摘要

本論文では，過去に作成した並列プログラムを再利用することにより，並列プログラミングの負担を軽減する方法，また並列実行状況を可視化することで，並列処理の概念や構造の理解を深める方法について述べる．

並列処理は高速処理，高いコストパフォーマンスなどが科学技術計算，ビジネスなどの分野で期待されているが，プログラミングが難しく，一部の専門家だけにしか実現できないのが現状である．そこで過去の事例を再利用して問題を解決する事例ベース推論の考え方を用いる．

本研究は新たな問題の並列化を行う時，過去に作成した並列プログラムを蓄積している事例ベースから，最も類似した事例を検索・再利用することで，並列プログラミングの負担を軽減することを目的とする．事例ベースにはインデックス，スケレトン，並列プログラム，並列効果，及び履歴が蓄積されており，インデックスは問題の特徴を項目毎に解析・作成，スケレトンは並列プログラムの骨格を抽出したものである．システムはインデックスを用いて類似事例を検索し，類似事例のスケレトンにユーザが逐次プログラムを肉付け・修正することで並列プログラムを作成する．

また，並列計算が行われている各プロセッサの実行，待機，同期などの状況や，全体的な並列実行構造をユーザが把握するのは大変に難しい．そこで実行，待機時間などを取得し，このデータを用いて並列実行状況を可視化することで，並列実行の把握，負荷均衡の取れた並列性の高いプログラムの作成などを支援する．

事例修正による並列プログラミングで3次スプライン，エッジ抽出，包装アルゴリズム，巡回セールスマン問題のそれぞれについて，並列化のための問題解析，インデックスの作成，類似事例検索，並列プログラミング，事例修正の評価を行った．そして並列プログラムの実行状況の可視化について，ツールの作成，初期評価，検討を行った．また問題の並列化で重要となるタスク分割，並列プログラミングを行う上で重要となるスレッド操作，同期などの並列実行構造の可視化について検討した．

目 次

1 はじめに **1**

2 並列プログラミング **3**

2.1 並列アルゴリズムの分類 3

2.1.1 一般的分類 3

2.1.2 BACS による分類 5

2.2 並列実行構造 5

2.3 並列実行の可視化 9

3 事例ベース並列プログラミングシステム **10**

3.1 システム構成 10

3.2 事例の表現 11

3.3 類似事例の検索 13

3.4 類似事例の修正 14

4 事例修正による並列プログラミング **16**

4.1 3 次スプライン 16

4.1.1 問題の定義と解析 16

4.1.2 並列化 17

4.1.3 類似事例 (KMP 法) の修正 18

4.2 エッジ抽出 21

4.2.1 問題の定義と解析 21

4.2.2 並列化 21

4.2.3 類似事例 (3 次スプライン) の修正 23

4.3 包装アルゴリズム 26

4.3.1 問題の定義と解析 26

4.3.2 並列化 26

4.3.3 類似事例 (DKA 法) の修正 27

4.4 巡回セールスマン問題 31

4.4.1 問題の定義と解析 31

4.4.2 並列化 31

4.4.3 類似事例 (クイックソート) の修正 32

4.5 事例修正並列プログラミングの評価と考察 36

5 並列実行の可視化 **37**

5.1 全体構成と機能 37

5.2 実行状況データの取得方法と表示方法 38

5.3 初期評価と考察 42

6 おわりに **43**

謝辞 **44**

参考文献 **45**

付録 **48**

図 目 次

1 並列アルゴリズムの処理構造 4
2 並列アルゴリズムの並列実行構造 7
3 事例ベース並列プログラミングシステムの構成 10
4 スプライン 16
5 3 次スプラインのタスク分割 18
6 KMP 法のスケレトン 20
7 3 次スプラインの並列効果 21
8 エッジ抽出 22
9 エッジの抽出パターン 22
10 エッジ抽出のタスク分割 22
11 3 次スプラインのスケレトン 24
12 エッジ抽出の並列効果 25
13 包装アルゴリズム 26
14 包装アルゴリズムのタスク分割 27
15 包装アルゴリズムの並列効果 29
16 DKA 法のスケレトン 30
17 巡回セールスマン問題 31
18 巡回セールスマン問題のタスク分割 32
19 巡回セールスマン問題の並列効果 34
20 クイックソートのスケレトン 35
21 巡回セールスマン問題の並列実行の可視化 37
22 データ取得支援ツール 39
23 実行状況の表示例 41

表 目 次

1 BACS の構成要素 5
2 インデックス 11
3 自動検索による項目の削除順番 14
4 3 次スプラインのインデックス 19
5 KMP 法のインデックス 19
6 3 次スプラインの事例修正による評価 20
7 エッジ抽出のインデックス 23
8 エッジ抽出の事例修正による評価 25
9 包装アルゴリズムのインデックス 28
10 DKA 法のインデックス 28
11 包装アルゴリズムの事例修正による評価 29
12 巡回セールスマン問題のインデックス 33
13 クイックソ- トのインデックス 33
14 巡回セールスマン問題の事例修正による評価 34
15 ヘッダファイルの関数の機能 38
16 システムの問い合わせ項目の説明 40

1　はじめに

近年，並列処理はワークステーションやスーパコンピュータなどの計算機の基本構成技術となっており，多数の高性能プロセッサをネットワークなどで接続して，それぞれのプロセッサで計算することにより，逐次処理よりも高速な処理を実現している．並列処理により，高速性と高いコストパフォーマンスが科学技術計算やビジネスなどの分野で期待されている．しかし並列処理は即座に，簡単に，誰にでも実現できるとは言えず，一部の専門家だけしか実現できないのが現状である．その理由として，プログラミングとデバッグが困難であることが挙げられる．並列プログラミングに関しては，各計算機のアーキテクチャを熟知し，プロセッサ間で行われる同期，相互排除，タスク分割などをユーザ自身が考えねばならない．そこで並列プログラミングの経験を十分に活用する必要がある．

並列プログラミングの困難さを支援するために，過去に並列プログラミングの専門家が経験した問題，知識などを再利用し，問題を解決する方法を用いる．問題を解決する技術の 1 つとして事例ベース推論 (Case-Based Reasoning: CBR)[1][2] がある．過去に経験した問題を再利用可能な形で事例ベースに蓄えておき，新たな問題を解決するときに類似した事例を事例ベースから検索し，その事例を再利用することにより問題を解決する．先の開発者が費やした時間と労力を，後の開発者が同様の苦労をすることになれば，大変に無駄なことになるため，この事例ベース推論が考え出された．すでに法律推論，レシピ作成などに応用されている [3]．

本研究では，過去に作成された並列プログラム中のタスク分割，同期，スレッド生成などを再利用して並列プログラミングを行い，その負担を軽減させることを目的とする．このシステムの流れは，新しく並列プログラムを作成するときに，新たな問題に対するタスク分割方法，データ依存関係による参照データの通信パターンなどの特徴付けを行い，その特徴を基に事例ベースから類似した過去の並列プログラムの事例を検索する．全ての特徴が一致した事例があれば良いが，無ければ検索条件を緩和して最も類似した事例を検索する．そして検索した事例のタスク分割，スレッド生成などが記述されている並列プログラムの骨組みとなるものに，新たな問題の逐次プログラムを肉付け・修正を行うことにより目的の並列プログラムを作り上げる．事例を用いて並列プログラミングを行うことで，タスク分割，スレッド生成などのプログラミングの負担を軽減できる．

また並列実行の可視化は，各プロセッサの実行，待機，同期などの状況を表示して，並列プログラミングを困難なものとしているデバッグを支援し，負荷均衡が取れた並列性の高いプログラムの作成を支援する．現在，並列実行状況の把握，並列プログラミングを効率良く進めるために，並列実行可視化ツール [5] [6] などが考え出されている．並列処理には各プロセッサのタスク分割，タスク間の相互排除，プロセッサ間の同期などが必要であり，並列計算機の性能を最大限に引き出すことのネックとなっている．現在，自動並列化コンパイラ，並列プログラムデバッガなどの多数の並列化支援ツール [7][8] が研究されてい

るが，確立された技術はまだ開発されていない．

過去の並列プログラミングの知識，経験として事例ベースに蓄積されている事例は，インデックス，スケレトン，並列プログラム，並列効果，及び履歴から構成している．インデックスは問題の定義，データ構造，タスクの分割方法，同期の取り方などの問題の特徴付けをしたもので，事例検索の際に使用する．スケレトンは並列プログラムの骨格となる部分で，スレッド操作方法，同期の取り方，タスク分割方法など並列処理における重要な部分が記述されている．このスケレトンに単位計算部分などを肉付け・修正を行うことで並列プログラムを作成する．並列プログラムは事例のプログラム本体，並列効果は実行時の速度向上を表したグラフ，履歴は類似事例としての参照回数などがそれぞれ記述されており，類似事例を決定するときに参考にする．

類似事例を検索，あるいは問題の特徴付けをする上で，それぞれの並列プログラムの実行構造を把握するのは困難であるので，特徴を明確に，かつ細かく分類する必要がある．そこで計算ステージの振る舞いのパターン，参照データの通信パターンなどの観点から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つに分類する．また実行構造，相互作用，プロセス構造などから並列アルゴリズムを分類し，簡単な略語と記号で並列プログラムを表現する方法として，バーゼル大学で提案されたBACS(Basel Algorithm Classification Scheme)がある[14]．このBACSを用いて並列プログラムを分類し，インデックスの中の特徴付けの1つとして使用する．

本研究の並列プログラミングは仮想共有メモリ型並列マシンKSR-1 [15][16][17]を用いて行っている．KSR-1は仮想的な共有メモリの実現で，あるアドレスに決まった物理的位置は存在せず，ハードウェアが要求に応じて参照ポイントにデータを移動させる特徴を持つ．よってユーザはソフトウェアによる煩雑な作業は必要としないので，容易にプログラミングができる．

本論文は2章で並列アルゴリズムの分類，並列実行構造，3章で事例ベース並列プログラミングシステムの概要，4章で事例修正による3次スプライン，エッジ抽出，包装アルゴリズム，巡回セールスマン問題の並列プログラミング，5章で並列実行の可視化について述べ，6章でまとめる．

2　並列プログラミング

2.1　並列アルゴリズムの分類

2.1.1　一般的分類

並列アルゴリズムは複数の計算ステージで構成されており，計算ステージ間でデータを参照しながら，各計算ステージを並列に実行することにより，並列処理が実現できる．計算ステージの振る舞いのパターンと，参照データの通信パターンの観点から，並列アルゴリズムは分割統治法，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できる．

(1) 分割統治法

分割統治法はデータを分割して，計算ステージ自身と他の計算ステージにデータを振り分け，条件が成立するまでこれを繰り返し，最終的にデータを結合することにより解を得る．図1(a)に分割統治法の処理構造を示す．また参照データの通信パターンは木構造(トリー)である．

(2) プロセッサファーム

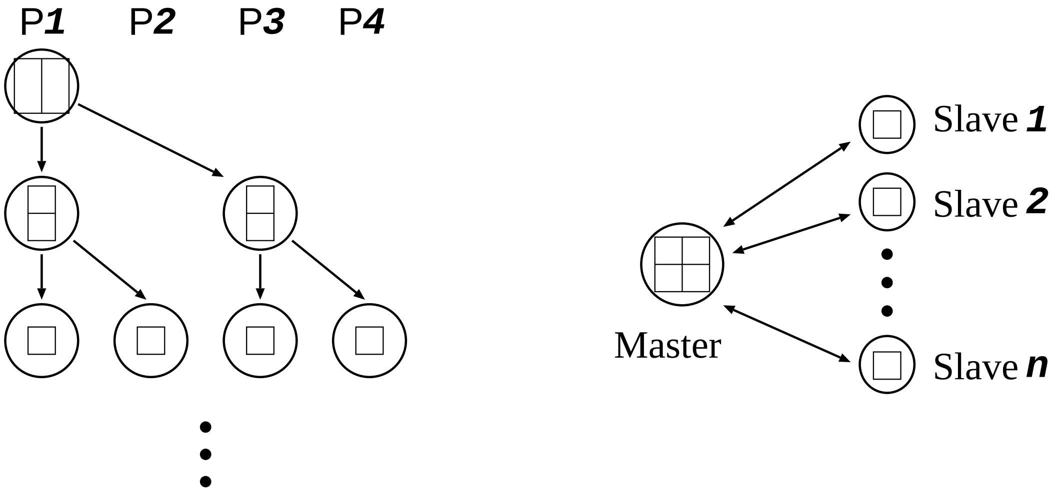
プロセッサファームはマスターがデータを分割し，それぞれの計算ステージとなるスレーブが完全に独立して分割データを計算する．そしてそれぞれのスレーブの結果をマスターが回収・結合して解を得る．図1(b)にプロセッサファームの処理構造を示す．参照データの通信パターンはマスター&ワーカーである．

(3) プロセスネットワーク

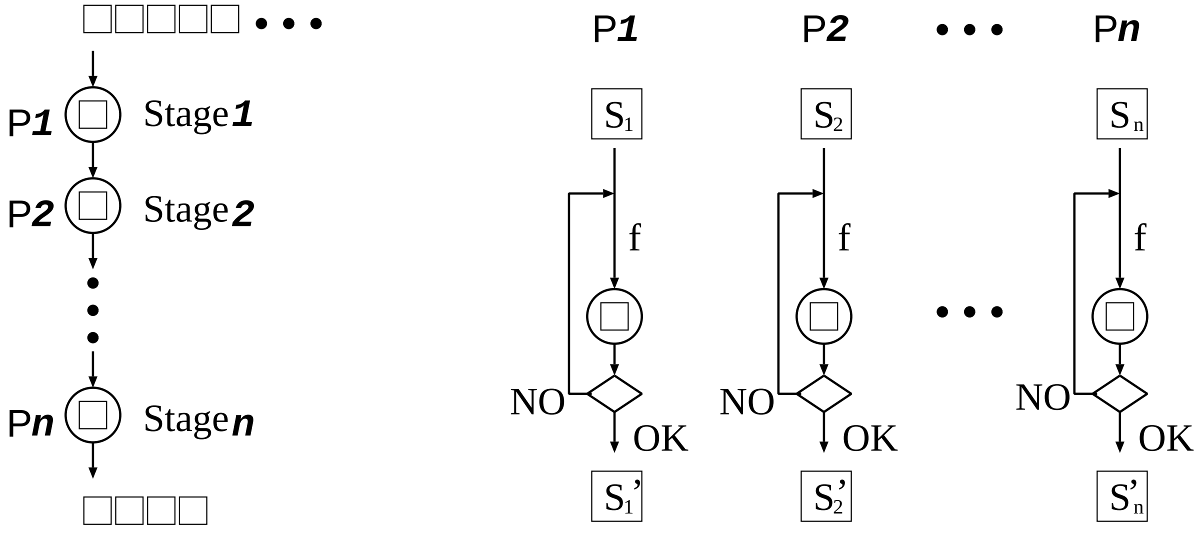
プロセスネットワークは計算ステージを複数に分けて，複数のデータがその各計算ステージを流れていくことで解を得る．図1(c)にプロセスネットワークの処理構造を示す．参照データの通信パターンはパイプなどがある．

(4) 繰り返し変換

繰り返し変換は各計算ステージでデータがある条件を満たすまで変換を繰り返し実行することで解を得る．図1(d)に繰り返し変換の処理構造を示す．参照データの通信パターンは部分的にデータを参照する部分照合などがある．



(a) 分割統治法　(b) プロセッサファーム



(c) プロセスネットワーク　(d) 繰り返し変換

図 1: 並列アルゴリズムの処理構造

2.1.2　BACS による分類

BACS(Basel Algorithm Classification Scheme) は並列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バーゼル大学で提案された．BACS はアルゴリズムトポロジ，プロセス構造，実行構造，相互作用メカニズム，及びデータ分散の観点から並列アルゴリズムを分類する．また簡単な略語と記号で並列プログラムを表現することができる．

表 1 に BACS の構成要素を示す．プロセスは構造，トポロジ，実行構造から構成し，実行構造の微視は計算 (calculation)，相互作用 (interaction)，デーモンコール (demon call) によるプロセスの合成順序を示し，巨視はアルゴリズムの全てのイベント列となる．相互作用の直接は 2 ノード間，広域は 3 ノード以上間で行われるものとなる．データはデータ構造の定義，データの分散方法から構成する．

表 1: BACS の構成要素

プロセス	構造	静的，動的
	トポロジ	ワーカー，パイプライン，メッシュ，トーラス，トリー， デバイド&コンカー，ハイパーキューブ，マスター・ワーカー
	実行構造	イベント = calculation(C)，interaction(I)，demon call(D) 微視：C，I，D によるプロセスの合成順序 巨視：全てのイベント列
相互作用	広域・陰	lock，unlock(mutex); read，write(shared variables)
	広域・陽	barrier
	直接・陽	signal，wait
データ	データ構造	ソースデータ，結果データ
	データ分散	分割 (partitioning): 行方向，列方向，メッシュ 配置 (binding): 共有，ブロック毎，サイクリック

2.2　並列実行構造

仮想共有メモリ型並列マシン KSR-1 に依存した並列実行構造について述べる．KSR-1 は計算ステージとなる複数のプロセッサを並列に実行することにより，並列処理が実現できる．共有メモリ並列プログラミングでは，共有変数の相互排除を実現するために，ロック・アンロックを陽に指定し，各プロセッサ間の情報授受を行う．KSR-1 には各プロセッサにスレッドがそれぞれ用意されており，各スレッド間ではデータの共有ができ，共有メモリによりデータ通信が可能である．これによりスレッド間の同期，相互排除が容易に実現で

きる．並列プログラムは同期，タスク分割などのイベントを問題に応じて組み合わせることによって，いろいろな動的振る舞いをする．KSR-1 上での並列実行にはいろいろなイベントがある．並列計算を行うステージとなるスレッドの自動的，あるいは数・タイミングが分かっている時の静的な生成・消去や，スレッド間の参照データ依存性を考慮してデータの一貫性保持のための同期・相互排除や．各計算ステージ間でデータの受け渡しをするための参照データ通信パターンや，各スレッドに計算対象となるデータを振り分けるためのタスク分割などがある．以下に KSR-1 に依存した並列アルゴリズム毎の実行構造について説明する．

(1) 分割統治法

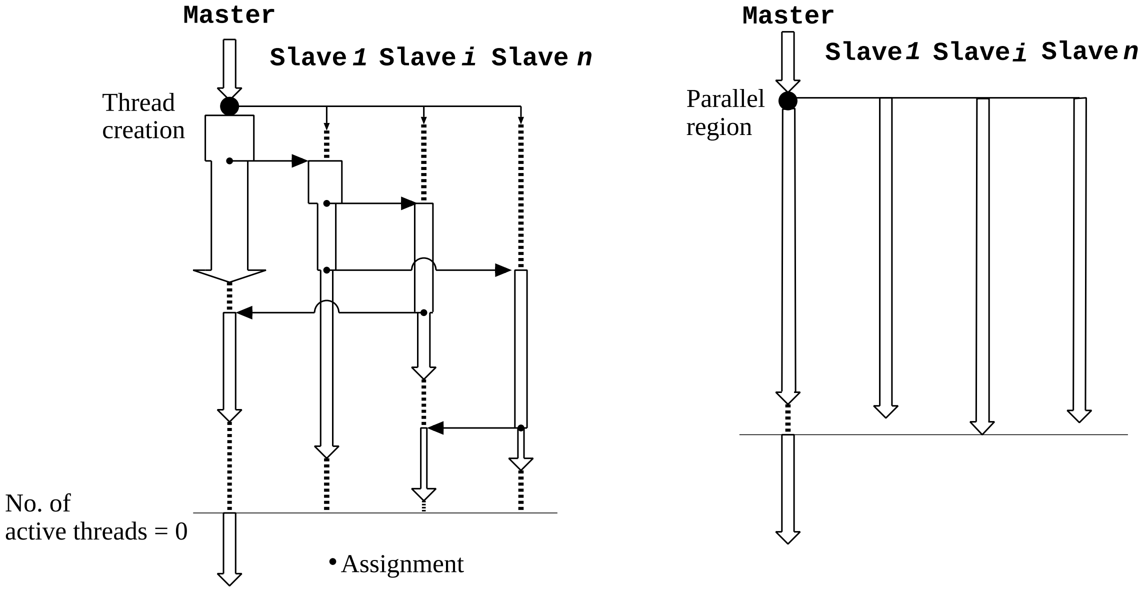
分割統治法はタスク分割を繰り返していき，結果を回収することで解を得るアルゴリズムである．図 2(a) に分割統治法の並列実行構造を示す．スレッドはタスクを分割し，1 つの分割タスクをそのままこのスレッド自身が受け持ち，他の分割タスクはアイドルスレッドに渡すが，アイドルスレッドが検索できなかった場合は，他の分割タスクもこのスレッド自身が受け持つことになる．そしてこのタスク分割を与えられた条件が成立するまで繰り返し，最終的に得られた結果をそれぞれのスレッドから回収し，結合する．各スレッドは受け取ったタスクの計算が終わっても消滅せずに，アイドルスレッドとなって分割タスクを待ち，全てのスレッドがアイドルスレッドとなった時点で並列計算が終了し，スレッドを消去する．

(2) プロセッサファーム

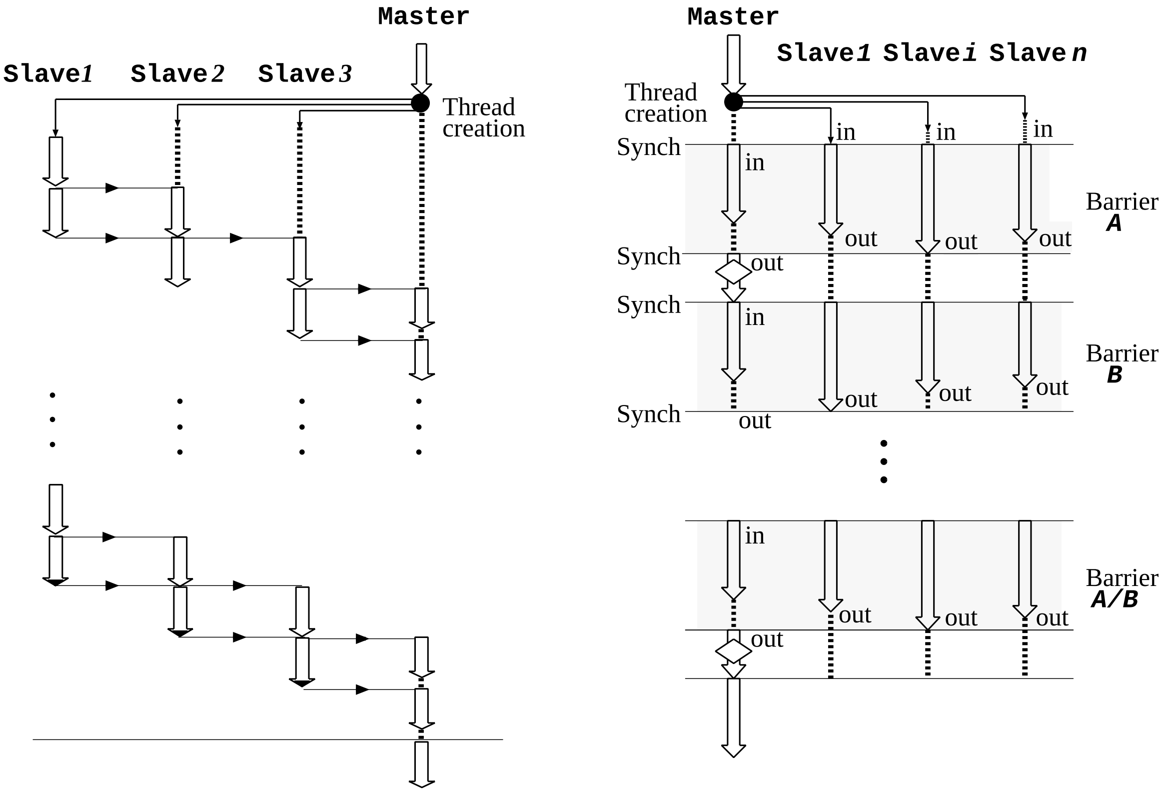
プロセッサファームは分割タスクをそれぞれのスレッドで独立に計算し，計算結果をまとめることで解を得るアルゴリズムである．図 2(b) にプロセッサファームの並列実行構造を示す．マスターがタスクを分割し，それぞれのスレッドに振り分ける．各スレッドは他のスレッドとは完全に独立して計算し，結果をマスターに返す．マスターがそれぞれのスレッドから回収した計算結果をまとめ，最終的な結果を得る．タスク分割は 1 回だけ行い，それぞれのスレッドはその 1 つだけの分割タスクで計算する．

(3) プロセスネットワーク

プロセスネットワークは計算ステージを複数に分け，複数のデータを各計算ステージに流れ作業のように流すことで解を得るアルゴリズムである．図 2(c) にプロセスネットワークの並列実行構造を示す．スレッドは上位スレッドからタスクを受け取り，そのタスクを基に計算する．そして下位スレッドにタスクを渡すことで下位スレッドは計算を開始し，そのスレッド自身は上位スレッドからの次のタスクを待つ．このようなパイプライン処理をそれぞれのスレッドが同期を取りながら並列に計算し，ある条件が成立するまで繰り返されて結果を得る．



(a) 分割統治法　(b) プロセッサファーム



(c) プロセスネットワーク　(d) 繰り返し変換

図 2: 並列アルゴリズムの並列実行構造

(4) 繰り返し変換

繰り返し変換はそれぞれのスレッドがタスクを変換し，マスターが変換結果を評価する流れを条件が成立するまで繰り返して解を得るアルゴリズムである．図 2(d) に繰り返し変換の並列実行構造を示す．各スレッドは同期を取って同時にそれぞれの分割タスクを計算して変換を行う．マスターはそれぞれのスレッドの変換結果を評価し，条件が成立すればそこで並列計算が終了となるが，成立しなければ条件が満たされるまで変換を繰り返す．他のスレッドより早く変換が終了したスレッドが再度変換が始まらないように，2 つのバリアを用いて同期を取る．

並列プログラムの主なイベントとして，スレッド生成，スレッド間，あるいはデータ間の同期がある．同期には複数のスレッド間で同期を取るバリア，参照データ間で同期を取るミューテックス，2 つのスレッド間で同期を取るシグナル・ウェイトがある．

スレッド生成にはプログラム中で自動的にスレッドを生成する動的な生成のパラレルリージョン，プログラムで指定したタイミングで手動でスレッドを生成する静的な生成のスレッドライブラリがある．パラレルリージョンは各スレッド間で同期を取って同時にスレッドを生成して並列計算開始の準備をする．スレッドライブラリは分割タスクが渡されて始めてスレッドを生成して計算開始の準備をし，担当する計算が終わればスレッドを消去して次の分割タスクが来るまで待つ場合に使用される．スレッドライブラリはこのような流れをユーザがプログラム中で陽にプログラミングすることで実現し，主に分割統治法で用いられる．

マスターがチェックインの指示を出して，各スレッドが同期を取って同時に実行を開始し，全スレッドがチェックアウトするのを待って各スレッドの計算結果をマスターが参照する．このメカニズムを繰り返し行いたい場合にバリアが用いられる．またバリアはマスターがチェックインしている時は，スレーブに対するチェックインの許可は続くので，バリアが 1 種類しかなければ，マスターより早くチェックアウトしたスレーブはすぐに再チェックインをしてしまうので，同期が成立しなくなる．よって 2 種類のバリア A/B を用意することで，バリア A をチェックアウトしたスレーブはマスターがバリア B のチェックイン許可を出すまで待つことになる．

ミューテックスは共有変数を参照する時，複数のスレッドが同時にデータの読み書きをして，データの一貫性を崩すのを防ぐために使用する同期である．スレッドはロックをしてから共有変数の読み出し/書き込みを行い，終わればアンロックを行う．あるスレッドがロック中は，他のスレッドは共有変数の読み出し/書き込みができなくなり，共有変数を参照する場合はロック中のスレッドがアンロックするまで待つことになる．

シグナル・ウェイトは共有変数を用いて，2 つのスレッド間だけで同期を取るために使用する．2 つのスレッドは互いに同期を取る相手を認識しており，それぞれのスレッドの関係が明確に定義される．それぞれのスレッドでビジー/アイドルを示す共有変数を用意する．計算を終了したスレッドが相手のスレッドの状態を共有変数から読み取り，ビジーで

あれば自分をアイドルにして相手のスレッドが計算を終了するのを待ち，アイドルであれば相手をビジーにして再び自分と相手のスレッドが計算を始める．

2.3 並列実行の可視化

並列プログラムの動的な振る舞いをプログラマが理解するのは大変に困難で，並列プログラムが非生産的で，一般ユーザに馴染みにくいものとなっている．一般的な並列処理の概念や，並列プログラムの観点からの振る舞いなどを可視化表示し，プログラマに提示することにより，並列処理の概念の理解を深めることができ，並列プログラムの生産性向上に役立つと考えられる．

並列処理の一般的な概念として，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 つの並列アルゴリズムの動的な振る舞いを提示することで，並列処理の全体的なメカニズムの理解が深められる．次に並列プログラミングを行う上で最初に検討することはタスク分割で，分割対象のデータ構造，データ間の依存性，及び問題の解法によっていろいろな分割方法が考えられる．分割方法の構成図，プログラム例，どういう時に使用するかの特徴などを提示することで，最適な分割方法のプログラムを例にして，プログラミングができる．

トポロジにはいろいろなパターンがあって，一部には簡単な動作をするものがあるが，参照データの分割タスク間でのやり取りが難しいものも多々ある．このトポロジも動作の構成図，どういう時に使用するかの特徴，プログラムがあればその例を提示することで，問題にあったトポロジを選択できる．同期は各スレッドにまたがって実行されるものであるので，各スレッドがそれぞれどのような動的な振る舞いを行うのか理解が難しい．動作例の図，特徴，プログラムがあればその例を提示することで，同期を取っているプログラミングの手助けが出来る．各プロセッサの計算ステージとなるスレッドの生成には自動的に生成する動的生成と，プログラムでタイミングを指定して手動で生成する静的生成の 2 通りの方法がある．これは生成タイミングが分かるような図を提示することで，プログラマが問題にあった方法を選択できる．

以上のように過去の経験を再利用するように，細かい観点から並列プログラムを眺めることにより，難しく考えることなく，着実に並列プログラムが出来上がると考えられる．

3　事例ベース並列プログラミングシステム

3.1　システム構成

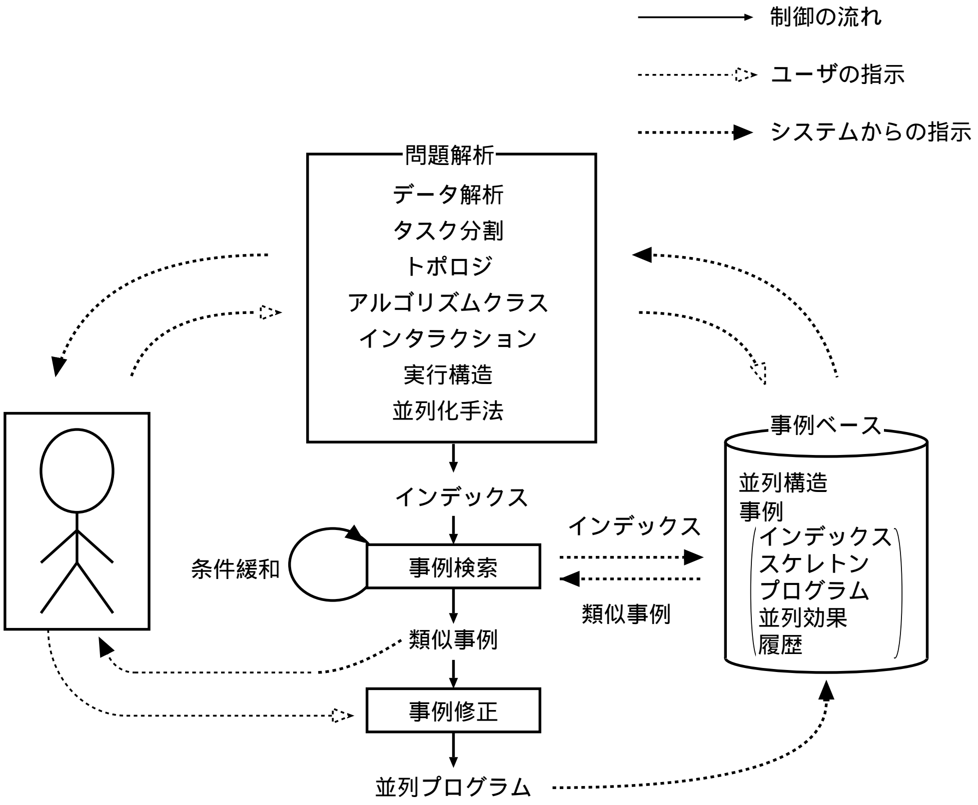


図 3: 事例ベース並列プログラミングシステムの構成

図 3 に事例ベース並列プログラミングシステムの構成を示す．事例ベースに蓄積されている事例はインデックス，スケレトン，並列プログラム，並列効果，及び履歴から構成する．インデックスには応用，仕様，ソースデータ，結果データ，タスク分割，終了条件，トポロジ，アルゴリズム，並列化手法，インタラクション，及び BACS の 11 個の項目に分けて，並列プログラムの特徴が記述されている．

ユーザは新たな問題を並列化する上での問題解析を行って，インデックスの項目を埋める．問題解析は分割対象となるタスク，タスク分割の方法，分割タスク間の依存性，共有変数の相互排除，アルゴリズムの終了条件，参照データ通信パターンのトポロジなどを行う．システムはインデックスを基に事例ベースから再利用可能な形で蓄積されている事例を検索する．全ての項目が一致した事例が検索されれば簡単な修正で並列プログラムが出来上がるので良いが，無ければ検索条件を緩和して最も類似した事例を検索することになる．また類似事例の並列効果，並列プログラム，履歴から速度向上の期待度，プログラミ

ングの容易さなどをユーザが予測し，最適な事例を決定する．

次に決定した事例のスレッド生成，タスク分割，同期などの並列プログラムの重要な部分が記述されたスケレトンに，ユーザが予め作成した逐次プログラムを肉付け・修正して目的の並列プログラムが出来上がる．ユーザが新たに作成した並列プログラムはインデックス，スケレトン，並列効果と一緒に事例ベースに蓄積し，新たな事例として今後の並列プログラミングに利用される．

3.2 事例の表現

事例ベースに蓄積されている事例のインデックス，スケレトン，並列プログラム，並列効果，及び履歴について，以下に説明する．

(1) インデックス

表 2: インデックス

応用	問題の応用分野 (数値計算，ソーティング，. . .)
仕様	問題の定義
ソースデータ	ソースデータ構造 (整数型，実数型 2 次元配列，. . .)
結果データ	結果データ構造
タスク分割	分割対象　　　　　，分割単位　　　，分割方法
終了条件	問題のアルゴリズムの終了条件
トポロジ	マスター&ワーカー，パイプ，完全照合，部分照合，トリー，. . .
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
並列化手法	パラレルリージョン，スレッドライブラリ
インタラクション	バリア，ミューテックス，シグナル・ウェイト
BACS	簡単な略語と記号による BACS 表現

表 2 にインデックスを示す．インデックスは並列プログラムの構造，特徴などを記述したもので，問題の並列プログラムがどんなものかを表すと同時に，類似事例の検索の際に検索条件の項目として利用される．ユーザが並列プログラミングを行う時にまずこのインデックスを作成する．以下にそれぞれの項目について説明する．

- 応用
  数値計算，画像処理，ソーティング，文字列照合，グラフ問題，経路探索など，並列化を行う問題の応用分野を記述する．

- 仕様
  問題の定義，解法，アルゴリズムを定義する．

- ソースデータ
  問題の並列計算対象となるデータのデータ構造とその変数名を記述する．

- 結果データ
  並列計算の結果を格納するデータのデータ構造とその変数名を記述する．

- タスク分割
  ソースデータ，結果データなどの分割対象となるデータの変数名と，要素，行，列などの分割する単位と，ブロック，サイクリック，共有などの分割の方法のそれぞれについて記述する．ユーザはタスク分割の解析を行う時にまず，分割対象データを分割してそれぞれ独立に計算するのか，あるいは全データが計算に必要で共有して計算するのか解析する．次にデータを分割するのであれば，それぞれのスレッドでの計算負荷を予測し，均等になるように複数要素によるブロック分割，1 要素によるサイクリックなどの分割を決定する．タスク分割は分割単位と分割方法の組合せにより，行ブロック，1 要素によるサイクリック，複数列ブロックによるサイクリックなどのいろいろな方法が考えられる．

- 終了条件
  ある条件が成立するまで計算を続けるというアルゴリズムの終了条件を記述する．

- トポロジ
  それぞれのスレッドの参照データがスレッド間でどのように流れるかの通信パターンを記述する．

- アルゴリズム
  終了条件，トポロジの他に，スレッド間での分割タスクの依存性，共有変数への読み出し・書き込みを行う相互排除などを検討して，並列アルゴリズムを決定して記述する．

- 並列化手法
  自動的にスレッドを生成するパラレルリージョンと，指定した所で静的にスレッドを生成するスレッドライブラリを問題に応じて決定し記述する．

- インタラクション
  複数のスレッド間での計算開始の同期を取るバリア，複数のスレッドが同時に共有変数へデータを書き込むのを防ぎ，データの一貫性を保つためのデータアクセス制御の同期を取るミューテックス，2 つのスレッド間で共有変数を用いて計算開始の同期を取るシグナル・ウェイトを問題に応じて決定し記述する．

- **BACS**
  簡単な略語と記号で並列アルゴリズムの構造を表現できる BACS 表現を記述する．

(2) スケルトン

スケルトンはスレッド生成，タスク分割，同期，相互排除などの並列プログラムの重要な部分がプログラムの骨格として記述されており，ユーザが逐次プログラムの単位計算，初期設定，結果出力部分などを肉付け・修正するだけで並列プログラムが出来上がるものである．

(3) 並列プログラム

事例の並列プログラム本体が記述されており，類似事例を決定する際にこの事例を用いた場合，どれぐらいの修正で並列プログラムが出来上がるかを予測するために用意されている．

(4) 並列効果

事例の並列プログラムを実行した時の速度向上比をグラフで表現したもので，ユーザはこの事例が並列プログラムとしてどれぐらい成功しているか，この事例を類似事例として並列プログラムを作成した時どれぐらいの速度向上が得られるかを検討するために用意されている．

(5) 履歴

類似事例としての使用頻度，並列プログラムとしての成功/失敗が記述されており，類似事例の決定のための参考として用意されている．

3.3 類似事例の検索

システムはユーザが作成したインデックスを用いて事例を検索し，1 つでも多くの項目が一致した事例を類似事例としてユーザに提示する．以下に類似事例検索の検索手順を示す．

1. システムはユーザにインデックスの作成を促し，作成したインデックスを基に事例を検索する．

2. 事例が検索されなかった場合，検索条件を緩和して再び検索を行う．条件緩和は手動/自動の 2 通りから選択し，インデックスのあまり重要としない項目を削除することで行う．手動の条件緩和はユーザ自身が検索にあまり重要でないインデックスの項目を判断して，削除する．自動の条件緩和はシステムが予め決められているインデックスの項目を自動的に削除する．表 3 に自動検索による項目の削除順番を示す．10 が最も重要な項目となる．
3. ユーザが作成したインデックスの横に，類似事例のインデックスを表示する．複数の類似事例が検索されれば，順に表示する．
4. 事例のスケレトン，並列プログラム本体，並列効果，履歴をユーザの必要に応じて表示する．
5. 検索した事例から，最終的にユーザが類似事例を決定する．

表 3: 自動検索による項目の削除順番

1. 応用	6. インタラクション
2. 仕様	7. 並列化手法
3. 終了条件	8. BACS
4. ソース/結果データ構造	9. トポロジ
5. タスク分割	10. アルゴリズム

3.4 類似事例の修正

検索された類似事例のスケレトンに，ユーザが予め作成しておいた問題の逐次プログラムの単位計算，初期設定，結果出力部分などを肉付け・修正することで，目的の並列プログラムを完成させる．

ユーザはまず，類似事例のスケレトンがそのまま再利用可能かどうかを検討する．おそらくスレッドを生成する部分は殆どの場合再利用できると考えられるが，タスク分割，同期の部分が大抵の場合多少の修正を加える必要がある．タスク分割は分割方法が違えばもちろん修正が必要で，データ構造が異なる場合も修正しなければならないことがある．タスク分割はこの時点で修正可能であるが，同期は，予めユーザが作成しておいた問題の逐次プログラムを肉付けしてから修正することになる．

次にスケレトンに逐次プログラムを肉付けする．肉付けするものとしては，単位計算，計算対象の変数の初期設定，計算結果を出力する部分などがある．肉付けの次は共有変数，

ローカル変数，相互排除，同期などの観点から，修正をしなければならない．共有変数は各スレッドが同時にデータの読み書きを行えばデータの一貫性が保てなくなるため，データのアクセス制御により相互排除を行う必要がある．共有変数の読み書きの前後で，ユーザが相互排除のプログラミングを行う．ローカル変数はたびたび共有変数を書き換えてそのたびに同期を取るのが無駄だと考えた場合に，それぞれのスレッドに用意する変数で，新規作成することになる．同期は単位計算部分のどこのタイミングで取れば良いのか考える必要がある．以上のように修正を施して，並列プログラムを作り上げる．

4　事例修正による並列プログラミング

4.1　3 次スプライン

4.1.1　問題の定義と解析

スプラインは図 4 のように与えられた複数の座標点 (● 印) の各区間で滑らかな曲線となるような座標点 (× 印) を求める問題である [18].

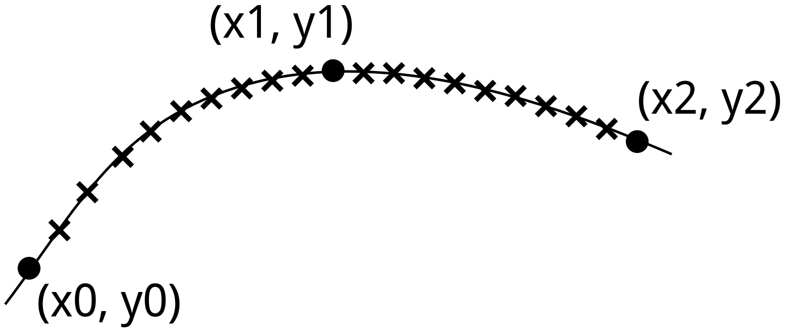


図 4: スプライン

3 次スプラインは，

$$x_i(u) = b_0^{(i)} + b_1^{(i)}u + b_2^{(i)}u^2 + b_3^{(i)}u^3 \qquad (0 \le u \le 1,\ i = 0, \cdots, N) \tag{1}$$

のように座標点を 3 次パラメータ表現して，u の値を変えることで補間を行う．100 個の点で補間を行うのであれば，u $(0 \le u \le 1)$ の値を 100 に区切って，(1) 式に代入して計算する．(1) 式を計算するには，$b_0^{(i)}, b_1^{(i)}, b_2^{(i)}, b_3^{(i)}$ の値を求める必要がある．これは (1) 式の 1 ・2 次微係数が一致し，かつ座標点を通ることを用いて求める．

(1) 式の 1 ・2 次微分は次のようになる．

$$x_i'(u) = b_1^{(i)} + 2b_2^{(i)}u + 3b_3^{(i)}u^2 \tag{2}$$

$$x_i''(u) = 2b_2^{(i)} + 6b_3^{(i)}u \tag{3}$$

(1) 式がデータ点を通ること，1 次微係数が連続であることから，

$$x_{i-1}(0) = b_0^{(i-1)} = x_{i-1} \tag{4}$$

$$x_{i-1}'(0) = b_1^{(i-1)} = x_{i-1}' \tag{5}$$

となる．また，$x_{i-1}(u_{i-1})$ と $x_{i-1}'(u_{i-1})$ を求めると，

$$x_{i-1}(u_{i-1}) = x_{i-1} + x_{i-1}'u_{i-1} + b_2^{(i-1)}u_{i-1}^2 + b_3^{(i-1)}u_{i-1}^3 = x_i \tag{6}$$

$$x_{i-1}'(u_{i-1}) = x_{i-1}' + 2b_2^{(i-1)}u_{i-1} + 3b_3^{(i-1)}u_{i-1}^2 = x_i' \tag{7}$$

となり，(6),(7) 式より，

$$\begin{aligned} b_2^{(i-1)} &= \frac{3(x_i - x_{i-1})}{u_{i-1}^2} - \frac{2x'_{i-1} + x'_i}{u_{i-1}} && (8) \\ b_3^{(i-1)} &= \frac{2(x_{i-1} - x_i)}{u_{i-1}^3} + \frac{x'_{i-1} + x'_i}{u_{i-1}^2} && (9) \end{aligned}$$

となる．(5),(8),(9) 式より，$b_1^{(i-1)}, b_2^{(i-1)}, b_3^{(i-1)}$ を求めるには，x'_i, x'_{i-1} の値が必要となる．そこで 2 次微係数が等しいことから，

$$\begin{aligned} x''_{i-1}(u_{i-1}) &= 2b_2^{(i-1)} + 6b_3^{(i-1)} u_{i-1} = x''_i && (10) \\ x''_i(0) &= 2b_2^{(i)} = x''_i && (11) \end{aligned}$$

となり，(10),(11) 式より，

$$\frac{u_{i-1}}{3} x'_{i+1} + \frac{2u_i + 2u_{i-1}}{3} x'_i + \frac{u_i}{3} x'_{i-1} = \frac{u_{i-1}}{u_i} x_{i+1} + \left(\frac{u_i}{u_{i-1}} - \frac{u_{i-1}}{u_i} \right) x_i - \frac{u_i}{u_{i-1}} x_{i-1} \qquad (12)$$

となる．(12) 式の右辺は，値を与えると定数が得られるので，(12) 式は x' を求める連立方程式となる．しかし，この連立方程式は式の数より，未知数の方が 2 つ多いので，以下のように条件式を与える．

$$\begin{aligned} x'_0 &= \frac{1}{2u_0}(x_1 - x_0) && (13) \\ x'_n &= \frac{1}{2u_{n-1}}(x_n - x_{n-1}) && (14) \end{aligned}$$

以上のことから，(13),(14) 式を計算した値で (12) 式より x'_k $(k = 0, \cdots, n)$ を求める．そして x'_k を用いて，$b_0^{(i)}, b_1^{(i)}, b_2^{(i)}, b_3^{(i)}$ のそれぞれを (4),(5),(8),(9) 式から求める．

4.1.2　並列化

ソースデータ，結果データを実数型構造体の 1 次元配列 $point[\,].x$,y , $result[\,].x$,yとする．それらをスレッド数分のブロックに分割し，ブロック毎にスレッドに割り当てる．図 5 にタスク分割を示す．分割タスク間の相互排除，スレッド間の同期，終了条件は無いのでアルゴリズムはプロセッサファームを使用できる．並列化手法はパラレルリージョン，全スレッドで同じ計算を行うので BACS 表現は C^t (全スレッドでの同一計算) となる．類似事例として文字列照合 (KMP 法) が検索される．

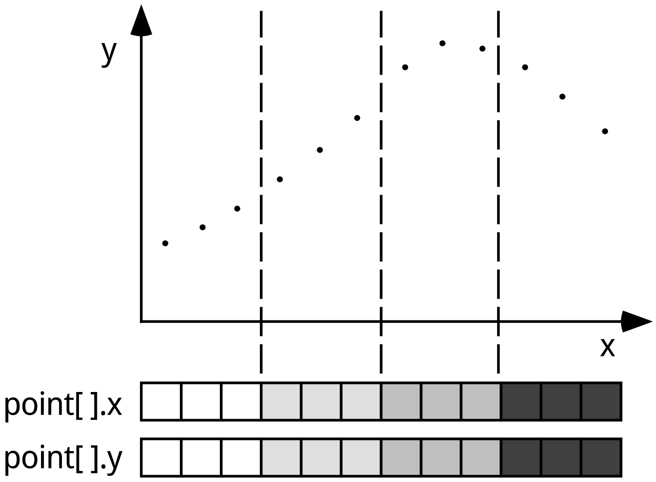


図 5: 3 次スプラインのタスク分割

4.1.3 類似事例 (KMP 法) の修正

(1) インデックス

表 4 に 3 次スプラインのインデックスを示す．問題解析を行った結果をインデックスのそれぞれの項目に書き込んでいる．問題の応用分野には数値計算，仕様には問題の定義，ソース/結果データには実数型構造体 1 次元配列，タスク分割には要素ブロックによる分割をそれぞれ記述している．また表 5 に類似事例として検索された文字列照合 (KMP 法) のインデックスを示す．KMP 法は予め検索文字列 (PAT) の構造を解析して，文字列 (TEXT) 内で PAT と一致する文字列を順次照合・検索する問題で，一致する見込のない文字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っている．

3 次スプラインと KMP 法の相違点はタスク分割だけである．KMP 法のタスク分割は隣接する 2 つの分割ブロックの境界に PAT が存在する可能性があるので，境界付近の一定領域を重複して分割している．3 次スプラインのタスク分割はこの重複ブロックを重複しない要素ブロックに，そして 1 次元配列を構造体 1 次元配列に修正する必要がある．またインデックスの他の項目は全て一致するので，スレッド生成部分を再利用し，単位計算部分を肉付けするだけで並列プログラムが完成できると予想できる．

(2) 事例の修正

KMP 法の事例を修正して 3 次スプラインの並列プログラムを作成する．図 6 に KMP 法のスケレトンを示す．このスケレトンは main からパラレルリージョンを起動して，全スレッドで work を実行するという並列プログラムの骨格となる部分が示している．スレッド，同期はスケレトンをそのまま再利用できる．単位計算は逐次プログラムを殆ど再利用し，一部修正している．タスク分割では KMP 法の重複ブロックをブロックに修正し，新

表 4: 3 次スプラインのインデックス

	応用	数値計算		
	仕様	連続する複数の座標点 (x,y) 間の曲線が滑らかになるような座標点を求め，補間を行う.		
	ソースデータ	実数型構造体 1 次元配列 point[ ].x，point[ ].y		
	結果データ	実数型構造体 1 次元配列 result[ ].x，result[ ].y		
△	タスク分割	point[ ].x，y	，要素ブロック	，ブロック
		result[ ].x，y	，要素ブロック	，ブロック
○	終了条件	なし		
○	トポロジ	マスター&ワーカー		
○	アルゴリズム	プロセッサファーム		
○	並列化手法	パラレルリージョン		
○	インタラクション	なし		
○	BACS	C^t		

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．		
	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
○	終了条件	なし		
○	トポロジ	マスター&ワーカー		
○	アルゴリズム	プロセッサファーム		
○	並列化手法	パラレルリージョン		
○	インタラクション	なし		
○	BACS	C^t		

規作成も必要とした．表 6 に 3 次スプラインの事例修正による並列プログラミングの評価を示す．

```
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 );
  結果出力;
}
work{
  THREADno = スレッド番号の取得;
  TEXT のタスク分割;
  for(PAT 数){
    for(k=0; k$\leq$ (分割$TEXT$長); k++){
      単位計算;
} } }
```

図 6: KMP 法のスケルトン

表 6: 3 次スプラインの事例修正による評価

(行数)

	スレッド	同期	タスク分割	単位計算
再利用	8	1	0	107
修正	0	0	11	3
新規	0	0	12	0

(3) 並列効果

1 つの曲線には 120 個の座標点があり，それぞれの座標点間の各区間を 100 個の座標点で補間を行った．8，16，24 の 3 通りで曲線の数を変えて計算を行った．図 7 に 3 次スプラインの並列効果を示す．補間を行う座標点の数を増やすことにより，スレッド操作のオーバヘッドに比べ，各スレッドが行う計算量が増加するために，良い結果を得ている．曲線の数が 24 本でプロセッサ台数が 16 台の時，約 9.9 倍の並列効果が得られた．

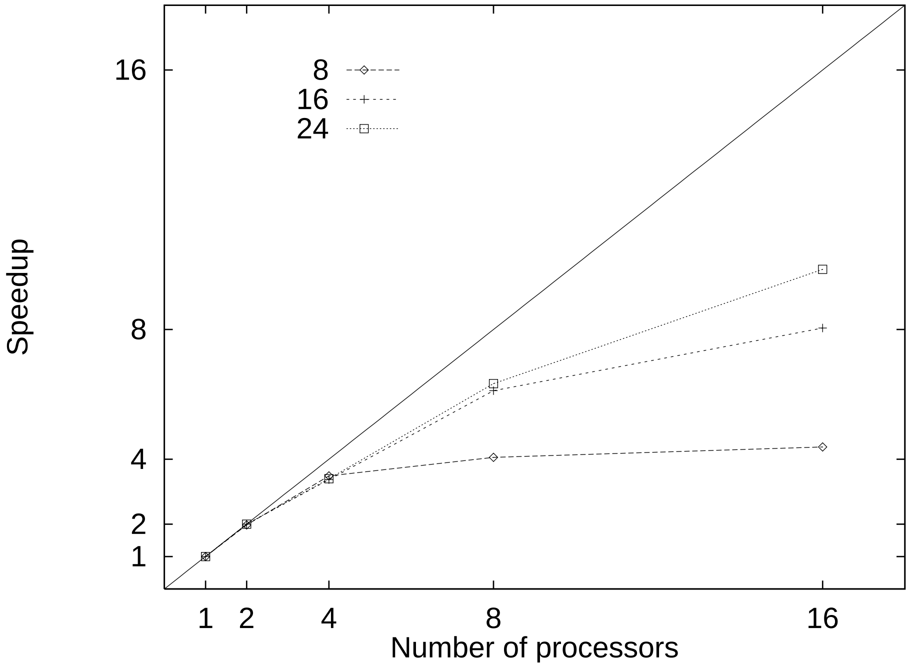


図 7: 3 次スプラインの並列効果

4.2　エッジ抽出

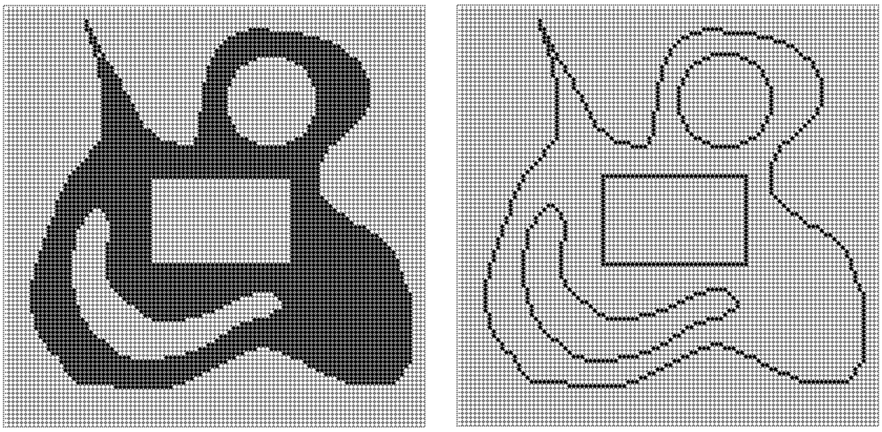
4.2.1　問題の定義と解析

エッジ抽出とは，画像の輪郭となる部分を取り出すことである [19]．図 8 に入力データとなる画像 (a) からエッジを抽出した画像 (b) を表したものを示す．入力画像データの 2 値化を行って，以下のような方法でエッジを抽出する．また図 9 にエッジの抽出パターンを示す．

- X(真ん中の太枠線内) の値が 1 で，Xの上下左右に 1 つでも 0 があれば，Xはエッジとなる．(a)
- Xの値が 0 であれば，Xには物体が存在しないのでエッジも存在しない．(b)
- Xの値が 1 で，Xの上下左右の値がすべて 1 であれば，Xはエッジではないので，Xに 0 を代入する．(c)

4.2.2　並列化

ソースデータ，結果データを整数型 2 次元配列 $indata$，$outdata$ とする．ソースデータの各単位要素の計算負荷は十分にばらついていると考えられるので，ブロックによる分割



(a)入力画像

(b)エッジ抽出画像

図 8: エッジ抽出

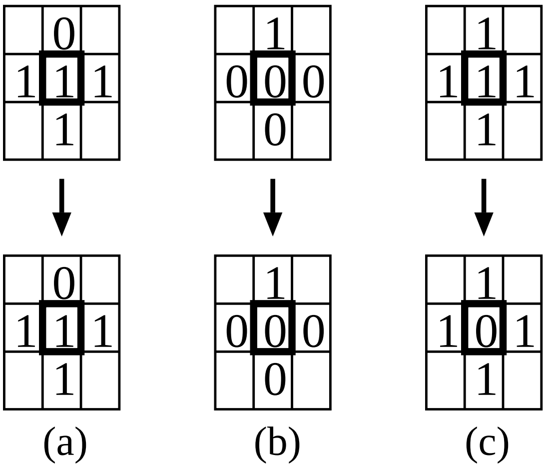


図 9: エッジの抽出パターン

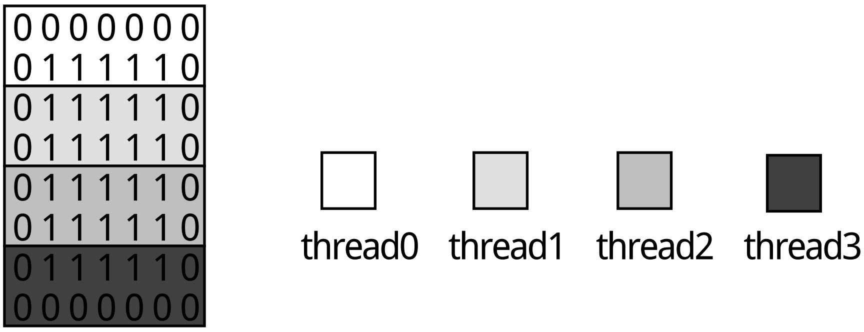


図 10: エッジ抽出のタスク分割

を適用する．複数行から成るブロックに分割し，ブロック毎にスレッドに割り当て，スレッドではそれぞれに割り当てられた範囲でエッジを抽出する．図 10 にエッジ抽出のタスク分割を示す．各スレッドのタスクは他のスレッドのタスクとの依存関係がないので，全スレッドは独立に計算できる．終了条件はなく，また同期は必要としないので，アルゴリズムはプロセッサファームを使用できる．各スレッドがマスターからタスクを受け取り，その計算結果をマスターに返す構造であるので，トポロジはマスター&ワーカーである．全スレッドの計算終了をマスターが待つための同期が必要となるため，並列化手法はパラレルリージョン，全スレッドで同じ計算を行うので BACS 表現は C^t となる．類似事例として 3 次スプラインが検索される．

4.2.3　類似事例 (3 次スプライン) の修正

(1) インデックス

表 7 にエッジ抽出のインデックスを示す．タスク分割はソース/結果データの 2 次元配列を行によるブロックで分割する．応用分野は画像処理となり，アルゴリズムがプロセッサファームであることから，終了条件，インタラクションは共になし，トポロジはマスター&ワーカー，並列化手法はパラレルリージョンとなる．また表 4 に類似事例として検索された 3 次スプラインのインデックスを示す．

表 7: エッジ抽出のインデックス

	応用	画像処理
	仕様	2 値化画像から輪郭となる部分を見つけ出し，抽出する．
	ソースデータ	整数型 2 次元配列 indata[ ][ ]
	結果データ	整数型 2 次元配列 outdata[ ][ ]
△	タスク分割	indata[ ][ ]　　，行ブロック　　，ブロック
		outdata[ ][ ]　　，行ブロック　　，ブロック
○	終了条件	なし
○	トポロジ	マスター&ワーカー
○	アルゴリズム	プロセッサファーム
○	並列化手法	パラレルリージョン
○	インタラクション	なし
○	BACS	C^t

エッジ抽出と 3 次スプラインの相違点はタスク分割だけとなる．3 次スプラインのタスク分割は 1 次元配列の要素ブロックによる分割で，エッジ抽出は 2 次元配列の行による要

素ブロックの分割であるので，基本的には分割方法は同じとなる．よって分割対象を修正する必要がある．またインデックスの他の項目は全て一致するので，スレッド生成部分は再利用でき，終了条件，インタラクションはないので，修正・新規作成の必要はない．

(2) 事例の修正

3 次スプラインの事例を修正してエッジ抽出の並列プログラムを作成する．図 11 に 3 次スプラインのスケルトンを示す．このスケルトンは main からパラレルリージョンを起動して，全スレッドで work を実行する．表 4 と表 7 はソースデータ，結果データのデータ構造が違うだけなので，タスク分割は一部を再利用し，他は新規作成を行った．単位計算部分は逐次プログラムを肉付けし，一部のタスク分割に関わる部分を修正することで作成できた．またスレッド，同期はスケルトンをそのまま再利用できた．表 8 にエッジ抽出の事例修正による並列プログラミングの評価を示す．

```
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 );
    結果出力;
  }
work
  {
    THREADno = スレッド番号取得;
    offset = 全データ数 / スレッド数;
    tmp   = 全データ数 % スレッド数;
    計算開始位置 = スレッド番号 * offset;
    計算終了位置 = 計算開始位置 + offset - 1;
    if (スレッド番号 < tmp)  計算終了位置++;
    for(i=計算開始位置; i<計算終了位置 + 1; i++)
      {
        単位計算;
      }
  }
```

図 11: 3 次スプラインのスケルトン

表 8: エッジ抽出の事例修正による評価

(行数)

	スレッド	同期	タスク分割	単位計算
再利用	8	1	6	60
修正	0	0	0	5
新規	0	0	15	0

(3) 並列効果

ピクセル値が 245x298，320x400 のそれぞれの画像をソースデータとしてエッジ抽出の計算を行った．図 12 にエッジ抽出の並列効果を示す．320x400 の画像でプロセッサが 16 台の時，約 11 倍の速度向上が得られた．計算量に対してスレッド操作のオーバヘッドの占める割合が高くなるため，計算負荷の大きい処理が必要である．

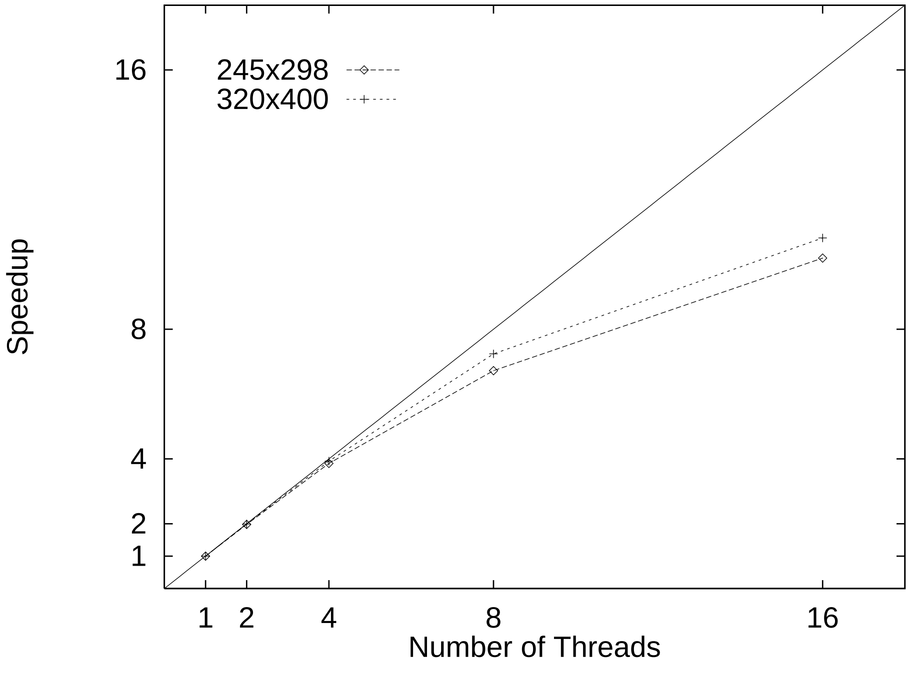


図 12: エッジ抽出の並列効果

4.3　包装アルゴリズム

4.3.1　問題の定義と解析

包装アルゴリズムはある点集合において，すべての点を包み込むことができる任意の点を直線で結んだ凸多角形を求める問題である [20]．図 13 のように点集合の一番端にある点から，すべての点を包み込んでいくように，1 つ 1 つの座標点を順に求めていく．以下のような方法で座標点を求める．

- まず，例えば y 座標が最小の点は求める凸包上にあることが明らかであるので，その点を出発点 (A 点) とする．
- 次に，A 点から水平正方向に線分を引き，A 点と他のすべての点の線分とのなす角度をすべて求める．その求めた角度が最小のものが凸包上の点 (B 点) となる．
- B 点では，A 点を除く他のすべての点と水平正方向とのなす角度を求め，線分 AB と水平正方向とのなす角度より大きく，かつ最小のものが凸包上の点 (C 点) となる．
- 以上のことを繰り返し行い，最終的に A 点に戻れば終了となる．

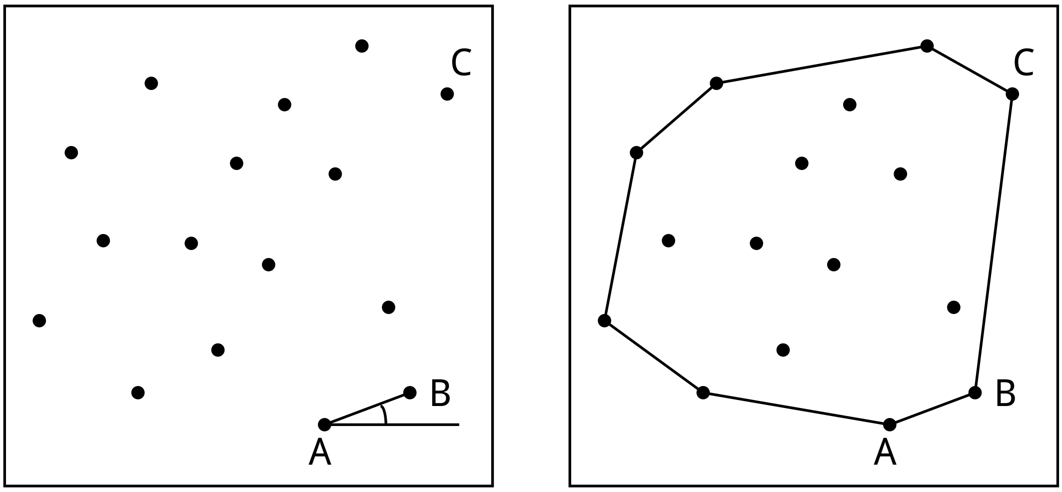


図 13: 包装アルゴリズム

4.3.2　並列化

ソースデータ，結果データを実数型構造体の 1 次元配列 $point[\ ].x$,yとする．任意の点と点集合のすべての点とのなす角度の計算はそれぞれ独立に計算できるので，分割対象はソー

スデータである点集合の座標値とし，計算負荷を均等にするために，サイクリックによる分割を適用する．図 14 に包装アルゴリズムのタスク分割を示す．また結果データはソースデータの先頭から順に格納するので，計算を 1 回繰り返す毎に分割タスクが 1 つ減ることになる．各スレッドでは割り振られるタスクは異なるが，同じ処理を行い，マスターは全スレッドの計算終了を確認後，計算結果の評価を行うのでバリアによる同期が必要となり，計算結果が条件を満たしていなければ，繰り返し計算を行うので，アルゴリズムは繰り返し変換を使用できる．また各スレッドでは計算に必要なデータだけを参照するので，トポロジは部分照合となり，BACS 表現は $Cond\{I^t_{barrier(A/B)}\{C^t\}, C^{p(master)}\}$ となる．ソースデータが 1 次元配列で，1 要素ずつサイクリックにタスク分割を行うことが一致したので，類似事例として高次方程式の解法 (DKA 法) が検索される．

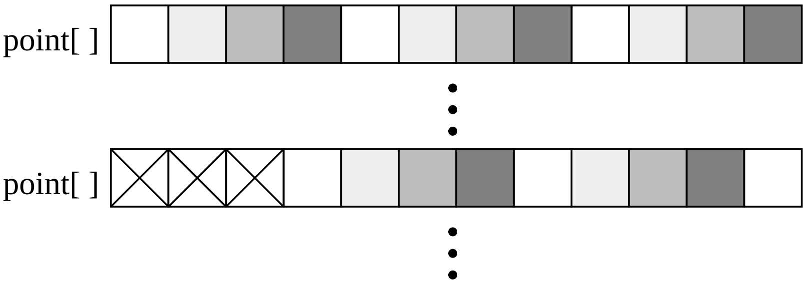


図 14: 包装アルゴリズムのタスク分割

4.3.3 類似事例 (DKA 法) の修正

(1) インデックス

表 9 に包装アルゴリズムのインデックスを示す．タスク分割は 1 要素ずつ順にスレッドに割り振るサイクリックを採用し，トポロジは計算に必要となるデータだけを部分的に参照する部分照合，アルゴリズムは繰り返し変換，インタラクションはバリアを使用している．BACS 表現はバリア A/B を用いて同期を取って全スレッドで同一の計算を行い，マスタースレッドが終了条件の判定を行うことを示している．また表 10 に類似事例として検索された高次方程式の解法 (DKA 法) のインデックスを示す．DKA 法は実根を持つ n 次の多項式から n 個の解 $X_1, X_2, \cdots, X_n$ を同時に求める問題で，X_i の k 番目の近似値 $X_i^{(k)}$ を設定することで逐次近似値を計算する．

包装アルゴリズムと DKA 法の相違点はタスク分割，終了条件，トポロジである．タスク分割は DKA 法が共有変数を用いているのに対して，包装アルゴリズムはサイクリックに分割しているので修正し，また終了条件判定，データ参照部分の修正・新規作成の必要がある．アルゴリズム，並列化手法，BACS 表現は一致するので全体的な骨格は同じと考えられ，インタラクションはバリアを再利用できる．

表 9: 包装アルゴリズムのインデックス

	応用	数値計算
	仕様	点集合において，すべての点を包み込む複数の点を直線で結んだ凸多角形を求める．
	ソースデータ	実数型構造体 1 次元配列 point[ ].x, y
	結果データ	point[ ].x, y
△	タスク分割	point[ ].x, y　　　　，要素　　　　，サイクリック
△	終了条件	始点=終点
△	トポロジ	部分照合
○	アルゴリズム	繰り返し変換
○	並列化手法	パラレルリージョン
○	インタラクション	バリア
○	BACS	$Cond\{I^t_{barrier(A/B)}\{C^t\}, C^{p(master)}\}$

表 10: DKA 法のインデックス

	応用	数値計算
	仕様	$X_i^{(k+1)} = X_i^{(k)} - \dfrac{f(X_i^{(k)})}{\prod\limits_{j=1,j\neq i}^{n} (X_i^{(k)} - X_j^{(k)})}$
	ソ– スデ– タ	実数型 1 次元配列 $X[n]$:　解 実数型 1 次元配列 $a[n]$:　係数
	結果デ– タ	$X[n]$
△	タスク分割	$X[n]$　　　　，　　　　，コピ–
		$a[n]$　　　　，　　　　，コピ–
		$X[n]$　　　　，要素　　　　，サイクリック
△	終了条件	$\lvert X_i^{(k+1)} - X_i^{(k)}\rvert < 10^{-6}$ 又は変換ステップ数 $= 5000$
△	トポロジ	完全照合
○	アルゴリズム	繰り返し変換
○	並列化手法	パラレルリ– ジョン
○	インタラクション	バリア
○	BACS	$Cond\{I^t_{barrier(A/B)}\{C^t\}, C^{p(master)}\}$

(2) 事例の修正

DKA 法の事例を修正して包装アルゴリズムの並列プログラムを作成する．図 16 に DKA 法のスケレトンを示す．このスケレトンは main でパラレルリージョンを起動して worker を呼び出し，worker ではバリア同期が取られて object(単位計算) を実行し，マスタースレッドが変換結果の評価を行っている．スレッド，同期はスケレトンをそのまま再利用でき，タスク分割はスケレトンを修正するだけで実現できた．単位計算は殆ど逐次プログラムを再利用しているが，共有変数の相互排除のために各スレッドでローカル変数の新規作成，修正を必要とした．表 11 に包装アルゴリズムの事例修正による並列プログラミングの評価を示す．

表 11: 包装アルゴリズムの事例修正による評価

(行数)

	スレッド	同期	タスク分割	単位計算
再利用	10	7	0	67
修正	0	0	2	4
新規	0	0	0	18

(3) 並列効果

入力座標点の数を 8,000，16,000，80,000 として包装アルゴリズムの計算を行った．図 15 に包装アルゴリズムの並列効果を示す．座標点が 16,000 個でプロセッサの台数が 16 台の時に，約 10.6 倍，80,000 個で 16 台の時に，約 10.7 倍の速度向上が得られた．

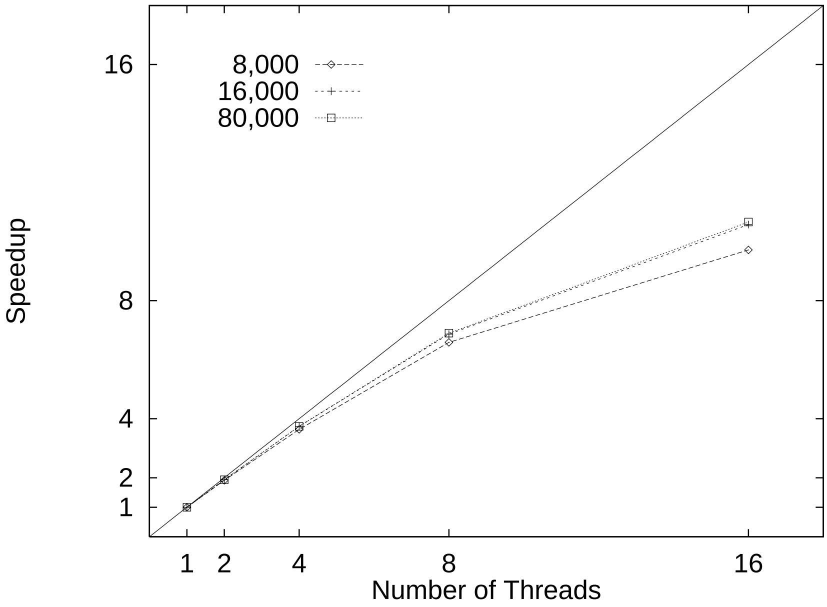


図 15: 包装アルゴリズムの並列効果

```
main {
  スレッド数の入力;
  バリア初期化 ( バリア A );   バリア初期化 ( バリア 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 {
  THREADno = スレッド番号取得;
  ステップ番号 = 0;
  while(1) {
      ステップ番号++;
      バリアチェックイン ( &バリア A, スレッド番号 );
        object( スレッド番号 );
      バリアチェックアウト ( &バリア A, スレッド番号 );
      if ( ( flag == 0 ) && ( ステップ番号 < 最大ステップ数 ) )
        if ( マスタースレッド )
          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++;
      バリアチェックイン ( &バリア B , スレッド番号 );
        object( スレッド番号 );
      バリアチェックアウト ( &バリア B , スレッド番号 );
      if ( ( flag == 0 ) && ( ステップ番号 < 最大ステップ数 ) )
        if ( マスタースレッド )
          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;
}
```

図 16: DKA 法のスケルトン

4.4　巡回セールスマン問題

4.4.1　問題の定義と解析

巡回セールスマン問題は図 17 のように，ある点を始点とし，全ての N個の点をただ一度だけ通って始点に戻る時に，移動距離の総和が最短になるような経路を求める探索法である [21]．すでに通った点はマークを付け，総当たり探索で経路を求めていく．行き止まりになったり，移動距離が今までに記録した最短経路より大きくなった場合は，経路を戻って別の経路を進んでいく．

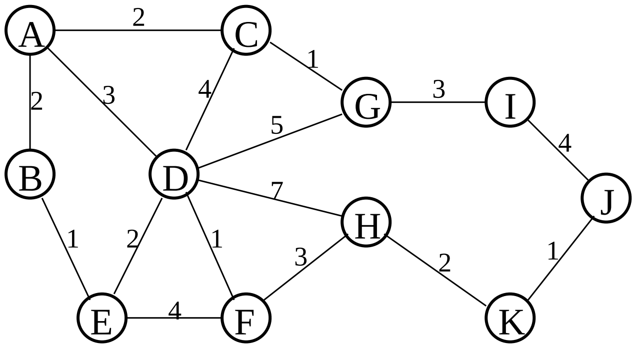


図 17: 巡回セールスマン問題

4.4.2　並列化

ソースデータ，結果データをそれぞれ整数型構造体の 2 次元配列 $data[\ ][\ ].a$,b，整数型 1 次元配列 $result$ とする．探索経路が分岐するとき，1 つの検索開始位置となるタスクをそのスレッド自身に，他方のタスクを他のスレッドに分割する．各スレッドでの計算には全データが必要となるので，ソースデータは共有変数となる．図 18に巡回セールスマン問題のタスク分割を示す．計算結果が得られるまでタスクを分割し，各スレッドに割り振ることを繰り返すので，トポロジはトリーとなり，アルゴリズムは分割統治法が使用できる．アイドルスレッドを検索するときに相互排除を行うので同期にミューテックスを用いる．また全スレッドで共通の関数を実行するので並列化手法はパラレルリージョン，終了条件は全経路の探索完了，BACS 表現は $Cond\{I^{p(i)}_{Wait\ New\_task}, Cond(C^{p(i)}, I^{p(i)}_{Search\ Idle\_thread})\}$ となる．並列化手法，BACS 表現が一致したことから，類似事例としてクイックソートが検索される．

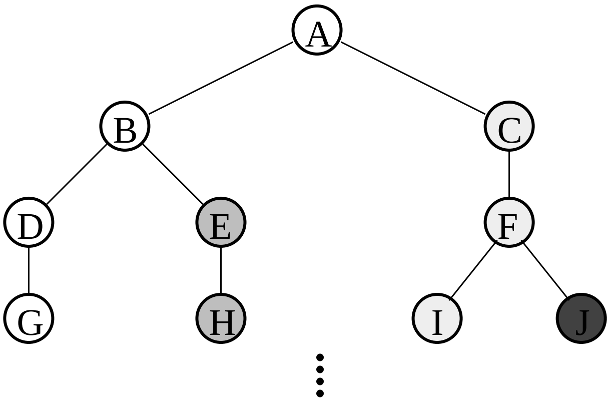


図 18: 巡回セールスマン問題のタスク分割

4.4.3 類似事例 (クイックソート) の修正

(1) インデックス

表 12 に巡回セールスマン問題のインデックスを示す．応用分野は経路探索，タスク分割はソース/結果データ共に各スレッドで全データを参照するので共有となる．アルゴリズムは分割統治法，インタラクションはミューテックスを使用する．BACS 表現は新たなタスクが来るのを待ち，タスクを受け取ったら計算を始め，そしてアイドルスレッドがあれば同期を取ってタスクを渡すということを繰り返し行うことを示している．また表 13 に類似事例として検索されたクイックソートのインデックスを示す．クイックソートは要素列中からサンプル要素を 1 つ選び，それを基準に値がそれより小さいグループと大きいグループに分割し，各々のグループでこれを再帰的に適用することでソートする問題である．

巡回セールスマン問題とクイックソートの相違点はタスク分割と終了条件である．クイックソートではタスクを要素ブロックで分割しているのに対して，巡回セールスマン問題はタスクを共有変数にしている．タスク分割，終了条件共に修正･新規作成が必要となる．アルゴリズム，並列化手法，BACS 表現の項目は一致するので，全体的な骨格は同じであると考えられる．インタラクションの項目も一致するので再利用できる．

(2) 事例の修正

クイックソートの事例を修正して巡回セールスマン問題の並列プログラムを作成した．図 20 にクイックソートのスケレトンを示す．このスケレトンは main でパラレルリージョンを起動して，全スレッドで thread_quick を実行する．そして thread_quick ではタスク待ちのシグナルが変われば quick の処理を始める．アクティブスレッド数，スレッドのアクティブ状態，タスク割り当ての一貫性を保持するためにミューテックスによる同期が取られている．表 14 に巡回セールスマン問題の事例修正による並列プログラミングの評価を

表 12: 巡回セールスマン問題のインデックス

	応用	経路探索
	仕様	ある点から全ての点をただ一度だけ通って始点に戻る時の最短移動経路を求める．
	ソースデータ	整数型構造体 2 次元配列 data[ ][ ].a，data[ ][ ].b
	結果データ	整数型 1 次元配列 result[ ]
△	タスク分割	data[ ][ ].a,b ， ，共有
		result ， ，共有
△	終了条件	全経路の探索完了
○	トポロジ	トリー
○	アルゴリズム	分割統治法
○	並列化手法	パラレルリージョン
○	インタラクション	ミューテックス
○	BACS	$Cond\{I^{p(i)}_{Wait\ New\_task}, Cond(C^{p(i)}, I^{p(i)}_{Search\ Idle\_thread})\}$

表 13: クイックソ-トのインデックス

	応用	ソ-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．
	ソ-スデ-タ	整数型 1 次元配列 $data[L] \sim data[R]$
	結果デ-タ	$data[L] \sim data[R]$
△	タスク分割	$data$ ，要素ブロック ，ブロック
△	終了条件	全ての部分要素列がソ-ト完了
○	トポロジ	トリ-
○	アルゴリズム	分割統治法
○	並列化手法	パラレルリ-ジョン
○	インタラクション	ミュ-テックス
○	BACS	$Cond\{I^{p(i)}_{Wait\ New\_task}, Cond(C^{p(i)}, I^{p(i)}_{Search\ Idle\_thread})\}$

示す．全体的な構造が同じであったので，スレッド操作全部と，同期の一部を再利用できた．他の同期は共有変数への読み出し/書き込みのために新規作成を行い，タスク分割も新規作成を必要とした．単位計算は，共有変数へ書き込む前に各スレッド毎でローカル変数に書き込む必要があったために，一部を修正・新規作成を行った．

表 14: 巡回セールスマン問題の事例修正による評価 (行数)

	スレッド	同期	タスク分割	単位計算
再利用	28	8	0	75
修正	0	0	0	15
新規	0	7	8	7

(3) 並列効果

訪問場所数を 144，196 箇所の 2 通りで最短経路探索を行った．図 19 に巡回セールスマン問題の並列効果を示す．訪問場所数 196 でプロセッサ台数が 16 台の時，約 5.8 倍の並列効果が得られた．共有変数のための同期のミューテックスが多くなったが，各スレッドでの実行時間を小さくしてアイドルスレッドを少なくしたので，良い結果が得られている．

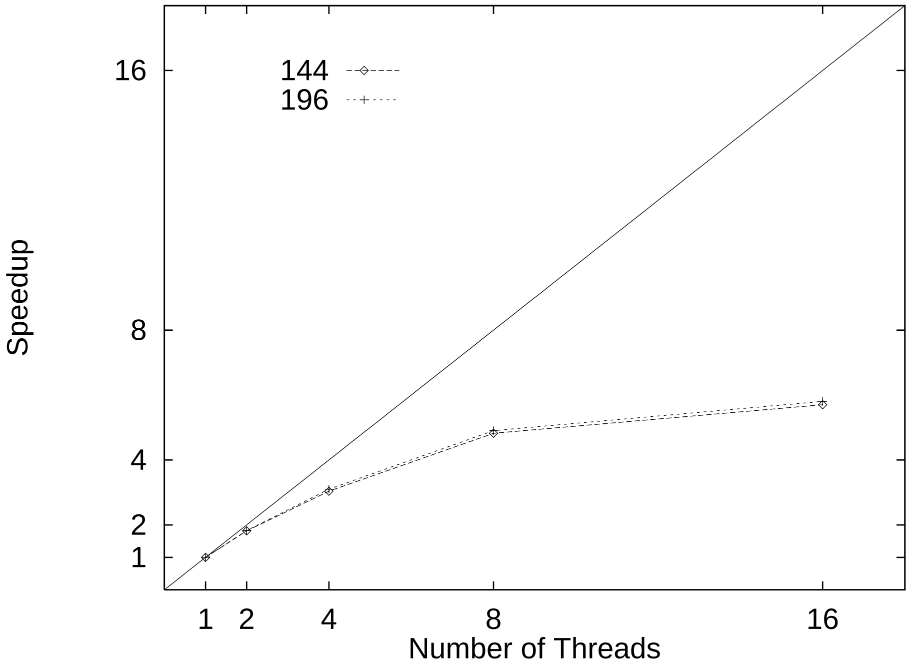


図 19: 巡回セールスマン問題の並列効果

```
main{
    スレッド数の入力;    ミューテックス初期化;    アクティブスレッド数 = 1;
    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, thread_quick );
    ソート結果を出力;
}
thread_quick{
    THREADno = スレッド番号の取得;
    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);
}
```

図 20: クイックソートのスケルトン

4.5 事例修正並列プログラミングの評価と考察

4 つの問題について事例修正による並列プログラミングを行った．4 つの事例修正による評価を見ると，スレッド操作の部分を殆ど再利用できているのが分かる．これは同じ並列アルゴリズムであれば，全体的な並列構造が似ているからだと考えられる．また同期の部分も基本的な部分は再利用でき，問題に応じて必要な部分は付け加えている．タスク分割は再利用できているのが少なく，殆どの場合修正・新規作成が必要となっている．これはデータ構造と分割方法のどちらか 1 つでも異なれば修正が必要となるからである．単位計算の修正・新規作成を行っているのは，殆どの場合タスク分割，参照データを共有変数としているところに関係がある．タスク分割の場合は計算範囲を示す変数を，共有変数の場合は各スレッド毎にローカル変数を必要としているために修正・新規作成を行っている．

並列プログラミングを行う上で最も重要な部分はタスク分割で，事例修正による評価を見てもタスク分割に一番労力を費やしているのが分かる．しかしタスク分割はデータ構造毎である程度の分割パターンが決まり，データ構造も基本的には 1・2 次元配列を使用している．そこでタスク分割をデータ構造と分割方法で分類して，それぞれのスケレトン，あるいは並列プログラムを用意することでタスク分割を支援し，並列プログラミング全体を軽減することができると考えられる．

3 次スプライン，エッジ抽出，包装アルゴリズム，巡回セールスマン問題のそれぞれの並列プログラムを付録に示す．プログラムの行頭に書かれている記号は，S がスケレトンを再利用，MS がスケレトンを修正，MP が逐次プログラムを修正，N が新規作成，空白が逐次プログラムを再利用していることをそれぞれ示す．

5　並列実行の可視化

5.1　全体構成と機能

並列実行の可視化は実行時間，タスク開始・終了時刻，同期を取った時刻を各プロセッサ毎に計測してファイルに収め，そのファイルを用いて実行状況を表示する．図 21 に巡回セールスマン問題の並列実行の可視化を示す．左側に各プロセッサの番号，中央に実行状況を可視化したもの，右側には各プロセッサの稼働率をそれぞれ表示している．稼働率は各プロセッサにおける実行時間の合計を全実行時間で割った割合を示している．この並列実行の可視化は Tcl/Tk プログラミング [22] で実現している．

実行状況を可視化する機能には以下のようなものがある．

- 可視化対象のタイトル名，並列アルゴリズム，全実行経過時間の表示．
- プロセッサ間の負荷均衡を検討するための稼働率の表示．
- 実行状況の細かい部分を見るための拡大表示．
- 各並列アルゴリズムの一般的な構造の表示．

図 21: 巡回セールスマン問題の並列実行の可視化

5.2 実行状況データの取得方法と表示方法

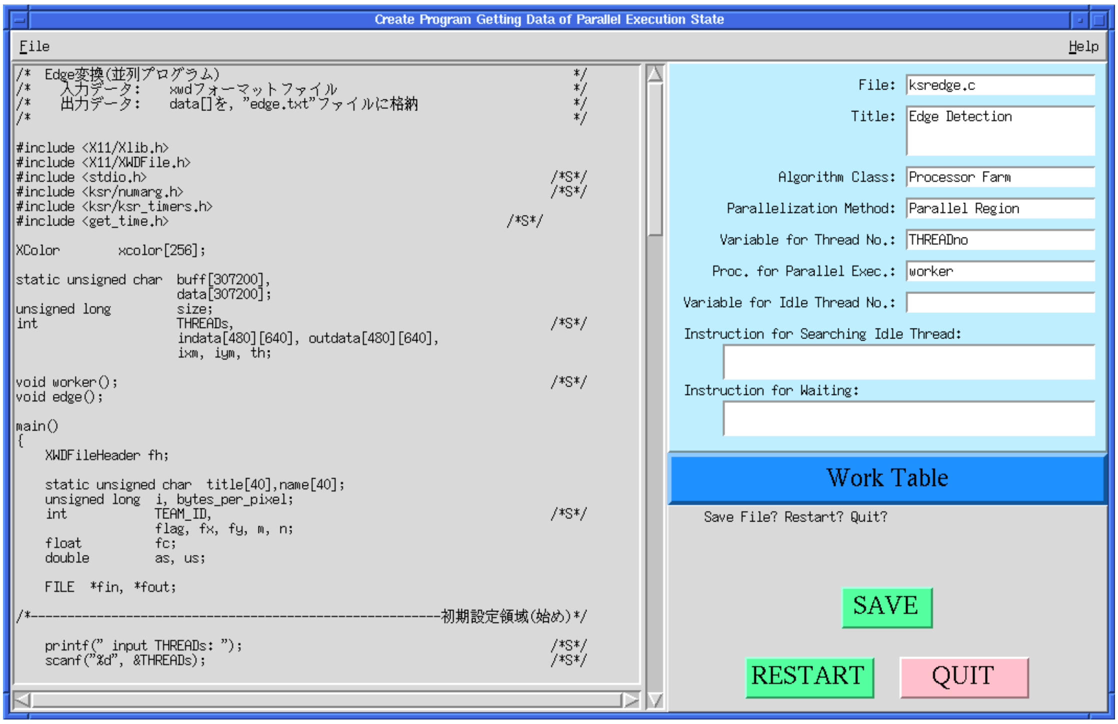
実行状況を表示するためには，実行，待機，同期を取った時のそれぞれの時刻とスレッド番号のデータが必要となる．それらのデータを取得する命令を挿入した並列プログラムを実行することにより，表示データを作成する．挿入する命令を簡単化するためにヘッダファイルを用意している．

ヘッダファイルには実行状況を取得する10個の関数があり，ファイルの操作や，時刻・スレッド番号のファイルへの書き込みなどを行う．この関数を呼び出すことで実行状況を取得し，表示データファイルを作成する．表15に各関数の機能を示す．実行する主な命令は，ファイル操作，計算開始・終了時刻，同期を取った直前・直後の時刻，スレッド番号などの取得がある．

表15: ヘッダファイルの関数の機能

time_write(int)	スレッド番号と時刻をファイルへ書き込む．
start_seconds()	実行状況取得の決定，ファイルのオープン，実行開始時刻の取得を行う．
stop_seconds()	実行終了時刻の取得，ファイルのクローズを行う．
divide_time1(int)	スレッドの単位計算開始・終了時刻を取得する． (分割統治法で使用する．)
divide_time2(int)	分割タスクを生成したスレッド番号と分割タスクが渡されたアイドルスレッド番号を取得する． (分割統治法で使用する．)
thread_start()	スレッドの計算開始時刻を取得する． (プロセッサファーム，繰り返し変換で使用する．)
thread_stop(int)	スレッドの計算終了時刻を取得する． (プロセッサファーム，繰り返し変換で使用する．)
pro_net_time(int)	スレッドの単位計算終了時刻，下位スレッドとの同期を取った直後の時刻，上位スレッドとの同期を取った直後の時刻を取得する． (プロセスネットワークで使用する．)
barrier_time1(int)	バリア同期を取る直前の各スレッドの単位計算終了時刻を取得する． (繰り返し変換で使用する．)
barrier_time2(int)	バリア同期を取った直後の各スレッドの単位計算開始時刻を取得する． (繰り返し変換で使用する．)

ユーザがデータ取得命令を挿入する場所を解析するのは困難で，並列プログラムを作成する度にこの作業をするのは大変に時間の無駄である．そこでデータ取得命令の挿入を支援するツールを用意した．図 22 に実行状況データの取得支援ツールを示す．このツールは並列プログラムのファイル名，並列アルゴリズム，スレッド番号取得変数などをユーザから対話的に問い合わせて，挿入する命令と挿入場所を決定し，実行状況データを取得する並列プログラムを作り上げる．図 22 の左側は並列プログラムを編集するエディタ，右下の Work Table は問い合わせ情報のエントリなどを行うユーザの作業領域である．右上の 9 個の項目はシステムがユーザから問い合わせた情報を表示する．表 16 にシステムが問い合わせる項目の説明をする．システム側は主に並列アルゴリズム，スレッド番号取得変数名，並列実行関数名などの情報を必要とする．また項目が空白となっているのは，並列アルゴリズムによって実行構造が変わるので，問い合わせ情報が並列アルゴリズム毎で必要となる場合とそうでない場合があるからである．



```
/*  Edge変換(並列プログラム)                                                */
/*     入力データ:    xwdフォーマットファイル                                */
/*     出力データ:    data[]を, ”edge.txt”ファイルに格納                     */
/*                                                                          */

#include <X11/Xlib.h>
#include <X11/XWDFile.h>
#include <stdio.h>                                                       /*S*/
#include <ksr/numarg.h>                                                  /*S*/
#include <ksr/ksr_timers.h>
#include <get_time.h>                                              /*S*/

XColor        xcolor[256];

static unsigned char  buff[307200],
                      data[307200];
unsigned long         size;
int                   THREADs,                                           /*S*/
                      indata[480][640], outdata[480][640],
                      ixm, iym, th;

void worker();                                                           /*S*/
void edge();

main()
{
    XWDFileHeader fh;

    static unsigned char  title[40],name[40];
    unsigned long  i, bytes_per_pixel;
    int            TEAM_ID,                                              /*S*/
                   flag, fx, fy, m, n;
    float          fc;
    double         as, us;

    FILE  *fin, *fout;

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

    printf(” input THREADs: ”);                                          /*S*/
    scanf(”%d”, &THREADs);                                               /*S*/
```

File: ksredge.c
Title: Edge Detection
Algorithm Class: Processor Farm
Parallelization Method: Parallel Region
Variable for Thread No.: THREADno
Proc. for Parallel Exec.: worker
Variable for Idle Thread No.:
Instruction for Searching Idle Thread:
Instruction for Waiting:

Save File? Restart? Quit?

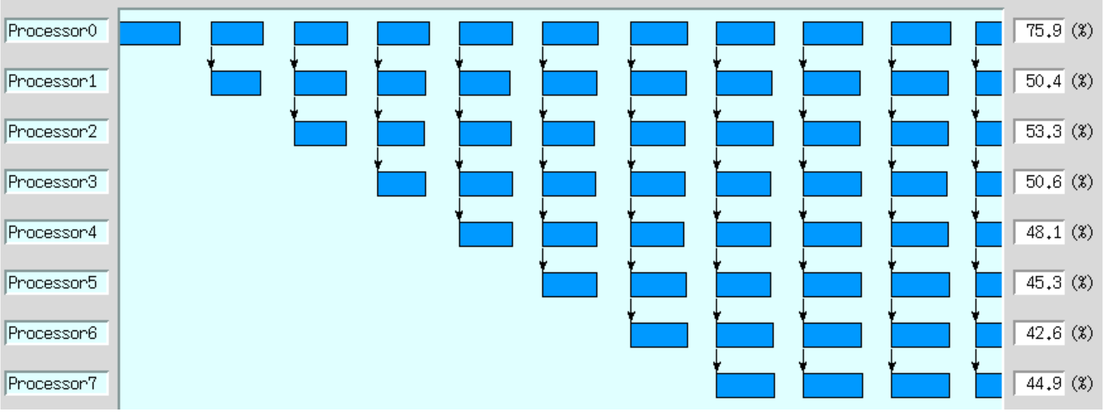
図 22: データ取得支援ツール

表 16: システムの問い合わせ項目の説明

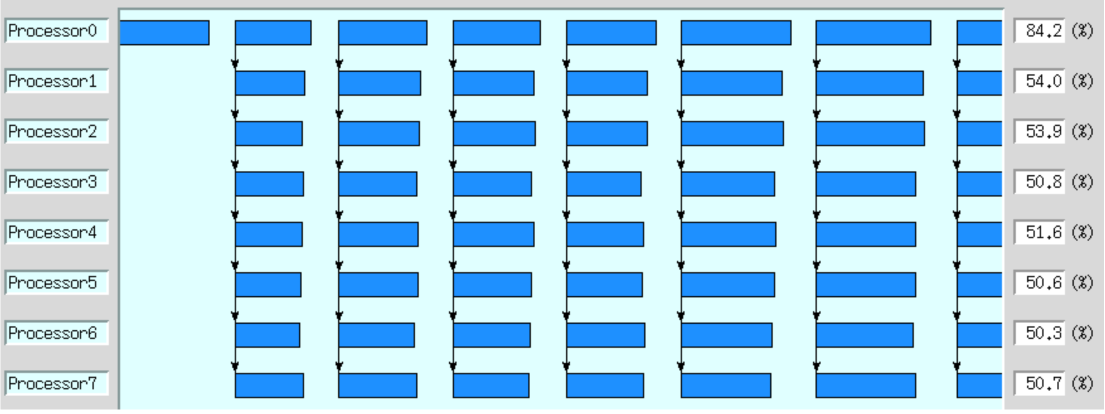
File	並列プログラムのファイル名．
Title	並列プログラムのタイトル．
Algorithm Class	並列アルゴリズム．
Parallelization Method	並列化手法．
Variable for Thread No.	スレッド番号を取得する変数名．
Proc. for Parallel Exec.	各スレッドで並列実行する関数名． (分割統治法，プロセッサファーム，繰り返し変換で必要．)
Variable for Idle Thread No.	アイドルスレッド番号を取得する変数名． (分割統治法で必要．)
Instruction for Searching Idle Thread	アイドルスレッドを探索するコマンド列． (分割統治法で必要．)
Instruction for Waiting	タスクが渡されるまで待つコマンド列． (分割統治法で必要．)

実際に並列プログラムを実行し，取得した実行状況データは並列実行可視化ツールで表示する．このツールも並列プログラムのタイトル名，ファイル名，並列アルゴリズムなどをユーザから問い合わせて表示する．表示するものとしては実行，待機，同期がある．図 23 に実行状況の表示例を示す．(a) はプロセッサファームによる 3 次スプライン，(b) はプロセスネットワークによるバブルソート，(c) は繰り返し変換による包装アルゴリズムをそれぞれ可視化したものである．稼働率を見ると，3 次スプラインが大変に高い割合を示しており，スレッドが無駄なく計算しているのが分かる．またバブルソート，包装アルゴリズムは全スレッドで何回も同期を取っているので，待機時間が蓄積し割合が低くなっているが，各スレッドのタスクの負荷均衡が取れているのが分かる．

(a) 3次スプライン (プロセッサファーム)



(b) バブルソート (プロセスネットワーク)



(c) 包装アルゴリズム (繰り返し変換)

図 23: 実行状況の表示例

5.3 初期評価と考察

並列実行状況を可視化することで，全体的な実行構造，各プロセッサの負荷均衡，同期を取った時のタイミング，同期の対象となるプロセッサ番号などが分かり，並列プログラムの検討，デバッグを支援することができる．全体の構造を見ることで，ユーザ自身が間違った振る舞いをするプロセッサを見つけることができたり，各プロセッサの実行状態を見て，タスク分割はうまく行っているのか検討することができる．

しかしユーザが可視化表示を見てデバッグしたい場所を見つけても，それがプログラム中ではどこになるのか特定が難しい場合もある．これは同じ単位計算部分を繰り返し実行していたり，分割タスクを次々に計算したりするので，システム側でも判断が難しい．そこでユーザがプロセッサの負荷均衡が取れていないのでどうすれば良いかという要求に対して，システムがいくつかのタスク分割方法を提示したりすることで，対話的に助言を与えていくことが考えられる．またユーザが可視化表示で特定の場所を指定することで，システムがそれはどこを指しているのかをスケレトンを用いてその場所を知らせることも考えられる．

プログラム中のデバッグ場所を特定するのは難しいが，おおよその場所を予測・検出することはできるかも知れない．そのために，スレッドの単位計算の最初・終わりの所，タスク待ちをしている所，あるいはスレッド操作を行っている所などをシステム側がデバッグ場所を予め区切っておく．そしてユーザがデバッグ場所を指定した時，それらが並列プログラムで対応する場所を検出することによって，デバッグ場所の特定ができることも考えられる．

6　おわりに

プロセッサファームによる3次スプラインとエッジ抽出，繰り返し変換による包装アルゴリズム，分割統治法による巡回セールスマン問題のそれぞれについて事例修正による並列プログラミングを行った．そしてスレッド操作，同期，タスク分割，単位計算のそれぞれについて事例修正による作成を評価し，スレッド操作と同期の殆どを事例から再利用できたことを述べた．事例修正ではタスク分割が最も時間を要したので，タスク分割の修正が容易にできれば並列プログラミングを軽減できると考え，タスク分割方法のユーザへの提示などについて考察を行った．またトポロジ，同期，スレッド操作などの動的な振る舞いもユーザには理解が難しいので，動作，特徴，プログラム例などを提示することについて検討した．

並列処理はプロセッサなどの動的な振る舞いを把握するのが難しいので，プロセッサの実行，待機，同期のそれぞれを可視化表示することで，並列処理の概念の理解や並列プログラミングを支援するツールを作成した．各プロセッサの負荷均衡が取れているか判断するための稼働率の表示，並列実行状況の細かい部分を見るための拡大表示などの機能を用意した．また，並列実行状況を表示するデータは並列プログラムに命令を書き込んで，実際にそれを実行することで取得する必要がある．そこでこのプログラムの作成を支援するためのツールを用意した．ユーザと対話的に命令を挿入することで，可視化データを取得する並列プログラムを作り上げる．

今後の課題としては，タスク分割方法や，スレッド，同期などの動的な振る舞いを示した並列構造と，その特徴，プログラム例などを表示することで，ユーザに並列プログラミングのアドバイスを提示することが挙げられる．また，事例検索部とそのユーザインタフェース部の実現も挙げられる．

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に心より深く感謝の意を表します．また，本研究に際し，共に研究を行ってきた安藤彰一氏，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] 松原仁 :" 推論技術の観点からみた事例に基づく推論", 人工知能学会誌, Vol.7, No.4, pp.567-575, Jul. 1992.

[2] 河野毅, 末田直道, 田中利一 :" CBR システムの構築環境", 人工知能学会誌, Vol.9, No.1, pp.91-99, Jan. 1994.

[3] 小林康弘 : " 事例ベース推論 ", 人工知能学会誌, Vol.8, No.1, pp.26-36, Jan. 1993.

[4] Ian D. Watson :"Progress in Case-Based Reasoning", First United Kingdom Workshop Salford, UK, Springer, Jan. 1995.

[5] 笠原博徳 : " 並列処理のためのシステムソフトウェア ", 情報処理, Vol.34, No.9, pp.1134-1138, Sep. 1993.

[6] 菊地純男 :" 並列化支援システム", 情報処理, Vol.34, No.9, pp.1158-1169, Sep. 1993.

[7] 本多弘樹 : " 自動並列化コンパイラ ", 情報処理, Vol.34, No.9, pp.1150-1157, Sep. 1993.

[8] 山田剛 : " 並列処理システムにおけるプログラムデバッギング ", 情報処理, Vol.34, No.9, pp.1170-1178, Sep. 1993.

[9] M.T.Heath, A.D.Malony, and D.T.Rover : " The Visual Display of Parallel Performance Data ", Computer, Vol.28, No.11, pp.21-28, Nov. 1995.

[10] M.T.Heath, A.D.Malony, and D.T.Rover : " Parallel Performance Visualization : From Practice to Theory", IEEE Parallel and Distributed Technology, Vol.3, No.4, pp.44-60, Winter 1995.

[11] 田中信一, 飯塚肇 : " スレッドを用いた並列プログラミングのためのソフトウェアモニタリングツール ", 成蹊大学工学研究報告, Vol.31, No.2, pp.99-102, 1994.

[12] 久保田和人, 小柳滋 : " 並列プログラムを多角的に表示する可視化ツール ", 並列処理シンポジウム *JSPP'96*, pp.259-266, Jun. 1996.

[13] 笹倉万里子, 木和田智子, 城和貴, 荒木啓二郎 :" 並列化支援視覚化システム NaraView におけるプログラム情報の 3 次元表示法", 並列処理シンポジウム *JSPP'96*, pp.267-274, Jun. 1996.

[14] Burkhart Helmar et al. :" BACS: Basel Algorithm Classification Scheme", Technical Report *93-3*, Universität Basel, 1993.

[15] Kendall Square Research ：”KSR Parallel Programming”，1992．

[16] Kendall Square Research ：”KSR C Programming”，1993．

[17] Canon Supercomputing S.I. Inc. ：”C ユーザーのための KSR 仕様手引書”，Nov. 1994．

[18] 小池慎一 ：”C による科学技術計算”，pp.189-197，CQ 出版社 (1990)．

[19] 長尾真 ：”画像認識論”，pp.45-59，コロナ社 (1983)．

[20] R. セジウィック著，野下浩平 ・星守 ・佐藤創 ・田口東共訳 ：”アルゴリズム [原書第 2 版] 第 2 巻”，pp.185-191，近代科学社 (1992)．

[21] R. セジウィック著，野下浩平 ・星守 ・佐藤創 ・田口東共訳 ：”アルゴリズム [原書第 2 版] 第 3 巻”，pp.111-116，近代科学社 (1993)．

[22] 宮田重明，芳賀敏彦 ：”Tcl/Tk プログラミング入門”，オーム社 (1995)．

[23] 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．

[24] 松田浩一，山崎勝弘 ：”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”，情報処理学会第 *52* 回全国大会，*3L-4*, 6-113, 1996．

[25] 安藤彰一，松田浩一，山崎勝弘 ：”並列プログラミング用事例ベースの作成”，第 *40* 回システム制御情報学会研究発表講演会，4002, 1996．

[26] 山崎勝弘，松田浩一，安藤彰一 ：”並列プログラミング用事例ベースの構築”，信学技法，*CPSY96-39*, pp.1-6, 1996．

[27] 山崎勝弘，松田浩一，安藤彰一 ：”類似事例の修正による並列プログラミング”，情報処理学会第 *53* 回全国大会，*2E-4*, 1-327, 1996．

[28] 山崎勝弘，松田浩一，安藤彰一，朝倉啓之 ：”事例ベース並列プログラミング”，第 *38* 回プログラミングシンポジウム，Jan. 1997.

[29] 朝倉啓之，安藤彰一，山崎勝弘 ：”事例修正による並列プログラミングと並列実行の可視化”，情報処理学会第 *54* 回全国大会，*6P-5*, 1997．

[30] 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システム学専攻修士論文，1995.

[31] 松田浩一 :” 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”，立命館大学大学院理工学研究科情報システム学専攻修士論文，1996.

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

付録