Temă de casă

Algoritmi paraleli şi distribuiţi

Student: Grigoraş Alexandru, 1407B

1. Descrierea problemei

Tema de casă presupune realizarea unui algoritm de tipul MapReduce. Acesta are rolul de a indexa cuvintele unui set de fişiere dintr-un folder precizat [4]. Acesta va contoriza numărul apariţiilor cuvintelor din fiecare fişier precizat şi va afişa la ieşire lista tuturor cuvintelor şi apariţia lor în documentele căutate.

Paradigma MapReduce este prezentată în Figura 1.

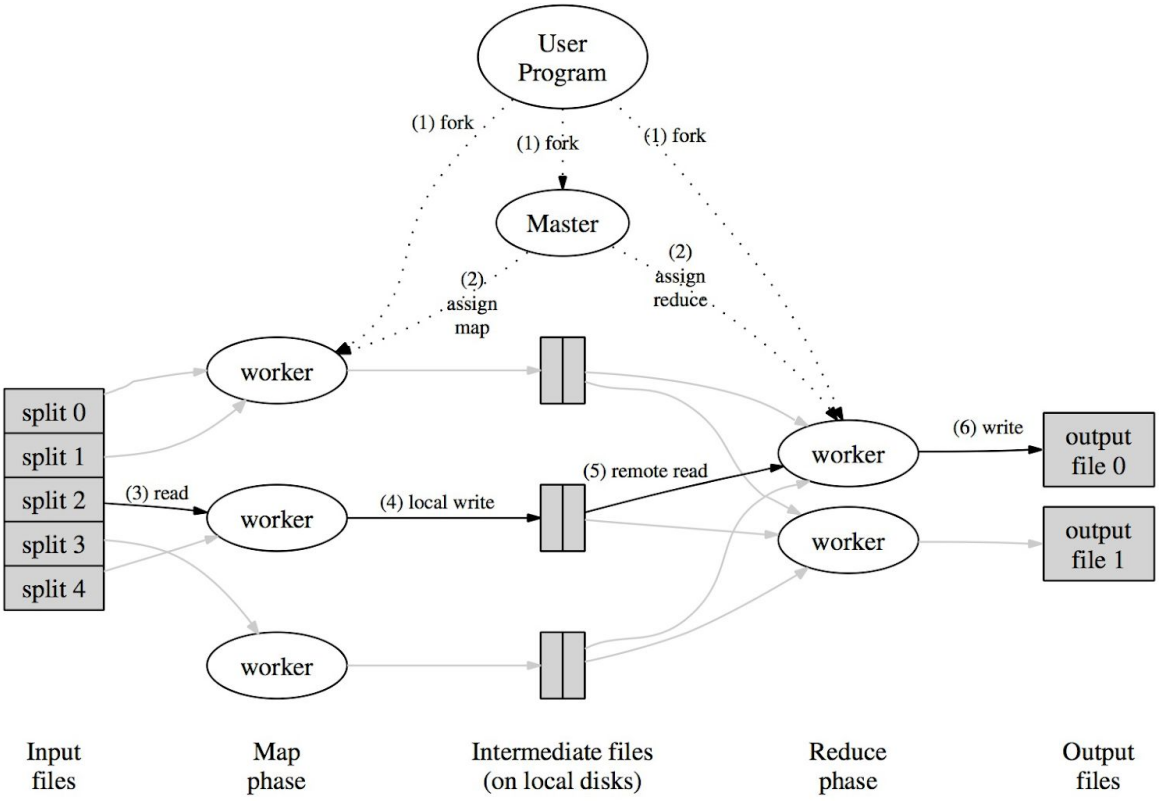

Figura 1. Paradigma MapReduce [4]

Problema cerută primeşte la intrare un folder, iar la ieşire, folosind paradigma anterioară, are o listă de cuvinte ce conţine documentul în care se găseşte şi frecvenţa lor.

2. Modalitatea de rezolvare

Implementarea aleasă conţine 2 tipuri de procese, referite în [1]:

- MASTER - Procesul principal ce distribuie acţiunile către Workeri;
- WORKER - Procese secundare ce fac procesările necesare atribuite de Master;

În începerea programului, se lansează în execuţie mai multe procese. Procesul *Master* selectează folderul dat şi caută toate numele fişierelor. Trimite astefel fiecărui proces *Worker* numele fişierului (prin MPI_Send, referit în [3]).

Procesările ce urmează sunt clasificate în două etape:

2.1. Etapa de mapare

Procesul *Worker* ce a primit mesajul de la *Master* (MPI_Recv, referit în [3]) deschide fişierul cu numele specificat şi începe citirea lui, linie cu linie, identificând cuvintele.

Fiecare cuvânt găsit este astfel adăugat într-o structură ce conţine: text-ul, numărul de documente din care face parte (iniţial 1) şi o listă cu numele documentelor şi frecvenţa de apariţie în acestea. Cuvintele sunt apoi adăugate în tabela Hash corespunzătoarea fiecărui Worker. Structura de tip Hash este preluată din [2].

Tabela Hash considerată conţine un număr ales de chei (cheie_maximă), funcţia de dispersie pentru determinarea cheii verifică suma codurilor ASCII ale literelor şi le mapează în intervalul [0 - cheie_maximă]. Coliziunile sau cuvintele care au aceeaşi cheie sunt legate printr-o listă înlanţuită.

După ce a terminat toate adăugările de cuvinte în tabelă, procesul curent salvează înregistrările tabelei Hash într-un fişier temporar pe disk, având acelaşi nume cu fişierul iniţial, dar într-un alt folder.

Ultimul pas este de a trimite confirmare procesului *Master* că a terminat procesarea, sub forma unui mesaj MPI, folosind instrucţiunile din referinţa [5].

2.2. Etapa de reducere

Procesul *Master* aşteaptă să primească de la *Workeri* mesajul de confirmare că au terminat procesarea. Odată ce un proces a terminat, *Master*-ul procesează fişierul temporar corespunzător şi adaugă cuvintele în propria tabelă Hash.

Acest lucru se repetă pentru fiecare mesaj primit, rezultatul fiind unificarea tuturor cuvintelor din toate fişierele în tabela Hash a procesului Master. Acesta poate astfel să afişeze lista completă a cuvintelor şi apariţia lor în documente sau să caute un anumit cuvânt.

3. Exemplu rulare

3.1. Intrare:

O parte dintr-un text în limba engleză din fișierele de test.

When I wake up, the other side of the bed is cold. My fingers stretch out, seeking Prims warmth but finding only the rough canvas cover of the mattress. She must have had bad dreams and climbed in with our mother. Of course, she did. This is the day of the reaping. I prop myself up on one elbow. Theres enough light in the bedroom to see them. My little sister, Prim, curled up on her side, cocooned in my mothers body, their cheeks pressed together. In sleep, my mother looks younger, still worn but not so beaten-down. Prims face is as fresh as a raindrop, as lovely as the primrose for which she was named. My mother was very beautiful once, too.

3.2. Ieșire:

O parte din cuvintele dintr-un fișier de test.

<matteroffactly, {17.txt:1,2.txt:1}>

<brace, {17.txt:1,13.txt:1,1.txt:3,11.txt:1,18.txt:1,10.txt:1,2.txt:4}>

<cabin, {12.txt:1,11.txt:1,10.txt:2,15.txt:5}>

<structure, {12.txt:3,21.txt:1,17.txt:1,16.txt:1,8.txt:1,5.txt:1,11.txt:4,24.txt:1,3.txt:2,10.txt:1,2.txt:1,25.txt:4,19.txt:6}>

<diana, {23.txt:1,22.txt:1}>

<hedge, {6.txt:7,8.txt:1,13.txt:2,15.txt:1,2.txt:1,19.txt:4}>

<considerations, {6.txt:1,23.txt:7,22.txt:7,21.txt:1,14.txt:2,17.txt:1,16.txt:3,5.txt:3,11.txt:1,24.txt:3,10.txt:2,25.txt:20}>

<manifestations, {6.txt:1,16.txt:2,5.txt:2,11.txt:1,24.txt:2,10.txt:1,25.txt:4,19.txt:1}>

<began, {4.txt:16,6.txt:3,9.txt:2,12.txt:23,16.txt:2,8.txt:11,20.txt:23,5.txt:3,13.txt:49,1.txt:13,11.txt:49,24.txt:3,3.txt:1,18.txt:6,10.txt:115,15.txt:44,7.txt:88,2.txt:13,25.txt:3,19.txt:93}>

<black, {4.txt:2,6.txt:1,9.txt:7,12.txt:40,21.txt:1,17.txt:15,16.txt:2,8.txt:3,20.txt:21,13.txt:11,1.txt:33,11.txt:17,3.txt:28,18.txt:12,10.txt:34,15.txt:45,7.txt:50,2.txt:28,19.txt:91}>

4. Secvențe de cod

4.1. Mapare

```
/// if the process is ROOT
if (myRank == ROOT) {
  get_file_names(DIR_NAME, fileNames); // get file names
}
/// if the process is worker
else {
  /// receive file name
  MPI_Recv(message, strlen(message) + 1, MPI_CHAR, ROOT, tag, MPI_COMM_WORLD, &
status);
  snprintf(filePath, strlen(DIR_NAME) + 1 + strlen(message) + 1, "%s%c%s", DIR_NAME, '/',
message);
# ifdef SHOW_RECEIVED_FILE
  printf(" [%d] received file name: %s\n", myRank, filePath);
# endif
  // open file with received name
  err = fopen_s( & fp_read, filePath, "r");
  if (err) {
    perror("> Error while opening the file received.\n");
    exit(EXIT_FAILURE);
  }
  // measure time
  clock_t begin = clock();
  read_words(HT, fp_read, message);
  fclose(fp_read);
# ifdef WRITE_HT_FILE
  /// write words on file
  snprintf(filePathResult, strlen(DIR_NAME_RESULT) + 1 + strlen(message) + 1, "%s%c%s",
DIR_NAME_RESULT, '/', message);
  err_write = fopen_s( & fp_write, filePathResult, "w");
  /// verify if the file was opened successfully
  if (err_write) {
    perror("> Error while opening the file result.\n");
    exit(EXIT_FAILURE);
  }
  write_HT_to_file(HT, fp_write, false);
  fclose(fp_write);
# endif
  /// send to master that process finished the processing
  MPI_Send(message, strlen(message) + 1, MPI_CHAR, ROOT, tag, MPI_COMM_WORLD);
```

```
# ifdef SHOW_HT
  display_HT(HT);
# endif
  /// get elapsed time
  clock_t end = clock();
  elapsedSecs = double(end - begin) / CLOCKS_PER_SEC;
}
```

4.2. Reducere

```
/// if the process is ROOT
if (myRank == ROOT) {
  get_file_names(DIR_NAME, fileNames); // get file names
}
/// if the process is worker
else {
  /// receive file name
  MPI_Recv(message, strlen(message) + 1, MPI_CHAR, ROOT, tag, MPI_COMM_WORLD, &
status);
  snprintf(filePath, strlen(DIR_NAME) + 1 + strlen(message) + 1, "%s%c%s", DIR_NAME, '/',
message);
# ifdef SHOW_RECEIVED_FILE
  printf(" [%d] received file name: %s\n", myRank, filePath);
# endif
  // open file with received name
  err = fopen_s( & fp_read, filePath, "r");
  if (err) {
    perror("> Error while opening the file received.\n");
    exit(EXIT_FAILURE);
  }
  // measure time
  clock_t begin = clock();
  read_words(HT, fp_read, message);
  fclose(fp_read);
# ifdef WRITE_HT_FILE
  /// write words on file
  snprintf(filePathResult, strlen(DIR_NAME_RESULT) + 1 + strlen(message) + 1, "%s%c%s",
DIR_NAME_RESULT, '/', message);
  err_write = fopen_s( & fp_write, filePathResult, "w");
  /// verify if the file was opened successfully
  if (err_write) {
    perror("> Error while opening the file result.\n");
    exit(EXIT_FAILURE);
  }
  write_HT_to_file(HT, fp_write, false);
  fclose(fp_write);#
```

```
  endif
  /// send to master that process finished the processing
  MPI_Send(message, strlen(message) + 1, MPI_CHAR, ROOT, tag, MPI_COMM_WORLD);
# ifdef SHOW_HT
  display_HT(HT);
# endif
  /// get elapsed time
  clock_t end = clock();
  elapsedSecs = double(end - begin) / CLOCKS_PER_SEC;
}
```

4.3. Funcțiile pentru etapele anterioare

4.3.1. Tabela Hash

```
// dispersion function
int dispersion_func(char * key) {
  int i, sum;
  sum = 0;
  for (i = 0; i < strlen(key); i++) {
    sum = sum + * (key + i);
  }
  return sum % M;
}
// search a word in the hash table
TYPE_NODE * search_HT(TYPE_NODE * HT[], char * text) {
  int h = dispersion_func(text);
  TYPE_NODE * p = HT[h];
  while (p != 0) {
    if (strcmp(text, p - > word.text) == 0) {
      return p;
    }
    p = p - > next;
  }
  return 0;
}
// write the hash table records on a specified file
void write_HT_to_file(TYPE_NODE * HT[], FILE * fp, bool format) {
  for (int i = 0; i < M; i++) {
    if (HT[i] != 0) {
      TYPE_NODE * p = HT[i];
      int count = 0;
      while (p != 0) {
```

```
      count++;
      if (!format) {
       write_word_to_file(p - > word, fp);
      } else {
       write_word_to_file_formatted(p - > word, fp);
      }
      p = p - > next;
     }
    }
   }
}
// insert a word in the hash table
void insert_HT(TYPE_NODE * HT[], S_WORD w) {
  int i, foundDoc = false;
  int docNumber = 0;
  TYPE_NODE * p = new TYPE_NODE;
  p - > word = w;
  int h = dispersion_func(p - > word.text);
  if (HT[h] == 0) // it doesn't exists a record with index h
  {
    HT[h] = p; // puts the pointer
    p - > next = 0; // chaining information
  } else {
    TYPE_NODE * q = search_HT(HT, p - > word.text);
    if (q == 0) // it doesn't exists a record with the word
    {
      p - > next = HT[h]; // insert at beginning
      HT[h] = p;
    } else {
      foundDoc = false;
      docNumber = q - > word.nr_docs;
      for (i = 0; i < docNumber; i++) {
       if (strcmp(p - > word.document[0], q - > word.document[i]) == 0) {
         q - > word.frequency[i] = q - > word.frequency[i] + 1;
         foundDoc = true;
       }
      }
      if (!foundDoc) {
       if (docNumber > 0 && docNumber < MAX_NR_FILES) {
         q - > word.nr_docs = docNumber + 1;
         q - > word.frequency[docNumber] = p - > word.frequency[0];
         q - > word.document[docNumber] = new char[256];
         strcpy_s(q - > word.document[docNumber], strlen(p - > word.document[0]) + 1, p - >
word.document[0]);
       }
```

```
    }
   }
  }
}
```

4.3.2. Parsare linie din fișier

```
// read words from a file (already mapped) and inserts it in a hash table
void read_words(TYPE_NODE * HT[], FILE * fp, char * document) {
  char c;
  bool nextWord = false;
  char word[1024] = "";
  S_WORD w;
  while ((c = fgetc(fp)) != EOF) {
    if ((c == ' ' || c == '\n') && !nextWord) {
      nextWord = true;
      w = make_word(word, document); /// create new word
      insert_HT(HT, w);
      strcpy_s(word, strlen(EMPTY_CHAR) + 1, EMPTY_CHAR); /// empty read word
    } else if (c >= -1 && c <= 255 && isalpha(c)) {
      nextWord = false;
      append_char(word, tolower(c));
    }
  }
}
// reads a line from a file and returns a word struct
S_WORD parse_line(char * string, char * delimiter) {
  char * token;
  char * next_token;
  int fld = 0;
  char arr[MAX_NR_FILES * 2 + 2][NAME_SIZE] = {
    0x0
  };
  int i;
  token = strtok_s(string, delimiter, & next_token);
  while (token) {
    strcpy_s(arr[fld], strlen(token) + 1, token);
    fld++;
    token = strtok_s(NULL, delimiter, & next_token);
  }
  S_WORD word;
  int nr_docs = atoi(arr[1]);
  int freq = atoi(arr[3]);
  word = make_word(arr[0], arr[2], nr_docs, freq);
```

```
  return word;
}
```

4.3.3. Creare structură cuvânt

```
// create a new word
S_WORD make_word(char * text, char * document, int nr_of_documents = 1, int
frequency = 1) {
  S_WORD w;
  w.text = new char[strlen(text) + 1];
  strcpy_s(w.text, strlen(text) + 1, text);
  if (nr_of_documents >= 1 && frequency >= 1) {
    w.nr_docs = nr_of_documents;
    w.frequency[0] = frequency;
    w.document[0] = new char[strlen(document) + 1];
    strcpy_s(w.document[0], strlen(document) + 1, document);
  } else {
    w.nr_docs = 1;
    w.frequency[0] = 1;
    w.document[0] = new char[strlen(document) + 1];
    strcpy_s(w.document[0], strlen(document) + 1, document);
  }
  return w;
}
```

5. Concluzie

Algoritmul prezentat, de tipul MapReduce, este foarte util pentru motoarele de căutare. Chiar dacă vorbim despre căutarea fişierelor dintr-un calculator personal sau despre căutarea de pe Internet (cel mai bun exemplu fiind www.google.com), acesta va furniza datele cerute de utilizator.

Pentru obţinerea unui timp bun de execuţie s-a apelat la anumite îmbunătăţiri. S-a folosit o tabelă de hash-uri pentru indexare, timpul de acces la aceasta fiind unul foarte mic. Citirea şi parsarea fişierelor s-a efectuat în paralel, pe mai multe procese de tip Worker.

În concluzie, aplicând tehnici de paralelizare şi indexare, căutarea informaţiilor devine mai rapidă şi mai eficientă, singurul dezavantaj fiind spaţiul ocupat de indexarea fişierelor, puterea de calcul nefiind o problemă mare datorită faptului că majoritatea procesoarelor actuale au mai multe nuclee şi thread-uri pe fiecare nucleu.

6. Bibliografie

1. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. An Introduction to Information Retrieval. Cambridge University Press, Cambridge, England, Online, Cambridge UP edition, 2009.
2. Facultatea de Automatică şi Calculatoare, Universitatea Tehnică Gheorghe Asachi, Iaşi - Structuri de Date, Laboratorul 7;
3. Facultatea de Automatică şi Calculatoare, Universitatea Tehnică Gheorghe Asachi, Iaşi - Algoritmi Paraleli şi Distribuiţi, Laborator 2, 3;
4. Facultatea de Automatică şi Calculatoare, Universitatea Tehnică Gheorghe Asachi, Iaşi - Algoritmi Paraleli şi Distribuiţi, Laborator 11;
5. Daniel Thomasset, Michael Grobe, Academic Computing Services, The University of Kansas: An introduction to the Message Passing Interface (MPI) using C, la adresa: http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml.