内容梗概

並列プログラミングでは, 負荷均衡の実現とプロセッサ間の情報授受のオーバーヘッドの軽減が最も重要である. そのためにはプロセス間の同期, 相互排除, スレッドなど並列プログラミングに関する知識が必要となる. 本論文では, 過去の代表的な並列プログラムを事例として蓄積し, 新たな並列プログラムを作成する際にその中から最も類似した並列プログラムを検索/再利用することによって並列プログラミングの負担を軽減させる方法について述べる.

仮想共有メモリ並列マシン上でのプログラミングは, 従来の分散メモリ型マシンでのプログラミングに比べて容易であるとは言え, 物理的に複数のプロセッサを使用していることに変わりはなく並列プログラミング独自の問題を含んでいる. 各プロセッサ間の通信は共有変数を用いることにより実現されるが, その同期の取り方などはユーザが陽に記述するため, 各問題の適切な解と並列効果を得るためにはある程度の同期, 相互排除, スレッドに対する知識が必要となる.

そこで本研究では事例ベース推論の考え方を参考にした並列プログラミング支援システムの検討を行っている.CBR は問題解決に過去の事例を有効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, 各種設計支援などのシステムが構築されている.

本手法では並列プログラムの骨格をスケレトンとして用意する. 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/スケレトン/プログラム本体/履歴/並列効果 } を一つの事例として事例ベースに格納し, 新たな問題に対して再利用可能な類似事例を事例ベースから検索する. 検索されたスケレトンに自動/手動で肉付けを行ってプログラムを生成することを目指す. インデックスは { 応用分野/問題の定義/並列アルゴリズム/変数/終了条件/同期方法/並列化手法 } の要素を持つ. スケレトンにはタスク分割, 相互排除, 同期の取り方, スレッド使用法について記述されている. 履歴にはその事例の参照頻度, 事例の再利用の成功・失敗, 事例ベースの更新について記録される. その事例が過去にどの程度の実行時間を要し, どの程度の並列効果があったのか, またどの程度のプロセッサ数で並列効果の限界があったのかを知り, 新たな並列プログラミングの参考にするため, 並列効果を事例の一部として蓄積しておく.

インデックスによって検索された事例のスケレトンに, 自動的な事例の修正/派生的類推による修正/ユーザによる修正のいずれかの方法によって各問題に適した形に修正を施し, 新たなプログラムが生成されることを目指す.

目 次

図 目 次

1 はじめに

仮想共有メモリ (virtual shared memory) 方式の並列マシン [1] は, 物理的には分散したプロセッサ配置でありながら, 共有メモリプログラミングが可能であり, かつ拡張可能が容易である. 従来の分散メモリ型並列マシンと共有メモリ型並列マシンには, それぞれ利点と欠点があった. 分散メモリ型並列マシンは システムの拡張が比較的容易であるが, プロセッサ間の通信をプログラマが陽に記述する必要があるため, 一旦 データ送受信部でのプログラミングミスを犯すとデータの停滞個所を見つけるのが難しいなど, プログラミング ・デバッギングが困難であった. 一方, 共有メモリ型並列マシン上でのプログラミングは, 分散メモリ型のそれに比べて比較的逐次プログラミングに近いが, システムの拡張は困難であった. そこで, 両者の利点を兼ね備えた仮想共有メモリ型マシンが登場することになり, 単一の仮想アドレス空間 ・自動アドレス変換 ・データ管理の透過性により, 分散メモリ型マシン同様のスケーラビリティを持ち, かつ共有メモリ型マシン並みに物理的に分散したプロセッサを意識することなくプログラミングを行うことが可能になった. このような仮想共有メモリ型マシンは 現在商用化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.

しかし, 仮想共有メモリ並列マシン上でのプログラミングは, 従来の分散メモリ型マシンでのプログラミングに比べて容易であるとは言え, 物理的に複数のプロセッサを使用していることに変わりはなく, 並列プログラミング独自の問題を含んでいる. 各プロセッサ間の通信は共有変数を用いることにより実現されるが, その同期の取り方などはユーザが記述するため, 各問題の適切な解と並列効果を得るためにはある程度の同期 ・相互排除 ・スレッドに対する知識が必要となる.

そこで本研究では事例ベース推論の考え方を参考にした並列プログラミング支援システムの提案を行っている [2] . 事例ベース推論 (Case-Based Reasoning:CBR) の発想は記憶と想起に関する認知科学的研究に求められる [3].CBR は問題解決に過去の事例を有効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 [4][5] などのシステムが構築されている.

並列プログラミングでは, 負荷均衡の実現とプロセッサ間の情報授受のオーバーヘッドの軽減が最も重要である. そのためにはプロセス間の同期, 相互排除, スレッドなど並列プログラミングに関する知識が必要となる. 本研究では, 過去の代表的な並列プログラムを事例として蓄積し, 新たな並列プログラムを作成する際にその中から最も類似した並列プログラムを検索し, それを再利用することによって並列プログラミングの負担を軽減させる方法について検討する.

本手法では並列プログラムの骨格をスケレトンとして用意する. 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 ・スケレトン ・プログラム本体 ・履歴 ・並列効果を一つの事例として事例ベースに格納し, 新たな

問題に対して再利用可能な類似事例を事例ベースから検索する. 検索されたスケルトンに自動 / 手動で肉付けを行ってプログラムを生成することを目指す. インデックスは応用分野・問題の定義・並列アルゴリズム・変数・終了条件・同期方法・並列化手法 の要素を持つ. スケルトンにはタスク分割・相互排除・同期の取り方・スレッド使用法について記述されている.

一般的に事例は再利用可能な状態で蓄積されていなければならない. 類似した過去の並列プログラムが事例として存在していてもそれを効率良く検索することができなければ, 事例を再利用することは困難である. また, 検索された事例が与えられた時にその事例が全く抽象化されていないもの, つまり過去の並列プログラムそのものであっては類似事例を再利用することは容易ではない. そこで本手法では, 8 つの要素を持つインデックスを作成し, 事例の一部として保持することによって事例の検索を可能にする. また, 並列プログラムそのものも事例の中に含めておくが, それだけでなく並列プログラム独特の部分を並列プログラムの骨格として記述しておくことによって新たな並列プログラミングを行う際に事例の再利用を容易にする. 並列プログラムの骨格であるスケルトンはスレッドの扱い方, 共有変数の同期, タスク分割などと共に, 変数初期化, 計算部, 計算結果の回収などの一連のプログラミングプロセスが記述されている. 履歴にはその事例の参照頻度, 事例の再利用の成功・失敗, 事例ベースの更新について記録される. また, その事例が過去にどの程度の実行時間を要したのか, どの程度の並列効果があったのか, またどの程度のプロセッサ数で並列効果の限界があったのかを知り, 新たな並列プログラミングの参考にするために並列効果を事例の一部として蓄積しておく.

インデックスによって検索された事例のスケルトンに自動的な事例の修正・派生的類推による修正・ユーザによる修正のいづれかの方法によって各問題に適した形に修正を施し新たなプログラムが生成されることを目指す.

並列プログラムを作成するのにはそれなりの知識が必要であり, またある程度の知識を持ち合わせている場合であっても実際にプログラミングを行うとなると戸惑う場面に出会うことも多い. ようやく解決することが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とき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. このような並列プログラミングの非効率的な作業を避けるのが, 本支援システムの目的である.

本論文では, 第 2 章で一般的な並列プログラミングの分類と類似事例を用いた並列プログラミング支援システム, 第 3 章で事例のインデックスの作成に必要な並列アルゴリズムの分類, 第 4 章で事例の一般形式と例, 第 5 章では並列効果のいくつかの例, 第 6 章では事例の検索と修正方法について述べる.

最後に関連した研究について述べる.BACS によるアルゴリズムの分類に基づいて, スケルトン指向プログラミング (SOP:Skel eton-Oriented Programming) が提案されている [6]. アルゴリズムスケルトンはそこに計算部分を補うことによって, 最終プログラムとなる実行・同期のパターンである. プロセス間の同期はアルゴリズムの主要構造に埋め込ま

れている. まず, プログラマは主要なアルゴリズムスケレトンを保持したデータベースを外観して, 適当なスケレトンを選ぶ. アルゴリズムの記述はスクリプト言語によっている. 次に, その記述はスケレトンジェネレータにより, プログラムスケレトンに変換される. プログラムスケレトンは仮想マシン上のある言語に対するアルゴリズムを示す. 最後に, プログラマによりプログラムスケレトンに必要な情報が補われてプログラムが完成する.

SOP は本手法で検討する手法と極めて近いが, プログラムの移植性とスケレトンの検索の点で異なっている.SOP ではプログラムの移植性を実現するために, 仮想マシン上のある言語に対するアルゴリズムを示すプログラムスケレトンが生成されるが, ユーザがデータベースからスケレトンを手動で探さなければならない. 本手法では事例ベースからのスケレトンの自動検索を目指しており, プログラムの移植性は考慮せずに仮想共有メモリ型並列マシン KSR 1 を対象としたスケレトンの作成を行っている.

その他に, トランスピュータ (分散メモリ型並列マシン) を対象としてスケレトンプログラミング環境を提案している OLGA [7] や BACS によるベンチマーク [8], スケレトン関数を用いた並列プログラミング [9] がある.

2 並列プログラミング

2.1 並列プログラミングの分類

並列プログラミングは共有メモリ並列プログラミング, 分散メモリ並列プログラミング, データ並列プログラミング, オブジェクト指向並列プログラミング, 関数型データフロープログラミングなどに分類される.

分散メモリ並列プログラミングでは, ホーアの提唱したCPSモデルが代表であり, プロセス間のセンド・レシーブを陽に指定する. 各プロセッサにおいて, 演算時間を最大に, 通信時間を最小にする必要がある. このために, 計算と通信をオーバーラップさせ, 通信オーバヘッドを極力減少させる工夫が要求される. 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する. 共有変数による各プロセッサ間の情報授受は, 分散メモリ並列プログラミングにおけるセンド・レシーブに対応する.

データ並列プログラミングでは, 大規模なデータセットに対して同一演算を行い, 細粒度の並列性を実現する. 大規模データに対する演算を逐次プログラムと同じ制御フローで実現できるので, 前二者よりもプログラミングは容易であると考えられる.

分散メモリ並列プログラミングと共有メモリ並列プログラミングはSIMD,SPMD,MIMDいずれにも適しているが, データ並列プログラミングはSIMD,SPMDに適している. どれが最も適しているかは, 通信コスト・負荷均衡・プロセッサスケジューリングなどに依存する.

並列プログラミングの要点は以下のように要約される.

- 各プロセッサにおける負荷均衡を実現するために, いかにタスクを分割するか.
- 各プロセッサ間の情報授受のためのオーバーヘッドをいかに減少させるか.

代表的な分散メモリ型マイクロプロセッサであるトランスピュータ用のParallel Cにはスレッドが用意されている. スレッドとは軽いプロセスであり, コード, ヒープ, 静的・外部データメモリを他のスレッドと共有する. 各スレッドはデータを共有でき, チャネルまたは共有メモリによって通信できる. スレッドを用いることにより, 計算と通信をオーバーラップできる.

共有メモリ型並列マシンの一種として, 仮想共有メモリ方式が近年注目されている. この方式ではメモリは実際には分散しているが, 単一の仮想アドレス空間を実現し, そこでのプログラミングを可能とするものである. マルチスレッドによる並列プログラミングでは, 単一プログラム（SPMD）を用意して, 各スレッドを各プロセッサで実行する. スレッド間の同期, 及び共有変数の相互排除をいかに実現するかが要点である.

2.2 仮想共有メモリ並列マシン KSR 1

KSR1 メモリシステム

ALLCACHE メモリ システムは, プログラマに命令とデータのための単一 2^{64} byte アドレス空間を提供する. この空間は, システム仮想アドレス（System Virtual Address : SVA）

表 1　KSR1 のキャッシュ構成

階層レベル	Working Set Size
Local Cache	32MByte
AG:0 Cache	1GByte
AG:1 Cache	34GByte

と呼ばれる.SVA 位置の内容は, 分散した形で物理的に格納されている.

ALLCACHE は, 共有連続で一貫性を持つアドレス 空間 プログラミング モデル を実現する. この実現の形式は, プログラマにとってプログラムが最も直感的な行動をとることを保証する (例えば, マルチプロセッサのプログラム実行の結果がマルチタスキングを使用した単一プロセッサ上のプログラムの実行と同等である).

ALLCACHE は物理的には個々に 32 MBytes 格納可能なローカルキャッシュと呼ばれるメモリ配列のセットから成る. システムでは, 各プロセッサに対して 1 つのローカルキャッシュが存在する. ハードウェア機構 (以下 ALLCACHE Engine) は,SVA アドレスとその内容をアドレスが参照されたとき, そのプロセッサのローカルキャッシュ内で実体化させる. 他の何かによってその空間が必要とされるまで, そのアドレスとデータはそのローカルキャッシュに留まる.

その名が示すように,ALLCACHE の振舞いはキャッシュの振舞いに似ている :データは要求で参照の場所へと移動する. けれども, 典型的なキャッシュアーキテクチャ (SOMECACHE メモリ) とは違い, ローカルキャッシュで実体化するデータの元は, メインメモリではなく, むしろ他のローカルキャッシュである. 実際に, マシンの全てのメモリは, 大きな通信可能なローカルキャッシュで構成されている. マシンのメインメモリは, ローカルキャッシュの集まりと同一である.

KSR1 システム

ALLCACHE は , KSR1 に高性能 , 拡張可能性と結合した伝統的共有メモリのプログラミングの容易さや並列処理の低コストを供給することを可能にする基礎を形成した . KSR1 は , 計算 , メモリや I/O においてバランスのとれた拡張可能性を供給する .

各々の KSR1 ALLCACHE プロセッサ , ルータや ディレクトリセル (APRD) は , 64bit の超スカラープロセッサ , 32MB の local cache メモリと ALLCACHE Engine:0 の 1 部から成る . ALLCACHE Engine:1 は ALLCACHE ルータ ディレクトリセル (ARD) から成る .

KSR1 プロセッサは , 64bit のアドレス , 64bit の整数と 64bit の浮動点データ型を用い , 40mflops のピークの割合で IEEE 標準 64bit 浮動点演算を実行することができる . プロセッサはサイクル毎に 2 つの命令を実行させる (1 つの整数または浮動点実行と 1 つのメモリ参照またはブランチ) .

各々のプロセッサは，30MB/sec の I/O が外部のソースやユーザデータに移るのを支援する．32 の APRD セルで構成される KSR1-32 は，従って総計 450MB/sec の I/O の割合に達する．510I/O チャンネルまでもつ KSR1-1088 は，総計 15300MB/sec の I/O 容量をもつ．KSR1 は，マルチプルチャネルディスク，マルチプルチャネルイーサーネット，マルチプルチャネル FDDI やシングルチャネル HiPPI を含むいくつかの型の直接 I/O アダプタを支援する．VME チャネルコントローラは，他のネットワーキングや特定要求のカスタマのためのオープンインターフェースを供給する．

マルチプルチャネルディスクアダプタは，貯蔵サブシステムにディスクアレイ (RAID) が集まるのを支援するために 5 つの異なる SCSI チャンネルを供給する．KSR OS はマルチプルアダプタによってディスクストライピングを実現する．

2.3 スレッドライブラリ

スレッドとは他のスレッドと問題解決のために協力しあうプロセス内の逐次的な制御の流れである.KSR1 では, スレッドライブラリが提供されている. 以下にスレッドライブラリの代表的なものについて説明する.

- スレッド生成
  スレッドライブラリの pthread_create を用いることによりスレッドを生成する. 生成したスレッドに整数型の引数を一つしか渡すことができないために, 複数あるいは他の型の引数を受け渡す場合には大域変数を用いて引数の受渡しを行わなくてはならない.
- スレッド結合
  スレッドライブラリの pthread_join を用いてスレッドの結合を行う. スレーブがマスタよりも先に処理を終えた場合には, 何らかの方法でマスタが pthread_join の処理を行うまで, スレーブ側は待っていなければならない. 待ちの処理を行わないとマスタは子プロセス（スレーブ）が終了したことを知らないため, いつまでも子プロセスの終了を待ち続けることになる. その場合, 子プロセスが待ち状態にあることをマスタに知らせる手段も必要となる.
- ミューテックス・ロック
  複数の pthread は同時に共有データを書き換えることができるので, データの一貫性を守るためにデータアクセスの同期を取る必要がある. 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取ることにより, 共有データに逐次にアクセスするために使われるロックである. ミューテックスロック・アンロックには pthread_mutex_lock, pthread_mutex_unlock を用いる.
- コンディション変数
  コンディション変数は, ミューテックスをロックしている pthread に代わって他の pthread がそのミューテックスをロックすることを可能にさせることにより, pthread の実行を制御する. ミューテックスをロックしている pthread の実行の継続が不可能で一時的に待避する必要がある場合, あるコンディションに出会うか

時間が経過するまでミューテックスをロックする pthread が代わる.

- バリア同期

  スレーブはチェックインの状態になると, マスタから信号を得るまで待ちの状態になる. マスタがバリアにチェックインした時点で全てのスレッドが同時にコードを実行し始める. マスタがチェックアウトに到達した時, 全てのスレーブが完了の報告をするまでマスタは待ちの状態になる. バリア同期を実現するためには pthread_barrier_checkin, pthread_barrier_checkout を用いる.
- スレッド実行プロセッサの指定

  pthread_move は ある pthread が動作する場所 (hardware location) を明記するために用いる.pthread_move の第 3 引数に次の定数を与えることにより, スレッド実行プロセッサの指定を行う.
  - TH_BOUND_NOWHERE

    スレッドは OS のスケジューラにより, プロセッサセット内のいずれかのプロセッサに割り当てられる.
  - TH_BOUND_RING0

    同じリング内のいずれかのプロセッサに割り当てられる.
  - TH_BOUND_PROSESSOR

    pthread_move の第 2 引数に明記されたプロセッサに割り当てられる. もし, 現在そのプロセッサ上で実行されていないなら指定のプロセッサへの移動が起こる.
- プロセッサ **ID** の取得

  pthread_getproc を用いてプロセッサ ID を取得する. 第一引数で示すスレッドの実行プロセッサ ID を取得し, 第 2 引数に格納する.

2.4 並列プログラミング支援システム

図 6 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 ・履歴 ・並列効果 から成る事例が蓄積されている. システムは複数のインデックスを用いて事例ベースの中から最も類似した事例を検索する. ユーザはシステムと会話的にインデックスを作成する. 作成されたインデックスはシステムによって事例ベース内のインデックスと比較され類似事例が検索される. 完全にマッチする類似事例が検索できないときは, 段階的に検索条件を緩めて, 類似した事例を探す. 最悪の場合でも条件の緩和によって何らかの類似事例をユーザに提供する.

最善の場合, 非常に類似した事例が検索され, 与えられた問題への変換が自動的になされる. 一方, 最悪の場合でも, 何らかの関連性のある事例が検索されるので, ユーザによる並列プログラミングの手助けになる. 中間の場合には, スケレトンへの情報の追加が自動あるいは手動でなされる. スケレトンとは並列プログラムの骨格を示す最も重要な部分である. スケレトンにはスレッドの使用法, 共有変数の同期, タスク分割などと共に, 変数初期化, 計算, 結果の回収などの一連のプログラミングプロセスが記述されている.

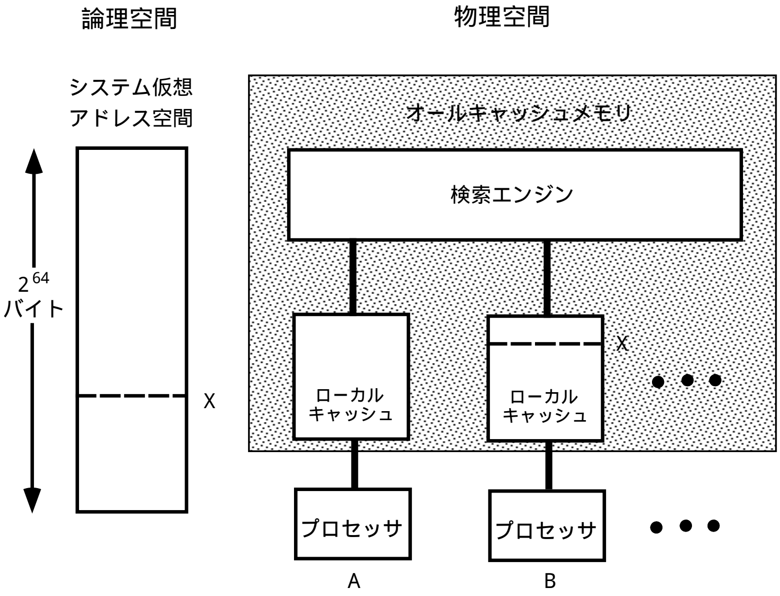


図 1　KSR1 のメモリ構成

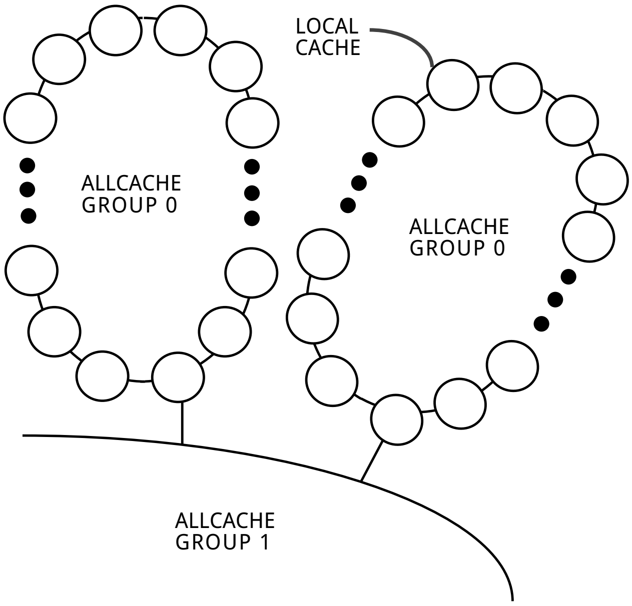


図 2　KSR1 のシステム構成

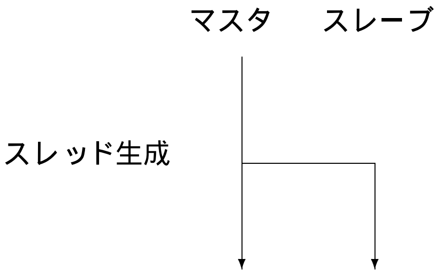


図 3　スレッド生成

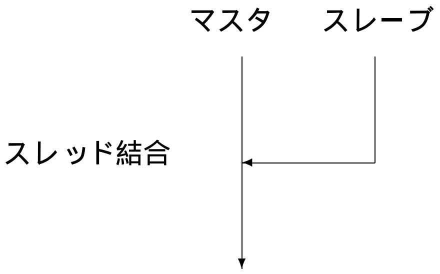


図 4　スレッド結合

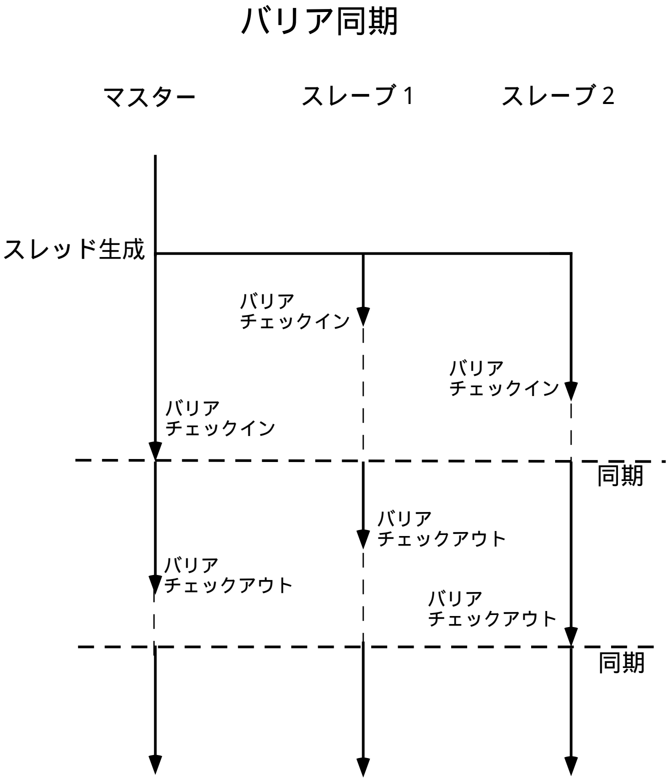

バリア同期
マスター
スレーブ 1
スレーブ 2
スレッド生成
バリア
チェックイン
バリア
チェックイン
バリア
チェックイン
同期
バリア
チェックアウト
バリア
チェックアウト
バリア
チェックアウト
同期

図 5　バリア同期

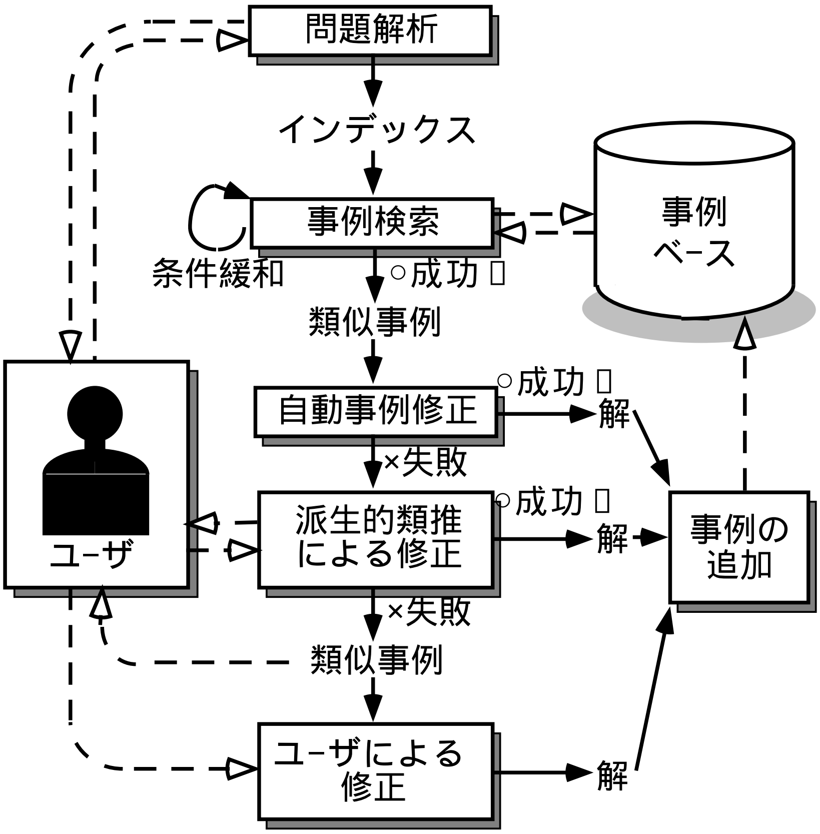


図 6　並列プログラミング支援システムの構成

3　並列アルゴリズムの分類

基本的なデータ操作, 数値計算, 記号処理, ビジネスデータ処理など広範な応用分野において, 並列アルゴリズムを開発するための努力がなされてきた. 最近では, １９９０年代のグランドチャレンジとして, 気象予測, プラズマモデリング, 財政モデリング, 乱流計算など非常に大規模な問題が対象になっている.

3.1　標準的な分類

一般的に, 並列アルゴリズムをプロセッサファーム (Processor Farms), 分割統治法 (Divide and Conquer), プロセスネットワーク (Process Networks), 繰り返し変換 (Iterative Transformation) の４つのパラダイムに分類することが考えられる [10].

3.1.1　プロセッサファーム

問題は複数の独立な計算に分割され, それらの結果が統合される. 全体の制御は一ヶ所で行われ, 独立した計算の数はプログラマあるいは使用可能なスレーブ数によって決まる. プロセッサファームとして並列処理できるのは問題そのものに並列性があるものであるため, 高い並列効果が期待できる.

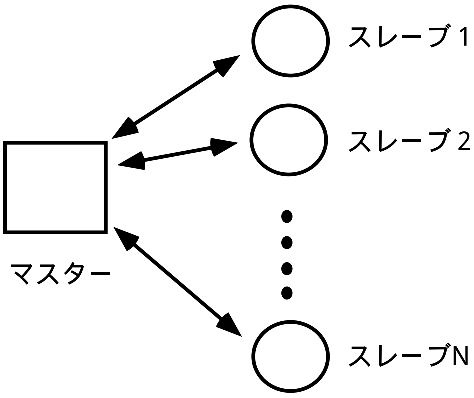


図 7　プロセッサファーム

3.1.2　分割統治法

問題は下位の部分問題に分割され, それら自身がさらに分割されて再帰的に解かれる. 最終結果は部分問題の解を再帰的に結合することにより得られる. タスクの分割数, 計算の精度があらかじめ決まっている場合にはプロセッサファームによる並列処理を行えば良いが, 計算結果がある精度に達するまで処理が行われなければならない場合には分割統治法を用いるのが良い.

3.1.3 プロセスネットワーク

計算を複数のステージに分け, データがステージを流れる. 入力データの有無によって, ステージは並行に動作する. データフロー, あるいはパイプライン処理的なイメージに近い.

3.1.4 繰り返し変換

オブジェクトは複数の繰り返しステップにより変換され, 局所・広域・結合オペレータが適用される. 局所・広域オペレータはデータの種類により分類される. 結合オブジェクトでは, オブジェクトのグループが他のオブジェクトの集合を形成する.

3.2 BACS の分類

BACS(Basel Algorithm Classification Scheme) 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 は主にアルゴリズムのトポロジー, プロセス構造, 相互作用の方式, データ分散, 及び実行構造の観点から並列アルゴリズムを分類する. アルゴリズムトポロジーは規則性と均質性から分類される. 規則的・均質的なトポロジーとして, 木, トーラス, メッシュ, ハイパーキューブが, 規則的・不均質なトポロジーとして, ワーカーがある. 不規則なトポロジーとは各プロセスが異なった構成ルールをもつものである. プロセス構造には静的と動的がある. 相互作用は 2 ノード間か (直接) / 3 ノード以上間 (広域), 及び陽になされるか/ 陰になされるかの両点から分類される. 例えば, ミューテックスによる相互作用は広域かつ陰である. 実行構造として, 微細 (microscopic) と巨視 (macroscopic) がある. 微細構造は計算, 相互作用, 及びデーモンコールの基本要素によるプロセスの合成順序を示し, 並列アルゴリズムのスケレトンの基礎となる. 一方, 巨視的構造はアルゴリズムの全てのイベントの列である. 殆どの並列アルゴリズムの実行構造は規則的であり, それらはイベントの同一の繰り返し系列から成っている.

~ **BACS** による実行構造の表現 ~

- イベント:
  イベントは計算部分, インタラクション, デーモンのいずれかで C, I, D で表す.
- トータルイベント:
  全てのプロセスが実行するイベント. 右上に t を書く. (e.g. C^t)
- パーシャルイベント:
  一部のプロセスが実行するイベント. 右上に p を書く. (e.g. C^p)
- インタラクション:
  インタラクションの名前かその略語を添え字に付ける.(e.g. I_{bcst})
- デーモンコール: プロセスの生成 (D_c), プロセスの消滅 (D_d).
- **FOR LOOP:**
  実行前に繰り返し回数の決まっているループ.($FIX_{topology}$)
- **WHILE LOOP:**
  実行時に条件によって繰り返し回数の決まるループ.($COND_{topology}$)

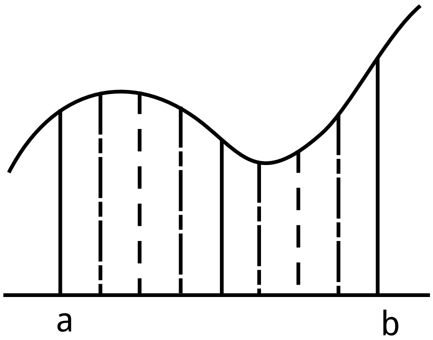


図 8　分割統治法

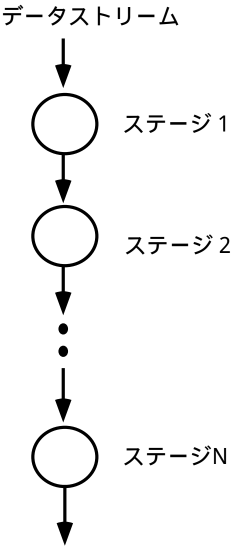
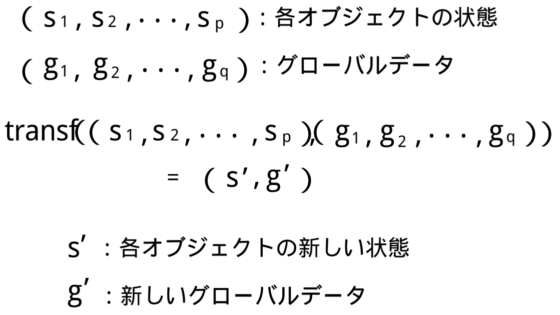


図 9　プロセスネットワーク



$(s_1, s_2, \cdots, s_p)$：各オブジェクトの状態

$(g_1, g_2, \cdots, g_q)$：グローバルデータ

$$\mathrm{transf}((s_1, s_2, \cdots, s_p), (g_1, g_2, \cdots, g_q))$$
$$= (s', g')$$

s'：各オブジェクトの新しい状態

g'：新しいグローバルデータ

図 10　繰り返し変換

4 事例の表現と例

事例はインデックス, スケレトン, プログラム本体, 履歴, 及び並列効果から成る. 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, 変数, 及び終了条件はマシン独立なインデックスであり, 一方, 同期と並列化手法は Posix 標準に準拠している.

並列プログラムの骨格であるスケレトンはスレッドの扱い方, 共有変数の同期, タスク分割などと共に, 変数初期化, 計算部, 計算結果の回収などの一連のプログラミングプロセスが記述されている. スケレトンはプログラムの骨格を示し, もし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.

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. 並列効果はその事例においてどの程度の並列効果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.

この章ではまず事例の一般表現を示す. 次に事例の例として, レイトレーシング, 2分木ソート, 積分計算, マンデルブロ, 高次方程式のインデックスとスケレトンを示す. 並列効果については次章で述べ, プログラム本体は付録とする.

4.1 一般的表現

(1) インデックス

応用	データ操作, 数値計算, 記号処理, ビジネスデータ処理
仕様	問題の定義
アルゴリズム	分割統治法, プロセスネットワーク, プロセッサファーム, 繰り返し変換
BACS による分類	
変数	共有 / 局所, リダクション, コモン
終了条件	一定, 条件式
同期	ミューテックス, バリア, コンディション変数
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

(2) スケルトン

- 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

(4) 履歴

- 参照頻度, 成功 / 失敗
- 事例ベースの更新

(5) 並列効果

4.2 レイトレーシング

レイトレーシング法はコンピュータグラフィックスのレンダリングアルゴリズムである. 各ピクセル毎の計算は独立であるため高い並列性を持つ. 本事例では画面を横方向にプロセッサ数だけ等分割して並列処理を行う. 各プロセッサでは, スレッド生成時に与えられたスレッド番号により計算範囲が決められ, 各プロセッサで同一回数の計算を行う. 各プロセッサでは, 物体の配置の偏りによって計算量の差が生じるために演算時間にも時間差が生じる. マスタ側とスレーブ側のどちらが先に処理を終えるかをあらかじめ知ることは困難なので, どちらが先に処理を終えても良いようにバリアで同期を取っておく.

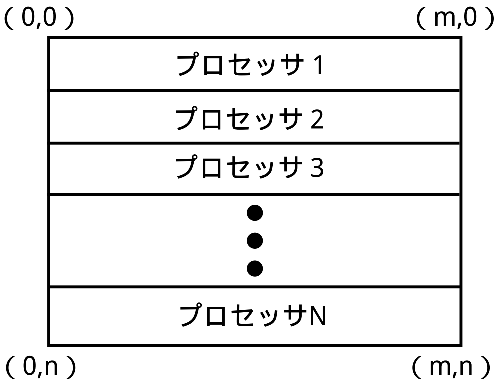


図 11 画面の分割方向

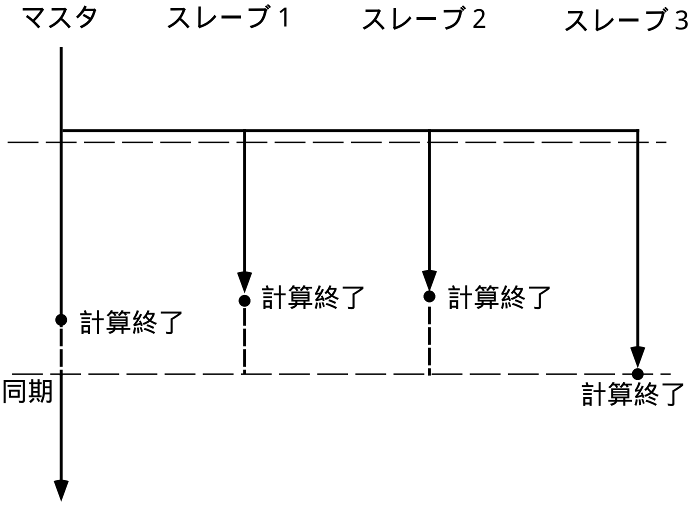


図 12 レイトレーシングの実行構造

(1) インデックス

レイトレーシング

応用	数値計算
仕様	$At^2 + Bt + C = 0$ $A = E_x^2 + E_y^2 + E_z^2 = 1$ $B = 2E_x(v_x - x_0) + 2E_y(v_y - y_0) + 2E_z(v_z - z_0)$ $C = (v_x - x_0)^2 + (v_y - y_0)^2 + (v_z - z_0)^2 - r^2$
アルゴリズム	プロセッサファーム
BACS による分類	$I^t_{barrier}, Fix_w(C^t)$
変数	共有 / 局所
終了条件	一定
同期	バリア
並列化手法	スレッドライブラリ

(2) スケレトン

```
pthread_barrier_t A;
画像データ用変数の宣言;
main()
{
        変数の初期化;
        pthread_barrier_init(&A, プロセッサ数);
        for(プロセッサ数)
           pthread_create(&pt,ray,(void *) スレッド番号);
        ray((void *)0);
        画像データのファイルへの出力;
}
void *ray(スレッド番号)
{
        pthread_barrier_checkin(&A, スレッド番号);
        スレッド番号によるタスクの割り付け;
        for(スレッド番号による計算範囲){
           レイトレーシングの計算部;
           画像データ用変数へのデータの格納;
        }
        pthread_barrier_checkout(&A, スレッド番号);
}
```

4.3 2分木ソート

現在保持している列の中央値 M を逐次アルゴリズムを用いて見いだす。列の左から操作を行う場合 M よりも小さいデータは左に M よりも大きなデータは右に流す。また、部分列について同様の操作を繰り返し並列に行う。プロセッサ数に等しい個数の部分列に分割した後、保持している部分列を逐次アルゴリズムを用いてソートする。

各部分列のソートは独立な作業であるため, 各プロセッサ間の同期・通信は必要ではない。

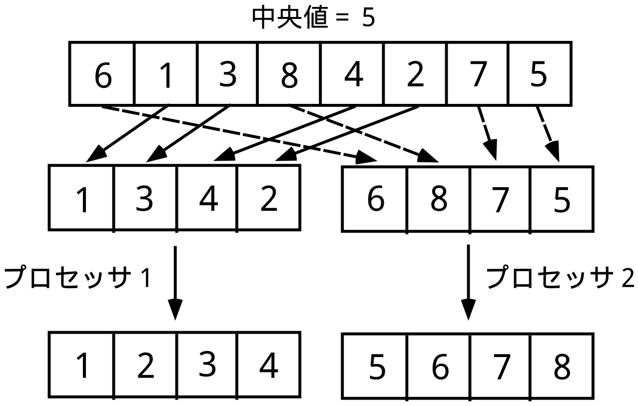


図 13 2分木ソート アルゴリズム

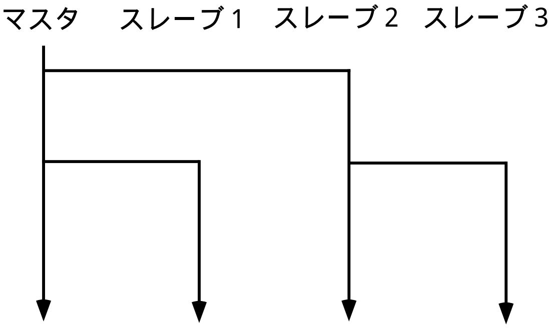


図 14 2分木ソートの実行構造

(1) インデックス

2分木ソート	
応用	データ操作
仕様	要素 (< 中央値) ならば 左へ 要素 (> 中央値) ならば 右へ
アルゴリズム	分割統治法
BACS による分類	C^t
変数	共有 / 局所
終了条件	一定
同期	なし
並列化手法	スレッドライブラリ

(2) スケルトン

```
スレッド間通信用変数の宣言;
データ用変数の宣言;
void main(void)
{
          変数の初期化 ;
          sort(1);
          全ての処理の終了待ち;
}
void *sort(int no)
{
           if(no < 終了条件){
              中央値の探索;
              pn=no*2;
              pthread_create(&pt[pn], sort,(void *)pn);
              sort(no*2+1);
           }
           else break;
}
```

4.4 積分計算 (1)

ある精度を得るまで分割・計算を繰り返す求積法 (Adaptive Quadrature) を用いた積分計算を行う. 図 15 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.

終了条件を早く満たすものとそうでないものとが存在するため, 各プロセッサでの負荷は均一ではない.

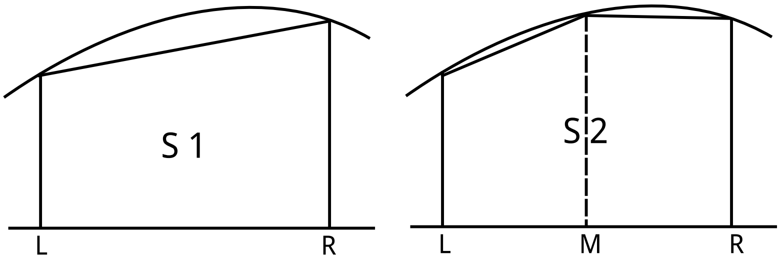


図 15 積分計算の誤差

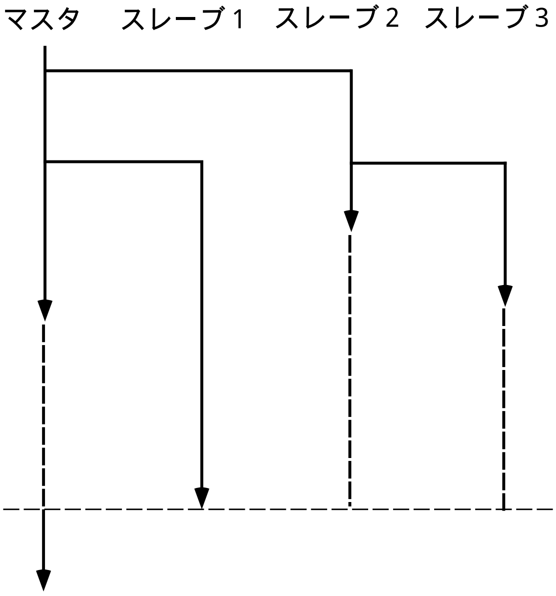


図 16 積分計算 (1) の実行構造

(1) インデックス

積分計算 (1)

応用	数値計算
仕様	$y = \int_a^b f(x)dx$
アルゴリズム	分割統治法
BACS による分類	$I^t_{barrier}, Cond_w(C^t)$
変数	共有 / 局所
終了条件	$\epsilon < 0.0001$
同期	バリア
並列化手法	スレッドライブラリ

(2) スケルトン

```
pthread_barrier_t A;
double ans[CELL];
スレッド間通信用の変数の宣言;
main()
{
        double total;
        変数の初期化;
        pthread_barrier_init(&A, プロセッサ数);
        submain(左端, 右端, スレッド番号);
        各プロセッサの結果を合計;
}
void submain(double L, double R, スレッド番号);
{
        pthread_barrier_checkin(&A, スレッド番号);
        estimate(L,R, スレッド番号);
        pthread_barrier_checkout(&A, スレッド番号);
}
void thread_start(スレッド番号);
{
        大域変数によりスレッドの引数 (L,R) を受け取る;
        submain(L,R, スレッド番号);
        各プロセッサの計算結果を合計する;
}
```

```
estimate(double L,double R, スレッド番号)
{
    M=(L+R)/2;
    積分計算;
    if(誤差 $>\epsilon$){
        if(スレッド実行数 < プロセッサ数){
            大域変数に引数 L,M を書き出す;
            pthread_create(&pt,thread_start,(void *) 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 各プロセッサ用の変数に結果を足し合わせる;
}
```

4.5 積分計算 (2)

基礎となる求積法は 積分計算 (1) と同じである. 適応求積法は各プロセッサの負荷の分散に偏りが生じる可能性が高いが, 終了条件に合致するまで計算を繰り返すというアルゴリズムであるため, 計算前に静的に負荷の均衡を図ることは難しい. そこで動的に少しでも負荷の均衡を図ることを考える.

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. しかし, その時点でどのプロセッサに一番負荷がかかっているのかがわからないことと未処理部分のほんの一部分しかスレッドとして移動させることができないために完全な負荷の均衡を図ることはできない.

スケルトン中のスレッド実行数とはその時点で実際に実行されているスレッド数であり, 全プロセッサ数から空きプロセッサ数を引いたものに等しい. またスレッド生成数とはその時点までに生成されたスレッドの累計である.

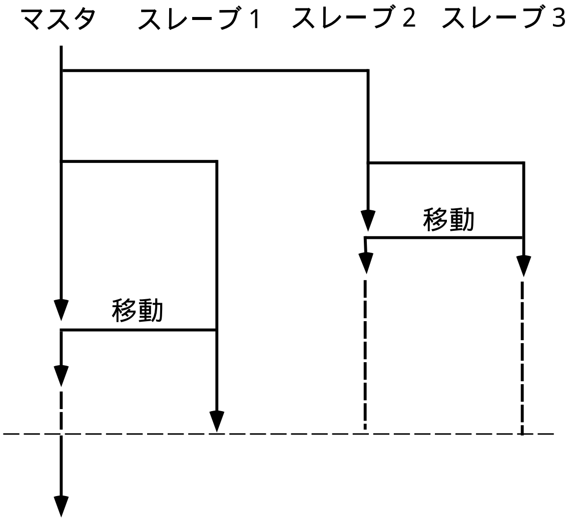


図 17 積分計算 (2) の実行構造

(1) インデックス

積分計算 (2)

応用	数値計算
仕様	$y = \int_a^b f(x)dx$
アルゴリズム	分割統治法
BACS による分類	$Cond_w(C^t)$
変数	共有 / 局所
終了条件	$\epsilon < 0.0001$
同期	なし
並列化手法	スレッドライブラリ

(2) スケレトン

```
pthread_barrier_t A;
double ans[CELL];
スレッド間通信用の変数の宣言;
main()
{
        double total;
        変数の初期化;
        submain(左端, 右端, スレッド番号);
        while(1){
            全プロセッサの処理が終われば終了;
        }
        各プロセッサの結果を合計;
}
void submain(double L, double R, スレッド番号);
{
        estimate(L,R, スレッド番号);
        空きプロセッサ有のフラグを立てる;
        スレッド実行数 −−;
        pthread_getproc(pt[no],&name[no]);
}
void thread_start(スレッド番号);
{
        大域変数によりスレッドの引数 (L,R) を受け取る;
```

```
    submain(L,R, スレッド番号);
    各プロセッサの計算結果を合計する;
}
estimate(double L,double R, スレッド番号)
{
  M=(L+R)/2;
  積分計算;
  if(誤差 > $\epsilon$){
    if(スレッド実行数 < プロセッサ数){
      スレッド実行数++;
      if(スレッド生成数 < プロセッサ数){
        大域変数に引数 L,M を書き出す;
        pthread_create(&pt,thread_start,(void *) スレッド番号);
      }
      else if(スレッド生成数 > プロセッサ数*3/2){
        i=空きプロセッサのスレッド番号;
        pthread_create(&pt,thread_start,(void *) スレッド番号);
        pthread_move(pt[i],name[i],TH_BOUND_PROCESSOR);
      }
      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 各プロセッサ用の変数に結果を足し合わせる;
}
```

4.6 マンデルブロ

マンデルブロ集合は, 複素数 z, c に対して次の手続きを終了しない手続きとして定義される.

$$z := 0;$$

$$while|z| < 2.0 \qquad do \quad z := z^2 + c$$

範囲内の座標点を無限に細かく計算することはできないので有限個数の座標点において計算を行う. また, 定められた有限回数の間にループを抜けなければマンデルブロ集合と見なす. マンデルブロ集合の計算は各座標において独立した計算であるので高い並列性を有する. また, 本事例では x 座標方向に 1 行ずつ各プロセッサに処理を行わせるため, 静的な負荷分散によって各プロセッサの負荷の均衡が図れる.

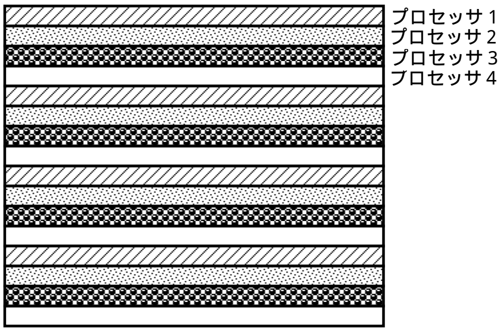


図 18 マンデルブロの分割方法

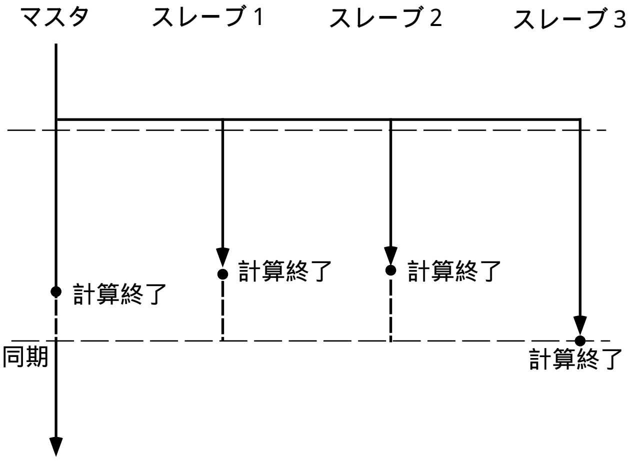


図 19 マンデルブロの実行構造

(1) インデックス

マンデルブロ

応用	数値計算
仕様	$Z_{i+1} = Z_i^2 + C\ i = 0, 1, 2, ...$
アルゴリズム	プロセッサファーム
BACS による分類	$I^t_{barrier}, Cond_w(C^t)$
変数	共有／局所
終了条件	$Z > 2.0$
同期	バリア
並列化手法	スレッドライブラリ

(2) スケレトン

```
pthread_barrier_t A;
計算結果用変数の宣言;
main()
{
        変数の初期化;
        pthread_barrier_init(&A, プロセッサ数);
        for(プロセッサ数)
           pthread_create(&pt,work,(void *) スレッド番号);
        work((void *)0);
        計算結果のファイルへの出力;
}
void *work(スレッド番号)
{
        pthread_barrier_checkin(&A, スレッド番号);
        スレッド番号によるタスクの割り付け;
        for(スレッド番号による計算範囲){
           while(ループの条件 && (Z < 2.0)){
              マンデルブロの計算部;
           }
        }
        計算結果用変数にデータを格納;
        pthread_barrier_checkout(&A, スレッド番号);
}
```

4.7 高次方程式の解法 (DKA 法)

DKA(Duran,Kerner,Aberth) 法を用いて n 次方程式

$$a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_nx + a_{n+1} = 0$$

の n 個の解

$$x_{n-1}, x_{n-2}, x_{n-3}, \cdots; x_2, x_1, x_0$$

を求める. まず, 各 x_i の初期値

$$x_{n-1}^{(0)}, x_{n-2}^{(0)}, x_{n-3}^{(0)}, \cdots; x_2^{(0)}, x_1^{(0)}, x_0^{(0)}$$

を求める. 次に最大ループ数に達するか x_{n-1} ~ x_0 の全てが前回の計算結果と比較して変化を生じなくなるまで計算を繰り返す. 条件に合えば計算を終了する.

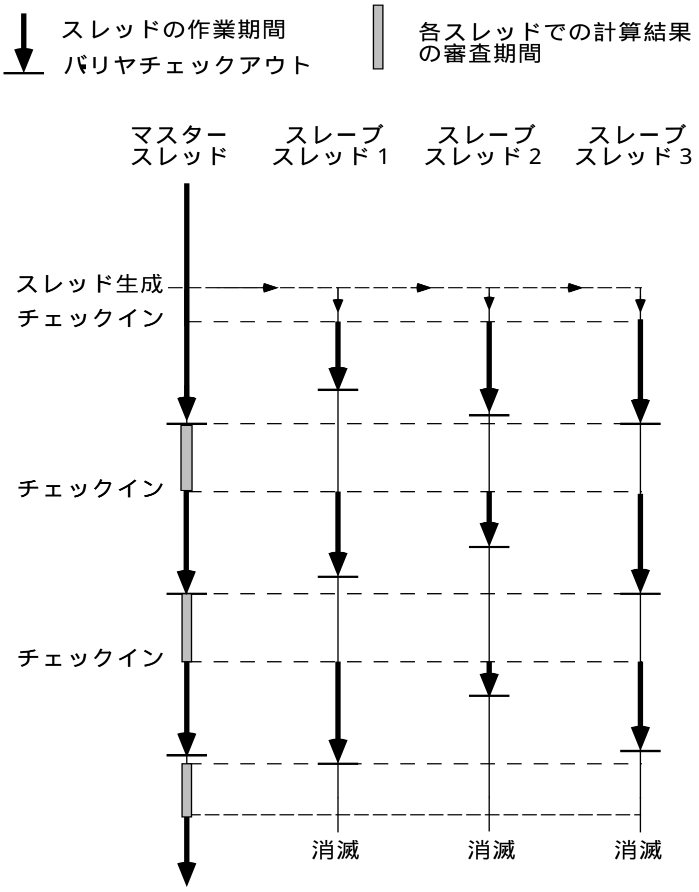


図 20 DKA 法の実行構造

(1) インデックス

DKA 法	
応用	数値計算
仕様	$Z_i^{k+i} = Z_i^k - \frac{f(Z_i^k)}{\prod_{j=1(j\neq i)}^{n}(Z_i^k - Z_j^k)}$
アルゴリズム	プロセッサファーム
BACS による分類	$I_{barrier}^t, C^t, Cond_w(I_{barrier}^t, C^t)$
変数	共有 / 局所
終了条件	$x_i^{(j)} = x_i^{(j-1)} \quad (i = n-1, n-2, ..., 1, 0)$
同期	バリア, ミューテックス
並列化手法	スレッドライブラリ

(2) スケルトン

```
pthread_barrier_t barrierA;
pthread_mutex_t LOCK;
int check;
main()
{
    n 次方程式の n 個の係数の入力;
    pthread_barrier_init( &barrierA, プロセッサ数 );
    for (プロセッサ数 )
      pthread_create( &pt , work1, (void *) スレッド番号 );
    work1(0);
    pthread_barrier_destroy( &barrierA);
    pthread_mutex_init( &LOCK, pthread_barrier_default );
    check1 = 1;
    while ( 最大ループ回数以内 || check == 1 ){
        check = 0;
        pthread_barrier_init( &barrierA, プロセッサ数 );
        for ( プロセッサ数 )
          pthread_create( &pt , work2, (void *) スレッド番号 );
        work2(0);
        pthread_barrier_destroy( &barrierA );
    }
```

```
        結果の出力；
}
void *work1(int no)
{
        pthread_barrier_checkin( &barrierA, スレッド番号 );
        for ( スレッド番号による計算範囲 ){
            スレッド番号により割り振られた全a(i)の計算；
            初期値の計算；
        }
        pthread_barrier_checkout( &barrierA, スレッド番号 );
}
void *work2(int no)
{
        pthread_barrier_checkin( &barrierA, スレッド番号 );
        for ( スレッド番号による計算範囲 ){
            今回計算したaと,前回計算したaとの比較；
            if ( !比較の結果が条件){
                pthread_mutex_lock( &LOCK );
                check = 1;
                pthread_mutex_unlock( &LOCK );
            }
        }
        pthread_barrier_checkout( &barrierA, スレッド番号 );
}
```

5 並列効果

過去においてその事例がどの程度の並列効果を得ることができたのかという情報を事例の一部として残しておく. 新しく対象となる問題を並列処理した場合に逐次プログラミングに比べてどの程度の速度向上が得られるのか, あるいはどの程度のプロセッサ数で速度向上が限界となるのかを並列プログラミングを行う前にある程度 知ることができればユーザにとって意義深いが, 実際に前もって明確にどの程度の並列効果があるのかを予測することは困難である. しかし, 過去に類似した並列処理でどの程度の並列効果が得られたのかが解っていれば, ある程度の予測を行うことも可能であるかも知れない. 新たな並列プログラミングの参考にするために過去の並列効果を事例の一部とする.

検索された事例の類似度が高い場合, 新たな並列プログラミングにおいても過去と同程度の並列効果を得ることが期待できる. 類似度が低い場合にはどの程度の並列効果を得ることができるかは解らないが, 事例数が増えれば類似度の低いものが検索される可能性が低くなるためそのような問題は事例数を増やすことによって解決されると考えられる.

一般的に並列処理の速度向上を妨げる原因は, 同期 ･通信など並列プログラミングの方法によるものと, データの依存関係などの問題独自の要因によるものとに分類することができる. 対象とする問題が並列性を持つかどうかを自動的に知るためにはデータの依存関係を解析するなどの必要があり, 本研究で検討する手法の範囲を越える. 一方, スレッドの同期 ･通信方法など並列プログラミングの不慣れが原因で速度の向上を得られない場合には, 過去の類似事例を参照することにより速度向上がより良く, また正しい解の求まる並列プログラムを作成することが期待できる. ある程度の並列効果を達成するには同期の取り方, 通信方法, 各プロセッサに対するタスクの分割, スレッドの生成個数, 共有変数の使用法など逐次プログラミングの際には不必要な知識が必要となるため並列プログラミングは面倒である. また, ある程度並列処理の知識があっても実際にプログラミングを行うとデバッギングに多くの時間を費やさなくてはいけないのも事実であり, 本研究では並列効果の高い並列プログラミングを行うために過去の事例を利用することを考える. 過去の事例から解決すべき問題の並列効果がどの程度のものかを推測するためには, 類似した過去の事例において どの程度の計算量でどの程度の速度向上を得られたのかという知識が必要となる. 実行時間から計算量を速度向上グラフから並列効果を知るために両者を事例の一部とする.

実行時間の例として表 2 から表 5 にレイトレーシング ･積分計算 ･マンデルブロの実行時間をを示す. また, 並列効果の例として図 21 から図 24 にレイトレーシング ･積分計算 ･マンデルブロの並列効果を示す.

レイトレーシングは 640*480 の画面上に平面と１６個の球を描いた場合の結果である. また, 積分計算 (1)(2) は共に４次関数を $(-100.0,100.0)$ までの範囲で, マンデルブロは実軸 $(-2.0,0.5)$, 虚軸 $(-1.25,1.25)$ の範囲を 200*150,400*300,800*600 の有限個数の点において計算を行ったものである.

レイトレーシイングはプロセッサ数が 1 ~ 8 では高い並列効果を得ているが, プロセッサ数 16 では少し並列効果が落ちている. これは物体の配置の偏りにより各プロセッサに負荷の不均衡が生じているためである.

積分計算 (1) は終了条件を満たすまで繰り返す計算であるために, プロセッサによってかなりの負荷の不均衡が生じる. (N-1) 台のプロセッサが短時間で処理を終えても, 残りの 1 台が処理に長時間を要せば,N 台での並列効果は低いものとなる.

積分計算 (2) はスレッドの移動を行うことで積分計算 (1) の実行時間を短縮したものである. 短時間で処理を終えたプロセッサに他のプロセッサの処理の一部を移動するこで多少の高速化を行っている.

マンデルブロは問題そのものに並列性があり, 静的な負荷分散により各プロセッサの負荷を均衡させせることができるので高い並列効果を得る.

表 2　レイトレーシングの実行時間 (s)

セル数	1	2	4	8	16
経過時間	73.18	40.36	20.33	10.54	7.06

表 3　積分計算 (1) の実行時間 (s)

セル数	1	2	4	8	16
経過時間	47.88	33.47	26.27	19.15	15.37

表 4　積分計算 (2) の実行時間 (s)

セル数	1	2	4	8	16
経過時間	47.90	29.98	18.42	11.98	7.04

表 5　マンデルブロの実行時間 (s)

セル数	1	2	4	8	16
経過時間 1(200*150)	5.66	2.88	1.51	0.87	0.77
経過時間 2(400*300)	22.68	12.32	5.77	3.03	2.30
経過時間 3(800*600)	90.00	44.71	23.30	11.43	6.08

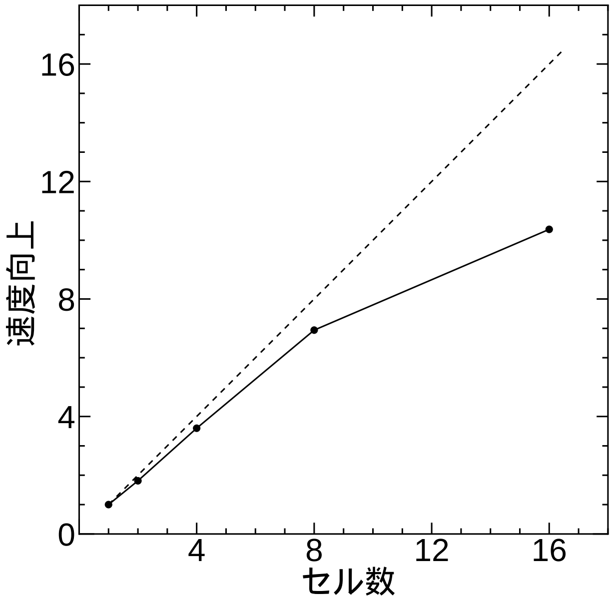


図 21　レイトレーシングの並列効果

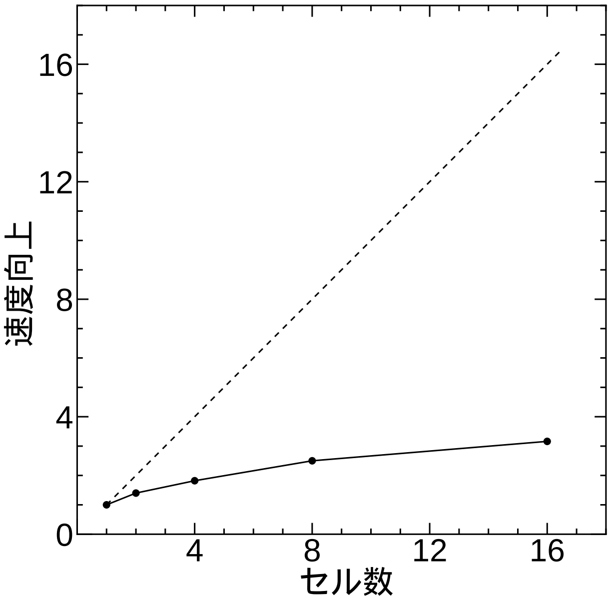


図 22　積分計算 (1) の並列効果

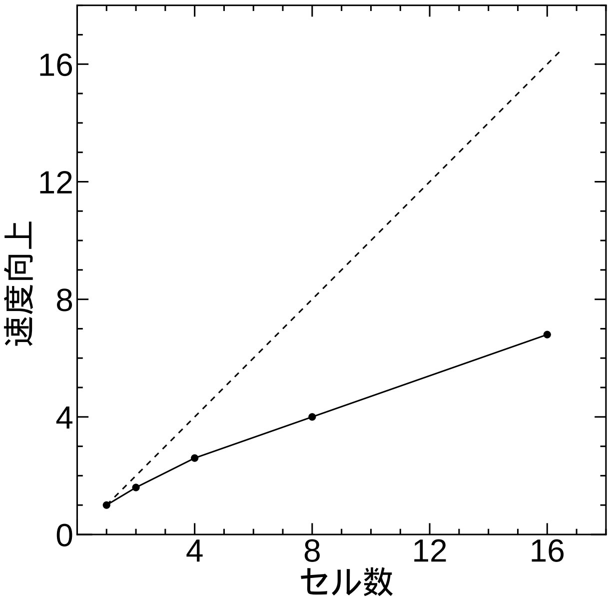


図 23　積分計算 (2) の並列効果

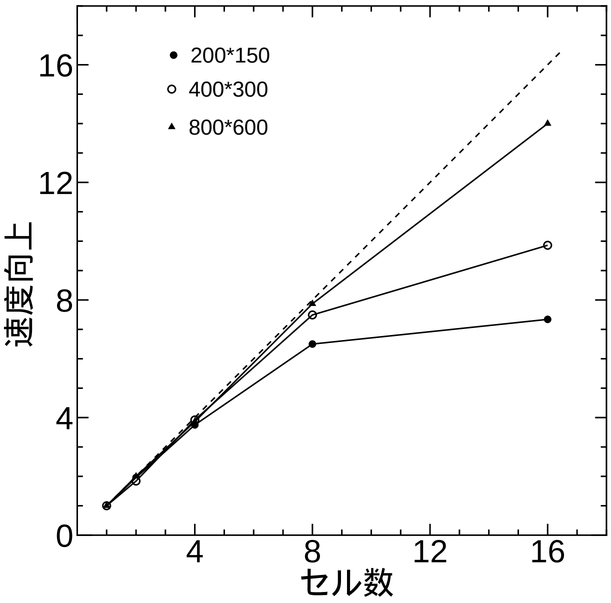


図 24　マンデルブロの並列効果

6 事例の検索と修正

前章までの形式で並列プログラミングに関する知識が事例として蓄積されているとしても, 知識を再利用するには事例の中から最適な事例を検索する必要がある. この章では, インデックスを使用した事例の検索方法について検討する.

6.1 類似事例の検索

インデックスは抽象的なものから具体的なものへと並べられている. 抽象化のレベルの観点から見ると, 変数と終了条件はプログラムレベルに対応し, 同期と並列化手法は実現レベルに対応する. 事例検索では全てのインデックスが合致するもの, さもなければ検索条件を緩めて類似度の高い事例を検索する.

6.2 類似度の評価方法

インデックスには,{ 応用/仕様/アルゴリズム/BACS による分類/変数/終了条件/同期/並列化手法 } の要素がある. 事例の類似度を評価するため, インデックスを { 仕様/BACS/選択式要素 } の 3 つに分割し, それぞれの類似度を求める. 各事例の類似度は, 仕様類似度,BACS 類似度, 選択式要素の類似度を総合して評価する.

仕様類似度

仕様に関する類似度である. 仕様は問題の定義であり, 数値計算ならば数式が, その他の場合には手続きが示されている. 仕様類似度が高ければ, システムによる自動的な修正が期待できる.

BACS 類似度

BACS の分類方法では, 並列プログラムの実行構造を $I, D, C, Fix, Cond$ などの文字列で表す. BACS 類似度は, 各並列プログラムに対する実行構造表記の類似度を表すもであり,BACS 類似度が高ければ, 派生的類推による修正が期待できる.

選択式要素の類似度

選択式要素の類似度は,{ 仕様/BACS の分類 } 以外のインデックスに関する類似度である. ただし, 並列化手法には { 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 } の選択肢があるが,{ パラレルリージョン/パラレルセクション } は FORTRAN に関する並列化手法であり,C プログラミングではスレッドライブラリによる並列化手法しか利用できないため, 類似度の評価には含めない. 選択式の要素の選択肢は有限であるので, 全ての選択肢に合致する多数の事例が存在すれば, 選択式要素の類似度が 100%の事例を常に提示することができる. しかし, 事例を選択肢数だけ用意するのは現状では難しいため, 選択式要素の類似度を用いて類似した事例を探し, 派生的類推による修正/ユーザによる修正に用いる. 選択式要素の類似度は, 複数の選択肢の中からいくつかのものを選択する要素に関する類似度なので, 容易に類似度を示すことができる. 各要素に { 応用/アルゴリズム/終了条件/同期/変数 } の順に優先順位を付け, 優先順位を考慮して類似事例を

検索する.

6.3 類似事例の修正

事例の修正方法には, システムによる自動的な修正, 派生的類推による修正, ユーザによる修正の 3 つの方法が考えられる. 最善の場合には, 自動的な修正により過去の事例から自動的に新たな並列プログラムが作成される. 最悪の場合には, ユーザによる全面的な事例の修正が必要となるが, ユーザには過去の並列プログラムに関する類似した事例が提示され, 過去の知識を参考にすることができる.

自動的な事例の修正

新しい問題と検索された事例の仕様が式のレベルで似ていれば, 自動的な修正の可能性がある. 例えば, マンデルブロの変形で, 右辺が Z_i の関係である式が与えられれば, システムは右辺の計算部分だけを修正すれば良い.

派生的類推による修正

プロセッサファームはデータ並列性を含んでいる. データ並列計算の代表的な例は粒子シミュレーション, 流体フローモデリング, 情報検索, コンピュータグラフィックスなどである. これらの問題は同一サイズのタスクに分割され, 結果を格納するための共有 ･リダクション変数が必要である. 従って, マンデルブロのスケレトンがこれらの問題に使用できる可能性がある. スケレトン内の変数初期化, タスク割り当て, 計算, 共有変数への書き込みの部分は各プログラム毎に異なる部分である. 従って, スケレトンをユーザに提示し, これらの部分への書き込みをシステムとユーザが会話的に行うのが現実的な修正法であると考えられる.

ユーザによる修正

インデックスの合致数が極端に少なく事例ベース内に類似した事例が見つからない場合には, システムとユーザとの会話的な修正は不可能である. 従ってスケレトンをもとにユーザが自力で事例の修正を行うことになる.

7 おわりに

過去の代表的な並列プログラムの構造をスケレトンとして事例ベースに保存し, 新しい問題に対する類似したスケレトンを検索・修正して並列プログラムを生成する手法について検討した. 並列プログラムにおけるタスク分割, 共有変数の同期, 並列化手法はプログラミングが最も困難であるが, 同じ種類のアルゴリズムに対しては, 同一の構造を適用できる.

これまでに本システムに用いる一部の事例を作成した. プロセッサファームとして処理できる問題については, 計算前の静的な負荷分散によってある程度高い並列効果を得ることができる. また, 終了条件を伴う分割統治アルゴリズムは, 終了条件によってプロセッサ間で負荷の不均衡が起こるが, 計算前にそれを知ることはできない. そこで, そのような場合にはスレッドの移動による動的な負荷分散を行うことで多少の相対速度向上を得ることができる. また, プロセスネットワークアルゴリズムを仮想共有メモリ型並列マシン上で実現するためには, 多くのロック・アンロックによるオーバーヘッドを伴うことになり, 現状では高い並列効果は望めない. このような知識は検索・再利用可能な形式で蓄積されていなければ, 再び同様なプログラミングを行う際に全てを初めから組み上げなければならないことになる. それを避けるためには本論文で検討した事例の蓄積が役立つものと考えられる.

今後の課題としては, インデックスに対応する個数の事例を数多く作成し, 可能な限り速度向上を果たした上で事例に加えることが挙げられる. また, 事例検索部やユーザインターフェースの作成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本研究室博士前期課程の松田浩一氏, 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] Kendall Square Research: “KSR Parallel Programming”, 1992.

[2] 山崎, 古川: “事例ベース推論による並列プログラミング支援システム”, 情報処理学会、94-OS-67-14, 1994.

[3] 小林重信: “経験に基づく推論と学習”、電子情報通信学会 pp. 25-31, 1994.

[4] 吉村晋、黄錦法、白鳥則郎: “事例ベースを推論を適用した通信ソフトウェア開発支援環境”、人工知能学会誌 vol.8 No.6 pp.713-716 , 1993.

[5] 服部雅一、田中利一、末田直道: “事例ベース推論による機械設計”、人工知能学会誌 vol.9 No.1 pp.82-90 , 1994.

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

[7] Stephan Waser, Helmar Burkhart: “OLGA - A Modelling Tool for Algorithmic Skeletons”, Transputer Applications and Systems ’93 IOS Press pp.395-409, 1993.

[8] Stephan Waser: “Benchmarking Parallel Computers Using Algorithmic Classes”, PhD thesis, Univ.Basel, 1993.

[9] J.Darlington, A.J.Field, P.G.Harrison, P.H.J.Kelly, D.W.N.Sharp, Q.Wu: “Parallel Programming Using Skeleton Functions”

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

[11] Ted Lewis, Hesham El-Rewini: “Introduction to Parallel Computing”, Prentice-Hall International, 1992.

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

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