成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.42 No.1 (2005) pp.41-49

# C 言語自動並列化トランスレータの開発

美濃本 一浩*1，甲斐　宗徳*2

# Implementation of Automatic Translator from C Programs to Parallel Programs using MPI

Kazuhiro MINOMOTO*1,　Munenori KAI*2

**ABSTRACT** : The work of automatic parallelization can be divided and considered on each stage, that is, the parallelism analysis of a source program, execution-time analysis, task grain-size analysis, task scheduling, and parallelized code generation. Although there was the result of research individual about each stage, unifying and treating them to one was not realized. Then, in developing an automatic parallelization translator, we proposed and implemented the common resource for summarizing all the analysis stages as a series of flows. A common resource plays the role which tells the result of each analysis stage of a C automatic parallelization translator as an input of the next analysis stage. A parallelized code can be automatically generated now from a source program using this common resource. As a parallelized code, MPI, which is a message communication library of the present industry standard, is used. The contents of the common resource were changed into the suitable communication command using MPI, and parallelization in a DOALL loop and a block were performed. Moreover, the analysis stage which takes the lead for fully improving parallel processing performance, and the tuning method in parallel code conversion, etc. are clarified.

**Keywords** : Parallelism Analysis, Parallelized Translator, Task Scheduling, Parallel Processing, MPI

(Received March 25, 2005)

## 1．はじめに

### 1．1　背景

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，近年のスーパーコンピュータの多くに利用され，高い性能を実現している[1)]ことからも明らかである。そして，プロセッサ単独での速度向上も限界に近づきつつあることを考えると，マルチプロセッサシステムは今後もより一層重要な基盤技術になっていくであろう。

しかし，マルチプロセッサシステムには，並列性と効率を考慮したプログラムで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。故に，プログラムを並列処理向きに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大きな障害になっている。

### 1．2　研究目的

上記のような問題点を解決するための手段として，自動並列化コンパイラの利用が挙げられる。しかし，マルチプロセッサシステムの性能を十分に発揮させることができず，プログラムの実行性能をかえって低下させてしまう傾向を示している[2)]，というのが現状である。そこで，プログラム内にある単純なループの並列性のみを抽出するような単一粒度を対象とした並列化を行うのではなく，関数や関数を含んだループ及びループ間などの粗粒度の並列性や，細粒度であるステートメントレベルの並列性を引き出すことも，今後の自動並列化コンパイラには望まれている[3)]。

*1 情報処理専攻大学院生
*2 情報処理専攻助教授(kai@st.seikei.ac.jp)
Associate Professor, Dept. of Information Sciences

本研究では，実用的な自動並列化コンパイラが存在していないC言語に対してそのような複数粒度の並列化を行うことで，プログラムの実行性能の向上を図る自動並列化Cコンパイラの完成を最終目的としている。

## 2．自動並列化Cコンパイラ

自動並列化Cコンパイラとは，C言語で記述されたソースプログラム内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。本研究で開発する自動並列化Cコンパイラの全体

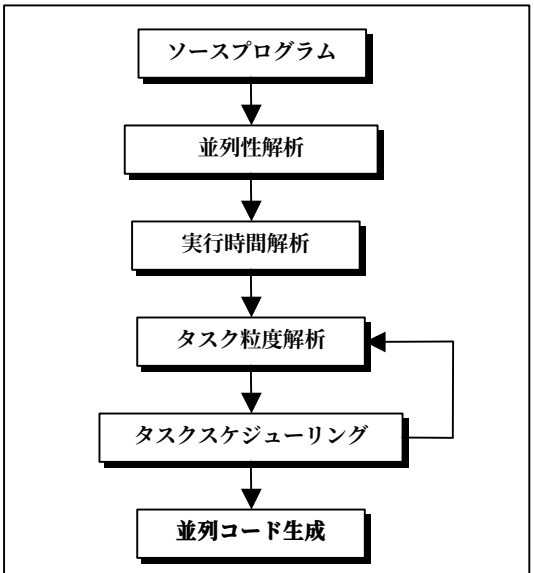


図2.1　自動並列化Cコンパイラの処理の流れ

的な処理の流れを図2.1に示す。

初期段階の作業として，解析対象となるソースプログラムの実行時間解析と並列性解析を行う。並列性解析では，プログラム内に存在するプログラムの一行(以下，ステートメント)レベルでの並列性を抽出する。実行時間解析では，ステートメントごとの実行時間を求める。

次の段階の作業として，タスク粒度解析とタスクスケジューリングを行う。タスク粒度解析では，実行時間解析と並列性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よ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。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よいのかを決める。

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，Message Passing Interface(以下，MPI)を用いたC言語プログラムの自動並列化トランスレータの開発を行う。ここでのトランスレータとはC言語で書かれたソースプログラムをMPIを用いたC言語ベースのプログラムへと変換することである。実行時には生成されたソースプログラムをMPIライブラリと合わせてコンパイルし，実行ファイルに直してから実行する。MPIを用いることでより汎用性・移植性の高いコード生成が出来る。動作環境としては分散メモリアーキテクチャ上を想定している。プログラムモデルとしてはSingle Program Multiple Data(SPMD)型での自動生成を行う。

MPIは，現在の業界標準のメッセージ交換用のライブラリである。マルチプロセッサ上で並列に実行することに伴い，シングルプロセッサでは必要のなかったデータ通信処理を加えなければ正しい答を導き出せなくなる。この不足分を補うために必要最小限度，通信を介してプロセッサにローカルなメモリにあるデータのやり取りをする。この通信の機能とその呼び出し方法を規定しているのがMPIである。MPIは規格であり，MPIフォーラムという会議によって規格されている。

メッセージパッシングによるプログラミングは基本的に分散メモリアーキテクチャのためのモデルである。

並列計算でのモデルには図2.2に示されるようにSPMD( Single Program Multiple Data )型とMPMD ( Multiple Program Multiple Data )型が存在する。SPMD型プログラミングモデルというのは図2.2(a)で表されているように一つのプログラムで並列処理を記述するものである。MPMD型プログラミングモデルは図2.2(b)で表されるように複数のプログラムを協調的に動作させて並列処理を行うものである。またMPIを用いた並列処理では各プロセッサ上にプロセスを生成し処理を並列に行う。その際には各プロセスを識別する必要がある。MPIでは各プロセスの識別を行うためにランクという番号が0番より順々に割り振られる。

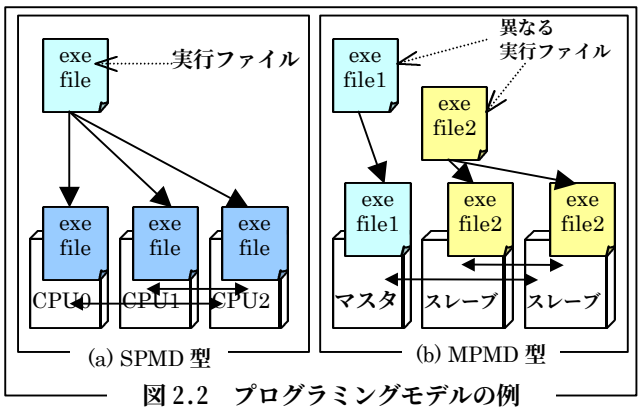


図2.2　プログラミングモデルの例

## 3．共有リソースとインタフェース

昨年度までは図2.1に示した各解析フェーズにおいて，各々の解析結果は後の解析に対して適した形とは言えなかった。そこで各解析間に連携を持たせ，一つの流れとして仕上げることを考えた。そのためには，全ての解析間で使用できるリソースの共有が必要である。その共有可能なリソースをインタフェースとして，各解析は情報を追加していく。そして本年度の最終形であるMPIコードの自動生成を行う。以下に共有リソースの詳細とその共有リソースを使用する各解析間でのインタフェースとしての使われ方について説明する

### 3．1　共有リソース

共有リソースは2つあり，一つはソースコードインタフェースであり，もう一つは通信情報インタフェースである。この2つのリソースは並列性解析の結果により生成される。そして実行時間解析，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，解析された情報を順々に追加していくことになる。最終的には全ての情報を並列コード生成部で読み込み並列化のコードを生成することになる。図 3.1 は各解析フェーズが共有リソースである2つの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状態を表している。

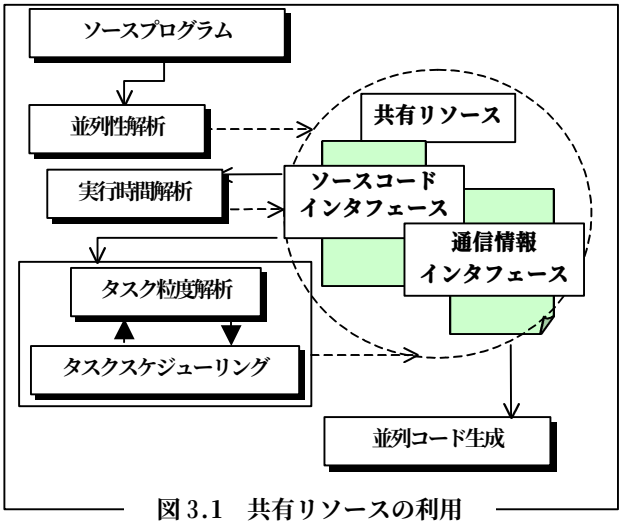


図 3.1　共有リソースの利用

またタスク粒度解析・タスクスケジューリングフェーズでは，共有リソースに対してコード並列生成になくてはならない重要なスケジュールや各タスクに対するプロセスの割り当ての情報が求められる。

次に共有リソースの2つのインタフェースについて説明する。

#### 3．1．1　ソースコードインタフェース

ソースコードインタフェースの内部ではタスクブロックと呼ばれる単位で情報が表現されている。タスクブロックとは並列コード生成時に各プロセスが実行する最小単位であり，最小の粒度としてはプログラムのステートメントと等価な物である。このタスクブロックは解析対象のソースコードに#pragma ディレクティブを挿入することで表現される。#pragma ディレクティブとはプリプロセス命令であり，処理系の実装に依存した方法で指示を出すことが出来る。#pragma ディレクティブは各ベンダにおいても独自の形で多々利用されている。本年度の研究では **#pragma acpt** で始まる #pragma ディレクティブを用いている。#pragma に続く acpt は本研究の題目である C 言語並列化トランスレータの略称である。

ソースコードインタフェースを用いることにより表現することが出来る情報は以下の通りである。

- タスクブロックの区間情報
- タスクブロックの識別子
- タスクブロックのタイプ
- タスクブロックの大きさ
- タスクブロックの実行時間
- 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

これらの情報を表現する#pragma ディレクティブのフォーマットは図 3.2 で示すとおりである。 図 3.2 のフォーマットで記述されている“***[*]***” は，“Dct”の個数分だけ“**( Src CT )**” が繰り返されることを意味している。最初の小括弧で表されるのが，この#pragma ディレクティブにより表される個々のタスクブロック情報である。その次に記述される小括弧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・SELECT_ELSE・FUNC** などがある。

フォーマット

**#pragma acpt （ Bid ET BT Op ）( Dct ( Src 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
- Dct：直接先行ブロック数
- Src：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3.2　ソースコードインタフェースのフォーマット

BASIC タイプは基本タイプであり，C プログラム内の全ての情報を表現することが出来る。LOOP タイプはそのブロックが C プログラム内の for や while 文であることを意味する。そして SELECT・SELECT_ELSE タイプはそれが if-then-else 文であることを示している。そして FUNC タイプはそのステートメントが解析対象プログラム内にユーザが定義した関数の呼び出しであることを意味している。つまりライブラリ関数の呼び出し時に FUNC タイプを持つことはない。

次に，オプション情報である OP はこのブロックタイプ別に必要な情報が追加されることになる。特に LOOP タイプのタスクブロック時にはその LOOP のタイプを DOALL，DOACROSS といった情報や，イタレーションの繰り返し回数，ループを並列実行する時の１イタレーションに費やされるプロセス数，このプロセス数はタスク粒度解析・スケジューリング時に追加される，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。そして 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。具体的なソースコードインタフェース例を図 3.3 と図 3.4 で示す。この図では各タスクブロックの実行時間と通信時間をそれぞれ JOB()と COMM()と表現しているがこの部分に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として自動的に置換されることになる。

図 3.3 は先ほど示した BASIC タイプのタスクブロックのみで表現されるソースコードインタフェースである。この例ではタスクブロック 3 が 2 つの先行するタスクブロック 1 と 2 を持つ事が読み取ることが出来る。この場合は，タスクブロック１で書き込まれた変数ｂとタスクブロック 2 で書き込まれた変数 c をタスクブロックが読み込むため，このような形で先行制約が表されている。

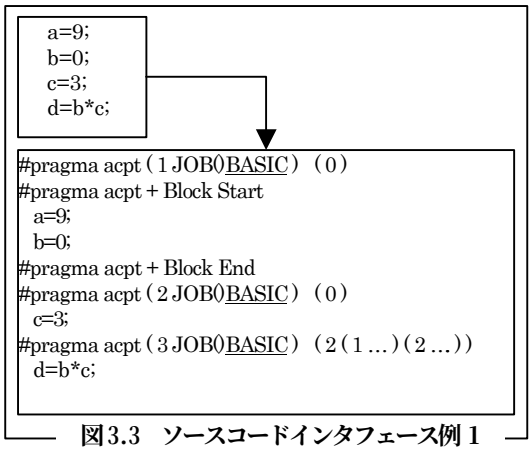


図 3.3　ソースコードインタフェース例１

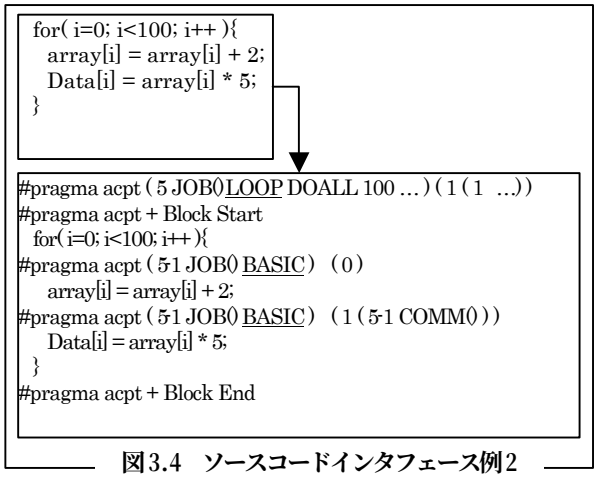


図 3.4　ソースコードインタフェース例 2

図 3.4 はループ文である for 文のソースコードインタフェースである。タスクブロック 5 を見ると，LOOP タイプとして表現されていることが分かる。その後このループ文が DOALL 処理可能でありループのサイクル数が 100 回であると記述されている。

そして, 図 3.3, 図 3.4 両方にて表現されている#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内部に複数のタスクブロックを持つ場合に利用される。

### 3．1．2　通信情報インタフェース

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一元的に管理するものであり，ソースコードインタフェース上で表現される先行制約区間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。

- 送/受信を行う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
- 実際に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

特に注意したいの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ところである。この情報は時に異なる。例えば LOOP タイプのタスクブロックのように内部にネストした形で内部にタスクブロックを持つものをここではマクロタスクブロックと呼び，その内部へ依存のパスが張られる場合，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。このことと同じ理由で，マクロタスクブロック内部から外部に向かって依存のパスが張られる場合で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も記述されることになる。

通信のタイプには 2 種類あり, 一つは COMM であり, もう一つは BARRIER である。図 3.5 で表されるようにこの 2 つの通信のタイプには特徴があり，基本的には並列性解析時に FLOW 依存(RAW)を持つものには COMM。逆依存(WAR)や出力依存(WAW)を持つものには BARRIER というタイプで区別を行っている。通信タイプが COMM の場合，図 3.5(a)を例に取ると，タスクブロック 1 とタスクブロック 3 が仮に異なるプロセス上で実行されるのであるならばタスクブロック 1 はタスクブロック 3 へ変数 X を送らなければならない。もし同一のプロセス上であればその必要はない。通信タイプが BARRIER の場合，基本的に MPI のように，分散メモリアーキテクチャ上で動作する場合においてはそれが異なるプロセス上で実行されたとしてもデータ通信や，プロセス間での同期は起こらない。なぜならばタスクブロック 3 では新しく変数 X への書き込みを行うのであり，それまでに更新された変数 X の値を必要としないためである。また同一のプロセスで実行されるのであればタスクブロック 1 はタスクブロック 3 の後に実行されることが保証される。しかし，異なるプロセス上で実行されるのであるならばタスクブロック 3 が先に実行される可能性がある。タスクブロック 3 以降に変数 X への読み込み命令が発行されるとその読み込み命令先へタスクブロック 3 からのデータ通信が起こる。

通信されるデータの詳細はこの先行制約区間を流れるデータの詳細な情報を持つもので，内部には変数名やその個数，配列データなどに備えた先頭の位置，ポインタの情報などを含んでいる。

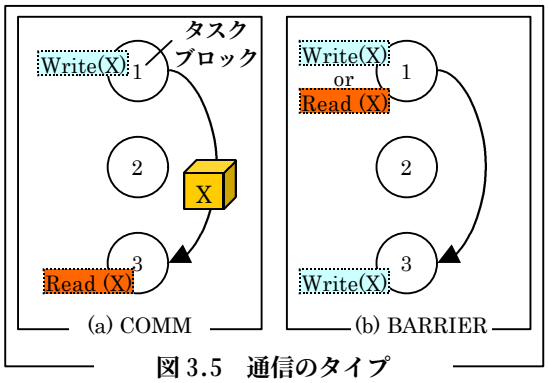


図 3.5　通信のタイプ

## 4．並列コード生成

並列コード生成は図 3.1 で示したように共有リソースを入力として並列コード生成が行われる。ここでは並列コード生成フェーズについて順を追って説明する。

### 4．1　タスクブロック融合とスケジューリングの決定

タスク粒度解析・タスクスケジューリングからの情報は MPI を用いた並列コード生成時になくてはならない必須の情報である。この情報がなければ解析したソースプログラムを最適なスケジューリングにより並列コードとして生成することが出来ない。図 4.1 はタスク粒度解析とタスクスケジューリングにより各タスクのスケジューリングとタスク融合，プロセスへの割り当てが決定されている状態を示している。図 4.1 は，(a)がタスクスケジューリング・粒度解析前，(b)がタスクスケジューリング・粒度解析後である。図 4.1 ではスクスケジューリング・粒度解析の際にタスクが融合され，各タスクを処理するプロセスの識別子，P0,P1 など，が決定されていることが分かる。図 4.1(b)で示されているどのタスクが融合されたのかという情報や各タスクをどのプロセスで実行すればよいのかという情報を並列コード生成は受け取り，その情報を用いてソースコードレベルでタスクブロックの融合や処理プロセスの割り当てを行う。図 4.2 は図 4.1 の(b)で示された結果から各タスクをスケジューリング結果のままプロセスへ割り当てたものである。並列コード生成では図 4.2 に示しているような各プロセスへのタスクの割り当てやプロセス間通信処理などを MPI を用いて記述する。この際 SPMD 型プログラムモデルで表現する。

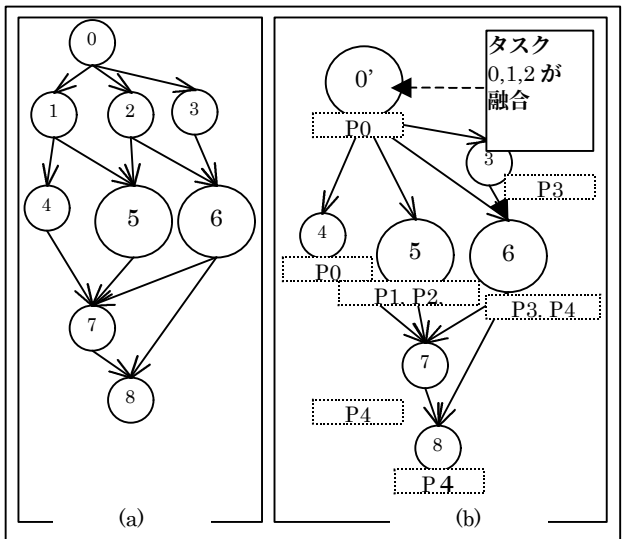


図 4.1　タスクスケジューリング・粒度解析事例

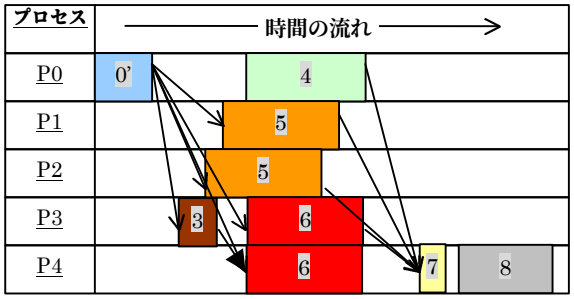


図 4.2　各プロセスへのタスクの割り当て例

## 4．2　プロセスランク別の処理コード生成

先に述べたように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制御を行えるように処理コードを記述すればよい。図 4.3 ではその例を示している。マクロタスクブロックのように複数のプロセスにより実行される場合はプログラムとして記述する際には if( my_rank==N || my_rank==M || …… )と制御間で論理和を取るように記述すればよい。

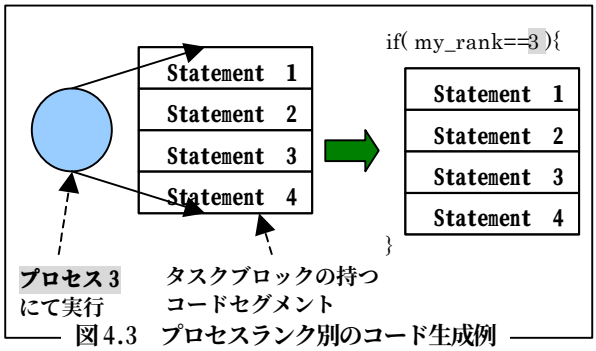


図 4.3　プロセスランク別のコード生成例

## 4．3　タスクブロック間通信

タスクスケジューリングにより各タスクブロックを処理するプロセッサが決定される。この時，処理プロセッサが異なるタスクブロック間に COMM タイプの依存があるならば，その区間に通信が発生することになる。本年度は通信を MPI が定義している MPI_Send 命令によるデータ送信操作と MPI_Irecv 命令によるデータ受信操作により実現した。図 4.4 はその具体的な様子を表している。

図 4.4 は(a)で表現されるタスクブロック間の通信情報を元にデータ通信命令を挟んだコードを(b)で表現している。(a)はタスクブロック 1,2,3 がタスクブロック 4 へ COMM タイプの通信を持つ状態を示している。各タスクブロックを処理するプロセスは示されるとおりであり，特にタスクブロック 2 とタスクブロック 4 は処理を行うプロセスが同じ事から，通信コードが(b)では挿入されていないことが分かる。送受信する際には送信時には相手のランクを指定し，受信時には送信側のランクを指定する。ここでは書かれていないがそのほかにも送受信時にはタグと呼ばれる各メッセージ通信を識別する値を指定することになる。そして全ての通信が確実に行われるように各メッセージに固有のタグを C 言語の列挙子型を用いて実現している。この例では各タスクブロックが唯一つのプロセスにより処理される例を示した。

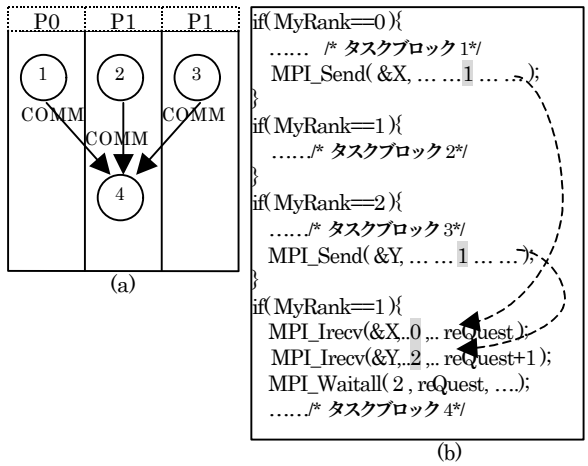


図 4.4　タスクブロック間通信例 1

次に送受信時のタスクブロックがマクロタスクブロックの場合の例を図 4.5 で示す。この場合あるタスクブロックを処理するプロセスが複数個存在することになるため，実際の通信を行うプロセスはその中のいずれかとなる。特にこの例では送信側，受信側共にマクロタスクブロックではあるが，いずれか一方がマクロタスクブロックである場合も同様に，実際の依存関係を元にして，処理するプロセスが適切な場所でデータ通信を行う。通信情報インタフェースにはこのための情報として通信を行うタスクブロック以外に実際にデータが依存しているタスクブロック間に関する情報が記述されている。

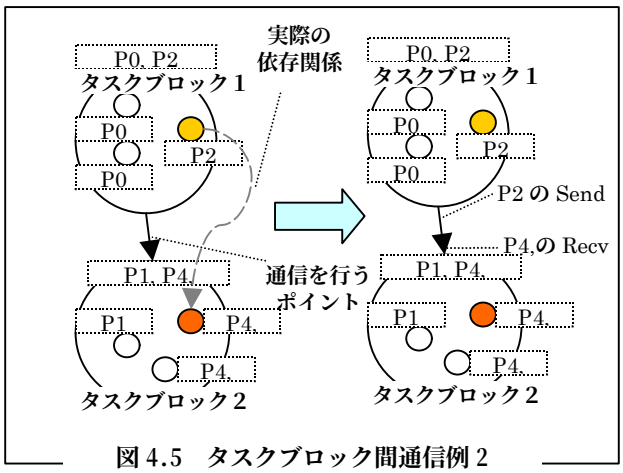


図 4.5　タスクブロック間通信例 2

またタスクブロックが DOALL 処理を可能とするマクロタスクブロックである場合は必要なデータを受信したプロセスがさらに全プロセスに必要なデータを分散し，そして処理後のデータを収集することになる。DOALL の並列化に関しては 4.5 節にて述べる。

### 4．4　通信サブグループの生成

MPI を用いた並列処理ではプロセスの起動の際に，処理する全てのプロセスを持つグループ(コミュニケータ)が生成される。そして，そのプロセスグループの内部でのみ通信が行われることを MPI は保障している。更に，MPI は起動時のグループの内部にサブグループを生成する機能を持っている。本年度はユーザ定義関数部(main 関数を除く)や DOALL ループでは MPI の提供するサブグループを生成する機能を用いて実現した。

本年度の C 言語自動並列化トランスレータではユーザ定義のプログラムのうち main 関数から呼ばれた先の一つ目の関数までを並列化している。特に処理が大きい関数はマクロタスクブロックとなる可能性が高く，呼び出されるたびにその内部を処理するプロセスの集合が異なる可能性がある。このような場合に対応するため，本年度はサブグループを形成し，その内部のみの通信を保障することにした。図 4.6 はその例を示している。

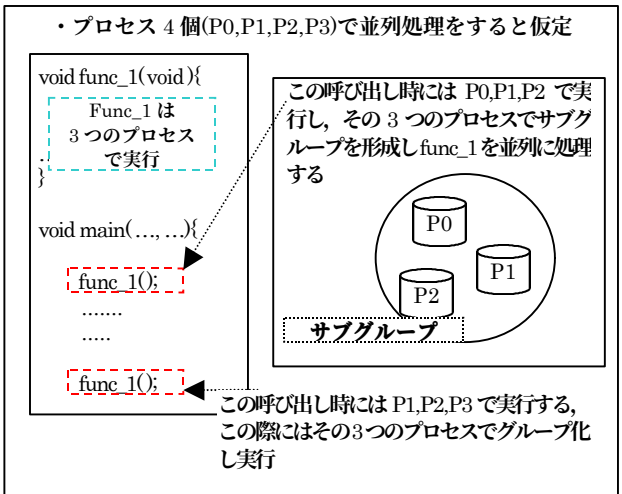


図 4.6　サブグループの生成例

図 4.6 では同一の関数が複数回呼ばれ，呼ばれるごとにその関数を処理するプロセスの要素が異なる場合の例を示している。この場合では関数が呼ばれるごとに，まず呼び出し時のプロセスグループでサブグループの生成を行う。次に関数を処理する。最後に生成したグループの解放を行う。こうすることで複数回同一の関数呼び出しが起こってもソースプログラム上では一つの関数の並列コードの記述で，複数回の呼び出しに対応した処理を行うことが可能となる。また DOALL ループに関しても同様にサブグループを生成し並列に実行を行うようにした。

### 4．5　DOALL ループの並列化

本年度実現した DOALL ループの並列化では，配列データのブロック分割やリダクション演算(総和・総積)ループの自動並列化を行った。4.4 節にて述べたとおり，DOALL ループ全体はサブグループにて実行されることで DOALL ループを処理するプロセスはそのグループ内のみの通信が保障されている。DOALL ループの並列化においてはループの全イタレーションをブロック分割する。そして，その内部のイタレーション自体もスケジューリングにより複数個のプロセスにより処理される可能性があることを考慮し，イタレーション内部でも並列に処理している。

DOALL ループの並列実行には前処理として各プロセスにに必要なデータ分配を行う工程がある。その後 DOALL ループ自体を並列に実行する工程があり，後処理として分散したデータの収集処理を行うことで実現している。

## 5．評　価

本年度は評価の対象として数値演算プログラムに対して自動並列コード生成を試みた。プログラムの詳細は台形法を用いた数値積分プログラムである。コードとしては小規模であるが本年度開発した C 言語自動並列化トランスレータに備わった並列化手法の数々の評価を行うことが出来るものとなっている。また並列コード生成の評価を優先したため，タスク粒度，タスクスケジューリングは相当無駄があると考えられるものを採用している。図 5.1 で示されているのは外部関数の呼び出しが行われた際に自動的に通信グループの生成を行い，処理を行っている例を示している。本システムでは自動並列化を行うと自動的にグループの生成と解放を集中的に行う関数が生成されるこれが

- MPI_CreateSubComm_acpt(…)
- MPI_FreeSubComm_acpt( … )

である。前者がグループの生成，後者がグループの解放を行う関数となっている。この関数を用いて DOALL ループや関数コール時のグループ生成が行われる。図 5.1 では 2 つのグループを生成しているが，一つは関数用のグループ，もう一つは関数内のループ文を実行するのに必要となるグループ生成である。その後 init 関数の実行後生成されたグループの解放を行っているのが分かる。

図 5.2 では DOALL ループ文に対してリダクション演算を用いて総和を求めることが出来るようになった例を示している。この例では特に分割されたイタレーション毎の和を MPI が提供する集団メッセージパッシング命令である MPI_Reduce を用いてまとめることで総和を求めている。最終的な解が解析対象コードを逐次実行した際の結果と同じになっていることからも正しく並列化が

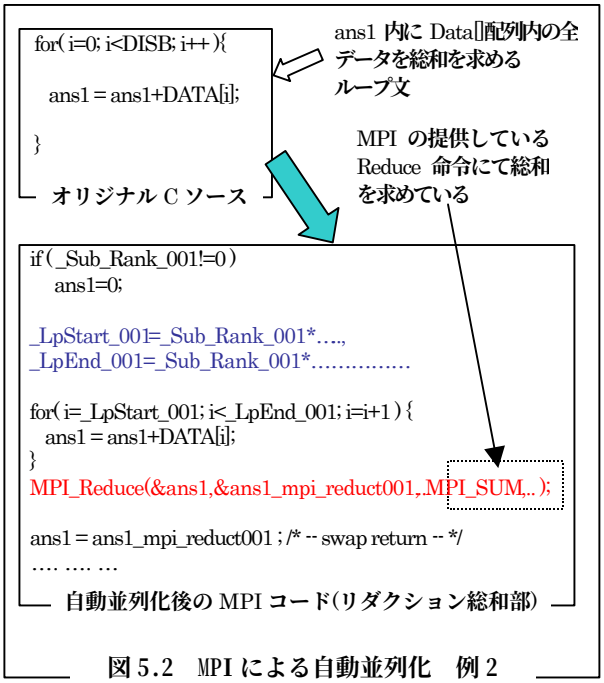


図 5.2 MPI による自動並列化 例 2

行われていることが確認できたと言える。

また，評価に利用したプログラムを成蹊大学学園情報センターに設置してある HPC クラスタマシンにて計測した。そこではＰＣ48 台＋専用管理マシン１台を使用して，2 つのクラスタ環境を構成している。今回はそのうちの 32 台ノードで構成される環境を利用した。各ノード全てに CPU に Pentium4 (2.4GHz)，2 次キャッシュに 512KB，メモリに 512MB，LAN に Gigabit LAN 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の際に使用した並列化プログラムをプロセッサ 4 台利用して動作させたところ，逐次実行に比べて自動並列化プログラムは 225%という実行時間となった。これは通信時間が全体の 80%を占められているためであった。今回評価に利用したプログラムは，DOALL ループが複数個連続するものとなっている。4.5 節で述べたように DOALL ループの実行には必ず前処理と後処理として，データの分散と収集作業を行う。そのため，このような通信時間の肥大化に繋がったものと考えられる。この自動生成されたプログラムに対して，個人的に最適だと考えられるスケジューリングと最適なデータ転送を考慮して変更を加えた並列プログラムの場合は逐次実行に比べ 26%の処理時間となった。この最適なスケジューリングは，DOALL ループ内の各イタレーション内の細かい並列性を無視し大きなまとまりとして各プロセスに割付けることを行い，そして各 DOALL ループで必要となるデータ分配と収集作業を削減できる箇所を探したことによるものである。このことは，より精度の高い並列性解析や実行時間解析と通信時間の算出を元にしたタスクスケジューリング・粒度解析によるところが大きい。並列コード生成に関してもタスクブロック間通信を最適化するような新しい方法が必要と考えられる。

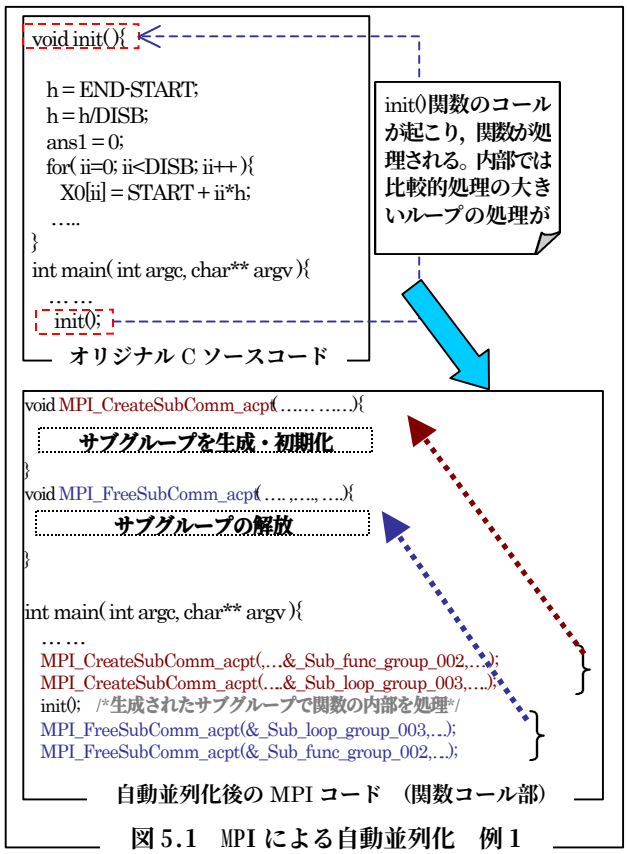


図 5.1 MPI による自動並列化 例 1

## 6．最後に

### 6．1 まとめ

本年度は並列性解析から実行時間解析，タスク粒度解析，タスクスケジューリングの一連の流れを繋ぐ共有リソースの作成と利用を通して最終的な並列コード生成までを全体を通しておこなってきた。まだまだ考えるべき点はあるものの，より実システムとして形のあるものが残せる結果になったと考えられる。特に各解析間を結ぶ共有リソースである二つのインタフェースを通すことで各解析間を結ぶことが出来るようになった，という流れは今後の自動並列化 C コンパイラのために役立つものだと考えられる。そして全自動とはいえないながらも，ス

ケジューリングからの結果を手動で生成した物を用いることで自動並列化トランスレータとして MPI を用いた自動並列コードを生成することが出来るようになった。結果としては，実行時間が逐次実効に比べて遅くなっているが，今後の方針となる明確な考案が出来たことも今後に繋がるものだと考えている。

### 6．2　今後の課題

**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。

**通信時のデッドロックの考慮とデータの通信の効率化**

本年度の通信には，ブロッキング命令である MPI_Send 命令を用いて実装している。ブロッキング命令を用いている限り，システム側に依存する傾向が高まりデッドロックを起こす可能性がある。このことへの対応とデータの通信時に変数ごとの Send/Recv を行うのではなくデータを一まとめにした Pack 通信を行うことによる効率化を考えていくべきである。また，DOALL ループなどに必要となっているデータ分配や収集時にかかる通信コストの削減方法の考案も考えていくべきである。

**ユーザの知識を利用する方法の検討** [3)]

プログラムの並列性解析には動的情報が必要となる場合があるため，静的情報のみに頼る方法に固執する必要はないと考えている。そこで，ユーザの知識を反映させるような指示文の導入も検討する必要があるだろう。しかし，より多くのユーザに利用されることを想定しているため，自動並列化の実現も念頭に入れることを忘れてはならない。

**精度の高い通信時間や実行時間を利用することによる最適タスク粒度とスケジューリングの考案**

細かい並列性を無視することにより最適なスケジューリングを求めることが出来る場合がある。その逆もあると考えられる。より正確な通信時間と実行時間の値を参考とし，そして動作環境に最も適した最適なスケジューリングや粒度の決定を行うべきである。

## 参考文献


1) TOP500　http://www.top500.org/
2) 「アドバンスト並列化コンパイラ技術」産業科学技術研究開発基本計画 http://www.nedo.go.jp/informations/koubo/120324_4/ad-kihon.html
3) 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
4) 三木光範. “並列処理入門”. PC クラスタ超入門 講習会資料. pp166-177，超並列計算研究会. 2000.
5) John L. Hennessy and David A.Patterson. “Computer Architecture A Quantitative Approach, Second Edition”. Morgan Kaufmann Publishers, Inc. 1996.
6) 湯淺太一，安村通晃，中田登志之. ”はじめての並列プログラミング”. 共立出版. 1999.
7) Kai Hwang. “Advanced Computer Architecture: Parallelism, Scalability, Programmability”. McGraw-Hill, Inc. 1993.
8) 手代木進. “自動並列化 C コンパイラのための並列性解析”. Master’s thesis, 成蹊大学工学部工学研究科情報処理専攻. 2002.
9) 斉藤 義功. “ 動的メモリ確保を含む C プログラムの自動並列性解析”. Master’s thesis, 成蹊大学工学部工学研究科情報処理専攻. 2003.
10) Message Passing Interface Forum -MPI2- http://www.mpi-forum.org/
11) The Message Passing Interface (MPI) standard http://www-unix.mcs.anl.gov/mpi/
12) 林晴比古. “新 C 言語入門 シニア編”. ソフトバンク. 1991.
13) Mark Williams Company. “ANIS C a lexical guide”. 1990