Vol.2018-BIO-53 No.7
2018/3/10

配列相同性検索ツールGHOSTZ-GPUのMPI環境における高速化

後藤 公太[1,2,a)]　大上 雅史[1]　石田 貴士[1,2]　秋山 泰[1,2,b)]

概要：GHOSTZ-GPU はメタゲノム解析において、配列相同性検索を高速に行う事のできるソフトウェアである。しかし、シーケンサが DNA 配列を読み取る速度は配列相同性検索よりも高速であり，効率的な解析のためには大規模な計算環境が求められている。本研究では先行研究の GHOSTZ-MP で挙げられた問題点とユーザビリティを改善する提案と実装を行い，評価実験を行った．評価実験の結果，64 ノードの大規模並列環境においては提案手法は先行研究と比較して 1.27 倍の高速化率を達成した．

キーワード：配列相同性検索，GHOSTZ-GPU，MPI，分散処理，タスクスケジューリング

Acceleration of Sequence Homology Search Tool GHOSTZ-GPU on MPI environment

Kota Goto[1,2,a)]　Masahito Ohue[1]　Takashi Ishida[1,2]　Yutaka Akiyama[1,2,b)]

Abstract: GHOSTZ-GPU is a fast sequence homology search tool for metagenome analysis. However, futher large-scale parallelized analysis system is required becauce the amount of DNA sequence data obtained from a sequencer is bigger than that processed by homology search tool. In this study, we solved problems in previous study and evaluated previous and proposed method. Proposed method is 1.27 times faster than previous method on 64 node environment.

Keywords: Sequence homology search, GHOSTZ-GPU, MPI, Distributed processing, Task scheduling

1. 導入

土壌・海洋などの自然環境や，生物の腸内・口腔内等に生息する様々な微生物，すなわち微生物叢の解析は，疾患の原因特定や個人ごとの罹患のしやすさなど有用な情報が得られるとして注目されている [1,2]. 環境中から取得した微生物叢のサンプルにどのような微生物が存在しているか網羅的にを調査するためには，微生物の持つゲノム情報をシーケンシングによって DNA 配列を読み取り，BLAST [3]・RAPSearch2 [4]・DIAMOND [5]・GHOSTZ-GPU [6] などによる配列相同性検索が行われる [7]. シーケンサが 1 日に読み取る塩基配列は最大で 1.5×10^{12} 塩基程度[*1]だが，相同性検索は GHOSTZ-GPU を用いた場合，1 日あたり 2.0×10^{9} 塩基程度である[*2]．シーケンサーの読み取り能力はその後の配列相同性検索の約 750 倍であり，配列相同性検索は解析の中で大きなボトルネックとなるため，より大規模な並列処理が必要である．GHOSTZ-GPU のノード間並列実装として GHOSTZ-MP [11] が挙げられる．GHOSTZ-MP では 64 ノード，128 ノードで動作させたと

[1] 東京工業大学 情報理工学院 情報工学系，
Department of Computer Science, School of Computing, Tokyo Institute of Technology
[2] 東京工業大学 情報生命博士教育院，
Education Academy of Computational Life Sciences, Tokyo Institute of Technology
[a)] goto@bi.c.titech.ac.jp
[b)] akiyama@c.titech.ac.jp

[*1] illumina, NovaSeq6000 [8]
[*2] TSUBAME 3.0 にて計測（28 cores, 4 GPUs）．データベース: NCBI nr（74 GB）[9], クエリ: ERR315856 [10] よりランダムに選んだ 1,000 万本（2.6 GB）より算出.

きの効率低下のほか，ユーザーによる前処理が必要である点など改善すべき点がある．そこで本研究では，先行研究で性能低下の原因とされている部分の改良とユーザビリティの向上を目的とし，MPI 環境で動作する GHOSTZ-GPU の計算手法を提案し，先行研究との評価実験を行った．

2. 先行研究

2.1 概要

GHOSTZ-MP は藤井らによって開発されたソフトウェアであり，GHOSTZ-GPU を MPIDP [12] と称する，計算機クラスタなどの MPI 環境上で動作する分散処理のフレームワークを用いて並列化している．MPIDP は同じく配列相同性検索ソフトウェアである GHOST-MP [13] にも用いられており，データ並列の性質をもつ生命情報解析ソフトウェアに適したフレームワークである．

2.2 計算の流れ

GHOSTZ-MP での計算の流れは 3 つのステップにわかれる．概要図を図 **1** に示す．

(1) **クエリの前処理・ジョブファイル作成**
MPIDP は分散処理を行う内容を UNIX コマンドの列として記述したジョブファイルを必要とする．GHOSTZ-MP では，ワーカープロセスと同じ数だけにクエリファイルを分割して 1 ファイルを 1 タスクとしている．

(2) **データベースファイルの配置**
データベースファイルがシステム全体で共有するファイルシステム上にある場合には，各ワーカープロセスからのアクセスが集中して性能が低下するなどの問題が発生する．計算のはじめに各ノードが持つローカル SSD へマスターノードが転送を行うことで，この問題を回避している．

(3) **MPIDP によるタスク実行**
必要な資源などの準備ができたら，MPIDP によるタスク実行が行われる．タスクはワーカープロセスと同数存在しているため，1 プロセスに 1 タスクを固定で割り当てて計算を行う．

2.3 問題点

GHOSTZ-MP ではいくつかの問題点が挙げられている．

(1) **ユーザーによる前処理が必要**
クエリファイルの分割時には各ファイルが同じ数のリード本数となるように分割される．この処理はファイルの全体を読み込む必要があり，並列処理されていないため，大規模並列時にはボトルネックとなる．

(2) **タスクサイズが減少する場合の性能低下**
クエリファイルが一定サイズより大きい場合には，シングルプロセスの GHOSTZ-GPU がもつデータベースファイルの先読み機能により，ファイル IO を隠蔽できるが，プロセス数が多い場合には相対的にタスクのサイズが小さくなり，問題となる．

(3) **計算の開始時に固定のスケジューリング**
1 ワーカープロセスに対して 1 タスクを固定で割り当てているため，他のタスクの終了を待つ時間などのオーバーヘッドが発生している．藤井らによると，ワーカープロセスの数に対するタスクの数について実験されているが，タスクの数はワーカープロセスの数と同じである場合が最も効率が良かったとしている．

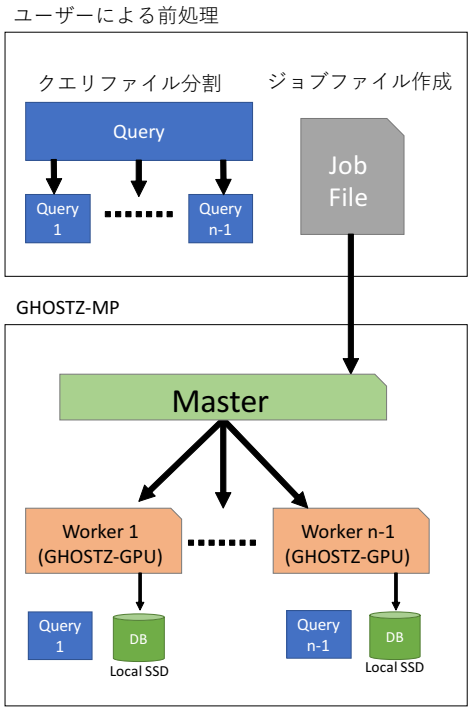

図 **1** GHOSTZ-MP 計算の流れ
n プロセスの場合．

3. 提案手法

3.1 計算の流れ

提案手法では計算全体は 3 つのフェーズに分かれる．各フェーズは 1 回だけ行われ，Report Phase の終了後にソフトウェアは終了する．

(1) Init Phase
計算に必要な資源の配置，スケジューリングの準備等を行う．

(2) Search Phase
GHOSTZ-GPU の検索処理に相当する計算を．マスターワーカー方式によるスケジューリングを用いて行う．

(3) Report Phase
各プロセスで分散処理された結果の集約とファイルへの出力を行う．

3.1.1 Init Phase

Init Phase では計算に必要な資源の配置，事前情報の共

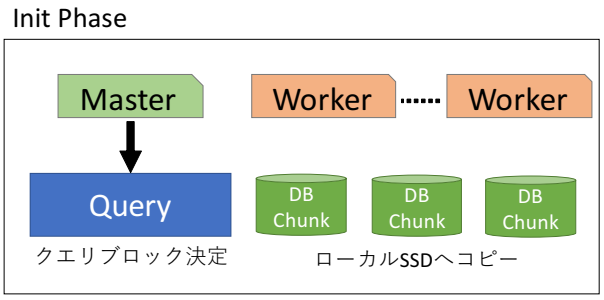

図 2　Init Phase 処理の概要図

有を行う．図 2 に Init Phase での処理の概要図を示した．

- マスタープロセス
  マスタープロセスはスケジューリングの準備のためにクエリファイルを読み込み, 全体でいくつのクエリブロックが存在するかを数える．GHOSTZ-GPU ではクエリブロックは前のブロックの検索が終了するごとに逐次読み込まれていたが，全体のタスク数を決めスケジューリングを行うために，事前にブロック数の決定を行う．クエリ分割の方法については 3.2.2 節で説明する．
- ワーカープロセス
  ワーカープロセスはデータベースファイルの SSD へのコピーを行う．先行研究である GHOSTZ-MP ではマスタープロセスが代表して読み込み，各ワーカープロセスへ全体のコピーを転送していたが，提案手法ではワーカープロセスがチャンクごとに分担してローカル SSD への転送を行う．各ノードが持つローカル SSD を仮想的に結合させて専用のスクラッチディスクとしてアクセスできる，BeeGFS [14] を用いた．ローカル SSD への転送終了後は，ラウンドロビンに次の Search Phase で最初に担当するデータベースチャンクを割り当ててロードを行う．BeeGFS などのスクラッチ領域が提供されていない計算機システムを用いる場合には，事前配置処理をコンパイル時に無効化することができる．無効化時には，チャンク毎に 1 プロセスが代表して読み込みを行い，そのデータベースを必要とするプロセスに転送を行うことで，ファイルシステムへのアクセス集中を防ぐ．

3.1.2　Search Phase

Search Phase では GHOSTZ-GPU の主となる検索処理を行う．**図 3** に Search Phase での処理の概要図を示した．

- マスタープロセス
  マスタープロセスはスケジューラとして働き，ワーカープロセスからのタスク要求に対して適切なタスクを決定し割り当てる．スケジューリングの方法は 3.2.3 節で説明する．
- ワーカープロセス
  ワーカープロセスはマスタープロセスより割り当てら

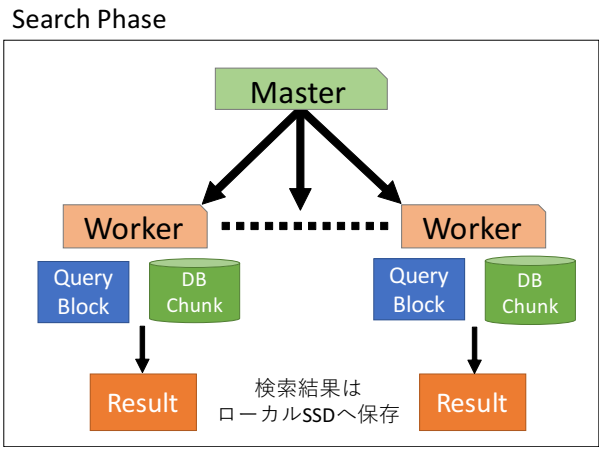

図 3　Search Phase 処理の概要図

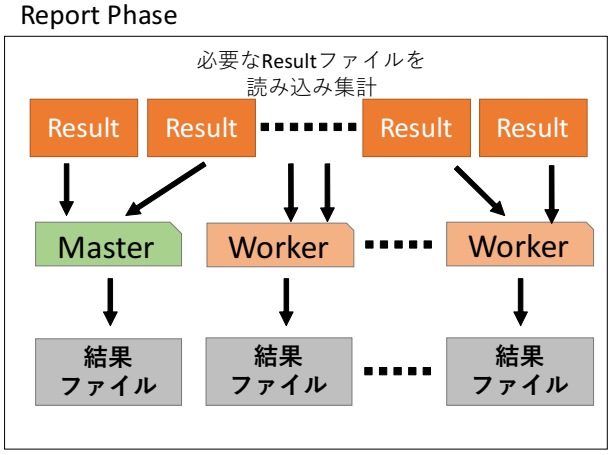

図 4　Report Phase 処理の概要図

れた GHOSTZ-GPU のタスクを行う．割り当てられたタスクの実行に必要なクエリファイルは各ワーカープロセスがそれぞれファイルシステムから読み込み，データベースファイルは Init Phase で配置されたローカル SSD 上のファイルを各ワーカープロセスが読み込む．1 つのタスクに対する検索結果はローカル SSD へ一時的に保存され，次のタスクの要求をマスタープロセスに行う．割り当てられるタスクがなかった場合には，全プロセスのタスク終了を待ち Report Phase に移行する．

3.1.3　Report Phase

Report Phase ではマスタープロセスとワーカープロセスは同一の処理を行う．**図 4** に Report Phase での処理の概要図を示した．Search Phase での計算結果はすべてローカル SSD に保存され，BeeGFS によって全プロセスがアクセスすることができる．GHOSTZ-GPU では 1 本のリードにつき最大で上位何件までの結果を出力するかを，オプションによってユーザーが決定する（デフォルト値は 10）．提案手法においても同一の結果を出力することを保証するために，各タスクごとに最大の出力件数までを保存して

いる．同じクエリブロックのタスクの結果を集め，上位の結果のみを出力することで GHOSTZ-GPU と同一の結果を出力している．全プロセスにラウンドロビンにクエリブロックを割り当てて集計処理を行う．出力ファイルはクエリブロックの個数に分かれたファイルとなり，cat コマンドなどで結合すると GHOSTZ-GPU と同一の出力となる．

3.2 提案手法における改善方針

3.2.1 タスク粒度の変更

シングルプロセスの GHOSTZ-GPU ではクエリファイルの処理単位として “クエリブロック”，データベースの処理単位として “データベースチャンク” が存在する．先行研究の GHOSTZ-MP ではこの内部のクエリブロックの単位とは無関係に，リード本数によってタスクを分割していた．GHOSTZ-GPU では 1 つのクエリブロックを処理するためにデータベース全体を 2 度読み込み，検索を行ってるが，提案手法では 1 つのクエリブロックと 1 つのデータベースチャンクに対する処理をタスクの単位とした．

連続して処理するタスクが，同じデータベースチャンクに対するタスクであった場合にデータベースを再利用しファイル IO を削減することができる．また，先行研究で動的なスケジューリングが用いられなかった理由として挙げられている，クエリサイズの減少による効率低下とファイル IO 律速の問題を防げると考えられる．

3.2.2 クエリ分割方法の変更

GHOSTZ-GPU ではクエリブロックのサイズは各リードの塩基数によって決定し，ブロックごとの計算時間や使用メモリ量などを揃えている．GHOSTZ-MP ではクエリファイル全体をリードの本数に基いてワーカープロセスと同じ数に分割したファイルをワーカープロセスへの入力ファイルとして扱い，各プロセスは GHOSTZ-GPU と同じく塩基数によってクエリを分割している．提案手法では，事前にクエリブロックが何ブロック存在するかを計算の初めに決定する必要がある．塩基数・リード本数によってクエリを分割する場合にはクエリファイル全体の読み込みと処理を行う．この処理は Init Phase においてボトルネックとなるため，分割方法の変更を行った．

提案する方法では，クエリファイルのサイズによってクエリブロックを決定する．FASTA ファイルは複数行にまたがって 1 本のリードを構成している．オプションで与えられたクエリブロックのサイズをそのまま適用して分割を行った場合には，境界上に存在しているリードは正しく読み込めなくなる．そのため，次のようにクエリブロックの境界を決定した．

- オプションに与えられたファイルサイズごとの位置をクエリブロックの境界とする．
- その位置がリードの途中であった場合，該当するリードの最後までを前のブロックに含める．

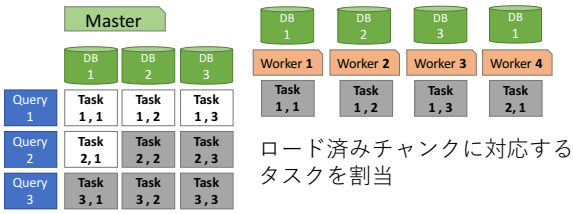

図 5　スケジューリングの概要図 1

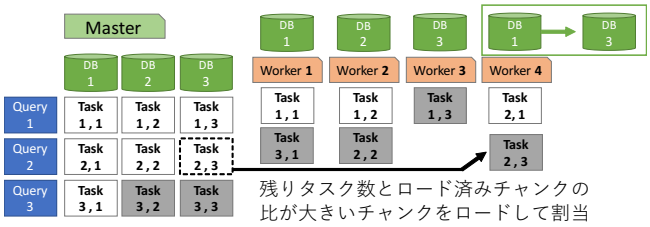

図 6　スケジューリングの概要図 2

境界付近のデータと全体のファイルサイズのみからクエリブロックの数と境界を決定できるため，クエリファイルの読み込みを削減して Init Phase における処理の高速化ができる．

3.2.3 スケジューリング方法の変更

提案手法では動的なスケジューリングを行う．各タスクを実行するためには 1 つのクエリブロックと 1 つのデータベースチャンクが必要である．クエリブロックのサイズはデフォルトのパラメータを用いた場合は 128 MB 分のクエリファイルに相当し，データベースチャンクのサイズはおよそ 4 GB である．そこで，データベースチャンクの再利用をすることでファイル IO を削減できるスケジューリング方法を用いる．

図 5 に初期段階でのスケジューリングの概要図を示した．（ワーカープロセス数 4・データベースチャンク 3・クエリブロック 3 の場合）ワーカープロセスは Init Phase で 1 つのデータベースチャンクをロードしている．図 5 中では，チャンク 1 を Worker 1 と Worker 4 が，チャンク 2 を Worker 2 が，チャンク 3 を Worker 3 がロードしている．このときに，マスタープロセスはデータベースの再読込が不要となるように，ロード済みのデータベースチャンクが必要なタスクを割り当てる．ワーカープロセスはクエリファイルだけを読み込むため，タスクの開始までの時間を削減できる．図 5 の例では，チャンク 1 をロードしているワーカープロセスは Worker 1 と Worker 4 の 2 つ存在するため，チャンク 1 を必要とするタスクは他のタスクに比べて早く終了する．あるチャンクを必要とするタスクがなくなった場合，そのチャンクをロードしているワーカープロセスは別のチャンクをロードし直す．

図 6 にスケジューリングが進んだ場合の処理の概要図を示した．ワーカープロセスはその時ロードしているチャンクの対応するタスクのみが割り当てられるため，残りのタ

スク数とそれを処理するプロセス数の比が最も大きいチャンクをロードすることで，処理能力が不足しているチャンクへの転換を行う．図 6 の例ではチャンク 1 のタスクがなくなったため，Worker 4 は他のデータベースチャンクへ転換する．データベースチャンク 3 は，残りのタスク数が 2 タスク，ロードしているプロセス数は Worker 2 の 1 プロセスのみである．残りのタスク数とプロセス数の比が最大であり，処理に時間がかかることが予想されるため，Worker 4 はチャンク 3 へ転換する．シングルプロセスの GHOSTZ-GPU ではクエリブロックを処理する毎にデータベース全体を読み込んでいたが，提案するスケジューリング方法ではデータベースチャンクを再利用することでファイル IO を削減し，他チャンクへの転換によって処理に時間のかかるチャンクがある場合にも適切にスケジューリングが行える．

3.2.4 ソフトウェアの利用方法の変更

先行研究の GHOSTZ-MP ではシングルプロセスの GHOSTZ-GPU と比較して，並列処理フレームワーク MPIDP 用のジョブファイルの作成など，ユーザーによる操作が必要な処理が追加されている．一方，提案手法ではユーザーによる事前準備は必要とせず，ソフトウェアに与えるパラメータも GHOSTZ-GPU のパラメータに 1 個を追加したのみである．**図 7** にシングルプロセスの GHOSTZ-GPU，先行研究，提案手法の実行時のコマンドの例を示した．先行研究ではクエリの前処理とジョブファイルの作成がコマンド実行前に必要である．提案手法で新たに追加したパラメータは，Init Phase でデータベースを事前に配置する際に利用するスクラッチ領域を指定するパラメータであり，本研究では BeeGFS によって作成された領域を与えている．このパラメータは利用する計算機システムによって提供される方法が違うことが想定されるために，実行時のパラメータとした．出力結果は GHOSTZ-MP と提案手法の両方でいくつかのファイルに分かれているが，cat コマンドで結合することで元の GHOSTZ-GPU と同じ出力となる．提案手法ではユーザーに要する操作を削減したことで，解析パイプラインとしてのユーザビリティが向上したと考えられる．

```
## シングルプロセス GHOSTZ-GPU 実行コマンド
ghostz-gpu aln -i ［クエリファイル］
 -d ［データベースファイル］ -o ［結果ファイル］

## 先行研究 GHOSTZ-MP 実行コマンド
mpirun -np ［プロセス数］ mpidp -rt ［リトライ回数］
-tb ［ジョブファイル］

## 提案手法実行コマンド
mpirun -np ［プロセス数］ ghostz-gpu aln
-i ［クエリファイル］ -d ［データベースファイル］
-o ［結果ファイル］ -m ［BeeGFS スクラッチ領域パス］
```

図 7 実行コマンドの例．CPU・GPU 並列数のオプションは除き，最低限必要なオプションを示している．

表 1 TSUBAME 3.0 ハードウェアスペック

名称	詳細
CPU	Intel Xeon E5–2680 v4 2.4 GHz 14 cores × 2
Memory	256 GB
GPU	Nvidia Tesla P100 NVLink (Memory: 16 GB) × 4
SSD	2 TB
ファイルシステム	DDN SFA14KXE, EXAScaler (Lustre)

表 2 ソフトウェアバージョン

名称	バージョン
GCC	4.8.5
CUDA	8.0
OpenMPI	2.1.2
Boost C++ Libraries	1.64.0
MPIDP	1.0.2
GHOSTZ-GPU	1.1.0

表 3 実験に用いたデータ

	名称	サイズ
クエリ ファイル	ERR315856 （海洋メタゲノム）	89,259,915 リード （23 GB）
データベース ファイル	NCBI nr [9] (2017/8/25 取得)	130,076,561 リード （74 GB）

4. 評価実験

4.1 実験環境

表 1 に実験に用いた TSUBAME 3.0 のハードウェアスペックを，**表 2** に使用したライブラリ等の詳細を示す．本研究では，先行研究と提案手法の評価のために 3 つの実験を行った．測定はすべて 3 回行い，中央値を示している．実験には**表 3** に示すデータベースとクエリファイルを用いた．

- 実験 1：GHOSTZ-GPU による実行時間の測定
  先行研究の GHOSTZ-MP と提案手法で高速化率を算出するために，1 プロセスの場合の速度としてシングルプロセスの GHOSTZ-GPU の実行時間を測定した．
- 実験 2：クエリ前処理 実行時間の測定
  GHOSTZ-MP で必要であるユーザによる前処理に要する時間を測定するために，クエリ分割の処理に要する時間を測定した．
- 実験 3：処理全体の実行時間の測定
  GHOSTZ-MP と提案手法について，検索全体の処理を測定して比較を行った．

4.2 実験 1：GHOSTZ-GPU による実行時間の測定

表 4 にクエリが 1,000 万本（2.6 GB）の場合のシング

表 4　GHOSTZ-GPU 実行時間

	SSD 利用	直接読み込み
データベース転送時間 [秒]	320	–
GHOSTZ-GPU 実行時間 [秒]	42,536	44,629

表 5　クエリの前処理に要する時間

リード本数 [万本]	1,000	2,000	4,000	8,000
実行時間 [秒]	347	670	1,317	2,649

ルプロセスの GHOSTZ-GPU による実行時間を示した．1,000 万本のクエリは表 3 で示したものからランダムに選択した．SSD を使わず直接共有ファイルシステムのデータベースファイルを参照する場合には，最初の転送時間がかからないため，‘–’ と表記した．SSD を利用する方法がおよそ 5%高速であった．以後の実験で高速化率を算出する場合には，この実験結果の SSD を利用する方法の測定時間を 1 倍として算出した．

4.3　実験 2：クエリ前処理 実行時間の測定

表 5 にクエリ分割に要した時間を示した．プログラムは Python スクリプトで記述されているため速度向上の余地はあるが，FASTA ファイルの処理には Python で提供されているライブラリなどが広く利用されているため，ユーザーからみたコストの指標としては妥当と考えられる．

4.4　実験 3：検索処理全体の実行時間の測定

表 6, **表 7** に先行研究である GHOSTZ-MP と提案手法のそれぞれの実行時間を示した．クエリ前処理の値は実験 2 の結果より，中央値を参考として載せている．提案手法との比較のために，マスタープロセスによるデータベースのコピーを Init Phase, その後の検索処理を Search Phase として示している．**図 8**, **表 8** に GHOSTZ-MP と提案手法の高速化率を示した．高速化率は実験 1 で測定したシングルプロセスの GHOSTZ-GPU の実行時間を基準として算出した．グラフ中の値は 3 回の測定の中央値を示しており，最小値と最大値をエラーバーで示している．高速化率は 3 回の測定結果の中央値を用いている．GHOSTZ-MP はノード数が増えると並列化効率が低下しているが，提案手法はノード数の増加に伴って並列化効率が上昇するため，64 ノードの場合においては提案手法の方が良い高速化率となった．

5.　考察

5.1　検索処理単独の効率

実験 3 ではクエリ分割の前処理を含めて，解析処理全体のワークフローを比較している．表 6 を見ると，8 プロセスの場合とくらべて 64 プロセスでは全体のワークフローのうち，クエリ分割の処理が占める割合が大きくなっていることがわかる．このことから，先行研究ではプロセス数

表 6　先行研究の実行時間（クエリ 1,000 万本）
クエリ前処理の時間には表 5 の値を参考として載せた．

各 Phase での 実行時間 [秒]	プロセス数 8	16	32	64
クエリ分割処理	347	347	347	347
Init Phase	299	302	287	300
Search Phase	6,510	3,478	2,064	847
Total	7,156	4,127	2,698	1,494

表 7　提案手法の実行時間（クエリ 1,000 万本）

各 Phase での 実行時間 [秒]	プロセス数 8	16	32	64
Init Phase	257	168	92	43
Search Phase	11,434	5,354	2,597	946
Report Phase	1,663	955	485	181
Total	13,354	6,477	3,174	1,170

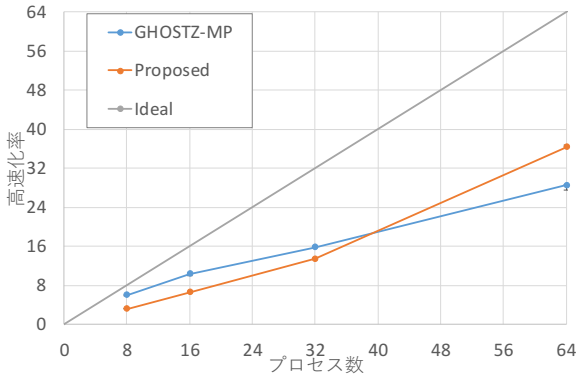

図 8　先行研究と提案手法の高速化率の比較（クエリ 1,000 万本）

表 8　先行研究と提案手法の高速化率 （クエリ 1,000 万本）

高速化率 [倍]	プロセス数 8	16	32	64
GHOSTZ-MP	5.94	10.31	15.77	28.51
提案手法	3.19	6.57	13.43	36.32

が増えるに従って効率が低下していると考えられる．**図 9**, **図 10** にクエリ本数が 1,000 万本と 8,000 万本の場合の検索処理（Search Phase）のみを見た場合の高速化率を示した．提案手法はどちらのクエリのサイズでも，64 プロセスにおいて高速化率が約 45 倍，並列化効率では 70%程度であり，クエリが増加した場合わずかに性能が向上している．また，GHOSTZ-MP では常に提案手法よりも高い効率であり，クエリが 8,000 万本の場合には高速化率は 57 倍，並列化効率では約 89%の効率となった．これは GHOSTZ-MP は始めにタスクを割り当てた後，追加のタスク割り当てがなくスケジューリングのオーバーヘッドが発生しないことと，データベースを BeeGFS による高速なスクラッチ領域に配置しており，Lustre ファイルシステムよりも高速にアクセスできるためと考えられる．藤井らによる TSUBAME2.5

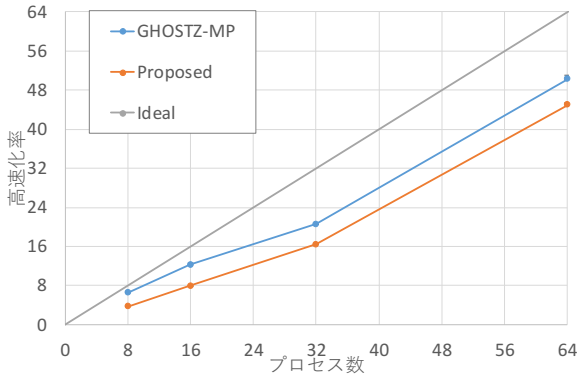

図 9 検索処理のみの並列化効率（クエリ 1,000 万本）

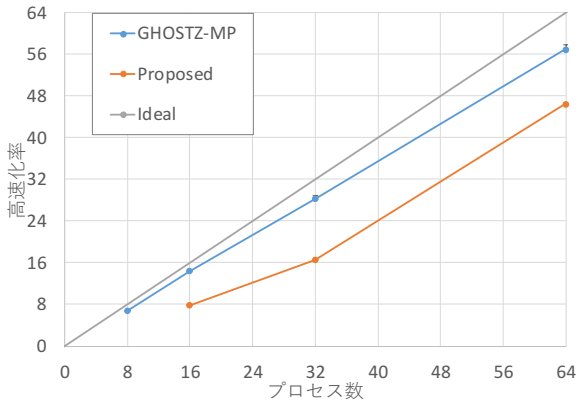

図 10 検索処理のみの並列化効率（クエリ 8,000 万本）

における実験では 64 プロセスでは約 70%程度の効率と述べられているが，これは実験に用いるデータのサイズが，ユースケースとして考えられる大規模な解析を評価するには小さかったためと考えられる．

このことから，タスクの粒度の決定方法・スケジューリングの手法は先行研究である GHOSTZ-MP による手法が高い効率を得られると考えられる．

5.2 高速化率の変化

5.1 節で述べたようにスケジューリング手法については GHOSTZ-MP による手法が高い効率を得ることができる．しかし，実験 3 の結果では 64 プロセスの場合には高速化率が逆転し，提案手法のほうが良い結果となった．**図 11**, **図 12** にクエリが 1,000 万本の場合の GHOSTZ-MP と提案手法の各フェーズの内訳のグラフを示す．GHOSTZ-MP ではクエリの前処理と Init Phase の時間はプロセス数に関わらず固定となる．一方で提案手法では全てのフェーズはプロセス数が増加した場合に処理時間が減少するため，64 プロセス以上では処理時間の逆転が発生したと考えられる．クエリファイルサイズが増加すると Init Phase の割合は相対的に小さくなるが，クエリ前処理に必要な時間は

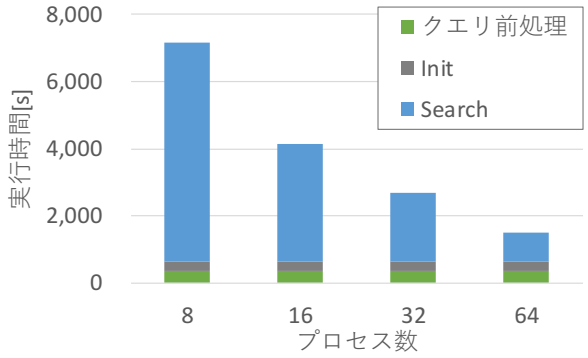

図 11 先行研究の各フェーズの内訳（クエリ 1,000 万本）

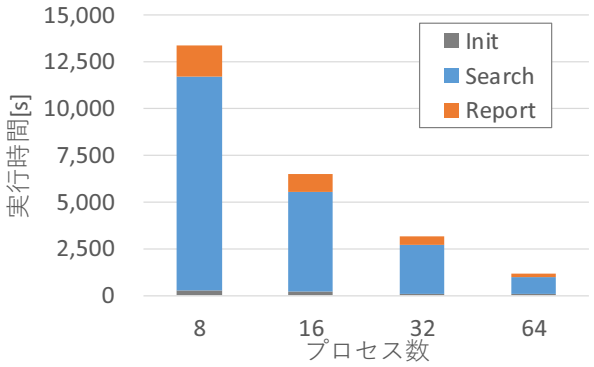

図 12 提案手法の各フェーズの内訳（クエリ 1,000 万本）

クエリサイズに比例するため，割合は大きく変化しない．提案手法では Init Phase がデータベースの転送が律速である場合，全てのフェーズがプロセス数増加によって処理時間が減少するために，64 プロセスなどの大規模並列環境で GHOSTZ-MP よりも高い並列化効率となったと考えられる．

5.3 実装の提案

提案手法では先行研究である GHOSTZ-MP の検索処理以外に行っている処理について，処理そのものを不要にする改良や並列処理へ変更を行うことで，64 プロセスを用いた大規模並列環境においては先行研究よりも高い高速化率を達成した．最も高速化率に影響のあった点は, 事前処理であったクエリの前処理を不要にしてソフトウェア内に組み込んだ点であった．一方で検索処理のみを見ると，先行研究の方が高い並列化効率であり，タスク粒度の設定・スケジューリング方法などについては先行研究での手法が適していると考えられる．このことから，先行研究の GHOSTZ-MP でのクエリの前処理を不要とできれば高い並列化効率が得られると考えられる．そこで，本研究における提案手法と先行研究の評価実験を行った結果から，より効率的と考えられる手法を提案する．

本研究の提案手法ではタスクの粒度を 1 クエリブロック・1 データベースチャンクに対する検索としていたが，新

たに提案する計算手法では，GHOSTZ-MP のスケジューリング方法を用いて，クエリをプロセス数で分割して 1 プロセスのタスクとする．また，GHOSTZ-MP では事前に分割したクエリファイルとジョブファイルが必要であったが，本研究で提案したクエリブロックの分割ルールならば，クエリファイルの全体のサイズが分かればワーカープロセスは独立してクエリブロックの境界を決めることができる．そのため，クエリの前処理とジョブファイルの生成は必要とせず，シングルプロセスの GHOSTZ-GPU と同様の操作方法で利用可能だと考えられる．

6. まとめ

6.1 結論

本研究では，配列相同性検索ツール GHOSTZ-GPU を MPI 環境で大規模に動作させる GHOSTZ-MP の高速化を目的として，手法の提案・実装・評価実験を行った．

先行研究の GHOSTZ-MP では，ユーザーが事前にジョブファイルを準備することや，解析するクエリファイルを分割する前処理が必要であり，ユーザーに要する操作と，前処理にかかる時間の点で問題がある．本研究では，先行研究で挙げられていた問題点について改良した実装を提案し，64 ノードを用いた大規模並列での評価実験では，シングルプロセスの GHOSTZ-GPU の実行時間と比較して先行研究よりも高い高速化率を達成した．

評価実験の結果，64 ノードを用いた大規模並列環境で 1,000 万本のクエリを用いた場合には，シングルプロセスの GHOSTZ-GPU と比較して，先行研究が 28.5 倍の高速化率であるのに対して，提案手法は 36.3 倍であり，提案手法の方が高い高速化率を達成した．最も効果のあった点はクエリの前処理を不要にした点であり，この点はユーザビリティの向上にも貢献している．手法の比較の結果有効であった，先行手法におけるスケジューリング手法と，提案手法におけるクエリの分割方法などから，より効率的だと考えられる手法についての提案を行った．

6.2 今後の課題

- BeeGFS について
  本研究では BeeGFS を全プロセスが共有してアクセスできるスクラッチ領域として利用した．BeeGFS では透過的に全プロセスがアクセス可能であるが，データの実体がどのノードに配置されるかはユーザーには制御できない．また，BeeGFS のようなシステムが利用できない場合には，Report Phase において各プロセスが各タスクの結果を送受信する必要があり，その順序について議論の余地がある．
- スケジューリングについて
  提案手法でのスケジューリング時にデータベースの再読込はタスクが割り当てられた後に行われるが，残りのタスクの状況を見て投機的に読み込みを行う方法などが考えられる．

謝辞 本研究の一部は JST CREST「Extream BigData」(JPMJCR1303) の支援を受けて行われた．

参考文献

[1] Huttenhower C., Gevers D., Knight R., *et al.* Structure, function and diversity of the healthy human microbiome. *Nature*, 486(7402), 207, (2012).

[2] Arumugam M., Raes J., Pelletier E., *et al.* Enterotypes of the human gut microbiome. *Nature*, 473(7346), 174, (2011).

[3] Altschul S. F., Gish W., Miller W., *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 403–410, (1990).

[4] Zhao Y., Tang H., and Ye Y. RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. *Bioinformatics*, 28(1), 125–126, (2011).

[5] Buchfink B., Xie C., and Huson D. H. Fast and sensitive protein alignment using DIAMOND. *Nature Methods*, 12(1), 59–60, (2015).

[6] Suzuki S., Kakuta M., Ishida T., *et al.* GPU-acceleration of sequence homology searches with database subsequence clustering. *PLOS ONE*, 11(8), e0157338, (2016).

[7] Sharpton T. J. An introduction to the analysis of shotgun metagenomic data. *Frontiers in Plant Science*, 5, (2014).

[8] illumina NovaSeq 6000, https://jp.illumina.com/systems/sequencing-platforms/novaseq.html

[9] National Center for Biotechnology Information, https://www.ncbi.nlm.nih.gov/

[10] DNA Data Bank of Japan, ERR315856, https://trace.ddbj.nig.ac.jp/DRASearch/run?acc=ERR315856

[11] 藤井智也, 角田将典, 大上雅史, ほか GPU を用いたメタゲノム配列相同性解析ツールの MPI 並列化と応用. 研究報告バイオ情報学 *(BIO)*, 45, 1-4, (2016).

[12] Matsuzaki Y., Uchikoga N., Ohue M., *et al.* MEGADOCK 3.0: a high-performance protein-protein interaction prediction software using hybrid parallel computing for petascale supercomputing environments. *Source Code for Biology and Medicine*, 8(1), 18, (2013).

[13] Kakuta M., Suzuki S., Izawa K., *et al.* A Massively Parallel Sequence Similarity Search for Metagenomic Sequencing Data. *International Journal of Molecular Sciences*, 18(10), 2124, (2017).

[14] BeeGFS, https://www.beegfs.io/content/