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における配列間パディング

石坂一久[†]　小幡元樹[††]　笠原博徳[†]

[†]早稲田大学　[††]日立製作所

マルチプロセッサシステムの普及に伴い，自動並列化コンパイラの重要性がますます高まっている．マルチプロセッサシステムの実効性能をこれまで以上に向上させるには，従来のループ並列処理に加え，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階層的に利用するマルチグレイン並列処理が重要になっている．またデータローカリティ最適化によりキャッシュなどのプロセッサに近接した高速メモリを有効利用することも性能向上に重要である．本論文では粗粒度タスク並列処理時のキャッシュ最適化のためのデータローカライゼーション手法における，コンフリクトミス削減を目的とした配列間パディング手法について述べる．本手法の性能を Sun Ultra 80 上で SPEC CFP95 を用いて評価した結果では，Sun Forte コンパイラの自動並列化に対して，最大 5.5 倍の性能向上が得られることが確かめられた．

Inter-Array Padding for Data Localization with Static Scheduling

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

[†]Waseda University　[††]Hitachi Ltd.

The growth of multiprocessor system usage makes automatic parallelizing compilers more important. To improve the performance of multiprocessor system, multigrain parallelization is important. In multigrain parallelization, coarse grain task parallelism among loops and subroutines and near fine grain parallelism among statements are used in addition to the traditional loop parallelism. Also, locality optimization to use cache effectively is also important for the performance improvement. This paper describes inter-array padding for data localization to minimize cache conflict misses. In the evaluation on Sun Ultra 80 using SPEC CFP95, the OSCAR multigrain compiler gave us up to 5.5 times speedup against Sun Forte automatic paralleling compiler.

1　はじめに

現在マルチプロセッサアーキテクチャはハイパフォーマンスコンピュータをはじめエントリーレベルサーバー，デスクトップワークステーション，ゲーム等で幅広く利用されるようになっている．そのようなマルチプロセッサシステムの実効性能やユーザーの使い易さを向上させるには，高性能の自動並列化コンパイラが重要で，従来から多くの研究が行われいると共に，すでに多くの商用自動並列化コンパイラが販売されている．しかし，マルチプロセッサシステムの最高性能と実効性能の差は拡大しており，さらに実効性能を向上させるためには，従来から自動並列コンパイラで利用されているループ並列性に加え，ループ，サブルーチンなどの粗粒度タスク間の並列性や，ステートメント間の並列性を利用するマルチグレイン並列処理が重要となる．

また，プロセッサの速度向上によるメモリとの速度差の増大や，共有メモリへのアクセス集中による速度低下を避けるため，キャッシュなどのプロセッサに近接した小容量で高速なメモリを有効利用することは性能向上に欠かすことができない．キャッシュ最適化に関する研究は幅広く研究が行われてきた．OS のページ変換アルゴリズムによりキャッシュ最適化を行う手法 [1] や，ページカラーリングにコンパイラからの情報を利用する手法 [2] など OS を利用するものや，コンパイラがデータレイアウトを変換する手法 [3),4] などが提案されている．

筆者らは，従来より粗粒度タスク並列処理においてデータローカリティを向上させるデータローカライゼーション手法を提案している．データローカライゼーションでは，まず複数のループ間のデータ依存を解析し，各ループを分割したときにデータ依存している複数ループ間でのデータローカリティが最大となるようにループを分割する．次に分割により生成された小ループ（タスクとして扱う）の最早実行可能条件を解析し，分割ループ間の実行順序をキャッシュの利用効率が向上するように変更する．本論文では，この実行順序の最適化に加えて，パディングを用いたデータレイアウト変換を適用することによりラインコンフリクトミスを削減する手法について述べる．従来のコンパイラによるデータレイアウト変換は，単一ループもしくはループフュージョン後のループを対象としているのに対し，本手法はデータローカライゼーションと組み合わせることより，

複数ループ間でのキャッシュ最適化を可能とする．

以下本論文では，2 章でデータローカライゼーション手法，3 章でデータレイアウト変換手法について，4 章で本手法の性能評価について述べる．

2　データローカライゼーション

本節では粗粒度タスク並列処理およびその性能を向上させるデータローカライゼーション手法について述べる．

2.1　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）の三種類の粗粒度タスク（マクロタスク）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．

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．

2.2　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，そのループの実行中にループ前半でアクセスしたデータを自分自身でキャッシュから追い出してしまうため，同一配列データにアクセスする後続ループの実行時にキャッシュミスが生じてしまい，キャッシュの効率的な利用ができない．このようなキャッシュ利用効率の低下を避けるため，データローカライゼーションでは，まずデータアクセス量の多いループに対してループ整合分割 [5] 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 1（a）に示すマクロタスクグラフは，最早実行可能条件を表したもので，マクロタスク間の並列性を示す．グラフ中の各ノードがマクロタスクを表し，ノードを結ぶエッジはデータ依存を表す．エッジを結ぶアークはデータ依存が AND 関係にあることを示す．なお，マクロタスクグラフはコントロール依存も表すことができるが，この例ではデータ依存のみである．

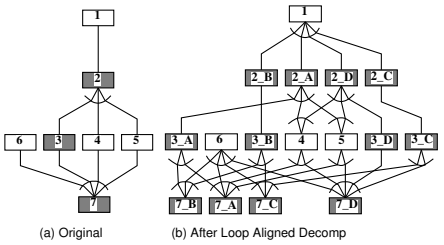

図 1: ループ整合分割

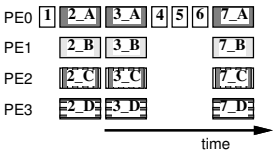

図 2: 4PE へのスケジューリング例

図 1（a）のマクロタスク 2，3，7 がキャッシュサイズより大きなデータにアクセスする並列処理可能ループであるときに，これらのループに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 1（b）である．例えば図 1（a）のマクロタスク 2 は，図 1（b）のマクロタスク 2_A，2_B，2_C，2_D に分割されている．

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，最早実行可能条件が解析される．さらに分割後の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複数のマクロタスクはデータローカライザブルグループ（DLG）にグループ化される．DLG は同一のプロセッサに割り当てられ，キャッシュを利用してデータの授受を行うべきマクロタスクの集合である．図 1（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 は同一 DLG にグループ化されている．

2.3　マクロタスクスケジューリング

マクロタスクは最早実行可能条件を用いて，コンパイル時のスタティックスケジューリング，もしくはコンパイラが生成してユーザコード中に埋め込んだダイナミックスケジューリングルーチンによる実行時スケジューリングによってプロセッサに割り当てられる．

マクロタスクスケジューリング時には，ループ整合分割によって生成された DLG を考慮して，同一 DLG に含まれるマクロタスクを可能な限り同一プロセッサ上に連続に割り当てる [6),7)]．

図 1（b）に示すマクロタスクグラフを 4PE に対して，スケジューリングを行った例を図 2 に示す．

図に示すように同一の DLG に含まれるマクロタスクが同一プロセッサに割り当てられており，キャッシュを利用したデータ転送が可能になる．なお，図ではマクロタスク 4,5,6 を PE0 で逐次実行され負荷バランスが悪いように見えるが，実際には他のマクロタスクと比べて 4,5,6 の実行時間は非常に小さいため問題にならない．また，この例では簡単のため DLG 数と PE 数は同一であるが，データサイズが大きい場合は PE 数以上の数の DLG が生成され，複数の DLG が一つの PE に割り当てられる．この場合も各 DLG 内のマクロタスクは最早実行可能条件にしたがって可能な限り連続実行される．

3 コンフリクトミス削減

本節では，同一 DLG に属するマクロタスク間でのコンフリクトミスを削減するための，パディングを用いたデータレイアウト変換手法について述べる．

3.1 DLG 内キャッシュコンフリクト

データローカライゼーションでは，一 DLG がアクセスするデータ（以下 DLG 内データ）のサイズがキャッシュサイズに収まるようにループを分割し，同一 DLG に属するマクロタスク（分割後のループ）は同一プロセッサ上で連続実行される．したがって，キャパシティミスが起こる前にデータの再利用を行うことが可能となるが，それらのマクロタスクでアクセスされるデータ間がキャッシュ上の同一セットに割り当てられる場合は，ダイレクトマップなど連想度が小さいキャッシュでは，DLG 内データサイズがキャッシュサイズより小さいにも関わらず，DLG 内データでコンフリクトミスが発生する．

例えば，図 3 に SPEC CFP95 の swim の 513x513 要素の各 1MB の二次元配列 13 個を 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．図 3 中の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ラインに割り当てられ，コンフリクトする可能性があることを示している．また，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そのうち 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 のループでアクセス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は，DLG 内マクロタスクの連続実行によるデータ再利用を無効にしてしまう．そこでパディングを用いることでデータレイアウトを変換して，DLG 内データ間でのコンフリクトミスを削減することにより，複数ループ間に渡るキャッシュ最適化を可能にする．以下では，DLG 内データのコンフリクトを削減するためのパディング手法について述べる．

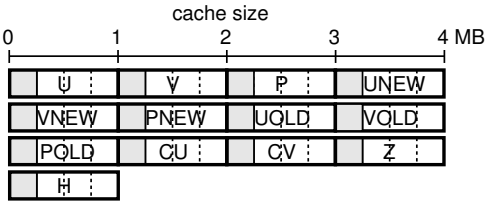

図 3: swim のキャッシュイメージ

3.2 配列間パディング

本節ではローカル変数の配列，もしくは全てのモジュールで宣言の形が同じコモン領域に属する配列に対するパディング手法を述べる．これらの配列に対しては，配列の宣言サイズを越えてアクセスがされない場合は，配列間にパディングを行ってもプログラムの意味を壊さない．そこで，配列間にパディングを行うことによって，同一ラインに割り当てられていた配列間の相対位置をずらしてコンフリクトミスを削減する．以下にその手順を述べる．

対象配列グループの選択　本手法を実装した OSCAR コンパイラは逐次 FORTRAN プログラムを並列化して OpenMP FORTRAN プログラムを出力する．その後ターゲットマシン上のネイティブコンパイラによって実行ファイルを生成する．したがって，実際の配列のメモリ割り当てはネイティブコンパイラが行うため，OSCAR コンパイラは実際に配列をどのような順番でメモリに割り当てるかを決定することはできない．そこで同一の大きさの配列をパディング対象に選ぶことによって，メモリ割り当ての順番による影響を受けない複数の配列を対象にパディングを行う．すなわち，実際にネイティブコンパイラによって異る順番に割り当てられたとしても，その場合は配列名を読み変えれば良い．

また，コモン領域はメモリ上での並び順が決められるため，コモン領域に属する配列のメモリ割り当て順を知ることはできるが，今回は実装を簡単にするため全モジュールで宣言の形が等しく，かつコモン領域内の全配列の大きさが等しい場合は，ローカル変数と同様にして扱った．モジュール間で形状が異るコモン領域に対するパディングは次節で述べる．

キャッシュ割り当てイメージの計算　選択されたグループ内の配列のキャッシュ上での相対アドレスを求め，キャッシュ上での各配列の割り当てイメージを求める．ここでは前述のように大きさの等しい配列を対象としているため，図 3 に示すように，単純に宣言順に連続に割り当てていくことでアドレスを求める．

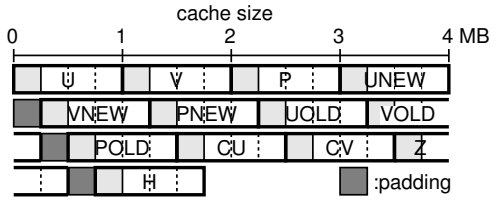

図 4: swim のパディング位置

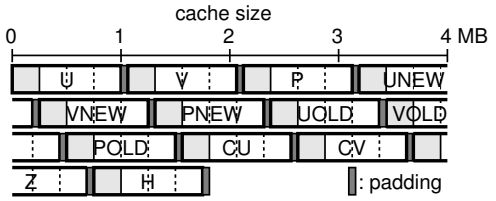

図 5: 各配列サイズの変更によるパディング

最小分割数の決定　対象配列の合計サイズをキャッシュサイズで割ることにより，一 DLG で利用されるデータサイズがキャッシュサイズ以下になるための，各ループの最小の分割数 div_num を求める．

図 3 にあげた例では，総データサイズ 13 MB に対してキャッシュサイズは 4MB なので $div\_num = ceil(13/4) = 4$ である．

最大 DLG アクセスサイズの決定　各ループでは各配列を先頭から末尾へ均等にアクセスすると想定すると，配列サイズを div_num で割ったサイズ $part\_size$ が一つの DLG によってアクセスされる各配列の最大の大きさになる．

コンフリクト判定　キャッシュイメージ上で各配列の $part\_size$（図 3 の薄い灰色部分）に重なりがある場合は，同一 DLG でアクセスされる部分配列間でコンフリクトが起る．なお，div_num は最小の分割数であるため，それ以上の分割数の場合でも $part\_size$ 部分に重なりがなければコンフリクトは起らない．コンフリクトが起らない場合は手順は終了し，パディングは行わない．図 3 の例では，$part\_size$ 部分に重なりがあるため，コンフリクトが起る．

パディングサイズの決定　重なり部分をなくすためには，各配列をキャッシュ上に割り当てていったときに，キャッシュサイズを越えて折り返した次の配列（図 3 では配列 VNEW）のキャッシュ上での先頭アドレスが，先頭配列（図 3 では U）の先頭アドレス（すなわちキャッシュの先頭）から $part\_size$（図 3 の薄い灰色部分）だけ離れていれば良い．

したがって，先頭配列から順番に割り付けていった際に，キャッシュサイズ（$cache\_size$）を越えた次の配列の先頭アドレス（$base\_address$）が，($cache\_size$+$part\_size$) より小さい場合は，($cache\_size$+$part\_size$−$base\_address$) の大きさのパディングを行うことにより，キャッシュ上での配列の相対位置を $part\_size$ だけずらし，同一 DLG アクセス部分でのコンフリクトを消す．以下同様に順番に配列を割り当てていき，次のキャッシュサイズでの折り返し時にも同様にパディングを行う．

図 3 に対して求められたパディングを図 4 に示す．図 4 では，キャッシュサイズで折り返した配列 UNEW, VOLD, Z の後ろにパディング（濃い灰色部分）を行っている．

配列サイズの変更　ソースプログラム上でのパディングの実現には配列をコモン領域に入れ，ダミーの配列によってパディングを行う手法も考えられるが，ローカル変数をコモン変数にすることによる最適化への悪影響を避けるため，本手法では配列サイズを変更する．

前述したように各配列のメモリ上での順番がソース上での並び順にはならない可能性もあるため，図 4 の様に特定の配列の後ろにのみパディングを行うように，その配列のサイズを変更するのではなく，全ての配列にパディングサイズを等しく分散させるように，各配列の大きさの変更を行う．この場合並び順が変わったとしても，各配列に行ったパディングを合計すると，キャッシュサイズで折り返す位置に求めた大きさのパディング行われることになる．

具体的には，先頭配列から $cache\_size$+$part\_size$ までの間に含まれる配列数を $narrays$ とすると，各配列を $padding\_size/narrays$ だけ大きくするように，配列の最遅変化次元（FORTRAN の場合は最大次元）の次元数を増やす．図 5 に各配列のサイズを変更させてパディングを実現させた場合のキャッシュイメージを示す．

3.3　コモン領域間パディング

コモン領域はモジュール毎に形状が異ることがある．このようなコモン領域に属する配列に対して，配列間にパディングを挿入するとプログラムの意味を変えてしまう可能性があるため，パディングを行うのは困難である．

例えば SPEC CFP95 の hydro2d のコモン領域の一つ VARH は図 6 に示すように二つサブルーチンでは形状が異る．ここで MP, NP はパラメータ文で定義されており，全て同一の値である．この例では，サブルーチン TISTEP の先頭配列 VZ は 514560 バイトであるのに対して，サブルーチン TRANS1 での先頭配列 VZ1 は 515840 バイトである．

このように，コモン領域の形がサブルーチン間で異るため，前節で述べた配列間パディングは適用できないが，図 6 を見ると VARH はどちらのサブルーチンでも，ほぼ同じ大きさの四つの配列から構成されていることが分かる．同様に hydro2d の他のコモン領域 VAR1, VAR2, SCRA もそれぞれほぼ VARH のそれと同じ大きさの四つの配列から構成されている．こられのコモン領域を宣言の順番に

```
  SUBROUTINE TISTEP
    COMMON /VARH/VZ(MP,NP),VR(MP,NP),
&    PR(MP,NP),TST(MN),DU3(4*MP+4*NP+4)
  SUBROUTINE TRANS1
    COMMON /VARH/VZ1(0:MP,NP),VR1(0:MP,NP),
&    PR1(0:MP,NP),FL1(0:MP,NP),DU3(4*MP+4)
```

図 6: 宣言形状の異るコモン領域（hydro2d）

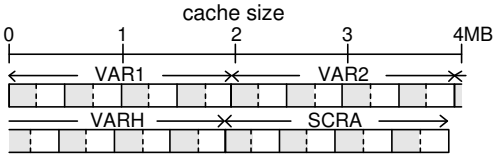

図 7: hydro2d のキャッシュイメージ

VAR1, VAR2, VARH, SCRA の順にメモリ割り当てを行った際の，4MB のダイレクトマップのキャッシュ上でのイメージは図 7 の様になる．図中の薄い灰色部分は，コモン領域を構成する各配列を 2 分割したうちの前半の部分配列を示している．ここで分割数に 2 を選んだのは，四つのコモン領域の合計サイズが約 7.9MB であるため，2 分割するとキャッシュサイズに収まるからである．

コモン領域を構成する各配列がプログラム中のループで先頭から末尾へと均等にアクセスすると想定すると，各ループを 2 分割した場合の前半ループ（すなわち DLG0）のアクセスする部分配列は，図 7 の薄い灰色部分になる．したがって，図で重なりがあることから分かるように，DLG 間データのコンフリクトが起ることになる．

実際のメモリ割り当ては，ネイティブコンパイラが行うので必ずしもソースコード上での宣言順に割り当てられるわけではないが，もしこの順で割り当てられた場合はコンフリクトミスが起る．そこで，本手法ではネイティブコンパイラにコモン領域の割り当て順を任せず，これらのコモン領域（小コモン領域）を結合して一つのコモン領域を生成することで，小コモン領域の割り当て順を決定する．また同時に，小コモン領域間にダミー配列を挿入することによってパディングを行い，小コモン領域を構成する配列間でのコンフリクトミスを防ぐ．

以下にコモン領域間パディングの手順を示す．

配列サイズの推定　本手法ではほぼ同じ大きさの配列で構成されているコモン領域を対象にするため，まず各コモン領域が大きさのほぼ等しい配列のみから構成されているかどうかを判定する．構成されている場合，その配列の大きさを *common_array_size* とする．

簡単のためプログラム中の実行文でのコモン領域のメンバ変数のアクセスパターンを調べずに，プログラムの全モジュールでの変数宣言部のみ調べて求

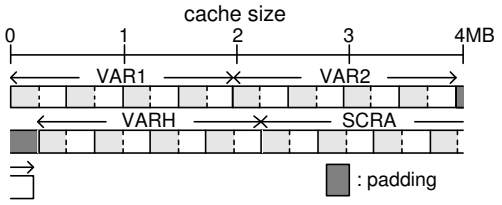

図 8: hydro2d でのコモン領域間パディング

める．コモン領域のメンバになっている配列のうち，ダミー変数（すなわちループ等の実行文で利用されない）と考えられる (1) コモン領域全体と同じ大きさの配列，(2) コモン領域の最後のメンバの配列を除いた全配列の大きさが，一定の差（現在の実装では 5%）以内に収まる場合に，その配列の大きさを *common_array_size* とする．また，コモン領域中のスカラー変数はサイズ小さいので無視する．

対象コモン領域グループ選択　大きさのほぼ等しい配列から構成されているコモン領域中で，*common_array_size* がほぼ等しいコモン領域をグループ化して，コモン領域間パディングの対象とする．

hydro2d のコモン領域の内 VAR1, VAR2, VARH, SCRA はほぼ同一の大きさの 4 つの配列から構成されており，対象コモン領域グループに選ばれる．

パディングサイズの決定　コモン領域の合計サイズをキャッシュサイズで割って *div_num* を求め，また *common_array_size* を *div_num* で割って *part_size* を求める．

配列間パディングと同様に，コモン領域グループに属するコモン領域をキャッシュ上に割り当てたイメージを求め，キャッシュサイズをまたぐコモン領域（図 7 の VARH）の先頭アドレスが *cache_size*+*part_size* となるように，コモン領域間（図 7 の VAR2, VARH 間）にパディングを行う．

hydro2d にコモン領域間パディングを適用したキャッシュイメージを図 8 に示す．

4　性能評価

本節では，SPEC CFP95 の tomcatv, swim, hydro2d, turb3d を用いて Sun Ultra 80 上で行った本手法の性能評価について述べる．Ultra 80 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 キャッシュはダイレクトマップ方式で 4MB である．メインメモリは 1GB で，OS は Solaris 8，ネイティブコンパイラには Forte 6 update 2 を用いた．

本評価では，OSCAR コンパイラによるパディングを適用した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の性能を，用いない場合の性能および Forte コンパイラの自動並列化と比較した．また，tomcatv, swim, hydro2d については SPEC CFP95 のオリジナルのソースコードをそ

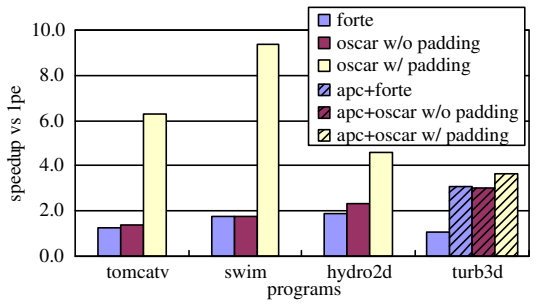

図 9: Sun Ultra 80（4PE）上での性能

のまま用いたが，turb3d については，OSCAR コンパイラ，Forte コンパイラともにループ並列性を抽出できないため，オリジナルソースコードをそのまま Forte コンパイラに通したものに加えて，APC コンパイラ [8] により並列化した OpenMP FORTRAN を入力として用いた評価も行った．

パディングありの OSCAR コンパイラでは，tomcatv にはローカル変数に対する配列間パディング，swim, turb3d には全モジュールで同一形状のコモン領域のメンバ配列に対する配列間パディング，hydro2d にはコモン領域間パディングが適用された．

評価結果を図 9 に示す．図は Forte のみよる逐次実行時間に対する速度向上率を示している．評価結果よりパディングを行わない際の OSCAR コンパイラは，Forte コンパイラとほぼ同等の性能となっているが，パディングを行わない場合はラインコンフリクトミスによりデータローカライゼーションの効果が得られないため，ループ並列性が中心の今回のプログラムに対して，両コンパイラともほぼ同様のループ並列性を利用したために性能差がでない．

Tomcatv, swim はパディングを行うことにより，パディングを行わない場合に対してそれぞれ 4.7 倍，5.5 倍と高い性能向上が得られているが，これは tomcatv, swim のデータサイズは 4 プロセッサの合計キャッシュサイズより小さいため，キャッシュ最適化によりほぼ全てのデータがキャッシュにのるためである．一方コモン領域間パディングを適用した hydro2d は，これに比べて速度向上率は低いが，パディングにより 2.0 倍の性能向上がえられた．また trub3d は配列を先頭から順番にアクセスしていくループと，とびとびにストライドアクセスしていくループがあり，本手法は前者のループを対象とするが，後者のループとは同一プロセッサがアクセスする配列部分が異るためキャッシュの利用効率が低下する．したがってパディングによる性能向上は他の 3 プログラムと比較して低くなっているものの，APC コンパイラを利用した OSCAR コンパイラはパディングを行わない場合に対して 1.2 倍，Forte 単体に対しては 3.6 倍の性能向上が得られた．

5 まとめ

本論文では，データローカライゼーション手法において，コンフリクトミスを削減するための配列間パディング手法について述べた．本手法では，複数のループをキャッシュサイズを考慮して分割し，分割後のループを可能な限り同一プロセッサ上で連続実行させることにより，複数ループ間におけるテンポラルローカリティを向上させる．さらに連続実行されるループ間でのコンフリクトミスを配列間パディングによって削減することによって，キャッシュの利用効率を向上させる．本手法の Sun Ultra 80 上での性能評価では，Sun Forte コンパイラの自動並列化に対して，4 プロセッサ使用時に SPEC CFP95 tomcatv で 5.1 倍，swim で 5.5 倍，hydro2d で 2.5 倍，turb3d で 1.2 倍の速度向上が得られた．

本論文ではプログラム中での配列のアクセスパターンを調べず，配列が先頭から均等にアクセスされることを想定して，宣言サイズのみを見てヒューリスティックにパディングを行った．今後の課題としてはより多くのプログラムを調査してその仮定の妥当性を検証するとともに，ループ整合分割部との協調によってパディングサイズを求める手法を考案することが挙げられる．

謝辞 本研究の一部はミレニアムプロジェクト Advanced Parallelizing Compiler の支援により行われた．

参考文献

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

[2] Bugnion, E., Anderson, J. M., Mowry, T. C., Rosenblum, M. R. and Lam, M. S.: Compiler-Directed Page Coloring for Multiprocessors, *Proc. of ASPLOS* (1996).

[3] Manjikian, N. and Abdelrahman, T. S.: Array Data Layout for the Reduction of Cache Conflicts, *Proc. of PDCS* (1995).

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

[5] 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).

[6] 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).

[7] 中野, 石坂, 小幡, 木村, 笠原: 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 ARC 研究報告 (2001).

[8] APC: Advanced Parallelizing Compiler Project, *http://www.apc.waseda.ac.jp/*.