ストリーム指向プログラムのための
並列性抽出アルゴリズムに関する研究

２０１９年3月

王 古玥

ストリーム指向プログラムのための
並列性抽出アルゴリズムに関する研究

王 古玥

システム情報工学研究科

筑波大学

２０１９年３月

目次

図目次

表目次

第１章
序論

１.１　研究背景

今の IoT 時代では、温度等の環境データ、画像や音声などのデータが、センサやデバイスを通じてストリームデータとして大量に、継続的に生成される。そのため、ストリームデータ処理も継続的に、かつリアルタイムに行う必要がある。

従来の処理では、全てのデータを溜めてから結果を計算するが、ストリームデータ処理では、入力されたストリームデータを溜めることなく、入力時点で処理を開始する。ストリームデータ処理により、例えば、店舗に設置したカメラから顧客の年齢や性別、行動などの、従来取得できなかった情報をリアルタイムに分析し、顧客の回遊率を向上させることなどが可能になる。また、製造業や設備管理などでは、センサデータをリアルタイムに分析し、異常予兆検知が可能になる。また、身近な例で見ると、Twitter や Facebook などの SNS では、メッセージやコメントなどがリアルタイムに投稿される。運営者がユーザのニーズに合わせて常に新しいトピックやニュースを配信する必要がある。駅などにある防犯監視カメラにもストリームデータ処理を活用すれば、リアルタイムに複数イベント処理（CEP：Complex event processing）で分析し、異常を検知することが可能になる。このように、ビジネスの問題を解決し、新たな可能性を見出せる。

上述のアプリケーションを開発するためには、流れてくるストリームデータを扱えるアルゴリズムを設計する必要がある。ストリームデータを処理するプログラムをストリーム指向プログラムと呼ぶ。プログラマがストリーム指向プログラムを作成し、用途に合わせてストリームデータを分析し処理する分析シナリオを書く必要がある。このようなプログラミングはストリーム指向プログラミングと呼ぶ。これは、様々なデータソースから続々と生成される大量の構造化・非構造化データをリアルタイムに分析する新しい計算パラダイムである[1]。

一方、近年のマルチコア/メニーコアアーキテクチャの急成長により、プログラムは、アプリケーションから並列性を抽出し、高速化を図る並列プログラミング手法が現れ、主流になった。演算処理の計算量が膨大な各種シミュレーション等、科学技術計算において高い処理性能が求められてきた。これらの需要を背景として、アクセラレータやメモリに関する高性能コンピュータ技術は発展してきた。アクセラレータとは、CPU の計算処理を支援し、特定の演算処理などを加速し、システム全体の処理性能を向上させるデバイスである。例え

ば、グラフィック処理に特化した GPU(Graphic Processing Unit)、デジタル信号処理に特化した DSP(Digital Signal Processor)[2]、内部構成を書き換え可能な FPGA(field-programmable gate array)[3]などがある。これらのデバイスのアーキテクチャはそれぞれ異なるが、計算処理の実行時間の長い部分を CPU に代わって高速に処理させることを目的にしていることは共通である。

GPU のようなアクセラレータでは、個々の計算対象データが各演算ユニットに割り当てられ、並列に計算される。例えば、$r_c = r_a + r_b$ のようにベクトルの和を求める場合、プログラマはその計算がベクトルの各要素に分離されることを考えなければならない。すなわち、$r_c[id] = r_a[id] + r_b[id]$である。ここで、id はベクトルの要素のインデックスである。各 $r_c[id]$の計算は各演算ユニットに割り当てられ、複数のベクトルの要素の和の計算が並列に実行される。演算ユニットの数が r_c の長さより大きい場合には、単一の「+」演算を計算するだけの処理時間で済む。このように、メニーコアアーキテクチャを利用することで、全体的な計算時間は大きく減らすことができる。プログラマは、各演算ユニットに割り当てられる各演算要素のインデックス化、および、依存性なく並列実行できる計算を考慮する必要がある。このようなインデックス化による処理スタイルは、ストリーム指向プログラミングの 1 つの形態である。そのため、GPU のようなアクセラレータを用いることで、膨大なストリームデータを低遅延で処理し、継続的に流入するストリームデータに対して高性能な処理能力を実現できる。

上述のように、インデックスベースの計算ができる GPU であれば、大規模なストリームデータを高性能な並列処理で処理することができ、さらなる性能向上を得ることが期待できる。

従来のストリーム指向プログラミングでは、Cg[4]や Brook for GPUs[5]のプログラミング言語のように、単一ストリームプロセッサでの小規模な問題の計算に限定されていた。しかし、GPU クラスタのような並列分散環境において、通信を含むため、従来のストリームコンピューティングのプログラミングインタフェースでの実装は簡単ではない。また、複数の専用プロセッサに適用される場合も効率的な実装は困難である。そのため、複数のアクセラレータを対象としたストリーム指向プログラミングに関しては、新しいプログラミング方式が必要である。

アクセラレータを用いた並列プログラミングでは、従来のプログラミングパラダイムと異なり、ホスト CPU によるアクセラレータの制御が不可欠である。すなわち、実行環境が異なる制御プログラムと計算プログラムの両方を記述する必要がある。アクセラレータ用のプログラム開発については、プログラミング言語およびランタイムが提供されている。例えば、アクセラレータの統合開発環境として、NVIDIA CUDA (Compute Unified Device Architecture) [6]や OpenCL (Open Computing Language) [7]などが提供される。しかし、これらの開発環境では、プログラマはアクセラレータ向けの並列プログラミングを行う前に、アクセラレータの種類と特徴、並列計算の様々なコンセプト、および、並列プログラミング言語

のコンセプトなどを理解する必要がある。さらに、最大の並列効率を引き出すには、実環境のハードウェア構成に適したプログラミングが必要となる。これらの手間はプログラマにとって低効率で困難な作業となる。そのため、アクセラレータを用いたストリーム指向プログラミングのユーザビリティを向上させる必要がある。

ストリームコンピューティングのプログラミングインタフェースを提供するプラットフォームとして、Caravela プラットフォームが提案された[8]。プログラマは、カーネルプログラムを含む flow-model というモジュールを作り、引数の I/O 関係とターゲットランタイムのタイプを指定することで、カーネルプログラムの開発に注力できる。カーネルプログラムとは、CUDA や OpenCL においての GPU 側に実行させるプログラムのことである。以降、カーネルと表記する。一般に、CPU の代わりにアクセラレータ上で実行するプログラムとして理解することもできる。また、CPU 側とアクセラレータ側の両方のプログラミングが必要な問題を解決するため、CarSh と呼ばれる Caravela フレームワークが、コマンドラインベースのプログラミングツールとして提案された[9]。CarSh は、executable または batch XML を読み込めるシェルのようなインタフェースを提供する。CarSh を使用することにより、プログラマはホスト CPU 側の制御プログラムを作成する必要がなくなり、batch ファイルのみを作成すれば実行できる。CarSh フレームワーク上に、アプリケーションを開発するための GUI も実装された[10]。Flow-model をノードとして、データストリームを接続用の矢印として使用することで、プログラマは複数の flow-model を矢印で接続し、全体として一つの処理フローを構築できる。このように、ストリーム指向プログラミングのプログラマビリティの向上を目指してきた。

しかし、継続的、かつ大量に流入するストリームデータを効率よく処理するために、処理フローの中の並列性を最大限に抽出する必要がある。並列性を抽出する際、プログラマはハードウェアの構造や、並列計算のコンセプトを理解する必要がある。そのため、並列ストリームコンピューティングはプログラマにとって極めて困難な作業になっており、並列ストリームコンピューティングにおけるプログラマビリティ向上は一つ大きな課題となる。

1.2 研究目的

プログラムの持つ並列性にはビットレベルや命令レベルの並列性から、データ並列性、タスク並列性まで様々なレベルがある。GPU のようなアクセラレータの場合、GPU 内の複数の演算回路間からカーネル間までの並列性がある。その中で、カーネル間の並列性は複数のアクセラレータによって性能を引き出すことができる。

アクセラレータ上でストリーム指向プログラムを開発する際、高性能を達成するために、アクセラレータの持つ並列性を十分に引き出す高度なプログラミングが必要となる。また、カーネルレベルの並列性を最大限に抽出するには、アクセラレータを搭載したハードウェアの構成に関する知識も必要となる。この点に関しては、ハードウェアを抽象化・標準化す

る OpenCL などのようなアクセラレータ向けのプログラミングインタフェースを利用することで、ハードウェアの詳細を隠蔽し、一定の高性能化が可能である。一方、現実の並列分散環境に合わせてカーネル間の並列性を抽出し利用することは、プログラマにとって極めて困難な作業となる。さらに、複数のアクセラレータ間の通信が必要な場合、プログラミングがさらに複雑となる。

そこで、本研究では、ストリーム指向プログラムの開発の負担を軽減することに着目し、プログラマが複数のアクセラレータの高い性能を引き出すために、下層ハードウェアの構成に関する知識を持たなくても、カーネル間の並列性を自動的に引き出すことを主な目的とする。

その目的を達成するために、複雑な依存関係を持つ処理フローから最適な順序を決定し、並列性を抽出する。処理フローが単純な場合、プログラマはカーネル間の並列性といくつかの並列実行パターンを簡単に見つけることができる。しかしながら、処理フローが大規模または複雑になると、並列、もしくはパイプライン方式で処理フローを実行するための最適な順序を決定することは困難である。

そのため、本研究では、複数のカーネルで構成された静的な処理フローから、カーネル間の空間的および時間的な並列性を抽出できる新たなアルゴリズム PEA-ST を提案する。また、複数のアプリケーションに本アルゴリズムを適用することで性能評価を行う。

1.3 本研究の貢献

本研究の貢献を以下に示す。

- アクセラレータの高い演算性能を利用するために、プログラマは並列計算の様々なコンセプトを理解する必要があり、それに連れてプログラマの負担も増加している。ストリームコンピューティングにおけるプログラマビリティを向上させるために、本研究では、並列性抽出アルゴリズム PEA-ST を提案した。本アルゴリズムでは、データ並列化、タスク並列化とパイプライン化により、スループットの向上とレイテンシの削減で、高い計算能力を実現できることを示した。
- 三つのアプリケーションを用いて本アルゴリズムの評価を行った。本アルゴリズムにより、大量の処理データを持ち、かつ複数回の演算を必要とするストリーム指向プログラムに向けた、GPU のようなアクセラレータに応用でき、高性能化を実現できることを示した。その結果、PEA-ST アルゴリズムを用いることで、プログラマが並列実行環境に合わせてカーネル間の実行順序を設計する必要がなくなり、自動並列化により一定の並列効率が得られることを示し、性能を効果的に改善できることを示した。
- 本アルゴリズムの並列パターンと通信パターンの自動生成機能により、プログラマが下層ハードウェアの構成を知ることなく効率良いプログラムを開発できる環境を

整備した。すなわち、データストリームで接続された処理フローがあれば、コードの中に明示的に並列処理に関する記述を書かなくても、PEA-ST アルゴリズムを用いることで自動的に並列化できることを示した。通信パターンの最適化により、処理フローにフィードバックがある場合にも対応できる。さらに、静的なロードバランスも考慮し、さらなる性能向上も得られることを示した。

1.4　本論文の構成

本論文は以下のように構成されている。

第 2 章では、並列ストリームコンピューティングを本研究の背景として詳しく述べる。まず、並列計算におけるユーザビリティについて概説する。マルチコア/メニーコアアーキテクチャや、アクセラレータでの並列コンピューティング環境について述べる。次に、ストリームコンピューティングのプログラマビリティについて概説する。ストリームコンピューティングの現状について述べる。また、Caravela プラットフォームについて述べる。コマンドラインベースのプログラミングツール CarSh および開発用 GUI も概説する。最後に、並列プログラミングにおける課題について論ずる。

第 3 章では、並列性抽出アルゴリズム PEA-ST を提案し、本アルゴリズムの設計と実装について述べる。まず、並列性抽出方法として Spanning Tree の詳細を述べる。Spanning Tree の定義、および、Spanning Tree を生成するアルゴリズム DFST について述べる。続いて、アルゴリズムの全体の流れと実装を示す。最後に、まとめを述べる。

第 4 章では、並列性抽出アルゴリズム PEA-ST の性能最適化について述べる。まず、計算環境に対応した並列化手法や通信パターンの自動生成手法を述べる。次に、それらの機能を実現するための本アルゴリズムへの追加実装について述べる。さらに、通信パターン生成における最適化とロードバランスの最適化についても述べる。

第 5 章では、PEA-ST アルゴリズムに関連する研究について述べ、本論文で提出した自動並列化手法と最適化手法の特徴と優位性について比較しながら論ずる。

第 6 章では、PEA-ST アルゴリズムの有効性と性能最適化に関する評価を行う。本アルゴリズムの有効性は FFT を用いた画像フーリエ変換、LU 分解の二つのアプリケーションを用いて示す。本アルゴリズムの性能最適化による効果については、k-means と前述の二つのアプリケーションを用いて評価する。

最後に、第 7 章で本論文のまとめと今後の課題を述べる。

第２章
並列ストリームコンピューティング

２.１　並列計算のユーザビリティ

ユーザビリティ（usability）とは、一般にはユーザに対してある製品やソフトウェアなどの使いやすさや使い勝手といった意味で使われることが多い。異なる分野では、ユーザビリティの定義は多少異なる。代表的なユーザビリティの定義としては、国際標準化機構によるISO9241-11 や、ヤコブ・ニールセンによる定義が挙げられる。ISO9241-11 では、ユーザビリティを「特定の利用状況において、特定のユーザによって、ある製品が、指定された目標を達成するために用いられる際の、有効さ、効率、ユーザの満足度の度合い」と定義されている[11]。また、「ユーザビリティエンジニアリング原論」で、ヤコブ・ニールセンは「ユーザビリティは学習しやすさ、効率性、記憶しやすさ、エラーと主観的な満足度の 5 つの要素を持つ」と定義した[12]。この定義は「スモールユーザビリティ」とも言われている。上述の 2 つの定義から、ユーザに無駄な負担をかけさせない、効率を向上させることをユーザビリティの重要な要素として捉える。

計算コンピューティング領域では、プログラマは、1 つのユーザとして開発言語やツールを利用し、システムやソフトウェアの開発を行う。その際、プログラマがスムーズかつ効率的にプログラム開発できることは非常に重要になる。上述のユーザビリティの定義から、プログラマが如何に効率的にプログラムを書くことができるかは、開発言語やツールの使いやすさによって決まる。優れた開発環境は、プログラムの読み取り、書き込み、保守が容易であり、より柔軟で強力なプログラムを書くことをサポートでき、安全性も確保できる必要がある[13]。そのため、多くの研究者たちはプログラマのユーザビリティを向上させる方法を追求してきた[14]。

それに合わせたプログラミングの発展により、プログラミングパラダイムは最も基本的で単純な逐次型プログラミングから、命令型プログラミング[15][13]、構造化プログラミング[16]、関数型プログラミング[17]、オブジェクト指向プログラミング[18]まで様々なパラダイムがある。近年、プログラミングパラダイムがさらに進化し、ビジュアルプログラミングとデータフロープログラミング[19]も現れた。また、複数のパラダイムを含むマルチパラダイムプログラミングモデルも存在する。

次に、各プログラミングパラダイムの定義や、開発、具体例などについて述べる。

(1) 逐次型プログラミング

逐次型プログラミングとは、プログラマがハードウェアに実行させたい処理を順次に書き、一度に一つの処理が順次に動作することを意味する。これと対照的なプログラミングパラダイムが並列プログラミングである。一般的に、同じ入力データが与えられると、逐次型プログラムは常に同じ線形命令列を実行し、常に同じ結果を生成する。すなわち、逐次型プログラムの実行は確定的である。

歴史的に最初に出現したアセンブル言語は最もローレベルの逐次型プログラミング言語である[20]。機械語は、プロセッサが直接に実行できる言語であるが、実行したい計算をハードウェアに依存し、単純な操作に分割したもので、人間として理解しにくいである。そのため、機械語で直接プログラムを書くことは、プログラマにとって大きな負担になる。そこで、機械語を直接記述するのではなく、簡略化した英単語や記号に対応させたプログラミングができるアセンブリ言語が開発された。ただし、ハイレベルなプログラミング言語と異なり、アセンブリ言語はプロセッサに依存する。また、アセンブリ言語で記述されたプログラムを自動的に機械語へ変換するコンパイラをアセンブラと言われる。アセンブリ言語は、プログラミングの入力エラーやプログラミング時間を減少することができる。そして、プログラムは面倒な作業をする必要がなくなった。

(2) 命令型プログラミング

命令型プログラミングとは、問題を解決ためのプロセスをコンピュータが実行すべき命令列として記述することを意味する。実行順序により、計算すべき命令や手続きは命令型プログラムに記述される。そのため、一般的に命令型プログラミングは、手続き型プログラミングとも扱われる。

命令型プログラミング言語は、機械語から発展されてきた。上述したように、機械語は単純な命令のみ実行できるため、プログラマが複雑なプログラムを構成するには困難であった。そのため、一つずつで順に記述した基本的な命令列から、複数の命令をまとめた手続きをより高いレベルの一つの命令で呼び出す技術が出現した。1954 年から、初めての高レベルプログラミング言語として、IBM のジョン・バッカスが開発された FORTRAN が登場した[15]。FORTRAN は、命令型プログラミング言語で一般的な他の多くの機能を可能にするコンパイル言語である。1960 年代前後の間、ALGOL は数学的アルゴリズムをより簡潔に表現できるように開発された[21]。その後、BASIC[22]や Pascal[23]、C 言語[24]も手続き型プログラミング言語として登場した。これにより、機械語で記述するのが難しい、複雑で大規模なプログラムを開発するのがより簡単になった。

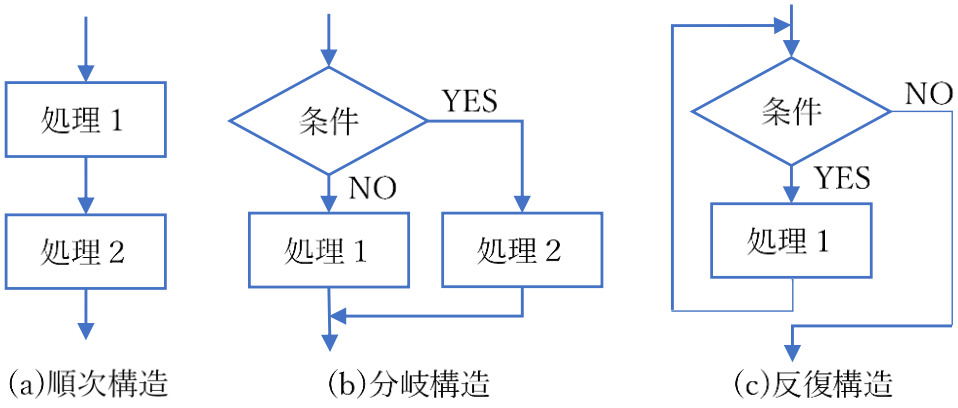


図 2.1　順次、反復、分岐の三つの基本制御構造

(3) 構造化プログラミング

構造化プログラミングとは、大規模なプログラムを作成する際に、全体的な処理を細かな単位に分割し、それらの処理を連鎖し、階層的な構造となるようにプログラミングを行うことである。構造化プログラミングは、ダイクストラが 1969 年の NATO のソフトウェア工学会議で提出されたプログラミングパラダイムである。大きなプログラムでも正しさの証明ができる良く構造化されたプログラムや、段階的な抽象化と詳細化、抽象化データ構造とその上で操作する抽象化文との共同詳細化、および、プログラムの階層化について述べた[16]。

また、別の理論として、構造化定理により、全てのアルゴリズムは、順次、分岐、反復の三つの基本制御構造を組み合わせて表現できる[25]。当然、この三つの要素も構造化プログラミングの一部である。図 2.1 に示すように、順次構造はプログラムに書かれた順で逐次処理を行う構造である。反復構造は同じ処理を繰り返す場合に使われる。分岐構造はある条件を判断し、該当する処理を行う構造である

典型的な構造化プログラミング言語は、C や Pascal 言語などである。これらの構造化プログラミング言語により、プログラムの構造を明確にでき、アルゴリズムによって生じる間違いを減少することができ、ソフトウェア開発の効率を向上させることができる。

(4) 関数型プログラミング

関数型プログラミングとは、解決すべき問題に対し、問題の性質を関数の組み合わせで記述するプログラミングパラダイムである。すなわち、関数だけで全体が構成されているプログラムを意味する。メインプログラムは、プログラムへの入力を引数として受け取り、プログラムの出力を結果として供給する関数を書く[26]。

図 2.2　子ども向けのビジュアルプログラミングソフト Scratch

最初の関数型プログラミング言語は LISP 言語である。この言語の特徴は純粋な関数型モデルを持つことである[27]。その後、宣言型の静的型付けコードを持つ、高度で純粋な関数型プログラミング言語として Haskell 言語がリリースされた[28]。2013 年から、共通言語ランタイムの出現と伴い、Scala[29]、Clojure[30]、F#[31]などの言語が開発された。

論文[26]により、関数型プログラミングの特別な特性と利点は、次のように要約されている。関数型プログラムは代入文を含まれていないので、一度値が与えられると変数は決して変更されない。関数呼び出しはその結果を計算する以外には効果がないため、主要なバグの発生源が排除される。そのため、副作用が式の値を変更することはないため、いつでも評価できる。これにより、プログラマは制御の流れを規定する負担から解放される。さらに、関数型プログラムは従来のものより数学的な扱いやすくするのに役立てる。

(5) オブジェクト指向プログラミング

オブジェクト指向プログラミングは、オブジェクト指向の概念を用いたプログラミングパラダイムの一種である。オブジェクト指向プログラムでは、データとメソッドを一つのオブジェクトとしてまとめて扱い、多数のオブジェクトで構成され、カプセル化、継承、多態性と動的束縛の特性を持っている。カプセル化はデータやオブジェクトの振る舞いを隠蔽することを指す。継承により、あるオブジェクトが他のオブジェクトの特性を引き継ぐことができ、コードを再利用できる。多態性は、異なるタイプの実体への単一のインタフェースを提供する、または、複数の異なるタイプを表

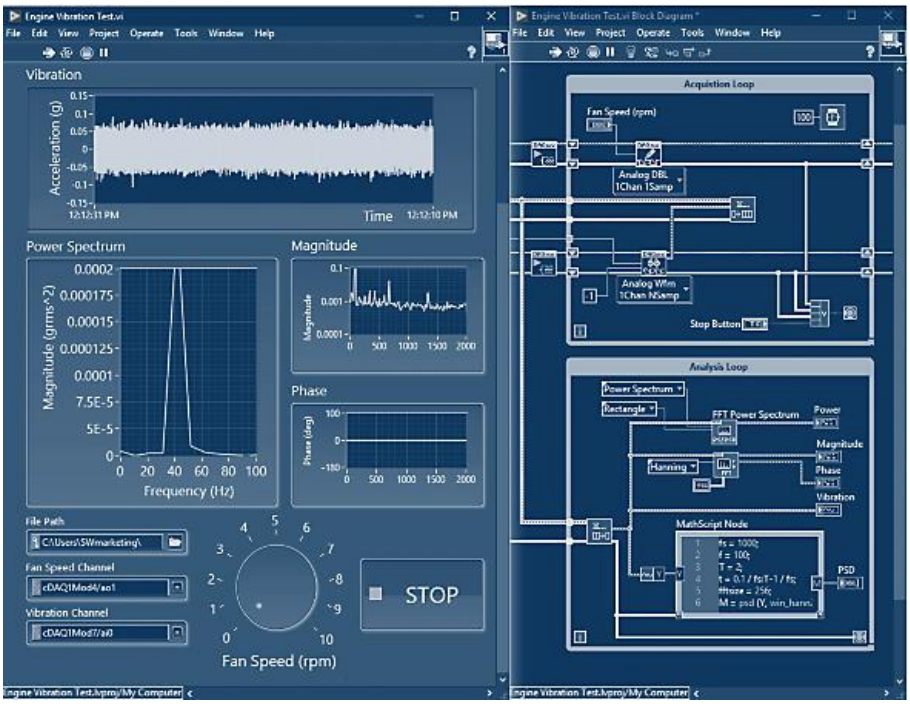


図 2.4　LabVIEW 2018 インタフェース

すために単一シンボルで標記できる性質である。動的束縛は、オブジェクトにより呼び出されるメソッド、または、引数付きで呼び出される関数が実行時に名前によって検索されるコンピュータプログラミングメカニズムである。

オブジェクト指向を最初に取り入れたプログラム言語は、1970 年代にアラン・ケイが開発した Smalltalk である[18]。C++[32]や Java[33] などの C 言語をベースとしたオブジェクト指向機能を持つ言語は、オブジェクト指向プログラミング用に設計された代表的なプログラミング言語である。

プログラムの内部動作や操作手順の詳細を知ることなく利用でき、また、コードの再利用が容易になるため、ソフトウェアの設計や開発のコストを軽減できる。

(6) ビジュアルプログラミング

ビジュアルプログラミングとは、従来のコードでプログラムを書くことではなく、ビジュアルな操作や直感的なインタフェースでプログラミングできるプログラミング方式である。ほとんどのビジュアルプログラミング言語は GUI を提供する。

この方式により、プロのプログラマではなく、一般人にも簡単なプログラミングができる。ロボット工学や音楽制作などの特定の領域に応用され、特定のプログラミングが非常に簡単になる。近年、図 2.2 のような子ども向けのビジュアルプログラミングソフトも現れ、ドラッグとドロップといった簡単な操作で実際のプログラミングを体験できる[34]。

(7) **データフロープログラミング**

データフロープログラミングとは、データフローの概念を用いて、有向グラフでデータと処理の流れを表すことによりプログラムを作ることができるプログラミングパラダイムである。そのプログラムは入力および出力ポートを持つブロックの集合で構成される。これらのブロックはシステム内に流れている情報のソースや、処理ブロックなどを含む[35]。

Progprah 言語はより根本的なデータフロー言語であり、入力から出力に線を引くようにグラフィカルにプログラムを構築できる[36]。図２.3 のように、LabVIEW 言語はより実用的なもので、ハードウェア構成、計測データ、デバッグといった、アプリケーションのあらゆる側面を視覚化するグラフィカルプログラミングアプローチである[37]。

ビジュアルプログラミングとデータフロープログラミングの融合は、プログラムを効率的に構成でき、デバッグや自動プログラム修正などが容易になる手段である。

上述では、様々な基本的なプログラミングパラダイムについて述べたが、マルチコア/メニーコアアーキテクチャの急成長により、新しいプログラミングパラダイムが登場した。アプリケーションからスレッドレベルの並列性を抽出し、高速化を図る並列プログラミング手法が現れ、主流になってきている。さらに、LSI（大規模集積回路）チップの CPU コアの数が増加しており、複数のスレッド/タスクを実行できる環境は、デスクトップ、ラップトップコンピュータ、モバイルデバイスでも使用可能になった。このように、マルチコア技術に関する並列分散コンピューティング技術[38]は生活の中で様々な場所で利用されるようになった。

マルチコア/メニーコアは、シングルコアと言った単一のコアしか持っていないプロセッサを発展させて、単一のチップパッケージに複数のプロセッサコアを集積する技術のことである。一般的には、マイクロプロセッサでは、パッケージの中に命令レジスタや演算器などを組み合わせた、1 つの部品として動作するプロセッサコアをワンセットとして持っている。マルチコアプロセッサは、複数のプロセッサコアを用いて、複数のマイクロプロセッサを搭載している。プロセッサコアレベルの並列性を利用することで、性能を向上させることができる。また、コア数が 2 個であればデュアルコアと呼ばれ、4 個であればクアッドコアと呼ばれる。数十個以上のコアを持つ高性能な専用プロセッサがマルチコア/メニーコアと呼ばれる[39]。

1999 年に IBM から発表された POWER4 は商用サーバ向けのプロセッサとしたデュアルコアプロセッサであり、CPU のマルチコア化を先導した[40]。そして、2005 年に、AMD は大幅にメモリのレイテンシを低減できるメモリ・コントローラを搭載する K8 アーキテクチャを発表し、デュアルコア製品を提供し始めた。さらに、Power Architecture シリーズでのは、

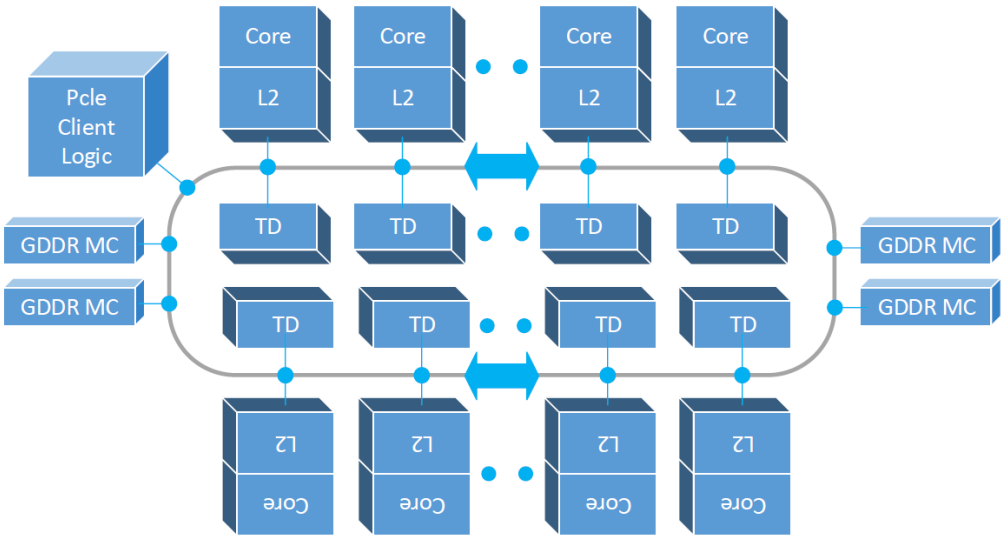


図 2 .5　インテルの Xeon Phi のマイクロアーキテクチャ

2006 年に発表された 8 個のコアを持つ Cell [41]や、2010 年に発表された 8 個のコアを持つ POWER7[42]などがある。続いて、2012 年 8 月に開催された Hot Chips 会議で第 3 セッション「メニーコアと GPU」の 3 つのプレゼンテーションでも明らかにされたように、マルチコアからメニーコアへの移行が進んでいた[43][44][45]。このように、マルチコア/メニーコアプロセッサの概念が一般的になり[46]、プロセッサのコア数がさらに増加している[47]。前述の会議で、インテル社の George Chrysos により、同社の Xeon Phi チップが発表された[45]。図 2 .5 に示すように、このデバイスは、50 個以上の x86 プロセッサコアと 4 個の GDDR メモリと PCI Express 接続を搭載したコプロセッサである。また、各プロセッサはベクトル演算装置と 512 KB の L2 キャッシュを搭載し、双方向環状バスによってリンクされている。Xeon Phi により、マルチコア/メニーコアへの進化がさらに飛躍した。

マルチコア/メニーコアプロセッサはペタフロップス級の演算処理性能を有するスーパーコンピュータおよび小規模スパコンでの高性能計算、情報家電や画像処理などのメディア処理、ロボットなどでの組込みシステムや制御等に広く利用されている。そのため、特定の演算処理などを加速できるハードウェアとしてアクセラレータの概念が出現した[48]。アクセラレータとは、CPU の計算処理を支援し、特定の演算処理などを加速し、システム全体の処理性能を向上させるデバイスである。例えば、グラフィック処理に特化した GPU(Graphic Processing Unit)、デジタル信号処理に特化した DSP(Digital Signal Processor)[2]、内部構成を書き換え可能な FPGA(field-programmable gate array)[3]などがある。これらのデバイスのアーキテクチャはそれぞれ異なるが、計算処理の実行時間の長い部分を CPU に代わって高速に処理させることを目的にしていることは共通である。

図 2 .6 では、NVIDIA GPU と Altera OpenCL Accelerator の 2 つのメニーコアコンピューティングシステムのアーキテクチャ例を示している。メニーコアアーキテクチャでは、LSI に大規模なコンピューティングユニットが実装されている。GPU の場合を図 2 .6(a)に示す。

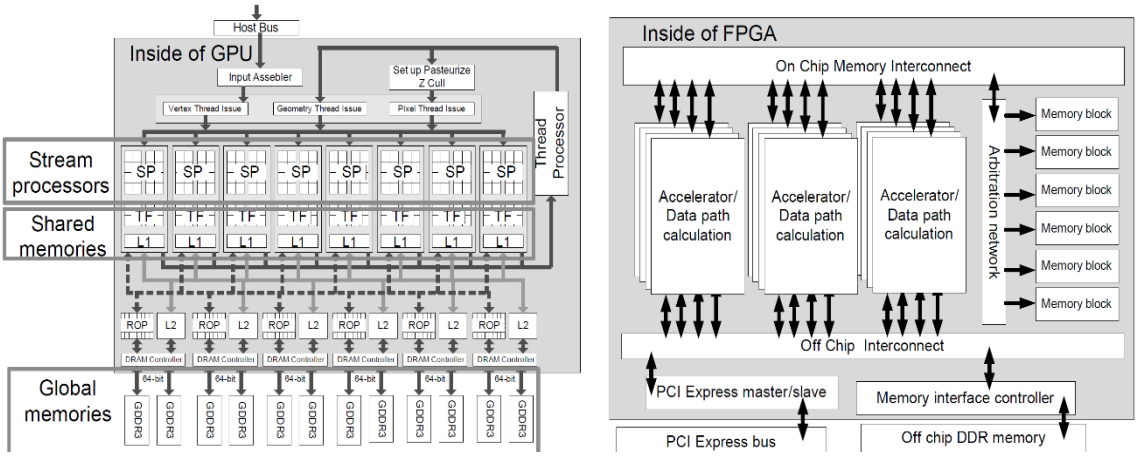


図２.6　メニーコアアーキテクチャの例：NVIDIA GPU と Altera OpenCL Accelerator

数百から数千の再構成可能なハードウェアアクセラレータを搭載している。そして、図２.6(b)に示されているのは OepnCL フレームワークのためのアクセラレータアーキテクチャである[48][49]。GPU の場合と同様に、アクセラレータが並列に動作することにより、高性能化を実現する。

チップあたりのトランジスタ数の増加に関連するムーアの法則により、マルチコア/メニーコアアーキテクチャの発展が上述のように促進されている。特にエンターテイメント市場における動的グラフィックスの高いフレームレートを達成するための高速な計算に活用されている[50]。このようなグラフィックス処理は、小型の処理ユニットによって実行される。その処理ユニットは、ピクセルを計算するために割り当てられ、数百の処理ユニットによって同時処理される。このようなグラフィックス処理要求に応じて、メニーコアアーキテクチャが発展してきた。

また、シンプルな演算ユニットを多数搭載している GPU は、CPU と比べて性能比で低価格、かつ、高いピーク演算性能を持っている。そのため、高度な演算密度や並列性を持つ処理を行う場合、GPU はより高い処理性能を得ることができる。その高い演算性能を画像処理以外の領域に応用できるために、NVIDIA は GPU 向けの C 言語の統合開発環境として CUDA[6]を開発した。このように、GPU の特性を汎用的に活用する技術は GPGPU (General-Purpose computing on Graphics Processing Units)[51]と呼ばれる。低価格で高い演算性能を持つ GPU は、高性能コンピューティングプラットフォームを実装するために不可欠なツールの 1 つとなっている。

アクセラレータを用いた並列プログラミング方式では、拡張バスで CPU に結合されているため、従来のプログラミングパラダイムと異なり、ホスト CPU によるアクセラレータの制御が不可欠である。すなわち、実行環境が異なる制御プログラムと計算プログラムの両方を記述する必要がある。アクセラレータ用のプロフラム開発については、プログラミング言語およびランタイムが提供されている。例えば、GPGPU 技術の統合開発環境として、NVIDIA

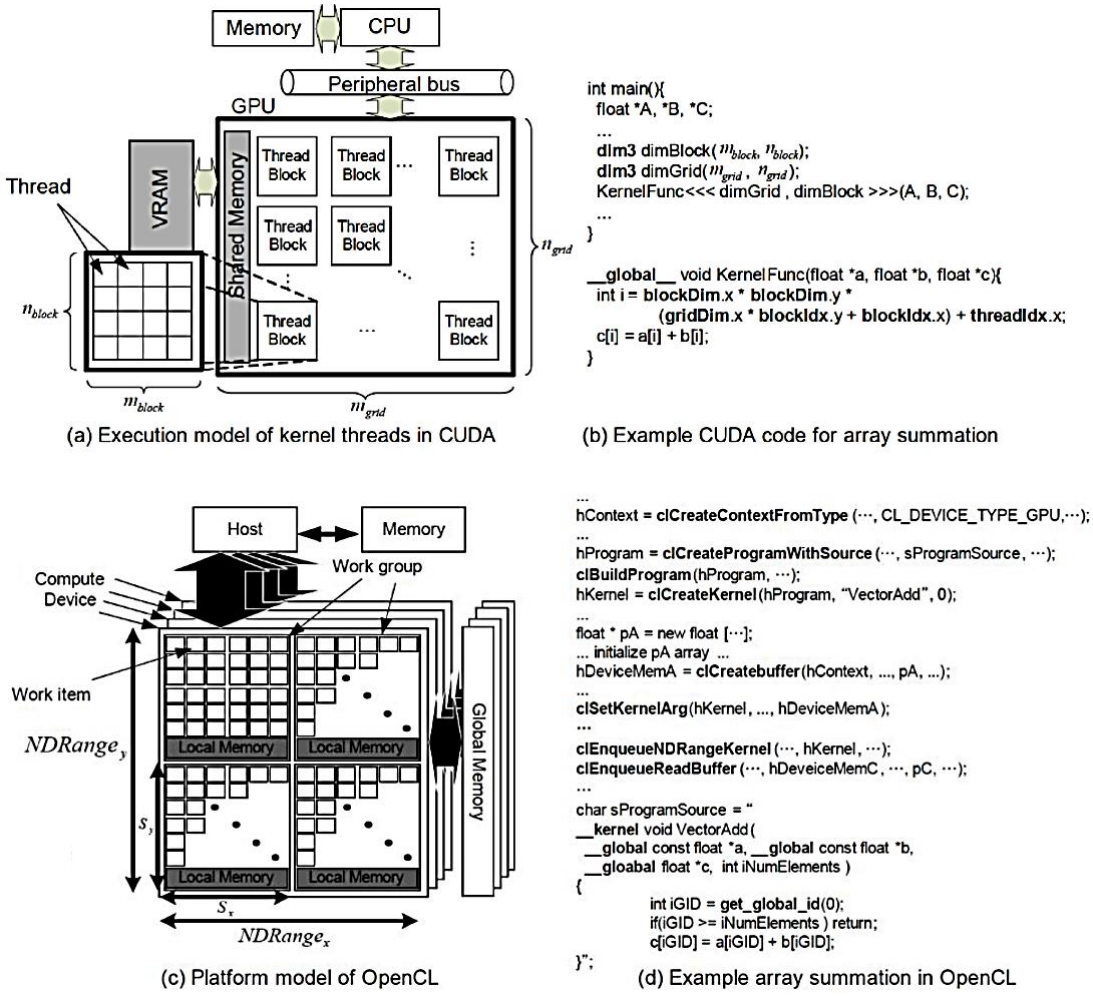


図 2.7　CUDA と OpenCL のアーキテクチャモデルとベクトル和の例

CUDA(Compute Unified Device Architecture) [6]や OpenCL(Open Computing Language) [7]などが提供される。これらのプログラミングインタフェースは、アクセラレータ側のカーネルプログラムのためのプログラミング言語、および、カーネルプログラムの実行タイミングとプログラムが入力/出力データを制御するためのランタイム関数を提供する。

CUDA は、NVIDIA が開発された GPU 向けの C 言語ベースの統合開発環境であり、コンパイラやライブラリなどが提供されている。CUDA では、まず、データをメインメモリから GPU 用メモリにコピーし、次に、CPU から指示された処理に応じて、GPU 用メモリから計算データを読み取って各演算コアまで並列実行し、最後に、計算された結果をメインメモリにコピーし、計算を完了する。

CUDA は、図 2.7 (a)に示すようなアーキテクチャモデルを想定している。モデルは、CPU の周辺バスに接続されている GPU を定義する。VRAM は、GPU 上の計算に使用されるデータを保持する。カーネルプログラムは、ホスト CPU によって GPU にダウンロードされ、データもホストメモリからコピーされる。プログラムは、複数のスレッドがグループ化され、

スレッドブロック内の 1 つのスレッドとして実行される。スレッドブロックは 1 次元から 3 次元までの行列にタイル化されている。図２.7 では、スレッドブロックが二次元にタイル化される様子を示す。グリッドサイズは $n_{grid} \times m_{grid}$ である。各スレッドブロックは、$n_{block} \times m_{block}$ 個のスレッドで構成される。図２.7(b)に示されているプログラムは CUDA を使って書かれたベクトル和の例である。カーネルプログラムは GPU 上で実行されるように__global__指令を使用して定義されている。カーネル内では、CUDA ランタイムによって宣言される gridDim、blockDim、blockIdx、threadIdx のグローバル変数が、それぞれ、グリッドのサイズ、スレッドブロックのサイズ、スレッドブロックのインデックス、スレッドのインデックスを指定するために使用される。このカーネル関数は、スレッドの数を指定するホスト CPU プログラムによって<<< >>>記号を用いた指令で呼び出される。

一方、OpenCL（Open Computing Language）は、OpenCL C 言語の開発環境であり、CPU や GPU、FPGA、およびその他のアクセラレータで構成される異種プラットフォーム間で実行できる並列コンピューティングのためのフレームワークである。データおよびタスクベースの並列処理を利用できる標準インタフェースを提供する。

OpenCL は、要素プロセッサおよびメモリ階層を含む一般的なプラットフォームのモデルを定義する。図２.7(c) では、要素プロセッサのためのプラットフォームモデルを示している。ホスト CPU は、OpenCL のデバイスに接続されている。OpenCL のデバイスは、計算ユニットと呼ばれる個々の要素プロセッサで構成される。計算ユニットに 1 つまたは複数のワークグループを含んでいる。さらに、ワークグループは複数のワークアイテムを含む。ワークアイテムは唯一の ID によって識別され、その ID に関連する入力データを処理する。ワークアイテムの数は、プログラムによってパラメータ NDRange で与えられる。それを 1 次元から 3 次元までに定義できる。図２.7(c)では、$NDRange_x \times NDRange_y$ 個のワークアイテムがある。OpenCL のプログラムは、ホスト CPU 側により、C 言語で記述されている。OpenCL のリソースは、ランタイム関数で作成されたコンテキストによって得られる。図２.7(d)の場合、始めに GPU のためのコンテキストは、CL_DEVICE_TYPE_GPU を引数として指定することにより定義される。この引数に使用することで、異なるタイプのアクセラレータを選択できる。次に、カーネルプログラムが char の配列として定義されるソース文字列によって提供されるため、その文字列はランタイム関数に渡され、アクセラレータでの実行可能なコードにコンパイルされる。

そして、I/O データストリーム用のバッファは、CPU 側の「new」または「malloc」のような従来の関数で割り当てられる。プログラマは、バッファをアクセラレータから直接アクセスすること、または、バッファミラーをアクセラレータのメモリに割り当てることを選択できる。その後に、ホストデバイスにある実際のバッファをポイントしているカーネル関数の引数ポインタはアクセラレータに渡される。最後に、カーネルが clEnqueueNDRangeKernel 関数によって実行され、アクセラレータのメモリにある出力データストリームがホスト CPU 側にコピーされる。

OpenCL と比べて、CUDA はより柔軟に抽象化され、カーネルの発行を一般の関数呼び出しに近い簡潔な表記で記述できる。この利点に対し、OpenCL では、カーネルが OpenCL API により発行するため、ランタイムを準備する作業が必要である。しかし、異なるアクセラレータに共通のインタフェースを提供するクロスプラットフォームな OpenCL と比べて、CUDA は NVIDIA 製の GPU のみのサポートである欠点があり、アプリケーションの移植が困難である。

また、OpenMP[52]、OpenACC[53]、HPF(High Performance Fortran)[54]などの並列プログラミング言語もある。これらを使用した並列プログラミングでは、データ分割などをプログラマが明示的に指示する必要がある。

上述のように、プログラマはアクセラレータ向けの並列プログラミングを行う前に、アクセラレータの種類と特徴、並列計算の様々なコンセプト、および、並列プログラミング言語のコンセプトなどを理解する必要がある。高性能化を目指すには、逐次型プログラムを人手により並列化する場合や、アルゴリズム外部の並列性を考慮する場合などもある。さらに、最大の並列効率を引き出すには、実環境のハードウェア構成に適したプログラミングが必要となる。これらの手間はプログラマにとって低効率で困難な作業となる。そのため、並列計算のユーザビリティは向上させる必要があり、非常に重要な課題となる。

２.２　ストリームコンピューティングにおけるプログラマビリティ

一方、今の IoT 時代では、温度等の環境データ、画像や音声などのデータが、センサやデバイスを通じてストリームデータとして大量に生成される。ストリームデータは流入するデータを入力、流出するデータを出力として扱う抽象データ型である[55]。そのようなストリームデータは、膨大なデータソースから継続的に生成され。そのため、ストリームデータ処理も継続的に、かつ時間の経過とともにストリームデータの変化などをリアルタイムに計算し、更新処理を行う必要がある。

従来の処理では、全てのデータを溜めてから結果を計算するが、ストリームデータ処理では、入力されたストリームデータを溜まることなく、入力された時点で処理を開始する。この特性から、様々な分野でシステム上の検知や分析には適している。例えば、Twitter や Facebook などの SNS では、メッセージやコメントなどをリアルタイムに投稿される。ユーザのニーズに合わせて常に新しいトピックやニュースを配信し、古いトピックを除外する必要がある。このように、ストリームデータ処理は、随時生成されるデータをリアルタイムに分析し処理するような応用にに向いている。また、駅などにある防犯監視カメラに対しても、ストリームデータ処理を活用すれば、リアルタイムに複数イベント処理（CEP）で分析し、異常を検知することができる。

上述のアプリケーションを開発するために、流れてくるストリームデータを扱い、アルゴリズムを設計する必要がある。ストリームデータを処理するプログラムをストリーム指向プログラムと呼ぶ。プログラマがストリーム指向プログラムを作成し、用途に合わせてストリームデータを分析し処理する分析シナリオを書く必要がある。このようなプログラミングはストリーム指向プログラミングと呼ぶ。様々なデータソースから続々と生成される大量の構造化・非構造化データをリアルタイムに分析する新しい計算パラダイムである[1]。膨大なストリームデータを低遅延で処理するために、継続的に流入するストリームデータに対して高性能な処理能力を必要とする。そのストリームデータを低遅延で処理することをストリームコンピューティングと呼ぶ。データの処理及び分析の速度と精度を高めるため、次から次へと流れてくる大量で多様なデータを、保存することなく、リアルタイムにサーバで処理する技術である。

また、ストリームプロセッシングは、データフロープログラミングやイベントストリーム処理と同等のプログラミングパラダイムである。これにより、アプリケーションは一定の並列処理をより簡単に活用できる。ストリームデータが与えられると、カーネル関数がストリーム内の各要素に適用される。GPU や FPGA などの複数の計算ユニットを使用し、割り当て、同期、または通信を明示的に管理する必要はない。

図２.6(a)に示すように、GPU において演算器の最小単位をストリームプロセッサと呼ぶことがある。ストリームプロセッサは、アプリケーションによって与えられ、対応するデータ要素に対する個々の計算に対応するコンピューティングユニットとして機能する。すべてのストリームプロセッサに並列実行を行われることで、高い計算性能を実現できる。GPU のようなメニーコアアーキテクチャでは、個々の計算対象データが各演算ユニットに割り当てられ、並列に計算される。例えば、$r_c = r_a + r_b$ のようにベクトルの和を求める場合、プログラマはその計算がベクトルの各要素に分離されることを考えなければならない。すなわち、$r_c[id] = r_a[id]+r_b[id]$である。ここで、id は、ベクトルの要素のインデックスである。各$r_c[id]$の計算は各演算ユニットに割り当てられ、複数のベクトルの要素の和の計算が並列に実行される。演算ユニットの数が r_c の長さより大きい場合には、単一の「+」演算を計算するだけの処理時間で済む。このように、メニーコアアーキテクチャを利用することで、全体的な計算時間は大きく減らすことができる。プログラマは、各演算ユニットに割り当てられる各演算要素のインデックス化、および、依存性なく並列実行できる計算を考慮する必要がある。このようなインデックス化による処理スタイルは、ストリームコンピューティングの 1 つの形態である。すなわち、GPU などのアクセラレータはストリームコンピューティングにおける補助であり、性能を向上させることができる。

2007 年に IBM 社により、System S と呼ばれる体系化されていない大量のデータをリアルタイムに分析するストリームコンピューティングシステムが発表された[56]。System S のソフトウェアは、計算タスクを分割し、結果を再構成することができる。その後、Streaming Analytics on IBM Cloud や IBM Streams Quick Start などのストリームコンピューティング製

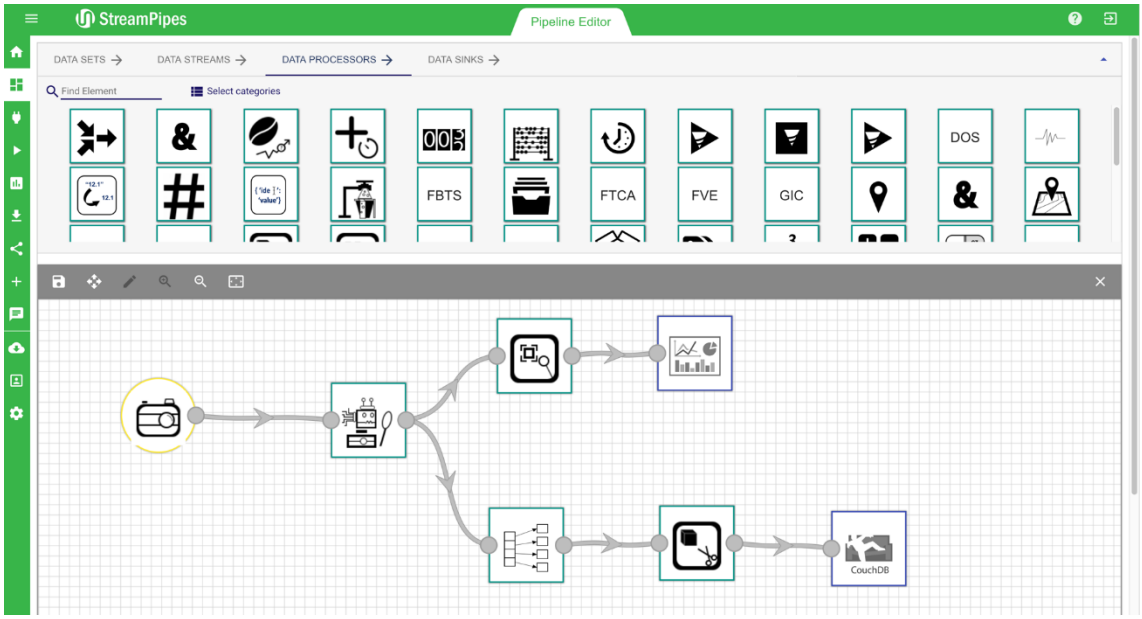


図 2.8　StreamPipes のインタフェース

品が現在提供されている。

2.1 節で述べた CUDA と OpenCL ストリームコンピューティングパラダイムを含む。その他、BrookGPU[5]、PeakStream[57]、StreamPipes[58]などのストリームコンピューティング支援ツールがある。Brook for GPUs は、GPU 向けの Brook ストリームプログラミング言語のコンパイラおよびランタイム実装である。PeakStream は、ATI のグラフィックプロセッサ向けに並列処理ライブラリである。StreamPipes は 2015 年から提案されたリアルタイムのビッグデータストリームを統合するための単一プラットフォームである。リアルタイムのビッグデータを誰もが利用できるようにするソリューションであり、グラフィカルモデラーにより、ビジネスアナリストはビッグデータの専門家やデータ科学者を必要とせずに、アドホックな方法で処理パイプラインを定義することができる。図 2.8 に示すように、ストリーム処理パイプラインを作成するための Web ベースのユーザインタフェースを通じて、ビジュアルプログラミング方式を提供し、プログラマビリティの向上を注力している。

アクセラレータのアプリケーションを開発する際に、プログラマは上述のランタイム環境のいずれかを選択する必要があり、並列ストリームコンピューティングのコンセプトを理解する必要もあり、コーディング作業が非常に困難である。また、異なるタイプのアクセラレータ間でのプリグラムの移植も困難である。さらに、ストリームコンピューティングに適用する場合、新しいプログラミング方法が必要とする。それを実現するために、Caravela プラットフォームと呼ばれる統一されたインタフェースが提案された[8]。プログラマは、Caravela プラットフォームを使用することで、アクセラレータ上のローレベルランタイムを気にせず、カーネルプログラムを開発することができる。プログラマは、カーネルプログラムのコードをパックし、カーネルプログラムの引数の I/O 関係とターゲットランタイムの

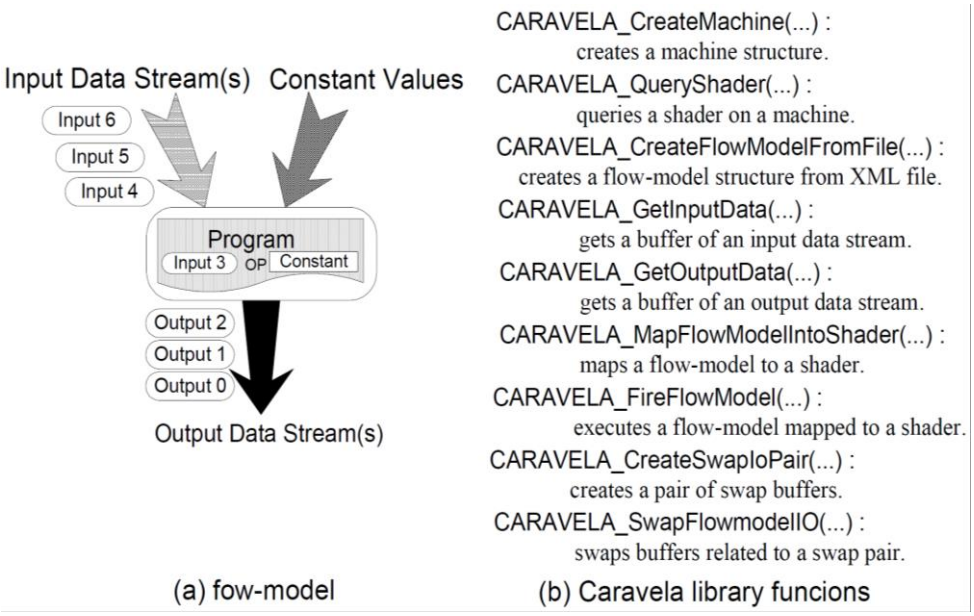


図２.9　Caravela プラットフォーム(a)アクセラレータで実行される flow-model、(b)ホスト CPU 側の Caravela ライブラリ

タイプを指定する過程だけに注力することができる。この手順に従い、ホスト CPU 側で Caravela ライブラリ関数を使用すれば、Caravela プラットフォームが自動的にシステムで利用できるランタイムを選択し、カーネルプログラムを実行する。

Caravela プラットフォームでは、与えられたタスクをプログラミングするため、flow-model という新しい概念が使用されている。図２.9(a)に示すように、flow-model は 1 次元の I/O データストリーム、入力定数、および入力データストリームを処理し、出力データストリームを生成するプログラムで構成される。プログラマは、Caravela ライブラリを使用することで、アプリケーションを設計できる。Caravela ライブラリにより、アプリケーションは、利用可能なアクセラレータに自動的にマッピングされることができる。図２.10 に記載されているように、プログラマは、flow-model 内の指定されたタスクを実行するメソッドを XML ファイルに封入することができる。I/O データストリームとカーネルプログラムのプロパティは、テキスト形式で保存されるため、XML ファイル内のタグで指定されることができる。実際に、カーネル関数は XML ファイル内のランタイムタイプというタグで指定された言語で作成されており、指定されたローレベルのランタイムを介してアクセラレータで実行される。

Caravela ライブラリはリソース階層の定義をサポートする。その定義は周辺バスアダプターのホストマシンである Machine によって階層化される。Adapter は、1 つまたは複数のアクセラレータを含む周辺バスアダプターである。最後に、Shader はアクセラレータである。図２.9 (b)では、与えられた flow-model を実行するための基本的な Caravela 関数を示している。プログラマは、これらの関数を使用すれば、flow-model フレームワークを通して複数の flow-model をシェーダにマッピングするだけで、アプリケーションをより簡単に実装することができる。

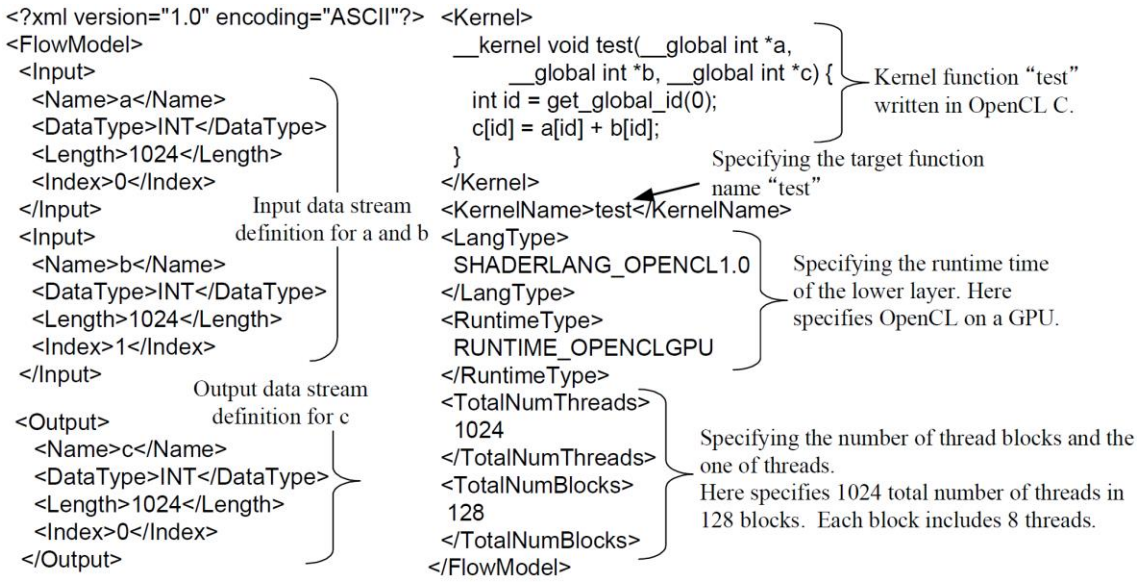


図 2.10　OpenCL GPU をランタイムとしたベクトル和を求める flow-model の例

図 2.10 では、OpenCL のランタイム上で 1024 個の演算ユニットでベクトル和を実行する flow-model の一例を示している。アクセラレータの実行スタイルは、Caravela ライブラリで統一された flow-model 実行フレームワークによって提供されている。すなわち、カーネルプログラムを作成する際に使用されるプログラミング言語が CUDA のようなローレベルランタイムと異なる場合も、その実行スタイルは変わらない。

上述のように、Caravela はターゲットアクセラレータに応じてローレベルのランタイムの差を吸収することができる。しかし、アクセラレータでのアプリケーション開発に、プログラマは依然としてホスト CPU 側とアクセラレータ側の二重のプログラミングをする必要がある。プログラマがアクセラレータ側で実行されるプログラムのみを作成したいことが考えられる。そのため、ストリームコンピューティングにおけるプログラマビリティを向上するため、革新的なプログラミングインタフェース CarSh が開発された[9]。

CarSh はストリームコンピューティングのためのコマンドラインベースの開発ツールである。CarSh を使用することで、プログラマはホスト CPU 側の制御プログラムを作成する必要がない。CarSh は executable ファイルを読み込むことにより、その executable ファイルの I/O データの割り当てに従い、入力/出力データの読み/書きを行う。executable ファイルには、Caravela フレームワークの flow-model および I/O 定義がパックされる。この I/O 定義は、カーネルプログラム内の引数を入力/出力にリンクする。図 2.11(a)に示すに、リンクされた I/O 情報が executable ファイルで定義されている。CarSh は executable ファイルから flow-model およびリンクされた I/O 情報を抽出し、flow-model を実行するために Caravela ランタイムライブラリに渡す。そして、Caravela ライブラリは、ターゲットアクセラレータに最も適合するランタイムを選択することにより、flow-model を実行する。

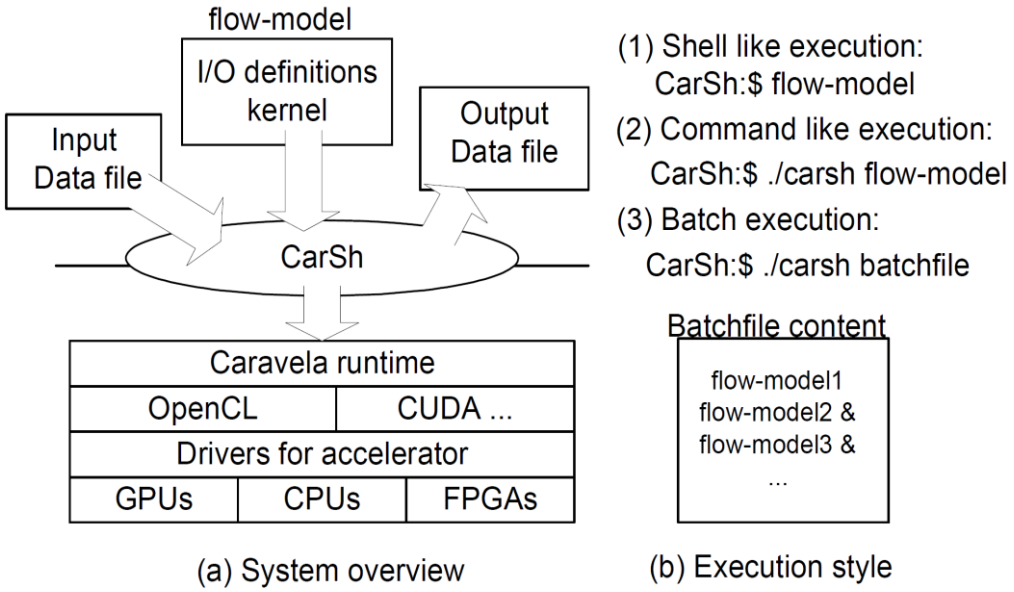


図２.11　(a) CarSh システムの概要と(b) コマンドラインでの実行スタイル

executable ファイルを読み込めるように、CarSh は、シェルのようなインタフェースを提供している。図２.11(b)に示すように、(1) シェル実行スタイル、(2) コマンド実行スタイル、および、(3) バッチ実行スタイルを提供している。また、CarSh は executable ファイルと batch ファイルを読み込むことができる。batch ファイルには、executable ファイルおよび I/O バッファに関するコマンドを記述しているシナリオを含んでいる。すなわち、batch ファイルは CarSh のコマンドをバッチ実行するための XML ファイルのフォーマットである。ここで、CarSh のコマンドラインの最後に”&”を追加することで、executable ファイル(すなわち、flow-model)と batch ファイルをバックグラウンドで実行することができる。これは、複数の flow-model の同時実行を可能にする。

executable ファイルの XML 記述は、図２.12(a)に記載されている。I/O データは、CSV ファイルによってマッピングされる。また、バーチャルバッファ(virtual buffers) を定義することができる。バーチャルバッファはバッチファイル内のコマンドによって割り当てられ、中間の出力データを保存し、次の executable ファイルにバッファ名を渡すことができる。図２.12(b)に示すように、プログラマは、batch ファイルを作成することができる。

さらに、CarSh では、flow-model を実行するための制御コマンドを提供している。プロセス管理のためのコマンド ps と kill をサポートしている。また、コマンド virtubuf は、バーチャルバッファの管理インタフェースをサポートしている。コマンド sync を使用することで、このコマンドの前に呼び出されたすべての executable ファイル、または batch ファイルを同期させる。コマンド repeat で executable ファイルまたは batch ファイルは指定された回数で繰り返し実行される。プログラマは、これらのコマンドを executable ファイル、または batch ファイル中で使用することにより、パイプライン化された処理フローの処理順序の作成に注力できる。

```
<?xml version="1.0" encoding ="ASCII"?>
<CarshEx>
 <ModelFile>Flow-model XML file name</ModelFile >
 <Input>
  <Name>Input valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Input>
 <Input>
  ....
 </Input>
 <Output>
  <Name>Output valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Output>
 <SwapPair>
  <Input>Input valuable name in kernel function</Input>
  <Output>Output valuable name in kernel function</Output>
  <NumSwap>Number of swaps</NumSwap>
 </SwapPair>
</CarshEx>
```

(a) CarSh executable format in XML

```
<?xml version="1.0" encoding ="ASCII"?>
<CarshBat>
virtbuf create int  1024 buf_a
virtbuf create int  1024 buf_b
virtbuf fill sample 1_a.csv buf_a
virtbuf fill sample 1_b.csv buf_b
virtbuf create int  1024 buf_c
sample_virt &
sync
repeat 10 sample_virt.xml
sync
virtbuf dump c_tmp.csv buf_c
virtbuf swap buf_a buf_c
virtbuf delete buf_a
virtbuf delete buf_b
virtbuf delete buf_c
exit
</CarshBat>
```

(b) CarSh batch format in XML

図 2.12　(a) CarSh の executable ファイルと(b) batch ファイルの XML 定義

一般的に、プログラマがアプリケーションを開発する際に、２個以上の flow-model が必要と考えられる。例えば、2 つの flow-model があり、flow-model1 と flow-model2 である。flow-model1 の出力データストリームを flow-model2 の入力データストリームとして扱うことで、その 2 つの flow-model を連続して実行することが可能になる。ここで、flow-model の入出力データストリームをエッジと扱い、カーネルプログラムをノードと扱うことにより、連結される複数の flow-model を有向グラフと扱うことができる。すなわち、多数の flow-model で大規模な処理フローを構築できる。しかし、大規模または複雑な処理フローをイメージし、手作業で構築することは困難である。そのために、図２.13 に示すように、Caravela プラットフォームの GUI が開発された。

プログラマは、複数の flow-model とそれらの間の接続をグラフィックな方法で全体として一つの処理フローとして構築できる。そして、CarSh に必要な executable ファイルと batch ファイルを自動的に生成できる。このようなストリームプログラミングはビジュアルプログラミングとデータフロープログラミングの融合に属する。

さらに、継続的かつ大量に流入するストリームデータを効率よく処理するために、並列プログラミングにより処理フローの中の並列性を最大限に抽出する必要がある。プログラマが下層ハードウェアの構成を知る必要がなくなり、コンパイラが勝手に処理フローを並列化できれば、プログラマビリティを大幅に向上することができると考えられる。

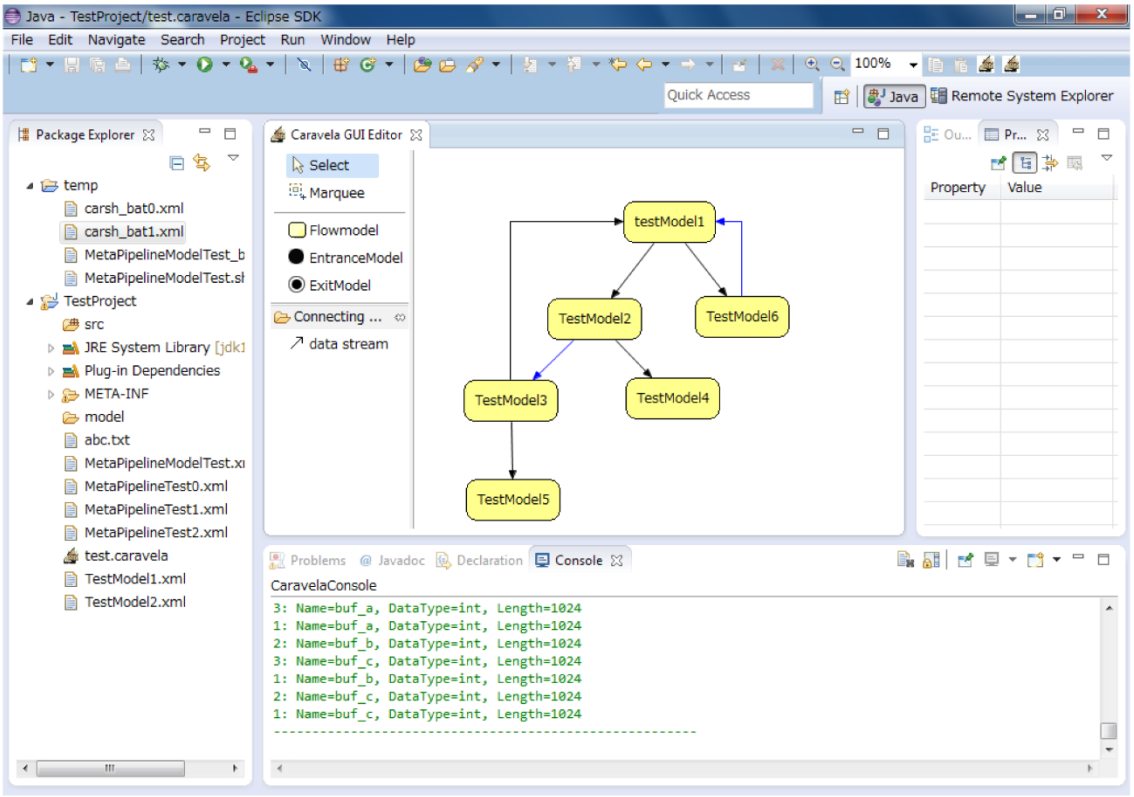


図 2.13　Caravela GUI

2.3　並列プログラミングの問題点に関する議論

並列プログラミングを行う際、プログラマは、データの依存性によりプログラムの並列性を見出す必要がある。データ依存性により、各演算の実行順序は制限される。プログラムの持つ並列性にはビットレベルや命令レベルの並列性から、データ並列性、タスク並列性まで様々なレベルがある。

- まずは、ビットレベルの並列性である。マイクロプロセッサの出現により、コンピュータアーキテクチャはワードサイズの増大をともに進展してきた。ワードサイズとは、プロセッサが一度に処理できるビット幅のことである。ワードが大きくなるほど、少数の命令で多数の命令を必要とする大きな変数を処理することが可能になる。
- 次は、プロセッサが実行すべき命令列に、命令レベルの並列性が存在する。プログラムの結果に影響を与えずに命令を並列に実行することを指す。命令レベルの並列性はパイプライン化により抽出することができる。パイプラインの各ステージは、独立し実行できる処理である。各処理を並列化して、全体として処理時間を大きく削減することが可能になる。
- また、データ並列性は、プログラムのループを対象として、異なる並列計算ノードにデータの一部を分配することで演算を行うことである。すなわち、ループは大きなデ

ータ構造の各要素に同じ処理を行うことで並列化される。一般的に、データ並列性を持つアプリケーションは多い。

- さらに、タスクレベルの並列性も並列性の一種である。タスク並列性は、同じデータセットまたは異なるデータセットに対して完全に異なる計算を実行できるという並列プログラムの特徴である。データ群をより小さなデータ要素に分けて同じ処理を並列実行するデータ並列性と異なり、タスク並列性は大規模な計算をより小さな異なる計算に分割し、並列化する。タスクレベルの並列性は粗粒度な並列性とも言われる。

さらに、GPU のようなアクセラレータを用いた並列プログラミングでは、GPU 内の複数の演算回路間からカーネル間までの並列性がある。カーネル間の並列性は複数のアクセラレータによって性能を引き出すことができる。複数のタスクが各アクセラレータで独立して並列実行するため、理想的な場合、理論上にアクセラレータの数と比例した性能向上が得られる。理想値に近づこうとし、最大の並列度で並列実行したい場合、アクセラレータの持つ並列性を十分に引き出す高度なプログラミングが必要とする。また、カーネルレベルの並列性を最大限に抽出するには、アクセラレータを搭載したハードウェアの構成に関する知識も必要となる。この点に関しては、ハードウェアを抽象化・標準化する OpenCL などのようなアクセラレータ向けのプログラミングインタフェースを利用することで、ハードウェアの詳細を隠蔽し、一定の高性能化が可能である。しかし、このようなインタフェースでは、処理単位であるカーネル間の依存関係や、現実の並列分散環境に合わせて適切な並列パターンを考慮した処理を自動生成できない。そのため、カーネル間の並列性を抽出し利用することは、プログラマにとって極めて困難な作業となる。さらに、複数のアクセラレータ間の通信が必要な場合、プログラミングがさらに複雑となり、プログラマビリティが低下していく可能性がある。そこで、本研究ではタスクレベルの並列性を着目し、並列性を抽出する際のプログラマビリティの向上を目指す。

したがって、本研究では、アクセラレータ上で高性能なアプリケーション開発の負担を軽減することに着目し、プログラマが下層ハードウェアの構成に関する知識が持たなくても、アクセラレータの高い性能を活かしてカーネル間の並列性を自動的に引き出すことを主な目的とする。その目的を達成するために、複雑な依存関係を持つ処理フローから最適な順序を決定し、並列性を抽出することに取り込む。

並列ストリームコンピューティングでは、カーネル間の依存性は空間的な並列性と時間的な並列性があると考えられる。図２.14 では空間的な並列性と時間的な並列性の簡単な例を示している。例えば、図２.14(a)の処理フローの例では、A と B の間にはエッジが存在するため、直接データ依存性が存在する。そのため、A と B を並列実行できない。しかし、B と C のように、直接に接続するエッジが存在しない、直接データ依存性がないため、B と C を並列実行できる。すなわち、実際にカーネル B とカーネル C を同時に 2 つのアクセラレ

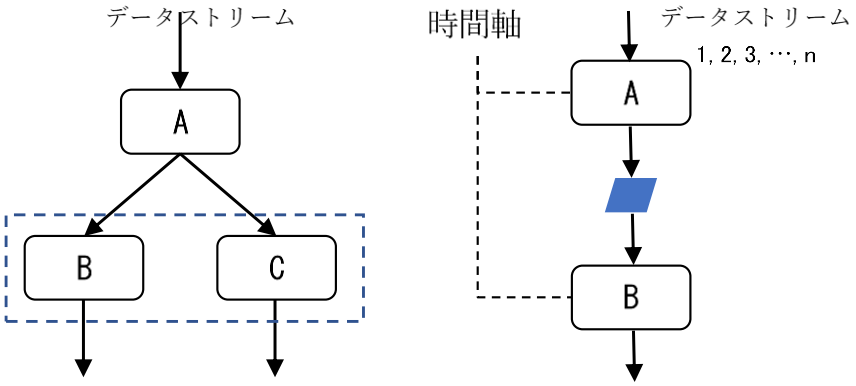


図 2 .14　空間的な並列性と時間的な並列性の例

ータでそれぞれ実行することが可能になる。これをカーネル間の空間的な並列性と呼ぶ。処理フローの中で直接データ依存性がないカーネル間では、空間的な並列性を利用し並列実行を行うことができる。

ストリームデータの特性を考慮し、カーネル間には時間的な並列性も存在すると考えられる。例えば、図 2 .14(b)の連続している 2 つのカーネル間にバッファの読み書きが行われる。データが正しく処理するために、A がバッファを書いた後に B がそのバッファを読む必要がある。前述の空間的な並列性で述べたように、一般的な並列プログラミングでは A と B を同時に実行できない。しかし、ストリームコンピューティングにおいて、時間軸があると考えられる。例えば、図 2 .14(b)の処理フローでは、A が先に実行し、データの書きが終われば、B が実行始める。この時点、A はデータを処理せずに待つことになる。ここで、A に次のデータストリームを処理させ、時間の無駄をなくすことができる。B がデータストリーム n-1 を処理すると同時に、A がデータストリーム n の処理を行うように、時間の流れと関係する並列性を時間的な並列性と呼ぶ。すなわち、並列ストリームコンピューティングでは、連続している 2 つのカーネルの実行について、時間の通過と伴い、現時点のデータストリームの処理が終われば、次のデータストリームを処理させ、パイプライン方式での並列実行ができる。

上述の例のように、処理フローが単純な場合、プログラマはカーネル間の並列性といくつかの並列実行パターンを簡単に見つけることができる。しかしながら、処理フローが大規模または複雑になると、並列、もしくはパイプライン方式で処理フローを実行するための最適な順序を決定することは困難である。

例えば、図 2 .15 に示すようなストリーム指向プログラムの処理フローを何回繰り返し実行する際、並列実行順序を見出すのは簡単ではない。例えば、最初の入力データが flowmodel1 に与えられる場合、どの flow-model を最初に実行すべきか、さらに、同時に実行できる flow-model が存在するかを適切に決定しなければならない。それから、どのように潜在的な並列

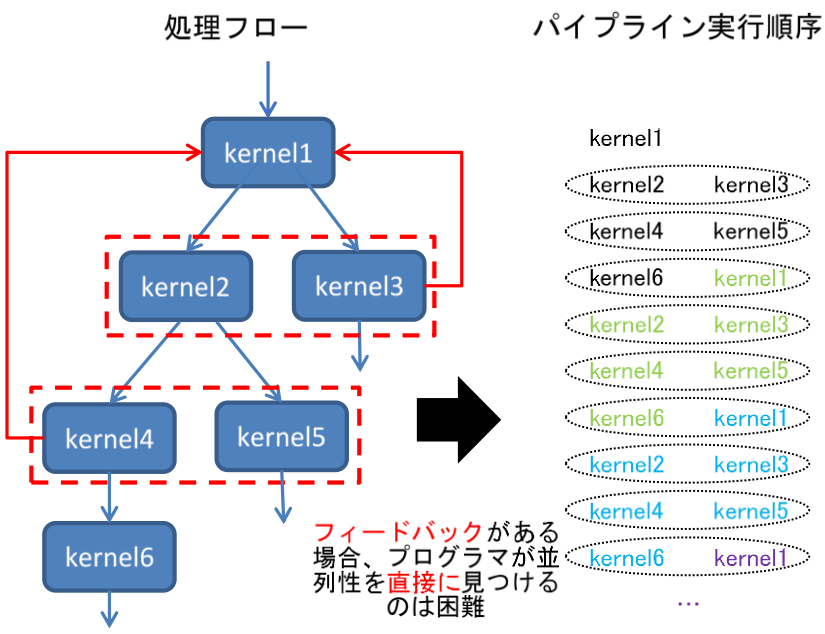


図 2.15　複雑の処理フローを並列化する例

性を持っているかを見つけ出す必要がある。図 2.15 にカーネル 4 からカーネル 1 までとカーネル 3 からカーネル 1 までのフィードバックがない場合、カーネル 2 と 3、カーネル 4 と 5 の間にエッジが存在しないため、直接データ依存性がないので、空間的な並列性を利用し並列実行できる。また、カーネル 1 とカーネル 2、カーネル 1 とカーネル 3 は直接データ依存性があるが、時間的な並列性を利用しパイプライン実行できる。同様に、連続しているカーネル 2 と 4、2 と 5、4 と 6 もパイプライン実行できる。しかし、図 2.15 のようにカーネル 4 からカーネル 1 までとカーネル 3 からカーネル 1 までのフィードバックがある場合、それの影響を考慮する必要がある。カーネル 1 とカーネル 4 が並列実行できるように見えるが、実際にはフィードバックで接続しているため並列実行できない。カーネル 3 とカーネル 6 のような一見で発見できない並列性も存在している。すなわち、このような複雑な場合、特にフィードバックを有する場合、プログラマが処理フローの並列性を直接に見つけることが困難だと考えられる。

そこで、ビジュアルプログラミング方式とデータフロープログラミング方式の融合である Caravela プラットフォームを用いて、コンパイラが勝手にカーネル間の並列性を見つけて並列実行を行うことにより、マクロなレベルでのプログラミングができれば、プログラマが下層ハードウェアの構成やコンセプトを理解する必要がなくなり、並列ストリームコンピューティングのプログラマビリティを向上させることができる。そのため、処理パイプラインから自動的に並列実行順序を決定し、並列性を見つけるためのアルゴリズムの開発が極めて重要である。それを実現するために、本研究では、複数のカーネルで構成された静的な処理フローから、カーネル間の空間的および時間的な並列性を自動的に抽出できる新た

なアルゴリズム PEA-ST を提案する。

２.４　まとめ

本章では、並列ストリームコンピューティングを本研究の背景として概説した。

２.１節では、並列計算のユーザビリティについて概説した。まず、一般的なユーザビリティの定義を述べ、計算コンピューティングにおけるユーザビリティの向上について概説した。逐次型プログラミングから、命令型プログラミング、構造化プログラミング、関数型プログラミング、オブジェクト指向プログラミング、ビジュアルプログラミング、データフロープログラミングまで 7 つのプログラミングパラダイムを簡単に述べた。この発展により、多くの研究者たちがプログラマのユーザビリティの向上を追求してきたことが分かる。次に、マルチコア/メニーコアアーキテクチャやアクセラレータについて詳しく述べた。CUDA と OpenCL を主な例として、並列プログラミングについて述べた。最後に、実環境のハードウェア構成に適した並列プログラミングがプログラマにとって低効率で困難な作業であることを述べた。

２.２節では、ストリームコンピューティングのプログラマビリティについて概説した。まず、ストリームコンピューティングの現状や発展について述べた。次に、ストリームコンピューティングのプログラミングインタフェースを提供する Caravela プラットフォームの詳細と flow-model の定義について述べた。また、コマンドラインベースのプログラミングツール CarSh および開発用 GUI も概説した。Caravela プラットフォームを用いたことで、ローレベルのランタイムを気にせず、カーネルの開発に集中できることが確認された。

0 節では、、並列プログラミングにおける課題について議論し、本研究の目的について述べた。まず、ビットレベルから、命令レベル、データレベル、タスクレベルまで各レベルのプログラムの並列性について述べた。次に、カーネル間の並列性を自動的に抽出し、プログラマビリティを向上させることの重要性について述べた。そして、本研究で検討されたカーネル間の並列性が空間的な並列性と時間的な並列性に分けられることを例で述べた。最後に、上述の潜在的な並列性を自動的に抽出できるアルゴリズムの開発を本研究の目的として述べた。

次章では、並列性抽出アルゴリズムの開発について詳しく述べる。

第３章
並列性抽出アルゴリズムの開発

本研究では、ストリームコンピューティングにおける並列性を、時間的な並列性と空間的な並列性の２種類に分類し取り扱う。２.３節で述べたように、時間的な並列性は、複数カーネルのパイプライン実行に適用でき、スループットの向上を図る。空間的な並列性は、マルチ GPU 環境の並列処理に適用でき、レイテンシの削減を図る。コンピューティングリソースの規模を考慮すると、処理フローのカーネル間の空間的および時間的な並列性の自動抽出技術は、ストリーム指向プログラムの性能を改善し、プログラマビリティを向上するための重要な課題である。

本章では、並列性抽出に関する課題について論じ、本研究で提案する PEA-ST アルゴリズムの詳細を述べる。

３.１　並列性抽出に関する問題点と解決法

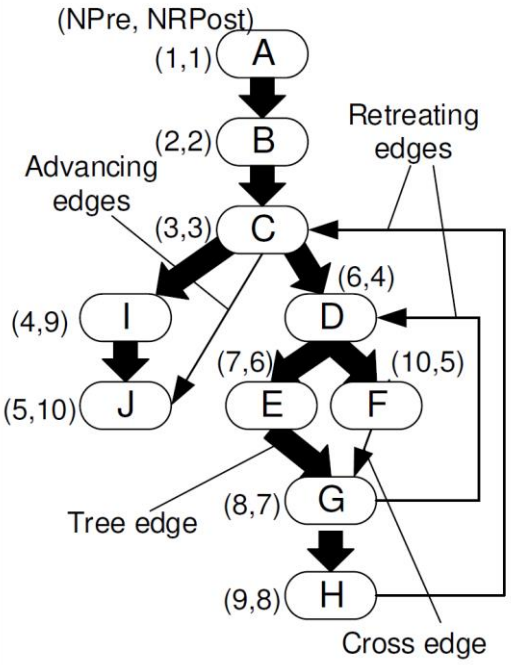
並列性を抽出する際には処理の依存関係や、適切な並列パターンを考慮する必要がある。処理フローが単純な場合、プログラマは flow-model 間の並列性といくつかの並列実行パターンを簡単に見つけることができる。しかしながら、処理フローが大規模または複雑になると、並列、もしくはパイプライン方式で処理フローを実行するための最適な順序を決定することは困難である。本研究では、本アルゴリズムの目標の一つとして、処理フローの並列性を抽出するために、カーネル間の依存性をすべて自動的に見つけることが必要となる。一般的に、処理フローはツリーとして扱うことができる。データストリームをエッジとして扱い、カーネルをノードとして扱うことができる。すなわち、ツリーで表現された処理フローの各ノード間の関係やエッジの情報をアルゴリズムで見つけ出す必要がある。さらに、ループへの対応も一つの問題点である。エッジを分類し、フィードバックやフィードフォワードを見つけることができればループへの対策も立てられる。

一方、Spanning Tree という概念が存在する。STP(Spanning Tree Protocol)[59]などの通信ネットワーク分野では Spanning Tree 方式が主に適用されている。Spanning Tree プロトコル（STP）は、任意のブリッジイーサネットベースのローカルエリアネットワークのためのループのないトポロジを保証するネットワークプロトコルである。リンクの数が最も少ない、ループを排除した最短ネットワーク経路を検索する際に用いられる。また、[60]で導入された Spanning Tree はプログラムの制御フローを定義するために使用される。

Algorithm 1 Depth-first spanning tree (DFST) algorithm.

```
void DFST(int x){
  pre_num ++;
  NPre[x] = pre_num;
  for(int y=0; y<N; y++){
    if(edges[x][y].connect){
      if(NPre[y] == 0){
        //x -> y is a tree edge
        DFST(y);
      }
      else if(Npre[x] < Npre[y])
      { // x -> y is an advancing edge }
      else if(NRPost[y] == 0)
      {  //x -> y is an retreating edge }
      else { // x -> y is an cross edge }
    }
  }
  NRPost[x] = rpost_num;
  rpost_num --;
}
```

(a)



(b)

図 3.1　Spanning Tree の例と DFST アルゴリズム

(1) Spanning Tree の定義

有向グラフ G(V, E) (V はグラフ G の頂点の集合であり、E はグラフ G のエッジの集合である、以下も同様に) において、エッジの部分集合 T が T ⊆ E を満たす、かつ、グラフ S(V, T) がツリー構造である場合、S(V, T) のことを有向グラフ G(V, E) の Spanning Tree と呼ぶ。すなわち、S(V, T) はループを持っていないグラフである。

Spanning Tree が定義されると、元の有向グラフ G のエッジを以下の 4 種類に分類できる。ここで、X → Y は X から Y へのエッジのことである。

- **Tree edges:**
  Spanning Tree になるエッジの集合。
- **Advancing edges:**
  X → Y が Spanning Tree のエッジにならない、かつ、Y が X の子孫ノードであるエッジの集合。すなわち、エッジは、ツリー構造の下部構造にある頂点にジャンプする。
- **Retreating edges:**
  X → Y が Spanning Tree のエッジにならない、かつ、Y が X の先祖ノードであるエッジの集合。すなわち、エッジは、ツリー構造の上部構造にある頂点にジャンプする。

- **Cross edges:**
  X → Y が上述の 3 種類の中のどちらにも属さないエッジの集合。すなわち、Y が X の子孫でなく、かつ、Y が X の先祖でもない、残されたエッジの集合。

ただし、Spanning Tree のルートに関する条件が存在する。ノード自身から残りの全てのノードまで到達できるノードは Spanning Tree のルートになれる。あるノードが G の全ての頂点に到達できない場合、そのノードからの Spanning Tree は存在しない。ルートが固定されていた場合、このルートから生成される Spanning Tree は唯一である。すなわち、同じグラフの異なるルートから、異なる Spanning Tree を生成する可能性がある。

図 3 .1(b)に生成された Spanning Tree の例を示している。C → J は C → I → J → C のループがあるため、Spanning Tree のエッジにならない。また、J は C の子孫であるため、C → J は Advancing edge に分類される。H → C および G → D は同じように複数のループを生成してしまうため、Spanning Tree のエッジになることはできない。相対的に、H は C、G は D の先祖であるため、この 2 つのエッジは Retreating edge になる。E → G と F → G を見ると、ツリー構造の定義により、G は同時に E と F の子孫になることができないことが明白である。そこで、F → G を Cross edge に分類される。これらのエッジの分類が終了すると、Spanning Tree が得られる。

(2) DFST アルゴリズム

図 3 .1(a)のアルゴリズム 1 は、Tarjan の深さ優先探索アルゴリズム[60]に基づいて開発された Depth-First Spanning Tree(DFST）と呼ぶアルゴリズムである。DFST 関数にルートが与えられると、上記の Spanning Tree に関する 4 種類のエッジの定義により、指定された有向グラフのエッジを分類し、Spanning Tree を生成する。

DFST アルゴリズムでは、各ノードの先行順番号 Npre()と逆後行順番号 NRPost()が付けられる。先行順番号を付ける手順には、Npre(X)＜Npre(Y)であれば、X は Y の先祖、あるいは、Y の左側である。逆後行順番号を付ける手順には、NRPost(X)＜NRPost(Y) であれば、X は Y の先祖、あるいは、Y の右側である。まず、現在のノードに先行順番号を付け、次の接続されたノードの先行順番号をチェックする。それがゼロの場合、そのエッジは tree edge として検出される。探索が葉ノードに到達すると、エッジによって戻り、逆後行順番号を付けていく。この間に、retreating edges、advancing edges と cross edges を検出する。

図 3 .1(b)では、アルゴリズム DFST によって生成された Spanning Tree の一例を示している。(NPre, NRPost) は各ノードの先行順と逆後行順番号である。A をルートとして選択する。まず、A の先行順番号が 1 になる。この時点、B の先行順番号がまだ付けていないので、A → B のエッジは tree edge に分類される。同じく、B → C、C →

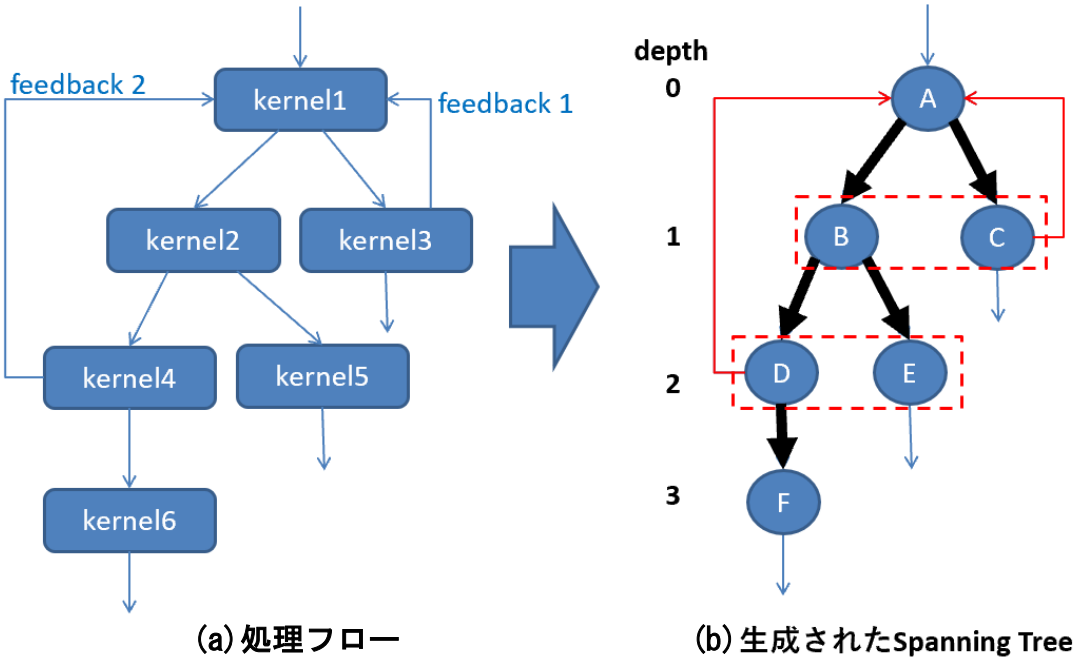


図 3.2　ストリーム処理フローへの Spanning Tree のマッピング

I、I→J がすべて tree edge に分類される。そして、葉ノードから戻り、逆後行順番号を付ける。合計 10 個のノードがあるので、逆後行順番号は 10 から始まり 1 つずつ減少していく。すなわち、最初の逆後行順番号として、J の逆後行順番号は 10 で付ける。そして、I に戻り、I の逆後行順番号は減少し 9 になる。C に戻り、C→J のエッジを分類する。C の先行順番号の 3 は J の先行順番号の 5 より小さいため、C → J のエッジは advancing edge に分類される。次に、D、E、G、H まで先行順番号を付け、tree edge で標記する。H→C のエッジに到達し、C の逆後行順番号はまだ付けていないため、H → C のエッジは Retreating edge に分類される。ここで、探索が一旦終わり、H の逆後行順番号を 8 で付ける。G → D のエッジも Retreating edge として分類される。そして、G、E の逆後行順番号を付けて D に戻り、D→F のエッジが tree edge に分類される。次に、F→G のエッジが cross edge に分類される。最後に、F から D、C、B、A まで戻り、全てのノードの番号付けが終わった。

すなわち、全体的には、A→B→C→I→J→D→E→G→H→F の順番で先行順番号は付けられ、J→I→H→G→E→F→D→C→B→A の順番で逆後行順番号も付けられる。このように、ノードの先行順番号と逆後行順番号を付ける過程にすべてのエッジを分類できる。

以上のように、Spanning Tree の生成により、ノード間の関係とエッジの分類情報を効果的に見つけることができる。

この節の最初に述べたように、並列性を抽出するには、カーネル間の依存性をすべて見つける必要がある。処理フローをグラフと見なし、各ノード間の関係やエッジの情報を自動的に抽出する必要がある。さらに、フィードバックやフィードフォワードの対応も課題である。ここで、Spanning Tree を処理フローに適用してみる。例えば、図３.2(a)に示すような入出力データストリームで接続された処理フローをツリーとして扱い、Spanning Tree を生成すると、図３.2(b)のようになる。フィードバックの I/O が retreating edges として検出される。すなわち、各ノード間の依存性を全部発見できることを確認した。

さらに、各ノードに depth を付けられる。横方向を見ると、B と C、D と E のように、ルート A から同じ深さを持つノードの間にエッジを持っていない。すなわち、直接データ依存性がないため、空間的な並列性を利用し、B と C を 1 つのステージに、D と E を 1 つのステージとして並列実行できる。また、縦方向を見ると、直接データ依存性があるため、データストリームで接続されたノードは時間的な並列性を利用し、パイプライン実行できる。

したがって、Spanning Tree により、横方向に空間的な並列性があり、縦方向に時間的な並列性がある特性から、処理フローに含まれるすべてのカーネル間の並列性を非常に効果的に見出すことができる。本研究では、Spanning Tree を利用し、複数のカーネルで構成された静的な処理フローから、自動的に時間的な並列性と空間的な並列性を抽出し、パイプライン実行順序を決定する新たな並列性抽出アルゴリズム Parallelism extraction algorithm with spanning tree(PEA-ST)を提案する。

３.２　PEA-ST アルゴリズムの設計

実行順序と並列性を見つけるために、以下の三つの手順が必要となる。1）最初に実行できるノードを見つける。次に、2）I/O バッファが競合しない実行順序を発見する。3）処理フローから並列性を抽出する。本研究では、flow-model をノードとして扱い、入力/出力データストリームをエッジとして扱うことで、処理フローを有向グラフとして取り扱う。

PEA-ST アルゴリズムの全体の流れを図３.3 の例で示す。まず、ルートノードになることができるノードを探す。次に、ルートノードから Spanning Tree を生成する。ここで、エッジの分類および consistency shift の計算を行う。そして、consistency shift を適用し、ノード間の並列性を見つけ、Execute matrix を作成する。最後に、並列実行順序を決定し、Execute matrix から startup と repeat batch を生成する。

以下に PEA-ST アルゴリズムの詳細を述べる。

ステップ１：　最初に実行できるノードの確定

まず、executable ノードと root ノードの定義を前提として説明する。あるノードへのすべてのエッジ（すなわち、そのノードの全ての入力ストリーム）が与えられた場合、そのノードを executable ノードと呼ぶ。また、そのノードから出てくるエッジは、ノードの実

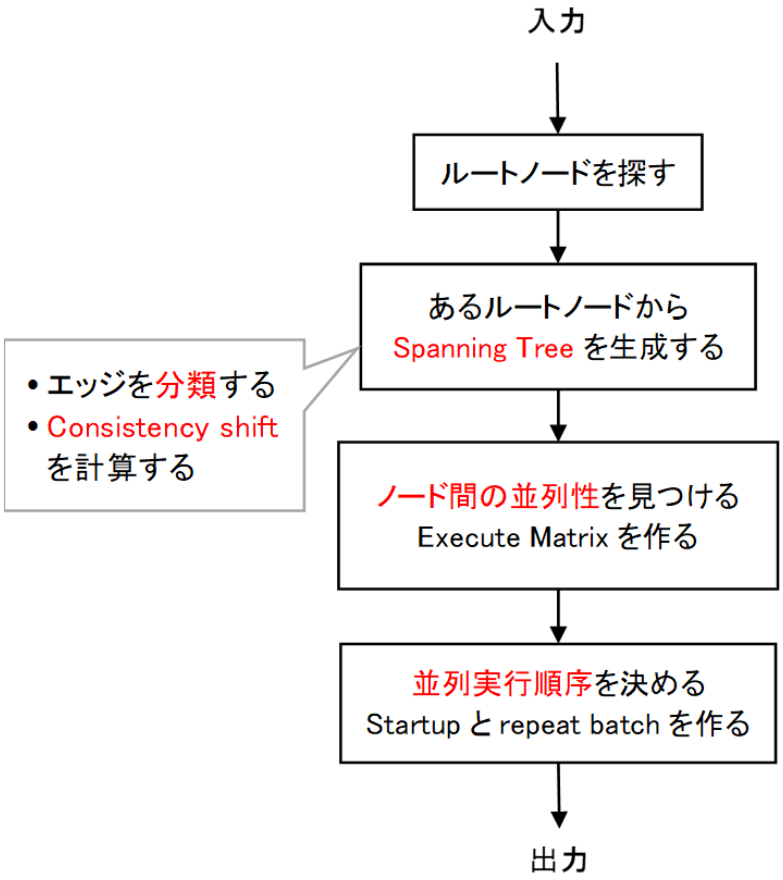


図 3 .3　PEA-ST アルゴリズムの流れ

行が終了した後、初期化された状態になる。ここで、初期化状態はノードに入るすべての入力データストリームの準備ができた状態を意味する。グラフ内のすべてのノードを実行する前に、あるノードのすべての入力データストリームが既に初期化された場合、そのノードを root ノードと呼ぶ。すなわち、root ノードを特殊な executable ノードとして定義する。

図 3 .2 に示すように、あるグラフが kernel4 → kernel1 のようなフィードバックエッジを幾つか有する場合、そのグラフのパイプライン実行には、必ずデッドロックが発生する。この場合、kernel1 → kernel2 → kernel4 → kernel1 のようなサイクリックパス（cyclic path）の内の 1 つのエッジは初期化されなければならない。有向グラフでサイクリックパスの最小セットを見つけるためのアルゴリズムは既に提案されている[10]。サイクリックパスの最小セットを作成する手順を以下のように要約できる：

1. すべてのノードからすべてのサイクリックパスを列挙する
   例えば、図 3 .2 のすべてのノードからすべてのサイクリックパスは示されている（図 3 .4[Step 1]）。

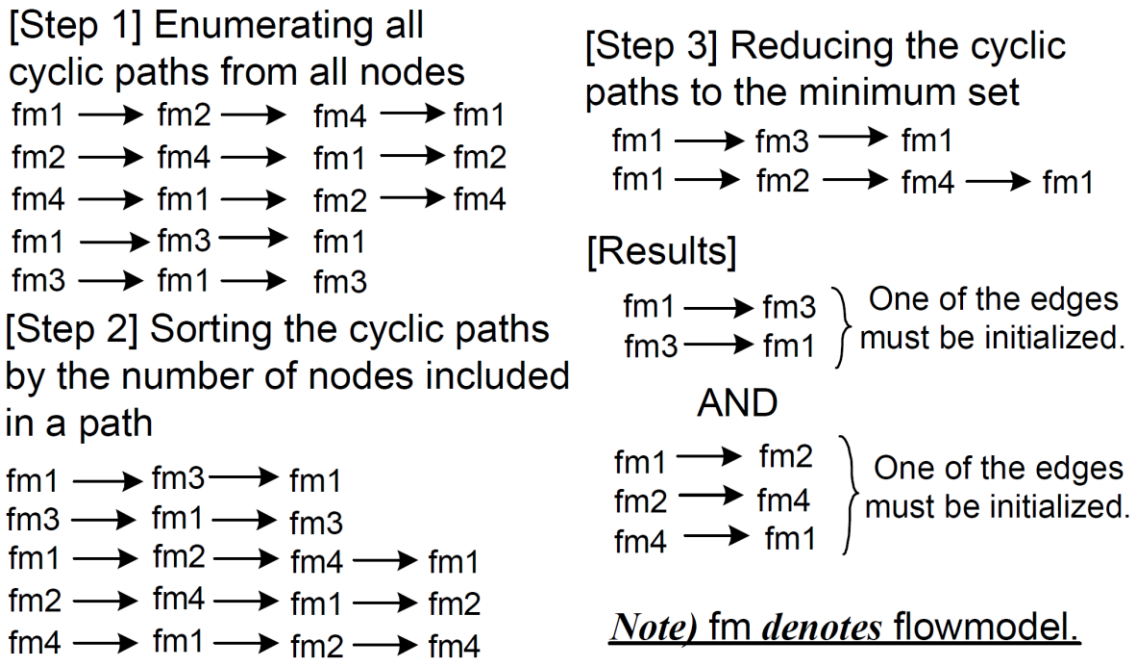


図 3.4　デッドロックの回避の例

2. パスに含まれるノード数によってサイクリックパスを配列する
   各サイクリックパスに含まれるノードの数を比較し、すべてのパスセットをソートする（図 3.4[Step 2]）。
3. サイクリックパスを最小セットへ減らす
   最小ノード数を持つパスから最大ノード数を持つパスへの比較を行う間に、比較されたパスが現在のものと同じである場合、前者は削除される。最後に残されるパスセットが最小サイクリックパスになる。このセット内のすべてのパスに対し、実行時のデッドロックを回避するために、各パスの 1 つのエッジを初期化する必要がある。例えば、fm1→fm3→fm1 のサイクリックパスを実行する場合、デッドロックを回帰するために、fm1→fm3 のエッジあるいは fm3→fm1 をダミーデータで初期化する（図 3.4 [Results]）。

このアルゴリズムにより、すべてのサイクリックパスを発見し、必要なエッジを初期化した後に、root ノードを発見できる。あるグラフがいくつかの root ノードを持つ場合、そのグラフを実行可能な有向グラフ（executable directed graph）と呼ぶ。

ある実行可能な有向グラフに必ず一個以上の executable ノードが存在する。これらのノードは、グラフの root ノードに対応する。ここで flow-model で構成された処理パイプラインを用いて説明する。root ノードは、パイプライン実行の開始に実行される flow-model と同等である。この初期化方法によると、処理パイプラインの最初に実行できる flow-model を見つけることができる。

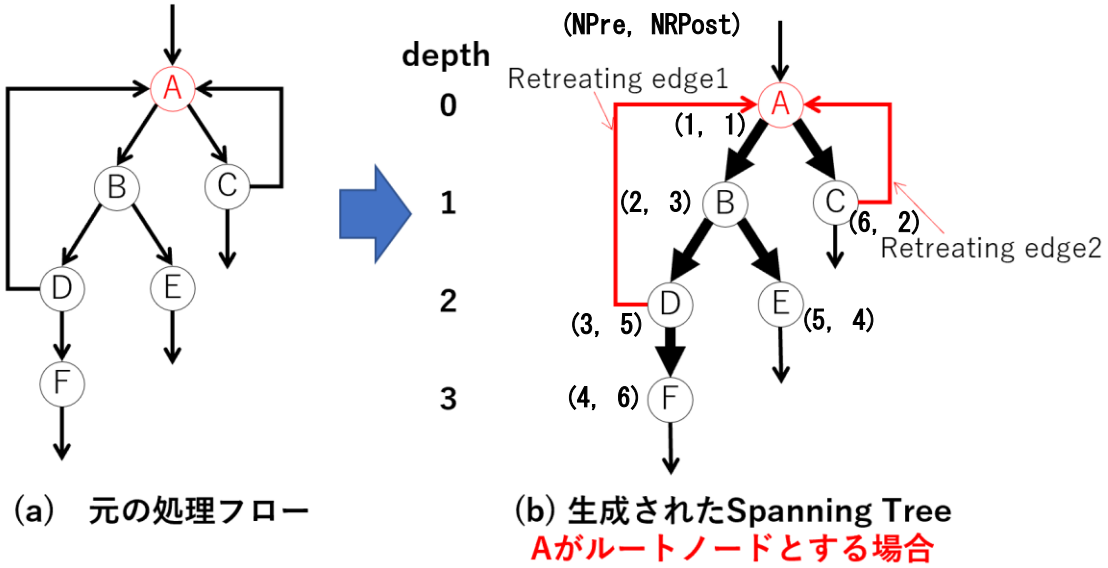


図 3 .5　Spanning Tree の生成の例

したがって、前述したアルゴリズムを適用することで、有向グラフのルートノードを定義できる。最小サイクリックパス内のノードのいずれかをルートノードとして選択すると、Spanning Tree を生成できる。実行ステップがツリーエッジに沿ってルートノードから下向きへ進める場合、唯一なツリーパスが得られる。このパスは、flow-models で構成された有向グラフの実行フローになる。

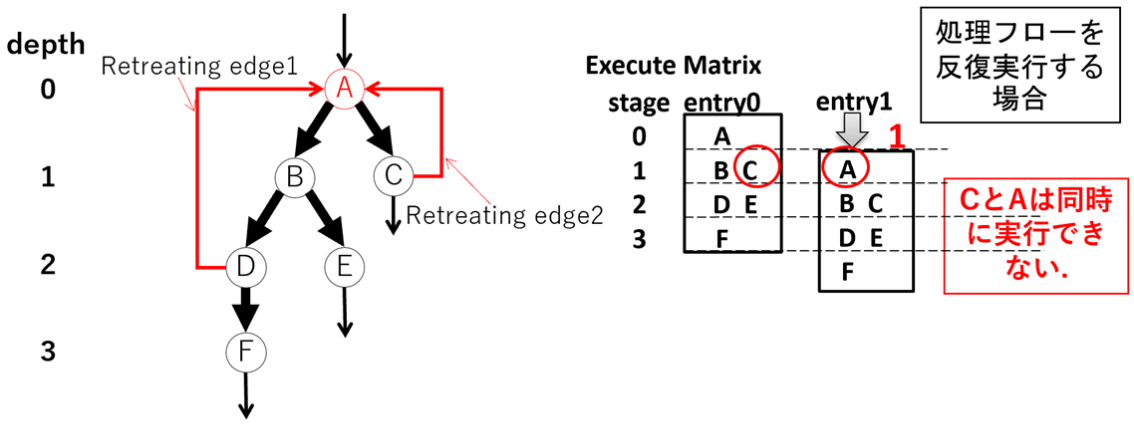
ステップ２： Spanning tree の生成

このステップでは、上述のようにルートノードになることができるノードを探し、そのルートノードから Spanning Tree を生成する。

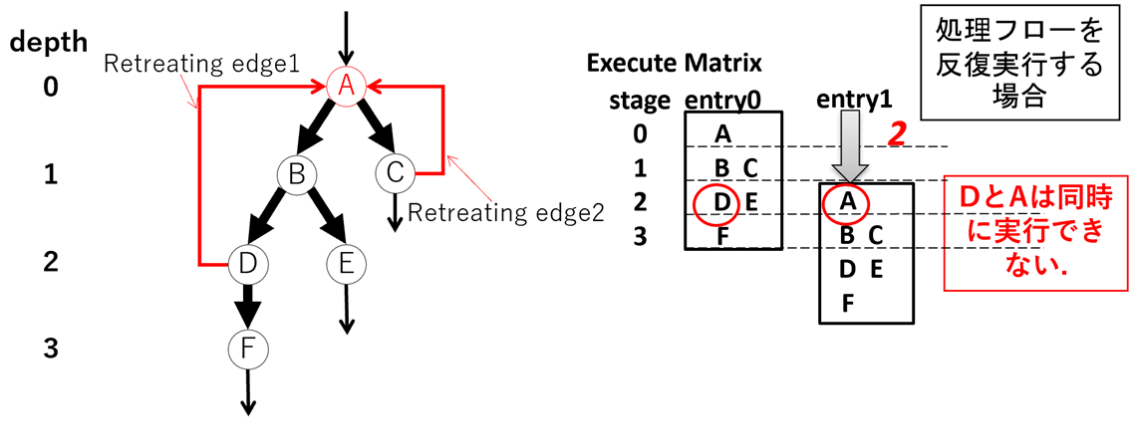
例えば、図 3 .5(a)のような処理フローが与えられ、D → A と C → A は初期化されたことを仮定する場合、A をルートノードとして選択すると、生成された Spanning Tree が図 3 .5(b)に示されている。初期化されたエッジは retreating edges に分類される。その他は、tree edges になる。

Spanning Tree を生成するアルゴリズム DFST により、(NPre, NRPost) は各ノードの先行順番号と逆後行順番号である。図 3 .5 に示されように、A がルートとして選択されたため、まず、A → B → D → F → E → C の順番で各ノードの先行順番号は付けられた。次に、バックワード探索の番号付けの際に、F → D → E → B → C → A のように逆後行順番号も付けられた。前のステップで、太線の矢印で表示した 5 つの tree edge を発見した。そして、バックワード探索の間に、retreating edge1 と retreating edge2 は発見された。

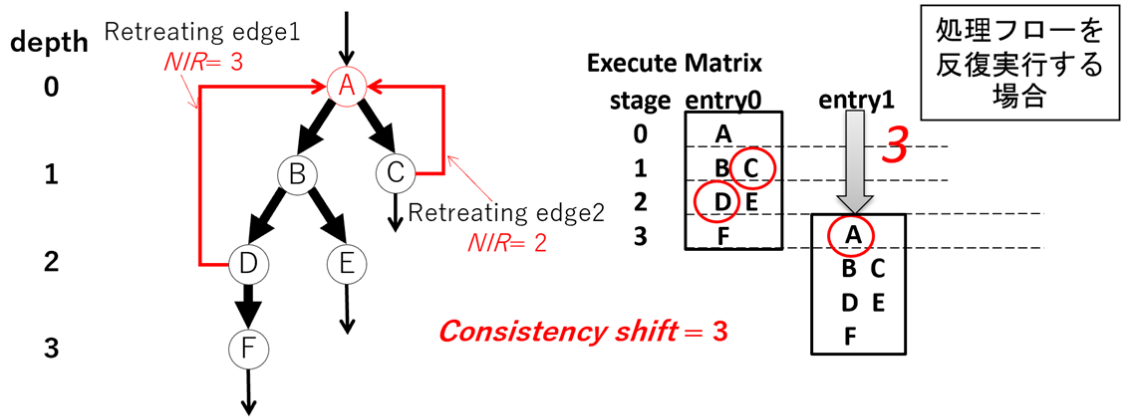
Spanning Tree から並列性を抽出するために、処理パイプラインの depth が定義される。ノードの depth は、そのノードからルートノードまでのパスにある tree edges の数として



(a) consistency shift を使用しない場合（1 で下にシフト）



(b) consistency shift を使用しない場合（2 で下にシフト）



(c) consistency shift を使用する場合（3 で下にシフト）

図 3.6　ノード間の並列性の抽出の例

定義される。例えば、図３.5 の左側にある Spanning Tree の depth を生成する。A → B → E のパスに 2 つの tree edge があるため、ノード E の depth は 2 になる。また、A→B→D → F のパスに 3 つの tree edge があるため、ノード F の depth は 3 になる。また、この例の depth の深さは 4 になる。

ステップ３： ノード間の並列性の抽出

このステップでは、Spanning Tree の並列度を解析する。ここで、ノード間の並列性を議論するために、stage と entry を定義する。stage は I/O バッファの競合なしに同時に実行できるノードの集合である。entry はすべてのノードが含まれているノードの最小集合である。

depth と stage の定義により、同じ depth を持つノードが同じ stage に加えることができる。3.1 節で議論したように、同じエッジを共有するノードは I/O 競合のために同じ stage に存在できない。すなわち、同じ深さを持つノードの間にエッジが存在しないため、それらを同じ stage に加えることができる。ここでは、1 つの tree edge がパイプラインの 1 つの stage を有することを定義する。例えば、ノード B と C、D と E の間に I/O 競合が存在しないため、同じ stage に入り、並列実行できる。Execute Matrix の中に、「下」の方向が現在の stage より小さい stage 番号を持つ方向を意味し、「上」の方向が現在の stage より大きい stage 番号を持つ方向を意味する。

前に生成された Spanning Tree を用いて、並列性を抽出する。まず、同じ depth のノードを各 stage に埋め、entry0 を A → (B, C) → (D, E) → F のようにパイプラインを構築できる。次に、パイプライン実行順序を決定するために entry1 を生成する。普通の直線型パイプラインでは、一つ後にシフトすれば、パイプライン実行は成り立てる。しかし、この例で 1 行下にシフトすると、図３.6(a)のように、パイプラインは A → (B, C, A) → (D, E, B, C) → (F, D, E)…。A と C は同じステージになる。Retreating edge2 が存在するため、A が C の後に起動しなければならないため、C と A は同時に実行できない。すなわち、I/O 競合が発生してしまう。さらに 1 行下にシフトする、すなわち、2 行下にシフトすると、パイプラインは A→(B, C)→(D, E, A)→(F, B, C)…。今回、A と D は同じステージになる。Retreating edge1 が存在するため、A が D の後に起動しなければならないため、D と A は同時に実行できない。すなわち、2 行下にシフトしても、I/O 競合が発生してしまう。

I/O 競合を回避するために、処理グロー中の retreating edges の影響を考慮する必要がある。Retreating edge に対して NIR と Consistency shift を定義する。NIR (node in retreating edge) を、retreating edge を含むループに、tree edge で接続されるノード数として定義する。例えば、retreating edge1 D → A がループ D → A → B → D にあるため、tree edge で接続されている A→B→D は 3 つのノードを含む。その結果、retreating edge1 の NIR は 3 にる。同様に、C → A → C のループは 2 つのノードを含んでいるため、retreating edge2 の NIR は 2 になる。さらに、複数の Retreating edge のために、Consistency shift を定義する。

Consistency shift は処理フローにある最大の NIR の値である。図 3 .6 の例では、Consistency shift は 3 になる。

今回、consistency shift で、entry1 を下にシフトする。すなわち、entry1 を 3 行下にシフトする。これで、retreating edge1 と 2 があるため、A が D、C と同時に実行できない問題が解決された。すなわち、I/O 競合なしに並列性を抽出することができた。Entry のすべての stage を consistency shift で下にシフトすると、retreating edges に関連するすべてのノードは必ず正しく実行される（定理 1 を参照）。その定理と証明は以下のように記述される。

定理 1 (consistency shift)**.** spanning Tree の理論により、ある強連結有向グラフのエッジをすでに分類された場合、いくつかの retreating edge があり、かつ、その中で consistency shift を持つものが存在すると仮定する。execute matrix で entry を consistency shift 行で下にシフトする場合、consistency shift を持つ retreating edge の新しい終点ノードは必ず旧い始点ノードの丁度一つ上の stage にあり、かつ、その他の retreating edge の新しい終点ノードは必ず旧い始点ノードのより上の stage にあることになる。

証明. 定理中の consistency shift を持つ retreating edge を M→N と、その consistency shift を K と仮定する。また、M は stage p にあり、N は stage q にあることを仮定する。

ノードの表記は「node(stage)」にする。移動後のノードの名前に「′」をつける。

ゆえに、定理 1 の「consistency shift を持つ retreating edge の新しい終点ノードは必ず旧い始点ノードの丁度一つ上の stage にあり」は

補助定理 1. M(p) → N(q) を K 行の下にシフトし、M′(p + K) → N′(q + K) になる場合、M(p) は N′(q + K) の丁度一つ上の stage にあることになる。すなわち、

$$(q + K) - p = 1 \tag{3.1}$$

を証明することになる。

M(p)→N(q) は retreating edge であり、かつ、その NIR が K であり、ゆえに、retreating edge と consistency shift の定義により、

$$p - q = K - 1 \tag{3.2}$$

が得ることができる。

よって、式(3.1) は 式(3.2)から導出できることにより、同じであることは明確である。

ゆえに、命題 1 は証明された。

元のグラフに任意の retreating edge：L → O が存在し、その NIR を J と仮定する。また、L は stage r にあり、O は stage s にあることを仮定する。

表記は前と同じであれば、定理 1 の「その他の retreating edge の新しい終点ノードは必ず旧い始点ノードのより上の stage にある」は

補助定理 2. L(r) → O(s) を K 行の下にシフトし、L′(r + K) → O′(s + K) になる場合、L(r) は O′(s + K) の上の stage にあることになる。すなわち、

$$r < s + K \tag{3.3}$$

を証明することになる。

同様、L(r) → O(s) は retreating edge であり、かつ、その NIR が J であり、ゆえに、retreating edge と consistency shift の定義により、

$$r - s = J - 1 \tag{3.4}$$

が得ることができる。

また、L→O は任意の retreating edge であるため、その NIR：J は必ず consistency shift より小さいか等しい、すなわち、

$$J \leq K \tag{3.5}$$

式(3.4) と式(3.5) によると、

$$r - s + 1 \leq K$$

$$r - s \leq K - 1$$

$$r - s < K$$

ゆえに、式(3.3) を得ることができる。命題 2 は証明された。

すなわち、定理 1 が成立する。

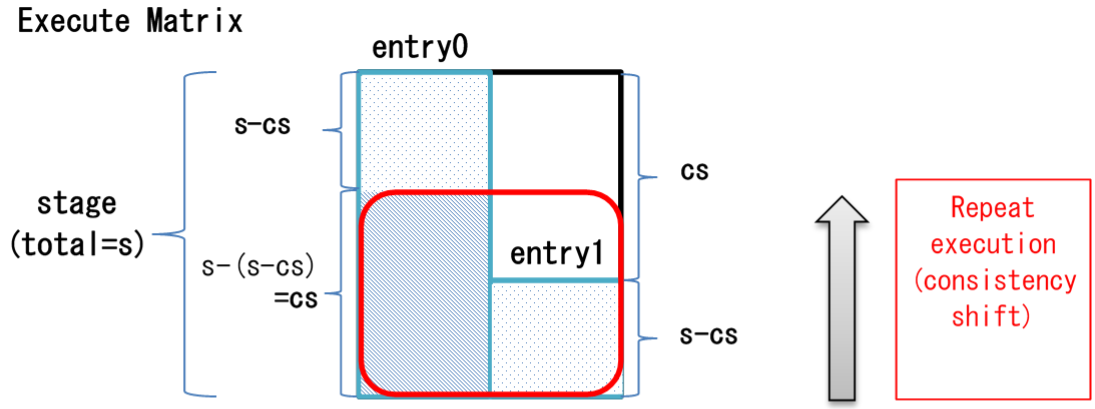
ステップ４： 並列実行順序の決定

並列度の解析の後、このステップでは前に生成した実行順序を用いて、並列処理 batch を生成する。並列化されたパイプラインは、2 つの段階に分けられる。Startup と呼ぶ一回だけ実行する初期段階と、repeat batch と呼ぶ連続して繰り返し実行する段階である。

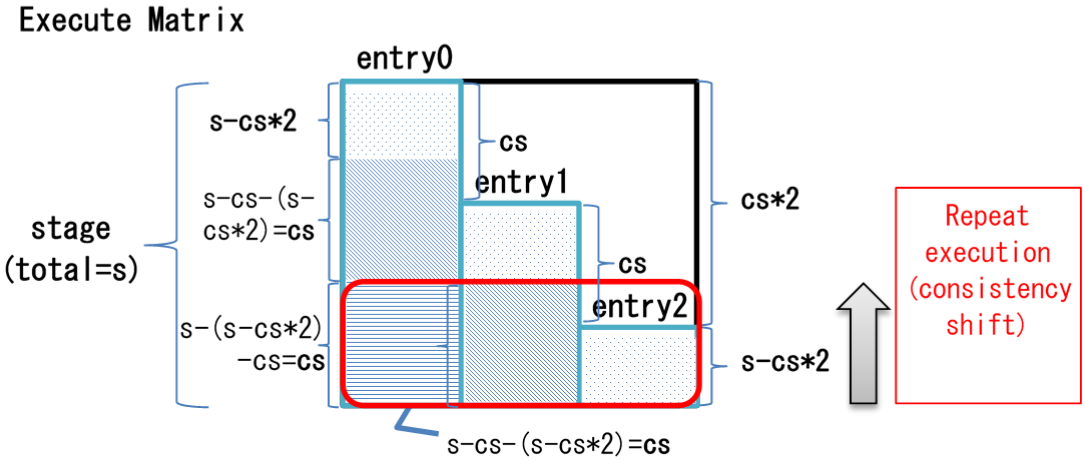
Consistency shift の計算が終了すると、startup と repeat batch を分割する。repeat batch 部分は繰り返し実行されるので、全てのノードを含む必要がある。そこで、repeat batch に含まれる stage 数は consistency shift により確定される。残りの startup に含まれる stage 数は depth － consistency shift で計算できる。また、下記の定理 2 の証明により、ステップ 3 の結果の並列度が変化しても、consistency shift で確定された repeat batch が全てのノードを含む。その定理 2 は以下のように証明される。

定理 2 (Creating startup and repeat batch). execute matrix がすでに作られた場合、その最終 stage からよりより小さな番号を持つ stage への max consistency shift 行の stages は repeat batch 部分になり、残りの上部の stages は startup 部分になる。すなわち、分割された repeat batch 部分は必ず entry0 の全てのノードを含む。

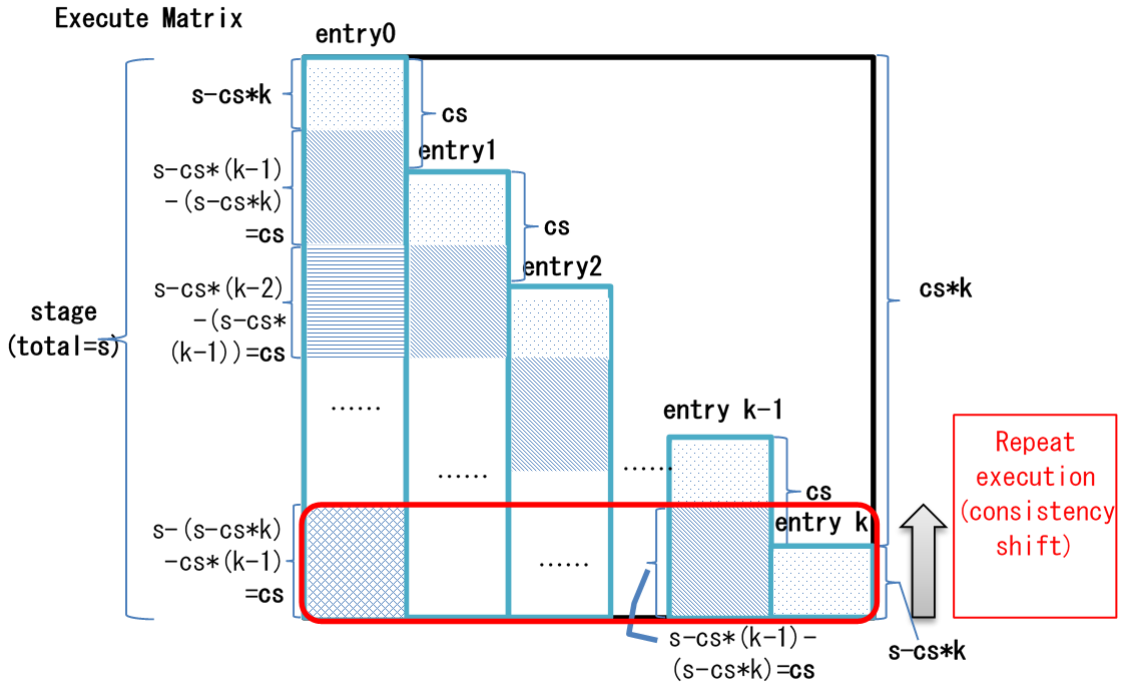
証明. stages が合計 s 行と仮定する。cs は consistency shift の略である。また、下記の証明において、「下へ」は「より大きな番号を持つ stage へ」の意味であり、「上へ」ことは「より小さな番号を持つ stage へ」の意味である。



(a) より大きな番号を持つ stage へ entry0 が一回のみシフトした場合



(b) より大きな番号を持つ stage へ entry0 が二回シフトした場合



(c) より大きな番号を持つ stage へ entry0 が k 回シフトした場合

図 3.7　Execute Matrix の生成の例（定理 2 の証明）

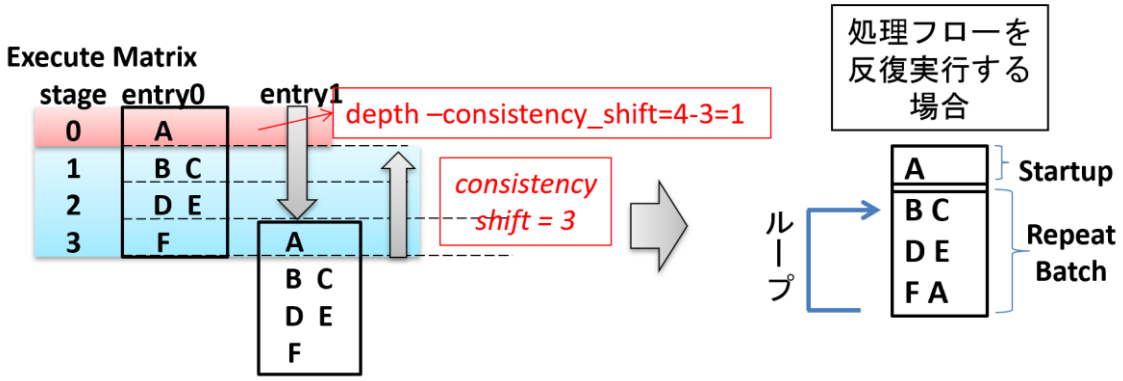


図 3.8　並列実行順序の決定の例

1. entry0 が一回のみ下へシフトされた場合（2 つの entry で生成された場合）：

   図 3.7(a)に示すように、execute matrix に、entry0 は s 行を占める。entry1 は、entry0 を cs で下へシフトしたものなので、s - cs 行を占める。ゆえに、entry0 の最初の stage から s - cs 行は entry1 の最初の stage から s - cs 行と同じである（図 3.7(a)に同じ模様の部分）。よって、entry0 の残りの部分は底部の s - (s - cs) = cs 行である。すなわち、赤線の中の部分は entry0 の全ての stages を含む。すなわち、底部の cs 行は全てのノードを含め、repeat batch 部分になる。

2. entry0 が二回下へシフトされた場合（3 つの entry で生成された場合）：

   図 3.7(b)に示すように、execute matrix に、entry0 は s 行を占める。entry1 と entry2 は entry0 を cs の間隔で下へシフトしたものなので、entry1 は s - cs 行を占め、entry2 は s – cs*2 行を占める。ゆえに、entry0 の最初の stage から s – cs*2 行は entry1 の最初の stage から s - cs*2 行、entry2 の各行と同じである（図 3.7(b)に同じ模様の三つの部分）。ゆえに、entry1 の残りの部分は s – cs - (s – cs*2) = cs 行である。同じく、entry0 にそれと同じ部分を作る（図 3.7(b)の entry1 の真ん中の部分）。そして計算により、entry0 の残りの最後の部分が s - (s – cs*2) - cs = cs 行である。すなわち、赤線の中の部分は丁度に entry0 の全ての stages を含む。すなわち、底部の cs 行は全てのノードを含め、repeat batch 部分になる。

3. entry0 が k 回下へシフトされた場合（k+1 個の entry で生成された場合）：

   図 3.7(c)に示すように、execute matrix に、entry0 は s 行を占める。entry1 から entry k は entry0 を cs の間隔で下へシフトしたものなので、entry1 は s - cs 行を占め、entry2 は s – cs*2 行を占め、同じ、entry k は s - cs*k 行を占める。ゆえに、entry0 から entry k - 1 の最初の stage から s – cs*2 行は entry k の各行と同じである（図 3.7(c)に同じ模様の k + 1 個の部分）。ゆえに、entry k - 1 の残りの部分は s - cs*(k - 1) - (s – cs*k) = cs 行である。同じく、entry0 にそれと同じ部分を作る（図 3.7(c)の entry1 の真ん中の部分）。この過程を繰り返し、そして計算により、entry0 の残りの最後の部分が

$$s - (s - cs * k) - cs * (k - 1) = cs \tag{3.5}$$

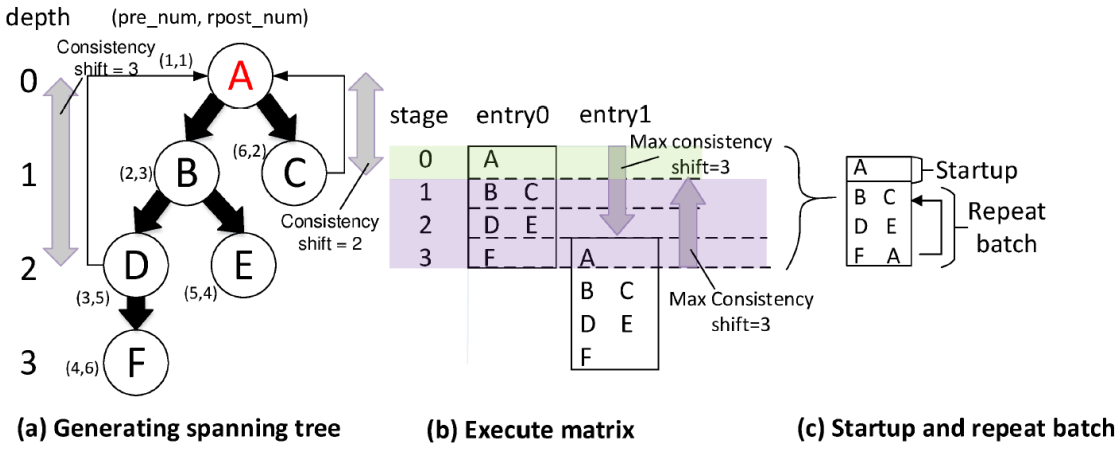


図 3.9　PEA-ST アルゴリズムの全体的な例

行である。赤線の中の部分は丁度に entry0 の全ての stages を含む。すなわち、底部の cs 行は全てのノードを含め、repeat batch 部分になる。

要するに、entry0 の中身を entry1 から entry k との同じ部分で置き換える。最後に entry0 に残りの部分は必ず cs (consistency shift)行である。すなわち、定理 2 を証明した。

図 3.8 に示されている例の場合、consistency shift = 3 のため、repeat batch は底部の 3 行の stage を含む。すなわち、stages (B, C) → (D, E) → (F, A) は repeat batch になる。また、残りの startup は、depth - consistency shift = 4 - 3 = 1 の計算結果により、最初の stage (A)のみになる。このように、構成されたパイプラインにより、同じ stage のノードたちが並列実行し、startup が一度だけ実行する。そして、repeat batch が繰り返し実行する。

3.3　PEA-ST アルゴリズムの実装

C-like コードで記述した PEA-ST アルゴリズムを図 3.10 に示す。Main 関数では、3 つの機能を実現した。1）Spanning Tree の生成、2) consistency shift の演算、と execute matrix の作成、3）startup と repeat batch の作成である。

最初に node[N]と edge[N][N]のデータ構造を作り、処理フローグラフのノードとエッジの情報を保存する。ノード情報には、ノードの名前、depth と更新したかどうかを記録する。エッジ情報には、ノード間に接続しているかどうかを判断する変数とエッジのタイプを記録する変数がある。

まず、循環で Spanning Tree のルートノードになれるノードを探す。3.1 節で述べたように、他の全てのノードに到達できるノードを Spanning Tree のルートノードになれる。最初に見つけたルートノードで Spanning Tree を生成する。

Spanning Tree の作成は、再帰的に呼び出される DFST_Modified 関数によって処理される。関数内ですべてのエッジは、Spanning Tree によって定義された 4 種類に分類される。エッ

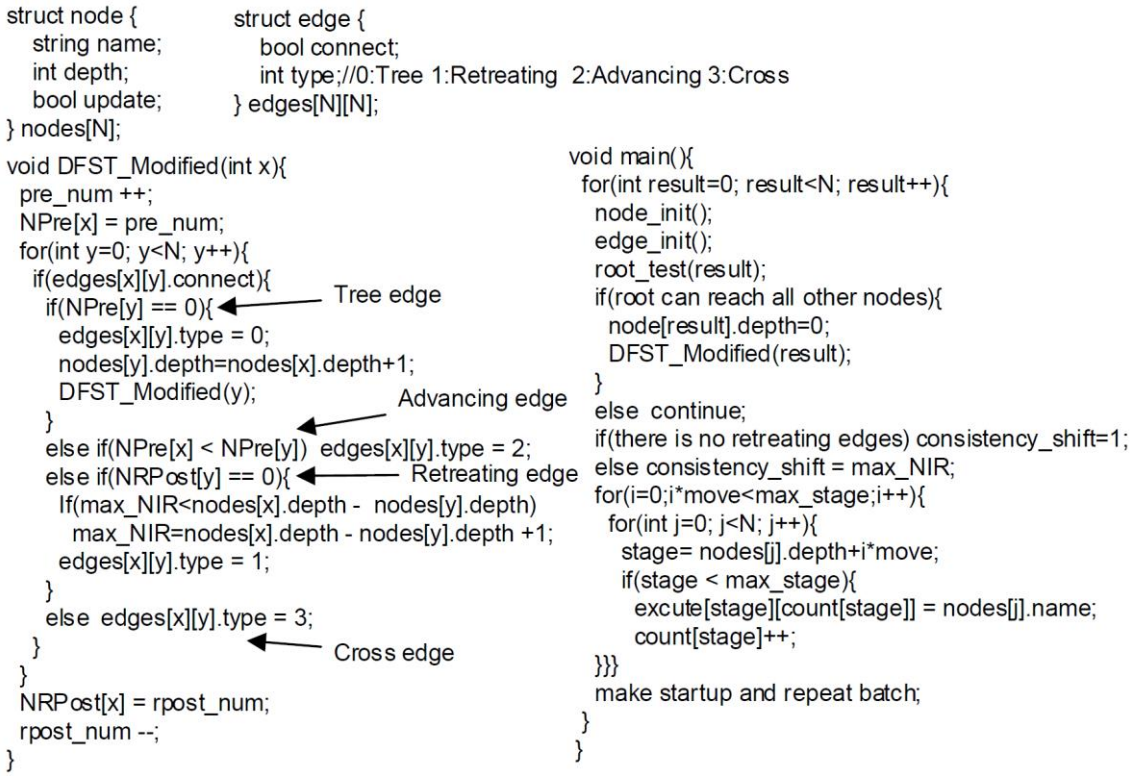


図 3.10　PEA-ST アルゴリズム

ジが分類されると、tree edges を格納する edges 行列は更新される。エッジが retreating edge に分類された場合、max_NIR が retreating edge の最大の NIR（すなわち、consistency shift）に更新される。

Spanning Tree の作成後に、main 関数では、consistency shift を計算する。Retreating edge が存在する場合のみ、consistency shift は使用される。Retreating edge が存在しない場合、consistency shift は 1 になる。これにより、entry と stage を含む execute matrix が作成され、startup 及び repeat batch も作成する。

図 3.9 を用いて全体の処理を説明する。最初の entry0 のように、同じ深さのノードを各 stage に埋める。2 回目では、最大の consistency shift によって 3 行をシフトし、entry1 のように再び stages を埋める。そして、最後の consistency shift 行が repeat batch になり、残りが startup になる。

図 3.11 と図 3.12 に示す 2 つの代表的な例を用いて、PEA-ST アルゴリズムについて記述する。図 3.11 のような直線形の場合、フィードバックがないため、consistency shift は 1 である。execute matrix を生成するため、5 回の繰り返しシフトが必要である。その結果、最大並列度は 5 になる。また、図 3.12 のようなより複雑な場合、入れ子のフィードバックがあり、consistency shift が 5 になる。最大並列度は 3 になる。したがって、アルゴリズム PEA-ST は、フィードバックがない場合とフィードバックがある場合のどちらでも、並列性を抽出できる。2 つの例の処理ステップを以下のように詳しく示している。

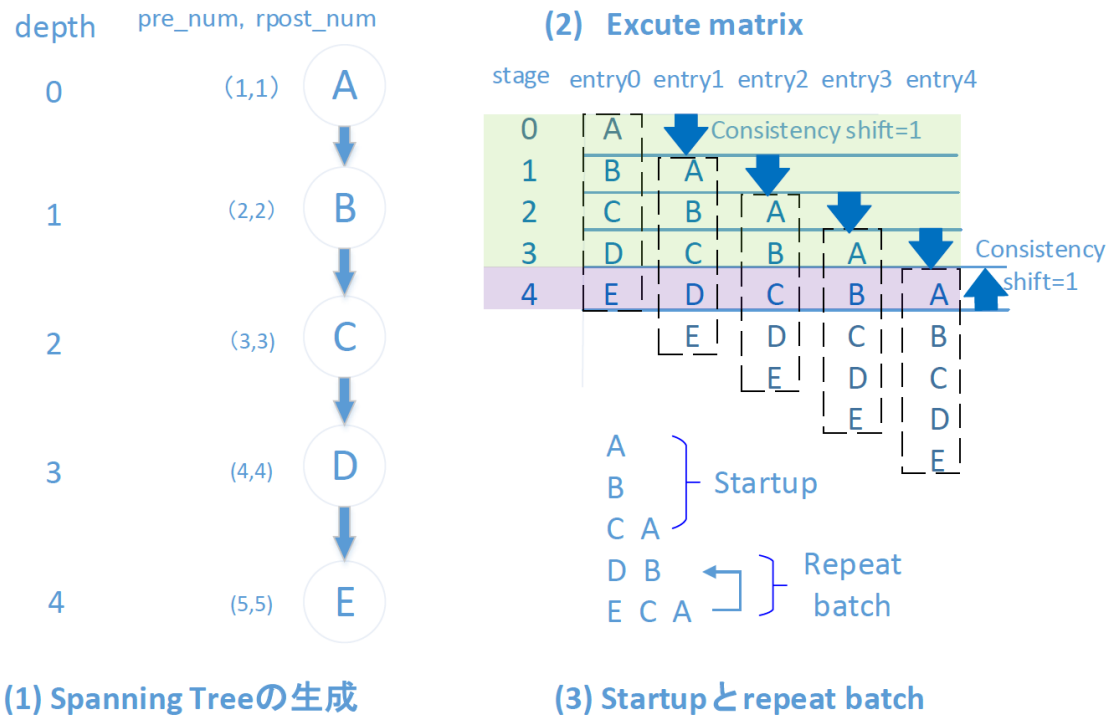


図 3 .11　直線型の PEA-ST アルゴリズの処理ステップ

- 直線型の例

処理ステップ：

(1) ルートになることができるノードを探す。この場合、ルートノードになることができるノードは A のみである。パイプライン化した結果は 1 つのみである。

(2) DFST Modified 関数により、元の処理フローを Spanning Tree に変換する。結果は太い線の部分になる。この間に、retreating edge が発見されないため、max_NIR は 0 のままである。

(3) depth 0 から 4 までの順で図の entry0 のように excute matrix に加える。

(4) max_NIR が 0 のため、consistency shift はデフォルト値の 1 になる。2 行目から entry1 を excute matrix の右側に加える。さらに、entry2 のように、entry1 を下 1 行にシフトして右側に加える。この動作を繰り返す。ただし、stage 4 以後は必要ない。

(5) consistency shift が 1 のため、repeat batch は紫の 1 行の部分である。残りの上部の 3 行は startup になる。並列実行順序の結果は図 3.15 右下のように示されている。

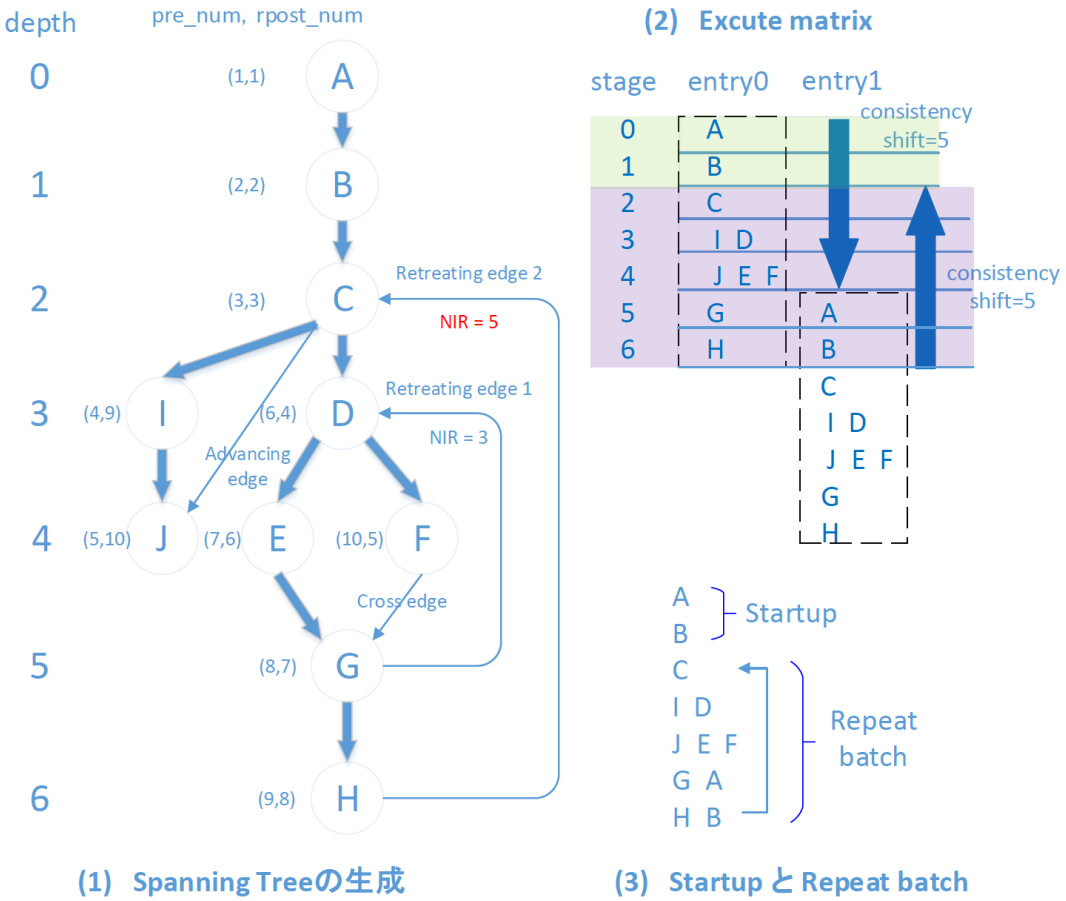


図 3.12　フィードバックを有する複雑型の PEA-ST アルゴリズの処理ステップ

● フィードバックを有する複雑型の例

処理ステップ：

(1) ルートノードになることができるノードを探す。この場合、ルートになることができるノードは A のみである。パイプライン化した結果は 1 つのみである。

(2) DFST Modified 関数により、元の処理フローを Spanning Tree に変換する。結果は太い線の部分になる。この間に、retreating edge を発見し、最長の深度 max_NIR を探す。この図の場合に、H → C は G → D の外側にあるので、max_NIR は 5 になる。

(3) depth 0 から 6 までの順で図の entry0 のように excute matrix に加える。

(4) max_NIR が 5 のため、consistency shift も 5 になる。5 行目から entry1 を excute matrix の右側に加える。ただし、stage 6 以後は必要ない。

(5) consistency shift が 5 のため、repeat batch は紫の 5 行の部分である。残りの上部の 2 行は startup になる。並列実行順序の結果は図 3.16 の右下のように示されている。これで、C が H の結果を必要、同じ、D が G の結果を必要とすることも対応できた。

３.４　まとめ

本章では、並列ストリームコンピューティングにおけるプログラマビリティを向上するために、ストリーム指向プログラムのための並列性抽出アルゴリズム PEA-ST の開発について詳しく述べた。

３.１節では、並列性抽出に関する課題と解決法を述べた。カーネル間の空間的な並列性を時間的な並列性を自動的に抽出し、スループットの向上とレイテンシの削減を図る。それを実現するために、カーネル間の依存性をすべて自動的に見つける必要がある。一方、Spanning Tree の定義と DFST アルゴリズムを述べたことで、Spanning Tree の生成により、ノード間の関係とエッジの分類情報を効果的に見つけることができることを確認した。また、フィードバックやフィードフォワードの I/O が retreating edges や advancing edge として検出されることも確認した。すなわち、Spanning Tree により、各ノード間の依存性を全て発見できることができる。さらに、横方向にある空間的な並列性と、縦方向にある時間的な並列性の特性から、処理フローに含まれるすべてのカーネル間の並列性を非常に効果的に見つけることができることを述べた。

３.１節と３.３節では、PEA-ST アルゴリズムの設計と実装を詳しく述べた。実行順序と並列性を見つけるために、以下の三つの手順が必要となる。1）最初に実行できるノードを見つける。次に、2）I/O バッファが競合しない実行順序を発見する。最後に、3）処理フローから並列性を抽出する。PEA-ST アルゴリズムの全体の流れは図３.3 の例で示されている。まず、ステップ１にルートノードになることができるノードを探す。次に、ステップ２にルートノードから Spanning Tree を生成する。ここで、エッジの分類および consistency shift の計算を行う。そして、ステップ３に consistency shift を適用し、ノード間の並列性を見つけ、Execute matrix を作成する。最後に、ステップ４に並列実行順序を決定し、Execute matrix から startup と repeat batch を生成する。ステップ３とステップ４の説明において、定理 1 と定理 2 の提示と証明を行った。C-like コードで記述した PEA-ST アルゴリズムの実装を示した。Main 関数では、3 つの機能を実現したことを述べた。1）Spanning Tree の生成、2) consistency shift の演算、と execute matrix の作成、3）startup と repeat batch の作成である。

次章では、PEA-ST アルゴリズムの最適化について詳しく述べる。

第４章
並列性抽出アルゴリズムの性能最適化技法

第３章で述べた並列性抽出アルゴリズム PEA-ST は、複数のアクセラレータ環境に適用することもできる。その場合、実環境に合わせた複数の組み合わせから、最適な解を求める必要がある。すなわち、並列化結果が実環境に適しているかどうかは重要な評価指標である。

また、複数のアクセラレータ環境では、通信を考慮しなければならない。本来、実際の計算環境に合わせた通信パターンはプログラマ自身が考える必要がある。並列ストリームコンピューティングにおけるプログラマビリティを向上するために、通信の自動生成も重要な課題である。

さらに、アルゴリズムを実行する際の性能や効率も重要な評価指標となる。本研究では、ロードバランスにより最適化も行う。

本研究は上述のように、幾つかの最適化手法を PEA-ST アルゴリズムに適用し、実環境を考慮しつつ、アルゴリズムの性能効率化を図ることを目的とする。本章は計算環境に基づく並列化手法と通信パターンの自動生成手法を述べ、その二つの機能を実現した PEA-ST アルゴリズムの追加実装を述べ、通信パターンの生成に関する最適化とロードバランスによる最適化を述べる。

４.１　計算環境に基づく並列化手法

３.２節のステップ１：最初に実行できるノードの確定でサイクリックパスについて述べた。一般的に、サイクリックパス内のノード数は１つに限られない。また、最小サイクリックパス内の任意のノードは、Spanning Tree のルートノードとして扱うことができる。すなわち、すべてのノードは Spanning Tree のルートになる可能性がある。各ルートから生成された Spanning Tree は異なるため、それによって構成されたパイプラインも異なる可能性がある。すなわち、ルートが変わると、並列度が変わり、stages の長さも変わる可能性があると考えられる。例えば、３.２節で使用した例で説明すれば、ルートノード A を除く、他の 3 つのノードがルートノードになることもできる。ノード B、C と D をルートノードとした場合の生成された Spanning Tree を図４.1 に示す。この例により、ルートノードが変わると、並列度も変化し、stage 長さも変わることが分かる。

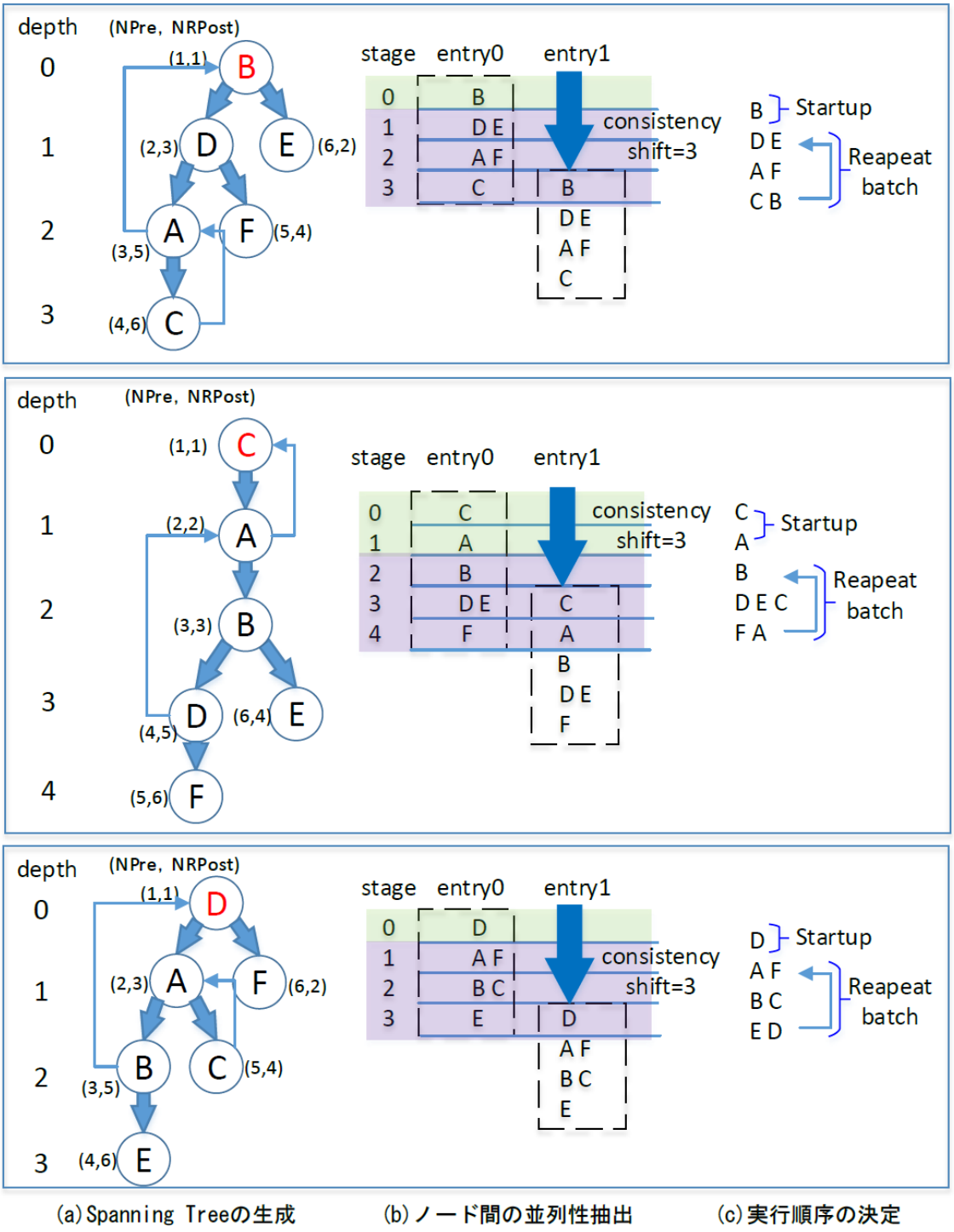


図 4.1　異なるルートノードから生成された Spanning Tree

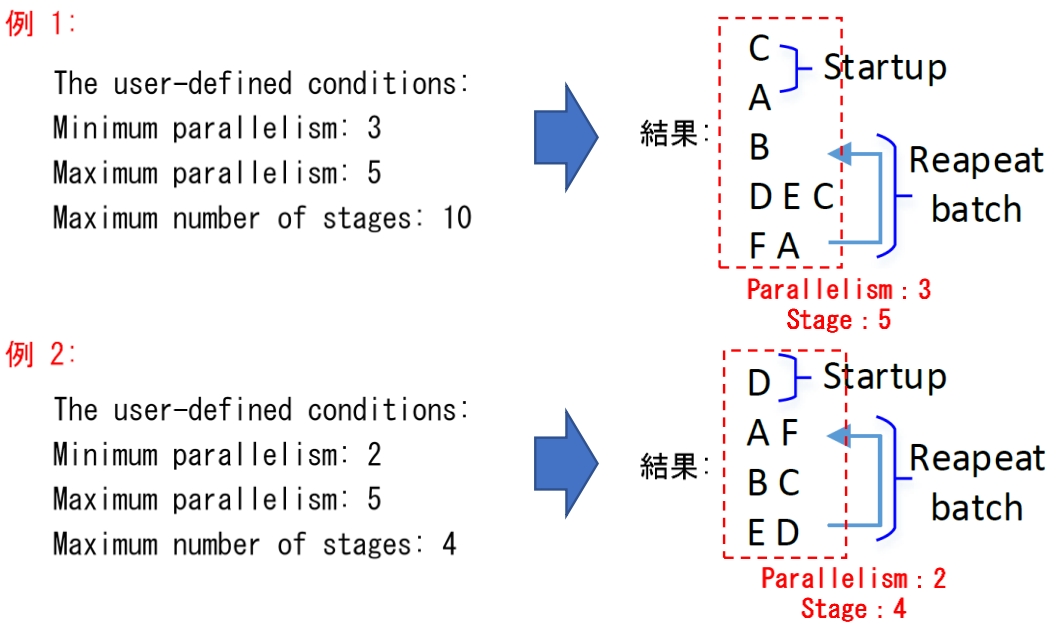


図 4 .2　ユーザ定義条件に基づいた適切な並列化の例

実際の計算環境は様々であり、プログラマは実環境に合わせタスクレベル並列プログラミングが必要とされる。例えば、GPU のようなアクセラレータの数がコンピュータにより異なることがあるため、パイプライン化した結果は異なる計算環境に応じて対応できるようにしなければならない。そのため、PEA-ST アルゴリズムがユーザ定義の条件によって適切な並列パターンを探す必要がある。

ここで、ユーザが最大/最小並列度、最大ステージ数などを定義できるように、アルゴリズムの 1 つの手順として加える。また、entry を consistency shift 以上の行数で下へシフトすることも可能である。そして、PEA-ST アルゴリズムはルートノードを変える方法をベースに、ユーザ定義の条件に当て嵌まる並列パターンを検索し、結果を出力する。図 4 .2 に示す例のように、プログラマが定義した条件に応じて、例をパイプライン化した結果を出力する。すなわち、PEA-ST アルゴリズムの最後に、ユーザ定義条件をチェックする。その条件に当て嵌まる場合、並列パターンをアルゴリズムの結果として出力する。当て嵌まらない場合、アルゴリズムは次のルートノードを探し、Spanning Tree の生成に戻る。

4 .2　通信パターンの自動生成手法

本節では、複数のアクセラレータ環境における通信の自動生成手法について述べる。この手法により、実環境に合わせた通信パターンを自動的に生成できる。

PEA-ST アルゴリズムでは、PE（processing element）は、クラスタのノードまたはアクセラレータの基本となる計算ユニットを意味する。生成された Spanning Tree から得られたエッジ情報に従い、Execute Matrix 内のすべてのカーネルに対し、すべての受信ノードを見つ

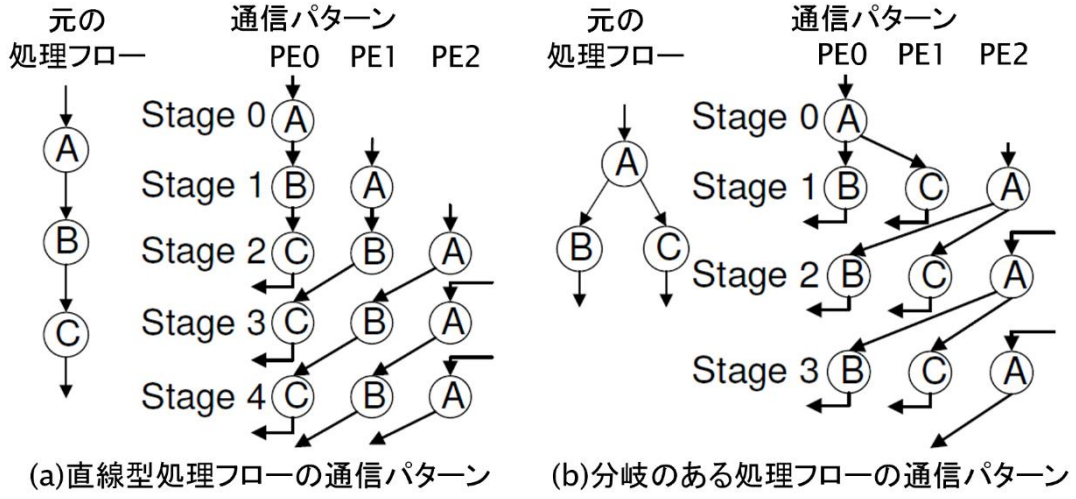


図４.3　通信パターンの生成の例

けて決定する。フィードバックまたはフィードフォワードは特殊なケースとして扱う必要があるため、送信先 PE と受信カーネルの名前を含めて送信カーネルと受信カーネルの間の advancing edge または retreating edge を標記する。Tree edge と cross edge については標記せず、送信先 PE のみを格納する。

ここで、図４.3(a)と図４.3(b)の簡単な例で説明する。図４.3(a)では直線型処理フローの通信パターンを生成する例を示す。左側が元の処理フローであり、右側が生成された通信パターンである。例えば、元の処理フローが A → B → C であり、パイプライン結果が A → (B, A) → (C, B, A) → (C, B, A)…であることは簡単に確認できる。次に、データの流れにより、B → C のエッジが存在するため、通信パターンにはカーネル B が PE0 のカーネル C まで、データを送信する必要がある。このように、Execute matrix 内のすべてのノードを探索した後、図４.3(a)の右側のように全体の通信パターンを得ることができる。図４.3(b)では、分岐のある処理フローの通信パターンを生成する例を示す。この場合、カーネル A はカーネル B と C へそれぞれにデータを送信する必要があるため、PE0 と PE1 を送信先として格納する必要がある。

４.３　PEA-ST アルゴリズムの追加実装

４.１節と４.２節で述べた計算環境に基づく並列化と通信パターンの自動生成の手順を PEA-ST アルゴリズムに追加実装を行った。図４.4 に PEA-ST アルゴリズの全体的な流れを示す。図にある 6 つの処理ステップに合わせ、図４.5 に示すように新しい Main 関数では、6 つの機能を実現した。1）最初に実行できるノードの確定、2）Spanning Tree の生成、3）consistency shift の演算、と execute matrix の作成、4）startup と batch の作成、5)ユーザ定義の要求に応じて結果の更新、6）通信パターンの生成である。この関数ではすべての可能

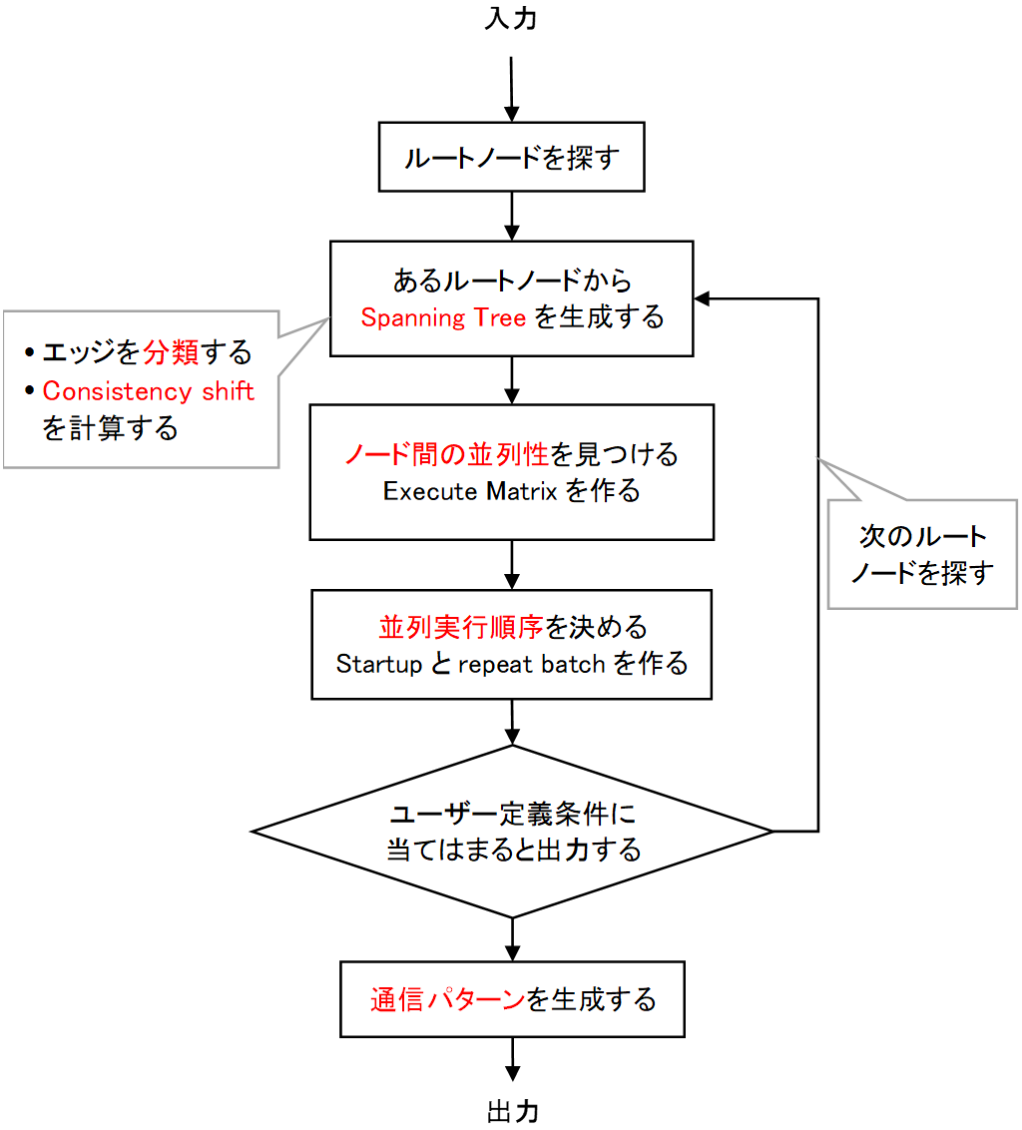


図 4 .4　PEA-ST アルゴリズの全体的な流れ

な Spanning Tree をチェックし、適切な並列化結果を出力する。

手順 4）では、ルートノードを選択するループをブレークし、適切な startup と repeat batch を返す。図 4 .5 の右側にある上の 2 つの赤いボックス内には、手順 4）を実現した。可能なルートノードを循環でトラバースし、entry のシフト行数もインクリメントすることで、すべての並列パターンを見つけ出す。その過程では、最大並列度などのユーザ定義条件がチェックされる。リソースの制限（アクセラレータの数）によって決定される最大並列度で実行できる並列パターンを見つけられる。

通信パターンを自動的に生成するために、図 4 .5 の左側の赤いボックスのように通信パターンを格納するデータ構造を作成した。PE_id_feedback_or_forward という変数はフィードバックあるいはフィードフォアド（i.e. retreating edge or advancing edge）の送信先カーネルの

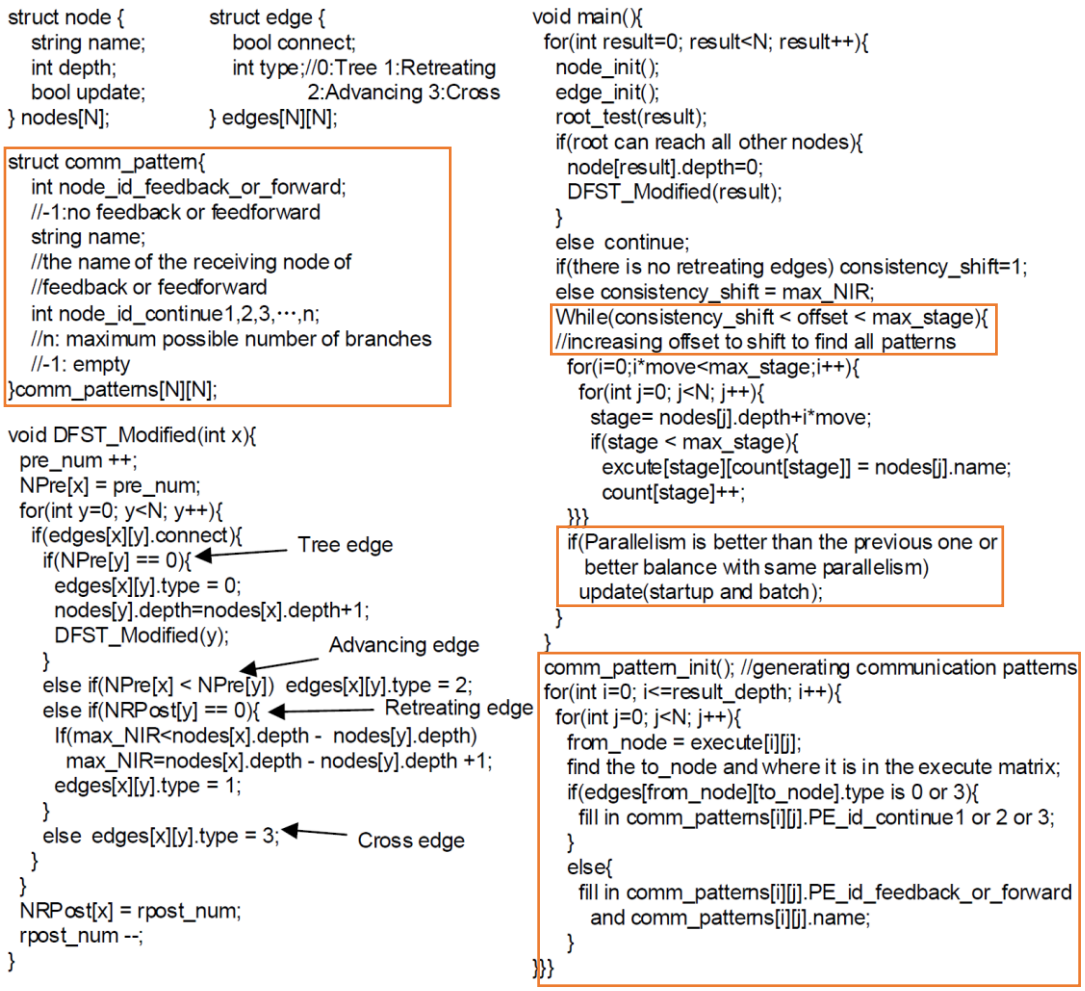


図 4 .5　PEA-ST アルゴリズムの全体的な実装

PE を格納するために使用される。送信先カーネルの名前は第 2 メンバーとして格納される。また、カーネルからのフィードバックやフィードフォワードがない場合、PE_id_feedback_or_forward は-1 になり、名前も空になる。特殊な場合を除き、通常（i.e. tree edge or cross edge）の送信先は、PE_id_continue に格納する。PE_id_continue は 1 から n（n：可能な最大分岐数）までである。PEA-ST アルゴリズムの終わりに、Execute Matrix 内の各カーネルの送信先を全部見つけ出し、通信パターンの行列を埋め込む。

N をノード数、E をエッジ数としたとき、PEA-ST の複雑性は、DFST アルゴリズムをベースにして考えられる。先行順番号と逆後行順番号付けは O(NE) の複雑性を持つ。しかし、PEA-ST はすべての可能なルートから Spanning Tree を生成する必要がある。したがって、ルート数を M としたとき、複雑性が O(NEM) になる。

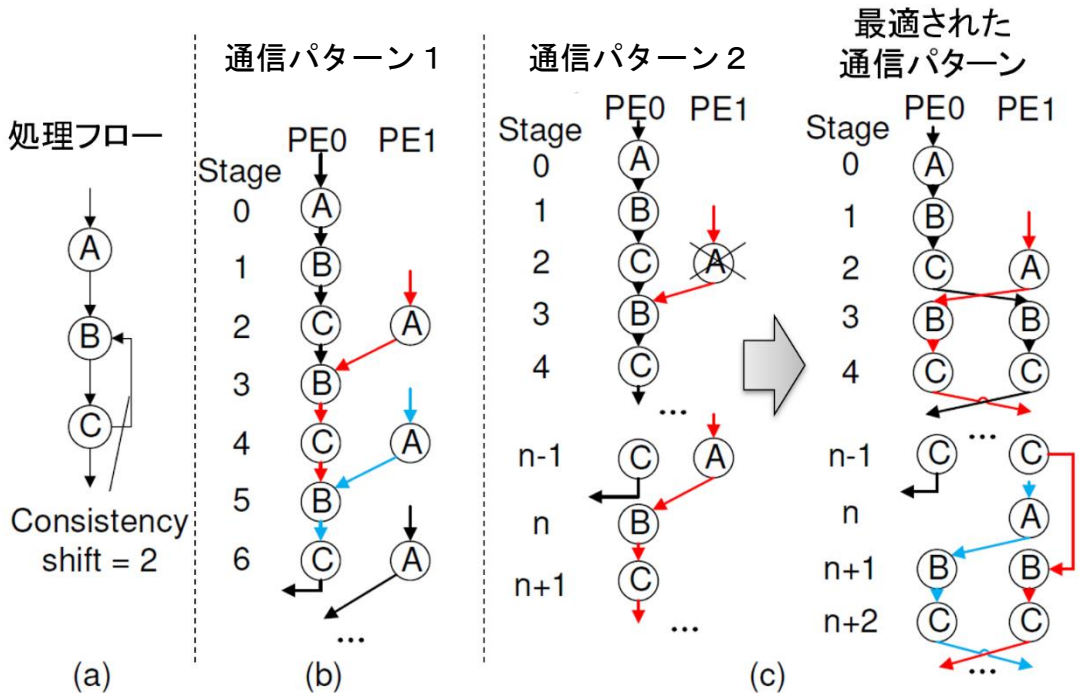


図４.6　フィードバックを用いて通信パターンの最適化を行った例

４.４　通信パターンの生成に関する最適化

この節では、フィードバックを有する処理フローの通信パターンを考える。各回のループの終わりに、フィードバックに従い、ループを継続するか終了するかは、ループの終了条件に応じて決定する必要がある。すなわち、送信先ノードは未定であり、実行が retreating edge または tree edge を通過するかどうかに依存する。PEA-ST アルゴリズムは、両方の場合の送信先ノードを格納する必要がある。

フィードバックまたはフィードフォワードを有する処理フローの通信パターンを生成するために、以下の 2 つのケースを考慮する必要がある。ケース 1：対象ノードが tree edge と retreating edge の両方のデータストリームを必要とする場合である；ケース 2：もう一つは、対象ノードが retreating edge のデータストリームのみを必要とする場合である。

これらの 2 つのケースを図４.6 の例で詳しく説明する。図４.6(a) では、単一のフィードバックを有する処理フローの例を示す。Retreating edge の C → B の NIR の値は 2 である。この処理フローを前述のケース 1 として考えると、ノード B は tree edge である A → B と retreating edge である C → B の両方のデータストリームを必要とする。したがって、PEA-ST は図４.6(b)のような結果を出力する。パイプライン実行順序は A → B →(C, A) → B → (C, A)...になる。ループが stage 6 で終了する場合、パイプライン実行は新しいデータストリームの処理を開始する。

しかし、図４.6(a)の処理フローをケース 2 として考えると、図４.6(c)の左側のように、ノード B は tree edge の A → B のデータを一回だけ処理した後、retreating edge の C → B のデ

ータだけを必要とする。パイプライン実行順序は図４.6(c)の通信パターン 1 で示す。ノード A は PE0 上のループ中のノード B に送信できないため、通信パターン 1 では並列性が得られない。すなわち、ノード A は、PE0 上のループの完了を待つ必要があり、時間の無駄を引き起こす。そこで、図４.6(c)の右側のように最適化することができる。Stage 2 では、ノード C と A が並列に実行され、結果を交換する。ループが PE0 上の stage n-1 で終了する場合、PE1 上の C の結果は一時的に記憶され、stage n+1 に渡される。そして、ノード A は PE1 上の stage n で再び実行され、パイプライン実行も継続できる。これにより、通信パターン 2 で処理フローの並列度を高めることができた。

上述の例では、2 つの PE 上の並列実行に対して通信パターンの最適化を述べたが、これを一般化し、p 個の PE で処理する場合について考える。p 個の PE で実行されるフィードバックのある処理フローを考える。PE1 上での実行は、PE0 上のループの完了を待たなければならない。同様に、PE2 上での実行は、PE1 上のループの完了を待たなければならない。すなわち、PE p での実行は、常に PE p-1 上のループの完了を待つ必要があり、時間の無駄を引き起こす。そこで、PE の数に応じて n 個の PE 間で結果を交換する方法で通信パターン生成の最適化を一般化することができる。すなわち、特殊ケースも含めて、すべての処理フローの通信パターンを自動的に生成できる機能を完成できる。

４.５　ロードバランスによる最適化

より高いパフォーマンスを得るために、ロードバランスを考慮する必要がある。その重要なポイントは PE アイドル時間を最小限に抑えることである。PE 間でワークロードを均等に分散させることで、アイドル時間が減少し、パフォーマンスを向上させることができる。

PEA-ST アルゴリズでは、各 stage の実行時間はその stage にあるカーネルに依存する。一般的には、各カーネルの実行時間はそれぞれ異なる。該当 stage の実行時間はその中の実行時間が最も長いカーネルによって決定される。そのため、PE 間の計算のロードバランスは、PEA-ST アルゴリズムの最適化として考慮する必要がある。

ここでは、PEA-ST アルゴリズムにおいて静的なロードバランスを考慮する。例えば、図３.9(c)では、ノード B と C、ノード D と E、ノード F と A はそれぞれの stage で PE0 と PE1 上で実行する。B と C の実行時間、D と E の実行時間、F と A の実行時間が同じである場合のみ、負荷が均衡している。しかし、実環境では、常に同じ実行時間で終わるとは限らない。したがって、PEA-ST アルゴリズムは、カーネルの実行時間をもとに、それらを PE 間で均等に分散させる方法を考慮する必要がある。

詳しい説明は 5.2 節のロードバランスによる最適化のところに例で述べる。

４.６　まとめ

本章では、ストリーム指向プログラムのための並列性抽出アルゴリズム PEA-ST の性能を向上させるための最適化手法について詳しく述べた。

４.１節では、計算環境に合わせた組み合わせから、適切な並列パターンを選択し出力できる機能を PEA-ST アルゴリズムの最適化として実現した。この手法を、複数アクセラレータ環境に適用し、プログラマがアクセラレータの数、すなわち最大並列度、を入力するだけで並列パターンを決定できる。

４.２節では、複数アクセラレータ環境でプログラミングする場合、PEA-ST アルゴリズムが実際の計算環境に合わせた通信パターンを決定する機能を実現した。プログラマが通信パターンを考える必要がなくなり、複雑な作業から解放される。４.１節の計算環境に対応した最適化手法を用いて、複数アクセラレータ環境での並列ストリームコンピューティングのプログラマビリティを向上させることができる。

４.３節では、４.１節の計算環境に対応した最適化手法と４.２節での通信パターンの自動生成手法を PEA-ST アルゴリズムの追加機能として実装した。

４.４節と４.５節では、PEA-ST アルゴリズムの性能効率化を目的として、フィードバックが存在する場合に生じた並列性が得られない問題に対して最適化を行い、各カーネルの実行時間を考慮したロードバランスによる最適化も行った。

次章では、PEA-ST アルゴリズムの性能評価について詳しく述べる。

第５章
関連研究

第２章で述べたように、アクセラレータを用いた並列プログラミングはプログラマにとって極めて困難な作業である。そのプログラマビリティを向上するために、様々な言語やツールが開発されてきた。GPU のための並列プログラミング言語については、２.１節に述べた OpenCL と CUDA が広く使われている。また、いくつかのほかの手法が存在する。一つは、プログラマがプログラムに並列化のための指示文を追加する手法である。もう一つは、Pareon [61]および ParaWise[62][62]のような、プログラマと並列化ツール/コンパイラの間にインタラクティブシステムを構築する手法である。

SkelCL[63]はドイツのミュンスター大学において開発が進められている研究プロジェクトである。最近の異種並列システムでのプログラミング負担を軽減するための高レベルの抽象化を提供するライブラリである。マルチコア CPU や GPU などを含む異種並列システムでのプログラミングは、複雑でエラーを招きやすい。CUDA や OpenCL のような既存のプログラミング言語は、本質的に低レベルである。複数の計算デバイスの取り扱いは複雑で、手動でこれらの間でデータを移動させる必要がある。そこで、SkelCL ライブラリは、OpenCL 標準の上に構築され、異種並列システムのプログラミングを大幅に簡素化できる反復計算および通信パターンのスケルトンを提供する。しかし、SkelCL ライブラリでは、抽象的なベクトルデータ型とハイレベルデータ型メカニズムを提供するが、ストリーム指向プログラムに適用できない。そのため、自動パイプライン化も提供しない。さらに、処理フローにフィードバックがある場合、プログラマ自身が手動で処理する必要がある。

Pareon[61]は、オランダの Vector Fabrics がアプリケーションをマルチコアシステム向けに最適化するために開発されたツールである。Pareon はスレッド周りのバグ検出を試みる分析ツールと、開発者がパフォーマンスボトルネックを発見するためのハードウェアモデリングエンジンで構成される。分析ツールにより、プログラムが可視化され、開発者にプログラムの実行、内部依存関係、メモリ動作などを提示する。Point と Click のインタフェースにより、開発者は並列化したい箇所を追加し、並列化を妨げるコードがあればすばやく特定し、取り除くことができる。Pareon は単一のマルチコアプロセッサのメモリやキャッシュのボトルネック、同期や通信、スレッドスケジューリングのオーバヘッドを考慮した。しかし、マルチ GPU 環境を考慮していない、ストリーム指向プログラム向けのパイプライン化もサ

表 5.1　関連研究との比較

研究	複数 GPU 環境	自動タスク並列化	自動パイプライン化	フィードバックやフィードフォアドを対応できる	明示的な指示を書く必要がない	通信パターンの自動生成	ロードバランス
PEA-ST	○	○	○	○	○	○	○
SkelCL	○	○	×	×	×	○	×
Pareon	×	×	×	×	×	○	○
ParaWise	×	×	○	×	×	○	○
Copperhead	×	×	×	×	○	×	×
Falcon	○	×	×	×	×	×	×
Par4All	×	×	×	×	○	×	○
Bones	○	×	×	×	○	○	×
SkePU 2	△	○	×	×	×	△	○
StarPU	○	○	○	×	×	○	○
work in [70]	○	○	×	×	×	○	○
Streamlt	×	○	○	×	○	×	○
work in [72]	×	○	○	○	○	○	○
work in [73]	○	○	○	×	○	○	○

ポートしない。

ParaWise[62]は、シリアル C および Fortran コードを読み取り、MPI、OpenMP またはその両方のハイブリッドに基づいて、効果的で拡張性と移植性のある並列コードを生成することにより、並列ソフトウェア開発のプロセスをサポートする。このコード変換により、スーパーコンピュータからマルチコアプロセッサまでのような並列システムの性能を引き出し、アプリケーションのコードを高速化できる。科学者とアプリケーション開発者に対し、OpenMP での専門知識を必要とせずにアプリケーションコードを高速に並列化できる。しかし、ParaWise は細粒度の並列性を抽出することができるが、タスクレベルの粗粒度の並列性は自動抽出できない。また、他のほとんどのツールと同じく、プログラマ自身が並列化したい箇所を見出し、指示を出す必要がある。

Copperhead[64]は、Python に埋め込まれたデータ並列言語であり、効率的な並列コードを作成するコンパイラも組み込まれている。現在、Copperhead コンパイラとランタイムは、CUDA 対応 GPU、および、TBB（Threading Building Blocks）または OpenMP を用いたマルチコア CPU をターゲットにしている。Falcon[65]は C 言語の拡張としたグラフ操作言語である。グラフアルゴリズムに関連する Point、Edge、Graph、Set と Collection のデータタイプを提供し、CUDA カーネルあるいは OpenMP 指示文を自動に生成できる。Par4All[66]は、C および Fortran の逐次プログラム用の自動並列化および最適化コンパイラである。コンパイラは完全自動であるが、入力コードに制限があり、静的アフィンループネストのみが変換される。また、マルチ GPU もサポートしない。そのため、Bones[67]では並列プログラミング

のプログラマビリティを向上させるために、並列プロセッサ用のコードを自動的に生成する技術を提案した。algorithmic species というプログラムのコード分類手法により、C コードを CUDA コードに変換できる。効率的なコードを生成するために、Bones はホストアクセラレータ間の転送最適化とカーネル融合を含む。Bones は指示文なしにコード変換の自動化を実現したが、タスク並列化やパイプライン化などの粗粒度の並列性を考慮していない。

SkePU 2[68]は SkePU からの進化であり、マルチコア CPU およびマルチ GPU システムのためのオートチューニング可能なマルチバックエンドに対応できるスケルトンプログラミングフレームワークである。Map、Reduce、MapReduce、MapOverlap、Scan と Call のスケルトンを提供する。データ並列化とタスク並列化のスケルトンも提供する。また、同期、通信、メモリ管理、アクセラレータの使用、その他の最適化などもサポートする。SkePU 2 では、ローレベルとプラットフォーム固有の詳細情報をカプセル化することで、プログラマビリティを向上させる。しかし、MPI を含んだノード間の通信は実現されていない。ストリーム指向プログラムに適用できるが、パイプライン化に関するスケルトンも提供していない。

StarPU[69]はマルチ CPU やマルチ GPU を持つハイブリッドアーキテクチャ用のタスクプログラミングライブラリである。可搬性は、システムの統一された抽象化によって得られる。StarPU は、統一されたオフロード可能なタスク抽象化コードレットを提供する。プログラマは、コード全体を書き直すことではなく、既存の関数をカプセル化することができる。StarPU は、複数の GPU を含め、マシン全体で可能な限り効率的にスケジューリングと実行を行うことができる。明示的にデータ転送に関する指示を書く負担からプログラマを解放するために、高レベルのデータ管理ライブラリが提供される。コードレットを開始する前に、すべてのデータが自動的に計算リソースで利用できるようになる。また、プログラマに柔軟性を提供するために、タスク間の依存関係は、完全に分散された方法で計算される。また、複数のタスクに対して異なる処理ユニットによって達成される相対的な性能を決定し、それにより処理ユニットに自動的に最適なタスクを実行させる。クラスタで実行する場合、MPI 通信も自動的に生成できる。したがって、StarPU により、ローレベルの問題を扱うことなく、プログラマはアルゴリズムレベルの問題に集中できる。しかし、プログラマが明示的に指示を書く必要がある。さらに、タスク間での並列性抽出には制限もある。

論文[70]では、シリアルコードから利用可能なリソースへの計算を適応するタスクスーパースカラ実行環境を使用することにより、並列性とスケーラビリティを実現できるプログラミングモデルを提案する。この研究では、マルチレベルの並列性を利用でき、動的タスクスケジューリングを提供する。CUDA stream をサポートするが、ストリーム指向プログラムからパイプライン化することはできない。前述の研究と同じ、プログラマが明示的に指示を書く必要がある。

StreamIt[71]はストリーム指向プログラムのためのプログラミング言語であり、大規模なストリーミングアプリケーションのプログラミングを容易にするとともに、市販のユニプロセッサ、マルチコアアーキテクチャ、ワークステーションのクラスタなど、さまざまなタ

ーゲットアーキテクチャへの効率的かつ効果的なマッピングをより簡単にできるように設計されている。階層構造化ストリーム、グラフパラメータ化、循環バッファの管理、テレポートメッセージングなど、プログラマのプログラミング効率を向上させる言語構造を持っている。完全自動ロードバランシング、グラフレイアウト、通信スケジューリング、およびルーティングを備えた最適化コンパイラである。しかし、パイプライン化は実現されているものの、複数の GPU 環境での実行は考慮されていない。また、これらの研究は、フィードバックを有する処理フローに対応していない。論文[72]では、StreamIt プログラムを C および CUDA コードへの変換手法が提案されている。しかし、通信を必要とする複数 GPU 環境での実行はサポートしていない。また、論文[73]では、タスクレベルの並列性を利用するために、複雑な StreamIt アプリケーションを複数のパーティションに分割できる効率的なグラフ分割アルゴリズムを提案されている。しかし、フィードバックを伴うループを扱うことはできない。

一方、ソフトウェアのパイプライン化は高速実行を実現するためにループを再編成する手法である。モジュロスケジューリングはソフトウェアパイプライン化のループを生成するためのコンパイラ技術であり、効果的な手法として知られている。モジュロスケジューリングの目的は、循環する操作の間に相互依存かつリソース使用競合を発生しない間隔で、パイプラインを繰り返すことである。Rau と Glaeser は、最初の多環状アーキテクチャ向けのソフトウェアパイプライン機能を持つコンパイラを開発した[74]。また、modulo renaming は Lam の技術として実際に広く使用されている[75]。これに関する最近の研究として、Decoupled Software Pipelining(DSWP) と呼ばれる自動スレッド抽出手法がある。DSWP は長時間で同時に実行しているスレッドを抽出するため、アプリケーションに潜在する細粒度パイプライン並列性を利用する。DSWP は実行効率を向上させ、大幅なレイテンシの許容範囲を提供することができる[76]。これらに対し、アルゴリズム PEA-ST はタスクレベルパイプライン処理を実現している。

論文[77]は、ステンシルの通信パターンでタスクの配置を最適化することで並列処理のパフォーマンスを向上させることに重点を置かれている。MPI ランク情報に基づいて、コアごとに実行タスクと通信パターンが決定される。主に MPI ランクごとにコアへのタスクマッピングが構成されるため、複数 GPU 環境での並列実行をサポートしない。それと比べ、PEA-ST アルゴリズムは、処理フローから MPI 通信パターンを生成する。論文[78]は遺伝的アルゴリズムを使用してタスクをリソースへマッピングするが、データ並列化、タスク並列化とパイプライン化は利用しない。

ロードバランスに関する問題も長年にわたって研究されてきた。これは、タスクをアイドルコアに動的に割り当てる動的ロードバランス[79]と、タスクを実行する前に決定する静的ロードバランス[80]の 2 種類に分けることができる。PEA-ST アルゴリズムは、カーネルの実行時間で並列パターンを生成する際に静的ロードバランスを考慮する。

上述のように、多くのプログラミング言語は、細粒度並列プログラミングとベクトル処理

のために設計されている。プログラマは、どのように複数のプロセッサにデータを配布するか、および、どのような並列実行をスケジュールするかを設計する必要がある。PEA-ST アルゴリズムはカーネル間、すなわち、タスクレベルの並列性に注目し、プログラマがタスクの配布やスケジュールなどを設計する必要がない。

すなわち、本研究は、空間的並列性、時間的並列性の双方を考慮しており、データ並列化、タスク並列化とパイプライン化の全てを実現可能な並列化手法であり、スループットの向上とレイテンシの削減で、高い計算能力を実現できる。また、フィードバックを有する処理フローに対しても効率良い並列化が可能な点に特徴がある。ストリームコンピューティングにおけるプログラマビリティを向上させるために、本研究の並列パターンと通信パターンの自動生成機能により、プログラマが下層ハードウェアの構成を知る必要がない。上述の研究のようにコードの中に明示的に並列処理に関する記述を書かなくても、PEA-ST アルゴリズムはデータストリームで接続された処理フローを自動的に並列化できる。また、本研究により、大量処理データを持つ複数回の演算が必要とするストリーム指向プログラムに向け、GPU、FPGA のような異なるアクセラレータに応用でき、高性能化を実現できる。

表 5.1 に、本研究と関連研究の特徴、機能の比較を示す。

第６章
性能評価

本論文では、PEA-ST アルゴリズムの有効性と性能最適化を評価するために、画像処理と数値解析に関連する 3 つのアプリケーションを開発した。FFT を用いた画像フーリエ変換と LU 分解のアプリケーションを用いて本アルゴリズムの有効性に関する性能評価を行った。k-means と前述の 2 つのアプリケーションを用いて本アルゴリズムの性能最適化に関する性能評価を行った。k-means アプリケーションを用いた評価ではフィードバックを有する通信パターンの最適化に用いて評価した。また、FFT フーリエ変換と LU 分解の結果を用いてロードバランスによる最適化の評価に関して説明する。

全ての実験は、16 ノードで構成されたクラスタで行なった。クラスタのノード間は、40GByte/秒の Infiniband ネットワークで接続される。各ノードは、12 GB DDR3 メモリを搭載した Intel(R) Xeon(R) E5645 @ 2.40GHz の CPU と PCI Express 2.0 x16 インタフェースを介して接続された NVIDIA Tesla M2050 の GPU を持つ。アプリケーションは、OpenCL、および C++言語を用いて開発し、並列化に関しては、OpenCL1.0 ランタイムと Open MPI ライブラリを使用した。

６.１　PEA-ST アルゴリズムの有効性に関する評価

６.１.１　　画像フーリエ変換

(1)　アプリケーション概説

PEA-ST アルゴリズムの性能を評価するために、プログラマが書いた画像フィルタ処理と、それを PEA-ST アルゴリズムにより並列化した処理フローを比較しながら解説する。図 6 .1 の左側に FFT を用いた画像フーリエ変換の例を示す。FFT 処理により、画像は空間周波数に変換され、IFFT 処理により、周波数領域の情報から画像に戻すことができる。したがって、周波数ベースの画像フィルタリングが可能になる。High-pass フィルタリングは、低周波成分がカットされることを意味し、画像の急激な変化が残る。これは、図 6 .1 の左上隅に示すように、「Edge Enhancement」とも呼ばれる。それと反対に、low-pass フィルタリングは、高周波成分がカットし、図 6 .1 の左下隅に示すようにエッジがぼかすことを意味する。

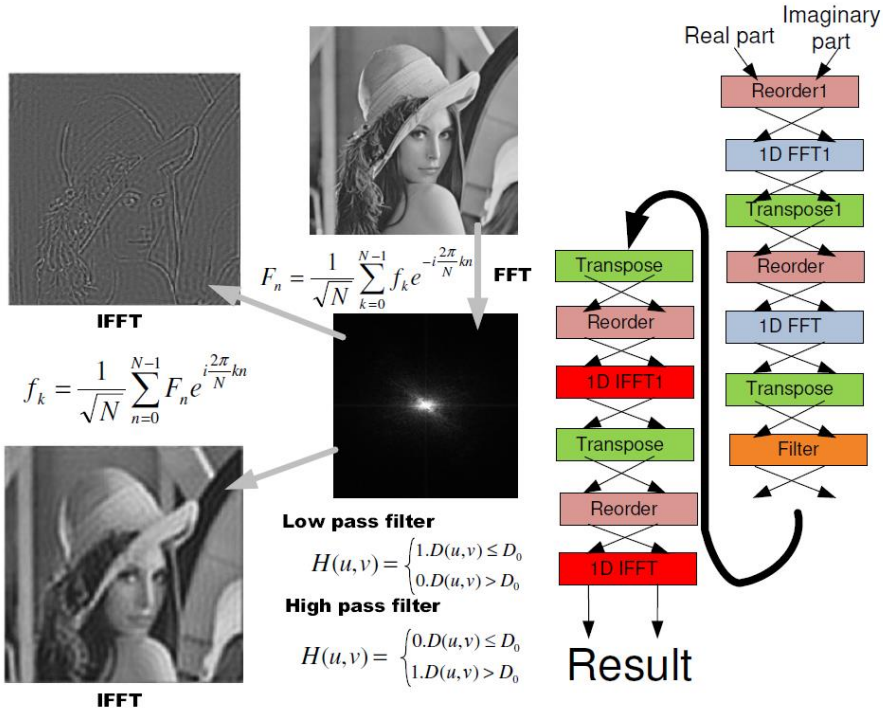


図 6 .1　FFT を用いた画像フーリエ変換の処理フローの例

図 6 .1 の右側に処理フローの例を示している。本アプリケーションは、13 個、5 種類のカーネルで構成される。Reorder ではビット反転インデックスによって入力データを並べ替える。FFT と IFFT では 1D FFT と 1D IFFT 処理を行う。Transpose では、2 次元の行列データの転置を行う。Filter では、high-pass または low-pass フィルタリングを行う。

(2)　PEA-ST による並列化

図 6 .2 では、PEA-ST アルゴリズムによって生成された処理パイプラインを示している。パイプライン方式では、フィードバックがないため、consistency shift が 1 である場合、処理フローの最大並列度は 13 になる。ただし、必要な通信量も最大になるため、シリアル実行に対するスピードアップは最大並列度よりはるかに低くなる可能性がある。

また、前述したように、PEA-ST アルゴリズムによって生成される並列パターンは、複数の並列パターンから MAX_PARALLELISM パラメータによって決定される。ここでは、4 個の GPU を使用する際の並列化手順について説明する。例えば、ユーザ定義条件の MAX_PARALLELISM が 4 に従い、PEA-ST アルゴリズムにより生成された並列パターンは図 6 .2 に示した consistency shift = 4 の場合の結果である。まず、Spanning Tree が生成され、retreating edge がないため、consistency shift の最小値が 1 になる。すなわち、パイプラインを作る際に間隔を 1 以上に空ける必要がある。次に、PEA-ST アルゴリズムは、シフト数を増加させることにより、適切な結果を見つける。execute matrix を生成するには、entry0 のように同じ深さのノードで各ステージを埋める。consistency shift が 4 である場合、entry1 の

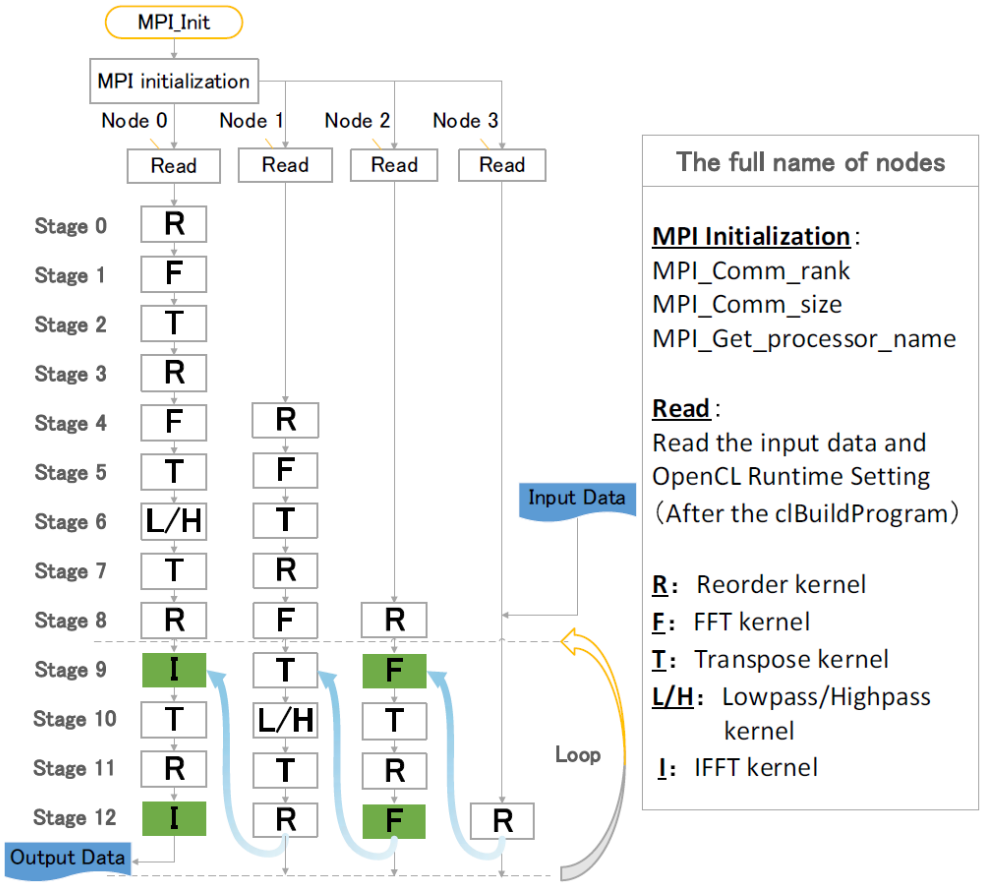


図 6 .2　FFT 例の並列パターン（consistency_shift = 4 の場合）

ように entry0 を 4 行下にシフトする。同様に、entry2 と entry3 を生成し、execute matrix を完成する。ここで、consistency shift が 4 になると、並列度は 4 になり、ユーザ定義条件も満たすため、生成された並列パターンを結果として保存する。PEA-ST アルゴリズムにより生成された execute matrix は、13 の stage と 4 つの entry を持つ。各 entry は、クラスタの各ノード上で実行される。最後に、execute matrix から startup と repeat batch を生成する。図 6 .2 に示すように、上の方が startup であり、下の 4 つの stage は repeat batch である。

また、図 6 .2 に青色の矢印で通信パターンも示している。各ノードに最後の stage（eq. stage 12）の実行が終われば、一時的な結果はノード N（N:1~3）からノード N-1 へ次の stage に送られる。受信が完了すると、ループは継続する。これらのノード間の通信に関しては、同期通信と非同期通信の両方で実験した。

(3)　実行結果

性能を評価するために、シリアル化と並列化の最後の IFFT までの実行時間を測定した。毎回の実行時間が短いため、どちらの場合も repeat batch のループは 100 回実行した。パイプライン化されたバージョンの実行時間は、図 6 .2 に示すように、Node 0 に最初のステー

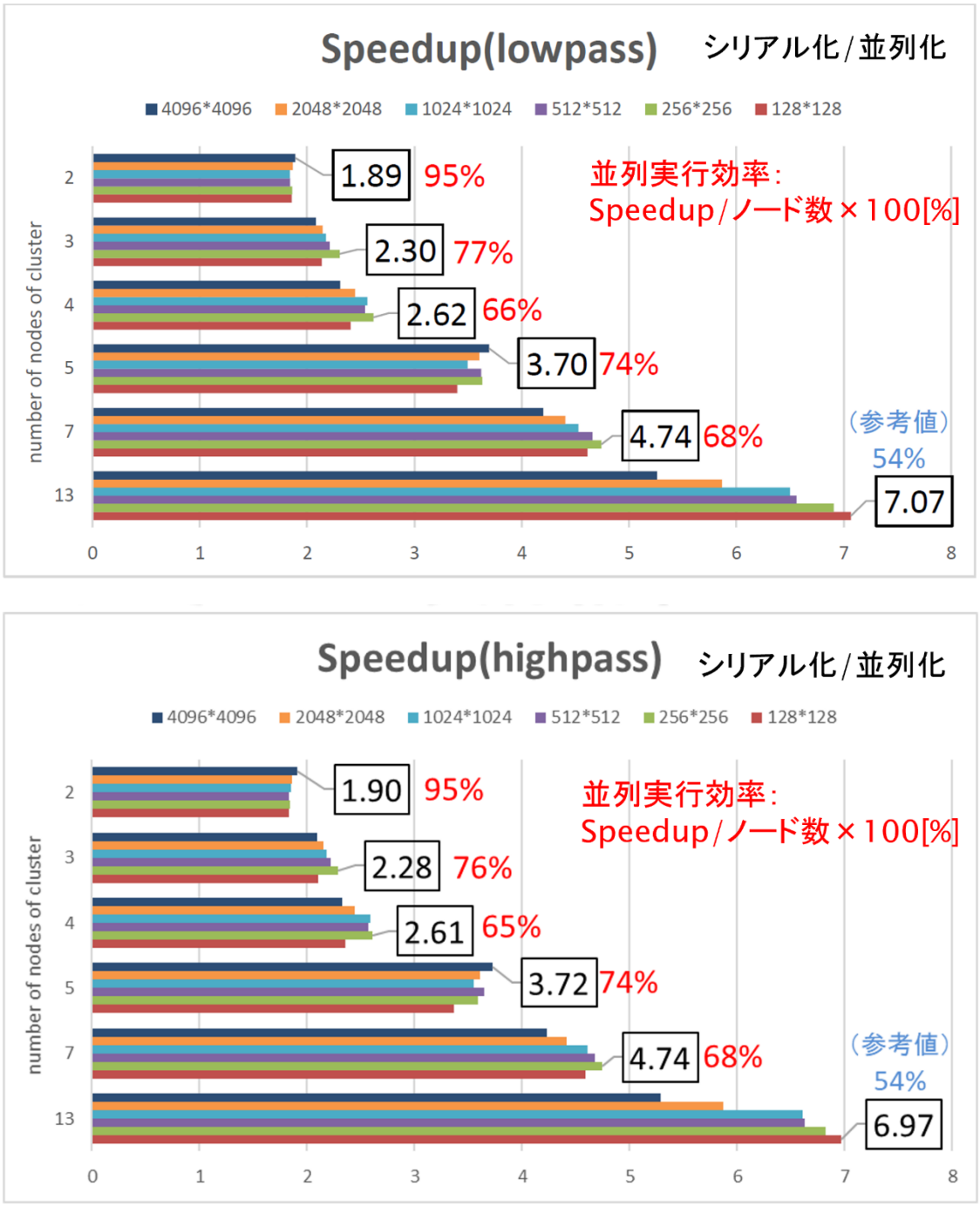


図 6 .3　FFT 例の実行結果（low-pass と high-pass）

ジから最終の画像出力まで測定した。OpenCL のランタイム設定やデータファイルの読み込み/書き込みの時間を含まないが、ノード間の MPI 通信に掛かる時間を含んでいる。入力イメージのサイズは、128^2 から 4096^2 までである。入力イメージの幅は、各カーネルプログラムのスレッド数として設定される。

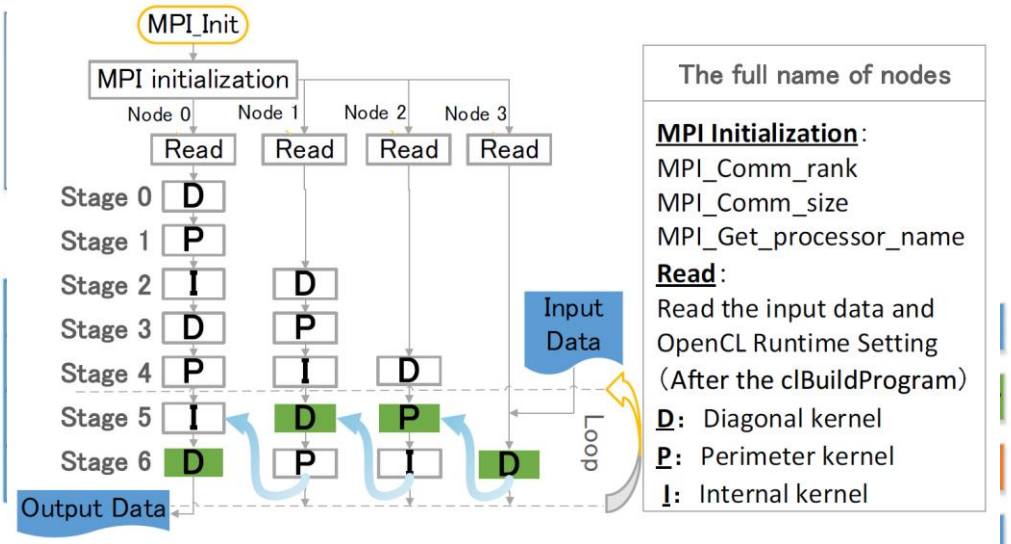


図 6.5　LU 分解例の並列パターン（consistency_shift = 2 の場合）

図 6.4　ブロックで実行される LU 分解の例

図 6.3 に、low-pass と high-pass による画像フーリエ変換の非同期通信でのスピードアップを示す。各グループの最大値を図中に数値で示し、並列実行効率もスピードアップの隣に表示している。並列実行効率はスピードアップとノード数の比を百分率で示している。13 ノードで実行する場合、1 つの GPU で 1 つのカーネルのみ実行するため、通信のオーバヘッドが極めて大きいため、実用的な実行パターンとは言えず、参考値として示している。したがって、FFT を用いた画像フーリエ変換の実験では、並列実行効率は 65%から 95%である。

通信オーバヘッドのため、各並列パターンのスピードアップは並列度より低いが、PEA-ST アルゴリズムを使用することにより、自動並列化で性能を効果的に改善できることがわかる。すなわち、プログラマが並列実行環境に合わせてカーネル間の実行順序を設計せずに、PEA-ST アルゴリズにより高速化が可能であることを示した。

6.1.2　LU 分解

(1)　アプリケーション概説

PEA-ST アルゴリズムの性能を評価するために、プログラマが書いた LU 分解と PEA-ST アルゴリズムにより並列化された処理フローを比較しながら解説する。図 6.4 の左側に LU 分解の例を示す。LU 分解とは、正方行列 A を下三角行列 L と上三角行列 U の積に分解することである。すなわち、A＝LU が成立する時に L 行列と U 行列を求めることを意味する。

LU 分解のアプリケーションは、Rodinia[81]の LUD ベンチマークプログラムを使用する。元の行列を幾つかのブロックに分けて計算する。図 6.4 の真中に処理フローの例を示して

いる。本アプリケーションは、3 種類のカーネルで構成される。まず、Diagonal では対角ブロック（DB）の分解を行う。次に、Perimeter では対角ブロックを除いた縦横ブロック（PB）の分解を行う。そして、Internal では残りの内部ブロックの分解を行う。ブロック数に応じて、LU 分解処理は 1 つのブロックのみ残されるまで一定回数繰り返し実行する。最後に残った対角カーネルに対し、Diagonal カーネルは一回実行する。

本アプリケーションでは、4 個と 16 個のブロックに分ける場合の実験を行った。図 6 .4 の右側に、それぞれの処理フローが示されている。4 個のブロックに分ける場合、処理フローは図 6 .4 の右上に示すように合計 7 個のカーネルで構成される。16 個のブロックに分ける場合、処理フローは図 6 .4 の右下に示すように合計 13 個のカーネルで構成される。

(2) PEA-ST による並列化

図 6 .5 では、PEA-ST アルゴリズムによって生成された処理パイプラインを示している。パイプライン方式では、フィードバックがないため、consistency shift が 1 である。4 個のブロックに分ける場合、処理フローの最大並列度は 7 になる。16 個のブロックに分ける場合、処理フローの最大並列度は 13 になる。ただし、必要な通信量も最大になるため、シリアル実行に対するスピードアップは最大並列度よりはるかに低くなる可能性がある。

また、前述したように、PEA-ST アルゴリズムによって生成された並列パターンは、複数の並列パターンから MAX_PARALLELISM パラメータによって決定される。ここでは、4 個のブロックに分ける場合、4 個の GPU を使用する際の並列化手順について説明する。例えば、ユーザ定義条件の MAX_PARALLELISM が 4 に従い、PEA-ST アルゴリズムにより生成された並列パターンは図 6 .5 に示した consistency shift = 2 の場合の結果である。まず、Spanning Tree が生成され、retreating edge がないため、consistency shift の最小値が 1 になる。

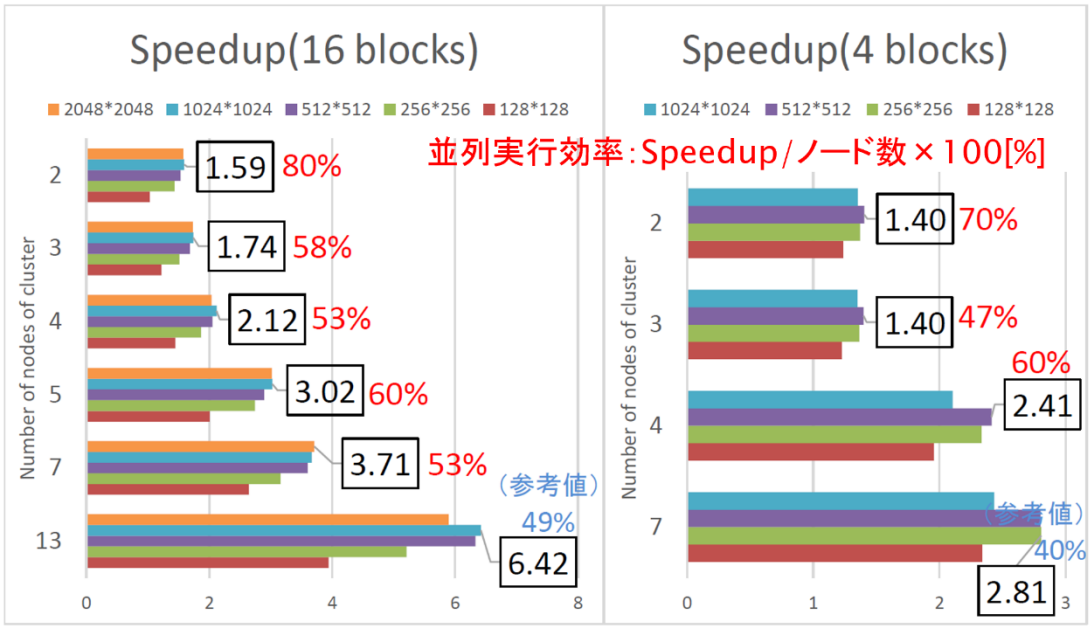


図 6 .6　LU 分解の例の実行結果

すなわち、パイプラインを作る際に間隔を 1 以上に空ける必要がある。次に、PEA-ST アルゴリズムは、シフト数を増加させることにより、適切な結果を見つける。execute matrix を生成するには、entry0 のように同じ深さのノードで各ステージを埋める。consistency shift が 2 である場合、entry1 のように entry0 を 2 行下にシフトする。同様に、entry2 と entry3 を生成し、execute matrix を完成する。ここで、consistency shift が 2 になると、並列度は 4 になり、ユーザ定義条件も満たすため、生成された並列パターンを結果として保存する。PEA-ST アルゴリズムにより生成された execute matrix は、7 つの stage と 4 つの entry を持つ。各 entry は、クラスタの各ノード上で実行される。最後に、execute matrix から startup と repeat batch を生成する。図 6 .5 に示すように、上の方が startup であり、下の 2 つの stage は repeat batch である。

また、図 6 .5 に青色の矢印で通信パターンも示している。各ノードに最後の stage（eq. stage 6）の実行が終われば、一時的な結果はノード N（N:1~3）からノード N-1 へ次の stage に送られる。受信が完了すると、ループは継続する。これらのノード間の通信に関しては、同期通信と非同期通信の両方で実験した。

(3)　実行結果

性能評価を行うために、シリアル化と並列化について最後の Diagonal までの実行時間を測定した。毎回の実行時間が短いため、どちらの場合も repeat batch のループを 100 回実行した。パイプライン化されたバージョンの実行時間は、図 6 .5 に示すように、Node 0 で最初のステージから最終のステージまで実行した場合の時間である。OpenCL のランタイム設

定やデータファイルの読み込み/書き込みの時間を含まないが、ノード間の MPI 通信に掛かる時間を含んでいる。入力データのサイズは、128^2 から 2048^2 までである。ブロックの幅はブロック数に合わせて設定している。

図 6.6 に 4 個と 16 個のブロックに分けた場合について、LU 分解の非同期通信でのスピードアップを示す。各グループの最大値を図中に数値で示し、並列実行効率もスピードアップの隣に表示している。並列実行効率は、スピードアップとノード数の比を百分率で示している。4 個のブロックに分けて 7 ノードで実行する場合と 16 個のブロックに分けて 13 ノードで実行する場合、1 つの GPU で 1 つのカーネルのみ実行するため、通信のオーバヘッドが極めて大きいため、実用的な実行パターンとは言えず、参考値として示している。したがって、LU 分解の実験では、4 個のブロックに分ける場合の並列実行効率は 47%から 70%、16 個のブロックに分ける場合の並列実行効率は 53%から 80%である。

通信オーバヘッドのため、各並列パターンのスピードアップは並列度より低いが、PEA-ST アルゴリズムを使用することにより、自動並列化で性能を効果的に改善できることがわかる。

６.２　PEA-ST アルゴリズムの性能最適化に関する評価

６.２.１　　通信パターン自動生成の最適化に対する評価

本節では、４.4 節で述べたフィードバックを有する通信パターンの最適化に対する評価について述べる、評価では、フィードバックを有する k-means アプリケーションを用いた。

(1)　　アプリケーション概説

フィードバックを有する処理フローとして、k-means クラスタリングアルゴリズムによるカラー画像量子化のアプリケーションを開発した。カラー画像量子化とは、画像のカラーパレットを固定数 k 種類の色に低減するタスクである。図 6.7 にカラー画像量子化の処理グローの例を示す。処理フローは 4 種類のカーネルで構成される。InitCenterGroup ではランダムに選択されたピクセルグループの中心の初期化を行う。SetGroup ではすべてのピクセルを k 個のグループに分割する。このカーネルにより RGB ピクセルから各中心までの距離を計算し、各ピクセルを最も近いピクセルグループに割り当てる。SetNewCenter では 新しい中心を計算する。そして、新しい中心の値を古い中心の値と比較する。SetGroup と SetNewCenter はそれらが同じ値に収束するまで繰り返し実行される。ChangeColor では画像の色を新しいものに変換する。

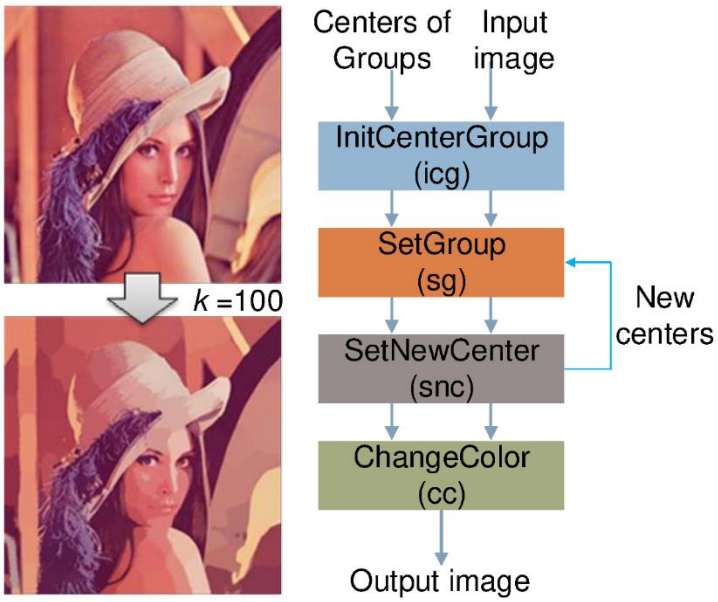


図 6.7　k-means アプリケーションの例

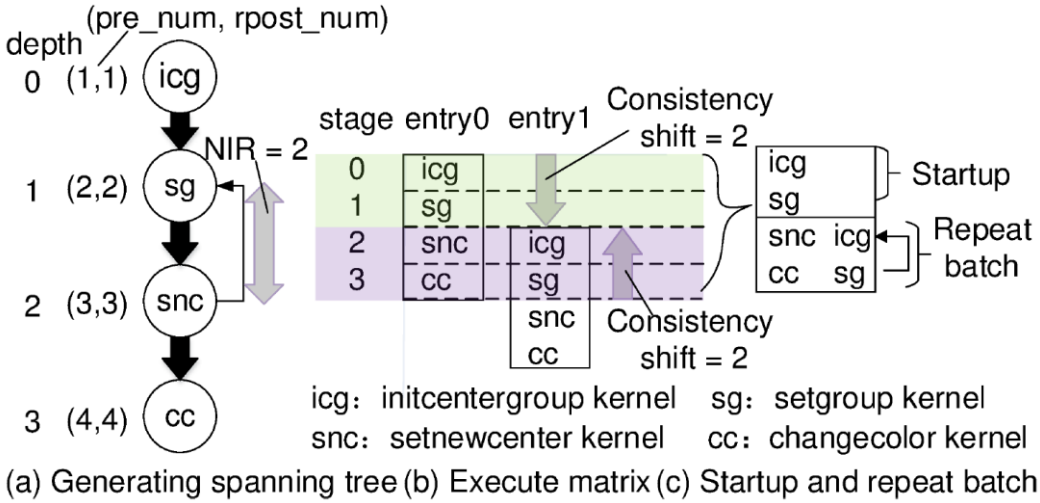


図 6.8　k-means の並列パターン

(2)　PEA-ST による並列化

図 6.8 に、PEA-ST アルゴリズムにより生成されたパイプライン処理フローを示す。この図を用いて、PEAST アルゴリズムによって k-means のパイプライン処理フローが生成される手順を詳細に説明する。ここで、図 6.7 の 4 種類のカーネル構造を図 6.8 のグラフに単純化する。まず、Spanning Tree が生成される。ルートになれるノードは icc ノードのみであるため、Spanning Tree は 1 つのみになる。生成された Spanning Tree から、retreating edge の snc → sg が存在し、NIR は 2 である。retreating edge が 1 つのみであるため、consistency shift は 2 になる。次に、PEA-ST アルゴリズムは consistency shift を用いて Execute Matrix を生成

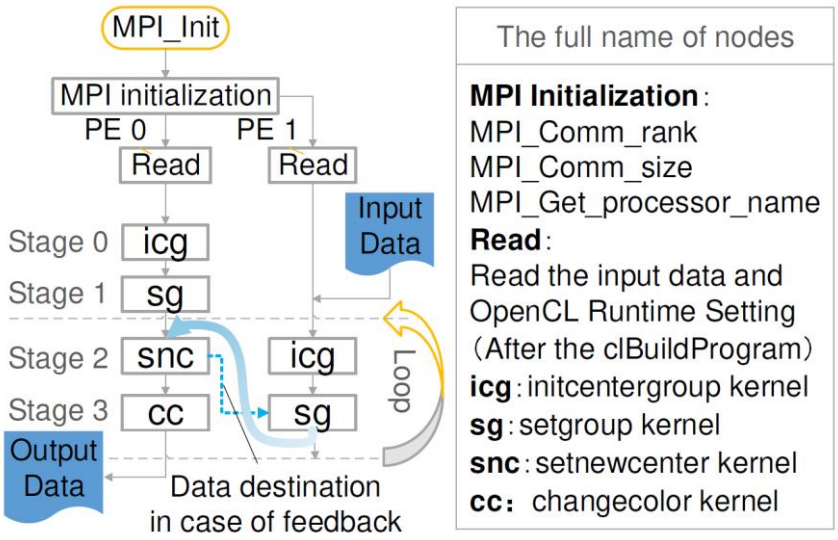


図 6 .9　k-means の例の通信パターン

する。図 6 .8(b)に示すように、4 つの stage と 2 つの entry が生成された。Startup と repeat batch を図 6 .8 (c)に示す。

(3)　通信パターンの最適化

k-means の通信パターンを含めたパイプラインの最終結果を図 6 .9 に示す。Repeat batch 部分では、入力データストリームはカーネル icc の入力として扱われる。次に、カーネル sg が呼び出される。青色の実線矢印は、PE1 から PE0 へのデータ転送を示す。カーネル snc が完了すると、実行がフィードバックに達する。カーネル snc の結果を古い中心と比較することにより、カーネル cc またはカーネル sg のどちらを実行するかを決定する。

図 6 .10 に k-means アプリケーションにおいて 3 つの通信パターンを示す。PEA-ST によって生成された k-means の通信パターンを左側に示す。しかし、k-means の処理フローは、4 .4 節に通信パターンの最適化のところで述べたケース 2 に属する。図 6 .10 の真中に示すように、カーネル sg が snc のデータのみ必要とし、カーネル icg はカーネル cc の後に実行しなければならない。すなわち、リソースがアイドルになるため、元の通信パターンでは並列性は得られない。そこで、図 6 .10 の右側のように PE0 の stage 3 にカーネル sg を追加し、各ステージの結果を PE 間で交換する。ループが PE0 の stage n-1 で終了する場合、カーネル cc を実行する。PE1 上のカーネル snc の結果が PE0 に送られ、カーネル icg が再び呼び出される。ループが PE1 の stage n-1 で終了する場合、PE0 でのカーネル snc の結果は一時的に記憶され、stage n+1 に渡される。そして、カーネル cc の実行と同時に、カーネル icg は PE1 の stage n で再び実行され、パイプライン実行も継続できる。このような最適化により、k-means の並列度は 2 まで向上できる。

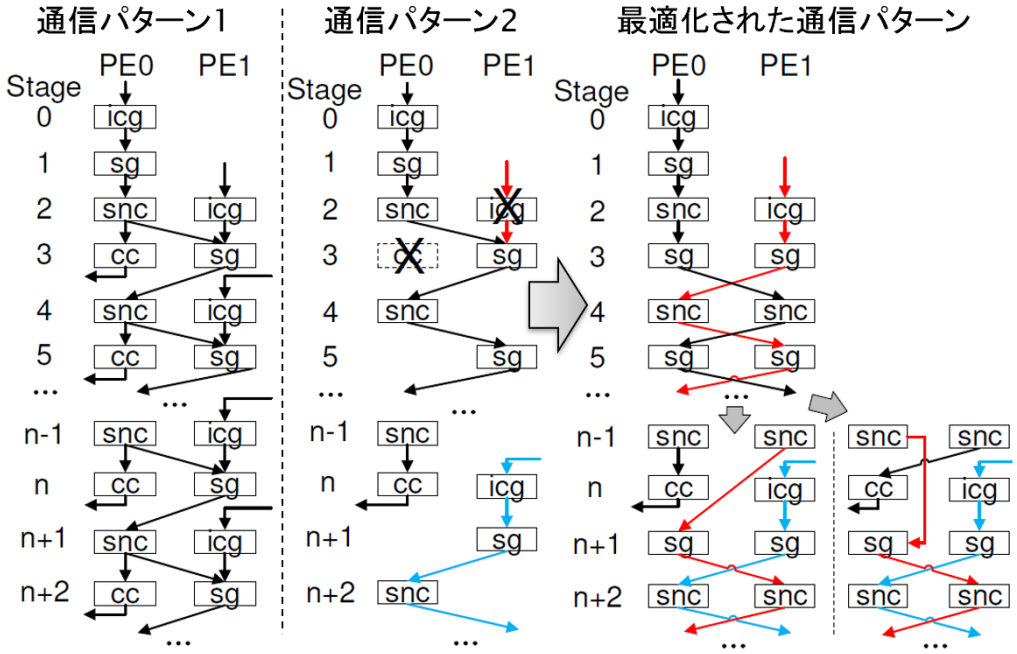


図 6.10　k-means の通信パターン最適化の例

表 6.1　k-means アプリケーションの実行結果

Image size		128^2	256^2	512^2	1024^2	2048^2
serialized execution time(sec)		31.90	118.27	519.91	1559.00	10394.84
parallelized execution time(sec)	Asynchron-ous mode	21.19	74.39	347.95	1067.80	7027.13
speedup		1.51	1.59	1.49	1.46	1.48

(4)　実行結果

性能評価を行うために、シリアル化と並列化について最後の ChangeColor までの実行時間を測定した。毎回の実行時間が短いため、どちらの場合も repeat batch のループは 100 回実行した。パイプライン化されたバージョンの実行時間は、node 0 で最初のステージから最終のステージまで実行した場合の時間である。OpenCL のランタイム設定やデータファイルの読み込み/書き込みの時間を含まないが、ノード間の MPI 通信に掛かる時間を含んでいる。入力データのサイズは、128^2 から 2048^2 までである。

表 6.1 に k-means アプリケーションの実行結果を示す。非同期通信での実行時間とスピードアップを示す。並列度が 2 で、2 ノードを使用した場合、1.5 倍のスピードアップが得られた。並列実行効率は約 75%である。通信オーバヘッドのため、スピードアップは 2 に満たないが、通信パターンの最適化により、フィードバックを有する処理フローに対しても効率良い通信パターンを生成できることを確認した。

表 6.2　各カーネルの実行時間

Band-pass filter example (Image size:4096^2, highpass filter)				LU decomposition example (Matrix size:1024^2, with 16 blocks)			
No.	Name	Time(sec)	Combination	No.	Name	Time(ms)	Combination
1	Reorder	0.3572	1.6784	1	diagonal	76.26	163.25
2	FFT	1.1234		2	perimeter	86.99	
3	Transpose	0.1979		3	internal	35.34	196.25
4	Reorder	0.3288	1.6413	4	diagonal	73.68	
5	FFT	1.1148		5	perimeter	87.23	
6	Transpose	0.1977		6	internal	23.26	183.85
7	highpass	0.1268	1.7680	7	diagonal	73.58	
8	Trans	0.1977		8	perimeter	87.01	
9	Reorder	0.3288		9	internal	13.21	167.20
10	IFFT	1.1147		10	diagonal	73.53	
11	Transpose	0.1978	1.6411	11	perimeter	3.55	
12	Reorder	0.3287		12	internal	3.55	
13	IFFT	1.1147		13	diagonal	73.37	
Total		6.7294		Total		710.55	

6.2.2　ロードバランスによる最適化に対する評価

(1)　各カーネルの実行時間と最適化方法

4.5 節で述べたように、実際に各カーネルの実行時間は異なるため、ロードバランスを考慮した最適化を行った。前述の k-means アプリケーションでは、stage 3 以降に各 PE で同じカーネルが実行するため、ロードは均衡している。一方、6.1 節で実験した画像フーリエ変換と LU 分解の例では、ロードが不均衡になっていると考えられる。そこで、各カーネルの実行時間を測定し、ロードの均衡化を図った。画像フーリエ変換の例として high-pass フィルタを用いた実行時間を示す。 Low-pass フィルタの状況はほぼ同じであるため省略する。

表 6.2 では、セルの色が濃いほど、実行時間が長いことを表している。表より、カーネルの実行時間が互いに大きな違いがあることがわかる。また、並列化バージョンにおいて、ロードバランスしていないことが確認できる。例えば、画像フーリエ変換の例では、最も実行時間の長いカーネルは FFT と IFFT である。前述の実験では、2 つの IFFT カーネルが node0 で、2 つの FFT カーネルが node2 でそれぞれ実行された。本実験では、ロードバランスをとるために、表 6.2 に示すようにカーネルを 4 つのグループに分けた。すなわち、1~3/4~6/7~10/11~13 と 1~2/3~5/6~8/9~13 とした。

図 6.11(a)にある処理フローは FFT を用いた画像フーリエ変換の例である。ロードバランスを考慮した並列パターンは図 6.11(b)のようになる。同様に、 LU 分解のアプリケーショ

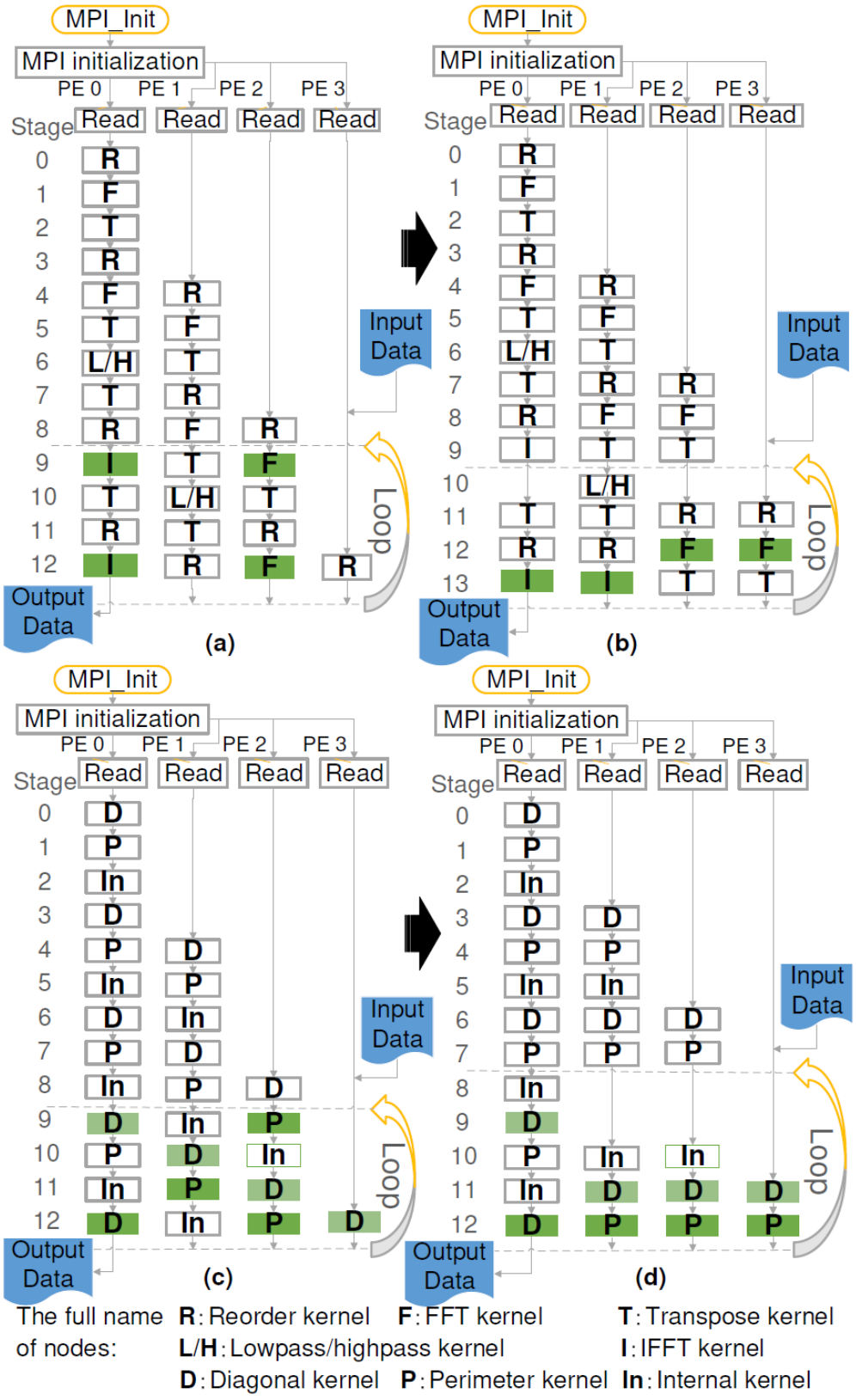


図 6 .11　(a)(c)ロードバランスを考慮しない場合と(b)(d)ロードバランスを考慮した場合の並列パターンの比較

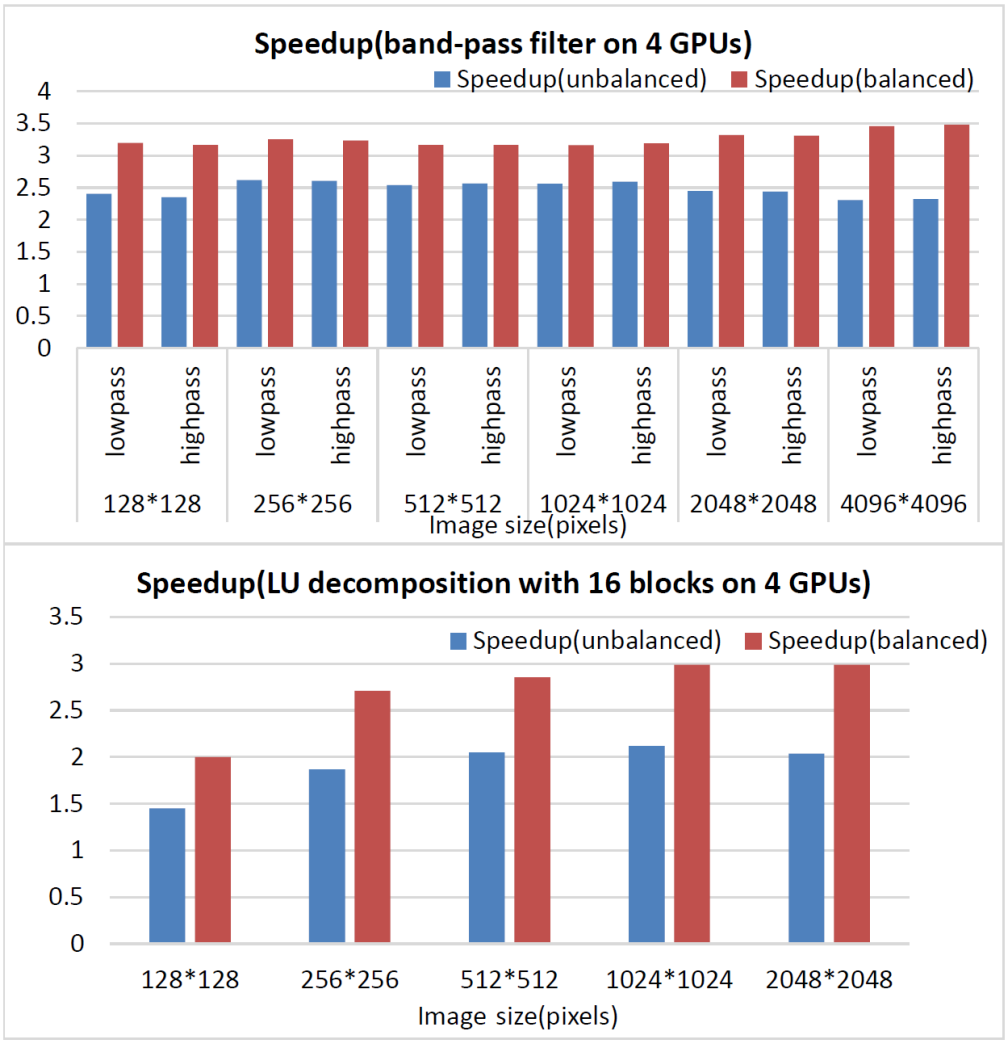


図 6 .12　4 個の GPU でのロードバランスを考慮しない場合と
考慮した場合のスピードアップの比較

ンに対して、カーネルを 4 つのグループに分け、並列パターンを図 6 .11(d)に示すように変更した。

(2)　結果

図 6 .12 に、4 つの GPU で実行した場合の、ロードバランスを考慮した場合としない場合のスピードアップを示す。図の上部に bandpass フィルタの結果を示す。

この図から、bandpass フィルタの例では、特にデータサイズが大きい場合、スピードアップは 2.3 から 3.5 に向上した。また、16 個のブロックに分けて実行された LU 分解の例では、スピードアップは 2 から 3 に向上した。これらの実験結果より、PEA-ST アルゴリズムから抽出された並列パターンに基づいて、ロードバランスを考慮することにより、さらなる性能の向上が得られることを確認した。

６.３　まとめ

本章では、ストリーム指向プログラムのための並列性抽出アルゴリズム PEA-ST の性能を評価した。

６.１節では、PEA-ST アルゴリズムの有効性を評価するために、画像処理と数値解析に関連する FFT を用いた画像フーリエ変換と LU 分解のアプリケーションを用いて本アルゴリズムの性能評価を行った。FFT を用いた画像フーリエ変換の実験では、65%から 95%の並列実行効率が得られた。LU 分解の実験では、4 個のブロックに分ける場合 47%から 70%の並列実行効率、16 個のブロックに分ける場合 53%から 80%の並列実行効率が得られた。通信オーバヘッドのため、各アプリケーションにおいて並列パターンのスピードアップは並列度より低いが、PEA-ST アルゴリズムを使用することにより、性能を効果的に改善できることを示した。プログラマが並列実行環境に合わせてカーネル間の実行順序を設計する必要がなくなり、PEA-ST アルゴリズにより一定の並列効果が得られることを確認した。

６.２節では、PEA-ST アルゴリズムの性能最適化を評価するために、k-means と前述の 2 つのアプリケーションを用いて性能評価を行った。k-means アプリケーションを用いた評価ではフィードバックを有する通信パターンの最適化に用いて評価した。その結果、並列度が 2 で、2 ノードを使用し、1.5 倍のスピードアップが得られた。すなわち、約 75％の並列実行効率が得られた。また、FFT フーリエ変換と LU 分解の結果を用いてロードバランスによる最適化に関して評価した。画像フーリエ変換の例では、スピードアップが 2.3 から 3.5 に向上した。16 個のブロックに分けて実行された LU 分解の例では、スピードアップが 2 から 3 に向上した。評価の結果、PEA-ST アルゴリズムから抽出された並列パターンに基づいて、ロードバランスを考慮することにより、さらなる性能の向上が得られることが確認できた。

第 7 章
結論

本研究では、ストリーム指向プログラミングに関して、アクセラレータのための高性能なアプリケーション開発の負担を軽減することを目指し、プログラマが下層ハードウェアの構成に関する知識を持たなくても、カーネル間の並列性を自動的に引き出して、アクセラレータの高い計算性能を活用できる開発環境を実現することを目的とする。本論文では、複数のカーネルで構成される静的な処理フローから、並列性を抽出し、処理順序を決定する Parallelism Extraction Algorithm with Spanning Tree(PEA-ST) と呼ぶ新たな並列性抽出アルゴリズムを提案した。本アルゴリズムは、Spanning Tree を使用することで、フィードバックやフィードフォワードがある場合を含め、処理フローから空間的な並列性と時間的な並列性を抽出可能とするものである。

実行順序と並列性を見つけるために、本アルゴリズムには三つの手順があることを述べた。1）最初に実行可能なノードを見つける。次に、2）I/O バッファが競合しない実行順序を発見する。最後に、3）処理フローから並列性を抽出する。また、実際の並列環境に合わせて、適切な並列パターンと通信パターンを自動的に生成する。4）ユーザ定義条件をチェックし、適切な並列パターンを選択すること、5）データストリームの流れにより通信パターンを生成することを本アルゴリズムの手順として追加した。本アルゴリズムの実装も行った。1）最初に実行できるノードの確定、2）Spanning Tree の生成、3）consistency shift の演算、と execute matrix の作成、4）startup と batch の作成、5)ユーザ定義の要求に応じて結果の更新、6）通信パターンの生成という六つの機能を実現した。さらに、フィードバックに対する通信パターンの最適化と、ロードバランスによる最適化を行った。

本アルゴリズムの有効性と性能最適化について検証するため、アプリケーションを用いた性能評価を行った。画像処理と数値解析に関連する FFT を用いた画像フーリエ変換と LU 分解のアプリケーションを用いて本アルゴリズムの有効性に関する性能評価を行った。FFT を用いた画像フーリエ変換の実験では、65%から 95%の並列実行効率が得られた。LU 分解の実験では、4 個のブロックに分ける場合 47%から 70%の並列実行効率、16 個のブロックに分ける場合 53%から 80%の並列実行効率が得られた。プログラマが並列実行環境に合わせてカーネル間の実行順序を設計する必要がなくなり、PEA-ST アルゴリズムにより高い並列効率が得られることを確認した。

また、k-means と前述の 2 つのアプリケーションを用いて本アルゴリズムの性能最適化に関する性能評価を行った。前者のアプリケーションを用いてフィードバックを有する処理

フローに対する通信パターン生成の最適化に関して評価した。並列度が 2 で、2 ノードを使用し、約 75%の並列実行効率が得られた。これにより、フィードバックを有する処理フローに対しても高い並列性の抽出が可能であることを明らかにした。また、FFT フーリエ変換と LU 分解の結果を用いてロードバランスによる最適化に関する評価を行った。画像フーリエ変換と LU 分解のアプリケーションでは、計算ノード間のロードバランスを考慮した並列化を行った場合、4 個の計算ノードを用いて 3 倍以上のスピードアップが得られることを確認した。評価の結果、PEA-ST アルゴリズムから抽出された並列パターンに基づいて、ロードバランスを考慮することにより、さらなる性能の向上が得られることを確認した。

今後は、PEA-ST をロードバランスも考慮したアルゴリズムに拡張する予定である。また、処理フローのノードを stage 間に上下方向で移動させることで、Spanning Tree の等価変形による本アルゴリズムのさらなる最適化も 1 つの課題である。

謝辞

本論文を執筆するにあたり、丁寧で熱心なご指導をして頂きました筑波大学システム情報工学研究科教授・和田耕一先生に心より感謝を申し上げます。日本に来て研究生になってから現在に至るまで 7 年間という長い期間、快適な研究環境を提供して下さり、国際学会や論文誌において発表することができました。さらに、研究だけでなく、日常生活まで多くのご指導、ご助力を頂きました。ここにて深く御礼申し上げます。また日頃から研究に関して熱心にご指導をいただいた筑波大学システム情報工学研究科准教授・山際伸一先生に心より感謝を申し上げます。

また、お忙しい中、本論文の副査を引き受けていただいたきました筑波大学システム情報工学研究科准教授・阿部洋丈先生、筑波大学システム情報工学研究科教授・安永守利先生、筑波大学システム情報工学研究科教授・丸山勉先生に深く感謝いたします。

快適で楽しい研究生活を送らせていただきました並列分散処理研究室の皆様に深く感謝いたします。研究の相談・論文の校正などでお世話になりました。心より感謝申し上げます。

最後に 7 年間に渡る大学生活を生活面で支えてくださった家族に心から感謝いたします。

参考文献

[1] K.-L.Wu, K.W. Hildrum,W. Fan, P. S. Yu, C. C. Aggarwal, D. A. George, B. Gedik, E. Bouillet, X. Gu, G. Luo, and H. Wang, “Challenges and Experience in Prototyping a Multi-modal Stream Analytic and Monitoring Application on System S,” in Proceedings of the 33rd International Conference on Very Large Data Bases, ser. VLDB ’07. VLDB Endowment, pp. 1185–1196, 2007.

[2] A. Savadi and R. Yanamshetti, "A survey on design of digital signal processor," 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, pp. 2483-2486, 2016.

[3] Brown, Stephen D., et al. Field-programmable gate arrays. vol. 180. Springer Science & Business Media, 2012.

[4] Cg language, http://developer.download.nvidia.com/cg/Cg_language.html.

[5] BrookGPU, http://graphics.stanford.edu/projects/brookgpu/.

[6] NVIDIA Corporation, “CUDA: Compute Unified Device Architecture programming guide.” http://developer.nvidia.com/cuda

[7] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, OpenCL Programming Guide. Addison Wesley, 2011.

[8] S. Yamagiwa and L. Sousa, “Caravela: A Novel Stream-Based Distributed Computing Environment,” IEEE Computer, vol. 40, no. 5, pp. 70–77, 2007.

[9] S. Yamagiwa and S. Zhang, “CarSh: A Commandline Execution Support for Stream-based Acceleration Environment,” in Procedia Computer Science, Proceedings of the International Conference on Computational Science, ICCS 2013. ELSEVIER, 2013.

[10] S. Yamagiwa and L. Sousa, “Modelling and Programming Stream-based Distributed Computing based on the Meta-pipeline Approach,” Int. J. Parallel Emerg. Distrib. Syst., vol. 24, no. 4, pp. 311–330, 2009.

[11] ISO/TC 159/SC 4, ISO 9241-11:2018: Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts Ed. 2, 2018.

[12] Nielsen, Jakob. Usability engineering. Elsevier, 1994.

[13] Charles Antony Richard Hoare. The emperor's old clothes, Commun ACM, vol. 24, no. 2, pp. 75-83, 1981.

[14] Thomas J. Bergin, Jr. and Richard G. Gibson, History of Programming languages---II. ACM, New York, USA, 1996.

[15] John Backus, The history of FORTRAN I, II, and III, SIGPLAN Not, ACM, vol. 13, no. 8, pp. 165-180. 1978.

[16] E. W. Dijkstra, “Structured Programming”, In Software Engineering Techniques, B. Randell and J.N. Buxton, (Eds.), NATO Scientific Affairs Division, Brussels, Belgium, pp. 84–88, 1970.

[17] P. J. Landin, The next 700 programming languages, Commun. ACM, vol. 9, no. 3, pp. 157-166, 1966.

[18] Goldberg, A., & Robson, D. Smalltalk-80: The Language and Its Implementation, 1983.

[19] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar, Advances in dataflow programming languages, ACM Comput. Surv., vol. 36, no. 1, 1-34, 2004.

[20] Nell Dale, John Lewis, The History of Computing of Computer Science Illuminated, Jones & Bartlett Publishers, 2015.

[21] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger. Revised report on the algorithm language ALGOL 60. Commun. ACM, vol. 6, no. 1, pp. 1-17, 1963.

[22] John G. Kemeny, Thomas E. Kurtz, Back to BASIC: the history, corruption, and future of the language, Addison-Wesley Pub. Co., 1985

[23] Nell B. Dale, Chip Weems, Introduction to Pascal and Structured Design, Jones & Bartlett Learning, 1996.

[24] Dennis M. Ritchie. 1993. The development of the C language. SIGPLAN Not. vol. 28, no. 3, pp. 201-208, 1993.

[25] Corrado Böhm and Giuseppe Jacopini, Flow diagrams, turing machines and languages with only two formation rules. Commun. ACM, vol. 9, no. 5, pp. 366-371, 1996.

[26] J. Hughes, Why functional programming matters, The computer journal, vol. 32, no. 2, pp. 98-107, 1989.

[27] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I. Commun. ACM, vol. 3, no. 4, pp. 184-195, 1960.

[28] Haskell: An advanced, purely functional programming language, https://www.haskell.org/.

[29] The Scala Programming Language, https://www.scala-lang.org/.

[30] The Clojure Programming Language, https://clojure.org/.

[31] F#: a mature, open source, cross-platform, functional-first programming language, https://fsharp.org/.

[32] Bjarne Stroustrup, The C++ Programming Language (4th Edition), Addison-Wesley Professional, 2013.

[33] Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, and Sharon Biocca Zakhour, The Java Tutorial: A Short Course on the Basics (6th Edition). Addison-Wesley Professional, 2014.

[34] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. Commun. ACM, vol. 52, no. 11, pp. 60-67. 2009.

[35] Tiago Boldt Sousa, Dataflow programming concept, languages and applications, Doctoral Symposium on Informatics Engineering, vol. 130, 2012.

[36] P. T. Cox, F. R. Giles and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in Proceedings of 1989 IEEE Workshop on Visual Languages, Italy, pp. 150-156, 1989.

[37] LabVIEW, http://www.ni.com/ja-jp/shop/labview.html.

[38] P. Pacheco, An Introduction to Parallel Programming, Morgan Kaufman, 2011.

[39] A. Vajda, “Multi-core and Many-core Processor Architectures,” in Programming Many-Core Chips.

Springer US, pp. 9–43, 2011.

[40] K. Diefendorff, “Power4 focuses on Memory Bandwidth,” Microprocessor Report, vol. 13, no. 13, pp. 1–8, 1999.

[41] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell Broadband Engine Architecture and its First Implementation-A Performance View,” IBM Journal of Research and Development, vol. 51, no. 5, pp. 559–572, 2007.

[42] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, “Power7: IBM’s Next-generation Server Processor,” IEEE micro, vol. 30, no. 2, pp. 7–15, 2010.

[43] M. Mantor, “AMD HD7970 Graphics Core Next (GCN) Architecture,” in Proceedings of the Hot Chips, pp. 248–282, 2012.

[44] S. Nussbaum, “AMD Trinity Fusion APU,” in Proceedings of the Hot Chips, pp. 283–322, 2012.

[45] G. Chrysos, “Knights Corner, Intel ’s first Many Integrated Core (MIC) Architecture Product,” in Proceedings of the Hot Chips, pp. 323–353, 2012.

[46] T. P. Chen and Y.-K. Chen, “Challenges and Opportunities of Obtaining Performance from Multi-core CPUs and Many-core GPUs,” in Acoustics, Speech and Signal Processing, ICASSP 2009. IEEE International Conference on. IEEE, pp. 613–616, 2009.

[47] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, “Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends,” in Proceedings of the 50th Annual Design Automation Conference, ser. DAC ’13, ACM, New York, USA, pp. 1–10, 2013.

[48] Scott, Noel D., Daniel M. Olsen, and Ethan W. Gannett. "An overview of the visualize fx graphics accelerator hardware." HEWLETT PACKARD JOURNAL, vol. 49, pp. 28-29, 1998.

[49] Altera, OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity, http://www.altera.com/products/software/opencl/opencl-index.html.

[50] H. Nguyen, GPU Gems 3, 1st ed, Addison-Wesley Professional, 2007.

[51] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, “GPGPU: General-purpose Computation on Graphics Hardware,” in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, ser. SC ’06. New York, NY, USA: ACM, 2006.

[52] OpenMP, https://www.openmp.org/.

[53] OpenACC, https://www.openacc.org/.

[54] H. Richardson, High Performance Fortran: history, overview and current developments, Tech. Rep. TMC-261, Thinking Machines Corporation, 1996.

[55] B. Babcock, S. Babu, M. Datar, R. Motwani, and J.Widom, “Models and Issues in Data Stream Systems,” in Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ser. PODS ’02. New York, NY, USA: ACM, pp. 1–16, 2002.

[56] J. Gummaraju and M. Rosenblum, “Stream Programming on General-Purpose Processors,” in Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, pp. 343–354, 2005.

[57] Jon Stokes, Peakstream unveils multicore and cpu/gpu programming solution, 2006.

[58] StreamPipes, https://www.streampipes.org/en/, 2018.

[59] Radia Perlman, An Algorithm for Distributed Computation of a Spanningtree in an Extended LAN, ACM SIGCOMM Computer Communication Review, vol. 15, no. 4, pp. 44-53, 1985.

[60] Michael Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.

[61] Vector Fabrics. Pareon, 2012.

[62] ParaWise Automatic Parallelisation of C and Fortran, http://www.parallelsp.com/index.html.

[63] S. Breuer, M. Steuwer, and S. Gorlatch: Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. In Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils), Vienna, Austria, 2014.

[64] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11). ACM, pp. 47-56, 2011.

[65] U. C, R. Nasre, and Y. N. Srikant, “Falcon: A Graph Manipulation Language for Heterogeneous Systems,” ACM Trans. Archit. Code Optim., vol. 12, no. 4, pp. 54:1–54:27, 2015.

[66] Amini, Mehdi, et al. "Par4all: From convex array regions to heterogeneous computing", in IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012.

[67] C. Nugteren and H. Corporaal, “Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs,” ACM Trans. Archit. Code Optim., vol. 11, no. 4, pp. 35:1–35:25, 2014.

[68] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. Int. J. Parallel Program. 46, 1 (February 2018), 62-80.

[69] StarPU, http://starpu.gforge.inria.fr/.

[70] A. Haidar, Y. Jia, P. Luszczek, and orthers, “Weighted Dynamic Scheduling with Many Parallelism Grains for Offloading of Numerical Workloads to Multiple Varied Accelerators,” in Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. ACM, pp. 5:1–5:8, 2015.

[71] StreamIt, http://groups.csail.mit.edu/cag/streamit/.

[72] R. Kumar Thakur and Y. N. Srikant, “Efficient Compilation of Stream Programs for Heterogeneous Architectures: A Model-Checking Based Approach,” in Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems. ACM, pp. 38–47, 2015.

[73] H. P. Huynh, A. Hagiescu, W.-F. Wong, and orthers, “Scalable Framework for Mapping Streaming Applications Onto multi-GPU Systems,” SIGPLAN Not., vol. 47, no. 8, pp. 1–10, 2012.

[74] B. R. Rau and C. D. Glaeser, “Some Scheduling Techniques and An Easily Schedulable Horizontal Architecture for High Performance Scientific Computing,” in ACM SIGMICRO Newsletter, vol. 12, no. 4, pp. 183–198, 1981.

[75] M. S. Lam, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” SIGPLAN Not., vol. 39, no. 4, pp. 244–256, April 2004.

[76] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, “Automatic Thread Extraction with Decoupled Software Pipelining,” in Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, pp. 105–118, 2005.

[77] Vitus J. Leung, David P. Bunde, Jonathan Ebbers, Stefan P. Feer, Nickolas W.Price, Zachary D. Rhodes

and Matthew Swank: Task mapping stencil computations for non-contiguous allocations, In Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 377-378, 2014.

[78] Andreas Diavastos and Pedro Trancoso: Auto-tuning Static Schedules for Task Data-flow Applications, In Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, Article 1, pp. 1-6, 2017.

[79] Junhua Fang, Rong Zhang, Tom Z.J. Fu, Zhenjie Zhang, Aoying Zhou and Junhua Zhu: Parallel Stream Processing Against Workload Skewness and Variance, In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 15-26, 2017.

[80] Yuri Alexeev, Ashutosh Mahajan, Sven Leyffer, Graham Fletcher and Dmitri G. Fedorov: Heuristic static load-balancing algorithm applied to the fragment molecular orbital method, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article 56, pp. 1-13, 2012.

[81] Rodinia: Accelerating Compute-Intensive Applications with Accelerators. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php.

付録 A　公表論文リスト

- 査読付き雑誌論文

1. Shinichi Yamagiwa, Guyue Wang, and Koichi Wada, Development of an Algorithm for Extracting Parallelism and Pipeline Structure from Stream-based Processing flow with Spanning Tree. International Journal of Networking and Computing, Vol. 5, No. 1, pp.159-179, 2015.
2. Guyue Wang, Koichi Wada, and Shinichi Yamagiwa, Optimization in Parallelism Extraction Algorithm with Spanning Tree on a Multi-GPU Environment, IEEJ Transactions on Electrical and Electronic Engineering, Vol. 14, No. 6, 2019(to appear).

- 査読付き国際会議論文

1. Guyue Wang, Shinichi Yamagiwa, and Koichi Wada, Parallelism Extraction Algorithm from Stream-based Processing Flow applying Spanning Tree, In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, APDCM14, IEEE CS, pp.632-641, 2014.
2. Guyue Wang, Koichi Wada, and Shinichi Yamagiwa, Performance Evaluation of Parallelizing Algorithm Using Spanning Tree for Stream-Based Computing, In Computing and Networking (CANDAR), 2016 Fourth International Symposium on Computing and Networking, IEEE, pp. 497-503, 2016.