Vol.2018-BIO-53 No.4
2018/3/9

de novo アセンブリアルゴリズムにおける *k*-mer カウント処理に用いるハッシュ法の検討

清川 綾乃[1]　外山 史[1]　森 博志[1]　東海林 健二[1]

概要：生物の大規模ゲノム，遺伝情報全体を解読することは，生物科学の分野だけではなく，医学，薬学において重要とされている．ゲノムを解読するためには DNA の断片をシーケンサーと呼ばれる DNA 分子の構造を決める機械に読み込ませ，DNA の断片をつなぎ合わせて元の長い DNA の塩基配列を作る必要がある．その際に既存の DNA の塩基配列を使わずに未知の塩基配列として再構成する方法として *de novo* アセンブリアルゴリズムがある．*de novo* アセンブリアルゴリズムには SOAPdenovo，Velvet，ABySS などがあり，これらは読み込ませる DNA の断片が膨大な大規模ゲノムのアセンブリではメモリ不足になる可能性がある．膨大なメモリを必要とする問題を解決するために，遠藤らのメモリ効率の良いアセンブリアルゴリズムがある．*de novo* アセンブリアルゴリズムでは，DNA の断片を *k*-mer と呼ばれる *k* 文字のパターンに分解し，読み込んだファイル中にそれぞれの *k*-mer がどれくらい存在するか数え上げる処理がある．これを *k*-mer カウント処理と呼び，多大な計算時間とメモリが必要となる．本研究では，遠藤らのアセンブリアルゴリズムで使用するハッシュテーブルを Cuckoo Hashing から他のハッシュテーブルを適用し，消費メモリ量および，計算時間の改善を行った．

1. はじめに

生物の大規模ゲノム (遺伝情報全体) を解読することは生物化学の分野だけでなく医学や薬学において重要であり，このゲノムを解読するためには DNA の全塩基配列を決定する必要がある．ゲノムの解読にはシーケンサーと呼ばれるゲノムを読み取る装置が用いられる．シーケンサーから読み出された大量の DNA の断片 (リード) をつなぎ合わせることによって元の長い DNA の塩基配列（コンティグ）を決定できる．この DNA の断片であるリードを既存の DNA の塩基配列を使わずに，未知の塩基配列として再構成する方法は *de novo* アセンブリと呼ばれ，このアセンブリ（元の配列につなぎ戻す）を行うツールをアセンブラと呼ぶ．次世代シーケンサーは大量の短いリードを出力するが，短いリードをアセンブリするために，Velvet[1] や ABySS[2] などのアセンブラが開発されている．リードをアセンブリする際に *k*-mer と呼ばれる *k* 文字の塩基配列のパターンをリードから読み出し，それぞれの *k*-mer が全リード中から何回出現したかカウントし，その出現回数をメインメモリ上に保存する必要がある．次世代シーケンサーでは大量の短いリードを出力するため，*k*-mer をカウントしメインメモリ上に出現回数を保存する処理に多くの時間と多くのメモリを消費してしまう．このため *de novo* アセンブリは多大なメモリと計算時間を必要とすることがあり，大規模なゲノムを解読するためには数 GB，数 TB のメモリや，数時間，数十時間の計算時間を必要とすることがある．*k*-mer をリードから読み込み，その出現回数をそれぞれ記録するための *k*-mer カウント処理を高速化，また省メモリ化するためにハッシュテーブルが用いられている．

本研究では，*de novo* アセンブリの一つである遠藤らの手法 [3] が *k*-mer カウント処理で用いているハッシュ法を改良する．遠藤らの手法では *k*-mer の保存に Cuckoo Hashing が用いられているが，提案手法では要素の充填率を上げるためのハッシュ法とハッシュテーブルサイズを動的に変更できるハッシュ法，他のアセンブラで用いられているハッシュ法を適用する事により *k*-mer 登録に適したハッシュ法を検討し，消費メモリ量および計算時間の改善を目指す．

2. *k*-mer 整数

読み込んだすべてのリードから *k*-mer と呼ばれる *k* 文字の塩基配列のパターンを作り出し，ハッシュテーブルと呼ばれるデーターベースにこの *k*-mer を登録する．この時，各 *k*-mer のパターンがリード中に出現する回数も同時に保存する．*k*-mer を登録する際，そのまま文字として登録するのではなく *k*-mer 整数と呼ばれる形式に変換して登

[1] 宇都宮大学
Utsunomiya University

© 2018 Information Processing Society of Japan

録する．塩基配列を構成する各塩基 A，C，G，T はそれぞれ表 1 のように 0，1，2，3 に対応させて塩基配列を 4 進数として表現し，この 4 進数で表された塩基配列を 10 進数で表した値が k-mer 整数となる．例として，8-mer の AACCGGTT は 4 進数表記で 00112233，10 進数表記では 1455 という k-mer 整数に変換した後，ハッシュテーブルにこの k-mer 整数を登録する．実際のメインメモリ上ではこの 10 進数の k-mer 整数は 2 進数のビット列として保存されるため，0000 0101 1010 1111 が保存される．各塩基 A，C，G，T を ASCII 文字コードを利用して文字列として表現する場合に比べて，2 進数のビット列に割り当て直す事によって，1 塩基の表現に必要なビット数が 8bit から 2bit まで削減可能である．また，プログラム内で塩基を文字列として扱うよりも 2 進数のビット列として扱うことによって塩基配列の比較や演算を高速に行うことができる．

表 1: 各塩基と対応する 2 進数と 4 進数

塩基	2 進数	4 進数
A	00	0
C	01	1
G	10	2
T	11	3

3. ハッシュテーブル

ハッシュテーブルとはハッシュ表とも呼ばれるデータ構造で，ハッシュテーブル内に要素を高速に格納，参照することができる．ハッシュテーブルは k-mer 整数を元にしたハッシュ値を添え字とした配列であり，ハッシュ値はハッシュ関数を用いて k-mer 整数から算出する．リードから k-mer 整数に変換し，ハッシュテーブルに出現回数と一緒に登録する流れを図 1 に示す． 図 1 ではリードである

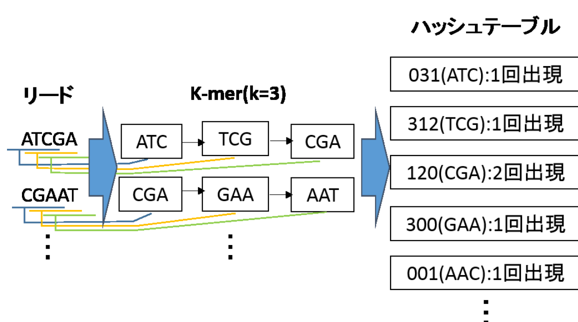


図 1: リードとハッシュテーブルに登録された k-mer

ATCGA と CGAAT から k=3 として 3-mer を取り出している．CGA は 2 つのリードから 2 回出現しているため出現回数を 2 として登録し，それ以外の 3-mer は出現回数を 1 として登録している．

Velvet[1] などのようなアセンブラの多くではリード内の何塩基目からその k-mer が出現したかの位置も併せて登録し，コンティグの長さや精度を高めるが，本手法では登録せず k-mer とその出現回数だけ登録する．また，リードをすべて数え上げるまで出現した k-mer の回数を記録するが，実際にアセンブリに用いられる k-mer は出現回数が多いものを利用する．

4. 従来手法 (Cuckoo Hashing)

従来の遠藤らの研究のハッシュ法では Cuckoo Hashing が用いられていた．このハッシュ法では 2 つの配列と 2 つの異なるハッシュ関数を用いる．配列とハッシュ関数は，それぞれセットになっており，1 対 1 で対応している．すでに格納されている要素を検索する場合，2 つのハッシュ関数で 2 つの配列をそれぞれ見れば良いため，最悪でも 2 回の探索で目的の要素が格納済みかどうか，もしくはハッシュテーブルのどこにあるかがわかる．k-mer カウント処理では要素の検索と挿入が頻繁に行われるため，それらについて説明する．

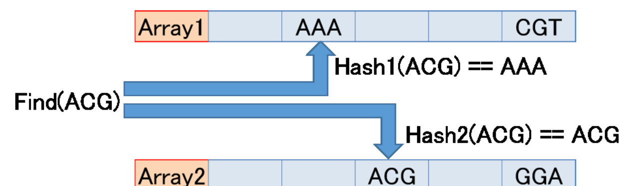


図 2: Cuckoo Hashing の要素の検索

図 2 に Cuckoo Hashing における要素の検索例を示す．図 2 では 3-mer において ACG を検索し，ハッシュ関数である Hash1 と Hash2 を用いて配列 Array1 と Array2 から ACG を探している．要素の検索は用意する配列とハッシュ関数のセットの数だけ (通常 2 セット) で済むため，検

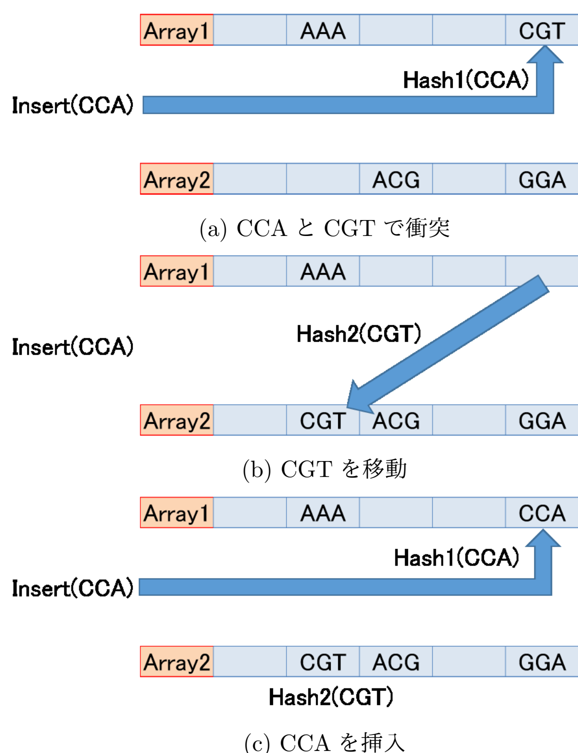


図 3: Cuckoo Hashing の要素の挿入

© 2018 Information Processing Society of Japan

索を定数時間で高速に行える．

図 3 に Cuckoo Hashing の要素の挿入例を示す．図 3(a) は，CCA を挿入しようとした場所に CGT が既に格納されていて衝突した場合である．この場合，既に格納されている要素を別のハッシュ関数 (Hash2) で別の配列 (Array2) に再格納 (図 3(b)) し，Array1 に CCA を挿入する (図 3(c))．再格納先でまた衝突した場合は同様な操作を繰り返し，対象の配列から別のハッシュ関数で別の配列に再格納する．要素を再格納する作業が循環する可能性があり，その場合はハッシュテーブルの大きさを大きくし再ハッシュする必要がある．

5. 提案手法

本研究では，遠藤らの手法において k-mer の数をカウントする処理で用いているハッシュ法について検討する．本研究では遠藤らの手法のハッシュ法の改良であるが，一般的な k-mer カウント処理にも適用可能である．検討するハッシュ法として，動的ハッシュ法の一つである拡張可能ハッシュ法 [4], 遠藤らの手法で用いている Cuckoo Hashing の発展系である Hopscotch Hashing[5] や別の *de novo* アセンブラである ABySS で用いられている Google Sparse Hash[6] を遠藤らの手法に適用し，元々の Cuckoo Hashing と処理時間や消費メモリ量を比較，検討する．

5.1 拡張可能ハッシュ法

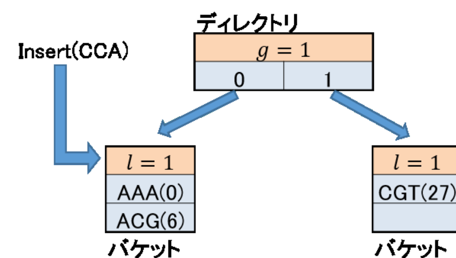
拡張可能ハッシュ法とは動的ハッシュ法の一種である．Cuckoo Hashing は静的ハッシュ法であり，静的ハッシュ法はハッシュテーブルの大きさ，つまり格納できる要素数の数をある値に決め，その大きさを変更する際には格納している要素のハッシュ値をすべて計算し直す再ハッシュと呼ばれる作業を行わければいけない．動的ハッシュ法では，要素を追加するに従ってテーブルサイズを随時大きくしていくが，静的ハッシュ法と異なりハッシュテーブルを一度に全部メモリ上に確保する必要がなく，必要な分だけメモリを追加で確保していき，全要素のコピーが生じず，一部の要素だけコピーと再ハッシュを必要とする．そのため，静的ハッシュ法に比べてあらかじめ格納する要素数を予測する必要がなく，いくつ格納するかわからない場合でも柔軟に対応できる．遠藤らの手法ではハッシュテーブルサイズは読み込むリードファイルのサイズに従って予測していたが，もし予測を誤った場合 k-mer カウント処理を行うことが不可能であったが，拡張可能ハッシュ法ではこの問題を解決することができる．

拡張可能ハッシュ法では要素を格納するバケットのポインタをディレクトリと呼ばれるものに格納し，バケット内の要素数とディレクトリの大きさを depth と呼ばれるものによって管理する．ディレクトリはそれぞれのバケットを指すポインタとビットパターンで構成される．ディレクトリの大きさを管理する depth を g，バケット内に格納できる要素数の数を管理する depth を l とすると，要素 v がどのバケットに格納されるかは式 (1) によって求まる．

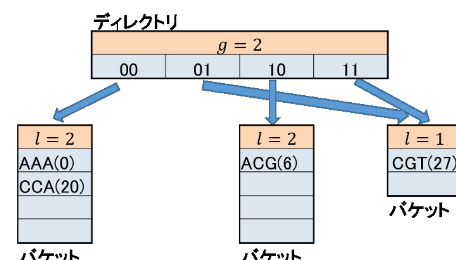
$$Hash(v) = v \mod 2^g \quad (1)$$

ここで，任意のハッシュ関数を使いたい場合，v にそのハッシュ関数で算出したハッシュ値を入れて計算する．また，ディレクトリの大きさは 2^g，それぞれのバケット内に格納できる要素の数は 2^l となる．格納可能な要素の数よりも多い要素をバケットに追加する場合，l を増やし，バケット内の要素数をそれぞれ 2 倍の大きさにする．さらに新しく別のバケットを作り，対象のバケットの中身を新しいバケットと 2 つに振り分ける．バケットの l を増やす際にディレクトリの g より大きくなった場合，g も増やし $l \leqq g$ の関係を保つようにし，ディレクトリの大きさを 2 倍にする．ディレクトリはそれぞれのバケットを指し，どのバケットを指すかはディレクトリのビットパターンによって管理する．バケットの要素の値の　mod 2^g から計算された値の等しいもの同士が同じバケットに入る．g はディレクトリのビットパターンの bit 数を指し，l は g の bit 数のうちいくつ bit 数を使うかを指す．

図 4 に拡張可能ハッシュ法の挿入の例を示す．AAA と ACG，CGT がハッシュテーブルに登録されているとして，新たに CCA をハッシュテーブルに登録する場合，CCA の k-mer 整数を要素 v として (1) 式から計算すると AAA と ACG が既に登録されているバケットに登録される（図 4(a)）．CCA を登録しようとするバケットの大きさが現在 2 つしかなく，このままでは登録できないため，図 4(b) はバケットの大きさを二倍にし，別に新しくバケットを作り，バケットの中身を 2 つで振り分けている．



(a) CCA の挿入



(b) バケットを拡張

図 4: 拡張可能ハッシュ法の挿入

要素の検索は (1) 式より対象の要素が格納されているバ

ケットを探し，そのバケットの中から探すことによって行う．

5.2 Hopscotch Hashing

Hopscotch Hashing[5] は Cuckoo Hashing の発展系のハッシュ法である．オープンアドレス法の一種であり，Cuckoo Hashing では，要素のハッシュ値が衝突した場合，別のテーブルに入れるためキャッシュの局所性が効きづらい特徴があったが，Hopscotch Hashing では同じ配列に入れるため，要素が衝突している場合でもキャッシュの局所性を活かすことができる．また Cuckoo Hashing では，ある程度のハッシュテーブルの充填率で衝突した場合，要素を入れる場所を解決することができず，充填率が 50%ほどで要素を追加することができなくなり，テーブルのメモリ効率が悪い．Cuckoo Hashing で衝突した要素を別のテーブルに入れる代わりに，衝突した要素のバケット近辺のバケットに新しく要素を追加するのが Hopscotch Hashing である．配列上のそれぞれのバケットは衝突した要素がどこのバケットに入ってる可能性があるかを示す Hop info というものを持つ．Hop info は bit 列であり，挿入時において，要素を入れようとしたときに既にそのバケットが埋まっていた場合，そのバケットに入れる代わりに後ろのどのバケットに要素を入れたかを示す．Hop info の先頭 bit が 1 であればそのバケットにそのハッシュ値の要素が入っていることを示し，次の bit が 1 の場合はそのすぐ後ろのバケットにそのハッシュ値の要素が入っていることを示す．つまり，Hop info はそのバケットから後ろ向きにどこのバケットにその対象バケットの要素が入っているかを示し，Hop info の bit 列の先頭 bit を 1 番目として 1 が立っているビットを k 番目とすると，対象バケットの位置+(k-1) にアクセスすることで衝突した要素を見つけることができる．-1 をするのは Hop info の先頭ビットをその Hop info に付随しているバケットにそのハッシュ値の要素が入っているかどうかを示すためである．そのため，目的の要素を探す場合，最悪でも Hop info のビット列のビット数分だけ探索を行えば良いため定数時間で済む．

図 5 に Hopscotch Hashing の挿入例を示す．配列の上の行に k-mer，下の行に Hop info を示している．また Hop info は 4bit とした．図 5 では CCA を挿入しようとした場所に AAA が既に格納されていて衝突した場合である．挿入する場合，後ろから開いている場所を順番に Hop info を更新して前に持ってくる．まず，AAA から後ろの配列で空いているバケットを探す (図 5(a))．一番近いバケットが GGA のすぐ後ろである．GGA のすぐ後ろのバケットに入れることができる要素は，そのバケットから Hop info の bit 数分だけ前のところまでの要素なので CGT，ACG，GGA であるが，GGA のすぐ後ろから一番遠いところを探すので CGT が選ばれる．CGT を GGA の後ろに移動し，CGT が元々あったバケットと新しく CGT を入れるバケットの Hop info を更新する (図 5(b))．Hop info は対象のバケットに本来入るはずであった要素が衝突で後ろのどのバケットに入っているかを示すので，CGT を後ろに 3 つ移動し，CGT が元々入っていたバケットの Hop info は 3+1(先頭 bit は Hop info が入っているバケットを示すので+1 する) の 4bit 目を 1 にするので，1000 から 0001 に書き換える．そして次に AAA を CGT があったバケットに移動し，Hop info を更新する (図 5(c))．AAA が元々入っていたバケットの Hop info は 1010 なので，1 つ後ろにしたため 1+1 で 2bit 目を 1 に書き換える．つまり，1010 から 1110 に書き換える．AAA が入っていたバケットが空いたのでそこに元々挿入しようとしていた CCA を挿入する (図 5(d))．以上の操作で挿入を行う．

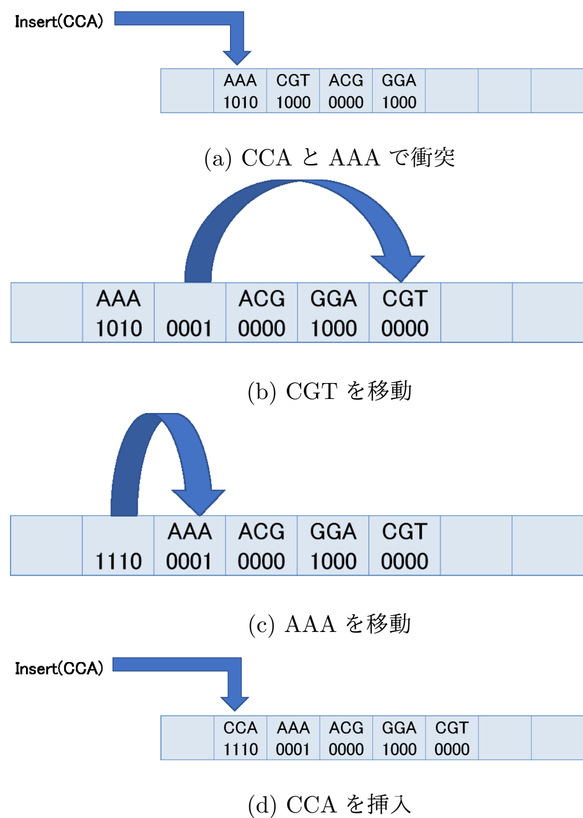


図 5: Hopscotch Hashing の挿入

要素の検索はハッシュ値からバケットを探し，そのバケットの Hop info の 1 が立っている bit が示す後ろのバケットだけを調べれば良い．

5.3 Google Sparse Hash

Google Sparse Hash[6] は Google 社が作成したハッシュテーブルライブラリであり，*de novo* アセンブラである ABySS でも使われている．Google Sparse Hash は要素 1 つ当たり 1bit から 2bit のオーバーヘッド，つまり要素 1 つ当たりにかかる別の余分なメモリとして 1bit から 2bit だけしか消費しないためメモリ効率が良い．特に論文等は

無く，具体的なアルゴリズムは公開されていないため，実際には提供されているライブラリを利用した．

6. 実験結果

従来の遠藤らの手法と提案手法で処理時間およびバケット数, 消費メモリ量の比較を行った結果を表 2 に示す．使用したデータは次世代シーケンサから抽出したヒト 14 番染色体のリード (101bp) である．リード数は 36,504,800 であり，k-mer のパラメータは $k = 64$ に設定した．従来手法ではハッシュテーブルのバケット数は入力するリードデータのサイズから 2 次関数のフィッティングによって決定され，このリードデータからはバケット数が 197,237,849 となった．Cuckoo Hash では同じバケット数のハッシュテーブルを数本用いる．従来手法では 3 本用いるため総バケット数は 591,713,547 となる．拡張可能ハッシュ法のバケット数は自動で 2^g の大きさで拡張され，最終的にバケット数は 536,870,912 となった．どちらとも実際に要素が入っているバケットは 237,188,137 である．処理時間は 10 回測定した平均値であり，ハッシュ関数は従来手法と提案手法ともに同じものを使用している．拡張可能ハッシュ法はバケット数を自動で動的に拡張していくが，初めからバケット数を指定することもできるため，遠藤らの手法のバケット数の予測を元にバケット数を初めから最適化した結果も示す．

表 2: 処理時間とバケット数

	処理時間	バケット数	メモリ量
Cuckoo Hashing	354s	591713547	6722MB
拡張可能ハッシュ法	342s	536870912	14920MB
拡張可能ハッシュ法 (バケット数最適化)	301s	536870912	14920MB
Hopscotch Hashing	298s	289939638	6813MB
Google Sparse Hash	495s	536870912	3939MB

表 2 より従来手法の Cuckoo Hashing に比べて，Google Sparse Hash 以外はすべて処理時間を短くすることができた．その中でも Hopscotch Hashing を用いたものが一番処理時間を短縮することができ，15%短縮できた．拡張可能ハッシュ法は，要素数に応じてテーブルサイズを適切な値に変更できるため，処理時間を Cuckoo Hashing よりも短縮できたと考えられる．また，拡張可能ハッシュ法のバケット数を初めから最適化した場合，テーブルサイズを拡張する処理を行わないため，最適化しないよりもさらに処理時間の短縮を行えたと考えられる．Hopscotch Hashing は比較を行ったハッシュ法，ハッシュテーブルの中で唯一キャッシュが効きやすい構造で，k-mer の出現回数を読み込む処理が早く済むため，一番速く処理を行うことができたと考えられる．拡張可能ハッシュ法はバケット数が 2^n の値しか取れなく，これより少ないと k-mer を格納することができないため，これが一番最適なバケット数となる．バケット数は一番効率が悪い Cuckoo Hashing が一番多くのバケットを用意しなければならなく，次に拡張可能ハッシュ法と Google Sparse Hash が多く，一番少なくバケット数の効率が良かったのが Hopscotch Hashing であった．消費メモリ量はバケット数の数に対して Google Sparse Hash が一番少なく，その次に Cuckoo Hashing，そしてほぼ同じくらいの量で Hopscotch Hashing，大きく差をつけて拡張可能ハッシュ法となった．Google Sparse Hash では処理時間が従来手法よりもかかってしまったが，消費メモリ量は一番少ない．よって，消費メモリ量と処理時間のバランスが取れてるのが Hopscotch Hashing であると考えられる．

7. 終わりに

本研究では，大規模なゲノムのアセンブリの消費メモリ量，及び処理時間を削減するために k-mer カウント処理に用いられるハッシュ法を検討した．k-mer がリード中にどれくらい出現するか予測しなくても良い利点を持つ拡張可能ハッシュ法をはじめ，既存のハッシュ法よりも消費メモリ量が少ない Google Sparse Hash，処理時間を大幅に短縮することができた Hopscotch Hashing など，既存の手法よりも優位な点を見つけることができた．検討の結果，消費メモリ量を重視するならば Google Sparse Hash，消費メモリ量と処理時間のバランスが一番良いと考えられるのは Hopscotch Hashing と考えられる．本実験では，一番時間がかかる k-mer カウント処理のみしか処理時間を計測していないため，プログラム全体の de bruijn グラフを構築する部分ではハッシュ法の変更がどれくらい影響するのかを検討することが今後の課題である．また，今回比較したアルゴリズム以外にもハッシュ法，ハッシュテーブルは存在するため，それらを適用し検討することも今後の課題とする．

参考文献

[1] Daniel R. Zerbino and Ewan Birney, ”Velvet: Algorithms for de novo short read assembly using de Bruijn graphs”, Genome Res. 18, pp821-829, 2008.

[2] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein and Steven J. M. Jones, ”ABySS: A parallel assembler for short read sequence data”, Genome Res. 19, pp1117-1123, 2009.

[3] Yuki Endo, Fubito Toyama, Chikafumi Chiba, Hiroshi Mori and Kenji Shoji,“A Memory Efficient Short Read De Novo Assembly Algorithm”, IPSJ Transactions on Bioinformatics, Vol.8, pp.2-8, 2015.

[4] Ronald Fagin, Jurg Nievergelt, Nicholas Pippernger, H.Raymond Strong, ”Extendible hashing?a fast access method for dynamic files”, Journal ACM Transactions on Database Systems, vol.4 Issue 3, pp.315-344, 1979.

[5] Maurice Herlihy, Nir Shavit and Moran Tzafrir, ”Hopscotch Hashing”, Lecture Notes in Computer Science, vol.5218, pp350-364, 2008.

[6] Google,”Google Sparse Hash”, http://google-sparsehash.sourceforge.net/.

© 2018 Information Processing Society of Japan