アルゴリズム 71−4
(2000. 1. 17)

分割統治法アルゴリズムの効率的な並列化手法とそのコンパイラの実装

中島大輔　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

並列再帰呼出を用いた分割統治法アルゴリズムを並列計算機上で実行する場合，そのアルゴリズムの動きや計算量，計算機の通信性能などにより，性能が大きく左右される．このため全ての分割統治法アルゴリズムに対してただ1つの並列実行方式を適用しても十分な性能は得られない．その一方で，それぞれの分割統治法アルゴリズムのソースコード記述から，コンパイラがアルゴリズムの特徴を機械的に解析するのは極めて困難である．本稿では，各アルゴリズムの適切な並列実行に必要な情報をプログラマが明示的に指定することによりコンパイラに効率良い並列プログラムを生成させる手法を提案する．そして，特徴の異なる分割統治法アルゴリズムを提案手法に基づいて実際にコンパイルし，生成された並列プログラムの性能を測定した結果を示す．

A Compilation Technique for Effective Parallelization of Divide-and-Conquer Algorithms

Daisuke Nakajima, Noriyuki Fujimoto and Ken-ichi Hagihara

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

In parallel execution of a divide-and-conquer algorithm, the performance of the program depends on the behavior or the complexity of the algorithm, a communication performance of the machine, and so on. Therefore, one execution method is not always effective for all types of divide-and-conquer algorithms. However, it is quite difficult for a compiler to analyze the characteristics of the algorithms automatically from the source program. In this paper, we propose the technique to make a compiler generate effective parallel programs by programmer's selection of a suitable way for the execution of each type of divide-and-conquer algorithms. After that, we show the result of experiment to evaluate the performance of the different types of parallel divide-and-conquer programs compiled by proposed technique.

1. はじめに

分割統治法とは，ある問題を複数の部分問題に分割しそれぞれの部分問題の解を統合することで全体の解を得るアルゴリズムである．大規模な問題を複数のプロセッサを用いて解く並列アルゴリズムの設計の際にもしばしば現れる重要な手法である [1]．

この分割統治法アルゴリズムにおいて，部分問題の処理を独立に実行可能な再帰関数として表せる場合に，それら複数の再帰呼出を並列実行するような並列プログラムとして記述できれば，その並列アルゴリズムを簡潔にプログラミングすることができ，かつ並列実行による実行時間の短縮が期待できる．この再帰関数の並列呼出を並列再帰とよぶ．

一方，これまで提案されている並列プログラミング言語処理系においては，並列再帰を記述およびコンパイル可能なものは少なかった．また，並列再帰を実装していたとしても，プログラムの記述が煩雑である，特定のアルゴリズムを除いては低い実行効率であるといった問題点があった [7][8][9]．本稿では，それぞれの分割統治法アルゴリズムの特徴に適した効率良い並列実行の手法を紹介し，プログラマがそれらの並列実行方式を明示的に選択・指定することによりコンパイラに実行性能の良い並列プログラムを生成させる手法を提案する．

2. 並列再帰による分割統治法の並列実行

2.1 分割統治法アルゴリズム

分割統治法アルゴリズムでは，次のように問題を解く．

(1)　問題を複数の部分問題に分割する．
(2)　それぞれの部分問題を解く．
(3)　部分問題の解から元の問題の解を求める．

(2) における部分問題の解が (1)〜(3) によって同様に計算される場合がしばしばある．このような場合には，(1)〜(3) を再帰関数として実装し，(2) では各部分問題

についてその再帰関数を再帰呼出することにより，アルゴリズムを簡潔に記述することができる．

分割統治法アルゴリズムが「ある部分問題の計算が別の部分問題の計算に影響しない (独立)」という条件を満たすとき，各部分問題の計算を並列に実行することができる．このように分割統治法を並列実行することでプログラム実行時間の短縮が期待できる．

本稿で対象とする分割統治法アルゴリズムは次の 2 つの特徴を持つものとする．

- 部分問題の計算を再帰関数で表現可能．
- 部分問題を独立に実行可能．

2.2 並列再帰

分割統治法アルゴリズムを並列実行する手法として，「並列再帰」がある．これは，部分問題に対する処理が複数の独立な再帰関数で実装されている場合，それらの再帰呼出を並列に実行するものである．

一度の並列再帰呼出で実行可能な再帰呼出の数を並列再帰の「分岐数」と呼ぶ．分岐数は分割された部分問題の数に相当する．分岐数が F のとき，F 個の再帰呼出をそれぞれ $Call_1, ..., Call_F$ とする．また，$Call_i$ の引数のうち入力として用いるものを $IArg_i$，出力として用いるものを $OArg_i$ とする．$Call_i$ を $Call(IArg_i, OArg_i)$ とも書く．

分岐数 F の並列再帰呼出は次のように表せる．

par i=1 to F
　　Call($IArg_i$, $OArg_i$)

ある再帰呼出 $Call_i$ を割り当てるプロセッサの決定方法として，次の 2 つが考えられる．

- 単一プロセッサへの割当て:
  1 つの再帰呼出 $Call_i$ を 1 台のプロセッサ P へ割り当てる．P が $Call_i$ を計算する．
- プロセッサグループへの割当て:
  1 つの再帰呼出 $Call_i$ を複数のプロセッサの集合であるプロセッサグループ G へ割り当てる．G に含まれるプロセッサが協調して $Call_i$ を計算する．

現在の再帰呼出を実行している P または G をそれぞれ P_{src} または G_{src} とする．再帰呼出を割り当てられる P または G をそれぞれ P_{dst} または G_{dst} とする．

分岐数 F の並列再帰呼出においては，$Call_1, ..., Call_F$ のうち少なくとも 1 つの再帰呼出は自プロセッサで行うのが一般的である．ここでは，$Call_1$ を自プロセッサで実行し，残りの $Call_2, ..., Call_F$ はできるかぎり他のプロセッサへの割当てを試みるものとする．使用可能なプロセッサ数が分岐数 F に満たない場合は，$Call_1$ を終了し他プロセッサから $OArg_i$ を受信した後，同様の操作を未計算の $Call_i$ について繰り返す．

並列再帰呼出における各プロセッサの動作は次のように記述できる．Send(D,P) はプロセッサ P へのデータ D の送信，Recv(D,P) はプロセッサ P からのデータ D の受信を表す．P_{src} または G_{src} は，1. 再帰呼出実行の割当て，2. 自プロセッサでの実行，3. 再帰呼出結果の受信を行う．

- P_{src} または G_{src} の動作
  1.for i=2 to F
  　　Send ($IArg_i$, P_{dst_i} または G_{dst_i})
  3.Call ($IArg_1$, $OArg_1$)
  2.for i=2 to F
  　　Recv ($OArg_i$, P_{dst_i} または G_{dst_i})
- P_{dst_i} または G_{dst_i} の動作
  1.Recv ($IArg_i$, P_{src} または G_{src})
  2.Call ($IArg_i$, $OArg_i$)
  3.Send ($OArg_i$, P_{src} または G_{src})

3. 分割統治法の並列実行における問題点

上で述べた分割統治法の並列実行方式において，実行性能を低下させる問題点として次の 2 つが考えられる．

問題点 1: プロセッサ間の負荷バランスが予測不可能
分割統治法アルゴリズムの動きによっては，各プロセッサに割り当てられる部分問題の計算量に大小の差が生じる．このようなプロセッサ間での負荷の不均等はコンパイル時には予測できず，プロセッサ間の負荷バランスを均等に保つことができないことが多い．このため，あるプロセッサは早い時刻に処理を完了し待ち状態にあるにもかかわらず，別のプロセッサは大量の計算を続けているという状況を引き起こす可能性がある．このプロセッサ効率の低下は全体の実行性能の低下につながる．

問題点 2: 過剰な並列化によるオーバーヘッド
再帰呼出を他プロセッサに割り当てて並列計算を行うと，次のオーバーヘッドが生じる．

- 再帰関数の入力引数/出力引数の送受信時間
- 割当て先プロセッサ決定のための処理時間

このため，現在の再帰呼出の計算時間が十分に小さくなっているにもかかわらず並列再帰呼出により並列化を行うと，その再帰呼出を自プロセッサのみで計算した場合よりも実行時間がかかる．

4. 分割統治法の並列実行方式の改善

本稿で提案する手法では，プロセッサ間の負荷の不均等を解決するために二種類の「負荷分散方式」をプログラマが選択する．また過剰な並列化によるオーバーヘッドを抑制するために「並列化条件」を導入する．

4.1 並列化条件の設定

並列化条件とは，並列再帰呼出の際に，他のプロセッ

サに再帰呼出実行を依頼するか，あるいは自プロセッサで全ての再帰呼出を実行するかを決定する条件式である．

適切な並列化条件を設定すれば，過剰な並列化によるオーバーヘッドを避けることができる (6.3.2 参照)．適切な並列化条件は個々の分割統治法アルゴリズムに依存し，あるアルゴリズムでは，「配列データサイズが 4000 以上」であり，「再帰呼出の深さが 4 以下」である．適切な並列化条件はコンパイラが機械的に解析することが極めて困難なため，プログラマがアルゴリズムの動きや計算量，並列計算機の通信性能等を考慮してソースプログラム中に記述する．

4.2 負荷分散方式の指定

本研究では，分割統治法アルゴリズムの各部分問題の計算量が均等であるか不均等であるかに対応した適切な負荷分散を実現するために「静的負荷分散方式」および「動的負荷分散方式」を採用する．

4.2.1 静的負荷分散方式

静的負荷分散とは，プロセッサへの処理の割当てが固定的に行われる負荷分散方式である．この方式では，プロセッサは割り当てられた処理を完了後，再び処理を割り当てられることはない．

並列に実行される各再帰呼出の計算量がほぼ均等であると考えられる場合には，余分なオーバーヘッドのかからない静的負荷分散方式が効果的である．

この方式では，1 つの再帰呼出をプロセッサグループ G に割り当てる．ここで G に含まれるプロセッサ数を S_G とする．G には 1 台のリーダ L_G が存在し割り当てられた再帰関数の実行を開始する．ある 1 台のプロセッサ L_{root} が全てのデータを自メモリに持ち，最初の再帰呼出の実行を開始する．全プロセッサは「計算状態」または「待ち状態」のいずれかの状態をとる．リーダ以外のプロセッサは「待ち状態」にあり，リーダからの指令を待つ．再帰関数中で並列再帰呼出が出現すると，L_G は G を複数のサブプロセッサグループ G_i に分割し，各 G_i について 1 台のサブリーダ L_{G_i} を決定する．L_G は複数の再帰呼出を G_i へ割り当てる．

また，並列再帰呼出の前後に配列代入文などデータ並列計算可能な部分があれば，L_G は G の他のプロセッサを用いて並列計算をすることができる．

プログラム実行中にはプロセッサ間でメッセージ送受信を行う．プロセッサ間で送受信する各種メッセージを表 1 に示す．各プロセッサの動作を以下に示す．

計算状態のプロセッサ L の動作 (再帰関数 $Call$)

(1) 並列再帰呼出の前の計算．

(2) 並列再帰呼出の実行

- 未計算の再帰呼出 UC=$\{Call_1, ..., Call_K\}$.
- 「並列化条件」の判定

• 「並列化条件」を満たす場合
  - UC が空となるまで以下を繰り返す．
    – $X = min(K, S_G)$
    – G をサブグループ $G_1 ... G_X$ に分割．
    – 各 G_i から 1 台のサブリーダ L_{G_i} を選ぶ．
    – for i=2 to X Send($Order_i(src)$, L_{G_i})
    – for i=2 to X Send($Member_i$, L_{G_i})
    – for i=2 to X Send($IArg_i$, L_{G_i})
    – Call($IArg_1$, $OArg_1$)
      $Call_1$ を UC から除去．
    – for i=2 to X Recv($OArg_i$, L_{G_i})
      $Call_i$ を UC から除去．

• 「並列化条件」を満たさない場合
  - for i=1 to K Call($IArg_i$, $OArg_i$)

(3) 並列再帰呼出の後の計算．

待ち状態のプロセッサの動作

(1) リーダからのメッセージを待つ．[待ち状態]

(2) メッセージを受信すると以下のように動作する．

• メッセージが $Order(L)$ の場合
  - Recv($IArg$, L)
  - Recv($Member$, L)
  - Call($IArg$, $OArg$) [計算状態へ推移]
  - Send($OArg$, L) [待ち状態へ推移]

• メッセージが $Quit$ の場合
  - 全ての処理を終了．

4.2.2 動的負荷分散方式

動的負荷分散方式とは，プロセッサへの処理の割当てが計算の実行中に変更可能である負荷分散方式である．この方式では，プロセッサは割り当てられた処理を完了後，再び未計算の処理を割り当てられる可能性がある．

並列に実行される各再帰呼出の計算量に偏りがあり，かつその計算量の偏りがコンパイル時には予測できない場合に，動的負荷分散によりプログラム実行中における適切な負荷の均等化の実現が期待できる．

本研究では，マネージャ・ワーカ法 [4] を用いた動的負荷分散方式を実装する．マネージャ・ワーカ法による動的負荷分散方式では，N 台のプロセッサを 1 台のマネージャM と (N-1) 台のワーカ W_i に分ける．M は全ての W_i の状態の管理および W_i への処理の割当てを行う．W_i が再帰関数の計算を実行する．ある 1 台のワーカ W_1 が全てのデータを自メモリに持ち，最初の再帰呼出の実行を開始する．W_i は「計算状態」あるいは「待ち状態」のいずれかの状態をとる．計算状態の W_i がある再帰呼出の実行を完了すると再び待ち状態となり，M によって新たに他の W_i の処理の一部を割り当てられる．

プログラム実行中にはマネージャ–ワーカ間およびワーカ–ワーカ間でメッセージ送受信を行う．プロセッサ間で送受信する各メッセージを表 2 に示す．各プロセッサの動作の詳細を以下に示す．

表 1　メッセージの種類 (静的負荷分散方式)

種類	パラメータ	方向	意味
$Order_i$	L の ID, $IArg_i$ および $OArg_i$ のサイズ	$L \rightarrow L_{G_i}$	サブリーダ L_{G_i} に再帰呼出の実行を指令
$Member_i$	サブグループ G_i のメンバプロセッサ ID	$L \rightarrow L_{G_i}$	サブリーダ L_{G_i} に G_i のメンバプロセッサ ID を通知
$Quit$	なし	$L_{root} \rightarrow ALL$	全てのプロセッサに計算の終了を通知

表 2　メッセージの種類 (動的負荷分散方式)

種類	パラメータ	方向	意味
$Request_i$	W_{src} の ID, $IArg_i$ および $OArg_i$ のサイズ	$W_{src} \rightarrow M$	M に再帰呼出の割当てを依頼
$Order_i$	W_{src} の ID, $IArg_i$ および $OArg_i$ のサイズ	$M \rightarrow W_{dst}$	W_{dst} に再帰呼出の割当てを指令
Accept	W_{dst} の ID	$M \rightarrow W_{src}$	割当て依頼を受理し, W_{dst} の ID を返す
Discard	なし	$M \rightarrow W_{src}$	割当て依頼を却下し, -1 を返す
Finish	W の ID	$W \rightarrow M$	再帰呼出実行の完了を通知
Quit	なし	$M \rightarrow W$	全てのワーカに計算の終了を通知

マネージャM の動作

(1)　ワーカ W_1 を「計算状態」とする.
　　W_1 以外のワーカを「待ち状態」とする.

(2)　ワーカからのメッセージを待つ.

(3)　メッセージを受信すると以下のように動作する.

- メッセージが $Request(src)$ の場合
  - 待ち状態ワーカが存在するか調べる.
    - 待ち状態ワーカが存在する場合
      - 待ち状態ワーカの 1 つ W_{dst} を選択.
      - Send($Order(src)$, W_{dst})
      - W_{dst} を「計算状態」とする.
      - Send($Accept(dst)$, W_{src})
    - 待ち状態ワーカが存在しない場合
      - Send($Discard$, W_{src})
  - (2) で再びメッセージを待つ.
- メッセージが $Finish(src)$ の場合
  - W_{src} が W_1 であった場合
    - for i=1 to N-1 Send($Quit$, W_i)
    - 並列再帰計算終了.
  - W_{src} が W_1 以外であった場合
    - W_{src} を「待ち状態」とする.

計算状態のワーカ W_{src} の動作 (再帰関数 $Call$)

(1)　並列再帰呼出の前の計算.

(2)　並列再帰呼出の実行

- 未計算の再帰呼出 UC=$\{Call_1, ..., Call_K\}$.
- 「並列化条件」の判定.
  - 「並列化条件」を満たす場合
    - UC が空となるまで以下を繰り返す.
      - for i=2 to K
        - Send($Request_i(src)$, M)
        - M からの応答メッセージを待つ.
          * メッセージが $Accept(dst)$ の場合
            - Send($IArg_i$, W_{dst})
          * メッセージが $Discard$ の場合
            - $Call_i$ は未計算のまま保留.
      - Call($IArg_1$, $OArg_1$)
        $Call_1$ を UC から除去.
      - for i=2 to K Recv($OArg_i$, W_{dst})
        $Call_i$ を UC から除去.
  - 「並列化条件」を満たさない場合
    - for i=1 to K Call($IArg_i$,$OArg_i$)

(3)　並列再帰呼出の後の計算.

待ち状態のワーカ W_{dst} の動作

(1)　M からのメッセージを待つ. [待ち状態]

(2)　メッセージを受信すると以下のように動作する.

- メッセージが $Order(src)$ の場合
  - Recv($IArg$, W_{src})
  - Call($IArg$, $OArg$) [計算状態へ推移]
  - Send($OArg$, W_{src})
  - Send($Finish(dst)$, M) [待ち状態へ推移]
- メッセージが $Quit$ の場合
  - 全ての処理を終了.

複数マネージャプロセッサの使用

使用するワーカ数が多い場合, または並列再帰呼出の分岐数が大きい場合には, M と W_i 間の通信頻度が高くなる. その結果, M に高負荷がかかり M から W_i への応答および指令が遅れ, プログラム全体の実行効率が低下する. このような場合には, 起動する M を複数として, 各 M_j が管理するワーカ数を減少させることで負荷を分散させる (6.3.3 参照).

5. 分割統治法プログラムのコンパイル

上で述べた並列実行方式を直接記述することはプログラマに大きな負担をかける. そこで, 我々が提案している Work-Time C 言語を用いて分割統治法アルゴリズムを記述し, コンパイラにより上記の並列実行方式に基づいて並列計算機上で動作するプログラムを生成する.

5.1 Work-Time C によるプログラムの記述

Work-Time C 言語 [2] は，標準 C 言語に並列構文 par..to..do を追加した高水準言語であり，実際のプロセス数やプロセス間通信を意識しない抽象度で並列アルゴリズムを記述することが可能である．

Work-Time C では，並列再帰を次のように記述する．

```
void Mergesort(int *ary, int n)
in:  ary;
out: ary;
L:   n>4000;
{
  int i,j,k,m,top[2],size[2],*b;
  if(n<=1){
    return;
  }
  else{
    b = (int *)malloc(n*sizeof(int));
    m = (n-1)/2;
    top[0] = 0;   size[0] = m+1;
    top[1] = m+1; size[1] = n-(m+1);
    par x=0 to 1 do
      Mergesort(&ary[top[x]],size[x]);
    for(i=m+1; i>0; i--)
      b[i-1]=ary[i-1];
    for(j=m;   j<n-1; j++)
      b[n-1+m-j]=ary[j+1];
    for(k=0; k<=n-1; k++)
      ary[k] = (b[i]<b[j]) ? b[i++] : b[j--];
    free(b);
  }
}
```

この例では，再帰関数 Mergesort 中で 2 個の独立な再帰呼出を並列に実行することを示している．引数は配列データ領域の先頭アドレス ary とその領域のサイズ n である．このように，再帰関数の引数には，j 番目の引数 argj，j 番目の引数のサイズ sizej を並べて指定する．実際の引数 argj だけでなく，各引数のサイズ sizej を付加しているのは，プロセッサ間で引数データを送受信する際に，データサイズを指定する必要があるからである．また，入力引数を in:で指定し，出力引数を out:で指定する．L:に並列化条件を設定する．この例では n>4000 が成立するときのみ，par 内の再帰呼出を並列実行する．

また，静的負荷分散方式におけるプロセッサグループによるデータ並列計算は次のように記述する．

```
......
par i=0 to M-1 do
  A[i] = x * B[i] + y * C[i];
......
```

この例では，各 i に対応する M 個の配列代入文を並列に行うことを示している．

5.2 並列実行方式指定に基づくコンパイル

5.2.1 コンパイルの概要

本研究では，Work-Time C 言語で記述された分割統治法アルゴリズムのソースプログラムを，通信ライブラリ MPI[6] を用いて C 言語で記述された中間ソースプログラムに変換するコンパイラを実装した．中間ソースプログラムは，対象とする並列計算機の C コンパイラでコンパイルし，最終的な実行コードを得る．

Work-Time C 言語で記述されたソースプログラムをコンパイルする際に，静的負荷分散/動的負荷分散のどちらの実行方式を用いるかをコンパイラオプション (-static/-dynamic) により指定する．また-dynamic を指定した際には，起動するマネージャ数を指定する．

```
wtc2mpi program.c -dynamic -m2
```

5.2.2 中間ソースプログラムの生成

Work-Time C ソースプログラムをコンパイルすると次の関数を含む中間ソースプログラムが生成される．

- 再帰関数 RecFunc
  静的負荷分散方式では，4.2.1 で述べた「計算状態のプロセッサの動作」を担当する．動的負荷分散方式では，4.2.2 で述べた「計算状態のワーカの動作」を担当する．Work-Time C ソースプログラムにおける再帰関数 RecFunc の par 文による並列再帰呼出を変形して生成する．
- MainForWorker
  静的負荷分散方式では，4.2.1 で述べた「待ち状態のプロセッサの動作」を担当する．動的負荷分散方式では，4.2.2 で述べた「待ち状態のワーカの動作」を担当する．Work-Time C ソースプログラムから，再帰呼出の関数名 RecFunc，実引数の型，実引数の名前を解析し，W_{src} から受信した実引数を用いて *RecFunc* を呼び出すコードを生成する．また，静的負荷分散方式では，データ並列計算における配列代入文も含む．
- MainForRoot
  最初に再帰計算を開始する 1 台のプロセッサが実行する．元々の main 関数を変形する．

これらに加えて，動的負荷分散方式では，4.2.2 で述べた「マネージャの動作」を担当する関数 MainForManager が必要である．この関数は各分割統治法アルゴリズムのソースプログラム記述に依存しないため，他の補助関数群とともに実行時ライブラリとして実装しリンクする．

6. 性能評価実験と考察

特徴の異なる分割統治法アルゴリズムをコンパイルした並列プログラムの性能を測定する実験を行った．

6.1 評価基準と実行環境

性能評価の基準としてスピードアップ S_p を用いる．プロセッサ p 台を用いたときの S_p は次式で定義する．

$$S_p = \frac{\text{プロセッサ1台での実行時間}}{\text{プロセッサ}\ p\ \text{台での実行時間}}$$

S_p の値が高いほど，プログラムの並列化による実行効率の向上が大きいことを示す．

また，実行環境として日本電気(株)の並列計算機Cenju-3[5] (プロセッサ VR4400SC，最大PE数128，通信性能 40MB/秒，メモリ 64MB) を使用した．

6.2 アルゴリズムの特徴

分割される部分問題の負荷バランスの傾向および並列再帰の分岐数，アルゴリズムの計算量が異なる5種類のアルゴリズムについて性能評価を行う．各アルゴリズムの特徴を表3に示す．

表3 分割統治法アルゴリズムの特徴

アルゴリズム	負荷	分岐数	計算量
マージソート	均等	2	$O(NlogN)$
高速フーリエ変換	均等	2	$O(NlogN)$
Strassen の行列積	均等	7	$O(N^{log7})$
クイックソート	不均等	2	$O(NlogN)$
Nクイーン問題	不均等	N	$O(N^N)$

6.3 実験結果

6.3.1 負荷分散方式の比較

各アルゴリズムを静的負荷分散方式/動的負荷分散方式を指定してコンパイルした並列プログラムのスピードアップを測定した結果を図1～図5に示す．グラフ中のstatic は静的負荷分散，dynamic は動的負荷分散の結果を示す．また，L:は並列化条件を表す．N は入力データサイズを表し，マージソート，高速フーリエ変換，クイックソートでは配列データサイズN，Strassen の行列積では行列サイズN×N，Nクイーン問題では盤面サイズN×Nである．

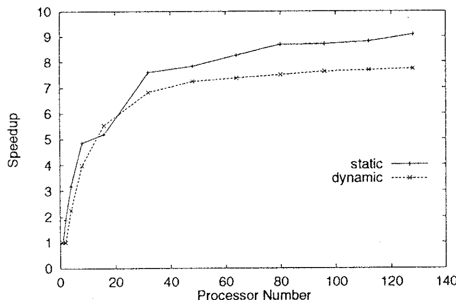

図1 マージソート (N=4M,L:n>4K)

マージソート (図1)，高速フーリエ変換 (図2)，Strassen の行列積 (図3) では，動的負荷分散方式よりも静的負荷分散方式の方が高いスピードアップを示している．分割される部分問題の計算量がほぼ均等であるため，静的負荷分散方式で並列実行した方が，マネージャ・ワーカ間のメッセージ送受信時間がない分，性能向上が高くなったと言える．

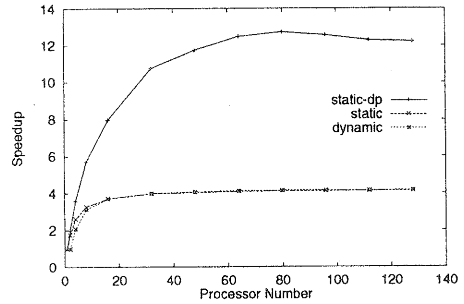

図2 高速フーリエ変換 (N=1M,L:n>4K)

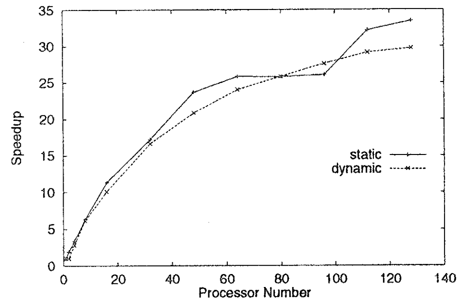

図3 Strassen の行列積 (N=1024,L:n>32)

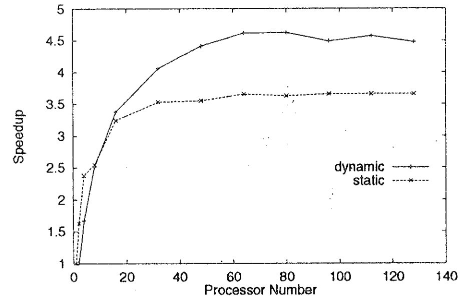

図4 クイックソート (N=4M,L:n>4K)

また，高速フーリエ変換では，並列再帰呼出前後の配列代入文が実行時間のうち大きな部分を占めるため，その負荷をプロセッサグループによるデータ並列計算で分散することにより大きな性能向上を得ることができた(static-dp)．

クイックソート (図4)，Nクイーン問題 (図5) については，静的負荷分散方式よりも動的負荷分散方式の方が高いスピードアップを示している．これは，動的負荷分散により各プロセッサ間に生じた負荷の偏りを均等化することができたためである．

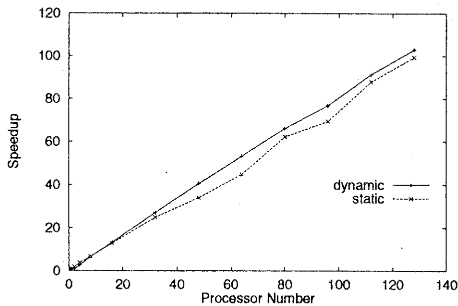

図 5 Nクイーン問題 (N=14,L:d<4)

また，Strassen の行列積，Nクイーン問題は，並列再帰呼出の分岐数がそれぞれ 7，Nと大きいため、他のアルゴリズムと比べて高いスピードアップが得られる．一般に，分岐数が大きいほど早い時間に多くのプロセッサに処理を割り当てることが可能であるため，各プロセッサが処理の割当てを待つ時間は小さくなりプロセッサ効率が向上する．

6.3.2 並列化条件による性能の変化

Nクイーン問題のプログラムについて，動的負荷分散方式における並列化条件 L:d<x(x=2...6) を変えてスピードアップを測定した結果を図 6 に示す．d は再帰呼出の深さに相当する値である．また，実行時間の内訳を全ワーカについて平均した値を表 4 に示す．

表中の，com は $IArg$ および $OArg$ の送受信時間，ctrl は M からの Order/Quit メッセージ待ち時間，sync は W_{dst} からの $OArg$ 受信までの待ち時間，req は M からの Accept/Discard メッセージ待ち時間であり，calc はその他の計算時間，total は全実行時間である．

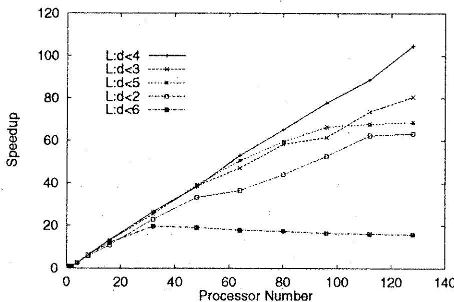

図 6 Nクイーン問題 (N=14):
並列化条件 L:d<x による性能の変化

並列化条件が性能に大きな影響を与えることがわかる．このプログラムにおける適切な並列化条件は L:d<4 である．不適切な並列化条件 L:d<6 を設定した場合，マネージャ–ワーカ間の通信が頻発しマネージャからの応答を待つ時間 (req) が極めて大きくなる．

表 4 各並列化条件 L:d<x における実行時間の内訳 (単位: 秒)

x	calc	com	ctrl	sync	req	total
2	11.2	0.00	11.1	0.22	0.00	22.5
3	11.2	0.01	5.91	0.49	0.03	17.6
4	11.2	0.04	1.38	0.84	0.12	13.6
5	11.4	0.19	3.68	0.97	4.49	20.7
6	12.0	0.75	28.0	3.71	45.5	89.9

6.3.3 複数マネージャを用いた動的負荷分散の効果

Nクイーン問題のプログラムについて，動的負荷分散方式におけるマネージャ数 m を変化させてスピードアップを測定した結果を図 7 に示す．また，プログラム実行時間の内訳を全ワーカについて平均した値を表 5 に示す．

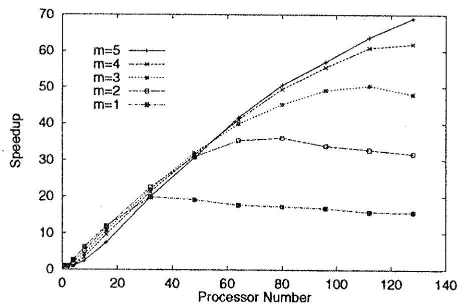

図 7 Nクイーン問題 (N=14,L:d<6):
マネージャ数 m による性能の変化

表 5 各マネージャ数 m における実行時間の内訳 (単位: 秒)

m	calc	com	ctrl	sync	req	total
1	11.9	0.77	28.9	3.78	45.5	90.9
2	11.7	0.79	13.5	2.28	16.5	44.8
3	11.7	0.77	7.26	1.88	7.99	29.6
4	11.8	0.71	4.29	1.83	4.36	22.9
5	11.8	0.67	3.34	1.81	2.94	20.6

この並列化条件 L:d<6 では，マネージャ–ワーカ間の通信頻度増大のためワーカ数増加にともなうスピードアップ向上が大きく抑えられる．そこで，複数マネージャを起動することによりマネージャの負荷を分散させ，ワーカがマネージャからの応答を待つ時間 (req) を大きく削減している．このプログラムでは，マネージャ数 5 の場合に最大のスピードアップを示している．このように，不適切な並列化条件が設定された場合にも，複数マネージャ起動により性能を大きく向上させることができる．

7. 関連研究

本研究と同じく分割統治法を並列再帰により実装した言語処理系として，PRP[8] と Machiavelli[9] を紹介する．

7.1 PRP

PRP は，標準 C 言語に独自のディレクティブを拡張した言語で分割統治法のソースプログラムを記述し，通

信ライブラリ PVM を用いた並列プログラムに変換する．生成される並列プログラムは，マネージャ・ワーカ法に基づいて動作する．PRP においては，複数のワーカで並列実行される再帰呼出の数は一定である．まず，マネージャが単独でプログラムを実行し，並列実行可能な再帰呼出を一定数になるまで生成し，タスクプールに格納する．その後，待ち状態のワーカに再帰呼出を動的に割り当てる．PRP により生成される並列プログラムは，Nクイーン問題のように並列再帰の分岐数が大きく，また扱うデータサイズが小さい場合に良好な性能を示す．しかし，クイックソートのように大量のデータを扱うプログラムはシステムの機能の制限により実装することができない．また，動的負荷分散のみしか対応していないため，部分問題の計算量が元々均等である場合にはオーバーヘッドが避けられない．

7.2 Machiavelli

Machiavelli も，分割統治法の並列実行を目標とした C 言語ツールキットであり，通信ライブラリ MPI を用いて実装されている．Machiavelli では，並列に実行する再帰呼出をプロセッサグループに割り当てる．再帰関数内で配列代入文などのデータ並列計算を指定した場合，そのプロセッサグループでデータ並列実行を行う．再帰的にグループ分割されグループサイズが 1 となると，マネージャ・ワーカ法による動的負荷分散を行う．Machiavelli では，再帰関数内にデータ並列計算可能部分があるプログラムについては良好な性能の並列プログラムを記述可能であるが，クイックソートのようにデータ並列計算の困難な部分が実行時間の大部分を占めるプログラムに対してはプロセッサグループ処理がオーバーヘッドとなる．

8. ま と め

分割統治法アルゴリズムにはそれぞれ異なる特徴があり，その特徴に応じた効率良い並列実行方式を適切に選択し並列プログラムを作成することが重要であることを示した．そして，効率良い並列実行に必要な情報を，プログラマがコンパイル時に明示的に選択・指定することにより，それぞれの分割統治法アルゴリズムに適した並列プログラムを生成するコンパイラを実装した．性能の良い並列プログラムを生成するために，プロセッサ間の負荷を均等化させる負荷分散方式として，静的負荷分散方式，動的負荷分散方式を実装し，また，過剰な並列化によるオーバーヘッドを抑制する並列化条件を導入した．最後に，特徴の異なる分割統治法アルゴリズムを提案手法に基づきコンパイルし生成された並列プログラムの性能評価実験を行い，各並列プログラムが良好な実行性能を得ることを示した．

9. 今後の課題

現段階では，並列再帰呼出を行うか否かを判断する並列化条件は，実際にプログラムを実行させることでしか適切な条件を決定することができない．この並列化条件は，再帰関数の計算量，プロセッサ間の通信データサイズ，通信頻度，ネットワーク性能に依存する[3]．これらのパラメータを利用することによりコンパイル時あるいはプログラム実行中に適切な並列化条件を決定する手法を考案することが今後の課題としてあげられる．

また現時点では，並列再帰呼出を開始する時点において 1 台のプロセッサが自メモリに全てのデータを格納しておき，再帰呼出を他プロセッサに割り当てる時点で引数データを送信するという実装をしている．このため，全データを 1 台のプロセッサのメモリに格納できなければならないという制限がある．大量のデータを複数プロセッサのメモリに分散させて格納した状態で並列再帰を実行する手法を考案することも重要な課題である．

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．並列計算機 Cenju-3 を利用させて頂いた日本電気 (株) に感謝する．

参 考 文 献

1) Joseph JáJá, "An Introduction to Parallel Algorithms", Addison-Wesley Publishing Company, Inc. (1992)
2) 岸部祥典, 小河原徹, 藤本典幸, 萩原兼一: "SPMD プログラムを生成する Work-Time C 処理系の実現", 情報処理学会研究報告, アルゴリズム 60-8, pp.57-64 (1998)
3) 小河原徹, 中島大輔, 藤本典幸, 萩原兼一: "分割統治法プログラムを並列実行するコンパイル手法の提案と評価", 情報処理学会研究報告, アルゴリズム 66-10, pp.73-80 (1998)
4) George S.Almasi, Allan Gottlieb, "Highly Parallel Computing", The Venjamin/Cummings Publishing Company, Inc. (1994)
5) 並列コンピュータ Cenju-3 本体系諸元表 http://www.ppc.nec.co.jp/heiretu/Cenju-3.html
6) Message Passing Interface Forum: "MPI: A Message-Passing Interface Standard Version 1.1", http://phase.etl.go.jp/mpif/docs/mpi-11-html/mpi-report.html (1995)
7) Bernd Freisleben, Thilo Kielmann: "Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs", Computers and Artificial Intelligence, vol.14, no.16, pp.579-596 (1995)
8) Viktor Eide: "Parallel Recursive Procedures A manager/worker approach", http://www.ifi.uio.no/arnem/PRP (1998)
9) Jonathan C.Hardwick: "Practical Parallel Divide-and-Conquer Algorithms", SCS TECHNICAL REPORT COLLECTION http://www.reports-archive.adm.cs.cmu.edu/anon/1997/abstracts/97-197.html (1997)