Critical resources management and scheduling under StarPU

CHEVALIER Arthur

Supervised by:
WACRENIER Pierre-André
GUERMOUCHE Abdou

April-September 2017

Contents

Abstract

The advanced users of StarPU have the ability to regulate the task submission to limit/control the consumption of critical resources (memory, network communication). This technique can penalize performance (parallelism deficit) and is not adapted to the concurrent management of several types of resources. The use of hierarchical tasks allow to refocus this management at the level of the scheduling engine: high-level tasks are used to express parallelism and resources requests. Once this graph is submitted, StarPU has the clairvoyance necessary to optimize the parallel execution of the application while respecting the constraints expressed.

Greetings

This brief was made possible thanks to the cooperation of several to whom I would like to express my gratitude.

I would like to begin by expressing my gratitude to my internship directors, WACRENIER Pierre-André and GUERMOUCHE Abdou, for their patience, availability and, above all, their judicious advice, which has contributed to my reflection.

I would also like to thanks the two INRIA teams (STORM and HiePACS) who welcomed me and made my stay here a moment that I would not forget.

And finally, I would like to express my gratitude to the friends and colleagues who have given me their moral and intellectual support throughout my journey.

Chapter 1

Introduction

I realized my internship in the STORM ("STatic Optimizations - Runtime Methods") team of the research center Bordeaux Sud-Ouest of INRIA. This team works on the high-performance computing which is the use of parallel processing for running advanced application programs efficiently, reliably and quickly[1]. To cope with the increase in the amount of digital information and the complexity of modern systems, it is necessary to improve the structures allowing the processing of this information, since high-performance computing is one of the keys to this. Currently, a multitude of fields is interested such as weather forecasting, object modeling, physical simulations, material resistance calculations, and simulation in finance and insurance.

The rise of the multi-core architectures equipped with some accelerators in the high-performance platforms introduced hybrid configurations which have a direct impact on the high-performance application development. It has now become crucial to provide abstractions in order to use the full potential of those machines. The classic way of running an application on regular processors while sending predetermined parts of the code on accelerators is not sufficient anymore. The new advanced programs are programs which are spread across the whole machine with a runtime dynamic management of the processing units (GPU included). The problem of scheduling dynamically the program over all the available processing units have been answered in the thesis of AUGONNET Cédric[2]. This task-based abstraction paradigm allows us to apply some advanced scheduling techniques on top of a multi-typed processing units machine easily.

The objective of this stage was to implement within StarPU a generic scheduler to apply classical resource management policies (eg. the Dijkstra Banker algorithm). In a second phase, we will concentrate on memory management by applying recent techniques developed within the ANR SOLHAR.

[1] The term applies especially to systems that function above a teraflop or 10^{12} floating-point operations per second.

Chapter 2

StarPU

2.1 State of the Art

The architecture of computers has widely changed over the ages (little ages). The very first attempt to add parallelism in high-performance computing was adding instruction level parallelism and increase the clock speed. The next step was to add cores to increase the thread level parallelism but this leads to a bus contention. Indeed the symmetric access to the main memory with all those cores represented a lot of transfer. The solution found was the cache hierarchy which was a good solution at the time, decreasing the number of access to the common memory but this wasn't made for scale on several dozens of processing units on a big configuration for high-performance computing.

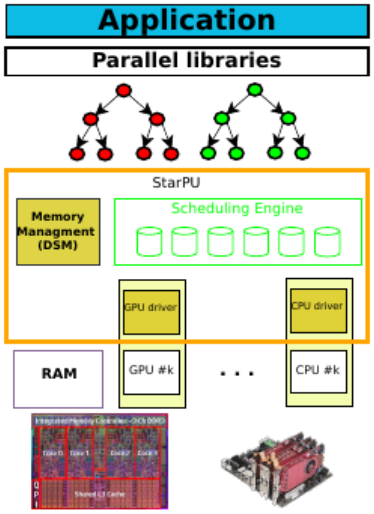
NUMA (Non-Uniform Memory Access) solve this problem with a distributed memory around the machine. The processing units grouped in a node have fast access to their memory but a longer delay to access others nodes memories. Despite all of this the contention on the memory bus is still a big problem unless the application handles it with very fine grain.

The evolution of computers is now the addition of more and more cores per processors, what was anticipated by GPU at the time. The problem was those GPU was designed to process images, but now with the evolution of the mind for the high-computing performance science, they are better as specialized computing devices with a well-defined goal: a large power of computation and fine grain of parallelism. With this trend, several Application Programming Interface (API) were developed, like CUDA and OpenCL, and evolve fastly. Today programs for GPU can be written in common languages like C according to a Single Program Multiple Data (SPMD) parallel programming model which opened the way to General Purposes GPU (GPGPU) computing.

However, the application programmers find it difficult to achieve considerable performance when using it in addition to the work needed to implement the kernels. Not only the way of coding is different in GPU but the memory bus contention (again) become hard to handle when using others processing units. Despite this problems, the hybrid configuration is the new trend because of the need to exploit and computing power in a machine.

Complex applications which want to use some GPUs with CPUs requires a big amount of work in order to efficiently make a profit of the heterogeneity

Figure 2.1: Runtime system structure (StarPU here)



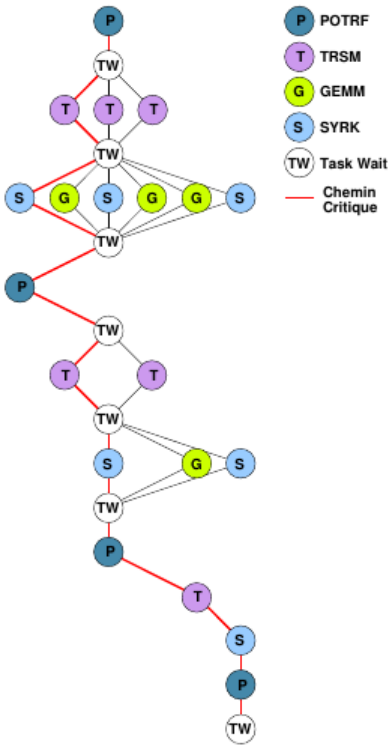
of the processors, their frequency, the hierarchical memory and the memory capacity. A solution to simplify the interaction between software and hardware is runtime systems. They represent a user level software which brings the basic, general purposes functions provided by the operating system. The application delegate all the parallelism management to them (scheduling tasks for example) and the runtime optimize the use of the heterogeneity of the configuration. Over many existing runtime systems, we will interest only in StarPU[3].

2.2 Task-graph structure

Most of the times, parallel applications can be represented as a graph of nodes representing operations and edges representing the dependencies between those tasks. With this representation, we can schedule each node independently instead of the whole program with some control points. In this report, we will not be interested in how such task-graph is generated and so we will assume that a parallel program can be represented by a task graph as in the figure 2.2.

With this approach we can do what we called sequential task flow (STF), we can submit all function tasks and the runtime system will compute a tree and add parallelism. In the algorithm 2.1 you can see a simple sequential code which does things to two variables x and y. The first function will read and write those two variables as the fourth one but the two in the middle (sub_b and sub_c) will only read those variables to do something else. We can modify the code into a sequential task flow like the algorithm 2.2, we submit in this algorithm the four tasks (which are the four functions) with the read/write markers. The generated task-graph of this algorithm will be the one in figure 2.3 and we can see that parallelism has been added thanks to the STF.

Figure 2.2: Example task graph



Algorithm 2.1 Example of sequential code

Input:

```
1: a(x, y); // Read and write for x and y
2: b(x); // Read for x
3: c(y); // Read for y
4: d(x, y); // Read and write for x and y
```

Algorithm 2.2 Equivalent of algorithm 2.1 with STF code

Input:

```
1: submit(a, x:RW, y:RW);
2: submit(b, x:R);
3: submit(c, y:R);
4: submit(d, x:RW, y:RW);
5: wait_tasks_completion();
```

Figure 2.3: Example STF code graph

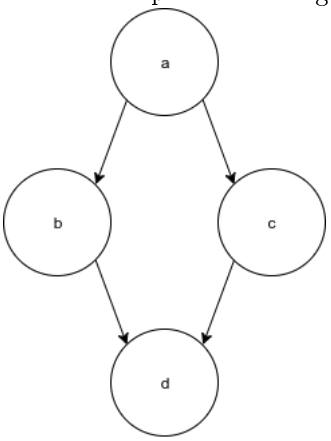
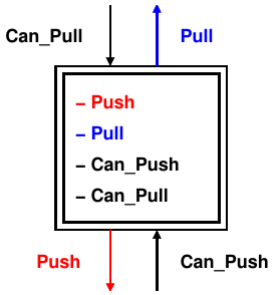


Figure 2.4: Scheduler component[9]



2.3 StarPU scheduling

Now that we see how task-graph are generated and how works the sequential task flow to submit those tasks, we need to see how StarPU handle those tasks and how the scheduling works in it. First, StarPU has two control points to handles the tasks: push and pop. The push function is called when a task is ready (all its dependencies are resolved) and we want to send it in the scheduler. The pop function is called when a worker (a CPU or GPU) has nothing to do and want a task to execute. Pushing only ready tasks allow not to overload the scheduler but decrease the clairvoyance in it. For example, the scheduler cannot know if tasks with more priority will be pushed in it later.

StarPU allow to implement our own scheduler with some "components", it is called a modular scheduler (see 4.2 in the thesis of Marc SERGENT [9]). The principle of such scheduler is that you can assemble components to handle the flow of tasks as you want. Each component has specific functions (see figure 2.4) to compute the tasks as you want. The functions push and pull allow to, respectively, push task on a son component and ask for a task in the father component. The two control functions mentioned before will call those functions to push and get tasks. Each component can, therefore, do some treatment on the tasks which goes in it and redirect to a specific son if it has several.

2.4 Bubbles

As explained above, pushing only the ready tasks decrease the clairvoyance in the scheduler. That's why the bubbles principle is actually developed in StarPU. Bubbles are hierarchical tasks, that is when a bubble is executed (on a worker) a subgraph of tasks is generated. This behavior allows new scheduling strategies depending on several criteria, the opportunity to exploit some computation unit types at a given time or even reserving resources for future tasks.

If we take the example of the algorithm 2.3 and the figure 2.5, we can see that four bubbles are present and when the bubble 1 will be executed then 4 tasks will be submitted in StarPU (tasks 1, 2, 3 and 4). The gain of the bubbles is more control over the task-graph submission and a power to make decisions (like block the bubble for a lack of resources at a time t for example) before all the tasks are created. We have more control because we can decide to execute a bubble before another and so change the execution order of the task-graph due to some information we get (worker occupation, resource constraint, model prediction). In addition, the grain of the bubbles gives us a look at what will be executed in the future and once again be able to choose whether or not we want to submit all the sub-graph of tasks. This abstraction allows to previsualize the task graph without having to submit all tasks (which can be very costly).

In this example the graph of bubbles in the figure 2.6 will generate the task graph in figure 2.5 when they will be executed !

Another advantage is the critical resources management, indeed the usage of bubbles could allow to reserve resources or anticipate a lack of resources before it happens without having the consumption of the tasks on the machine. If for example, a bubble possesses information on the memory consumption of all its tasks then we can know in advance the impact it will have on the scheduler et so take decisions.

Finally, the sequential submission of tasks in StarPU can have an impact on performances of an application in function of the number of tasks. The utilization of bubbles allows making this submission parallel. Indeed, in the example of the figure 2.5 we can see that only 4 bubbles are submitted and that the tasks 1-4 and 5-6 will be submitted in parallel while StarPU is executing others tasks which allow to "smooth" the submission of tasks and not having thousands of tasks from the beginning.

In this report, we will be interested only in the critical resources management (more precisely the memory) that we can do thanks to the control given by the bubbles when they are scheduled. Depending on which resources are available and what are the requests we will let go a bubble or not.

Figure 2.5: Big task-graph with bubbles

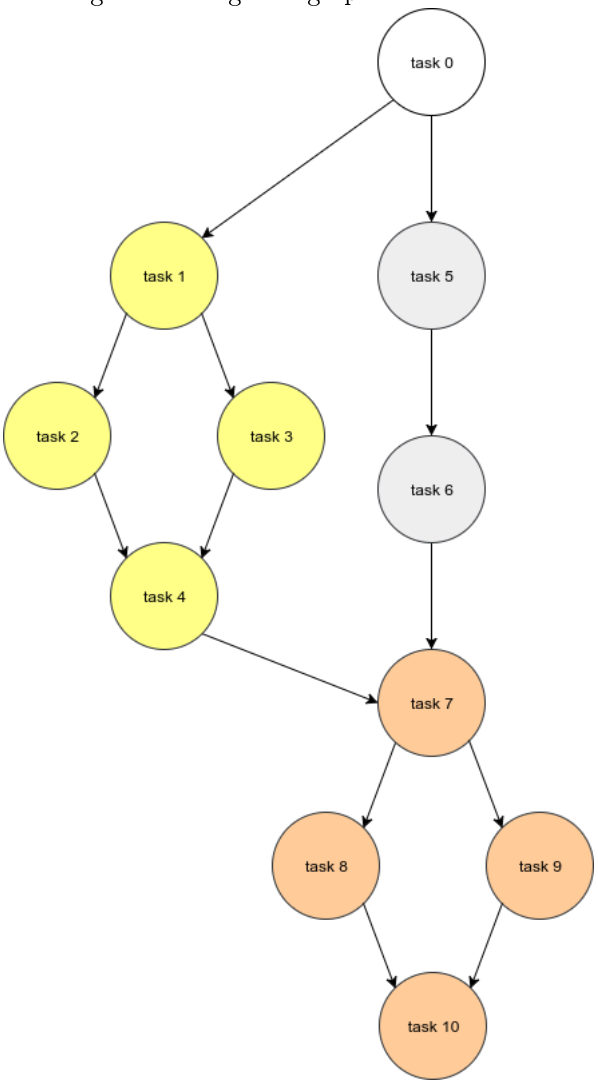
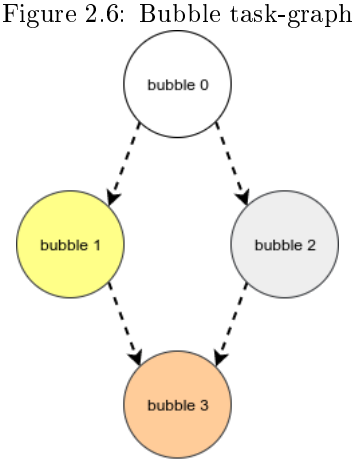


Figure 2.6: Bubble task-graph



Algorithm 2.3 Bubble STF example

Input:

```
1:  procedure BUBBLE_0
2:      submit(task0, x:RW, y:RW, z:RW);
3:  end procedure
4:  procedure BUBBLE_1
5:      submit(task1, x:RW, z:RW);
6:      submit(task2, x:R);
7:      submit(task3, z:R);
8:      submit(task4, x:RW, z:RW);
9:  end procedure
10: procedure BUBBLE_2
11:     submit(task5, y:RW);
12:     submit(task6, y:R);
13: end procedure
14: procedure BUBBLE_3
15:     submit(task7, x:RW, y:RW);
16:     submit(task8, x:R);
17:     submit(task9, y:R);
18:     submit(task10, x:RW, y:RW);
19: end procedure
20: submit_bubble(bubble_0);
21: submit_bubble(bubble_1);
22: submit_bubble(bubble_2);
23: submit_bubble(bubble_3);
24: wait_tasks_completion();
```

Chapter 3

Memory constrainted scheduling

In this chapter, we will talk about the memory constraint scheduling techniques (previous and actual) and we will present two algorithms that do such things. The first one is a basic algorithm which can ensure to stay under a memory peak and the second one is a recent algorithm proposed in late 2016 ([4]) that do the same with a better approach to add parallelism. First, we have to introduce the model used and the differents notations for those algorithms.

3.1 Notations and application model

3.1.1 Model

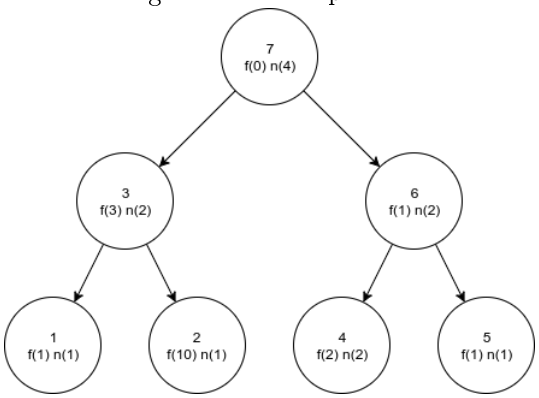
For those algorithms, we consider a tree-shaped task graph T composed of n nodes, or tasks, numbered from *1* to *n*. Nodes in the tree generate temporary data, which are released at the end of the execution, and terminal data, which stay in memory until the end of the whole graph. Each node has also input data, more precisely:

- Each node i in the tree generate terminal data n_i
- Each node i generate temporaries data f_i. If i is not the root, those temporaries data are used as input data for its parent *parent(i)*; If i is the root, it's temporaries data are of size 0.
- Each non-leaf node i in the tree has inputs data from its sons. We denote *Children(i)* the sons of i. For each child $j \in Children(i)$, the task j produce a temporary data f_j for i. If i is a leaf, then $Children(i) = \emptyset$ and i has no input data.

During the processing of task i, the memory must contain temporaries data, terminal data and input data. The memory needed for this processing is thus

$$\left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i$$

Figure 3.1: Example tree



After i has been processed, its input data are released and in the case of out-of-core, the terminal data are also released. Only the temporaries data stay in memory (until the processing of its parent).

Each algorithm has three inputs, the tree to process, a sequential order of execution and a memory peak, which has to be at least the memory peak reached when processing the tree with the given order.

We can see in figure 3.2 how memory varies over bubble execution with the example tree (figure 3.1).

3.1.2 Notations

In the following algorithm we will use some notations for functions or variables that can be baffling. The tree in the input is the task-graph of the problem and the postorder is the tree traversal we want to use to compute the memory peak AND ordering the bubbles, it has to be the same for both. The function *inputs* is the sum of the sons temporaries memories or in a mathematical expression:

$inputs(i) = \sum_{j \in Children(i)} f_j$

The function PO returns the position in the post-order of the node in argument. As you will see in the algorithm 3.2 two priorities are used AO and EO. AO is the Activation Order, it's the post-order for the traversal of the tree to schedule tasks. EO is the Execution Order, it's the priority to execute on the processors, a higher priority will be executed before a smaller one. Those two notations are used to respect the ones in the algorithm paper [4].

3.1.3 Example of execution

If we consider the post-order 1,2,3,4,5,6,7 with a sequential execution the memory consumption will look like the figure 3.2. With the bubbles numbers in the x-axis and the memory on the y-axis, we see that when the bubble 1 finish, the memory occupied is 2 and the execution of the bubble 2 bring 11 more memory ($f(2) + n(2) = 10 + 1 = 11$). When a father is executed like the bubble 3 it will start to occupied the maximum memory so we add $f(3) + n(3)$ and when this bubble finally end, the temporaries memory of these sons are released ($f(1) + f(2)$) so we decrease the memory of 11 and so on. This flow of execution

Figure 3.2: Memory consumption in example tree

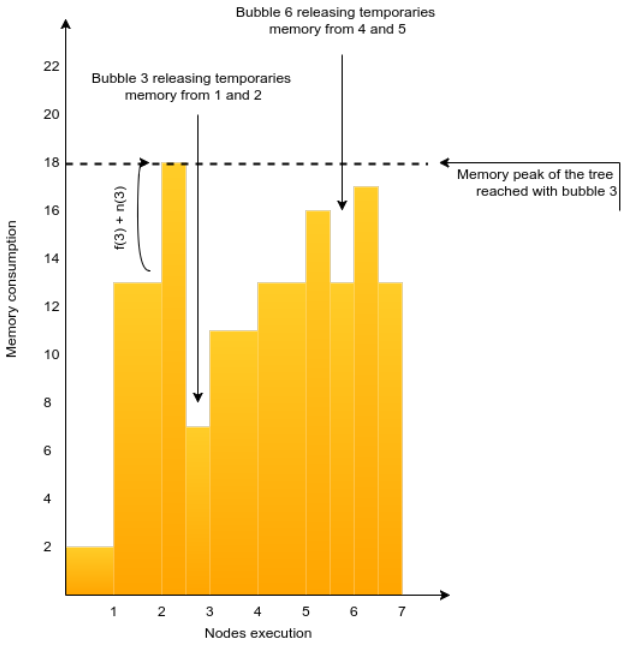
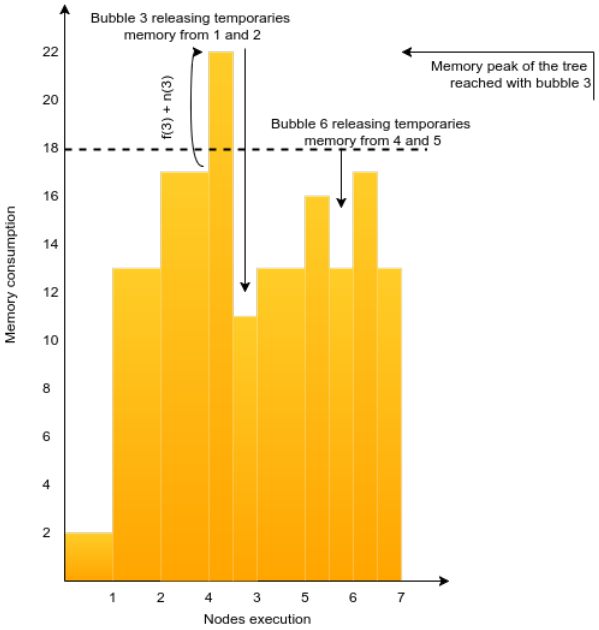


Figure 3.3: Memory consumption in example tree with parallelism



brings clearly the memory peak reached on 18 of memory. Now if we consider the same post-order but with a parallel execution we could end up with the figure 3.3 where we see an increase of memory during the bubble 3 because the bubble 4 added memory before the release of $f(1) + f(2)$. In this case, if we compute the memory peak with the sequential order so 18 and if we let the bubble 4 be executed before the bubble 3 we raise a deadlock. This is why we need memory constraint algorithm which is capable of detect deadlock sooner.

3.2 Peak-memory minimizing: State of the Art

The problem of scheduling task-graphs under memory constraint is a well-known problem, first, the sequential approach has been extensively studied by Liu. He first proposed an algorithm to find a peak-memory minimizing traversal (with post-order) which we use to generate the sequential order of execution for the bubbles[7]. Several algorithms were then found which allow staying under a memory bound while allowing to execute nodes in parallel [7]. Then, he presents an optimal algorithm to solve the general problem, without post-order [8]. After this, the parallel case was studied and a first paper of peak-memory minimizing in parallel ruled that it is a NP-Hard problem and so they proposed to study a variant of this problem [6]. This proposition has strongs hypothesis and recently (end of 2016) another paper proposed an advanced algorithm in the general case that could ensure to stay under the memory peak while adding more parallelism [4].

3.3 Two algorithms answering the memory constraint problem

3.3.1 Base algorithm

The goal of the base algorithm is to limit the execution of the nodes in order to be very conservative on the memory. To ensure the constraint we allow a node to be processed when all precedent non-booked nodes in the sequential order get into the memory. After allowing a node we book all the memory needed for those nodes and remember the last node accepted. Despite the fact that this algorithm is sure it presents some defects. First, we have to be omniscient of the current state of the graph and secondly, if the memory peak of the tree is on the bottom the algorithm will fastly block submission of tasks. Due to the fact that it has to book all the memory in the sequential order; If we take a tree in the form of two big subtree the first leaf of the second one will have to book all the first subtree nodes, which is disastrous for parallelism.

3.3.2 Dynamic memory booking algorithm

The dynamic memory booking algorithm[4] is a smarter algorithm, it's a enhanced version of the memory booking algorithm[6]. The goal of this algorithm is to reuse memory of the children when we process the parent. Each node, when it has finished its execution, will reserve memory for its father and, if this father has already enough memory to its execution, it will reserve memory for

Algorithm 3.1 Basic algorithm(T, PO, M)

Input: tree T, postorder PO, memory limit M (not smaller than the peak memory of the sequential traversal defined by PO)

1: $M_{used} \leftarrow 0$
2: $LastNodeProcessed \leftarrow -1$
3: **while** the whole tree is not processed **do**
4: Wait for an event (task finished or starting point of the algorithm)
5: **for** each finished task i **do**
6: $M_{used} \leftarrow M_{used} - inputs(i)$;
7: **end for**
8: **for** each ready node i **do**
9: $Needed_i \leftarrow \sum_{j=LastNodeProcessed}^{PO(i)} (f_j + n_j)$;
10: **if** $M_{used} + Needed_i \leq M$ **then**
11: $LastNodeProcessed \leftarrow PO(i)$
12: $M_{used} \leftarrow m_{used} + Needed_i$;
13: Break;
14: **end if**
15: **end for**
16: **end while**

its grand-father and so on. This technique allows consuming far less memory than normal because a father node can be in the memory footprint of its sons or grandsons. In fact, the higher a node is in the tree the more memory footprint it will be allowed to use for its execution. This method of booking memory allow a faster process of the tree because far less memory is needed due to the reuse of memory. You can find the algorithm written in algorithm 3.3.

3.3.3 Comparison of execution

In this subsection, we will compare the two executions of those two algorithms on the tree for the figure 3.4.

Algorithm 3.2 Init of Dynamic Memory Booking Algorithm[4]

Input: T, AO, EO

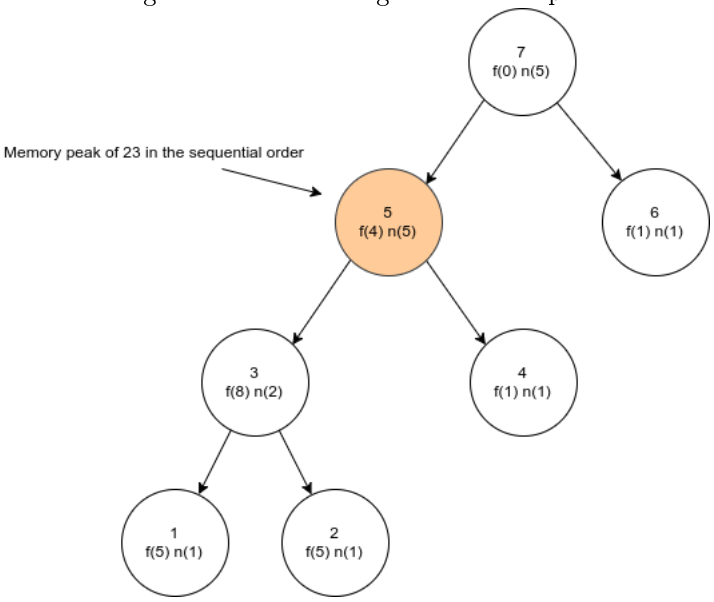
1: $CAND \leftarrow AO$ sorted-heap (init: Leaves(T); fun: AO-insert, AO-remove, AO-min)
2: $ACTf \leftarrow EO$ sorted-heap (init: empty; fun: EO-insert, EO-remove, EO-min)
3: ChNotAct[1..n] $\leftarrow$ array of size n (init: $\forall i$, ChNotAct[i] $\leftarrow |Children(i)|$)
4: ChNotFin[1..n] $\leftarrow$ array of size n (init: $\forall i$, ChNotFin[i] $\leftarrow |Children(i)|$)
5: NotUnCand[1..n] $\leftarrow$ array of size n (init: $\forall i$, NotUnCand[i] = false)
6: Booked[1..n] $\leftarrow$ array of size n (init: $\forall i$, Booked[i] = 0)
7: BookedBySubtree[1..n] $\leftarrow$ array of size n (init: $\forall i$, BookedBySubtree[i] = -1)
8: $M_{Booked} \leftarrow 0$

Algorithm 3.3 Dynamic memory booking algorithm[4]

Input: T, p, AO, EO, M

1: $INIT(T, AO, EO)$
2: **while** the whole tree is not processed **do**
3: Wait for an event (task finished or $t = 0$)
4: **for** each just finished node j **do**
5: $B = Booked[j]$
6: $\begin{cases} Booked[j] \leftarrow 0 \\ M_{Booked} \leftarrow M_{Booked} - B \\ BookedBySubtree[j] \leftarrow 0 \end{cases}$
7: $i \leftarrow parent(j)$
8: **if** $i \neq NULL$ **then**
9: $ChNotFin[i] \leftarrow ChNotFind[i] - 1$
10: **if** $ChNotFin[i] = 0$ and $BookedBySubtree[i] \geq MemNeeded_i$ **then** EO-insert$(i, ACTf)$
11: **end if**
12: $\begin{cases} Booked[i] \leftarrow Booked[i] + f_j \\ M_{Booked} \leftarrow M_{Booked} + f_j \end{cases}$
13: $B = B - f_j$
14: **end if**
15: **while** $i \neq NULL$ and $BookedBySubtree[i] \neq -1$ and $B \neq 0$ **do**
16: $C_{j,i} = min(B, max(0, MemNeeded_i - (BookedBySubtree[i] - B)))$
17: $\begin{cases} Booked[i] \leftarrow Booked[i] + C_{j,i} \\ M_{Booked} \leftarrow M_{Booked} + C_{j,i} \\ BookedBySubtree[i] \leftarrow BookedBySubtree[i] - (B - C_{j,i}) \end{cases}$
18: $B = B - C_{j,i}$
19: $i \leftarrow parent(i)$
20: **end while**
21: **end for**
22: $WaitForMoreMem \leftarrow false$
23: **while** !$(WaitForMoreMem)$ and $CAND \neq \emptyset$ **do**
24: $i \leftarrow AO$-min$(CAND)$
25: **if** $BookedBySubtree[i] = -1$ **then** $BookedBySubtree[i] \leftarrow Booked[i] + \sum_{j \in Children(i)} BookedBySubtree[j]$
26: **end if**
27: $MissingMem_i = max(0, MemNeeded_i - BookedBySubtree[i])$
28: **if** $M_{Booked} + MissingMem_i \leq M$ **then**
29: $\begin{cases} Booked[i] \leftarrow Booked[i] + MissingMem_i \\ M_{Booked} \leftarrow M_{Booked} + MissingMem_i \\ BookedBySubtree[i] \leftarrow Booked[i] + \sum_{j \in Children(i)} BookedBySubtree[j] \end{cases}$
30: AO-remove$(i, CAND)$;
31: **if** ChNotFin$[i] = 0$ **then** EO-insert$(i, ACTf)$
32: **end if**
33: ChNotAct$[parent(i)] \leftarrow$ ChNotAct$[parent(i)] - 1$
34: **if** ChNotAct$[parent(i)] = 0$ **then** AO-insert$(parent(i), CAND)$
35: **end if**
36: **else**
37: $WaitForMoreMem \leftarrow true$
38: **end if**
39: **end while**
40: **while** there is an available processor P_u and $ACTf \neq \emptyset$ **do**
41: $i \leftarrow EO$-min$(ACTf)$; EO-remove$(i, ACTf)$
42: Make P_u process i
43: **end while**
44: **end while**

Figure 3.4: Tree for algorithms examples



We will use a memory peak of 24 like the sequential order and submit the tasks in the post-order of the id of the bubbles. For the first algorithm (the basic one), we can see that the submission stop when we try to push bubble 4 (see figure 3.5). Indeed, when the first bubble arrives we only need 6 memory which brings us to a consumption of 6 when the second bubble arrives. For the second bubble, we need 6 more memory which fit so we let go the bubble. For the fourth bubble (the third is not ready yet), we have to book memory for the third and the fourth (the last bubble processed is 2) and it is the last bubble that will fit. In fact, with this algorithm, while the bubble 3 is not ended the bubble 6 will never be able to go which is a big bottleneck.

For the second algorithm, the two first bubbles are placed in memory (if they don't pass the process will not go far), and then as you can see in the figure 3.6 the bubble 4 is placed right after. Indeed the bubble 3 will use the memory of bubbles 1 and 2 when it will be executed and, with this principle, all the bubbles fit in memory for a peak of 24. You can see that those two algorithms have completely different behaviors and that they handle memory with two ways. Finally, if we play with the number of tasks in each bubble we can make the two algorithms converge in performances, indeed, in the extreme case of one bubble the two algorithms will have the same behavior. Same if we had two bubbles or three, so if we have a few bubbles the two algorithms will be likely the same, but if we have many bubbles the second algorithm will be far better than the first.

Figure 3.5: Basic algorithm consumption example

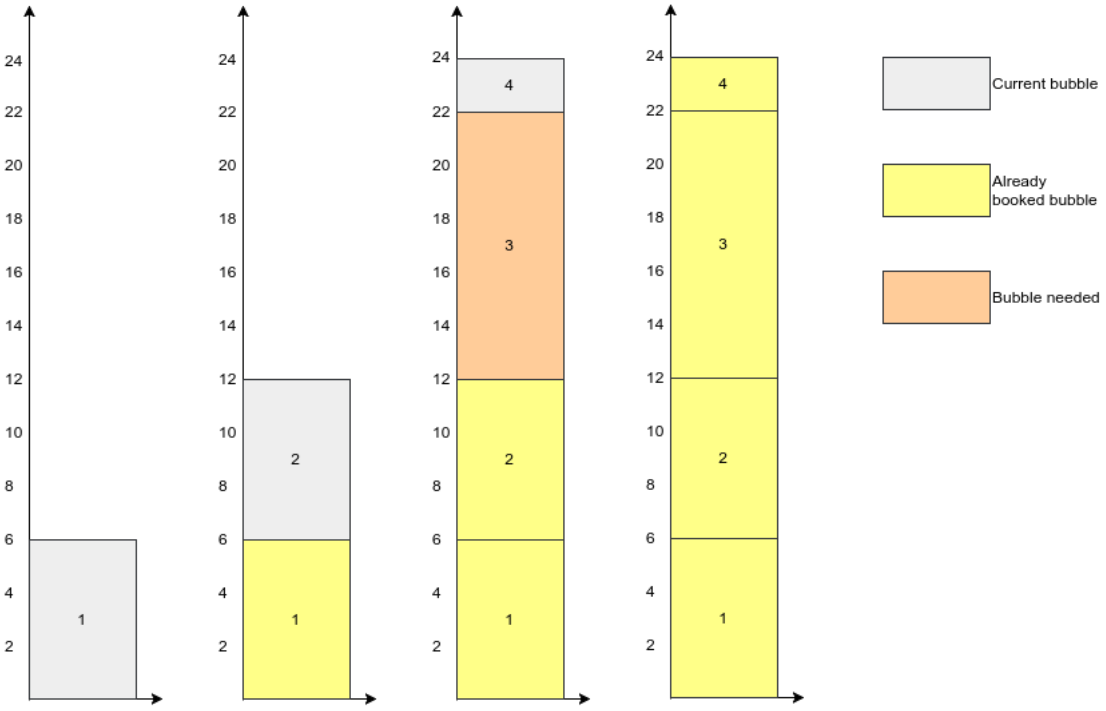
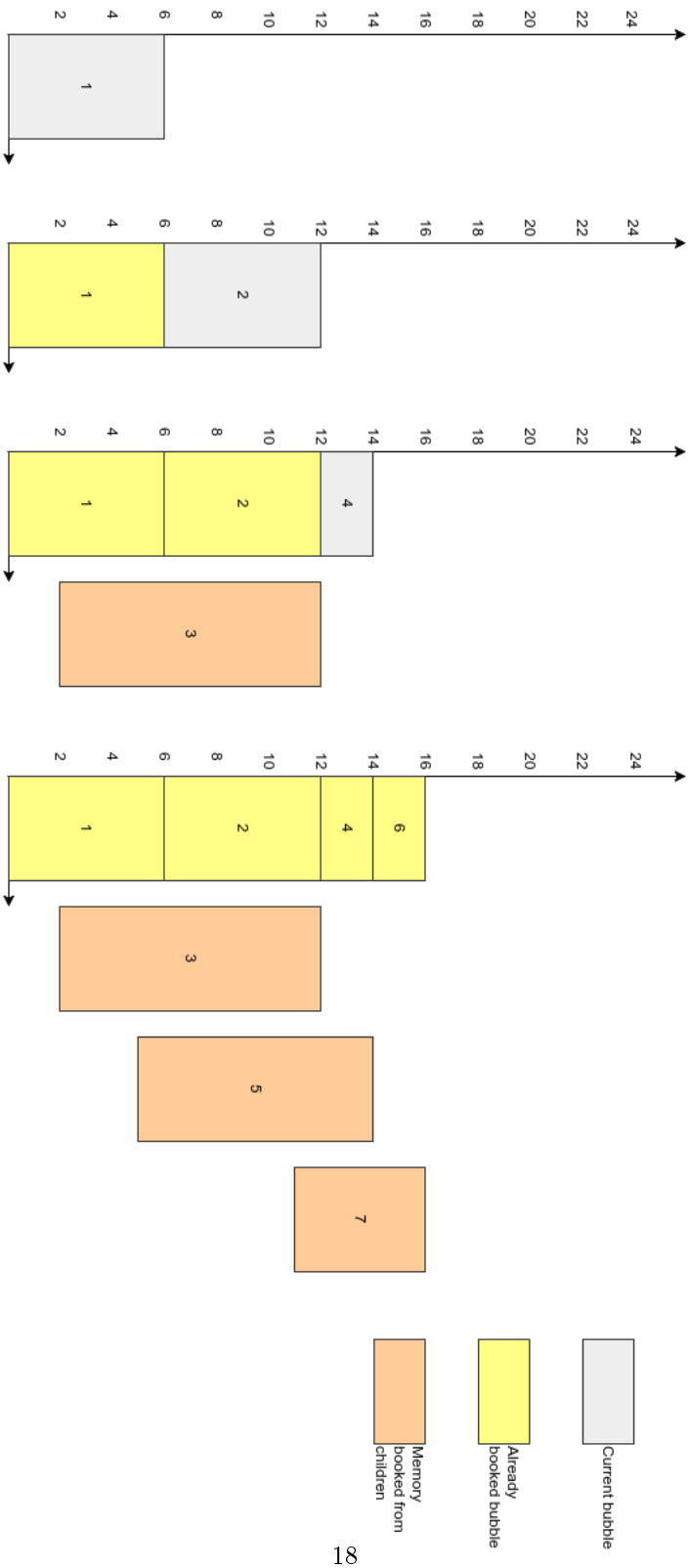


Figure 3.6: Dynamic memory booking algorithm consumption example



Chapter 4

Contribution

4.1 Introduction

In this chapter we will present our contribution to the problems evoked upper. The previously explained concept of modular scheduler allows an easier way to describe a scheduler that can apply some deadlock avoidance algorithms. The key concept of our contribution is a component which may be capable of communicate with others components in order to get information about a termination of a task or algorithm actions for example as you will see later. The scheduler will have to handle bubbles and tasks without increasing too much the computation cost of this last. We also contribute to the communication inter-components which does not exist in StarPU without giving pointers to message's target components. Another problem was the lack of ways to describe and specify constraints that we wanted to apply to the scheduler, in 4.2.2 we will propose a structure to fill the gap. We also have implemented all those solutions in StarPU and test them with an existing and hypothesis conforming application which is qr_mumps. All those tests are shown in the chapter 6.

Algorithm 4.1 Previous processing flow of constraint management in applications

Input: tasks set TS, max memory $AvailableMemory$

```
1: CurrentMemory ← 0
2: for T ∈ TS do
3:     while CurrentMemory + MemoryNeeded_T > AvailableMemory do
4:         Wait();
5:     end while
6:     CurrentMemory ← CurrentMemory + MemoryNeeded_T
7:     Insert task T
8: end for
9: WaitForCompletion();
```

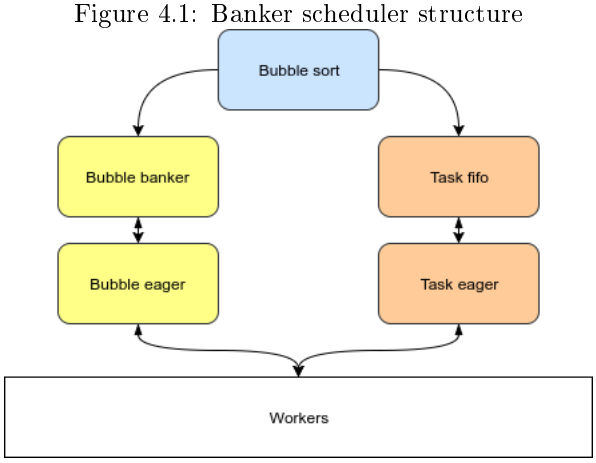
4.2 Banker scheduler

Currently, the applications that want to do some deadlock avoidance with constraints computed in the application but not in the execution support. Our contribution here is a way to move this detection further down in StarPU to avoid non-experience users having to do this.

As you can see in the algorithm 4.1, when a task needs too much memory the application waits for the available memory to decrease before inserting the task and update the current memory.

The banker scheduler is a modular scheduler which allows applying some constraints to the parallel execution of the tasks. It can differentiate bubbles from tasks and detect when it's smart not to let pass a bubble because of a future deadlock. The banker component in this scheduler hold a queue of bubbles and compute when to let them pass or not thanks to the differents algorithms applied by the app's user. Now they can insert all the bubbles with the correct information, don't care much about the constraints application and use an advanced algorithm developed by experts in the domain.

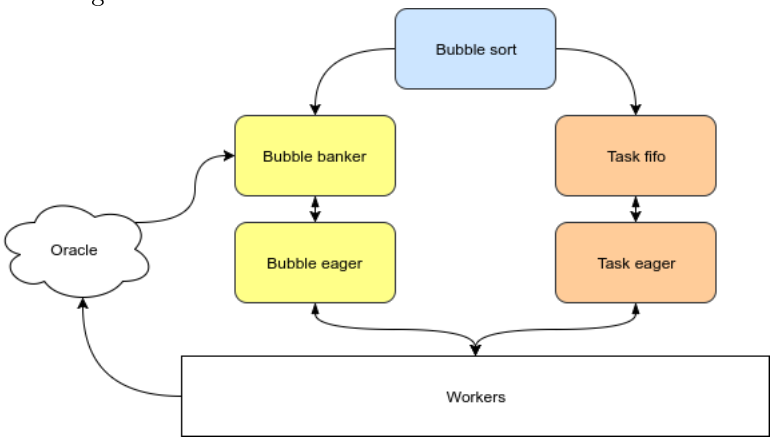
Figure 4.1: Banker scheduler structure



4.2.1 Oracle

In order to communicate efficiently between components, we must have implemented a way to send messages without performance loss (not to much at least) with parameters to indicate to the banker when an event happened. In the general case, we could have done an event queue and with the specific structure of a modular scheduler send all the events to the first component which then transmit them to its sons and so on. The problem is the scheduler is used thousands of times in one second which totally prevent this method. We developed a more high-performance appliance solution which can send messages to another component fastly without losing much performance and with a ratio contribution/cost much higher.

Figure 4.2: Banker scheduler structure with the oracle



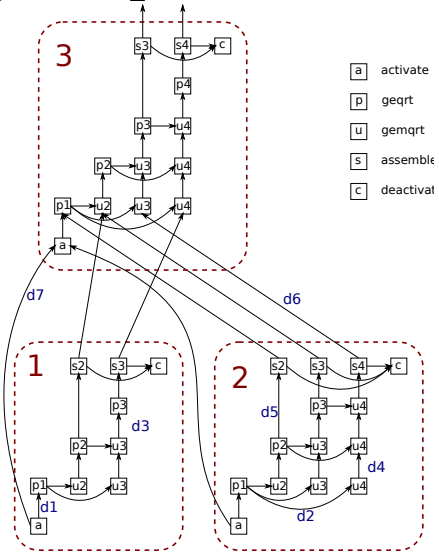
4.2.2 Constraints

The final problem was to find a way to describe efficiently the constraints with a generalist approach in order to be able to implement every constraint we want. On top of that, a constraint can be implemented with several algorithms, each having its strengths and weaknesses. The user must be able to apply some constraints and choose between the available algorithms (or even implement some for the courageous) easily without necessarily knowing much about constraint and deadlock avoidance. A smaller problem in this approach is that this structure is called thousands of times too can't take much time because of the high-performance objective so if the scheduling time (including constraint computation) take some percents of the total time, the scheduler is not good at all.

4.3 StarPU integration

The bubble features of StarPU being fresh new, the addition of constraint management must have started with the development of an interface compatible with the possible future utilization of this last. In order to add the constraints, we must have added a structure which holds the information of this constraint and the corresponding algorithm. Another problem was how to add information for each bubble and how to link these two when the bubble is scheduled. Our contribution in StarPU is a way to create a constraint, link it to the scheduler and add information to each bubble with the ability to retrieve them when they go through the banker component. Thanks to that the user can implement its own algorithm for a constraint without the need of modification in the banker scheduler.

Figure 4.3: qr_mumps bubbles example [1]



4.4 qr_mumps integration

4.4.1 What's qr_mumps ?

qr_mumps[5, 1] is an application which solves sparse, linear matrix on multicore computers. It implements a direct solution method base on the QR factorization of the input matrix. The solving of the matrix is split into four phases: the analysis, the factorization and the solution in two operations:

$Rx = b$ or $R^T x = b$

$y = Qx$ or $y = Q^T x$

We will be interested only in the factorization phase which computes the QR factorization. We selected qr_mumps to use the constraints algorithms because the tree model used in it matches ours perfectly and this application offers a wide range of matrix to test. As you can see in figure 4.3 qr_mumps already is capable of regroup some of its tasks in the QR factorization in order to generate bubbles to insert into the scheduler. With those bubbles we are able to apply some constraint management in the parallel execution of the second phase.

The second (or third) reason of using qr_mumps is because its not a toy application. qr_mumps is:

- 30k lines of fortran code
- single/double precision real/complex arithmetic
- multiple ordering tools: SCOTCH, METIS and COLAMD
- handles large matrix

4.4.2 Contribution

In order to use the constraints algorithms (written in C) in qr_mumps (written in F90) we had to implement all the interfaces to make the conversion from

one to another. qr_mumps was using the process described in the algorithm 4.1 so we must have added a compilation flag to enable or not the constraints. This flag, when activated, disable the previous system to allow the application to submit all the bubbles continuously and enable the constraints specification creation and specification for StarPU. The bubbles being an in-develop feature the branch of qr_mumps which uses the bubbles was very experimental and so adding the constraints bring a lot of quirks. The majority of the work here was to implement a stable system to make some performances tests on very different matrices (in shape and size).

4.5 Validation

To validate the implementation of the constraints and show the contribution in qr_mumps, we save all the tests with formatted names in order to reproduce them easily or retrieve the results needed. These files allow a fast comparison between all sort of configuration different by:

- The features: no bubbles, only bubbles, with constraints
- The excess of memory
- The number of cores
- The matrix used

We, therefore, have a big results database (in the order of 1000 tests) to compare with.

Chapter 5

Implementation elements

5.1 Modular scheduler

We will now detail the implementation elements of our banker scheduler which we proposed in 4.2. First, we will describe the key components of this scheduler and how they are connected. Then we will focus on the banker component and its base functions. Finally, the accumulation pool will be explained to show that the PO (see 3.3) is respected.

5.1.1 Scheduler structure

This is the structure of the modular scheduler we have implemented, as you can it is composed of two sides. The right one is the task path essentially composed of a FIFO storage and a distributor at the end. The left one, the path of the bubbles, possess the banker component which applies the differents algorithms to compute constraints and a distributor to send bubbles to the workers. The first component, bubble sort, is capable of differentiating a simple task from a bubble and send this last on the right path.

5.1.2 Structure functions of banker component

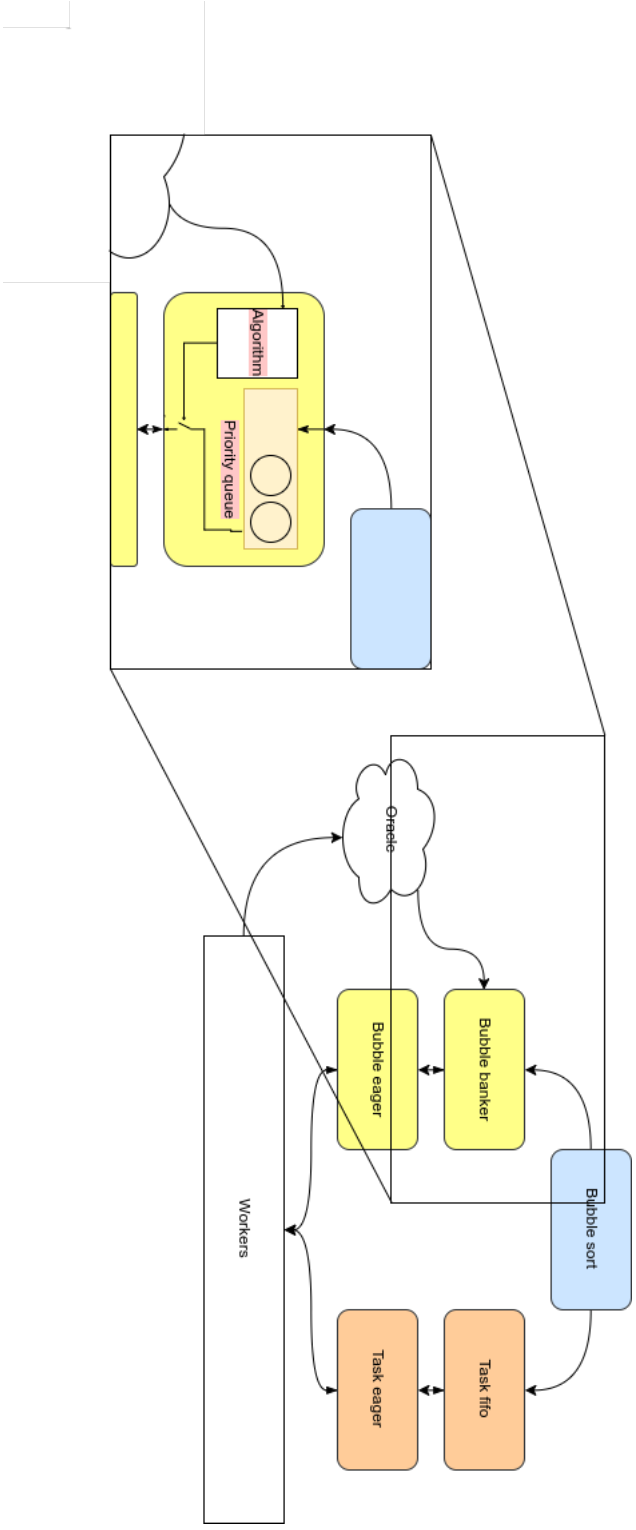
As we see in 2.3 the component needs several functions in order to functions. We implemented only the functions push and can_push (as well as notify but we will see this one later) so we can stock the incoming bubbles and when the workers need something to do they call can_push.

Algorithm 5.1 Push function of the banker component

Input: task T, priority queue PQ
1: $ID \leftarrow T.id$;
2: Lock mutex for concurrent access
3: Insert in PQ the task T with priority ID;
4: Unlock mutex for concurrent access

The banker_filter function looks the specified constraints and computes if the bubble can be executed safely or not. If there is no constraint all the bubbles

Figure 5.1: Zoom on banker component



are pushed when they arrived. In the other case, the algorithm can return two options: push the bubble or not. The algorithms themselves keep track of the required variables to compute this result.

Algorithm 5.2 Can_push function of the banker component

Input: priority queue PQ
1: Lock mutex for concurrent access
2: **if** empty(PQ) **then**
3: return;
4: **end if**
5: Task $T \leftarrow pop(PQ)$;
6: Result $R \leftarrow banker\_filter(T)$;
7: **if** R = no constraint defined **then**
8: Push task T to children
9: **else if** R = push bubble **then**
10: Push task T to children
11: **else**
12: $ID \leftarrow T.id$;
13: Insert in PQ the task T with priority ID;
14: **end if**
15: Unlock mutex for concurrent access

5.1.3 Accumulation pool storage structure

The accumulation pool storage is implemented as a fast efficient priority queue. The priorities are the IDs of the bubbles to not break the sequential order hypothesis in 3.1. We choose to implement a Fibonacci heap because of its efficient removal and insertion of its smallest element (which are called fairly often). The Fibonacci heap offers these complexities in the worst case:

Insert $\mathcal{O}(1)$

Find-min $\mathcal{O}(1)$

Delete-min $\mathcal{O}(n)$

Decrease-key $\mathcal{O}(1)$

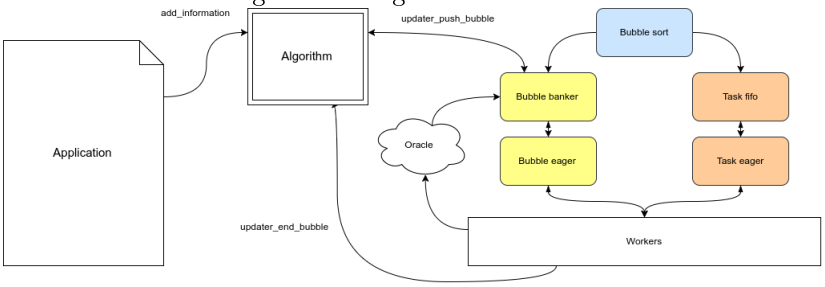
Merge $\mathcal{O}(1)$

5.2 Oracle implementation

The oracle is designed after the observer design pattern. Each component which wants to receive messages from others subscribe to it and a function allows to send an event with a parameter to a specified oracle, this last redirect this event to all subscribed components. The structure of the scheduler become:

As you can see only the banker component subscribe to the oracle. When a worker ends a task it sends an event to the banker saying which task is ended and if it was a task or a bubble. In order to know when a bubble is really

Figure 5.2: Algorithm structure



finished all the tasks created in it must be finished and the bubble itself too. Only after this point, we say to the algorithm a bubble is finished and we apply the corresponding function.

5.3 Constraints

We propose here a way to implement an algorithm structure (figure 5.2) which can represent all algorithms with task consumption (like memory). The structure is in two parts; First, we must specify the constraint type, an algorithm to apply and the number of information we will add later. Then, the algorithm is defined by several functions:

initializer The initialization function of the algorithm

destroyer The destroy function of the algorithm

updater_push_bubble What do we do when a bubble try to be pushed

updater_end_bubble What do we do when a bubble ended

add_information How we handle an information given by the app

The app's user just has to specify a constraint type, the algorithm to use, the number of information and then fill them.

The main problem was the need of high performance, the algorithms functions being called very often they must be fast and we can't afford to lose time accessing variables and dereferencing pointers. As we must provide function pointers for the algorithm functions to allow the app's user to write its own we created a structure which holds all the pointers. The constraint structure is composed of a copy of this structure, a mutex for the concurrent accesses and some variables. These variables are the number of information, the type of the constraint (memory or others...) and the limit of the constraint. With this structure, we only have to access the main structure and dereference the function pointer to call an algorithm function like push_bubble. It's the minimum time wasted to do this while allowing the user to write its own constraint algorithm.

5.4 Tests

To be able to test easily some matrix we developed a script which allows running tests on specific matrix with different memory size and on a different number of cores while keeping the results of all the runs. We then have another script to parse those logs and print readable statistics about the number of bubbles in the scheduler at one time, the number of tasks, the time taken to end or even the parallelism added by the differents algorithms. With all those tests we have been able to generate some interesting results shown in the next chapter.

Chapter 6

Validation and tests

6.1 Introduction

In this chapter we will present the tests that ensure the validation of the experimental structures and some performances results. First we will show a test with a deep memory peak to show that the behavior of the algorithms are correct and then we will present the results on an actually used matrix in qr_mumps.

6.2 Validation

For this validation, we used the tree in figure 3.4. We integrated a sleep of one second in the computations to see where the bubbles were blocked. As you can see below in the different tests the behavior is normal and as we predicted for each memory peak. You can see in the figure 6.6 a real execution of this little program.

In the figure 6.1, the memory is very limited and so the constraint block every bubbles possible. When we add more memory available we can see that the bubbles are allowed to be executed much faster (figures 6.2 and 6.3). The third algorithm is efficient enough to let go all the bubbles with a critical memory peak as you can see in the figure 6.4 and even with a lower memory peak! (see figure 6.5)

6.3 qr_mumps performances

The tests in qr_mumps showed several results, first, we don't lose too many performances with the utilization of the constraint management as you can see below in the figures and. Secondly, the second algorithm performances are not that good for now, it's explained with the implementation of the algorithm in C which has no optimization for now. Indeed, the computations of the third algorithm are heavier than the first and the algorithm functions being called thousands of times a second it has a sufficient overhead to be slower than the first. An optimization could be to remember if an event has happened before calling the functions or keeping in a variable the memory amount to free before it can be possible to push the next bubble (without recalling the function !).

Figure 6.1: Basic algorithm test with a memory peak of 23

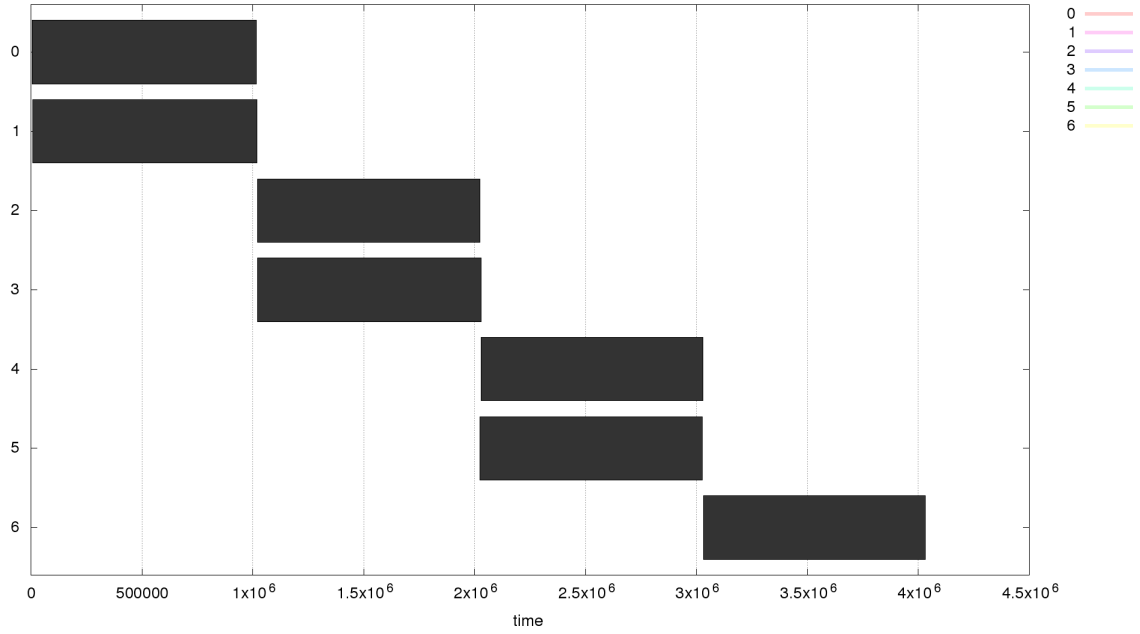


Figure 6.2: Basic algorithm with a peak memory of 24

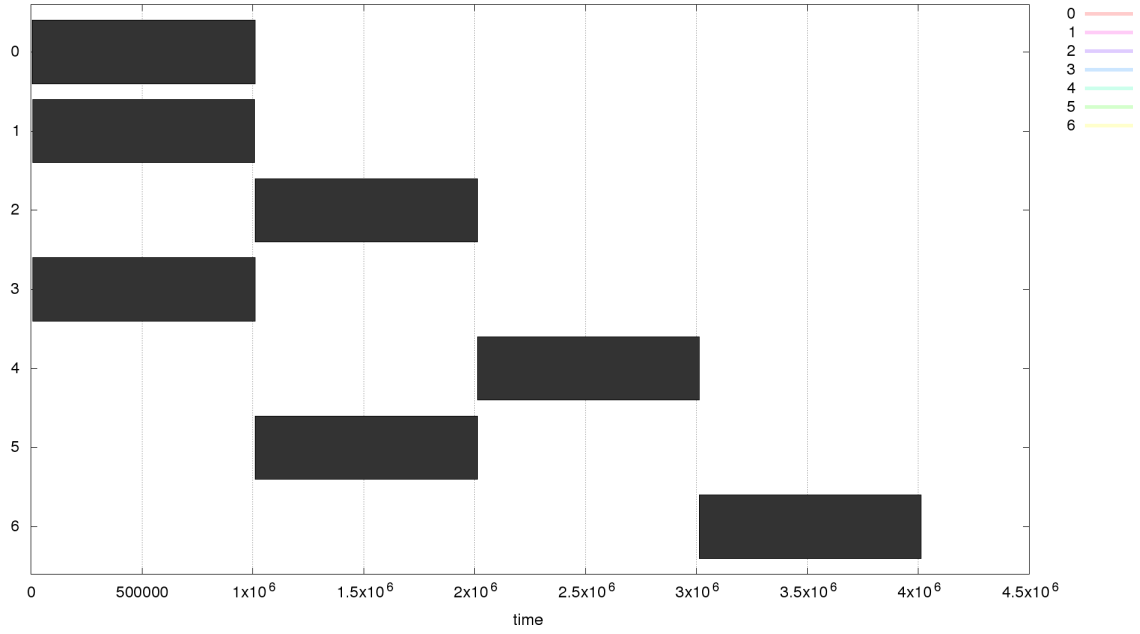


Figure 6.3: Basic algorithm with a peak memory of 40

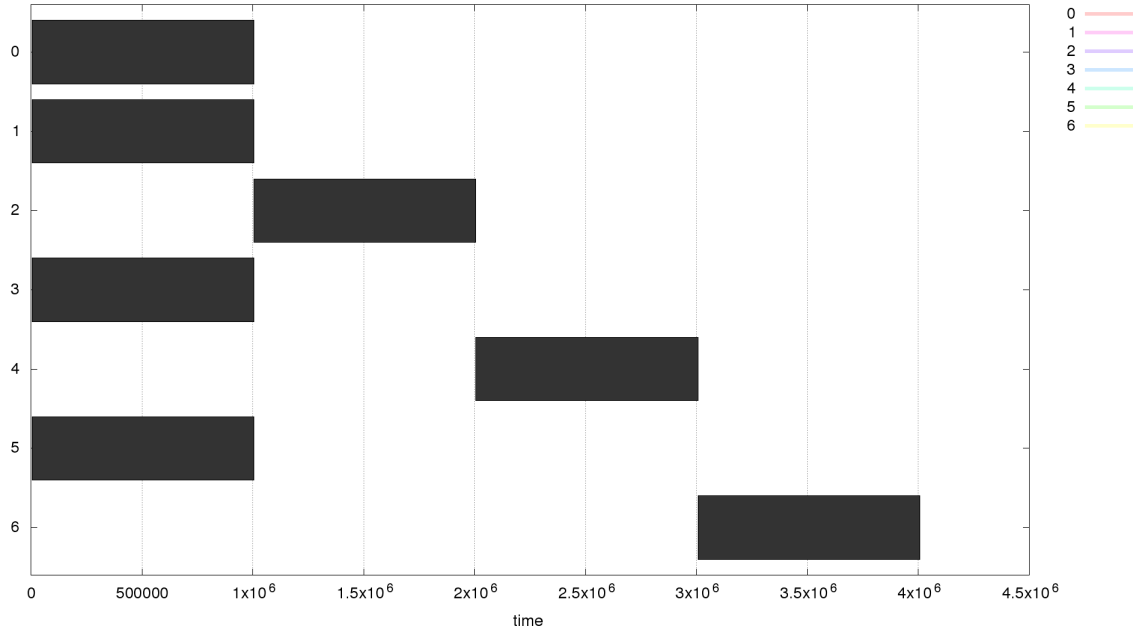


Figure 6.4: Second algorithm with a peak memory of 23

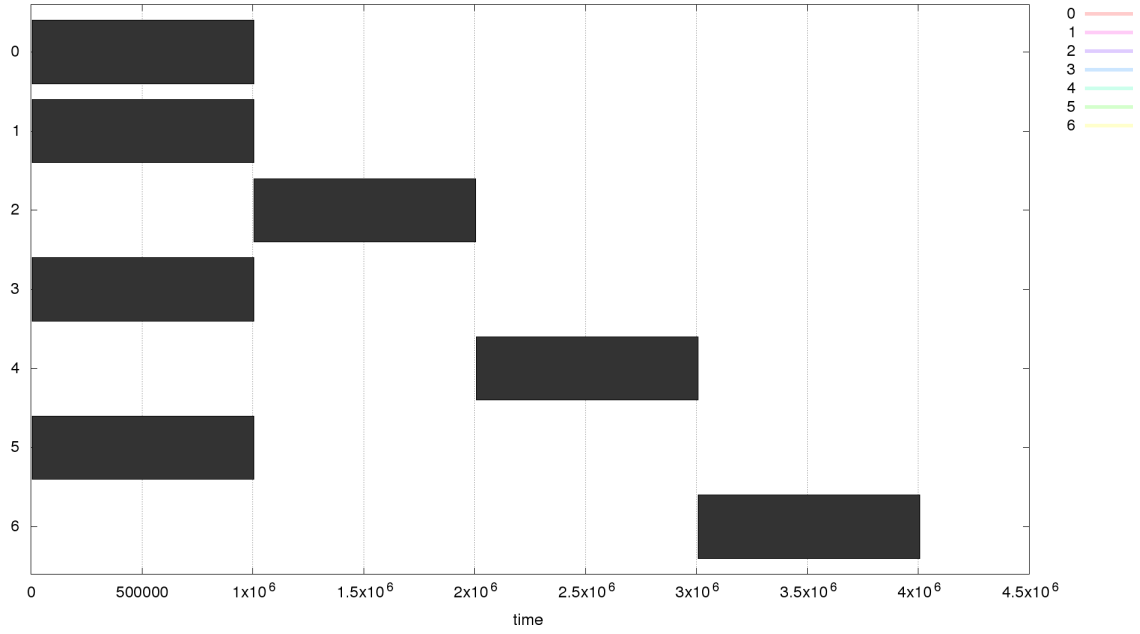


Figure 6.5: Second algorithm with a peak memory of 16

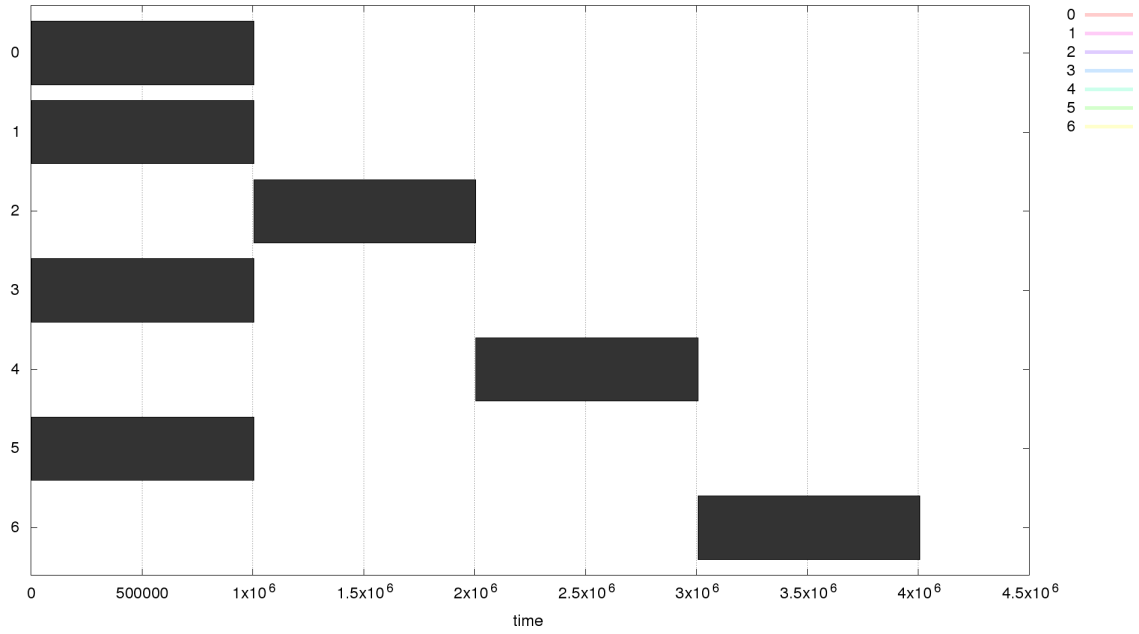


Figure 6.6: Real execution time for the basic algorithm with a memory peak of 23

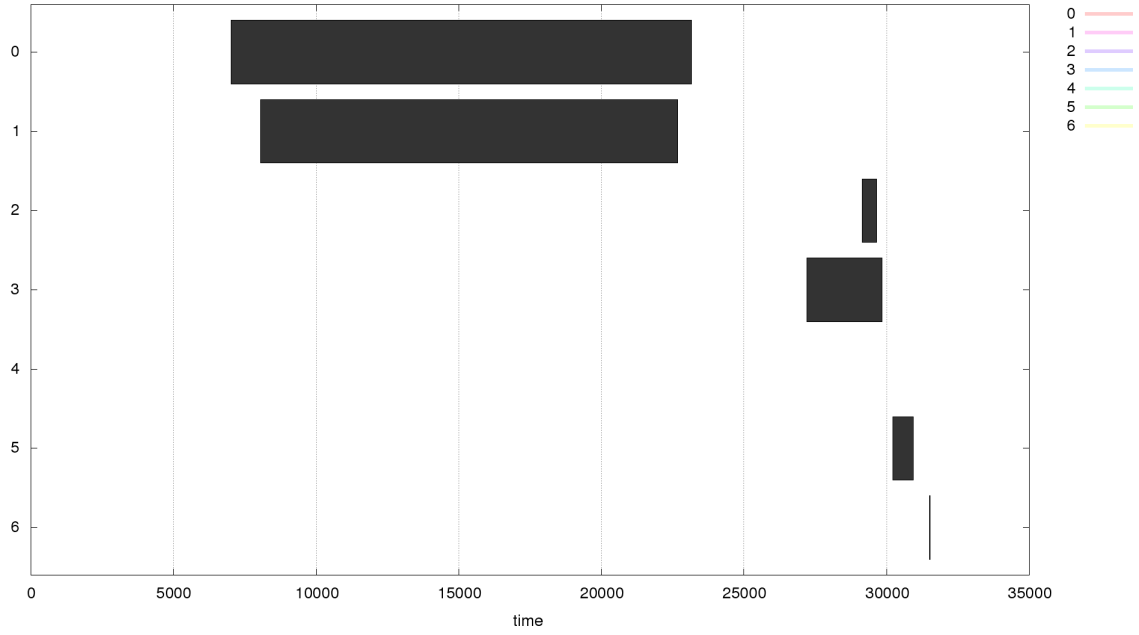
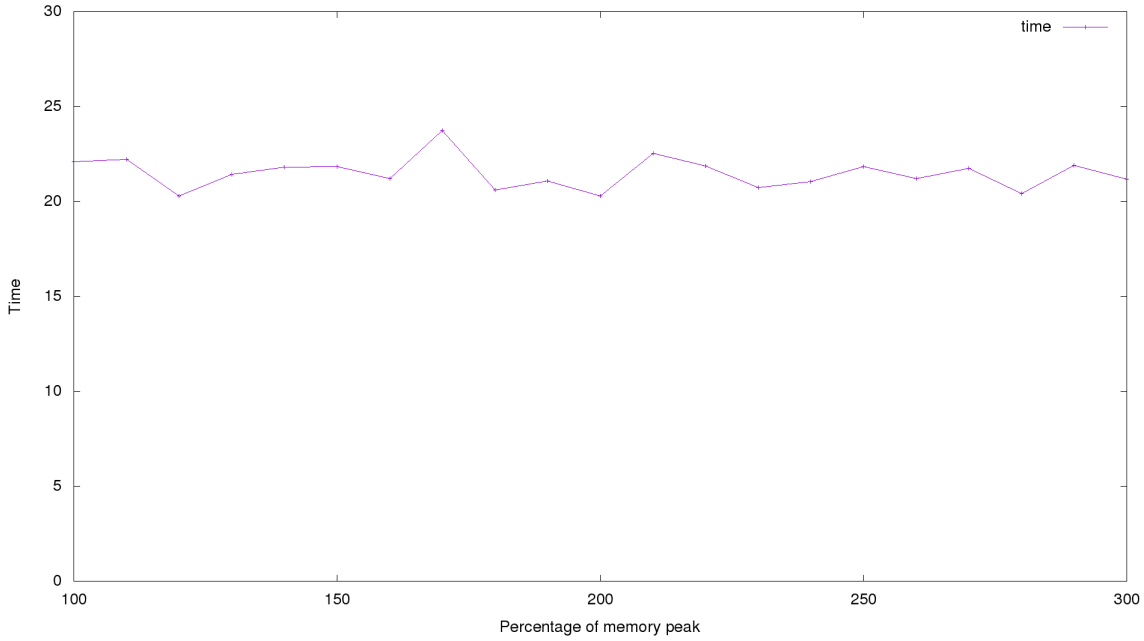


Figure 6.7: Test with qr_mumps without anything



All those figures were generated by running 20 times qr_mumps by memory percentage to have a good mean. We can see in the three figures (6.7, 6.8 and 6.9) that the performances are likely the same even if the second algorithm will be better after optimizations.

Figure 6.8: Test with qr_mumps with the first algorithm

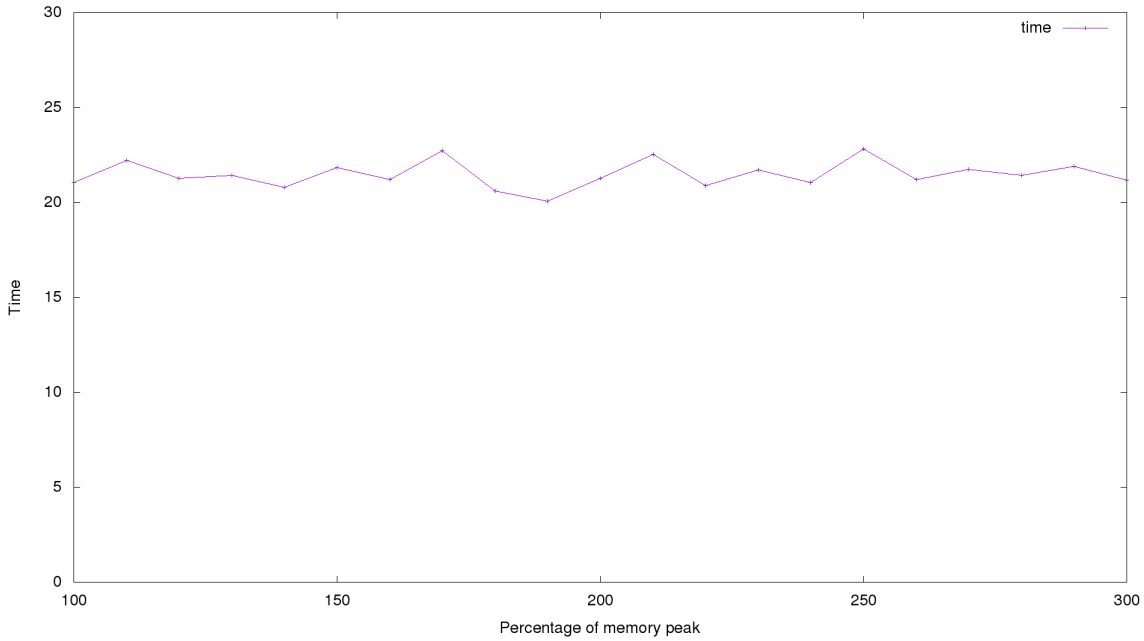
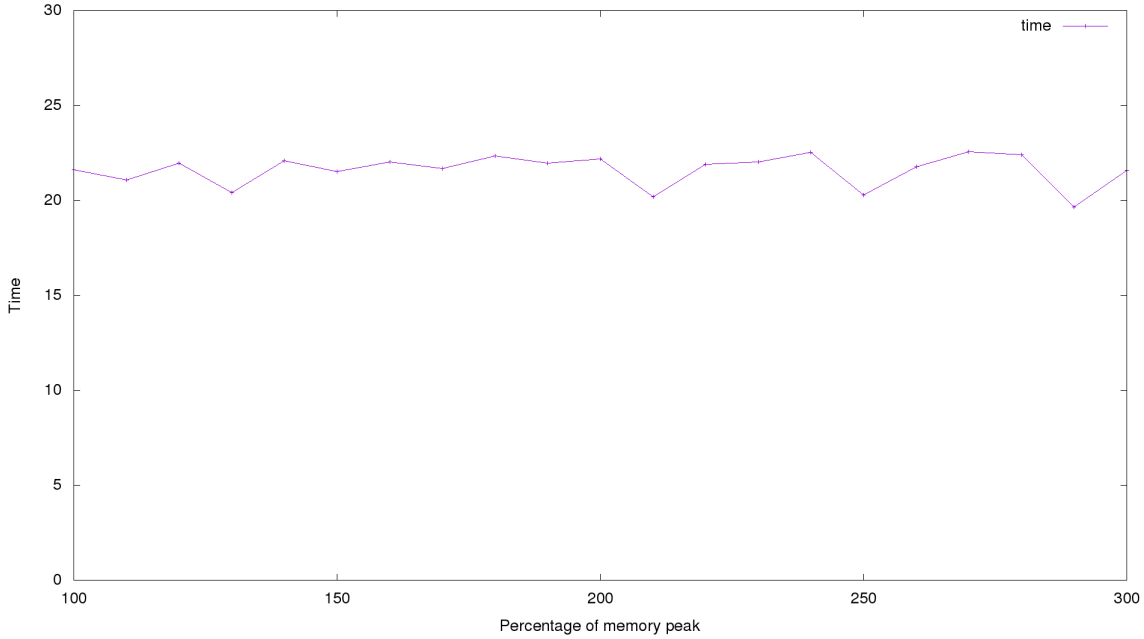


Figure 6.9: Test with qr_mumps with the second algorithm



Chapter 7

Discussions

7.1 Conclusion

Before this internship, the memory constraint management was in the application and it was the user's job to handle it. Some experienced users had very smart algorithms to avoid deadlock when submitting a bubble but others novice users couldn't do this. Now the experience of managing memory constraint is down in StarPU so the experienced users can submit all their bubbles at once without having to handle this constraint (just some StarPU calls), and the more novice users have access to advanced algorithms without having to know how they work. In addition, every user of StarPU can implement their own constraint management algorithm and add it to the runtime system effortlessly (except the effort to think a new performant algorithm). Also, the schedulers users of StarPU have now access to the oracle to implement new techniques of scheduling. And finally qr_mumps now have a way to handle the constraints directly with StarPU and they will not have to handle the memory management anymore, they'll just have to submit all the needed bubbles and improve the differents algorithms to increase performance thanks to the runtime system.

7.2 Short-term perspectives

Constraint management In the future we could improve the constraints management in order to add more constraints handled at one time, currently we only can link one. Improving this point would allow us to check multiple constraints in one run which is, in some case, needed.

Several banker components in a scheduler If we could add different banker component in several flow of the modular scheduler it would allow us to check different constraints on different types of data, like having a memory constraint on bubbles and a number constraints on the tasks to play with the ratio bubbles/tasks in the scheduler.

Apply constraint on distributed architecture The constraint, for now, only apply on one local machine. The addition of distributed architecture sup-

port would allow to compute constraints on a cluster (like an eco-constraint) or to share constraints informations between several contexts.

Apply split constraint locally If we can split the memory constraint and apply it on each core (a little amount of memory is allocated to each core) we could play with it instead of the globally allocated one and thus modify the behavior of the whole algorithm.

Improving algorithm and structures Finally, the improvement of the structures (like the priority queue) of the different algorithms will bring more performance and/or parallelism which is what we want.

7.3 Long-term perspectives

In the long term perspectives, we could create new constraints like the network contention for example. In addition, the same critical resources management could be applied to other application that relies on StarPU as runtime system. Finally, we could imagine a scheduler that does advanced techniques like changing the algorithm used in real time with the progression of the tree to adjust to its shape or to the information it has. Another technique would be to add priorities to activated nodes after being accepted by the banker component and so add several well-known scheduling techniques on top of critical resources management.

Bibliography

[1] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13:1–13:22, 2016.

[2] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime System's Perspective*. PhD thesis, 2011. Thèse de doctorat dirigée par Namyst, Raymond et Thibault, Samuel Informatique Bordeaux 1 2011.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Guillaume Aupy, Clément Brasseur, and Loris Marchal. Dynamic memory-aware task-tree scheduling. Research Report RR-8966, INRIA Grenoble - Rhone-Alpes, October 2016.

[5] Alfredo Buttari. qr_mumps. `http://buttari.perso.enseeiht.fr/qr_mumps/`.

[6] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Parallel scheduling of task trees with limited memory. *ACM Trans. Parallel Comput.*, 2(2):13:1–13:37, June 2015.

[7] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw.*, 12(3):249–264, September 1986.

[8] Joseph W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM Journal on Algebraic Discrete Methods*, 8(3):375–395, 1987.

[9] Marc Sergent. *Passage à l'echelle d'un support d'exécution à base de tâches pour l'algèbre linéaire dense*. PhD thesis, 2016. Thèse de doctorat dirigée par Namyst, Raymond et Goudin, David Informatique Bordeaux 2016.

List of Figures

List of Algorithms