並列プログラミングライブラリ PPElib における適応型メモリバッファリングの実装と評価

手塚 忠則[†,††] 末吉 敏則[†††] 有田 五次郎[††]

我々は，ローカルエリアネットワーク接続されたワークステーション（WS）群を利用して分散共有メモリモデルの並列計算機の機能を提供するクラスターコンピューティング環境 DSE（Distributed Supercomputing Environmnet）を実装し，クラスタ環境を用いた並列処理に関する研究を行っている．本研究では，並列アプリケーションを記述するために，C++によるクラスライブラリ（PPElib）を提供し，これによりプログラミングモデルを制限しプログラムの記述性を向上を図っている．また，ライブラリの提供するプログラミングモデルでは，並列実行部の各 WS での共有メモリアクセスの独立性が高く，複雑なコヒーレント処理なしにバッファリング制御が効果的に行うことが可能であるという点に着目し，各仮想プロセッサでの共有メモリの適応的なバッファリング手法について研究している．本稿では，このクラスライブラリによる共有メモリのバッファリング手法について述べ，評価実験の結果を示した．実験の結果，ウェーブレット変換のような画像処理アプリケーションにおいては，適応型のバッファリング制御が効果的であることが確認できた．

Implementation and Evaluation of the Adaptive Memory Buffering Scheme for the Parallel Programming Library

Tadanori Tezuka,[†,††] Toshinori Sueyoshi[†††] and Itsujirou Arita[††]

Distributed Supercomputing Environment (DSE) is a distributed shared-memory based cluster computing environment which was implemented on a cluster of workstations connected by a local area network. In DSE, we have provided the C++-based class library which called PPElib for writing the parallel applications. This class library restricts the programming model and it does simplify the parallel programming. In this restricted programming model, each Light-Weight process (LWP) within a task have less dependencies. So we can use the shared-memory-buffering scheme without a complicated buffer-coherence mechanism. In this paper, we describe the adaptive buffering scheme that is implemented in our class library, and show the experimental results of parallel wavelet transformation with adaptive buffering. This result has shown us the adaptive buffering is fifth-times faster than normal buffering.

1. はじめに

我々は，高速なネットワークにより接続された計算機群を利用し並列処理を行うクラスタコンピューティング環境である DSE（Distributed Supercomputing Environment）を構築し，クラスタコンピューティングに関する研究のためのテスト環境として利用している[1),2)]．

DSE は，LAN 接続された計算機群を利用して DSM 環境を提供する．DSE では，現在広く普及している Ethernet（10 Mbps，100 Mpbs）程度の LAN で十分な並列実行性能を達成できるように，共有メモリへのアクセス方式として read/write 関数による I/O インタフェース方式を用いている．しかしながら，この方法はユーザに共有メモリアクセスやデータ配置に対する負担を強いるため，プログラミングの利便性を考慮した場合は，ローカルなメモリと同様なアクセスを可能とするメモリ R/W 方式を採用することが望ましい．

クラスタ環境で，メモリ R/W 方式による共有メモリアクセスを効率良く実現するためには，ネットワー

† 松下電器産業株式会社九州マルチメディアシステム研究所
Kyushu Multimedia Systems Research Laboratory, Matsushita Electric Industrial Co., Ltd.
†† 九州工業大学情報工学部知能情報工学科
Department of Artifical Intelligence, Kyushu Institute of Technology
††† 熊本大学工学部数理情報システム工学科
Department of Computer Science, Kumamoto University

クの遅延を隠蔽するソフトウェアキャッシュ機構が必須である．このようなキャッシュ機構をソフトウェアで実現する手法としては，コンパイラによりコードを埋めこむ UDMA [3]や，ユーザレベルのソフトウェアのみでソフトウェアキャッシュを提供する TreadMarks [4]や Lemuria [5]などがある．このような方式では，共有アクセス発生時の一貫性保証のために，複雑なコヒーレントプロトコル[4),6),7)]を利用し，キャッシュ制御のための通信量の削減を図っている．たとえば，TreadMarks が実装する Lazy Release Consistency（LRC）モデルでは，コヒーレント処理を同期時まで遅らせることで通信処理を大幅に削減し，一貫性保証のための処理量を大きく削減している．また，コンパイラを用いた手法では，プログラムへのキャッシュ制御コードの追加により，一貫性保証に要する処理を削減している．このような，ソフトウェアキャッシュを提供する DSM 環境では，セグメントあるいはページというまとまったメモリ単位でメモリ管理を行うのが普通である．

しかしながら，DSM のセグメントあるいはページは，実行するアプリケーションのメモリアクセスの単位とは異なるため，キャッシュ単位はアプリケーションのメモリアクセスの特徴とは無関係になる．このため，アプリケーションによっては，無駄なデータのキャッシングが行われ，効率の良いキャッシングが行えない場合がある．たとえば，画像の垂直方向に走査するフィルタ処理などは，このようなアプリケーションの典型である．

我々は DSE 向けに，オブジェクト指向の並列アプリケーション記述ライブラリ PPElib を提供しており，このライブラリで記述したアプリケーションでは，プログラムの性質を把握しやすいという特徴がある．本研究では，これを利用してアプリケーションのメモリアクセスの特徴に合わせた共有メモリのバッファリングを試みた．具体的には，まず，タスクの前後で同期処理が自動的に行われることを利用して，このプログラム区間に対して Release Consistency（RC）モデルに基づく簡易なコヒーレント処理によるバッファリング機構を実装した．さらに，DCT やフィルタ処理といった画像処理アプリケーションでは，1 つのタスクに含まれるの軽量プロセス（LWP）のアクセスパターンは似ているという特徴を利用し，これまでに実行した LWP の共有メモリへのアクセス履歴を用いて，次に実行される軽量プロセスの共有メモリアクセスを予測し，バッファリングする領域を変化させる適応的なバッファリングを行い，ミス率を削減した．

本稿では，プログラミングモデルとクラスライブラリの概要および，適応型バッファリング方式について説明し，ウェーブレット変換を行うアプリケーションを用いた評価実験の結果について述べる．

2. クラスライブラリ概要

2.1 プログラミングモデル

並列アプリケーション記述ライブラリ（PPElib）のプログラミングモデルでは，プログラムは大きくアプリケーションとタスクから構成される．**図 1** は，クラスライブラリの提供するプログラミングモデルを示したものである．図 1 に示すように，アプリケーションではタスクの実行順序やタスクの実行結果による条件分岐などの，逐次的な処理を行う．タスクでは，たとえば行列積の演算などのように，データ依存関係が少なく複数の処理に分割できる処理を行う．タスク中の分割された処理は軽量プロセス（Light Weight Process; LWP）として仮想プロセッサに割り当てられ並列実行される．なお，仮想プロセッサとは DSE が提供する仮想的なプロセッサである．

このように，本モデルにおいては，タスクは内部に多数の LWP を含み並列実行される部品，アプリケーションはこの部品の実行順序を決定する逐次的な部分となる．なお，タスク内の LWP は，タスクにより自動的に各仮想プロセッサに割り当てられ，実行される．このため，LWP がどの順で実行されるのかは，このモデルにおいては保証されない．したがって，このモデルでアプリケーションを記述する場合には，LWP 間で共有メモリへの書き込みが衝突しないようにしなければ，アプリケーションの動作の保証はされない．これが，このモデルでプログラムを記述する場合の制約事項になる．

しかしながら，これでは LWP 内で共有変数の書き換えが行えないため，データ依存がある処理を記述することができないという問題がある．そこで，他の

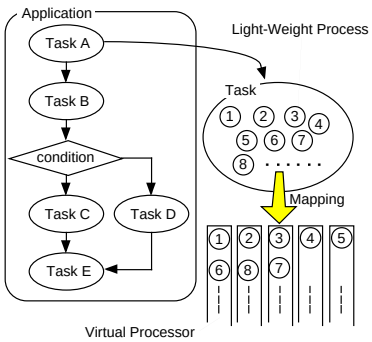


図 1 プログラムモデル
Fig. 1 A programming model.

```
void myTask::Process()
{
  int line   = mCurrentProcess;

  // (1)line 行に対応する処理の読み出し
  // (2) 読み出したデータに対する処理
  // (3) 結果の書き込み
  return;
}
```

図 2 Process メソッドの例
Fig. 2 A example of A Process method.

LWP との排他制御を可能とする Synchronized メソッドを用意し，これを用いて呼び出されたメソッド（関数）は，タスクオブジェクト中の LWP の 1 つだけが実行することが保証されるようにした．これにより，LWP 内で共有変数の書き換えを可能にしている．

このモデルは，並列実行可能な部品を用意しておき，これを組み合わせて処理を行うスタイルが適合する並列アプリケーションの記述に有効であり，並列プログラミングに付随する LWP のプロセッサへの割当てや同期処理を記述しなくてよいという利点がある．

このように，プログラムを部品として用意しておき，それを利用してプログラミングするというスタイルはオブジェクト指向言語と相性が良い．そこで，本研究では，ライブラリの実装にオブジェクト指向言語である C++言語を利用した．具体的にはアプリケーションとタスクを，それぞれが必要とする機能を含んだ基本クラスとして提供し，ユーザはこれを継承してプログラムを行うようにした．継承を利用するプログラミングは，ユーザがモデルから外れたプログラムを記述することを抑制できるという利点があり，モデルにそったプログラミングを容易にする．さらに，必要であれば，デフォルトで用意されているプロセッサ割当てなどをユーザが書き換えられるという特徴もある．これを利用すれば，LWP の割当て方針変更などがユーザ側で可能になる．

なお，LWP の記述は，タスククラスを継承したユーザ定義クラスの Process メソッドを記述することにより行う．Process メソッドでは，どの LWP を実行するかを，タスク内の LWP に一意に割り振られた LWP 番号をデータメンバ mCurrentProcess により参照し，決定する．この Process メソッドの記述は，たとえば **図 2** のようになる．

2.2 共有メモリアクセス

PPElib の共有メモリクラスでは，I/O 方式による共有メモリへのアクセスと，RW 方式による共有メモリアクセス方式の 2 つを提供する．**図 3** は，この 2 つの方式によるアクセスの違いを示したものである．

```
PPE_Memory<char> theMem(1024);
double theDat[10];
theMem.Read(0, sizeof(double)*10, theDat);
theDat[0] = theDat[1] + theDat[2] + ...
   :
   :
theMem.Write(0, sizeof(double)*10, theDat);
```

a) A shared memory I/O interface

```
PPE_Memory<double> theMem(1024);
theMem[0] = theMem[1] + theMem[2] + ...
   :
   :
```

b) A shared memory R/W interface

図 3 共有メモリアクセス方式
Fig. 3 The shared memory access types.

I/O 方式は Read/Write メソッドにより共有メモリのデータをローカルメモリに移してから処理を行う方式で，プロセッサ性能に対して共有メモリアクセス速度が極端に遅い DSE のような環境で並列処理効率の向上には効果的な方式である．一方，RW 方式は，共有メモリを通常のローカルメモリのようにアクセスするもので，共有メモリをローカルメモリ感覚で操作できるため，ユーザの利便性という点からは優れた方式であるが，共有メモリアクセス速度が著しく遅い環境では，I/O 方式に比べ通信が多数発生するため，並列処理による速度向上を得にくいという欠点がある．

このような，RW 方式の欠点を克服する方法としては，データと処理する LWP の配置を静的に決定しておくという方法がある．しかし，PPElib では仮想プロセッサの処理能力に応じて動的に LWP の割当てを行うため，静的にデータと LWP をプロセッサに割り当てておくことはできない．

他の方法として，キャッシュを用いる方法が考えられる．DSE でも，ソフトウェアキャッシュを実装した Cached DSE[8] による評価実験の結果から，高速ネットワーク環境での共有メモリのキャッシング効果が確認されている．この Cached DSE は，DSE カーネル側に Sequential Consistency（SC）および RC モデルに基づくメモリコンシステンシ機構を実装し，コヒーレンスプロトコルを用いて仮想プロセッサ間のデータの一貫性を保つという手法がとられていた．

本研究では，タスクが，複数の LWP によりある意味のある処理を行う固まりであるということ，および，画像処理などのタスクではタスク内の LWP のメモリアクセスパターンが類似するという特徴を利用して，一貫性を保つ処理を簡略化すると同時に，バッファ領域を可変にする適応型バッファリングを行った．

2.3 クラスタコンピューティング環境 DSE

PPElib でのバッファリング手法を説明する前に，

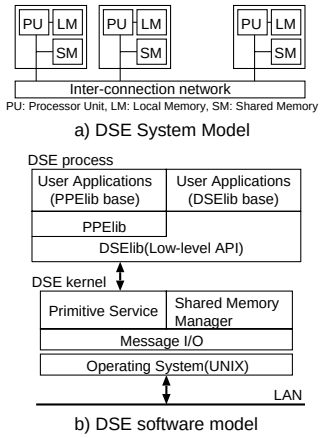


図 4 DSE 概要
Fig. 4 DSE system model and software model.

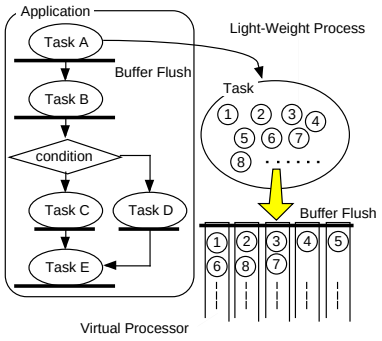


図 5 バッファフラッシュ操作挿入部
Fig. 5 The insert points of shared memory buffer flush function.

PPElib の動作環境である DSE について簡単に説明する．**図 4** は，DSE の提供する並列計算機のシステムモデル (a) と DSE のソフトウェア構成 (b) である．DSE の提供する並列計算機のモデルでは，プロセッサ要素は，プロセッサ（PU），各 PU のみがアクセス可能なローカルメモリ（LM），すべての PU がアクセス可能な共有メモリ（SM）により構成され，これが相互結合網により接続されている．共有メモリは，各プロセッサ要素に分散配置されており 4 Kbyte 単位のページで管理されている．なお，PU による共有メモリのアクセスは，プロセッサ要素番号とプロセッサ要素内アドレスによる 2 次元的なアドレッシングにより行われる．DSE では，相互結合網を LAN で，プロセッサ要素を UNIX 上で動作するソフトウェア（仮想プロセッサ）により実現する．なお，LM は UNIX プロセスが確保するメモリ空間としている．

図 4 (b) は，DSE の仮想プロセッサのソフトウェアモデルである．DSE は，大きく分けて DSE カーネルと DSE プロセスの 2 つから構成される．DSE カーネルでは，LAN を経由したメッセージ送受信，共有メモリ管理，およびプリミティブサービスを行う．また，ユーザのアプリケーションは DSE プロセス内で動作し，低レベル API の DSElib，またはクラスライブラリ PPElib を利用して DSE カーネルと通信する．なお，DSE カーネルによる DSE プロセスのスケジューリングは FCFS（First-In First-Served）により行われる．また，計算機間の通信には TCP/IP を利用している．

本研究でメモリバッファリング機能を導入したのは，図 4 (b) のクラスライブラリ（PPElib）部分である．また，適応型バッファリング機能実現のために，DSE カーネル内の Primitive Service にも新たに機能を 1 つ追加した．なお，このバッファリング機能の追加は，ユーザのプログラミングインタフェースの変更なしに実現した．

3. バッファリング機能の実装

3.1 プログラミングモデルによるアクセス特性

PPElib では，タスクは複数の LWP を含み，これらの仮想プロセッサへの割当てと終了同期を行うことは先に述べたとおりである．タスクでは，タスクの開始（各プロセッサへの LWP の割当て開始）と，タスクの終了（割り当てる LWP がなくなったときの同期処理）時に，内部で同期処理を行う．したがって，タスク開始を aquire，タスク終了を release と考えると，タスクでは，RC に基づいたメモリコンシステンシモデルが適用できる．また，LWP 内の排他制御部である Synchornized メソッドにおいても，ユーザのメソッド実行前後で同期処理が呼び出されるため，RC に基づいたメモリコンシステンシモデルが適用できる．

3.2 バッファ制御

以上より，一時バッファの制御としては，タスクの終了時と Synchoronized メソッドの前後で，バッファリングされたデータのコヒーレント処理を適切に行えばよいことが分かる．ここでは，この処理としてバッファフラッシュを用いた．

図 5 は，図 1 のプログラムのどの位置で一時バッファの破棄（バッファフラッシュ）を行うのかを示したものである．本実装においては，バッファフラッシュは図 5 に示すように，仮想プロセッサへ LWP を割り当てる前と，タスクの終了時に挿入した．また，LWP 内の Synchronized メソッドによるメソッド実行では，メソッドの入口でバッファフラッシュを行うようにした．以上のバッファフラッシュ挿入により，適切なバッ

ファ制御が行われるので，バッファのコヒーレントを保つために仮想プロセッサ間で通信を行う必要はなく，比較的軽い処理でバッファ制御が実現できる．

4. 適応型バッファリング

4.1 適応型バッファリングの概要

本研究では，プログラミングモデルの特徴を利用した一時バッファの制御に加え，タスク内の LWP のメモリアクセスパターンは似ている可能性が高いという特徴を利用して，LWP のメモリアクセスログから，バッファリングする領域を変化させていくという適応型バッファリングを実装した．

平滑化フィルタや DCT 処理などに代表される典型的な画像処理では，画像を一定の領域に区切って処理を行うことが多い．たとえば，JPEG や MPEG に代表される画像圧縮では，マクロブロック（MB）と呼ぶ 8×8 の画像領域に対して，DCT，Q，IQ，IDCT，VLD といった処理を順に行う．このような，画像圧縮アプリケーションを PPElib を用いてプログラミングする場合，まず，DCT，Q，IQ，IDCT といった処理をタスクとして用意しておき，アプリケーション（クラス）でこれらの実行順序を制御することになる．このとき，各タスクの LWP を 1 つの MB に対する処理とすれば，各 LWP の処理はほとんど同じになり，各 LWP の差異は処理する MB 位置（画像領域）の違いだけになる．言い換えれば，各 LWP の差異は，処理するメモリの先頭アドレスの違いになる．

画像圧縮に限らず，代表的な画像処理である平滑化フィルタやディザリング[9]，ウェーブレット変換[10]を考えた場合も，画像の水平ラインまたは垂直ラインを単位として，または，$N \times N$ ピクセルを単位として処理が行われるため，これを LWP とすれば，各 LWP の処理の差は先頭アドレスの差となる．

また，MB を単位として LWP とした場合，画像圧縮方式 H.263 で利用される CIF（352 × 288）サイズの画像の処理を行うタスクは，396 個の LWP を含むことになる．このタスクを，LWP 数より少ないプロセッサを利用して並列実行する場合，1 つのプロセッサに何個かの LWP が割り当てられ実行されることになる．

したがって，仮想プロセッサで実行した LWP の共有メモリアクセスパターンの収集と予測を行い，次に割り当てられた LWP 実行時に予測結果をもとにバッファリングを行うことで，バッファのヒット率を向上させることが可能になる．

適応型バッファリングは，このようなタスクに含まれる LWP の特徴を利用して，バッファリング領域を変化させる方式である．以下，実装した適応型バッファリングについて詳しく説明する．

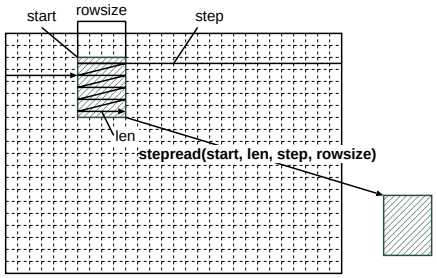


図 6 2 次元メモリアクセス
Fig. 6 The 2-D memory access.

4.2 DSE へのプリミティブ追加

先に説明したように，画像処理では，水平，垂直ラインおよびブロック領域に対する処理が行われる．したがって，画像処理に対するバッファリングでは，**図 6** に示すような矩形領域を一度のアクセスで読めば効果的である．このような共有メモリの読み出しを可能にするために，先頭位置 (start)，読み出し長 (len)，列サイズ (rowsize)，読み出しステップ (step) をパラメータとして与えることで，図 6 のような矩形領域を一度に読み出せる stepread プリミティブを DSE に実装した．たとえば，start=0，len=64，rowsize=8，step=640 と指定すれば，640 × 480 ピクセルの画像の (0,0) の座標から 8 × 8 ピクセルのブロックを取り出すことができる．以降に説明する適応型バッファリングでは，この stepread を利用して共有メモリからデータを読み出す．

4.3 ログ収集処理

適応型バッファリングでは，LWP 実行中の共有メモリアクセスのログを収集し，これをもとに次の LWP のメモリアクセスを予測する．図 6 に示すような矩形を予測してデータを獲得しようとした場合，先ほどの stepread のパラメータと同じ start，len，step，rowsize の情報が必要になる．しかしながら，一度に読み出すデータサイズをバッファサイズに固定した場合には，これらのパラメータのうち len は必要なくなる．したがって，アクセス予測のために必要となるパラメータは，バッファリング開始アドレス (start) を除いた step と rowsize になる．

アクセス予測のためのログ収集では，アクセス間隔を示す (STEP) と，rowsize に対応するデータ長 (LEN) と，そのパターンでアクセスした回数を記録する．アクセスした回数としては，このパターンでアクセスし

表 1 共有メモリのアクセスデータの項目（フィールド）一覧
Table 1 A list of shared memory access data's fields.

項目名	説明
STEP	直前のアクセス位置からの相対アドレス STEP = SA − PSa
LEN	データ長 Len = EA − SA
COUNTER A	このパターンでのアクセス回数
COUNTER B	プログラム中で READ を発行した回数

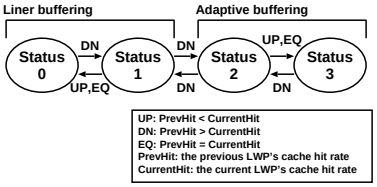


図 7 バッファリング切替えダイアグラム
Fig. 7 The Liner/Adaptive buffering switching state diagrams.

た回数 (COUNTER A) と，プログラム中での読み出しが発生した回数 (COUNTER B) を収集する．カウンタが 2 つあるのは，後述するように，近傍へのアクセスをまとめる処理があるためである．たとえば，この処理により 2 バイト間隔に 4 回アクセスした場合は，LEN = 8 のアクセスとしてまとめられる．この場合，COUNTER A は 1 に，COUNTER B は 4 になる．

以上で説明した記録される情報をまとめたものが，**表 1** である．表中の SA はアクセスの先頭アドレス，EA は末尾アドレス，Len はアクセスデータ長，PSa は前回のアクセスアドレスを示している．また，以降の説明では，PLen を前回アクセスしたデータ長，PEA を前回のアクセス末尾アドレスとする．

画像のフィルタリング処理では，ある画素を中心として上下左右の数ピクセルを交互にアクセスする．この場合，単純に STEP と LEN を記録するだけでは，連続したアクセスはいくつかのアクセスパターンに分割されてしまう可能性がある．たとえば，$x[0]+x[-1]+x[1]$ と $x[-1]+x[0]+x[1]$ は，基本的に同じアクセスパターンを持つプログラムであるが，単にログ収集した場合は前者は 3 つのアクセスに分割されてしまう．

そこで，ログ収集では直前のアクセスと現在のアドレスの関係が以下のどちらかの条件を満たす場合には 2 つのアクセスを 1 つにまとめる処理を行う．

$$\mathrm{SA} - \mathrm{PSa} + \mathrm{PLen} \leq \delta$$
$$\mathrm{SA} + \mathrm{Len} - \mathrm{PSa} \leq \delta$$

これにより，近傍のアドレスへの不連続なアクセスを 1 つのアクセスとしてまとめることができる．本稿の実装では，$\delta = 64$ とした．

4.4 アクセス予測処理

予測処理では，LWP 内でアクセスしたメモリに以下の関係がある場合には，ログに関係なく通常のバッファリングが行われる．なお，式中の Sa は LWP がアクセスした最小アドレス，La は最大アドレスである．

$$\mathrm{La} - \mathrm{Sa} \leq \text{Buffer size}$$

式においてバッファサイズは，共有メモリから一度に読み出されるサイズである．上式は，LWP がアクセスした共有メモリの範囲がバッファサイズ以下であるということを意味しており，この場合は通常のバッファリングを行ってもアクセス領域全体がカバーできるので，適応バッファリングの必要性はない．したがって，予測処理は，この条件が成立しない場合のみ行う．予測処理では，すべてのログエントリ (e_i) から，以下の条件式を満たすものを検索する．なお，式中の C_{bi} はエントリ (e_i) の COUNTER B を，C_{ai} は COUNTER A を表している．

$$E_1 = \quad \{e_i : C_{bi}, \{C_{bi}\} = \max\{C_{b1}, \cdots C_{bn}\}\}$$

これにより，LWP が共有メモリアクセス (read) を行った回数が最も多いエントリが選択される．次に，選択されたものから以下の条件を満たすエントリを検索する．

$$E_2 = \quad \{e_i : e_i \in E_1, \\ C_a i = \max\{C_{a1}, \cdots C_{an}\}\}$$

なお，E_2 に含まれるエントリが複数ある場合は，ログデータの先に出現したものを選択する．

以上，予測では，実際のアクセス数（ユーザプログラムからの読み出し命令発行数）の最も多いもののうち，連続アドレスへのアクセスをまとめたパターン数が多いものを選択する．

4.5 予測ミスのペナルティの削減

実装した適応型バッファリングは，主に画像処理や行列演算処理などのように空間的に局所的なデータアクセスを行うものに対して効率良くバッファリングできるように設計した．しかし，ターゲットとしている以外のアプリケーションに対しては，適応処理により逆にヒット率が低下することが予想される．そこで，実装した適応型バッファリング機構では，予測がはずれた場合のペナルティを小さくするために，予測ミス時には通常のバッファリングに切り替える機構を実装した．

図 7 は，バッファリング方式の切替えを行う状態遷移図を示したものである．図 7 中の UP，DN，EQ は，前回の LWP 実行時のバッファヒット率と今回のヒット率を比較して，ヒット率が上がったのか（UP），

下がったのか（DN），同じであるのか（EQ）を表すものである．

LWP 実行時の初期状態は Status1 である．以降は LWP 実行ごとにヒット率を比較し，それに応じて Status 間を移動する．なお，Status0，1 は，通常のバッファリングであり，Status2，3 は適応型バッファリングとなる．このように，4 つの状態を持つことでヒット率の変動に対してバッファ方式が激しく変化するといった状態を防いでいる．

4.6 適応型バッファリングの実装

バッファリングのための機構は，PPElib の共有メモリクラスの read/write 処理をトラップすることにより実装した．実装した適応型バッファリングでは，バッファ数は 1 とし，バッファサイズ分のデータ（領域）を stepread を用いて一括して読み出すようにした．したがって，適応型バッファリングの総バッファサイズは，stepread により一度に読み出すデータサイズと同じである．なお，このバッファ内にデータが存在しない場合には，バッファをフラッシュし，新たなデータを共有メモリから読み込む．

5. 評 価 実 験

5.1 実 験 環 境

実験では，100Base-T のスイッチングハブにより接続された，PC/AT 互換機（PentiumII 450 Mhz, 128 Mbyte，FreeBSD）5 台を利用した．なお，実験は，他のユーザのプロセスがなるべく実行されていない状態で行っており，DSE 以外には負荷の高い処理はなかった．また，5 台のうち 1 台はアプリケーションオブジェクトを実行するプロセッサとなり，この計算機はタスク実行（並列実行）に参加しない．したがって，並列実行は最大 4 台で行われる．

なお，この環境では，DSE の共有メモリ読み込み処理には，1 バイトの場合で 270 usec，2 K バイトの場合で 560 usec 必要であった．

5.2 ウェーブレット変換

1 つめの評価実験には，画像に対してウェーブレット変換[10]を行うアプリケーションを利用した．このアプリケーションのソフトウェア構成を**図 8** に示す．図 8 のように，このアプリケーションは水平方向の走査を行うタスクと，垂直方向への走査を行うタスクの 2 つから構成される．また，処理する画像のファイルから読み込み処理と，実行後の変換画像の保存処理がアプリケーションクラスの処理として行われる．なお，それぞれのタスク内の LWP では，1 ライン（水平ライン・垂直ライン）のデータに対して，ローパスフィルタ（LPF）を施した後，ハイパスフィルタ（HPF）を施す．この処理を行った後の画像データが**図 9** である．図 9 のように，変換後は，左上が LL 成分に，その他の各部分は LH，HL，HH 成分となる．実験では 640 × 480 pixel の画像にウェーブレット変換をほどこした．

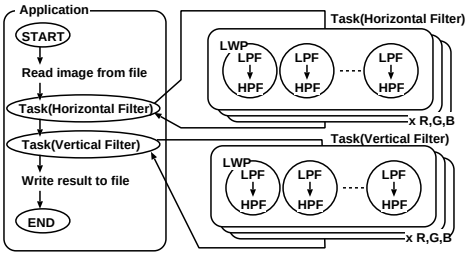


図 8 ウェーブレット変換アプリケーションのソフトウェア構成
Fig. 8 The software block of the wavelet transform application.

図 9 変換後の画像
Fig. 9 The result of wavelet transformation.

```
Horizontal Filter
LPF  Vaule = H0[0]*(img[x-2][y] + img[x+2][y]) +
             H0[1]*(img[x-1][y] + img[x+1][y]) +
             H0[2]*imt[x][y];
HPF  Vaule = H1[1]*(img[x-1][y] + img[x+1][y]) +
             H1[2]*imt[x][y];
Vertical Filter
LPF  Vaule = H0[0]*(img[x][y-2] + img[x][y+2]) +
             H0[1]*(img[x][y-1] + img[x][y+1]) +
             H0[2]*imt[x][y];
HPF  Vaule = H1[1]*(img[x][y-1] + img[x][y+1]) +
             H1[2]*imt[x][y];
```

図 10 LPF/HPF の演算
Fig. 10 The expression of LPF/HPF.

図 10 は，それぞれの LWP のフィルタ処理の演算を示したものである．このように，ウェーブレット変換においては，水平方向は，$x \pm 1$ または $x \pm 2$ へのアクセスが図 10 の順で行われる．水平方向については，横方向への 1 列のアクセスとなるため，通常のバッファリングでも効果的にバッファリング可能である．一方，垂直方向はメモリアクセスで見ると 640 pixel

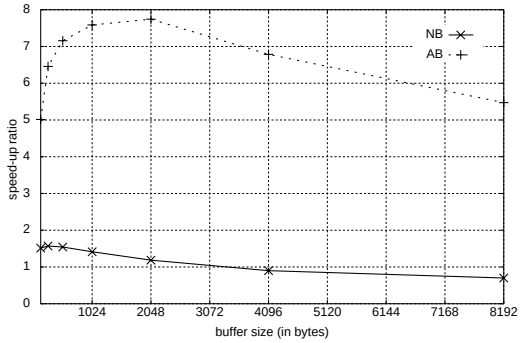


図 11 実験結果 1（速度向上率）
Fig. 11 The execution result 1 (speed up ratio).

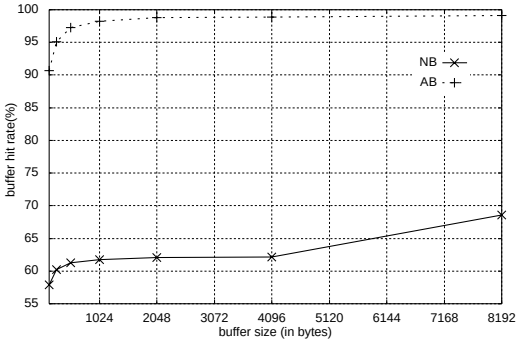


図 12 実験結果 2（ヒット率）
Fig. 12 The execution result 2 (buffer's hit ratio).

単位ではなれたメモリへのアクセスとなるため，通常のバッファリングでは効率が悪い．プログラムでは，フィルタ演算を行う前に画素データはすべて `double` 型のに変換されているため，640 pixel 離れているデータは，実際には 640×`sizeof(double)` バイト離れているデータということになる．

5.2.1 実 験 1

この実験では，バッファリングを行わないもの（必要データをつねにネットワークを経由して所得するもの）と，通常のバッファリング（Normal Buffering; NB）を行うもの（連続アドレスを一定量バッファリングするもの）と，適応型バッファリング（Adaptive Buffering; AB）を行うものの 3 つを用意し比較を行った．なお，プログラムの変更はライブラリ側だけであり，アプリケーション側のプログラムはすべて同じものを利用している．**図 11** は，ウェーブレット変換を行うアプリケーションを実行した結果のグラフである．グラフの横軸はバッファサイズ，縦軸は「バッファリングなし」と比較した場合の速度向上率であり，1 以上であればバッファリングなしに比べ速いことを意味する．グラフより，AB は，バッファサイズが 2 K バイトのときに最も効果的で，約 7.74 倍の速度向上が得られていることが分かる．一方，NB では最大 1.57 倍しか速度向上が得られていない．

図 12 は，バッファのヒット率を示したものであるが，これより NB のヒット率はおおよそ 58〜68%前後であることが分かる．この理由は，水平方向のフィルタ処理ではほぼ 100%に近いヒット率が得られるのに対して，垂直方向のフィルタ処理ではミスヒットが頻繁に発生してしまうためである．これに比べ AB では，バッファサイズが 128 バイトでも 90%以上のヒット率を実現することができた．

なお，一定以上バッファサイズが大きくなると効果が低くなる理由は，読み込みオーバヘッドが高くなるため，ヒット率が高くてもバッファリング効果が低下するからである．

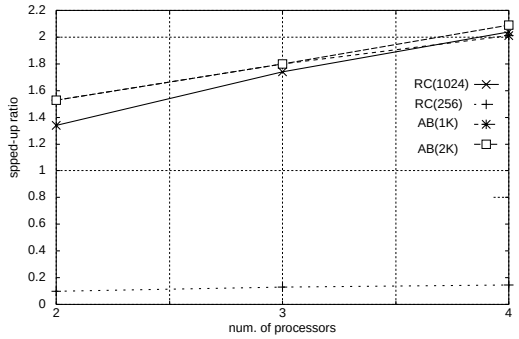


図 13 実験結果 3（ウェーブレット，AB vs. RC）
Fig. 13 The execution result 3 (wavelet, AB vs. RC).

5.2.2 実 験 2

ここでは，ページベースでメモリ管理を行う Lazy Release Consistency（LRC）モデルに基づくキャッシュ機構を実装した PPElib と，適応型バッファリングの比較を行った．実装した LRC では，コヒーレンス制御としてキャッシュ無効化操作を行う．実験では，適応型バッファリングについてはバッファサイズが 1 K バイト（AB(1K)）と 2 K バイト（AB(2K)）のものを，LRC ではページ数が 1024 ページ（RC(1024)）のものと 256 ページ（RC(256)）のものを用意した．なお，LRC のページサイズは DSE のページサイズと同じ 4 K バイトとした．したがって，LC(1024) と LC(256) の総キャッシュサイズは，それぞれ 4 M バイト，1 M バイトとなる．また，ページ置換のアルゴリズムは LRU とした．

図 13 は，実験を行った結果である．図 13 の横軸はプロセッサ数を，縦軸は逐次処理に対する速度向上率を表している．グラフより，RC(1024) と AB(1K),

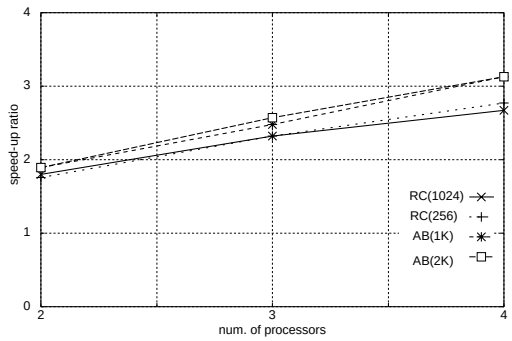


図 14 実験結果 4（knight tour, AB vs. RC）
Fig. 14 The execution result 4 (knight tour, AB vs. RC).

AB(2K) は，ほぼ同じ程度の速度向上が得られていることが分かる．RC(256) で速度向上が得られていない理由は，垂直方向のフィルタ処理時に，処理する画像の垂直 1 ライン分のデータを格納するにはキャッシュサイズが小さいため，結果としてキャッシュミスが頻繁に発生してしまうためである．一方，RC(1024) では，画像データがキャッシュに入りきってしまうので，垂直方向の処理でもミスヒットは発生しない．

実験より，このアプリケーションにおいては，AB は，1 K または 2 K バイトという少ない総バッファサイズで，より多くのキャッシュを持つ RC と同等の性能が得られていることが分かる．

5.3 巡回騎士問題（Knight Tour）

適応型バッファリングが効果的に動かないアプリケーション例として，Knight Tour を利用して実験を行った．Knight Tour は，チェスのナイトをルールに従って動かし，すべてのマス目を一度だけ通る順路を求める問題である．このプログラムでは，5 手まで展開し，その後の処理を並列に実行する．各プロセッサ上で実行される LWP では，展開された盤面のデータを共有メモリから読み出し，続きの探索を行う．

図 14 は，Knight Tour を実行した結果である．この実験でも，実験 2 と同様に AB と RC の比較を行った．Knight Tour の場合，AB では，適応型バッファリングでは矩形領域のバッファリングは行われず通常のバッファリングが行われていた．図 14 から，このアプリケーションにおいては，RC と AB はほぼ同じ速度向上が得られたことが分かる．なお，RC の方が若干速度向上が得られていない理由は，ライブラリにキャッシュ機構を組み込んだため，アプリケーションの実行時にキャッシュの初期化が行われるからである．

以上の結果から，適応型バッファリングが典型的な画像処理（ウェーブレット変換）においては効果的に動作し，高いヒット率を得られることが分かった．また，このようなアプリケーションでは，RC に比べて少ないキャッシュサイズで，RC と同様の効果が得られることを確認できた．

6. 関 連 研 究

オブジェクト指向言語により並列プログラミングを行うものとしては，HPC++Lib [11] や OPBLib [12]，POOMA [13] などがある．継承を用いてプログラミングするという意味では，本クラスライブラリは，HPC++Lib に近い．しかしながら，HPC++Lib がスレッド単位でクラスを用意しているのに対して，我々は，複数のスレッドをまとめたタスクをクラスとして提供している．このため，プログラミングの自由度は HPC++Lib に比べて低下するが，各仮想プロセッサへのプロセス割当てや同期処理を記述しなくてよいという利点がある．

また，アクセスパターンによりバッファリングを変更するものとしては Hu ら [14] の方式がある．これは，ページベースの DSM において，アクセスパターンに合わせてメモリの配置を入れ換えることでページにアクセスを集中させるというものである．この方式では，アクセスパターンを指定するために，プログラム中に新たに数行を挿入する必要がある．一方，我々のものでは，プログラムインタフェースの変更を必要としない．

7. ま と め

本稿では，プログラミングモデルをライブラリで制限することで，簡易なコヒーレント処理で共有メモリバッファリングが可能なことを示した．また，タスクがまとまった 1 つの処理を行うという特徴を利用することで，タスク内の LWP の共有メモリアクセスを予測し，特に画像処理のような規則性のある処理に対して，プログラミングインタフェースを変更せずに適応的なバファリングが可能であることを示した．

また，実験結果より，適応バッファリングでは少ないサイズの一時バッファで LRC モデルに基づくページベースのキャッシングと同様の効果を得られることが分かった．

今後は，より多くのアプリケーションでの適応バッファリングの評価を行うと同時に，ページ数を増やした適応バッファリングや，たとえばヒット率により適応バッファリングとページベースのバッファリングを切り替えるような方式について検討する予定である．

参 考 文 献

1) Tezuka, T., Ryokai, K., Apduhan, B. and Sueyoshi, T.: Implementation and Evaluation of a Distributed Supercomputing Envionment on a Clustor of Workstations, *Proc. 1992 International Conference on Parallel And Distributed System*, pp.58–65 (1992).
2) 大西淑雅，了戒 清，末吉敏則：分散共有メモリモデルに基づく HPC 環境の高性能実装と性能評価，情報処理学会論文誌，Vol.36, No.7, pp.1729–1737 (1995).
3) Matsumoto, T., Niwa, J. and Hiraki, K.: Compiler-Assisted Distributed Shared Memory Scheme Using Memory-Based Communication Facilities, *Proc. 1998 PDPTA*, Vol.2 (1998).
4) Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer*, Vol.29, No.2, pp.18–28 (1996).
5) 斎藤彰一，國枝義敏，大久保英嗣：分散並列処理のためのプラットフォーム Lemuria における分散共有メモリの性能評価，電子情報通信学会論文誌 D-I，Vol.82, No.3, pp.457–466 (1999).
6) Carter, J.B., Bennet, J.K. and Zwaenepoel, W.: Implementation and Performance Evaluation of Munin, *Proc. 13th ACM Symposium on Operation System Principles*, pp.152–164.
7) Iftode, L., Dubincki, C., Felton, E.W. and Li, K.: Improving Release-Consistent Shared Virtual Memory using Automatic Update (1996).
8) 宮原敦夫，岡 雅樹，大西淑雅，末吉敏則：クラスタコンピューティングにおけるソフトウェア・キャッシュ機構の評価，情報処理学会九州支部研究会報告，pp.250–259 (1997).
9) 尾上守男 ほか（編）：画像処理ハンドブック，昭晃堂 (1987).
10) Chui, C.K.: *An Introduction to Wavelets*, ACADEMIC PRESS, INC. (1992).
11) Beckman, P., Gannon, D. and Johnson, E.: Portable Parallel Programming in HPC++. http://www.extreme.indiana.edu/hpc++/index.html (1996).
12) Matsuda, M., Sato, M. and Ishikawa, Y.: OBPlib: An Object-Oriented Parallel Library and its Performance, *2nd France-Japan Workshop Object Based Parallel and Distributed Computing* (1997).
13) Williams, T.J.: POOMA User Guide. http://www.acl.lanl.gov/pooma (1998).
14) Hu, Y.C., Cox, A. and Zwaenepoel, W.: Improving Fine-Grained Irregular Shared-Memory Benchmarks by Data Reordering. http://www.cs.rice.edu/~willy/TreadMarks/papers.html

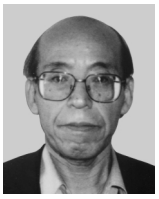
(平成 11 年 9 月 1 日受付)
(平成 12 年 2 月 4 日採録)

手塚 忠則（正会員）

1968 年生．1993 年九州工業大学大学院情報科学専攻博士前期課程修了．同年，松下電器産業（株）入社．現在，同社九州マルチメディアシステム研究所および九州工業大学大学院情報科学専攻博士後期課程在籍．ネットワークを用いた並列処理，および画像処理技術の研究開発に従事．

末吉 敏則（正会員）

1953 年生．1976 年九州大学工学部情報工学科卒業．1978 年同大学院工学研究科情報工学専攻修士課程修了．同年九州大学工学部情報工学科助手．同大学院総合理工学研究科助教授，九州工業大学情報工学部知能情報工学科助教授を経て，1997 年より熊本大学工学部数理情報システム工学科教授．工学博士．計算機アーキテクチャ，システムソフトウェア，計算機ネットワーク，VLSI システム設計等の研究に従事．著書「並列処理マシン」（共著，オーム社）．



有田五次郎（正会員）

1939 年生．1963 年九州大学工学部電子工学科卒業．1965 年同大学院修士課程修了．同年九州大学講師（中央計数施設所属）．1984 年九州工業大学工学部教授を経て，1988 年同情報工学部教授（知能情報工学科）．工学博士．計算機アーキテクチャ，並列処理システム，計算機ネットワークの研究に従事．電子情報通信学会，ソフトウェア科学会会員．