SoftEng306

Project 1: Using AI and parallel processing power to solve difficult scheduling problem

Dr Oliver Sinnen

Version 1.2

1 Introduction

This project is about using artificial intelligence and parallel processing power to solve a difficult scheduling problem. It is about Speed! In this project of the SoftEng306 design course your client is the leader of a Parallel Computing Centre, who needs you to solve a difficult scheduling problem for his parallel computer systems. This client happens to be Oliver Sinnen. The project is well defined and scoped, but the particular challenge of this project are the contradicting objectives: developing a solution with a Software Engineering approach and design while having a very fast execution speed. At the end of the project your solutions will compete against each other in a speed comparison. Your team might choose to contribute the solution to the PARC lab for their work.

2 The problem

A few years ago, processor technology started to reach physical limitations. Since then, more and more computing systems have become parallel, i.e. use more than one processor, because this is the only way to further increase performance. As a consequence, parallel computing became omnipresent, where even smartwatches possess more than one processor (core) and supercomputers use millions.

To efficiently use more than one processor, a software program must be written correspondingly. The program must be divided or split into multiple tasks and they must be executed by the different processors. A crucial problem for the efficiency of executing such a parallel program is how tasks are mapped to the available processors and in which order they are executed.

This so called scheduling problem is very difficult, not only in practise, but even in theory. In order to study this problem and to find good solutions, it has been formalized. A program is described by a task graph, where the nodes of the graph represent tasks and the edges represent data transfer or dependences between the tasks. The data transfers (or communications) are directed and there must not be any cycle (otherwise there would be a dependence cycle), which makes the graph a directed acyclic graph (DAG). Our scheduling problem now becomes the assignment of the nodes (tasks) of the graph to a given set of p processors (corresponding to the processor number of the system where we want to execute our program). We also need to define the order in which the tasks are executed on the processors. The objective is to do the mapping and ordering of the tasks in such a way that the total execution time of the program is minimized.

In its general form, this scheduling problem belongs to the class of so called NP-hard problems. All NP-hard problems share the property that no efficient algorithm has been found to optimally solve them. Efficient means here that all the known algorithms that can *optimally* solve these problems have a runtime complexity that grows exponentially (!) with the size of the problem. In our case that means that the runtime grows with the number n of tasks, i.e. it is $O(c^n)$, where c is a constant. Many practically relevant problems belong to the class of NP-hard problems, e.g. the Traveling Salesperson Problem, aircraft crew scheduling, optimal routing in electronic circuits, university timetabling (!), project allocation to students (part 4 project selection (!)), map colouring, bin packing problem, Boolean satisfiability problem (SAT), berth allocation problem, dependence analysis in compilers, integer linear programming, ...

So normally heuristic algorithms with short runtimes are used for these problems, which deliver often good, but not optimal solutions. This is mostly good enough in practise, but sometimes it is desirable to have an optimal solution nevertheless. For example for a critical system or to verify the quality of the solutions found by heuristics. The only option is then to "brute force" the problem, by simply trying out all possible solutions and to see which one is best. Of course, this only works for small to mid-sized problems, after all the runtime will grow exponentially (but we might be able to do something about the constant c).

Our above described scheduling problem is a combinatorial optimisation problem. This means there is a finite number of possible solutions, even though this number can be extremely large. Every task can be allocated to every processor, i.e. there are p^n possible different allocations, and we can execute the tasks in any possible order, i.e. there are $O(n!)$ possible permutations (there are not exactly $n!$ permutations, as edges in the graph prohibit some permutations). This results in a total complexity of $O(p^n n!)$ and will be an extremely large number of possibilities even for very tiny values of n. So we need to be smart about the way we try out solutions and to avoid all solutions that cannot be optimal.

These kind of combinatorial optimization problems are often tackled with algorithms from Artificial Intelligence, in particular branch-and-bound algorithms. In that approach, all possible solutions are explored in the structure of a tree. We build new solutions by modifying one aspect of the current solution and this naturally leads to a tree structure spawning all solutions. To find the optimal solution we need to search the tree. Two prominent branch-and-bound algorithms to do this are called depth-first search branch-and-bound and A*.

3 Project

Your assignment is to develop a branch-and-bound type algorithm that solves the above described scheduling problem optimally for small input graphs. To achieve the highest possible performance, you also need to parallelise this branch-and-bound algorithm so that it itself uses multiple processors to speed up the search.

3.1 Scheduling problem

The input is the directed acyclic task graph $G = (V, E, w, c)$, where V is the set of tasks and E is the set of edges between the tasks. A weight $w(n)$, $n \in V$, associated with task n represents its execution time and weight $c(e_{ij})$, $e_{ij} \in E$, represent the time the data transfer (communication) between task n_i and task n_j takes if the two tasks are executed on different processors. If they are executed on the same processor, it is assumed that the data transfer time is negligible and set to 0. Figure 1 shows a simple example task graph with four tasks.

In a schedule of the graph G, we denote the start time of a task with $t_s(n)$. At any point in time only one task is active on every processor and it cannot be interrupted. It follows that if two tasks n_i and n_j are executed on the same processor, i.e. $proc(n_i) = proc(n_j)$ then either $t_s(n_j) \geq t_s(n_i) + w(n_i)$ or $t_s(n_i) \geq t_s(n_j) + w(n_j)$, i.e. one task must completely finish before the next task starts. Edges impose another restriction on the start time of tasks. For each edge $e_{ij} \in E$ it must be true that $t_s(n_j) \geq t_s(n_i) + w(n_i) + c(e_{ij})$, if the tasks are executed on different processors, otherwise just $t_s(n_j) \geq t_s(n_i) + w(n_i)$. Figure 2 depicts a schedule of the example task graph of Figure 1 on two

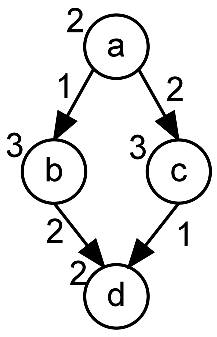

Figure 1: Task graph example

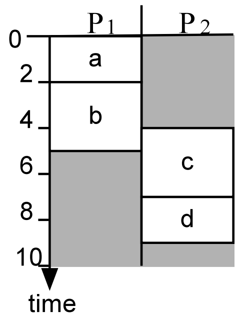

Figure 2: Schedule of example DAG (Fig. 1) on 2 processors

processors. Observe that task c on processor P_2 can only start at time unit 4 due to the incoming edge from task a with a communication delay of $w(e_{ac}) = 2$.

The objective of the scheduling problem is to minimize the schedule length (also called makespan), which is the finish time of the last task: $sl = \max_{n \in V}\{t_s(n) + w(n)\}$ (assuming the first tasks starts at time 0).

3.2 Branch-and-bound

Exhaustive search algorithms, like branch-and-bound, consider all possible solutions to an optimization problem. This is done in a tree structure, where every node (also called state) of the tree represents a (partial) solution to the problem. For example, in a problem to find the shortest route on a map, a tree node represents a part of the route, e.g. three stages of a route. A leaf node of the tree is the complete route to the destination. A partial solution (node) is expanded by adding one more stage, thereby creating the children of the node. There is one child for each possible choice. In the routing example this means to add the next town to the current route, where every town choice is a different child node. To accelerate the search, we estimate the cost of a node. In the route example, this is the distance already traveled plus the direct line to the destination. If this is larger than an already known route, we can stop to further expand the node, i.e. we bound the search. Like this we can exclude entire subtrees from the search.

In our scheduling problem, even though we assign start times to tasks, it is sufficient to find the allocation of tasks to processors and the ordering of the tasks on the processors. We then simply calculate the start time as the earliest possible time given the availability of the processor and the arrival time of incoming communications. Hence, we can think of a node in the search tree as a partial schedule, where some tasks have already been assigned to processors and/or ordered for execution. A partial schedule is expanded by taking one more scheduling decision, in terms of allocation or ordering

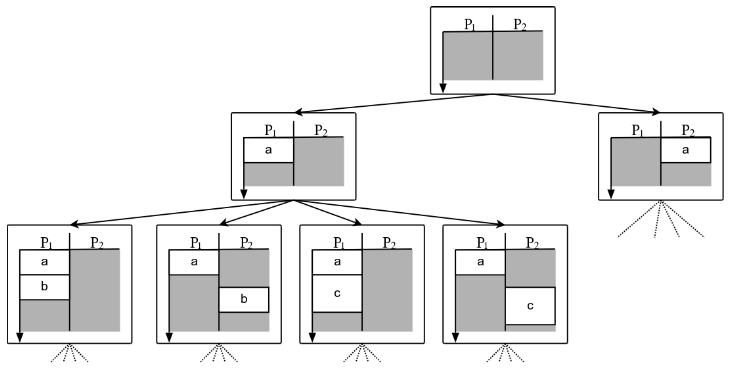

Figure 3: Example solution tree for scheduling graph on 2 processors

or both. Figure 3 shows a possible solution tree for the scheduling of a task graph on two processors, where a child is created by scheduling one more task on a chosen processor.

There are several alternatives to search through such a solution tree. A very common method is to use depth-first search branch-and-bound. Here we start at the root of the tree and go straight down to create a complete solution. Then the algorithms backtracks (going one level up in the tree), modifying one decision and creating a new complete solution. This method uses limited memory and creates a good valid solution quickly, but the entire tree needs to be searched before one can be certain that the optimal solution has been found. Of course, bounding is used to safely exclude as many parts of the search tree as possible. An alternative, which usually is faster, is a best-first search called A*. Here, we also start at the root node, but we use a priority queue to pick the next node to expand. The order in the queue is determined how promising the nodes are in order to obtain the optimal solution. In our case, the highest priority is given to the node with the lowest estimated schedule length (based on the partial schedule represented by the node and an under-estimate of the execution of the remaining tasks).

Both approaches, DFS branch-and-bound or A* can be used to search for the best schedule. Developing good bounding and estimation function will be essential to achieve good performance.

3.3 Parallelization of search

Even when using smart search techniques as described in the previous section, the search space even for small problem instances is very large and searching it can take very long. It seems very meaningful to use more than one processor in parallel to search for the optimal solution.

Hence, after developing a functional sequential implementation of the search, a parallel version needs to be developed. This is to be done as a solution for a single system with multiple processors (cores) and shared memory (as opposed to a distributed solution involving multiple systems connected in a network). All current multicore systems belong to this category, including the PCs in the labs. Essentially this implies the development of a multithreaded implementation of the search that accelerates it in comparison to the sequential version.

3.4 Visualization of search

Using an exhaustive search method can take long, even for small graphs and when done in parallel. Hence, in the spirit of good software engineering, the user needs to be provided with a live visual feedback about the search. There are different things that can be visualised, e.g. a Gantt chart of the current partial schedule that is examined, statistical values about the search or the areas of the complete search tree that have been visited. The visualisation needs to live update with the search in progress and can have interactive elements, such as zoom or focus on certain aspects. The presented information should be meaningful and a reflection of the search, not just showing a progress bar or similar. The exact visualisation is open-ended and is up to the project team to design and implement.

4 Implementation technologies

Programming language and libraries The project is to be implemented in **Java** (compatible with Java 1.8). External libraries (not part of Java SDK) can be used (given proper license), but they need to be clearly separated from own work. For the parallelisation, Parallel Task or Pyjama (http://parallel.auckland.ac.nz/ParallelIT/) can be used.

OS The targeted OS is Linux (solution must run under Remote Linux), but given that it will be a Java project, the project should also work under Windows.

Input The input to the program will be a graph and the number of processors on which to schedule the graph as command line parameters. The graph is given in dot format (file name ending with .dot). Here is an example of the simple graph from Figure 1 in dot format.

```
digraph "example" {
        a           [Weight=2];
        b           [Weight=3];
        a −> b      [Weight=1];
        c           [Weight=3];
        a −> c      [Weight=2];
        d           [Weight=2];
        b −> d      [Weight=2];
        c −> d      [Weight=1];
}
```

Output The output is a file in dot format (with .dot ending), which is essentially a copy of the input with two added attributes to each task: start time and allocated processor. For example the schedule in Figure 2 is given in dot format as:

```
digraph "outputExample" {
        a           [Weight=2,Start=0,Processor=1];
        b           [Weight=3,Start=2,Processor=1];
        a −> b      [Weight=1];
        c           [Weight=3,Start=4,Processor=2];
        a −> c      [Weight=2];
        d           [Weight=2,Start=7,Processor=2];
        b −> d      [Weight=2];
        c −> d      [Weight=1];
}
```

Interface Any milestone of the project needs to be packed in a jar package so that it can be invoked from the command line with the following parameters.

```
java −jar scheduler.jar INPUT.dot P [OPTION]
INPUT.dot   a task graph with integer weights in dot format
P           number of processors to schedule the INPUT graph on

Optional:
−p N        use N threads for execution in parallel (default is sequential)
−v            visualise the search
−o OUPUT    output file is named OUTPUT (default is INPUT−output.dot)
```

Additional arguments can be used to influence the visualisation, to fine-tune or to debug the search, but the best parameters should be chosen as default.

5 Organization and assessment

5.1 Teams

This project is to be undertaken in 6 weeks by teams of 5 students each (a few groups will have 6 students). Teams are to be self-formed and need to select a **leader**. The role of the leader is to keep everybody focused, resolve conflicts, track progress etc. It is not the role of a boss and should not be a micro-manger.

The leader then enrolls the team on canvas under "Project 1 team" (the first member to join a group is automatically the leader). The first groups in the list on canvas allow for 6 members, all other groups only allow for 5 members, so depending on your team size, select a corresponding group number (first come first served, i.e. if there is no 6-member team left, you need to form a team with 5 members).

The work must be divided up among team members as the project progresses, giving each member a role in programming, and also at least one other role e.g. to work on software installation, planning, design, testing, wiki and documentation. However do not divide into independent subgroups because that will lose marks for your team. Share the tasks around so everyone contributes across the whole project and is aware of the overall project development. We want to see clear evidence that the whole team is working together.

Also select a **name** (and optionally logo) for your team and put this information on the team's homepage on canvas. Add a team photo and identify the members on it.

5.2 Development tools

For version control this project will be using **github**. Create a ***private*** (important) project on github and give full access to all team members, each having their own account. Each member needs to push/commit their work under their own account. If you work in pairs, take turns committing (and indicate pair coding in the commits messages). Add a link to the repository on your team's homepage on canvas and indicate the user names of each team member.

Github is to be used for version control of your code. Use the wiki for documentation and the issue tracker. Discussion between team members can be done in the discussion area on canvas.

5.3 Milestones

There are three milestones for the project implementation.

Plan Develop a plan for your project, using a Work Breakdown Structure (WBS), derive a Network Diagram from the WBS to show dependences between the tasks and put this into a Gantt Chart for detailed planning.

Basic milestone For the most basic milestone your implementation needs to be able to:

- Read the input file and number of processors and create an output file
- The output file needs to contain a valid schedule (which does not need to be optimal at this point)

Final milestone The final milestone is the complete implementation. It needs to be able to (in order of importance),

- Create an optimal schedule for small input graphs in reasonable time (say under less than 30 minutes)

- Have a parallel version of the search that demonstrates speedup in comparison to the sequential version
- Have a meaningful and interesting live visualisation of the search

5.4 Assessment

Total for this project: 50%.

- Project plan 5%, due end of week 3 (Friday, 5 August)
- Basic milestone, 15%, demo/interview/implementation, end of week 4 (Friday, 12 August)
- Final milestone, end of week 6 (Friday, 26 August)
  - Demo/interview/implementation, 20%
  - Written report, 10%
  - Confidential peer evaluation form (each student)

The milestones will be graded according to various criteria. These are functionality (finding optimal schedule), speed (time to find the optimal schedule), high quality of coding standards, comments, documentation and testing.

This project has the particular challenge of the contradicting objectives of high software engineering standards of design and code, and the need to have very fast execution speed.

5.5 Lectures, labs, interviews

Our allocated slots are Wednesday 8am-10am and Friday 9am-11am for lectures and Thursday 12pm-2pm, 2pm-4pm and Friday 12pm-2pm for lab.

These slots will be used as follows (subject to change!):

week	Mon	Tue	Wed	Thu	Fri
1			lecture		lecture
2			lecture	lab	team presentations + lab
3				lab	team presentations + lab
4					interviews
5				lab	team presentations + lab
6					interviews

Table 1: Schedule of first 6 weeks

During the first two weeks, you will be introduced to the course, the project will be presented and project management methods will be briefly reviewed.

Starting from week 2, TAs will be available in UG4 (301.1062) during the lab slots to answer questions and to help with problems. They will leave after 30 minutes if no questions are asked.

Interviews will be conducted on the Friday in week 4 and week 6 in UG4. This will be mostly during the lecture and lab slots, but additional times are necessary.

In-class team presentations During the Friday lecture of week 2, 3, and 5, teams will be briefly presenting their progress on the project (5 minutes). All team members need to be present and participate.

5.6 Speed competition

At the end of the project in week six we will hold a speed competition, measuring the performance of the sequential and parallel versions of the submitted implementation and will determine winners in each category.