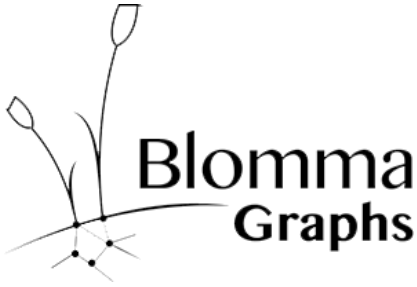

BlommaGraphs

A scheduling framework

Faculty of Computer Science
Summer semester 2013
Semester Bac4

26th July 2013

Project team: Benjamin Wöhrl
Manuel Oswald
Ziad Nörpel
Simon Kerler
Richard Stromer

Table of contents

How to run the project

There are fourclasses that can be run:

- GraphSetGenerator
  Is used to create the set of TaskGraphs. It generates 100 graphs for each group of 10, 50, 100, 300 and 500 nodes and saves them in ./exports/graphs.
- GraphSetScheduler
  Takes the graphs generated by the GraphSetGenerator, schedules them and saves them into ./export/schedules.
- StatisticsBuilder
  The StatisticsBuilder also takes the TaskGraphs generated by the GraphSetGenerator, schedules them, calculates statistics and saves them in ./exports/statistics/statistics.html. For configuration of the StatisticsBuilder see the chapter in of sprint 5.
- EvaluationBuilder
  Run the practical evaluation for a given task graph or schedule and exports a side by side comparison to the ./exports/evaluation/ folder.

Sprint 1

Presentation

The presentation for the first sprint can be found here:

http://noxan.github.io/blomma-graphs/presentations/sprint1

Data structure

Generator

For the Generator we implemented a interface for different kinds of GraphGenerators. It includes several methods: Setter methods to set attributes which vary the generated graphs and a generate method to produce a random graph depending on the attributes and the seed. The seed could be set to rebuild the generated graphs if necessary.

For our project we implemented a DefaulTaskGraphGenerator which generates a TaskGraph object depending to the setted attributes and seed by calling the generator method.

To generate the whole amount of graphs we implemented a class called GraphSetGenerator. This class produces a new DefaultTaskGraph instance and uses the setters to set the attributes

for the graphs. After that, it generates the needed graphs, serialize them and safe the strings into files ordered in directories representing the nodecount.

Serializer

We defined a simple interface for TaskGraph serializers with two methods: serialize and deserialize. The first one takes a TaskGraph instance and returns a String with the serialization. The second one just takes a serialization string and returns a newly build TaskGraph.

We implemented a serializer for the STG format described in http://www.kasahara.elec.waseda.ac.jp/schedule/ which uses a helper class called STGNode to grab the required data from the graph easier.

Sprint 2

Presentation

The presentation for the second sprint can be found here:

http://noxan.github.io/blomma-graphs/presentations/sprint2/

Scheduler

SystemMetaInformation

The SystemMetaInformation class is used to pass information e.g. number of cpus to the schedulers.

LAST

The basic idea of the LAST algorithm is to calculate a DNODE value which is used to determine the task which should be scheduled next. However, the two papers linked below come up with two very different implementations of the algorithm.

At first we tried to implement the second one which is much more complicated. Every processor has a so called frontier list in which all tasks that have a connection to a task that is already scheduled on this cpu are stored. That means that it is possible to choose a task for scheduling for which not all dependencies are fulfilled. To solve this problem it uses a mechanism that is called backward allocation: All previous tasks that are not scheduled are compressed into an input tree which again is scheduled by DNODE values until all dependencies are scheduled.

After a long discussion we decided to use the simpler implementation described in the first paper which is faster to implement and takes less computation time. In this approach frontiers are only

filled with tasks which dependencies are already scheduled.

We still have some methods in the LASTScheduler class related to the complex implementation which are currently not used.

References:
http://homepage.lnu.se/staff/oldaaa/2dv005-2013/kwok.pdf
http://www.eng.auburn.edu/files/acad_depts/csse/csse_technical_reports/CSSE91-14.pdf

DLS

The next task to be scheduled will be chosen by the highest Dynamic Level. The dynamic level is the difference between the B-Level and the earliest start-time of a task on a CPU.

There are several elements in the algorithm:

B-Level: path to the end task which takes the most time
earliest start-time: earliest start time of a task on every CPU
dynamic level: B-Level - earliest start-time per CPU
ready pool: pool of ready task to be scheduled

At the beginning the ready pool will be initialized with the first task of the graph and it's B-Level. The main loop will after this run till all tasks has been scheduled. For all tasks in the ready pool the earliest start-time per CPU will be computed. In the next step the one with the highest dynamic level will be chosen and scheduled to the right CPU. In the last step of the loop the new tasks with their B-Level will be added to the ready pool.

References:
http://homepage.lnu.se/staff/oldaaa/2dv005-2013/kwok.pdf

Genetic

The genetic scheduler tries to optimize a given initial schedule or wraps another basic scheduler to optimize this schedule furthermore. It is very flexible to the user's needs and covers highly optimised schedules as well as high performance concerning the algorithm duration itself. This can be easily configured by setting the number of generations.

Internally it utilizes chromosomes which represent a schedule. Or to be more precise they save in a two dimensional data structure the mapping of tasks to processors, which is handled during the encoding process. Later the chromosomes are decoded into a schedule again which will select the first ready node, which has all of it's dependencies fulfilled, and assigns it to the processor defined in the chromosome to the earliest possible start time. A single schedule will always be encoded in the same chromosome as well as a chromosome will always be decoded into the same schedule.

There are three kinds of chromosomes concerning their generation process. One is generated out of the given initial schedule. For each processor one is generated which maps all tasks to a single processor. And additionally there is the option to add more random generated chromosomes, which obviously assign the tasks to a random processor.

For each generation the genetic scheduler splits it's population in two pools, a elitism one and a mating one and applies a swap mutation on the mating pool, while the elitism one will stay untouched. At the end of each generation the two pools are mixed and the selection algorithm picks the chromosomes which result in the best schedule - the one with the earliest finish time. This process is repeated for the given number of generations, as defined by the user on his or her needs.

The results of the genetic scheduler are most likely to be better than the given schedule but the selection during each generation guarantees in the worst case to end with a schedule equal to the initial one.

References:
http://homepage.lnu.se/staff/oldaaa/2dv005-2013/kwok.pdf
Link

Serializer

Format

We are using a text based format to serialize ScheduledTaskLists with a short and an extended version. The short one consists of starttime, cpu_id and node_id. The extended format adds computation and communication time.

For example "`10 0 1`" means that task number 1 is scheduled on cpu 0 and starts at time 10.

Serializers

At first we just implemented serializers for both formats which work nearly the same as the TaskGraph serializers: they take a ScheduledTaskList and serialize it to String and backwards.

In sprint four we also implemented a HTML serializer which is described later on in this documentation.

Sprint 3

Presentation

The presentation for the third sprint can be found here:

http://noxan.github.io/blomma-graphs/presentations/sprint3

Stream schedulers

Instead of scheduling a single TaskGraph our StreamSchedulers take a set of TaskGraphs. A helper class StreamSchedulableArrayGenerator is used to create those sets. It takes a single TaskGraph and an array with deadlines, copies the incoming Graph and sets the deadline until there are as many TaskGraphs as entries in the deadlines array.

Basic stream scheduler

The basic stream scheduler is most likely a wrapper for the basic schedulers implemented in the third sprint. It takes the set of TaskGraphs and merges them into a single TaskGraph which is passed to the underlying basic scheduler and just returns its result.

Custom stream scheduler

The idea was to try to get a schedule with the smallest gaps to get the highest throughput. We used the following elements which are partly pretty similar to the one of the dls:

- earliest start-time: earliest start time of a task on every CPU
- ready set: set of ready task to be scheduled
- gap: time between the finish-time of the previous task and the earliest start-time of the task to be scheduled on a cpu
- phantom task: ready task with its EST and gap depending to a CPU
- phantom task list: list of current phantom tasks

After initializing the ready set the CSS will build the phantom task list and chose the next task depending to the gap, deadline and earliest start-time. After choosing the first task of the phantom task list which is the one with the smallest gap it will repeat this step till every task has been scheduled. After scheduling all tasks the deadlines will be checked. If the schedule doesn't meet all deadlines the CSS will chose the next task from the phantom list of the last step. If all tasks from the phantom list of the last step have been checked it will chose the next task of the previous step. It will repeat this till the schedule will meet all deadlines or no more possibilities are left.

Sprint 4

Presentation

The presentation for the fourth sprint can be found here:

http://noxan.github.io/blomma-graphs/presentations/sprint4

Framework updates

Performance improvements

We started with profiling our framework during different load scenarios and successfully evaluated the bottlenecks of our framework - without touching any of the scheduling algorithm implementations, which will be covered during the 5th sprint. As profiler we used the Java VisualVM, which is shipped together with the default Java SDK.

The major focus of our performance improvements has been the implementation of the task graph data structure. We added multiple caches of different internal collections to reduce the cpu load and more important the access times in exchange for a slightly increased memory consumption. Looking at the results multiple functions, like the findEdge and containsEdge, which were the reason for about 99.8% of our cpu time, now do not show up anymore or just consume close to 0.2%.

Furthermore we added sorting to many other internal collections and replaced the hash based with tree implementations. This had the benefit to reduce search times drastically and gave the framework another significant performance boost.

We additionally implemented a binary object stream based serializer for our task graphs as well as we tried some other improvements but those have never been actively used because the performance improvements were minor compared to the other factors.

Schedule validation

We implemented a simple validation for our schedules, which performs the following checks:

- whether all deadlines are fulfilled
- whether all dependencies are met

and returns an enum, which has defined the values ‘valid’, ‘invalid deadline’ and ‘invalid dependency’.

HTMLSerializer for ScheduledTaskLists

A HTMLSerializer for ScheduledTaskLists was built, to display the scheduled tasks on each processor scaled by time, to get a visual feedback.

Practical evaluation

Approach

We implemented the practical evaluation to start a Java thread for each processor and execute the tasks of the given schedule as soon as all their dependencies are met. So we use the theoretical start time of the given schedule just to sort the tasks. We feature a highly flexible interface for the fake work implementations. At the moment there are two different implementations, one time and one load based. Those will be called to simulate the computation time as well as the communication time with the reason to keep a correct ratio of the communication to computation. In the end the practical evaluation returns a scheduled list, so it is the same data structure for input and output. But it contains the values from the execution which are scaled to be better comparable to the input values.

Time based worker

It saves it start time and yields its cpu time for the given work to fake multiplied by a constant factor until the time it should wait is finally exceeded.

Load based worker

The current implementation calculates fibonacci numbers to a given value which is again scaled by a constant factor. We used the fibonacci numbers because they provide a good amount of load, the algorithm is easy to understand and implement and the Java VM is not able to optimize it to much during its multiple executions, unlike a simple loop or counter.

Result

The results of the practical evaluation are compared to the original schedule and are presented as an HTML file for the user. For this the results are scaled to be better comparable and viewed in a side by side comparison (on the left the original schedule, on the right the result of the practical evaluation). Furthermore the initial task graph is exported to easily check the dependencies and input parameters.
An example for the result of a practical evaluation can be found on the following website:
http://noxan.github.io/blomma-graphs/presentations/sprint4/practical-evaluation.html

Theoretical evaluation

The theoretical evaluation is done by the StatisticsBuilder. An example for theoretical evaluation can be found here:
http://noxan.github.io/blomma-graphs/presentations/sprint4/statistics.html

The StatisticsBuilder takes the TaskGraphs generated by the GraphSetGenerator, schedules them, calculates the statistical values mentioned below and saves them in a statistics.html file. It also generates files for every TaskGraph which contains information about the graph and its schedule.

Configuration

The StatisticsBuilder can be configured in its source code:

- taskGroupCount: The number of task graph groups that are scheduled. 1 means that only TaskGraphs of size 10 are scheduled and 5 means that all sizes of task graphs are used.
- taskGraphGroupSize: How many TaskGraphs in each group should be scheduled.
- cpuCount: The number of CPUs of the system we schedule for.
- blockSize: How often the TaskGraph appears in one block.
- schedulers: An array that holds instances of all schedulers for which statistics are calcualted.

Statistic values

- Graph
  ID of the TaskGraph
- Nodes
  (Average) number of nodes in TaskGraph
- Edges
  (Average) number of edges in TaskGraph
- T
  (Average of) number of TaskGraphs per block divided by single block execution time
- CpD
  (Average) critical Path Duration
- AD
  (Average of) how long the scheduler took to schedule
- SBET
  (Average) single block execution time
- SCpR
  (Average of) single block execution time divided by critical path duration
- SCpV
  (Average of) critical path minus single block execution time
- AC
  Average communication time

Main file

The main file contains three tabs:

1. Graph
   Information about a single task graph ordered by scheduler. By clicking on a row you get the description file forone TaskGraph (see below).

2. Group
   Information about a group of task graphs with a particular number of nodes ordered by scheduler.

3. Scheduler
   Average statistics for one particular scheduler.

TaskGraph files

The TaskGraph file is split into two areas. At the top you can see the TaskGraph rendered with arbor.js and its STG format representation. At the bottom you can see the acutal schedule rendered by our HTMLSerializer and in our extended scheduling format.

Sprint 5

Optimization

LASTScheduler

Idea

Currently the LASTScheduler appends tasks only at the end of a processor queue which means there can be gaps of idle time between two processes that are never filled up. Considering this we developed (but not implemented because we had not enough time for testing etc.) a strategy that does not only append a task but also checks if the task can be put into an idle time gap to improve throughput.

Procedure

1. Calculation of latestDependencyTime
   a. Iterate through all tasks on which the task depends that is scheduled next.
   b. latestDependencyTime is calculated by finishing time of dependend task that finishes last plus communication time.
2. Find fitting gap
   a. Iterate through all taks on the processor and calculate gaps by subtracting the finishing time of a task from the start time of the following task.
   b. Check for each gap if the task that is scheduled next fits into it.
   c. If all dependencies are fulfilled for the gap (latestDependencyTime >= start time of

gap), put the task into it otherwise continue searching for the next gap (b).

3. If no gap is found append the task like we did it until now.

Complications

- Gaps are bypassed if the latestDependencyTime is greater than the start time of the gap although the task could fit into it by offsetting it until the dependency time is fulfilled.
- It is possible that this optimization improves the throughput just a little but increases the algorithm duration much more.
- Takes long to get a error resistant implementation. There are also many special cases that have to be considered.

CustomStreamScheduler

In general the time to execute the CSS depends on the deadlines. The throughput could become better with shorter deadlines but will mostly cause a longer execution time. Especially the shorter the deadlines are the much longer time will it takes to execute.

Idea

Currently the chosen task from the phantom task list depends only on the gap, deadline and earliest start-time. With taking the communication time in account it could be possible to get a higher throughput. If tasks have the same gap, deadline and earliest start-time and depends on the last task of a cpu it would be better to schedule the one with the higher communication time to the cpu to save the difference between the communication times in the best case.

Genetic Scheduler

Problem

First of all the genetic scheduler was not designed for stream processing and just tries to minimize the finish time. The modified version for stream processing also drops all schedules with invalid deadlines which is rather a simple workaround. Furthermore the selection of the next task during chromosome decoding is just based on the task IDs to always get the same schedule with the same chromosome.

Idea

The genetic scheduler would benefit most from using more selection criterias than just the finish time. So in terms of optimizing throughput it would be the most important criteria to use this as most important selection criteria. Another important value would be the reduction of communication times and the reduction of computation gaps or better known as idle time. Another point to start from would be the mutation of the chromosomes. Currently only a simple swap mutation is applied as well as there is no more complex or more efficient logtic to the inheritance over multiple generations.

Other improvements

There are plenty of other improvements possible most of them are related to the algorithm duration and performance. So it would be good to cache the decoded schedule together with

each chromosome, because this schedule is used several times during each generation and is only changed during the mutation where the cached has to be updated. And there are multiple other values to cache, like the earliest start time on each processor during the decoding of a chromosome, the finish time of a decoded chromosome schedule and several others.

Summary of project results

Since the steps of the project work were to get a introduction to task graphs, scheduling and stream processing, the main goal over the last sprints was to optimize our framework and schedulers in particular for throughput.

Over the time we found several factors which influence the throughput. The ones most important in a negative way are gaps (idle times), waiting for dependencies, idle times during communication times and unnecessary communication time, which could be saved by scheduling on the same processor.

So obviously it was our goal trying to minimize those negative factors already during the scheduling process itself. Furthermore we tried to use as many as possible processors to make the most of the our available resources without wasting too much time on counterproductive communication costs. Another difficulty which was added with the stream processing were the deadlines, which were like in real time system. We always tried to give the deadline handling a top priority because of its importance.