メタプロセスモデルに基づくポータブルな並列プログラミングインタフェースMpC

緑川博子† 飯塚肇†

並列プログラミングモデルとしてすでに提案したメタプロセスモデルのためのインタフェースとしてMpCを開発し，計算機クラスタおよび共有メモリ型並列マシンにおける一貫性のある並列プログラミングを可能にした．メタプロセスモデルは，従来の共有メモリプログラミングモデルが持つ並列プログラムの可読性の良さと記述の簡便性，さらにメッセージパッシングモデルの持つ並列動作記述上の柔軟性の両方を提供する．実装には計算機クラスタにはユーザレベルソフトウエアDSMを，共有メモリ型並列マシンにはpthreadを用いている．またMpCコンパイラは下位実装にgccを使用している．このため幅広い計算機アーキテクチャやOSへの対応が可能で，MpCプログラムとMpCコンパイラの移植性は高い．MpCは，クラスタや共有メモリマシンにおけるOpenMPやUPCとの比較でも同等以上の性能が得られている．さらに特別な通信ハードウェア，OSの変更などを行わず，一般ユーザレベルの権限で，容易にMpCを用いる並列処理環境を構築できる．

MpC: Portable Parallel Programming Interface Based on Meta Process Model

Hiroko Midorikawa† and Hajime Iizuka†

This paper proposes a new portable parallel programming interface MpC, Meta Process C, for Meta Process model. The Meta Process model gives us good readability/writability of parallel programs as shared memory programming model does, in addition to high flexibility for parallel program description given by message passing model. Meta Process model implementatioin employs user level software DSM for clusters and pthread for shared memory parallel machines. This enables MpC programs portable to a wide variety of computer architectures and UnixOSs. The MpC compiler uses gcc in its under layer, so it is highly portable. Moreover, the paper shows good MpC performance in comparison with OpenMP on clusters and gnuUPC on a SMP machine. Without special communication devices or OS modification, the Meta Process model and MpC can be used to build a parallel programming environment with good portability.

1. はじめに

並列プログラムのための代表的なプログラミングモデルには，PVM，MPIに代表されるメッセージパッシングモデル（MPモデル）とOpenMP，pthreadなどの共有メモリモデル（SMモデル）がある．MPモデルでは，MPIが計算機の物理的メモリ構成にかかわらず標準APIとして多くのシステムで使用できるようになり，並列プログラムの一般化と普及に大きく貢献した．一方，SMモデルは，従来の逐次プログラムからの継続性が良く，MPモデルに比べ，煩雑なメッセージ送受信の記述がない点で記述が簡便で，可読性が良い．このような背景から，SMモデルの標準化をめざしてOpenMP[21]が提案され，いくつかの並列計算機で実装されてきている[14),15)]．また，性能価格比の良い並列システムとして広く用いられるようになった計算機クラスタなどにおいても，ソフトウエア分散共有メモリ（SDSM）を構築し，その上でOpenMPを利用する研究が行われている[16)~18),20)]．しかし，計算機クラスタでは，CPU速度に比べ遅いネットワークでノードがつながれており，ノード内外のメモリアクセス速度差が大きいため，SMモデル用に提案されたAPIや言語をそのまま適用しようとすると，分散メモリを意識したデータの分散割付け機能がなかったり，メモリ一貫性のための無駄な同期が自動挿入されたりなど，速度性能を低下させる要因が多々ある[16),19)]．

そこで我々は，SMモデルとは異なる分散共有メモ

† 成蹊大学工学部経営・情報工学科
Faculty of Engineering, Seikei University

リプログラミングモデルであるメタプロセスモデルを提案し[1)]，これを実現するための API として，MpC 言語を開発した．このモデルでは，*shared* というプロセス共有データ型を導入し，プロセス局所データと明確に区別する．また，無駄なメモリ一貫性同期を避け，記述の柔軟性を増すために，プログラマによる明示的な並列処理記述を前提とする．このため従来の SM モデルに比べ，計算機クラスタなどでの並列処理を，より効率良く実行させることが可能である．また実装に関しては，ユーザレベルソフトウエア DSM や，pthread[22)] を用いて，クラスタだけでなく共有メモリ型並列マシンにおいても実行できるように移植性を高めた．これにより pthread やここで用いる代表的な SDSM が稼働する幅広いアーキテクチャと OS のプラットホーム上で MpC プログラムを実行することができる．

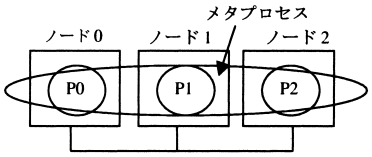


図 1 メタプロセスとプロセスの関係
Fig. 1 Meta Process and its component processes.

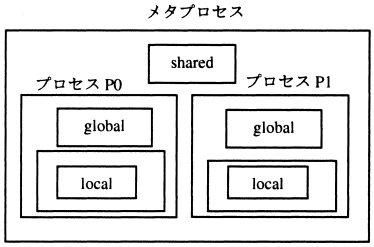


図 2 shared 変数と階層データスコープ
Fig. 2 Shared variables and the hierarchical data scope.

2. メタプロセスモデル

通常，クラスタ上での並列処理は，**図 1** のように，各計算ノードにあるプロセスが複数で共同して 1 つの処理を行う．メタプロセスモデルでは，ある処理を共同で行う複数のプロセス全体をメタプロセスと名付けている．メタプロセス中のプロセスは，通常のプロセスと同じで，ファイルやメモリなどの資源に関連づけられた OS 上の 1 つの実行単位である．一方，メタプロセスはここで導入した新しい概念で，通常 OS では認識されないが，分散共有メモリ機能をサポートするなんらかのシステムソフトウエアなどによって認識される．ユーザにとっては，メタプロセスは 1 つの応用処理に対応する実行単位である．メタプロセスモデルでは，ある 1 つのメタプロセスに属するすべてのプロセスからアクセスが可能な，*shared* という共有データを導入し，プロセス局所とプロセス共有の 2 種のデータを，階層スコープを用いて明確に区別する．しかし，メタプロセスを構成するすべてのプロセスは，共有データに関して共通の単一アドレス空間を持つ．このため，従来の MP モデルとは異なり，煩雑なメッセージパッシングの記述は不要である．メタプロセスモデルは，従来の MP モデルの持つ並列動作記述上の柔軟性と，SM モデルの持つ並列プログラムの可読性の良さや記述の簡便性の両方を兼ね備える．

このモデルでは，メタプロセス中にあるプロセス群が共有するのは shared で定義された特別なデータだけで，それ以外のデータは共有されない．すなわち，メタプロセス中の各プロセスはもともと独自のアドレス空間を持ち，さらに追加的に shared データのための共有のアドレス空間が付加される．これが，このモデルでの実行実体をスレッドといわずプロセスと呼ぶ理由である．これは，OpenMP や UPC などの他の多くの共有メモリモデルで，スレッドという言葉を使っているのと対照的である．スレッドは，本来，すべての資源を共有する実行単位で，スコープや権限の関係で，実際にアクセスするか否かを別にすれば，ファイルやメモリ，アドレス空間もすべて共有していることが基本である．すなわち C 言語のようなアドレスを直に扱うプログラム環境では，アドレスが同一なら，本来，同じデータがアクセスできなければスレッドと呼ぶのにふさわしくないと考えている．

本モデルで導入されている付加的に追加されたアドレス空間と共有データ shared は，従来のプロセスやスレッドにはない概念なので，メタプロセスという言葉を導入した．また本モデルは，純粋な共有メモリモデルとは考えておらず，分散共有メモリモデルと呼んでいる．

2.1 共有データのための共有アドレス空間

メタプロセスとその中のプロセスという 2 つの実行単位に対応させ，プロセス共有データとプロセス局所データを区別し，そのデータスコープを**図 2** のようにした．これにより共有データへはすべてのプロセスからアクセスができ，アドレスも同一で C におけるポインタ操作も可能になっている．また階層データスコープを導入することにより，プロセス内データは，他のプロセスから直接参照することを事実上，禁止している．

2.2 共有データの緩和型メモリ一貫性

共有データに関しては緩和型メモリ一貫性モデル[2),11)~13)]を前提とする．これらは各種の SDSM で用いられているものであるが，この前提はクラスタなどの分散メモリ型並列マシンなどでの実行において特に性能上の利点をもたらす．一方，UMA 型，NUMA 型を問わず共有メモリ型並列マシンに対しても性能上なんら害は及ぼさない．また，一貫性同期は，バリア，ロックなどを用いてプログラマが明示的にプログラム中に記述するため，暗黙にシステムが行うメモリ一貫性同期は存在せず，アルゴリズム上，本当に必要なときのみに一貫性同期を行うことができる．

2.3 共有データとプロセスとの関連づけ

共有データは，メタプロセス内のプロセスのいずれかに関連づけることができる．あるプロセスがある特定の共有データや，共有データの特定部分に多頻度でアクセスすることが分かっているときは，プロセスと共有データを関連づけることで，実装システム（SDSM など）にその情報（ヒント）を与え，より効率的なデータの配置を行うことができるようにしている．データへのアクセスパターンに特徴がなかったり，ユーザがアクセスパターンを把握できなかったりする場合には，このような関連づけ記述を省略することもできる．その場合には実装系に依存した任意のプロセスに関連づけられる．

2.4 高 移 植 性

メタプロセスモデルは分散メモリ型だけでなく共有メモリの並列マシンなどに対しても実装可能である．代表的 SDSM や pthread で用いられている API の基本部分を MpC に組み込むことで，MpC プログラムは，SDSM から他の SDSM へ，あるいは共有メモリ型マシンからクラスタへと移植が可能で，SDSM や pthread の利用が可能な様々なアーキテクチャや OS 上で MpC プログラムの実行ができる．

3. MpC

MpC（メタプロセス C）は，付録 MpC 文法規則のように ANSI C を拡張した言語で，メタプロセスモデルをクラスタや共有メモリ型マシン上で実現するためのポータブルな並列プログラミングインタフェースである．

3.1 共有データ型 shared

新しく *shared* を C の記憶クラス指定子として組み込み，shared 型のデータのスコープはメタプロセス全体で，図 2 のように shared，global，local データの順のスコープ階層になる．同じ名前が複数のスコープのデータに使用されると，最も内側のスコープのデータを示すものとして扱われる．

MpC のポインタは従来の C と同じで，ポインタの中身の記憶クラス（register，auto，static，extern）は考慮せず，型指定子（int，float，double，char など）にのみに注目する．すなわち，ポインタの指す中身が *shared* か否かについては関知しない．このような区別をしても，性能上，有利に扱うような仕組みが多くの SDSM にはないうえ，キャストや多段の間接参照のポインタに対してのそのような区別は際限がなく，いたずらにデータ宣言記述を複雑にし，コンパイラやプログラマに不必要な負担を増やすと判断したからである．

したがって，たとえば整数への pointer は MpC では以下の 2 つの表現のみである．

(1) int を指す局所ポインタ変数

```
int *p1;
```

(2) int を指す共有ポインタ変数

```
shared int *p2;
```

3.2 共有データの分散マッピング宣言

スカラ，ベクタを問わず共有データを任意のプロセスへ割り付けることができる．特に配列データに関しては，柔軟性のある割付けが可能で，**図 3** に示すような，バンド，タイル，ライン，キューブなどのサイクリック割付けや様々な割付けが容易に行える．

共有変数とプロセスとを関連づける割付け API は **図 4** のような宣言形式で書くことができる．付録 MpC 文法規則に示すように，従来の C の変数宣言の先頭に *shared* を付加した共有配列データに関しては，後尾に :: から始まる分散割付け指定子（mapping-specifier）を付加することができる．分散割付け指定子は，割付け分割数情報（div-information）（図 4 の [dm]..[di]..[d0]）と割付けプロセス情報（owner-information）（図 4 の (st,n)）からなる．割付け分割数情報とは，データの各次元を何等分するかを次元ごとに指定する（図 4 では次元 i が di 等分される）．割付けプロセス情報は，割付け開始プロセス番号（図 4 の st）とサイクリック割付けに使用するプロセス数（図 4 の n）を指定する．各パラメータは，1 以上，あるいは 0 以上の整数定数，もしくは定数計算式で指定する．また配列変数だけでなく，スカラ変数のプロセス割付け指定も可能になっている．分割領域とプロセス番号の対応は，プロセス番号の昇順に，メモリアドレスの小さいほうから割り付けられる．C 言語の配列記述では，右側に記述した次元の [ ] にある分割から順にプロセスに割り付けられる．

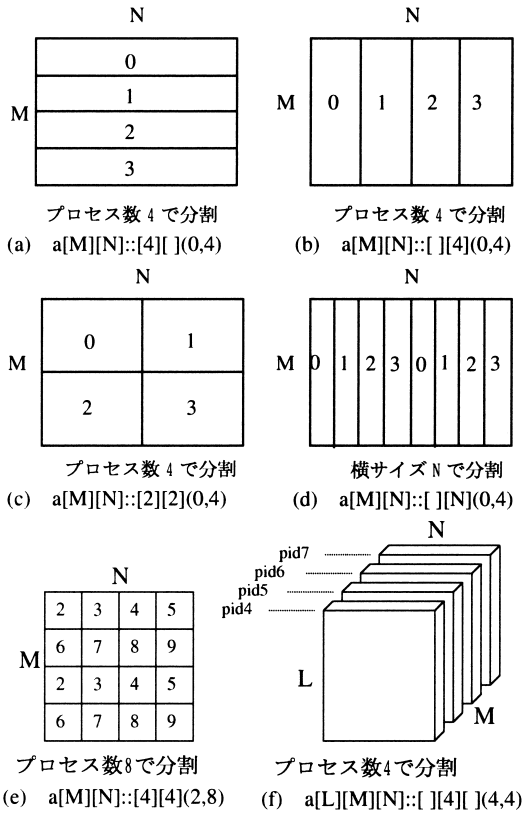


図 3 共有変数のプロセスへの分散割付け例
Fig. 3 Examples of shared variable distributed mapping.

shared 型名 変数名[sm]..[si]..[s0]::**[dm]..[di]..[d0](st, n)**
変数名[sm]..[si]..[s0]： 変数の各次元サイズ
[dm]..[di]..[d0]： 各次元の分割数（省略時は分割数１）
st ： 割付開始プロセス番号(省略時は 0 か任意プロセス)
n ： 割付プロセス数 (省略時は全使用プロセス数)

図 4 共有変数の宣言形式
Fig. 4 Shared variable declaration.

```
(a) shared double a;                          任意プロセスに割付
(b) shared int x::(st);                       指定プロセス st に割付
(c) shared double y[M];                       任意プロセスに一括割付
(d) shared char b[K][L]::[NPROCS][ ](st); NPROCS 分割
(e) shared float z[M]::[n](st, n);            n 分割 st〜st+n-1
(f) shared int q[M][L][N]::[n][ ][ ];         n 分割 0〜n-1
```

図 5 共有変数の宣言例
Fig. 5 Examples of shared variable declaration.

実際には，**図 5** の宣言例に示すように，分散割付け指定子全体を省略する（a，c）ことも可能であるし，割付け開始プロセス番号と割付けプロセス数の両方を省略（f），あるいは割付けプロセス数のみを省略する（b，d）ことも可能である．

割付けプロセス数のみを省略した場合（d）は，メタプロセスを構成する全プロセス数（NPROCS）を指定したと見なされる．NPROCS とは，後述する MpC 定数でメタプロセスを構成する全プロセス数を表す．分散割付け指定子全体を省略した配列変数（c）やスカラ変数（a）の場合は，実装系に依存するある 1 つのプロセスに一括して割り付ける．割付け分割数情報のみ指定し，割付けプロセス情報を省略した場合（f）は，プロセス 0 から NPROCS-1 に割り付けられる．

図 3 (a)〜(d) は 4 プロセスに割り付けた例で，(a) は横バンド割付け，(b) は縦バンド割付けで，対応する配列の次元に分割数 4 を指定する．(c) はタイル割付けで各次元を 2 分割している．(d) は縦方向のライン割付けで，データの縦の次元サイズ N を分割数とし 4 プロセスにサイクリックに割り付けている．(e) はプロセス数 8 に対し 4×4 のタイル割付けしたものである．ここでは割付け開始プロセス番号を 2 としているので，2〜9 の 8 プロセスが使用される．もし，指定した割付けプロセス数が実行時にプロセス数（NPROCS）よりも大きかったり，指定割付けプロセス番号の範囲が，実際に使用できるプロセス番号の範囲を超えていたりする場合にも，st〜st+n の NPROCS の剰余が実際には使用される．たとえば，実際のプロセス数が 8 個（0〜7）しかなかった場合は，(e) の図中の 8，9 の代わりに 0，1 が使用される．(f) は 3 次元配列の中間の次元で 4 プロセスに割り付けたものである．割付け開始プロセス番号を 4 としているので，プロセス 4〜7 に割り付けられる．3 次元配列の場合には，さらに他の次元でも分割し，直方体型に分割することもできる．実際のプログラムでのパラメータ指定には，NPROCS などを用い，実行時のプロセスに依存して分散割付けと並列処理が対応するように，プログラムを記述することが多い．

3.3 MpC 定数

MpC では，コンパイル時には不定であるが，メタプロセス実行時には定数として使用できる MpC 定数という実行時定数を用意している．NPROCS は，その 1 つでメタプロセスを構成する全プロセス数を表す．MYPID は，0 から NPROCS-1 までの整数で，プロセスにユニークな識別番号である．MYPID は使用するプロセスごとに異なった値が実行時には設定されている．MpC プログラムでは，ユーザが定義せずにこれらの定数を使用できる．

この 2 つの MpC 定数を利用して，前述の共有データの分散割付け指定の分割数や使用プロセス数を記述

表 1 MpC 標準ライブラリ関数（一部）
Table 1 MpC standard library functions.

	MpC 標準関数（一部）
初期化	mpc_init(int argc, char *argv)
終了	mpc_exit( )
エラー終了	mpc_err(char *msg)
バリア	mpc_barrier(int barrierid)
ロック	mpc_lock(int lockid)
アンロック	mpc_unlock(int lockid)
条件シグナル	mpc_cond_signal(int condid)
条件シグナル	mpc_cond_broadcast(int condid)
条件シグナル待ち	mpc_cond_wait(int condid, int lockid)
共有データ割付	void *mpc_alloc(int size)
共有データ割付	void *mpc_alloc(char *declare)

したり，処理における計算に用いたりすることが可能である．

3.4 MpC 標準ライブラリ関数

MpC では，C 言語からの変更を，共有データの定義とスコープ処理，分散割付けにとどめ，並列処理の同期，データの排他制御，データ一貫性同期などの操作は，すべて MpC 標準関数を用いる設計にしている．これは，従来の pthread ライブラリや，多くの SDSM，MPI などと同様である．実装系ごとの動作の違いはすべてライブラリが吸収する．これによりコンパイラの負荷が軽減され，実装系ごとのライブラリセットを用意するだけで容易に実装系の変更ができる．

MpC の API として**表 1** に示すような関数を用意している．メタプロセスの初期化と終了処理，バリア同期，ロック/アンロック同期，条件変数，共有データ動的割付けなどがある．これらの関数コールは，MpC コンパイラにより実装に依存した関数コールに置き換えられる．

3.5 MpC プログラム例

以上で述べた共有データ分散割付け宣言，MpC 標準関数，MpC 定数などを用いた簡単なプログラム例を**図 6** に示す．これは 2 次元配列データに何らかの処理 process() をした結果の総和を求めるプログラムである．2 次元配列を横方向に全使用プロセス数で分割して，部分データ領域を各プロセスで分担し，なんらかの処理 process() を行う．最後に結果を総和 sum に加える．プロセス 0 はファイルからのデータ入力と初期化，最後の結果出力を行う．

この例では，単純化のために，多くの並列型言語で使用可能な parallel for 型の SPMD 型プログラムを示したが，プロセスごとに異なった処理を行う MPMD 型のプログラムなど，不規則構造を持つプログラムの記述も，明示的記述を基本とする MpC では柔軟に書くことができる．その一方で，MP モデルにあるメッセージ送受信記述がないので，並列アルゴリズム本来の流れが読みやすく，記述もしやすい．

```
#include <stdio.h>
#include <mpc.h>
#definr M 1024
#define N 2048

shared double matrix[M][N]::[NPROCS][ ];
shared double sum::(0);

int main(int argc, char **argv)
{
      FILE fp;
      double mysum=0;
      int start, end, i, j;

      mpc_init(argc, argv);
      if(MYPID == 0){
            fp=fopen("initial.dat", "r");
            for(i=0; i<M; i++)
                  for(j=0; j<N; j++) fscanf(fp,"%lf", &matirx[i][j]);
            sum = 0;
      }
      mpc_barrier(0);

      start = M/NPROCS*MYPID;
      end = start+M/NPROCS;
      for(i=start; i<end; i++)
            for(j=0; j<N; j++) mysum += process(matrix[i][j]);

      mpc_lock(0);
         sum += mysum;
      mpc_unlock(0);

      mpc_barrier(0);
      if(MYPID == 0)    printf("Result=%f¥n",sum);
      mpc_exit( );
}
```

図 6 MpC プログラム例
Fig. 6 A MpC program sample.

4. UPC と MpC の比較

MpC と同様に分散共有メモリモデルを意識した言語に UPC[7)~9)] がある．UPC は，MpC とは実装や設計思想が異なるが API が似ている言語で，いくつかの組織が開発している．UPC と MpC との主な違いを**表 2** に示す．

4.1 *shared* とポインタ

UPC では，MpC と同様に *shared* という予約語を用いるが，ANSI C の型修飾子（const や volatile を含む）の 1 つとして導入している．このため，C でのポインタの扱いが異なってくる．たとえば，UPC では int へのポインタとして次の 4 種が存在する．

(1) 局所データの int を指す局所ポインタ変数
　　int *q1;
(2) 共有データの int を指す局所ポインタ変数
　　shared int *q2;
(3) 局所データの int を指す共有ポインタ変数

表 2 MpC と UPC の比較
Table 2 Comparison of MpC and UPC.

	MpC	UPC
分散割付 プロセス指定	任意の開始プロセス番号, 任意プロセス使用数	配列データ 開始プロセス番号は 0 固定 使用数 全プロセス数固定 スカラーデータは 0 固定割付
分散割付の記述性	柔軟 ○	制限あり △
メモリ一貫性モデル	relaxed	strict / relaxed 選択
ポインタ参照先の shared の区別	区別しない	区別する
shared の実装	型修飾子	記憶クラス指定子
対象プログラム型	SPMD / MPMD	SPMD
同期機能の提供	ライブラリ関数	構文，ライブラリ関数
並列構文	なし	forall, sizeof, threadof など
実装システムの違い	ライブラリ関数で吸収	コンパイラ内で処理
コンパイラの移植性	高い	低い

```
MpC:   shared int mat[M][N];
UPC:   shared [ ] int mat[M][N];
        (a) 集中一括割付

MpC:   shared int mat[M][N]::[M][N];
UPC:   shared int mat[M][N];
        (b) 1 要素サイクリック割付

MpC:   shared int mat[M][N]::[NPROCS][ ];
UPC:   shared [M*N/THREADS] int mat[M][N];
        (c) 横バンド割付

MpC:   shared int mat[M][N]::[ ][NPROCS];
UPC:   shared [N/THREADS] int mat[M][N];
        (d) 縦バンド割付
```

図 7 UPC と MpC の分散割付け宣言の比較
Fig. 7 Difference of UPC and MpC distributed mapping.

```
int *shared q3;
```

(4) 共有データの int を指す共有ポインタ変数

```
shared int *shared q4;
```

MpC では，3.1 節で述べたようにポインタの指す先が *shared* か否かを区別しない設計としている．上述のようにポインタを区別することに大きな実際上の利点が少ないこと（特に (3) などはスコープの観点からも必要性が低い）や，*shared* という特性は，値の表現に関連したデータ型よりもむしろスコープにも関連がある storage の種類（register，auto，static，extern）として扱ったほうが自然であると考えたからである．

4.2 共有データ分散記述方式

UPC は共有データの分散割付け手法においても MpC と異なる．比較のために，最も単純ないくつかの分散割付け宣言例を，**図 7** に示す．UPC では，MpC とは異なり，分割数ではなく，分割する部分ブロックサイズを指定する方式を用いているが，基本的に MpC で提供するような柔軟な分割方式の指定はできない．たとえば，UPC では，図 3 にあるようなタイル型や直方体型のような複数方向の分割指定はできない．また，もともと UPC とその元となっている Split-C は，計算機クラスタよりも細粒度並列の共有メモリマシン用に設計されている．このため，共有配列データのデフォルトの分散方式は，(b) に示すように配列 1 要素ずつをサイクリックに各スレッドに割り付けするというもので，計算機クラスタのようなマシンにとっては非常に効率が悪いものになっている．

UPC では，OpenMP などと同様に共有メモリを前提にしたスレッドモデルを採用しており，SPMD 型プログラムを想定しているため，割付けに用いるスレッド数は，必ず全スレッドでなければならない．すなわち，任意の複数スレッドのみにサイクリックに割り付けたり，任意の 1 スレッドに集中割付けしたりすることができない．このため，縦分割指定では，分割数は実際の使用全スレッド数の倍数に限られ，3 次元配列以上になると任意の次元で分割することも難しい．

4.3 メモリ一貫性モデル

MpC は，前述のように緩和型メモリ一貫性によるプログラムを前提にしているが，UPC では strict/relaxed の 2 種のメモリ一貫性のうちの 1 つを，プログラムブロックごとあるいはプログラム全体に対して指定する方式となっている．

4.4 コンパイラ実装方式

MpC では，下層に用いる SDSM や OS，アーキテクチャの違いなどの実装依存の部分は，SDSM や pthread のライブラリ関数で吸収する設計になっており，コンパイラ（トランスレータ）の処理は，最終的な実行コード生成部以外は汎用になっている．このため，プログラム中に実装系の初期化，終了などを行うための関数（mpc_init()，mpc_exit()）の記述を必要とするが，MpC コンパイラの移植は実装系によらないので容易である．

一方，UPC には，API として同期関数はサポートしているが，初期化，終了処理関数などはなく，コンパイラにより実装システムに依存した部分を組み入れる．それ以外にも，SPMD モデルを前提にした forall 構文や threadof，同期なども構文として言語に組み込んでいる．このためコンパイラはそれぞれのターゲットマシンアーキテクチャに応じて開発されている．

5. MpC コンパイラ

MpC コンパイラは，MpC から C への変換と，MpC

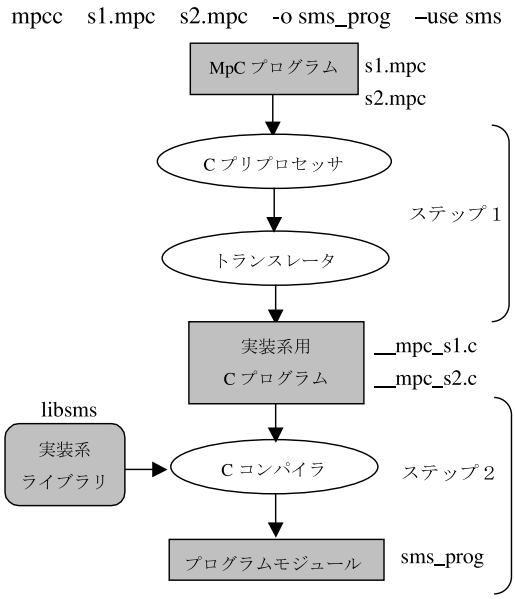


図 8 MpC コンパイラの構成
Fig. 8 Structure of MpC Compiler.

実行モジュール生成の 2 ステップがある[1]．**図 8** は，下層の実装システムとして SDSM（SMS）を指定した場合のコマンド例とその処理の手順を示す．第 1 ステップでは，まずプリプロセッサによりヘッダファイルなどを展開する．次にトランスレータで *shared* 型のデータと元々の C のデータとの階層スコープチェックを行い，C 文法に適合するようにリネームやプログラムコードの挿入を行う．さらに MpC の並列 API をすべてユーザの指定した下層の実装システムライブラリ関数へ変換する．

第 2 ステップでは，ターゲットマシンの C コンパイラを用いて，指定実装システムのライブラリ関数とリンクする．実装ライブラリとしては，クラスタ用にはユーザレベル SDSM を用い，共有メモリ型マシン用には pthread を用いている．

現在のところ，下層 C コンパイラには gcc または icc を想定している．MpC コンパイラでは，まず下層 C コンパイラのプリプロセッサを利用して，ヘッダファイルを展開する．本モデルでの shared データは，スコープ階層で，大域データよりも上の最上位に位置しているため，ヘッダ内に書かれたすべての大域データを把握したうえで，スコープ処理をする必要が生じる．しかし一般に C コンパイラは，プリプロセッサと後段のコンパイラが密接に関連しており，プリプロセッサ出力には標準仕様が特にないため，下層に用いるコンパイラのプリプロセッサに応じたトランスレータの書き方が必要になる．gcc のプリプロセス後のファイルには，gcc が後段のコンパイラで用いるための C 言語標準にはない gcc 特有の予約語などが現れる．このためこれらの関連のない語を読み飛ばす処理などを MpC トランスレータでは行っている．またプリプロセッサが出力したヘッダファイル名や行番号などのコメントを，MpC トランスレートのエラー時のヘッダファイル名や行数の表示に利用している．icc の出力形式は gcc と同様なので，x86 では，より高性能な icc を利用することもできる．gcc は多くのアーキテクチャで広く利用できるため，MpC コンパイラのターゲットマシンは多様で，移植性が高い．

MpC プログラムが，どのように C にトランスレートされるかについて，変換前の MpC プログラムと，下層に SMS を用いた場合の変換後の C プログラムを**図 9** に示す．ただし，ヘッダファイルの展開部分については省略している．図中の ⟨ 番号 ⟩ は，それぞれのプログラムの対応を示す．図 9 の MpC プログラム自体に意味はないが，MpC プログラムにおける変数 a は異なるスコープで使用されても，スコープチェックにより必要なところのみリネームされていることが分かる（⟨3⟩，⟨5⟩）．shared データの宣言は下層の SDSM の割付け関数を用いて動的に割り付けられる（⟨1⟩，⟨2⟩）．MpC の分散マッピング宣言は，下層 SDSM のデータ割付け関数に十分な機能がない場合には生かされない場合もある．MpC 標準関数は，それぞれ下層の SDSM の対応する関数に変換される（⟨6⟩，⟨10⟩）．下層に pthread を用いた場合の変換後の C プログラムは，SDSM の場合よりも複雑になる．

6. 計算機クラスタにおける実装と評価

6.1 計算機クラスタにおける実装

現在，クラスタにおける MpC プログラムの実行には SMS[2]，TreadMarks[3]，JIAJIA[5] などの SDSM を用い，MpC プログラムの API を各 SDSM の API に変換している．各 SDSM の API，記述方式には多少の違いがあるが，バリア，ロック/アンロック，共有データの動的割付けなどの基本 API はほぼ同等である．多くの既存の SDSM プログラムはこの基本 API のみを使用して書かれているため，異なる SDSM プログラムへの変換が可能である．

さらに，ここで用いた SDSM はユーザレベルソフトウエアなのでインストールも容易で，いずれも移植性が高く，多くのアーキテクチャや OS 上で稼働している．TreadMarks は，AIX（SR6000），Linux（Alpha/x86），SunOS/Solaris（SPARC/x86），IRIX

```
#include <stdio.h>
#include <mpc.h>
#define M 1024
#define N 512

shared double mat[M][N]::[NPROCS][ ](0,NPROCS);   <1>
shared int a;                                      <2>
void func(int b)
{     mat[M/NPROCS*MYPID][0]=a;                    <3>
      mat[M/NPROCS*MYPID][1]=b;                    <4>
}
void func2()
{     int a=22;
      mat[M/NPROCS*MYPID][2]=a;                    <5>
}
int main(int argc,char *argv[])
{
      int i,j, c=44,start,end;
      mpc_init(argc,argv);      <6>

      start=N/NPROCS*MYPID;
      end= N/NPROCS*(MYPID+1);
      for(i=start;i<end;i++)
            for(j=0;j<N;j++)    mat[i][j]=0.5;       <7>
      a = -100;                                      <8>
      if(MYPID%NPROCS==0)      func(c);
      if(MYPID%NPROCS==1)      func2();
      printf("%d %d¥n",mat[0][0],mat[1][1]);         <9>

      mpc_exit( );      <10>
}
```

```
double (*__mpc_sh_mat)[512];          <1>
int *__mpc_sh_a;                      <2>

void func(int b)
{    __mpc_sh_mat[1024/NPROCS*MYPID][0]=(*__mpc_sh_a);      <3>
     __mpc_sh_mat[1024/NPROCS*MYPID][1]=b;                  <4>
}
 void func2()
{    int a=22;
     __mpc_sh_mat[1024/NPROCS*MYPID][2]=a;      <5>
}

int main(int argc,char *argv[])
{
   int i,j, c=44,start,end;
   sms_init(argc,argv);              <6>

   __sms_dim[0] = -1;   __sms_div[0] = -1;
   __mpc_sh_a =                                          <2>
   ( int *)sms_mapalloc(__sms_dim, __sms_div, sizeof( int ), 0, 0);

   __sms_dim[0] = 1024;   __sms_dim[1] = 512;   __sms_dim[2] = -1;
   __sms_div[0] = NPROCS;   __sms_div[1] = 1; __sms_div[2] = -1;      <1>
   __mpc_sh_mat=
   (double(*)[512])sms_mapalloc(__sms_dim,__sms_div,sizeof(double),0,NPROCS);

   start=512/NPROCS*MYPID;
   end= 512/NPROCS*(MYPID+1);
   for(i=start;i<end;i++)
        for(j=0;j<512;j++)     __mpc_sh_mat[i][j]=0.5;          <7>
   (*__mpc_sh_a) = -100;                                  <8>
   if(MYPID%NPROCS==0) func(c);
   if(MYPID%NPROCS==1) func2();
   printf("%d %d¥n",__mpc_sh_mat[0][0],__mpc_sh_mat[1][1]);      <9>

   sms_exit(0);      <10>
}
```

図 9 変換前の MpC プログラムと変換後の SMS 用 C プログラム
Fig. 9 MpC Program and translated C program for SMS.

(SGI)，FreeBSD，HPUX など[4]，JIAJIA は，SunOS/Solaris (SPARC)，AIX (SP2)，Linux (x86) などで[6]，SMS は Linux/FreeBSD（x86）で稼働ずみである．

6.2 SDSM による MpC プログラムの実行

MpC プログラムを SMS，TreadMarks，JIAJIA の 3 つの SDSM を用いて実行した結果を**図 10**，**図 11**，**図 12**，**図 13** に示す．用いたプログラムは，**表 3** に示す 4 種で，ep，tsp（TreadMarks），lu（Splash ベンチマーク），行列乗算 mm（JIAJIA）である．また測定環境を**表 4** に示す．MpC プログラムは異なる 3 種の SDSM で，変更なしに実行することができ，移植性は高い．lu や ep といった計算中心の応用では性能向上比は高い．tsp は他のプログラムに比べロックが非常に多いという特徴があり，実装 SDSM による違いが現れている．

MpC の API は，ここで用いた各 SDSM の基本 API とほぼ同等なので，MpC プログラムの実行と，SDSM プログラムを直接実行した場合との差はほとんどない．

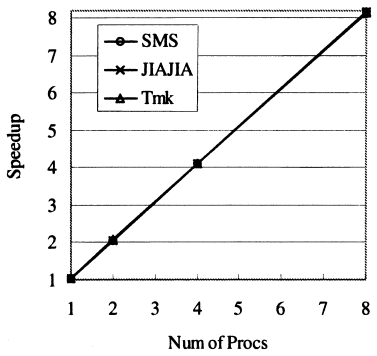


図 10 各種 SDSM における ep の性能向上比
Fig. 10 Speedup of ep on SDSMs.

6.3 計算機クラスタにおける OpenMP プログラムとの比較

SCore[24] を実装したクラスタにおいて，MpC と OpenMP（Omni）[23] との性能を比較した．Omni は，SCore で開発された SDSM である SCASH 上に実装されている．MpC が用いる SDSM はいずれもユーザレベルソフトウエアによるものであるが，SCore は，

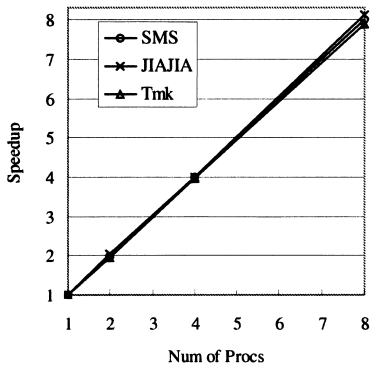


図 11 各種 SDSM における lu の性能向上比
Fig. 11 Speedup of lu on SDSMs.

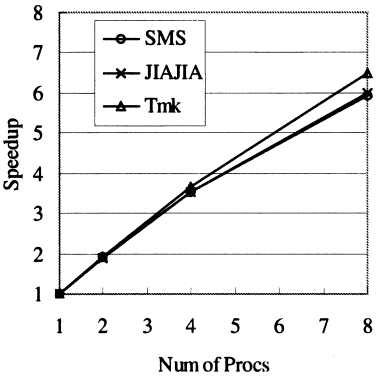


図 12 各種 SDSM における mm の性能向上比
Fig. 12 Speedup of mm on SDSMs.

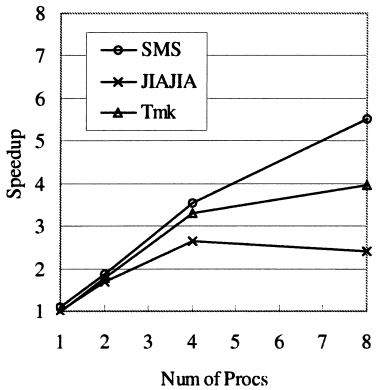


図 13 各種 SDSM における tsp の性能向上比
Fig. 13 Speedup of tsp on SDSMs.

OS カーネルに変更を加えたソフトウエア環境を前提にしており，システム管理者による OS のカーネル変更，インストールなどの設定が必要となる．

測定は**表 5**，**表 6** のような環境のもとで行った．MpC，Omni OpenMP の各コンパイラはどちらも下層には gcc を用いているので，mpcc で用いる gcc の version を，omni で用いている version2.96 に合わせて測定した．各プログラムのパラメータ，共有データサイズ，データ分散割付け方式を**表 7** に示した．ここ

表 3 ベンチマークプログラムとパラメータ
Table 3 Benchmark programs and parameters.

プログラム	パラメタ	shared データサイズ	バリア回数／プロセス	ロック回数／プロセス
ep	M=28,MK=10	44B	2	1
tsp	19cities(19b)	100MB	4	75-122
lu	2048 x 2048 double, 32bloks	34MB	135	0
mm	2048 x 2048 double	96MB	3	0

表 4 実行環境
Table 4 Experiment environment.

gcc version 2.96 -O3
SMS 0.4.16
JIAJIA2.2
TreadMarks 1.0.3.2

CPU	Intel PentiumIII-S 1.13GHZ
Memory	512MB
Network	Intel PRO/1000T 3Com SuperStack3 Switch
OS	RedhatLinux7.1.2 kernel 2.4.7.10

表 5 コンパイル環境
Table 5 Compilation environment.

	MpC	OpenMP
compiler	mpcc	omcc 1.6
OS	Red Hat Linux 7.3	
kernel	2.4.18-3	
gcc version	2.96	
optimization	-O3	

表 6 実行環境
Table 6 Execution environment.

	MpC	OpenMP
CPU	Intel PentiumIII 1.13GHz	
No of CPU	1	
memory	512MB	
OS	Red Hat Linux 7.2	
kernel	2.4.21-SCORE	
network	Giga	Giga, Myrinet
protcol	socket (UDP)	PM
SDSM	SMS	SCASH

表 7 測定プログラムのパラメータ
Table 7 Program parameters.

プログラム	パラメタ	shared データサイズ	shared データ割付方式
floyd	1024 x 1024 double	12MB	横バンド
laplace	1024 x 1024 double	16MB	横バンド
mandel	1024 x 1024 char	1MB	proc0 集中
mm	1024 x 1024 double	24MB	横バンド
galaxy	1000-body 10steps 100times	72KB	proc0 集中
ep	S	80B	proc0 集中

表 8 MpC と OpenMP の性能比較
Table 8 Performance of MpC and OpenMP programs.

(sec)

Language		MpC	OpenMP		
Compiler		mpcc	omcc		gcc
Network		giga	giga	myri	seq
floyd (1024x1024) 128MB	1	66.65	65.21	65.12	66.80
	2	35.01	69.80	46.57	
	4	19.42	44.94	25.30	
	8	11.38	41.62	14.71	
laplace (1024x1024)	1	3.08	3.10	3.07	3.08
	2	2.03	2.45	2.33	
	4	1.16	1.55	1.21	
	8	0.71	1.29	0.66	
mandelbrot (1024x1024) static	1	1.37	1.46	1.46	1.37
	2	0.76	0.93	0.86	
	4	0.40	0.53	0.50	
	8	0.22	0.29	0.26	
mandelbrot (1024x1024) dynamic	1	1.35	1.47	1.47	1.37
	2	0.80	0.87	0.81	
	4	0.48	0.51	0.43	
	8	0.25	0.35	0.23	
mm (1024x1024) blocking	1	6.78	10.14	10.43	8.15
	2	3.80	5.46	4.87	
	4	2.33	3.58	2.68	
	8	1.88	3.10	1.96	
mm (1024x1024) nonblocking	1	11.73	11.77	11.74	11.63
	2	6.39	17.83	11.26	
	4	3.77	17.89	6.62	
	8	2.73	25.68	4.75	
galaxy 1000-body 10steps 100times	1	9.00	9.02	9.04	8.73
	2	4.56	2.99	2.49	
	4	2.33	2.12	1.26	
	8	1.26	1.26	0.66	
ep class S	1	12.84	13.36	13.36	12.93
	2	6.42	7.08	6.73	
	4	3.21	4.30	3.43	
	8	1.61	2.50	1.80	

で横バンドというのは，図 3 (a) に示すような 2 次元配列を使用プロセス NPROCS 個で分割して割り付ける方式である．mandel，galaxy，ep はプロセス 0 に割り付けている．

各プログラムの測定実行時間を**表 8** に示す．MpC はギガビットイーサネット（以下 Giga）で UDP ソケットを使用し，Omni は Giga と Myrinet[25]（PCI32C, Lanai4）で PM を用いている．比較は，MpC プログラムを Giga で実行した場合と，OpenMP プログラムを Giga と Myrinet のそれぞれで実行した場合，そして参考として OpenMP プログラムを逐次コンパイラ（gcc2.96）でコンパイルした場合（表 8 seq 列）の，4 種の場合について行った．MpC の実行は SMS0.4.19a を利用し，OpenMP の実行は SCASH（SCore5.6.1）を利用している．

測定したプログラムは以下の 6 種で，SPMD モデルに基づく OpenMP の最も得意とする for ループを基本とした規則的構造を持つプログラムである．

6.3.1 floyd

Floyd の最短経路探索アルゴリズムを行うものである．大小比較など計算量は少ないが，共有データへの更新アクセス頻度は高い．パラレルリージョン内部でのループの回数が多いため，並列効果が得られている．このプログラムは，クラスタノード PC のメモリを 128 MB から 512 MB に増設した後，OpenMP の実行時エラーが発生し計測不可能であったため，このプログラムに関しては MpC，OpenMP ともメモリが 128 MB のときの計測値を示している．

MpC での測定結果は，OpenMP を Myrinet で実行したものと比べて約 25%の速度向上が見られる．OpenMP を Giga で実行した場合の速度向上率は良くない．

6.3.2 laplace

Laplace は 4 近傍値の平均を自身の値として更新するプログラムで，SCore でサンプルとして提供されている．共有データへのアクセス頻度は高く，計算量は少ない．

MpC の測定結果は，OpenMP を Myrinet で実行した場合と同程度であった．また，floyd と同様，OpenMP において Giga で実行した場合は速度向上率は低い．

6.3.3 mandel

mandel はマンデルブロー集合を作図するプログラムで，値が一定範囲に収束するまで計算を繰り返す．座標ごとに処理は独立で，共有データへのアクセスは，各プロセスの担当領域へ最後に 1 回書き込みをするだけで少ない．座標領域によって，収束までの繰返し数が異なるため，プロセスあたりの計算量にばらつきができる．このため，処理の終わったプロセスに動的に一定区間を割り当ててていくワークプール方式（dynamic）と，最初から静的に領域をプロセスに割り当てる方式（static）の 2 種を測定した．

結果は，Giga の OpenMP での実行時間は少し遅くなっているものの，MpC と OpenMP の Myrinet との大きな違いはない．このプログラムは，共有データサイズも小さく，共有データへのアクセス頻度も少ないため，OpenMP との差は少ない．

6.3.4 mm

mm は JIAJIA に添付されている行列積計算プログラムで，共有データへのアクセス頻度が高く，計算量は少ない．キャッシュ効果を考えて，ブロック化アクセ

スに書き換えたものと（blocked），単純アクセスのもの（nonblocked）の 2 種類について測定した．ブロック化した場合，MpC の結果は OpenMP の Myrinet よりも少し速い．Giga の OpenMP での実行時間は，8 ノード使用時で MpC より 60%以上遅い．またこのプログラムでは，OpenMP と MpC の 1 台と seq での実行時間にも差がある．同じ seq プログラムでも，SCore カーネルであるノード PC（表 6）では 8.15 sec であるが，通常の LINUX カーネルのホスト PC（表 5）で計測すると 7.2 sec くらいで異なる．ブロックサイズの効果が違うのかもしれない．

ブロック化しない場合では，MpC は明らかに OpenMP よりも高い速度性能を得ている．MpC ではほぼ台数分の効果が出ているのに対し，OpenMP では Myrinet でも低い速度向上率で，Giga に関しては実行時間が増えている．

6.3.5 galaxy

galaxy は星の間の引力を計算する多体問題である．共有データへのアクセス頻度は高いが，計算量が多いので並列効果が得られやすい．星の総数が 1000 個，シミュレート時間 100，ステップ時間 10 で，測定した．

結果は，MpC はほぼプロセス数比例の効果が得られた．一方，OpenMP はプロセス数以上の並列効果が出ているが，計測値にばらつきがある．SCASH における OpenMP の実装などが影響しているのか，原因は不明である．

当初，この galaxy や NPB の一部のプログラムのように通信負荷の高いプログラムを，SCore で 32 ビットバスのギガビットイーサを使って実行すると，非常に頻繁に SCoreOS のフリーズが発生し，計測が難航した．関係部署に問い合わせて我々のクラスタや通信環境に合わせた SCore の通信パラメータ，NIC ドライバのチューニングなどを行って，ようやく実行できるようにした経緯がある．現在は，フリーズすることは少なくなってきたものの，このプログラムに関しては，SCASH のエラーが発生することがあり，現在調査中である．

6.3.6 ep

ep は NPB2.3 の OpenMP C 版（NPB2.3-omp-C）とそれを MpC で書き直したものを用いた．共有データはほとんどなく，計算量が多いため高い並列効果が得られるプログラムである．結果は，全体的に MpC が OpenMP よりも高い性能を示している．

以上の結果から，ここで用いた for 文を主体とするデータ並列型のプログラムにおいても多くの場合，

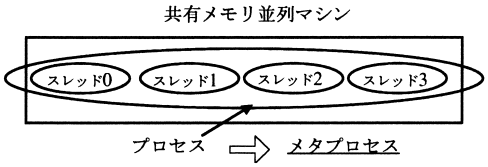


図 14 共有メモリ型マシン上でのメタプロセスモデル
Fig. 14 Meta Process Model on shared memory machines.

MpC は OpenMP と同等かそれ以上の性能を引き出せることが分かった．また OpenMP を速度の遅い Giga で実行させた場合には Myrinet に比べ十分な速度向上が得られないことが多い．また，共有データのサイズが大きく，アクセス頻度が高い問題に関しては，一般に MpC は OpenMP よりも高速であった．特に安価に準備できるギガビットイーサネットを用いたシステムでの実行では，多くのプログラムにおいて明らかに高速に動作することが示された．

MpC は，単純ループを主体とするデータ並列型ではない tsp などの非定形構造を持つプログラムの記述も容易であるが，OpenMP による非定形プログラムの例がほとんどないため，現在のところこのようなプログラムの比較は行っていない．

7. 共有メモリ型マシンにおける実装と評価

7.1 共有メモリ型マシンにおける実装

共有メモリ型並列マシンにおけるメタプロセスモデルの実装には pthread を使う．**図 14** のようにメタプロセスモデルにおけるメタプロセスを通常プロセスに，メタプロセス内プロセスをスレッドに対応させる．したがって，メタプロセス内のプロセス局所データは，pthread 実装ではスレッド局所データとして表現し，*shared* データのみをスレッド間で共有するデータとしている．

MpC コンパイラは，もともとの MpC プログラムの main 関数を個々のスレッドで実行させるサブルーチンに変換し，新しい main 関数を付け加える．この新規 main 関数で，指定数のスレッドを生成し，すべてのスレッドの実行終了を待つ．この main 関数には，もとの MpC プログラムにおける大域変数データをスレッド固有（thread specific）データ[22] に変換するコードや pthread のロック変数，条件変数などの初期化を行うコードなども含まれる．クラスタの場合との実装の違いを，実行実体と各階層データのそれぞれで比較したものを，**表 9** と**表 10** に示す．

7.2 pthread による MpC プログラムの実行

表 11 は，2 種の共有メモリ型マシンとクラスタで，前節で用いたいくつかの MpC プログラムを実行

表 9 メタプロセスモデルの実行実体の実装
Table 9 Execution entities in the Meta Process Model.

	SDSM，クラスタでの実装	pthread，共有メモリ並列マシンでの実装
一つの応用（メタプロセス）	プロセス群	プロセス
並列実行単位	1 プロセス	1 スレッド

表 10 メタプロセスモデルの階層データの実装
Table 10 Hierarchical data in the Meta Process Model.

メタプロセスモデルにおけるデータ型	SDSM，クラスタでの実装	pthread，共有メモリ並列マシンでの実装
shared	プロセス共有（SDSM 機能利用）	スレッド共有（global データ）
global	プロセス局所（global データ）	スレッド局所（local データ）
local	プロセス局所（local データ）	スレッド局所（local データ）

表 11 MpC と UPC の性能比較
Table 11 Performance of MpC and UPC programs.

(sec)

programs	Num of Proc	MpC pthread smp 2CPUs LINUX2.4.20 -6smp RedHat9	MpC pthread rs6000 4CPUs AIX5.2	MpC SMS GigaEther pc cluster LINUX 2.4.20-6 RedHat9	gnuUPC 3.2.3.5 smp 2CPUs Pentium3 LINUX2.4.20-6smp RedHat9
ep S class	1	10.82	10.45	11.19	10.15
	2	5.54	5.23	6.09	5.09
	4		2.65	2.56	
	8			1.28	mapping 1ele
galaxy 1000body 10 steps 100time	1	9.02	8.81	8.17	64.05/62.96
	2	4.52	4.42	4.53	32.89/31.5
	4		2.24	2.53	
	8			1.63	mapping div/0proc
mandel d 1024x1024 0.3<x<0.4 0.5<y<0.6	1	1.35	1.03	1.39	1.56/1.47
	2	0.68	0.53	0.87	0.79/0.75
	4		0.38	0.54	
	8			0.43	mapping 1ele/0proc
mm512 512x512 double array blocked	1	0.77	0.61	0.80	15.2/15.2/35.26/9.21
	2	0.40	0.31	0.47	8.16/7.69/17.83/4.64
	4		0.17	0.31	mapping
	8			0.29	vb/hb/1ele/0proc
mm1024 1024x1024 double array blocked	1	6.18	4.9	6.90	132.63
	2	3.17	2.47	3.47	66.35
	4		1.49	2.04	
	8			1.57	mapping 0proc

した結果である．表 11 の 3，4 列目は，SMP マシン（Pentium3，2CPU，Linux）と，共有メモリ型のサーバ（IBM RS6000，4CPU，AIX5.2）で，5 列目は比較のために，3 列目の SMP をノードとするクラスタで実行した場合の結果である．これにより，OS やアーキテクチャに依存せずに pthread をサポートする計算機システムでは，変更なしに MpC プログラムのコンパイルと実行ができることが分かる．

表 12 UPC，MpC，OpenMP の有効行数比較
Table 12 Number of lines in UPC, MpC and OpenMP programs.

プログラム	有効行数 MpC	UPC	OpenMP	OpenMP に対する比率 MpC	UPC
floyd	81		71	1.141	
laplace	83		66	1.258	
mandeld	138	127	106	1.302	1.198
mandels	112		98	1.143	
mm bloked	104	100	91	1.143	1.099
mm nonblocked	91	91	82	1.110	1.110
galaxy	191	200	180	1.061	1.111
ep	174	172	154	1.130	1.117
平均				1.161	1.127

7.3 共有メモリ型マシンにおける UPC プログラムとの比較

表 11 の右端の列は，3 列目と同じ SMP マシンにおいて，UPC（gnuUPC [10]）プログラムを実行した結果である．gnuUPC は現在 Linux，x86 に関しては SMP 版しか用意されていない．gnuUPC では，ep と mandel は MpC とほぼ同性能であるが，多体問題 galaxy や mm は，MpC に比べ非常に遅い．そこで，UPC で用意されている様々な共有データマッピング（hb：水平バンド割付け，vb：垂直バンド割付け，1ele：1 要素サイクリック割付け（UPC の default），0proc：THREAD0 への集中割付け）を試してみたが，いずれの方式でも gnuUPC の性能は著しく低い．これらプログラムはいずれも共有データアクセスが比較的多い．gnuUPC の x86 系 SMP への実装の詳細は明らかでないため，データ割付けの点で問題があるのか，コンパイラの翻訳の点で問題があるのかは不明であるが，現状では，MpC に比べ，共有データアクセスの多い応用に対しては劣るといわざるをえない．

8. MpC の生産性と記述性

8.1 UPC，OpenMP，MpC のプログラム行数の比較

プログラムの生産性や可読性などを示す 1 つの指標として，評価に用いたプログラムを UPC，MpC，OpenMP の各言語で記述した場合のプログラムの有効行数を調べた．有効行数とは，コメント行や空行などを除いたソースコードの行数である．**表 12** にそれぞれの有効行数と OpenMP に対する比率を示す．UPC と MpC は記述上の大きな違いはないが，それぞれ 13%，16%ほど OpenMP に比べて行数が増えることが分かる．

MpC ではプログラマによる明示的並列処理記述を基本にしているので，各プロセスの分担領域の範囲の

表 13 NPB プログラムの NPB3.0-SER に対する行数比
Table 13 Relative number of lines in NPB programs to NPB3.0-SER.

Benchmark	SER	HPF	JAVA	MPI	OpenMP	omni-C	MpC	C-SER	MPI	SER	プログラム
Version	NPB 3.0	NPB 3.0	NPB 3.0	NPB 2.4	NPB 3.0	(NPB 2.3)	(NPB 2.3)	(NPB 2.3)	NPB 2.3	NPB 2.3	平均
プログラム｜言語	F (C)	F	J	F (C)	F	C	C	C	F (C)	F (C)	
BT	1.0000	1.2011	1.6389	1.6817	1.0012	1.0360	−	1.0120	1.4658	1.0368	1.2592
CG	1.0000	1.0544	1.2951	2.0311	1.0233	0.9864	1.0932	0.9068	2.0117	0.9767	1.2643
EP	1.0000	−	−	1.3143	1.0286	1.1000	1.2429	1.0429	1.2857	0.9286	1.1347
FT	1.0000	1.0638	1.6393	2.4281	1.2951	1.3698	−	1.3242	2.3024	1.2823	1.5881
IS (C)	1.0000	−	0.8489	1.6333	−	0.9089	0.9200	0.8822	1.4778	0.9378	1.0870
LU	1.0000	1.1280	1.7835	1.3743	1.0305	1.0368	−	1.0121	1.3677	1.0333	1.2208
MG	1.0000	1.1192	1.7509	1.9374	1.0024	0.9374	1.0260	0.8985	1.9327	1.0708	1.2973
SP	1.0000	1.0461	1.8188	1.5397	1.0323	1.0233	−	0.9853	1.5321	1.0100	1.2485
平均		1.1021	1.5394	1.7425	1.0591	1.0700	1.0705	1.0080	1.6720	1.0345	

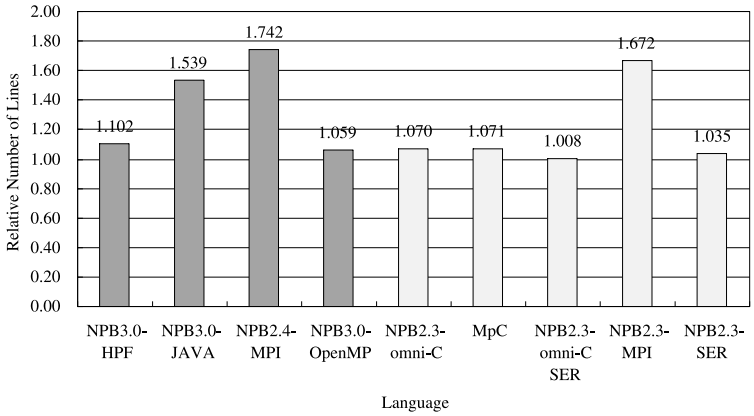


図 15 NPB プログラムの NPB3.0-SER に対する行数比
Fig. 15 Relative number of lines NPB programs to NPB3.0-SER.

計算や，合計値などを各プロセスから集めて計算するなどの reduce 関数や gather 関数などを提供していないため，そのような構文を組み入れた OpenMP に比べ，1 割強程度プログラム行数が増加する．ただし，このような定型的な処理は，MpC においてもマクロで提供することも可能なので，今後，定型文マクロの導入などにより，行数の増加を防ぎ，ユーザの生産性を改善することもできると考えている．

8.2 NPB におけるプログラム行数の比較

さらに，他の言語とも比較するために NPB を例に各言語で記述した場合のソースプログラムの有効行数を調べた．NPB3.0 には，JAVA，HPF，OpenMP などが含まれているので，NPB3.0-SER の逐次プログラムの有効行数を 1 にしたときの各言語の行数比率を**表 13** と**図 15** に示す．ただし，NPB3.0-MPI は存在せず，NPB2.4-MPI を使用することになっているので，MPI にはこれを用いた．また，NPB の SER，MPI，OpenMP は，IS だけが C で，それ以外は FORTRAN で記述されている．また，今回の測定に用いた OpenMP（NPB2.3-omni-c）と MpC（NPB2.3-omni-c をもとにしている）の比較をするために，NPB2.3-SER とともに，NPB3.0-SER に対する有効行数比も表 13 と図 15 の右側に示してある．

これを見ると，同じ内容を記述するにも，NPB3.0-SER に比べて，MPI は 1.742 倍もの記述が必要になることが分かる．また JAVA の記述量も 1.539 とソースコードは増加する．HPF は 1.102 とそれほど多くの増加はみられない．同様に OpenMP は 1.059 で NPB3.0 の中では最も少ない増加量になっている．

一方，図中右側の NPB2.3 については，NPB2.3-SER は 1.035 で，NPB3.0-SER よりも行数が多いことが分かる．OpenMP の C 版（NPB2.3-omni-C）の pragma 文を取り除いた逐次 C プログラム（NPB2.3-omni-C-SER）の比率も 1.008 で，NPB3.0-SER よりも行数が多いが，NPB2.3-SER の FORTRAN 版に比べ，C のほうが行数が少なくなっている．NPB2.3-MPI はやはり行数比が大きく 1.67 である．MpC と OpenMP の C 版（NPB2.3-omni-C）を比べると，

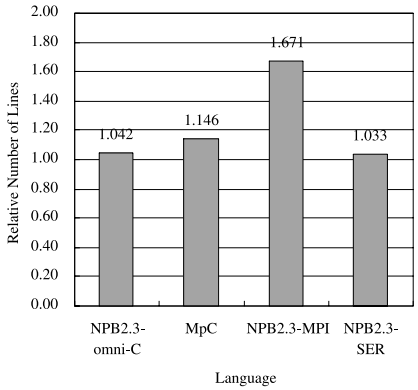


図 16 NPB2.3 プログラムの NPB2.3-omni-c-SER に対する行数比
Fig. 16 Relative number of lines in NPB2.3 programs to NPB2.3-omni-c-SER.

1.070 と 1.071 と差が少ない．しかし NPB ベンチマークセットのうち MpC で書き換えたプログラムが，現在のところ CG，IS，EP，MG の 4 つのみであるため，表 13 の一番右のコラムに示すように行数が増加率の高い FT などを含まない影響があると思われる．そこで，上記 4 つのプログラムのみに対する OpenMP の C 版（NPB2.3-omni-C）の行数の平均比率を調べると，1.012 となり，MpC の 1.075 よりも低くなった．

図 16 は，NPB3.0-SER（FORTRAN）を基準にするのではなく，NPB2.3-omniC-SER の C の逐次プログラムの有効行数を 1 としたときの NPB2.3-omni-C，MpC，NPB2.3-MPI（FORTRAN），NPB2.3-SER（FORTRAN）を比較したものである．この結果，C の逐次プログラムと比較しても，OpenMP は 1.044 の増加率で，NPB2.3-SER（FORTRAN）の 1.033 とほぼ同等の増加率といえる．MPI（FORTRAN）は 1.67 と高い．MpC は 1.147 と OpenMP よりは少し高い値になっている．

OpenMP と MpC は MPI は比べるとメッセージパッシング文が不要であるため，非常に行数が少なくなっている．OpenMP の逐次プログラムに対する増加率は約 5%程度で，さらに OpenMP に比べ MpC は 1 割ほど行数が増加する．

プログラム行数が即，プログラム記述の生産性や可読性に結びつくわけではないが，もともとのプログラムの内容やアルゴリズムを理解するうえで，また実際に並列プログラムを作りあげるうえで，行数の増加率が高いものは高コストといえる．その意味で，OpenMP，UPC，MpC は，MPI に比べ生産性，可読性とも向上している．

MPI では，OpenMP，UPC，MpC では隠蔽しているプロセス間通信を明示的にユーザが記述する必要があるため行数が増加するが，一方では，ユーザが通信の細かなチューニングができ，応用によっては性能の向上が期待できる．したがって，応用の性質と目的に応じ，生産性と性能のトレードオフにより，ユーザが好みの言語を選ぶことになるであろう．

8.3 MpC の記述性

今回測定に用いた omni と OpenMP 標準仕様と MpC を言語機能や実装システム運用の観点から比較してみた．omni は，OpenMP の標準仕様に加えていくつかの拡張機能を設けている．**表 14** にその一部の比較を示す．

OpenMP はもともと共有メモリ型の計算機で運用することを前提に設計されており，データはすべて共有されることになっているが，実際には，パラレルリジョンの出入りの際に，必要に応じて，システムによって暗黙に，あるいはプログラマによって明示的に，様々な private 構文を使用して，同じデータ名を用いながら，スレッド局所データの作成や共有データとスレッド局所データとのコピーなどを行う構造になっている．

一方，MpC では，プロセス局所データと共有データがまったく違うデータ（違うデータ名）として扱われており，コピーはプログラマによる明示的な指示だけによる．

また OpenMP では，共有メモリを前提にしているため，分散マッピングや共有データの動的割付けなど，クラスタ上での運用で望ましい機能も現在のところ標準仕様には入っていない．

OpenMP では，parallel for 文や reduce 文，schedule 文などの高機能構文が充実しているため，for ループやベクターデータを単純分割して並列化するなどの処理は，記述が容易で定形化しており，ユーザには使いやすい．

しかしこれらの処理はいずれも SPMD 型のスレッド fork/join を基本とするため，参加するスレッド数の変更はできるものの，MPI や MpC で書けるような，参加プロセス数の違う複数のプロセスグループを構成して，並列にそれぞれに異なる処理をさせたりするような自由度の高い処理構造を書くには適さない．すなわち MPMD 型，非対称型の処理には不向きといえる．

8.4 実装システムの運用性

MpC の実行環境は，MPICH や LAM などの MPI システムと同様に，ユーザレベルソフトウエアで構築されているため，一般ユーザによるインストール，運用が容易である．また MpC コンパイラも，ターゲッ

表 14 MpC と OpenMP の言語機能と実装システムの比較
Table 14 Language features and implementation systems of MpC and OpenMP.

		MpC	OpenMP 標準仕様 (ver.2.0)	Omni
言語機能	プログラミングモデル	分散共有メモリモデル プロセスモデル	共有メモリモデル スレッドモデル	同左
	処理モデル	SPMD/MPMD 型，非対称処理など	SPMD 型向き スレッド fork/join モデル	同左
	共有データ	shared と明示されたデータのみ共有	すべてのデータは共有	同左
	共有データの動的割り付け機能	動的割り付けあり	なし (malloc のみ)	拡張機能：共有データ用の割付関数増設
	データ分散マッピング	柔軟な機能あり	なし	拡張機能：一次元に限ったマッピングのみ
	実行実体内の局所データ	プログラマの指示した shared データ以外は，プロセス局所データ	parallel region に入る都度，private 化する共有データ名を指示	同左
	局所データと共有データのコピー，データ値の反映	コピーの必要がある場合のみプログラマが明示的に記述 局所データと共有データは階層スコープで異なるデータ、変数として扱われる	同じ変数名における共有－局所データコピー，データ値の反映などを，private, threadprivate, lastprivate, firstprivate などを利用して行う	同左
	共有データのメモリ一貫性	緩和型メモリ一貫性モデル バリア同期，ロックを用いてメモリ一貫性をとる時期を明示的にプログラマが指示	逐次型メモリ一貫性モデル 実際上は，共有－局所データとのやりとり，データ値反映と時期に配慮する	同左
	同期	必要な箇所のみにプログラマがバリア同期を記述 暗黙の同期はなし	parallel regiion や各種構文内で暗黙に挿入，性能低下を避けるには、同期回避指示を毎回記述する 明示的同期挿入もあり	同左
	高機能構文	なし	parallel for, reduction, schedule など多種	同左
実装システムと方式	実装システムの移植性	SDSM 版：多種な OS，アーキテクチャへの移植性が可能. pthread 版：アーキテクチャ，OS に非依存	－	OS の種類，バージョンに依存し，その都度開発変更が必要
	実装システム運用・保守の容易さ	ユーザレベルソフトウエアであるため一般ユーザによるインストール、運用、保守が容易 エラー時においても他のユーザへの悪影響がない	－	OS カーネルの変更が必要，システム管理者によるインストール，運用，保守が必要 エラー時には OS ダウンなどの危険性あり

トマシンのアーキテクチャに依存する部分をライブラリ関数で吸収する構造になっており，コンパイラの処理は汎用である．このため，ターゲットマシンごとにコンパイラを開発する必要がなく，移植性は高い．

一方，今回用いた omni（SCore）は OS カーネルを変更する必要があるなど，システム管理者によるインストール・運用が必要である．またシステムの安定した運用環境を作りあげるには，個々の実装ハードウェアに依存したチューニングもときとしては必要で，多くの作業を要する．したがって，一般ユーザが容易にインストールして使える状況にはなっていない．また，OS に依存していることから，OS の種類の違いだけでなく，OS の version 変更にも毎回のシステムの変更と対処が必要で，利用者にとっては，そのつど，カーネル変更や設定などの作業が発生し，実装システム開発者にとっても OS 変更ごとに開発が必要になる．

9. おわりに

異なる SDSM や pthread を使用して，MpC プログラムを変更なしにクラスタや共有メモリ型並列マシン上で実行できることを示した．ここで用いた SDSM は様々な OS，アーキテクチャ上での稼働が確認されており，ここでは試せなかった様々な並列マシンについても，MpC プログラムをそれぞれの SDSM プログラムや pthread プログラムにソース変換することにより，実行させることが可能である．

また，MpC 言語と類似した API を持つ UPC や，共有メモリモデルを基本とする OpenMP と比較しても，同等以上の性能が得られた．また，現在，並列プログラミングに広く用いられている MPI に比べ，プ

ログラムの生産性や可読性に優れている

さらに MpC プログラムの実行環境は，OS の変更や特別な通信デバイスを用いることなく，一般ユーザレベルの権限で容易に構築できる．

メタプロセスモデルと MpC は，その性能とプログラム生産性の点において，従来のメッセージパッシングモデルと純粋な共有メモリモデルの中間的な特徴を持つ並列プログラミングモデルとして，1 つの選択肢になりうる可能性を示した．

謝辞 本学大学院学生片野真吾氏（現日立）と渡辺義人氏の MpC コンパイラ開発における貢献と，OpenMP との性能比較における同長尾知幸氏の貢献に，深謝いたします．

参 考 文 献

1) Midorikawa, H.: Meta Process Model: A New Distributed Shared Memeory programming Model, *Proc. 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp.295–300 (2003).
2) 緑川，飯塚：ユーザレベル・ソフトウエア分散共有メモリ SMS の設計と実装，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.42, No.SIG9 (HPS 3), pp.170–190 (2001).
3) Keleher, P., Dwarkadas, S., Cox, A.L. and Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. Winter 94 Usenix Conf.*, pp.115–131 (1994).
4) http://www.cs.rice.edu/ willy/TreadMarks/overview.html
5) Hu, W., Shi, W. and Tang, Z.: JIAJIA: An SVM System Based on A New Cache Coherence Protocol, *Proc. High Performance Computing and Networking* (*HPCN'99*), LNCS 1593, pp.463–472 (1999).
6) http://www.ict.ac.cn/chpc/dsm/index.html
7) Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E. and Warren, K.: Introduction to UPC and Language Specification, CCS-TR-99-157, IDA Center for Computing Sciences (1999).
8) El Ghazawi, T. and Cantonnet, F.: UPC performance and potential: A NPB experimental study, *Proc. IEEE/ACM Super Computing2002* (2002).
9) UPC. http://upc.gwu.edu/
10) gnuUPC. http://www.intrepid.com/upc/
11) Adve, S.V. and Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Vol.29, No.12, pp.66–76 (1996).
12) Keleher, P., Cox, A.L. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th Symp. on Computer Architecture*, pp.13–21 (1992).
13) Iftode, L., Singh, J.P. and Li, K.: Scope Consistency: A Bridge between Release Consistency and Entry Consistency, *Theory of Computing Systems*, 31, pp.451–473 (1998).
14) Jin, H., Frumkin, M. and Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, NAS Technical Report NAS-99-011 (1999).
15) Takahashi, D., Sato, M. and Boku, T.: Performance Evaluation of the Hitachi SR8000 Using OpemMP Benchmarks, *Workshop on OpenMP, Int'l Workshop on OpenMP* (*WOMPEI'02*) (2002).
16) Basumallik, A., Min, S.-J. and Eigenmann, R.: Towards OpenMP execution on software distributed shared memory systems, *Int'l Workshop on OpenMP* (*WOMPEI'02*) (2002).
17) Lu, H., Hu, Y.C. and Zwaenepoel, W.: OpenMP on Networks of Workstations, *Proc. IEEE/ACM Super Computing 98* (1998).
18) Hu, Y.C., Lu, H., Cox, A.L. and Zwaenepoel, W.: OpenMP for Networks of SMPs, *Journal of Parallel and Distributed Computing*, Vol.60, No.12, pp.1512–1530 (2000).
19) 小島，佐藤，原田，石川，朴，高橋：Ethernet によるクラスタ上での分散共有メモリ OpenMP Omni/SCASH の性能評価，情報処理学会研究報告，Vol.2002, No.80, HPC 91-21, pp.119–124 (2002).
20) 佐藤，草野，佐藤：OpenMP 向けコンパイラ支援ソフトウエア DSM，情報処理学会論文誌，Vol.42, No.4, pp.788–801 (2001).
21) OpenMP. http://www.openmp.org
22) Butenhof, D.R.: *Programming with POSIX Threads*, Addison-Wesley (1997).
23) http://phase.hpcc.jp/Omni/home.html
24) http://www.pccluster.org/index.html.en
25) http://www.myri.com/

付 録

A.1 MpC 文法規則

C Programming Language（2nd Edition, by Kernighan, B.W. and Ritchie, D.）の文法規則の追加/変更部分のみを以下に示す．

storage-class-specifier:
auto | register | static | extern | typedef | shared

declarator :
pointer$_{opt}$ direct-declarator
| pointer$_{opt}$ direct-declarator mapping-specifier

mapping-specifier :
:: div-information
| :: owner-information
| :: div-information owner-information

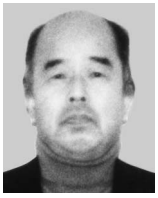
div-information :
[ ]
| [constant-expression]
| div-information [constant-expression]

owner-information :
(constant-expression)
| (constant-expression , constant-expression)

(平成 16 年 7 月 23 日受付)
(平成 16 年 11 月 24 日採録)

緑川 博子（正会員）

慶應義塾大学工学部電気工学科卒業．日本電気（株）C & C システム研究所にて，データフロー型プロセッサ，マルチプロセッサの研究開発，パターン認識，並列処理応用研究開発に従事．現在，成蹊大学工学部経営・情報工学科助手．並列処理，並列システムソフトウェア，並列アルゴリズム，並列プログラミングモデル等に興味を持つ．IEEE，電子情報通信学会各会員．



飯塚 肇（正会員）

1939 年生．1964 年東京大学大学院数物系研究科応用物理学専攻修士課程修了．1977 年東京大学より工学博士（情報工学）の学位を取得．三菱電機（1964〜1966 年），電子技術総合研究所（1966〜1980 年）においてコンピュータアーキテクチャ，並列処理等の研究開発に従事．1980 年 4 月より成蹊大学工学部教授．著書：『電子計算機 2』（電子情報通信学会大学シリーズ），『コンピュータシステム』（オーム社）等．電子情報通信学会，IEEE，ACM 各会員．