修士論文

広域分散環境での効率的なデータ集約的アプリケーション実行システム

An Efficient Execution System of Data-Intensive Applications in Wide-Area and Distributed Environments

平成25年2月6日提出

指導教員　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-116457 堀内 美希

概 要

マルチプロセッサ環境，広帯域なスループットを達成するネットワークの普及により，並列計算はより身近なものとなりつつある．中でも近年では，大量のデータを処理し，解析を行う大規模データ集約的アプリケーションが至るところで用いられるようになった．アプリケーションが計算中に扱うデータ量は益々増大し，有限の資源を用いてデータを効率的に扱うシステムへの要請が高まっている．特に，資源が複数拠点にまたがった状態で計算を行うと，非効率なデータ転送により性能低下が顕著となる．

本研究では，データ集約的アプリケーションを分散環境で簡単に行うためのシステムを提案する．提案システムは，既存のワークフロー実行エンジン GXP Make と，本論文で提案する分散ファイルシステム Mogami により成る．ユーザがデータ集約的ワークフローアプリケーションを，手軽に実行，開発可能なシステムを目指している．なかでも，本研究で提案するシステムは，(1) 各ジョブ間のデータの流れ（各ジョブのノード間で共有するファイルへの読み込み，書き込み）をプログラマが記述せずとも，一度のプロファイルにより自動でノード間データ転送を効率化する，(2) 高遅延環境のネットワークを介したノード間データ転送でも，アクセスパターンを自動検知して先読みを行うことでスループットを低下させない，といった特徴を有する．

評価では，各提案手法に対するベンチマークによる評価と，実際に実用化されている 5 つのワークフローアプリケーションを用いて実験を行った．評価の結果，提案手法により Montage などのファイルアクセス比率が高いアプリケーションでは，アプリケーション全体実行時間を 50%以下に短縮するなど，提案システムの有用性を示すことができた．

目 次

図目次

表目次

第1章　序論

1.1　背景：大規模データ集約的コンピューティング

マルチコアアーキテクチャの普及，分散コンピューティング技術の進歩に伴い，並列計算が様々な箇所で用いられている．中でも近年は，大規模なデータを扱い，解析を行う大規模データ集約的コンピューティングが頻繁に行われるようになった．アプリケーションが扱うデータのサイズは肥大化し，用いられる分野は科学技術計算のみならず機械学習，大規模テキスト処理など多岐に渡っている [25]．例えば，Google 社においては大規模文書のインデクシングや，大量のウェブ上に存在するデータのクロールが行われており，大量の顧客データ管理と共に様々なアプリケーションが運用されている．Apache Software Foundation の開発した Hadoop[48] は，MapReduce[17] フレームワークの実行システムとして，企業で使われる事例が増えている．日本国内でも，例えば楽天ではユーザへのレコメンド商品の算出，NTT データでは全国の渋滞情報の可視化に Hadoop が用いられ，大量のデータが大規模環境で処理されている．科学技術計算では，大量の天文画像を組み合わせて 1 つのモザイクイメージを作成する Montage[5] や，大量の医学書文書を解析する MEDLINE to MEDIE[16] などが広く用いられている．

ムーアの法則の速度を越えて増大するデータを効率的に，かつ現実的な時間で処理することは，今後のデータ中心社会の課題である．また，アプリケーション開発者の視点からみれば，最低限の負担で簡単に分散環境を用いた並列計算プログラミングが行えるシステム，フレームワークが存在することが望ましい．近年は，並列計算のためにスーパーコンピュータを用いずとも，汎用コンピュータで構成される PC クラスタを用いたグリッドコンピューティング，企業などにより提供されるリソースをオンデマンドに利用できるクラウドコンピューティングが容易に行える．このように分散計算を行うための資源も多様化しており，それらをユーザが少ない労力で扱い，計算の効率化を行うための仕組みが必要である．本研究では，これらの要請を踏まえ，並列アプリケーション実行システム GXP Make を用い，データ集約的コンピューティング中のノード間ファイル共有を担う分散ファイルシステム Mogami の設計・実装を行う．また，これらのシステムを用いデータ集約的計算中のデータ転送効率化手法を提案・実装・評価する．

以降では，本研究で提案するシステムのターゲットとするアプリケーション，計算中のデータ管理手法などの背景について述べる．

1.2　ワークフローアプリケーションとその実行エンジン

分散環境で実行する大規模アプリケーションは，互いに依存関係を持つ細粒度タスク（本論文では “ジョブ” と呼ぶ）の組み合わせにより実現されることが多い．このようなアプリケーションを “ワークフローアプリケーション” と呼ぶ．ワークフローアプリケーションは，図 1.1 に示す DAG

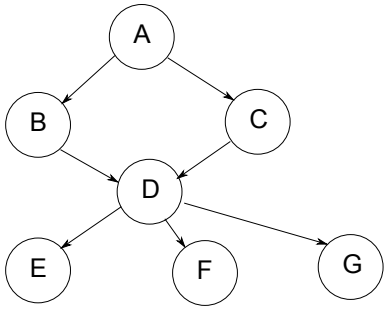

図 1.1. Directed Acyclic Graph (DAG)

表 1.1. ワークフローアプリケーション実行時のファイル共有手法分類とその特徴.

	Pre-staging	On-demand
Pros	ワークフロースケジューリングの最適化がしやすい 高遅延環境でも遅延の影響を受けにくい通信	アプリケーションの変更は必要なし ファイルの必要とされた部分のみ転送
Cons	Input/Output ファイルの指定が必要 必要とされていなくともファイル全体を転送	ワークフロースケジューリングの最適化が難しい 高遅延環境では遅延の影響を受けスループットの低下

（無閉路有向グラフ[1]）の形で各ジョブの依存関係を表現することができる．グラフ中の各ノードはジョブを表し，エッジはジョブ間の依存関係を表している．

このワークフローアプリケーションを分散計算資源を用いて実行するシステムとして，ワークフロー実行エンジン Pegasus[19], Swift[52], GXP Make[43], Pwrake[40], Dryad[26] などがある．これらの実行エンジンを用いる際，ユーザはジョブの依存関係をなんらかの形で記述する必要がある．そしてその依存関係に基づき，依存関係の解決したジョブのみ分散環境で自動並列で実行する．これらのワークフロー実行システムを用いると，ユーザは複数の逐次実行用プログラムと，それらの依存関係を記述したファイルを用意するだけで，容易に分散環境を用いた並列計算を行える．

1.3　ワークフローアプリケーション実行時のデータ管理手法

ワークフローアプリケーションでは，実行される各ジョブはプロセスとして扱われる．プロセス間通信の手段としてファイルに書き出すということは，パイプを用いることと並んで一般的な手法であり，ワークフローアプリケーションでも典型的に用いられている．また，ファイルへの書き出しはパイプを用いるのに比べて分散計算との相性も良い．そのため，ワークフローアプリケーション実行時には，分散環境でファイルを共有するためのシステムが必要となる．この手法は大きく分けて *Pre-staging* と *On-demand* の二種類に分類することができる [13]．それぞれの手法について以下に述べる．

[1]厳密にはある処理の loop など，閉路が存在することもある

- Pre-staging
  ジョブをディスパッチし，実行する前にそのジョブの Input ファイルを持っているノードからファイルコンテンツを予め複製し，その完了を待ってからジョブを実行する．
- On-demand
  アプリケーションがファイルアクセスを行ったときに，行った分だけネットワークを通じてデータ転送が行われ，ファイルアクセスが可能となる．分散ファイルシステム，またはそのようなファイル共有システムを使うことを基本としている [27].

これら手法にはそれぞれ長所短所がある．それぞれの長所短所を表 1.1 にまとめる．

Pre-staging なデータ転送手法では，ユーザが各ジョブの Input/Output ファイルを全て明示的に指定する必要がある．また，ファイル全体の転送が必要となり，アプリケーションが必要としない部分も事前にデータ転送する．しかし，各ジョブの Input/Output ファイルがどれであるか（その内どこを読むかまでは分からない），全て事前に判明しているため，アプリケーション実行時にデータ転送の最適化を綿密に行うことができる．対して On-demand なデータ転送手法では，通常 NFS[37], Lustre[4], GPFS[36], Ceph[47], PVFS[11] などの分散ストレージをバックエンドに，ファイル共有を行う．この場合には，基本的に各ジョブのアプリケーションを変更する必要はなく[2]，ユーザは各ジョブの Input/Output ファイルを記述する必要もない．しかし，逆にアプリケーションからファイルアクセス要求があるまで，システム側からはどのファイルが必要とされるか分からないため，データ転送の最適化が難しい．

本研究では，ユーザの負担を最小限に抑えるため，分散ファイルシステムによる On-demand なデータ転送，ファイル共有手法を用いる．分散ファイルシステムを用いることで，アプリケーションから共有ファイルに，ローカルファイルと同じ通常の API を用いたアクセスができる．これらの長所を活かしつつ，短所となる性能改善の難しさを取り除くための手法を提案する．

1.4　ジョブスケジューリングの最適化とユーザの負担

ワークフローアプリケーションの実行時は，依存関係の解決したジョブを計算資源として使用可能なノードにディスパッチする必要がある．これらの作業は通常ワークフロー実行エンジンが自動で行い，ユーザが手動で行う必要はない．データ集約的アプリケーションの場合，実行するノードの選出によっては大量の入力データを転送する必要があるため，データ転送の効率化を目指したジョブスケジューリングを考えることは性能向上のため必須である．しかし，ディスパッチするジョブが読み書きするファイルは，一般に実行するまで判明しない．プログラムによっては，プログラムの実行過程で読み書きを行うファイルを決定する場合も存在する．

ジョブをディスパッチするノードはジョブの実行前に選出する必要があるため，データ転送効率化を考慮したジョブスケジューリングのためには，最も簡単にはユーザが全てのジョブの Input/Output ファイルを記述する必要がある．これは，ワークフローアプリケーションの開発者に負担を強いることとなる．ワークフローアプリケーション開発者は，時には他人の記述したプログラムを使い回す場合もある．また，大量のファイルを読み書きするジョブに対しても全ての Input/Output ファイルを記述する必要がある．例えば Montage[5] アプリケーションのあるジョブは，入力ファイルだけで 2700 以上のファイルが存在する．これらのケースでアクセス対象のファイルを人間の

[2] POSIX 互換のファイルシステムを用いる場合

図 1.2. 高遅延環境（RTT: 13.9msec）でのファイルデータ転送スループット

手で全て列挙するのは非現実的である．そこで，本研究ではユーザにそのような負担を強いることなく，一度アプリケーションを事前実行し，各ジョブがどのようなファイルにアクセスするかのデータを取るだけで，システムが自動でデータ転送を効率化するジョブスケジューリングを行うための手法を提案する．

1.5　高遅延環境での分散ファイルシステムによるデータ転送

広域環境，特に複数拠点にまたがった環境でワークフローアプリケーションを実行する際，高遅延ネットワークで接続されたノード間でのファイルデータ転送が起こる場合がある．このような環境で分散ファイルシステムを基盤としたデータ転送を行うと，遅延の影響を受け本来達成できるはずの物理スループットを遥かに下回るスループットしか達成できない場合がある．例えば，北海道大学-東京大学間 10Gbps 環境で sshfs[7] を用い，1GB のファイルを `cat` コマンドにより読んだところ，全てのファイル内容の読み出しに約 2 分程かかり，データの平均転送速度は約 70Mbps であった（図 1.2）．全く同じ環境で，`iperf` による通信では，約 600Mbps のスループットを達成している．Gfarm 分散ファイルシステム [31] による通信では，約 130Mbps のスループットを達成しているが，それでも `iperf` のものには遠く及ばない．分散ファイルシステムでは通常ファイルデータを，ある一定の大きさ（“ブロックサイズ” と呼ぶ）で管理・転送している．そのため，ファイルデータを転送する際にはブロック毎に転送要求と転送が繰り返し起こり，高遅延環境ではネットワークスループットを大きく下回る読み込みスループットしか達成できない場合がある．本研究では，データの先読みを適応的に行い，この問題を回避する．また，データ先読みを行う際は，先読みが外れるリスクも考えてネットワーク性能を限界まで引き出せる最小の大きさで先読みを行うのが望ましいが，その大きさは環境に応じて変化する．本研究では，この大きさの自動調整手法を提案する．

1.6　本研究の全体像

以上の背景の下，本研究では分散環境でデータ集約的ワークフローアプリケーション実行を行うためのシステムと，そのデータ転送最適化手法を提案する．提案するシステムは以下の特徴を

有する．

- ユーザはワークフローアプリケーションの各ジョブ用実行プログラムとその依存関係を記述するだけで，分散環境に確保した資源を用いて並列計算を行うことができる
- ノード間でのファイル共有には POSIX 互換の分散ファイルシステムを用い，アプリケーションから共有ファイルへは透過的なアクセスが行える．そのため，アプリケーションの開発者は既存の逐次プログラムを書き換えることなくワークフローアプリケーションでジョブ用プログラムとして用いることができる
- 各ジョブ間のデータの流れ（各ジョブの共有ファイルへの読み書き）をプログラマが記述することなく，自動でノード間データ転送の効率化を行う（第 4 章）
- ファイルデータ転送の際，高遅延環境を介した通信が発生した場合にも，アクセスパターンを検出して適応的な先読みを自動で行う（第 5 章）

これらの特徴により，アプリケーション開発者の負担を増大させることなく，データ集約的計算を効率的に分散計算資源を用いて行うための基盤システムとして，ワークフロー実行エンジン GXP Make と分散ファイルシステム Mogami を提案する．

1.7　本稿の構成

以降，本稿は以下の通り構成されている．

第 2 章では，本研究の関連研究を紹介し，第 3 章で，本研究で提案するワークフロー実行システム概要を述べる．次に，第 4 章でファイルアクセス履歴を用いたジョブスケジューリングとデータ転送最適化手法を提案し，第 5 章で高遅延環境におけるデータ転送効率化のための適応的先読み手法を提案する．最後に第 6 章で提案システムの評価を行い，第 8 章で本稿をまとめる．

第2章　関連研究

本章では，本研究に関連する研究を紹介する．本論文では，

1. ワークフローアプリケーションのデータローカリティを向上させるためのスケジューリング手法を目指す
2. 分散ファイルシステムを用いた高遅延ネットワークのデータ転送を先読みにより効率化する

以上 2 点に基づいた効率的なワークフローアプリケーション実行システムを提案する．以降ではそれぞれに関連する研究を取り上げ，本研究の位置づけを明確にする．

2.1　データローカリティを高めたデータ集約的並列計算

2.1.1　ワークフローのジョブスケジューリングとデータ位置

データ集約的ワークフローアプリケーションの高速化のために，計算中のデータを効率的に管理，処理することを目指す研究は多数行われている [18, 21]．Kosar らは，計算リソースとして CPU を重視したジョブスケジューリングを行う，従来のジョブスケジューリング手法の限界を指摘し，新しいデータ集約的コンピューティングに合致したスケジューリング手法を提案した（データアウェアスケジューラ）[30]．彼らはケーススタディとして，データ配置スケジューラ Stork を紹介し，評価を行っている．

分散ストレージによるレプリケーションをジョブスケジューリングと組み合わせて行い，ワークフローアプリケーションの効率化を図る研究 [15, 20, 12] は数多く行われており，これらも計

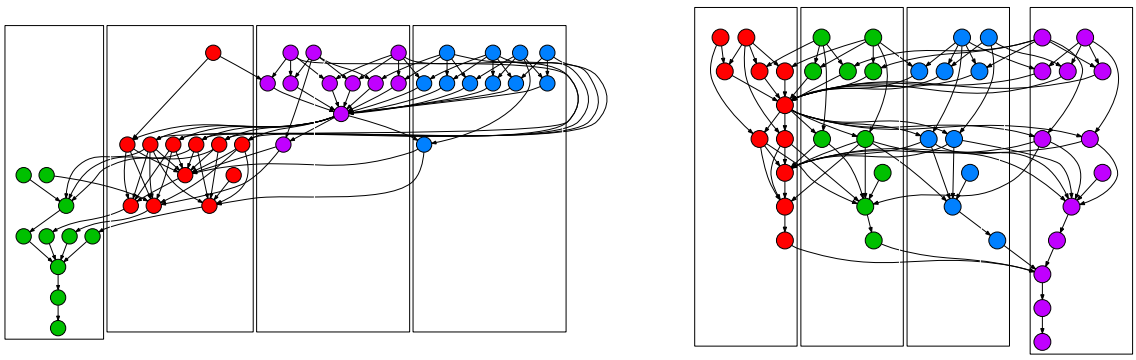

図 2.1. 標準グラフ分割を用いた既存ジョブスケジューリング手法（左）と MCGP を用いたジョブスケジューリング手法（右）[41]

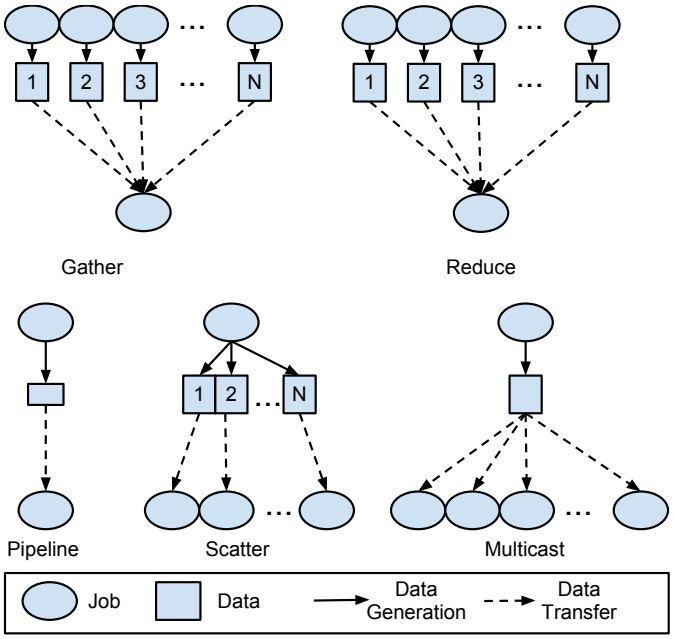

図 2.2. ワークフローアプリケーションの典型的データ入出力パターン [51]

算時のデータローカリティを高めるために行われている．また，田中らは Multi-Constraint Graph Partitioning[28] を用いたワークフローのスケジューリングを行い，ノードやクラスタ間をまたがったデータ転送を減らすジョブスケジューリング手法を提案している [41]（図 2.1）.

ワークフローアプリケーションの入出力のパターン・傾向はよく調査されている [50, 49]．これらの調査に基づき，Chakrabati らはワークフローアプリケーションのデータ入出力パターンを `Gather`, `Reduce`, `Pipeline`, `Scatter`, `Multicast` の 5 種類（図 2.2）に分類し，それぞれについてデータ転送の最適化を行った [46]．分散ストレージ MosaStore[6] にこのデータ転送の最適化を施し，その効果を実験により得た．しかし，この実験は，期待される性能を観測するためのものに過ぎず，入出力パターンはユーザが直接指定し設定を変更する必要がある．またいくつかの最適化は，原理上同時に行うことができないものもある．それに対して Zhang らは，同様の 5 種類のデータ入出力パターンの自動判定手法，特に集約的パターンに対してのデータ転送最適化を行った [51].

上記の研究は，ワークフローアプリケーション実行時のデータ転送を減らし，データローカリティを高めることを目指しており，本研究と類似している．しかし，上記既存研究では，各ジョブのアクセスファイルはユーザにより事前に設定ファイルに明記されており，ジョブディスパッチの時点で確定していることが前提になっている．本研究ではユーザにその手間をかけさせることなく，自動で各ジョブのアクセスファイルを予測し，効率的なデータ転送を行うことを目指す.

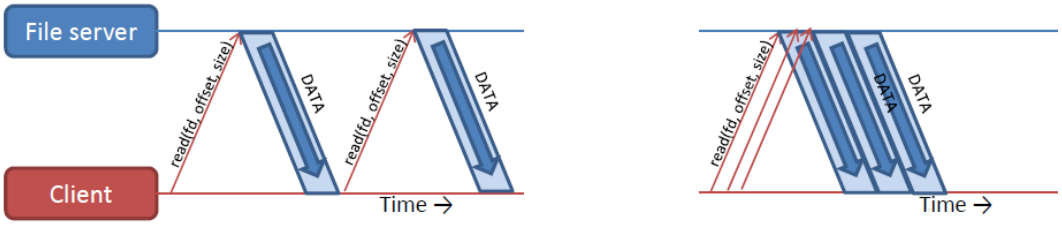

図 2.3. Non-blocking RPC を用いた遠隔ファイルアクセスの高速化 [54]

2.1.2　データローカリティを担保する並列計算フレームワーク

近年では，MapReduce[17] やその実装である Hadoop[48] が，研究や企業で広く用いられている．Hadoop が主流のデータプロセシングフレームワークとなり得たのは，良好なスケーラビリティを持つとされることが理由として挙げられる．Hadoop はこのスケーラビリティを，データ処理のフレームワークを限定し，計算タスクを入力データを持つノードに割り当てやすいものとすることで達成している．Hadoop では HDFS[39] と呼ばれる分散ファイルシステムに対しデータの読み込み，書き込みを行っている．

また，Gu らは Sector 分散ファイルシステムと，それに付随したプログラミングフレームワーク Sphere を提案している [24]．このプログラミングフレームワークでは，ユーザはデータに対して行う処理を UDF (User-Defined Funcation) で記述し，Sector で管理しているデータセグメントに対して独立にその処理を適応する．

これらのフレームワークは適応できる処理を限定することで，処理データが存在するノードでの処理を行うことができ，結果的に良好なスケーラビリティを達成している．しかし，複雑な処理，分岐を行うアプリケーションだと，このフレームワークには当てはまらず，計算が困難な場合もある．

2.2　高遅延環境を介すファイルデータアクセス

2.2.1　広域分散ファイルシステム

高遅延環境でのファイルデータアクセスには，しばしば広域分散ファイルシステムが用いられる．Hadoop でのデータ共有基盤システムには HDFS (Hadoop Distributed File System)[39] が用いられている．また，Gfarm[31] は広域分散環境での使用を前提に設計された分散ファイルシステムで，これらの分散ファイルシステムではレプリケーションを行い，一番遅延の小さい場所のレプリカにアクセスする等，高遅延環境を介したファイルアクセスを削減する試みがなされている．

また大辻らは，高遅延ネットワークを介した遠隔ファイルアクセスを，Non-blocking RPC を用いて高速化する手法を提案している [54]（図 2.3）．これは，本研究で提案する先読みによる高速化とほぼ同じ原理であるが，Non-blocking RPC を送っておく量を自動調整する手法は提案されておらず，ユーザが手動で設定を行わなければいけない．また，手動で設定したとしても，ネットワーク状況の変化などには対応できない．

2.2.2　アプリケーションによるプリフェッチ

GPFS[36] では MPI-IO によるデータ先読みを行いファイルアクセススループットを向上させる研究が行われている [33]．また，MPI プログラムのコードを解析し，ファイルアクセスに関する部分を別スレッドの処理へ切り出し，先読みを実現する研究も行われている [14]．評価には，PVFS[11] を用いている．

これらの研究は分散計算中のファイルデータ先読みという点で本研究と共通する部分があるが，いずれの場合も，先読みは並列アプリケーションそのものを変更することでアプリケーションが行うものである．先読みを行う場所，サイズはアプリケーション内でユーザにより事前に決定されなければならない．

2.2.3　高遅延環境でのファイルデータ転送手法

高遅延環境での大容量のファイルデータを効率的に転送するためのプロトコルとして，GridFTP[9] の研究が行われている．GridFTP では，TCP ソケットバッファサイズの自動調整，複数ソケットを用いたデータの並列転送等が自動で行われる．並列コネクション数の調整を自動で行うための手法も提案されている [53]．高遅延環境での既存の TCP を用いた通信の問題点を改善するという点で，本研究と類似しているが，しかしこれは通常の FTP と同様ファイル全体の転送を目的としている．本研究ではファイルシステム API をサポートし，オンデマンドにデータ転送を行う分散ファイルシステムを提案する．

第3章　提案ワークフローアプリケーション実行システム

本研究では，ワークフローアプリケーションを実行するためのシステムとして，ワークフロー実行エンジン GXP Make と分散ファイルシステム Mogami を用いる．本章では，これら 2 つのシステムについて，それぞれ述べる．

3.1　GXP Make: ワークフローアプリケーション実行エンジン

GXP Make[43] は，分散したリソース（クラスタなど）でワークフローを分散実行する，ワークフロー実行エンジンである．以降に全体のシステム概要と，ワークフローのジョブスケジューリング手法に関して述べる．

3.1.1　概要

GXP Make はワークフローアプリケーションの分散実行の際に，GXP Shell[42] と GNU Make を利用している．

GXP Shell は，ssh や torque[8] を通してリモート環境でコマンドを実行できるノードをワーカノードとして確保し，マスターノードから効率的に他のノードを扱うためのシステムである．GXP のプロセス構成を図 3.1 に示す．このように GXP には 1 台のマスターノードが存在し，残りのノードはワーカノードとしてマスターノードからジョブ（コマンド）を受け取り実行する仕組みになっている．GXP ではマスターノードから全てのワーカノードへ直接 ssh ログインなどが可能な場合はもちろん，NAT 環境などでも内部で木構造を作り，動作する．また，GXP や GXP Make は Python で実装されており，マスターノードのみソースコードをダウンロードすれば，残りのワー

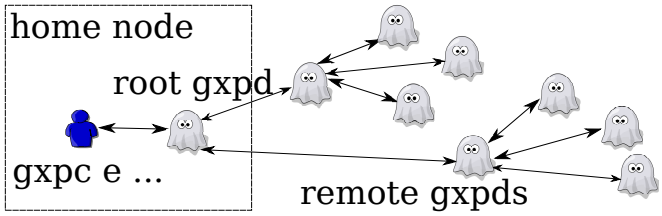

図 3.1. GXP のプロセス構成 [43]

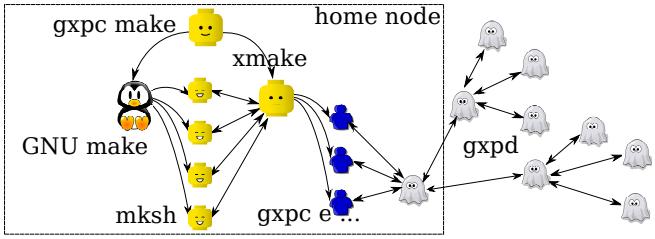

図 3.2. GXP Make の仕組み [43]

カノードに自動でソースコードを分散する．そのためインストールは非常に簡単なものとなっている．

GXP Make でワークフローアプリケーションを記述する際には，実行可能なプログラムを用意し[1]，各ジョブのコマンドと依存関係を GNU Make の様式（`Makefile` と呼ばれるもの）に従って記述する．`Makefile` は元々ソフトウェアのビルドなど広く用いられており，多くのプログラマにとっては既に馴染みのあるもので学習コストが低い．また，GNU Make はパターンルールをサポートしており，大量のルールを一つの記述で記載することができる．この特徴は，特にワークフローアプリケーションを構築するユーザにとって非常に扱いやすいものとなっている．加えて，再帰的に Makefile を呼び出す（再帰 Make と呼ばれる）などで静的なワークフローだけでなく，動的に変わるワークフローを記述することも可能である．

ワークフローの実行には，カレントディレクトリに Makefile を置いた状態で，

```
$ gxpc make -j 200
```

などとすると，確保してある計算資源を用いて自動的にジョブを依存関係を考慮しつつ並列実行が開始される．ここで`-j` は GNU Make に渡されるオプションで，ジョブを最大 200 個まで同時並列することを示す．ユーザは，このオプションによりジョブの並列度を指定することができる．

図 3.2 に，GXP Make 実行時のプロセス関係を示す．ユーザが書いた Makefile は通常の GNU Make のプロセスにより解釈され，`xmake` に実行コマンドが送られる．`xmake` は，それらのコマンドを GXP Shell を用いてワーカノードで実行する．

GXP Make には，ワークフロー実行時のファイル共有のための仕組みは存在しておらず，NFS[37] や Lustre[4]，GPFS[36] といったネットワークファイルシステムを基盤とすることが想定されている．本研究では，ワークフローアプリケーション実行時のデータ共有のための分散ファイルシステム Mogami を提案し，評価などに用いる．

3.1.2　GXP Make のジョブスケジューリング手法

GXP Make では，基本的にジョブをディスパッチする際には，計算リソースが十分に利用可能な状態かどうかのみを考慮する．ユーザがジョブを実行するワーカノードをコマンドにより直接

[1] もちろん unix コマンドなども使用可能

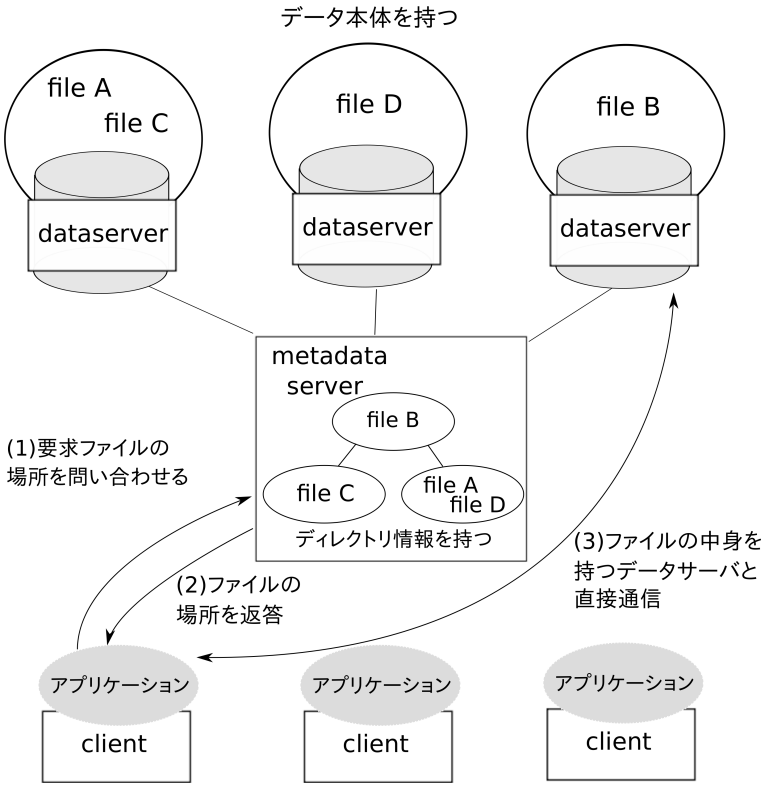

図 3.3. Mogami のシステムコンポーネント

指定することは可能であるものの，デフォルトではほぼランダムにジョブが割り振られる．そのため，ファイルに大容量のデータを読み書きするアプリケーションでは，ファイルデータのネットワークを介した転送が過剰に行われるようなジョブアロケーションになる可能性がある．本研究では，これを改善し，データ転送を可能な限り行わなくて済むジョブスケジューリングをするよう，GXP Make のジョブスケジューリング部分を改良する．

3.2　Mogami: 分散ファイルシステム

次に，ワークフローアプリケーション実行時にノード間でファイル共有するために用いる，Mogami 分散ファイルシステムの詳細に関して述べる．

3.2.1　概要

ワークフローアプリケーション実行基盤としての扱いやすい分散ファイルシステムとしては，インストールが簡単で，必要なときに管理者権限なしでマウント可能であることが要請の 1 つと考えられている [44]．Mogami は GXP Make と同じく Python で開発されており，FUSE[1], python-fuse[2] という一般的なソフトウェアさえインストールされていれば，管理者権限なしで簡単にマウント可能である．

Mogami にはクライアント，メタデータサーバ，データサーバの 3 つの主要コンポーネントが存在する．これらの 3 つのコンポーネントの働きを図 3.3 に図示し，以下に述べる．

- クライアント
  Mogami を POSIX 互換のファイルシステムとしてマウントするクライアント．マウントした PATH 以下に通常のローカルファイルと同様の手順で，分散ファイルシステム上にファイルを作成，読み込み，書き込みができる．
- メタデータサーバ
  分散ファイルシステム上に存在する全てのファイルのメタデータ管理を行う．システム内に一台のみしか存在しない．メタデータの管理は，ローカルファイルを用いる方法と SQLite3 データベースを用いる方法と 2 種類あり，ユーザにより設定可能である．クライアントからファイルの open 要求があった際には，そのファイルのコンテンツデータを保持するデータサーバの位置を，クライアントに通知する．メタデータ照会・変更要求がクライアントからあった際には，データサーバを介さずに，クライアントとメタデータサーバのやり取りのみで処理が完結する．
- データサーバ
  ファイルのコンテンツデータを保持するノード．クライアントからデータ読み出し要求があった際に要求ファイルデータをクライアントに送信し，その逆としてクライアントからデータ書き込み要求があった際に書き込みデータを受信し，ファイルに書き込む．

Mogami では，多くの一般的な分散ファイルシステムに見られるように，ファイルデータをあるサイズのブロックに分けて管理し，データサーバからのクライアントへのデータ転送の際には，ブロック毎の転送が行われる[2]．

Mogami は POSIX 互換の API を備えていることから，ワークフローアプリケーションの開発者は，ファイルシステム上のファイルを読み書きするアプリケーションを，ワークフロー構築に修正なしで用いることができる．

3.2.2　データを配置するノードの選出

Mogami では，ファイルの新規作成時に，自身がデータサーバでもある場合は自身に新規ファイルのコンテンツを作成する．これにより，ファイルを作成し，`write` オペレーションを行うときのローカリティを担保できる．自身がデータサーバでない場合には，全データサーバの中からランダムにノードが一台選出され，選出されたデータノードに新規ファイルコンテンツを配置する仕組みになっている．

3.2.3　レプリケーションとコンシステンシモデル

分散ファイルシステムでは，ネットワーク遅延による影響を防ぐため様々な処理を非同期に行うため，採用するコンシステンシモデルがよく議論される．Mogami では，NFS や Gfarm などが採用している **Close-to-open コンシステンシモデル**というセマンティクスに基づき，ファイルを管

[2] ブロックサイズはユーザが設定．デフォルトでは 1MB

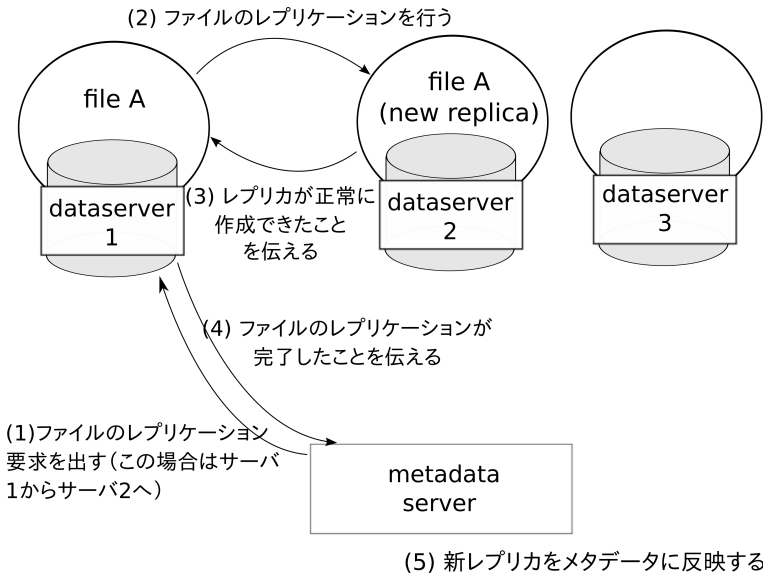

図 3.4. レプリケーションの流れ

理する．このセマンティクスは，更新されたファイルの内容が他のプロセスから参照されるのは，そのファイルを close した後というものである．また，Mogami はファイルを更新した当のプロセスに対しては，即座にその変更を反映しているように見えることを保証する．ワークフローアプリケーションの実行には，ジョブの依存関係によって，あるジョブがファイルを close した後にあるジョブが実行されるといった順序を保証することが可能である．そのため，ワークフローアプリケーションの実行には Close-to-open コンシステンシモデルで十分であると考え，採用する．

Mogami には，ファイルコンテンツのレプリケーションの機能が存在する．レプリケーションは，基本的にメタデータサーバを介して行う仕組みになっており，その流れを図 3.4 に示す．メタデータサーバが元々データを持っているノードにレプリケーション要求を出し（図中 (1)），要求を受け取ったデータサーバによりレプリケーションが行われる（図中 (2), (3)）．その後メタデータサーバに対してレプリケーションの完了通知が行われ（図中 (4)），メタデータサーバ上のメタデータが更新される（図中 (5)）．ここで作成されたレプリカは，その後ファイルの更新が行われた場合，更新後ファイルを閉じた時点で使用不可能（折を見て削除する）とすることで，Mogami の意図する Close-to-open コンシステンシが保たれる．

第4章　提案手法I：ファイルアクセス履歴に基づくジョブスケジューリングとデータ転送最適化

本章では，ファイルアクセス履歴に基づいたワークフローアプリケーションスケジューリング改良手法に関して述べる．

4.1　概要

一般に大量のデータを扱うワークフローアプリケーションの実行には，データローカリティを考慮したジョブ配置を行うことが望ましい．しかし，そのためにはジョブをディスパッチするシステムが，ディスパッチするジョブがどのようなファイルアクセスを行うのか事前に得ておく必要がある．このためにユーザに実行する全ジョブのファイルアクセスを，予め設定ファイルに記述することを強いるスケジューラも少なくない．しかし，大量のファイルを読み書きするアプリケーションも多い中，全てのファイルアクセスを記述するのはユーザにとって重荷である．ワークフローアプリケーションに限らず，アプリケーションのアクセスするファイルをユーザからのアノテーションなしに決定することは容易ではない．しかし，ワークフローアプリケーションでは，

- Input データを変更する
- 実行時のパラメタを変更する

などの各ジョブのためのプログラムを使い回した実行を行うものが多い．そのため，同じプログラムを使っているもののファイルアクセスは何かしらの傾向があることが考えられる．

提案手法の流れを図 4.1 に示す．本研究では，予めアプリケーションを一度事前実行してファイルアクセス履歴をプロファイルし（図 4.1 中 (1)），ワークフロー内で用いられるプログラムのアクセス傾向をプロファイル結果から得る（図 4.1 中 (2)）．次回のアプリケーション実行の際に，その結果を用いて各ジョブのアクセスファイル予測を行い（図 4.1 中 (3)），データローカリティを向上させるスケジューリングに役立てる（図 4.1 中 (4)）ことを目指す．

図 4.1 中 (1) でワークフローアプリケーションを事前に予め実行するときには，**異なるデータ，パラメタなどを用いて行ってもよい．**提案手法では，各ジョブとして実行されるプログラムのファイルアクセス傾向を得るため，同じプログラムを用いているものであれば，入出力ファイルの予測が可能となる．これは例えば，比較的小さなデータを用いた短時間で終わる実行を，プロファイルのための事前実行とすることができるということを意味する．また，一度プロファイルを行えば，以降の実行ではそれを使い回した提案手法の適用が可能となる．

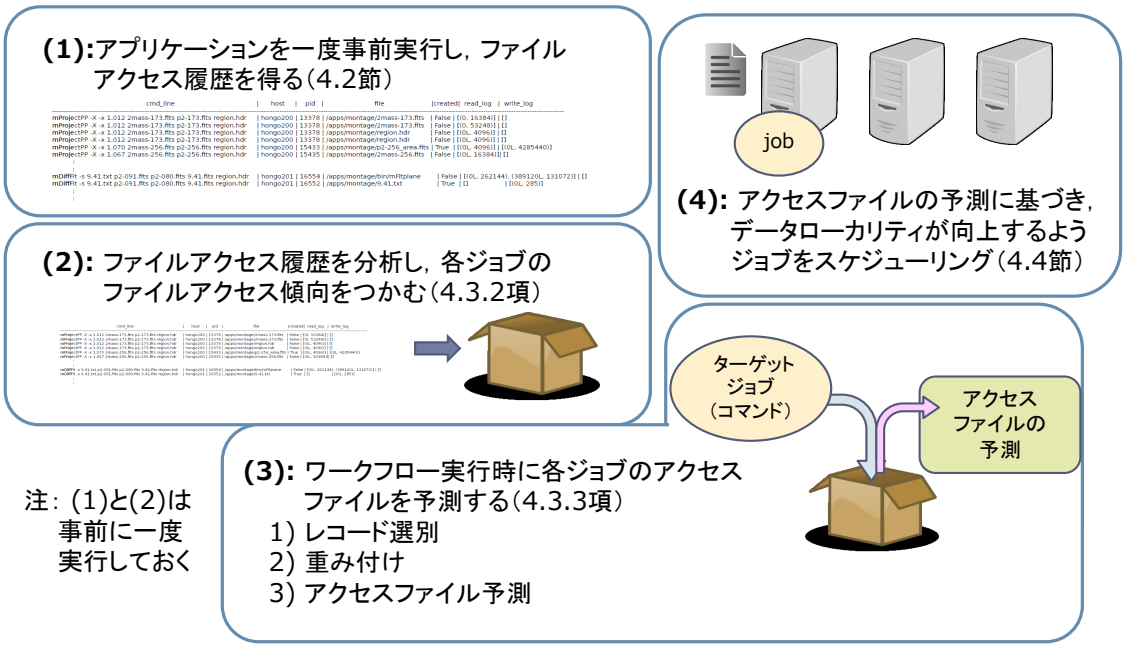

図 4.1. アクセスファイル履歴に基づくジョブスケジューリング最適化の全体像

```
                    cmd_line                          |    host    |  pid  |                file                |created|  read_log  | write_log
-------------------------------------------------------------------------------------------------------------------------------------------------------------
mProjectPP -X -x 1.012 2mass-173.fits p2-173.fits region.hdr   | hongo200 | 13378 | /apps/montage/2mass-173.fits   | False | [(0, 16384)] | []
mProjectPP -X -x 1.012 2mass-173.fits p2-173.fits region.hdr   | hongo200 | 13378 | /apps/montage/2mass-173.fits   | False | [(0, 53248)] | []
mProjectPP -X -x 1.012 2mass-173.fits p2-173.fits region.hdr   | hongo200 | 13378 | /apps/montage/region.hdr       | False | [(0L, 4096)] | []
mProjectPP -X -x 1.012 2mass-173.fits p2-173.fits region.hdr   | hongo200 | 13378 | /apps/montage/region.hdr       | False | [(0L, 4096)] | []
mProjectPP -X -x 1.070 2mass-256.fits p2-256.fits region.hdr   | hongo200 | 15433 | /apps/montage/p2-256_area.fits | True  | [(0L, 4096)] | [(0L, 4285440)]
mProjectPP -X -x 1.067 2mass-256.fits p2-256.fits region.hdr   | hongo200 | 15435 | /apps/montage/2mass-256.fits   | False | [(0L, 16384)]| []
        :
        :
mDiffFit -s 9.41.txt p2-091.fits p2-080.fits 9.41.fits region.hdr   | hongo201 | 16554 | /apps/montage/bin/mFitplane      | False | [(0L, 262144), (389120L, 131072)] | []
mDiffFit -s 9.41.txt p2-091.fits p2-080.fits 9.41.fits region.hdr   | hongo201 | 16552 | /apps/montage/9.41.txt           | True  | []                | [(0L, 285)]
        :
        :
```

図 4.2. 収集したファイルアクセス履歴例

4.2　ファイルアクセス履歴の収集

本節では，一度用いるデータ・パラメタなどの異なる実行を事前に行い，アプリケーションのファイルアクセス履歴を収集する手法を説明する．ワークフローアプリケーションのファイルアクセス解析は様々な研究により行われており [22, 38, 10]，そのためのプロファイラ Paratrac[23] も存在する．しかし，本研究では分散ファイルシステム Mogami にファイルアクセス収集のための機能を設け，GXP Make でファイルアクセス履歴収集のためのオプションを付けた実行を行う．それにより自動でワークフローを実行し，ファイルアクセス履歴を取得するといった，ユーザが簡単にファイルアクセス履歴を取得可能なシステムを構築する．

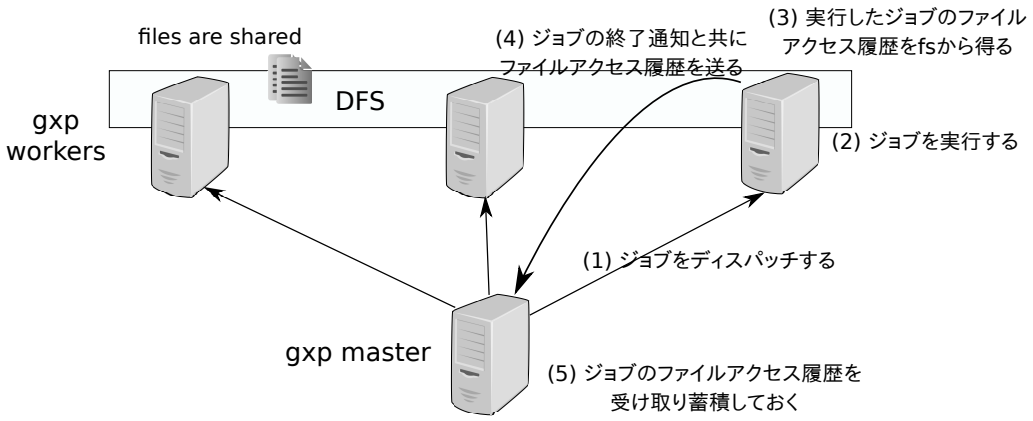

図 4.3. ファイルアクセス履歴収集機構の実装

4.2.1　収集するファイルアクセス

まず，どのようなファイルアクセス履歴を収集するかを述べる．集めるファイルアクセス履歴のフォーマットは以下のものとする（図 4.2）.

1. 実行時コマンド
2. PID
3. 実行ノードのホスト名
4. ファイルパス
5. ファイルが新しく作成されたかどうか (Boolean)
6. read 履歴
7. write 履歴

この情報をワークフローアプリケーション実行時に全ジョブが行うファイルアクセスに関して収集する．

4.2.2　ファイルアクセス履歴収集のためのアーキテクチャ

本項では，ファイルアクセス履歴の収集を行うための実装に関して述べる．ワークフローアプリケーションの実行時に各ジョブの行ったファイルアクセス履歴を収集するために実装に変更を加えた箇所は，以下の 4 点である．

Mogami 側

- FUSE の API を用い，open, read, write, close 時に，その要求プロセスの PID を取得する（そこから `/proc/pid/cmdline` を参照することによりそのプロセスの実行コマンドも得る）機構を実装
- GXP のプロセスとの通信用パイプを用意し，そのパイプを通じて，ある PID が行ったファイルアクセス履歴を返答する機構を実装

GXP Make 側

- ディスパッチしたジョブを実行した後に，実行したジョブの PID によりファイルアクセス履歴を分散ファイルシステムから取得する機構を実装
- 得たファイルアクセス履歴をジョブの終了通知とまとめてマスタノードに送信する機構を実装

これらの実装を行い，ワークフローアプリケーション実行の際に各ジョブが行ったファイルアクセスの履歴を，GXP のマスタノードが得ることができる（図 4.3）.

4.3　ファイルアクセス履歴を用いたアクセスファイル予測

本節では，収集したファイルアクセス履歴を用い，各ジョブがアクセスするファイルを予測する方法を述べる.

4.3.1　用語定義

ファイルアクセス履歴を用いた Input/Output ファイルを予測する手法を説明する前に，いくつか説明に使用する用語を定義する．以降，本稿ではプログラムを実行するためにシェルに与えられた文字列全体を**コマンドライン**，その最も先頭の引数を**コマンドネーム**，それ以降に続く引数を**コマンドライン引数**と呼ぶことにする．例えば,

```
$ cat A B C
```

という文字列がシェルに与えられた時，‘`cat A B C`’がコマンドライン，‘`cat`’がコマンドネーム，‘`A`’，‘`B`’，‘`C`’がコマンドライン引数という扱いになる.

4.3.2　ファイルアクセスルール作成

提案手法では，各ジョブの Input/Output ファイル予測のために，収集したファイルアクセス履歴の各レコードに対して**ファイルアクセスルール**を作成する．これは，そのファイルアクセスレコードにおいてどのようなファイルにアクセスしたかをより抽象化したものであり，異なるデータセット，パラメタによる実行のファイルアクセス履歴からのアクセスファイル予測を行うためである.

抽出しようとするルールは例えば，“**もしジョブのコマンドラインで** `-f` **の次の引数が** *x*`.jmn` **であった場合，**`x.rst` **という名前のファイルをアクセスする**といったものである．提案手法では，以下に示す 5 種類のルールを作成する.

- ReplacePos(n, a, b) : n は整数値，a と b は文字列．このルールでは，ジョブが n 番目の引数の，a を b に置き換えたもの，にアクセスすることを示す．例えば，ReplcePos(3, “jmn”, “knp”) というルールは，もしそのコマンドの第 3 引数が *x*`.jmn` であるならば，*x*.knp というファイルにアクセスすることを意味する.

- InsertPos(n, i, b) : n と i は整数値，b は文字列．このルールでは，ジョブが n 番目の引数の，i 番目の文字の後に b を挿入したもの，にアクセスすることを示す．例えば，InsertPos(3, 10, ".txt") というルールは，もしそのコマンドの第 3 引数が x であるならば，x[0:10] + ".txt" + x[10:] というファイルにアクセスすることを意味する．
- ReplaceOpt(f, a, b) : ReplacePos と類似しているが f は `-f`，`--config_file` のような文字列である．このルールは f の次に続くコマンドライン引数に適用される．
- InsertOpt(f, i, b) : InsertPos と類似しているが f は ReplacePos 同様の文字列であり，ルールは f の次に続くコマンドライン引数に適用される．
- DependentOutput(d) : d は $0 < d \leq 1$ の小数値．`Makefile` そのジョブが依存しているとされるファイル[1]の割合 d のファイルをアクセスしたこと示す．

このように，収集したファイルアクセス履歴の各レコードの，J というコマンドラインのジョブが F というファイルにアクセスした，という情報を，以下の手順により抽象化されたルールに変更する．

1. まず，DependentOutput(d) に当てはまるかチェックする．当てはまる場合は DependentOutput(d) を作成する
2. コマンドラインの n 番目の引数 S について，S 中の部分文字列 p を q という文字列に置換すると F となる p を見つけ出す．ここで，p の S 中の先頭文字列オフセットを i, p の長さを l とする．
   (a) p が空文字列の場合，すなわち, S にある文字列を挿入することで F が得られる場合は，InsertPos(n, i, q) というルールを作成する．
   (b) p が空文字列でなく S と同一でもない場合，ReplacePos(n, p, q) というルールを作成する．
   (c) 上記のどれも当てはまらなければ（すなわち S は F は共通の接頭文字列を持たない場合)，ルールを作成しない．
3. 同様にして，コマンドラインオプションのように思われる全ての引数（すなわち，'-' や '--' で始まる引数）に対して，上記の手順を適用し p, l, i を取得し，InsertOpt(f, i, q) や ReplaceOpt(f, p, q) 等のルールを生成する．

あるファイルアクセスレコードに対して，作られた DependentOutput 以外の全てのルールから，最も短い q を持つものを選び出し，DependentOutput と共にそのファイルアクセスレコードのファイルアクセスルールとする．

4.3.3　ジョブのアクセスファイル予測

各ファイルアクセスレコードに対応するファイルアクセスルールを得たところで，それらを用いてワークフロー実行時に各ジョブに対するアクセスファイルを予測するための手順を示す．ジョブのアクセスファイル予測は以下の手順で行われる．

[1]現在の実装では静的に依存関係を記述したファイルを用意することで与えている，しかし理想としては `Makefile` から，何らかの方法で自動で得ることが考えられ，これは今後の課題とする

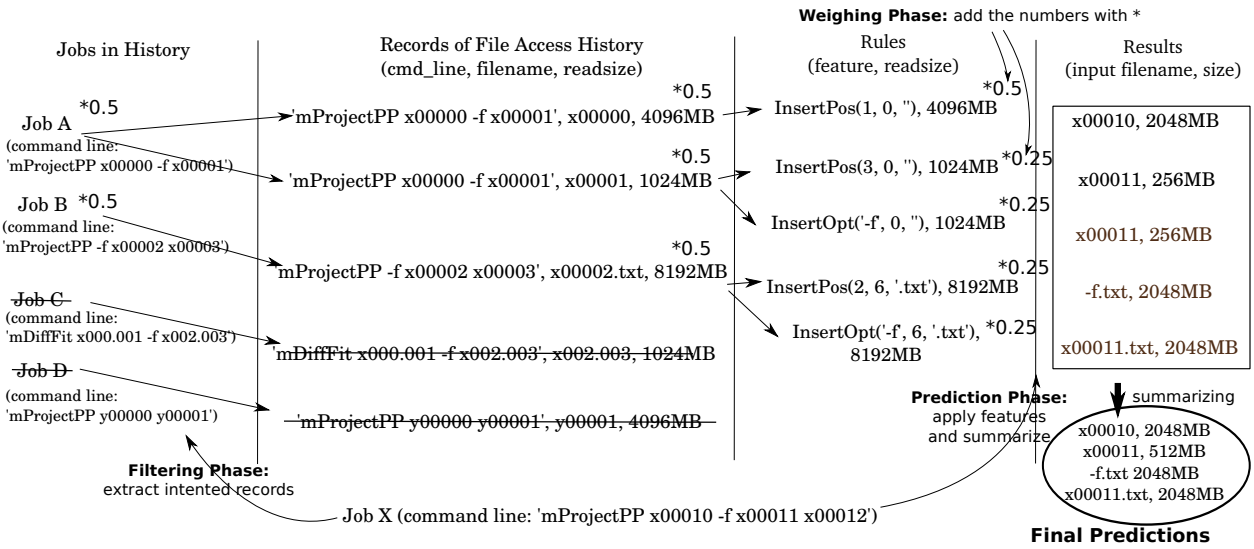

図 4.4. ジョブの入力ファイルとサイズの予測の例

1. **レコード選別:** アクセスファイル予測の対象とするジョブにもっとも近いファイルアクセスレコードを，収集したファイルアクセス履歴の中から選出する
2. **重み付け:** 選出したレコードに対応するルールに，それぞれ重みをつける
3. **アクセスファイル予測:** 最終的にジョブの入力ファイル名と，その読み込みサイズ予測を得る

以降では，図 4.4 に示す例を用いて，各手順を説明する．この例では，ジョブ X（コマンドライン：“mProjectPP x00010 -f x00011 x00012”）の入力ファイルとその読み込みサイズを，ファイルアクセス履歴を用いて予測する．事前に収集しているファイルアクセス履歴には，以下のファイルアクセスレコードが含まれている．

- ジョブ A（コマンドライン:“mProjectPP x00000 -f x00001”）が x00000 から 4096 MB，x00001 から 1024MB のデータを読み込んだ
- ジョブ B（コマンドライン：“mProjectPP -f x00002 x00003”）が x00002.txt から 8192 MB のデータを読み込んだ
- ジョブ C（コマンドライン：“mDiffFit x000.001 -f x002.003”）が x002.003 から 1024 MB のデータを読み込んだ
- ジョブ D（コマンドライン：“mProjectPP y00000 y00001”）が y00001 から 4096 MB のデータを読み込んだ

1. レコード選別

ここでは，ファイルアクセス履歴全体から，もっとも予測対象のジョブに “似ている” コマンドのファイルアクセスレコードを選出する．まず第一に，全てのファイルアクセスレコードから，同じコマンドネームを持たないファイルアクセスレコードを排除する．これは，コマンドネームが異なるアプリケーションは，いくら他の引数が同一であろうとも似たファイルアクセスは行わな

いだろうとする推論に基づいている．そして，残されたファイルアクセスレコードから，同一のコマンドライン引数を最も多く持つものを，“似ている” コマンドのファイルアクセスレコードとして選出する．条件に当てはまるファイルアクセスレコードが複数存在する場合には，それら全てを選出する．

図 4.4 に示した例では，上側 2 つのレコード（ジョブ A とジョブ B に対するファイルアクセスレコード）がジョブ X と似ているものとして選出される．ジョブ C に対するファイルアクセスレコードは，コマンドネーム（ここでは mDiffFit）がジョブ X と異なるため排除され，ジョブ D に対するファイルアクセスレコードは，コマンドネームはジョブ X と同一であるものの，他に同一なコマンドライン引数を持たないため排除される．

2. 重み付け

ここでは，選出した各ファイルアクセスレコードに重み付けを行う．先のレコード選別で，J 個のジョブにより生成された，n 個のファイルアクセスレコードが選出されたとする．その場合に，それぞれのファイルアクセスレコードに $\frac{1}{nJ}$ の重みを付ける．

例えば，図 4.4 の例を再び用いると，図中 ‘*’ で示す重みが各ファイルアクセスレコードに負荷される．

3. アクセスファイル予測

最終的に，ここで予測対象ジョブの入力ファイルとそれぞれのファイルが読み込まれる可能性が予測される．また，ここではそれぞれファイルの読み込みサイズも同時に予測される．

入力ファイルとして読み込まれる可能性のあるファイルのリストは，単に選出したファイルアクセスレコードを，予測対象ジョブのコマンドラインに適用することで得られる．また，あるファイルが読み込まれる可能性は，そのファイルが読み込まれるという予測の際に用いたファイルアクセスルールに付けた重みを足し合わせることにより得られる．同様にして，読み込みサイズの予測は，各ファイルアクセスレコードに対応する（事前実行の際に）読み込んだデータサイズと，重みを掛け合わせたものを足し合わせて得る．いずれにせよ，最終的に得た予測ファイルが存在するかどうかをメタデータサーバに確認し，もし存在しなかった場合には，その予測を排除する．

図 4.4 に示した例では，以下の通り入力ファイルとその読み込みサイズが予測されることとなる．

- x00010 : 可能性 0.5 (InsertPos(1, 0, ‘’) ルールより) = 0.5, 予測読み込みサイズ $0.5 \times 4096 = 2048$ (MB).
- x00011 : 可能性 0.25 (InsertPos(3, 0, ‘’)) ルールより） + 0.25 (InsertOpt(‘-f’, 0, ‘’) ルールより) = 0.5, 予測読み込みサイズ $0.25 \times 1024 + 0.25 * 1024 = 512$ (MB).
- -f.txt: 可能性 0.25 (InsertPos(2, 6, ‘.txt’) ルールより) = 0.25, 予測読み込みサイズ $0.25 \times 8192 = 2048$ (MB).
- x00011.txt: likelihood 0.25 (InsertOpt(‘-f’, 6, ‘.txt’) ルールより) = 0.25, 予測読み込みサイズ $0.25 \times 8192 = 2048$ (MB).

Algorithm 4.4.1 ファイルの位置を考慮したディスパッチノード決定アルゴリズム

```
for all job in ディスパッチ可能なジョブ群 do
  job の入力ファイルを予測
  予測した入力ファイルそれぞれについて，データを持つノードをメタデータサーバから得る
  bestnode = 予測されるデータの読み込み量が最も多いノードを選出
  if bestnode のリソースが job の必要とする分空いている then
    job を bestnode にディスパッチ
  else
    job を leftjob 群に追加
  end if
end for
leftjob 群のジョブを従来のディスパッチ機構によりリソースの空いているノードにディスパッチ
```

4.4　ジョブスケジューリングとデータ転送最適化

本節では，アクセスファイルの予測に基づき，ジョブスケジューリングとデータ転送を最適化する手法を述べる．ジョブスケジューラにとって，あるジョブをディスパッチするときに，どのノードにディスパッチすることが最適であるかは決して自明な問題ではない．ジョブの実行にかかる時間は CPU，ネットワーク，ストレージの性能に加え，実行中のジョブの数，各ジョブのネットワーク使用状況など，アプリケーション実行中に状況が変化するものの影響も受ける．本研究ではデータ集約的アプリケーションの高速化のため，ファイルアクセスに費す時間の短縮を図る．そのため，以下の方針により可能な限りローカルストレージを通してファイルアクセスを行い，リモートノードへの通信を削減することを目指す．

1. ジョブを可能な限り入力（と予測される）ファイルが存在するノードにディスパッチする
2. ワークフローで先に実行されるジョブで読み込むファイルを事前チェックし，レプリケーションを用いて効率的に事前にファイルデータ転送を行っておく
3. ローカルノードにコンテンツが存在するファイルを開く際にはメタデータサーバへの通信を行うことなく，ノード内通信でファイルを保持するノードを検知する

以降では，これらそれぞれについての効率化手法を提案する．

4.4.1　入力データの位置を重視したジョブ配置手法

本研究では，できるだけリモートノードとの通信を介したファイルアクセスを行わないスケジューリングを目指す．このためには，ジョブスケジューラは任意のファイルについて，どのデータサーバがそのファイルコンテンツを保持しているか知ることが必要である．従って，本手法ではメタデータサーバに任意のファイルの，コンテンツを持っているサーバを問い合わせるための API を実装する．

ある実行可能なジョブ群をディスパッチするノードを決めるアルゴリズムを **Algorithm 4.4.1** に示す．このアルゴリズムに従い，ジョブを実行するノードを決定する．

Algorithm 4.4.2 レプリケーション決定時のアルゴリズム

```
for all filename in アクセスが予測されるファイル do
  id = filename を持つノードの id
  N = filename を読み込むジョブの数
  if filename これ以降変更されない then
    if p_id < N then
      /* ジョブの実行プログラムや皆が読み込む設定ファイルなどが考えられる */
      while sum(p_r) (r ∈ rep_node_id) < N do
        rep_node_id に適当にノードを加える
      end while
      rep_node_id のノードへのレプリケーション要求を行う
    end if
  else
    /* どこかでファイルが作成される or 変更が加えられる */
    if filename を読みこむジョブ X が他のジョブの output を含め集約処理を行う（図 4.5 左）
    then
      if X の入力ファイルを集めるノード gather_node が仮決定されていない then
        X を実行するノード gather_node を仮に定める
      end if
      filename が生成され次第 gathre_node へのレプリケーションを行う要求をする
    else if filename を作成するジョブが他のファイル output_files を作成する and それらのファ
    イルが複数のジョブから読まれる（図 4.5 右） then
      if average(p_k) > output_files を読み込むジョブの数 then
        while sum(p_r) (r ∈ rep_node_id) < N do
          rep_node_id に適当にノードを加える
        end while
        output_files が作成され次第 rep_node_id のノードへのレプリケーションを行う要求を
        する
      end if
    end if
  end if
end for
```

ここで注意すべきなのは，このアルゴリズムにより入力ファイルを保持するノードがディスパッチノードとして，“優先” されることである．ここで優先でなく決定をしない理由は，他のノードのリソースが空いているにも関わらず，入力ファイルを保持するノードのリソースが空いてない場合にジョブディスパッチが遅れることを防ぐためである．

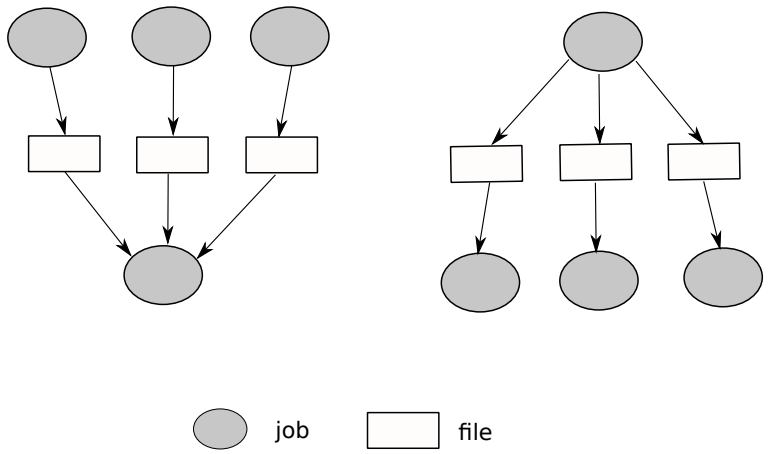

図 4.5. レプリケーションを行うデータ入出力流れの例

4.4.2　ジョブ間のデータの流れに応じたファイルレプリケーション

前節で述べた入力ファイルの存在するノードを優先するジョブディスパッチ手法では，例えば入力ファイルが多数のノードに分散して同程度存在する場合に，どのノードを実行ノードとして選出しようとも，多くのファイルアクセスが依然リモートファイルアクセスとなる．この状況を避けるために本研究では，ワークフローの先を考慮したファイルのレプリケーションを行い，更なるデータローカリティの向上を目指す．

GXP Make では，ジョブ間の依存関係の解析に GNU Make をそのまま用いているため，通常であれば先に実行されるジョブに関しての情報を，ジョブスケジューラは持つことができない．本提案手法では，`make -n` コマンド[2]により得られる出力を用い，`make` が実行される度に，今後実行されるジョブを得る．それらのジョブに対して，提案手法によりアクセスファイル予測（出力ファイルも入力ファイルと同様に予測する）を行い，全ての予測アクセスファイルを照合することでジョブ間のデータの流れを得る．

レプリケーションを行うかどうかは，ジョブ間のデータの流れを考慮し，ファイル毎に考慮する．レプリケーションを行うべきとし，メタデータサーバにレプリケーション要求を出す条件と要求内容は，実行環境のノード数を n, 各ノードの CPU 数を p_k $(0 < k \leq n, k \in \mathbb{N})$ として，**Algorithm 4.4.2** に従う．

4.4.3　入力ファイルのローカル探索

ここまでの手法により，データローカリティを考慮したジョブディスパッチが期待される．ここでは，データローカリティの向上を前提として，更なる発展としてファイルを開く際のメタデータへの問い合わせを削減する手法を提案する．

データローカリティを高め，ジョブ実行時のファイルアクセスをネットワークを介さず行えたとしても，ファイル open 時，close 時のメタデータサーバへのアクセスは削減することができな

[2] `make` は `-n` オプションを付加すると，コマンドを実際に実行することはせず表示のみを行う

い．それらを削減するために，データサーバでは自身がコンテンツを保持するファイルパスを管理しておく．そして，クライアントがファイルを開く際に，自身がデータサーバでもある場合のみデータサーバに問い合わせを行う．ファイルコンテンツがノード内に存在する場合のみ，ここでファイルコンテンツを保持するノードが判明するため，メタデータサーバへの問い合わせは行わない．ノード内に存在しなかった場合には，通常通りメタデータサーバに問い合わせを行う．

この手法を用いつつ Close-to-open コンシステンシを保つためには，ファイルの削除，名前変更時に，同期的にデータサーバへの通知も行う必要がある．そのため，ファイルの削除，名前変更のパフォーマンスが劣化することとなる．しかし，ワークフローアプリケーションの実行中にこれらの操作を行うことは，ファイル読み込み，書き込みに比べ比較的稀であることを想定し，本提案手法ではこれを採用する．

第5章　提案手法II：高遅延環境におけるデータ転送効率化のための適応的先読み手法

本章では，分散ファイルシステムを基盤とした高遅延環境におけるファイルアクセスのための，適応的な先読み手法を提案する．

5.1　概要

分散ファイルシステムを基盤とし，ワークフローアプリケーションを複数拠点にまたがり実行すると，第 4 章で提案した手法によりネットワーク通信を介したファイルアクセスを抑えようとも，拠点間でデータを転送する必要が生じる場合がある．この時，通常の分散ファイルシステムと同じくアプリケーションからデータ要求があったときオンデマンドにデータを転送すると，データ転送毎にネットワーク遅延の影響を受けることとなり，結果としてネットワーク帯域よりはるかに低いスループットの読み込みとなってしまう（第 1 章図 1.2）．そこで，本研究では広域分散ファイルシステムのための適応的なデータ先読み手法を提案する．提案手法の全体像を図 5.1 に示す．分散ファイルシステムの先読みを行うために考えられるべき問題は，先読み要求を出すファイルの位置とそのサイズである．本研究では，それらを適応的に環境とアプリケーションに応じて決定するするため，以下の手法を提案する．

- ファイルアクセスの規則性を自動で検出し，その規則性に基づいて先読み範囲を決めるための手法（5.2 節）
- ネットワーク環境やその状況により最適値の異なる先読み要求サイズを，標準の TCP 上でシンプルに決定するための手法（5.4 節）

以降では，それぞれの手法に関して詳細を述べる．

5.2　アクセスパターン検出手法

広域分散ファイルシステムに限らず，ファイルシステムのデータ先読みのためにはデータアクセスパターンを検知し，今後必要とされるであろう部分を予測することが必要である．本節では，提案するファイルアクセスパターンの検知手法に関して述べる．

ファイルのアクセスパターンにより次のアクセスを予測するために，以前行われたファイルアクセスの規則性に注目する．世の中でファイルアクセスを行うアプリケーションには，ファイルをシーケンシャルにアクセスするアプリケーションが比較的多い．その他にも，一定の大きさのファイル読み込み，読み飛ばしを繰り返すアクセスでは，その規則性を検知して先読みを行うこ

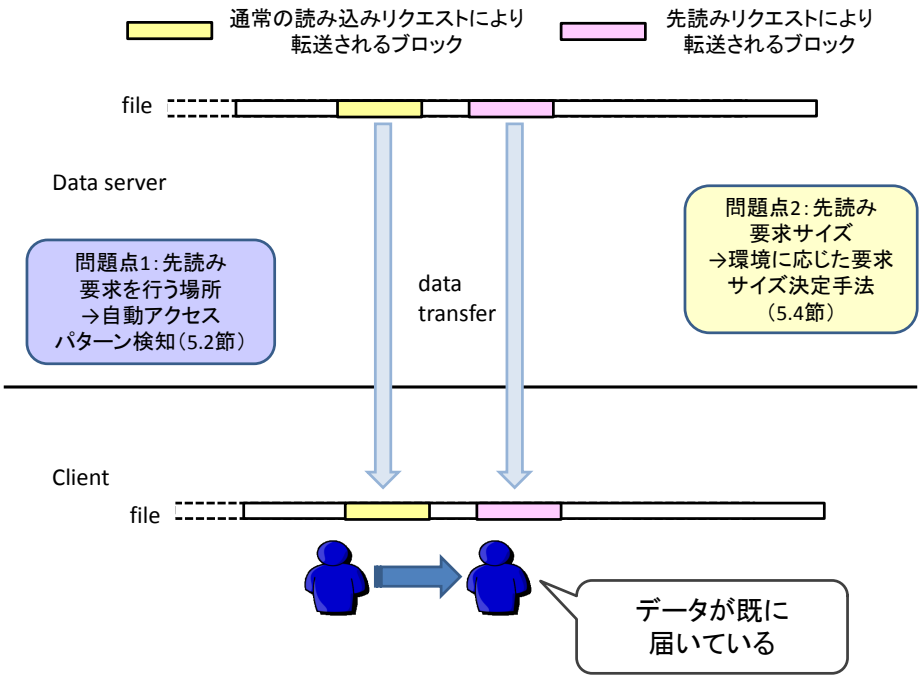

図 5.1. 高遅延環境におけるデータ転送効率化のための適応的先読み手法の全体像

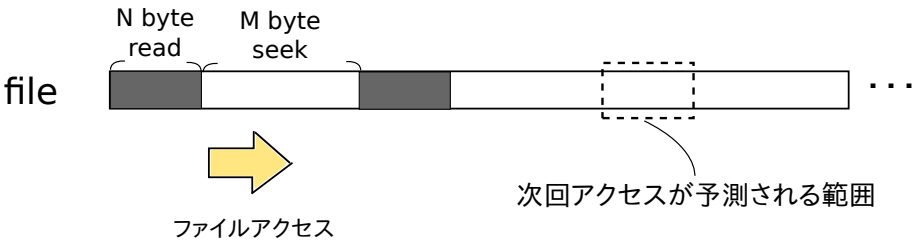

図 5.2. ストライドアクセス

とができる．そこで本章では，ファイルアクセスの規則性に応じてアクセスパターン分類を行い，先でアクセスされそうな場所を予測する手法を提案する．

以下でははじめにアクセスパターンの分類と，ファイルシステムのレイヤで保持しておく情報（パラメタ）に関して述べる．次に，分類したファイルのアクセスパターンをファイルアクセス中に検出する手法を提案する．その後，そのアルゴリズムによるアクセスパターン検出を用いた先読み要求の流れに関して述べる．

5.2.1　アクセスパターンの分類と保持しておくべきパラメタ

本研究では規則性のあるアクセスパターンとして，シーケンシャルアクセスとストライドアクセスを取り上げ，その規則性に基づいて先読みを行う．ストライドアクセスとは，ファイルを飛ばし飛ばし読み込みを行うアクセスのことを指すが，ここでは簡単のため，一定の大きさ N byte のデータを読み込み，一定の大きさ M byte のデータを飛ばす動作を繰り返すアクセス（図 5.2）のこととする．これらの 2 つのアクセスパターンに合致しないタイプのアクセスを，規則性がなく予測が難しいアクセスであるランダムアクセスと分類することにする．

次に，これらのアクセスパターンに分類する際に，利用するためのパラメタに関して述べる．分散ファイルシステムのレイヤでアプリケーションがファイル読み込みを行う際に得られる情報は，各 read の offset, size のみである．提案手法では，これらと以下に述べるパラメタ群を利用することによってアクセスパターンを検知・分類する．

mode 現在アプリケーションがどのようなファイルアクセスモードと判断されているか．シーケンシャルアクセスモード，ストライドアクセスモード，ランダムアクセスモードの三つの種類がある．

stride_read_size ストライドアクセスのときに，シーケンシャルに読み込むと推定されるデータ量.

stride_seek_size ストライドアクセスのときに，シーケンシャルな読み込みのあと，シークを行うデータ量.

last_seq_read_start 最後にシーケンシャルアクセスがはじまったファイル内 offset．各 read で前回の read の続きを読んだ場合は更新されず，そうでない場合にその read の offset が入る．

last_read 一つ前に呼び出された read で読んだデータ直後の offset.

これらの情報を保持し，利用することによりシーケンシャルアクセスモード，ストライドアクセスモード，ランダムアクセスモードの 3 つのアクセスモードを遷移するための判断が行える．具体的な遷移アルゴリズムに関しては次節にて説明する．

5.2.2　各モード中のアクセスパターン検出アルゴリズムと先読み

次に，前項で述べたパラメタと，アプリケーションからの各 read の offset と size を元にアクセスパターンのモード（パラメタの `mode`）を変更しつつ，そのモードに合わせた先読みを行うアルゴリズムに関して説明する．ファイルを open した際には，ファイルをシーケンシャルにアクセスするアプリケーションが比較的多いことからファイルアクセスモードはシーケンシャルモードではじめる．また，`last_seq_read_start`, `last_read` の値は 0 に設定しておく．

シーケンシャルアクセスモード

現在の先読みモードがシーケンシャルアクセスモードである場合には，**Algorithm 5.2.1** に示したアルゴリズムに従い，今回の読み込み offset と last_read のみに着目してアクセスモードを推定する．このモードから他のモードに変わらない間は，現在アクセスしてる部分からシーケンシャルにそのままアクセスが続くことを予測し，シーケンシャルな先読み要求を行う．

Algorithm 5.2.1 シーケンシャルアクセスモードのときのアクセスパターン検出アルゴリズム

```
if offset == last_read then
    mode = シーケンシャルアクセスモード
else if offset > last_read then
    mode = ストライドアクセスモード
    stride_read_size = last_read - last_seq_read_start
    stride_seek_size = offset − last_read
else
    mode = ランダムアクセスモード
end if
```

Algorithm 5.2.2 ストライドアクセスモードのときのアクセスパターン検出アルゴリズム

```
if offset == last_read then
    # 読み込まれる N バイトのデータが分割されている
    # 場合も考慮した分岐
    if (offset + size) が次にアクセスが予測される部分を越える then
        mode = シーケンシャルアクセスモード
    else
        mode = （引き続き）ストライドアクセスモード
    end if
else if last_read, last_seq_read_start, offset の間隔がストライドアクセスパラメタにマッチ then
    mode = （引き続き）ストライドアクセスモード
else
    mode = ランダムアクセスモード
end if
```

ストライドアクセスモード

シーケンシャルアクセスモードに続いて，ストライドアクセスモードのときのアクセスパターン検知と先読みアルゴリズムを **Algorithm 5.2.2** に示す．先にも述べた通り本提案手法でストライドアクセスとして扱うのは，一定の大きさ N byte のデータを読み込み，一定の大きさ M byte のデータを飛ばすアクセスのみである．現在の提案手法では，このアクセスパターンから少しでもずれてしまうとストライドアクセスとは判断しない．

ストライドアクセスモードのときは，このままのサイズの read と seek を繰り返した場合にアクセスすることになる部分を計算し，先読み要求を行う．このとき，パラメタ `stride_read_size`，`stride_seek_size`，`last_seq_read_start` を用いる．

ランダムアクセスモード

ランダムアクセスモードのときは各読み込みの offset が last_read と一致したとき，つまり前回の read の続きにアクセスした場合のみシーケンシャルアクセスモードに切り替える．またこのモー

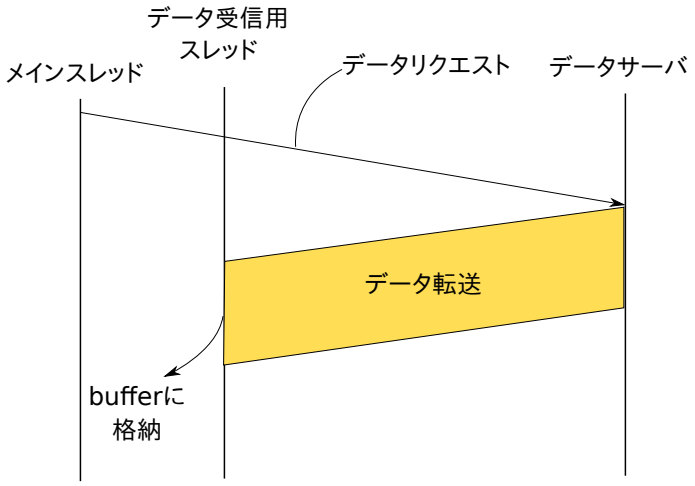

図 5.3. データ受信用スレッドとデータリクエスト

ドでは将来読み込まれる部分を予測することが困難であるため，先読み要求は行わない．

5.3　アクセスパターン検出と先読み要求の流れ

分散ファイルシステムでは多くの場合，ファイルはある大きさのブロック単位（これをブロックサイズと呼ぶ）で管理され，データ転送もこのブロックの単位で行われる．このブロックはアプリケーションから意識されることはない．ここでは提案手法におけるファイル読み込みの流れに関して述べる．まず，ファイルを開いたときに目的のファイルを持っているサーバ（データサーバとする）とコネクションを持つが，それと同時にデータの受信用スレッドを作成する．以降ファイルを読み込むときにはこのデータ受信用スレッドがデータを受け取り，バッファに格納する (図 5.3). メインスレッドではアプリケーションからの要求を受け取り，アプリケーションに要求されたデータを返すまでの手順を分散ファイルシステムの read システムコールとして実装する．実装した read システムコールの大まかな流れを以下に示す．

1. read 要求の offset, size と保持しているパラメタを用いて，アクセスパターンを推定する（5.2 節）．
2. 要求されたデータがバッファに存在しない場合のみデータサーバにリクエストを出し，受信用スレッドがデータを受信してバッファに格納するまで待つ．
3. 1. で判断したアクセスパターンとパラメタにより次にアクセスされそうなブロックの集合を決定する．このときのブロック数の決定方法は 5.4 節で述べる．
4. 3. で計算したブロックのうち，データサーバに要求を未だ出しておらず，バッファにも格納されていないブロックの転送要求をデータサーバに出す．
5. 2. で得た要求データを用いて，read システムコールを返す．

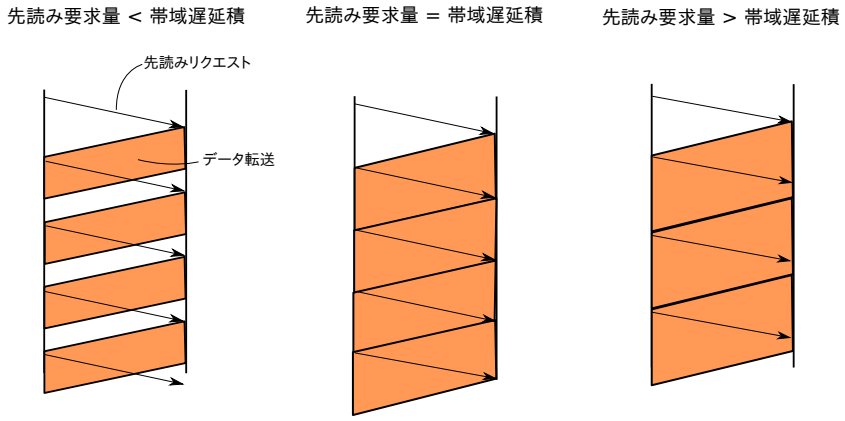

図 5.4. 先読み要求量と帯域帯域遅延積の関係

5.4　適応的なデータ先読み量自動調整手法

本節では，5.2 章で延べたアクセスパターンの分類・検出手法を用い，どの程度の量の先読み要求をデータサーバに出すかの調整方法に関して述べる．

5.4.1　理想的なデータ先読み要求量

まず本提案手法で理想とする先読みデータ要求量に関して説明する．ここでは研究の第一歩としてシンプルな状況を仮定することにし，データサーバ上でファイルデータがキャッシュに存在していることを前提として考える．この場合，遠隔ファイルのアクセススループットはディスクの読み込み帯域に律速されず，ネットワーク帯域に律速されることになる．分散ファイルシステムでネットワークの性能を極限まで引き出したデータ転送を行うためには，データサーバがあるデータ先読み要求を受け取ってから，次のデータ先読み要求を受け取るまでの間にデータをネットワークに転送し続ける必要がある．これをちょうど実現するためには (RTT × ネットワーク帯域幅)，帯域遅延積分のデータ先読み要求を送信するようにすればよい．帯域遅延積より小さい量を先読み要求量にした場合には図 5.4 の左側に示した図のようなデータ転送となり，ネットワークの性能を活かしきれない．逆に帯域遅延積以上のデータ先読み要求を出してしまうと，図 5.4 の右側に示した図のようなデータ転送となり，先読みが外れたときに前の先読み要求によるデータの転送後，実際にアプリケーションが要求したデータの転送が行われるため，先読みを行うことによるリスクが大きくなってしまう．従って，データ先読み要求量を帯域遅延積とすることによって，ネットワーク帯域を十分に活かしつつ，先読みが外れたときのリスクは最小限とすることができる．この場合，先読みが外れたときの，次データ送信にかかる遅延時間は最悪で RTT となる.（先読み要求を出した直後にアプリケーションが先読みから外れた場所をアクセスした場合）

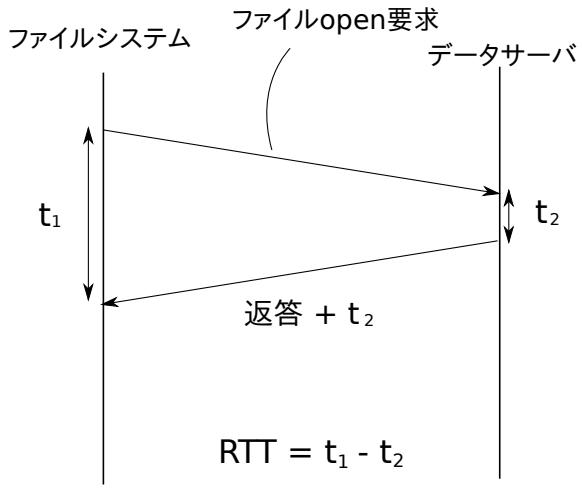

図 5.5. RTT の算出手順

5.4.2　先読み量自動調整手法

理想的なデータ先読み要求量を帯域遅延積とした上で，分散ファイルシステムでの帯域遅延積の推定方法に関して述べる．簡単な方法としてユーザに帯域遅延積の値を設定として与えてもらうということが考えられるが，この方法ではユーザは，(ファイルシステムのクライアントの数 × ファイルサーバの数) 通りの帯域遅延積を把握し，設定する必要がある．しかし，その設定は困難な上にユーザに負担を強いることとなるため，可能ならば自動で推定し調整を行いたい．また，ネットワークの状態は刻一刻と変化するため，帯域遅延積もネットワークの状態に応じて変化する．手動調整ではその変化に対応することはできず，その意味でも自動調整を動的に行うことは必要である．

本節では，標準的な TCP の用い方で得られる情報を元に，シンプルに自動決定する手法を提案する．先読み要求量の初期値は分散ファイルシステムの実装ブロック 1 つ分の大きさに設定する．

帯域遅延積を推定するため，まず遅延（RTT）の測定をファイルを開いた際に行う．アプリケーションからあるファイルの open 要求を受け取ったのちに，データサーバに open 要求を送ってからその返答を受け取るまでの時間 t_1 を測定しておき，データサーバ側では open 要求を受け取ってから処理を行い返答を返すまでの時間 t_2 を測定する．そして t_2 を返答と共に送信し，ファイルシステム側で $t_1 - t_2$ を計算することで RTT を得る（図 5.5）．

次に，帯域推定の手法に関して述べる．データを受信し，帯域推定から，データ先読み要求量を帯域遅延積に設定するまでのアルゴリズムを **Algorithm 5.4.1** に示す．帯域推定は，データの受信用に作成したスレッドでデータを受信する際の時間計測により行う．ただし，データ受信時に既に受け取り側受信バッファにデータが到着しており，バッファからデータを読み込むだけになった場合は，データ受信にかかった時間から正確な帯域を計算することはできない．しかし，そのような場合は図 5.4 の左側に示したようなデータ転送の待ち時間は生じてないと判断できるため，ネットワーク帯域を十分に活かした通信が行われており，先読み要求量を増やす必要はない．この場合を検知するために，データ受信の前にそのソケットが読み込み可能であるかどうかをチェックし，既に読み込み可能であった場合には，帯域推定やデータ先読み要求量の変更を行わないこ

Algorithm 5.4.1 データ受信用スレッドのアルゴリズム

```
while ファイルが開かれている do
  if socket からデータ読み出し可能 then
    帯域推定を行わないモードに変更
  else
    帯域推定を行うモードに変更
  end if
  データのヘッダを受信する
  start_time = NOW()
  ブロックデータを受信する
  end_time = NOW()
  if 帯域推定を行うモード then
    推定帯域 = 受信したデータサイズ / (end_time − start_time)
    先読み要求量 = 推定帯域 × 推定 RTT
  end if
end while
```

ととする.

第6章　評価I：各提案手法に対する評価

本章では，第 4, 5 章で述べた提案手法の評価を行う．まず 6.1 において，実験で用いた環境に関して述べ，6.2 でアクセスファイル履歴に基づくジョブスケジューリングとデータ転送最適化手法（第 4 章）の評価，6.3 で高遅延環境におけるデータ転送効率化手法（第 5 章）の評価に関して述べる．

6.1　実験環境

本実験で用いた実験環境は表 6.1 に示す 4 つのクラスタ（hongo, huscs, kyoto, tsukuba）である．これらのクラスタは InTrigger[3] プラットフォームの一部である．各クラスタのノード間ネットワークを図 6.1 に示す．hongo, huscs, kyoto, tsukuba クラスタのノードはそれぞれ，東京，札幌，京都，筑波に位置している．kyoto クラスタで実験に用いたノードのスペックは 2 種類存在し，これらを kyoto0, kyoto1 として扱う．以降の実験ではこれらのクラスタからいくつかのノードを選出して実験を行う．

6.2　アクセスファイル履歴に基づくジョブスケジューリングとデータ転送最適化手法評価

本節では，第 4 章にて提案したデータ転送効率化手法を，数種類のベンチマークアプリケーションにより評価する．

表 6.1. 評価用クラスタのスペック

クラスタ名	OS	CPU	メモリ
hongo	Linux 2.6.32 (64bit)	Intel Xeon E5560 8cores (w/ HT 16cores)	24GB
huscs	Linux 2.6.26 (64bit)	Intel Xeon E5530 8cores (w/ HT 16cores)	
tsukuba	Linux 2.6.26 (64bit)	Intel Xeon E5620 8cores (w/ HT 16cores)	
kyoto1	Linux 2.6.26 (64bit)	Intel Xeon E5530 8cores (w/ HT 16cores)	
kyoto0	Linux 2.6.26 (64bit)	Intel Core2 6400 2cores	4 GB

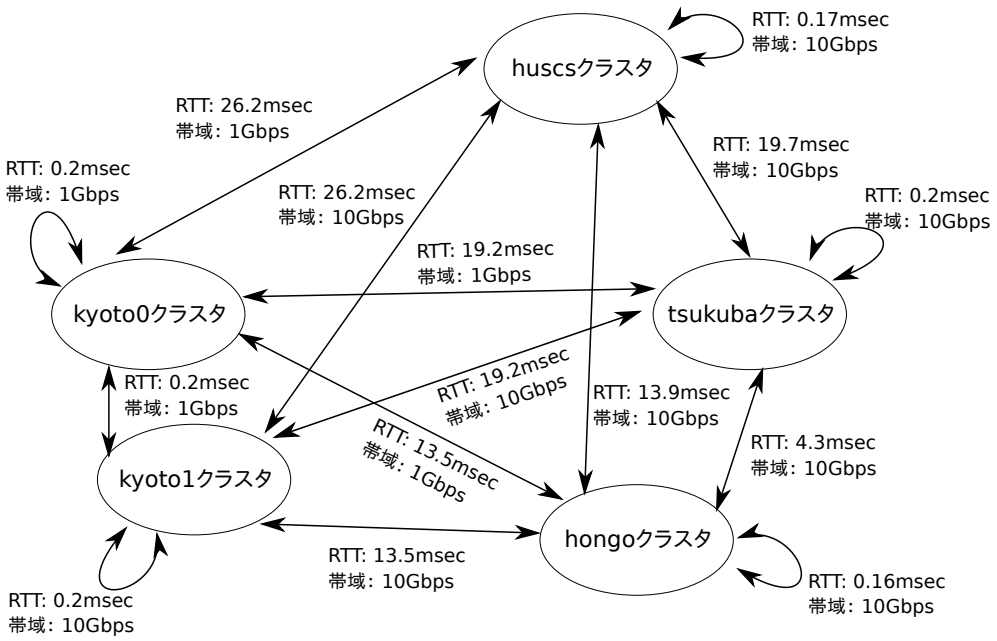

図 6.1. 評価用クラスタネットワーク環境

表 6.2. データアウェアなジョブスケジューリング効果の評価

	ローカルファイルアクセス率 (%)	ジョブの平均実行時間 (sec)	ワークフロー全体実行時間 (sec)
提案手法あり	100.0	10.11	57.404
提案手法なし	8.5	54.23	149.652

6.2.1　データアウェアなジョブスケジューリング効果の評価

本実験では，データアウェアなジョブスケジューリング手法，すなわちジョブをデータの位置するノードに割り当てることにより，期待できる効果を測定する．実験は huscs クラスタのノードを 11 ノード用いて行った．そのうち 1 台は Mogami のメタデータサーバとしてセットアップし，残り 10 ノードによりワークフローアプリケーションを実行する．初期セットアップとして各ノードに 1GByte のファイルを 8 個ずつ，合計 80 個作成する．ベンチマークワークフローアプリケーションは，80 個の全く依存関係のないジョブを並列実行し，それらのジョブはそれぞれ別のファイルを読み込むだけのジョブである．各ノードはジョブを 8 つまで並列実行可能な設定とし[1]，合計 80 個の全ジョブは開始と同時に並列実行される．本実験では，実験結果は 10 回実行の平均を算出して用いている．

実験結果を表 6.2 に示す．表中には提案手法を用いたジョブスケジューリングと，用いないジョ

[1]各ノードが同時並列実行可能なジョブの数は，デフォルトでは CPU の数に設定されるが，GXP の設定により指定可能

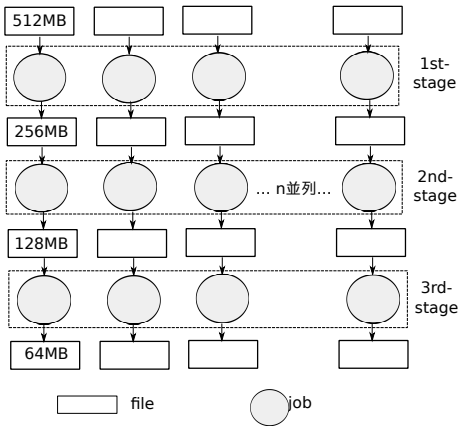

図 6.2. Pipeline 型ベンチマークアプリケーションの DAG

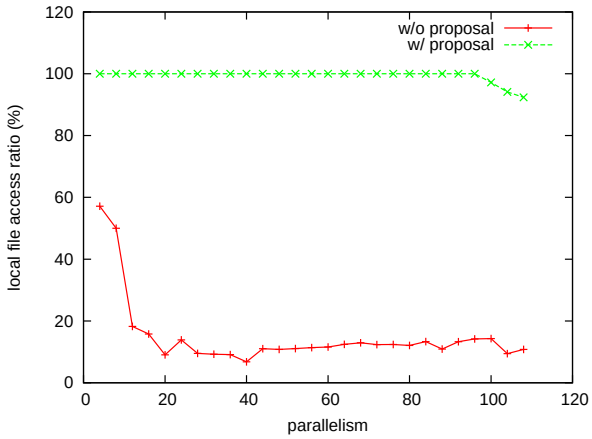

図 6.3. ローカルファイルアクセス比率（Pipeline）

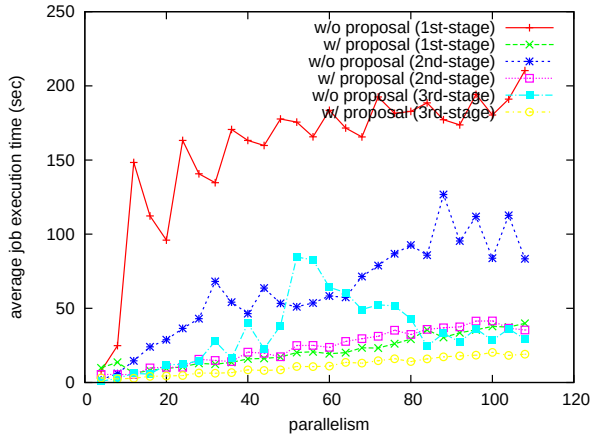

図 6.4. ジョブの平均実行時間（Pipeline）

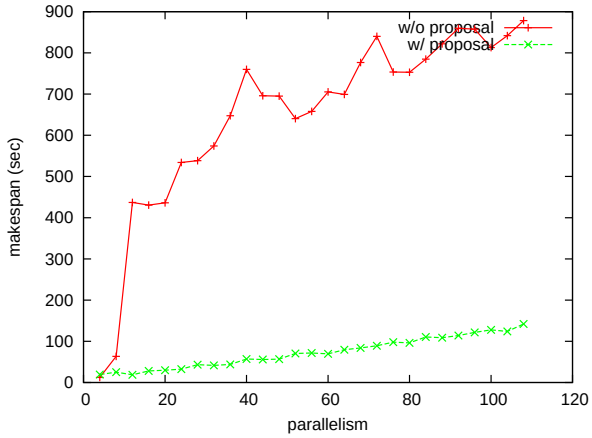

図 6.5. Makespan（Pipeline）

ブスケジューリングそれぞれの，ファイルアクセス中のローカルファイルアクセスの比率，全ジョブの平均実行時間，ワークフロー実行開始から，全てのジョブが終了するまでにかかった時間（これはしばしば *Makespan* と呼ばれる [32]．以降では，各ジョブの実行時間と，ワークフロー全体の実行時間の混同を避けるため，Makespan と標記する）を示す．提案手法を用い，データローカリティの高いジョブスケジューリングを行うことで，ローカルファイルアクセス比率は期待通り 100%まで向上している．その結果，データを読み込むだけの単純なジョブでは，ジョブの平均実行時間を約 55 秒から約 10 秒に，Makespan を約 150 秒から約 57 秒に短縮することができている．実際のアプリケーションではジョブはファイルを読み込むだけではなくその他の処理を様々に行うため，これほどの効果は期待できないことが多い．しかし，基本的にはデータローカリティを向上させることにより，ジョブの実行時間時間短縮が可能なことを本実験により示した．

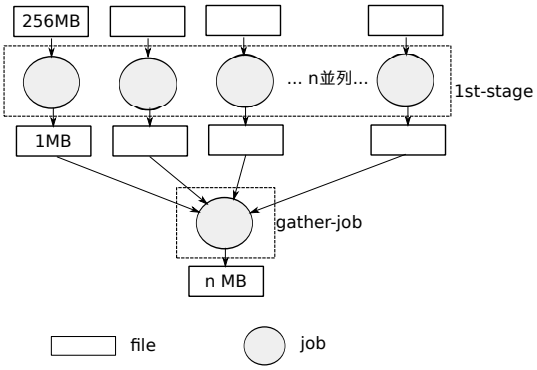

図 6.6. Gather 型ベンチマークアプリケーションの DAG

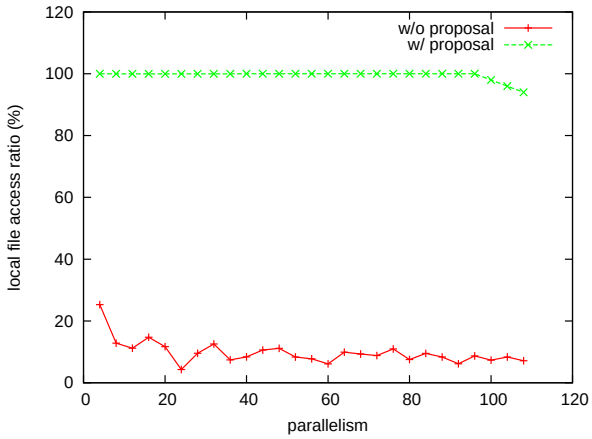

図 6.7. ローカルファイルアクセス比率（Gather）

6.2.2　ベンチマークアプリケーションによるデータ転送効率化手法の効果

本項では，以下二種類の擬似的に作成したベンチマークアプリケーションを準備し提案手法の評価を行う．評価には，huscs クラスタのノード 5 台，hongo クラスタのノード 4 台，kyoto クラスタのノード 4 台を用いた．huscs クラスタのノードの内，1 台を Mogami のメタデータサーバ，他の 1 台を GXP のマスターノードとしてセットアップを行う．残りの huscs クラスタのノード 3 台，hongo クラスタのノード 4 台，kyoto クラスタのノード 4 台を GXP のワーカノードとする．ジョブの実行は，各ノードで 8 個まで，全体で 96 個まで同時に並列で実行できるよう設定した．各実験の結果は 10 回の実行の平均を算出し，評価に用いる．

Pipeline

Pipeline 型ベンチマークアプリケーションの DAG を図 6.2 に示す．このベンチマークアプリケーションでは，ジョブはあるファイルを読み，その半分の量を新規作成したファイルに出力する．従って，パイプラインの段階を図中に示すように，1st-stage，2nd-stage，3rd-stage とすると，1st-stage から 3rd-stage に向かう過程で，ジョブのファイルアクセス量は段階的に少なくなる．評価は，パイプラインの並列数 N を変化させて実行にかかる時間を測定し，評価を行う．

評価実験中に行われたファイルアクセスのローカルファイルアクセスの比率を図 6.3，各ジョブの実行時間を段階毎に平均した結果を図 6.4，アプリケーション Makespan を図 6.5 に示す．実験結果からは，提案手法を用いることで，ほぼ全てのネットワークを介したリモートアクセスを削減できたことがわかる．特にジョブが飽和していない $N \leq 96$ の状態では，全てのファイルアクセスがローカルノードに対するものとなっている．それにより実行時間に関しても，ジョブの実行時間平均，Makespan，共に大幅に削減できていることが確認できる．ある程度大きな N を用いた実行では，Makespan が約 83%程削減できている．

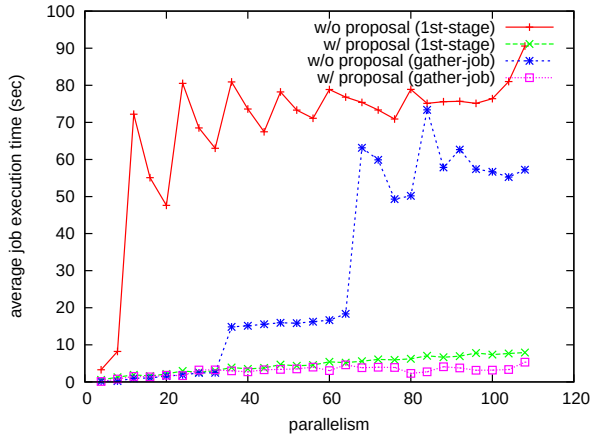

図 6.8. ジョブの平均実行時間（Gather）

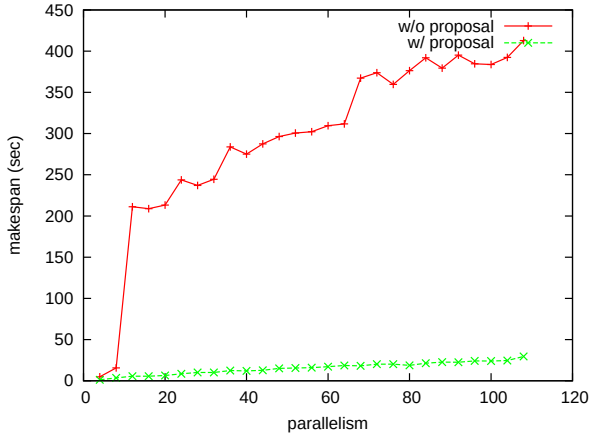

図 6.9. Makespan（Gather）

Gather

Gather 型ベンチマークアプリケーションの DAG を図 6.6 に示す．このベンチマークアプリケーションでは，図中 1st-stage とされるジョブが，各入力ファイル（512MB）の内容を 1MB に集約し（この実験ではただ最後の 1MB をそのまま出力する），最後に gather-job とされるジョブが全結果を集約するアプリケーションを模擬的に表している．

評価実験中に行われたファイルアクセスのローカルファイルアクセスの比率を図 6.7，各ジョブの実行時間を段階毎に平均した結果を図 6.8，アプリケーション Makespan を図 6.9 に示す．実験結果からは，提案手法を用いることで，ほぼ全てのネットワークを介したリモートアクセスを削減できたことがわかる．パイプラインのものと同じく，特にジョブが飽和していない $N \leq 96$ の状態では，全てのファイルアクセスがローカルノードに対するものとなっている．また，それによりジョブの実行時間平均，Makespan，共に大幅に削減できていることが確認できる．ある程度より大きな N を用いた実行では，Makespan が約 94%程短くなっていることがわかる．gather-job の平均実行時間が，既存手法より短くなっていることから，レプリケーションを用いたデータの予備転送も，効果を発揮していることがいえる．

6.3　高遅延環境におけるデータ転送効率化手法評価

本節では，第 5 章で述べた提案手法に対する評価を行う．まず，6.3.1, 6.3.2 において，アクセスパターン検知・分類手法を用いたファイルデータの先読みの効果を評価する．6.3.3 で，適応的に先読みデータ量を自動調節するための手法を評価する．また，最後に 6.3.4 で，複数プロセスによる提案手法を用いた同時アクセス性能の評価を行う．

用いた環境は，huscs クラスタ，tsukuba クラスタ，kyoto クラスタである．Mogami 上でそれぞれファイルを作成し（ファイルコンテンツデータは huscs のノード上に位置），tsukuba, kyoto クラスタのノードから読み込む．ファイルはデータサーバ上の OS ファイルキャッシュに存在する状態で実験を行い，ディスクの帯域に律速されることはない．Mogami のメタデータサーバは huscs クラスタ内のノードとする．高遅延環境で通信を行う際，Linux 標準の TCP 送信受信バッファサイズで通信

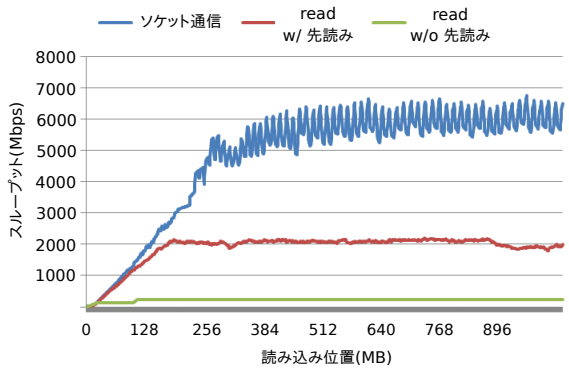

図 6.10. シーケンシャルアクセスの読み込みスループット (huscs → tsukuba)

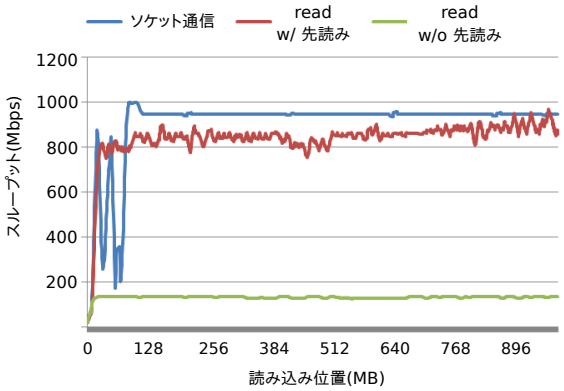

図 6.11. シーケンシャルアクセスの読み込みスループット (huscs → kyoto)

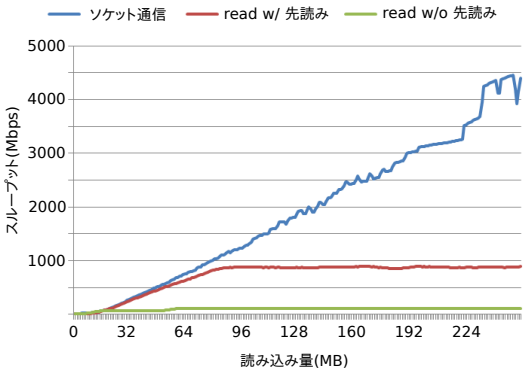

図 6.12. ストライドアクセスの読み込みスループット (huscs → tsukuba)

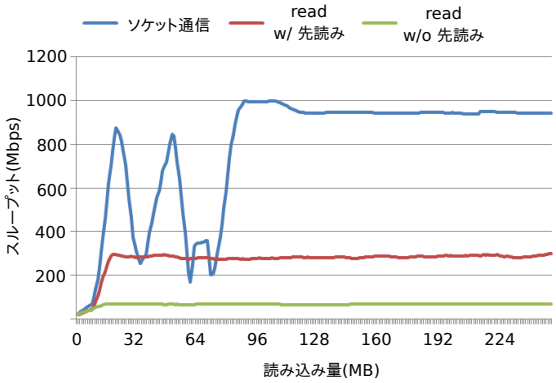

図 6.13. ストライドアクセスの読み込みスループット (huscs → kyoto)

を行うと，バッファあふれによりパケットロスが起こり，スループットが十分に大きくならないことがある．そのため，本実験では各サーバの TCP 送信受信バッファを十分に大きくとって行う．具体的には，`sysctl` コマンドにより，`net.core.wmem_default`, `net.core.rmem_default` を 1MB, `net.core.wmem_max`, `net.core.rmem_max` を 1GB，`net.ipv4.tcp_mem`, `net.ipv4.tcp_rmem`, `net.ipv4.tcp_wmem` を ‘‘`4kB 1MB 1GB`’’ に設定する．使用する TCP 輻輳制御アルゴリズムは Linux 標準の TCP-CUBIC である．

6.3.1　シーケンシャル・ストライドアクセスの先読み効果の評価

提案アクセスパターン検知手法を用いて行う先読みの効果を測定するために，シーケンシャルアクセス，ストライドアクセスそれぞれに関して，ファイルの読み込みスループットを測定した．

読み込むファイルは 1GB の大きさのファイルである．シーケンシャルアクセスではファイルを 1MB ずつ順番に読み込み，ストライドアクセスでは 1MB 読み込み，3MB 読み飛ばすという処理をファイルの末尾まで繰り返し行い，1MB 読み込むごとのスループットを測定した．

評価は，実装した分散ファイルシステムの先読みを無効化したもの，先読みを有効にして手動設定により先読み要求量を決定したもの（huscs → tsukuba 環境では 25MB, huscs → kyoto 環境では 10MB に設定），単にソケット通信でデータを送り続ける簡易プログラムの比較で行う．ソケット通信のみのものは理論上可能な最大スループットを与えるためのものであり，先読みを無効化したものと，先読みを有効にして手動設定により先読み要求量を決定したものの比較により，先読み効果の評価を行う．この評価のシーケンシャルアクセスの結果を図 6.10，図 6.11 に，ストライドアクセスの結果を図 6.12，図 6.13 に示す．

まずシーケンシャルアクセスに関して，先読みを行うことによってどちらの環境でも読み込み性能が大幅に改善されていることが分かる．しかし，図 6.11 ではソケット通信の結果で示された理想スループットに近いアクセススループットが得られているが，図 6.10 ではこのスループットの 1/3 程度のスループットしか得られていない．

また，ストライドアクセスに関しては，先読みにより大幅にアクセス性能が向上することを確認できたが，シーケンシャルアクセスのときの読み込みスループットには達していない．うまくアクセス予測ができていればシーケンシャルアクセスとほぼ同じスループットが得られるはずである．これに関して調査を行った結果，これは FUSE の先読み機能が引き起こしていることが分かった．FUSE にはデフォルト設定で先読みの機能がついており，アプリケーションが要求したところの少し先までデータ要求が起こる．そのため，アプリケーションが 1MB の大きさのデータ要求を行ったとき，その少し先までのデータ読み込みが行われる．評価用分散ファイルシステムは 1MB の大きさのブロックでデータを転送・管理するため，結果的にアプリケーションが要求した 1MB とその次の 1MB の転送が行われることとなり，スループットが約半分になっていた．FUSE の先読み機能はオプションで無効化することが可能だが，FUSE の先読み機能には，FUSE モジュールによる内部通信，メモリコピー等の先読みを行える利点がある．これを無効にしてしまうと，いくらデータを先読みしてもスループットが大きく低下するため，簡単に改善することは難しい．

6.3.2　可変なアクセスパターン時のアクセスパターン検知手法の評価

次に，5.2 節で述べたアクセスパターン検出手法による先読みの精度を，可変パターンのアクセスを行い評価する．ここでは，前項と同様に 1GB のファイルを読み込む．はじめに 128MB シーケンシャルに読み込み，次の 128MB を読み飛ばす．そしてその先の 128MB で，1MB の大きさ read, 3MB の大きさ seek を続けるストライドアクセスで読み込み，次の 128MB を読み飛ばす．ここで 128MB ずつ読み飛ばしているのは次のアクセスパターンに遷移する際に以前先読みを行ったところをアクセスしないためである．ここまでのアクセス全体を 2 回繰り返し，ファイルの先頭から末尾までアクセスした際の，1MB 毎の read スループットを測定する．ここでは先読みサイズの調整を，提案手法による自動調整で行う．この実験を huscs → tsukuba 間で行った結果を図 6.14 に示し，この実験中の提案手法による帯域推定と先読み要求量の推移を図 6.15 に示す．

図 6.14 の赤丸で示した部分のみ，先読みされていないデータへのアクセスが起こっており，その部分のスループットは大きく低下している．しかし，それ以外の部分はうまくアクセスパター

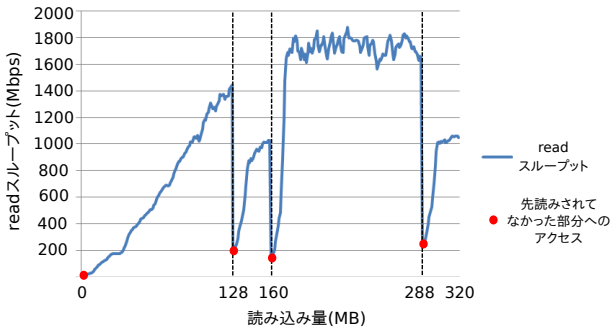

図 6.14. アクセスパターン可変時の読み込みスループット

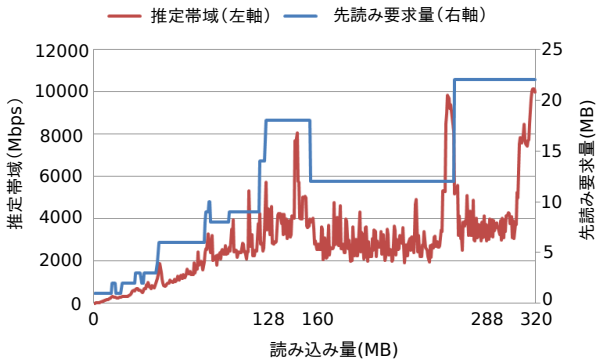

図 6.15. アクセスパターン可変時の推定帯域と先読み要求量

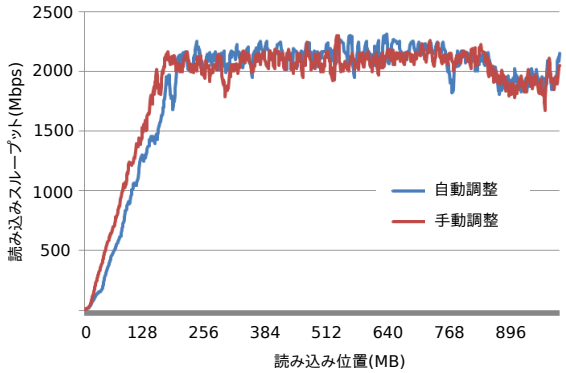

図 6.16. 先読み要求量自動調整手法の評価（huscs→tsukuba）

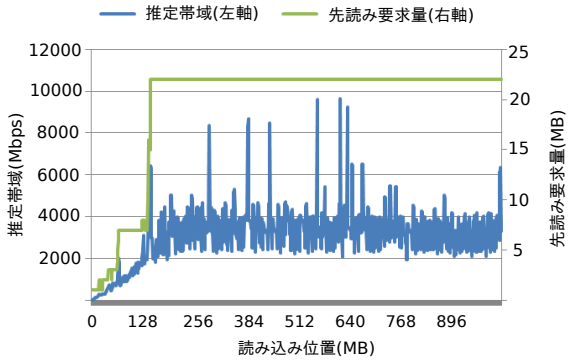

図 6.17. 推定帯域と先読み要求量（huscs→tsukuba）

ンの検出ができ，先読みが行えていることを示している．

6.3.3　先読み要求量自動調整による効果

次に，提案手法による先読み要求量の自動調整手法を評価する．評価は 6.3 と同じく 1GB のファイルを，シーケンシャルに読み込んだときの読み込みスループットを測定する．評価対象として，先読み要求量を手動で調整した場合と，自動調整を行った場合を比較する．また自動調整を行ったときのデータ受信用スレッドによる推定帯域と，そこから計算した先読み要求量も測定を行う．huscs → tsukuba で評価を行った結果を図 6.16，図 6.17 に，huscs → kyoto で行った結果を図 6.18，図 6.19 に示す．

どちらの環境でも提案手法による自動先読み量調整手法は，管理者が手動でパラメタを設定した場合と比べて，大きくアクセススループットを落とすようなことはなかった．ただし，huscs → tsukuba では先読み要求量が増加するのが比較的遅く，ファイルの先頭付近のアクセススループッ

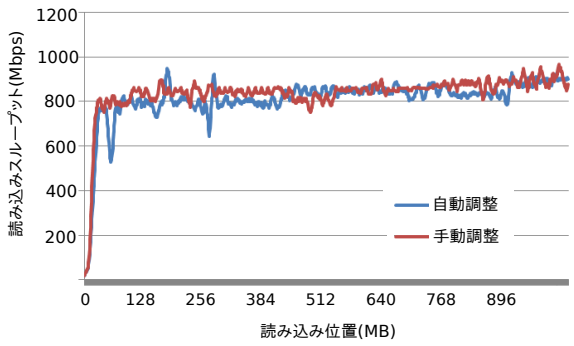

図 6.18. 先読み要求量自動調整手法の評価（huscs→kyoto）

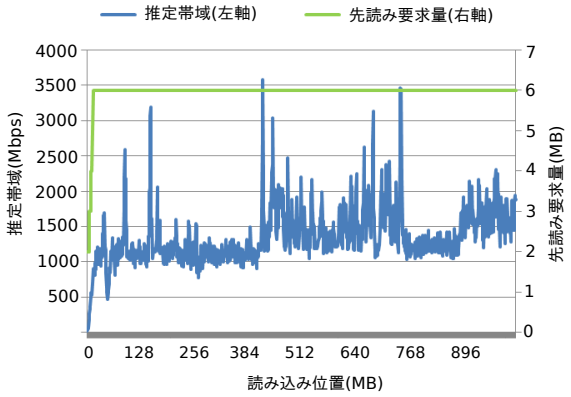

図 6.19. 推定帯域と先読み要求量（huscs→kyoto）

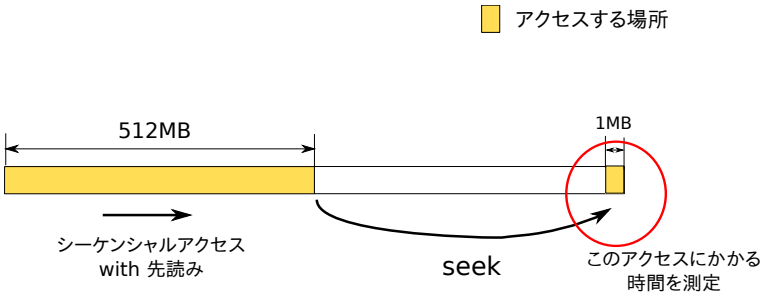

図 6.20. 先読みモード切り替えの性能評価

トが少し低い．つまり，高遅延環境で小さなファイルを読み込む場合は広帯域なネットワーク性能を活かした通信を行うところまで至らないことが分かる．従って，ファイル先頭を読み込むときの帯域推定アルゴリズムに関してはもう少し改良の余地があると思える．

先読みモード切り替え時の性能

次に先読みを行う際に生じるリスクに関して，提案手法を用いて先読み要求量を自動調整する場合と，先読みを無効化した場合を比較して評価する．先読みが外れたのちのデータ転送の遅延を測定するため，1GB の大きさのファイルを用意し，はじめに 512MB をシーケンシャルにアクセスし，その後ファイル末尾 1MB にアクセスを行うときに読み込みにかかる時間を計測する（図 6.20）．実験を huscs → tsukuba で行った結果を図 6.21, huscs → kyoto で行った結果を図 6.22 に示している．

まずどちらの環境においても提案手法を用いて先読みを行った場合のほうが，連続領域をアクセスしている途中で他の場所をアクセスしたときのアクセスにかかる時間が短くなっている．提案手法では最悪 RTT 分の遅延を想定していたが，実際は先読みを行っているときのほうがネットワーク帯域が出ている状態であるため，先読みを外して無駄なデータ転送が生じても，この実験

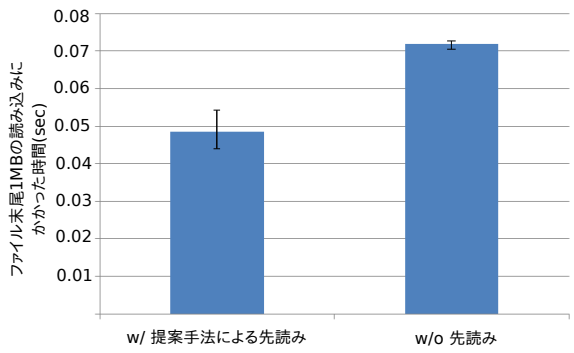

図 6.21. 先読みモード切り替え時のリスク評価（huscs→tsukuba）

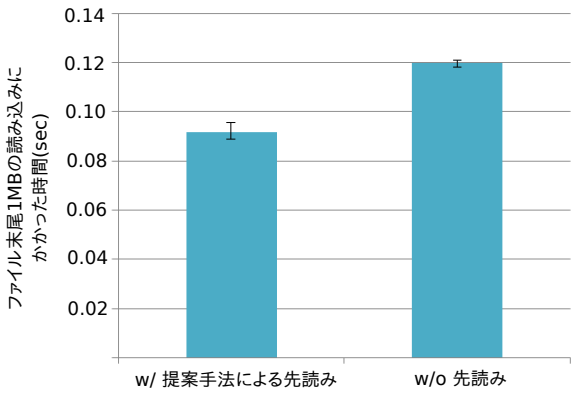

図 6.22. 先読みモード切り替え時のリスク評価（huscs→kyoto）

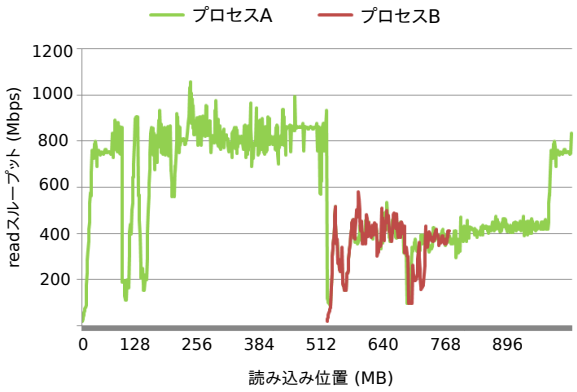

図 6.23. 複数プロセスでの読み込みスループット

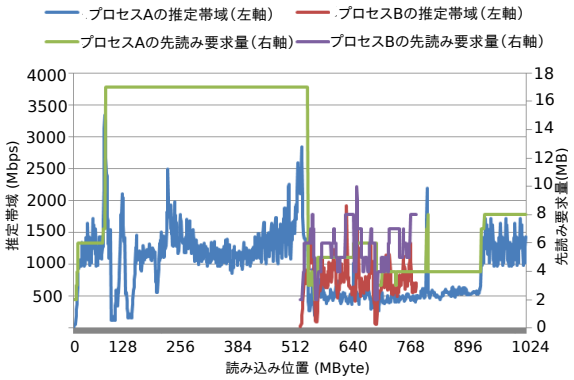

図 6.24. 複数プロセスからの推定帯域と先読み要求量

環境では短い時間でデータ転送が行えている．

6.3.4　複数プロセスからの同時アクセス性能

最後に提案手法による適応的な先読みを，複数プロセスで実行した場合のファイル読み込みスループットを測定する．1GB のファイルを用意し，あるプロセス（プロセス A とする）はシーケンシャルに 1MB ずつファイルを読み込む．先頭 512MB の読み込みが終わった直後にもう一つプロセスを立てて（プロセス B とする），プロセス A と同時に同じファイルを読み込む．プロセス B はファイルのオフセット 512MB の位置から 256MB を読み込んだのちに終了する．これらの読み込み 1MB 毎に読み込み時間を測定し，そこからアクセススループットを算出したものを図 6.23 に示す．また，この実験中の提案手法による帯域推定と先読み要求量の推移を図 6.24 に示す．

プロセス A と B の間では同時アクセスが行われている間ほとんどスループットが同じ状態を遷移しており，限られた帯域を共有してデータ転送が行えていることが分かる．しかし，プロセス B が読み込みを終了してからプロセス A が元のスループットまで回復するまでにかなりの時間を要している．このことからも一旦スループットが落ちたあとに先読み量を増やしていく部分のアルゴリズムは改良の余地が残っていると思われる．

第7章　評価II：実アプリケーションを用いた提案システム総合評価

本章では，実際に実用化されている実アプリケーションを用い，提案システムの総合的な評価を行う．

7.1　評価に用いるアプリケーションと環境

以降の評価では，実用化されているアプリケーションとして，Similar Pages Extraction, Japanese Word Count, Supernovae Explosion Detection, Case Frame Construction[29, 35], Montage[5] の 5 つのアプリケーションを用いる．これらのアプリケーションの特徴を表 7.1 にまとめる．

評価では，評価 I で述べた実験環境と同じく InTrigger のクラスタで huscs, hongo, kyoto1 の 3 つのクラスタを用いる．huscs クラスタのあるノードを Mogami のメタデータサーバとし，別のあるノードを GXP のマスタサーバとする．それら以外のノードで huscs クラスタ 15 ノード，hongo クラスタ 4 ノード，kyoto1 クラスタを 7 ノードを GXP のワーカノードとして用いる．GXP ではマスタノードもジョブを実行するワーカノードを兼ねるため，マスタノードとワーカノードの計 27 ノードでジョブを実行し，各ノードはジョブを 8 つまで並列実行可能な設定にする．従って，以降の実験では同時に 216 個のジョブが並列実行可能となる．実験時は，はじめに全てのデータ（実行プログラムを含む）をファイルコンテンツが GXP のワーカノードに位置するよう設置し，実験を行う．実際にワークフローアプリケーションが用いられる際には，ユーザが保持しているデータをまとめて分散ファイルシステムにコピーすることが考えられるため，今回はこの手法を取っている．しかし，実運用としては，実行プログラムを除く入力データセットが全ノードを分散している場合も考えられる．

表 7.1. 使用アプリケーションの種類と特徴

	Similar Pages Extraction	Japanese Word Count	Supernovae Explosion Detection	Case Frame Construction	Montage
CPU 負荷	中	中	高	中	低
I/O 量	中	高	中	高	高
Read(R)/Write(W) 量	R >> W	R > W	R > W	R ≈ W	R > W
使用分野	Web 解析	テキスト解析	天文学	自然言語処理	天文学

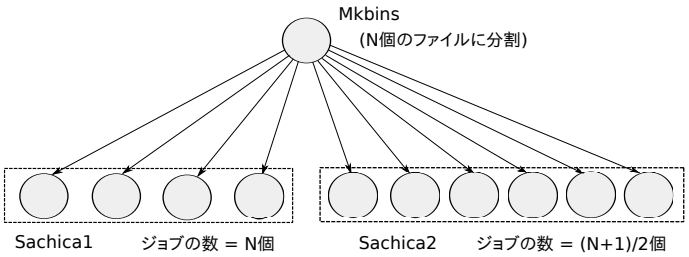

図 7.1. Similar Pages Extraction の DAG

表 7.2. Similar Pages Extraction の使用データセット

	data（小）	data（大）
Input ファイルサイズ	69B ~ 7.5kB	16MB ~ 37MB
Input ファイルの個数	100	100
合計 Input データサイズ	1.4MB	2.4GB
ジョブの個数	466	466

表 7.3. Similar Pages Extraction の入力ファイル予測精度

Application	Number of Jobs	Precision (%)	Recall (%)
Mkbins	1	100 (102/102)	100 (102/102)
Sachica1	30	100 (60/60)	100 (60/60)
Sachica2	435	100 (1740/1740)	100 (1740/1740)

7.2　Similar Pages Extraction

Similar Pages Extraction は，Web から収集した大量のテキストデータを入力とし，‘類似した’ 文章を出力するためのアプリケーションである．ここでは，類似した文章として，文章間のハミング距離が指定したしきい値以下のものを抽出する．このワークフローでは，固定長の文字列の集合から，しきい値以下のハミング距離を持つペアを全て高速に列挙するためのアルゴリズム Sachica[45] が使用されている．このアプリケーションの DAG 一例を図 7.1 に示す．ジョブは大きく分けて 3 種類に分類され，それぞれを `Mkbins`, `Sachica1`, `Sachica2` とする．このアプリケーションでは，Mkbins で全ての入力ファイルを一度まとめ，N（ユーザが指定，本実験ではデフォルト値の 30 に設定）個のファイルに分割する．その後，`Sachica1` があるファイル内の類似文を全て列挙，`Sachica2` が 2 つのファイル間の類似文を全て列挙する．

7.2.1　データセット

本実験で用いたデータセットを表 7.2 に示す．データセット小を用いた実行でファイルアクセス履歴を取得し，データセット大の実行により提案手法の評価を行う．

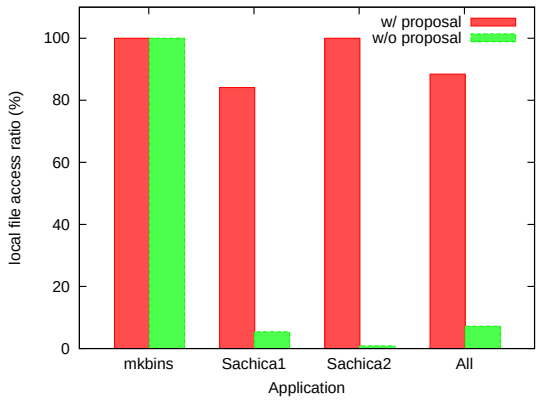

図 7.2. Similar Pages Extraction のローカルファイルアクセス比率

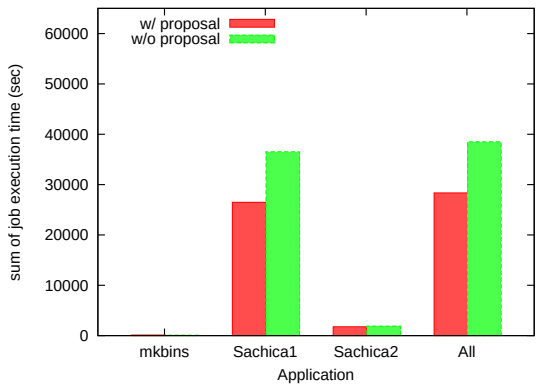

図 7.3. Similar Pages Extraction のジョブ実行時間合計

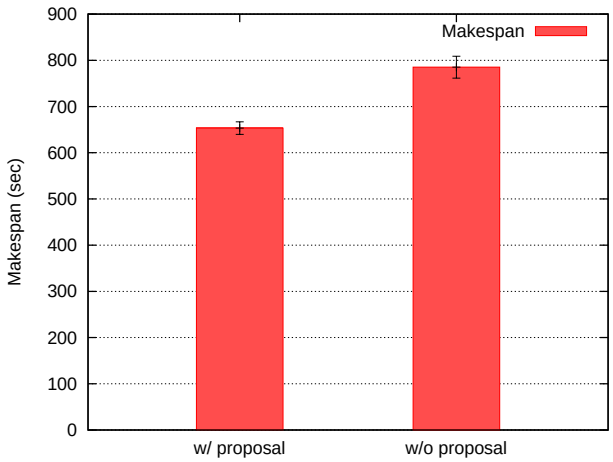

図 7.4. Similar Pages Extraction の Makespan

7.2.2　実験結果

提案したジョブの入力ファイル予測手法の正確さの評価結果を表 7.3 に示す．このアプリケーションでは，提案手法により各ジョブの入出力ファイルが全て予測できている．これは基本的にジョブの種類が少なく，分かりやすい形で引数に入力ファイル名を取っているためである．提案手法を用いた実行と，用いない実行の，全ファイルアクセス中のファイルアクセス比率の比較を図 7.2，ジョブ実行時間の単純合計の比較を図 7.3，Makespan の比較を図 7.4 に示す．

提案手法によりローカルアクセス比率が全体で約 7%から約 88%に向上，全ジョブ実行時間の合計が約 38500 秒から約 28300 秒に短縮された．その結果，Makespan は 785 秒から 653 秒に短くなり，実行時間の約 16%が削減できた．

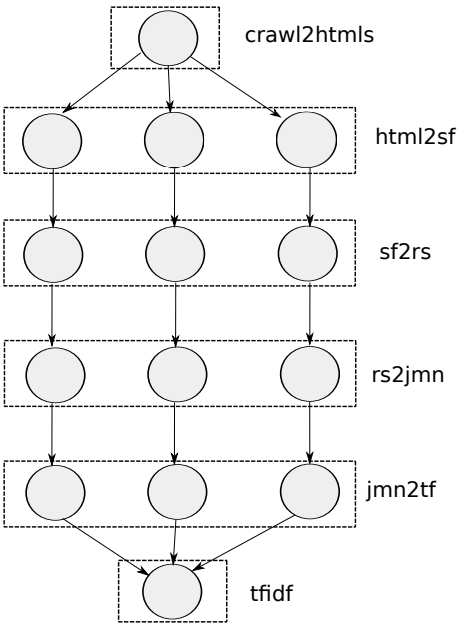

図 7.5. Japanese Word Count の DAG

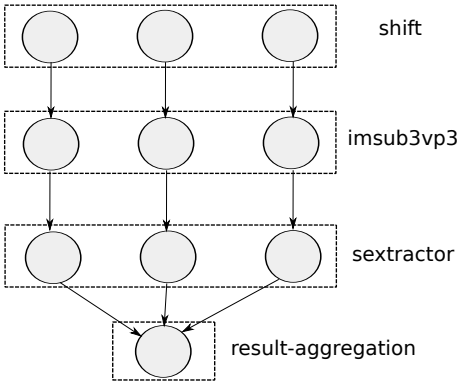

図 7.6. Supernovae Explosion Detection の DAG

7.2.3　考察

Similar Pages Extraction では，`Sachica1` で 2 種類のファイルの中の，`Sachic2` で 1 つのファイル中の文章を比較する．このアプリケーションでは，アクセスファイルの予測は非常に高い精度で行われているものの，`Sachica1` で考慮すべき入力ファイルが 2 つ存在するため，そのままではローカルファイルアクセス率があまり向上しない．しかし，自動レプリケーションの機能によりレプリケーションが行われ，結果的に 88%のファイルがローカルから読み込まれている．レプリケーションの機能を無効としたときのローカルファイルアクセス率は 25%弱，Makespan の削減は約 6%にとどまった．この結果により，レプリケーションによる効果が確認できた．

7.3　Japanese Word Count

Japanese Word Cound は，大量の日本語 Web コーパスを解析し，日本語の単語の出現数を勘定するアプリケーションである．元来，Word Count アプリケーションは，文章からキーワードやキーフレーズを抽出し，ホットな話題を特定したりするため，広く用いられている．本アプリケーションは，入力ファイルに含まれる単語の tf-idf[34] を計算する．用いられるアプリケーションは `crawl2htmls`, `html2sf`，`sf2rs`，`rs2jmn`，`jmn2tf`，`tfidf` の 5 種類で，その DAG を図 7.5 に示す．本アプリケーションは，それぞれの HTML ファイルから tf を計算するところまでが完全に並列実行可能な構成となっており，それらの結果を集約して最後に idf を計算する．

表 7.4. Japanese Word Count の使用データセット

	data（小）	data（大）
Input ファイルサイズ	902kB	212MB
Input ファイルの個数	1	1
合計 Input データサイズ	902kB	212MB
ジョブの個数	10	1550

表 7.5. Supernovae Explosion Detection の使用データセット

	data（小）	data（大）
Input ファイルサイズ	34B or 41MB	34B or 33MB
Input ファイルの個数	3	1501
合計 Input データサイズ	81MB	35GB
ジョブの個数	15	7001

表 7.6. Japanese Word Count の入力ファイル予測精度

Application	Number of Jobs	Precision (%)	Recall (%)
crawl2htmls	1	100 (2/2)	100 (2/2)
html2sf	386	100 (1158/1158)	100 (1158/1158)
sf2rs	386	52.59 (406/772)	100 (406/406)
rs2jmn	386	100 (772/772)	100 (772/772)
jmn2tf	386	67.1 (518/772)	100 (518/518)
tfidf	1	100 (387/387)	100 (387/387)

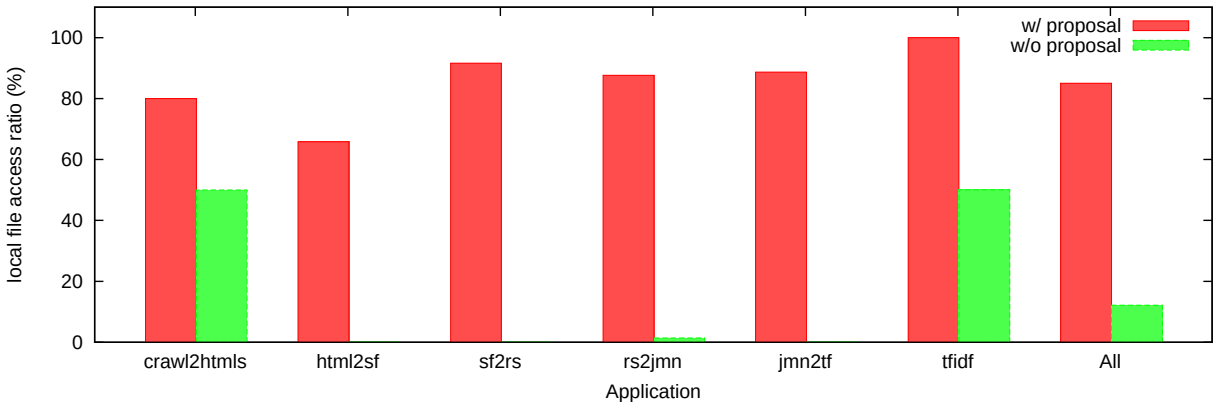

図 7.7. Japanese Word Count のローカルファイルアクセス比率

7.3.1　データセット

本実験で用いたデータセットを表 7.4 に示す．データセット小を用いた実行でファイルアクセス履歴を取得し，データセット大の実行により提案手法の評価を行う．

7.3.2　実験結果

提案したジョブの入力ファイル予測手法の正確さの評価結果を表 7.6 に示す．Japanese Word Count の各ジョブは，`crawl2htmls` と `tfidf` を除き基本的に 1 つのファイルを入力として，1 つのファイルを出力するものである．そのようなジョブの入力ファイルは提案手法により予測できる場合が多く，高い予測精度を示している．提案手法を用いた実行と，用いない実行の，全ファイ

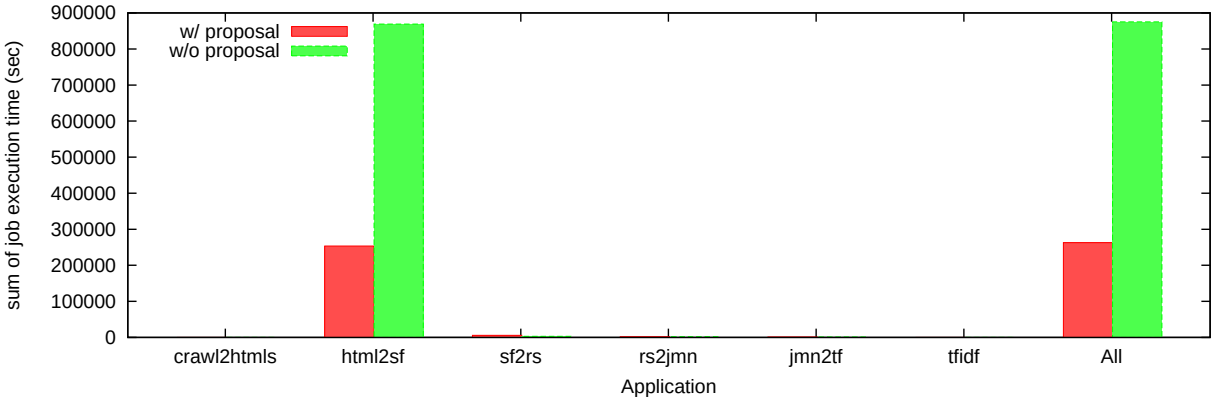

図 7.8. Japanese Word Count のジョブ実行時間合計

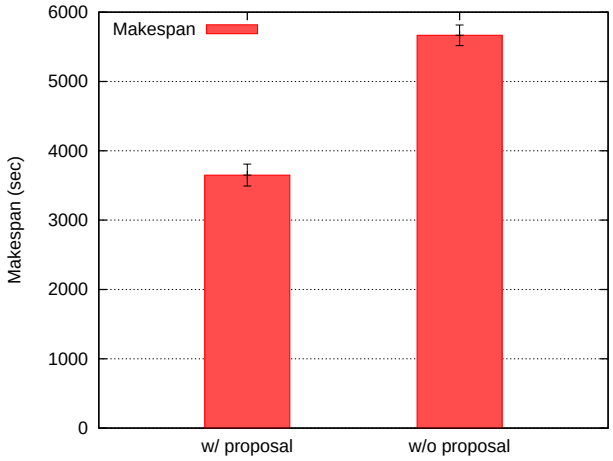

図 7.9. Japanese Word Count の Makespan

ルアクセス中のファイルアクセス比率の比較を図 7.7，ジョブ実行時間の単純合計の比較を図 7.8，Makespan の比較を図 7.9 に示す．提案手法によりローカルアクセス比率が全体で約 12%から約 87%に向上，全ジョブ実行時間の合計が約 875000 秒から約 360000 秒に短縮された．その結果，Makespan は 5666 秒から 3649 秒に短くなり，実行時間の約 35%が削減できた．

7.3.3　考察

Japanese Word Count で，最も大きな実行時間を占めているのは，`html2sf` である．このジョブでは，数多くの Perl プロセスが作成され，Perl のモジュールや共通設定用ファイルの `open`, `read`, `stat` が非常に多く行われる．その数はデータにより様々であるが，1 度 Perl のプロセスが生成される際に少なくとも 20 以上の小さいファイルが読み込まれる．そのため特にメタデータサーバからネットワーク的に離れたノードでジョブが実行された場合，メタデータサーバとの通信にも多くの時間が費やされる．図 7.10 に，提案手法の内いくつかの機能を無効にしたときの Makespan の

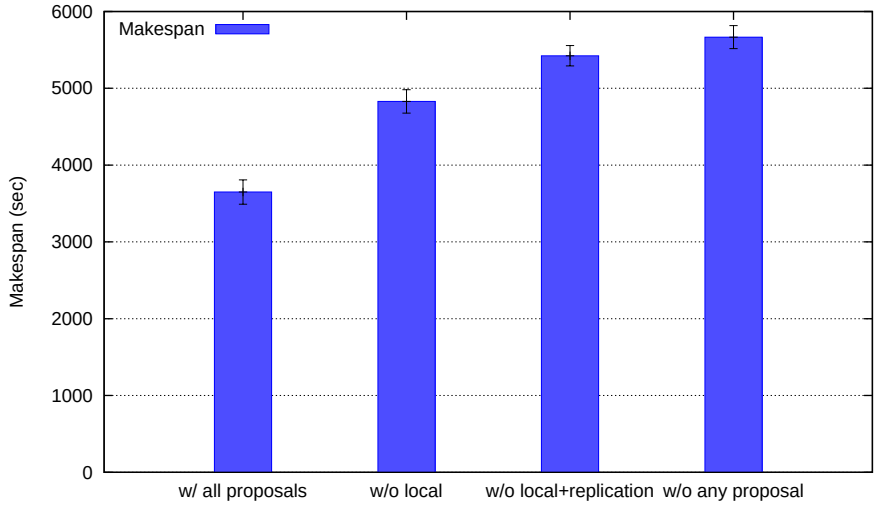

図 7.10. Japanese Word Count の提案手法の効果比較

比較を示す．図中 “w/ all proposals” は全ての機能を有効にしたもの，“w/o local” はメタデータのローカル探索（4.4.3 項）のみ無効にしたもの，“w/o local+replication” はメタデータのローカル探索機能と，効率的な自動レプリケーション機能（4.4.2 項）を無効にしたもの，“w/o any proposal” は既存のジョブスケジューラによるものを示している．図 7.10 から，最も Makespan を削減しているのは，メタデータのローカル探索機能であり，それだけで 1200 秒近くの Makespan を短縮を行っていることがわかる．これは，メタデータサーバがシステム内で単一の存在であり，ボトルネックとなっていることが考えられる．もちろんこの機能が効果を発揮するためには，ローカルに存在するファイルにアクセスしなければならないため，自動レプリケーションなどのデータローカリティを向上させる他の提案手法も重要であることは間違いない．

7.4　Supernovae Explosion Detection

Supernovae Explosion Detection は天文学のアプリケーションで，大量の天文画像データから，超新星と思われる星の座標を列挙する．図 7.11 は，すばる望遠鏡で撮影された複数画像を用い，超新星を発見する例である．Supernovae Explosion Detection は IEEE Cluster/Grid 2008 でデータ解析チャレンジ問題として出題された[1]．ワークフローの DAG を図 7.6 に示す．

[1] http://www.cluster2008.org/challenge/

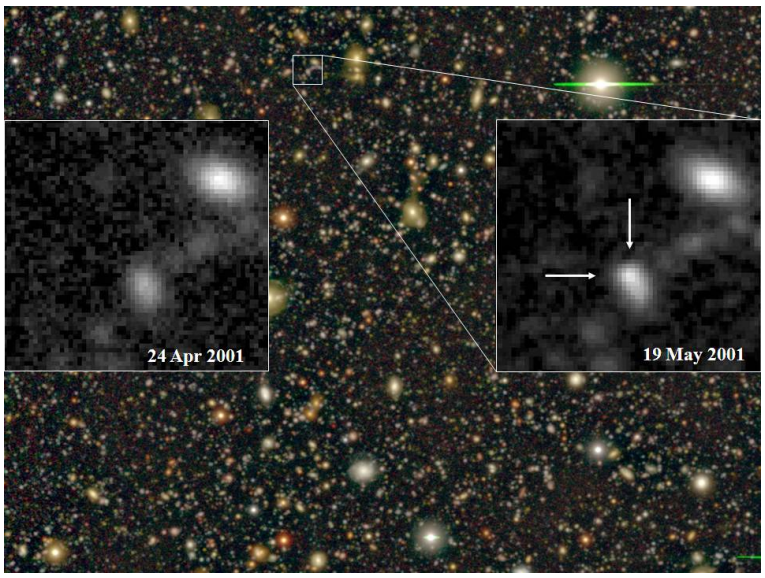

図 7.11. 画像比較による超新星発見例（IEEE Cluster/Grid 2008 HP より引用）

表 7.7. Supernovae Explosion Detection の入力ファイル予測精度

Application	Number of Jobs	Precision (%)	Recall (%)
shift	4000	84.29 (10734/12734)	100 (10734/10734)
imsub3vp3	1500	100 (12552/12552)	97.666 (12552/12852)
sextractor	1500	81.924 (3943/4813)	100 (3943/3943)
result-aggregation	1	100 (497/497)	100 (497/497)

7.4.1　データセット

本実験で用いたデータセットを表 7.5 に示す．データセット小を用いた実行でファイルアクセス履歴を取得し，データセット大の実行により提案手法の評価を行う．

7.4.2　実験結果

提案したジョブの入力ファイル予測手法の正確さの評価結果を表 7.7 に示す．また，提案手法を用いた実行と，用いない実行の，全ファイルアクセス中のファイルアクセス比率の比較を図 7.12，ジョブ実行時間の単純合計の比較を図 7.13，Makespan の比較を図 7.14 に示す．提案手法によりローカルアクセス比率が全体で約 8%から約 40%に向上，全ジョブ実行時間の合計が約 162000 秒から約 159300 秒に短縮された．その結果，Makespan は 2063 秒から 1844 秒に短くなり，実行時間の約 10%が削減できた．

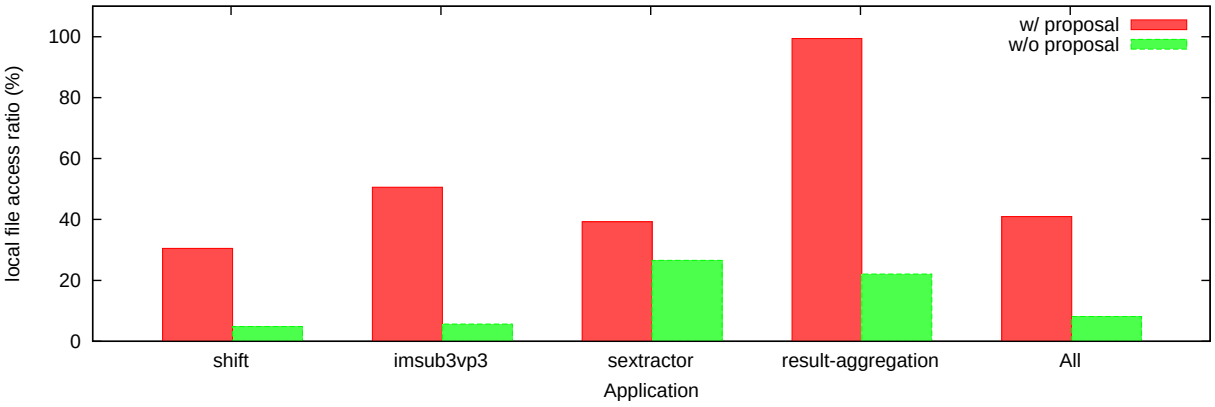

図 7.12. Supernovae Explosion Detection のローカルファイルアクセス比率

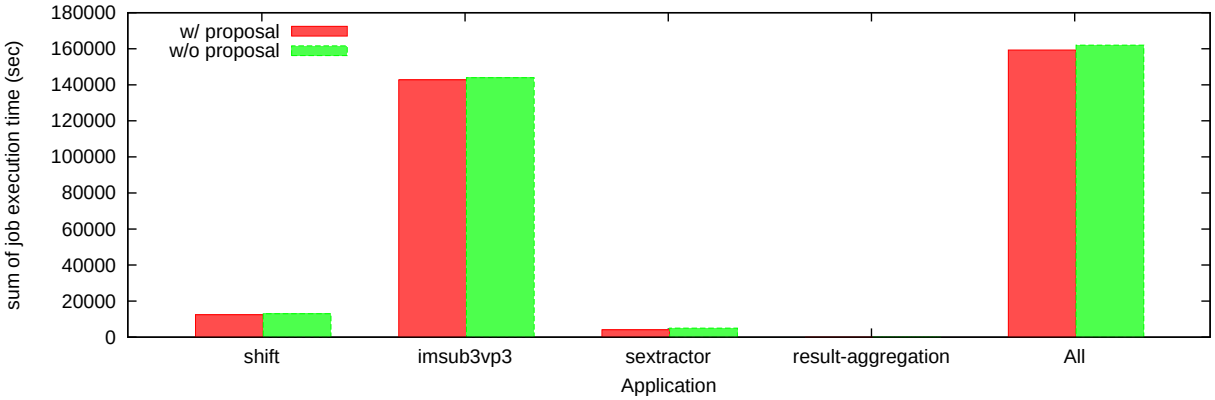

図 7.13. Supernovae Explosion Detection のジョブ実行時間合計

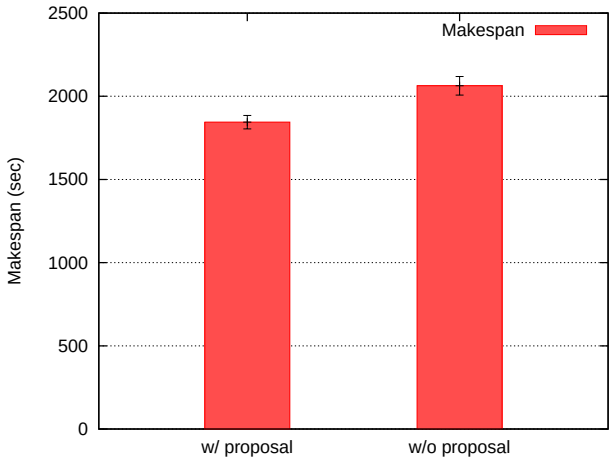

図 7.14. Supernovae Explosion Detection の Makespan

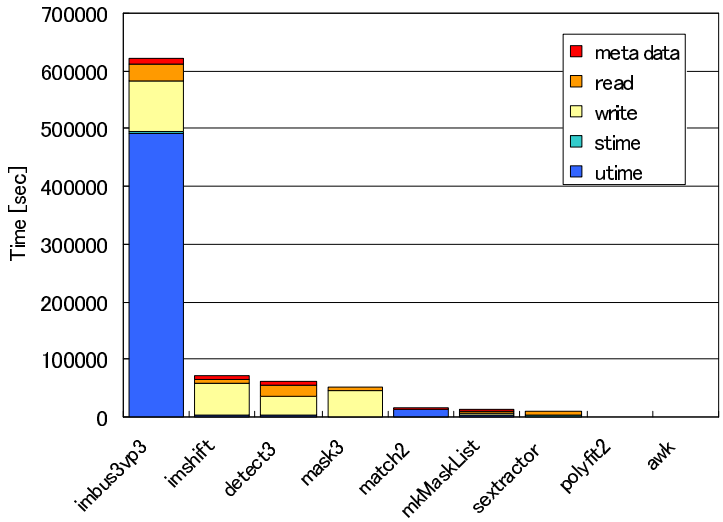

図 7.15. Supernovae Explosion Detection のジョブ実行時間内訳 [38]

7.4.3　考察

Supernovae Explosion Detection では，`imsub3vp3` というアプリケーションが大部分のジョブ実行時間を占めている．しかしこのジョブは極めて計算インテンシブなものであり，実行時間のほとんどを CPU 使用時間が占めている．図 7.15 に，柴田らによるこのアプリケーション実行時間の内訳を示す．このため，ファイルアクセスの時間を削減しても実行時間はほとんど変化しない．結果として Makespan の短縮率も小さくなり，提案手法の有するファイルアクセスにかかる時間を削減する利点があまり活かせていない．

7.5　Case Frame Construction

Case Frame Construction は，Web 上の日本語文章から**格フレーム** *(case frame)*[29, 35] と呼ばれるデータ構造を構築するための自然言語処理分野のワークフローアプリケーションである．格フレームとは，頻出する名詞と動詞の格ごとの共起を記述したもので，構築した格フレームは，検索，要約，翻訳などの様々なテキスト解析アプリケーションに応用できる．ワークフローの DAG 一例を図 7.16 に示す．本アプリケーションで用いられるジョブは，以下の 5 つのに大きく分類できる．

1. juman-and-knp: 日本語のウェブコーパスを JUMAN や KNP[29] を用いて解析する
2. mk-predicate-arg: 解析結果を用い，述語項構造を抽出する
3. aggr-and-divi: 一度抽出した述語項構造をまとめ，用言ごとにデータを分割する
4. mk-caseframes: 用言ごとに抽出された述語項構造から，格フレームを構築する
5. mk-final-dict: 構築された全ての格フレームをまとめ，最終的な格フレームの辞書を構築する

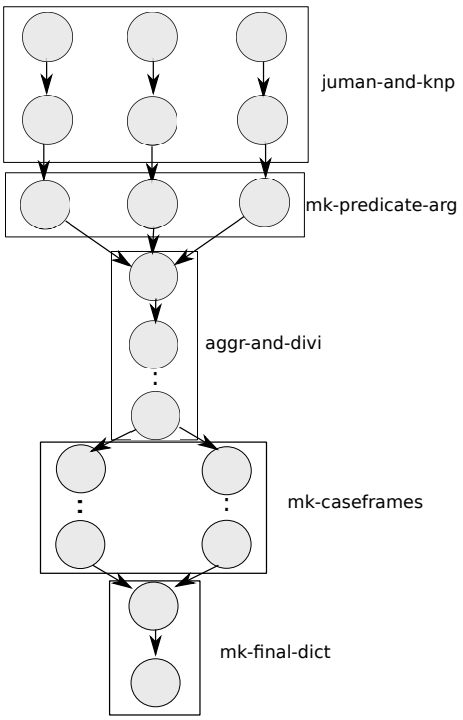

図 7.16. Case Frame Construction の DAG

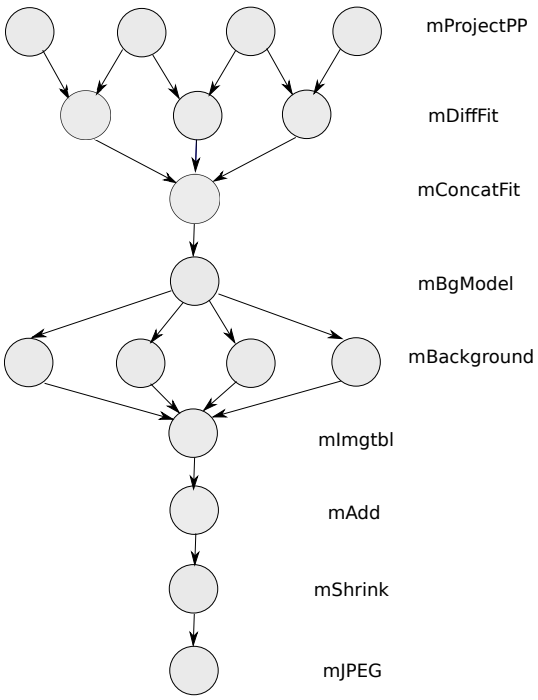

図 7.17. Montage の DAG

表 7.8. Case Frame Construction の使用データセット

	data（小）	data（大）
Input ファイルサイズ	約 400kB	約 1.8MB
Input ファイルの個数	4	24
合計 Input データサイズ	1.7MB	39MB
ジョブの個数	45	1600

表 7.9. Montage の使用データセット

	data（小）	data（大）
Input ファイルサイズ	2.1MB	1.7MB or 2.1MB
Input ファイルの個数	6	609
合計 Input データサイズ	12.6MB	1.24GB
ジョブの個数	19	1542

7.5.1　データセット

本実験で用いたデータセットを表 7.8 に示す．データセット小を用いた実行でファイルアクセス履歴を取得し，データセット大の実行により提案手法の評価を行う．

7.5.2　実験結果

提案したジョブの入力ファイル予測手法の正確さの評価結果を表 7.10 に示す．また，提案手法を用いた実行と，用いない実行の，全ファイルアクセス中のファイルアクセス比率の比較を図 7.18，ジョブ実行時間の単純合計の比較を図 7.19，Makespan の比較を図 7.20 に示す．提案手法によりローカルアクセス比率が全体で約 65%から約 99%に向上，全ジョブ実行時間の合計が約 38900 秒から約 16000 秒に短縮された．その結果，Makespan は 3149 秒から 2706 秒に短くなり，実行時間の約 14%が削減できた．

表 7.10. Case Frame Construction の入力ファイル予測の精度

Application	Number of Jobs	Precision (%)	Recall (%)
juman-and-knp	48	100 (362/362)	95.263 (362/380)
mk-predicate-arg	24	97.959 (144/147)	100 (144/144)
aggr-and-divi	5	86.667 (52/60)	91.228 (52/57)
mk-caseframes	1521	100 (27714/27714)	100 (27714/27714)
mk-final-dict	2	100 (104/104)	97.196 (104/107)

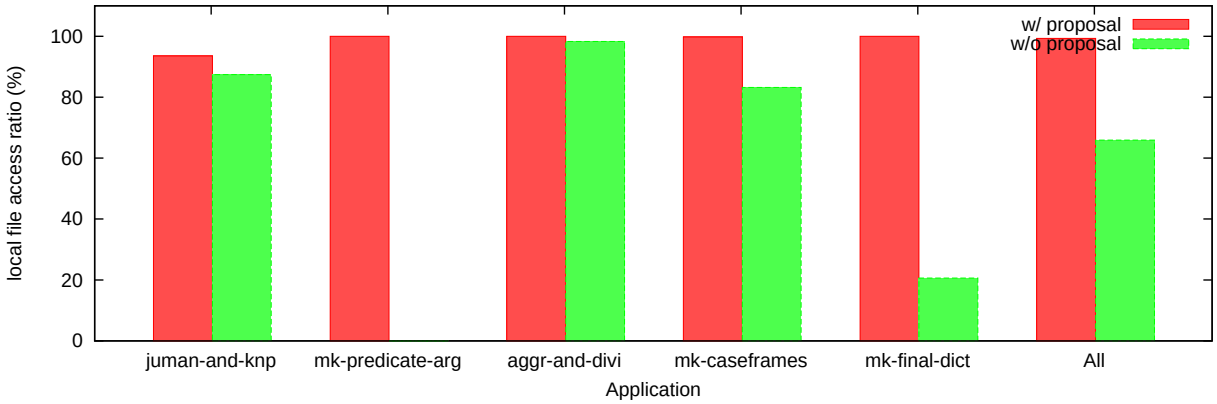

図 7.18. Case Frame Construction のローカルファイルアクセス比率

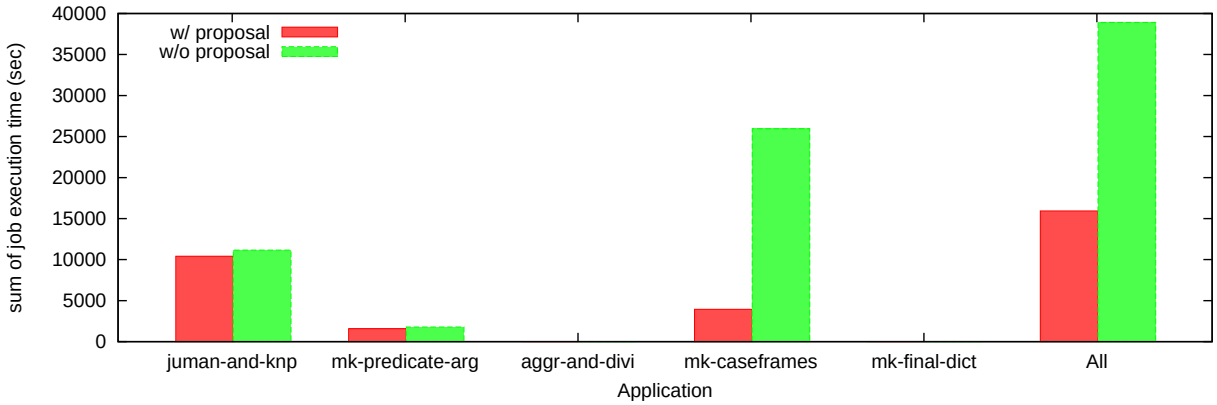

図 7.19. Case Frame Construction のジョブ実行時間合計

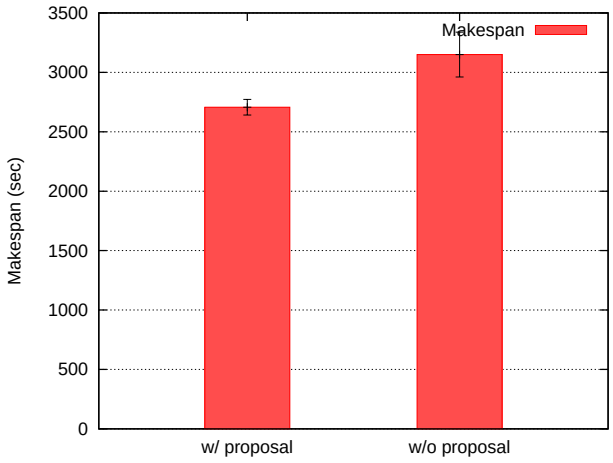

図 7.20. Case Frame Construction の Makespan

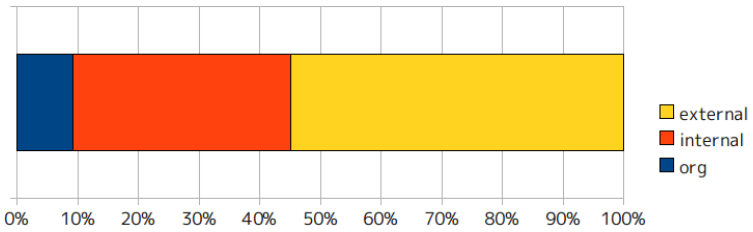

図 7.21. Case Frame Construction のファイルアクセス分類

7.5.3　考察

Case Frame Construction では，他のアプリケーションに比べて元々のローカルファイルアクセス比率が高い．この理由としては，特に juman-and-knp などのジョブで，自身が作成し，書き込みを行ったファイルの読み込みを多く行っていることが挙げられる．図 7.21 に，アプリケーション中のファイル読み込みを以下の 3 種類に分類した時の内訳を示す．

input 初期アプリケーション入力ファイルの読み込み．アプリケーション実行時に存在しているファイルへの読み込み．

internal あるジョブで作成され，作成したジョブ自身が書き込み・読み込みを行うファイルへの読み込み．

external 自分以外のジョブが作成したファイルの読み込み．主にワークフローアプリケーションではジョブ間通信のために使われる．通常のワークフローアプリケーションでは多くのファイルアクセスがこのタイプに分類される．

表 7.11. Montage の Input ファイル予測の精度

Application	Number of Jobs	Precision (%)	Recall (%)
mProjectPP	308	99.84 (1230/1232)	100 (1230/1230)
mDiffFit	913	55.88 (6437/11520)	88.01 (6437/7314)
mConcatFit	1	100 (914/914)	100 (914/914)
mBgModel	1	100 (2/2)	100 (2/2)
mBackground	308	99.89 (1846/1848)	100 (1846/1846)
mImgtbl	1	100 (6/6)	100 (6/6)
mAdd	5	100 (720/720)	100 (720/720)
mShrink	4	50 (4/8)	100 (4/4)
mJPEG	1	100 (1/1)	100 (1/1)

図 7.21 から，Case Frame Construction では全体の 35.7%のファイル読み込みが internal のタイプであることがわかる．Mogami では，ファイルの作成時には，作成を行ったノードに優先してデータコンテンツが配置されるポリシーになっている．そのため，それらのファイルアクセスは常にローカルファイルアクセスとなる．これらは特に注意を払わなくともローカルファイルアクセスとなるため，それ以外の約 64%のファイルアクセスを提案手法によりローカルファイルアクセスとなるようスケジューリングが行えたこととなる．

7.6 Montage

Montage アプリケーションは，天文学分野で非常に有名なアプリケーションで，複数画像を組み合わせて一つのモザイクイメージを作成するために用いられる．`mProjectPP`, `DiffFit`, `mConcatFit`, `mBgModel`, `Background`, `mImgtbl`, `mAdd`, `mShrink`, `mJPEG` の九種類の C 言語で書かれたプログラムによって構成され，それぞれの依存関係は図 7.17 に示す DAG で表現される．Montage アプリケーションは複雑なファイルアクセスを行い，アプリケーション中のファイルアクセスにかかる時間比率が高いことが知られている．

7.6.1 データセット

実験で用いるデータセットを表 7.9 に示す．表中のデータセット小を用いた実行でファイルアクセス履歴を取得し，データセット大の実行により提案手法の評価を行う．本実験で評価する実行では，全体で約 28GB バイトのファイルアクセスを行う．

7.6.2 実験結果

提案したジョブの入力ファイル予測手法の正確さの評価結果を表 7.11 に示す．提案手法を用い，ほとんどのアプリケーションでは入力ファイル予測が正確に行われている．これらの中には，ジョブがアクセスファイルが書かれた設定ファイルを読み込み，その内容に応じたファイルアクセス

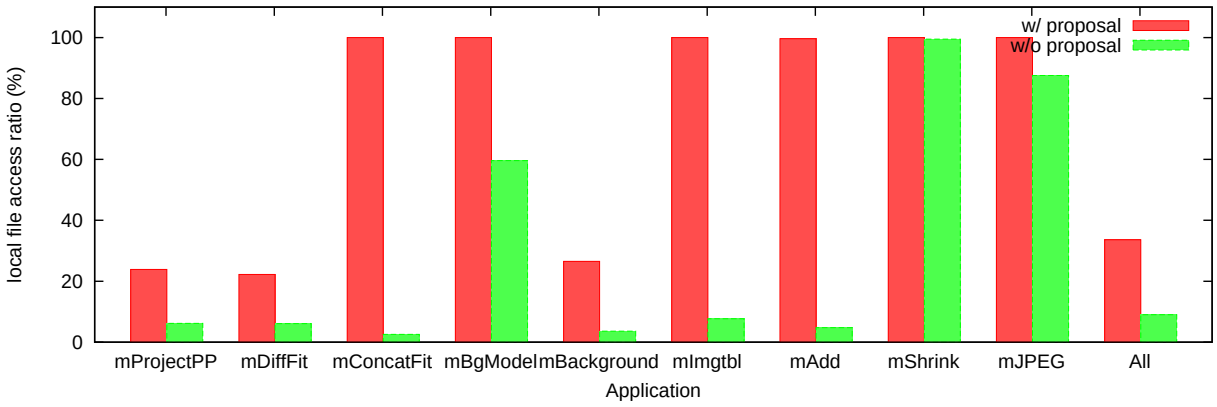

図 7.22. Montage のローカルファイルアクセス比率

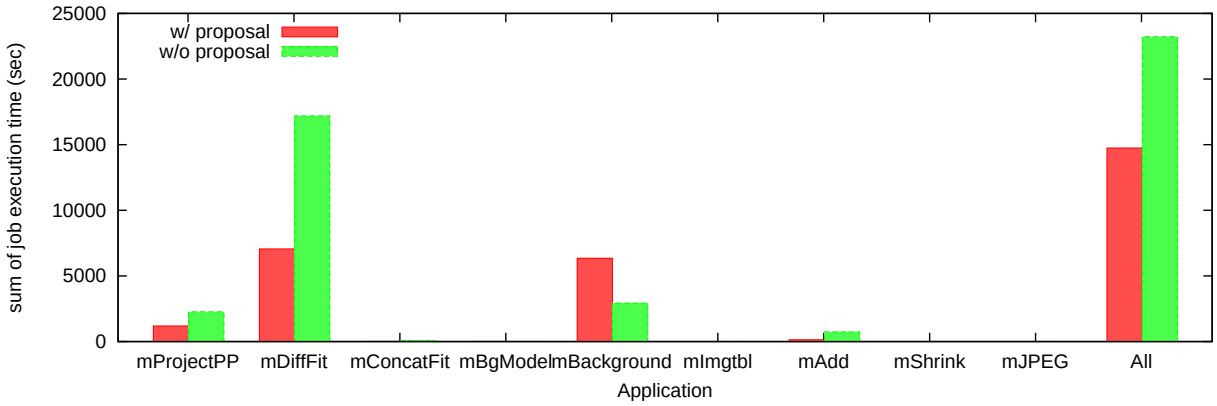

図 7.23. Montage のジョブ実行時間合計

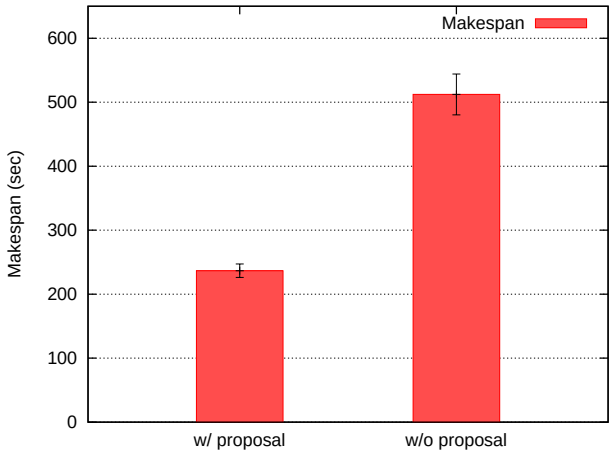

図 7.24. Montage の Makespan

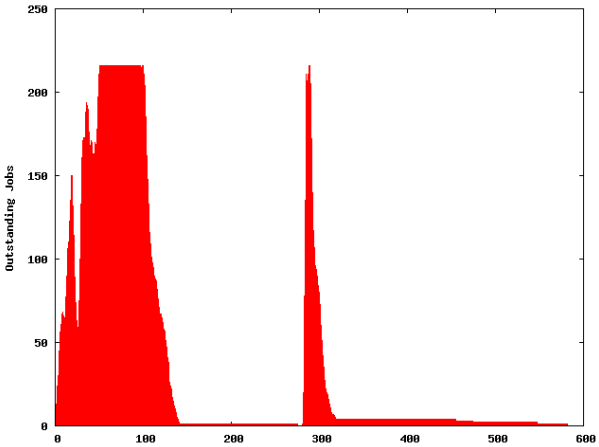

図 7.25. Montage 実行時のジョブの並列度（提案手法無し）

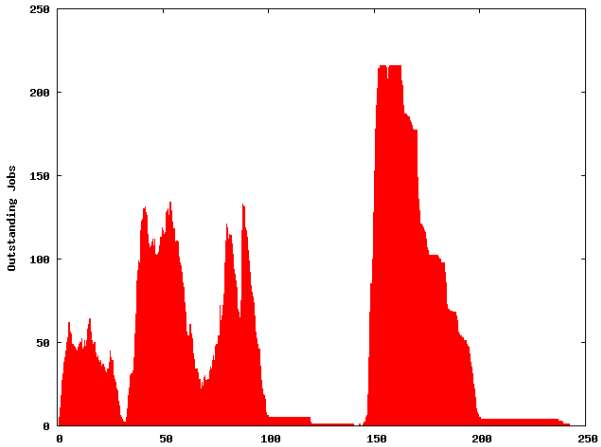

図 7.26. Montage 実行時のジョブの並列度（提案手法有り）

を行うものも含まれている．提案手法を用いた実行と，用いない実行の，全ファイルアクセス中のファイルアクセス比率の比較を図 7.22，ジョブ実行時間の単純合計の比較を図 7.23，Makespan の比較を図 7.24 に示す．提案手法によりローカルアクセス比率が全体で約 8%から約 33%に向上，全ジョブ実行時間の合計が約 23200 秒から約 14800 秒に短縮された．その結果，Makespan は約 510 秒から約 240 秒に短くなり，実行時間の 50%以上の時間が削減できた．

7.6.3　考察

Montage アプリケーションでは，ジョブの実行時間，ジョブの個数，ファイルアクセス量に関して `mDiffFit` が最も多くを占めている．提案手法では，そのファイルアクセスをできるだけネットワークを介さず行うことで，`mDiffFit` のジョブ実行時間合計を約 17200 秒から約 7000 秒に大幅に削減した．これはグラフを見る限りでは，最も大きな貢献であり，Makespan の削減はほとんどこれが原因であるように思える．しかし，実際は `mConcatFit` や `mAdd` の実行時間短縮も，大きな貢献となっている．これらのジョブは，数多くの小さなファイルをシーケンシャルに読み込み，データの集約処理を行うもので，アプリケーション全体で `mConcatFit` は 1 個，`mAdd` は 5 個しかジョブが存在しない．また，これらのジョブが実行されている間は他のジョブは実行されず，アプリケーションの中でも非常に並列度の低い部分である．ジョブの実行時間合計を示すグラフ（図 7.23）で見ると，ジョブ自体の数が少なく提案手法の効果は確認できないものの，実際は `mConcatFit` は 73.01 秒から 11.94 秒，`mAdd` は 746.5 秒から 138.49 秒に短縮されている．図 7.25, 図 7.26 に，Montage の実行中のジョブ並列度を示す．両者の横軸の最大値が全くことなるため，見た目を単純に比較することはできない．図 7.25 中のおおよそ 150 ~ 250 秒, 図 7.26 中のおおよそ 120 ~ 145 秒の範囲が `mConcatFit` の実行されていた時間を示す．また，図 7.25 中のおおよそ 310 ~ 580 秒, 図 7.26 中のおおよそ 200 ~ 245 秒の範囲が `mAdd` の実行されていた時間を示す．単純比較はできないものの，並列度の見た目から，明らかにこれらのアプリケーションの実行時間削減が Makespan の削減に寄与したことがわかる．これらのジョブの高速化には，データを予め

ジョブの実行予定ノードにレプリケーションしておく手法が，有効に働いたと考えられる．そのレプリケーションは，mDiffFit, Background の実行中に行われる．特に mBackground は，比較的ジョブの実行時間が短いため，レプリケーションを同時に行うことで影響を受け実行時間が延びてしまっている．しかし，それに見合った効果が全体で得られていると考える．

第8章　結論

8.1　まとめ

本論文では，データ集約的アプリケーションを分散環境で効率的に行うためのシステムを提案した．提案システムには，既存のワークフロー実行エンジン GXP Make を用い，実行時のデータ管理用分散ファイルシステム **Mogami** と，Mogami を用いたデータ転送の効率化手法の提案，実装，評価を行った．

本論文で提案したワークフローアプリケーション実行のためのシステムは，以下の有用性を有している．

1. 簡易性 1：ユーザはワークフローアプリケーションの各ジョブ用実行プログラムとその依存関係を記述するだけで，分散環境に確保した資源を用いて並列計算を行うことができる
2. 簡易性 2：ノード間でのファイル共有には POSIX 互換の分散ファイルシステムを用い，アプリケーションから共有ファイルへは透過的なアクセスが行える．そのため，アプリケーションの開発者は既存の逐次プログラムを書き換えることなくワークフローアプリケーションでジョブ用プログラムとして用いることができる
3. 自動効率化 1：各ジョブ間のデータの流れ（各ジョブの共有ファイルへの読み書き）をプログラマが記述することなく，自動でノード間データ転送の効率化を行う
4. 自動効率化 2：ファイルデータ転送の際，高遅延環境を介した通信が発生した場合にも，アクセスパターンを検出して適応的な先読みを自動で行う

以上のため，ユーザやアプリケーションの開発者にとっては敷居が低く，かつユーザの負担を最小限に抑えた性能最適化を行うシステムを構築することができた．

評価においては，まず各提案手法それぞれに対しての有用性を評価した．ファイルアクセス履歴を用いたデータ転送最適化手法では単純なベンチマークを用意し，評価を行った．複数クラスタに属するノードを用いた評価では，全体実行時間を約 83 ~ 94 %程度削減することにも成功した．高遅延環境を用いた適応的な先読みの評価では，高遅延広帯域なネットワークで接続されたノード間通信でシーケンシャルアクセスの際に約 700 ~ 800 %，ストライドアクセスの際に約 300 ~ 400 %のスループット向上を確認することができた．

また，実アプリケーションによる評価では，Similar Pages Extraction, Japanese Word Count, Supernovae Explosion Detection, Case Frame Construction, Montage の 5 つのアプリケーションを用いて提案システムの全体の評価を行った．評価結果では，ファイルアクセスよりも計算に時間を費やすアプリケーションではそれほどの差がでなかったものの，Montage などのファイルアクセスにかかる時間がジョブ実行時間の大半を占めるアプリケーションでは，全体実行時間を約 50%に削減するなどの効果が確認できた．

8.2　今後の展望

最後に，今後の展望として考えられることをまとめる．本研究の目的は，ワークフローアプリケションの実行をユーザへの負担を増やすことなく自動効率化することであった．そのために，計算のデータローカリティを向上させせることで，計算時間の短縮を図った．しかし，実際にはジョブの計算時間は様々な要因により変化する．例えば CPU の性能差，ネットワーク環境（遅延，帯域共に重要），ストレージの性能などに起因して，ジョブの実行時間は増減する．これらの影響を全く考慮しないジョブスケジューリング手法では，ヘテロな環境で何かしらの不利益を被る場合がある．また，アプリケーションの特徴によってもどのようなスケジューリングが好ましいかが変化し得る．例えば CPU インテンシブなアプリケーションでは，データのローカリティを考えるより，CPU の性能を考えたほうが処理時間の短縮が期待できる．特に本研究ではワークフローアプリケーションのプロファイリングを一度行っている．そのとき得られた情報を下に，更に効率的なジョブスケジューリングを行うことは可能であると考える．

また，本研究ではメタデータサーバが 1 台のみ存在し，メタデータサーバへの遅延の考慮などは行われていない．現時点では，できるだけ影響の出ないような配置にユーザが意図的にせざるをえず，更なる発展のためにはこの点の改善が期待される．

謝辞

本研究を進めるにあたり，指導教官の田浦健次朗准教授には，日々のミーティングなどを通じ，綿密にご指導いただきました．事あるごとに細かい相談にも応じてもらい，研究の進め方や，時にはプログラムのデバッグなどに付き合っていただいたりしました．卒論生として研究室に配属されたときは，決してプログラミング，並列計算など得意な私ではありませんでしたが，先生の熱意ある指導によってここまでこぎつけることができました．本当にありがとうございました．また，研究室に配属されたときから今まで，数多くの研究室の先輩方に基礎からご指導をいただきました．特に既に卒業した原健太朗先輩や加辺友也先輩には，近しい先輩として本当に細かい質問や相談にも応じていただきました．忙しいときでも快く私の細かい質問にお時間を割いてくださり，それがどんなにありがたいことだったのか，先輩となった今痛感します．横山大作助教授には，卒論生のときから研究室のミーティング等を通じ，研究に対するアドバイスをいただきました．ポスドクの Dun Nan さんには，卒論生のときから，私と同じ分野の研究を今までやっていらした先輩として，数多くのアドバイスと，論文投稿の際には論文の添削などしていただきました．博士課程の Ting Chen さんには，慣れない海外の学会などで一緒に過ごさせていただくなどして，様々なことを教えていただきました．研究室の OB でもありサイボウズ・ラボユースを通じてご指導いただいた星野喬さんには，週 1 回ではありましたが毎回研究の，特に実装面での相談に乗っていただきました．また，先輩方だけではなく，後輩も含めた現研究室のメンバーには日々の生活を送るにあたり非常にお世話になりました．研究室の同期である河野瑛さん，池上克明さんには，日々の生活はもちろん，サーバ管理の仕事などでもお世話になりました．また，現在修士 1 年の中谷翔さんには，研究の細かい悩みや私生活に関することまで幅広く相談に乗っていただきました．最後に，研究生活は，遠い広島の地で生活する両親，妹の支えがあって成り立つものでした．この場を借りて，本研究を支えてくださった皆様に，深くお礼申し上げます．

発表文献

論文誌

- 堀内美希, 田浦健次朗. 広域分散ファイルシステムのための適応的な先読み手法. **情報処理学会論文誌コンピューティングシステム** *(ACS)* **第** *40* **号**, 2012/10.

国際発表（査読あり）

- Miki Horiuchi, Kenjiro Taura. Acceleration of Data-Intensive Workflow Applications by Using File Access History. *The 7th Workshop on Workflows in Support of Large-Scale Science (WORKS2012)*, Salt Lake City, 2012/11.

国内発表（査読あり）

- 堀内美希, 田浦健次朗. Mogami: 高遅延環境において広帯域を達成する分散ファイルシステム. **先進的計算基盤システムシンポジウム** *(SACSIS2011)*, 東京, 2011/5. Poster Publication
- 堀内美希, 田浦健次朗. 広域分散ファイルシステムのための適応的な先読み手法. **先進的計算基盤システムシンポジウム** *(SACSIS2012)*, 神戸, 2012/5.

国内発表（査読なし）

- 堀内美希, 田浦健次朗. Mogami: 高遅延環境において広帯域を達成する分散ファイルシステム. **並列／分散／協調処理に関するサマーワークショップ（***SWoPP2011***）**, 鹿児島, 2011/8.
- 堀内美希, 田浦健次朗. . **並列／分散／協調処理に関するサマーワークショップ** *(SWoPP2012)*, 鳥取, 2012/8.

受賞

- SACSIS2012 優秀若手研究賞．2012/5.
- 第 1 回 ICT プログラミングコンテスト 受賞. 2012/5.

参考文献

[1] Fuse. http://fuse.sourceforge.net/.

[2] Fuse python. http://pypi.python.org/pypi/fuse-python.

[3] Intrigger platform. http://www.intrigger.jp/.

[4] Lustre file system. http://www.lustre.org/.

[5] Montage : An astronomical image mosaic engine. http://montage.ipac.caltech.edu/.

[6] Mosastore. http://mosastore.net.

[7] sshfs. http://fuse.sourceforge.net/sshfs.html.

[8] Torque resource manager. http:www.clusterresources.com/products/torque-resource-manager.php.

[9] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Tuecke, Status Of This Memo, L. Liming, and S. Tuecke. Gridftp: Protocol extensions to ftp for the grid. *GWD-R (Recommendation*, page 3, 2001.

[10] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1 –10, nov. 2008.

[11] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[12] Anirban Chakrabarti, R. A. Dheepak, and Shubhashis Sengupta. Integration of scheduling and replication in data grids. In *Proceedings of the 11th international conference on High Performance Computing*, HiPC'04, pages 375–385, Berlin, Heidelberg, 2004. Springer-Verlag.

[13] Po-Cheng Chen, Jyh-Biau Chang, Yi-Chang Zhuang, Ce-Kuen Shieh, and Tyng-Yeu Liang. Memory-mapped file approach for on-demand data co-allocation on grids. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 300–307, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *Proceedings of the 2008 ACM/IEEE*

conference on Supercomputing, SC '08, pages 40:1–40:10, Piscataway, NJ, USA, 2008. IEEE Press.

[15] Ann Chervenak, Ewa Deelman, Miron Livny, Mei-Hui Su, Rob Schuler, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Data placement for scientific applications in distributed environments. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Chun, Hong-woo, Yoshimasa Tsuruoka, Jin-Dong Kim, Rie Shiba, Naoki Nagata, Teruyoshi Hishiki, and Jun'ichi Tsujii. Extraction of gene-disease relations from medline using domain dictionaries and machine learning. In *The Pacific Symposium on Biocomputing (PSB) Maui, Hawaii, USA*, pages 4–15, January 2006.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Ewa Deelman and Ann Chervenak. Data management challenges of data-intensive scientific workflows. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 687–692, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[20] Frederic Desprez and Antoine Vernois. Simultaneous scheduling of replication and computation for data-intensive applications on the grid. *Journal of Grid Computing*, 4:19–31, 2006. 10.1007/s10723-005-9016-2.

[21] Fangpeng Dong and Selim G. Akl. Technical report no. 2006-504 scheduling algorithms for grid computing: State of the art and open problems, 2006.

[22] Nan Dun, Kenjiro Taura, and Akinori Yonezawa. Fine-grained profiling for data-intensive workflows. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 571–572. IEEE, 2010.

[23] Nan Dun, Kenjiro Taura, and Akinori Yonezawa. Paratrac: A fine-grained profiler for data-intensive workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 37–48. ACM, 2010.

[24] Yunhong Gu and Robert Grossman. Toward efficient and simplified distributed data intensive computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):974–984, June 2011.

[25] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[26] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[27] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Data sharing options for scientific workflows on amazon ec2. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[28] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[29] Daisuke Kawahara, Nobuhiro Kaji, and Sadao Kurohashi. Japanese case structure analysis by unsupervised construction of a case frame dictionary. In *Proceedings of the 18th conference on Computational linguistics - Volume 1*, COLING '00, pages 432–438, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[30] Tevfik Kosar and Mehmet Balman. A new paradigm: Data-aware scheduling in grid computing. *Future Generation Computer Systems*, 25(4):406 – 413, 2009.

[31] Kohei Hiraga Osamu Tatebe and Noriyuki Soda. Gfarm grid file system. *New Generation Computing*, 28, 2010.

[32] Simon Ostermann, Radu Prodan, and Thomas Fahringer. Trace-based characteristics of grid workflows. *From Grids to Service and Pervasive Computing*, 10:191–204, 2008.

[33] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs. *SC Conference*, 0:58, 2001.

[34] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, 2 edition, 1983.

[35] Ryohei Sasano, Daisuke Kawahara, and Sadao Kurohashi. The effect of corpus size on case frame acquisition for discourse analysis. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, pages 521–529, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[36] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceeding of the Conference on File and Storage Technologies*, pages 231–244, January 2002.

[37] S Shepler, B Callaghan, D Robinson, R Thurlow, C Beame, M Eisler, Zambeel Inc, and D Noveck. Nfs version 4 protocol, 2000.

[38] Takeshi Shibata, SungJun Choi, and Kenjiro Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 746–755. ACM, 2010.

[39] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –10, May 2010.

[40] Masahiro Tanaka and Osamu Tatebe. Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 356–359, New York, NY, USA, 2010. ACM.

[41] Masahiro Tanaka and Osamu Tatebe. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *12th International Symposium on Clusters, Cloud, and Grid Computing (CCGrid'12)*, pages 65–72, May 2012.

[42] Kenjiro Taura. GXP: An interactive shell for the grid environment. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '04, pages 59–67, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun'ichi Tsujii. Design and implementation of gxp make – a workflow system based on make. *2010 IEEE Sixth International Conference on e-Science*, pages 214–221, December 2010.

[44] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7:51–72, 2009.

[45] Takeaki Uno. New approach for speeding up enumeration algorithms. In Kyung-Yong Chwa and OscarH. Ibarra, editors, *Algorithms and Computation*, volume 1533 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 1998.

[46] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Costa, Zhao Zhang, Daniel Katz, Michael Wilde, and Matei Ripeanu. A workflow-aware storage system: An opportunity study. In *12th International Symposium on Clusters, Cloud, and Grid Computing (CCGrid'12)*, May 2012.

[47] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[48] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., June 2009.

[49] M. Wilde, I. Foster, K. Iskra, P. Beckman, Zhao Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50 –60, nov. 2009.

[50] Justin M. Wozniak and Michael Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 16–20, New York, NY, USA, 2009. ACM.

[51] Zhao Zhang, Daniel S. Katz, Justin M. Wozniak, Allan Espinosa, and Ian Foster. Design and analysis of data management in scalable parallel scripting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 85:1–85:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] Yong Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199 –206, july 2007.

[53] 伊藤 建志, 大崎 博之, and 今瀬 眞. Gridftp-apt : データ転送プロトコル gridftp の並列 tcp コネクション数調整機構 (インターネットの測定・性能評価技術及び一般). **電子情報通信学会技術研究報告**. *IN,* **情報ネットワーク**, 105(472):19–24, 2005-12-08.

[54] 大辻弘貴 and 建部修見. Non-blocking rpc を用いた遠隔ファイルアクセスの実装と性能評価. **研究報告ハイパフォーマンスコンピューティング（*HPC*）**, 2011-HPC-132(16):1–5, nov 2011.