平成23年度　　卒業研究論文

スレッド統合に基づくスケジューラ Thread Tailor におけるグラフ分割の改良

指導教員
津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成19年度入学　19115089番

辻 孝辰

平成24年2月8日

目 次

1 はじめに 1

2 Thread Tailor 3

2.1 Thread Tailor の基本概念 3

2.2 各ユーザスレッドの挙動の解析 5

2.3 統合するユーザスレッドの決定 9

2.4 ユーザスレッドの統合 12

3 グラフ分割方法の改良案 16

3.1 パラメータ算出方法の改良 16

3.2 新たなパラメータの導入 16

3.3 分割アルゴリズムの静的選択 17

4 高精度なグラフ分割の実装 18

4.1 サブセットのメモリバンド幅算出方法の変更 18

4.2 新たなパラメータの導入 18

5 評価 21

5.1 評価環境 21

5.2 評価結果 22

5.3 考察 23

6 おわりに 25

1 はじめに

これまで，プログラムの実行を高速化する手法として，スーパスカラやアウト・オブ・オーダー実行といった命令レベル並列性（Instruction-Level Parallelism: ILP）に着目したものが研究されてきた．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．一方，半導体技術の向上によって集積回路の微細化が進み，単一コアの性能向上が図られてきた．しかし，消費電力や配線遅延の相対的増大により，単一コアの性能向上による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コアを集積したメニーコアプロセッサが広く普及してきている．このメニーコアプロセッサの性能を有効利用するために，プログラマはスレッド並列性を利用したプログラミングを行う必要がある．そのために， MPI, OpenMP といった並列化フレームワークや Pthread などのライブラリが提供されている．

しかし，プログラマや，これらフレームワーク，ライブラリにとっての課題の一つが，適切なスレッド数を選択することである．例えば，生成するスレッドが少なければメニー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・同期回数の増大を招き，プログラムの実行速度を低下させてしまう．また，プログラム実行時のシステムの状態やシステム構成，プログラムへの入力の変化などによって，適切なスレッド数はプログラムを実行する度に変化するため，スレッド数をプログラム実行時に動的に選択する必要がある．

これを実現するための研究として Thread Tailor [1] がある．Thread Tailor では，プログラム実行時に適切なスレッド数を判断し，ユーザが実装したスレッドであるユーザ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．そして，統合後のスレッド内でのスケジューリングにより通信・同期のコスト削減を図る．この機構において重要となるのが，どのユーザスレッドを統合すべきかを決定する方法である．同じスレッド数でも，どのユーザスレッド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．そこで，Thread Tailor では，まずユーザスレッドをノードとし，ユーザスレッド間の通信・同期をエッジとする，ユーザスレッドの相互関係を表すグラフ

を作成する．そして，どのユーザスレッドを統合すべきかを決定するために，グラフ分割アルゴリズムを適用してノードをサブセットに分割する．そして，このサブセットを統合後のスレッドとして実際に統合を行う．しかし，このグラフの分割において，分割アルゴリズムの精度や，分割の際に考慮されるパラメータの算出精度が悪いという問題点が存在する．そこで，本研究では Thread Tailor におけるグラフ分割方法に着目し，適宜改良する．これにより，メニーコアプロセッサの性能を最大限に引き出し，プログラム実行速度の向上を目指す．

以下，2 章では本研究で扱う Thread Tailor の概要と処理の流れを説明する．3 章ではグラフ分割方法を改良するための 3 つの手法を提案し, 4 章ではその実装方法を述べる．そして，5 章で本提案手法の評価と考察を行い，最後の 6 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 の概要について説明する．

2.1 Thread Tailor の基本概念

Thread Tailor では，まずプログラム実行時にユーザ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．その後，統合したスレッド内でのユーザスレッド間の通信・同期をできるだけ削減できるようにスケジューリングする．これにより，マルチスレッドプログラムを高速に実行できる．また，Thread Tailor はプログラム実行開始時に適切なスレッド数を判断するので，プログラマがプログラム実行時の環境などを意識する必要はない．

Thread Tailor の処理の流れを図 1 に示す．処理全体は，図 1 に示されているように，プログラム実行前に静的に行う処理と，プログラム実行時に動的に行う処理からなる．Thread Tailor では，まずどのユーザスレッドを統合するかを決定するために，各ユーザスレッドの挙動を事前に解析してデータを収集する．挙動の解析のために，実行プログラムのバイトコードを改変する．具体的には，通信コストを算出するために必要となる，各ユーザスレッドの各メモリアドレスへのアクセス回数をカウントできるように改変する．そして，改変されたバイトコードをプロファイルし，そこから得られた結果を解析する．そして，解析によって得られた各ユーザスレッドの実行サイクル数や，各ユーザスレッド間の通信・同期コストなどのデータから，Communication Graph という本手法独自の形式のグラフを生成する．このグラフは，ユーザスレッドの実行サイクル数などをノードとして，ユーザスレッド間の通信・同期コストをエッジとして構成される．このグラフはメタデータとして保存され，プログラム実行時に動的コンパイラによって使用される．プログラムの実行が始まると，動的コンパイラはシステムのスナップショットを取ることにより，システムの状態を低いオーバヘッドで解析する．そして，その解析結果から利用可能な空きリソースの量を算出する．この算出結果を基にして，Communication Graph にグラフ分割アルゴリズムを適用することにより，ノードをサブセットに分割し，どのユーザスレッドを統合するかを決定する．こ

こでは，スレッド間での資源競合を起こさないという制約の下で，各コアでの処理量を平均化し，且つサブセット間の通信・同期の量をなるべく抑えられるようにグラフの分割を行う．そして，どのユーザスレッドを統合すべきかが決定された後，実際にユーザスレッドを統合するために，動的コンパイラがバイトコードを書き換える．以降各ユーザスレッドの挙動の解析，統合するユーザスレッドの決定，ユーザスレッドの統合について，各々の動作を詳しく述べる．

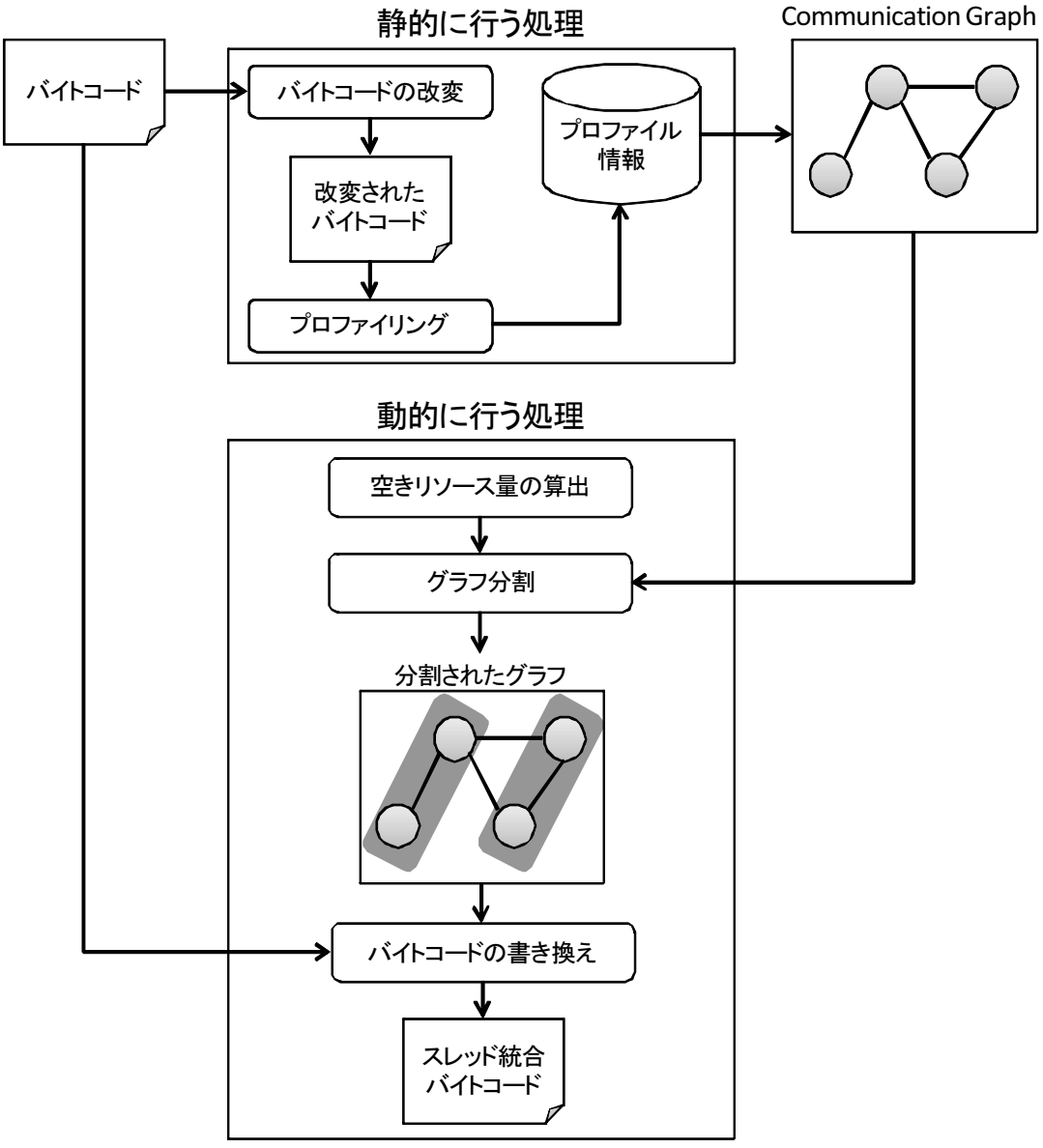

図 1: Thread Tailor の処理の流れ

2.2 各ユーザスレッドの挙動の解析

Thread Tailor は，初めに各ユーザスレッドの挙動を解析し，その結果得られたデータから Communication Graph を作成する．このグラフは，図 2 に示すようなものであり，ユーザスレッドの実行サイクル数などをノードとして，ユーザスレッド間の通信・同期コストをエッジとして構成される．各ノードは，ストールサイクル数を含めた実行サイクル数（$Cycles$），ワーキングセットサイズ（$WorkSet$），必要とするメモリバンド幅（BW）の 3 つの重みを持つ．各エッジはエッジの両端ノード，つまりユーザスレッド間で発生する通信と同期に要するサイクル数の合計値（$CommCost$）を重みとして持つ．この内，実行サイクル数は統合後のスレッドの総実行サイクル数を見積るために使用される．ワーキングセットサイズはユーザスレッドが必要とする共有キャッシュの大きさを表し，統合後のスレッドのワーキングセットサイズが，使用可能な共有キャッシュ容量を上回らないようにグラフを分割する．これにより，統合後のスレッド間での共有キャッシュの競合を抑制する．メモリバンド幅は，統合後のスレッドの必要とする量が，プログラムの使用可能な量を超えないようにグラフを分割するためのパラメータとして使用される．これにより，統合後のスレッド間でのメモリアクセスの競合を抑制する．最後に，通信・同期コストは統合後のスレッド間での通信・同期コストを見積もるために使用される．また，統合後のスレッド内のユーザスレッド間では通信・同期の必要が無くなるため，ユーザスレッドを統合することによってどれだけ通信・同期サイクルが削減されるのかを見積ることもできる．

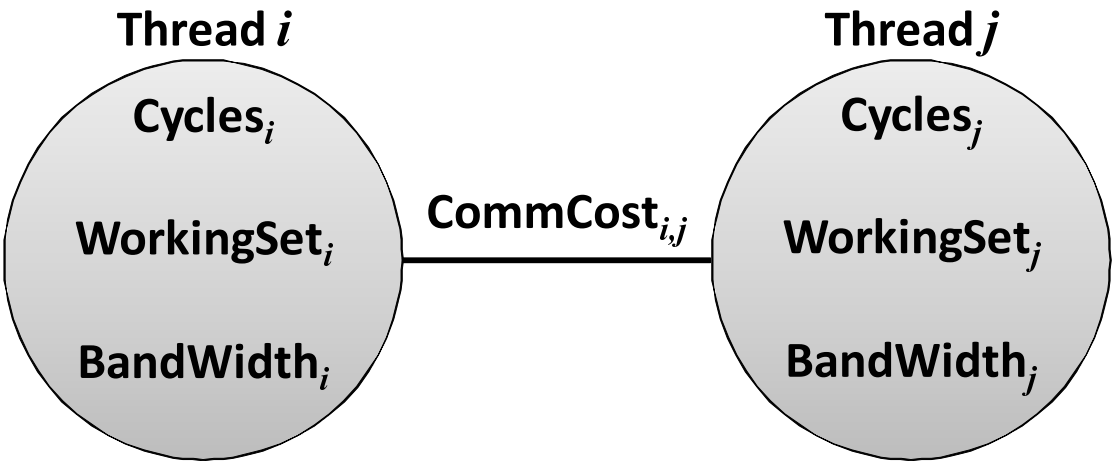

図 2: Communication Graph の構成

この Communication Graph を作成するために，一度プログラムを実行して解析し，グラフ作成に必要なデータを収集する．実行サイクル数とメモリバンド幅については CPU に備わっているハードウェア・パフォーマンス・カウンタを使用して計測できる．一方，ワーキングセットサイズと通信・同期コストについては，ハードウェア・パフォーマンス・カウンタでは計測できないため何らかの方法でおおよその値を見積る必要がある．

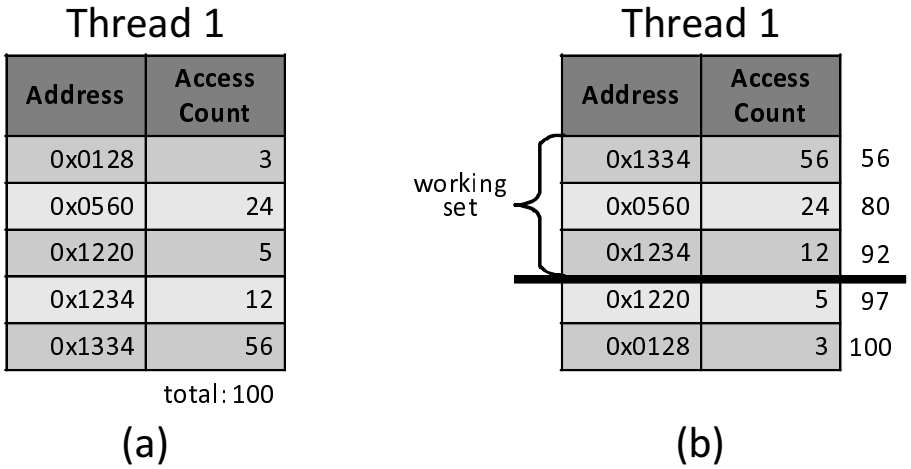

図 3: ワーキングセットの推定

まず，ワーキングセットサイズを算出するために，Thread Tailor では各ユーザス

レッドの全てのロード・ストア命令についてそのアクセス先アドレスを調べる．そして，ユーザスレッド毎に各アドレスへのアクセス回数を調べる．アクセス回数は，ロード回数とストア回数の和である．その結果，図 3(a) に示すような表を得る．そして，図 3(b) に示すように，アドレスをアクセス回数について降順にソートし，アクセス回数が多い方から順にアクセス回数を加算していく．その結果が，メモリへの総アクセス回数の 90%を超えた時点で，そのアドレスまでをワーキングセットとする．したがって，ワーキングセットサイズはこれらのアドレスの個数とキャッシュブロックサイズを掛けた値になる．

Thread 1

Address	Load Count	Store Count	Access Count
0x1234	5	10	15
0x1338	4	7	11
0x4000	7	7	14

Thread 2

Address	Load Count	Store Count	Access Count
0x1234	0	7	7
0x2000	4	4	8
0x4000	3	8	11

図 4: メモリアクセス情報

通信コストもこのメモリアクセス情報を使用して算出する．ある 2 ユーザ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，現在多くのプロセッサでは，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 ユーザスレッド間で通信する必要がある．例えば，図 4 に示すように，Thread2 がアドレス 0x1234 に 7 回ストアし，Thread1 がそのアドレスから 5 回ロードすることが分かっているとする．この場合，2 ユーザスレッド間で 5 回通信が発生すると予測できる．そして，その他にアドレス 0x1234 に関して通信が発生する可能性があるのは，Thread1 がストアし，Thread2 がロードした場合と，両ユーザスレッドがストアした

場合である．したがって，アドレス 0x1234 に関して，Thread1，Thread2 間で通信が発生する回数は，

$$\min(5,7) + \min(10,0) + \min(10,7) = 12 \tag{1}$$

といった計算により，12 回であると予測できる．また，アドレス 0x4000 に対しても同様に計算すると，

$$\min(7,8) + \min(7,3) + \min(7,8) = 17 \tag{2}$$

となる．図 4 に示す例では，両ユーザスレッドからアクセスされるアドレスは，0x1234 と 0x4000 のみである．つまり，0x1234 と 0x4000 へアクセスする際に通信が発生し得ると考えられる．したがって，$12 + 17 = 29$ より，Thread1，Thread2 間の総通信回数は 29 回であると予測できる．このように，両ユーザスレッドからアクセスされる全てのアドレスに関して，2 ユーザスレッド間で通信が発生する回数を計算し，それらの和をとることで，2 ユーザスレッド間の総通信回数を予測する．そして，通信にかかるサイクル数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（コミュニケーション・レイテンシ）を乗じる必要がある．コア数を $Cores$，共有キャッシュのレイテンシを L とすると，コミュニケーション・レイテンシは，経験則に基づき，$3 \times \sqrt{Cores} \times L$ で算出できる．この式では，共有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シプロトコルにより一貫性を保つものであることを想定している．そのため，$\sqrt{Cores}$ は 2 コア間の距離の平均である．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の 3 回のメッセージを意味する．

最後に，同期コストについては同期にかかる待ちサイクル数を計測することにより算出する．例えば，`pthread_join()` による待ちサイクル数を計算するためには，その開始時間と終了時間を取得し差分をとる．しかし，開始と終了が検出できない同期処理に関しては待ちサイクル数を算出することができない．例えば，共有変数がある値になるまで待つ，という同期処理を，`pthread_cond_wait` のようなライブラリ関数を使わずに，共有変数へのポーリングによって実装した場合，Thread Tailor はこの同期処理の開始と終了を検出することができない．しかし，このような同期処理にかかるサイクル数に関しては，ユーザスレッドを統合しても削減できないので，計測する

必要がない．

以上で述べた計測方法により，各データを取得する．そして，取得したデータを基に Communication Graph が作成される．

2.3 統合するユーザスレッドの決定

2.2 節で述べた方法により作成されたグラフのノードは，プログラム実行開始時に動的コンパイラによっていくつかのサブセットに分割される．このサブセットは，統合後のスレッドに相当する．つまり，ノードをサブセットに分割することは，どのユーザスレッドを統合するかを決定することと同義である．この処理はグラフ分割と呼ばれる．そして，各サブセットの実行サイクル数ができるだけ均等になり，各サブセット間の通信・同期コストがなるべく小さくなるようにグラフは分割される．ただし，サブセット間での資源競合を抑制するため，各サブセットのワーキングセットサイズが，使用可能な共有キャッシュ容量を超えずに，且つ各サブセットの要求するメモリバンド幅が，使用可能なメモリバンド幅を超えないように配慮しなければならない．したがって，まずはグラフ分割前に使用可能な共有キャッシュ容量とメモリバンド幅を算出する必要がある．実行するプログラムが使用可能な共有キャッシュ容量は，共有キャッシュの総容量からシステム内で実行されている他のプログラムが使用している容量を減じたものである．また，プログラムが使用可能なメモリバンド幅については，システムによって決まるメモリバンド幅の上限値からシステム内で実行されている他のプログラムガ利用しているメモリバンド幅を減じたものである．

なお，各サブセットの実行サイクル数等の各パラメータを見積もるために，統合後のスレッドのパラメータの算出方法を定義する必要がある．まず，前提条件として，サブセット内の各ユーザスレッドは 1 つのスレッドに統合され，統合後のスレッド内でコンテキスト切り替えにより逐次に実行される．よって，単純に考えるとサブセットの実行サイクル数はサブセット内の各ユーザスレッドの実行サイクル数の総和となる．しかし，サブセット内ではユーザスレッド間で通信・同期の必要がなくなると考えられるため，それらのサイクル数を除くべきである．したがって，あるサブセット A の

実行サイクル数$Cycles_A$は，

$$Cycles_A = \sum_{i \in A} Cycles_i - \sum_{i \in A, j \in A} CommCost_{i,j} \tag{3}$$

として計算される．ワーキングセットサイズとメモリバンド幅については，単にサブセット内の各ユーザスレッドの値の合計値とするので，各々次のような式になる．

$$WorkSet_A = \sum_{i \in A} WorkSet_i \tag{4}$$

$$BW_A = \sum_{i \in A} BW_i \tag{5}$$

そして，グラフの分割時には，資源競合を抑えるために各サブセットのメモリバンド幅とワーキングセットサイズのそれぞれが使用可能な量を超えないようにしなければならない．そのため，プログラムが使用可能なメモリバンド幅を$MemBW$，共有キャッシュ容量を$CacheAllocation$とすると，$BW_A \leq MemBW$，$WorkSet_A \leq CacheAllocation$として表される制約式を満たさなければならない．また，各サブセット同士は並列に実行されるため，プログラムの実行時間は最も実行サイクル数の大きいサブセットの実行時間に影響されることになる．よって，各サブセットの実行サイクル数ができるだけ平均化されるようにグラフを分割し，実行時間の増加を抑える．ただし，これを完全に達成することはNP困難であることが知られており，膨大な時間がかかってしまう．そこで，Thread Tailor では Kernighan-Lin アルゴリズム（KL アルゴリズム）[2] を用いて，グラフを分割する．KL アルゴリズムはNP困難な問題において部分最適解を求めるもので，巡回セールスマン問題などにも応用されている．さらに，グラフ分割問題においてはグラフのサイズのオーダーで問題を解けることが知られている．

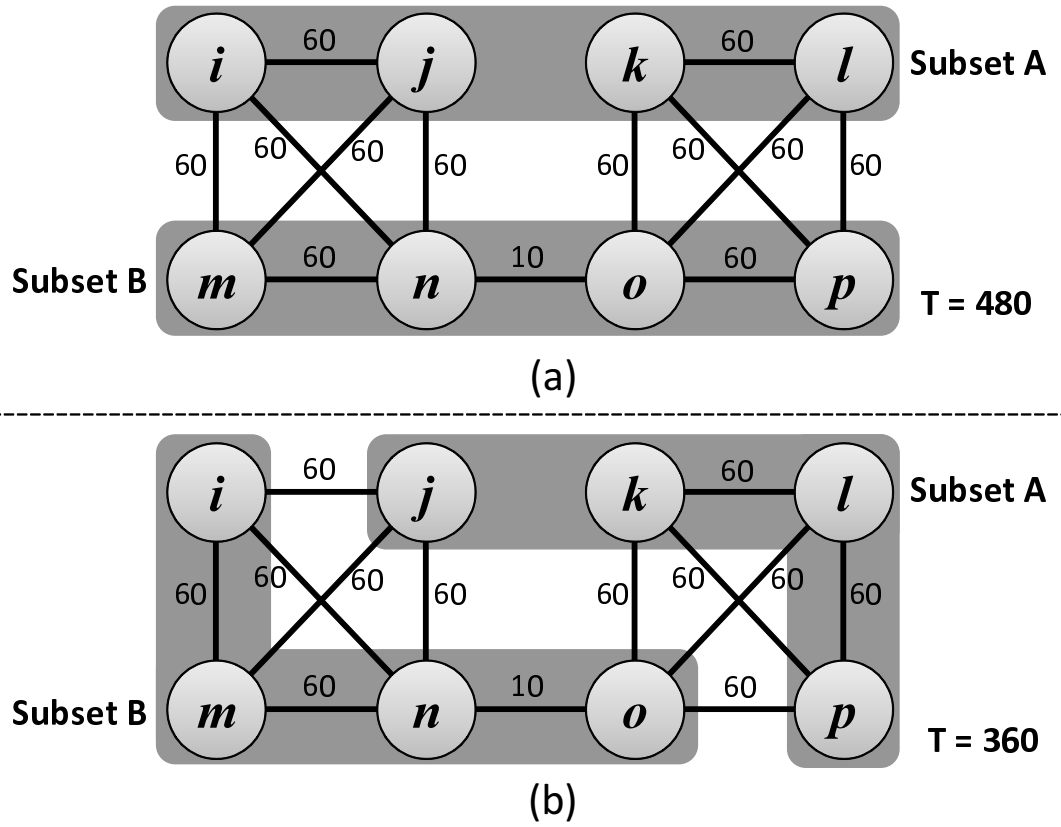

図 5: KL アルゴリズムによるグラフ分割

KL アルゴリズムは，サブセット間にあるエッジコストの合計値（以下 T とする）を最小化する．これにより，各サブセットの実行サイクル数を平均化し，最も実行時間のかかるスレッドが実行している処理の終了をなるべく早める．KL アルゴリズムでは，初めに，グラフを同じ数のノードを持つ 2 つのサブセットに分割する．これは，どのような分け方でもよい．そして，2 つのサブセット間でノードの交換を行うことにより，T の削減を図る．ここでいうノードの交換とは，ノードの所属を一方のサブセットからもう一方のサブセットへ移すという処理を，相互に行うことである．KL アルゴリズムの動作例を，図 5 に示す．まず初期の分割状態が，図 5(a) に示すように，グラフがノード i,j,k,l からなるサブセット A とノード m,n,o,p からなるサブセット B の 2 つのサブセットに分けられているとする．このときの T は 480 となる．ここで，交換することによって T ができるだけ削減されるような，A 内のノードと B 内のノードの組み合わせを探す．A 内のあるノード x と B 内のあるノード y を交換することによっ

て削減される T のコストを $Gain$ とすると，

$$Gain = D_x + D_y - 2C_{x,y} \tag{6}$$

として計算される．ここで，$C_{x,y}$ は x, y 間のエッジコストである．D_x とは x と他サブセット内のノードとの間にあるエッジコストの総和 E_x から，x と自サブセット内のノードとの間にあるエッジコストの総和 I_x を減じたものである．例えば，ノード i は他サブセットであるサブセット B のノード m,n との間にエッジを持つ．したがって，$E_i = C_{i,m} + C_{i,n} = 120$ となる．また，ノード i は自サブセットであるサブセット A のノード j との間にエッジを持つ．したがって，$I_i = C_{i,j} = 60$ となる．以上より，$D_i = E_i - I_i = 60$ となる．このようにして，A,B 間の全てのノードの組み合わせにおいて $Gain$ を計算し，それが正で且つ最も大きくなる 2 つのノードを交換する．図 5 に示したグラフの例では，i,p を交換することによって得られる $Gain$ が 120 となり，最大である．したがって，i と p を交換する．この交換により，サブセットは図 5(b) のようになり，T は 480 から 360 になる．そして，一度交換されたノードを除いて，再び A,B 間で $Gain$ を計算し，全ての $Gain$ が 0 以下になるまで交換を繰り返す．その結果，図 5 に示したグラフの例では最終的に，A が k,l,o,p，B が i,j,m,n となる．このようにして，KL アルゴリズムを 1 回適用すると実行サイクル数がほぼ同じである 2 つのサブセットができる．そして，サブセット数がシステムで使用しているプロセッサのコア数と同じ数になるまで KL アルゴリズムを繰り返し適用する．

2.4 ユーザスレッドの統合

グラフ分割によって，どのユーザスレッドを統合するかが決定されると，実際にユーザスレッドを統合する処理に移る．そのために，プログラム実行中，動的コンパイラはスレッド生成や通信，同期を行う関数の呼び出しを検出すると，それらを Thread Tailor に制御を移すためのラッパ関数に書き換える．これにより，通常 OS によって制御される，スレッドの生成や通信，同期の処理を Thread Tailor が代行できるようになる．つまり，OS でなく，Thread Tailor によってスレッドをスケジューリングする．

まず，同じサブセットに属するユーザスレッドは，全て同じスレッド内で実行されるようにする必要がある．そして，スレッド内で複数のユーザスレッドを実行するためには，各ユーザスレッドをコンテキストスイッチによって切り替える必要がある．そのため，Thread Tailor では，統合対象となるユーザスレッドの生成時に，スレッド内にユーザスレッドのコンテキスト情報を保存する．例えば，`pthread_create()` のようなスレッド生成関数の呼び出しは，`vm_thread_create()` というラッパ関数の呼び出しに書き換えられる．これが呼び出されると動的コンパイラに制御が移る．制御が移ってからの処理のフローを図 6 に示す．まず動的コンパイラは，生成対象のユーザスレッドが統合の対象であるかどうかを，分割後のグラフを用いて判断する．そして，もし統合対象でなかった場合，新たにスレッドを生成する．一方，統合対象であった場合，処理は 2 通りに分かれる．例として，生成対象のユーザスレッド 1 はサブセット A に属し，サブセット A 内のユーザスレッドは統合によりスレッド A 内で実行されることになるとする．このとき，スレッド A が生成されていなければ，動的コンパイラはスレッド A を生成する．このスレッドには，ユーザスレッドのコンテキスト情報を保存しておくための配列が作成され，その中にユーザスレッド 1 のコンテキスト情報が保存される．一方，スレッド A が既に生成済みの場合は，新たにスレッドを生成せず，ユーザスレッド 1 のコンテキスト情報をスレッド A 内の配列に保存する．このようにして，同じサブセット内に属するユーザスレッドは，同じスレッド内で実行されるようになる．

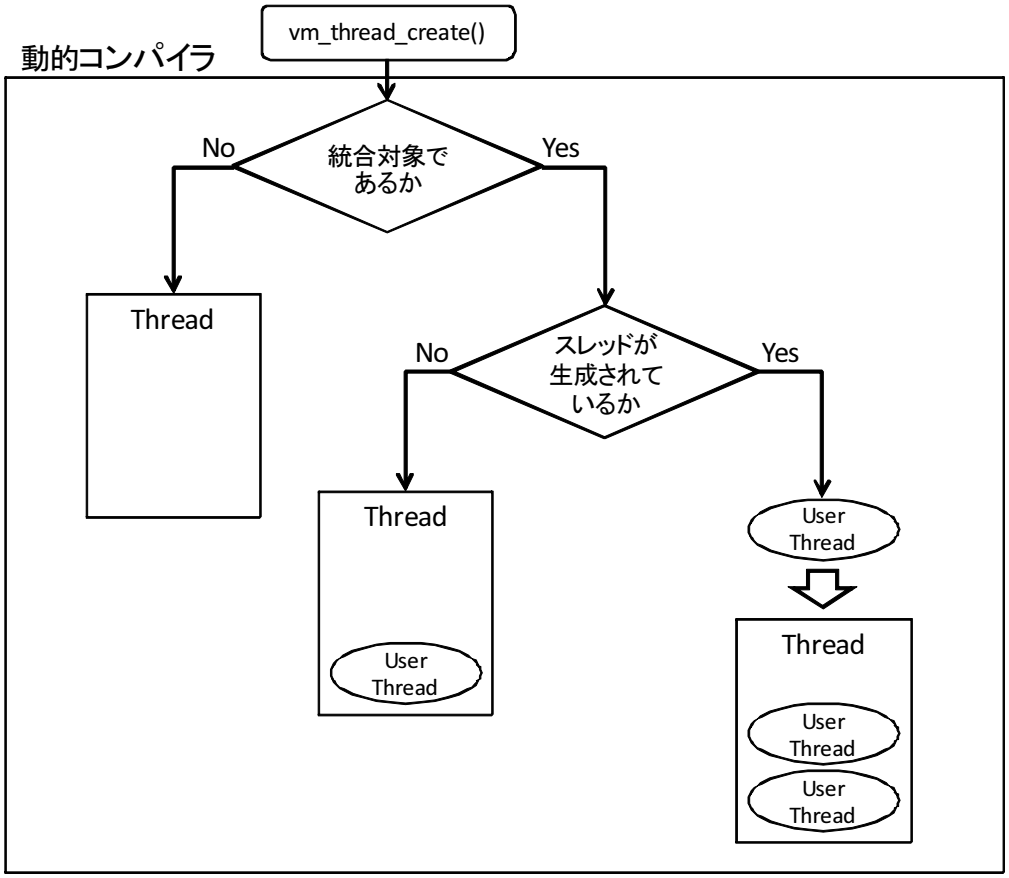

図 6: 動的コンパイラによるユーザスレッド統合

ただし，以上の処理だけでは，同じスレッド内のユーザスレッド間で不必要な通信や同期による待ちサイクルが発生し，スレッドを統合することによるメリットは希薄になってしまう．そこで，同じスレッド内のユーザスレッド間の通信・同期コストをできるだけ削減する．まず，MPI_Send のような明示的な通信については，メッセージの送信者と受信者が同じスレッド内のユーザスレッド同士であれば，ブロッキングによるコストを削減できる．このような通信には，1 回の通信に対してメッセージ保存用のバッファが 1 つ確保される．そして，MPI_Send はバッファへの write 処理，MPI_Recv はバッファからの read 処理に変換される．これによって，送信者となるユーザスレッドは，MPI_Recv を待たずに次の処理を続けられる．一方，キャッシュコヒーレンシを保つための暗黙的な通信については，キャッシュコヒーレンシプロトコルによって管理されているため，Thread Tailor によって制御する必要がない．

また，排他制御を削減できるケースもある．それは，あるクリティカルセクションへ侵入するユーザスレッドが，全て同じスレッド内のユーザスレッドである場合の，クリティカルセクションへの排他制御である．この場合，あるユーザスレッドがクリティカルセクション実行中に，スケジューラによってユーザスレッドが切り替えられなければ，他のユーザスレッドがクリティカルセクションに侵入しようとすることは有り得ない．したがって，排他制御の必要がなくなり，それにかかるコストを削減できる，各クリティカルセクションへ侵入するユーザスレッドは，Andersen によるメモリ依存解析 [3] を利用することで特定できる．

3 グラフ分割方法の改良案

Thread Tailor では，グラフの分割結果によってプログラムの実行時間が大きく変化する．したがって，分割方法を改良することは実行速度の大幅な向上に繋がると考えられる．そこで，本章ではグラフ分割方法を改良する 3 つの手法を提案する．

3.1 パラメータ算出方法の改良

2.2 節でも述べたように，グラフをサブセットに分割する際には，サブセットのメモリバンド幅やワーキングセットサイズが制約を満たすようにしなければならない．したがって，サブセットの各パラメータの算出結果は分割結果に影響を与え得る．

例えば，サブセットのメモリバンド幅の算出精度が悪く，算出された値が実際の値よりもかなり大きな値であった場合，初期のグラフ分割状態からどのノードを交換しても各サブセットのメモリバンド幅が $MemBW$ を超えてしまうので，ノードの交換が一切行なわれない．つまり，初期の分割状態がそのまま分割結果になってしまうため，初期の分割の仕方によっては，あるサブセットの実行サイクル数が突出してしまい，プログラムの実行時間が増加してしまう可能性もある．逆に，実際の値よりもかなり小さな値であった場合，計算上では $MemBW$ を超えることはないが，実際にプログラムを実行してみるとメモリアクセスの競合が頻発してしまう可能性がある．そのため，サブセットの各パラメータをより正確に算出することで，より現実に則した分割結果が得られると考えられる．そして，その結果に従ってユーザスレッドを統合することによって，資源競合を抑えつつ，資源をより有効に利用できる．

3.2 新たなパラメータの導入

既存手法において，Communication Graph を構成するパラメータは，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，通信コスト，同期コストの 5 種類である．そして，これらのパラメータを考慮してグラフは分割される．つまり既存手法では，これら 5 種類のパラメータがプログラム実行の効率に関わってくると考えられているしかし，プログラム実行の効率に関わるパラメータがこれら 5 種類だけである

という保証はない．したがって，既存手法では考慮されていないパラメータを新たに導入し，それを考慮に入れてグラフを分割することによって，より精度の高い分割結果が得られると考えられる．事実，Thread Tailor ではその前身となる研究 [4] で考慮されていなかった，ワーキングセットサイズや通信コストを考慮に入れることで，グラフ分割の精度を上げ，プログラム実行速度の向上を実現した．本論文でも，新たなパラメータの導入により，プログラム実行の高速化を目指す．

3.3 分割アルゴリズムの静的選択

グラフ分割問題を含む，最適化問題を解くアルゴリズムでは，一般的に解の精度と計算時間はトレードオフの関係にある．そのため，短時間で精度の良い解を出すことは難しく，どちらか一方のみを優先せざるを得ない．そして，Thread Tailor のグラフ分割では，分割アルゴリズムの解の精度はプログラムの実行時間に影響し，計算時間はグラフ分割にかかる時間に影響する．既存手法で使用されている KL アルゴリズムは初めの分割状態から，サブセット間でノードを交換することによって，局所的な近似解を求めるものであるため，実行速度は速い．つまり，計算時間を優先したアルゴリズムである．Thread Tailor では，グラフ分割を動的に行う必要があるため，それにかかる時間をなるべく短縮した方が良いという考えに基づいて，KL アルゴリズムを採用している．しかし，実行するプログラムによっては，グラフ分割に時間をかけてプログラムの実行を高速化させた方が良い場合もある．例えば，実行時間の長いプログラムでは，グラフ分割にかかる時間が相対的に小さいため，多少時間のかかるアリゴリズムを利用してプログラムの実行時間を短縮させることにより，全体の実行時間も短縮できると考えられる．逆に，実行時間の短いプログラムでは，グラフ分割にかかる時間が相対的に大きいため，高速なアルゴリズムを利用して高速にグラフ分割を行うことにより，全体の実行時間を短縮できると考えられる．

そこで，複数のグラフ分割アルゴリズムを用意して，その中から利用するアルゴリズムをプログラム実行前に選択する手法を提案する．これにより，実行するプログラムに適したグラフ分割が可能となる．そして，プログラム実行前にアルゴリズムを選択するため，実質的には選択にコストがかからないことになる．

4 高精度なグラフ分割の実装

以下のように Thread Tailor のグラフ分割部を改良することにより，3 章で提案した手法を実装した．

4.1 サブセットのメモリバンド幅算出方法の変更

2.3 節で述べたように，既存手法では，サブセットのメモリバンド幅は，単にサブセット内の各ユーザスレッドのメモリバンド幅の合計値としている．そして，サブセットのメモリバンド幅が，プログラムが使用可能なメモリバンド幅を超えないようにグラフは分割される．しかし，サブセット内の各ユーザスレッドはコンテキストスイッチにより，切り替えられながら実行されるため，複数のユーザスレッドが同時に実行されることはない．すなわち，サブセット内の複数のユーザスレッドが同時にメモリアクセスすることもない．よって，既存手法におけるサブセットのメモリバンド幅の算出方法では，実際に統合後のスレッドが要求するメモリバンド幅よりも，かなり大きな値を算出してしまうことになる．

そこで，サブセット内の各ユーザスレッドのメモリバンド幅のうち，最大のものを，そのサブセットのメモリバンド幅とした．例えば，あるサブセット A のメモリバンド幅 BW_A の算出式は，

$$BW_A = \max_{i \in A} BW_i \tag{7}$$

となる．

4.2 新たなパラメータの導入

グラフ分割の精度を上げるために各ユーザスレッドのパイプラインフラッシュ回数とストールサイクル数をグラフ分割の際に考慮されるパラメータとして新たに導入した．しかし，この 2 種類のパラメータを一度に導入したわけではなく，それぞれを個別に導入した．つまり，2 パターンの実装を行った．そして，双方の実装において，統合後のスレッド間で導入したパラメータ値が平均化されるようにグラフを分割するために，グラフ分割アルゴリズムを一部改変した．

まず，パイプラインフラッシュはパイプラインアーキテクチャにおいて，パイプライン中の命令を全て破棄することである．そのため，これが頻発するとプログラムの実行効率を大きく低下させてしまう恐れがある．パイプラインフラッシュが発生する主な原因は，分岐予測が外れた場合であり，これをできるだけ回避するために分岐予測性能の向上が図られている．しかし，ユーザスレッド間でのパイプラインフラッシュ回数の偏りは小さいとは言えない．事実，SPLASH-2 [5] ベンチマークスイートのプログラムである water-n^2 を 16 スレッドで実行したところ，パイプラインフラッシュ回数が最少のスレッドと最多のスレッドでは，その回数に約 6 倍の差があることが確認できた．このように，スレッド間でパイプラインフラッシュ回数に偏りがあると，各スレッドの実行時間に偏りが発生してしまう恐れがある．既存手法では，実行サイクル数の平均化はしているが，実行時間が平均化されるという保証はない．よって，パイプラインフラッシュ回数を統合後のスレッド間で平均化することで，各スレッドの実行時間がより確実に平均化されると考えた．また，ストールサイクル数については，パイプラインフラッシュによるストールを対象とした．したがって，これを導入し平均化することにより，パイプラインフラッシュ回数の場合と同じような効果が期待できる．

これらのパラメータ値は，ハードウェア・パフォーマンス・カウンタを利用して計測した．しかし，パイプラインフラッシュ回数については直接計測することができない．そこで，パイプラインフラッシュの主な原因である分岐予測ミスの回数をパイプラインフラッシュ回数として扱った．また，他のパラメータと同様に，各サブセットのパラメータ値の算出方法を定義する必要がある．パイプラインフラッシュや，それに伴うストールは，ユーザスレッドの統合によって変化するものではないと想定し，単にサブセット内の各ユーザスレッドの値の合計値とした．つまり，これらのパラメータの値を P とすると，あるサブセット A のパラメータ値 P_A は，

$$P_A = \sum_{i \in A} P_i \tag{8}$$

として計算される．

そして，これらのパラメータ値を統合後のスレッド間で平均化する．ただし，実行サ

イクル数も平均化しなければならず，これらを同時に達成するのは不可能である．そこで，既存手法で利用されているグラフ分割アルゴリズムにおいて，ノードを交換する際の条件を追加した．その条件とは，ノードを交換することによって，2 つのサブセット間のパラメータ値の差が小さくなることである．ノードを交換する度に双方の差が小さくなれば，徐々に分散が小さくなる．そして，サブセット A のノード i とサブセット B のノード j を交換する場合の条件を式で示すと，

$$|(P_A - P_i + P_j) - (P_B - P_j + P_i)| \leq |P_A - P_B| \tag{9}$$

となる．この条件式が満たされれば，ノード i とノード j を交換する．

表 1: 評価環境

OS	Fedora 15
CPU	Intel Core2 Quad Q9550
動作周波数	2.83 GHz
コア数	4
一次命令キャッシュ	4 × 32 KB
一次データキャッシュ	4 × 32 KB
二次キャッシュ	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

5 評価

本章では各提案手法について行った評価について述べる．

5.1 評価環境

評価環境を表 1 に示す．提案手法を評価するために，Thread Tailor の簡易版を作成し，それに対して提案手法を実装した．簡易版では，実行時にシステム内で他のプログラムが動いていることを想定していない．また，実行プログラムへの入力は常に同じであると想定している．そして，スレッドのスケジューリングは実装しておらず，各ユーザスレッドは OS によってスケジューリングされる．そのため，他にプログラムが動いていない環境で評価し，実行プログラムには常に同じ入力を与えた．また，簡易版と簡易版に提案手法を実装したもので比較することにより，提案手法の有効性を評価した．

ベンチマークプログラムには，SPLASH-2 ベンチマークスイートの water-n^2 を使用した．ベンチマークプログラムで生成されるユーザスレッド数は 16 とした．そして，プログラムを簡易版とそれぞれの提案手法を実装したもので実行し，その実行時間を比較した．それに加え，グラフ分割に要した時間も比較した．

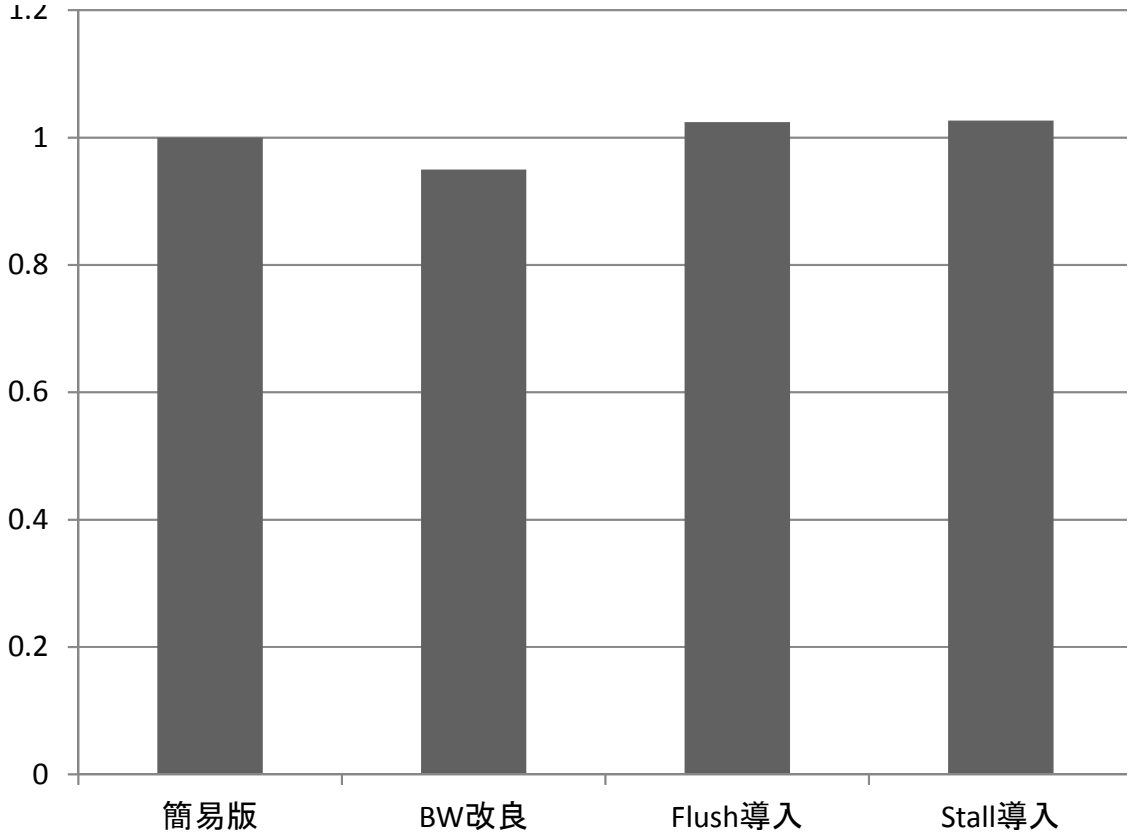

図 7: 全体の実行時間の比較

5.2 評価結果

評価結果を図 7，図 8 に示す．図 7 は，グラフ分割とプログラム本体の実行に要した時間である．また，図 8 は，グラフ分割に要した時間である．それぞれ，簡易版での実行時間を 1 として正規化しており，左から簡易版，サブセットのメモリバンド幅の算出方法を改良したもの，パイプラインフラッシュ回数を導入したもの，ストールサイクル数を導入したものの実行時間を表している．

まず，図 7 から，サブセットのメモリバンド幅の算出方法を改良することによって，全体の実行時間が 5.0%削減されたことを確認できる．また，パラメータの導入による改良では，それぞれ実行時間が増大してしまう結果となり，パイプラインフラッシュの導入によって 2.4%，ストールサイクル数の導入によって 2.6%の悪化となった．

そして，図 8 から，どの改良手法においてもグラフ分割に要する時間が増大したこ

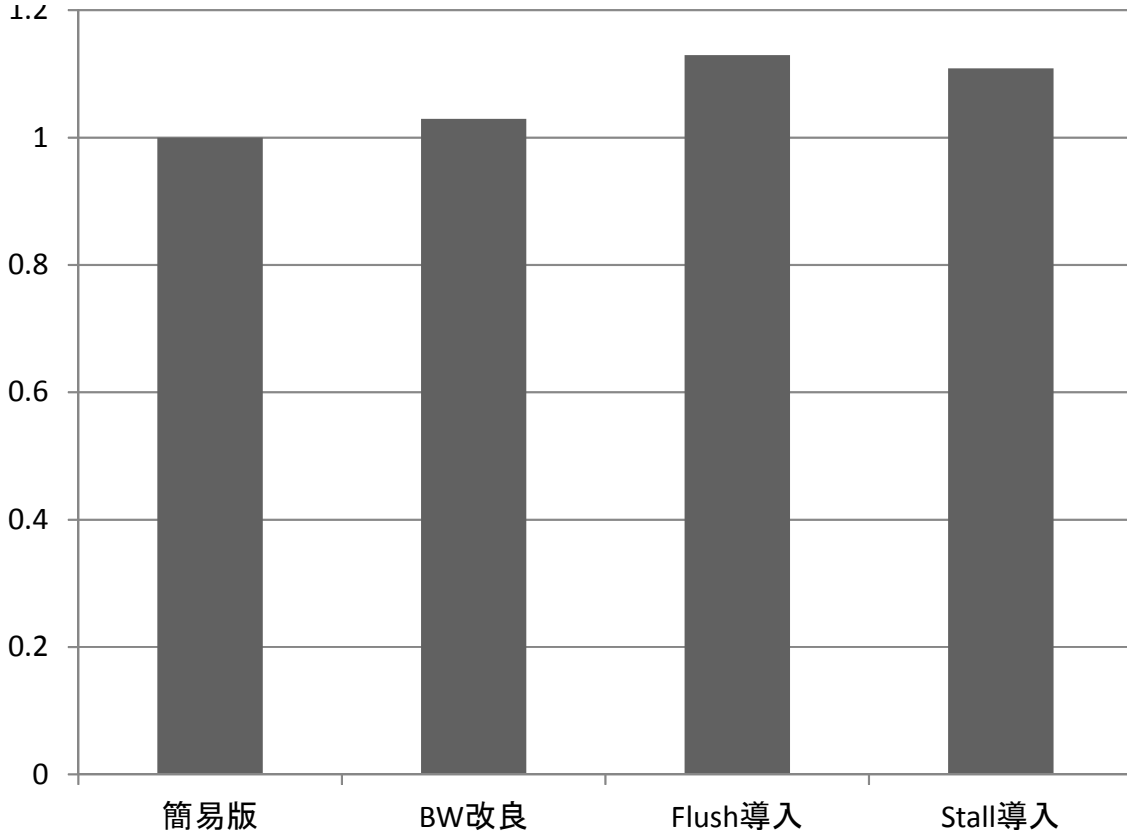

図 8: グラフ分割に要した時間の比較

とが分かる．また，全体の実行時間のうち，グラフ分割に要する時間が占める割合は，それぞれ 0.14%〜0.17% 程度であった．

5.3 考察

5.2 節の結果より，サブセットのメモリバンド幅の算出方法を改良することにより，性能向上が得られた．これは，各サブセットのメモリバンド幅が既存手法によって算出される値より小さくなったことで，メモリバンド幅に関する制約が相対的に緩和されたためであると考えられる．一方，新たなパラメータの導入によって，性能が悪化してしまった．その中でも，グラフ分割に要する時間の増大が目立つように思える．しかし，グラフ分割に要する時間は，全体の実行時間に対して微々たるものであるため，向上するはずのグラフ分割の精度が悪化してしまっていると考えられる．これは，新

たに設けたノードを交換する際の条件が厳しすぎたため，各サブセット間の実行サイクル数が平均化されなかったことが原因の一つであると考えられる．

6 おわりに

本研究では，Thread Tailor におけるグラフ分割方法の改良手法を 3 つ提案した．そのうち，サブセットのメモリバンド幅の算出方法を改良する手法と，新たなパラメータを導入することによる改良手法の有効性を確認するため，それらを簡易版に実装し，SPLASH-2 ベンチマークスイートの water-n^2 プログラムを用いて実行時間を評価した．その結果，簡易版と比較すると，サブセットのメモリバンド幅の算出方法を改良することによって約 5%の実行時間の削減を確認できた．一方，新たなパラメータを導入することによって，それぞれ約 2.5% 実行時間が増大してしまう結果となった．

本研究の今後の課題としては，まず様々な環境での評価を行うことが挙げられる．本論文では，簡易版を作成し，water-n^2 プログラムで 16 のユーザスレッドを統合した際の評価を行った．しかし，提案手法の有効性をより正確に確認するためには，まず既存手法の評価に用いられてた，PARSEC [6] ベンチマークスイートや，CUDA SDK Code Samples [7] のプログラムでの評価を行う必要がある．さらに，生成されるユーザスレッド数を変化させることで，提案手法による実行時間の削減率も変化する可能性がある．

そして，本論文では実装しなかった，グラフ分割アルゴリズムの静的選択による改良手法を実装することも課題として挙げられる．本論文の評価結果より，プログラムの実行時間に対して，グラフ分割に要する時間が微々たるものであることが確認できた．したがって，今後は既存手法で使用されているアルゴリズムよりも，時間はかかるが精度の良いアルゴリズムを実装して予備評価を行い，この手法の有効性を確認し，実装するべきかを考える必要がある．

また，新たなパラメータの導入についてはさらなる検討が必要である．本論文で行った評価結果より，グラフ分割の精度が向上するどころか悪化してしまうことが分かった．しかし，グラフ分割に要する時間の増大は小さかった．よって，導入するパラメータ自体の見直しや，そのパラメータをどう考慮に入れるのかの見直しをすることによって，性能を向上できると考えられる．

謝辞

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名古屋工業大学の津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室及び松井研究室の先輩，同期，そして研究グループ内の方々に深く感謝致します．特に，研究に関して貴重な意見を下さった祖父江宏祐氏，安井裕亮氏に感謝致します．ありがとうございました．

参考文献

[1] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, pp. 270–279, June 2010.

[2] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, Vol. 49, No. 1, pp. 291–307, 1970.

[3] L O Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Citeseer, 1994.

[4] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *SIGARCH Comput. Archit. News*, Vol. 36, pp. 277–286, March 2008.

[5] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, Vol. 23, pp. 24–36, May 1995.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 72–81, New York, NY, USA, 2008. ACM.

[7] Nvidia. Cuda sdk code samples.