CC-005

広帯域低レイテンシの Approximate ネットワークに対する自動チューニング手法
An Auto-tuning Method for High-bandwidth Low-latency Approximate Networks

平澤 将一 †　Truong Thao Nguyen‡　鯉渕 道紘 †
Shoichi Hirasawa　Truong Thao Nguyen　Michihiro Koibuchi

1 概要

我々は，通信路上で一定のビット誤りを許容することで，相互結合網の広帯域化と低遅延化を両立し，その結果，並列計算アプリケーション高速化を達成する Approximate ネットワークを提案している [1]．誤りを許容する相互結合網を活用した上で正しい (許容誤差範囲内の) 計算結果を得るためには，アプリケーション側の対応が必要である．アプリケーションが許容できる誤りの大きさと誤りを許容する相互結合網のパラメータに依存して，実行性能は大きく変化する．現状では，アプリケーション開発者が，このパラメータ値を手動で時間をかけて探索，設定しており，Approximate ネットワークの導入の大きな障壁となっている．そこで本研究では，誤りを許容する相互結合網を用いてアプリケーションを実行する環境において，正しい結果を得つつ，高速に実行できるパラメータを発見する自動チューニング手法を提案する．SimGrid Event Discrete シミュレーションの評価の結果，本提案自動チューニングによって，256 プロセスを用いた共役勾配法に対して，10^{18} 通りのパラメータの組み合わせの中から，932 回および 1099 回の試行により，精度と実行時間の面で最適解を得ることができた．

2 はじめに

半導体デバイスの微細化と動作電圧が下がることにより，最近のスーパーコンピュータ (以後，スパコンと呼ぶ) から携帯端末の組込プロセッサに至る様々なコンピュータではビット化け対策が重要となってきている．ビット化けとは，ハードウェアの故障による恒久的な不具合ではなく，メモリに格納されているデータなどの一部のビットが，反転 (0↔1) してしまう不良が非決定的に発生する不具合である．現在のスパコン，データセンターのコンピュータには様々なビット化けを検出訂正する機構が搭載されているが，現実的なハードウェア/ソフトウェアコストで，この不良から完璧に回復することを保証することは難しい．つまり，アプリケーションの実行が強制終了とならず，しかし，アプリケーションの計算結果が変造されるサイレントエラーが起こりうる．

一方で，いくつかの並列計算ではビット化けへの耐性を持っている．つまりビット化けが生じても計算結果の大勢に影響せず十分な場合がある．この点に注目して，許容誤差を若干大きくすることで計算の精度を落とし消費電力を削減，ハードウェアのスループットを向上させる Approximate Computing の研究が進んでいる [2]．特に，多くのディープラーニング系の計算をプロセッサの倍精度演算ではなく，半精度演算で行っても結果の大勢に影響しないことが報告されているなど，最近のビッグデータ処理では Approximate Computing への期待が大きい．その結果，特定のニューラルネットワークの処理を (ノイズが伝搬し，ビット化けが高い確率発生しうる) アナログコンピュータにより高いエネルギー効率で実行する技術など新たな計算基盤の潮流が生じている [3]．

ただし，現在，ほぼすべての Approximate Computing の研究がコンピュータノード内 (特にプロセッサ，メモリと言語処理系) を対象に行われている．しかし，現状のネットワーク機器は，ビット化けについて標準規格を厳密に守っているため，Approximate Computing の考え方に基づく研究開発は見られない．例えば，イーサネットの規格では 10^{-12} のビットエラー率*を定めている．つまり，その高信頼性を確保するために，設計コストを払い，また，自らが性能限界をつくっているともいえる．我々は，この標準規格から逸脱することで数倍～10 倍の通信帯域の増加と，大幅な通信遅延の低下を両立できることを提示した [1]．

ただし，Approximate ネットワークを活用した上で正しい計算結果を得るためには，アプリケーション側の対応が必要である．我々の研究成果 [1] より，エラー耐性を持つ並列アプリケーションでは，ビット化けを許容できるプロセス間通信と，完璧なデータ転送が要求されるプロセス間通信があることが分かった．さらに，ビット化けを許容する通信でも，一部のビットについては完璧な転送が必要となる場合が多い．これは，IEEE754 浮動小数点数表現における符号部，指数部，仮数部の各ビット化けがもたらす数値誤差は均一ではないことに起因する．例えば，符号部にビット化けが生じた場合，値が反転するため，その数値の絶対値の 2 倍の誤差が生じることになる．一方，仮数部の下端ビットが反転した場合，極めて小さな値の誤差に留まる．

† 国立情報学研究所, National Institute of Informatics,　‡ 総合研究大学院大学, SOKENDAI

*平たく述べると，誤ったデータの受信確率

つまり，現状，Approximate ネットワークにおいて，既存の並列アプリケーションを実行する場合，(1) ビット化けを許容できる通信と完璧なデータする通信の区別，(2) ビット化けを許容できる通信の場合，どのビットまでがビット化けを許容できるのか，を都度，プログラムコードと実行結果から解析する必要がある．つまり，従来のアプリケーション設計が「性能」「消費電力」のトレードオフを探求するチューニングをしていたことに対して，Approximate アプリケーションは，加えて「精度」という軸を加えてトレードオフを探求することになる (図 1)．そのため，このアプリケーション設計パラメータの探索空間は極めて大きく，効率的に精度と性能の最適化を施すことが難しくなっている．

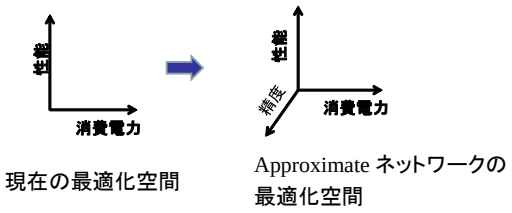

図 1: Approximate アプリケーションの設計空間

そこで本研究では，Approximate ネットワークを用いてアプリケーションを実行する環境において，正しく動作することを保証した上で高速に実行できるパラメータを発見する自動チューニング手法を提案し，性能評価を行う．

概要を図 2 に示す．ネットワーク側では光リンクを想定する．電気から光に多値変調する場合に，重要なデータの通信とそうでない通信毎にコーディングのシンボルマッピングを動的に変更する (詳細は 4 章)．つまり，通信帯域を大きくするためには多値変調が必要となるが，多値変調を行うとビットエラー率が悪くなる．一方，2 値のみを用いて変調すると，ビットエラー率が大幅に改良されるが，通信帯域が小さくなってしまう．

一方，アプリケーション側では重要なデータ通信 (図 2 では *MPI_DOUBLE* 型) と，そうでない場合 (同 *Approx_MPI_DOUBLE* 型) を識別する必要がある．よって，ネットワークとアプリケーションのコデザインが極めて重要であり，この点で本自動チューニングが，アプリケーションの実行誤差を許容範囲内に抑えつつ，通信性能，ひいてはアプリケーションの実行性能を最大化する．

本稿の貢献は以下である．

- 既存の自動チューニングフレームワークを拡張することで，Approximate ネットワークで必要となる精度と性能に関するアプリケーション毎の自動最適化を可能とした．
- 256 プロセスを用いた共益勾配法の並列計算において，932 回および 1099 回の試行により，完璧な計算をした場合と比べて必要となる精度を満たしつつ，実行時間を約 10%削減した．また，この実行時間は必要となる精度を満たした中での最速実行であった．

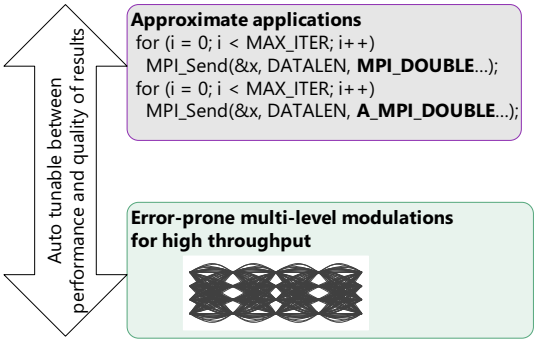

図 2: Approximate ネットワークにおける自動チューニング

本稿は以下，3 章で研究の背景を述べ，4 章で Approximate ネットワークについて概観する．5 章で提案する自動チューニング手法について述べ，6 章で評価する．7 章で議論を行い，8 章でまとめる．

3 背景

3.1 エラー検出訂正と通信遅延

現状のスパコンの相互結合網は，Cray YARC ルータ [4] や InfiniBand QDR スイッチに代表されるように，転送データに対して hop-by-hop で CRC(巡回冗長検査) 処理を行うことでビットエラー検出を行うことが多い．しかし，今後さらなる高バンド幅リンクの実現が望まれており，その場合エラーフリー転送を実現するために FEC(Forward Error Correction) によるエラー検出訂正が必要となる． FEC 処理では，直前のビット列情報が必要となるため，エラー検出訂正の計算に加えて，バッファリング遅延が必ず生じる． 25Gbps 通信において，Base-R FEC を用いた場合，2,112 ビットエンコーディングブロックの受信のために 84 ns 必要となる． よって，FEC 処理は計算時間を含めて，1 ホップあたり 100ns を見込んでいる．25Gbps×4 100G イーサネットの場合も，この遅延 (100ns) となる． また，次世代の相互結合網である 200GbE/400GbE では 50Gbps×4 という強度方向に 4 値で変調する PAM 4（4-level Pulse Amplitude Modulation） の使用が見込まれている．そこではリードソロモン符号 RS(544,514) という極めて大きいバッファリングを前提としているため，50ns 以上の遅延が指摘されている．

現在，最先端のスパコンの相互結合網で採用されているスイッチのパケット通過遅延が 40〜100 ナノ秒，また，ホストでの送受信メッセージオーバーヘッドが 100 ナノ秒 [5, 6] である．よって，低遅延 HPC ネットワーク (例:10 万 endpoint で 1 μ 秒通信 [7]) において，これらの FEC 処理は採用が望ましくない．

3.2 自動チューニング

近年の計算機は，アウトオブオーダ実行や分岐予測，キャッシュなど動的な制御による高速化手法を多数取り入れて複雑化しており，性能をモデル化することによってアプリケーションを静的に最適化することが困難になった．コンパイラが静的に最適化するだけでは，アプリケーションコードが実際に実行される計算機の各パラメータに対応できないためである．

自動チューニングは，複雑化した計算機環境に対して，特定のアプリケーションコードおよび試行データ，アプリケーションを実行する計算機を定めることによって実際に実行して性能を測定し，測定された性能の結果を最適化に活用する方法として注目されている．アプリケーションコードおよび実行計算機を定めることによって，一般的には適用不可能な最適化を適用可能とし，その結果高速化を達成することが可能となることが知られている．

これまで自動チューニングは，高速フーリエ変換 (FFT:fast fourier transform) や BLAS(Basic Linear Algebra Subprograms) など数値計算ライブラリを中心に研究されてきた．これは，計算内容が既知である場合に，あらかじめ開発者が定めたチューニング手順に従って，ユーザが実行する計算機上でチューニングを完了する手法であり，一般のアプリケーションに対してそれぞれの手法を適用することは困難だった．

これに対して，チューニングの手順を汎用のプログラミング言語で記述することで，任意のチューニング手順を定義可能とする汎用の自動チューニングフレームワーク [8] が提案されている．しかしながら，汎用の自動チューニングフレームワークは適用するアプリケーションコードを修正する必要があることに加えて，計算精度を低下させる最適化および計算環境に適用できていない．

4 Approximate ネットワーク

本章では，光リンクと電気スイッチで構成される現状のスパコン，データセンターを対象とした，Approximate ネットワークについて述べる．より詳細な設計については文献 [1] を参照いただきたい．

4.1 物理/リンク層

200GbE, 40GbE といった広帯域通信規格では，多値変調が用いられる見込みである．多値変調のトレードオフとして，2 値変調と比べて同一信号パワーでは SNR（Signal to Noise Ratio）が劣化するため，受信側でのビットエラー率悪化につながる点がある．Approximate ネットワークではこの点を利用する．

Approximate ネットワークでは最大 1,024-QAM(Quadrature Amplitude Modulation)，あるいは PAM により 1 シンボルあたり最大 10 ビット転送することで 1Tbps(250Gbps×4) リンクをベースラインとする．そして，エラー検出訂正に関して，FEC を用いない．ただし，この場合，我々が想定する変調速度 250 Gbps の場合ではリンク上の伝送においてエラーフリー ($< 10^{-12}$) を達成できなくなる．なお，信号からビット列への変換については，現状通り，信号の隣接符号間のハミング距離を 1 とする Gray コードを想定する．

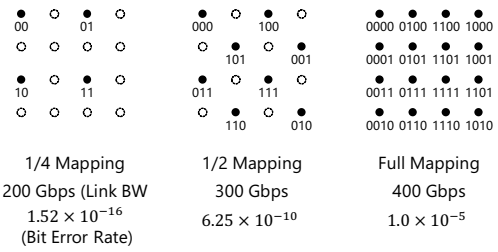

図 3: 16-QAM における 4 値 (2bit)，8 値 (3bit)，16 値 (4bit) 転送の例．16-QAM の一部のシンボルのみをデータ転送に用いることで，隣接符号間の距離が大きくなるため，ノイズ耐性が高くなる (ビットエラー率の解析は文献 [1] による．)

4.2 アプリケーション層

Approximate Computing ネットワークでは伝送路上においてビットエラーが生じることを前提にデータ転送する．しかし，ビットエラーが生じた場合でも原則再送せず，データを受信したプロセス (プログラム) はそのまま計算処理を続行する．

そこで，ビット列から数値の符号化を工夫することで，ビットエラーがもたらす転送データ値の誤差を最小化することが望ましい．しかし，本報告で対象とする並列アプリケーションにおいてプロセス間通信の転送データのフォーマットである IEEE754 浮動小数点数表現における符号部，指数部，仮数部の各ビットエラーがもたらす数値誤差は均一ではない． 例えば，符号部にエラーが生じた場合，値が反転するため，その数値の絶対値の 2 倍の誤差が生じることになる． 一方，仮数部の下端ビットが反転した場合，極めて小さな値の誤差に留まる．そこで，我々はビットエラーがもたらすアプリケーションの転送データの数値への影響を最小化するように符号部，指数部の上位ビットに対して M-QAM の一部の符号のみを用いている．つまり，保護したいビット列の転送時にはビットエラー率を改善させる．図 3 の例では，文献 3 において解析したビットエラー率とバンド幅の関係を示している．

5 Approximate ネットワークにおける自動チューニング

5.1 概要

Approximate ネットワークは，Approximate 通信の際に保護するビット数を設定可能である [1]．また，Approximate ネットワークを活用するアプリケーションは，ビット化けを許容する通信と，ビット化けを許容しない通信を指定可能である．

つまり，ビット化けを許容する通信が Approximate ネットワークにより高速に転送されることによりアプリケーションの高速化が期待できる．一方，多大なビットエラーにより求める解に収束しないことが起こる．

そこで，Approximate ネットワークにおける自動チューニングでは，Approximate ネットワークのパラメータであるビットマスクと，アプリケーションのパラメータである，ビット化けを許容する通信の組み合わせ全体に対してチューニングする必要がある．さらに，既存の自動チューニングとは異なり，パラメータの組み合わせに対する試行において，性能だけではなく，アプリケーションが求める解で正しく実行を完了したことを確認する必要がある．

以上を満たす，Approximate ネットワークにおける自動チューニング手法について以下詳細を述べる．

5.2 自動チューニングの探索空間

Approximate ネットワークを利用するアプリケーションでは，並列実行を進めるためにプロセス間通信を行う．既存のアプリケーションは正確な通信を行うネットワークを前提としているため，Approximate ネットワークを活用しても正しく計算を完了することができるプロセス間通信を特定し，あるいはそのようなプロセス間通信があった場合に，どの程度正確な通信が必要であるか，通常，事前には分からない．そこで，提案する自動チューニング手法を活用するために，アプリケーション中のプロセス間通信を，正確なプロセス間通信が必要な通信と，Approximate ネットワークを活用してビット化けを許容する通信に明示的に分類し，これを自動チューニングによって探索する．

Approximate ネットワークを活用するプロセス間通信が定義されたとしても，各データのどのビットまでは正確に送信する必要があり，どのビットからはビット化けを許容するか，未知である．そこで，Approximate ネットワークを活用するプロセス間通信で許容するビット化けの数をパラメータ化し，自動チューニングによって探索する．

Approximate ネットワークでは，通信速度とエラー率のトレードオフが存在する．すなわち，許容するビット化けの頻度が低い (エラー率が低い) 場合には必要なエラー補正が多くなり通信速度が低く，逆に許容するビット化けの頻度が高い (エラー率が高い) 場合には必要なエラー補正が少ないため通信速度が高い．このトレードオフを静的に決定することは困難であるため，自動チューニングによって探索する．

これらのパラメータはそれぞれ従属関係にあるために独立して探索することは困難である．すなわち，アプリケーションが正しく動作するために許容できるプロセス間通信に依存して正確に送信する必要があるビット数が定まり，また同時にその組み合わせにおいてアプリケーションが正しく動く場合に最も高速となるエラー率と速度のトレードオフポイントが定まる．したがって，提案する自動チューニング手法においては，これらのパラメータ全体が構成する空間を探索する．

5.3 解の保証

自動チューニングにおいてアプリケーションのあるパラメータセットに対する試行の終了後に，正しい解が出力されたかどうかを確認する．解の正しさは一般には定義できないが，アプリケーション側で正しさを評価する方法やコードが提供されている場合や，アプリケーションのユーザが固有の正しさを定義する場合がある [2]．

提案する自動チューニング手法においては，試行するアプリケーションの実行終了後に正しい解が出力されたか確認する手段があらかじめ提供されていることを前提とする．これは，アプリケーション自体が正しさを確認して結果を出力するコードを含む場合および，アプリケーションとは独立に，アプリケーションの出力を入力として，正しさを確認した上でその確認結果を出力するコードが提供されている場合のいずれも含む．

アプリケーションの実行が終了して正しくない解が出力された場合には直ちにその結果が確認可能であり，対応するパラメータセットは自動チューニングの結果として選択されない．

5.4 探索手法

アプリケーションと Approximate ネットワークのパラメータを独立した次元として構成した探索空間は広大であり，全体をくまなく探索することは現実的な時間では不可能である．

したがって提案する自動チューニング手法では，Approximate ネットワークを用いてビット化けが起こった場合にアプリケーションの正しくない実行結果が出力されることと，それを確認して検出可能である性質を利用して探索空間を大幅に狭める．つまり，正しくない実行結果が出力された場合に，それに至るパラメータセットの近くを探索しないような探索を行う．

具体的には，アプリケーション中のプロセス間通信に注目し，特定のプロセス間通信において Approximate ネットワークを利用した場合に起きたビット化けに対して正しくない実行結果が出力された場合にはペナルティを課し，そのプロセス間通信に対応する探索空間での探索数を少なくする．逆に，特定のプロセス間通信において Approximate ネットワークを利用した場合

に起きたビット化けに対して正しい実行結果が出力された場合には，そのプロセス間通信に対応する探索空間での探索数を多くする．

これにより，正しくない解に至ったパラメータセット近辺の探索よりも他のパラメータセットの探索を優先させ，探索に要する時間を低減する．

6 評価

6.1 CG 法の実装

6.1.1 CG 法

共益勾配法 (Conjugate Gradient method:CG 法) は連立一次方程式を反復法により解くアルゴリズムである．反復解法においては，仮の解を設定した上で，反復的に最終的な解に近づいていく性質を持つ．この性質を活用することにより，Approximate ネットワークを活用することができる．すなわち，計算の途中経過に現れる仮の解は当然に厳密解ではなく，その後の反復計算において求めたい解に漸近していく動作が，本質的に Approximate ネットワークが持つ非厳密性に対する耐性を持つためである．

最終的に求めたい解が計算終了時に出力されることが保証される限り，Approximate ネットワークが持つ高速性を活用することができる．それにより，アプリケーション実行全体の高速化が期待できる．

6.1.2 Approximate ネットワークにおける実装

本評価においては，MPI により並列化された CG 法ベンチマークプログラムとして NAS Parralel Benchmark の共益勾配法 (以下 NPB CG) を用いる．

NPB CG においては，合計約 1800 行からなる FORTRAN プログラムであり，合計 20 個の MPI 送受信の呼び出しが存在する．MPI 送受信は特定の 2 個が送受信の組として記述されており，すなわち，10 個の MPI 送受信の組が存在する．

NPB CG に対して，図 4 のようにアプリケーションコードを変更させた上で試行を行う自動チューナを記述した．自動チューナは SimGrid Event Discrete シミュレータ v3.12[9] を用いて，保護するビット数とエラー率を変更しながら試行し，NPB CG が正しい解を出力していることを確認しつつ，実行性能を記録する．

6.2 シミュレーション方法とパラメータ

Approximate ネットワークを用いた評価に SimGrid Event Discrete シミュレータ v3.12 [9] を用いた．SimGrid Event Discrete シミュレータにおいて，Message Passing Interface (MPI) で記述された並列計算アプリケーションを改変なしに実行シミュレーションすることができる [10].

ネットワークは 256 台のスイッチからなり，各スイッチには 100 GFlops の計算ノード 256 台が接続されていると仮定した．光リンクのビット化け許容時のリンク帯域 1Tbps，許容しない時はリンク帯域 375Gbps で転送することとした．なお，我々は文献 [1] において，この 2 つのシンボルマッピングをビット単位で動的に行うオーバヘッドはネットワークインタフェースの面積，スループット面で現状ではボトルネックにならないことは示されている．

スイッチ遅延は最小 60 ナノ秒とし，スイッチ間はトーラスネットワークトポロジ，およびドラゴンフライトポロジで相互接続し，ダイクストラ法による最短経路ルーティングを採用した．その他のパラメータは文献 [1] と同様である．

```
...
BINS = [
  'approx.100000',
  'approx.010000',
  'approx.001000',
  'approx.000100',
  'approx.000010',
  'approx.000001',
  ...
  'approx.111111'
]

... # add a demesion to define the parameter space
manipulator.add_parameter(
    EnumParameter('bin', BINS)
  )

... # reflect an instance in the parameter space
run_cmd += './bin/cg.A.256.{0}'.format(cfg['bin'])

... # verification
assert 'VERIFICATION SUCCESSFUL' in run_result['stdout']
```

図 4: OpenTuner を用いて記述された自動チューナのコード (抜粋)

6.3 評価結果 (2 次元トーラストポロジ)

CG 法を問題サイズ A，プロセス数 256 にて実行した．10 個の MPI 送受信の組の中で，6 個の MPI 送受信の組だけがビット化けに対する耐性があることがわかった．6 個の MPI 送受信の組のうち，Approximate ネットワークでビット化けを許容する呼び出しを用いるペアを 1，それ以外を 0 とし，6 個のペアをビットマスクとして表現する．

1 個のペアだけビット化けを許容する場合の実行時間を図 5 に示す．用いたパラメータは，

```
PROTECTION_MASK=0xffff000000000000
ERR_RATE="1e-5“
BW_FACTOR=‘‘0.375’’
```

である．

それぞれのビット化けを許容する呼び出しにおいて，NPB CG の実行終了までに実際に起きたビット化けの数を図 6 に示す．

本稿における自動チューナで試行した場合の全実行時間を図 7 に示す．図 7 は，横軸が試行したパラメー

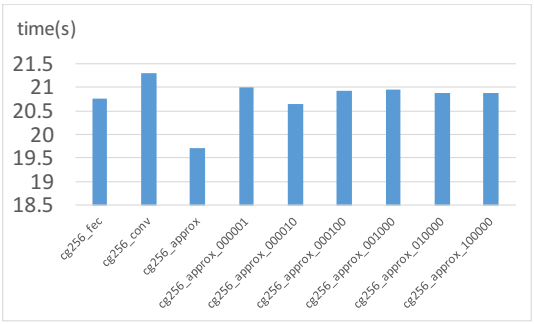

図 5: 1 ペアの MPI 送受信だけ Approxmate ネットワークでビット化けを許容する呼び出しを行った場合の実行時間比較

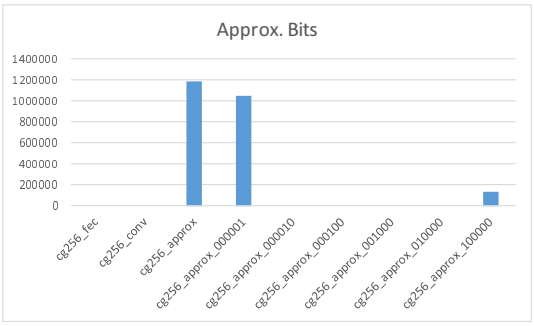

図 6: CG 法の実行において生じたビット化けの総数

タセット，縦軸が実行時間を表す．

パラメータセットの母集団として

```
PROTECTION_MASKS:
0x0000000000000000 <-> 0xffffffffffffffff
BINS:
approx.111111, approx.100000 <-> approx.000001
ERR_RATES:
1e-0 <-> 1e-15
```

を用いる

それぞれの試行において，それまでの試行の中で最高性能を発揮した場合である”temporary best” を 1，そうでない場合を 0 と表現した結果を図 8 に示す．試行の開始時には多数の試行結果が”temporary best” とな

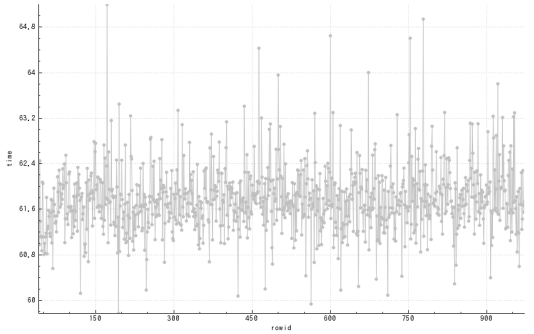

図 7: 本提案自動チューナーの試行毎の CG 法の実行時間．既存のネットワークを用いた場合の実行時間は 65.9s である．

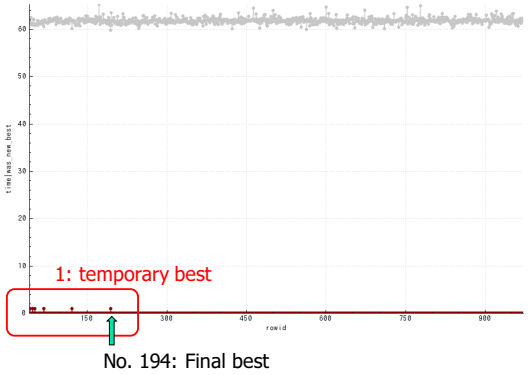

図 8: 各試行における最短実行時間の更新の有無

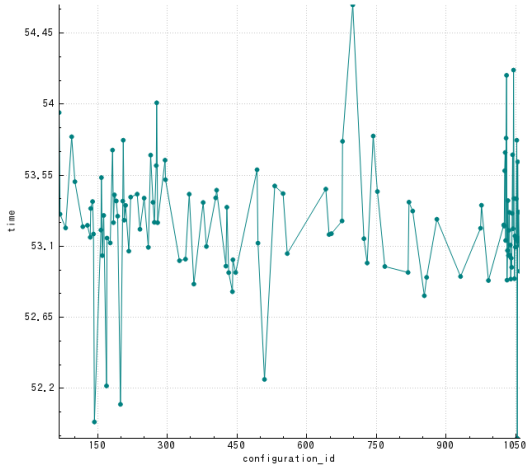

図 9: ドラゴンフライトポロジにおけるチューニング結果．既存のネットワークを用いた場合の実行時間は 56.6s である．

り，試行が進むに従って少なくなることがわかる．これは，試行の開始時には”temporary best” が低いためであり，試行が進むと高い性能を持つパラメータセットが”temporary best” として発見されるからである．

6.4 評価結果 (ドラゴンフライトポロジ)

ネットワークトポロジを次数 11 のドラゴンフライとした場合の評価結果を示す．その他のパラメータは前節と同じである．

試行したパラメータセットを横軸，対応する実行時間を縦軸としたチューニング結果を図 9 に示す．平均実行時間が 2 次元トーラストポロジより短い理由は，ネットワークトポロジの差異による性能差であると考えられる．試行回数は合計で 1099 であり，最適解は第 1050 回目に発見されている．最適解は”bin”: ”approx.100000”, ”mask”: ”0xfff0000000000000”, ”error”: ”1e-8” であり，これは 2 次元トーラストポロジの場合と異なる．したがって，ネットワークトポロジに依存して最適なパラメータセットが異なり，ネットワークトポロジに適応したアプリケーションのチューニングが必要であることがわかった．

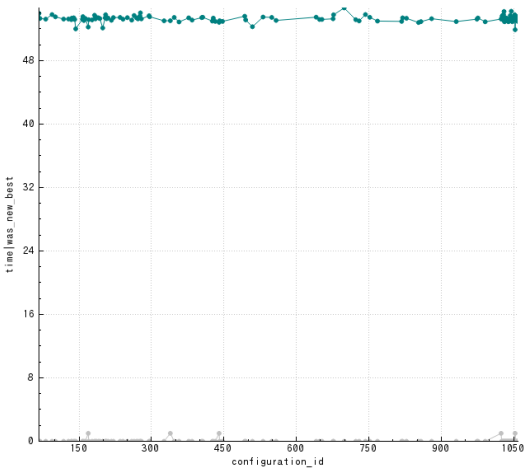

図 10: ドラゴンフライトポロジにおける各試行における最短実行時間の更新の有無

図 9 においては，試行の最後半に最適解が出ている．しかしながら，試行の前半に擬似最適があり，試行時間の差異は大きい．これについて，最終的な最適を探す必要があるのか，あるいは疑似最適で十分であるか，自動チューニングフレームワークが決定することは困難である．ユーザによる何かしらの方針を指示できるにより決められる方針が考えられ，今後の課題である．

7 議論

7.1 自動チューニングによる高速化

本 CG 法において自動チューニングで探索したパラメータは次の通りである．10 組の MPI 通信，各 MPI 通信のデータ (64 ビットの double 型) ビット列の中でビット化けを許容する範囲を示す 16 パターンのビットマスク，Approximate ネットワークリンクのビットエラー率 16 通り (10^{-0}〜10^{-15}) であり，その組み合わせは計 10^{18}(エクサ) 通りである．したがって，Approximate ネットワークにおいて十分な解の精度を得つつ，最速実行となる Approximate 通信の設定を発見するためには，単純に順番に実行したと仮定したとすると，1 つの実行時間 $\times 10^{18}$ が必要となり，現実的ではない．

これに対して，Approximate ネットワークにおいて，2 次元トーラストポロジを用いた Approximate ネットワークで本自動チューニングが実行した探索数 (つまり実行回数) が合計 932，最適のパラメータセットを 194 番目で発見できている．これは正しくない実行結果を出力するパラメータセットに対する探索を可能な限り少なくする自動チューニング手法が探索数を削減したためであり，劇的に探索数を削減できたと理由と考えられる．

7.2 最適解と自動チューニング解との剥離

CG 法の探索結果から，本自動チューニング手法により，既存手法 [1] において手作業により発見したパラメータセットと同じ “優れた” パラメータセットを発見できることが示された．

手動により 10^{18} 通りのパラメータセットを探索することは一般に不可能であり，自動実行であっても多大な探索時間を必要とするが，解の正しさに注目した本自動チューニング手法により，最適解を発見できる探索空間を維持したまま，解の正しさに注目した探索空間の削減により現実的な試行時間で発見することがわかった．

ただし探索手法の組み合わせおよび探索の停止条件に依存して自動チューニングが部分最適解で停止する可能性はあり，最適解を出力できる停止条件と探索空間の組み合わせについては今後の課題とする．

7.3 実行結果のばらつき

Approximate ネットワークにおいてビット化けは非決定的に生じる．そこで，自動チューニングにより選択されたパラメータセットに対して，エラー率のみ”1e-6” に設定し繰り返し 20 回試行した結果，全試行で正しい実行結果を得られることがわかった。この結果から，自動チューニングにより選択されたパラメータセットはビット化けに対する十分な耐性を持つといえる．

ただし一般には，各パラメータセットに対して非常に多数の試行を行った場合には，Approximate ネットワークを用いず，”1e-12”以下のエラー率に抑えることができる既存のネットワークであってもビット化けが起こり，その結果正しくない実行結果が出力される可能性がある．現代的な計算機システムを使用する計算においてはアプリケーションレベルの繰り返し実行などによりこのようなビット化けに対する耐性を担保して再現性を持たせる ABFT などの方法が取られることがあるが，ユーザにとって大きな負担である．この問題に対して，本自動チューニングフレームワークを拡張することにより，実行結果のばらつきを保存し，ユーザが求める成功確率 (たとえば 95%の信頼区間，99%の信頼区間など) に収まるパラメータセットを選択するよう指定することが可能であると考えられ，これは我々の今後の課題である．

7.4 現実的な Approximate ネットワークにおけるフレームワーク

並列アプリケーションの実行は，多くの場合パラメータサーベイが目的となる．したがって，繰り返し実行することが多い．これが自動チューニングを支持する 1 つの理由である．我々は，SimGrid Event Discrete シミュレータを用いることでさらに高速化できると考えている．つまり，実機の自動チューニング試行をすべて，SimGrid 内で行うのである．幸い，SimGrid と実機との実行時間の誤差は，ホモジニアスな高性能スパコン，データセンターシステムを対象としている場合，極めて小さいことが報告されている [10]．具体的なフレームワークは以下である．

1. 対象とする Approximate ネットワークのパラメータを SimGrid のネットワークで設定する．
2. 本 Approximate 対応版 SimGrid Event Discrete において対象アプリケーションのシミュレーションにより最適化する．
3. 対象とする Approximate ネットワークにおいて，対象アプリケーションを最適化済みパラメータで実行する．

SimGrid Event Discrete シミュレータでは，MPI NAS 並列ベンチマーク 3.3.1 のクラス B，1,024 プロセス数までの大きさであれば，最新の計算機の 1 つを用いることで 1 つの実行あたり数時間以内で実行が完了する．したがって，本フレームワークは，この規模までの並列プログラムであれば現実的と考えられる．

7.5 本自動チューニングの探索限界

本自動チューニングの対象とする MPI プログラムのパラメータの探索空間の大きさの限界について検討する．これは，対象とする MPI 送受信対の数が大きく影響する．CG 法の場合は MPI 送受信対が 10 個あり，これにより 2^{10} 回の Approximate/完全データ転送の選択探索空間が生じる．加えて，各 MPI 通信のデータ型における誤りを許容するビット列の範囲を指定するビットマスク値についても探索空間が生じる．

一方，OpenTuner による 10^{100} 程度の探索空間での動作を仮定した場合，実行結果が必要とされる精度を満たさなかった場合に可能となる探索空間の枝刈りにより探索空間を小さくすることができることを考慮したとしても，対象とするプログラム中で推奨される MPI 送受信対は 300 個程度が 1 つの目安となる．

8 おわりに

一定のビット誤りを許容する Approximate ネットワークは，既存のネットワークと比較して広帯域かつ低レイテンシであるため，アプリケーションの高速化に有効である．しかし，Approximate ネットワークを活用した上で正しい計算結果を得るためには，アプリケーション側の対応が必要である．さらに，アプリケーション中で許容できる誤りの大きさと Approximate ネットワークのパラメータに依存して，実行性能は大きく変化する．そこで本研究では，Approximate ネットワークを用いてアプリケーションを実行する環境において，正しくない動作をした場合にペナルティを適切に挿入することによって高速に実行できるパラメータを発見する自動チューニング手法を提案した．

Approximate ネットワークにおいて，本提案自動チューニング手法はアプリケーションが許容範囲内の精度で実行することを保証する点で，極めて重要な技術である．256 プロセスを用いた共益勾配法の並列計算において，932 回および 1099 回の試行により，完璧な計算をした場合と比べて必要となる精度を満たしつつ，実行時間を約 10%削減という最速実行を実現した．

本報告では CG 法について詳細に解析を行ったが，今後は様々なアプリケーションに対して本自動チューニングを適用したい．幸い，我々はすでに文献 [1] にて高速フーリエ変換，K-means クラスタリング，行列計算を Approximate ネットワークに適用，初期評価を行っている．よって，これらに対してまずは自動チューニングを適用する予定である．

謝辞

本研究は科研費#16H02816 の支援による．

参考文献

[1] Daichi Fujiki, Kiyo Ishii, Ikki Fujiwara, Hiroki Matsutani, Hideharu Amano, Henri Casanova, and Michihiro Koibuchi. High-bandwidth low-latency approximate interconnection networks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, 2017.

[2] Luis Ceze. Disciplined approximate computing: From language to hardware, and beyond. https://homes.cs.washington.edu/~luisceze/ceze-approx-overview.pdf.

[3] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture*, pp. 14–26, 2016.

[4] S. Scott, D. Abts, J. Kim, and W. J. Dally. The BlackWidow High-Radix Clos Network. In *ISCA*, pp. 16–28, 2006.

[5] N. Tanabe, J.Yamamoto, H. Nishi, T. Kudoh, Y. Hamada, H. Nakajo, and H. Amano. Low latency high bandwidth message transfer mechanisms for a network interface plugged into a memory slot. *Cluster Computing*, Vol. 5, No. 1, pp. 7–17, 2002.

[6] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the new normal for computer architecture. *Computing in Science and Engineering*, Vol. 15, No. 6, pp. 16–26, 2013.

[7] K. S. Hemmert and J. S. Vetter and K. Bergman and C. Das and A. E. and C. Janssen and D. K. Panda and C. Stunkel and K. Underwood and S. Yalamanchili. Report on Institute for Advanced Architectures and Algorithms, Interconnection Networks Workshop 2008. http://ft.ornl.gov/doku/_media/iaaicw/iaa-ic-2008-workshop-report-v09.pdf.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 303–316, New York, NY, USA, 2014. ACM.

[9] SimGrid: Versatile Simulation of Distributed Systems. http://simgrid.gforge.inria.fr/.

[10] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, Vol. 74, No. 10, pp. 2899–2917, 2014.