Bachelor's thesis

Compression of the Prediction Models Required for Plan Based Scheduling on HPC Nodes

Mihai Renea

January 13, 2020

Student ID:	5039729
email:	mihai.renea@fu-berlin.de
First Reviewer:	Barry Linnert
Second Reviewer:	Prof. Dr.-Ing. Jochen Schiller
Supervisor:	Barry Linnert

Abstract

An alternative way of scheduling jobs inside a High Performance Computing (HPC) node is by making use of a plan-based scheduler. A plan based scheduler, as opposed to traditional schedulers, which are tailored for responsiveness rather than performance, assigns resources for a job according to a prediction model. Such a model is at it's core a graph-like structure that describes the precedence and distribution across threads of resource requirements. As HPC programs are of extensive nature, their models may scale up in size accordingly. The scope of this thesis is to formalize some aspects of the model and to provide insights into possible methods for reducing its size. A subset of these methods is discussed and implemented as a C library. The effectiveness is evaluated firstly on an abstract level, where the interim results of each method are discussed, and secondly on a practical level, where the qualities of the compressed model are compared to those of the original. The test results show that even simple implementations of the discussed methods can find a considerable amount of patterns in the model and that the compression rates can compete with and even outperform those of traditional file compression.

Statutory Declaration

Berlin (Germany), January 13, 2020

Mihai Renea

Contents

1 Introduction

Most of the present day HPC systems use traditional schedulers to schedule program workload within a node. While convenient, it is hard to predict a program's resource requirements over its lifetime, leading to inefficient allocation of the resources. An alternative approach is the use of a plan-based scheduler, where the execution of the program follows a predefined plan (prediction model). Such a model describes the behavior of a HPC program in terms of precedence and distribution across threads of so called *tasks*. The model has the form of a Directed Acyclic Graph (DAG), where tasks are weighted nodes, representing resource requirements (CPU instructions for calculation or bytes for communication).

1.1 The Program Prediction Model (PPM)

The prediction model's purpose is to provide both spatial and temporal information about a program's behavior and its resource requirements. For a better intuition, we will first take a look at an example model. Afterwards, we will see how it can be simplified and formalized.

1.1.1 PPM as DAG

In abstract terms, at any moment in time, a thread in a HPC program can only do one of three things: execute a so called *task*, fork a child thread or join another thread (either as parent or child). A task can be a *communication task* or *calculation task*. Each task has a resource requirement: CPU instructions for calculation or bytes for communication. A calculation task is a portion of a thread that is bounded by a system call. The system call is done for I/O (thus marking the beginning of a communication task) or for a fork/join. A communication task ends after the system call that evoked it returns. For a more in-depth explanation of these mechanics from the scheduler's perspective, see [Gla19]. In fig 1, we can observe how the task precedence and distribution across threads can be modeled as a DAG.

1.1.2 PPM as tree

The PPM can also be modeled as a tree. The greatest advantage is lower algorithmic complexity when mining for patterns: firstly, trees are easier to handle than DAGs; and secondly, generic DAG mining is NP-complete, while tree mining can be accomplished in polynomial time[WDW$^+$08]. However, in order for this reduction to work, the fork-join model must be restricted. We must think of the forks and joins in the form of Dijkstra's PARBEGIN and PAREND statements [Dij68] rather than the traditional UNIX system calls. They ensure the following:

- multiple forks and joins are either sequential or nested (like pushes and pops on a stack), but not crossed, as illustrated in fig 2. Such a situation can be easily resolved without breaking the task precedence by deferring the problematic fork.
- a child thread can only join its parent.

We now observe two properties of the model that lead to a simpler definition using ternary trees and sets:

- for a fork-join sequence, the join vertex J can be removed, as it is implied by the vertex following J (let's call it H). The fork vertex F can be now linked directly to H. This reduces the graph's structure to a ternary tree.
- a sequence of task vertices is a simply linked list, which can be modeled as a totally ordered set.

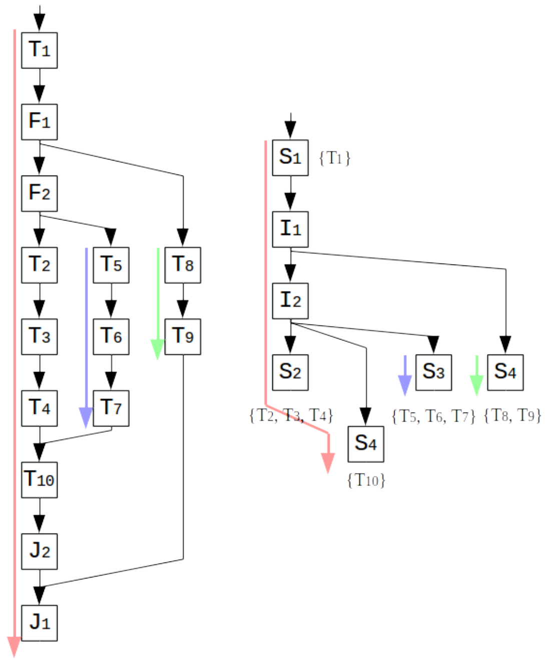


Figure 1: Example of a PPM: as a DAG (left) and as a tree (right). The colored arrows represent the threads.

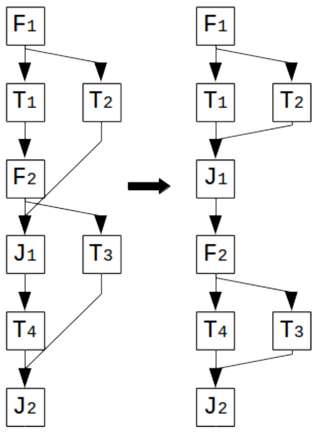


Figure 2: Example of crossed forks (left), and their serialization by fork deferring (right).

Definition 1. A PPM is a ternary tree with three types of vertices:

- Void vertex V.
- Segment vertex S, with $next(S)$ another vertex and $job(S) = \{T_1, T_2, ..., T_n\}$ a totally ordered set, called job. For each task T_i, $type(T_i) \in \{\text{com, calc}\}$ determines the type and $weight(T_i) \in \mathbb{R}^+$ the resource requirements. The ordering relation for the job is determined by the task execution order.
- Inosculation[1] vertex I, with $next(I)$ another vertex and $parent(I)$ and $child(I)$ other non-void vertices.

The *next()* function denotes the precedence relation between vertices, starting from the root of the tree. Inside a segment vertex S, the execution of the job precedes the descend into the $next(S)$ vertex. Inside an inosculation vertex I, the descends into the $parent(I)$ and $child(I)$ vertices precede the descend into the $next(I)$ vertex. Note that there is no ordering relation between the $parent(I)$ and $child(I)$ vertices (parallel branches). The void vertex has no functions defined for it, as its only purpose is termination.

Remark. The reason the tasks are encapsulated into a segment and not represented as vertices in the PPM tree is to break down the compression of the graph in two categories: the structural compression, which deals with the fork-join models in the graph; and the contents layer, which handles the segment compression. Also, since a segment is just a sequential arrangement of tasks, it can be handled as a string, which opens up a wide range of possibilities for compression.

[1]Inosculation is a natural phenomenon, in which branches of a tree merge together.

1.2 PPM comparison

For pattern mining inside the model tree, some ways of comparing parts of the model have to be defined.

Definition 2. For two PPM trees rooted at the vertices G and H, the PPM similarity relation $G \approx H$ is true (i.e. the PPMs are similar), iff all the following conditions are met:

- G and H share the same vertex type and $\mathit{next}(G) \approx \mathit{next}(H)$
- if G and H are inosculations, $\mathit{parent}(G) \approx \mathit{parent}(H)$ and $\mathit{child}(G) \approx \mathit{child}(H)$

Definition 3. For two PPM trees rooted at the vertices G and H, the PPM equivalence relation $G \equiv H$ is true (i.e. the PPMs are equivalent), iff all the following conditions are met:

- G and H share the same vertex type and $\mathit{next}(G) \equiv \mathit{next}(H)$
- if G and H are inosculation vertices, $\mathit{parent}(G) \equiv \mathit{parent}(H)$ and $\mathit{child}(G) \equiv \mathit{child}(H)$
- if G and H are segment vertices, their jobs are equivalent (see 1.3)

Intuitively, two PPM's are similar, iff they structurally overlap, but no requirements have to be met about the segments. Equivalence also requires the segment jobs to be equivalent.

1.3 Job comparison (segment comparison)

Remark. As stated in 1.1.2, segments will be dealt with separately from the rest of the graph. For that reason, segment vertices serve solely as containers for their jobs. To keep the term count low, wherever the scope is clear, the word segment will be used for both the vertex in the tree and the corresponding job.

Definition 4. For a segment S, a requirement list is a set

$$Q_q = \{w \mid \mathit{weight}(T),\ \text{with } T \in \{T \mid T \in \mathit{job}(S) \wedge \mathit{type}(T) = q\}\},$$

with $q \in \{\mathit{calc},\ \mathit{com}\}$.

A requirement list is also a set of all the weights of either communication or calculation tasks of a segment.

Definition 5. Two segments S_a, S_b are similar, iff $|\mathit{job}(S_a)| = |\mathit{job}(S_b)|$ and $\mathbb{S}_{summary}(S_a, S_b)$ is true and for each $T_{a_i} \in \mathit{job}(S_a),\ T_{b_i} \in \mathit{job}(S_b),\ \mathit{type}(T_{a_i}) = \mathit{type}(T_{b_i})$. $\mathbb{S}_{summary}$ is the segment summary comparison function, currently defined as following:

$$\mathbb{S}_{summary}(S_a, S_b) = \begin{cases} 1, \text{ if } \frac{max(\mu(Q_{q_a}),\mu(Q_{q_b}))}{min(\mu(Q_{q_a}),\mu(Q_{q_b}))} < \mu_{max} \text{ and } \frac{max(\sigma(Q_{q_a}),\sigma(Q_{q_b}))}{min(\sigma(Q_{q_a}),\sigma(Q_{q_b}))} < \sigma_{max} \\ 0 \text{ otherwise} \end{cases}$$

with $q \in \{\mathit{calc},\ \mathit{com}\}$ and $Q_{q_a},\ Q_{q_b}$ requirement lists for $S_a,\ S_b$ respectively, μ is the arithmetical mean, σ is the standard deviation and $\mu_{max},\ \sigma_{max}\ > 1$.

Definition 6. Two segments S_a and S_b are equivalent, iff $|\mathit{job}(S_a)| = |\mathit{job}(S_b)|$ and $\forall T_{a_i} \in \mathit{job}(S_a),\ T_{b_i} \in \mathit{job}(S_b),\ \mathit{type}(T_{a_i}) = \mathit{type}(T_{b_i}) \wedge \mathit{weight}(T_{a_i}) = \mathit{weight}(T_{b_i})$.

2 Model preservation

The model's preservation is a subject to the trade-off between compression effectiveness and reliable scheduling. In this section, metrics for both structural and content preservation are discussed and defined. However, finding thresholds for what may or may not be considered reliable is outside the scope of this thesis.

2.1 Content preservation

The content compression focuses on segment compression. When looking for patterns inside of or between segments, it becomes clear that task comparison cannot follow directly because of variations in their weights, so a simplification is needed:

1. Set all the task weights to a fixed value, possibly a mean value across a part of or the whole model. This would reduce the problem complexity by only having to deal with two types of objects. However, from the scheduler perspective, there is not much relevant information left after reconstruction: while the total resource costs are known, no information about the execution/communication requirements at a specific moment is available, which defeats the purpose of plan based scheduling.

2. For a requirement list Q_q and an alphabet Σ with $|\Sigma| \leq |Q_q|$, define a bucketing function $\mathcal{B} : Q_q \mapsto \Sigma$. The comparison function would then run on a string of alphabet Σ. The classifying function then has to be tuned to a compromise between reduced alphabet size and maximal information loss that would still lead to reliable scheduling.

Choosing the task classifying function There are multiples ways of defining the classifying function that need further inspection:

1. Naive: fixed values for alphabet size and weight bounds for each letter. $\mathcal{B}$ would then simply map to the letter in whose bounds a task's weight falls.

2. Fixed: fixed alphabet size, but $\mathcal{B}$ is at least aware of minimum, maximum and expected task weights.

3. Dynamic: Dynamically compute alphabet size and bounds. Might be slow and more intricate, but reliable. To improve speed, it could be run on a sample set of tasks weights.

Defining the task classifying function For the scope of this thesis, a simple dynamic bucketing was implemented. The number of buckets is the size of the alphabet Σ.

Definition 7. A bucket is a set $B \subseteq Q_q$, where

$$\forall w \in Q_q \backslash B,\ w < min(B) \vee w > max(B)$$

A bucketing $\Sigma = \{B_1, B_2, ..., B_n\}$ is a totally ordered set of buckets, where

$$\bigcup_{B_i \in \Sigma} B_i = Q_q \text{ and } \forall B_i,\ B_{i+1} \in \Sigma,\ max(B_i) < min(B_{i+1})$$

For a bucketing Σ, its dictionary is a data structure with following operations:

$key(W) = k$, with $min(B_1) \leq W \leq max(B_n)$ and $\exists B_k \in \Sigma$ so that $min(B_k) \leq W \leq max(B_k)$
$value(k) = W$, with $1 \leq k \leq |\Sigma|$ and $W = \mu(B_k)$

where μ is the arithmetical mean of the weights in a bucket.

The dictionary is used to assign values to or retrieve values from a bucketed segment.

Definition 8. For a set of weights W, its badness function $bad : W \mapsto \mathbb{R}$, with $bad(W) = [0...1]$ is defined as:

$$bad(W) = \frac{\sigma(W)}{\mu(W)},$$

with σ is the standard deviation and μ the arithmetical mean.

The badness function is expressed relatively to the mean of the set because the larger the values in the set are, the more they should naturally drift from each other. Setting an absolute reference value would lead to an unfair distribution of buckets towards the larger values.

Definition 9. For a badness threshold $k \in \mathbb{R}, k = (0, 1]$ and a set of weights W, the bucketing function $\mathcal{B} : W \mapsto \Sigma$ is then defined as:

$$\mathcal{B}(W) = \begin{cases} \{W\}, & \text{if } bad(W) \leq k \\ \{\mathcal{B}(W_1) \cup \mathcal{B}(W_2)\} & \text{otherwise} \end{cases}$$

$$\text{with } W_1 = \{w \mid w \in W \wedge w < \mu(W)\},$$
$$W_2 = \{w \mid w \in W \wedge w \geq \mu(W)\}$$

Alternatively, σ can be replaced with another statistical function that evaluates how spread out the values are. Researching the fitness of such functions is, however, outside the scope if this thesis.

Tweaking the bucketing function

Definition 10. The badness of a bucketing Σ is defined as the average of the badness of its buckets. Formally:

$$bad(\Sigma) = \frac{1}{|\Sigma|} \sum_{B_i \in \Sigma} bad(B_i)$$

Tuning the classifying function is a matter of adjusting its badness threshold k. The tuning criteria are the size of the bucketing and its badness. A large bucketing will deliver more accurate prediction values for the scheduler, but could hinder the pattern recognition required for compression. A small one on the other hand will be easier to compress (less symbols) but could be of little use for the scheduler. The tuning could be static - fixed before program run (possibly based on some meta-information about the model) - or dynamic, changing its value based on feedback. Settling for a completely different bucketing function is also possible. For the scope of this thesis, the threshold will remain fixed. Nevertheless, the tweaking could be an interesting topic for future research.

2.2 Structural preservation

A part of the compression relies on the ability to compress the model tree (the structure). The tree represents the precedence of the tasks and their distribution across threads. A scheduler may be able to tolerate slight changes in resource requirements for individual tasks because of the compensating effect, as discussed in 5.1.1. However, misplacement of entire segments would be hard to handle, at least for the design of the scheduler (see [Gla19]) the models are thought for. Therefore it is crucial that the structural integrity of the model remains untouched.

3 Model compression

The prediction model compression is approached on two levels: graph compression, or *structural compression*, where subgraphs are compared for similarity; and segment compression, or *content compression*, where inter-segment and intra-segment patterns are searched for. It is performed in multiple steps, in the following order:

1. **Aimed discovery**: similar sub-tree discovery in the scope of segment compression.
2. **Inter-segment compression**.
3. **Intra-segment compression**.
4. **Bulk discovery**: similar sub-tree discovery in the scope of graph compression.

3.1 Structural compression

The discovery of similar subgraphs is important for two reasons:

1. Aimed discovery: Similar sub-trees provide good candidates for equivalent segments, thus aiding in segment compression, as it will be discussed in 3.3. This process focuses on the quality of the graphs rather than on quantity, that is, they provide good candidates for segment compression.
2. Bulk discovery: This process focuses entirely on finding as much similarities as possible. For similar graphs, their structure can be stored only once, with their contents (segments) stored separately. For the scope of this thesis, this process was not implemented, as the test results have shown that the aimed discovery already delivers decent tree compression.

Remark. The reason for having two type of sub-tree discovery is that, unless they are at specific places in the graph, any two similar graphs do not necessarily provide good segment compression candidates, thus bloating the pool of candidates.

3.1.1 Aimed discovery

PPM graphs have a relatively simple structure, with spots of high probability for recurring patterns. Determining where these spots are is a matter of exploiting known behavior of HPC programs. For example, after a fork, there is expected that (at least a part of) the concurrent threads behave similarly (worker threads). Alternatively, the compression algorithm could be fed with a meta-model, providing hints about such hot spots, but researching this topic is out of the scope of this thesis.

The search is done in multiple iterations, with each iteration searching for different classes of patterns. The choice of patterns is what seems to cover generic cases of HPC programs and is only meant to serve as a proof of concept. In the future, the pattern choice can be further improved and tailored for different classes of HPC programs.

1. Symmetric inosculations: in a symmetric inosculation, the parent and child branches are similar PPMs. This is typical behavior for worker threads doing similar work.
2. Recursively-symmetric inosculations: one limitation of the symmetric inosculations is that in the case of nested inosculations, they can only find similarities within inosculations that contain a number of threads that is a power of two. This approach tries to recursively find the branch with a lower depth in the other branch. A number of worker threads that is not a power of two reflects this behavior.
3. Repetition: After an inosculation joins, a following one that has the same structure could also be a hint for repetitive work.

3.2 Compressed Prediction Model

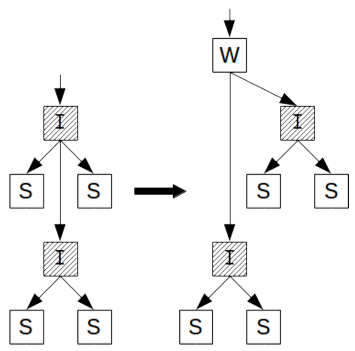


Figure 3: An example of a PPM, before and after the insertion of a wrapper.

For the compression of the prediction model, a new type of vertex is added to the model, namely the wrapper vertex. A wrapper is just a container for a PPM. Because the PPM similarity is defined recursively until the leaves of the tree, it is not possible to compare parts of the graph that do not terminate.

Definition 11. A wrapper vertex W is an addition to the three vertex types defined in 1.1.2, with $\mathit{wrapped}(W)$ and $\mathit{next}(W)$ other non-void vertices. The descend into the $\mathit{wrapped}(W)$ vertex precedes the descend into the $\mathit{next}(W)$ vertex.

Intuitively, a wrapper vertex is like a segment vertex, with the only difference that it encapsulates a PPM tree instead of a job. The usefulness of the wrapper vertex is illustrated in fig. 3. The trees rooted at the hatched vertices are by definition 2 not similar, since one includes the other. By inserting the wrapper, the two trees are now similar.

Remark. Because wrappers do not change the structure of the graph semantically, the similarity and equivalence functions were not redefined to include them. Instead, it is implicitly assumed that, when a wrapper W is encountered, it will temporary be removed and $\mathit{wrapped}(W)$ will be concatenated with $\mathit{next}(W)$. This is, in fact, how the implementation manages the wrappers.

After the similarities have been established, the compression is done in three steps:

1. **Job array creation**: So far, each segment vertex holds its job. The first step in the preparation for compression is to store all jobs in an array, by traversing the tree in a predetermined manner. The tree traversal technique of choice is DFS-Pre-Order, as described in the following pseudo-code snippet:

```
traverse(vertex, jobArr, index):
    switch (type(vertex)):
    case VOID:
        return index
    case SEGMENT:
        vertexAction(vertex, jobArr, index)
        index := index + 1
        break
    case INOSCULATION:
        index := traverse(parent(vertex), jobArr, index)
        index := traverse(child(vertex), jobArr, index)
        break
    case WRAPPER:
        index := traverse(wrapped(vertex), jobArr, index)

    index := traverse(next(vertex), jobArr, index)
    return index
```

2. **Segment job removal**: The jobs from each segment vertex can now be safely removed. Removing the jobs from the segment vertices renders similar subtrees, by definition, equivalent.

3. **Subtree merging:** In the last step, the subtrees that form an equivalence class are reduced to only one representative.

The decompression is achieved in a series of steps analogue to those above, in reversed order.

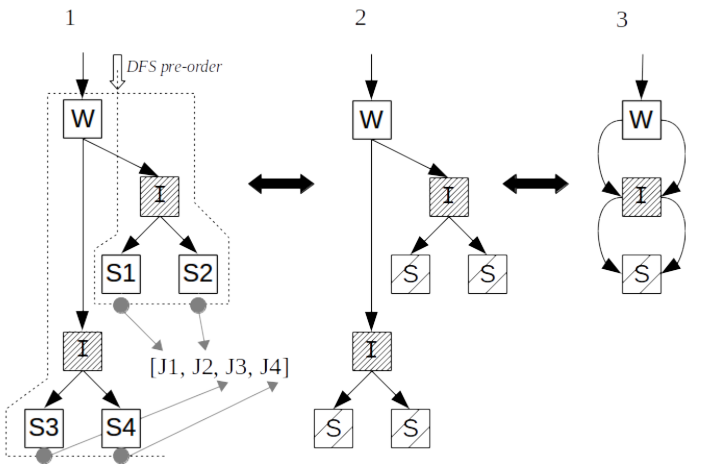


Figure 4: The tree model after each step in the compression. Note that the process is reversible.

3.3 Segment compression

Inter-segment compression This process searches for equivalent segments. As explained in 2.1, the tasks in the segments have to be bucketized before comparing them for equivalence.

There are two possibilities for bucketing:

- A bucketing is created for each segment separately. The advantage of doing so is that the bucketing dictionaries will be better tailored, thus improving the accuracy. The drawback is, however, that there is a high chance of inducing false-negatives: even slight differences in weights have the chance of creating different dictionaries, which will implicitly render the segments not equivalent. Also, for models with short segments, this could lead to bloating, since each segment has it's own dictionary.
- A bucketing is created for a pool of candidates for equivalent segments. Instead of bucketing each segment separately, the requirement lists of the candidates are concatenated and a bucketing is created for it. While being less accurate, it also decreases the chance of false negatives, because all the segments in the group use the same dictionary. Since the scope of this process is to find equivalent segments, this is method of choice. It also has the added benefit that a dictionary is stored for each pool instead for each segment separately.

First step in the segment compression is therefore to perform the aimed discovery, as described in 3.1.1. This delivers segment groups with good odds of being similar. Each group is then split in clusters of similar segments. These clusters are the aforementioned pools of candidates for equivalent segments. A bucketing is then created for each pool.
The bucketized segments in each cluster can now be compared for equivalence. Only a representative of each equivalence class is then needed to be stored.

Remark. As indicated in 3.1.1, the candidates for equivalent segments are usually parallel or sequential worker threads. The ability to extract equivalent segments from them is thus highly dependent on the type of workload. The more the workload differs between workers, the less effective the inter-segment compression will be. However, if there are at least many similar segments, having to store the dictionaries only once is still an improvement.

Intra-segment compression This procedure searches for recurring patterns inside one segment. As the segment at this point is basically a string, regular string compression algorithms are good candidates. However, the segment also has to be long enough in order to avoid bloating. For the scope of this thesis, this step is not performed, since string compression is already a highly researched topic.

4 Implementation

The methods presented so far were implemented as a C library. As the requirements for the implementation are not clear yet, the library is a collection of loosely coupled methods.

4.1 Features

The library offers so far:

- A way to extract the model from the original model file.
- An implementation for the aimed discovery techniques presented in 3.1.1.
- An implementation for the inter-segment compression.
- The possibility to export both the compressed and uncompressed model as binary files.
- Methods for managing the model tree. These are used for the structural compression algorithms, and serve as an important tool for developing further tree mining algorithms.

4.2 Model representation and compression

While tackling different approaches regarding the compression, it was noticed that the data structure containing the compressed graph should provide at least the following:

- as noted in 1.1.2, it should handle *contents*(segments) and *structure*(vertices) in separate ways.
- in the scope of segment compression: can present identified similar segments in a way that is transparent regarding how they were discovered, but also preserves their position in the original model.
- in the scope of graph compression: a way to group identified similar sub-trees.

The PPM implementation follows the theoretical model, with a few additions in order to satisfy these requirements:

- each vertex in the model tree is part of a vertex group. Two vertices G and H may be part of the same group only if the model trees rooted at G and H are similar. References to the groups are maintained in a separate list to facilitate access outside of the model context.
- Each vertex holds a set of values called *vertex summary.* They are meant to provide some information about the tree rooted at that vertex (e.g. hash value, depth, size). Additionally, each vertex provides a container for external algorithms to store intermediate data.

After the aimed discovery (see 3.1.1), extracting the candidates for equivalent segments is done by traversing the vertex group list (the candidates are part of the same segment vertex group). The graph structural compression is achieved by linking all the vertex groups, as if they were normal vertices.

4.3 File format

The compressed model is exported at the end in a binary file format. The data structures are grouped by type, serialized in arrays, and then the pointers replaced with indices. The arrays are afterwards flushed into a file. The file format will not be analyzed any further, as it is only meant to provide a basis for file size comparison.
The original models were provided in a human-readable format. Size comparison between human-readable and binary files is unfair, as the former is obviously less space-efficient. Moreover, the original file format also contains redundant and additional information, which for the scope of this thesis was ignored. For this reason, a binary file for the uncompressed model was also exported. All file size comparisons mentioned in this thesis were done between binary files.

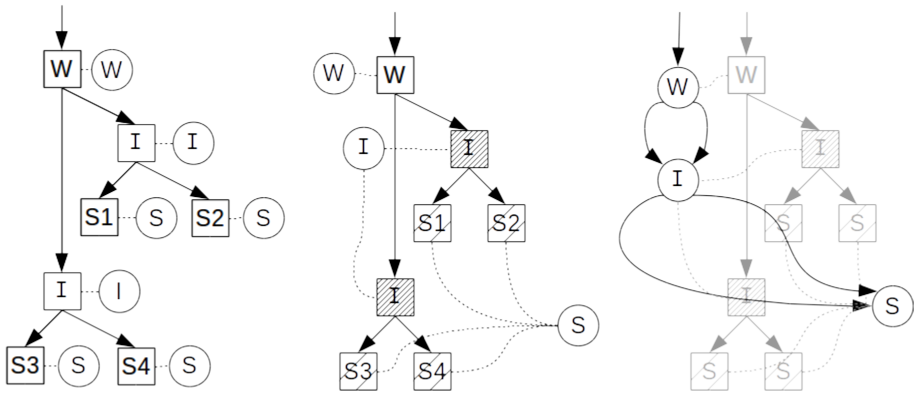


Figure 5: Different stages of the implemented model (the circles represent vertex groups), from left to right: 1. the original model; 2. after similar subgraphs discovery; 3. the linked vertex groups form the compressed model.

4.4 Limitations

The library firstly serves as a an evaluation environment for the presented methods, and secondly as a starting point for future implementations. Therefore, the algorithms and the data structures they build upon were implemented with simplicity in mind, instead of performance. Many routines require polynomial time, although faster but more complicated alternatives exist. Optimizing the library could also be a topic for future work.

4.5 Functional tests

The data structures and basic algorithms were tested with edge cases and a few examples. The more complicated algorithms (e.g. the mining algorithms) were run on smaller models and their results were visualized with gnuplot. To increase the chance of error detection, some data structures contain redundant information and all the algorithms feature multiple redundant assertions.

5 Evaluation

In this section, the effectiveness and limitations of the presented solution are discussed. This is done from two perspectives: firstly, model evaluation, where the different methods used throughout the process and their interim results will be analyzed and evaluated individually; and secondly, the final result evaluation, where the usefulness and compression rate of the end result will be evaluated. The example models that served as a testing basis were given and assumed to represent real world HPC programs.

5.1 Model evaluation

As mentioned in section 3, the model compression is split in two major parts, namely the segment compression (contents) and graph compression (structure). The model evaluation follows therefore the same breakdown.

5.1.1 Segment evaluation

For the segment evaluation, the compression results of two different models are analyzed. Two especially noticeable differences between these models are the average segment length and the total number of segments. As we will see, this leads to different behavior of the tuning parameters. The tuned parameters are the bucketing badness threshold k (see subsection 2.1) and the μ_{max}, σ_{max} parameters of the segment summary similarity function $\$_{summary}(S_a, S_b)$ (see subsection 1.3).

Segment accuracy The accuracy of a bucketed segment is a needed quality for predictable results when scheduling. For Q_{q_r}, Q_{q_b} corresponding requirement lists of a raw respectively a bucketed segment, the segment accuracy has two variables: segment badness $B_\Sigma(Q_{q_r}, Q_{q_b})$ and task badness $B_\Delta(Q_{q_r}, Q_{q_b})$, defined as follows:

$$B_\Sigma(Q_{q_r}, Q_{q_b}) = \frac{\sum_{i=1}^{|Q_r|}(b_i - r_i)}{\sum_{i=1}^{|Q_r|} r_i},$$

$$B_\Delta(Q_{q_r}, Q_{q_b}) = \frac{\mu(\{r \mid r = |b_i - r_i|\})}{\mu(\{r \mid r_i\})},$$

with $r_i \in Q_{q_r}$, $b_i \in Q_{q_b}$ and μ is the arithmetical mean.

B_Σ represents by how much the bucketed segment overestimates/underestimates the actual requirements across the whole segment. B_Δ on the other hand reflects by how much, on average, each task diverges from its original.

Because of the compensating effect, the segment badness proved to be negligible under all tests, having an absolute average value across the whole model between 0,0 and 0,000047 under all tested conditions.

Remark. As stated in section 2, it is not discussed whether a specific metric value is acceptable or not for reliable scheduling, as there is no available data that defines such limits. We will assume the limits $B_\Sigma(Q_{q_r}, Q_{q_b}) < 0,001$ and $B_\Delta(Q_{q_r}, Q_{q_b}) < 0,05$ represent reasonable values for reliable scheduling.

Inter-segment compression rate This is the ratio between the number of total segments in the model, before and after the inter-segment compression. As discussed in 3.3, this is strongly dependent on the type of HPC program.

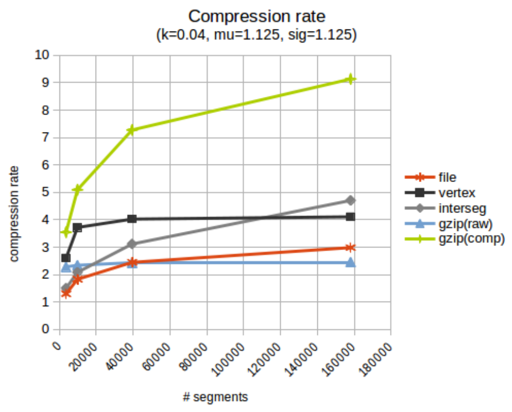
The test results (see fig. A.1, A.2) reflect a somehow expected behavior of the parameter changes. μ_{max} and σ_{max} tend to have only limited effect on the task badness and inter-segment compression rate as they only control the strictness when electing candidates to form a segment compression group. On the other hand, they have a bigger impact on the number of segment compression dictionaries. This could allow for bigger dictionaries to be created, thus improving

the bucketing accuracy. The bucketing badness threshold k proved to have the most influence on task badness and inter-segment compression rate, as it directly controls the coarseness of the segment bucketing.
However, when comparing between the test results of the two models, we can see that the impact of the parameter tuning is not consistent, especially for the μ_{max} and σ_{max}. This robustness issue is a hint that the parameter tuning and/or algorithmic choice should be based on meta-information about the individual model's structure.

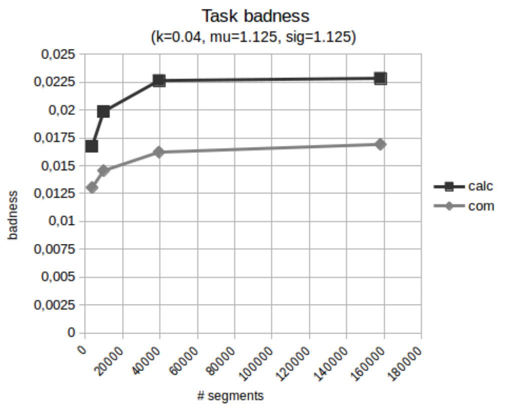
5.1.2 Structure evaluation

For the structural compression evaluation, four models were used. They share the same meta-model (alike in structure), but have different sizes. The parameters μ_{max}, σ_{max} and k were fixed, as they play no role in structural compression.

The structural compression rate is expressed as the ratio between the number of vertices before and after the compression. As shown in fig. 6, the vertex compression rate grows with the number of segments (which is proportional to the number of vertices). It should be noted that this is achieved solely by the aimed discovery, which is tailored for finding inter-segment compression candidates, and not for structural compression. Thus, by performing an additional bulk discovery, higher compression ratios could be achieved. This process should, like the inter-segment compression, be weighted against the type of model: models with higher number of vertices and shorter segments could benefit more from additional bulk discovery algorithms.



gzip(raw) is the ratio between uncompressed file size and with tar/gzip compressed file size. *gzip(comp)* is the ratio between uncompressed file size and the compressed file size with both the presented solution and tar/gzip.



Mean task badness for calculation and communication tasks.

Figure 6: Compression rates and task badness for structurally alike models, with respect to number of segments.

5.2 Final result evaluation

For the final result evaluation, the compression rate of the exported files, the time requirements and, again, the task badness are analyzed.

File compression The file compression rate is expressed as the ratio between the uncompressed and compressed file size. In fig. A.3, we can observe the effect of the parameters μ_{max}, σ_{max} and k on the two different models from 5.1.1 is more uniform. However, the model with longer

segments exhibits significantly higher compression rate (around a third), although it has a much lower segment count (about 50 times). This again reinforces the statement that algorithmic choice should be tailored for different types of models.
The file compression rate also depends on the model size, as seen in fig. 6. It's slope seems to be between that of the vertex and inter-segment compression, suggesting that, at least for this kind of model, they both have an important role.
The compression rate was also compared to that of tar/gzip. The uncompressed file, as well as the compressed file using the presented solution were compressed using tar/gzip with the maximum compression level (`env GZIP=-9 tar cvzf <source_file>.tar.gz <source_file>`). The results show that tar/gzip has an advantage over the presented solution up to a specific model size threshold. Moreover, by additionally running the compression results produced by the presented solution trough tar/gzip, the compression rate is tripled.

Task badness The task badness also tends to increase with the model size. The reason might be the induced noise trough the higher inter-segment compression rate, as more segments are bucketed using the same dictionary.

Execution time The execution time of the program is polynomial. This will not be discussed in depth, as for the purpose of this thesis, the implementation of the algorithms and especially the data structures they rely on are rather on the naive side. Nevertheless, any generic subtree mining algorithm is expected to have polynomial complexity, as stated in 1.1.2.

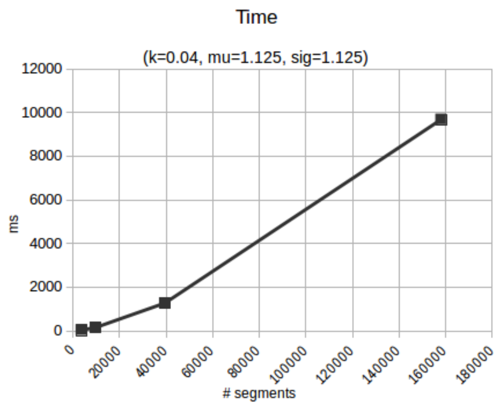


Figure 7: Time requirements for compressing models of similar structure but different sizes on an Intel®Core™i5-4210U CPU (single core).

6 Conclusion and future work

This thesis provides insights into the compression of prediction models for HPC programs that operates at a higher level than plain file compression, and presents a few methods for achieving it. Furthermore, it is shown that even with simple, proof-of-concept algorithms and metrics, decent compression rates with relatively low information loss can be achieved. The test results suggest that having access to information about the model could allow for the development of meta-model aware algorithms, thus achieving more robust results across different types of models. In future work, this information could be provided together with the model or extracted directly from the model itself. What was not achieved is the development of tree mining algorithms in the scope of structural compression (*bulk discovery*), and the compression of the task segments (*intra-segment compression*). The implemented library can nevertheless serve as a practical framework for such features to be added.

Another important topic that was not approached is the probabilistic nature of the models: multiple runs of the same program could produce models varying in both structure (e.g. forks may be added or removed) and weight (tasks may require inconsistent amounts of resources).

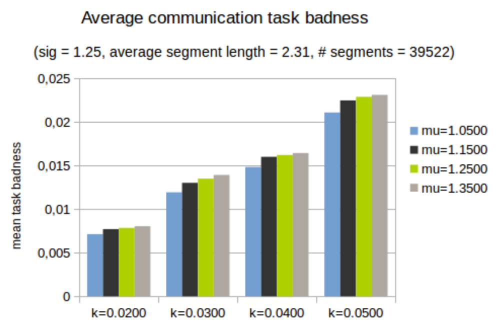
References

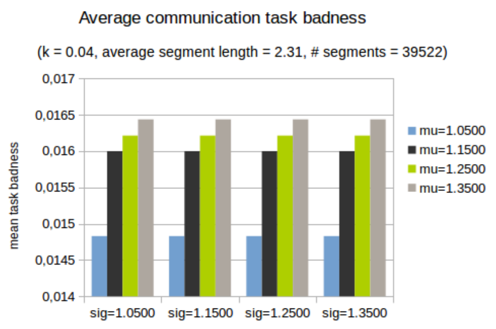
[Dij68] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, Sep 1968.

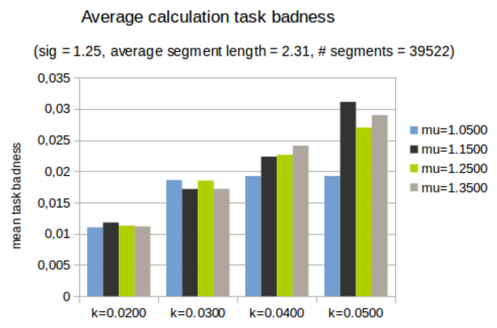
[Gla19] Kelvin Glaß. Plan based thread scheduling on hpc nodes, 2019.

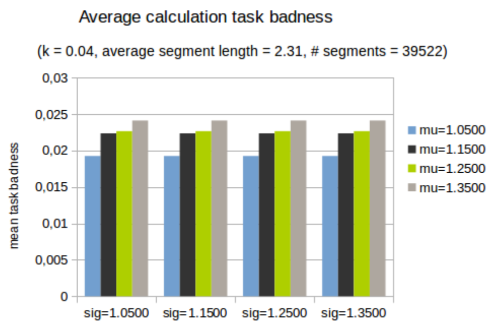
[WDW$^+$08] Tobias Werth, Alexander Dreweke, Marc Wörlein, Ingrid Fischer, and Michael Philippsen. Dagma: Mining directed acyclic graphs. In *IADIS European Conference on Data Mining 2008, Amsterdam, The Netherlands, 24. - 26. July 2008. IADIS Press, 2008*, pages 11–18. IADIS Press, 2008.

A Appendix

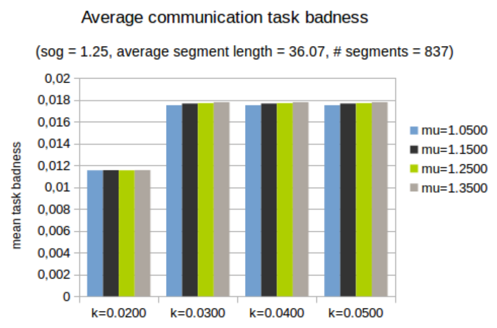


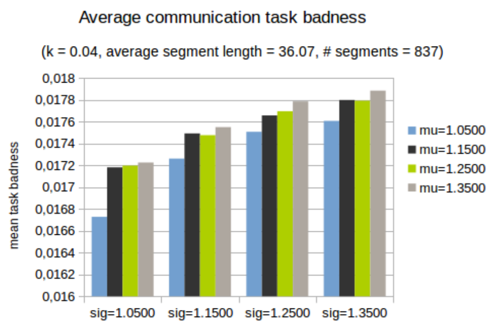


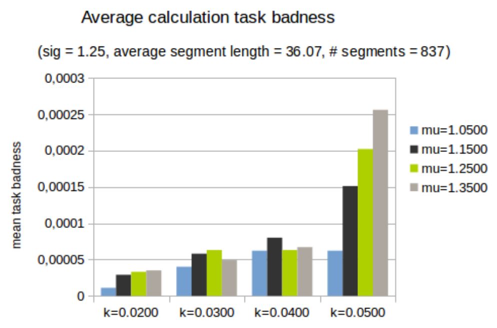


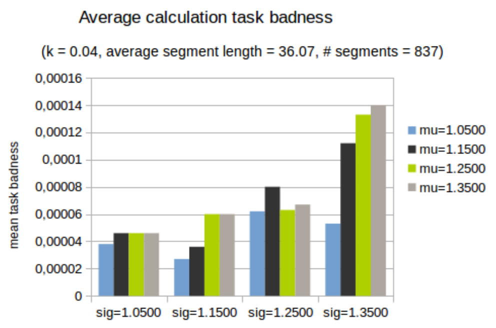


average segment length = 2,31, segment count = 39522









average segment length = 36,07, segment count = 837

Figure A.1: Test results: badness for two different models. Left half: fixed σ_{max}. Right half: fixed k. μ_{max} and σ_{max} have generally lower impact than k, but behavior is not consistent across different models.

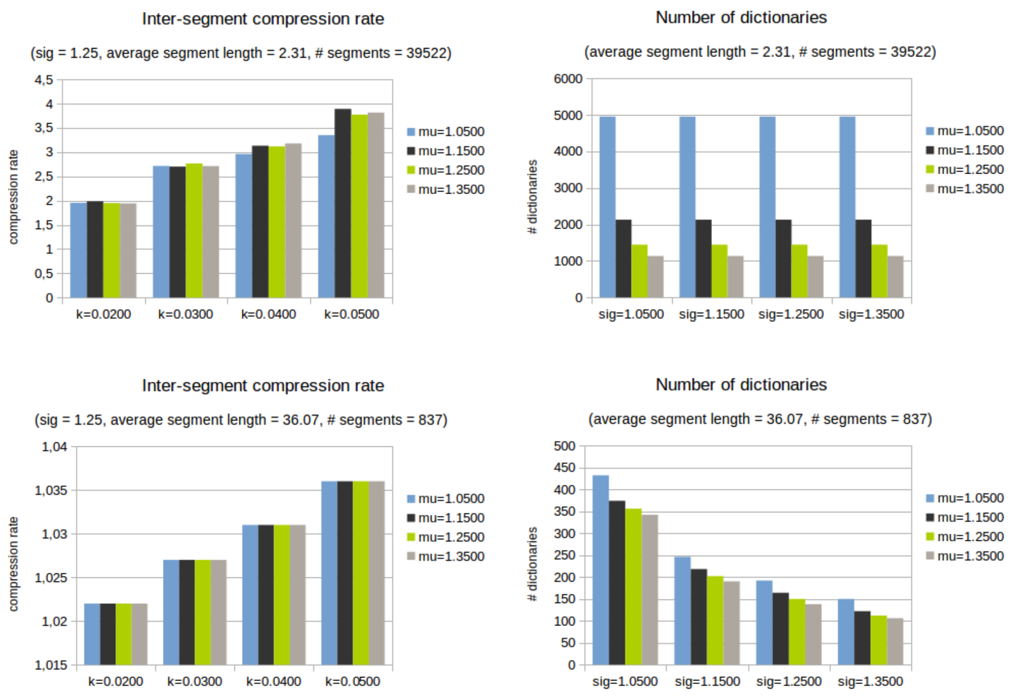


Figure A.2: Test results: number of dictionaries and inter-segment compression rate for different models. k has the greatest impact on inter-segment compression rate. μ_{max} and σ_{max} display inconsistent behavior.

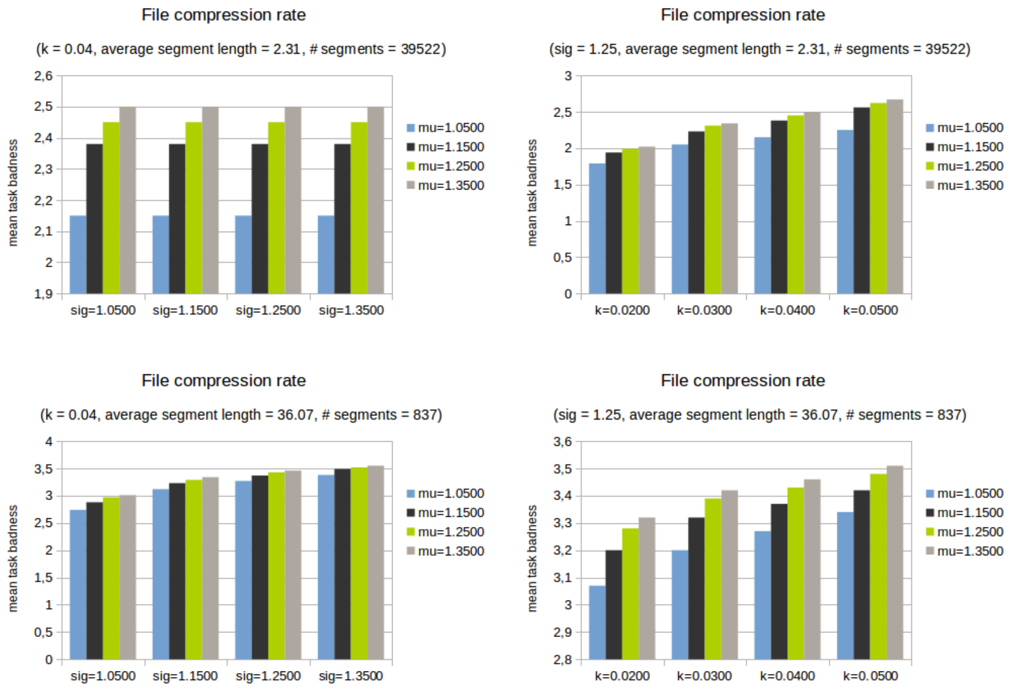


Figure A.3: Test results: file compression rates for two different models. The impact of parameter changes are more consistent, but differ in magnitude across different models.