package kpi.os.cours.graph;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.HashSet;

/\*\*

\* Task Graph

\* @author Pustovit Michael, pustovitm@gmail.com

\*/

public class TaskGraph {

private ArrayList<Task> tasks;

//Root element (bottom element)

private Task root;

public TaskGraph(ArrayList<Task> nodes) {

this.tasks = nodes;

root = getRootNodes().get(0);

}

private ArrayList<Task> getRootNodes() {

ArrayList<Task> rez = new ArrayList<Task>();

for (Task node : tasks) {

if (node.getOutEdges().size() == 0) {

rez.add(node);

}

}

return rez;

}

//========= Processing of critical path and b-level =========

private int recursPass(

Task curTask,

ClusterPool pool)

{

Cluster currentCl, childCl;

for (Edge t : curTask.getOutEdges()) {

currentCl = pool.getTaskCluster(curTask);

childCl = pool.getTaskCluster(t.getEnd());

//========== Calculationg of new path length===

int newPath = 0;

//if tasks in one cluster

if (currentCl != null && currentCl.equals(childCl)) {

newPath = curTask.getWeight() +

currentCl.getPathBeetwen(curTask, t.getEnd()) +

t.getEnd().getPath();

} else {

//if tasks in different clusters

newPath = curTask.getWeight() +

t.getWeight() +

t.getEnd().getPath();

}

//============================================

//If new path bigger then old one - replace old

if (curTask.getPath() < newPath)

{

curTask.setPath(newPath);

}

}

int max = curTask.getPath();

for (Edge t : curTask.getInEdges()) {

int temp = recursPass(t.getEnd(), pool);

if (max < temp) {

max = temp;

}

}

return max;

}

public int getCriticalPass(ClusterPool pool) {

for (Task task : tasks) {

task.setPath(0);

}

int max = 0;

int temp;

ArrayList<Task> roots = getRootNodes();

for (Task t : roots) {

t.setPath(t.getWeight());

temp = recursPass(t, pool);

if (temp > max) {

max = temp;

}

}

root.setPath(root.getWeight());

return max;

}

//==============================================

//========= Processing of static level =========

private void recursStaticLev(Task curTask, HashSet<Task> visited) {

visited.add(curTask);

for (Edge t : curTask.getOutEdges()) {

if (curTask.getStatPath() <

t.getEnd().getStatPath() + curTask.getWeight())

{

curTask.setStatPath(t.getEnd().getStatPath()

+ curTask.getWeight());

}

}

for (Edge t : curTask.getInEdges()) {

recursStaticLev(t.getEnd(), visited);

}

}

public void processStaticLevel() {

root.setStatPath(root.getWeight());

HashSet<Task> visited = new HashSet<Task>();

recursStaticLev(root, visited);

}

//==============================================

//================= EZ algorithm ===============

private class Pair {

private Task task;

private Edge edge;

public Pair(Task task, Edge edge) {

super();

this.task = task;

this.edge = edge;

}

public Task getTask() {

return task;

}

public Edge getEdge() {

return edge;

}

public String toString() {

return "from " + task.getNum() +

" to " + edge.getEnd().getNum() +

" W = " + edge.getWeight();

}

public Task getStart() {

return task;

}

public Task getEnd() {

return edge.getEnd();

}

}

public ClusterPool makeReduction() {

ClusterPool pool = new ClusterPool();

ClusterPool tempPool;

ArrayList<Pair> pairs = new ArrayList<Pair>();

//Get list of all edges

for (Task t : tasks) {

for (Edge e : t.getOutEdges()) {

pairs.add(new Pair(t, e));

}

}

//Sort edge list by descending of edges weight

Collections.sort(pairs, new Comparator<Pair>() {

@Override

public int compare(Pair o1, Pair o2) {

return o2.getEdge().getWeight() - o1.getEdge().getWeight();

}

});

// //???TEST OUTPUT

// for (Pair p : pairs) {

// System.out.println(p);

// }

for (Pair p : pairs) {

tempPool = pool.clone();

Cluster startCl = tempPool.getTaskCluster(p.getStart());

Cluster endCl = tempPool.getTaskCluster(p.getEnd());

//if start and end edges aren't in one cluster already

if (startCl == null || !startCl.equals(endCl)) {

tempPool.addTask(p.getStart(), p.getEnd());

}

int oldCritPath = getCriticalPass(pool);

int newCritPath = getCriticalPass(tempPool);

//if new critical path isn't worse then old one - accept

//new cluster

if (oldCritPath >= newCritPath) {

pool = tempPool.clone();

}

}

return pool;

}

//============================================

public ArrayList<Task> getTasks() {

return tasks;

}

public void setTasks(ArrayList<Task> tasks) {

this.tasks = tasks;

}

public String toString() {

String str = "";

for (Task d : tasks) {

str += "Node " + d.getNum() +

" W = " + d.getWeight() +

" Path = " + d.getPath() +

" Stat = " + d.getStatPath() + "\n";

str += "IN:\n";

for (Edge e : d.getInEdges()) {

str += e + "\n";

}

str += "OUT:\n";

for (Edge e : d.getOutEdges()) {

str += e + "\n";

}

str += "\n";

}

return str;

}

public TaskGraph clone() {

ArrayList<Task> cloneTasks = new ArrayList<Task>();

HashMap<Task, Task> corresp = new HashMap<Task, Task>();

for (Task t : tasks) {

Task tempT = new Task(t.getNum(), t.getWeight());

tempT.setPath(t.getPath());

tempT.setStatPath(t.getStatPath());

cloneTasks.add(t);

corresp.put(t, tempT);

}

for (Task t : tasks) {

Task tempT = corresp.get(t);

ArrayList<Edge> in = t.getInEdges();

for (Edge e : in) {

Task end = corresp.get(e.getEnd());

Edge temp = new Edge(e.getWeight(), end);

tempT.addInEdge(temp);

}

ArrayList<Edge> out = t.getOutEdges();

for (Edge e : out) {

Task end = corresp.get(e.getEnd());

Edge temp = new Edge(e.getWeight(), end);

tempT.addOutEdge(temp);

}

}

return new TaskGraph(cloneTasks);

}

}

package kpi.os.cours.graph;

import java.util.ArrayList;

import java.util.TreeSet;

/\*\*

\*

\* @author Pustovit Michael, pustovitm@gmail.com

\*/

public class ClusterPool {

ArrayList<Cluster> clusters;

public ClusterPool() {

clusters = new ArrayList<Cluster>();

}

/\*\*

\* Adding of task to cluster. Key task point in which cluster Value key

\* will be put. If any container doesn't contain Key task - new cluster

\* will be created.

\* @param key key task

\* @param value value task

\*/

public void addTask(Task key, Task value) {

Cluster keyCl = getTaskCluster(key);

Cluster valueCl = getTaskCluster(value);

//If tasks aren't in any clusters

if (keyCl == null && valueCl == null) {

Cluster container = new Cluster();

container.addTask(key);

container.addTask(value);

clusters.add(container);

} else

//If key task in cluster, value - isn't in cluster

if (keyCl != null && valueCl == null) {

keyCl.addTask(value);

} else

//If current task isn't in cluster, child task isn't in cluster

if (keyCl == null && valueCl != null) {

valueCl.addTask(key);

}

//If tasks are in one cluster - all edges between them are zeroed

if (keyCl != null && valueCl != null && !keyCl.equals(valueCl)) {

Cluster container = mergeClusters(keyCl, valueCl);

clusters.remove(keyCl);

clusters.remove(valueCl);

clusters.add(container);

}

}

public Cluster mergeClusters(Cluster cl1, Cluster cl2) {

Cluster rez = new Cluster();

TreeSet<Task> sum = new TreeSet<Task>();

sum.addAll(cl1.getTasks());

sum.addAll(cl2.getTasks());

rez.setTasks(sum);

return rez;

}

/\*\*

\* Searching cluster which contains task Task. If such cluster doesn't

\* exist - return NULL

\* @param task

\* @return cluster which contains this task

\*/

public Cluster getTaskCluster(Task task) {

Cluster rez = null;

for (Cluster c : clusters) {

if (c.contains(task)) {

rez = c;

break;

}

}

return rez;

}

public boolean contains(Task task) {

Cluster cl = null;

for (Cluster c : clusters) {

if (c.contains(task)) {

cl = c;

break;

}

}

return cl != null;

}

public ClusterPool clone() {

ArrayList<Cluster> cloneClust = new ArrayList<Cluster>();

for (Cluster cl : clusters) {

cloneClust.add(cl.clone());

}

ClusterPool clone = new ClusterPool();

clone.setClusters(cloneClust);

return clone;

}

public void setClusters(ArrayList<Cluster> clusters) {

this.clusters = clusters;

}

public ArrayList<Cluster> getClusters() {

return clusters;

}

public String toString() {

String str = "";

for (Cluster c : clusters) {

str += c;

}

return str;

}

}

package kpi.os.cours.graph;

import java.util.ArrayList;

public class Task implements Comparable<Task> {

private int num; //task number

private ArrayList<Edge> outEdges, inEdges;

private int weight; //processing level

private int path; //b-level

private int statPath; //static level

public Task(int num, int weight) {

this.num = num;

this.weight = weight;

outEdges = new ArrayList<Edge>();

inEdges = new ArrayList<Edge>();

path = 0;

}

public void addInEdge(Edge edge) {

inEdges.add(edge);

}

public void addOutEdge(Edge edge) {

outEdges.add(edge);

}

public int getNum() {

return num;

}

public ArrayList<Edge> getInEdges() {

return inEdges;

}

public ArrayList<Edge> getOutEdges() {

return outEdges;

}

public int getWeight() {

return weight;

}

public int getPath() {

return path;

}

public void setPath(int path) {

this.path = path;

}

public int getStatPath() {

return statPath;

}

public void setStatPath(int statPath) {

this.statPath = statPath;

}

public String toString() {

return Integer.toString(num) + "\n" + Integer.toString(weight);

}

@Override

public int compareTo(Task o) {

Task o1 = this;

Task o2 = o;

if (o2.getStatPath() - o1.getStatPath() != 0) {

return o2.getStatPath() - o1.getStatPath();

} else {

return o1.getNum() - o2.getNum();

}

}

}

package kpi.os.cours.graph;

public class Edge {

private int weight;

private Task end;

public Edge(int weight, Task end) {

super();

this.weight = weight;

this.end = end;

}

public int getWeight() {

return weight;

}

public Task getEnd() {

return end;

}

public String toString() {

return "to " + end.getNum() + " W = " + weight;

}

}

package kpi.os.cours.graph;

import java.util.Comparator;

import java.util.TreeSet;

/\*\*

\* Cluster

\* @author Pustovit Michael, pustovitm@gmail.com

\*/

public class Cluster {

private TreeSet<Task> tasks;

public Cluster() {

tasks = new TreeSet<Task>(new Comparator<Task>() {

@Override

public int compare(Task o1, Task o2) {

if (o2.getStatPath() - o1.getStatPath() != 0) {

return o2.getStatPath() - o1.getStatPath();

} else {

return o1.getNum() - o2.getNum();

}

}

});

}

public boolean contains(Task task) {

return tasks.contains(task);

}

public void addTask(Task task) {

tasks.add(task);

}

public int getOveralWeight() {

int sum = 0;

for (Task t : tasks) {

sum += t.getWeight();

}

return sum;

}

/\*\*

\* Getting sum weight of tasks which will have been processed before task

\* "task"

\* @param task task - limiter

\* @return sum weight of tasks which will have been processed before task

\* "task". If task isn't in this cluster - return -1

\*/

public int getAboveWeight(Task task) {

if (!tasks.contains(task))

return -1;

int sum = 0;

for (Task t : tasks) {

if ( (t.getStatPath() > task.getStatPath()) ||

(t.getStatPath() == task.getStatPath()

&& t.getNum() > task.getNum()) )

{

sum += t.getWeight();

}

}

return sum;

}

public int getPathBeetwen(Task t1, Task t2) {

if (!tasks.contains(t1) || !tasks.contains(t2)) {

return -1;

}

int sum = 0;

for (Task t : tasks) {

if ( t1.compareTo(t) < 0

&& t2.compareTo(t) > 0)

{

sum += t.getWeight();

}

}

return sum;

}

public Cluster clone() {

TreeSet<Task> cloneTasks = (TreeSet<Task>) tasks.clone();

Cluster clone = new Cluster();

clone.setTasks(cloneTasks);

return clone;

}

public void setTasks(TreeSet<Task> tasks) {

this.tasks = tasks;

}

public TreeSet<Task> getTasks() {

return tasks;

}

public String toString() {

String str = "Cluster: ";

for (Task t : tasks) {

str += t.getNum() + " ";

}

return str + "\n";

}

}

package kpi.os.cours.emulating;

import java.util.ArrayList;

import kpi.os.cours.forms.DiagramData;

import kpi.os.cours.forms.DiagramLineData;

import kpi.os.cours.graph.ClusterPool;

import kpi.os.cours.graph.TaskGraph;

public class Controller {

private ProcessorPool procPool;

public Controller(TaskGraph graph, ClusterPool genPool) {

procPool = new ProcessorPool(graph, genPool);

//TEST OUTPUT

System.out.println(procPool);

}

public DiagramData start() {

Processor proc = procPool.getProcessor(0);

int marker = 0;

procPool.getProcessor(0).giveMarker();

int i = 0;

while (!procPool.allFinished()) {

proc.makeStep();

proc = procPool.nextProc();

i++;

if (i % procPool.getProcCount() == 0) {

i = 0;

if (procPool.getProcessor(marker).takeMarker()) {

marker = procPool.getNextProcNum(marker);

procPool.getProcessor(marker).giveMarker();

}

}

}

ArrayList<DiagramLineData> lines = new ArrayList<DiagramLineData>();

for (Processor p : procPool.getProcessors()) {

lines.add(new DiagramLineData("P" + p.getNum(), p.getHistory()));

}

return new DiagramData(

procPool.getOveralTime() + 2, 1, lines);

}

}

package kpi.os.cours.emulating;

import kpi.os.cours.graph.Edge;

import kpi.os.cours.graph.Task;

public class Pair {

public Edge edge;

public Task task;

private int startTime;

public Pair(Edge edge, Task task) {

this.edge = edge;

this.task = task;

}

public void setStartTime(int startTime) {

this.startTime = startTime;

}

public int getStartTime() {

return startTime;

}

}

package kpi.os.cours.emulating;

import java.util.ArrayList;

import java.util.HashSet;

import kpi.os.cours.forms.Interval;

import kpi.os.cours.forms.IntervalType;

import kpi.os.cours.graph.Edge;

import kpi.os.cours.graph.Task;

public class Processor {

private int num; //Processor number

//Очередь вычислений

private ArrayList<Task> procQueue; //Очередь выполнения заданий

private Task currentTask; //Текущее задание

private int remainProcTime = 0; //Время до конца текущего задания

//Очередь передач

private ArrayList<Pair> sendQueue; //Очередь выполнения пересылок

private Pair currentTrans; //Текущая пересылка

private int remainTransTime = 0; //Время до конца текущей пересылки

private int curTime = 0; //Время моделирования

private boolean marker = false; //Наличие маркера

private boolean transferable = true;//Может ли участвовать в пересылке

private HashSet<Task> receivedData;

ArrayList<Interval> history; //История всех событий

private ProcessorPool procPool;

private HashSet<Task> allTasks;

private boolean started = false;

private HashSet<Task> buffer;

public Processor(int num, ArrayList<Task> tasks, ProcessorPool procPool) {

this.num = num;

this.procQueue = tasks;

sendQueue = new ArrayList<Pair>();

history = new ArrayList<Interval>();

curTime = 0;

receivedData = new HashSet<Task>();

allTasks = new HashSet<Task>(tasks);

this.procPool = procPool;

buffer = new HashSet<Task>();

}

public void makeStep() {

started = true;

manageTrans();

manageProc();

if (marker) {

history.add(new Interval(curTime, 0, IntervalType.MARKER, ""));

}

if (buffer.size() > 0) {

receivedData.addAll(buffer);

buffer = new HashSet<Task>();

}

curTime++;

}

//===========Task service=============

/\*\*

\* Task service

\*/

public void manageProc() {

//if processor is free

if (remainProcTime == 0) {

tryScheduleTask();

}

if (remainProcTime > 0){

remainProcTime--;

//if task is finished in this moment

if (remainProcTime == 0) {

for (Edge e : currentTask.getOutEdges()) {

if (!allTasks.contains(e.getEnd())) {

sendQueue.add(new Pair(e, currentTask));

} else {

receivedData.add(currentTask);

}

}

currentTask = null;

}

}

}

private void tryScheduleTask() {

//if we have some tasks in queue

if (procQueue.size() > 0) {

Task tempT = procQueue.get(0);

if (isTaskReady(tempT)) {

setTask(tempT);

procQueue.remove(0);

}

}

}

/\*\*

\* Task is ready if all required data is received

\* @param task exemined task

\* @return ready of task

\*/

private boolean isTaskReady(Task task) {

for (Edge t : task.getInEdges()) {

if (!receivedData.contains(t.getEnd())) {

return false;

}

}

return true;

}

/\*\*

\* Start of task processing

\* @param task processing task

\*/

private void setTask(Task task) {

remainProcTime = task.getWeight();

currentTask = task;

history.add(new Interval(curTime, curTime + task.getWeight(),

IntervalType.PROCESSING, Integer.toString(task.getNum())));

}

//======================================

//=============Trans service============

/\*\*

\* Transition service

\*/

public void manageTrans() {

//not transfers in this moment

if (remainTransTime == 0 && marker) {

tryScheduleSend();

}

if (remainTransTime > 0) {

remainTransTime--;

if (remainTransTime == 0) {

Processor receiver = procPool.getTaskProcessor(

currentTrans.edge.getEnd());

receiver.forceReceive(currentTrans);

currentTrans = null;

}

}

}

private void tryScheduleSend() {

if (sendQueue.size() > 0) {

sendQueue.get(0).setStartTime(curTime);

Edge edge = sendQueue.get(0).edge;

remainTransTime = edge.getWeight();

history.add(new Interval(

curTime, curTime + edge.getWeight(),

IntervalType.SENDING,

Integer.toString(edge.getEnd().getNum())));

currentTrans = sendQueue.get(0);

sendQueue.remove(0);

}

}

public void forceReceive(Pair transition) {

//remainTransTime = transition.edge.getWeight();

history.add(new Interval(

transition.getStartTime(),

transition.getStartTime() + transition.edge.getWeight(),

IntervalType.RECEIVING,

Integer.toString(transition.task.getNum())));

if (procPool.getTaskProcessor(transition.task).getNum() > num) {

receivedData.add(transition.task);

} else {

buffer.add(transition.task);

}

//currentTrans = transition;

}

//======================================

public void giveMarker() {

marker = true;

}

public boolean takeMarker() {

if (remainTransTime == 0 && marker) {

marker = false;

return true;

}

return false;

}

public boolean isFinished() {

if (procQueue.size() == 0

&& sendQueue.size() == 0

&& started

&& currentTask == null)

{

return true;

} else {

return false;

}

}

/\*\*

\* @return If can processor transfer something

\*/

public boolean isTransferable() {

return transferable;

}

public int getNum() {

return num;

}

public ArrayList<Interval> getHistory() {

return history;

}

public int getCurTime() {

return curTime;

}

public String toString() {

String str = "Processor #" + num + (isFinished()?" FINISHED":" WORK")

+ (marker?" M":"") + "\n";

str += "Time = " + curTime + "\n";

str += "Cur task: " + ((currentTask == null)?"-":currentTask.getNum()) + "\n";

str += "Remain proc time: " + remainProcTime + "\n";

str += "Proc queue: ";

for (Task t : procQueue) {

str += t.getNum() + " ";

}

str += "\nCur trans: " +

((currentTrans == null)?"-":currentTrans.edge.getEnd().getNum()) + "\n";

str += "Remain trans time: " + remainTransTime + "\n";

str += "Trans queue: ";

for (Pair t : sendQueue) {

str += t.edge.getEnd().getNum() + " ";

}

return str;

}

}

package kpi.os.cours.emulating;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import kpi.os.cours.graph.Cluster;

import kpi.os.cours.graph.ClusterPool;

import kpi.os.cours.graph.Task;

import kpi.os.cours.graph.TaskGraph;

public class ProcessorPool {

HashMap<Cluster, Processor> table;

private ClusterPool pool;

private TaskGraph graph;

private ArrayList<Processor> processors;

private int overalTime = 0;

private int currProc;

public ProcessorPool(TaskGraph graph, ClusterPool genPool) {

this.graph = graph;

this.pool = getFullPool(genPool);

table = new HashMap<Cluster, Processor>();

int i = 0;

for (Cluster cl : pool.getClusters()) {

ArrayList<Task> tasks = new ArrayList<Task>();

for (Task t : cl.getTasks()) {

tasks.add(t);

}

table.put(cl, new Processor(i, tasks, this));

i++;

}

processors = new ArrayList<Processor>(table.values());

currProc = 0;

Collections.sort(processors, new Comparator<Processor>() {

@Override

public int compare(Processor o1, Processor o2) {

return o1.getNum() - o2.getNum();

}

});

}

private ClusterPool getFullPool(ClusterPool genPool) {

ClusterPool fullPool = genPool.clone();

for (Task t : graph.getTasks()) {

if (!genPool.contains(t)) {

fullPool.addTask(t, t);

}

}

return fullPool;

}

/\*\*

\* Processor which process "task"

\* @param task -

\* @return Processor which process "task"

\*/

public Processor getTaskProcessor(Task task) {

for (Cluster cl : pool.getClusters()) {

if (cl.contains(task)) {

return table.get(cl);

}

}

return null;

}

public Processor getProcessor(int i) {

return processors.get(i);

}

public int getProcCount() {

return table.size();

}

public Processor getCurrentProc() {

return processors.get(currProc);

}

public int getNextProcNum(int currProc) {

int next = currProc + 1;

if (next == processors.size()) {

next = 0;

overalTime++;

}

return next;

}

public boolean allFinished() {

for (Processor p : processors) {

if (!p.isFinished()) {

return false;

}

}

return true;

}

public Processor nextProc() {

currProc = getNextProcNum(currProc);

return processors.get(currProc);

}

public String toString() {

String str = "Processors:\n";

for (Processor p : processors) {

str += p + "\n\n";

}

return str;

}

public ArrayList<Processor> getProcessors() {

return processors;

}

public int getOveralTime() {

return overalTime;

}

public TaskGraph getGraph() {

return graph;

}

public ClusterPool getPool() {

return pool;

}

}