修　士　論　文

相対プロファイリング: 比較に基づいた並列アプリケーションの性能解析手法

Relative Profiling: Performance Analysis Method for Parallel Applications Based on Comparison

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

氏 名　48-106404 **加辺 友也**

提 出 日　**平成 24 年 2 月 8 日**

概 要

アプリケーションの速度面での性能を向上させるため, 旧来からプロファイラが開発され, 利用されてきた. プロファイラはアプリケーションの性能を解析し, 開発者がそれを理解し, 高速化につなげる手助けをする役割を果たしてきた. これまでの性能解析はアプリケーションを 1 回実行し, そのときの挙動を解析することによって為されるものが大半であり, 近年では複数回の実行性能を比較して解析するためのツールも開発されている. また, 現在では並列アプリケーションへの対応は事実上必須となっている.

本研究ではこれらのものとは異なり, 並列アプリケーションの性能解析手法として新たに相対プロファイリングを提案する. 相対プロファイリングでは異なる実行条件での性能差を解析することにより, 原因を突き止め, 性能改善につなげる.

本稿では相対プロファイリングを行うための実装 PARP を提案し, これを用いて評価を行った. PageRank を計算する MPI アプリケーションを用いた例では, 通信部分の性能が著しく劣化してしまった原因が誤った MPI の利用方法によるものであることを明らかにした. 日本語文書の tf-idf を計算するワークフローアプリケーションを用いた例では, Tsubame2 と Amazon EC2 を用い, 性能差の原因を CPU 計算とそれ以外の部分に切り分け, CPU 計算部分はほぼ期待される CPU 性能の差と等しく, またそれ以外の部分が性能差の大部分を占めていることを明らかにした. さらに, Tsubame2 において 2 つのファイルシステムで同じアプリケーションを動作させ, ファイルアクセス性能を解析することにより, 性能差に関わるファイルアクセス操作がメタデータアクセスであったことを明らかにし, 相対プロファイリングの有用性を示した.

目 次

第 1 章　はじめに **1**
1.1 背景: プロファイラ 1
1.2 並列アプリケーションのプロファイラ 1
1.3 比較に基づくアプリケーションの性能解析 2
1.4 本研究: 相対プロファイリングと PARP 2
1.5 本研究の貢献 2
1.6 本稿の構成 3

第 2 章　関連研究 **4**
2.1 性能情報データベース 4
2.1.1 Perfbase 4
2.1.2 PPerfGrid 5
2.1.3 PerfDMF 5
2.2 比較に基づく性能解析 6
2.2.1 CUBE 6
2.2.2 PerfExplorer 7
2.2.3 Periscope 7

第 3 章　対象アプリケーションと用語の定義および要素技術 **9**
3.1 MPI アプリケーション 9
3.2 アプリケーション内の計量 9
3.3 ワークフローアプリケーション 10
3.3.1 GXP Make 12
3.4 ParaTrac 12

第 4 章　設計 **14**
4.1 設計概要 14
4.2 性能情報データベース 14
4.2.1 MPI 向け性能情報データベース 14
4.2.2 ワークフロー向け性能情報データベース 14
4.3 コマンドラインインタフェース 16
4.4 Web インタフェース 16

第 5 章　実装 **18**
5.1　性能情報データベース 18
5.1.1　MPI 向け性能情報データベース 18
5.1.2　ワークフロー向け性能情報データベース 20
5.2　GXP Make の拡張 21
5.3　コマンドラインインタフェース 21
5.4　Web インタフェース 22
5.4.1　memcached の利用 22
5.4.2　MPI 向け Web インタフェース 23
5.4.3　ワークフロー向け Web インタフェース 25
5.4.4　PARP 順序計量 28
5.5　ParaTrac データの統合 29

第 6 章　評価 **33**
6.1　評価環境 33
6.1.1　クラスタ構成 33
6.1.2　Amazon EC2 33
6.1.3　Tsubame2 33
6.2　MPI アプリケーションを対象とした解析 34
6.2.1　PageRank アプリケーション 34
6.3　ワークフローアプリケーションを対象とした解析 40
6.3.1　日本語ワードカウント 41
6.3.2　ParaTrac の情報の統合 44

第 7 章　おわりに **48**
7.1　まとめ 48
7.2　今後の課題 48
謝辞 49
発表文献 50
参考文献 51

付 録 A　ソースリスト **57**
A.1　SQL 57
A.2　Shell Script 64

図目次

2.1 PerfDMF のアーキテクチャ ([43] より) 6
2.2 PerfExplorer による S3D アプリケーション解析結果 ([26] より) 8

3.1 内包的実行時間と排他的実行時間 10
3.2 ペトリネットと DAG の対応 11
3.3 ParaTrac の概要 13

4.1 PARP 概要図 15

5.1 MPI 向け性能情報データベースの E-R 図 19
5.2 ワークフロー向け性能情報データベースの E-R 図 20
5.3 PARP Web インタフェースにおけるプロファイル選択 (MPI 向け) 24
5.4 MPI アプリケーションの実行条件サマリ画面 30
5.5 ワークフローアプリケーションのアプリケーション選択画面 31
5.6 ワークフローアプリケーションの実行条件サマリ画面 31
5.7 ワークフローアプリケーションの実行可視化例 32

6.1 評価に用いたクラスタのネットワーク構成図 34
6.2 PageRank アプリケーションの平均実行時間と標準偏差 35
6.3 PageRank アプリケーションの比較に用いるカラム指定画面 36
6.4 PageRank アプリケーションの比較用グラフに表示する軸設定画面 37
6.5 PageRank アプリケーションの比較グラフ 38
6.6 PageRank アプリケーションの MPI_Send によるメッセージ送信数 (ParaProf より) . 39
6.7 PageRank アプリケーションの MPI_Send によるメッセージ総サイズ (ParaProf より) 40
6.8 PageRank アプリケーションの比較グラフ（PARP 順序計量を用いた場合） 42
6.9 日本語ワードカウントアプリケーションの DAG 43
6.10 PageRank アプリケーションの比較グラフ（PARP 順序計量を用いた場合） 43
6.11 日本語ワードカウント Tsubame2 と Amazon EC2 の比較 44
6.12 リスト 6.5 による性能比較結果 45
6.13 日本語ワードカウント Lustre と GPFS の性能比較 46
6.14 Tsubame2 におけるファイルアクセス性能比較結果 47

表目次

6.1 評価に用いたクラスタ環境の一覧 33
6.2 Amazon EC2 システム概要 34
6.3 Tsubame2 システム概要 35
6.4 評価に用いる PageRank アプリケーションの入力データ 35

第1章　はじめに

1.1　背景: プロファイラ

プロセッサがマルチコアとなり, それを搭載したサーバ機が安価になりつつあることから, 並列計算を行う機会が増えつつある. 並列処理はノード内であればマルチスレッドを用いた並列化が行われ, POSIX thread を更に使いやすくするために OpenMP[10], Cilk[9], MassiveThreads[53] などが提案されている. 科学技術計算分野のアプリケーションを並列化するにあたっては, 複数のノードにまたがる計算に対応するため, 以前から MPI[17] が用いられ続けている.

これら並列アプリケーションの開発者にとって, デバッグと高速化は逐次型プログラムに比して非常に困難な作業である [29, 50]. いずれも困難な課題であるが, 本稿では後者の高速化に焦点を当てる.

一般に, アプリケーションの高速化は性能情報を取得する部分と, それを解析する部分に分離することができる. 多くの場合, これらはまとめてプロファイラと呼ばれている. 古くから用いられている逐次型アプリケーション向けのプロファイラには gprof[20] や sprof[5] をはじめとして, 言語, 目的に応じて様々なものがある. たとえば OS のシステムコールの性能解析に特化した OProfile[37] や, 呼び出しパス [22, 40] に着目したもの [8] がある. これらは対象プログラムが 1 プロセスしか動作しないことを前提としている. 一方, 並列アプリケーションを対象としたプロファイラではあるプログラムのプロセスが複数実行され, それぞれについて性能情報を取得および記録しなければならないことから, 逐次型のものをそのまま用いることはできず, ある程度複雑な機構を要する. これについては後述する.

プロファイラにはアプリケーションを実行させながら情報を取得し, リアルタイムに性能解析を行う On-line なもの [39, 33, 13] と, アプリケーションが終了する際に性能情報のログをストレージ（ファイルまたはデータベース）に出力する Post-mortem なものとがある. On-line なものはサーバのように長時間動作させ続ける必要があるアプリケーションや, 非常に並列度が高いアプリケーションに用いられることを想定している. 長時間の動作によって出力される性能情報が全体として非常に大きくなってしまい, これをアプリケーション終了時まで全て蓄えて処理することが現実的でない場合を考慮しているためである. ここでは性能情報の解析処理がある程度必要となることを考慮し, Post-mortem なものを対象とする.

1.2　並列アプリケーションのプロファイラ

並列アプリケーション向けのプロファイラは, それぞれがある程度独立に動作していることから, 性能情報の出力も独立に行われるのが普通である. すなわち, 各並列ユニット（プロセス, スレッドなど）ごとに別々のファイルへとログが出力される. ファイルの数が増えることにより, 並列プ

ログラムではそれぞれのログがどのプロセスのものであるかを区別するのが容易でなくなるので, ファイル名が別々であるだけでなく, ログファイルの中にメタ情報として実行時の環境（MPI であれば rank など）が含まれることがある. 並列ユニット同士の通信情報をログする場合, 通信相手や通信のサイズの情報を取得し, 通信の様子を時系列で表示することがある.

1.3　比較に基づくアプリケーションの性能解析

ほとんどのプロファイラはアプリケーションを 1 回実行し, その時の性能情報を可視化することに特化している. ユーザはそのなかで時間のかかっている箇所を特定し, 高速化を試みることになる. しかし, 並列度が増加すると, 単に時系列での動きをみても何が起こっているかを正確に判断するのは難しく, また長時間のプロセスでは多くのイベントが発生しているため, 実際に性能のボトルネックを見付けることは困難になる. 時系列での表示ではなく, 自動的に性能のボトルネックを見付けるプロファイラも存在するが [37, 46], どの部分が性能面で実際に問題となっているかを判定するのに, 1 回の実行では問題かどうかが分かりにくい.

そこで, 2 つ以上のログを用いることでそれらを比較し, 違いをみつけることで性能解析を行うというアイデアが生まれる. 性能が異なるところがあるということは, 一方がもう一方より遅いということであり, 性能が大きく異なっている箇所は性能向上の余地があるといえる. この部分を改良することにより, アプリケーション全体としての性能向上を見込むことができる.

1.4　本研究: 相対プロファイリングと PARP

これらの背景を踏まえ, 本研究では**相対プロファイリング**を提案する. これは並列アプリケーションの性能比較手法であり, 性能情報の比較に基づいていることが特徴である. また, この相対プロファイリングを用い, 並列アプリケーションの性能解析を目的とし, 並列アプリケーションの性能を比較に基づいて解析するツール *PARP* を提案する.

基本的な設計は既存の研究に倣い, 性能情報データベースと解析用インタフェースを分離する. 性能情報データベースを共通とし, PARP 標準ツール以外からも扱いやすくすることを意図している. PARP は性能情報データベースに登録された性能情報を一元管理し, 性能情報を比較するのを容易にする.

PARP は以下を特徴とする性能解析ツールである:

1. 2 つの異なる環境における, あるアプリケーションの性能を比較することができる.
2. ワークフローアプリケーションの性能を比較によって解析することができる.
3. I/O 性能の解析を行うことができる.

1.5　本研究の貢献

本研究は並列アプリケーションの高速化に役立つことを目指しており, 本稿では実際に性能改善に至った例を示す. この例では特に, アプリケーションの実行のさせ方により大きく性能が変わりうることを示している. たとえば MPI アプリケーションではネットワークの使い方により非常に

大きな性能差が出る. ワークフローアプリケーションでは CPU 性能とそれ以外の性能差がどのような割合であったかを簡単な操作で明らかにする. ParaTrac の情報を統合することにより, I/O 性能を解析し, 異なるファイルシステムを用いたとき, ファイルアクセス性能がどのように性能に影響するかを明らかにする. 本研究により多くの並列アプリケーションが性能解析の対象となり, 通常なら見過ごされてしまう上記のような性能劣化をユーザに気付かせ, 性能を改善させることができる.

1.6　本稿の構成

本稿は以下の通り構成されている. 第 2 章では本研究に関連する, 性能情報データベースと性能情報の比較に関する既存の研究について述べる. 第 3 章では本研究で解析対象となるアプリケーションと, それに関する用語について定義を述べ, 要素技術について述べる. 第 4 章では本研究で提案するシステムの設計について述べる. 第 5 章では本研究での提案システムの実装を行ったため, この実装について詳細を述べる. 第 6 章でこのシステムの評価を行い, 第 7 章で本稿をまとめる.

第2章　関連研究

本研究は性能情報データベースの設計及び比較に基づく並列アプリケーションの性能解析からなっている. それぞれについて関連研究を述べる.

2.1　性能情報データベース

逐次アプリケーションの性能解析を目的とした研究は数多くなされてきて, これまでに多くの業績が示されていることは広く知られた事実である. 現在では, アプリケーションの性能解析は多くの研究が並列アプリケーションを対象とするものになっている. 並列アプリケーションの性能は, そのログを最初から 1 箇所に出力するとアプリケーションの性能が劣化しがちであるため, 各並列ユニットごとに別々のログファイルに性能情報を出力するのが普通である. 結果として, N 並列のアプリケーションを動作させると N 個の性能情報を含んだログファイルが出力されることとなり, ログの管理が困難になることが指摘されている [51]. また, このときの性能情報ログはテキスト形式である場合が多いが, この形式もツールによってまちまちであり, 解析のたびにその形式のデータを読み取る努力を, ユーザ側が強いられてしまうという問題点があった. そこで, **性能情報の管理**が 1 つの分野として研究されている. ここでは前述の問題点を踏まえ,

1. 並列アプリケーションを 1 回動作させた性能情報ログを, 1 つの塊となったデータとして扱えるようにすること
2. 複数の形式のデータであっても統一された形式に変換し, ユーザがデータを解析しようとするときはその統一形式によるアクセスのみで済むようにすること

を目指している.

2.1.1　Perfbase

Perfbase[51] は並列アプリケーションの実行, 性能情報のデータベースへの格納, XML によるクエリ記述による取り出したい情報の抽出, 出力形式の変換を自動で一括に行えるシステムである. Perfbase では単位変換にも着目している. プロファイルに含まれる情報は時間情報や回数（関数の呼び出し回数など）, 割合など, いくつかの異なる意味をもつ情報が含まれている. 時間であれば, プロファイラによってミリ秒, マイクロ秒と, 同じ時間であっても異なる補助単位をもつ場合がある. Perfbase ではデータベースに情報を登録する際にこれらの情報も保持しておき, 取り出すときも好きな形式に変換して取り出すことができる. 出力はテキスト形式と GNUPLOT によるグラフ出力に対応している.

Perfbase を使うにあたって, ユーザは主に 3 つの XML ファイルを作成し, システムの利用に用いる.

1. アプリケーションの実行のため, 実行のための入力となるパラメタ（パラメタスイープを含む）, 出力されるデータの形式を指定する.
2. アプリケーションの結果を読み取り, データベースに格納するための情報を記述する. 出力ファイル名や出力ファイルからのデータ読み取り形式を指定する.
3. データベースに格納されたデータを必要な分取り出し, 何らかの形式で出力させるために用いる. 複数回実行したときに条件を満たしたもののみを利用する記述, GNUPLOT によるグラフとして出力させたい場合はどのようにグラフを作成するかなどを指定する.

これにより, 性能の差を出力するなど, 単純な演算結果をみることが可能になっている. しかし Perfbase ではバックエンドに SQL をサポートしたデータベースを用いているにも関わらず, すでに準備された形式での入力や出力, 演算処理にしか対応しておらず, また, ある条件で複数回実行したときに, その平均を用いて処理をするといったことができない. 本研究では, SQL のデータベースでサポートされている全演算を利用可能にし, 同じ条件での複数回実行であっても, 平均化など自由な処理を可能にしている.

2.1.2 PPerfGrid

PPerfDB およびその後継である PPerfGrid[23] は章頭で述べた目的を達成するため, データベースを置くというアプローチではない, 別の手段を用いている. アプリケーションは様々なものが様々な場所で実行され, ログは実行直後には各地にばらばらになっている. データベースに格納するというアプローチではそれをある 1 箇所に集めなければならない. PPerfDB や PPerfGrid ではデータそのものを集めるのではなく, そこにあるという情報を登録し, 仮想ビューを提供することによって統一された形式をユーザに提供することを目指している.

PPerfDB や PPerfGrid は性能情報を集めないという考え方をとっている. これらの研究の論文では大規模な実験データに対するツールそのものの性能評価は行われていないが, 元のデータは独立したファイルのままであり, 大きなデータを処理しようとすれば性能がよいものであるとは考えにくい. また, 現在まで実装が公開されておらず, 実際に利用することができないという問題がある.

2.1.3 PerfDMF

PerfDMF[25] は TAU[43] とともにオレゴン大学で開発されている性能情報管理ソフトウェアである. PerfDMF のアーキテクチャ図を図 2.1 に示す. TAU も多くの並列プロファイラと同じく, 基本的にはテキストファイルを並列ユニットごとに出力するため, ログファイルの数は膨大になる. PerfDMF はその問題点に着目し, 実行直後には多くのファイルが存在しても, データベースに格納することで統一された性能データベースを提供することを目的としている.

PerfDMF は TAU の形式のみならず, gprof, mpiP, psrun, HPMtoolkit, dynaprof といった別のプロファイラの性能情報形式のデータであってもデータベースにデータを格納できるよう, ツールが充実している. また SQL のインタフェースを標準として提供しているが, それだけでなく pprof, ParaProf[7], JumpShot[52] といった性能情報可視化ツールとも密に連携しており, 性能の解析に関して一通りを TAU を中心として可能にしている. 主に MPI での並列化を行ったアプリケーション向けに実装されており, 複数回実行した際の平均を取るような操作も行いやすいようにデータベー

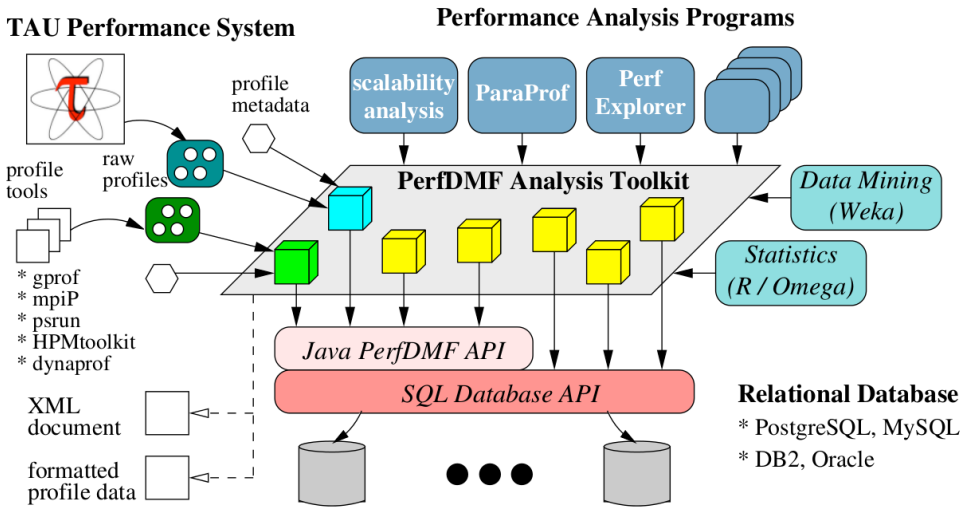

図 2.1. PerfDMF のアーキテクチャ ([43] より)

スが設計されている. 本研究で用いるデータベースの設計は多くの部分で PerfDMF を参考にしており, これに拡張を加えたものとなっている.

2.2　比較に基づく性能解析

2.2.1　CUBE

CUBE[44] は並列アプリケーションの性能を比較するインタフェースを備えた性能解析ツールの実装としては最も古い部類にあたる. CUBE では並列アプリケーションの実行ログが必ず集合 S の要素であるとし, 任意の S の要素を可視化できるツールを実装した. さらに, 複数の実行ログ $A_i \in S (i = 0, 1, \ldots)$ に対し, 3 つの演算を定義した.

- 2 つの実行の差

$$D = A_j - A_i (i, j \in \{0, \ldots, N-1\}) \tag{2.1}$$

- N 個の実行の平均

$$M = \frac{\sum_{i=0}^{N-1} A_i}{N} \tag{2.2}$$

- 2 つの実行の併合

$$G = A_i \cup A_j \tag{2.3}$$

3 つ目の併合は, 1 つのツールのみからでは十分なデータが得られない場合に, 異なる複数のツールからのデータを統合するために用いる演算である.

ここで注意すべきは, D, M, G のいずれも S の要素であることである. この性質を満たすことにより, CUBE では単に複数回の実行平均をとることのみならず, 条件 α での実行ログ $\alpha_0, \alpha_1, \ldots, \alpha_M$

と 条件 β での実行ログ $\beta_0, \beta_1, \ldots, \beta_N$ があったとき, その平均の差

$$D_{\alpha\beta} = \frac{\sum_{i=0}^{M-1} \alpha_i}{M} - \frac{\sum_{j=0}^{N-1} \beta_j}{N} \tag{2.4}$$

といったものであっても S の要素であるから, CUBE により可視化を行うことができる.

CUBE を提案している論文は上記の演算と, これにより統一された可視化ツールで様々なものを可視化できることに主眼をおいており, 並列アプリケーションの性能を比較すること自体には多くの注目をしていなかった. 可視化する情報も, プロファイラに格納されている数値をそのまま, または上記の演算を通して利用することしかできなかった. また, 一度に可視化できる情報も 1 種類のみとなっており, 様々な性能解析を行う上では自由度の低いものとなってしまっていた. しかし CUBE は初期の実装という点では意義の大きなものであり, 以後のツールではこれを大いに参考にしたと思われる可視化が行われている.

2.2.2 PerfExplorer

PerfExplorer[24, 26, 27, 28] は TAU と連携し, 性能情報を可視化するツールとして開発されている. 中でも文献 [26] では, 性能情報を比較することにより解析を行う例が示されている. ここでは Oak Ridge National Laboratory (ORNL) における Cray XT3 と Cray XT4 の混在計算環境において MPI アプリケーションを動作させ, `MPI_Wait` に要した時間が MPI プロセスによって 20%異なっていることに着目した. PerfExplorer を用い, これらを性能に基づいて 2 つのクラスタに分離したところ, 計算時間は短いが `MPI_Wait` の時間は長いものと, 計算時間は長いが `MPI_Wait` の時間は短いものとに分離された. これに加え, さらにその MPI プロセスの番号 `rank` と, 計算ノードのメタデータを加えて解析したところ, この 2 つのクラスタはそれぞれ Cray XT3 と Cray XT4 のノード上で動作していたことが明らかになった. XT3 ノードは計算力が低く, CPU インテンシブな計算に大きな時間がかかり, 先に計算が終わってしまった XT4 が待たされた結果, `MPI_Wait` の時間が延びてしまうという結果になったことがわかる. 最終的に著者らはこの計算を XT4 ノードのみで行うように設定し, 12% の性能向上を達成した.

ここで示された例はある 1 回の実行の結果を用い, その中で 2 つの性能を比較するものであった. 本研究ではこのような比較ではなく, 2 つの異なる実行条件での性能を解析することを目指す.

2.2.3 Periscope

Periscope[18, 38] は並列アプリケーションの性能解析をする上で, アプリケーションの実行, プロファイリング, データ解析を全て統合するよう実装された Eclipse プラグインである. Periscope 自身は現在アプリケーションの性能を比較する機能は実装されておらず, 1 回の実行について詳しく性能を解析する機能が実装されている. しかし, 文献 [38] では将来的に性能を比較する機能が必要であり, 実装する予定であることが述べられている.

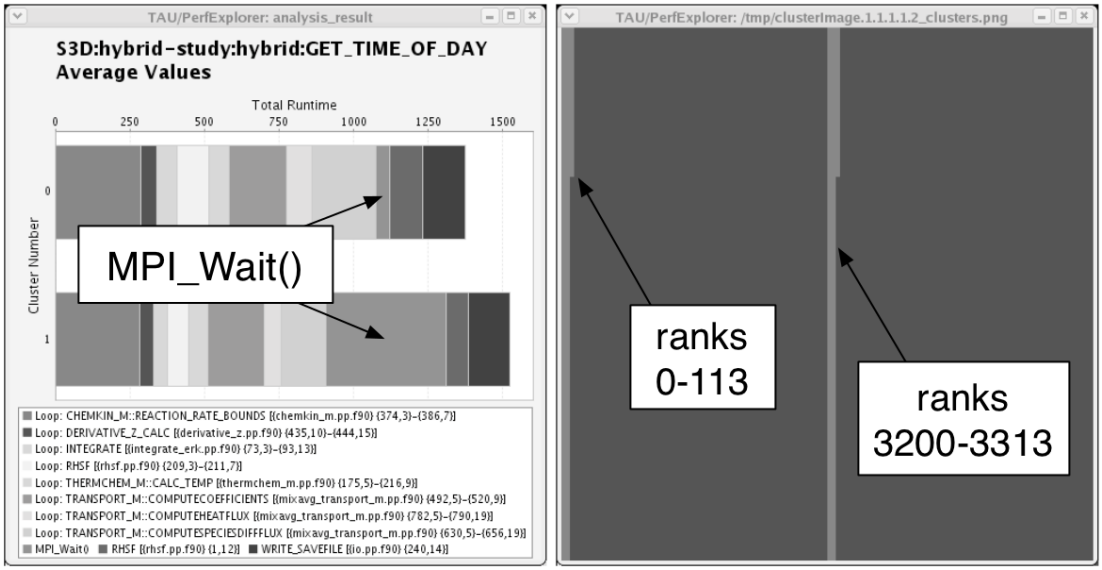

図 2.2. PerfExplorer による S3D アプリケーション解析結果 ([26] より)

第3章　対象アプリケーションと用語の定義および要素技術

本システムで対象とするアプリケーションの性質の定義および, 用語の定義を述べる. また, 本研究で用いる既存の要素技術について述べる.

3.1　MPI アプリケーション

Message Passing Interface[17] (MPI) は大規模な並列アプリケーションを書く上で現在では事実上の標準となっていて, 並列アプリケーションにおけるアセンブリ言語[1]とまで言われることもある [21, 11] メッセージパッシングのインタフェース規格である. MPI では 1 つのプログラムを基本的に同じパラメタで起動させる. もちろんこれでは全く同じ処理をするプログラムが並列に動作するのみで意味がないため, 並列ユニットの数を N としたとき, 各並列ユニットごとに独立な整数である $\texttt{rank} \in \{0, \ldots, N-1\}$ を割り当てる. 各並列ユニットが rank によって適宜分岐したり, 内部の変数を切り替えることにより扱うデータを変え, 並列動作していく. このように 1 つのプログラムの中ではプログラマの意図によって好きなように各並列ユニットを動作させることができるが, おおむねどの時点をとっても同じ関数で, 同じ部分を実行していることから, よく SPMD スタイルと呼ばれている. MPI 規格では各並列ユニットが実際にどのように実装されるかまでは規定していないが, 多くの実装では各並列ユニットをプロセスとして実装している.

MPI で書くアプリケーションが全て SPMD スタイルで書かれたアプリケーションであることはないのだが, 多くのものが SPMD スタイルで書かれているため, MPI アプリケーションの性能解析のためのプロファイラも SPMD スタイルのアプリケーションを対象にしているものがほとんどである. 本研究では必ずしも SPMD スタイルのアプリケーションに対象を限定しないが, 評価では SPMD スタイルのアプリケーションを用いる.

3.2　アプリケーション内の計量

現在, 多くのアプリケーションは 1 つのプログラム中でいくつかの手続きに分けて処理を記述する. これらは通常**関数** *(function)* と呼ばれる. この類のアプリケーションを対象とするプロファイラでは, 性能を計測する際にこの関数という単位で扱うことが多い. 一方, SCALEA[49, 16], Periscope[18], HPCToolkit[45] などではさらに細粒度に, ある関数中の特定の**領域** *(region)* という単位まで絞り込んで計測することができる. 本研究では MPI アプリケーションのプロファイラとして TAU (Tuning and Analysis Utilities) を用い, 関数レベルでの性能計測を行った結果を用いる.

[1]MPI は言語ではなくインタフェースであるが, しばしば言語と同じ扱いをされる.

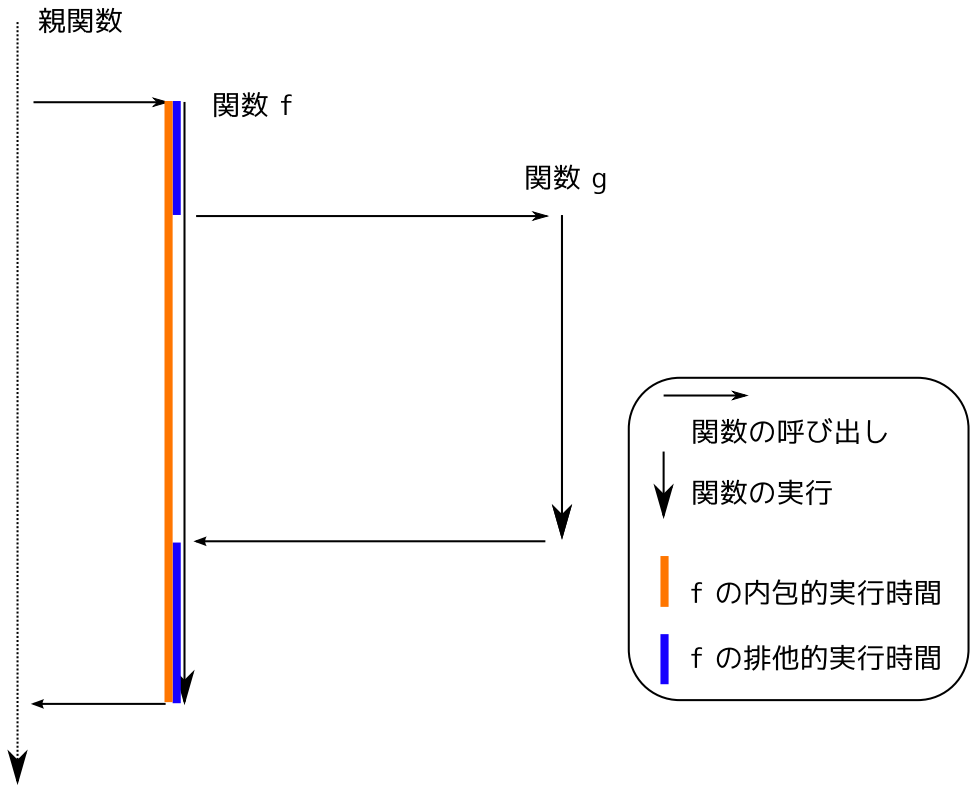

図 3.1. 内包的実行時間と排他的実行時間

関数レベルでの性能計測は関数の実行に要した時間を用いるが, これには 2 種類の時間が存在する. 関数 f が関数 g を呼ぶようなコードがあった場合の, 関数 f の実行時間について, 2 種類は以下のように定義される [19, 32].

内包的実行時間 (inclusive execution time) は関数 f が関数 g を呼んでいたとしても, 関数 f のフレームに入ってから関数 f のフレームを抜けるまで, 継続した時間を示す.

排他的実行時間 (exclusive execution time) は関数 f が関数 g を呼んでいた場合, 関数 g の実行時間を含めず, プログラムが関数 f の領域を実際に実行している時間のみを示す.

この状態を図 3.1 に示す.

TAU ではこの両方の情報を取得している. それぞれ解析したい内容によってどちらも使われるが, 本研究では後者の排他的実行時間をベースに解析を進める例を示す.

3.3　ワークフローアプリケーション

ワークフローアプリケーションはいくつかのアプリケーションの組み合わせにより全体として 1 つの動作をするアプリケーションであり, ここでは個々のアプリケーション間のデータのやりとりをファイルを用いて行うものとする. 入出力のファイルはそれぞれ複数でありうる. アプリケー

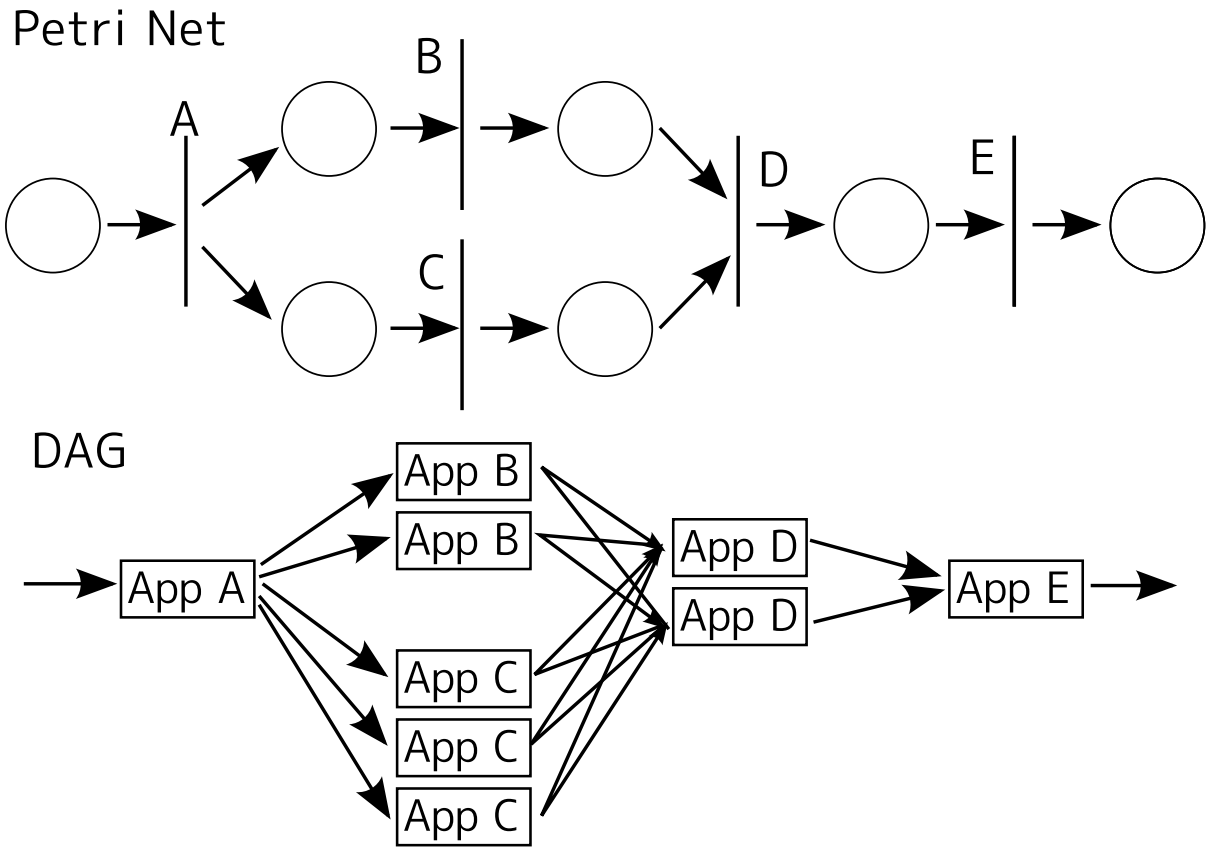

図 3.2. ペトリネットと DAG の対応

ション 1 つ 1 つの実行をジョブ（タスクとも呼ばれる）と呼ぶことにする. すなわち, あるアプリケーションが実行開始するのは必要な入力が全て揃った後である必要がある. したがって, ジョブ間には入出力ファイルに関する依存関係が存在するものがある. 互いに依存しないジョブも存在し, それらは資源があれば並列実行が可能である.

ワークフローアプリケーションは内部のアプリケーションをノードとし, 入出力ファイルの依存関係をアークとする DAG (Directed Acyclic Graph) として, あるいはペトリネット [31] によりモデル化することができる [41, 30]. すなわち, 入出力ファイルをトークンとし, システムの（特に入出力ファイルの）状態をプレース, そしてアプリケーションがトランジションとして記述される. この対応と, 依存関係に基づいた並列実行可能性を図 3.2 に簡略化した図として示す.

ワークフローは処理の流れであり, 記述されたペトリネットを駆動するものはワークフローエンジンと呼ばれる. ワークフロー中で行われる各ジョブの実行時間はジョブ毎に異なるため, ワークフローエンジンは計算資源とジョブを動的に結びつけ, スケジューリングを行っていくことで実行時間の短縮を図っている. 本研究ではワークフローエンジンとして GXP Make を用いる. これについて次節で述べる.

3.3.1 GXP Make

GXP Make[48] はワークフロー中のジョブ同士の依存関係を GNU Make の受理するスクリプト (`Makefile` として知られるもの) として記述し, それを並列に実行するためのワークフローエンジンの 1 つである. GXP Make では NFS や Lustre などによりファイルシステムがノード間で共有されていることを仮定し, GXP[47] の提供する遠隔シェルを利用してジョブをリモート実行する. GXP Make は多くの UNIX 環境で利用できる Python や GNU Make をベースに動作し, 展開コスト, 学習コストともに低い一方, 強力な記述力により複雑なワークフローも記述することができる. GXP Make は実行した各ジョブに関する性能情報を出力（現在は CSV 形式のテキストファイルのみを出力）しているため, 本研究の実装においてはこの情報を用いて性能解析を行うこととした.

3.4 ParaTrac

ParaTrac[14, 15] はファイルアクセスの性能を計測するためのシステムである. ParaTrac は FUSE[3] を利用し, ファイルアクセスのためのシステムコールを一度トラップし, ログを取りながら下位のファイルシステムへと引き渡す. ParaTrac のユーザは ParaTrac にマウントポイントを指定する. すると ParaTrac はそのマウントポイント以下に `/`（ルートファイルシステム）以下のディレクトリ構造を再現するので, ユーザが`/path/to/file` にアクセスしようとし, マウントポイントが`/foo/mountpoint` であれば `/foo/mountpoint/path/to/file` にアクセスすることで, ParaTrac の機能を利用してログを取得することができる. 逆に, ParaTrac を動作させつつもログ取得が不要であれば, マウントポイント`/foo/mountpoint` 以下にはアクセスしなければよい. ParaTrac はカーネルモジュールを読み込ませることで VFS からの命令を取り, ユーザレベルでは `libfuse` を用いることでユーザレベルの FUSE プロセスからの処理をトラップする. その後実際のファイルアクセスを本来のパスにより行い, 自身はそのときのログを出力する (図 3.3).

ParaTrac は終了時にまずテキスト形式のログを出力し, 後から SQLite3 のデータベースへと変換することができる. このデータベースは 4 つのテーブルからなっており, ParaTrac が実行された計算機の状況, ParaTrac の実行に関する情報, ParaTrac 経由でアクセスされたファイルの情報, そして ParaTrac がトラップしたアクセスログである. 最後のテーブルには `libfuse` から渡ってきた命令 1 つにつき 1 レコードが格納されている.

また, FUSE を使っているため, `read`, `write` 命令の要求サイズが大きいと分割される場合がある. これは FUSE のカーネルモジュール内で上限が設定されているためであり, FUSE version 2.8.7 現在では 128KB が上限となっている. これを越える命令は上限ごとに区切られ, `libfuse` 経由では別の命令として FUSE プロセスに渡る. どのように区切られたかは FUSE プロセス側では区別がつかないため, ParaTrac でもそのままとなっている. この最大サイズの変更は検討されている [2] が, カーネルモジュールの更新を伴うため, 広く行き渡るには時間がかかるものと思われる.

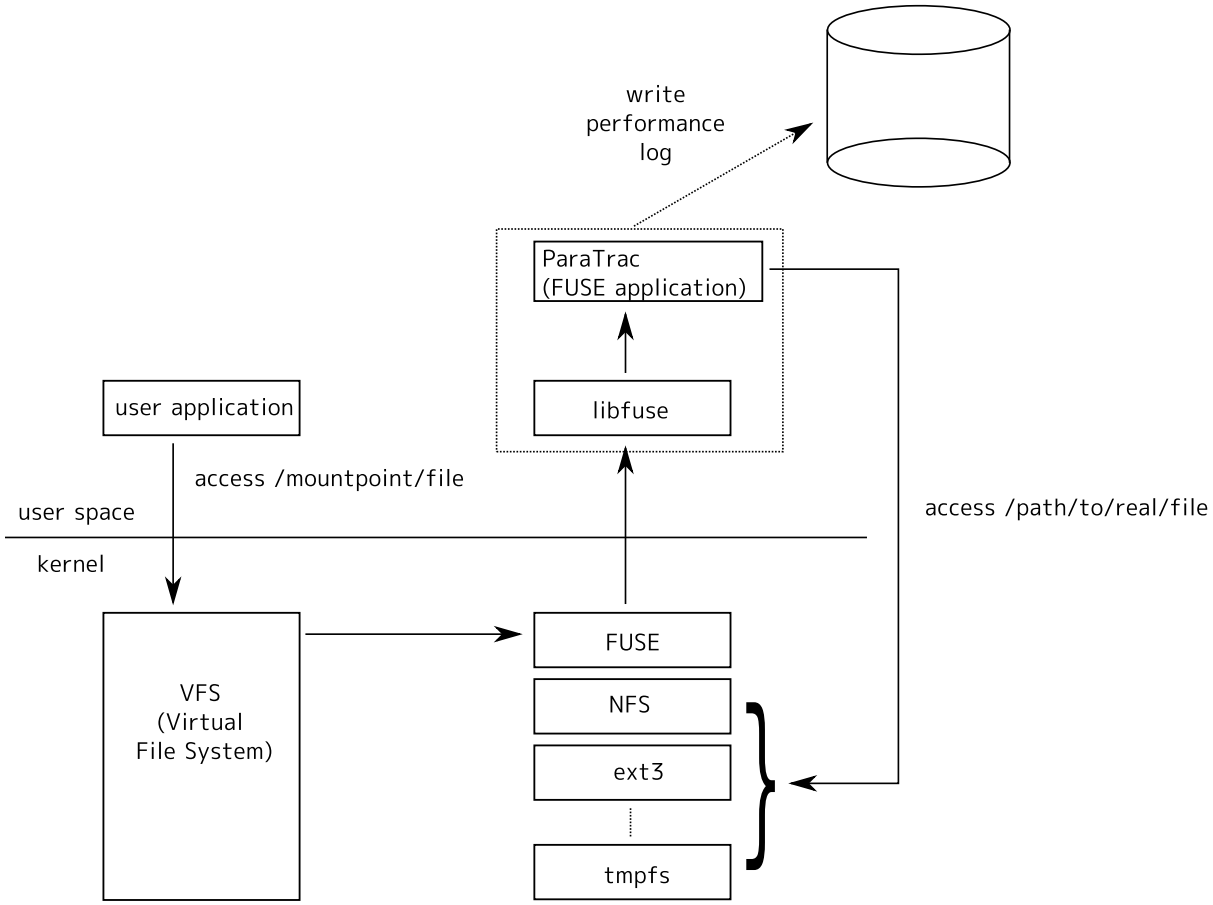

図 3.3. ParaTrac の概要

第4章　設計

4.1　設計概要

本システム PARP の設計について述べる. PARP の概要は図 4.1 に示すとおりである.

1. ユーザが MPI アプリケーションやワークフローアプリケーションを実行する. このとき, TAU や GXP Make を用いてプロファイル情報を取得する.
2. コマンドラインインタフェースを用いてプロファイル情報を PARP データベースに登録する.
3. ユーザは Web インタフェースを通し, データベースに格納されたアプリケーションの性能情報を解析し, 性能改善につなげる.

4.2　性能情報データベース

解析対象とするアプリケーションの性能情報を格納するのが性能情報データベースである. 現在, データベースは MPI 向けとワークフロー向けで別々の設計となっているが, 実際には MPI 向けをワークフロー向けに統合する形でまとめることができる.

4.2.1　MPI 向け性能情報データベース

それぞれのテーブルの役割, および保持する情報は以下のとおりである.

profgroup 複数の `profexec` を束ねる役割をもつ. ある条件での複数回のアプリケーション実行を表わす. アプリケーション名, 実行環境の名前, プロセス数などを保持する.

profexec ある条件での 1 回のアプリケーション実行を表わす. 他の実行と区別するため, 全体としての実行時間および開始時刻（タイムスタンプ）を保持する.

profile ある実行のうちでの各関数の挙動を表わす. ここでは TAU [43] の出力データに倣い, 1 つのレコードは 1 つの MPI プロセス (rank) の 1 つの関数に対応する. 呼び出し回数, 被呼び出し回数, 包括的所要時間および排他的所要時間を保持する.

4.2.2　ワークフロー向け性能情報データベース

ワークフロー向け性能情報データベースは MPI 向けのもので得られた反省点を踏まえ, 様々な実行条件に対応し, また今後の拡張にも支障がないよう, テーブルの構成方法を変更している. それぞれのテーブルの役割を以下に示す.

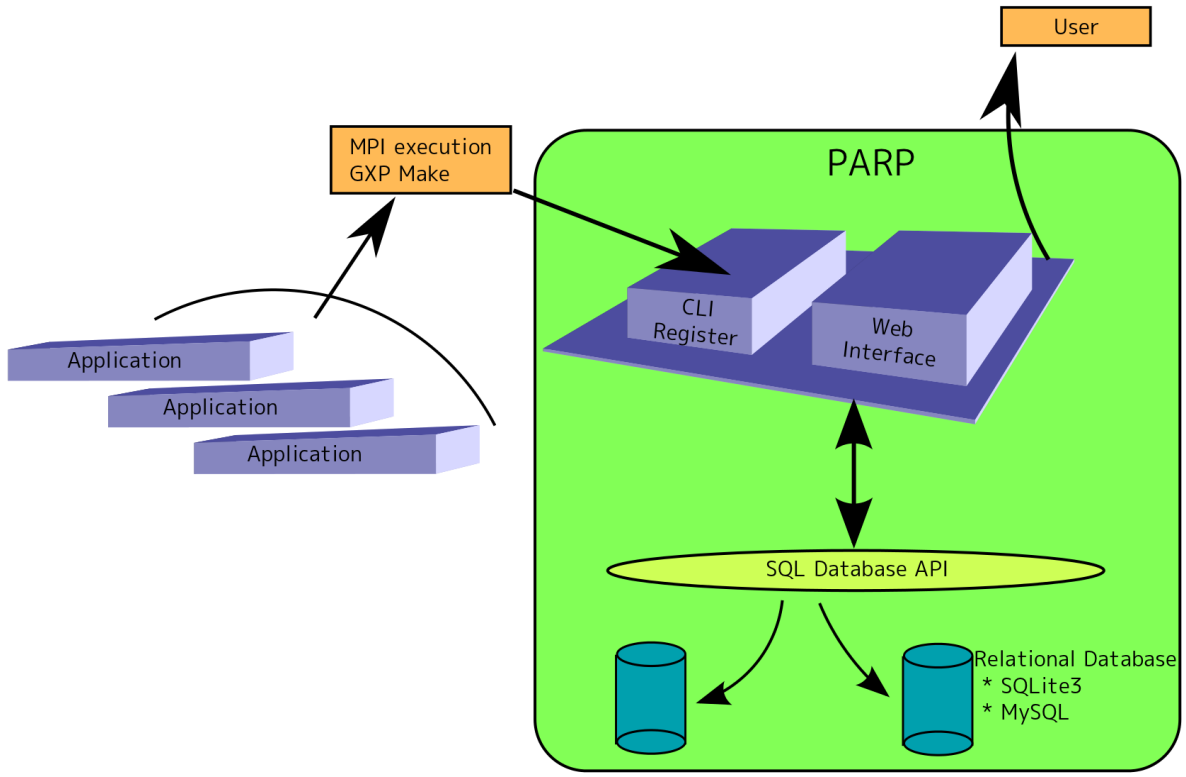

図 4.1. PARP 概要図

workflow ワークフローアプリケーション 1 つを認識するために用いる.

workflow_condition ワークフローアプリケーションの実行条件を示す. この ID が等しい `workflow_trial` は同じ実行条件とみなす.

workflow_trial ワークフローアプリケーションの 1 回の実行を表現する. ワークフローアプリケーションとワークフローの実行条件, そして実行開始時刻により 1 回の実行を識別する.

wf_worker ワークフローアプリケーションの実行時, 計算に参加したコンピュータ（計算ノード）の情報を格納する. 本来は `workflow_condition` に結びつけられるべきものだが, スーパコンピュータ環境では特定の計算ノードを指定することはできず, ジョブキューイングシステムにより自動的に計算ノードを指定されてしまうことが普通であるため, 同じ環境として使っていても実行のたびに計算ノードが変わってしまう. そこで, 本設計上は計算ノード情報をそれぞれの実行 (`workflow_trial`) に結びつける.

application ワークフローアプリケーションの内部に含まれるアプリケーション（コマンド）それぞれを示す. これはワークフローに結びつけられる.

job ワークフローアプリケーションを実行する際の各タスク/ジョブを示す. ここに各タスクの実行時間を格納するため, これは各実行 (`workflow_trial`) に結びつける.

metric 各タスクの中で, さらに細かい情報を格納するために用いる. 本研究では用いておらず, 今

後の拡張のために用意してあるのみとなっている.

paratrac_runtime ParaTrac を用いてファイルシステムに関するシステムコールの情報を取得する際, 各計算ノードについて 1 つの ParaTrac デーモンが起動し, そのノードでの情報を取得しているため, ワークフローアプリケーションを実行するときに N 台の計算ノードを使っていれば N 個の ParaTrac デーモンからログが別々に出力されることになる. 本テーブルは PARP でこれを 1 回の実行に結びつける役割を果たす.

paratrac_file ParaTrac 経由でアクセスされたファイルの一覧を管理する.

paratrac_syscall ParaTrac 経由でのファイルシステムへのアクセス全てを, 1 命令 1 レコードで記録する.

paratrac_syscall_name ParaTrac 内では Linux のものとは異なるシステムコール番号, 命令セットを用いている. したがって, このテーブルと結合することにより, ParaTrac のシステムコール番号と命令名を結びつける.

pt_sc_mt_summary ParaTrac から取得した情報のサマリを格納する.

以上の設計を行うことで, 以下の恩恵を受けることができる.

- 実行環境ごとに情報をひとまとめにすることができる.
- ある実行環境で複数回実行したとき, それらの平均をとることでノイズを除去することができる.
- 並列実行を行っている各プロセスの挙動をみることもでき, それをあまり気にせず平均化して解析することもできる.
- 通常であればワークフローアプリケーションの実行とは完全に別のデータとして扱わなければならない, ファイルシステムに関する性能情報も解析することができる.

4.3 コマンドラインインタフェース

プロファイラを通して得られた性能情報は PARP 実装に含まれるコマンドラインツールを通してデータベースへと登録する. MPI アプリケーションの性能情報を格納するとき, TAU は出力するプロファイル情報に多くのメタデータを埋め込んでいるため, そこから自動的に情報を取り出し, 格納することができる. そこに含まれていない情報や, ユーザから与えたい情報については, コマンドラインオプションにより指定することができる. ワークフローアプリケーションの場合, GXP Make から直接は多くのメタデータを得ることができなかった. そこで, GXP Make へのパッチを作成し, 一部のメタデータを出力するようにすることで, こちらも MPI アプリケーション向けのものとほぼ同じようにデータベースへの登録作業を自動化することができる.

4.4 Web インタフェース

既存の性能解析ツールのインタフェースはコマンドラインベースのものが主流であり, 大量の情報を解析するようになってからリッチな情報提供を行うようグラフィカルなインタフェースを備

えるものが増えている. 本研究でもグラフィカルなインタフェースをユーザに提供することにより, 大量のデータを相手にする場合でも無理なく性能情報の解析が行えるように設計する.

Web インタフェースではデータベースの情報をある程度まとめ, どれを解析対象にするかを選びやすいようにする. 対象となるアプリケーションを選択すると, そのアプリケーションが実行された様々な実行環境あるいは実行条件が一覧表示される. そのとき, それぞれの実行条件では何回か実行されている可能性があるが, 平均値を計算し, 自動的に 1 つだけ表示するようにする. これにより, ユーザが一度に目にしなければならない情報を減らしつつも, 利便性は確保することができる. 本システムでは 2 つの実行条件での比較による性能解析を行うことを基本とする. 2 つの実行条件をユーザが選択し, どのようにグラフを作成したいかを指定することで, パラメタを可視化したグラフを作成し, ユーザに提示することができる. ユーザはその結果を用いて解析を行うが, グラフを 1 回作るだけでは有効な解析が行えるとは考えにくい. 本システムではユーザ側からの指示により, 何度でも自由にグラフの内容を変更し, ユーザに提示することで, 有効な性能解析を狙う.

本システムでは 2 つの実行条件での比較による性能解析を行うことを基本とすることを先に述べたが, ワークフローアプリケーションの解析では一部比較ではなく, 1 回の実行を対象に解析の手助けをできるような仕組みを導入している. これの詳細は次章で述べる.

第5章　実装

PARP の性能情報データベースと Web インタフェースそれぞれについて, 実装を述べる.

5.1　性能情報データベース

MPI 向け, ワークフロー向けそれぞれに性能情報データベースを実装している. 将来的には MPI 向けのものをワークフロー向けのものに統合することができる.

5.1.1　MPI 向け性能情報データベース

MPI 向け性能情報データベースには SQLite3 を用いている. このテーブル定義 SQL をリスト A.1 に, E-R 図を図 5.1 に示す.

各テーブル内の主要カラムについて, 意味するところを述べる.

TABLE profgroup

application TEXT MPI のアプリケーション名を保持する. これはアプリケーションのファイルシステム上のパス名である.

例: `/home/user/bin/mpi/pagerank.mpi`

app_viewname TEXT MPI のアプリケーション名だが, ユーザに Web インタフェースを通じて示すための名前である.

例: `pagerank`

nodes INTEGER 計算ノードの数を示す.

procs INTEGER MPI プロセスの数を示す.

place TEXT 実行された場所を示す. これは Web インタフェースを通じてユーザに提示されるもので, ユーザが自由に決めて構わない. 指定しなかった場合, 計算ノードの名前から自動的に検出され, 設定される.

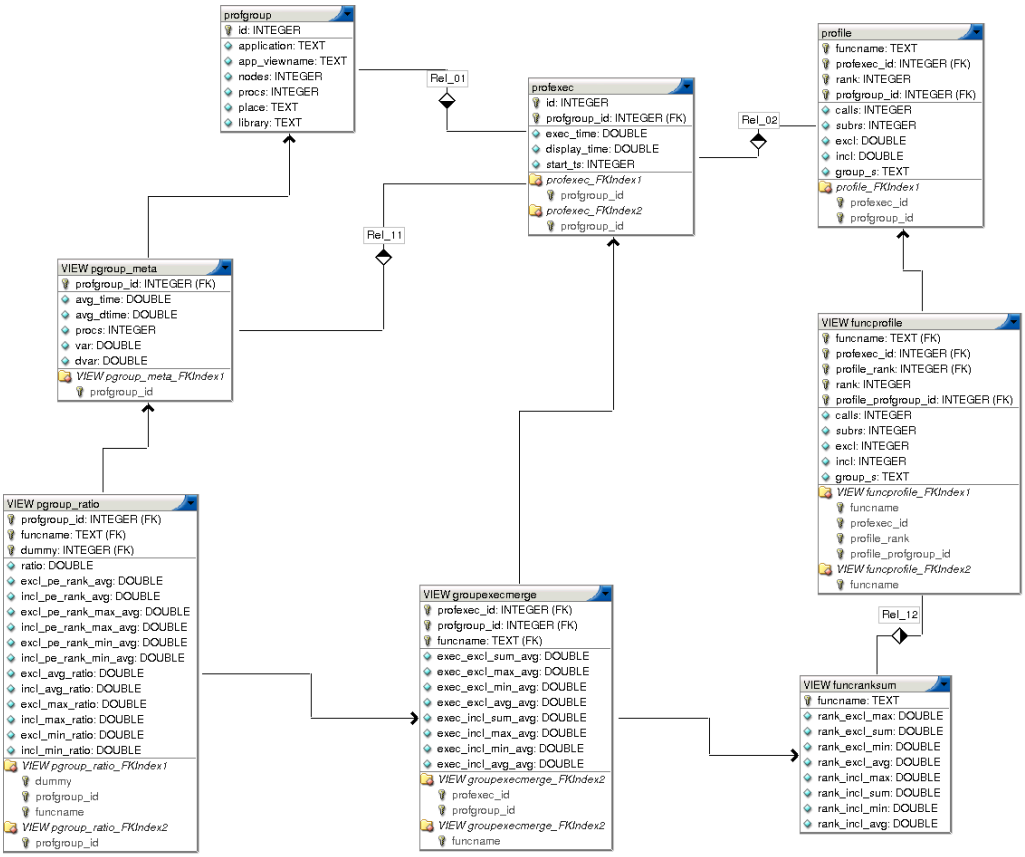

図 5.1. MPI 向け性能情報データベースの E-R 図

library TEXT　利用したライブラリの名前を示す. `place` で指定した場所が同じであっても, その中で実行条件が異なる場合にこの欄で区別することができる. 利用した MPI ライブラリの名前であったり, データファイルを配置したファイルシステムの名前を指定したりすることを意図した, いわば意味をもったコメント欄である.

TABLE profexec

exec_time DOUBLE　MPI アプリケーションの全体としての実行時間を秒単位で示す.

display_time DOUBLE　MPI アプリケーションの中で, 評価に関係ある時間と, 性能評価に用いない無関係な関数がある場合がある. そこで, PARP では `display_time` を Web インタフェースを通じてユーザに示すことで, ユーザ視点からの性能評価をしやすくしている.

start_ts INTEGER　アプリケーションの実行開始時刻を UNIX Time で格納している. これは全く同じ条件下での複数回の実行を区別するために用いる.

TABLE profile

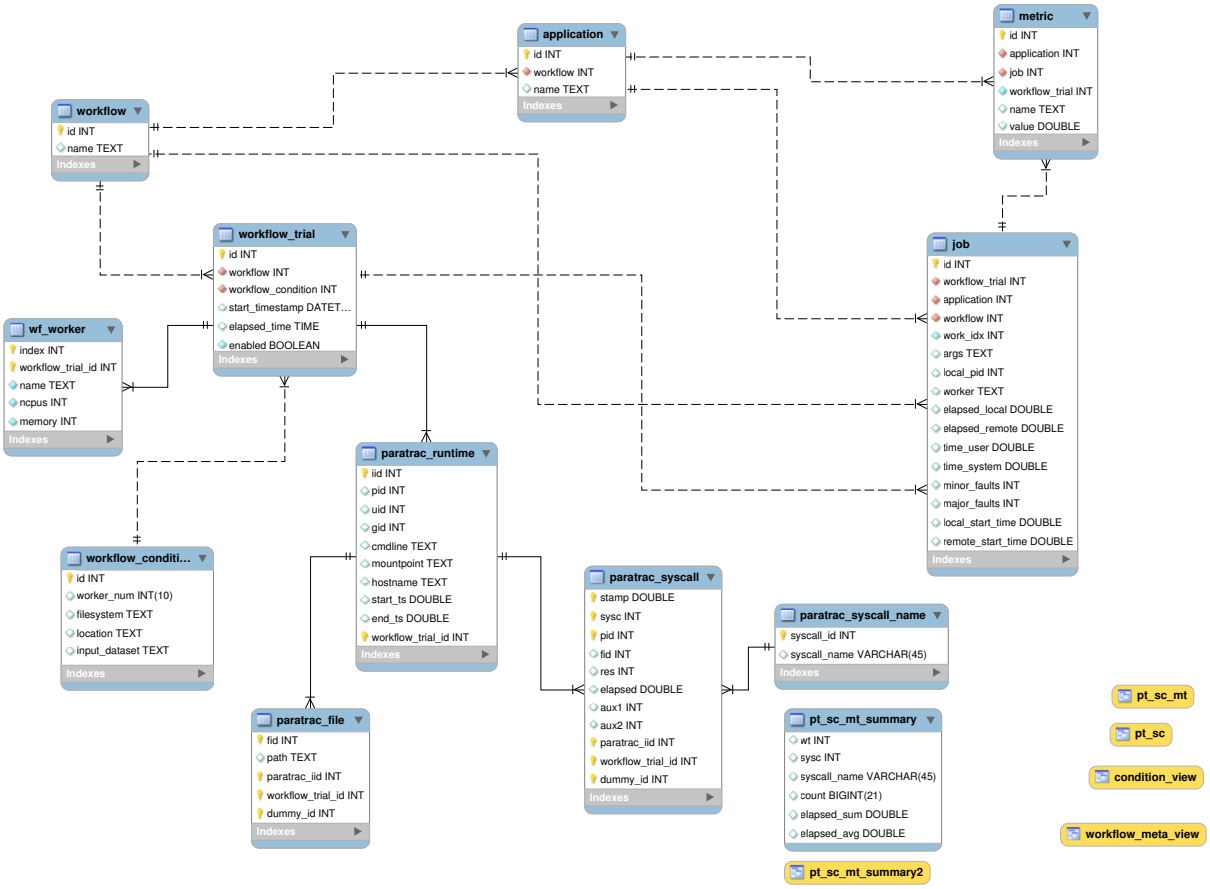

図 5.2. ワークフロー向け性能情報データベースの E-R 図

funcname TEXT　関数名を示す.

calls INTEGER　この関数が**呼ばれた回数**を示す.

subrs INTEGER　この関数からの**関数呼び出し回数**を示す.

excl DOUBLE　この関数の排他的実行時間を示す.

incl DOUBLE　この関数の内包的実行時間を示す.

ここで, 各カラムは各 rank 内での合計値となっている.

5.1.2　ワークフロー向け性能情報データベース

ワークフロー向け性能情報データベースには MySQL を用いている. テーブル定義 SQL をリスト A.2 に, その E-R 図を図 5.2 に示す.

各テーブル内の主要カラムについて, 意味するところを述べる.

TABLE wf_worker

name TEXT 計算ノード名を示す. これは各計算ノードのホスト名である.

ncpus INTEGER この計算ノードに搭載された CPU 数を示す. これは GXP Make が自動的に検出するか, ユーザが明示的に指定したものとなる.

memory INTEGER この計算ノードで利用可能なメモリサイズを示す.

TABLE workflow_condition

worker_num INTEGER ワーカ（計算ノード）の数を示す.

filesystem TEXT データを配置したファイルシステム名を示す.

location TEXT アプリケーションを実行した場所を示す.

input_dataset TEXT 入力データセットの名前を示す.

TABLE workflow_trial

start_timestamp DATETIME ワークフローアプリケーションの実行開始時刻を示す.

elapsed_time TIME このワークフロー実行の継続時間を示す. 実行時間は秒単位で記録され, 本来浮動小数値であるが, MySQL の実装の問題により, ここでは整数値となっている.

enabled BOOLEAN 解析にこの実行のものを用いるかどうかを示すフラグである.

TABLE job　このテーブルの内容は GXP Make から得られたものをそのまま利用しているにすぎないため, ここで詳細は述べない.

5.2　GXP Make の拡張

GXP Make は単独で各タスクに関する性能情報を出力するが, それ以外にも GXP Make が内部的には持っており, 解析で必要な情報が存在する. そこで GXP Make を拡張し, ワーカのリストおよびワークフロー実行開始時刻, ワークフロー全体の実行時間 (Makespan[34]) を別途データベースとして出力している.

5.3　コマンドラインインタフェース

MPI アプリケーションからの情報は TAU から, またワークフローアプリケーションからの情報は GXP Make (前節の拡張を施したもの) から得ることができる. これらは数多くのファイルとしてファイルシステム上に保管されているが, これをデータベースに登録するインタフェースを実装した. これは MPI 向け, ワークフロー向けで別々に実装されている. 5.5 節で述べる ParaTrac からの情報のインポートも, このインタフェースのオプションとして行うことができる.

これらの実装は別々に実装されているものの, 基本的な構造は同じである. 異なるデータベースを扱っているため接続部分が異なったり, 性能情報の読み取り方法などは異なっているが, 適切に

テーブルへレコードを追加すべきか否かを判断し, 必要なレコードをインポートしていくという点では変わらない.

ユーザから与えられたオプションおよびプロファイラからの性能情報から, 実行条件が同じものがあれば, それらを 1 つのグループとして扱えるよう, MPI 向けであれば `profgroup` の ID を, ワークフロー向けであれば `workflow_condition` の ID を, 既存テーブルから取り出して用いることで実現している.

5.4　Web インタフェース

Web インタフェースの実装について述べる. Web インタフェースは MPI 向け, ワークフロー向けいずれも 1 つの Django アプリケーションとして実装されているが, アクセスするパスによって完全に異なる動作をしているため, 本節でも別々に実装について記述する. アクセスパスに基づき処理を切り替える動作は Django フレームワークを利用し, リスト 5.1 のようになっている（抜粋）.

リスト 5.1. MPI 向け性能情報データベースのテーブル定義 SQL リスト

```
urlpatterns = patterns(
    '',
    # MPI
    (r'^pgd2/(?P<sortmode>\w+)/$', 'parpview.viewer.views.pgd2'),
    (r'^pgview/(?P<pg_id>\d+)/$', 'parpview.viewer.views.pgview'),
    # Workflow
    (r'^wf/?$', 'parpview.viewer.views.wf'),
    (r'^wfinfo/(?P<wf>\d+)', 'parpview.viewer.views.workflow_info'),
    (r'^wfcinfo/(?P<wf>\d+)/(?P<wfc>\d+)',
     'parpview.viewer.views.workflow_condinfo'),
    (r'^wfdiff/(?P<wf>\d+)/(?P<wfc1>\d+)/(?P<wfc2>\d+)',
     'parpview.viewer.views.wfdiff'),
    (r'^wft-tcw/(?P<trial_id>\d+)',
     'parpview.viewer.views.wf_timechart_img'),
    (r'^wft-tc/(?P<trial_id>\d+)',
     'parpview.viewer.views.get_wf_timechart'),
    # Static / Image / Resources
    (r'^img/(?P<imgpath>.+\.png)$', 'parpview.viewer.views.getpng'),
    (r'^style/(?P<stylefile>[^/]+)$', 'parpview.viewer.views.getstyle'),
    (r'^script/(?P<path>.+\.js)$', 'parpview.viewer.views.getjs'),
    (r'^resources/(?P<path>.*)$',
     'django.views.static.serve',
     {'document_root': settings.MEDIA_ROOT}),
)
```

5.4.1　memcached の利用

本システムではユーザからのクエリを受け, データベースへ問い合わせを行い, その結果に基づいて解析を行う. しかし, 同じクエリであってもグラフの軸設定のために何度も実行結果を用いたくなる場合がある. 扱うデータが大きいため, クエリの実行にもある程度時間を要してしまい, 何もしなければ解析に多くの時間が費されてしまう.

そこで, 本システムでは同じクエリの結果を必要とするページロードが多いことに注目し, memcached によるキャッシュを行う. 各クエリは 1 回目にはクエリの情報と, 直列化した結果を memcached に格納し, 2 回目以降必要とされた場合はそのキャッシュを取り出して非直列化して用いる. これはシンプルな方法であるが, ユーザの利便性のためには重要な手段の 1 つである.

5.4.2　MPI 向け Web インタフェース

MPI 向け Web インタフェースは初期的な実装であり, 最小限ユーザが MPI アプリケーションの性能を比較できるよう, 単純な機能のみを備えている.

比較インタフェース

比較インタフェース (リスト 5.1 での `pgd2`) では, まずデータベースから `profgroup` テーブルの内容を抽出し, ユーザに提示する. ただし, `profgroup` テーブルの中身だけでは実行条件しか示しておらず, ユーザはそれをさらに深く確認しなければ実行に要した時間がわからず, 不便である. そこで, `profgroup` テーブルの中身だけでなく, それぞれの条件で, それぞれの実行の実行時間を保持している `profexec` テーブルと結合したビュー `pgroup_meta` を追加した. これらの情報を用いることで, 図 5.3 のような表を作成し, ユーザはこの中から 2 つの実行条件を選択し, 性能を比較することができる. ここでは同じ表に複数のアプリケーションが混在して表示されているが, PARP では関数名をベースに性能を比較する実装を提供しているため, 中身が異なっている異なるアプリケーション同士を比較するよう指定しても, 意味のある結果を得ることはできない. したがって, ユーザが異なるアプリケーションの性能同士を比較しようとした場合にはエラー画面を表示している.

比較は「どの値をグラフに表示するか」をユーザが決定し, 設定する. これを Web フォームに入力することで実現する. 実際の例は 6.2 節で示す. フォームの例は図 6.3 に示されている. この表の各カラムの意味するところを以下に示す.

Use? カラムはユーザが自由に選択することができ, 追加と削除ができなければならない. このチェックボックスを外すことでカラムを削除することができる.

Column Name カラムの名前をつける. これは SQL の文中でカラムの名前となる.

Column definition SQL カラムの中身になる SQL を記述する.

ORDER BY 指定されたカラムの中でどれを基準にソートを行うかを, このラジオボタンを用いて指定する.

Graph Style グラフ表示の際, どの形式で表示するかをカラムごとに指定する. 本インタフェース (MPI) ではグラフ作成に GNUPLOT を利用しているため, GNUPLOT で利用可能なグラフのスタイルがリストされており, その中からユーザが 1 つを選ぶ.

`Column Name` と `Column definition SQL` について説明を加えておく. これらはデータベースから実際の性能情報を取り出すための SQL を実際に構築する際, また, グラフを生成した際のラベルにも用いられる. これらを SQL の生成に用いる基本的なアルゴリズムをリスト 5.4.1 に示す.

Select Comparison Targets

PG 1	PG 2	Application	Comment	Place	# of nodes	# of Processes	Avg. Time [sec]	StdDev. [sec]
○	○	FileAccess	MPICH2 MX GCC hsfs Detail	ha8000	4	16	1274.739	0.000
○	○	FileAccess	MPICH2 MX GCC lustre Detail	ha8000	4	16	151.820	0.000
○	○	FileAccess	MPICH2 MX GCC nfs Detail	ha8000	4	16	33.085	0.000
○	◉	PageRank	MPICH2 GCC NFS Detail	hosei	16	128	10.012	5.566
◉	○	PageRank	OpenMPI GCC NFS Detail	hosei	16	128	64.616	11.744
○	○	PageRank	OpenMPI-t GCC NFS Detail	hosei	16	128	8.256	0.622
○	○	PageRank	MPICH2 GCC NFS TRACE Detail	kyutech	16	128	48.514	3.122
○	○	PageRank	OpenMPI GCC NFS TRACE Detail	kyutech	16	128	108.440	9.095
○	○	PageRank	MPICH2 GCC NFS Detail	kyutech	16	128	49.846	7.046
○	○	PageRank	OpenMPI GCC NFS Detail	kyutech	16	128	107.999	9.589
○	○	Solver	MPICH2 GCC NFS Detail	cloko	8	128	85.289	5.455
○	○	Solver	MPICH2 GCC NFS Detail	cloko	8	64	145.565	3.790
○	○	Solver	MPICH2 GCC NFS Detail	cloko	16	128	68.264	1.503
○	○	Solver	MPICH2 MX GCC HSFS Detail	ha8000	4	64	129.667	12.335
○	○	Solver	MPICH2 MX GCC NFS Detail	ha8000	4	64	122.233	5.959
○	○	Solver	MPICH2 MX GCC LUSTRE Detail	ha8000	4	64	124.194	6.111
○	○	Solver	OPENMPI MX GCC HSFS Detail	ha8000	4	64	127.759	7.772
○	○	Solver	OPENMPI MX GCC NFS Detail	ha8000	4	64	126.329	7.067
○	○	Solver	OPENMPI MX GCC LUSTRE Detail	ha8000	4	64	125.803	10.459
○	○	Solver	MPICH2 MX GCC HSFS Detail	ha8000	8	128	82.352	3.909
○	○	Solver	MPICH2 MX GCC NFS Detail	ha8000	8	128	83.379	3.291
○	○	Solver	MPICH2 MX GCC LUSTRE Detail	ha8000	8	128	79.956	2.571
○	○	Solver	OPENMPI MX GCC HSFS Detail	ha8000	8	128	1756.912	2719.423
○	○	Solver	OPENMPI MX GCC NFS Detail	ha8000	8	128	277.167	126.822
○	○	Solver	OPENMPI MX GCC LUSTRE Detail	ha8000	8	128	226.932	39.046
○	○	Solver	MPICH2 GCC NFS Detail	kyutech	16	128	123.603	5.459
○	○	Solver	OpenMPI GCC NFS Detail	kyutech	16	128	223.366	15.316
○	○	Solver	MPICH2 GCC NFS Detail	kyutech	8	64	187.856	1.035

Create Graph!

図 5.3. PARP Web インタフェースにおけるプロファイル選択 (MPI 向け)

リスト 5.1 の pgd2 では sortmode では関数のソートを行う際, ORDER BY で指定したカラムの昇順でソートするか, 降順でソートするかを選択できる. 実行時間をはじめとして, 多くの指標が「大きいほど意味のある」ものであることから, PARP ではデフォルトで降順ソートを行う.

実行条件のサマリ

比較インタフェースの表で示すことができる情報は非常に圧縮されており, 限られてしまっている. しかし, ユーザの視点からみればある実行条件についてしっかり見ておくことも重要な場合があると考えられるため, 実行条件を選択したときにデータベースの情報からある程度詳しい情報をユーザに提示する実装を行った. 比較インタフェースでは複数回実行のうち, 実行時間の平均値と標準偏差のみを示していたが, 実際に 1 つ 1 つの実行にどの程度時間がかかったのかを表示している. 画面の例を図 5.4 に示す.

Algorithm 5.4.1 Column Name および Column definition SQL のリストから SQL の生成

defs ← pairs of (column name, column definition SQL)
$S \leftarrow \emptyset$
for all pair in defs **do**
  $target \leftarrow pair[0] + ' \; ' + pair[1]$
  $S \leftarrow S + \{target\}$
end for
select_targets_string ← join(',', S)

5.4.3　ワークフロー向け Web インタフェース

ワークフロー向け Web インタフェースの実装について述べる. 前節で述べた MPI 向け実装では比較対象を選ぶ画面に全てのアプリケーション, 全ての実行条件を提示していたが, ワークフロー向けの実装ではまず初めの画面 (リスト 5.1 における `wf`) で解析するアプリケーションがリストアップされ, ユーザはこれを選択する (図 5.5).

ワークフローアプリケーションのサマリ

ユーザがワークフローアプリケーションを選択すると, 指定されたワークフローのサマリページが表示される. ここではワークフローに含まれるコマンドの一覧や, データベースに登録されている実行条件の一覧が表示されている. 図 5.2 のデータベース実装に基づき, コマンドの一覧は リスト 5.2 により取得することができる. ここで "?" はプレースホルダであり, 指定されているワークフローの ID が入る.

リスト 5.2. ワークフローに含まれるアプリケーションの一覧を得る SQL

```
1  SELECT id, name FROM application
2  WHERE workflow=?
```

ワークフローの実行条件および平均値などのサマリ情報を得るための SQL 文をリスト 5.3 に示す.

リスト 5.3. ワークフローの実行条件とサマリを得る SQL

```
1  SELECT
2    wft.id id,
3    wf.name name,
4    wft.workflow_condition wf_condition,
5    wfc.id wfc_id,
6    wfc.location location,
7    wfc.filesystem filesystem,
8    wfc.worker_num worker_num,
9    wfc.input_dataset input_dataset,
10   SUM(wft.enabled) num_enabled,
11   COUNT(*) num_trials,
12   AVG(TIME_TO_SEC(wft.elapsed_time)) elapsed
13 FROM
14   workflow_trial AS wft,
```

```
15    workflow AS wf,
16    workflow_condition AS wfc
17 WHERE
18    wf.id = wft.workflow AND
19    wfc.id = wft.workflow_condition AND
20    wf.id = ?
21 GROUP BY wft.workflow_condition;
```

また, アプリケーションの実行時間を

$$\text{実行時間} = \text{I/O 時間} + \text{CPU 時間} \tag{5.1}$$

と仮定したときの, 実行時間と CPU 時間がわかっていることから, I/O 時間を推定している. このための SQL をリスト 5.4 に示す. この仮定は必ずしも正しくないが, ワークフローアプリケーションではタスクという単位で解析を行っており, 粗粒度なデータであるため, ある程度役に立つ仮定であると考えられる.

リスト 5.4. ワークフローの実行条件における IO 時間を推定する SQL

```
 1 SELECT
 2   cv.workflow workflow,
 3   cv.wf_condition wf_condition,
 4   SUM(cv.app_count * cv.elapsed_local) elapsed_local,
 5   SUM(cv.app_count * cv.time_user) time_user,
 6   SUM(cv.app_count * (cv.elapsed_local - cv.time_user)) est_io
 7 FROM
 8    condition_view cv
 9 WHERE
10    cv.workflow = ?
11 GROUP BY
12    cv.workflow, cv.wf_condition;
```

実行条件のサマリ

MPI アプリケーション向けのものと同じく, ワークフロー向けの Web インタフェースでも実行条件ごとにサマリ画面を用意している. 例を図 5.6 に示す. それぞれの実行に要した時間が表示されている点は MPI アプリケーション向けのものと同様である. `TimeChart Link` については次節で述べる. 最右に `Enabled` というチェックボックスがある. これにより, データベース登録後にこの実行の不具合があり, 解析対象にしたくない場合など, 解析する対象から外す処理を行うことができる.

ワークフロー実行の可視化

ワークフローアプリケーションはタスクの実行の様子によっても性能が大きく変化しうる. したがって, 1 回のワークフロー実行を取り出して, その様子を調べる機構も重要といえる. たとえば, ワークフローアプリケーション自体はうまく書かれているはずなのに, それを実行するワークフロードライバの実装がよいものでなく, 実行できるタスクが即座に実行されなかったり, データローカリティの悪い場所にタスクを割り当ててしまって期待した性能が出ない場合も考えられる.

PARP ではそれぞれのワークフロー実行について, GXP Make から得た全タスクの実行開始時刻, およびその継続時間をデータベースに格納している. この情報を用いて, ワークフロー実行の様子を表現する図を生成する機能を実装した. これは上記の情報をもとに PostScript を記述し, これを Web ブラウザで表示可能な形式 (ここでは PNG 形式) に変換することで実現している. この例を図 5.7 に示す. 縦軸に計算ノードが, 横軸に時間が示してあり, アプリケーションごとに色が分かれて, 実行しているものが四角で示される.

比較

ワークフロー向け PARP の比較インタフェースについて述べる. MPI 向けのものを先に実装したため, そこでの反省点を踏まえた実装となっている.

カラム定義のテーブルは MPI 向けのものとほぼ同じなので省略する.

大きく異なるのは比較結果を示すグラフの作成方法である. MPI 向けの実装では GNUPLOT を用いていたが, 将来的に出てくるであろう様々な要求に対し, GNUPLOT で実現できることは限られていると考えられる. そこで, Web インタフェースと親和性が高い JavaScript でグラフ作成を行うライブラリ highcharts.js [4] を用い, そこにグラフ作成のためのデータを与えることでグラフを作成している.

ワークフロー向けの Web インタフェースでは, ユーザが完全に自由な SQL 文を書き, 自由にグラフを作成できるようなインタフェースを提供している. デフォルトではリスト 5.5 に示す SQL 文による解析を行い, その結果を示しているが, これを自由に変更し, グラフを作成することができる. このデフォルトのグラフからは 13 個の数値カラムが得られるが, 全てを表示するとグラフが非常に詰まってしまい, 性能解析には使いづらくなってしまう. そのため, デフォルト動作として, `Accum1, Accum2, IO1, IO2` の 4 カラムのみを最初に表示するよう設定している. highcharts.js の機能により, 判例の該当カラム名をクリックすることで, グラフへのカラムの表示と非表示を切り替えることができ, インタラクティブな性能解析が可能となっている.

リスト 5.5. ワークフローアプリケーション向けデフォルト解析 SQL

```
1  SELECT
2    app.name ApplicationName,
3    cv1.elapsed_local ElapsedL1,
4    cv2.elapsed_local ElapsedL2,
5    cv1.app_count AppCount1,
6    cv2.app_count AppCount2,
7    cv1.elapsed_remote ElapsedR1,
8    cv2.elapsed_remote ElapsedR2,
9    cv1.elapsed_local * cv1.app_count AccumL1,
10   cv2.elapsed_local * cv2.app_count AccumL2,
11   cv1.time_user / cv1.elapsed_local CPUEff1,
12   cv2.time_user / cv2.elapsed_local CPUEff2,
13   cv1.time_user UTime1,
14   cv2.time_user UTime2,
15   (cv1.elapsed_local - cv1.time_user) * cv1.app_count IO1,
16   (cv2.elapsed_local - cv2.time_user) * cv2.app_count IO2,
17   cv1.time_system STime1,
18   cv2.time_system STime2,
19   cv1.minor_faults MinFlt1,
20   cv2.minor_faults MinFlt2,
```

```
21    cv1.major_faults MajFlt1,
22    cv2.major_faults MajFlt2
23  FROM
24    condition_view cv1,
25    condition_view cv2,
26    application app
27  WHERE
28    cv1.application = cv2.application AND
29    cv1.workflow = cv2.workflow AND
30    cv1.wf_condition = ? AND
31    cv2.wf_condition = ? AND
32    cv1.application = app.id AND
33    cv1.workflow = ?
34  ORDER BY
35    AccumL2 DESC;
```

5.4.4　PARP 順序計量

MPI 向け PARP では 2 つの実行条件で, それぞれの関数について複数の値を表示するようなグラフを作成する. このとき, 関数は任意の個数存在しうるものであるが, 全ての関数をグラフに載せてユーザに全てを見せることは現実的でなく, 重要なものをいくつか絞り込み, ユーザに提示することが重要である. そこで, 本研究では性能差の原因に関して重要さを示す順序計量を提案する. この順序計量では以下を意図している:

- 重要さを示す計量といっても, その部分の改善により性能が改善されなければ意味が大きいとはいえない. そこで, 全体の実行時間に占める実行時間が長い関数ほど大きくなるような計量とする.
- 実行時間の差が大きいものほど, 改善によって得られる恩恵が大きいことが期待できる. そこで, 実行時間の差が大きいものほど大きくなるような計量とする.

これらは**片方だけみてもそれほど意味のあるものではない**が, これらをうまく**組み合わせることで**, **性能差の解析に基づく性能改善に結びつきやすい**計量となると考えられる.

本研究では以下の式により関数 f の順序計量 m_f を与える.

$$m_f = \frac{t_f^a}{T_a} \times \log \frac{t_f^a}{t_f^b} \tag{5.2}$$

ここで, t_f^a は実行時間が長い方の関数 f の実行時間, t_f^b は実行時間が短い方の関数 f の実行時間, T_a は実行時間が長い方の全体としての実行時間である. 順序付けをする際, T_a は固定値となるため, 以下の式を用いれば十分である.

$$m'_f = t_f^a \times \log \frac{t_f^a}{t_f^b} \tag{5.3}$$

5.5　ParaTrac データの統合

ワークフロー向け PARP では基本的に GXP Make の情報をインポートして利用するように設計, 実装されているが, データベース設計で示しているように ParaTrac から取得した情報をインポートして利用することで, ファイルシステムの性能に関わる解析を行うこともできるようになっている. これは, 特にワークフローアプリケーションでは大きなデータを扱うことが多く, その際に下で動いているファイルシステムの性能が, ワークフロー全体の性能に対して非常に大きく関係してくる場合が多いと考えられたためである.

ParaTrac から得られた情報も先述のようにユーザが自由に SQL を記述する方法を利用することにより, 好きな情報を得ることができる. しかし, ワークフローアプリケーションの 1 回の実行中に呼ばれるファイルシステムへのアクセス関数の呼び出し回数は非常に大きい. このテーブルに直接アクセスして処理を行おうとすると, 数十秒から数時間まで解析に要してしまう. これは最初の 1 回で済ませるのならよいが, 解析を何度も行っていくことを考えると, 何度も大規模な SQL 処理を待つのは現実的でない. そこで, ParaTrac のデータをインポートしてから 1 回バッチ処理を行い, よく使われるであろうデータのアグリゲーションを行ったテーブルを作成し, それをユーザが利用できるようにする. これにより, ユーザは何度も解析を行っていくが, データベース処理によって待たされることを防ぐことができる.

これは `pt_sc_mt` ビューの定義と同様であり, 以下のようにして物理テーブル `pt_sc_mt_summary` を作成し, ユーザが使えるようにしている.

リスト 5.6. ParaTrac からの情報のサマリテーブル定義

```
1  CREATE VIEW `testparpdb`.`pt_sc` AS
2  SELECT *
3    FROM paratrac_syscall sc, paratrac_syscall_name scn
4    WHERE sc.sysc = scn.syscall_id;
5
6  CREATE VIEW `testparpdb`.`pt_sc_mt` AS
7  SELECT
8   wt.id wt,
9   p.sysc,
10  p.syscall_name,
11  COUNT(*) count,
12  SUM(p.elapsed) elapsed_sum,
13  AVG(p.elapsed) elapsed_avg
14   FROM pt_sc p, workflow_trial wt
15   WHERE p.workflow_trial_id = wt.id
16   AND wt.enabled = 1
17   GROUP BY wt.id, p.sysc
18 ;
19
20 CREATE TABLE pt_sc_mt_summary
21 SELECT * FROM pt_sc_mt;
```

ProfGroup Detail #22

Execution Summary

Application
　　/home/kabe/solver/solver_mpich2_tau
# of nodes
　　16
# of processes
　　128
Place/Location
　　cloko
Library
　　MPICH2 GCC NFS

Each Execution Information

There are 10 execution logs.

ProfExec ID	Execution Time	Start TimeStamp
211	68.796506	1298613703108843
212	67.874299	1298613772892059
213	66.841724	1298613841765232
214	69.05403	1298613909708809
215	65.643318	1298613979747787
216	66.716533	1298614046384141
217	70.722617	1298614113971593
218	68.924956	1298614185783454
219	70.173506	1298614255695244
220	68.2141	1298614327539868

図 5.4. MPI アプリケーションの実行条件サマリ画面

Workflow PARP

Workflow List

ID	Workflow Name
227	jawc
234	jawc-pt
228	montage

図 5.5. ワークフローアプリケーションのアプリケーション選択画面

Trials

Trial ID	Location	Filesystem	Worker #	Input Dataset	Execution Time [sec]	TimeChart Link	Enabled
288	tsubame2	gpfs	128	data2	235	Timechart	☑
289	tsubame2	gpfs	128	data2	237	Timechart	☑
290	tsubame2	gpfs	128	data2	235	Timechart	☑
291	tsubame2	gpfs	128	data2	241	Timechart	☑
292	tsubame2	gpfs	128	data2	240	Timechart	☑
293	tsubame2	gpfs	128	data2	236	Timechart	☑
294	tsubame2	gpfs	128	data2	232	Timechart	☑
295	tsubame2	gpfs	128	data2	237	Timechart	☑
296	tsubame2	gpfs	128	data2	232	Timechart	☑
297	tsubame2	gpfs	128	data2	236	Timechart	☑
							Switch

図 5.6. ワークフローアプリケーションの実行条件サマリ画面

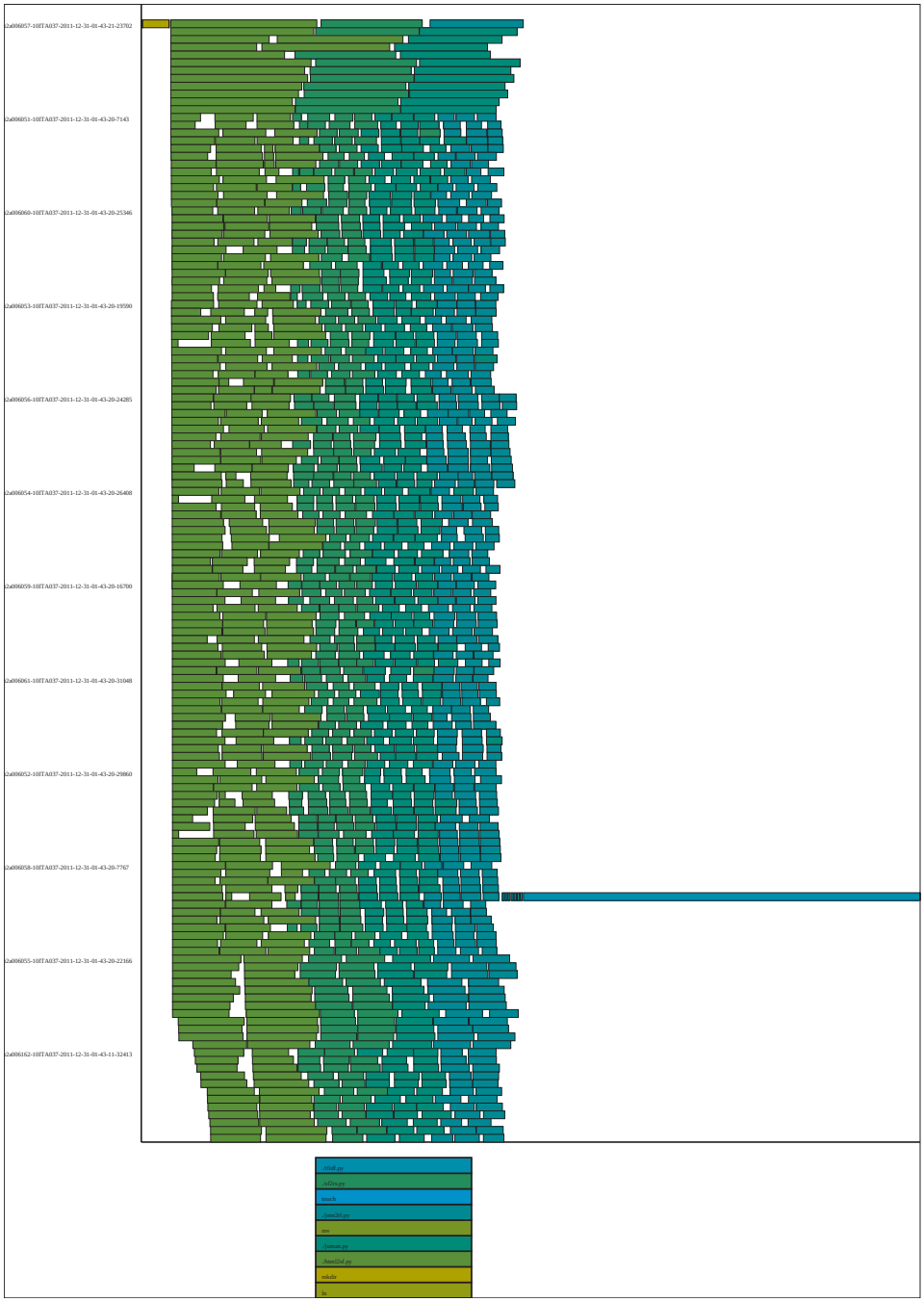

図 5.7. ワークフローアプリケーションの実行可視化例

第6章　評価

6.1　評価環境

まず, 評価全体についてある程度共通している, 評価環境について述べる.

6.1.1　クラスタ構成

評価に用いたクラスタ環境の概要を表 6.1 に示す.

このクラスタは図 6.1 のようなネットワーク構成になっている Beowulf クラスタである. 2 つのネットワークインタフェースが利用可能になっているが, 通常は高速な 10GbE のネットワークを用いることが想定されており, 1GbE のネットワークはクラスタの管理用に利用されている.

MPI ライブラリ　クラスタ環境では Ethernet に対応し, 広く利用されている MPI ライブラリとして MPICH2 と Open MPI という 2 つの実装を用いた. MPICH2 はバージョン 1.2.1p1 と hydra[6, 12] の組み合わせ, Open MPI はバージョン 1.4.2 を用いた.

6.1.2　Amazon EC2

ワークフローアプリケーションの実行には Amazon EC2 環境を用いている. ここでもクラスタ構成とおおむね同じようにネットワークを構成している. システム構成の諸元を 表 6.2 に示す.

6.1.3　Tsubame2

ワークフローアプリケーションの実行には Tsubame2 を利用した. Tsubame2 では NFS, Lustre, GPFS といった共有ファイルシステムが利用可能であり, アクセスの様子によって性能が異なっている. システム構成の諸元を表 6.3 に示す.

表 6.1. 評価に用いたクラスタ環境の一覧

環境名	CPU	ネットワーク
hosei	Intel Xeon E5530 @ 2.40GHz	10GbE & 1GbE

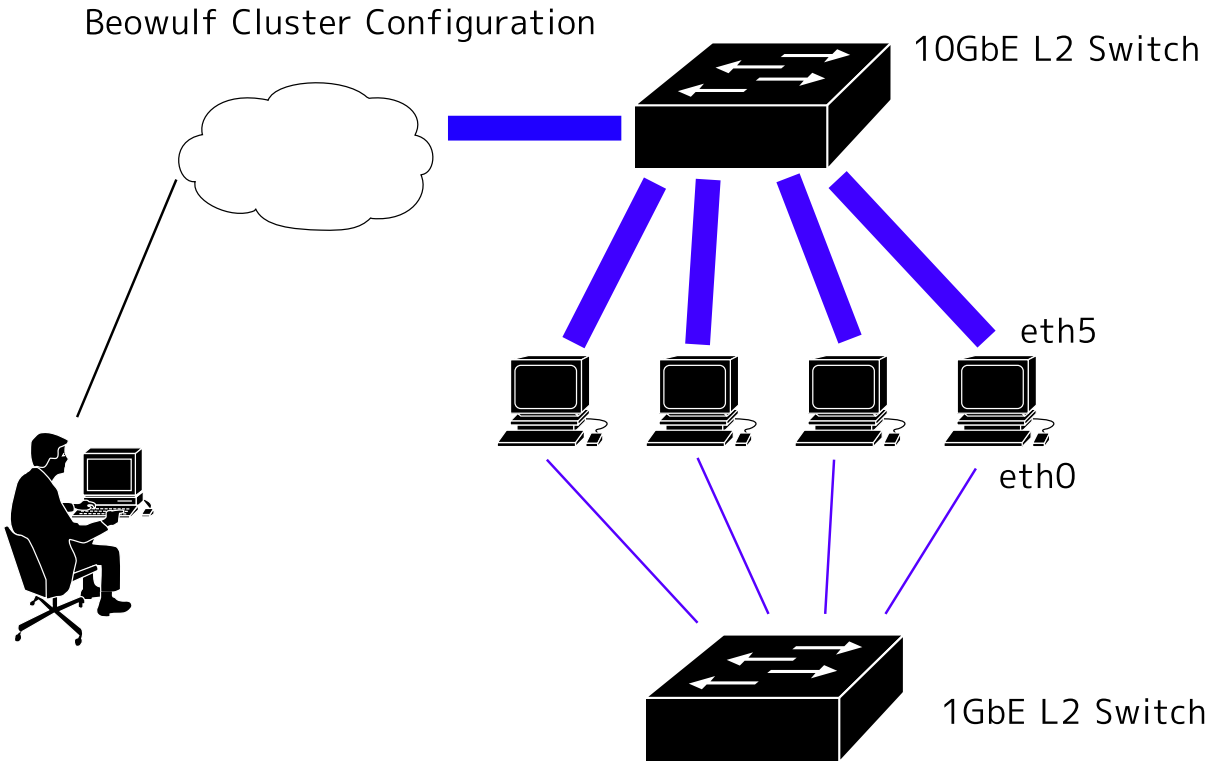

図 6.1. 評価に用いたクラスタのネットワーク構成図

表 6.2. Amazon EC2 システム概要

利用ノード	CPU	Interconnect	ファイルシステム
16 台 (c1-xlarge)	ParaVirtualized @2.33GHz 4cores × 2	Virtual Network	NFS

6.2　MPI アプリケーションを対象とした解析

6.2.1　PageRank アプリケーション

MPI アプリケーションの 1 つとして, PageRank[35] を計算するアプリケーションを例に挙げる. このアプリケーションの性能解析をケーススタディとして用いる. アプリケーションの入力データは人工的に生成したものであり, 表 6.4 に示すようなパラメータをもつ.

PageRank の計算内容については詳細を省略するが, このアプリケーションの内部で発生している計算と通信の様子については述べておく. 入力データはウェブページ同士の参照関係を模した有向グラフとなっている. このグラフを MPI プロセスの数分で事前に分割したものが実際に MPI プロセスが読み込む入力データとなっている. これは全体グラフのサブグラフと, 隣接サブグラフとの結合に関する情報を保持している. 各 `rank` はローカルで PageRank の更新を行い, その後隣接サブグラフと値を交換する集合通信を行う. そして同期を行ったのち, またローカルの計算に戻る. 実際の PageRank アプリケーションではこれを収束まで繰り返すことになっているが, 本評価

表 6.3. Tsubame2 システム概要

利用ノード	CPU	Interconnect	ファイルシステム
11 台 (Thin)	Intel Xeon X5670 @ 2.93GHz (12 cores)	Infiniband (80Gbps)	NFS, Lustre, GPFS

表 6.4. 評価に用いる PageRank アプリケーションの入力データ

合計節点数	合計エッジ数	サブグラフ間のエッジカット	サブグラフ数
128,000,000	約 512,000,000	約 51,200,000	128

ではこの繰り返しを 10 回行った時点で計算を終了している.

この入力データの場合, 128 個の MPI プロセスにより計算を行ったときの通信は各 MPI プロセスの組み合わせあたり平均しておよそ 300KB 程度, 平均 13 回であった.

評価ではこのアプリケーションを MPICH2, Open MPI を用い, クラスタ上でそれぞれを 10 回ずつ実行した. それぞれの実行コマンドをリスト 6.1, リスト 6.2 に示す. これを PARP のコマンドラインツールを用い, データベースへと登録した.

リスト 6.1. MPICH2 を用いた PageRank アプリケーションの実行コマンド

```
1 $ mpiexec -n 128 -f hosei_mpich2.txt ./pagerank_mpich2 ...
```

リスト 6.2. Open MPI を用いた PageRank アプリケーションの実行コマンド

```
1 $ mpiexec -np 128 -machinefile hosei_openmpi.txt ./pagerank_ompi ...
```

PARP Web インタフェースからはそれぞれの条件（ここでは MPICH2 と Open MPI）について, 複数回での実行の平均実行時間をサマリテーブルに表示している. ここから実行時間を確認すると, MPICH2 は約 10 秒, Open MPI は約 64 秒となっており, 大きな性能差が出ていることがわかる (図 6.2). ユーザサイドのコード, すなわち MPI を利用する側のコードは一切変更していないため, この結果から何が起こっているのかを理解するのは容易ではない.

そこで, この 2 つの条件において, 各関数の実行時間を比較することにより, 性能差が出ている部分を絞り込むことを考える. PARP が提供するビューは複数回実行の平均の, さらに複数並列ユニットの平均値（128 並列のときにその平均）を提供しているため, ここではその値を用いる. ビューの定義 (リスト A.3) は Web インタフェース上で表示されている. また, 性能差をみるため, 関数実行時間の差 `timediff` カラムおよび実行時間の比 `ratio` カラムを追加するよう設定する (図 6.3). この `Column Definition SQL` の内容は直接 SQL クエリに利用されるため, 自由に SQL による

PG 1	PG 2	Application	Comment	Place	# of nodes	# of Processes	Avg. Time [sec]	StdDev. [sec]
○	○	PageRank	MPICH2 GCC NFS Detail	hosei	16	128	10.012	5.566
○	○	PageRank	OpenMPI GCC NFS Detail	hosei	16	128	64.616	11.744

図 6.2. PageRank アプリケーションの平均実行時間と標準偏差

Column Definitions

Use?	Column Name	Column definition SQL	ORDER BY	Graph Style
☑	funcname	pr1.funcname	○	boxes
☑	exectime1	pr1.excl_pe_rank_avg	◉	boxes
☑	exectime2	pr2.excl_pe_rank_avg	○	boxes
☑	timediff	(pr1.excl_pe_rank_avg - pr2.excl_pe_rank_avg)	○	boxes
☑	ratio	(pr1.excl_pe_rank_avg / pr2.excl_pe_rank_avg)	○	points
☐			○	boxes
☐			○	boxes
☐			○	boxes
☐			○	boxes
☐			○	boxes

図 6.3. PageRank アプリケーションの比較に用いるカラム指定画面

記述をして構わない. このときさらに, グラフ出力時にどのような形で出力するかを指定することができる (GRAPH STYLE). ここでは実行時間を縦棒 (`boxes`) で, 実行時間の比 (`ratio`) を十字点 (`points`) で表示するよう指定した. この操作により, 図 6.4 に示す表がユーザに提示される. これは実際にユーザが指定したカラムの情報を取得するように SQL を実行した結果, どのようなデータが得られたかを示している. ここでは関数の名前, 実行時間が 2 つ, 時間の差, そして時間の比が順に得られている. そして, `exectime1` の降順に（図 6.3 の “ORDER BY” により指定）, 上位 10 関数が示されている. “See More ...” ボタンをクリックすると, さらに下位の関数も表示されるが, 下位の関数は実行時間にそれほど影響していないと思われるので, 今回は利用しない. この結果, 図 6.5 が得られた. ここで**図 *6.5* は *PARP* からの出力から一切変更していない**ことに注意されたい. ユーザからパラメタが与えられたのち, グラフの作成は PARP が完全に自動的に行う.

このグラフでは赤い棒 (`exectime1`) が Open MPI を用いたときの各関数の実行時間, 緑の棒 (`exectime2`) が MPICH2 を用いたときの各関数の実行時間, そして十字の点により実行時間の比 (`ratio = exectime1 / exectime2`) を示している. ここでは このグラフを見ると, `MPI_Wait` をはじめとして, MPI に関係する関数が上位に上がっていることがわかる. 5 番目に `graph_propagate` という MPI でない関数がある. これはユーザ定義の関数であり, 実際に内部では通信は行っておらず, 各 MPI プロセス内部で完結する処理である. この時間にはほぼ差がない. もともとプロセッサなどの環境が全く同じであり, 通信を行っていないので, 差がないのが自然である.

一方, 4 番目に挙がっている `MPI_Send` 関数は実行時間は 4 番目となっているが, 実行時間の比が非常に大きい（具体的には図 6.4 から 173 倍の差であると読み取れる）. ここには何か大きなヒントがあると考えてよいと思われる. 一方, `MPI_Wait` 関数は MPI で様々な使われ方をし, また同期をとる関数なので, この関数から調べていっても直接性能改善に結びつけるのは困難であると考えられる. ここではこの `MPI_Send` に注目し, 性能改善を試みる.

Comparison Table

Create Graph!

funcname	exectime1 y1 ☑ y2 ☐	exectime2 ▼ y1 ☑ y2 ☐	timediff ▼ y1 ☐ y2 ☐	ratio ▼ y1 ☐ y2 ☑
MPI_Wait()	24621578.731	3594479.682	21027099.049	6.850
MPI_Allreduce()	18611032.551	2559182.858	16051849.693	7.272
MPI_Alltoallv()	12004672.638	1602304.490	10402368.148	7.492
MPI_Send()	5643023.398	32590.871	5610432.527	173.147
graph_propagate()	1640448.305	1740932.347	-100484.042	0.942
MPI_Init()	1353648.838	207257.883	1146390.955	6.531
MPI_Alltoall()	503285.508	26117.565	477167.943	19.270
graph_allreduce()	97643.109	97863.217	-220.108	0.998
fastsort()	69401.746	70056.655	-654.909	0.991
MPI_Finalize()	23914.880	34900.775	-10985.895	0.685
See more ...				

図 6.4. PageRank アプリケーションの比較用グラフに表示する軸設定画面

このアプリケーションでの MPI_Send の使われ方を ParaProf を用いて調べてみると, 各 rank-rank 間で合計 21 万回以上呼ばれているものの, 総通信サイズは非常に小さいことがわかる (図 6.6, 図 6.7). この実行環境では 10GbE, 1GbE の 2 ネットワークが接続されているが, このネットワークの速度による差ではないように思われる. そこで, MPICH2, Open MPI 2 つの MPI 実装の中でも, MPI_Send の実装から, 性能差がどこにあるかを調べてみる. いずれの実装も様々な通信インタフェースに対応しており, それによって実装は異なっているが, ここでは TCP 通信を利用しているので, TCP による通信を行っている部分のコードを解析した結果を示す.

MPICH2 の MPI_Send　MPICH2 は hydra を用いている. hydra は MPI の実行開始時に各計算ノード上に hydra デーモンを起動し, この hydra デーモン同士が通信を管理する. MPI プロセスはこの hydra デーモンを用い, 各 rank 間の通信を行う. ある計算ノードのペアを考えたとき, そこには複数の MPI プロセスが存在するにもかかわらず, socket が 1 本では通信性能が悪化してしまうため, hydra では自動的に複数の socket を計算ノード同士の間で確立する. 既に socket が存在しているため, src から dest にメッセージを送信するとき, MPI_Send の中身は限りなく単純に dest への socket を探し出し, send(2) を呼び出すものに等しい.

Open MPI の MPI_Send　Open MPI は MPICH2 と大きく異なり, MPI プロセス同士が直接 rank 間の通信を行う. ある rank でプロセス src がメッセージを dest へ送信しようとしたとき, src から verb へ利用可能な通信路があればそれを用いるが, 何らかの理由（既に close されている, 通信路が応答しない, など）があれば新たに socket を生成し, connect(2) を呼び出す. これにより利用可能な socket があることを保証したのち, send(2) を呼び出す. このアルゴリズムを Algorithm 6.2.1 に示す.

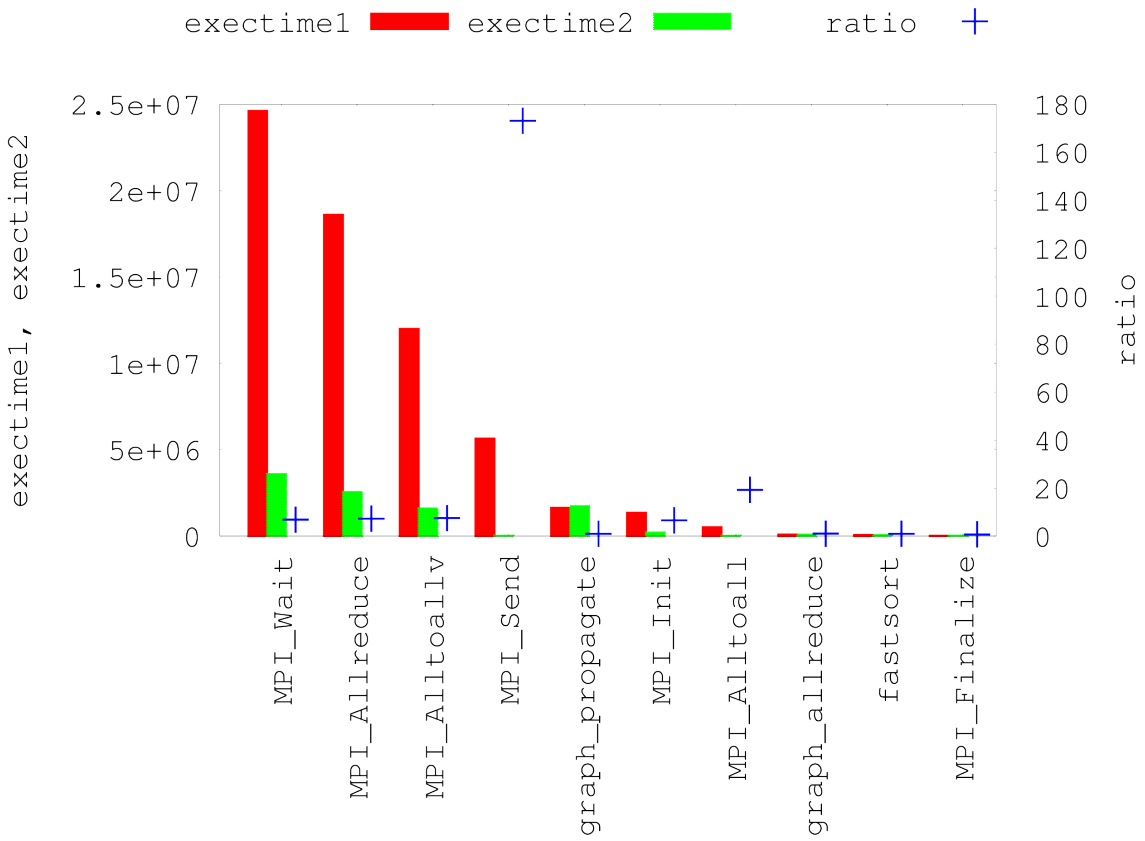

図 6.5. PageRank アプリケーションの比較グラフ

性能差の原因の理解

これらの実装の理解をもとに, Open MPI で性能が悪かった理由を考えてみると, 何らかの理由で socket が大量に生成されてしまったのではないかという推論ができる. そこで, 実際に Open MPI でアプリケーションを実行している途中に netstat(8) を用いてどのような socket が存在しているかを確認した. すると, 実際に 1 台の計算ノード上で瞬間 3600 本以上の socket が作成されていたことがわかった. これは TCP の状態が ESTABLISHED だけでなく, TIME_WAIT のものも含んでいる. 一方 MPICH2 では最大 1100 本程度であった. すなわち, socket の本数が 3 倍以上あるということになる. さて, netstat(8) では単に socket の数だけでなく, 実際に TCP レイヤでどのように接続されているかをみることができる. すると, MPICH2 では全ての socket が 10GbE のインタフェース経由であったのに対し, Open MPI では 10GbE のインタフェースと 1GbE のインタフェースが 2 : 5 程度で混在していることが明らかになった. Open MPI では [1, 36] で示される手法により, 計算ノード間の接続が新たに必要となったとき, 複数のインタフェースが利用可能な場合, どのインタフェースを実際に利用するかを決定している. 今回のケースでは, 何らかの事情により, このインタフェースの決定アルゴリズムがうまく機能せず, 非常に多くの socket が生成されるようになってしまったものと考えられる.

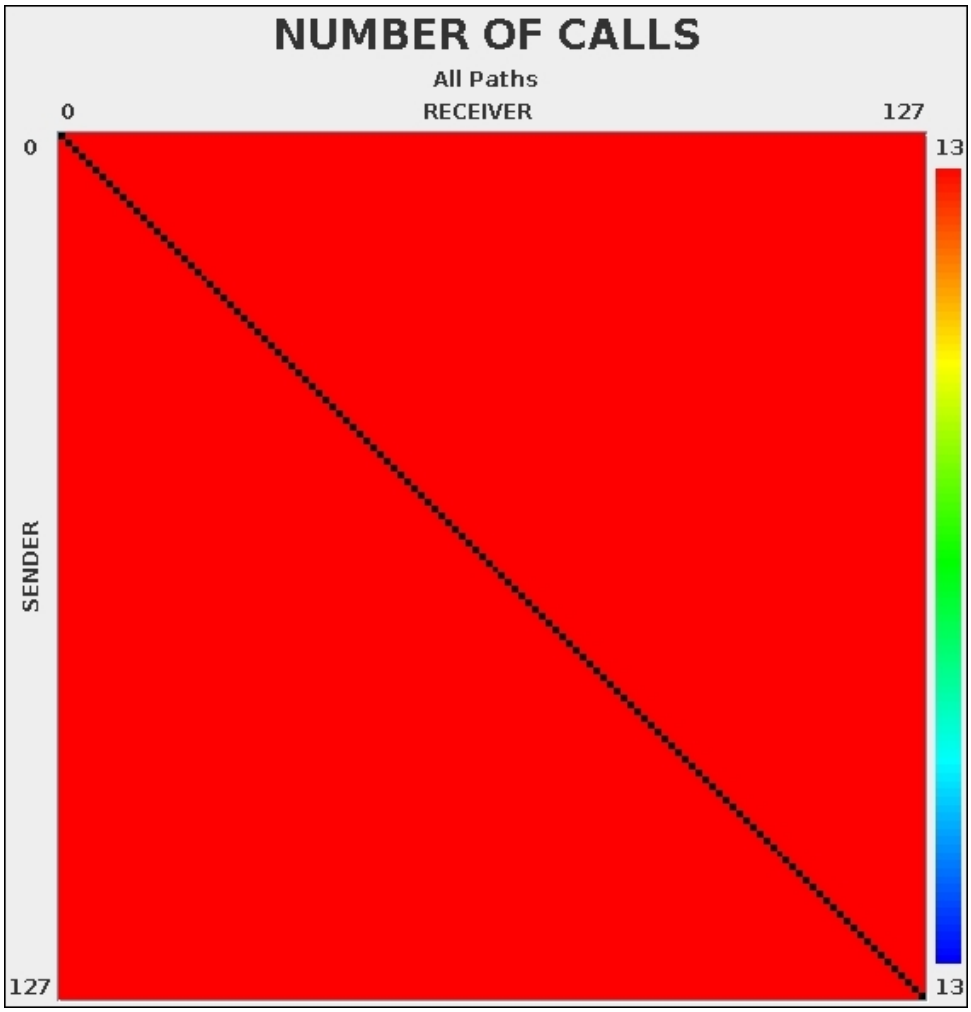

図 6.6. PageRank アプリケーションの MPI_Send によるメッセージ送信数 (ParaProf より)
各 MPI プロセス同士は 13 回ずつ `MPI_Send` を呼び出したことがわかる.

性能の改善

この観察と推論から, 通信インタフェースとして 10GbE の側のみを使用するようにすれば性能が改善すると見込むことができる. そこで, MPI アプリケーションの実行時にネットワークインタフェースとしてループバックアドレスである `lo`, および 1GbE のインタフェースである `eth0` を利用しないようにすることで, 通信の際確実に 10GbE のインタフェースである `eth5` を利用するよう設定した (リスト 6.3). その結果, ソケットの数が最大でも 1400 本弱となり, MPICH2 の場合との差が非常に小さくなった. 実際の実行時間もおよそ 8.3 秒となり, 設定を行わなかった 64.4 秒に比べて 7.7 倍以上の性能向上が得られた.

リスト 6.3. Open MPI を用いて適切なインタフェースを設定した PageRank の実行コマンド

```
1 mpiexec --mca btl self,sm,tcp --mca btl_tcp_if_exclude lo,eth0 \
2     -np 128 -machinefile hosei_openmpi.txt ./pagerank_ompi ...
```

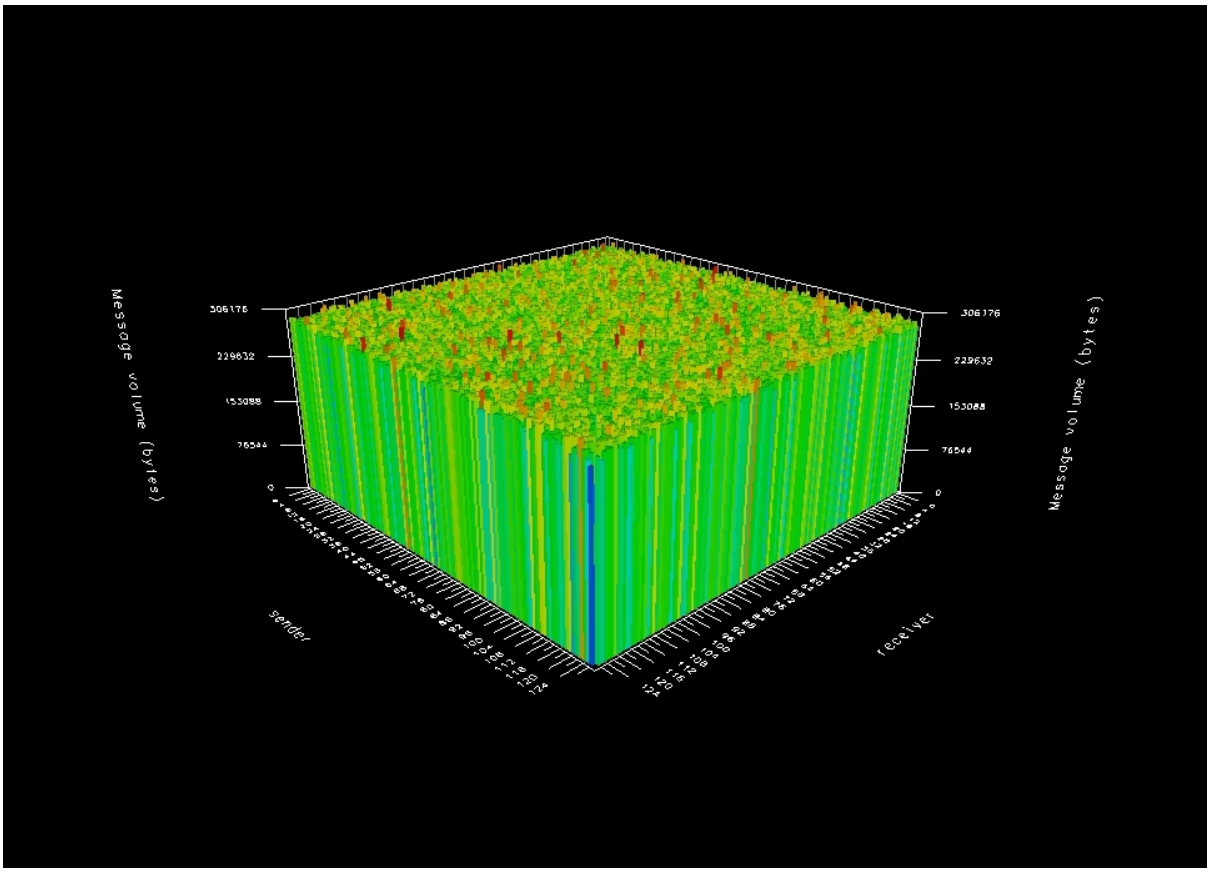

図 6.7. PageRank アプリケーションの MPI_Send によるメッセージ総サイズ (ParaProf より)
縦軸がメッセージ総サイズになっており, 最大値が 300KB 程度である.

PARP 順序計量の導入

図 6.5 では実行時間が長い方の実行時間 `exectime1` でソートしたデータを用いてグラフを生成したが, 5.4.4 節で導入した PARP 順序計量を用いるとグラフがどのようになるかを 図 6.8 に示す.

図 6.5 では `graph_propagate` のように, 性能差がほとんどない, あるいは全体の実行時間が長い側の方が実行時間が短い関数が比較的上位にリストされていた. 一方, 図 6.8 では `graph_propagate` は上位 10 関数から外れ (実際には最下位となった) , 一方前節で解析対象とした `MPI_Send` は 4 位から 3 位に上がった. このアプリケーションは関数の数が少なかったためそれほど目立たないが, 関数の数が増えてきたとき, ユーザに見せる情報の量を一定にし, そのなかでより意味のある情報を提示できるようになったと考えられる.

6.3　ワークフローアプリケーションを対象とした解析

ここではワークフローアプリケーションを実際に PARP を用いて解析した例を示す.

Algorithm 6.2.1 Open MPI 実装における src から dest へのメッセージ送信

$sock \leftarrow find_sock_from_to(src, dest)$
if $sock$ is not available **then**
　$sock \leftarrow create_new_socket()$
　$connect_sock_from_to(sock, src, dest)$
　$register_socket_from_to(sock, src, dest)$
else
　$send_message(sock)$
end if

6.3.1　日本語ワードカウント

本節で性能の解析対象とするのは日本語の HTML 文書を複数入力とし, それらのうちに含まれる単語の tf-idf[42] を計算するアプリケーションである. それぞれの HTML ファイルから tf を計算するところまでは各ファイルにおいて完全に並列実行可能なアプリケーションとなっており, 最後に tf から idf を計算するところでは並列実行しないような実装になっている. このアプリケーションの DAG は 図 6.9 で示される.

このアプリケーションの入力として全体で 221MB のデータを用い, Amazon EC2 のクラスタおよび Tsubame2 を用いた. この入力データでは, 並列実行される部分が 386 並列となっている.

Amazon EC2 では 8 コアマシンを 16 台用いて仮想 128 CPU を利用したため, Tsubame2 でも並列度を揃え, 最大 128 並列での動作としている. これは GXP Make による実行の際, 並列度を設定する機能により実現する. このコマンドをリスト 6.4 に示す. `gxpc make` 直後のオプションはそのまま GNU Make に引き渡される. `--` 以降は GXP Make のみが解釈するオプションである.

リスト 6.4. GXP Make による JAWC の実行 (128 並列)

```
1 dataset=data2
2 PARALLELISM=128
3 outdir=$HOME/jawc_result/
4 gxpc make \
5     INPUT=$dataset all \
6     -j ${PARALLELISM} \
7     -- -a state_dir=$outdir
```

2 つの環境で実行を行い, データベースに性能情報を格納したところ, Tsubame2 での実行に 172.8 秒, Amazon EC2 での実行に 254.7 秒を要したことが Web インタフェースからわかる (図 6.10).

この結果そのものは Tsubame2 の方が高速に動作したということで, 異常な性能差ということはないが, 実際にどのように性能が異なっていたのか, 調べてみるにはよい題材であると考えられる. 2 つの結果を比較するよう指定すると, 図 6.11 が得られる. このグラフには, 各コマンドの総実行時間と, 式 5.1 から推定した I/O 時間の総和が, 2 つの実行について示されている. ここで, 添字に 1 がついているのが Tsubame2 の数値, 2 がついているのが Amazon EC2 の数値である. 他にも主だった数値について, デフォルトで軸を設定し, 非表示にしてある. ユーザが見たい情報がこの中に含まれていれば, 図 6.11 の右上にある凡例の該当する数値をクリックすることで, 表示/非表示を自由に切り替えることができる.

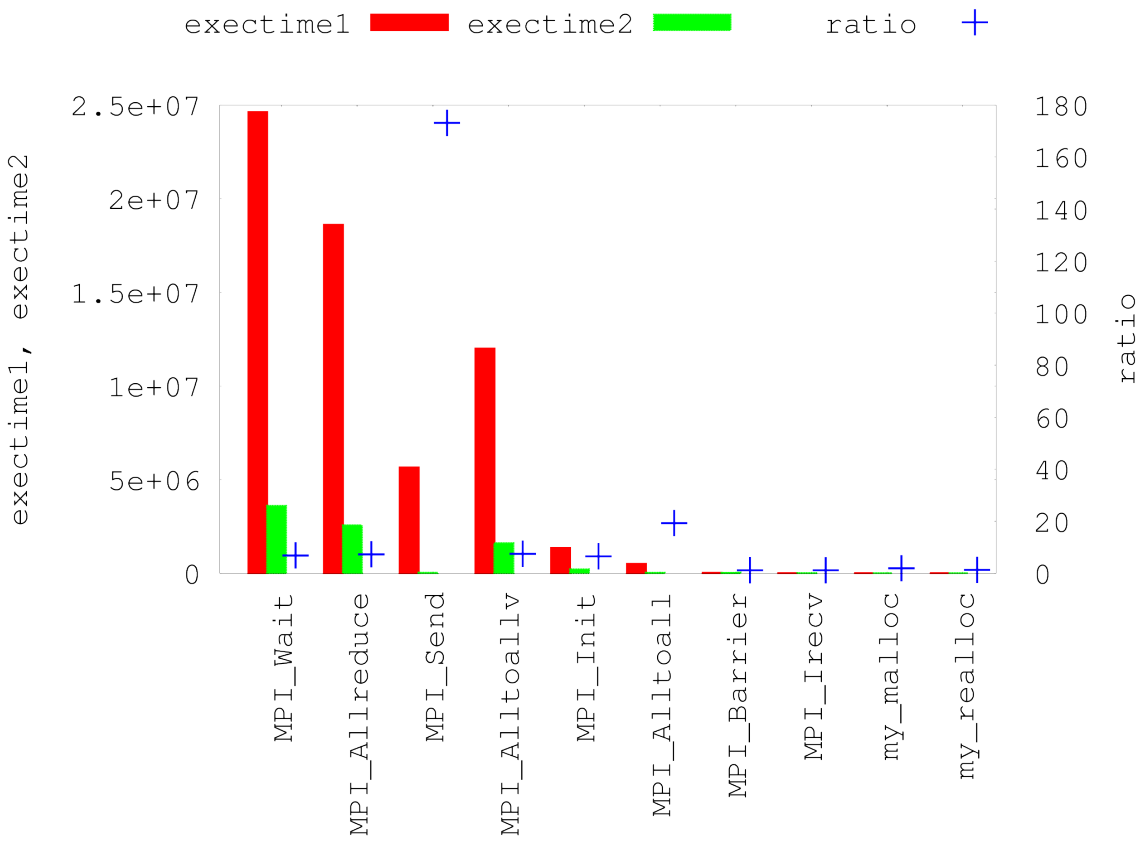

図 6.8. PageRank アプリケーションの比較グラフ（PARP 順序計量を用いた場合）

現在デフォルトでこのように軸を設定しているが, ここに含まれていない値をグラフに出力したい場合というのも考えられる. そのとき, MPI 向け実装と同じく, Web フォームを埋めることで軸の設定をすることもできる. それと別に, ワークフロー向け実装ではユーザからの明示的な SQL 文の指定に対応している. これは Web インタフェース内に設けられたテキストボックスに問い合わせ用の SQL を記述し, 実行させるというものである. ここでは実際に, ユーザ CPU 時間, I/O 時間の推定値, そして実行時間の 3 つを, それぞれの実行条件のものについてグラフ出力するような SQL を記述し, グラフを作成してみる. 実際に利用した SQL はリスト 6.5 に示した. ここから得られたグラフを 図 6.12 に示す. **このグラフは *PARP* からのものそのままであり, 一切編集していない**. このグラフをみると, 特に `./html2sf.py` というコマンドが長く時間を消費しているようなので, 以後, このコマンドに絞って性能をみていく. 全体としての実行時間 `Accum1` と `Accum2` の差に比べ, ユーザ CPU 時間 `User1`, `User2` の差は比較的小さく, I/O 時間の推定値 `IO1`, `IO2` の差がかなりの部分を占めていることがわかる. 実際のところ, Amazon EC2 上でのファイル I/O はそれほど高い性能を期待することはできず, Tsubame2 では Lustre を利用していたため, ある程度高い性能を得ることができたと考えられる. またユーザ CPU 時間についてみてみると, CPU の周波数の比が 2.93/2.33 = 1.25 倍の性能差であり, ユーザ CPU 時間の比が 3577.0/2780.0 = 1.28 倍の時間となり, ほぼ一致していることから, CPU の性能差についてはユーザ CPU 時間にほぼそのまま反映

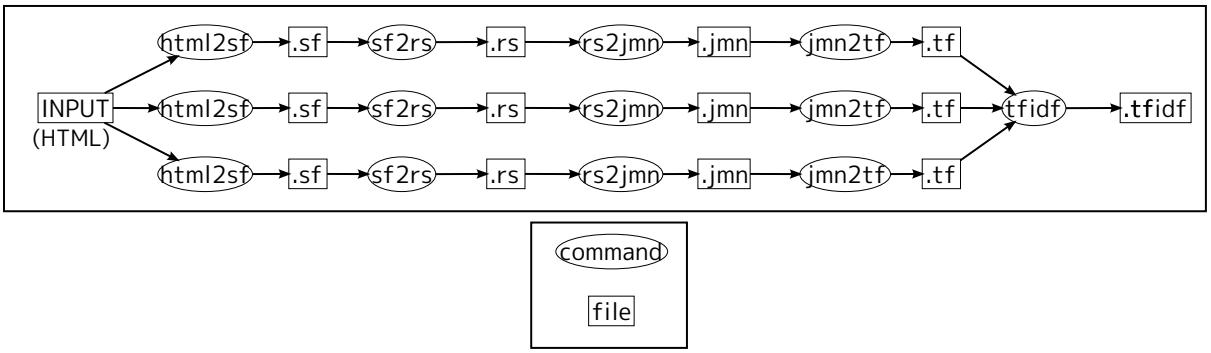

図 6.9. 日本語ワードカウントアプリケーションの DAG

Conditions

Condition ID	Location	Filesystem	Worker #	Input Dataset	Average Time [sec]	# of Enabled/Trials	SUM of elapsed local [sec]	User Time [sec]	Estimated I/O [sec]
63	tsubame2	lustre	128	data2	172.8000	10/10	5046.55	2802.68	2243.88
64	ec2	nfs	128	data2	254.7000	10/10	9434.41	4119.99	5314.42

図 6.10. PageRank アプリケーションの比較グラフ（PARP 順序計量を用いた場合）

されたと考えられる.

リスト 6.5. 日本語ワードカウントの性能解析 SQL の一例

```
SELECT
  app.name ApplicationName,
  (cv1.time_user) * cv1.app_count User1,
  (cv1.elapsed_local - cv1.time_user) * cv1.app_count IO1,
  cv1.elapsed_local * cv1.app_count AccumL1,
  (cv2.time_user) * cv2.app_count User2,
  (cv2.elapsed_local - cv2.time_user) * cv2.app_count IO2,
  cv2.elapsed_local * cv2.app_count AccumL2
FROM
  condition_view cv1,
  condition_view cv2,
  application app
WHERE
  cv1.application = cv2.application AND
  cv1.workflow = cv2.workflow AND
  cv1.wf_condition = 193 AND
  cv2.wf_condition = 194 AND
  cv1.application = app.id AND
  cv1.workflow = 227
ORDER BY
  AccumL2 DESC;
```

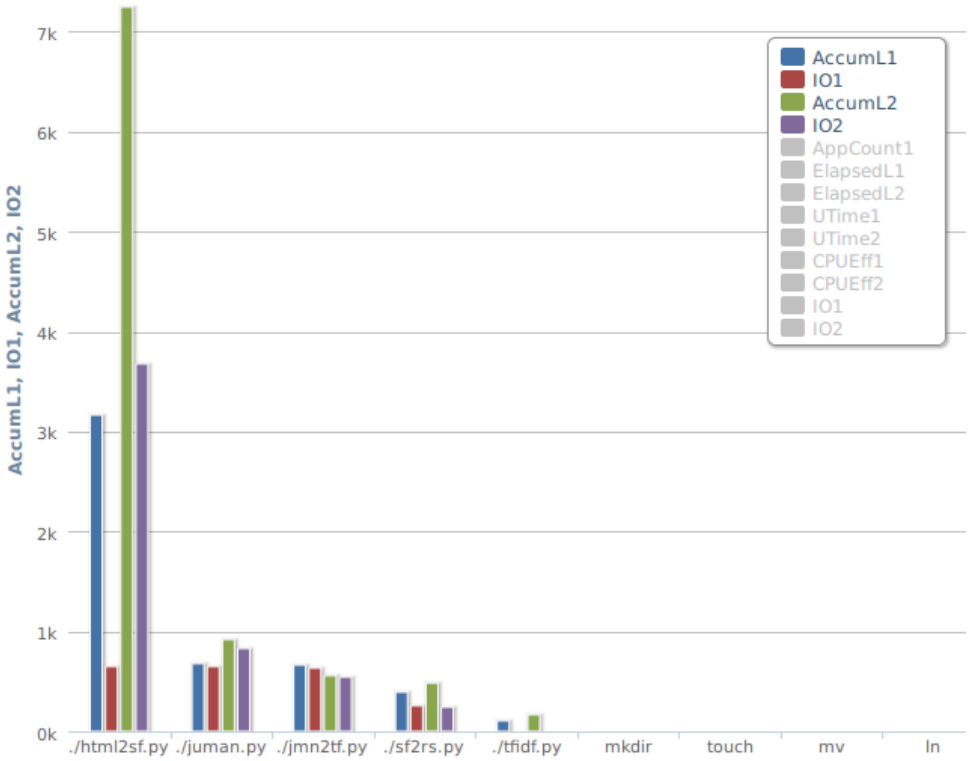

図 6.11. 日本語ワードカウント Tsubame2 と Amazon EC2 の比較

6.3.2　ParaTrac の情報の統合

前節まででユーザ CPU 時間とそれ以外という性能情報の切り分けを行うことができた. ここでは ParaTrac を用いた場合, どのように性能差が表われるかを, 同じく日本語の tf-idf を求めるアプリケーション, Tsubame2 を用いて実験した結果を用いて示す.

本実験は Tsubame2 で利用可能なファイルシステムのうち, GPFS と Lustre を用いて行った. NFS を用いた実験も行ったが, 正しくログを取得することができなかった. このとき, Lustre が非常に高負荷状態での実験となってしまい, *Lustre* **の性能が著しく劣化してしまっているが**, **これは本来の** *Lustre* **の性能ではない**ことに注意されたい.

本実験は各計算ノードで ParaTrac を動作させ, ParaTrac のマウントポイントを通じて全てのファイルアクセスを行うように環境を準備して行った (リスト A.4 には GPFS を用いたときのスクリプト例を示した).

この実験では, Lustre において 788 秒, GPFS において 236 秒という実行時間となった (図 6.13). この Lustre の結果は前述の通り本来の性能ではなく, 本来の Lustre であれば 200 秒程度となったはずである.

ここではファイルアクセスの性能差が, どのような操作によって出ていたのかを解析する例を示す. 既に PARP によりサマリテーブルが作られていることは述べた. これを用いることで簡単な SQL 文を用いて各ファイルアクセス関数ごとの性能を調べることができる. この解析に用いた SQL 文をリスト 6.6 に示す.

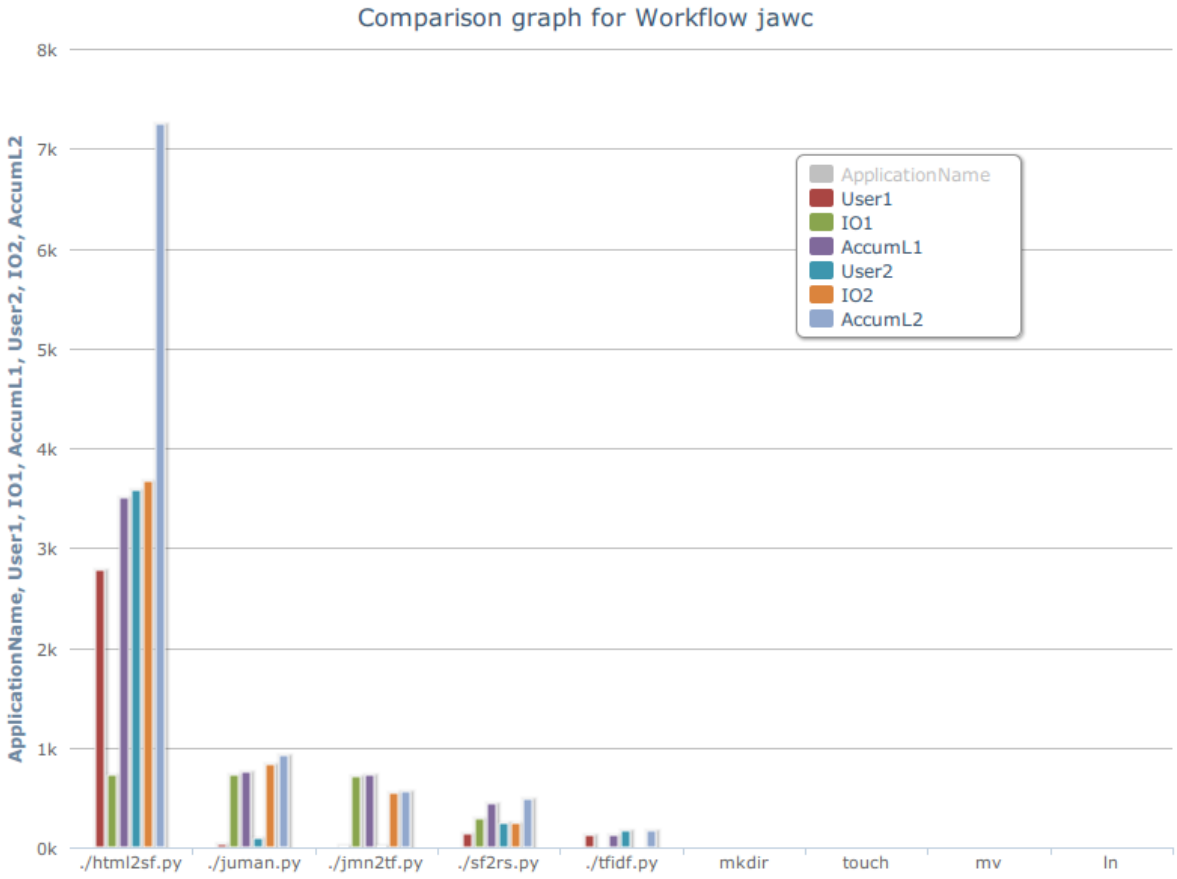

図 6.12. リスト 6.5 による性能比較結果

リスト 6.6. ParaTrac からの情報を用いたファイルアクセス性能の解析用 SQL

```
SELECT
  psms_a.syscall_name,
  psms_a.filesystem filesystem_a,
  psms_b.filesystem filesystem_b,
  AVG(psms_a.elapsed_sum) sum_a,
  AVG(psms_a.elapsed_avg) avg_a,
  AVG(psms_a.count) count_a,
  AVG(psms_b.elapsed_sum) sum_b,
  AVG(psms_b.elapsed_avg) avg_b,
  AVG(psms_b.count) count_b,
  AVG(psms_a.elapsed_sum - psms_b.elapsed_sum) sub_sum,
  AVG(psms_a.elapsed_avg - psms_b.elapsed_avg) sub_avg,
  ABS(AVG(psms_a.elapsed_sum - psms_b.elapsed_sum)) sub_sum_abs,
  ABS(AVG(psms_a.elapsed_avg - psms_b.elapsed_avg)) sub_avg_abs
  FROM pt_sc_mt_summary2 psms_a, pt_sc_mt_summary2 psms_b
  WHERE
    psms_a.workflow_id = psms_b.workflow_id AND
     psms_a.sysc = psms_b.sysc AND
     psms_a.wf_condition = 193 AND
```

Condition ID	Location	Filesystem	Worker #	Input Dataset	Average Time [sec]	# of Enabled/Trials	SUM of elapsed local [sec]	User Time [sec]	Estimated I/O [sec]
193	tsubame2	lustre	128	data2	788.1000	10/10	21292.33	3158.98	18133.35
196	tsubame2	gpfs	128	data2	236.1000	10/10	12800.14	3034.21	9765.93

図 6.13. 日本語ワードカウント Lustre と GPFS の性能比較

```
    psms_b.wf_condition = 196
GROUP BY psms_a.sysc, psms_b.sysc, psms_a.wf_condition
ORDER BY sub_sum_abs DESC;
```

この SQL では「各ファイルアクセス関数ごとの実行時間の総和の差」の絶対値を降順にソートしている. 実行時間の総和をみてみると, 上位 10 関数は順に `open`, `lstat`, `fstat|,` verb—creat—, `close`, `readlink`, `read`, `write`, `readdir`, `flush` となった. しかし, 時間として目立つのは上位 4 関数程度であり, それ以降の関数はほとんど性能に関与しているとはいえないような性能差である. PARP で得られたグラフを 図 6.14 に示す. この結果から, 今回の性能差はファイルの中身を読み書きする性能より, 主にファイルのメタデータアクセスを行う性能の差によるものであることがわかった. 実際, Lustre は高負荷によりメタデータアクセスの性能が劣化してしまうことから, 現象が理解できる.

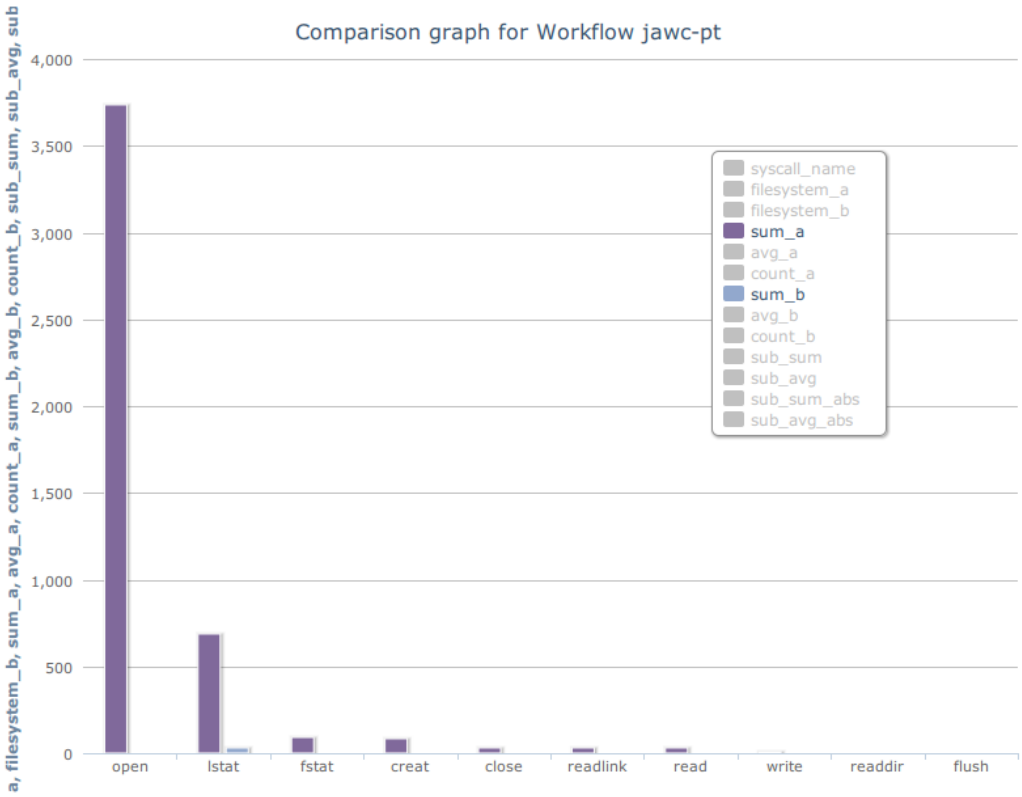

図 6.14. Tsubame2 におけるファイルアクセス性能比較結果
sum_a が Lustre での実行時間の総和, sum_b が GPFS でのものである. Lustre の性能は高負荷時のものであるため, Lustre 本来の性能を示すものではない.

第7章　おわりに

7.1　まとめ

本研究では並列アプリケーションの性能を解析する手法として**相対プロファイリング**を提案した. 相対プロファイリングはアプリケーションの性能を比較することに基づいており, これにより1回の実行のみを対象とするプロファイリングではみえないような性能劣化に気付いたり, 性能差の原因を絞り込むことで, アプリケーション全体としての性能を向上させることを目指している.

また, この実装として PARP を実装した. この実装はデータベース設計, データのインポートを行うコマンドラインツール, またユーザが解析に用いる Web インタフェースからなっている. 実装 PARP では広く使われている並列計算のライブラリを用いた MPI アプリケーションに加え, 広く使われはじめているワークフローアプリケーションの性能を解析することができる. また, ワークフローアプリケーションの性能に加え, ParaTrac から取得した情報を格納することで, ファイルアクセス性能についても解析を行うことができる.

評価において MPI アプリケーションとして PageRank を計算するアプリケーションそについて性能の評価を実際に行い, 性能を比較し, 今までは見過ごされてしまったであろう, 通信関数のオーバヘッドに着目することにより, インターコネクトに特化した部分において, 適切でないインターコネクトのインタフェースが利用されていることに気付き, 64.4 秒から 8.3 秒へ, 7 倍以上の性能改善を行うことができることを示した. ワークフローアプリケーションとしては日本語ワードカウントを行うアプリケーションを例に挙げた. ここでも性能を比較することにより, CPU 性能の差がほぼそのまま存在し, I/O をしていると推定されるその他の部分での差が大部分の性能差に結びついていることを明らかにすることができた. また, 2 つのファイルシステムで性能が大きく異なっていたとき, ParaTrac の情報を統合することにより, さらに踏み込んだ性能の解析が可能になり, どのようなファイルアクセス関数が性能差につながっていたかを示した.

これらの結果から, 相対プロファイリングを既存のプロファイリング手法と合わせて用いることで, より様々な環境での並列アプリケーションの性能改善に結びつけることができると期待される.

7.2　今後の課題

アプリケーションの性能を比較する研究はまだはじまったばかりである. 実際にどのような値を比較するべきか, 汎用的に使えるパラメタがあるのかどうか, など, まだ多くのことがわかっていない.

本実装の性能情報データベースは MPI 向けのものとワークフロー向けのものが分離してしまっている. 今後, これらのデータベースを統合し, 全く同じビューが提供できるようになればより汎用的になると考えられる. これに加え, データベース設計についても再考が必要である. 特に実行

条件を格納するテーブルは, 現在の実装では表現力が低く, 汎用的ではない. これらについてもより汎用的な実装にしていくべきと考えられる.

また, 性能を比較するインタフェースについても改良が必要である. 現在のインタフェースでは 2 つの条件で同じ値を見ようと思っても, それぞれ独立なカラムとして設定をしなければならないなど, ユーザ視点からみて必ずしも使い勝手がよいものとはいえない. これらについても, よいインタフェースの設計や実装が必要である.

謝辞

まず, 本研究を行うにあたり非常に数多くの助言をいただいた田浦健次朗准教授に深く感謝します. 研究のテーマおよび方針を決めたときには私にはまだ 1 歩先も見えるか見えないかという状態でした. 何度か完全に見失い, 途方に暮れたことも何度かありました. それでも常に「次にどうすればよいか」を示しながらもさらに先を示し続けていただいたおかげで, なんとかここまで漕ぎ着けることができました. そして近山隆教授には研究のまとめ方について, 貴重な助言を多くいただいたことに感謝いたします.

また, 研究に関して議論に加わっていただいた研究室の皆様に感謝します. 横山大作助教, Nan Dun 先輩, Ting Chen 先輩, 浦晃先輩, 中島潤先輩には多くの助言をいただきました. また, 原健太朗先輩には非常に数多くの部分であらゆる助言をいただき, 議論に参加していただき, 深く感謝しています. 池上克明君, 河野瑛君, 堀内美希さん, 中谷翔君, 藤田晃史君には突然研究室で研究の様子を話して強引に議論に巻き込んでしまうこともありましたが, おかげで自分の考えをまとめることができました. 吉本晴洋先輩には PARP の実装にあたり, highcharts.js を用いるコードのプロトタイプを手伝っていただきました. これにより実装がよりよいものになったと考えています.

I would also like to express my gratitude to Dr. Chee Wai Lee, at the Performance Research Lab of the Neuroinformatics Center in the University of Oregon, for giving me a great deal of help on collecting performance information from the internal of MPI implementations with TAU performance analyser.

また, この研究は私自身がコンピュータでできあがったシステムを深く理解しなければ進めることが到底できないものでした. ここまでの知識は本研究室にやってきて, 初めて身についたものです. この知識と付随する技術は多くの部分が吉富翔太先輩, 藤澤徹さんの指導によるものです. そして, 何をやらかすか分からない私に格好の遊び場を与えてくれたこの研究室の環境と, 数多くのサーバ機器, 周辺機器にも感謝いたします. 特に kiwi, kanga, tako ではかなり遊び, 知識と技術, そして代えがたい経験を積むことができました. 私が研究室に入って 1 週間後, 初めて root となった shepherd への愛着は今でも薄れません. ここまで 2 年間の研究生活は boron, palladium の 2 台に強く支えられました.

この修士課程の間, 多くの同期と共に大学院生活を過ごすことができました. それぞれが非常に秀れた人物であり, お互いに様々な面で成長することができたと思います.

最後になりますが, 私の日常生活は一緒に暮らす家族なしでは到底存在し得ませんでした. 父, 母, そして弟に, この日々を感謝します.

発表文献

[1] 加辺友也, 田浦健次朗. PARP: データベースを用いた並列アプリケーションのプロファイル比較. **先進的計算基盤システムシンポジウム** *(SACSIS2011)*, 2011. Poster Publication

[2] 加辺友也, 田浦健次朗. PARP: プロファイル比較に基づく並列アプリケーションの性能解析. **並列／分散／協調処理に関するサマーワークショップ（***SWoPP2011***）**, 2011.

参考文献

[1] FAQ: Tuning the run-time characteristics of MPI TCP communications. http://www.open-mpi.org/faq/?category=tcp.

[2] FUSE-devel Mailing List archive
"Re: [fuse-devel] Large read buffers". http://sourceforge.net/mailarchive/message.php?msg_id=28647216.

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

[4] Highcharts - Interactive JavaScript charts for your webpage. http://www.highcharts.com/.

[5] SPROF. GNU shared object profiling utility. http://www.gnu.org/software/libc.

[6] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Jayesh Krishna, Ewing Lusk, and Rajeev Thakur. PMI: a scalable parallel process-management interface for extreme-scale systems. In Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pp. 31–41. Springer Berlin / Heidelberg, 2010.

[7] Robert Bell, AD Malony, and Sameer Shende. ParaProf: A portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. *In Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pp. 17–26, 2003.

[8] Andrew R Bernat and Barton P Miller. Incremental call-path profiling. *Concurr. Comput. : Pract. Exper.*, Vol. 19, No. October 2006, pp. 1533–1547, 2007.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '95)*, pp. 207–216, 1995.

[10] OpenMP Architecture Review Board. OpenMP Application Program Interface, 2011. http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[11] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, Vol. 1, No. 1/2/3, p. 91, 2004.

[12] George Bosilca, Thomas Herault, Ala Rezmerita, and Jack Dongarra. On Scalability for MPI Runtime Systems. *2011 IEEE International Conference on Cluster Computing*, pp. 187–195, September 2011.

[13] Isaac Dooley, Chee Wai Lee, and Laxmikant V. Kale. Continuous Performance Monitoring for Large-Scale Parallel Applications. *2009 International Conference on High Performance Computing (HiPC)*, pp. 445–452, December 2009.

[14] 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)*, pp. 571–572. IEEE, 2010.

[15] 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*, pp. 37–48. ACM, 2010.

[16] Thomas Fahringer. Tools for Parallel and Distributed Computing. In Roman Trobec, Marián Vajteršic, and Peter Zinterhof, editors, *Parallel Computing*, chapter 3, pp. 81–115. Springer London, London, 2009.

[17] Message Passing Interface Forum. Mpi: A message-passing interface standard version 2.2, 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[18] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 6, pp. 736–748, 2010.

[19] Justyna Gidzinski. *μProfiler : A Concurrent Profiler for Concurrent C++ (μC++)*. Master's thesis, The University of Waterloo, 2007.

[20] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, Vol. 17, No. 6, pp. 120–126, 1982.

[21] William D. Gropp. Learning from the Success of MPI. In Burkhard Monien, Viktor Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing (HiPC 2001)*, Vol. 2228 of *Lecture Notes in Computer Science*, pp. 81–92. Springer Berlin / Heidelberg, 2001.

[22] R.J. Hall. Call path profiling. In *Proceedings of the 14th international conference on Software engineering*, pp. 296–306. ACM, 1992.

[23] John J. Hoffman, Andrew Byrd, Kathryn M. Mohror, and Karen L. Karavanic. PPerfGrid: A Grid Services-based Tool for the Exchange of Heterogeneous Parallel Performance Data. *HIPS-HPGC 2005 Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, in conjunction with IPDPS 2005*, 2005.

[24] K.A. Huck and A.D. Malony. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *ACM/IEEE SC 2005 Conference (SC'05)*, pp. 41–41. IEEE, 2005.

[25] Kevin A. Huck, Allen D. Malony, Robert Bell, and Alan Morris. Design and implementation of a parallel performance data management framework. In *International Conference on Parallel Processing (ICPP)*, pp. 473–482, Oslo, Norway, 2005. IEEE Computer Society.

[26] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. Scalable, Automated Performance Analysis with TAU and PerfExplorer. In *Parallel Computing (ParCo), Aachen, Germany*, pp. 1–8. Citeseer, 2007.

[27] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. Knowledge support and automation for performance analysis with PerfExplorer 2.0. *Scientific Programming*, Vol. 16, No. 2, pp. 123–134, 2008.

[28] Kevin A. Huck, Wyatt Spear, Allen D. Malony, Sameer Shende, and Alan Morris. Parametric Studies in Eclipse with TAU and PerfExplorer. In *Euro-Par 2008 Workshops-Parallel Processing*, pp. 283–294. Springer, 2009.

[29] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, Vol. C-36, No. 4, pp. 471–482, 1987.

[30] JianQiang Li, YuShun Fan, and MengChu Zhou. Performance modeling and analysis of workflow. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, Vol. 34, No. 2, pp. 229–242, March 2004.

[31] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, Vol. 88, No. 2, pp. 105–155, October 1990.

[32] Mubrak S. Mohsen, Rosni Abdullah, and Yong M. Teo. A Survey on Performance Tools for OpenMP. *World Academy of Science, Engineering and Technology*, No. 49, pp. 754–765, 2009.

[33] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen D. Malony, and Sameer Shende. TAUover-Supermon: Low-overhead online parallel performance monitoring. *Euro-Par 2007 Parallel*, pp. 85–96, 2007.

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

[35] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[36] Thomas Peiselt. *Improved Address Selection for Message Passing in Multi-Domain Cluster Systems*. Diploma, Friedrich-Schiller-Universitaet Jena, 2007.

[37] Aleksey Pesterev. *Locating Cache Performance Bottlenecks Using Data Profiling*. Master thesis, Massachusetts Institute of Technology, 2010.

[38] Ventsislav Petkov and Michael Gerndt. Integrating Parallel Application Development with Performance Analysis in Periscope. *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, April 2010.

[39] Philip C. Roth and Barton P. Miller. On-line automated performance diagnosis on thousands of processes. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'06)*, pp. 69–80. ACM, 2006.

[40] B.G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, pp. 216–226, May 1979.

[41] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems : An overview. *European journal of operational research*, Vol. 134, No. 3, pp. 664–676, November 2001.

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

[43] S. S. Shende and Allen D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 287–311, May 2006.

[44] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. *International Conference on Parallel Processing, 2004. ICPP 2004.*, Vol. 1, pp. 63–72, 2004.

[45] Nathan R. Tallent, Laksono Adhianto, and John.M. Mellor-Crummey. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, No. November, pp. 1–11. IEEE Computer Society, 2010.

[46] Nathan R. Tallent and John M. Mellor-Crummey. Identifying Performance Bottlenecks in Work-Stealing Computations. *Computer*, Vol. 44, No. 12, pp. 44–50, 2009.

[47] Kenjiro Taura. Gxp: An interactive shell for the grid environment. *IWIA2004*, Vol. 1, pp. 59–67, 2004.

[48] 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*, pp. 214–221, December 2010.

[49] Hong-Linh Truong and Thomas Fahringer. SCALEA: A performance analysis tool for distributed and parallel programs. In *Euro-Par 2002 Parallel Processing*, Vol. 2400, pp. 41–55. Springer, 2002.

[50] Guru Venkataramani, Christopher J. Hughes, Sanjeev Kumar, and Milos Prvulovic. Coherence Miss Classification For Performance Debugging in Multi-Core Processors. In *13th Workshop on Interaction between Compilers and Computer Architecture (Interact-13), held in conjunction with HPCA'09*. IEEE, 2009.

[51] Joachim Worringen. Experiment Management and Analysis with perfbase. In *2005 IEEE International Conference on Cluster Computing*, pp. 1–11. Ieee, September 2005.

[52] Omer. Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High Performance Computing Applications*, Vol. 13, No. 3, pp. 277–288, August 1999.

[53] 中島潤, 田浦健次朗. 高効率な I/O と軽量性を両立させるマルチスレッド処理系. 情報処理学会論文誌 プログラミング (PRO), Vol. 4, No. 1, pp. 13–26, 2011.

付 録A　ソースリスト

A.1　SQL

リスト A.1. MPI 向け性能情報データベースのテーブル定義 SQL リスト

```
CREATE TABLE funcmap (
    funcname TEXT,
    addr INTEGER,
    application TEXT,
    PRIMARY KEY (application, funcname)
);

CREATE TABLE profexec (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    profgroup_id INTEGER,
    exec_time DOUBLE PRECISION NOT NULL,
    display_time DOUBLE PRECISION,
    start_ts INT,
    UNIQUE (profgroup_id, start_ts),
    FOREIGN KEY (profgroup_id) REFERENCES profgroup(id) ON DELETE CASCADE
);

CREATE TABLE profgroup (
    application TEXT,
    app_viewname TEXT,
    nodes INTEGER,
    procs INTEGER,
    place TEXT,
    library TEXT,
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
    CONSTRAINT profgroup_unique UNIQUE (application, nodes, procs, place, library)
);

CREATE TABLE profile (
    funcname TEXT,
    calls INTEGER,
    subrs INTEGER,
    excl DOUBLE PRECISION,
    incl DOUBLE PRECISION,
    group_s TEXT,
    profexec_id INTEGER,
    profgroup_id INTEGER,
    rank INTEGER,
    PRIMARY KEY (profexec_id, profgroup_id, rank, funcname),
    FOREIGN KEY (profexec_id)  REFERENCES profexec(id) ON DELETE CASCADE,
    FOREIGN KEY (profgroup_id) REFERENCES profgroup(id) ON DELETE CASCADE
);
```

リスト A.2. ワークフロー向け性能情報データベースのテーブル定義 SQL リスト

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

CREATE SCHEMA IF NOT EXISTS `parpdb` DEFAULT CHARACTER SET utf8 ;
USE `parpdb` ;

-- Table `parpdb`.`workflow`
CREATE TABLE IF NOT EXISTS `parpdb`.`workflow` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `name` TEXT NULL DEFAULT NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- Table `parpdb`.`application`
CREATE TABLE IF NOT EXISTS `parpdb`.`application` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `workflow` INT UNSIGNED NOT NULL ,
  `name` TEXT NULL DEFAULT NULL ,
  PRIMARY KEY (`id`) ,
  CONSTRAINT `fk_application_workflow`
    FOREIGN KEY (`workflow` )
    REFERENCES `parpdb`.`workflow` (`id` )
    ON DELETE CASCADE
    ON UPDATE NO ACTION)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

CREATE INDEX `fk_application_workflow` ON `parpdb`.`application` (`workflow` ASC) ;

-- Table `parpdb`.`workflow_condition`
CREATE TABLE IF NOT EXISTS `parpdb`.`workflow_condition` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `worker_num` INT(10) UNSIGNED NULL DEFAULT NULL ,
  `filesystem` TEXT NULL ,
  `location` TEXT NULL ,
  `input_dataset` TEXT NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;

-- Table `parpdb`.`workflow_trial`
CREATE TABLE IF NOT EXISTS `parpdb`.`workflow_trial` (
  `id` INT NOT NULL AUTO_INCREMENT ,
  `workflow` INT UNSIGNED NOT NULL ,
  `workflow_condition` INT UNSIGNED NOT NULL ,
  `start_timestamp` DATETIME NULL ,
  `elapsed_time` TIME NULL ,
  `enabled` TINYINT(1) NOT NULL DEFAULT 1 ,
  PRIMARY KEY (`id`) ,
  CONSTRAINT `fk_workflow_trial_workflow1`
    FOREIGN KEY (`workflow` )
    REFERENCES `parpdb`.`workflow` (`id` )
    ON DELETE CASCADE
    ON UPDATE NO ACTION,
```

```
57    CONSTRAINT `fk_workflow_trial_condition1`
58      FOREIGN KEY (`workflow_condition`)
59      REFERENCES `parpdb`.`workflow_condition` (`id`)
60      ON DELETE CASCADE
61      ON UPDATE NO ACTION)
62  ENGINE = InnoDB
63  DEFAULT CHARACTER SET = utf8;
64
65  CREATE INDEX `fk_workflow_trial_workflow1`
66      ON `parpdb`.`workflow_trial` (`workflow` ASC);
67  CREATE INDEX `fk_workflow_trial_condition1`
68      ON `parpdb`.`workflow_trial` (`workflow_condition` ASC);
69  CREATE INDEX `workflow_reverse`
70      ON `parpdb`.`workflow_trial` (`workflow` ASC, `workflow_condition` ASC);
71
72  -- Table `parpdb`.`job`
73  CREATE TABLE IF NOT EXISTS `parpdb`.`job` (
74    `id` INT NOT NULL AUTO_INCREMENT COMMENT 'Job ID',
75    `workflow_trial` INT NOT NULL,
76    `application` INT UNSIGNED NOT NULL,
77    `workflow` INT UNSIGNED NOT NULL,
78    `work_idx` INT UNSIGNED NOT NULL,
79    `args` TEXT NULL COMMENT 'Command-line arguments',
80    `local_pid` INT NULL,
81    `worker` TEXT NULL COMMENT 'Workers name',
82    `elapsed_local` DOUBLE NULL,
83    `elapsed_remote` DOUBLE NULL,
84    `time_user` DOUBLE NULL,
85    `time_system` DOUBLE NULL,
86    `minor_faults` INT NULL,
87    `major_faults` INT NULL,
88    `local_start_time` DOUBLE NULL,
89    `remote_start_time` DOUBLE NULL,
90    PRIMARY KEY (`id`),
91    CONSTRAINT `fk_job_workflow_trial1`
92      FOREIGN KEY (`workflow_trial`)
93      REFERENCES `parpdb`.`workflow_trial` (`id`)
94      ON DELETE CASCADE
95      ON UPDATE NO ACTION,
96    CONSTRAINT `fk_job_application1`
97      FOREIGN KEY (`application`)
98      REFERENCES `parpdb`.`application` (`id`)
99      ON DELETE CASCADE
100     ON UPDATE NO ACTION,
101   CONSTRAINT `fk_job_workflow1`
102     FOREIGN KEY (`workflow`)
103     REFERENCES `parpdb`.`workflow` (`id`)
104     ON DELETE CASCADE
105     ON UPDATE NO ACTION)
106 ENGINE = InnoDB
107 DEFAULT CHARACTER SET = utf8;
108
109 CREATE INDEX `fk_job_workflow_trial1`
110     ON `parpdb`.`job` (`workflow_trial` ASC);
111 CREATE INDEX `fk_job_application1`
112     ON `parpdb`.`job` (`application` ASC);
113 CREATE INDEX `fk_job_workflow1`
```

```
114     ON `parpdb`.`job` (`workflow` ASC) ;
115
116 -- Table `parpdb`.`metric`
117 CREATE TABLE IF NOT EXISTS `parpdb`.`metric` (
118   `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
119   `application` INT UNSIGNED NOT NULL ,
120   `job` INT NOT NULL ,
121   `workflow_trial` INT NOT NULL ,
122   `name` TEXT NULL DEFAULT NULL ,
123   `value` DOUBLE NULL DEFAULT NULL ,
124   PRIMARY KEY (`id`) ,
125   CONSTRAINT `fk_metric_application1`
126     FOREIGN KEY (`application` )
127     REFERENCES `parpdb`.`application` (`id` )
128     ON DELETE CASCADE
129     ON UPDATE NO ACTION,
130   CONSTRAINT `fk_metric_job1`
131     FOREIGN KEY (`job` )
132     REFERENCES `parpdb`.`job` (`id` )
133     ON DELETE CASCADE
134     ON UPDATE NO ACTION)
135 ENGINE = InnoDB
136 DEFAULT CHARACTER SET = utf8;
137
138 CREATE INDEX `fk_metric_application1`
139     ON `parpdb`.`metric` (`application` ASC) ;
140 CREATE INDEX `fk_metric_job1`
141     ON `parpdb`.`metric` (`job` ASC) ;
142
143 -- Table `parpdb`.`wf_worker`
144 CREATE TABLE IF NOT EXISTS `parpdb`.`wf_worker` (
145   `index` INT NOT NULL COMMENT 'Worker index for a trial' ,
146   `workflow_trial_id` INT NOT NULL ,
147   `name` TEXT NOT NULL COMMENT 'Name of worker' ,
148   `ncpus` INT NOT NULL ,
149   `memory` INT NOT NULL ,
150   PRIMARY KEY (`index`, `workflow_trial_id`) ,
151   CONSTRAINT `fk_wf_worker_workflow_trial1`
152     FOREIGN KEY (`workflow_trial_id` )
153     REFERENCES `parpdb`.`workflow_trial` (`id` )
154     ON DELETE CASCADE
155     ON UPDATE NO ACTION)
156 ENGINE = InnoDB
157 DEFAULT CHARACTER SET = utf8;
158
159 CREATE INDEX `fk_wf_worker_workflow_trial1`
160     ON `parpdb`.`wf_worker` (`workflow_trial_id` ASC) ;
161
162 -- Table `parpdb`.`paratrac_runtime`
163 CREATE TABLE IF NOT EXISTS `parpdb`.`paratrac_runtime` (
164   `iid` INT NOT NULL AUTO_INCREMENT ,
165   `pid` INT NULL ,
166   `uid` INT NULL ,
167   `gid` INT NULL ,
168   `cmdline` TEXT NULL ,
169   `mountpoint` TEXT NULL ,
170   `hostname` TEXT NULL ,
```

```
171    `start_ts` DOUBLE NULL ,
172    `end_ts` DOUBLE NULL ,
173    `workflow_trial_id` INT NOT NULL ,
174    PRIMARY KEY (`iid`, `workflow_trial_id`) ,
175    CONSTRAINT `fk_paratrac_runtime_workflow_trial1`
176      FOREIGN KEY (`workflow_trial_id` )
177      REFERENCES `parpdb`.`workflow_trial` (`id` )
178      ON DELETE CASCADE
179      ON UPDATE NO ACTION)
180 ENGINE = InnoDB
181 DEFAULT CHARACTER SET = utf8;
182
183 CREATE INDEX `fk_paratrac_runtime_workflow_trial1`
184      ON `parpdb`.`paratrac_runtime` (`workflow_trial_id` ASC) ;
185
186 -- Table `parpdb`.`paratrac_syscall_name`
187 CREATE TABLE IF NOT EXISTS `parpdb`.`paratrac_syscall_name` (
188    `syscall_id` INT NOT NULL ,
189    `syscall_name` VARCHAR(45) NULL ,
190    PRIMARY KEY (`syscall_id`) )
191 ENGINE = InnoDB
192 DEFAULT CHARACTER SET = utf8;
193
194 -- Table `parpdb`.`paratrac_syscall`
195 CREATE TABLE IF NOT EXISTS `parpdb`.`paratrac_syscall` (
196    `stamp` DOUBLE NOT NULL ,
197    `sysc` INT NOT NULL ,
198    `pid` INT NOT NULL ,
199    `fid` INT NULL ,
200    `res` INT NULL ,
201    `elapsed` DOUBLE NULL ,
202    `aux1` INT NULL ,
203    `aux2` INT NULL ,
204    `paratrac_iid` INT NOT NULL ,
205    `workflow_trial_id` INT NOT NULL ,
206    `dummy_id` INT NOT NULL AUTO_INCREMENT ,
207    PRIMARY KEY (`dummy_id`, `paratrac_iid`, `sysc`, `stamp`, `workflow_trial_id`, `pid`) ,
208    CONSTRAINT `fk_syscall_paratrac_runtime1`
209      FOREIGN KEY (`paratrac_iid` , `workflow_trial_id` )
210      REFERENCES `parpdb`.`paratrac_runtime` (`iid` , `workflow_trial_id` )
211      ON DELETE CASCADE
212      ON UPDATE NO ACTION,
213    CONSTRAINT `fk_paratrac_syscall_syscall_name1`
214      FOREIGN KEY (`sysc` )
215      REFERENCES `parpdb`.`paratrac_syscall_name` (`syscall_id` )
216      ON DELETE CASCADE
217      ON UPDATE NO ACTION)
218 ENGINE = InnoDB
219 DEFAULT CHARACTER SET = utf8;
220
221 CREATE INDEX `fk_syscall_paratrac_runtime1`
222      ON `parpdb`.`paratrac_syscall` (`paratrac_iid` ASC, `workflow_trial_id` ASC) ;
223 CREATE INDEX `fk_paratrac_syscall_syscall_name1`
224      ON `parpdb`.`paratrac_syscall` (`sysc` ASC) ;
225
226 -- Table `parpdb`.`paratrac_file`
227 CREATE TABLE IF NOT EXISTS `parpdb`.`paratrac_file` (
```

```
228    `fid` INT NOT NULL ,
229    `path` TEXT NULL ,
230    `paratrac_iid` INT NOT NULL ,
231    `workflow_trial_id` INT NOT NULL ,
232    `dummy_id` INT NOT NULL AUTO_INCREMENT ,
233    PRIMARY KEY (`dummy_id`, `fid`, `paratrac_iid`, `workflow_trial_id`) ,
234    CONSTRAINT `fk_paratrac_file_paratrac_runtime1`
235      FOREIGN KEY (`paratrac_iid` , `workflow_trial_id` )
236      REFERENCES `parpdb`.`paratrac_runtime` (`iid` , `workflow_trial_id` )
237      ON DELETE CASCADE
238      ON UPDATE NO ACTION)
239 ENGINE = InnoDB
240 DEFAULT CHARACTER SET = utf8;
241
242 CREATE INDEX `fk_paratrac_file_paratrac_runtime1`
243      ON `parpdb`.`paratrac_file` (`paratrac_iid` ASC, `workflow_trial_id` ASC) ;
244
245 SET SQL_MODE=@OLD_SQL_MODE;
246 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
247 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

リスト A.3. MPI 向け性能情報データベースのビュー定義 SQL リスト

```
1  CREATE VIEW funcprofile AS
2  SELECT *
3  FROM profile
4  WHERE group_s = 'MPI' OR group_s = 'TAU_DEFAULT';
5
6  CREATE VIEW funcranksum AS
7  SELECT
8       funcname,
9       profexec_id,
10      SUM(excl) rank_excl_sum,
11      MAX(excl) rank_excl_max,
12      MIN(excl) rank_excl_min,
13      AVG(excl) rank_excl_avg,
14      SUM(incl) rank_incl_sum,
15      MAX(incl) rank_incl_max,
16      MIN(incl) rank_incl_min,
17      AVG(incl) rank_incl_avg
18 FROM funcprofile
19 GROUP BY funcname, profexec_id;
20
21 CREATE VIEW groupexecmerge
22 AS
23 SELECT
24      frs.funcname funcname,
25      pe.profgroup_id profgroup_id,
26      AVG(frs.rank_excl_sum) exec_excl_sum_avg,
27      AVG(frs.rank_excl_max) exec_excl_max_avg,
28      AVG(frs.rank_excl_min) exec_excl_min_avg,
29      AVG(frs.rank_excl_avg) exec_excl_avg_avg,
30      AVG(frs.rank_incl_sum) exec_incl_sum_avg,
31      AVG(frs.rank_incl_max) exec_incl_max_avg,
32      AVG(frs.rank_incl_min) exec_incl_min_avg,
33      AVG(frs.rank_incl_avg) exec_incl_avg_avg
34 FROM funcranksum frs, profexec pe
```

```
35 WHERE frs.profexec_id = pe.id
36 GROUP BY pe.profgroup_id, frs.funcname;
37
38 CREATE VIEW mpifuncprofile AS
39 SELECT *
40 FROM profile
41 WHERE funcname LIKE 'MPI_%';
42
43 CREATE VIEW pgroup_meta AS
44 SELECT pe.profgroup_id profgroup_id,
45        AVG(pe.exec_time) avg_time,
46        AVG(pe.display_time) avg_dtime,
47        pg.procs procs,
48        (sums.sum2 - sums.sum1 * sums.sum1 / sums.c ) / sums.c var,
49        (sums.sum_d2 - sums.sum_d1 * sums.sum_d1 / sums.c ) / sums.c dvar
50 FROM profexec pe,
51      profgroup pg,
52      (SELECT profgroup_id pgid,
53              SUM(exec_time * exec_time) sum2,
54              SUM(exec_time) sum1,
55              SUM(display_time * display_time) sum_d2,
56              SUM(display_time) sum_d1,
57              COUNT(*) c
58       FROM profexec
59       GROUP BY profgroup_id) sums
60 WHERE pe.profgroup_id = pg.id
61   AND pg.id = sums.pgid
62 GROUP BY profgroup_id, pg.procs;
63
64 CREATE VIEW pgroup_ratio AS
65 SELECT gem.funcname funcname,
66        gem.profgroup_id profgroup_id,
67        gem.exec_excl_avg_avg excl_pe_rank_avg,
68        gem.exec_incl_avg_avg incl_pe_rank_avg,
69        gem.exec_excl_max_avg excl_pe_rank_max_avg,
70        gem.exec_incl_max_avg incl_pe_rank_max_avg,
71        gem.exec_excl_min_avg excl_pe_rank_min_avg,
72        gem.exec_incl_min_avg incl_pe_rank_min_avg,
73        (gem.exec_excl_avg_avg / pgm.avg_time / 1000000) excl_avg_ratio,
74        (gem.exec_incl_avg_avg / pgm.avg_time / 1000000) incl_avg_ratio,
75        (gem.exec_excl_max_avg / pgm.avg_time / 1000000) excl_max_ratio,
76        (gem.exec_incl_max_avg / pgm.avg_time / 1000000) incl_max_ratio,
77        (gem.exec_excl_min_avg / pgm.avg_time / 1000000) excl_min_ratio,
78        (gem.exec_incl_min_avg / pgm.avg_time / 1000000) incl_min_ratio
79 FROM groupexecmerge gem,
80      pgroup_meta pgm
81 WHERE gem.profgroup_id = pgm.profgroup_id;
```

A.2 Shell Script

リスト A.4. ParaTrac を用いた GXP Make による JAWC の実行スクリプト

```
#!/bin/sh

cd /data0/t2g-ppc-all/kabe/workflow/
START=2
ITER=9
USE_FUSE=1
PARALLELISM=128
application=jawc
dataset=data2
datestr=`date +%Y-%m-%d`
if [ $USE_FUSE = 1 ]; then
    OUTDIR_PREFIX=$HOME/public_html/gpfs/${application}_\
pt_${dataset}_${PARALLELISM}_${datestr}_
else
    OUTDIR_PREFIX=$HOME/public_html/${application}_\
${dataset}_${PARALLELISM}_${datestr}_
fi
WF_DIR=/data0/t2g-ppc-all/kabe/workflow/${application}/solvers/gxp_make
FTRAC_PDIR=/tmp/kabe/workflow/
FTRAC_MOUNTPOINT=/tmp/kabe/workflow/fmpt
FTRAC_DIR=$HOME/hg/paratrac
FTRAC_OPT=-obig_writes
REAL_WF_DIR=$WF_DIR
if [ "x$USE_FUSE" = "x1" ]; then
    mkdir -p $FTRAC_MOUNTPOINT
    cd $FTRAC_PDIR
    # Relative path
    REAL_WF_DIR=$FTRAC_MOUNTPOINT/$REAL_WF_DIR
fi
# Environments
gxpc e mkdir -p $FTRAC_MOUNTPOINT
gxpc export "FTRAC_DIR=$FTRAC_DIR"
gxpc export "REAL_WF_DIR=$REAL_WF_DIR"
gxpc export 'PATH=$FTRAC_DIR/fs/fuse:$PATH'
gxpc export 'PATH=${REAL_WF_DIR}/../../modules/bin:$PATH'
gxpc export 'PATH=${REAL_WF_DIR}/../../modules/src/WWW2sf/tool/:$PATH'
# libraries
gxpc export 'PYTHONPATH=$HOME/jawc-extmods/BeautifulSoup-3.2.0:$PYTHONPATH'
gxpc export 'PERLLIB=$HOME/jawc-extmods/Unicode-Japanese-0.47/blib/lib:$PERLLIB'
gxpc export 'PERLLIB=$HOME/jawc-extmods/Text-Darts-0.09/blib/lib:$PERLLIB'
gxpc export 'LD_LIBRARY_PATH=$HOME/jawc-extmods/Text-Darts-0.09/\
blib/arch/auto/Text/Darts:$LD_LIBRARY_PATH'
for iter in `seq $START $(($START + $ITER - 1))`
do
    if [ "x$USE_FUSE" = "x1" ]; then
        # Ensure unmounted and mount
        gxpc e fusermount -u $FTRAC_MOUNTPOINT
        # Clear log directory
        ftrac_logdir=`/bin/ls -d ${FTRAC_PDIR}/ftrac-*`
        if [ -d $ftrac_logdir ]; then
            /bin/rm -rf $ftrac_logdir
        fi
        # Mount
```

```
54          gxpc cd $FTRAC_PDIR
55          gxpc e ftrac $FTRAC_MOUNTPOINT $FTRAC_OPT
56      fi
57      gxpc cd $REAL_WF_DIR
58      iterstr=`printf "%02d" $iter`
59      outdir=$OUTDIR_PREFIX$iterstr
60      cd $REAL_WF_DIR
61      gxpc make INPUT=$dataset -B all -j ${PARALLELISM} \
62          -- -a state_dir=$outdir
63      if [ "x$USE_FUSE" = "x1" ]; then
64          gxpc cd ~
65          cd ~
66          sync
67          sleep 5
68          gxpc e fusermount -u $FTRAC_MOUNTPOINT
69          ftrac_logdir=`/bin/ls -d ${FTRAC_PDIR}/ftrac-*`
70          /bin/rm -r $outdir/ftrac-*
71          sync
72          sleep 5
73          # Save ParaTrac Log directory
74          if [ -d $ftrac_logdir ]; then
75              gxpc e "rm -rf $outdir/"`hostname`
76              gxpc e "mv $ftrac_logdir $outdir/"`hostname`
77          fi
78      fi
79      cd $WF_DIR
80      gxpc cd $WF_DIR
81      rm -r out/$dataset
82  done
83  gxpc quit
```