Developing an Application to Introduce Parallel Task Scheduling

Juntong Liu

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2019

Abstract

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

Computing resource is in high demand both in industry and for academic research. In industry, companies are collecting terabytes or even petabytes of user data to provide customized services. Also, machine learning algorithms are widely used to provide suggestions and to extract information from media files. Processing these files and executing these algorithms all requires numerous amount of computing resource. In academic research, many researchers in subjects like hydromechanics and electrics rely on simulation to predict the performance of models. In this case, more computing resource is consumed for better resolution.

However, researchers said Moore's Law will not be effective in the near future (ref), meaning speed of improvement in single core performance may be far behind the increasing demand. Therefore, the typical way to utilize more computing resource is to use multiple processors to work on the same task in parallel. Traditional code for single core execution cannot be used directly for parallel execution. They have to be modified. One common solution to parallelize a big task is to divide it into small tasks that can be executed separately on multiple processors. However, in most of cases, the small tasks cannot be independent because they might require data produced by other tasks. The dependency can be usually represented by a DAG (directed acyclic graph) called task graph.

In large-scaled systems, tasks in a task graph are managed and scheduled to processors by a scheduler dynamically based on certain scheduling algorithm. To have better understanding of task graph scheduling, students need to learn the algorithms. However, learning such algorithms are not easy for many students for several reasons:

- There are many models to describe the behavior of processors in real life. Students can be confused by the variety.

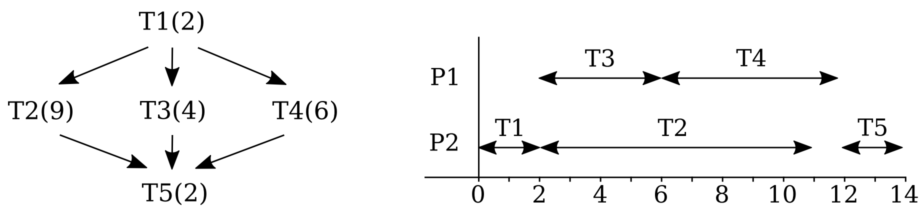
- Algorithms are usually given based on a certain model. For other models, there might be many variants that are slightly different, making things more confusing.
- Task scheduling requires predicting states of the cluster for a long duration. This is hard because it requires good imagination and detailed understanding of the behavior of models.
- Some algorithms requires sophisticated control over the timeline, or have complex mathematical model which is hard to understand.

This project aims to develop an game-like application to help the students learn concepts in task graph scheduling, in addition to algorithms. For any schedule, it can simulate the execution timeline based on a variety of cluster configurations. It also provides a step to step tutorial to help students learn the mechanisms and algorithms. For tutors, this application can also be used for demonstration.

Chapter 2

Background

2.1 Task Graph Scheduling



(a) task graph

(b) schedule

Figure 2.1: Example of task graph and one possible schedule

The topic of this project is task graph scheduling. Figure 2.1a shows an example task graph.

2.1.1 Communication Models

2.1.2 HLEFT Algorithm

2.2 Educational Software

Chapter 3

Design

3.1 Learning Experience

3.1.1 Game modes

Similar to games, this application is divided into many levels. Each level can have different game mode for different purposes:

- **Static mode:** In this mode, students are given several task graphs and a cluster. Students can create schedules by scheduling tasks in task graphs to processors in the cluster. When one schedule is created, it can be executed to generate the timeline, so that the student can improve the schedule according to the execution timeline. Each level can have several target times. When all the tasks are finished, the performance of the schedule will be evaluated according to the targets.

- **Dynamic mode:** Different from static mode, the state of the cluster will be simulated in real time. After the start of game, several task graphs will be revealed at certain time point. The student is required to schedule the tasks to processors when the simulation is running. This mode requires the student to analyze task graphs quickly and make schedules immediately. Similar to static mode, the time cost to finish all the tasks will be recorded and the performance will be evaluated based on target times.

- **Tutorial mode:** Levels in this mode are usually developed based on static mode. For tutorial levels, some help text will be displayed to help the student learn concepts, operations and algorithms. A tutorial can operate on the game engine freely. By listening to operations made by the student, tutorials can be made into an interactive process to help students learn faster.

- **Sandbox mode:** This mode is developed based on static mode for testing purposes. The user can create games by selecting clusters and task graphs, then play the the created game freely. This mode also provides several standard algorithms, so that the user can try these algorithms to check the scheduling result, making it good for demonstrations.

3.1.2 Design of Interface

3.2 Execution Logic

3.2.1 Communication Models

As is described in section 2.1.1, one difficulty in task graph scheduling is variety of communication models. To reflect the variety, four different communication models are selected as follows:

1. **Ideal (immediate) communication (IC):** Communication do not cost any time. Tasks will only be delayed if any of its dependency is not finished.

2. **Background communication with multiple channels (BCMC):** One processor can communicate with unlimited amount of processors in both directions. The only limit is one processor can only have one channel sending to another processor, meaning no more than one communication block can be sent from one processor to another at any time. The limit is made since bandwidth of one connection is always limited in real life, although there could be multiple connections.

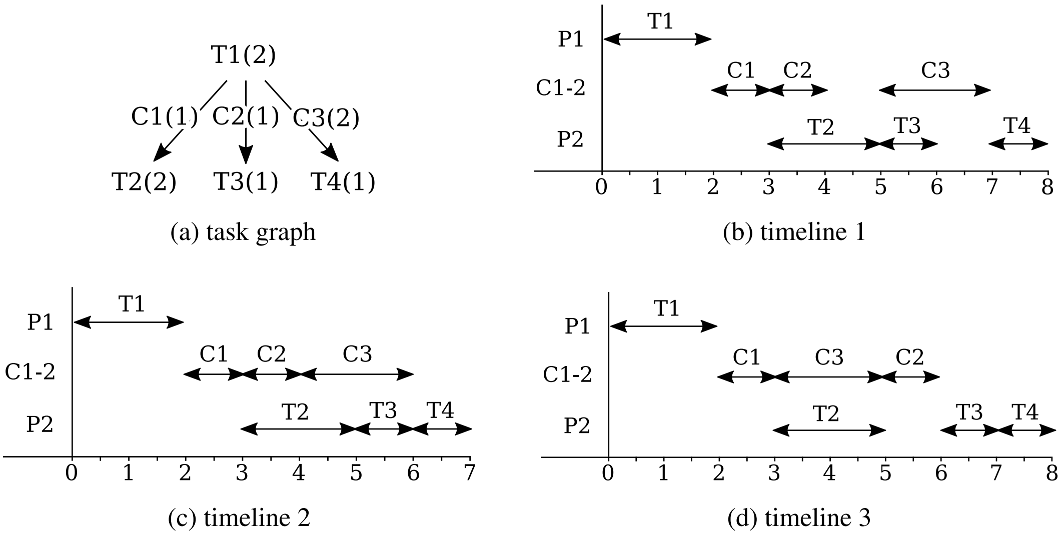
3. **Background communication with single channel (BCSC):** One processor can send to or receive from only one processor at any time. Instead of having multiple connections, this model describes processors with single connection, and the total bandwidth is limited. Therefore, one communication in progress can occupy the entire bandwidth, making other communications blocked.

4. **Synchronous (blocked) communication (SC):** One processor cannot execute tasks and communicate with other processors at the same time. Also, it allows

only one channel in one direction like described in mode 3. This model describes the scenario when using synchronous communication libraries like Java IO or MPI synchronous mode in single thread.

3.2.2 First Conflict and Rule 1

With more strict communication models, there can be more conflicts. One option is to let the student decide how to solve the conflicts, but sometimes it makes the learning experience too detailed and annoying, so the program have to add more rules as tie breaking strategies to simplify the process in such cases.



(a) task graph

(b) timeline 1

(c) timeline 2

(d) timeline 3

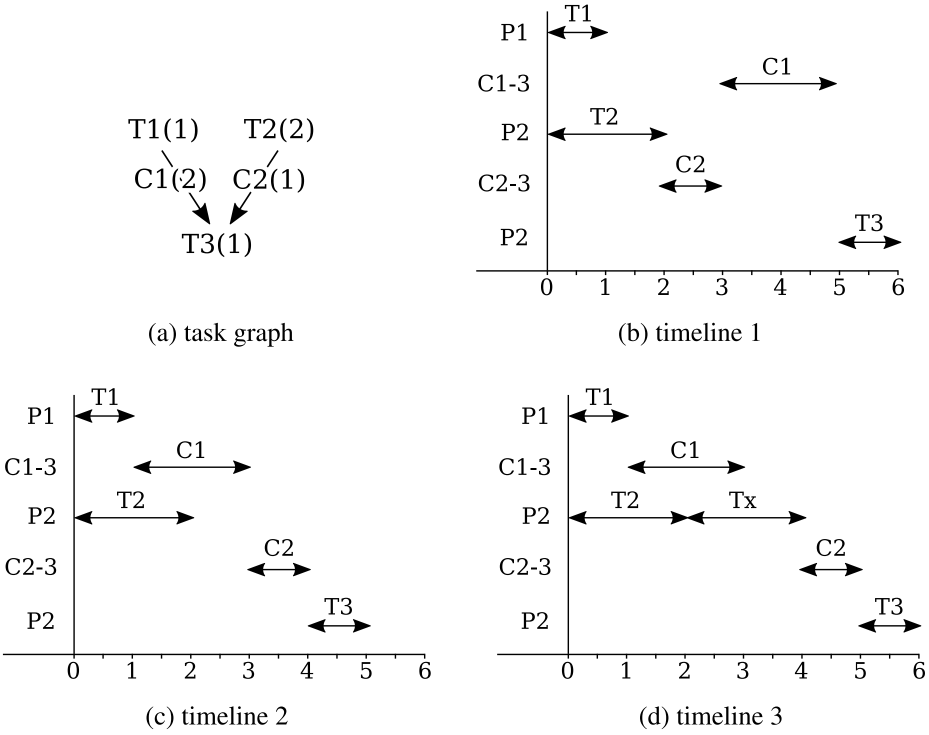
Figure 3.1: One task graph and possible timelines to demonstrate effect of communication order in non-ideal models

In all non-ideal modes, one conflict is when two tasks scheduled on one processor relies on data from another processor, because two data packages have to be sent in certain order. The strategy used is to allow communication required by one task only when it is the first task in the schedule (rule 1). In other words, communication required by one task will be delayed if there is any task ahead of it in the schedule. Figure 3.1 shows one example task graph and three possible execution timelines if T1 is scheduled to processor 1 (P1) and remaining tasks are scheduled to processor 2 (P2). For the task graph, tasks are labeled as “Tn(Duration)” and communications are labeled as “Cn(Duration)”.

As shown in the timelines, for one given schedule, there could be many results if the order and time of tasks are not explicitly specified. However, according to ”rule 1”, the execution result will always be timeline 1. Although other strategies can provide better performance like in timeline 2 (also known as early fetch), this rule is chosen for its reliability, simplicity, and less uncertainty. Another option is to leave the decision to students. However, there are two problems: 1) It have to be decided based on very precise estimation of execution, which might be too challenging for a student, even for many algorithms; 2) It will make the interface very complex because it requires precise control of time.

3.2.3 Second Conflict

Another conflict happens only for single channel models, which is the order of communication for one task. Figure 3.2 shows an example of the conflict. For task graph given in 3.2a, by scheduling T1 to P1, T2 to P2 and T3 to P3, even when rule 1 is applied, there are still multiple possible execution results, which are shown in figure 3.2b and 3.2c.



(a) task graph

(b) timeline 1

(c) timeline 2

(d) timeline 3

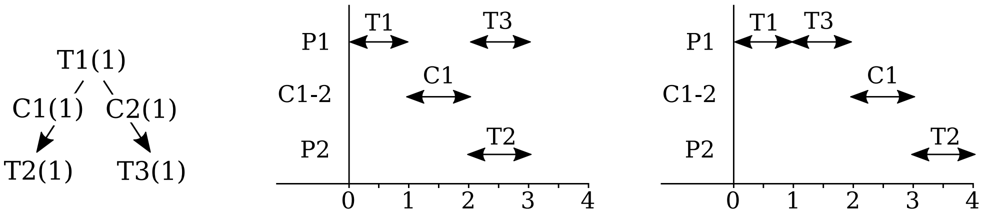
Figure 3.2: One task graph and possible timelines to demonstrate effect of communication order in single channel models

The main problem is the order of communications that T3 depends on. When C1 is executed first, the result is figure 3.2b and when C2 is executed first, it generates figure 3.2c. A straight forward solution is to always attempt to receive the data that is available earlier, which is C1 in this case, similar to greedy strategies. According to the figure, it provides better performance indeed. However, in some other cases, when another task Tx is assigned to P2, assuming synchronous communication model, C2 can be terribly delayed.

According to the timelines, it seems changing the order of communication can have significant effect over execution of other tasks, especially when the resources are limited. Therefore, the decision is left to the student. In multiple communication models, since there is no such conflict, the system will handle communication automatically, while in single communication models, the student have to decide the order manually. However, it does not mean tasks being executed can be paused to execute communications. When one task have been under execution, communications will be delayed until execution finishes.

3.2.4 Third Conflict and Rule 2

In synchronous communication model, there is also conflict between execution of tasks and communication. Figure 3.3 shows an example of the conflict when T2 is scheduled to P2, and remaining tasks are scheduled to P1. When T1 finishes, there are two options: communicate with P1 first (figure 3.3b), or execute T3 first (figure 3.3c).



(a) task graph

(b) timeline 1

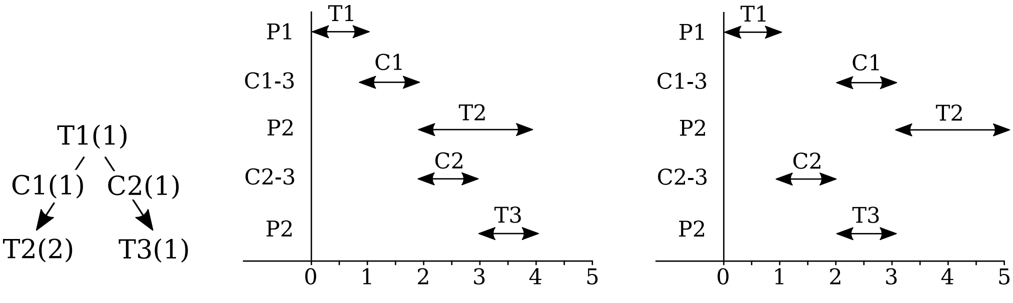
(c) timeline 2

Figure 3.3: One task graph and possible timelines to demonstrate conflict between execution and communication in synchronous communication model

As can be observed in figures, if processors are allowed to execute next task before communication, it is possible to block other processors for a long time, when there are many connected tasks. To remove unnecessary delay caused by this conflict, the choice is to force processors communicate before executing tasks (rule 2). Under this rule, the result execution will always be figure 3.3b.

3.2.5 Other Conflicts and Behavior

There are still many conflicts that are not mentioned. For example, figure 3.4 shows two possible execution results for the task graph given in figure 3.4a when scheduling T1 to P1, T2 to P2 and T3 to P3. However, since such conflicts do not happen as frequent as previously described ones, and the effect to general performance is negligible in most of cases, the behavior is not explicitly defined. Also, defining too much rules for details also brings more complexity for students to learn. Instead, the behavior when such conflicts happen depends on the implementation of simulation engine.



(a) task graph (b) timeline 1 (c) timeline 2

Figure 3.4: One task graph and possible timelines to demonstrate effect of different output order in single communication models

3.3 Platform and Libraries

3.3.1 Programming Language

This project is designed to be a cross-platform desktop application. While C++ is the typical choice for such requirements, Java is chosen as the main language in development.

The biggest difficulty of using C++ is compilation on different platforms. For example, programs on Windows are usually compiled against all DLLs (dynamically linked libraries) selected and provided by the developer and distributed with all its dependencies bundled. However on Linux, programs are usually compiled against SOs (shared objects) provided by system, then distributed as source file or single binary file. Such difference brings complexity in compilation and potential issues in distribution.

Oppositely, compilation of Java file is much easier. With help of virtual machine, compiled binary files can be executed on different platforms without any extra step.

Since Java executable files are usually packed in Jar files, distribution is also convenient.

3.3.2 GUI Library

As is said in previous sections, the interface is designed to be interactive, which means the application will heavily rely on operations like hovering and drag & drop. Also, it requires rendering overlays and transparency frequently. For widgets based traditional GUI frameworks, these operations usually requires usage of complex or low level APIs, which brings difficulty in development and potential compatibility issues. Therefore, games are usually developed based on dedicated GUI frameworks.

GUI frameworks used in games are usually built on low level libraries like DirectX and OpenGL. One reason is they are usually directly connected to hardware operations, which saves much performance in rendering complex shapes. Another reason is these libraries allows GUI frameworks developed in immediate mode, making it easier to develop highly dynamic scenes. Compared to retained mode, the developer do not need to refresh windows manually in immediate mode because every frame is refreshed and rendered separately.

OpenGL is chosen as the rendering library for its cross-platform availability and simplicity. Although OpenGL do not provide APIs in Java, there are several libraries in Java providing the bridge. Among these libraries, LWJGL 3 is chosen for several reasons:

- It includes bridges to several convenient native libraries like STB and GLFW.
- It has very good documents and community support.
- It provides full exposure of OpenGL APIs.
- It it up-to-date.

With help of LWJGL, programs written in Java can still easily access native libraries, while there is no need to have special compilation steps for different platforms because the dependencies are already included and handled by LWJGL.

Chapter 4

Implementation

4.1 General Architecture

4.2 Simulation Engine

4.2.1 Choice of Execution Method

There are mainly two ways to predict the execution results: calculate the timeline based on mathematical relationships (calculation), or simulate the execution process, then record the states (simulation). Assume this question: predict how much time does it take to execute one task that takes 2 seconds, on a processor with 2x speed. For calculation methods, the only thing required is calculate $2 \div 2 = 1$ second. On the other hand, here is an example for simulation methods. First cut the time into ticks, for example 20 ticks per second. Therefore, tasks that takes 2 seconds is equivalent to 40 ”work packages”. Secondly, calculate how much work can be done within one tick. For a processor with 2x speed, it is 2 ”work packages” per tick. The final step is to keep execute ticks until finished work on the processor reaches 20 ”work packages”. By calculating difference in tick count, which is 20 ticks in this case, it can be concluded the execution takes 1 second.

It seems calculation methods are much simpler and efficient, but the simulation method is used for several reasons:

- It’s hard to find a generic expression for different scenarios in calculation methods. For example, in ideal communication model, for one task, the earliest start time is maximum of the finish times of all its dependencies and processor’s earliest available time. While for BCMC model, the finish times of dependencies will be changed to finish time added by communication time, if the tasks are executed on different processors. For similar reasons, the expression will be very complex for some communication models. However for simulation, the developer only need to describe the actual behavior of one processor, according to requirements of the communication model, which is usually called transition functions. Therefore, changing in communication models is equivalent to changing the transition function, which is significantly easier to implement and validate.

- The execution uses tasks as unit, rather than time in calculation models. In other words, the tasks are atomic. For example, when one task is executed by calculation, it takes the entire period in the timeline immediately. Because of this nature, the developer have to check the states of all related processors in the entire period for conflicts. Also, the execution order of tasks have to be carefully arranged for correct behavior. Although this problem can be resolved by always executing tasks scheduled to the earliest available processor, for tasks with communications, it will become very complicated because workloads are tightly coupled and shared by processors.

- Simulation models are more suitable for animations. This application is required to “run” a schedule. When cursor on the timeline moves, simulation models are actually running the tick under the cursor. Therefore, the animated execution is actually rendering the history of simulation engine in real time. However for calculation modes, extra steps are required to ”make up” the animated execution.

4.2.2 Execution of Simulation

The simulation engine is constructed based on a model close to clusters in real life. It maintains a list of processors, each keeping an input buffer, a list of active communication channels, a task being executed and an output buffer, although the output buffer is not stored explicitly inside processor objects for global visibility.

When simulation is running, it follows a very simple logic: keep executing ticks until there is nothing to be executed on all processors. One tick has three phases, listed as follows:

1. `tickPre1`: Every processor checks task queue to prepare for next task to execute. If next task requires communication with other processors, establish communication channel with target processor.

2. `tickPre2`: If the processor is available and all dependencies are met, fetch the task from task queue and prepare for execution.

3. `tickPost`: Execute communications and task by updating the progress. If communication or task is finished, update the state of processor and buffers accordingly.

One tick is separated into 3 phases for two reasons:

- As required by rule 2 described in section 3.2.4, communications will always be executed first if there is conflict between communication and execution of tasks. Therefore, in the first phase, all processors will check for communications and allocate resource for it. When resource is occupied by communication in some communication models, task will not be fetched in the second phase, thus ensures higher priority for communications. The two phases cannot be merged because there is usually coupling between processors. For example, if processor A is ticked first and it fetches one task with no dependency, processor B, the later ticked, cannot establish communication channel with processor A because resource in processor A is already occupied, which violates rule 2.

- Between phase 2 and phase 3, the state is written to history. The basic concept is processors only decides what to do in one tick inside the two pre-tick phases, while no execution is performed, making the end of phase 2 suitable to record the state. If phase 2 and 3 are merged and history is updated at phase 3, one apparent consequence is that execution of tasks taking only 1 tick will never be recorded because the execution is finished inside the tick. Actually, the result is execution of every task will miss 1 tick in history. There is work-around for this problem, but to keep the states clean, phase 2 and phase 3 are kept separated.

The progresses of all communication and execution of tasks are stored as double-precision float numbers, because the base speed and speedups of processors are allowed to be float numbers, using 0 for started and 1 for finished. When using float numbers in simulations, the developer need to be careful with the error. For example, if a processor completes 20% of the task (0.2 out of 1), when 5 ticks are completed, the progress might not be 1. The progress is usually a number slightly smaller or larger than 1 because 0.2 cannot be precisely represented by float numbers.

4.3 GUI Framework

4.3.1 Rendering Method

4.3.2 Widgets and Layout

4.4 Data Driven Format

4.5 Algorithms and Estimator

4.6 Event System

4.6.1 Tutorials

Chapter 5

Result

5.1 Compilation and Distribution

5.2 Game Flow

5.2.1 Appearance and Components

5.2.2 Tutorial Levels

5.2.3 Game Levels

5.2.4 Sandbox Mode

Chapter 6

Evaluation

6.1 User Testing

6.2 Code Quality

Chapter 7

Conclusions

7.1 Future Suggestions

7.2 Final Comments

Bibliography