Tema 1 - Simularea activităţii unui cluster

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

15.03.2015

1 Obiective

- implementarea unei aplicații concurente folosind thread-uri
- folosirea eficientă a elementelor de sincronizare

2 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 eventual celelate noduri pentru seturile de date necesare, care nu sunt stocate local

3 Enunţ

Pentru această temă va trebui să implementați, folosind limbajul Python, o simulare a execuției task-urilor într-un cluster. Implementarea va consta în descrierea comportamentului unui nod al clusterului. El va primi task-uri de executat, va trebui să adune datele necesare task-ului de la celelate noduri (gather), va trebui să execute task-ul și apoi să distribuie rezultatul (scatter).

Fiecare nod al clusterului va fi simulat printr-un obiect al clasei `Node`, scrisă de voi. Această clasă trebuie să respecte o anumită interfață, prin intermediul căreia primește task-uri de la FEP și se sincronizează cu acesta. Pentru rezolvare puteți porni de la scheletul clasei din arhiva temei. Execuția în paralel a task-urilor

primite de fiecare nod va fi făcută, bineînțeles, cu ajutorului thread-urilor. Va trebui de asemenea să folosiți obiecte de sincronizare pentru a efectua comunicarea cu celelate noduri și cu FEP-ul.

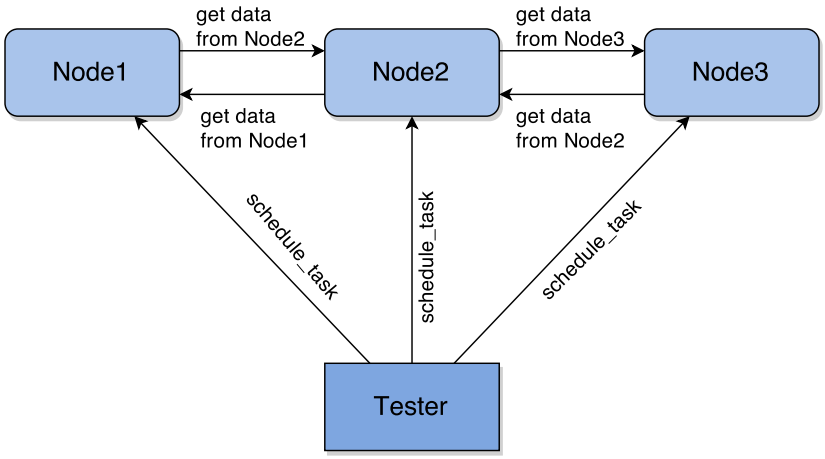

Figure 1: Arhitectura clusterului

În cazul acestei teme, vom privi nodurile ca având dublă funcție, și computaţională și de stocare, având astfel asociate date. Pentru simplificare vom considera că datele asupra cărora acționează task-urile constau într-un vector de intregi, distribuit în părți egale, pe fiecare nod.

Simularea FEP-ului este făcută de către infrastructura de testare (Tester). Acesta va crea obiectele `Node` din clusterul simulat, va informa fiecare nod despre celelalte noduri din cluster, le va da task-urile și va verifica apoi rezultatele.

4 Implementare

Pentru rezolvarea acestei teme va trebui să completați clasa `Node` cu implementarea metodelor deja definite în schelet. Pe lângă acestea puteți crea bineînțeles oricâte alte metode/clase/module aveți nevoie în rezolvare. Nu modificați alte clase prezente în scheletul temei decât pentru debugging. Detaliile metodelor clasei `Node` și ale celorlate clase din infrastructura de testare le găsiți în *docstring*-ul temei, în `doc/index.html`.

Un task reprezintă o operație efectuată pe o porțiune din setul total de date stocat pe noduri. Pentru simplificare, rezultatul unui task este un vector de date de dimensiune egală cu vectorul de date de intrare. Vectorul de date de intrare, însă, nu este întreg vectorul de date stocat de cluster și nu este neapărat stocat pe un singur nod. El trebuie construit de nodul care execută task-ul prin aducerea diferitelor bucăți stocate remote (gather). Operația inversă trebuie implementată pentru distribuirea rezultatului înapoi pe noduri (scatter). Distribuirea rezultatului nu se face însă neaparat pe aceleași noduri de unde au fost luate datele de intrare. De asemenea, distribuirea rezultatelor nu se face prin înlocuirea valorilor din destinație, ci prin `adunarea` rezultatelor în destinație.

Operațiile unui task sunt încapsulate de clasa `Task`, oferită în scheletul de cod, și care nu trebuie modificată. Rularea task-ului se face prin apelarea metodei `run`, cu vectorul de date de intrare adunat sub forma unei liste, urmând ca după un timp de procesare, rezultatul să fie întors tot sub forma unei liste.

Atribuirea task-ului unui nod, pentru rulare, se face prin metoda `schedule_task` a clasei `Node`. În același apel, se trimit nodului și locațiile de unde trebuie aduse datele de intrare și unde trebuie pus rezultatul task-ului. Acestea sunt reprezentare de două liste de tupluri de forma $(nod, index\_start, index\_end)$, una pentru datele de intrare și una pentru datele de ieșire.

Task-urile sunt executate pe noduri în runde. Toate task-urile primite de un nod într-o rundă trebuie să fie rulate cu datele disponibile pe noduri la începutul rundei. Sfârșitul rundei este semnalizat de apelarea metodei `sync_results` pe nodurile clusterului.

Un exemplu de adunare a datelor și distribuire a rezultatelor unui task este prezentat în continuare. Presupunem că lista de date de intrare este specificată de [(0, 10, 20), (3, 8, 100)], iar cea a datelor de iesire de [(1, 35, 40), (0, 1, 8), (3, 11, 101)]. Astfel, datele de intrare sunt reprezentate de un vector de 102 elemente:

- 10 elemente de la nodul 0: de la indexul 10 la 19 (inclusiv)
- 92 elementele de la nodul 3: de la indexul 8 la 99 (inclusiv)

Cele 102 elemente din rezultat vor fi distribuite astfel:

- 5 elemente vor fi adunate cu elementele nodului 1: de la indexul 35 la 39 (inclusiv)
- 7 elemente vor fi adunate cu elementele nodului 0: de la indexul 1 la 7 (inclusiv)
- 90 elemente vor fi adunate cu elementele nodului 3: de la indexul 11 la 100 (inclusiv)

5 Notare

Tema va fi verificată automat, folosind infrastructura de testare, pe baza a 10 teste definite în directorul `tests`. Testele sunt rulate de mai multe ori pentru a detecta eventualele bug-uri de sincronizare. Există un timeout, specific fiecărui test, în care trebuie să se încadreze execuţia tuturor iteraţiilor testului respectiv.

Fiecare iterație conține mai multe runde, iar în fiecare rundă fiecare nod primește acelaşi număr de taskuri (numărul de task-uri e definit în cadrul fiecărei runde). Dacă într-o rundă nodurile se blochează, aruncă o excepție sau execută incorect task-urile, atunci execuția iterației se încheie și este considerată incorectă. Punctajul dat unui test depinde de toate iterațiile acestuia, deci pot exista punctaje parțiale. De exemplu, dacă testul 1 pică o dată din 4 iterații, atunci se primește 75% din punctajul aferente testului.

Infrastructura de testare, scheletul clasei `Node`, precum şi documentaţia API-ului poate fi descărcată de aici [1].

Tema se va implementa în Python 2.7. Arhiva temei (fişier .zip) va fi 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 în engleză)

Notarea va consta în 100*pct* pe baza a 10 teste a câte 10*pct* fiecare. Depunctări posibile sunt:

- folosirea busy-waiting (între $-10pct$ și $-50pct$ în funcție de gravitate)
- folosirea lock-urilor globale ($-10pct$)
- folosirea variabilelor globale ($-5pct$)
- folosirea incorectă a variabilelor de sincronizare (e.g. lock care nu protejează toate accesele la o variabilă partajată $-2pct$)
- lipsa organizării codului, implementare încâlcită și nemodulară, cod duplicat, funcții foarte lungi (între $-1pct$ și $-5pct$ în funcție de gravitate)
- cod înghesuit/ilizibil, inconsistenţa stilului (între $-1pct$ și $-5pct$ în funcție de gravitate)
- lipsa comentariilor utile din cod ($-5pct$)

[1] `http://cs.curs.pub.ro/2014/pluginfile.php/11568/mod_assign/intro/tema1-skel.zip`

- fişier README sumar ($-10pct$ pentru fișier gol sau inexistent)
- print-uri de debug ($-1pct$)
- cod comentat/nefolosit ($-1pct$)
- alte situaţii nespecificate aici, dar considerate inadecvate

6 Precizări

- pot exista depunctări mai mari decât este specificat în secţiunea *Notare* pentru implementări care nu respectă obiectivele temei
- implementarea şi folosirea API-ul oferit este obligatorie
- toate operaţiile făcute de un nod (execuția taskului, comunicare cu alte noduri) trebuie făcute din thread-uri proprii nodului; atenție, metoda `schedule_task` este apelată de tester de pe thread-ul său
- pentru încadrare în timp, tema nu necesită optimizarea operațiilor (ex: planificarea task-urilor, scatter/gather), ci folosirea corectă a operațiilor blocante ale elementelor de sincronizare
- considerați că nodurile clusterului sunt echipate cu 16 core-uri de procesare
- 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 va fi cel dat de tester în momentul corectării pe sistemul asistentului
- recomandăm testarea temei în cât mai multe situații de load al sistemului și pe cât mai multe sisteme (fep, cluster etc.) pentru a descoperi bug-urile de sincronizare
- pentru code-style recomandăm ghidul oficial `https://www.python.org/dev/peps/pep-0008/`