View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Waseda University Repository

マルチコアプロセッサ上での
キャッシュ及び
ローカルメモリ管理手法に関する研究

Studies on a Local and Cache Memory Management Scheme on a Multicore Processor

2009 年 2 月

早稲田大学大学院 理工学研究科
情報・ネットワーク専攻
アドバンスト・コンピューティング・システム研究

中 野 啓 史

目次

図目次

表目次

第 1 章

序章

1.1　本研究の背景と目的

半導体微細化技術の向上に伴い，一つのチップ上に集積できるトランジスタ数は年々増加している．一方で，消費電力の増大，命令レベル並列性の限界から，動作周波数の向上やスーパースカラ技術による性能向上は限界を迎えている．そこで，一つの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が実用化されている [PAB+05, YKH+07, IHY+08]．マルチコアは従来よりも粒度の大きな並列性が利用可能であり，これを適切に利用することで，低消費電力化を図ることができる．このマルチコアは今後 PC やサーバ等に留まらず，携帯電話や家庭用ゲーム機，デジタルテレビ等の情報家電分野でも広く利用されるものと思われる．特にリアルタイム性が要求される情報家電分野では，キャッシュメモリの実行時不確定性を避ける目的で，ソフトウェア制御によるローカルメモリを搭載したマルチコアが採用されるものと考えられる．一チップに搭載されるプロセッサコア数の増大に伴い，従来の自動並列化コンパイラが対象としてきたループ並列性を利用するだけでは，予想される理論性能に対して，得られる実効性能が低く，両者の差が拡大してしまう問題が生じている．そこで，ループ並列性に加え，ループやサブルーチン間といった，より粒度の大きな並列性も合わせて利用する粗粒度タスク並列処理が重要となる．

一方で，演算速度とオフチップメモリアクセス速度の差が拡大するメモリウォール問題も性能向上を図る上で大きなボトルネックとなっている．チップとオフチップメモリ間のメモリバンド幅は容易に増やすことができないため，特に一チップ上に複数のプロセッサコアを集積するマルチコアでは，各コアのオフチップメモリアクセスが競合し，この傾向が顕著となる．プロセッサコア数の増大に応じた性能向上を得るためには，プロセッサコア近傍の高速なローカルメモ

リあるいはキャッシュメモリを有効利用し，プロセッサコアへのデータ供給能力を向上させることが重要となる．そのためには，プログラムのデータ局所性を高め，一旦ローカルメモリあるいはキャッシュメモリに載せたデータをなるべく長く使い回すキャッシュ・ローカルメモリ最適化が重要となる．ループ並列性を利用したキャッシュ最適化としては，ループインターチェンジやループフュージョン，ブロッキングなどのループリストラクチャリングやデータレイアウトを変換することにより，キャッシュ利用効率を向上させる手法が研究されている [LCL99, LLL01, LL01, VKB+99].

ローカルメモリ最適化としては以下に挙げる研究が行なわれている．プログラム開始時にローカルメモリに割り当てたデータやプログラムコードを実行終了まで同じアドレス上に保持する静的なローカルメモリ管理 [PDN97, ABC05, SRM06] では，ローカルメモリサイズを超えたデータやプログラムコードを適切に扱うことができない．ループやサブルーチンといった粗粒度タスク間にまたがり，データを動的に入れ出しするローカルメモリ管理が提案されている [UDB06, UB06, LNX07, LWFX07] が，シングルプロセッサを対象としており，マルチプロセッサには適用できない.

また，マルチプロセッサを対象としたコンパイラによるローカルメモリ管理手法として，単一の並列化ループを対象として，ループ並列性を利用したローカルメモリ管理 [KKC+04, IBDD06] では，粗粒度タスク間にまたがるデータローカリティの有効利用を図ることができない．粗粒度タスク間にまたがるデータローカリティの有効利用を図る研究 [KPR+07, RGB+06] では，ユーザによるデータの分割を仮定している．コンパイラにより，自動的にデータの分割を行ない，粗粒度タスク間の並列性を抽出し，データローカリティを生かす手法として，ループ整合分割を用いたデータローカライゼーション手法 [KY98, 吉田 99] が研究されている.

以上のような背景を踏まえて，本研究では，

(1) 自動並列化コンパイラにおける粗粒度タスク並列処理とループ整合分割を利用し，アクセスするデータサイズがキャッシュサイズ以下に分割されたマクロタスクを，データの共有量を考慮してスケジューリングするキャッシュ最適化の有効性について，従来のループ並列性のみを利用した自動並列化コンパイラと比較し検討する．

(2) キャッシュメモリアーキテクチャでは，分割されたマクロタスクのデータサイズがキャッシュサイズを超えていたとしても，ハードウェアがコヒーレンスをとるため，プロセッサからのアクセスに応じ，キャッシュメモリにデータが配置されながら動作することができる．一方で，ローカルメモリアーキテクチャでは，ソフトウェアが全てのデータ配置とデータ転送を管理するため，コンパイラがローカルメモリへ配置できなかったデータへのアクセスは全てオフチップメモリアクセスとなる．粗粒度タスク並列処理において，並列性とデータローカリティを考慮して，複数ループ間のデータ依存を解析し，それらのループ間でデータの授受がローカルメモリを介して行なわれるように分割するループ整合分割手法 [KY98, 吉田 99] が提案されている．このループ整合分割手法はループネストのうち，一重ネストのみを分割する．一重ネストの分割ではローカルメモリに収まらないような，多重にネストしたループ中でアクセスされる多次元配列を効率良くローカルメモリに割り当てるために，複数ネストにわたる分割手法を検討する．さらに，従来のループ整合分割手法が対象としていなかったループ間についてもデータ依存を解析して，プログラムのデータローカリティをより高める分割手法を検討する．

(3) ループ整合分割によるデータローカライゼーション手法を用いたローカル

メモリ管理では，対象となるループ群を分割し，ローカルメモリサイズ以下に分割されたデータを固定的にローカルメモリへ割り当てる．これをスケジューリング結果に基づく，データのアクセスタイミングに応じたより柔軟な管理を行なうことで，ローカルメモリをより有効に利用する手法を検討する．

を目的とする．

1.2 本論文の概要

本論文の第 2 章以降の概要を以下に述べる．

第 2 章「粗粒度タスク並列処理」では，本研究の基本要素である粗粒度タスク並列処理手法について述べる．粗粒度タスク並列処理では，従来のループ並列性に加え，ループやサブルーチンといった処理時間の大きな粗粒度タスク間の並列性を利用する．これにより，プログラムからより多くの並列性が抽出可能となる．

第 3 章「ループ整合分割手法を用いたキャッシュメモリ最適化」では，ループ整合分割を用いて，キャッシュメモリ利用効率を向上させ，粗粒度タスク並列処理性能を向上させる手法について述べる．ループ整合分割手法では，キャッシュメモリサイズを超える同一の配列を定義・参照する複数のループをターゲットループグループとして定義する．ターゲットループグループ中のループ間で，イタレーション間データ依存を解析し，部分ループに分割する．分割されたループのうち同一の部分配列を定義・参照する複数の部分ループは，データローカライゼーショングループと呼ぶタスク（ループ）集合にグループ化される．データローカライゼーショングループ内で定義・参照される部分配列の総サイズが

キャッシュメモリ以下となるように，複数のループにわたり整合して分割することで，キャッシュメモリを介した部分配列の授受を実現する．そして，粗粒度タスクスケジューリングでは，粗粒度タスク間の並列性と割り当てに伴うオフチップ共有メモリとキャッシュメモリ間のデータ転送削減量を考慮しながら，同一データローカライゼーショングループに属する部分ループが，可能な限り同一プロセッサ上で連続的に実行されるようにスケジューリングを行う．このように複数ループにわたる分割とデータ転送削減量を考慮した部分ループの連続実行を行う粗粒度タスクスケジューリングを組み合わせることで，複数の部分ループにわたり，一旦キャッシュメモリに配置された部分配列を追い出すことなく再利用するキャッシュメモリ最適化を行う．キャッシュメモリアーキテクチャである Sun Ultra450（4 プロセッサ）上で行なった性能評価では，Sun Forte HPC6 update1 の最高性能に対して，本手法を用いることにより，SPEC CFP95 ベンチマークの tomcatv で約 2.4 倍，swim で約 4.6 倍の性能向上が得られ，本手法の有効性が確かめられた．

第 4 章「ローカルメモリ最適化」では，第 3 章で述べたループ整合分割手法を拡張したグローバルループ整合分割手法を用いてループを分割し，一旦ローカルメモリに配置した部分配列をなるべく長く使いまわすローカルメモリ最適化について述べる．ローカルメモリ最適化におけるグローバルループ整合分割はループ整合分割を拡張したものであり，多重にネストしたループの分割による多次元配列の複数次元にわたるサイズの減少と，より大きな粒度で配列形状を同一にする分割により，さらなるデータローカリティの有効利用を図る．ローカルメモリはキャッシュメモリと異なり，ソフトウェアにより明示的にデータを配置することが可能なので，コンパイラを用いて，ローカルメモリ上の適切なアドレスに部分配列を割り当てる．分割された部分配列単位でローカルメモリに割り当てることを考えると，部分ループ毎に定義・参照される部分配列の数，サイズそして次

元数が異なり得る．そのため，ローカルメモリの先頭から部分配列を詰め込んでいくとフラグメンテーションが頻繁に発生し得る．そこで，コンパイラはローカルメモリをアプリケーションに適したサイズのブロックと呼ぶ仮想的な連続領域に区切り，各ブロックに部分配列を一つずつ割り当て，フラグメンテーションを回避する．次に，部分配列単位でブロックへ割り当てを行なったとしても，適切なアドレスのブロックに割り当てないと，ローカルメモリ内でのデータの移動やオフチップメモリとローカルメモリ間のデータ転送が頻繁に発生し，却って性能が低下し得る．そのような事態を避けるために，ローカルメモリ最適化においても，予め粗粒度タスクを各プロセッサに割り当てる粗粒度タスクスケジューリングを行うことで，各部分ループ内でアクセスされる部分配列の定義・参照時間を求める．そして，アクセス時間の早い部分配列から順にローカルメモリへ割り当てることで，過去のローカルメモリ割り当て状況に応じた，最もデータ転送時間の短いブロックへの部分配列の割り当てを実現する．また，部分配列の割り当て時に，ブロックが不足した場合は，下記のように解放するブロックを選択する．現在ブロックに割り当てられている部分配列のうち，次回参照されるまでの時間が長い部分配列から順に選択し，その部分配列をオフチップ共有メモリにストアし，当該ブロックを解放する．このように，近い将来参照される部分配列をローカルメモリに残すことで，過去のアクセスパタンだけではなく，将来にわたるアクセスパタンに応じて，ローカルメモリ上のデータ局所性を高めることができる．提案手法をルネサス・日立・早大で開発した，ローカルメモリを持つアーキテクチャである，4 コアが集積された RP1 マルチコアと 8 コアが集積された RP2 マルチコア上で，マルチメディアアプリケーションである AAC エンコーダおよび MPEG2 エンコーダを用いて性能評価した．RP1, 4 コア上では，オフチップ共有メモリにすべての配列を配置した場合に比べ，AAC エンコーダで約 5.0 倍，MPEG2 エンコーダで約 3.8 倍，また RP2, 8 コア上では AAC エ

ンコーダで約 8.4 倍，MPEG2 エンコーダで約 8.4 倍の速度向上がそれぞれ得られ，本手法によるローカルメモリの有効利用が確認された．また，本手法を適用した逐次実行に対し，4 プロセッサ RP1 において，AAC エンコーダで約 3.3 倍，MPEG2 エンコーダで約 2.8 倍の速度向上がそれぞれ得られ，8 プロセッサ RP2 において，AAC エンコーダで約 4.4 倍，MPEG2 エンコーダで約 4.7 倍の速度向上がそれぞれ得られ，また，AAC エンコーダについて，従来のループ整合分割によるデータローカライゼーション手法を用いたローカルメモリ管理手法の処理時間が 2.8 秒であるのに対し，提案したローカルメモリ管理手法を用いることで 1.1 秒と 2.6 倍の，8 コアのマルチコア RP2 上では 1.7 秒が 0.7 秒と，2.5 倍の速度向上がそれぞれ得られた．MPEG2 エンコーダについて，4 コアの RP1 上で，従来手法が 23.2 秒であるのに対し，提案手法を用いることで 20.8 秒と 1.1 倍の，8 コアの RP2 上で，10.6 秒が 8.6 秒と 1.2 倍の速度向上がそれぞれ得られた．これらの性能評価結果より，本手法の有効性が確かめられた．

第 5 章「結論」では，本研究により得られた成果と今後の課題について述べる．

第 2 章

粗粒度タスク並列処理

2.1　まえがき

本章では，本研究の基本要素である粗粒度タスク並列処理手法について述べる．粗粒度タスク並列処理とは，マルチグレイン並列処理における最も粒度の粗いタスク並列処理手法であり，従来のループ並列性に加え，対象プログラムを階層的に疑似代入文ブロック，繰り返しブロックそしてサブルーチンブロックの 3 種類の粗粒度タスク（マクロタスク）に分割し，並列処理を行なう [KWN+05, 白子 04].

2.2　OSCAR コンパイラ処理フロー

粗粒度タスク並列処理を含むマルチグレイン自動並列化を行なう OSCAR 自動並列化コンパイラの処理フローについて説明する．

OSCAR コンパイラの構成を図 2.1 に示す．OSCAR コンパイラはフロントエンド，ミドルパスそして様々なターゲット計算機環境用の複数のバックエンドから構成される．

フロントエンドは FORTRAN あるいは制約付き C（Parallelizable C）[間瀬 06] で記述されたプログラムを読み込み，中間言語を生成する．制約付き C は並列化アプリケーション開発の生産性向上のため，コンパイラが自動で並列化可能な記述を目指した C プログラミング記述のガイドラインである．現状の主な制約として以下が挙げられる．まず，C で記述されたプログラムの解析を困難なものとしているポインタ変数の使用を制限している．例外的に関数の仮引数におけるポインタの使用は認められるが，仮引数ポインタ同士がエイリアスしていてはならない．また，構造体はメンバ変数毎にばらされ，動的に確保されるヒープ領域

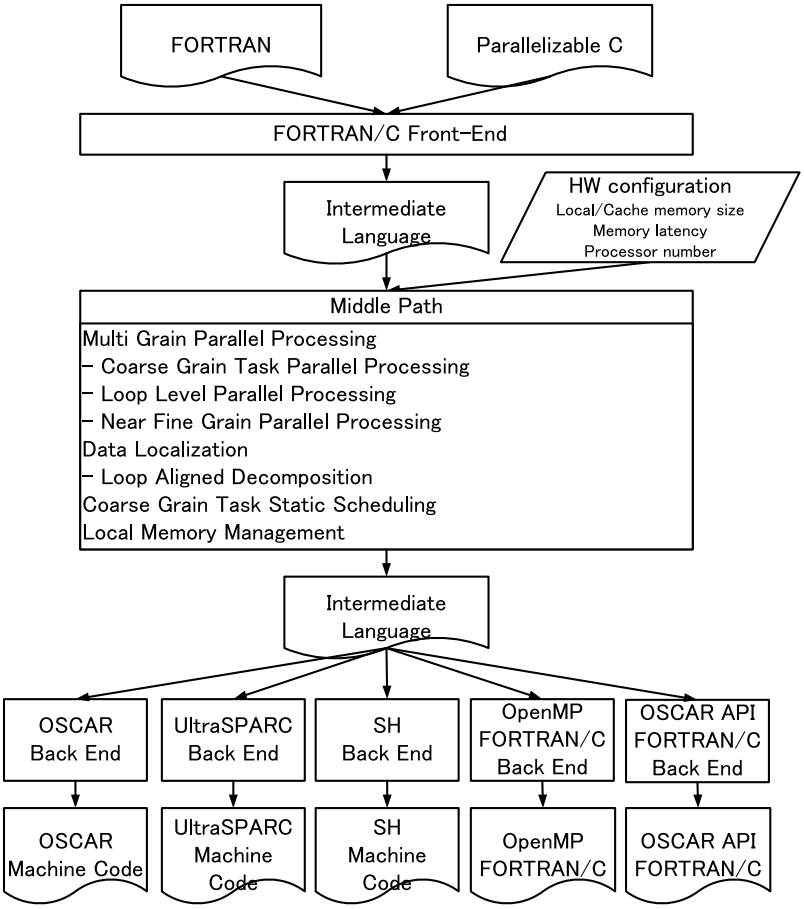


図 2.1 OSCAR コンパイラの構成

は，多次元配列として宣言される．それから，再帰呼び出しの利用も制限されている．

ミドルパスはフロントエンドで生成された中間言語を読み込み，オプションとして入力されるローカルメモリあるいはキャッシュメモリサイズやレイテンシ，プロセッサ数をもとにマルチグレイン自動並列処理およびメモリ最適化を実現する．

ここで，マルチグレイン自動並列処理は，ループやサブルーチンといった粗粒度タスク間の並列性を抽出する粗粒度タスク並列処理，ループイタレーション間

の並列性を抽出するループ並列処理そしてステートメント間の並列性を抽出する近細粒度並列処理を組み合わせ，プログラム全域から並列性を抽出する並列処理手法である．

一方で，メモリ最適化は演算速度とオフチップメモリアクセス速度の差が拡大するメモリウォール問題の打開を目的とする．粗粒度タスク内でアクセスされるデータサイズがローカルメモリあるいはキャッシュメモリサイズ以下となるように粗粒度タスクを分割する．そして，同一のデータにアクセスする粗粒度タスクを同一のプロセッサに連続的に割り当て，一旦プロセッサ近傍の高速なローカルメモリあるいはキャッシュメモリに配置したデータをなるべく長く使い回し，プログラムのデータローカリティを高める．

更に，ソフトウェアにより制御されるローカルメモリでは，解析結果に基づいたデータ転送やメモリ配置をミドルパスで指定する．最後にミドルパスは並列化された中間言語を生成する．

バックエンドは最適化された中間言語を読みとり，アーキテクチャに応じた実行オブジェクトや OpenMP や OSCAR API によって並列化された FORTRAN あるいは C 言語を出力する．OpenMP は共有メモリ型計算機環境における並列処理フレームワークである．OpenMP はプログラム中に挿入されたディレクティブで，並列処理箇所を指定する．OSCAR API はローカルメモリを持つマルチコア上での効率的な並列処理を実現するために，OpenMP のサブセットとなるディレクティブにデータ転送ディレクティブやメモリ配置ディレクティブ等を追加したものである．

2.3 マクロタスクの生成

粗粒度タスク並列処理では，対象プログラム全体（第 0 階層とする）を，疑似代入文ブロック（BPA），繰り返しブロック（RB），サブルーチンブロック（SB）の 3 種類の第 1 階層マクロタスク（MT）に分割する．第 1 階層マクロタスクのうち，繰り返しブロックおよびサブルーチンブロックの内部の第 1 階層において，第 2 階層マクロタスクを定義する．以後，同様に第 i 階層において，第 $(i+1)$ 階層マクロタスクを定義する．

ループ並列処理不可能な実行時間の大きい RB やインライン展開を効果的に適用できない SB に対しては，その内部を階層的に粗粒度タスクに分割して並列処理を行う．

2.4 マクロフローグラフ (MFG) の生成

マクロタスクの生成後，マクロタスク間のコントロールフローとデータ依存を解析し，その結果を表す図 2.2 に示すようなマクロフローグラフ (MFG) を生成する．

図 2.2 の各ノードはマクロタスクを表し，実線エッジはデータ依存を，点線エッジはコントロールフローを表す．また，ノード内の小円は条件分岐を表す．MFG ではエッジの矢印は省略されているが，エッジの方向は下向を仮定している．

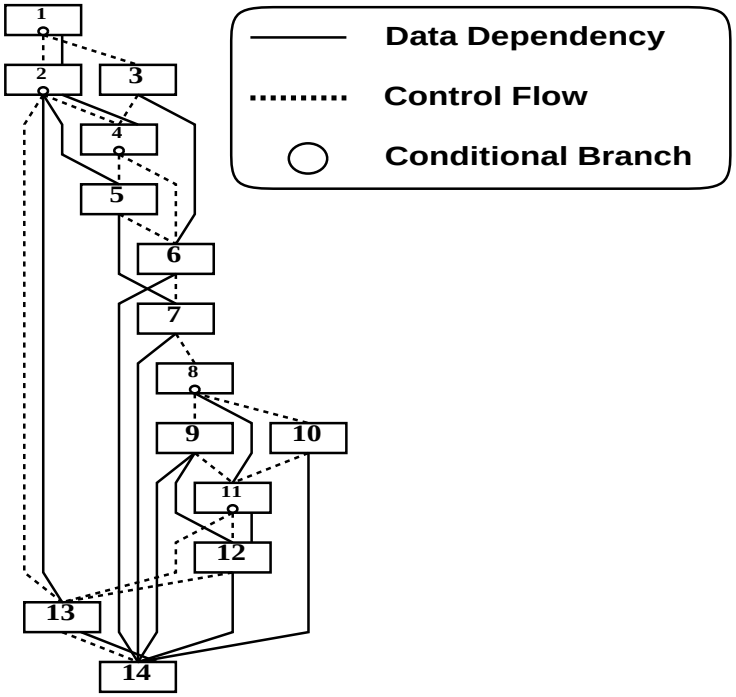


図 2.2　マクロフローグラフの例

2.5　マクロタスクグラフ (MTG) の生成

MFG はマクロタスク間のコントロールフローとデータ依存は表すが，並列性は表していない．並列性を抽出するためには，コントロールフローとデータ依存の両方を考慮した最早実行可能条件解析をマクロフローグラフに対して行う．マクロタスクの最早実行可能条件とは，そのマクロタスクが最も早い時点で実行可能になる条件である．

マクロタスクの最早実行可能条件は図 2.3 に示すようなマクロタスクグラフ (MTG) で表される．

MFG と同様に，MTG におけるノードはマクロタスクを表し，ノード内の小

円はマクロタスク内の条件分岐を表している．実線のエッジはデータ依存を表し，点線のエッジは拡張されたコントロール依存を表す．拡張されたコントロール依存とは，通常のコントロール依存だけでなく，データ依存とコントロールフローを複合的に満足させるため先行ノードが実行されないことを確定する条件分岐を含んでいる．

また，エッジを束ねるアークには 2 つの種類がある．実線アークはアークによって束ねられたエッジが AND 関係にあることを，点線アークは束ねられたエッジが OR 関係にあることを示している．

MTG においてはエッジの矢印は省略されているが，下向きが想定されている．また，矢印を持つエッジはオリジナルのコントロールフローを表す．

2.6 マクロタスク配列範囲解析

粗粒度タスク並列処理におけるデータ依存解析では，マクロタスク間のデータ依存の有無だけではなく，データ依存した配列の範囲について解析することで，データ転送時間の推定や効率的なローカルメモリ管理，またデータ転送命令生成に用いる．マクロタスク配列範囲解析では，マクロタスク毎に確実な定義（Kill），不確実な定義（MayMod），不確実な参照（MayUse），前方露出参照（ExpUse）の四つの配列範囲について解析する．

まず，マクロタスク毎の配列範囲解析について説明する．ここで，n 次元配列 a の配列範囲は以下のように表すものとする．

$$a[l_1 : u_1, ..., l_n : u_n] \tag{2.1}$$

ただし，$l_i, u_i (1 \leq i \leq n)$ は i 次元目の配列範囲の下限値，上限値をそれぞれ表すものとする．

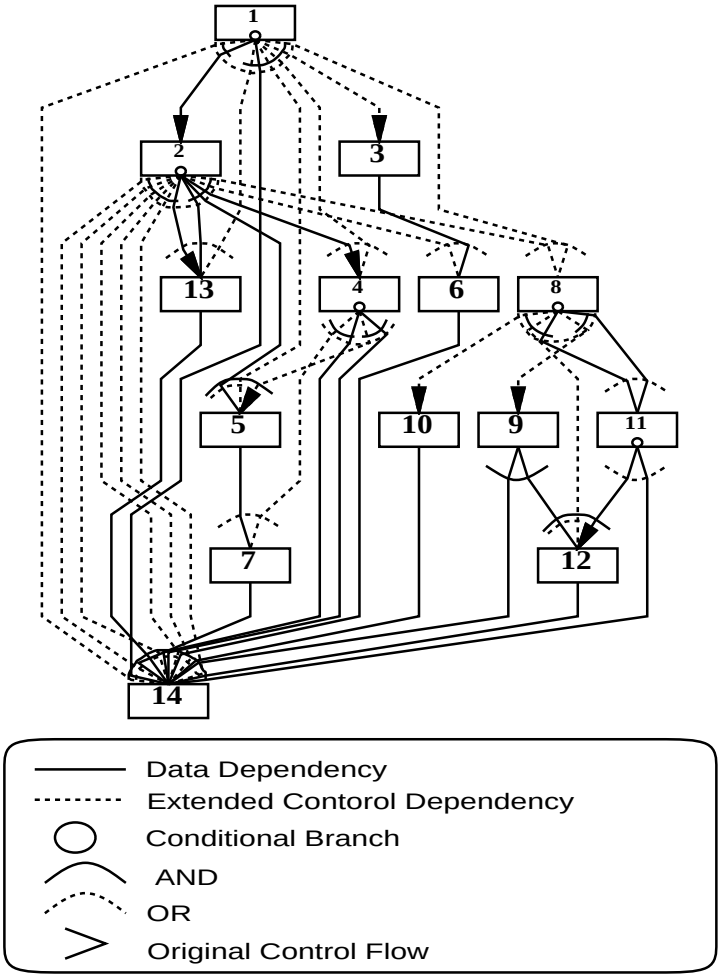


図 2.3　マクロタスクグラフの例

配列範囲同士の演算を次のように定める．積（intersect, ∩）は配列範囲同士の重複する範囲を表す．差（subtract, −）は前者の配列範囲から，後者の範囲を除いた残りの範囲を表す．和（union, ∪）は 2 つの配列範囲を合わせた範囲を表す．

2 次元配列 a の

$$a[l_1^{R1} : u_1^{R1}, l_2^{R1} : u_2^{R1}] \tag{2.2}$$

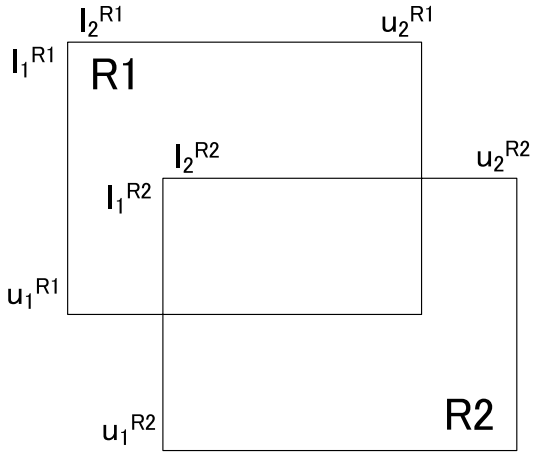


図 2.4　配列範囲

と表される配列範囲を R1 とする．

$$a[l_1^{R2} : u_1^{R2}, l_2^{R2} : u_2^{R2}] \tag{2.3}$$

と表される配列範囲を R2 とする．このとき，R1 と R2 は図 2.4 のように重複しているものとする．配列範囲 R1 と R2 の演算結果は以下のようになる．R1 ∩ R2 を図 2.5 に，R1 − R2 を図 2.6 に，R1 ∪ R2 を図 2.7 にそれぞれ示す．ここで，intersect の結果は範囲一つで，subtract の結果は範囲二つで，union の結果は範囲三つでそれぞれ表現されている．

つづいて，マクロタスクにおいて解析する確実な定義，不確実な定義，不確実な参照そして前方露出参照の四つの配列範囲について，図 2.8 を用いて説明する．

図 2.8 の左側は階層的にマクロタスクに分割されたプログラムコードを，右側は各階層のマクロタスク毎の配列範囲解析結果を表す．解析は深い階層から順に行なわれる．

まず，マクロタスクにおける配列範囲の確実な定義（Kill）とは，マクロタスク内部がどのように実行されたとしても，そのマクロタスク中で必ず定義される配列範囲を表す．このため，条件分岐を含まない基本ブロック中の配列の定義は

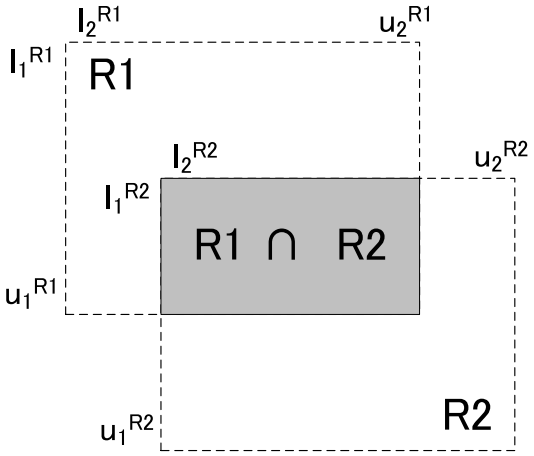


図 2.5　配列範囲同士の積（intersect）

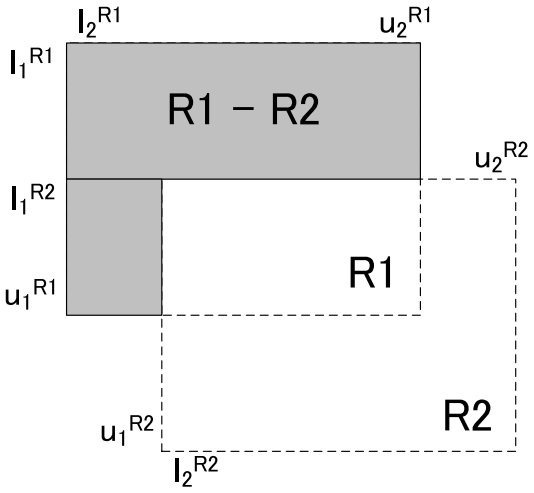


図 2.6　配列範囲同士の差（subtract）

確実な定義となる．たとえば，基本ブロック BB1_1 の c[i:i] や BB1_2_2 の b[i:i, j:j] が該当する．RB1_2 の内側階層のマクロタスク BB1_2_1 は条件分岐の分岐先によらず，必ず実行されるので，配列 a の定義範囲 a[i:i, 0:n] は RB1_2 における確実な定義となる．

マクロタスクにおける配列範囲の不確実な定義（MayMod）とは，そのマクロ

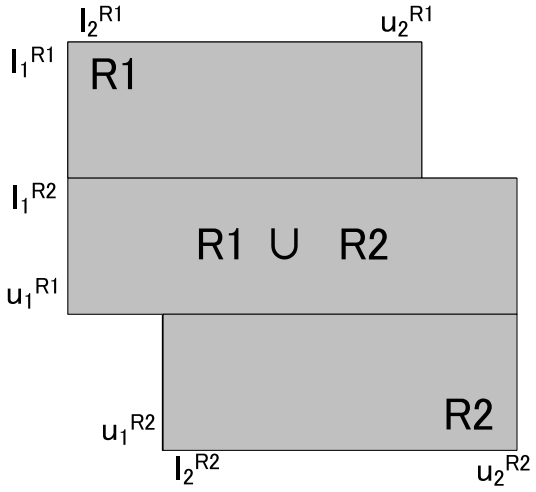


図 2.7 配列範囲同士の和（union）

タスク内部で定義されうる全ての配列範囲を表す．不確実な定義配列範囲には確実な定義の配列範囲も含む．RB1_2 の内側階層のマクロタスク BB1_2_2 は，条件分岐の分岐先によって，実行されるか否かが決まる．このため，BB1_2_2 で定義される配列 b の定義範囲 b[i:i, 0:n] は，RB1_2 において不確実な定義となる．

マクロタスクにおける配列範囲の不確実な参照（MayUse）とは，そのマクロタスク内部で参照されうる全ての配列範囲を表す．条件分岐によらず，全ての参照される配列範囲を含むので，RB1_2 における配列範囲の不確実な参照は a[i:i, 0:n], c[i:i] となる．

マクロタスクにおける配列範囲の前方露出参照（ExpUse）とは，そのマクロタスク内部で確実に定義される前に参照されうる配列範囲を表す．RB1_2 の内側階層において，条件分岐の分岐先によらず，配列 a は確実に定義されるので，BB1_2_2 における配列 a の参照は前方露出参照とはならない．RB1_2 における配列範囲の前方露出参照は c[i:i] となる．

<table>
<tr><th rowspan="2">プログラムコード</th><th colspan="3">配列範囲</th></tr>
<tr><th>layer 3</th><th>layer 2</th><th>layer 1</th></tr>
<tr><td>RB1
for (i = 0; i <= m; i++) {</td><td></td><td></td><td rowspan="4">Kill:
a[0:m, 0:n],
c[0:m]
MayMod:
a[0:m, 0:n],
b[0:m, 0:n],
c[0:m]
MayUse:
a[0:m, 0:n],
c[0:m]</td></tr>
<tr><td>BB1_1
c[i] = ...;</td><td></td><td>Kill, MayMod:
c[i:i]</td></tr>
<tr><td>RB1_2
for (j = 0; j <= n; j++) {</td><td></td><td>Kill:
a[i:i, 0:n]
MayMod:
a[i:i, 0:n],
b[i:i, 0:n]
MayUse:
a[i:i, 0:n],
c[i:i]
ExpUse:
c[i:i]</td></tr>
<tr><td>BB1_2_1
a[i][j] = c[i] + ...;
if (condition)</td><td>Kill, MayMod:
a[i:i, j:j]
MayUse, ExpUse:
c[i:i]</td></tr>
<tr><td>BB1_2_2
b[i][j] = a[i][j]
+ c[i] + ...; }}</td><td>Kill, MayMod:
b[i:i, j:j]
MayUse, ExpUse:
a[i:i, j:j], c[i:i]</td><td></td><td></td></tr>
</table>

図 2.8　マクロタスク配列範囲解析結果

2.7　マクロタスク間配列依存範囲解析

マクロタスク配列範囲解析結果を元にマクロタスク間の配列依存範囲解析を行なう．マクロタスクグラフにおけるデータ依存であるフロー依存配列範囲およびデータ転送量削減に用いられる入力依存配列範囲を計算するものである．

以下に，マクロタスク間の配列依存範囲解析の計算手順を示す．n 個のマクロタスクを持つ解析対象マクロタスクグラフ中のマクロタスクを MT_i とする $(1 \leq i \leq n)$．MT_i における確実な定義を $Kill_i$，不確実な定義を $MayMod_i$，前方露出参照を $ExpUse_i$ とする．

(1) MT_i の配列依存範囲解析を行う．MT_i について，以下のように定義された $Consumption_i$，$Production_i$ を求める．

$$Consumption_i = ExposedUse_i \cup (MayMod_i - Kill_i) \quad (2.4)$$
$$Production_i = MayMod_i \quad (2.5)$$

(2) $Production_i$ を $valid_i$ とする．

(3) MT_i の各後続タスク MT_j に対し，コントロールフロー順に $valid_i$ と $Consumption_j$ と積をとり，重なる配列範囲を MT_i から MT_j への配列フロー依存範囲 $FlowDep_{i-j}$ とする．$valid_i$ と MT_j の $Production_j$ の差をとり，新たな $valid_i$ とする．

$$FlowDep_{i-j} = valid_i \cap Consumption_j \quad (2.6)$$
$$valid_i = valid_i - Production_j \quad (2.7)$$

(4) $valid_i$ の範囲が空になるか階層の最後のマクロタスクまで到達したら，解析を終了する．

(5) MT_i から MT_j，MT_k に対してフロー依存
$FlowDep_{i-j}$，$FlowDep_{i-k}$ が得られた場合，
$FlowDep_{i-j}$ と $FlowDep_{i-k}$ の重なる配列範囲を MT_j から MT_k に対する入力依存範囲 $InputDep_{j-k}$ とする．

$$InputDep_{j-k} = FlowDep_{i-j} \cap FlowDep_{i-k} \quad (2.8)$$

(6) 解析対象の階層の外側ブロックが繰り返しブロックであるとき，階層の最後まで到達した $valid_i$ のうち，次のイタレーションで参照される範囲と重なる部分はイタレーション間でのデータ依存範囲であり，これを $LoopCarriedDep_i$ とする．

マクロタスク間配列依存範囲解析を図 2.9 のコントローフローグラフを用いて説明する．マクロタスク配列範囲解析によって，MT1, MT2, MT3, MT4 につ

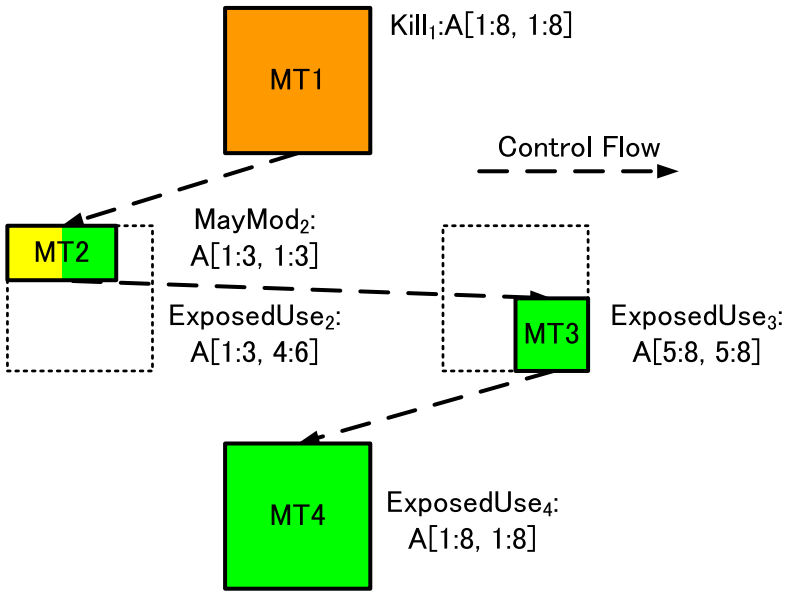


図 2.9　コントロールフローグラフ

いて，図中に示したように，確実な定義，不確実な定義，前方露出参照が求まっているものとする．

式 (2.4), (2.5) に基づき，図中の各マクロタスクについて Consumption と Production を求めた結果を図 2.10 に示す．ここでは，MT_1 を例に説明する．まず，$Production_1$，A[1:8, 1:8] を $valid_1$ とする．コントロールフローに従い，MT_2 の $Consumption_2$ と積を取り，$FlowDep_{1-2}$ は A[1:3, 1:6] と求まる．$valid_1$ から MT_2 の $Production_2$ を引いた範囲を新たに $valid_1$ とする．同様に MT_3 の $Consumption_3$ と積をとり，$FlowDep_{1-3}$ は A[5:8, 5:8] と求まる．$valid_1$ と MT_4 の $Consumption_4$ と積をとり，$FlowDep_{1-4}$ は A[4:8, 1:3] ∪ A[1:8, 4:8] と求まる．最後に $FlowDep_{1-3}$ と $FlowDep_{1-4}$ の積から，$InputDep_{3-4}$ として，A[5:8, 5:8] が求まる．マクロタスク間の配列依存範囲解析結果を図 2.11 に示す．

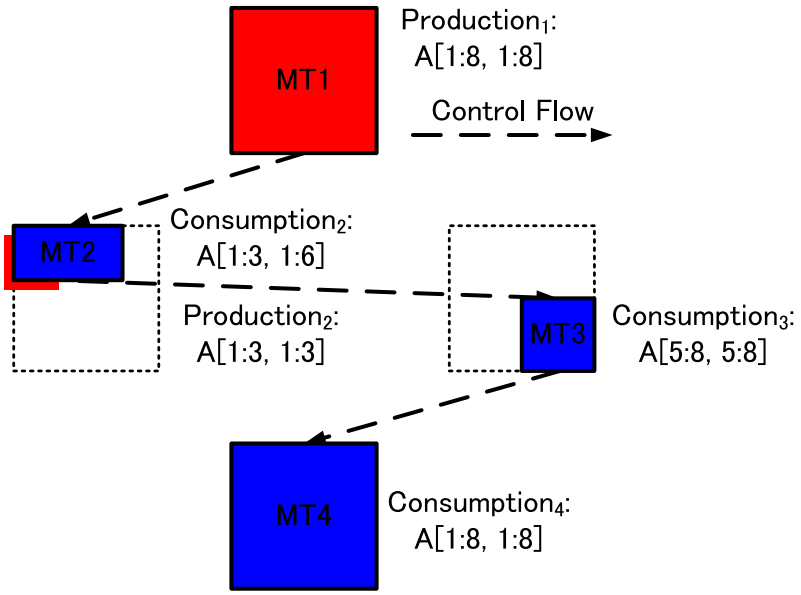


図 2.10　Consumption / Production 範囲

2.8　スケジューリングコードの生成

粗粒度タスク並列処理では，生成されたマクロタスクはプロセッサグループ (PG) に割り当てられて実行される．PG にマクロタスクを割り当てるスケジューリング手法として，コンパイル時に割り当てを決めるスタティックスケジューリングと実行時に割り当てを決めるダイナミックスケジューリングがあり，マクロタスクグラフの形状，実行時不確定性などを元に選択される．

2.8.1　スタティックスケジューリング

スタティックスケジューリングは，マクロタスクグラフがデータ依存エッジのみを持つ場合に適用され，コンパイラがコンパイル時にマクロタスクの PG への割り当てを決定する方式である．スタティックスケジューリングでは，実行時ス

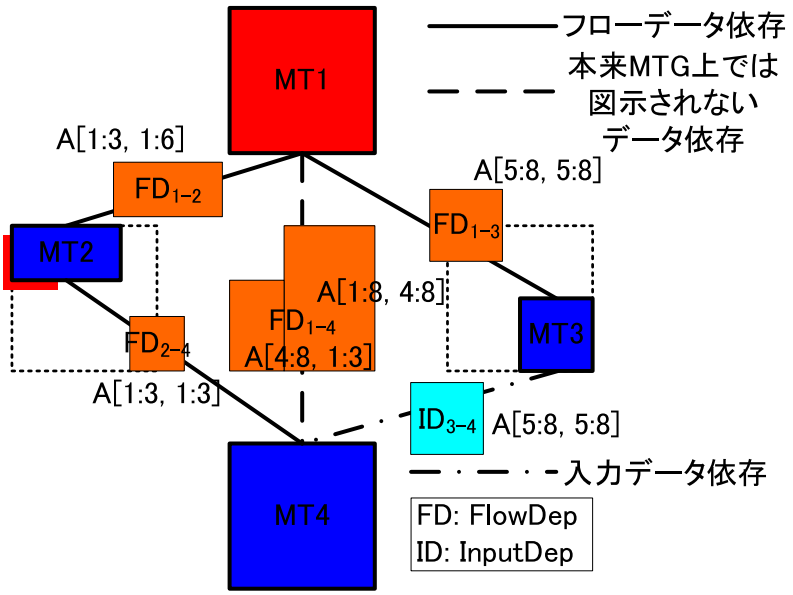


図 2.11　マクロタスク間配列依存範囲解析結果

ケジューリングオーバーヘッドを無くし，データ転送と同期のオーバーヘッドを最小化することが可能である．

本論文で提案するスタティックスケジューリングアルゴリズムの詳細については第 3.3 節で説明する．

2.8.2　ダイナミックスケジューリング

ダイナミックスケジューリングは，条件分岐などの実行時不確定性に対処するために，実行時にマクロタスクの割り当てを決める方式である．

ダイナミックスケジューリングの手法には一つの専用のプロセッサがスケジューリングを行う集中スケジューリング方式と，スケジューリング機能を各プロセッサに分散した分散スケジューリング方式を，使用するプロセッサ台数，システムの同期オーバーヘッドを考慮して使い分けることができる．

2.9 第 2 章のまとめ

本章では，マルチプロセッサシステムの実効性能を高めるための粗粒度タスク並列処理およびマクロタスク間のデータ転送時間の推定や効率的なローカルメモリ管理，そしてデータ転送命令生成に用いるマクロタスク間配列依存範囲解析について説明した．

粗粒度タスク並列処理では，従来のコンパイラで行なわれてきたループ並列性の利用に加え，ループやサブルーチン等の粗粒度タスク間の並列性を利用することができる．

第 3 章

ループ整合分割手法を用いたキャッシュメモリ最適化

3.1　まえがき

本章では，ループ整合分割を用いて，キャッシュメモリ利用効率を向上させ，粗粒度タスク並列処理性能を向上させる手法について述べる．コンパイラによるキャッシュメモリ最適化では，ループインターチェンジやディストリビューション，ブロッキングなどのループリストラクチャリングやデータレイアウトを変換することにより，単一のループに対して，キャッシュ利用効率を向上させる手法が研究されている [LCL99, LLL01, LL01, VKB+99].

ループ整合分割手法は，複数ループにわたり分割を行なうことで，粗粒度タスク間の並列性利用し，データローカリティを高める点に特徴を持つ．ループ整合分割では，キャッシュメモリサイズを超える同一の配列を定義・参照する複数のループをターゲットループグループとして定義する．ターゲットループグループ中のループ間で，イタレーション間データ依存を解析し，部分ループに分割する．分割されたループのうち同一の部分配列を定義・参照する複数の部分ループは，データローカライゼーショングループと呼ぶタスク（ループ）集合にグループ化される．データローカライゼーショングループ内で定義・参照される部分配列の総サイズがキャッシュメモリ以下となるように，複数のループにわたり整合して分割することで，キャッシュメモリを介した部分配列の授受を実現する．そして，粗粒度タスクスケジューリングでは，粗粒度タスク間の並列性と割り当てに伴うオフチップ共有メモリとキャッシュメモリ間のデータ転送削減量を考慮しながら，同一データローカライゼーショングループに属する部分ループが，可能な限り同一プロセッサ上で連続的に実行されるようにスケジューリングを行う．このように複数ループにわたる分割とデータ転送削減量を考慮した部分ループの連続実行を行う粗粒度タスクスケジューリングを組み合わせることで，複数の部

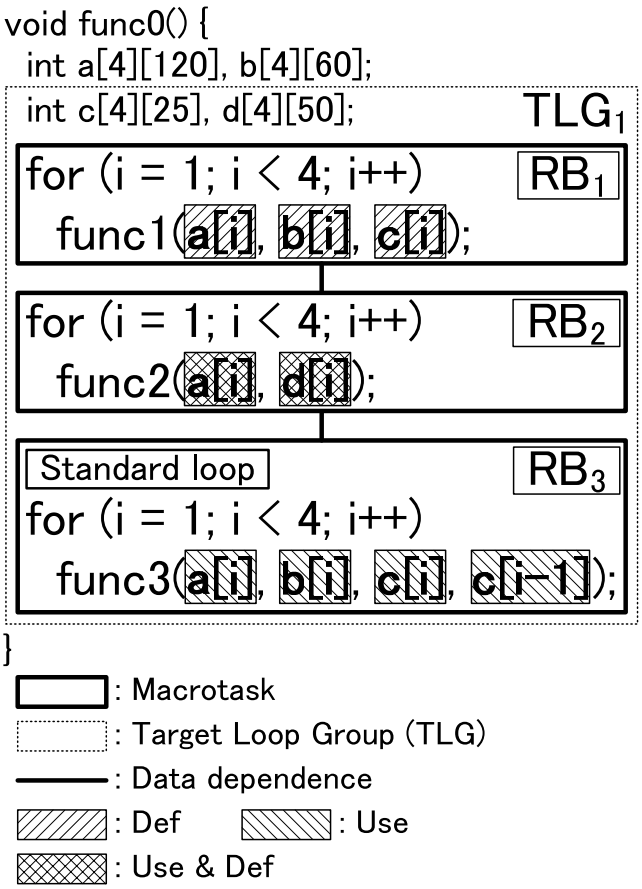


図 3.1　マクロタスクグラフ

分ループにわたり，一旦キャッシュメモリに配置された部分配列を追い出すことなく再利用するキャッシュメモリ最適化を行う．

3.2　ループ整合分割

ループ整合分割 [KY98, 吉田 99] は，ループやサブルーチンといったマクロタスク間の並列性を抽出し，マクロタスク間でデータローカリティを活用する手法である．

図 3.1 のマクロタスクグラフを用いてループ整合分割について説明する．図中の RB_1, RB_2, RB_3 はそれぞれマクロタスクを表す．マクロタスク間の実線はマクロタスク同士が互いにデータ依存していることを表す．ループ整合分割では，同一の配列を定義・参照し，データ依存するループ（RB）を集め，これらのループ群をターゲットループグループ（Target Loop Group: TLG）に選択する．ターゲットループグループ中で最も推定コストの大きなループを標準ループ（Standard loop）と定める．標準ループとターゲットループグループ中の他のループ間で，どのイタレーション同士が依存するかを解析する．この解析をループ間データ依存解析と呼ぶ.

図中では，RB_1, RB_2, RB_3 がターゲットループグループ TLG_1 に選択されている．各関数呼び出し先では，引数である各配列の二次元目全体が，図中に示した網かけの通り，定義あるいは参照されるものとする．たとえば，関数 $func1$ の引数配列 a は，関数呼出先で a[i:i, 0:119] にあたる範囲が定義される．ただし，関数 $func2$ の引数配列 a は，定義の前に参照される前方露出参照とする．また，標準ループ RB_3 の $i = K$ のイタレーションと RB_1 の $i = K, i = K - 1$，RB_2 の $i = K$ のイタレーションがそれぞれループ間でデータ依存していると解析されるものとする．

解析後，ターゲットループグループ中のループを分割する．そして，多量のデータを共有する，分割された部分ループ群を，同一プロセッサに割り当てるべきマクロタスク集合，データローカライゼーショングループ（DLG）として定義する．TLG_1 を 3 で分割した場合のマクロタスクグラフを図 3.2 に示す．図中には DLG_1, DLG_2, DLG_3 の 3 つのデータローカライゼーショングループが定義されており，$DLG_m (1 \leq m \leq 3)$ 中には RB_1^m, RB_2^m, RB_3^m がそれぞれ含まれる．また，ループ間データ依存解析結果に従い，たとえば RB_3^2 は RB_2^2 と RB_1^1 にデータ依存している．

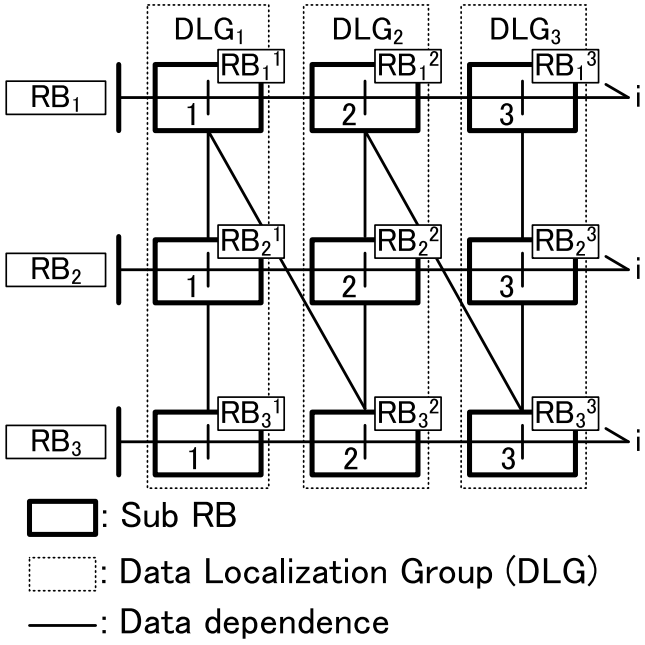


図 3.2　データローカライゼーショングループ

3.3　キャッシュ最適化を考慮したスケジューリングアルゴリズム

データを共有している二つのマクロタスクを異なるプロセッサグループへ割り当てた場合，それらのプロセッサグループ間でデータ転送が必要になる．データ転送を最小化するためには，あるプロセッサグループ PG_i に既に割り当てられたマクロタスク MT_j とデータ共有量の多いマクロタスク MT_k は並列性を損なわない範囲で同一のプロセッサグループ PG_i に割り当てる必要がある．そこで，あるプロセッサグループとマクロタスクの組合わせ毎に次のようにデータ転送ゲインを定義し，プロセッサグループ間でのデータ転送の最小化を行う．ここで，プロセッサグループ PG_i とマクロタスク MT_j のデータ転送ゲイン $Gain_{ij}$

は，PG_i に既に割り当てられたマクロタスクと MT_j とのデータ共有量，及び PG_i への割り当てが決まっているデータローカライゼーショングループに属するマクロタスクと MT_j とのデータ共有量の和と定義する．

以上の前提を元にして，本論文で評価する DLG を考慮したスケジューリングアルゴリズムデータ転送ゲイン/CP/MISF スケジューリング法 [NIO+02, NIO+03] は以下に示す DLG 外マクロタスク割り当てフェーズ（Phase1）と DLG マクロタスク割り当てフェーズ（Phase2）を全てのマクロタスクを割り当てるまで繰り返すことで行う．ただし，DLG_MT とはデータローカライゼーショングループに属するマクロタスク，NOT_DLG_MT とは DLG_MT 以外のマクロタスクとする．

3.3.1 DLG 外マクロタスク割り当てフェーズ（Phase1）

Phase1 では NOT_DLG_MT のみを DT-Gain/CP/MISF のプライオリティに従い割り当てる．Phase1 では NOT_DLG_MT のみをレディタスクとして扱う．

ここで，DT-Gain/CP/MISF 法の手順は以下のようになる．

> 手順 1 あるスケジューリング時点において，レディタスクとアイドルプロセッサグループとの組合わせのうち最大のデータ転送ゲイン $Gain_{ij}$ を与えるマクロタスク MT_j のプロセッサグループ PG_i への割り当てを 1 つ選ぶ．その際，同一のデータ転送ゲインを与える組合わせが 2 つ以上あれば CP/MISF [笠原 91] のプライオリティに従い選択する．任意のマクロタスク MT_k と PG_i とのデータ転送ゲイン $Gain_{ik}$ に MT_j と MT_k のデータ共有量を加算する．
>
> もしレディタスク，もしくはアイドルプロセッサがなければ手順 2 へ，も

しあれば手順 1 を繰り返す.

手順 2 少なくとも 1 つのプロセッサグループがマクロタスクの実行を終了する，次のスケジューリング時点を計算し，そのときのレディタスクを見つける．もしレディタスクがあれば手順 1 へ行き，なければ手順 2 を次のスケジューリング時点がなくなるまで繰り返し，なくなったら Phase1 を終了する.

3.3.2 DLG マクロタスク割り当てフェーズ（Phase2）

Phase2 では DLG_MT のみを以下の手順に従い割り当てる．Phase2 では DLG_MT のみをレディタスクとして扱う.

手順 1 PG_i に既に割り当てられたマクロタスクと同一の DLG に属するマクロタスクがレディタスク中にあれば，手順 2 を，なければ手順 3 を選択する.

手順 2 レディタスク中に存在する DLG の内， PG_i に最後に割り当てられた DLG_MT と同一の DLG_j に属する $DLG_j\_MT_k$ を割り当てる．手順 4 に行く.

手順 3 CP/MISF [笠原 91] のプライオリティに従い，レディタスク中の MT_k を割り当てる．手順 4 に行く.

手順 4 任意のマクロタスク MT_l と割り当て先プロセッサグループ PG_i とのデータ転送ゲイン $Gain_{il}$ に MT_k と MT_l のデータ共有量を加算する.

もしレディタスクがあれば手順 1 へ行き，なければ Phase2 を終了する.

3.4 粗粒度タスク並列処理とキャッシュ最適化の性能評価

本節では粗粒度タスク並列処理とキャッシュ最適化を適用した場合の性能評価を行なう．そして，ループ並列性のみを利用し，粗粒度タスク間でのキャッシュ最適化を行なわない市販の Sun Forte コンパイラと比較を行なった結果について述べる．

3.5 評価環境

本節では本手法の評価に用いたマルチプロセッサシステム Sun Ultra80 とそのコンパイラ及びベンチマークについて述べる．性能評価に使用した 4 プロセッサの SMP Sun Ultra80 のスペック及び使用した Forte ループ自動並列化コンパイラの諸元を表 3.1 に示す．また，評価を行ったときの Forte コンパイラのコンパイルオプションを表 3.2 に示す．表中， Forte とは Forte コンパイラのみによるコンパイルに用いたオプション，OSCAR とは本手法を適用後のプログラムのコンパイルに用いたオプションをそれぞれ意味する．評価ベンチマークには，SPEC 95fp の swim 及び tomcatv を用いた． swim は有限差分近似を用いた shallow water 方程式の求解プログラム，tomcatv はベクトル化メッシュ生成プログラムである．入力データとしては ref を用いた．

本手法の評価のために各プログラムのソースに以下のような改変を加えた．swim は CALC1, CALC2, CALC3 という三つのサブルーチンが実行時間の大半を占める．これらの異なるサブルーチン内部のループ間同士でデータ共有量が大きい．これら三つのサブルーチンをインライン展開後，一つのサブルーチンと

表 3.1 Ultra80 の仕様

Vender	Sun Microsystems
CPU	450MHz UltraSPARC-II 4 台からなる SMP
L1 命令 キャッシュ	16Kbyte Pseudo 2-Way Set Associative line size: 32byte
L1 データ キャッシュ	16Kbyte, Direct-Map line size: 32byte (two 16byte sub-blocks) write-through non-allocating
L2 ユニファイド キャッシュ	4Mbyte, Direct-Map line size: 64byte write-back, allocating
メインメモリ	1024Mbyte
OS	Solaris8
コンパイラ	Forte[tm] HPC 6 update 1

し，新たにできたサブルーチンに対して本手法を適用した． tomcatv は実行時に使用配列サイズをファイルから読み込むが，スタティックスケジューリングによるキャッシュ最適化を考えているので，コンパイル時にデータサイズが確定している必要がある．そこで，ref データセットの使用配列サイズをプログラム中に明記した．また，データレイアウトの変換を行った．

3.6 評価結果

swim を用いた性能評価結果を図 3.3 に， tomcatv を用いた性能評価結果を図 3.4 にそれぞれ示す．なお，プロセッサ台数が 4 台と少ないので 1 プロセッ

表 3.2　Forte コンパイラのコンパイルオプション

	シングル プロセッサ用	マルチ プロセッサ用
Forte	-fast	-fast -parallel -reduction -stackvar
OSCAR		-fast -explicitpar -mp=openmp

サエレメントを 1 プロセッサグループとした．図中，速度向上率とは “ Forte のみでコンパイルしたプログラムの 1 プロセッサでの実行時間/本手法を適用したプログラムの実行時間 × 100 [%]” である．計測はそれぞれ 5 回ずつ行い，最速値を計測値とした．

図 3.3 において，本手法を適用したプログラムの実行時間は 1 プロセッサ時 81.3 秒，2 プロセッサ時 47.9 秒，3 プロセッサ時 27.8 秒，4 プロセッサ時 13.2 秒となっている．2, 3, 4 プロセッサ時の実行時間は 1 プロセッサ時の実行時間に対し，それぞれ 1.70 倍，2.92 倍，6.16 倍とプロセッサ台数に対しスケーラブルな速度向上を示している．また，4 プロセッサ時の性能評価結果はスーパーリニアとなっている．まず， swim で使われている配列の総サイズは約 13MB である．一方，4 プロセッサ時の L2 キャッシュの総容量は 16MB となる．このため，配列がほとんど L2 キャッシュに収まり，キャッシュの有効利用とそれに

伴うパイプラインストールの減少により，このような結果が得られたと考えられる．

図 3.4 において，本手法を適用したプログラムの実行時間は 1 プロセッサ時 101.1 秒，2 プロセッサ時 64.0 秒，3 プロセッサ時 50.9 秒，4 プロセッサ時 33.1 秒となっている．2, 3, 4 プロセッサ時の実行時間は 1 プロセッサ時の実行時間に対し，それぞれ 1.58 倍，1.98 倍，3.05 倍とプロセッサ台数に対しスケーラブルな速度向上を示している．

次に本手法を適用したプログラムの実行時間と Forte のみでコンパイルしたプログラムの実行時間を比較する． Forte のみの場合プロセッサ台数に対して速度向上があまり見られないのに対して，本手法はスケーラブルな速度向上を示している．また，いずれのベンチマーク，いずれのプロセッサ台数に対しても本手法は Forte よりも高い速度向上を示している．図 3.3 において，4 プロセッサ時，Forte のみでコンパイルしたプログラムの実行時間が 60.2 秒であるのに対し, 本手法を適用したプログラムの実行時間は 13.2 秒となり，4.56 倍の速度向上を得た．また，図 3.4 において，4 プロセッサ時， Forte のみでコンパイルしたプログラムの実行時間が 78.6 秒であるのに対し，本手法を適用したプログラムの実行時間は 33.1 秒となり，2.37 倍の速度向上を得た．

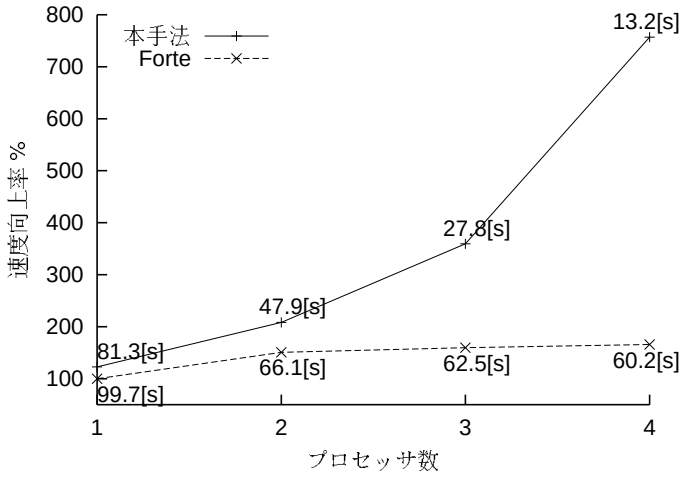


図 3.3　swim の速度向上率

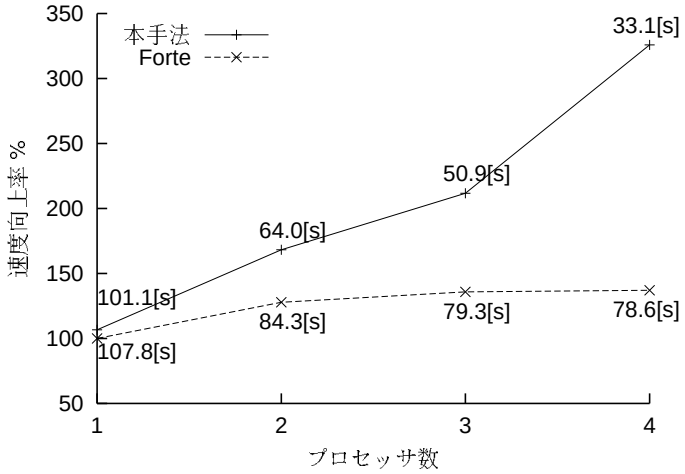


図 3.4　tomcatv の速度向上率

3.7　第 3 章のまとめ

近年マルチプロセッサシステムにおいて，演算速度とオフチップメモリアクセス速度の差が拡大するメモリウォール問題も性能向上を図る上で大きなボトル

ネックの差の拡大が問題となっている．これを解決するためには，プロセッサ近傍の高速なキャッシュメモリを有効利用し，プログラムの持つデータローカリティを最大限引き出す必要がある．本章では，粗粒度タスク並列処理において，ループ整合分割を用いてマクロタスクをキャッシュサイズに収まるサイズに分割し，分割されたマクロタスク同士のデータ共有量を考慮した粗粒度タスクスタティックスケジューリングを行なうキャッシュ最適化手法について述べた．Sun Ultra80（4 プロセッサ）上での性能評価の結果，SPEC CFP95 ベンチマークの swim について，Sun Forte HPC 6 update 1 コンパイラの最小処理時間が 60.2 秒であるのに対し，本手法を用いることで 13.2 秒と 4.6 倍の速度向上を得た．同じく tomcatv について，Sun Forte コンパイラが 78.6 秒であるのに対し，本手法を用いることで 33.1 秒と 2.4 倍の速度向上を得た．これらの性能向上から，粗粒度タスク並列性の利用と粗粒度タスク間のキャッシュ最適化の有効性が確かめられた．

第 4 章

ローカルメモリ最適化

4.1　はじめに

リアルタイム性の要求される組み込みシステムにおいては，キャッシュミスヒット時の実行時不確定性を避けつつ，メモリウォール問題を打開する目的で，ソニー，東芝，IBM の Cell[PAB+05]，ルネサス，日立，早稲田の RP1[YKH+07],RP2[IHY+08] のように，キャッシュとは別に，ローカルメモリを搭載したマルチコアが開発されている．今後，情報家電，自動車のような組み込み用途として，要求されるリアルタイム処理性能向上と消費電力低減を両立するために，ローカルメモリを搭載した組み込み用マルチコアが採用されると予測される．

しかしながら，プログラマが手動で容量制約のあるローカルメモリを活用する並列化プログラムを開発することは非常に困難である．そこで，コンパイラによる各プロセッサへの負荷分散とローカルメモリへのデータ割り当てを自動的に行なうローカルメモリ最適化コンパイル手法が望まれる．

ループ並列性を利用し，各並列化ループが終了する度に，オフチップメモリにデータを書き戻すというコンパイラによるローカルメモリ管理手法 [KKC+04, IBDD06] が提案されている．しかしながら，マルチコアに搭載されるコア数の増大に伴い，ループ並列性のみの利用では，予想される理論性能と得られる実効性能の差が拡大してしまう．そのため，ループ並列性に加え，ループやサブルーチン間といったより粒度の大きな並列性を利用する粗粒度タスク並列処理が重要となる．また，メモリウォール問題の深刻化に伴い，一旦ローカルメモリに配置したデータは，ループ毎にオフチップメモリに書き戻すのではなく，なるべく長時間ローカルメモリに保持することが望ましい．そのため，粗粒度タスク間にわたるデータローカリティ利用が必須となる．

ループやサブルーチンといった粗粒度タスク間の並列性を利用し，粗粒度タスク間でデータローカリティを活用する手法として，ループ整合分割を用いたデータローカライゼーション手法 [KY98, 吉田 99] が提案されている．

データローカライゼーション手法によるローカルメモリ管理では，単一のターゲットループグループを対象として，一重ネストの分割のみを行なっていた．このため，更なる性能向上を得るためには，異なるターゲットループグループ間で分割された部分配列を同一の形状に揃えることで，より多くのデータローカリティを利用したり，多重ネストにわたる分割を行ない，多次元配列を適切なサイズに縮小する必要がある．

さらに，データローカライゼーション手法では，データ依存したマクロタスクを分割し，分割された部分ループでアクセスされる部分配列をローカルメモリ上の領域に固定的に割り当てる．限られたローカルメモリをより効率的に利用するためには，部分配列をローカルメモリ上へ固定的に割り当てるのではなく，粗粒度タスク間にわたるデータの使用状況とローカルメモリの使用状況に応じた柔軟な割り当てを行なうローカルメモリ管理が重要となる．

4.2 ループ整合分割を用いたデータローカライゼーション手法

最初に 3.2 節で説明したループ整合分割を用いたデータローカライゼーション手法によるローカルメモリ管理について説明する．データローカライゼーション手法では，同一ターゲットループグループ中の全てのマクロタスクについて，分割された部分配列を固定的にローカルメモリに割り当てる．このため，ターゲットループグループ中で一時的にしかアクセスされないような配列用のローカルメモリ上の領域は，実行時間の大半において利用されない．また，ターゲットルー

プグループ単位でのローカルメモリ管理を行なうため，ターゲットループグループ終了時点で，ローカルメモリ上に割り当てられている配列範囲は，異なるターゲットループグループで参照されていたとしても，ローカルメモリから解放される．多重にネストしたループで，多次元配列にアクセスするような場合，一重ネストの分割では，分割後の部分ループで定義あるいは参照される配列サイズをローカルメモリサイズに比べ十分縮小できないことが考えられる．

ループ整合分割を用いたデータローカライゼーション手法によるローカルメモリ管理について，具体的に図 4.1 のマクロタスクグラフを 1024bytes のローカルメモリに割り当てた場合を想定し説明する．

図 4.1 では，4 つのループ RB1, RB2, RB3 そして RB4 が互いにデータ依存している．各ループの内側階層にはそれぞれ RB5, RB6, RB7, RB8 が存在しており，いずれも 2 重ループを形成している．これらのうち，RB1, RB5, RB4, RB8 は並列化ループであるものとする．RB2, RB6, RB3, RB7 はシーケンシャルループであるものとする．また，RB2 の 2 重ループは，ループインデクス (i, j) の組を (0, 1), (0, 2), ..., (2, 119) という順でアクセスする．一方，RB3 の 2 重ループは (2, 118), (2, 117), ..., (0, 0) という順でアクセスする．図中の左辺に現れる配列は，そのマクロタスク内で定義されることを，右辺に現れる配列は参照されることをそれぞれ表す．このマクロタスクグラフにループ整合分割を適用すると，RB2, RB3 の回転方向が異なるため，RB1, RB2 を含む TLG1 と RB3, RB4 を含む TLG2 の二つのターゲットループグループが生成される．図 4.2 にループ整合分割によって生成された TLG1 と TLG2 を示す．

TLG1 中の RB1, RB2 に注目すると，配列 a, b, c がそれぞれ 1 イタレーションあたり定義あるいは参照されるサイズはそれぞれ 480bytes となり，合計すると 1440bytes となる．データローカライゼーション手法では，同一ターゲットループグループ中の全てのマクロタスクについて，分割された部分配列を固定的

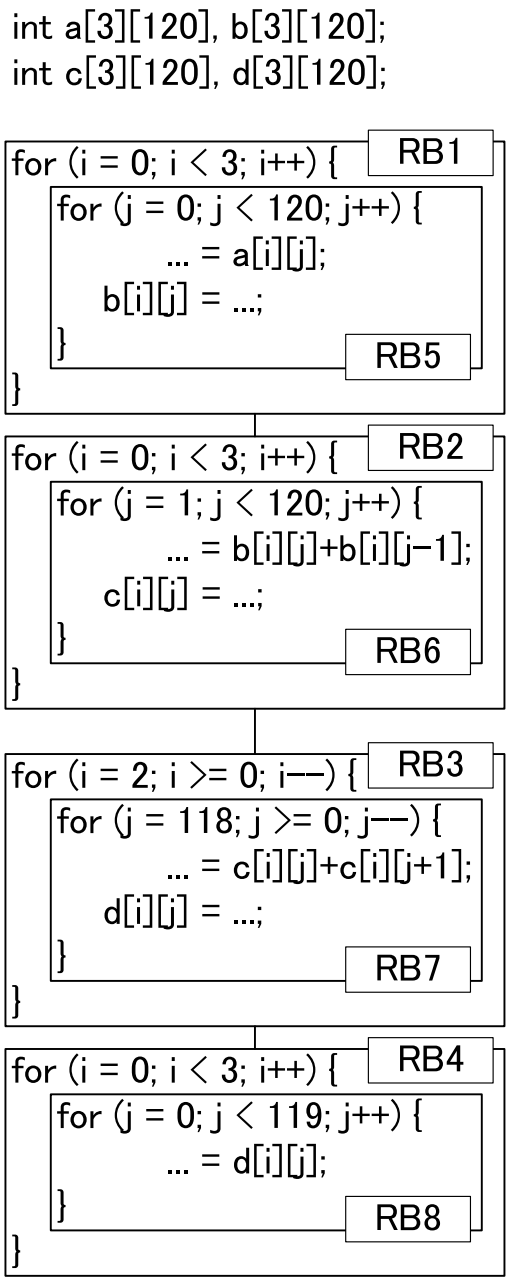


図 4.1　マクロタスクグラフ

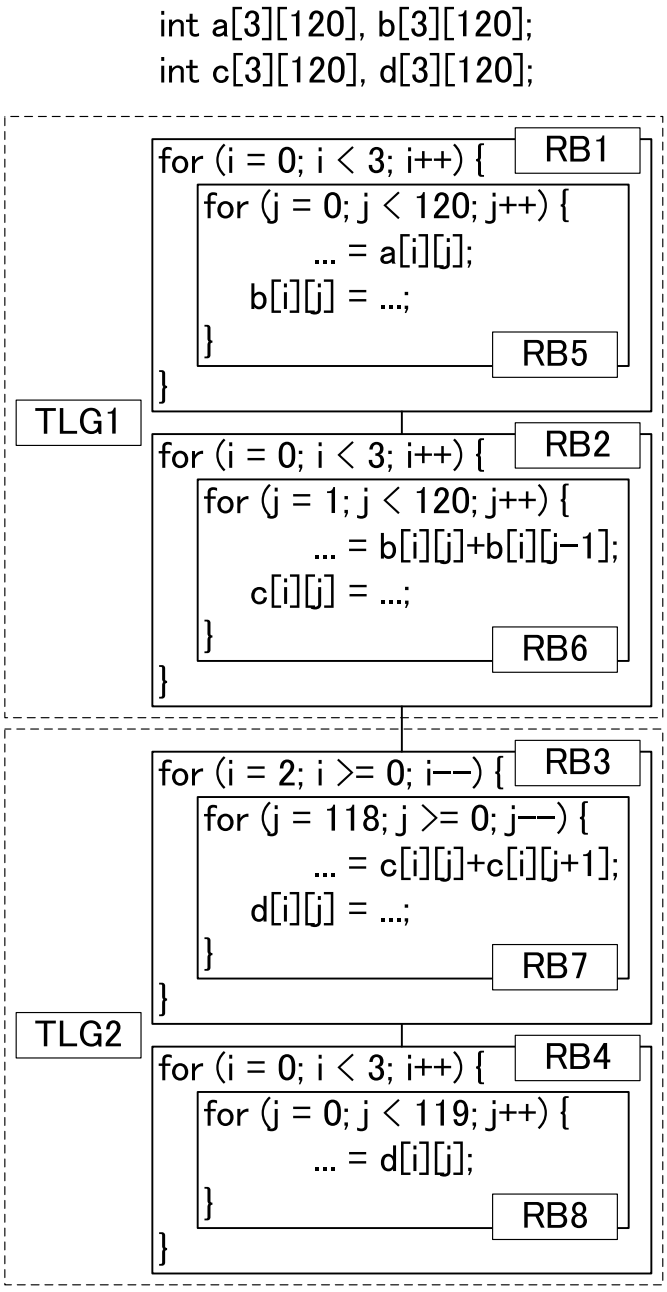


図 4.2　ループ整合分割におけるターゲットループグループ

にローカルメモリに割り当てる．そのため，TLG1 を 3 で最大限分割したとしても，必要なローカルメモリサイズは 1440bytes となり，1024bytes のローカルメモリには，全ての配列を割り当てることができない．このような場合，アクセス回数とデータ転送をもとに，ローカルメモリ割り当て利得の大きな配列から順にローカルメモリへ割り当てる．ここでは，配列 b, c が割り当て対象に選択されたものとする．同様に TLG2 も内部で定義あるいは参照される配列サイズから，3 で分割される．

TLG1, TLG2 をそれぞれ 3 で分割した時のマクロタスクグラフを図 4.3 に示す．図中では，TLG1, TLG2 毎に DLG1, DLG2, DLG3 がそれぞれ定義されている．TLG1 および TLG2 の DLG_m 中の分割された部分ループ RB1_m, RB2_m, RB3_m, RB4_m ($1 \leq m \leq 3$) を 1024bytes のローカルメモリに割り当てた場合を図 4.4 に示す．固定的な割り当てを行なうため，RB1_m の c[m-1:m-1, 0:119] のように未使用な領域があるにも関わらず，配列 a[m-1:m-1, 0:119] を割り当てることができない．また，ターゲットループグループ単位でのローカルメモリ管理を行なうため，RB2_3 の終了時点でローカルメモリ上に生きて残っている c[2:2, 0:119] はオフチップメモリに一旦ストアされる．そして，RB3_3 の開始時点で，同じ範囲をオフチップメモリからローカルメモリにロードする．

ローカルメモリサイズが半分の 512bytes である場合を想定してみると，ループ整合分割を用いたデータローカライゼーション手法では，ローカルメモリに割り当て可能な配列数は 1 つとなり，TLG1 では配列 b が，TLG2 では配列 d が選択されるものとしたときの割り当ての様子を図 4.5 に示す．

4.3　ローカルメモリ管理概要

本節では提案するローカルメモリ管理の概要について述べる．まず，2 章で説明した粗粒度タスク並列処理により，ループやサブルーチンといった粗粒度タスク間の並列性を抽出する．4.2 節で説明したループ整合分割を用いたデータローカライゼーションによるローカルメモリ管理に対し，更にデータローカリティを高め，データ転送を削減し，性能向上を得るにためには，より大域的なマクロタスク分割と，より柔軟なローカルメモリ管理が必要となる．

より大域的なマクロタスク分割を行なうために，ループ整合分割を拡張したグローバルループ整合分割を用いる．グローバルループ整合分割について，4.6 節で述べる．柔軟なローカルメモリ管理を行なう，スケジューリング時刻に基づいたローカルメモリ割り当てと解放について 4.8 節で説明する．

4.4　OSCAR マルチコアアーキテクチャ

本節では, 提案手法が想定する OSCAR マルチコアアーキテクチャ [KWN+05] について説明する．OSCAR マルチコアアーキテクチャは自動マルチグレイン並列化コンパイラとの協調動作により，実効性能が高く価格性能比のよいコンピュータシステムの実現を目指したコンパイラ協調型アーキテクチャである．

OSCAR マルチコアアーキテクチャを図 4.6 に示す．OSCAR マルチコアは 1 つのチップ上に複数のプロセッサエレメント（PE）を持つ．各 PE は単純な一命令発行の in-order プロセッサコア，1 ポートのローカルデータメモリ（LDM），2 ポートの分散共有メモリ（DSM），プログラムコードを保持するローカルプログラムメモリ（LPM）あるいは命令キャッシュ（I-$）を持つ．そして CPU と非

同期にバースト転送が可能なデータ転送ユニット（DTU）を持つ．チップ上の全ての PE とオンチップ集中共有メモリ（OnChipCSM）はバスやクロスバといったフラットな Interconnection Network によって接続されている．さらにオフチップ集中共有メモリ (OffChipCSM) がチップ外に接続されている．ここで，提案手法はローカルデータメモリを対象としてローカルメモリ管理を行なう．

複数の PE からアクセス可能な DSM, OnChipCSM そして OffChipCSM は，マルチグレイン並列処理に必要な様々な粒度の通信をサポートするために，それぞれ以下のように領域が配置される．DSM 上には粗粒度タスク並列処理におけるスタティックスケジューリング時のタスク終了フラグやダイナミックスケジューリング時のタスク開始・終了フラグそして条件分岐信号といった小規模な通信用領域を配置する．OnChipCSM 上には粗粒度タスク並列処理における分散ダイナミックスケジューリング時のレディタスクキューや最早実行可能条件といった領域を配置する．また，OffChipCSM はプログラム中のデータに対し，十分大きく，プログラム開始時，全てのデータが OffChipCSM 上に配置されているものとする．なお，本論文では条件分岐を含まないスタティックスケジューリング可能な階層を対象とする．そのため，本論文では OnChipCSM は利用せず，OffChipCSM のみを利用する．

DTU の駆動には二種類の方法がある．一つはコンパイラが DTU に対するデータ転送命令をあらかじめ転送パラメータとして LDM 上に設定し，実行時に転送パラメータの先頭アドレスを DTU に通知し，DTU を駆動する方法である．このとき，複数のパラメータを LDM 上の連続する領域に設定しておけば，パラメータチェインが形成され，CPU による一度の駆動で複数の領域の転送が可能となる．もう一つは CPU が転送パラメータ値を直接 DTU のレジスタに設定し，駆動する方法である．

4.5　ローカルメモリ管理適用対象

本節では，提案手法の適用対象について説明する．提案手法は FOTRAN あるいは制約付き C（Parallelizable C）[間瀬 06] で記述されたプログラムを入力として想定している．

提案手法の適用対象データに関しては，プログラム中のグローバル配列とスタック上の自動配列変数を候補としてローカルメモリへ割り当てる．再帰呼び出しが含まれていないプログラムを前提としているので，プログラム中の自動配列変数は，一旦すべてグローバル配列へと変換される．変換された自動配列変数を含むグローバル配列はオフチップメモリ上に配置される．このオフチップメモリ上のグローバル配列を，ローカルメモリ上に確保した領域に動的に割り当て実行する．入力として FORTRAN あるいは制約付き C を想定しているため，実行時に使うサイズが決定されるヒープ領域は提案手法の対象としていない．

粗粒度タスク並列処理では階層的にマクロタスクグラフを生成する．ここで，提案手法の適用対象の粗粒度タスク並列処理階層と粗粒度タスク（マクロタスク）について述べる．コンパイラによる静的な解析情報に基づくローカルメモリ管理を実現するために，粗粒度タスク並列処理階層のうち，提案手法はスタティックスケジューリング可能な，条件分岐を含まない階層を対象とする．また，マクロタスク内で定義・参照される配列を直接プロセッサのローカルメモリに割り当てるために，PG 内プロセッサ数が 1 である階層を対象とする．これらの条件に合致する階層を対象階層と呼ぶ．

従来のループ整合分割を用いたデータローカライゼーション手法では，分割された部分ループをローカルメモリ割り当ての対象としているのに対し，提案手法はデータ分割の必要のない小データにアクセスするマクロタスクもローカルメモ

リ割り当ての対象としている．また，グローバルループ整合分割では，ローカルメモリサイズを超えるマクロタスクは内側階層をさらに分割して，内側階層をローカルメモリ割り当ての対象としている．内側階層の分割が可能でなければ，ローカルメモリ割り当て時の利得を配列毎に計算し，利得の大きい配列から順に収まる範囲でローカルメモリに割り当てている．

4.6 グローバルループ整合分割

キャッシュメモリアーキテクチャでは，粗粒度タスク内で定義あるいは参照される配列サイズがたとえキャッシュサイズを超えていたとしても，効率は悪化するが，キャッシュメモリを使って動作させることができる．それに対して，全てのデータ配置とデータ転送をソフトウェアが管理するローカルメモリアーキテクチャでは，粗粒度タスク内で配列がローカルメモリサイズを超えて，定義あるいは参照されると，超えた分はローカルメモリに割り当てることができず性能が低下する．

従来のループ整合分割における課題としては，ターゲットループグループ間でのデータローカリティを考慮していない点と多重ネストにわたる分割を考慮していない点の二点が挙げられる．ターゲットループグループ間でデータローカリティを生かすためには，アクセスされる配列形状を異なるターゲットループグループ間で揃える必要がある．そのために，従来のループ整合分割では，同一ターゲットループグループ内のループ間でのみ計算していたループ間データ依存解析を，異なるターゲットループ内のループ間でも計算するように拡張する．また，多重ネストにわたる分割を行なうために，ターゲットループグループを多重ネストにわたり定義し，異なる階層中のターゲットループグループ同士であっても，ループ間データ依存を解析することで，多重ネストにわたり，整合をとった

分割を行なう．これにより，多重にネストしたループで多次元配列にアクセスするような場合に，ローカルメモリサイズに応じた分割を実現する．

グローバルループ整合分割の処理手順について述べる．グローバルループ整合分割はループ整合分割をベースとしており，ネストした多重ループの分割や異なるターゲットループグループ間で，同一の配列形状への分割を目的とする．

(1) 従来のループ整合分割同様にターゲットループグループを生成する．ただし，ネストしたループについても，ターゲットループグループを生成する．図 4.1 のマクロタスクグラフに対して，ターゲットループグループを生成した場合を図 4.7 に示す．

(2) 同一の配列の，同一の次元にアクセスするターゲットループグループ同士をグループ化する．グループ化されたターゲットループグループを拡張ターゲットループグループ（ExTLG）と呼ぶ．図 4.7 のターゲットループグループに対して，拡張ターゲットループグループを生成した場合を図 4.8 に示す．

(3) 拡張ターゲットループグループ中で，最大のコストをもつループを拡張標準ループと呼ぶ．図 4.8 では，ExTLG1 に対しては RB2 が，ExTLG2 に対しては RB6 がそれぞれ拡張標準ループに選択されている．

(4) 拡張ターゲットループグループ中の各ループについて，拡張標準ループを基準として，ループ間データ依存解析を行なう．ExTLG1 に対するループ間データ依存解析結果を図 4.9 に，ExTLG2 を図 4.10 にそれぞれ示す．

ExTLG の分割数は容量的負荷の大きなマクロタスクを持つ ExTLG から順に決定する．マクロタスクの容量的な負荷とは，そのマクロタスクで定義・参照される配列範囲サイズに加え，そのマクロタスクを生きて通過する可能性のあるフロー依存配列範囲サイズおよび入力依存配列範囲サイズの合計値で定義される．

この容量的負荷が最大のマクロタスクがローカルメモリサイズに収まるように分割数を決定する．最大限分割したとしても，ローカルメモリに収まらない場合は，内側のネストや関数呼出先に存在する ExTLG を分割する．存在しなければ，ローカルメモリ配置の利得の大きな配列から順にローカルメモリ割り当て対象とする．

図 4.11 に ExTLG1 を 3 分割したマクロタスクグラフを示す．ターゲットループグループ間で整合をとって分割することで，RB1_3, RB2_3, RB3_3, RB4_3 にわたり，同一の配列形状となった．これらのマクロタスク間のデータローカリティを後述するスケジューラが検出し，プロセッサに連続的に割り当て，さらに同一のデータを同一のローカルメモリアドレスに割り当てることで，従来のループ整合分割では利用することができなかったデータローカリティを有効に利用する．

ExTLG1 を最大限分割しても，ローカルメモリに収まらなかった場合，ExTLG2 についても分割する．図 4.12 に ExTLG2 を 2 分割したマクロタスクグラフを示す．このとき，容量的負荷が最大のマクロタスクで必要となるサイズは 480bytes となり，512bytes のローカルメモリであっても割り当てることが可能となる．一方で，従来のループ整合分割では図 4.5 に示した通り，一つの配列しか割り当てることができない．さらに，複数の次元にわたって整合をとって分割することで，RB1_3_3, RB2_3_2, RB3_3_3, RB4_3_3 にわたってデータローカリティを有効に利用することができる．

4.7　ローカルメモリ管理単位ブロック

配列のような連続的なアドレスを持つデータを割り当てる場合，そのサイズに応じた連続領域をローカルメモリ上に確保する必要がある．サイズや次元数が異

なる配列を，ローカルメモリ上の空き領域に割り当てることは，未使用な連続領域を断片化させ，フラグメンテーションを発生させる．そこで，提案手法ではローカルメモリをブロックという単位に仮想的に区切り管理する．各ブロックには配列が一つ割り当てられる．また，割り当てられた配列をオフチップメモリに書き戻すことでブロックは解放される．異なるサイズのブロック同士は，互いに重複してローカルメモリ上にマッピングされる．

各ブロックには以下のように level とブロック番号が割り振られる．ローカルメモリ上に確保できる最大のブロックサイズを $full\_block\_size$ とし，その level を 0 とする．そして，level が l であるブロックの 1/2 のサイズのブロックの level を $l+1$ とする．また，各 level 毎にアドレスの低い方を 0 とし，順にブロック番号を割り振る．

ある配列を割り当てるブロックの level は次のように決まる．アクセス範囲の上下限値から，配列の次元毎に要素数を求めて掛け合わせる．ループ回転数が変数であるなど，ある次元のアクセス要素数がコンパイル時に決定できなければ，配列次元の宣言要素数で代用する．求めた全要素数に配列要素当たりのサイズを掛けて配列のサイズ S を求める．

S が

$$\frac{full\_block\_size}{2^{l+1}} < S \leq \frac{full\_block\_size}{2^{l}} \tag{4.1}$$

である場合，この配列は level l のブロックへ割り当てられる．また，この配列の level は l であるという．

図 4.13 に 1024bytes のローカルメモリへ level 0 から level 3 までのブロックをマッピングした例を示す．ただし，level が l で，ブロック番号が n のブロックを $Block_n^l$（または B_n^l）と表記する．たとえば，図中の $Block_5^3$ は $Block_0^0, Block_1^1, Block_2^2$ と互いに重複して配置されるので，同時に使用すること

はできない.

4.8　ローカルメモリ割り当てと解放

配列単位でブロックへ割り当てを行なう場合，適切なブロックに割り当てないと，ローカルメモリ内でのデータの移動やオフチップメモリとローカルメモリ間のデータ転送が頻繁に発生し，却って性能が低下し得る．そのような事態を避けるために，予めマクロタスクの実行順序を決定し，各配列の定義・参照時刻を求めることで，ブロック割り当てやブロック解放に必要なデータ転送を最適化する．

粗粒度タスクの実行順序の決定を行なう粗粒度タスクスタティックスケジューリングについて 4.8.1 節で述べる．そして，配列のブロックへの割り当てについて 4.8.2 節で，ブロックの解放について 4.8.3 節でそれぞれ説明する．

4.8.1　データローカリティを考慮した粗粒度タスクスタティックスケジューリング

グローバルループ整合分割によって分割されたループを含む，対象階層中のマクロタスクの実行順序を決めるために，粗粒度タスクスタティックスケジューリングを行なう．粗粒度タスクスタティックスケジューラは，3.3 節で述べたスケジューリングアルゴリズムに加え，クリティカルパス長，データ転送時間，そして後続タスク数をプライオリティとしたヒューリスティックなリストスケジューリングを行なう．このうち，もっとも短いスケジューリング長が得られたスケジューリング結果を採用する．スケジューリングによって，マクロタスクの実行順序と各マクロタスクのスケジューリング時刻を求める．

4.8.2 グローバルループ整合分割後の部分配列のブロックへの割り当て

先行タスクのプロセッサへの割り当て状況およびこれらのタスク内でアクセスされるデータのローカルメモリへの割り当て結果をもとにして，データ転送を最小化するために，スケジューリング時刻の早いマクロタスクで定義・参照される配列から順にブロックへの割り当てを行なう．

マクロタスクで定義・参照されるローカルメモリ割り当て対象の配列のうち，level の小さな，すなわちサイズの大きな配列から順に，ブロックへ割り当てる．同名の配列が既に割り当て済みのブロックおよび割り当て可能な空いているブロックに，それぞれ割り当てた時のデータ転送時間を計算し，データ転送時間が最小となるブロックへ割り当てる．割り当て可能なブロックが見つからなければ，4.8.3 節で述べるブロックの解放を行なう．

必要なデータ転送時間に差がないブロックが複数見つかった場合，level の小さなブロックがなるべく割り当て可能なまま残るように，割り当てるブロックを選択する．たとえば，図 4.13 において，level が 2 のサイズの配列を割り当てる際に，$Block_0^2, Block_1^2, Block_3^2$ が割り当て可能で，いずれのブロックに割り当てもデータ転送時間が等しかったとする．このとき，$Block_5^3$ が割り当て済みならば，配列を $Block_3^2$ に割り当てることで，$Block_0^1$ を割り当て可能なまま残す．

配列をブロックに割り当てたら，そのマクロタスクで参照する配列範囲のうち，ブロック上に載っていない範囲をオフチップメモリからロードする．

4.8.3　ブロックの解放

過去に割り当てられた配列がそのプロセッサ上で再度参照されることがなければ，その配列が割り当てられているブロックは解放する．そうでなければ，参照される配列のサイズを，参照されるまでの時間で割った値である再参照度をブロック毎に求める．参照されるまでの時間はマクロタスクのスケジューリング時刻から求める．単に参照されるまでの時間で解放候補を選択すると，サイズが大きくデータ転送時間の長い配列が頻繁に解放されてしまうことがあるので，サイズも合わせて考慮する．この再参照度が小さいブロックほど将来にわたり，参照される頻度が低いことを表すので，再参照度の低いブロックから順に必要に応じ解放する．

ここで，ブロックが割り当て済みである場合，ブロック $Block_n^l$ の再参照度 $R_{Block_n^l}$ は

$$R_{Block_n^l} = \sum_j \frac{S_j}{T_j - T_i} \tag{4.2}$$

とする．ただし，マクロタスク MT_i を割り当て中のマクロタスクとし，マクロタスク MT_j を MT_i 以降に同一プロセッサに割り当てられたマクロタスクとする．T_i, T_j をそれぞれ MT_i, MT_j のスケジューリング時刻とする．ブロックに割り当てられた配列範囲のうち，MT_j で参照されるサイズを S_j とする．

割り当て対象の配列のサイズに対応するブロック自体には他の配列は割り当てられていないが，level がより大きな，重複するブロックが割り当て済みの場合，再参照度は以下のように下位のレベルに向かって再帰的に求める．

$$R_{Block_n^l} = R_{Block_{2n}^{l+1}} + R_{Block_{2n+1}^{l+1}} \tag{4.3}$$

ただし，解放済みの割り当て可能なブロックの再参照度は 0 として計算する．

解放するブロックが決定したら，そのブロック上に割り当てられていた配列をオフチップメモリに書き戻す．書き戻すタイミングはその配列を定義したマクロタスクの直後とする．

図 4.14 を使って，図 3.2 で示した $DLG_m(1 \leq m \leq 3)$ 中のマクロタスクで定義・参照される配列のブロックへの割り当てと解放について説明する．図中の左側の列からスケジューリング時刻，マクロタスクのスケジューリング結果，そして右側の列がプログラムコード（Code）か，ローカルメモリの割り当て結果（LM）かの区別をそれぞれ表す．右側の列はローカルメモリ割り当て結果と，RB_2^m 以降の配列の定義・参照関係を含むプログラムコードを表す．図中の矢印は，その配列が起点で定義され，終点で参照されることを表すフロー依存を示す．ただし，スケジューリング時刻は $T_1^m < T_2^m < T_3^m$ であるものとする．

今，RB_1^m までの割り当てが終わっているものとする．このとき，配列 a, b, c が $Block_0^1, Block_2^2, Block_3^2$ にそれぞれ割り当てられている．スケジューリング結果に従い，配列 a, d が定義・参照される RB_2^m をブロックに割り当てる．level が 1 の配列 a から割り当てる．$Block_0^1$ に同名の配列が割り当て済みなので，そこへ割り当てる．level が 2 の配列 d を割り当てるブロックがないので，$Block_2^2, Block_3^2$ について再参照度を計算する．$Block_2^2$ の配列 b と $Block_3^2$ の配列 c は参照されるまでの時間は等しいが，配列 b の方がサイズが大きく，再参照度が大きくなる．そこで，配列 b を残し，$Block_3^2$ を解放する．解放により，空いた $Block_3^2$ に配列 d を割り当てる．最後に，RB_1^m の直後で，配列 c をオフチップメモリに書き戻す．RB_3^m では，配列 a, b はそのまま利用され，空いている $Block_3^2$ に $c[m-1:m][0:24]$ がロードされる．

ここで，図 3.2 で示した $DLG_m(1 \leq m \leq 3)$ 中のマクロタスクで定義・参照される配列従来のターゲットループグループ内での固定的な割り当てにより，1024bytes のローカルメモリに割り当てた場合を，図 4.15 に示す．このとき，

RB_1^m における配列 d のように，未使用領域があるにも関わらず，固定的な割り当てでは一部の部分配列をローカルメモリに載せることができず，利用効率が低下する．

それに対し，提案手法では，同じ 1024bytes のローカルメモリを使い，図 4.14 の LM の各行に示した通り，RB_1^m, RB_2^m, RB_3^m にわたり，定義・参照される部分配列が全てローカルメモリに割り当てられ，利用効率が向上している．このようにターゲットループグループを最大限分割したとしても，データローカライゼーショングループ内の各部分ループで定義・参照される配列サイズの合計がローカルメモリサイズに収まらないような場合，配列の定義・参照に応じたローカルメモリ管理を行なう提案手法は，ローカルメモリをより有効に利用することができる．

4.9 テンプレートを用いたローカルメモリ管理出力コード

提案手法により，図 3.1 をローカルメモリ管理した出力コード例を図 4.16 に示す．図中のグローバル配列 LM （1 行目）はローカルメモリに割り当てられるものとする．自動配列変数からグローバル配列に変換された func0_a, func0_b, func0_c, func0_d （2, 3 行目）はオフチップメモリに割り当てられる．ローカルメモリに重複して配置される配列を記述するために，テンプレートポインタと呼ぶポインタ変数を導入する．テンプレートポインタは，プログラム中に出現する配列の次元数やサイズ毎に出力され，それぞれ LM の先頭アドレスで初期化される（5–7 行目）．テンプレートポインタの一次元目は，利用するブロック番号を表す．残りの次元は元の配列の各次元に対応する．たとえば，9 行目の func1 の第一引数である t1 は，level が 1 であり，一次元目の添字が 0 なので，$Block_0^1$

を利用することを意味する．10, 13, 14 行目にはローカルメモリ管理の結果生じたデータ転送命令が追加されている．

4.10　性能評価

本章では，提案手法を OSCAR マルチグレイン自動並列化コンパイラ [KWN+05] に実装し，4.10.1 節で述べる RP1 および RP2 マルチコア上で性能評価した結果について報告する．

4.10.1　評価に用いるマルチコア RP1 および RP2

提案手法が任意のプロセッサ数や異なるローカルメモリサイズであっても管理できることを確かめるために，ローカルメモリサイズとプロセッサ数がそれぞれ異なる RP1 および RP2 マルチコア [YKH+07, IHY+08] 上で性能評価を行なった．RP1 および RP2 マルチコアは NEDO 半導体アプリケーションチップ「リアルタイム情報家電用マルチコア」プロジェクトにおいてルネサステクノロジ，日立製作所，早稲田大学により開発された，コンパイラ協調型マルチコアである．それぞれ，SH4A プロセッサコアを 4，8 コア集積している．

RP1 および RP2 マルチコアは 4.4 節で説明した OSCAR マルチコア型のアーキテクチャ構成となっている．RP1 マルチコアの構成図を図 4.17(a) に，RP2 マルチコアを図 4.17(b) にそれぞれ示す．また，RP1 および RP2 の仕様を表 4.1 にまとめる．図中の OLRAM がローカルデータメモリに，URAM が分散共有メモリにそれぞれ対応する．チップの内外に集中共有メモリを持つ．OLRAM が提案手法が対象とするローカルメモリである．OffChipCSM がオフチップメモリに相当する．なお，本論文における評価では，OnChipCSM を使用せず，集中共有メモリとしては OffChipCSM のみを利用するものとする．

表 4.1 にある通り，OLRAM のサイズが RP1 で 16KB，RP2 では 32KB となっている．これら 2 種類のマルチコアを用いることにより，提案手法のローカルメモリサイズによる効果の違いを評価する．

表 4.1 RP1 および RP2 の仕様

	RP1	RP2
プロセッサ数	4	8
OLRAM レイテンシ	1–2 クロック (*)	1–2 クロック (*)
OLRAM サイズ	16KB	32KB
URAM レイテンシ	2 クロック (**)	2 クロック (**)
URAM サイズ	128KB	64KB
I-キャッシュ レイテンシ	1 クロック	1 クロック
I-キャッシュ サイズ	32KB	16KB
OnChipCSM レイテンシ	約 12 クロック	約 12 クロック
OnChipCSM サイズ	128KB	128KB
OffChipCSM レイテンシ	約 55 クロック	約 55 クロック

(*): ページ競合の有無による

(**): URAM 前段のバッファヒット／ミスヒットによる平均

4.10.2　評価アプリケーション

AAC エンコーダおよび MPEG2 エンコーダを用いて提案手法の性能評価を行なった．これらのアプリケーションは制約付き C （Parallelizable C）[間瀬 06] で記述されている．入力データが OffChipCSM 上に配置された状態から出力結果を OffChipCSM に書き戻すまでの時間を評価の対象とした．

AAC エンコーダは 30 秒の音源を入力データとし，出力データのビットレートは 128Kbps とした．MPEG2 エンコーダは MediaBench[LPMS97] に収録されている
“mpeg2encode” を参照実装したプログラムを用いた．入力データには 352x256 ピクセルの画像 30 フレームをそれぞれ用いた．

4.10.3　マルチメディア処理に対する性能

AAC エンコーダ, MPEG2 エンコーダに対して, 提案手法を実装した OSCAR コンパイラでローカルメモリ管理を行ない，RP1, RP2 上で評価した．AAC エンコーダにおける実行時間を図 4.18 に，MPEG2 エンコーダにおける実行時間を図 4.19 にそれぞれ示す．図中の横軸は RP1 の 1, 2, 4PE，RP2 の 1, 2, 4, 8PE の適用手法別の結果をそれぞれ表す．適用手法のうち，CSM はプログラム中の全ての配列を一切ローカルメモリに配置せず，全て OffChipCSM に配置した場合の実行時間を表す．こちらはローカルメモリ管理手法がない場合の最も単純な処理方式を想定している．LAD は 4.2 節で述べた従来のループ整合分割を用いたデータローカライゼーション手法 [KY98, 吉田 99] の実行時間を表す．PROPOSED は提案手法の実行時間を表す．また，各バーはメモリアクセス時間とその残りに分けられている．メモリアクセス時間はオフチップメモリにある

かローカルメモリにあるかに関わらず，プログラム中でメモリアクセスにかかった時間および DTU によるデータ転送時間の合計を表す．

図 4.18 の AAC エンコーダを用いた評価において，実行時間からメモリアクセス時間を引いたそれ以外の時間が，各マルチコア，各 PE 数毎に，適用手法によらずほぼ一定であることがグラフから読みとれる．このことから，適用手法ごとの実行時間の差はメモリアクセス時間の差に起因することが分かる．

RP1 上の 4PE 実行で，CSM と提案手法を比較すると，約 5.0 倍の速度向上を実現した．同様に RP2 上の 8PE 実行では，提案手法は CSM に対し，約 8.4 倍の速度向上を実現した．CSM ではすべての配列が OffChipCSM に配置されているため，メモリアクセスに長大な時間がかかっているのに対し，提案手法ではコンパイラによる自動的なローカルメモリ管理により，プロセッサ近傍のローカルメモリを有効利用できたことが分かる．

次に，LAD と提案手法を比較すると，RP1, RP2 のいずれのプロセッサ数においても，提案手法がより高い性能を示していることが分かる．RP1 の 4PE 実行で比較してみると，約 2.6 倍の，RP2 の 8PE 実行では，約 2.5 倍の速度向上がそれぞれ得られた．このときのメモリアクセス時間を比較してみると，RP1 の LAD では，約 2.2 秒であるのに対し，提案手法では約 0.5 秒と，約 1.7 秒向上している．同様に RP2 の LAD では，約 1.2 秒であるのに対し，提案手法では約 0.2 秒と，約 1.0 秒向上している．オフチップメモリに割り当てられたままの配列の総サイズを比較すると，プロセッサ数に関わらず，16KB のローカルメモリを持つ RP1 では，LAD は約 81KB，提案手法は 0KB，32KB のローカルメモリを持つ RP2 では，LAD は約 50KB，提案手法は 0KB であった．これらより，提案手法は固定的な割り当てを行なう LAD ではオフチップメモリに配置されてしまった配列を全てローカルメモリに割り当てることで，メモリアクセスにかかる時間を削減し，実行時間の短縮を実現したことが分かる．

提案手法がローカルメモリ割り当てによってメモリアクセス時間を削減しているだけでなく，並列性を引き出しているかを見るために提案手法同士で比較する．RP1 上で逐次実行に対して，4PE で約 3.3 倍，RP2 上で 8PE で約 4.4 倍とプロセッサ数に応じた速度向上が得られていることから，並列性を抽出できていることが分かる．

同一プロセッサ台数の RP1 と RP2 での提案手法のメモリアクセス時間をそれぞれ比較してみると，いずれの結果でも，16KB のローカルメモリを持つ RP1 に対し，32KB のローカルメモリを持つ RP2 のメモリアクセス時間は短くなっている．たとえば，4PE 同士で比較してみると，メモリアクセス時間は RP1 では約 0.5 秒であるのに対し，RP2 では約 0.3 秒と約 0.2 秒短縮している．提案手法ではすべての配列をローカルメモリに割り当てているため，この差はほぼデータ転送時間であり，提案手法は RP2 において RP1 に比べ，より少ないデータ転送で全ての配列をローカルメモリに割り当てている．これより，提案手法はローカルメモリサイズに応じたローカルメモリ管理を実現していることが分かる．

図 4.19 の MPEG2 エンコーダを用いた評価において，AAC エンコーダ同様，実行時間からメモリアクセス時間を引いたそれ以外の時間が，各マルチコア，各 PE 数で，適用手法によらずほぼ一定である．このことから，適用手法毎の実行時間の差はメモリアクセス時間の差に起因することが分かる．

RP1 上の 4PE 実行で，CSM と提案手法を比較すると，約 3.8 倍の速度向上を実現した．同様に RP2 上の 8PE 実行では，提案手法は CSM に対し，約 8.4 倍の速度向上を実現した．MPEG2 エンコーダでも AAC エンコーダ同様，コンパイラによる自動的なローカルメモリ管理により，プロセッサ近傍のローカルメモリを有効に利用していることが分かる．

次に，LAD と提案手法を比較すると，RP1, RP2 のいずれのプロセッサ数においても，提案手法がより高い性能を示していることが分かる．RP1 の 4PE 実

行で比較してみると，約 1.1 倍の，RP2 の 8PE 実行では，約 1.2 倍の速度向上がそれぞれ得られた．このときのメモリアクセス時間を比較してみると，RP1, 4PE の LAD では，約 13.5 秒であるのに対し，提案手法では約 11.3 秒と，約 2.2 秒向上している．同様に RP2, 8PE の LAD では，約 5.7 秒であるのに対し，提案手法では約 3.8 秒と，約 1.9 秒向上している．オフチップメモリに割り当てられたままの配列の総サイズを比較すると，プロセッサ数に関わらず，16KB のローカルメモリを持つ RP1 上で，LAD は約 9924KB，提案手法は約 6336KB，32KB のローカルメモリを持つ RP2 上で，LAD は約 5032KB，提案手法は約 3960KB であった．これより，提案手法は LAD に比べ，同一サイズのローカルメモリにより多くの配列を配置することで，メモリアクセス時間を削減し，実行時間短縮を達成したことが分かる．

提案手法の並列性能を見るために提案手法同士で比較すると，逐次実行に対して，RP1 上の 4PE で約 2.8 倍，RP2 上の 8PE で約 4.7 倍の速度向上が得られていることから，プロセッサ数に応じた負荷分散が行なわれていることが分かる．

また，提案手法において同一プロセッサ台数の RP1 と RP2 でのメモリアクセス時間をそれぞれ比較してみると，16KB のローカルメモリを持つ RP1 に対し，2 倍の 32KB のローカルメモリを持つ RP2 の時間の方が短い．たとえば，4PE 同士で比較してみると，メモリアクセス時間は RP1 では約 11.3 秒であるのに対し，RP2 では約 4.1 秒と約 7.2 秒短縮している．また，上述した通り，RP1 では約 6336KB の配列がオフチップメモリに残っているのに対し，RP2 では約 3960KB まで削減している．このことから，提案手法はローカルメモリサイズに応じたローカルメモリ管理を実現していることが分かる．

最後に，AAC エンコーダと MPEG2 エンコーダにおける LAD と提案手法の速度向上率の差について考察する．提案手法は AAC エンコーダにおいて，LAD がローカルメモリに割り当てられなかった配列を全てローカルメモリに割り当て

ている．それに対し，MPEG2 エンコーダでは，提案手法は一部の配列をローカルメモリに割り当てることができなかった．いずれの場合も提案手法は LAD に比べ，速度向上を達成しているが，オフチップメモリに残る配列の有無によって大きく速度向上率が変わることが分かる．

4.11 第 4 章のまとめ

ローカルメモリサイズに応じ，多次元配列を複数次元にわたり分割し，異なるターゲットループグループ間のデータローカリティを有効利用するために配列形状を同一になるように揃えるグローバルループ整合分割について述べた．

また，限られたローカルメモリをより効率的に利用するためには，粗粒度タスクスタティックスケジューリング結果を利用した，粗粒度タスク間にわたるデータの使用状況とローカルメモリの使用状況に応じた柔軟な割り当てを行なうローカルメモリ管理について述べた．

4 コアのマルチコアである RP1 上での性能評価の結果，メディアベンチマークである AAC エンコーダについて，従来のループ整合分割によるデータローカライゼーション手法を用いたローカルメモリ管理手法の処理時間が 2.8 秒であるのに対し，提案したローカルメモリ管理手法を用いることで 1.1 秒と 2.6 倍の，8 コアのマルチコア RP2 上では 1.7 秒が 0.7 秒と，2.5 倍の速度向上がそれぞれ得られた．MPEG2 エンコーダについて，4 コアの RP1 上で，従来手法が 23.2 秒であるのに対し，提案手法を用いることで 20.8 秒と 1.1 倍の，8 コアの RP2 上で，10.6 秒が 8.6 秒と 1.2 倍の速度向上がそれぞれ得られた．これらの性能向上が提案手法の有効性が確かめられた．

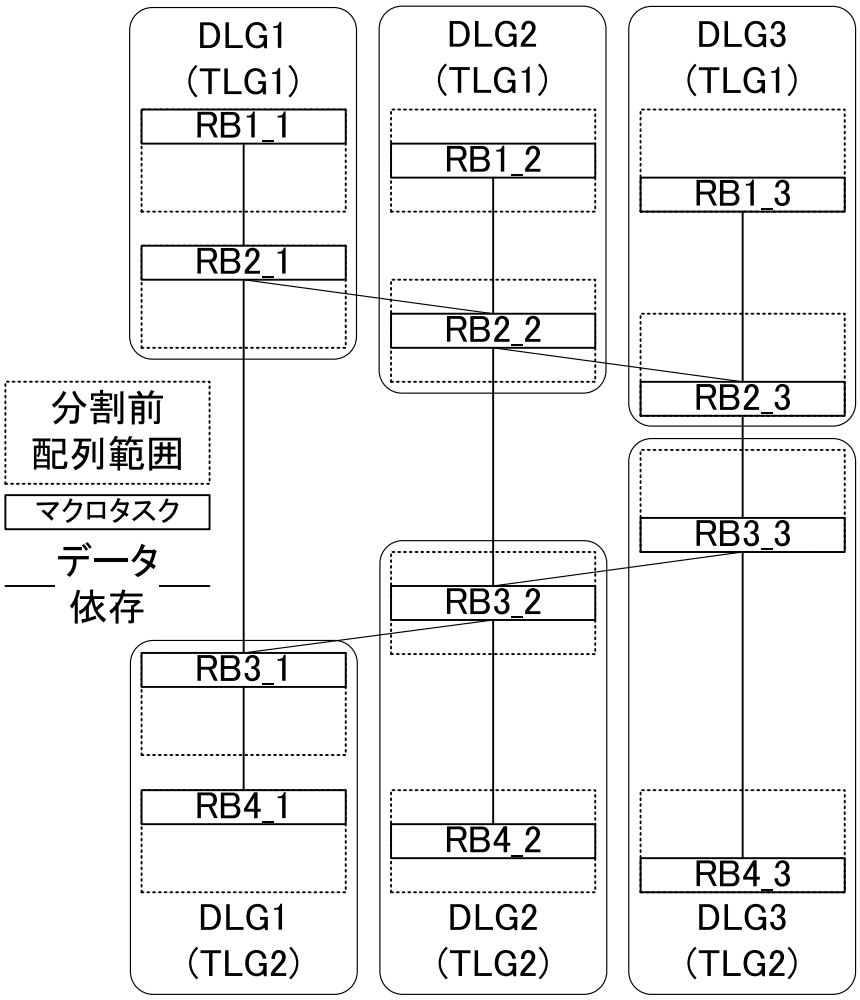


図 4.3　ループ整合分割後のマクロタスクグラフ

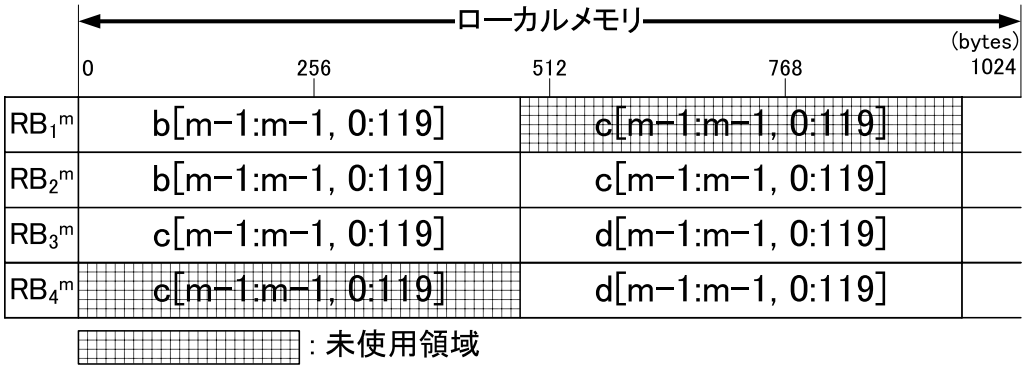


図 4.4　データローカライゼーション手法によるローカルメモリ割り当て（1024bytes）

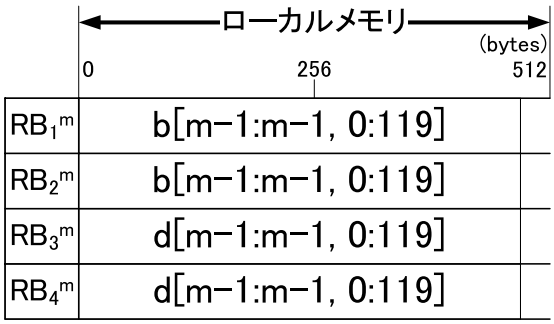


図 4.5　データローカライゼーション手法によるローカルメモリ割り当て（512bytes）

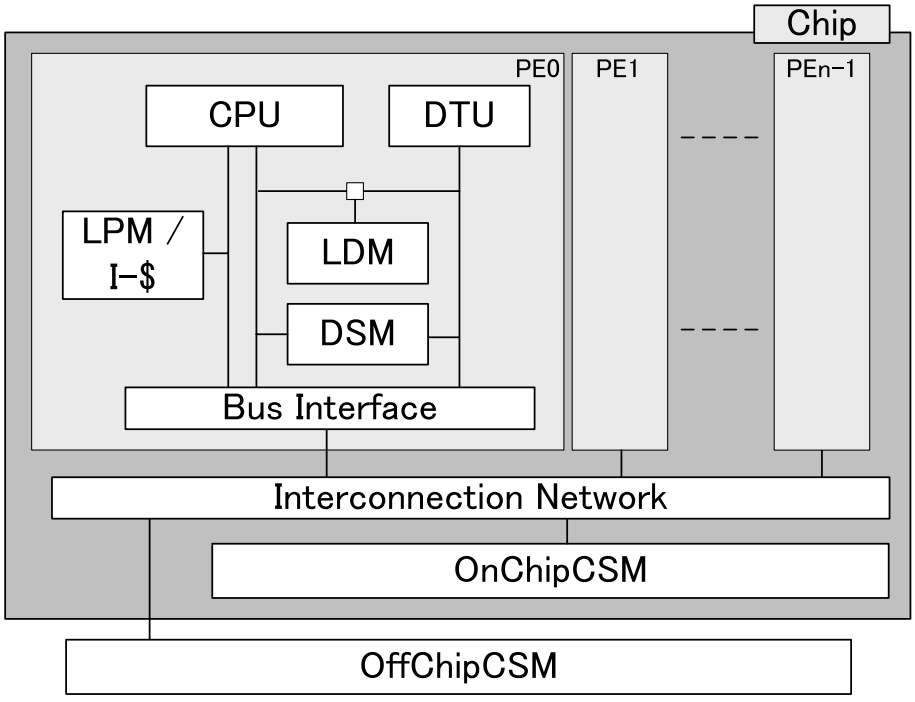


図 4.6　OSCAR マルチコアアーキテクチャ

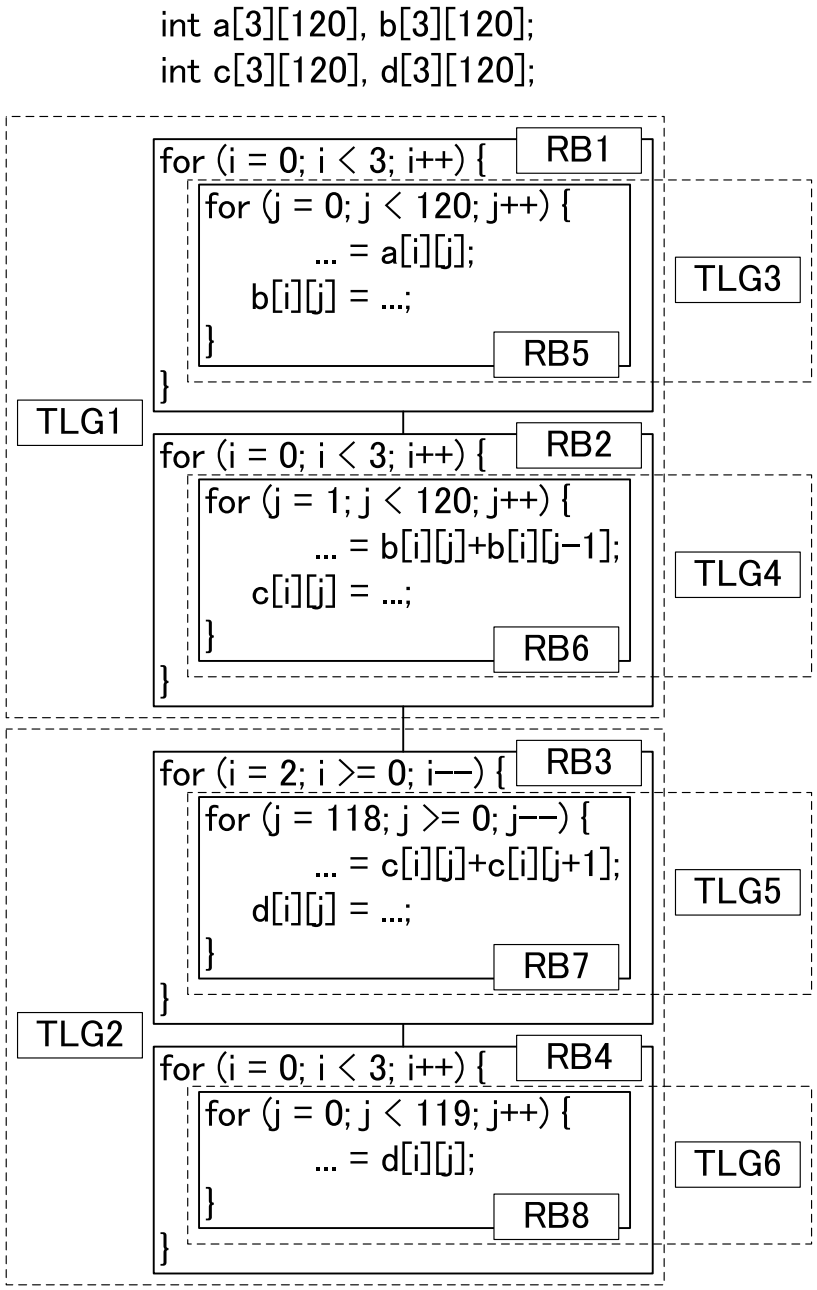


図 4.7　グローバルループ整合分割におけるターゲットループグループ

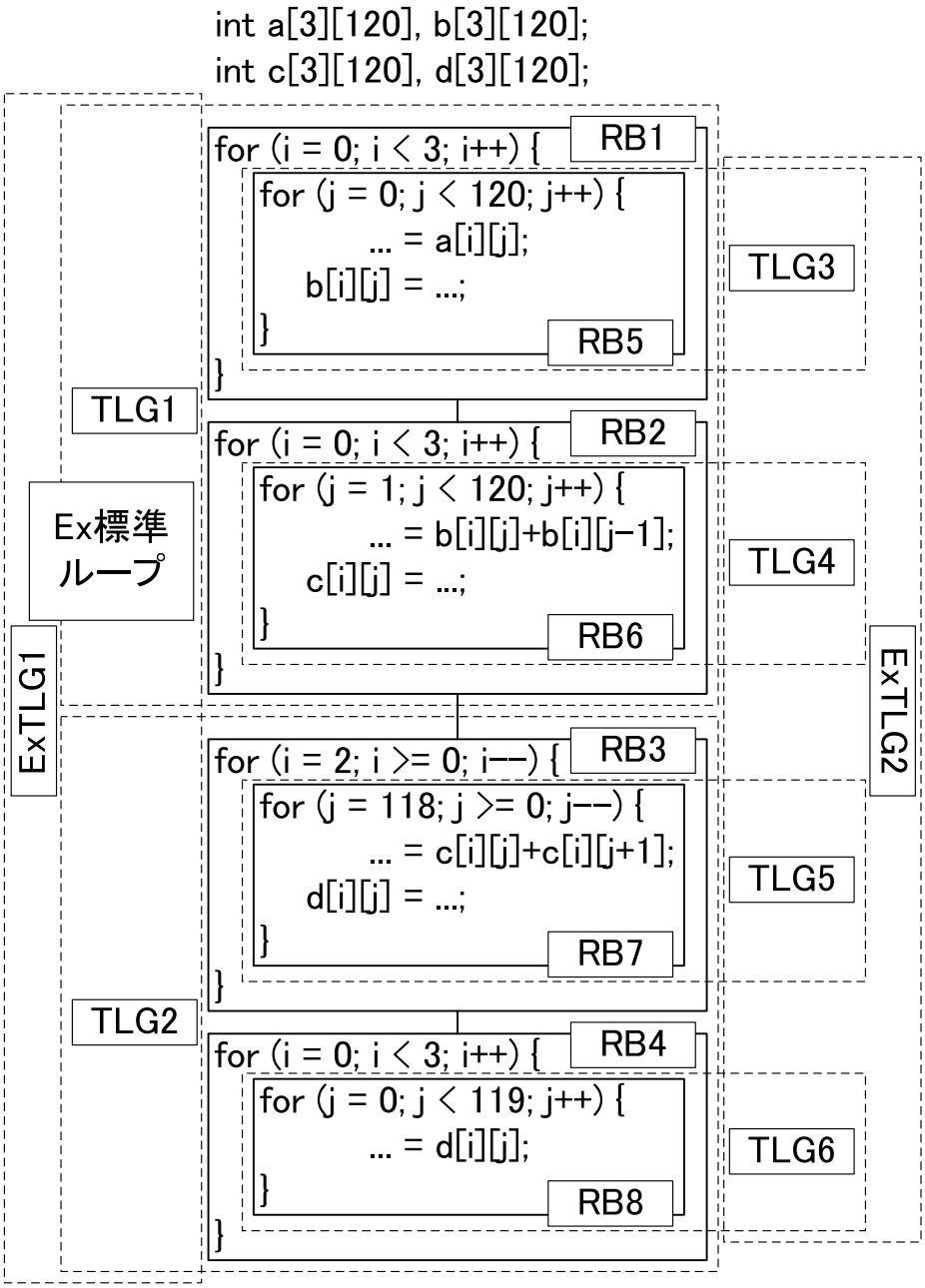


図 4.8　グローバルループ整合分割における拡張ターゲットループグループ

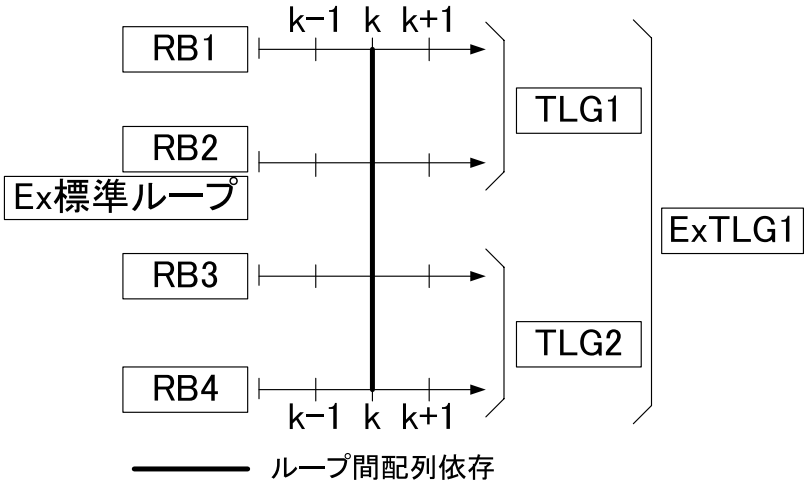


図 4.9　ExTLG1 に対するループ間データ依存解析結果

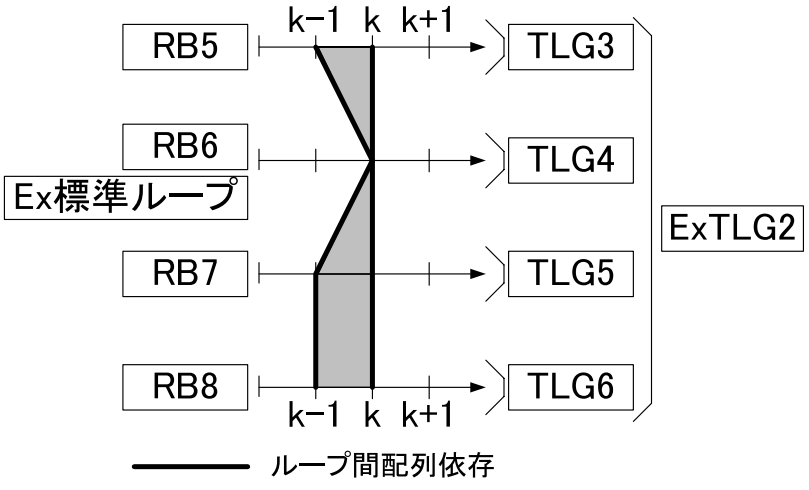


図 4.10　ExTLG2 に対するループ間データ依存解析結果

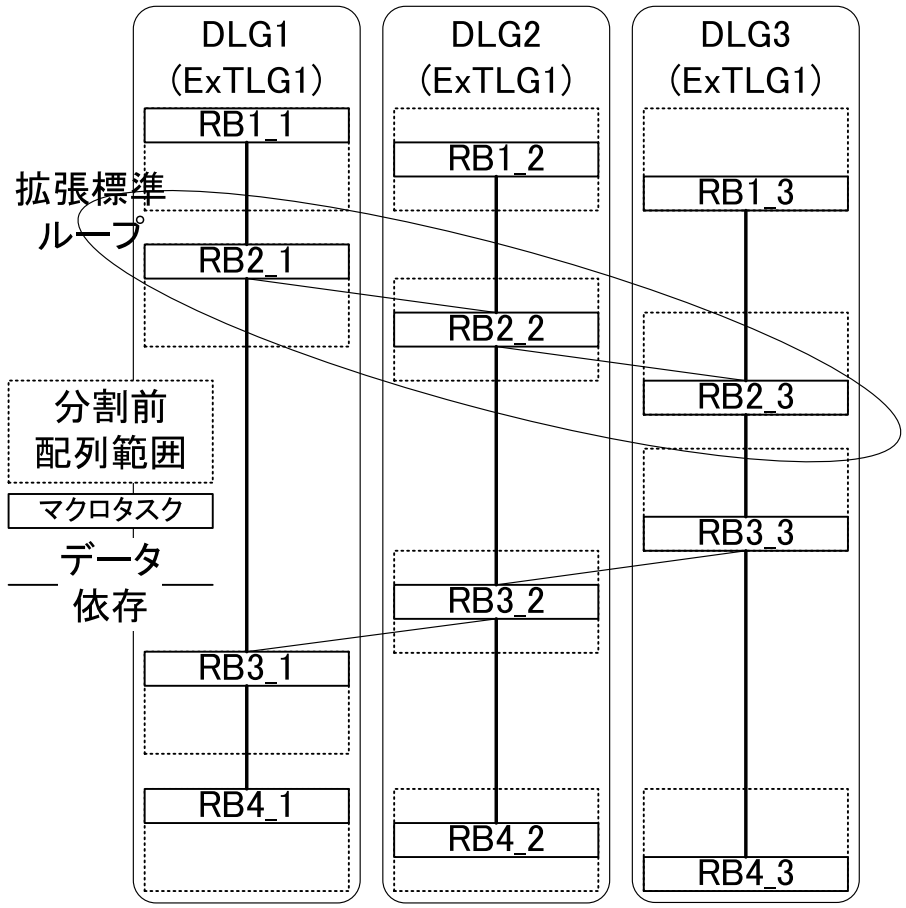


図 4.11　グローバルループ整合分割後のマクロタスクグラフ（1 重ネスト分割）

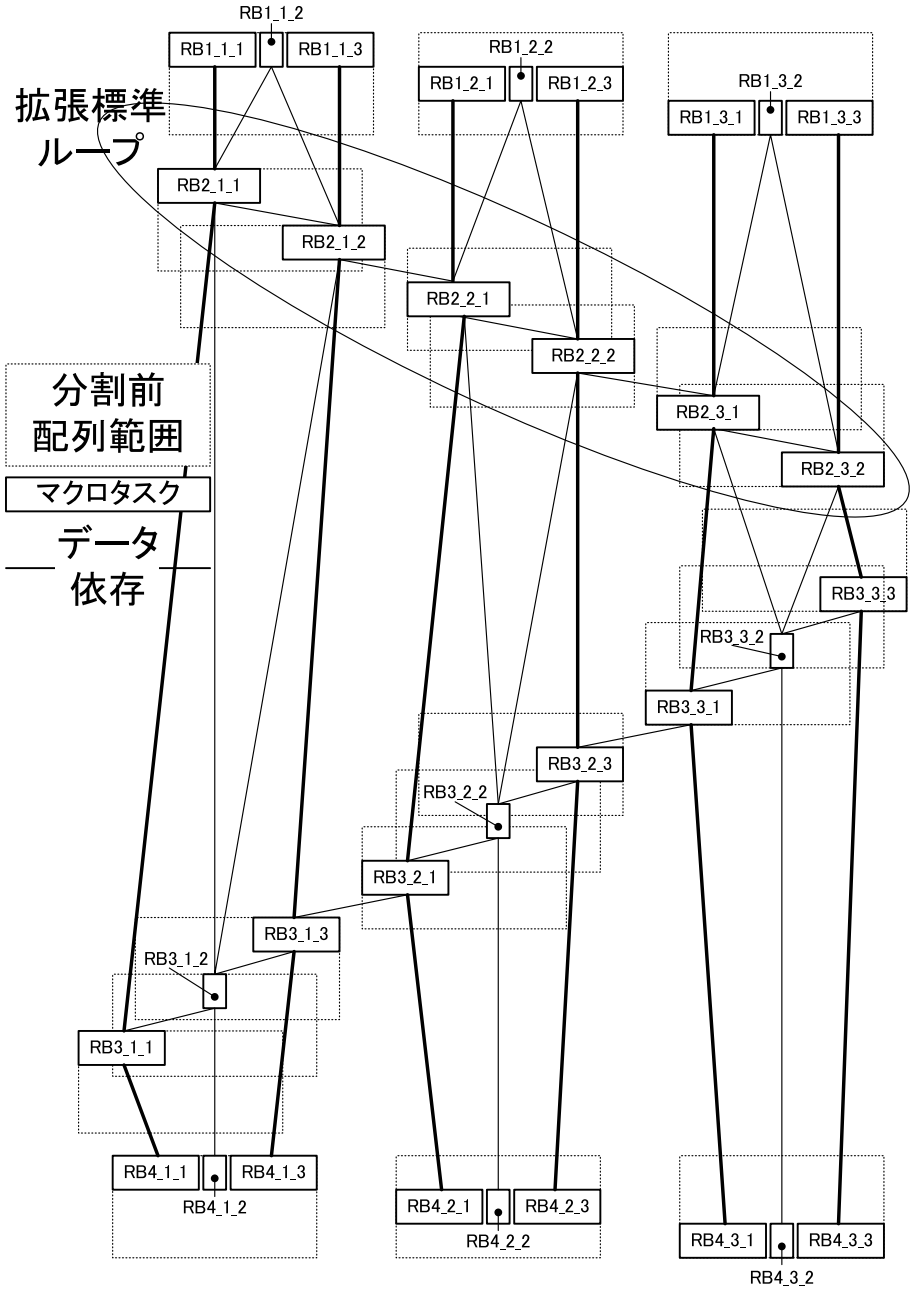


図 4.12　グローバルループ整合分割後のマクロタスクグラフ（2 重ネスト分割）

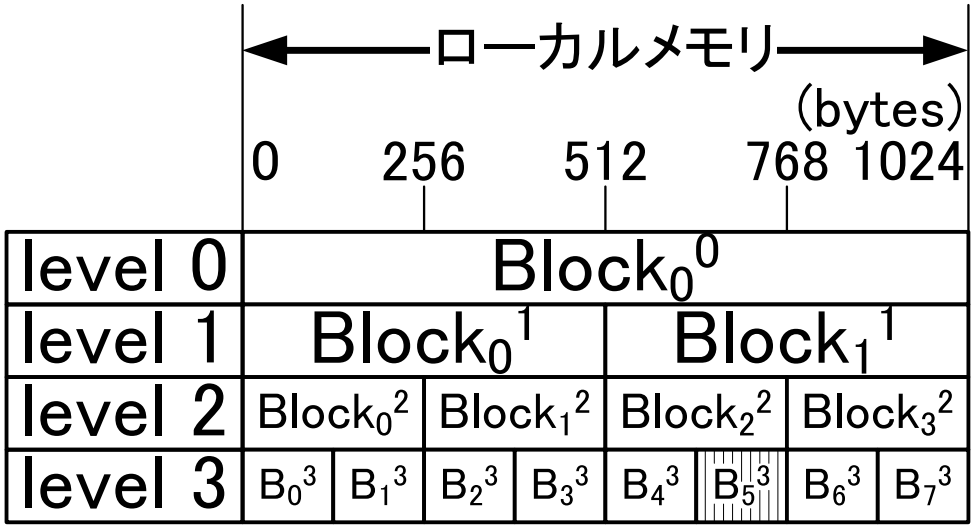


図 4.13　ローカルメモリへのブロック配置

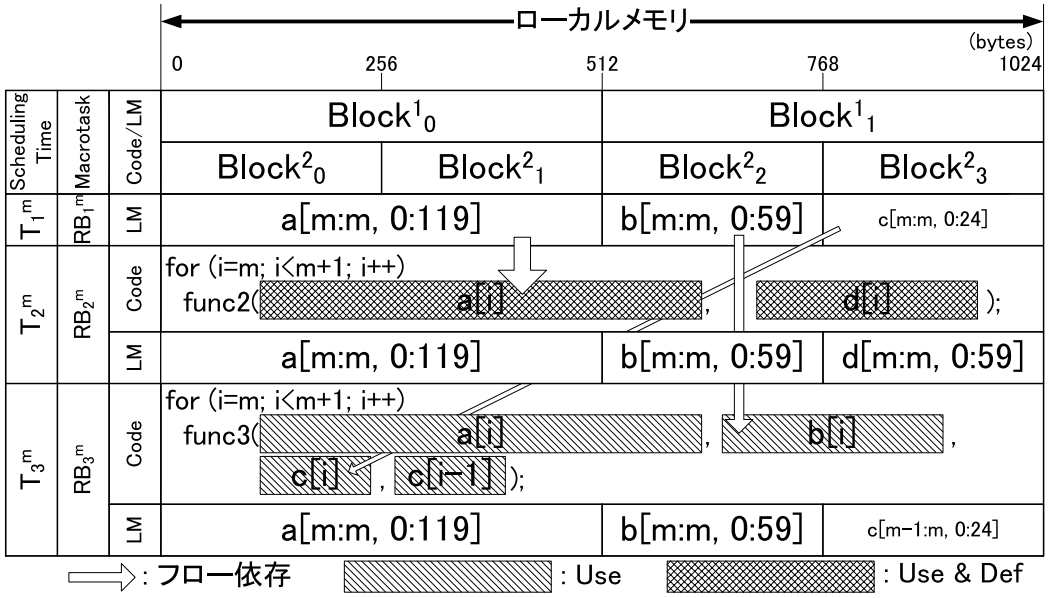


図 4.14　ローカルメモリ割り当てと解放

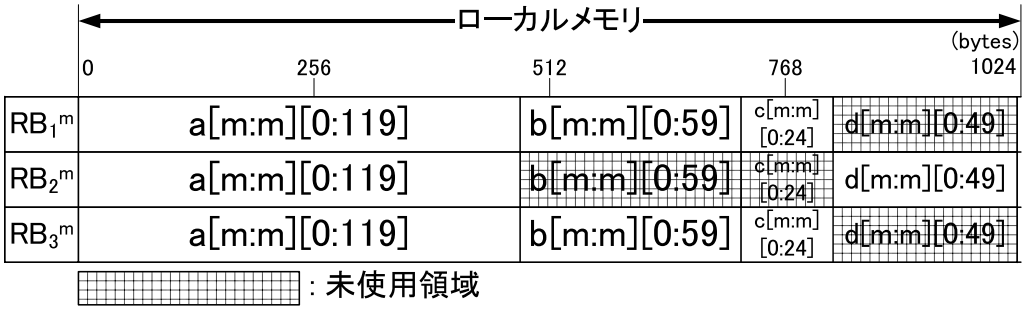
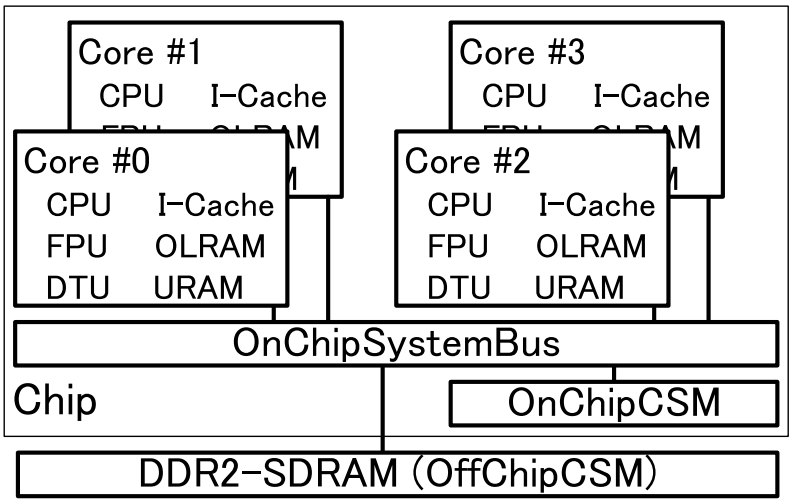


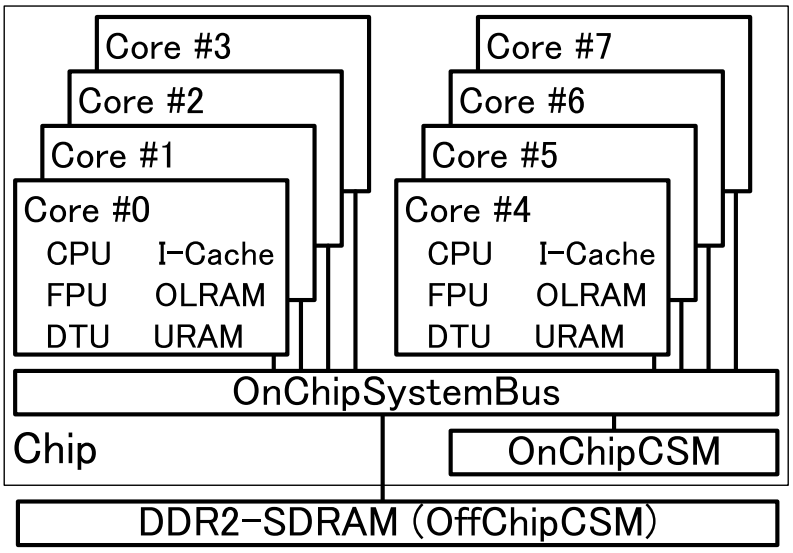
図 4.15　ループ整合分割によるローカルメモリ割り当て

```
   /* global variables */
   /** Local Memory */
 1 int LM[256];
   /** Off Chip Memory */
 2 int func0_a[4][120], func0_b[4][60];
 3 int func0_c[4][25], func0_d[4][50];
 4 void func0() {
    /* template pointer */
    /** level 1 */
 5  int (*t1)[1][128] = (int (*)[1][128]) (&LM[0]);
    /** level 2 */
 6  int (*t2)[1][64] = (int (*)[1][64]) (&LM[0]);
 7  int (*t3)[2][32] = (int (*)[2][32]) (&LM[0]);
    /* RB_1^1 */
 8  for (i = 1; i < 2; i++)
 9   func1(t1[0][i-1], t2[2][i-1], t3[3][i]);
    /* Local Memory to Off Chip Memory */
10  store(func0_c[1], t3[3][1], 25);
    /* RB_2^1 */
11  for (i = 1; i < 2; i++)
12   func2(t1[0][i-1], t2[3][i-1]);
    /* Off Chip Memory to Local Memory */
13  load(t3[3][0], func0_c[0], 25);
14  load(t3[3][1], func0_c[1], 25);
    /* RB_3^1 */
15  for (i = 1; i < 2; i++)
16   func3(t1[0][i-1], t2[2][i-1], t3[3][i], t3[3][i-1]);
    ...
   }
```

図 4.16　出力コード



(a) RP1



(b) RP2

図 4.17　RP1 および RP2 の構成図

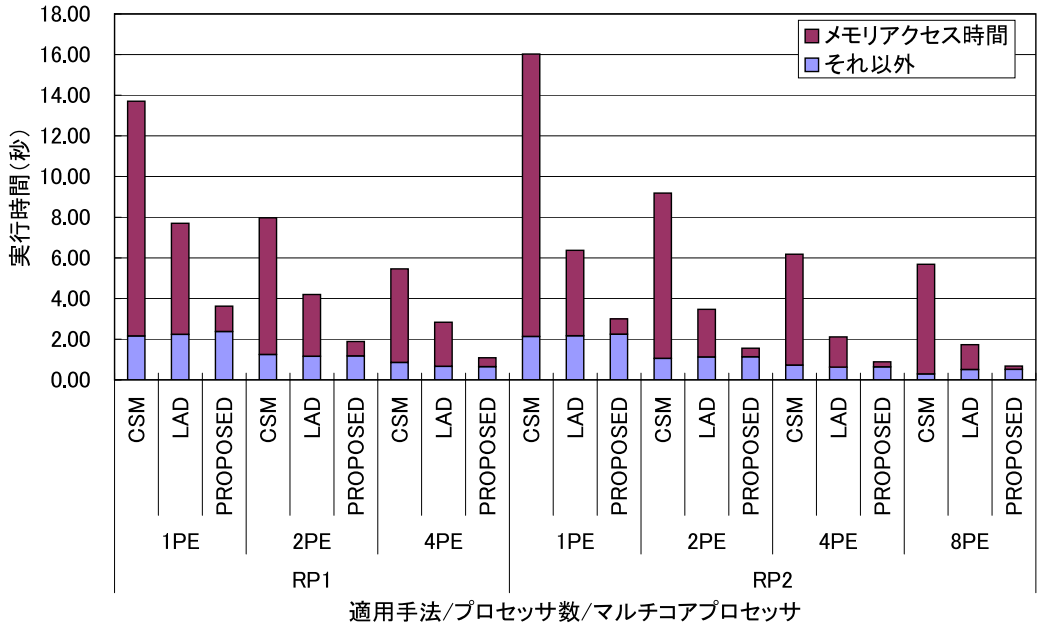


図 4.18　AAC エンコーダの実行時間

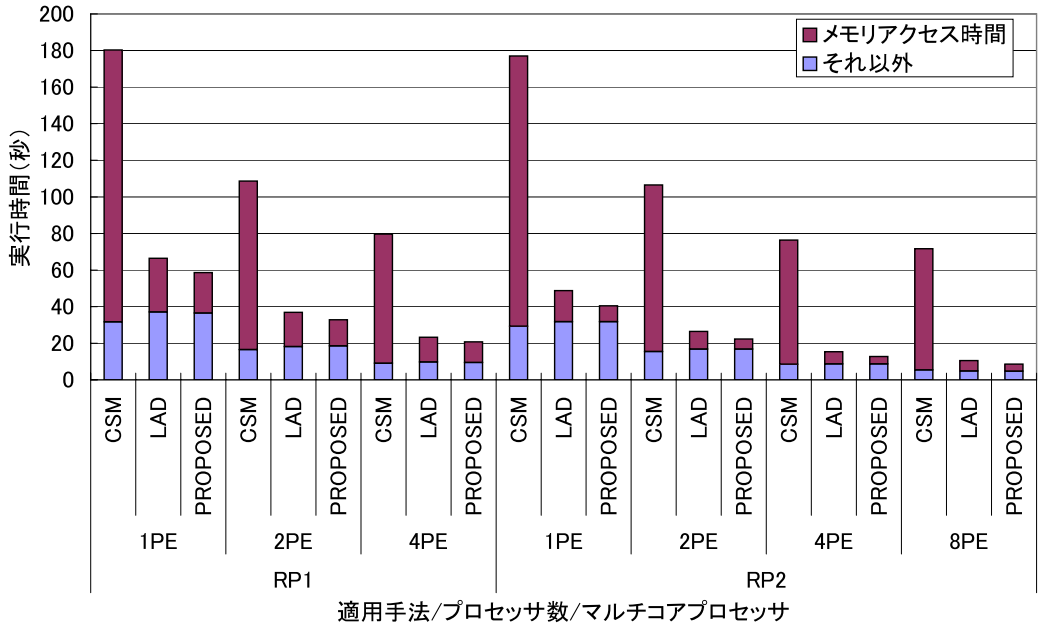


図 4.19　MPEG2 エンコーダの実行時間

第 5 章

結論

5.1　本研究により得られた成果

本論文では，OSCAR 自動並列化コンパイラにおけるキャッシュ・ローカルメモリ最適化について述べた．キャッシュ最適化手法では，粗粒度タスク並列処理において，ループ整合分割により粗粒度タスクでアクセスされるデータサイズをキャッシュにフィットさせ，粗粒度タスク間でのデータ共有量を考慮したスケジューリングを行なうことで，キャッシュの利用効率を向上させた．ローカルメモリ最適化では，よりデータローカリティを有効利用できるようにループ整合分割を拡張したグローバルループ整合分割と，データのスケジューリング結果を基にした，アクセスタイミングに応じたより柔軟なローカルメモリ管理を行なう手法について述べた．

以下に，本研究により得られた成果を総括する．

(1) 近年マルチプロセッサシステムにおいて，予想される理論性能に対する得られる実効性能の差の拡大が問題となっている．これを解決するために，従来の並列化コンパイラが利用してきたループ並列性に加え，ループやサブルーチン間の並列性を利用する粗粒度タスク並列処理が重要となる．さらに演算速度とオフチップメモリアクセス速度差が拡大するメモリウォール問題を解決するために，プロセッサ近傍の高速なキャッシュメモリの有効利用が重要である．そこで，OSCAR 自動並列化コンパイラの粗粒度タスク並列処理におけるループ整合分割によってキャッシュメモリサイズ以下に分割された，粗粒度タスク同士のデータ共有量を考慮したスケジューリングを行なうキャッシュ最適化手法を実装した．Sun Ultra80（4 プロセッサ）上での性能評価の結果，SPEC CFP95 ベンチマークの swim に

ついて，Sun Forte HPC 6 update 1 コンパイラの最小処理時間が 60.2 秒であるのに対し，本手法を用いることで 13.2 秒と 4.6 倍の速度向上を得た．同じく tomcatv について，Sun Forte コンパイラが 78.6 秒であるのに対し，本手法を用いることで 33.1 秒と 2.4 倍の速度向上を得た．これらの性能向上から，粗粒度タスク並列性の利用と粗粒度タスク間のキャッシュ最適化の有効性が確かめられた．

(2) 従来のループ整合分割によるデータローカライゼーション手法を用いたローカルメモリ管理では，異なるターゲットループグループ間のデータローカリティの利用や，多次元配列の複数次元にわたる分割は考慮されていなかった．ネストした多重ループの分割や異なるターゲットループグループ間で，同一の配列形状への分割を行なうグローバルループ整合分割を提案した．これにより，従来行なわれなかったターゲットループグループ間でのデータローカリティの利用や一重ネストの分割では，ローカルメモリに収まらない多次元配列のローカルメモリへの割り当てを実現する．

(3) 従来のループ整合分割によるデータローカライゼーション手法を用いたローカルメモリ管理では，ターゲットループグループ内の配列をローカルメモリに固定的に割り当てる．そのため，一時的にしかアクセスされない配列用の領域を，別の配列に割り当てるなどの効率的な利用が行なわれない．これに対し，スケジューリング結果をもとにした配列の定義あるいは参照タイミングに応じた，より柔軟なローカルメモリへの割り当てと解放を行なうローカルメモリ管理手法を提案し，グローバルループ整合分割とともに OSCAR コンパイラに実装した．4 コアのマルチコアである RP1 上での性能評価の結果，メディアベンチマークである AAC エンコーダについて，従来のループ整合分割によるデータローカライゼーション手法を用いたローカルメモリ管理手法の処理時間が 2.8 秒であるのに対し，提案

したローカルメモリ管理手法を用いることで 1.1 秒と 2.6 倍の，8 コアのマルチコア RP2 上では 1.7 秒が 0.7 秒と，2.5 倍の速度向上がそれぞれ得られた．MPEG2 エンコーダについて，4 コアの RP1 上で，従来手法が 23.2 秒であるのに対し，提案手法を用いることで 20.8 秒と 1.1 倍の，8 コアの RP2 上で，10.6 秒が 8.6 秒と 1.2 倍の速度向上がそれぞれ得られた．これらの性能向上から提案手法の有効性が確かめられた．

5.2 今後の課題

本研究に関する今後の課題をまとめる．

(1) 本論文で提案したグローバルループ整合分割では，同一関数内の配列について，形状が同一となるように分割を行なった．プログラム全域にわたるローカルメモリ管理を行なうために，関数を超えて，配列形状を揃える分割手法の提案は課題となる．

(2) 本論文ではマクロタスクでアクセスされる配列範囲を，各次元毎の上下限値として解析を行なった．ストライドアクセスなど，より複雑なアクセスパタン持つ配列を効率良くローカルメモリに割り当てるためには，コンパイラ内部の解析器の精度を向上させるとともに，それらのアクセスパタンに応じたローカルメモリ管理手法の提案は課題となる．

(3) 本論文の評価では，DTU をプロセッサと非同期に動作させ，プロセッサの処理の裏側でデータ転送を行なうことで，データ転送時間の隠蔽を行なうオーバラップ転送は行なっていない．データ転送オーバラップまで考慮したローカルメモリ管理手法の提案は課題となる．

参考文献

[ABC05] F. Angiolini, L. Benini, and A. Caprara. An Efficient Profile-Based Algorithm for Scratchpad Memory Partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 24, No. 11, pp. 1660–1676, 2005.

[IBDD06] Ilya Issenin, Erik Brockmeyer, Bart Durinck, and Nikil Dutt. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pp. 49–52, New York, NY, USA, 2006. ACM.

[IHY+08] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara. An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler. *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 90–91, 598, Feb. 2008.

[KKC+04] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and

I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 12, No. 3, pp. 281–287, 2004.

[KPR+07] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 226–236, New York, NY, USA, 2007. ACM.

[KWN+05] K. Kimura, Y. Wada, H. Nakano, T. Kodaka, J. Shirako, K. Ishizaka, and Hironori Kasahara. Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor. In *Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9)*, Feb 2005.

[KY98] H. Kasahara and A. Yoshida. A Data-Localization Compilation Scheme Using Partial Static Task Assignment for Fortran Coarse Grain Parallel Processing. *Journal of Parallel Computing*, Vol. Special Issue on Languages and Compilers for Parallel Computers, , May 1998.

[LCL99] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proc. 13th ACM SIGARCH International Conference on Supercomputing*, Jun 1999.

[LL01] A. W. Lim and M. S. Lam. Cache optimizations with affine

partitioning. In *Proc. of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 2001.

[LLL01] A. W. Lim, S. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jun 2001.

[LNX07] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. In *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pp. 207–216, New York, NY, USA, 2007. ACM.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia andcommunications systems. In *In 30th Annual IEEE/ACM International Symposium on Microarchitecture*, 1997.

[LWFX07] Lian Li, Hui Wu, Hui Feng, and Jingling Xue. Towards data tiling for whole programs in scratchpad memory allocation. In *Advances in Computer Systems Architecture*, Vol. Volume 4697/2007, pp. 63–74. Springer Berlin / Heidelberg, 2007.

[NIO+02] Hirofumi Nakano, Kazuhisa Ishizaka, Motoki Obata, Keiji Kimura, and Hironori Kasahara. Static coarse grain task scheduling with cache optimization using openmp. In *High Performance Computing, 4th International Symposium, ISHPC 2002, Kansai Science City, Japan, May 15-17, 2002, Proceedings*, Vol. 2327, pp. 479–489. Springer-Verlag, 2002.

[NIO+03] Hirofumi Nakano, Kazuhisa Ishizaka, Motoki Obata, Keiji Kimura, and Hironori Kasahara. Static coarse grain task scheduling with cache optimization using openmp. In *International Journal of Parallel Programming*, Vol. 31, pp. 211–223. Kluwer Academic/Plenum Publishers, Jun 2003.

[PAB+05] D. Pham, S. Asano, M. Bolliger, MN Day, HP Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The design and implementation of a first-generation CELL processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 184–185, 592, 2005.

[PDN97] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *1997 European Design and Test Conference (ED&TC '97)*, p. 7, 1997.

[RGB+06] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pp. 3–8, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[SRM06] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis*

for embedded systems, pp. 401–410, New York, NY, USA, 2006. ACM.

[UB06] Sumesh Udayakumaran and Rajeev Barua. An integrated scratch-pad allocator for affine and non-affine code. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pp. 925–930, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[UDB06] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, Vol. 5, No. 2, pp. 472–511, 2006.

[VKB+99] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. Smarts: exploiting temporal locality and parallelism through vertical execution. In *Proc. of the 1999 international conference on Supercomputing*, Jun 1999.

[YKH+07] Y. Yoshida, T. Kamei, K. Hayase, S. Shibahara, O. Nishii, T. Hattori, A. Hasegawa, M. Takada, N. Irie, K. Uchiyama, et al. A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 100–101, 590, 2007.

[笠原 91] 笠原博徳. 並列処理技術. コロナ社, 1991.

[間瀬 06] 間瀬正啓, 馬場大介, 長山晴美, 田野裕秋, 益浦健, 深津幸二, 宮本孝道, 白子準, 中野啓史, 木村啓二, 笠原博徳. Oscar コンパイラにお

ける制約付き c プログラムの自動並列化. 情報処理学会研究会報告 2006-ARC-170-01(デザインガイア 2006), Nov 2006.

[吉田 99] 吉田明正, 越塚健一, 岡本雅巳, 笠原博徳. 階層型粗粒度並列処理における同一階層内ループ間データローカライゼーション手法. 情報処理学会論文誌, Vol. 40, No. 5, pp. 2054–2063, May 1999.

[白子 04] 白子準, 長澤耕平, 石坂一久, 小幡元樹, 笠原博徳. マルチグレイン並列性向上のための選択的インライン展開手法. 情報処理学会論文誌, Vol. 45, No. 5, pp. 1345–1356, 2004.

謝辞

本研究は，著者が早稲田大学大学院博士課程在学中になされたものである．本研究を進めるにあたり，終始あたたかいご指導を賜わりました笠原博徳教授に心から感謝の意を表します．木村啓二准教授には，本研究に際し様々なご指導を頂きました．深く感謝致します．また，本論文をまとめるにあたり，有益なご助言とご指導をいただきました松山泰男教授，小林哲則教授に感謝の意を表します．

本研究の一部は，STARC「自動並列化コンパイラ協調型シングルチップマルチプロセッサの研究」，METI/NEDO「ミレニアムプロジェクト IT21 アドバンスト並列化コンパイラ」「リアルタイム情報家電用マルチコア」「情報家電用ヘテロジニアスマルチコア」プロジェクト，早稲田大学グローバル COE「アンビエント SoC」および日本学術振興会特別研究員奨励費（課題番号 1501202）により行なわれました．

石坂一久氏（現 NEC）には，直接ミーティングでご指導いただくとともに，残されたソースコードからも数多く学ばせていただきました．ありがとうございました．

本研究の各段階では，小幡元樹氏（現日立），飛田高雄氏（現ソニー），小高剛氏（現東芝），鈴木貴久氏（現富士通），丸山貴紀氏（現パナソニック），内藤陽介氏（現野村総研），仁藤拓実氏（現日立），三浦剛氏（現ソニー），田川友博氏（現

野村総研)，中川正洋氏（現アクセンチュア）をはじめとする笠原・木村研究室の皆様には数多くのご助言，ご協力をいただきました．ありがとうございました．

本論文の成果は，現在の笠原・木村研究室のメンバーである白子準氏，和田康孝氏，宮本孝道氏，鹿野裕明氏，鷹野芙美代氏，間瀬正啓氏，林明宏氏，平瀬吉也氏，高木翔氏，田中裕士氏，見神広紀氏，村田雄太氏，村松裕介氏，八木勇樹氏，園田訓之氏をはじめとする笠原・木村研究室の多くの学生・卒業生の皆様のご協力があってのものです．とくに桃園拓氏には研究・開発の面で大きくご協力いただきました．皆様に深く感謝致します．

最後に，今日までの著者の研究生活を支えていただいた家族に感謝致します．

著者研究業績

種類別	題名	発表掲載誌名	発表年月	著者名
論文				
○	Memory Management for Data Localization on OSCAR Chip Multiprocessor	Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, IEEE Computer Society Press, U.S., pp. 82–88	2004 年 1 月	H. Nakano T. Kodaka K. Kimura H. Kasahara
○	Static Coarse Grain Task Scheduling with Cache Optimization Using OpenMP	International Journal of Parallel Programming, Kluwer Academic/Plenum Publishers, vol. 31, no. 3, pp. 211–223	2003 年 6 月	H. Nakano K. Ishizaka M. Obata K. Kimura H. Kasahara
○	Static Coarse Grain Task Scheduling with Cache Optimization Using OpenMP	Proc. of the 4th International Symposium on High Performance Computing (ISHPC '02), International Workshop on OpenMP: Experiences and Implementations, Lecture Notes in Computer Science (LNCS) of Springer-Verlag, vol. 2327, pp. 479-489	2002 年 1 月	H. Nakano K. Ishizaka M. Obata K. Kimura H. Kasahara
○	マルチコアプロセッサ上での粗粒度タスク並列処理のためのコンパイラによるローカルメモリ管理手法	情報処理学会論文誌コンピューティングシステム（2009 年 1 月 14 日採録決定）	2009 年 4 月	中野 啓史 桃園 拓 間瀬 正啓 木村 啓二 笠原 博徳
	Performance Evaluation of Compiler Controlled Power Saving Scheme	Proc. of 20th ACM International Conference on Supercomputing Workshop on Advanced Low Power Systems(ALPS2006), pp. 362-369	2006 年 6 月	J. Shirako M. Yoshida N. Oshiyama Y. Wada H. Nakano H. Shikano K. Kimura H. Kasahara

種類別	題名	発表掲載誌名	発表年月	著者名
	マルチコアプロセッサにおけるコンパイラ制御低消費電力化手法	情報処理学会論文誌コンピューティングシステム, Vol. 47(ACS15), pp. 147-158	2006 年 9 月	白子 準 吉田 宗弘 押山 直人 和田 康孝 中野 啓史 鹿野 裕明 木村 啓二 笠原 博徳
	チップマルチプロセッサ上での MPEG2 エンコードの並列処理	情報処理学会論文誌, Vol. 46, No. 9, pp.2311-2325	2005 年 9 月	小高 剛 中野 啓史 木村 啓二 笠原 博徳
	Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor	Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9), IEEE Computer Society Press, U.S., pp. 11-20	2005 年 2 月	K. Kimura Y. Wada H. Nakano T. Kodaka J. Shirako K. Ishizaka H. Kasahara
	Parallel Processing using Data Localization for MPEG2 Encoding on OSCAR Chip Multiprocessor	Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, IEEE Computer Society, pp. 119-127.	2004 年 1 月	T. Kodaka H. Nakano K. Kimura H. Kasahara
	共有メモリマルチプロセッサ上でのキャッシュ最適化を考慮した粗粒度タスク並列処理	情報処理学会論文誌, Vol. 43, No. 4, pp. 958-970	2002 年 4 月	石坂 一久 中野 啓史 八木 哲志 小幡 元樹 笠原 博徳
研究会	ヘテロジニアスマルチコア上でのコンパイラによる低消費電力制御	情報処理学会研究会報告 2007-ARC-174-18(SWoPP2007)	2007 年 8 月	林 明宏 伊能 健人 中川 亮 松本 繁 山田 海斗 押山 直人 白子 準 和田 康孝 中野 啓史 鹿野 裕明 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
	ヘテロジニアスマルチコア上での階層的粗粒度タスクスタティックスケジューリング手法	情報処理学会研究会報告 2007-ARC-174-17(SWoPP2007)	2007 年 8 月	和田 康孝 林 明宏 伊能 健人 白子 準 中野 啓史 鹿野 裕明 木村 啓二 笠原 博徳
	情報家電用マルチコア SMP 実行モードにおけるマルチグレイン並列処理	情報処理学会研究会報告 2007-ARC-173-05（第 165 回計算機アーキテクチャ研究会）	2007 年 5 月	間瀬 正啓 馬場 大介 長山 晴美 田野 裕秋 益浦 健 宮本 孝道 白子 準 中野 啓史 木村 啓二 亀井 達也 服部 俊洋 長谷川 淳 佐藤 真琴 伊藤 雅樹 内山 邦男 小高 俊彦 笠原 博徳
	マルチグレイン並列化コンパイラにおけるローカルメモリ管理手法	情報処理学会研究会報告 2007-ARC-172/HPC-109-11 (HOKKE2007)	2007 年 3 月	三浦 剛 田川 友博 村松 裕介 池見 明紀 中川 正洋 中野 啓史 白子 準 木村 啓二 笠原 博徳
	マルチコア上でのマルチメディアアプリケーションの自動並列化	情報処理学会研究会報告 2007-ARC-171-13	2007 年 1 月	宮本 孝道 浅香 沙織 鎌倉 信仁 山内 宏真 間瀬 正啓 白子 準 中野 啓史 木村 啓二 笠原 博徳
	OSCAR マルチコア上でのローカルメモリ管理手法	情報処理学会研究報告 2006-ARC-169-28, pp. 163-168	2006 年 8 月	中野 啓史 仁藤 拓実 丸山 貴紀 中川 正洋 鈴木 裕貴 内藤 陽介 宮本 孝道 和田 康孝 木村 啓二 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
	マルチコアプロセッサ上での粗粒度タスク並列処理におけるデータ転送オーバラップ方式	第 159 回計算機アーキテクチャ・第 105 回ハイパフォーマンスコンピューティング合同研究発表会 (HOKKE-2006)	2006 年 2 月	宮本 孝道 中川 正洋 浅野 尚一郎 内藤 陽介 仁藤 拓実 中野 啓史 木村 啓二 笠原 博徳
	マルチコアプロセッサ上でのデータローカライゼーション	情報処理学会研究会報告 2005-ARC-165-10, pp.51-56	2004 年 12 月	中野 啓史 浅野 尚一郎 内藤 陽介 仁藤 拓実 田川 友博 宮本 孝道 小高 剛 木村 啓二 笠原 博徳
	OSCAR チップマルチプロセッサ上での MPEG2 エンコードの並列処理	情報処理学会研究会報告 2004-ARC-160-07	2004 年 12 月	小高 剛 中野 啓史 木村 啓二 笠原 博徳
	OSCAR チップマルチプロセッサ上でのデータ転送ユニットを用いたデータローカライゼーション	情報処理学会研究会報告 2004-ARC-159-20	2004 年 7 月	中野 啓史 内藤 陽介 鈴木 貴久 小高 剛 石坂 一久 木村 啓二 笠原 博徳
	データローカライゼーションを伴う MPEG2 エンコーディングの並列処理	情報処理学会研究会報告 2004-ARC-156-3	2004 年 2 月	小高 剛 中野 啓史 木村 啓二 笠原 博徳
	OSCAR CMP 上でのスタティックスケジューリングを用いたデータローカライゼーション手法	情報処理学会研究会報告 2003-ARC-154-14	2003 年 8 月	中野 啓史 小高 剛 木村 啓二 笠原 博徳
	OSCAR マルチプロセッサシステム上での MPEG2 エンコーディングの並列処理	情報処理学会研究会報告 2003-ARC-154-10	2003 年 8 月	小高 剛 中野 啓史 木村 啓二 笠原 博徳
	チップマルチプロセッサ上での粗粒度タスク並列処理によるデータローカライゼーション	情報処理学会研究会報告 ARC2003-151-3 (SHINING2003)	2003 年 1 月	中野 啓史 小高 剛 木村 啓二 笠原 博徳
	ラインコンフリクトミスを考慮した粗粒度タスク間キャッシュ最適化	情報処理学会研究報告 ARC2002-149-25(SWoPP2002)	2002 年 8 月	石坂 一久 中野 啓史 小幡 元樹 笠原 博徳

種類別	題名	発表掲載誌名	発表年月	著者名
	共有メモリマルチプロセッサ上でのデータローカライゼーション対象マクロタスク決定手法	情報処理学会研究報告 ARC	2002 年 3 月	八木 哲志 板垣 裕樹 中野 啓史 石坂 一久 小幡 元樹 吉田 明正 笠原 博徳
	商用 SMP 上での粗粒度タスク並列処理	情報処理学会研究報告 ARC, ARC2002-146-10	2002 年 2 月	小幡 元樹 石坂 一久 神長 浩気 中野 啓史 吉田 明正 笠原 博徳
	キャッシュ最適化を考慮したマルチプロセッサシステム上での粗粒度タスクスタティックスケジューリング手法	情報処理学会研究報告 ARC2001-140-12	2001 年 8 月	中野 啓史 石坂 一久 小幡 元樹 木村 啓二 笠原 博徳
全国大会				
	マルチプロセッサシステム上でのキャッシュ最適化を考慮した粗粒度タスクスタティックスケジューリング手法	情報処理学会第 62 回全国大会, 4R-02	2001 年 3 月	中野 啓史 石坂 一久 小幡 元樹 木村 啓二 笠原 博徳
特許				
	コンパイル方法、コンパイラ、およびコンパイル装置	特許第 4177681 号	2008 年 8 月	笠原 博徳 石坂 一久 中野 啓史 小幡 元樹
	メモリ管理方法、情報処理装置、プログラムの作成方法及びプログラム	特願 2007-050269 特開 2008-217134 PCT/JP2008/053891	2007 年 2 月 2008 年 9 月 2008 年 2 月	笠原 博徳 木村 啓二 中野 啓史 仁藤 拓実 丸山 貴紀 三浦 剛 田川 友博