On Fast Timing Closure: Speeding Up Incremental Path-Based Timing Analysis with MapReduce

Tsung-Wei Huang* and Martin D. F. Wong‡
*twh760812@gmail.com, ‡mdfwong@illinois.edu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—Incremental path-based timing analysis (PBA) is a pivotal step in the timing optimization flow. A core building block analyzes the timing path-by-path subject to a critical amount of incremental changes on the design. However, this process in nature demands an extremely high computational complexity and has been a major bottleneck in accelerating timing closure. Therefore, we introduce in this paper a fast and scalable algorithm of incremental PBA with *MapReduce* – a recently popular programming paradigm in big-data era. Inspired by the spirit of MapReduce, we formulate our problem into tasks that are associated with keys and values and perform massively-parallel map and reduce operations on a distributed system. Experimental results demonstrated that our approach can not only easily analyze huge deisgns in a few minutes, but also quickly revalidate the timing after the incremental changes. Our results are beneficial for speeding up the lengthy design cycle of timing closure.**

I. Introduction

The lack of accurate and fast algorithms for incremental path-based timing analysis (PBA) has been recently pointed out as a major weakness in the timing optimization flow [4]. Among timing analysis applications, timing-driven operations are imperative for the success of optimization flows, such as logic synthesis, placement, and routing. Optimization transforms change the design and therefore have the potential to significantly affect timing information. The timer must reflect to such changes and update timing information incrementally and accurately in order to ensure slack integrity and reasonable turnaround time and performance. However, such process requires an extremely high complexity in particular when path-specific update such as common-path-pessimism removal (CPPR) is taken into account [7], [9]. A high-quality incremental PBA tool is definitely positive for speeding up timing closure.

Unfortunately, current literatures are still short of novel ideas for fast incremental PBA. As pointed out by 2015 TAU timing analysis contest, algorithms that highlight multi-threaded and massively-parallel accelerations are strongly encouraged [4]. Nonetheless, parallelizing incremental PBA is a tough challenge primarily because a path can be prototypically various in an incremental environment. For instance, a path can exhibit arbitrary lengths and span different logical cones and physical boundaries, while such environmental properties are hard to be quantified. Computations in this way are typically difficult to be issued in parallel. Although a few prior works claimed to have a solution, the results are usually either compromised with accuracy or runtime, or incapable of incremental processing [5], [12], [13]. These deficiencies severely restrain the capability of incremental PBA in the timing optimization flow.

As a consequence, we introduce in this paper a fast incremental PBA algorithm with MapReduce, a programming paradigm that was recently introduced by Google for big-data processing [8]. As shown in Figure 1, a MapReduce program applies parallel map operations to input tasks and generates a set of temporary key/value pairs. Parallel reduce operations are then applied to all values that are associated with the same key in order to collate the derived data properly. Users only program desired map and reduce functions while parallelization details such as work distribution and fault tolerance are automatically encapsulated in a MapReduce library [1], [2]. This programming paradigm inspires us to rethink the incremental PBA problem as a set of "map" and "reduce" operations. In other words, we transform the incremental PBA problem into tasks with keys and values that are solvable using massively-parallel map and reduce operations.

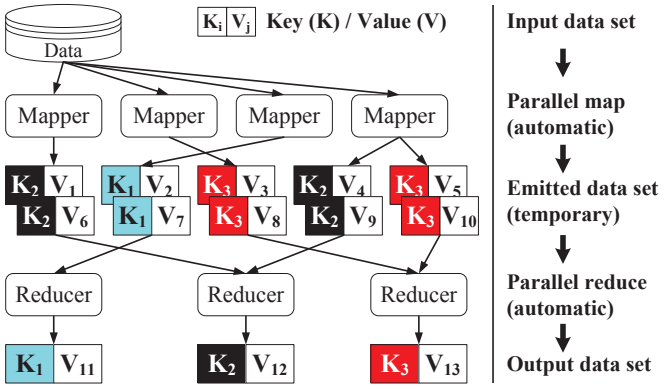

Figure 1. The execution flow of a MapReduce program.

Our contributions are summarized as follows. 1) We successfully investigated the applicability of MapReduce to accelerate incremental PBA. Our algorithm is very general in gaining massively-parallel computations, imposing no physical and logic constraints. 2) Our algorithm increases the productivity as designers can focus on timing-driven turnaround, leaving all hassle of parallelization details to the MapReduce library. 3) We have seen a substantial speedup from the experimental results. On a large distributed system, millions of cells subject to a critical amount of incremental updates can be easily analyzed quickly and accurately. These features all add up to faster design cycle of timing optimization, which can be beneficial for speeding up the lengthy signoff timing closure.

II. Incremental Path-Based Timing Analysis

Static-timing-analysis (STA) is a crucial step in verifying the expected timing behaviors of an integrated circuit [6]. During the STA, both graph-based timing analysis (GBA) and path-based timing analysis (PBA) are used. GBA performs linear scan on the circuit graph and estimates the worst timing quantities at each endpoint. GBA is very fast but the results are too pessimistic. Thus, PBA is often performed after GBA to remove unwanted pessimism. Starting from a negative endpoint, a core PBA procedure peels out a set of paths in non-increasing order of criticality and applies path-specific timing update such as CPPR and advanced-on-chip-variation (AOCV) derating to each of these paths [7]. By analyzing the path with reduced pessimism, many timing violations can be waived which in turn tells better timing signoff.

In practice, timing-driven optimizations are ubiquitous along with the entire design flow. Practical PBA tools must be adaptive to incremental environment. In other words, whenever the design experiences

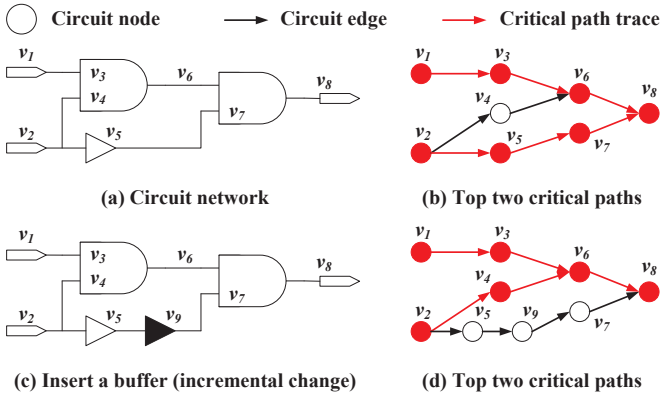

Figure 2. The incremental PBA problem. (a) Initial circuit network. (b) Top two critical paths on $< v_1, v_3, v_6, v_8 >$ and $< v_2, v_5, v_7, v_8 >$. (c) Optimization transform by buffer insertion. (d) Top two critical paths on $< v_1, v_3, v_6, v_8 >$ and $< v_2, v_4, v_6, v_8 >$.

optimization transforms (e.g., gate sizing, wire sizing, etc.), the timing information needs to be revalidated in order to guarantee the slack integrity and legal optimization turnaround. A practical example of the incremental PBA problem is demonstrated in Figure 2. In (a)–(b), the timer is queried about the top two critical paths. In (c)–(d), an optimization transform changes the design via a buffer insertion. The timer updates the timing and reports the new top two critical paths. Despite the importance of incremental PBA in the timing optimization flow, few researches have been reported so far [3], [4], [9]. We summarize the reason as follows.

- In contrast to GBA, PBA demands much higher complexity. A careless incremental PBA algorithm might end up with repeatedly enumerating all paths in a design.
- Efficient incremental PBA requires more complicated data manipulations. A specialized algorithm is necessary.
- Incremental PBA on a distributed system was barely discovered. Programmers typically suffer from a unacceptably large amount of development efforts for the hard-crafting of distributed/parallel infrastructure.

To sum up, an incremental PBA tool that supports massive parallelism, high scalability, and general flexibility is in demand when we move to multi-core era. As reported in [14], nowadays up to 40% of the design flow are normally spent on timing optimizations. If the runtime of incremental PBA can be significantly improved, the design cycle of timing optimization is able to be shortened substantially thereby making a breakthrough in timing signoff. As a result, researchers must continue to provide viable parallel solutions along with the rapid evolution of the computational power.

III. Problem Formulation

The circuit network is input as a directed-acyclic graph $G = \{V, E, T\}$, where V is the pin set of circuit elements, E is the edge set specifying pin-to-pin connections, and T is the set of timing tests to be analyzed. Each edge e is associated with a tuple of earliest and latest delays. A path is an ordered sequence of nodes or edges and the path delay is the sum of delays through all edges. In this paper, we are in particular emphasizing on the *data path*, which is defined as a path from either the primary input pin or the clock pin of a launching flip-flop (FF) to the data pin of a capturing FF. A test $t \in T$ is defined with respect to (w.r.t.) an FF as either hold or setup check on any data paths captured by this FF. The incremental PBA problem deals with the following online (i.e., input disclosure is prohibited) operations.

- *update_edge*(e, w): update the delay of the edge e to w.
- *remove_edge*(e): remove the edge $e \in E$ from the graph.
- *remove_node*(v): remove the node $v \in V$ from the graph.
- *insert_edge*(e): insert an edge $e \notin E$ into the graph.
- *insert_node*(v): insert a node $v \notin V$ into the graph.
- *report_path*(t, k): report the top k critical paths for the test t.
- *report_path*(k): report the top k critical paths across all tests T.

The above operations we have defined are in fact graph-level abstraction of the incremental PBA problem. In reality, optimization transforms are expressed as either gate-level, net-level, or pin-level operations [4]. For algorithmic purpose, these optimization transforms can be, without loss of generality, reflected as graph-level operations so as to compute the timing in a systematical manner.

IV. MapReduce Programming Paradigm

Since being first introduced by Google in 2004, MapReduce has become a popular programming paradigm. It focuses on providing users-friendly API for programming distributed algorithms whiel managing the parallelization details invisibly [8]. The spirit of a MapReduce program consists of "*keys*" and "*values*" which are generated and manipulated by user-defined functions "*mapper*" and "*reducer*" [8]. A key and a value are simply bytes of strings of arbitrary length and thus can represent generic data types. The MapReduce library automatically schedules parallel map and reduce operations linking mapper and reducer to handle the input data on a distributed system. State-of-the-art libraries for this purpose such as Apache Hadoop and MR-MPI from Sandia National Laboratory are available in the public domain [1], [2].

Algorithm 1: StandardForm(D, mapper, reducer)

Input: input data D, user-defined mapper and reducer

```
1 O ← new MapReduceObject;
2 {M | <tmp_key : tmp_value>} ← Map(O, D, mapper) ;
3 {C | <unique_key : value_list>} ← Collate(O);
4 {R | <key : value>} ← Reduce(O, reducer);
5 return R
```

A standard MapReduce program is presented in Algorithm 1. The first is the map step, which takes a set of data and converts it into another set of data produced by the function mapper, where individual elements are represented as temporary key/value pairs. The collate step aggregates across temporary key/value pairs where each unique key appears exactly once and the corresponding value is a concatenated list of all the values associated with the same key. The reduce step then takes a single entry from the aggregated key/value pairs and creates a new key/value pair which stores the output generated by the function reducer. Parallelism is evident since function calls by map and reduce are independent to each other and can be executed on different processors simultaneously.

V. Incremental PBA with MapReduce

We discuss in this section our incremental PBA algorithm with MapReduce. In a rough view, we decompose the incremental processing into a series of map and reduce operations. The map operation is responsible for 1) the track of affected tests for which timing information needs to be updated and 2) the generation of critical paths. The reduce operation is responsible for peeling out the true critical paths among tests.

A. Generation of Task Graph

We define the task graph g_t for a test t as the subgraph that spans from the launching FF (source FF) of the test to the capturing FF (testing FF) of the test. The task graph g_t is uniquely defined for each test t and can be viewed as a connectivity graph that covers all data paths feeding the testing FF of the test t. Figure 3 illustrates an example of three task graphs that are derived from testing FFs 5, 7, and 8, respectively. In our MapReduce program, we create a key/value pair for a task graph. The key indicates the test index to which this task graph belongs while the value field is a string that stores the user-defined data structure for graphs.

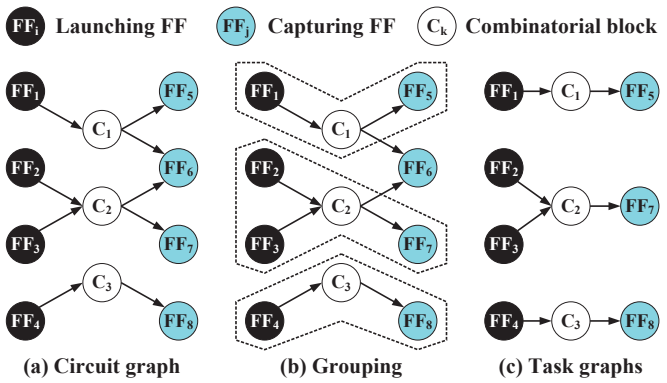

Figure 3. An example formulation of the task graphs.

The generation of the task graph is presented in Algorithm 2. For a given test, we first identify the testing FF. Starting at the data pin of the testing FF, the task graph can be induced via either a backward breadth-first search (BFS) or depth-first search (DFS) ending at any source FFs of the testing FF (line 2:3). Each induced task graph is emitted as a key/value pair in the end of the procedure (line 4).

Algorithm 2: Generator(t)

Input: a test t
Output: a key/value pair for the task graph g_t

1 $F_t \leftarrow$ testing FF of the test t;
2 $d \leftarrow$ data pin of the testing FF F_t;
3 $g_t \leftarrow$ subgraph induced from d backward to source FFs of F_t;
4 Emit make_pair(t, g_t);

B. Path Extraction and Peeling

Considering a task graph g_t, we can apply any path ranking algorithms to peel out a set of critical paths. The optimal and exact path ranking algorithm proposed by the first-place winner in TAU 2014 CAD contest, UI-Timer, is employed as our path extraction engine [10], [11]. The function of path extraction is presented in Algorithm 3. Algorithm 3 takes one explicit input argument of path count and one implicit input argument of a test index that is passed by the MapReduce caller. We then apply the path extraction algorithm by UI-Timer to the task graph and generate the set of top k critical paths (line 1:2). Each of these paths is emitted as a key/value pair to the MapReduce caller (line 3:5).

It can be inferred a MapReduce call of Algorithm 3 will produce a set of key/value pairs. The value field stores the path trace and the key indicates the test index to which this path belongs. After the collate method, those paths having the same key will be grouped together. Reporting the set of top k critical paths is identical to a reduce operation that peels out the top k critical ones from the path group. This can be simply achieved by a simple sorting process as presented in Algorithm 4.

Algorithm 3: Extracter(k)

Input: a parameter k, a test index t (implicit)
Output: key/value pairs for top k critical paths for test t

1 $g_t \leftarrow$ task graph parsed from Generator(t);
2 $P \leftarrow$ top k critical paths extracted from g_t using Algorithm by [10];
3 **foreach** *path* $p_i \in P$ **do**
4 | Emit make_pair(t, make_string(p_i));
5 **end**

Algorithm 4: Peeler(r)

Input: an unique key/value (value list) pair r
Output: an emitted key/value pair

1 *key* $\leftarrow$ *r.key*;
2 $P \leftarrow$ paths parsed from *r.value*;
3 sort(P);
4 $P' \leftarrow \{k$ frontmost elements in $P\}$;
5 *value* $\leftarrow$ make_string(P');
6 Emit make_pair(*key*, *value*);

C. Incremental Graph Operations

Optimization transforms are reflected by a set of graph-level operations that alter the graph incrementally. Whenever the graph experiences an incremental change, the circuit timing must be kept up-to-date for interactive queries. The idea of our algorithm is to identify a subset of tests for which the timing is affected, and quickly revalidate the slacks and critical paths for each of the affected tests. As any incremental operation is applied to either a node or an edge, identifying affected tests is equivalent to finding those tests whose task graphs include such changes. As a result, we consider the subroutine in Algorithm 5 that determines if a test is affected by a given node or edge. Using the subroutine in Algorithm 5, finding the set of affected tests can be carried out in parallel since the existence of each task graph is independent to each other. As presented in Algorithm 6, we create a new MapReduce object and conduct map operation linking Algorithm 5 as the mapper to find the test set that is affected by a given incremental change (line 1:3).

Algorithm 5: Affecter(v, e)

Input: a node v, an edge e
Output: a key/value pair for the test t if g_t includes v or e

1 $t \leftarrow$ test idnex passed by MapReduce caller;
2 $g_t \leftarrow$ task graph of the test t;
3 **if** $v \in g_t$ **or** $e \in g_t$ **then**
4 | Emit make_pair(t, **null**);
5 **end**

On the basis of Algorithm 6, we develop the solution to incremental graph operations. It is observed that a single operation *insert_node* produces no impact on the entire circuit graph unless it is connected to any portion of the graph using the operation *insert_edge*. Hence, the operation *insert_node* can be absorbed into the operation *insert_edge*. The discovery of affected tests in response to each of the four graph altering operations, *update_edge*, *remove_edge*, *remove_node*, and *insert_edge*, is presented in Algorithm 7, Algorithm 8, Algorithm 9,

Algorithm 6: GetAffectedTests(v, e)

Input: a node v, an edge e
Output: a set of affected tests T'

```
1 O' ← new MapReduceObject;
2 M' ← Map(O', T, Affecter(v, e));
3 T' ← test set parsed from M';
4 return T';
```

Algorithm 10, respectively. Notice that in order to obtain the affected tests correctly, these incremental graph operations will be applied to the graph 1) after the function calls of Algorithms 7–9 and 2) before the function call of Algorithm 10.

Algorithm 7: update_edge(e, w)

Input: an edge e, new edge delay w
Output: a set of affected tests T'

```
1 T' ← GetAffectedTests(null, e);
2 update the delay of the edge e to w;
3 return T';
```

Algorithm 8: remove_edge(e)

Input: an edge e
Output: a set of affected tests T'

```
1 T' ← GetAffectedTests(null, e);
2 remove the edge e from the circuit graph;
3 return T';
```

Algorithm 9: remove_node(v)

Input: a node v
Output: a set of affected tests T'

```
1 T' ← GetAffectedTests(v, null);
2 remove the node v from the circuit graph;
3 return T';
```

D. Incremental PBA Algorithm

Using Algorithms 1–10 as infrastructure, the overall algorithm of incremental PBA with MapReduce is presented in Algorithm 11. In a rough view, Algorithm 11 is executed in an interactive environment, incrementally accepting one of the seven graph operations that were defined in Section III. Except for the operation *node_insertion* which is directly applied to the graph (line 13:14), the other graph altering operations are handled by Algorithms 7–10 so as to identify the set of affected tests (line 5:12). The non-trivial part of Algorithm 11 is the response to path report (line 15:46). For the sake of coding ease, we manipulate two MapReduce objects O and O'. The MapReduce object O stores the key/value pairs of paths that are generated by the mapper call, while the MapReduce object O' is mainly operated by the reducer call for path peeling. The first step is to remove from O all paths that were previously generated from a given set of affected test over a series of incremental operations (line 16:17 and line 31:32). Because multiple operations might query about different path counts, any test with less number of paths in O than the present query of path count is marked as affected test as well (line 19:21 and line 34:37). The second step is to update the timing of affected tests by operating parallel mappers on Algorithm 3 (line 23 and line 40). The final step is to apply collate and reduce operations to the key/value pairs and

Algorithm 10: insert_edge(e)

Input: an edge e
Output: a set of affected tests T'

```
1 insert the edge e into the circuit graph;
2 T' ← GetAffectedTests(null, e);
3 return T';
```

Algorithm 11: IncrementalPBA(G)

Input: a circuit graph $G = \{V, E, T\}$

```
1  O ← new MapReduceObject;
2  O' ← new MapReduceObject;
3  T' ← T;
4  while op ← read_operation() do
5      if op = update_edge(e, w) then
6          T' ← T' ∪ update_edge(e, w);
7      else if op = remove_edge(e) then
8          T' ← T' ∪ remove_edge(e);
9      else if op = remove_node(v) then
10         T' ← T' ∪ remove_node(v);
11     else if op = insert_edge(e) then
12         T' ← T' ∪ insert_edge(e);
13     else if op = insert_node(v) then
14         insert the node v into the graph G;
15     else if op = report_path(t, k) then
16         if t ∈ T' then
17             remove_kv_objects(O, t);
18         else
19             if num_kv_objects(O, t) < k then
20                 remove_kv_objects(O, t);
21             end
22         end
23         Map(O, {t}, Extracter(k));
24         copy_kv_objects(O', O, t);
25         Collate(O');
26         Reduce(O', Peeler(k));
27         parse_and_report_paths_from_kv_objects(O', t);
28         T' ← T' − {t};
29     else if op = report_path(k) then
30         foreach key ∈ O do
31             if key ∈ T' then
32                 remove_kv_objects(O, key);
33             else
34                 if num_kv_objects(O, key) < k then
35                     remove_kv_objects(O, key);
36                     T' ← T' ∪ key;
37                 end
38             end
39         end
40         Map(O, T', Extracter(k));
41         O' ← O;
42         replace_all_keys(O', −1);
43         C' ← Collate(O');
44         Reduce(O', Peeler(k));
45         parse_and_report_paths_from_kv_objects(O', −1);
46         T' ← φ;
47     else
48         break;
49     end
50 end
```

TABLE I
PERFORMANCE OF THE PROPOSED MAPREDUCE-BASED PBA ALGORITHM ON BENCHMARKS FROM TAU 2014 CAD CONTEST [3].

Circuit	$\|V\|$	$\|E\|$	$\|C\|$	$\|I\|$	$\|O\|$	# Tests	# Paths	Baseline		MapReduce-Base PBA				
								mem (GB)	cpu (hr)	mem (GB)	cpu (hr)			
											Map	Collate	Reduce	Total
combo2	260636	284091	171529	170	218	29574	62938	0.4	18.1	4.1	0.56	0.43	0.01	0.7
combo3	181831	215733	73784	353	215	8294	129854	0.3	21.1	7.3	0.69	0.20	0.02	0.9
combo4	778638	866099	469516	260	169	53520	19227963	1.3	23.1	8.7	0.89	0.18	0.02	1.1
combo5	2051804	2228611	1456195	432	164	79050	19227963	-	> 24	29.3	1.46	0.28	0.02	1.8
combo6	3577926	3843033	2659426	486	174	128266	19227963	-	> 24	42.6	4.11	0.61	0.05	4.7
combo7	2817561	3011233	2136913	459	148	109568	19227963	-	> 24	40.9	2.88	0.51	0.03	3.4

$|V|$: size of node set. $|E|$: size of edge set. $|C|$: size of clock tree. $|I|$: # of primary inputs. $|O|$: # of primary outputs. # Tests: # of setup tests and hold tests. # Paths: max # of data paths per test. mem: max peak of memory usage (GB). cpu: program runtime (hours). -: unknown result due to runtime overhead.

derive the desirable path set (line 24:28 and line 41:46). We can see the benefit of maintaining two MapReduce objects because reporting the top k critical paths among all tests requires a reduce operation on all key/value pairs. In this case, the key field of all key/value pairs in O' is replaced with a nominal value (line 42). The solution to the path query can be retrieved and parsed from the key/value pair in the MapReduce object O' (line 27 and line 45).

VI. EXPERIMENTAL RESULTS

Our program is implemented in C++ language on a 64-bit linux operating system. The C++ based MR-MPI API is used as our MapReduce library [2]. Evaluation is taken on an academic cluster which has over 500 compute nodes. Each compute node is configured with 16 Intel 2.60GHz cores and 128GB RAM. The network infrastructure uses 384-port Mellanox MSX6518-NR FDR InfiniBand for high speed interconnect between clusters. Due to the limited resource per user, our communication world comprises 10 nodes with 10 cores residing on each node. Evaluations are undertaken on the six largest benchmarks, combo2–combo7 from 2014 TAU CAD contest [9]. In order to build up the incremental environment, we simulate the optimization transforms (i.e., repower gate, replace gate, etc.) from 2015 TAU CAD contest and randomly and uniformly generate one million incremental graph operations for each benchmark [4]. Table I lists the benchmark statistics and the overall performance of our MapReduce-based PBA algorithm.

For comparison purpose, we consider the UI-Timer as the baseline. UI-Timer is the first-place PBA timer of TAU 2014 CAD contest and its source code has been released to the public domain [3], [10]. The core part of UI-Timer is the algorithm of path extraction and its multi-threaded strategy for speeding up the process of multiple tests. In order to enable incremental processing, we adopt the same method as our MapReduce-based program. Whenever any graph altering operations are applied, the set of affected tests is identified and will be used for the subsequent path query. The baseline program is run on a single computing node with a total of 16 cores that supports multi-threading. In other words, we are interested in discovering the performance difference of incremental PBA algorithms between multi-threading and distributed MapReduce.

The performance of our MapReduce-based incremental PBA algorithm is quantified in Table I. In terms of runtime value, distributed MapReduce outperforms the baseline by more than an order of magnitude across all benchmarks. In the first three benchmarks, combo2, combo3, and combo4, the strength of distributed MapReduce is clearly shown by $\times 25$ faster in combo2, $\times 23$ faster in combo3, and $\times 21$ faster in combo4 than that of baseline. This runtime gap becomes even pronounced in the three largest benchmarks, combo5, combo6, and combo7, from which we can see the baseline method cannot accomplish the incremental processing within 24 hours [1] while the proposed method is able to reach the goal very fast by at most 4.7 hours in combo6. On the other hand, the higher memory usage of our program is expected because distributed MapReduce requires a unique and independent memory block for each process core. Extra storages of key/value pairs and MapReduce objects are taken into account as well. However, the amount of memory invoked by our program is fairly reasonable. From the average point of view, each core consumes less than 1 GB memory.

TABLE II
RUNTIME PERCENTAGE OF OUR PROGRAM ON DIFFERENT INCREMENTAL OPERATIONS.

Operation	cpu percentage (%)				
	update edge	remove edge	remove node	insert edge	report path
combo2	12.13	11.34	12.07	12.13	52.33
combo3	11.01	11.14	11.29	11.21	55.35
combo4	12.31	12.71	12.81	12.33	49.84
combo5	12.09	12.48	11.99	12.37	51.07
combo6	11.49	11.53	11.38	11.10	54.50
combo7	11.71	12.04	12.76	12.69	50.80

"report path" contains the two report_path operations.

Next we disclose the individual runtime of each incremental operation in Table II. The runtime of the operation *insert_node* is not listed since in our algorithm it is directly applied to the graph and the corresponding runtime value is ignorable. It can be seen that dealing with path report requires higher runtime than graph altering operations (i.e., *update_edge*, *remove_edge*, *remove_node*, and *insert_edge*). In most cases, the runtime percentage taken by path report is more than 50% (except for combo4). In comparison to path report, graph altering operations require simpler procedure, that is, identifying the set of affected tests via the graph traversal. It is also observed that the runtime percentage on the four graph altering operations is uniformly distributed around 11%–12%. This observation is predictable because each of the four graph altering operations executes the same mapper call.

The performance details of our program under different numbers of computing nodes (each configured with 10 cores) for the three largest benchmarks, combo5, combo6, and combo7, are plotted in Figure 4. We discover here different portions of runtime and memory that are taken by map, collate, and reduce operations. In general, the

[1] Due to the user policy in our cluster, we are not allowed to run a program in by more than 24 hours.

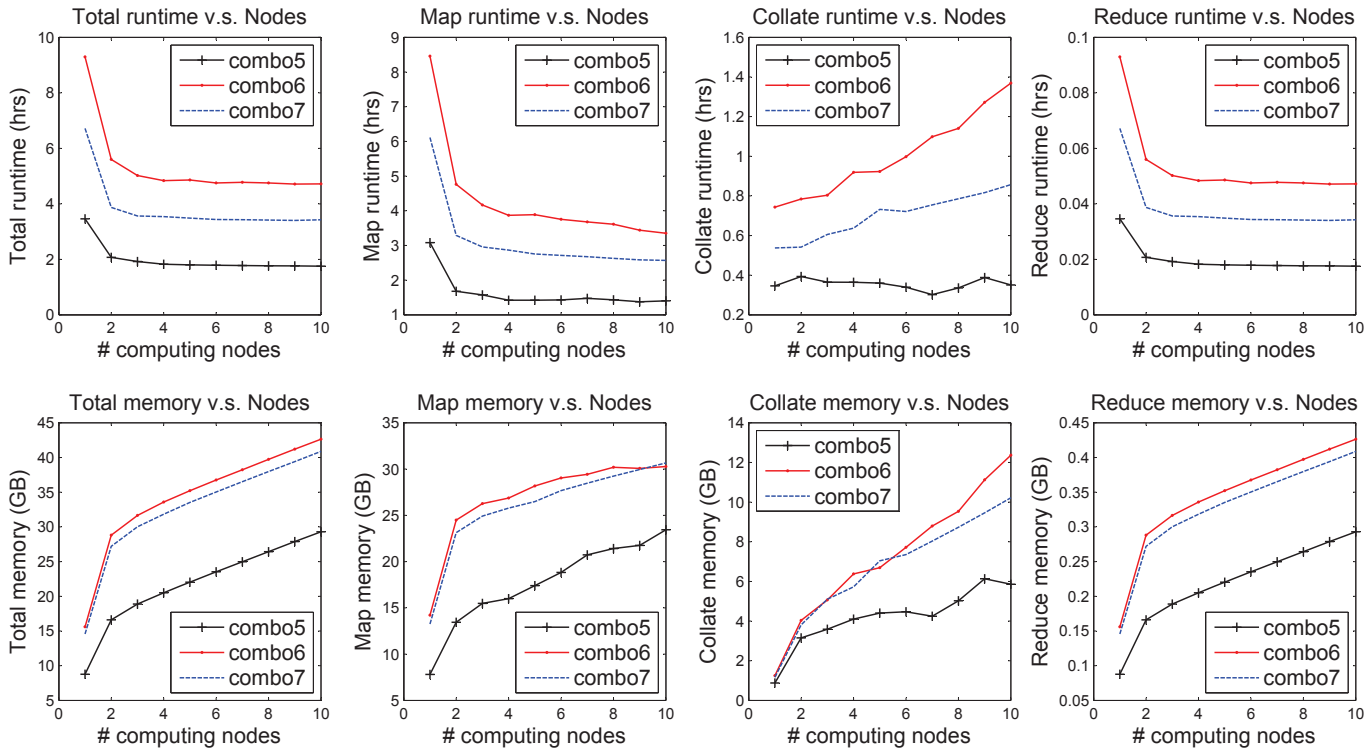

Figure 4. Performance plot of the proposed MapReduce-Based incremental PBA under different number of computing nodes. Each computing node is configured with 10 process cores. Plots in the first row demonstrate (from left to right) the runtime of the program and the runtime portions of map operations, collate operations, and reduce operations, respectively. Plots in the second row demonstrate the respective memory usage.

total runtime decreases as the number of computing nodes increases. However, the rate of runtime reduction does not scale linearly but tends to be gradually saturated after 5 nodes. The reason comes from the cost of process communication. We can see the runtime portion spent on the collate operation begins growing as the number of nodes increases. At a more detailed view, it is observed that the map operation takes the majority of runtime while the runtime spent on the reduce operation is almost negligible. This is because the map operation is responsible for the generation of task graphs and extraction of critical paths, which is more time-consuming than the sorting process in the reduce operation. Exact runtime values can be referred to Table I. As aforementioned, the more the number of computing nodes we use, the larger the amount of memory our program invokes. This property is reflected on all map, collate, and reduce operations.

VII. Conclusion

In this paper we have presented a fast incremental PBA algorithm with MapReduce. To the best knowledge of the authors, this work is the first attempt to deal with the incremental PBA problem using distributed MapReduce. We have successfully formulated the incremental PBA problem into key/value tasks that are solvable by the popular MapReduce programming paradigm from big-data community. The experimental results have demonstrated the promising performance of our approach whereby million-scale circuit graphs subject to a critical amount of optimization transforms can be quickly and accurately analyzed on a computer cluster. Our work can be beneficial in assisting designers in speeding up the lengthy design cycles of timing optimizations and timing closure.

References

[1] Apache Hadoop: http://hadoop.apache.org/

[2] MapReduce MPI Library: http://mapreduce.sandia.gov/

[3] TAU 2014 CAD Contest on Common Path Pessimism Removal: https://sites.google.com/site/taucontest2014/

[4] TAU 2015 CAD Contest on Incremental Timing Analysis: https://sites.google.com/site/taucontest2015/

[5] S. Bhardwaj, K. Rahmat, and K. Kucukcakar, "Clock-Reconvergence Pessimism Removal in Hierarchical Static Timing Analysis," US patent 8434040, 2013.

[6] J. Bhasker and R. Chadha, "Static Timing Analysis for Nanometer Designs: A Practical Approach," *Springer*, 2009.

[7] S. Cristian, N. H. Rachid, and R. Khalid, "Efficient exhaustive path-based static timing analysis using a fast estimation technique," US patent 8079004, 2009

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *CACM*, vol. 51, no. 1, 107–113, 2008

[9] J. Hu, D. Sinha, and I. Keller, "TAU 2014 Contest on Removing Common Path Pessimism during Timing Analysis," *Proc. ACM ISPD*, pp. 153–160, 2014.

[10] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm," *Proc. IEEE/ACM ICCAD*, pp. 758–765, 2014.

[11] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Fast Path-Based Timing Analysis for CPPR," *Proc. IEEE/ACM ICCAD*, pp. 596–599, 2014.

[12] T.-W. Huang and M. D. F. Wong, "Accelerated Path-Based Timing Analysis with MapReduce," *Proc. ACM ISPD*, pp. 103–110, 2015.

[13] O. Levitsky, "Sign Off Quality Hierarchical Timing Constraints: Wishful Thinking or Reality?" *TAU workshop*, 2014.

[14] R. Molina, "EDA Vendors should Improve the Runtime Performance of Path-Based Timing Analysis," *Electronic Design*, 2013