成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.47 No.1 (2010) pp.49-57

C 言語自動並列化トランスレータの開発
― ポインタ／配列の依存解析に基づくタスク粒度の決定手法 ―

高野　裕太*1，甲斐　宗徳*2

Development of Automatic Translator from C Programs to Parallel Programs Using MPI
- Decision Support of Task Granularity based on Pointer/Array Data-dependent Analysis -

Hirota TAKANO*1, Munenori KAI*2

ABSTRACT:
The authors have been developing an automatically parallelizing translator for serial C program into parallel MPI program. The translator consists of a series of some analysis stages such as parallelism analysis stage, task granularity analysis stage, task scheduling stage, parallelized code generation stage. At the parallelism analysis, statement level parallelism is extracted, and a lot of fine-grained tasks such as statement level tasks are found. A lot of tasks make the task scheduling stage very difficult, because this task scheduling problem belongs to the class of strong NP complete combinatorial optimization problem. In this paper, in order to reduce the number of tasks and simplify the topology and the structure of the task graph, we propose a task fusion scheme and generating macro tasks to reduce the number of tasks and the number of communication edges among task graph. It is shown that the overall task scheduling is divided into small sub-task scheduling problems which can be solved easier.

Keywords: automatic parallelization, task scheduling, task fusion, macro task, task granularity

(Received March 25, 2010)

1. はじめに

1. 1 背景と目的

マルチプロセッサシステムで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がある。これを解決するためには並列性の抽出作業，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い，効率的な並列プログラムを作成しなければならない。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 言語に対する自動並列化トランスレータの開発を行っている。ポインタの依存解析やその解析に基づく配列アクセスの動的な依存解析を行い，その結果として得られたタスクグラフを元に粗粒度および細粒度並列化を行うことで，実行性能の向上を図った並列プログラムに変換する。

1. 2　タスク粒度の決定

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ッサ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[3]。最適な並列化を行うためにはすべてのタスクを最小の粒度としてスケジューリングを行うべきである。しかし，スケジューリングに要する時間はタスク数に対して指数関数的に増大するため，そのようなスケジューリングは現実的でない。また，細粒度のタスクの場合，通信によるオーバヘッドを考慮すると，並列性がある場合でも異なるプロセッサに割り当てるよりも同じプロセッサに割り当てたほうが実行時間の

*1：工学研究科情報処理専攻修士学生
*2：工学研究科情報処理専攻教授(kai@st.seikei.ac.jp)

短縮になる，というスケジューリング結果が出る場合も多い。そのような場合には，同じプロセッサに割り当てたほうがよい，またはその可能性が高いとあらかじめ解析できるタスクに関してはタスクの融合を行い，一つのタスクとして扱ったほうがスケジューリングに要する時間を削減できる。今年度の研究では，このようなタスクに関して依存解析に基づいてタスクの融合を行い，タスク粒度の決定を行った。

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プログラム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の流れを図 2.1 に示す。

初期作業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。その時点のソースコードでは並列処理不可能と判定されたループが，ループリストラクチャリングによって並列処理可能なループに変換可能と判断されるとソースコードの変換作業を行い，変換後の新たなソースコードに対して再び並列性解析を行う[1][2]。

中盤の作業としてタスク粒度解析，部分タスクスケジューリングを行う。タスク粒度解析は，前段の解析の結果に基づき，各タスクをどのような粒度でプロセッサに割り当てれば効率の良い並列処理が行えるかを考慮しタスク粒度を決定する。マクロタスクなどの粗粒度タスクに対して，通信時間を考慮した部分タスクスケジューリングを適用することで，どのような順序で実行すれば最短時間でその粗粒度タスクの処理が完了するかを解析する。この二つの処理はよりよい結果を求めるために繰り返し行われる。

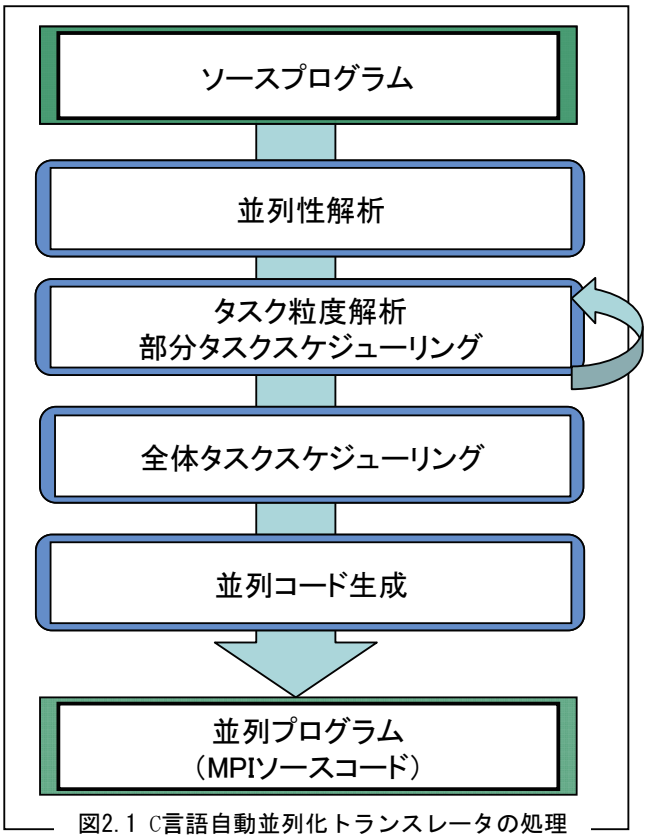

図2.1 C言語自動並列化トランスレータの処理

次に全体タスクスケジューリングを行う。ここではタスク粒度解析，部分タスクスケジューリングを繰り返した後の最終的な粒度のタスク集合をプロセッサに割り当てるスケジューリングを行う。

最終段階で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プログラムの自動生成を行う。

３．タスクグラフ

C 言語自動並列化トランスレータ内では，タスクと呼ばれる単位でプログラム内の依存関係を表現する。タス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なる。逐次処理ではプロセッサ 1 台で処理を行うので通信を行う必要はないが並列処理の場合，プロセッサを複数台使用するため通信が発生する。従ってタスクの処理時間と通信時間のトレードオフを考慮しなければならない。細粒度タスクでは逐次処理を行った方が効率の良い場合があり，すべての並列性を活かせば良いとは限らない。

また，タスクスケジューリングは一般に強 NP 困難な最適化問題である。そのため，タスク数が多い場合には最適解を求めるのに非常に多くの時間を要する。従って，C 言語自動並列化トランスレータでは通信のデータ量を減らすように粗粒度タスクの生成を行うことでタスク数を削減している。具体的には，タスク間で依存している変数のペアの数を依存強度と定義し，依存強度の高い細粒度タスク同士を融合している。しかし，細粒度タスクで扱うデータは比較的サイズが小さく，通信に要する時間は通信量に依存しない場合が多い。ボトルネックになるのはむしろ通信のレイテンシ，つまり同期による待ち時間である。このような通信が複数ある場合，それらをパッキングし，一回の通信で必要なデータをまとめて送信することによって通信時間を低減できる。そのため本研究では通信の量ではなく回数を減らすことを念頭に置いた新たなタスクの生成方法を考案した。

3. 1　タスクの種類

タスクの種類は以下の 4 種類が存在する。

① SELECT(IF, ELSE_IF, ELSE)タスク

if else，switch といった制御文内のボディを含むタスクである。SELECT タスクは必ずマクロタスクとなり，サブタスクとして 1 つの IF タスクと任意の数の ELSE タスク，ELSE_IF タスクが存在する。マクロタスクについては 3.2 で述べる。

② LOOP タスク

for 文や while 文などループ文全体を包含するタスクとして定義される。SELECT タスクと同様にマクロタスクとなる。

③ FUNC タスク

ユーザ定義関数を呼び出すステートメントとして定義される。

④ BASIC タスク

上記のタスクの種類に当てはまらないタスクの種類として定義される。最小単位はソースコードのステートメントとしている。マクロタスクである場合とそうでない場合のどちらも存在する。

3. 2　マクロタスク

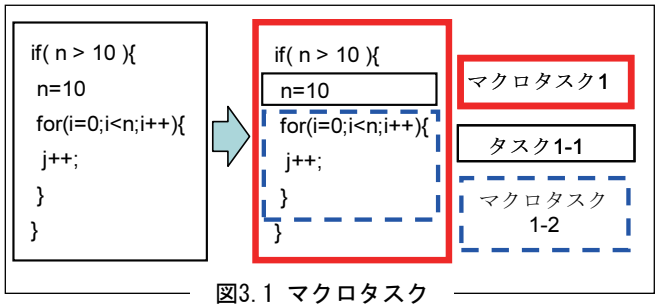

図3. 1 マクロタスク

図 3.1 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を階層的に含むようなタスクがある。このようなタスクを「マクロタスク」と呼ぶ[4]。

マクロタスクは，スケジューリング時にそのタスクの内部構造を隠蔽するために用いられる。マクロタスクの外部のタスクとマクロタスク内のサブタスクに依存関係がある場合，外部タスクとサブタスクが直接依存関係をもつのではなく，外部タスクとマクロタスクが依存関係をもつものとして扱う。そのため外部タスクと複数のサブタスクに依存関係がある場合には，複数のエッジが外部タスクとマクロタスク間の一本のエッジにまとめられることになる(図 3.2)。

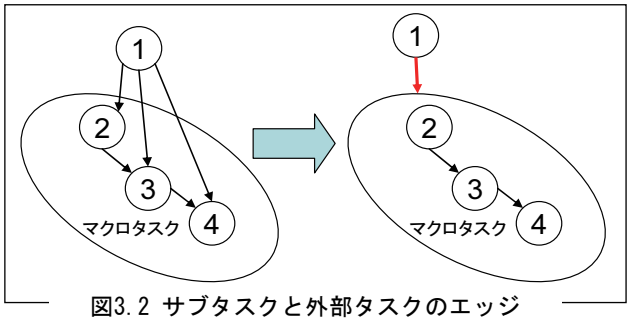

図3. 2 サブタスクと外部タスクのエッジ

タスクの融合を行う際，融合されるタスクにマクロタスクが含まれる場合にはマクロタスクの融合は行わない。融合後のタスクは融合されるタスクをサブタスクとして持つマクロタスクとなる。この場合，マクロタスクが 1 つ増えるため，全体としてのタスク数は融合を行う前と比較して増加してしまう。しかし，融合が行われた階層のタスクグラフとしてはタスク数が減少すること，マクロタスク化によって依存のエッジが減少することから，スケジューリングに要する時間は減少することが見込める。

3. 3　タスク間の依存の種類

タスクグラフには以下の 4 つの依存関係が存在する。

① フロー依存

書き込みの後に読み込みを行う。

② 逆依存

読み込みの後に書き込みを行う。

③ 出力依存

書き込みの後に書き込みを行う

④ 制御依存

分岐命令と分岐後に処理されるタスクに関する依存

このうち，①，②，③はデータに関する依存関係，④は制御に関する依存関係である。データに関する依存関係のうち，フロー依存は真の依存とも呼ばれており，この依存が存在するタスク間を別プロセッサに割り当てる場合には，必ず通信が必要となる。一方，逆依存，出力依存に関しては，別プロセッサに割り当てた場合に通信を発生させるものではなく，同一プロセッサ上にタスクを割り当てた場合の先行関係を示すものである。そのため，タスクの融合を行う際には逆依存，出力依存は依存ではないものとして扱い，以降の図でも出力依存，逆依存のエッジは表記しない。ただし，復元時にタスクグラフに循環エッジが発生するのを防ぐため，タスクの融合時に各タスクの逆依存，出力依存の情報の参照を行っている(図 3.3)。

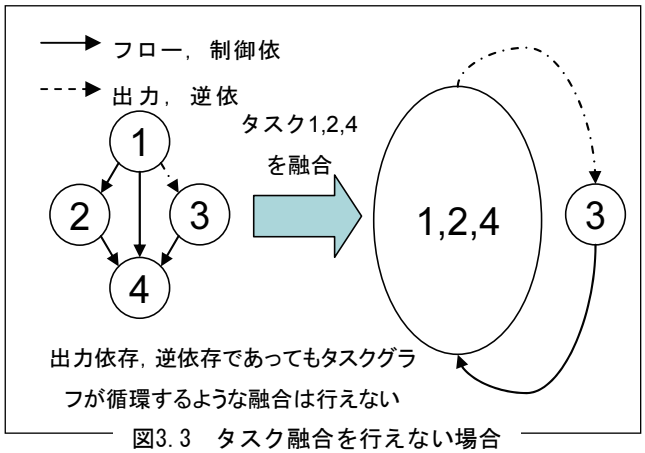

図3.3　タスク融合を行えない場合

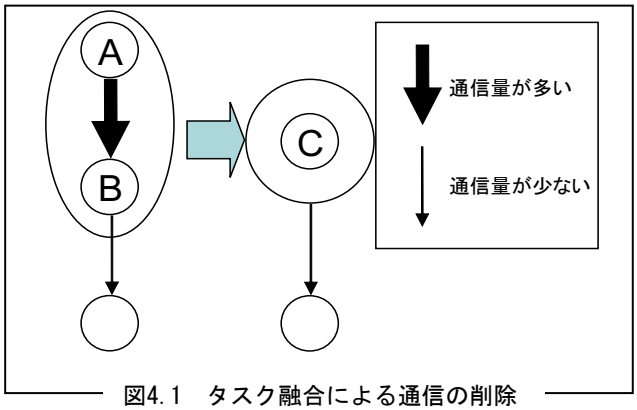

図4.1　タスク融合による通信の削除

4．タスク粒度の決定手法

タスク粒度解析では，複数のタスクを一つにまとめあげている。変数間の依存関係，依存の強さからスケジューリングをする必要が無い，あるいはその可能性が高い部分を全体スケジューリングに先立って見つけておき，それらをまとめ上げることで，全体スケジューリングの負担を軽減することができる。

これまでのC言語自動並列化トランスレータでは時間的局所性，タスク間の通信量を考慮してタスク融合を行っていた[5][6]。しかし細粒度タスクでは通信量が少ない場合が多く，この方法によるタスク融合だけでは融合できるタスクはごく少数だけであった。今年度の研究ではそれに加えて，タスク間の通信を一つにまとめ，タスクグラフのエッジを削減する新たなタスクグラフの決定手法を考案した。これまでの手法を 4.1 に，新たに考案した手法を 4.2 で述べる。

4．1　通信削除のためのタスク融合

先行関係にあるタスクが処理を終えると，後続タスクは処理を行うために必要なデータを通信によって受け取る必要がある。従って後続タスクが処理を行う上で必要なデータが配列のような大きなデータ量を持つものであると，通信時間が膨大にかかってしまう。この問題を避けるために，通信量の多いタスク間を融合し，通信を除去してしまうという方法を利用した。概念としては図 4.1 のようになる。具体的には，タスク間で依存している変数のペアの数を依存強度と呼び，依存強度が指定した閾値を超えた場合に融合を行う。依存強度の重みは変数一つのペアは 1 とし，配列の場合はその配列の要素数とする。

図 4.2(a)のようなコードセグメントの場合，各タスクの依存強度は以下のようになる。

タスクA：タスクグラフ上のどのタスクにも依存していない
　→依存強度 0

タスクB：タスク A の変数 a に依存している
　→依存強度 1

タスクC：タスク A の変数 a に依存している。タスク A の要素数 10 の配列 b に依存がある
　→依存強度 1+10＝11

タスクD：タスク B の変数 c に依存している。タスク C の変数 d に依存している。
　→依存強度 1+1＝2

閾値を超える候補が複数あった場合には，

①　依存強度が大きい
②　融合後のタスクの先行タスク数が最も少ない
③　融合後のタスクの依存強度が最も低い
④　融合後のタスクのステートメント数が最も多い

の順で検討し，最終的にタスク融合を行うタスク集合を

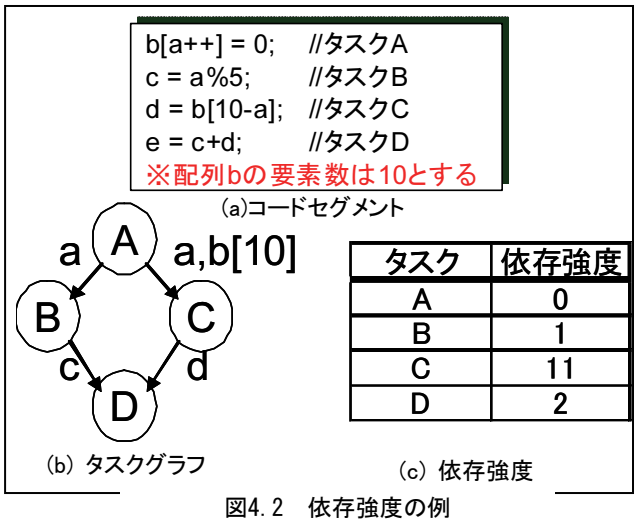

図4.2　依存強度の例

決定する。全てのタスクが融合されてしまう，すなわち逐次処理に戻ってしまうことを避けるために，タスク融合後のタスク粒度の上限を決めておくことが必要である。本研究では上限を設定するためにタスクの持つステートメント数を用いている。なお，LOOP タスクは他のタスクと融合を行わなくてもある程度大きな粒度を持つと仮定し，LOOP タスクを含むタスク融合は行わない。

前年度までの研究ではこの方法を用いてタスクの融合を行っていた。しかし，ステートメント単位という細粒度のタスクでは依存強度が小さい場合が多いこと等から，タスク数の減少はごくわずかなものとなっていた。

4. 2 通信統合のためのタスク融合

より多くのタスクを融合し，スケジューリングの時間を削減するために，今年度の研究では通信量ではなく，通信回数，つまりタスクグラフのエッジに着目して新たなタスクの融合手法を考案した。3 章で述べたように，細粒度タスクで扱うデータはパッキングによって通信をまとめあげることでボトルネックとなる通信のレイテンシ，同期による待ち時間を削減できる場合が多い。そこで，共通のタスクへの通信が存在するタスクに着目した。その中でも並列性を極力損なわないパターンとして，直接後続タスクに対し，他の到達経路がある場合をタスク融合の対象とした。図 4.3 のタスク 1，タスク 2，タスク 3 は BASIC タスクとして一つにまとめられる。このタスクグラフの場合 1 から 4 に到達する経路は 1→4，1→2→3→4 の二通り存在する。このような場合，1，2，3 を一つのタスクに融合し，1→2，2→3 という 2 つの通信を不要とし，4 への 2 つの通信を一つの通信にまとめることができる。このような形状のタスクグラフの場合，タスクを融合しない場合でもタスクの実行順序は 1,2,3,4 の順以外にすることはできない。つまり融合しても並列性が損なわれないため，タスク粒度の上限を設けることなくタスクの融合を行うことができる。

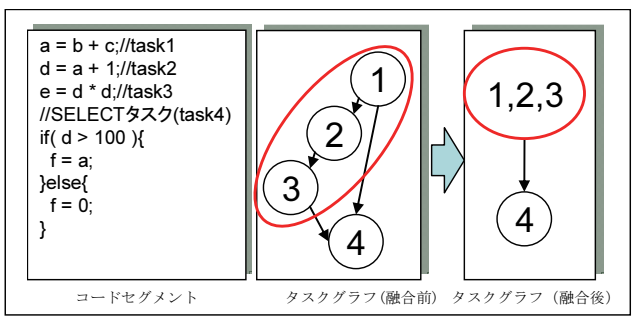

図4. 3 共通のタスクへの通信があるタスクの融合

図 4.3 の融合を行うことによって，タスクの実行順序はタスク 1，2，3 の融合タスク，タスク 4 の順序であることが確定する。また，タスク 1→2 間，タスク 2→3 間の通信についてはスケジューリングで考慮する必要が無くなり，タスク 1→4 間，タスク 3→4 間の通信は一つの通信となるため，それぞれを別個に処理するよりも負担は軽減される。これらの結果，全体スケジューリングに要する時間の削減が見込める。

実際にタスクグラフ上でこの手法による融合を行う際には，マクロタスクの有無，融合されるタスクとそれ以外のタスクの依存関係を考慮する必要がある。また，タスク 1,2,3 が融合可能であり，同時にタスク 1,4,5 が融合可能である，といった場合には，タスク 1 が競合しているためどちらか一方の融合しか行うことができない。これらの扱いについては次節で説明する。

4. 2. 1 マクロタスクの融合

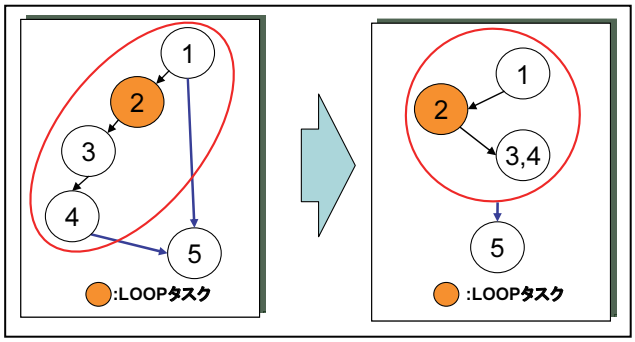

図4. 4 マクロタスク生成を伴うタスク融合

これまでのタスク融合の手法においては，LOOP タスク等マクロタスクはある程度の粒度の大きいタスクとして，融合を行わないものとしていた。これは，他のタスクと融合してしまうことで並列化可能なループを逐次処理を行う一つのタスクに置き換えてしまうことを防ぐためである。

通信統合のためのタスク融合では，マクロタスクを含むタスクを融合する場合，マクロタスクを挟まないタスクのみを一つのタスクへと融合し，全体としてはタスクではなく新たなマクロタスクへと融合することで，マクロタスクを融合可能なタスクとして取り扱えるようにした。通信削除のタスク融合の場合はマクロタスク内に通信が残存してしまうことになるので，同じようにマクロタスクとして融合することはできない。マクロタスクにすることでタスク数そのものは増加してしまう可能性があるが，タスク融合によりエッジが減少するためスケジューリング時間の減少が見込める。

4. 2. 2 タスク融合候補と外部の依存関係

これまでの例では融合するタスク間の依存のみがあるタスクグラフを記載していた。実際のタスクグラフ上では多くの場合，融合するタスクとその他のタスク間に依

存関係がある。そのような例を図 4.5 に示す。

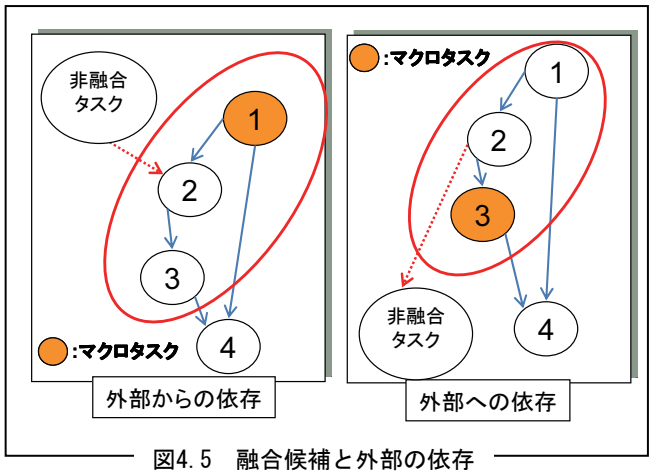

図4. 5　融合候補と外部の依存

図 4. 5 の左は融合候補のタスク 2 が外部のタスクに依存しているため，融合を行うとタスク 1 は外部タスクと並列に実行できなくなる。同様に右は外部タスクが融合候補のタスク 2 に依存しており，融合を行うとタスク 3 と外部タスクを並列実行できなくなる。特に並列実行できなくなるタスクが LOOP などのマクロタスクである場合には融合による影響が大きいと考えられるので，融合を行わない。

4. 2. 3 融合候補の選択

4.2.2 で除外されたものを除いた融合候補について，以下のような優先順位でタスクの融合を行う。

① マクロタスクを含まない候補を融合する
② 融合するタスク間での依存強度の高いものを優先する
③ タスク数の少ないものを優先する
④ マクロタスクを含む候補を融合する
⑤ より多くのエッジを削減できる候補を優先する
⑥ タスク数の少ない候補を優先する

マクロタスクを含まない候補の場合，タスクの融合によってエッジの削減だけでなく不要な通信を除去できる。そのため，マクロタスクを含む候補よりも先に融合を行い，特に依存強度の高いタスクを優先的に融合する。同じタスクを融合する融合候補は複数ある可能性があり，その場合融合可能な候補はそのうちのいずれか一つだけとなる。このようなタスクの競合が起きることを最小限に抑えるため，マクロタスクを含む場合，含まない場合ともに，他の条件が同じならタスク数の少ない候補を融合する。

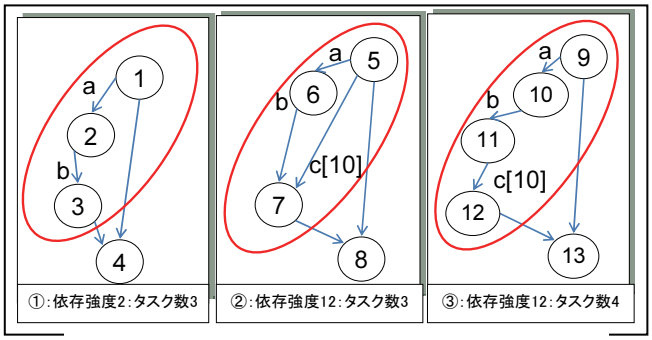

図4. 6　マクロタスクを含まない候補の融合

図 4.6 の場合，依存強度が最も高いのは②と③，そのうちでタスク数が少ないのは②なので，融合の優先順位は高い方から②，③，①の順番となる。

マクロタスクを含む候補の場合，融合するタスク間の通信のうちマクロタスクを挟むものはそのまま残るため，その部分の通信は削除されない。かわりに候補が融合された場合に外部のタスクとの通信の回数を最小限に抑えるため，タスクグラフのエッジを多く削減できる候補を優先する。

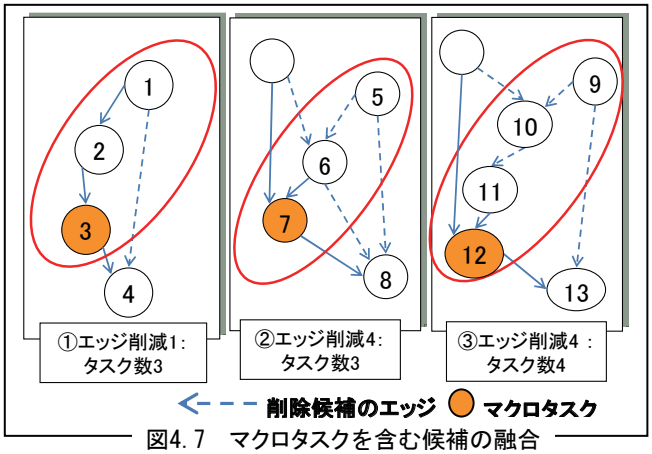

図4. 7　マクロタスクを含む候補の融合

図 4.7 の場合，エッジ削減が最も多いのは②と③である。そのうちタスク数が少ないのは②なので，優先順位は図 4.6 と同様に高いほうから②，③，①の順番となる。②を融合した結果が図 4.8 である。一つのタスク融合が行われた後，残っている候補，及び融合によって新たに発生した候補がある場合にはその候補を含めた融合候補の中から同様に優先順位に基づいて融合候補が無くなるまで融合を繰り返す。たとえば図 4.6 の場合には，②の融合によって①，③のタスクが融合不可能な形に変わっていない限り③，①の順番で融合が続けて行われる。図 4.7 に関しても同様である。

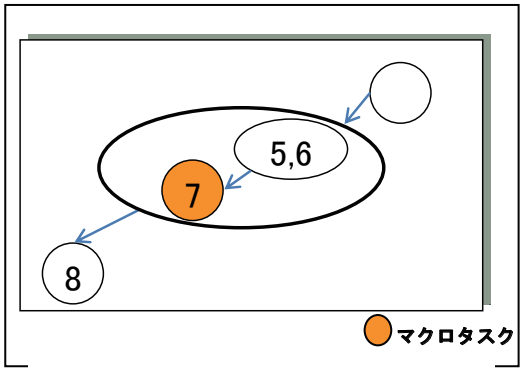

図4.8 マクロタスクを含む候補の融合

5．評　価

今回ベンチマークプログラムとして NPB(NAS Parallel Benchmarks)のプログラム IS(Integer Sort)を採用した[8]。現状では関数間解析の実装が十分に行われておらず，ユーザ定義の関数コールの部分が解析できないため，関数呼び出し側のコード中にユーザ定義関数を展開した。表 5.1，表 5.2 はタスク融合前後のタスクグラフの状態である。

表5.1 メインタスクグラフのタスク数，エッジ数の変化

	融合前	融合後
タスク数	44	40
エッジ数	54	21
マクロタスク数	14	12
融合によって作られたマクロタスク	0	3

表5.2 最大のタスク数を持つマクロタスクの
タスク数，エッジ数の変化

	融合前	融合後
タスク数	76	54
エッジ数	401	198
マクロタスク数	4	4
融合によって作られたマクロタスク	0	0

表 5.1 は解析を行うタスクグラフそのものであり，表 5.2 は表 5.1 のタスクグラフに含まれるタスクグラフのうち，タスク数が最も多いものである。それぞれについてその他のマクロタスクについては記載していないが，これらはタスク数 1～14，エッジ数 0～35 と表 5.1，表 5.2 と比較して少ない値に収まっており，スケジューリング全体に大きな影響を与えないと考えられる。

どちらの場合もエッジ数，タスク数ともに減少し，特にエッジ数に関しては半分以下となった。タスクスケジューリングに要する時間はタスク数とエッジ数の増加にともない指数関数的に増大するので，最も多くのタスク数，エッジ数をもつマクロタスクに対してタスク数，エッジ数を削減できたことはスケジューリング時間の短縮に大きな効果があると考えられる。

図 5.1，図 5.2 はソースコード上でどのタスクが融合されたかを示している。

それぞれの図中の枠で囲まれたステートメントが融合されて 1 つのタスクとなった部分である。図 5.1 の場合，上から 5 つのステートメント，その下の 3 つのステートメントが融合されてそれぞれ一つのタスクとなっている。

```
randlc_T1 = randlc_R23 * create_seq_seed;
randlc_j  = randlc_T1;
randlc_X1 = randlc_j;

randlc_X2 = create_seq_seed - randlc_T23 * randlc_X1;
randlc_T1 = randlc_A1 * randlc_X2 + randlc_A2 * randlc_X1;
randlc_j  = randlc_R23 * randlc_T1;
randlc_T2 = randlc_j;
randlc_Z = randlc_T1 - randlc_T23 * randlc_T2;

randlc_T3 = randlc_T23 * randlc_Z + randlc_A2 * randlc_X2;
randlc_j  = randlc_R46 * randlc_T3;
randlc_T4 = randlc_j;

create_seq_seed = randlc_T3 - randlc_T46 * randlc_T4;
```

図5.1 マクロタスク化を伴わないタスク融合

```
for( rank_i=0; rank_i<MAX_KEY-1; rank_i++ )
        rank_key_buff_ptr[rank_i+1] += rank_key_buff_ptr[rank_i];
for( rank_i=0; rank_i<TEST_ARRAY_SIZE; rank_i++ ){
        rank_k = partial_verify_vals[rank_i];       /* test vals were put here */
        if( 0 <= rank_k  &&  rank_k <= NUM_KEYS-1 )
         if( rank_i <= 2 )
         {
          if( rank_key_buff_ptr[rank_k-1] != test_rank_array[rank_i]+rank_iteration )
          {
           printf( "Failed partial verification: iteration %d, test key %d¥n",
             rank_iteration, rank_i );
          }
          else
           passed_verification++;
         }
         else
         {
          if( rank_key_buff_ptr[rank_k-1] != test_rank_array[rank_i]-rank_iteration )
          {
           printf( "Failed partial verification: iteration %d, test key %d¥n",
             rank_iteration, rank_i );
          }
          else
           passed_verification++;
         }
}

if( rank_iteration == MAX_ITERATIONS )
        key_buff_ptr_global = rank_key_buff_ptr;
```

図5.2 マクロタスク化を伴うタスク融合

図 5.1 の場合，マクロタスク化は行われず，通信は削除されるため，ステートメント 1 つを細粒度タスクとし，上部のステートメントから順にタスク 1，タスク 2…タスク 12 とすると，タスク数の変化は図 5.3 のようになる。

なお，図の外部の変数との依存関係も存在するが，ここでは省略する。この場合，タスク数は 6 個，エッジは 10 本減少する。図 5.3 の場合，2 つの for 文，つまり 2

つの LOOP タスクが融合され，1つのマクロタスクとなる。通信の削除は行われないため，図中でのエッジ数の削減は図 5.4 のように 1 本のみとなる。

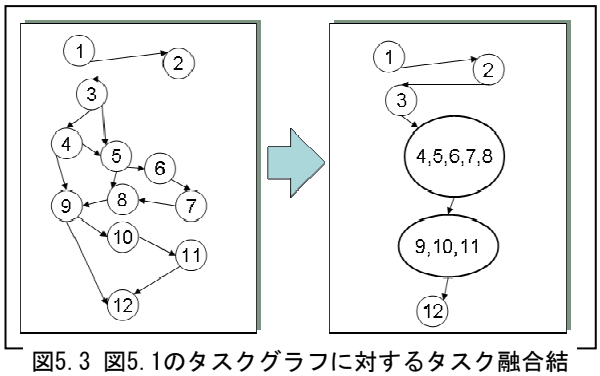

図5.3 図5.1のタスクグラフに対するタスク融合結

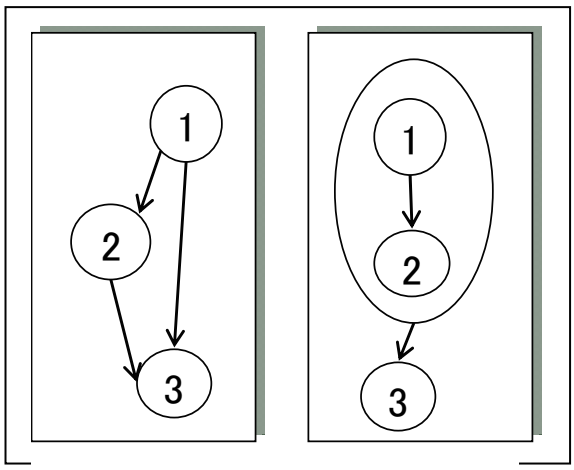

図5.4　図5.2のタスクグラフに対するタスク融合

６．終わりに

6．1　まとめ

今年度の研究は通信量，通信のレイテンシを考慮し，並列化しないほうがよい，またはその可能性が高いタスクを融合し，一つのタスク又はマクロタスクにまとめることによって不要な通信の削除，通信の統合を行い，タスクグラフのタスク数，エッジ数を減少させることで強 NP 困難な最適化問題であるタスクスケジューリングの処理時間の削減を図るものである。新たなタスクの融合手法によって，より多くのタスクの融合が可能となった。また，タスクの融合によってマクロタスクを生成することで，ループ等のマクロタスクもタスク融合の対象とし，より多くのエッジを削減できるようになった。これによってタスクスケジューリングに要する時間を大きく短縮できると考えられる。

6．2 今後の課題

今後の C 言語自動並列化トランスレータにおける課題について，以下のようなものが考えられる。

- 関数間での依存解析
  現状では関数間の依存解析が十分に行われていない。そのため，ポインタ情報を関数に map しての解析や再帰への対応が必要となる。
- より多くの C 言語の構造への対応
  上記の関数間の依存解析だけでなく，C 言語特有の自由度の非常に高いポインタの利用方法についても考えなければならない。たとえば関数ポインタの場合など，対応できていない C 言語の構造も数多く存在するため，これらへの対応が必要となる。
- 部分タスクスケジューリング
  DO-ALL 可能な LOOP タスクなど，複数のプロセッサで並列処理可能なタスクが存在する。全体タスクスケジューリングの前段階として，どのタスクにいくつまでプロセッサを割り振って良いか等を決定する，部分タスクスケジューリングが必要となる。

謝　辞

本研究の一部は，文部科学省戦略的研究基盤形成支援事業の補助を受けて行ったことをここに記し，謝意を表します。

参考文献

[1] 手代木 進：“自動並列化 C コンパイラのための並列性解析”，修士論文，成蹊大学工学部工学研究科情報処理専攻.2002.

[2] 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”，修士論文，成蹊大学工学部工学研究科情報処理専攻.2003.

[3] 美濃本 一浩：“C 言語自動並列化トランスレータの開発”，修士論文，成蹊大学工学部工学研究科情報処理専攻.2005.

[4] 田中康之：“C 言語自動並列化トランスレータの開発～マクロタスクの取り扱いに伴う拡張”，修士論文，成蹊大学工学部工学研究科情報処理専攻. 2007.

[5] 尾高　輝: “通信遅延を考慮したタスクスケジューリングのためのタスク粒度解析”，修士論文，成蹊大学工学部工学研究科情報処理専攻.2007.

[6] 三浦　純: “C 言語自動並列化トランスレータの開発―ポインタ／配列依存解析の改良とタスク粒度の決定―”, 修士論文, 成蹊大学工学部工学研究科情報処理専攻.2008.

[7] Alfred V.Aho, and Monica S.Lam, and Ravi Sethi, and Jeffrey D.Ullman : “Compilers principles, Techniques, & Tools Second Edition”, Addison Wesley, 2007

[8] NAS Parallel Benchmarks Changes http://www.nas.nasa.gov/Resources/Software/npb.html 2010/03/24 現在, 参照可能