Tema 1 - Simularea activităţii unui cluster

Dan Dragomir (dan.dragomir at cs pub ro)
Adriana Drăghici (adriana.draghici at cs pub ro)

March 9, 2013

1 Introducere

Un cluster este o colecţie de noduri (calculatoare) individuale conectate între ele, care lucrează împreună. Într-un cluster există mai multe tipuri de noduri:

- Front-End Processors (FEPs) - coordonează accesul utilizatorilor la cluster
- Noduri computaţionale - prelucrează cererile utilizatorilor
- Noduri de stocare - stochează datele din cluster (în unele arhitecturi de cluster funcţiile de stocare şi de prelucrare sunt executate pe acelaşi nod)

În linii mari activitatea unui cluster poate fi descrisă astfel:

- utilizatorii trimit FEP-urilor job-urile care vor fi prelucrate de cluster; fiecare job este format din mai multe task-uri
- FEP-urile distribuie task-urile din job-urile primite către nodurile computaţionale din cluster; deoarece nodurile sunt echipate cu procesoare multi-core mai multe task-uri pot fi atribuite unui singur nod
- nodurile computaţionale încep execuţia task-urilor interogând nodurile de stocare în funcţie de necesităţile task-ului

2 Enunţ

Pentru această temă va trebui să simulaţi, folosind limbajul Python, comportamentul nodurilor din cluster. Job-urile trimise de utilizatori presupun înmulţirea a două matrici pătratice de dimensiuni mari. Fiecare job va fi format din mai multe task-uri, fiecare task necesitând calcularea unui bloc din matricea finală.

Datele de intrare se află distribuite pe nodurile din cluster. În cadrul temei vom considera că funcţiile de stocare şi prelucrare sunt executate de acelaşi nod. Este însă posibil ca datele necesare unui nod să nu fie stocate local, caz în care el trebuie să facă o cerere de date către un alt nod.

Pentru rezolvarea unui job nodurile vor fi organizate sub forma unei matrici pătratice, fiecare nod primind la pornire, ca identificator, un tuplu (i, j), cu $i, j \in [0, \sqrt{nr\_noduri})$, reprezentând poziţia lui în această matrice. Distribuirea celor două matrici de intrare A şi B pe noduri se face împărţind matricile în blocuri pătratice de dimensiuni egale, fiecare nod stocând un astfel de bloc din fiecare matrice, A şi B. Astfel, nodul

cu indentificatorul (i, j) va primi blocul cu indexul (i, j) din fiecare matrice. Blocul cu indexul (i, j) dintr-o matrice X este format din elementele $X[i * block\_size : (i+1) * block\_size][j * block\_size : (j+1) * block\_size]$

2.1 Nodul

Concret, va trebui sa implementaţi un obiect Python, denumit *Node*, care va realiza funcţionalitatea unui nod. API-ul minimal care trebuie oferit de către această clasă este:

- un constructor cu următorul prototip

```
def __init__(self, node_ID, block_size, matrix_size, data_store):
    """
        Constructor.

        @param node_ID: a pair of IDs uniquely identifying the node;
            IDs are integers between 0 and matrix_size/block_size
        @param block_size: the size of the matrix blocks stored in this
            node's datastore
        @param matrix_size: the size of the matrix
        @param data_store: reference to the node's local data store
    """
```

- o metodă care primeşte lista tuturor nodurilor care participă la job-ul curent

```
def set_nodes(self, nodes):
    """
        Informs the current node of the other nodes in the cluster.
        Guaranteed to be called before the first call to
        compute_matrix_block.

        @param nodes: a list containing all the nodes in the cluster
    """
```

- o metodă pentru a primi task-uri

```
def compute_matrix_block(self, start_row, start_column,
        num_rows, num_columns):
    """
        Computes a given block of the result matrix.
        The method invoked by FEP nodes.

        @param start_row: the index of the first row in the block
        @param start_column: the index of the first column in the block
        @param num_rows: number of rows in the block
        @param num_columns: number of columns in the block

        @return: the block of the result matrix encoded as a row-order
            list of lists of integers
```

```
    """
```

- o metodă care indică nodului când trebuie să-şi termine execuţia

```
def shutdown(self):
    """
        Instructs the node to shutdown (terminate all threads).
    """
```

2.2 Datastore-ul

Accesarea matricilor de intrare se face prin intermediul unui obiect *datastore* primit de fiecare nod în constructorul său. Fiecare nod are asociat un datastore propriu, care poate fi accesat doar de către thread-urile nodului respectiv. API-ul oferit de către obiectul *datastore* este prezentat mai jos:

- o metodă pentru a înregistra thread-urile nodului cu obiectul datastore; este **obligatoriu** ca toate thread-urile nodului care vor să acceseze datastore-ul să se inregistreze la acesta înainte de a cere date

```
def register_thread(self, node):
    """
        Registers the current thread of the given node to the data store.

        @param node: the node owning the data store
    """
```

- o metodă care returnează elementul aflat pe poziţia $(row, column)$ în cadrul blocului din matricea A stocat de acest datastore

```
def get_element_from_a(self, node, row, column):
    """
        Returns an element from the matrix A stored in the data store.
        This is a blocking operation. The maximum number of
        in-flight requests is limited, see get_max_pending_requests().

        @param node: the node accessing the data store;
            must be the node that owns the data store
        @param row: the element row
        @param column: the element column

        @return: the element of matrix A at position (row, column)
    """
```

- o metodă care returnează elementul aflat pe poziţia $(row, column)$ în cadrul blocului din matricea B stocat de acest datastore

```
def get_element_from_b(self, node, row, column):
    """
        Returns an element from the matrix B stored in the data store.
```

```
        This is a blocking operation. The maximum number of
        in-flight requests is limited, see get_max_pending_requests().

        @param node: the node accessing the data store;
            must be the node that owns the data store
        @param row: the element row
        @param column: the element column

        @return: the element of matrix B at position (row, column)
    """
```

- o metodă care returnează numărul maxime de cereri simultane suportate de către datastore; este responsabilitatea nodului de a nu depăşi acest număr de cereri

```
def get_max_pending_requests(self, node):
    """
        Returns the maximum number of in-flight requests
        supported by this data store.

        @param node: the node accessing the data store;
            must be the node that owns the data store

        @return: the maximum number of in-flight requests
            supported by this data store
    """
```

Fiecare cerere pentru un element din datele de intrare, fie ca este necesar nodului local sau reprezintă răspunsul pentru o cerere a unui nod remote, trebuie să fie citit din datastore. Este **interzis** să cache-uiţi conţinul datastore-ului în cadrul clasei *Node*.

3 Testare şi notare

Tema va fi testată automat folosind infrastructura de testare. Aceasta generează aleator două matrici de intrare care sunt distribuite pe datastore-uri, iar apoi trimite task-uri către nodurile implementate de către voi. Fiecare test este rulat de mai multe ori pentru a detecta eventualele bug-uri de sincronizare. Există un timeout, specific fiecărui test, pentru terminarea execuţiei tuturor rulărilor testului respectiv.

Infrastructura de testare, scheletul clasei Node, precum şi documentaţia API-ului poate fi descărcată de aici.

Tema se va implementa în Python 2.7. Arhiva temei (fişier .zip) va uploadată pe site-ul cursului şi trebuie să conţină:

- fişierul node.py cu implementarea clasei Node
- alte surse .py folosite de soluţia voastră (nu includeţi fişierele infrastructurii de testare)
- fişierul README cu detaliile implementării temei (poate fi şi în engleză)

Notarea se va face în felul următor:

- 80 pct - 4 teste a câte 20 de puncte
- 20 pct - calitatea implementării:
  - organizarea, eleganţa şi simplitatea implementării
  - lizibilitatea codului, consistenţa stilului
  - comentariile utile din cod
  - fişierul README (-10 puncte pentru lipsa acestuia sau lipsa conţinutului lui)
  - alte situaţii nespecificate aici, dar considerate inadecvate

4 Observaţii

- pot exista depunctări mai mari decât este specificat în secţiunea *Notare* pentru implementări care nu respectă spiritul temei
- implementarea şi folosirea API-ul oferit este obligatorie
- toate operaţiile făcute de un nod (calcule, accesare datastore, comunicare cu alte noduri) trebuie făcute din thread-uri proprii nodurilor; în consecinţă nodul vostru trebuie să conţină cel puţin un thread
- este interzis să cache-uiţi datele citite din datastore
- bug-urile de sincronizare, prin natura lor sunt nedeterministe; o temă care conţine astfel de bug-uri poate obţine punctaje diferite la rulări succesive; în acest caz punctajul temei rămâne la latitudinea celui care o corectează