計算機アーキテクチャ 149ー25
（２００２．８．２３）

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

石坂一久[†,††]　中野啓史[†]
小幡元樹[†,††]　笠原博徳[†,††]

[†] 早稲田大学　[††] アドバンスト並列化コンパイラプロジェクト

プロセッサの高速化に伴うメインメモリとの速度差の増大により，キャッシュの有効利用は実効性能の向上に重要な役割を占めるようになっている．本論文では，プログラムを基本ブロック，ループ，サブルーチンといった粗粒度タスクに分割し，それらの間の並列性を利用する粗粒度タスク並列処理における，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手法について述べる．本手法では，キャッシュサイズを考慮して複数のループを整合分割することによって，分割後のループがアクセスするデータサイズがキャッシュに収まるようにした後，各分割ループを粗粒度タスクと定義し，同一データを使用する粗粒度タスクを同一プロセッサ上で可能なかぎり連続に実行することのより，複数ループ間でキャッシュの有効利用を図る．さらに，連続実行される粗粒度タスク集合がアクセスするデータに対して，定義された配列サイズを拡大する方式のパディングを用いたデータレイアウトの変更によりラインコンフリクトミスの削減を行う． 本手法の性能評価を Sun Ultra80 上で spec95 の swim 用いて行った．合計キャッシュサイズが 16MB となる 4PE での実行では，swim の約 13MB のデータセットはパディングによるコンフリクトミスの削減により，ほとんどがキャッシュ上に収まるため，Forte のみを用いた場合の 4PE での最小処理に対して，本手法により 6.02 倍の性能向上が得られた．一方，データサイズがキャッシュサイズより大きい場合の 1PE での実行では，粗粒度タスク間キャッシュ最適化とパディングの併用することにより処理時間は 79.1 秒となり，パディングのみを用いた Forte の逐次実行時間 93.5 秒に対して 18.2%，OSCAR による粗粒度タスク間キャッシュ最適化のみの処理時間 90.1 秒に対しては 13.9% の性能向上が得られることがわかり，両者を組み合わせて適用する本手法の有効性が確かめられた．また，RS6000 SP 604e 上では，本手法での 8PE の処理時間は 52.0 秒と，粗粒度タスク間キャッシュ最適化のみを適用した場合の 8PE の処理時間 59.2 秒と比べ 14%向上し，XLF コンパイラが 8PE までで最も良い値を出した 6PE の 108.0 秒に対して 2.08 倍の性能向上が得られた．

Cache Optimization among Coarse Grain Tasks considering Line Conflict Miss

Kazuhisa Ishizaka[†,††], Hirofumi Nakano[†], Motoki Obata[†,††]
and Hironori Kasahara [†,††]

[†]Waseda University　[††]Advanced Parallelizing Compiler Project

Effective use of cache is getting important with the increase of the speed gap between processors and memories. In this paper, cache optimization for coarse grain task parallel processing is described. Coarse grain task parallel processing uses the parallelism among coarse grain tasks such like basic blocks, loops and subroutines to increase effective performance of multiprocessor. In the proposed cache optimization, loops are decomposed to the small loops which access smaller data than cache size. Moreover, these loops are executed as consecutively as possible on the same processor to use cache effectively for data transfer among loops. In addition, the proposed cache optimization eliminates conflict misses among the data used in macro tasks which are consecutively executed on same processor by intra-variable padding which changes array dimension size. The proposed scheme is evaluated on Sun Ultra80 using spec95 swim. The performance of cache optimization among macro tasks (10.0s) gave us 10 times speedup against the sequential execution (99.8s) by elimination of conflict misses for 4 processors on which all data can be put on cache after padding because total cache size exceeds data size. Total speedup using padding and cache optimization among macro tasks (79.1s) is 18% against Sun Forte compiler on single processor (93.5s). Also, in the evaluation on IBM RS6000 SP 604e, the proposed scheme improve the performance of coarse grain task parallel processing by 14% (59.2s to 52.0s) for 8pe, and gave us 2.08 times speedup against XLF compiler for 6pe which gave us the best performance (108.0s).

1　はじめに

マルチプロセッサシステムの実効性能を向上させるためには，従来から利用されてきたループ並列処理に加え，基本ブロック，ループ，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，ステートメント間の並列性を利用する近細粒度並列処理を組み合わせたマルチグレイン並列処理が重要である．また，増大するプロセッサとメインメモリの速度差を隠蔽し，プロセッサへのデータ供給性能を向上させるために，キャッシュを有効利用することも実効性能の向上に重要となっている．

キャッシュを用いデータローカリティを有効利用するため，ループ融合，ループパーミュテーション，ブロッキングなどのループを対象としたプログラムリストラクチャリングや，パディング，配列の転置などのデータレイアウト変換の研究が行われてきた [1)~3)]．

プログラムの実効性能の向上には，キャッシュの利用効率を高めること，すなわちキャッシュのミス回数を削減することが重要である．キャッシュミスは，初めてデータにアクセスする際に起るコンパルソリーミス（compulsory miss），データサイズがキャッシュサイズを越えることによって起るキャパシティーミス（capacity miss），キャッシュのアソシアティビティが足りないため他のアドレスのデータによってキャッシュから追い出されるコンフリクトミス（conflict miss）の三種類に分類される．

コンパルソリーミスは回数が少なく，多くのキャッシュはフルアソシタティブではないことから，特にダイレクトマップなどアソシアティビティが低いキャッシュにおいて

は，コンフリクトミスの削減が重要であることが知られている[4]．コンフリクトミスを削減する手法としては，変数の宣言サイズを変更する変数内パディング（intra-variable padding），複数変数の間にダミー変数を入れる変数間パディング（inter-variable padding）などのデータレイアウト変換による手法[2),5)]などが研究されている．また，物理アドレスキャッシュにおいては，オペレーティングシステムによる論理アドレスと物理アドレスの変換がコンフリクトミス回数に重要な影響をおよぼすことが知られており，ハードウェアと OS が協調してコンフリクトミスを減らす手法[4]やコンパイラと OS が協調する手法[6]などが提案されている．

一方，マルチプロセッサシステムの実効性能を向上させるためのマルチグレイン並列処理の粗粒度タスク並列処理においても，OSCAR アーキテクチャのローカルメモリを対象としたデータローカライゼーション手法[7]が研究されてきた．データローカライゼーション手法を SMP マシン上のキャッシュに応用し，データを共有する粗粒度タスクを同一プロセッサ上で連続実行させることによって，粗粒度タスク間でのデータ転送にキャッシュを有効利用する手法が筆者らによって提案されている[8]．

本論文では，データローカライゼーションにより連続実行される粗粒度タスクがアクセスするデータ間でのコンフリクトミスを削減するために，変数内パディングを用いたデータレイアウト変換を適用した粗粒度タスク間キャッシュ最適化手法について述べる．従来のコンパイラによるコンフリクトミスの削減は主に単一ループを対象としているのに対し，本手法では複数のループ間でのローカルティの有効利用を行う粗粒度タスク間キャッシュ最適化と組み合わせることにより，複数ループ間でのコンフリクトミスの削減を可能とする．

以下，本論文では 2 章で粗粒度タスク並列処理，3 章で粗粒度タスク間キャッシュ最適化について述べ，4 章でコンフリクトミス削減について述べる．5 章で性能評価について述べ，6 章でまとめる．

2　粗粒度タスク並列処理

粗粒度タスク並列処理ではソースプログラムを，単一の基本ブロック，スケジューリングオーバーヘッドを考慮し複数の小基本ブロックを融合あるいは並列性向上のため単一の基本ブロックを分割して生成される疑似代入文ブロック（BPA），DO ループまたは IF 文による分岐によって生成されるループ，すなわち最外側ナチュラルループからなる繰り返しブロック（RB），サブルーチンブロック（SB）に分割する．

さらに，繰り返しブロック RB の内，データ依存等により DOALL 処理ができないシーケンシャルループのループボディ部や IF 文による分岐で作られるループの内部，またコード長などを考慮しインライン展開を行なわなかったサブルーチンの内部に対しては，階層的にその内部をサブマクロタスクに分割する．

OSCAR マルチグレインコンパイラにおける，粗粒度タスク並列処理の手順は次のようになる．

1. FORTRAN ソースプログラムを階層的に分割してマクロタスクを生成.
2. マクロタスク間のコントロールフロー，データ依存を解析した後に，各マクロタスクの最早実行可能条件を解析しマクロタスクグラフを生成.
3. 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は，スタティックスケジューリングによって，マクロタスクをプロセッサに割り当て，スケジューリング結果にしたがって並列化コードを生成．一方，マクロタスクが条件分岐等の実行時不確定性を持つ場合は，コンパイラはダイナミックスケジューリングルーチンを生成し，生成する並列化コードの中にマクロタスクコードと共に埋め込む.

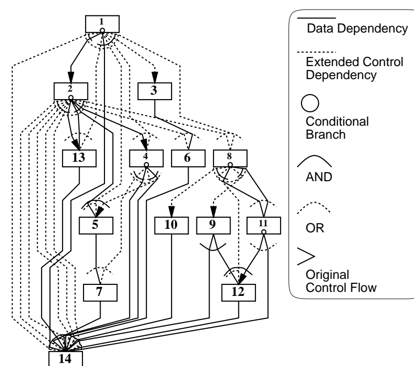

図 1: マクロタスクグラフ

2.1　マクロタスクグラフの生成

次にコンパイラは，生成された各階層において，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た後に，マクロタスク間の並列性を抽出するために，各マクロタスクに対して最早実行可能条件解析を行う．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もっとも早い時点で実行可能になる条件である．

各階層のマクロタスクの最早実行可能条件は，図 1 に示すようなマクロタスクグラフ（MTG）で表わされる．マクロタスクグラフ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．また，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，データ依存と制御依存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，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．

2.2　マクロタスクスケジューリング

粗粒度タスク並列処理では，各階層のマクロタスクのプロセッサへの割り当てには，スタティックスケジューリングまたはダイナミックスケジューリングが用いられる．

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しかもたない場合に適用され，コンパイル時にマクロタスクのプロセッサへの割り当てを決定し，スケジューリング結果に従って各プロセッサ毎に異なった並列化コードを生成する．

一方，マクロタスクグラフが条件分岐などの実行時不確定性を持つ場合は，実行時にマクロタスクを割り当てるダイナミックスケジューリングを用いる．ダイナミックスケジューリングを適用した場合，コンパイラは実行時スケジューリングのためのスケジューリングルーチンを生成し，マクロタスクコードと共に出力する並列化コードに埋め込む．OSCAR コンパイラでは，ダイナミックスケジューリングルーチンをユーザープログラムとして生成し，それを粒度の大きな粗粒度タスクのスケジューリングに用いることによって，実行時スケジューリングのオーバーヘッドを相対的に低くしている．

また，ダイナミックスケジューリングとしては，一つのプロセッサがスケジューリング専用となり，その他のプロセッサがマクロタスクの実行を行う集中ダイナミック

スケジューリングと，全てのプロセッサにスケジューリング機能を分散し，各プロセッサが共有スケジューリング情報に排他的にアクセスすることによってスケジューリングを行なうと共にマクロタスクの実行も行う，分散ダイナミックスケジューリングを用いることができる．

3　粗粒度タスク間キャッシュ最適化

本章では，キャッシュを有効に利用により，粗粒度タスク並列処理の性能を向上させる手法について述べる．

プログラムの持つローカリティを利用して粗粒度タスク並列処理の性能を向上させる手法として，筆者等は従来よりデータローカライゼーション手法 [7] を提案している．同手法は，コンパイラにより制御可能な OSCAR アーキテクチャのローカルメモリ及び分散共有メモリを対象としている．しかし，本手法が対象とする主記憶共有型マルチプロセッサは，ハードウェアでコントロールされるキャッシュを備えている．このため，本論文では従来のローカライズ手法のように共有データの生死解析によるデータのリプレースを自由に制御する方式ではなく，キャッシュの LRU リプレース方式でも有効に働くデータ分散及びスケジューリング手法を用いる．

3.1　ループ整合分割

キャッシュサイズと比較して大きなデータを使うループでは，そのループの最初の方のイタレーションでアクセスされたデータは，最後の方のイタレーションの実行時にキャッシュから追い出されてしまっている可能性があるため，後続の同一配列データにアクセスする他のループとの間でキャッシュの効率良い利用ができない．このようなキャッシュ利用効率の低下を避けるため，粗粒度タスク間キャッシュ最適化では，まずデータ使用量の多いループに対してループ整合分割 [7] を用いて，複数のループ間のデータ依存を考慮しながら，アクセスするデータがキャッシュサイズより小さくなるような小ループに分割する．

図 2（a）に示すマクロタスクグラフで，マクロタスク 2，3，7 がキャッシュサイズより大きなデータを使用する並列処理可能ループであるとする．これらのループに対してループ整合分割を適用して，それぞれ 4 つの小ループに分割したときのマクロタスクグラフが図 2（b）である．例えば図 2（a）のマクロタスク 2 は，図 2（b）のマクロタスク 2_A，2_B，2_C，2_D に分割されている．

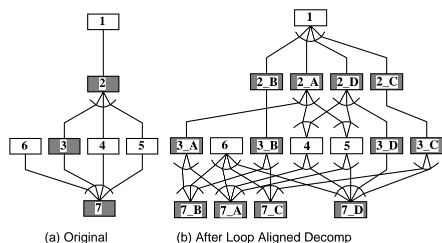

図 2: ループ整合分割

ループ整合分割によって生成されたループもまたマクロタスクとして扱われ，さらにマクロタスクグラフ上でデータ依存エッジで結ばれたデータ共有量の多いマクロタスク群はデータローカライゼーショングループ（DLG）にグループ化される．図 2（b）上で同じサフィックスを持つマクロタスクが同一の DLG にグループ化されている．たとえば，マクロタスク 2_A，3_A，7_A は DLG_A にグループ化されている．

3.2　パーシャルスタティック割り当て

図 2（a）のマクロタスクグラフのオリジナルプログラム上でのマクロタスクの実行順番は，マクロタスク番号の増加順である．分割後のマクロタスクグラフで表すと，ループ整合分割で生成されたマクロタスクのオリジナルプログラム上での実行順はアルファベット順であるので，例えばマクロタスク 2_A から 3_D までのオリジナルプログラム上での実行順は，2_A, 2_B, 2_C, 2_D，3_A 3_B, 3_C, 3_D となる．一方，図 2（b）の分割後のマクロタスクグラフで表されている最早実行可能条件を見ると，例えばマクロタスク 3_B はマクロタスク 2_B のみにデータ依存するため，2_B の実行終了直後に 3_B の実行を行うことが可能であることが分かる．

粗粒度タスク間キャッシュ最適化では，粗粒度タスク間のスケジューリングにおいて，最早実行可能条件と DLG を利用し，同一 DLG に属するマクロタスクを可能な限り同一プロセッサ上で連続実行し，キャッシュの利用効率を向上するようにプログラムの実行順が変更される．同一 DLG に属するマクロタスクを同じプロセッサに連続的に割り当てるためのダイナミックスケジューリングの拡張を，“パーシャルスタティック割り当て” [8] と呼ぶ．

今回の性能評価では，ダイナミックスケジューリングの方が条件分岐を含むマクロタスクグラフにも適用でき汎用性が高いこと，評価に使用したマシンのプロセッサ台数が少ないことを考慮して，全プロセッサをマクロタスクの実行に使用できる分散ダイナミックスケジューリング方式を使用する．

図 2（b）に示すマクロタスクグラフをシングルプロセッサに対して提案手法でスケジューリングを行った例を図 3 に示す．オリジナルプログラムでの実行順では，同一 DLG に含まれるマクロタスクは連続して実行されないため，キャッシュ利用効率が悪が，提案手法を適用した場合は，図 3 の DLG_B に含まれるマクロタスク 2_B，3_B，7_B や DLG_C の 2_C，3_C，7_C などに示されるように，同一 DLG に含まれるマクロタスクが連続して実行されるため，キャッシュを効果的に利用することができる．

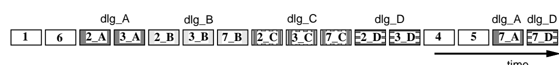

図 3: シングルプロセッサへのスケジューリング例

4　コンフリクトミス削減

ダイレクトマップなどのアソシアティビティの低いキャッシュにおいては，コンフリクトミスを考慮することが重要となる．本章では，粗粒度タスク間キャッシュ最適化における配列次元サイズ拡張パディングを用いたコンフリクトミス削減について述べる．

従来より単一ループ，もしくは複数のループを融合したループに対するコンフリクトミスの削減について研究されている．同一イタレーションで使用する配列間でコンフリクトミスが発生することによって，キャッシュラインによる空間的ローカリティの利用効果を打ち消してしまう深刻なコンフリクトミス（severe conflict miss）を削減するためのパディングや，外側ループのイタレーションで使用したデータを次のイタレーションで使用する場合（group reuse）に，それらの間でデータがキャッシュから追い出されないようにするためのパディングなどが研究されている [2]．

従来のパディングはコンフリクトミス削減により単一ループ内でのローカリティの向上を目的とするのに対し，

本手法では DLG に含まれる複数のループ間でアクセスされるデータに対して，コンフリクトミスの削減を行うことにより，複数ループ間でのデータローカリティの向上を行う．

4.1 DLG 内でのコンフリクトミスの削減

単一ループを対象としたパディングを用いても，ループ内で使用するデータキャッシュサイズを越える場合は，そのループの実行によってデータの追い出しが起るため，複数ループ間での時間的ローカリティを利用することはできない．

粗粒度タスク間キャッシュ最適化では，ループ分割を用いることで，同一 DLG 内のループで使用するデータを全てキャッシュに格納することが可能となる．したがって，同一 DLG 内ループ集合での共有データ間でコンフリクトミスを削減することにより，複数ループ間での時間的ローカリティを利用することが可能となる．

Sun Ultra80 上で spec95 の swim 実行する場合を例に考える．Swim は real 型の 513 × 513 要素の約 1MB の二次元配列を 13 個使用する．13 個の配列はコモン文で宣言され，仮想アドレス上に連続に割り当てられる．Ultra80 は 4MB のダイレクトマップの L2 キャッシュを持つ Ultra Sparc II を 4 台備えている．

図 4 に，swim の各配列を 4MB のダイレクトマップのキャッシュに割り当てた場合のイメージを示す．ここでは，仮想アドレスとキャッシュサイズからキャッシュ上のアドレスを求めている．Ultra Sparc II の L2 キャッシュは physically-indexed-physically-tagged であるが，これについては後述する．図 4 中各太線のボックスが各配列を表し，ボックス中の文字列は変数名を表す．水平方向はキャッシュサイズを表し，先頭アドレスからキャッシュサイズ（4MB）を越える毎に一段下げて図示してある．すなわち，垂直方向に同一の位置にあるアドレスは同一セットに割り当てられ，コンフリクトする可能性があることを示している．

また，キャッシュコンフリクトを考える場合，各配列間の距離に着目すれば良いので，先頭配列（図中 U）の先頭アドレスを，キャッシュの先頭（図中左端）として図示した．図では縮尺および説明の関係上，変数 U, VNEW, POLD, H の様にいくつかの配列の先頭アドレスがキャッシュの同一セットに割り当てられているように示しているが，実際の swim では各配列は real 型の 513 × 513 要素の二次元配列なので，正確には 1MB よりも大きいため，キャッシュ上での各配列の先頭アドレスはずれる．その距離はラインサイズより大きいため，同一セットには割り当てられない．したがって，この例の Ultra80 上の swim では，同一イタレーションで使用する配列間でのコンフリクトミス（severe conflict miss）は起らない．

また，図中点線は，プログラム中のループを 4 分割した場合の各 DLG でアクセスされるデータの分割位置を示しており，灰色に塗ってある部分は 4 分割して生成した DLG0 内のループがアクセスする範囲を示している．図から分かるように，同一 DLG のループにおいて使用する変数間でコンフリクトミスが起る配置となっている．

このようなコンフリクトミスを避けるように，配列開始アドレスをコンフリクトが起こらない位置までずらすために配列宣言サイズを拡張する変数内パディング（intra-variable padding）によってデータレイアウトを変換した場合のキャッシュイメージを図 5 に示す．ここでは，各配列の先頭アドレスを変更することが目的であり，Fortran は Column-major order であることから配列の二次元目

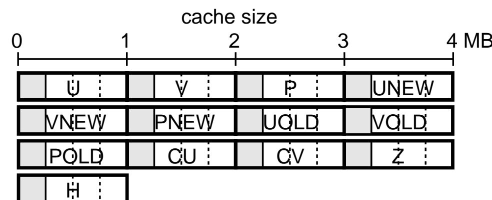

図 4: swim のキャッシュイメージ

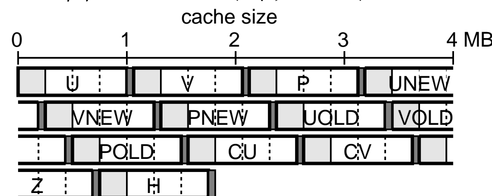

図 5: intra-variable padding 後のキャッシュイメージ

にパディングをし，各配列の宣言サイズを 513 × 544 に変更している．図の各配列間の濃い灰色部分がパディングされた部分である．図中の薄い灰色部分で示された DLG0 の使用範囲がキャッシュ上に広く分散されていることが示されている．このように本手法では複数のループに渡りキャッシュを有効利用できるように，同一データにアクセスするループ集合をキャッシュサイズにフィットするように整合分割し，分割後の同一配列にアクセスするループ集合をデータローカライゼーショングループ DLG とし，その DLG で使用される各配列がコンフリクトミスを起こさないように配列の宣言サイズを拡張するという方式により，図 5 のように，各 DLG で使用するデータはそれぞれコンフリクトミスの起らない位置に配置され，DLG 間でのローカリティを向上させることができる．

4.2 論理/物理アドレス変換による影響

前節では論理アドレス上でコンフリクトミスの削減を考えたが，Ultra Sparc II の L2 キャッシュのように，physically-indexed キャッシュでは，物理アドレスでキャッシングされるため，オペレーティングシステムの論理アドレスと物理アドレスのマッピングアルゴリズムが重要な影響を及ぼす．オペレーティングシステムでのアドレス変換を利用することによって，コンフリクトミスを削減することができることが知られている [9]．

今回の性能評価で用いた Solaris 8 では，論理アドレスからハッシュを用いて物理アドレスを決定するアルゴリズム（Hashed VA），物理アドレスを論理アドレスに直接マッピングするアルゴリズム（P.Addr = V.Addr）および Bin Hoppoing の三つのアルゴリズムが利用できる [10]．Bin Hopping のような論理アドレス上で連続するアドレスが物理アドレス上では連続しないマッピングアルゴリズムでは，一般的には論理アドレス上でのデータレイアウト変換は意味を持たなくなる．P.addr = V.addr 方式では，論理アドレス上での連続アドレスは物理アドレス上でも連続アドレスとなり，前節で示したキャッシュイメージと同じモデルである．したがって，論理アドレス上でのデータレイアウト変換は有効である．Solaris 8 のデフォルトのアルゴリズムである Hashed VA では，大部分で論理アドレス上の連続アドレスは物理アドレス上でも連続となるが，論理アドレス上で L2 キャッシュサイズ境界をまたぐときに，L1 キャッシュとページサイズを考慮して少量のギャップを入れることによって，論理アドレス上で L2 キャッシュサイズだけ離れたアドレス間でのコンフリクトが避けられている．この方式でも，ギャップ

のサイズが小さいことと，キャッシュサイズの境界をまたがない大半の部分では論理アドレス上での連続性は保たれることから，論理アドレス上でのデータレイアウト変換は有効である．

5　性能評価

本節では Sun Ultra80 および IBM RS6000 SP 604e 上で spec95 の swim を用いて行った本手法の性能評価について述べる．Ultra80 は 450MHz の Ultra Sparc II を 4 台持ち，各プロセッサの L1 キャッシュは命令 16KB，データ 16KB，L2 キャッシュは Direct Map 方式で 4MB である．OS は Solaris 8，ネイティブコンパイラには Forte 6 update 1 を用いた．使用した RS6000 は，200MHz の PowerPC 604e を 8 台持ち，L1 キャッシュは命令 16K，データ 16K で，L2 キャッシュは 1MB である．ネイティブコンパイラは XL Fortran ver. 7.1，OS は AIX 4.3.3 である．コンパイルオプションは最小処理時間を与えるものを選び，コンパイラオプション以外の特別なチューニングは行っていない．

5.1　評価方法

データローカライゼーションをもちいた粗粒度タスク間キャッシュ最適化は OSCAR コンパライを用いて行い，その結果を OpenMP コードとして出力し，その後ネイティブコンパイラを用いて実行ファイルを生成した [8]．粗粒度タスク間キャッシュ最適化を行わないときの逐次実行はネイティブコンパイラを単体で使用し，並列実行は Forte コンパイラの自動並列化機能を用いた．また，swim に対して粗粒度キャッシュ間最適化を適用する場合は，インライン展開を適用したが，今回の性能評価では OSCAR コンパイラの OpenMP 出力をネイティブコンパイラがソースコードサイズの増大のためにコンパイルできないという問題が生じたため，インライン展開をしたループを手動で一つのサブルーチンに戻したものを入力ファイルとして用いた [8]．パディングを適用する際はソースコードの配列宣言サイズを変更することによって行い，それをネイティブおよび OSCAR コンパイラの入力とした．

性能評価は，粗粒度タスク間キャッシュ最適化を適用した場合としない場合，パディングを適用した場合としない場合に加え，オペレーティングシステムの論理アドレスと物理アドレスの変換方式を変えた場合についても行った．また，swim で使用する 13MB のデータセットが使用プロセッサの合計 L2 キャッシュサイズに収まる 4PE による実行と，データセットが L2 キャッシュサイズを越える 1PE による実行を行い，両者を比較する．

5.2　評価結果

Solaris 8 のデフォルトである Hashed VA 方式の場合の評価結果を表 1 に示す．表 1 の一番左の列は説明用の通し番号である．その次の “compile” 列は，粗粒度タスク間キャッシュ最適化を適用したかどうかを示し，“forte” はキャッシュ最適化を適用せず forte コンパイラのみでコンパイルした場合，“localize” は OSCAR コンパイラによるキャッシュ最適化を適用した場合である．“pad ” 列は 4 章で述べたパディングを用いたか否かを示している．また，“pe” 列は評価に使用したプロセッサ台数，“time” 列は実行時間，“L2 miss” 列は L2 ミス回数である．“speedup” 列は 1PE 数のパディング，粗粒度タスク間キャッシュ最適化を用いない場合に対する速度向上率である．

まず，パディング単体の性能向上を見ると，1PE でパディングを適用しなかった場合の Forte の性能が 99.8 秒（no.1）から 93.5 秒（no.2）へ 7%，4PE 時には 60.2 秒（no.5）から 10.1 秒（no.6）へ 596%の速度向上が得られた．1PE 時に比べ 4PE 時に非常に高い性能向上を示しているが，これは swim のデータサイズば 13MB であるため，4PE 時には各 PE の使用するデータサイズは約 3.3MB となり，パディングによりコンフリクトミスが解消されると，全てのデータがキャッシュに収まり，プログラム全体にわたる非常に高い時間的ローカリティが得られるためである． 表 1 中の対応する L2 キャッシュミス回数（no.5，no.6）を見ると，パディングによりミス回数はパディングをしない場合の 3.2%に削減されており，ほとんどのコンフリクトミスが削減されたことが分かる．

表 1: Hashed VA 方式での評価結果

no.	compile	pad	pe	time sec	L2 miss $\times 10^6$	speed up
1	forte	なし	1pe	99.8	344.3	1.00
2	forte	あり	1pe	93.5	344.7	1.07
3	localize	なし	1pe	90.1	219.6	1.11
4	localize	あり	1pe	79.1	203.4	1.26
5	forte	なし	4pe	60.2	303.0	1.66
6	forte	あり	4pe	10.1	11.1	9.88
7	localize	なし	4pe	47.1	198.7	2.12
8	localize	あり	4pe	10.0	12.7	9.98

次にローカライズを単体で適用した場合は，1PE 時に 90.1 秒（no.3），4PE 時には 47.1 秒（no.7）と，Forte 単体（no.1，no.5）と比べ，それぞれ 10.7%，27.5%の性能向上が得られた．さらにパディングを合わせて適用することで，1PE 時に 79.1 秒（no.4），4PE 時に 10.0 秒（no.8）となり，パディングによるコンフリクトミスの削減の効果が確認された．パディングの適用により粗粒度タスク間キャッシュ最適化の性能は，1PE 時に 13.9%向上（no.3 から no.4），4PE 時に 471%向上（no.7 から no.8）した．4PE 時が高い性能向上を示すのは，パディングを Forte コンパイラに単体で適用した場合と同様に，各 PE の使用するデータがすべてキャッシュに収まるためである．

また，パディングを Forte コンパイラに単体で適用した場合に比べ，粗粒度タスク間キャッシュ最適化も合わせて適用した場合は，1PE 時に 18.2%の性能向上（no.2 から no.4）が得られたのに対して，4PE 時には性能向上は得られなかった（no.6 から no.8）．1PE 時で粗粒度タスク間キャッシュ最適化を行わない場合は，データセットがキャッシュサイズに収まらないため，複数ループ間での時間的ローカリティの有効利用ができない．しかし，粗粒度タスク間キャッシュ最適化を適用した場合は，ループ整合分割とパーシャルスタティックスケジューリングにより DLG 内ループの連続実行が可能となり，DLG 内ループ間で時間的ローカリティが効果的に利用できるため，Forte コンパイラにパディングのみの場合に対して性能向上が得られた．逆に 4PE 時には前述したようにパディングにより全てのデータがキャッシュに収まるため，粗粒度タスク間キャッシュ最適化を用いなくても十分に時間的ローカリティが利用されているため，両者を組み合わせることによる性能向上は得られない．

次に，オペレーティングシステムの論理/物理アドレス変換を変えた場合の評価結果を表 2 に示す．“time” 列および “L2miss” 列中の “v=p” 列が V.addr=P.addr 方式，“bin” 列が Bin Hoppoing 方式の場合である．表 2 の V.addr=P.addr 方式を用いた場合と表 1 の Hashed VA 方式の場合を比較すると，ほぼ同等の性能が得られていることが分かり，4.2 節で述べたように，Solaris のデフォルトの方式である Hashed VA 方式でも本手法が有効であることが確かめられた．また Bin Hopping 方式の場合

表 2: V.adder=P.addr，Bin Hopping 方式での評価結果

compile	pad	pe	time sec		L2 miss $\times 10^6$	
			v=p	bin	v=p	bin
forte	なし	1pe	100.5	107.0	379.5	344.4
forte	あり	1pe	93.8	96.9	360.6	345.0
localize	なし	1pe	93.2	92.1	232.0	218.3
localize	あり	1pe	79.3	81.6	236.9	202.1
forte	なし	4pe	64.5	35.5	182.1	323.4
forte	あり	4pe	9.84	35.9	185.1	9.0
localize	なし	4pe	52.8	30.6	144.5	221.7
localize	あり	4pe	9.83	29.3	131.5	9.1

でパディングおよび粗粒度タスク間キャッシュ最適化の両方を適用しない場合の 4PE の実行時間は 35.9 秒であり，Hashed VA 方式の場合の同条件の 60.2 秒に比べて 67.7%の性能向上が得られている．これは Hashed VA 方式の場合はパディングをしない場合にはコンフリクトミスのため，キャッシュの利用効率が悪いが，Bin Hoppoing 方式では論理アドレス上の連続アドレスがが物理アドレス上では連続にならないため，Hashed VA 方式に比べコンフリクトミスが削減されたためと考えられる．逆に，Bin Hoppoing 方式ではコンパイル時にコンフリクトミスを避けるような最適化を行うのが難しいため，ローカライゼーションによるキャッシュ最適化及びパディングとも性能向上は見られるが，Hashed VA 方式方式のような処理時間の短縮は得られない．

5.2.1 RS6000 上での評価結果

本手法の性能評価を IBM RS6000 SP 604e High Node 上で行った結果を表 3 に示す．“compile” 列の “xlf” は XLF コンパイラのみを用いたキャッシュ最適化を適用しない場合を示している．また，RS6000 上では L2 ミス回数を計測するツールを入手できなかったので表 3 では示していない．XLF コンパイラのみを用いた場合は 8pe 時が最速でなかったため，最小処理時間を与えた PE でのデータを示した．XLF にパディングを適用することにより処理時間は 103.6 秒となり，逐次処理 523.5 秒に比べ 5.05 倍（no.10）の速度向上が得られた．また OSCAR で粗粒度タスク間キャッシュ最適化を行うことにより 59.3 秒となり，逐次に比べ 8.83 倍（no.11）の速度向上が得らた．また，パディングと粗粒度タスク間キャッシュ最適化を合わせて適用した場合の処理時間は 52.0 上であり，逐次処理に対し 10.07 倍の速度向上が得られ，粗粒度タスク間キャッシュ最適化のみに対して 14%（no.11 から no.12）の性能向上が得られた．

表 3: RS6000 での評価結果

no.	compile	pad	pe	time sec	speedup
9	xlf	なし	6pe	108.0	4.85
10	xlf	あり	5pe	103.6	5.05
11	localize	なし	8pe	59.3	8.83
12	localize	あり	8pe	52.0	10.07

6 まとめ

マルチプロセッサシステムの実効性能の向上には，ループ並列性に加え粗粒度タスク間の並列性を利用すること及びメモリレイテンシの隠蔽のためのメモリ階層の有効利用が重要である．本論文では，粗粒度タスク間キャッシュ最適化手法，およびその性能を向上させるためのコンフリクトミスの削減手法について述べた．

本手法では複数のループに渡りキャッシュを有効利用するため，同一データにアクセスするループ集合をキャッシュサイズにフィットするように整合分割し，分割後の同一配列にアクセスするループ集合をデータローカライゼーショングループ DLG とし，DLG で使用される各配列がコンフリクトミスを起こさないように配列の宣言サイズを拡張する方式のパディング適用した上で，DLG 内ループを同一プロセッサ上で連続実行してキャッシュの有効利用する．

本手法の性能評価を Sun Ultra80 上で spec95 の swim を用いて，データセットが使用 PE の合計キャッシュサイズに収まる場合と越える場合について行った．合計キャッシュサイズが 16MB となる 4PE での実行では，swim の約 13MB のデータセットはパディングによるコンフリクトミスの削減により，ほとんどがキャッシュ上に収まるため，Forte のみを用いた場合の 4PE での処理時間 60.2 秒に対して，本手法では 10.0 秒となり 6.02 倍の性能向上が得られた．一方，データサイズがキャッシュサイズより大きい場合の 1PE での実行では，粗粒度タスク間キャッシュ最適化とパディングの併用することにより処理時間は 79.1 秒となり，パディングのみを用いた Forte の逐次実行時間 93.5 秒に対して 18.2%，OSCAR による粗粒度タスク間キャッシュ最適化のみの処理時間 90.1 秒に対しては 13.9% の性能向上が得られることがわかり，両者を組み合わせて適用する本手法の有効性が確かめられた．また，RS6000 SP 604e 上では，本手法での 8PE の処理時間は 52.0 秒と，粗粒度タスク間キャッシュ最適化のみを適用した場合の 8PE の処理時間 59.2 秒と比べ 14%向上し，XLF コンパイラが 8PE までで最も良い値を出した 6PE の 108.0 秒に対して 2.08 倍の速度向上が得られた．

参考文献

[1] Anderson, J. M., Amarasinghe, S. P. and Lam, M. S.: Data and Computation Transformations for Multiprocessors, *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing* (1995).

[2] Rivera, G. and Tseng, C.-W.: Eliminating Conflict Misses for High Performance Architectures, *Proc. of the 1998 ACM International Conference on Supercomputing* (1998).

[3] Lim, A. W., Cheong, G. I. and Lam, M. S.: An Affine Partitoning Algorithm to Maximize Parallelism and Minimize Communication, *Proc. of the 13th ACM SIGARCH International Conference on Supercomputing* (1999).

[4] Bershad, B. N., Lee, D., Romer, T. H. and Chen, J. B.: Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches, *Proc. of the Sixth Internatinal Symposium of Architectural Support for Programing Languages and Operating Systems* (1994).

[5] Bacon, D. F., Chow, J.-H., ching R. Ju, D., Muthukumar, K. and Sarkar, V.: A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness, *Proc. of CASCON '94 conference* (1994).

[6] Bugnion, E., Anderson, J. M., Mowry, T. C., Rosenblum, M. R. and Lam, M. S.: Compiler-Directed Page Coloring for Multiprocessors, *Proc. of the Seventh Internatinal Symposium of Architectural Support for Programing Languages and Operating Systems* (1996).

[7] 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 (1999).

[8] 石坂, 中野, 八木, 小幡, 笠原: 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 (2002).

[9] Kessler, R. E. and Hill, M. D.: Page Placement Algorithms for Large Real-Indexed Caches, *ACM transaction of Computer Systems* (1992).

[10] Mauro, J. and McDougall, R.: *SOLARIS Internal*, Prentice hall.