GXPシステムとそれを用いた大規模テキスト処理の実行

柴田知秀　姜ナウン　黒橋禎夫
京都大学大学院情報学研究科
田浦健次朗　Choi SungJun　Dun Nan　松崎拓也　辻井潤一
東京大学大学院情報理工学系研究科
河原大輔
情報通信研究機構
宇野毅明
国立情報学研究所

1　はじめに

本プロジェクトではのちに述べる二つの大規模な情報検索 (3 節), 情報抽出処理 (4 節) を実行した. それらのタスク自体の学術的な意義の他に, 本プロジェクトでは, HA8000 システムをデータ処理が中心のワークロードに適用すること, それを著者らが設計・実装した分散並列シェル GXP [6] [1]およびその上のワークフローシステム GXP make を用いて, 高い生産性で実行する事により, HA8000 の新しい適用分野を開拓する事を目指した.

本プロジェクトで実行した, データ処理中心のワークロードには一般に以下のような特徴がある.

1. 大量, 多数のデータの入出力を行う.
2. 処理時間がデータの性質に大きく依存し, 少数のパラメータからの計算量予想などがしにくい (稀なデータに対する実行時間やメモリ使用量などの「サプライズ」が頻繁に発生している).
3. 場合によっては, 新しいデータに対して処理が失敗することもあり, プログラムの修正と実行のサイクル (試行錯誤) を繰り返す必要がある.
4. ワークロード全体は, 複雑なデータ処理を行ういくつものプログラムを組み合わせて構築されており, 短いカーネルループの性能を向上させるような努力で, 全体性能が向上することはあまり期待できない. 性能を FLOPS 値で測ったり, ましてそれをマシンの最高理論 FLOPS 値に近づける, などといった評価基準を適用することは難しい.

[1] http://www.logos.t.u-tokyo.ac.jp/gxp/

全体として, プログラムの最高性能の向上を求めてプログラムをチューニングする努力は報われにくく, それよりもプログラムの生産性 (最初から失敗せずに並列化でき, 効果的な台数効果を得られること, 失敗の原因が容易に分かり, 並列化のためのプログラム統合に費やす時間が少ないこと) が重要である. また, 計算機の性能に対する要求としては, ファイルシステムの IO 性能がある程度高い (劣悪でない) ことが上げられる.

改めて, システム開発や評価に関連する本プロジェクトの目的を述べると以下のようになる.

1. GXP および, その上のワークフロー処理系 GXP make の HA8000 への移植を行い, HA8000 ユーザへ提供する準備を整えること.

2. GXP make のスケーラビリティを HA8000 規模 (数千〜15,000 並列度) のシステムで検証すること.

3. データ中心のワークロードに対する HA8000 の性能, 適用可能性を評価すること.

結果として, 本プロジェクトのために提供いただいた 8,192 CPU core 程度の並列性に対して, GXP make 自身は良好なスケーラビリティを有していることが確認されたが, HA8000 のファイルシステム HSFS の性能が非常に悪いために, 本プロジェクトで対象としたようなワークロードを効率的に実行するには, 現状の HA8000 には大きな問題があることが明らかになった.

もちろん, HSFS の性能が悪いことについて事前の情報と, 実験結果があったため, 我々はその条件下で, 極力 HSFS を経由せずにデータを受け渡すためのプログラムを開発するなど, できる限りの準備を行っていた. にもかかわらず残念ながら 8,192 CPU core の割り当て時間中には, 進行状況を監視するための数分おきのファイルアクセスを行うだけで, 計算がほとんど進行しない状態となってしまった. そのため本報告書で述べる実験結果には, 後日, 東工大 TSUBAME や, HA8000 上の通常の割り当て時間を用いて小規模に少しずつ実行した結果が含まれていることをお断りしておく.

2 GXP, GXP make

本プロジェクトで用いる処理系 GXP, およびその上のワークフロー実行系である GXP make について簡単に説明する.

GXP は並列・分散環境用のシェルであり, 本プロジェクトで用いた HA8000 は当然として, SSH でアクセスできる Beowulf クラスタでも, その他のバッチスケジューラ環境でも, それらの混在で有っても同じ対話的環境を提供する. 本節では, GXP の基本的なユーザインタフェースと, HA8000 のバッチスケジューラコマンドが規定する利用モデルへ GXP を適合させるために, 必要な概念について説明する. なお, GXP は以下の URL から入手可能である.

http://www.logos.t.u-tokyo.ac.jp/gxp/

2.1 GXP の基本ユーザインターフェース

GXP は, (1) 多数の計算ホストへ同時にログインし, (2) それらの任意のホスト上でコマンドを起動する, ことを基本とする. 例えば以下は, ホスト y001 から, ノード群 y002〜y031 に同時にログインし, それらの上で hostname コマンドを実行するための一連の操作である. ログインには SSH を用いるものとする.

```
$ gxpc use ssh y
gxpc: no daemon found, create one
[1/1/1]$ gxpc explore y[[002-031]]
reached : y007 (y007)
reached : y030 (y030)
reached : y019 (y019)
...
reached : y008 (y008)
reached : y025 (y025)
gxpc : took 1.653 sec to explore 29 hosts
[31/31/31]$ gxpc e hostname
y001
y013
...
y023
y009
[31/31/31]$
```

1 行目 *gxpc use ssh y* で, y で始まるホスト間 (正確には, 正規表現 y にマッチするホスト間) では, SSH コマンドを用いれば到達可能であることを指示し, 2 行目 *gxpc explore y[[002-031]]* で実際にそれを用いて, y001〜y031 のホストへの SSH ログインが行われる. 3 行目 *gxpc e hostname* は, それらすべてのホストで hostname コマンドを実行する.

e コマンドではオプションの指定により, 一つのノードや, 一部のノードの集合を指定して, それらでのみコマンドを実行することができる.

一般に GXP は,

- use コマンドを用いて, どのホストからどのホストへ, どういう手段を用いて直接ログインが可能かを指定し,
- explore コマンドを用いて実際にホストへ到達し,
- e コマンドを用いてそれらのホストへコマンドを投入する,

という手順で用いる. それらをどういう順序で何度用いてもよく, ひとたび explore コマンドでホストへ到達した後は, 個々の e コマンドはホスト数が多くても対話的な速度で高速に実行される.

2.2 GXP とバッチスケジューラ

HA8000 のように, 計算ノードへのアクセスが SSH ではなく, バッチスケジューラ (qsub コマンド) を介して行われる場合, *gxpc use ssh y* の部分で代わりのコマンドを指定する．qsub コマンドを, 何のオプションもなく用いればよい TORQUE 環境であれば, 以下で, バッチスケジューラ経由で一つのプロセスを立ち上げることができる

$ *gxpc use torque y001 node*
gxpc: no daemon found, create one
[1/1/1]$ *gxpc explore node*

gxpc explore node のところで実際に, qsub コマンドを用いて計算ノード上でプロセスが起動される．もちろんどの計算ノードで起動するかはバッチスケジューラ次第であり, explore は (調節可能な) タイムアウト以内にプロセスが起動しなければ, ノードの獲得に失敗したとみなす．

ここで, 計算ノードを複数 (N 個) リクエストするには, 通常 *gxpc explore node* の代わりに, *gxpc explore node* N のように指定すればよい．これは, qsub コマンドを N 回立ち上げることに相当する．

バッチスケジューラからは, N 個の独立なリクエストが来たように見えるため, 同時に実行可能なジョブ数に制限のある環境では, この方法を用いて同時に到達可能なノード数は, その数に制限される．HA8000 では, 1 ユーザが同時に走らせることができるジョブ数は 2 以内に制限されているため, この方法を用いて獲得できるのは 2 ノードまでとなる．

2.3 HA8000 上の GXP

HA8000 のように, 1 ユーザが同時に実行可能なジョブ数を少なく (2) 制限している環境で, 多数のノードを用いた計算を行うには, 1 ジョブで多数のノードをリクエストする必要が有る．HA8000 はバッチスケジューラとして日立製 NQS を利用しており, それは実際には TORQUE へのフロントエンドとして動作している．1 ジョブで多数のノードを要求することが可能であり, その場合でもあくまでプロセスは一つだけ起動される．残りのプロセスを起動するのはユーザの役割である．それには主に二つの方法が有る．

1. どのノードがそのジョブが割り当てられているかは環境変数 (PBS_NODEFILE) に記されたファイルを介して取得できる．これを参照し, それを元に SSH 経由でそれらのノードへ到達する．

2. TORQUE API を用いる．残りのホスト上では, TORQUE API を用いてプロセスを起動することができる．この際ホスト名を直接指定することはできず番号で指定する．そのため, TORQUE から割り当てられていないホスト上でプロセスを起動することは原理的にできない．

HPC 特別プロジェクト実施時には方法 1 を用いていたが, プログラムの間違いなどで, 割り当てられていないノードへもログインできてしまうこと, SSH で起動されたプロセスには資源のアカウンティングが行われないこと, などにより, 方法 2 を開発した. これは本プロジェクトを通じて得た, GXP の改良である. もちろんこの方法は, TORQUE を用いたあらゆる環境で用いることができる方式である.

GXP を経由して HA8000 を用いることにより, バッチスケジューリング環境では難しい, 対話的なジョブの実行やデバッグが可能になる.

2.4 GXP make

GXP make は, GXP の e コマンドの上に構築された, make の分散並列処理系である. ユーザは通常の Makefile を記述し, 通常の make コマンドの代わりに, *gxpc make* コマンドを実行する. GNU make には, -j オプションをつけることで依存関係のないジョブがノード内で並列実行される機能があるが, GXP make では-j オプションで分散並列実行される (explore で到達したノードへジョブが分散される).

GXP make は, 内部で無変更の GNU make を起動している. GNU make はプロセス起動のためのシェル (デフォルトでは/bin/sh) を変更する機能を備えているため, それを用いてプロセス起動部分を intercept することができる. GXP make はその機能を用いてすべてのプロセス起動 (以下, make ジョブ) を intercept して, スケジューラプロセス (xmake) にキューイングする. xmake は実行すべき make ジョブと, 現在ジョブが割り当てられていないノードを管理しながら, make ジョブをディスパッチする.

GXP make は通常の make とまったく同じ記述で分散並列実行をサポートし, すべての GNU make の拡張構文や機能なども, GNU make を無変更で用いているが故に, 自動的にサポートする. 大規模な並列処理向けの機能として, ジョブの進行状況や並列度の時間推移などを表示する (html を生成する) などの機能を持つ.

2.5 GXP make のスケーラビリティ

GXP make は, 1 つのスケジューラプロセスがすべての make ジョブリクエストを受け取ってジョブをディスパッチするという構成をとるため, 高並列で実行する際にはこの部分がボトルネックになる可能性がある. 一方で, 依存関係の解析を並列に行うのは難しく, まして無変更の GNU make を用いるという制約下では, この部分の潜在的ボトルネックを完全に取り除くのは難しい. そこでまず GXP make が本タスクを実行するのに十分なスケーラビリティを持つかどうかを, 予備実験によって定量的に検証した.

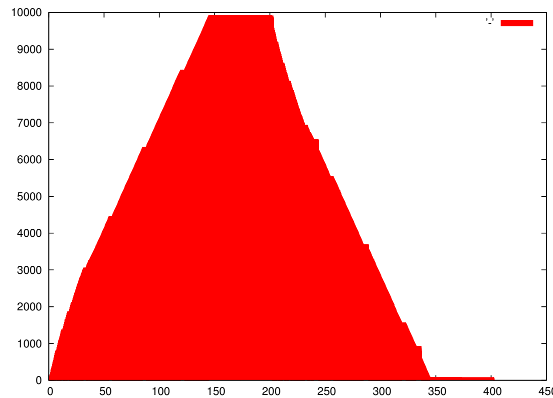

図 1: GXP make のジョブディスパッチ速度. 横軸=経過時間, 縦軸=並列度で, 開始 150 秒で頂上で並列度 10,000 に達している

図 1 は HA8000 システムにおいて, 200 秒 sleep するだけのジョブを 10,000 個投入したときの, 並列度の時間推移を示したものである. 約 150 秒ほどで 10,000 のジョブを投入し終え (並列度 10,000 へ到達), 200 秒経過後から, 徐々にジョブが終了している事が見て取れる. 言い換えれば, 60 ジョブ/秒程度のジョブディスパッチ性能を持っており, この程度の粒度以上のジョブが実行時間の大半を占めるようなタスクに対して, GXP make は 10,000 程度の並列性を十分に引き出せるということがわかる.

ただしこの速度は, GXP の処理系, および GXP を実行する Python の処理系がともにローカルファイルシステムにあったときの性能である. GXP 処理系や Python 処理系が共有ファイルシステムに置かれたときの性能は, 用いているファイルシステム, 設定 (PATH 環境変数など) で大きく異なっており, 条件により数倍〜10 倍程度のスローダウンが観測された. 同じ共有ファイルシステム (NFS) を用いていても大きく性能が異なっており, 今後詳細な調査を行う予定である. 本実験に置いては, GXP, Python をローカルファイルシステムに配置している.

3 生物医学テキストの, 深い自然言語処理を用いた索引づけ

3.1 タスクの目的と構成

本タスクは, 生物医学テキストの文献データベース MEDLINE[2] の抄録に対して, 高速な HPSG 構文解析器 [3] を核技術とした自然言語解析を行い, 構文情報を加味した索引づけを行う. 本タスクは, 東京大学情報理工学系研究科辻井潤一研究室により開発された.

この索引付けにより, 同じ意味を持つ構文的なバリエーションにとらわれない, 意味上の主語や意味上の目的語を指定した検索を行うことができる. この索引を用いた検索システム MEDIE[2]

[2] www.pubmed.gov

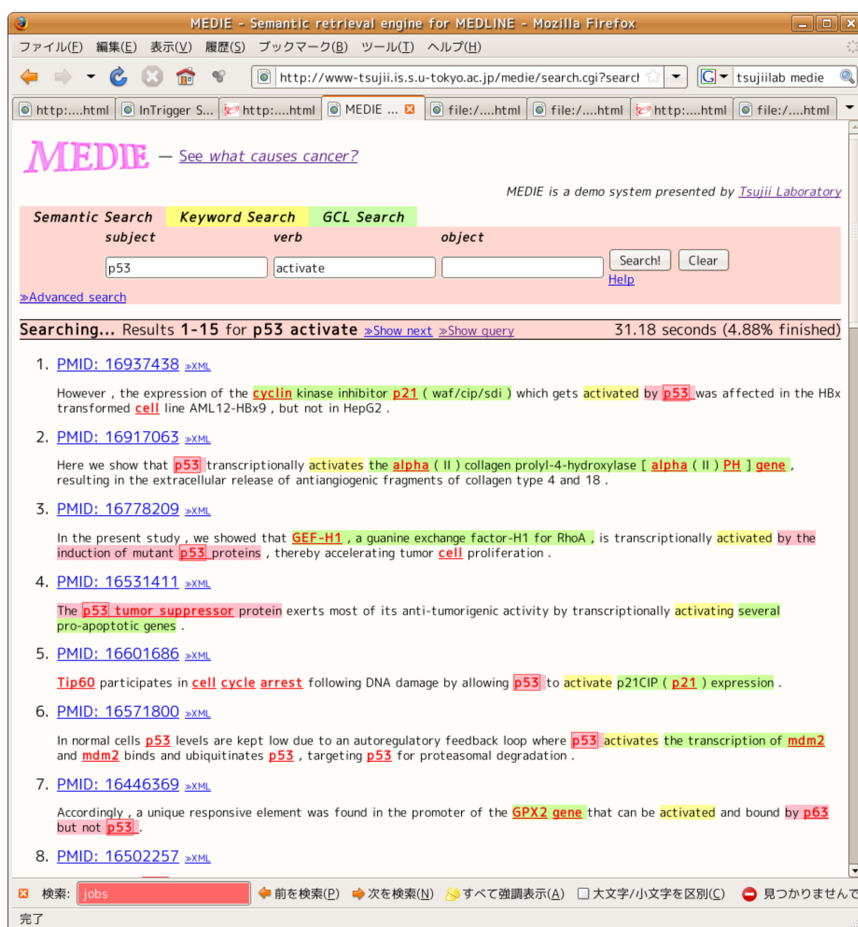

図 2: MEDIE 検索システム

が稼働しており, http://www-tsujii.is.s.u-tokyo.ac.jp/medie/ を通してアクセス可能である．そこで以下では本ワークフローのことを, **MEDLINE2MEDIE ワークフロー**と呼ぶことにする．

図 2 に, MEDIE で可能な検索と, 結果の一例を示す．図では, Subject (意味上の主語) が p53, Verb (動詞) が, activate であるような文の Object (目的語) を検索しており, p53 遺伝子が活性化するものを検索する．検索結果においては,

- . . . X which gets activated by p53 . . .
- . . . p53 . . . activates X . . .
- X, . . . , is . . . activated by . . . of p53

などの多様な構文的構造に対して, 正確に主語と述語をとらえた結果が返されている．

入力は, MEDLINE に収録されている, 1865 年から 2008 年 7 月 10 日までの全記事 (抄録が

```
<MedlineCitation Owner="NLM" Status="MEDLINE">
<PMID>17548109</PMID>
 ...
<ArticleTitle>Antimicrobial activity of truncated alpha-defensin
(human neutrophil peptide (HNP)-1) analogues without disulphide bridges.
</ArticleTitle>
 ...
<Abstract>
<AbstractText>Antimicrobial peptides play an important role in host defence,
 ...
 ...
 ...
use in therapeutic interventions.</AbstractText>
</Abstract>
 ...
</MedlineCitation>
```

図 3: 入力ファイルの抜粋

収録されているものと, いないものがある) で, 抄録数にして約 1,000 万, 文数にして約 1 億からなる.

それらが, いくつかの記事ごとに 1 ファイルにまとめられ, 入力ファイルとしては合計で 767 個のファイルからなる. 入力ファイルの一部分を抜粋したものを図 3 に示す. 出力は検索システム MEDIE が必要とする索引 (medie_db) であり, これを元に上記のような検索が実現される.

MEDLINE データベースに登録されている文献数は, 時間とともに指数関数的に増大しており, 検索システムの新鮮さを保つため, 計算能力にもそれに対応できるだけの向上が要求される. また, 抄録だけでなく論文の全文を検索の対象としたり, 全文から蛋白質の相互作用などのイベント抽出を行って新しい反応経路などを抽出・発見するなど, 処理能力に対する要求は今後も大きくなって行く.

異なるファイルに対する処理は独立に実行可能であるが, それで引き出せる並列度 (767) は十分ではなく, 一つのファイルをさらに分割して並列処理する必要がある. 特に時間を要する部分は構文解析部分で, ファイルごとに含まれる抄録数が大幅に異なること, 構文解析に要する時間が文により異なり, 事前に予測するのが困難であることなどから, この部分はある程度細かい粒度に分割する必要がある. 言い換えれば, 一つの入力ファイル自体をワークフローとして実行する必要がある. この記述は GXP make を適用する以前から, Makefile を用いて行われており, GXP make によって並列実行をするために施した変更はほとんどない. 図 4 に, 1 入力ファイルに対するワークフローを図式化したものを示す. ノードが一つのジョブ (逐次コマンド), 矢印はタスク間の依存関係を示し, それは同時にジョブ間で中間ファイルの形で受け渡されるデータを意味している.

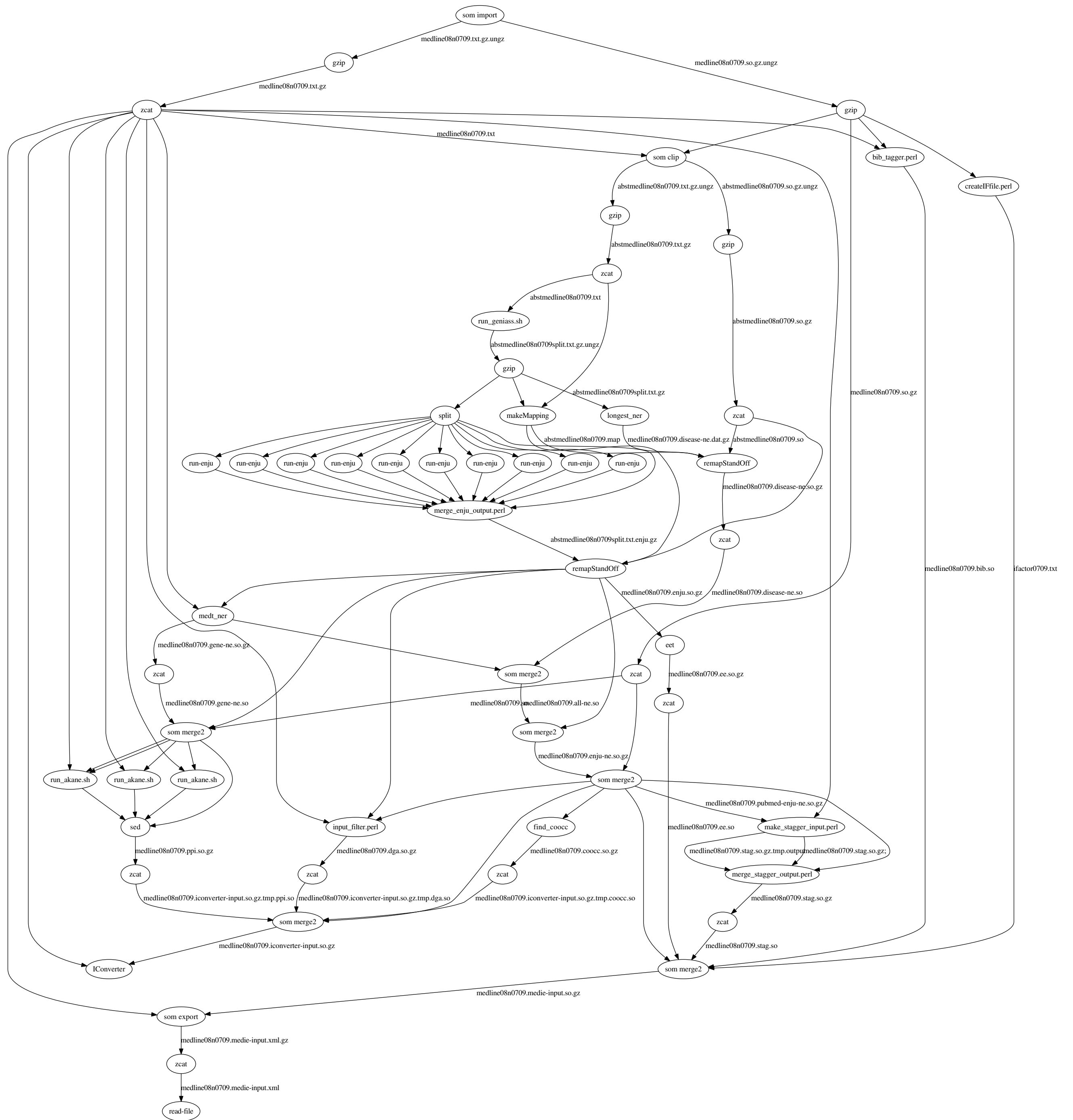

図 4: MEDLINE2MEDIE のワークフロー (1 入力ファイルにつき)

グラフの中央部で, “run-enju” という名前のノードが横に多数並んでいるが, それが構文解析を並列に実行している部分であり, 典型的には 40 個〜200 個程度である (図では見やすさのために少なくしている). CPU 時間の大部分はここで消費される. 左側に “run_akane.sh” という名前のノードが 3 つ程並んでいる部分では, 蛋白質-蛋白質の相互作用の抽出が行われており, ここでも 1 入力ファイルに対する並列化が行われている. CPU 消費量や並列度という観点から見ると, run-enju が占める部分が非常に多く, これ以外の部分は処理量という観点からは, 前処理・後処理と言ってよい.

中間ファイルは, 通常は共有ファイルシステムに置かれることを想定している. ユーザにとっての利便性や開発コストを考えると, そのようにして実行できることが望ましいのはもちろんであるが, 後に述べるように, 共有ファイルシステムとして HSFS を用いると, 大規模な実行を効果的に行うことは困難であったため, 今回の実験では次節で述べるような, ファイルを明示的にコピー (ステージング) する枠組みを追加して実験を行った.

3.2 準備

GXP make は, その make に準じた実行モデルから, 中間結果の受け渡しにファイルを利用する. これには, 長時間ジョブの中断と再開を自然に可能にするという, チェックポイントの効果もある. それらのファイルは共有ファイルシステムに置くのが簡便であるが, 転送性能を引き出し, ファイルサーバへの負荷の集中を避けるには, ノード間で明示的なコピーが望ましい場合もある.

HA8000 上においては 512 ノードの実行が割り当てられる以前の準備段階で, ワークフロー中の前処理・後処理に要する時間が他のシステムと比較して長く, それにより全体性能が悪化するという問題が明らかになっていた. 例えば図 5 は, 執筆時点で, 100 の入力ファイルを処理した際の, 最初の 2,500 秒程度の並列度の時間推移を, 二つのシステムで比較している. 左は HA8000 (16 CPU core × 8 ノード), 右は InTrigger システム[3]の筑波大学に設置されたクラスタ (8 CPU core ×16 ノード) を用いたものである. 前・後処理部分の比較のため, 構文解析部分を小さなデータに置き換えて比較している (したがって, 実際の性能比はここまでの差は出ない. あくまで前・後処理の比較が目的であることを断っておく). どちらも, 1 コアあたり 4 つまでのジョブを許容し, 合計で 500 程度の並列度を許容している. この様にした理由は, 我々が普段使える HA8000 環境の実コア数が 128 に限られているからである.

後者の共有ファイルシステムは NFS, ネットワークは 1000Base-T を 2 本トランクしたものである (計算ノード, ファイルシステム共). 前処理にかかっている時間は, グラフ上で並列度が最高に達するまでの時間で見積もれ, およそ 6 倍程度の開き (前者が 250 秒程度, 後者が 1,500 秒程度) がある.

[3]www.intrigger.jp

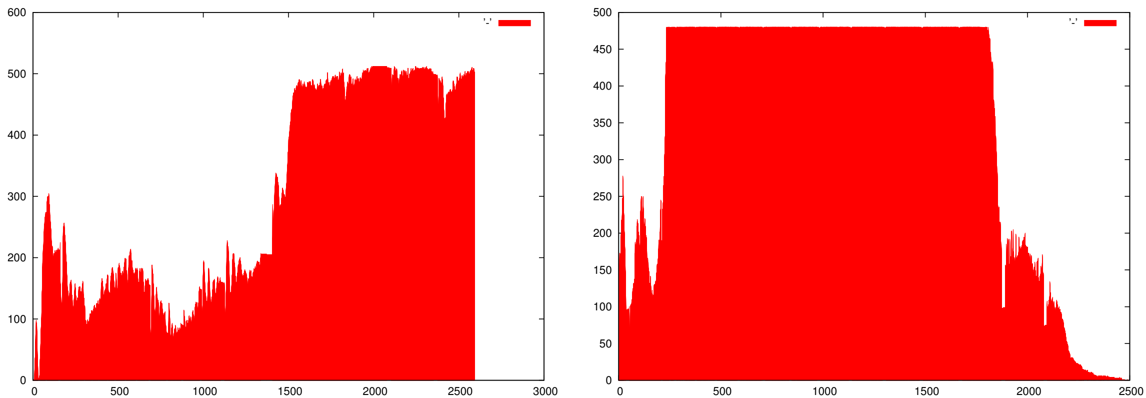

図 5: 100 ファイルを処理した際の並列度プロファイル (開始から 2,500 秒まで). HA8000 HSFS 使用 (左), InTrigger 筑波 NFS 使用 (右)

このまま大規模な実行を HSFS を用いて行うことは困難であると判断し, HSFS を中間ファイルの格納に用いない方式を採用した. 具体的には, ジョブの直前でファイルを明示的にコピー (転送) する補助システム FCP を実装した. FCP を用いた実行では, ファイルの読み書きはすべて計算ノードのローカルディスクに対して行われる. 個々の make ジョブ (make によって起動される個々のコマンド) の終了時は, ローカルディスクに書かれた実行結果をそのままにしておく. 逆に make ジョブの開始時は, そのジョブが読み込むであろうファイルを, それを作成した計算ノードから (scp コマンドを用いて) コピーする. ある make ジョブが読み込むファイルは, 一般には make ジョブ開始時に知ることはできないが, ここでは多くの Makefile で記述されたジョブに通用し, このタスクにもほぼ通用した経験則として, コマンドラインを見て, そこに現れる, パス名らしき文字列を入力ファイル (を包含する集合) とみなす, という方法をとった.

GXP make 自身には, コマンドラインをユーザが書き換えて, 前・後処理を挿入することができる枠組みを追加し, アプリケーション固有の前・後処理を自由に挿入できるようにした. 前処理で, 上記の方法で認識した入力ファイルと思しきファイルの作成ノードを突き止めて, それを scp でコピーする. あるファイルを作成したノードを見つけるために, メタデータサーバ (FCP サーバ) を走らせる. FCP サーバは, あるファイルに対する問い合わせを初めて受け取った際はすべての計算ノードに問い合わせを発行し, 返された結果を以降の問い合わせのためにキャッシュする. もちろん FCP サーバ自体がボトルネックになり得るが, ファイル作成時のオーバーヘッド (メタデータ更新) がないこと, 転送その物はサーバを介さずに行われること, 実装が単純であることなどから今回の目的に合致していると考える.

ファイル番号	実行時間 (秒)
001-199	20,664
200-299	36,032
300-399	48,615
400-499	54,695
500-599	45,824
600-767	29,784

表 1: HA8000 30 ノード (HSFS 不使用. FCP 使用) での実行時間

3.3 512 ノードでの実行結果

512 ノード (8,192 コア) が割り当てられた約 12 時間のスロットの後半を利用して, すべての入力ファイルを処理することを目標にした. 入力ファイルとノードを 2, 3 のグループに分割して, 一度に処理するファイル数を制限した. 中間ファイルはもちろん, GXP 処理系や Python を含めて, 極力 HSFS を参照しない工夫を行った. それでも構文解析に至るまでの前処理が数時間, 遅々として進まなかった. 実行の途中でそれは, 計算結果を蓄積するために時折行われている, ローカルディレクトリからホームディレクトリへのファイルのコピーに起因することが明かになった. そのコピーは, ジョブ終了後に計算ノードのローカルディスクへアクセスできない以上必須のものであり, 必要最低限のファイルアクセスとして実行していたが, これがスローダウンを引き起こすこととなってしまった. 調査のためそのコピーをするプロセスを殺すと, 並列度が急激に向上した. しかしながら割り当てられた時間の大部分を使いきっており, まとまった結果は残念ながら得られなかった.

3.4 事後処理

8,192 並列度を用いた処理の結果が芳しくないものであったため, その後 HA8000 の 30 ノード (480 コア. FCP 使用) を用いて全ファイルを分割処理した. 前者の結果の要約を表 1 に示す.

また, 東工大 TSUBAME (1024 コア, Lustre ファイルシステム使用) を用いて上記の 400-499 番の 100 個の入力ファイルを同時処理した結果が図 6 である. 実行時間は 35,000 秒程度で, HA8000 上での 480 コアの結果 54,695 秒と比較すると, Lustre を用いた TSUBAME と, HSFS を用いずに FCP を用いた HA8000 との間に大きな性能の差はないと言える.

3.5 まとめ

本タスクに関して現状で, 実証できたこと, 達成できなかったことは以下のとおりである

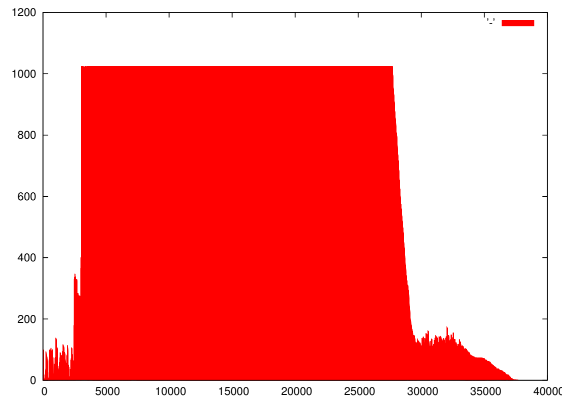

図 6: TSUBAME 上で 100 ファイルを 1,024 コアで処理した際の並列度プロファイル (Lustre ファイルシステム使用)

1. GXP make 自身は, GXP 処理系や Python をローカルディスクに置いた場合, 150 秒程度で 10,000 タスクを分散させるスループット (約 60 タスク/秒) を持っており, 一タスクが数分を要するワークフローであれば HA8000 規模の並列度を有効活用できることが実証された.

2. しかし HA8000 のファイルシステム (HSFS) 性能は, 多数のファイル作成や, 小さい単位での入出力を伴うワークロードに対して問題があり, はるかに廉価な選択肢 (NFS) に大きく劣っている.

3. Lustre ファイルシステムを用いれば 100 入力ファイルを 1,000 程度の並列度で, 効果的に実行できることが確認できた. NFS を用いた場合については, 500 程度の並列度でそれが確認できた. これ以上の規模については実験の機会を得ることができておらず, 実証できていない.

4. ファイルのステージングを行うツール FCP は有効に機能し, 500 程度の並列度で良好な性能を示したが, それ以上の規模にスケールすることは, まだ実験の機会を得ることができておらず, 実証できていない.

4 類似文字列検出による大規模ウェブページコレクションからの類似ページの発見

4.1 研究の背景・目的

本研究では日本語ウェブページ 1 億件という大規模ウェブページコレクションを対象として，あらゆるページのペア間に含まれる類似文字列の検出を行ない，その結果から，ミラーページ，

引用ページ，スパムページなどを同定する．本稿では類似文字列を含むページペアを類似ページと定義する．

類似ページを検出することは，大規模ウェブコレクションを検索エンジンの検索対象とする際に，以下の点で検索エンジンの質を向上させることができる．

- 同一・類似ページを検出し，それらをまとめて提示することにより，ユーザの検索結果把握を阻害させないようにできる．
- 他サイトの引用のみで記事を作ったようなスパムページを検索対象から除外する．

また，検索エンジンの質の向上だけでなく，実際のウェブに，盗作や，悪意のあるスパム的なものも含め，どのような引用関係がどのような割合で存在しているかなどの興味深い分析が可能となる．

本研究で対象とする類似ページを以下のように分類する．(図 7 を参照)

- **同一**: 2 つのページが同一のものである．例えば，ミラーページや，定型ページ (apache が生成するページなど)，盗作ページなどがある．
- **包含**: 1 つのページが他方のページに包含される場合である．例えば，ブログの月別ページと日別ページなどがある．
- **部分共有**: 2 つのページでその部分を共有するページである．例えば，引用ページと被引用ページ，文集合を共有するページ (例えば,「当Ｗｅｂサイトの手法，レイアウト，電子ファイル等は当社及び，グループ会社が権利を有します．以下の内容を無断で行う事を禁じます.」という文集合を共有するページ)，スパムページ，盗作ページと被盗作ページなどがある．

本研究では，まず，高速類似文字列検出アルゴリズム SACHICA を用いて，あらゆるページペア間に含まれる類似文字列の検出を行なう．その際には，ウェブコレクションを適当な大きさのデータを分割し，GXP を用いて並列に処理する．64 ノード (1,500CPU コア) を用いると，500 万ページにおける類似文字列検出を約 1 時間半で行なえることを確認することができた．1 億ページに対して全ての処理を行なうにはこの 400 倍の処理を行なう必要があるが，このうち，2,000 万ページにおけるあらゆる類似文字列を検出した．次に，この結果から，類似ページを検出し，同一ページ，引用ページなどに分類した．

関連研究として，Manku らは Web ページクローリングの際にクロールしたページがすでにクロールしたページの類似ページかどうかを判定し，類似ページであればクロールしないといった手法を提案している [1]．類似ページの判定はまず各ページを f ビットの fingerprint に変換し，2 つのページの fingerprint の違いが k ビット以内であるかをチェックすることによって行

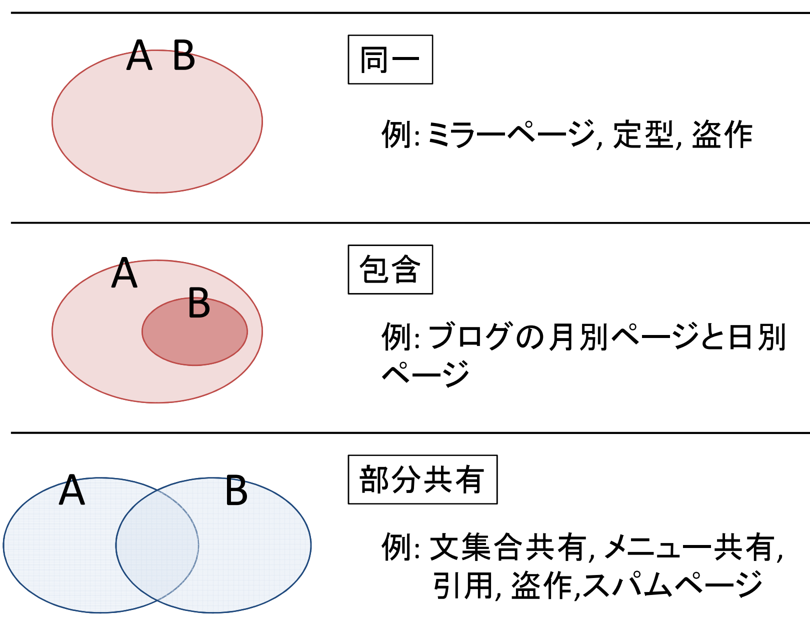

図 7: 類似ページの分類

なっている．この研究では上記の分類での同一もしくはほぼ同一ページしか検出することができず，ページの一部が類似しているページペアは検出できない．

4.2 ウェブコレクションと高速類似文字列検出プログラム SACHICA

ここでは，本研究で用いたウェブコレクション，ならびに，高速類似文字列検出プログラム SACHICA について簡単に説明する．

4.2.1 ウェブコレクション

ウェブページコレクションとして，検索エンジン基盤 TSUBAKI[5] で検索対象となっている 1 億ページを用いた．各ページは HTML タグの削除・文抽出などの処理が行なわれ，標準フォーマット [4] と呼ばれる XML 形式に変換されて管理されており，URL・文字コードなどのメタ情報，文集合，インリンク (このページにリンクしているページ集合)，アウトリンク (このページがリンクしているページ集合) などの情報が含まれている[4]． 図 8 に標準フォーマットの例を示す．<RawString>タグで囲まれた部分が HTML から抽出された文を示す．

[4]実際には形態素解析結果・構文解析なども含まれている．

```
<?xml version="1.0" encoding="utf-8" ?>
<StandardFormat Url="http://www.kikuchinobuhide.com/assembly/10index.html" OriginalEncoding=
"shiftjis">
<Header>
<Title Offset="345" Length="15">
 <RawString>議会答弁集</RawString>
</Title>
<OutLinks>
 <OutLink>
  <RawString>平成１６年市議会答弁集</RawString>
  <DocIDs>
  <DocID Url="www.kikuchinobuhide.com/assembly/16index.html">050947710</DocID>
  </DocIDs>
  </OutLink>
 <Text Type="default">
 <S Offset="8248" Length="88" is_Japanese_Sentence="1" Id="1">
  <RawString>●平成１９年市議会答弁集</RawString>
 </S>
 <S>
  ...
 </S>
 </Text>
</StandardFormat>
```

図 8: 標準フォーマットの例

4.2.2 高速類似文字列検出プログラム SACHICA

SACHICA(Scalable Algorithm for Characteristic/Homogenous Interval CAlculation)[5] とは入力した文字列ファイルから，決まった長さの部分列の組でハミング距離が閾値以下のものを全て見つけ出すアルゴリズムである．この種の他のアルゴリズムに比べ，正確性を犠牲にせずかつより高速であり，主にゲノム相同性解析を目的として開発された．本研究ではこのプログラムをウェブページ中の類似文字列検出タスクに利用する．

4.3 ジョブの分割

4.3.1 前処理

簡単な URL の正規化でわかる同一ページは今回の処理からは除外した．表 2 に 5 つのタイプとそのページ数を示す．例えば 2 番目のタイプの場合,「http://.../%7E...」の「%7E」を「~」

[5] http://research.nii.ac.jp/~uno/code/sachica05.zip

表 2: URL の正規化

タイプ	ページ数
スラッシュの重複を削除	2,560,297
%7E をチルダに置換	802,902
ホスト名の最後のドットを削除	250,732
index.(html|html|cgi|php) を削除	1,405,483
「http://www.」の「www」の有無	5,604,769

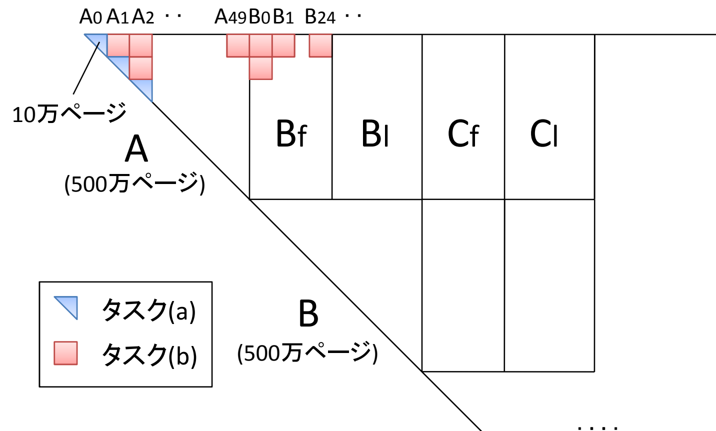

図 9: ジョブの分割

に置換したページが 1 億ページ中に存在する場合，このページを処理対象から外した．

4.3.2　ジョブの分割

以下で，類似文字列検出を行なうジョブの分割方法について述べる．分割方法を図 9 に示す．

1. 1 億ページを 20 分割し，500 万ページずつにする．これを **chunk** と呼ぶ．(順に，A, B, C, ... と名前をつける)．
2. chunk それぞれをさらに 50 分割し，1bin とする．(A を A_0,A_1,...,A_{49}，B を B_0,B_1,...,B_{49}，... に分割する．)

   1bin は約 10 万ページの集合からなる 1 ファイルであり，図 10 に例を示す．各ページから標準フォーマットの<RawString>タグで囲まれたものを文として抽出し，euc-jp に変換した後，各ページの文集合を空行で連結したものである．図 10 では,「●平成１９年市議会答弁集」から「Ｋｉｋｕｃｈｉ　Ｎｏｂｕｈｉｄｅ　Ｓｕｐｐｏｒｔｅｒｓ’Ａｓｓｏｃｉａｔｉｏｎｓ.」までが 1 ページ,「ｕｐｄａｔｅｄ／２００３／７／２９」から「戻る」までが 1 ページを示す.

●平成１９年市議会答弁集
●平成１８年市議会答弁集
...
ａ）現状と問題点。
ｂ）１１９番受信体制について。
ＡｌｌＲｉｇｈｔｓ　Ｒｅｓｅｒｖｅｄ．Ｃｏｐｙｒｉｇｈｔ　（ｃ）　２００５．
Ｋｉｋｕｃｈｉ　Ｎｏｂｕｈｉｄｅ　Ｓｕｐｐｏｒｔｅｒｓ’Ａｓｓｏｃｉａｔｉｏｎｓ．

ｕｐｄａｔｅｄ／２００３／７／２９
Ｔｈｅ　Ｉｎｔｅｒｎｅｔ　Ｇｕｉｄｅ　ｔｏ　Ｌｏｄｇｉｎｇ　ｉｎ　Ｊａｐａｎ
変更、抹消、登録データー用ＦＯＲＭ
。
変更の場合は施設名と変更箇所のみ記入してください。
変更、抹消、　データーの変更抹消
登録済みサイト
日本語英語、両方日本語のみ　英語のみ
お願い：施設名の頭にＬＧｘｘｏｏとある番号を必ずお書きください。
...
戻る

北海道の病院
Ｔｈａｎｋ　ｙｏｕ　ｆｏｒ　ｖｉｓｉｔｉｎｇ　ｍｙ　ｓｉｔｅ．
...

図 10: 入力ファイルの例

3. 2 種類のタスクを考える．図 9 において，以下のタスク (a) を青三角，タスク (b) を赤四角で表す．以下のいずれの場合においても，同一ページ内にある類似文字列は類似文字列とみなさないようにする．

   (a) 1bin 内の類似文字列を検出するタスク

   (b) 2 つの bin 間にある類似文字列を検出するタスク
   この場合，SACHICA には 2 つのファイルを引数として渡す．

4. 並列計算するにあたり，上記で設定したタスクの集合からなるジョブを以下の 2 種類考え，1 ジョブを並列計算の最小単位とする．

   (a) 1chunk 内の解析をするもの (例: A 内の解析をするもの)
   このジョブはタスク (a) とタスク (b) からなり，タスク (a) が 50 タスク (A_0,A_1,...)，タスク (b) が 50 * 49 / 2=1,225 タスク (A_0-A_1, A_0-A_2,..., A_{48}-A_{49})，計 1,275 タスクからなる．

pageid1	pos1	pageid2	pos2	length
010001159	89	011522466	100	75
010001179	733	010004257	960	71
010001179	734	010005184	916	77
010001179	734	010005625	216	76
010001179	88	010040879	205	76
010001179	734	010041164	1657	77

図 11: 出力の例

(b) ある chunk と別の chunk の半分の解析をするもの (例: A と B_f(B 前半) の解析をするもの．B_f は B_0,B_1,...,B_{24} からなる)
このジョブは，50 * 25 = 1,250 タスク (A_0-B_0, A_0-B_1, ..., A_{49}-B_{24} の解析) からなる．
同様に，A と B_l(B 後半)，A と C_f，...，B と C_f，... の解析を行なう．

このようにジョブを分割することによって，1 ジョブあたりの時間がなるべく均一になるようにする．

1 億ページに対しては，4-a タイプを 20 回，4-b タイプを，38 + 36 + ... + 2 = 380 回，計 400 回ジョブを投げる必要がある．

各並列計算処理の先頭では，入力ファイルを各ノードのローカルディスクにコピーすることによって，ファイルサーバへの負荷がかからないようにした．ローカルディスクへのコピーには GXP のコマンド `bcp` を用いた．また，結果は `bzip2` で圧縮しながらローカルディスクに書き出し，終わるとファイルサーバにコピーした．

SACHICA の出力例を図 11 に示す．一行が一類似文字列ペアに相当し，pos と length は euc-jp での文字数で表す．そして，ページペアごとにまとめ，URL と類似文字列の情報を付与したものを図 12 に示す．

4.4 実験・考察

4.4.1 GXP による並列処理

GXP を使って，1 億ページのうちの 2,000 万ページに対して類似ページ検出を行なった．SACHICA のパラメータとして以下を用いた．

- 類似文字列とみなす最小文字列長: 70
- 20 文字で 1 文字あたりの異なりを許す

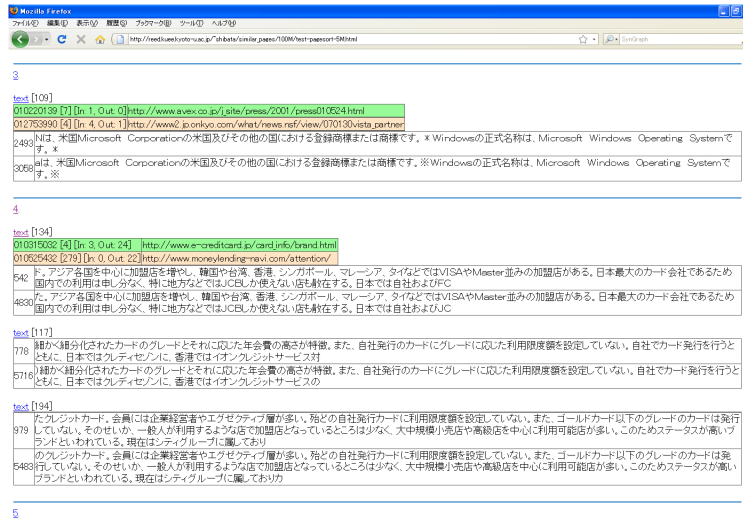

図 12: 類似ページの例 (各類似ページペアに対して，そのページペアに含まれる類似文字列と両ページの URL を表示している)

64 ノードを用い，ノードあたり 14CPU コアを利用したところ，4-a,4-b のどちらのタイプのジョブも約 1 時間半で終了した．図 13 に並列計算が進行する様子を示す．図中の横軸は経過時間 (秒)，縦軸は走っているタスク数を表す．1bin のサイズは約 300M(1chunk は gzip で圧縮して約 5.6G) であり，1 ジョブあたりの出力は bzip2 で圧縮して約 300M であった．

1 億ページの解析を行なうには 400 ジョブを走らせる必要があるが，そのうちの 20 ジョブを走らせることにより，2,000 万ページの解析を行なった[6]．

4.4.2 類似ページの分類

2,000 万ページの処理結果のうち 1,000 万ページにおいて，検出した類似文字列をもとに，1,000 万ページ中に含まれる類似ページの自動分類を行なった．出力された類似文字列のペアをページペア単位で集計すると，約 325 億の類似ページペアが得られた．

まず，あらゆるページペアにおいて，各ページの類似文字列重複率 (類似文字列とみなされた文字数/全文字数) を計算し，2 つのページの重複率をプロットしたものを図 14 に示す．重複率によって以下のように分類することができる．

ちょうど 1 2つのページが同一である．この中には URLが類似したものだけでなく，定型ページ，IP アドレスとドメイン名・ホスト名のアドレスのページペア (例: `http://219.166.24.90/dainichi/rensai/furusato/furusato040728.html` と `http://ns.nnn.co.jp/dainichi/`

[6]東京工業大学の TSUBAME も併用して解析を行なった．

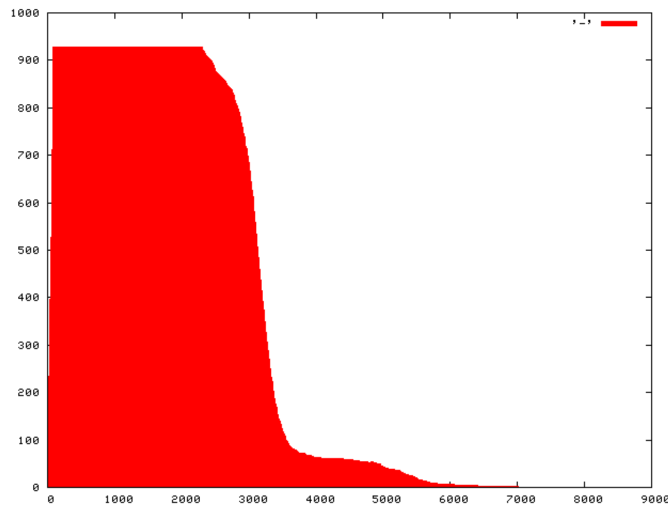

図 13: 並列計算における経過時間と走っているタスク数の関係

rensai/furusato/furusato040728.html)，ドメイン名が類似していないミラーページ (例: http://www.caj.co.jp/focus/security/spyware_list.htm と http://casupport.jp/focus/security/spyware_list.htm) が含まれていた．

1 より少し小さい ブログの「トップページ」と「月別アーカイブ」などといった包含ページであり，例えば，http://kroko.maxs.jp/~kroko/mt/archives/003687.shtml と http://kroko.maxs.jp/~kroko/mt/archives/003802.shtml などがあった．

0 より少し大きい 部分共有ページであり，ページの一部に定型文が含まれているものが多数あった．

重複率が 0.25 から 0.75 の場合は様々なページタイプが含まれていたが，標準フォーマットにうめこまれているインリンク，アウトリンクの情報を用いて，2 つのページがリンク関係にあるもののみに限定したところ，様々なページにリンクしているページ，様々なページからリンクされているページが得られ，以下のようなものであった．

様々なページにリンクしているページ いろいろなページをつなぎあわせたようなページ (図 15) などが得られた．いろいろなページをつなぎあわせたようなページは一種のスパムページであり有用なページではないので，今後，検索エンジンのインデックスから除外する予定である．

様々なページからリンクされているページ SEO 対策で様々なページ間でリンクしあっているものやトラックバックがはられているものがあった．SEO 対策で様々なページ間でリンクしあっているものは上記のスパムページと同様，検索エンジンのインデックスから除外する予定である．

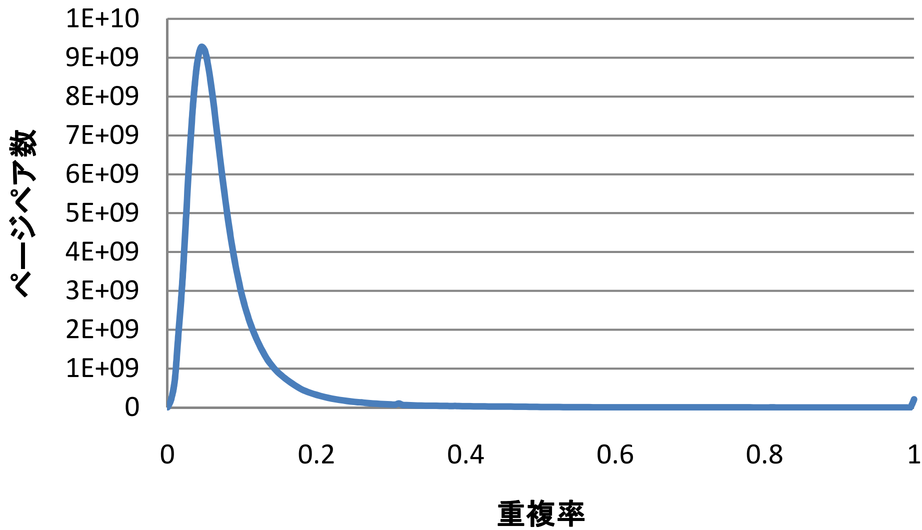

図 14: 重複率とページペア数

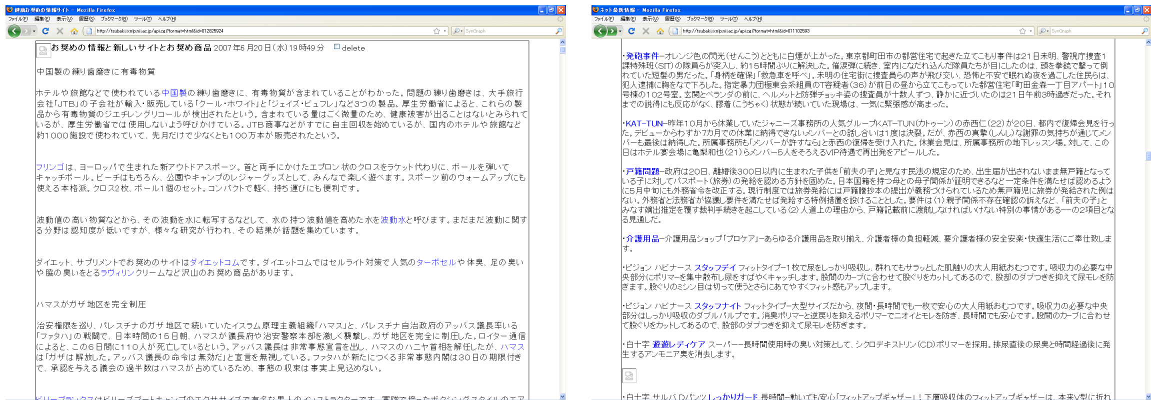

図 15: いろいろなページをつなぎあわせただけのページの例

4.5 まとめ

本研究では，大規模ウェブページコレクションを対象として，あらゆるページのペア間に含まれる類似文字列の検出を行ない，その結果から，同一・包含・部分共有関係にあるページなどを同定した．

今後の課題としては 1 億ページに対して処理を完了することや，類似ページ検出結果を検索エンジン基盤 TSUBAKI に反映させることなどがあげられる．

5　おわりに

本プロジェクトで実施したデータ集約的タスク, タスク間のデータの受け渡しをファイルを介して行うタスクは, もともとファイルシステムのスケーラビリティに対する要求, 負荷が大きく, チャレンジングなタスクである. また, 多くの逐次プログラムを, 少ない統合の手間で組み合わせて並列処理を行うワークフローシステムは, 統合の仕方を変更しては実行を繰り返す, 生産性に対する要求が大きいタスクで, その意味からもソフトウェア開発・実行環境にとってチャレンジングな課題を提供する.

本プロジェクトでは, 総じて, 500-1,000 並列程度の規模に対して, 複雑なワークフローを慣れた記述 (Makefile) で生産性高く実行できることを実証できた. また, それ以上の規模に対する実験の機会を与えていただいたことは, 有益な経験であり感謝したい. 今後も機会あるごとに, ぜひ 10,000 並列レベルのワークフローを簡単に, 効果的に並列化できる枠組みとして, 研究と実証実験を続けて行きたい. また, GXP を, 並列処理を手軽に実行できる仕組みとして, HA8000 ユーザの日常的ツールとして提供して行きたいと考えている.

参考文献

[1] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, pp. 141–150, 2007.

[2] Yusuke Miyao, Tomoko Ohta, Katsuya Masuda, Yoshimasa Tsuruoka, Kazuhiro Yoshida, Takashi Ninomiya, and Jun'ichi Tsujii. Semantic retrieval for the accurate identification of relational concepts in massive textbases. In *the Proceedings of COLING-ACL 2006*, pp. 1017–1024, July 2006.

[3] Takashi Ninomiya, Takuya Matsuzaki, Yoshimasa Tsuruoka, Yusuke Miyao, and Jun'ichi Tsujii. Extremely lexicalized models for accurate and fast hpsg parsing. In *the proceedings of EMNLP*, pp. 155–163, 2006.

[4] Keiji Shinzato, Daisuke Kawahara, Chikara Hashimoto, and Sadao Kurohashi. A large-scale web data collection as a natural language processing infrastructure. In *Proceedings of the 6th International Conference on Language Resources and Evaluation (LREC08)*, 2008.

[5] Keiji Shinzato, Tomohide Shibata, Daisuke Kawahara, Chikara Hashimoto, and Sadao Kurohashi. TSUBAKI: An open search engine infrastructure for developing new infor-

mation access methodology. In *Proceedings of Third International Joint Conference on Natural Language Processing (IJCNLP2008)*, pp. 189–196, 2008.

[6] Kenjiro Taura. Gxp: An interactive shell for the grid environment. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 59–67, 2004.