The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Parallel Algorithms for Operations on Multi-Valued Decision Diagrams

Guillaume Perez
Dept. of Computer Science
Cornell University, Ithaca, NY 14850 USA
gdp35@cornell.edu

Jean-Charles Régin
Universit Nice Sophia Antipolis, CNRS, I3S
UMR 7271, 06900 Sophia Antipolis, France
jcregin@gmail.com

Abstract

Multi-valued Decision Diagrams (MDDs) have been extensively studied in the last ten years. Recently, efficient algorithms implementing operators such as reduction, union, intersection, difference, etc., have been designed. They directly deal with the graph structure of the MDD and a time reduction of several orders of magnitude in comparison to other existing algorithms have been observed. These operators have permitted a new look at MDDs, because extremely large MDDs can finally be manipulated as shown by the models used to solve complex application in music generation. However, MDDs become so large (50GB) that minutes are sometimes required to perform some operations. In order to accelerate the manipulation of MDDs, parallel algorithms are required. In this paper, we introduce such algorithms. We carefully design them in order to overcome inherent difficulties of the parallelization of sequential algorithms such as data dependencies, software lock-out, false sharing, or load balancing. As a result, we observe a speed-up , i.e. ratio between parallel and sequential runtimes, growing linearly with the number of cores.

Introduction

Multi-valued Decision Diagrams (MDDs) have been extensively studied in the last ten years, notably by the team of J. Hooker, who recently published a book on this subject (Bergman et al. 2016). They are a compressed representation of a set (Bryant 1986) and are used to model the set of tuples of constraints. They are implemented in almost all constraint programming solvers and have been increasingly used to build models (Roy et al. 2016; Perez and Régin 2015; Andersen et al. 2007; Hadzic et al. 2008; Hoda, van Hoeve, and Hooker 2010; Bergman, van Hoeve, and Hooker 2011; Gange, Stuckey, and Van Hentenryck 2013; Cheng and Yap 2010). They can be constructed in several ways, from tables, automata, dynamic programming, etc.

MDDs have several advantages. They have a high compression efficiency. For instance, an MDD having 14,000 nodes and 600,000 arcs and representing 10^{90} tuples has been used to solve a music synchronization problem (Roy et al. 2016). Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression. The combinations of constraints represented by MDDs correspond to operations performed on MDDs. Conjunction of constraints is accomplished by intersecting MDDs and disjunction by union.

Recently, efficient operators such as union, difference negation, intersection, etc., have been designed (Perez and Régin 2015; 2016). These are based on algorithms dealing directly with the graph structure of the MDD instead of function decomposition, which requires reaching the leaves of a branch as Shannon or Bryant did (Bryant 1986). By explicitly working on the structure, they dramatically accelerated the algorithm and observed a time reduction of several orders of magnitude in comparison to other existing algorithms. The design of efficient algorithms has permitted a new look at MDDs, because extremely large MDDs can finally be manipulated now. One successful application of these ideas is the modelling of the maxOrder problem proposed by Pachet in his ERC project Flow Machine (Pachet 2011). Perez and Régin (Perez and Régin 2015) solved the problem by defining two MDDs, computing their difference, applying the result 16 times, and performing 16 intersections. The time to compute the final MDD is equivalent to that required for the algorithm of Pachet's group, and 52GB of memory was needed to represent it.

However, very large MDDs imply long times for performing operations even if these operations have a low complexity. For example, in a recent study, more than 2,000 seconds are needed for performing only two intersections (Perez and Régin 2017). Thus, parallelism seems the only one way to reduce these times.

Using a parallel CP solver is not an adequate answer because classical method, such as work-stealing (Jaffar et al. 2004; Michel, See, and Hentenryck 2009; Chu, Schulte, and Stuckey 2009) or embarrassingly parallel search (Régin, Rezgui, and Malapert 2013) requires to duplicate some parts of the problem which is not possible with MDDs requiring tens of gigabytes of memory. Therefore, parallel versions of the sequential algorithms implementing the operators are needed.

In this paper, we propose to parallelize the sequential algorithms implementing operations for MDDs. More precisely, we design parallel versions of the algorithms proposed in (Perez and Régin 2015), that is the reduction algorithm, which merge equivalent nodes, and the generic apply algorithm from which the other operators are built.

The parallelization of sequential algorithms usually encounters several difficulties such as data dependencies, software lock-out, false sharing, or load balancing. We will explain how the algorithms we propose overcome these difficulties. We will focus more particularly on the last two which become very penalizing in practice when more than four cores are involved. Notably, we will give parallel algorithms without any software lock-out that are as simple as their sequential version. We will present results showing an acceleration growing linearly with the number of cores and so proving that these difficulties have been avoided.

Note that the parallel algorithms we propose can also be used for minimizing an acyclic automaton.

The paper is organized as follows. First, we present some related works. Then, we recall some definitions about parallelism and MDDs. Next, we detail the parallelization of the reduction operator. Afterwards, we consider the APPLY algorithm and detail its parallel version. Finally, we give some experimental results for different sizes of MDDs and involving different number of cores, and we conclude.

Related Work

The processing in parallel of MDDs or automata has been well studied and is still a hot topic Bergman et al. introduced a parallel B&B search that branches on nodes in the MDDs instead of branching on variable-value pairs as it is done in conventional search methods (Bergman et al. 2014). Other works focus on minimizing an automaton in parallel, using specific algorithms (Ravikumar and Xiong 1996; Tewari, Srivastava, and Gupta 2002) or by using the well-known map-reduce principles (Hedayati Somarin 2016; Dean and Ghemawat 2008). Finally for Binary Decision Diagrams, parallel creation and manipulation algorithms (Kimura and Clarke 1990; Stornetta and Brewer 1996) have been designed. These algorithms use global hash-tables and they are organized so that locks are only needed for these global hash tables and the global tree nodes. In addition, a thread safe unordered queue using asynchronous messages is required. These algorithms and their implementation are quite complex, define complex dedicated data structures and use locks.

Background

Parallelism

When a parallel program is correct, that is when race condition and deadlock issues[1] have been resolved, several other aspects must be taken into account to reach a good scalability. At least four difficulties that may impact that scalability can be identified:

- *Data dependencies*: it is a situation in which an instruction refers to the data of a preceding instruction. Thus, no program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order.
- *Software lock-out*: it is the issue of performance degradation due to the idle wait times spent by the CPUs in kernel-level critical sections.
- *Resource contention* and particularly *false sharing*: it is a conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal buses or external network devices. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line.
- *Load balancing*: it refers to the distribution of workloads across multiple computing resources, such as cores.

For convenience we will use the word *worker* to represent an entity performing computation. Usually it corresponds to a core.

Multi-valued Decision Diagram

The decision diagrams considered in this paper are reduced, ordered multi-valued decision diagrams (MDD), which are a generalization of binary decision diagrams (Bryant 1986). They use a fixed variable ordering for canonical representation and shared sub-graphs for compression obtained by means of a reduction operation. An MDD is a rooted directed acyclic graph (DAG) used to represent some multi-valued function $f : \{0...d-1\}^n \rightarrow true, false$. Given the n input variables, the DAG contains $n+1$ layers of nodes, such that each variable is represented at a specific layer of the graph. Each node on a given layer has at most d outgoing arcs to nodes in the next layer of the graph. Each arc is labeled by its corresponding integer. The arc (u, a, v) is from node u to node v and labeled by a. All outgoing arcs of the layer n reach tt, the true terminal node (the false terminal node is typically omitted). There is an equivalence between $f(a_1, ..., a_n) = true$ and the existence of a path from the root node to the tt whose arcs are labeled $a_1, ..., a_n$.

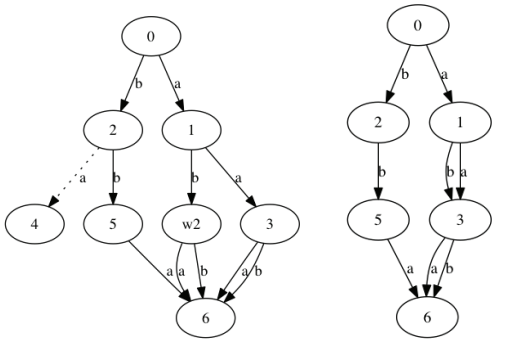
Let u be a node. The outgoing arcs of u are ordered by their labels and denoted by $\omega^+(u)$. The **signature** of u is denoted by $sig(u)$ and defined by the ordered list of ordered pairs defined from the outgoing arcs by removing the first element of the triplet representing the arc. Thus, $(l_i, u_j) \in sig(u) \Leftrightarrow (u, l_i, u_j) \in \omega^+(u)$. The neighbors of u are the nodes u_j s.t. $\exists (u, l_j, u_j) \in \omega^+(u)$. When considering u it is equivalent to deal with $\omega^+(u)$ or $sig(u)$.

Parallel Reduction

The reduction of an MDD consists in removing nodes that have no successor and merging equivalent nodes, i.e. nodes having the same set of neighbors associated with the same labels. This means that only nodes of the same layer can be merged. In addition, two nodes can be merged iff they have the same signature. Figure 1 gives an example of reduction.

The main difficulty is to identify nodes having the same signature. The PREDUCE algorithm (Perez and Régin 2015) improved previous algorithms that checked for each node whether it can be merged with another node or not. It works

[1] Race conditions depends on the sequence or timing of processes or threads for it to operate properly. A deadlock is a state in which each member of a group is waiting for some other member to take action (Coulouris, Dollimore, and Kindberg 2005).



MDD before reduction MDD after reduction

Figure 1: Reduction of an MDD. Nodes without successors are deleted (e.g. node 4). Equivalent nodes are merged (e.g. nodes 3 and w2).

per layer and groups the nodes having the same suffix of signatures into packs[2]. Nodes that remain in the same pack with their entire signature, and not only a suffix, can be merged together. The worst-case time complexity is bounded by the sum of the size of common suffix of the nodes.

Efficient parallelization of this algorithm is not trivial, thus a simplified version of the sequential algorithm is considered first. Then, a more complex algorithm is presented to fit the best complexity of the sequential algorithm.

Parallel Sort

The identification of nodes having the same signature can be simply performed by sorting the node according to their signature. Since nodes and labels are integers, a linear sort algorithm can be used. We propose to consider a radix sort.

We reproduce the presentation in (Cormen, Leiserson, and Rivest 1990). Consider that each element in the q-element array A has δ digits, where digit 1 (resp. δ) is the least (resp. most) significant digit. The radix sort algorithm consists of calling for $r = 1..\delta$ a stable sort to sort array A on digit r. The counting sort can be used as a stable sort. Counting sort assumes that each of the q input elements is an integer in $[0, k]$, for some integer k. It determines, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 17 elements less than x, then x belongs in output position 18. When several elements have the same value we distinguish them by their order of appearance in order to have a stable sort. Thus, the time complexity of the radix sort is $\delta O(q + k)$.

The parallel radix sort with w workers (Zagha and Blelloch 1991). It uses a parallel counting sort as stable sort. Let V be a vector of q elements. The parallel counting sort splits V into w subvectors, one for each worker. Then each worker applies a counting sort on its subvector. Finally, the workers put the nodes in their new position.

[2]The algorithm normally works with prefixes, but it can be straightforwardly adapted to deal with suffixes.

Example. We propose to detail the parallel radix sort for a vector of nodes of the MDD. We assume that there each node has only one neighbors and one label. For the sake of clarity we represent an ordered pair (l_i, u_j) by ij, i.e. (l_0, u_1) is written 01.

Consider the following vector of nodes, the second line gives the index of the node, the third line is associated with the signatures.

a					b					
0	1	2	3	4	5	6	7	8	9	V
10	**00**	**11**	**11**	**00**	**10**	**10**	**01**	**11**	**01**	sig

Using two workers (a and b), we can split the vector into two independent parts and apply a counting sort. The two parts are $[0, 4]$ and $[5, 9]$, b is always after a in order to avoid collision. The first step of the radix sort considers the rightmost digit. Let $a\#_i$ (resp. $b\#_i$) be the number of i counted by worker a (resp. b). These numbers are computed by traversing the values. This is the counting step of the counting sort. We obtain:

$a\#_0 = 3$	$a\#_1 = 2$	$b\#_0 = 2$	$b\#_1 = 3$

Then we determine the global indices of each digit by workers, let ia_r (resp. ib_r) be the position in the resulting vector of the first value of the elements of part a (resp. b) whose current digit is r. We have $ia_0 = 0$; $ib_0 = ia_0 + a\#_0 = 3$; $ia_1 = ib_0 + b\#_0 = 3 + 2 = 5$ and $ib_1 = ia_1 + a\#_1 = 7$. When there are more than two workers the same principle applies: the information of the previous worker is used for the current worker. We have:

$ia_0 = 0$	$ia_1 = 5$	$ib_0 = 3$	$ib_1 = 7$

This step, denoted by the cumulative step, can also be performed in parallel. Each worker receives a set of values ($[0..k]$ is divided into w subranges) and performs the computations for its set of values. Each worker j computes the number of time each of its value is taken and $cs(j)$ the cumulative sum of these numbers. Then, we compute for each worker j the sum of the cumulative sum of the previous workers: $scs(j) = \sum_{i=1}^{j-1} cs(i)$. This can be globally done in $O(w)$. From these scs values, each worker computes the global indices of its values.

Using these positions, the workers a and b can independently build the global vector without any collision and thus without lock. So this last step, named the position step, can also be performed in parallel by assigning to each vector its initial subvector of elements. For example, a puts the value 10 of node 0 in position $ia_0 = 0$ then, it increments ia_0, so $ia_0 = 1$, then it puts the value 00 of node 1 in position $ia_0 = 1$ and increments it again, etc. The global vector is:

0	1	4	5	6	2	3	7	8	9
10	**00**	**00**	**10**	**10**	**11**	**11**	**01**	**11**	**01**

The same process has to be applied to the second digit in order to sort the nodes. We finally obtain:

a					b				
1	4	7	9	0	5	6	2	3	8
00	00	01	01	10	10	10	11	11	11

Complexity. For each worker, the counting step[3] is in $O(q/w)$, the cumulative step is in $O(k/w)$ and the position step is in $O(q/w)$. Thus, the overall time complexity of the parallel counting sort is in $O(q/w + k/w)$. The parallel radix sort considers all digits of the signatures, thus its time complexity is in $\delta \times O(q/w+k/w)$, which is more than the sum of common suffix of the PREDUCE algorithm. For instance, consider the following signatures of nodes: 0010, 1101, 1012, 0113, 1004, 0015, 0116. The algorithm will process for each node all four digits while the sequential algorithm will process only the least significant one.

In order to remedy this issue, the PREDUCE algorithm works with packs, which are common suffixes. Two nodes belong to the same pack when their signature have the same suffix defined by the pack. Only nodes in the same pack can be merged. So, if at a moment, the signature of a node u has a different digit value for a given position than any other node of its current pack, then u cannot be merged with any node and it can be ignored. Nodes that remain in the same pack will be merged at the end.

Working by digits such as the radix sort is similar as working by common suffixes. Thus, we propose to apply the same mechanism for the parallel algorithm by introducing the pack notion.

Parallel PREDUCE

Packs and leaders. For the sake of clarity, node at position p in the layer is denoted by u_p. Consider the iteration r of the radix sort. A pack is a set of nodes having signatures sharing the same digits from 1 to r (1 being the least significant digit). This means that for any iteration all nodes of a pack are consecutive. A particular node of a pack can be identified: its leader. The leader is the node of the pack having the smallest position. Then, for any node u_p with $p > 0$, either u_p is a leader and u_{p-1} belongs to another pack than u_p, or u_p is not a leader and u_{p-1} belongs to the same pack as u_p. Therefore, packs can be deduced from leader nodes.

At the beginning of the algorithm, all nodes are in the same pack, and the pack leader is the first node. For the previous example, all nodes have 0 as leader (first line):

0	0	0	0	0	0	0	0	0	0	ldr
0	1	2	3	4	5	6	7	8	9	V
10	**00**	**11**	**11**	**00**	**10**	**10**	**01**	**11**	**01**	sig

Then, packs are refined depending on the current digit values, because nodes in a pack have signatures having the same suffix. When a new pack is created, its leader is the node having the smallest index of the nodes in the pack. The identification of packs and leaders is performed after the position step of the counting sort.

For the previous example if we have one worker then the initial pack is split into two packs when considering the least significant digit. Node 2 is the leader of the new pack.

0	0	0	0	0	2	2	2	2	2	ldr
0	1	4	5	6	2	3	7	8	9	V
10	**00**	**00**	**10**	**10**	**11**	**11**	**01**	**11**	**01**	sig

[3]The count array can be initialized by traversing the elements to set the count of their values to 0 after the different steps.

We precisely define when a pack is created for the iteration r. Pack are defined by their leader. Node u_0 is always a leader. Consider $p > 0$. First, we assume that u_{p-1} has already been set. Node u_p is a leader iff u_{p-1} was in the same pack at the previous iteration and the r^{th} digit of the signature of u_p and u_{p-1} are different, or u_{p-1} was not in the same pack at the previous iteration.

For the second iteration of the previous example we obtain the following result:

1	1	7	7	0	0	0	2	2	2	ldr
1	4	7	9	0	5	6	2	3	8	V
00	00	01	01	10	10	10	11	11	11	sig

Nodes having the same leader at the end must be merged.

Parallel computation of packs and leaders. When workers are introduced, a problem arises because when u_p is set, it is possible that u_{p-1} has not been set. After the positioning step of the counting sort, each worker will define the leader of its part by checking whether each element is a leader or not. Then, a problem arises for consecutive nodes that are handled by different workers, known as junction.

For instance in the previous example, after the first counting sort, nodes are ordered as follows: $0, 1, 4, 5, 6, 2, 3, 7, 8, 9$. Then, worker a considers the five first nodes, and worker b the five last nodes. That is, a deals with $0, 1, 4, 5, 6$. Since all signatures have the same first digit (0), then 0 is the leader of this group. In addition node at position 0 is always a leader so 0 is a global leader. Worker b deals with $2, 3, 7, 8, 9$. Since all signatures have the same first digit (1), then 2 is the leader of this group. However, there is not enough information to deduce that 2 is a global leader. The leadership of node 2, at position 5, will be deduced by comparing the previous pack of 2 and the current digit of its signature of 2 with the data of node at position $5 - 1 = 4$, which is not necessary available. The $w - 1$ junctions will be studied when all the workers have finished their work. The relation between nodes and leader can be maintained by using a union find data structure (Tarjan 1975). Each tree represents nodes of a pack. By performing merge according to decreasing indices, the depth of the tree can never be more than 2, so the time complexity of these operations is globally linear, that is in $O(1)$ per node.

At last and for reducing the time complexity in practice, if at any moment a pack contains only one element, then it is removed from the vector of nodes and ranges of indices of workers are accordingly redefined.

Algorithm. Algorithm 1 is a possible implementation. It works by layer. For a layer, the successive digits of the signatures are considered from the least significant digit. The vector of nodes is partitioned into as many parts as workers, each part having the same number of elements. Each worker w_j performs a counting sort on this set of nodes V_j and puts the result in V (Function COUNTINGSORT(V_j, r, V)). Then, each worker computes the leader of its part of the nodes (Function COMPUTELEADERS(V_j, r, V)). The junctions are

Algorithm 1 parallel reduce of an MDD.

```
paraReduce(mdd, W)
    // W is the set of workers
    for each i from n − 1 to 0 do
        V ← L[i], the set of nodes in layer i.
        δ ← max_{u∈V}(size of sig(u))
        for each r from 1 to δ while V ≠ ∅ do
            Partition V into |W| parts: V_1, ..., V_|W|
            parallel for w_j ∈ W do
                w_j.initUnionFind(V_j, r, V)
            parallel for w_j ∈ W do
                w_j.countingSort(V_j, r, V)
            parallel for w_j ∈ W do
                w_j.computeLeaders(V_j, r, V)
            Define leaders for junctions
            Remove from V nodes in singleton packs
        in parallel Merge in L[i] nodes u ∈ V with its leader
```

managed and nodes that cannot be merged are removed from V. Note that the internal data structures are managed at the beginning of each loop. Finally, nodes that remain in V are merged in $L[i]$ in parallel by partitioning V and by keeping only the leaders. If the leader of a node in V_j is not managed by w_j, then the node in V_j with leftmost index becomes a local leader and nodes in V_j are merged with it instead of the global leader. Then, local leaders are merged to global leaders. Note that there is at most one local leader per worker, so it does not impact the time complexity, which remains the same as the one of the pReduce algorithms.

Parallelization difficulties. This algorithm overcomes the four difficulties of the parallelization of a sequential algorithm. The data dependencies are controlled by working by layer. There is no software lock. Workers get the information from different places, i.e. the V_i parts, and write the result in different positions and these two actions are performed separately and so the chance of false sharing is reduced. At last, the vector of nodes is always split in equal parts, thus the workload is well balanced.

Discussion

The time complexity of the parallel reduction algorithm is the same as the parallel radix sort involving w workers. It is in $O(\delta(q/w + k/w))$ (1) where q is the number of elements and k the greatest possible value and δ the number of digits. If $q = O(k)$ or if k is clearly smaller than q then the complexity is in $O(\delta q/w)$ (2).

If k is clearly greater than q, then there are two ways to reduce the complexity. First, a different algorithm (Perez and Régin 2015) can be used. Indeed, the counting sort can be relaxed because the reduction algorithm does not require to sort the elements. Instead it searches to identify elements having the same digits. Thus, we can use an algorithm similar to the counting sort whose complexity is based only on the number of values, and not on their range, i.e. $[0..k]$. This algorithm uses A, an array of values ranging from $[0..k]$ initially at zero. Like the counting sort, it traverses the elements to count the number of times a value appears. However at the same time it builds the list of taken values and it uses this list to define the position instead of the range $[0..k]$. In this way, the ordering between the values is lost, but the algorithm still groups together the elements having the same value, without traversing the range $[0..k]$. So it can be used for our purpose. Unfortunately this algorithm is quite complex to parallelize because it requires to use a local queue per worker and the compare-and-swap instruction to be correct and efficient. The detail of this algorithm is out of the scope of this paper but can be found in (Perez 2017).

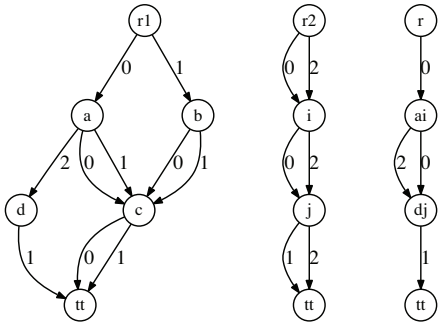


Figure 2: Intersection of two MDDs.

Second, the same algorithm is used but the number of digits of the signature is increased. If k is greater than q then we can reduce its size by splitting the number k into several digits. The number k can be written with $\log_m(k)$ digits in base m. So, we can express the overall time complexity (1) by $\log_m(k)O(\delta(q/w + m/w))$. By using $m = 256$ we have $\log_{256}(k)O(\delta(q/w + 256/w)) \leq 4O(\delta(q/w + 256/w))$ for $k \leq 2^{32}$. This complexity is equals to $4O(\delta q/w)$ if $q \geq 256$. So, in practice (i.e. $k \leq 2^{32}$), by writing k in base 256 we multiply the initial complexity (2) by at most a factor of 4.

Parallel Apply

Perez and Régin introduced an efficient apply algorithm (Perez and Régin 2015) in order to define operations between MDDs. From the MDDs mdd_1 and mdd_2 it computes $mdd_r = mdd_1 \oplus mdd_2$, where $\oplus$ is union, intersection, difference, symmetric difference, complementary of union and complementary of intersection[4].

Their algorithm proceeds by associating nodes of the two MDDs. Each node u of the resulting MDD is associated with a node u_1 of the first MDD and a node u_2 of the second MDD. So, each node of the resulting MDD can be represented either by an index, or by a pair (u_1, u_2). First, the root is created from the two roots. Then, layers are successively built. From the nodes of layer $i - 1$, nodes of layer

[4]Unlike Perez and Régin, the complementary of an MDD M is computed by making the difference between the universal MDD and M. This avoids the need of a dedicated algorithm.

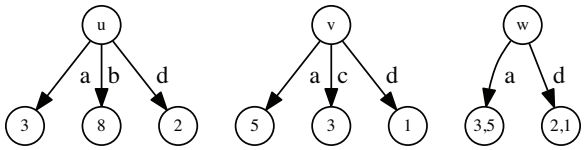


Figure 3: Intersection of two nodes

i are built as follows. For each node $u = (u_1, u_2)$ of layer $i-1$, arcs outgoing from nodes u_1 and u_2 and labeled by the same value l are considered. Note that there is only one arc leaving a node u with a given label. Thus, there are four possibilities depending on whether there are v_1 and v_2 such that (u_1, l, v_1) and (u_2, l, v_2) exist or not. The action that is performed for each of these possibilities defines the operation that is performed for the given layer. For instance, a union is defined by creating a node $v = (v_1, v_2)$ and an arc (u, l, v) each time one of the arcs (u_1, l, v_1) or (u_2, l, v_2) exists. An intersection is defined by creating a node $v = (v_1, v_2)$ and an arc (u, l, v) when both arcs (u_1, l, v_1) and (u_2, l, v_2) exist. Figure 3 gives an example of the intersection of two nodes. Thus, these operations can be simply defined by expressing the condition for creating a node and an arc. More details can be found in (Perez and Régin 2015). We assume that Function BUILDARCS&NODES implements this mechanism and returns the array of created nodes with its length.

After each layer, the algorithm merges equivalent ordered pairs (x_1, x_2), because a lot of them can be created. For instance, consider a node u_1 of the first MDD at layer i with an arc (u_1, l, v_1) and v_2 a node of the second MDD at layer $i+1$. Then, every arc of the second MDD labeled by l and reaching v_2 will provoke the creation of the ordered pair (v_1, v_2). Function MERGEORDEREDPAIRS is in charge of this task. At last, the computed MDD is reduced.

Parallelization. Function BUILDARCS&NODES can be easily parallelized by splitting the nodes of the layer according to the number of workers that are involved. The returned arrays must be merged into one an array of created nodes. This can be done in parallel, by using the length of each array to distribute the workload among the workers. Since, an ordered pair of node can be seen as a signature containing two digits, algorithm PARAREDUCE can be used for implementing Function MERGEORDEREDPAIRS.

Algorithm. Algorithm 2 is a possible implementation of the parallel version of APPLY.

Complexity. Let mdd_1 be the first MDD, mdd_2 be the second, n_1 (resp. n_2) be the number of nodes of mdd_1 (resp. mdd_2), and d be the maximum number of labels of a layer. For any layer, for each node of mdd_1 of this layer and for each node of mdd_2 of this layer, a node may be built. In addition, this created node may have d outgoing arcs. Thus, the complexity of the sequential APPLY algorithm is in $O(n_1 n_2 d)$. The time complexity of the parallel version of APPLY can be divided by the number of workers.

Algorithm 2 Parallel Apply.

```
APPLY(mdd1, mdd2, op, W): MDD
    // W is the set of workers
    // We assume that each node can access its signature
    Define mdd s.t. L[i] is the set of nodes in layer i.
    root ← CREATENODE(root(mdd1), root(mdd2))
    L[0] ← {root}
    for each i ∈ 1..n do
        Partition L[i−1] into |W| parts: V1, ..., V|W|
        L[i] ← ∅
        parallel for wj ∈ W do
            arrj ← wj.BUILDARCS&NODES(Vj, op)
        Build in parallel L[i] from arr1, ..., arr|W|
        parallel for wj ∈ W do
            wj.MERGEORDEREDPAIRS(L[i])
    PARAREDUCE(mdd, W)
    return mdd
```

Parallelization difficulties. The data dependencies are controlled by working by layer. There is no explicit software lock, but objects are created therefore it is important to manage the memory per worker and independently from the others. Some false sharings have been observed (See Experiments) because the array of nodes per layer is shared. It was solved by postponing as much as possible the access to common cache (Sutter 2009).

Experiments

All the experiments have been made on a Dell machine having four E7- 4870 Intel processors, each having 10 cores with 256 GB of memory, 16 memory channels and running under Scientific Linux. Each tested combination is followed by a reduction. The dedicated algorithms is used when there are very large values (See Discussion in Section Parallel Reduction).

Real instances. First, we have run the instances from (Perez and Régin 2015; 2017). The first one named MaxOrder consists of intersecting and applying a symetric difference between two very large MDDs, the result contains more than 1 million of nodes and 200 millions of arcs. The second one is named Dispersion, consists of the intersection of three MDDs representing different sum functions. Curves from Figure 4 show the speed-up, i.e. ratio between parallel and sequential runtimes, for these two problems. As we can see, for these instances, the speed-up is significant. Thanks to parallel algorithms, we are able to close previously hard problems in seconds.

Random instances. We have also tested many random MDDs generated by fixing a lower and upper bound on the number of nodes on each layer and then building the outgoing arcs with respect to some probabilities. This allow to handle the different sizes of MDDs, like small ones with high density, big ones with low density, etc.

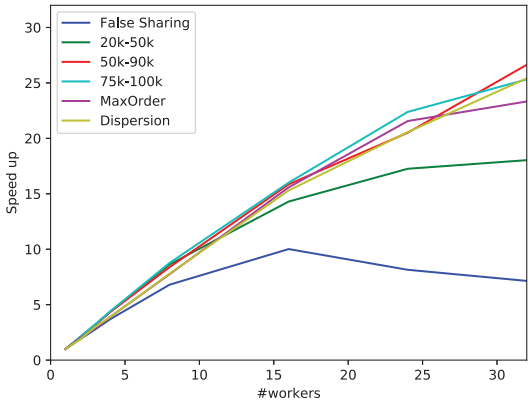


Figure 4: Relative speed-ups according to the number of workers.

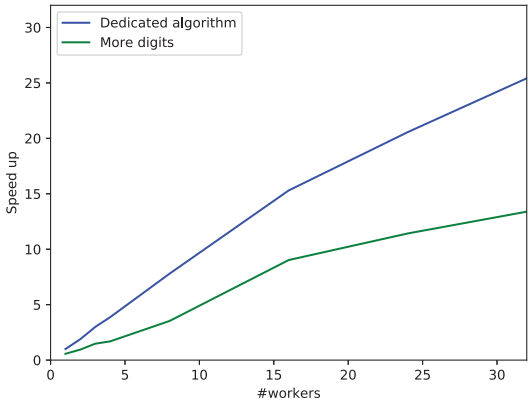


Figure 5: Very large values management.

We have run instances with a high arc density (> 0.60). The number of variables does not have an impact on the results. Figure 4 shows the application of the parallel intersection between two MDDs. The number of nodes by layer varies from 20k to 100k and three main curves are presented:

- For 50k-90k and 75k-100k, the speed-up is a straight line from 1 to 16 cores with 1 as coefficient, which corresponds to the number of available memory channels. Then, it is a line with 0.65 as coefficient. Note that this comes from the fact that the resulting MDDs are bigger than 70 GB and so the memory channels are saturated.
- For 20k-50k, the speed-up ratio is lower, because the amount of work is low and thus the time is hardly reducible.

Thanks to these speed-ups, we are able to build MDDs representing very large constraints in seconds or minutes while hours was required before. This implies that we can reinforce our model by intersecting constraints as a preprocess before running the search for solutions.

Remarks. The sequential algorithm is less than 10% more efficient than the parallel version running with one worker. When some operations are done in less than one second, our parallel algorithms are slower. An explanation can be the time needed to create the required memory.

False Sharing. The blue line shows the False sharing problem we had, as we can see, the algorithm scales until eight workers, then fails to keep a good speed-up, and finally loses efficiency with the growing number of workers.

Dedicated algorithm or more digits? Figure 5 compares the complex algorithm (See Discussion in Section Parallel Reduction) and the method of the augmentation of digits for the parallel reduction when the values are very large. The dedicated algorithm outperforms the second method by at most a factor of 2.5.

Laptop. On a simple Macbook pro 2013 using four cores, we observe the following speed-up:

40 variables	20 values	40 values
5k-100k nodes/layer	3.78	3.37
10k-200k nodes/layer	3.67	3.84

Conclusion

A parallel version of the reduction and the apply algorithms for MDDs have been presented. These algorithms do not need any complex data structure and are simple to implement. They overcome the common difficulties encountered when parallelizing a sequential algorithm. Experimental results show that they accelerate the sequential algorithms by a linear factor according to the number of involved workers.

References

Andersen, H. R.; Hadzic, T.; Hooker, J. N.; and Tiedemann, P. 2007. A constraint store based on multivalued decision diagrams. In *CP 2017 - Principles and Practice of Constraint Programming, 13th International Conference, Providence, USA, Proceedings*, 118–132.

Bergman, D.; Ciré, A. A.; Sabharwal, A.; Samulowitz, H.; Saraswat, V. A.; and van Hoeve, W.-J. 2014. Parallel combinatorial optimization with decision diagrams. In *CPAIOR 2014 - Integration of AI and OR Techniques in Constraint Programming, 11th International Conference, Cork, Ireland, Proceedings*, 351–367.

Bergman, D.; Ciré, A. A.; van Hoeve, W.-J.; and Hooker, J. N. 2016. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer.

Bergman, D.; van Hoeve, W. J.; and Hooker, J. N. 2011. Manipulating MDD relaxations for combinatorial optimiza-

tion. In *CPAIOR 2011 - Integration of AI and OR Techniques in Constraint Programming, 8th International Conference, Berlin, Germany, Proceedings*, 20–35.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* 35(8):677–691.

Cheng, K. C. K., and Yap, R. H. C. 2010. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints* 15(2):265–304.

Chu, G.; Schulte, C.; and Stuckey, P. J. 2009. Confidence-Based Work Stealing in Parallel Constraint Programming. In *CP 2009 - Principles and Pratice of Constraint Programming, 15th International Conference , Lisbon, Portugal, Proceedings*, 226–241.

Cormen, T.; Leiserson, C.; and Rivest, R. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Coulouris, G. F.; Dollimore, J.; and Kindberg, T. 2005. *Distributed systems: concepts and design*. pearson education.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1):107–113.

Gange, G.; Stuckey, P. J.; and Van Hentenryck, P. 2013. Explaining propagators for edge-valued decision diagrams. In *CP 2013 - Principles and Practice of Constraint Programming, 19th International Conference, Uppsala, Sweden, Proceedings*, 340–355. Springer.

Hadzic, T.; Hooker, J. N.; O'Sullivan, B.; and Tiedemann, P. 2008. Approximate compilation of constraints into multivalued decision diagrams. In *CP 2008, 14th International Conference, Sydney, Australia, Proceedings*, 448–462.

Hedayati Somarin, I. 2016. *DFA Minimization Algorithms in Map-Reduce*. Ph.D. Dissertation, Concordia University.

Hoda, S.; van Hoeve, W.-J.; and Hooker, J. N. 2010. A systematic approach to MDD-based constraint programming. In *CP 2010 - Principles and Practice of Constraint Programming, 16th International Conference, St. Andrews, Scotland, Proceedings*, 266–280.

Jaffar, J.; Santosa, A. E.; Yap, R. H. C.; and Zhu, K. Q. 2004. Scalable Distributed Depth-First Search with Greedy Work Stealing. In *16th IEEE International Conference on Tools with Artificial Intelligence*, 98–103.

Kimura, S., and Clarke, E. M. 1990. A parallel algorithm for constructing binary decision diagrams. In *ICCD 1990 - IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, USA, Proceedings*, 220–223.

Michel, L.; See, A.; and Hentenryck, P. V. 2009. Transparent Parallelization of Constraint Programming. *INFORMS Journal on Computing* 21:363–382.

Pachet, F. 2011. Flow machine project, ERC FP/2007-2013 grant agreement n 291156.

Perez, G., and Régin, J.-C. 2015. Efficient operations on MDDs for building constraint programming models. In *IJCAI 2015 - Twenty-Fourth International Joint Conference on Artificial Intelligence, Buenos Aires, Argentina, Proceedings*, 374–380.

Perez, G., and Régin, J.-C. 2016. Constructions and in-place operations for MDDs based constraints. In *CPAIOR 2016 - Integration of AI and OR Techniques in Constraint Programming, 13th International Conference, Banff, Canada, Proceedings*, 279–293.

Perez, G., and Régin, J.-C. 2017. MDDs are efficient modeling tools: An application to some statistical constraints. In *CPAIOR 2017 - Integration of AI and OR Techniques in Constraint Programming, 14th International Conference, Padua, Italy, Proceedings*, 30–40.

Perez, G. 2017. *Decision Diagrams: Constraints and Algorithms*. Ph.D. Dissertation.

Ravikumar, B., and Xiong, X. 1996. A parallel algorithm for minimization of finite automata. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, 187–191. IEEE.

Régin, J.-C.; Rezgui, M.; and Malapert, A. 2013. Embarrassingly parallel search. In *CP 2013 - Principles and Practice of Constraint Programming, 19th International Conference, Uppsala, Sweden, Proceedings*, 596–610.

Roy, P.; Perez, G.; Régin, J.-C.; Papadopoulos, A.; Pachet, F.; and Marchini, M. 2016. Enforcing structure on temporal sequences: The Allen constraint. In *CP 2016, 22nd International Conference, Toulouse, France, Proceedings*, 786–801.

Stornetta, T., and Brewer, F. 1996. Implementation of an efficient parallel BDD package. In *33st Conference on Design Automation, Las Vegas, USA, Proceedings*, 641–644.

Sutter, H. 2009. Eliminate false sharing. *Dr. Dobb's*.

Tarjan, R. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery* 22:215–225.

Tewari, A.; Srivastava, U.; and Gupta, P. 2002. A parallel DFA minimization algorithm. In *International Conference on High-Performance Computing*, 34–40. Springer.

Zagha, M., and Blelloch, G. E. 1991. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 712–721. ACM.