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

石 坂 一 久[†,††] 中 野 啓 史[†] 八 木 哲 志[†]
小 幡 元 樹[†,††] 笠 原 博 徳[†,††]

主記憶共有型マルチプロセッサシステムは幅広く使われているが，プロセッサ数の増加に伴いその最大性能と実効性能の差が拡大してきている．このような問題を解決するためには，従来のループ並列処理に加えて，粗粒度タスク並列性，近細粒度並列性等のマルチグレイン並列性の利用が重要な技術である．また，プロセッサ技術の進歩と共に，プロセッサとメモリの速度差が顕在化し，その速度差を補うためのメモリ階層，特にキャッシュの有効利用は，マルチプロセッサシステムの性能向上に重要な要因となっている．本論文では，ループ並列化に加え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（マクロタスク）に分割し，それらの間の並列性を効果的に利用すると共に，従来 OSCAR 型マルチプロセッサアーキテクチャにおけるローカルメモリおよび分散共有メモリ用に提案されていたデータローカライゼーション手法を主記憶共有型マルチプロセッサ上のキャッシュ最適化用に発展させたデータ分散・ダイナミックスケジューリング手法を提案する．本手法は OSCAR マルチグレインコンパイラ上で実現され，逐次 FOTRAN プログラムを入力すると，共有メモリマシンにおける標準 API である OpenMP を用いて複数マクロタスク間でキャッシュ上の共有データを再利用する並列化コードを自動生成する．本手法の評価を商用 SMP マシンである IBM RS/6000 604e High Node，Sun Ultra80 上で spec95fp ベンチマークの tomcatv，swim，mgrid を用いて行った結果，IBM RS/6000 上では IBM XL FORTRAN version 6.1 コンパイラの自動ループ並列化を行った場合の最小実行時間に対して本手法は最大 5.8 倍の性能向上を示し，Sun Ultra80 上では Sun Forte 6 update 1 コンパイラの最小時間に対して最大 3.6 倍の性能向上が得られることが確かめられた．

Coarse Grain Task Parallel Processing with Cache Optimization on Shared Memory Multiprocessor

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

In multiprocessor systems, the gap between peak and effective performance has getting larger. To cope with this performance gap, it is important to use multigrain parallelism in addition to ordinary loop level parallelism. Also, effective use of memory hierarchy is important for the performance improvement of multiprocessor systems because the speed gap between processors and memories is getting larger. This paper describes coarse grain task parallel processing that uses parallelism among macro-tasks like loops and subroutines considering cache optimization using data localization scheme. The proposed scheme is implemented on OSCAR automatic multigrain parallelizing compiler. OSCAR compiler generates OpenMP FORTRAN program realizing the proposed scheme from an ordinary FORTRAN77 program. Its performance is evaluated on IBM RS6000 SP 604e High Node 8 processors SMP machine and Sun Ultra80 4 processors SMP machine. In the evaluation, OSCAR compiler gives us up to 5.8 times speedup against the minimum execution time of IBM XL FORTRAN compiler on IBM RS/6000 and up to 3.6 times speedup against Sun Forte 6 update 1 compiler on Sun Ultra80.

1. はじめに

主記憶共有型マルチプロセッサシステムは，シングルチップマルチプロセッサからワークステーション，ハ

† 早稲田大学
Waseda University
†† Advanced Parallelizing Compiler Project

イパフォーマンスコンピュータまで幅広く利用されている．しかし，マルチプロセッサシステムの最大性能と実効性能の差は，プロセッサ数の増加と共に拡大しており，実効性能を向上させるためには，従来のループ並列処理の限界を越えた並列処理技術が必要となっている．また，プロセッサ速度の進歩により，メモリとの速度差が拡大し，主記憶へのアクセスレイテンシが性能向上のボトルネックとなっている．そこで，マルチプロセッサシステムの実効性能向上には，タスク並列性の利用や階層メモリの有効利用などの最適化が重要となる．しかし，それらの最適化には並列処理，階層メモリに対するデータ分散，マルチプロセッサスケジューリングなどの高い専門知識が必要とされる．したがって，マルチプロセッサシステムの実効性能，使い易さ，および価格性能比を向上させるためには，従来からのループ並列処理に加えて粗粒度タスク並列処理を実現し，かつメモリ階層の最適化を行う自動並列化コンパイラが求められる．

従来の自動並列化コンパイラは，ループ並列化コンパイラが主流であり，強力なデータ依存解析とプログラムリストラクチャリング手法を用いたループ並列化に重点がおかれている．イリノイ大学の Polaris コンパイラ[1]は，シンボリック解析，Array Privatization[2]，run-timeデータ依存解析[3]などを用いてループ並列性を抽出する．スタンフォード大学の SUIF コンパイラ[4]は，インタープロシージャ解析，ユニモジュラ変換[5]，Affine Partitioning を用いたキャッシュ最適化[6),7]などを用いたループ並列処理を行っている．また，データ局所性の最適化に関しては Blocking，Padding，Data Localization などのプログラムリストラクチャリングを用いた研究が行われている[8),9]．

しかし，これらのループ並列化技術は成熟期に達しているため，今後大幅な性能向上は望めない．したがって，さら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，従来のループ並列化に加えて，ループ間やサブルーチン間といった粗粒度レベルの並列性や，基本ブロック内での命令・ステートメント間の並列性を利用する必要がある．Parafrase2 をベースとするカタルーニャ大学の NANOS コンパイラは，拡張した OpenMP API によって粗粒度並列性を含むマルチレベル並列性を抽出しようとしている．また，PROMIS コンパイラ[10]は，フロントエンドとバックエンドで共通の中間表現を用いてループ並列性と命令レベル並列性を統合しようとしている．

また，OSCAR マルチグレイン並列化コンパイラは，ステートメント間の並列性を利用した近細粒度並列処理，基本ブロックやサブルーチン・ループ間の並列性を利用した粗粒度タスク並列処理を従来のループ並列化手法を組み合わせたマルチグレイン並列処理を実現している[11]．

本論文では，粗粒度タスク並列処理および，データローカライゼーション手法[12]を利用した粗粒度タスク間のキャッシュ最適化手法について述べる．本手法は OSCAR マルチグレインコンパイラに実装されており，今回の性能評価における共有メモリマシン上での実現では，逐次 FORTRAN プログラムから粗粒度タスク並列処理を実現する共有メモリマシン用並列化標準 API OpenMP FORTRAN プログラムを生成する．

以下，2 章では OSCAR コンパイラでの粗粒度タスク並列化手法およびその OpenMP を用いた実現方法について，3 章では粗粒度タスク並列処理に適用するキャッシュ最適化手法について述べる．4 章で本手法の性能評価について述べ，5 章でまとめを述べる．

2. 粗粒度タスク並列処理

本章では粗粒度タスク並列処理について述べる．粗粒度タスク並列処理では，ソースプログラムを以下の三つのマクロタスクに分割し，それらの間の並列性を利用する．

疑似代入文ブロック Block of Pseudo Assignments（BPA）．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したブロック，または並列性向上のため単一の基本ブロックを分割して得られるブロック

繰り返しブロック Repetition Block（RB）．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

サブルーチンブロック Subroutine Block（SB）．コード長の増大などを考慮した結果インライン展開が有効に適用できないと判断したサブルーチン

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，マクロフローグラフを生成する．
(3) 各マクロタスクの最早実行可能条件[13),14]を解析し，マクロタスクグラフを生成する．
(4) 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出力する．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．この場合，マクロタスクはダイナミックスケジューリングによって実行時にプロセッサに割り当てられる．

2.1　マクロタスク生成

粗粒度タスク並列処理では，コンパイラはソースプログラムを前述の 3 種類の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．また，3 章で述べるように，キャッシュ最適化を行なう場合，ループはプロセッサ数やキャッシュサイズを考慮して，複数の小ループに分割される．生成された小ループはマクロタスクとして扱われ，粗粒度タスク並列処理において，イタレーション間の並列性がこのループ分割により粗粒度タスク並列性として利用され，キャッシュ最適化が適用される．

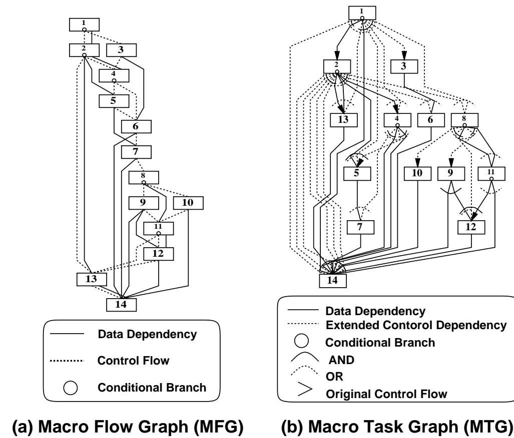
2.2　マクロフローグラフの生成

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する．解析結果は図 1（a）に示されるような，無サイクル有効グラフであるマクロフローグラフ（MFG）で表わされる．マクロフローグラフにおいて，ノードはマクロタスクを表わし，ノード内の小円は条件分岐を表わしている．また，実線はデータ依存，点線はコントロールフローを表わしている．マクロフローグラフ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きを仮定している．

2.3　マクロタスクグラフの生成

マクロフローグラフからマクロタスク間の並列性を抽出するために，コンパイラは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[13),14)] 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（b）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．マクロタスクグラフにおいても矢印が省略されているエッジの向きは，下向きが仮定されている．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．



(a) Macro Flow Graph (MFG)　(b) Macro Task Graph (MTG)

図 1　マクロフローグラフとマクロタスクグラフ
Fig. 1　Macro flow graph and macro task graph

2.4　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が決定される．コンパイラは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．スタティックスケジューリングでは，実行時スケジューリングのオーバーヘッドを無くし，データ転送オーバーヘッド，同期オーバーヘッドを最小化することができる．

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．両者のスケジューリング方式は，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等を考慮して使い分ける．

2.5 OpenMP によるコード生成

OSCAR コンパイラは複数のバックエンドを持ち，OSCAR タイプ分散/共有メモリシングルチップマルチプロセッサ[15]，UltraSparc-II，MPI-2 などのターゲット用にコード生成を行うことができる．本論文では，SMP 上での並列化標準 API である OpenMP を用いてコード生成を行なうことで，ポータビリティの高い粗粒度タスク並列処理を実現した．本節では，OpenMP を用いた粗粒度タスク並列処理の実現[16),17)]について述べる．

OSCAR コンパイラによって生成された OpenMP FORTRAN では，プログラム開始時に，PARALLEL SECTIONS ディレクティブを用いて，一度だけプロセッサ台数分のスレッドを生成する．本実現方式では，スレッドの fork/join は 1 度しか行わないため，スレッド生成のオーバーヘッドは低く抑えられる．

各 OpenMP セクションには，各階層毎に適用されるスケジューリング方式にしたがって，ダイナミックスケジューリング時にはマクロタスクコードとスケジューリングコードが，またスタティックスケジューリング時にはマクロタスクコードと同期コードがコンパイラにより生成される．以下に例を用いて，各スケジューリング方式を適用した場合のコード生成法について述べる．

2.5.1 スタティックスケジューリング

プログラムから図 2 に示すようなマクロタスクグラフが得られた場合に，8 プロセッサで粗粒度タスク並列処理を実現する OpenMP FORTRAN プログラムのコードイメージの例を図 3 に示す．

この例では，プログラム開始時に 8 プロセッサに対応して 8 スレッドが fork される．図 2 の第 1 階層（プログラム全体）の内部にはマクロタスク 1_1，1_2，１_3 が定義されている．図 3 は，この第 1 階層に対して，生成された 8 スレッドを 4 スレッドずつにグループ化して，スタティックスケジューリングを適用した場合のコードイメージを示している．スレッドグループは複数のプロセッサエレメントから構成されたプロセッサクラスタに相当し，このグループ単位でマクロタスクの割り当て，実行が行われる．スタティックスケジューリングを適用した場合，コンパイラは各 OpenMP セクションに対して，スケジューリング結果に従い異なったコードを生成する．この例では，コンパイラのスタティックスケジューリングによりスレッド 0 からスレッド 3 のグループ 0 にマクロタスク 1_1，1_3 が，スレッド 4 からスレッド 7 のグループ 1 にマクロタスク 1_2 が割り当てられており，各スレッドに対応する OpenMP セクションには，割り当てられたマクロタスクのコードが，図に示すように生成される．

2.5.2 集中ダイナミックスケジューリング

サブルーチンブロックであるマクロタスク 1_2（SB）の内部に，図 2 に示すように第 2 階層 1（図中 2nd Layer1）が定義され，サブマクロタスク 1_2_1 から 1_2_6 が生成されたとする．図 3 はこの第 2 階層 1 に集中ダイナミックスケジューリングを適用し，マクロタスク 1_2 を割り当てられたスレッドグループ 1 内の 4 スレッドによって粗粒度タスク並列処理を階層的に行う場合のコードイメージを示している．集中スケジューリングの場合，一つのスレッドがスケジューラとして働き，そのスレッドはマクロタスクの実行は行なわず，その他のスレッドへのマクロタスクの割り当てのみを行なう．したがって，集中スケジューラとなるスレッドのセクションにはスケジューリングルーチンが生成される．図 3 の例では，スレッド 7 が集中スケジューラである．一方，スケジューラ以外のスレッドは，マクロタスクの実行を担当するが，実行するマクロタスクは実行時にスケジューラによって決められるため，各スレッドは全てのマクロタスクを実行する可能性を持つ．したがって，それらのスレッドのセクションには図 3 のスレッド 4 からスレッド 6 に示すように，全ての第 2 階層 1 のマクロタスクのコードが生成され (図では MT1_2_5 以降は省略)，各スレッドは実行時にスケジューラの割り当て結果に従い，これらのマクロタスクを選択的に実行する．また，コンパイラは階層の終わりを示す特別なマクロタスク，エンドマクロタスクを生成する．スケジューラはこのエ

ンドマクロタスクを各スレッドに割り当てたら，スケジューリングルーチンの実行を終了する．スケジューラ以外のマクロタスク実行スレッドは，エンドマクロタスクを実行したら，この階層の実行を終了する．

2.5.3 分散ダイナミックスケジューリング

一方，繰り返しブロックであるマクロタスク 1_3 の内部を分割して生成された，図 2 に示すような第 2 階層 2（図中 2nd Layer2）のサブマクロタスク 1_3_1 から 1_3_4 に対して，分散ダイナミックスケジューリングを適用した場合のコードイメージが図 3 に示されている．この階層では，マクロタスク 1_3 を実行するスレッドグループ 0 内の 4 スレッドを 2 スレッドずつにグループ化している．分散ダイナミックスケジューリングでは，各スレッドがスケジューリングとマクロタスクの実行の両方を行うので，各セクションには全マクロタスクのコードとスケジューリングルーチンの両方が生成される (図 3 では紙面の都合上 MT1_3_3 以降は省略)．各スレッドは，スケジューリングルーチンを実行し，自身が次に実行するマクロタスクを決定した後，そのマクロタスクを実行する．マクロタスクの実行終了後は，マクロタスクコードの後に生成されたスケジューリングルーチンを実行し，次に実行するマクロタスクを探す．この動作を，エンドマクロタスクを実行するまで繰り返す．

また，図 3 の第 2 階層 2（2nd Layer2）のマクロタスクは，2 スレッドからなるスレッドグループによって実行されるが，この時スレッドグループ内のスレッドによって，マクロタスク内部をどのように並列処理するかは，そのマクロタスクのスケジューリング方式によって異なる．したがって，マクロタスク 1_3_1 から 1_3_4 の内部には，選択されたスケジューリング方式に応じたコードがグループ内の各スレッドに対して第 3 階層（3rd Layer1, 3rd Layer2,...）として生成される．図 3 では，MT1_3_1a，MT1_3_1b のように記述することにより，グループ内のスレッドが異なったコードを持つことを示している．

2.5.4 本実現手法の特徴

本節で述べた各スレッド用にコードを生成する OpenMP を用いた粗粒度タスク並列処理の実現方式では，ネストした並列スレッド生成を行なわないシングルレベルのスレッド生成を用い，OpenMP の拡張を行うこと無く，比較的低オーバーヘッドで階層的な並列性を記述できるという特徴がある．

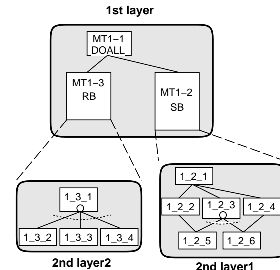


図 2 マクロタスクグラフ
Fig. 2 An example of macro task graph

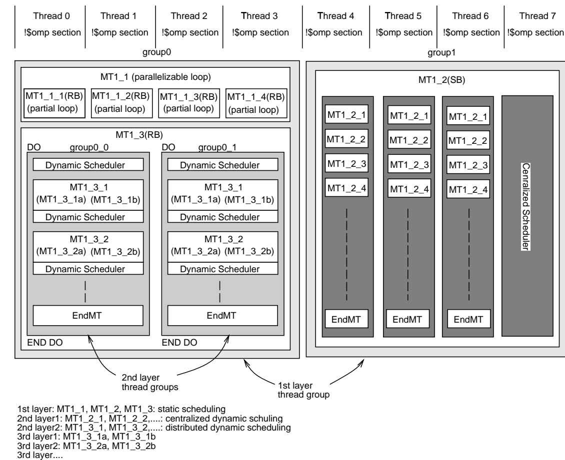


図 3 OpenMP で生成されたコードイメージ
Fig. 3 Generated code image using OpenMP

3. 粗粒度タスク並列処理におけるキャッシュ最適化

本章では，キャッシュを有効に利用することによって，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**局所性**を利用して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[12]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[15]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，一般にコンパイラから制御可能なローカルメモリを持たず，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ようにデータ分散及びスケジューリング手法を開発した．

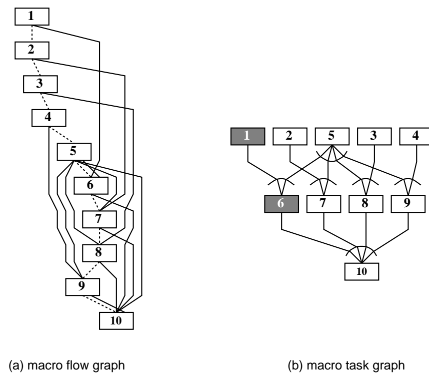


図 4 キャッシュ最適化の例
Fig. 4 An example of cache optimization for a macro task graph

主記憶共有型マルチプロセッサ上での粗粒度タスク並列処理において，データ共有量の多いマクロタスクを同じプロセッサで連続的に実行されるように割り当てると，それらのマクロタスク間でのデータ転送を，分散キャッシュを用いて高速に行うことが可能となる．

図 4 に示すマクロフローグラフとマクロタスクグラフを例に説明する．図 4（a）に示すマクロフローグラフでは，点線はマクロタスク間のコントロールフローを表わしている．したがって，この例におけるプログラムの通常の実行順は，マクロタスク番号の増加順である．ここで，マクロタスク 1 と 6 の間でデータ共有量が大きいものとする．通常の実行順では，マクロタスク 1 の実行後は，マクロタスク 6 より先にマクロタスク 2，3，4，5 が実行される．したがって，マクロタスク 1 の実行によってキャッシュに存在したデータがマクロタスク 6 の実行前にキャッシュから追い出され，キャッシュの利用効率が悪くなる可能性が高い．一方，図 4（b）は最早実行可能条件解析によって図 4（a）から得られたマクロタスクグラフである．このグラフが示すように，最早実行可能条件解析によって，マクロタスク 6 はマクロタスク 1，5 にのみデータ依存しているため，マクロタスク 2，3，4 よりも先に実行を開始することができることが分かる．したがって，マクロタスク 5 の実行をマクロタスク 1 より先に終了させておけば，マクロタスク 1，6 は連続実行することができるため，マクロタスク間でキャッシュを用いて高速にデータを受け渡すことが可能となる．

提案するキャッシュ最適化手法は，**ループ整合分割**，**パーシャルスタティック割り当て**の主に二つの技術から構成される[12]．

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う可能性があるため，他のループとの間でキャッシュを用いて効率よくデータ転送を行うことができない．このようなキャッシュ利用効率の低下を避けるため，そのようなデータ使用量の多いループに対してループ整合分割[12]を行う．ループ整合分割によってループは，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小ループに分割される．

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，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グループ化される．このグループは**データローカライゼーショングループ**（*DLG*）[12] と呼ばれ，同一 DLG 内のマクロタスク群は，データ共有量が多いため同じプロセッサに割り当てられるように，ダイナミックタスクスケジューラに指示される．

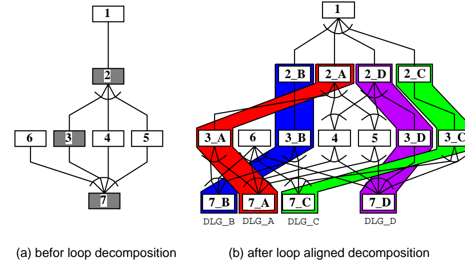


図 5 ループ整合分割とデータローカライゼーショングループ
Fig. 5 An example of Loop Align Decomposition

図 5 にループ整合分割の例を示す．図 5（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にアクセス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 5（b）である．例えば図 5（a）のマクロタスク 2 は，図 5（b）のマクロタスク 2_A，2_B，2_C，2_D に分割されている．また，図 5（b）の色のついた帯は各データローカライゼーショングループを表わす．例えば，図 5（b）のマクロタスク 2_A，3_A，7_A の間ではデータの共有量が多いため，それらのマクロタスクはデータローカライゼーショングループ A（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

2 章で述べたように，マクロタスクはコンパイラによるスタティックスケジューリングまたはコンパイラ

の生成したダイナミックスケジューリングルーチンによって実行時にプロセッサに割り当てられる．提案するキャッシュ最適化手法では，同じ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が，マクロタスクの最早実行可能条件を考慮しつつ，同一プロセッサで可能な限り連続して実行されるようにタスクスケジューリングアルゴリズムを拡張する．

本論文では，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にも適用できることから，ダイナミックスケジューリングを用いて性能評価を行なった．ダイナミックスケジューリングにおいて，同じ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を同じプロセッサに割り当てるための拡張を，“パーシャルスタティック割り当て”[12] と呼ぶ．

従来のローカルメモリを対象としたパーシャルスタティック割り当てを伴うデータローカライゼーション手法では，ローカルメモリに対するデータの割り当てや，データ転送をコンパイラが制御可能なため，コンパイラがローカルメモリに配置したデータは，コンパイラが明示的に追い出さない限り，ローカルメモリ中にあることを想定している．したがって，パーシャルスタティック割り当てとしては，同じ DLG に属するマクロタスクを同一プロセッサに割り当てることのみが提案されている．一方キャッシュを対象とした本手法では，データサイズがキャッシュサイズと比較して大きい場合，キャッシュに載ったデータが LRU(Least Recently Used) 方式で他のマクロタスクの実行により追い出される可能がある．したがって，本キャッシュ用パーシャルスタティック割り当て手法では，多くのアプリケーションでデータサイズがキャッシュサイズよりも大きいことを考慮しデータをキャッシュサイズを考慮して分割すると共に，同じ DLG に属するマクロタスクを同一プロセッサに割り当てることだけでなく，可能な限り連続して実行させるというスケジューリング手法をとっている．

また，今回の性能評価では使用する SMP マシンのプロセッサ数が少ないことを考慮して，今回開発したバージョンのコンパイラでは，集中スケジューラ方式により一つのプロセッサをスケジューリング専用とするのは実行効率上好ましくないため，全てのプロセッサをマクロタスクの処理に使用する分散ダイナミックスケジューリング方式を採用している．

またプログラムによってはスタティックスケジューリング方式[18] も適用可能であるが，スタティックスケジューリングはマクロタスクグラフ中にデータ依存エッジしか存在しない場合には有効であるものの，条件分岐への対応が難しいため本論文ではより汎用的なダイナミックスケジューリングを用いている．

パーシャルスタティック割り当てを適用した分散ダイナミックスケジューリングにおける各プロセッサの動作は次のようになる．

1 実行マクロタスクの選定 各プロセッサに分散されているスケジューリングルーチンを実行し，共有レディータスクキューより自分が実行するべきマクロタスクを探す．この時の探し方は，直前に実行したマクロタスクが無い場合 (初期状態) または直前に実行したマクロタスクが DLG に含まれない場合は 1.1，直前に実行したマクロタスクが DLG に含まれる場合は 1.2 となる．

- **1.1** 直前に実行した MT がないまたは MT が DLG に含まれないとき．まずどの DLG にも含まれない実行可能マクロタスクの中から探す．見つからない場合は，DLG に含まれる実行可能マクロタスクの中から探す．
- **1.2** 直前に実行したマクロタスクが DLG に含まれるとき．まず直前に実行した DLG と同じ DLG に含まれる実行可能マクロタスクを探す．見つからなければ，それ以外のマクロタスクから実行可能マクロ探す．

実行マクロタスクが決定されるまで，このスケジューリングルーチンを繰り返す．実行するマクロタスクが決まったら 2 へ．

2 マクロタスクの実行 自身に割り当てたマクロタスクを実行する．実行が終了したら 3 へ．

3 階層終了判定 実行したマクロタスクが，この階層のマクロタスクグラフの終了を表わす特別なマクロタスク，エンドマクロタスクの場合，この階層の実行を終了する．この場合は 4 へ進む．そうでない場合は 1 へ．

4 階層終了 現在実行している階層の実行を終了し，この階層を内包する外側階層の実行に移る．この階層の外側階層が存在しない場合，すなわちこの階層が最外層の場合はプログラムの実行を終了する．

図 5 (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 6 に示す．前述した通り，プログラムのオリジナルの実行順は，マクロタスクの番号順であり，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い．一方，提案手法を適用した場合は，図 6 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

図 6 図 5 (b) のマクロタスクグラフのシングルプロセッサへのスケジューリング例

Fig. 6 An example of schedule for single processor

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．

4. 性 能 評 価

本章では，提案手法の性能評価について述べる．本性能評価では OSCAR コンパイラを並列化プリプロセッサとして使用し，逐次 FORTRAN77 プログラムから提案手法を実現する OpenMP FORTRAN プログラムを生成して，対象マルチプロセッサシステム上のネイティブコンパイラによってコンパイルして実行させた．また，比較対象としてネイティブコンパイラによる自動並列化を行った．

4.1 評 価 環 境

性能評価は，IBM RS/6000 604e SP High Node 上で spec95fp ベンチマークの tomcatv, swim, mgrid を用いて行なった．

Tomcatv はベクトル化メッシュ生成プログラムであり，実行時間の 99%以上を占める収束ループを持つ．収束ループの内部には複数のループがあるが，これらの各ループはキャッシュサイズ以上のデータにアクセスするため，収束ループの内側に対して，提案するキャッシュ最適化を考慮した粗粒度タスク並列処理を適用する．

Swim は Shallow water 方程式の求解プログラムである．Swim の実行時間の大部分は，メインループから呼ばれるサブルーチン CALC1，CALC2，CALC3 によって占められる．これらのサブルーチン内には，同一データにアクセスするループがあるが，各ループはキャッシュサイズよりも大きなサイズのデータを扱う．

Mgrid は 3 次元のマルチグリッドソルバーである．実行時間の約 70%はサブルーチン RESID と PSINV によって占められる．これらのサブルーチンはキャッシュサイズを越えるデータを使うループを持ち，それらのループはデータを共有している．

今回のインプリメントに使用した現バージョンの OSCAR コンパイラは，研究コンパイラとして世界との研究レベルでの競争力をつけることを主眼に置き，他の市販コンパイラで行われていない新しい機能の実現に高いプライオリティをおいて開発を進めているため，技術的には既存の市販コンパイラでは実現されている新規性がないプログラム変換の内，OSCAR コンパイラでは組み込み，あるいはデバッグ作業が終了していないものがある．このような事情から研究レベルでなく他のコンパイラでも実現可能な変換あるいは，今回の局所性最適化に直接関係しない変換に対しては手動でシーケンシャルソースプログラムを書き換えて，その後 OSCAR コンパイラでキャッシュ最適化を含んだ自動並列化を行った．以下に今回行ってる改変について述べる．

今回の性能評価では，tomcatv に対して二重ループの外側ループを並列処理可能ループとするためループインターチェンジを行った．一般的なループ並列処理では，二重ループの場合外側ループでの並列処理を行った方が有利で，多くの市販コンパイラではループインターチェンジが可能だが，OSCAR コンパイラのインターチェンジルーチンのバグを避けるため，今回の評価では手動でインターチェンジを行った．

ループインターチェンジ後の tomcatv では，二重ループでの二次元配列へのアクセスの際に，内側ループインデックスが配列の二次元目となるため，アクセスがメモリ上で非連続となり効率が悪い．したがって配列の転置を行うことにより，配列へのアクセスがメモリ上で連続となるようにする．ループでの配列へのアクセスが非連続となる場合は，本手法に関係なくキャッシュの利用効率が悪くなるため，このような配列の転置は MIPSpro コンパイラでも行われている．OSCAR コンパイラでは実装が終了していないが，特別な変換ではなく一般的に有効な最適化であるため，今回の性能評価では手動で行った．これらの変換後のコードは OSCAR コンパイラだけでなく，XLF コンパイラの入力としても使っており，これらの変換によって提案手法のみが有利になることはない．

また，ループ回転数が実行時まで不明な場合は，ループで使用するデータ量がコンパイル時に決定できないため，コンパイル時に分割数を適切に設定できなくなる．これに対し，ループ回転数をループ内部で使用する配列の定義サイズから推定するという方法での検討が進められており，メモリの動的確保が不可能な FORTRAN77 では，妥当な推定が可能であると考えられるが，本論文とは異なる技術であるため，今回の評価プログラムに対してはループ回転数を定数に変換した．

また今回の性能評価では swim，mgrid に対してインライン展開を適用したが，OSCAR コンパイラの

表 1　評価使用マシンパラメータ
Table 1　Parameters of the machine used for performance evaluation

	IBM RS/6000	Sun Ultra80
プロセッサ	PowerPC 604e	UltraSparc-II
クロック周波数	200MHz	450MHz
プロセッサ数	8	4
L1 キャッシュ(I)	32KB	16K
L1 キャッシュ(D)	32KB	16K
L2 キャッシュ(U)	1MB	4MB
OS	AIX 4.3	Solaris 8
コンパイラ	IBM XLF 6.1	Sun Forte 6u1

OpenMP 出力をコンパイルする市販のネイティブコンパイラがソースコードサイズの増大により，コンパイル不能となる場合があったため，インライン展開されたループを一つのサブルーチンに手動で戻した．これは，本来ならばインライン展開されたコードをそのままネイティブコンパイラがコンパイルすれば良いのであるが，今回の性能評価では OpenMP コンパイラのバグを避けるためにサブルーチンに戻したものであり，本キャッシュ最適化手法の性能とは関係ないものである．

また，IBM RS/6000 604e 用のキャッシュ動作測定ツールが入手できないため，CPU Performance counter によりキャッシュミス回数の測定が可能な Sun Ultra80 上でも評価を行ない，本手法によるキャッシュミスの削減効果を調べた．各ターゲットマシンのパラメータを表 1 に示す．

4.2 IBM RS/6000 604e SP High Node 上での評価

IBM RS/6000 ではネイティブコンパイラとして，IBM XL FORTRAN version 6.1（以下 XLF コンパイラ）を用いた．逐次実行時間を得るときに用いた XLF コンパイラのオプションは “-O3 -qhot -qmaxmem=-1 -qarch=auto -qtune=auto -qcache=auto” である．“-O” は最適化オプションであり，“-O3” レベルの最適化では，メモリとコンパイル時間を大量に使用する最適化を行う．“-qhot” は最適化時にループや配列に対する高レベルの変換，およびキャッシュミスを避けるためのパディングを行うことを指定する．また，“-qmaxmem=-1” は最適化時に使用するメモリ量を制限しないことを示し，“-qarch -qtune -qcache” は対象となるアーキテクチャを指定するものである．“auto” では自動判定が行なわれ，コンパイル時の環境と同じ環境でプログラムが実行される場合に指定する．また，自動並列化を行った際のコンパイルオプションは “-qsmp=auto -O3 -qhot -qmaxmem=-1 -qarch=auto -qtune=auto -qcache=auto” である．“-qsmp” は SMP 用にコード生成を行なうことを示し，“auto” では自動並列化が行なわれる．OSCAR コンパイラで並列化した OpenMP FORTRAN をコンパイルする際は “-qsmp=noauto -O3 -qhot -qmaxmem=-1 -qarch=auto -qtune=auto -qcache=auto” を用いた．“-qsmp=noauto” は自動並列化は行なわずに SMP 用にコード生成を行なうことを示す．

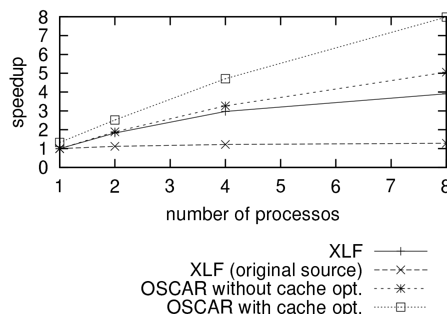


図 7　Speedup of tomcatv on RS/6000 604e
Fig. 7　Speedup of tomcatv on RS/6000 604e

4.2.1 Tomcatv

RS/6000 上での tomcatv を用いた場合の XLF コンパイラのみを用いた場合の逐次実行に対する，それぞれのコンパイラの速度向上率を図 7 に示す．Tomcatv の逐次実行時間は 693 秒であるのに対し，XLF コンパイラで自動並列化を行った場合の実行時間は，2PE で 370 秒，4PE で 233 秒，8PE で 177 秒であり，速度向上率はそれぞれ，1.83，2,98，3.92 であった．また，前述の変換を行わないソースコードを入力とした場合の XLF コンパイラで自動並列化の結果（図中 XLF (original source)）は 2PE で 620 秒，4PE で 571 秒，8PE で 542 秒であり，速度向上率はそれぞれ 1.12，1.21，1.28 であった．一方，OSCAR コンパイラを XLF のプリプロセッサとして用いて，提案手法を適用した場合は．1PE で 523 秒，2PE で 275 秒，4PE で 147 秒，8PE で 87 秒で，XLF での逐次実行に対する速度向上率は 1.33，2.52，4.71，7.98 となり，XLF コンパイラの速度向上率と比較して最高で約 2.0 倍上回る性能を示した．また，本キャッシュ最適化手法を適用しなかった場合の OSCAR コンパイラ[16),17)] を用いた実行時間は，1PE で 701 秒，2PE で 367 秒，4PE で 212 秒，8PE で 137 秒であった．したがって，本キャッシュ最適化により，1PE で 25%，2PE で 25%，4PE で 31%，8PE で 37%の速度向上が得られることが確認できた．

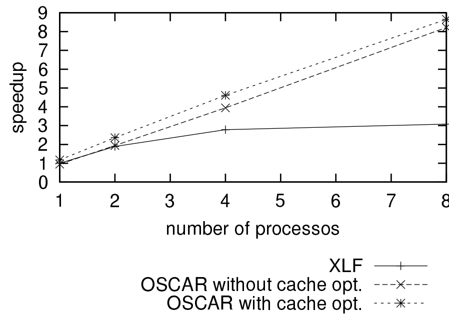


図 8 Speedup of swim on RS/6000 604e
Fig. 8 Speedup of swim on RS/6000 604e

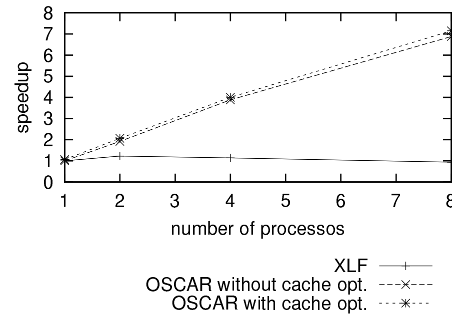


図 9 Speedup of mgrid on RS/6000 604e
Fig. 9 Speedup of mgrid on RS/6000 604e

4.2.2 Swim

Swim を用いた RS/6000 上での速度向上率を図 8 に示す．Swim の XLF のみを用いた場合の逐次実行時間は 512 秒であり，XLF コンパイラの自動並列化では，2PE で 227 秒，4PE で 183 秒，8PE で 169 秒へと速度向上が得られた．一方，OSCAR コンパイラを用いた場合は，1PE で 443 秒，2PE で 221 秒，4PE で 113 秒，8PE で 60 秒となり，XLF コンパイラの最高性能と比較して 2.8 倍上回る性能向上を示した．キャッシュ最適化を適用しなかった場合の OSCAR コンパイラでは，1PE で 551 秒，2PE で 269 秒，4PE で 132 秒，8PE で 63 秒であり，本キャッシュ最適化手法は従来のキャッシュ最適化を行わない手法に比べ，1PE で 20%，2PE で 18%，4PE で 14%，8PE で 5%の速度向上を示していることが分かり，本手法の有効性が確認された．

4.2.3 Mgrid

Mgrid の RS/6000 での速度向上率を図 9 に示す．Mgrid では，逐次実行時間の 676 秒に対して，OSCAR コンパイラを用いた場合は，2PE で 352 秒，4PE で 174 秒，8PE で 95 秒というスケーラブルな速度向上を示し，図に示されるように速度低下が見られる XLF コンパイラの自動並列化の最高性能（2PE）と比較して，5.8 倍の速度向上を示した．また，キャッシュ最適化なしの従来手法に比べ，最大 7%の速度向上が得られた．mgrid におけるキャッシュ最適化の効果が tomcatv，swim に比べ小さいのは，mgrid のデータサイズが両アプリケーションよりも小さいため，本キャッシュ最適化を行なわない場合でもデータがキャッシュ上にうまく収容されたことと，キャッシュ最適化適用部分のプログラム全体に占める実行時間が小さいため，相対的に効果が低くなっていることが原因と考えられる．

4.3 L2 ミス回数

本節では，CPU Performance counter によるキャッシュミス回数の測定が可能な Sun Ultra80 において，L2 キャッシュのミス回数を計測し，本キャッシュ最適化手法によりキャッシュ利用効率が改善されていることを確認する．Sun Ultra80 でのネイティブコンパイラには，Sun Forte 6 update 1 を用いた．逐次プログラムのコンパイルオプションは "-fast" を用い，自動並列化を行う際は "-fast -parallel -reduction -stackvar" を使用した．OSCAR コンパイラと共に使用するときは "-fast -mp=opnemp -explicitpar -stackvar" を指定した．"-fast" は最高レベルの最適化を行なう "-O5"，対象アーキテクチャを指定する "-xtarget" などの複数の最適化オプションの組合わせであり，プログラムの実行速度を向上させるために使用するオプションである．また，"-parallel" は自動最適化を行なうためのオプションであり，"-reduction" はリダクションループの自動並列化を行なうことを示す．"-stackvar" は局所変数をスタックに割り当てるためのオプションで，ループの並列化の自由度が高くなるため，並列化時に利用が推奨されているオプションである．"-explicitpar" はソースプログラム中に明示的な並列化の指定があることを示し，"-mp=openmp" は並列化の指定方法が OpenMP であることを示す．

4.3.1 Tomcatv

Ultra80 上での tomcatv の Forte コンパイラのみを用いた場合の逐次実行時間に対する両コンパイラ速度向上率を図 10 に示す．Ultra80 上で Forte コンパイラを用いた場合の tomcatv の逐次実行時間は 109 秒である．これに対し Forte コンパイラの自動並列化では，2PE で 84 秒，3PE で 80 秒，4PE で 79 秒となるのに対し，OSCAR コンパイラを用いた場合は，2PE で 76 秒，3PE で 48 秒，4PE で 32 秒となり，Forte に比べ最大 2.5 倍の性能向上を示した．

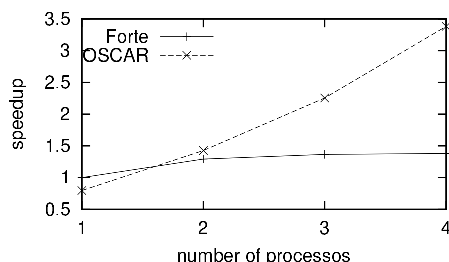


図 10　Speedup of tomcatv on Ultra80
Fig. 10　Speedup of tomcatv on Ultra80

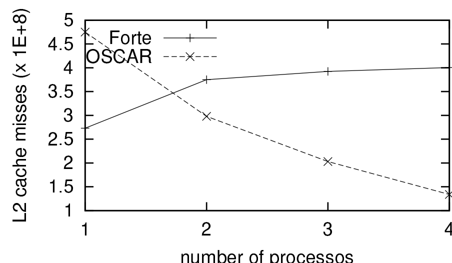


図 11　Number of L2 cache misses of tomcatv on Ultra80
Fig. 11　Number of L2 cache misses of tomcatv on Ultra80

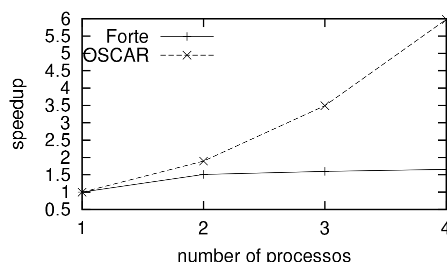


図 12　Speedup of swim on Ultra80
Fig. 12　Speedup of swim on Ultra80

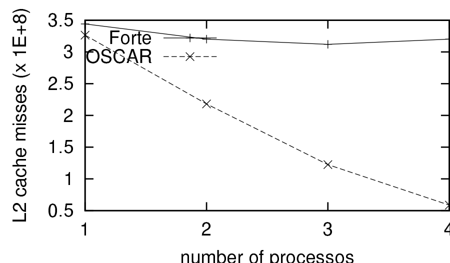


図 13　Number of L2 cache misses of swim on Ultra80
Fig. 13　Number of L2 cache misses of swim on Ultra80

また，それぞれの場合の L2 キャッシュミス回数を図 11 に示す．Tomcatv では，Forte コンパイラを用いたときの 4 プロセッサでの L2 キャッシュミス回数は約 4.0 億回であったのに対し，OSCAR コンパイラを用いた時は約 1.3 億回であり，本手法により，キャッシュミス回数が軽減されていることが確認された．

また，1PE の場合のみ Forte コンパイラが OSCAR コンパイラの性能を上回っている．これは，Forte コンパイラの 1PE 向けキャッシュ最適化が効き，L2 キャッシュミス回数が下がっているのに対し，OSCAR コンパイラが生成したループ実行順序を変更した逐次 FORTRAN コードに対して Forte コンパイラを使用した場合には，オリジナルコードに対しては有効であった最適化が効果的に適用されなかったためであると考えられる．

4.3.2　Swim

Ultra80 での swim の速度向上率を図 12 に示す．Swim の逐次実行時間は 100 秒である．Forte の自動並列化では，2PE で 66 秒，3PE で 63 秒，4PE で 60 秒と推移するにに対し，OSCAR コンパイラを用いた場合は，2PE で 53 秒，3PE で 27 秒，4PE で 17 秒となり，Forte と比べて最大 3.6 倍の性能向上を示した．

Swim における L2 キャッシュミス回数を図 13 に示す．swim の 4 プロセッサでの L2 キャッシュミス回数は，Forte コンパイラでは 4PE で約 3.2 億回であったのに対し，OSCAR コンパイラでは約 0.6 億回に削減されていることが分かり，OSCAR コンパイラを用いて本手法を適用した場合のキャッシュミス回数は，Forte の場合よりも少なくなっていることが確認された．

5. ま と め

本論文では，主記憶共有型マルチプロセッサマシンの実効性能を向上させるための，キャッシュ最適化を考慮した粗粒度タスク並列処理について述べた．本手法の実現にはポータビリティを考慮して，SMP 上での標準 API である OpenMP を用いた．本手法を，OSCAR マルチグレインコンパイラに実装し，逐次 FORTRAN プログラムから本手法を実現した OpenMP で並列化されたプログラムを生成して，性能評価を行なった．

性能評価では，OSCAR コンパイラで並列化した spec95fp の tomcatv，swim，mgrid を商用 SMP マシン IBM RS/6000 604e SP High Node（8 プロセッサ），Sun Ultra80（4 プロセッサ）で実行させることにより評価を行なった．その結果，IBM XL FORTRAN version 6.1 による自動並列化に対して，各プログラムの 8 プロセッサ上での最高性能で比較すると，OSCAR コンパイラは tomcatv で 2.0，swim で 2.8，mgrid で 5.8 倍の性能向上を可能にすることが確かめられた．

また，Sun Ultra80 上での評価では，OSCAR コンパイラは Sun Forte 6 update 1 コンパイラによる自動

並列化に対して，各プログラムの最高性能は tomcatv で 2.5，swim で 3.6 倍を示した．また，Ultra80 での測定により，キャッシュミス回数が本手法により削減されていることが確かめられた．

また，今後の研究課題として，各階層のプロセッサ数割り当ての自動決定や，各階層毎の粗粒度タスクスケジューリング方式の自動決定があげられる．また，今後より多くのベンチマークプログラムを用いてより多くのマルチプロセッサシステム上で評価を行っていきたいと考えている．

謝辞 本研究の一部は METI/NEDO 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) Eigenmann, R., Hoeflinger, J. and Padua, D.: On the Automatic Parallelization of the Perfect Benchmarks, *IEEE Trans. on parallel and distributed systems*, Vol. 9, No. 1 (1998).
2) Tu, P. and Padua, D.: Automatic Array Privatization, *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing* (1993).
3) Rauchwerger, L., Amato, N. M. and Padua, D.A.: Run-Time Methods for Parallelizing Partially Parallel Loops, *Article of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pp. 137–146 (1995).
4) Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S., Bugnion, E. and Lam, M. S.: Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer* (1996).
5) 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).
6) Lim, A.W., Liao, S.-W. and Lam, M.S.: Blocking and Array Constraction Across Arbitrarily Loops Using Affin Partitioning, *Article of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001).
7) Lim, A. W. and Lam, M. S.: Cache Optimizations With Affine Partitioning, *Article of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth* (2001).
8) Han, H., Rivera, G. and Tseng, C.-W.: Software Support for Improving Locality in Scientific Codes, *8th Workshop on Compilers for Parallel Computers* (2000).
9) Rivera, G. and Tseng, C.-W.: Locality Optimizations for Multi-Level Caches, *Super Computing '99* (1999).
10) Brownhill, C. J., Nicolau, A., Novack, S. and Polychronopoulos, C. D.: Achieving Multilevel Parallelization, *Proc. of the International Symposium on High Performance Computing* (1997).
11) 岡本, 合田, 宮沢, 本多, 笠原: OSCAR マルチグレインコンパイラにおける階層型マクロデータフロー処理手法, 情報処理学会論文誌, Vol. 35, No. 4, pp. 513–521 (1994).
12) 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).
13) 本多弘樹, 岩田雅彦, 笠原博徳: Fortran プログラム粗粒度タスク間の並列性検出手法, 電子情報通信学会論文誌, Vol. J73-D-1, No. 12, pp. 951–960 (1990).
14) 笠原博徳: 並列処理技術, コロナ社 (1991).
15) Kimura, K. and Kasahara, H.: Near Fine Grain Parallel Processing Using Static Scheduling on Single Chip Multiprocessors, *Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (1999).
16) Kasahara, H., Obata, M. and Ishizaka, K.: Automatic Coarse Grain Task Parallel Processing on SMP using OpenMP, *Proc. of 13 the International Workshop on Languages and Compilers for Parallel Computing 2000* (2000).
17) 笠原, 小幡, 石坂: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理, 情報処理学会論文誌 (2001).
18) 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).

(平成 13 年 9 月 10 日受付)
(平成 14 年 2 月 13 日採録)

石坂　一久（学生会員）

昭和 51 年生．平成 11 年早稲田大学理工学部電気電子情報工学科卒業．平成 13 年同大学大学院修士課程修了．平成 13 年同大学大学院博士課程進学，現在に至る．

中野　啓史（学生会員）

昭和 52 年生．平成 13 年早稲田大学理工学部電気電子情報工学科卒業．平成 13 年同大学大学院修士課程進学，現在に至る．

八木　哲志

昭和 51 年生．平成 12 年早稲田大学理工学部電気電子情報工学科卒業．平成 14 年同大学大学院修士課程修了．同年日本電信電話 (株) 入社，現在に至る．

小幡　元樹（正会員）

昭和 48 年生．平成 8 年早稲田大学理工学部電気工学科卒業．平成 10 年同大学大学院修士課程修了．平成 12 年同大学同学部助手，現在に至る．

笠原　博徳（正会員）

昭和 32 年生．昭和 55 年早稲田大学理工学部電気工学科卒業．昭和 60 年同大学大学院博士課程修了．工学博士．昭和 58 年同大学同学部助手．昭和 60 年学術振興会特別研究員．昭和 61 年早稲田大学理工学部電気工学科専任講師．昭和 63 年同助教授．平成 9 年同大学電気電子情報工学科教授，現在に至る．平成元年〜2 年イリノイ大学 Center for Supercomputing Research & Development 客員研究員．昭和 62 年 IFAC World Congress 第一回 Young Author Prize．平成 9 年度情報処理学会坂井記念特別賞受賞．著書「並列処理技術」(コロナ社)．情報処理学会，電子情報通信学会，IEEE などの会員．