2004－HPC－99　(12)
2004／7／30

SMP クラスタ上での MPI と OpenMP を用いたマクロデータフロー処理

那須 弘志 †　　田邊 浩志 †　　本多 弘樹 †　　弓場 敏嗣 †

† 電気通信大学 大学院情報システム学研究科

概 要

SMP などの共有メモリシステムや PC クラスタなどの分散メモリシステム上でのプログラムの並列処理方式として，粗粒度タスク (マクロタスク) 間の並列性を利用するマクロデータフロー処理が注目されている．本稿では，SMP クラスタ上でのマクロデータフロー処理において，マクロタスク間の並列性に加えて，マクロタスク内のループイタレーション間の並列性を利用する実行方式を提案する．本方式では，SMP クラスタのノード間でマクロタスク並列処理を行い，SMP ノード内のプロセッサ間でループ並列処理を行う．提案方式を MPI と OpenMP を用いて SMP クラスタ上に実装し，その予備評価を行った．

Macro-Data-Flow using MPI and OpenMP on an SMP-cluster

Hiroshi NASU †,　　Hiroshi TANABE †,　　Hiroki HONDA †,
and Toshitsugu YUBA †

†Graduate School of Information Systems, The University of Electro-Communications.

Abstract

On shared memory systems such as SMP and distributed memory systems such as PC-cluster, macro-dataflow processing, which exploits parallelism among coarse grain tasks (macro-tasks), is considered as a promising scheme for breaking the performance limits of the loop parallelism. This paper proposes an execution scheme which exploits the loop parallelism in addition to the macro-task parallelism for macro-dataflow on an SMP-cluster. This paper discribes the implementations using MPI and OpenMP.

1　はじめに

プログラムの基本ブロックやループ，サブルーチンといった粗粒度タスク (マクロタスク) レベルの並列性を利用するマクロデータフロー処理が，SMP (Symmetrical Multi-Processor) などの共有メモリ型並列計算機上や，PC クラスタなどの分散メモリ型並列計算機上で，高い実効性能を与えることが報告されている [1, 2, 3, 4, 5].

マクロデータフロー処理とは，コンパイル時にプログラムをマクロタスク単位に分割し，マクロタスク間の並列性を実行開始条件として，またマクロタスク間でのデータ授受の関係をデータ到達条件として求め，実行時にはその 2 つの条件を検査しながら，実行開始条件が成立したマクロタスクを順次プロセッサへ割り当てたり，必要となるデータの送信元・送信先プロセッサを決定したりして，並列実行を進めるものである．

近年，SMP マシンを高速なネットワークで接続した SMP クラスタが，並列計算環境として広く普及している．SMP クラスタのアーキテクチャは，クラスタノードをタスクの処理単位とするノード間並列の階層と SMP ノード内のプロセッサをタスクの処理単位とするノード内並列の階層の 2 階層で構成される．一方，プログラムは，マクロタスク間の粗粒度レベル並列性，ループイタレーション間の中粒度レベル並列性，およびステートメントや命令間の細粒度レベル並列性の，3 種類の並列性を持つ．

SMP クラスタ上での並列処理において SMP クラスタアーキテクチャの階層性を最大限に利用するためには，ハードウェアの階層に対してプログラムの階層を効率よく写像することが重要となる．

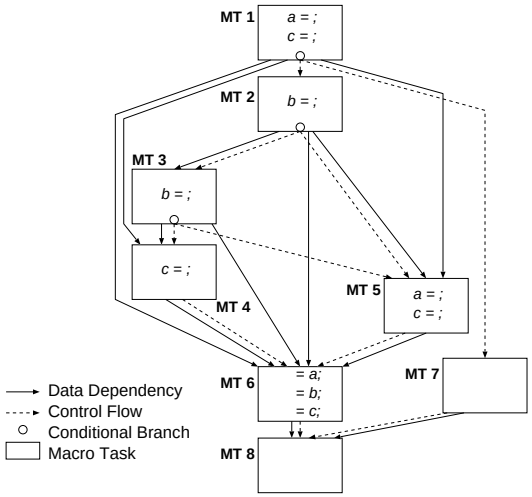

図 1: マクロフローグラフの例

SMP クラスタ上でマクロデータフロー処理を行う場合にも，SMP クラスタの階層性やマクロタスク間並列性およびマクロタスク内並列性を考慮することで，ハードウェアとプログラムの階層性を充分に引き出すことができると考えられる．

本稿では，SMP クラスタ上において，クラスタノード間でマクロタスク並列処理を行い，SMP ノード内でループ並列処理を行うことによって，マクロタスク間とマクロタスク内のループイタレーション間の 2 つのレベルでの並列処理を実現するマクロデータフロー処理の MPI と OpenMP を用いた実装方式を示す．

2 マクロデータフロー処理

2.1 マクロタスク

マクロデータフロー処理では，プログラムの基本ブロックやループ，サブルーチンなどの粒度の大きい処理をマクロタスク [6] とし，このマクロタスクをプロセッサへの割当て単位として並列に処理する．

コンパイル時には，プログラムをマクロタスクに分割し，マクロタスク間の制御フローとデータ依存を図 1 のようなマクロフローグラフで表現する．マクロタスク分割にあたっては，制御フローグラフが非循環になるように分割する．

2.2 実行開始条件

実行開始条件 [7] とは，あるマクロタスクの実行開始が可能となるための条件で，他のマクロタスクの実行状況を項とした論理式で表現したものである．この論理式は＜マクロタスク終了＞と＜分岐方向決定＞の 2 種類の原子条件および論理演算子 $\vee$ (論理和) と $\wedge$ (論理積) で構成される．

2.3 データ到達条件

分散メモリシステム上でのマクロデータフロー処理では，データ依存するマクロタスクが異なるプロセッサで実行される際に，明示的なデータ通信を必要とする場合がある．そのためにはどのマクロタスク間でどの変数に関するデータの授受を行うかが明確でなくてはならない．

しかしながら，あるマクロタスクで変数 v の参照があり，v の定義が複数のマクロタスクで行われる際，それらのうちどのマクロタスクで定義された v の値を参照するべきなのかは，一般的に実行時にならなければ決定できない．よって，どのマクロタスク間でどの変数に関する通信を行うかの判断も実行時に行うことになる．

データ到達条件 [3] とは，あるマクロタスク MTi (の先頭) に到達する [8] 変数 v の定義を行うマクロタスクの集合を S_v^i としたとき，MTi (の先頭の文) での v の値が，$MTj \subset S_v^i$ で定義した値となることが確定するための条件で，実行開始条件と同様に他のマクロタスクの実行状況を項とした論理式で表現する．

2.4 マクロタスクおよびデータ送受信のスケジューリング

マクロデータフロー処理では，プログラム実行時にスケジューラが複数のプロセッサに対して，マクロタスク割当てや，必要なデータの送信・受信指示を順次行うことで処理を進める．

スケジューラの実装には集中型ダイナミックスケジューリング方式と，分散型ダイナミックスケジューリング方式が可能である [1]．集中型では特定のプロセッサがスケジューリングコードを実行するのに対し，分散型では各プロセッサにスケジューリングコードを分散させ，全てのプロセッサがスケジューリングコードを実行する．

分散メモリシステム上でのマクロデータフロー処理におけるスケジューラの動作 [3] を以下に示す．

(1) 実行開始条件の検査： 各マクロタスクから通達される終了および分岐方向決定の情報を元に実行開始条件を検査し，条件が成立したマクロタスクをレディマクロタスクとする．

(2) マクロタスクの割当て： 所定の割当て戦略に従って，レディマクロタスク中のどのマクロタスク (MTi) をどのプロセッサ (Pi) に割り当てるかを決定する．

(3) データ到達条件の検査： MTi の実行中に参照する変数 v に関するデータ到達条件を評価し，条件が成立したマクロタスク MTj とそれが割り当てられたプロセッサ Pj を求める．

(4) 通信の割り当て： MTj が定義した v の値が既に Pi に存在する場合 (Pi と Pj が同一の場合，もしくは Pi で既に実行されたマクロタスクで参照するために v の値が Pi に通信済みの場合)

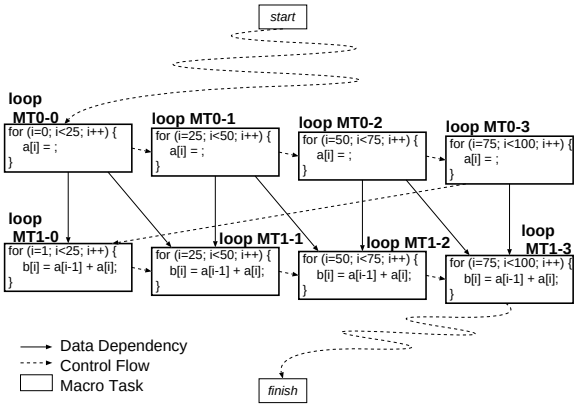

図 2: マクロタスク間とマクロタスク内の並列性

には，データ通信の指示は行わない．そうでない場合は，Pj に対して MTj で定義された v の値を Pi へ送信するよう指示し，さらに Pi に対しては MTi の実行に先立ち Pj から v の値を受信するよう指示する．

3 SMP クラスタ上でのマクロデータフロー処理

3.1 SMP クラスタの階層性とプログラムの階層性

SMP クラスタのアーキテクチャは，クラスタノード間並列性と SMP ノード内並列性の 2 階層の並列性を持つ．

マクロデータフロー処理においてプログラムをマクロタスクに分割する場合，図 2 のように並列化可能なループはプロセッサ数を考慮した数に分割し，それぞれ異なるマクロタスクとして定義する．並列化可能なループを持つプログラムには，マクロタスク間並列性と，並列化可能なループを含むマクロタスク内のループイタレーション間並列性の，2 階層の並列性が存在する．

我々はこれまでに，PC クラスタなどの分散メモリシステム上でマクロデータフロー処理を実現する方式を提案してきた [3, 4, 5]．提案方式により，クラスタノード間においてマクロタスク間並列性を利用した並列処理が可能となる．SMP クラスタ上でマクロデータフロー処理を行う場合には，クラスタノード間では従来の方式を用いたマクロタスクの並列処理を行い，SMP ノード内では並列化可能なループを含むマクロタスク内のループイタレーションの並列処理を行うことで，SMP クラスタとプログラムの両者の階層性をうまく利用することができる．

本稿では，SMP クラスタ上で，プログラム内のマクロタスク間とマクロタスク内ループイタレーション間の 2 つのレベルでの並列処理を実現するマクロデータフロー処理方式について述べる．

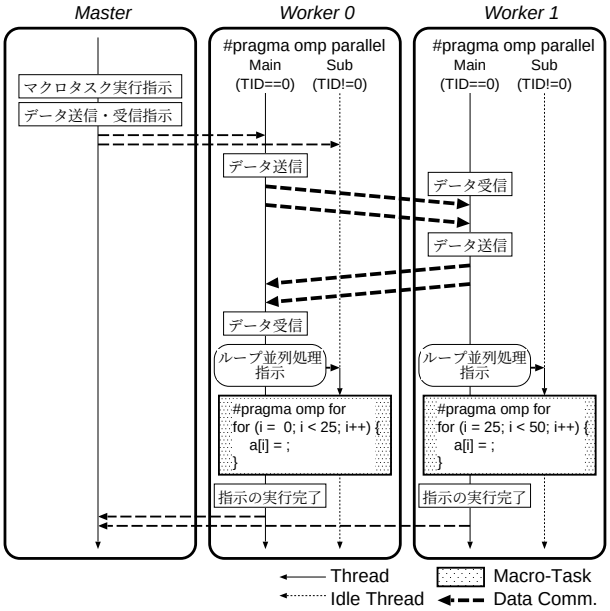

図 3: 階層性を考慮したマクロデータフロー処理

3.2 階層性を考慮したマクロデータフロー処理

図 3 に，プログラムのマクロタスク間並列性とマクロタスク内のループイタレーション間並列性の 2 つの階層を考慮したマクロデータフロー処理手法の概要を示す．

1 つのプロセス (マスタプロセス) ではマクロタスク実行およびアプリケーションデータ送受信の指示を行うスケジューラの処理を，その他のプロセス (ワーカプロセス) でデータ送受信とマクロタスクの処理を行う集中スケジューリング方式で SMP クラスタ上でのマクロデータフロー処理を行う．

ワーカプロセス内では複数のスレッドを起動し，そのうちのある 1 つのスレッド (メインスレッド) では，アプリケーションデータの送受信およびマクロタスクの処理を行う．その他の 1 つ以上のスレッド (サブスレッド) では，マスタプロセスから割り当てられたマクロタスクが並列化可能なループを含む場合に限り，メインスレッドと協調してマクロタスク内のループ並列処理を実行する．スレッドセーフを考慮するため，プロセス間通信はメインスレッドのみが行うようにする．

図 3 において，ワーカプロセス間 (メインスレッド間) ではマクロタスクの並列処理が，ワーカプロセス内のスレッド間ではループ並列処理が行われる．

3.3 MPI と OpenMP による実装方式

図 3 で示すマクロデータフロー処理手法を，MPI と OpenMP を用いて SMP クラスタ上に実装する方法について述べる．

SMP クラスタの各ノードで，図 3 のプロセスを 1 つ起動する．つまり，ある 1 ノード (マスタノード) でマスタプロセスを，その他のノード (ワーカノード) でワーカプロセスを起動する．マスタ-ワーカノード間でのスケジューラの指示やワーカノード間でのアプリケーションデータの送受信などのプロセス間通信には MPI を用いる．

図 4 にワーカノードで処理するプログラムの基本的な構成を示す．

ワーカプロセス内では OpenMP の `parallel` 指示文を用いてノード内のプロセッサ数分のスレッドを起動する．ノード内の各スレッドのうち，Thread-ID が 0 のスレッドによってメインスレッドの処理を，その他のスレッドによってサブスレッドの処理を行う．

メインスレッドは，ループ並列処理指示またはプログラム終了指示を受信するまでサブスレッドを待機させておくために，`omp_set_lock()` と `omp_unset_lock()` を用いてロックを操作することでサブスレッドを制御する．スケジューラからのループ並列処理指示またはプログラム終了指示が無い間は，メインスレッドはロックを開放せずに別の処理を行い，サブスレッドはロックが獲得できるまで処理を進めないようにする．

ワーカプロセス内のメインスレッドとサブスレッドの動作を以下に示す．

メインスレッドの動作:

1. あらかじめ `omp_set_lock()` を実行し，サブスレッドより先にロックを獲得しておく．
2. マスタプロセスからのマクロタスク実行指示およびデータ送信･受信指示を受信する．もしマスタプロセスから並列実行可能なループを含むマクロタスクの処理の指示もしくはプログラムの終了指示を受信した場合，`omp_unset_lock()` を実行しロックを解放する．ループ並列処理の指示があれば，その指示をサブスレッドに伝えてから 3 へ．プログラムの終了指示があれば，指示内容をサブスレッドに伝えてから処理を終了する．
3. 各マクロタスクの処理を行う関数 `macrotask()` を実行する．
4. 2 へ戻る．3 の `macrotask()` でループ並列処理を行った場合においては，`omp_set_lock()` を実行しロックを獲得し全サブスレッドと同期をとって 2 へ．

サブスレッドの動作:

1. `omp_set_lock()` によってロックが獲得できるまで待機する．ロックを獲得できたら，その他のサブスレッドの処理も進めさせるためにすぐに `omp_unset_lock()` を実行してそのロックを解放し，次の処理へ進む．
2. もしメインスレッドからループ並列処理の指示があれば次の処理へ進む．プログラムの終了指

```
worker(int pid)
{
  flag = TRUE;
#pragma omp parallel private(tid)
  {
    /* スレッドIDの取得 */
    tid = omp_get_thread_num();
    if (tid == 0) {
      /*
       * ***** メインスレッド(スレッドID == 0)の処理 *****
       */
      omp_init_lock(&para_exe); // ロック変数の初期化
      omp_set_lock(&para_exe); // ロックの獲得
#pragma omp barrier // バリア同期0
      while (1) {
        /* マスタプロセスからの指示の受信 */
        MPI_Recv();
        if (プログラム終了指示を受信) {
          /* 他のスレッドへの終了指示 */
          flag = FALSE;
          omp_unset_lock(&para_exe); // ロックの解放
          /* マクロデータフロー処理の終了 */
          break;
        }
        /* ***** マスタプロセスからの指示の実行 ***** */
        for (;;) {
          if (アプリケーションデータの送信・受信指示がある) {
            MPI_Send(); or MPI_Recv();
          }
          if (マクロタスクの実行指示がある) {
            if (マクロタスクがループ並列実行可能) {
              omp_unset_lock(&para_exe); // ロックの解放
#pragma omp barrier // バリア同期1
            }
            /* マクロタスク実行 */
            macrotask(mt_num, &br_num);
            if (マクロタスクがループ並列実行可能) {
              omp_set_lock(&para_exe); // ロックの獲得
#pragma omp barrier // バリア同期2
            }
          }
          if (指示の実行完了) {
            break;
          }
        } // for (;;)
        /* マスタプロセスへの処理終了の送信 */
        MPI_Send();
      } // while (1)
    } else {
      /*
       * ***** サブスレッド(スレッドID != 0)の処理 *****
       */
#pragma omp barrier // バリア同期0
      while (1) {
        omp_set_lock(&para_exe); // ロックの獲得
        /* ループ並列実行orプログラム終了時までここで待機 */
        omp_unset_lock(&para_exe); // ロックの解放
        if (flag == TRUE) {
#pragma omp barrier // バリア同期1
          /* ループマクロタスクの並列処理 */
          macrotask(mt_num, &br_num);
#pragma omp barrier // バリア同期2
        } else if (flag == FALSE) {
          /* マクロデータフロー処理の終了 */
          break;
        }
      } // while (1)
    }
  } // #pragma omp parallel
}
```

図 4: ワーカノードのソースコード

```
macrotask(int mt_num, int *br_num)
{
  /* ***** マクロタスク実行 ***** */
  switch (mt_num) {

  case START_MT:
    break;

  case BLOCK0:
    {
      x = ... ;
      y = ... ;
    }
    break;

  case LOOP0_0:
    {
#pragma omp for
      for (i = 0; i < 25; i++) {
        a[i] = ;
      }
    }
    break;

  case LOOP0_1:
    {
#pragma omp for
      for (i = 25; i < 50; i++) {
        a[i] = ;
      }
    }
    break;

  case BLOCK1:
    {
      if (condition A == TRUE) {
        *br_num = BLOCK1_BR0;
      } else {
        *br_num = BLOCK1_BR1;
      }
    }
    break;

    …   …   …   …   …   …

  case FINAL_MT:
    break;

  default:
    break;
  }
}
```

ループマクロタスク

図 5: マクロタスク処理コード

示があれば，そのまま処理を終了する．

3. 各マクロタスクの処理を行う関数 `macrotask()` を実行する．
4. メインスレッドと同期をとって 1 へ戻る．

図 5 に，各マクロタスクの処理を行う関数 `macrotask()` の構成を示す．

`macrotask()` は，1 マクロタスクを 1 つの `case` 文とした `switch-case` 構文を用いた．ダイナミックスケジューリングでは，全てのワーカノードはスケジューラによる実行時のマクロタスクの割当ての結果次第で全てのマクロタスクを実行する可能性があるため，`macrotask()` の `case` 文には全マクロタスクコードが記述されている．

ワーカノードのメインスレッドは，マスタノードからマクロタスク `mt_num` の実行指示を受信したら，`macrotask()` の `mt_num` の case 文へジャンプし，`mt_num` の処理を行う．`mt_num` が並列処理可能なループを含むマクロタスクの場合は，メインスレッドと同様にサブスレッドも `mt_num` の case 文へジャンプし，メインスレッドとサブスレッドによるループ並列処理が行われる．`macrotask()` の中の並列処理可能なループには，OpenMP の `for` 指示文 (`#pragma omp for`) によってループ並列化を指定する．

OpenMP によるループ並列処理部分においては，`schedule` 指示節の指定によるループイタレーションのスケジューリング方法の変更やチャンクサイズの調整など，マクロデータフロー処理では困難な細かいパフォーマンスチューニングが可能となる．

表 1: システム環境

Processor	PentiumIII 866MHz x 2
Memory	1 GB
NIC	Myrinet-2000
OS	RedHat Linux 7.3, SCore version 5.6.1
MPI	MPICH-SCore
OpenMP	RWCP Omni OpenMP Compiler version 1.4a
C Compiler	GNU C Compiler version 2.96

4 予備評価

提案方式を PentiumIII プロセッサを 2 つ搭載する SMP マシンを Myrinet で結合した SMP クラスタ上に C 言語により実装し，マクロタスク並列処理の性能に加えてループ並列処理の性能がどの程度得られるのかを検証した．表 1 に SMP クラスタの各ノードの構成要素を示す．

性能評価プログラムとして，SPEC fp95 の swim を用いた．swim は差分近似による浅瀬式の解を求めるプログラムである．swim のメインループ内で呼ばれるサブルーチン CALC1，CALC2，CALC3 はインライン展開した．swim のマクロタスク分割では，並列化可能な 2 重ループは，ワーカノード数分に外側ループを分割し，`#pragma omp for` によってループ並列化を指定した．プログラムのデータサイズは 2048x2048 とし，主ループ回転数を 100 とした．swim のループ並列実行可能なマクロタスクには，OpenMP の `for` 指示文の schedule 指示節の指定は行っていない．なお，プログラムのマクロタスク分割や並列性検出などは人手で行った．

表 2 に，swim での評価結果を示す．表 2 の (P,T) はそれぞれワーカプロセス数 (ワーカノード数)，ループ並列処理スレッド数 (メインスレッド数 (1) + サブスレッド数) を指す．T=1 の場合は，ワーカプロセス内ではメインスレッド 1 つのみが起動され，SMP ノード内でのループ並列処理は行われない．

表 2 において，T=2 の場合は T=1 の場合と比較して，P=2 では 1.82 倍，P=4 では 1.38 倍の OpenMP

表 2: swim の実行時間と台数効果

	実行時間	台数効果
逐次	360.94 [sec.]	1.00
mdf (P=2,T=1)	192.15 [sec.]	1.88
mdf (P=2,T=2)	105.10 [sec.]	3.43
mdf (P=4,T=1)	96.25 [sec.]	3.75
mdf (P=4,T=2)	69.62 [sec.]	5.18

ループ並列化による性能向上を得ることができた．

SMP ノード内でのメモリアクセス競合の影響により，クラスタノード間の並列効果に比べて SMP ノード内の並列効果は低い．ただ，ループ並列性の高い swim では，OpenMP によるループ並列処理部分でループイタレーションのスケジューリング方法やチャンクサイズを適度に変更することで表 2 の結果以上の大幅な性能向上を期待できる．

5 おわりに

本稿では，SMP クラスタ上でのマクロデータフロー処理において，SMP クラスタとプログラムの階層性を考慮し，マクロタスクおよびループイタレーションの両方のレベルでの並列処理を MPI と OpenMP を用いて実装する方法を示した．

本稿で示したマクロデータフロー処理の実装方式では，並列化可能なループ部分を OpenMP の `for` 指示文を用いてループ並列化を行うことで，クラスタノード間でのマクロタスク間並列性に加えて，SMP 上での OpenMP による Doall ループ並列処理の性能向上を得ることができる．

本稿で示した方式以外にも，ループ内部やサブルーチン内部でマクロタスクを階層的に生成し，他の階層間のマクロタスク間の並列性を利用する階層型マクロデータフロー処理手法 [9, 10] を SMP クラスタ向けに拡張することも考えられる．階層型マクロデータフロー処理はプログラムから多くの階層性を利用するので，本手法よりも汎用性が大きい．本手法をより発展させるための選択肢の一つとして，階層型マクロデータフロー処理手法に，本手法を追加した実行方式を提案することが挙げられる．

今後は，SMP クラスタ上での階層型マクロデータフロー処理手法や階層型マクロデータフロー処理手法への適用方法を検討していきたい．

また，本手法における OpenMP ループ並列化部分のパフォーマンスチューニングや，本稿で使用した SMP クラスタ以外の環境での実装および評価を行う予定である．

謝辞 本研究の一部は文部科学省科学研究費 (基盤 C, 2, 16500025) によって行われた．

参考文献

[1] 石坂 一久，中野 啓史，八木 哲志，小幡 元樹，笠原 博徳: “共有メモリマルチプロセッサシステム上でのキャッシュ最適化を考慮した粗粒度タスク並列処理”，情報処理学会論文誌 Vol.43 No.4，pp.958-970 (2002)．

[2] 小幡 元樹，白子 準，神長 浩気，石坂 一久，笠原 博徳: “マルチグレイン並列処理のための階層的並列性制御手法”，情報処理学会論文誌 Vol.44 No.4，pp.1044-1055 (2003)．

[3] 本多 弘樹，上田 哲平，深川 保，弓場 敏嗣: “分散メモリシステム上でのマクロデータフロー処理のためのデータ到達条件”，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム Vol. 43 No. SIG6 (HPS5)，pp.45-55 (2002)．

[4] 田邊 浩志，本多 弘樹，弓場 敏嗣: “PC クラスタ上でのマクロデータフロー処理の評価”，情報処理学会研究報告，ARC-157 HPC-097，pp.103-108 (2004)．

[5] 田邊 浩志，本多 弘樹，弓場 敏嗣: “ソフトウェア分散共有メモリを用いたマクロデータフロー処理”，先進的計算基盤シンポジウム SACSIS2004 論文集，pp.35-42 (2004)．

[6] 笠原 博徳，合田 憲人，吉田 明正，岡本 雅巳，本多 弘樹: “Fortran マクロデータフロー処理のマクロタスク生成手法”，電子情報通信学会論文誌 Vol.J75-D-I No.8，pp.511-525 (1992)．

[7] 本多 弘樹，岩田 雅彦，笠原 博徳: “Fortran プログラム粗粒度タスク間の並列性検出手法”，電子情報通信学会論文誌 Vol.J75-D-I No.12，pp.951-960 (1990)．

[8] Alfred V. Aho，Ravi Sethi，Jeffrey D. Ullman: “Compilers - Principles, Techniques, and Tools”，Addison-Wesley (1988)．

[9] 岡本 雅巳，合田 憲人，宮沢 稔，本多 弘樹，笠原 博徳: “OSCAR マルチグレインコンパイラにおける階層型マクロデータフロー処理”，情報処理学会論文誌 Vol.35 No.4，pp.513-521 (1994)．

[10] 吉田 明正: “階層型マクロタスクグラフのための異階層タスクの統合実行制御手法”，情報処理学会研究報告，ARC-154，pp.73-78 (2003)．