修士論文

プログラム区間ごとの特徴を考慮した効率的なスレッド統合手法

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 23 年度入学　23413541 番

祖父江 宏祐

平成 25 年 2 月 5 日

プログラム区間ごとの特徴を考慮した効率的なスレッド統合手法

祖父江 宏祐

内容梗概

消費電力や配線遅延の相対的増大という問題から，単一コアの性能向上によるプロセッサの高速化は難しくなってきている．この流れを受け，単一チップ上に複数のプロセッサコアを集積したマルチコア・プロセッサが広く普及してきている．このマルチコア・プロセッサのリソースを有効利用するための方法の一つとして，プログラムのマルチスレッド化がある．マルチスレッドプログラムにおいて，生成されるスレッドの数はプログラム実行効率に大きな影響を与えるため，適切なスレッド数を自動的に決定するような機構の必要性は高い．これを実現する機構として，Thread Tailor が提案されている． Thread Tailor は，プログラムを実行するプロセッサに適切なスレッド数を判断し，各プロセッサコアの処理量が均衡化されるようにスレッドを統合する．

しかしながら，Thread Tailor には 2 つの問題が存在する．1 つ目の問題として，プログラム開始時に 1 度しかスレッドを統合しないため，特定の処理区間で，あるスレッドの処理量が変動する場合に対応できず，実行速度の低下を招く恐れがあることが挙げられる．また 2 つ目の問題として，スレッドを統合することにより，ビジーウェイトを実行する際にデッドロックが発生してしまうことが挙げられる．

そこで本研究では 1 つ目の問題に対して，動的にスレッドを再統合する手法を提案する．この手法では，バリア同期ごとにスレッドの組み合わせを変化させ，各プロセッサコアの処理量を均衡化させる．また，2 つ目の問題であるデッドロックを防ぐために，トランスレータを作成し，それを用いてプログラムを書き換える手法を提案する．これにより Thread Tailor で実行可能なプログラムを拡充する．

提案した手法を実装し，SPLASH-2 ベンチマークプログラムおよび PARSEC ベンチマークプログラムを用いて評価を行った結果，動的スレッド再統合手法では既存の Thread Tailor に比べ，バリア同期が存在するプログラムでは平均 4.4%，最大 6.0%の実行速度の向上が得られることを確認した．また，プログラムを改変することによりデッドロックの発生を防ぎ，Thread Tailor の適用可能対象プログラムが拡大したことを確認した．さらに，これら 2 つの手法を組み合わせた手法では，コンテキストスイッチ回数増加に伴う実行時間の悪化を抑制することができた．

プログラム区間ごとの特徴を考慮した効率的なスレッド統合手法

目次

1 はじめに

これまでのプロセッサ高速化技術はスーパスカラに代表されるような命令レベル並列性 (Instruction-Level Parallelism: ILP) にもとづくさまざまな手法を中心としつつ，集積回路の微細化による高クロック化を半導体技術の向上により実現することで支えられてきた．しかしながらプログラム中の ILP には限界があり，命令レベルの並列化を行うだけではプロセッサの性能向上が頭打ちになりつつある．これらの流れを受け，単一チップ上に複数のプロセッサコアを集積したマルチコア・プロセッサが広く普及してきている．マルチコア・プロセッサでは，今までひとつのコアが担っていた仕事を複数のプロセッサコアで分担することで，単一コアでの実行よりもスループットを向上させることができる．つまり，スレッド並列性を利用してプログラムを並列に実行することで，実行時間の短縮が期待できる．

このマルチコア・プロセッサの性能を引き出すためには，プログラマはマルチスレッドプログラムを記述する必要がある．しかし，マルチスレッドプログラムを記述することは一般的に難しい．そのため，これを補助するために，MPI[1]，OpenMP[2] といった並列化フレームワークや Pthread[3] などのライブラリが提供されている．しかし，これらのフレームワークや Pthread ライブラリを用いたとしても，マルチコア・プロセッサの性能を十分引き出すことが可能なスレッド数をプログラマが選択することは容易ではない． それは，プログラマがプログラムを実行するプロセッサのリソースを考慮する必要があるからである．加えて，プログラムはさまざまなプロセッサ上で実行されることが想定されるため，一意にスレッド数を決定してしまうと，そのスレッド数に適さないプロセッサでは実行速度の低下を招く可能性があるからである．例えば，適切なスレッド数よりプログラマが指定したスレッド数が少なければ，マルチコア・プロセッサの提供するリソースを十分に利用できない．一方で，生成するスレッドが多ければ，スレッド間でのリソース競合の頻発や通信の増大を招き，プログラムの実行速度を低下させてしまう恐れがある．

この問題を解決するための研究として，**Thread Tailor**[4] がある．Thread Tailor は，プログラムを実行するプロセッサにとって適切なスレッド数を判断し，複数のスレッドを統合することによってスレッド数を調整する．Thread Tailor において重要となるのが，どのスレッド同士を統合すべきか決定する方法である．なぜなら，各プロセッサコアの負荷やスレッド間の通信量の視点から，同じスレッド数でも，どのスレッド同士を統合するのかによってプログラムの実行速度は大きく変化するからである．そ

こで，Thread Tailor では，まずプログラムをプロファイリングし，各スレッドの実行情報を収集する．そして，プロファイリング結果に対して，グラフ分割アルゴリズムを適用し，どのスレッド同士を統合すべきか決定する．最後に，その決定にもとづいてプログラム実行時にスレッドを統合する．このようにして，Thread Tailor はプログラム実行の高速化を実現している．

しかしながら，Thread Tailor には 2 つの問題が存在する．1 つ目の問題として，既存の Thread Tailor はプログラム開始時に 1 度しかスレッドを統合しないため，プログラムの負荷の変動に対応できないことが挙げられる．このため各プロセッサコアの負荷を均衡させることができず，実行速度の低下を招く恐れがある．2 つ目の問題として，プログラム実行時にデッドロック発生の恐れがあることが挙げられる．Thread Tailor ではスレッド同士の統合を実現する手段としてユーザスレッドが用いられている．しかし，ユーザスレッドを用いたプログラム実行では，スレッド間のコンテキストスイッチが自動的に行われないため，あらかじめプログラマがこれを明示する必要がある．Thread Tailor を用いた実行ではプログラマが明示しなくとも，ユーザスレッドがバリア関数に到達した際，Thread Tailor がスレッドをコンテキストスイッチする．しかしバリア関数のみの場合，あるユーザスレッドの処理の完了を他のユーザスレッドがループ内で待つ**ビジーウェイト**に対応できない．そのため，ビジーウェイトを含むプログラムを Thread Tailor を用いて実行した場合，複数のユーザスレッドが互いの処理の終了を待ち続けるデッドロックが発生し，実行が完了しない．

本研究では，1 つ目の問題を解決するために，全スレッドがバリア関数に到達した際にユーザスレッドの組み合わせを変更する手法を提案する．これにより，バリア関数ごとにより適切なスレッド統合を実現できるため，実行速度の向上が図れる．また，2 つ目の問題であるデッドロックを防ぐために，プログラムを改変するトランスレータを作成し，それを用いてプログラムコード内のすべてのループに対してユーザスレッドのコンテキストスイッチのためのコードを挿入する．これにより，プログラム内にビジーウェイトを行うループが存在したとしても，他のユーザスレッドに処理が移るためデッドロックを防止できる．つまり，Thread Tailor を用いてより多くのプログラムが実行可能になる．

以下，2 章では本研究で扱う Thread Tailor の概要および処理の流れを説明する．3 章ではバリア同期間ごとにスレッドを再統合する手法および Thread Tailor を用いて実行可能なプログラムを拡充する手法を提案し，4 章ではそれらの実装方法について述べる．そして，5 章で本提案手法の評価および考察を行い，6 章で関連研究について

説明する．最後に 7 章で結論を述べる．

2 Thread Tailor

本章では，本研究で対象とする Thread Tailor の概要および処理の流れについて説明する．

2.1 Thread Tailor の実行フロー

プログラマがマルチスレッドプログラムを書く際，マルチコア・プロセッサの性能を十分引き出すことが可能なスレッド数をプログラマが選択することは容易ではない．それは，プログラマがプログラムを実行するプロセッサのリソースを考慮する必要があるからである．加えて，プログラムはさまざまなプロセッサ上で実行されることが想定されるため，一意にスレッド数を決定してしまうと，そのスレッド数に適さないプロセッサでは実行速度の低下を招く可能性がある．この問題を解決するための研究として，Thread Tailor がある．Thread Tailor では，プロセッサのリソース情報およびプログラムのプロファイリング結果を利用して，実行スレッド数の決定および各プロセッサコアの処理量の均衡化を行う．これにより，プログラマがスレッド数を明示しなくとも Thread Tailor が適切なスレッド数を決定するため，マルチコア・プロセッサのリソースを効率的に使用できる．

Thread Tailor の処理の流れを**図 1** に示す．処理全体は，2 つのステージから構成される．1 つはプログラム実行前の処理（**静的ステージ**）であり，もう 1 つはプログラム実行時の処理（**動的ステージ**）である．

Thread Tailor では静的ステージにおいて，実行するスレッド数，およびどのスレッド同士を統合するかを決定するために，まずプロセッサの使用可能リソースを調査する．調査する項目は，プロセッサのコア数，2 次キャッシュ容量，メモリバンド幅容量である．次に，各スレッドの挙動を事前にプロファイリングする．この際に収集するデータは，並列化対象関数を実行した際の各スレッドの実行サイクル数，ワーキングセットサイズ，使用メモリバンド幅，そしてスレッド間の通信コストである．そして，プロファイリングによって得られた結果からコミュニケーショングラフと呼ばれる Thread Tailor 独自の形式のグラフを生成する．このグラフの各ノードは，スレッドの実行サイクル数，メモリバンド幅，ワーキングセットサイズを表し，ノード間のエッジはスレッド間の通信コストを表す．そして，このコミュニケーショングラフと先に取得したプロセッサのリソース情報を入力としてグラフ分割アルゴリズムを適用する

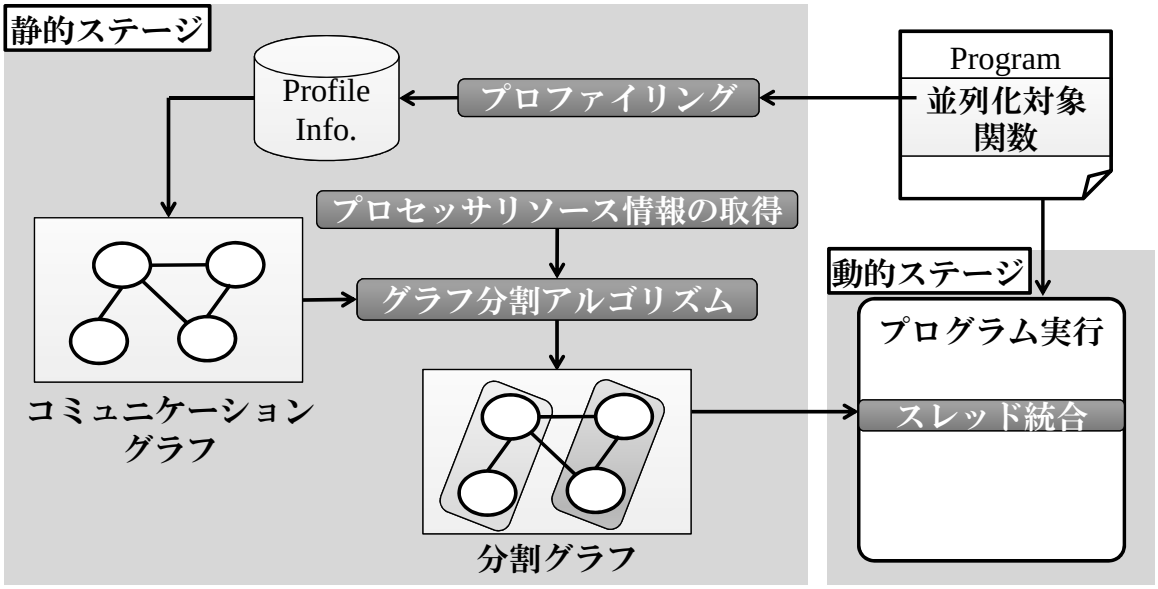


図 1: Thread Tailor の実行フロー

ことにより，複数のノードで構成されるグループを作成する．そして，そのグループにもとづいてどのスレッド同士を統合するかを決定する．ここでは，スレッド間でのリソース競合を起こさないという制約の下で，各プロセッサコアでの処理量を均衡化し，かつグループ間の通信の量をできる限り抑えるようにコミュニケーショングラフの分割を行う．

どのスレッド同士を統合すべきかが決定された後，動的ステージにおいて分割グラフにもとづいてプログラム実行開始時にスレッドを統合する．そして，統合されたスレッドは Thread Tailor によってスケジューリングされることで，プログラムの実行が進む．以降，本章では静的ステージにおけるプロセッサリソース情報の取得，プロファイリング，グラフ分割アルゴリズムについて順に説明し，最後に動的ステージにおけるスレッドの統合およびスレッドスケジューリングのための関数について詳しく述べる．

2.2 静的ステージ

本節では，Thread Tailor の静的ステージにおける各処理について概説する．

2.2.1 プロセッサリソース情報の取得

Thread Tailor は各プロセッサコアの処理量の均衡化およびスレッド間の通信量の削減のために，プログラムの実行に先駆けて，プログラムを実行するプロセッサの

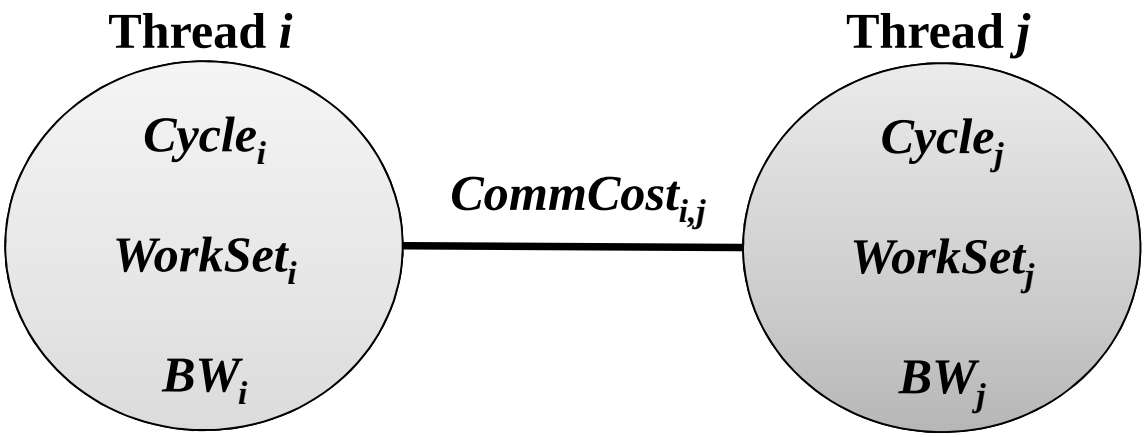


図 2: コミュニケーショングラフの構成

リソース情報を収集する．収集する情報は，プロセッサコア数，2 次キャッシュ容量（$CacheCapacity$），メモリバンド幅容量の近似値（$MemBW$）である．

まず，プロセッサコア数および 2 次キャッシュ容量を取得するために，Linux システム内にある/proc/cpuinfo を利用する．/proc/cpuinfo とは，Linux 内に存在するファイルであり，プロセッサの CPU 情報がテキスト形式で記載されている．Thread Tailor はこの/proc/cpuinfo 内に記載されたプロセッサコア数，2 次キャッシュ容量を取得する．

また，プロセッサのメモリバンド幅容量を取得するために，Thread Tailor は STREAM ベンチマークプログラム [5] を用いる．STREAM ベンチマークプログラムは，実行することでプロセッサのメモリバンド幅の実測値を計測することが可能である． このベンチマークプログラムは，複数のスレッドを生成し，値のコピー，値の加算，値の乗算，およびその全部の処理を繰り返し行うことでメモリバンド幅を飽和させる．そして，各処理ごとにプロセッサコアとメモリ間で発生した一定時間当たりの通信量をそれぞれ計測し，メモリバンド幅を算出する．Thread Tailor では，STREAM ベンチマークプログラムから得られたメモリバンド幅の平均値をプロセッサのメモリバンド幅容量の近似値として用いる．

2.2.2　各スレッドの挙動の解析

Thread Tailor は，プロセッサのリソース情報を取得した後，プログラムを実行した際の各スレッドの挙動を解析する．そして，解析によって得られたデータからコミュニケーショングラフを作成する．作成するコミュニケーショングラフの構成を**図 2** に示す．

この図は Thread i，Thread j から構成されるコミュニケーショングラフを表してい

る．コミュニケーショングラフのノードはそれぞれ各スレッドに対応し，当該スレッドのバリア同期によるストールサイクル数を除いた実行サイクル数（$Cycle_i$，$Cycle_j$），ワーキングセットサイズ（$WorkSet_i$，$WorkSet_j$），実行の際に必要とするメモリバンド幅（BW_i，BW_j）の 3 つの値を持つ．また，グラフのエッジは，その両端ノードが対応するスレッド間で発生する通信コスト（$CommCost_{i,j}$）を表す．なお，通信コストは通信に要するサイクル数の合計値である．このコミュニケーショングラフを作成するために一度プログラムを実行し，グラフ作成に必要なデータを収集する．

実行サイクル数およびメモリバンド幅については各プロセッサコアに備わっているハードウェア・パフォーマンス・カウンタ（HPC）[6] を使用して計測する．まず，HPC を使用して実行サイクル数を計測するために，Thread Tailor は RDTSC 命令 [7] を用いる．RDTSC 命令は，X86 アーキテクチャで定義されている命令で，CPU クロックごとに加算されるタイムスタンプカウンタの値を読み出す． Thread Tailor では，この命令をバリア命令の前後に挿入し，プログラムを事前に実行しておくことで実行サイクル数を計測する．しかしながら，スレッドは OS カーネルによりコンテキストスイッチされる場合があるため，スレッドを複数走らせ計測する方法では，あるスレッドの計測された実行時間に他のスレッドの実行時間まで含まれてしまう可能性がある．それを防ぐため，1 つのカーネルスレッド内で複数のユーザスレッドを用いてプログラムを実行することで他のスレッドの実行時間を含まない実行時間を計測する．

またメモリバンド幅を計測するために，Thread Tailor は OProfile[8] を用いる．OProfile とは，システム全体をプロファイル可能な Linux 向けのプロファイラであり，プロセッサコア内で発生したキャッシュミスなどのイベントの回数を取得することができる．Thread Tailor では，プロセッサコアとメモリ間で発生した通信回数を OProfile を用いて計測し，以下の計算式にその値を代入することでメモリバンド幅を算出する．

$$BW_i = \frac{Width \times Bus\_Tran\_Mem \times Freq}{Cycle_i} \tag{1}$$

式 (1) の $Width$ はプロセッサのバス幅，Bus_Tran_Mem は OProfile で取得するプロセッサコアとメモリ間で発生した通信回数，$Freq$ はプロセッサの実行周波数を表す．メモリバンド幅とは，一定時間内にプロセッサコアとメモリ間で発生した通信量であるため，プログラム実行時に発生した通信量（$Width \times Bus\_Tran\_Mem$）をスレッドの実行時間で除算することで算出できる．しかし，Thread Tailor では各スレッドの実行時間ではなく実行サイクル数を計測しているため，式 (1) のように通信量と $Freq$ を乗算し，$Cycle_i$ で除算することでメモリバンド幅を算出する．

Thread *i*

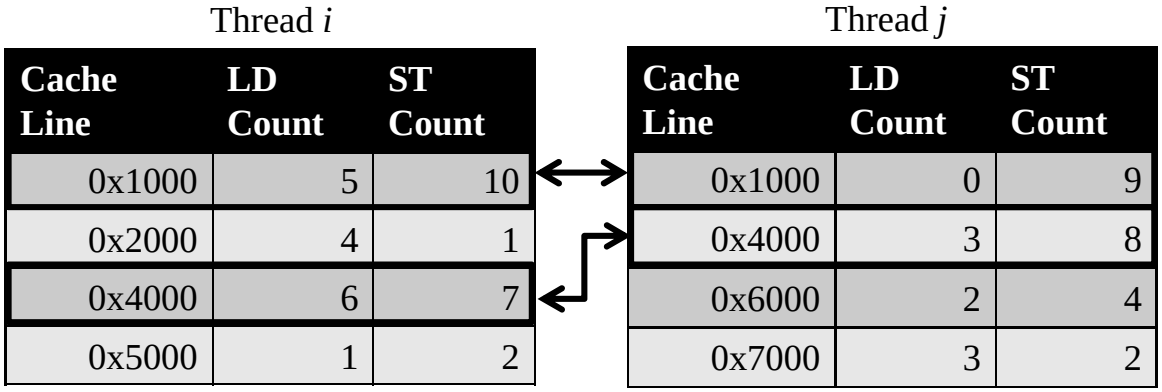
Cache Line	LD Count	ST Count	Sum
0x2000	5	10	15
0x1000	4	8	12
0x3000	1	2	3

Total: 30

図 3: Thread i がアクセスしたキャッシュラインをソートした結果

一方，ワーキングセットサイズと通信コストについては，HPC を用いても計測することができないため，何らかの方法でおおよその値を計測する必要がある．まず，ワーキングセットサイズを算出するために，Thread Tailor では各スレッドの全てのロード・ストア命令についてそれらのアクセス先アドレスを調べる．そのために，実行プログラムを llvm-gcc[9] を用いてバイトコードに変換する．そして，スレッド ID，ロード・ストア命令，アクセス先アドレスを出力するコードをバイトコードに挿入し，実行する．その後，得られた結果からスレッドごとの各キャッシュラインへのアクセス回数を調べる．その結果，**図 3** に示すような表を得る．Thread Tailor では，キャッシュラインをアクセス回数の総和（Sum）について降順にソートし，Sum が多い方から順にアクセス回数を加算していく．そしてその結果が，キャッシュラインへの総アクセス回数（Total）の 90%を超えた時点で，そのラインまでをワーキングセットとする．図 3 の場合，Thread i が 3 つのラインにアクセスしており，その合計が 30 回であるため，その 90%のアクセス回数を占める `0x2000` と `0x1000` の 2 ラインが，Thread i のワーキングセットと Thread Tailor は計測する．したがって，ワーキングセットサイズは，この 2 ラインにキャッシュブロックサイズを乗じた値になる．

なお通信コストも，このキャッシュラインへのアクセス情報を使用して算出する．ある 2 スレッド間において通信が発生するのは，一方がストアしたアドレスに対して，もう一方がロード又はストアした場合である．このとき，一方がストアすることで，キャッシュ上のストア先アドレスのデータが更新される．しかし，一般的なプロセッサでは，メインメモリのデータ更新方式としてライトバック方式がとられているため，メ



Cache Line	LD Count	ST Count
0x1000	5	10
0x2000	4	1
0x4000	6	7
0x5000	1	2

Cache Line	LD Count	ST Count
0x1000	0	9
0x4000	3	8
0x6000	2	4
0x7000	3	2

図 4: Thread i および Thread j のキャッシュラインアクセス情報

インメモリ上のデータは即座に更新されない．そして，ストアされたアドレスに対して，もう一方がロード又はストアによってアクセスしようとした場合，最新のデータを参照するために 2 スレッド間で通信する必要がある．このようなことから，Thread Tailor ではスレッド間の通信回数を求めるために，スレッド間で共通にアクセスされたキャッシュラインごとに以下の式を適用する．

$$\min(LDCount_i, STCount_j)+\min(STCount_i, LDCount_j)+\min(STCount_i, STCount_j) \quad (2)$$

式中の $LDCount_i$ は，Thread i がそのキャッシュラインからデータをロードした回数を表し，$STCount_i$ は Thread i がそのキャッシュラインにデータをストアした回数を表す．また，min 関数は括弧内の最小の値を返す関数である．この式から得られた通信回数を足し合わせた値を，スレッド間で発生した通信回数と Thread Tailor は計測する．

例えば，あるプログラムを実行した際の Thread i および Thread j のキャッシュラインアクセス情報を調査ところ，**図 4** に示す結果が得られたとする．この例では，Thread i および Thread j が互いにアクセスしているキャッシュラインは，`0x1000` および `0x4000` であるため，このラインにアクセスする場合，スレッド間で通信が発生する可能性がある．この時キャッシュライン `0x1000` に関して，Thread i，Thread j 間で通信が発生する回数は，上記の式にもとづいて $\min(5,9)+\min(10,0)+\min(10,9)=14$ であると予測する．また，キャッシュライン `0x4000` に対しても同様に計算し，$\min(6,8)+\min(7,3)+\min(7,8)=16$ となる．図 4 に示す例では，両スレッドからアクセスされるキャッシュラインは，`0x1000` および `0x4000` のみであり，これらのキャッシュライン以外へアクセスする際には通信は発生しない．したがって，Thread i，Thread j 間の

総通信回数は $14 + 16 = 30$ 回であると予測できる．このように，両スレッド間で通信が発生する回数を計算し，総通信回数を予測する．

また，通信にかかるサイクル数の合計を算出するためには，通信回数に 1 回の通信にかかるサイクル数（通信レイテンシ）を乗じる必要がある．プロセッサコア数を $Cores$，2 次キャッシュへのアクセスレイテンシを $L2AccessLatency$ とすると，通信レイテンシは，経験則にもとづき，

$$3 \times \sqrt{Cores} \times L2AccessLatency \tag{3}$$

で算出される．式 (3) ではキャッシュがメッシュ構造で，ディレクトリ型コヒーレンスプロトコルによりその一貫性が保たれていることを想定している．つまり，$\sqrt{Cores}$ は 2 コア間の距離の平均を表している．そして，3 という数字はデータの要求元からディレクトリへ，ディレクトリからデータの所持先へ，所持先から要求元へと 1 つデータあたり 3 回の通信が発生することを意味する．

以上で述べた計測方法により，各パラメータを取得する．そして，取得したパラメータをグラフ形式にまとめることでコミュニケーショングラフが作成される．

2.2.3 統合するスレッドの決定

前項で述べた方法により作成されたコミュニケーショングラフを，いくつかのノードグループに分けることで統合するスレッドを決定する．この処理は，グラフ分割と呼ばれる．グラフ分割では，各グループ内のノードの実行サイクル数の合計ができるだけ均等になり，かつ各グループ間の通信コストが小さくなるようにコミュニケーショングラフが分割される．これを実現するために，Thread Tailor は Kernighan-Lin アルゴリズム（KL アルゴリズム）[10] と呼ばれるグラフ分割アルゴリズムを用いる．一般的にグラフ分割問題は NP 困難であることが知られているが，KL アルゴリズムは，NP 困難な問題においてもグラフのサイズのオーダーで部分最適解を求めることが可能である．KL アルゴリズムでは，はじめに，グラフを同じ数のノードを持つ 2 つのグループに分割する．そして，2 つのグループ間でノードを交換することにより，グループ間を跨ぐエッジの本数を削減していく．しかし，コミュニケーショングラフにはノードも値を持っているため，エッジのみを考慮しただけではグループ間で実行サイクル数を均衡化することはできない．そのため，Thread Tailor は KL アルゴリズムを改良し，各グループ内のノードが持つ実行サイクル数の合計が均衡化するようにノードを何度も交換する．

グラフ分割アルゴリズムを使用してコミュニケーショングラフを 2 つのグループに

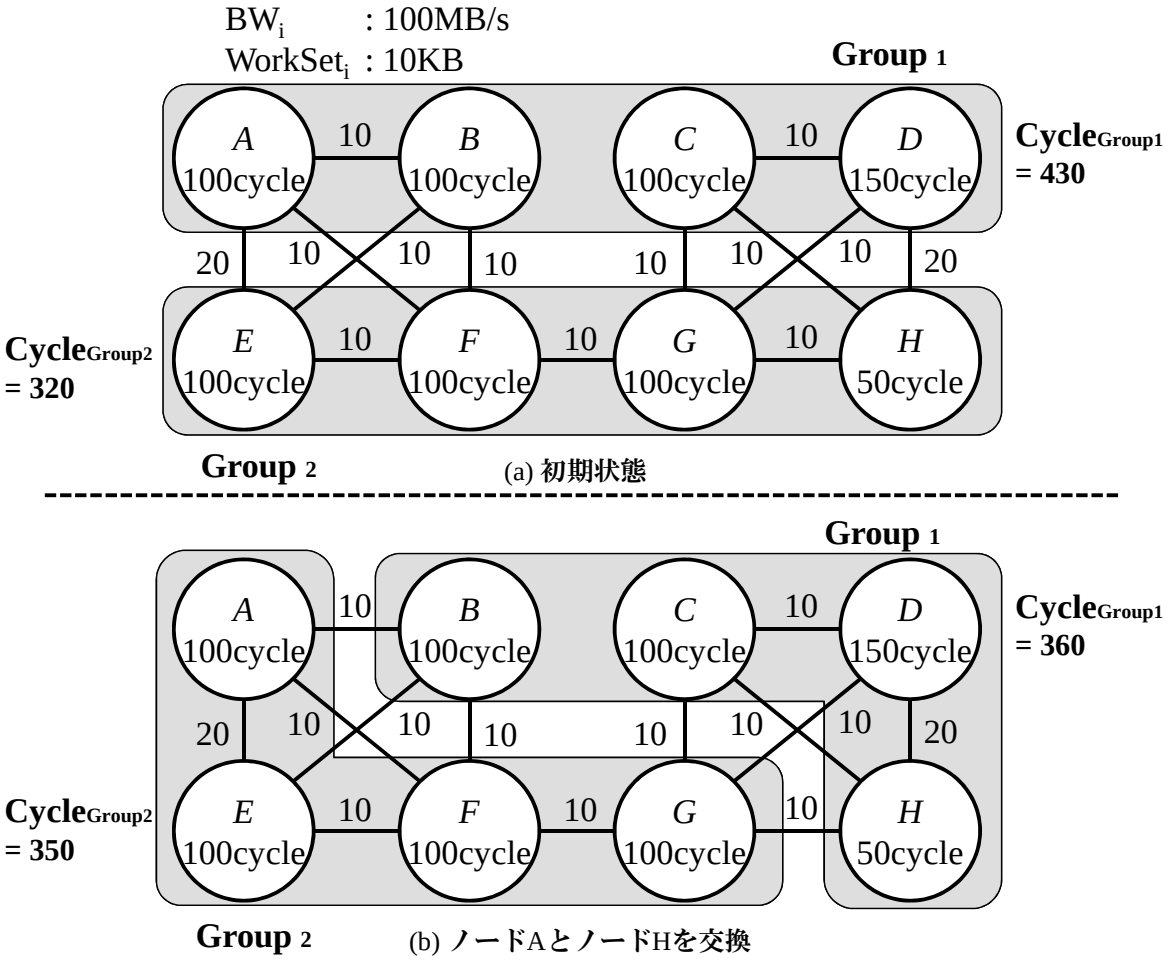


図 5: グラフ分割アルゴリズムの動作例

分割し，ノードを交換する様子を**図 5** に示す．各ノードは図 5(a) に示す実行サイクル数，100MB/s のメモリバンド幅，10KB のワーキングセットを持つ．まず図 5(a) に示すように，ノード A，B，C，D からなるグループ 1 と，ノード E，F，G，H からなるグループ 2 の 2 つのノードグループに初期分割されたとする．Thread Tailor では，各グループの実行サイクル数，ワーキングセットサイズ，メモリバンド幅が以下の式で表される．

$$Cycles_{Group_k} = \sum_{i \in Group_k} Cycles_i - \sum_{i \in Group_k, j \in Group_k} CommCost_{i,j} \tag{4}$$

$$WorkSet_{Group_k} = \sum_{i \in Group_k} WorkSet_i \tag{5}$$

$$BW_{Group_k} = \sum_{i \in Group_k} BW_i \tag{6}$$

式 (4) に示すように各グループの実行サイクル数は，ノードの持つ実行サイクル数の総和からグループ内の通信サイクルを減じた値である．また，式 (5)，式 (6) に示すように各グループのワーキングセットサイズおよびメモリバンド幅は，グループ内のそれぞれの値の総和である．そのため，グループ 1 およびグループ 2 の実行サイクル数はそれぞれ，430cycle，320cycle，メモリバンド幅は 400MB/s，400MB/s，ワーキングセットサイズは 40KB，40KB と算出できる．

ここで，最大の実行サイクル数を持つグループの実行サイクル数が減少するように，グループ 1 とグループ 2 から交換するノードの組み合わせを探す．その際に，Thread Tailor では各スレッド間のリソース競合を抑えるために，各グループのワーキングセットサイズとメモリバンド幅のそれぞれがプロセッサの 2 次キャッシュサイズとメモリバンド幅容量を超えないようにしなければならない．そのため，以下の条件を満たしながらノードを交換する必要がある．

$$WorkSet_{Group_k} \leq CacheCapacity \tag{7}$$

$$BW_{Group_k} \leq MemBW \tag{8}$$

図 5(a) の場合，ワーキングセットサイズ，メモリバンド幅は全てのノードで等しいため，どのノードを交換しても常にこれら条件を満たしていることが分かる．

また Thread Tailor では，交換することで最も実行サイクル数を削減できるノードを見つけるために，以下の計算式を用いる．

$$T_{Group_k} = \sum_{i \in Group_k} Cycles_i - Cycle_{my} + Cycle_{other} - I_{Group_k} \tag{9}$$

$$Gain = MaxExe - Max(T_{Group_k}) \tag{10}$$

式 (9) に示す T_{Group_k} は，自グループ内のノード（$Cycle_{my}$）および他グループ内のノード（$Cycle_{other}$）を交換した場合の自グループの実行サイクル数を表し，I_{Group_k} は $Cycle_{my}$ と $Cycle_{other}$ を交換した場合のグループ k 内の通信に要するサイクル数を表す．また，式 (10) に示す $MaxExe$ は，最大の実行サイクル数を表し，$Gain$ はノードを交換することによって最大の実行サイクル数がどれだけ削減されるのかを表す．

例えば図 5(a) において，ノード A およびノード G を交換した場合に得られる $Gain$ を考える．ノード A およびノード G は共に実行サイクル数が 100cycle であり，また，これらのノードを交換した場合の各グループ内の通信サイクルは $I_{Group_1} = 30$，$I_{Group_2} = 40$ と分かる．そのため，$T_{Group_1} = 450 - 100 + 100 - 30 = 420$，$T_{Group_2} = 350 - 100 +$

$100 - 40 = 310$ と算出できる．以上の結果より，ノード A およびノード G を交換した場合，全グループ内で最大の実行サイクル数を持つグループはグループ 1 の 420cycle と分かるため，$MaxExe$（430cycle）から T_{Group_1} を減算し $Gain = 430 - 420 = 10$ であると分かる．

このようにして，グループ 1，グループ 2 の間の全てのノードの組み合わせにおいて $Gain$ を計算し，その値が正でかつ最も大きくなる 2 つのノードを交換する．図 5(a) に示したグラフの例では，ノード A とノード H を交換することにより最も大きい $Gain$ が得られるとグラフ分割アルゴリズムにもとづいて判断されるため，これらのノードを交換する．これによって，各グループの実行サイクル数は図 5(b) に示すように，$Cycle_{Group_1} = 360$，$Cycle_{Group_2} = 350$ となる．ノード A とノード H を交換した後，再度各グループのノードを交換することにより，正の $Gain$ が得られるか調査する．図 5(b) の場合，これ以上ノードを交換しても正の $Gain$ が得られないため，Thread Tailor は図 5(b) に示す結果が最適解であると判断する．Thread Tailor では統合するスレッドを決定するために，この様な手順でコミュニケーショングラフから複数のノードグループを作成する．

また，これらの動作はノードグループ数がシステムのプロセッサコア数と同じ数になるまで繰り返し適用される．そのため，Thread Tailor では 4 つ以上のノードグループを作成する場合，上記のようにノードを交換して作成された各グループをさらに 2 つのノードグループに分割する．そして，2 グループ間ごとにグラフ分割アルゴリズムを適用する．この動作を繰り返し，プロセッサのコア数と同数のグループを生成した時点でアルゴリズムは停止する．

2.3 動的ステージ

本節では，Thread Tailor の動的ステージにおける処理およびユーザスレッドを制御するための関数について概説する．

2.3.1 スレッド統合

2.2.3 項で述べたグラフ分割アルゴリズムの結果にもとづいて，プログラム実行開始時に Thread Tailor は複数のスレッドを統合し，スレッド数を調整する．これを実現するために，Thread Tailor ではカーネルスレッドとユーザスレッドという 2 種類のスレッドを用いる．カーネルスレッドとは，OS カーネルによりスケジューリングされるスレッドであり，マルチコア・プロセッサ上であれば，同じプロセス内の複数のスレッドを並行して実行することもできる．一方，ユーザスレッドとは，プログラマが明示

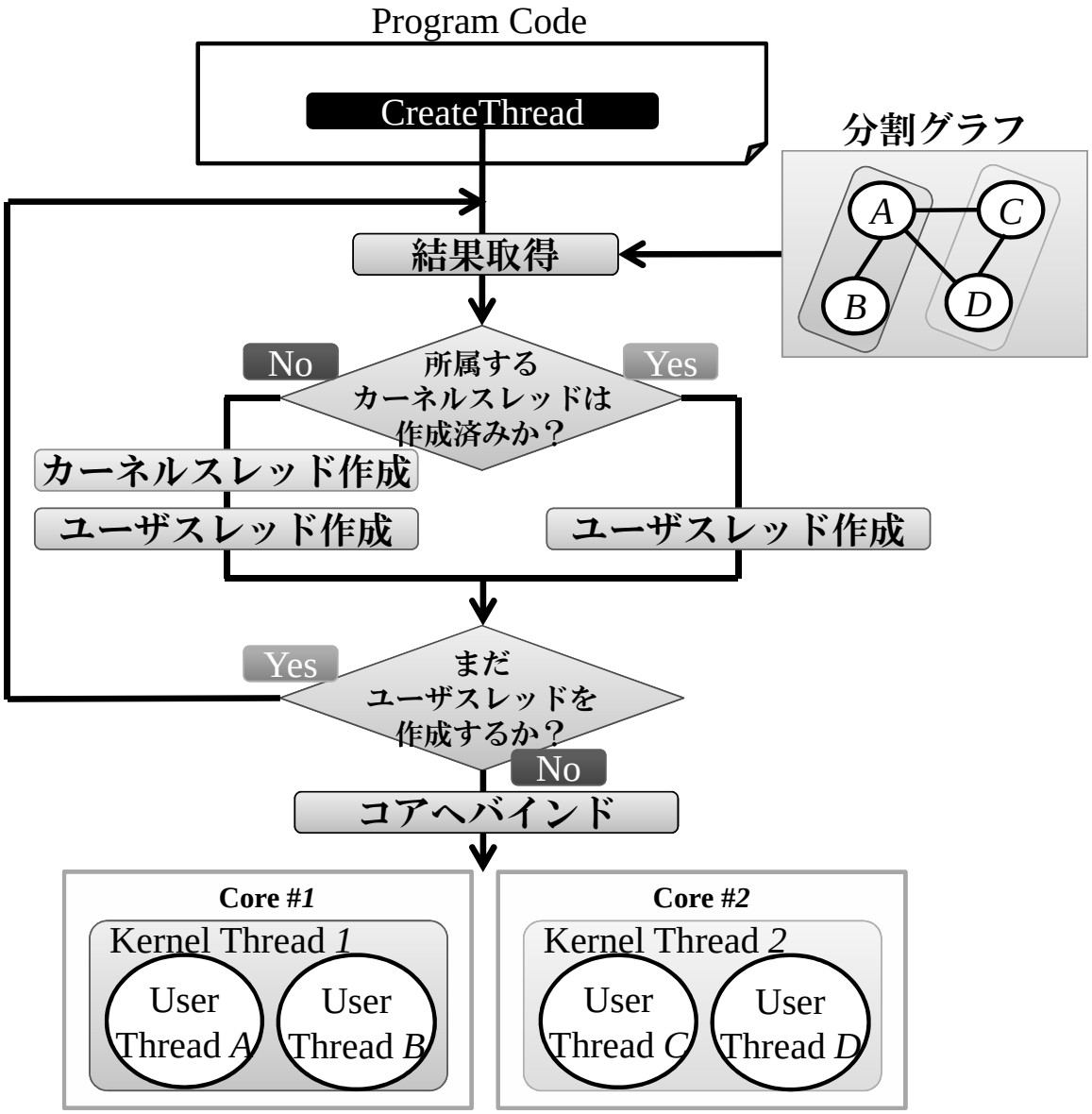


図 6: スレッド統合の流れ

的にスケジューリングするスレッドであり，プロセス内で複数のユーザスレッドを作成したとしても，OS カーネルからは単一のプロセスとして認識される．Thread Tailor では，分割グラフの各グループをカーネルスレッドに対応付け，各ノードをユーザスレッドに対応付けることでスレッド統合を実現する．つまり，Thread Tailor を用いたプログラム実行では，カーネルスレッド内で複数のユーザスレッドを実行させる．

スレッド統合の手順を**図 6** に示す．この例で使用する分割グラフでは，ノード A とノード B が，ノード C とノード D が同一のグループに所属している．まず，スレッドを生

成する関数（CreateThread）が呼び出されると，Thread Tailor に処理が移る．Thread Tailor はまず，ユーザスレッド A（User Thread A）を生成するために静的ステージで生成した分割グラフを参照する．この時，User Thread A が所属するカーネルスレッド（Kernel Thread 1）はまだ作成されていないので，Thread Tailor は pthread_create 関数を使用して Kernel Thread 1 を作成し，User Thread A のコンテキスト情報を生成する．次に，User Thread B を作成するために，再び分割グラフを参照する．分割グラフから User Thread B と User Thread A は同じカーネルスレッドに所属することが分かる．しかし，User Thread B が所属するカーネルスレッドは，User Thread A 作成時に生成されているため，User Thread B のコンテキスト情報のみ生成すれば良い．このような手順にしたがって全てのカーネルスレッドおよびユーザスレッドが生成されると，各カーネルスレッドはコアへバインドされる．そして，各ユーザスレッドは並列化対象関数の実行を開始する．Thread Tailor を用いた実行では，このようにしてスレッドの統合が実現される．

2.3.2 ユーザスレッド制御関数

前項で述べたように，Thread Tailor では，複数のユーザスレッドを用いて実行しているため，ユーザスレッドをコンテキストスイッチする必要がある．しかしながら，OS カーネルはユーザスレッドのコンテキストスイッチ処理を担当していないため，Thread Tailor がこれを行う必要がある．また，Thread Tailor を用いてバリア同期が存在するプログラムを実行する場合，バリア関数で全てのユーザスレッドを同期する必要がある．しかし，並列プログラミングで一般的に使用される Pthread ライブラリはユーザスレッド間のバリア同期に対応していないため，バリア同期に関しても Thread Tailor がこれを行う必要がある．以上のことから，Thread Tailor にはユーザスレッド用のコンテキストスイッチ関数およびバリア同期関数が実装されている．

まず，Thread Tailor 内に存在するユーザスレッドコンテキストスイッチ用の関数 ContextSwitch のフローを**図 7** に示す．まず，あるユーザスレッドが ContextSwitch 関数内に到達すると，そのユーザスレッドは所属するカーネルスレッド番号を調査する．そして，自身と同じカーネルスレッド内に所属するユーザスレッドの内，次に実行されるべきユーザスレッドのコンテキストを取得する．最後に自身のユーザスレッドコンテキストと，コンテキストスイッチ先のユーザスレッドコンテキストを引数に swapcontext 関数を実行する．swapcontext 関数内では，現在のコンテキストを保存し，次に実行するユーザスレッドのコンテキストを有効にする処理が行われる．Thread Tailor は，このようにしてユーザスレッド間でのコンテキストスイッチを実現する．

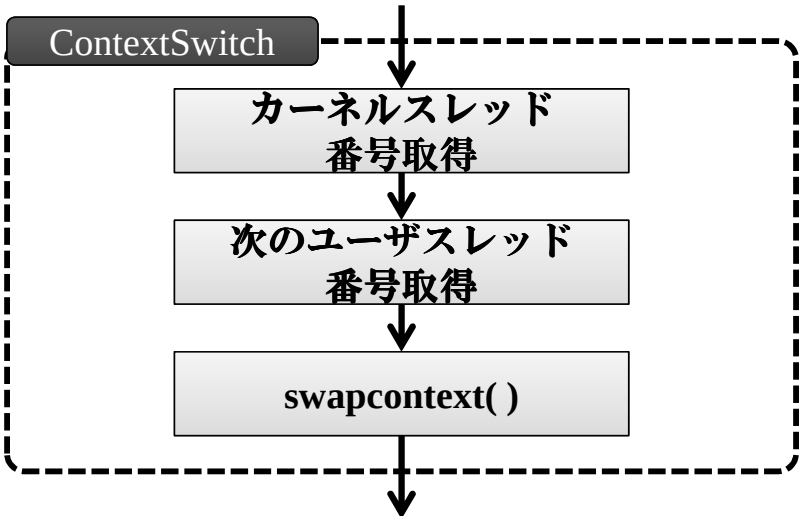


図 7: ユーザスレッドコンテキストスイッチ関数の処理の流れ

もうひとつ Thread Tailor が提供するバリア同期関数 Barrier のフローを**図 8** に示す．Thread Tailor ではバリア同期を実現するために，2 種類のフラグと Counter 変数を使用する．フラグはそれぞれ，各ユーザスレッドがバリア関数に到達したことを示す Local_flag と，全てのユーザスレッドがバリア関数に到達したことを示す Global_flag である．なお，これらのフラグは 0 で初期化されている．Counter 変数は，バリア関数に到達していないユーザスレッドが存在するカーネルスレッドの個数（N）を表している．そのため，この変数の値は N で初期化されている．また，この Counter 変数は全てのカーネルスレッド内のユーザスレッドから共通して読み書きされる．そのため，Counter 変数を操作する命令はクリティカルセクションで保護されている．図 8 に示すように，まず，あるユーザスレッドがバリア関数に到達すると，自身に対応する Local_flag を反転させる．そして，自身が所属するカーネルスレッド内の他のユーザスレッドの Local_flag を参照し，自身がそのカーネルスレッド内で最後にバリア関数に到達したユーザスレッドであるか調べる．もし，まだバリア関数に到達していないユーザスレッドが存在するならば，そのユーザスレッドにコンテキストスイッチする．一方，自身がそのカーネルスレッド内で最後にバリア関数に到達したユーザスレッドであるならば，他のカーネルスレッド内に存在するユーザスレッドが全てバリア関数に到達しているかどうか調査するために Counter 変数の値を参照する．ここでもし，自身の属するカーネルスレッドが最後に当該バリアに到達したカーネルスレッドではない場合，つまり，Counter 変数の値が 1 よりも大きい場合，Counter 変数をデクリメントし，バリア関数を通過する許可が出るまで待機する．一方，自身が最後に到達した

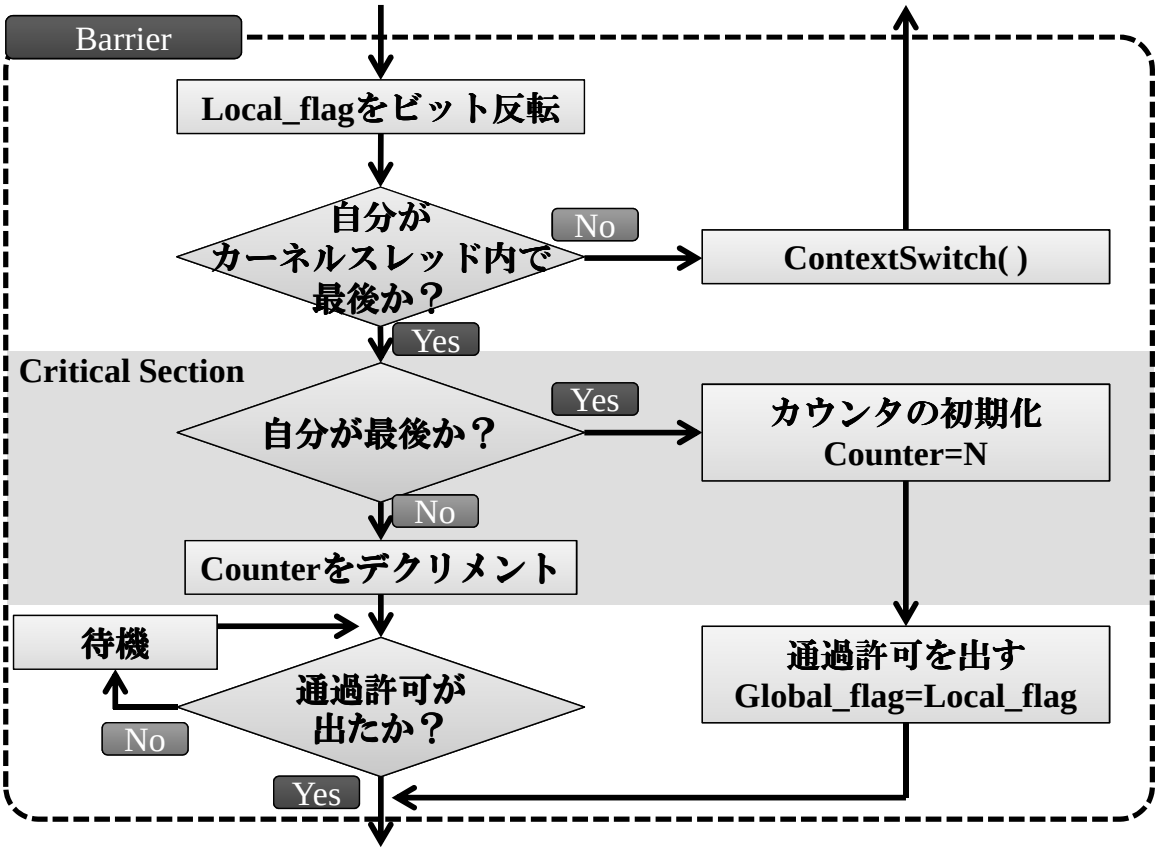


図 8: ユーザスレッドバリア同期関数の処理の流れ

ユーザスレッドならば，Counter 変数を N で初期化し，Local_flag の値を Global_flag に代入することで全ユーザスレッドに対して通過許可を出す．このようにして，Thread Tailor はユーザスレッド間のバリア同期を実現する．

3 Thread Tailor の改良

2 章で述べた Thread Tailor にはいくつかの問題点が存在する．本章では Thread Tailor の問題点を示し，それらの問題を解決する手法をそれぞれ提案する．

3.1 Thread Tailor の問題点

本節では，Thread Tailor を用いてプログラムを実行した際に発生する 2 つの問題点について述べる．

3.1.1 スレッドの負荷の変動に伴う性能低下

Thread Tailor を用いた実行では，スレッド生成のための関数が呼ばれると，カーネルスレッドおよびユーザスレッドが生成される．そして，分割グラフにもとづいて，複数のユーザスレッドがカーネルスレッド内にまとめられ，並列化対象関数の実行が開始される．しかしながら，スレッド統合はプログラム開始時，1 度しか行われないため，プログラム実行中にユーザスレッドの組み合わせが変更されることはない．これは，Thread Tailor は並列化対象関数のプロファイリング結果にもとづいてどのユーザスレッドを統合するべきであるかを考慮しているからである．それゆえ，特定の処理区間で，あるユーザスレッドの処理量が変動するようなプログラムを Thread Tailor を用いて実行する場合，各プロセッサコアの負荷の不均衡化および通信量の増大のために実行速度が低下する恐れがある．そのようなプログラムの例を**図 9** に示す．

図 9 は，Parallel 関数内の処理を 3 本のスレッドを用いて実行するプログラムを表している．この Parallel 関数は主な処理として，複数のスレッドによって大域変数 A および B の値をそれぞれ 1000 回ずつインクリメントする処理と，単一のスレッドによってローカル変数 C の値を 1000 回インクリメントする処理から構成されている．各大域変数は複数のスレッドから読み書きされるため，排他制御により資源の競合を防いでいる．また，$Thread\_ID$ 変数は各スレッドの識別子が格納される変数である．

まず，Parallel 関数が開始されると if 文（4 行目）により，各スレッドの処理が分岐する．そのため，Thread 1 および Thread 2 は大域変数 A を互いにインクリメントし（7 行目），Thread 3 は，ローカル変数 C をインクリメントする（12 行目）．この区間（4-13 行目）では，Thread 1 および Thread 2 が大域変数に対して読み書きするため，これらのスレッド間で多量の通信が発生する一方，Thread 3 は自身のローカル変数の読み書きをするため，通信は発生しないと考えられる．また，Thread 2 および Thread 3 が大域変数 B のインクリメントを含むループ（15-20 行目）を実行する際も，上記と同様の理由から Thread 2 および Thread 3 の間で多量の通信が発生すると考えられる．つまり，このプログラムを上記のスレッド構成で実行すると，最初の if，else 内の処理（4-13 行目）と 2 つ目の if，else 内の処理（15-24 行目）で異なる通信パターンが観測されると考えられる．

しかしながら，Thread Tailor を用いてこのプログラムを実行する場合，Parallel 関数の実行を通して得られたプロファイリング結果にもとづいて適切な組み合わせでスレッドが統合される．つまり，このプログラムのような処理区間によって処理量やスレッド間の通信に偏りがあるプログラムでは，Thread Tailor を用いた実行においても

```
1 int A, B;
2 void Parallel(int Thread_ID){
3   int i, C;
4   if(Thread_ID = 1 || Thread_ID = 2){
5     for(i=0; i<1000; i++){
6       lock();    // スレッド1とスレッド2が
7       A++;       // 大域変数Aを
8       unlock(); // インクリメント
9     }}
10  else{
11    for(i=0; i<1000; i++)
12      C++;
13  }
14  :
15  if(Thread_ID == 2 || Thread_ID == 3){
16    for(i=0; i<1000; i++){
17      lock();    // スレッド2とスレッド3が
18      B++;       // 大域変数Bを
19      unlock(); // インクリメント
20    }}
21  else{
22    for(i=0; i<1000; i++)
23      C++;
24  }
25  :
26 }
```

図 9: 実行速度低下の恐れがあるプログラム

実行速度が向上しない恐れがあることが分かる．以上のことから，プログラムの挙動の変化にもとづいてユーザスレッドの組み合わせを変更することで，さらなる高速化を達成できると考えられる．

3.1.2 実行が進まないスレッドの発生

2.3 節で述べた様に，Thread Tailor を用いたプログラム実行では，ユーザスレッドを使用しているため Thread Tailor がユーザスレッドをコンテキストスイッチする必要がある．しかし Thread Tailor では，コンテキストスイッチのタイミングをバリア同期時のみに限定しているが，バリア同期時のみではユーザスレッドがループから抜けら

```
1  :
2  pthread_mutex_lock(&(global->pbar_lock));
3  global->pbar_count++ ;
4  pthread_mutex_unlock(&(global->pbar_lock));
5
6  while( global->pbar_count < n_processors ){
7    :
8  }
9  :
```

図 10: radiosity のプログラムコード（部分）

```
1  :
2  while (!tasks[MyNum].taskQ && !tasks[MyNum].probeQ){
3    ;
4  }
5  :
```

図 11: cholesky のプログラムコード（部分）

れず実行が完了しないプログラムが存在する．

Thread Tailor を使用した場合，実行が完了しないプログラム例を**図 10** および**図 11** に示す． 図 10 は，SPLASH-2 ベンチマークプログラム [11] に含まれる radiosity の部分コードを表している．なお図 10 に示す部分コードの $global->pbar\_count$ 変数はループに到達したスレッドの個数を表し，$n\_processors$ 変数は並列化対象関数を実行しているスレッドの個数を表している．$global->pbar\_count$ 変数は複数のスレッドから読み書きされるため，排他制御（2 行目，4 行目）により資源の競合を防いでいる．各スレッドは，自身が while ループ内に入る前に，$global->pbar\_count$ 変数の値をインクリメントする（3 行目）．そして，そのスレッドは全てのスレッドが while ループ（6 行目）に到達するのを待つ．全てのスレッドが while ループに到達すると，$global->pbar\_count$ の値が $n\_processors$ の値と等しくなるため，各スレッドは while ループを抜けて次の処理に移る．

しかし，Thread Tailor を用いてこのプログラムを実行した場合，全てのユーザスレッドがこの while ループに到達せず，実行が完了しない．それは，この while ループ内にバリア同期命令またはユーザスレッドコンテキストスイッチ命令が存在しないた

め，あるユーザスレッドがプロセッサコアを占有し続け，他のユーザスレッドが待機させられ続けられるからである．例えば，4 本のユーザスレッドを 2 本ずつに統合し，2 本のカーネルスレッドを用いてこのコードを実行した場合を考える．この場合，各カーネルスレッド内の片方のユーザスレッドは while ループに到達することができるが，そのユーザスレッドはコンテキストスイッチしないため，もう一方のユーザスレッドは待機し続ける．つまり，カーネルスレッド 1 本あたり，ユーザスレッド 1 本しかループ内に到達できない．このようなことから，デッドロック状態に陥ってしまう．

一方，図 11 は SPLASH-2 ベンチマークプログラムに含まれている cholesky の部分コードを示している．図 11 の $MyNum$ 変数は並列化対象関数を実行しているスレッドの ID を表している．$tasks$ は各スレッドが処理する $taskQ$ および $probeQ$ を保持する構造体であり，スレッドごとに生成されるため，自身の構造体を示す識別子として $MyNum$ 変数が使用されている．また，自スレッドの処理対象である $taskQ$ および $probeQ$ は，他のスレッドから書き込まれるため，図 11 に示す while ループで $tasks[MyNum].taskQ$ および $tasks[MyNum].probeQ$ が書き込まれるまでスレッドは待機する．

しかし，Thread Tailor を用いてこのプログラムを実行した場合，この while ループを抜けることができないスレッドが存在するため，実行が完了しない．それは，radiosity と同様に，while ループ内にユーザスレッドコンテキストスイッチのためのコードが存在しないため，$tasks$ 構造体の $taskQ$ および $probeQ$ に値を書き込むスレッドに CPU リソースが割り当てられず，その処理の完了を待つスレッドは while ループを抜けられないためである．このような理由から，図 11 に示すような処理を含むプログラムを Thread Tailor を用いて実行する場合もプログラムの実行が完了しない．以上のことから，全てのスレッドがループ内に到達するのを待つ処理，またはあるスレッドの処理が終了するのを他のスレッドがループ内で待つ処理がプログラム中に存在する場合，Thread Tailor を用いた実行は完了しないことが分かる．

3.2 動的スレッド再統合

本節では，まず 3.1.1 項で説明した問題を解決するために，バリア同期ごとにスレッドを再統合する手法を提案する．

3.2.1 バリア同期間ごとのスレッド再統合

3.1.1 項で述べた様に，特定の処理区間で，あるユーザスレッドの処理量が変動するプログラムを，Thread Tailor を用いて実行する場合，各プロセッサコアの負荷が均衡化せず，また通信量の増大のために実行速度が低下してしまう恐れがある．この問題

を解決するためには，プログラムの挙動が変化するタイミングでユーザスレッドの組み合わせを変更し，各プロセッサコアの負荷の均衡化およびスレッド間の通信量の削減を図るべきであると考えられる．しかし，事前プロファイリング等によりプログラムの挙動が変化するタイミングを正確に検出することは困難である．なぜなら，並列化対象関数の処理内容と実行サイクル数を照らしあわせ，どのような処理区間でどのスレッドの処理量が増加したのか調査する必要があるからである．

そこで本研究では，簡易的に動的スレッド再統合手法を実現するため，バリア同期に着目し，バリア同期時にユーザスレッドを再統合する手法を提案する．提案する手法は Thread Tailor とは異なり短い区間でユーザスレッドの組み合わせを変更するため，Thread Tailor より高速にプログラムを実行できると考えられる．

なお本研究でバリア同期に着目した理由は 2 つある．1 つ目の理由として，バリア同期では全てのスレッドが一時的に実行を停止することが挙げられる．スレッドの組み合わせを変更する場合，スレッドの実行を一度停止させる必要があるが，バリア同期で再統合をする場合，Thread Tailor に特別な処理を追加することなく，容易に提案手法を実装することが可能である．2 つ目の理由として，バリアの前後でプログラムの挙動が変化する可能性が高いことが挙げられる．バリア同期が存在するプログラムでは，結果の整合性を保つため，各スレッドが処理した結果を一度集計した後，次の処理に進む可能性が高い．つまり，各スレッドのバリア同期後の処理内容は同期前の処理内容と異なることが多いと予想されるため，各スレッドの処理量も変化する場合が多いと考えた．以上 2 つの理由から，バリア関数でユーザスレッドを再統合する手法を提案する．

3.2.2 予備評価

バリア関数でユーザスレッドを再統合することの妥当性を検証するために予備評価を行った．予備評価では，プログラム内のバリア同期間ごとの実行サイクル数および並列化対象関数を通した実行サイクル数を計測した．予備評価には，SPLASH-2 ベンチマークプログラムの barnes を使用し，4 つのコアが内蔵されているプロセッサ上で 16 本のスレッドを使用し実行した．**表 1** に barnes を実行した際の一部のスレッドの実行サイクル数をそれぞれ示す．

表 1 内の Phase はバリア同期間ごとの実行サイクル数を表し，Sum は並列化対象関数の実行サイクル数を表している．また，下線が引かれている実行サイクル数は，その Phase および Sum 内の最大の実行サイクル数を表す．表 1 を見ると，Phase X，Phase Y，Sum の全てで最大の実行サイクル数を持つスレッドが異なっていることが分かる．

表 1: barnes を実行した際のバリア同期間ごとの実行サイクル数（一部）

	Thread 0	...	Thread 3	...	Thread 15
...	...	...	...	...	...
Phase X	2,336,700	...	1,699,472	...	2,355,357
Phase Y	296,850	...	176,227	...	288,654
...	...	...	...	...	...
Sum	342,073,821	...	529,612,583	...	449,629,402

詳しく見てみると，Phase X では，Thread 15 の実行サイクル数が Thread 3 の実行サイクル数に比べ約 30%大きいと分かる．一方で Sum では，Thread 10 の方が Thread 15 に比べ約 15%大きい．ここから，バリア同期間ごとに各スレッドの処理量は比較的異なり，それゆえ既存手法のスレッド統合は適切でないことが分かる．

以上のことから，バリア同期ごとにスレッドを再統合することで既存の Thread Tailor 以上にプロセッサコアの処理量を均衡化させることができ，プログラムの実行速度を向上させる可能性があると考えられる．

3.2.3 新たに追加するパラメータ

3.2.1 項で述べた様に，動的スレッド再統合手法は，バリア同期ごとにユーザスレッドを再統合する．しかし，ユーザスレッドの組み合わせを変化させることにより，既存の Thread Tailor では想定されていないユーザスレッドマイグレーションが発生する可能性がある．スレッドマイグレーションとは，プロセッサの負荷を均衡化させるために，実行中のスレッドが一度実行を停止し，他のプロセッサコアに移動することである．一般的にスレッドマイグレーションを行うとそれに伴ってオーバヘッドが発生するが，マイグレーションが発生しない既存の Thread Tailor のパラメータのみを使用してグラフを分割した場合，このオーバヘッドによる性能低下を考慮できない．それゆえ，ユーザスレッドのより適切な組み合わせを発見するために，動的スレッド再統合手法ではスレッドマイグレーションに伴って発生するオーバヘッドを新たに計測し，グラフ分割でそれを使用する必要があると考えられる．

ユーザスレッドマイグレーションに伴って発生するオーバヘッドに関して**図 12** を用いて説明する．図 12(a) に示す例では，Core 1，Core 2 にそれぞれ Kernel Thread 1，Kernel Thread 2 がバインドされ，そして各カーネルスレッド内で User Thread A と User Thread B，および User Thread C と User Thread D がそれぞれ統合されている．まず，User Thread B がバリア同期前に，あるメモリアドレスにアクセスしたとす

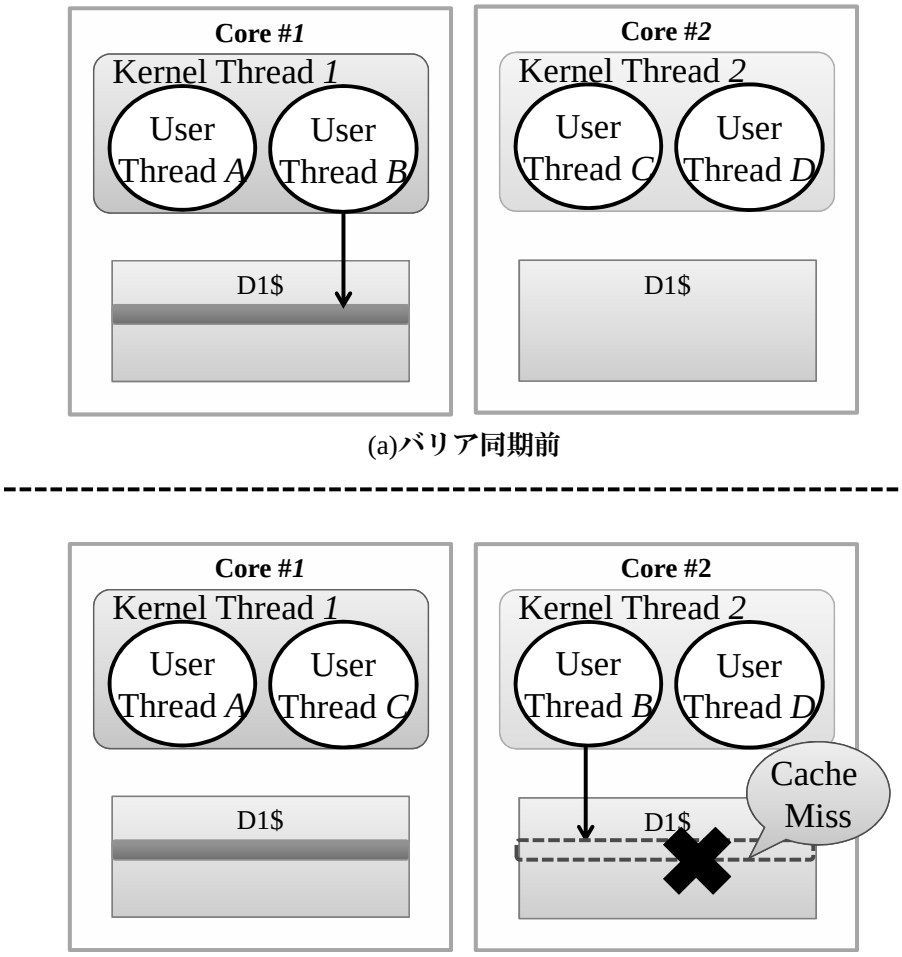


図 12: スレッドマイグレーションに伴ってキャッシュミスが発生する例

る．そして，当該アドレス上のデータは Core 1 の 1 次キャッシュに存在し，キャッシュヒットしたとする．その後，全てのユーザスレッドがバリア同期に到達すると，ユーザスレッドは再統合される．この例では，図 12(b) に示す様に，User Thread B と User Thread C が交換される形で他のプロセッサコアにマイグレートされたとする．そして，バリア同期後に User Thread B が再び同メモリアドレスにアクセスしたとする．しかし，当該アドレス上のデータは Core 2 の 1 次キャッシュに存在せず，User Thread B はバリア同期後に 1 次キャッシュミスを引き起こす．これは，User Thread B がマイグレートしたことによって発生したキャッシュミスであり，既存の Thread Tailor を用いた実行では発生しない．つまり，動的スレッド再統合手法を用いた実行の場合，より適切にコミュニケーショングラフを分割するために，このキャッシュミスを考慮す

る必要がある．

そのため，スレッドマイグレーションに伴って発生する 1 次キャッシュミスをオーバヘッドと考え，動的スレッド再統合手法ではキャッシュミス回数を表すパラメータを新たにノードの重みとして追加する．動的スレッド再統合手法では，このオーバヘッドを静的ステージのプロファイリングで計測し，グラフ分割アルゴリズムで使用する．

3.3 実行可能プログラムの拡充

3.1.2 項で述べた様に，Thread Tailor を用いて実行した場合，完了しないプログラムが存在する．これらのプログラムには，ビジーウェイトが含まれる．ビジーウェイトを含むプログラムをカーネルスレッドのみを用いて実行した場合，一定期間ごとに OS カーネルによってスレッドがコンテキストスイッチされるため，ビジーウェイトであっても実行が完了しないという状況は発生しない．しかし，Thread Tailor を使用した実行では，カーネルスレッドに加えユーザスレッドを使用しているため，ユーザスレッド間でコンテキストスイッチが発生せず，デッドロックが発生してしまう．デッドロックを回避するためには，ビジーウェイト中にユーザスレッドをコンテキストスイッチする必要がある．しかしながら，デッドロックが発生するループ文を事前に特定することは困難である．なぜならば，プログラムの意味論，変数に書き込まれる値，実行時の各スレッドの動作などを静的に解析し，デッドロックの発生箇所を特定する必要があるからである．

そこで本研究では，全てのループ文に対してユーザスレッドをコンテキストスイッチするコードを挿入することにより，デッドロックが発生する問題を解決することを考える．これにより，ビジーウェイトにおいても各ユーザスレッドが順にプロセッサコアに割り当てられるため，デッドロックを回避することができる．しかしこの解決策では，ビジーウェイトでないループ文に対してもコンテキストスイッチのコードが挿入されてしまう．一般的にユーザスレッドのコンテキストスイッチのコストは低いと言われているが，ループ内の処理を実行するごとに他のユーザスレッドにコンテキストスイッチすると，オーバヘッドが増大し，実行時間に多大な影響を与える可能性がある．

そこで，本提案手法では閾値を定め，それを超えるイテレーション回数に到達した場合にのみユーザスレッドをコンテキストスイッチするコードを，各ループ内に挿入することで上記の問題を解決する．これにより，閾値を超える回数ループ内の処理が実行された場合，ビジーウェイトと見なされ，ユーザスレッドがコンテキストスイッ

```
1  :
2  while( ){
3    // ループ回数が閾値の倍数になったら
4    if(trans_loop_num % trans_threshold == 0){
5      // ユーザスレッドコンテキストスイッチ
6      CotextSwitch( );
7    }
8    // ループ回数を記憶
9    trans_loop_num++;
10   :
11 }
12 :
```

図 13: コンテキストスイッチのためコード

チする．

図 13 は，プログラムコード内に存在するループ文に対してコンテキストスイッチのためコード（3-6 行目）を挿入した例を示している．Thread Tailor を用いて図 13 に示すプログラムコードを実行した場合，あるユーザスレッドが，while ループ（2 行目）に到達すると，本来のループ内の処理に加えて，ループする度にカウンタ変数（*trans_loop_num*）をインクリメントしていく（6 行目）．そして，ループ回数が閾値（*trans_threshold*）に到達すると if 文の条件が真となるため（3 行目），ユーザスレッドがコンテキストスイッチされる（4 行目）．

このプログラムコード（2-6 行目）をループ文に対して挿入することで，ビジーウェイトであっても定期的に他のユーザスレッドに処理が移るため，実行が完了しないという状況は発生しない．本研究では，上記の手法によりユーザスレッド間で発生するデッドロックを回避し，Thread Tailor で実行できるプログラムを拡充する．

4　提案手法の実装

3 章で提案した，動的スレッド再統合手法および実行可能プログラム拡充のためのコンテキストスイッチ挿入手法を実現するため，Thread Tailor を改良する．本章ではまず，提案手法の実行モデルを説明した後，それぞれの手法を実現するための各処理の改良およびトランスレータの実装について述べる．

4.1 実行モデル

3.2 節で述べた動的スレッド再統合手法を実現するために，本研究では，バリア同期間ごとの分割グラフを作成する．そして，それぞれ分割グラフにもとづいてバリア同期間ごとにユーザスレッドの組み合わせを変更する．また，3.3 節で述べた Thread Tailor を用いて実行するとデッドロックが発生してしまう問題に対して，実行プログラムのコードを変換することで対応する．**図 14**，**図 15** を用いて提案手法の実行時の動作を説明する．

図 14 は提案手法を用いて実行するプログラムを，図 15 は図 14 に示すプログラムを実行する際のスレッドの動作を表している．図 15 に示す分割グラフ X は，Parallel 関数の開始時（図 14・2 行目）から，バリア同期まで（図 14・19 行目）の区間のユーザスレッドの組み合わせを表しており，分割グラフ Y はバリア同期後（図 14・21 行目）から Parallel 関数の終了時（図 14・24 行目）までの区間のユーザスレッドの組み合わせを表している．また，図 14 に示すプログラム内にはデッドロックを防止するためのユーザスレッドコンテキストスイッチコードが挿入されている（図 14・6-11 行目）．

まず，図 14 に示すプログラムの実行が開始され（26 行目），スレッド生成のための関数（28 行目）が呼ばれると，提案手法は図 15 に示す分割グラフ X にもとづいてスレッドを統合する．この例の場合，User Thread A および User Thread B が統合され，そして User Thread C および User Thread D が統合される．スレッド統合の後，Kernel Thread 1 内の User Thread B および Kernel Thread 2 内の User Thread C が Parallel 関数の実行を開始する．

User Thread B は if 文（図 14・3 行目）の条件が真であるため，ループ内の処理（図 14・4-13 行目）を実行する．そしてループ内の処理実行回数が閾値に到達すると，挿入されたコンテキストスイッチ関数により，図 15 に示すように User Thread A にコンテキストスイッチし，User Thread A も同様にループ処理実行回数が閾値に到達すると User Thread B にコンテキストスイッチする．一方，User Thread C は if 文（図 14・3 行目）の条件が偽であるため，else 文（図 14・15 行目）に分岐し，バリア関数に到達した際に，User Thread D にコンテキストスイッチする．さらに実行が進み，User Thread B が while ループを抜けた後バリア関数に到達するため User Thread A にコンテキストスイッチする一方で，User Thread D もバリア関数に到達する．

全てのユーザスレッドがバリア関数に到達すると，ユーザスレッドは分割グラフ Y にもとづいて再統合される．この例の場合，User Thread A は Kernel Thread 2 へ，User Thread C は Kernel Thread 1 へマイグレートされる．最後に，新しいユーザスレッド

```
1  void Parallel(int Thread_ID){
2    // 分割グラフX ここから
3    if(Thread_ID == A || Thread_ID == B){
4      while( ){
5        // ループ回数が閾値の倍数になったら
6        if(trans_loop_num % trans_threshold == 0){
7          // ユーザスレッドコンテキストスイッチ
8          CotextSwitch( );
9        }
10       // ループ回数を記憶
11       trans_loop_num++;
12       :
13     }
14   }
15   else{
16     :
17   }
18   :
19   // 分割グラフX ここまで
20   Barrier( );                    // 全ユーザスレッドが同期
21   // 分割グラフY ここから
22   :
23   // 分割グラフY ここまで
24 }
25
26 int main(void){
27
28   CreateThread(Parallel);
29 }
```

図 14: 実行対象プログラム

の組み合わせでバリア同期後の実行が開始される．このプログラムの場合，再度バリア関数が出現することはないため，Parallel 関数が終了するまでこのスレッドの組み合わせで実行される．

以上の動作を実現するために，Thread Tailor の実行フロー内の処理を改良する．提案手法の実行フローを**図 16** に示す．提案手法では，まずトランスレータを用いて実行

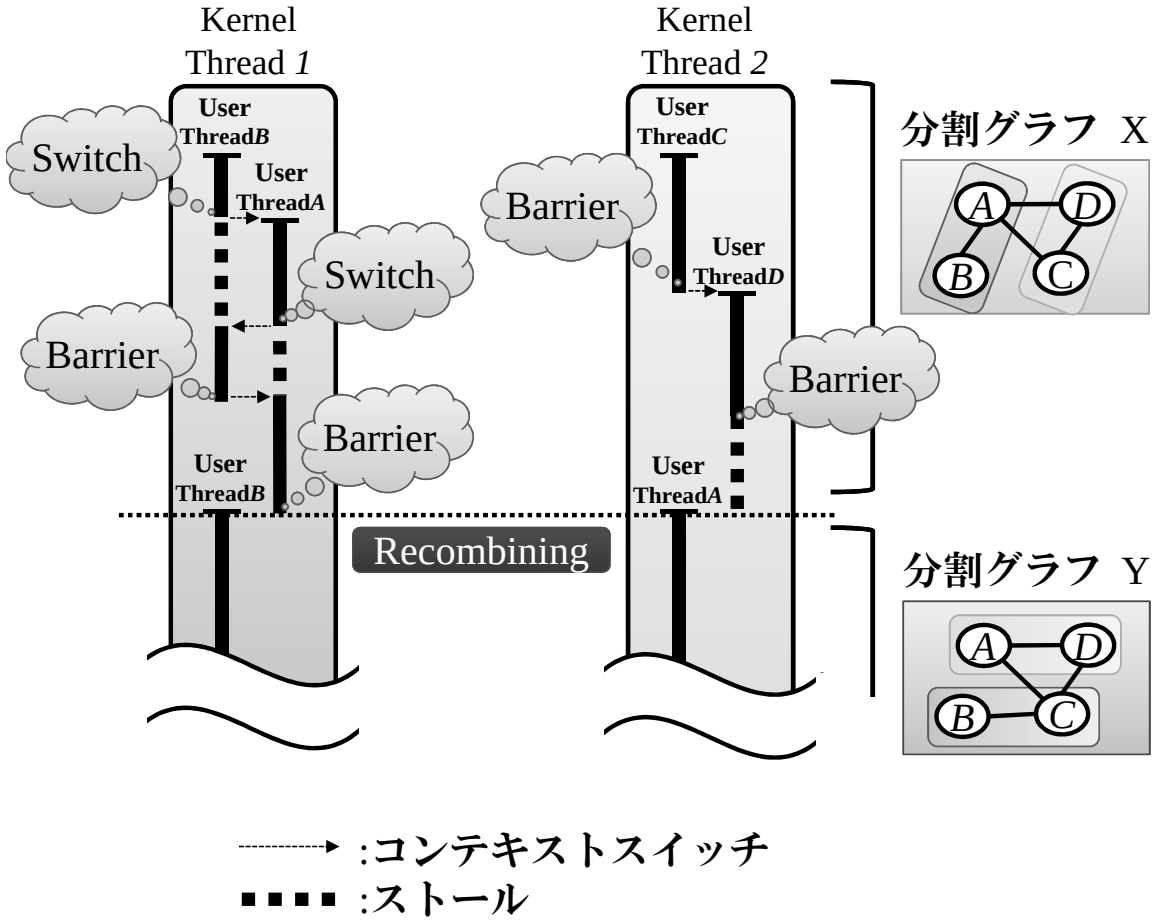


図 15: 提案手法を用いて実行した際のスレッドの動作

するプログラムに対して，ユーザスレッドコンテキストスイッチのためのコードを挿入し，デッドロックが発生しないコードに書き換える．そして，変換されたプログラムをバリア同期間ごとにプロファイリングし，各区間のコミュニケーショングラフを作成する．各区間のコミュニケーショングラフは，グラフ分割アルゴリズムを使用してそれぞれ分割され，その結果にもとづいてプログラム実行中にスレッドが再統合される．

以上の動作を実現するために，既存の Thread Tailor のプロファイリングおよびグラフ分割アルゴリズムを改良し，新たにトランスレータを追加実装した．加えて，バリア同期時にユーザスレッドの実行権限を委譲するプロセスを追加する．これを新たに追加する理由として，提案手法ではユーザスレッドを再統合するため，実行権限を持つユーザスレッドがカーネルスレッド内に存在しないという状況が発生しうるためで

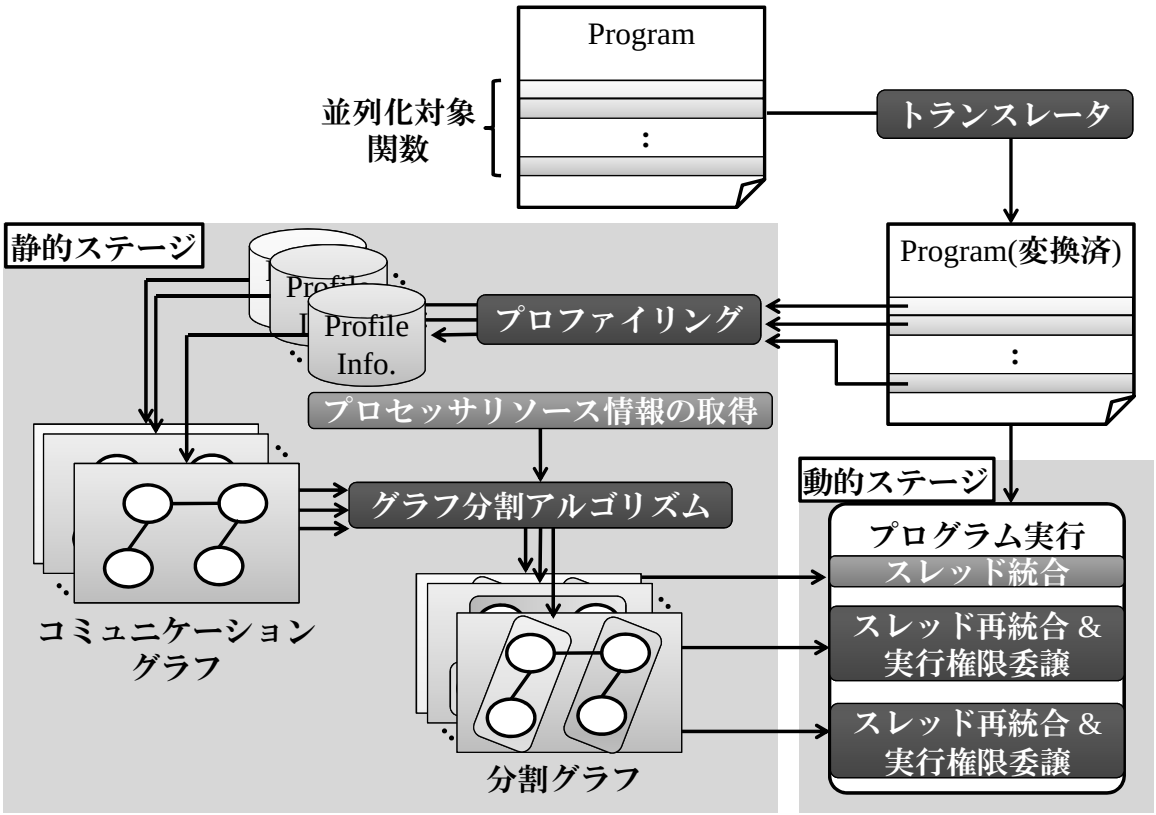


図 16: 提案手法の実行フロー

ある．これらの具体的な実装について，以下それぞれ 4.2 節，4.3 節で詳しく述べる．

4.2　スレッド再統合の実現方法

動的スレッド再統合手法を実現するために，既存の Thread Tailor を改良した．本節では，Thread Tailor の静的ステージおよび動的ステージ内の処理をどのように改良したのかについて述べる．

4.2.1　プロファイリングの改良

4.1 節で述べたように，動的スレッド再統合手法は各バリア同期間のプロファイリング情報を使用するため，静的ステージのプロファイリングを改良する必要がある．提案手法では，既存の Thread Tailor でも用いていたパラメータである $Cycle_i$，$WorkSet_i$，BW_i，$CommCost_{i,j}$ を収集し，それに加えてユーザスレッドマイグレーションに伴って発生する 1 次キャッシュミス回数（$MigrationCacheMiss_i$）を収集する．これらのパラメータをバリア同期間ごとに取得し，各区間のコミュニケーショングラフを作成

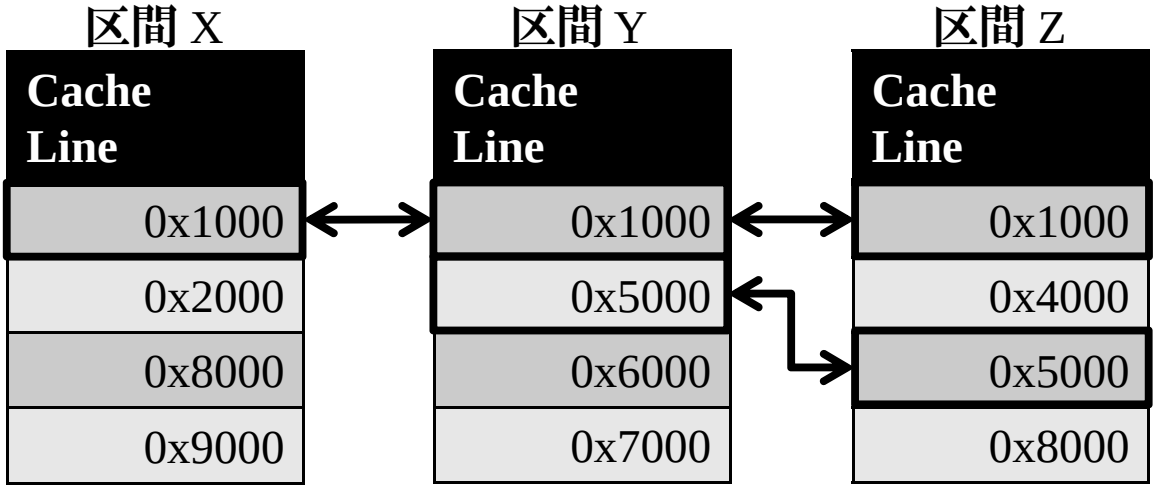


図 17: Thread i が各区間でアクセスしたキャッシュライン

する．

既存の Thread Tailor でも使用していたパラメータをバリア同期間ごとに取得するために，取得方法を次のように変更する．$Cycle_i$ はバリア同期によるストールサイクル数を除いたサイクル数であるため，取得方法に変更はない．$WorkSet_i$ および $CommCost_{i,j}$ を取得するために，バリア同期間ごとに図 3，図 4 で示した表を作成する．そして，各区間の $WorkSet_i$ および $CommCost_{i,j}$ を取得する．また，各区間の BW_i を計測するために，OProfile から値をスヌープする．提案手法では OProfile から値をスヌープするために，プログラム内からシェルコマンドを実行するための system 関数を用いる．system 関数を用いてベンチマークプログラム内のバリア関数実行前に OProfile の中断スクリプトを呼び出し，以前のバリア関数から当該バリア関数までの通信量を取得する．そして，バリア関数実行後に再び system 関数を用いて再開スクリプトを呼び出し，次の区間の通信量の計測を開始する．そして得られた通信量とその区間の $Cycle_i$ を用いて，BW_i を算出する．

これに加えて，提案手法ではバリア同期間ごとの $MigrationCacheMiss_i$ を計測する．このパラメータを計測するために，各スレッドがアクセスしたキャッシュラインの情報を用いて，隣り合う区間ごとに共通に使用するキャッシュラインの個数を調査する．**図 17** は 2 つのバリア同期が存在するプログラムをバリア同期ごとにプロファイリングした際に，各区間で，ある Thread i がアクセスしたアドレスを示している．また，区間 X，区間 Y，区間 Z はバリア同期で区切られたそれぞれ連続した区間を表す．まず，Thread i は，区間 X および区間 Y でアドレス `0x1000` にアクセスしていることが

図から分かる．そのため，Thread i が区間 Y を区間 X とは異なるプロセッサコアで実行する場合，区間 Y で 1 度だけ 1 次キャッシュミスが発生すると提案手法は予測する．つまり，Thread i における区間 Y の $MigrationCacheMiss_1$ は 1 となる．次に，区間 Y と区間 Z を見てみると，Thread i はアドレス `0x1000` および `0x5000` にアクセスしていることが分かる． そのため，区間 Y，Z 間のバリア同期で Thread i が他のプロセッサコアにマイグレートされた場合，区間 Z で 2 度キャッシュミスが発生すると提案手法は予測し，区間 Z の $MigrationCacheMiss_i$ は 2 となる．

最後に区間 X および区間 Z では，両区間で Thread i はアドレス `0x8000` にアクセスしていることが分かる．このため，Thread i を実行するプロセッサコアが区間 X と区間 Z で異なる場合，区間 Z で Thread i がキャッシュミスする可能性がある．しかし，もしあるスレッドが区間 Y で `0x8000` にアクセスしており，区間 Y でそのスレッドを実行したプロセッサコアと区間 Z で Thread i を実行するプロセッサコアが同じであった場合，Thread i がマイグレートされたとしてもそれに伴ってキャッシュミスは発生しない．つまり，区間を跨いでマイグレーションペナルティを考える場合，各スレッドがどのプロセッサコア上でどのキャッシュラインにアクセスしたのか全て考慮する必要があるため，$MigrationCacheMiss_i$ の取得方法が複雑になる．そのため提案手法では，`0x8000` の様な隣り合わない区間の共通のキャッシュラインは考慮しないこととする．

上記で述べた方法により収集したキャッシュミス回数をグラフ分割アルゴリズムで使用するために，このキャッシュミスペナルティを以下の式を用いてサイクル換算する．

$$CacheMissPenalty_i = MigrationCacheMiss_i \times L2AccessLatency \qquad (11)$$

式 (11) の $CacheMissPenalty_i$ はマイグレーションに伴って発生したキャッシュミスペナルティのサイクル数を表し，$L2AccessLatency$ はプロセッサの 2 次キャッシュへのアクセスレイテンシを表している．提案手法では 1 次キャッシュミスしたデータは，少なくとも 2 次キャッシュでヒットするであろうと予想し，この式を用いる．以上のようにして，提案手法はバリア同期間ごとにプロファイリングすることでバリア同期間ごとのパラメータの計測およびコミュニケーショングラフの作成を行う．

4.2.2 グラフ分割アルゴリズムの改良

4.1 節で述べたように，動的スレッド再統合手法では，バリア同期間ごとのコミュニケーショングラフに対してグラフ分割アルゴリズムを適用する．この際，既存の Thread Tailor が使用するパラメータに加え，前項で述べたマイグレーションに伴って発生す

```
1  :
2  // コミュニケーショングラフを初期分割
3  Initial_Partition( );
4
5  // どのノード同士を交換してもＧａｉｎが得られないならループを抜ける
6  while( ){
7
8    // 各グループからノードを一つずつ選択
9    Select_Two_Nodes( );
10
11   // 交換するべきか調査
12   Estimate_Gain_by_Exchanging_Nodes( );
13
14   // 交換することでＧａｉｎが得られ、制約を満たすなら
15   if(Gain > 0 && fulfill evaluation formula){
16
17     // グループ間でノードを交換
18     Exchange_Nodes( );
19
20     // 各ノードが持つ実行サイクル数にキャッシュミスペナルティを加算
21     Cycle_i = Cycle_i + CacheMissCost_i;
22     Cycle_j = Cycle_j + CacheMissCost_j;
23   }
24 }
25 :
```

図 18: 改良したグラフ分割アルゴリズムの擬似コード

るキャッシュミスを考慮して，統合するユーザスレッドを決定する．そのため，既存の Thread Tailor で使用されていたグラフ分割アルゴリズムを改良する必要がある．

改良したグラフ分割アルゴリズムの擬似コード**図 18** に示す．まず，提案手法で使用するアルゴリズムは，与えられたコミュニケーショングラフを，1 つ前の区間の実行に用いたノードグループと同じノードの組み合わせに分割する（3 行目）．その後，各ノードグループからノードを一つずつ選択し（9 行目），そのノード同士を交換するべきか調査する（12 行目）．ノードを交換することにより正の *Gain* が得られ，かつ各グループが各制約を満たすのであれば（15 行目），グループ間でノードを交換し（18 行目），交換したノードの実行サイクル数にマイグレーションに伴って発生するキャッ

シュミスオーバヘッドを加算する（21-22 行目）．これらの処理は，各グループ間のどのノード同士を交換しても，正の $Gain$ が得られなくなった時点で終了する（6-24 行目）．

また，動的スレッド再統合手法に適したグラフ分割を実現するため，式（8）で示した，グラフ分割アルゴリズムで使用するメモリバンド幅に関する制約も併せて変更する．Thread Tailor を用いた実行では，同じカーネルスレッドに所属するユーザスレッド間でコンテキストスイッチしながら実行が進む．言い換えると，Thread Tailor では，同じカーネルスレッドに所属するユーザスレッドは同時に実行されることはない．そのため，グループ内の複数のユーザスレッドが同時にメモリへアクセスすることもない．つまり，同じグループ内の複数のユーザスレッドが同時にメモリへアクセスすることを前提とした既存の Thread Tailor のメモリバンド幅に関する制約は厳しすぎる．加えてこの制約は，提案手法にとってはより厳しいものとなるという問題がある．これは，提案手法は Thread Tailor と異なり，短い区間で取得したプロファイル結果を使用しているため，ある処理区間を実行した際にメモリバンド幅使用量が増加した場合，制約を満たさない可能性が Thread Tailor よりも増加するためである．そのため，この不適切な制約の不利益を被りやすい．

この様な理由から，提案手法では以下の制約に変更した．

$$BW_i = \max_{i \in Group_k} BW_i \leq MemBW \tag{12}$$

提案手法では，グループ内の最大のメモリバンド幅を持つユーザスレッドがプロセッサのメモリバンド幅容量を超えない場合に限りノードの交換を許可する．このように，メモリバンド幅の制約を緩和することで，通常なら適切にノードが交換されないようなプログラムに対しても，ノードが交換されるようになり，適切なスレッド統合が実現できる．

4.2.3 ユーザスレッドの実行権限受け渡し

4.1 節で述べたように，動的スレッド再統合手法では，全てのユーザスレッドがバリア関数に到達した際，ユーザスレッドは再統合される．そのため提案手法を用いた実行では，バリア同期後に実行権限を持つユーザスレッドがカーネルスレッド内に存在しないという状況が発生する可能性がある．それは，カーネルスレッド内の実行権限を持つユーザスレッドが，スレッド再統合によって他のカーネルスレッドにマイグレートされてしまう可能性があるからである．これによって，実行が進まないユーザスレッドが発生してしまう．

図 19(a) に示す例では，Kernel Thread 1 内に User Thread A および User Thread B

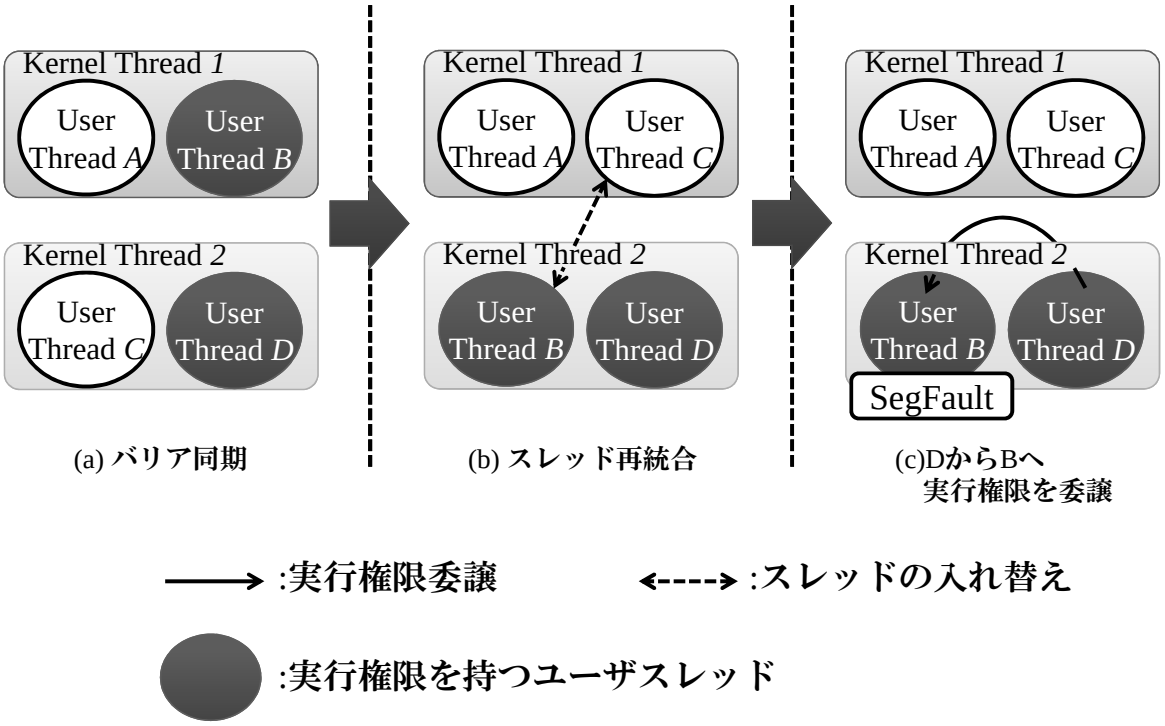


図 19: スレッド再統合時に問題が発生する様子

が，Kernel Thread 2 内に User Thread C および User Thread D が存在している．また，黒塗りのユーザスレッドは実行権限を持つユーザスレッドを表している．まず，全てのユーザスレッドがバリア関数に到達し（図 19(a)），User Thread B および User Thread C が他方のカーネルスレッドにマイグレートされたとする（図 19(b)）．この場合，Kernel Thread 1 に実行権限を持つユーザスレッドが存在しなくなる．一方，Kernel Thread 2 には実行権限を持つユーザスレッドが複数存在してしまう．このため，Kernel Thread 1 内に存在する User Thread A および User Thread C の実行がこれ以降進まなくなり，プログラムの実行が終了しない．これを解決するためには，Kernel Thread 2 に存在する実行権限のどちらかを Kernel Thread 1 内のユーザスレッドのどちらかに委譲する必要がある．ここでは，User Thread B の実行権限を User Thread A に委譲しようとしたとする．しかし，この委譲より先に，もし User Thread D の実行権限が User Thread B に委譲された場合（図 19(c)），Kernel Thread 2 でセグメンテーションフォールトが発生し，実行が停止する．

この問題を解決するために，提案手法は実行権限退避スレッドと呼ぶユーザスレッドをカーネルスレッドごとに作成する．この実行権限退避スレッドは，プログラムの

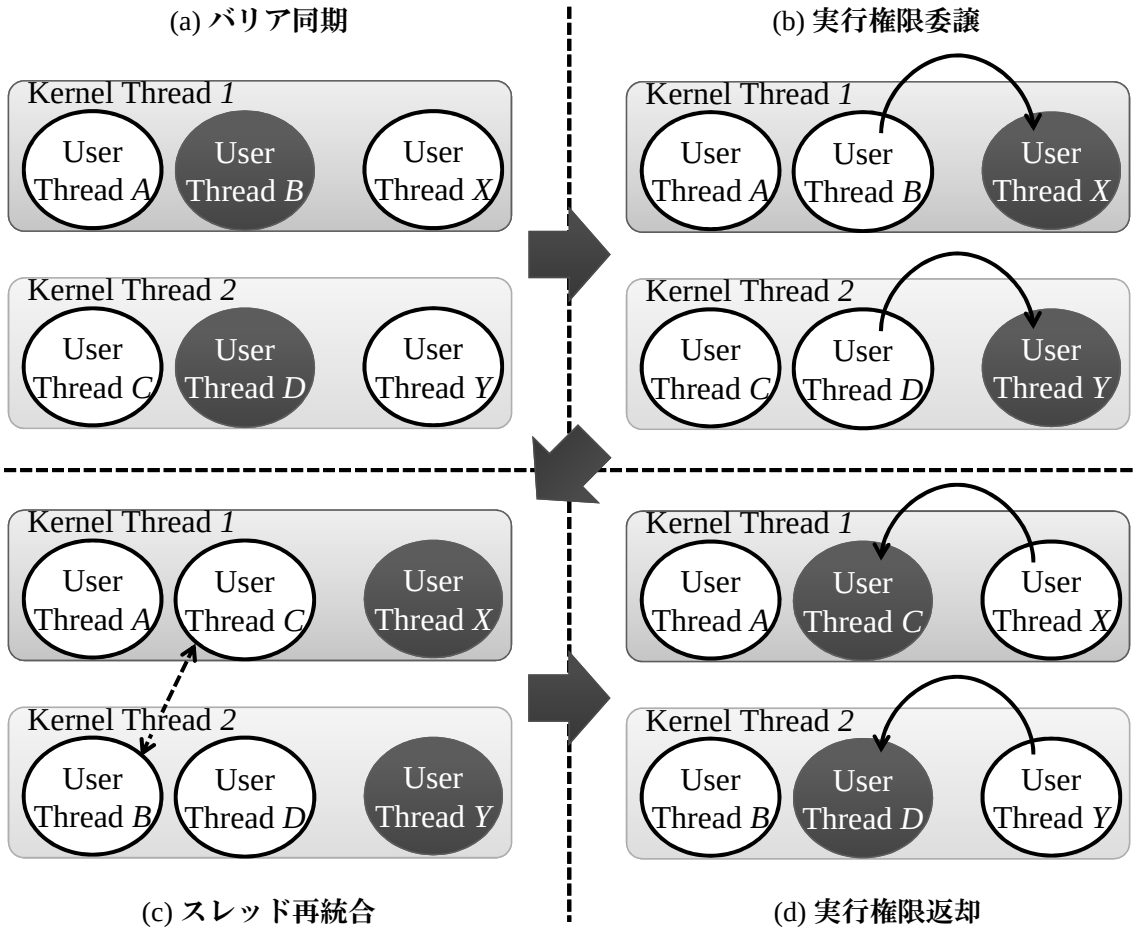


図 20: 実行権限退避スレッドを用いた実行権限の委譲手順

実行に関与しないユーザスレッドであり，提案手法では，バリア同期時にこのスレッドを介してユーザスレッド間で実行権限を受け渡す．**図 20**(a) では，図 19(a) のスレッド構成に加え，実行権限退避スレッドである User Thread X および User Thread Y が各カーネルスレッドに作成されている．まず，カーネルスレッド内の全てのユーザスレッドがバリアに到達した際（図 20(a)），動的スレッド再統合手法では，User Thread B および User Thread D が持つ実行権限が各カーネルスレッド内の実行権限退避スレッド User Thread X および User Thread Y に委譲される（図 20(b)）．そして委譲された後，分割グラフを用いてスレッドが再統合される（図 20(c)）．この例では，User Thread B および User Thread C が交換される形で他のカーネルスレッドにマイグレートされたとする．そしてその後，User Thread X および User Thread Y の実行権限を各カーネルスレッド内のユーザスレッドに戻す（図 20(d)）．この例では，User Thread

C および User Thread D に実行権限を戻し，バリア同期後の処理が開始されている．このように実行権限を受け渡すことで，カーネルスレッド内に存在する実行権限を持つユーザスレッドは常に 1 つとなり，実行の停止およびセグメンテーションフォールトの発生を回避できる．

ここで，プログラム実行に関与しない実行権限退避スレッドが作成され，そしてこのスレッドにプロセッサのリソースが割り当てられてしまうことで，実行速度が低下する可能性がある．しかしながら，このスレッドは各カーネルスレッドに 1 つずつしか作成されず，またこのスレッドによる処理はほとんど存在しないため，1 つのプロセッサコア当たりの使用リソース量はほとんどないと考えられる．以上のことから，実行権限退避スレッドを作成および使用することによる，プログラムの実行時間に与える影響はほとんどないと考えられる．

4.3 実行可能プログラム拡充のためのコード変換

Thread Tailor を適用可能なプログラムを増やすため，プログラムにユーザスレッドコンテキストスイッチのためのコードを挿入する機能を実装する．本節では，プログラムの変換方針および，それを実現するためのトランスレータの実装について述べる．

4.3.1 コード変換方針

ビジーウェイトを含むプログラムを Thread Tailor を用いて実行するために，提案手法では全てのループ文に対して図 13 に示したコードを挿入する．さらに，このコード内で使用されている 2 つの変数（*trans_threshold*，*trans_loop_num*）は実行対象プログラム内で新たに定義される必要がある．閾値を表す *trans_threshold* 変数は，全てのスレッドで使用されるため，大域変数として定義される．また，*trans_loop_num* 変数を用いて各スレッドがループごとに独立してループ回数をカウントするため，この変数は，ループの個数分の要素を持つ配列として関数内に定義される．

以上で述べたプログラム改変を実現するために，本研究ではトランスレータを実装する．**図 21** にトランスレータのプログラム変換フローを示す．トランスレータは，プログラム解析とコンテキストスイッチコードの挿入の 2 つの処理から構成されている．まず，トランスレータに対して実行対象プログラムが渡されると，プログラムを解析する．この解析では，2 つのプログラム情報を取得する．1 つは，実行対象プログラム内で使用されている変数名である．これは，トランスレータによって挿入される変数の変数名がプログラム内で既に使用されていることを避けるため，前もってこれを検出する必要があるからである．もう 1 つは，各関数内に存在するループ文の個数であ

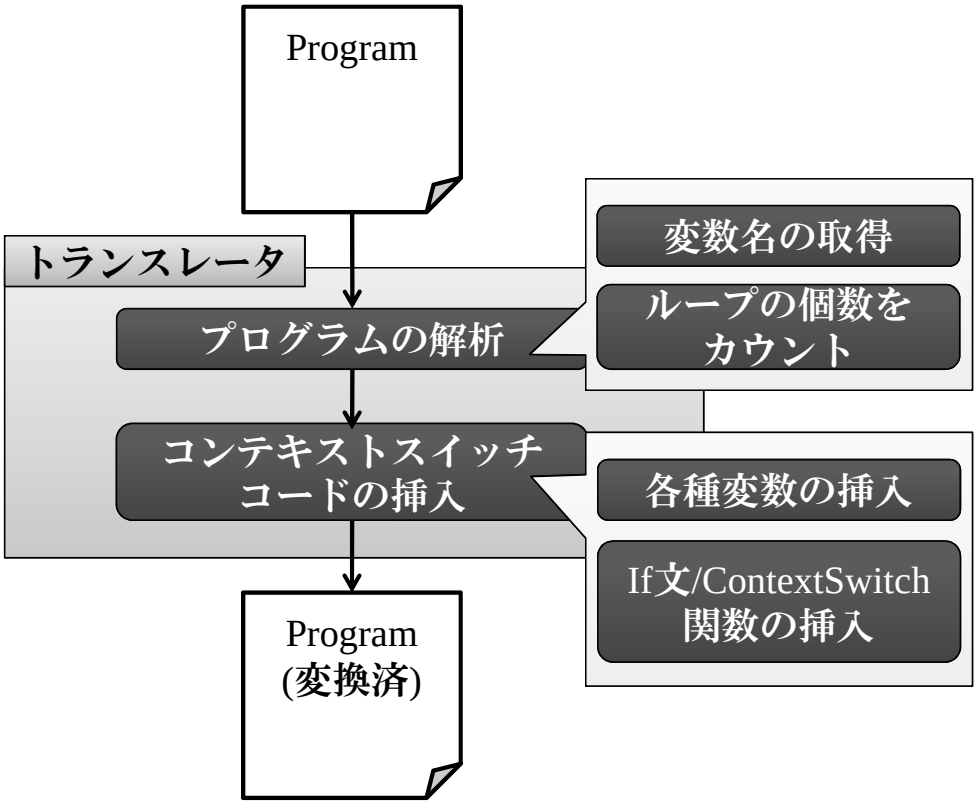


図 21: コードの変換手順

る．これは，ループ回数を記憶するための変数をループの個数分だけ定義する必要があるためである．

プログラムの解析が終了すると，コンテキストスイッチコードを挿入する処理に移る．この処理では，トランスレータは再度プログラムを読み出し，解析により得られたプログラム情報にもとづいて，各種変数の定義文の挿入およびループ文に対してコンテキストスイッチのためのコードの挿入を行う．そして全ての変換が終わると,トランスレータは変換結果をファイルに出力する．次項以降，この 2 ステップについてより詳細に説明する．

4.3.2 プログラム解析

トランスレータはまず実行対象プログラムを解析する．ここでは**図 22** に示すコードを参照しながら，プログラム解析の詳細について述べる．なお，図中の Parallel 関数が並列化対象関数の関数であり，この関数は内部に while ループおよび for ループを 1 つずつ持っているとする．

まず，トランスレータはプログラムを読み出し，プログラム内の大域変数名を取得

```
1  int A=100, B;
2
3
4
5  void Parallel(void){
6    int i;
7
8
9
10
11   while( ){
12
13
14
15
16
17
18
19
20
21     :
22   }
23
24
25   for(;;){
26
27
28
29
30
31
32
33
34     :
35   }
36
37     :
38
39 }
```

図 22: 変換前のコード

```
1  int A=100, B;
2  // 閾値を定義
3  int trans_threshold = xx;
4
5  void Parallel(void){
6    int i;
7    // ループ回数を記憶するための変数を定義
8    int trans_loop_num[2] = {0,0};
9
10
11   while( ){
12     // ループ回数が閾値の倍数になったら
13     if(trans_loop_num[0] % trans_threshold
14        == 0){
15       // コンテキストスイッチ
16       CotextSwitch( );
17     }
18     // ループ回数を記憶
19     trans_loop_num[0]++;
20
21     :
22   }
23
24
25     for(;;){
26       if(trans_loop_num[1] % trans_threshold
27          == 0){
28
29         CotextSwitch( );
30       }
31
32       trans_loop_num[1]++;
33
34       :
35     }
36
37     :
38
39 }
```

図 23: 変換後のコード

する．このプログラムに存在する大域変数は A，B（1 行目）であるので，この変数名をトランスレータは記憶する．次に，関数名と関数内で定義されているローカル変数名を取得する．そのため，$Parallel$（5 行目），i（6 行目）がそれぞれ記憶される．また，この時得られた変数名と $trans\_threshold$ 変数および $trans\_loop\_num$ 変数の名前を比較し，挿入する変数がプログラム内で既に使用されていないか確認する．このコードには，挿入する変数と同名の変数は存在しないが，同名の変数を検出した場合には，その変数名を記憶する．

次に，関数内に存在するループ文の個数を調査する．調査対象のループは while 文，for 文，do 文である．トランスレータは，while，for の後に丸括弧が続く文を，また，do の後に波括弧が続く文をループ文の開始と捉え，それらの個数をカウントする．このコードの場合，トランスレータは Parallel 関数内で while 文（11 行目）および for 文（25 行目）を 1 つずつ検出する．そのため，Parallel 関数内に存在するループは 2 つと判断される．このようにプログラムを解析する．

4.3.3　コンテキストスイッチコードの挿入

前項で述べたプログラム解析をした後，トランスレータは実行対象プログラムに対してコンテキストスイッチコードを挿入する．図 22 に示すコードに対して各種コードを挿入する手順を説明する．

まず，トランスレータは実行対象プログラムを再度読み出し，コードの先頭に閾値を表す $trans\_threshold$ 変数の定義文を挿入する（**図 23**・3 行目）．この変数の定義文を挿入した後，トランスレータは $trans\_loop\_num$ 変数の定義文およびコンテキストスイッチのためのコードを挿入する処理に移る．まずトランスレータは，Parallel 関数内に $trans\_loop\_num$ 変数の定義文を挿入するために，プログラム解析の結果から，Parallel 関数内のループの個数を取得する．Parallel 関数内にはループ文が 2 つ存在すると判明しているので，$int\ trans\_loop\_num[2] = \{0, 0\}$ を挿入する（図 23・8 行目）．また，もしプログラム解析の際に，挿入する変数と同名の変数がプログラム内で既に使用されていることが確認された場合，挿入する変数の変数名の末尾に乱数を付加する．

次に，Parallel 関数内を 1 行ずつ読み出し，while 文（図 22・11 行目）検出すると，このループ文の先頭にコンテキストスイッチコードを挿入する（図 23・13-19 行目）．また，終了条件が明記されていない for 文（図 22・25 行目）に対しても同様のコードを挿入する（図 23・26-32 行目）．

以後，同様の手順で各関数に対して $trans\_loop\_num$ 変数の定義およびコンテキストスイッチのためのコードを挿入する．この様な手順で実行プログラムを変換し，Thread

表 2: 評価環境

OS	Fedora 15
CPU	Intel Core2 Quad Q9550
動作周波数	2.83 GHz
コア数	4
1 次命令キャッシュ	4 × 32 KB
1 次データキャッシュ	4 × 32 KB
2 次キャッシュ	2 × 6 MB (各キャッシュは 2 コア間で共有)
メモリ	3 GB
コンパイラ	llvm-gcc 4.2.1
最適化オプション	-O3

表 3: ベンチマークプログラムとその入力

SPLASH-2	
barnes	32768 BODIES
water-nsquared	4096 MOL, 6 STEP
cholesky	tk14.0
radiosity	-batch -room
PARSEC	
fluidanimate	simlarge
blackscholes	simlarge
swaptions	simlarge

Tailor で実行可能なプログラムを得る.

5 評価

本章では，提案手法の有効性を示すためにベンチマークプログラムを用いて評価し，その結果について考察する．また，トランスレータにより変換したプログラムが正しく実行されているかどうかの検証も併せて行う.

5.1 評価環境

提案手法を Thread Tailor に実装し，**表 2** に示す環境で評価した．ベンチマークプログラムには，SPLASH-2 ベンチマークプログラムに含まれる barnes，water-nsquared，radiosity，cholesky に加えて，PARSEC ベンチマークプログラム [12] に含まれる fluidanimate，blackscholes，swapthions を使用した．またベンチマークプログラムへの入力は，一般的にプロファイリング時と実行時で必ずしも同一とは限らず，入力が異なる際は，各スレッドの処理量が変化するため，実行速度が低下する恐れがある．しかし本研究では，提案手法の限界性能を確認するため，プロファイリング時，実行時ともに同様とし，**表 3** で示す入力を使用した．

ベンチマークプログラムを実行する際に使用したカーネルスレッドはプロセッサコア数と同数の 4 本，そしてユーザスレッドは 16 本使用した．Thread Tailor および提案手法では，負荷分散の観点から各カーネルスレッド内に均等にユーザスレッドが分配されるため，各カーネルスレッド内で 4 本のユーザスレッドが動作する．また，2 次キャッシュへのアクセスレイテンシ，および 2 次キャッシュラインサイズをシステムレベルで調査することは困難であるため，本実験では一般的なプロセッサの環境に準じてそれぞれ 50 サイクル，64 バイトと仮定した．コンパイラには llvm-gcc(Ver 4.2.1) を使用し，最適化オプションとして-O3 を使用した．さらに，コンテキストスイッチ挿入手法で使用するループ回数の閾値は 200 とした．

なお，本実験は実環境を用いて評価しているため，実行するたびに実行時間が変動する．そのため，各評価対象につき試行を 20 回繰り返し，得られた実行時間の平均値を比較する．

5.2 評価結果

図 24 には各ベンチマークプログラムと提案手法との組み合わせによる結果がそれぞれ 5 本のグラフで表されている．5 本のグラフはそれぞれ左から順に

（**N**） 通常実行 (ベースライン)
（**T**） Thread Tailor
（**R**） 動的スレッド再統合手法
（**C**） コンテキストスイッチコード挿入手法
（**H**） 手法（R）と手法（C）を組み合わせた手法

の実行時間比を表しており，通常実行（N）の実行時間を 1 として正規化している．実行速度に関して見ると，その特徴は主に 3 種類分けられる．手法（R）を使用しても

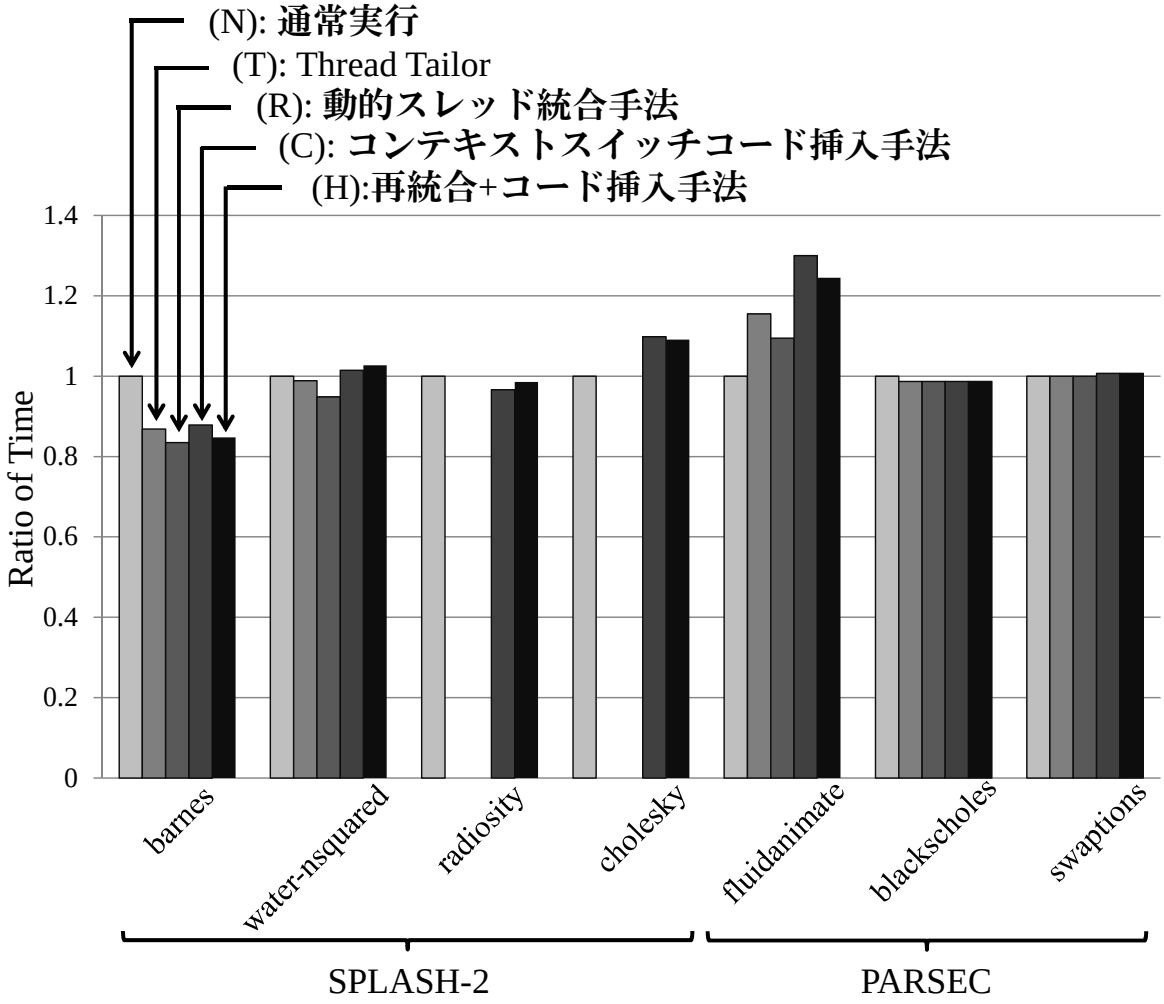


図 24: 評価結果

（T）と比べ実行速度に変化が無い blackscholes および swaptions，手法（R）で高速化した barnes，water-nsquared および fluidanimate，手法（C）により実行が可能になった radiosity，cholesky である．

まず，blacksholes および swaptions は**表 4** に示すように，バリア同期がプログラム実行時に出現しない．そのため，手法（T）および手法（R）で動作に違いが生じず，実行時間に差がなかった．

次に，手法（R）により実行速度が向上した barnes，water-nsquared では表 4 に示すようにそれぞれ 18 回および 38 回のバリア同期が実行され，バリア同期間ごとにスレッドが再統合されたことが高速化に繋がった．詳しく見ていくと，barnes は water-nsquared に比べ実行速度向上率が 3.0%とやや低い．これは，barnes にはホットスポッ

表 4: バリア同期回数

SPLASH-2	
barnes	18 回
water-nsquared	38 回
cholesky	4 回
radiosity	18 回
PARSEC	
fluidanimate	41 回
blackscholes	0 回
swaptions	0 回

表 5: water-nsquared のバリア同期間ごとの高速化率 (一部)

	手法（T）	手法（R）	高速化率
...	...	...	...
Phase 14	4,975	4,897	1.5%
Phase 15	2,226,670	2,224,019	0.1%
Phase 16	1,007,777,785	998,783,181	0.8%
Phase 17	586,529	576,464	1.7%
Phase 18	464,651	461,106	0.7%
Phase 19	1,073,697,468	1,072,429,653	0.1%
Phase 20	77,396	77,326	0.1%
...	...	...	...

トが存在し，その区間だけから判断した適切なスレッド統合の組み合わせと並列化対象関数全体から判断した適切なスレッド統合の組み合わせとが同じであったことが原因と考えられる．一方，water-nsquared は手法（T）に比べ 4.1%実行速度が向上した．このプログラムには複数のホットスポットが存在し，それぞれのホットスポットで，各スレッドの処理量が異なるため，適切なスレッドの組み合わせが異なる．そのため，手法（R）で高速化したと考えられる．詳細を**表 5** に示す．表 5 内の Phase は，バリア同期間ごとで，最後にバリア同期に到達したカーネルスレッドの実行サイクル数を表し，高速化率は手法（R）における手法（T）からの速度向上率を表している．表の高速化

率を見ると，Phase 14 から Phase 20 では手法（R）は高速化していることが分かる．また，ホットスポットである Phase 16，Phase 19 は異なる処理区間であり，適切なスレッドの組み合わせも異なる．手法（R）では，それぞれの処理区間に対して適切にスレッドを統合できたため，手法（T）に比べそれぞれ 0.8%，0.1%と実行速度が向上したと考えられる．さらに，このプログラムではこれらの処理区間が複数回実行されるため，全体で 4.1%の高速化に繋がったと考えられる．

また fluidanimate は，メモリバンド幅を多量に使用するプログラムであり，手法（T）ではグラフ分割アルゴリズムのメモリバンド幅の制約が厳しいため，グループ間でノードの交換が一切行われなかった．そのため，適切なユーザスレッドの組み合わせでプログラムを実行することができなかった．しかし手法（R）では，メモリバンド幅の制約を緩和したことにより，適切にコミュニケーショングラフが分割されたため，手法（T）に比べ 6.0%実行速度が高速化した．

最後に，radiosity および cholesky は手法（T）および手法（R）で評価結果を得ることができなかった．それは，3.1.2 項で述べたように，これらのプログラムにはビジーウェイトが存在するためである．実行速度について見ると，radiosity では通常実行（N）に比べ手法（C）の実行速度が向上していることが分かる．radiosity の実行が高速化した理由として，Thread Tailor を利用したことによる各プロセッサコアの処理量の均衡化の他に，挿入されたコンテキストスイッチコードにより，通常実行（N）に比べビジーウェイトの実行時間が削減されたことが考えられる．

一方，cholesky は手法（C）で実行時間が悪化しているのが分かる．cholesky は各スレッドが並列化対象関数の実行に要したサイクル数がほぼ等しいため，Thread Tailor を用いることによる実行時間の増減はほぼ無いと考えられる．そのため，実行速度の低下はコンテキストスイッチコード挿入によるコンテキストスイッチ回数の増加に起因するものであると考えられる．上述した考察の正否を確認するため，cholesky を Thread Tailor を用いて実行した際，デッドロックが発生する箇所にのみコンテキストスイッチコードを挿入し，再度評価を行った．その結果，通常実行（N）に比べて約 18%の高速化が達成できた．つまり，cholesky では，ビジーウェイトの実行時間の削減以上にユーザスレッドコンテキストスイッチによるオーバヘッドが大きいために手法（C）では実行速度が低下したと考えられる．以上のことから，ユーザスレッドコンテキストスイッチによるオーバヘッドが実行時間に多大な影響を与えているプログラムも存在することが分かった．そのため，コンテキストスイッチコードの挿入条件を厳格化することで，さらに cholesky，radiosity の高速化が図れると考えられる．

表 6: 実行時間の削減率

		SPLASH-2	PARSEC	all
動的スレッド再統合手法	平均	3.5%	2.8%	2.9%
	最大	4.1%	6.0%	6.0%
コンテキストスイッチコード挿入手法	平均	-2.4%	-5.0%	-4.0%
	最大	-2.6%	-0.1%	-0.1%
再統合+コード挿入手法	平均	0%	-3.2%	-2.2%
	最大	2.0%	-0.1%	2.0%

提案手法を用いた実行における Thread Tailor からの実行時間削減率を**表 6** に示す．動的スレッド再統合手法を用いることにより，Thread Tailor に比べ平均 2.9%，最大 6.0%の実行速度の向上が見られた．また，バリア同期が存在するプログラムに限定すると，Thread Tailor に比べて平均 4.4%，最大 6.0%の実行速度向上が得られた．一方コンテキストスイッチ挿入手法では，Thread Tailor を用いて実行できるプログラムが拡充された一方で，多くのプログラムで実行速度の低下が見られた．また両手法を組み合わせた手法では，ユーザスレッドコンテキストスイッチの回数増加に伴う実行時間の悪化を多くのプログラムで抑制することができた．

6 関連研究

プロセッサの環境に合わせてスレッド数を決定する研究やスレッドをスケジューリングする研究は OS およびハードウェア分野で広く行われている．例えば，Lina ら [13] や Chen ら [14] はマルチコアプロセッサ環境で，Mohan ら [15] や Jonathan ら [16] はメニーコアプロセッサ環境でスレッドを適切にスケジューリングする研究を行った．これらの研究では主に，各スレッドの IPC を動的にサンプリングし，その結果にもとづいてスレッドを実行するプロセッサコアを決定する手法やスレッド間で共通してアクセスされるデータを物理的に近距離に配置することで，通信ペナルティを削減する手法が提案されている．

しかし，適切なスレッド数の決定およびスレッドスケジューリングのための十分な情報を OS およびハードウェアレベルで取得することは困難であるという問題があり，適切なスレッド数を選択できない場合も多く存在する．そこで，より適切にスレッド数を決定するための手法として，Jung らの手法 [17]，Suleman らの手法 [18] がこれま

で提案されてきた．

Jung らの手法では OpenMP を用い，SMT 環境でプログラムを実行した際の適切なスレッド数を自動的に決定する．これを実現するために，ループ実行時の実行時間およびキャッシュミス回数を取得するコードを実行プログラムに対して挿入し，そのプログラムを用いて実行する．そして，その際に得られた実行時間およびキャッシュミス回数からループごとに作成すべきスレッド数を決定する．Suleman らの手法も同様に，並列化対象ループを実行する際の適切なスレッド数を自動的に決定する．Suleman らは，これを実現するために 2 つの手法を提案している．1 つ目は，同期に着目した手法である．並列化対象ループ内を実行した際の時間と，そのループ内に存在するクリティカルセクションを実行した際の時間をそれぞれ計測し，その結果からそのループを並列化することによって削減できる実行時間を予測する．そしてその予測結果を用いて，最も高速に実行可能なスレッド数を選択する．2 つ目は，メモリバンド幅に着目した手法である．この手法ではまず，単一スレッドで並列化対象ループを実行した際に使用したメモリバンド幅を計測する．そして，その結果とプロセッサのメモリバンド幅容量から並列化した際のメモリバスの競合による実行速度の低下を予測し，その予測に従って実行スレッドを作成する．

しかしながらこれら 2 つの手法は，Thread Tailor と同様にプロセッサ環境に適したスレッド数でプログラムを実行できる一方で，並列化対象はループに限定されており，並列化の恩恵を受けにくいという問題がある．

7　おわりに

本研究では Thread Tailor に対し，ユーザスレッドをバリア同期時に再統合することによって既存手法に比べ高速にプログラムを実行できる手法を提案した．また，Thread Tailor を用いて実行した際に発生するデッドロックによりプログラム実行が進まなくなる問題を解決するために，トランスレータを実装した．これにより，Thread Tailor で実行可能なプログラムを拡充した．

これらの提案手法を実装し，SPLASH-2 ベンチマークプログラムおよび PERSEC ベンチマークプログラムを用いて実行時間を評価した．その結果，動的スレッド再統合手法では，Thread Tailor に比べ各プロセッサの負荷を均衡化し，かつスレッド間の通信を削減できたことにより，既存の Thread Tailor に比べてバリア同期が存在するプログラムでは，平均 4.4%，最大 6.0%の実行時間が削減されたことを確認した．なお，バリア同期が存在しないプログラムに関しては，実行速度に変化は無かった．

また，トランスレータを使用してプログラムを改変することによりデッドロックの発生を防ぎ，既存の Thread Tailor では実行不可能であった radiosity，cholesky の実行が可能になったことを確認した．さらに，ユーザスレッドコンテキストスイッチのためのコードを挿入することで，本来ならビジーウェイトする時間を削減することができたことにより，radiosity において通常実行に比べて 4.0%の実行時間の削減を達成した．ただし cholesky では，ビジーウェイトの実行時間の削減に比べコンテキストスイッチに伴って発生するオーバヘッドが上回り，9.8%実行時間が悪化した．また，ビジーウェイトが存在しないその他のベンチマークプログラムでは，ユーザスレッドのコンテキストスイッチ回数が増加したため一様に低速化した．しかし，2 つの手法を組み合わせた手法では，全ての評価対象プログラムを実行することが可能となり，さらにユーザスレッドコンテキストスイッチの回数増加に伴う実行時間の悪化を多くのプログラムで抑制することができた．

今後の課題として，以下の 5 つが考えられる．1 つ目の課題として，プログラムの挙動を正確に把握し，それに合わせてスレッドを動的に再統合することが挙げられる．今回提案した動的スレッド再統合手法では，スレッドを再統合するタイミングをバリア同期に限定した．しかし，バリア同期は動的スレッド再統合手法を実現するための簡易的な指標であるため，この手法ではプログラムの挙動の変化を正確にとらえてスレッドを再統合しているとは言えない．そのため，プログラムの挙動の変化を正確に検出する方法，さらには非同期に動作しているユーザスレッドをどのように再統合するのかの 2 点を今後考える必要がある．

2 つ目の課題として，グラフ分割アルゴリズムで使用するパラメータの再検討が挙げられる．動的スレッド再統合手法では，Thread Tailor が使用するパラメータに加え，スレッドマイグレーションに伴って発生するキャッシュミスペナルティをパラメータとして使用した．しかし，これらのパラメータは実行の際に発生する全てのオーバヘッドを考慮しているわけではなく，コミュニケーショングラフを適切に分割できていない可能性がある．そのため，今後どのようなパラメータを使用することで，より適切にグラフを分割することができるのか検討する必要がある．

3 つ目の課題として，Thread Tailor で実行できるプログラムのさらなる拡充が挙げられる．本研究ではビジーウェイトに着目し，Thread Tailor の適応可能プログラムの拡大を図った．しかし，ビジーウェイトを含むプログラム以外にも Thread Tailor を利用すると実行が完了しないプログラムがまだ存在する可能性がある．このことから今後は，さまざまなプログラムコードを解析し，Thread Tailor では実行できないプログ

ラムコードの特徴を見つけること，そして，そのような特徴を持つプログラムに対してどのように対応するべきか考えることが必要である．

4つ目の課題として，コンテキストスイッチによる実行速度の低下を抑制することが挙げられる．コンテキストスイッチコードを挿入することによって，Thread Tailorを用いて実行できるプログラムは拡充された一方で，多くのプログラムで実行速度の低下が見られた．さらに，cholesky に関してはコンテキストスイッチ回数を削減することで実行速度の向上が得られたことから，コンテキストスイッチ回数を削減する重要性は高いと考えられる．そのため，今後トランスレータを改良し，コンテキストスイッチコードの挿入条件を厳格化する必要がある．

最後の課題として，プロファイリングの精度の向上が挙げられる．静的ステージにおいて，プログラムをプロファイリングしている際に，OS カーネルが一度プログラムの実行を停止させ，プロセッサコアに他のプロセスを割り当てることがある．既存手法，提案手法共に，このような状況を想定してプロファイリングしていないため，プロファイリング結果が乱れることが多々ある．不正確なプロファイリング結果は，誤ったスレッド統合につながり，実行速度を大きく低下させる要因となる．そのため，今後こうした状況にも対応可能なプロファイリング方法の再検討が必要であると考えられる．

謝辞

本研究を進めるにあたり，研究の機会を与えて下さり，何度も貴重なご意見を賜わり，夜遅くまで相談に付き合って頂くなど，終止熱心にかつ丁寧に御指導頂いた名古屋工業大学の津邑公暁准教授に深く感謝致します．そして，本研究のために，多大な御尽力を頂き，御指導を賜わり，幾度となく貴重な助言を頂いた松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に感謝の意を表します．また，研究していく上で多くの指摘や刺激を頂いた江藤正通氏，小田遼亮氏，山田龍寛氏，内山寛章氏，澤田晃平氏，柳田大輝氏，安井裕亮氏には感謝の念に堪えません．

最後になりましたが，本研究はもちろん，実生活の面でも多くの助言，協力，激励を頂いた研究グループ内のメンバー，ならびに松尾・津邑研究室，齋藤研究室，松井研究室の方々に深く感謝致します．

著者発表論文

論文

1. Kosuke SOBUE, Tomoaki TSUMURA, Hiroshi MATSUO: “An Efficient Thread Recombining at Program Phase Changes”, Proc. 3rd Int’l. Workshop on Advances in Networking and Computing (WANC’12), Okinawa, Japan, pp.316–320 (Dec. 2012)

参考文献

[1] Forum, M. P. I.: MPI: A Message Passing Interface Standard, *International Journal of Supercomputer Applications*, Vol. 8, No. 3/4, pp. 159–416 (1994).

[2] Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5 (1998).

[3] Nichols, B., Buttlar, D. and Farrell, J. P.: *Pthreads Programming*, O’Reilly (1998).

[4] Lee, J., Wu, H., Ravichandran, M. and Clark, N.: Thread Tailor:Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications, *Proc. 37th Annual Int’l Symp. on Computer Architecture*, pp. 270–279 (2010).

[5] McCalpin, J. D.: Memory Bandwidth and Machine Balance in Current High Performance Computers, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25 (1995).

[6] http://en.wikipedia.org/wiki/Hardware_performance_counter: .

[7] Intel: *Using the RDTSC Instruction for Performance Monitoring* (1997).

[8] Cohen, W. E.: Tuning Programs with OProfile, *WIDE OPEN MAGAZINE*, pp. 53–62 (2004).

[9] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, *Proc. 2004 Int’l Symp. on Code Generation and Optimization (CGO’04)*, pp. 75–86 (2004).

[10] Kernighan, B. W. and Lin, S.: An Efficient Heuristic Procedure for Partitioning Graph, *Bell System Technical Journal*, pp. 291–307 (1970).

[11] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd*

Annual Int'l Symp. on Computer Architecture (ISCA'95), pp. 24–36 (1995).

[12] Bienia, C., Kumar, S., Singh, J. P. and Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications, *Proc. 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 72–81 (2008).

[13] Sawalha, L., Tull, M. P. and Barnes, R. D.: Thread scheduling for heterogeneous multicore processors using phase identification, *SIGMETRICS Perform. Eval. Rev.*, Vol. 39, No. 3, pp. 125–127 (2011).

[14] Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C. and Wilkerson, C.: Scheduling threads for constructive cache sharing on CMPs, *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, New York, NY, USA, ACM, pp. 105–115 (2007).

[15] Rajagopalan, M., Lewis, B. T. and Anderson, T. A.: Thread scheduling for multicore platforms, *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, Berkeley, CA, USA, USENIX Association, pp. 2:1–2:6 (2007).

[16] Winter, J. A., Albonesi, D. H. and Shoemaker, C. A.: Scalable thread scheduling and global power management for heterogeneous many-core architectures, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, New York, NY, USA, ACM, pp. 29–40 (2010).

[17] Jung, C., Lim, D., Lee, J. and Han, S.: Adaptive Execution Techniques for SMT Multiprocessor Architectures, *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 236–246 (2005).

[18] Suleman, M. A., Qureshi, M. K. and Patt, Y. N.: Feedback Driven Threading: Power-Efficient and High-Performance Execution of Multithreaded Workloads on CMPs, *Proc 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 277–286 (2008).