1. Методология PCAM

Чем сложнее задача, тем сложнее ее решение. Этот незыблемый принцип совсем не радует инженеров — тех, кому необходимо разрабатывать алгоритмы их решения. Причем использование параллельных вычислений, столь необходимых в наше время и серьезно повышающих скорость решения задач, только усложняет эту задачу в разы.

Первым делом с математической точки зрения разрабатывается вычислительная схема решения задачи. Они известны уже для многих задач (как последовательные, так и параллельные), однако если дело касается какой-либо новой задачи, то этот процесс занимает существенную часть выполняемых работ. Данная процедура является фундаментом для разработки параллельного метода, но, тем не менее, к самой разработке имеет лишь косвенное отношением.

В общем, разработку параллельного алгоритма можно разделить на следующие этапы:

1) Дифференциация вычислений на самостоятельные компоненты;

2) Определение связей между компонентами вычислений;

3) Объединение компонентов в наборы или их детализация;

4) Распределение наборов компонентов между процессорами.

Для дифференциации вычислений на самостоятельные компоненты необходимо провести анализ вычислительной схемы решения задачи. Основным требованием при выполнение данного этапа является то, что компоненты обладают одинаковой вычислительной нагрузкой, а также имеют минимальные зависимости друг перед другом. Если компоненты будут независимы друг от друга в плане информации, то их можно будет выполнять одновременно в несколько потоков (поток — это некоторая последовательность команд, логически выделенная и выполняемая на некотором вычислительном элементе). Одинаковая же вычислительная нагрузка позволяет равномерно и правильно распределить процесс решения.

При дифференциации может быть выбран один из двух возможных подходов: на основе параллелизма по данным или на основе функционального параллелизма. При параллелизме по данным различные компоненты работают с различными наборами данных. Эти набору могут выделяться либо согласно ленточной схемы разделения, либо согласно блочной схемы разделения данных. В свою очередь, при функциональном параллелизме декомпозиция производится согласно выполняемым функциям метода, причем компоненты могут работать над одними и теми же данными.

Также необходимо отметить и различные подходы к проведению самой дифференциации. Можно либо провести разбиение на максимально возможное количество компонентов, либо на довольно большие компоненты, когда становится довольно понятной сама схема вычисления. Преимуществом первого подхода является эффективность вычислений, а преимуществом второго — ясная схема работы. Возможен также и третий вариант, при котором компоненты являются довольно простыми, но схема вычисления все еще является понятной.

Вопрос понятности вычислительной схемы решения является крайне важным, так как именно на основе нее производится второй этап разработки параллельного алгоритма — определение связей между компонентами вычислений. Данный этап, фактически, выполняется одновременно с первым, но, тем не менее, выделяется логически в отдельный. Происходит это потому, что дифференциация должна производится на основе анализа того, какие компоненты и как обмениваются информацией. То есть производится дифференциация, анализируются информационные зависимости, вновь проводится дифференциация и так далее.

При проведении процедуры анализа информационных зависимостей очень важно понять, какой является схема информационного взаимодействия (локальной, когда между собой зависимо небольшое число компонентов, или глобальной, когда все или почти все компоненты так или иначе зависимы друг от друга), каковы способы взаимодействия (структурные, когда взаимодействие понятно, типично и однородно, или произвольные, когда отсутствует какая-либо структура взаимодействия), а также какова схема информационной зависимости (статическая, когда все понятно на этапе разработки, или динамическая, когда зависимость определяется при выполнении вычислений). Все это помогает принять правильное решение о дифференциации вычислении на самостоятельные компоненты.

Третий этап, на котором производится объединение компонентов в наборы или их детализация, проводится для адаптации параллельного метода к числу вычислительных элементов. Объединение компонентов в наборы производится при недостатке вычислительных мощностей, причем полученные наборы, как и сами компоненты, должны обладать одинаковой вычислительной нагрузкой, а также иметь минимальные зависимости друг перед другом. Детализация же компонентов производится при избыточном числе вычислительных элементов.

Четвертый этап, а именно распределение наборов компонентов между процессорами, является необязательным и выполняется для управления нагрузкой в вычислительных системах с распределенной памятью при условии недостатка или избыточности вычислительных элементов. При успешном выполнении работ на данном этапе можно максимально эффективно использовать вычислительные элементы.

Разработанный параллельный алгоритм после подлежит программной реализации. Программный код делится по вычислительным элементам компьютерной системы в соответствии с выбранной схемой. После отладки остается лишь запустить программу на исполнение и провести вычисления. Помимо самих вычислительных элементов система должна поддерживать передачу информацию между ними в силу информационной зависимости. Важным моментом является то, что применение параллельных методов возможно и в системах с общей памятью, где задача передачи информации будет заменена на задачу доступа к общим данным, которые рассматривается потоками как единый общий ресурс. Решается вопроса доступа к общим данным, например, с помощью механизма блокировок. Например, если один поток читает некоторый набор данных, то прочитать эти данные может любой поток, но ни один не сможет их изменить. Если же один поток занял некоторые данные для их изменения, то ни один поток не сможет получить к ним доступа. Знание подобной схемы доступа является довольно важным, так как это позволит более правильно распределить нагрузку и исключить ситуации, когда несколько потоков ждут освобождения некоторого набора данных определенным потоком.

Необходимо отметить, что существует и абсолютно противоположный способ разработки параллельного метода. В таком случае опускаются все процессы проектирования, а основу берется некоторый последовательный метод, который постепенно превращается в параллельный. Данный способ позволяет получить начальный варианты параллельной работы очень быстро и с минимальными усилиями, однако результат будет сопровождаться значительно меньшей эффективностью, нежели тот, в котором проектированию уделяется чуть ли ни центральная роль.

2 Практическая часть

На языке Java разработано клиентское приложение, которое осуществляет передачу данных на сервер и получает от него ответ. Так же на языке Java реализовано серверное приложение, которое получает строку запроса формата «Параметр1:Значение1;Параметр2:Значение2 …». Далее он обрабатывает запрос и преобразует его в лексемы вида «Параметр:Значение» и записывает данные в массив. Обработав запрос, сервер оправляет ответ в виде «ПараметрN:ЗначениеN; … Параметр2:Значение2; Параметр1:Значение1».

Код сервера:

import java.io.\*;  
import java.net.\*;  
import java.util.\*;  
public class Server {  
public static void main(String[] args) throws IOException {  
System.out.println("Server BMS (povered by Krasnov)");  
BufferedReader atClient = null;  
PrintWriter toClient= null;  
ServerSocket serverSoket = null;  
Socket clientSoket = null;  
String input,output;  
input="";  
try {  
serverSoket = new ServerSocket(8080);  
System.out.println("Socked accept");  
} catch (IOException e) {  
System.out.println("Port error");  
System.exit(-1);}  
try {  
System.out.print("Waiting client...");  
clientSoket= serverSoket.accept();  
System.out.println("Client connected");  
} catch (IOException e) {  
System.out.println("Error connection");  
System.exit(-1);}  
atClient = new BufferedReader(new  
InputStreamReader(clientSoket.getInputStream()));  
toClient = new PrintWriter(clientSoket.getOutputStream(),true);  
System.out.println("Wait for messages...");  
while ((input = atClient.readLine()) != null) {  
if (input.equalsIgnoreCase("exit")) break;  
System.out.println(input);  
StringTokenizer lexema = new StringTokenizer(input, ";");  
int i=0;  
String[] otvet =new String[10];  
String out="";  
while (lexema.hasMoreTokens()) {  
otvet[i]=lexema.nextToken();  
System.out.println((i+1)+"-ая лекскма: "+otvet[i]);  
if (out==""){  
out=otvet[i];  
}else{  
out=otvet[i]+";"+out;}  
i=i+1;}  
toClient.println("Server: " + out);}  
toClient.close();  
atClient.close();  
clientSoket.close();  
serverSoket.close();}}

Код клиента:

import java.io.\*;  
import java.net.\*;  
import java.lang.String;  
public class Main {  
public static void main(String[] args) throws IOException {  
InetAddress addr = InetAddress.getByName(null);  
System.out.println("address = " + addr);  
System.out.println("Client start");  
Socket fromserver = null;  
System.out.println("Connecting to: "+addr);  
try {  
fromserver = new Socket(addr, 8080);  
} catch (IOException e){  
System.exit(-1);}  
BufferedReader atServer = new  
BufferedReader(new  
InputStreamReader(fromserver.getInputStream()));  
PrintWriter toServer = new  
PrintWriter(fromserver.getOutputStream(),true);  
try (BufferedReader clientBuffer = new  
BufferedReader(new InputStreamReader(System.in))) {  
String fuser, fserver;  
while ((fuser = clientBuffer.readLine()) != null) {  
toServer.println(fuser);  
fserver = atServer.readLine();  
System.out.println(fserver);  
if (fuser.equalsIgnoreCase("close")) break;  
if (fuser.equalsIgnoreCase("exit")) break;}  
toServer.close();  
atServer.close();  
clientBuffer.close();}  
fromserver.close();}}

Пример общения клиента и сервера представлено на рисунках 1 и 2.

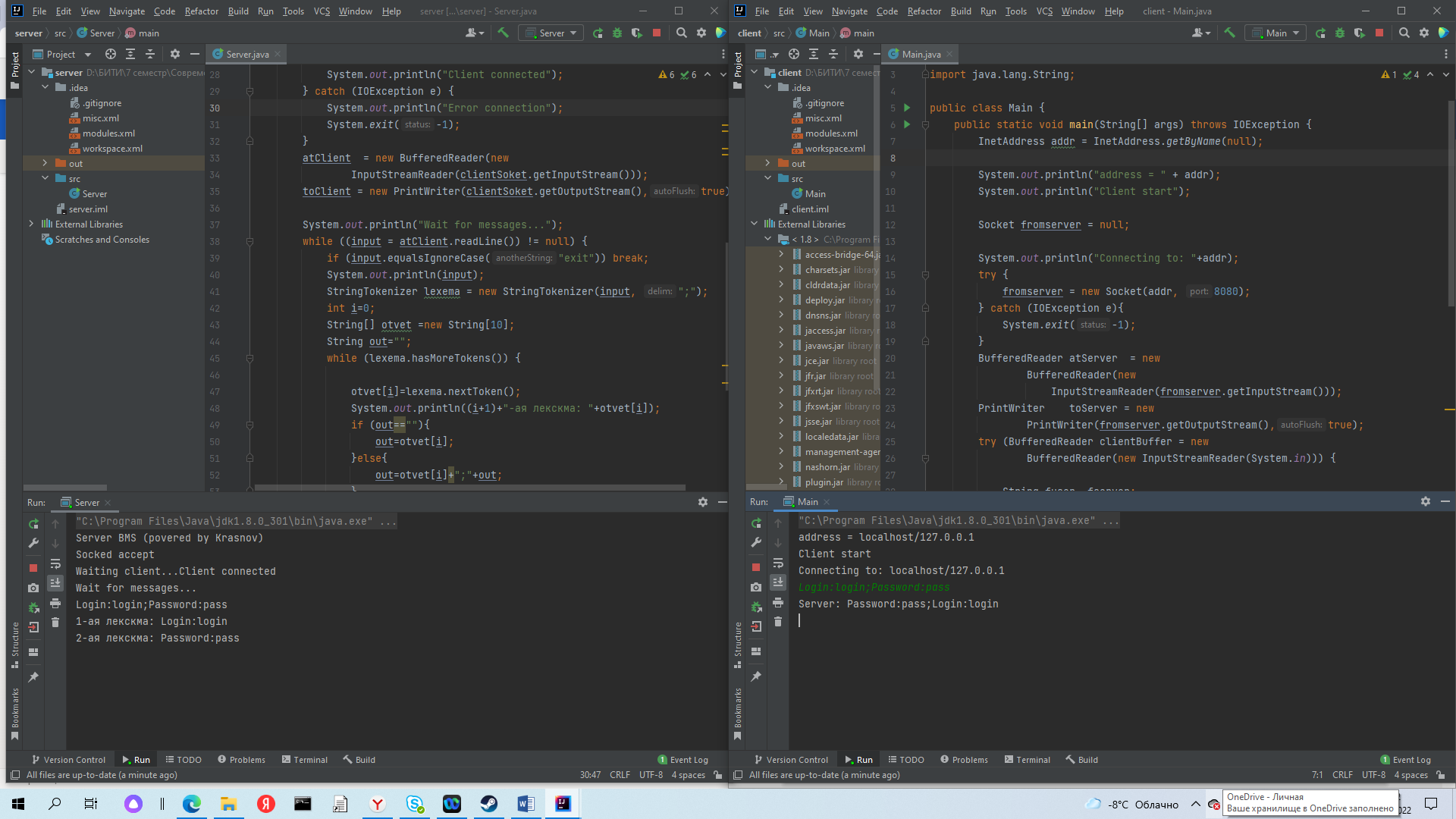


Рисунок 1 – Работа сервера

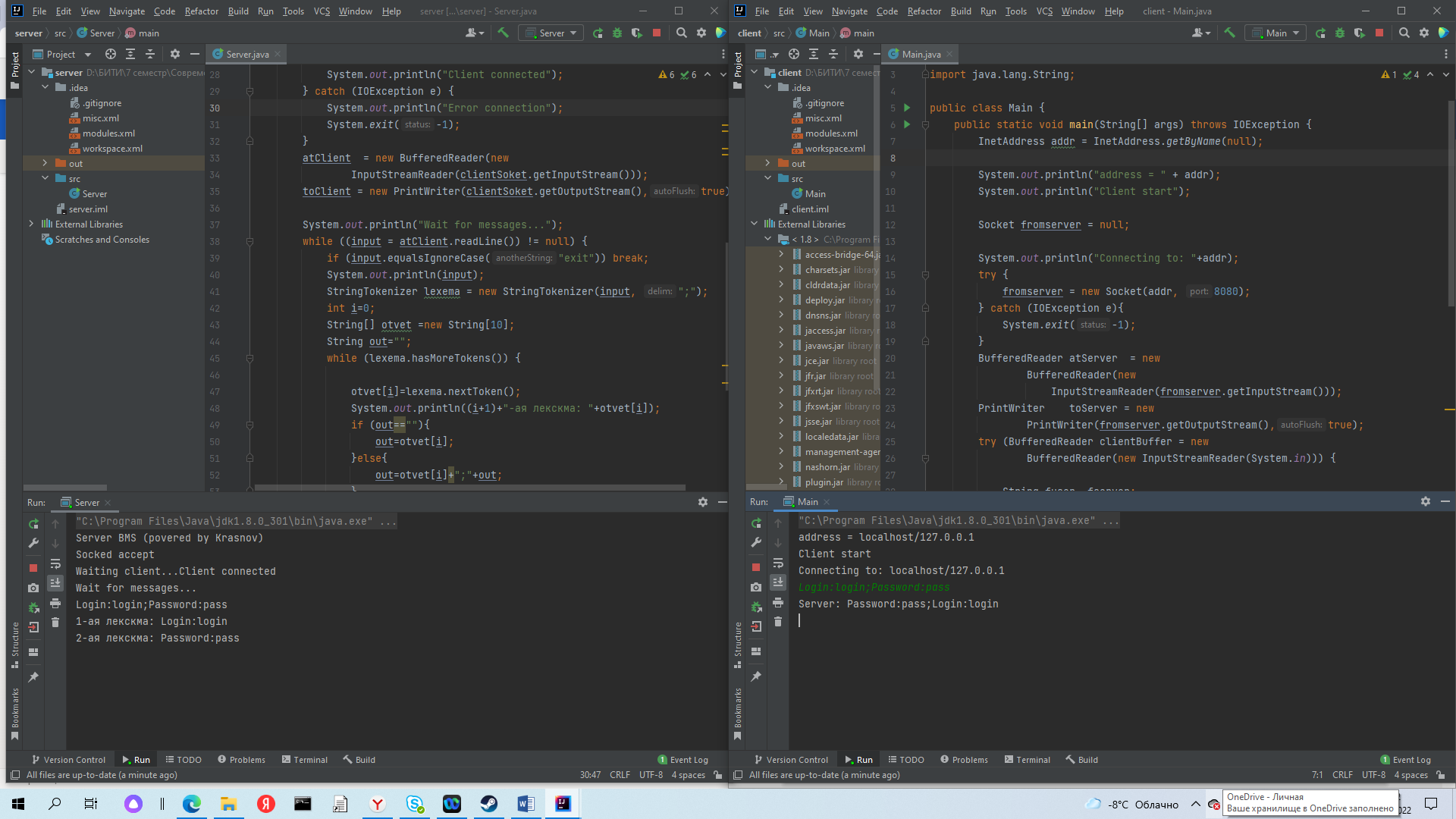


Рисунок 2 – работа клиента