A Map-Reduce Parallel Approach to Automatic Synthesis of Control Software

Vadim Alimguzhin[1,2], Federico Mari[1], Igor Melatti[1], Ivano Salvo[1], and Enrico Tronci[1]

[1] Computer Science Department, Sapienza University of Rome, Italy,
{alimguzhin,mari,melatti,salvo,tronci}@di.uniroma1.it.
[2] Department of Computer Science and Robotics, Ufa State Aviation Technical University, Russian Federation.

Abstract. Many Control Systems are indeed Software Based Control Systems, i.e. control systems whose controller consists of control software running on a microcontroller device. This motivates investigation on Formal Model Based Design approaches for automatic synthesis of control software.
Available algorithms and tools (e.g., *QKS*) may require weeks or even months of computation to synthesize control software for large-size systems. This motivates search for parallel algorithms for control software synthesis.
In this paper, we present a Map-Reduce style parallel algorithm for control software synthesis when the controlled system (*plant*) is modeled as a discrete time linear hybrid system. Furthermore we present an MPI-based implementation *PQKS* of our algorithm. To the best of our knowledge, this is the first parallel approach for control software synthesis.
We experimentally show effectiveness of *PQKS* on two classical control synthesis problems: the inverted pendulum and the multi-input buck DC/DC converter. Experiments show that *PQKS* efficiency is above 60%. As an example, *PQKS* requires about 16 hours to complete the synthesis of control software for the pendulum on a cluster with 60 processors, instead of the 25 days needed by the sequential algorithm implemented in *QKS*.

1 Introduction

Many Embedded Systems are indeed Software Based Control Systems (SBCSs). An SBCS consists of two main subsystems: the controller and the plant. Typically, the plant is a physical system consisting, for example, of mechanical or electrical devices whereas the controller consists of control software running on a microcontroller. In an endless loop, at discrete time instants (*sampling*), the controller reads plant sensor outputs from the plant and computes commands to be sent back to plant actuators. Being the control software discrete and the physical system typically continuous, sensor outputs go through an Analog-to-Digital (AD) conversion (*quantization*) before being read from the control software. Analogously, controller commands need a Digital-to-Analog (DA) conversion before being sent to plant actuators. The controller selects commands in order to guarantee that the closed-loop system (that is, the system consisting of both plant and controller) meets given safety and liveness specifications (System Level Formal Specifications).

Software generation from models and formal specifications forms the core of Model Based Design of embedded software [1]. This approach is particularly interesting for SBCSs since in such a case system level (formal) specifications are much easier to define than the control software behavior itself.

1.1 Motivations

In this paper we focus on the algorithm presented in [2,3,4], which returns correct-by-construction control software starting from system level formal specifications. This algorithm is implemented in *QKS* (*Quantized Kontroller Synthesizer*), which takes as input: i) a formal model of the controlled system, modeled as a Discrete Time Linear Hybrid System (DTLHS), ii) safety and liveness requirements (goal region) and iii) b and b_u as the number of bits for, respectively, AD and DA conversions. Given this, *QKS* outputs a correct-by-construction control software together with the controlled region on which the software is guaranteed to work.

To this aim, *QKS* first computes a suitable finite state abstraction (*control abstraction* [4]) $\hat{\mathcal{H}}$ of the DTLHS plant model $\mathcal{H}$, where $\hat{\mathcal{H}}$ depends on the quantization schema (i.e. number of bits b, b_u needed for AD/DA conversions) and it is the plant as it can be seen from the control software after AD conversion and before DA conversion. Then, given an abstraction $\hat{G}$ of the goal states G, it is computed a controller $\hat{K}$ that, starting from any initial abstract state, drives $\hat{\mathcal{H}}$ to $\hat{G}$ regardless of possible nondeterminism. Control abstraction properties ensure that $\hat{K}$ is indeed a (quantized representation of a) controller for the original plant $\mathcal{H}$. Finally, $\hat{K}$ is translated into control software (C code).

While effective on moderate-size systems, *QKS* requires a huge amount of computational resources when applied to larger systems. In fact, the most critical step of *QKS* is the control abstraction $\hat{\mathcal{H}}$ generation (which is responsible for more than 95% of the overall computation, see [3]). This stems from the fact that $\hat{\mathcal{H}}$ is computed explicitly, by solving a Mixed Integer Linear Programming (MILP) problem for each triple $(\hat{x}, \hat{u}, \hat{x}')$, where $\hat{x}, \hat{x}'$ are abstract states of $\hat{\mathcal{H}}$ and $\hat{u}$ is an abstract action of $\hat{\mathcal{H}}$. Thus *QKS* is based on an *hybrid* approach, being both *explicit* in the abstract state space enumeration and *symbolic* in the usage of MILP solvers. Since the number of abstract states is 2^b, being b the number of bits needed for AD conversion of all variables describing the plant, and since the number of abstract actions is 2^{b_u}, we have that *QKS* computation time is exponential in $2b + b_u$. In *QKS*, suitable optimizations reduce the complexity to be exponential in $b + b_u$, and thus in b since $b_u << b$. However, in large-size systems b may be large for two typical reasons. First, since each plant state variable needs to be quantized (if a state variable v is discrete, then the number of bits for v is not an input, since $\lfloor \log_2 |\mathrm{dom}(v)| \rfloor + 1$ bits are needed), the number of bits is necessarily high when the plant model consists of many variables. As an example, the plane collision avoidance control system in [5] is described by 4 continuous variables and 7 discrete variables. Second, controllers synthesized by considering a finer quantization schema (i.e., with an higher value of b) usually have a better behavior with respect to non-functional requirements, such as *ripple* and *set-up time*. Therefore, when a high precision is required, a large number of quantization bits must be considered.

As an example, experimental results show that *QKS* takes nearly one month (25 days) of CPU time to synthesize the controller for a 26 bits quantized inverted pendulum (which is described by only two continuous state variables, see Sect. 5.1). Moreover, 99% of those 25 days of computation is due to control abstraction generation. This may result in a loss in terms of time-to-market in control software design when *QKS* is used.

This motivates search of parallel versions of *QKS* synthesis algorithm.

1.2 Main Contributions

To overcome the computation time bottleneck in *QKS*, we present a *Map-Reduce* style parallel algorithm for control abstraction generation in control software synthesis.

Map-Reduce [6] is a (LISP inspired) programming paradigm advocating a form of embarrassing parallelism for effective massive parallel processing. An implementation of such an approach is in Hadoop (e.g., see [7]). The effectiveness of the Map-Reduce approach stems from the minimal communication overhead of embarrassing parallelism. This motivates our goal of looking for a Map-Reduce style parallel algorithm for control software synthesis from system level formal specifications.

To this aim, we design a parallel version of *QKS*, that is inspired to the Map-Reduce programming style and that we call *Parallel QKS* (*PQKS* in the following). *PQKS* is actually implemented using MPI (Message Passing Interface [8]) in order to exploit the computational power available in modern computer clusters (distributed memory model). Such an algorithm will be presented in Sect. 4, after a discussion of the basic notions needed to understand our approach (Sect. 2) and the description of the standalone (i.e. serial) algorithm of *QKS* (Sect. 3).

We show the effectiveness of *PQKS* by using it to synthesize control software for two widely used embedded systems, namely the multi-input buck DC-DC converter [9] and the inverted pendulum [10] benchmarks. These are challenging examples for the automatic synthesis of correct-by-construction control software. Experimental results on the above described benchmarks will be discussed in Sect. 5. Such results show that we achieve a nearly linear speedup w.r.t. *QKS*, with efficiency above 60%. As an example, *PQKS* requires about 16 hours to complete the above mentioned synthesis of the 26-bits pendulum on a cluster with 60 processors, instead of the 25 days of *QKS*.

2 Background on Control Abstraction for DTLHSs

To make this paper self-contained, in this section we briefly summarize the notions necessary to understand our parallel approach to control software synthesis. For more details, we refer the reader to [4].

Guarded Constraints We denote with $[n]$ an initial segment $\{1, \ldots, n\}$ of the natural numbers. We denote with $X = [x_1, \ldots, x_n]$ a finite sequence of variables that we may regard, when convenient, as a set. Each variable x ranges on a known (bounded or unbounded) interval $\mathcal{D}_x$ either of the reals (continuous variables) or of the integers (discrete variables). We denote with $\mathcal{D}_X$ the set $\prod_{x \in X} \mathcal{D}_x$. Boolean variables are discrete variables ranging on the set $\mathbb{B} = \{0, 1\}$. If x is a boolean variable, we write $\bar{x}$

for $(1 - x)$. A *linear expression* over a list of variables X is a linear combination of variables in X with rational coefficients. A *linear constraint* over X (or simply a *constraint*) is an expression of the form $L(X) \leq b$, where $L(X)$ is a linear expression over X and b is a rational constant. Given a constraint $C(X)$ and a fresh boolean variable (*guard*) $y \notin X$, a *guarded constraint* has either the form $y \rightarrow C(X)$ (if y then $C(X)$) or $\bar{y} \rightarrow C(X)$ (if not y then $C(X)$). A *guarded predicate* is a conjunction of either constraints or guarded constraints.

Labeled Transition Systems A *Labeled Transition System* (LTS) is a tuple $\mathcal{S} = (S, \mathcal{A}, T)$ where S is a (possibly infinite) set of states, $\mathcal{A}$ is a (possibly infinite) set of *actions*, and $T : S \times \mathcal{A} \times S \rightarrow \mathbb{B}$ is the *transition relation* of $\mathcal{S}$. Let $s \in S$ and $a \in \mathcal{A}$. We call *self loop* a transition of the form (s, a, s). A *run* or *path* for an LTS $\mathcal{S}$ is a sequence $\pi = s_0, a_0, s_1, a_1, s_2, a_2, \ldots$ of states s_t and actions a_t such that $\forall t \geq 0$ $T(s_t, a_t, s_{t+1})$. The length $|\pi|$ of a finite run π is the number of actions in π.

Discrete Time Linear Hybrid Systems A *Discrete Time Linear Hybrid System* is a tuple $\mathcal{H} = (X, U, Y, N)$ where:

- $X = X^r \cup X^d$ is a finite sequence of real (X^r) and discrete (X^d) *present state* variables. We denote with X' the sequence of *next state* variables obtained by decorating with $'$ all variables in X.
- $U = U^r \cup U^d$ is a finite sequence of *input* variables.
- $Y = Y^r \cup Y^d$ is a finite sequence of *auxiliary* variables that are typically used to model *modes* (e.g., from switching elements such as diodes) or "local" variables.
- $N(X, U, Y, X')$ is a guarded predicate over $X \cup U \cup Y \cup X'$ defining the *transition relation* (*next state*).

The semantics of a DTLHS $\mathcal{H}$ is an LTS $\text{LTS}(\mathcal{H}) = (\mathcal{D}_X, \mathcal{D}_U, \tilde{N})$ where $\tilde{N} : \mathcal{D}_X \times \mathcal{D}_U \times \mathcal{D}_X \rightarrow \mathbb{B}$ is a function s.t. $\tilde{N}(x, u, x') \equiv \exists\, y \in \mathcal{D}_Y\ N(x, u, y, x')$.

Quantizations for DTLHSs A *quantization function* γ for a real interval $I = [a, b]$ is a non-decreasing function $\gamma : I \mapsto \mathbb{Z}$ s.t. $\gamma(I)$ is a bounded integer interval. In the following we will only consider quantization functions γ s.t.: i) $\gamma(I) = \{0, \ldots, 2^b - 1\}$ for some $b \in \mathbb{N}$ (number of bits); ii) γ divides the interval $[a, b]$ into 2^b equal subintervals, so that $\gamma(x) = i - 1$ iff x is in the i-th subinterval. Thus we will specify quantizations by only defining the number of bits b. Finally, if I is a discrete set $I \subseteq \mathbb{Z}$, then $\gamma(x) = x - \min I$.

Let $\mathcal{H} = (X, U, Y, N)$ be a DTLHS, and $W = X \cup U \cup Y$. A *quantization* $\mathcal{Q}$ for $\mathcal{H}$ is a pair (A, Γ), where:

- A explicitly bounds each variable in W (i.e., $A = \bigwedge_{w \in W} \alpha_w \leq w \leq \beta_w$, with $\alpha_w, \beta_w \in \mathcal{D}_W$). For each $w \in W$, we denote with $A_w = [\alpha_w, \beta_w]$ its *admissible region* and with $A_W = \prod_{w \in W} A_w$.
- Γ is a set of maps $\Gamma = \{\gamma_w \mid w \in W \text{ and } \gamma_w \text{ is a quantization function for } A_w\}$.

Let $W = [w_1, \ldots, w_k]$ and $v = [v_1, \ldots, v_k] \in A_V$, with $V \subseteq W$. We write $\Gamma(v)$ for the tuple $[\gamma_{w_1}(v_1), \ldots, \gamma_{w_k}(v_k)]$, $\Gamma^{-1}(\hat{v})$ for the set $\{v \in A_V \mid \Gamma(v) = \hat{v}\}$, and $\Gamma(A_W) = \{\Gamma(v) \mid v \in A_W\}$. Finally, we call *abstract states (resp., actions)* the elements in the finite set $\Gamma(A_X)$ (resp., $\Gamma(A_U)$).

3 Control Abstraction Computation

As explained in Sect. 1.1, the heaviest computation step for *QKS* is the computation of the control abstraction. In this section, we recall the definition of control abstraction, as well as how it is computed by *QKS*.

In the following, let $\mathcal{H} = (X, U, Y, N)$ and $\mathcal{Q} = (A, \Gamma)$ be, respectively, a DTLHS and a quantization for $\mathcal{H}$. We say that an abstract action $\hat{u} \in \Gamma(A_U)$ is $\mathcal{Q}$-admissible in an abstract state $\hat{x} \in \Gamma(A_X)$ iff actions in $\hat{u}$ always maintain the plant inside its admissible region when starting from states in $\hat{x}$ (i.e., for all plant states $x \in \Gamma^{-1}(\hat{x})$, plant actions $u \in \Gamma^{-1}(\hat{u})$, and plant states x', if (x, u, x') is a transition in LTS$(\mathcal{H})$ then $x' \in A_X$).

Definition 1. *The $\mathcal{Q}$ control abstraction of a DTLHS $\mathcal{H}$ is an LTS $\hat{\mathcal{H}} = (\Gamma(A_X), \Gamma(A_U), \hat{N})$, where for $\hat{N}$ the following holds:*

1. *each abstract transition in $\hat{N}$ stems from a concrete transition in N;*
2. *each concrete transition (x, u, x') in N is faithfully represented by an abstract transition $(\Gamma(x), \Gamma(u), \Gamma(x'))$ in $\hat{N}$, provided that $\Gamma(x) \neq \Gamma(x')$ and $\Gamma(u)$ is $\mathcal{Q}$-admissible in $\Gamma(x)$;*
3. *if there is no upper bound to the length of concrete paths in* LTS$(\mathcal{H})$ *s.t. all states are inside the counter-image of an abstract state $\hat{x}$ and all actions are inside the counter-image of an abstract action $\hat{u}$, then there is an abstract self loop $(\hat{x}, \hat{u}, \hat{x})$ in $\hat{N}$.*

Algorithm 1 Building a control abstraction

Input: DTLHS $\mathcal{H} = (X, U, Y, N)$, quantization $\mathcal{Q} = (A, \Gamma)$.
function *ctrAbs* $(\mathcal{H}, \mathcal{Q})$
1. $\hat{N} \leftarrow \varnothing$
2. **for all** $\hat{x} \in \Gamma(A_X)$ **do**
3. $\quad \hat{N} \leftarrow$ *ctrAbsAux*$(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{N})$
4. **return** $(\Gamma(A_X), \Gamma(A_U), \hat{N})$

Given a quantization $\mathcal{Q} = (A, \Gamma)$ for a DTLHS $\mathcal{H} = (X, U, Y, N)$, Function *ctrAbs* in Alg. 1 computes a $\mathcal{Q}$-control abstraction $(\Gamma(A_X), \Gamma(A_U), \hat{N})$ of $\mathcal{H}$ following Def. 1. Namely, the control abstraction transition relation $\hat{N}$ is incrementally computed by starting with the empty relation (line 1) and then adding, for all abstract states $\hat{x}$ (line 2), all transitions which starts from $\hat{x}$ and fulfills Def. 1 (line 3). This is done by calling the auxiliary function *ctrAbsAux*, which is detailed in Alg. 2. Namely, function *ctrAbsAux* checks, for all abstract actions $\hat{u}$ (line 1) and all possible next abstract states $\hat{x}' \in \mathcal{O}$ (line 5), if $(\hat{x}, \hat{u}, \hat{x}')$ may be added to the current $\hat{N}$. Self loops are

separately handled in line 3. Note that the checks in lines 2, 3 and 6, and the computation in line 4 are performed by properly defining MILP problems, which are solved using known algorithms (available in the GLPK package).

Algorithm 2 Building a control abstraction: transitions from a given abstract state

Input: DTLHS $\mathcal{H}$, quantization $\mathcal{Q}$, abstract state $\hat{x}$, partial control abstraction $\hat{N}$.
function *ctrAbsAux* $(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{N})$
1. **for all** $\hat{u} \in \Gamma(A_U)$ **do**
2. **if** $\neg$ $\mathcal{Q}$-*admissible*$(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{u})$ **then**
3. **if** *selfLoop*$(\mathcal{H},\mathcal{Q}, \hat{x},\hat{u})$ **then** $\hat{N} \leftarrow \hat{N} \cup \{(\hat{x}, \hat{u}, \hat{x})\}$
4. $\mathcal{O} \leftarrow$ *overImg*$(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{u})$
5. **for all** $\hat{x}' \in \Gamma(\mathcal{O})$ **do**
6. **if** $\hat{x} \neq \hat{x}' \wedge$ *existsTrans*$(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{u}, \hat{x}')$ **then**
7. $\hat{N} \leftarrow \hat{N} \cup \{(\hat{x}, \hat{u}, \hat{x}')\}$
8. **return** $\hat{N}$

4 Parallel Synthesis of Control Software

In this section we present our novel parallel algorithm for the control abstraction generation of a given DTLHS. Such algorithm is a parallel version of the standalone Alg. 1. In this way we significantly improve the performance on the control abstraction generation (which is the bottleneck of *QKS*), thus obtaining a huge speedup for the whole approach to the synthesis of control software for DTLHSs.

In the following, let $\mathcal{H} = (X, U, Y, N)$, $\mathcal{Q} = (A, \Gamma)$ be, respectively, the DTLHS and the quantization in input to our algorithm for control abstraction generation. Moreover, let b be the overall number of bits needed in $\mathcal{Q}$ to quantize plant states (i.e., $b = \sum_{x \in X} b_x$, where b_x is the number of bits for $\gamma_x \in \Gamma$). Finally, let p be the number of processors available for parallel computation.

Our parallel algorithm rests on the observation that all calls to function *ctrAbsAux* (see Alg. 2) are independent of each other, thus they may be performed by independent processes without communication overhead. This observation allows us to use parallel methods targeting *embarrassingly parallel* problems in order to obtain a significant speedup on the control abstraction generation phase. To this aim, we use a Map-Reduce based parallelization technique to design a parallel version of Alg. 1. Namely, our parallel computation is designed as follows (see Fig. 1 for an example).

1. A *master* process assigns (*maps*) the computations needed for an abstract state $\hat{x}$ (i.e., the execution of a call to function *ctrAbsAux* of Alg. 2) to one of p computing processes (*workers*, enumerated from 1 to p). This is done in a way so that each worker approximately handles $\frac{|\Gamma(A_X)|}{p}$ abstract states, thus balancing the parallel workload. Namely, abstract states are enumerated from 1 to 2^b, and abstract state i is assigned to worker $1 + ((i-1) \bmod p)$. We denote with $\Gamma^{(i,p)}(A_X) \subseteq \Gamma(A_X)$ the set of abstract states mapped to worker i out of p available workers. Note that worker i may locally decide which abstract states are in $\Gamma^{(i,p)}(A_X)$ by only knowing i and p (together with the overall input $\mathcal{H}$ and $\mathcal{Q}$). This allows us to avoid

sending to each worker the explicit list of abstract states it has to work on, since it is sufficient that the master sends i and p (plus $\mathcal{H}$ and $\mathcal{Q}$) to worker i.

2. Each worker *works* on its abstract states partition $\Gamma^{(i,p)}(A_X)$, by calling *ctrAbsAux* for each abstract state in such partition. Once worker i has completed its task (i.e., all abstract states in $\Gamma^{(i,p)}(A_X)$ have been considered), a local (partial) control abstraction $\hat{N}_i$ is obtained, which is sent back to the master.
3. The master collects the local control abstractions coming from the workers and composes (*reduces*) them in order to obtain the desired complete control abstraction for $\mathcal{H}$. Note that, as in embarrassingly parallel tasks, communication only takes place at the beginning and at the end of local computations.

Algorithm 3 Building a control abstraction in parallel: master process

Input: DTLHS $\mathcal{H}$, quantization $\mathcal{Q}$, workers number p
function *ctrAbsMaster* $(\mathcal{H}, \mathcal{Q}, p)$
1. **for all** $i \in \{1, \ldots, p\}$ **do**
2. create a worker and send $\mathcal{H}$, $\mathcal{Q}$, i and p to it
3. wait to get $\hat{N}_1, \ldots, \hat{N}_p$ from workers
4. **return** $(\Gamma(A_X), \Gamma(A_U), \cup_{j=1}^{p} \hat{N}_j)$

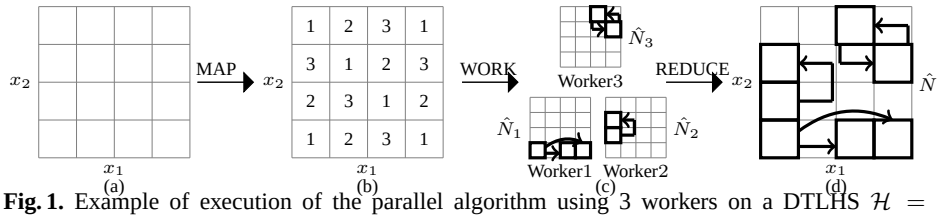

Fig. 1. Example of execution of the parallel algorithm using 3 workers on a DTLHS $\mathcal{H} = (X, U, Y, N)$ and a quantization $\mathcal{Q}$ for $\mathcal{H}$ s.t. $X = [x_1, x_2]$ and $\mathcal{Q}$ discretizes both x_1, x_2 with two bits. In (a) the starting point is shown, where each cell corresponds to an abstract state. In (b), function *ctrAbsMaster* maps the workload among the 3 workers (abstract states labeled with $i \in [3]$ are handled by worker i). In (c) each worker i computes its local control abstraction $\hat{N}_i$, which is assumed to have the shown transitions only. Finally, in (d) the master rejoins the local control abstractions in order to get the final one, i.e. $\hat{N}$.

Our parallel algorithm is described in Algs. 3 (for the master) and 4 (for workers).

4.1 Implementation with MPI

We actually implemented Algs. 3 and 4 in *PQKS* by using MPI (Message Passing Interface, see [8]). Since MPI is widely used, this allows us to run *PQKS* on nearly all computer clusters. Note that in MPI all computing processes execute the same program, each one knowing its rank i and the overall number of computing processes p (Single Program Multiple Data paradigm). Thus lines 1–2 of Alg. 3 are directly implemented by the MPI framework. Moreover, in our implementation the master is not a separate node,

Algorithm 4 Building a control abstraction in parallel: worker processes

Input: DTLHS $\mathcal{H} = (X, U, Y, N)$, quantization $\mathcal{Q} = (A, \Gamma)$, index i, workers number p
function *parCtrAbs* ($\mathcal{H}$, $\mathcal{Q}$, i, p)
1. $\hat{N}_i \leftarrow \varnothing$
2. **for all** $\hat{x} \in \Gamma^{(i,p)}(A_X)$ **do**
3. $\quad \hat{N}_i \leftarrow$ *ctrAbsAux*$(\mathcal{H}, \mathcal{Q}, \hat{x}, \hat{N}_i)$
4. send $\hat{N}_i$ to the master

but it actually performs as worker with id 1 while waiting for local control abstractions from other workers. Local control abstraction from other workers are collected once the master local control abstraction (i.e., $\hat{N}_1$) has been completed. This allows us to use p nodes instead of $p + 1$, as well as to save communication time ($\hat{N}_1$ is already available to the master node, thus it needs not to be sent).

Note that lines 3 and 4 of, respectively, Algs. 3 and 4 require workers to send their local control abstraction to the master. Being control abstractions represented as OBDDs (*Ordered Binary Decision Diagrams* [11]), which are sparse data structures, this step may be inefficient if implemented with a call to MPI_Send (as it is usually done in MPI programs), which is designed for contiguous data. In order to make *PQKS* efficient, MPI_Send is not used. Instead, workers use known algorithms (implemented in the CUDD package) to efficiently dump the OBDD representing their local control abstraction on the shared filesystem. Since current MPI implementations are typically based on a shared filesystem, this is not a limitation for *PQKS*. Then each computing process calls MPI_Barrier, in order to synchronize all workers with the master. After this, the master node collects local control abstraction from workers, by reloading them from the shared filesystem, in order to build the final global one. Consequently, when presenting experimental results in Sect. 5, we include I/O time in communication time. Note that communication based on shared filesystem is very common also in Map-Reduce native implementations like Hadoop [7].

Finally, we note that Algs. 3 and 4 may conceptually be implemented on multi-threaded systems with shared memory. However, in our implementation we use GLPK as external library to solve MILP problems required in computations inside function *ctrAbsAux* (see Alg. 2). Since GLPK is not thread-safe, we may not implement Algs. 3 and 4 on multithreaded shared memory systems.

5 Experimental Results

We implement functions *ctrAbsMaster* and *parCtrAbs* of Algs. 3 and 4 in C programming language using the CUDD package for OBDD based computations and the GLPK package for MILP problems solving, and MPI for the parallel setting and communication. The resulting tool, *PQKS* (*Parallel QKS*), extends the tool *QKS* [3] by replacing function *ctrAbs* of Alg. 1 with function *ctrAbsMaster* of Alg. 3.

In this section we present experimental results obtained by using *PQKS* on two meaningful and challenging examples for the automatic synthesis of correct-by-construction control software, namely the inverted pendulum and multi-input buck DC-

DC converter. In such experiments, we show the gain of the parallel approach with respect to the serial algorithm, also providing standard measures such as communication and I/O time.

This section is organized as follows. In Sects. 5.1 and 5.2 we will present the inverted pendulum and the multi-input buck DC-DC converter, on which our experiments focus. In Sect. 5.3 we give the details of the experimental setting, and finally, in Sect. 5.4, we discuss experimental results.

5.1 The Inverted Pendulum Case Study

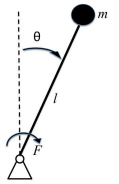

Fig. 2. Inverted Pendulum with Stationary Pivot Point.

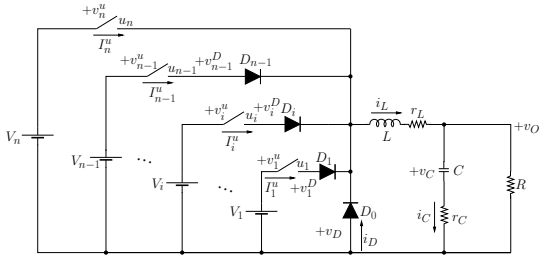

Fig. 3. Multi-input Buck DC-DC converter.

The inverted pendulum [10] (see Fig. 2) is modeled by taking the angle θ and the angular velocity $\dot{\theta}$ as state variables. The input of the system is the torquing force $u \cdot F$, that can influence the velocity in both directions. Here, the variable u models the direction and the constant F models the intensity of the force. Differently from [10], we consider the problem of finding a discrete controller, whose decisions may be only "apply the force clockwise" ($u = 1$), "apply the force counterclockwise" ($u = -1$)", or "do nothing" ($u = 0$). The behavior of the system depends on the pendulum mass m, the length of the pendulum l, and the gravitational acceleration g. Given such parameters, the motion of the system is described by the differential equation $\ddot{\theta} = \frac{g}{l}\sin\theta + \frac{1}{ml^2}uF$, which may be normalized and discretized in the following transition relation (being T the sampling time constant, $x_1 = \theta$ and $x_2 = \dot{\theta}$): $N(x_1, x_2, u, x_1', x_2') \equiv (x_1' = x_1 + Tx_2) \wedge (x_2' = x_2 + T\frac{g}{l}\sin x_1 + T\frac{1}{ml^2}uF)$. Such transition relation is not linear, as it contains the function $\sin x_1$. A linear model can be found by under- and over-approximating the non-linear function $\sin x$ on different intervals for x. Namely, we may proceed as follows [12]. First of all, in order to exploit sinus periodicity, we consider the equation $x_1 = 2\pi y_k + y_\alpha$, where y_k represents the period in which x_1 lies and $y_\alpha \in [-\pi, \pi]$[3] represents the actual x_1 inside a given period. Then, we partition the interval $[-\pi, \pi]$ in four intervals: $I_1 = \left[-\pi, -\frac{\pi}{2}\right]$, $I_2 = \left[-\frac{\pi}{2}, 0\right]$, $I_3 = \left[0, \frac{\pi}{2}\right]$, $I_4 = \left[\frac{\pi}{2}, \pi\right]$. In each interval I_i ($i \in [4]$), we consider two linear functions $f_i^+(x)$ and and $f_i^-(x)$, such that for all $x \in I_i$, we have that $f_i^-(x) \leq \sin x \leq f_i^+(x)$. As an example, $f_1^+(y_\alpha) = -0.637y_\alpha - 2$ and $f_1^-(y_\alpha) = -0.707y_\alpha - 2.373$.

[3] In this section we write π for a rational approximation of it.

Let us consider the set of fresh continuous variables $Y^r = \{y_\alpha, y_{\sin}\}$ and the set of fresh discrete variables $Y^d = \{y_k, y_q, y_1, y_2, y_3, y_4\}$, being $y_1, \ldots, y_4$ boolean variables. The DTLHS model $\mathcal{I}_F$ for the inverted pendulum is the tuple (X, U, Y, N), where $X = \{x_1, x_2\}$ is the set of continuous state variables, $U = \{u\}$ is the set of input variables, $Y = Y^r \cup Y^d$ is the set of auxiliary variables, and the transition relation $N(X, U, Y, X')$ is the following guarded predicate:

$$
\begin{aligned}
&(x_1' = x_1 + 2\pi y_q + T x_2) \wedge (x_2' = x_2 + T\frac{g}{l} y_{\sin} + T\frac{1}{ml^2} uF) \\
&\wedge \textstyle\bigwedge_{i\in[4]} y_i \rightarrow f_i^-(y_\alpha) \leq y_{\sin} \leq f_i^+(y_\alpha) \\
&\wedge \textstyle\bigwedge_{i\in[4]} y_i \rightarrow y_\alpha \in I_i \wedge \sum_{i\in[4]} y_i \geq 1 \\
&\wedge x_1 = 2\pi y_k + y_\alpha \ \wedge \ -\pi \leq x_1' \leq \pi
\end{aligned}
$$

Overapproximations of the system behaviour increase system nondeterminism. Since $\mathcal{I}_F$ dynamics overapproximates the dynamics of the non-linear model, the controllers that we synthesize are inherently *robust*, that is they meet the given closed loop requirements *notwithstanding* nondeterministic small *disturbances* such as variations in the plant parameters. Tighter overapproximations of non-linear functions makes finding a controller easier, whereas coarser overapproximations makes controllers more robust.

The typical goal for the inverted pendulum is to turn the pendulum steady to the upright position, starting from any possible initial position, within a given speed interval.

5.2 The Multi-input Buck DC-DC Converter Case Study

The *multi-input* buck DC-DC converter [9] in Fig. 3 is a mixed-mode analog circuit converting the DC input voltage (V_i in Fig. 3) to a desired DC output voltage (v_O in Fig. 3). As an example, buck DC-DC converters are used off-chip to scale down the typical laptop battery voltage (12-24) to the just few volts needed by the laptop processor (e.g. [13]) as well as on-chip to support *Dynamic Voltage and Frequency Scaling* (DVFS) in multicore processors (e.g. [14]). Because of its widespread use, control schemas for buck DC-DC converters have been widely studied (e.g. see [14,13]). The typical software based approach (e.g. see [13]) is to control the switches $u_1, \ldots, u_n$ in Fig. 3 (typically implemented with a MOSFET) with a microcontroller.

In such a converter (Fig. 3), there are n power supplies with voltage values $V_1, \ldots, V_n$, n switches with voltage values $v_1^u, \ldots, v_n^u$ and current values $I_1^u, \ldots, I_n^u$, and n input diodes $D_0, \ldots, D_{n-1}$ with voltage values $v_0^D, \ldots, v_{n-1}^D$ and current $i_0^D, \ldots, i_{n-1}^D$ (in the following, we will write v_D for v_0^D and i_D for i_0^D).

The circuit state variables are i_L and v_C. However we can also use the pair i_L, v_O as state variables in the DTLHS model since there is a linear relationship between i_L, v_C and v_O, namely: $v_O = \frac{r_C R}{r_C+R} i_L + \frac{R}{r_C+R} v_C$. We model the n-input buck DC-DC converter with the DTLHS $\mathcal{B}_n = (X, U, Y, N)$, with $X = [i_L, v_O]$, $U = [u_1, \ldots, u_n]$, $Y = [v_D, v_1^D, \ldots, v_{n-1}^D, i_D, I_1^u, \ldots, I_n^u, v_1^u, \ldots, v_n^u, q_0, \ldots, q_{n-1}]$.

Finally, the transition relation N, depending on variables in X, U and Y (as well as on circuit parameters V_i, R, R_{on}, R_{off}, r_L, r_C, L and C), may be derived from simple circuit analysis [15]. Namely, we have the following equations:

$$
\dot{i}_L = a_{1,1} i_L + a_{1,2} v_O + a_{1,3} v_D, \qquad \dot{v}_O = a_{2,1} i_L + a_{2,2} v_O + a_{2,3} v_D
$$

where the coefficients $a_{i,j}$ depend on the circuit parameters R, r_L, r_C, L and C in the following way: $a_{1,1} = -\frac{r_L}{L}$, $a_{1,2} = -\frac{1}{L}$, $a_{1,3} = -\frac{1}{L}$, $a_{2,1} = \frac{R}{r_c+R}[-\frac{r_c r_L}{L} + \frac{1}{C}]$, $a_{2,2} = \frac{-1}{r_c+R}[\frac{r_c R}{L} + \frac{1}{C}]$, $a_{2,3} = -\frac{1}{L}\frac{r_c R}{r_c+R}$. Using a discrete time model with sampling time T (writing x' for $x(t+1)$) we have:

$$
\begin{aligned}
i'_L &= (1 + Ta_{1,1})i_L + Ta_{1,2}v_O + Ta_{1,3}v_D \\
v'_O &= Ta_{2,1}i_L + (1 + Ta_{2,2})v_O + Ta_{2,3}v_D.
\end{aligned}
$$

The algebraic constraints stemming from the constitutive equations of the switching elements are the following:

$$
\begin{aligned}
q_0 &\to v_D = R_{\text{on}} i_D \\
q_0 &\to i_D \geq 0 \\
\bigwedge_{i=1}^{n-1} q_i &\to v_i^D = R_{\text{on}} I_i^u \\
\bigwedge_{i=1}^{n-1} q_i &\to I_i^u \geq 0 \\
\bigwedge_{j=1}^{n} u_j &\to v_j^u = R_{\text{on}} I_j^u \\
i_L &= i_D + \sum_{i=1}^{n} I_i^u
\end{aligned}
\qquad
\begin{aligned}
\bar{q}_0 &\to v_D = R_{\text{off}} i_D \\
\bar{q}_0 &\to v_D \leq 0 \\
\bigwedge_{i=1}^{n-1} \bar{q}_i &\to v_i^D = R_{\text{off}} I_i^u \\
\bigwedge_{i=1}^{n-1} \bar{q}_i &\to v_i^D \leq 0 \\
\bigwedge_{j=1}^{n} \bar{u}_j &\to v_j^u = R_{\text{off}} I_j^u \\
\bigwedge_{i=1}^{n-1} v_D &= v_i^u + v_i^D - V_i \\
v_D &= v_n^u - V_n
\end{aligned}
$$

The typical goal for a multi-input buck is to drive i_L and v_O within given goal intervals.

5.3 Experimental Setting

All experiments have been carried out on a cluster with 4 nodes and Open MPI implementation of MPI. Each node contains 4 quad-core 2.83 GHz Intel Xeon E5440 processors with 25 GB of RAM. This allows us to run fully parallel experiments by configuring the MPI computation to use up to 16 processes per node. However, in order not to overload each node, we run maximum 15 processes per node, thus our upper bound for the number of processes is 60.

In the inverted pendulum $\mathcal{I}_F$ with force intensity F, as in [10], we set pendulum parameters l and m in such a way that $\frac{g}{l} = 1$ (i.e. $l = g$) $\frac{1}{ml^2} = 1$ (i.e. $m = \frac{1}{l^2}$). As for the admissible region, we set $A_{x_1} = [-1.1\pi, 1.1\pi]$ (we write π for a rational approximation of it) and $A_{x_2} = [-4, 4]$.

In the multi-input buck DC-DC converter with n inputs $\mathcal{B}_n$, we set constant parameters as follows: $L = 2 \cdot 10^{-4}$ H, $r_L = 0.1\ \Omega$, $r_C = 0.1\ \Omega$, $R = 5\ \Omega$, $R_{\text{on}} = 0\ \Omega$, $R_{\text{off}} = 10^4\ \Omega$, $C = 5 \cdot 10^{-5}$ F, and $V_i = 10i$ V for $i \in [n]$. As for the admissible region, we set $A_{i_L} = [-4, 4]$ and $A_{v_O} = [-1, 7]$.

As for quantization, we will use an even number of bits b, so that each state variable of each case study is quantized with $\frac{b}{2}$ bits. We recall that the number of abstract states is exactly 2^b.

We run *QKS* and *PQKS* on the inverted pendulum model $\mathcal{I}_F$ with $F = 0.5N$ (force intensity), and on the multi-input buck DC-DC model $\mathcal{B}_n$, with $n = 5$ (number of inputs). For the inverted pendulum, we use sampling time $T = 0.01$ seconds. For the multi-input buck, we set $T = 10^{-6}$ seconds. For both systems, we run experiments varying the number of bits $b = 18, 20$ (also 22 for the inverted pendulum) and the number of processors (workers) $p = 1, 10, 20, 30, 40, 50, 60$. Furthermore, each single experiment (corresponding to a (b, p) pair) is repeated 10 times, and all experimental measures are obtained by averaging among the 10 different runs.

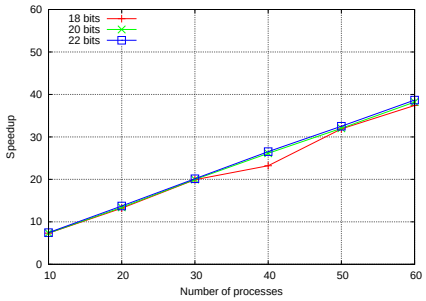

Fig. 4. Inverted pendulum: speedup.

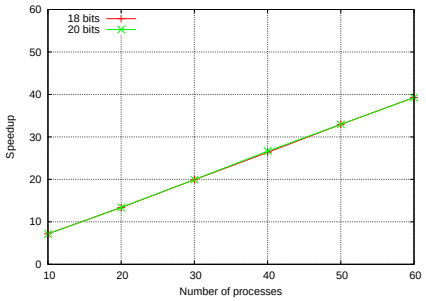

Fig. 5. Multi-input buck: speedup.

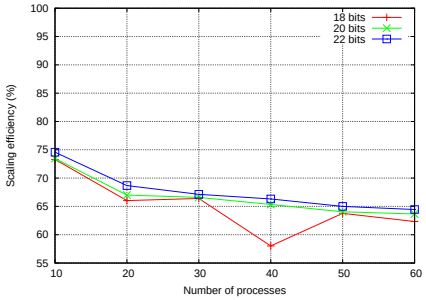

Fig. 6. Inverted pendulum: scaling efficiency.

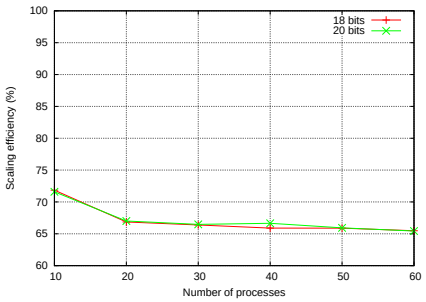

Fig. 7. Multi-input buck: scaling efficiency.

In order to evaluate effectiveness of our approach, we use the following standard measures: speedup, efficiency, communication time (in seconds) and I/O time (in seconds). The *speedup* of our approach is given by the percentage ratio between the serial CPU time and the parallel CPU time, i.e. $\text{Speedup} = \frac{\text{serial CPU}}{\text{parallel CPU}}\%$. To evaluate scalability of our approach, we define the *scaling efficiency* (or simply *efficiency*) as the percentage ratio between speedup and number of processors p, i.e. $\text{Efficiency} =$

$\frac{\text{Speedup}}{p}\%$. W.r.t. Algs. 3 and 4, the *communication time* is given by $\sum_{i=2}^{p} t_i$, being t_i the time needed by worker i to communicate with the master (we recall that worker 1 coincides with the master). Essentially, each t_i includes the time for MPI_Barrier syncronization (see Sect. 4.1) and local control abstraction $\hat{N}_i$ sending. In agreement with Sect. 4.1, the communication time is increased by the I/O time, that is the overall time spent by processors in input/output activities. The I/O time measure will also be shown separately in our experimental results.

Figs. 4, 6, 8 and 10 show, respectively, the speedup, the scaling efficiency, the communication time and the I/O time of Algs. 3 and 4 as a function of p, for the inverted pendulum with $b = 18, 20, 22$. Analogously, Figs. 5, 7, 9 and 11 show the same measures for the multi-input buck with $b = 18, 20$.

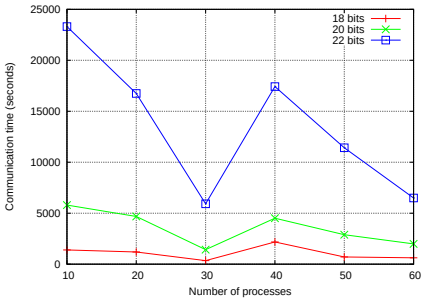

Fig. 8. Inverted pendulum: communication time

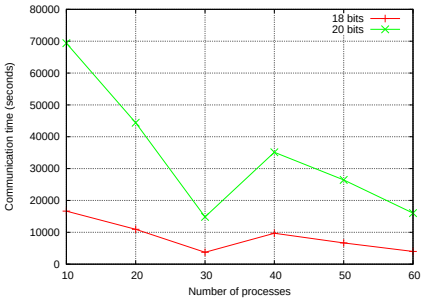

Fig. 9. Multi-input buck: communication time

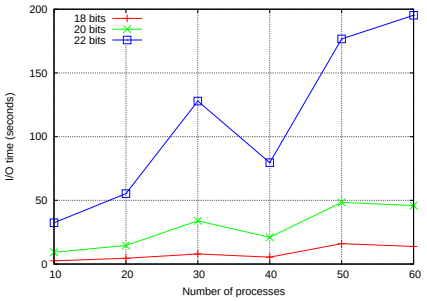

Fig. 10. Inverted pendulum: I/O time.

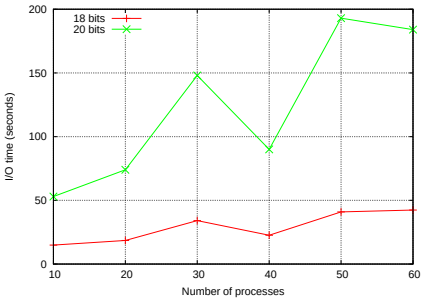

Fig. 11. Multi-input buck: I/O time.

We also show the absolute values for such experiments in Tabs. 1 (for the pendulum) and 2 (for the buck). Tabs. 1 and 2 have common columns. The meaning of such common columns is as follows. Column **b** is the number of bits used for quantization. Column **QKS** reports the execution time in seconds (averaged on 10 runs, with maximum standard deviation 0.9%) needed by *QKS* to compute the control abstraction (i.e. Alg. 1). Columns **PQKS** report experimental values for *PQKS*. Namely, column

p shows the number of processors, column **CPU** reports the execution time in seconds (averaged on 10 runs, with maximum standard deviation 4.2%) for Alg. 3 (i.e., the master execution time, since it wraps the overall parallel computation), column **CT** shows the communication time (averaged on 10 runs, with maximum standard deviation 21%; we recall that I/O time is included in this measure), column **IO** shows the I/O time only (averaged on 10 runs, with maximum standard deviation 31%), column **Speedup** reports the speedup and column **Efficiency** reports the scaling efficiency. Finally, column **CPU K** shows the execution time in seconds for the control software generation (i.e., the remaining computation of *QKS*, after the control abstraction generation).

Table 1. Experimental Results for inverted pendulum.

	QKS	*PQKS*						
b	CPU	p	CPU	CT	IO	Speedup	Efficiency	CPU K
18	6.141e+03	10	8.378e+02	1.395e+03	2.545e+00	7.330	73.297	2.000e+01
18	6.141e+03	20	4.650e+02	1.195e+03	4.500e+00	13.206	66.032	2.000e+01
18	6.141e+03	30	3.083e+02	3.477e+02	7.900e+00	19.919	66.396	2.000e+01
18	6.141e+03	40	2.646e+02	2.176e+03	5.400e+00	23.209	58.022	2.000e+01
18	6.141e+03	50	1.926e+02	7.065e+02	1.600e+01	31.885	63.770	2.000e+01
18	6.141e+03	60	1.642e+02	6.254e+02	1.380e+01	37.400	62.333	2.000e+01
20	2.608e+04	10	3.551e+03	5.800e+03	9.222e+00	7.346	73.456	8.500e+01
20	2.608e+04	20	1.946e+03	4.680e+03	1.460e+01	13.402	67.008	8.500e+01
20	2.608e+04	30	1.306e+03	1.425e+03	3.390e+01	19.978	66.593	8.500e+01
20	2.608e+04	40	9.981e+02	4.511e+03	2.100e+01	26.135	65.337	8.500e+01
20	2.608e+04	50	8.145e+02	2.889e+03	4.840e+01	32.026	64.052	8.500e+01
20	2.608e+04	60	6.828e+02	1.991e+03	4.590e+01	38.203	63.672	8.500e+01
22	1.106e+05	10	1.484e+04	2.331e+04	3.240e+01	7.457	74.566	3.520e+02
22	1.106e+05	20	8.055e+03	1.675e+04	5.530e+01	13.736	68.681	3.520e+02
22	1.106e+05	30	5.494e+03	5.923e+03	1.279e+02	20.141	67.136	3.520e+02
22	1.106e+05	40	4.171e+03	1.742e+04	7.960e+01	26.526	66.314	3.520e+02
22	1.106e+05	50	3.404e+03	1.142e+04	1.767e+02	32.503	65.005	3.520e+02
22	1.106e+05	60	2.861e+03	6.491e+03	1.952e+02	38.672	64.453	3.520e+02

5.4 Experiments Discussion

From Figs. 4 and 5 we note that the speedup is almost linear, with a $\frac{3}{5}$ slope. From Figs. 6 and 7 we note that scaling efficiency remains high when increasing the number of processors p. For example, for $b = 22$ bits, our approach efficiency is in a range from 74% (10 processors) to 64% (60 processors). In any case, efficiency is almost always above 60%, especially for bigger values of b.

Figs. 8 and 9 show that communication time almost always decreases when p increases. This is motivated by the fact that, in our MPI implementation, communication among nodes takes place mostly when workers send their local control abstractions to the master via the shared filesystem. Since in our implementation this happens only after an MPI_Barrier (i.e., the parallel computation may proceed only when all nodes

have reached an MPI_Barrier statement), the communication time also includes waiting time for workers which finishes their local computation before the other ones. Thus, if all workers need about the same time to complete the local computation, then the communication time is low. Note that this explains also the discontinuity when passing from 30 to 40 nodes which may be observed in the figures above. In fact, each worker has (almost) the same workload in terms of abstract states number, but some abstract states may need more computation time than others (i.e., computation time of function *minCtrAbsAux* in Alg. 2 may have significant variations on different abstract states). If such "hard" abstract states are well distributed among workers, communication time is low (with higher efficiency), otherwise it is high. Figs. 12 and 13 show such phenomenon on the inverted pendulum quantized with 18 bits, when the parallel algorithm is executed by 30 and 40 workers, respectively. In such figures, the x-axis represents computation time, the y-axis the workers, and hard abstract states are represented in red. Indeed, in Fig. 12 hard abstract states are well distributed among workers, which corresponds to a low communication time in Fig. 8 (and high speedup and efficiency in Figs. 4 and 6). On the other hand, in Fig. 13 hard abstract states are mainly distributed on only a dozen of the 40 workers (thus, about 30% of the workers performs the most part of the total workload), which corresponds to a high communication time in Fig. 8 (and low speedup and efficiency in Figs. 4 and 6). Note that I/O time is nearly always at least 2 orders of magnitude less than communication time, thus hard abstract states distribution is indeed the cause of the above described phenomenon.

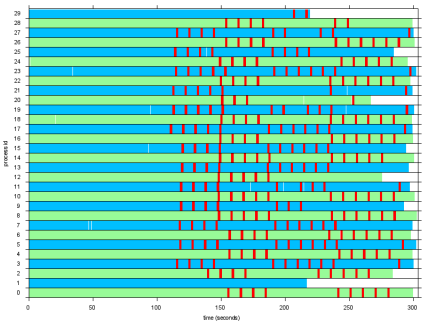

Fig. 12. Details about pendulum computation time (30 nodes, 18 bits).

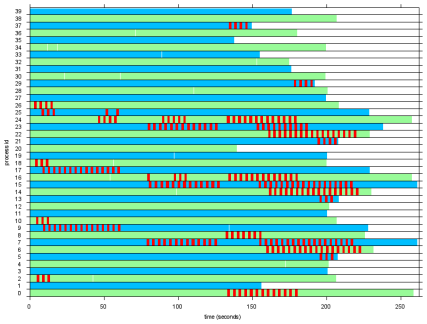

Fig. 13. Details about pendulum computation time (40 nodes, 18 bits).

Finally, in order to show feasibility of our approach also on DTLHSs requiring a huge computation time to generate the control abstraction, we run *PQKS* on the inverted pendulum with $b = 26$. We estimate the computation time for control abstraction generation for $p = 1$ to be 25 days. On the other hand, with $p = 60$, we are able to compute the control abstraction generation in only 16 hours.

6 Related Work

Algorithms (and tools) for the automatic synthesis of control software under different assumptions (e.g., discrete or continuous time, linear or non-linear systems, hybrid or

Table 2. Experimental Results for multi-input buck DC-DC converter.

	QKS	*PQKS*						
b	CPU	p	CPU	CT	IO	Speedup	Efficiency	CPU K
18	6.484e+04	10	9.024e+03	1.666e+04	1.490e+01	7.185	71.847	2.600e+01
18	6.484e+04	20	4.849e+03	1.095e+04	1.850e+01	13.371	66.854	2.600e+01
18	6.484e+04	30	3.256e+03	3.721e+03	3.410e+01	19.914	66.381	2.600e+01
18	6.484e+04	40	2.460e+03	9.710e+03	2.260e+01	26.358	65.895	2.600e+01
18	6.484e+04	50	1.968e+03	6.677e+03	4.090e+01	32.945	65.889	2.600e+01
18	6.484e+04	60	1.650e+03	4.001e+03	4.240e+01	39.287	65.478	2.600e+01
20	2.629e+05	10	3.673e+04	6.938e+04	5.300e+01	7.159	71.590	8.000e+01
20	2.629e+05	20	1.962e+04	4.439e+04	7.400e+01	13.401	67.007	8.000e+01
20	2.629e+05	30	1.318e+04	1.484e+04	1.480e+02	19.945	66.484	8.000e+01
20	2.629e+05	40	9.862e+03	3.513e+04	9.000e+01	26.662	66.654	8.000e+01
20	2.629e+05	50	7.976e+03	2.645e+04	1.930e+02	32.966	65.932	8.000e+01
20	2.629e+05	60	6.697e+03	1.603e+04	1.840e+02	39.262	65.436	8.000e+01

discrete systems, etc.) have been widely investigated in the last decades. As an example, see [16,17,18,10,19,20,21,22] and citations thereof. However, no one of such approaches has a parallel version of any type, our focus here. On the other hand, parallel algorithms have been widely investigated for formal verification (e.g., see [23,24,25]).

A parallel algorithm for control software synthesis has been presented in [26], where however non-hybrid systems are addressed, control is obtained by Monte Carlo simulation and quantization is not taken into account. Moreover, note that in literature "parallel controller synthesis" often refers to synthesizing parallel controllers (e.g., see [27] and [28] and citations thereof), while here we parallelize the (offline) computation required to synthesize a standalone controller. Summing up, to the best of our knowledge, no previous parallel algorithm for control software synthesis from formal specifications has been published.

As discussed in Sect. 1.1, the present paper builds mainly upon the tool *QKS* presented in [2,3]. Other works about *QKS* comprise the following ones. In [29] it is shown that expressing the input system as a linear predicate over a set of continuous as well as discrete variables (as it is done in *QKS*) is not a limitation on the modeling power. In [12] it is shown how non-linear systems may be modeled by using suitable linearization techniques. The paper in [15] addresses model based synthesis of control software by trading system level non-functional requirements (such us optimal set-up time, ripple) with software non-functional requirements (its footprint, i.e. size). The procedure which generates the actual control software (C code) starting from a finite states automaton of a control law is described in [30]. In [31] it is shown how to automatically generate a picture illustrating control software coverage. Finally, in [32] it is shown that the quantized control synthesis problem underlying *QKS* approach is undecidable. As a consequence, *QKS* is based on a correct but non-complete algorithm. Namely, *QKS* output is one of the following: i) SOL, in which case a correct-by-construction control software is returned; ii) NOSOL, in which case no controller exists for the given specifications; iii) UNK, in which case *QKS* was not able to compute a controller (but a controller may exist).

7 Conclusions and Future Work

In this paper we presented a Map-Reduce style parallel algorithm (and its MPI implementation for computer clusters, *PQKS*) for automatic synthesis of correct-by-construction control software for discrete time linear hybrid systems, starting from a formal model of the controlled system, safety and liveness requirements and number of bits for analog-to-digital conversion. Such an algorithm significantly improves performance of an existing standalone approach (implemented in the tool *QKS*), which may require weeks or even months of computation when applied to large-sized hybrid systems.

Experimental results on two classical control synthesis problems (the inverted pendulum and the multi-input buck DC/DC converter) show that our parallel approach efficiency is above 60%. As an example, with 60 processors *PQKS* outputs the control software for the 26-bits quantized inverted pendulum in about 16 hours, while *QKS* needs about 25 days of computation.

Future work consists in further improving the communication among processors by making the mapping phase aware of "hard" abstract states (see Sect. 5.4), as well as designing a parallel version for other architectures than computer clusters, such as GPGPU architectures. Finally, future work also includes extending the presented approach so as to provide a general parallelization framework for abstraction procedures (of a suitable type).

Acknowledgments We are grateful to our anonymous referees for their helpful comments. Our work has been partially supported by: i) MIUR project DM24283 (TRAMP); ii) EC FP7 project GA600773 (PAEON); iii) EC FP7 project GA317761 (SmartHG); and iv) Erasmus Mundus MULTIC scholarship from the European Commission (EMA 2 MULTIC 10-837).

References

1. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: FM. LNCS 4085 (2006) 1–15
2. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Synthesis of quantized feedback control software for discrete time linear hybrid systems. In: CAV. LNCS 6174 (2010) 180–195
3. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Model based synthesis of control software from system level formal specifications. ACM Trans. on Soft. Eng. and Meth. **To appear**
4. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Quantized feedback control software synthesis from system level formal specifications. CoRR **abs/1107.5638v1** (2011)
5. Tomlin, C., Lygeros, J., Sastry, S.: Computing controllers for nonlinear hybrid systems. In: HSCC. LNCS 1569 (1999) 238–255
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI. (2004) 137–150
7. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers (2010)
8. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann (1997)
9. Rodriguez, M., Fernandez-Miaja, P., Rodriguez, A., Sebastian, J.: A multiple-input digitally controlled buck converter for envelope tracking applications in radiofrequency power amplifiers. IEEE Trans on Pow El **25**(2) (2010) 369–381

10. Kreisselmeier, G., Birkhölzer, T.: Numerical nonlinear regulator design. IEEE Trans. on Automatic Control **39**(1) (1994) 33–46
11. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. on Computers **C-35**(8) (1986) 677–691
12. Alimguzhin, V., Mari, F., Melatti, I., Salvo, I., Tronci, E.: Automatic control software synthesis for quantized discrete time hybrid systems. In: CDC, IEEE (2012) 6120–6125
13. So, W.C., Tse, C., Lee, Y.S.: Development of a fuzzy logic controller for dc/dc converters: design, computer simulation, and experimental evaluation. IEEE Trans. on Power Electronics **11**(1) (1996) 24–32
14. Kim, W., Gupta, M.S., Wei, G.Y., Brooks, D.M.: Enabling on-chip switching regulators for multi-core processors using current staggering. In: ASGI. (2007)
15. Alimguzhin, V., Mari, F., Melatti, I., Salvo, I., Tronci, E.: On model based synthesis of embedded control software. In: EMSOFT. (2012)
16. Bemporad, A., Giorgetti, N.: A sat-based hybrid solver for optimal control of hybrid systems. In: HSCC. LNCS 2993 (2004) 126–141
17. Della Penna, G., Magazzeni, D., Tofani, A., Intrigila, B., Melatti, I., Tronci, E.: Automated Generation of Optimal Controllers through Model Checking Techniques. Volume 15 of Lecture Notes in Electrical Engineering. Springer (2008)
18. Della Penna, G., Magazzeni, D., Mercorio, F., Intrigila, B.: UPMurphi: A tool for universal planning on pddl+ problems. In: ICAPS. (2009)
19. Mazo, M.J., Tabuada, P.: Symbolic approximate time-optimal control. Systems & Control Letters **60**(4) (2011) 256–263
20. Jha, S., Seshia, S.A., Tiwari, A.: Synthesis of optimal switching logic for hybrid systems. In: EMSOFT, ACM (2011) 107–116
21. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal: Status & developments. In: CAV. LNCS 1254 (1997) 456–459
22. Cassez, F., Jessen, J.J., Larsen, K.G., Raskin, J.F., Reynier, P.A.: Automatic synthesis of robust and optimal controllers - an industrial case study. In: HSCC. (2009) 90–104
23. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. Int. J. Softw. Tools Technol. Transf. **11**(1) (2009) 13–25
24. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Legay, A.: Distributed parametric and statistical model checking. In: PDMC. (2011) 30–42
25. Barnat, J., Brim, L., Ceska, M., Rockai, P.: Divine: Parallel distributed model checker. In: PDMC. PDMC-HIBI '10, Washington, DC, USA, IEEE Computer Society (2010) 4–7
26. Schubert, W., Stengel, R.: Parallel synthesis of robust control systems. IEEE Trans. on Contr. Sys. Techn **6**(6) (1998) 701–706
27. Jurikovič, M., Čičák, P., Jelemenská, K.: Parallel controller design and synthesis. In: Proceedings of the 7th FPGAworld Conference. FPGAworld '10, New York, NY, USA, ACM (2010) 35–40
28. Pardey, J., Amroun, A., Bolton, M., Adamski, M.: Parallel controller synthesis for programmable logic devices. Microprocessors and Microsystems **18**(8) (1994) 451 – 457
29. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Linear constraints as a modeling language for discrete time hybrid systems. In: ICSEA, IARIA (2012)
30. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Synthesizing control software from boolean relations. Int. J. on Advances in SW **5**(3&4) (2012) 212–223
31. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Control software visualization. In: INFOCOMP, IARIA (2012)
32. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Undecidability of quantized state feedback control for discrete time linear hybrid systems. In: ICTAC. LNCS 7521 (2012) 243–258