On the CALM Principle for BSP Computation

Matteo Interlandi [1] and Letizia Tanca [2]

[1] University of California, Los Angeles
minterlandi@cs.ucla.edu
[2] Politecnico di Milano,
letizia.tanca@polimi.it

Abstract. In recent times, considerable emphasis has been given to two apparently disjoint research topics: *data-parallel* and *eventually consistent, distributed* systems. In this paper we propose a study on an *eventually consistent, data-parallel computational model*, the keystone of which is provided by the recent finding that a class of programs exists that can be computed in an eventually consistent, coordination-free way: *monotonic programs*. This principle is called CALM and has been proven by Ameloot et al. for distributed, asynchronous settings. We advocate that CALM should be employed as a basic theoretical tool also for data-parallel systems, wherein computation usually proceeds synchronously in rounds and where communication is assumed to be reliable. We deem this problem relevant and interesting, especially for what concerns *parallel workflow optimization*, and make the case that CALM does not hold in general for data-parallel systems if the techniques developed by Ameloot et al. are directly used. In this paper we sketch how, using novel techniques, the satisfiability of the if direction of the CALM principle can still be obtained, although just for a subclass of monotonic queries.

1 Introduction

Recent research has explored ways to exploit different levels of *consistency* in order to improve the performance of distributed systems *w.r.t.* specific tasks and network configurations, while maintaining correctness [18]. A topic strictly related to consistency is *coordination*, usually informally interpreted as a mechanism to accomplish a distributed agreement on some system property [8]. Indeed, coordination can be used to enforce consistency when, in the natural execution of a system, this is not guaranteed in general. In this paper we sketch some theoretical problems springing from the use of eventually consistent, coordination-free computation over *synchronous systems with reliable communication* (*rsync*). Informally, such systems have the following properties: (i) a global clock is defined and accessible by every node; (ii) the relative difference between the time clock values of any two nodes is bounded; and (iii) the results emitted by a node arrive at destination at most after a certain *bounded physical time* (the so-called *bounded delay guarantee*).

Rsync is a common setting in modern data-parallel frameworks - such as MapReduce - in which computation is usually performed in *rounds*, where each task is blocked and cannot start the new round until a *synchronization barrier* is reached, *i.e.,* every

other task has completed its local computation. In this work we consider synchronization (barrier) and coordination as two different, although related entities: the former is a *mechanism* enforcing the *rsync* model, the latter a *property of executions*. Identifying under what circumstances eventually consistent, coordination-free computation can be employed over *rsync* systems would enable us to "stretch" the declarativeness of parallel programs, freeing execution plans of the restriction to follow predefined (synchronous) patterns. In fact, all recent high-level data-parallel languages suffer from this limitation, for instance both Hive [16] and Pig [15] sacrifice pipelining in order to fit query plans into MapReduce workflows. Our aim is then to understand when a synchronous "blocking" computation is actually required by the program semantics – and therefore must be strictly enforced by the system – and when, instead, a pipelined execution can be performed as optimization. For batch parallel processing, the benefits of understanding where the former can be replaced by the latter are considerable [7]: thanks to the fact that data is processed as soon as it is produced, *online computation* is possible, *i.e.,* the final result can be refined during the execution; as a consequence, new data can be incrementally added to the input, making *continuous computation* possible. Overall, pipelining is highly desirable in the Big Data context, where full materialization is often problematic.

Recently, a class of programs that can be computed in an eventually consistent, coordination-free way has been identified: *monotonic programs* [9]; this principle is called *CALM* (Consistency and Logical Monotonicity) and has been proven in [4]. While CALM was originally proposed to simplify the specification of distributed (asynchronous) data management systems, in this paper we advocate that CALM should be employed as a basic theoretical tool also for the declarative specification of data-parallel (synchronous) systems. As a matter of fact, CALM permits to link a property of the execution (coordination-freedom) with a class of programs (monotonic queries). But to which extent CALM can be applied over data-parallel systems? Surprisingly enough, the demonstration of the CALM principle in *rsync* systems is not trivial and, with the communication model and the notion of coordination as defined in [4], the CALM principle does not hold in general in *rsync* settings (cf. Example 3). Thus, in order to extend CALM over data-parallel synchronous computation, in this paper we sketch a new *generic parallel computation model* leveraging previous works on *synchronous Datalog* [10, 12] and *transducer networks* [4], and grounding *rsync* computation on the well-known *Bulk Synchronous Parallel* (BSP) model [17] equipped with content-based addressing. With BSP, computation proceeds as a series of global rounds, each composed by three phases: (i) a *computation phase*, in which nodes parallely perform local computations; (ii) a *communication phase*, where data are exchanged among the nodes; and (iii) the synchronization barrier. Exploiting this new type of transducer network, we will then show that the CALM principle is satisfied for synchronous and reliable systems *under a new definition of coordination-freedom*, although, surprisingly enough, just for a subclass of monotonic queries, *i.e.,* the *chained monotonic queries* (cf. Definition 5.7). When defining coordination-freedom we will take advantage of recent results describing how knowledge can be acquired in synchronous systems [5, 6].

Organization: The paper is organized as follows: Section 2 introduces some preliminary notation. Section 3 defines our model of synchronous and reliable parallel system,

and shows that the CALM principle is not satisfied for systems of this type. Section 3.2 proposes a new computational model based on hashing, while Section 4 introduces the new definition of coordination. Finally, Section 5 discusses CALM under the new setting. The paper ends with some concluding remarks. We refer the reader to [11] for proofs and more detailed discussions.

2 Relational Transducers

In this paper we expect the reader to be familiar with the basic notions of database theory and relational transducer (networks). In this section we use some example to set forth our notation, which is close to that of [1] and [4].

We employ a transducer (resp. a transducer network) as an abstraction modeling the behavior of a single computing node (resp. a network of computing nodes): this abstract computational model permits us to make our results as general as possible without having to rely on a particular framework, since transducers and transducer networks can be easily imposed over any modern data-parallel system. We consider each node to be equipped with an immutable *database* and a *memory* used to store useful data between any two consecutive computation steps. In addition, a node can produce an *output* for the user and can also *communicate* some data to other nodes (the concept of data communication in a transducer network will appear clearer in Section 3). Finally, an internal *time*, and *system* data are kept mainly for configuration purposes. Every node executes a *program* that operates on (input) instances of the database, the memory and the communication channel, and produces new instances that are either saved in memory, or directly output to the user, or addressed to other nodes.

Example 1. A first example of relational transducer is the following UCQ-transducer $\mathcal{T}$, with schema Υ, that computes the ternary relation Q as the join between two binary relations R and T:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{mem} = \emptyset, \Upsilon_{com} = \emptyset, \Upsilon_{out} = \{Q^{(3)}\}$

Program: $Q_{out}(u, v, w) \leftarrow R(u, v), T(v, w).$

Let $\mathbf{I}$ be an initial instance over which we want to compute the join. Then, let us define $I_{db} = \mathbf{I}$ as an instance over the $database$ schema Υ_{db}. A transition $I \rightarrow J$ for $\mathcal{T}$ is such that $I = \mathbf{I} \cup I_{sys}$, I_{rcv} and J_{snd} are empty (no communication query exists), and $J = \mathbf{I} \cup I_{out} \cup I_{sys}$, where I_{out} is the result of the query Q_{out}, *i.e.,* the join between R and T. Note that the subscript in Q_{out} means that this is an *output* query, that is, it specifies the final result of the whole computation.

3 Computation in rsync

In order to allow *query evaluation in parallel settings*, we will sketch a novel *transducer network* [4], where computation is *synchronous*, and communication is *reliable*. This permits us to define how a set of relational transducers can be assembled to obtain an abstract computational model for *distributed data-parallel systems*. To be consistent with [4], we will assume broadcasting as the addressing model.

Example 2. Assume we want to compute a distributed version of the join of Example 1. We can implement it using a broadcasting synchronous transducer network which emits one of the two relations, say T, and then joins R with the received facts over T. Note that the sent facts will be used just starting from the successive round, and the program will then employ two rounds to compute the distributed join. UCQ is again expressive enough. The transducer network can be written as follows – where S_{snd} denotes a *communication* query and this time schema Υ_{com} is non-empty because communication is needed:

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{com} = \{S^{(2)}\}, \Upsilon_{out} = \{Q^{(3)}\}$
Program: $S_{snd}(u, v) \leftarrow T(u, v).$
$Q_{out}(u, v, w) \leftarrow R(u, v), S(u, w).$

Synchronous specifications have the required expressive power:

Lemma 1 *Let $\mathcal{L}$ be a language containing* UCQ *and contained in* DATALOG$^{\neg}$. *Every query expressible in $\mathcal{L}$ can be distributively computed in 2 rounds by a broadcasting $\mathcal{L}$-transducer network.*

The above lemma permits us to draw the following conclusion: under the *rsync* semantics, monotonic and non-monotonic queries behave in the same way: two rounds are needed in both cases. This is due to the fact that, contrary to what happens in the asynchronous case of [4], starting from the second round we are guaranteed – by the reliability of the communication and the synchronous assumption – that every node will compute the query over every emitted instance. Conversely, in the asynchronous case, as a result of the non-determinism of the communication, we are never guaranteed, without coordination, that every sent fact will be actually received.

3.1 The CALM Conjecture

The *CALM conjecture* [9] specifies that a well-defined class of programs can be distributively computed in an *eventually consistent*, *coordination-free* way: *monotonic programs*. CALM has been proven in this (revisited) form for asynchronous systems [4]:

Conjecture 1 *A query can be distributively computed by a coordination-free transducer network if and only if it is monotonic.*

The concept of coordination suggests that all the nodes in a network must exchange information and wait until an agreement is reached about a common property of interest. Following this intuition, Ameloot et al. established that a specification is coordination-free if communication is not strictly necessary to obtain a consistent final result. Surprisingly enough, under this definition of coordination-freedom, CALM does not hold in *rsync* settings under the broadcasting communication model:

Example 3. Let $\mathcal{Q}_{out}$ be the "emptiness" query of [4]: given a nullary database relation $R^{(0)}$ and a nullary output relation $T^{(0)}$, $\mathcal{Q}_{out}$ outputs true (*i.e.,* a nullary fact over T) iff I_R is empty. The query is non-monotonic: if I_R is initially empty, then T is produced, but if just one fact is added to R, T is not derived, *i.e.,* I_T must be empty. A FO-transducer network $\mathcal{N}$ can be easily generated to distributively compute $\mathcal{Q}_{out}$: first every node emits R if its local partition is not empty, and then each node locally evaluates the emptiness of R. Since the whole initial instance is installed on every node when R is checked for emptiness, T is true only if R is actually empty on the initial instance. The complete specification follows.

Schema: $\Upsilon_{db} = \{R^{(0)}\}, \Upsilon_{mem} = \{\texttt{Ready}^{(0)}\}, \Upsilon_{com} = \{S^{(0)}\}, \Upsilon_{out} = \{T^{(0)}\}$.
Program: $S_{snd}() \leftarrow R()$.
$\texttt{Ready}_{ins}() \leftarrow \neg\texttt{Ready}()$.
$T_{out}() \leftarrow \neg S(), \texttt{Ready}()$.

One can show [11] that, if communication is switched off, the above transducer is still able to obtain the correct result if, for example, **I** is installed on every node. That is, a partitioning exists, making communication not strictly necessary to reach the proper result. Note that the same query requires coordination in asynchronous settings: since emitted facts are non-deterministically received, the only way to compute the correct result is that nodes coordinate to understand if the input instance is *globally* empty.

The result we have is indeed interesting although expected: when we move from the general asynchronous model to the more restrictive *rsync* setting, we no longer have a complete understanding of which queries can be computed without coordination, and which ones, instead, do require coordination. It turns out that both the communication model and the definition of coordination proposed in [4] are not strong enough to work in general for synchronous systems. As the reader may have realized, this is due to the fact that, in broadcasting synchronous systems, coordination – as defined by Ameloot et al. – is already "baked" into the model. In the next sections we will see that our definition of coordination-freedom guarantees eventually consistent computation for those queries that do not rely on broadcasting in order to progress. That is, the discriminating condition for eventual consistency is not monotonicity, but the fact that it *is not necessary* to send a fact to all the nodes composing a network.

3.2 Hashing Transducer Networks

Broadcasting specifications are not really convenient from a practical perspective. Following other parallel programming models such as MapReduce, in this section we are going to introduce *hashing transducer networks*: *i.e.,* synchronous networks of relational transducers equipped with a content-based communication model founded on hashing. Under this new model, the node to which an emitted fact must be addressed is derived using a hash function applied to a subset of its terms called *keys*.

Example 4. This program is the hashed version of Example 2, where every tuple emitted over S and U is hashed on the first term (this is specified by the schema definition

$S^{(1,2)}$ and $U^{(1,2)}$, where the pair $(1, 2)$ means that the related relation has arity 2 and the first term is the key-term). In this way we are assured that, for each pair of joining tuples, at least a node exists containing the pair. This because S and U are joined over their key-terms, and hence the joining tuples are addressed to the same node.

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(2)}\}, \Upsilon_{com} = \{S^{(1,2)}, U^{(1,2)}\}, \Upsilon_{out} = \{J^{(3)}\}$
Program: $S_{snd}(u, v) \leftarrow R(u, v).$
$U_{snd}(u, v) \leftarrow T(u, v).$
$J_{out}(u, v, w) \leftarrow S(u, v), U(u, w).$

---

4 Coordination-freedom Refined

We have seen in Section 3.1 that, for *rsynch* systems, a particular notion of coordination-freedom is needed. In fact we have shown that, under such model, certain non-monotonic queries – Example 3 – requiring coordination under the asynchronous model can be computed in a coordination-free way. The key-point is that, as observed in [4], in asynchronous systems coordination-freedom is directly related to communication-freedom under ideal partitioning. That is, if the partitioning is correct, no communication is required to correctly compute a coordination-free query because (i) no data must be sent (the partition is correct), and (ii) no "control message" is required to obtain a consistent result (the query is coordination-free). However, due to its synchronous nature, in *rsync* settings non-monotonic queries can be computed in general without resorting to coordination because coordination is already "baked" into the *rsync* model: each node is synchronized with every other one, hence "control messages" are somehow implicitly assumed. In this section we introduce a novel knowledge-oriented perspective linking coordination with the way in which explicit and implicit information flows in the network. Under this perspective, we will see that coordination is needed if, to maintain consistency, a node must have some form of information exchange with all the other nodes.

4.1 Syncausality

Achieving coordination in asynchronous systems is a costly task. A necessary condition for coordination in such systems is the existence of primitives that enforce some control over the ordering of events. In a seminal paper [13], Lamport proposed a synchronization algorithm based on the relation of *potential causality* ($\rightarrow$) over asynchronous events. According to Lamport, given two events e, e', we have that $e \rightarrow e'$ if e happens before e' and e might have caused e'. From a high-level perspective, the potential causality relation models how information flows among processes, and therefore can be employed as a tool to reason on the patterns which cause coordination in asynchronous systems. A question now arises: what is the counterpart of the potential causality relation for synchronous systems? *Synchronous potential causality* (*syncausality* in

short) has been recently proposed [5] to generalize Lamport's potential causality to synchronous systems. Using syncausality we are able to model how information flows among nodes with the passing of time. Consider a parallel execution trace ρ – called a *run* – and two *points* in this execution (ρ^i, t), (ρ^j, t') for (possibly not distinct) nodes i, j, identifying the local state for i, j at time t and t' respectively. We say that (ρ^j, t') *causally depends* on (ρ^i, t) if either $i = j$ and $t \leq t'$ – *i.e.,* a local state depends on the previous one – or a tuple has been emitted by node i at time t, addressed to node j, with $t < t'$[1]. We refer to these two types of dependencies as *direct*.

Definition 4.1. *Given a run ρ, we say that two points (ρ^i, t), (ρ^j, t') are related by a* direct potential causality *relation $\rightarrow$, if one of the following is true:*

1. *$t' = t + 1$ and $i = j$;*
2. *$t' \geq t + 1$ and node i sent a tuple at time t addressed to j;*
3. *there is a point (ρ^k, t'') s.t. $(\rho^i, t) \rightarrow (\rho^k, t'')$ and $(\rho^k, t'') \rightarrow (\rho^j, t')$.*

Note that direct dependencies define precisely Lamport's happen-before relation – and hence we maintain the same signature $\rightarrow$.

Differently from asynchronous systems, we however have that a point on node j can occasionally *indirectly* depend on another point on node i even if no fact addressed to j is actually sent by i. This is because j can still draw some conclusion simply as a consequence of the *bounded delay guarantee* of synchronous systems. That is, each node can use the common knowledge that every sent tuple is received at most after a certain bounded delay to reason about the state of the system. The bounded delay guarantee can be modelled as an imaginary $NULL$ fact, like in [14]. Under this perspective, indirect dependencies appear the same as the direct ones, although, instead of a flow generated by "informative" facts, with the indirect relationship we model the flow of "non-informative", $NULL$ facts.

Definition 4.2. *Given a run ρ, we say that two points (ρ^i, t), (ρ^j, t') are related by an* indirect potential causality *relation $\dashrightarrow$, if $i \neq j$, $t' \geq t + 1$ and a $NULL_R^i$ fact addressed to node j has been (virtually) sent by node i at round t.*

An interesting fact about the bounded delay guarantee is that it can be employed to specify when negation can be safely applied to a predicate. In general, negation can be applied to a literal $R(\bar{u})$ when the content of R is sealed for what concerns the current round. In local settings, we have that such condition holds for a predicate at round t' if its content has been completely generated at round t, with $t' > t$. In distributed settings, we have that if R is a $communication$ relation, being in a new round t' is not enough, in general, for establishing that its content is sealed. This is because tuples can still be floating, and therefore, until we are assured that every tuple has been delivered, the above condition does not hold. The result is that negation cannot be applied safely. We can reason in the same way also for every other negative literal depending on R. We will then model the fact that the content of a $communication$ relation R is stable because of the bounded delay guarantee, by having every node i emit a fact $NULL_R^i$ at round t, for every $communication$ relation R, which will be delivered at node j exactly by the

[1] Note that a point in a synchronous system is what Lamport defines as an event in an asynchronous system.

next round. We then have that the content of R is stable once j has received a $NULL_R^i$ fact from every node i contained in the set N of nodes composing the network. The sealing of a $communication$ relation at a certain round is then ascertained only when $|N|$ $NULL_R$ facts have been counted. Recall that not necessarily the $NULL_R^i$ facts must be physically sent. This in particular is true under our *rsync* model, where the strike of a new round automatically seals all the $communication$ relations. Example 5 shows one situation in which this applies.

Example 5. Consider the hashing version of the program of Example 3. Let **I** be an initial instance. At round $t+1$ we have that the relation S is stable, and hence negation can be applied. Note that if R is empty in the initial instance, no fact is sent. Despite this, every node can still conclude at round $t+1$ that the content of S is stable. In this situation we clearly have an indirect potential causality relation.

We are now able to introduce the definition of syncausality: a generalization of Lamport's happen-before relation which considers not only the direct information flow, but also the flow generated by indirect dependencies.

Definition 4.3. *Let ρ be a run. The* syncausality *relation $\rightsquigarrow$ is the smallest relation s.t.:*

1. *if $(\rho^i, t) \rightarrow (\rho^j, t')$, then $(\rho^i, t) \rightsquigarrow (\rho^j, t')$;*
2. *if $(\rho^i, t) \dashrightarrow (\rho^j, t')$, then $(\rho^i, t) \rightsquigarrow (\rho^j, t')$; and*
3. *if $(\rho^i, t) \rightsquigarrow (\rho^j, t')$ and $(\rho^j, t') \rightsquigarrow (\rho^k, t'')$, then $(\rho^i, t) \rightsquigarrow (\rho^k, t'')$.*

4.2 From Syncausality to Coordination

We next propose the *predicate-level syncausality* relationship, modeling causal relations at the predicate level. That is, instead of considering how (direct and indirect) information flows between nodes, we introduce a more fine-grained relationship modelling the flows between *predicates and nodes*.

Definition 4.4. *Given a run ρ, we say that two points (ρ^i, t), (ρ^j, t') are linked by a relation of* predicate-level syncausality $\overset{R}{\rightsquigarrow}$*, if any of the following holds:*

1. *$i = j$, $t' = t+1$ and a tuple over $R \in \Upsilon_{mem} \cup \Upsilon_{out}$ has been derived by a query in $Q_{ins} \cup Q_{out}$ at time t';*
2. *$R \in \Upsilon_{com}$ and node i sends a tuple over R at time t addressed to node j, with $t' \geq t+1$;*
3. *$R \in \Upsilon_{com}$ and node i (virtually) sends a $NULL_R^i$ fact at time t addressed to node j, with $t' \geq t+1$;*
4. *there is a point (ρ^k, t'') s.t. $(\rho^i, t) \overset{R}{\rightsquigarrow} (\rho^k, t'')$ and $(\rho^k, t'') \overset{R}{\rightsquigarrow} (\rho^j, t')$.*

We are now able to specify a condition for achieving coordination. Informally, we have that coordination exists when all the nodes of a network reach a common agreement that some event happened. But the only way to reach such an agreement is that a (direct or indirect) information flow exists between the node in which the event actually occurs, and every other node. This is a sufficient and necessary condition because of the reliability and bounded-delay guarantee of *rsync* systems. Formalizing this intuition by means of the (predicate level) syncausality relationship we have that:

Definition 4.5. *Let N be a set of nodes. We say that a synchronous relational transducer network manifests the* coordination pattern *if, for all possible initial instances $\boldsymbol{I} \in inst(\Upsilon_{db})$, whichever run we select, a point (ρ^i, t) and a communication relation R exist so that $\forall j \in N$ there is a predicate-level syncausality relation such that $(\rho^i, t) \overset{R}{\rightsquigarrow} (\rho^j, t')$.*

We call node i the *coordination master*. A pattern with a similar role has been named *broom* in [6].

Remark: The reader can now appreciate to which extent coordination was already "baked" inside the broadcasting synchronous specifications of Section 3. Note that broadcasting, in *rsync*, brings coordination. This is not true in asynchronous systems.

Intuitively, the coordination master is where the event occurs. If a broadcasting of (informative or non-informative) fact occurs, then such event will become *common knowledge* [8] among the nodes. On the contrary, if broadcasting is not occurring, common knowledge cannot be obtained and therefore, if the correct final outcome is still reached, this is obtained without coordination. That is, if at least a non-trivial configuration exists s.t. the coordination pattern doesn't manifest itself, we have coordination-freedom.

5 CALM in rsync Systems

The original version of the CALM principle is not satisfiable in *rsync* systems because a monotonic class of queries exists—*i.e., unchained queries*, introduced next—which is not coordination-free. Informally, a query is chained if every relation is connected through a join-path with every other relation composing the same query.

Definition 5.6. *Let body(q_R) be a conjunction of literals defining the body of a query q_R. We say that two different positive litteral occurrences $R_i(\bar{u}_i)$, $R_j(\bar{u}_j) \in$ body(q_R) are* chained *in q_R if either:*

- *$\bar{u}_i \cap \bar{u}_j \neq \emptyset$; or*
- *a third relation $R_k \in q_R$ different from R_i, R_j exists such that R_i is chained with R_k, and R_k is chained with R_j.*

Definition 5.7. *A query $\mathcal{Q}_{out}$ is said* chained *if, for every rule $q_R \in \mathcal{Q}_{out}$, each relation occurrence $R_i \in$ body(q_R) is chained with every other relation occurrence $R_j \in$ body(q_R).*

Remark: Nullary relations are not chained by definition.

Example 6. Assume two relations $R^{(2)}$ and $T^{(1)}$, and the following query $\mathcal{Q}_{out}$ returning the full R-instance if T is nonempty.

$$Q(u, v) \leftarrow R(u, v), T(\_).$$

The query is clearly monotonic. Let $\mathcal{T}$ be the following broadcasting UCQ-transducer program computing $\mathcal{Q}_{out}$.

Schema: $\Upsilon_{db} = \{R^{(2)}, T^{(1)}\}, \Upsilon_{com} = \{S^{(2)}, U^{(1)}\}, \Upsilon_{out} = \{Q^{(2)}\}$

Program: $S_{snd}(u, v) \leftarrow R(u, v).$

$U_{snd}(u) \leftarrow T(u).$

$Q_{out}(u, v) \leftarrow S(u, v), U(\_).$

Assume now we want to make the above transducer a hashing one. We have that, whichever key we chose, the related specification might be no more consistent. Indeed, consider an initial instance $\mathbf{I}$ and a set of keys spanning all the terms of S and U. Assume $\mathbf{I}$ such that $adom(I_R) \supset adom(I_T)$, and a network composed by a large number of nodes. In this situation, it may happen that a nonempty set of facts over R is hashed to a certain node i, while no fact over T is hashed to i. This because a constant may exist in $adom(I_R)$ that is not in $adom(I_T)$ and for which the hashing function returns a node i not returned by hashing any constant in $adom(I_T)$. Hence no tuple emitted to i will ever appear in the output, although they do appear in $\mathcal{Q}_{out}(\mathbf{I})$. Thus this transducer is not eventually consistent.

From the above example we can intuitively see that, for *rsync*, a final consistent result can be obtained without coordination only for queries that are chained and monotonic. That is, the following restricted version of the CALM conjecture holds for *rsync* systems:

Theorem 1 *A query can be parallelly computed by a coordination-free transducer network if it is chained and monotonic [11].*

We will leave for future works the investigation on whether every monotone and chained query is also coordination-free.

Remark: For the readers familiar with the works [2, 3] our result state that under the *rsync* model, a query is computable in a coordination-free way if monotonic and distributing over components.

6 Conclusions

In this paper the CALM principle is analyzed under synchronous and reliable settings. By exploiting CALM, in fact, we would be able to break the synchronous cage of modern parallel computation models, and provide pipelined coordination-free executions when allowed by the program logic. In order to reach our goal, we have introduced a new abstract model emulating BSP computation, and a novel interpretation of coordination with sound logical foundations in distributed knowledge reasoning. By exploiting such techniques, we have shown that the if direction of the CALM principle indeed holds also in *rsync* settings, but just for the subclass of monotonic queries defined as chained.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the calm-conjecture. In *PODS*, pages 64–75. ACM, 2014.

[3] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Datalog queries distributing over components. In *ICDT*. ACM, 2015.

[4] T. J. Ameloot, F. Neven, and J. Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, May 2013.

[5] I. Ben-Zvi and Y. Moses. Beyond lamport's *happened-before*: On the role of time bounds in synchronous systems. In N. A. Lynch and A. A. Shvartsman, editors, *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2010.

[6] I. Ben-Zvi and Y. Moses. On interactive knowledge with bounded communication. *Journal of Applied Non-Classical Logics*, 21(3-4):323–354, 2011.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[8] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 2003.

[9] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[10] M. Interlandi. Reasoning about knowledge in distributed systems using datalog. In *Datalog*, pages 99–110, 2012.

[11] M. Interlandi and L. Tanca. On the calm principle for bulk synchronous parallel computation. `arXiv:1405.7264`.

[12] M. Interlandi, L. Tanca, and S. Bergamaschi. Datalog in time and space, synchronously. In L. Bravo and M. Lenzerini, editors, *AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[14] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, Apr. 1984.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[17] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[18] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.