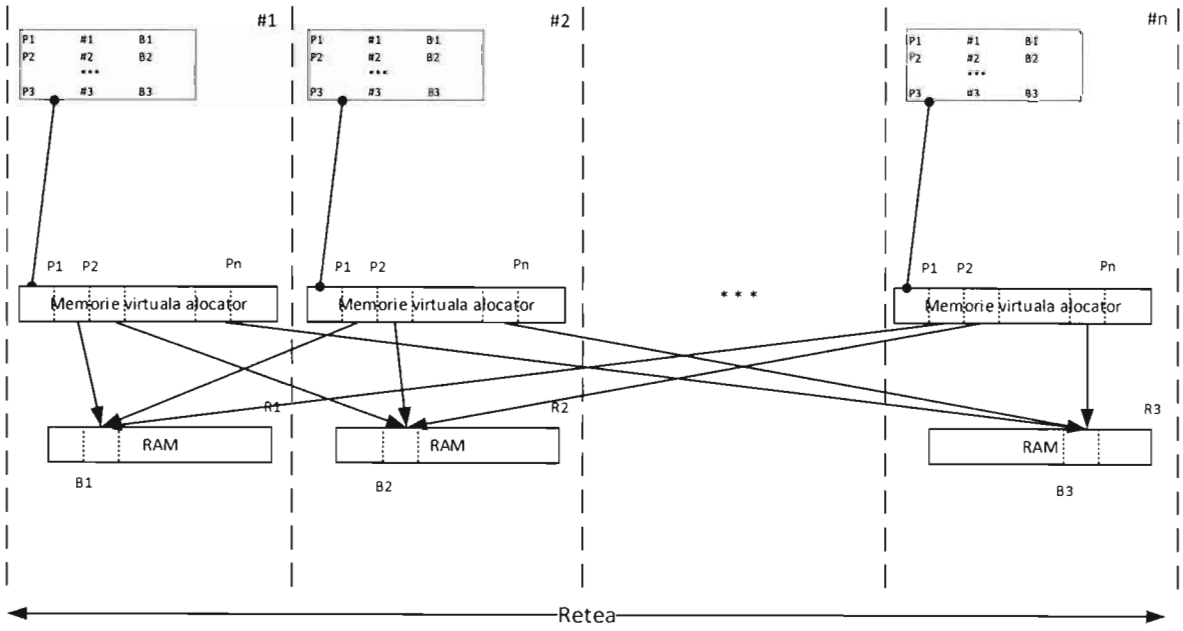

Figura 2. Alocatoare sincronizate

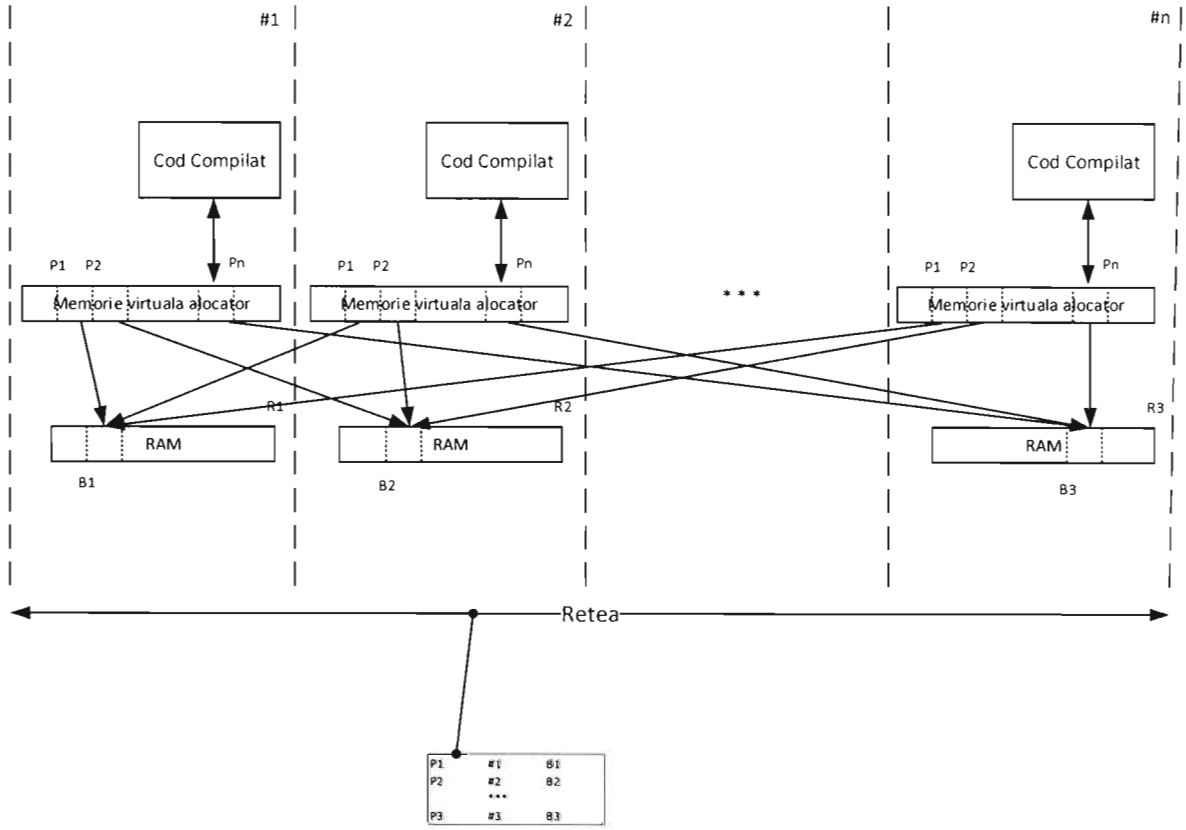

Figura 3. Dispunere cod compilat.

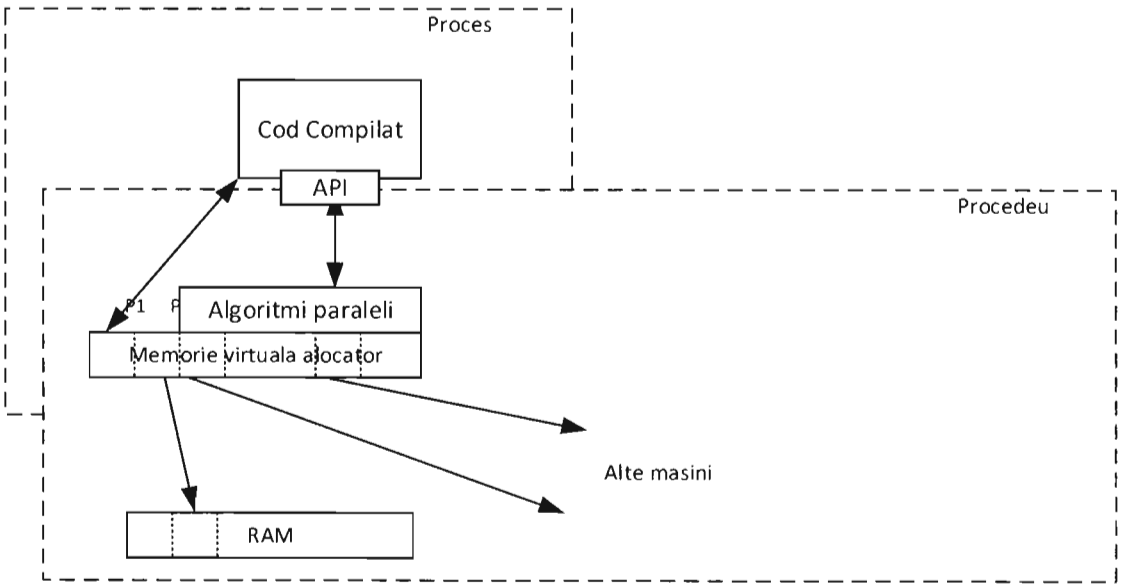

Figura 4. Unitate de execuție pe o mașina arbitrara