アルゴリズム 71－3
（2000．1．17）

タスクスケジューリングを用いた並列プログラム生成における タスク粒度の調整とその評価

橋本貴至　森雅博　西村晃一
藤本典幸　萩原兼一

大阪大学 大学院基礎工学研究科 情報数理系専攻

本論文では，静的スケジューリングを用いた並列プログラム生成においてのタスクグラフの粒度の調整とその影響について述べる．これまで我々は通信の一括化を考慮したスケジューリングアルゴリズム BCSH を開発し，細粒度タスクグラフから並列プログラムを生成する研究を行ってきた．BCSH では，通信のオーバヘッドが大きい細粒度タスクグラフにおいても通信の一括化を利用することで性能のよいプログラムを生成することができる．しかし，1 代入文を 1 タスクとする細粒度タスクグラフはタスク数がプログラムの計算量に比例して増加するためサイズの大きな問題を扱うことが難しい．そこで，1 タスクを基本ブロックやループへと拡大することにより，タスク数を削減する．BCSH を用いた場合，タスクの粒度を上げると性能低下が予想される．そこで，粒度を上げた際のタスクグラフの形状の変化が生成される並列プログラムの性能に与える影響について，ガウスジョルダン法と FFT を用いた適用実験によって評価を行った．結果として，各タスクの出次数が少なく，割り当てるプロセッサ数に対して十分な並列度を持ったタスクグラフを用いることが，並列プログラムの性能にとって重要であることがわかった．

Task Granularity Adjustment to Generate a Parallel Program with Task Scheduling

Takashi Hashimoto, Masahiro Mori, Kouichi Nishimura,
Noriyuki Fujimoto and Kenichi Hagihara

Department of Informatics and Mathematical Science,
Graduate School of Engineering Science, Osaka University

In this paper, we describe task granularity adjustment to generate a parallel program with the task scheduling algorithm BCSH. BCSH is an algorithm with consideration for message aggregation. Therefore a fine grained task graph is suited for BCSH. However, it is difficult for BCSH to deal with fine grained task graphs for large size program, because BCSH requires $O(N^4)$ time where N is the number of nodes in a task graph. So, we need to generate coarser grained task graph to deal with large size program. However, in BCSH, coarsing task granularity has an influence on performance of generated parallel programs. We present experiments on various grained task graphs for Gauss-Jordan and FFT, and evaluate the adjustment. The result is that the task graph which has small outdegree and much more parallelism relative to the number of available processors is effective for BCSH.

1. はじめに

タスクスケジューリングを用いた並列プログラムの生成において，タスクグラフの形状はタスクの粒度の決め方によって変化する．プログラムをタスクグラフを用いて表す場合，タスクの単位を細かくするとプログラムの持つ小さな並列性まで表現できるが，タスクグラフのノード数は多くなる．一方，タスクの単位を粗くするとプログラムの細かい並列性は表せなくなるが，サイズの大きな問題についてもタスクグラフのノード数はそれほど多くならない．

これまで我々は，プログラム中の 1 代入文を 1 タスクとする細粒度タスクグラフにおいて，通信の一括化を考慮したスケジューリングアルゴリズム (BCSH[2]) を開発してきた．BCSH は，DSH[3] などのようにタスク単位でプロセッサへの割り当てを行うスケジューリングアルゴリズムでなく，実行対象の並列計算機の通信性能を考慮してタスクグラフのタスク複製を伴うク

ラスタリングを行い，クラスタ単位で割り当てを行うタイプのアルゴリズムである．クラスタリングによって，生成される並列プログラムの粒度を適度に粗くし，効率の良いスケジュールを生成する．よって，BCSHにおいてはプログラムの並列性を最大限に表した代入文レベルの細粒度タスクグラフを用いたほうが並列化の効果は得やすいと我々は考えている．

しかし，1代入文を1タスクとする細粒度タスクグラフでは，もとのプログラムで実行される代入文の数だけタスクが存在することになる．BCSHの計算量はタスク数Nについて最悪時で$O(N^4)$であり，一般的なタスクグラフの形状を考慮しても$O(N^3)$程度となる．よってタスク数の増加にともないスケジューリングに要する時間が著しく増大し，スケジュールを得ることが難しくなる．そのため，サイズの大きな問題についてはタスク粒度を粗くしタスク数を削減することが必要である．

本稿では，タスクの単位をプログラム中の1代入文から複数の代入文やループへ変更することで，タスクグラフの粒度を粗くしタスクグラフのタスク数を削減し，サイズの大きな問題についても並列プログラムの生成を行えるようにする．また，粒度を粗くした場合のタスクグラフの変化や，それがスケジュールに及ぼす影響について評価を行い，BCSHの入力としてどのようなタスクグラフを与えるのが適切かについて検討を行う．

2. タスクスケジューリング

2.1 タスクグラフ

タスクグラフとは，タスクを重みつきの節点，タスク間の先行制約を重みつきの有向辺として並列計算を閉路のない有向グラフで表したものである．

タスク間の先行制約は，タスク間にデータ依存が存在することによって生じる．タスク間のデータ依存にはフロー依存，逆依存，出力依存[8]の3つがあり，タスクAからタスクBの間に依存が存在するとは以下のような場合である．

フロー依存　タスクAである変数に対する書き込みが行われ，タスクBでその変数からの読み出しを行うとき

逆依存　タスクAである変数からの読み出しが行われ，タスクBでその変数への書き込みを行うとき

出力依存　タスクAである変数への書き込みが行われ，タスクBで再度その変数への書き込みを行うとき

タスク間にデータ依存が存在する場合，それをタスク間の有向辺で表す．以降ではタスクiからタスクjへの有向辺を(i,j)で表す．図1にタスクグラフの例を示す．図1の節点に添えてある数値は節点の重みを表し，有向辺に添えてある数値は有向辺の重みを表す．節点の重みは対応するタスクの処理時間を表す．有向辺(i,j)がフロー依存を表す場合，タスクjを処理するためにはタスクiの結果が必要であることを示し，その重みはタスクiとタスクjが異なるプロセッサに割り当てられた場合の，タスク間のデータ転送に要する時間を表す．タスクiとタスクjが同じプロセッサに割り当てられた場合の通信遅延は0とみなす．また，有向辺が逆依存と出力依存を表す場合，重みは0となる．

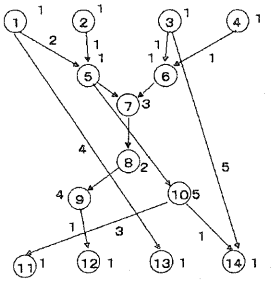

図1　タスクグラフの例

(i,j)が存在するときタスクiをタスクjの**直接先行タスク**，タスクjをタスクiの**直接後続タスク**という．各タスクの直接後続タスクの数をそのタスクの**出次数**，直接先行タスクの数をそのタスクの**入次数**という．また，出次数が0であるタスクを**終了タスク**といい，入次数が0であるタスクを**開始タスク**という．タスクiから終了タスクまでタスクの重みの和の最大値をタスクiの**レベル**といい，レベルlのタスクの数をレベルのlの**並列度**という．タスクグラフのタスクのレベルの最大値を**クリティカルパス長**(CP)といい，レベルの並列度の最大値を**最大並列度**という．

2.2 データフローグラフとデータ依存グラフ

タスク間のデータ依存のうち，フロー依存のみを先行制約としたものをデータフローグラフ，3つのデータ依存全てを先行制約としたものをデータ依存グラフと呼ぶ．

本研究ではタスクグラフとしてデータフローグラフを用いている．しかしコード生成を行うことを考えた場合，データフローグラフは逆依存と出力依存を考慮していないため，これを解決するために**変数リネーミング**[8]が必要になる．この問題については4節で詳

しく述べる．

3. BCSH

3.1 BCSH の概要

BCSH は最終的に図 2· 右図のような計算層と通信層が交互に現れるスケジュール (バルク同期型[7] のスケジュール) を生成する．計算層では各プロセッサは，他のプロセッサと独立してタスクの実行のみを行い，通信は行わない．一方，通信層では通信処理のみを行い，タスクの実行を行わない．BCSH はタスクグラフをもとにクラスタリングを行い (図 2· 左図)，図 2· 右図のような計算層と通信層が交互に表れるスケジュールを生成する．各計算層に含まれるクラスタは互いにフロー依存がないので，プロセッサ間通信は通信層のみで行う．クラスタリングは，対象となる実行環境の通信性能を考慮して行われるため，生成する並列プログラムの粒度は適度に調整される．

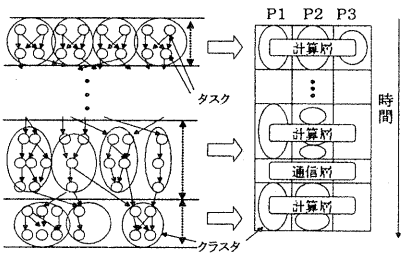

図 2 BCSH のスケジューリング方法

3.2 クラスタリングアルゴリズム

ここでは，紙面の都合上 BCSH のクラスタリングの流れについてのみ簡単に説明する．アルゴリズムの詳細については文献 2) を参照されたい．

あるレベルのクラスタを種として，レベル毎にクラスタへタスクを追加してく．このとき，タスクの複製を利用してクラスタは互いに依存が生じないようにする．その際，複製の数やプロセッサへ割り当てたときの負荷バランス等を考慮して，計算層の生成を中断し通信層を挿入する．

以下の流れを繰り返してクラスタリングを行う．

step1. クラスタリングを始めるレベル (最初は終了レベル) の各タスクについてそのタスクだけからなるクラスタを生成し，次のレベルに進む．

step2. 現在のレベルのタスクを，それぞれの直接後続タスクの存在するクラスタへ加える．この時，直接後続タスクの存在するクラスタが複数ある場合は，そのタスクを複製し各クラスタへ加える (図 3)．すなわち，複製を用いることでクラスタ間に依存が生じないようにクラスタリングを進めていく．

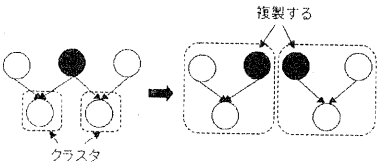

図 3 クラスタリング時の複製

step3. 次に，過剰な複製を削減するため，同一の複製タスクを多く持つクラスタ同士のマージを行う．マージを行うことによってクラスタ間のサイズにばらつきができると，後のプロセッサへの割り当て時にプロセッサ間の負荷のバランスが悪くなる．そのため，ばらつきが起きないようにマージを進めていく．

step4. 生成したクラスタにおいて，複製の数やプロセッサへ割り当てたときの負荷バランスについての判定を行い，次のレベルへ進むか，計算層の生成を終了し通信層を挿入するかを決める．

4. 並列プログラムの生成

タスクスケジューリングを用いた並列プログラム生成の流れは，タスクグラフの生成，スケジューリング，コード生成の 3 つに分かれる．まず初めにプログラムをもとに，タスクの単位を決定しデータフローグラフを生成する．これをタスクグラフ G とする．そして G について BCSH に基づきスケジュール S を生成する．最後にトランスレータを用いて S から通信ライブラリ MPI[4] を用いた C プログラムを生成する．

タスクグラフとしてデータ依存グラフではなくデータフローグラフを用いた場合，一般に出力依存と逆依存を解決するために変数リネーミングが必要になる．しかし，本研究では，以下の条件のいずれかを満たす逐次プログラムについては，変数リネーミングを行わずに逆依存と出力依存を解決している．

条件 1 任意の出力依存 (i, j) あるいは逆依存 (i, j) に対してがフロー依存 (i, j) が存在する．すなわち，出力依存と逆依存はフロー依存に重なる．

条件 2 任意のタスクのフロー依存に基づくレベルは，出力依存と逆依存についての直接後続タスクのフロー依存に基づくレベルより小さくはない．

条件 1 を満たす場合，データフローグラフを用いても逆依存と出力依存は解決されるので問題ない．BCSH

の生成するスケジュールは以下の性質を持つ.

- 同一レベルに存在するタスクは，全て同じ計算層に割り当てられる.
- ある計算層に存在する任意のタスクのレベルは，それ以降に実行される計算層に存在する任意のタスクのレベルより大きい.

よって得られたスケジュールの各計算層において，タスクをレベル順に並べ，各レベル内では逐次実行の順序に並べ替えることで，条件 2 を満たすタスクグラフの出力依存と逆依存を解決できる．また，この並び替えを行ってもフロー依存に反しないのは明らかである.

なお，6 節の適用実験で用いる逐次プログラムは，この条件を満たしている.

5. タスク粒度の調整

プログラムをタスクグラフで表す場合，タスクの粒度を細かくするとプログラムのもつ小さな並列性まで表現できる．1 代入文を 1 タスクとしたタスクグラフを**細粒度タスクグラフ**と呼ぶ．プログラムの並列性を代入文レベルで表現できるが，タスク数はプログラムで実行される代入文の数に等しく，問題の計算量に比例してタスク数は多くなる．BCSH の計算量はタスク数 N について最悪時 $O(N^4)$ である．そのため，現実的な時間でスケジューリングを行うためには，対象とする問題のサイズに応じてタスクの粒度を細粒度より粗くし，タスク数を適度な数に減らす必要がある.

ここでは粒度調整の方針と，それに伴うタスクグラフの形状の変化が並列プログラムの性能に与える影響について述べる.

5.1 基本ブロックに基づくタスク生成

基本ブロック [1] とはプログラム中の分岐や繰り返しのない連続部分である．基本ブロックにおいては最後の文以外では分岐を行わず，先頭の文以外は分岐先にならない．ここでは，基本ブロックに基づいたタスク粒度の調整について述べる.

基本ブロック内の並列性を考慮して，以下のように粒度を調整する．基本ブロックについて 1 代入文を 1 タスクとするデータ依存グラフを生成し，基本ブロックから得られたグラフに独立した部分グラフが存在するかを調べる．存在しなければ基本ブロック全体を 1 タスクとし，存在すれば各部分グラフを 1 タスクとする．また，それらの複数の独立した部分グラフのうちいくつかを合わせて 1 タスクとすることで，さらにタスク粒度を調整することができる.

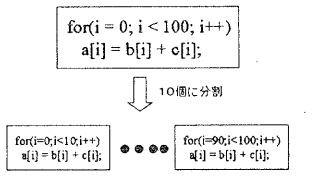

図 4 DOALL 型ループの粒度調整

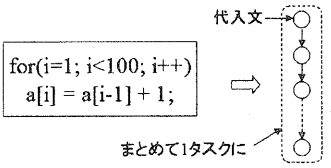

図 5 逐次性の強いループ

5.2 DOALL 型ループに基づくタスク生成

DOALL 型ループ [8] とは，ループの各イタレーション間にデータ依存のないループである．連続した複数回のイタレーションを 1 タスクにすることによって，タスクの粒度を調節することができる (図 4)．1 タスクに含まれるイタレーションの数を多くすると，タスクの粒度は粗くなり，ループの並列度は下がる.

また，以下のような場合はコードの変換を行い，DOALL 型ループを生成する.

- そのままでは並列性のないループでも，**ループ再構築** [8] を行うことで DOALL 型ループへ変換可能な場合.
- **ループコアレシング** [8] を適用できるネストした DOALL 型ループ.

(1 重化することで，粒度調整の自由度が上がる.)

このようなループの変換についての研究は盛んである．それらの結果を利用することで，効果的な粒度調整を行っていく.

また，DOALL 型に変換できない明らかに逐次性の強いループについてはループ全体を 1 タスクとする (図 5).

5.3 タスクグラフの形状とスケジュールの関係

タスク粒度の調整を行うと，タスクグラフの形状は変化する．タスクグラフの形状は並列プログラムの性能に大きな影響を与える．そこで，ここではタスクグラフの形状について考察を行う.

3.2 節で述べたように，BCSH のクラスタリングの過程において，出次数が 2 以上のタスクは直接後続タスクの属するクラスタ全てに複製される．そのため，出次数の大きいタスクは複製されやすい．また，複製が多くなるとそれを削減するためクラスタのマージを行うが，マージによってクラスタの大きさにばらつきが生じるとクラスタリングは中断され，通信層を挿入

する．そのため，出次数が大きいタスクが多く存在するタスクグラフの場合，スケジュールはタスクの複製や通信層の数が多くなると考えられる．

タスク粒度を粗くした場合，細粒度に比べてタスクグラフの並列度は低下し CP は大きくなる．しかし，スケジューリングによって最終的に利用する並列度は割り当てるプロセッサの数 P だけである．そのため，グラフの並列度が下がり CP が大きくなっても，並列度が P に比べて十分大きければ，生成する並列プログラムの性能への悪影響は少ないと考えられる．

よって，BCSH を用いて性能の良い並列プログラムを生成するためには，粒度調整を行う際は，以下のようなタスクグラフを生成すべきである．

- 各タスクの出次数が小さい．
- 利用可能なプロセッサ数に対してタスクグラフの並列度が十分大きい．

6. 適用実験

ここでは，実際のプログラムについて粒度の異なるタスクグラフを生成し，それをもとに並列プログラムを生成し評価を行う．並列プログラムの実行はワークステーションクラスタで行う．ワークステーションクラスタの構成は，ノードプロセッサが AT 互換機 (CPU: PentiumII 450MHz Memory: 512MB)，ネットワークが Myrinet[6] である．MPI ライブラリの実装は MPICH[5] で，通信性能は 1 対 1 通信で約 $40MB/$ 秒である．

また，スケジューリングの実行環境は AT 互換機 (CPU: PentiumII 450MHz Memory: 2GB) である．

生成した並列プログラムの性能評価の指標にはスピードアップ率を用いる．

$$\text{スピードアップ率} = \frac{\text{逐次プログラムの実行時間}}{\text{並列プログラムの実行時間}}$$

6.1 有向辺の重み

タスクグラフの有向辺の重みはタスク間の通信に要する時間を表している．BCSH を用いたスケジュールから並列プログラムを生成した場合，通信は通信層での全対全通信 (`MPI_Alltoallv` 命令) によって行われる．この全対全通信に要する時間は，送受信データの量や通信に参加するプロセッサの数，送受信を行うプロセッサの組合わせ等によって異なり，スケジュールを生成する前にこの値を決めることは難しい．有向辺の重みを正確に見積もることが，必ずしも性能の良い並列プログラムを生成することにつながるわけではないが，ここでは有向辺の重みを以下のような方法で与えている．

まず初めに，利用するプロセッサ全てが参加する全対全通信の時間を用いて有向辺の重みを与える．そして，スケジューリングを行い並列プログラムを生成し，実際に実行して通信層の時間を計測する．そして，その値を有向辺の重みとして与え直し並列プログラムを生成する．これを数回繰り返し，与えた有向辺の重みと実際の実行時の通信層に要する時間が近くなった時点で適切な有向辺の重みと見なす．

6.2 ガウスジョルダン法

ガウスジョルダン法 (以降 GJ) のプログラムを図 6 に示す．これを元にタスクグラフを生成する場合，1 代入文レベルの細粒度タスクグラフではタスク数は問題サイズ N に対して $N^2(N+1)/2$ となる．そのため，問題サイズ 128 変数でタスク数は約 100 万となり，表 2 のように，スケジュール生成に要する時間は 10 数時間となるため，さらに大きな問題サイズに対して細粒度のタスクグラフを用いてスケジュールを生成するのは現実的でない．そこで，タスク数を削減するため粒度を粗くしていく．

j のループ及び i のループが DOALL 型であることから次の粒度が考えられる．

- i のループの n 回を 1 タスクとする．(**粒度** A_n)
- j のループの本体を 1 タスクとする．(**粒度** B)
- j のループの n 回を 1 タスクとする．このとき，次の 2 つの場合でタスクグラフの形状が異なる．
  - $j \ mod \ n = 1$ の j から $j \ mod \ n = 0$ の j までを 1 タスクとする (**粒度** C_n)
  - $(j-k-1) \ mod \ n = 0$ の j から $(j-k-1) \ mod \ n = n-1$ の j までを 1 タスクとする (**粒度** C'_n)

粒度 C_n と C'_n の違いを図 7 に示す．丸が j のループのイタレーション 1 回 (粒度 B の 1 タスク) を表し，点線が粒度 C_2 と C'_2 の 1 タスクを表している．矢印は粒度 B のタスクグラフの有向辺を表す．両者はタスク数と並列度の減り方は同じだが，粒度 C_2 の方は有向辺が 1 つにまとまるため粒度 C'_2 に比べて有向辺の数が少ない．

```
for(k = 0; k < N; k++)
  for(j = k + 1; j <= N; j++){
    a[k][j] = a[k][j] / a[k][k];
    for(i = 0; i < N; i++)
      if(i != k)
        a[i][j] = a[i][j] - a[i][k] * a[k][j];
  }
```

図 6 ガウスジョルダン法

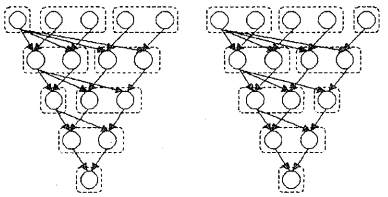

図 7　GJ(サイズ 5) の粒度 C_2 と C_2'

表 1　GJ(サイズ 128) の各粒度のタスクグラフ

	タスク数	最大並列度	CP	平均出次数
細粒度	1056768	16383	256	2.96
粒度 A_{32}	41280	256	4100	3.16
粒度 B	8257	128	16384	1.97
粒度 C_2	4128	64	32740	1.92
粒度 C_2'	4128	64	32740	2.90

問題サイズ 128 変数のときの各粒度のタスクグラフの形状を表 1 に示す．ここで，平均出次数とは全てのタスクの出次数の平均値である．各粒度のタスクグラフについて，スケジュール生成を行った．スケジューリング時間を表 2 に示す．スケジューリング時間は粒度を上昇させると大幅に削減されている．

表 2　GJ のスケジューリング時間 (プロセッサ 4 台)

粒度	128 変数	256 変数	512 変数
細粒度	12 時間	***	***
粒度 A_{32}	25 分	3 時間	***
粒度 A_{64}	80 秒	20 分	***
粒度 B	10 秒	2 分	1 時間
粒度 C_2	2 秒	12 秒	6 分
粒度 C_4	1 秒以下	3 秒	50 秒

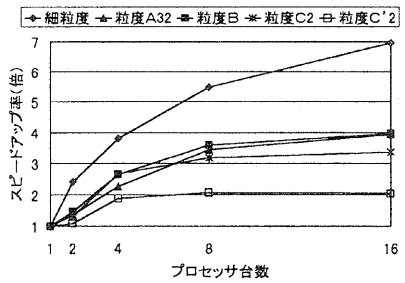

図 8　GJ(サイズ 128) の実行結果

生成した並列プログラムの実行結果を図 8 に示す．細粒度が最もよい性能が得られている．また，表 1 より粒度 A_{32} は粒度 B より並列度が大きく，CP は短いにも関わらず性能は粒度 B とあまり変わらない．粒度 A_{32} は粒度 B に比べて平均出次数が大きく，表 4 に示したように出次数が 4 以上のタスクが 20%存在するのに対し，粒度 B は 95%以上が出次数 1 である．5.3 節で述べたように出次数の大きいタスクはクラスタリングの過程で複製されやすい．そのため，粒度 A_{32} では複製が多くなっている．また，通信層も多くなっているが，3.2 節で述べたように，タスクの複製は通信を削減するために行われる．そのため，複製と通信層がともに多い粒度 A_{32} は性能が良くない．粒度 C_2' と粒度 C_2 の差についても同様に，粒度 C_2' の方が出次数の大きいタスクが多く存在するためである．

表 3　各出次数のタスクの割合 (単位%)

	0	1	2	3	4 以上
細粒度	0.2	95	0.2	0.2	4.4
粒度 A_{32}	0.01	59	20	0.01	21
粒度 B	0.05	96	0.05	0.05	3.85
粒度 C_2	0.05	96	0.05	0.05	3.85
粒度 C_2'	0.02	1.5	95	0.05	0.3.43

表 4　タスク複製率と通信層数 (プロセッサ 8 台)

	複製率	通信層
細粒度	1.046192	19
粒度 A_{32}	1.535771	18
粒度 B	1.392113	14
粒度 C_2	1.327519	16
粒度 C_2'	1.694471	31

細粒度では問題サイズ 128 までしか扱えなかったが，粒度を粗くすることにより問題サイズ 2048 まで扱えるようになった．問題サイズ 2048 での各粒度の実行結果を図 9 に示す．プロセッサ 2 台の場合は各粒度ともスピードアップ率が約 2 倍という理論上の理想値が得られているが，割り当てるプロセッサ数を多くしていくと，粒度間の性能差が大きくなる．これは，割り当てるプロセッサ数とグラフの並列度が近くなると，スケジューリングによる並列化の効果が低下するためである．

6.3　FFT

FFT における主要計算部分を図 10 に示す．ここで添字計算を除く計算本体の代入文の実行回数は問題サイズ N に対して $3NlogN$ となり，これが代入文レベルの細粒度タスクグラフのタスク数となる．そのため，サイズ 16384 でタスク数は 688128 となり，表 6 のようにスケジュール生成に要する時間は約 10 時間となり，これ以上大きな問題について細粒度タスクグラフを用いるのは現実的でない．そこで，タスク数を削減

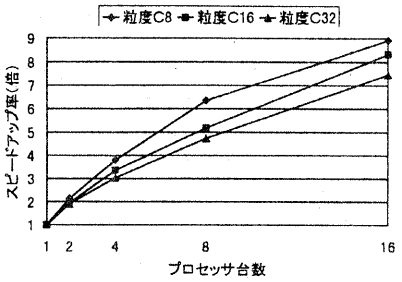

図 9　GJ(サイズ 2K) の実行結果

するために粒度を粗くしていく．

```
for(i = 1; i <= P; i++){ /* P = logN */
  m = pow(2, i); h = pow(2, i-1);
  for(j = 0; j < h; j++)
    for(k = j; k < N; k += m){
      /* 添字計算 */
      kp=k+h; p=j*(N/m);
      /* 計算本体 */
      vR=fR[kp]*w[p]-fI[kp]*w[p+N/2];
      vI=fR[kp]*w[p+N/2]+fI[kp]*w[p];
      fR[kp]=fR[k]-vR; fR[k]=fR[k]+vR;
      fI[kp]=fI[k]-vI; fI[k]=fI[k]+vI;
    }
}
```

図 10　FFT

j のループと k のループは 2 重のネストした DOALL 型ループであるためループコアレシングの手法 [8] を用いて 1 重の DOALL 型ループとする (図 11)．そして，得られた DOALL 型ループの定数回を 1 タスクとする．

```
for(i = 1; i <= P; i++){ /* P = logN */
  m = pow(2, i); h = pow(2, i-1);
  g = pow(2, P-i);
  for(q = 0; q < N / 2; q++){
    /* 添字計算 */
    j = q / g; k = j + m * (q % g);
    kp=k+h, p=j*(N/m)
    /* 計算本体 */
        (省略)
  }
}
```

図 11　FFT(ループコアレシング適用後)

以降，各粒度を以下のように呼ぶ．

- ループ本体を 1 タスクとする．(粒度 A)
- ループコアレシング適用後，ループの n 回を 1 タスクとする．(粒度 B_n)

粒度 A から粒度 B_2 へのタスクグラフの変化を図 12 に示す．実線の丸が粒度 A のタスク，点線が粒度 B_2 のタスクを表し，矢印は粒度 A の有向辺を表している．

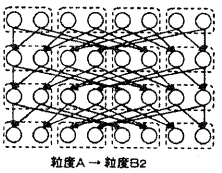

図 12　FFT(サイズ 16) のタスクグラフ

表 5　FFT(サイズ 16K) の各粒度のタスクグラフ

	タスク数	最大並列度	CP	平均出次数
細粒度	688128	32768	28	1.90
粒度 A	114688	8192	84	1.86
粒度 B_4	28672	2048	326	1.86
粒度 B_{32}	3584	256	2608	1.86

各粒度のタスクグラフの形状を表 5 に示す．粒度を粗くすると並列度が減少し．CP は長くなる．出次数については，ほぼ変わらなかった．

表 6　FFT のスケジューリング時間 (プロセッサ 4 台)

サイズ	細粒度	粒度 A	粒度 B_4	粒度 B_{64}	粒度 B_{256}
16K	10 時間	6 分	16 秒	1 秒	1 秒以下
128K	***	***	10 分	3 秒	1 秒以下
2M	***	***	***	1 時間	2 分

次に各粒度のタスクグラフについてスケジューリングを行い，スケジューリングに要した時間を測定した (表 6)．粒度を上げるとスケジューリング時間が短縮されており，サイズ 131072 や 2097152 など細粒度ではスケジューリングに膨大な時間がかかると思われるものについても，粒度を上げることでスケジュールを得ることができた．サイズ 16384 について各粒度のタスクグラフをもとに生成した並列プログラムの実行結果を図 13 に示す．各粒度の性能を比較すると，細粒度よりループ本体を 1 タスクとした粒度 A の方が性能が良く，全体で最も良い結果が得られている，これは先に述べた，BCSH においては細粒度タスクグラフを用いた方が性能の良い並列プログラムを生成できるという予想に反する．これはコード生成時に行われる次のような最適化が原因である．プログラムのループを展開してタスクグラフを生成し，並列プログラムを生成する場合，そのままでは生成されるコードもループを展開した状態になり，プログラムのコードサイズが大きくなる．そのため，コード生成時に，再びループに戻せる部分についてはループに戻し (ループ再構成

), コードサイズを小さくする処理を行っている．図7にループ再構成を行う場合と行わない場合の比較を示す．細粒度はループ再構成を行ってもコードサイズはあまり小さくなっていない．これは，粒度 A がループの本体を1タスクにしているのに対し，細粒度はループの本体を分割してしまうためループ再構成が効率的に行われなかったためである．また，ループ再構成をしない場合，実行速度が遅くなるのは，コードサイズが大きくなるとプログラム実行時の命令キャッシュのヒット率が悪くなるためと考えられる．

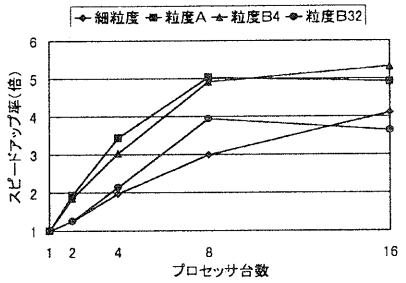

図 13　FFT(サイズ 16K) の実行結果

表 7　ループ再構成の影響 (FFT サイズ 16K プロセッサ 2 台)

		細粒度	粒度 A
ループ再構成有	コードサイズ	1.4MB	21KB
	実行時間 (秒)	0.034395	0.023986
ループ再構成無	コードサイズ	3.8MB	4.1MB
	実行時間 (秒)	0.050125	0.051437

問題サイズ 2097152 での各粒度のタスクグラフの最大並列度と生成した並列プログラムの実行結果を表8と図14に示す．プロセッサ2台の場合は粒度 C_{4096} の最大並列度256はプロセッサ2台に対して十分大きいため十分な性能が得られているが，プロセッサ16台の場合は粒度 C_{128} や粒度 C_{1024} に比べて性能の低下が大きい．これはプロセッサ16台に対して粒度 C_{4096} の最大並列度256は十分大きな値とはいえないためである．

表 8　FFT(サイズ 2M) のタスクグラフの最大並列度

粒度 C_{128}	粒度 C_{1024}	粒度 C_{4096}
8192	1024	256

7. おわりに

本稿では，タスクスケジューリングを用いた並列プログラム生成においてのタスク粒度の調整について述べた．BCSH を用いた場合，粒度が細かいタスクグラフを生成した方がプログラムの並列性を十分に引き出せるが，タスク数が多くなるためスケジューリング時間が長くなりサイズの大きな問題が扱えない．そこで粒度を粗くしタスク数の削減を行った．また，粒度を粗くした場合のタスクグラフの変化が並列プログラムの性能に与える影響について評価を行った．

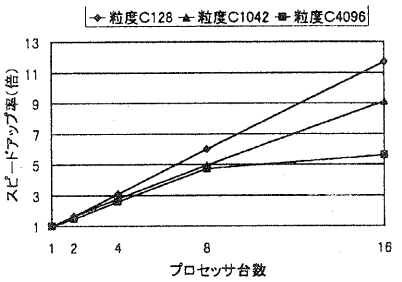

図 14　FFT(サイズ 2M) の実行結果

結果として，粒度を粗くする場合はタスクの出次数を小さくすることが重要であるとわかった．また，粒度を粗くした場合のグラフの並列度の減少については，並列度がプロセッサ数より十分大きい場合は影響が少なかった．よって，出次数が少なく，割り当てるプロセッサ数に対して並列度を十分にとるような粒度の上げ方が有効であることがわかった．

謝　辞

本研究は一部平成11～12年度文部省科学研究費補助金・基盤研究 (C) (11680357) およびＰＤＣ (並列・分散処理研究推進機構) の補助による．

参考文献

1) Aho, A. V., Sethi, R., and Ullman, J. D.: "Compilers Principles, Techniques, and Tools", *Addison-Wesley Publishing Company*, (1986).
2) Fujimoto, N., Baba, T., Hashimoto, T., and Hagihara, K.: "A Task Scheduling Algorithm to Package Messages on Distributed Memory Parallel Machines", *Proc. of I-SPAN'99*, pp.236-241, (1999).
3) Kruatrachue, B. : "static task scheduling and packing in parallel processing systems", Ph.D. diss., *Department of Electrical and Computer Engineering*, Oregon State University, Corvallis, (1987).
4) Message Passing Interface Forum: "*MPI: A Message-Passing Interface Standard Version 1.1*", (1995).
5) MPICH: A Portable MPI Implementation, http://www.mcs.anl.gov/mpi/mpich/(1999).
6) Myrinet: Myricom, http://www.myri.com/(1999).
7) Valiant, L.G.: "A Bridging Model for Parallel Computation", *Communication of ACM*, Vol.33,No8, (1990).
8) Wolf, M.: "High Performance Compilers for Parallel Computing", *Addison-Wesley Publishing Company*, (1996).