The Expressive Power and Complexity of Dynamic Process Graphs*

Andreas Jakoby[1,2] Maciej Liśkiewicz[2] Rüdiger Reischuk[2]
[1]University of Toronto [2]Med. Universität zu Lübeck

Abstract

A model for parallel and distributed programs, the *dynamic process graph,* is investigated under graph-theoretic and complexity aspects. Such graphs are capable of representing all possible executions of a parallel or distributed program in a very compact way. The size of this representation is small – in many cases only logarithmic with respect to the size of any execution of the program. An important feature of this model is that the encoded executions are directed acyclic graphs with a *regular* structure, which is typical of parallel programs, and that it embeds constructors for parallel programs, synchronization mechanisms as well as conditional branches.

In a previous paper we have analysed the expressive power of the general model and various restrictions. Furthermore, from an algorithmic point of view it is important to decide whether a given dynamic process graph can be executed correctly and to estimate the minimal deadline given enough parallelism. Our model takes into account communication delays between processors when exchanging data.

In this paper we study a variant with output restriction. It is appropriate in many situations, but its expressive power has not been known exactly. First, we investigate structural properties of the executions of such dynamic process graphs $\mathcal{G}$. A natural graph-theoretic conjecture that executions must always split into components isomorphic to subgraphs of $\mathcal{G}$ turns out to be wrong. We are able to establish a weaker property. This implies a quadratic bound on the maximal deadline in contrast to the general case, where the execution time may be exponential. However, we show that the problem to determine the minimal deadline is still intractable, namely this problem is $\mathcal{NEXPTIME}$-complete as is the general case. The lower bound is obtained by showing that this kind of dynamic process graphs can represent certain Boolean formulas in a highly succint way.

1 Introduction

Large parallel or distributed computations tend to have a lot of regularities. For example, the same instruction sequence may be executed by many processors in parallel. To describe the elementary steps and the logical dependencies among them one can use graphs, often called *data flow graphs.* One would like to keep this description as compact as possible, for example not to unfold parallelism if this is not necessary. For this purpose, we have introduced the model of dynamic process graphs in [3]. For a more detailed motivation the reader may check there.

The novel property of this graph model are different modes for the input and output behaviour that can be assigned to each node, which models a process of the computation. This allows us to model basic primitives for writing parallel programs, like *fork* and *join.* The two modes are called ALT and PAR. To represent such parallel programs in a natural way, we introduce dynamic process graphs, which are generalizations of standard precedence graphs. A dynamic process graph is an acyclic graph $G = (V, E)$ with two sets of labels $I(v), O(v) \in \{\mathsf{PAR}, \mathsf{ALT}\}$ attached to the nodes $v \in V$. Nodes represent tasks, edges dependencies between tasks. A complete formal definition will be given in the next section. The label $I(v)$ describes the *input mode* of task v. If $I(v) = \mathsf{ALT}$ then to execute v at least one of the direct predecessor tasks has to be completed. $I(v) = \mathsf{PAR}$ requires that executions of all direct predecessors of v have to be completed before v can start. If task v gets executed then according to the *output mode*

*This is an extended abstract to be presented at 26th International Workshop on Graph-Theoretic Concepts in Computer Science, Konstanz, Germany, 2000.

$O(v)$ one of its direct successors in case $O(v) = \mathsf{ALT}$ (resp. all of them in case $O(v) = \mathsf{PAR}$) has to be executed as well.

If one restricts the mode of nodes to PAR only, both for input and output, then this will be equivalent to ordinary data flow graphs. Using both modes, however, the representation of parallel programs by dynamic process graphs can provide an exponential compaction compared to the length of the actual execution sequences. Given a dynamic process graph, the first question that arises is whether it describes a legal parallel program. If yes then one would like to find an efficient execution of the program specified in such a compact form. We assume here that enough parallelism is available, so that the question turns into the problem to execute the program as fast as possible.

Dynamic process graphs and Boolean circuits are somehow related. We have shown that such graphs can be used to model computations of a circuit. This has then been used to prove that indeed dynamic process graphs that can use arbitrary combinations of modes provide a very compact nontrivial representation. To find an optimal schedule, which is $\mathcal{NP}$-complete for ordinary graphs, turns out to be $\mathcal{NEXPTIME}$-complete. A similar complexity jump has been observed for classical graph problems in [1, 5, 4]. These papers have shown that simple graph properties become $\mathcal{NP}$-complete when the graph is represented in a particular succinct form using generating circuits or a hierarchical decomposition. Under the same representation graph properties that are ordinarily $\mathcal{NP}$-complete, like HAMILTON CYCLE, 3-COLORABILITY, CLIQUE (with half the graph size), etc., become $\mathcal{NEXPTIME}$-complete.

If we put restrictions on the modes of a dynamic process graph its execution becomes easier. We have given a precise classification in [3] for all possible combinations, except the most interesting case where the output mode is restricted to PAR, but the input mode may use both alternatives. Such graphs are still able to model, for example, the two natural ways in which objects of an object-oriented languages can be activated: the total case, where all input parameters have to be specified before activation, and the partial case, where an object fires for any specified parameter once.

For dynamic process graphs with PAR output mode we could show that finding an optimal schedule is at least $\mathcal{BH}_2$-hard. There was no matching upper bound, only the $\mathcal{NEXPTIME}$ upper bound of the unrestricted case. Other combinations turn out to be less powerful. Restricting the input mode to ALT the problem becomes $\mathcal{P}$-complete, and also fixing the output mode the problem gets $\mathcal{NL}$-complete.

As main results of this paper we show the following two complexity bounds. A linear upper bound on the length of executing dynamic process graphs with PAR output mode, whereas in the unrestricted case this bound may be exponential. Thus, restricting the expressive power of this graph representation decreases the time complexity of parallel and distributed programs drastically. On the other hand, we prove that computing an optimal execution in a setting that takes communication delays into account remains hard – this problem is still $\mathcal{NEXPTIME}$-complete.

In the next section, we will give a formal definition of this graph model. In section 3 we exhibit structural properties of executing dynamic process graphs with PAR output mode restriction, which will imply the linear upper bound. Section 4 considers the scheduling problem and gives a general outline how the $\mathcal{NEXPTIME}$-hardness result can be shown. Due to space limitations, a bunch of technical details have to be omitted.

2 Dynamic Process Graphs and Runs

Given a DAG $G = (V, E)$ with node set V and edge set E, for $v \in V$ let **pred**$(\boldsymbol{v}) := \{u_1, \ldots, u_p\}$ denote the set of its direct predecessors, and **succ**$(\boldsymbol{v}) := \{w_1, \ldots, w_r\}$ its direct successors. **pred**$^*(\boldsymbol{v})$ is the set of all anchestors of v.

Definition 1 *A* **dynamic process graph**, DPG *for short,* $\mathcal{G} = (V, E, I, O)$ *consists of a* DAG *(directed acyclic graph) with node set V and edges E and two node labellings* $I, O : V \to \{\mathsf{ALT}, \mathsf{PAR}\}$. $V = \{v_1, v_2, \ldots, v_n\}$ *represents a set of* **processes** *and E dependencies among them. I and O describe* **input**, *(resp.* **output***)* **modes** *of the v_i.*

A finite DAG $H_{\mathcal{G}} = (W, F)$ is a **run** *of $\mathcal{G}$ iff the following conditions are fulfilled:*

1. *The set W is partitioned into subsets $W(v_1) \dot\cup W(v_2) \dot\cup \ldots \dot\cup W(v_n)$. The nodes in $W(v_i)$ are* **execution instances** *of the process v_i, and will be called* **tasks**.

2. *Each source node of $\mathcal{G}$, which represents a starting operation of the program modelled by $\mathcal{G}$, has exactly one execution instance in $H_{\mathcal{G}}$.*

3. *For every $v \in V$ with pred$(v) = \{u_1, \ldots, u_p\}$ and succ$(v) = \{w_1, \ldots, w_r\}$ and every execution instance $x \in W(v)$ it holds:*

- *if $I(v) =$ ALT then x has a unique predecessor $y \in W(u_i)$ for some $i \leq p$;*
- *if $I(v) =$ PAR then pred$(x) = \{y_1, \ldots, y_p\}$ with $y_i \in W(u_i)$ for each $i \leq p$;*
- *if $O(v) =$ ALT then x has a unique successor $z \in W(w_j)$ for some $j \leq r$;*
- *if $O(v) =$ PAR then succ$(x) = \{z_1, \ldots, z_r\}$ with $z_j \in W(w_j)$ for each $j \leq r$.*

We call a DPG $\mathcal{G}$ **executable** *iff there exists a run for it.*
Given $\mathcal{G}$ with run $H_{\mathcal{G}} = (W, F)$, for each edge (u, v) in $\mathcal{G}$ we define $F(u,v) := \{(y, z) \in F \mid y \in W(u)$ and $z \in W(v)\}$.

Throughout the paper we will illustrate the ALT-mode by a white box and the PAR-mode by a black box. To represent a node of a DPG with input and output mode e.g. ALT, resp. PAR we will use two boxes: white at the top representing the input mode and the black box at bottom for output mode. For a source or a node with indegree 1 the input mode obviously is inessential. Hence we will ignore such a label, and similarly the output label in case of a sink or a node with outdegree 1.

Fig. 1 gives an example of a DPG and two runs of it. DPGs can be used to specify parallel programs in a compact way. A run corresponds to an actual execution of the program. Observe that a run can be smaller than its defining DPG (see [3] Fig. 6). More typically, however, a run will be larger than the DPG itself since the PAR-constructor allows process duplications. The runs in Fig 1 illustrate this property. The following lemma gives an upper bound on the blow-up, resp. the possible compaction ratio of DPGs.

Lemma 1 ([3]) *Let $\mathcal{G} = (V, E, I, O)$ be a DPG and $H_{\mathcal{G}} = (W, F)$ be a corresponding run. Then $|W| \leq 2^{|V|-1}$, and this general upper bound is best possible.*

Certain DPGs thus have processes with exponential many execution instances.

The motivating question for our study of DPGs is *how efficiently a compactly specified parallel program Π can be executed.* Note that this is a crucial problem during compile-time. Furthermore, we like to construct an optimal schedule for the execution tasks of Π (compare e.g. [2]). Lemma 1 implies an upper bound for the execution time of such a compactly presented program. However, this bound may be much too pessimistic for massive parallelism. With respect to their size, DPGs may require exponential many execution instances for some of its processes. However, if enough processors are available it may be possible to construct runs that use only linear time by executing many tasks in parallel.

Definition 2 *Let $\mathcal{G} = (V, E, I, O)$ be a DPG and $H = (W, F)$ be a run of $\mathcal{G}$. A* **subrun** $\boldsymbol{\mathcal{R}(y)}$ *of H is a subgraph that is induced by one of its sink nodes y, i.e. it consists of a sink y and all nodes in* $\mathbf{pred}^*(y)$ *together with all their connections in F. We call a subrun $\mathcal{R}(y)$ of H* $\boldsymbol{k}$**-overlapping** *iff for every node $v \in V$ it holds $|\mathcal{R}(y) \cap W(v)| \leq k + 1$. A run H is k-overlapping if all its subruns are k-overlapping. We call H non-overlapping if it is 0-overlapping.*

Note that non-overlapping is a slightly stronger condition than the requirement that each subrun has to be isomorphic to a subgraph of $\mathcal{G}$. The isomorphism requirement seems to be a very natural condition. Fig. 1, however, shows that it does not have to hold, even for DPGs with output restriction PAR. The two execution instances of the sink w induce two different subruns. For the run illustrated in (b) each subrun is non-overlapping, it is isomorphic to a subgraph of $\mathcal{G}$. In (c) each subrun contains both execution instances of process t, that means it is 1-overlapping.

The maximal size of a subrun of H gives a better upper time bound for executing H than just the size of H because all subruns can be executed independently in parallel. Moreover, if a DPG $\mathcal{G}$ has a non-overlapping run then its processes can be executed in linear time with respect to the size of $\mathcal{G}$.

Next, we give a formal definition for scheduling DPGs. Let $\boldsymbol{\delta}$ be a parameter describing the communication delay of the system. For the lower bounds it suffices to treat δ as a constant independent of the processes between which data is exchanged. Furthermore, it suffices to assume unit-execution time

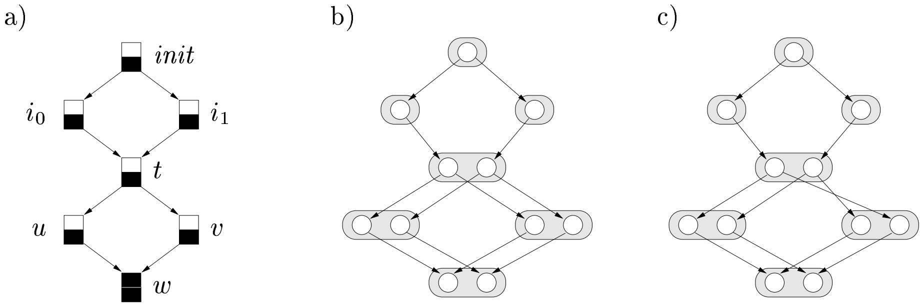

Figure 1: *(a) A DPG with 2 nonisomorphic runs: process t has 2 execution instances, and the same holds for all its successors; (b) a non-overlapping run, (c) a run with overlap at $W(t)$. Recall, that in a schematic representation of node v of a DPG a white box pictures the* ALT*-mode and a black box – the* PAR*-mode.*

for all processes. Our algorithms achieving matching upper bounds can even handle variable delays and execution times.

To schedule a task graph $H = (W, F)$ with delay δ we assume that an unbounded number of processors can be used. In each unit-time interval a processor P can execute one single task w. In order to schedule w at time t each direct predecessor of w must have already been executed – either by P itself in previous time intervals or by some other processor by time interval $t-1-\delta$ such that the result of this predecessor can arrive at P on time. Scheduling tasks graphs in the presence of communication delays has first been considered in [6].

Definition 3 *A schedule S for a DPG $\mathcal{G}$ with delay δ is a schedule of a run $H = (W, F)$. Let $T(S)$ denote the duration of S, i.e. the point of time when S has executed all tasks, and* $\boldsymbol{T_{opt}(\mathcal{G}, \delta)} := \min_{S \text{ schedule for } (\mathcal{G},\delta)} T(S)$.

For lower bounds on the complexity of this problem, instead of the optimization version to compute $T_{opt}(\mathcal{G}, \delta)$ it will suffice to consider the decision problem.

Definition 4 DYNAMIC PROCESS GRAPH SCHEDULE (DPGS)
Given a DPG $\mathcal{G}$ with communication delay δ and a deadline T^*, does $T_{opt}(\mathcal{G}, \delta) \leq T^*$ hold?

3 The Structure of Runs

Lemma 1 has shown that the size of a run can be exponential with respect to the size of its underlying DPG. But this does not imply that an optimal parallel execution time of a run requires exponential length, too, because each of its subruns can be small. In this section we investigate the problem what is the *minimal-overlapping* run for a given DPG $\mathcal{G}$. A simple thought shows that if $\mathcal{G} = (V, E, I, O)$ possesses a non-overlapping run then independently of the communication delay one can achieve the bound $T_{opt}(\mathcal{G}, \delta) \leq |V|$.

Lemma 2 *Let $\mathcal{G}$ be a DPG with input mode restriction, either $I \equiv$ ALT, or $I \equiv$ PAR. Then every run of $\mathcal{G}$ is non-overlapping.*

The *proof* of this claim is easy: In case $I \equiv$ ALT each subrun is a simple path from a sink back to a single source. If $I \equiv$ PAR then one can show by a topological induction starting with the sources that no process can have more than one execution instance.

The non-overlapping property does not hold anymore when the input mode may vary – even if one restricts the output mode. In case of $O \equiv$ ALT it is easy to construct a DPG $\mathcal{G}$ such that each run of $\mathcal{G}$ is not non-overlapping. However, we can show:

Lemma 3 *Let $\mathcal{G} = (V, E, I, O)$ be a DPG with κ sources and $O \equiv$ ALT. Then $\mathcal{G}$ has a $(\kappa-1)$-overlapping run.*

This claim can be shown by a flow-argument. The requirement for the sources to have exactly one execution instance generates an initial flow of size κ from the sources to their direct successors. The ALT output mode implies that the flow does not increase. On the contrary, an internal node v with input mode PAR will decrease the flow by $|\mathrm{pred}(v)| - 1$. The flow leaving a node corresponds exactly to the number of execution instances this process requires.

Thus, in case of ALT output mode runs have a quadratic increase at most. In the unrestricted case, this property does not hold anymore as the following result shows.

Lemma 4 *There exists a family $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \ldots$ of DPGs $\mathcal{G}_k = (V_k, E_k, I_k, O_k)$ with $|V_k| = 2k+1$ such that every run H_k of $\mathcal{G}_k$ has a unique subrun $\mathcal{R}$ – which is $(2^{(|V_k|-1)/2-1} - 1)$-overlapping.*

The DPGs are constructed as follows. $\mathcal{G}_k$ consists of $2k+1$ nodes $V_k := \{ v_i \mid 1 \leq i \leq 2k+1 \}$ such that the first $k+1$ nodes form a complete DAG as well as the last $k+1$ nodes:

$$E_k := \{ (v_i, v_j) \mid 1 \leq i < j \leq k+1 \} \cup \{ (v_i, v_j) \mid k+1 \leq i < j \leq 2k+1 \}.$$

Furthermore, define the modes as follows: $I(v_i)$ is ALT for $i \leq k+1$ and PAR for $i > k+1$. Moreover $O(v_i)$ is PAR for $i < k+1$ and ALT otherwise. Now, if the delay δ_k for $\mathcal{G}_k$ is chosen large enough – $\delta_k = |H_k| = 3 \cdot 2^{k-1}$ suffices – then it does not pay to utilize more than 1 processor, since H_k has only one sink.

Corollary 1 *For the family $\mathcal{G}_1, \mathcal{G}_2, \ldots$ defined above it holds $T_{opt}(\mathcal{G}_k, \delta) \geq 3 \cdot 2^{(|V|-1)/2-1}$ if the communication delay is sufficiently large.*

Now, let us investigate the most interesting restricted case, namely DPGs with $O \equiv$ PAR, in the following called **PAR-output DPGs**. As one can see in Fig. 1 such DPGs may have overlapping as well as non-overlapping runs. Hence, a natural question is whether for any executable PAR-output DPG $\mathcal{G}$ one can always find a non-overlapping run. The example in Fig. 2 gives a negative answer to this question. However, the situation cannot be too bad, we can show an exponential improvement of the trivial upper bound.

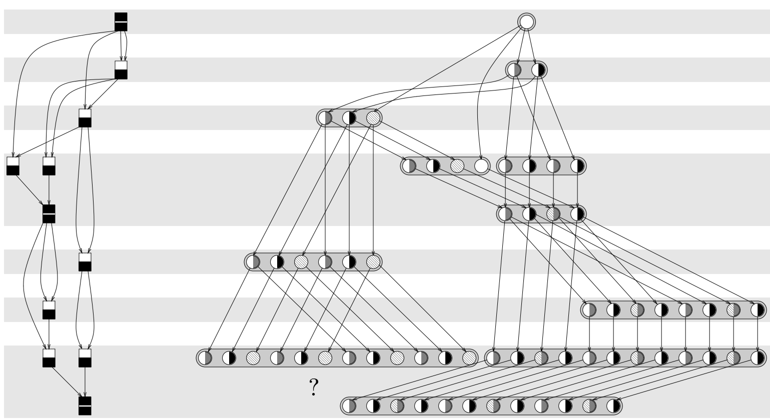

Figure 2: An executable PAR-output DPG (left) with no non-overlapping run.

Theorem 1 *Every executable PAR-output DPG $\mathcal{G} = (V, E, I, O)$ has a k-overlapping run where $k := \lfloor \log_2 \max_{v \in V} |W(v)| \rfloor$.*

A k-overlapping run can be executed in parallel: for every subrun a unique processor completely processes all of its execution instances. This requires at most $|V| \cdot (k+1)$ many steps. Together with Lemma 1 we thus get

Corollary 2 *If a PAR-output DPG $\mathcal{G} = (V, E, I, O)$ can be executed at all it can be executed in time $O(|V|^2)$.*

Below we sketch the idea how to prove Theorem 1. Assume that $\mathcal{G}$ is an executable PAR-output DPG with nodes $V = \{v_1, \ldots, v_n\}$. For a run $H_\mathcal{G}$ of $\mathcal{G}$ define the **characteristic vector** as the sequence $(|W(v_1)|, |W(v_2)|, \ldots, |W(v_n)|)$ and $\boldsymbol{\gamma(\mathcal{G})} := \lfloor \log_2 \max_{v \in V} |W(v)| \rfloor$. Note that on any path $v_{i_1}, \ldots, v_{i_d}$ from a source to a sink the sequence of multiplicities $|W(v_{i_j})|$ must be increasing. Furthermore, we have proven that for PAR-output DPGs $\mathcal{G}$ all runs have exactly the same characteristic vector [3], thus $\gamma(\mathcal{G})$ is independent of a particular run and hence well defined. We will use five steps to transform $\mathcal{G}$ into a graph $\tilde{\mathcal{G}}$

$$\mathcal{G} \xrightarrow{\text{(A)}} \boldsymbol{\mathcal{G}_A} \xrightarrow{\text{(B)}} \boldsymbol{\mathcal{G}_B} \xrightarrow{\text{(C)}} \boldsymbol{\mathcal{G}_C} \xrightarrow{\text{(D)}} \boldsymbol{\mathcal{G}_D} \xrightarrow{\text{(E)}} \boldsymbol{\mathcal{G}_E} = \tilde{\mathcal{G}}$$

in such a way that the intermediate DPGs fulfill the following increasing list of properties (that means $\mathcal{G}_A$ fulfills (A), then $\mathcal{G}_B$ fulfills (A) and (B), etc):

(A) every node with input mode ALT has indegree exactly two;

(B) every node with input mode PAR has only direct predecessors and successors with input mode ALT;

(C) the sources and sinks have input mode PAR: on every path from a source to a sink the input mode alternates between PAR and ALT;

(D) for every value $k \le \overline{k} := \max_{v \in \mathcal{G}} |W(v)|$ there exists a unique PAR-input node $\boldsymbol{u_k}$ with $|W(u_k)| = k$;

(E) for all triples (i, j, k) with $i+j = k \le \overline{k}$ considering the unique nodes u_i, u_j, u_k according to (D), there exists a unique node $v_{i,j,k}$ in $\mathcal{G}_E$ with input mode ALT such that $(u_i, v_{i,j,k}), (u_j, v_{i,j,k}), (v_{i,j,k}, u_k)$ are edges in $\mathcal{G}_E$.

Property (D) and the monotonicity of the multiplicities $|W(u_k)|$ imply that the sink is unique and achieves maximal multiplicity $\overline{k}$.

These transformations expand as well as shrink subgraphs of $\mathcal{G}$ in order to achieve this unique layered structure. Although the size of $\tilde{\mathcal{G}}$ can be exponential in the size of $\mathcal{G}$, we can guarantee that the γ-values do not change, in particular $\gamma(\mathcal{G}) = \gamma(\tilde{\mathcal{G}})$. Furthermore, the property that $\gamma(\mathcal{G})$-overlapping runs exist is preserved as well. We call a DPG fulfilling property (A) to (E) a **complete** PAR-output DPG $\mathcal{C}_k$, where k denotes the multiplicity of the sink of $\mathcal{C}_k$.

Lemma 5 *Every complete* PAR*-output DPG $\mathcal{C}_k$ has a $\gamma(\mathcal{C}_k)$-overlapping run.*

One might be temptered to guess that a run with minimal overlap serves as the basis for a schedule of minimal length. But this is not correct. Overlapping may be crucial to design optimal executions of DPGs as it is shown by the following result. A proof can be found in the full paper, where we construct a family of DPGs exhibiting this behaviour.

Theorem 2 *There exist* PAR*-output DPGs $\mathcal{G}$ with an appropriate communication delay δ such that no non-overlapping run of $\mathcal{G}$ can achieve the minimal schedule length $T_{\text{opt}}(\mathcal{G}, \delta)$.*

4 The Complexity of Scheduling

The previous section has shown a linear bound on the maximal execution time for PAR-output DPGs. On the other hand, execution times of unrestricted DPGs may be exponential. In this section we prove that the computational complexity to determine the minimal execution time for PAR-output DPGs is still very difficult, namely the scheduling problem for such DPGs is $\mathcal{NEXPTIME}$-complete as in the general case.

The $\mathcal{NEXPTIME}$-completeness of the scheduling problem in the unrestricted case has been proved in [3]. Furthermore, we have shown that for any variant with input or output mode restrictions, except the case of PAR-output DPGs, the computational complexity of the scheduling problem decreases drastically, namely it can be solved in $\mathcal{NP}$, in some cases even with less effort. These upper bounds have been matched with corresponding lower bounds.

For scheduling PAR-output DPGs we were able to prove $\mathcal{BH}_2$-hardness. This means that the scheduling problem is probably more difficult for PAR-output DPGs than for the other restrictions. In this section we improve the lower bound showing that the problem is $\mathcal{NEXPTIME}$-hard. Since an optimal schedule of PAR-output DPGs can always be found in $\mathcal{NEXPTIME}$ as in the unrestricted case the lower bound is best possible. The scheduling problem is thus $\mathcal{NEXPTIME}$-complete.

Theorem 3 *Scheduling* PAR*-output DPGs with constant communication delay is* $\mathcal{NEXPTIME}$*-hard.*

The hardness result is obtained by showing that PAR-output DPGs can represent certain Boolean formulas in a highly succinct way. A sketch of this construction will be given in the rest of this section. For the reduction we will use the following $\mathcal{NEXPTIME}$-complete problem [5]:

Definition 5 SUCCINCT-3SAT: *As input we are given a Boolean circuit over the standard AND, OR, NOT-basis that succinctly codes a Boolean formula in conjunctive normal form with the additional property that each clause has exactly three literals and each literal appears exactly three times. Suppose that the encoded formula consists of n variables and m clauses. On input* $(0, i, k)$ *with* $i \in \{0, \dots, n-1\}$ *and* $k \in \{1, 2, 3\}$ *(appropriately coded in binary), the coding circuit returns the index of the clause where the literal* $\neg x_i$ *appears the k-th time. On input* $(1, i, k)$ *it returns the index of the clause where* x_i *appears for the k-th time. On input* $(2, j, k)$ *with* $j \in \{0, \dots, m-1\}$ *and* $k \in \{1, 2, 3\}$*, it returns the k-th literal of the j-th clause. The problem is to decide whether the encoded formula is satisfiable.*

4.1 Boolean Circuits and Dynamic Process Graphs

In [3] the first step of the $\mathcal{NEXPTIME}$-hardness proof for the general case made a transformation of a given Boolean circuit B into a DPG $\mathcal{G}_B$ such that runs of $\mathcal{G}_B$ simulate computations of B on arbitrary input assignments. To encode an arbitrary Boolean circuit B by a DPG $\mathcal{G}_B$ we code each gate of B separately by its own subgraph and draw edges from the output nodes of each gate to the appropriate input nodes (for details of this construction see [3]). The simulation of a gate-subgraph is based on testing the different possible input constellations. For this simulation the ALT-output mode is essential and it cannot be simulated using only the PAR-output mode.

In this subsection we describe a different technique to *transform* a Boolean circuit into a DPG that requires only nodes with output mode PAR. Let B have n input variables. To construct a PAR-output DPG $\mathcal{G}_B$ we proceed in two steps:

1. Construct an equivalent circuit B' that contains only AND and OR gates (of fanin two), NOT gates are superfluous since B' receives as input also the negation of each variable. B' computes the same function as B and has the additional properties:
   (a) the lengths of all paths from an input to an output are equal,
   (b) on every input output path the AND and OR gates alternate.
2. From B' generate $\boldsymbol{\mathcal{G}_B}$ as follows: Let $G = (V, E)$ be the graph of B', then $\mathcal{G}_B := (V, E, I, O)$ with $O \equiv$ PAR. For $u \in V$ we choose $I(u) :=$ ALT iff the corresponding gate of u in B' is an OR -gate, else $I(u) :=$ PAR.

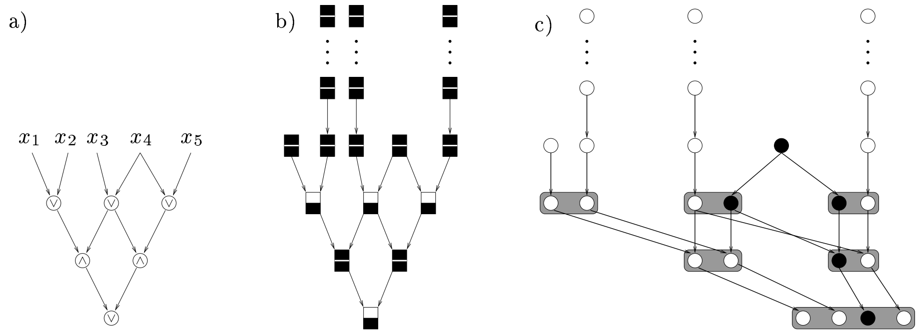

Figure 3: *A Boolean circuit* B*, its DPG* $\mathcal{G}_{B,\alpha}$*, and a run for an input* $x_1 = x_4 = 1$ *and* $x_2 = x_3 = x_5 = 0$*.*

One can show that $\mathcal{G}_B$ is executable and that for any communication delay δ it holds: $T_{opt}(\mathcal{G}_B, \delta) \leq |B'|$.

To understand the functionality of $\mathcal{G}_B$ we will now modify $\mathcal{G}_B$, such that the run of the resulting DPG simulates a computation of B on an arbitrary input vector $\alpha \in \{0,1\}^n$.

Define the line DPG $\mathcal{L}$ of length $k \geq 1$ as a DPG consisting of k vertices $v_1, v_2, \ldots, v_k$ and $k-1$ edges: $(v_1, v_2), \ldots, (v_{k-1}, v_k)$. The input and output modes are inessential. For completeness we set $I(v_i) = O(v_i) = \mathsf{PAR}$. Now let ℓ denote the number of nodes of $\mathcal{G}_B$ and let $k := 2\ell$. We add n lines $\mathcal{L}_1, \ldots, \mathcal{L}_n$ of length k to $\mathcal{G}_B$ such that $\mathcal{L}_i$ corresponds to the variable x_i as follows: If $x_i = 1$ then we connect the sink node $v_{k,i}$ of $\mathcal{L}_i$ to the input node of $\mathcal{G}_B$ that corresponds to the literal $\overline{x}_i$ in B'. If $x_i = 0$ we connect $v_{k,i}$ to the input node of $\mathcal{G}_B$ that corresponds to the literal x_i in B'. The resulting DPG will be denoted by $\boldsymbol{\mathcal{G}_{B,\alpha}}$. An example for such a DPG $\mathcal{G}_{B,\alpha}$ is shown in Fig. 3(b).

Lemma 6 *Let $y_1, \ldots, y_m$ be the output nodes of circuit B, let α be an input assignmet, and let $\beta \in \{0,1\}^m$ be the output vector of B for input α. Then for every run H of $\mathcal{G}_{B,\alpha}$ and for any $i \in [1..m]$ the following holds:*

$$\beta(i) = 0 \;\Rightarrow\; \forall w \in W(y_i): \text{ the subrun induced by } w \text{ has size} > 2\ell.$$

Moreover there exists such a run H_α for $\mathcal{G}_{B,\alpha}$ that for any sequence $i_1 < i_2 < \ldots < i_k$, with $\beta(i_j) = 1$, one can chose tasks $w_{i_j} \in W(y_{i_j})$ in H_α such that the subgraph induced by the tasks w_{i_j} and their predecessors has size bounded by ℓ, i.e. $|\bigcup_{1 \leq j \leq k} \mathcal{R}(w_{i_k})| \;\leq\; \ell$.

Hence to test some output gates $y_{i_1}, \ldots, y_{i_k}$ of B for a particular input assignment α one can check for the run H_α whether for some tasks of $y_{i_1}, \ldots, y_{i_k}$ the corresponding subgraph has size at most ℓ.

4.2 The Reduction

Let B be an input of the SUCCINCT-3SAT problem, that means a circuit coding a Boolean 3CNF formula $\mathcal{F}$. We may assume that $\mathcal{F}$ has exactly $m = 2^d$ clauses. Then there will be $n = 2^{d-1}$ variables.

The first step of our reduction is to construct two circuits A_1 and A_2. The first circuit A_1 verifies the syntax of the formula encoded by B. Speaking more formally, A_1 with the input j returns 1 iff the given circuit B returns j on exactly three inputs $\boldsymbol{(k,\ B(2,j,k'),\ k'')}$ for all $k \in \{0,1\}$ and $k', k'' \in \{1,2,3\}$. Otherwise A_1 returns 0.

The second circuit A_2 will be used to check whether the encoded formula is satisfiable. A_2 with input $\boldsymbol{(k,j,i)}$, is defined as follows. $A_2(k,j,i)$ returns $\boldsymbol{(1,\beta)}$ if x_i^β is the k-th literal of the j-th clause, where x_i^0 denotes $\neg x_i$ and x_i^1 means x_i. In any other case, A_2 returns $\boldsymbol{(0,0)}$. Obviously, these circuits can easily be constructed for a given B. Let us denote the DPG that corresponds to circuit A_i by $\mathcal{A}_i$. We will call $\mathcal{A}_1$ the **syntax-verifier**, and $\mathcal{A}_2$ the **index-verifier** graph. Because of lack of space only the syntax-verifier will be described in the following.

Let us call the sources of $\mathcal{A}_1$ $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \mathbf{j_{t2}}, \mathbf{j_{f2}}, \ldots, \mathbf{j_{td}}, \mathbf{j_{fd}}$. They encode the integer j as follows: If the ℓ-th digit of the binary representation of j is 1 then in a run for $\mathcal{A}_1$ that represents j the node $\mathbf{j_{t\ell}}$ has an execution instance that can be computed within a certain time, while the execution instance of $\mathbf{j_{f\ell}}$ requires slightly more time. If the ℓ-th digit of j is 0 then $\mathbf{j_{f\ell}}$ can be computed fast and $\mathbf{j_{t\ell}}$ requires the larger amount of time. The unique instance of either $\mathbf{j_{t\ell}}$ or $\mathbf{j_{f\ell}}$ that can be computed fast will be denoted by $t(\ell)$.

Our first goal is to construct a DPG that represents the input values of j. More precisely, the run of this DPG represents all possible 2^d values for j. The crucial point of the whole construction is then to couple this value genarator DPG with $\mathcal{A}_1$ in an appropriate way.

To generate the binary representation of j we will use a DPG called a **counting graph** $\boldsymbol{\mathcal{TC}_{k,\ell_1,\ell_2}}$ with parameters k, ℓ_1 and ℓ_2. Their main building blocks are subgraphs $\mathcal{C}_\ell := (V,E,I,O)$ with $V = \{u_i, v_{0,i}, v_{1,i} |\ 1 \leq i \leq \ell\} \cup \{u_0\}$, $E := \{(u_{i-1}, v_{j,i}), (v_{j,i}, u_i) | 1 \leq i \leq \ell,\ j \in \{0,1\}\}$, $O \equiv \mathsf{PAR}$, and $I \equiv \mathsf{ALT}$. Note that for each $\mathcal{C}_\ell$ if considered separately the sink process u_ℓ has 2^ℓ execution nodes.

Using these subgraphs one can achieve the following property which will be important for the reduction. $\mathcal{TC}_{k,\ell_1,\ell_2}$ has $2k$ sink nodes, which will be denoted by $y_1^0, y_1^1, \ldots, y_k^0, y_k^1$. For every run of $\mathcal{TC}_{k,\ell_1,\ell_2}$ it holds that each y_i^j has exactly 2^k execution tasks and for exactly 2^{k-1} of these tasks each corresponding subrun has size $S_t := k(\ell_1 + 2) + k^2 + 4$. For the remaining 2^{k-1} tasks each subrun has size $S_f := S_t + (k-1) \cdot (\ell_2 + 1) + 2$. This means that for the suitable values ℓ_1 and ℓ_2, exactly 2^{k-1} subruns have small size whereas the remaining subruns are much bigger. Furthermore, any task t_i^β of a node y_i^β is associated with a task $t_i^{1-\beta}$ of a node $y_i^{1-\beta}$ such that $|\mathcal{R}(t_i^\beta)| = S_t$ iff $|\mathcal{R}(t_i^{1-\beta})| = S_f$. Finally, the construction guarantees that any processor can only compute one task for any pair y_i^0, y_i^1.

Drawing edges $(y_i^0, j_{fi}), (y_i^1, j_{t1})$ from the sinks of the counting graph to the sources of $\mathcal{A}_1$ we obtain a DPG which has similar properties as the DPG $\mathcal{G}_{B,\alpha}$ discussed in the previuous subsection. Note that the pair y_i^0, y_i^1 of the counting graph specifies the value of $t(i)$. Using Lemma 6 one can prove that for any input j the circuit A_1 returns the value 1 iff for any value j (represented by the particular tasks t_i^0, t_i^1) there exists a subrun for the sink of $\mathcal{A}_1$ of moderate size, i.e. $\leq k(\ell_1+2) + k(k^2+4) + |\mathcal{A}_1|$.

The DPG for checking whether the encoded formula is satisfiable is technically more involved. This graph has to check whether the input values i, j, and k fit together (which is something like a local test). Furthermore, it requires a mechanism to choose the value of the variables x_i and decide if these values satisfy each clause C_j (which is something like a global test). The details can be found in the full paper.

5 Conclusion

We have shown that the computational complexity to determine the optimal schedule length T_{opt} for PAR-output DPGs is quite high, this problem is complete for $\mathcal{NEXPTIME}$. On the other hand, we have shown that the maximal size of a subrun of a minimal-overlapping run H gives a good approximation for T_{opt}. However, this bound is not tight and an interesting problem is to give some other *natural* structural properties of runs which allow better approximations.

Another interesting question is how efficiently a minimal-overlapping run can be constructed for a given PAR-output DPG $\mathcal{G}$. In this paper we have shown that all such runs have the same size. Can we find a minimal-overlapping run H in linear time with respect to the size of H? Note that Lemma 1 and Lemma 2 in [3] imply that for certain DPGs the number of different runs can be exponential with respect to the size of H. Therefore, a brute-force searching will be inefficient.

References

[1] H. Galperin, A. Wigderson, Succinct Representations of Graphs, Information and Control 56, 1983, 183-198.

[2] S. Ha, E. Lee, *Compile-time Scheduling and Assignment of Data-flow Program Graphs with Data-dependent Iteration*, IEEE Trans. Computers 40, 1991, 1225-1238.

[3] A. Jakoby, M. Liśkiewicz, R. Reischuk, *Scheduling Dynamic Graphs,* in Proc. 16th STACS'99, LNCS 1563, Springer-Verlag, 1999, 383-392; for a complete version see TR A-00-02, Universität Lübeck, 2000.

[4] T. Lengauer, K. Wagner, *The Correlation between the Complexities of the Nonhierarchical and Hierarchical Versions of Graph Problems,* J. CSS 44, 1992, 63-93.

[5] C. Papadimitriou and M. Yannakakis, *A Note on Succinct Representations of Graphs*, Information and Control 71, 1986, 181-185.

[6] C. Papadimitriou and M. Yannakakis, *Towards an Architecture-Independent Analysis of Parallel Algorithms,* Proc. 20. STOC, 1988, 510-513, see also SIAM J. Comput. 19, 1990, 322-328.