**ДОДАТОК**

**Лістинг частини програмного коду**

**package** org.mazur.stplan.sim

**import** org.jfree.data.gantt.Task;

/\*\*

\*

\* @version: $Id$

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

**class** SimManager {

SimState simState

**public** **int** getTime() { **return** simState.time }

**private** String taskName(**final** **def** p, **final** **def** task) { **return** p.value.name + "($task.value.name)" }

**private** **void** log(**def** message) { simState.log(message) }

**private** **def** resolvePath(**def** task, **def** targetProcessor) {

**def** paths = [:]

task.inputs().**each** {

**def** sourceProcessor = simState.getTaskProcessor(**it**.source)

paths[**it**.source] = simState.findPath(sourceProcessor, targetProcessor)

}

**return** paths

}

**private** **def** planTask(**def** task) {

log "Plan $task"

**def** selection = simState.selectProcessor(task)

**def** p = selection[0], delay = selection[1]

log "Processor: $p"

**if** (!p) { **return** **null** }

**def** diagramParts = []

// make transitions

**def** messagePaths = resolvePath(task, p)

log "Transitions: $messagePaths"

**int** planTime = simState.time

messagePaths.**each** { **def** sourceTask, **def** path ->

**if** (path.size() < 2) { **return** }

**int** pt = simState.getTaskFinishTime(sourceTask)

**int** weight = (sourceTask.connections().**find** { **it**.target == task }).value **as** **int**

log "Transitions from $sourceTask ($weight)"

**def** source = path.remove(0)

**while** (source != p) {

**def** target = path.remove(0)

log " --> to $target"

**def** rt = simState.resolveTransition(source, target, pt, weight)

pt = rt[0]

diagramParts += **new** TimeDescriptor(

type : "$source.value.name",

task : **new** Task("$sourceTask.value.name - $task.value.name ($target.value.name)", **new** Date(pt), **new** Date(pt + rt[1]))

)

source = target

pt += rt[1]

}

**if** (pt > planTime) { planTime = pt }

}

**if** (delay) { planTime += delay }

**def** taskWrapper = simState.useNode(task, p, planTime)

diagramParts += **new** TimeDescriptor(

type : "$p.value.name",

task : **new** Task(taskName(p, task), **new** Date(taskWrapper.startTime), **new** Date(taskWrapper.finishTime))

)

**return** diagramParts

}

**public** **def** **next**() {

simState.nextStep()

**def** task = simState.nextTask()

**def** out = []

**while** (task) {

**def** planResult = planTask(task)

**if** (!planResult) { **break** }

simState.confirmTaskPlanned task

out += planResult

task = simState.nextTask()

}

**return** out

}

**public** **def** all() {

**def** result = []

**def** stepResult = **next**()

**while** (stepResult) {

result += stepResult

stepResult = **next**()

}

**return** result

}

}

**class** TimeDescriptor {

String type

org.jfree.data.gantt.Task task

}

package org.mazur.stplan

import org.mazur.stplan.model.ModelUtil;

import org.mazur.stplan.model.SystemParameters

import org.mazur.stplan.model.TaskVertex

/\*\*

\* @version: $Id$

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

enum State {

/\*\* State instance. \*/

INSTANCE

/\*\* Current queue algorithm. \*/

QueueAlg queueAlg = QueueAlg.CRITICAL\_DOWN

/\*\* Select processor algorithm. \*/

SelectProcessorAlg selectProcessorAlg = SelectProcessorAlg.PRIORITY

def processorsAlg = { p, m ->

ModelUtil mu = new ModelUtil()

def result = mu.getProcessors(p, m)

def message = new StringBuilder()

result.each { message << "$it.value (${it.connections().size()})\n" }

return [result, message.toString()]

}

SystemParameters sysParams

@Override

String toString() { return "State: $queueAlg / $selectProcessorAlg" }

}

enum QueueAlg {

CRITICAL\_DOWN("За пронормованими критичними шляхами", { p, m ->

ModelUtil mu = new ModelUtil()

def map = new HashMap(mu.criticalTaskPaths(p, m, true, true))

def mapL = new HashMap(mu.criticalTaskPaths(p, m, false, true))

int maxPath = map.values().max(), maxPathL = mapL.values().max()

def f = { v -> (map[v] / maxPath) + (mapL[v] / maxPathL) }

def q = map.keySet().sort { v1, v2 -> f(v2) <=> f(v1) }

def message = new StringBuilder()

q.each { message << "$it -> ${map[it]}/${mapL[it]}/${f(it)}\n" }

return [q, message.toString()]

}),

READY("За часом готовності", { p, m ->

ModelUtil mu = new ModelUtil()

def q = new ArrayList(mu.getAllVertexes(p, m))

def message = new StringBuilder()

q.each { message << "$it \n" }

return [q, message]

}),

CRITICAL\_UP("За критичним шляхом від початку графа", { p, m ->

ModelUtil mu = new ModelUtil()

def map = new HashMap(mu.criticalTaskPaths(p, m, false, false))

def q = map.keySet().sort { v1, v2 ->

int res = map[v1] <=> map[v2]

res == 0 ? v2.value.volume <=> v1.value.volume : res

}

def message = new StringBuilder()

q.each { message << "$it -> ${map[it]}/${it.value.volume}\n" }

return [q, message]

})

def formQueue

def caption

private QueueAlg(String name, def alg) {

this.caption = name

this.formQueue = alg

}

}

enum SelectProcessorAlg {

PRIORITY("За приорітетами ", { def params ->

return [params.freeProcessors.max { a, b -> a.connections().size() <=> b.connections().size() }, 0]

}),

NEAR\_WITH\_ALL("Сусіднє призначення з у рахуванням усіх процесорів", { def params ->

if (params.firstStep) { return PRIORITY.select(params) }

def allP = []; allP += params.freeProcessors; allP += params.usedProcessors.keySet()

def calculateForP = { p ->

def upT = params.usedProcessors[p]

int startTime = upT ? upT : 0

int sendTime = (params.task.inputs().collect { [params.processorSelector(it.source), it.value as int] }).inject(0) { sum, value ->

sum += params.distanceSelector(p, value[0]) \* value[1]

}

println "$p -> $startTime, $sendTime"

return [startTime, sendTime].max()

}

def result = allP.min { pa, pb -> calculateForP(pa) <=> calculateForP(pb) }

return [result, params.usedProcessors[result]]

})

def caption

def select

private SelectProcessorAlg(def caption, def select) {

this.caption = caption

this.select = select

}

}

package org.mazur.stplan.sim

import org.mazur.stplan.State

import org.mazur.stplan.model.SystemParameters

/\*\*

\* Simulation state.

\* @version: $Id$

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

class SimState {

/\*\* Tasks queue. \*/

def tasksQueue

/\*\* Completed tasks. \*/

private def completedTasks = [:]

/\*\* Simultation time line. \*/

private TimeLine timeLine = new TimeLine()

/\*\* Collection of free processors. \*/

def freeProcessors, usedProcessors = []

private def tasksMap = [:]

private def routes = [:]

private void log(def what) { println "[$time]: $what" }

private NodeState[] nodeStates

def getTaskProcessor(def task) { return tasksMap[task] }

void setFreeProcessors(def freeProcessors) {

this.freeProcessors = freeProcessors

buildRouteTables freeProcessors

nodeStates = new NodeState[freeProcessors.size()]

nodeStates.length.times {

def state = new NodeState(linkStates : new Object[State.INSTANCE.sysParams.linksCount])

nodeStates[it] = state

state.linkStates.length.times { state.linkStates[it] = [] }

}

}

private def testNextTask() {

return tasksQueue.find {

def inp = it.inputs()

inp.empty || inp.every { completedTasks[it.source] }

}

}

def nextTask() {

return testNextTask()

}

void confirmTaskPlanned(def t) { tasksQueue.remove(t) }

int getTime() { return timeLine.time }

void complete(def task) {

log "$task is completed"

completedTasks[task] = time

releaseNode tasksMap[task]

}

int getTaskFinishTime(def task) { return completedTasks[task] }

void nextStep() {

log "Next step. Free P: $freeProcessors. Tasks: $tasksQueue."

log "Time line: $timeLine"

boolean hasReadyTasks = false

int counter = 400

while (!hasReadyTasks) {

timeLine.getCompletedTasksAndChangeTime().each { complete it.task }

def nt = testNextTask()

hasReadyTasks = (nt != null || tasksQueue.empty)

if (!hasReadyTasks) { counter-- }

if (!counter) { throw new RuntimeException("Cycle forever!!!") }

}

}

def useNode(final def task, final def p, final int time) {

usedProcessors += p; freeProcessors -= p

log "$task starts at $time"

tasksMap[task] = p

return timeLine.addTask(time, task)

}

void releaseNode(final def p) {

freeProcessors += p; usedProcessors -= p

}

def selectProcessor(def task) {

def usedProcessorsMap = [:]

def processorSelector = { def t -> return tasksMap[t] }

this.usedProcessors.each {

usedProcessorsMap[it] = timeLine.getProcessorReleaseTimeDelay(it, processorSelector)

}

def parameters = [

freeProcessors : freeProcessors,

task : task,

usedProcessors : usedProcessorsMap,

distanceSelector : { def a, b -> return findPath(a, b).size() - 1 },

processorSelector : processorSelector,

firstStep : tasksMap.isEmpty()

]

return State.INSTANCE.selectProcessorAlg.select(parameters)

}

private int minNotVisited(final PathMetric[] row, final BitSet visited) {

int n = visited.nextClearBit(0), minValue = row[n].d

int s = n

for (int i in s..<row.length) {

if (visited.get(i)) { continue; }

if ((minValue > row[i].d && row[i].d >= 0) || minValue < 0) {

minValue = row[i].d

n = i

}

}

return visited.get(n) ? -1 : n

}

private PathMetric[][] minDistances(final boolean[][] matrix) {

int n = matrix.length

PathMetric[][] result = new PathMetric[n][n]

for (int i in 0 .. n - 1) {

for (int j in 0 .. n - 1) {

result[i][j] = new PathMetric(d : (i == j ? 0 : matrix[i][j] ? 1 : -1), nextVertex : j)

}

}

def sourceDistances = { int srcVertexIndex ->

BitSet visited = new BitSet(n)

PathMetric[] row = result[srcVertexIndex]

int haveToFind = n - 1

row[srcVertexIndex].d = 0

visited.set(srcVertexIndex)

n.times() {

if (!haveToFind) { return }

// i - current vertex index

int i = minNotVisited(row, visited)

matrix[i].eachWithIndex() { mv, index ->

// if connected -> modify distance

if (mv && i != index && !visited.get(index)) {

int d = row[i].d + 1

PathMetric pm = row[index]

if (pm.d == -1 || pm.d > d) {

pm.nextVertex = i

pm.d = d

}

}

}

visited.set(i)

haveToFind--

}

}

int k = n.div(2)

if (n % 2) { k++; }

Thread t1 = new CThread(closure : {

k.times() { sourceDistances(it) }

})

Thread t2 = new CThread(closure : {

(n - k).times() { sourceDistances(k + it) }

})

t1.start()

t2.start()

t1.join()

t2.join()

return result

}

private void buildRouteTables(def processors) {

def visited = new HashSet()

boolean[][] matrix = new boolean[processors.size()][processors.size()]

processors.each {

if (visited.contains(it)) { return }

def stack = []

stack.push it

while (stack) {

def s1 = stack.pop()

if (visited.contains(s1)) { continue }

visited += s1

s1.connections().each() {

def s2 = it.target

int i1 = s1.value.number, i2 = s2.value.number

matrix[i1][i2] = matrix[i2][i1] = true

stack.push s2

}

}

}

// FIXME incorrect min distances!!!!

PathMetric[][] routingTable = minDistances(matrix)

routingTable.each {

println it

}

for (int i in 0..<routingTable.length) {

def source = processors.find { it.value.number == i }

for (int j in 0..<routingTable[0].length) {

if (i == j) { continue }

def target = processors.find { it.value.number == j }

def nextV = processors.find { it.value.number == routingTable[i][j].nextVertex }

this.routes[new PathPair(source : source, target : target)] = nextV

}

}

}

public def getNextVertex(def source, def target) { return routes[new PathPair(source : source, target : target)] }

public def findPath(def source, def target) {

def path = [source]

def src = source

while (src != target) {

src = getNextVertex(src, target)

path += src

}

return path

}

public def resolveTransition(def src, def dst, int planTime, int weigth) {

int duration = (int)(weigth \* State.INSTANCE.sysParams.chanelSpeed)

NodeState srcState = nodeStates[src.value.number], dstState = nodeStates[dst.value.number]

log "Source: $srcState"

log "Destination: $dstState"

boolean adjust = true

int pt = planTime, srcLink, dstLink

while (adjust) {

def srcResolution = srcState.resolveStartTime(pt, duration, dst, false), dstResolution = dstState.resolveStartTime(pt, duration, src, true)

srcLink = srcResolution[1]

dstLink = dstResolution[1]

println "Resolution: $srcResolution vs $dstResolution"

pt = srcResolution[0] > dstResolution[0] ? srcResolution[0] : dstResolution[0]

adjust = (srcResolution[0] != dstResolution[0])

}

srcState.useLink(srcLink, pt, duration, dst, false)

dstState.useLink(dstLink, pt, duration, src, true)

return [pt, duration]

}

}

class PathPair {

def source, target

int hashCode() { return source.hashCode() + target.hashCode() }

boolean equals(final Object o) {

return source.equals(o.source) && target.equals(o.target)

}

}

class PathMetric {

int nextVertex

int d

String toString() { return "$d/$nextVertex" }

}

class CThread extends Thread {

def closure

void run() { closure() }

}

class NodeState {

def linkStates

/\*\*

\* @param periods

\* @param startTime

\* @param duration

\* @param node

\* @param direction true for incoming

\* @return

\*/

private def getClosestTime(final def periods, final int startTime, final int duration, final def node, final boolean direction) {

int st = startTime, et = startTime + duration

boolean obligative = false

boolean directionBusy = false

periods.each {

def range = it[0]..it[1], range2 = st..et

boolean intersection = range.contains(st) || range.contains(et) || range2.contains(it[0]) || range2.contains(it[1])

boolean sameNode = it[2] == node

if (intersection && sameNode && it[2] == direction) { directionBusy = true }

if (State.INSTANCE.sysParams.duplex) {

if (intersection && directionBusy) {

st = it[1]

et = st + duration

obligative = true

}

} else {

if (intersection) {

st = it[1]

et = st + duration

if (sameNode) { obligative = true } // for half-duplex

}

}

}

return [st, obligative]

}

public def resolveStartTime(final int startTime, final int duration, final def node, final boolean direction) {

int i = 0

def tms = linkStates.collect {

def res = getClosestTime(it, startTime, duration, node, direction)

res += (i++)

return res

}

def result = tms.min { a, b ->

boolean o1 = a[1], o2 = b[1]

if (!(o1 ^ o2)) { return a[0] <=> b[0] }

return o1 ? -1 : 1

}

return [result[0], result[2]]

}

public void useLink(final int linkIndex, final int startTime, final int duration, final def node, final boolean direction) {

int st = startTime, et = startTime + duration

def linkState = linkStates[linkIndex]

int insertIndex = -1

for (int i = 0; i < linkState.size(); i++) {

if (startTime < linkState[i][0]) { insertIndex = i; break }

}

def data = [startTime, et, node, direction]

if (insertIndex != -1) {

linkState.add insertIndex, data

} else {

linkState.add(data)

}

}

String toString() {

def res = new StringBuilder()

linkStates.each { res << "$it\n" }

return res

}

}

package org.mazur.stplan.sim

import org.mazur.stplan.State

import org.mazur.stplan.model.TaskVertex;

/\*\*

\*

\* @version: $Id$

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

class TimeLine {

/\*\* Current time. \*/

int time = 0

/\*\* Time line. \*/

private TreeSet<TaskWrapper> line = new TreeSet<TaskWrapper>({ TaskWrapper t1, TaskWrapper t2 ->

int res = t1.finishTime <=> t2.finishTime

if (!res) { return t1.task.value.name <=> t2.task.value.name }

return res

} as Comparator<TaskWrapper>)

def addTask(final int time, final TaskVertex t) {

def res = new TaskWrapper(startTime : time, task : t)

line.add res

return res

}

def getCompletedTasksAndChangeTime() {

if (!line) { return Collections.emptyList() }

def result = []

TaskWrapper t = line.pollFirst()

result += t

time = t.finishTime

if (!line) { return result }

TaskWrapper nextT = line.first()

while (nextT?.finishTime == t.finishTime) {

result += nextT

line.pollFirst()

nextT = line ? line.first() : null

}

return result

}

String toString() { "Timeline/$time -> $line" }

int getProcessorReleaseTimeDelay(def p, def pSelector) {

return ((line.findAll { pSelector(it.task) == p }).max { a, b -> a.finishTime <=> b.finishTime }).finishTime - time

}

}

class TaskWrapper {

int startTime

TaskVertex task

int getFinishTime() {

float k = State.INSTANCE.sysParams.processorSpeed

return startTime + task.value.volume \* k

}

String toString() { return "$finishTime: $task" }

}

package org.mazur.stplan.gui

import org.jfree.data.gantt.TaskSeries;

import org.mazur.stplan.State

import org.mazur.stplan.model.SystemParameters;

import org.mazur.stplan.sim.SimManager;

import org.mazur.stplan.sim.SimState;

import java.awt.BorderLayout;

import org.jfree.chart.ChartFactory;

import org.jfree.chart.ChartPanel;

import org.jfree.chart.JFreeChart;

import org.jfree.data.category.IntervalCategoryDataset;

import org.jfree.data.gantt.TaskSeries;

import org.jfree.data.gantt.TaskSeriesCollection;

import groovy.swing.SwingBuilder;

/\*\*

\*

\* @version: $Id$

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

class SimWindow {

private IntervalCategoryDataset dataset = new TaskSeriesCollection()

private def seriesList = [:]

private JFreeChart chart

private def tasksQueueMessage, pQueueMessage

private SimManager simManager

private SwingBuilder swing = new SwingBuilder()

private def nextAction = swing.action(

name : "Наступний крок",

closure : {

def data = simManager.next()

if (!data) {

GUIBuilder.locate swing.dialog(title : "Завершено", pack : true, visible : true) {

label("OK: " + simManager.time)

}

return

}

addToSet data

}

)

private def allAction = swing.action(

name : "До кінця",

closure : {

def waitD = swing.dialog(title : "Моделювання", pack : true, visible : true) {

borderLayout()

label(constraints : BorderLayout.NORTH, text : "Зачекайте будь ласка...")

progressBar(indeterminate : true)

}

GUIBuilder.locate waitD

def worker = new Thread(new CRunnable({

def result = simManager.all()

GUIBuilder.invokeSwing {

addToSet result

waitD.visible = false

}

}))

worker.start()

}

)

private void addToSet(def data) {

data?.each {

TaskSeries ts = seriesList[it.type]

if (!ts) { return }

ts.add it.task

}

}

public void init(final def taskM, final def processorM, final SystemParameters sysParams) {

State.INSTANCE.sysParams = sysParams

def tq = State.INSTANCE.queueAlg.formQueue(taskM[0], taskM[1])

tasksQueueMessage = tq[1]

def pq = State.INSTANCE.processorsAlg(processorM[0], processorM[1])

pQueueMessage = pq[1]

simManager = new SimManager(simState : new SimState(

freeProcessors : pq[0],

tasksQueue : tq[0]

))

(pq[0].sort {a,b-> a.value.name <=> b.value.name}).each {

TaskSeries ts = new TaskSeries("$it.value.name")

seriesList[it.value.name] = ts

dataset.add ts

}

chart = ChartFactory.createGanttChart("Діаграма Ганта", "Вузол", "Час", dataset, true, true, false)

}