共有メモリ型マルチプロセッサシステム上での Fortran 粗粒度タスク並列処理の性能評価

合田 憲人[†] 岩崎 清[††] 岡本 雅巳[†]
笠原 博徳[†] 成田 誠之助[†]

本論文は，共有メモリ型マルチプロセッサシステム上での粗粒度並列処理手法である Fortran マクロデータフロー処理の実現方法と性能評価について述べる．マクロデータフロー処理手法では，コンパイラが，プログラムの粗粒度タスク（マクロタスク）への分割，マクロタスク間の並列性抽出，マクロタスクをプロセッサへ割り当てるダイナミックスケジューリングコード生成，を自動的に行う．従来よりマルチプロセッサシステム上で用いられているマルチタスキング等の粗粒度並列処理では，ユーザによる粗粒度タスク間の並列性抽出が困難である，OS 等によるダイナミックスケジューリングオーバーヘッドが大きい，という問題があるが，本マクロデータフロー処理では，コンパイラが自動的にマクロタスク間の並列性を抽出するとともに，各ソースプログラム用に最適化したダイナミックスケジューリングコードを生成するためオーバーヘッドを低く抑えることが可能である．本手法の性能評価を Alliant FX/4 および Kendall Square Research KSR1 上で行った結果，マクロデータフロー処理がプログラムの実行時間を，ループ並列化，マルチタスキング等の従来手法を適用した場合の 1/1.92 から 1/8.10 に短縮することが確認された．

Performance Evaluation of Fortran Coarse Grain Parallel Processing on Shared Memory Multi-processor Systems

KENTO AIDA,[†] KIYOSHI IWASAKI,[††] MASAMI OKAMOTO,[†]
HIRONORI KASAHARA[†] and SEINOSUKE NARITA[†]

This paper presents an implementation and performance evaluation of the macrodataflow computation scheme that is a coarse grain parallel processing scheme on shared memory multi-processor systems. In macrodataflow computation, the compiler automatically generates coarse grain tasks called macrotasks, exploits parallelism among macrotasks and generates a dynamic scheduling routine to assign macrotasks to processors. A conventional coarse grain parallel processing scheme such as multi-tasking has drawbacks such as difficulty in the extraction of parallelism among coarse grain tasks by users and large dynamic scheduling overhead caused by OS calls. However, in macrodataflow computation, the compiler exploits parallelism among macrotasks automatically and dynamic scheduling overhead is small because the compiler optimizes a dynamic scheduling routine for each source program. Performance evaluation of the macrodataflow computation scheme on an Alliant FX/4 and a Kendall Square Research KSR1 shows that macrodataflow computation reduces execution time of programs to 1/1.92 - 1/8.10 of execution time by conventional schemes such as loop parallelization and multi-tasking.

1. はじめに

CRAY C90, FUJITSU VP2400, NEC SX-3 等の共有メモリ型マルチプロセッサシステム上での Fortran プログラムの並列処理では，サブルーチン等の粗粒度タスク間の並列処理手法であるマルチタスキングやループ並列処理手法であるマイクロタスキングが用いられている[1)~5)]．

マイクロタスキングは，従来から最も広く用いられているループ（中粒度タスク）並列処理[6)~8)]であり，最近ではデータ依存解析[6),9)]やプログラムリストラクチャリング技術の進歩[8)~10)]により，多くのループの並列化が可能となっている．しかし，ループキャリィドデータ依存やループ外への条件分岐等により並列化できないシーケンシャルループが依然存在する．またこのマイクロタスキングでは，本質的にループ以外の

† 早稲田大学理工学部
School of Science and Engineering, Waseda University
†† 大日本印刷株式会社
Dai Nippon Printing Co., Ltd.

部分の並列性を有効利用することはできない．

マルチタスキングは，サブルーチン等の粗粒度タスク間の並列処理手法であり，ユーザがソースプログラム中にコンパイラディレクティブを挿入することによってサブルーチン等の粗粒度タスク間の並列実行を指示し，OS またはランタイムライブラリが実行時に粗粒度タスクをプロセッサへスケジューリングする方式がとられている．しかしこの方式は，

(1) 一般ユーザによる粗粒度タスク間の並列性指定は非常に困難であり，一部のエキスパートユーザしか効果的な粗粒度並列性を抽出することができない．
(2) OS またはライブラリコールによるダイナミックスケジューリングオーバーヘッドが大きいため，処理時間の大きな粗粒度タスク間でしか良好な並列処理効果を得られない．

といった問題がある．

著者らは以上のような問題を解決するため，Fortran プログラム上のサブルーチン，ループ，基本ブロック間の粗粒度並列処理手法であるマクロデータフロー処理手法[11)~17)]を提案している．本手法では，コンパイラが以下の手順でプログラムの粗粒度並列化を行う．

(1) プログラムを粗粒度タスク（マクロタスク）へ分割
(2) マクロタスク間の制御依存，データ依存関係解析によりマクロタスク間の並列性を最早実行可能条件[11)]の形で抽出
(3) 実行時にマクロタスクをプロセッサへ割り当てるためのスケジューリングコード生成

以上のように，本手法ではコンパイラが自動的にマクロタスク間の並列性を抽出するため，効果的なプログラムの粗粒度並列性を抽出することができる．また，上述のようにOS 等によるダイナミックスケジューリングのオーバーヘッドは大きいが，本手法ではコンパイラがユーザコードの一部として最適化したスケジューリングコードを自動生成するため，スケジューリングオーバーヘッドを低く抑えることが可能である．

本論文では，共有メモリ型マルチプロセッサシステム上での従来の並列処理手法の持つ問題点を解決し，効果的な並列処理を実現するために，集中共有メモリ型マルチプロセッサシステムである Alliant FX/4 および分散共有キャッシュメモリ型マルチプロセッサシステムである Kendall Square Research KSR1 等の実マシン上でマクロデータフロー処理を実現し，性能評価を行う．

以後，2 章ではマクロデータフロー処理について概説する．3 章では共有メモリ型マルチプロセッサシステム上でのマクロデータフロー処理の実現方法について述べる．4 章ではマクロデータフロー処理の性能評価について述べる．

2. マクロデータフロー処理手法

本章では，Fortran プログラムのマクロデータフロー処理手法について概説する．マクロデータフロー処理は，サブルーチン，ループ，基本ブロック等の粗粒度タスク間の並列処理手法である．

2.1 プログラム分割

マクロデータフロー処理では，はじめに Fortran プログラムをマクロタスクと呼ぶ粗い粒度を持つタスクに分割する．マクロタスクは，基本ブロックあるいは複数の基本ブロックからなるブロック（BPA），繰り返しブロック（RB），サブルーチンブロック（SB）から構成される．

BPA は，基本的に単一の基本ブロックから構成される．ただし，並列処理効率向上のためのコンパイル時の最適化により，単一基本ブロックの分割または複数基本ブロックの融合が行われた場合，分割または融合後生成されたブロックをそれぞれ BPA と定義する[13)]．

RBは，DO ループまたはバックワードブランチによって生成されるループであり，より厳密には最外側ナチュラルループを意味する．ここで，RBが DO-all ループである場合，DO-all ループ内の並列性を抽出するために，DO-all ループをその処理時間とスケジューリングオーバーヘッドを考慮して k 個の小ループに分割する．ここで，k の値は，使用するマルチプロセッサシステムのプロセッサ数またはプロセッサ数の倍数とする[13)]．

サブルーチンに関しては，基本的に可能な限りインライン展開を行うが，コード長等を考慮してインライン展開が効果的でない場合は，そのサブルーチンを 1 つのマクロタスク（SB）として定義する．ただし，SBとほかのマクロタスク間の並列性を最大限に抽出するためには，強力なインタープロシージャ解析[18)]が必要となる．

RBおよびSBに関しては，**図 1** に示すように RB およびSB内部で階層的にマクロタスク（サブマクロタスク）を生成する．このようなサブマクロタスク間で粗粒度並列性が得られる場合は，階層型マクロデータフロー処理手法[17)]を適用する．

2.2 マクロタスク間並列性抽出

プログラム分割終了後，マクロタスク間の制御フ

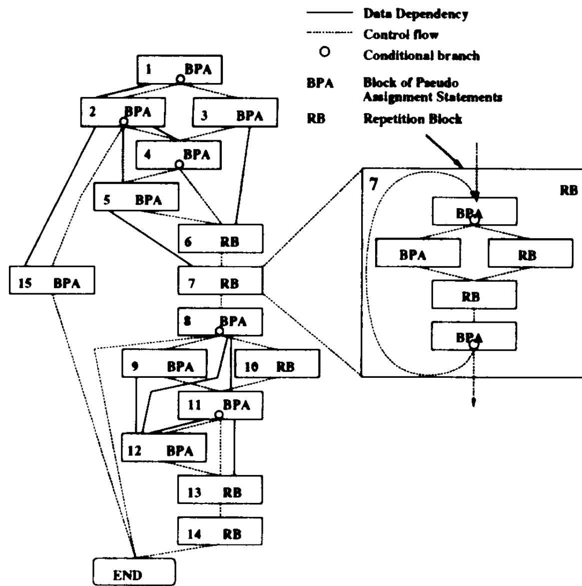


図1 マクロフローグラフ
Fig. 1 Macroflow graph.

ロー，データ依存を解析することにより，図1のようなマクロフローグラフを生成する．図1において，各ノードはマクロタスクを表し，ノード間の点線エッジ，実線エッジはそれぞれコントロールフロー，データ依存を表す．また，ノード内の小円はそのマクロタスク中に存在するほかのマクロタスクへの条件分岐文を表している．

次にマクロフローグラフから，マクロタスク間の制御依存，データ依存関係を解析することにより，各マクロタスクの最早実行可能条件[11]を求め，それをグラフ表現したマクロタスクグラフを生成する．**表1**に図1のマクロフローグラフを持つプログラムの最早実行可能条件を示し，**図2**にそのマクロタスクグラフを示す．図2において，各ノードはマクロフローグラフ同様，マクロタスクを表し，ノード内の小円は条件分岐文を表す．ノード間の点線エッジ，実線エッジは，それぞれ拡張制御依存，データ依存を表す．また，ノード内の小円を起点とするデータ依存エッジつまり実線エッジは，制御依存とデータ依存の2つを同時に表している．ノードのエッジ入力部，出力部につけられた点線弧，実線弧は，それぞれ弧に接続されたエッジの論理和，論理積をとること意味しており，各マクロタスクの最早実行可能条件は，ノードの入力エッジであるデータ依存，拡張制御依存を満足する条件の論理式によって表現される．たとえば表1中のマクロタスク6の最早実行可能条件は，[マクロタスク3の実行終了]∨[マクロタスク2からマクロタスク4への分岐発生]であるが，これは図2上では，マクロタスク6の入力部につけられた点線弧によって接続されるマクロタスク3からの実線エッジおよびマクロタスク2からの点線エッジによって表される．

表1 最早実行可能条件
Table 1 Earliest executable conditions.

マクロタスク番号	最早実行可能条件
1	
2	1_2
3	$(1)_3$
4	$2_4 \vee (1)_3$
5	$(4)_5 \wedge (2_4 \vee (1)_3)$
6	$3 \vee (2)_4$
7	$5 \vee (4)_6$
8	$(2)_4 \vee (1)_3$
9	$(8)_9$
10	$(8)_{10}$
11	$8_9 \vee 8_{10}$
12	$11_{12} \wedge (9 \vee (8)_{10})$
13	$11_{13} \vee 11_{12}$
14	$(8)_9 \vee (8)_{10}$
15	2_{15}

i: MT_i の実行が終了する
$(i)_j$: MT_i が MT_j に分岐する
i_j: MT_i が MT_j に分岐し MT_i の実行が終了する

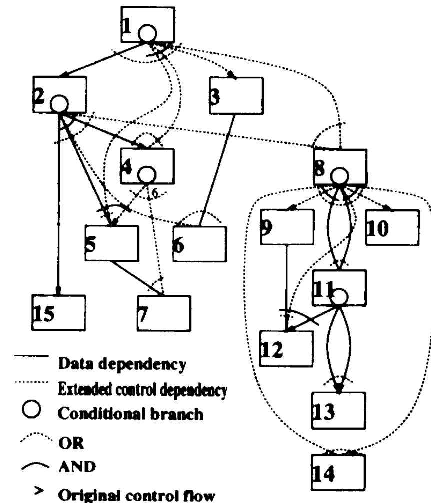


図2 マクロタスクグラフ
Fig. 2 Macrotask graph.

2.3 スケジューリングコード生成

各マクロタスクは，コンパイラの自動生成するスケジューリングコードによって，条件分岐や処理時間の変動に対処するために，Dynamic-CP 法を用いて実行時にプロセッサに割り当てられる[14]．

スケジューリングコード生成では，プログラムの並列性やマルチプロセッサシステムのアーキテクチャを考慮して，Fortran ソースプログラム用に最適化されたコードを生成する．この際，スケジューリング方式とし

て，スケジューリングコードの実行を全プロセッサに分散させる分散スケジューラ方式と単一プロセッサに集中させる集中スケジューラ方式[14]とを用いることができる．分散スケジューラ方式では，Auto-scheduling[7]で行われている方式と同様に，スケジューリングコードはユーザプログラムの一部として各マクロタスクの前後に挿入され，各プロセッサがマクロタスクを実行する度にスケジューリング処理を行う．集中スケジューラ方式では，スケジューリングコードとユーザプログラムコードが別々に生成され，単一プロセッサがスケジューリング処理を行い，ほかのプロセッサがマクロタスクを実行する．

3. 共有メモリ型マルチプロセッサシステム上でのマクロデータフロー処理実現方法

本章では，共有メモリ型マルチプロセッサシステム上でのマクロデータフロー処理の実現方法について述べる．

3.1 分散スケジューラの実現方式

分散スケジューラ方式では，コンパイラは，スケジューリングコードを各プロセッサ用のプログラムコード中すなわち各マクロタスクの前後に埋め込み，各プロセッサはこのスケジューリングコードを埋め込まれた同一のプログラムコードを実行する．各プロセッサは，全プロセッサ間で共有されるスケジューリング用データを排他アクセスしながら，マクロタスクの最早実行可能条件検査，マクロタスクの実行を行う．全プロセッサ間で共有されるマクロタスクのスケジューリング用データとしては，以下の3種類が用意される．

(1) MT_State[マクロタスク番号]：その番号のマクロタスクの状態が登録される．ここでマクロタスクの状態とは，未実行可能，実行可能，実行終了等である．

(2) MT_Branch[マクロタスク番号]：その番号のマクロタスク中で評価される分岐方向が登録される．

(3) ReadyMT_queue：実行可能マクロタスクを登録するキュー．

分散スケジューラ方式においてコンパイラが生成するコードの基本構成を**図3**に示す．図3に示すように，コードはマクロタスク割当てルーチンとマクロタスク処理ルーチンに分けられる．マクロタスク割当てルーチンでは，実行可能マクロタスクの情報をReadyMT_queueから取り出し，取り出したマクロタスクの情報に対応するマクロタスク処理ルーチンを起動する．マクロタスク処理ルーチンでは，マクロタスク本体の処理を行うとともに，マクロタスク中で評価される分岐方向，マクロタスク実行終了等の条件に応じて共有されるスケジューリング用データを更新する．また，マクロタスク本体の処理終了後，未実行可能マクロタスクの最早実行可能条件検査を行い，実行可能マクロタスクをReadyMT_queueに登録する．またプロセッサ0は，プログラムコード（全プロセッサが同一のコードを持つ）に加えて初期化処理のためのコードを実行する．初期化処理では，スケジューリング用データの初期化，プログラム実行開始時点での実行可能マクロタスクのReadyMT_queueへの登録を行う．

3.2 集中スケジューラの実現方式

集中スケジューラ方式では，コンパイラがスケジューリングコードとプログラムコードをそれぞれ別に生成し，1台のプロセッサがスケジューリングコード，他のプロセッサがプログラムコードを実行する．以後，スケジューリングコードを実行するプロセッサをスケジューラと呼ぶ．

スケジューラは，マクロタスクの最早実行可能条件を管理するとともに，マクロタスクのプロセッサへの割り当てを行う．プロセッサは，割り当てられたマクロタスクを実行するとともに，実行中のマクロタスクの分岐方向，実行終了等の情報をスケジューラへ通知する．

集中スケジューラ方式では，スケジューリング用データとして以下の2種類がスケジューラのローカル変数として用意される．

(1) MT_Condition[state 番号]：マクロタスク内で評価される分岐方向，マクロタスク実行終了等の状態をstate番号の形で管理する．マクロタスクの最早実行可能条件検査はこの変数を用いて行われる．

(2) ReadyMT_queue：実行可能マクロタスクが登録されるキュー．

ここで用いられるstate番号は，マクロタスクの分岐方向，実行終了等に一意につけられた番号である．

また，スケジューラと各プロセッサ間の通信用データとして，以下の3種類の変数が用意される．

(1) PE_State[プロセッサ番号]：スケジューラがプロセッサへ割り当てるマクロタスクの番号を登録する．スケジューラがプロセッサへマクロタスクを割り当てるために用いる．

(2) Branch[プロセッサ番号]：プロセッサが実行中のマクロタスク内で評価される分岐方向を示すstate番号を登録する．プロセッサがスケジューラへマクロタスク内で評価された分岐方向を通

```
#define MTnum MT(マクロタスク)数
#define PEnum PE(プロセッサエレメント)数

int MT_State[MTnum];            /* スケジューリング用共有変数 */
int MT_Branch[MTnum];           /* スケジューリング用共有変数 */
int ReadyMT_queue[MTnum];       /* スケジューリング用共有変数 */
int flag_queue, flag_insert; /* 排他制御用フラグ */

DISTRIBUTED_SCHEDULING(id) /* マクロタスク割り当てルーチン */
  int    id;                      /* PE番号 */
{
  int    newMT;                   /* 割り当て対象MT */

  if(id == 0){ /* PE0のみが実行する */
    スケジューリング用変数の初期化;
    実行可能MTのReadyMT_queue登録;
  }

  while(!プログラム終了){
    排他制御変数ロック(flag_queue);
    newMT = GET_MT_FROM_QUEUE; /* ReadyMT_queueからのMT取り出し */
    排他制御変数ロック解除(flag_queue);
    switch(newMT){
    case 1:  MT_1();    break;
                 :
    case MT番号:  MT_MT番号();    break;
        :
    case MT_num:  MT_MTnum();  break;
    }
  }
}

MT_1() /* マクロタスク処理ルーチン */
  :
MT_MT番号() /* マクロタスク処理ルーチン */
{
  MT本体の処理;
  if(分岐発生)
    MT_Branch[MT番号] = 分岐方向;
  MT本体の処理;
  MT_State[MT番号] = 実行終了;
  MT番号に応じたMT最早実行可能条件検査(MT_State,MT_Branch);
  if(実行可能MTが存在){
    排他制御変数ロック(flag_insert);
    実行可能MTのReadyMT_queue登録;
    排他制御変数ロック解除(flag_insert);
  }
}
  :
MT_MTnum() /* マクロタスク処理ルーチン */
```

図3 分散スケジューラ方式における生成コード
Fig. 3 Generated code for a processor in the distributed scheduling method.

知するために用いる.

(3) Finish[プロセッサ番号]：プロセッサが実行中のマクロタスクの実行終了を示す state 番号を登録する．プロセッサがスケジューラへマクロタスク実行終了を通知するために用いる.

分散スケジューラ方式では，マクロタスクのスケジューリング用データを全プロセッサ間で共有するのに対し，集中スケジューラ方式では，マクロタスクのスケジューリング用データはスケジューラのみが管理し，通信用データを用いてプロセッサへのマクロタスク割当てやプロセッサからのマクロタスクの分岐方向，実行終了等の情報収集を行う.

集中スケジューラ方式においてコンパイラが生成するコードの基本構成を図**4**，図**5** に示す．図 4 に示すように，スケジューラ用コードは，マクロタスク割当て処理とマクロタスク最早実行可能条件検査に分けられる．マクロタスク割当て処理では，PE_State[プロセッサ番号] をポーリングしアイドルプロセッサに実行可能マクロタスクを割り当てる．マクロタスク最早実行可能条件検査では，マクロタスク実行中のプロセッサから通知された state 番号に応じて未実行可能マクロタスクの最早実行可能条件検査を行い，実行可能マクロタスクを ReadyMT_queue に登録する．また，図 5 に示すように，プロセッサ用コードでは，スケジューラから割り当てられたマクロタスク本体の処理と実行中のマクロタスクの分岐方向，実行終了等の情報をスケジューラへ通知する処理から構成される.

4. 性能評価

本章では，Alliant FX/4 および Kendall Square Research KSR1 上でのマクロデータフロー処理の性能評価について述べる.

4.1 マルチプロセッサシステムのアーキテクチャ

本節では，性能評価に用いた FX/4 と KSR1 のアーキテクチャについて述べる.

4.1.1 Alliant FX/4

FX/4 は，図 **6** に示すように，ベクトル，スカラユニットを持つコンピューテーショナルエレメント (CE) 4 台を 256 KB キャッシュ，メモリバスを介して 32 MB の共有メモリに接続した構成であり，さらに各

```
#define MTnum MT(マクロタスク)数
#define PEnum PE(プロセッサエレメント)数
#define STATEnum MTのstate番号数

int     PE_State[PEnum];          /* スケジューラ・PE間通信用変数 */
int     Branch[PEnum];            /* スケジューラ・PE間通信用変数 */
int     Finish[PEnum];            /* スケジューラ・PE間通信用変数 */

SCHEDULER()
{
  int    MT_Condition[STATEnum]; /* MT最早実行可能条件管理変数 */
  int    ReadyMT_queue[MTnum];   /* 実行可能MTが登録されるキュー */
  int    newMT;                  /* 割り当て対象MT */
  int    StateNo;                /* state番号 */
  int    i;

  スケジューリング用変数の初期化;
  実行可能MTのReadyMT_queue登録;

  while(!プログラム終了){
    /* マクロタスク割り当て処理 */
    for(i = 0; i < PEnum; i++)
      if(PE_State[i] == 0) {
        newMT = GET_MT_FROM_QUEUE; /* ReadyMT_queueからのMT取り出し */
        PE_State[i] = newMT
      }
    /* マクロタスク最早実行可能条件検査 */
    for(i = 0; i < PEnum; i++){
      if(Branch[i] != 0){
        StateNo = Branch[i];
        StateNoに応じたMT最早実行可能条件検査(StateNo,MT_Condition);
        if(実行可能MTが存在)
          実行可能MTのReadyMT_queue登録;
        Branch[i] = 0;
      }
      if(Finish[i] != 0){
        StateNo = Finish[i];
        StateNoに応じたMT最早実行可能条件検査(StateNo,MT_Condition);
        if(実行可能MTが存在)
          実行可能MTのReadyMT_queue登録;
        Finish[i] = 0;
      }
    }
  }
}
```

図4 集中スケジューラ方式におけるスケジューラ用生成コード

Fig. 4 Generated code for a scheduler in the centralized scheduling method.

```
#define MTnum MT(マクロタスク)数
#define PEnum PE(プロセッサエレメント)数

int     PE_State[PEnum];          /* スケジューラ・PE間通信用変数 */
int     Branch[PEnum];            /* スケジューラ・PE間通信用変数 */
int     Finish[PEnum];            /* スケジューラ・PE間通信用変数 */

PROCESSOR(id)
  int    id;                      /* PE番号 */
{
  while(!プログラム終了){
    while(PE_State[id] != 0){
      switch(PE_State[id]){
      case 1:  MT_1(id);   break;
          :
      case MT番号:  MT_番号(id);   break;
        :
      case MT_num:  MT_MTnum(id);  break;
      }
    PE_State[id] = 0;
    }
  }
}

MT_1(id)
  :
MT_MT番号(id)
  int    id;                      /* PE番号 */
{
  MT本体の処理;
  if(分岐発生) Branch[id] = 分岐に対応したstate番号;
  MT本体の処理;
  Finish[id] = 終了に対応したstate番号;
}
  :
MT_MTnum(id)
```

図5 集中スケジューラ方式におけるプロセッサ用生成コード

Fig. 5 Generated code for an ordinary processor in the centralized scheduling method.

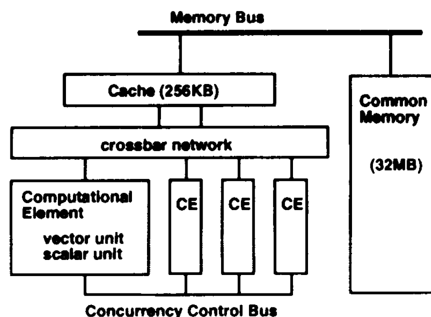


図 6 Alliant FX/4 のアーキテクチャ
Fig. 6 Architecture of an Alliant FX/4.

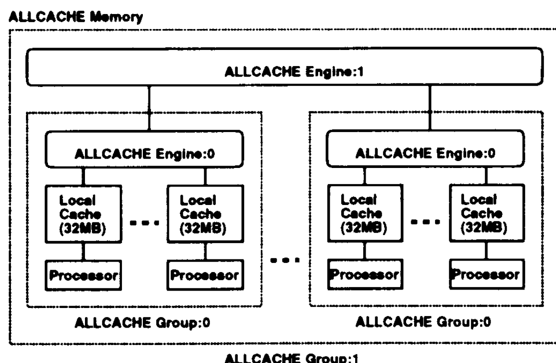


図 7 Kendall Square Research KSR1 のアーキテクチャ
Fig. 7 Architecture of a Kendall Square Research KSR1.

CE はコンカレンシーコントロールバスにより接続されている．1 CPU あたりのピーク性能は，10 MIPS, 11.8 MFLOPS である[19]．

4.1.2 Kendall Square Research KSR1

KSR1 は，**図 7** に示すように，ローカルキャッシュを持つプロセッサを階層的なリング状ネットワーク（ALLCACHE Engine）により接続した構成である．階層構造の最下層では，32 MB のローカルキャッシュを持つプロセッサ 32 台がリング状ネットワーク（ALLCACHE Engine:0）に接続され，ALLCACHE Group:0 を構成している．さらに，34 個の ALLCACHE Group:0 が上位層のリング状ネットワーク（ALLCACHE Engine:1）に接続され，ALLCACHE Group:1 が構成されており，全体として階層的なネットワークを構成している．各プロセッサは，ALLCACHE Engine によって全プロセッサのローカルキャッシュ上のデータへのアクセスが可能であり，論理的には全プロセッサ上のローカルキャッシュを共有メモリとして扱うことができる．1 CPU あたりのピーク性能は，40 MIPS, 40 MFLOPS である[20),21]．

4.2 マクロデータフロー処理の性能評価結果

本節では，FX/4 および KSR1 上でのマクロデータフロー処理の性能評価結果について述べる．

本節ではまずはじめに，ループ自動並列化プリプロセッサである KAP[20),21] によるループ並列化，マルチスレッディング，マクロデータフロー処理の性能比較について述べる．ここでマルチスレッディングは，並列プロセス生成やメモリ領域確保に要する時間を短縮するために，軽量プロセス（スレッド）を用いて粗粒度タスク並列処理を行う手法である．KSR1 は KAP およびマルチスレッディング環境[20),21] を提供しているため，本評価は KSR1 上で行う．

次に，小規模アプリケーションプログラムに対する，FX/4 および KSR1 上でのマルチタスキングまたはマルチスレッディング，マクロデータフロー処理の性能比較について述べ，最後に，マルチタスキング，マルチスレッディングおよびマクロデータフロー処理のスケジューリングオーバーヘッドの比較について述べる．

4.2.1 KSR1 上でのループ並列化，マルチスレッディング，マクロデータフロー処理の性能評価

図 8 のプログラムは，2 個の DO-all ループ，4 個のシーケンシャルループ，3 個の *BPA* から構成されるループ並列化が有効でないプログラムの例である．

表 2 に，図 8 のプログラムを KAP を用いたループ並列処理，マルチスレッディング，マクロデータフロー処理を用いて，KSR1 の 4 プロセッサ上で実行した結果を示す．

まず KAP を用いたループ並列処理では，KAP の自動並列化機能により 2 個の DO-all ループ（MT2, MT4）内の並列性が抽出され，これらのループに対し，ループ並列処理を適用した．次にマルチスレッディングでは，2 個の DO-all ループ（MT2, MT4）内の並列性を利用するため，これらの DO-all ループをそれぞれ使用するプロセッサ数分，すなわち 4 個の小ループに分割し，まず MT2 から分割された小ループをマルチスレッディングライブラリを用いて並列実行した．次に MT4 から分割された小ループについても同様に並列実行した．また，MT6 および MT7 のシーケンシャルループは並列実行可能なのでこれらをマルチスレッディングライブラリを用いて並列実行した．マクロデータフロー処理では，まず 2 個の DO-all ループ（MT2, MT4）それぞれを使用するプロセッサ数分，すなわち 4 個の小ループに分割し，DO ループ内部の並列性を抽出した．また，マクロタスク間の最早実行可能条件解析により，4 個のシーケンシャルループ

```
      REAL X1(400000),Y1(400000),Z1(400000)
      REAL A(100)
      REAL D(4000)
      INTEGER B(500000),R(500000)
      INTEGER Q(480000),P(480000),C(480000)
      REAL AJ4,X,DP
C
C     ----- MT1 BPA -----
      X1(1)=0.1
      Y1(2)=0.2
      Z1(3)=0.3
      X = 0.0
      A(1)=1.0
      B(1)=1
      B(2)=2
      C(1)=0
      D(1)=1.0
      DP = 0.0
C
C     ----- MT2 DOALL LOOP -----
      DO 10 I0=1,480000
        C(I0)=I0*2
        Q(I0)=480001-I0
        P(I0)=Q(I0)*I0/480000
   10 CONTINUE
C
C     ----- MT3 SEQUENTIAL LOOP -----
      DO 20 I1=2,400000-2,1
         X1(I1)=I1+X1(I1-1)
         Y1(I1+1)=(I1+1)+Y1(I1)
         Z1(I1+2)=(I1+2)+Z1(I1+1)
   20 CONTINUE
C
C     ----- MT4 DOALL LOOP (REDUCTION LOOP) ----
      DO 30 I2=1,480000
         DP = DP+P(I2)*Q(I2)
   30 CONTINUE
C
C     ----- MT5 BPA -----
      IF(DP.LT.0.0) GOTO 100
C
C     ----- MT6 SEQUENTIAL LOOP -----
      DO 40 I3=1,450000
        X=C(P(I3))
        C(Q(I3))=X+Q(I3)
   40 CONTINUE
C
C     ----- MT7 SEQUENTIAL LOOP -----
      DO 50 I4=3,500000
        R(I4)=B(I4-1)+B(I4-2)
        B(I4)=R(I4)/2+1
   50 CONTINUE
C
C     ----- MT8 SEQUENTIAL LOOP -----
  100 DO 60 J4=2,4000
        AJ4=J4
        DO 61 I5=2,100
           A(I5)=(A(I5-1)+AJ4/A(I5-1))/2.0
           IF (ABS(A(I5)-A(I5-1)).GT.10000) GO TO 62
   61   CONTINUE
   62   D(J4)=AJ4*D(J4-1)*0.0001+AJ4
   60 CONTINUE
C
C     ----- MT9 BPA -----
      WRITE(6,*) (X1(II),II=1,5)
      WRITE(6,*) (A(II),II=1,5)
      WRITE(6,*) (B(II),II=1,5)
      WRITE(6,*) (C(II),II=1,5)
      WRITE(6,*) (D(II),II=1,5)
C
      END
```

図 8 例題 Fortran プログラム
Fig. 8 An example Fortran program.

(MT3, MT6, MT7, MT8) 間の並列性を抽出した．またマクロデータフロー処理における各マクロタスクの実行制御は，プログラムコード中に埋め込まれたスケジューリングコードによって行った．

表 2 に示すように，KAP を用いたループ並列処理では実行時間がシーケンシャル実行時の 1/1.37 に短縮されているのに対し，マルチスレッディングでは 1/1.64 に短縮されている．これは，マルチスレッディングでは，2 個の DO-all ループ（MT2, MT4）内の並列性に加えて，2 個のシーケンシャルループ（MT6, MT7）間の並列性が抽出されているためである．これに対してマクロデータフロー処理では，さらに実行時間をシーケンシャル実行時の 1/3.15 に短縮できている．これは，マクロデータフロー処理では 2 個の DO-all ループ内の並列性に加えて，マクロタスクの最早実行可能条件解析により，4 個のシーケンシャルループ（MT3, MT6, MT7, MT8）間の並列性を抽出できること，またマルチスレッディングに比べて，プログラム中に埋め込まれたスケジューリングコードの実行はオーバーヘッドが小さいためである．

表 2 KSR1 上でのループ並列化，マルチスレッディング，マクロデータフロー処理の性能比較
Table 2 Comparison of loop parallelization, multi-threading and macrodataflow computation on a KSR1.

	実行時間 [s]	速度比
Seq	8.39	1.00
DO-all	6.11	1.37
M-thread	5.13	1.64
MDF	2.66	3.15

Seq: シーケンシャル実行
DO-all: ループ並列処理（KAP）
M-thread: マルチスレッディング
MDF: マクロデータフロー処理

4.2.2 FX/4 および KSR1 上での CG プログラムに対する性能評価

図 9 に対称帯マトリクス係数行列を持つ連立方程式求解のための CG プログラムのマクロタスクグラフを示す．図 9 中，マクロタスク 14（MT14）は収束ループで，プログラムの処理時間のほとんどはこのマクロタスク（ループ）の処理時間であるため，このマクロタスクの高速処理が全体の処理時間短縮のために重要である．また MT14 は，点線で囲まれた部分に示すように，内部で階層的にサブマクロタスクを生成できる．以上を考慮して本 CG プログラムでは，MT14 内部のサブマクロタスク集合に対して階層的にマクロデータフロー処理を適用する[17]．また，KSR1 は分散共有キャッシュメモリ型マルチプロセッサシステムであるため，KSR1 上でのマクロデータフロー処理では，データローカライゼーション手法[15),16)]を適用することにより，プロセッサ間データ転送オーバーヘッドを軽減する．

マルチタスキングおよびマルチスレッディングでは，図 9 中，MT14 内の DO-all ループ（MT14-1, MT14-3, MT14-5, MT14-7, MT14-9, MT14-10）をそれぞれ使用するプロセッサ数分に分割し，それぞれ FX/4 上のマルチタスキングライブラリ，KSR1/上のマルチスレッディングライブラリを用いて並列実行した．次にマクロデータフロー処理では，同様に MT14 内の

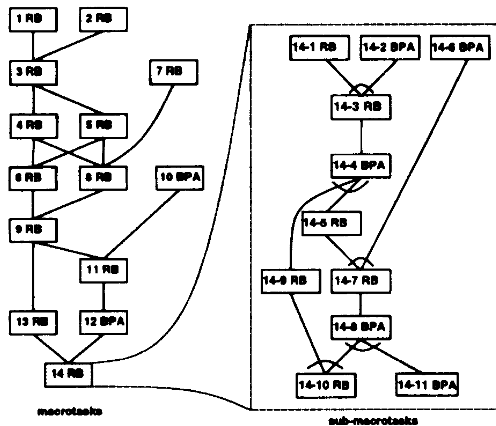


図9 CG プログラムのマクロタスクグラフ
Fig. 9 Macrotask graph of CG program.

DO-all ループを使用するプロセッサ数分に分割するとともに，図9のマクロタスクグラフに示されるようなループおよび基本ブロック間の並列性も抽出した．これらマクロタスクのスケジューリング方式としては，FX/4 上でのマクロデータフロー処理では，FX/4 の全プロセッサ数が4台と小さいため，分散スケジューラ方式を用いた．KSR1 上でのマクロデータフロー処理では，集中および分散スケジューラ方式をそれぞれ用いた．

表3 および **表4** は，それぞれ4プロセッサ構成の FX/4 および16プロセッサ構成の KSR1 上での CG プログラムのシーケンシャル実行，マルチタスキングまたはマルチスレッディング，マクロデータフロー処理の実行結果を示す．表中，matrix size は，対象とする連立方程式の係数行列サイズを表している．

表3に示すように，4プロセッサ構成の FX/4 上でのマルチタスキングでは，シーケンシャル実行時に比べて実行時間短縮ができていないのに対し，マクロデータフロー処理では，matrix size が 240 × 240，スカラ実行の場合で 1/3.67，ベクトル実行の場合で 1/3.44 に短縮できている．

また，表4に示すように，16プロセッサ構成の KSR1 上でのマルチスレッディングでは，matrix size が 256 × 256 の場合で実行時間をシーケンシャル実行時の 1/3.24 に短縮できるのに対し，マクロデータフロー処理では，分散スケジューラ方式の場合で 1/14.95，集中スケジューラ方式の場合で 1/14.73 に短縮できている．

これらの結果より，マクロデータフロー処理では，FX/4 上でのマルチタスキングおよび KSR1 上でのマルチスレッディングに比べて，高い並列性を抽出できるとともにスケジューリングオーバーヘッドが小さいため，プログラムの実行時間を短縮できることが確かめられる．

KSR1 上でのマクロデータフロー処理における集中および分散スケジューラ方式の比較では，matrix size が 256×256 の場合，集中スケジューラ方式では実行時間をシーケンシャル実行時の 1/14.73 に短縮しているのに対し，分散スケジューラ方式では 1/14.95 に短縮できている．集中スケジューラ方式では，スケジューラとして1プロセッサを占有するため，この場合は，分散スケジューラ方式が集中スケジューラ方式より高い並列処理効果を示している．

一方，matrix size が 128 × 128 の場合，分散スケジューラ方式では実行時間をシーケンシャル実行時の 1/7.27 に短縮しているのに対し，集中スケジューラ方式では 1/10.16 に短縮できている．分散スケジューラ方式では，マクロタスクのスケジューリング時に全プロセッサに共有されるスケジューリング用データアクセスのための排他制御が必要となり，スケジューリ

表3 FX/4 上での CG プログラムに対する並列処理手法の性能比較
Table 3 Comparison of parallel processing schemes on an FX/4 using CG program.

	matrix size			
	128×128		240×240	
	実行時間 [s]	速度比	実行時間 [s]	速度比
Seq(s)	98.24	1.00	611.12	1.00
Seq(v)	39.54	1.00	249.38	1.00
M-Task(s)	173.22	0.57	600.74	1.02
M-Task(v)	154.04	0.26	522.34	0.48
MDF(s)	27.49	3.57	166.69	3.67
MDF(v)	11.62	3.40	72.41	3.44

Seq: シーケンシャル実行
M-Task: マルチタスキング
MDF: マクロデータフロー処理
(s): スカラ実行
(v): ベクトル実行

表4 KSR1 上での CG プログラムに対する並列処理手法の性能比較
Table 4 Comparison of parallel processing schemes on a KSR1 using CG program.

	matrix size			
	128×128		256×256	
	実行時間 [s]	速度比	実行時間 [s]	速度比
Seq	44.08	1.00	328.24	1.00
M-thread	35.16	1.25	101.39	3.24
MDF(d)	6.06	7.27	21.95	14.95
MDF(c)	4.34	10.16	22.28	14.73

Seq: シーケンシャル実行
M-thread: マルチスレッディング
MDF(d): マクロデータフロー処理（分散スケジューラ方式）
MDF(c): マクロデータフロー処理（集中スケジューラ方式）

ングオーバーヘッドが集中スケジューラ方式に比べて大きい．そのため，matrix size が 128×128 の場合，すなわちマクロタスクの処理時間が比較的小さい場合は，集中スケジューラ方式が分散スケジューラ方式よりも高い並列処理効果を示している．

4.2.3 スケジューリングオーバーヘッドの性能評価

表5 に FX/4 および KSR1 上でのマルチタスキング，マルチスレッディングおよびマクロデータフロー処理におけるスケジューリングオーバーヘッドを示す．

FX/4 上でのマルチタスキングおよび KSR1 上でのマルチスレッディングにおけるオーバーヘッドは，それぞれ FX/4 上のマルチタスキングライブラリである fork[22]，KSR1 上でのマルチスレッディングライブラリである pthread_create[21] の実行時間を測定したものである．マクロデータフロー処理におけるオーバーヘッドは，プログラムコード中に埋め込まれたスケジューリングコードがマクロタスク1個をプロセッサへ割り当てる処理（マクロタスクの ReadyMT_queue への登録および ReadyMT_queue からのマクロタスクの取り出し）に要する実行時間を測定したものである．

表5 より，FX/4 上でのマクロデータフロー処理のスケジューリングオーバーヘッドは，マルチタスキングの 1/2518 であり，また KSR1 上でのマクロデータフロー処理のスケジューリングオーバーヘッドは，マルチスレッディングに比べて，分散スケジューラ方式の場合で 1/77，集中スケジューラ方式の場合で 1/226 であることが分かる．これらの結果より，FX/4 上でのマルチタスキングおよび KSR1 上でのマルチスレッディングでは，プロセスおよびスレッドのスケジューリングが OS によって行われる（OS の内部状態により数千から数万クロックが必要になる場合がある）ため，オーバーヘッドが大きいことが分かる．それに対して，マクロデータフロー処理では，プログラムコード中に埋め込まれたスケジューリングコードがスケジューリング処理を行うために，オーバーヘッドがこれらの手法に比べて非常に小さいことが分かる．

また，KSR1 上でのマクロデータフロー処理において，分散スケジューラ方式のスケジューリングオーバーヘッドが集中スケジューラ方式よりも大きいのは，分散スケジューラ方式におけるスケジューリング用データアクセス時に必要となる排他制御のためのオーバーヘッドが主な原因である．FX/4 および KSR1 上で排他制御に用いるライブラリおよびその実行時間を**表6** に示す．

表5 ダイナミックスケジューリングオーバーヘッドの比較
Table 5 Comparison of dynamic scheduling overhead.

アーキテクチャ	処理方式	オーバーヘッド [μs]
FX/4	MDF(d)	45.56
	M-task	114739.50
KSR1	MDF(d)	29.66
	MDF(c)	10.08
	M-thread	2273.50

表6 排他制御ライブラリ実行時間
Table 6 Execution time of libraries for mutual exclusion.

アーキテクチャ	ライブラリ名	実行時間 [μs]
FX/4	lock, unlock	16.89
KSR1	pthread_mutex_lock, pthread_mutex_unlock	13.00

5. むすび

本論文では，共有メモリ型マルチプロセッサシステム Alliant FX/4 および Kendall Square Research KSR1 上での Fortran マクロデータフロー処理の実現方法について述べ，性能評価を行った．

これらのマシン上での性能評価の結果，マクロデータフロー処理は，特殊なアーキテクチャを持たない商用マルチプロセッサシステム上で実現可能であり，それらのマシン上で，

(1) 従来のループ並列性に加え，粗粒度並列性も有効に利用できる，

(2) コンパイラが生成したスケジューリングコードによるダイナミックスケジューリングが，従来のマルチタスキングやマルチスレッディングに比べてスケジューリングオーバーヘッドを非常に小さく抑えることができる，

ため，全体として，従来のループ並列化，マルチタスキング，マルチスレッディング等の並列処理手法に比べてより効果的な並列処理を実現できることが確認された．

また，マクロデータフロー処理におけるスケジューリング方式については，使用するマルチプロセッサシステムのアーキテクチャやプログラムの並列性により，集中スケジューラ方式と分散スケジューラ方式のどちらが適しているかが変わるため，今後，プログラムあるいはマクロタスクごとに適切なスケジューリング方式を選択する手法の開発が必要である．さらに，現在著者らは，実用レベルマクロデータフローコンパイラを開発中であり，本コンパイラを用いて Perfect Benchmarks 等の大規模プログラムに対するマクロデータフ

ロー処理の性能評価を行っていく予定である．

謝辞 KSR1 上での性能評価にご協力いただいたキヤノンスーパーコンピューティング S.I.（株）に感謝いたします．本研究の一部は，文部省科学研究費（一般研究 (b)05452354，(c)05680284，奨励研究 (a)07780288）による．

参考文献

1) Guzzi, M.D., Padua, D.A., Hoeflinger, J.P. and Lawrie, D.H.: Cedar Fortran and Other Vector and Parallel Fortran Dialects, *Supercomputing '88*, pp.114–121 (1988).
2) Karp, A.H. and Babb II, R.G.: A Comparison of 12 Parallel Fortran Dialects, *IEEE Software*, Vol.5, No.5, pp.52–67 (1988).
3) CRAY RESEARCH JAPAN LTD.: C90 SERIES Cray Supercomputer (1993).
4) 富士通株式会社：FUJITSU Vol.41, No.1 VP2000 シリーズ特集号 (1990).
5) 日本電気株式会社：NEC 技報 Vol.45, No.2 スーパーコンピュータ SX-3 シリーズ特集 (1992).
6) Padua, D.A. and Wolfe, M.J.: Advanced Compiler Optimizations for Supercomputers, *Comm. ACM*, Vol.29, No.12, pp.1184–1201 (1986).
7) Polychronopoulos, C.D.: *Parallel Programming and Compilers*, Kluwer Academic Publishers (1988).
8) Banerjee, U.: *Loop Transformations for Restructuring Compilers - The Foundations*, Kluwer Academic Publishers (1993).
9) Banerjee, U.: *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers (1988).
10) 笠原博徳：並列処理技術，コロナ社 (1991).
11) 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，信学論，Vol.J73-D-I, No.12, pp.951–960 (1990).
12) Honda, H., Aida, K., Okamoto, M., Yoshida, A., Ogata, W. and Kasahara, H.: Fortran Macro-dataflow Compiler, *Fourth International Workshop on Compilers for Parallel Computers*, pp.265–286 (1993).
13) 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，信学論，Vol.J75-D-I, No.8, pp.511–525 (1992).
14) 本多弘樹，合田憲人，岡本雅巳，笠原博徳：Fortran プログラム粗粒度タスクの OSCAR における並列実行方式，信学論，Vol.J75-D-I, No.8, pp.526–535 (1992).
15) 吉田明正，前田誠司，尾形 航，笠原博徳：Fortran マクロデータフロー処理におけるデータローカライゼーション手法，情処学論，Vol.35, No.9, pp.1848–1860 (1994).
16) 吉田明正，前田誠司，尾形 航，笠原博徳：Fortran 粗粒度並列処理における Doall/シーケンシャルループ間データローカライゼーション手法，信学論，Vol.J78-D-I, No.2, pp.162–169 (1995).
17) 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手法，情処学論，Vol.35, No.4, pp.513–521 (1994).
18) Bruke, M. and Cyton, R.: Interprocedural Dependence Analysis and Parallelization, *ACM SIGPLAN '86 Symp. on Compiler Construction* (1986).
19) Alliant Computer Systems Corp.: FX/Fortran Programmer's Handbook (1988).
20) Kendall Square Research Corp.: Kendall Square Research Technical Summary (1992).
21) Kendall Square Research Corp.: KSR Parallel Programming (1993).
22) Alliant Computer Systems Corp.: FX/Fortran Language Manual (1987).

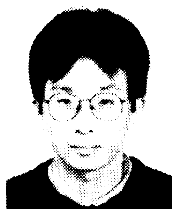
（平成 7 年 6 月 8 日受付）
（平成 8 年 1 月 10 日採録）

合田 憲人（正会員）

昭和 42 年生．平成 2 年早稲田大学理工学部電気工学科卒業．平成 4 年同大学院修士課程修了．現在，同大学院博士課程在学中．平成 4 年同大情報科学研究教育センター助手．並列処理方式，並列化コンパイラ，マルチプロセッサ OS の研究に従事．電子情報通信学会会員．

岩崎 清（正会員）

昭和 45 年生．平成 5 年早稲田大学理工学部電気工学科卒業．平成 7 年同大学院修士課程修了．現在，大日本印刷株式会社中央研究所勤務．並列処理に興味を持つ．



岡本 雅巳（正会員）

昭和41年生．平成2年早稲田大学理工学部電気工学科卒業．平成4年同大学院修士課程修了．現在，同大学院博士課程在学中．平成5年同大情報科学研究教育センター助手．並列化コンパイラ，並列処理方式の研究に従事．電子情報通信学会会員．

笠原 博徳（正会員）

昭和55年早稲田大学理工学部電気工学科卒業．昭和60年同大学院博士課程修了．工学博士．昭和58～60年早稲田大学理工学部助手．昭和60年カリフォルニア大バークレー短期客員研究員，日本学術振興会第1回特別研究員．昭和61年早稲田大学電気工学科専任講師．昭和63年同助教授，現在に至る．平成元年～2年イリノイ大学Center for Supercomputing R & D客員研究員．昭和62年IFAC World Congress 第1回 Young Author Prize授賞．主な著書「並列処理技術」（コロナ社）．電子情報通信学会，電気学会，シミュレーション学会，ロボット学会，IEEE，ACM等会員．

成田誠之助

昭和37年早稲田大学大学院修士課程修了．37年アメリカ・パデュー大学大学院留学（フルブライト留学生）．38年早稲田大学理工学部助手，以後，講師・助教授を経て，48年教授，現在に至る．工学博士．分散計算機制御システム，並列処理，産業用ロボット制御，デジタル制御理論，CIM等の研究に従事．計測自動制御学会，電気学会，ロボット学会，IEEE各会員．