修　　士　　論　　文

並列分散環境における
プロセス毎の分散ファイルシステムへ
の負荷推定

Estimating Per Process Network IO Load for Distributed Systems

指導教員　**田浦健次朗 准教授**

東京大学 情報理工学系研究科
電子情報学専攻

氏 名　**086410 佐伯勇樹**

提 出 日　**平成22年2月9日**

概 要

近年, グリッドコンピューティングなどのように, ネットワークを用いて多数の計算資源を集めて, 大規模な科学技術計算や, Web テキストなどの膨大なデータの処理を行う技術に対する注目が集まっている. このような環境で, 複数の計算機が協調して一つの大きなタスクを分割して処理を行うような場合, 分割されたタスクの間において, あるタスクで出た結果を次のタスクに利用する, といったようにタスクの間にデータによる依存関係が生じる. そのため, 複数の計算機に存在するプロセスの間でデータの共有を行う必要があるが, その場合において用いられるツールの一つが広域分散ファイルシステムである.

広域分散ファイルシステムでは, ある計算ノードでプロセスがファイルを変更した時, それをファイルサーバで反映させ, その後に他の計算ノードのプロセスがファイルサーバを参照することでデータのやり取りを実現している. このシステムで問題となるのが複数のプロセスが同時に大きな write を行った場合で, ファイルサーバに負荷が集中し, 処理の終了までに時間がかかったり, 他プロセスの read の実行を妨げたりする原因となる. しかし, システムの複雑さから, どのプロセスが原因であるか, ということを容易に特定することはできない. そこで, 本研究では統計的手法を用いて NFS などの広域分散ファイルシステムに対する負荷の割合を推定する方法の提案, 実装を行った. この手法は, 分散ファイルシステムの負荷が大きくなった際にどのプロセスが大きな負荷の原因となっているのか, その原因追究の補助をし, 分散ファイルシステムを用いた並列分散プログラミングにおけるスケジューリングによるパフォーマンスの改善等へつなげることを目的としている. 本稿ではその実装と, 部分的な評価について述べる.

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景と目的

1.1.1　背景

近年, 多数の計算機をネットワークを用いて接続し, 大規模な科学計算や大量の Web テキストの解析などを, 並列・協調させて処理を行う並列分散処理が注目を浴びている. 並列分散処理を扱う環境としては, 1 か所に多数の CPU, メモリを集めて, 専用の通信ポートで接続し, 並列計算をさせる超並列型のスーパーコンピュータが主流であったが, 近年では, コンピュータクラスタのように一つ一つの計算ノードの結合性を超並列型のものと比べゆるくした形態が増え, さらには大規模な計算を行えるよう, LAN 内にとどまらずに WAN も用いたより広域な環境をつなぎ, より計算ノード数の大きな環境を扱える形態も登場してきている.

そのうちの一つとしてあげられるのがグリッドコンピューティングである. グリッドコンピューティングでは, インターネットのような広域ネットワーク上で分散しているコンピュータ資源を使い, 一つのタスクを行う手法である. 日本では, InTrigger プラットフォーム [6] がその代表的な例の一つで, 日本各地に分散したコンピュータクラスタが SINET によってつながれており, 環境全体で 1000 コアを超える CPU を用いて計算を行うことができる. 近年では, たくさんの利用者が一度に利用でき, 計算規模に応じてより柔軟に使う資源の量を自動で制御してくれるクラウドコンピューティングといった新たなシステムにもグリッドコンピューティングが使われるようになってきており, 多数の利用者が別々の目的でその大規模な環境を同時に使うような状況も増え, 今後の分散処理の分野として重要となっていく分野の一つであると言える.

そのような分散環境において, 並列処理をさせる際にパフォーマンスを決定する一つの重要な要素がデータのやり取りである. 並列処理では, あるタスクを分割して分散環境にある計算ノードに並列に実行させる, という処理を行うが, その分割したタスクに必要なデータを各計算ノードに分散させたり, タスクを終了して出た結果を各計算ノードから集める, というような処理を行う必要がある. また, あるタスクの結果が別のタスクの実行に依存しており, その 2 つのタスクが別々の計算ノードに割り振られた場合は, その依存関係を作っているデータを計算ノード間でやり取りしなければならない.

データを互いにやり取りする方法としては, TCP やその他並列計算用の独自のプロトコルを用いた通信によって直接プロセス間でデータ受け渡しする方法の他に, 計算ノード間でファイルを共有させ, そのファイルへの read write を通じてデータのやり取りをする方法があり, そのシステムを広域分散ファイルシステムという. 例としては Bittorrent[1] のように計算ノード間で直接データをやり取りする P2P 方式でファイルを共有するものと, NFS[11], gfarm[31] などのようにデータサーバもしくはメタデータサーバが存在し, そのサーバを起点としてファイルを共有する図 1.1 のようなサーバクライアント方式でファイルを共有するものがある. 一般に複数のタスクで共通に読み込む必要のあるデー

タを共有させる場合には, 後者のサーバクライアント方式を用いることが多い.

しかし, 後者のサーバクライアント方式で並列処理を行う際, その環境やプログラミングの複雑さから様々なトラブルの発生が見られる. たとえば, 複数の計算ノードがタスクの結果を同時刻にデータサーバのファイルへと書き込む場合を考えると, サーバのストレージへ書き込む速度に対して, IO のリクエストの量が多ければ, サーバがリクエストを処理しきれずパフォーマンスの低下が発生する. 他にも, ある計算ノードが他の計算ノードがファイルへ write をしている際に read をしようとした場合, もし write の量が多かった場合, read でストレージからデータを読み込むためのメモリ領域の確保に時間がかかり, read のタイミングによってはパフォーマンスの低下が発生する可能性がある. それだけではなく, もしこの write が意図せず集中してしまった場合, ファイル情報へのアクセスやデータサーバが他の機能を兼ねている場合にそのサービスへのアクセスを行うことなどによって他の計算ノードの処理にも影響を与える可能性があり, InTrigger のように多数の利用者が別々に利用している場合においては, 自分以外の利用者の利用の妨げとなる可能性もある. しかし, ローカルにもストレージを持っているような場合では, 各プロセスがどのタイミングでどこにどれだけ IO したかを判別するのは難しく, データサーバへの負荷が大きくなった原因を特定するのは困難である.

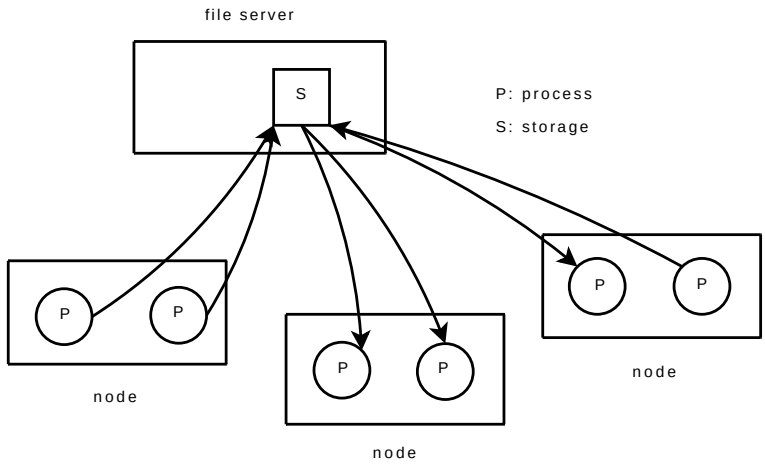

図 1.1: サーバクライアント方式でのノード間のデータのやり取りの例

このような分散環境での負荷原因を探る手段として, 各計算機に監視用デーモンやログ出力デーモンを常駐させ, システムの CPU 使用率などの統計量や Web サービスの応答状況などをユーザに見やすい形にして出力するモニタリングツールがあげられる. しかしこれらのツールはあくまで大まかな状態を可視化するものであり, より詳細な情報を得たり, その状態の原因の特定をするとなるとこれらのツールだけは難しい.

1.1.2 目的

本研究の目的であるが, ファイル共有システムに対して負荷がかかった場合に, その負荷の原因につながる IO を行っているプロセスを追求したい場合や, ネットワーク越しの IO をする際に, 負荷のかからないスケジューリング方法を考えたい場合に, その手助けをするため, 各計算ノードの各プロセスが, ファイル共有を管理しているサーバに対する IO 負荷の割合を推定する手法を提案する.

この手法では, 簡単に取得でき, オーバヘッドを与えにくい情報のみを用いて, プロセス毎のネットワーク上への IO の割合を推定し, それを可視化したり, ユーザにデータとして取得できるようにさせ, 今現在 IO にどのプロセスが負荷をかけているかを理解することができるようになる.

1.2 本論文の構成

以降, 本論文は次のような構成になっている. 第 2 章では本論文に関係する事項として一般に用いられているモニタリングツールや一般の異常解析の手法について取り上げる. 第 3 章では, 本研究が対象としている環境の構成と得られる情報について述べる. 第 4 章では, 3 章で得た情報を元に, IO 負荷の推定において各プロセスのデータサーバに対する IO の比率を求める手法として, 重回帰分析による推定法と, プロセスの IO の変動とそれに対応するデータサーバの IO の変動から推定を行う手法を述べる. 第 5 章では, 4 章で述べた手法の検証を行った結果を述べ, その有用性を検討する. 第 6 章でまとめと考えられる課題を述べる.

第2章　関連研究, 関連手法

2.1　モニタリングツールについて

負荷原因を探索する際に利用するものとして単純なものとしては, モニタリングツールを使うという例があげられる.

ganglia[3] は, 各計算ノード, データサーバなどの被監視ノードに CPU 使用率, ネットワーク通信量, ストレージへの swap in out の量などを一定間隔で取得する gmond というデーモンを常駐させ, 取得したデータを計算機群の監視サーバがネットワークを通じて gmetad というデーモンで受け取り, 監視サーバに http リクエストを送ると, それを rrdtool, PHP によって視覚化して図 2.1 のように表示する. 似たようなモニタリングとしては munin[9] があり, これも監視対象に munin-node デーモンを常駐させ, munin-server デーモンによってそれを rrdtool によって視覚化してファイルに保存, 外部から http リクエストを送ることでそのファイルをみることのできるツールである. このモニタリングシステムでは, 各ノードの通信量情報やディスクへの IO の時系列変化を得ることができ, どの計算ノードがデータサーバに対して IO 負荷をかけているのかを知ることが可能であるが, どのプロセスがどのストレージにどのくらい通信しているのか, といったプロセスに関する細かな情報は見ることができない.

nagios[10] は, 計算機群を監視しているサーバから各計算ノードやデータサーバに対し, 設定ファイルや各種プラグインに基づいて決まったリクエストを指定したポートに対して定期的に送り, その応答状況を記録することが可能である. これによって, ssh や http, NFS などリクエストをサービスが決まったポートで待ち受けているものに関してその応答確認, 応答速度を監視することができる. また、nrpe デーモンを計算ノードやデータサーバに常駐させておくことにより、監視サーバから各ノードへの nrpe デーモンの待ち受けるポートに対してリクエストを出すことで, そのノードのディスク IO 量などのリソースの状態についてデータを得ることができる. 得ることのできるリソースの状態に関する情報は、デフォルトで用意されているものの他, プラグインを作ることによって好きな項目を監視することが可能であり, プロセスの IO について監視したい場合は, それのためのプラグインを自分で作る必要がある.

Daniele Cesini らの研究 [19] では, gLite WMS/LB[4] というグリッド用ミドルウェアにモニタリングエージェントを仕込み, モニタリングサーバからアクセスすることによって投入されたジョブ数やその中で実際に稼働しているジョブ数, CPU 使用率などを可視化するグリッド環境でのモニタリングシステムを提案した. このシステムでは、投入されたジョブを一塊に見ることができ, ジョブ全体でどのような負荷をかけているかの把握に役立つ. しかし, 他のミドルウェアを用いている環境ではそれに合わせた移植作業等が必要になる.

このように, 現存するモニタリングシステムでは, プロセスに関する詳細な情報を見ることはそもそもできないか, そのためのプラグインなどを自作しなければならず, またプラグインによってデータを取得する場合は, プロセス数やスレッド数の増減によるオーバヘッドなども考えなければならない.

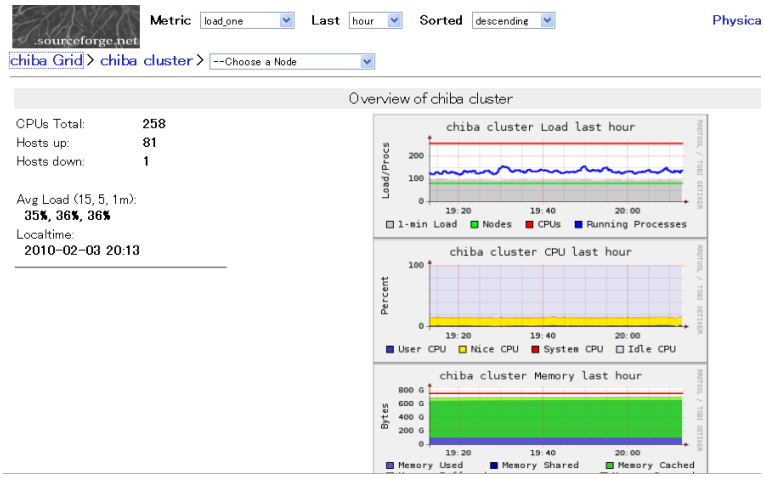

図 2.1: モニタリングツール ganglia

2.2 カーネル情報の取得

プロセスの詳細な IO 情報を得る手段として考えられる方法の一つに, カーネル内部の情報を利用する方法がある.

LTTng[26] と呼ばれるツールは, Linux カーネルソースコードを変更し, カーネル内での各種処理の追跡を可能にしている. 仕組みとしては, ユーザースペースで lttctl というコマンドラインアプリケーションによって netlink ソケットを通じてカーネル内の ltt-core に監視したい項目を伝え, ltt-core が各種モジュールにデータを集めるよう指示を出し, そのデータを ltt-base が RelayFS を通じてユーザ空間に伝達, lttd というデーモンがそれを受け取ってファイルに書き込む, という流れでデータを集めている. 全体の動作の流れは図 2.2 のようになっている.

dtrace[18] や systemtap[13] では, Linux カーネルのソースコードの変更はせず, カーネルモジュールを利用し, カーネル内でのイベントの発生情報を取得している. 何を取得しどのような形で取り出すかは, ユーザがそのアプリ用の言語を用いてスクリプトを書き, それを読み込ませることによってある程度ユーザの望む形で出力させることができる. 同じくカーネルモジュールを利用するツールとしては kstrax[7] というものがあり, カーネルソースコードを変更せず, カーネルモジュールを利用することで, カーネル内でシステムコール情報を収集することができる. さらに, 集めたシステムコールの呼び出しと復帰のタイミング, 各システムコールの回数などの統計情報を表示することができる.

これらのカーネル内の情報を直接取得する方法は, 実際にどの時刻にどのような IO が行われたかを厳密に解析することが可能であるという利点がある. しかし, 問題としてカーネルのソースにパッチをあてたり, Linux カーネルモジュールを利用したり等, 動的静的なカーネルの変更を伴うため, 複数の利用者が用いるような大規模なグリッド環境においては, 利用ポリシーによりそのようなカーネルの変更を行うことのできないケースも考えられる. また, ツールによってはカーネル内でのオーバヘッドが大きいものも存在し, InTrigger のように研究などに用いられる場合には, 性能評価に影響を与えてしまうケースも多く , できる限りこのようなカーネルの変更を行わない解析手法が求められている.

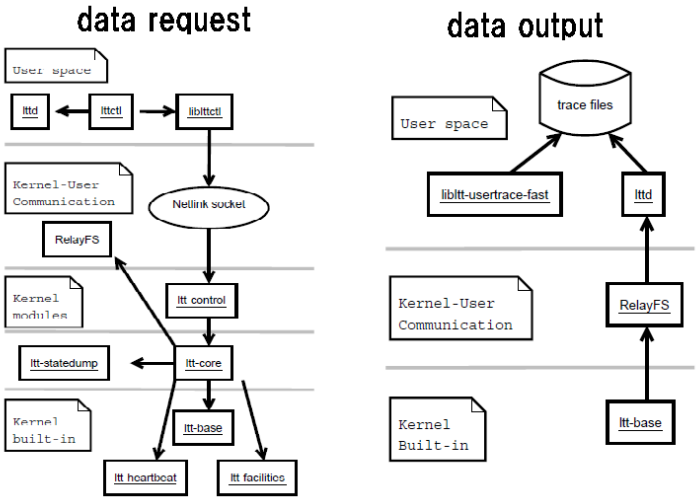

図 2.2: Lttng の動作の流れ

2.3 ファイルシステムからの IO データの取得

カーネルモジュールを利用した手法の中でも, IO データに特化した統計量の取得方法としては, 仮想ファイルシステムや FUSE など, ファイルシステムに手を加えて用いる手法がいくつか研究されている.

Akshat Aranya らが提案した tracefs [15] では, カーネルの仮想ファイルシステムと実ファイルシステムの間に tracefs システムを置き, IO が仮想ファイルシステムから実ファイルシステムへ渡る瞬間の IO のトレースをし, その結果を netlink ソケットを通じてユーザ空間へと出力するシステムが実装されている. そのシステムでは, さらにプロセス ID などでフィルタリングし, 必要な IO 情報のみを選択的にログに残すことができる.

Paratrac[12] は, FUSE[2] を使った IO トレーシングを行っている. FUSE は, ユーザ空間でファイルシステムのコードを実行することで, 本来のファイルシステムでは利用できない, より広範囲のリソースをファイルシステムとして見せることができるシステムである. このシステムのもう一つの利点として, 本来カーネル空間で実行するファイルシステムのコードがユーザ空間で実行されるため, ユーザ空間でファイルシステムのトレースを行うことができる, という点が存在する. これを利用し,Paratrac ではファイルシステム内での処理をトレースすることでどのような IO が行われたかをユーザ空間で取得することができる. さらに, 各タスクのファイルアクセスパターンから各タスクの依存関係を表す DAG の生成を行ったりすることも可能にする.

このようなファイルシステムを利用した IO データの取得は, プロセス毎の詳細な IO に関するデータを得ることができ, 導入も比較的容易である. これらを事前にバックグラウンドで動かしておくことで, 並列処理の IO 部分での性能の評価やチューニングを行うことができる. 問題としては, 多くのユーザが利用する場合, オーバヘッドが大きくなりやすいという点がある.

2.4 統計量, メッセージログ解析による異常原因の発見

データ IO 量に限らない統計量やメッセージのログなどから異常原因を解析する手法は多くの研究がなされている.

統計量を単純に閾値分類に用いた例としては, Eichenhardt らの考案したシステム [25] がある. このシステムではマシン全体に関係する統計量とジョブに関連する統計量を集め, ヒューリスティックスによる複数のフィルタにより, 各マシンで走っているジョブの状態として low,middle,high の 3 種類のラベルのどれかをつける. ラベルはフィルタ一つにつき一つつけられ, 最後にクラスター全体でジョブのラベルの付き方がどうなっているかを調べることで, 異常な動作をしているジョブを検出する. この手法の手順は, gLite[4] に含まれるモニタリングデータベースシステム R-GMA[16] を利用し, モニタリングデーモンを動かして分散環境上のマシン一つ一つに関する統計量と, そのマシン上で動いているジョブごとの統計量をマシン上に記録する. そうして記録したデータを R-GMA によって集め, それぞれのマシンの一つ一つのジョブごとに別々にフィルタを適用してラベルづけをおこなう. フィルタを適用する手順は次の通りである.

- フィルタに指定されたパラメータのセットを作成する. パラメータセットには, 複数の統計量の値から値を一つ生成するようなものも含まれる.
- フィルタにパラメータセットのそれぞれのパラメータに対して決まった順番に閾値による条件判定を行う.
- もしどこかで条件にマッチしたらそれ以降の判定は行わず, その判定に対応したラベルを付ける.
- 最後までマッチしなかった場合は最後の 3 種類の分岐判定によりラベルを付ける.

統計量ではなく, メッセージログを用いて解析する方法にもいくつかの研究がなされている.

Yuan らの研究 [29] や Liang らの研究 [24] は, syslog などに残されたエラーメッセージのクラスタリングを行うことで, ログから異常原因を示す可能性のあるメッセージの絞り込みを行ったり, メッセージを可能性のある異常の種類別に分類する方法を提案した. エラーメッセージログを時間, 種類, 場所などの情報をもとにクラスタリングし, そのクラスタから異常原因を示すものを取り出す方法を提案した. クラスタリングには, ログ中の単語, エラーの種類などを統計量として用いている.

Varrandi[28] は syslog から有用な情報を抜き出す方法として, 単語の出現場所とその数をカウントし, それに基づいて, 同じ位置に同じ単語が表れる組ごとにメッセージを分類することで膨大な量のログを調べやすい数に圧縮する方法を提案している.

Jon Stearley らの研究 [27] では, 計算機クラスタの各ノードの syslog から, syslog のある 1 ライン中の単語とその出現位置を組み合わせた要素を計算ノード毎にカウントし, その TFIDF 値を求めることで, syslog 内で異常を表す可能性の高い単語とその出現位置の組み合わせを求めた.

また, 収集した統計量を元に異常に関するモデルを作成する手法の研究も行われている.

Peter Bodik らの研究 [17] では, 視覚化手法として 5 分間のアクセス頻度に閾値を設け, その閾値を超えたかどうかでカラーリングする手法をとった. 異常検知の手法として, 5 分間の各ページへのアクセス頻度のベクトルを, 正常時と実際に異常かどうか調べたい時間帯で取り出し, この 2 ベクトルが似ているかどうかをカイ二乗検定と Naive Bayes の 2 つの手法で判定し, 似ているかどうかの度合いをスコアとしてそのスコアの遷移から異常を発見することを提案した.

Ira Cohen らの研究では, アプリケーションサーバ, データベースサーバなどのノード毎のパラメータを集めたベクトルを作り, TAN[22] という Naive Bayes を拡張した学習モデルを使い, 調べたい時刻のパラメータからその時刻で環境全体が異常かそうでないかを判定する手法を提案し [20], それを元に正答率から複数の学習モデルを生成して状況の変化に応じた適切なモデルを学習する方法を示し [30], その複数の学習モデルを使って各時刻において各パラメータが正常であるか, 異常であるか, どちらでもないかを判定し, それによってクラスタリングを行うシステムを提案した [21].

Aguilera らの研究 [14] では, 分散システムの各ノードが行うジョブの種類がノード毎に決まっているようなシステムにおいて, ネットワーク上の通信の送信時刻と受信時刻を元に, どのノードの処理がボトルネックになっているかを表す causal path を発見するシステムを提案した.

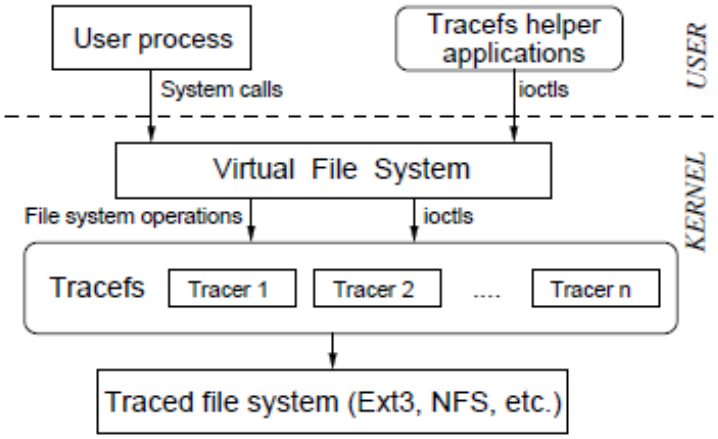

図 2.3: tracefs の基本概念

これらの研究は, 情報の蓄積を行うことで, 過去に起こった異常が再現された場合に発見することができるというものと, 似たようなデータグループが存在するときにそれをシステムの状態として分類するものである. 新たな異常を解析するには人手で行う部分も大きくなる.

```
$ slct-0.01/slct -o outliers -r -s 8% myhost.mydomain-log

Dec 18 * myhost.mydomain sshd[*]: connect from 172.26.242.178
Support: 262

Dec 18 * myhost.mydomain sshd[*]: log: Connection from
172.26.242.178 port *
Support: 262

Dec 18 * myhost.mydomain sshd[*]: fatal: Did not receive ident
string.
Support: 289

Dec 18 * myhost.mydomain * log: * * * *
Support: 176

Dec 18 * myhost.mydomain sshd[*]: connect from 1*
Support: 308

Dec 18 * myhost.mydomain sshd[*]: log: Connection from 1* port *
Support: 308

Dec 18 * myhost.mydomain sshd[*]: log: * authentication for *
accepted.
Support: 171

Dec 18 * myhost.mydomain sshd[*]: log: Closing connection to 1*
Support: 201

$ wc -l outliers
    168 outliers
```

図 2.4: 単語の出現位置と頻度から学習したログデータフィルタ

第3章　本研究での問題設定

ここでは、本研究が想定する問題設定について述べる.

3.1　想定する環境

本研究が想定している環境は, 次のような状況を想定している.

- ファイル共有システムとして, サーバクライアント方式による分散ファイルシステムを用いている
- 各計算ノードにはデータが共有されないローカルのストレージも存在する
- 共有されているファイルのメタデータ (更新日時などのファイル情報) はキャッシュされず、共有しているノード間ですぐに反映される
- 計算ノード, データサーバに分散ファイルシステムを管理するデーモンプロセスが常駐している
- 共有されている階層にあるファイルへの IO は, 必ず分散ファイルシステムを管理しているデーモンを通じて行われる
- 負荷の増大によるデーモンの故障はないとする

この状況設定は, 分散環境でデータのやり取りにサーバクライアント方式を用いている並列処理でも, 特に IO パフォーマンスが重要となるデータインテンシブな処理が実行されている状況を想定している. ここから, 実際の例を交えつつこの状況設定の意味について説明していく.

データインテンシブな並列処理でよく取り上げられるのが大規模ワークフローの並列分散処理で, 例としては gxp make による並列処理 [5] などがある. gxp make の例では, make の生成規則と生成コマンドのセットを一つのジョブと考え, makefile に記述された生成規則と生成コマンドのセットで, 生成規則から並列に実行できる場所を判断して並列化できる部分を自動で並列に実行させることでジョブの並列化を行う. gxp make では生成規則のターゲットと依存ファイル部分に各ジョブの依存する入出力ファイルを記述し, 全体としては図 3.1 のような DAG であらわされる. 図 3.1 の赤で囲まれた部分が実際に並列に実行可能な部分となる.

このようなワークフローを実行するときに, すべてのファイルを分散ファイルシステムによって同じサーバで共有すると, 並列に実行されたジョブが同時にサーバへの書き込みを行う確率が高くなり, 思ったようなパフォーマンスが出ない場合が考えられる. そのため, あるジョブを実行して出力されたファイルに依存するジョブがすべて同じ計算ノードで実行されるようにスケジューリングされた場合には, 共有されている部分に出力せず, ローカルのディスクに出力したほうがよいケースがある. また, 負荷を分散させるために複数のデータサーバを使うような事例も考えることができる. 図 3.1 の例では, もしジョブ J1, J2 が同じ計算ノードで実行され, J5 が同じ計算ノードにスケジューリングされるのであれば, ファイル f2, f3 は全体で共有させずに J1, J2, J5 の実行がされるノードと同じ計算ノードのローカルストレージにおいておく方が効率がよい. ここから, タスクの出力先が分散ファイルシステムだけでなくローカルストレージも存在するという条件設定をしている.

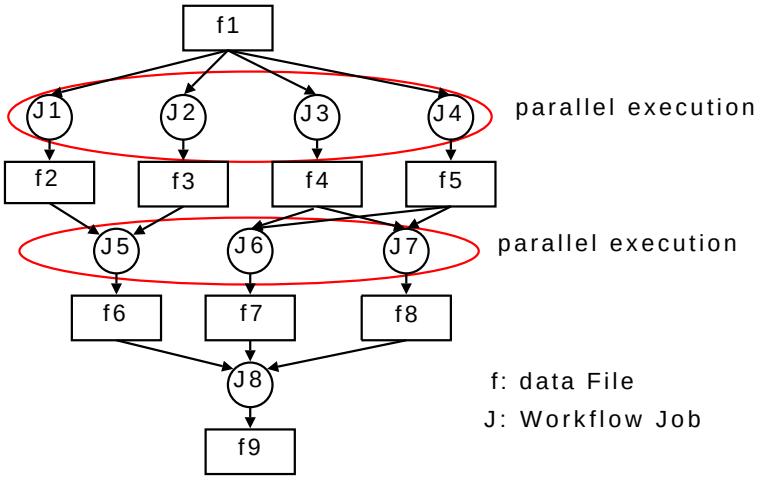

図 3.1: 並列実行可能なワークフローの例

また, ある計算ノードで実行されるタスクが依存するファイルが他の計算ノードによって出力されるものである場合, ファイルの変更がある程度キャッシュされ, すぐに反映がされない環境で実行を始めようとすると存在するはずのファイルがない, といったような状況が起こりうる. そのため, ファイルの変更に関する情報が共有しているノード間ですぐ反映される環境でなければ, このような並列実行ができない. 図 3.1 の例では, ジョブ J1-4 の実行が終了したときに, ファイル f4, 5 の生成がすぐに反映されなかった場合, ジョブ J6, 7 を実行しようとした時にファイルを参照できずにエラーとなることになる. よってメタデータをキャッシュせずファイル変更などの情報がすぐ反映される状況, という設定をしている. NFS の例では, mount のオプションとして noac を指定してファイルやディレクトリの属性のキャッシュを禁止している状況になる.

そして, IO の全体の負荷状況を把握する方法としては, 後で詳しく述べるが IO を管理しているサーバデーモンが IO をしている状況をモニタリングして行うため, デーモンが常駐していて, すべての IO がそれを通して行われる状況を仮定している. たとえば, NFS ではデータサーバの nfsd が各計算ノードの nfsiod を通じてファイルの共有部分のデータのやり取りを行っており, 計算ノードのファイル共有部分への IO は, 計算ノードのプロセスのファイル変更やファイルアクセスのリクエストを nfsiod が受け取って データサーバの nfsd と通信し, nfsd が共有部分のストレージへのファイルデータを受け渡ししている.

最後に問題を簡単にするため, サーバや計算ノードで動いている管理デーモンの故障は考えず, プロセス側で IO に成功したがデーモン側では失敗したという状況が発生しないと考える.

環境全体のイメージとしては, 図 3.2 のような環境になる. このような環境において, データサーバや共有されているファイルがあるストレージへの負荷の原因を調べる手がかりとなる要素を取り出すことが, 本研究の目的となる. 直観的には, ファイル共有部分への IO に対応するのは, 図 3.2 に示される各計算ノードのプロセスから, サーバへ向かって IO のデータを送るクライアントデーモンプロセス iod へ向かう矢印であらわされるデータの流れと考えることができる. このデータの流れを軽量かつ簡単に調べることができれば, 共有サーバへの IO の負荷の原因についても考えられそうである.

3.2 モニタリングを行うデータについて

次に, 前節の最後に述べたプロセスからファイル共有部分へのデータの流れの調査のためにモニタリングをするデータについて述べる. 本研究の目的として, モニタリングの方法について求められる

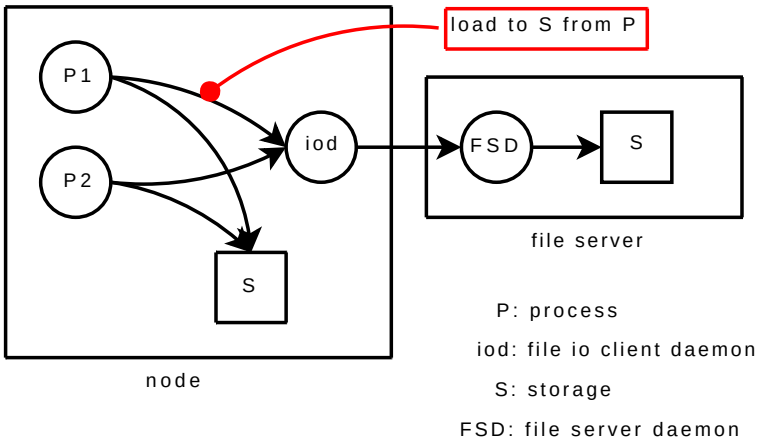

図 3.2: 想定する環境の例

要件は次の通りである.

- モニタリングによるオーバヘッドをできる限り抑える
- 複数の利用者が共同で利用する, 利用ポリシーによって制限のある環境であることを考え, 極力カーネルコードの変更やカーネルモジュールの利用をしない形態である
- ファイルシステムなどのソフトウェア環境やマシンの種類などのハードウェア環境が多少異なっても適用できる, 汎用性の広い方法である

この要件を満たしつつ, プロセスからファイル共有部分へのデータの流れを求めるのにつながる統計量を考える.

一般に, ファイルシステムはカーネルの内部で処理が動くために, ファイルシステムに関する情報を得るにはカーネルコードの静的, 動的変更などを利用する必要がある. しかし, ユーザが Linux カーネルモジュールなどを明示的に用いずとも, デフォルトでカーネルが提供している機能を使うことでオーバヘッドを抑えて情報を取得する機能が用意されている.

話を簡単にするため, ここではこの後の実験に用いた環境である Linux 環境に関して話を進めるが, 他の OS においても似たような機能があれば, 同じ手法を適用することが可能である.

Linux では, カーネル内部の情報を取得するための機能として, /proc ファイルシステムが用意されている. /proc ファイルシステムでは, プロセス, CPU, メモリ, ファイルシステムに関する情報を /proc をルートディレクトリとする仮想的なファイルシステムにアクセスする形で取得することができる. 現在の Debian の stable ディストリビューションである Linux-kernel-2.6.26 では, 主に次のような情報を取得できる.

- /proc/[pid]/fd
  プロセス ID が [pid] であるプロセスが開いているソケット, ファイルへのリンクを表す. リンク先のファイルが何を mount したディレクトリの下のファイルを指しているかから, そのプロセスがデータサーバのファイルやローカルのストレージのファイルを開いているかを判断することができる.

```
yuuki-s@guppygw:~\$ ls -lh /proc/7727/fd
total 0
```

```
lrwx------ 1 yuuki-s yuuki-s 64 2010-02-03 16:34 0 -> /dev/pts/1
lrwx------ 1 yuuki-s yuuki-s 64 2010-02-03 16:34 1 -> /dev/pts/1
lrwx------ 1 yuuki-s yuuki-s 64 2010-02-03 16:34 2 -> /dev/pts/1
l-wx------ 1 yuuki-s yuuki-s 64 2010-02-03 16:34 3 -> /home/yuuki-s/test.dat
```

- /proc/[pid]/io
  linux-kernel-2.6.20 以降で導入された, IOaccounting 機能による IO に関する統計量を表す. プロセス ID が [pid] が生じたときから参照した時までの IO に関する情報を表している.

  rchar, wchar システムコール read, write を行い, リターンした際に成功した byte 数を表す

  syscr, syscw システムコール read, write を行い, それがリターンした回数を表す

  read_bytes そのプロセスの操作によってブロックデバイスからメモリ上のファイルキャッシュへのデータの読み込みを起こした場合, その byte 数を表す

  write_bytes そのプロセスの操作によってメモリ上のファイルキャッシュのデータの dirty フラグが立った場合, その byte 数を表す

  cancelled_write_bytes 上の write_bytes のうち, ストレージにライトバックされる前に別のデータによってデータが書きかえられ, 実際のストレージにはライトバックされなかった byte 数を表す

```
yuuki-s@guppygw:~ cat /proc/7727/io
rchar: 149455
wchar: 249
syscr: 107
syscw: 36
read_bytes: 389120
write_bytes: 0
cancelled_write_bytes: 0
```

- /proc/diskstats
  そのマシンの実ファイルシステムが mount しているブロックデバイスへの IO に関する統計量. 主にデバイス名の文字列の後にある統計量の 3, 7 番目にあるデバイスへの読み込み, 書き込みのセクタ数を表す項目が重要となる. 1 セクタあたりのデータの byte 数は決まっている (たとえば Linux では 512 byte) ので, ここから実際のデバイスへの IO 量を知ることができる.

```
yuuki-s@guppy0:~ cat /proc/diskstats
   8    0 sda 148985 163595 7991652 1418700 4583648 76380858 647719890 ...
   8    1 sda1 79273 20247 3099988 466572 389742 1521197 15287560 ...
   8    2 sda2 2 0 4 120 0 0 0 0 0 120 120
   8    5 sda5 17975 52233 560308 87044 4753 186630 1532248 3221836 ...
   8    6 sda6 16594 4749 327324 209732 1280574 1396012 21416192 ...
   8    7 sda7 23855 81110 2850838 97320 60559 420819 3848058 2066740 0 ...
   8    8 sda8 11257 5233 1152774 557608 2848020 72856200 605635832 ...
```

この/proc ファイルシステムの仮想ファイルを定期的にアクセスし, そのデータを読み込むデーモンをバックグラウンドで動かすことで, プロセスの IO の推移に関する情報を得ることができる.

このような/proc ファイルシステムを用いたデータの取得で問題となるのは, プロセスの終了時のデータである. /proc ファイルシステムでは, プロセスが終了すると/proc/[pid] ディレクトリが消え, その後はその終了したプロセスのデータへアクセスすることはできなくなる. もしプロセスが終了時に大きな write を行う場合, それを取り逃してしまうケースが考えられる.

そのような終了時のプロセスのデータを得る方法として, netlink ソケットを用いる方法がある. netlink ソケットは, linux においてカーネルとユーザのプロセスの間でデータをやり取りする時に用いられる特殊なソケットである. 各種カーネルモジュールなどを利用するときにもこのソケットが用いられており, カーネル内の情報を取得するときには必ず用いられる機能の一つである.

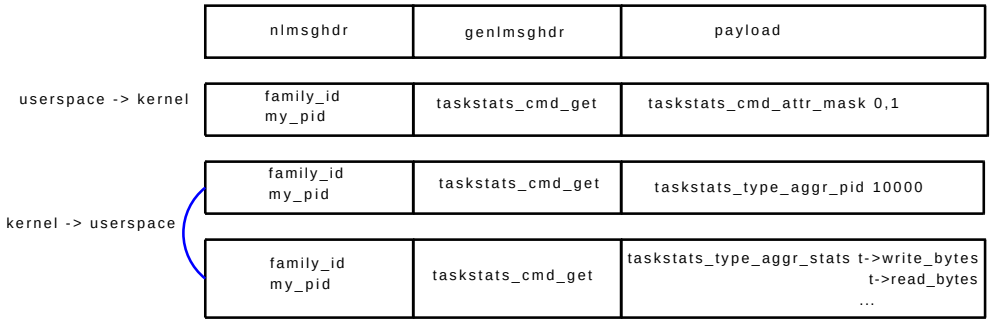

図 3.3: netlink ソケットを通るメッセージの構成

linux-kernel-2.6.20 以降ではプロセスの IO データを取得する IO accounting 機能がカーネルに付け加えられている. 先ほどの/proc ファイルシステムもこれを用いているが, この機能のデータは netlink ソケットを使って受け取ることが可能である. netlink ソケットは, カーネルと情報をやり取りするためのソケットであり, コマンドをメッセージで流すとその実行結果を受け取ることができる. IO accounting のデータを受け取るには NETLINK_GENERIC グループと通信するソケットを作り, 通信するための family id を調べてからその id にプロセス終了データを送るコマンドを送ればよい.

その際に用いる netlink メッセージは図 3.3 のような構造をしており, nlmsghdr にメッセージの送信先や受信元や全体の長さ, genlmsghdr に実行コマンド, payload に実行コマンドに必要なデータや実行結果のデータが入っている.

IO accounting のデータを受け取るための手順は, 次のようにまとめられる.

1. netlink ソケットを作る
2. generic netlink を使うための family id を netlink ソケットを通じてカーネルから取得する
3. その ID に対し, 終了したプロセスのデータを送るように指定するコマンドと, 終了プロセス情報を受け取りたい CPU の cpumask の番号のデータを送る
4. netlink ソケットからプロセスが終了するたびにプロセスの終了データが送られてくるので, payload 部分からその情報を取り出す

これを図にしたものを図 3.4 に示す.

実際にどのようなデータがメッセージに含まれるかの例の一つとして, 終了プロセスのデータ送信依頼とその結果のメッセージを図 3.3 に示す。

このカーネルモジュールはデフォルトで使用されているモジュールであり, オーバヘッドも非常に小さく取得することができる. プロセスの終了データから /proc/[pid]/io と同じデータを取り出すことで, 終了前の IO に関するデータを得ることができるので, 先ほどの/proc ファイルシステムの統

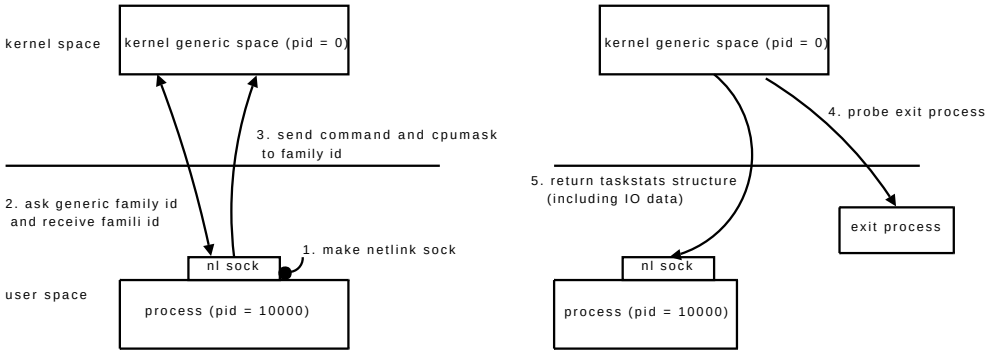

図 3.4: netlink ソケットによる終了プロセスデータの取得手順

計量と組み合わせることで, ある決まった間隔で各プロセスが行った IO の量に関する情報を把握することができる.

これらで得られる統計量を, 図 3.2 と照らし合わせる.

まず, プロセスが実際に iod によって拾われるデータを出力しているかどうか, つまり各プロセスから iod に向かう矢印が存在するかどうかは, 単純にファイルを開いているかどうかで判断できるため, それは/proc/[pid]/fd から確認することができる.

プロセスはファイルへの IO をすべて iod プロセスを通じて行うが, iod プロセスがプロセスからの IO データを受け取るのは, 実際には図 3.5 にあるメモリのファイルキャッシュの変更を感知してその変更部分を IO データとして受け取っている. ここで, 先ほどの条件設定からファイルの変更はキャッシュされずに即座に反映されるとしているから, iod はファイルキャッシュの変更を即座に FSD に伝える. よって, データサーバと計算ノードの遅延が十分に小さければ, 図 3.5 のデータサーバと計算ノードのメモリ上のファイルキャッシュFC の同期は完全に取れており, 同時刻に同じ内容になっていると考えてよい.

実際のストレージへの書き込みは, ファイルキャッシュが十分な量だけ変更された場合か, 変更されてから決まった時間経過してから起こるので, そのディスクへの書き込みの要因はプロセスのファイルキャッシュへの書き込み量から判断できるということになる. ここで, 単純なシステムコールによるプロセスの IO 量を見ても, 実際にはメモリマップファイルなどを利用しているケースもあり, またファイルサーバデーモンは実際に read write のシステムコールを使って読み書きをしていないため, read write システムコールによる IO 量と実際の IO 量の対応がとれないケースが発生する. なので, プロセスがメモリキャッシュの変更をして dirty フラグを立てた量をカウントしている write_bytes と, ストレージへのアクセス量をカウントしている read_bytes をそれぞれプロセスから出ていく, 入っていく IO 量とすることで, 各プロセスの IO によってかけた負荷の大きさを知ることができ, また全プロセスによってサーバにかかっている負荷の総量を サーバデーモン の write_bytes から判断することができる.

ただし, この write_bytes 値は図 3.5 の青枠で囲った領域を通る IO であり, つまりプロセス側ではローカルのファイルキャッシュの量とデータサーバのファイルキャッシュの量をすべて合わせた値となっており, ローカルかサーバかどちらに出力をどれだけしたのかを判別することはできない.

これより, 各プロセスがどの程度データサーバに IO しているかは, /proc ファイルシステムや netlink ソケットの情報だけでは判断が難しい. IO のうちどれだけがデータサーバへ向かっているかを知るには, システムコールの発行時にどこへの書き込みかを記録したり, メモリマップファイルでどのファイルキャッシュを触ったかを記録する必要があるが, このような記録機能はデフォルトのカーネルには存在せず, これらがすべてカーネル内で行われる処理であるため, システムコールのフックを行ったり, ファイルキャッシュへのアクセスを調べたりするにはカーネル内で情報収集をする機構を組み

込む必要がある. だがこの章の最初に述べたとおり, そのようなカーネルに変更を加える手法に頼らずにそれと同等の情報を得たい. そこで, IO の出力先の判別をモニタリングによって得た値から推定するということを考えたい.

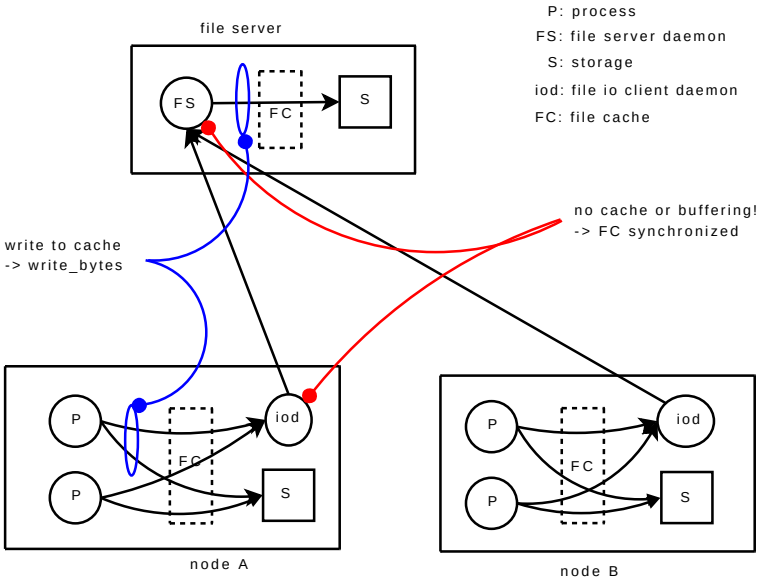

図 3.5: write の例

第4章　IO負荷の推定

ここでは, 3 章で述べたような状況において, データサーバへの負荷をどのように見積もるかを述べる.

4.1　推定すべき値について

3 章の 3.1 で述べた環境において, 各プロセスの全 IO のうちのデータサーバへの IO の割合が, データサーバへの負荷につながる要素であるが, 3 章 3.2 で述べたように, データとしては各プロセスの全体の IO やデータサーバデーモンの IO の量は知ることができても, 各プロセスがそのうちどのくらいデータサーバに向かって IO をしているのかを知ることはできない. そこで, 3 章 3.2 で得たモニタリングデータをサンプル値として, そこからプロセスの IO のうちデータサーバへ向かう IO の割合を推定することを行う.

3 章 3.1 であげた条件を仮定したとき, 計算ノードがファイルキャッシュにデータを書き込むと, 計算ノードとデータサーバ間でキャッシュがおこわなれないために, 即座にサーバデーモンがサーバのファイルキャッシュにそのデータを書き込む. よって, IO をしているプロセスが 1 つで, データサーバにのみ IO を行っている場合, その IO を行っている区間でのそのプロセスの write_bytes の値の増分と, サーバデーモンの write_bytes の値の増分は一致する. これより, ある時刻 t_0 と t_1 に 3 章 3.2 で述べた手法によって取得したサーバデーモンの write_bytes の値と, 各計算ノードのプロセスの write_bytes の間には, 全プロセス数を N とすると次のような関係式が成り立つ.

$$\Delta y_s = \beta_1 \Delta x_1 + \beta_2 \Delta x_2 + \cdots + \beta_i \Delta x_i + \cdots + \beta_N \Delta x_N \tag{4.1}$$

$$\Delta y_s = y_s(t_1) - y_s(t_0) \tag{4.2}$$

$$\Delta x_i = x_i(t_1) - x_i(t_0) \tag{4.3}$$

ここで $y_s(t)$ はファイルサーバデーモンの時刻 t での write_bytes の値, $x_i(t)$ はプロセス i の時刻 t での write_bytes の値, N は全プロセス数を表している.

ここで, 3 章 3.1 の最後の条件から, プロセスとデータサーバの間のやり取りにおいてデータの整合性がとれなくなる場合はないとしており, IO が成功し, 同期がとれた場合に限りプロセスとデータサーバの write_bytes の値がカウントされるとすると, データサーバに IO をしたつもりでプロセスの write_bytes が増えても, それに対応する write_bytes が増えない, という状況がないということになる.

よって, β_i は各計算ノードでの各プロセスの全 IO におけるデータサーバに対する IO の割合と考えることができるので, この β が求まれば, そこからデータサーバへの各プロセスの IO 量を $\beta_i \Delta x_i$ と推定することができることが分かる。しかし、x, y の値を 3 章 3.2 の方法から得たとしても, 式ひとつに対し未知数 β がプロセス数 N だけ存在し、そのままでは求めることができない。

4.2　重回帰分析による推定手法

各プロセスの IO は, スケジューリングされるタイミングとモニタリングでデータをとるタイミングによってまちまちにスケジューリングされうるので, 先ほどの β の値はモニタリングするタイミン

グによって時々刻々と変化する. 図 4.1 の例では, A への IO と B への IO の比率が, 各区間ではバラバラである. しかし, IO をマクロな視点でみれば, データサーバに対して出力している大よその割合というものが存在するはずである. この図 4.1 の例では, 全体としては A, B に 1:1 の割合で出力をしている. その大よその割合 β がある長さの区間で決まるとし, モニタリングによって得た値に対してもっともらしい値が分かればそれを β として用いることができると考えられる.

4.1 の回帰式 4.1 を満たす β を最小二乗法による最尤法によって推定する. ある決まった区間での β を求めるとするとき, その区間でサンプリングした点をすべて 4.1 式にあてはめ, 左辺から右辺を引いた誤差の二乗和 e を最小とすればよく, その誤差は次の式であらわされる.

$$e = \sum_t \left(y_s^t - \left(\beta_1 x_1^t + \beta_2 x_2^t + \cdots + \beta_N x_N^t\right)\right)^2 \tag{4.4}$$

この誤差 e がもっとも小さくなるように β を決めれば, それがもっともらしい β であるとすることができる. 誤差が最小となるとき, 全プロセスの β に関して微分値はすべて 0 になるはずであるから, e に対しすべての p, i に関して, $\frac{d}{d\beta_{p_i}} e = 0$ となればよいことになる.

これより次の式が導かれる.

$$\begin{aligned} \frac{d}{d\beta_i} e &= -2x_i \sum_t \left(y_s^t - \left(\beta_1 x_1^t + \beta_2 x_2^t + \cdots + \beta_N x_N^t\right)\right) && (4.5) \\ &= -2 \sum_t \left\{\left(y_s^t - \beta_1 x_1^t - \beta_2 x_2^t - \beta_N x_N^t \cdots\right) x_i\right\} = 0 && (4.6) \end{aligned}$$

この式は次のような行列式に変形できる.

$$\begin{pmatrix} \sum x_1^2 & \sum x_1 x_2 & \cdots & \sum x_1 x_N \\ \sum x_2 x_1 & \sum x_2^2 & \cdots & \sum x_2 x_N \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_N x_1 & \sum x_N x_2 & \cdots & \sum x_N^2 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{pmatrix} = \begin{pmatrix} \sum y_s x_1 \\ \sum y_s x_2 \\ \vdots \\ \sum y_s x_N \end{pmatrix} \tag{4.7}$$

つまり, サンプリングした x, y からこの行列の成分を求めることによって, β を決定することができることが分かる. これは, 4.1 式において, データサーバの IO 量を従属変数, 各プロセスの IO 量を説明変数とする重回帰分析による推定をするということを意味する. .

この手法によって負荷の割合が求められる根拠は, 各プロセスが各 IO 測定区間で行う IO スケジュールにぶれがあるためである.

図 4.2 の例では, プロセス A とプロセス B それぞれの総 IO 量は, プロセス B が IO を開始した時刻から各測定区間でまちまちになっている. プロセス A の IO が多くスケジューリングされた区間もあれば, プロセス B の IO が多くスケジューリングされた区間も存在している. この例ではプロセス A の IO のうち 76 %がデータサーバに対する IO, プロセス B は 100 %データサーバに対する IO を行っているが, もしプロセス A, B ともに各サンプリング区間で同じように IO がスケジューリングされた

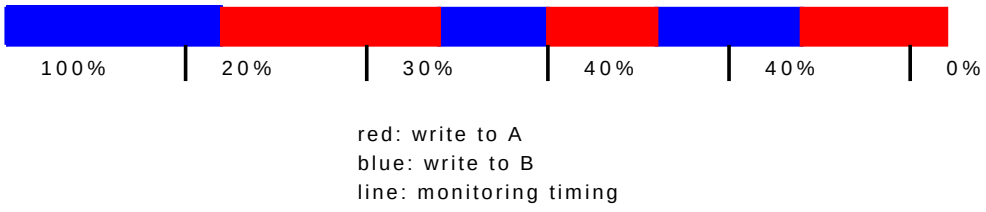

図 4.1: 2 つの出力先に IO を行うプロセスの例

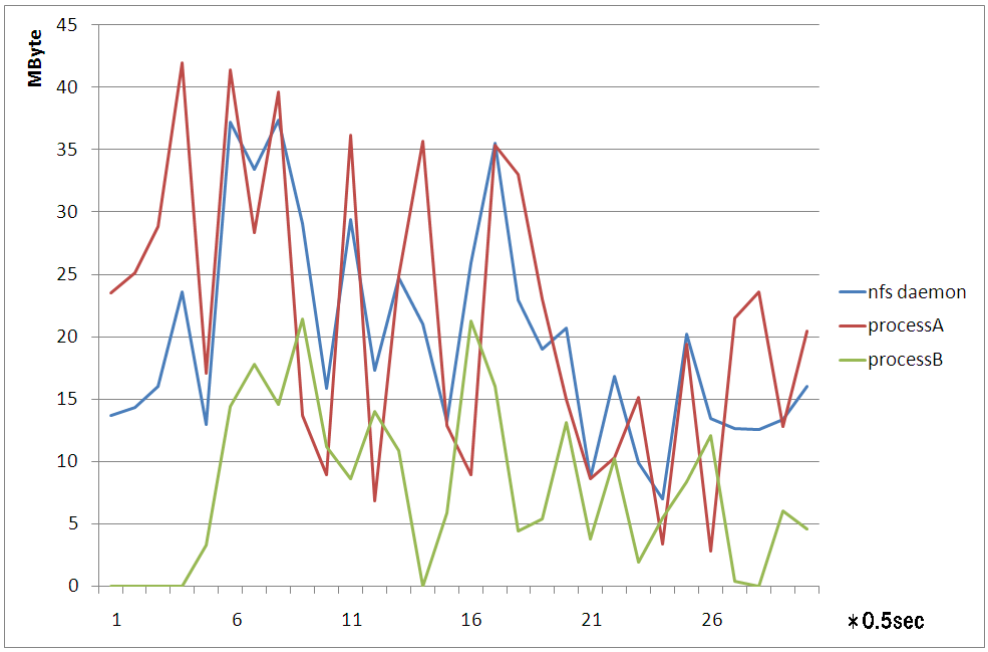

図 4.2: IO スケジューリングによる IO 値のぶれ

場合, つまりは $x_B = ax_A$ と常に同じ比率で A, B の IO がスケジューリングされた場合, 4.7 式の左辺の行列のランクが行の次元数と一致しなくなるため, β を決定することができないが, 実際には異なるため, 比率の推定が可能になっている.

しかし, この手法には問題点もある. この手法では, すべての説明変数が独立である必要がある. 説明変数が独立でない場合, 説明変数のうちの一つは他の説明変数の式によって決定されることになり, 係数 β が一意に決まらなくなる可能性がある. これを多重共線性という. 先ほどの例で IO スケジュールにぶれが小さい場合, 誤差が大きくなる可能性が高い. また, 推定には十分なサンプリング点が必要で, プロセス数が多くなるとそれだけサンプリング点はより多く必要となる. サンプリング点を多くすると, β を決める区間が大きくなってしまい, プロセスの β が早く切り替わるような場面では推定した値の意味が薄れてしまう. その場面について推定できる別の手法を考える必要がある.

4.3 プロセス IO の変動タイミングに着目した推定手法

ここでは, ストレージの最大書き込み速度で書き込みを行っている場合の推定手法について考える.

まず仮定として, サンプリングの区間が小さく, その区間に複数のプロセスで同時に全 IO に対するデータサーバへの IO の出力の割合 β が変化する確率が十分小さいとする. この時, 前の区間でのデータサーバへの IO 比率 β が全プロセスで分かっている場合, 前の区間 β を用いた 4.1 式に一つのプロセスの β が変化した新しい区間のサンプリングしたデータサーバ, 全プロセスの IO 値を代入すると, 当然 β が変化したプロセスの影響により誤差が大きくなっている. このように, 誤差が大きくなったときにいずれかのプロセスで β が変化したと判断し, 変化した新たな β を導く, という変動タイミングに着目した推定を行う.

推定の方法は単純で, 次のような手順を踏む.

1. すべてのプロセスで β は 1 であるとする
2. 4.1 式にサンプリングした区間のデータサーバ, プロセスの IO 値をあてはめ, 誤差が閾値を超える区間を見つける
3. 見つけた区間に対し, あるプロセス 1 つで β が変化し, それ以外では変化していないと仮定する
4. 4.1 式に変化していない β とサンプリングしたプロセスとデータサーバの IO 値を代入し, そこから IO 割合の変化したプロセスの β を求める.
5. 3, 4 を全プロセスで行い, すべてのプロセスで仮定の元に変化した β を求める
6. この複数の仮定から求めた β の変化の仕方として “ありうる” 変化を取り出す.
7. 再び 2 に戻り, 繰り返す

“ありうる” 変化を判断する方法としては, 次のような方法がある.

- β はプロセスの IO 全体における, データサーバへの IO 割合を意味するので, $0 \leq \beta \leq 1$ である.
- β はすぐには変化しないと考え, その直近の区間の 4.1 式の誤差を見て, 誤差のもっとも小さい β を変化したものとして採用する

直近の区間の誤差から判断する方法は, 同じペースで IO をしていれば一見適用しても効果がないように思われる. ここで同じペースとは, 各プロセス, データサーバの IO 量 x_i, y の値が時刻によって変化しないということを意味する. もし, 全プロセスの IO が毎回の区間で同じペースでスケジューリングされていれば, どのプロセス β が変化したと考えても誤差が似たような値になる可能性がある. しかし, 実際には 4.2 のように, プロセスの IO スケジューリングにはばらつきがある. このデータサーバの IO 速度のぶれによって, 誤差をはっきりと見ることが可能である.

この状況の具体例を考える. プロセスのファイルへの write は, 3 章で述べたようにファイルキャッシュへの書き込みによって行われる. ファイルキャッシュへの書き込みが増え, dirty なページの割合が全体に対して一定以上になると, ストレージへのフラッシュを行うデーモンによって実際のストレージへの書き込みが行われる. このデーモンの動作時間が増える他, データサーバの他のデーモンプロセスなどが CPU を使うことにより, ファイルサーバデーモンが CPU を占有する時間は相対的に短くなることになる. よって, IO 負荷が大きいと, スケジューリングのタイミングによって, 各区間にデータサーバデーモンの IO のスケジューリングの量がまちまちになる. この様子は 4.2 にも見ることができる. この IO のスケジューリング量のランダムさにより, もしローカルのストレージへほとんど IO をしているプロセスがいれば, データサーバのこの IO のランダムさにほとんど影響を受けることなく IO ができるはずである. よって, β が大きいプロセスはデータサーバデーモンの IO のぶれに対し

て大きく変動していなければならず, そうでなければそれが誤差として認識され, β が小さくなったと推定してくれることになる.

また, 現実には複数のプロセスの IO 比率が変化する事例が起こるほか, 直近の区間で 4.1 式をあてはめる方法を使う問題点として, IO 比率の変化した区間では, 変化前と変化後の比率の中間のような状態が存在する可能性があり. 直近の区間の 4.1 式の誤差のもっとも小さい β を採用しても誤差が大きく信頼できる値が得られない場合がある. その場合の対応方法としては次のようなものが考えられる.

- 直近の区間で 4.1 式の誤差の小さい β がない場合, そこを IO 比率の変化の中間状態として β を不定にし, 次の区間で判断をする
- 直近の区間の 4.1 式の誤差が大きくても他の β とくらべ最小であればとりあえず “中間状態” として決定し, 先へ進む

さて, このような推定を行うためには IO 比率が一度に複数のプロセスで変化しない, 直近の区間で変化しない, という仮定のほかに, 暗に 1 つの仮定を含んでいる. あるプロセス p がデータサーバとローカルのストレージ双方に IO をしている状態であるとし, そこへ別のプロセス q がデータサーバへの IO を開始したとする. この時, p がデータサーバへの書き込みをデータサーバのストレージの限界の書き込み速度で行っていた場合, q の IO によって p のデータサーバへの IO が抑制されることになる. よって, p が q の影響によって IO 比率を変化させるということが起こらない, ということが言えないと, IO 比率が複数のプロセスで一度に変化しないという仮定をおくことができない. この手法をあてはめる場合には, このことが言えるケースにのみ適用できるということになる.

この手法をここからは単一変動推定と呼ぶ.

第5章　実環境における検証

ここでは, 実際の環境においてここまでに述べた手法の検証を行う. 5.1 のような環境で実験を行った.

ノードの種類	計算ノード	データサーバ
台数	5	1
CPU 数	2	1
メモリ	2GB	1GB
分散ファイルシステム	NFS	NFS
ストレージの IO 速度	20〜30Mbyte/sec	40〜50Mbyte/sec

表 5.1: 検証環境

5.1　モニタリングプロセスのオーバヘッド

まず, 3 章で述べたプロセス IO のモニタリングシステムのオーバヘッドについて述べる. ここで, 次のような実験を行った.

- 並列分散処理の例として, ワークフローの Montage[23] の gxp make による並列化したものを実行する
- データセットは gxp make [5] 向けに用意されたデータセットの data1, data2 を用いた
- 3 種類の環境でこれを実行する
  - モニタリングプロセスを何も動かさない
  - 3 で述べた IO のモニタリングプロセスを動かす
  - オーバヘッドのあるモニタリングプロセスの例として, logfuse[8] を動かす
- 各ノード 2 プロセスの計 10 並列
- データサーバ IO をモニタリングし, その終了までの時間を見る

結果は表 5.2 のようになった.
結果から, FUSE などのファイルシステムを使うがカーネルの直接的変更をしないモニタリングシステムと比べ, オーバヘッドが十分小さいということが分かる.

5.2　2 プロセスの場合における推定

ここでは, もっとも単純な例である 2 プロセスにおける推定を行った.
実験方法としては, 次のように行った.

データセット	モニタリングなし	logfuse	提案手法
data1 1	76.545sec	94.660sec	76.775sec
data1 2	76.910sec	96.120sec	76.822sec
data1 3	75.131sec	94.216sec	77.402sec
data2 1	949.948sec	2105.406sec	979.645sec
data2 2	943.300sec	2009.735sec	1000.065sec

表 5.2: 測定のオーバヘッド

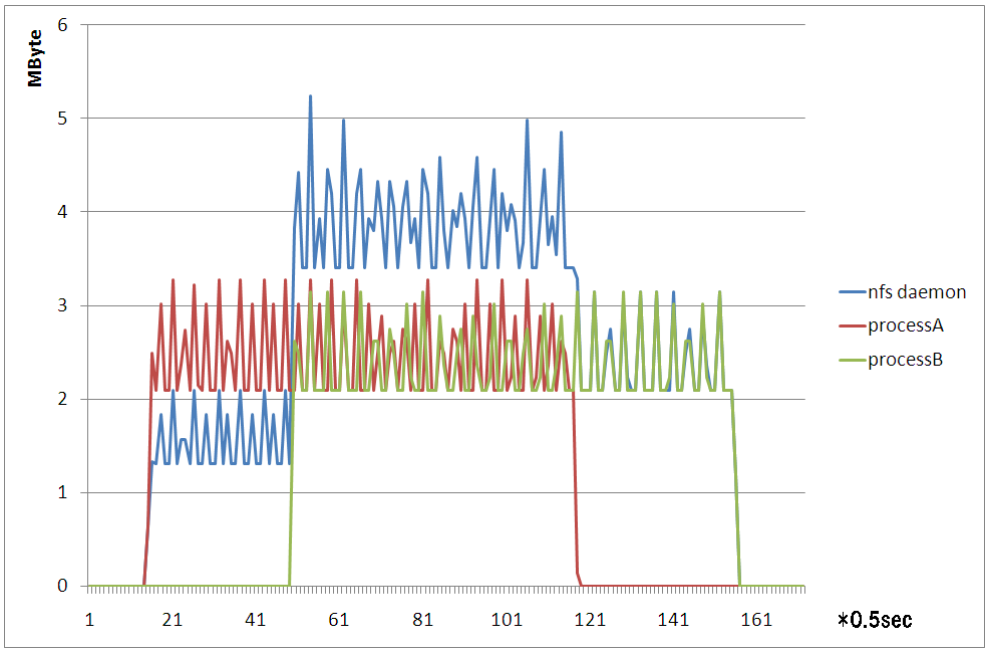

図 5.1: 2 プロセスによる IO の推移

- IO を行うプロセスは 2 つ存在し, それぞれ別々の計算ノードで動いている
- コード内ではストレージの速度 (40Mbyte/sec) に対し早くならないように sleep をはさみ速度をある程度抑えている (大よそ 2〜4Mbyte/sec)
- 1 つめのプロセスが先に IO を始める. プロセスの IO の比率はデータサーバとローカルのストレージが 2 : 3 になっている
- 2 つめのプロセスが後から IO を始める. プロセスの IO はすべてデータサーバに行っている.
- IO の値のサンプリングは 0.5sec 間隔で行う

図 5.1 は, 実際の IO の様子である. 図から, プログラム内での IO 比率と実際の IO 比率がしっかりと対応していることが分かる.

これに対し, 前に述べた重回帰分析による方法と, 単一変動推定による 2 つの推定手法をあてはめた. 2 つのプロセスの IO が混じり, 推定の必要が生じる時間帯での推定の結果を図 5.2, 5.3, 5.4, 5.5, に示す. 順に図 5.2, 5.3, 5.4, は重回帰分析の 1 回の推定に用いるサンプリング数を 5, 10, 15 にしたもの, 図 5.5 は単一変動推定による推定の閾値を 10KByte, 50KByte, 100KByte にしたものである. なお,

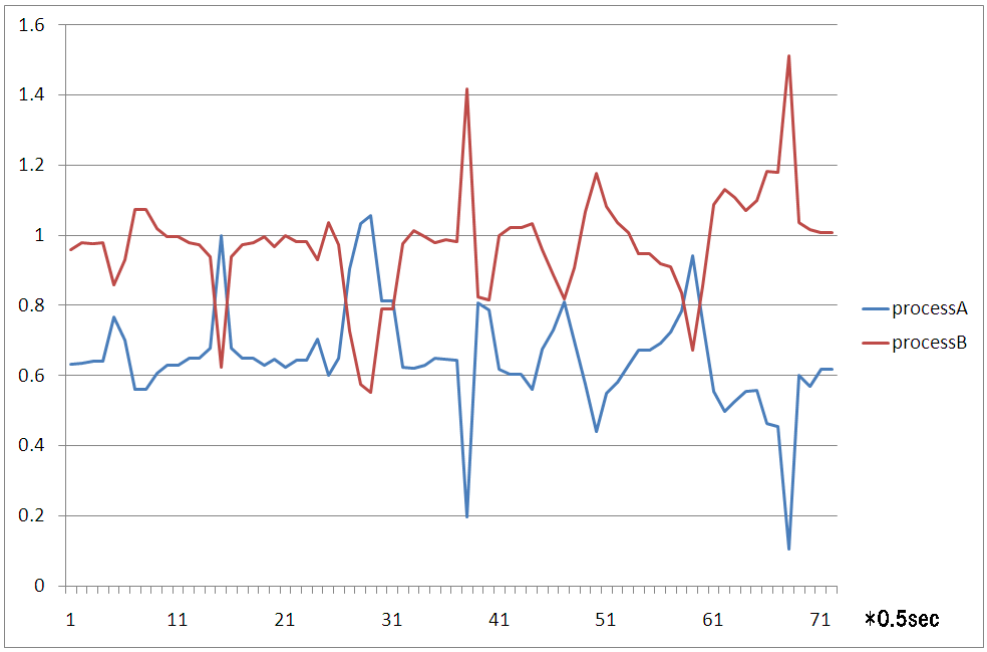

図 5.2: 低 IO 速度プロセス 2 つのサンプリング数 5 の重回帰分析による IO 比率推定

ここではβの値が 1 より大きな値と 0 より小さな値をとることを許しているが, これは特に単一変動推定ではβがこの値域の境界近傍にあり, 誤差などによってわずかに値域から外れてしまった値を見逃すことがあり, これを防ぐためである. ここでは仮に $-0.1 < \beta < 1.1$ とした.

重回帰分析による推定の場合は, サンプリング数が少ないと当然正確な推定が行われない. 今回の例では, サンプリング数が 15 以上で比較的安定した推定が行われている. 10 以下では一部データサーバに 100 % IO をしているプロセスのβの値が 0.8 程度にまで落ちている部分も見られる.

閾値による推定は, 2/3 だけデータサーバに IO をしているプロセス A のβがどの閾値でも 0.66 近辺に落ち着いている.

5.3 複数プロセス時の検証

次に, より複数のプロセスにおける推定を行った.
実験方法としては, 次のように行った.

- IO を行うプロセスは 5 つ存在し, それぞれ別々の計算ノードで動いている
- コード内ではサーバのストレージの速度 (40Mbyte/sec) に対し早くならないように sleep をはさみ速度をある程度抑えている (大よそ 2〜4Mbyte/sec)
- プロセス A から IO をはじめ, B, C, D, E と順に IO が始まっていく
- IO 比率は表 5.3 の様になっている
- IO の値のサンプリングは 0.5sec 間隔で行う

実際のプロセスの IO の様子は図 5.6 のようになっている.

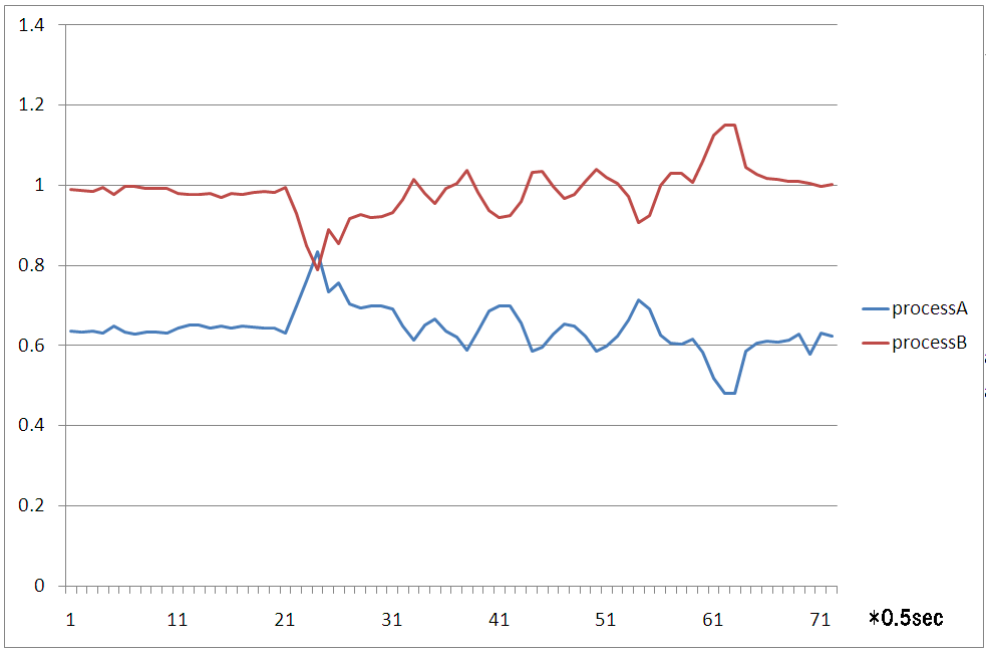

図 5.3: 低 IO 速度プロセス 2 つのサンプリング数 10 の重回帰分析による IO 比率推定

プロセス	A	B	C	D	E
サーバへの IO の割合	66%	100%	33%	66%	100%

表 5.3: 低 IO 速度の各プロセスの IO 比率

この時, 推定の必要性が生じるプロセス B の IO が開始した以降の各プロセスの β の推定値は図 5.7, 5.8 のようになった. 図 5.7 は重回帰分析でサンプリング数は 10, 図 5.8 は単一変動推定で閾値が 10000 Byte で求めた推定値である.

どちらの例においても, 3 プロセスまでは近い値が推定出来ているが, 4 プロセス目が始まると推定値が安定しなくなり, 正しい比率を求めることができなくなった.

重回帰分析の手法ではサンプリング点を増やすことで推定の確実性を上げられるが, この例ではサンプリング点を増やしても, 状況は図 5.7 よりは改善しなかった.

単一変動推定においても, 閾値を変化させても特に改善する様子は見られなかった.

この例では, 4 章の 4.2, 4.3 節で述べたような, 各プロセスの IO 量 x, y がほとんど変化しない場合に当たり, 正しい表 5.3 の比率を 4.1 式にあてはめたときの各計算ノードのサンプリング時間のずれによって生じた誤差と, 誤って推定された比率を 4.1 式にあてはめたときの誤差の値がほぼ変わらない値となり, 誤って推定された比率を誤差が大きく誤っていると判断できないという状態になっていた.

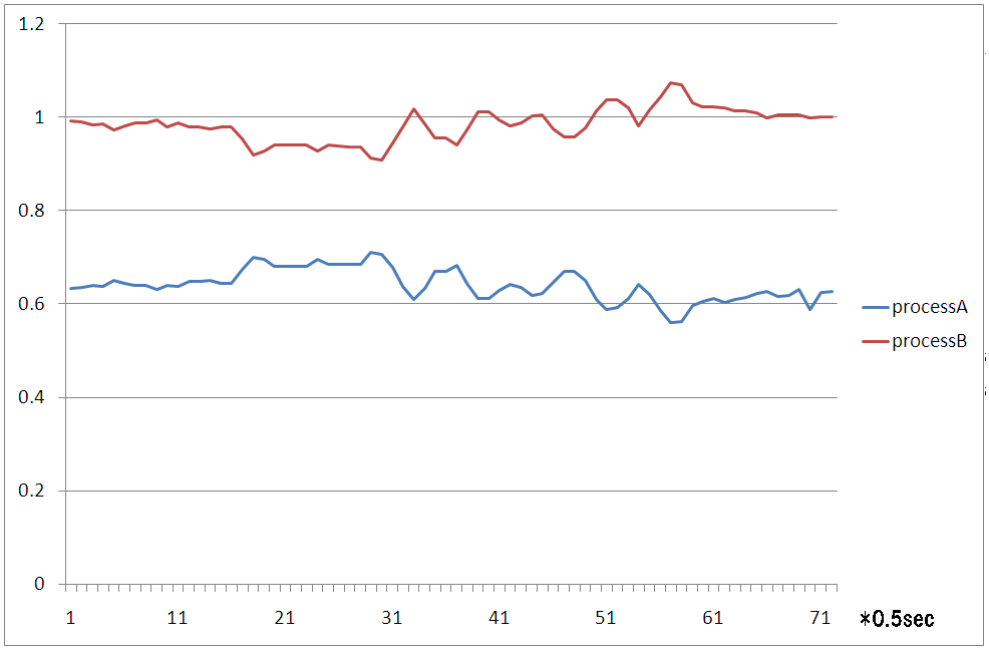

図 5.4: 低 IO 速度プロセス 2 つのサンプリング数 15 の重回帰分析による IO 比率推定

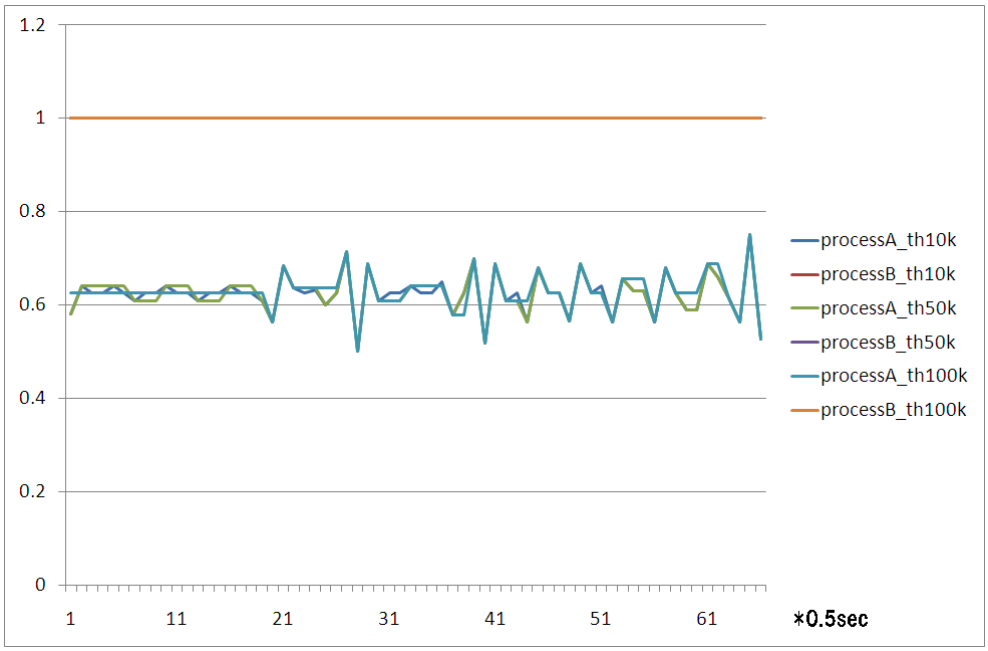

図 5.5: 低 IO 速度プロセス 2 つの単一変動推定の 3 つの閾値による IO 比率推定

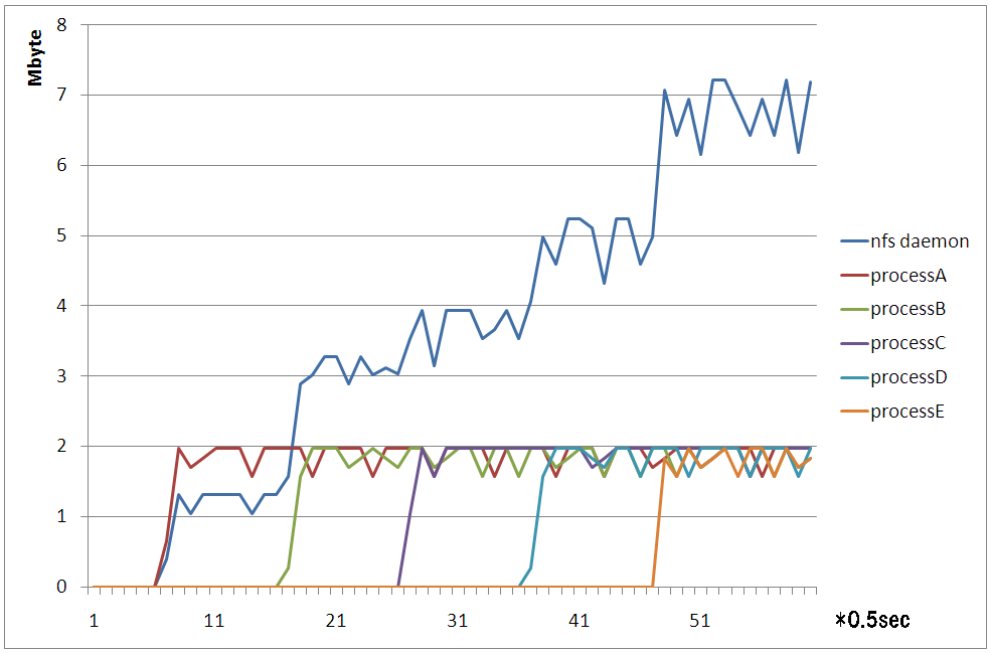

図 5.6: 低 IO 速度プロセスの実際の IO 値の変化の様子

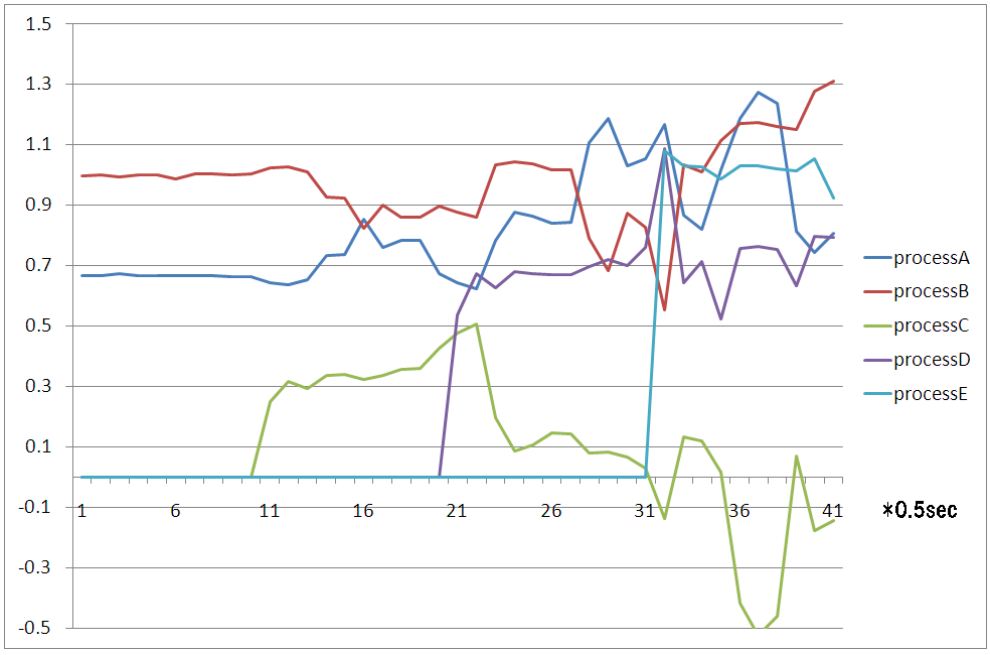

図 5.7: 重回帰分析による低 IO 速度 5 プロセスの IO 比率の推定

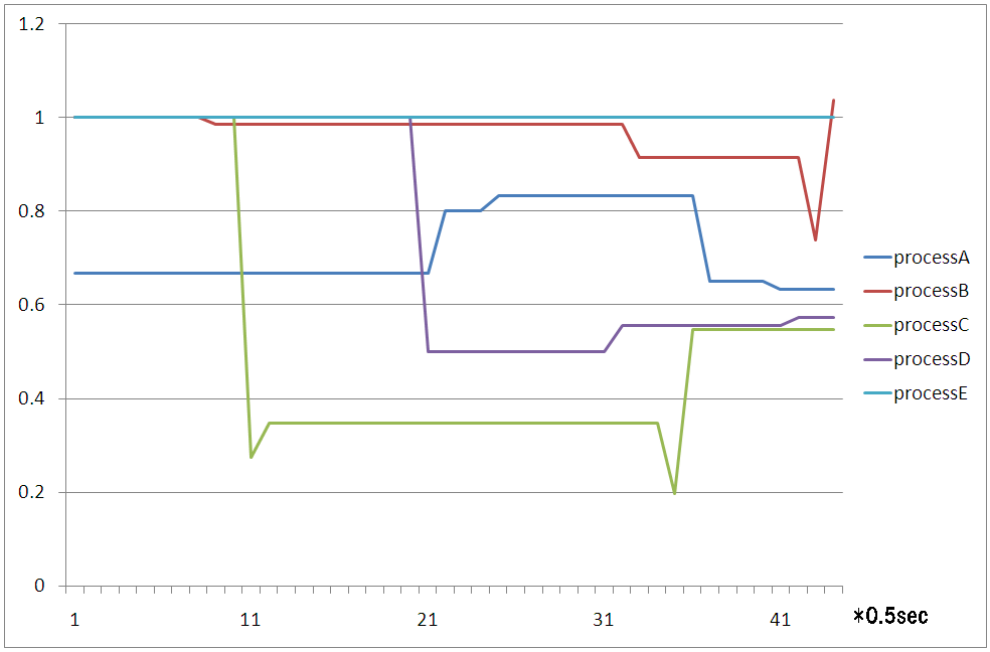

図 5.8: 単一変動推定による低 IO 速度 5 プロセスの IO 比率の推定

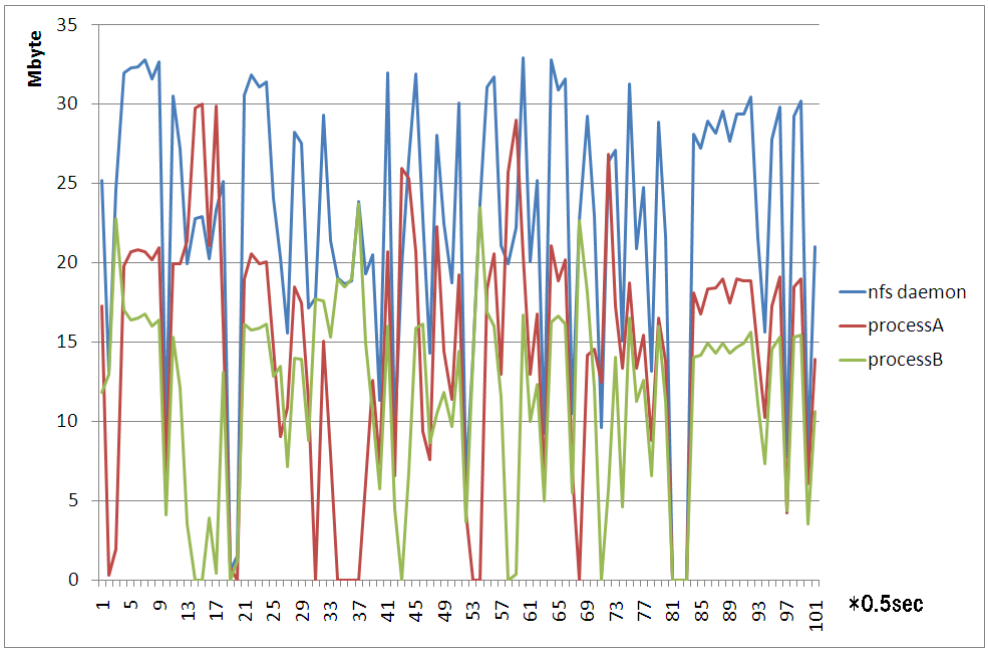

図 5.9: 2 プロセスが限界速度で IO をする場合の IO の推移

5.4 ストレージに最大の IO 速度で IO をしている場合の推定の検証

次に, プロセスが最大の IO 速度で IO をしている場合の推定について検証する. ここで, 最大の IO 速度とは, IO の間に sleep 等を挟まず, IO をするためにブロックしている状態が十分に長いを指す. まず, 次のような実験を行った.

- IO を行うプロセスは 2 つ存在し, それぞれ別々の計算ノードで動いている
- 1 つめのプロセスが先に IO を始める. プロセスの全 IO のうち, データサーバに出力する割合は 76.6% になっている
- 2 つめのプロセスが後から IO を始める. プロセスの IO はすべてデータサーバに行っている.
- IO の値のサンプリングは 0.5sec 間隔で行う

出力割合が 76.6% となっているのは, この比率で書き込む場合にローカルストレージとデータサーバのストレージの IO 速度比と同じとなり, 1 プロセス時に IO 速度が十分大きくなるためである.

図 5.1 は, 実際の IO の様子である. 先ほどの速度を抑えた例と異なり, 時間によって IO のスケジューリングがまちまちになり, プロセスの IO の度合いが大きく変化している.

これに対し, 重回帰分析による方法と, 単一変動推定による 2 つの推定手法をあてはめた. 2 つのプロセスの IO が混じり, 推定の必要が生じる時間帯での推定の結果を図 5.10, 5.11, 5.12, 5.13, 5.14 に示す.

重回帰分析については, サンプリング数 10 以上で安定し, サンプリング数 15 でかなり正確な比率が求められている. 単一変動推定については, 閾値が 100KByte, つまり 1 区間の IO の大よそ 1/100 程度の閾値でも十分安定しており, 閾値を 1Mbyte にしたところ, 比率を一定として求めることができた.

このような乱雑さを持った環境においても IO 比率が求められる理由としては, IO 処理が FIFO に処理されていることが考えられる. 一般に, コード内で IO のリクエストを出した場合, それぞれの書き込み先が異なっていても, 処理は FIFO に行われる. たとえば, 次のようなコードがあったとする.

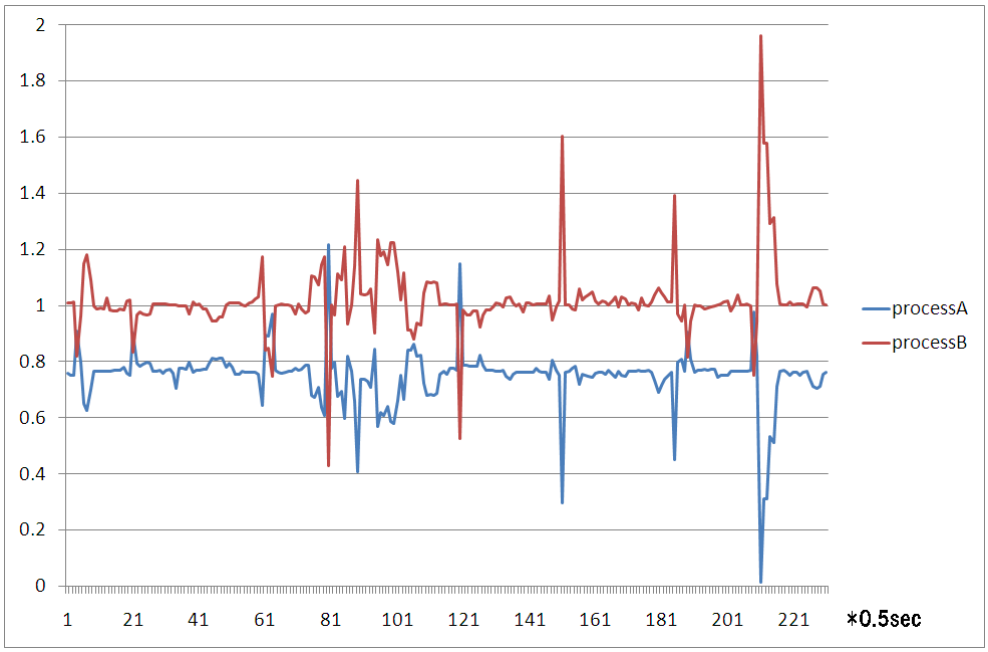

図 5.10: 高 IO 速度プロセス 2 つのサンプリング数 5 の重回帰分析による IO 比率推定の結果

```
for x in range(10):
    f.write(("x"*1023+"\n")*64)
    g.write(("x"*1023+"\n")*64)
    g.write(("x"*1023+"\n")*64)
```

このコードにおいて, f がデータサーバへの IO, g がローカルのストレージへの IO とすると, もしデータサーバの負荷が高まり IO のできない状況であったとすると, たとえ g が書き込みの行える状況であっても f の write でブロックすることになる. 高水準入出力関数によってバッファリングが行われていても, g の書き込みが先に実行されることはない.

また, IO が極力ブロックされないタイミングをはかって IO を行う非同期 IO も同様である. 非同期 IO の例としては libaio がある. libaio は io_context 構造体に IO するデータへのポインタとファイルディスクリプタの情報を持った iocb というタスクを表す構造体をセットして io_submit を実行することにより, ブロックしないタイミングで IO を行い, ブロックしそうなタイミングでは処理を先に進めるという動きをする. この iocb 構造体の配列の並びの順に必ず IO が処理されるため, あいているストレージから先に, といった順番の最適化は起こらない.

IO の順番が変わる可能性がある方法としては, IO を行うプロセス, スレッドを別に立て, そのプロセス, スレッドが IO を行う形式が考えられるが, これはモニタリングの仕組みから IO リクエストを出したプロセスとは別に IO データを集めることができるため気にする必要がない.

しかし, 問題として先に示したコードの 1 回当たりの書き込みが長い場合, この 1 ループがモニタリングの区間に入りきらず, 毎回 IO の比率が変化しているように見えてしまうということが考えられる. 毎回比率が変化する場合, 単一変動推定では比率の推定が難しい. モニタリングの区間を変化させて適切な区間で分析する必要がある.

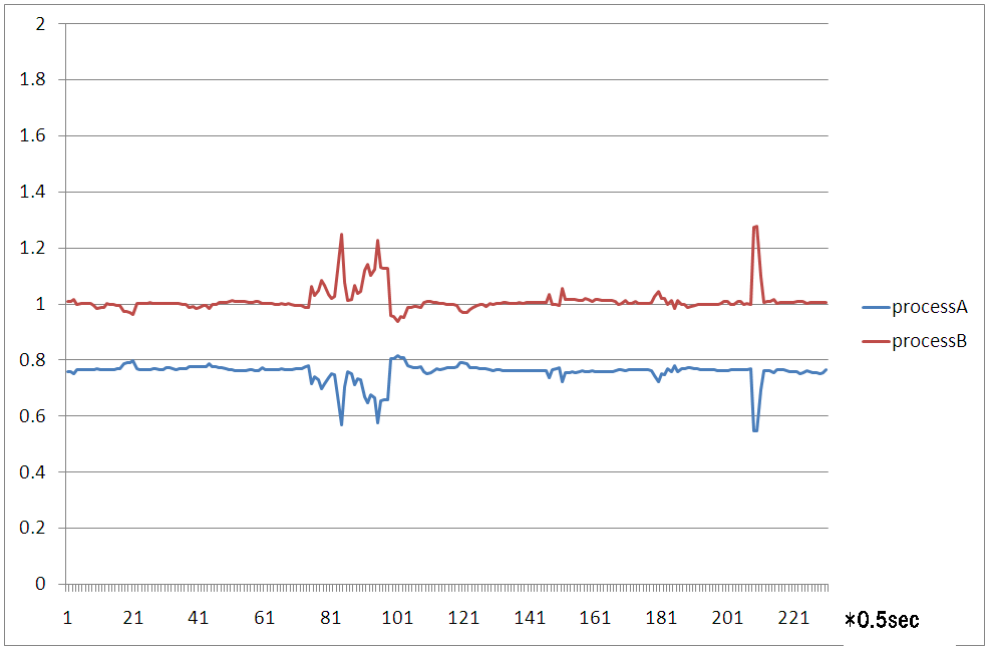

図 5.11: 高 IO 速度プロセス 2 つのサンプリング数 10 の重回帰分析による IO 比率推定の結果

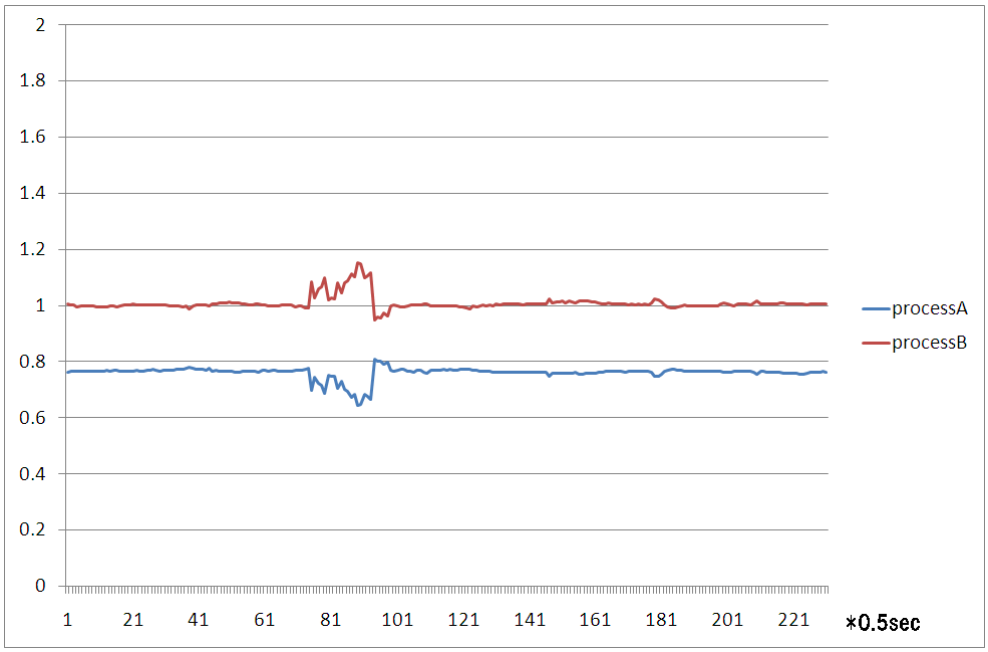

図 5.12: 高 IO 速度プロセス 2 つのサンプリング数 15 の重回帰分析による IO 比率推定の結果

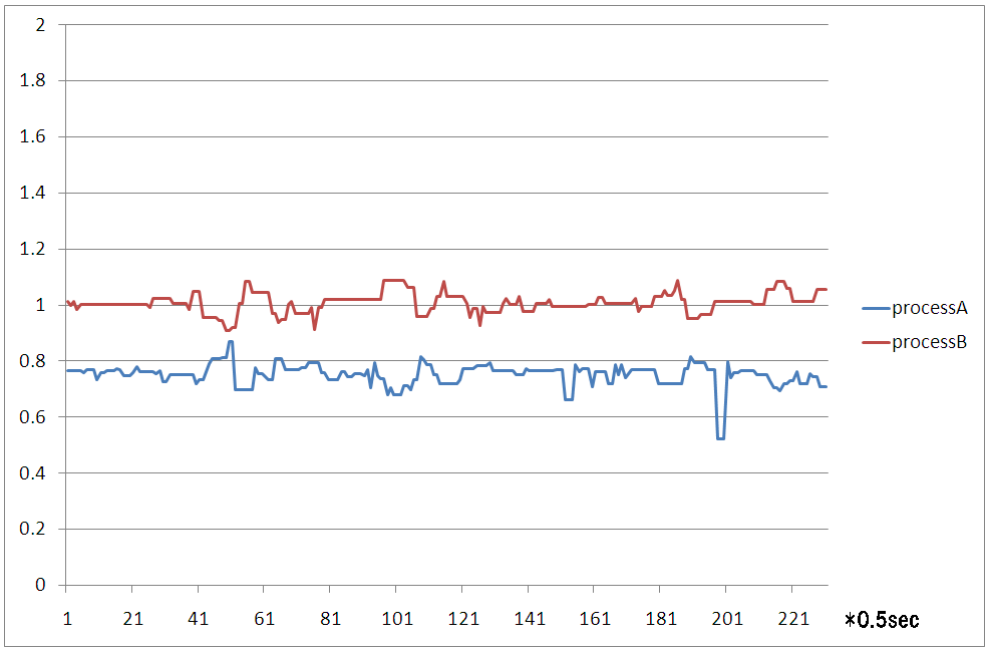

図 5.13: 高 IO 速度プロセス 2 つの閾値 100KByte での単一変動推定の IO 比率推定の結果

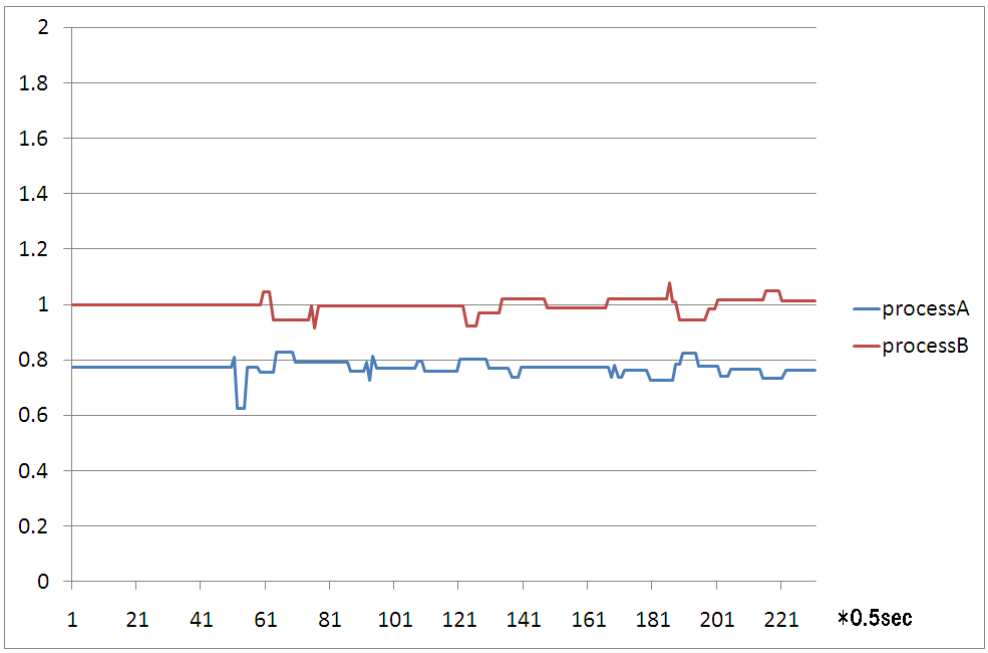

図 5.14: 高 IO 速度プロセス 2 つの閾値 500KByte での単一変動推定の IO 比率推定の結果

プロセス	A	B	C	D	E
サーバへの IO の割合	68%	12%	100%	68%	12%

表 5.4: 高 IO 速度実験の各プロセスの IO 比率

プロセス	A	B	C	D	E	F	G	H	I	J
サーバへの IO の割合	100%	100%	0%	100%	100%	0%	100%	0%	100%	69%

表 5.5: 高 IO 速度実験 2 の各プロセスの IO 比率

5.5 最大 IO 速度かつプロセスが複数の場合の推定

次に, プロセス数を増やして同じ検証を行った.

- IO を行うプロセスは 5 つ存在し, それぞれ別々の計算ノードで動いている
- プロセス A から IO をはじめ, 1 つ目の実験では B, C, D, E と順に IO が始まっていく. 2 つ目の実験ではさらに F, G, H, I, J が加わる.
- IO 比率は一つ目の実験では表 5.4, 二つ目の実験では表 5.5 の様になっている
- IO の値のサンプリングは 0.5sec 間隔で行う

実際のプロセスの IO の様子は図 5.15, 5.16 のようになった.

これに対し, 先ほどと同様に重回帰分析による方法と, 単一変動推定による 2 つの推定手法をあてはめた. すべてのプロセスの IO が混じり, 推定の必要が生じる時間帯での推定の結果を図 5.17, 5.18, 5.19, 5.20, 5.21, 5.22 に示す. 図 5.17, 5.18, が 5 プロセスでの推定を重回帰分析でそれぞれサンプリング数 10, 15 で行ったもの, 図 5.19, 5.20 が 5 プロセスでの推定を単一変動推定で閾値を 100KByte, 500KByte にして行ったもの, 図 5.21 が 10 プロセスでの推定を重回帰分析でサンプリング数 30 で行ったもの, 図 5.22, 10 プロセスでの推定を単一変動推定で閾値を 500KByte にして行ったものである.

これらの図から, プロセス数が 5 の場合, 一見乱雑に見える IO から, 重回帰分析であればサンプリング数が 15 以上, 単一変動推定であればデータサーバの IO 値の 1/100 程度の閾値であれば, かなり正確な比率が求められることが分かる. しかし, プロセス数が 10 になった場合, 推定の値がかなり乱雑になっている部分が見られる.

プロセス数 10 の例では, ほとんどのプロセスのデータサーバへの IO の比率が 100% または 0% になっている. このような単純な例においても, 安定した値に推定出来ていない. そこで誤差の検証をした.

表 5.5 であらわされるデータサーバへの IO の比率を β と考え, データサーバ, 各プロセスの IO 量とこの β を 4.1 式に代入した際の誤差と重回帰分析, 単一変動推定で得た β を使った 4.1 式の誤差を比較すると, ほとんど変わりない値が得られる.

図 5.21 を見ると, 後半部分では, 本来 100% と β が推定されるべきプロセスが, 100% より大きい値と小さい値となっている 2 つにわかれているものがあるが, その分かれ方が 100% を軸に対称になっているものが存在する. この図では, プロセス D と I, プロセス B と G が対称になっている. この 2 組のプロセスは, サンプリングデータでは同じサンプリング区間において対になっているプロセスとほぼ同じ量の IO がたまたまスケジューリングされていた. そのため, 片方のプロセスのデータサーバへの IO の比率を多めにとった分, もう片方のプロセスのデータサーバへの IO の比率を減らすことで偶然つじつまがあってしまい, それが正しい比率として推定されてしまっていた. この例の場合, 最大比

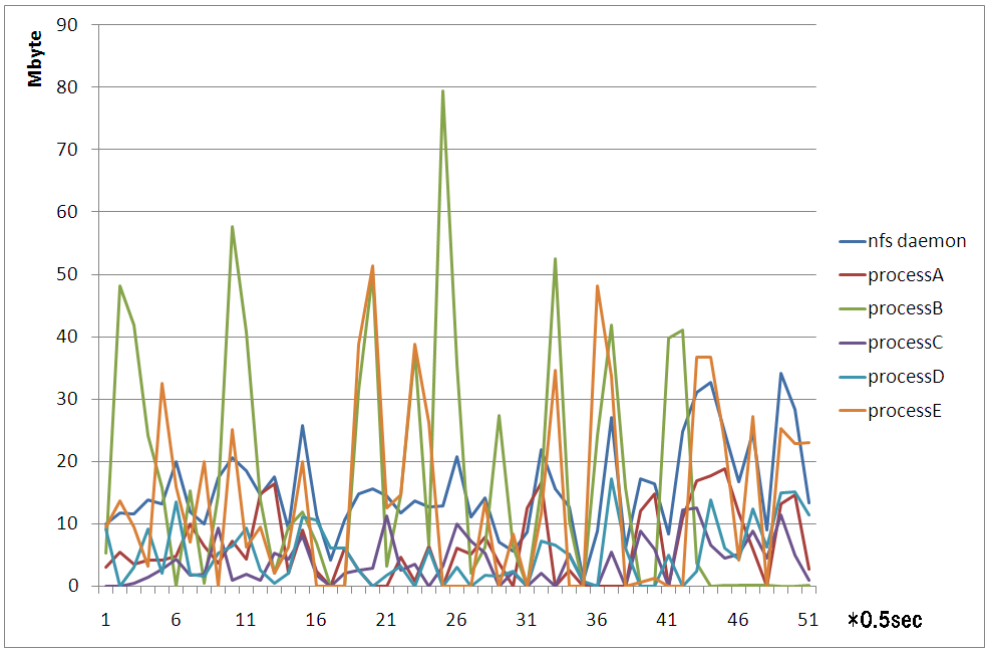

図 5.15: 5 プロセスが限界速度で IO をする場合の IO の推移

率を 100% 以下にすればよいようにも見えるが, 時刻 20〜40 において, プロセス C, H が同じように比率が対称に動いている現象が見られ, 比率の上限下限を 100% 0% で固定したとしても, その枠内に収まるように動いてしまえばやはり推定に失敗してしまう. プロセスのデータサーバへの IO の比率を変え, 同様の実験を行ったが, 似たような現象は何度も確認された. さらに, 図 5.21 の例では同じ IO 比率を持つプロセスが対称に動いているが, 同様の実験で比率を変えたものでは, 比率の違うプロセス同士が対称に動いている例が確認された.

この現象は, この対になっているプロセスの IO 量がたまたま同じだったため, この 2 プロセスに依存関係が見出されてしまい, 4 章の 4.2 節の最後で述べた多重共線性の例に当たってしまったものだと考えられる. “たまたま” とは述べたが, 値を変えて実験をした結果, 5 プロセス以上で IO をする場合, この “たまたま” の場合に陥る可能性が大きいということが分かった.

また, 2 つの推定手法に共通して影響を与えたこととして, 測定間隔の誤差が存在していた. IO 量の測定を行っているプロセスは 0.5sec 間隔で起きて IO 値を調べるということを繰り返しているが, この間隔の誤差が大よそ ±10msec 程度存在している. 今回の実験で用いたストレージの IO 速度は 40Mbyte/sec 程度であり, 10msec では 400KByte 程度の IO ができることになる. データサーバで −10msec, 計算ノードで +10msec の測定誤差が起これば, 2 つのノードで最大 800KByte の誤差が起こりうることになる. また, 数は少ないがタイミングによっては ±400msec の誤差が出る場合も確認された.

図 5.21 の前半部分でプロセス J のサーバへの IO 比率が 69% を下回っていたのは, この大きな誤差によってデータサーバの IO が各プロセスの IO の合計より小さくなっており, そのタイミングでプロセス J の IO が多めにスケジューリングされていたために起こされたものだと考えられる. この誤差はやがて逆の方向へ現れ, データサーバの IO が各プロセスの IO の合計より大きくなる瞬間が存在するが, プロセス J がその時に IO がほとんどスケジューリングされておらず, その時点でプロセス F, H などのほとんどデータサーバに IO を行っていないプロセスがスケジューリングされてしまい, データサーバへの IO の合計のつじつまを合わせようとそれらのプロセスのデータサーバへの IO 比率を大きめに見積もったため, プロセス F, H のデータサーバへの IO 比率の上昇が起こったものと考えら

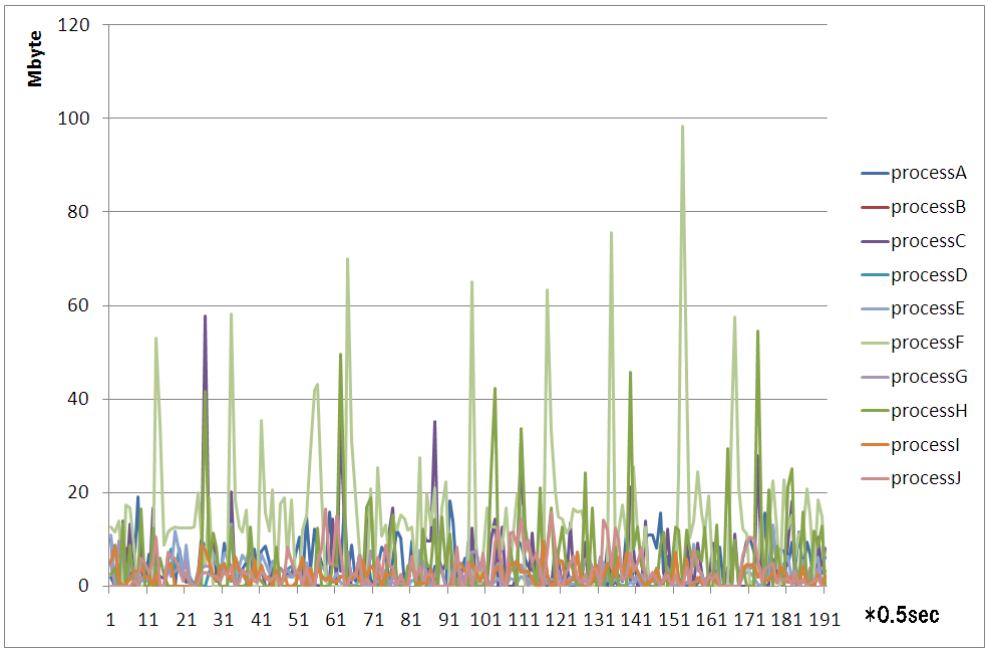

図 5.16: 10 プロセスが限界速度で IO をする場合の IO の推移

れる.

また, 図 5.22 は誤差の影響によって, データサーバ IO の減少をプロセス D や E の IO 比率の減少でつじつま合わせしようとしたが, 次のデータサーバ IO が誤差により大きめに出た瞬間それらの IO スケジューリングが抑えられたため, 測定時間の誤差によるものと見えず, 直近の値の誤差判定が小さく見えてしまい, それが正しいものと認識してしまっていると考えられる.

5.6 検証からの結論

ここまでの検証から, いくつかのことが言える.

- 重回帰分析の場合, プロセス数に対し推定に必要なサンプリング数が増える. よって, プロセス数が多いと細かな比率の推定が困難になる.
- 単一変動推定の場合, 閾値はデータサーバの IO 値の 1/100〜1/50 程度が望ましい.
- IO が限界速度に達していない場合, スケジューリングによるばらつきが少なくなり, プロセス数が多くなるとどちらの推定手法も判別が難しくなる.
- 測定時間間隔による誤差がプロセス数が増えると大きくなり, 推定が困難になる
- スケジューリングのばらつき方が偏り, 複数のプロセスで似たようなスケジューリングが行われると, 多重共線性により重回帰分析による推定が困難になる.

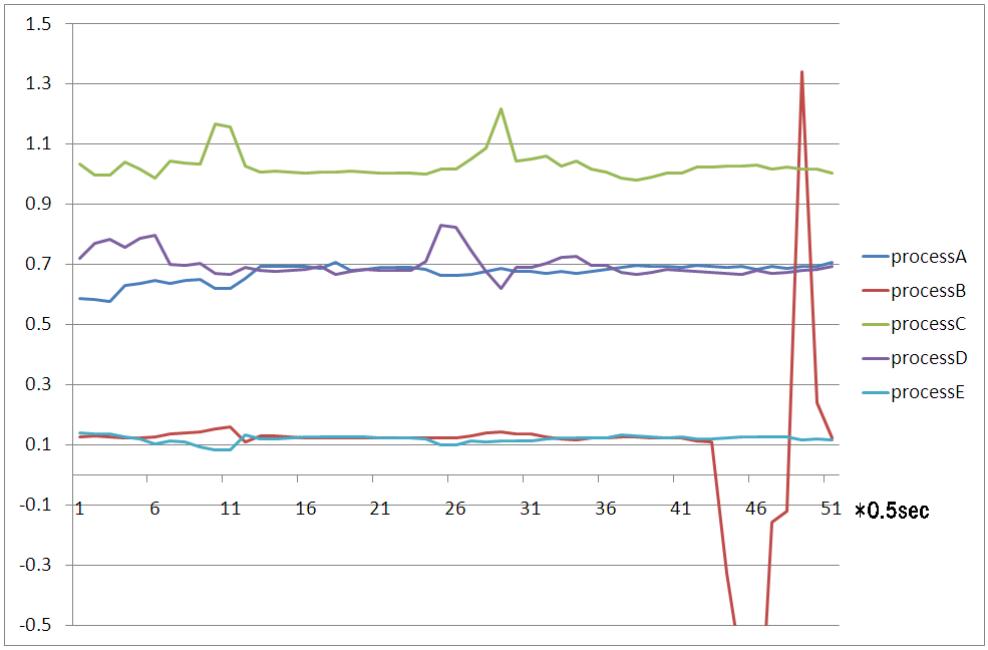

図 5.17: 高 IO 速度プロセス 5 つのサンプリング数 10 での重回帰分析による IO 比率推定の結果

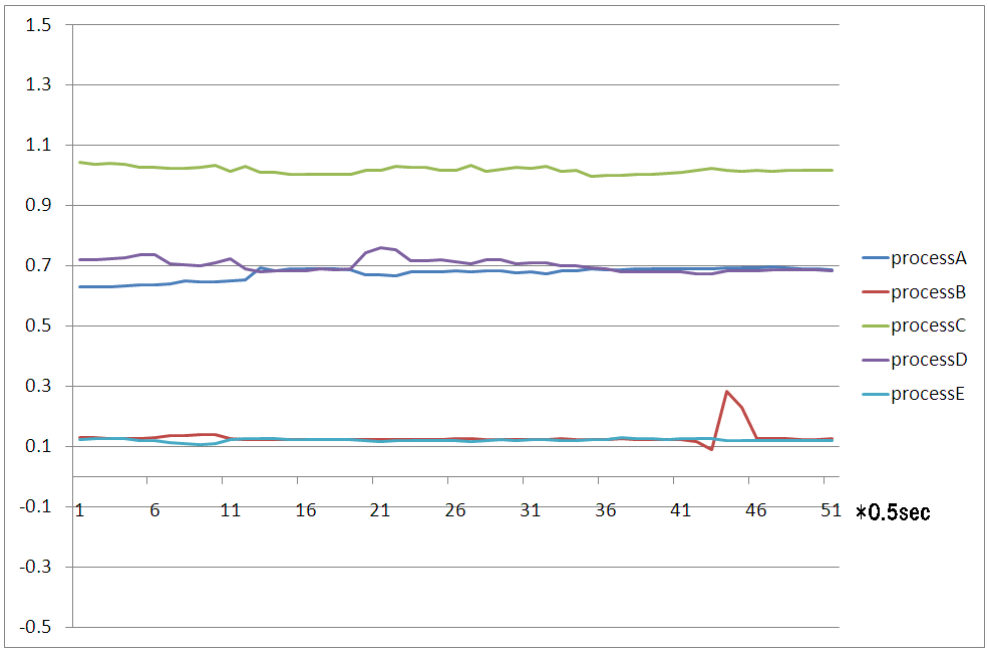

図 5.18: 高 IO 速度プロセス 5 つのサンプリング数 15 での重回帰分析による IO 比率推定の結果

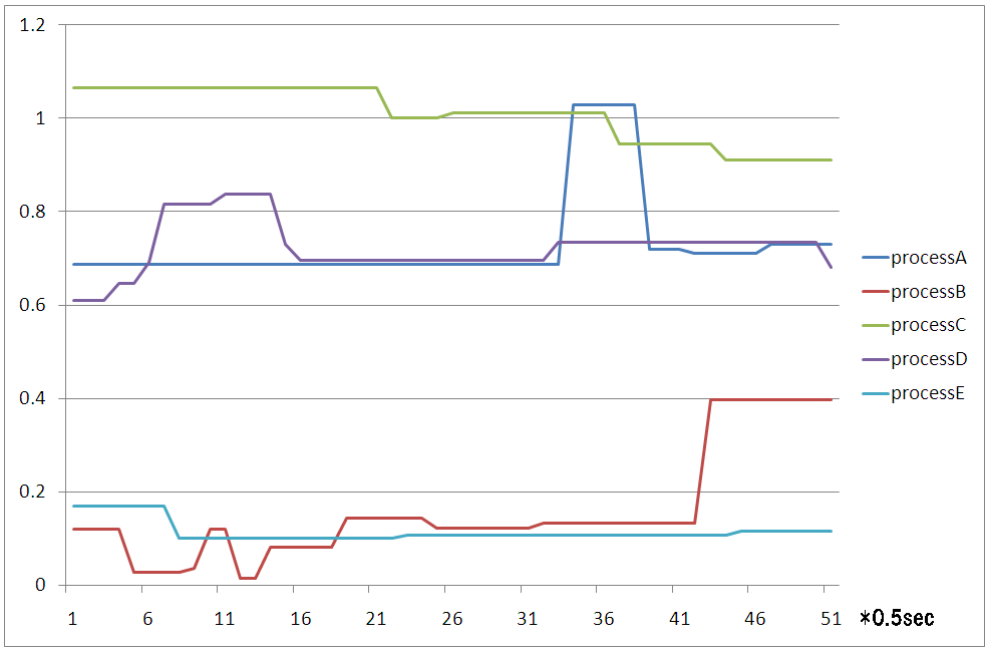

図 5.19: 高 IO 速度プロセス 5 つの閾値 100KByte での単一変動推定による IO 比率推定の結果

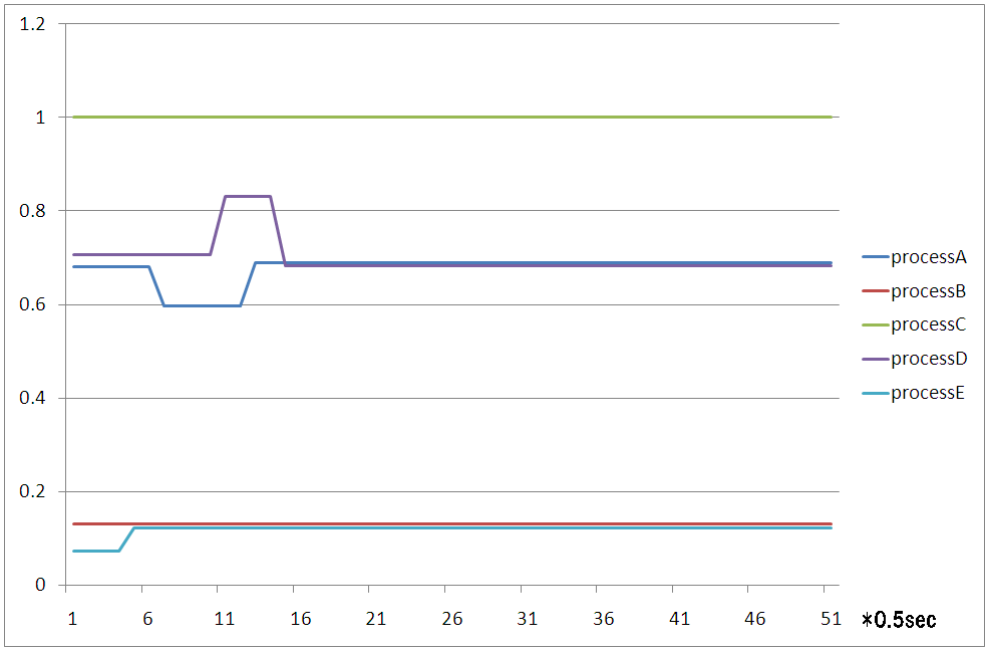

図 5.20: 高 IO 速度プロセス 5 つの閾値 500KByte での単一変動推定による IO 比率推定の結果

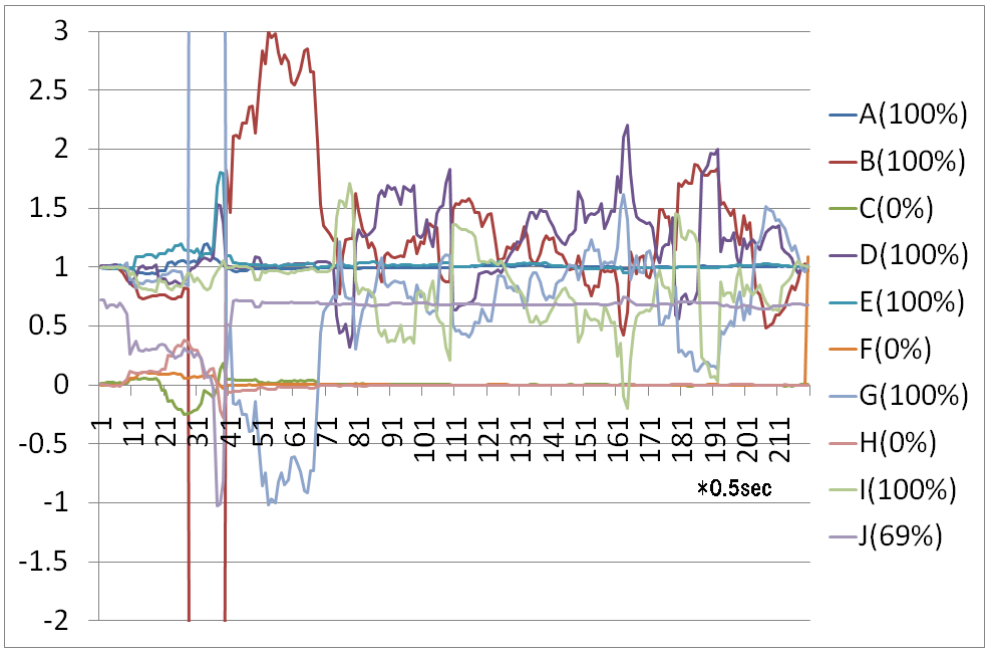

図 5.21: 高 IO 速度プロセス 10 つのサンプリング数 30 での重回帰分析による IO 比率推定の結果

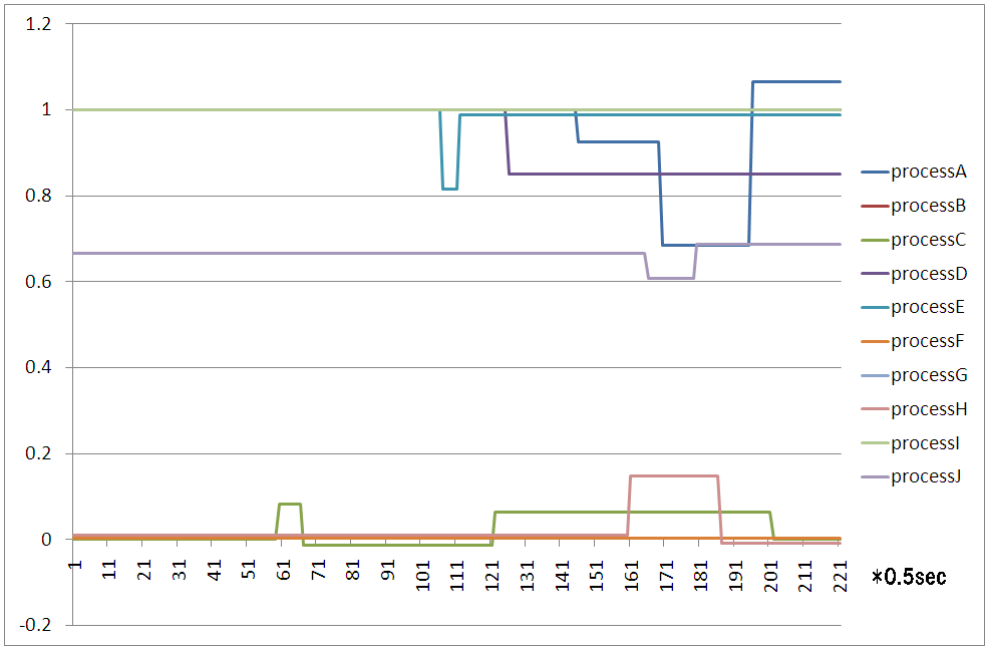

図 5.22: 高 IO 速度プロセス 10 つの閾値 500KByte での単一変動推定による IO 比率推定の結果

第6章　終わりに

6.1　まとめ

本論文では, プロセスの IO とデータサーバデーモンの IO に着目し, プロセス IO のデータサーバへの負荷の割合を, カーネルの変更や FUSE などのオーバヘッドが大きい, もしくは環境の共同利用をするにあたりポリシー上適用の難しい方法を使わずに, 推定する方法について述べた. 情報としては, プロセスがファイルを開いているか, プロセスのデータサーバとローカルを区別しない総 IO 量だけを用い, これらの情報を取得するには /proc ファイルシステムや netlink ソケットなど, 直接的なカーネルの変更を要しない手段しか用いない. また, これらの情報の取得のオーバヘッドは軽量であり, パフォーマンスにも影響を与えにくい.

そして, 推定手法としては重回帰分析と単一変動推定の 2 つの方法を紹介した. 重回帰分析は, 取得した IO 情報をサンプルとして 4.1 式の誤差を最小二乗法によって最小化するように IO 比率 β を決定する方法である. 単一変動推定は, 一つのサンプリング区間の間に一つのプロセスでのみ比率が変化するとして, どの比率が変化したと考えるともっともらしいかを直近の IO の状況から判断する方法である. どちらの方法も, スケジューリングによる各プロセスの IO 速度のぶれにより判断を行っていると考えられる.

この手法を用いて NFS を例に推定手法の検証を行った. 検証においては, 5 プロセス程度まででストレージの限界速度に達するような速度で書き込みを行った場合はその IO の比率を推定できることを確認した. しかし, IO 速度が限界速度に達していない場合に 4 プロセス以上で推定がうまくいかないケースを確認したほか, 5〜6 プロセス以上の場合に IO スケジューリングの偏りや測定時間間隔の誤差によって推定がうまくいかないケースを確認した.

6.2　考えられる課題

本手法における課題として考えられることがいくつかある.

- サンプリング間隔の誤差を小さくすることでより多いプロセス数での推定が可能であるので, その誤差を小さくする.
- 重回帰分析において IO 比率に閾値を設けることで, 誤った推定をはじける可能性を作る.
- カーネルモジュールの積極的利用. 本手法では netlink による間接的利用にとどめていたが, 軽量な利用法も存在しているため, その中から利用できるものを探し, より詳細な情報を用いた解析を行える可能性を探る.
- 今回検証に用いたプロセスよりも一度に行う write の量が多い場合, その量が十分に大きければ各区間を β を 0, 1 の 2 値で決めればよいが, そうでない場合はサンプリング区間による比のばらつきが非常に大きくなり, 重回帰分析では誤差が大きくなって信頼性が低くなり, 単一変動推定では直近の値による変化した β の検証が困難になる. その場合の対処法を考える.
- 1 回のサンプリング区間ですぐに終わってしまう IO プロセスでは推定ができない. その場合の対処法を考える.

References

[1] Bittorrent. http://www.bittorrent.co.jp/.

[2] Fuse: Filesystem in userspace. http://fuse.sourceforge.net/.

[3] ganglia monitoring system. http://ganglia.info/.

[4] glite - lightweight middleware for grid computing. http://www.glite.org/.

[5] gxp make workflow application. https://www.intrigger.jp/wiki/index.php/Applications.

[6] Intrigger. https://www.intrigger.jp/wiki/index.php/InTrigger.

[7] kstrax - kernel systemcall tracer for linux. http://kstrax.sourceforge.jp/.

[8] logfuse. https://www.intrigger.jp/wiki/index.php/Profiler_for_GXP_make.

[9] munin - trac -. http://munin.projects.linpro.no/.

[10] nagios. http://www.nagios.org/.

[11] Network file system. http://www.ietf.org/dyn/wg/charter/nfsv4-charter.html.

[12] paratrac. http://code.google.com/p/paratrac/.

[13] Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium*, Ottawa, Ontario Canada, 2005.

[14] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 74–89, New York, NY, USA, 2003. ACM.

[15] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A file system to trace them all. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 129–145, Berkeley, CA, USA, 2004. USENIX Association.

[16] P. Bhatti, A. Duncan, M. Jiang S. M. Fisher, A. O. Kuseju, A.Paventhan, and A. J.Wilson. Building a robust distributed system: some lessons from r-gma. CHEP '07, 2007.

[17] Peter Bodic, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael I. Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pp. 89–100, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[19] D. Cesini, D. Dongiovanni, E. Fattibene, and T. Ferrari. Wmsmonitor: A monitoring tool for workload and job lifecycle in grids. In *GRID '08: Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pp. 209–216, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pp. 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[21] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. *SIGOPS Oper. Syst. Rev.*, Vol. 39, No. 5, pp. 105–118, 2005.

[22] Dan Geiger, Moises Goldszmidt, and Gregory Provan. Bayesian network classifiers. In *Machine Learning*, pp. 131–163, 1997.

[23] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John C. Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas A. Prince, and Roy Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Int. J. Comput. Sci. Eng.*, Vol. 4, No. 2, pp. 73–87, 2009.

[24] Yinglung Liang, Anand Sivasubramaniam, and Jose Moreira. Filtering failure logs for a bluegene/l prototype. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp. 476–485, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Yinglung Liang, Anand Sivasubramaniam, and Jose Moreira. Filtering failure logs for a bluegene/l prototype. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pp. 225–232, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Michel Dagenais. Mathieu Desnoyers. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Proceedings of the Linux Symposium*, Ottawa, Ontario Canada, 2008.

[27] Jon Stearley and Adam J. Oliner. Bad words: Finding faults in spirit's syslogs. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 765–770, Washington, DC, USA, 2008. IEEE Computer Society.

[28] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IP Operations and Management, 2003. (IPOM 2003). 3rd IEEE Workshop on*, pp. 119–126, 2003.

[29] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp. 375–388, New York, NY, USA, 2006. ACM.

[30] Steve Zhang, Ira Cohen, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp. 644–653, Washington, DC, USA, 2005. IEEE Computer Society.

[31] 曽田 哲之. 建部 修見. 広域分散ファイルシステム gfarm v2 の実装と評価. 情報処理学会研究報告. 2007-HPC-113., pp. 7–12, 2007.

発表文献

[1] 佐伯勇樹, 田浦健次朗, 近山隆. 分散計算機環境における異常動作の原因の特定手法. 2008 年並列／分散／協調処理に関する『佐賀』サマー・ワークショップ, 8 2008.

[2] 佐伯勇樹, 田浦健次朗, 鴨志田良和. 並列分散環境におけるファイル共有システムの負荷原因探索システム. 2009 年並列／分散／協調処理に関する『仙台』サマー・ワークショップ, 8 2009.

謝辞

本研究を進めるにあたり, 多くの方に大変にお世話になりました. 特に田浦健次朗准教授には, ミーティングにて何度も研究の方向性について討論をし, ご指導, ご教示をいただきました. また, 情報基盤センター特認助教の鴨志田良和さんには, ミーティング以外の場においても何度も相談に乗っていただきました. そして, 近山田浦研究室の皆様とも, 議論を通してたくさんのことを学ぶことができました.

ここに, 心から感謝の意を述べます. ありがとうございました.

February 9, 2010