Mapped Taskgraphs as a Tool for Optimization in Static Taskgraph Scheduling

Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Joerg.Keller@FernUni-Hagen.de

Patrick Eitschberger
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Patrick.Eitschberger@FernUni-Hagen.de

***Abstract*—We present the concept of mapped taskgraphs, which comprise all information of a static taskgraph and the corresponding static schedule. Mapped taskgraphs allow to reason about and optimize static schedules for taskgraphs in the language of graphs. This allows to employ the wealth of graph algorithmics, and still be able to extract easily a schedule with a task order similar to the original schedule, but possibly smaller makespan. We prove that our construction has the above properties, and present two exemplary applications that use mapped taskgraphs. One is new, the other a number of years old but only mentions a variant of this technique in passing and thus does not explicitly or formally explain it or deal with it.**

***Index Terms*—static taskgraph scheduling, graph algorithms, makespan optimization**

I. Introduction

Static taskgraphs are a widely-used model for parallel applications [1], and static scheduling of taskgraphs for later execution on parallel machines has been researched for decades [2]. When a static schedule has been computed, there are a number of techniques that do post-processing with the schedule, which leave the task order on the processors as it was, but change the schedule in other ways. For example, one might be interested how the makespan of the schedule is affected if some of the tasks have a runtime that slightly and randomly differs from the execution time that was forecasted in the taskgraph (a situation that occurs quite frequently in practice, but is normally not considered in static taskgraph scheduling). As a second example, one might ask which task would have to be accelerated by a certain percentage to maximally reduce the makespan of the resulting schedule, without having to schedule again (a question that occurs when parallel codes shall be tuned with restricted effort). This means, the tasks remain mapped and ordered on the cores as previously, and only their start times change.

These questions can be answered with the help of graph algorithms. For example, in the second application scenario it suffices to look at tasks on the critical path of the taskgraph [3]. However, the constraint that the task order in the schedule should remain unaltered excludes to only use the critical path in the taskgraph as given to the scheduler. Hence, a technique is needed to import this constraint into the taskgraph. In addition, when the most promising task has been identified and the code improved, it should suffice to adapt this task runtime in the taskgraph to re-construct the starting times of the tasks in the schedule without much effort.

As a solution to the above challenge, we present the concept of *mapped taskgraph*, by which we mean a taskgraph that is extended to represent also the constraints imposed by the corresponding schedule. Without being too formal, we will provide arguments that the mapped taskgraph has the desired properties from the previous paragraph. We demonstrate the applicability of our concept by describing two examples of post-processing, which we have sketched already above: taskgraph tuning [3] and static scheduling for taskgraphs with randomized task runtimes. While the latter scenario is rather new, especially for normally distributed task runtimes, the former is almost 10 years old, but has used this technique only implicitly, only alluding to it within a single paragraph, and not arguing formally. We are aware that similar ideas have been used in a number of works. However, they only use the mapped taskgraph as a tool but do not explain or present it explicitly. Hence, we see our contribution in making this method and possible applications visible. By describing it in detail, we enable its use by other researchers without the necessity to re-invent it over and over.

The remainder of this article is structured as follows. Sect. II provides definitions and properties of static taskgraphs and taskgraph scheduling. In Sect. III we present the concept of a mapped taskgraph and argue about its properties. Sect. IV explains how to use mapped taskgraphs in two example applications. Finally, in Sect. V we give conclusions and an outlook on future work.

II. Basics on Taskgraphs and Static Scheduling

A *taskgraph*[1] is a directed, acyclic graph $G = (V, E)$ where the nodes in V represent the tasks (work) to be executed, and are annotated with their runtimes (denoted by $t(v)$), while the edges in E represent dependencies between tasks, and are annotated with their communication time (denoted by $c(u, v)$). Instead of annotations, we will also talk of weights in the sequel. This model assumes that a task receives input (via incoming edges) only at the beginning, and sends output (via outgoing edges) only when it is completely executed. Hence,

[1]In the grid community, taskgraphs are also called workflows [1].

a task can only start when all of its inputs are present. For each task, we can compute a static t-level, which is defined as follows. For a task v with no incoming edges, $tl(v) = 0$. Otherwise,

$$tl(v) = \max\{tl(u) + t(u) + c(u,v) \mid (u,v) \in E\} , \quad (1)$$

i.e. for each predecessor task (whose t-level is already computed, because we have an acyclic graph) we compute its completion time by adding its runtime, and also incorporate the communication time between predecessor and actual task.

We assume that the taskgraph has a single source node α and a single sink node Ω, each with runtime 0 and with outgoing (resp. incoming) edges of weight 0. If the taskgraph does not have this, these nodes can easily be added.

The t-levels can be computed in time $O(|V| + |E|)$ from a topological sorting of the nodes. When we mark (for each node) the edge that determines the maximum in Eq. (1), and follow these edges backward starting from Ω, we get the *critical path* of the taskgraph, i.e. the longest path from α to Ω. Please note that there may be several critical paths.

Figure 1(a) depicts a graphical representation of a taskgraph with 6 tasks (plus α and Ω), where for simplicity the runtimes are all the same (i.e. can be assumed to be unit) and the communication times are 0. The critical path has length 4 (either 0-1-4-5 or 0-2-4-5), i.e. the critical path of a taskgraph need not be and in this example is not unique.

Taskgraphs can be derived from parallel program code either with automatic analysis tools (see e.g. Gordon's method from Streamit benchmarks [4]), can be known as e.g. for parallel FFT or parallel mergesort, or can be derived by a manual analysis (e.g. Foster's PCAM method [5]).

A *static schedule* for a given taskgraph G and a machine with a given number p of cores is a mapping of the tasks (excluding α and Ω) onto the cores such that

- Each task v is mapped onto exactly one core, and given a start time $s(v)$.
- A core never executes more than one task at once, so for any two tasks u and v mapped to the same core with $s(u) \leq s(v)$, we require $s(v) \geq s(u) + t(u)$.
- Task start times must respect the dependencies, so for any edge (u,v), we require $s(v) \geq s(u) + t(u)$. If the tasks are mapped to different cores, also the communication must be respected, i.e. $s(v) \geq s(u) + t(u) + c(u,v)$.

Sometimes, a schedule is defined with only the first two properties, and schedules also having the third property are called *feasible*.

The idea of the static schedule is that it is computed only once and offline, i.e. prior to execution, so that the execution scheme is rather simple: a processor executes the tasks assigned to it one by one, and starts the next task as soon as the previous one is completed and all inputs for the next task are available. If the runtimes and communication times are known exactly (as assumed in this model), then each task will start exactly at its assigned start time.

Please note that the previous sentence assumes a hidden assumption: that there is no schedule with identical mapping but shorter makespan, i.e. that there is no possibility to simply set a task to an earlier start time without violating one of the requirements above. As the goal typically is to compute a schedule with shortest possible makespan, this assumption seems obvious, but as it is not included as a requirement in the definition of a schedule, it should be made explicit, especially as we later want to refer on it. We will call this property *trivial incompressibility*.

From the second requirement above we can derive the following property. If each core i, where $i \in \{0, \ldots, p-1\}$, has n_i tasks mapped to it (we assume $n_i \geq 1$ in the sequel), we can index each mapped task as $t_{i,j}$ if it is mapped to core i and is the j-th task executed by that core.

It is customary to set $s(\alpha) = 0$ (or more exactly: $s(v) = 0$ for all tasks solely dependent on α, as α itself is not mapped), and thus the time when the last task completes execution, i.e. $s(\Omega) = \max\{s(v) + t(v) \mid v \in V\}$, defines the length of the execution and is called the *makespan* of the schedule.

Please note that the makespan of a schedule can be longer or shorter than the critical path of the underlying taskgraph. Figure 1(b) shows a schedule corresponding to the taskgraph from (a), mapped onto two cores. Here the makespan is 5 and thus longer than the critical path, whose length is 4. For a taskgraph consisting of a chain of tasks with communication times, which are all mapped onto one core, the makespan is shorter than the critical path, because the communication times can be ignored in the schedule, as all tasks are on the same core.

Computing the static schedule of a taskgraph with minimum makespan is known to be an $\mathcal{NP}$-hard problem, and hence many scheduling heuristics have been developed [2]. The existence of heuristics also explains the existence of post-processors or boosters that try to locally improve a schedule after its computation and thus reduce the makespan, without computing a completely new schedule. Such boosters might profit from the concept presented in the next section.

III. The Concept of Mapped Taskgraphs

We define a mapped taskgraph as follows. Given a static taskgraph $G = (V, E)$ and a corresponding schedule S for a machine with p processors, the *mapped taskgraph* $G_M = (V_M, E_M)$ is a directed, acyclic graph with

$$V_M = V \cup \{c_1, \ldots, c_p\} \quad (2)$$

$$E_M = E \cup \{(\alpha, c_1), \ldots, (\alpha, c_p)\} \cup \{(c_i, t_{i,1}), (t_{i,1}, t_{i,2}), \ldots (t_{i,n_i-1}, t_{i,n_i}) | i = 1 \ldots p\} . \quad (3)$$

The node weights for the extra nodes and the edge weights for the extra edges are 0. Finally, if E comprises an edge $(t_{i,j}, t_{i,k})$, i.e. an edge between tasks mapped to the same core, then the edge weight is set to 0.

Speaking in general terms, we add a node for each core, and link all tasks mapped to one core into a chain. This is illustrated in Fig. 1, where we see a taskgraph in part (a). For simplicity, all communication edges have weight 0, and all tasks have the same execution time. Part (b) shows the

corresponding schedule for two cores. Part (c) depicts the mapped taskgraph. The colored edges indicate the chains of tasks mapped to each core. For illustration, we draw two edges when such a chain edge is added and tasks already have a dependency edge (e.g. between tasks 3 and 5.) Please note that this is just for demonstration. As E_M is defined as a set, there is only one edge between this pair of nodes. However, we might annotate an edge as being a dependency edge, a mapping (or chain) edge, or both.

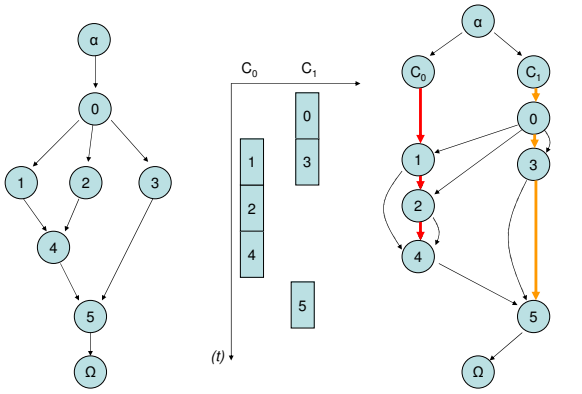

Fig. 1. Mapped Taskgraph Example: (a) Taskgraph, (b) Schedule for 2 processors, (c) Mapped Taskgraph.

The first usable property of the mapped taskgraph is given in the following Prop. 1.

Proposition 1: The length of the critical path in the mapped taskgraph equals the makespan of the underlying schedule. ◊

Before we prove this, please note that this property is not given in the original taskgraph. In Fig. 1(a), the critical path is 0-1-4-5 (there is another of the same length: 0-2-4-5) and has length 4, while the makespan of the schedule is 5.

We argue that Prop. 1 holds by considering two cases: if the makespan is shorter (to be exact: not longer) than the critical path in the original taskgraph, this can only be because of ignoring communication times when mapping tasks onto the same core. As these communication times are set to 0 in the mapped taskgraph, the length of its critical path will reduce accordingly.

If the makespan is longer than the critical path in the original taskgraph, then this is due to the fact that a task is started later that it could be given the task dependencies, because a core is busy before. As this constraint is added by adding chain edges, the critical path is increased accordingly.

Please note that in this argument we use the fact that the schedule has the trivial incompressibility property.

As a corollary, we get that the start times of the tasks in the schedule correspond exactly to the t-levels of the tasks in the mapped taskgraph.

Another helpful property can thus be derived in Prop. 2.

Proposition 2: If we change the runtime of a task, then recomputing the t-levels in the mapped taskgraph and using these as the new start times of the tasks in a schedule with unchanged mapping will produce a feasible schedule. ◊

IV. Applications

In this section, we present two applications that profit from and use the concept of a mapped taskgraph.

A. Taskgraph tuning

Assume that you have a taskgraph and the corresponding schedule. In order to speedup the computation, we plan to investigate the underlying code, and we are confident that we can speedup a task, i.e. reduce its runtime, by some given percentage. However, due to a lack of human resources, we can only improve one task. The question to be solved is: which task should we choose for improvement, so that the makespan reduction of the schedule is maximum? The schedule structure, i.e. the mapping of tasks to cores, and the order of tasks on each core, should remain intact (we will give some explanation for this constraint below).

As a brute force approach, we can compute the mapped taskgraph, reduce the runtime of a task, and compute the length of the critical path. This is done for each task, and the task code leading to the shorted critical path is investigated for improvement.

In [3], a faster solution is given, that reduces the mapped taskgraph to a linegraph comprising only the critical path (plus some edges that implement constraints from other paths through the graph), and solves a linear optimization problem (on the much smaller number of tasks of the critical path).

Please note that in that paper, the mapped taskgraph is only mentioned (without using that expression) in a paragraph on the second page. The reason behind is trivial: the taskgraphs considered in that paper were all mapped taskgraphs. They were derived from executions of MPI programs. In such a program, it is obvious that a task starts with a receive instruction (or because the previous task in that MPI process has ended) and ends with either a send instruction or before a receive instruction [6]. As all the code pieces (tasks) that are executed during execution of an MPI process are also linked in a chain, there are as many chains as there are MPI processes.

With the explicit use of the mapped taskgraph concept, the approach proposed in [3] for traces of MPI programs can also be used for taskgraphs and schedules computed in other settings.

B. Taskgraphs with random runtimes

In Sect. II, we have assumed that task runtimes are known exactly. However, this might not be true in practice for a number of reasons. On the one hand, actions of the operating system like executing interrupts, scheduling some other system process inbetween or adapting the processor frequency in case of heating up might influence the runtime, but normally these influences only lead to some “noise” in the runtimes comprising some percent at most, and are ignored.

On the other hand, the algorithm that executes a task may have runtime differences depending on the structure of its

input data (e.g. presorted or not) or might be a randomized algorithm so that the runtime is a random variable following some distribution (e.g. normal distribution) with a mean and a variance.

As typical scheduling heuristics are given deterministic task runtimes (which might be the mean in case of a task with random runtime), we are interested how the makespan of the corresponding schedule is influenced, as now the time when a task will start, i.e. when all of its inputs are present, does not necessarily correspond to its assigned start time anymore. The makespan has become a random variable itself.

While computing the runtime of a chain of dependent tasks is straightforward (if the task runtimes are random variables X_1 and X_2, then the compound $X_1 + X_2$ can e.g. be easily expressed for normal distributions), the case of a task depending from two predecessors is more difficult. Here, the completion time of the dependent task is $X_3 + \max(X_1, X2)$, if its own runtime is expressed as a random variable X_3, and the completion times of its predecessors are given as random variables X_1 and X_2. Skipping the addition of X_3, the maximum $X = \max(X_1, X2)$ can be expressed e.g. by

$$P(X \leq x) = P(X_1 \leq x) \cdot P(X_2 \leq x) \;, \tag{4}$$

if the random variables X_1 and X_2 can be considered to be independent.

Again, we have expressed this computation on a taskgraph. As we are interested in the makespan of the schedule, we can use the mapped taskgraph in this case, and by Prop. 1 computing the distribution of the length of the critical path allows to compute the mean of the schedule's makespan.

As complex dependency structures in a taskgraph might lead to multiple nested maximums in the form of Eq. (4), the analytical solution for the expectation can be difficult. Eq. (4) uses the cumulative distribution function $F_X(x) = P(X \leq x)$ while computing the expectation needs the probability density function $f_X(x) = F'_X(x)$ which is the derivate of the cumulative distribution function, i.e. it is necessary to differentiate a product of many factors. For the example from Eq. (4) we get $f_X(x) = f_{X_1}(x)F_{X_2}(x) + f_{X_2}(x)F_{X_1}(x)$, i.e. the derivate also contains terms coming from distribution functions. To compute the expectation $E[X] = \int_{-\infty}^{\infty} x \cdot f_X(x)\, dx$ we also have to multiply such products with x, but to compute the variance $\sigma^2(X) = E[X^2] - (E[X])^2$ one has to multiply even with x^2 when computing $E[X^2]$. Thus, analytical solutions can become difficult especially in case of normal distributions where already the antiderivate of the product from cumulative distribution function, i.e. Gaussian error function, and probability density function poses challenges. This may explain why Martin [7], who was investigating PERT diagrams (where the problem of scheduling does not appear given that enough human resources are available), concentrated on polynomial distributions.

As an example, Fig. 2 illustrates that the distribution of the maximum of two normally distributed variables (same variance, but different mean, both approximated by binomial distributions) is not normally distributed.

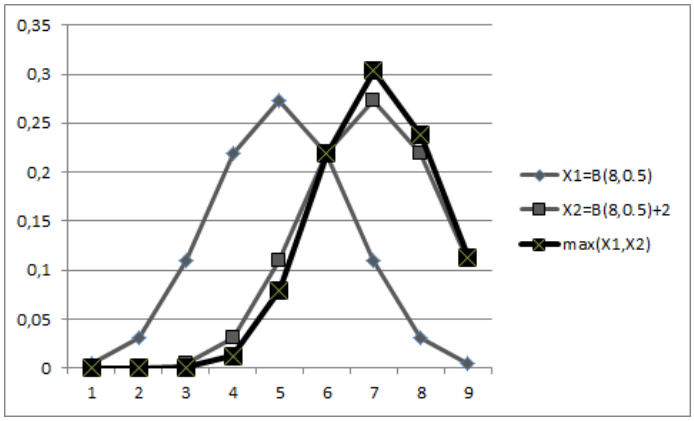

Fig. 2. Maximum of independent, normally distributed random variables with different means.

For practical purposes, another approach is proposed, using the concept of mapped taskgraphs. Given a taskgraph with known, randomly distributed task runtimes, we compute a schedule using the means as task runtimes, and then compute the mapped taskgraph. Now, a large number of samples can be computed by randomly choosing concrete task runtimes according to the distributions, and computing the length of the critical path, i.e. makespan, in the mapped taskgraph for each sample. Thus, the mean and variance of the makespan can be computed approximately. Colajanni et al. [8] provided bounds from below and above on the distribution, to compute the mean, but they did not consider the variance, and they seemingly did not use normal distributions.

V. Conclusions

We have presented the concept of mapped taskgraphs to allow post processing of static schedules for taskgraphs in the language of graphs. While the concept has been used previously, to our knowledge it has not been presented explicitly before.

Moreover, we have presented two example applications of this concept, in particular computation of the makespan distribution if task runtimes have random distributions. The latter application shows the necessity of a tool for quick calculations on schedules, as for normal distribution of task runtimes (which might seem a natural model), analytical computation of the expected value of the makespan would necessitate to integrate products containing the Gaussian error function as a factor. Thus, the distribution and mean of the makespan might better be approximated by creating many samples of the mapped taskgraph with randomly chosen task runtimes, and computing the makespan for each.

In the future, we plan to use mapped taskgraphs especially for further exploration of scheduling static taskgraphs with random task runtimes. In particular, it might be interesting to know which (deterministic) task runtimes should be assumed in a static scheduler (mean plus one standard deviation, or even larger values) to have a schedule whose expected makespan is not much larger than the expected makespan of a schedule

that works with mean task runtimes, but where the makespan distribution is asymmetric in the sense that values that are much larger than the projected makespan are extemely seldom.

ACKNOWLEDGMENT

We are very grateful to Wolfram Schiffmann, who helped to shape the mapped taskgraph concept in several discussions, and to Wolfgang Spitzer, who introduced us to analytic solutions for taskgraphs with randomized task runtimes.

REFERENCES

[1] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *J. Grid Computing*, no. 3–4, pp. 171–200, 2005.

[2] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, 1999.

[3] J. Keller and W. Schiffmann, "Guiding performance tuning for grid schedules," in *Proc. 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, May 2009.

[4] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data and pipeline parallelism in stream programs," in *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, 2006, pp. 151–162.

[5] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[6] M. Schulz, "Extracting critical path graphs from MPI applications," in *Proc. IEEE International Conference on Cluster Computing (Cluster 2005)*, 2005, pp. 1–10.

[7] J. J. Martin, "Distribution of the time through a directed acyclic network," *Operations Research*, vol. 13, pp. 46–66, 1965.

[8] M. Colajanni, F. L. Presti, and S. Tucci, "A hierarchical approach for bounding the completion time distribution of stochastic task graphs," *Performance Evaluation*, vol. 41, pp. 1–22, 2000.