情報処理学会論文誌　コンピューティングシステム　Vol. 4　No. 3　22–35 (May 2011)

HPC アプリケーションの最適化プロセスの自動化

村田浩樹[†1]　根岸康[†1]　森山孝男[†1]
グオジン コン[†2]　イシン チュン[†2]
フイファン ウェン[†2]　デヴィッド クレパッキ[†2]

High Performance Computing 分野向けのシステムは，性能向上のためにハードウェアの複雑さを増しており，既存のソフトウェアが想定しているハードウェア構成とのギャップが広がり続けている．このギャップを埋めることは難しく，言語，コンパイラ，そしてパフォーマンスツールの著しい進歩にもかかわらず，アプリケーションの最適化は多くが手作業として残され，通常専門家によって行われている．本論文では，専門家の知識，コンパイラ技術，そして，性能問題の解析と解決手段の発見といった性能に関する研究を連携させるためのフレームワークを構築し，専門家によって行われている最適化の作業を自動化する方法について述べる．このフレームワークでは，いったん構築された性能問題の解析と解決手段は，オープンで拡張可能なデータベースに蓄積され，再利用できる．HPC アプリケーションへの適用例を通じて，最適化の自動化の様子と，性能向上について述べる．

Automating Optimization Process of HPC Applications

Hiroki Murata,[†1] Yasushi Negishi,[†1]
Takao Moriyama,[†1] Guojing Cong,[†2] I-Hsin Chung,[†2]
Hui-Fang Wen[†2] and David Klepacki[†2]

The hardware of HPC system becomes more complex to improve its performance. The gap between its complexity and the hardware architecture assumed by software continues to spread. It is a challenge to bridge the gap. Despite significant progress in language, compiler, and performance tools, tuning an application remains largely a manual task, and needs to be performed mostly by experts. This paper explains how to automate the optimization being done by experts. We aims to build a framework that facilitates the combination of expert knowledge, compiler techniques, and performance research for performance diagnosis and solution discovery. With this framework, once a diagnosis and tuning strategy has been developed, it can be stored in an open and extensible database and reused. Automated optimization process and performance improvement are explained through applying this framework to HPC applications.

1. はじめに

High Performance Computing 分野向けのシステムは，性能向上のために，ハードウェアの複雑さが増しており，現在のスーパコンピュータの計算性能を引き出すには，その複雑さを考慮してアプリケーションを開発する必要がある．

米国国防高等研究計画局（US DARPA）は，生産性のギャップを埋める技術を開発する High productivity computing system initiative [1)] を後援している．生産性を向上させ，高性能を維持することは，計算機科学の多くの分野にわたる研究が必要となる．技術の著しい進歩にもかかわらず，複雑なアーキテクチャのマシン上に，アプリケーションを実装して高い性能を引き出すのは難しく，そのほとんどを専門知識を持つ技術者の手作業によっている．

一般的な最適化のサイクルは，以下のようなものである．アプリケーションの性能が見込みよりも低い場合，ユーザは挙動を観察し，仮説を立て，確認テストを実行する．まず，アプリケーションの性能データを収集し，データから簡単な仮説を立て，その仮説を改善もしくは検証するために性能データのトレースの観察を続ける．トレースは，しばしば巨大になり，扱いが難しくなるが，トレースが効率良く扱えたとしても，ユーザは，実行時の挙動と，プログラムの特徴を関連づけなくてはならない．アプリケーションとアーキテクチャ/システムの間に，ミスマッチが観察された場合，ユーザは，ソースプログラムに戻り，ミスマッチがどこで発生しているかを見つけ出す必要がある．この際，プログラムは，コンパイラによって変換されているため，コンパイラの振舞いをよく知っていることが必要となる．ミスマッチは，コンパイラ内の制約により，最適な変換が行われないことで発生したり，性能の低いアルゴリズムを実装したソースプログラムによっても発生したりする．いったん，性能上の問題の原因が特定されれば，ユーザは，最適化手法のレパートリの中から適切なも

†1 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research - Tokyo
†2 IBM T.J. ワトソン研究所
IBM T.J. Watson Research Center
This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No.HR0011-07-9-0002.

のを選び出し，適用する．適用する際には，その変換が，プログラムの制約を犯さないことを確かめるとともに，ポータビリティなどの望ましい特性を保護する必要がある．

このように，性能解析には，アルゴリズム，アーキテクチャ，コンパイラ，そして，実行時の挙動に対する深い知識が必要になる．また，性能データの収集，フィルタリング，検索，そして，解釈のための作業を要する．最適化では，さらに，システムの複数の構成要素の間のやりとりを調整する必要があり，性能解析と最適化は，経験者にとっても難しく，時間がかかる．本研究は，最適化プロセスの自動化をサポートするツールを提供することで，最適化の生産性を向上させることを目的としている．

ここで，関連する研究について簡単に触れる．最適化コンパイラは，最も重要な自動最適化ツールである．Profile guided compiler[2] は，静的情報と実行時情報をともに利用することで，プログラム中の最適化機会をより多く見つける．しかし，特別で強力な特殊操作は，コンパイラの制御下にない部分が多いため，コンパイラは，それらが多用されたプログラムをあまり最適化しない．そういった特殊操作の例としては，MPI 通信や，I/O 命令，そして，ライブラリなどがあげられる．また，コンパイラは，通常，アルゴリズムを変更することはなく，また，入力に依存するような最適化に重要な，応用分野に関する知識を持っていない．標準的なループ変換であっても，コンパイル時間の制限から最適なパラメータの組合せを探し出せないことがままある．

自動最適化ライブラリ[3),4)] は，与えられたアーキテクチャ構成に対して，適切なパラメータの組を探すなどして，それら自身を最適化できる．あらかじめ定義されたプログラムのセットについて，制限された変換で，最適化が行われるので，正確なモデリングとインテリジェントな検索が効果的に行える技術である．これらの技術単体では，一般的には，アプリケーションは最適化できない．

性能測定ツール[5)–8)] は，伝統的に，性能データの収集と表示を容易にしてきた．それらのツールは，専門家に，潜在的な性能問題の手がかりを与えることを目的としていて，一般的には，問題それ自身を明示することはない．また，性能データを整理するために多大な努力を必要とする．

本論文では，性能解析と最適化の自動化に向けての我々の取り組みについて述べる．我々は，専門家がアプリケーションを最適化する方法を観察するため，専門家と協力し，専門家の知識をマイニングすることで，手続きを自動化しようと試み，単にトレース情報を記録する代わりに，すでに知られている性能パターンを能動的に検索するフレームワークを開発した．このフレームワークは，新しい性能パターンを収容できるように，オープンかつ拡張可能に設計されている．また，性能解析のために，複数の性能測定ツールから，性能データを収集，比較，関連付ける機構も提供する．フレームワークは，性能上の問題に対してソリューションを提案，実装することで，ユーザが，性能上の問題を緩和する助けをする．我々の取り組みは，性能測定ツール，コンパイラ，そして，専門家の知識を，自動性能最適化のために統合しようとするものである[9)]．

本論文の構成は，以下のとおりである．2 章で，自動性能最適化ツールの開発の方針について述べる．3 章では，性能問題診断で，複数の性能測定ツールとコンパイラによる解析を合体させる方針について述べる．4 章では，ソリューションの自動提案について，5 章では，自動最適化の事例について，6 章では，最適化の効果について，7 章で，まとめと今後の課題について述べる．

2. 性能最適化の方針

我々のフレームワークは，アプリケーションのビルド，性能データの収集，ボトルネックの特定，ソリューションの発見と実装という最適化プロセスの各要素を自動化する（**図 1**）．ボトルネックの特定はボトルネックディスカバリーエンジンが，ソリューションの発見はソリューションディターミネーションエンジンが，ソリューションの実装はソリューションインプリメンテーションエンジンが行う．ヴィジュアライゼーションは，ユーザ・インタフェースであり，全体の制御と，性能データや，特定されたボトルネック，発見されたソリューショ

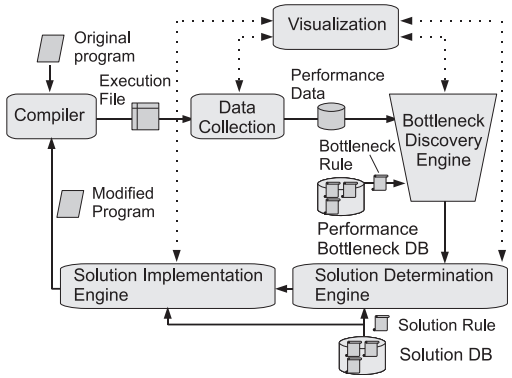


図 1　フレームワーク
Fig. 1　Framework.

ン，ソリューションの適用結果などをユーザに提示する．性能測定ツール，コンパイラ，そして，専門家の知識を統合し，性能データを比較，関連付ける方法を提供，ソリューションを提案，実装することで，ユーザの最適化作業の効率を向上させる．

生産性を向上させるためのサポートには，3 つのレベルがあると考えられる．第 1 に，静的解析，実行時の挙動，アルゴリズムの特性，アーキテクチャの特徴，そして専門家の知識といった広い範囲の情報への容易なアクセスの提供．第 2 に，そういった情報に基づいて，異なる側面（たとえば，計算，メモリ，通信，I/O）からの性能情報を比較，関連付けし，性能阻害要因を的確に提示するメカニズムの開発，提供．第 3 に，最終的な目標として，専門家がするように，自動的に最適化することである．これには，ソリューションの提案と，その実装が必要となる．現在（そして，近い将来），我々は，我々のフレームワークが知識を生み出すことはできないと考えており，このフレームワークは，専門家のある最適化作業をたどるように構成されている．専門家からできるだけ多くの知識を引き出しデータベース化することは，彼らを，繰返し作業から解放し，また，通常のユーザの生産性を大きく向上させると期待される．

我々の方法論は，以下のように要約される．まず，性能上の問題の原因を文献や専門家から収集し，それらを，性能の記述法に基づいたボトルネックルールとして，パフォーマンスボトルネックデータベースに保存する．ボトルネックディスカバリーエンジンは，アプリケーションの性能上の問題を詳しく調べるとともに，対応するボトルネックルールを発見するために，パフォーマンスボトルネックデータベースを検索する．いったん，ボトルネックルールが発見されると，関連するボトルネックが発見されたとして，ソリューションディターミネーションエンジンは，ソリューションデータベースに，適用可能なソリューションを問い合わせる．ソリューションは，ソリューションディターミネーションエンジンによって評価され，ユーザが必要と考えれば，最適化を実施する．本研究には，2 つの側面がある．1 つは，性能解析と最適化の自動化のための基盤のサポートである．これは，ルール以外の部分，すなわち，データ収集やヴィジュアライゼーション，ボトルネックディスカバリーエンジンやソリューションディターミネーションエンジン，ソリューションインプリメンテーションエンジンなど，ルールを実行するために必要なシステムに相当する．もう 1 つは，ボトルネックルール・ソリューションルール自身のサポートである．最適化に効果があり，適用範囲が広いルールをより多く提供することで，ツールキットをより有用なものにできる．我々は，フレームワークをオープンそして拡張可能にするとともに，共通ユーティリティを用意し，専門化がデータベースを拡張しやすくしている．

3. ボトルネックの発見

ボトルネックは，性能を律速するシステムの要素であり，最高性能を達成するには，膨大な組合せの最適化問題を解く必要がある．実際には，ボトルネックの発見は，アプリケーションやコンパイラ，ソフトウェアシステム，そして，アーキテクチャについての知識が必要で，伝統的に，専門家の小さなグループで実行されている．最適化プロセスを自動化するためには，そのような専門家の知識をマイニングする必要があるが，しばしば，知識があいまいな言葉で表現されるため，容易ではない．

我々のフレームワークにおいて，ボトルネックは，性能測定や静的解析から得られる性能情報（性能指標）の組を用いて定義されたルール（パターン）を用いて記述される．ルールは，小さな言語で表現され，現在のほとんどのルールは，論理式である．この言語の設計方針は，よく現れる問題のパターンをカバーするのに十分な柔軟性と表現力である．ルールの定義は，文献と専門家の知識から獲得され，ボトルネックについての知識が開発されるに従い，新しい言語機能が追加される．パフォーマンスボトルネックデータベースは，拡張とカスタマイズのためにオープンに設計されている．

我々の設計では，性能指標は，アプリケーションの性能もしくは，それに関係する数値である．例としては，与えられたループのパイプラインストール数や，プリフェッチ可能なストリーム数，あるプロセッサから送られたパケット数，物理メモリのサイズ，どのループがコンパイラによってタイリングされたか，などがあげられる．ボトルネックルールは，複数のソースと異なる側面からの性能指標を比較，関連付けする手段を提供し，問題を正しく診断する助けとなる．

3.1 既存の性能測定ツールの性能指標

既存の性能測定ツールは，特定の性能指標を収集することに特化している．これらの性能指標は，我々のツールが提供する性能指標を取り込むための機構を通じて，意味のあるボトルネックルールを構築するために使うことができる．

我々の現在の実装では，フレームワークとして，IBM®high performance computing toolkit（IHPCT）[10] を利用して，多くの性能指標を収集する．IHPCT は，プロファイリングツール[11]，hardware performance monitor（HPM），simulation guided memory analyzer（SiGMA）[12]，MPI プロファイリング，トレーシング・ツール，OpenMP トレーシング・ツール，modular I/O ツールから構成される．これらの要素は，それぞれ，アプリケーションのある性能の側面を評価もしくは測定し，収集された性能データは，ボトルネッ

表 1　既存の性能分析プログラムから収集された性能指標の例
Table 1　Examples of metrics collected from performance analysis programs.

性能指標	説明	性能取得ツール
PM_INST_CMPL	実行命令数	HPM
L1_miss_rate	L1 キャッシュミス率	HPM
Avg_msg_size	平均メッセージ長	MPI profiler
Thread_imbalance	スレッド間負荷アンバランス	Open MP profiler
#prefetch	プリフェッチキャッシュライン数	SiGMA
mpi_latesender	受信プロセスのメッセージ待ち時間	Scalasca

クを定義するための性能指標として使われる．たとえば，HPM は，それ自身で，各種ハードウェアイベントについて，数百の性能指標を収集する．

異なるツールが取得した性能指標は共存できる．たとえば，我々は，TAU [5] と Scalasca [13] から性能指標を収集する実験をしている．**表 1** は，既存のツールによって収集した性能指標の例である．

収集した性能指標を用いてルールを定義することで，複数のツールの分析を組み合わせることができる．たとえば，以下のルールは，ループ内の時間のかかる除算演算子が，パイプラインストール問題を引き起こしている可能性があることを示す．

$$\#divides > 0 \;\&\&\; \frac{PM\_STALL\_FPU}{PM\_RUN\_CYC} > t \&\& \; vectorized = 0$$

ここで，#devides は，静的解析モジュールで収集したループ内の除算演算子の数を示す性能指標．PM_STALL_FPU と PM_RUN_CYC は，HPM によって収集された性能指標で，それぞれ，浮動小数点演算ユニットのストールしたサイクル数と総実行サイクル数である．t は，定数の閾値である．そして，vectorized は，コンパイラによって，浮動小数点演算がベクタライズされたかどうかを示す性能指標で，値 0 は，ベクタライズされていないことを示す．ただし，除算演算子の数が静的に解析できない場合（DO ループや IF などで実行時に除算の回数が変化する場合は正しい評価を行う保障はない）のように，ボトルネックルールに含まれる性能指標の値が得られない場合，フレームワークは，ルールが適用できないという判定を下し，他のルールを適用する．

3.2　コンパイラからの性能指標

性能上の問題を正確に発見するためには，プログラムの構造を理解する必要があるため，しばしば，静的解析が必要となる．コンパイラは，内部で静的解析を行っており，この解析結果は，ボトルネックの発見にも有用である．しかし，たいていのコンパイラは，通常，外部のツールに解析結果を提供する機能を持たない．また，コンパイラ自身も複雑なので，コンパイラを修正してこの機能をサポートすることも困難である．現在，我々は，IBM コンパイラグループと協力し，標準コンパイラの分析データの利用を試みている．

コンパイラによって提供される性能指標としては，プリフェッチストリーム数の概算や，パイプラインストールの見積り，基本ブロックの数などがあり，これらは，ボトルネックルールを構成するのに利用される．さらに重要なこととしては，性能上の問題の多くが，理論的には適用可能だが，コンパイラによって適用されなかった最適化に関連しているため，最適化についての詳細な情報は非常に有用である．ホットスポットにコンパイラが適用した最適化が何であるかを知らずに，意味のある解析を示すことは，事実上不可能である．特に，数値計算主体のホットスポットでは生成された命令列の解析が重要となるので，コンパイラからの情報が重要である．

以下，ループアンロールの解析を例として取り上げる．ループアンロールアンドジャムは，よく研究されているコンパイラテクニックであり，実際，多くのコンパイラは，アンロール変換を行うコンポーネントを持っている．しかし，業界標準のコンパイラであっても，アウタループアンローリングによる最適化はうまくいかないことが多い．理由としては，非常に高い最適化レベルでしかアウタループアンローリングが実行されない，コンパイラがアンローリングを不可能にするような他の形の変換が適当と判断するなどがあげられる．あるアプリケーションのコンパイル後に解析を行い，いくつかのコードの領域に最適化を行ったところ，アンロールベクタのコストと利益を，より正確に見積もるための詳細な分析をすることができ，実験の結果，業界標準のコンパイラよりも高速なコードを生成することが確認できた[14]．この例では，我々のモジュールによって見積もられたパラメータと，コンパイラによって提示されたパラメータの間に食い違いがあり，それが性能ボトルネックになっていた．コンパイラが，性能向上が見込まれる最適化を適用しなかった理由を出力し，それを利用することによりアプリケーションをさらに最適化することが可能になる．

ボトルネックの発見のために，コンパイラの分析結果を XML 形式で出力したデータを利用する．このデータから，いくつかの性能指標と特定のコード領域への最適化の適用結果が得られる．

4.　ソリューションの構成と実装

我々のフレームワークは，性能上の問題を緩和する手段（ソリューション）をユーザに提案する．ソリューションの候補は，ソリューションデータベースにソリューションルールと

して保存される．ソリューションルールは，一般化された形式になっているので，アプリケーションに合わせて適用する必要がある．たとえば，ブロッキング MPI コールで，過大な時間がかかっているようなボトルネックのための一般化されたソリューションは，通信と計算をオーバラップさせる一方，オーバラップができるかできないか，どのくらいできるかは，アプリケーションに依存する．ソリューションの適用は，以下の 3 つの手順，すなわち，

(1)　データ依存性を維持するための，ルール適用の正当性のチェック

(2)　ルールのパラメータの計算（たとえば MPI プログラムのための通信と計算のオーバラップでは，ノンブロッキングコールと，それらの位置がパラメータである．次に，ソリューションを適用したアプリケーションをモデル化するか，実行することで，性能向上が見積もられる．）

(3)　コードの変更と環境設定の決定

で実施される．

パラメータの値は，ソリューションの効果に，大きく影響するので最適なパラメータの値を見つけることは重要である．これは，CPU に関連する問題について最適化コンパイラによってなされる分析と同様である．我々のフレームワークの最適化プロセスではコンパイラほど時間的に切迫していないので，性能評価において，より最適なパラメータを探し出すことが可能である．

たとえば，ループアンロールをソリューションとする場合，アンロールベクトル（アンロール回数）がパラメータとなる．最適なアンロールベクトルを決定するために，候補となる各アンロールベクトルに対して標準命令セット（universal machine model）に基づいた命令を生成する．生成の際レジスタ数は対象プロセッサに一致させ，対象プロセッサのプロセッサ内のロードストアユニットや整数演算装置，浮動小数点演算装置について，同時動作可能なものを考慮して，ループ本体で費やされるサイクル数を数える．すべてのアンロールベクトルの候補についてサイクル数を計算し，最小のサイクル数となるアンロールベクトルを最適な候補として選択する．非常に単純な例として，たとえばループ本体が $s = s + a(i) * b(i)$ の場合を考えると，標準命令セットは以下のようになる．

```
load R2, a(R4)
load R3, b(R5)
fma R1, R2, R3
```

1 番目の load で a(i) をレジスタ R2 にロード，2 番目の load で b(i) をレジスタ R3 にロードし，3 番目の fma で R2 と R3 の積をとり R1 に加える．この場合のサイクル数を，対象プロセッサを IBM®POWER5™ プロセッサとして計算してみる．POWER5™ プロセッサは浮動小数点積和演算の実行に 6 サイクルかかる．ロードは，キャッシュヒットしている場合 1 サイクルで 2 ストリーム実行でき，演算と同時に実行できるため，ループアンロールしない場合，この命令セットの実行には 6 サイクルかかるが，積和演算はパイプライン化されているので，ループアンロールすることで，実質的に 1 サイクルで実行できるようになる．この場合の最適なアンロールベクトルは 6 である．

ソリューションの有効性は，発見されたボトルネックの正確さと密接に関係している．ボトルネックの記述が詳細であるほど，システムは，ソリューションを提案しやすくなる．たとえば，フレームワークが，あるコード領域で，過大なパイプラインストールが発生するというボトルネックパターンを検出したとする．この現象は，非常に多くの理由で起きるので，詳細な情報がないと，意味のあるソリューションを提案することは難しい．もし，ストールの大部分がデータキャッシュミスによるものであることが検出されると，データ局所性の向上がそのソリューションとなる．もし，静的解析で，特定の配列への不規則なアクセスが発生していることが明らかになれば，ソリューション発見は，その配列に集中することができる．

ソリューションには次の 3 種類の実装がある：コンパイラへの変換の指示，ソースコード書き換え，そしてユーザへの最適化方針の提案である．コンパイラへの変換の指示では実際にソースコードを変更する必要がないのでフレームワークは，より良いパラメータ値の検索に集中し，実際の変更については，コンパイラに任せることができる．コンパイラに変換を指示する 2 つの異なる機構を開発するために，我々は IBM コンパイラグループと協業している．1 つは，コンパイラディレクティブを通じて，もう 1 つは，コンパイラに特別に設けたフレームワークとの間のチャネルを通じて行うものである．ディレクティブは，コンパイラへの示唆として働き，チャネルは，ツールが，必要な最適化を，標準的な変換から構成，実装するための柔軟なインタフェースを提供する．ソースコード書き換えの形のソリューションは，変換にコンパイラサポートがないような場合に必要となる，たとえば，MPI 通信への最適化があげられる．5 章では，ソリューションを実装するために，コンパイラディレクティブとソースコード書き換えを採用する例を示す．

5. 最適化例

現在，我々のフレームワークは，ボトルネック検出のための多くの組み込み性能指標とルールを含んでいる．また，ハードウェアイベントカウンタによって集められた非常に多く

の性能指標と，静的解析とコンパイラ解析によって集められた性能指標がある．ソリューションについては，フレームワークは，いくつかの性能問題を自動で最適化できる．HPC アプリケーションは，通常，計算，通信，そして I/O を含んでいて，それぞれに特有の問題のパターンがあり，その解析には特有の性能指標が必要となる．ここで，フレームワークを使って，それぞれの問題についてアプリケーションを解析，最適化する例を示す．

5.1　計算に関する最適化例

ここで，検討するアプリケーションは，Lattice Boltzmann Magneto-Hydrodynamics（LBMHD）[15] である．Lattice Boltzmann（格子ボルツマン）法は，流体を衝突と移動を繰り返す粒子の集合体ととらえ，粒子の動きを計算することで流体の運動を解析する数値計算法で，LBMHD は，電磁流体の挙動をシミュレーションする．

フレームワークは，まず LBMHD を実行し性能評価ツールを用いてホットスポットを検出する．この例では一番時間がかかっているのは collision という関数で，2 番目は stream という関数である．collision では，多くのパイプラインストールが発生していて，システムの資源が十分に活用されていないことが明らかになる．最適化の専門家であれば，この結果から，collision 内のホットスポットでは多次元配列へのアクセス順序がストレージでの順序と一致していないため多くのパイプラインストールが発生しているのに対し，同じ配列への stream でのアクセス順序がストレージでの順序と一致していて，パイプラインストールの発生が少ないことから，アルゴリズムをプログラム化する際に同じ多次元配列へのアクセスの順序を関数によって異なるように実装したことが，性能上の問題を引き起こしていることを発見できる．

図 2 は，collision の中のホットスポットである．多次元配列 f，g，feq，geq のアクセス順序は，j，i，k の繰返しの順序から定まっていて，それらのストレージでの順序と一致しておらず，大量のパイプラインストールが発生する．この繰返しの順序とストレージの順序のミスマッチを検出するボトルネックルールは，以下のようになる．

$$STALL\_LSU/PM\_CYC > \alpha \; and \; STRIDE1\_RATE \leq \beta$$
$$and \; REGULAR\_RATE(n) > STRIDE1\_RATE + \gamma$$

STALL_LSU と PM_CYC は，それぞれ，ロードストアユニットでストールされているサイクル数と，総サイクル数である．STRIDE1_RATE は，ループ内のメモリアクセスが stride-1 である率の静的解析による見積り．REGULAR_RATE は，ループ内のメモリアクセスが規則的（stride-n）である率の見積りである．ここで，stride-1 は，配列へのアクセス順序がストレージでの順序と一致していて，連続したアクセスがストレージ上の連続した領域に行われることを示す．たとえば，Fortran では配列の 1 次元目の添え字が 1 ずつ変化する場合にストレージ内で連続する位置に配置されるので，1 次元目の添え字が 1 ずつ変化するようにアクセスされる場合，stride-1 となる．また，stride-n は，配列への連続したアクセスがストレージ上で連続せず，一定の間隔を置いたアクセスとなることをいう．たとえ

```
do j = jsta, jend
   do i = ista, iend
   ...
      do k = 1, 4
         vt1 = vt1 + c(k,1)*f(i,j,k) &
               + c(k+4,1)*f(i,j,k+4) &
         vt2 = vt2 + c(k,2)*f(i,j,k)
               + c(k+4,2)*f(i,j,k+4)
         Bt1 = Bt1 + g(i,j,k,1) + g(i,j,k+4,1)
         Bt2 = Bt2 + g(i,j,k,2) + g(i,j,k+4,2)
      enddo
      ...
      do k = 1, 8
         ...
         feq(i,j,k)=vfac*f(i,j,k)+vtauinv &
             *(temp1+trho*.25*vdotc+ &
            .5*(trho*vdotc**2- Bdotc**2))
         geq(i,j,k,1)= Bfac*g(i,j,k,1)+ &
             Btauinv*.125*(theta*Bt1+ &
            2.0*Bt1*vdotc- 2.0*vt1*Bdotc)
         ...
      enddo
      ...
   enddo
enddo
```

図 2　collision のホットスポット
Fig. 2　Hotspot in collision.

ば，Fortran では，配列の 1 次元目ではなく，2 次元目以降の添え字が 1 ずつ変化するようにアクセスされるような場合に stride-n となる．α，β，γ は，閾値として用いられる定数である．異なる閾値は，性能問題の深刻さのレベルが異なることを示す．また，ボトルネックの除去によりユーザが期待する性能向上の最小値（これ以下なら最適化はしない）にも影響を受ける．

ループ内の配列のアクセス順序とストレージの順序を一致させるソリューションルールのアルゴリズムは，**図 3** のようになる．このアルゴリズムを，図 2 の collision のホットスポットに適用すると，2 つの inner most loop 内に現れる配列 f，feq，g，geq のアクセス順序とストレージの順序が一致していないことが分かり，それらを一致させるようにプログラムを書き換えようとするが，ループ交換が適用できないため，配列の添え字の順序を変更することになる．ただし，添え字の順序の変更は，影響を受ける配列へのすべてのアクセスの性能に影響し，また，ソースコードへの変更量（インパクト）が大きくなるので，ソリューションの実装には，配慮が必要となる．

配列 f，feq では，制御変数は，ネストの内側から k，i，j の順である．Fortran では，配列の 1 次元目の添え字が 1 ずつ変化する場合に，ストレージ内で連続する位置に配置するので，新しい配置は，k で指定されている 3 次元目を 1 次元目に，i で指定されている 1 次元目を 2 次元目に，そして，j で指定されている 2 次元目を 3 次元目に変換する．これを，$(3,1,2)$ と示すことにする．つまり，1 桁目の 3 が，3 次元目を 1 次元目にすることを示し，2 桁目の 1 が，1 次元目を 2 次元目に，3 桁目の 2 が，2 次元目を 3 次元目にすることを示す．配列 g，geq では，4 次元目が，最も内側のループの中で連続的に変化するので，新しい配置では，もとの 4 次元目を 1 次元目とする．そして，k，i，j で指定される添え字については，f，feq のものと同じである．これは，$(4,3,1,2)$ と示される．この新しいストレージの順序を実装する方法として，配列の宣言と，配列への参照をすべて書き換える方法があるが，我々は，コード変更量を抑えるために，コンパイラのディレクティブを使用した．たとえば，IBM®の XLF コンパイラは，$!IBM\ SUBSCRIPTORDER$ ディレクティブを提供しており，これによって，ソースコード上の添え字の順序を変えずに，異なるストレージ順序に変更することができる．LBMHD については，以下のディレクティブで，4 つの配列のストレージ順序が変更される．

$!IBM\ SUBSCRIPTORDER(f(3,1,2),$
$feq(3,1,2), g(4,3,1,2), geq(4,3,1,2))$

配列のストレージの順序を変えると，プログラムの他の部分に，望ましくない副作用を生じることがある．LBMHD では，配列 f，feq，g，geq は，いくつかの関数で共有されている．ストレージ順序の変更により，collision では，これらの配列のストレージ上の順序が適切になった．

しかし，ストレージ順序の変更により，stream のホットスポット（**図 4**）に，新たな問題が生じる．stream での配列 g，geq へのアクセスは，元々のストレージ順序では問題なかったが，collision の最適化のために変更されてしまった．これは collision と同じボトル

```
1:  for ホットスポット内の各 inner most loop について do 2:から 10:を実行する
 2:  for inner most loop 内の各配列について do 3:から 9:を実行する
  3:  if loop のネストから決まる配列のアクセス順序とストレージの順序が一致していない then
   4:  if ループ交換が適用できる then
    5:  アクセス順序とストレージの順序を一致させるループ交換をソリューションの候補として記録
   6:  else
    7:  アクセス順序とストレージの順序を一致させる配列の形の変更をソリューションの候補として記録
   8:  endif
  9:  endif
 10: enddo
11: enddo
12: 一番多くの配列のアクセス順序とストレージの順序を一致させるソリューションを選び出し適用する
```

図 3　ループ内の配列のアクセスの順序とストレージの順序を一致させるアルゴリズム
Fig. 3　Algorithm to achieve an alignment of array access sequence and storage sequence in loop.

```
do k = 1, 2
   do j = jsta, jend
      do i = ista, iend
         g(i,j,1,k)= geq(i-1,j,1,k)
         ....
      enddo
   enddo
   do j = jsta, jend
      do i = ista, iend
         g(i,j,2,k)= w1*geq(i,j,2,k) &
            + w2*geq(i-1,j-1,2,k) &
            + w3*geq(i-2,j-2,2,k)
         g(i,j,4,k)= w1*geq(i,j,4,k) &
            + w2*geq(i+1,j-1,4,k) &
            + w3*geq(i+1,j-2,4,k)
         ...
      enddo
   enddo
enddo
```

図 4　stream のホットスポット
Fig. 4　Hotspot in stream.

ネックルールで検出できる．これを矯正するためのソリューションルールのアルゴリズムは，**図 5** のようになる．

このアルゴリズムを，図 4 の stream のホットスポットに適用すると，配列 g，geq へのアクセス順序とストレージの順序が一致しておらず，アクセス順序とストレージの順序を一致させる必要があることが分かる．ここで，ループ交換が必要となるが，stream のホットスポットのループは，ループ交換を適用できる．すべてのネストしたループは，2 つの perfectly nested ループに分割でき，2 つのループを互いに交換できる．この 2 つのループは，ループ融合でき，結果として，最外ループ（do k = 1, 2）は，一番内側に移され，最内ループ（do i = ista, iend）は，中間に，（do j = jsta, jend）のループは，一番外側に移される．

```
1:  if 配列の形の変更がソリューションとして適用
    された then
2:  配列の形の変更でアクセス順序とストレージ順
    序を一致させたホットスポットを H とする
3:  形を変えた配列を Ai とする
4:  for プログラム内の H に含まれない
    各 inner most loop について
    do 5:から 11:を実行する
5:  for inner most loop 内の各 Ai について
    do 6:から 10:を実行する
6:  if loop のネストから決まる配列のアクセス
    順序とストレージの順序が一致していない
    then
7:  if ループ交換が適用できる then
8:  アクセス順序とストレージの順序を一致さ
    せるループ交換をソリューションの候補と
    して記録
9:  endif
10: endif
11: enddo
12: enddo
13: 一番多くの配列のアクセス順序とストレージの
    順序を一致させるソリューションを選び出し適用
    する
14: endif
```

図 5　配列の形の変更による弊害を減らすアルゴリズム
Fig. 5　Algorithm to eliminate a harmful effect of array form change.

以上，述べてきた collision，stream に対するソリューションルールは，図 3 と図 5 のアルゴリズムに基づいて，C 言語で実装されており，文字列もしくは行を単位とするソースコードに対する書換の指示を生成する．ソリューションインプリメンテーションエンジンは，これらのソリューションルールを起動し，生成された書換の指示に従ってソースコード

を書き換える．

このようなボトルネックルールとソリューションにより，我々のフレームワークは，LBMHD の最適化を自動化できる．他のアプリケーションについても，同様の性能問題を検出し，緩和できる．

5.2　通信に関する最適化例

CPU が通信の完了を待つと，システム資源に待ち時間が生じる．この問題は，非ブロッキング通信を使ってプログラムを書き直し，通信を計算とオーバラップさせることで，解消できることが多い．これは，I/O についてもあてはまり，同期 I/O 命令が使われていた場合，非同期 I/O 命令を使ってプログラムを書きなおすことで，I/O と計算がオーバラップされ，性能が向上する場合がある．このパターンの問題は，対象となる科学の分野の専門家によって書かれた科学計算プログラムに多く見られる．

たとえば，ブロッキング通信呼び出しは，本来存在しない直線的な依存パターン（たとえば，各プロセスが，その左隣からデータを受け取り，右隣にデータを送るような場合）を形成する場合があり，性能を大きく低下させる．この問題を，MPI の場合について検出するボトルネックルールは，以下のようになる．

$$\frac{ElapsedTime - \frac{PM\_CYC}{CPUFrequency}}{ElapsedTime} > \alpha \textbf{ and}$$
$$mpi\_hotspot = 1 \textbf{ and}$$
$$\#blocking\_mpicalls > 0$$

ElapsedTime は，ホットスポットの実行にかかる時間を示す．PM_CYC は，ホットスポットに費やされる CPU サイクル数であり，CPUFrequency は，プロセッサのクロック周波数である．mpi_hotspot は，ホットスポットが，MPI のホットスポットである場合に，1 となる性能指標であり，#blocking_mpicalls は，ホットスポット内のブロッキング MPI 呼び出しの数でを示す．このルールは，MPI ホットスポットで，CPU のアイドル時間が多く，かつ，ブロッキング MPI 呼び出しがあることから，そのホットスポットが，ボトルネックである可能性があることを示している．このボトルネックは，ブロッキング MPI 呼び出しを非ブロッキングのものに置き換えることで解消できる場合がある．

ソリューションは，計算と通信をオーバラップさせることである．一般的には，ブロッキング通信呼び出しを非ブロッキング通信呼び出しで置き換えると，プログラムの挙動が変わってしまい，実行結果の正しさが保障されない．計算と通信，もしくは，通信と通信のオーバラップの量は，MPI の呼び出しの位置によって決まるので，呼び出しの位置が，性能の最大化に重大な影響を持つ．呼び出しの位置を決める際の方針は，通信の開始をできるだけ早くし，終了待ちをできるだけ遅くすることである．その結果，通信が行われている間に，より多くの計算を実行できるようになる．ソリューションルールのアルゴリズムは，**図 6** のようになる．

1 つのブロッキング通信呼び出しは，3 つの呼び出しで置き換えられる．対応する非ブロッキング通信呼び出しと，wait（通信完了待ち），そして，それらの対応付けに必要な変数の宣言である．このアルゴリズムは，ブロッキング MPI 呼び出しの入力変数と出力変数の依存関係を考慮して，プログラムの挙動を変化させずに，性能を最大化するように，非ブロッキング MPI 呼び出しへの置き換えを行う．

このソリューションルールは，図 6 のアルゴリズムに基づいて，C 言語で実装されており，フレームワークに含まれる MPI をサポートした静的解析機能と書換機能を利用している．これらの機能は，新しいソリューションを構築する際にも利用できる．

このボトルネックパターンは，LBMHD にも含まれており，output サブルーチンに現れる．**図 7** の左側は，元の通信パターンを示している．通信は隣接プロセッサ間の糊代の部分を交換し，準備されたデータは，MPI_SEND と MPI_RECV のブロッキングプリミティブを使って転送され，その間，プロセッサは，アイドル状態になる．このコードでは，tempB1s, tempB2s，tempV1s，tempV2s が送信バッファであり，tempB1r，tempB2r，tempV1r, tempV2r が受信バッファである．

右の最適化コードでは，元のブロッキングプリミティブが，対応する非ブロッキングプリミティブに置き換えられ，ブロッキングプリミティブ（MPI_WAIT）が，データ依存性を維持するために挿入されている．これによって，通信と計算が，データ依存性を破ることなくオーバラップされる．また，この場合には，通信プリミティブの間でもオーバラップが起きる．

5.3　I/O に関する最適化例

同期 I/O の終了を待つ間に CPU がアイドルするボトルネックのほかに，よく発生する I/O ボトルネックとしては，Fortran プログラムで，I/O の準備にかかるオーバヘッドが，ループの中で繰り返されて，ボトルネックとなるものがある．write 文のみが，繰返し回数の多いループの中に置かれた場合，バッファアロケーションなどのオーバヘッドが，CPU タイムを浪費する場合がある．このボトルネックは，中間結果を定期的にファイルに出力するようなアプリケーションでよく見られる．この問題を検出するボトルネックルールは，以下のようになる．

1: for ホットスポット内の，各 MPI 呼び出しについて do 2:から 20:を実行する
2: 入力変数と出力変数のリストを生成する．
3: for ホットスポット内で，MPI 呼び出しの前にある文について，MPI 呼び出し文から前に向かって do 4:から 8:を実行する
4: if 入力変数群を変更しない then
5: 入力変数群が変更されない最初の位置 (E) の候補として記録
6: else
7: この for loop を抜ける
8: endif
9: enddo
10: for ホットスポット内で，MPI 呼び出しの後にある文について，MPI 呼び出し文から後ろに向かって do 11:から 15:を実行する
11: if 出力変数群を参照しない then
12: 出力変数群を参照しない最後の位置 (L) の候補として記録
13: else
14: この for loop を抜ける
15: endif
16: enddo
17: MPI 呼び出しを取り除く
18: 位置 E の文の前に，対応する非ブロッキング MPI 呼び出しを挿入する
19: 位置 L の文の後ろに，wait を挿入する
20: 新たに必要となる変数を宣言する
21: enddo

図 6　計算と通信をオーバラップさせるソリューションルールのアルゴリズム
Fig. 6　Algorithm of the solution rule to overlap calculation and communication.

```
!compute tempB1s,tempB2s, tempV1s, tempV2s
...
CALL MPI_SEND(tempB1s(jsta),...,inext,...)
CALL MPI_RECV(tempB1r(jsta),...,iprev,...)
CALL MPI_SEND(tempB2s(jsta),...,iprev,...)
CALL MPI_RECV(tempB2r(jsta),...,inext,...)

CALL MPI_SEND(tempV1s(jsta),...,inext,...)
CALL MPI_RECV(tempV1r(jsta),...,iprev,...)
CALL MPI_SEND(tempV2s(jsta),...,iprev,...)
CALL MPI_RECV(tempV2r(jsta),...,inext,...)

!use tempB1r,tempB2r, tempV1r, tempV2r
...
```

```
call MPI_IRECV(tempB1r(jsta), ..., iprev,...)
call MPI_IRECV(tempB2r(jsta), ..., inext,...)
call MPI_IRECV(tempV1r(jsta), ..., iprev,...)
call MPI_IRECV(tempV2r(jsta), ..., inext,...)

! compute tempB1s,tempB2s, tempV1s, tempV2s
...

call MPI_ISEND(tempB1s(jsta), ..., inext,...)
call MPI_ISEND(tempB2s(jsta), ..., iprev,...)
call MPI_ISEND(tempV1s(jsta), ..., inext,...)
call MPI_ISEND(tempV2s(jsta), ..., iprev,...)

call MPI_WAIT(NEW1_1, istatus, ierr)
call MPI_WAIT(NEW3_1, istatus, ierr)
call MPI_WAIT(NEW5_1, istatus, ierr)
call MPI_WAIT(NEW7_1, istatus, ierr)

!use tempB1r,tempB2r, tempV1r, tempV2r
...
```

図 7　元の通信コードと最適化されたコード
Fig. 7　Original and optimized communications.

$$is\_Fortran \textbf{ and } \frac{write.t}{ElapsedTime} > \alpha \textbf{ and } \frac{write.prep.n}{write.io.n} > \beta$$

is_Fortran は，Fortran プログラムであるかを示す性能指標であり，write.t は，write 文の実行にかかる時間，ElapsedTime は，プログラムの実行にかかる時間，write.prep.n は，write の準備関数を呼び出す回数，write.io.n は，write の出力関数を呼び出す回数で，α と β は，閾値として用いられる定数である．このボトルネックルールは，I/O の準備にかかるオーバヘッドが大きいことによるボトルネックがある可能性を示す．このボトルネックに対するソリューションは，write 文を含む do 文を write 文の implied-do に変更することである．write 文の implied-do を使うと，write の準備関数は，すべての出力関数呼び出しに対して 1 回だけ呼び出されるようになる．ただし，この書き換えがプログラムの挙動に影響を与えないように，出力する変数が，ループ内で不変であるなどの，データ依存解析をする必要がある．ソリューションルールのアルゴリズムは，**図 8** のようになる．

このアルゴリズムは，write 文を含む do 文を，データ依存性を考慮して，write 文の implied-do に変更して，I/O の準備にかかるオーバヘッドを削減し，ボトルネックを解消す

1:　for ホットスポット内の各 write 文について do 2:から 5:を実行する
2:　if 出力する変数群が，ループ内で不変である then
3:　write 文を含む do 文を implied-do に変更した write 文を作成し，ループの後に挿入
4:　write 文を取り除く
5:　endif
6:　enddo
7:　ループ内に文が残っていなければ，ループを取り除く

図 8　write 文を含む do 文を implied-do に変更するリューションルールのアルゴリズム
Fig. 8　Algorithm of the solution rule to change nested loops including write statement to implied-do.

```
...
do j = jmin, jmax
  do i = imin, imax
     write(myrank+5000,*) 'CURR(',i,j,')=', &
     (-B(i,j+1,1) +B(i,j-1,1) -B(i-1,j,2) &
     +B(i+1,j,2))
     write(myrank+7000,*)'VORT(',i,j,')=', &
     (-v(i,j+1,1) +v(i,j-1,1) -v(i-1,j,2) &
     -v(i+1,j,2))
  enddo
enddo
...
```

```
...
35|          CL.168:
36| 0008FC   CALL gr3=_xlfBeginIO,7,gr3,gr4, c
36| 000920   CALL _xlfWriteFmt,5,gr3,x,gr4-gr7, c
36| 000940   CALL _xlfWriteFmt,5,gr3,x,gr4-gr7, c
36| 00094C   CALL gr3=_xlfEndIO,1,gr3, c
37| 00095C   BT   CL.168,cr0,0x8/llt,taken=80%(80,20)
...
```

図 9　サブルーチン output 内のネストしたループの Fortran ソースコードとコンパイラの出力したリスト
Fig. 9　Fortran source and compiler listing for code excerpt in *output*.

る．ソリューションルールは，図 8 のアルゴリズムに基づいて，C 言語で実装されている．

このボトルネックは，LBMHD にも含まれており，output サブルーチンに含まれている．**図 9** の左側は，対象となるネストしたループを示し，右側は，IBM XLF コンパイラでコンパイルされたループ部分のリストを示す．リストでは，write 文が 1 つの_xlfBeginIO 関数と，2 つの_xlfWriteFmt 関数，そして 1 つの_xlfEndIO 関数に変換されている．_xlfWriteFmt 関数が実際の出力を行い，_xlfBeginIO と_xlfEndIO は，I/O バッファの管理などを行う．

```
...
write(myrank+5000,*)(' CURR(',i,j,')=', &
     (-B(i,j+1,1) + B(i,j-1,1) - B(i-1,j,2) &
     + B(i+1,j,2)),'\n', &
     i=imin,imax), j=jmin,jmax)
write(myrank+7000,*)(' VORT(',i,j,')=', &
     (-v(i,j+1,1) + v(i,j-1,1) - v(i-1,j,2) &
     + v(i+1,j,2)),'\n', &
     i=imin,imax), j=jmin,jmax)
  ...
```

```
...
34| 0008D0  CALL  gr3=_xlfBeginIO,7,gr3,gr4, c
34|         CL.168:
34| 000918  CALL  _xlfWriteFmt,5,gr3,x,gr4-gr7, c
34| 000934  CALL  _xlfWriteFmt,5,gr3,x,gr4-gr7, c
34| 000944  BT    CL.168,cr0,0x8/llt,taken=80%(80,20)
34| 00094C  CALL  gr3=_xlfEndIO,1,gr3, c
...
```

図 10　サブルーチン output の最適化されたコードと，そのリスト
Fig. 10　Optimized code of *output* with listing.

_xlfBeginIO と_xlfEndIO の引数がループ内で不変であれば，オーバヘッドを減らすために，ループ外に出すことができる．implied-do を使うと，_xlfBeginIO と_xlfEndIO は，ループ内で繰り返されない．**図 10** は，最適化されたコードとそのリストであり，ネストしたループが implied-do に変換され，2 つの write 文になっている．そして，_xlfBeginIO と_xlfEndIO は，ループの外に出ている．

6. 最適化の効果

この章では，前章で述べた最適化による性能向上を示す．実験環境としては，4 台の POWER5+^{TM} マシンをインフィニバンドで接続したものを用意した．それぞれのマシンは，8 台の 1.9 GHz POWER5+^{TM} プロセッサと 64 GB DDR2 メモリを搭載している．

まず，計算に関する最適化を LBMHD に適用した場合について，1 プロセッサ上で測定した結果を示す．IBM XLF コンパイラを用いてコンパイルし，最適化オプションは，-O3 -qhot とした．グリッドサイズ 2,048 × 2,048，50 回繰返しで実行したところ，実行時間が約 20%向上した（**図 11**）．stream のループ交換なしでは，collision の性能が向上したにもかかわらず，全体の性能が 5%低下する．ループ交換が，ストレージ順序の変更による stream への不利な影響を緩和するが，stream の性能低下は残る．これは，配列 g，geq へのアクセスパターンの変化によるもので，変更前のコードでは，一番内側の次元が（iend - ista）の範囲に広がっていたが，変更後は，1 から 2 に狭くなっているためと考えられる．

次に，計算，通信，I/O に関する最適化を LBMHD に適用した場合について，4 台の POWER5+^{TM} マシンで，各 1 台のプロセッサを用いて測定した結果を示す．LBMHD は，2,048 × 2,048 のグリッドサイズで，50 回計算し，10 回ごとに中間結果をファイルに出力する．ファイルシステムとしては，通信の影響を避けるために，ノードに直接接続されたハー

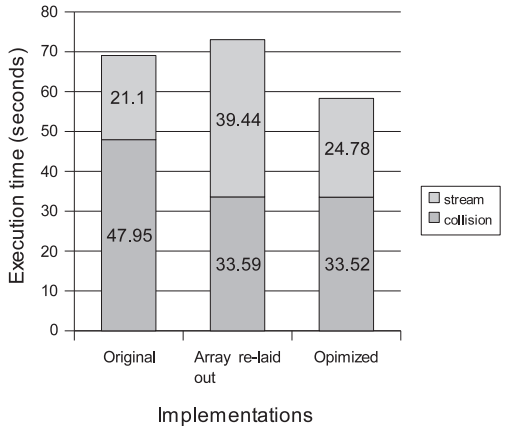


図 11　計算に関する最適化による性能向上
Fig. 11　Performance impact of optimization for calculation.

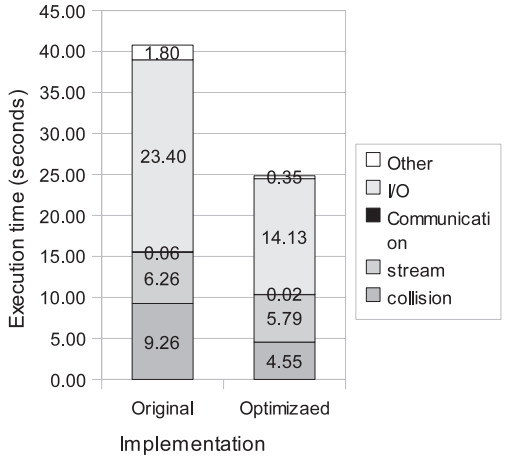


図 12　最適化による性能向上
Fig. 12　Performance impact of optimizations.

ドディスクを用いるローカルファイルシステムを使用した．IBM XLF コンパイラを用いてコンパイルし，最適化オプションは，-O3 -qhot とした．

図 12 に，最適化適用後の性能向上を示す．図の中で，collision と stream は，上でも述

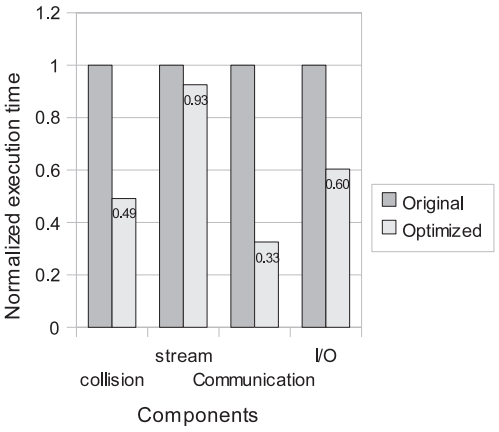


図 13　各要素の比較
Fig. 13　Comparison for each component.

べた，それぞれ対応するサブルーチンの実行にかかる時間であり，配列のアクセス順序とストレージの順序を合わせるソリューションによって，性能が向上している．1 プロセッサの場合と異なり，最適化適用後に stream が高速化しているが，これは，並列化によって，1 台のプロセッサに割り当てられるデータが少なくなり，パイプラインストールが減少したためと考えられる．図 12 の Communication は，output サブルーチンで通信にかかる時間である．通信は隣接プロセス間でしか発生しないため，CPU や I/O と比べて実行時間は短いが，非同期化ソリューションによって減少していることが分かる．また，I/O の実行時間は，implied-do 化ソリューションによって，大きく減少している．残った時間は，プログラムの他の部分の実行にかかる時間で，図 12 では，Other となっているが，非常に小さい．全体の性能向上は，39.1%である．

図 13 は，最適化前後での各要素の実行時間の比較である．どの要素についても，左が元プログラムでの実行時間で，右が，最適化プログラムでの実行時間である．実行時間は，元プログラムの実行時間で正規化されている．計算，通信，I/O，いずれも元プログラムと比較して高速化されている．

フレームワークは，IBM®pSeries®システム，AIX®上に，IBM®XLC/C++で実装されており，コード規模は，およそ 10 万行である．ボトルネックを特定する部分の一部が，IBM®alphaWorks® [16] で公開されている．

7. まとめと今後の課題

本論文では，生産性の高い性能最適化のために，性能測定ツール，コンパイラ，そして専門家の知識を統合して提供するフレームワークについて述べた．フレームワークは，ボトルネックを検出し，ソリューションを提案，適用するプロセスを繰り返す．また，既存の性能ツールによって収集された性能データを指標として利用することができ，複数のツールの分析を，ボトルネックルールによって関連付け，まとめることができる．その際，コンパイラによる分析と最適化は，フレームワークの中で重要な役割を果たしている．

我々は，HPC アプリケーションの最適化を通して，フレームワークによる性能最適化について説明した．最適化プロセスは，高度に自動化され，性能向上は著しいものであった．

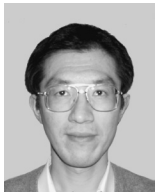
我々は，フレームワークの有効性は，データベースのボトルネックルールとソリューションの量と質によると考えており，今後はルールとソリューションを増やしていく予定である．また，フレームワークが拡張やカスタマイズのために提供するサービスとユーティリティも改良していく予定である．

謝辞　本研究は，米国国防高等研究計画局（US DARPA）契約番号 HR0011-07-9-0002 による．

参 考 文 献

1) High productivity computer systems, online (2005). http://highproductivity.org
2) Chen, W.Y., Mahlke, S.A., Warter, N.J., Hank, R.E., Bringmann, R.A., Anik, S. and Hwu, W.W.: Using profile information to assist advanced compiler optimization and scheduling, Lecture Notes in Computer Science, Vol.757, pp.31–48 (Jan. 1993).
3) Whaley, R.C. and Dongarra, J.J.: Automatically tuned linear algebra software (atlas), *Supercomputing 98*, Orlando, FL (Nov. 1998).
4) Vuduc, R., Demmel, J.W. and Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels, *SciDAC 2005, Journal of Physics: Conference Series* (2005).
5) Malony, A.D., Shende, S., Bell, R., Li, K., Li, L. and Trebon, N.: Advances in the tau performance system, *Performance analysis and grid computing*, pp.129–144, Kluwer Academic Publishers, Norwell, MA (2004).
6) Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K. and Newhall, T.: The paradyn parallel performance measurement tool, *IEEE Computer*, Vol.28, pp.37–46 (Nov. 1995).
7) Mohr, B. and Wolf, F.: Kojak – a tool set for automatic performance analysis of parallel applications, *Euro-Par 2003: Proc. International Conference on Parallel and Distributed Computing* (Sep. 2003).
8) Pillet, V., Labarta, J., Cortes, T. and Girona, S.: Paraver: A tool to visualise and analyze parallel code, *WoTUG-18: Transputer and occam Developments*, Vol.44, pp.17–31, Amsterdam, IOS Press (1995).
9) Cong, G., Chung, I-H., Wen, H.-F., Klepacki, D.J., Murata, H., Negishi, Y. and Moriyama, T.: A holistic approach towards automated performance analysis and tuning, *Proc. Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference*, pp.33–44 (Aug. 2009).
10) Wen, H.-F., Sbaraglia, S., Seelam, S.R., Chung, I-H., Cong, G. and Klepacki, D.J.: A productivity centered tools framework for application performance tuning, *QEST '07: Proc. 4th International Conference on the Quantitative Evaluation of Systems* (*QEST 2007*), pp.273–274 (2007).
11) Bhatele, A. and Cong, G.: A selective profiling tool: Towards automatic performance tuning, *3rd Workshop on System Management Techniques, Processes and Services* (*SMTPS' 07*), Long Beach, California (Mar. 2007).
12) DeRose, L., Ekanadham, K., Hollingsworth, J.K. and Sbaraglia, S.: Sigma: A simulator infrastructure to guide memory analysis, *2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, pp.1–13 (Nov. 2002).
13) Geimer, M., Wolf, F., Wylie, B.J.N., Abraham, E., Becker, D. and Mohr, B.: The scalasca performance toolset architecture, *International Workshop on Scalable Tools for High-End Computing* (*STHEC*), Kos, Greece (2008).
14) Cong, G., Seelam, S.R., Chung, I-H., Wen, H.-F. and Klepacki, D.J.: Towards next-generation performance optimization tools: A case study, *1st Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems*, Austin, Texas (Mar. 2007).
15) MacNab, A., Vahala, G., Pavlo, P., Vahala, L. and Soe, M.: Lattice Boltzmann Model for Dissipative Incompressible MHD, *28th EPS Conference on Contr. Fusion and Plasma Phys*, Vol.25A, pp.853–856 (June 2001).
16) High productivity computing systems toolkit, IBM®alphaWorks®. http://www.alphaworks.ibm.com/tech/hpcst

(平成 22 年 10 月 1 日受付)
(平成 23 年 1 月 13 日採録)



村田　浩樹（正会員）

昭和 39 年生．平成元年早稲田大学大学院理工学研究科電子通信学専門分野専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．東京基礎研究所にて並列計算機のハードウェア，分散システム，超並列計算機プログラムの利用技術・自動最適化技術等に関する研究に従事．

根岸　　康（正会員）

昭和 39 年生．平成元年東京工業大学大学院理工学研究科情報科学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．東京基礎研究所にてシステムソフトウェア，通信プロトコル，超並列計算機プログラムの利用技術・自動最適化技術等に関する研究に従事．ACM 会員．

森山　孝男（正会員）

昭和 37 年生．昭和 62 年東京工業大学工学部情報理工学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．東京基礎研究所にて並列計算機のシステムソフトウェア，高速 3D 表示装置，ハイブリッドアーキテクチャ，超並列計算機の利用技術等に関する研究に従事．ACM 会員．

グオジン　コン

平成 16 年米国ニューメキシコ大学より Ph.D.（工学）．現在，IBM T.J. Watson 研究所リサーチ・サイエンティスト．本プロジェクト技術リーダー．High Performance Computing システム上での大規模グラフ解析，科学技術系アプリケーションの性能解析，最適化等の研究に興味を持つ．IEEE シニアメンバ．

イシン　チュン

IBM T.J. Watson 研究所リサーチ・スタッフ・メンバ．最適化，性能解析，性能測定ツールの研究に興味を持つ．IBM®POWER®システム AIX®および Linux®，そして Blue Gene®システム等の IBM プラットフォーム用の性能測定ツールの設計，開発等に従事．IBM 入社前，平成 16 年米国メリーランド大学カレッジパーク校より Ph.D.（計算機科学）．

フイファン　ウェン

IBM T.J. Watson 研究所アドバイザリー・ソフトウェア・エンジニア．IBM アドバンスト・コンピューティング・テクノロジー・センタにて性能測定ツールと GUI の設計に従事．米国メリーランド大学カレッジパーク校より修士（計算機科学）．平成 17 年 IBM 入社．

デヴィッド　クレパッキ

平成元年米国パーデュー大学より Ph.D.．同年 IBM 入社．現在，IBM T.J. Watson 研究所シニア・スタッフ・メンバ，アドバンスト・コンピューテイング・テクノロジー部長．2009 年のナショナル・メダル・オブ・テクノロジー・アンド・イノベーションを受賞した IBM®Blue Gene®システム等大規模並列コンピュータシステムの最適化技術の研究開発に従事．