１９９５年度

修士論文

論題：並列プログラミング用事例ベースの作成と類似事例の検索・修正法

指導教員	山崎 勝弘 教授

立命館大学大学院理工学研究科

修士課程 情報システム学 専攻 No. 6114940078-7

氏名	松田 浩一

目次

図目次

表目次

あらまし

本論文では, 過去の類似した並列プログラムを極力再利用して, 並列プログラムの負担を軽減させる方法について述べる.

並列処理は, 大規模な数値演算からオンライントランザクション処理やデータベース管理/検索まで広い範囲のアプリケーションに利用されている. しかし, タスク分割, 同期, 通信, スレッドなどを扱う困難さが並列プログラミング特有の問題として存在し, 並列処理本来の能力を発揮しているとは言い難い.

本研究では事例ベース推論の考え方を用いた並列プログラミング支援システムについて検討する. 本手法では並列プログラムの骨格を取り出し, それをスケレトンとして事例ベースに保存する. 事例には, インデックス, プログラム本体, 並列効果, 及び履歴がスケレトンと共に含まれている. 新たな並列プログラム作成の際, 問題の特徴抽出を項目毎に行い, インデックスを作成する. システムはインデックスを基に事例ベースの中から最も類似する事例を検索する. 検索された事例に含まれるスケレトンに肉付け・修正を施すことにより目的の並列プログラムを完成させる. このような方法により並列プログラミングの負担を軽減し, 正確で有効な並列プログラムの作成を目指す.

並列アルゴリズムは一般的に分割統治法, プロセッサファーム, プロセスネットワーク, 繰り返し変換に分類でき, 同一クラス内では共通した並列構造を持つプログラムが作成できる. 本研究は昨年に引き続き, 各アルゴリズム毎に代表的な並列プログラムを作成し, その中から並列構造, スケレトンを抽出し, 並列プログラミング用事例ベースの作成を行った. また, 事例検索のための問題解析法, 及び類似事例の修正法について検討した.

1 はじめに

近年, 従来のベクトルプロセッサ方式の多数のスーパーコンピュータが超並列マシンへ移行され, 大規模な数値演算からオンライントランザクション処理やデータベース管理/検索まで広い範囲のアプリケーションに利用されている. 並列処理は, 今後あらゆる情報処理を支える基盤的技術となることが期待されている. しかし, 現状では並列処理本来の能力を発揮しているとは言い難い. その理由の一つとして並列プログラミングの難しさ, 並列プログラミング環境の未熟さがあげられる. 並列処理では, ソフトウェアによる負荷均衡の実現とプロセッサ間の情報授受のオーバヘッドの軽減が大変重要である. また, プログラム中の同期, 通信などのエラーは並列プログラム特有の問題として存在し, そのデバッギングに非常に多くの時間と労力が費やされている. 共有メモリ方式の並列マシンでは, 単一メモリを複数のプロセッサで共有することにより, プロセッサ間通信の概念が無く, 逐次プログラムと似た比較的簡単なプログラムを書くことができる. しかし, プロセス間の同期, 相互排除を陽に示さなければならないという新たな問題が生じる. いずれにしても並列処理の効果を高めるためには, 対象マシンのアーキテクチャを熟知し, 並列プログラミングの十分な経験が必要である.

ようやく解決できた問題であっても, 知識の蓄積がなければ他のプログラマが同じ問題に出会った時に再び同様の時間と労力を必要とする. これは非常に非効率的な作業であり, 非生産的である. そこで過去の経験の再利用を図るため事例ベース推論 (Case-Based Reasoning: CBR)[1] を用いる. 事例ベース推論は記憶と想起に関する認知科学的研究に由来し, 問題解決に過去の体験を有効に活用するために提案された. いかなる問題解決においても, 専門的な知識が再利用可能な形で保持されていなければ, 後続の開発者は全く同様の苦労をしなければならない. それを避けるために知識を事例として蓄積し, 再利用することが考えられ, すでに裁判の判決, 故障診断, 各種設計支援 [2][3] などのシステムが構築されている.

本システムでは, 過去の並列プログラムを再利用可能な形で事例として事例ベースに保存しておく. 新たな問題に対しては, その問題に最も類似した事例を事例ベースの中から検索する. 検索事例に修正・肉付けを行うことにより新たな並列プログラムを完成させる. このような方法により, 並列プログラミングの負担を軽減し, 正確で有効な並列プログラムの作成を行うことを目的としている.

事例はインデックス, スケレトン, プログラム本体, 並列効果, 履歴から成る. プログラムをそのままの形で再利用するのは困難であるため, 本手法では並列プログラムの骨格部分をスケレトンとして用意する. スケレトンにはタスク分割, 相互排除, 同期の取り方, スレッド操作法などプログラミングプロセスにおける最も重要な部分が記述されている. これに, 問題に応じて計算部分, 計算範囲の指定, 入出力部分などを組み込むことにより新たな並列プログラムを完成させる. また, 類似したスケレトンを検索するために, その問題の特徴をインデックス付けする. インデックスには応用, 仕様, アルゴリズム, 終了条件, データ構造, タスク分割法, トポロジ, インタラクション, 並列化手法, 実行構造が含まれる. 並列効果は, 並列処理を行った結果, どの程度の速度向上が得られたかをグラフで示したものである. 履歴には, 事例の参照回数や類似事例として再利用された時の利用状況が記録されている.

本研究は並列アルゴリズムの一般性に重点を置いている. 並列プログラムは問題に応じて様々

なものが存在するが, プロセスの生成や消去, 同期, 相互排除などのプロセス操作に関わる部分, 及びタスク分割法などはいくつかのパターンに分類することができる. Rabhi は並列アルゴリズムを分割統治法, プロセッサファーム, プロセスネットワーク, 繰り返し変換の 4 つのアルゴリズムに分類している [4]. バーゼル大学で開発された BACS(Basel Algorithm Classification Scheme) は, プロセス構造, インタラクション, タスク分割, 実行構造など並列アルゴリズムを構成する各項目毎に分けて解析, 分類を行う. 特に, 実行構造に関しては, 規則的かつ同様のイベントの繰り返しで構成される構造を, 簡単な略語と記号で表現する方法を提案している [5]. 本システムはこれらの考えに基づき, 並列アルゴリズムを分類した形で保存する. そして新たな問題に対しても同様に解析, 分類を行い, その結果をもとに類似事例の検索を行う.

並列計算では, 計算対象の構造を複数の計算対象の集合と, 計算対象を分割したものの集合の 2 つのパターンに分類することができる. 1 つめの構造の例として, CG 計算の場合, 1 ピクセルについての計算をディスプレイ上の全ピクセルについて計算を行う. 2 つめの構造の例としては, 積分計算の場合, 与えられた 1 つの積分区間をはじめに複数の小さな区間に分割し, それぞれの部分区間についての計算を個別に行い, 最後に結果をまとめる. 本研究では, これら集合の 1 要素を単位要素, 1 単位要素についての計算を単位計算と呼ぶ. CG 計算では, 1 ピクセルについての計算, 積分計算では, 最も小さい部分区間についての計算がそれぞれ単位計算となる. 問題の解析ではこの単位要素の発見から始める. 単位要素の各プロセスへの分散方法と単位計算に用いられるデータを明らかにする. そしてプロセス間にまたがるデータの流れを基に, 最も適するアルゴリズムを Rabhi の分類による 4 つのアルゴリズムから選択する. それに応じて, 必要とされるインタラクション, 並列化手法を決定する. 最後に, システムが提示する並列構造の中から最も問題に適するものをユーザが選択し, その実行構造を BACS 表現したものを加え, インデックスを完成させる.

インデックスを基に類似事例の検索を行う. インデックスの全項目が一致する場合, 非常に類似した事例が検索され, スケレトンの修正が容易になり, 自動的な修正も期待される. そうでない場合でも, 部分的に関連性をもつ事例が検索され, プログラミングの手助けとなりうる. いずれの場合でも, 新たに作成された並列プログラムは, 事例検索の際作成したインデックスと共に, 新たな事例として事例ベースに追加され, 後に利用される.

本研究は仮想共有メモリ型並列マシン KSR-1[6] を対象に進めている. 仮想共有メモリ方式は, 分散型のメモリ構造をもつが, 単一のアドレス空間, 自動アドレス変換, データ管理の透過性により仮想的な共有メモリを実現し, 共有メモリ型マシン上での場合と同様に比較的簡単なプログラミングが可能となる [7][8]. また, 分散メモリ型マシンと同様のスケーラビリティーを持ち, 双方のメモリ構造が持つ特徴を兼ね備えてたものである [9]. 現在, 積分計算, マンデルブロ, DKA 法による高次方程式の解法, LU 分解法などいくつかの事例が完成している. これらの事例に含まれる並列構造を基に新たな並列プログラミングを行い, 事例ベースの作成を進めている.

本論文では, 第 2 章で事例ベース推論を用いた並列プログラミング支援システムの説明を行う. 第 3 章で並列アルゴリズムの一般的分類と BACS による分類について述べ, 第 4 章では作成した事例をアルゴリズム毎に紹介する. 第 5 章では, 類似事例の検索, 修正の方法を具体例を交えて説明する.

2 並列プログラミング支援システム

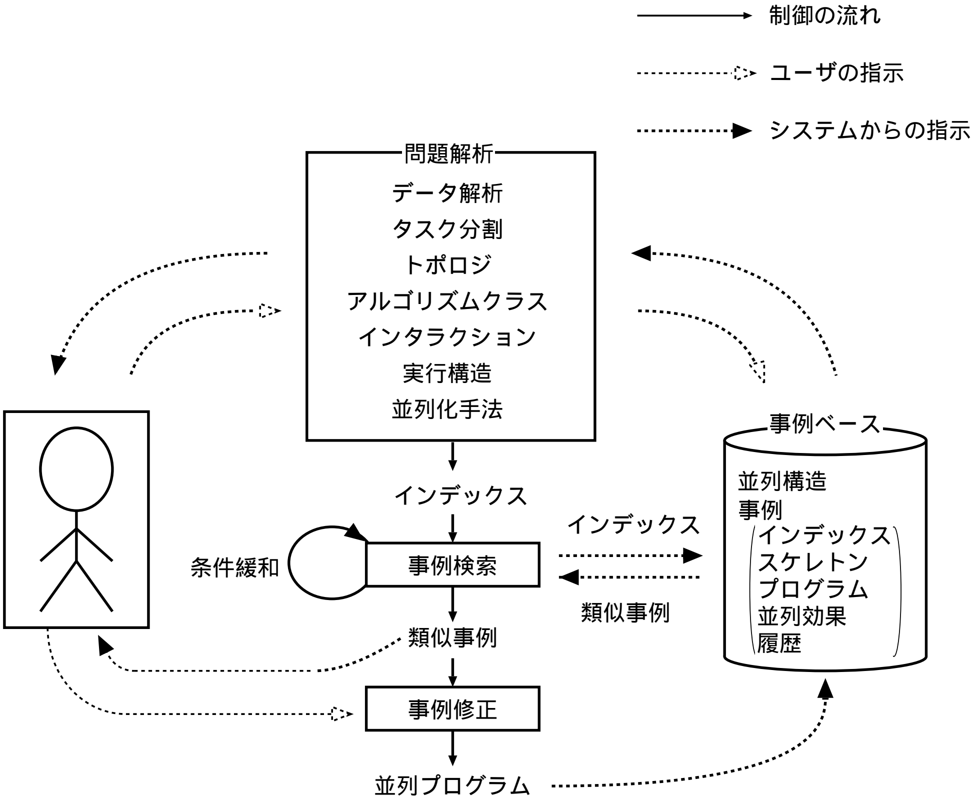


図 1: 並列プログラミング支援システム

図 1にシステム構成を示す．事例ベースの中には, インデックス, スケレトン, プログラム本体, 並列効果, 及び履歴からなる事例が蓄積されている．インデックスには並列プログラムの特徴が応用, 仕様, アルゴリズム, 終了条件, データ構造, タスク分割, トポロジ, インタラクション, 並列化手法, 及び実行構造の 10 個の項目に分けて抽出されている．

ユーザはインデックスの項目毎に問題解析を行う．各項目ではシステムから幾つかの選択例が提示され, その中から最も問題に適するものを選択する．データ依存性, 終了条件, トポロジをもとに並列アルゴリズムを決定する．並列アルゴリズムは, 一般的に分割統治法, プロセッサファーム, プロセスネットワーク, 繰り返し変換の 4 つに分類できる．各アルゴリズムに対する並列構造がシステムから提示され, インタラクション, 並列化手法を基に 1 つの並列構造を決定する．各並列構造には, その実行構造が BACS 表現で与えられているのでこれをインデックスに加える．

システムはインデックスを基に最も類似した事例を事例ベースの中から検索する．インデックスの全項目が一致する事例が検索された場合, 非常に類似度の高い事例が検索され, 容易な修正が期待される．全項目が一致しない場合でも, 検索条件を緩め, 1 つでも多くの項目が一致する事例を検索する．また, 事例に含まれる並列効果やプログラム本体, 履歴情報を基に, 速度向上の期待度や プログラミングの容易さなどを予測し, 幾つか検索される類似事例の中から最適な事例を決定する．

スケレトンには, タスク分割, 同期, 相互排除, 並列化手法など並列プログラムの中で最も重要な部分が記述されている. これを用いて並列プログラムの骨格部分を完成させる. さらに, プログラムの初期設定, 計算式, 結果出力, 及び終了条件やタスク分割条件の各種条件値など, 問題固有の部分はユー ザが肉付け又は修正を行うことにより, 目的の並列プログラムを完成させる.

作成された並列プログラムは, インデックスと共に新たな事例として事例ベー スに加えられ, 今後の新たな並列プログラム作成のために利用される.

3 並列アルゴリズムの分類

3.1 一般的分類

並列アルゴリズムは同時に実行される複数の計算ステージで構成される．それらは実行時に動的に生成される場合と，事前に生成数と生成時期が知らされている静的の 2 つのパターンに分類される．また，各計算ステージのデータ参照パターンは，一般的にチェーン，トリー，ハイパーキューブなどの規則的な通信パターンに対応している．

ハル大学の Rahi は計算ステージの生成パターンとデータ参照パターンの規則性により，並列アルゴリズムを分割統治法，プロセッサファーム，プロセスネットワーク，繰り返し変換の 4 つのアルゴリムに分類している [4].

3.1.1 分割統治法

図 2に示すように，分割統治法は問題を部分問題に分割し，解を得るアルゴリズムである．部分問題はそれ自身もさらに分割され，ある条件を満たすまで分割は繰り返される．最終結果は部分問題の結果を繰り返し結合することにより得ることができる．各部分問題は他と独立して計算を行うことが可能なことから並列性が生まれる．タスクは動的に生成され，木構造の通信パターンをもつ．

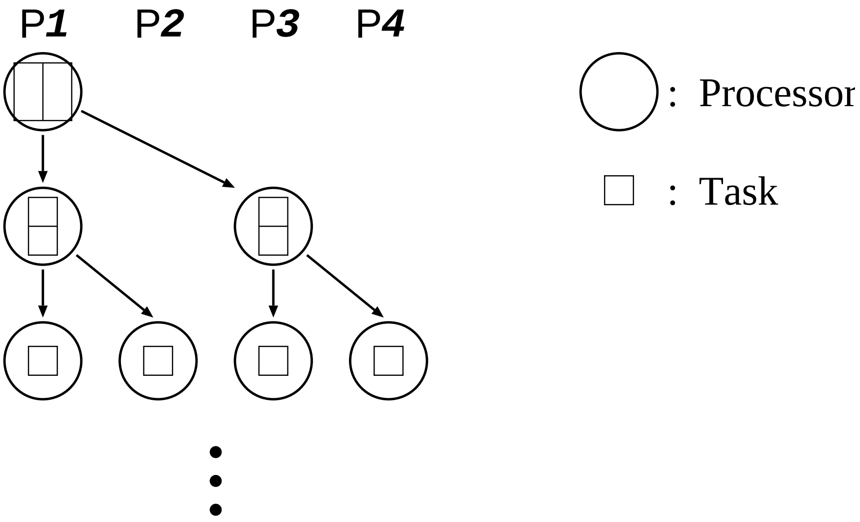


図 2: 分割統治法

3.1.2 プロセッサファーム

図 3に示すように，プロセッサファームは，初期段階でのみ問題の分割を行い，分割された問題は他とは完全に独立して計算が行われる．そして最後に計算結果を回収することにより解を得る方法である．1 つのマスターと複数のスレーブで構成され，マスターが問題の分割を行い，それら

を各スレーブに割り振る．スレーブは受け取った問題に対し計算を行い，結果をマスターに返す．マスターは全ての結果を回収した後それらをまとめ，最終的な結果を求める．分割方法は事前に明らかであり，プロセッサ数に応じた静的なタスク生成パターンを持つことができる．通信パターンはマスター&ワーカーとなる．

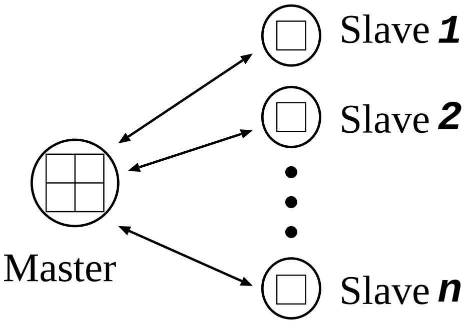


図 3: プロセッサファーム

3.1.3 プロセスネットワーク

プロセスネットワークは計算を複数のステージに分割するところに特徴がある．図4に示すように複数のデータが各ステージを流れる．入力データに依存しながら，各ステージは並行に動作することにより並列性が生まれる．全ステージで同じ命令が繰り返される場合と，各ステージで異なる命令が繰り返される場合とがある．通信パターンは一般的にパイプ，リングの形態を持つ．代表的なものとしてパイプライン処理が挙げられる．

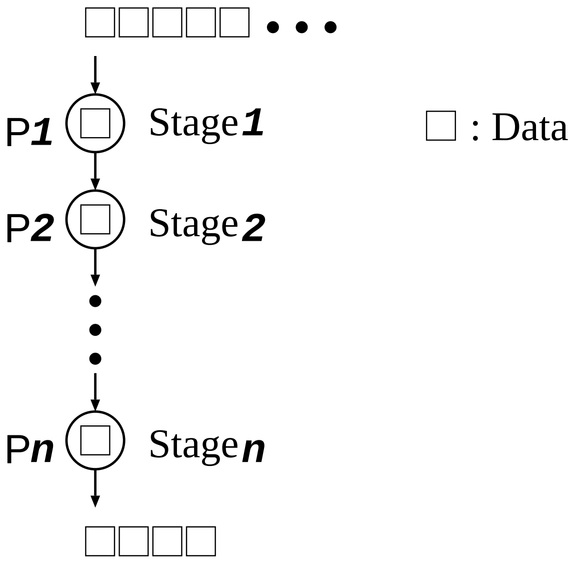


図 4: プロセスネットワーク

3.1.4 繰り返し変換

繰り返し変換は, 複数のオブジェクトを並列計算の対象として扱うところに特徴がある. 各オブジェクトは複数のローカルデータと, 全オブジェクト間で共有するグローバルデータを用いて, 新たな状態に変換される. 各オブジェクトが条件を満たす状態に変換されるまで繰り返し実行する. 図 5に示すように, 複数のオブジェクトに対し同時に変換を行うところに並列性が存在する. 1 回の変換ステップは次のオペレーションの内 1 つまたは幾つかで構成される.

- ローカルオペレーション
  各オブジェクトが持つローカルデータと他のオブジェクトが持つローカルデータをもとに局所計算を行う.
- 結合オペレーション
  いくつかのオブジェクトが結合し, 新たなオブジェクトグループを形成する.
- グローバルオペレーション
  グローバルデータの更新を行う.

結合オペレーションが存在しない場合は静的, 結合オペレーションによりオブジェクトの数が変化する場合は動的となる. 各オブジェクトは他の様々なオブジェクトからデータ参照を行うため, 通信パターンも様々な形態を持つ.

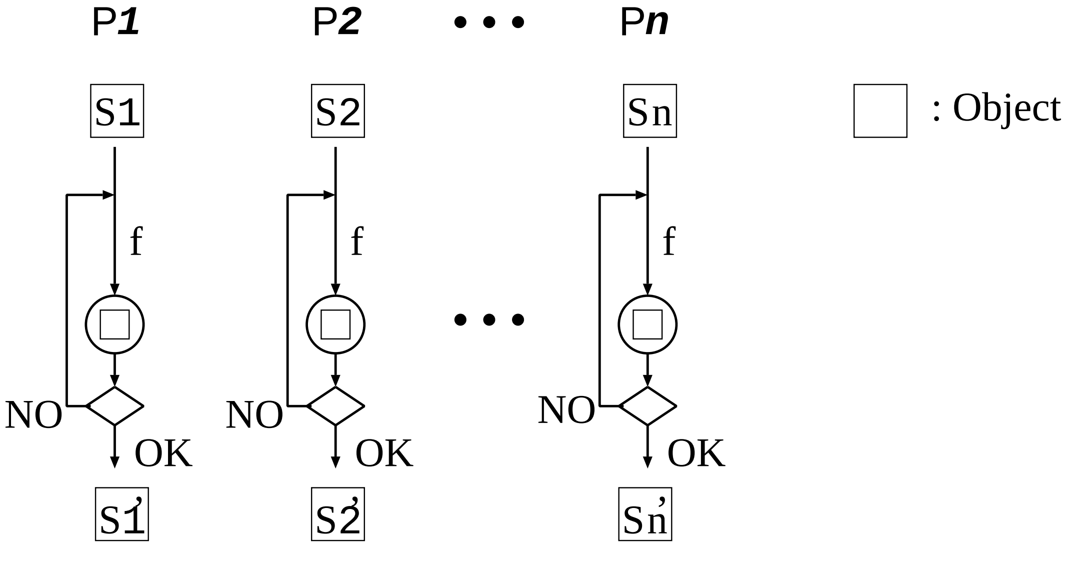


図 5: 繰り返し変換

3.2 BACS(Basel Algorithm Classification Scheme)

BACS はアルゴリズムの分類, 複雑さの解析などの方法論を明らかにするために提案された. BACS はアルゴリズムトポロジ, プロセス構造, データ分散, インタラクションメカニズム, 実行構造, 及びデータ/プロセス粒度の観点から並列アルゴリズムの分類を行っている [5]. 本研究では, これらの中からデータ分散, インタラクションメカニズム, 実行構造の分類に注目する.

3.2.1 データの分割, 分散

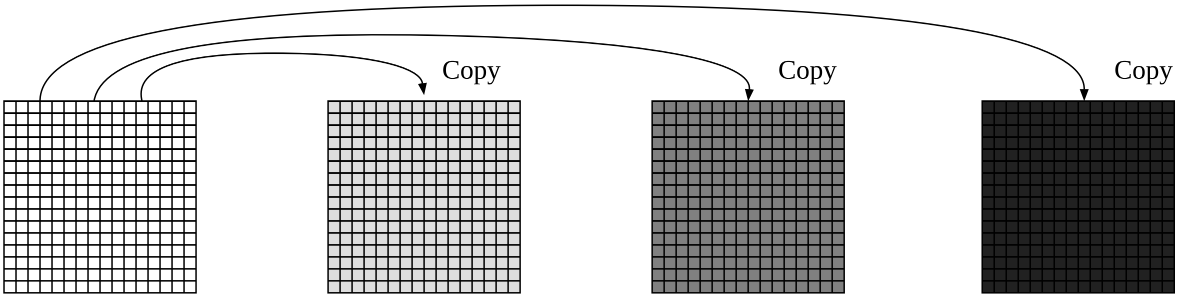
並列アルゴリズムでは, タスク分割の際, タスクと共にデータが与えられ, そのデータ上で各プロセスは動作する. データ分割コンセプトとして次の 2 点があげられる.

1. 何が分割されるのか (分割対象)
2. どの様に分割されるのか (分割方法)

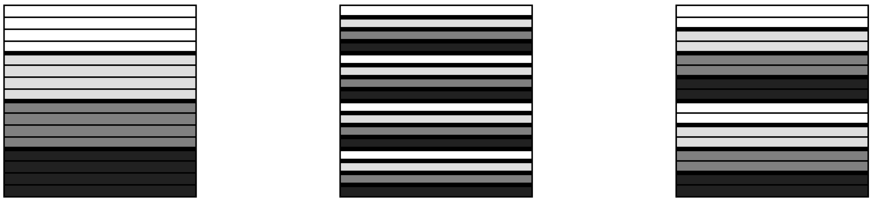
1 に対しては, スカラー変数などの場合, それをさらに小さい部分に分割することは不可能である. しかし, 配列, 集合, ファイルなどの構造化されたデータであれば, さらに小さいレベルに分割可能であるのか, それともこれ以上縮小することができない単体なのかを定義することができる. 2 に対しては, 分割可能であれば, データは単位要素の集合に分割される. 単位要素とは, 並列アルゴリズムで用いられる分割データの抽象レベルを意味する. これら単位要素を各プロセスへ割り振り, プロセスは受け取ったデータ上で作用する. 負荷の均衡, 通信の効率化, アルゴリズム依存など様々な理由と共に分割・分散方法もいくつか存在するが, その中で一般的に次の 4 つのパターンが多く用いられる.

- コピー
  データ全体のコピーを各プロセスに割り振る.
- ブロック
  データをプロセス数と同数のブロックに分割する. これらブロックは隣接した単位要素の集まりである. 各ブロックをプロセスに 1 対 1 に割り振る.
- サイクリック
  順番に並べられた単位要素をある周期に沿って各プロセスに割り振る.
- ブロック/サイクリック
  複数のブロックに分割し, それらをサイクリックに割り振る.

図 6では, これら分割・分散パターンを 2 次元配列を用いた例で示す. 括弧の中は分割単位を示す.



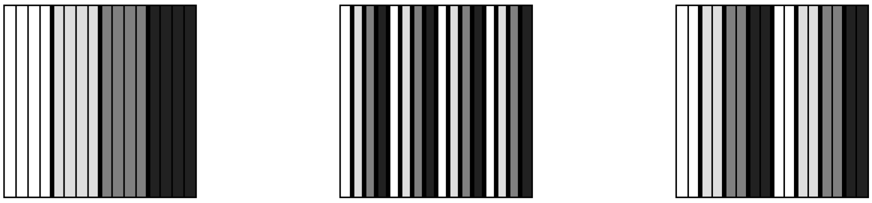
(a): コピー



(b-1): ブロック
(行ブロック)

(b-2): サイクリック
(行)

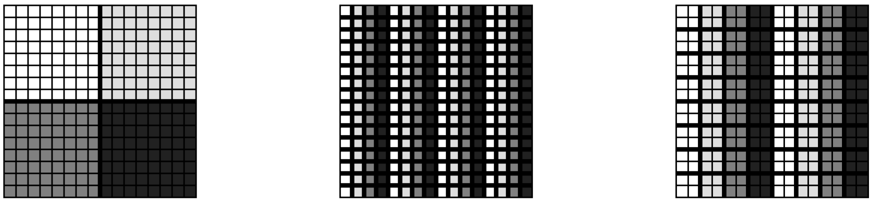
(b-3): ブロック&サイクリック
(行ブロック)



(c-1): ブロック
(列ブロック)

(c-2): サイクリック
(列)

(c-3): ブロック&サイクリック
(列ブロック)



(d-1): ブロック
(サブ行列)

(d-2) : サイクリック
(要素)

(d-3): ブロック&サイクリック
(サブ行列)

□ : プロセス1 ▒ : プロセス2 ▓ : プロセス3 ■ : プロセス4

図 6: データの分割, 分散

3.2.2 インタラクション

インタラクションは，インタラクションを行う目的，インタラクションに参加するプロセス数，及び参加するプロセス間の関係 の3点で次のように分類することができる．

1. 目的
   - データ交換
     プロセスが次の処理へ進むためにあるデータが必要な時，そのデータが他のプロセスから提供されなければならない場合，データ交換のためのインタラクションが発生する．
   - 命令の流れとデータの一貫性 (コード一貫性)
     排他制御やシグナル送信などを行うことにより，他のプロセスを制御する．共有データのアクセスや，他のプロセスが行う処理を待つなどの場合に用いるインタラクションである．
2. 参加プロセス数
   - 直接インタラクション
     2つのプロセス間のインタラクションのみ定義する．各プロセスは互いにインタラクションを行う相手を認識し，1対1の関係をもつ．
   - グローバルインタラクション
     3つ以上のプロセス間のインタラクションを定義する．
3. プロセス間関係
   - 分離インタラクション
     各プロセスは匿名でインタラクションに参加する．直接相手プロセスを指定するのではなく，共通にアクセスできるものを指定し，それを介して間接的なインタラクションを行う．
   - 結合インタラクション
     プロセス間の関係は明確に定義され，各プロセスは直接指定した相手とインタラクションを行う．

以上をまとめると，表1に示す6つのインタラクションパターンが構成される．対象とする並列マシンのアーキテクチャによっては，これら全ての機能を持つとは限らない．また，表には，各パターンの例が示されているが，ここに何が入るかはOS，並列処理言語により様々である．本研究で用いるKSR-1の場合，仮想共有メモリ型の性質により，コード一貫性の目的を持つ3つのインタラクションパターンが該当する．その理由として，仮想共有メモリ型のマシンにはメッセージパッシングによるデータ交換の概念が存在しない．その代わりに，共有変数に値を書き込み，それを読み出す方法によりデータ交換を実現する．そのためデータ交換インタラクションは存在せず，これに該当するインタラクションパターンは省かれる．しかし，コード一貫性を保つためのインタラクションが必要となるため，それらに該当する表の中の3つのインタラクションが必要となる．これら3つのインタラクションパターンの中で，KSR-1上で用いられるインタラクション例を紹介する．

表 1: インタラクションパターン

直接インタラクション		グローバルインタラクション			
結合インタラクション		分離インタラクション		結合インタラクション	
データ交換	コード一貫性	データ交換	コード一貫性	データ交換	コード一貫性
exchange send receive	signal, wait	read, write (共有変数) put get (*mailbox*)	lock, unlock (相互排除) PV(セマフォ)	broadcast fan_in	barrier

直接結合インタラクションの例としては, 共有変数を用いたシグナル-ウエイトが挙げられ, これを用いたインタラクションの様子を図 7に示す. 2 つのスレッドはフラグの役目を果たす共有変数 (flag1, flag2) を 1 つずつ持つ. フラグの値が 1 の時, そのスレッドはビジー , 0 の時はアイドル状態にあることを示す. 先に処理を終えたスレッドは, もう一方のスレッドの状態をフラグ変数から読み取る. その結果, ビジー状態であれば自分をアイドル状態にし, 相手側のスレッドが処理を終了するまで待つ. アイドル状態であればそのスレッドをビジーな状態 (flag=1) にし, 処理を再開させる.

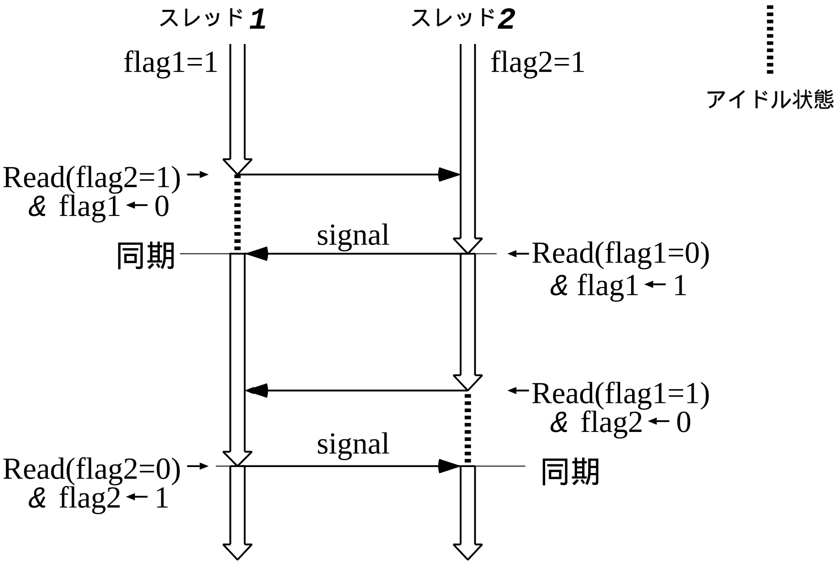


図 7: 共有変数を用いたシグナル·ウエイト

グローバル分離インタラクションの例として, ミューテックスを用いた排他制御が挙げられる. 複数のスレッドは同時に共有変数への書き込みを行うことが可能なため, データの一貫性を保つためのデータアクセス制御を行う同期が必要となる. ミューテックスは共有データにアクセスするコードフラグメントの実行の同期をとることにより, 共有データに逐次アクセスするために使われるロックである. 図 8では, 最初に要求を出したスレッド 2 がロックを行い, 共有変数への書き込みを行う. その間にスレッド 3 とスレッド 1 も要求を出すが, 既にスレッド 2 がロック中であ

るので, それが終了するまで待つ. 1 つのミュー テックスをロックできるのは必ず 1 つのスレッドのみであるため, 複数のスレッドが同時に 1 つの共有変数に書き込むことを防ぐことができる.

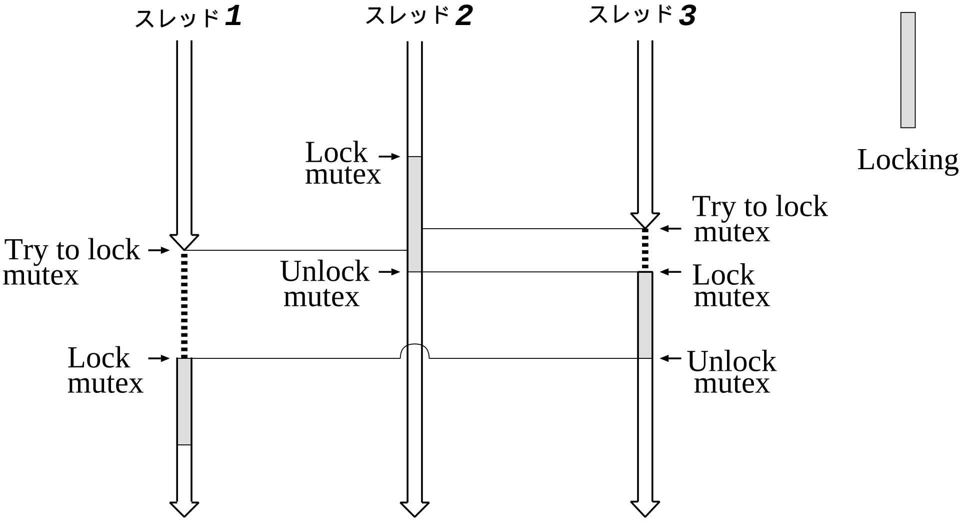


図 8: mutex による排他制御

グロー バル結合インタラクションの例として, バリヤ同期が挙げられる. バリヤ同期の様子を図9に示す. バリヤ同期ではチェックインとチェックアウトの 2 点で同期を行う. スレーブはチェックインを要求後, マスター からの信号を得るまで待ち状態になる. マスター がバリヤにチェックインした時点で全てのスレッドが同時にチェックインを行い, 同時にコード実行を開始する. 実行終了後, スレーブはマスター にチェックアウトを報告し, 即座に次の処理へと進むことができる. しかし, マスター はチェックアウトに到達した時, 全てのスレーブがチェックアウトの報告をするまで待ちの状態になる.

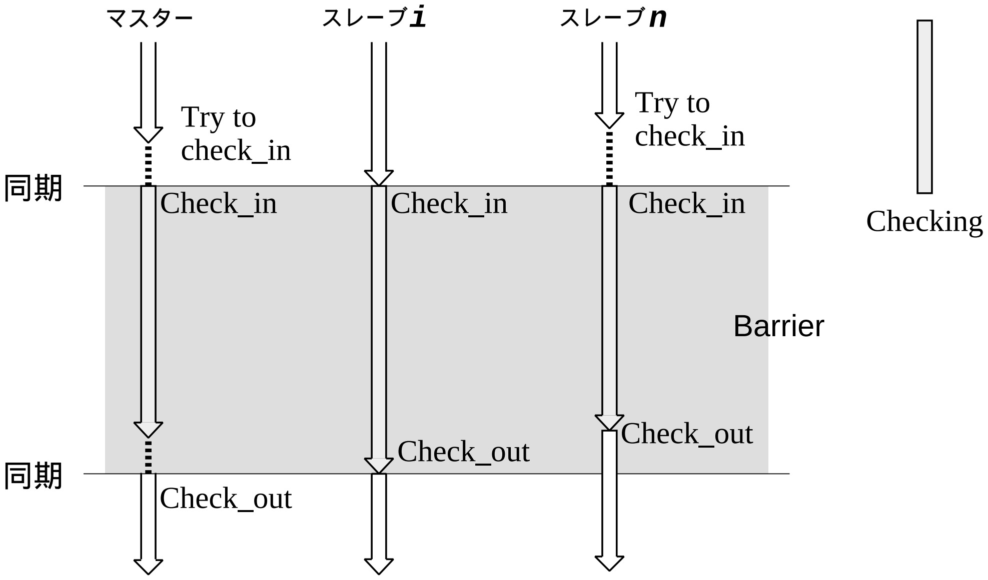


図 9: バリヤ同期

3.2.3 実行構造

一般的に, 並列アルゴリズムはそれを構成する数個のイベントの構成順序と, 各イベントに参加するプロセスの関係により, その実行構造を表現することができる. BACS では, 簡単な略語と記号を用いてこれらを表現する方法を提案している. ここでは, 各イベントの説明と BACS による並列アルゴリズム実行構造の表現方法の説明を行う.

(1) イベント

イベントには次に挙げる 3 つのイベントが存在し, それぞれアルファベット大文字を用いて表現する. また, イベント内容を必要に応じて下付文字で示す.

- 計算イベント (C)
  ロ- カル計算を表す. ロ- カルデ- タの変換などを行い, 他との相互作用は存在しない.
  (例) C_{ray_trace}, $C_{rendering}$
- インタラクションイベント (I)
  同期, 相互排除, デ- タ交換などプロセス間の相互作用を表す.
  (例) I_{mutex_Lock}, $I_{wait(thread_i)}$
- デ- モンコ- ルイベント (D)

  並列アルゴリズムを制御するための高い権力を持ち, 次の 2 つの機能を持つ.

  – プロセス管理
    プロセスの生成や除去を行う. プロセスはデ- モンに対し, 新たなプロセスの生成や, 既存プロセスの除去などの要求を行う.
  – デ- タの分散
    プロセス生成の際デ- タを伴わせたり, プロセス消去の際結果デ- タの回収や処分を行う.

  (例) $D_{create(thread_i)}$, $D_{delete(thread_j)}$

(2) イベント–プロセス 関係

1 つのイベントに複数のプロセスが参加することにより, そのイベントは多重した実行が行われ, それが複数のプロセッサ上であれば並列処理が行われる. プロセスのイベント参加方法は次の 2 つに分けることができ, 肩付文字を用いて, それらを識別する.

- ト- タルイベント (t)　全プロセスがそのイベントに参加.

$$\boxed{\text{イベント}}^{t}$$

- パ- シャルイベント (p)　全プロセスの内一部のプロセスがそのイベントに参加.

$$\boxed{\text{イベント}}^{p}$$

パ- シャルイベントの場合, 参加プロセスについての情報を p に続く括弧の中に示す. その内容には次の 2 つが存在する.

1. 定数の定義
   参加プロセスの数を定義．例えば，1 つのプロセスが計算イベントに参加する場合，$C^{p(1)}$ と示す．
2. 指定子の定義
   プロセスグループを示す指定子を定義．例えば，偶数のプロセス番号を持つプロセスグループを $even$ と定義しておく．$even$ グループに属するプロセスがインタラクションイベントに参加する場合，$I^{p(even)}$ と示す．

(3) ループ

並列アルゴリズムは一般的にイベントの流れで構成されるが，それらは単一のイベント，または数個のイベント群の繰り返し (ループ) となっている場合が多い．ループには次の 2 つの形態が存在する．

- 固定回の繰り返しを行うループ (Fix)
  ループ回数はループに入る以前から決められている．$For-Loop$ 文に相当．

$$Fix\ \{\ \text{イベント}\ 1,\ \text{イベント}\ 2,\ ...\ \}$$

- 不定回の繰り返しを行うループ ($Cond$)
  ループ回数はループ内の実行状態に依存．$While-Loop$ 文に相当

$$Cond\ \{\ \text{イベント}\ 1,\ \text{イベント}\ 2,\ ...\ \}$$

繰り返されるイベントは，ループを示すキーワードに続き，括弧内に記述される．

3.3 並列構造の分類

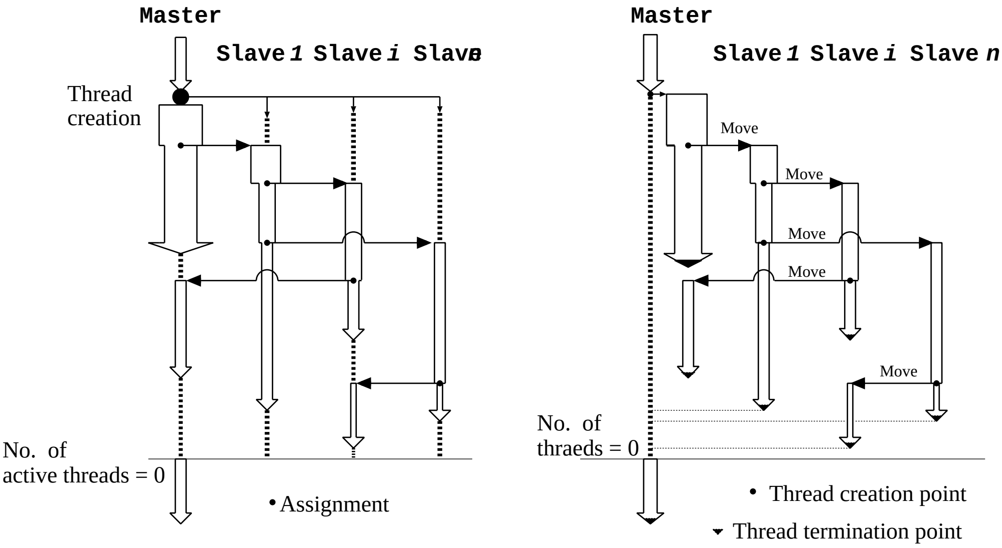
BACS 表現をユーザが理解することは困難である．また，マルチスレッドの動的振舞いや，同期メカニズムは BACS 上では実現しづらい．本システムではこれを解決するため，並列構造を用意する．並列構造は，マルチスレッドが同時に動作する区間 (並列ブロック) のスレッドの状態や，同期構造を図を用いて表現し，視覚的に捕らえることによりアルゴリズムの動的振舞いを確認する．また，後に検索されるプログラムやスケレトンの内容理解にも役立つ．これをアルゴリズム毎に代表的なものを幾つか用意する．事例検索の際，インデックスの選択項目を基に，該当する並列構造をいくつか提示し，ユーザがその中から適当なものを選択する．各並列構造には図による表現と共に，BACS 表現も収められている．選択した並列構造は，その BACS 表現がインデックスに加えられ，類似事例の検索に用いられる．次に代表的な並列構造をアルゴリズム毎に紹介する．

(1) 分割統治法

図 10に分割統治法の並列構造を 2 つ示す．共に与えられたタスクに対し計算を行い，そしてタスクを分割する．一方の分割タスクを他のスレッドに割り当て，もう一方の分割タスクは引き続き自分が担当する．これをある条件を満たすまで繰り返す．ただし，パラレルリージョン型は並列ブロック開始と同時にプロセッサ台数分スレッドを生成する．各スレッドはタスクが割り当てられ

るのを待つ．タスクを受け取り計算を行った後，アイドルスレッドの検索を行う．検索できた場合はそのスレッドに分割タスクを割り当てる．自分の処理が終了し，アイドルスレッドとなった場合は消滅せず，次のタスク割り当てを待つ．アクティブスレッド数が 0 になった時点で並列ブロックは終了する．

一方，スレッド生成型では 1 つのスレッドを生成することにより並列ブロックが開始する．タスクの分割は，まずアイドルプロセッサを検索する．検索できた場合は，新たなスレッドを生成し，そのスレッドに分割タスクを割り当てる．そして検索されたアイドルプロセッサに移し，そこで処理を行わせる．処理終了後スレッドは消滅する．スレッドの数が 0 になった時点で並列ブロックは終了する．



(a) パラレルリージョン型　(b) スレッド生成型

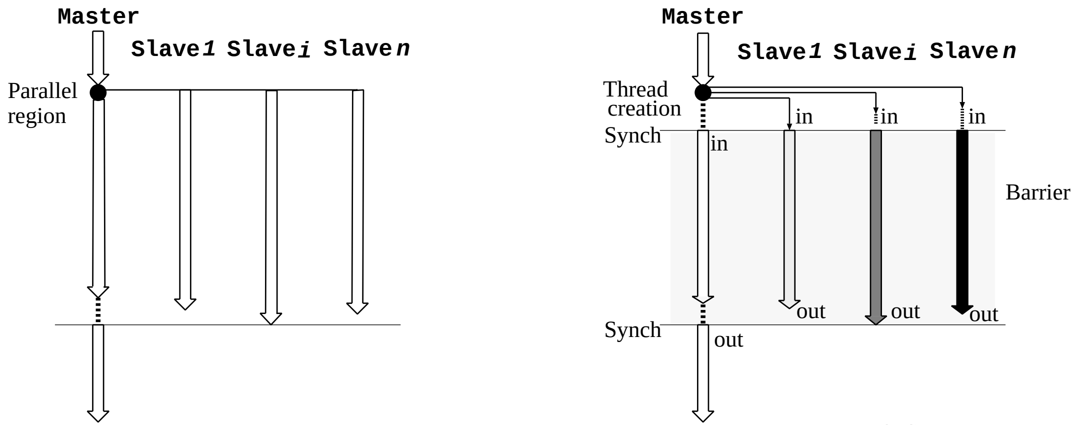
図 10: 並列構造 (分割統治法)

(2) プロセッサファーム

プロセッサファームの並列構造を図 11に示す．パラレルリージョン型ではその性質により，並列ブロックの開始と終了時にはバリヤ同期が自動的にとられ，各スレッドは同一の処理を行う．一般的なプロセッサファームはこのタイプに属する．スレッド生成型はスレッドの生成，消去，及び並列ブロック開始時と終了時の同期などをプログラム内で陽に指定することができる．各スレッドの処理は異なっても良く，問題に応じた柔軟な構造でスレッドを操作することができる．

(3) プロセスネットワーク

プロセスネットワークの並列構造を図 12に示す．マスターがスレーブを生成後，スレーブ 1，スレーブ 2，スレーブ 3，マスターの順にデータが流れ，パイプライン処理される．スレーブ 1 がパイプラインの最上位ステージ，マスターが最下位ステージとなる．あるステージと次のステージとの同期は図 7の方法を用いて行う．各ステージでの処理は均等であることが望ましい．



(a) パラレルリージョン型　(b) スレッド生成型

図 11: 並列構造 (プロセッサファーム)

(4) 繰り返し変換

繰り返し変換の並列構造を図 13に示す．マスターがスレーブスレッドを生成し, 各スレッドは同一処理を行う．各変換ステップは, バリヤで囲むことにより同期が取られ, 同時実行される．マスターは全スレッドの変換終了を確認後, 変換結果の評価を行う．変換結果が条件を満たした場合, マスターはスレーブスレッドの消去を行い, 並列ブロックは終了する．満たさない場合は次の変換ステップに入る．2 種類のバリヤが使われているのは, バリヤ機能の性質が関係する．マスターがチェックイン中は, 同一バリヤに対しスレーブのチェックイン許可は続く．従って, 1 つのバリヤのみ用いた場合, チェックアウトを行ったスレーブは即座に再チェックインが可能となり同期が成立しない．このような理由で 2 種類のバリヤを用いる．

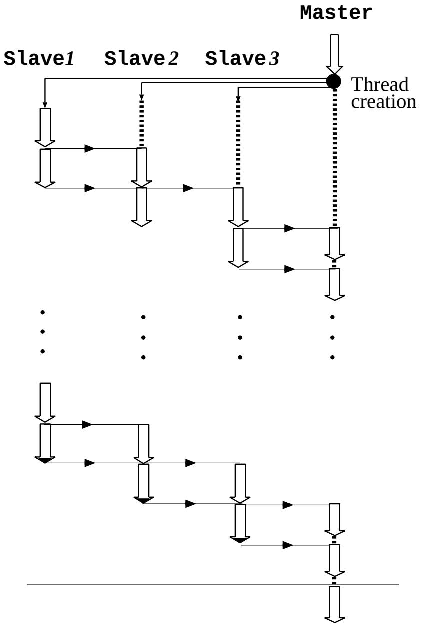


図 12: 並列構造 (プロセスネットワーク)

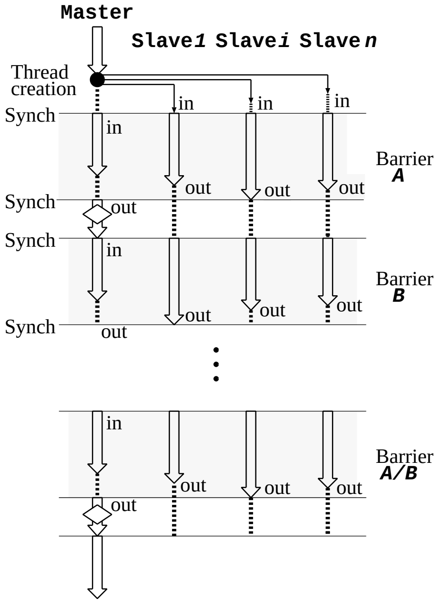


図 13: 並列構造 (繰り返し変換)

4 事例ベースの構築

事例はインデックス, スケレトン, プログラム本体, 並列効果, 及び履歴から成る. 優先度を持つ複数のインデックスが類似事例の検索に利用される. 応用, 仕様, アルゴリズム, 終了条件, データ構造, 及びトポロジはマシン独立なインデックスである. 一方, 同期と並列化手法は Posix 標準に準拠している. タスク分割, 実行構造は BACS に基づいた記述を行う.

並列プログラムの骨格であるスケレトンは, スレッド操作, 同期, 排他制御, タスク分割などの並列構造を示す部分と, 変数初期化, 計算部, 計算結果の回収などを行う部分が記述されている. スケレトンはプログラムの骨格を示し, これに必要な情報が補われれば, 目標のプログラムが生成される.

並列効果はその事例においてどの程度の速度向上が得られたのかを示し, 新たな並列プログラミングの際の参考にする.

履歴はその事例がどのように修正されたか, それが成功したか失敗したかを示す.

この章では, まず事例の一般表現を示す. 次に事例の例をアルゴリズム毎に 2 つずつ示し, さらに, それぞれのスケレトンを示す. なお, プログラム本体は付録とする.

- 分割統治法
    1. クイックソート
    2. 積分計算
- プロセッサファーム
    1. マンデルブロ
    2. KMP 法
- プロセスネットワーク
    1. LU 分解法
    2. バブルソート
- 繰り返し変換
    1. DKA 法
    2. ワーシャルアルゴリズム

4.1 事例の一般的表現

(1) インデックス

応用	問題の応用分野
仕様	問題の定義
アルゴリズム	分割統治法, プロセッサファーム, プロセスネットワーク, 繰り返し変換
終了条件	アルゴリズムの終了条件
データ構造	ソースデータ, 結果データ
タスク分割	(分割対象, 分割単位, 分散方法)
トポロジ	トリー , ワーカー , マスター &ワーカー , パイプ, リング, メッシュ, ハイパーキューブ, ...
インタラクション	シグナル·ウエイト, ミューテックス, バリヤ
並列化手法	パラレルリージョン, パラレルセクション, スレッドライブラリ
実行構造	$BACS$

(2) スケレトン

- 並列プログラムの骨格
- タスク分割, 相互排除, 同期, スレッド使用法

(3) プログラム本体

(6) 並列効果

- 並列処理による速度向上比
- 問題とアルゴリズムの適応度

(5) 履歴

- 参照頻度, 成功/失敗
- 事例ベースの更新

4.2 分割統治法

4.2.1 クイックソート

(1) 問題の定義

ソート対象となる要素列の中でサンプル要素を 1 つ選び, それを基準にそれより小さいグループと大きいグループに分割する. 各々のグループの整列はこの分割を再帰的に適用する, という方法で行うソート法である.

(2) 並列計算

要素列は 1 次元配列 data[ ] に格納されている. 中央の配列位置に格納されている要素を基準とした大小のグループ分けを行い, 2 つの部分要素列が生成される. アイドルスレッドが存在する場合, 大きい値の要素からなる部分要素列のソートをそのスレッドに行わせ, 小さい値の要素からなる部分要素列のソートは自分で行う. アイドルスレッドが存在しない場合は, 両方の部分要素列のソートを自分で行う. 以降, 各部分要素列の中で, 中央の配列位置に格納されている要素を基準とし, 再帰的にソート, 割り当てを繰り返す. 全スレッドがアイドルスレッドとなった時, ソート終了となる.

タスクの分割条件は,

- 現在与えられている部分要素列が未整列

また, アイドルスレッドに一方の分割タスクを割り当てるための条件は,

- タスク量が一定サイズ以上の時.
  (部分要素列の要素数 $\geq$ 全要素列の要素数の 1%)
- アイドルスレッドが存在する.

である.

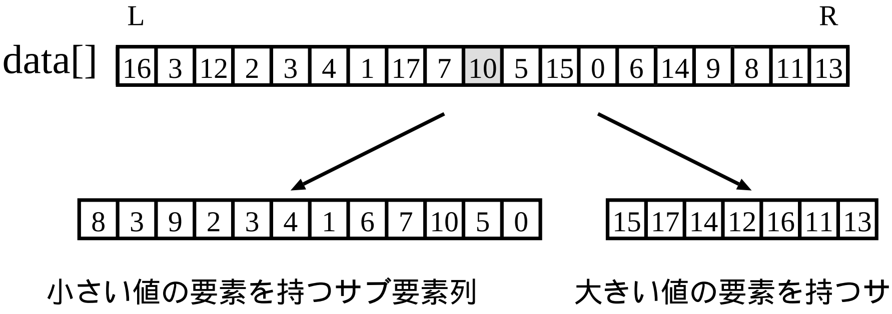


図 14: クイックソート

(3) インデックス

表 2: クイックソート のインデックス

応用	ソーティング
仕様	要素列 $(data[L] \sim data[R])$ を, $data[\frac{L+R}{2}]$ 未満の要素からなる部分要素列と, $data[\frac{L+R}{2}]$ 以上の要素からなる部分要素列の 2 つに分割.
アルゴリズム	分割統治法
終了条件	全ての部分要素列がソート完了
データ構造	ソーステ゛ータ: 整数型一次元配列 $data[L] \sim data[R]$
	結果データ ： 整数型一次元配列 $data[L] \sim data[R]$
タスク分割	($data$, 要素ブロック , ブロック)
トポロジ	トリー
インタラクション	ミューテックス
並列化手法	パラレルリージョン
実行構造	$\sum_{i=1}^{スレッド数} Cond\{I^{p(i)}_{Wait\ New-task}, Cond(C^{p(i)}, I^{p(i)}_{Search\ Idle-thread})\}$

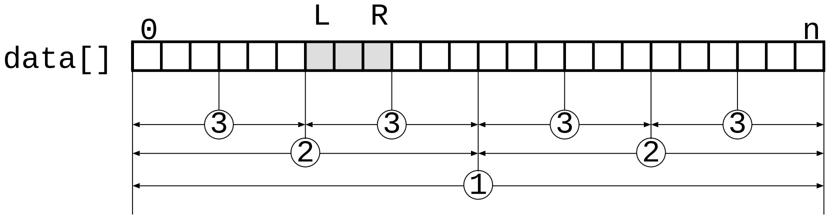


図 15: クイックソートのタスク分割

処理は図 15に示す番号に沿って進められる. 受け取ったタスクを 2 つに分割し, 一方を別のスレッドに割り当てる構造により, トポロジはトリーとなる. アイドルスレッドの検索時, 排他制御を行うためミューテックスを用いる. はじめにユーザが指定した数のスレッドを生成し, それらスレッドは全て共通の関数を実行するため並列化手法にはパラレルリージョンを用いる. 並列構造には, 図 10(a) に示す構造が適応される. 各スレッドの実行構造は次に示す通りである. これを BACS 表現したものがインデックに書かれている.

1. タスクが割り当てられるまで待つ.
2. 受け取ったタスクに対し計算, そして分割を行う.
3. アイドルスレッドを検索する. 検索が成功したら, 一方の分割タスクをそのスレッドに割り当て, もう一方の分割タスクは自分に割り当てる. 検索が失敗したら, 両方の分割タスクを自分に割り当てる.
4. 新たな分割の必要がなかったら, 1 へ.

(4) 並列効果

要素数 100, 1000, 10000 の 3 通りの方法で計算を行った. プロセッサ台数 16 台付近で速度向上の限界が見られる. タスクをアイドルスレッドに割り当てる際, プロセッサ間の通信が行われる. この際の通信量がクイックソートの場合非常に大きい. さらに, プロセッサ台数の増加と比例して通信回数が増加することが, 速度向上比の低下の原因と考えられる. このような並列構造に対しては, タスク割当ての際必要となる通信量に対し, 十分負荷の大きい計算が行われる問題の場合, 高い並列効果が期待できる.

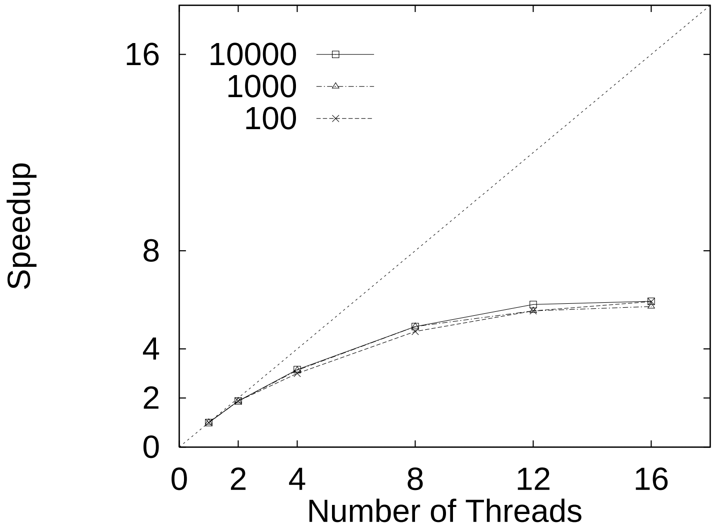


図 16: クイックソートの並列効果

4.2.2 積分計算

(1) 問題の定義

ある精度を得るまで分割・計算を繰り返す適応求積法 (Adaptive Quadrature) を用いた積分計算を行う. 図 17の左側の図における台形面積 $S1$ と, 右側の図における 2 つの台形面積の和である $S2$ との誤差がある一定値以下になるまで台形の分割を続ける. 誤差が一定値以下になったところで分割を止め, $S2$ の値を計算結果用の変数に加える.

(2) 並列計算

他のプロセッサよりも早く処理の終わったプロセッサの ID を得て, そのプロセッサに未処理の計算の一部をスレッドとして移動させる. プロセッサの空きが少なくなるために実行時間は短くなる.

タスクの分割条件は,

- $(S2 - S1) > \epsilon$ （ϵ はユーザが入力)

また, アイドルプロセッサに分割タスクと共に新たなスレッドを割り当てるための条件は,

- スレッド生成回数 < 96 かつ $(S2 - S1) > \epsilon \times 100000$

とする.

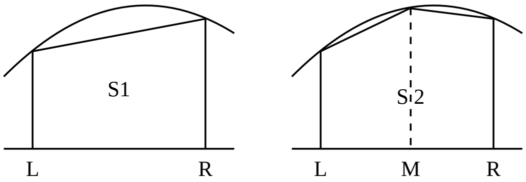


図 17: 適応求積法による積分計算

(3) インデックス

表 2: 積分計算のインデックス

応用	数値計算
仕様	2 つの x 値 (L, R) を得, 関数 $f(x)$ に対する図 17の台形 S1, S2 の面積, 及び 2 つの面積の誤差を求める.
アルゴリズム	分割統治法
終了条件	$(S2 - S1) < \epsilon$
データ構造	ソースデータ: 実数型 L ,R (積分区間を示す 2 つの x 座標値) 関数 $f(x)$ の式データ 結果データ : 実数型 1 次元配列 ans[スレッド数]
タスク分割	([SL ,SR]$_{(もとの積分区間)}$, [L , R]$_{(サブ積分区間)}$), ブロック) (関数 $f(x)$ の式データ , , コピー) (data , 1 要素 , ブロック)
トポロジ	トリー
インタラクション	ミューテックス
並列化手法	スレッドライブラリ
実行構造	$\sum_{i=1}^{96} Cond\{C^{p(i)}, I^{p(i)}_{Search\ Idle-processor}, D^{p(i)}_{create\_NewThread}\}$

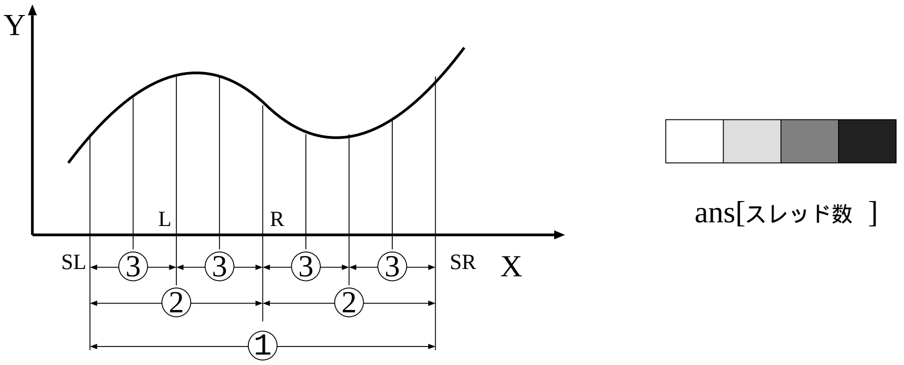


図 18: 積分計算のタスク分割

処理は図 18に示す番号に沿って進められる．クイックソート と同様に，受け取ったタスクを 2 つに分割，そして割り当てを行うため，トポロジはトリー となる．また，積分計算でもアイドルプロセッサの検索時に排他制御を行うため，ミューテックスを用いる．スレッドは全て共通の関数を実行する．しかし，1 つのスレッドから始まり，分割タスクの生成と共にスレッドが生成されるため，並列構造には図 10(b) に示すスレッド生成型を用いる．各スレッドの実行構造は次に示す通りである．

1. 生成と同時に割り当てられたタスクの計算を行う.
2. アイドルプロセッサを検索する.
3. 検索が成功したら，新たなスレッドを生成する．分割タスクをそのスレッドに割り当て，移動させる．もう一方の分割タスクは自分に割り当てる．
   検索が失敗したら，両方の分割タスクを自分に割り当てる．
4. 計算の結果，新たな分割の必要がなかったら，結果を格納し，消滅する．

(4) 並列効果

$f(x) = 4x^4 - 5x^3 + 3x^2 + 9x - 1$ に対し，区間 [-100, 100] で積分を行った．16 台のプロセッサ使用時，約 7 倍の速度向上が得られた．スレッド生成型の並列構造では，スレッドの生成と移動に伴うオーバヘッドが大きい．これらの影響をあまり受けないためには，計算負荷の大きいタスクのみ割り当てを行うための処理が必要である．

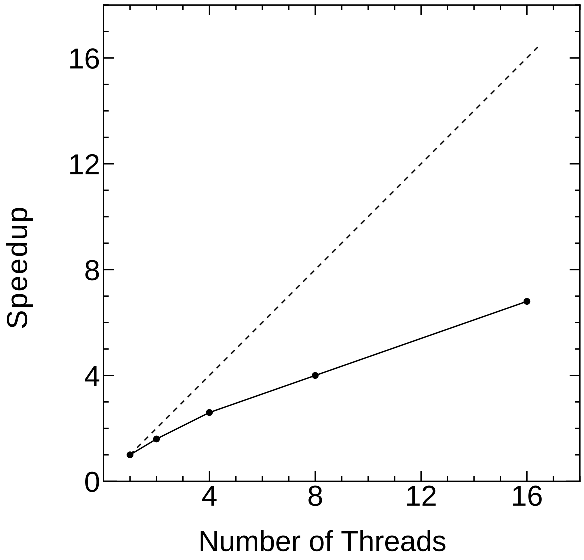


図 19: 積分計算の並列効果

4.3 プロセッサファーム

4.3.1 マンデルブロ

(1) 問題の定義

複素関数 ($f(Z) : Z_{i+1} = Z_i^2 + C$) に対して, 初期値を $Z_0 = (0, 0)$ と置き, $Z_1 = f(Z_0)$, $Z_2 = f(Z_1)$, $\cdots$, $Z_k = f(Z_{k-1})$, のように反復計算を繰り返す. 複素平面上の座標値 C の値を変化させ, $Z_k(k \to \infty)$ の値が収束か, 発散かを求める.

$(-2.0 \leq \text{実部} < 0.5)$, $(-1.25 \leq \text{虚部} < 1.25)$ の範囲の複素平面を $X \times Y$ のメッシュに区切り, $X \times Y$ 個の点について上の反復計算を行う. その結果, $|Z_i| > 2.0$ のとき発散, $i > 200$ のとき収束するとし, 後者の結果となる座標点がマンデルブロ集合の要素となる. 各座標点に対し, マンデルブロ集合に属する (1), 属さない (0) を 2 次元配列 data[X][Y] に格納する. これをもとに, マンデルブロ集合に属する座標点の合計数を $sum1$ に, 全ての座標点についてのループ回数を合計したものを $sum2$ に格納する.

$$
\begin{aligned}
f(Z) = Z_i^2 + C \ , \ Z_0 = (0, 0) \\
Z_1 = f(Z_0) \\
Z_2 = f(Z_1) \\
\cdot \\
\cdot \\
\cdot \\
Z_k = f(Z_{k-1}) \\
Z_k(k \to \infty) \ : \ (\underline{\text{収束}}or\text{発散})
\end{aligned}
$$

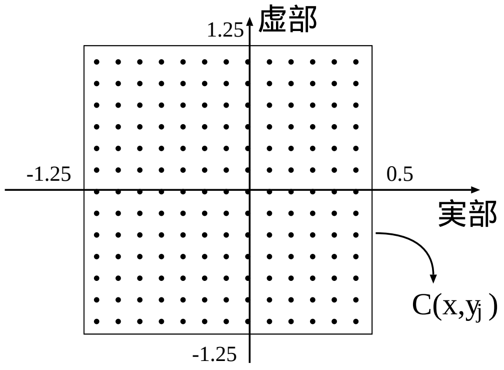


図 20: 複素座標平面

(2) 並列計算

複素座標平面を 1 列毎に分割し, それらをサイクリックにスレッドに割り振る. 各スレッドは割り当てられた範囲の中で計算を行い, その結果は 2 次元配列 data[X][Y] 中の各複素座標点に対応した配列位置に格納する. 全スレッドが計算を終了後, マスタースレッドが data[X][Y] に収められている結果をもとに $sum1$, $sum2$ を求める.

(3) インデックス

表 4: マンデルブロのインデックス

応用	数値計算
仕様	複素座標 $Z_i(x, yj)$ に対し, $Z_{i+1} = Z_i^2 + C \quad (i = 0, 1, 2, ...)$
アルゴリズム	プロセッサファーム
終了条件	なし
データ構造	ソースデータ: 複素平面座標値 $Z_i(x, yj)$ 結果データ : 整数型 2 次元配列 data[X][Y]
タスク分割	(複素座標平面 , 1 列 , サイクリック) (data[X][Y] , 1 列 , サイクリック)
トポロジ	マスター&ワーカー
インタラクション	なし
並列化手法	パラレルリージョン
実行構造	C^t

各スレッドは, マスタースレッドから割り当てられた分割タスクを受け取り, その結果を再びマスターに返す構造からトポロジはマスター&ワーカーとなる. 計算結果は割り当てられた座標値に対応する配列位置に格納するため, スレッド間の相互作用は無い. また, 結果データはアルゴリズムの終了条件に作用しないため, 終了条件も無い. 全スレッドの計算終了をマスタースレッドが待つための同期が必要なため, 並列化手法には図 11(a) に示すパラレルリージョン型が用いられる.

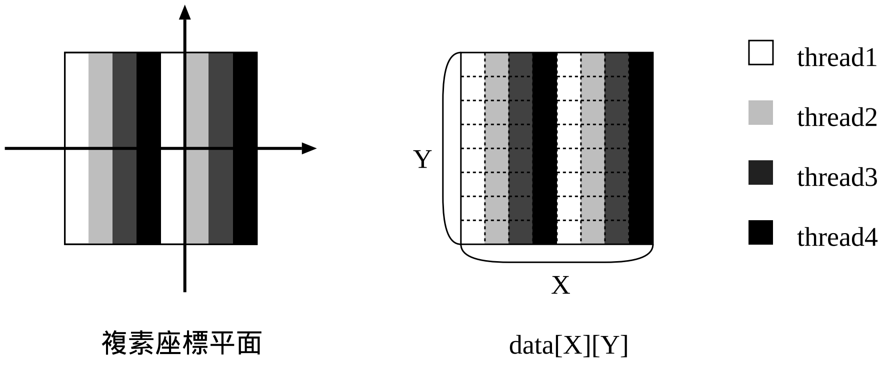


図 21: マンデルブロのタスク分割

(4) 並列効果

複素座標平面の分割数を変えて 3 通りの方法で計算を行った. 分割数が少ない場合, 計算量に対しスレッド操作のためのオーバヘッドに占める時間が高くなるため, 速度向上はあまり得られない. 分割数を多くすると, 計算を行う座標点数の増加に伴い, 計算量が増加するため, プロセッサ台数に比例した理想値に近い結果が得られる.

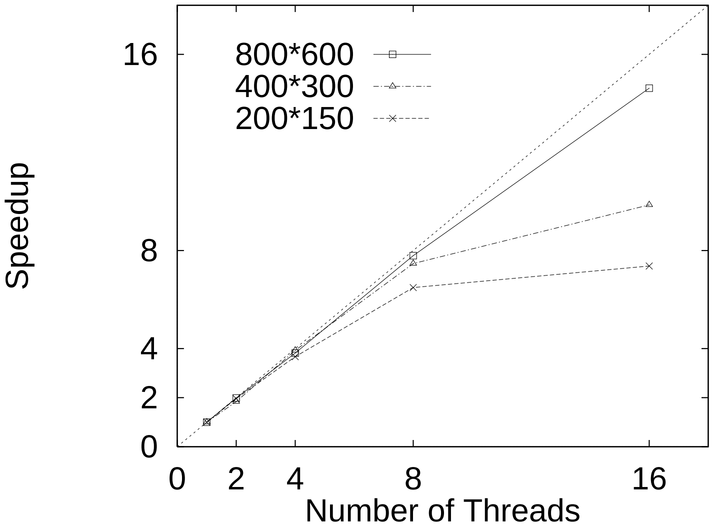


図 22: マンデルブロの並列効果

4.3.2 KMP 法

(1) 問題の定義

KMP 法は, あらかじめ検索文字 (PAT) の構造を解析して、一致する見込みのない位置は飛ばして次の検索を再開することにより, 迅速に文字列照合を行う方法である.

TEXT, PAT は共にファイルにあらかじめ格納しておく. 本プログラムで計算終了後, 何個目の PAT が何文字目にあるかを出力する. TEXT を 10 万文字, PAT を 40 個用意して計算を行った.

(2) 並列計算

TEXT を使用するスレッド数と同数のブロックに分割する. 各スレッドは自分に与えられた文字列範囲の中で検索を行う. 隣接した 2 つのブロックの境界をはさんで PAT が存在する場合があるので, 境界付近の一定領域は重複して 2 つのスレッドに割り振られる. PAT 情報については, 全 PAT データのコピー が全スレッドに割り振られる. 検索結果はコモン配列に PAT の発見位置と PAT 番号を登録する.

(3) インデックス

表 5: KMP 法のインデックス

応用	文字列照合
仕様	TEXT をスレッド数で分割したサブ TEXT 内で,PAT と一致する文字列を 1 文字ずつ順次照合, 検索を行う.
アルゴリズム	プロセッサファーム
終了条件	なし
データ構造	ソースデータ: 文字型 1 次元配列 TEXT[100000] : 文字型 2 次元配列 PAT[40][ ] 結果データ : 構造体 2 次元配列 searchbox[スレッド数][100]
タスク分割	(TEXT , 重複ブロック , ブロック) (PAT , , コピー) (searchbox , 列 , サイクリック)
トポロジ	マスター &ワーカー
インタラクション	なし
並列化手法	パラレルリージョン
実行構造	C^t

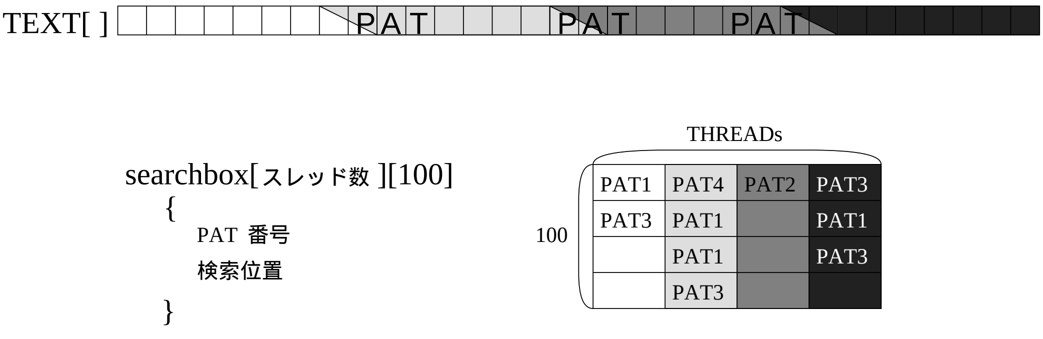


図 23: KMP 法のタスク分割

各スレッドは, マスタースレッドから割り当てられた分割タスクを受け取り, その結果を再びをマスターに返す構造からトポロジはマスター &ワーカーとなる. また, 並列計算上プロセッサ間の依存関係はなく, 同期に関しては全スレッドの計算終了をマスタースレッドが待つための同期以外必要としない. つまり, マンデルブロと全く同じ構造を持つことができ, 並列化手法も同様にパラレルリージョン型となる.

(4) 並列効果

TEXT のサイズを大きくするに連れて高い並列性が得られた. 図 24の結果は PAT 数が 40 個, TEXT が 10 万文字の場合の並列効果である. 各スレッドが担当する計算量が非常に大きく, 負荷

も均等であるので, 非常に理想値に近い結果が得られた. 結果の出力先をコモン配列とすることによりインタラクション をなくしたことも結果に大きく影響していると考えられる.

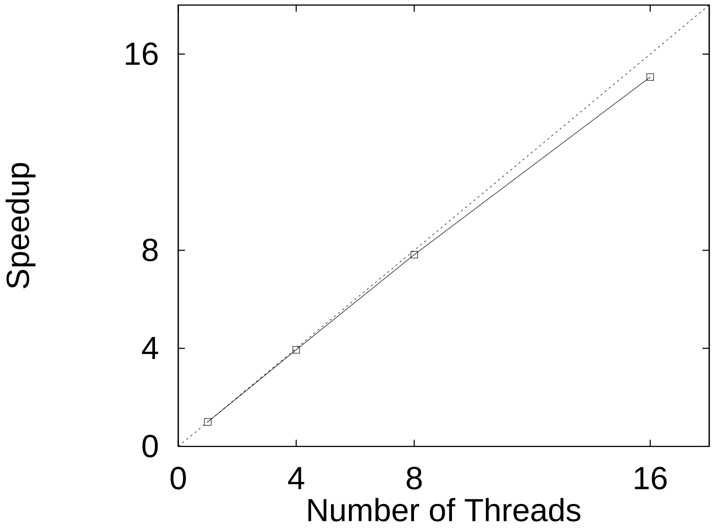


図 24: KMP 法の並列効果

4.4 プロセスネットワーク

4.4.1 LU 分解法

(1) 問題の定義

LU 分解は正方行列 A を A = LU のように，下三角行列 L と，上三角行列 U の積の形に分解することである．連立 1 次方程式 Ax = b を解く時，A = LU であるから LUx = b となる．ここで Ux = y とおくと Ly = b となり，まず Ly = b を解き，その解 y を定数項に移して Ux = y を解くことで連立 1 次方程式の解を得る．ちなみに正方行列を LU に分解する時，(i, j) 成分は

$$U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} l_{ik} U_{ki}) / l_{ii} \qquad (i < j)$$

$$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} l_{ik} U_{ki} \qquad (i >= j)$$

の計算で L，U の値が得られる．

(2) 並列計算

単位計算は, 1 つの行列要素についての計算である. 図 25の白丸に位置する要素についての計算では, 他に黒丸で示す 7 つの要素の値が必要である. さらに, これら 7 つの要素は計算済みでなければならない. そこで, 行列要素の各スレッドへの割り振りは, 図 26に示す様に, 列をサイクリックに割り振る. 計算ステップは, 丸で囲まれた数字の順番に進める. この方法により条件を満たすことができる.

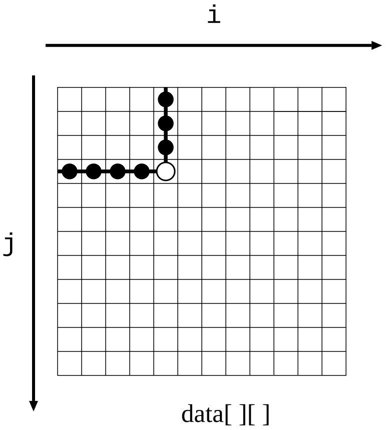


図 25: LU 分解法のソース データ

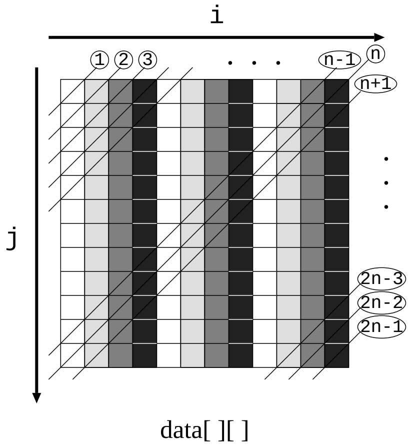


図 26: LU 分解法のタスク分割

(3) インデックス

表 6: LU 分解法のインデックス

応用	数値計算		
仕様	$U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} l_{ik}U_{ki})/l_{ii} \quad (i < j)$ と		
	$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} l_{ik}U_{ki} \quad (i >= j)$ を使い，		
	正方行列を L と U に分解		
アルゴリズム	プロセスネットワーク		
終了条件	ステップ $= 2n - 1$		
データ構造	ソースデータ: 整数型 2 次元配列 data[ ][ ](正方行列)		
	結果データ　: 整数型 2 次元配列 data[ ][ ](正方行列)		
タスク分割	(data[ ][ ](ソース データ)	,	, コピー)
	(data[ ][ ](結果データ)	, 一列	, サイクリック)
トポロジ	パイプ		
インタラクション	シグナル·ウエイト		
並列化手法	スレッドライブラリ		
実行構造	top_thread	$Fix\{C, I^{p(lower\_thread)}\}$	
	inter_thread	$Fix\{I^{p(upper\_thread)}, C, I^{p(lower\_thread)}\}$	
	tail_thread	$Fix\{I^{p(upper\_thread)}, C\}$	

図 26の様にタスク分割, 処理手順を決定すると, あるスレッド上で計算された結果データは, 別のスレッド上でソースデータとして利用される. このデータの流れは, 全てのスレッドをつなぐパイプを形成している. このようなアルゴリズムの特徴により, プロセスネットワークのアルゴリズムを用いることができる. 並列構造は図 12に示す構造が適応される.

パイプラインの各ステージを担当するスレッドの実行構造は, 次の処理の 2n-1 ステップ回の繰り返しである.

最上位ステージ (**top_thread**)

1. 計算.
2. 下位ステージと同期を取り, 結果データを渡す.

最上位と最下位の間のステージ (**inter_thread**)

1. 上位ステージと同期を取り, ソースデータを受け取る.
2. 受け取ったソースデータをもとに計算を行う.
3. 下位ステージと同期を取り, 結果データを渡す.

最下位ステージ (**tail_thread**)

1. 上位ステージと同期を取り, ソースデータを受け取る.
2. 受け取ったソースデータをもとに計算を行う.

(4) 並列効果

640 × 640 の行列と 1280×1280 行列を用いて計算を行った場合の並列効果を図 27に示す．行列のサイズを拡大すると共に計算量も増え，速度向上が見られる．1280×1280 の行列で 16 台使用時，約 13.5 倍の速度向上が得られた．ステージを流れるデータの量が多いため, 全ステージが同時に計算を行う割合が高い. また, 1 ステップで行われる計算量が比較的大きいことから, 通信によるオーバヘッドの影響が小さく, プロセスネットワークに適した問題であると考えられる.

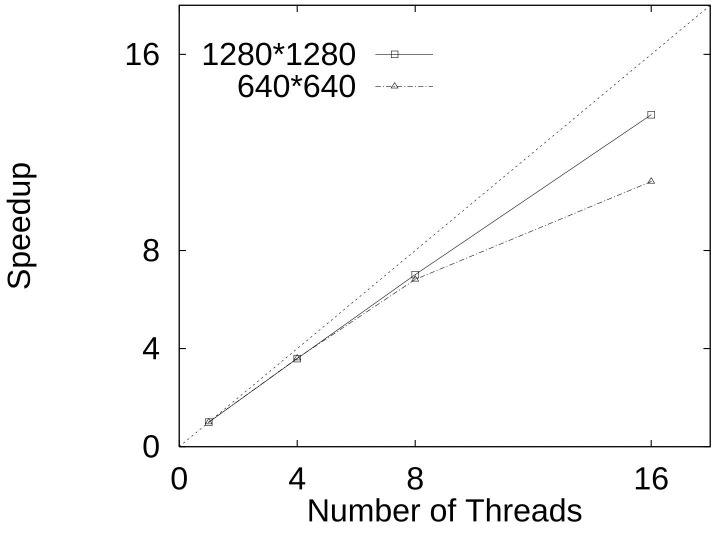


図 27: LU 分解法の並列効果

4.4.2 バブルソート

(1) 問題の定義

n 個のデータを昇順にソートする場合，まず最初に n 番目のデータと n-1 番目のデータを比較し，前者が後者より小さければ入れ換えを行い，そうでなければ何もしない．次は n-1 番目のデータと n-2 番目のデータを比較し... と続き, 最後に, 2 番目と 1 番目の比較·交換を行う．このように, 最小の値を持つデータ (バブル) が, 泡のように移動していく. 次のステップは前と同じように n 番目と n-1 番目のデータを比較を行い, 前回入れ換えを行った最後の位置の直前，つまり前回の最後の入れ換えが, K 番目と K-1 番目の間で行われていたら, 次回は K+1 番目と K 番目のところまで検索を繰り返す. これを, 入れ換えが起こらなくなるまで繰り返す (図 28参照）.

(2) 並列計算

n 個のデータをいくつかのブロックに分割し, それを各スレッドに割り振る．各スレッドはそれぞれに与えられたブロックの中で比較·交換を行う．はじめに, n-1 番目から, 1 番目まで比較·交換が行われると, 1 番目の位置まで移動されたデータは最小のデータと決定されるので, 次回からは 1 つ少ない n-1 個のデータに対して行う. このように比較·交換を行うデータの数が 1 つずつ減少していくため, 各スレッドが処理するブロック範囲も減少させる必要がある. このため, つぎのように各スレッドが担当するブロック範囲を減少させる. まず, バブルの進行方向の一番前のブロック範囲を 1 つ減らす. 次はその 1 つ手前のブロック範囲を 1 つ減らす. 次はその 1 つ手前のブロックの …と 1 番後ろのブロックまでいったら, 再び, 1 番前のブロック範囲を 1 つ減らす. 以降, この繰り返しを続ける.

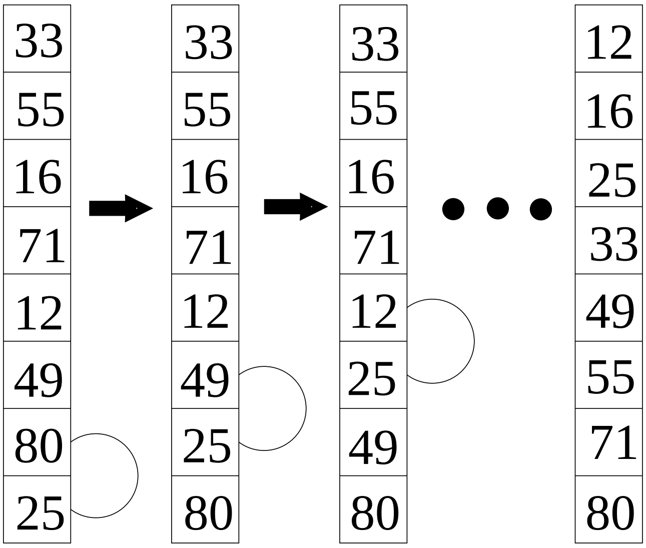


図 28: バブルソート

(3) インデックス

表 7: バブルソート のインデックス

応用	ソーティング	
仕様	バブル (局所最小データ) が上がってくるのを待ち, 自分の範囲内で新たなバブルを順番に検索	
アルゴリズム	プロセスネットワーク	
終了条件	ソースデータ = 0	
データ構造	ソースデータ: 整数型一次元配列 data[ ]$_{(ソートされるデータ)}$ 結果データ : data[ ]$_{(ソートされたデータ)}$	
タスク分割	(data[ ][ ] , 複数ブロック , ブロック)	
トポロジ	パイプ	
インタラクション	シグナル· ウエイト	
並列化手法	スレッドライブラリ	
実行構造	top_thread	$Fix\{C, I^{p(lower\_thread)}\}$
	inter_thread	$Fix\{I^{p(upper\_thread)}, C, I^{p(lower\_thread)}\}$
	tail_thread	$Fix\{I^{p(upper\_thread)}, C\}$

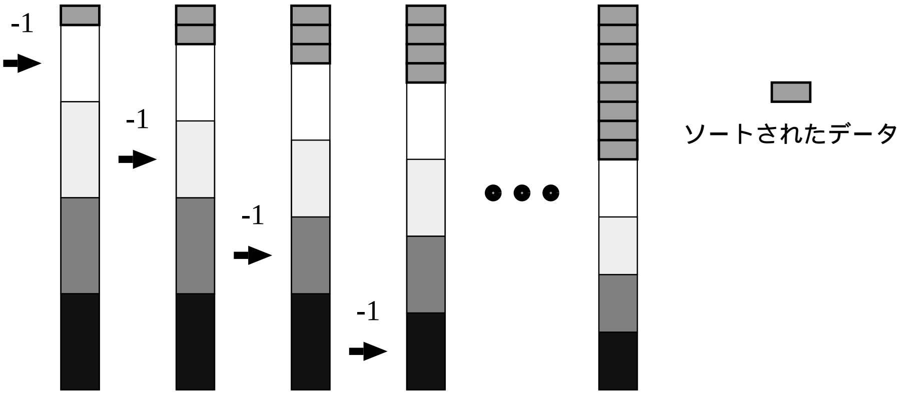


図 29: バブルソート のタスク分割

各スレッドは, 自分に割り当てられたブロック範囲の中で, 比較 ·交換を行い, 逐次新たなバブルとなるデータをを検索する. そして, 最終的なバブルを次のスレッドに渡す. バブルを受け取ったスレッドは, これをもとに自分に割り当てられたブロック内で同様に検索を行い, 次のスレッドへバブルを送る. このように, バブル (データ) の流れと共にスレッドは処理を行う. この際, データの流れが生むスレッドのトポロジは, LU 分解法と同様にパイプとなる. よって, アルゴリズムとしてプロセスネットワークが適応できる. 並列構造も LU 分解法の場合と同じである.

(4) 並列効果

25600 個と 51200 個のデータで計算を行った．その並列効果を図 30に示す．データを増やすと計算量も増え，速度向上も得られた．データ数が 51200 個で 16 台の時，約 9.8 倍の速度向上が得られた．

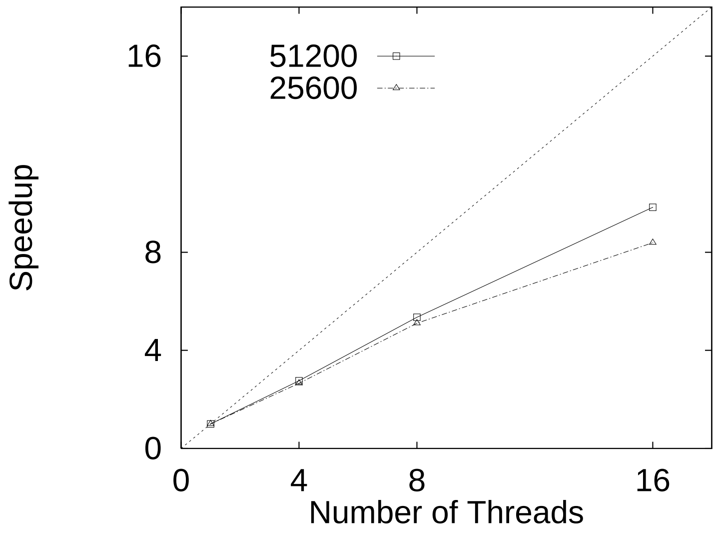


図 30: バブルソート の並列効果

4.5 繰り返し変換

4.5.1 DKA 法

(1) 問題の定義

実根を持つ n 次の多項式

$$f(x) = x^n + \alpha_1 x^{n-1} + \dots + \alpha_{n-1}x + \alpha_n$$

の n 個の解を求めるプログラムを, DKA 法 (Duran,Kerner,Aberth による) でプログラムする.

この方法は n 個の解 $X_1, X_2, \dots, X_n$ を同時に求めるものである. アルゴリズムは反復法で, X_i の k 番目の近似値を $X_i^{(k)}(i = 1, 2, \dots, n)$ とし,

$$X_i^{(k+1)} = X_i^{(k)} - \frac{f(X_i^{(k)})}{\prod_{j=1, j\neq i}^{n} (X_i^{(k)} - X_j^{(k)})}$$

により逐次近似値を求める. これを, X_i^{k+1} と X_i^k との差が小さくなるまで続ける.

(2) 並列計算

1. $X_1, X_2, \dots, X_n$ の初期値 $X_1^{(0)}, X_2^{(0)}, \dots, X_n^{(0)}$ を求める.
2. $X_1^{(k)}, X_2^{(k)}, \dots, X_n^{(k)}$ それぞれに対して計算を行い, $X_1^{(k+1)}, X_2^{(k+1)}, \dots, X_n^{(k+1)}$ を求める.
3. $X_1^{(k)}, X_2^{(k)}, \dots, X_n^{(k)}$ と $X_1^{(k+1)}, X_2^{(k+1)}, \dots, X_n^{(k+1)}$ との比較をそれぞれ行い, 差がある一定の値より小さければ 4 へ, 大きければ 2 へ戻る.

   2,3 の繰り返しにより, $X_1^{(k)}, X_2^{(k)}, \dots, X_n^{(k)}$ はそれぞれある一定の値に収束する.
4. このときの $X_1^{(k)}, X_2^{(k)}, \dots, X_n^{(k)}$ の値を最終的な $X_1, X_2, \dots, X_n$ の値とみなし, プログラムを終了する.

2, の中で, $X_1^{(k)}, X_2^{(k)}, \ldots, X_n^{(k)}$ に対しての計算はそれぞれ独立に行えるので, この計算をスレッドに分割し, 並列に行う. 2 を繰り返し変換のためのワンステップとし, ステップ毎に同期をとることにより, ステップ毎の $X_i^{(k)}$ の値の一貫性を保つ.

(3) インデックス

表 8: DKA 法のインデックス

応用	数値計算
仕様	$X_i^{(k+1)} = X_i^{(k)} - \frac{f(X_i^{(k)})}{\prod_{j=1,j\neq i}^{n} (X_i^{(k)} - X_j^{(k)})}$
アルゴリズム	繰り返し変換
終了条件	誤差 $(X_i^{(k+1)} - X_i^{(i)}) < 10^{-6}$ 又は 変換ステップ数 $= 5000$
データ構造	ソースデータ: 実数型 1 次元配列 X[n] (n: 方程式の次数) 実数型 1 次元配列 a[n] (方程式の係数) 結果データ : X[n]
タスク分割	(X[n]$_{(ソースデータ)}$, , コピー) (X[n]$_{(結果データ)}$, 1 要素 , サイクリック) (a[n] , , コピー)
トポロジ	完全照合
インタラクション	バリヤ
並列化手法	パラレルリージョン
実行構造	$Cond\{I^t_{Barrier(A/B)}\{C^t\}, C^{p(master)}\}$

図 31: DKA 法のタスク分割

各スレッドは割り振られた X_i に対して変換処理を行う. その際, 全ての $X(X_1 \sim X_n)$ の値を必要とする. つまり, 他の全スレッドに対してデータ参照を行う. よってトポロジは完全照合となる. データ参照は, バリヤ同期を用いて, そのデータに有効な値が書き込まれた後に行う. 各スレッドに割り振られるタスクは異なるが, 同一の処理ができる. 並列構造は, 図 13に示す構造を持つ. 実行構造は, 次の 1∼4 の処理で構成されるステップを, 終了条件を満たすまで (不定回) 繰り返すことを示す.

1. 全スレッドが, バリヤ (A 又は B) にチェックインする.

2. 全スレッドが, X_iに対して同時に変換処理を行う.

3. 全スレッドが, バリヤ (A 又は B) をチェックアウトする.

4. マスタースレッドは, 変換された X_iが条件を満たしているかを評価する.

(4) 並列効果

12 次, 16 次, 32 次と, 3 通りの方程式で計算を行った. 図 32に示す並列効果から分かるように, 次数の高い方程式, つまり 1 台のプロセッサが多くの解ついての計算を行う場合ほど高い並列性が得られている. 32 次の場合, 16 台のプロセッサを用いて約 10 倍の速度向上を得ることができた.

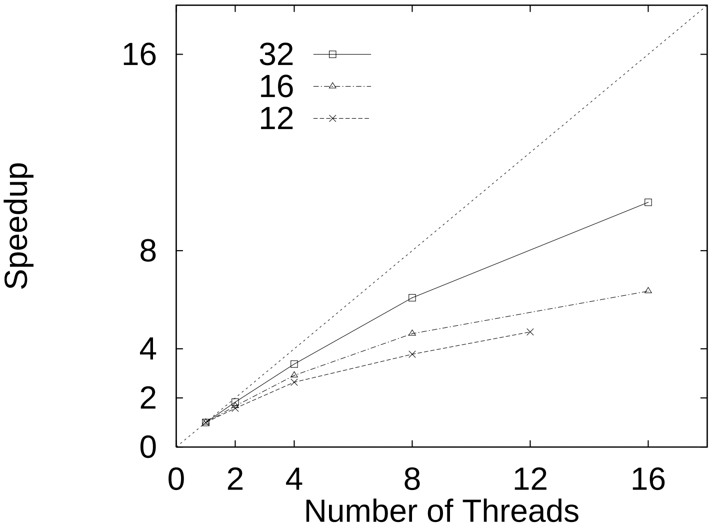


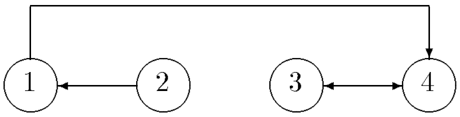
図 32: DKA 法の並列効果

4.5.2 ワーシャルアルゴリズム

(1) 問題の定義

グラフ問題の中でよく表れる, 有向グラフの推移閉包の配置についの問題である. 推移閉包とは他のバーテックスにたどり着けないバーテックスのことをいう. 図 33の 1 番のバーテックスは 3 番, 4 番のバーテクッスに到達することができる. グラフ理論では, そのようなグラフと推移閉包は論理変数を値に持つ T と A の正方行列で表される. A を隣接行列, T を推移閉包と呼ぶ. 行列 A の (j, i) の値 1 はノード j からノード i に直接つながるエッジを示す. 推移閉包行列 T の場合, 変数 1 は直接の結合でなくても, 到達することが可能な (j, i) を示している. ワーシャルアルゴリズムは行列 A を行列 T に変換する.

アルゴリズムの基本的な考えは, ノード j からノード i へのエッジが存在すれば, j は i が到達することのできる全てのノードへ到達することができるという考えである. このルールを i と j の全ての組合せに適応するとワーシャルアルゴリズムとなる.



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, T = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

図 33: ワーシャルアルゴリズム

(2) 並列計算

図 35のように, 行列 A を 1 行ずつサイクリックに各スレッドに割り振る. 計算の流れは, 1 ステップで 1 列についての計算を行い, 図 34のように行方向にステップは流れる. 処理は次のように進められる.

1. ステップ 1 でスレッド 1 とスレッド 3 は値 1 を 1 つずつ得る.
2. スレッド 1 とスレッド 3 はそれぞれ,1 行目 (ステップ値と同値の行番号) と 4 行目,1 行目 (ステップ値同値の行番号) と 3 行目の値の論理和を求め, その結果を,4 行目, 3 行目に格納する.
3. ステップ 2 に移る.

これをステップ 6 まで続ける.

ステップ

スレッド	1	2	3	4	5	6
①	0	1	0	0	0	0
②	0	0	1	0	1	0
③	1	0	1	0	0	0
①	1	1	1	0	0	0
②	0	0	1	1	1	0
③	0	1	0	0	1	0

図 34: ワーシャルアルゴリズムの並列計算

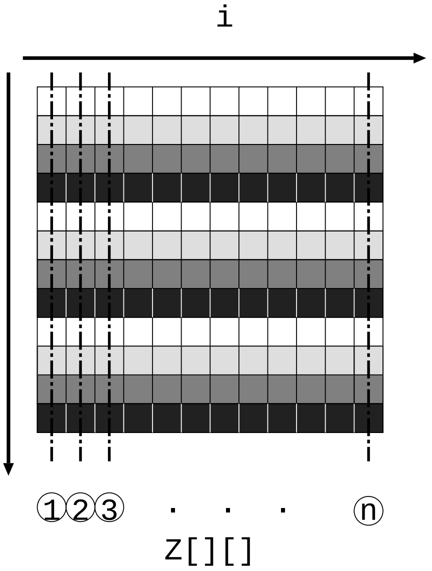


図 35: ワーシャルアルゴリズムのタスク分割

(3) インデックス

表 9: ワーシャルアルゴリズムのインデックス

応用	グラフ問題
仕様	2 次元ベクトルの列同士の論理和
アルゴリズム	繰り返し変換
終了条件	ステップ数 = ノード数
データ構造	ソースデータ: 整数型 2 次元配列 Z[ノード数][ノード数]
	結果データ : Z[ノード数][ノード数]
タスク分割	($Z_{(ソースデータ)}$, , コピー)
	($Z_{(結果データ)}$, 1 行 , サイクリック)
トポロジ	完全照合
インタラクション	バリヤ
並列化手法	パラレルリージョン
実行構造	$Fix\{I^t_{Barrier(A/B)}\{C^t\}\}$

ワーシャルアルゴリズムでは計算負荷の偏りが生じる．例えば，図 34の中で，ステップ 1 では，スレッド 3 と 2 は計算を行うが，スレッド 2 は何もしない．同様にステップ 4 では，スレッド 2 以外は計算をしない．この例ではノード数が少ないという理由が挙げられる．しかし，十分にノード数が多い場合でも一部のスレッドに処理が頼る可能性は十分あり得る．そのため，サイクリックによる分割・分散法を用いることによりタスクをランダムに割り振り，負荷の均衡を実現している．

各スレッドは，自分に割り振られた行と他のスレッドに割り振られた行との論理和を計算する．他のスレッドとは予め決められているのではなく，不明である．つまり他の全スレッドに対してデータ参照を行う可能性がある．よって，トポロジは完全照合となる．

データ参照は，バリヤ同期を用いて，その行に有効な値が書き込まれた後に行う．並列構造は，図 13に示す構造を持つ．実行構造は，次の 1∼3 の処理で構成されるステップを，終了条件を満たすまで (固定回) 繰り返すことを示す．

1. 全スレッドが，バリヤ (A 又は B) にチェックインする．
2. 全スレッドが，同時に変換処理を行う．
3. 全スレッドが，バリヤ (A 又は B) をチェックアウトする．

(4) 並列効果

ノード数 2000，各ノードからの手の数が 16 のグラフを対象として計算を行った結果，プロセッサ 16 台使用時，約 12.0 倍の速度向上が得られた．マスターによる結果評価のための計算がないため，高い並列性を得ることができた．

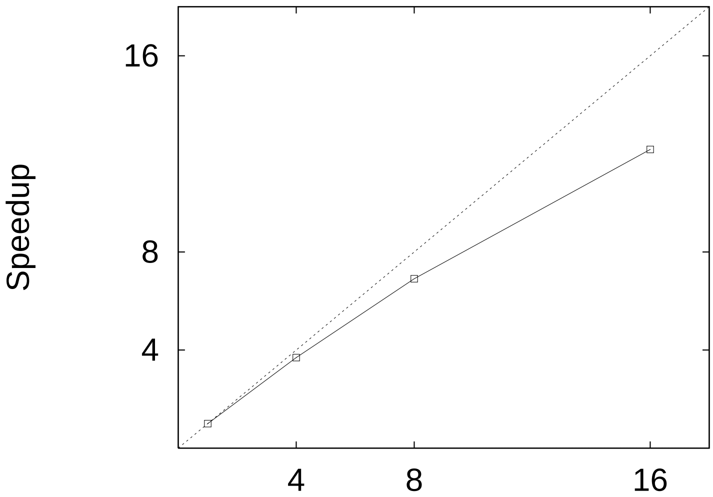


図 36: ワーシャルアルゴリズムの並列効果

4.6 スケルトン

4.6.1 分割統治法

クイックソート

```
main
  {
    スレッド数の入力;    要素列の読み込み;    ミューテックス初期化;
    チーム ID = チーム作成 ( スレッド数 );    アクティブスレッド数 = 1;
    パラレルリージョン起動 ( チーム ID, thread_quick );
    ソート結果を出力
  }

thread_quick
  {
    スレッド番号取得;
    while(1)
      {
        ( ソート開始命令 ) 又は ( アクティブスレッド数 == 0 ) のシグナル待ち;
        if ( アクティブスレッド数 == 0 ) break;
        quick( 部分要素列の始端: L, 部分要素列の終端: R );
        ミューテックスロック;
        アクティブスレッド数−−;      自分を非アクティブ状態とする.
        ミューテックスアンロック;
      }
  }
quick( L, R )
  {
    単位計算 ( 受け取った部分要素列をさらに 2 つの部分要素列に分割 );
    if ( 大きい方の部分要素列について )
      {
        if ( データ量 > 全体の 1% )
          {
            ミューテックスロック;
            if ( アクティブスレッド数 < 全スレッド数 )
              {
                アクティブスレッド数++;
                空きスレッドの番号を検索;
                タスク割り当て (部分要素列の始端: i と 終端:R )
                検索したスレッドをアクティブ状態にする;
                ミューテックスアンロック;
              }
            else { ミューテックスアンロック;    quick(i,R); }
          }
        else quick(i,R);
      }
    if ( 小さい方の部分要素列について ) quick(L,j);
  }
```

積分計算

```
main
  {
    プロセッサ数の入力;   ミュ-テックス初期化;
    アクティブスレッド数 = 1;   累積スレッド数 = 1;
    タスク割り当て ( 積分区間の左端値: L=-100.0, 右端値: R=100.0 );
    thread_start( 0 );
    アクティブスレッド数 = 0 となるまで待つ;
    $ans$[0] $\sim$ $ans$[プロセッサ数 $-$ 1] の値の合計を求め, 結果出力;
  }

thread_start( スレッド番号 )
  {
    estimete(L, R, スレッド番号);
    自分を非アクティブ状態にする.
    ミューテックスロック ;   アクティブスレッド数 $--$;   ミューテックスアンロック;
  }

estimate(積分区間の左端値:L, 積分区間の右端値:R, スレッド番号)
  {
    単位計算 ( 面積 $S1, S2$ を求める );
    if ( 誤差 ( $S1 - S2$ ) $> \epsilon$ ) {
        ミューテックスロック
        if ( アクティブスレッド数 $<$ プロセッサ数 )
            if ( 累積スレッド数 $<$ プロセッサ数 ) {
                累積スレッド数++;   アクティブスレッド数++;
                ミューテックスアンロック
                タスク割り当て ( L, M );
                スレッド生成 ( thread_start, 累積スレッド数 $-$ 1 );
              }
            else {
                if( ( 累積スレッド数 $<$ 96 ) && ( 誤差 $> \epsilon \times 100000$) ) {
                    累積スレッド数++;   アクティブスレッド数++;
                    アイドルプロセッサの検索 ;
                    ミューテックスアンロック
                    タスク割り当て ( L, M );
                    スレッド生成 ( thread_start, 検索されたプロセッサ番号);
                    スレッド移動;
                  }
                else estimate(L,M, スレッド番号);
            }
          else estimate(L, M, スレッド番号);
        estimate(M, R, スレッド番号);
      }
    else S2 の値を $ans$[スレッド番号] に加える;
  }
```

4.6.2 プロセッサファーム

マンデルブロ

```
main
  {
    スレッド数の入力;
    チームID = チーム作成 (スレッド数);
    パラレルリージョン起動 ( チームID, work);

    data[X][Y] をもとに結果出力
  }

work
  {
    スレッド番号取得

    for ( I = $X$座標最小値 + スレッド番号; I< $X$座標最大値; I+=スレッド数 )
      {
        for ( J = $Y$座標最小値 ; J< $Y$座標最大値 ; J++ )
          {
            ( I, J ) を複素座標値 ( i, j ) に変換;
            while ( ( | 計算結果 | $\leq$ 2.0 ) && (ループ回数 $<$ 200 ) )
              ( i, j ) をもとにマンデルブロの単位計算;
            計算結果を data[I][J] に格納
          }
      }
  }
```

KMP 法

```
main
  {
    スレッド数の入力;
    TEXT, PATs の初期化;
    KMP 計算の初期設定;

    チーム ID = チーム作成 (スレッド数);
    パラレルリージョン起動 ( チーム ID, work );

    searchbox[ ][ ] を出力;
  }

work
  {
    スレッド番号取得;

    サブ TEXT 長 = TEXT 長 / スレッド数;
    tmp   = TEXT 長 $mod$ スレッド数;
    if ( スレッド番号 < tmp ) サブ TEXT 長++;

    for (PAT 数)
      {
        for( k = 0; k $\leq$ (サブ$TEXT$長 + $PAT$長 − 1); k++ )
          {
            サブ TEXT の中で PAT 検索;
            if ( PAT 検索成功)
              {
                seachbox[スレッド番号][検索回数].PATno = PAT 番号;
                seachbox[スレッド番号][検索回数].location = 検索位置;
              }
          }
      }
  }
```

4.6.3 プロセスネットワーク

プロセスネットワークでは, 3.2.2 節で示したシグナル·ウエイトを用いて, ステージ間の同期を実現している. そこでは, 図 7でも示したように, 共有変数の値をもとにスレッドを制御する. 本研究では, プロセスネットワークの実現に用いられる, いくつかのプロセス制御用プログラムをマクロ化し, それらをヘッダファイル (process_network.h) にまとめた. ここでは, プロセスネットワークのスケレトンを示す前に, process_network.h の紹介を行う.

process_network.h にはパイプラインの各ステージを担当するスレッドに対し, 次のような関数が用意されている.

- Top Thread (パイプラインの最上位ステージを担当するスレッド) 用
  top_lower1(スレッド番号) ———— 下位スレッドとの同期
    · 下位スレッドが処理終了を宣言するまで待つ.
  top_lower2(スレッド番号) ———— 下位スレッドとの同期
    · 自分の処理終了を下位スレッドに対して宣言.
    · 下位スレッドがデータを読み出すまで待つ.

- Inter_Thread (最上位と最下位の間のステージを担当するスレッド) 用
  inter_upper1(スレッド番号) ———— 上位スレッドとの同期
    · 上位スレッドに対して, 処理終了を宣言
    · 上位スレッドの処理終了 (データ書き込み終了) の宣言を待つ.
  inter_upper2(スレッド番号) ———— 上位スレッドとの同期
    · 自分と上位スレッドに対して, 処理開始のサインを出す.
  inter_lower1(スレッド番号) ———— 下位スレッドとの同期
    · 下位スレッドが処理終了を宣言するまで待つ.
  inter_lower2(スレッド番号) ———— 下位スレッドとの同期
    · 自分の処理終了を下位スレッドに対して宣言.
    · 下位スレッドがデータを読み出すまで待つ.

- Tail_Thread (パイプラインの最下位ステージを担当するスレッド) 用
  tail_upper1(スレッド番号) ———— 上位スレッドとの同期
    · 上位スレッドに対して, 処理終了を宣言
    · 上位スレッドの処理終了 (データ書き込み終了) の宣言を待つ.
  tail_upper2(スレッド番号) ———— 上位スレッドとの同期
    · 自分と上位スレッドに対して, 処理開始のサインを出す.

また, スレッド制御用のフラグは, 隣接する 2 つのスレッドに 1 個ずつ与えられる. つまり, 4 ステージのパイプラインの場合, 全部で 6 個のフラグが与えられる. このように, フラグを (ステージ数 $-$ 1) $\times$ 2 個用意し, 値の初期化を行うのが,

allocate(スレッド数) ———— フラグの初期化

である. これは, Top_Thread 上で, 1 回だけ用いられる.

これらの関数は, 一般的にプログラム内で図 37のように配置される (Inter_Thread の場合).

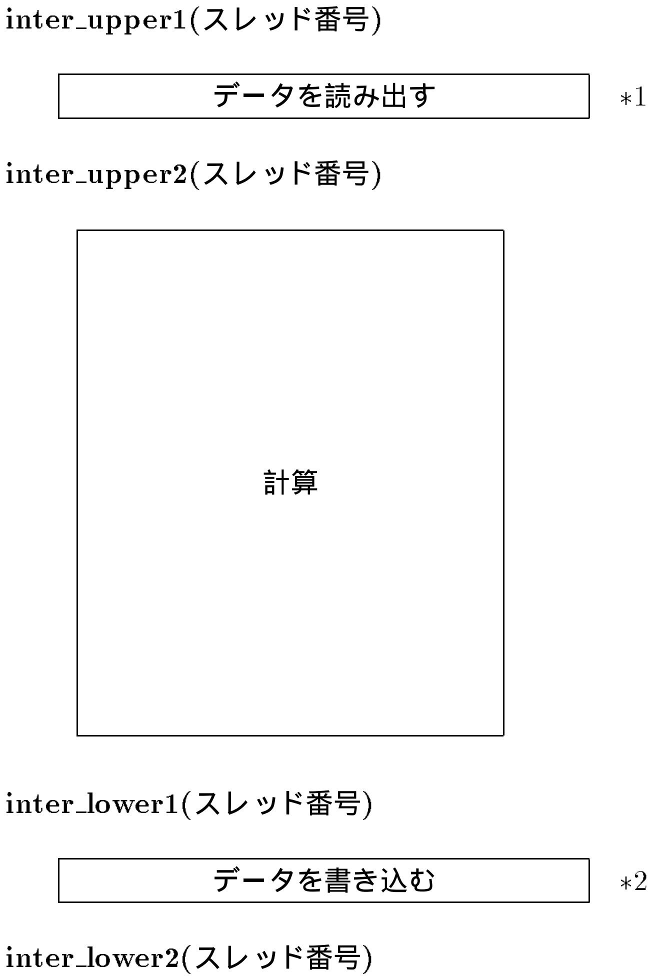


図 37: プロセスネットワー クのプログラミングスタイル (Inter_Thread)

プロセスネットワー クでは, スレッド間でデー タの受け渡しを行う. 2 つのスレッドが共有する変数には, ステップ毎に新たなデー タが書き込まれる. これをステップ毎に正確に参照できるように, *1,*2 のような配置を行う.

しかし, LU 分解法のように, 参照するデー タ位置がステー ジ毎に移動し, 参照の前には上位スレッドが有効な値を書き込むのを待ち, しかもそのデー タへの書き込みは 1 回だけで, 以降は参照のみ行われる場合, *1 や*2 は必要ない. inter_upper1 より, 上位スレッドの処理終了を確認するので, 計算の中で, デー タ読み出しができる. また, デー タ書き込みも計算と同時に行い, inter_lower2 で自分の処理終了を宣言することにより, 下位のスレッドは値を読み出す.

LU 分解法

```
main
  {
    行列 $A, B$の初期化;   スレッド数の入力;   フラグ変数の初期化;
    スレッド生成 ( top_thread, 0 )
    for (スレッド数 $-$ 2)  スレッド生成 ( inter_thread, スレッド番号 )
    tail_thread( スレッド数- 1 )
    行列 $L, U$を基に $Ax = B$ を解く;
  }
```

```
top_thread(スレッド番号)
  {
    ステップ番号 = 1;
    while ( ステップ番号 $\leq$ 2* n $-$ 1 )
      {
        タスク量, 始点の設定;
        LU 分解法の単位計算;
        下位スレッドとの同期;
        計算結果を下位スレッドに転送;
        下位スレッドとの同期;
        ステップ番号++;
      }
  }
```

```
inter_thread(スレッド番号)
  {
    ステップ番号 = 1 + スレッド番号;
    while ( ステップ番号 $\leq$ 2*n $-$ 1 )
      {
        上位スレッドとの同期;
        タスク量, 始点の設定;
        LU 分解法の単位計算;
        下位スレッドとの同期;
        計算結果を下位スレッドに転送;
        下位スレッドとの同期;
        ステップ番号++;
      }
  }
```

```
tail_thread(スレッド番号)
  {
    ステップ番号 = 1 + スレッド番号;
    while ( ステップ番号 $\leq$ 2*n $-$ 1 )
      {
        上位スレッドとの同期;
        タスク量, 始点の設定;
        LU 分解法の単位計算;
        ステップ番号++;
      }
  }
```

バブルソート

```
main
  {
    ソートデータの入力;   スレッド数の入力;   フラグ変数の初期化;
    for (スレッド数 $-$ 2)   スレッド生成 ( inter_thread, スレッド番号 )
    スレッド生成 ( tail_thread, スレッド数 $-$ 1 )
    top_thread( 0 );
    ソート結果の出力;
  }
```

```
top_thread(スレッド番号)
  {
    ソート範囲の左端, 右端の設定;
    while ( 左端値 $\le$ 右端値 )
      {
        while ( 右端 $\sim$ 左端 )
          値の大きいデータを左へ流す;
        左端の変更;
        下位スレッドとの同期;
      }
  }
```

```
inter_thread(スレッド番号)
  {
    ソート範囲の左端, 右端の設定;
    while ( 左端値 $\ge$ 0 )
      {
        上位スレッドとの同期;
        右端値とその左隣の値との比較, < 交換 >;
        上位スレッドとの同期;
        while ( 右端 $-$ 1 $\sim$ 左端 )
          値の大きいデータを左へ流す;
        左端, 右端の変更;
        下位スレッドとの同期;
      }
  }
```

```
tail_thread(スレッド番号)
  {
    ソート範囲の左端, 右端の設定;
    while ( 左端値 $\ge$ 0 )
      {
        上位スレッドとの同期;
        右端値とその左隣の値との比較, < 交換 >;
        上位スレッドとの同期;
        while ( 右端 $-$ 1 $\sim$ 左端 )
          値の大きいデータを左へ流す;
        左端, 右端の変更;
      }
  }
```

4.6.4 繰り返し変換

DKA法

```
main
  {
    スレッド数の入力;     方程式デ-タの入力;
    バリヤ初期化 ( バリヤ A );   バリヤ初期化 ( バリヤ B );
    チ-ム ID = チ-ム作成 ( スレッド数 );
    パラレルリ-ジョン起動 ( チ-ム ID, worker );
    結果出力
  }

worker
  {
    スレッド番号取得
    ステップ番号 = 0;
    $x_i$ の初期値計算;
    while(1)
      {  ステップ番号++;
        バリヤチェックイン ( &バリヤ A, スレッド番号 );
          object( スレッド番号 );
        バリヤチェックアウト ( &バリヤ A, スレッド番号 );
        if ( ( flag == 0 ) && ( ステップ番号 < 最大ステップ数 ) )
          if ( マスタ-スレッド )
            変換結果の評価 ( 全スレッドが終了条件を満たす結果を得ていれば flag=1 );
        else   break;

        ステップ番号++;
        バリヤチェックイン ( &バリヤ B , スレッド番号 );
          object( スレッド番号 );
        バリヤチェックアウト ( &バリヤ B , スレッド番号 );
        if ( ( flag == 0 ) && ( ステップ番号 < 最大ステップ数 ) )
          if ( マスタ-スレッド )
            変換結果の評価 ( 全スレッドが終了条件を満たす結果を得ていれば flag=1 );
        else   break;
      }
  }

object ( スレッド番号 )
  {
    for ( i=スレッド番号; i<n; i =+ スレッド数 )
      i についての単位計算;
    計算結果が, 終了条件を満たす:1/満たさない:0 を共有変数に書き込む;
  }
```

ワーシャルアルゴリズム

```
main
    {
        スレッド数の入力;
        隣接行列を定義;

        バリヤ初期化 ( バリヤ A );   バリヤ初期化 ( バリヤ B );
        チーム ID = チーム作成 ( スレッド数 );
        パラレルリージョン起動 ( チーム ID, worker );
        結果出力
    }

worker
    {
        スレッド番号取得;
        ステップ番号 = 0;
        while(1)
            {
                バリヤチェックイン ( &バリヤ A, スレッド番号 );
                    object( スレッド番号 , ステップ番号);
                 バリヤチェックアウト ( &バリヤ A, スレッド番号 );
                ステップ番号++;
                if ( ステップ番号 == ノード数 )  break;

                バリヤチェックイン ( &バリヤ B, スレッド番号 );
                    object( スレッド番号 , ステップ番号);
                 バリヤチェックアウト ( &バリヤ B, スレッド番号 );
                ステップ番号++;
                if ( ステップ番号 == ノード数 )  break;
            }
    }

object ( スレッド番号, ステップ番号 )
    {
        for ( i=スレッド番号; i<ノード数; i =+ スレッド数 )
            単位計算 ( 2 次元配列 $A[][]$ の (ステップ番号) 列についての計算 );
    }
```

5 類似事例の検索 ・修正法

5 章では, 問題解析, 事例の検索, 及び事例の修正法を具体例を交えて説明する. 例として, 画像から直線や円などの図形を抽出するための方法である Hough 変換を用いる.

5.1 Hough 変換

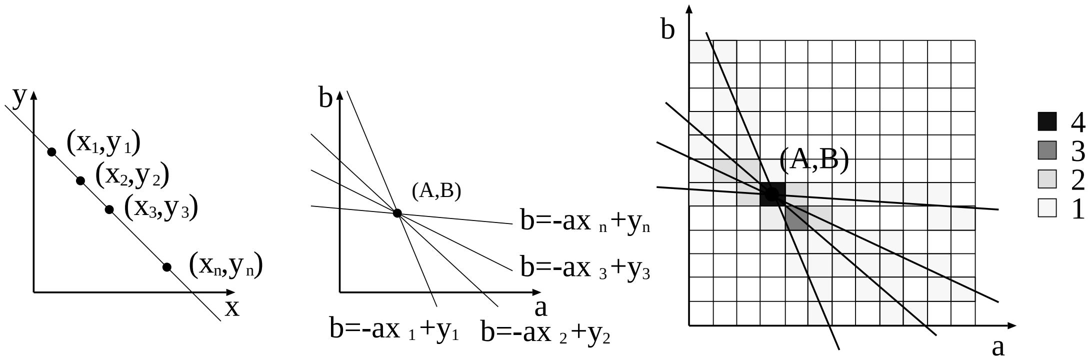
図 38(a) のように, 複数の点が 1 本の直線上に存在していると仮定すれば, この直線は

$$y_i = ax_i + b \quad (i = 1, 2, ..., n) \tag{1}$$

と表すことができる. 一方, これらの点を通る直線の式は, (a,b) を変数とみなして考えると, $a - b$ 平面上で

$$b = -x_i a + y_i \tag{2}$$

と表すことができる. これらの直線は (x_i, y_i) の値によって異なるが, (x_i, y_i) が全て同一直線上にあれば, 図 38(b) のように $a - b$ 平面上では 1 つの点 (A, B) で交わる. そこで, $a - b$ 平面を図 38(c) のように十分に細かいメッシュに区切る. 式 (2) で表される直線が, ある格子を通過すれば, その格子の計数を 1 増やす. これを式 (2) 全てについて行えば, (A, B) に対応する格子の計数値が最大値となり, 全ての点 (x_i, y_i) を通る直線は



(a) x-y平面 (b) a-b平面 (c) a-b平面をメッシュ分割

図 38: $x - y$平面から $a - b$ 平面への変換

$$B = -x_i A + y_i$$

と表すことができる.

$a - b$ 平面上への変換の場合, メッシュに区切る (a,b) の範囲を, 問題によってはかなり大きく取らなければならない場合が生じる. そこで, 直線を式 (2) の表現ではなく,

$$\rho = x\cos\theta + y\sin\theta \tag{3}$$

の式で, θ- ρ平面に変換する. これにより, メッシュに区切る範囲は, θ 方向は$-\pi$からπまで, ρ方向もある程度まで取れば十分な範囲となる.

5.2 問題解析

5.2.1 データ解析

タスクの分割要素となるものを発見するため, 計算対象を分割し, 単独に計算できる最小の計算単位 (単位要素) を明らかにする. 単位要素の計算を行うために必要とされるデータ (ソースデータ) と, その計算結果 (結果データ) の格納方法を明らかにした後, それらのデータ構造を決定する.

Hough 変換の問題では, 単位計算は $x-y$ 平面上のある 1 点 (x_i, y_i) を通る直線についての計算である. これに必要なソースデータと結果データは次の通りである.

ソースデータ : $x-y$平面上の点の座標値 (x_i, y_i)
結果データ : $data[Y][X]$

2 次元配列 $data[Y][X]$ には, $\theta-\rho$平面を $X \times Y$のメッシュに分割し, 式 (3) で表される線が格子 (p, q) を通過したら, それに対応する配列位置である $data[p][q]$ の値を 1 インクリメントすることにより, 各格子を通過する線の数を格納する.

図 39の場合, $\theta-\rho$平面を 6×6 のメッシュに分割している. そして, 線$\rho = x_a cos\theta + y_a cos\theta$が 9 つの格子を通過しているので, それに対応した配列位置 $data[1][1], data[1][2], data[1][3], data[1][4], data[2][0], data[2][1], data[2][4], data[2][5]$, data[3][0] の値を 1 インクリメントする.

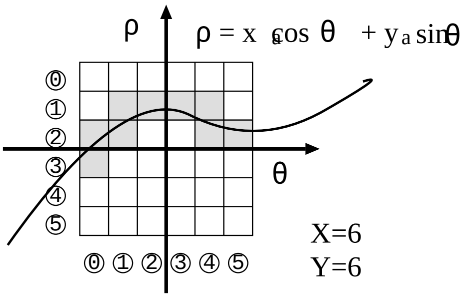


図 39: 結果データの格納方法

5.2.2 タスク分割

単位要素をプロセスに割り振ることにより, タスク分割が実現できる. 単位要素が割り振られることにより, その計算に必要なデータも必然的に割り振られることになる. よって, 計算対象, ソースデータ, 結果データが分割の対象となる. 該当するものに対して, (分割対象, 分割単位, 分散方法) を示す. 分割, そして各プロセスへの分散方法には, 3.2.1 節で示した,

- コピー
- ブロック
- サイクリック
- ブロック&サイクリック

の 4 つの方法のいずれかが適用される. コピーが選択されるのは, 全てのプロセスがそのデータを必要とする場合である. 参照のみ行う場合は特別なインタラクションは必要ないが, 書き込みを行う場合は, 排他制御により複数のプロセスが同時に書き込むのを防ぐ必要がある. また, 書き込まれたデータを参照する場合も, 書き込み終了を待つための同期が必要である. ブロックは, 連続

に配置されるデータを1塊にして割り振るため, 単純なプログラミングが可能で, 通信コストを下げるなどの利点がある. サイクリックは, 各単位要素の計算に負荷の偏りが存在する場合, 全体をランダムにプロセスへ割り振ることができるので負荷の均衡に役立つ.

コピーの場合は, 分割を行わず, 対象のデータ全体のコピーを全プロセスに割り振るため, 分割単位の欄は空白とする.

Hough 変換の場合, ソースデータとなる $x-y$平面座標値 $(x_i, y_i)_{i=0,1,2,....}$ は,

$$x_0\ y_0\ x_1\ y_1\ x_2\ y_2\ \ldots\ x_n\ y_n$$

の形でファイル $(points.txt)$ に収められているので, ファイルが分割対象となる. 各単位要素の計算負荷のばらつきは小さいと考えられるので, ブロックによる分割を適用する.

結果データは, このままの状態の場合, 複数のスレッドが同時に $data[Y][X]$ の値を更新する可能性がある. これを避けるため $data[Y][X]$ をスレッド数分用意する. つまり, $data[Y][X]$ の次元を1つ増やし, $data$[スレッド数]$[Y][X]$ とした. これにより, スレッド番号 i のスレッドは $data[i][Y][X]$ の領域に値を書き込む. 以上の分割方法を次のようにまとめ, インデックスに加える.

($points.txt$, ファイル長/スレッド数, ブロック)
($data$[スレッド数]$[Y][X]$, $data[Y][X]$, ブロック)

この2つのデータの分割の様子を図40に示す.

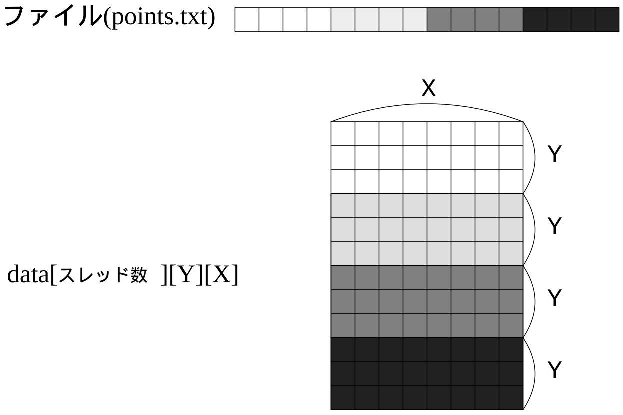


図40: Hough 変換のタスク分割

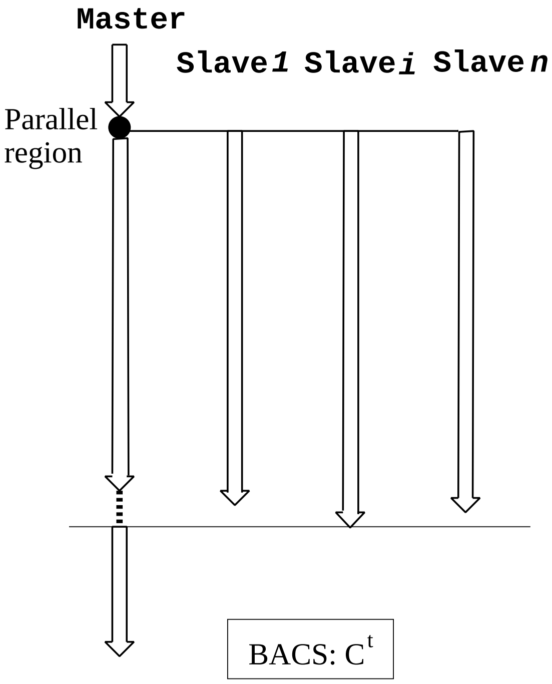


図41: Hough 変換の並列構造

5.2.3 アルゴリズムクラス

各プロセスは, ソースデータを他のプロセスから受け取り, それに対する結果データを別のプロセスへ送る. このようなデータの流れから生じるプロセストポロジを決定する. トポロジの形態とプロセス間のデータ依存性から, 問題に対する並列アルゴリズムを次のような条件で決定する.

分割統治法
ソースデータは親子関係にある親プロセスのソースデータと包含関係にある.

プロセッサファーム
ソースデータは他のプロセス上のソースデータ, 結果データと依存関係, 包含関係がなく独立である.

プロセスネットワーク
結果データは 1 列に並ぶプロセスの方向に流れ, パイプライン処理される.

繰り返し変換
結果データは他のプロセス上でソースデータとして広域的又は局所的に参照される場合がある.

また, アルゴリズムの終了条件が存在する場合は, 分割統治法または繰り返し変換が適用される.

Hough 変換では, 各プロセス上で単位計算に用いられるソースデータは, 他のプロセス上の単位計算で用いられるソースデータ, 結果データとの依存関係が無い. よって, 全プロセスは他のスレッドと独立に計算を行うことができる. また, 結果データはアルゴリズムの終了条件として作用せず, 全ての単位要素の計算が終了したらアルゴリズムは終了する. 以上の理由により, プロセッサファームが選択される.

5.2.4 並列構造

アルゴリズムを決定すると, システムはそのアルゴリズムに対する並列構造をいつくか提示する. 並列構造をもとにスレッドの動き, データの流れ, 同期の取り方などを視覚的に捕らえ, 問題に最も適するアルゴリズム実行構造を選択する.

並列構造には図による表現と共に BACS 表現が与えられているので, 並列構造決定後, これをインデックスに加える.

Hough 変換の場合, 図 5.2.2に示すパラレルリージョン型の並列構造を選択する. このような並列構造を選択した理由は次の 2 つである.

- 各スレッドは, 全てのタスクに対して同じ処理を行う.
- 並列計算上スレッド間の依存関係がないため, 特別なインタラクションを必要としない. 唯一必要なのは, 全スレッドの処理終了をマスタースレッドが待つための同期だけである.

並列構造と共に, BACS 表現 (C^t) が与えられるのでこれをインデックスに加える.

5.3 事例の検索·修正

問題解析を行った結果, 表 10に示すようなインデックスが得られた. これを基に, 類似事例の検索を行う. 既存の事例の中で, Hough 変換に最も類似した事例は KMP 法である.

KMP 法のインデックスを表 11に示す.

表 10: Hough 変換のインデックス

応用	画像処理
仕様	メッシュ(X×Y) に仕切った $\theta - \rho$ 平面上で, 直線 $\rho = x\cos\theta + y\sin\theta$ が通過する 格子の計数をインクリメント
アルゴリズム	プロセッサファーム
終了条件	なし
データ構造	ソースデータ: 画像平面座標値 (x_i, y_i) 結果データ : 整数型 3 次元配列 $data$[スレッド数][Y][X]
タスク分割	($points.txt$, ファイル長/スレッド数, ブロック) ($data$[スレッド数][Y][X], $data$[Y][X] , ブロック)
トポロジ	マスター&ワーカー
インタラクション	なし
並列化手法	パラレルリージョン
実行構造	C^t

表 11: KMP 法のインデックス (Hough 変換の類似事例)

応用	文字列照合
仕様	TEXT をスレッド数で分割したサブ TEXT 内で,PAT と一致する文字列を 1 文字ずつ順次照合, 検索を行う.
アルゴリズム	プロセッサファーム
終了条件	なし
データ構造	ソースデータ: 文字型 1 次元配列 TEXT[ ] : 文字型 2 次元配列 *PAT[PAT 数] 結果データ: 構造体 2 次元配列 searchbox[スレッド数][最大検索パット数]
タスク分割	TEXT, 重複ブロック, ブロック (PAT , , コピー) (searchbox , 列 , サイクリック)
トポロジ	マスター&ワーカー
インタラクション	なし
並列化手法	パラレルリージョン
実行構造	C^t

アルゴリズム, 終了条件, トポロジ, インタラクション, 並列化手法, 実行構造の項目内容が一致する. これにより同様の並列構造をもつプログラムが作成可能となる. しかし, タスク分割の点で若干異なる, Hough 変換も KMP 法も共にブロック分割を行うため, 共通した方法でタスク分割が実現できる. しかし, 分割対象となるデータの構造が両者で異なる. KMP 法はソースデータに, 文字列型 1 次元配列を用いるのに対して, Hough 変換はファイルを用いる. そのため, ファイルを扱うための特別な操作を用いたタスク分割が必要となり, このあたりが修正を要するところである.

KMP 法のプログラムレベルのスケレトンをリスト 1 に示す. 4.6 節で示したスケレトンは, 並列構造の理解を容易にし, アルゴリズムをユーザーがわかりやすいように表現したものである. 主に事例検索に用いられる. このスケレトンの中の, 並列構造に関する日本語の説明を, 実際のプログラムに置き換えたものをプログラムスケレトンと呼ぶ. これに直接肉付け, 修正を施すことにより目的の並列プログラムを作成する.

リスト 2 は Hough 変換の逐次プログラムである. データ入力, メッシュへの分割, 単位計算の計算式, 結果の出力などは逐次プログラムから取り出したものを直接スケレトンに肉付けできる.

リスト 3 にスケレトンを基に作成した Hough 変換の並列プログラムを示す. 行の先頭にアルファベット文字が置かれている箇所があるが, それらが示す意味は次の通りである.

N(New): 新たに作成した部分.

S(Skeleton): スケレトンの部分.

MP(Modify sequential Program): 逐次プログラムから取り出し, 修正した部分.

MS(Modify Skeleton): スケレトンに修正を加えた部分.

空白: 逐次プログラムから直接取り出したものをそのままの形で組み込んだ部分.

新たに作成された並列プログラムは, 大部分が逐次プログラムまたはスケレトンから構成される. 特に, 初期設定領域, 計算領域, 及び結果出力領域は逐次プログラムの再利用率が高く, 修正された部分が非常に少ない. また, 並列構造が共通する点から, スケレトンにはタスク分割以外の部分では殆ど修正が加えられていない. その中で肉付け及び修正がなされているのは, 各スレッドが持つ結果を 1 つにまとめる部分と, ファイルを扱う部分である. 特に, ファイル使用の影響が大きく現れているのは, タスク分割領域である. しかし, 分割·分散方法は共通するので, その部分に関してはスケレトンが直接利用されている.

このように, プログラマは類似事例と異なる部分のみプログラミングを行う. スレッド操作, 同期, 並列化手法などはスケレトンを, 初期設定領域, 計算領域, 結果出力領域の主な部分は逐次プログラムを利用することにより, 目的の並列プログラムを作成することができる. 今回の事例修正では, 分割対象が異なることから, ファイル処理関連の肉付け·修正が多く行われたが. この様な新たな構造を持つ並列プログラムは, スケレトンと共に新たな事例として事例ベース中に追加される. よって, 今後同様のタスク分割パターンを持つ問題が現れた場合, Hough 変換の事例が検索され, 少ない修正で新たな並列プログラムを作成することが可能となる.

リスト1 プログラムスケルトン (KMP 法)

```
#include <stdio.h>
#include <ksr/numarg.h>
#define PAT 40
#define TEX 100001
#define PT 16

struct  PATBOX {  int   location,  PATno;  }  searchbox[PT][100];
int  THREADs
char  text[TEX] ,  *pat[PAT];
void worker()

void main()
  {
    int   THREADs;
    printf( “input THREADs\ n” );   scanf( “%d”, &THREADs );
    TEXT, PATs の初期化;
    TEAM_ID = pr_create_team( THREADs );
    pr_pcall( TEAM_ID, work, copyargs( THREADs ) );
    searchbox[ ][ ] を出力;
  }

void work()
  {
    int   THREADno , offset , tmp, matchcount,   i,  k,   m;
    THREADno = pr_mid();
    offset = TEX / THREADs;
    tmp = TEX % THREADs;
    if ( THREADno < tmp ) offset++;
    for ( i = 0 ; i < PAT ; i++ )
      {
        matchcount = 1;
        m = strlen(pat[i]) - 1;
        for ( k = 0 ; k <= t + m ; k++ )
          {
            サブ TEXT の中で PAT 検索;
            if ( PAT 検索 O.K. )
              {
                searchbox[THREADno][matchcount].PATno = PAT 番号;
                searchbox[THREADno][matchcount].location = 検索位置;
                matchcount++;
              }
          }
      }
  }
```

リスト2 Hough 変換の逐次プログラム

```
#include<stdio.h>
#include<math.h>
#define X 1200                                              /* θ -ρ  平面を X−Y に分割                 */
#define Y 600

int yMAX;

int check( double , double , double , double * );
int change_to_function( double , double , double * , double * );

main()
  {
    unsigned short data[600][1201];
    int n;
    int xx, yy, result, count, points, noise, lowerrange, upperrange ;
    int xindex, yindex, a, b;
    double offsetx, offsety, side, all;
    double theta, rho, A, B;
    double x, y, destination;
    FILE *fin;

/* -------------------------------- 初期設定領域 (始め) -------------------------------- */

    printf( ”Input Range of Maximal Point (low)   (max) \n” );  /* 極大値となる格子のカウント数を              */
    scanf( ”%d %d”,&lowerrange, uperrange );                  /* (lowerrange<= <=upperrange) とする        */

    fin=fopen( ”points.txt”,”r” );
    fscanf( fin,”%d %d %d %d ”,&points, &noise, &xx, &yy );   /* 直線上の点の数 , ノイズの数,               */
                                                              /* |x 座標 | の最大値, |y 座標 | の最大値を入力*/
    n=points+noise;                                           /* x-y 平面上の全ての点の数                   */

    yMAX=xx+yy;
    side = (double)yMAX;

    offsetx=M_PI/(double)X;                                   /* θ 方向の分割幅                             */
    offsety=2.0*side/(double)Y;                               /* ρ 方向の分割幅                             */

/* -------------------------------- 初期設定領域 (終り) --------------------------------- */

/* ------------------------------ Hagh 変換の計算領域 (始め) ------------------------------ */
    count=0;
    while( count++<n )                                        /* 全ての ρ = a cos θ  + b sin θ  について  */
      {
        xindex=−1;
        fscanf( fin,”%d %d ”,&a, &b );
        while(xindex++ <= X)                                  /* 左側の列から調べる                         */
          {                                                   /* 式への代入を行い, */
            x=xindex*offsetx;
            result=check( (double)a, (double)b, x, &y );      /* ρ の値がその列のある格子を                 */
            if( result!=0 )                                   /* あるか調べる. 通れば                       */
              {
                destination = (y+side)/offsety;
                yindex=(int)destination;
                if( (destination − yindex) >= 0.5 ) yindex++;
                data[yindex][xindex]++;                       /* その時の (θ , ρ ) に対応する              */
              }                                               /* メッシュ格子のカウントをインク             */
          }                                                   /* リメントする                               */
      }
/* ------------------------------ Hagh 変換の計算領域 (終り) ------------------------------ */
    fclose(fin);
```

```
/* ―――――――――――――――――――――――――――― 結果出力領域 (始め) ―――――――――――――――――――――――――――― */
        count=0;
        for( yy=0; yy<Y; yy++ )                                     /* 全てのメッシュ格子点について                */
            for( xx=0; xx<X; xx++ )
                {
                    count++;
                    if( (lowerrange <= data[yy][xx]) && (data[yy][xx] <= upperrange) )/* 極大値に該当するものは    */
                        {
                            theta = xx*offsetx;
                            rho = yy*offsety-side;                  /* それを Y= Ax + B の形で                  */
                            result=change_to_function( rho, theta, &A, &B );  /* x-y 平面上に発見された直線     */
                            if( result==0 ) printf( ” X = %+5.4f \n”, B );    /* を出力する.                    */
                            else printf( ” Y = %+5.4fX %+5.4f\n”, A, B );
                        }
                }
/* ―――――――――――――――――――――――――――― 結果出力領域 (終り) ―――――――――――――――――――――――――――― */
      }

int check( double a, double b, double x, double *y )               /* 式 ρ = a cos θ + b cos θ に              */
    {                                                               /* 値 θ = x を与えた時の ρ の値を求め        */
        int i, j;                                                   /* ρ の値がマトリックスの範囲にあれば 1 を   */
        double result;                                              /* なければ 0 を返す                        */

        *y = a*cos(x) + b*sin(x);

        if( (−yMAX < *y) && (*y < yMAX) ) return 1;
        else return 0;
    }

int change_to_function( double rho, double theta, double *a, double *b )
    {                                                               /* ρ = a cos θ + b cos θ を                 */
        if( theta != 0 )                                            /* Y = aX + b の形に変換                    */
            {
                *a = −(cos(theta)/sin(theta));
                *b = rho/sin(theta);
                return 1;
            }
        else                                                        /* θ =0 の時には sin θ =0 となり            */
            {                                                       /* 上の式の cos θ /sin θ のところで         */
                *a = 0;                                             /* ダメなので                               */
                *b = rho;                                           /* Y = b の形に変換                         */
                return 0;
            }
    }
```

リスト3 Hough 変換の並列プログラム

```
S    #include <stdio.h>
     #include <math.h>
S    #include <ksr/numarg.h>
     #define X 1200                                        /* θ −ρ 平面を X−Y に分割              */
     #define Y 600
MP   int n, yMAX;                                          /* n: 点の数 , yMAX: ρ 値の絶対値最大値   */
MP   unsigned short data[24][600][1201];
S    int THREADs;                                          /* スレッド数                          */
MP   double offsetx, offsety, side, all;                   /* offsetx,offsety:θ , ρ 方向の分割幅   */
MP   FILE *fin;
S    void worker();
     int check( double , double , double , double * );
     int change_to_function( double , double , double * , double * );

S    main()
S      {
         int xx, yy, result, count, points, noise, lowerrange, upperrange;
S        int TEAM_ID;
         double theta, rho, A, B;
/* ------------------------------------------------------------------------- 初期設定領域 (始め) */
S        printf( ”input threads\n” );
S        scanf( ”%d”, &THREADs );
         printf( ”Input Range of Match Point (low) sin (max) backshashn”);
         scanf( ”%d %d”, &lowerrange, &upperrange );
         fin=fopen( ”points.txt”, ”r” );
         fscanf( fin, ”%d %d %d %d ”, &points, &noise, &xx, &yy );

         n=points+noise;
         yMAX=xx+yy;
         side = (double)yMAX;                              /* ρ 値 の絶対値最大値                   */
         offsetx=M_PI/(double)X;
         offsety=2.0*side/(double)Y;

/* ------------------------------------------------------ 初期設定領域 (終り), 並列ブロック (始まり) */

S        TEAM_ID = pr_create_team( THREADs );
S        pr_pcall( TEAM_ID, worker, copyargs() );
/* ------------------------------------------------------ 並列ブロック (終り), 結果出力領域 (始まり) */

N        for( yy=0; yy<Y; yy++ )                           /* 各スレッドがもつ結果                  */
N          for( xx=0; xx<=X; xx++ )                        /* (data[0][Y][X] ~ data[THREADS][Y][X]  */
N            for(i=1; i<THREADs; i++) data[0][yy][xx]+= data[i][yy][xx];   /* を data[0][Y][X] にまとめる */

         count=0;
         for(yy=0; yy<Y; yy++)
           for(xx=0; xx<X; xx++)
             {
               count++;
               if( (lowerrange <= data[0][yy][xx]) && (data[0][yy][xx] <= upperrange) )
                 {
                   theta = xx*offsetx;
                   rho = yy*offsety-side;
                   result=change_to_function( rho, theta, &A, &B );
                   if(result==0)   printf( ” X = %+5.4f \n”, B );
                   else            printf( ” Y = %+5.4fX %+5.4f\n”,A,B);
                 }
             }
/* -------------------------------------------------------------------------- 結果出力領域 (終り) */
       }
```

```
S    void worker()                                     /* pr_pcall により, この関数が呼ばれる     */
S      {                                               /* 各スレッドが並列に実行する関数          */
S        int THREADno;
S        int start, offset, tmp;
         int xx, yy, result, count, xindex, yindex, a, b ;
         double x, y, destination;
N        FILE *fin2;
S        THREADno = pr_mid();                          /* スレッド番号の取得                      */
/* ------------------------------------------------------------------------ タスク分割領域 (始め) */
N        if(THREADno != 0)   fin2 = fopen(”points.txt”,”r”);  /* マスタ- スレッド以外のスレッドはファイル */
N        else                fin2 = fin;               /*  ポインタを獲得していないので, それを得る. */

S        offset = n/THREADs;                           /* 各スレッドに割り当てる                  */
S        tmp = n%THREADs;                              /*     タスク量 (offset) を決定.           */
S        if( THREADno < tmp )
S          {
S            offset++;                                 /* 割り当てられたタスクのデ- タが収め      */
N            start = THREADno*offset*10 + 19;          /*   られているファイル位置 (start) を獲得. */
S          }
N        else start = ( tmp*(offset+1) + (THREADno -tmp)*offset )*10 + 19;
/* ------------------------------------------------------------------------- タスク分割領域 (終り) */
N        fseek( fin2, start, 0 );                      /* start にファイルポインタを移動する.     */
/* ------------------------------------------------------------------- Hagh 変換の計算領域 (始め) */
         count=0;
         while( count++<offset )
           {
             xindex=−1;
MP           fscanf( fin2, ”%d %d ”,&a, &b );
             while( xindex++ <= X )
               {
                 x=xindex*offsetx;
                 result=check( (double)a, (double)b, x, &y );
                 if( result!=0 )
                   {
                     destination = (y+side)/offsety;
                     yindex=(int)destination;
                     if( (destination − yindex) >= 0.5 ) yindex++;
                     data[THREADno][yindex][xindex]++;
                   }
               }
           }
/* ------------------------------------------------------------------- Hagh 変換の計算領域 (終り) */
MP       fclose( fin 2);
S      }


     int check( double a, double b, double x, double *y )      /* 逐次プログラムと同じ            */

     int change_to_function( double rho, double theta, double *a, double *b )
                                                               /* 逐次プログラムと同じ            */
```

6 おわりに

代表的な並列アルゴリズムを, 分割統治法, プロセッサファーム, プロセスネットワーク, 繰り返し変換の4つに分類し, これらアルゴリズムが適応可能な問題に対し, 並列プログラムを作成した. そして, 並列プログラムの骨格を抽出し, スケレトンとして事例ベースに保存する方法について述べた. また, 新たな問題に対して, 事例ベースの中から類似時例を検索し, スケレトンに修正を加えて新たな並列プログラムを作成する方法について検討した.

現在, 次の事例が完成している.

- 分割統治法
  1. クイックソート
  2. 積分計算
  3. コッホ曲線
- プロセッサファーム
  1. マンデルブロ
  2. KMP法
  3. レイトレーシング
  4. Hough変換
- プロセスネットワーク
  1. LU分解法
  2. バブルソート
- 繰り返し変換
  1. DKA法
  2. ワーシャルアルゴリズム

問題を解析し, どのアルゴリズムが適応可能であるかが決定すれば, 最も類似した事例のスレッド操作を用いて, 並列プログラムを作成することができる.

タスク分割方法は, 対象とするデータの構造の違い, 分割単位の動的変化, など様々な理由でパターン化することは難しい. しかし, 頻繁に使われる, 代表的な分割方法を持つ事例を保存することにより, かなりの分割パターンに対応できると考えられる.

また, 各アルゴリズムとインタラクションパターンの組み合わせを考えることにより, 新たな並列構造が考えられる. このような事例を作成することで, より適応範囲の広い並列構造をもつスケレトンの提供が可能となる.

今後の課題としては, 複数のスレッドの動的な振舞いを, ユーザに明確に提示できるよう, 動画などを用いて並列構造を表現することが提案されている. また, ユーザインターフェースを作成し, システムとの会話的な事例検索の実現も今後の課題である.

謝辞

本研究の機会を与えて下さり, 日頃, 貴重な御指導を頂きました山崎勝弘教授, 昨年度まで共に研究を行ってきた古川知之氏に心より深く感謝の意を表します. また, 本研究に際し, 事例作成に携わって頂いた安藤彰一氏, 並びに日頃より貴重な御意見を頂きました本研究室の皆様に深く感謝致します.

KSR1 を使用させて頂いているキャノン・スーパーコンピューティング S.I. 株式会社に感謝致します.

参考文献

[1] 小林 重信, “経験に基づく推論と学習”, 電子情報通信学会, Vol93, No.494, pp.25-32, 1994.

[2] 横田 一正, “演繹オブジェクト指向デ- タベ- ス Quixote の法的推論への応用”, 人工知能学会誌, Vol.10, No.1, pp.24-30, 1995.

[3] 山室 雅司, 川下 満, 中川 優, “デ- タベ- スエンジニアリングへの知識処理技術 “, 人工知能学会誌, Vol.10, No.1, pp.31-37, 1995.

[4] Rabhi, F.A., “Exploiting Parallelism in Functional Languages: Paradigm Oriented Approach”, *Second Workshop on Abstract Machine Models for Highly Parallel Computers*, pp.1-14, April 1993.

[5] Burkhart H. et al., “BACS: Basel Algorithm Classification Scheme”, *Technical Report 93-3,* Universität Basel, 1993.

[6] Kendall Square Research, “KSR Parallel Programming”, 1992.

[7] Kendall Square Research, “KSR C Programming”, 1993.

[8] Kendall Square Research, “KSR Fortran Programming”, 1992.

[9] Steven Frank, Henry Burkhardt and Dr.James Rothnie, “The KSR1: Bridging the Gap Between Shared Memory and MPPs”, Proc. of Compcon ’93, pp.285-294, 1993.

[10] J.Darlington, A.J.Field, P.G.Harrison, P.H.J.Kelly, K.W.N.Sharp, Q.Wn and R.L.While, “Parallel Programming Using Skeleton Functions”, PARLE’93 Parallel Architecture and Languages Europe, LNCS 694, Springer-Verlag, pp.146-160, 1993.

[11] S.G. アクル, “並列ソ- ティングアルゴリズム”, 啓学出版, 1988.

[12] 小池 慎一, “C による科学技術計算”, CQ 出版社, 1990.

[13] 長尾 真, “画像認識論”, コロナ社, 1990.

[14] 大須賀 節雄, “デ- タベ- スと知識ベ- ス”, オ- ム社, 1989.

[15] 山崎 勝弘, 古川 知之 , “事例ベ- ス推論による並列プログラミング支援システム”, 信学技法, *CPSY94-85*, pp.9-16, 1994.

[16] 古川 知之, 山崎 勝弘, “類似事例を用いた並列プログラミング支援”, 情報処理学会第 50 回全国大会, 2J-3, 1995.

[17] 松田 浩一, 山崎 勝弘, 古川 知之, “類似事例を用いた並列プログラミング支援”, 第 39 回システム制御情報学会研究発表講演会, 3007, 1995.

[18] 古川知之, “類似事例を用いた並列プログラミング支援システム”, 立命館大学大学院理工学研究科情報工学専攻修士論文, 1995.

[19] 安藤 彰一, “並列プログラミング用事例ベ- スの作成”, 立命館大学理工学部情報工学科卒業論文, 1996.

[20] 松田 浩一, 山崎 勝弘, “並列プログラミング用事例ベ- スの作成と類似例の検索 ·修正法”, 情報処理学会第 52 回全国大会, 3L-4, 1996.

付録