Using Static Weight Estimation of General Trees to Efficiently Parallelize the Execution of DEMKit Simulation Models

Niklas Lüpkes
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.n.lupkes@student.utwente.nl

ABSTRACT

To aid the development of control mechanisms of smart grids, the "Decentralized Energy Management Simulation and Demonstration Toolkit" (DEMKit) was developed. It may be used to test and evaluate control algorithms by simulating complex (smart) energy grids. This research paper discusses the inherent problems of the models' structure, a general tree, which hinders the setup of multi-processed simulations. An algorithm to automate the division of weighted general trees is presented, to spread workload and exploit parallelism. Weighted Tree Distribution (WTD) reduces a tree to a predefined number of leaves, based on tree contraction and respecting existing structures. Here, makespan minimization is used to optimize the distribution. The algorithm is tested using 1429 randomly generated trees of varying size. The tests confirm the time complexity of $\mathcal{O}(n \log n)$ and yield an average of efficiency of 1.58. Finally, WTD is integrated into DEMKit, where it is subjected to different models and distributed over four processes. The implemented automated model distribution performs best if the distributed structures are independent.

Keywords

General Tree, Parallelism, Smart Grid, Simulation

1. INTRODUCTION

The ongoing energy transition introduces new challenges to manage demand and supply. Supply of energy by renewable sources of energy is not controllable by energy providers as the production is heavily influenced by e.g. weather, tide, etc. On the other hand, consuming devices, such as smart charging electric vehicles and smart thermostats, offer an increasing amount of control over their energy consumption. As such, smart grids integrate the opportunities offered by new technologies to tackle the challenges imposed by renewable and other, less controllable energy sources. The envisioned system enables providers to closely match production and consumption, while also reducing e.g. grid stress and loss of energy. The energy managing structures, however, are still under research and development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
33rd Twente Student Conference on IT July. 3rd, 2020, Enschede, The Netherlands.
Copyright 2018, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

To aid, several tools have been developed, one of which is the "Decentralized Energy Management Simulation and Demonstration Toolkit" (DEMKit) [14]. It is used to simulate, test, and evaluate complex (smart) electricity grids and custom control algorithms, that e.g. optimize energy consumption schedules. The toolkit utilizes a tree hierarchy of abstract devices and controllers that may implement the specific functionality, e.g. of a washing machine or of a power plant. In theory, the scale of the simulation may therefore reach from a single household with multiple devices up until large grids consisting of multiple blocks of houses and more. However, as Hoogsteen et al. [13] have shown, the limit of scalability is computation time, which practically ceils the size of simulations. To improve computation times, the application supports parallel processing. However, the setup of multi-processed simulation models is difficult and needs to be configured manually.

Therefore, the goal of this research is to simplify the usage of parallel computation. To automate the parallel model setup, firstly, an approach needs to be found to estimate the weight of a general tree. It should specifically incorporate the inherent characteristics of DEMKit models. Secondly, the approximate tree weights shall be used to distribute the workload equally among available processes. The utilized technique should maximize speedup and efficiency. To limit the scope of this research the techniques researched are of static nature, meaning no at-runtime optimizations are discussed. Overall, this results in the following research questions:

1. How can the weight of a general tree, as utilized by DEMKit simulation models, be approximated?
2. Using the approximate weights of subtrees within a DEMKit simulation model, how can the respective subtrees and their corresponding computational workloads be (optimally) distributed among an arbitrary number of processes?
3. How does the proposed division and distribution algorithm affect speedup and efficiency of parallel execution of DEMKit simulations?

In the following, the sources of the difficulties regarding the setup of multiprocessed simulations are discussed. Subsequently, an algorithm is designed in accordance with the research goals, evaluated experimentally and integrated into DEMKit. As a basis, existing literature will be reviewed. The performance of the integrated algorithm is finally evaluated using hand-picked representative models, comparing automated to manual distribution.

2. BACKGROUND

The following section summarizes background information about evaluation metrics for parallelized applications. Furthermore, the structure of DEMKit models is introduced and the resulting problems for parallelized setups are analysed.

2.1 Evaluation metrics

As stated, this study aims to develop a setup process that ultimately decreases the computation time of simulations via parallel execution. The success of suggested algorithms must be evaluated using empirical measures. These measures are introduced in the following.

Starting, upon parallel execution let p be the fraction of time spent in the parallel part of the application. Accordingly, let s be the remaining fraction of time spent in the sequential part of it, such that $p + s = 1$. Next, let N denote the number of processes used. Subsequently, Amdahl's law [2] defines speedup as the fraction of execution with one process over execution with N processes:

$$speedup = \frac{s+p}{s+p/N} = \frac{1}{s+p/N} \quad (1)$$

For a given application with certain sequential and parallel fractions, this defines speedup to be a function of the number of processes available, by directly comparing achieved execution times.

However, Gustafson [11] argues that this approach is flawed, as the common goal is not simply to reduce execution times but rather to increase the computable problem size. This applies to DEMKit as well. The objective is not merely to decrease computation times, it is to enable the simulation of bigger, more complex models. Hence, Gustafson argues speedup should be expressed in terms of work computable using parallel execution versus sequential execution. The resulting equation is commonly known as Gustafson's law:

$$speedup = \frac{s+pN}{s+p} = s + pN = N + (1-N)s \quad (2)$$

Finally, efficiency shall then be defined as speedup per process.

So far, the metrics introduced purely cover the ideal circumstances without communication costs or other kinds of overhead. However, depending on the application, the influence of additional costs on the performance may be immense. Further, a change in efficiency may be caused by a variety of different factors, including communication costs, as well as idle time due to workload balancing issues.

To aid analysis, Karp and Flatt [16] suggested the use of the scaled serial fraction f_k, where k is the growth in problem size. In accordance with Gustafson's law, they define it as follows:

$$f_k = \frac{1/s_k - 1/p}{1 - 1/p} \quad (3)$$

The serial fraction $f = kf_k$ has the advantage that within an ideal system, it remains constant. This simplifies problem diagnosis greatly, as irregularities in the behaviour of f are easier to discover. Specifically, a decreasing serial fraction indicates that the problem size is not growing fast enough to utilize the resources by additional processes. On the other hand, an increasing serial fraction indicates that the additional costs, such as communication overhead, are increasing.

2.2 DEMKit model & simulation structure

As mentioned, the usability of multi-processed DEMKit simulations is lessened due to difficulties setting up the parallelized environment. The corresponding details are elaborated upon in the following.

To ensure high efficiency at all assigned processes, the overall workload should be distributed equally among them. In the case of the DEMKit, this means that the to be simulated model needs to be split up such that the resulting parts each equate approximately the same workload. Following, internal communication between processes is organized via the master-slave approach, supplemented by a messaging bus. Currently, division of workload is not defined due to challenges imposed by the structure of models, as presented subsequently.

Within a DEMKit model, all entities are organized via a general tree, as displayed in Figure 1. Generally, communication is only permitted between parent and child nodes. Any other flow of information or instruction, such as inter-sibling communication, is prohibited. Furthermore, control algorithms such as Profile Steering communicate significantly more with their children than their parents. Thus, to distribute the resulting workload onto N processes, the tree needs to be divided into (at least) N subtrees. Preferably, to optimize efficiency, the weights of the individual subtrees should be equal. If equal weight is not achievable, the size of the largest part should be minimized. It is important to note that the weight of a subtree is not equal to the number of nodes within. A multiplicity of different factors influences the processing time required for each node.

- **Unrestricted topology:** As stated, the model is structured via a general tree. Aside from technical limitations, this means there are no restrictions regarding the depth of a tree or the number of children. Hence, the size of subtrees may differ vastly, even if they reside on the same level.
- **Control algorithm:** The control algorithm and its complexity heavily influence the effect a tree's depth has. E.g. in the case of profile steering the behaviour is polynomial [9].
- **Component implementation:** Different components serve different purposes, thus requiring different computation times. Furthermore, the utilized control algorithm may vary per device class/instance. Apart from that, certain components exist that violate the restraint of exclusive vertical communication. E.g., smart meters communicate horizontally as well.
- **Simulated environment:** Components may only activate or adjust their tasks according to different environmental situations, such as time, weather, etc. This does not just influence the components themselves but also the respective subtrees of them.

It follows that computation times vary with the model, the used control algorithm and within each time interval simulated. Due to these uncertainties it is very difficult to predict or even precalculate the weight of a tree, especially if the to be simulated model has an asymmetric topology.

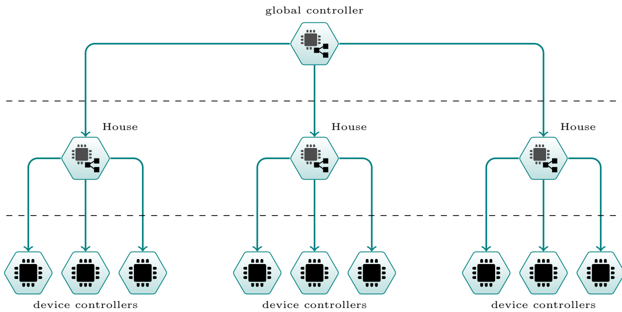


Figure 1: Example of DEMKit Control Hierarchy[12]

Summarizing, the inherent structure of DEMKit models creates difficulties distributing the workload across available processes while maintaining high efficiency. The absence of constraints regarding topology and the freedom in implementation, result in highly uncertain and inconsistent computation times, which impede division of responsibility and thus distribution of workload.

3. LITERATURE REVIEW

3.1 Makespan Minimization

The so-called job shop problem is an optimization problem that deals with the assignment of a set of tasks to a set of workers. The goal is to compute a schedule that minimizes the makespan, the completion time, of the entire task set. The problem is NP-hard [8], therefore current research focuses on polynomial time approximation algorithms. Following, given an optimal schedule s with minimal makespan w, an algorithm producing at worst a schedule s' with makespan w' is called c-competitive where $c = w'/w$. Thus, c defines the upper bound in performance of said algorithm. This problem is formulated and analysed in many different variations. However, the following review shall be limited by the subsequent constraints:

- All n tasks are known beforehand, including their associated computational cost.
- To complete a task, no skillset is required. Further, all machines are equally efficient working on each task.
- Tasks are atomic.
- The number of machines m is constant and known.

The first work in this area has been published by Graham in 1966. It proposed the `List` scheduling algorithm (`LS`), an approach for task sets that are structured using a partial order λ, that can be represented using a directed, acyclic graph [10]. Following, tasks are assigned to the respective machines with the least cumulative workload, respecting λ. This algorithm is $(2 - \frac{1}{m})$-competitive and can be implemented in $\mathcal{O}(n^2)$. It has been proven that $2 - \frac{1}{m}$ is the lower bound for $m = 2$ and $m = 3$, thus the List algorithm is optimal in these cases [5]. A slight variation to this is the `Longest Processing Time (LPT)` algorithm [10]. If the partial order is empty, meaning all tasks are independent, the taskset can be sorted in decreasing order and distributed sequentially. This achieves a competitive ratio of $\frac{4}{3} - \frac{1}{m}$ and can be implemented in $\mathcal{O}(n \log n)$. Graham further notes that his algorithm only considers fixed lists, where a task's weight has no dependence on future tasks. By using dynamically formed lists, i.e. by adding directly dependent task costs, the competitive ratio can be improved slightly to $2 - \frac{2}{n+1}$.

Over the years, several improvements to the upper bound achieved by Graham's `List` algorithm have been published [7][3][15][1]. The latest, most competitive algorithm, named `MR`, is published by Fleisher and Wahl in 2000 [6]. According to them, the worst-cases of `LS` are caused by situations where all machines have approximately the same workload. Hence, a metric for flatness is defined that helps identifying such situations. Tasks are then scheduled as specified by `LS` if the schedule is steep and a specific pivot machine M_i is exceeding the average load by a predefined factor. This yields a <1.9201-competitive algorithm, with unchanged quadratic complexity. Other algorithms with lower bounds exist for special cases, e.g. for $m = 4$ an algorithm is known that is 1.733-competitive [4]. Concluding, Table 1 provides an overview of the relevant presented algorithms.

Table 1: Overview of Makespan-Minimizing Algorithms

Algorithm	Competitive Ratio	Note
LS	$2 - \frac{1}{m}$	Optimal for $m \leq 3$
MR	<1.9201	Best known to date
LPT	$\frac{4}{3} - \frac{1}{m}$	For empty λ

3.2 Tree contraction

In 1989, Miller and Reif [17] published an algorithm to parallelize tree structured workloads, called dynamic tree contraction. Two operations are defined, namely `rake` and `compress`, that form the `contract` operation. The algorithm reduces a tree to its root when applied $\mathcal{O}(\log n)$ times. Let the `rake` operation remove all leaves of a tree by joining them to their respective parents. As Miller and Reif show, applied to an unbalanced tree, this operation could take up to $\mathcal{O}(n)$ steps. The reason for this inefficiency are chains of vertices, i.e. sequences of vertices that have exactly one child: $[v_i, v_{i+1}, ..., v_k]$ where v_{i+1} is the only child of v_i. Thus, to improve complexity, the `compress` operation is introduced, which reduces chains to half their length by joining every other vertex to its parent. The remaining chain is therefore $[v_i, v_{i+2}, ..., v_{k/2}]$. Applied after each `rake` operation, this quickly eliminates worst case structures, reducing the required number of steps to $\mathcal{O}(\log n)$. As both `rake` and `compress` are applied to the tree linearly, the overall complexity is $\mathcal{O}(n \log n)$.

4. WEIGHTED TREE DISTRIBUTION

In the following, an approach to static distribution of hierarchical model and control structures is introduced. To limit scope, the following focuses on static factors, i.e. the topology, the control algorithm behaviour and component implementation. Non-predictable runtime factors, such as the environment, are excluded from the subsequent process.

4.1 Algorithm

The goal of the subsequent distribution algorithm is to divide the hierarchy and assign each element to one processor. It is assumed that the hierarchy is readily available and weighted. Further, designate the number of processors assigned as the `target`. Starting, it would be possible to finely divide and distribute the entire tree using Fleisher and Wahl's `MR` algorithm, including a dependence relationship equalling the tree structure. This approach has two immediate downsides. Firstly, because of the absence of skillset constraints, any node on any level may be assigned to any process. As a result, heavy communication costs are to be expected, especially if controller-to-device connections are broken. Secondly, the MR algorithm does not account for repeated, recursive calls. Hence, huge losses in utilization due to idle and wait times are likely. Thus, any distributing algorithm should keep the current, coarse approach of dividing into and distributing entire subtrees. This minimizes implementation overhead, communication costs and does not interfere with repeated, recursive calls, therefore improving performance.

The to be introduced algorithm, `Weighted Tree Distribution (WTD)`, aims to achieve the formulated goals by employing tree contraction and a makespan minimizing algorithm. However, multiple modifications are added. Most importantly, instead of reducing the tree to its root, it is reduced until it contains the targeted number of leaves:

Function distribute(tree, target)

```
1 done ← numberOfLeaves(tree) ≤ target
2 while !done do
3 |   done ← rake(tree, target) ≤ target
4 |   compress(tree)
```

Following, the `rake` operation does not reduce all leaves, but at most half. In preparation, all leaves are gathered, ordered by weight, and the $\max(target, \lfloor len(leaves)/2 \rfloor)$ largest leaves are marked as persistent, meaning they are not to be removed during the following `rake` reduction:

Function rake(tree, target)

```
  // select the heaviest 50%
1 leaves ← leaves(tree)
2 target ← max(target, ⌊len(leaves)/2⌋)
3 sort(leaves)
4 for 0 ≤ i < target do
5 |   persistent(leaves[i]) ← True
  // apply selection
6 return rakeTree(tree)
```

Here, `rakeTree` is recursively removing leaves from the tree by means of a makespan minimization algorithm. In the current case, `LPT` is used, but should the need arise more advanced algorithms may be integrated. Starting, the number of remaining children is determined (Line 5-8). Note that (non-persistent) leaves may be merged into subtrees and other leaves, but subtrees remain unchanged, as well as persistent leaves. Apart from that, to improve granularity, at most half of the leaves are merged away (Line 9-10). Next, the to be merged items are gathered and assigned (Line 11-13). To improve performance, all merged items from previous iterations should be distributed again. Once the operation on the current vertex is complete, it can be forwarded to the remaining children. Finally, because this is the only operation that changes the number of leaves contained, it is helpful to immediately gather the remaining number of leaves:

Function rakeTree(tree)

```
   // exit condition
 1 if isLeaf(tree) then
 2 |   return 1
   // makespan minimization / LPT
 3 buckets ← PriorityQueue()
 4 items ← PriorityQueue()
 5 for child ∈ children(tree) do
 6 |   if persistent(child) then
 7 |   |   add child as new bucket
 8 |   add child and merged leaves to items
 9 while len(buckets) < ⌊len(children(tree))/2⌋ do
10 |   add empty bucket
11 for item ∈ items do
12 |   add item to least filled bucket in queue
13 |   update bucket position in queue
   // use buckets as new leaves
14 children(tree) ← buckets
   /* forward operation to children, return
      cumulative number of leaves                */
15 leaves ← 0
   for child ∈ children(tree) do
16 |   leaves ← leaves + rakeTree(child)
17 return leaves
```

Next, the compress operation is adjusted. The original implementation intended to execute respective operations upon joining children to their parent, hence it would only be possible to join every other vertex. However, as for the current purpose there are no associated operations, it is possible to remove chains entirely:

Function compress(tree)

```
1 for child ∈ children(tree) do
2 |   compress(child)
3 if isUnary(tree) then
4 |   children(tree) ← children(onlyChild(tree))
```

Concluding, the makespan `WTD` produces equals the heaviest weight of all leaves plus the weight by all parent vertices. Assuming the makespan minimization has linearythmic complexity, the `rake` operation has a theoretical complexity of $\mathcal{O}(nm \log m)$, where n is the number of vertices in the tree and m is the average number of children per vertex. However, it is reasonable to assume $n \gg m$, thus the complexity reduces to $\mathcal{O}(n)$. The complexity of the compress operation is $\mathcal{O}(n)$ as well. Considering that each iteration roughly half of the leaves are removed, `WTD` requires $\mathcal{O}(\log n)$ iterations to reach the targeted number of leaves. Overall, this yields a complexity of $\mathcal{O}(n \log n)$.

4.2 Vertex Weight Calculation

The previous has assumed the tree to be weighted in any form. To approximate the complex behaviour of different DEMKit entities, assigned a base weight of b to each entity class. Next, let the weight of vertex increase exponentially

in accordance with the recursive behaviour of simulations. Thus, the weight of a vertex equals bk^d, where k is the average number of recursive calls and d is the depth of the node. This approach can be further refined by adding the weight of all children to a parent vertex, which coincides with Graham's concept of dynamically formed lists to improve efficiency.

4.3 Example Case

The following displays a single iteration of `WTD` using a small example. Figure 2 displays the used example tree, which is to be reduced to three leaves. Subsequently, Figure 3 shows weights applied using $b = 1.0$ and $k = 3.0$:

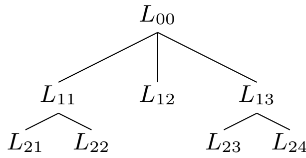


Figure 2: The Used Example Tree

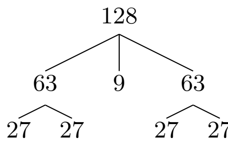


Figure 3: The Applied Weights to the Example Tree

The tree currently contains five leaves on different levels and in different subtrees. Starting with the `rake` operation, the heaviest $\max(3, \lfloor 5/2 \rfloor) = 3$ leaves are marked as persistent. In this case, these are L_{21}, L_{22} and L_{23}. The `rakeTree` operation subsequently applies this selection recursively. Within L_{00}, the two subtrees are persistent and hence lead to the creation of a bucket. Two buckets surpass the minimal amount required; thus, no further ones are added. As a result, L_{12} is merged into one of the other subtrees. Apart from that, the entire subtree L_{11} is marked as persistent, including the children, and therefore stays unchanged. Within L_{13} only L_{23} is selected and forms a bucket. Accordingly, L_{24} is merged into it. This finalizes this operation, the result of which can be seen in Figure 4. The remaining number of leaves equals the target, hence this is the final iteration.

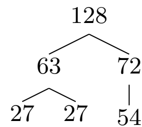


Figure 4: The Result of Applying `rake` to the Tree

Finally, the tree is `compress`ed. A single chain is present, L_{13} to L_{23}, and is removed entirely. The final tree is the following:

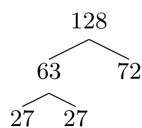


Figure 5: The Result of Applying `compress` to the Remaining Tree

4.4 Evaluation

The proposed algorithm is evaluated using randomly generated, representative trees of varying size, where both maximum width and depth range within $\{x \in \mathbb{N} \mid 4 \leq x \leq 8\}$. This covers different sizes, balanced and unbalanced trees, trees with many or only few chains, etc. Following, the respective trees are weighted using $b = 1.0$ and $k = 3.0$ and contracted for p processes, with $p \in \{2^x \mid x \in \mathbb{N}, 1 \leq x \leq 6\}$, corresponding with common core configurations of modern CPUs. Generated trees that already have less or equal leaves than targeted are omitted from the results shown in Figure 6. All simulations are executed on a quad core (eight threads) Intel®i7-4790K CPU, clocked at 4.3 GHz.

The measured execution times of the contraction (Figure 6a) fit the algorithm's complexity of $\mathcal{O}(n \log n)$. The largest tree was generated with over 85.000 vertices and was reduced to 64 leaves within 7.5 seconds. Furthermore, as shown in Figure 6b the computed makespans are close to the respective lower bound, achieved by dividing the tree's total weight by the targeted number of processes. Following, the average efficiency achieved is approximately 1.58.

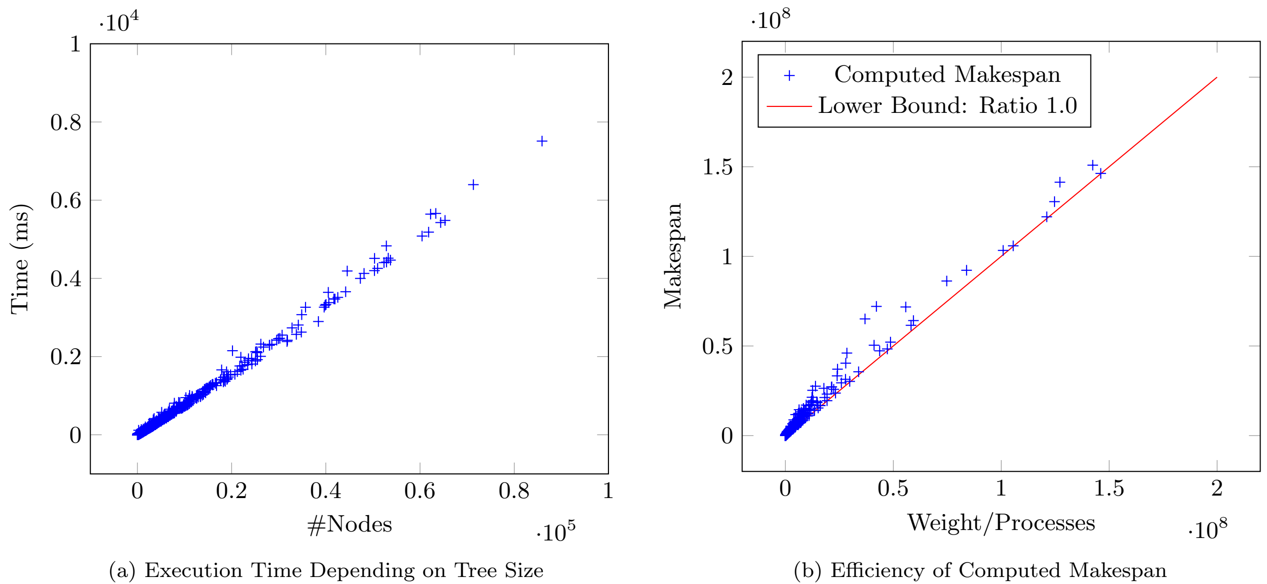
5. DEMKIT INTEGRATION

5.1 Analysis of DEMKit Structure & Implementation

To integrate `WTD` into DEMKit the tree structure of the model needs to be extracted. Subsequently, the model needs to be split up according to the result of `WTD`. The following analyses the assembly and structure of models to find an optimal point of integration.

Any DEMKit model consists of a tree of components managing various entities. The entity class offers a range of basic functions that may be extended upon via inheritance. The different implementations can be grouped roughly into devices and controllers. Most of the time, devices, such as washing machines, form the leaves of the model tree. Opposingly, controllers are vertices within the tree and manage a group of other entities. Unfortunately, this is not a constraint but rather a rule of thumb: The behaviour of child management is not defined within the entity baseclass, but within each implementation individually. Further, there does not exist a controller or device baseclass, all implementations may be separate. This also means that children may not be stored within a single object, e.g. a list, but may be stored using multiple objects. Finally, there exist devices that have children, such as buffers. Ideally, any approach is capable of extracting the required structure data without adding additional constraints or large complexity. The approach chosen offers an interface within the entity baseclass that returns a list of children, which is empty by default. Inheriting classes may override it to customize the returned structure data.

Next, the act of splitting and distributing the model needs to be integrated. This may happen some time between the model compilation phase and the simulation start. Currently, the manual model split is hardcoded into the model, and thus determined pre-compilation. This is of course not a viable option for an automated solution. Further, note that the model compiler does not construct a ready model but rather assembles a set of instructions that instantiate a model. Thus, it is not possible to partially load a model, any attempts would require severe modifications to the model code at runtime. These considerations lead to the following sequence of operations: First, load the model,



(a) Execution Time Depending on Tree Size

(b) Efficiency of Computed Makespan

Figure 6: Results of Reducing 1429 Randomly Generated Trees

extract the model hierarchy using the new, extended entity interface, distribute the tree structure using `WTD` and store the result. Secondly, create master and slave processes. Each of these again load the model in its entirety. Considering the memory footprint of average DEMKit simulations has a magnitude of megabytes, this should not be a limiting factor. Subsequently, replace all references to objects to be located exclusively on other processes by corresponding handles. This is a crucial step, as this not only enables garbage collection, but also eliminates any duplicate computation. Thirdly and finally, execute the simulation using the existing multiprocessing facilities for both master and slaves.

5.2 Evaluation

To evaluate the performance of the `WTD` integration, it is compared to calculated, theoretical performance as per the immediate `WTD` result and to the performance of manual divisions. For this purpose, different, hand-picked, representative simulations are set up, specifically to test scenarios of interest. The models consist of numerous streets, that together contain 400 houses (roughly 2000 entities), which are further distributed among four processes. All simulations are run using the newly developed multiprocessing implementation, with automated model distribution. Weights are calculated with $b = 1.0$ and $k = 4.0$. The respective parallel fractions are gathered by measuring the time spent within slave processes. All results are shown in Table 2.

Firstly, a base case is set up. The according model consists of four streets at 100 houses each. The entire tree is assigned a weight of 1.05 million, with leaves of 260 thousand. As expected, the `WTD` algorithm produces a one-to-one mapping, with one street per process. Accordingly, the distribution should have a theoretical sequential fraction of negligible magnitude, considering that only three entities (rootcontrol-, weather- and sun-entity) reside on the master process. This would therefore result in a perfect scaled speedup of $N = 4$. However, the experimentally measured parallel fraction is 0.9, hence the speedup is approximately 3.7, an efficiency of 92.5%. This observation is likely due to the root controller's higher than average workload. Lastly, this results in a scaled serial fraction of 7.6. The data gathered roughly coincides with results from previous, manually set up tests[13]. Model and process compositions like this have been used for parallelized simulations before, because of the comparatively trivial division. Hence, more complicated models are set up for testing in following cases.

Secondly, three streets at 133 houses each are distributed. Here, `WTD` assigns two processes one street each, whereas the final street is divided equally between the two remaining processes. The parallel fraction is slightly improved, measuring 0.95. Accordingly, the speedup is 3.8, an efficiency of 96.1%. However, in contradiction to this, the total execution time is almost 60% higher compared to the first test case. This inconsistency can be explained using the scaled serial fraction: In this test case it equals 15.0. As the scale factor k is roughly equal in both cases, the fractions should ideally be equal as well. Instead, the doubled value strongly indicates that a significant part of the measured parallel fraction is due to communication costs and/or idle times.

Thirdly, five streets at 80 houses each are distributed. This case displays inefficient distribution, as instead of splitting up the fifth street among all four processes, it is assigned in its entirety to a single process. This result is rooted at one of the first constraints set up for the development of `WTD`: Subtrees are to be assigned as a whole, once reduced it cannot be split up again. Despite this inefficiency, this model setup performs similar to the first test case in all aspects. This could be an indication of dominating communication cost or idle times. However, taking the slightly smaller street size into account, this could also be a direct consequence of the polynomial complexity of the control algorithm. In any case, further research is necessary to determine the definite cause of this observation.

Finally, a model with more heterogeneous properties is tested. Five streets are distributed, with different numbers of houses as detailed in Table 2. Here, `WTD` assigns the smallest two streets to one process, the other three streets are handled by one process each. Again, the measured performance is very similar to the base case.

Concluding, `WTD` successfully distributes models of different underlying structures. The test cases lead to two major observations. Firstly, if the need arises to split up a

Table 2: Overview of Tested Models and Respective Results

Streets	Houses per Street	Time (h)	p	Speedup	s_k
4	100	2.3	0.9	3.7	7.6
3	133	3.7	0.95	3.8	15.0
5	80	2.4	0.9	3.7	7.6
5	$[40, 60, 80, 100, 120]$	2.3	0.9	3.7	7.7

highly dependent substructure, heavy performance losses are to be expected due to communication costs. Secondly, if the distributed structures are mostly independent from another, losses due to imperfect distributions are negligible. This holds for homogeneous as well as heterogeneous model setups. Therefore, as a general guideline to optimize performance, models should contain many independent structures.

6. CONCLUSION

Concluding, an algorithm (`WTD`) to divide weighted trees into equal parts is presented, by utilizing concepts of tree contraction and makespan minimization. The algorithm's implementation executes in linearrythmic time, as determined theoretically and experimentally. Trees consisting of several 10^5 vertices can be reduced within a few seconds. To optimally weigh the tree with respect to recursive DEMKit simulations, vertices are assigned a value growing exponentially in terms of depth. Following, `WTD` is integrated into DEMKit successfully using preexisting multiprocessing components. Automated, simple configurations, such as direct street to process mapping, perform similar to manual distributions, as expected. Less trivial configurations, including heterogeneous and homogeneous models, all perform similar to this base case, with a speedup of approximately 3.7 and a serial fraction of 7.6. The exception to this is a test case of three streets to four processes, leading to the split of street across two processes. This case requires 60% more time to complete, caused by an increase in communication costs and idle times, as indicated by a significantly higher (15.0) serial fraction. Overall, automatically parallelized models perform best if the distributed structures are independent

Future improvements to the parallelization of DEMKit, such as a reduction in communication delays, could further improve the performance of automated distributions, as indicated by the second test case. Apart from that, it may be useful to experimentally determine the values of b and k individually for each entity class to improve accuracy of the applied weights. Lastly, should the need arise, e.g. due to further horizontal connections, a more complex makespan minimization algorithm, such as `MR`, could be integrated into `WTD`.

7. REFERENCES

[1] S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, 1999.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[3] R. Chandrasekaran, B. Chen, G. Galambos, P. R. Narayanan, A. Van Vliet, and G. J. Woeginger. A note on "an on-line scheduling heuristic with better worst case ratio than graham's list scheduling". *SIAM Journal on Computing*, 26(3):870–872, 1997.

[4] B. Chen, A. van Vliet, and G. J. Woeginger. New lower and upper bounds for on-line scheduling. *Operations Research Letters*, 16(4):221 – 230, 1994.

[5] U. Faigle, W. Kern, and G. Turán. On the performance of on-line algorithms for partition problems. *Acta cybernetica*, 9(2):107–119, 1989.

[6] R. Fleischer and M. Wahl. Online scheduling revisited. In M. S. Paterson, editor, *Algorithms - ESA 2000*, pages 202–210, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[7] G. Galambos and G. J. Woeginger. An on-line scheduling heuristic with better worst-case ratio than graham's list scheduling. *SIAM Journal on Computing*, 22(2):349–355, 1993.

[8] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[9] M. E. T. Gerards, H. A. Toersche, G. Hoogsteen, T. van der Klauw, J. L. Hurink, and G. J. M. Smit. Demand side management using profile steering. In *2015 IEEE Eindhoven PowerTech*, pages 1–6, 2015.

[10] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[11] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[12] G. Hoogsteen. *A Cyber-Physical Systems Perspective on Decentralized Energy Management*. PhD thesis, University of Twente, Netherlands, 12 2017.

[13] G. Hoogsteen, M. E. T. Gerards, and J. L. Hurink. On the scalability of decentralized energy management using profile steering. In *2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pages 1–6, 2018.

[14] G. Hoogsteen, J. L. Hurink, and G. J. M. Smit. Demkit: a decentralized energy management simulation and demonstration toolkit. In *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, pages 1–5, 2019.

[15] D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20(2):400 – 430, 1996.

[16] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990.

[17] G. L. Miller and J. H. Reif. Parallel tree contraction – part i: Fundamentals, 1989.