数理解析研究所講究録
第1744巻 2011年 201-204

2010年度冬のLAシンポジウム [S8]

大規模分散フレームワークHadoopを用いた接尾辞配列構築

田中 洋輔 * 定兼 邦彦 † 山下 雅史 ‡

1 概要

索引を用いることで, 大規模な文字列に対する高速な検策機能を実現できる. 索引には, 転置インデックスや接尾辞配列 (SA) [4] などがある. 転置インデックは比較的容量が小さい, SAは日本語やDNAデータのような単語に区切ることが難しいデータに対し索引付け可能という長所がある. SAの構築方法に関し, 多くの議論がなされており, Doubling [4] やDC3アルゴリズム [2] など様々な構築法が提案されている. また, [1] ではディスク (2次記憶) を用いる場合に適応させたDoublingとDC3が提案されている. さらに [3] では, DC3を, Message Passing Interface (MPI) を用いて分散化している. 本論文では, Javaソフトウェアフレームワークである Hadoop [5] を用いて, SAを分散構築する.

2 準備

2.1 接尾辞配列構築法

長さ n のテキスト $T[1 \ldots n]$ に対し接尾辞配列 (SA) を構築する. (テキスト中の) 出現位置 j の接尾辞は $T[j \ldots n]$ である. SAは "辞書順が i 番目の接尾辞の出現位置が $j \Leftrightarrow SA[i] = j$" と定義される配列である. 容量はint型で保存すると $4n$ バイトである.

接尾辞が辞書順に並ぶため, 検索は, 検索語に対応する (接頭辞に持つ) SA の範囲 $[l, r]$ を求めることで実現できる. 辞書順に並んだ接尾辞の任意の位置 (i 行, h 列) は, T と SA を用いて $T[SA[i] + h]$ で参照できるので, SA (辞書順に並んだ接尾辞) に対して二分探索を行うことで, 長さ m の検索フレーズの全ての出現を $O(m \lg n +$出現数) 時間で検索できる. 図1はSAでフレーズ "cga" を検索した例である.

*九州大学大学院システム情報科学府
†国立情報学研究所
‡九州大学大学院システム情報科学研究院

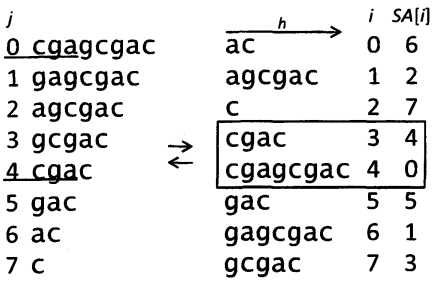

図 1: 接尾辞配列 (SA)

Doubling SAはテキスト中の全ての接尾辞を辞書順に並べることで構築される. つまり, テキスト位置 j を, 文字列 $T[j \ldots n]$ をキーとしてソートする. バケットソートなどを用い全接尾辞を先頭から1文字づつ比較していくと, 各接尾辞の全ての文字は (最高でも) 1度づつしか参照されないため, 計算量は $O(n^2)$ となる.

この手法は, Doubling [4] のテクニックを用い改良できる. 図2の例で, (図では完全にソートされているが) 2文字目までソートを行った時点を考える. ここで $i = 3$ と $i = 4$ の接尾辞を比較することを考える. 次の文字はともに 'a' なので, この文字でもソートは完了しない. しかし, $i = 3$ の3-4文字目の "ac" と $i = 4$ の3-4文字目の "ag" は, $i = 0$ の "ac" と $i = 1$ の "ag" に対応しており, この2つはすでに完全にソートされている. この $i = 0, 1$ の情報を用いれば, $i = 3, 4$ を次のステップで完全にソートできる.

これを実現するため, "名前 (name)" という言葉を導入する. 高さ h (h 文字目) での名前を, "高さ h のソートまでで確定した辞書順 (の範囲の最低値)" と定義する. 図2のnameは2文字目までソートが完了した時点の名前を表している. 名前0が "ac" を, 名前1が "ag" を表すので, この名前を $i = 3, 4$ の接尾辞に参照させれば, 次のステップでソートを完了できる. この参照を行うため, 名前を用いテキストの

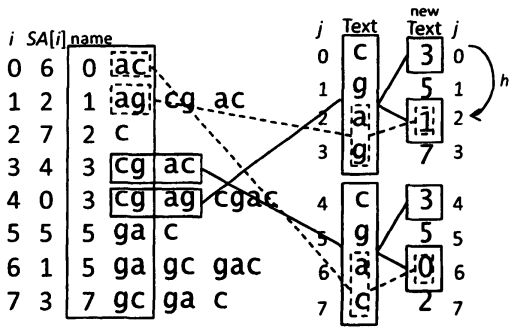

図 2: Doubling

更新を行う. この新しいテキストを参照するときは, h 個先の位置を参照する. 1 つの高さごとに, 名前が意味する文字列の長さが 2 倍になるので, この手法は $O(n \lg n)$ で SA を構築できる.

DC3 アルゴリズム [2] では SA を $O(n)$ 時間で構築する DC3 アルゴリズムが提案されている. DC3 では, 接尾辞 $T[j \ldots n]$ を, $j \mod 3{=}0$ の接尾辞の集合 S^0 と, $j \mod 3{=}1$ の S^1, $j \mod 3{=}2$ の S^2 を合わせた S^{12} に分け, まず S^{12} だけをソートし, その結果を用い S^0 をソート, 最後に 2 つをマージする.

S^{12} のソートのため, まず先頭 3 文字でバケットソート (Radix ソート) する. ここでソートが完了しない場合, Doubling と同様に名前を付け, テキストを更新し, この新しいテキストに対し再帰的にアルゴリズムを適応する (S^1 の後に S^2 を移動させ, テキストが 2 つ連なる形にする). 再帰のたびにテキスト長が 2/3 倍づつ小さくなる. この結果, S^{12} が完全にソートされたとする. すると, $j \mod 3 = 0$ である位置 j の次の $j+1$ の位置はソート済みであるため, そこを参照することで S^0 のソートが完了する. 最後の S^0 と S^{12} のマージでも, $T[j_a \ldots n] \in S^0$ と $T[j_b \ldots n] \in S^1$ の比較なら $j_a + 1$ と $j_b + 1$ の位置が, $T[j_a \ldots n] \in S^0$ と $T[j_c \ldots n] \in S^2$ の比較なら $j_a + 2$ と $j_c + 2$ の位置がソート済みなので, そこを参照すればソートが完了する.

2.2 Hadoop

MPI との比較 分散処理の手段として, MPI と Hadoop [5] がよく用いられる. MPI では, どのプロセスとどのプロセスを通信させるかを厳密に記述できる. Hadoop では, 通信は Hadoop が自動的に割り振るので, MPI のように通信を自由に記述できないが, 逆に言えば, 煩わしい通信のコードを書く必要がなく, プログラミングが非常に単純になる. また, 計算機の故障に対し, Hadoop が自動的に対処してくれるため, 耐故障性が非常に高いという利点がある.

MapReduce Hadoop では Google で考案された MapReduce という処理の仕組みが用いられている. 以下, MapReduce について説明する. まず, 入力となるデータを複数に分割し, (個別の計算機上で動く) 複数の Map タスクに渡していく. Map タスクはユーザが定義した処理を行い, (Key, Value) のペアを作成し, (個別の計算機上で動く) 複数の Reduce タスクに渡していく (Map ステップ). このとき, ⟨Key, Value⟩ のペアは, Key の値によってソートされる (Sort ステップ). 最後に Reduce タスクがユーザが定義した処理を行い, 結果を出力する (Reduce ステップ). この一連の処理をジョブと呼ぶ.

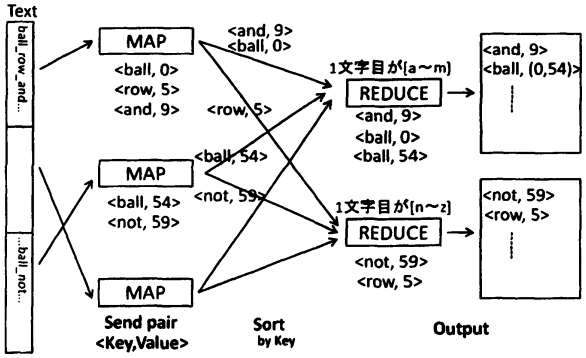

図 3: Hadoop による単純な索引の構築

以下では, Hadoop での MapReduce について, 単純な索引の構築の例を用い詳細に説明する (図 3). まず入力となるテキストを複数に分割し, その断片が Map タスクに渡される. Map タスクは, テキスト断片から単語を抜き出し, ⟨ 単語, 出現位置 ⟩ のペアを作成し, Ruduce タスクに渡していく. このとき, Key である単語でソートが行われるので (辞書順でソートされる), Reduce では同じ単語が連続して現れる. よって, ⟨ 単語, 出現位置のリスト ⟩ のペアを出力すれば索引が完成する. この出力は各 Reduce が動い

ている計算機のディスクに格納される. よって, この出力ファイルは複数のディスクに分散して配置されていることになる.

基本的に Map と Reduce の 2 つの手続きの記述だけで分散処理を実現できるが, 他に, Reduce への分割方法 (あるペアがどの Reduce タスクへ行くか), Key と Value のデータ型および Key の比較方法, 入出力のフォーマットなどもユーザが記述 (指定) でき, これにより複雑な処理を実現できる.

[補足] Reduce タスクは計算機にランダムに割り振られる. 一方, Map タスクは, 可能な限りデータが存在する場所に割り振られる.

3 Hadoop による接尾辞配列構築

単純な分散構築法 Hadoop を用いて SA を構築することを考える. 前述の単純な索引と同様の手法で SA 構築を試みる. Map タスクは, 前述の索引では単語を Key としてペアを作成したが, SA 構築では接尾辞の比較が必要で, Key として最悪の長さが n の文字列を付ける必要が生じる. そこで今度は, テキスト位置のみを Key として付け, Sort のステップ時にテキストを参照したいと考えるが, これは Hadoop の構造上実現できない.

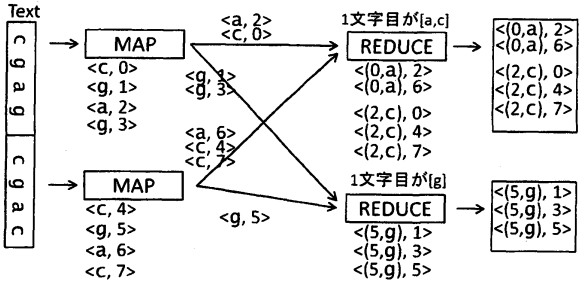

図 4: Hadoop による SA 構築 (1 文字目)

Hadoop を用いた SA 構築は, k 文字づつ比較を行っていくことで実現できる. つまり, Map タスクで Key として k 個の文字を付け, 比較を行い, 結果を出力し, その結果を再度入力して次の k 文字を比較するということを繰り返せばよい. これを実現するために, (Doubling で導入した) "名前 (name)" を用いる. 文字の比較後に, (位置情報 j に対する) 名前を付けることで, 現在までのソート状況の保持が可能になる.

図 4 は 1 文字づつ比較する場合の, 1 文字目の比較の例である. Map では, テキスト断片が入力され, ⟨1 文字目, 位置⟩ のペアを作成し, Reduce へ送る. ここで, (Reduce への分割方法を) 文字が 'a', 'c' ならば上の Reduce, 'g' ならば下の Reduce へ行くよう決めておく. これで 1 文字目のソートが完了するが, 2 文字目以降のソートのために名前を付けておく (名前 0 が 'a', 2 が 'c', 5 が 'g' に対応している).

Tref, Rename しかし, 接尾辞配列がランダムな数列であることが災いし, 辞書順で並んだ位置に対し, テキストを参照すると, (テキストが非常に大きい場合) 毎回ディスクアクセスが必要になるという問題がある.

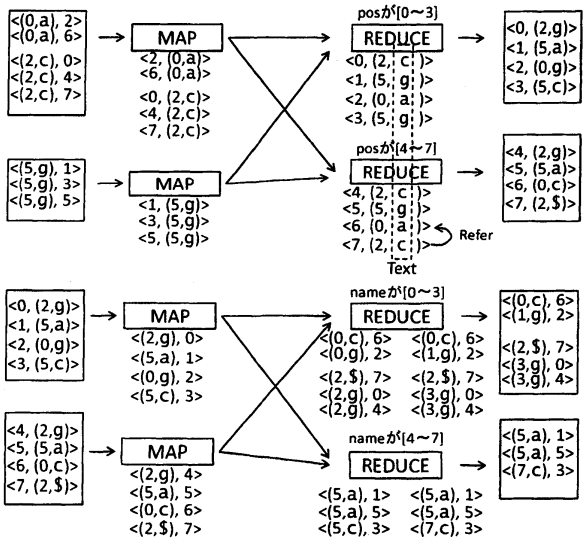

図 5: Tref/Rename

これを防ぐには, 位置情報 (pos) でソートした後にテキストの参照を行えばよい. つまり, pos でソートしテキストを参照するジョブ (Tref) と, name (と次の文字 (chara)) でソートし名前を更新するジョブ (Rename), 2 つの MapReduce ジョブを連鎖させる.

図 5 上は Tref の例である. 図 4 の出力ファイルが, 入力として与えられる. Map で Key と Value を反転し Reduce へ送ることで, pos によるソートを行っている. 結果として Reduce では, pos 順に並んだ chara の列がテキストに対応するため, 次の文字の参照が

行える．具体的には，$\langle(name_B, chara_B), j+1\rangle \rightarrow \langle(name_A, chara_A), j\rangle$ の順で reduce に入ってきたとき，$chara_A = chara_B$ と代入する．

図 5 下は Rename の例である．Tref の出力が入力として与えられ，まず Map で Key と Value を反転し name でソートされ Reduce へたどり着く．ここで，図 4 で，名前 0 が 'a'，2 が 'c'，5 が 'g' に対応したことを思い出してほしい．つまり，例えば (図 5 下の上の Reduce の) (0,'c') は "ac" に，(0,'g') は "ag" に対応しており，正しく 2 文字でソートされていることが確認できる．そしてここで名前を更新することで，この名前は 2 文字分の順序を表すことになるので，次回は 3 文字で比較が行えることになる．

[補足 1] Tref/Rename は基本的には，単純な Sort のジョブに，補助的な操作が加わったものである．Tref/Rename の実行時間はこの Sort ジョブの実行時間に依存する．つまり，Sort ジョブで実行時間を見積もれ，また，Sort ジョブを高速化できない限り，Tref/Rename はあまり高速化できないと推定できる．

[補足 2] 例えば，図 5 の Tref の Reduce の参照で，pos が 4 のペアは，別の Reduce タスクに存在する pos が 3 のペアを参照する必要があるが，これは今の Tref のジョブ時に通信するのではなく，その前のジョブで pos が 3 のペアの chara を覚えておき，今回のジョブで参照させる，もしくは逆に今回覚えておき次回のジョブで参照させればよい．

Doubling, DC3 上記の手法は単純に Doubling に拡張できる．変更点は，次の文字でなく，名前を付加する点と (上記の例で $chara_A = name_B$ と代入すればよい)，ある高さ h の pos でのソート時は，h 個後のペアを参照させる点である (順序対 $(j \mod h, \lfloor j/h \rfloor)$ を Key としてソートすれば実現可能)．結果，1 つの高さは，Tref/Rename 一回づつ，つまり MapReduce2 回で行える．また，ソート済みペアを放棄 (Discard) することで高速化できる [1]．

DC3 では，まず最初の 3 文字のソートで一回づつ Tref/Rename を行う．これは単純に pos に対し対応するテキスト中の 3 文字を付けてソートすればよい．その結果のペア列を再帰させ，ソートされたペア列が返ってくる．この後，S^0 のソート，S^0 と S^{12} のマージを行うが，この 2 つは同時に行え，Tref/Rename 一回づつで良い．結果，1 つの再帰ステップを MapReduce4 回で行える．

4 実験と総括

実験 Hadoop による Doubling, 放棄有り Doubling および DC3 の実装の比較実験を行う．50M バイトの英文に対し，2 台，3 台，5 台の計算機を用い分散させた場合の構築時間を調べる．Doubling では先頭 8 文字を同時比較する改良を行っている．Reduce タスクの分割数は，2 台と 5 台のときは 10，3 台のときは 9 に設定している．

表 1: 構築時間の測定 (sec)

計算機数	2	3	5
Doubling	13,278	9,917	7,193
Doubling+Discard	10,564	7,430	5,714
DC3	7,519	5,735	4,425

表 1 は実験結果である．分散させることで高速化できていることが確認できる．また Doubling より DC3 が高速となった．

総括 本論文では Hadoop を用いた接尾辞配列の分散構築法を提案した．今後の課題は，アルゴリズムの改良，Sort ジョブの高速化である．

参考文献

[1] R.Dementiev, J.Kärkkäinen, J.Mehnert, P.Sanders, "Better external memory suffix array construction," Workshop on Algorithm Engineering & Experiments, Vancouver, 2005.

[2] J. Kärkkäinen, P. Sanders, S. Burkhardt, "Linear work suffix array construction," Journal of the ACM, 53(6), November 2006.

[3] F.Kulla, P.Sanders "Scalable parallel suffix array construction," Parallel Computing, 33(9), September 2007.

[4] U. Manber, G. Myers, "Suffix arrays: a new method for on-line string searches," SIAM Journal on Computing, 22(5), 935-948, 1993.

[5] Hadoop, ⟨`http://hadoop.apache.org/`⟩.

[6] 田中洋輔 "圧縮接尾辞配列 SADIV と大規模文字列検索の効率化," 修士論文, 九州大学大学院システム情報科学府, 2011.