НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА

з дисципліни «Системне програмне забезпечення»

на тему: «Вирішення задачі статичного планування для топології вектор»

Виконав: Бояршин Ігор Іванович
Факультет: ІОТ
Група: ІО-52
Залікова книжка №5207

Керівник: Симоненко В.П.
Допущено до захисту _____

_____
(підпис керівника)

Київ - 2018 рік

ЗМІСТ

ВСТУП

З розвитком комп'ютерної техніки розвивалися й задачі, що потрібно вирішувати. Вони ставали все складнішими та вимогливішими до апаратури, адже все більше часу потрібно було для їхнього виконання.

Кожну задачу можна розглядати як сукупність ділянок коду (підзадач), кожна з яких має виконуватися послідовно. Ці ділянки є незалежними між собою у тому сенсі, що можуть виконуватися паралельно. Однак зазвичай між такими підзадачами існує залежність по даним: одній підзадачі потрібні результати роботи однієї чи декількох інших, і тому розпочатися вона може лише тоді, коли закінчать своє виконання і перешлють їй дані всі задачі, від яких вона залежить.

З метою прискорення виконання комплексних задач були створені паралельні обчислювальні системи. Вони складаються з множини обчислювальних елементів, що можуть бути з'єднані між собою різним чином, утворюючи безліч можливих топологій, кожна з яких має свої переваги та недоліки.

Ефективність розв'язання комплексної задачі на такій паралельній обчислювальній системі напряму залежить від топології системи та від алгоритму розподілення підзадач між окремими обчислювальними елементами системи (ядрами). Такі алгоритми отримали назву алгоритмів планування (або багатопроцесорний розклад).

Окремим видом планування є статичне планування. Суть статичного планування полягає в тому, що характеристики підзадач та зв'язки між ними відомі заздалегідь, що дає можливість більш ефективно вирішити задачу планування.

У цій роботі практично реалізовано один із алгоритмів статичного планування для системи з топологією "вектор".

РОЗДІЛ 1.
АНАЛІЗ ТОПОЛОГІЇ

У цій роботі розглядається паралельна обчислювальна система з топологією "вектор". Схематично ця топологія зображена на рис. 1.

Рис. 1. Топологія «вектор»

Вона представляє собою лінійну структуру. Кожний процесор *i* має двох сусідів: лівого *(i-1)* та правого *(i+1)*. Виключення складають крайній лівий та крайній правий процесори топології, що мають лише по одному сусіду.

Канал зв'язку в такій топології виглядає наступним чином: кожні два сусіди можуть обмінюватися інформацією між собою в обох напрямках, але лише в одному напрямі в один момент часу. Тобто шина – двонаправлена одноканальна.

До переваг такої топології потрібно віднести відносну простоту апаратної реалізації, адже майже всі елементи є ідентичними, а кожен має лише два зв'язки, що спрощує задачу комутації.

Проте є і недоліки: при виході з ладу будь-якого вузла зникає комунікація між лівою та правою частиною мережі. Ще одним недоліком є час пересилання даних від одного вузла до іншого, що зростає тим більше, чим далі один від одного ці вузли знаходяться. Окрім цього, неефективні дальні пересилки ще й завантажують канал зв'язку.

Через це дуже актуальним є зменшення пересилок загалом і зокрема їх довжини, що досягається розміщенням зв'язаних між собою задач на одному ядрі або якомога ближче один до одного.

РОЗДІЛ 2.
ОПИС ВХІДНИХ ДАНИХ

Вхідними даними для роботи алгоритму є вхідний граф задачі *G*, що складається з множини вершин (підзадач) *V* та множини ребер (зв'язків між підзадачами) *E*. Це направлений, ациклічний граф. Приклад такого графу, згенерованого випадковим чином, наведений на рис. 2.

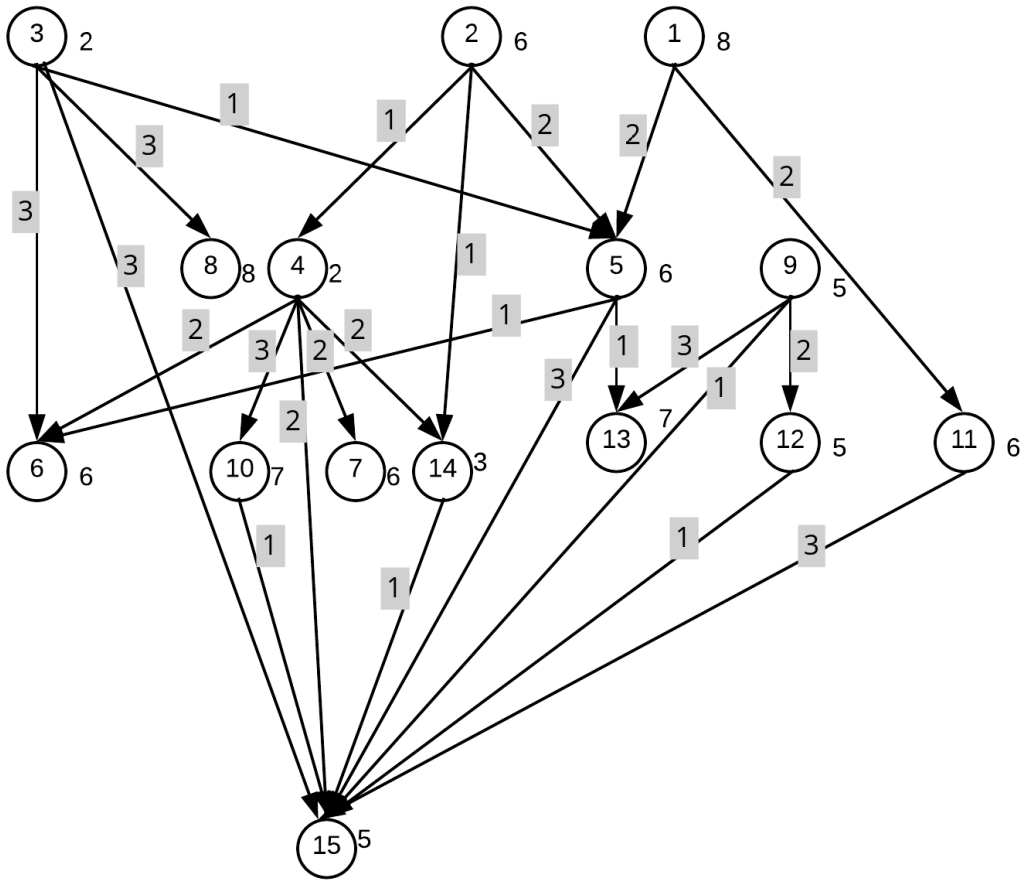

Рис. 2. Згенерований граф

Кожна підзадача представлена вершиною в графі і має два параметри: ідентифікатор підзадачі *(1..n)* та її вага *(w)*. Вага підзадачі відображає кількість тактів, що потрібні для вирішення цієї підзадачі.

Кожен зв'язок між підзадачами представлений ребром у графі і має такі параметри: ідентифікатор підзадачі, що відправляє дані, ідентифікатор задачі, що отримує дані, а також вага зв'язку, що відповідає часу, що потрібен для пересилки цих даних.

РОЗДІЛ 3.
ПРАВИЛА

При вирішенні задачі статичного планування були прийняті такі правила:

- підзадачі на процесорі не перериваються і виконуються одразу повністю. Її вихідні дані для залежних від неї підзадач доступні лише після повного завершення виконання цієї підзадачі;
- кожний окремий обчислювальний елемент (ядро) може одночасно та незалежно вирішувати задачу обчислення підзадачі і виконувати пересилку кожному зі своїх сусідів;
- підзадача може розпочати роботу на процесорі і лише після того як отримає на нього всі дані від підзадач, від яких вона залежить;
- вага зв'язку між підзадачами показує час, необхідний для передачі необхідних даних від відповідних вершин. Передача даних вважається закінченою і дані можуть використовуватися лише після завершення останнього такту передачі. Якщо, наприклад, вага зв'язку від підзадачі 1 до підзадачі 2 складає 3, і ці дві підзадачі були сплановані на ядрах 5 і 7 відповідно, то витрачений на безпосередню пересилку цього зв'язку складатиме 2 * 3 = 6, адже спочатку повідомлення потрібно відправити з процесора 5 на процесор 6, а потім з процесора 6 на процесор 7;
- якщо підзадача *А* та залежна від неї підзадача *Б* сплановані на один і той самий процесор, то дані від підзадачі *А* вважаються доступними для підзадачі *Б*, тобто додаткова пересилка не виконується.

РОЗДІЛ 4.
ОПИС АЛГОРИТМУ

Задана задача є *NP*-повною, що унеможливлює використання повного перебору для її вирішення. Через це велику популярність здобули евристичні алгоритми, що видають допустиме рішення за допустимий час.

Планування виконується в першу чергу для мінімізації загального часу виконання і в другу чергу для мінімізації використаних ядер.

У цій роботі використовувався наступний алгоритм статичного планування:

1) Перед початком роботи обчислити для кожної задачі такий критерій як *importance*, що є мірою "важливості" вершини і показує, наскільки пріоритетною є вершина у порівнянні з іншими. Цей параметр обчислюється як сума власної ваги та сумарного значення *importance* у всіх нащадків.
2) Вибрати задачу з найбільшим значенням *importance* (виходячи з визначення цього параметру, гарантується, що для вибраної задачі всі її предки вже були сплановані).
3) Для всієї множини вже задіяних процесорів, а також додаткового одного пустого зліва та справа, визначити *score* планування вибраної задачі на цей процесор. Параметр *score* є мірою для порівняння різних варіантів планування. В базовій версії він визначається просто як мінімальний час старту задачі на заданому ядрі з урахуванням виконання усіх необхідних пересилок. Чим *score* менший, тим кращим вважається конкретний "сценарій" планування.
4) Вибрати процесор з найменшим *score* та спланувати задачу на цей процесор (з урахуванням виконання всіх необхідних пересилок).
5) Якщо більше немає несплановних задач – завершити виконання, інакше перейти до кроку 2.

Сильними сторонами цього алгоритму є:

- Простота для розуміння та відносна простота реалізації.
- Так як в ході порівняння всіх сценаріїв планування чергової задачі до уваги одразу беруться додаткові "витрати" на виконання пересилки, це дозволяє не тільки

одразу врахувати їх негативний вплив на час старту задачі (коли пересилка виконується з дальнього процесора), але й ввести в алгоритм обчислення *score* степінь негативного впливу додаткової одиниці пересилки, то дозволяє відфільтровувати сценарії, які "перевантажують" канали передачі даних.

- При виборі ядра для планування задачі береться до уваги не тільки поточний час завершення всіх операцій на ядрі, але й всі минулі "зазори", що дозволяє "всунути" нову задачу в них, тим самим ефективніше використати ресурс процесора.

Проте, алгоритм має і деякі недоліки, зокрема те, що вже сплановані задачі і їх пересилки ніколи не змінюються. Тобто на кожному кроці виконується найбільш оптимальна на даний момент дія без урахування можливих покращень по мірі того, як до уваги беруться подальші підзадачі.

РОЗДІЛ 5.
ВИКОНАННЯ СТАТИЧНОГО ПЛАНУВАННЯ

Виконавши статичне планування графу, представленого на рис. 2, згідно описаного алгоритму, отримуємо графік Ганта, що наведений на рис. 3.

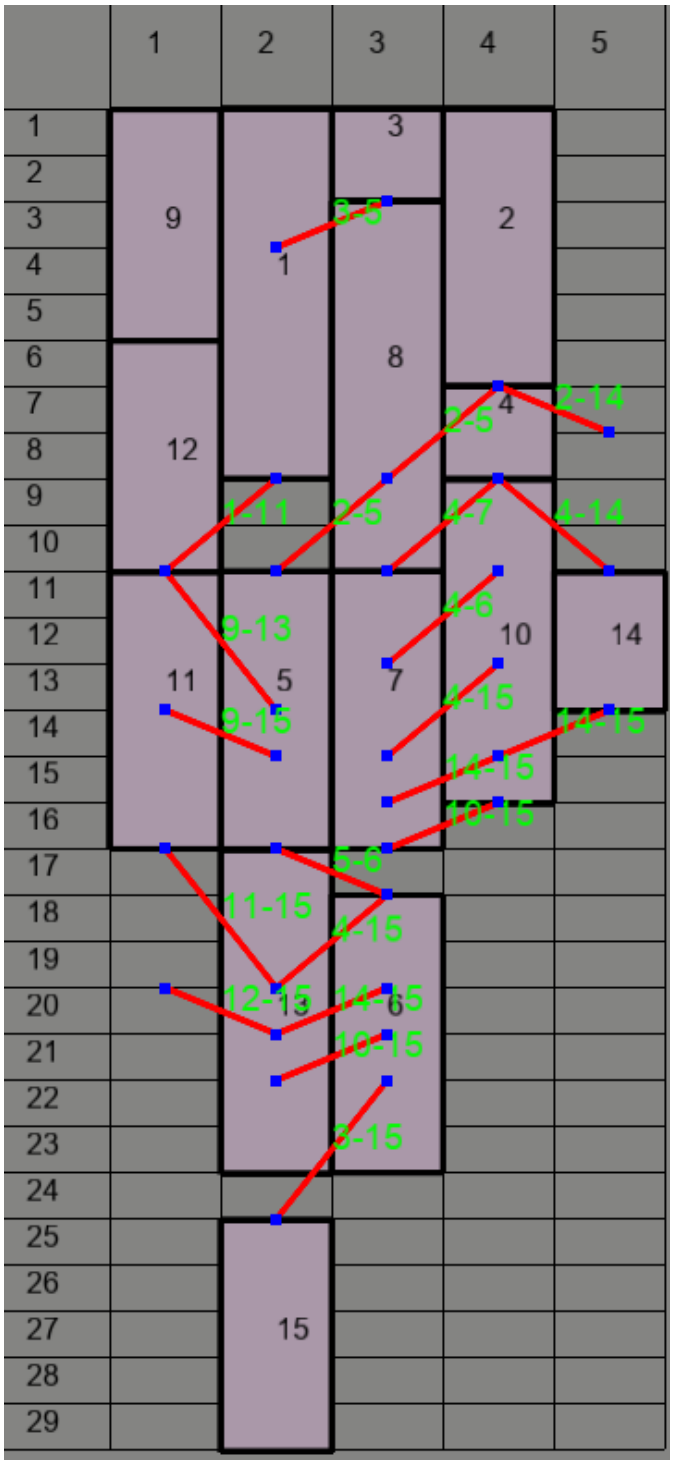

Рис.3. Графік Ганта

Як видно з графіку Ганта, загальний час виконання склав 29 тактів, при цьому було використано 5 ядер у топології "вектор".

Аналізуючи отриманий результат, можна зробити наступні висновки:

- Алгоритм добре показав себе, видавши непогану щільність заповнення ядер системи та допустимий час виконання.
- Значне використання каналів зв'язку особливо помітне на останніх стадіях виконання задачі при пересилках даних для 15-ї задачі, що цілком виправдано, адже вона залежить аж від 8-ми інших підзадач.
- При плануванні 12-ї задачі її було "всунуто" між уже спланованими 9-ю та 11-ю підзадачею на першому ядрі, що висвітлює одну з переваг алгоритму (адже інакше вона б була спланована на новий нульовий процесор).
- Незважаючи на те, що п'яте ядро виділяється цілком під одну підзадачу (що здається марною тратою), це дозволило почати передавати її дані для 15-ї задачі раніше, що дозволило зменшити загальний час виконання. В цьому проявляється правило оптимізації по загальному часу, і лише потім по кількості процесорів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Simonenko, Samofalov K.G., A new approach in solving problems in a distributed computer environment during dynamic scheduling, First international conference on parallel processing and applied mathematics - Poland, 1994.
2. Sakar V., Partition and Scheduling Parallel Programs for Execution on Multi-processor, The MIT Press, 1989.
3. Pham Hong Hanh and Simonenko Valery, Task Assignment for Scheduling Jobs and Resources in Parallel Distributed Systems, Journal Informatics and Kibernetic, Vol 12, N3, 1996, pp. 1-13.
4. Yang T. and Gerasoulis A., List Scheduling with and without Communication delays, Report DCS, Rutgers Uni., 1991.

ДОДАТКИ

Лістинг програми

```
use std::collections::HashMap;
use std::fs::File;
use std::io::{BufWriter, Write};

extern crate rand;
use rand::prelude::*;

#[derive(Debug)]
struct Vertex {
    id: VertId,
    w: Weight,
}

struct Link {
    src: VertId,
    dst: VertId,
    w: Weight,
}

type VertId = u32;
type ProcId = i32;
type Tick = u32;
type TaskId = u32;
type Weight = u32;
type Importance = u32;

#[derive(Debug)]
struct Task {
    id: TaskId,
    w: Weight,
    imp: Importance,
    children: Vec<(TaskId, Weight)>,
    parents: Vec<(TaskId, Weight)>,
}

fn tasks_from(vertices: Vec<Vertex>, links: Vec<Link>) -> Vec<Task> {
    let mut tasks = Vec::new();
    let length = vertices.len() as u32;

    // Create tasks
    for Vertex { id, w } in vertices.into_iter() {
        if id >= length {
            panic!("Using VertId beyond upper bound.")
        }
        if id != (tasks.len() as u32) {
            panic!("Pushing for wrong index");
        }

        tasks.push(Task {
            id: id,
            w: w,
            imp: 0,
            children: Vec::new(),
            parents: Vec::new(),
        });
    }
```

```
    // Add links
    for Link { src, dst, w } in links.into_iter() {
        if src >= (tasks.len() as u32) || dst >= (tasks.len() as u32) {
            panic!("Link from or to out-of-bounds Vertex");
        }
        tasks[src as usize].children.push((dst, w));
        tasks[dst as usize].parents.push((src, w));
    }

    // Set importance
    // Assumes that at each iteration there will be at least 1 Task whose all
    // children have their Importance set. Utilizes exactly 1 such Task at each iteration.
    for _ in 0..tasks.len() {
        let (index, imp) = tasks
            .iter()
            .filter(|Task { imp, .. }| *imp == 0)
            .find_map(
                |Task { id, w, children, .. }| {
                    children.iter()
                        .try_fold(w.clone(), |acc, &(child_id, _)| {
                            let imp = tasks[child_id as usize].imp;
                            if imp != 0 {
                                Some(acc + imp)
                            } else {
                                None
                            }
                        })
                        .and_then(|imp| Some((id.clone(), imp)))
                },
            )
            .unwrap(); // can safely unwrap because of the assumption described above
        tasks[index as usize].imp = imp;
        // println!("{} for {}", imp, index);
    }

    tasks
}


fn print_tasks(tasks: &Vec<Task>) {
    for Task {id, w, children, parents, imp} in tasks.iter() {
        println!("Task #{} [{}] <{}>:", id+1, w, imp);
        print!("\t->");
        for (dst, w) in children.iter() {
            print!(" #{}[{}]", dst+1, w);
        }
        println!();
        print!("\t<-");
        for (src, w) in parents.iter() {
            print!(" #{}[{}]", src+1, w);
        }
        println!();
    }
}

fn main() {
    let (vertices, links) = populate_random();
    // let (vertices, links) = (populate_vertices(), populate_links());
    let tasks = tasks_from(vertices, links);
    print_tasks(&tasks);
    let mut s = System::new(tasks);
    s.plan();
```

```
    s.export("planning.txt".to_string());
}


struct OutTask {
    start: Tick,
    proc: ProcId,
    weight: Weight,
    id: TaskId,
}

#[derive(Debug)]
struct OutLink {
    src_core: ProcId,
    dst_core: ProcId,
    start: Tick,
    weight: Weight,
    src_task: TaskId,
    dst_task: TaskId,
}

impl OutLink {
    fn serialize(&self, leftmost_proc: &ProcId) -> String {
        let mut s = "OutLink\n".to_string();

        s += "src_core:";
        s += &(self.src_core - leftmost_proc).to_string();
        s += "\n";

        s += "dst_core:";
        s += &(self.dst_core - leftmost_proc).to_string();
        s += "\n";

        s += "weight:";
        s += &self.weight.to_string();
        s += "\n";

        s += "start:";
        s += &(self.start + 1).to_string();
        s += "\n";

        s += "src_task:";
        s += &self.src_task.to_string();
        s += "\n";

        s += "dst_task:";
        s += &self.dst_task.to_string();
        s += "\n";

        s
    }
}

impl OutTask {
    fn serialize(&self, leftmost_proc: &ProcId) -> String {
        let mut s = "OutTask\n".to_string();

        s += "start:";
        s += &(self.start + 1).to_string();
        s += "\n";

        s += "proc:";
```

```
        s += &(self.proc - leftmost_proc).to_string();
        s += "\n";

        s += "weight:";
        s += &self.weight.to_string();
        s += "\n";

        s += "id:";
        s += &self.id.to_string();
        s += "\n";

        s
    }
}

#[derive(PartialEq, Eq, Hash, Clone, Copy, Debug)]
struct ProcPair {
    left: ProcId,
    right: ProcId,
}

impl ProcPair {
    fn new(left: ProcId, right: ProcId) -> ProcPair {
        if left < right { ProcPair {left: left,  right: right} }
        else            { ProcPair {left: right, right: left}  }
    }
}

fn clone_buses(buses: &Buses) -> Buses {
    let mut new_buses = HashMap::new();
    for (proc_pair, tick) in buses.iter() {
        new_buses.insert(proc_pair.clone(), tick.clone());
    }
    new_buses
}

// proc -> vec<is_busy>
type Processors = HashMap<ProcId, Vec<bool>>;
// leftproc
type Buses = HashMap<ProcPair, Tick>;
// where the tasks was planned and where it finished
type PlannedTasks = HashMap<TaskId, (ProcId, Tick)>;


struct System {
    unplanned_tasks: Vec<Task>,
    out_tasks: Vec<OutTask>,
    out_links: Vec<OutLink>,
    leftmost_proc: ProcId,
    rightmost_proc: ProcId,
    processors: Processors,
    buses: Buses,
    planned_tasks: PlannedTasks,
}

#[derive(Debug)]
struct Scenario {
    proc: ProcId,
    buses: Buses,
    start: Tick,
    new_links: Vec<OutLink>,
    score: u32,
```

```
}


fn gen_paths(src: ProcId, dst: ProcId) -> Vec<(ProcId, ProcId)> {
    if src < dst { (src..dst).map(|x| (x, x+1))      .collect() }
    else         { (dst..src).map(|x| (x+1, x)).rev().collect() }
}


impl System {
    fn export(&self, path: String) {
        let f = File::create(path).expect("Unable to create file");
        let mut f = BufWriter::new(f);

        self.out_tasks.iter().for_each(|task|
            f.write_all(task.serialize(&self.leftmost_proc).as_bytes())
                .expect("Unable to write data"));
        self.out_links.iter().for_each(|link|
            f.write_all(link.serialize(&self.leftmost_proc).as_bytes())
                .expect("Unable to write data"));
    }

    fn plan(&mut self) {
        while let Some(task) = self.pop_next() {
            // println!("Working with {:?}", task);
            let Scenario {proc, buses, start, mut new_links, ..} =
                (self.leftmost_proc..=self.rightmost_proc)
                .map(|proc| self.play_scenario(proc, &task))
                // .inspect(|scenario| println!("      Considering {:#?}", scenario))
                .min_by_key(|scenario| scenario.score)
                .unwrap();
            self.buses = buses;
            self.out_links.append(&mut new_links);
            self.out_tasks.push(OutTask {
                start,
                proc,
                weight: task.w,
                id: task.id,
            });
            self.place_at_proc(&proc, &start, &task.w);
            self.planned_tasks.insert(task.id, (proc, start + task.w));
            self.enlarge_if_needed(proc);
        }
    }

    fn place_at_proc(&mut self, proc: &ProcId, start: &Tick, w: &Weight) {
        let start = start.clone();
        let length = self.processors[proc].len();
        for _i in length..(start as usize) { // if start > length
            self.processors.get_mut(proc).unwrap().push(false);
        }
        let length = self.processors[proc].len();
        for i in (start as usize)..length {
            self.processors.get_mut(proc).unwrap()[i] = true;
        }
        for _i in length..((start+w) as usize) {
            self.processors.get_mut(proc).unwrap().push(true);
        }
    }

    fn enlarge_if_needed(&mut self, proc: ProcId) {
        if proc == self.leftmost_proc {
```

```
            self.leftmost_proc -= 1;
            self.processors.insert(self.leftmost_proc, Vec::new());
            self.buses.insert(ProcPair::new(self.leftmost_proc, self.leftmost_proc + 1), 0);
        }
        if proc == self.rightmost_proc {
            self.rightmost_proc += 1;
            self.processors.insert(self.rightmost_proc, Vec::new());
            self.buses.insert(ProcPair::new(self.rightmost_proc,  self.rightmost_proc  -  1),
0);
        }
    }

    fn finish_of(&self, task: &TaskId) -> Tick {
        let (_, tick) = self.planned_tasks.get(task).unwrap();
        tick.clone()
    }

    fn proc_of(&self, task: &TaskId) -> ProcId {
        let (proc, _) = self.planned_tasks.get(task).unwrap();
        proc.clone()
    }

    fn bus_finish_of(&self, src: &ProcId, dst: &ProcId, buses: &Buses) -> Tick {
        buses.get(&ProcPair::new(src.clone(), dst.clone())).unwrap().clone()
    }

    fn play_scenario(&self, proc: ProcId, task: &Task) -> Scenario {
        let mut buses = clone_buses(&self.buses);
        let (transmission_finish, new_links): (Tick, Vec<OutLink>) = task.parents.iter()
            .filter(|(parent, _)| self.proc_of(parent) != proc)
            .map(|(parent, weight)| {
                let  (transmission_finish,  new_links)  =  gen_paths(self.proc_of(&parent),
proc).into_iter()
                    .fold((self.finish_of(&parent), Vec::new()),
                            |(src_finish, mut links), (src_core, dst_core)| {
                        let start = std::cmp::max(
                            src_finish,
                            // work upon the being updated buses
                            self.bus_finish_of(&src_core, &dst_core, &buses)
                        );
                        links.push(OutLink {
                            src_core,
                            dst_core,
                            start: start.clone(),
                            weight: weight.clone(),
                            src_task: parent.clone(),
                            dst_task: task.id.clone(),
                        });
                        (start + weight.clone(), links)
                    });
                // Imprint these links, so that all consequent paths of links
                // work upon the updated buses.
                for OutLink {src_core, dst_core, start, weight, ..} in new_links.iter() {
                    buses.insert(
                        ProcPair::new(src_core.clone(), dst_core.clone()),
                        (start + weight).clone());
                }
                (transmission_finish, new_links)
            })
            .fold((0, Vec::new()), |(finish, mut links), (cur_finish, mut cur_links)| {
                links.append(&mut cur_links);
                (std::cmp::max(finish, cur_finish), links)
```

```
            });

        let start = self.find_consecutive_block(&proc, &transmission_finish, task.w.clone());
        // print!("Found block with w={} starting at {}", task.w, start);
        // let start = std::cmp::max(free_on_proc, &transmission_finish).clone();
        Scenario {
            score: start.clone(),
            start,
            new_links,
            buses,
            proc,
        }
    }

    fn find_consecutive_block(&self, proc_id: &ProcId, starting_tick: &Tick, w: Weight) ->
Tick {
        let proc = &self.processors[proc_id];
        let mut cur: Tick = starting_tick.clone();
        loop {
            if cur as usize >= proc.len() { return cur; }
            if !proc[cur as usize] { // is free
                let mut succ = true;
                for i in cur..cur+w {
                    if i as usize >= proc.len() { break; }
                    if proc[i as usize] { // is busy
                        succ = false;
                        break;
                    }
                }
                if succ { return cur; }
            }
            cur += 1;
        };
    }

    fn new(tasks: Vec<Task>) -> System {
        let mut processors = HashMap::new();
        processors.insert(0, Vec::new());
        System {
            unplanned_tasks: tasks,
            out_tasks: vec![],
            out_links: vec![],
            leftmost_proc: 0,
            rightmost_proc: 0,
            processors: processors,
            buses: HashMap::new(),
            planned_tasks: HashMap::new(),
        }
    }

    fn pop_next(&mut self) -> Option<Task> {
        if self.unplanned_tasks.is_empty() { None }
        else {
            let (index, _) = self.unplanned_tasks.iter().enumerate()
                .max_by_key(|(_, Task {imp, ..})| imp)
                .unwrap();
            Some(self.unplanned_tasks.remove(index))
        }
        // if self.unplanned_tasks.len() > 0
        //      { Some(self.unplanned_tasks.remove(0)) }
        // else { None }
    }
```

```
}


fn populate_random() -> (Vec<Vertex>, Vec<Link>) {
    let vertex_count = 15;
    let min_per_layer = 2;
    let max_per_layer = vertex_count / 2;
    let min_vertex_weight = 2;
    let max_vertex_weight = 8;
    let seed = [6,4,3,8, 7,9,8,10, 14,18,12,12, 14,15,16,17];
    let mut rng = SmallRng::from_seed(seed);

    // Vertices
    let mut done_vertices_count = 0;
    let mut id: VertId = 0;
    let mut layers = Vec::new();
    while done_vertices_count < vertex_count {
        let mut layer = Vec::new();
        let count: u32 = {
            let r = rng.gen_range(min_per_layer, max_per_layer + 1);
            let left = vertex_count - done_vertices_count;
            if left < r {left} else {r}
        };
        for _ in 0..count {
            let weight: Weight = rng.gen_range(min_vertex_weight, max_vertex_weight + 1);
            layer.push(Vertex {id: id, w: weight});
            id += 1;
        }
        done_vertices_count += count;

        layers.push(layer);
    }
    // println!("{:#?}", layers);

    // Links
    let links_count = (vertex_count * (vertex_count - 1) / 2) / 10;
    // println!("Will create {} links", links_count);
    let min_link_weight = 1;
    let max_link_weight = 3;
    let mut links = Vec::new();
    let belongs_to_chunk = |id: u32, chunks: &Vec<Vec<VertId>>| -> Option<usize> {
        chunks.iter()
            .enumerate()
            .find_map(|(index, chunk)|
                if chunk.iter()
                    .find(|&& x| x == id)
                    .is_some() {Some(index)}
                else {None}
            )
    };
    let mut done_links_count = 0;
    let mut chunks = Vec::new();
    let mut converged = false;
    while !converged || done_links_count < links_count {
        // if !converged { println!("Have chunks: {:#?}", chunks); }
        let layer_src_index = rng.gen_range(0, layers.len() - 1);
        let layer_dst_index = rng.gen_range(layer_src_index + 1, layers.len());
        let layer_src = layers.get(layer_src_index).unwrap();
        let layer_dst = layers.get(layer_dst_index).unwrap();
        let src_index = layer_src.get(rng.gen_range(0, layer_src.len())).unwrap().id.clone();
        let dst_index = layer_dst.get(rng.gen_range(0, layer_dst.len())).unwrap().id.clone();
        // Retry if such link already exists
```

```
        if links.iter()
            .find(|&& Link {src, dst, ..}| (src == src_index) && (dst == dst_index))
            .is_some() { continue; }
        let weight: u32 = rng.gen_range(min_link_weight, max_link_weight + 1);
        links.push(Link { src: src_index, dst: dst_index, w: weight });
        done_links_count += 1;

        // Have nothing to trach if have already converged
        if converged { continue; }

        let chunk_src = belongs_to_chunk(src_index, &chunks);
        let chunk_dst = belongs_to_chunk(dst_index, &chunks);
        // println!("Have: {} and {}", src_index, dst_index);
        if chunk_src.is_none() && chunk_dst.is_none() {
            // Create new chunk
            // println!("Creating new chunk");
            chunks.push(vec![src_index, dst_index]);
        } else if chunk_src.is_none() && chunk_dst.is_some() {
            // Add src to dst_chunk
            // println!("Adding src to dst");
            chunks.get_mut(chunk_dst.unwrap()).unwrap().push(src_index);
        } else if chunk_src.is_some() && chunk_dst.is_none() {
            // Add dst to src_chunk
            // println!("Adding dst to src");
            chunks.get_mut(chunk_src.unwrap()).unwrap().push(dst_index);
        } else { // both Some
            let src = chunk_src.unwrap();
            let dst = chunk_dst.unwrap();
            // println!("{} , {}", src, dst);
            if src != dst {
                // println!("Not same, merging");
                // Merge chunks
                let mut other = chunks.remove(dst);
                chunks.get_mut(if dst < src {src - 1} else {src}).unwrap().append(&mut other);
                // converted=true was here
            } else {
                // println!("Same, nothing");
            }
        }
        // Check if all are connected
        if chunks.len() == 1 && chunks[0].len() == vertex_count as usize {
            converged = true;
        }
    }

    // println!("Done {} links total", done_links_count);

    (layers.into_iter().flatten().collect(), links)
}

fn populate_vertices() -> Vec<Vertex> {
    vec![
        Vertex {
            id: 0,
            w: 3,
        },
        Vertex {
            id: 1,
            w: 4,
        },
        Vertex {
```

```
            id: 2,
            w: 5,
        },
        Vertex {
            id: 3,
            w: 3,
        },
        Vertex {
            id: 4,
            w: 3,
        },
        Vertex {
            id: 5,
            w: 2,
        },
        Vertex {
            id: 6,
            w: 4,
        },
    ]
}

fn populate_links() -> Vec<Link> {
    vec![
        Link {
            src: 0,
            dst: 3,
            w: 1,
        },
        Link {
            src: 0,
            dst: 2,
            w: 2,
        },
        Link {
            src: 1,
            dst: 2,
            w: 1,
        },
        Link {
            src: 1,
            dst: 6,
            w: 2,
        },
        Link {
            src: 3,
            dst: 4,
            w: 1,
        },
        Link {
            src: 3,
            dst: 5,
            w: 2,
        },
        Link {
            src: 2,
            dst: 5,
            w: 1,
        },
    ]
}
```