Procedia Computer Science

Procedia Computer Science 00 (2012) 1–10

International Conference on Computational Science, ICCS 2012

Parallel Tree Reduction on MapReduce

Kento Emoto[a], Hiroto Imachi[a]

[a]*Graduate School of Information Science and Technology, University of Tokyo*

Abstract

MapReduce, the de facto standard for large scale data-intensive applications, is a remarkable parallel programming model, allowing for easy parallelization of data intensive computations over many machines in a cloud. As huge tree data such as XML has achieved the status of the de facto standard for representing structured information, the situation calls for efficient MapReduce programs treating such a tree data structure in parallel. However, development of such MapReduce programs has remained a challenge. In this paper, restructuring our previous BSP algorithm for tree reduction computations, we propose a new MapReduce algorithm that can be used to implement various tree computations such as XPath queries. Our algorithm is designed to achieve linear speedup even for extreme inputs, and our experimental result shows that our prototype implementation actually achieves linear speedup even for monadic trees.

Keywords: Hadoop, MapReduce, Tree Homomorphism

1. Introduction

MapReduce [1], the de facto standard for large scale data-intensive applications, is a remarkable parallel programming model, allowing for easy parallelization of data intensive computations over many machines in a cloud. It is used routinely at companies such as Yahoo!, Google, Amazon, and Facebook. Its abstract interface effectively hides the details of parallelization, data distribution, load balancing, and fault tolerance.

XML has achieved the status of the de facto standard for representing structured information. Depending on the information an XML tree represents, the shape of the tree may be imbalanced and its size can be quite huge. This situation calls for an efficient MapReduce program treating such a tree data structure in parallel. However, development of such MapReduce programs has remained a challenge. The main difficulties here are that deserializing a huge tree (DOM tree) from the serialized format (XML file) costs very much, and that simple divide-and-conquer parallelism, such as naive use of MapReduce, suffers from the factor of the height of input trees.

In the previous work [2, 3], we proposed a parallel algorithm on the BSP model [4, 5] for tree reduction computations so-called *tree homomorphisms*, which covers XML computation such as XPath queries and dynamic programming [6, 7, 8]. The algorithm runs tree reductions without deserializing input trees, and achieves linear speedup independent of the shapes of input trees. Therefore, we can tackle the challenge by importing the BSP algorithm into MapReduce.

Email addresses: emoto@mist.i.u-tokyo.ac.jp (Kento Emoto), Hiroto_Imachi@mist.i.u-tokyo.ac.jp (Hiroto Imachi)

In this paper, restructuring the BSP algorithm, we propose a MapReduce algorithm for tree reductions, and report experimental results of our prototype implementation on Hadoop [9, 10], a popular implementation of MapReduce. The main contributions of this paper are listed as follows.

- We propose a MapReduce algorithm that achieves linear speedup even for extreme inputs, namely, monadic trees. This is the first MapReduce algorithm that has such nice feature.
- We also give a simplified version of the algorithm, and evaluates them by our prototype implementation on Hadoop. This is the first comparison of such two algorithms, and the evaluation suggests a hybrid algorithm.

The rest of this paper is organized as follows. Section 2 presents definitions of data structures and computation on them. Section 3 introduces the BSP algorithm for parallel tree reduction. Section 4 imports the algorithms into MapReduce, and Section 5 shows experimental results. Section 6 discusses related work, and Section 7 concludes this paper.

2. Preliminaries

The notation in this paper is reminiscent of Haskell [11]. Function application is denoted by a space and the argument may be written without brackets, so that $f\ a$ means $f(a)$ in ordinary notation. Functions are curried: they always take one argument and return a function or a value, and the function application associates to the left and binds more strongly than any other operator, so that $f\ a\ b$ means $(f\ a)\ b$ and $f\ a \otimes b$ means $(f\ a) \otimes b$. Function composition is denoted by $\circ$, and $(f \circ g)\ x = f\ (g\ x)$ according to its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$.

Lists are finite sequences of values of the same type. A list is either a singleton, or the concatenation of two other lists. We write $[a]$ for the singleton list with element a, and $xs \mathbin{+\!\!+} ys$ for the concatenation of two lists xs and ys. For example, the term $[1] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [3]$ denotes a list with three elements, often abbreviated as $[1, 2, 3]$.

2.1. Trees and Their Serialized Representation

We treat trees with unbound degree (trees whose nodes can have an arbitrary number of subtrees); Figure 1 shows an example. A definition of data structure *RTree* for trees with unbound degree is given as follows.

$$\mathbf{data}\ RTree\ \alpha = Node\ \alpha\ [RTree\ \alpha] \mid Leaf\ \alpha$$

Our internal representation is to keep tree-structured data in a serialized manner. The sequence of the middle in Figure 1 is our internal representation of the example tree. It is a simple abstraction of XML serialization; a combination of a preorder (for producing the open elements) and a postorder traversal (for producing the close elements afterwards). We assume *well-formedness*, with which sequences are guaranteed to be parsed back into trees, and without loss of information we simplify close elements to be "/". The figure also depicts their presentation according to the depth.

2.2. Tree Homomorphism and Extended Distributivity

We use the framework called *tree homomorphism* [6, 12], which specifies recursive tree reductions h using h', $\oplus$, and associative $\otimes$ with its unit $\iota_\otimes$:

$$\begin{aligned} h(\ Node\ a\ [t_1, \ldots, t_n]\ ) &= a \oplus (\ h(t_1) \otimes \cdots \otimes h(t_n)\ )\ , \\ h(\ Leaf\ a\ ) &= h'(a)\ . \end{aligned}$$

For example, consider a computation *maxPath* to find the maximum of the values each of which is a sum of values in the nodes from the root to each leaf. This "maximum path sum" computation is a tree homomorphism:

$$\begin{aligned} maxPath(\ Node\ a\ [t_1, \ldots, t_n]\ ) &= a + (\ maxPath(t_1) \uparrow \cdots \uparrow maxPath(t_n)\ )\ , \\ maxPath(\ Leaf\ a\ ) &= id(a) = a\ . \end{aligned}$$

Here, *id* is the identity function, and $\uparrow$ returns the bigger of two numbers whose unit is $-\infty$. When it is applied to the example tree in Figure 1, the result should be $17 = 3 + 7 + (-5) + 4 + 5 + 3$.

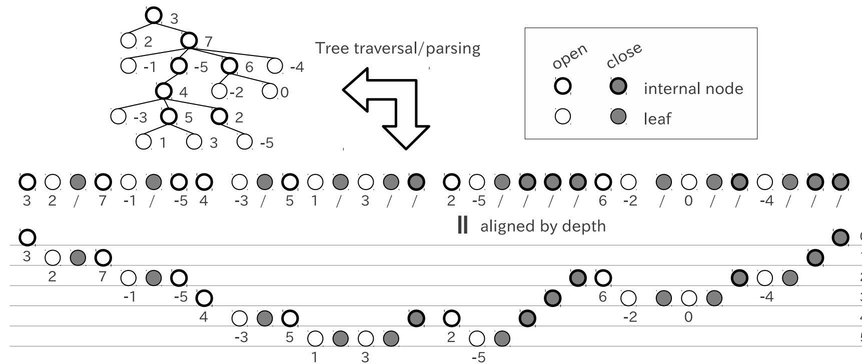


Figure 1: A rose tree and its serialized representation.

For brevity, we use this simple computation "maximum path sum" as a running example throughout this paper. It should be noted that our proposed algorithms in this paper are not limited to this example computation.

We can apply parallelization of tree homomorphism over serialized trees based on list homomorphism [13]. This naive approach, however, may suffer from the factor of tree depths. We need to review an additional property called *extended distributivity* [6, 14]. This property is explained, by introducing a new operator $\ominus$ defined as $(a, b, c) \ominus e = a \oplus (b \otimes e \otimes c)$, as follows: for any *triple*s (a_u, b_u, c_u), (a_l, b_l, c_l), and any expression e, there exists a *triple* (a, b, c) which satisfies

$$(a_u, b_u, c_u) \ominus (\ (a_l, b_l, c_l) \ominus e\ ) = (a, b, c) \ominus e\ .$$

Efficient parallel reduction requires these computations as well as $\otimes$ and $\oplus$ to be done in constant time. Our running example satisfies these properties as the following calculation shows.

$$\begin{aligned}
&(a_u, b_u, c_u) \ominus (\ (a_l, b_l, c_l) \ominus e\ ) \\
&= a_u + (b_u \uparrow (a_l + (b_l \uparrow e \uparrow c_l)) \uparrow c_u) \\
&= (a_u + a_l) + ((-a_l + b_u \uparrow b_l) \uparrow e \uparrow (c_l \uparrow -a_l + c_u)) \\
&= (\ (a_u + a_l),\ (-a_l + b_u \uparrow b_l),\ (c_l \uparrow -a_l + c_u)\ ) \ominus e
\end{aligned}$$

For some other examples under these formalizations, see previous work [6, 14].

3. BSP Algorithm for Parallel Tree Reduction

In this section, we briefly review our previous algorithm [2] on the BSP model [4, 5] for parallel computation of tree homomorphism. In general, a BSP algorithm consists of a sequence of super steps, and a super step consists of a local computation followed by a global communication synchronized by a global barrier.

Figure 2 shows an illustration of our BSP algorithm demonstrated for the maximum path sum computation for the example tree in Figure 1. The super steps are explained in the following subsections. Please refer to our previous work [2] for details.

3.1. First Super Step: Evaluating Local Subtrees to Make Hills and Sharing Their Shape Information

This step applies the given tree homomorphism in parallel to every subtrees within each subsequence of the serialized input, so that each subsequence is reduced into a possibly small *hill*.

In the local computation phase, each processor applies the given tree homomorphism to forests in its assigned subsequence of the serialized input. This process leaves fragments of results forming a hill, because every valley represents a subtree and thus is reduced into a value. For example, in Figure 2, the process 2 receives the subsequence of eight elements starting with −3, and the result is singleton value 8 indicated by a round rectangle, because the subsequence represents a forest of two complete subtrees. Similarly, the processor 3 takes the succeeding subsequence starting with 2, and builds a hill of height three, in which the left most value −3 is the result of the homomorphism

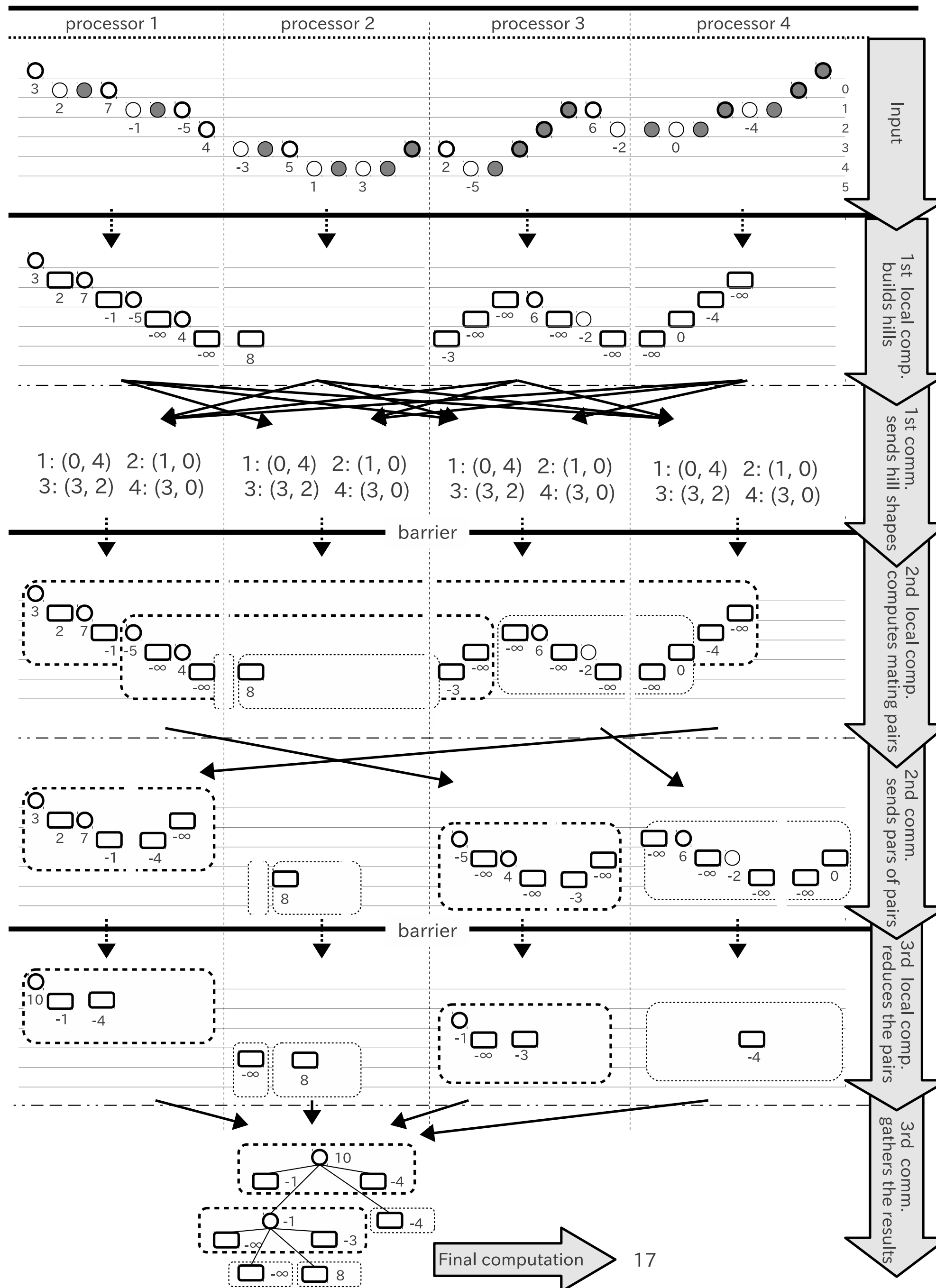


Figure 2: Flow of the BSP algorithm demonstrated for the maximum path sum computation.

applied to the complete subtree represented by the first four elements 2, -5, / and /. Since the other elements in the subsequence do not form complete trees, their values remain in the resulting hill with a few additional identities $-\infty$.

In the global communication phase, each processor sends the shape of its hill (the pair of lengths of left and right slopes of the hill) to all other processors, so that every processor can schedule remaining computation. For example, the processor 3 sends the pair $(3, 2)$ to other processors, because the left slope of its hill has three rounded rectangles and the right slope has two pairs of circles and rounded rectangles.

3.2. Second Super Step: Building Mating Pairs

This step matches data fragments kept in each processor into *triple*s using communication between processors, to execute the rest of the computation efficiently in parallel.

In the local computation phase, every processor computes *mating pairs* that represent communications necessary to build *triple*s. For example, in Figure 2, the first two values 3 and 2 of the processor 1 and the last value $-\infty$ of processor 4 are to form *triple* $(3, 2, -\infty)$ corresponding to the context $\lambda x.3 + (2 \uparrow x \uparrow -\infty)$. Similarly, values 7 and -1 of the processor 1 and value -4 of the processor 4 are to form the consecutive triple $(7, -1, -4)$. These *triple*s are enclosed by dotted round rectangles, each of which represents a mating pair. Other dotted round rectangles indicate other mating pairs.

In the communication phase, each processor sends parts of its hill to other processors according to the computed mating pairs. For example, the values $-\infty$ and -4 of the last processor are sent to the processor 1 according to the enclosing mating pair. Similarly, the latter half of values of the processor are sent to the processor 3 according to the second large mating pair, and values in the other mating pair of the processor 3 are sent to the processor 4. Note that some mating pairs include an additional left most value, such as the left most $-\infty$ of the mating pair in the processor 4. Please refer to our previous work [2] for details of the scheduling of the communication.

3.3. Third Super Step: Evaluating Mating Pairs and Gathering the Results

This step reduces triples in each mating pair into a triple or a value, and gathers the result into a binary tree to compute the final result.

For example, in Figure 2, on the processor 4 the mating pair consisting of two *triple*s with the left most additional $-\infty$ is reduced into one value -4, because they correspond to a forest in the input tree. *Triple*s in the mating pair on the processor 3 is merged into another *triple* that represents the context $\lambda x. -1 + (-\infty \uparrow x \uparrow -3) = (\lambda x. -5 + (-\infty \uparrow x \uparrow -\infty)) \circ (\lambda x.4 + (-\infty \uparrow x \uparrow 3))$ that is the composition of contexts represented by the *triple*s $(-5, -\infty, -\infty)$ and $4, -\infty, 3$.

In the communication phase, the resulting values and triples are sent to one processor to build a binary tree, as shown in the bottom of Figure 2.

3.4. Final Computation: Evaluating the Resulting Small Binary Tree

The final step of the algorithm computes the final result of the homomorphism by evaluating the small binary tree. For example, the final result in Figure 2 is 17, which is the maximum path sum of the example tree.

4. MapReduce Algorithm for Parallel Tree Reductions

We import the BSP algorithm described in Section 3 to MapReduce, so that we can enjoy MapReduce's nice features such as fault tolerance, balancing, etc. in parallel computation treating huge tree data. A MapReduce algorithm usually consists of a sequence of Map-Reduce rounds, and each Map-Reduce round consists of a Map phase and a Reduce phase, in which user-defined Map and Reduce functions are applied to the input and the intermediate data [1, 15].

Figure 3 shows our MapReduce algorithm demonstrated for the example tree shown in Figure 1, which basically restructures the BSP algorithm demonstrated in Figure 2. Our MapReduce algorithm consists of two Map-Reduce rounds. The following subsections explain the Map and Reduce functions.

The parallel time complexity of our MapReduce algorithm is $O(n/p + c)$ where n is the number of nodes in the input tree, p is the number of processors (Map and Reduce processes) and c is the number of chunks (fragments of the serialized input tree). Since c is usually much less than n/p, the complexity must be $O(n/p)$. In the following explanations, we use n, p and c to mean the same values as above.

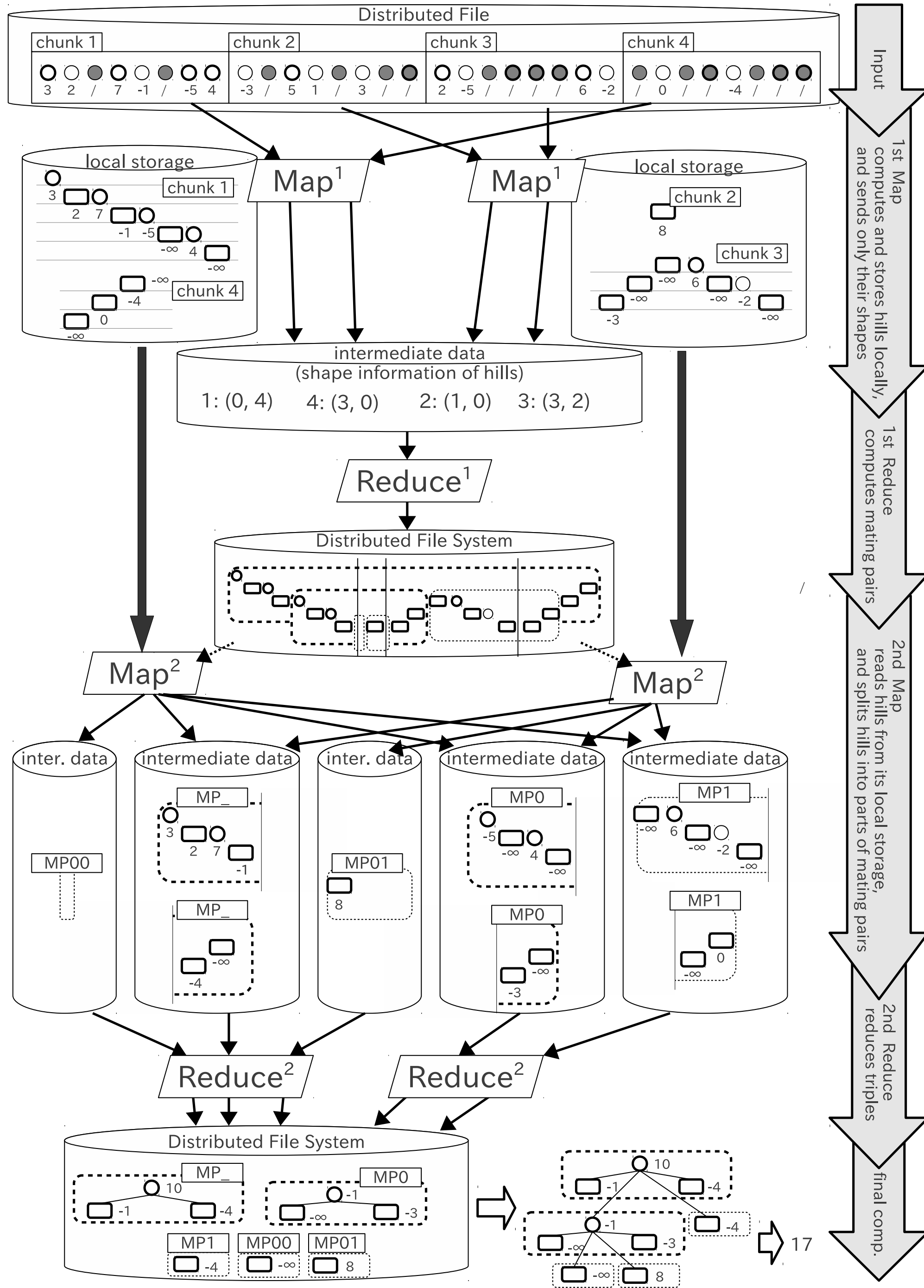


Figure 3: Flow of the MapReduce algorithm demonstrated for the maximum path sum computation.

4.1. The First Map

The first Map mainly corresponds to the first super step of the BSP algorithm: It computes a hill from a subsequence of the serialized input by applying the given tree homomorphism to forests, and sends shape information of hills to the first Reduce. The serialized input tree is stored as chunks on the distributed file system.

Receiving a chunk, the first Map builds a hill from the chunk. It then stores the hill into its local storage, and outputs the pair of the hill's shape information and the chunk's id.

One issue of importing BSP algorithm into MapReduce is a difference between two models: In the BSP model two local computations in different super steps on the same processor can share any local data without any overhead, but in MapReduce model two Maps in different Map-Reduce rounds cannot share any local data even if they are executed on the same processor. Basically, to share large local data among Maps separated by a Reduce, one Map needs to output the local data to the distributed file system and the other has to read it from the distributed file system. This may lead to non-negligible overhead due to network traffic. To reduce such overhead, we have selected an alternative approach: The first Map writes its computed hills into *its local storage*, and the hills are read by the second Map run on the same processor without any network traffic. It should be noted that this alternative approach has less fault tolerance than the approach storing local data into the distributed file system: If a processor dies in the second Map, we have to rerun the first Map to recover hills stored in its local storage, but such recovery over multiple Map-Reduce rounds is not supported by MapReduce. This is a trade-off of efficiency and fault tolerance.

There is another issue of importing BSP algorithm into MapReduce: we do not have a cost model on MapReduce, while we can enjoy the nice cost model of BSP, e.g., in optimizing BSP algorithms. This topic is beyond the scope of this paper, but it is an interesting direction of further research.

The parallel time complexity of this phase is $O(n/p + c)$, because p Maps process the entire input of length n in parallel, and the total size of their output (shape information) is proportional to c, the number of hills.

4.2. The First Reduce

The first Reduce corresponds to the local computation of the second super step: It receives shape information of all hills, computes all mating pairs from the information, and writes the whole information of mating pairs into the distributed file system. The information of the mating pairs is used by all of the second Maps.

The time complexity of this phase is $O(c)$, because the time complexity of the computation of all mating pairs is proportional to the number of hills, namely, c [2].

4.3. The Second Map

The second Map corresponds to the communication phase of the second super step: It reads hills from its local storage, and outputs parts of mating pairs as the intermediate data to the second Reduces. Two parts of a mating pair have the same key representing its position in the final binary tree. For example, the left most part of the hill of the chunk 1 has the key MP_{-} that represent the root. Similarly, its counterpart, namely, the right most part of the hill of chunk 4 has the same key. The rest of the hill of the chunk 1 and its counterpart have the key MP0 that represents the left child of the root. Since the singleton hill of the chunk 2 is the right child of MP0, it has key MP01.

The parallel time complexity of this phase is $O(n/p)$, because each mating pair has a unique key, and all mating pairs of total size $O(n)$ can be output in parallel by p Maps.

4.4. The Second Reduce

The second Reduce corresponds to the local computation of the third super step: Receiving the pair of parts of a mating pair, it reduces *triple*s in the mating pair into a value or a *triple*, and writes the result to the distributed file system.

The parallel time complexity of this phase is $O(n/p)$, because all mating pairs of total size $O(n)$ can be processed in parallel by p Reducers.

4.5. The Final Computation

The final computation builds the small binary tree from the results of the second Reduce, and evaluating the binary tree to get the final result of the homomorphism. This part should be done sequentially outside the Map-Reduce rounds. The time complexity is $O(c)$, because the size of the tree is proportional to the number of hills c [2].

Table 1: Input serialized trees for the evaluation.

name	# of nodes/leaves	depth	# of mating pairs
Shallow	3×10^8	10	3000
Deep	3×10^8	43536	5886
Monadic	3×10^8	3×10^8	5997

Table 2: Execution time (seconds) of MapReduce algorithm

	Shallow			Deep			Monadic		
P	M1&R1	M2&R2	Total	M1&R1	M2&R2	Total	M1&R1	M2&R2	Total
1	1294	649	1944	1322	661	1986	1298	9673	10973
2	722	260	990	779	334	1123	1026	5148	6186
4	391	158	557	414	172	595	580	2838	3426
8	215	99	322	224	103	337	296	1431	1736

Table 3: Execution time of Simplified algorithm

P	Shallow	Deep	Monadic
1	1294	1322	N.A.
2	696	769	N.A.
4	358	400	N.A.
8	192	234	N.A.

4.6. Simplified Algorithm

In typical cases, the size of hills built by the first Map is expected to be small, and we may simplify the algorithm by merging the hills sequentially on one processor in the first Reduce phase, instead of doing the second round of Map-Reduce to merge (reduce) the hills in parallel. This alternative algorithm corresponds to Algorithm 1 in the technical report [3]. When the hills and the number of them are small, the parallel time complexity keeps $O(n/p)$.

In the following sections, we call this alternative “simplified algorithm”, and the original algorithm with two Map-Reduce rounds “mating-pair algorithm”.

5. Experimental Results and Discussion

We have evaluated our two MapReduce algorithms on a small cluster of eight PCs (Intel Core 2 Duo and 2GB memory) connected by Giga-bit Ethernet. We used Hadoop 0.20.203.0 [9, 10], Linux 2.6.32, Java 1.6.0, and Scala 2.9.1. We have implemented the proposed algorithms as a library on Scala so that users can easily use our algorithms by simply implementing their homomorphisms in Scala’s concise notation. We prepared randomly generated trees of three types, namely (Shallow) shallow, (Deep) deep, and (Monadic) monadic tree. The depth of Shallow came from observations on XML documents [16]. Table 1 shows information of the input. The tree homomorphism executed by algorithms was of the maximum path sum computation, namely, *maxPath*.

Table 2 lists measured execution times of the mating-pair algorithm, in which the column M1&R1 lists execution times of the first Map-Reduce round, and M2&R2 the second round. For non-extreme inputs, namely, Shallow and Deep, the first round takes about two thirds of the total execution time, while for the extreme input Monadic the second round dominates the total execution time. The reason of the behavior is as follows. For Monadic, no valley is reduced in the first round, and every hill has the same length as the input chunk. Thus, almost all computation of the homomorphism is done in the Reduce of the second round. Figure 4 shows speedups of the algorithm, and it shows that our mating-pair algorithm scales even for the extreme input.

Table 3 lists measured execution times of the simplified algorithm. For Shallow and Deep, the simplified algorithm is faster than the mating-pair algorithm, because the size of the hills is small for these inputs and thus the sequential merging of the hills is cheaper than the computation using mating pairs. However, the simplified algorithm does not work for Monadic. This is because, for Monadic, its Reduce has to apply the homomorphism to the whole of the input, but the input is too large to be stored in the memory, and the process aborts.

These result suggest a hybrid algorithm that uses the simplified algorithm for non-extreme inputs and the mating-pair algorithm for extreme inputs.

Finally, Figure 6 shows experimental results on Amazon EC2 up to 96 instances, in which we used small instances and Hadoop 0.20.203.0 with the default configuration except for the chunk size of 4MB. Our proposed method scales well in the practical cloud environment.

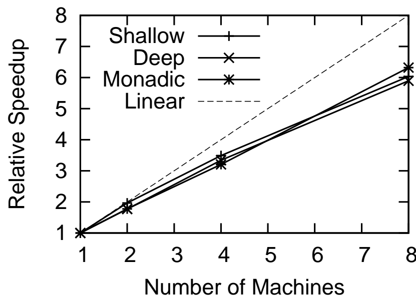


Figure 4: Relative speedup of MapReduce algorithm

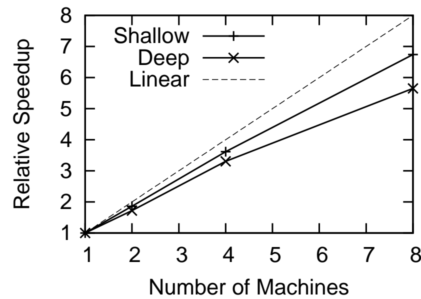
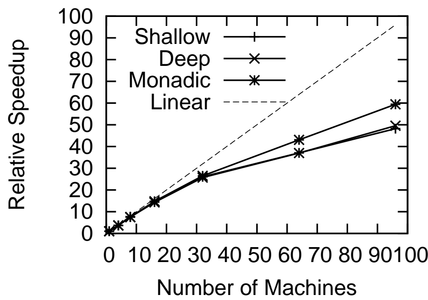


Figure 5: Relative speedup of Simplified algorithm



P	Shallow	Deep	Monadic
1	8729	8738	32299
8	1137	1165	4301
16	608	612	2188
32	334	340	1219
64	235	236	750
96	181	176	544

(seconds)

Figure 6: Experimental results of MapReduce algorithm on Amazon EC2

6. Related Work

Much effort has been devoted to developing tree homomorphisms for various problems, such as queries on trees [6, 8], dynamic programming [7], etc. [14, 17, 18]. Our proposed MapReduce algorithm enables computations of these problems to be run on MapReduce, in which we can enjoy nice features such as fault tolerance, load-balancing, etc.

Liu et al. [19] proposed a MapReduce algorithm of list homomorphism [13, 20]. Since many studies have been done for deriving list homomorphisms for large class of problems [21, 22, 23, 24, 25, 26], their work enables many computations treating lists to be run on MapReduce enjoying its nice features. Our work in this paper plays a similar role for tree problems.

Many studies have been devoted to graph computations on MapReduce, e.g., [27, 28, 29]. The common idea is to iterate MapReduce rounds (jobs) in which the Map is applied to each vertex to produce partial results from its internal state, the partial results are grouped and passed to its neighbor vertices (in the Shuffle phase), and for each vertex the Reducer computes vertex's new internal state based on the incoming partial results. Since a tree is a special case of graphs, to carry out some tree computaions we may use their proposed methods, but such an approach has drawbacks in expressiveness and efficiency: no recursive computation is allowed and the number of Map-Reduce rounds is at least $O(\log(n))$ for the input tree with n nodes. Our proposed method can carry out a recursive computation on a tree in parallel by using one or two MapReduce rounds.

There are few studies involving tree computations on MapReduce. Khatchadourian et al. proposed ChuQL [30] that extends XQuery with MapReduce primitives, in which the output sequence of XQuery can be supplied to MapReduce and vice versa seamlessly, and we can use XQuery in Map and Reduce. However, a query itself is not parallelized in ChuQL by MapReduce. Our proposed method is capable of parallelizing an XPath query [6].

7. Conclusion

This paper describes a MapReduce algorithm for tree reduction that can implement various tree computations such as XPath queries on XMLs and etc [6, 7, 8]. The described algorithm is the first one that can achieve linear speedup even for the extreme input, e.g.,, monadic trees. This paper also gives a simplified algorithm that achieves faster execution time for typical input. The experimental results suggests a hybrid algorithm of them.

Our future direction includes implementing various applications upon the proposed algorithm, and development of user-friendly interfaces to the algorithm.

References

[1] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.
[2] K. Kakehi, K. Matsuzaki, K. Emoto, Efficient parallel tree reductions on distributed memory environments, in: Computational Science - ICCS 2007, Part II, Vol. 4488 of LNCS, Springer, 2007, pp. 601–608.
[3] K. Kakehi, K. Matsuzaki, K. Emoto, Z. Hu, A practicable framework for tree reduction under distributed memory environments, Tech. Rep. METR 2006–64, Department of Mathematical Informatics, University of Tokyo, available online at `http://www.keisu.t.u-tokyo.ac.jp/research/techrep/2006.html` (2006).
[4] L. Valiant, A bridging model for parallel computation., Communication of the ACM 33 (8) (1990) 103–111.
[5] W. McColl, Scalable computing., in: Computer Science Today: Recent Trends and Developments, Vol. 1000 of LNCS, Springer, 1995, pp. 46–61.
[6] K. Matsuzaki, Parallel programming with tree skeletons, Ph.D. thesis, Graduate School of Information Science and Technology, University of Tokyo (2007).
[7] K. Matsuzaki, Z. Hu, M. Takeichi, Towards automatic parallelization of tree reductions in dynamic programming, in: SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2006, pp. 39–48.
[8] D. Skillicorn, Structured parallel computation in structured documents, Journal of Universal Computer Science 3 (1) (1997) 42–68.
[9] The Apache Software Foundation, Welcome to Apache Hadoop!, http://hadoop.apache.org/ (2011).
[10] T. White, Hadoop: The Definitive Guide, O'Reilly Media, 2009.
[11] R. Bird, Introduction to Functional Programming using Haskell, Prentice Hall, 1998.
[12] D. B. Skillicorn, Parallel implementation of tree skeletons., Journal of Parallel and Distributed Computing 39 (2) (1996) 115–125.
[13] M. Cole, Parallel programming with list homomorphisms., Parallel Processing Letters 5 (1995) 191–203.
[14] K. Matsuzaki, Z. Hu, K. Kakehi, M. Takeichi, Systematic derivation of tree contraction algorithms, Parallel Processing Letters 15 (3) (2005) 321–336.
[15] R. Lämmel, Google's mapreduce programming model—revisited, Science of Computer Programming 70 (1) (2008) 1–30.
[16] L. Mignet, D. Barbosa, P. Veltri, The xml web: a fist study., in: Proceedings of the Twelfth International World Wide Web Conference, ACM Press, 2003, pp. 300–510.
[17] A. Morihata, K. Matsuzaki, A practical tree contraction algorithm for parallel skeletons on trees of unbounded degree, Procedia CS 4 (2011) 7–16.
[18] A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer, in: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09), ACM, 2009, pp. 177–185.
[19] Y. Liu, Z. Hu, K. Matsuzaki, Towards systematic parallel programming over mapreduce, in: Euro-Par 2011 Parallel Processing, Part II, Vol. 6853 of LNCS, Springer, 2011, pp. 39–50.
[20] Z. Grant-Duff, P. Harrison, Parallelism via homomorphism, Parallel Processing Letters 6 (2) (1996) 279–295.
[21] S. Gorlatch, Systematic efficient parallelization of scan and other list homomorphisms, in: Euro-Par '96 Parallel Processing, Vol. 1124 of LNCS, Springer, 1996, pp. 401–408.
[22] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, M. Takeichi, Automatic inversion generates divide-and-conquer parallel programs, in: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07), ACM Press, 2007, pp. 146–155.
[23] A. L. Fisher, A. M. Ghuloum, Parallelizing complex scans and reductions, in: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94), ACM, 1994, pp. 135–146.
[24] Z. Hu, M. Takeichi, W.-N. Chin, Parallelization in calculational forms, in: 25th ACM Symposium on Principles of Programming Languages (POPL'98), ACM Press, 1998, pp. 316–328.
[25] W.-N. Chin, S.-C. Khoo, Z. Hu, M. Takeichi, Deriving parallel codes via invariants, in: Static Analysis, 7th International Symposium, SAS 2000, Vol. 1824 of LNCS, Springer, 2000, pp. 75–94.
[26] S. Sato, H. Iwasaki, Automatic parallelization via matrix multiplication, in: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11), ACM, 2011, pp. 470–479.
[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, 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, ACM, 2008, pp. 1099–1110.
[28] J. Lin, M. Schatz, Design patterns for efficient graph algorithms in mapreduce, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs, ACM, 2010, pp. 78–85.
[29] U. Kang, H. Tong, J. Sun, C.-Y. Lin, C. Faloutsos, Gbase: a scalable and general graph management system, in: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2011, pp. 1091–1099.
[30] S. Khatchadourian, M. P. Consens, J. Siméon, Having a chuql at xml on the cloud, in: Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Vol. 749 of CEUR Workshop Proceedings, CEUR-WS.org, 2011.