PCクラスタ上でのディスクベースサフィックス木の並列構築方式

澤田　祐介† 田村　慶一† 荒木康太郎† 高木　允†† 北上　始†

† 広島市立大学大学院情報科学研究科　〒 731–3194 広島市安佐南区大塚東三丁目 4 番 1 号
†† 広島市立大学大学院情報科学研究科/日本学術振興会 DC　〒 731–3194 広島市安佐南区大塚東三丁目 4 番 1 号
E-mail: †{sawada,kotaro,makoto}@db.its.hiroshima-cu.ac.jp, ††{ktamura,kitakami}@its.hiroshima-cu.ac.jp

あらまし　サフィックス木は，生物情報学の配列データベースからデータマイニングを効率的に実行可能なデータ構造である．しかしながら，メモリオーバーヘッドによる性能低下の問題から大規模な配列データベースのサフィックス木を構築するのは難しい．この問題を解決するために，本論文では，分散並列環境におけるディスクベースサフィックス木の並列構築方式を提案する．並列構築にはマスタ・ワーカモデルを適用する．本方式を PC クラスタ上に実装し，実験を行った結果，サフィックス木の構築時間を短縮することができ，大規模な配列データベースに対応可能なことを確認した．また，並列構築したサフィックス木（分散サフィックス木）を応用し，データマイニング処理の一機能である曖昧検索を行い，その有効性を評価した．

キーワード　サフィックス木，並列処理，データマイニング，バイオインフォマティクス

Parallel Construction Method of Disk-based Suffix Tree on a PC Cluster

Yusuke SAWADA†, Keiichi TAMURA†, Kotaro ARAKI†,

Makoto TAKAKI††, and Hajime KITAKAMI†

† Graduate school of Information Sciences, Hiroshima City University
3–4–1, Ozuka–higashi, Asaminami–ku, Hiroshima, 731–3194 Japan
†† Graduate school of Information Sciences, Hiroshima City University/JSPS Research Fellow
3–4–1, Ozuka–higashi, Asaminami–ku, Hiroshima, 731–3194 Japan
E-mail: †{sawada,kotaro,makoto}@db.its.hiroshima-cu.ac.jp, ††{ktamura,kitakami}@its.hiroshima-cu.ac.jp

Abstract　The suffix tree is a useful data structure to effectively extract frequent sequential patterns from the sequence databases in the field of molecular biology. However, a problem arises that it is difficult to construct the suffix tree for the large-scale sequence database, because the performance degradation is occurred by the memory overhead. In order to solve the problem, the authors propose a parallel construction method of the disk-based suffix tree on the distributed parallel environment. The parallel model applies a master-worker model. We have implemented the proposed method on a PC cluster, and have evaluated it. The experimental results show the construction time of suffix tree decrease, and we confirmed that the proposed method can support large-scale sequence database. In addition, we applied a suffix tree (a distributed suffix tree) which was constructed in parallel, and evaluated effectiveness of the vague search that was one ability of the data mining processing.

Key words　Suffix Tree, Parallel Processing, Data Mining, Bioinformatics

1. はじめに

サフィックス木は，アミノ酸配列や DNA 塩基配列など生物情報学の配列データベースからデータマイニングを効率的に実行可能なデータ構造である．線形時間でサフィックス木を構築するアルゴリズムとして，Ukkonen [1] のアルゴリズムが広く知られている．しかしながら，サフィックス木は構築に膨大なメモリ領域を必要とするため，メモリオーバーヘッドによる性能低下が問題となっている．ここで，メモリオーバーヘッドとは，メモリ不足のために生じるスラッシングによる頻繁なディスクページアクセスに要する時間のコストを指す．

サフィックス木の構築にはメモリベースの実装とディスクベースの実装が存在する．メモリベースの実装では，少ないメモリ領域と頻繁なディスクへのアクセスのため大規模な配列データ

ベースを扱った場合，サフィックス木は構築不可能かまたは，構築に現実的でない時間を要する．ディスクベースの実装では，大容量のディスク領域を利用するため，メモリベースでは扱えない大規模な配列データベースを扱うことは可能だが，メモリオーバーヘッドの問題は解消されない．この問題を解決するために，本論文では，分散並列環境におけるディスクベースサフィックス木の並列構築方式を提案する．分散並列環境ではネットワークにつながれた複数台の PC（PC クラスタ）を用いた．並列モデルにはマスタ・ワーカモデルを適用した．マスタプロセスは並列処理を管理し，ワーカプロセスは独立してサフィックス木を構築していく．並列構築されたサフィックス木は各ワーカプロセスのディスク上に分散される．

構築されたサフィックス木はデータマイニングに応用される．例えば，サフィックス木を用いたデータマイニング処理として，配列データベースから検索パターンに完全に一致した文字列を発見するパターンマッチングや，頻出な配列パターンを抽出する処理など様々存在する（サフィックス木の応用に関しては Gusfield [2] を参照）．本研究では，検索パターンに対する誤差がユーザにより指定された許容誤差内の文字列を発見する処理である曖昧検索に注目する．配列データベースから並列構築されたサフィックス木を用いて曖昧検索を行うことで，その有効性を評価する．

本論文の構成は以下の通りである．2 章ではサフィックス木の並列構築に関する関連研究について述べる．3 章では本研究で使用したサフィックス木の構造，構築手法について説明する．4 章では 3 章のサフィックス木の並列構築方式を説明し，評価実験の結果を示す．5 章では並列構築された分散サフィックス木の応用とその評価実験の結果を示し，6 章で本論文をまとめる．

2. 関 連 研 究

サフィックス木の構築には，膨大なメモリ領域が必要であり，メモリオーバーヘッドによる性能低下が問題となっている．この問題を解決するために，分散並列環境でサフィックス木を並列構築する研究が注目されている．

Chunxi Chen，Bertil Schmidt は，複数の PC クラスタをインターネットでつなげたグリッド環境下でメモリベースサフィックス木の並列構築を行っている [3]．PC クラスタ内の並列化モデルとして，本研究で扱っているマスタ・ワーカモデルを適用している．Chen らは，CPST（Common Prefix Suffix Tree）と呼ばれる新しいデータ構造を提案している．CPST とは線形時間でサフィックス木を構築する Ukkonen のアルゴリズムを並列処理向けに拡張したものである．CPST の目的は，大規模なサイズのサフィックス木をいくつかの小さいサイズの CPSTs に分割し，CPST をワーカプロセスで独立して構築することである．入力データには DNA の配列データベースを用いており，アルファベット $\{A, C, G, T\}$ の 4 文字で構成された 1 本の配列を対象としている．

Raphael Clifford の DST（Distributed Suffix Tree）[4] も同様に，複数の PC を用いてメモリベースサフィックス木を構築している．DST とは SST（Sparse Suffix Tree）と呼ばれるサフィックス木の部分木の集合であり，SST は各プロセスで独立して構築することができる．SST も CPST 同様，Ukkonen のアルゴリズムを拡張したデータ構造となっている．入力データはランダムなバイナリアルファベットを用いており，1 本の配列を対象としている．また，Clifford は DST の利用について検討しており，DST を用いた Exact set matching などのデータマイニング処理の有効性を導いている．

本研究では，入力データとしてアミノ酸配列など複数本の配列からなる配列データベースからディスクベースサフィックス木を並列構築するため，文献 [3], [4] が対象としているデータとは異なる複数本の配列に対応するサフィックス木の構築手法を提案し，PC クラスタ上で並列構築を行った．また，並列構築したサフィックス木に対してデータマイニング処理に含まれる曖昧検索を行った．

3. サフィックス木

サフィックス木の構造は，入力データとなる文字列の全てのサフィックスをトライ構造で表現したデータ構造である．サフィックスとは文字列 t の任意の位置から t の末尾までの範囲の文字列である．本研究では，ディスク上にサフィックス木の情報を保存するディスクベースサフィックス木を扱う．

ディスクベースサフィックス木の構築手法は様々存在する（文献 [5]~ [7]）が，本研究では，DynaCluster アルゴリズム [8] を用いる．DynaCluster は動的クラスタリング技術を用いている．この技術では，近いノードは同ページ，もしくは隣接したページに格納され，ノード挿入時に起こる $edge - splitting$ と呼ばれる枝の分割によるディスクページアクセスの数を抑える工夫がされている．また，DynaCluster は配列 1 本を対象としたサフィックス木の構築アルゴリズムであるので，n 本の配列データベースを処理するには不向きである．この問題を解決するため，本研究では，配列データベースに含まれる n 本の配列データ $DB = \{t_1, t_2, \ldots, t_n\}$ を 1 本につなげた統合配列 S を作成し，S に対するサフィックス木を構築する．構築後のサフィックス木は複数本の配列を用いたサフィックス木と同じ索引構造になる．

3.1 用語と記号の定義

配列データベース $DB = \{t_1, t_2, \ldots, t_n\}$ において，各要素は，$(sid, \langle SEQ_{sid} \rangle)$ と表現される（n は要素数，sid は配列識別子）．配列データベースの配列識別子の集合は $\Omega = \{1, 2, 3, \ldots, n\}$ と表現する．各 SEQ_{sid} は，配列識別子の値として sid を持つ配列データであり，あるアルファベット Σ で定義される文字列である．配列 SEQ_{sid} の先頭から j 番目の文字は，$SEQ_{sid}[j]$ として表される．表 1 は，$DB = \{t_1, t_2, t_3, t_4, t_5\}$，$t_1 = (1, \langle FKYAKWLCDN \rangle)$，$t_2 = (2, \langle SFVKTAEHNQC \rangle)$，$t_3 = (3, \langle ALR \rangle)$，$t_4 = (4, \langle MSKPL \rangle)$，$t_5 = (5, \langle FSKFLMAWEH \rangle)$ であることを示している．表 1 の例で言うと，アルファベット文字 $\langle A \rangle$ は，t_1, t_2, t_3, t_5 に含まれているので，$SEQ_1[4] = SEQ_2[6] = SEQ_3[1] = SEQ_5[7] =$ "A" と表現することができる．配列データのアルファベット文字数が k 個であるなら，その配列

表 1　配列データベースの例

sid	配列データ
1	$FKYAKWLCDN$
2	$SFVKTAEHNQC$
3	ALR
4	$MSKPL$
5	$FSKFLMAWEH$

データを $k-$ 配列データと呼び，最初の配列は 10− 配列データである．

ここでは，n 本の配列に対するサフィックス木を構築するために，予め，n 本の配列 $SEQ_1, SEQ_2, \ldots, SEQ_n$ をつなげ，1 本の統合配列 $S = \langle s_1, s_2, \ldots, s_n \rangle$ を作成する．ただし，s_i は配列 SEQ_i の末尾にその配列識別子（これを記号 $\#_i$ で表現する）が付与された配列を表す．これにより，1 本の統合配列 S を対象としたサフィックス木を構築する．このとき，S_i を i 文字目から最初に配列識別子が現れるまでの間に対応するサフィックスとする．i をサフィックス番号と呼ぶ．例えば，統合配列 $S = \langle AAC\#_1CAA\#_2 \rangle$ であるとき，$S_1 = \langle AAC\#_1 \rangle$，$S_2 = \langle AC\#_1 \rangle$，$S_5 = \langle CAA\#_2 \rangle$ となる．また，サフィックス S_k に対して長さ l の部分文字列 $S_k^l = \langle s_k s_{k+1} \ldots s_{k+l-1} \rangle$ をサフィックス S_k における長さ l のプレフィックスと呼ぶ．上記の例では，$S_1 = \langle AAC\#_1 \rangle$ において，$S_1^2 = \langle AA \rangle$ である．

3.2　サフィックス木の構築

本節では，DynaCluster のアルゴリズムを説明する．以下では，サフィックス木のあるノードとそのノードにつながっている子ノードの集合をクラスタと呼ぶ．サフィックス木を深さ優先に構築するために必要な情報は，プレフィックスデータベース（以下，PDB）と呼ばれるテーブルに格納される．PDB はクラスタごとに作成される．ルートノードから深さ k のあるノードまでの間に出現する $k-$ 部分文字列を $\langle string^k \rangle$ とすると，$k-$ 部分文字列 $\langle string^k \rangle$ に対する PDB（$\langle string^k \rangle$）には，その次に現れる文字 $\alpha \in \Sigma$ の集合，各文字に対応する数値符号，各文字 α が存在するすべての位置を出現する位置リスト PList の 3 種類の情報が格納されている．また，PList には各 $(k+1)-$ 部分文字列 $\langle string^k - \alpha \rangle$ が存在する統合配列のサフィックス番号 i と S_i に含まれる配列識別子 sid の組が格納されている．

サフィックス木の構築過程において，クラスタごとに作成される PList の要素数はサフィックス木のリーフノードに近くなると急激に減少する．このことから DynaCluster では，予め最適な PList の要素数を閾値 τ として実測し，その値を用いて終端クラスタを作成している．終端クラスタでは，PDB を用いずに，未処理の部分文字列どうしを比較しながら部分木が構築される．閾値 τ を大きくすると終端クラスタ中で発生するノードの分割の頻度が高まり，小さくすると生成される終端クラスタの数が増加する．DynaCluster の文献 [8] によれば，閾値 τ として 1024 が性能上最適であると報告されているので，本研究でもそれを採用する．サフィックス木の構築手順は以下の通りである．

(1)　配列データベース DB に含まれる n 本の配列データを 1 本につなげ，統合配列 S を作成する．次に，統合配列 S に対するサフィックス S_i を全て作成する $(1 \leqq i \leqq |S|)$．これらのサフィックスの集合を $T = \{S_1, S_2, \ldots, S_{|S|}\}$ とする．$k := 0$ とする．

(2)　ルートクラスタを作成するために，T に含まれる各サフィックス S_i における長さ 1 のプレフィックス（先頭 1 文字）S_{i+k}^1 を子ノードとし，それらの親ノードに相当するルートノードを作成する $(1 \leqq i \leqq |S|)$．このとき，サフィックス木を深さ方向に成長させるために必要な PDB を作成する．さらに，$k := k + 1$ とする．

(3)　PDB に含まれる各アルファベット文字に対して以下の処理を行う．

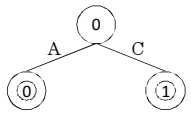
if　PList に含まれる要素数 $>$ 閾値 τ　then do;
　PList に含まれる各サフィックス番号 i に対して，S_{i+k}^1 を子ノードとする非終端クラスタとその PDB を作成する．
　$k := k + 1$ とし，再帰的に (3) の処理を行う．
end
else 終端クラスタを作成する．

(4)　処理を終了する．

3.3　構　築　例

前節のアルゴリズムに従って，サンプルデータとして配列データベース $t_1 = (1, \langle AAC \rangle)$，$t_2 = (2, \langle CAA \rangle)$ を用いたサフィックス木の構築を示す．この例では閾値 $\tau = 2$ とする．まず，前節の (1) の処理によって，統合配列 $S = \langle AAC\#_1CAA\#_2 \rangle$ が得られる．また，統合配列 S に対するサフィックス S_i をすべて作成する $(1 \leqq i \leqq 8)$．次に前節の (2) の処理によって，ルートの子ノードを作成する．

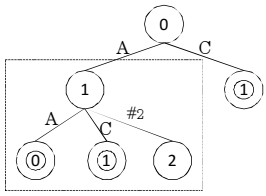
図 1(a) に示されるように，ルートノードを生成し，番号を 0 とする．次に，T に含まれる各サフィックスのプレフィックス S_i^1 を子ノードとして作成する．最初に，サフィックス $S_1 = \langle AAC\#_1 \rangle$ を扱う．プレフィックスは S_1^1 であるので，ルートノードの子として二重丸のノードを生成し，ラベルを $\langle A \rangle$ とする．同時に，サフィックス S_1 のサフィックス番号とその配列識別子の組 $(1, 1)$ が PDB のアルファベット文字 $\langle A \rangle$ に対応する PList に挿入される（図 1(b)）．2 番目に，サフィックス $S_2 = \langle AC\#_1 \rangle$ を扱う．プレフィックス $S_2^1 = \langle A \rangle$ であり，$\langle A \rangle$ は既にルートクラスタに現れているので，ノードを新たに生成する必要がない．よって，サフィックス番号と配列識別子の組 $(2, 1)$ を PDB のアルファベット文字 $\langle A \rangle$ に対応する PList に追加するだけである．3 番目に，サフィックス $S_3 = \langle C\#_1 \rangle$ を扱う．プレフィックスは $S_3^1 = \langle C \rangle$ であるので，ルートノードの 2 番目の子として二重丸のノードを生成し，ラベルを $\langle C \rangle$ とする．同時に，サフィックス S_3 のサフィックス番号とその配列識別子の組 $(3, 1)$ を PDB のアルファベット文字 $\langle C \rangle$ に対応する PList に追加する．4 番目に，サフィックス $S_4 = \langle \#_1 \rangle$ を扱うが終端記号のみの部分文字列であるので，無視する．続いて，サフィックス $S_5 = \langle CAA\#_2 \rangle$ を扱う．プレフィックス $S_5^1 = \langle C \rangle$ であり，$\langle C \rangle$ はルートクラスタに現れているので，ノードを新たに生成する必要がない．よって，サフィックス番号と配列識



(a)

Alphabet	code	PList
A	0	(1,1), (2,1), (6,2), (7,2)
C	1	(3,1), (5,2)
D	2	-
・・・	・・・	・・・
Y	19	-

(b)



(c)

Alphabet	code	PList
A	0	(1,1), (6,2)
C	1	(2,1)
D	2	-
・・・	・・・	・・・
Y	19	-

(d)

図 1　$S = \langle AAC\#_1CAA\#_2 \rangle$ に対する最初の 2 つのクラスタ生成

別子の組 $(5, 2)$ を PDB のアルファベット文字 $\langle C \rangle$ に対応する PList に追加するだけである．以下，すべてのサフィックスに対して同様の処理を行うと，ルートクラスタとそのクラスタにおける PDB は図 1(a)，図 1(b) に示しているようになる．ここで，次のクラスタ生成のため，$k := k + 1$ とする．

次に，(3) の処理を適用することで，深さ方向にクラスタを生成していく．アルファベット文字 $\langle A \rangle$ に対する PList に含まれる要素数 $4 > \tau$ であることから，ノード $\langle A \rangle$ を親ノードとした非終端クラスタを作成する．アルファベット文字 $\langle C \rangle$ に対する PList に含まれる要素数 $2 \leqq \tau$ であることからノード $\langle C \rangle$ を親ノードとした終端クラスタを生成する．図 1(c) のクラスタ（点で囲まれた四角形）は，図 1(b) のアルファベット文字 $\langle A \rangle$ に対して処理を行ったものである．以下は，アルファベット文字 $\langle A \rangle$ に対する非終端クラスタの生成を示す.

PList に含まれる各サフィックス番号 i に対して，プレフィックス S^1_{i+k} を子ノードとする非終端クラスタとその PDB を作成する．まず，S_1 に対して，プレフィックスは $S^1_{1+1} = \langle A \rangle$ である．$\langle A \rangle$ はクラスタに現れていないので，番号を 0 として二重丸のノードを生成する．同時に，サフィックス番号とその配列識別子の組 $(1, 1)$ を PDB のアルファベット文字 $\langle A \rangle$ に対応する PList に追加する．2 番目に，S_2 に対して，プレフィックス $S^1_{2+1} = \langle C \rangle$ はクラスタに現れていないので，番号を 1 として二重丸のノードを生成する．同時に，サフィックス番号とその配列識別子の組 $(2, 1)$ を PDB のアルファベット文字 $\langle C \rangle$ に対応する PList に追加する．3 番目に，S_6 に対して，プレフィックス $S^1_{6+1} = \langle A \rangle$ は既にクラスタに現れているので，ノードを新たに生成する必要がない．よって，サフィックス番号と配列識別子の組 $(6, 2)$ を PDB のアルファベット文字 $\langle A \rangle$ に対する PList に追加する．4 番目に，S_7 に対して $S^1_{7+1} = \langle \#_2 \rangle$ を処理する．これは終端記号であるので，ラベルを配列識別子としてリーフノードを生成する．PDB への追加はしない．ここで，親クラスタにおける PDB のアルファベット文字 $\langle A \rangle$ に含まれるすべてのサフィックスの処理を行ったのでこのクラスタの処理は終了する．以下，$k := k + 1$ とし，再帰的に同様の処理を行う．

図 2 は結果として構築されたサフィックス木を表す．3 つの終端クラスタを点線の枠で示している．左と右のクラスタは新しいクラスタとして生成された．中央のクラスタでは，扱われるサフィックスが 1 つしかないので，新しく生成する必要がない．よって，図 1(c) の二重丸のノードが取って代わる．

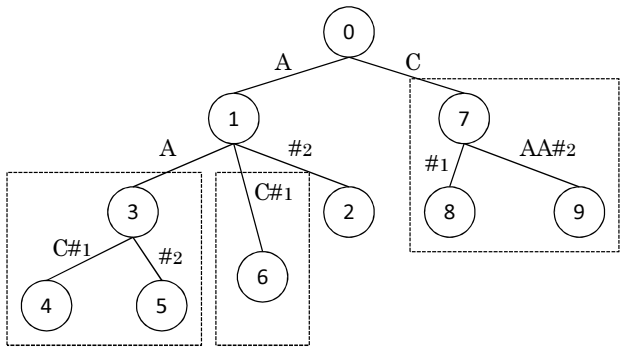


図 2　$S = \langle AAC\#_1CAA\#_2 \rangle$ のサフィックス木

4. サフィックス木の並列構築方式

本章では，前章で示した DynaCluster の並列構築方式を説明する．DynaCluster アルゴリズムは配列データベースから作成したサフィックスの集合と PDB を参照することでサフィックス木を深さ優先に構築していく．これは，サフィックス木を並列構築する際，プロセスが完全に独立してサフィックス木を構築可能なため，並列処理に適したアルゴリズムであるといえる．

4.1 並列構築方式

DynaCluster の並列構築にはマスタ・ワーカモデルを適用した．並列処理を管理し，ジョブを生成するプロセスをマスタプロセスとする．ジョブは配列データベースから作成したサフィックスのプレフィックスをコード化したものである．本研究では，プレフィックス長を 1（先頭 1 文字）に設定している．この場合，アミノ酸配列は 20 種類のアルファベットから構成される配列データベースであるから 20 個のジョブが生成されることになる．マスタプロセスから受け取ったジョブを基にサフィックス木を構築するプロセスをワーカプロセスとする．ワーカプロセスはまず，3.1.1 項 (1) の処理に従って，受け取ったジョブに関するルートクラスタを生成する．この時点では，ジョブ以外のアルファベットは無視するのでルートクラスタは 1 つのノードで構成される．以降，再帰的に非終端クラスタ，終端クラスタを生成していく．そして，新たに受け取ったジョブから生成されたルートクラスタは，以前に構築したサフィックス木のルートノードにつながれ，同様に非終端クラスタ，終端クラスタを生成していく．

構築されたサフィックス木は各ワーカプロセスのディスクに

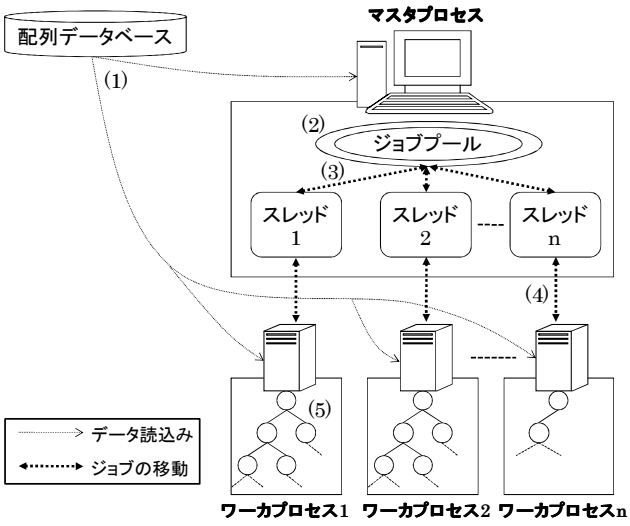


図 3　システム構成図

表 2　実験で用いたデータセット

データセット	データ件数	総長 (byte)
Kringle	72	50354
ZincFinger	744	426011

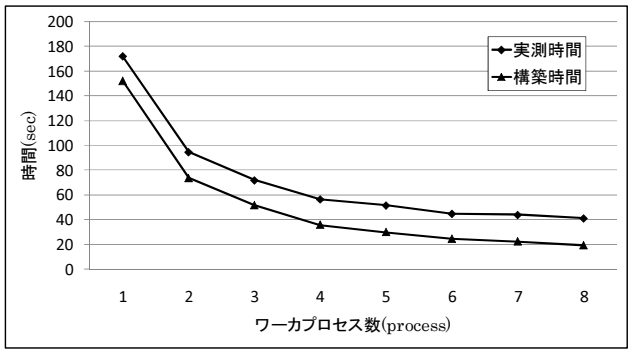


図 4　実測時間 (Kringle)

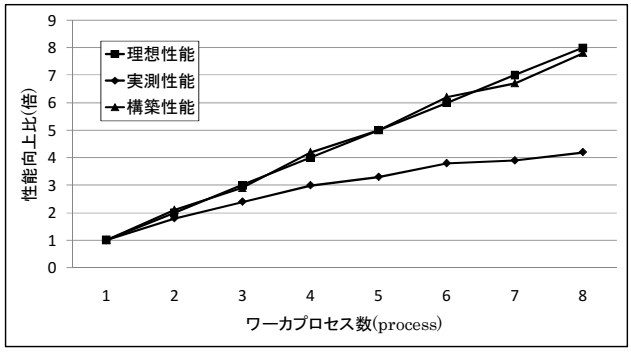


図 5　性能向上比 (Kringle)

保存される．つまり，並列構築後のサフィックス木は各ワーカプロセスのディスク上にサフィックス木の部分木として存在している状態である．このような並列構築によって得られたサフィックス木の部分木は完全なサフィックス木と比較して索引構造の機能は失わない．

マスタプロセスと各ワーカプロセスのデータの送受信にはソケット通信を利用し，ジョブを動的に管理するためにマルチスレッド方式を適用した．サフィックス木の並列構築方式のシステム構成を図 3 に示す．以下，図 3 のシステムの流れについて説明する．

(1)　配列データベースをすべてのプロセスがスキャンする．スキャンしたデータからサフィックスを作成する．

(2)　マスタプロセスでジョブを生成し，ジョブをジョブプールに挿入する．

(3)　各スレッドはジョブプールからジョブをひとつ取り出す．

(4)　各スレッドは対応したワーカプロセスからジョブの要求を受信すると，ワーカプロセスへジョブを送信する．

(5)　各ワーカプロセスでスレッドから受信したジョブのサフィックス木を構築する．構築が終了したら再びスレッドへジョブを要求する．

ジョブプールが空になるまで，(3)，(4)，(5) の処理を繰り返し実行する．

4.2　評価実験

本方式を実際に 4 台の 64bitPC から構成される PC クラスタ上に C 言語で実装し，実験を行った．実験内容は，PC クラスタ上の 4 台の PC を用いて逐次のサフィックス木を構築する実測時間と並列構築の実測時間の性能評価を行った．各 PC の性能は次の通りである．CPU Intel Core 2 Duo E6600（2.40GHz），Memory 1GB×2（合計 2GB），HDD 250GB を搭載し，各 PC 間は 100Mbit/sec イーサネットスイッチで接続されている．OS は Fedora7.0 を使用した．本実験で用いたデータセットは表 2 の通りである．また，これらのデータセットは 20 種類のアミノ酸文字（アルファベット $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$）から構成されるアミノ酸配列であるが，それに加え，それぞれ 1 種類の例外（アルファベット $\{X\}$）を含んでいる．

4.2.1　Kringle データセットを用いた実験

Kringle データセット（以下，Kringle）は 1 台の PC のメモリ領域で十分構築可能なサイズなので，マルチプロセッサの特性から 4 台の PC に各々2 つのワーカプロセスを割り当てた．そのうち 1 台は，マスタプロセスとワーカプロセスの 3 プロセスが動作するように設定した．Kringle を用いた実験で得られた結果を図 4，5 に示す．図 4 はプログラムの実測時間とサフィックス木の構築時間を示している．実測時間はすべてのプロセスのプログラムが終了するまでの時間を，サフィックス木の構築時間は各ワーカプロセスの構築部分の時間の平均を計算したものである．図 5 は図 4 をもとにワーカプロセス数 1 を基準とした性能向上比のグラフである．図 5 中の理想性能はワーカプロセス数倍の性能向上比として設定している．実測性能はプログラムの実測時間から，構築性能はサフィックス木の構築時間から算出したものである．また，約 50KB の Kringle からは，約 5.4GB のサフィックス木が得られる．

実測性能が理想性能に劣る理由として，マスタプロセスのジョブプール生成に要する時間が原因となっている．Kringle のような単一メモリ上でサフィックス木を十分構築できるサイズのデータで性能評価する場合，この時間がボトルネックにな

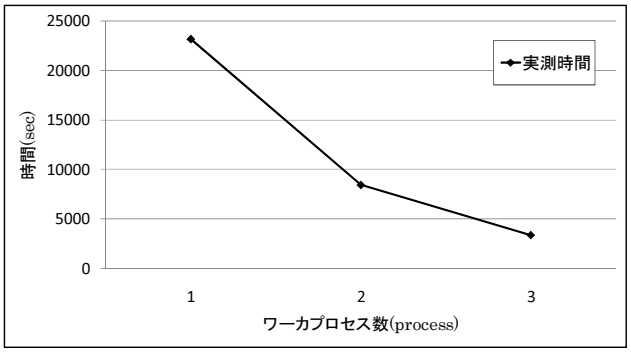


図 6　実測時間 (ZincFinger)

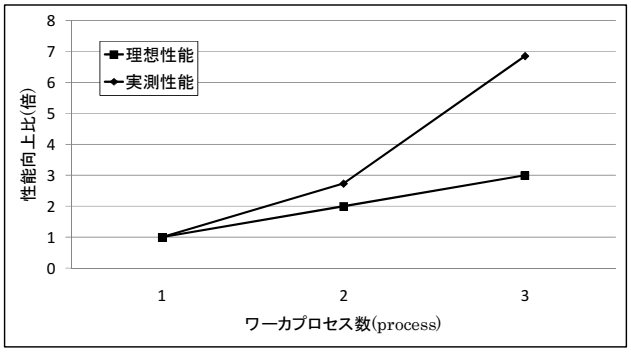


図 7　性能向上比 (ZincFinger)

る．しかしながら，この処理時間は入力データのサイズに比例して増大していくわけではないので，大規模なデータを用いる場合は性能低下に大きく影響はしない．並列処理と直接関係しないこの処理を除いた構築時間を評価すると，理想的な性能向上比が得られている．これは，Kringle は単一のメモリ上で十分構築可能なサイズであるため I/O コストが発生していないためである．ここで，I/O コストとは，I/O による CPU の待機時間のことを指す．また，マスタプロセスとワーカプロセスの通信時間はジョブ（ジョブのサイズは 4byte）のソケット送受信のみであるので性能低下には影響していない．各ワーカプロセスの負荷に着目すると，ワーカプロセス数 2 から 8 に対して 21 個のジョブを動的に配布しているので，ほぼ均等な各ワーカプロセス実測時間が得られていた．

4.2.2　ZincFinger データセットを用いた実験

ZincFinger データセット（以下，ZincFinger）では 4 台の PC のうち 1 台をマスタプロセス，残りの 3 台をワーカプロセスに割り当てた．ZincFinger を用いた実験で得られた結果を図 6，7 に示す．図 6，7 の内容は図 4，5 と同様である．ZincFinger は約 426KB から約 38GB のサフィックス木が得られる．

図 7 から，ワーカプロセス数 2 のとき約 2.7 倍，ワーカプロセス数 3 のとき約 6.9 倍と，ワーカプロセス数を増加するごとに台数以上の性能が得られている．ワーカプロセス数以上の性能が得られた理由として，大幅な I/O コストの削減が挙げられる．逐次でサフィックス木を構築した場合，実測時間の約 95%が I/O コストであり，CPU はほとんど稼働していない．本実験から，サフィックス木を複数台の PC を用いて並列構築することで，メモリ領域を増幅し，I/O コストを抑えることで大規模な配列データベースも扱えることを確証することができた．

表 3　ワーカプロセス数 2 の分散サフィックス木

ワーカプロセス ID	プレフィックス	サイズ (GB)
1	$A, C, E, L, N, Q, S, V, W, Y$	19.448
2	$D, F, G, H, I, K, M, P, R, T, X$	18.794

表 4　ワーカプロセス数 3 の分散サフィックス木

ワーカプロセス ID	プレフィックス	サイズ (GB)
1	A, D, G, I, T, V, W, Y	12.908
2	E, N, Q, R, S, F	13.889
3	C, H, K, L, M, P, X	12.254

5.　分散サフィックス木の応用

前章のサフィックス木の並列構築方式から得られるサフィックス木は，各ワーカプロセスのディスク上に保存される．以下では，並列構築したサフィックス木のことを分散サフィックス木と呼ぶ．例として，前章の ZincFinger の実験で得られた分散サフィックス木の情報を表 3，4 に示す．表 3，4 中のプレフィックスとは，各ワーカプロセスが受け取ったジョブのアルファベットであり，各ワーカプロセスが保持するサフィックス木はプレフィックスから始まる文字列の索引構造を持っている．サイズは，各ワーカプロセスが保持するサフィックス木のファイルサイズである．この分散サフィックス木を併合したものは，完全なサフィックス木と同じ索引構造を持つ．しかし，分散サフィックス木はそれぞれのサフィックス木を独立して走査することが可能である．

5.1　曖昧検索

曖昧検索とは，配列データベースから類似する部分文字列を検索するデータマイニング処理である．入力データとしては，配列データベース，検索パターン長 k，許容誤差 r が与えられる [9]．

5.1.1　配列データベース

以下，曖昧検索で用いる配列データベースについて説明する．曖昧文字表現は，蛋白質の生物学的機能をもつと考えられているモチーフにみられる．現在までに発見されたモチーフは PROSITE データベース [10] に登録され管理されている．例えば，Basic-leucine zipper(bZIP) モチーフ (PS00036) は以下のように表現されている．

$$\langle [KR] - x(1,3) - [RKSAQ] - N - \{VL\} - x - [SAQ] - \{L\} - [RKTAENQ] - x - R - \{S\} - [RK] \rangle \quad (2)$$

$[RKSAQ]$ は，R，K，S，A，Q の中の文字であれば，どれかのアミノ酸文字が対応することを示している．$\{VL\}$ は，他の曖昧文字表現であり，この中に現れない 1 文字のアミノ酸文字が対応することを示している．曖昧文字 $[KR]$ と曖昧文字 $[RKSAQ]$ の間の記号 $x(1,3)$ は，ワイルドカード文字の許容数を表現しており，この場合は，曖昧文字 $[KR]$ と曖昧文字 $[RKSAQ]$ の間に 1 文字から 3 文字までのワイルドカードが許

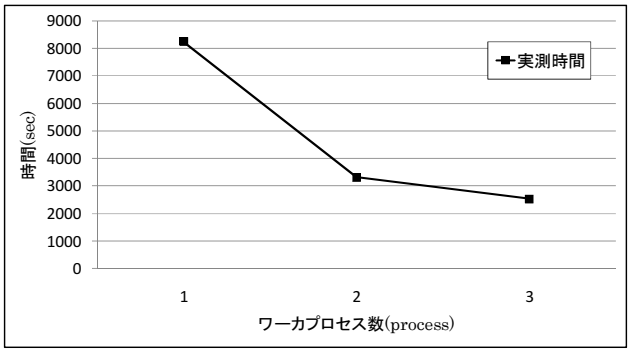


図 8　実 測 時 間

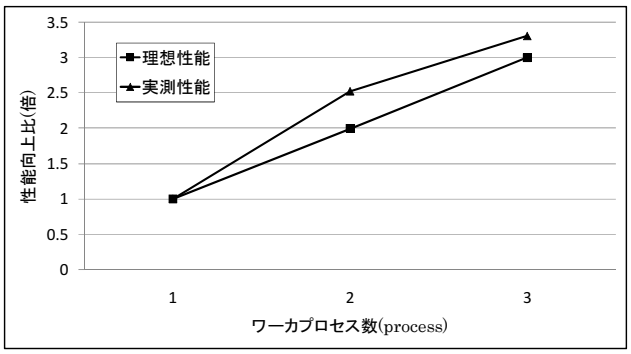


図 9　性能向上比

されていることを示している．また，要素 $[SAQ]$ の後の括弧内の数字は，その要素の繰り返しを表現している．

5.1.2　分散サフィックス木を用いた曖昧検索

サフィックス木を用いた曖昧検索は，サフィックス木を深さ優先に走査していくことで達成される．走査するサフィックス木の深さは最大で検索パターンの長さ k と同じである．

分散サフィックス木を用いた場合は，図 3 のマスタプロセスがユーザが入力した検索パターンとその長さ k，許容誤差 r をワーカプロセスにスレッドを介して送り，ワーカプロセスは独立して曖昧検索を行い，抽出したパターンを出力する．配列データベースはワーカプロセスのみが読み込む．

5.2　評 価 実 験

本実験では，ZincFinger から得られた分散サフィックス木 (表 3，4) を用いて曖昧検索を行う．4.2 節の実験環境で，PC1 台で逐次構築したサフィックス木を基準に PC2 台，3 台で並列構築した分散サフィックス木を用いた曖昧検索の実測時間の性能評価を行った．曖昧検索の実測時間は，全てのワーカプロセスの検索プログラムが終了するまでの時間とする．検索パターンは ZincFinger モチーフ

$$\langle C - x(2,4) - C - x(3) - [LIVMFYWC] - x(8) - H - x(3,5) - H \rangle$$

の一部分である $\langle C-x(4)-C-x(3)-L-x(8)-H-x(3)-H\rangle$ を与え，許容誤差は 1 に設定した．検索パターン長 $k = 23$ である．本実験では，ワイルドカード領域の変化は考慮せず，単純に分散サフィックス木を用いた曖昧検索の実測時間に着目している．また，この入力データから抽出されるパターン数は 316 個である．実験結果を図 8，9 と表 5，6 に示す．図 8 は曖昧検索処理の実測時間，図 9 は図 8 をもとにワーカプロセス数 1 を基準とした性能向上比のグラフである．表 5，6 は分散サフィックス木の各ワーカプロセスから得られた抽出パターン数と実測時間を示している．

表 5　ワーカプロセス数 2 の抽出パターン数と実測時間

ワーカプロセス ID	抽出パターン数	実測時間 (sec)
1	298	3212.21
2	18	3314.07

表 6　ワーカプロセス数 3 の抽出パターン数と実測時間

ワーカプロセス ID	抽出パターン数	実測時間 (sec)
1	30	2190.32
2	13	2531.73
3	273	1905.33

図 9 からワーカプロセス数 2 のとき約 2.5 倍，ワーカプロセス数 3 のとき約 3.3 倍の性能向上比が得られた．曖昧検索のような木を全体的に走査する処理では，表 3，4 の木のサイズと，表 5，6 の抽出パターン数，実測時間の関連づけが示すように，実測時間は抽出パターン数より各ワーカプロセスのサフィックス木のサイズに依存していることがわかる．これより，分散サフィックス木から効率的に曖昧検索を行うためには，均等なサイズの分散サフィックス木を構築しなければならない．よって，サフィックス木を並列構築する場合，マスタプロセスのワーカプロセス数を考慮したジョブの生成（ジョブ細分化）が重要であると考えられる．ジョブ細分化によって各ワーカプロセスの負荷の偏りを低減できる．マスタプロセスのジョブ生成においてプレフィックス長を 1 以上に設定することでジョブ細分化は可能となる．そして，この取り組みによってサフィックス木の構築時間も短縮できると考えられる．本実験では，表 3 の分散サフィックス木はほぼ均等にサフィックス木が配置され，表 4 ではやや偏りがあるといえる．構築過程と同様に実測時間の大部分がディスクページアクセスによる I/O コストが占めていたが，分散サフィックス木を用いて並列に曖昧検索を行うことで，分散サフィックス木を用いた曖昧検索の有効性を示すことができた．

6.　お わ り に

本研究では，マスタ・ワーカモデルを用いたディスクベースサフィックス木の並列構築方式を 4 台の PC から構成される PC クラスタ上に実装し，実験することでその特性を評価した．その結果，Kringle のような単一のメモリ上で構築可能なサイズのデータを用いた場合は理想的な性能向上が得られることを確認した．ZincFinger のような単一のメモリ上では構築不可能なサイズのデータを用いた場合は，複数のメモリ効果による I/O コストの削減に成功し，大幅な性能向上が得られた．これにより，サフィックス木を複数台の PC を用いて並列構築することで，大規模な配列データベースに十分対応できることを確証した．

並列構築後のサフィックス木を応用して，並列構築で得られた分散サフィックス木を用いて曖昧検索を行った．その結果，各ワーカプロセスに保存されたサフィックス木をそれぞれ独立した状態で並列処理が可能であることを示し，曖昧検索の処理時間を短縮することができた．

今後の課題は，ワーカプロセス数を増加したとき，各ワーカプロセスで均等なサイズのサフィックス木が構築されるようなジョブの設定が挙げられる．また，分散サフィックス木に対して，曖昧な頻出配列パターンを抽出することを考慮した曖昧検索を行い，それによって得られた集合から，最小汎化パターン集合を分散並列環境で抽出する方式の検討などが挙げられる．

謝　　辞

本研究の一部は，日本学術振興会，科学研究費補助金（基盤研究 (C)(一般)，課題番号 : 17500097），および文部科学省・科学研究費補助金（課題番号 : 18700094）の支援により行われた．

文　　献

[1] Esko Ukkonen. On-line construction of suffix trees. Algorithmica, Vol.14, No.3, pp.249–260, 1995.

[2] Dan Gusfield. Algorithms on strings, trees and sequences. Computer Science and Computational Biology. Cambridge University Press, 1997.

[3] Chunxi Chen, Bertil Schmidt. Construction large suffix trees on a computational grid. Journal of Parallel and Distributed Computing, Vol.66, No.12, pp.1512–1523, 2006.

[4] Raphael Clifford. Distributed suffix trees. Journal of Discrete Algorithms, Vol.3, pp.176–197, 2005.

[5] Martin Farach, Paoul Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. Proceedings of the 39th Annual Symposium on Foundations of Computer Science, pp.174–183, 1998.

[6] Ela Hunt, Malcolm P. Atkinson, Robert W. Irving. A database index to large biological sequences. In VLDB, pp.139–148, 2001.

[7] Sandeep Tata, Tichard A. Hankins, Jigesh M. Patel. Practical suffix tree construction. Proceedings of the 30th international conference on VLDB, Vol.30, pp.36–47, 2004.

[8] Ching-Fung Cheung, Jeffrey Xu Yu, and Hongjun Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans. Knowl. Data Eng, Vol.17, No1, pp.90–105, 2005.

[9] Kotaro Araki, Keiichi Tamura, Tomoyuki Kato and Hajime Kitakami. Extraction of ambiguous sequential patterns with least minimum generalization from mismatch clusters. The third international conference on SITIS'2007, IEEE Computer society press, pp.32–40, 2007.

[10] http://www.expasy.ch/prosite