成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.44 No.1 (2007) pp.35-44

C言語自動並列化トランスレータにおけるマクロタスク拡張

田中　康之[*1]，三浦　純[*1]，甲斐　宗徳[*2]

An Extension of Automatic Parallelizing Translator for C Program Using Macro Task

Yasuyuki TANAKA[*1], Jyun MIURA[*1], Munenori KAI[*2],

ABSTRACT：In order to parallelize a C program, a set of source codes, automatically, a series of some analysis stages are required. They consist of parallelism analysis stage, execution time analysis stage, task granularity analysis stage, task scheduling stage, parallelized code generation stage. Among these stages, the results of other stages are shared. The authors have proposed a set of shared resources which are the source codes embedded with dedicated pragma directives. In this paper, a concept of macro task representation for the pragma directives is proposed. The macro task representation is useful to hide inner structure in the nest of program control structures at the stages, such as execution time analysis stage and task scheduling stage. In the parallelized code generation stage, because some MPI communication function calls are arranged into a single MPI packing communication function, the efficient parallelized program with reduced overhead can be generated.

Keywords : automatic parallelization, Message Passing Interface, pragmadirective, source code translator

（Received March 26, 2007）

１．はじめに

1．1　背景と目的

コンピュータの性能向上の手段としてマルチプロセッサシステムが有効であることは明らか[1)]であるが，そのためには並列性と効率を考慮したプログラムを実行しないとその優れた性能を十分に引き出せないという側面もある。従って，効率的な並列プログラムを作成する必要があり，それには並列性の抽出作業や，処理すべきタスクの実行順序を決めるためのスケジューリング作業などを行わなければならない。ところが，その作業にはアプリケーションの内部構造および処理システムのアーキテクチャなどの知識が要求されるため，マルチプロセッサシステムの有効利用という点で利用者にとって大きな障害になっている。

本研究では，ポインタの存在などにより実用的な自動並列化が困難とされているC言語に対して粗粒度および細粒度並列化[2)]を行うことで，実行性能の向上を図った並列プログラムに変換するC言語自動並列化トランスレータの開発を行っている。

1．2　本研究におけるマクロタスクの位置づけ

C言語自動並列化トランスレータ内では，タスクブロックと呼ばれる単位でタスクごとの情報が表現される。タスクブロックとは並列実行する際に各プロセスが実行する最小単位であり，最小の粒度はソースコードのステートメントとしている。

図 1.1 の例のようにタスクブロック内部にはソースコードの構造上，if 文や for 文などのように複数のタスクブロックを階層的に含むようなタスクがある。このようなタスクブロックを**「マクロタスク」**と呼ぶ。マクロタ

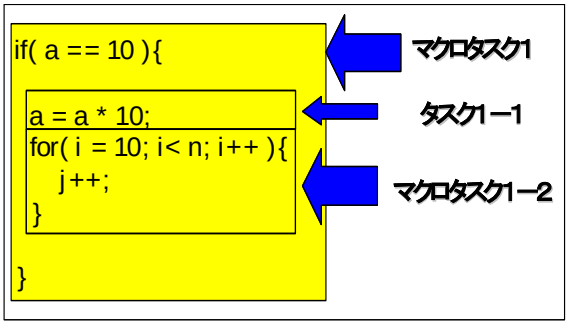


図 1．1　マクロタスク例

[*1]：工学研究科情報処理専攻修士学生

[*2]：情報処理専攻教授（kai@st.seikei.ac.jp）

スクは，スケジューリング時にそのタスクブロックの内部構造を隠蔽するために用いることができる。

本論文では，すでに提案していたC言語自動並列化トランスレータの処理の流れ[3)]にマクロタスクを考慮して解析を行う必要があることから，マクロタスクの取り扱いにともなう拡張を行った。

2．C言語自動並列化トランスレータ

C言語自動並列化トランスレータは，C言語で記述されたソースプログラムに存在する並列性を抽出し，それを活用することで自動的に並列実行コードを生成するものである。本研究で開発するC言語自動並列化トランスレータの全体的な処理の流れを図2.1に示す。次に各作業ステージでの解析について説明する。

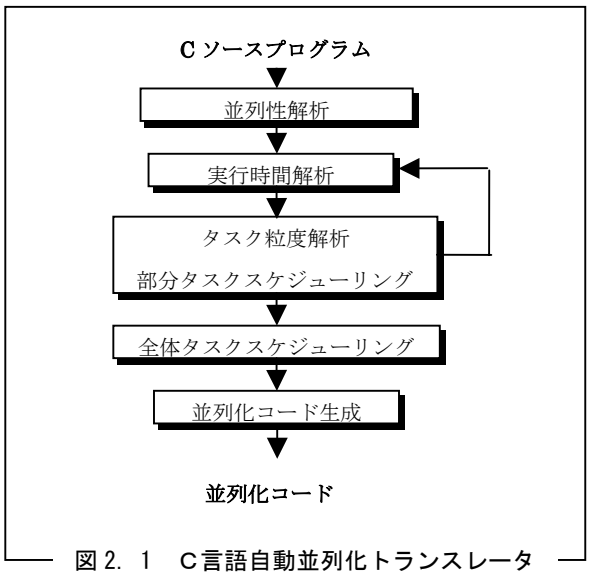


図2．1　C言語自動並列化トランスレータの処理の流れ

2．1　各作業ステージでの解析の流れ

前半の作業ステージとして，解析対象となるソースプログラムの並列性解析を行う。並列性解析では，プログラムに存在するステートメントレベルでの並列性を抽出する。

中盤の作業ステージとして，実行時間解析，タスク粒度解析，部分タスクスケジューリングを行う。実行時間解析では，並列性解析結果に基づき，逐次実行しなければならないコードセグメントごとの実行時間と，タスク間で行うデータ通信時間を求める。タスク粒度解析では，並列性解析と実行時間解析から得られた結果に基づき，各タスクをどのような大きさでプロセッサに割り当てれば効率の良い並列処理を行えるのかを考慮してタスク粒度を決める。部分タスクスケジューリングでは，それら処理すべきタスク集合をどのように各プロセッサに割り当て，どのような順序で実行すれば最短時間で処理が完了するのかを決める。

次に，全体タスクスケジューリングを行う。最終的な粒度での最適スケジューリングを行う。

最終段階の作業として，ターゲットマシンにおいて優れた実行性能を実現するような並列実行コードの自動生成を行う。

本研究では，C言語の逐次プログラムからMPI（Message Passing Interface）を用いた並列プログラムへの自動並列化トランスレータの開発を行った。MPIを用いることでより汎用性・移植性のあるコードを生成できる。また，生成される並列プログラムのモデルはSingle Program Multiple Data(SPMD)型である。

2.2　解析の流れの変更点

(1)　精度の高い実行時間解析への変更

タスク融合された複数のステートメントを含むタスクブロックの実行時間は個別のステートメントごとの実行時間の合計値とはならず，パイプライン処理やキャッシュの効果を含んだ時間となる。従って，タスク粒度解析によってまとめられた粒度で再び実行時間を計測する必要がある。

(2)　マクロタスクに対する解析

マクロタスクの実行時間および処理プロセス数はマクロタスク内部に含まれるサブタスクグラフの実行時間および，処理プロセッサ数となるが，そのサブタスクグラフもさらにマクロタスクを含む場合があるので，階層的なマクロタスクの最も内側のマクロタスクから順に実行時間解析，タスク粒度解析，部分的なタスクスケジューリングを行い，外側へと繰り返し解析を行うような流れへと変更を行った。

3．プラグマディレクティブによる共有リソースの改善

3．1　従来の共有リソース

図2.1のステージがスムーズに連携するためには，全てのステージ間で共有できるリソースが必要である。著者ら，すでにCソースコード上にプラグマディレクティブを用いてステージ間で情報を受け渡すことができるような共有リソースを設計した[3)]。共有リソースは並列性解析の結果により生成され，実行時間解析，タスクスケジューリングやタスク粒度解析時に読み込まれ，解析さ

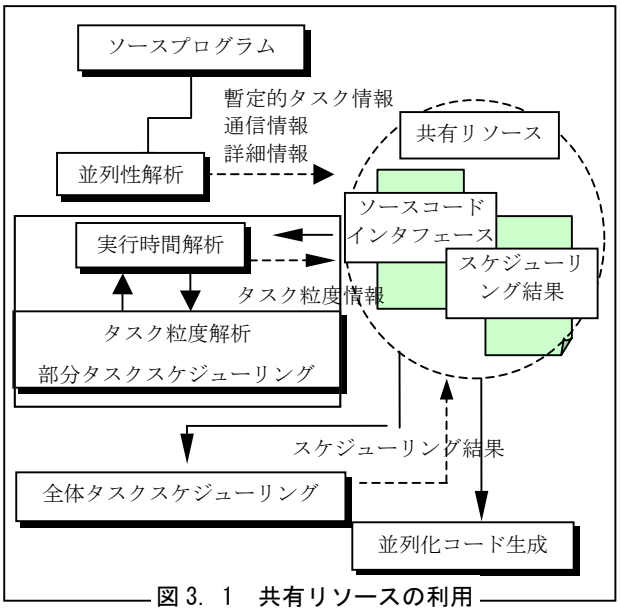


図 3. 1　共有リソースの利用

```
#pragma acpt TASK( …)
#pragma acpt DATA(…)
…
タスクコード
#pragma acpt TASK(…)
…
```

図 3. 2　ソースコードインタフェース例

表 3. 1　ソースコードインタフェース情報

[InfoType]	名称
TASK	タスク情報インタフェース
DATA	通信情報インタフェース
INFO	詳細情報インタフェース
REDUCT	リダクション情報インタフェース

れた情報を順々に追加していく。最終的にソースコードインタフェースとスケジューリング結果情報を並列コード生成部で読み込み，並列化コードを生成する。図 3.1 は各解析フェーズが共有リソースであるソースコードインタフェースを読み込んで解析を行い，その結果を共有リソースに戻している関係を表している。

従来は並列化のための情報は，タスクの情報を表すソースコードインタフェース，通信データに関する情報を持つ通信情報インタフェース，DOALL ループに関する詳細な情報，スケジューリング結果といった個別のファイルに分かれていた。本論文では，ソースコード上でこれらのインタフェースが一元管理できるように情報を統合し，並列化のための情報をすべてソースコード上のプラグマディレクティブに反映することを提案する。

その結果，最終ステージで生成されるスケジューリング結果を除く，すべての情報の受け渡しはソースコードインタフェースを通して行われる。以下で共有リソースの変更点について述べる。

3.2 共有リソースの変更点

3.2.1　ソースコードインタフェースと変更点

タスクブロックは解析対象のソースコード中にプラグマディレクティブを挿入することで表現される。プラグマディレクティブは#pragma acpt [InfoType] (…)のように記述され，[InfoType]の表記によってどの情報を表しているかを識別する。図 3.2 はソースコードインタフェースの例で InfoType の表記とその情報の名称は表 3.1 のようになっている。

これらが表す情報について，詳細を以下で説明する。

(1) タスク情報インタフェース

タスク情報インタフェースにより表現することが出来る情報は以下の通りである。

- ◆ タスクブロックの識別子
- ◆ タスクブロックのタイプ
- ◆ タスクブロックの実行時間
- ◆ タスクブロックの大きさ
- ◆ タスクブロック間の先行制約
- ◆ タスクブロック間のデータ通信時間
- ◆ タスクブロックの処理プロセス数

これらの情報を表現するプラグマディレクティブのフォーマットは図 3.3 で示すとおりである。

図 3.3 のフォーマットでは,”(Bid ET BT Op PN)”で表されているのが，個々のタスクブロック情報であり,”(DN (Did CT)　…)”はそのタスクブロックへの先行制約情報を表している。各要素の詳細は以下で説明する。“BT”で表されるタスクブロックのタイプには BASIC・LOOP・SELECT・FUNC がある。BASIC タイプのタスクブロックは定義している他のブロックタイプ以外のブロックすべてが当てはまる。LOOP タイプのタスクブロックはそのブロックが for や while 文であることを意味する。SELECT タイプのタスクブロックは if 文などの制御文を意味する。そして FUNC タイプのタスクブロックはユーザ定義関数の呼び出しであることを意味し，ライブラリ関数の呼び出しは FUNC タイプを持つことはなく BASIC タイプに該当する。

フォーマット

```
#pragma acpt TASK ( Bid ET BT Op PN )
                        ( DN ( Did CT ) …)
```

各要素の説明

- Bid ：タスクブロック識別子
- ET ：実行時間
- BT ：タスクブロックのタイプ
- Op ：オプション
- PN ：使用プロセス数
- DN ：直接先行ブロック数
- Did ：直接先行ブロック識別子
- CT ：直接先行ブロックとの通信時間

図 3．3　タスク情報インタフェース のフォーマット

次に，オプション情報である“Op”はブロックタイプ別に必要な情報を追加するためのものである。例えば，LOOP タイプのタスクブロックにはそれが DOALL か DOACROSS か, といった情報やイタレーションの繰り返し回数が記述される。また FUNC タイプでは呼び出される関数名が記述される。

使用プロセス数はタスクブロック内部を処理するプロセス数を表し，タスク粒度解析と部分タスクスケジューリングによって記述される。

“ET(実行時間)”や“CT(先行制約のブロックとの通信時間)”は実行時間解析時に情報が追加される。そして，“DN”や“Did”は，並列性解析時に得た先行ブロック数と先行ブロック識別子情報がそれぞれ表される。このとき，“(Did CT)”は先行ブロック数“DN ”の個数分だけ列挙される。

具体的なタスク情報インタフェースの例を図 3.4 と図 3.5 に示す。実際には，他の情報インタフェースも記述されるが，この例ではタスク情報インタフェース以外は省略している。図 3.5 で使用されている#pragma acpt + Block Start と#pragma acpt + Block End は，複数のステートメントを囲む場合やマクロタスクブロックの有効範囲を表わし，ネストされた形で表すことができる。また，各タスクブロックの実行時間と通信時間がそれぞれ JOB()と COMM()となっているが, これらは実行時間解析時に各タスクブロックの実行時間とタスクブロック間の通信時間に自動的に置換されることになる。

図 3.4 は BASIC タイプのタスクブロックと FUNC タイプのタスクブロックに対するタスク情報インタフェースの例である。この例ではタスクブロック４が 2 つの先行するタスクブロック２と３を持つ事を示している。この場合は，タスクブロック２で書き込まれた変数 b とタスクブロック３で書き込まれた変数 c をタスクブロック４が読み込むため，このような形で先行制約が表されている。タスクブロック５では，ユーザ定義の関数が呼ばれ，関数名が func1 であることを表わしている。

```
a=9;
b=0;
c=3;
d=b*c;
func1( );
```

```
#pragma acpt TASK ( 1 JOB() BASIC 1 ) ( 0 )
 a=9;
#pragma acpt TASK ( 2 JOB() BASIC 1 ) ( 0 )
 b=0;
#pragma acpt TASK ( 3 JOB() BASIC 1 ) ( 0 )
 c=3;
#pragma acpt TASK ( 4 JOB() BASIC 1 )
                                ( 2 ( 2 ... ) ( 3 .... ) )
 d=b*c;
#pragma acpt TASK ( 5 JOB() FUNC  func1 1 ) ( 0 )
 func1( );
```

図 3．4　タスク情報インタフェース例 1

```
for( i=0; i<100; i++ ){
  array[i] = array[i] + 2;
  Data[i] = array[i] * 5;
}
```

```
#pragma acpt TASK ( 5 JOB() LOOP DOALL  100  1 )
                                          ( 1 ( 1  ...) )
#pragma acpt + Block Start
 for( i=0; i<100; i++ ){
#pragma acpt TASK ( 5-1 JOB() BASIC 1 ) ( 0 )
   array[i] = array[i] + 2;
#pragma acpt TASK ( 5-1 JOB() BASIC 1 )
                                    ( 1 ( 5-1 COMM() ) )
   Data[i] = array[i] * 5;
 }
#pragma acpt + Block End
```

図 3．5　タスク情報インタフェース例 2

図 3.5 は BASIC と LOOP に対するタスク情報インタフェースの例である。タスクブロック 5 を見ると，LOOP タイプのタスクブロックとして判定されていることが分かる。また並列性解析により，このループ文が DOALL 処理可能で, イタレーション数が 100 回であるという情報が付加されている。

(1-1)　使用プロセス数の追加

“PN”で表される数はマクロタスク内部を処理するプロセス数である。並列性解析後のタスク情報インタフ

ェースには，どのタスクも 1 プロセスで処理されるようになっている。

階層的に，マクロタスク内部の部分タスクスケジューリングを行った結果から使用プロセス数が置き換えられ，この数をもとにマクロタスクをスケジューリングする。階層的にスケジューリングを行うため，割り当てプロセス数が足りない場合などは，内部のマクロタスクでの使用プロセス数を調節し，最適なプロセスの割り当て数を解析するために使用される。

(1-2) SELECT ブロックの変更点

制御文全体を SELECT ブロックとし，内部に IF, ELSE_IF, ELSE ブロックとして，それぞれ if, else if, else 文を表す。SELECT ブロックの実行時間には IF, ELSE_IF, ELSE ブロックそれぞれの実行時間を比較し，実行時間がもっとも大きい値をとる。図 3.6 のように SELECT ブロック内部のタスクである，タスク 1-1 とタスク 1-2 を比較したとき，大きい値がタスク 1 の実行時間に置き換えられる。これは部分的には最長の処理を行うケースでの最短の並列処理時間を求めるためである。

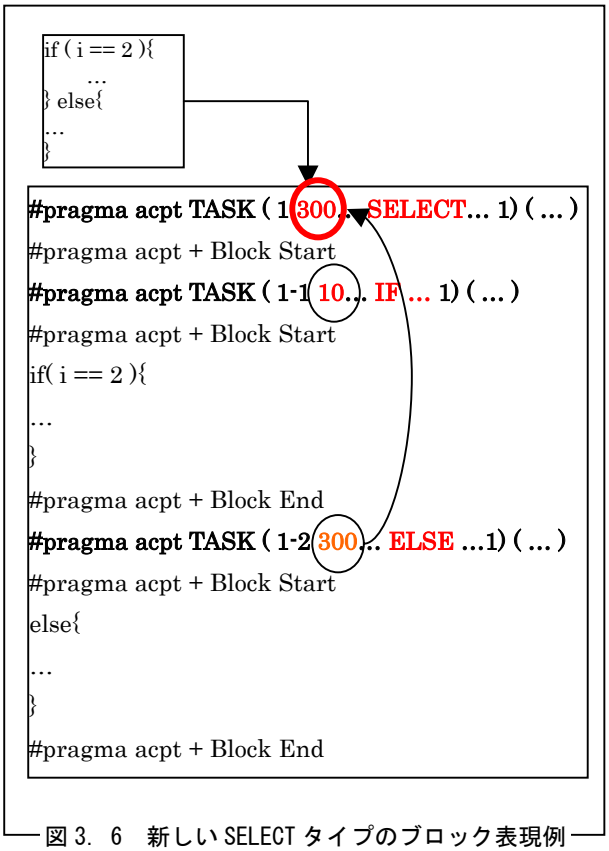


図 3. 6　新しい SELECT タイプのブロック表現例

(1-3) 融合されたタスク表現の追加

図 3.7 はタスク粒度解析によってタスクを融合していく様子を表わしている。融合されたタスクは新たなタスク番号が割り振られ階層が深くなる。融合後のタスクの範囲は“Block Start Merge”，“Block End”の記述で示す。“Ignore”の表記は実行時間が計測済みのタスクであることを示している。

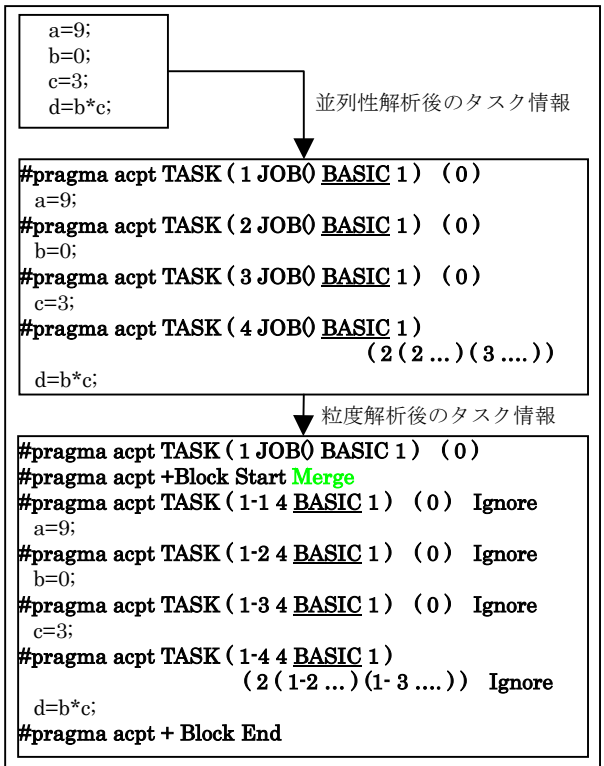


図 3. 7　融合されたタスク情報例

(2) 通信情報インタフェースと変更点

通信情報インタフェースとは，並列化を考える際に必要となるデータの流れを表すものであり，タスク間の先行制約を通るデータの流れが表現されている。記述される具体的な内容は以下の通りである。

- データパック通信を行うデータ情報
- 送信タスクブロック識別子
- 通信のタイプ
- 通信されるデータの詳細
- 並列性解析時に見出された依存元タスクブロック識別子と依存先タスクブロック識別子

図 3.8 は通信情報のフォーマットを表している。“(SendBid …)”は PackDataCount 分だけ記述される。“(CommSet) …”は CommSetCount の個数分だけ列挙されることを意味している。

このインタフェースは送受信を行うタスクブロック識別子と並列性解析時に見出されたタスクブロック識別子の両方を表現している。例えば LOOP タイプのマクロタスクブロックのように，その内部へ依存が存在する場合，実際にデータ通信を行うのはそのループ文が開始される直前でなければならない。もし，そのル

フォーマット

#pragma acpt DATA(PackDataCount (SendBid
CommType DataInfo
CommSetCount （ CommSet ）… ）…)
#pragma acpt DATA (…)
…

各要素の説明

- PackDataCount : パックを行う通信データの個数
- SendBid : 送信側タスクブロック識別子
- CommType : 通信のタイプ
- DataInfo : 通信データ情報
  - DataName : 変数名
  - DataType : 変数の型 (C 言語と同じ変数の型)
  - PointerDim : ポインタの次元数
  - DataCount : 通信データの個数
  - DataIndex : 通信データ DataName からの位置 (スカラ変数は0)
- CommSetCount: CommSet の個数
- CommSet : 並列性解析時のタスクブロック情報
  - SrcBid : 依存元タスク識別子
  - DestBid : 依存先タスク識別子

図 3．8 通信情報インタフェースのフォーマット

ープ文の内部で受信操作を行ってしまうと，ループの回転数分だけ受信操作を繰り返してしまうことになり並列コードとして正しくない。同様に，マクロタスクブロック内部から外部に向かって依存が存在する場合にはそのマクロタスクブロックの終了直後にデータの送信が行われる。そして，マクロタスクブロックはその処理を複数のプロセスにより並列に実行される可能性が高い。その際にデータの送受信をどのプロセスが行うのかということを明確に表現するためにも並列性解析による依存情報が記述されることになる。

通信のタイプは 2 つあり，並列性解析時にフロー依存(RAW)を持つものは COMM，逆依存(WAR)や出力依存(WAW)を持つものは BARRIER という分類を行っている。通信タイプが COMM の場合，異なるプロセスで処理される場合データ通信が起こる。通信タイプが BARRIER の場合は，異なるプロセスで処理される場合はローカルな情報の変更が起こらないため通信は起こらず，同一プロセス上で実行されるように割り当てられた場合のみ，実行順序を逐次と合わせる必要がある。

通信データ情報とは，ある先行制約を流れるデータの詳細な情報を持つもので，変数名やその個数，配列データ中の参照開始位置，ポインタの情報などを含んでいる。

従来は別ファイルとして扱われていた通信情報インタフェースは，タスク情報インタフェースからどのタスクの情報かを識別できるため，受信側のタスク識別子情報は不要となった。また，通信データの数は連続して記述することでその個数が分かり，通信データが存在しない場合には“0”が記述される。

なお，データパック通信を考慮できるように変更を行ったため，複数のデータをパックして送受信を行うデータの情報の数を PackDataCount が示している。PackDataCount の数が 2 以上であればパックを行って通信を行う。

(3) 詳細情報インタフェース

並列化の際に必要となるタスクに関する詳細な情報を含み，それはタスクタイプごとに異なる情報が記述される。現在，詳細な情報を持つ LOOP タイプのタスクにのみ記述される。LOOP タイプのタスクには，ループ回転インデックス，初期値，終了値，増減幅などが記述される。特に，DOALL タイプのループを並列化する際に使用される。

(4) リダクション情報インタフェース

リダクション操作を望むループに，どのようなリダクションを行うかを指示する情報が含まれる。簡単なリダクションループであれば並列性解析によって解析される。また，ユーザによる指示も可能であるようなフォーマットを意識し，“(演算子 : 変数名 : 変数データ型)”で表される。

現在，特に詳細情報がある LOOP にのみ有効であり，情報があるときのみ記述可能である。

3．2．2 スケジューリング結果情報と変更点

従来は，タスク融合が行われるタスクグループ情報と処理するプロセス番号，処理順序の情報が記述されていた。しかし，3.2.1 節(1-3)で述べたように本論文では，融合情報はソースコードインタフェース上に記述されるため，含まれないように変更された。

ここでは部分タスクスケジューリングおよび，全体タスクスケジューリング結果が記述される。

４．並列化コード生成フェーズの概要

従来の並列化コード生成部で行っている処理と，新たに追加したコード生成処理について説明する。

4．1　従来の並列化コード生成

従来のプロセスランク別の処理コード生成フェーズでは，各タスクブロックが処理する実行プログラムコードを MPI を用いて SPMD 型プログラムモデルとして記述するには，プロセス識別子であるランク番号を用いて各タスクブロックの割り当て制御を行えるように処理コードを記述される。

また，１対１通信命令には，送信側はブロッキング命令である“MPI_Send”命令，受信側はノンブロッキング命令である“MPI_Irecv+MPI_Waitall”を使用してデータ通信が行われる。特に，マクロタスクブロック間の通信命令には，処理プロセスが複数あるため，通信情報インタフェース上に記述された依存関係から適切なプロセスへと通信がおこなわれるようになっている。

そして，DOALL ループは配列のブロック分割，リダクション演算を行い，可能であれば各イタレーション内部も並列に実行されるようになっている。

従来の並列化コード生成部についての詳細は，参考文献 3)を参照してもらい，ここでは省略する。

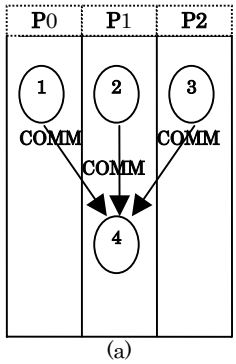
4.2　拡張した並列化コード

並列コード生成において新たに，パック・アンパックを使用した通信コード，連続した DOALL ループに関する扱いを加えた。

4．2．1　パッキング通信

タスクブロック間で通信するデータが複数ある場合，従来のタスクブロック間 1 対 1 通信は図 4.4 に示すとおりである。(b)のコードセグメントで P2 から P1 への通信が行われているとき，通信するそれぞれのデータに対して個別に通信処理を行っている。しかし，並列処理において通信回数の増加はオーバヘッドの増加につながる。また，タスクスケジューリングでは(a)のタスクグラフを生成するためこのようなコードは正しくない。同一プロセッサ間での複数データの通信には，MPI などの並列化言語で実装されているパック・アンパック処理を使用することで通信回数が最適化される。図 4.6(b')が新たに，パック・アンパックを考慮して生成されたコードセグメントである。

パック・アンパックのためのデータバッファが必要となるため，データバッファを全通信の中で最大通信データサイズだけメモリを割り当てる。“MPI_Pack_size”を使用するので MPI の初期化処理を行うときにバッファ領域を確保する。これは図 4.6 のように，MPI の初期化を行う関数内で最初に１回だけ処理される。



(a)

```
if( MyRank==0 ){
   ……   /* タスクブロック 1*/
   MPI_Send( &X, … … 1 … … );
}
if( MyRank==1 ){
   ……/* タスクブロック 2*/
}
if( MyRank==2 ){
   ……/* タスクブロック 3*/
   MPI_Send( &Y, … … 1 … … );
   MPI_Send( &Z, … … 1 … … );
}
if( MyRank==1 ){
   MPI_Irecv(&X,..0 ,reQuest );
   MPI_Irecv(&Y,..2,reQuest+1 );
   MPI_Irecv(&Z,..2, reQuest+2 );
   MPI_Waitall( 3 , reQuest, ….);
   ……/* タスクブロック 4*/

}
```

(b)

図 4．4　従来のタスクブロック間通信例

```
if( MyRank==0 ){
   ……   /* タスクブロック 1*/
   MPI_Send( &X, … … 1 … … );
}
if( MyRank==1 ){
   ……/* タスクブロック 2*/
}
if( MyRank==2 ){
   ……/* タスクブロック 3*/
   MPI_Pack( _static_Pack_buffer,…, );
   MPI_Pack( _static_Pack_buffer,…, );
   MPI_Send( _static_Pack_buffer, … … 1 … … );
}
if( MyRank==1 ){
   MPI_Irecv( &X,..0 ,.. reQuest );
   MPI_Irecv( _static_Pack_buffer,..1 ,.. reQuest+1 );
   MPI_Waitall( 2 , reQuest, ….);
   MPI_Unpack(_static_Pack_buffer, … );
   MPI_Unpack(_static_Pack_buffer, … );
   ……/* タスクブロック 4*/
}
```

(b')

図 4．5　パック・アンパックを
使用した１対１通信コード

```
void MPI_Initialize_acpt(int argc, char **argv)
{              /* 初期化関数 */
   MPI_Pack_size( … , …&_Pack_memSize )
   _Pack_memMax += _Pack_memSize;
   …
   _Pack_buffer = malloc( _Pack_memMax );
}
```

図 4．6　初期化関数内に記述されるパッキング
データバッファ領域確保を行うコード

4. 2. 2　DOALL ループのデータの分散・集約

従来の並列化コード生成フェーズでは，DOALL ループ前後ではデータの分散･集約を行っている。ここでは，分散・集約を行う通信データが複数ある場合，パッキング通信を行うように変更した。特に配列データを含む場合は，単一データに加えブロック分割した配列データをそれぞれのプロセスにまとめて分散・集約するようなコードとなる。

4. 2. 3　連続した DOALL ループの扱い

図 4.7 のように先行制約のタスクブロックも DOALL ループであり，内部を 1 プロセスで処理し，同じプロセスのグループによって処理されるような DOALL ループではデータの集約，分散を行わないように 2 つの DOALL ループを融合する。データの分散・集約は次のタスクがどのよう場合でも安全なコードとして動作するためには必要な処理である。しかし，DOALL ループでのデータの分散・集約の通信にかかる時間は大きく，このようなパターンの場合削減できるところは削減すべきと考える。

なお，タスク融合が現在，BASIC タイプのタスクの融合のみしか考慮していないため，暫定的な融合を並列コード生成部で行っている。

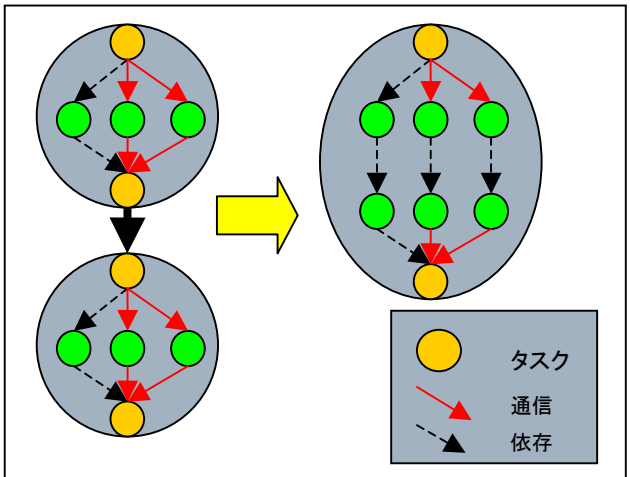


図 4. 7　連続した DOALL ループの例

5．評　価

評価対象として台形法を用いた積分計算プログラムの自動並列化コードを生成した。最終的な積分値が解析対象コードを逐次実行した結果と同じになっていることから正しく並列化されたことが確認できた。その上で，並列化コード生成部で考えた並列化コードを生成し，それぞれの実並列処理時間を計測した。

ここで，評価に使用したプログラムは 48 ノード構成の PC クラスタ上で実行した。各ノードはすべて CPU に Pentium4 (3.0 GHz)，メモリに 512MB，LAN に Gigabit Ether の性能を持つ。各ノードの相互接続にはギガビットスイッチングハブを用いている。

この環境下で，今回評価に使用したプログラムを 8 プロセッサで実行した。

5. 1　並列化コード

図 5.1，5.2 でそれぞれの並列化コード生成によって生成された評価プログラム中のある DOALL ループのデータ分散コードを例に示す。

PCG 1 は美濃本氏によって考えられた並列コード生成[3]によるもの，PCG1.1 は PCG1 に加えてパッキング通信を考慮したもの，PCG1.2 は PCG1.1 に加え連続した DOALL ループ融合を考慮した並列化コードを表わしている。なお，PCG1.2 では連続した DOALL ループと判断されたため，データ分散コードが記述されることはなかったため省略する。

```
…
if ( _Sub_Rank_000==0 ) {
      /* X0 と X1 の送信 */
    MPI_Send( X1+1250000, 1250000, ..., 1, ..., ...);
    MPI_Send( X1+2500000, 1250000, ..., 2, ..., ...);
    …
    MPI_Send( X0+7500000, 1250000, ..., 6, ..., ...);
    MPI_Send( X0+8750000, 1250000, ..., 7, ..., ...);
}
if ( _Sub_Rank_000!=0 ) {
     /* X0 と X1 の受信　*/
    MPI_Irecv( X1+(_Sub_Rank_000/1)*1250000,...,...);
    MPI_Irecv( X0+(_Sub_Rank_000/1)*1250000,...,...);
    MPI_Waitall( 2, _mPi_request, _mPi_status );
}
_LpStart_000=(_Sub_Rank_000/1)*1250000,
_LpEnd_000=(_Sub_Rank_000/1)*1250000+1250000;

for( i=_LpStart_000; i<_LpEnd_000; i=i+1 ) {
    DATA[i] = X0[i]*X0[i];
     …
     DATA[i] = DATA[i] + 2.0*X1[i];
     DATA[i] = DATA[i]*h/2;
```

図 5. 1　PCG1 によるデータ分散コード

```
   …
if ( _Sub_Rank_000==0 ) {   /* X0 と X1 のパック  */
      MPI_Pack(X0+1250000*0, 1250000,…,…);
      MPI_Pack(X1+1250000*0, 1250000,…,…);
     …
}
/*パッキング通信*/
MPI_Scatter(static_Pack_buffer,…, …);
if ( _Sub_Rank_000!=0 ) {   /* X0 と X1 のアンパック  */
      MPI_Unpack(static_Pack_buffer, …, …);
      MPI_Unpack(static_Pack_buffer, …, …);
}
_LpStart_000=(_Sub_Rank_000/1)*1250000,
_LpEnd_000=(_Sub_Rank_000/1)*1250000+1250000;
for( i=_LpStart_000; i<_LpEnd_000; i=i+1 ) {
     DATA[i] = X0[i]*X0[i];
      …
      DATA[i] = DATA[i] + 2.0*X1[i];
      DATA[i] = DATA[i]*h/2;
}
```

図 5.2　PCG1.1 によるデータ分散コード

5. 2　処理時間

表 5.1　並列化コード処理時間　　(sec)

逐 次	ＰＣＧ１	ＰＣＧ１．１	ＰＣＧ１．２
0.420	35.7	36.1	0.0575

各並列プログラムを実行した処理時間を表 5.1 に示す。

PCG1.1 では，パッキング通信による高速化がみられなかった。すなわち予想した通信回数の減少による通信を行う際のオーバヘッド時間の減少効果が得られなかった。これは今回使用した PC クラスタ環境のスイッチの性能がよかったためではないかと考える。従って，パッキングによる処理時間の向上はマシン環境によっては十分効果的と考えられる。

PCG1.2 では，DOALL ループの融合によるデータの分散・集約の通信時間削減によって逐次処理に比べて 13%の処理時間まで減少した。これは，いくつかある DOALL ループのうち開始と終了時に行っていたデータの分散･集約のための通信命令が削除されたためである。

６．終わりに

6. 1　まとめ

マクロタスクを考慮した流れへと変更したことで，精度の高いタスク粒度に関する情報が表わされるようになった。また，今回のように，共有リソースを一元管理することで解析間での情報の交換がスムーズにいくと考えられる。

並列化コード生成部では，このステージでしか解析できないような DOALL ループの融合処理を行うことで大幅な処理時間の向上をすることができた。パッキング通信は残念ながら期待した効果を得ることはできなかったが，パッキング通信の必要性は明らかなので必要な機能であると考えられる。

6. 2　今後に向けて

● 効率のよい通信コードの生成

MPI の関数にはその汎用性や移植性から，内在するソフトウエアオーバヘッドが大きいことが分かっている。[4] 従って，同じ動作をする通信コードではなるべく関数の呼び出しを少なくする必要がある。

● ユーザの知識を利用する方法の検討 [2]

共有リソースの一元化をした結果，一方でタスクの情報を開発ユーザが読解するには情報が多すぎるため可読性が低い。今後，より詳細な並列化情報をソースコード上に記述しかつ，その情報を開発ユーザがわかりやすい形で表現することは困難である。

また，実際には最適な並列化コードの候補がいくつもある場合がある。また，開発ユーザが目的とした環境によっても意図するコードが異なる。生成コードの候補を選択できるようにすることも重要である。

今後は，開発ユーザと，システム全体とのインタラクティブな関係を構築するためのツールが必要と考える。これは，共有リソースを基にしたコード生成に開発ユーザからのアプローチを並列化コードに反映させるものであり，例えば，最適な通信コードの候補からの生成される通信の選択，また，タスク粒度・スケジューリング結果の手動での調整を可能にする機能などが考えられる。

● 並列性の強化，特に制御依存による曖昧性の削減

制御依存の曖昧性により通信のパスを一意に決めることが出来ない現状となっている。より並列性解析時の動的解析などにより制御依存への対応を強化していくべきだろう。

● ポインタ変数に対する並列化

分散メモリアーキテクチャ上で通信処理をする以上各プロセス内でのメモリアドレス空間は異なるものとなる。このようなアーキテクチャに対応したより詳細なポインタ解析を行う必要がある。

● より多くの DOALL ループタイプの並列化

DOALL ループの並列化に関しては単純に DOALL 処理可能であるからといって単にループのイタレーションを分割すればよいだけではない場合も多々ある。このような場合に備え，ループ文の持つ固有の情報の取得，内部で使用される配列のアクセスパターンなどの詳細な解析が必要となるだろう。

参考文献

1) TOP500, http://www.top500.org/ , 2007 年 03 月現在参照可

2) 平成 13 年度 アドバンスト並列化コンパイラ技術報告書（概要編）, 財団法人日本情報処理開発協会

3) 美濃本　一浩, 甲斐　宗徳. 「C 言語自動並列化トランスレータの開発」. 成蹊大学理工学研究報告, Vol.42, No.1, pp.41-49，2005 年 6 月

4) Hirotaka Ogawa : “OMPI: Optimizing MPI programs using Partial Evaluation”, ACM/IEEE Supercomputing Conference , 1996 年

5) “Message Passing Interface Forum -MPI2-” , http://www.mpi-forum.org/ , 2007 年 03 月現在参照可

6) “The Message Passing Interface (MPI) standard” , http://www-unix.mcs.anl.gov/mpi/ , 2007 年 03 月現在参照可

7) Gropp, Lusk, and Skjellum : “Using MPI second Edition” , MIT Press , 1999 年

8) Gropp, Lusk, Thakur : “Using MPI2”, MIT Press , 1999 年

9) 平林雅英 : “ANSI C 言語辞典”, 技術評論社, 1997 年