# データベースにおける疾患関連解析のためのゲノム型処理実装と評価

宇治橋 善史† 河場 基行† 原田 リリアン†

† 株式会社富士通研究所 〒211-8588 川崎市中原区上小田中 4-1-1

E-mail: †{ujibashi, kawaba, harada.lilian}@jp.fujitsu.com

**概要** 近年、次世代シーケンサやゲノム解析技術の進歩により大量のゲノム情報・診療情報が蓄積されるようになった。これらの情報を活用することでゲノムレベルでの個人差の理解が進み、知見を医療に活用することでゲノム医療・個別医療が実現されつつある。ゲノム医療をさらに高度に発展させるためには、未知のゲノムと疾患、生活習慣等の関連を研究して医療に適用するゲノム関連解析が重要になっているが、現状はこのような大量データの分析・集計に非常に時間がかかる問題がある。これを解決するための技術として、我々は RDBMS 上に格納したデータのゲノム分析を高速化する、ゲノム型という PostgreSQL 上のユーザ定義型を提案した[1]。本稿は[1]で提案したゲノム型を PostgreSQL 上に実装し、そのうえで実際のデータを扱うためにさらに必要となった開発技術について紹介する。その一つは、ヒトゲノムの変異を効率的に格納するデータ構造である。変異には様々なパターンが存在し偏りがあるが、我々が提案したデータ構造によってこのようなデータを効率的なサイズで格納しつつ、分析処理も高速に実行することが可能になる。また、大量のゲノムデータを高速に分析するために、PostgreSQL に備わる並列機構や、Xeon Processor に備わる SIMD 命令を適用した。性能評価の結果、分析で大部分の時間を占める集計処理が、大規模データに対して数秒で完了することを確認した。

**キーワード** ゲノム、変異、SNP、関連分析、GWAS、RDBMS、PostgreSQL、並列処理、SIMD

## 1. はじめに

近年のシーケンシングや解析アルゴリズムの技術革新により、ゲノム情報を取得する性能的及び経済的コストが大幅に下がり、これまでにない膨大なゲノム情報が蓄積されつつある。ヒトゲノムは 30 億の DNA 配列で構成されている。各 DNA はアデニン、グアニン、チミン、シトシンの 4 種類の塩基で特徴づけられ、DNA 配列はこの 4 塩基のアルファベット（AGCT）の配列で表現することができる。個人毎の塩基配列の違い（これを変異と呼ぶ）が遺伝的な個人差の由来になっており、現在、そのような変異はヒトの塩基配列のなかで数千万か所あると考えられている。この変異の種類は、SNP(Single nucleotide polymorphism)と呼ばれる一塩基の置換や、INDEL(insertion-deletion)と呼ばれる塩基の挿入・欠失、CNV(Copy Number Variation)と呼ばれるコピー数多型など様々なパターンが知られている。変異と疾患の関連解析を行うコホート研究やケースコントロール研究により、ゲノムと疾患の関連研究がおこなわれてきたが、近年は生活習慣の情報や身体的特徴等の情報も組み合わせることでより複雑な疾患因子を探索する研究が注目を集めている。我々は、これらの多様なデータの組み合わせを関連解析するために、データを RDBMS に格納して利活用すると同時に、RDBMS 上で多様なデータの組み合わせ条件下の関連分析を高速に実行することができる新しい手法を提案した[1]。

ゲノムと疾患の関連分析は、ターゲットになる病気と関連するジェノタイプ（変異の型）を見つけることである。この分析は、全体群をターゲットとなる病気を持つ疾患者群と病気を持たない健常者群に分け、各群のジェノタイプ分布に相違がある変異を見つけることで行われる。図 1 はある病気の疾患者群（ケース群）と健常者群（コントロール群）の各変異におけるジェノタイプ分布の例を図示している。この例では、変異 0 と変異 1 ではケース群とコントロール群でジェノタイプ分布の相違がみられないが、変異 N では両群の分布に明らかに有意な相違がみられる。よって、変異 N がこの病気に関連していると予想することができる。この疾患と変異の関連分析では、帰無仮説下における（コクラン・アーミテージ検定や $\chi^2$ 検定などの）有意差検定を実行する。有意差検定では、各変異について、ケース群とコントロール群のジェノタイプ分布から生成できる分割表から p 値を算出し、p 値が有意水準を下回るかどうかを判定する。p 値が有意水準を下回ればその変異が病気と有意な関連があるとみなすことができる。

この有意差検定は 2 ステップで実行する。最初のステップは（1）各ジェノタイプを持つ人数を集計して

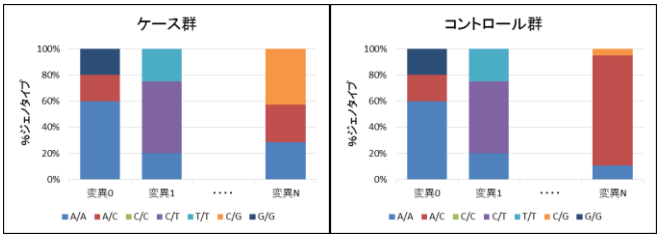


**図 1: ジェノタイプ分布の例**

分割表を作成する処理であり、次のステップは（２）分割表から p 値を計算する処理である。近年取得可能になっている大規模データを処理する場合は、（１）の処理が支配的になり、ナイーブな方法だと数日間要する可能性がある。関連解析を行う研究のなかでゲノム中の全変異を対象とした研究を GWAS（Genome Wide Association Study）と呼ぶが、GWAS 用の解析ツールとして標準となっている plink 等のツールではこのような大規模データを取り扱うための高速化の取り組みがなされて、近年、一定の成果が出ている[3]。これらのツールではゲノムデータを VCF[4]、pad[3]、bad[3]といったフラットなファイル形式で扱うのが通常である。しかしながら、関連分析では母集団をどう選択するかが統計結果の正確性に大きく影響を及ぼすので、正しい統計結果を得るために母集団の選択と統計処理を繰り返しながら意味ある結果を探索する必要がある。このようなデータの選択を繰り返す処理をするためには、ゲノムデータ、身体的特徴データ（性別、人種、年齢等）、診療データ（疾患有無、診断値等）のデータを一つの RDBMS に格納して管理するほうが容易に処理可能である。近年、必要な全データをデータベースに格納して管理する手法がいくつか研究されているが[5][6]、大規模なゲノムデータを高速に関連分析することに着目したものはまだない。そこで我々は、データベースに大規模なゲノムデータを格納したまま、そのデータに対して関連解析処理を高速に行う技術の研究開発に取りくみ、前論文[1]において、ジェノタイプを効率的に格納し、ジェノタイプの分布集計を高速処理することができる新しいゲノム型と集約関数を提案し、PostgreSQL の拡張ユーザ定義型として実装した。

しかしながら、実際のデータを扱うためには、可変長のジェノタイプを持つ変異を取り扱えるようにすることや、大規模なデータを扱うためのさらなる高速化が必要である。本稿では、このような実データを扱うために前論文のプロトタイプを発展させて実現した、可変数ジェノタイプを持つ変異に対応するデータ構造や並列化による高速化について紹介する。

本稿では、第 2 章において前論文のゲノム型とゲノム型集約関数を紹介し、第 3 章から第 5 章で、本稿の本題である実データを取り扱うためのデータ型と並列化による高速化とその評価を報告し、第 6 章で本稿をまとめる。

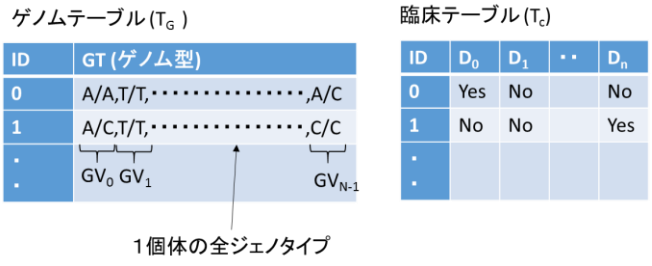


図 2: ゲノム型のデータベーススキーマ

## 2. ゲノム型と集約関数

我々は前論文[1]において、PostgreSQL の拡張として、ゲノム型とゲノム型集約関数を提案し、この拡張が RDBMS 上の効率的なゲノム分析に有効であることを示した。ゲノム型のテーブルスキーマを図 3 に示す。ゲノムテーブル($T_G$)はゲノム情報が格納されたテーブルであり、各行には個体の識別子（ID）と、その個体が持っている N 個の変異（$GV_0 ... GV_{N-1}$）のジェノタイプを 1 レコードにパックしたデータ（GT）が格納されている。各個人の身体的特徴、生活習慣病、臨床情報等の情報は、図 3 の臨床テーブル($T_C$)の例のように、管理を容易にするために他のテーブルに格納する。SQL 1 は、ゲノム型集約関数 fjgeno_count()関数を利用して集計を行うときの SQL 文例を示しており、この例では、臨床テーブル $T_C$ の病気 $D_0$ に罹患している患者の全変異におけるジェノタイプ分布を集計する。我々が提案したゲノム型に使って 1 個体の全ジェノタイプを 1 レコードに格納することで、ゲノム型集約関数が処理を効率的に行うことが可能になり、データベース上での集計処理を高速化することに成功した[1]。

```
SELECT fjgeno_count(TG.GT) FROM TG
WHERE TG.ID = Tc.ID AND Tc.D0 = YES;
```

SQL 1: ゲノム型集約関数の利用例

## 3. ゲノム型の新データ構造

### 3.1 ディクショナリエンコーディング

#### 3.1.1 データ構造

図 3 に示したように、前論文で紹介したゲノム型は、1 個体の全変異のジェノタイプ文字列をデリミタ（カンマ）で区切って 1 行にし、1 レコードとして格納している。このゲノム型より効率的にジェノタイプを格納するために、我々は、ディクショナリで数値と文字列パターンを管理し、ゲノム型にジェノタイプを数値

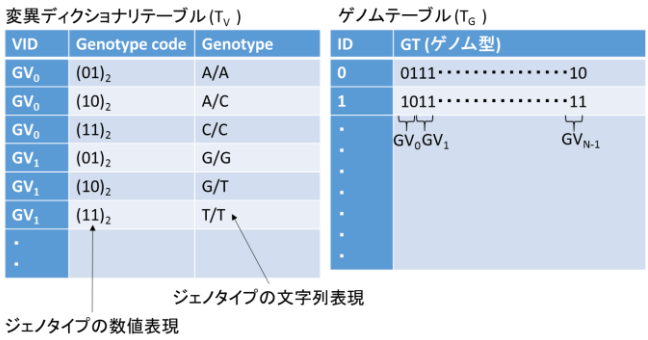


図 3: ディクショナリエンコーディング

にエンコードしてデータを圧縮する方式を導入した。多くの変異のジェノタイプパターンは数パターンに限られる。例えば、最も頻出する 2 アレルで構成される SNP は、'A/A', 'A/C', 'C/C'の 3 パターンである。このパターンの SNP は文字列表記で 3 バイトのサイズが必要で、変異を区切るデリミタのサイズを足すと 1 変異あたり 4 バイト（=32 ビット）必要である。一方で数値にエンコーディングすると 2 ビットのサイズしか必要としないので、大部分の変異がこのような 3 パターンで表せるとすると、サイズ を約 1/16 に圧縮可能である。

図 2 は、全変異が 3 パターンのジェノタイプを持つ場合にディクショナリエンコーディングしたテーブルスキーマの例である。変異ディクショナリテーブル（$T_V$）は、N 個の変異（$GV_0 \ldots GV_{N-1}$）のジェノタイプと数値コードのマッピング情報を管理している。この例では変異 $GV_0$ にジェノタイプパターン'A/A'、'A/C'、'C/C'、変異 $GV_1$ には'G/G'、'G/T'、'T/T'があり、各変異のジェノタイプにはそれぞれ、$(00)_2$、$(01)_2$、$(11)_2$ の 3 種類の 2 ビット数値表現が割りついている。このマッピング情報を用いて、N 変異のジェノタイプの情報から構成されるゲノム型を 2N ビットサイズの配列にエンコーディングして、ゲノムテーブル($T_G$)のゲノム型列(GT)に格納する。

#### 3.1.2 集計性能評価

3.1.1 節で述べたデータ構造は、サイズ削減だけではなく、集計処理の高速化にも寄与する 。それは、重い文字列のパーシング処理が不要になるからである。我々は、前論文の文字列ベースのゲノム型と今回のディクショナリエンエンコーデングベースのゲノム型で SQL1 のクエリの実行時間を計測して性能比較を行った。PRIMERGY RX2540 M1（Xeon E5-2660 3 @2.60GHz, 576GB メモリ）上で、PostgreSQL の設定を shared_buffers = 128GB とし、10 万人×10 万変異のデータに対して SQL1 を実行した。表 1 の実行時間計測結果に見られるように、テキストをパースする前論文の方式では、約 46 秒かかっていた集計処理が、本稿のディクショナリエンコーディング方式では約 8 秒に短縮し、約 5 倍高速化することが確認できた。

**表 1: SQL1 の実行時間**

| ゲノム型 | 実行時間 (sec) |
|---|---|
| テキスト（前論文） | 45.743 |
| ディクショナリエンコーディング（本稿） | 8.331 |

### 3.2 パターン数可変なジェノタイプ対応

#### 3.2.1 動的データ構造

3.1 節でみたように、ジェノタイプを固定長 2 ビットコードでディクショナリエンコーディングする方式により、サイズを圧縮し 集計処理も高速化することができる。大部分の変異のジェノタイプパターンは 3 種類であり 2 ビット長でエンコーディングすることが可能だが、しかしながら、3 種類よりも多いパターンを持つ変異も知られている。SNP や INDEL といった典型的な変異にも 3 種類以上のものが存在するし、特に、STR(Short tandem repeat)と呼ばれる変異にはいくつもの繰り返しパターンがあり百パターンを超えるものも存在することが知られている。このようなデータを固定長 2 ビットコードでエンコードすることはできない。このようなデータを扱うためのナイーブな方法が 2 つある。ひとつは全変異のなかで最大パターン数を持つ変異を格納するのに必要な固定ビット長を持つビットコードを全変異で利用する方式である。もうひとつは、各変異の全ジェノタイプパターン全てを格納するのに必要なサイズの領域長を変異毎に決めて、変異毎に可変なビットコードを用いることである。しかしながら、どちらの方式とも、データを挿入する前に全変異の最大ジェノタイプパターン数に応じたビットコード長が決定している必要がある。よって、もし新しい個体を追加で挿入するときにより長いビット長を必要とするパターンを持っていたら、変異ディクショナリを更新するだけではなく、ゲノムテーブルに既に格納されているゲノム型を全て再構築してビット長を拡張しなければならない。ゲノムデータを新しく取得する度に、新しいジェノタイプパターンが出現する可能性が十分にあるので、そのたびにゲノムテーブルの全データを再構築するのは非常に重い処理なので、その度に再構築するのは現実的ではない。

この問題を解決するため、我々は、パターン追加があっても再構築を必要としない動的なデータ構造とデータ更新手法を開発した。この手法では、最初に決定したビットコードの領域を拡張する必要があるようなパターンが出現すると、データ構造の最後尾に格納領域を追加して、その領域に新しいパターンを格納していく。

この追加処理を図 4 と図 5 の例を用いて説明する。ゲノムデータを追加する前の初期状態として、データがすでに図 4 のように格納されているとする。この初期状態では、変異ディクショナリテーブル($T_V$)に N 変異が登録されている。図 3 のテーブルスキーマと比較すると、変異ディクショナリテーブル（$T_V$）に location 列（ゲノム型位置）が追加されている違いがある。この列は、ゲノム型を 2 ビット配列とみなした

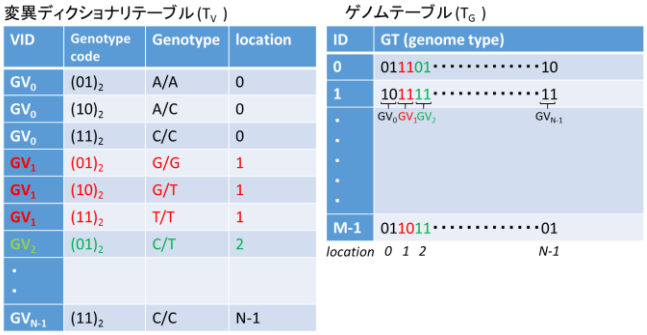


**図 4: 動的データ構造の初期状態**

時に何番目の位置に各変異のジェノタイプコードが格納されるかを表す（以降、ゲノム型位置と呼ぶ）。この例では、$GV_0$は 0 番目のゲノム型位置、$GV_{N-1}$は N-1 番目のゲノム型位置に格納されていることを表している。また、全変異がそれぞれ 3 パターンずつのジェノタイプを持っていて、各パターンにはジェノタイプコード$(01)_2$、$(10)_2$、$(11)_2$が割り付けられている。ゲノム型は、2 ビットジェノタイプコードの N 要素（$GV_0$～$GV_{N-1}$）の配列となっている。図 4 の例では、このゲノム型が M 個体分ゲノムテーブル($T_G$)の GT 列に格納されている。

次に、新しいジェノタイプパターンを追加する処理を図 5 で説明する。まず、変異$GV_1$に'A/G' というパターンを持つ個体 M（ゲノムテーブルの ID が M の個体）をゲノムテーブルに追加することを考える。変異$GV_1$には'G/G', 'G/T'、'T/T' の既存パターンしかないため、'A/G'は新規のパターンであり、既存の領域には格納できない。よって、個体 M のゲノム型配列は後ろに新たに 2 ビット領域を拡張する。そして、$GV_1$のために拡張した領域（N 番目のゲノム型位置）に$(01)_2$を割り当て、既存の$GV_1$の領域（1 番目のゲノム型位置）には、$GV_1$のパターン情報が他のゲノム型位置に存在することを表す$(00)_2$を格納する。また、変異ディクショナリテーブル($T_V$)には、変異$GV_1$のパターン'A/G'用にパターンコード$(01)_2$を割り付け、N 番目のゲノム型位置にその情報を格納することを表す新しいエントリを追加する。次に、図 5 を参照しながら、個体 M に続いてさらに個体 M+1～M+4 を追加する処理を説明する。

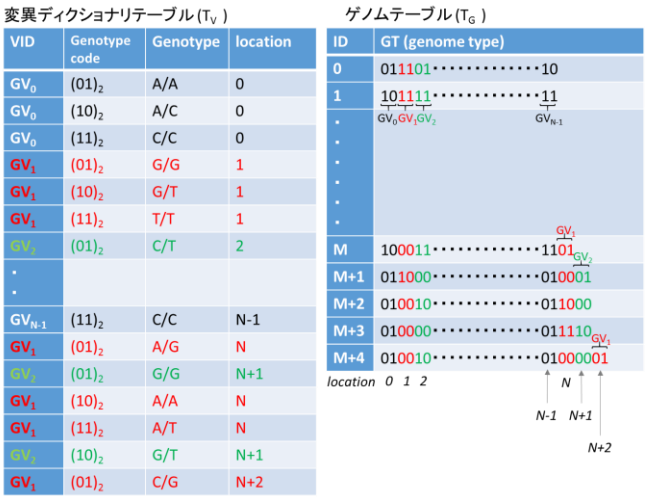


**図 5: 動的データ構造へジェノタイプパターン追加**

i. 個体 M+1 に、$GV_2$の 4 番目の新しいパターン'G/G'があると、$GV_1$の 2 番目の領域（N 番目の 2 ビット領域）の直後（N+1 番目の 2 ビット領域）に領域を追加し、$(01)_2$を格納する。$GV_2$の最初の領域には$(00)_2$を格納する。

ii. 個体 M+2 に、$GV_1$の 5 番目の新しいパターン'A/A'があると、$GV_1$の既存の 2 番目の領域（N 番目の 2 ビット領域）に'A/A'のための新しいパターンコード$(01)_2$を格納して GV1 の最初の領域に$(00)_2$を格納する。

iii. 個体 M+3 に、$GV_1$の 6 番目のパターン'A/T'と$GV_2$の 5 番目のパターンがあると、それぞれの 2 番目に追加された領域（N 番目の 2 ビット領域、N+1 番目の 2 ビット領域）にそれぞれ$(11)_2$と$(10)_2$を格納し、$GV_1$と$GV_2$の最初の領域には$(00)_2$を格納する。

iv. 個体 M+4 に、$GV_1$の 7 番目のパターン'C/G'があると、$GV_1$の 3 番目の領域が必要となり、N+2 番目の 2 ビット領域に追加して、$(01)_2$を格納する。$GV_1$のその他の領域（0 番目と N 番目の 2 ビット領域）には$(00)_2$を格納する。

また、新規パターンが出現する度に、変異ディクショナリテーブルにも対応するエントリを追加していく。このように、変異情報を格納するゲノム型上の位置は、個体を追加して新しいパターンが出現して新しい格納領域が必要になった時に決定し、必要な領域を追加する。

この動的データ構造では、一つの変異情報が様々な場所に分散して格納されているため、各変異の情報一つ一つにアクセスする処理があまり効率的に行えないように見える。しかしながら、固定長要素の配列になっているので、各配列要素に対する逐次処理は効率的に実行可能である。よって、変異毎のジェノタイプ集計の処理は、まず各固定長要素に対して集計処理を行い、その集計結果を使って変異毎の集計結果を求める、2 ステップで行うと高速に処理が可能である。具体的には以下の 2 ステップで行う。

i. 固定長要素配列の各 2 ビット要素に格納している 2 ビットコードの集計を実行して、固定長要素配列ベースの集計値を得る。固定長の各要素に対する集計なので、この処理は高速に実行可能である。

ii. 変異ディクショナリテーブルのマッピング情報から、i で得た固定長要素配列ベースの集計値を変換して、変異ベースの集計値を得る。図 6 は、図 5 の変異ディクショナリテーブルを前提にして、変異 $GV_1$ の変異ベース集計値への変換を例示したものである。変異 $GV_1$ のジェノタイプ'G/G', 'G/T', 'T/T'は 2 ビット固定長配列の 1 番目要素に対応し、'A/G', 'A/A', 'A/T'は N 番目、'C/G'は N+2 番目に対応しているので、その対応に従うと、2 ビット固定長配列ベースの集計値から変異 $GV_1$ の変異ベースの集計値へ変換することができる。同様にすれば、他の変異についても変異ベースの集計値を得ることができる。

このように、各変異情報が分散して格納されていても、全変異の集計処理を高速に実行することができる。

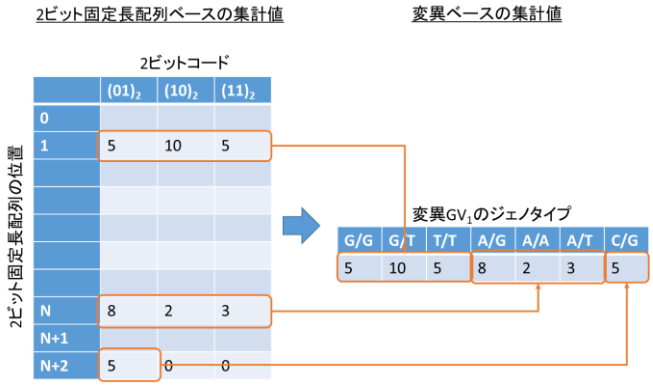


**図 6: 変異ベースの集計値への変換**

### 3.2.2 集計性能評価

このゲノム型の格納効率と集計処理性能を評価するために、例として、表 2 のような変異分布のデータを作成して評価を行った。このデータの 100000 変異のうち、90000 変異は 3 パターンのジェノタイプを持ち、これは 2 アレルの組み合わせから構成する SNP を想定している。9900 変異は 6 パターンで、3 アレルの組み合わせから構成する SNP の想定である。100 変異は 55 パターンで STR を想定している。表 3 は、この可変パターンデータ（可変）と、全 100000 変異が 3 パターン固定のデータ（固定）とで格納サイズと集計処理実行時間を計測した結果である。この結果から、パターン可変の場合でも、集計処理時間はパターン固定の場合から 20%の増加ですんでいることがわかる。また、この処理時間増加割合は、サイズの増加割合とほぼ一致し、処理時間は処理量が多くなった分だけ遅くなったことがわかる。このことから、我々が提案する動的データ構造は、各変異情報が分散しているにも関わらず、効率的に可変パターンデータを扱っていることがわかる。

**表 2: 評価データのジェノタイプパターン数分布**

| サンプル数 | ジェノタイプのパターン数 | 変異タイプ |
|---|---|---|
| 90000 | 3 | 2 アレル SNP |
| 9900 | 6 | 3 アレル SNP |
| 100 | 55 | STR |

**表 3: データサイズと集計処理の実行時間**

| | 固定 | 可変 |
|---|---|---|
| 実行時間(sec) | 8.331 | 9.958 |
| サイズ(GB) | 2.70 | 3.32 |

## 4. 並列処理による高速化

### 4.1 クエリ並列化

ゲノムデータはますます大規模になっていくため、集計処理の高速化は重要な課題である。近年のマシンコア数向上により、処理の並列化は高速化の非常に有効な手段であり、ゲノム型がベースにしている PostgreSQL は最新版の 9.6 からクエリを並列化が可能になっている[2]。しかしながら、このクエリ並列機能を試して SQL1 を実行したところ、並列化による性能向上がほとんどなかった。

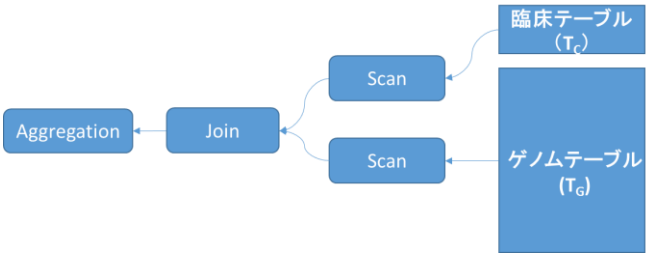


**図 7: SQL1 のクエリプラン**

その理由を説明するために、まず PostgreSQL 上で SQL 1 を逐次実行した場合の実行プランを図 7 に図示する。SQL 1 では、まずゲノムテーブルと臨床テーブルを読み込み(Scan)、両テーブルの結合(Join)を行い、最後にそのデータを集約(Aggregation)してクエリ結果を出力する。この実行プランの実行時間の中で、最も支配的な処理が Aggregation である。ゲノム型の集約処理では、非常に多くの変異に対して集計処理を行うので、その集計処理を内部で行う Aggregation が最も計算が重い処理になっている。よって、Aggregation 処理を並列化して並列プロセスに仕事を分散させることが並列化による高速化のために重要である。ところが、図 9 の上図の PostgreSQL9.6 上での SQL1 並列クエリ実行プランを見ると、Scan 処理と Join 処理は並列プ

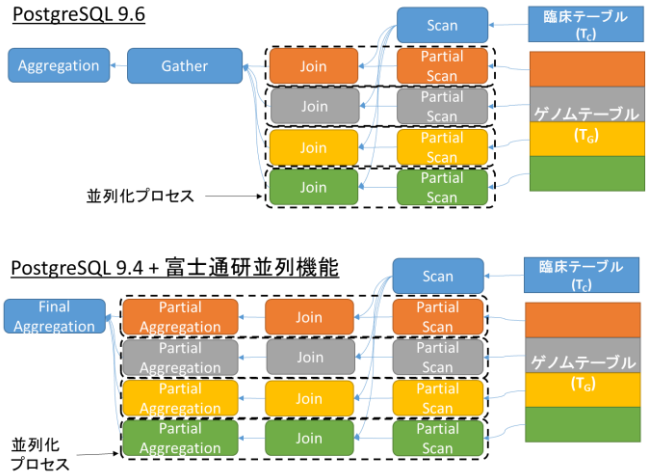


図 8: 並列クエリプラン

ロセス毎に実行されているが、Aggregation 処理は 1 つのプロセスで実行されている。つまり、バージョン 9.6 のクエリ並列化では、最も支配的な実行時間である Aggregation 処理が並列化されていないため、並列化の効果がないことがわかる。そこで我々は以前我々が開発した PostgreSQL9.4 ベースの独自並列機能[7]を導入して評価を行うことにした。この機能では、並列化可能なプランのなかで最もプランツリーの根に近いプランを起点にして並列化する。図 9 の下図のように、Aggregation 処理の並列化では、各並列プロセスで部分集約（Partial Aggregation）を行い、最後に 1 プロセスが各並列プロセスの部分集約結果を最終集約（Final Aggregation）する。最終集約では各プロセスで集約した結果をマージするだけで実際の集約処理は各プロセスの部分集約で行われるので、これが並列化することで、クエリ性能が高速化する。

クエリ並列処理によりクエリがどれだけ高速化するかは 4.3 節で説明する。

## 4.2 SIMD 処理

また、プロセッサに備わっている SIMD 命令を利用することでプロセッサ命令あたりの並列化が可能である。Intel Xeon processor では、SandyBridge 世代から備わっている AVX2 機能を利用して 256bit 幅のベクトル演算を実行することができる

集計処理は、主に次の 2 ステップで実行する。まずゲノム型の各変異に格納されているジェノタイプパターン値を取得し、次に、各変異の各パターンの集計値を管理する集計配列の対応する要素をカウントアップする。我々はこの処理をベクトル化するために、ゲノム型に格納されているパターン値のベクトルに対応するカウントアップするべき値をベクトル化したカウントアップベクトルを、あらかじめルックアップ配列として用意して、このルックアップ配列を利用してベクトル処理を実行する。ルックアップ配列を利用した処理の例を図 9 に示す。まず、ゲノム型から 4 個の連続した変異パターン値を取得する。図の例では、このベクトル値は $GV_n$～$GV_{n+3}$ のパターン値のベクトル値 $(00010110)_2$ である。ルックアップ配列は、このベクトル値の全パターン（256 パターン）に対応するカウントアップベクトル値があらかじめ格納している。よって、ベクトル値$(00010110)_2$に対応するカウントアップベクトルは、ルックアップ配列の 22 番目（パターン値の 10 進表現）の値を参照すると取得することができる。カウントアップベクトルの値は 、集計配列のカウントアップすべき箇所が 1 になっており、他の箇所は 0 になっている（図では、$GV_n$の$(00)_2$、$GV_{n+1}$の$(01)_2$、$GV_{n+2}$の$(01)_2$、$GV_{n+3}$の$(10)_2$の箇所が 1 となる）ので、このカウントアップベクトル値を、集計配列に対して SIMD 加算すると、1 個体の 4 変異分のカウントアップ処理が完了する。このカウントアップ処理を全個体の全変異に対して行うと集計処理が完了する。

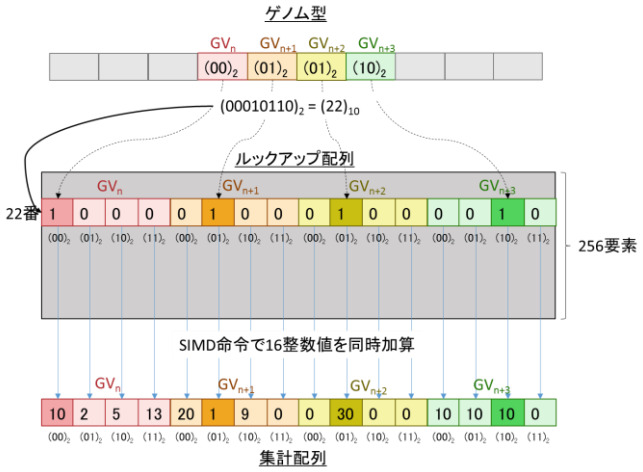


図 9: ルックアップ配列と SIMD 処理

ところで、集計配列は加算ベクトルと同一構造にする必要があるために 1 要素が 16bit 幅になっているので、個体数が 65535 を超えると加算値が溢れる問題がある。そこで、全個体の集計値を格納するのに十分なサイズを持つ最終集計配列を用意して、65535 回集計する毎に、集計配列の値に最終集計配列に反映して集計テーブルをクリアし、集計処理を続ける。これを繰り返して最終結果をえる。

SIMD 処理を導入することでどれだけクエリが高速化するかは 4.3 節で説明する。

## 4.3 集計性能評価

本節では並列処理によって集計処理がどれだけ高速化するか評価する。3.1.2 節の評価と同じ評価環境、データ、クエリを利用して、PostgreSQL9.6 と PostgreSQL9.4（＋富士通研並列機能）の集計処理実行

時間を計測し、比較した。その結果を図 10 に示す。まず、CPU を 1 コアだけ利用する逐次処理では、最新版である PostgreSQL 9.6 が 7.3 秒で、バージョン 9.4 の 8.3 秒よりも高速であった。PostgreSQL 本体のバージョンアップにより足回りが改善されたことを表している。しかし、複数コアを利用してクエリを並列化実行すると、4.1 節で述べたようにバージョン 9.6 の並列化では最も重い処理である Aggregation 処理が並列化できないため、利用コア数を増やすとバージョン 9.4（＋富士通研並列機能）上の処理のほうが高速になる。この並列機能を利用してクエリ処理時間を計測すると、1 コアで約 8.3 秒の処理が、CPU を 8 コア使うことで約 2.1 秒まで処理時間が短縮し、並列化効果で約 4 倍高速化することがわかった。また、SIMD 命令を利用することで、逐次処理も並列処理も約 2 割程度性能が向上して、最終的に 1.7 秒程度まで処理時間を削減できた。SIMD 処理で、論理的に 4 変異同時に集計を実行しているにもかかわらず性能向上率が 2 割程度にとどまった理由としては、SIMD レジスタのロード・ストアのオーバーヘッドや集計処理以外の処理時間が予想以上に大きかったことが挙げられる。

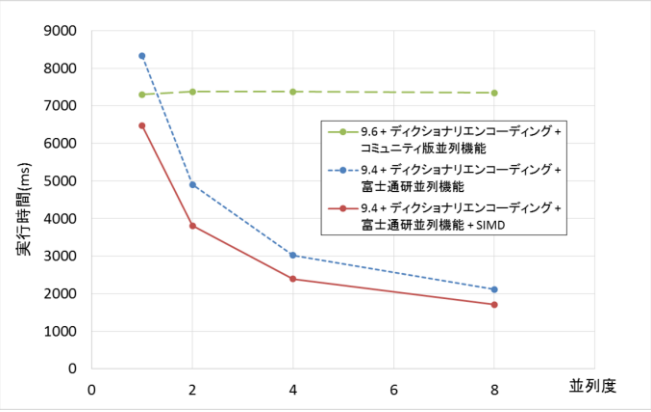


**図 10: 集計処理の実行時間**

## 5. まとめ

本稿では、実際の大規模で多様なゲノムデータをデータベース上で効率的かつ高速に集計処理するために、前論文で提案したゲノム型に動的データ構造の導入、クエリ並列処理、SIMD 命令の適用により、改良を行った。その結果、まず 3.1.1 節で述べたように、ゲノム型データ構造をディレクトリエンコーディング形式にすることによって、約 5 倍の性能向上があった。また、4.3 節で述べたように、クエリ並列化と SIMD 処理を適用することで、8 コア利用時に約 5 倍高速化した。これらをあわせると結果として約 25 倍高速化し、10 万変異×10 万人データが 2 秒以内で集計が完了するようになった。近年得られるゲノムデータでは、変異数が数千万か所に及ぶ場合もあるが、その場合でも数百秒で集計処理が完了すると予想している。疾患ゲノム関連解析研究では、目的変数を変更したり、研究対象データから低品質データをフィルタリングしたり、統計的なバイアスを取り除いたりするために、様々な条件でデータをフィルタリングして集計する処理を、トライ＆エラーしながら何回もまわすことが想定される。本稿の技術により、解析処理を繰り返して行えるようになり、ゲノム疾患関連研究の促進に寄与することで、近年注目が高まっているゲノム医療の発展に貢献できることを期待している。

## 参 考 文 献


[1] Y. Ujibashi, M. Kawaba, L. Harada “Proposal of Database Type and Aggregation Function for Accelerating Medical Genomics Study on DBMS” EDBT 2016 pp. 672-673 2015.

[2] The PostgreSQL Global Development Group. “PostgreSQL” http://www.postgresql.org/ 1996-2016.

[3] C. C Chang, C.C Chow, L.CAM Tellier, S. Vattikuti, S. Purcell, J. J Lee “Second-generation PLINK: rising to the challenge of larger and richer datasets.” GigaScience, 4 2015.

[4] “Variant Call Format” http://www.internationalgenome.org/wiki/Analysis/variant-call-format/ 2016

[5] A. Ameur, I. Bunkikis, S. Enroth, et al. “CanvasDB: a local database infrastructure for analysis of targeted- and whole-genome resequencing projects.” Database, Article ID bau098, Vol. 2014.

[6] U. Paila, B. A. Chapman, R. Kirchner “GEMINI: integrative exploration of genetic variation and genome annotations” PLOS Comput. Biol., 9, 1003153 2013.

[7] Y. Ujibashi, M. Nakamura, T. Tabaru, T. Hashida, M. Kawaba, L. Harada “Design of a Shared Memory mechanism for efficient parallel processing in PostgreSQL” IISA 2015 pp. 1-6 2015.