377.5
K-11-2
1-81

分散共有メモリに基づく クラスタコンピューティング環境の 高性能化に関する研究

手塚　忠則

九州工業大学附属図書館
0010459287

目次

第１章

はじめに

近年のHPC(High Performance Computing)の研究のトレンドの変化からもわかるように，高性能計算機の構成はベクトル並列計算機からクラスタに変化してきている．クラスタは，汎用的なプロセッサを高速なネットワークで接続した構成をとる並列計算機で，低価格で高いピークパフォーマンスを実現することができることが特徴である．このクラスタの構成としては，複数のプロセッサボードを専用のネットワークにより接続し，１つのラックに収納したような専用の並列計算機と，一般的なワークステーション（ＷＳ）やパソコン（ＰＣ）をMyrinetなどの高速ネットワークやFast Ethernetなどのネットワークにより接続したもの（以下，ＰＣクラスタと呼ぶ）の大きく２つがある．

前者としては，Linuxマシンをラックに収納し，高速ネットワークにより接続したLinuxクラスタと呼ばれるものがある．このようなクラスタは，並列処理専用にチューニングされたものであり，並列処理を効果的に行えるようにハードウェアおよびシステムに手を加えていることが一般的である．

一方，後者の汎用のＷＳやＰＣを利用したＰＣクラスタは，並列処理のために専用に用意されたものではなく，基本的に（空いている）計算機の余剰能力を利用して並列処理を行おうというアプローチである．このアプローチの場合，通常利用しているＷＳやＰＣが利用の対象となるため，通常利用しているＯＳなどの環境をなるべく変更せずにクラスタの機能を実現することが望まれる．したがって，このようなクラスタではハードウェアやＯＳなどのシステムソフトウェアに手を加えることは基本的には難しく，システムソフトウェア等の変更なしに高性能化を行わなければならない．

このようなＰＣクラスタ向けの基本ソフトウェアとしては，PVM(Parallel Virtual Machine)[1][2][3][4]およびMPI(Message Passing Interface)[5][6]といったものがあり，これらはＰＣクラスタはもとより，Linuxクラスタ等の専用の並列処理計算機でも利用されている．これらは，基本的にＯＳの変更なしにクラスタ環境を実現するソフトウェアで，メッセージ交換モデルに基づく並列プログラムの実行環境を提供する．また，近年では，メッセージ交換モデルでは記述しにくいプログラムを効率よく記述・実行できる環境として共有メモリモデルに基づく実行環境をＰＣクラスタ上に実現しようという研究が行われている．

我々の研究でもDSE(Distributed Supercomputing Environment)と呼ぶ分散共有メモリモデルに基づくクラスタコンピューティング環境を構築しており，ＰＣクラスタでの共有メモリモデルのクラスタ環境の高性能実装を目指して研究を行っている．DSEは，ＬＡＮ接続されたＷＳおよびＰＣを利用してクラスタ環境を実現するための基本ソフトウェアであり，SunOS, Solaris, AIX, Linux, FreeBSDといったUNIX系のＯＳとWindowsNT上で動作するユーザアプリケーションとして実現されている[50][51][52][53][54]．

本研究は，分散共有メモリモデルのクラスタ環境の高性能化を目指した取り組みであり，DSEのように特別なネットワークの利用や，ＯＳの変更を行わない環境での高性能化を実現するためのものである．特に，本研究では，(1)DSEの通信処理部と，(2)プロセスの割り当てや共有メモリのキャッシングといったプログラム中の制御を高性能化の対象として検討を行った．

(1) 通信処理

DSEでは，計算機上で実行される仮想的なプロセッサ要素（以下，仮想プロセッサと呼ぶ）の接続のためのプロトコルとしてＴＣＰを利用している．ＴＣＰはコネクション型のプロトコルであり，仮想プロセッサ間で１対１にコネクションをはる必要がある．ＯＳでは，コネクション毎にＯＳリソース（ポート）を割り当てるため，多数の仮想プロセッサを接続しようとするとコネクションによる大量のリソースの消費が発生し，これがＯＳリソースを圧迫していた．また，多くのＯＳには，１つのプロセスが同時にはることができるコネクションの数に制限があり，この制限により利用できる仮想プロ

セッサの数が制限される. DSEでは, この問題を回避するため, 鎖網やトーラス網といった仮想的なネットワークをLAN上に構築し，１つの仮想プロセッサが接続するコネクションの数を減らしていた．例えば，トーラス網では，１つの仮想プロセッサは４つの仮想プロセッサと接続すればよいため, 論理的には無制限に仮想プロセッサを接続できる．しかし，仮想ネットワークによる接続では，隣接していない仮想プロセッサとの通信に対して，経路上の仮想プロセッサを経由した通信が必要になり，これが通信オーバヘッドとなっていた．本研究では，通信プロトコルとしてＵＤＰを利用することで，この問題を回避した．ＵＤＰは，コネクションレス型のプロトコルで，１つのポートで複数の仮想プロセッサとの通信が可能である．このプロトコルを利用すれば，仮想プロセッサの数によらず常に仮想プロセッサ間で直接通信を行うことができるため, 前述したような通信オーバヘッドは発生しない．しかしながら，ＵＤＰは送信したデータが相手先に到着するという保証をしないプロトコルであるため, DSEのような通信の高信頼性を要求するアプリケーションにはそのままでは適用できない. これを解消するために, 本研究では，ＵＤＰの上に信頼性を保証するプロトコルを実装し，これにより信頼性のある通信を実現した.

(2) プログラム中の制御（プロセス割り当てとキャッシュ制御)

DSEでは，プロセス（軽量プロセス）のスケジューリングはFCFS(First-Come First-Served)で行われている. FCFSでは, 予約順に軽量プロセスが実行されるだけであり, 処理量や負荷を考えた軽量プロセスの仮想プロセッサへの割り当てはユーザに任されていた．また，共有メモリのアクセスを削減は，ユーザによる共有メモリのデータのローカルなメモリへの一括コピーにより行われていた．つまり，並列アプリケーションの実行効率を左右する要素の多くが，ユーザのプログラムに任されていた．本研究では，このようなプログラム中の制御の内, 軽量プロセスの割り当て処理と共有メモリのバッファリング処理（キャッシング）処理をシステム側で効率よく行うための検討を行った．このような処理を効率よく行うためには, ユーザによって記述されたプログラムの性質を把握することが重要であるが, これまでのDSEのプログラミングのようにユーザの自由度が高いプログラミング環境ではプログラムの性質を把握することが難しい. プログラムの性質を把握して効率のよい制御を行うためには, このような制御を行いやすいプログラミング環境を提供することが必要となる.

このような環境としては，HPF[29]やOpenMP[31]のように，コンパイラにより自動的に並列化を行うアプローチがある．これらは，適切な指示子をユーザに挿入させることで，プログラムの性質を把握し，自動的な並列化と制御を実現する．しかしながら，このようなコンパイラによる自動化のアプローチは，我々がターゲットとしている通信遅延の大きなＰＣクラスタに対して効率よく実行できるコードを出力することは難しい．

また，ライブラリによりこのような制御を行うものとしてOBPLib[23], POOMA[25], MPC++[24], HPC++Lib[27][28]などがある．これらは，オブジェクト指向言語の特徴を利用してプログラムの記述の簡略化を図ると共に自動的な制御を実現する．具体的には，OBPLibとPOOMAはC++言語の持つテンプレート機能を，HPC++LibとMPC++は，テンプレート機能とクラス機能を用いて自動的な並列化と実行制御を行う．

本研究では，オブジェクト指向言語のもつ継承という機能を利用して，ユーザが記述したアプリケーションの性質を把握し，自動的な制御を行った．具体的には，ユーザの記述できるプログラミングのスタイルを定義するクラスライブラリを提供し，このプログラミングスタイルに合ったプログラム記述をユーザに行わせる．ライブラリの提供するプログラミングスタイルでは，プログラム内で並列可能なまとまった処理をタスクと定義し，このタスク内で依存関係なく並列実行できる部分処理を軽量プロセスと定義する．このように１つのタスクが複数の軽量プロセスを持つモデルにすることで，タスク単位で軽量プロセスの制御を行うようにした．ライブラリでは，このタスクが１つのクラスとして定義されており，ユーザはこのタスククラスを継承して並列実行される処理を記述する．このプログラミングスタイルに沿ったアプリケーションでは，これまでに比べプログラム記述の自由度が制限されており，またプログラムの性質を把握しやすい．ライブラリでは，これを利用して，タスク内の軽量プロセスの動的なプロセス割り当てと，適応的な共有メモリバッファリング制御を実現した．これにより，これまでユーザが行っていたこれらの制御をシステムに実装すると共に，効率の良い実行を実現した．

本論文では，2章ではメッセージ交換モデルおよび共有メモリモデルといったシステムモデルの概要，MPI等の既存のクラスタコンピューティング環境およびDSEの概要について説明する．続く３章では，DSEの通信処理のUDPによる実現と評価実験の結果について述べる．また，４章では，オブジェクト指向並列プログラミングライブラリの設計

と他プログラミングライブラリとの比較を，5章ではライブラリに組み込んだ動的プロセス割り当て機構の設計と評価結果，6章では効率のよいバッファリングを実現するライブラリ内の適応型メモリバッファリング機構の設計と評価結果について述べ，7章でまとめる．

第２章

クラスタコンピューティング環境

2.1 概要

クラスタコンピューティング環境とは，ネットワーク接続されたワークステーションやPCを用いて並列計算機を構成するためのハードウェアおよびソフトウェアシステムである．クラスタコンピューティング環境を構築する手法としては，大きくメッセージ交換に基づく並列計算機の機能を実現する方法と，分散共有メモリに基づく並列計算機の機能を実現する方法がある．

前者は，ネットワーククラスタ環境との親和性が高く，早くから環境が提供されている．このメッセージ交換モデルの代表ともいえるPVM[1][2][3][4]やMPI[5][6]といった環境は，現在ではクラスタコンピュータだけでなく並列計算機の標準ライブラリとして提供されている．また，後者の分散共有メモリモデルのクラスタ環境は，メッセージ交換モデルに比べて少し遅れて登場することとなったが，現在では広く研究され，実用レベルとなっている．

この２つのシステムモデルは，また，プログラミングのモデルを決定する重要な要素である．一般的に，並列プログラムのモデルは共有メモリモデルと分散メモリモデルに大別され，前者は分散共有メモリモデルに，後者はメッセージ交換モデルのシステムに合ったモデルである．

共有メモリモデルでは，複数のプロセスやスレッドが実行する主体となり，共有されたメモリをアクセスすることで通信を行いながら計算処理を行う．このため，「通信」という処理が陽に現れず，プログラマは通信を意識しなくてよいという特徴を持つ．しかしながら，個々のプロセスやスレッドが実行順序を考えずに実行してしまうと，思った

結果を得ることができない．そこで，共有メモリモデルでは，プログラム中に同期処理(バリア同期，セマフォなど）を明示的に記述することによって同期をとる．

一方，分散メモリモデルでは，複数のプロセスは各々独立したメモリ空間で実行されるため，プロセス間でデータ交換を行うためにはメッセージ交換を行う必要がある．したがって，分散メモリモデルは，メッセージパッシング（交換）モデルのプログラミングが行われる．メッセージパッシングモデルでは，プロセス間でデータの送信と受信が行われ，これにより同期がとられるため，明示的に同期手続きをプログラムに挿入する必要は基本的にはない．

また，近年では，MPIで共有メモリモデルのプログラミングを実現するなど，共有メモリモデルとメッセージパッシングモデルのハイブリッド環境も出現してきている．このような環境では，双方の利点をうまく使うことで，効率がよくわかりやすいプログラムを記述することが可能である．

2.2 メッセージ交換モデルに基づく環境

ここでは，メッセージ交換モデルに基づく環境として，PVMとMPIについて説明する．

2.2.1 PVM(Parallel Virtual Machine)

PVM(Parallel Virtual Machine)は，Oak Ridge国立研究所(ORNL)にて1989年から開発が開始された環境である．PVMは，ネットワークに接続された異機種UNIXコンピュータ群を単一の仮想並列計算機として利用することを可能にするツールとライブラリより構成されるソフトウェアシステムである．また，近年ではUNIXだけでなく，Windowsなどの環境にも移植され，非常に多くのプラットフォーム上で動作するシステムの１つとなっている．

このPVMプロジェクトの研究成果であるソフトウェアは，無料で配布されているため，現在では，世界中の大学や研究機関で広く利用されている．

PVMでは，逐次コンピュータ，並列コンピュータ，およびベクトルコンピュータなどの集まりをユーザが定義し，１つの分散メモリ型並列計算機として利用する．PVMでは，この論理的な分散メモリ型並列計算機をバーチャルマシン（Virtual Machine）と定義し，コンピュータの集まりの１つ１つをホスト(Host)と定義している．各ホストには，タスクを起動する機能が備えられており，バーチャルマシンではタスク間の通信および同期に必要となる機能を実現する．タスクは，PVMにおける計算（処理）の単位であり，

UNIXのプロセスとよく似たものである．このため，タスクはUNIXのプロセスとして実装されることが多いが，必ずしもそうである必要はない．

PVMでのアプリケーションの記述には，C言語またはFortran言語を利用する．PVMでは，これらの言語に対応したメッセージパッシングライブラリが用意されており，ユーザはこれを利用してプログラムの並列化を行うことになる．

このPVMの大きな特徴は，「異機種間利用」をサポートしていることにある．このため，ユーザは様々なアーキテクチャの計算機をアプリケーションの実行に利用することができ，問題にもっとも適した計算機にタスクを割り当てることが可能になる．また，PVMのメッセージパッシングライブラリがデータフォーマットの差異を吸収するため，ユーザはアーキテクチャによるデータフォーマットの違いを意識せずデータ交換を行うことができる．

計算モデル

図2.1は，PVMの計算モデルを示したものである．図に示すとおり，PVMアプリケーションはいくつかのタスクから構成される．タスクは，大きく処理内容により分類される．図では，入力とデータ分割，計算(グループ1，2)，結果の出力と表示の4つのグループに分類されている．このように，機能毎に並列化を行うことを機能の並列化と呼ぶ．それぞれのグループ内では，データ並列化が行われる．これはSPMD(single-program, multiple-data)モデルの並列化である．このように，PVMでは機能並列化とデータ並列化のどちらか，および双方を合わせた計算モデルによる並列処理を可能としている．

システム構成

この，PVMシステムは大きく2つのパートから構成される．1つはデーモン(pvmd)であり，これはPVMシステムを構成する全ての計算機上で実行されている必要がある．pvmdは，ユーザが起動できるように設計されており，ユーザがPVMアプリケーションを実行する前に起動することで，仮想計算機を構成することを可能にする．もう1つは，PVMインタフェースルーチンであるライブラリである．ライブラリは，アプリケーションのタスクが協調動作するために必要なプリミティブを含み，ユーザはこのライブラリをコールすることでPVMの機能を呼び出す．このライブラリには，メッセージ交換，プロセスの生成，タスクのコーディネイト，仮想計算機のモディファイなどのファンクションが含まれている．

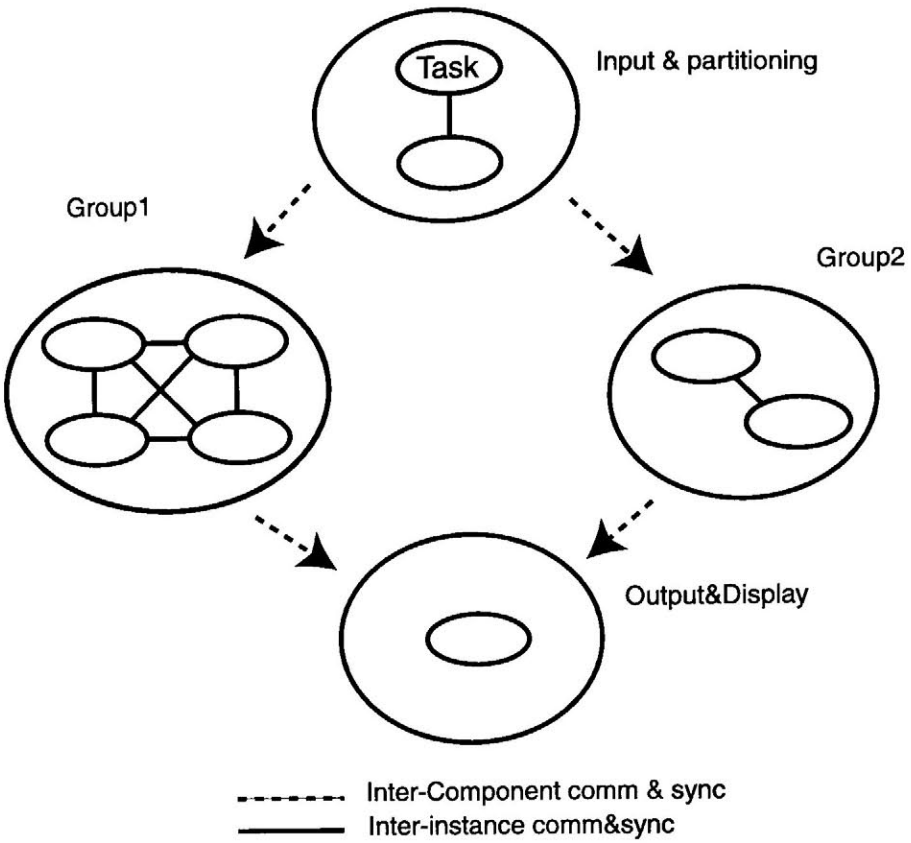


図 2.1　PVM の計算モデル

一般的に，PVM を利用する場合，以下の手順になる．まず，ユーザは 1 つ以上の逐次プログラムを記述する．記述された個々のプログラムはタスクに対応する．このプログラムは, PVMシステムで利用する計算機でコンパイルされ実行可能なオブジェクトとして各計算機に格納される．次に，PVM のコンソールを起動し，ここに，アプリケーションのメインプログラムとなるタスク名を入力することでアプリケーションの起動を行う．このタスクとしては，例えば図 2.2 のようなものである．

```
main()
{
   int cc, tid, msgtag;
   char buf[100];

   printf("i'm t%x\n", pvm_mytid());

   cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

   if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
   } else
      printf("can't start hello_other\n");

   pvm_exit();
}
```

図 2.2　PVM program hello.c

2.2.2 MPI(Message Passing Interface)

MPI(Message Passing Interface)は，約 40 の組織により標準化が行われた，並列処理のための通信ライブラリである．MPI は，それ以前に研究が行われていた Intel の NX/2[7], Express[8], nCUBE の Vertex[9], PARMACS[10][11], Chimp[12][13], PVM[1][4], Cameleon [14], PICL[15]などを参考にし，これらの機能の数々を採り入れることにより設計が行われている．

MPI の主な利点は，そのポータビリティと使いやすさである．MPI は，特に分散メモリ環境を考慮して設計されており，このような環境のために高レベルのルーチンを提供する．以下に MPI の設計目標を示す．

- アプリケーションプログラムが呼ぶためのインタフェース(API)の設計．
- メモリコピーを避け，計算と並行して通信を進めることができるような効率のよい通信の実現．
- 異種環境でも使用可能な処理系．
- C 言語や Fortran77 言語のための使いやすい呼出形式の実現．
- ユーザが通信障害に対処する必要性がない，信頼性の高い通信インタフェース．

・PVM, NX, Expressなどの既存のインタフェースと似ており，かつ，より拡張性の高いインタフェースの定義.

・(意味が)言語に依存しないインタフェース.

・マルチスレッド環境に対応可能なインタフェース設計.

MPIは，分散メモリのマルチプロセッサから，ワークステーションによるネットワーク，またはこれらの組み合わせの上でも動作する非常にポータビリティの高いシステムである.

このMPIの標準では，以下の機能を定義し，それに対応するインタフェースが用意されている.

・1対1通信

MPIの基本的な通信メカニズムであり，基本的な操作としては送信(MPI_Send)と受信(MPI_Recv)が用意されている.

・集団通信

複数のプロセスから構成されるグループに対する通信メカニズムであり，(1)全グループメンバによるバリア同期, (2)グループ内のあるメンバからのブロードキャスト, (3)全グループメンバからあるメンバへのデータのgather, (4)グループ内にあるメンバから全メンバへのデータのscatter, (5)sum, max, minまたはユーザ定義関数などの結果の全グループメンバへの返信や(6)リダクションなどの機能が提供されている.

・プロセスグループ

集団通信などを行うための「グループ」を定義するメカニズム.

・プロセストポロジー

仮想的なプロセスの配置(仮想トポロジー)を実現するメカニズム.

・環境管理と問い合わせ

MPIの実行環境(エラー処理など)に関連する各種パラメータの取得および設定を行うルーチンを提供するメカニズム.

・プロファイリング

プロファイリングを可能にするためのインタフェースの定義.

MPIでは，これらのインタフェースを定義することで，標準的なメッセージ交換型のライブラリの仕様を提供している．なお，このMPIの実装としては，並列計算機の各ベンダが提供するものや，ワークステーションクラスタで動作するMPICH[6]などがある．

2.3 共有メモリモデルに基づく環境

ここでは，共有メモリモデルに基づく環境としてTreadMarkおよびHPC++Libについて説明を行う．

2.3.1 TreadMark

TreadMark[16][17][18][19]は，Rice Universityで開発されたクラスタコンピューティング環境であり，ネットワーク接続されたUNIXシステム（SunOS, Ultrix）を利用して分散共有メモリを実現する．

TreadMarkの提供する並列計算機のモデルを図2.3に示す．図2.3のように，各プロセッサのメモリはネットワークで接続されており，全体で1つの共有メモリ空間を構成する．このように，TreadMarkでは，共有メモリが各プロセッサに分散されて配置された分散共有メモリモデルの並列計算機の機能を実現する．

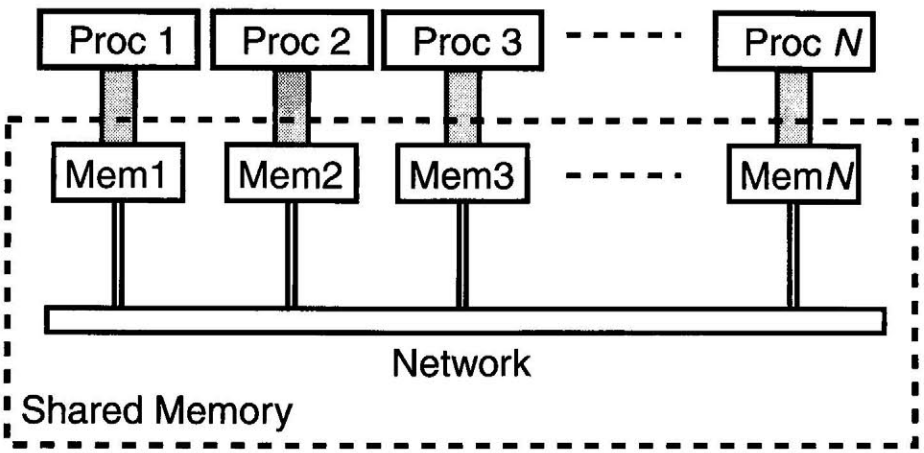


図 2.3　ThreadMark の SystemModel

このシステムの特徴は以下の通りである.

・LRC(Lazy Release Consistency)[21]モデルによるデータキャッシュ

TreadMark では共有メモリの一貫性を保証する方法として，Lazy Release consistency モデルが利用されている．この RC(Release Consistency)モデルでは，Releaseまで共有メモリの一貫性保証のための処理を遅らすことができるため，共有メモリのキャッシュ制御に必要な通信を削減可能である．LRCは，RC に加え，同期変数獲得操作を行ったプロセッサにのみ無効化／更新メッセージを送出することで通信処理を削減している．

・Multiple-Writer Protocols[22]

TreadMarkでは，共有メモリの書き込みに対してMultiple-Writerプロトコルを採用している．このプロトコルは，同じ共有メモリのページに対して複数のプロセッサが同時に変更することを許可するもので，これによりキャッシュページの無効化処理を削減可能となる．

・Lazy Diff Creation

TreadMark では，更新されたキャッシュページに対して，更新前のデータを保持している．更新時（Release時）には，更新前後の差分をとり，結果を圧縮して共有メモリを保存するプロセッサへ送る．これにより，書き込みによる通信を大幅に減らしている．

TreadMarkでは，C言語でアプリケーションのプログラミングを行う．プログラミングライブラリでは，共有メモリの割り当て／解放，プロセッサ間の同期などの機能が提供されており，例えば図 2.4 のようにプログラムを記述する．

```
#define M 1024
#define N 1024
float **grid;
float **scratch[M][N];

main()
{
 Tmk_startup();
 if (Tmk_proc_id == 0) {
  grid = Tmk_malloc( M * N * sizeof(float));
  initialize grid;
 }
 Tmk_barrier(0);
 length = M / Tmk_nprocs;
 begin = length * Tmk_proc_id;
 end = length * (Tmk_proc_id+1);
 for( number of iterations) {
  for(  i = begin; i < end ; i++)
   for( j = 0 ; j < N; j++)
     scratch[i][j] = (grid[i-1][j]+grid[i+1][j]+
                      grid[i][j-1]+grid[i][j+1])/4;
  Tmk_barrier(1);
  for( i = begin; i < end; i++)
   for( j = 0 ; j < N; j++)
    grid[i][j] = scratch[i][j];
  Tmk_barrier(2);
 }
}
```

共有メモリの割り当て

共有メモリへのアクセス

図 2.4　TreadMark による Jacobi プログラムの仮想コード

図のように，通常のメモリのように共有メモリをアクセスすることができるのが，TreadMark のプログラミング上の特徴である．

2.3.2 HPC++Lib

C++ 言語は，言語が持つ数式記述能力と効率のよい処理系の出現により，数学者，物理学者の間でも使われるポピュラーな言語になっきている．近年，このC++の持つオブジェクト指向プログラミング支援機能を生かした並列アプリケーション記述のための研究が盛んである(OPBlib[23], MPC++[24], POOMA[25])．このような研究の１つとして，HPC++[26]がある．HPC++は，(1)データ並列処理のためのコンパイラ指示子，(2)C++言語を拡張しないSTL の並列版の提供，および(3)Array クラスからなる level 1 仕様が公表されている．

HPC++Lib(High Performance C++ Library)[27]は，HPC++ Level1 を実現するためのライブラリとして定義されたものであり，Indiana University などで実装が行われている．

HPC++Lib は，CC++ 言語，MPC++ などの設計に似た設計が行われており，以下のような特徴を持つ．

- Java スレッドクラススタイルによるプログラミング
- 同期，リダクション，リモートメモリ参照（グローバルポインタ），PSTL(Parallel Standard Template Libaray)を含むクラスとテンプレートの提供
- CORBA ベースのリモートオブジェクトアクセス

HPC++Lib では，ユーザはスレッドクラスを継承してプログラミングを行う．このクラスは Java のスレッドとほぼ同じ機能を提供する．図 2.5 は，スレッドクラスのクラス定義である．

```
class HPCxx_Thread{
   public:
       HPCxx_Thread(const char *name = NULL);
       HPCxx_Thread(HPCxx_Runnable *runnable,
                    const char *name = NULL);
       virtual ~HPCxx_Thread();
       HPCxx_Thread& operator=(const HPCxx_Thread& thread);
       virtual void run();
       static void stop(void *status);
       static void yield();
       void resume();
       int isAlive();
       static HPCxx_Thread *currentThread();
       void join(long milliseconds = 0,
                 long nanoseconds = 0);
       void setName(const char *name);
       const char *getName();
       int getPriority();
       int setPriority(int priority);
       static void sleep(long milliseconds,
                         long nanoseconds = 0);
       void suspend();
       void start();
};
```

図 2.5　HPCxx_Thread クラスの定義

このクラスを利用したユーザのプログラムは，例えば図 2.6 のようになる．図のように，ユーザは HPCxx_Thread クラスを継承して新たなスレッドクラス（MyThread）クラ

スを定義することにより，独自のスレッドを作成する．

```
class MyThread : public HPCxx_Thread {
 char* x;
 public:
  MyThread(char* y) : x(y) , HPCxx_Thread() {}
  void run() { cout << X << endl << flush; }
};

int main(int argc, char** argv)
{
 HPCxx_Group* g;
 hpcxx_init(argc, argv, g);

 MyThread *t1 = new MyThread("hello\n");
 t1->start();
 sleep(5);

 cout << "DONE" << endl;
 return hpcxx_exit(g);
}
```

図 2.6　HPC++Lib のサンプルプログラム

2.4 DSE(Distributed Supercomputing Environment)

DSE(Distributed Supercomputing Environment)は，我々が開発したクラスタコンピューティング環境であり，ネットワーク接続された計算機上で動作するシステムソフトウェアである．本節では，このDSEの概要を説明し，DSEのメッセージ交換，プロセス管理，共有メモリ管理といった個々の機能について述べた後，プログラミング環境について述べる．

2.4.1 DSE 概要

DSEは，1990年に分散システム上で並列処理を可能とするために開発されたシステムである[52][52][[53][54]．DSEは，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コンピューティング環境で，プログラムの並列実行だけでなく各種並列処理実験のための機能を提供する．以下にDSEの主な特徴を示す．

(1) 並列処理機能

複数の計算機を利用して並列アプリケーションを実行する機能は，DSEの提供する最も基本的な機能である．プロセス起動や共有メモリアクセス，同

期処理などはC言語用のライブラリとして提供されており，ユーザはC言語を用いて並列アプリケーションを作成する．

(2) ネットワークトポロジーの可変性

DSEは，仮想的なネットワークトポロジーをユーザが自由に設定できる機能を備えている．DSEではTCP/IPがホスト間の接続に用いられているため，UNIXの同一プロセスが同時に開くことができるポート数の制限により，多数のホストを直接接続することができない．ネットワークトポロジーの可変性は，様々な接続形状でのアプリケーションの通信パターンのモニタリングとポート数の制限を回避してより多くのホストを接続する目的で利用される．

この場合ホスト間に他のホストを経由して間接接続される部分が発生するが，実行するアプリケーションの性質に合わせて仮想的なネットワークトポロジーを自由に設定することができるため，ネットワークトポロジーを最適に設定すれば通信の増加を防ぐことができる．

(3) DSEカーネル動作のモニタリング機能

DSEでは，DSEが行う通信，共有メモリアクセス，プロセス起動，同期処理などの各種処理の発生をすべてモニタリングし，ファイルに保存する機能を提供している．この機能により，マルチプロセッサシステムでは観察が難しかった共有メモリのアクセスなどの情報を容易に得ることが可能になる．また，このモニタリング機能により，アプリケーションの性質や動作を解析し，把握することができる．

特に(3)の機能は，テスト環境としての色合いが強いDSE独特の機能であり，このモニタリング情報を利用して各種解析を行なっている．

2.4.2 システム構成

DSEは，図2.7に示す分散共有メモリ型並列計算機の機能をネットワーク接続された計算機上に実現する．図のように，各プロセッサ要素は，それ自身がアクセスできる非共有メモリであるローカルメモリ(LM)と，すべてのプロセッサ要素で共有されるグローバルメモリ(GM)を持つ分散共有メモリ構成をとる．また，各プロセッサ要素間は相互結

合網により接続されており，これを経由して他プロセッサ要素の共有メモリアクセスと，プロセッサユニット(PU)間の通信が行われる．

ローカルメモリは，プロセッサが実行する命令コードと非共有データが配置される．プロセッサはグローバルメモリからの命令フェッチは行うことができず，グローバルメモリは共有データの配置だけに利用される．

DSEでは，図2.7の各プロセッサ要素が，1つのDSEカーネルとDSEプロセスの組として実現されている．以降，このDSEの提供するプロセッサ要素を「仮想プロセッサ」と呼ぶことにする．また，グローバルメモリは，DSEカーネルが管理するメモリに対応し，ローカルメモリはUNIXオペレーティングシステムがUNIXプロセスのために確保するメモリに対応する．

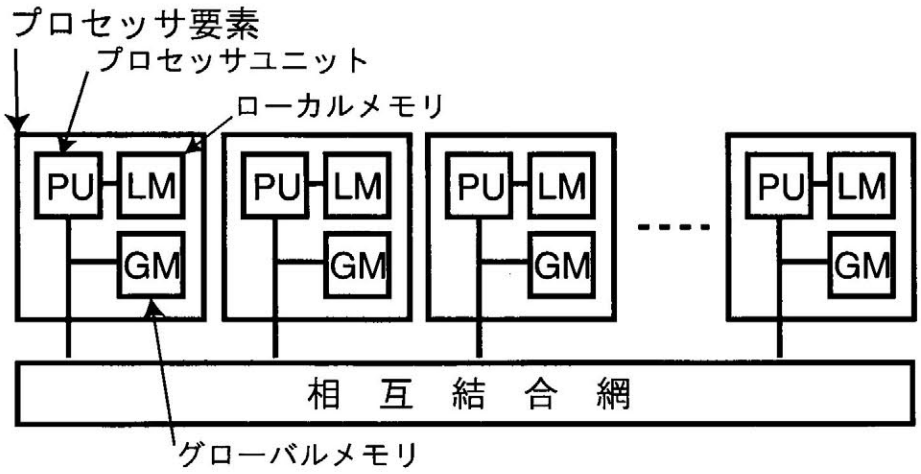


図 2.7 DSEの提供する並列計算機のモデル

ソフトウェア構成

このDSEのソフトウェア構成を図2.8に示す．DSEでは仮想プロセッサはUNIXプロセスにより実現される．また，初期のDSEでは，DSEカーネル(DSE kernel)とDSEプロセス(DSE process)は異なる2つのUNIXプロセスで実現されていたが，高性能実装を目標として開発されたSunOS向けに特化したバージョン以降はDSEカーネルとDSEプロセスの2つを1つのUNIXプロセスとして実現している[51]．また，仮想プロセッサ間はLAN(Local Area Network)により接続される．この仮想プロセッサ間の通信にはプロトコルとしてTCP/IPを用いており，DSEのシステム起動時にコネクションの確立が行われ

る．

なお，DSEカーネルでは，共有メモリの管理，DSEプロセスの管理，他の仮想プロセッサとのメッセージ交換を行う．DSEプロセスは，ユーザのアプリケーションを実行する部分であり，DSEカーネルとライブラリ(DSElib)を介した通信を行うことで並列処理を実現する．なお，DSEでは，サーバは存在せずすべてのDSEカーネルが等価な機能を提供する．従って，ユーザアプリケーションは，どの仮想プロセッサからでも起動することができるし，実行中断などを行うことができる．

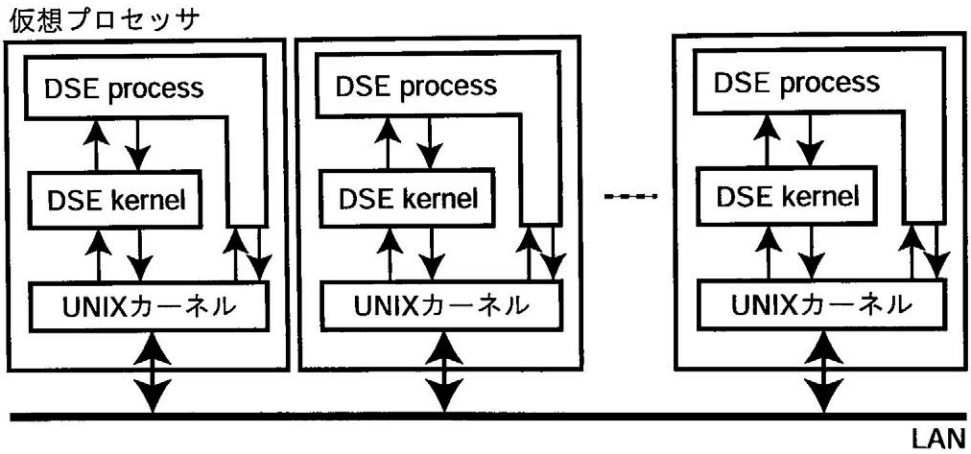


図 2.8 DSE ソフトウェア構成（１）

以下，DSEカーネルとDSEプロセスの内部モジュールについて詳しく説明を行う．図2.9は，１つの仮想プロセッサを構成するDSEのソフトウェア構成である．図のように，DSEカーネルは5つのモジュールより構成される．各モジュールは，基本的に独立しており，個々に変更が可能な構成をとる．これまでの実験では，共有メモリの実装として論理アドレスを持ちメモリをページ単位で管理するものと，論理アドレスと物理アドレスが一致するものの２つが実装されている．なお，この２つの変更は，共有メモリ管理モジュールの換装のみのよって行われた．

図中の矢印は，各モジュールの関係を示している．例えば，メッセージ解析モジュールとメモリ管理モジュールの間の矢印は，メッセージ解析モジュールとメモリ管理モジュールの間に呼び出し関係があることを示している．

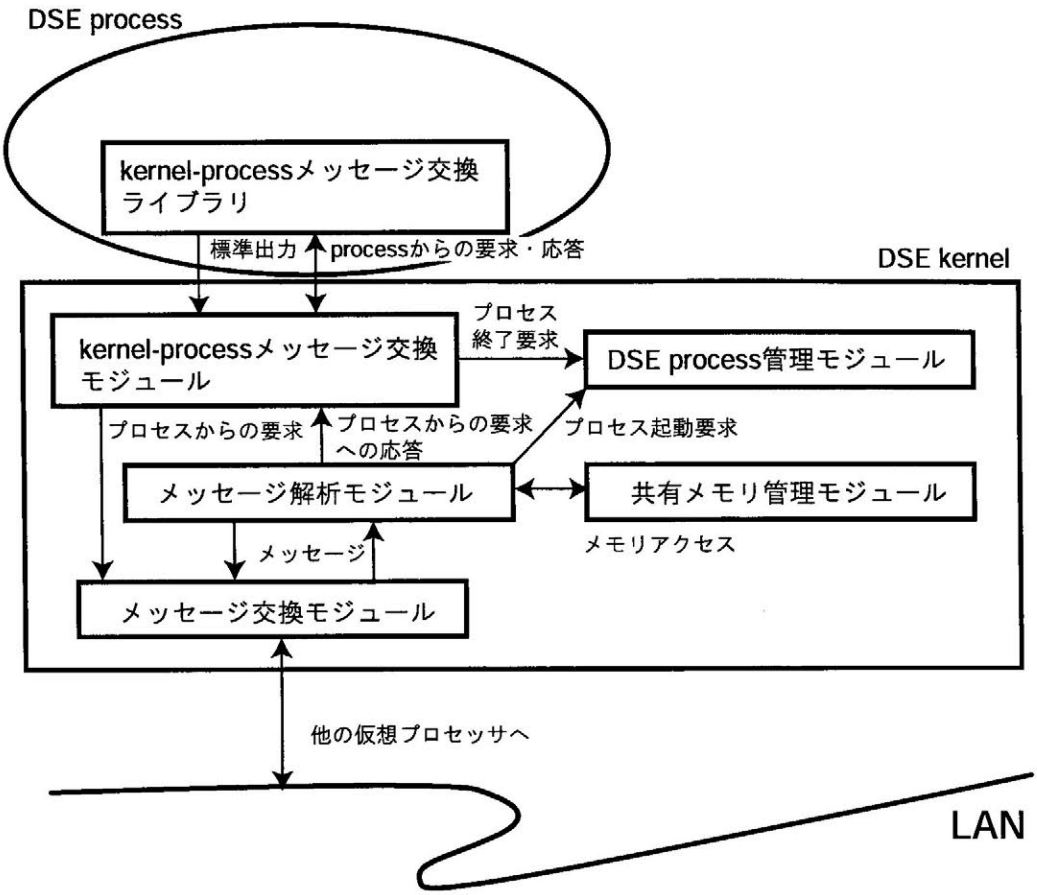


図2.9　DSEソフトウェア構成（2）

共有メモリアクセスのREAD手順

ここで,DSEの動作の例としてDSEプロセスが,他仮想プロセッサの共有メモリをreadする手順について説明する．ユーザプログラムからの共有メモリread要求は，kernel-processメッセージ交換ライブラリからkernel-processメッセージ交換モジュールへと伝えられる．ここで，readするメモリが仮想プロセッサ内かどうかチェックされ，ここでは，他の仮想プロセッサの共有メモリへのアクセスであるため，メッセージが生成されメッセージ交換モジュールに渡される．渡された共有メモリreadメッセージは，LANを経由して指定された仮想プロセッサへと送られ，送信先のメッセージ交換モジュールにより受信される．メッセージを受信したメッセージ交換モジュールでは，受信したメッセージをメッセージ解析モジュールへと渡す．メッセージ解析モジュールでは，これが共有メモリreadであることを判定し，指定されたアドレスの内容を読み出し，read応答メッ

セージを作成し，メッセージ交換モジュールへと渡す．メッセージ交換モジュールでは，要求元の仮想プロセッサへこのメッセージを送信する．要求元メッセージ交換モジュールで受信されたread応答メッセージは，メッセージ解析モジュールへ渡され，次にkernel-processメッセージ交換モジュールとライブラリを経由してユーザプログラムへと渡される．

以上のように，共有メモリアクセスは仮想プロセッサ間でのメッセージ交換により実現されている．また，メッセージ交換モジュールでは，受信したメッセージを１つの受信キューにより管理するため，メッセージ解析モジュールでは，受信した順番にメッセージの処理が行われることになる．したがって，複数の仮想プロセッサが同時に同一メモリにアクセスした場合は，共有メモリアクセスはメッセージの到着順に処理が行われることになる．また，DSEでは共有メモリのキャッシングは行われず，要求発生時には直接共有メモリを読みにいくため，TreadMarkのような共有メモリの一貫性を保つためのプロトコルは必要としない．しかしながら，共有メモリアクセスの度に，ネットワークを経由した通信が発生するため，頻繁に共有メモリアクセスを行うと効率が悪い．そこで，DSEで効率の良いアプリケーションを作成するには共有メモリの内容をまとめて読み出してローカルメモリにコピーし，結果についてもまとめて書き出すといった，ユーザによるプログラム側で通信量を減らす工夫が必要になる．

以下では，DSEのメッセージ交換，プロセス管理，共有メモリ管理機構について説明する．また，DSEの提供するプログラミング環境について説明する．

2.4.3 メッセージ交換機構

接続情報ファイル

メッセージ交換機構では，DSEの起動時の計算機間の接続と，終了時の切断，およびメッセージの送受信を受け持つ．ＤＳＥでは，計算機の接続は接続情報ファイル(Connection Information File, CIF)と呼ぶ情報を利用して行う．このCIFは，図2.10の構文規則に基づくファイルであり，基本的に接続する計算機名と接続ポートおよび接続方法が列挙されたリストである．

File → Machine File | Machine

Machine → Mname Connect

Connect → Con Connect | Con

Con → Name Port Type

Mname → (15 文字の文字列) ':' | (14-No の長さの文字列) '!' No ':'

Name → (16 文字の文字列) | (15-No の長さの文字列) '!' No

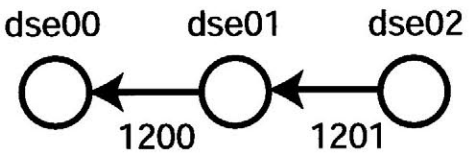
Port → 数字

No → 数字

Type → "connect" | "accept" | "port"

図 2.10 CIF の構文規則

CIF を用いて図 2.11(a)のような３プロセッサの接続を定義する場合，例えば図 2.11(b)のように記述する．この例では，dse00，dse01，dse02 の３プロセッサで仮想的な鎖網を構成している．図 2.11(b)のように，"dse00:" から次の "dse01:" までの間は，dse00 に対する接続の情報である．この例では，dse00 は dse01 と直接接続（accept 側）しており，dse02 とは dse01（1200 番のポート）を経由して接続していることを示している．また，dse01 は dse00 とは "connet" として接続している．このように，直接接続の場合には，"accept" と "connect" が対になるように記述する．なお，この accept と connect は TCP/IP を用いたソケット接続時の accept(), connect()に対応している．DSE では，このような接続ファイルを記述することで自由な仮想ネットワークを実現することが可能である．



(a)

```
dse00:
dse01  1200  accept
dse02  1200  port
dse01:
dse00  1200  connect
dse02  1201  accept
dse02:
dse00  1201  port
dse01  1201  connect
```

(b)

図 2.11 CIF 記述例（３プロセッサ鎖網）

通信メッセージフォーマット

次に，DSEカーネル間の通信に用いられるメッセージのフォーマットについて説明する．図2.12に示すように，DSEのメッセージには送信元(Source)２バイトと送信先(Destination)２バイト，そしてメッセージの長さを示す４バイトの計８バイトのヘッダがあり，その後ろにデータが続く形になっている．データの部分には，メッセージ解析モジュールで解析されるメッセージ本体が格納されていおり，共有メモリのアクセスやプロセス起動などによって異なるフォーマットとなる．このように，DSEのメッセージは，メッセージ交換モジュールで交換されるメッセージのデータ部分に解析モジュールのメッセージが含まれた階層構造をとる．

なお，送信元は応答データ作成のために，送信先は他仮想プロセッサを経由して通信を行う（接続タイプがportである場合）のルーティング等に利用される．

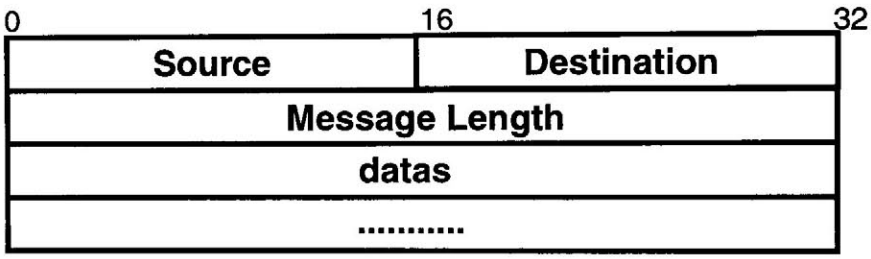


図2.12　メッセージフォーマット

また，前述したようにメッセージの送信および受信は先着順(FCFS, First-Come First-Served)で行われる．

2.4.4 プロセス管理機構

ここでは，DSEのプロセス管理機構について説明する．ここでは，まず，DSEにおけるタスクと軽量プロセス（LWP, Light-Weight Process）の定義を以下に示す．

・タスク(Task)

１つ以上の仮想プロセッサより実行される．共有メモリなどの資源は，このタスクに対して個々に割り当てられる．タスクは，基本的にユーザアプリ

ケーションに一致する．なお，タスクは１つ以上の軽量プロセスにより構成される．

・軽量プロセス（LWP）

１つの仮想プロセッサ上で実行されるプロセッサ割り当ての単位．同一プロセッサ上で実行されるLWP間ではローカルメモリの共有が可能である．

以上のように，軽量プロセスは仮想プロセッサに割り当てられ，またタスクは１つ以上の仮想プロセッサで実行される．UNIXの表現でいえば，タスクはプロセスに，軽量プロセスはスレッドに対応する．

なお，資源の割り当て単位であるタスクには，システムで一意な識別子であるシステムプロセスID(SPID)が割り当てられる．DSEでは，このSPIDを用いて資源の割り当て及び管理を行う．SPIDは32ビットのIDであり，上位16ビットがタスクを起動した仮想プロセッサの番号，下位16ビットがUNIXプロセスに割り当てられるプロセスIDにより構成される．UNIXのプロセスIDは計算機内で一意であり，また，仮想プロセッサ番号はDSEを構成する仮想プロセッサ内で一意であるため，SPIDがDSE内のタスクで一意であることが保証される．

このDSEでのLWPの割り当ては，先着順(FCFS)で行われる．また，１つの仮想プロセッサで同時に起動されるLWPの数は常に１であり，起動要求が行われたLWPはFIFOキューにより格納され，現在のLWPの終了後，起動される．なお，この割り当てスケジューリングはFIFOキューによるタスク起動を同期手段とする並列プログラSPP[53]による並列アプリケーションの実行を可能にする．

2.4.5 共有メモリ管理機構

DSEカーネルが保持する共有メモリは，デフォルトで各仮想プロセッサ毎に256Kバイトに設定されている．したがって，システム全体の総共有メモリサイズは（仮想プロセッサ数）×256Kバイトとなる．

図2.13は，DSEの持つ共有メモリの論理アドレス空間を示したものである．図のように，共有メモリは40ビットのメモリ空間を持ち，上位16ビットが仮想プロセッサ番号を，下位24ビットが仮想プロセッサ内アドレスとなっている．実際の共有メモリ（物理

空間）は各仮想プロセッサ毎に256Kバイトであり，これは4Kバイト単位で64ページに分けられている．この物理空間は，4Kバイト単位で論理空間にマッピングされ利用される．このため，64ページを越える参照が行われた場合にはmemory overflowとなり，システムの実行は中断される．

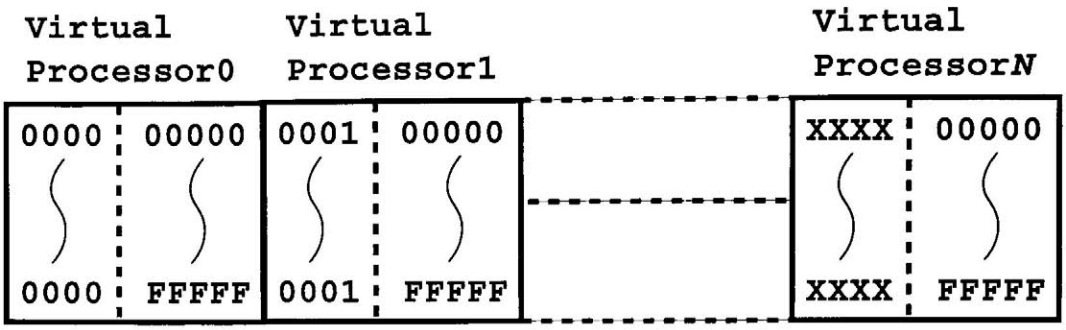


図2.13　共有メモリのアドレス空間

論理-物理アドレス変換

次に，ユーザのアプリケーションの利用する共有メモリの論理アドレスから物理アドレスへの変換について図2.14を用いて説明する．DSEでは，各ページの管理にハッシュを用いている．ユーザプログラムから論理アドレスを与えられると，論理アドレスとシステムプロセスID(SPID)からハッシュキーが作成される（図中(1)）．次に，キーを利用してハッシュテーブルが検索され，該当するテーブルエントリが検索される(図中(2))．なお，該当するエントリが存在しない場合には，新たにページを確保しハッシュテーブルとページテーブルを更新する．ハッシュにエントリが存在する場合は，そこからページテーブルを参照しページ番号を得る(図中(3))．このページ番号と論理アドレスのアドレス部から物理アドレスが作成され(図中(4))，共有メモリの該当アドレスにアクセスを行う(図中(5))

このように，共有メモリのアクセスではSPIDと論理アドレスから物理アドレスへの変換が行われる．なお，先に述べたように，各仮想プロセッサの共有メモリサイズは変更可能であり，ページサイズの変更，ページ数の変更または双方の組み合わせによりサイズを変更することができる．

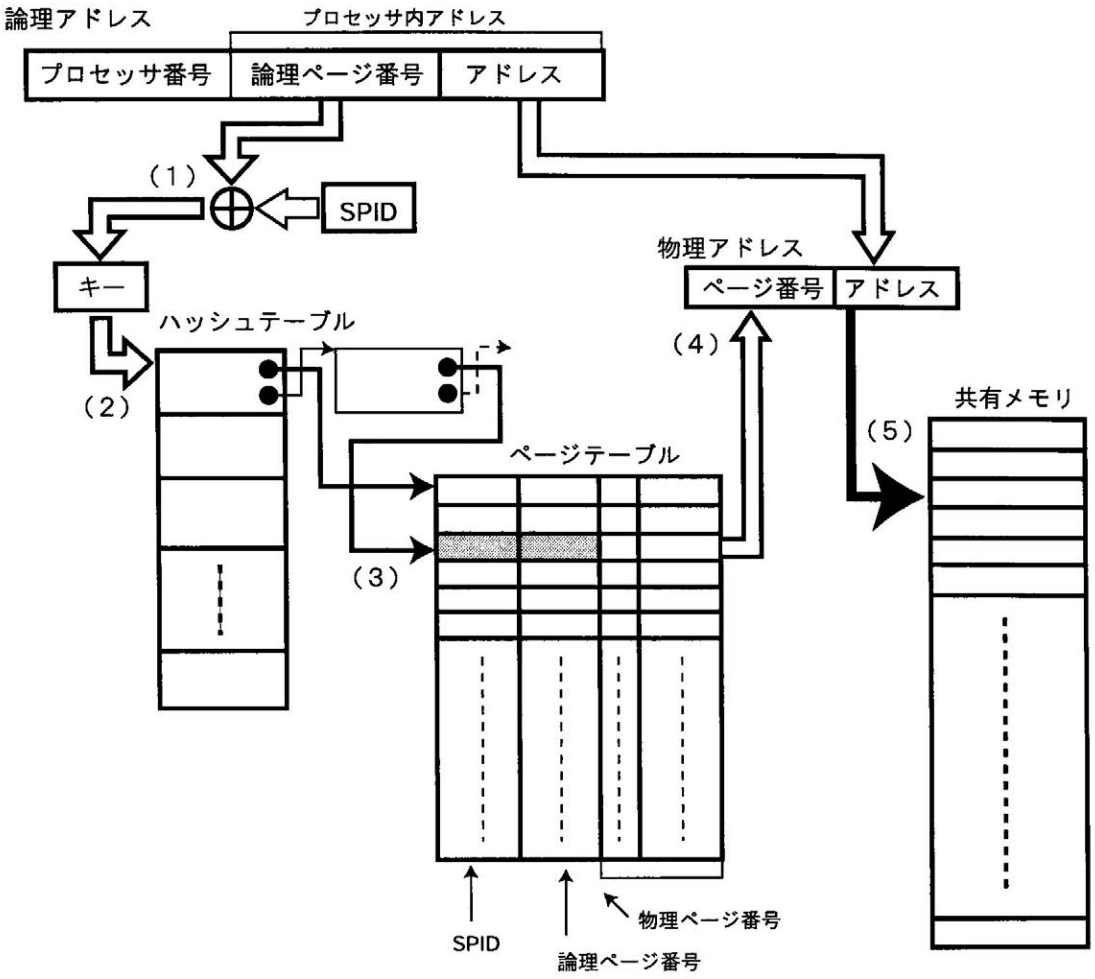


図 2.14　共有メモリのページングメカニズム

2.4.6 プログラミング環境

DSEでは，ユーザが並列アプリケーションを記述するたにDSElibとよぶC言語用のライブラリ(API)を提供している．このライブラリは，以下のような特徴を持つ．

・関数体系の統一

DSElibの提供する関数は，すべて"DSE_"で始まる．また，共有メモリのアクセスであれば，"DSE_shmem_"から始まるなど，機能別に分類され，表記の統一が行われている．

・軽量プロセス操作の簡単化

DSElib では，すべてのプロセッサに対して同一の LWP を起動する関数(DSE_lwp_reqcall())や，他プロセッサでのLWP起動要求後，要求したLWPが終了するまで実行を中断して待つ関数(DSE_lwp_call())など，多数のLWP操作関数が提供される．

・同期処理関数の体系化

DSEの同じ機能を使う場合であっても，セマフォやバリア同期などの目的別に関数が用意される．

図 2.15 に DSE のプログラムのサンプルを示す．図のように，DSE のプログラムでは，プロセスの起動要求(DSE_lwp_request)，プロセス終了(DSE_lwp_exit)，同期処理(DSE_barrier_init, DSE_barrier_signal, DSE_barrier_wait)などを適切に挿入することでプログラムの流れを制御する．

本章の最後に，表 2.1 に DSElib のライブラリ一覧を示す．

```
Process (*func[])()={P_main, P_1, P_2, P_3, P_4, P_5,P_6}

main(int argv, char  **argc)
{
  DSE_Init(func);
  DSE_lwp_open();
  DSE_Close();
}


Process P_main()
{
  DSE_barrier_init(SEM_A, 3); ←―― バリア同期変数の初期化
  DSE_barrier_init(SEM_B, 2);
  DSE_lwp_request(0, P_1, NULL, 0); ←―― 軽量プロセスの起動を要求
  DSE_lwp_request(1, P_2, NULL, 0);
  DSE_lwp_request(2, P_3, NULL, 0);
  DSE_lwp_exit();
}

Process P_1()
{
  DSE_shmem_write_int(MEM_A, 10); ←―― 共有メモリへの書き込み
  DSE_barrier_signal(SEM_A); ←―― バリア同期処理
  DSE_lwp_exit();
}

Process P_2()
{
  DSE_shmem_write_int(MEM_B, 20);
  DSE_barrier_signal(SEM_A); ←―― バリア同期処理
  DSE_barrier_wait(SEM_A); ←―― バリア同期を利用したプロセス終了待ち
  DSE_lwp_request(1, P_4, NULL, 0);
  DSE_lwp_request(2, P_5, NULL, 0);
  DSE_lwp_exit();
}
```


図 2.15　プログラムサンプル

初期化関数	
init	DSE_Init(Process(*[])())
void	DSE_Close(void)
LWP制御関数	
void	DSE_lwp_open(void)
void	DSE_lwp_close(void)
void	DSE_lwp_call(unsigned short, Process(*)(), char*, unsigned long)
void	DSE_lwp_return(char*, unsigned long)
void	DSE_lwp_request(unsigned short, Process(*)(), char*, unsigned long)
void	DSE_lwp_exit(void)
void	DSE_lwp_reqcall(unsigned short, unsigned short, Process(*)(), char*, unsigned long)
void	DSE_lwp_exreturn(void)
共有メモリアクセス	
void	DSE_shmem_read(unsigned short, unsigned long, char*, unsigned long)
void	DSE_shmem_write(unsigned short, unsigned long, char*, unsigned long)
void	DSE_shmem_writeS(unsigned short, unsigned long, char*, unsigned long)
short	DSE_shmem_read_short(unsigned short, unsigned long)
void	DSE_shmem_write_short(unsigned short, unsigned long, short)
void	DSE_shmem_writeS_short(unsigned short, unsigned long, short)
int	DSE_shmem_read_int(unsigned short, unsigned long)
void	DSE_shmem_write_int(unsigned short, unsigned long, int)
void	DSE_shmem_writeS_int(unsigned short, unsigned long, int)
long	DSE_shmem_read_long(unsigned short, unsigned long)
void	DSE_shmem_write_long(unsigned short, unsigned long, long)
void	DSE_shmem_writeS_long(unsigned short, unsigned long, long)
double	DSE_shmem_read_float(unsigned short, unsigned long)
void	DSE_shmem_write_float(unsigned short, unsigned long, double)
void	DSE_shmem_writeS_float(unsigned short, unsigned long, double)
double	DSE_shmem_read_double(unsigned short, unsigned long)
void	DSE_shmem_write_double(unsigned short, unsigned long, double)
void	DSE_shmem_writeS_double(unsigned short, unsigned long, double)
セマフォ操作関数	
void	DSE_sem_init(unsigned short, unsigned long, short)
void	DSE_sem_signal(unsigned short, unsigned long)
void	DSE_sem_wait(unsigned short, unsigned long)
バリア同期関数	
void	DSE_barrier_init(unsigned short, unsigned long, short)
void	DSE_barrier_signal(unsigned short, unsigned long)
void	DSE_barrier_wait(unsigned short, unsigned long)
ロック・アンロック関数	
int	DSE_lock(unsinged short, unsigned long)
void	DSE_unlock(unsigned short, unsigned long)
フェッチ・アンド・アッド関数	
void	DSE_faa_init(unsigned short, unsigned long, int)
int	DSE_faa(unsigned short, unsigned long, int)

表 2.1 DSElib ライブラリ一覧

第３章

DSEの高性能実装と評価

3.1 概要

前章で述べたように，DSEの通信処理機構では，通信プロトコルとしてTCPが利用されている．TCP接続を行う場合，UNIXなどで利用されているソケットインタフェースでは接続1つに対してポートと呼ばれるOS資源を1つ消費する．このため，n個の接続を行う場合には，n個のポートが必要になる．このポートは，プロセス毎に割り当て可能な最大数が決められており，例えば一般的なUNIXオペレーティングシステムでは，この数は32または64である．したがって，UNIXのユーザプロセスとして実装されるDSEの仮想プロセッサの場合，直接接続できる仮想プロセッサの数はポート数に制限されることになる．また，このようにオペレーティングシステムの持つ資源を大量に消費することは少なからず問題となる．このポートの消費の問題は，DSEのシステム全体で考えるとさらに深刻であり，例えば仮想プロセッサ数が60の場合には，完全網接続であれば全ての計算機の合計で3,540ポートを消費してしまう計算になる．

このポート数の制限と大量消費問題を回避するために，DSEでは仮想的なネットワーク接続を行うことにより消費ポート数を削減することが可能な設計になっている．例えば，先の例において完全接続からリング網接続に変更すれば，消費ポート数は3540ポートから60ポートまで，実に3480ポートを削減できる．しかしながら，完全接続以外の仮想ネットワークを利用する場合，直接接続されていない計算機同士の通信は，他の計算機を経由した通信となるため，通信オーバヘッドが増加してしまうという問題がある．

そこで，本研究では，このようなDSEのポート消費問題を解決し，かつ高速な通信を可能する手段として，これまでのTCP接続に代わり通信プロトコルとしてUDPを用いて

通信を行う方法をとった．UDPは，コネクションレス型の通信プロトコルであり，TCPと異なり1つのポートにより複数の計算機との通信が可能なプロトコルである．しかし，一方では，UDPはデータの到着を保証しないなど，DSEのような高信頼性を要求する場面において欠点となる要素を持っている．

そこで，本研究では，この信頼性の欠如を補うために，UDPの上に信頼性を実現するプロトコル(DMP, DSE Multi-communication Protorol)を実装した．DMPは，ユーザプロセスレベルで実現されるもので，OSに変更を加える必要がない．このため，OSを変更せずに動作可能であるというDSEの特徴を損なうことなく，少ないポートによる通信を実現することができる．

本章では，設計したプロトコルの詳細設計について述べるとともに，実装したプロトコルを用いた評価実験の結果について述べる．

3.2 UDPを用いた通信機構

DMPは，基本的にはTCPのサブセット的な機能を提供するが，以下の2点においてTCPと大きく異なる．

(1) 1つのポートで複数の計算機との送信メッセージをを受信し，それぞれのメッセージ毎に組み立てる機能．

(2) TCPで利用されているスライディングウィンドウ方式にくらべて，簡易なフロー制御の採用．

(1)の機能は，DSEの通信プロトコルとして利用するために必要な機能であり，設計するプロトコルの必須機能である．(2)の機能は，プロトコル設計の予備実験の結果[55]よりフロー制御が必要であることがわかったため，なるべく少ないオーバヘッドでフロー制御を行うために付加した機能である．

なおDMPは，DSEの通信部として利用するため，各計算機からのメッセージを最終的に1つのメッセージキューで渡すなど，インタフェース部についてDSEとの接続を考慮した設計となっている．

プロトコルヘッダ

プロトコルの動作の説明の前に，プロトコルのヘッダを図3.1に示す．ヘッダ中のSrc. CPU No. およびDes. CPU No. は，受信したパケットの送信元および受信先の仮想プロセッサ番号である．本プロトコルでは，この仮想プロセッサ番号により，受信したパケットの振り分けと組み立てを行っている．また，Src. Seq No. およびDes. Seq No. は，各仮想プロセッサ間毎の通信に対してに割り当てられるシーケンス番号である．このプロトコルでは，このシーケンス番号を利用してパケットロス時の再送処理などを行っている．なお，シーケンス番号を用いたこれらの処理については，後の節で詳しく説明する．

以上の情報は，主にパケットの到着順序確認やパケット喪失のチェックのための情報である．それ以外のData Lengthと Fragment No. およびMax Fragmentは，DSEの通信単位であるメッセージをパケットに分割したり，パケットからメッセージを組み立てるために利用する情報である．

また，Flagsには通信制御のためのフラグビットが格納されており，現在のところACKが付加されているかどうかを示す1ビットのみを利用している．このACKフラグが立っているパケットはACKパケットであることを示しており，フラグを利用することでACKを通常のパケットに付加して送信している．したがって，双方向で通信を行っている場合には，相手へ送るパケットにACKフラグを付加して送ることで，「ACKのみ」のパケットによる通信量の増加を抑えることが可能である．

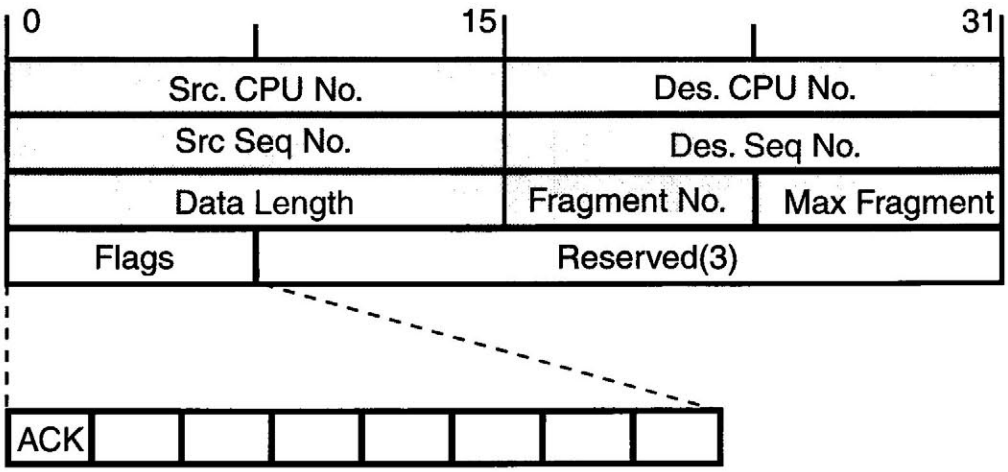


図3.1　プロトコルヘッダ

3.3 メッセージ分割と組み立て

DMPでは，同一ポートで受信した複数の仮想プロセッサからのパケットを振り分け，DSEのメッセージに組み立て直す処理を行う．この処理を行うために，本プロトコルでは，受信したパケットを各仮想プロセッサ毎に一時的に保存しておくバッファと，組み立て後のパケット（メッセージ）を格納するための１つの受信バッファを持つ．以下，図3.2を用いてメッセージ受信・組み立て処理について説明する．図3.2中のN-m/nという表記は，Nがパケットの番号を，nがメッセージの分割数を，そしてmがパケットに分割されたメッセージの何番目であるのかを示している．また，送信(1)(2)は送信側の仮想プロセッサを，buf1,buf2は２つの送信側の仮想プロセッサ1,2に対応する一時受信バッファ，そしてrecvは組み立てたメッセージを格納する受信バッファを示している．

図では，送信(1)ではパケット番号０～３のパケットを，送信(2)ではａ～ｄのパケットをそれぞれ送信している．なお，このパケットのうち，１と２およびａとｂはそれぞれ2つで1つのメッセージである．また，受信側では，送信されたパケットを図3.2に示すようにa,0,1,b,c,2,3,dの順で受信するとする．以下，このメッセージを受信した場合の受信側の動作について説明を行う．

送信(2)から送信されたパケットaは，2つに分割されたメッセージの1つめのパケットであるため，受信後buf2に一時的に保管される．次に，送信(1)が送信したパケット０を受信する．このパケット0は，分割されていないパケットであるため，一時受信バッファには格納されず，パケット中のメッセージが取り出された後，直接受信バッファに格納される．続いてパケット１を受信するが，このパケットは分割されたパケットであるため，buf1に格納される．次に，パケットｂを受信する．このメッセージは分割されたパケットの最終パケット(2/2)であるので，先にbuf1に格納されていたパケットａと合わせることでメッセージの組み立てが行われ，組み立てられたメッセージが受信バッファに格納される．以下，図中の(5)～(8)の順で処理が行われ，結果として受信バッファに受信したすべてのメッセージが格納される．

以上のように，DMPでは，仮想プロセッサ毎に個別の一時受信バッファを設けることで，同一ポートで受信したパケットの仮想プロセッサ毎の個別組み立てを可能にしている．また，パケットサイズより小さいパケットは，この一時受信バッファを介さずに直接バッファに格納するようにしたことで，パケット受信時のコピー処理を削減し，高速化を図っている．

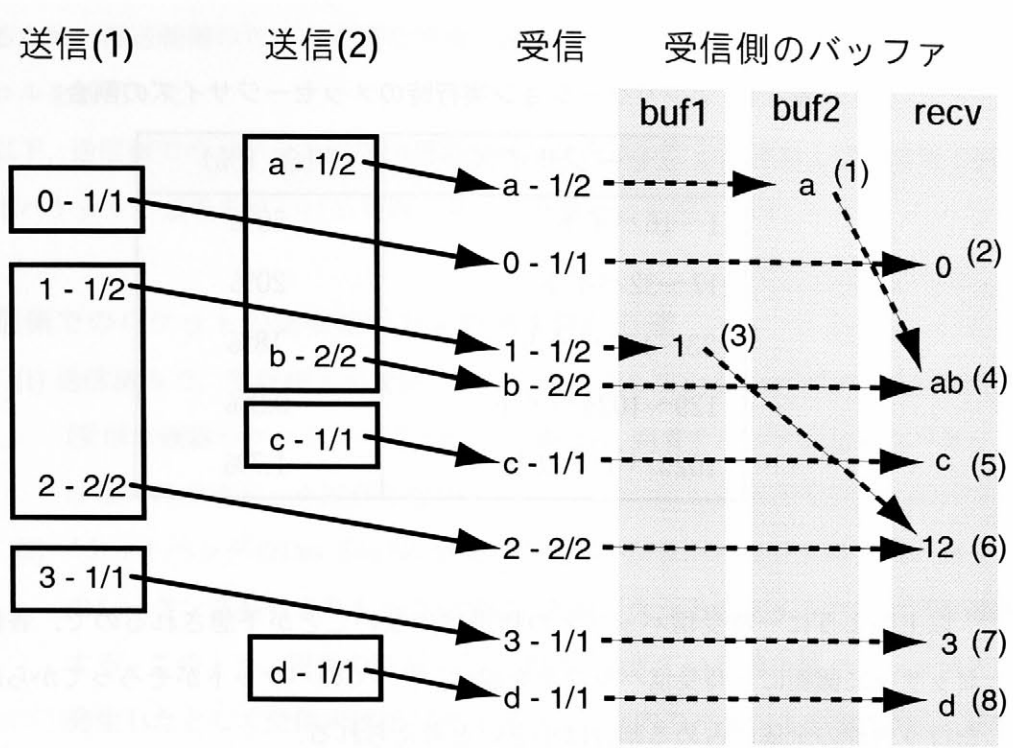


図 3.2　メッセージの分割と組み立て例

バッファサイズの検討

表3.1は，DSE上で動作する５つのアプリケーション(巡回騎士問題，オセロ，行列積，偏微分方程式(SOR)，画像に対する離散コサイン変換(DCT))を実行した時の実行履歴から，DSEのシステムが交換したメッセージを抽出し，サイズ毎に集計した結果である．これをみると，32バイト以下のメッセージで全体の80%を，128までのメッセージサイズを含めれば98%のメッセージがこのサイズ以下であることが分かる．以上のように，代表的なDSE用アプリケーションでは，実行中に送受信されたほとんどのメッセージが128バイト以下である．したがって，DMPのパケットサイズを128バイトより大きくすれば，DSEのメッセージの98%は１パケットで送信され，一時受信バッファを介さずに直接受信バッファに格納されることになる．この時，一時受信バッファに格納されるメッセージは全体のメッセージの2%となる．

表 3.1　アプリケーション実行時のメッセージサイズの割合

メッセージサイズ	割合（%）
1〜16バイト	60%
17〜32バイト	20%
33〜128バイト	18%
129〜1024バイト	0.3%
1025バイト以上	1.7%

以上のように一時受信バッファの利用は少ないことが予想されるので，各仮想プロセッサ毎に個別に一時受信バッファを設け，すべてのパケットがそろってから組み立てを行う処理の全体に占める割合は小さいと考えられる．

3.4 再送制御

DMPは，受信したパケットのDes. Seq No.をチェックし，送信したが相手が受け取っていないパケットがある場合に再送するという処理を基本として，再送処理を行っている．しかしながら，これだけでは，相手からパケットが送られてこなければパケットが届いたかどうかの確認を行うことができないので，これに加えてタイムアウト制御を利用している．タイムアウト制御では，相手からのパケットを受信したのち一定時間以上相手に対してパケットを送信することがない場合にACKのみのパケットを返すという処理行う．また，送信側では，一定時間以上このACKパケットが返ってこない場合には再送する．なお，このタイマはACK待ちを行っている相手から新しいパケットを受け取った場合には再設定され，結果として常に最後にパケットを受け取った時点から一定時間経過するまで待つことになる．

なお，パケットの消失などのエラーが発見された場合には，Go-Back-Nプロトコルに基づいてパケットの再送が行われる．Go-Back-Nプロトコルでは，たとえ届いているパケットがあるとしても，エラーが発生したパケット以降のパケットがすべて再送されてしまうという欠点があるが，DMPを利用する環境であるDSEの想定するネットワークはLANなどの狭いネットワークであり，このようなパケットロスの発生は少ないと考えら

れるため，再送制御のために必要な情報（通信するデータ量）が少ないGo-Back-Nプロトコルを採用した．

以下，送信側でのパケットの送信確認，パケット消失時の処理および受信側で消失またはパケットの重複を見つける処理と見つけた場合の処理を示す．

送信側でのパケットの送信確認および消失時の処理

(1) 送信済みで，現在相手が受信したことを確認していないパケットがバッファ(受信未確認パケットバッファ)にない場合は，到着チェックが必要なパケットはないのでチェックは行わない．

(2) パケットヘッダのDes. Seq No.が受信未確認パケットバッファの先頭パケットのシーケンス番号よりも小さい場合，先頭パケットのタイムスタンプを確認する．このとき，現在の時刻から一定時間過ぎている場合はパケット消失が発生したとして受信未確認パケットバッファ内のパケットを取り出し，送信予約を行う．これにより，受信未確認パケットバッファ内のパケットが再送対象として登録される．なお，パケットのACKフラグが立っている場合には，例外として時刻の経過に関係なく送信予約処理を行う．

(3)ヘッダ内のDes. Seq No.が受信未確認パケットバッファの先頭パケットのシーケンス番号より大きい場合，受信未確認パケットバッファからシーケンス番号がDes. Seq No.よりも小さいパケットを削除する．これにより，Des. Seq No.より小さいシーケンス番号のパケットの受信確認が完了する．

受信側でのパケット消失および重複の発見処理

(1) 受信したパケットのシーケンス番号Src. Seq No. が，受信済みのパケットのシーケンス番号＋１になっていない場合，以下の処理を行う．

(a)受信済みのパケットのシーケンス番号よりSrc. Seq No.が小さい場合には，パケットを重複して受け取ったものとして受け取ったパケットをそのまま破棄する．なお，このようなケースが発生するのは，ACKパケットが送信側に届かなかった場合か，送信側の再送タイマのタイムアウトがACKパケットの到着より先に発生した場合である．

(b)受信済みのパケットのシーケンス番号＋１よりもSrc. Seq No. が大きい場合には，パケットの消失が発生したものとして，ACKパケットを生成する．な

お, 送信予約されたパケットが存在する場合には, ACKパケットを生成せず, 送信予約されたパケットのACKフラグを立てる.

3.5 フロー制御

UDPでは, 受信側のバッファがいっぱいになった場合, 送信側から送られてくるデータは捨てられてしまい, パケットの消失が発生してしまう. このため, 大量のデータをUDPで送信する場合には, バッファオーバフローによるパケットロスが多発してしまう可能性がある. 図3.3は, 10,000回連続して片方向にデータを送信した場合のエラーパケットの発生率を示したものである. このように, UDPでは, もっともエラー率の低い16バイトのデータ転送の場合でも約72%のエラーが発生することが分かる.

転送エラーの発生率

図3.4は, 16バイトのデータ送信時の受信エラーがどのパケットで発生したのかを示すグラフである. このグラフでは横軸にパケット番号をとり, エラーが発生したパケットの部分に縦線が引かれている. これを見ると, 最初の230パケット程度までの受信では, エラーは発生しておらず, エラーが発生しだしてからは定期的にパケットロスが発生していることが分かる. このグラフは, 最初のうちは受信バッファが空であり, 送信されたデータを蓄えることができるが受信処理が間に合わないため受信バッファがいっぱいになりパケットロスを引き起こしていることを示している.

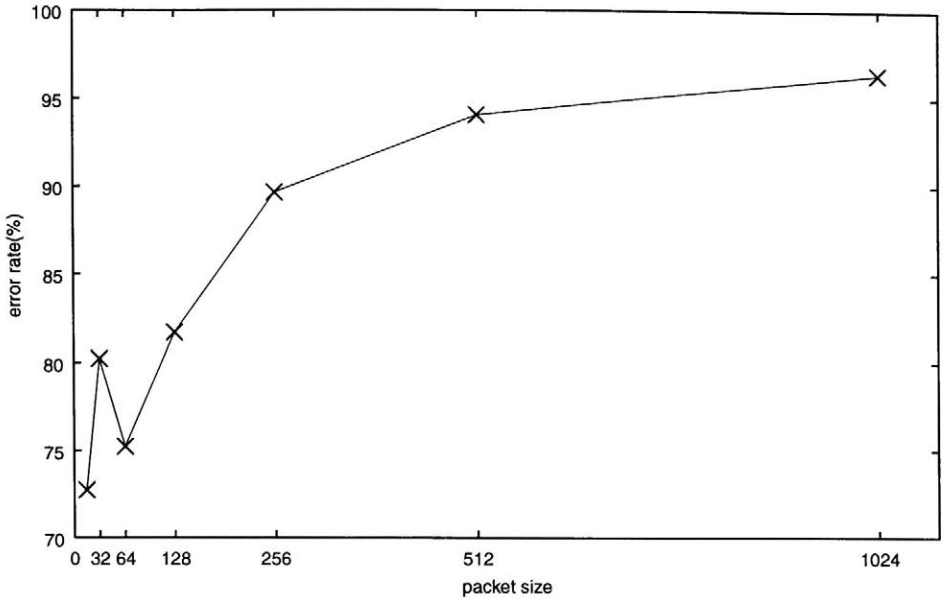


図 3.3　UDP を用いた連続送信時のエラー率

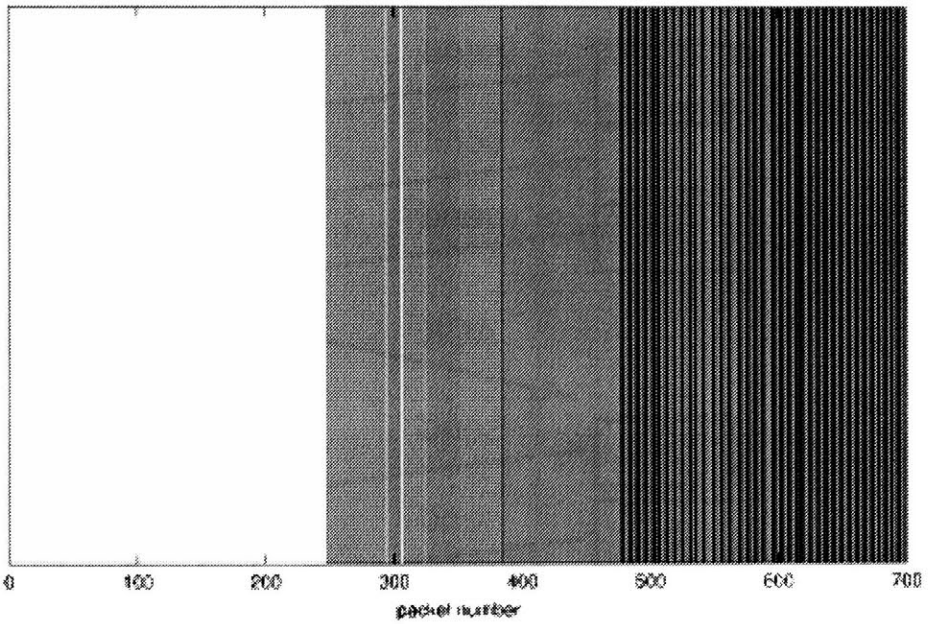


図 3.4　16 バイトのデータ送信時の受信エラーの発生

以上より，エラーの発生の原因は主にUDPの受信バッファのオーバフローであり，適切なフロー制御を設けることによりこのエラーの発生を削減できることが分かる．

フロー制御の実装

そこで，DMPでは，(1)このようなパケットロスが発生しないようにするためのフロー制御と，先に説明したように(2)パケットロスが発生した場合の再送処理を実装している．

図3.5は，DMPに実装したフロー制御を示したものである．DMPのフロー制御では，送信側ではいくつかのパケットをまとめて送信し，その後ACKパケットが返ってくるまで，残りのパケットの送信を止める．図は，この連続送信数が3である場合の例である．まず，送信側が1,2,3のパケットを受信側に送信する．これを受け取った受信側ではACKパケットを生成し，送信側に返す．このACKパケットを受け取った送信側では，送信予約されているパケットから3つのパケットを取り出し送信する．このように，連続送信パケット数を制限することで連続受信時の受信バッファ溢れの問題を解消できるとともに，受信バッファ溢れによるパケットロスに伴う再送による無駄な通信の増加を抑制できる．

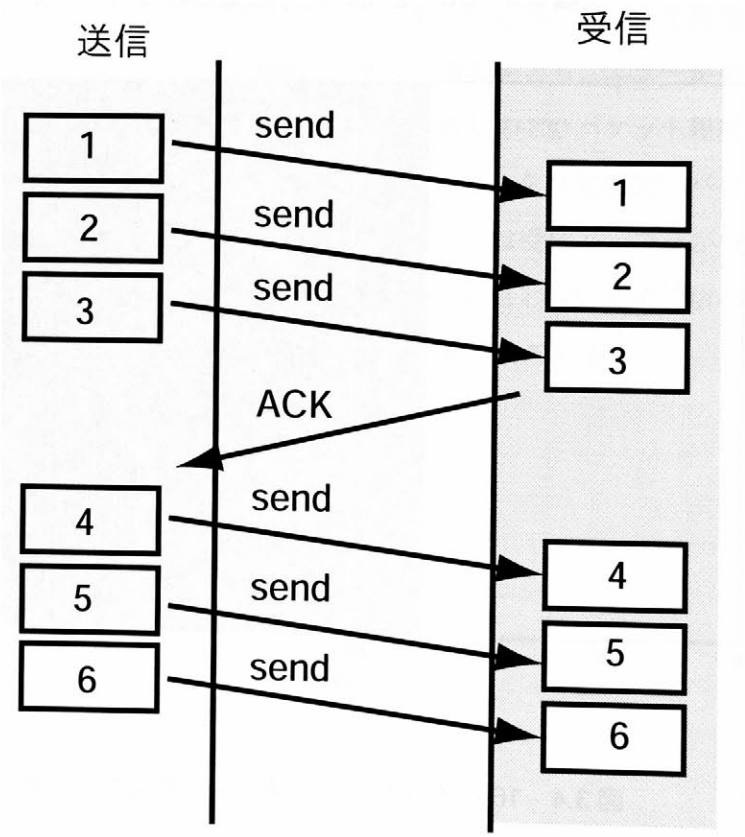


図3.5　フロー制御

パケット消失時の処理

次に，DMPにおいてパケット消失が発生した場合の処理について図3.6を用いて説明を行う．なお，例では，先ほど説明した連続送信数の制限によるフロー制御が行われているものとする．この例では，連続して送ったパケット1,2,3のうち3番目のパケットが何らかの障害により消失している．この場合，送信側でのタイムアウト（図3.6(1)），または，受信側での2番のパケットまでのACKパケットの送信(図3.6(2))のどちらかが発生しする．送信側でのタイムアウト発生時には，送信したパケット1,2,3が再び送信される．また，ACKパケットが送信された場合には，パケットが消失した3番のパケットからパケット3,4,5が送信される．このように，送信側と受信側のどちらでパケット消失の処理が発生したかで送信されるパケット数に違いが生じてしまう．しかしながら，DMPではACKパケット生成までの待ち時間が送信側のタイムアウトよりも十分小さく設定されているため，通常は図3.6(2)の処理が行われることになり，この例の場合であれば3番目のパケットだけが再送されることになる．

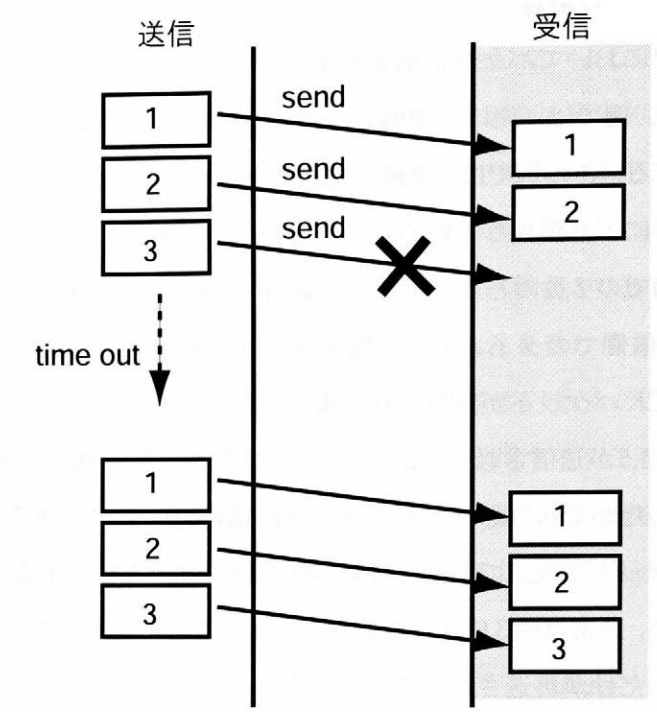


図 3.6　パケット再送制御

また，図中ではACKパケットは独立したパケットとしているが，実際のDSEの通信では双方向通信が頻繁に行われているため，多くの場合はACKのみのパケットは生成されず，他のデータパケットとパッキングされた形での送信となると予想される．したがって，このACK制御によるパケットの増加は抑えられる．

3.6 DSEへの組み込み

3.6.1 DMPのDSEへの組み込み

2章で説明したように，DSEは分散共有メモリ型並列計算機の機能をLAN接続された計算機クラスタで実現するためのUNIX上のソフトウェアである．DSEは，各機能モジュール毎に変更可能な設計となっており，DMPの組み込みに関しては図3.7で示すようにメッセージ交換モジュールとメッセージ解析モジュールの変更により組み込みを行った．

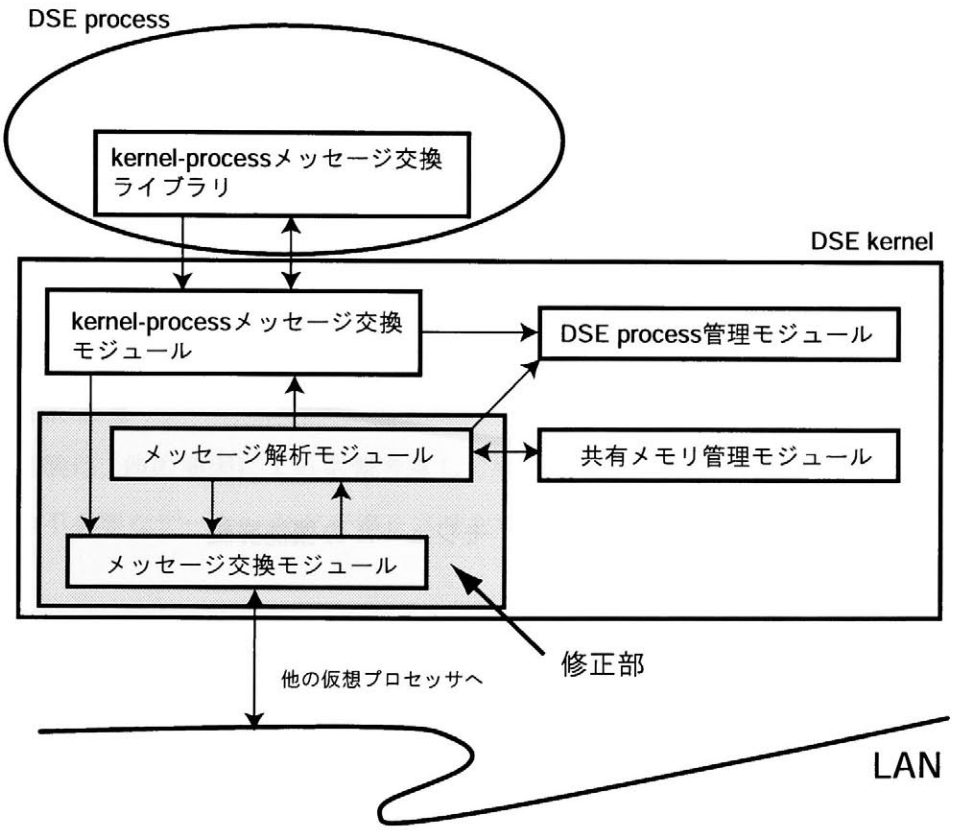


図3.7　DSEへのDMPの組み込み

実装したプロトコルはDSEのメッセージ交換モジュールに相当する機能をすべて含んでいる．具体的には図3.8のメッセージ交換モジュールの部分がDMPとなる．
このメッセージ解析モジュールに対する変更は，基本的にDMPの機能を効率よく呼び出せるようにするための，メッセージ交換モジュールとのインタフェースの変更である．また，この変更にともない，メッセージ解析モジュールのメッセージの解析・実行部についても若干の変更を行が必要であった．なお，これ以外のDSEのモジュールについては変更は行っていない．

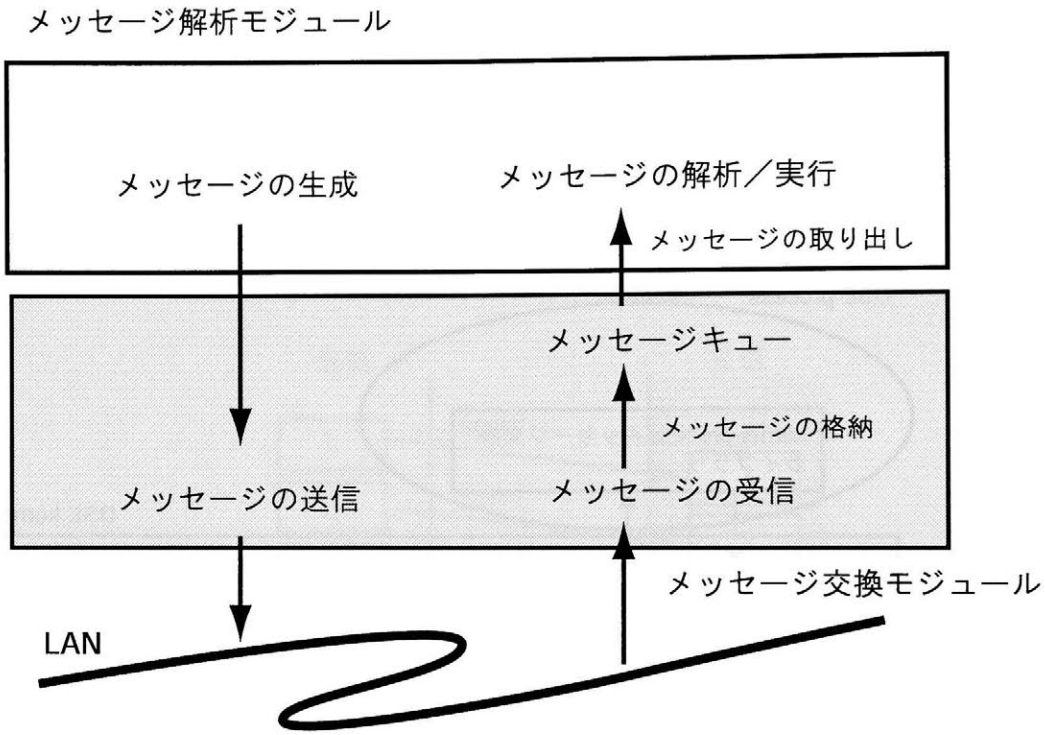


図3.8　モジュールの機能詳細

3.6.2 TCP版DSEとの実装上の比較

TCPを用いたDSE(以下，TCP版DSE)との実装上の比較を行った場合，DMPを用いたDSE(以下，DMP版DSE)ではTCP版DSEに比べてUNIXカーネルの呼び出し回数が少なくなる．

TCP版DSEの場合，送信されたメッセージはデータストリームとして連続したデータとして送られるため，メッセージ毎の切れ目を検出することができない．このため，TCP

版DSEでは，まずヘッダを読み出し，メッセージ長を調べた後にメッセージ本体を読み出すという処理が行われる．したがって，UNIXのシステムコールrecv()による読み出しが1メッセージの取り出しに対して，最低でも2回必要になる．

一方，DMP版DSEの場合，UDPで送信したデータは送信された単位で読み出すことができるため，TCP版DSEのようにヘッダを読み出してから本体を読み出す必要はない．メッセージがDMPのパケットサイズを超えるほど大きな場合については分割数分だけのrecv()システムコールの発行が必要となるが，1つのパケットで送信できるサイズであれば1回のrecv()システムコールの発行だけで済む．前述したように，DSEメッセージの多くはDMPの1パケットに収まるサイズであるため，DMP版DSEではほとんどのメッセージに対して1回のrecv()システムコールの発行だけで受信できる．

このように，DMP版DSEでは，UNIXのシステムコールの発行回数を削減することができ，これによりシステムコール発行によるオーバヘッドを削減効果による処理時間の短縮が期待できる．

以上，本章では実装したプロトコルの詳細について説明した．以降では，このプロトコルの評価のために行った実験の結果について説明する．

3.7 評価実験(1)

ここでは，DMPを用いて行った実験の結果について述べる．この実験では，DMP自体のパフォーマンスを評価するためTCPとの速度比較を行った．

3.7.1 実験環境

評価実験は，他の要因による影響考慮してネットワークのトラッフィクが少なく，計算機上の不必要なデーモンなどを停止させた状態で行った．表3.2に実験環境を示す．

表3.2　実験環境

計算機	SPARC station 20×3台
OS	SunOS 4.1.4
ネットワーク	イーサネット10Mbps

また，DMPのパラメータは表3.3のように設定した．

表3.3　DMPのパラメータ設定

パラメータ名	設定値
連続パケット送信数	10パケット
最大パケットサイズ	1024バイト

3.7.2 DMP単体パフォーマンス測定

実装したDMP単体の性能評価を行うために，転送するデータサイズを変更しながら，片方向への連続転送および双方向転送(ピンポン転送)に要する時間の2種類の時間を測定した．また，比較のためにTCPで同様の処理を行うプログラムを作成し，測定を行った．なお，本実験では，データサイズは16バイトから1024バイトまで変化させた．さらに，計算機の台数も，受信および送信を行う計算機の台数を１対１（１：１と表記）または１対２（１：２と表記）と変化させて測定を行った．図3.9は，この実験の概要を示したものである．図のように，TCPでは，1:2通信の場合に受信側には2つの計算機からのデータを受信するために2つのポートを必要とする．一方，DMPでは1:2の場合でも1つのポートしか必要としない．このように，一対多通信の場合には，DMPの方がポート数を必要とせず，OSのリソースを消費しない．

なお，TCP側の測定では，TCPの送信遅延が少なくなるようにPUSHオプションを設定している．このオプションを設定しない場合，TCPでは一定時間経過するか送信するデータが一定量を越えるまでデータ送信を遅らせるという性質がある．この制御は，通常の通信ではネットワークのトラッフィクを削減できるために有益な処理であるが，低遅延の通信が必要な場合には問題となる．そこで，本実験では，レイテンシの削減を目的にこのオプションを設定した．

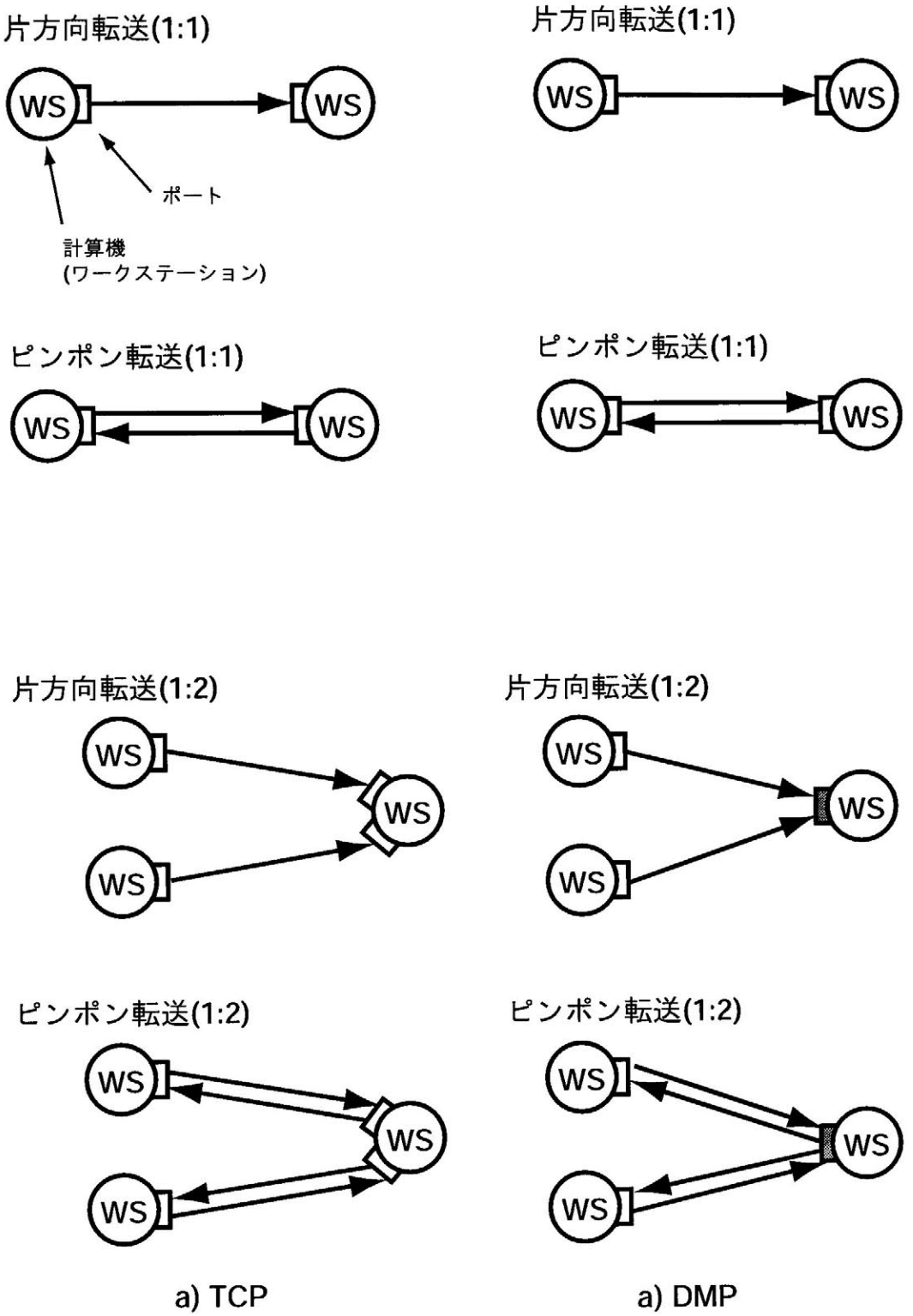

片方向転送(1:1)
WS
WS
ポート
計算機
(ワークステーション)
片方向転送(1:1)
WS
WS
ピンポン転送(1:1)
WS
WS
ピンポン転送(1:1)
WS
WS
片方向転送(1:2)
WS
WS
WS
片方向転送(1:2)
WS
WS
WS
ピンポン転送(1:2)
WS
WS
WS
ピンポン転送(1:2)
WS
WS
WS
a) TCP
a) DMP

図 3.9　転送実験の概要

(1)片方向への連続転送実験

図3.10は，片方向へ連続転送を行った結果のグラフである．グラフは，横軸に一度に転送したデータサイズを，縦軸にデータあたりの処理時間を示している．グラフ中のTCP(1:1)，TCP(1:2)はTCPを用いた1対1または1対2通信を，DMP(1:1)，DMP(1:2)はDMPを用いた1対1または1対2通信を表している．また，処理時間は，データ転送を10,000回繰り返す時間を測定し，これを転送回数で割ることにより求めた．このような測定を行った理由は，1データの転送に要する時間が測定に用いたタイマ制度に対して小さいため，1データあたりの転送を測定した場合には，タイマ誤差の影響が大きいからである．また，1:2の場合は，送信側が10,000回のデータ転送を行った処理時間を測定し，これを受信回数の20,000回で割ることにより処理時間を求めている．したがって，1:2の結果は，受信側で20,000個のデータの受信に要した時間の1個あたりの平均時間となる．

図3.10のグラフより，転送データサイズが小さい場合にはTCPの方が処理時間が短く，データサイズが256バイトを越えたあたりから両者の差は小さくなり，1024バイトでは1:1,1:2のどちらの場合についてもDMPの処理の方が速くなっていることが分かる．なお，1:2の時には512バイトからTCPよりも処理時間が短くなっている．

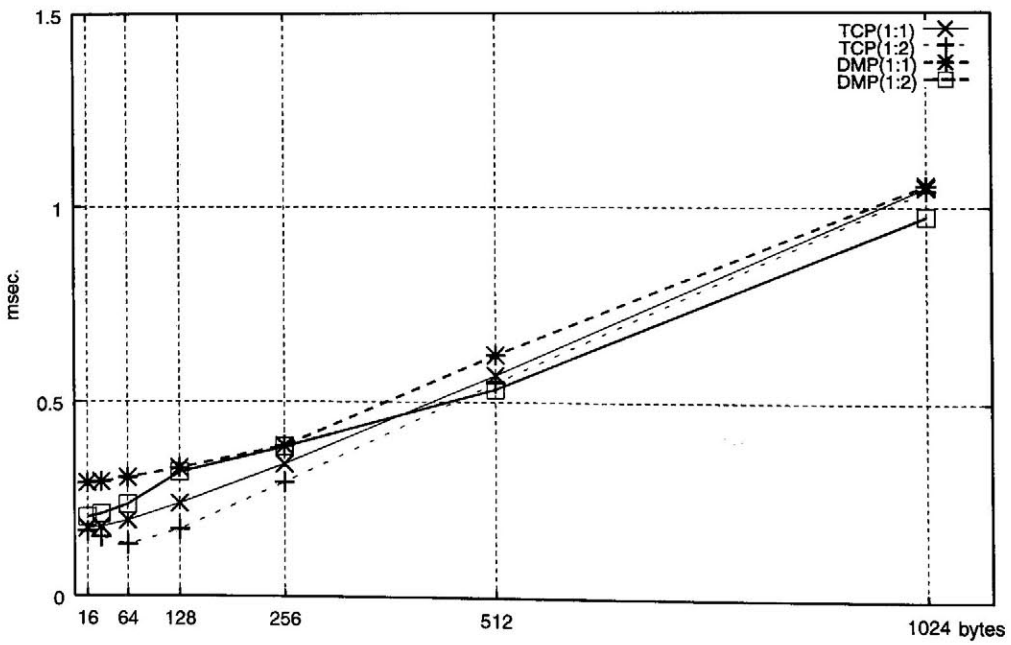


図 3.10　片方向への連続転送の結果

小さなデータ転送でTCPの方が高速な理由は，データをまとめて転送を行うというTCPの処理によるものである．実験では，このような処理を行わないようにするため，TCPのPUSHオプションを付加しておいた．しかし，SunOSに実装されているTCPでは，PUSHオプションが付加されていてもこのデータを一括送信する処理を行う．なお，これについては，Solarisのsnoopコマンドを用いて，ネットワークを流れるパケットをモニタリングすることにより確認した．しかしながら，snoopコマンドの性質上，すべてのパケットをモニタリングできるとはかぎらず，どの程度の割合で一括した転送が行われているのかを調べることができなかった．

以上の理由により，すべてのデータを個別に送るように設計したDMPに比べ，TCPでは小さなデータで約69%も処理時間が短くなっている．一方，TCPでのパケットをまとめる処理が行われない大きなデータサイズでは，立場が逆転しTCPの方が約3%遅くなった．

（2）ピンポン転送実験

DSEの通信の多くは，「要求」とそれに対する「返答」によりなることが，これまでのDSEのアプリケーションの通信パターンの解析から明らかになっている．例えば，共有メモリの読み出し(Read)などは，データの読み出し要求の送信と，要求に対するデータの返送の双方向の通信により実現される．このことを考えると，DSE向きのプロトコルとしては，片方向の通信速度よりも，双方向のピンポン転送の能力の方がより重要であると考えられる．

そこで，ここでは，TCPとDMPの双方でピンポン転送を行い，処理時間の測定を行った．なお，ピンポン転送では，データが送信されなければ，受信側からの応答のデータは戻されないため，片方向通信の場合のようにTCPでデータがまとめられてしまうというようなことはない．このため，片方向に比べてより同一の条件で2つのプロトコルの速度比較を行うことができる．

図3.11は，片方向通信と同様にデータサイズを変化させながら測定を行った結果のグラフである．図より，ほとんどの場合において，DMPに比べTCPの方が若干処理時間が短いことが分かる．これは，DMPでは，受け取ったデータをメッセージとしてメッセージキューにキューイングするという動作を行っているため，この処理分だけ処理時間が余分にかかるからである．この処理は，DSEのメッセージ交換モジュールが行っている処理に相当し，DSEに組み込んだ場合は，これまでTCP版DSEが行っていたこの部分の

処理がなくなるため，その分だけ処理時間が短くできる．なお，この部分は，これまでの研究からTCPの処理時間の5%～10%であることが分かっている[53][54]．

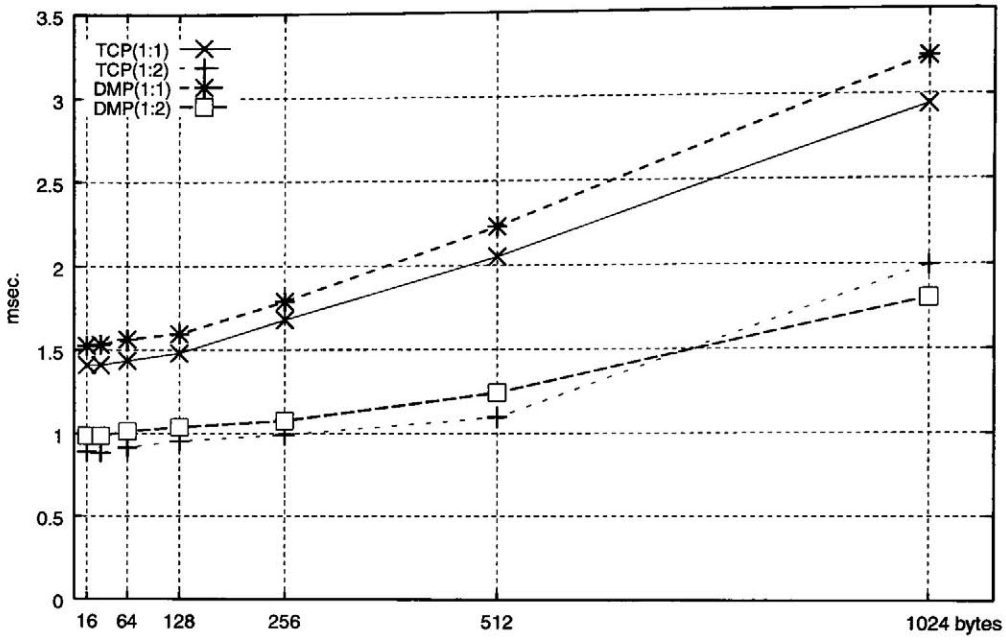


図 3.11　ピンポン転送の結果

したがって，TCPとDMPの速度差がこの範囲であれば，DSEに組み込んだ場合にはほとんど変わらないパフォーマンスを実現できると考えられる．そこで，図3.11の結果から，両者の速度比を求めたのが，図3.12である．図より，両者の差はほとんどの場合10%前後であることが分かる．したがって，ピンポン転送については，DSEに組み込んだ場合にはTCPとほとんど変わらないパフォーマンスを実現できることが推測される．

結果より，ピンポン転送についてはTCPとDMPとではTCPの方が若干処理速度が高速であることが分かった．しかし，DSEに組み込むことを考えた場合は，とほとんど変わらない性能を実現できると予想できる．

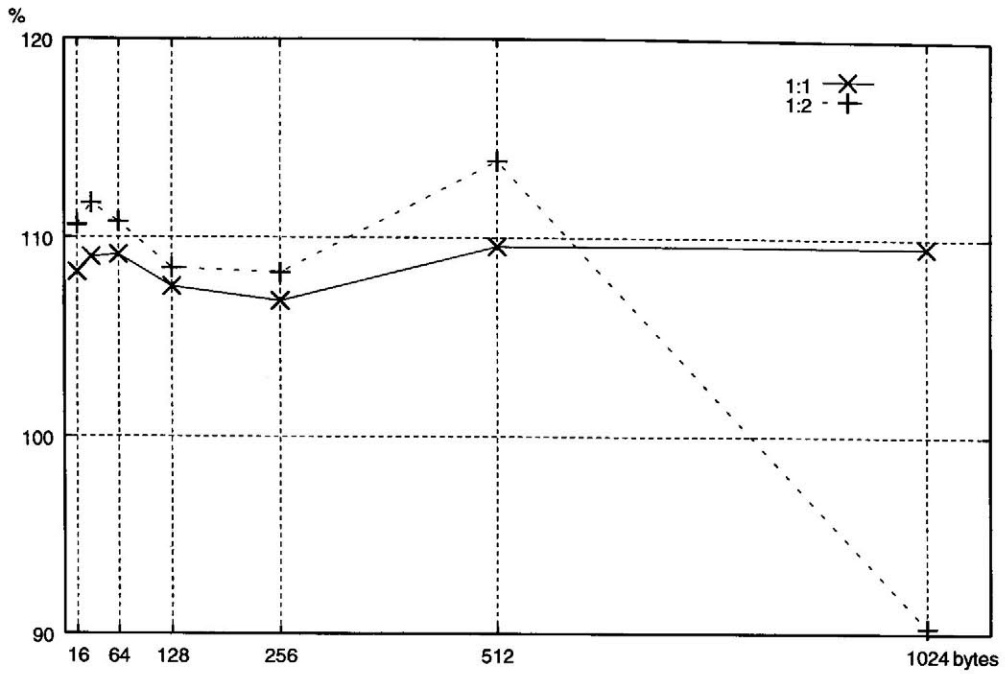


図 3.12　ピンポン転送における DMP と DSE の速度比（DMP ÷ TCP）

以上，片方向への連続転送および，双方向でのピンポン転送を行う評価用プログラムを用いて，DMPの基本性能について評価を行った．この結果，連続したデータ転送に関しては，小さなデータサイズではTCPが，大きなデータサイズではDMPの方が処理が高速にできることが分かった．また，ピンポン転送の結果より，全体的にTCPの方が高速であることが分かった．ただし，DSEへの組み込みを考慮した場合は，連続転送よりもピンポン転送の結果の方が重要であること，およびDMPがメッセージ交換モジュールの機能を含むことを考慮すると両者の差はわずかであると考えられる．

これらの結果からは，TCPとDMPでは，性能差はほとんど無いように感じられる．しかし，大規模な並列処理を行う場合，TCPではポートの制限により直接通信が行えず，他の計算機を経由した通信が必要であることを考えると，ポートを１つしか使用せず，制限なく複数の計算機と直接通信可能なDMPの方が有利になる．

そこで，以下のDMPをDSEへ実装しての，直接接続の場合だけでなく，TCP版DSEで仮想ネットワークを構築した場合とのDMPとの比較を行うことで，直接接続できなかった場合の影響についても評価している．

3.8 評価実験(2)

ここでは，DMPを組み込んだDSEを用いて行った実験の結果について述べる．

3.8.1 実験環境

評価実験は，他の要因による影響考慮してネットワークのトラッフィクが少なく，計算機上の不必要なデーモンなどを停止させた状態で行った．表3.3に実験環境を示す．

表3.3　実験環境

計算機	PC/AT互換機×8台
OS	FreeBSD 3.1
ネットワーク	イーサネット10Mbps

また，DMPのパラメータは3.6節の実験と同様に設定した．

3.7.3 DSEでの並列アプリケーションの実行

DMPを組み込んだDSEを用いた実験では，メッセージのパターンが特徴的な2つの並列アプリケーションを利用して測定を行った．実験に利用した並列アプリケーションは，１つはコンピュータ同士で対戦を行うオセロゲームであり，もう１つは偏微分方程式の解を求めるアプリケーション(SOR)である．

オセロゲームは，各仮想プロセッサからのアクセスが１つの仮想プロセッサに集中する，多数のプロセス生成およびバリア同期操作を含むアプリケーションである．また，SORは，基本的には隣接する仮想プロセッサ同士(仮想プロセッサNはN-1とN+1番目の仮想プロセッサと通信するという意味)の通信しか行わないプログラムであり，メッセージの送受信は隣接する仮想プロセッサ間に局所化される．簡単にいえば，オセロゲームは一極集中型の通信，SORは均等に分散された通信を行うアプリケーションであるといえる．

なお，この実験では，仮想プロセッサの数を1～8に変化させて実験を行った．また，TCP版DSEについては，接続が制限された場合を想定して，仮想的なネットワークとし

て，完全網，鎖網，木状網，リング網，トーラス網による接続を行った．

この実験の結果が，図3.13および図3.14である．なお，図3.13はオセロゲームの実行結果，図3.14はSORの実行結果であり，グラフの横軸が実行回数を，縦軸が処理時間を示している．このように，この実験で実行時間の平均値などではなく，実行毎の時間を横軸として結果を示している．

実行結果

図3.13の結果より，実装したプロトコルはTCP版の完全網での接続には劣るものの，それ以外の接続に比べて処理時間が短いことが分かった．結果を接続網順に並べると，完全網，トーラス網，リング網，木状網，鎖網となり，これはアクセスが集中する仮想プロセッサ0と他の仮想プロセッサとの距離平均の順に一致する．オセロゲームにおいて，DMPが完全網の場合に比べてパフォーマンスが出なかった理由としては以下が考えられる．

DMPでは，1つのポートですべての仮想プロセッサからのメッセージを受け取るため，アクセス集中が発生すると処理性能が低下しやすい．また，これに加えて，DMPはユーザレベルプロセスとして実装されているため，アクセス集中が発生するとDMP処理のためにUNIXシステム中の優先順位が下げられてしまう．これらの問題は，DMPがユーザレベルプロセスとして実装されていることに起因する．

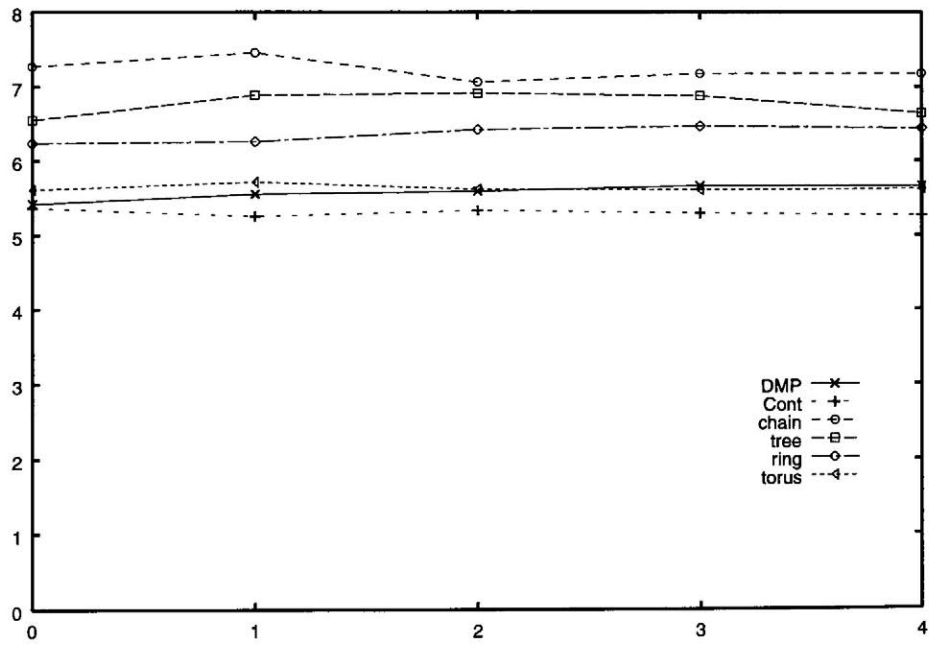


図 3.13　オセロゲームの実行結果

次に図3.14のSORの結果について述べる．SORの結果を見ると，完全網，DMP，リング網，トーラス網，木状網，鎖網の順で実行時間が短いことがわかる．リング網やトーラス網の実行性能がよいのは，SORの通信が基本的には隣接した仮想プロセッサとしか行われないためで，これらの結合網では通信を頻繁に行う隣接したプロセッサへの距離平均が短いからである．特に，リング網は隣接する仮想プロセッサとの距離だけをみれば完全網と同じである．にもかかわらず，完全網に比べてリング網の実行時間が長い理由は，中間結果表示などのメッセージはユーザコンソールとなる仮想プロセッサに送られるため，この通信が隣接した仮想プロセッサ外への通信となるためである．なお，この結果でもDMPは完全網接続に及ばなかった．この理由は，やはりユーザレベルプロセスとして実装されているDMPでは，処理量の増加に対してUNIXのプロセス優先度が低下するためである．これは，UNIXのプロセススケジューラが処理量の大きい（処理の重い）プロセスの優先度を低下させる処理が原因である．これは，実際の測定でもTCP版DSEとDMP版DSEのDSEカーネルへのCPU割り当て時間は3～5倍であることにより確認している．

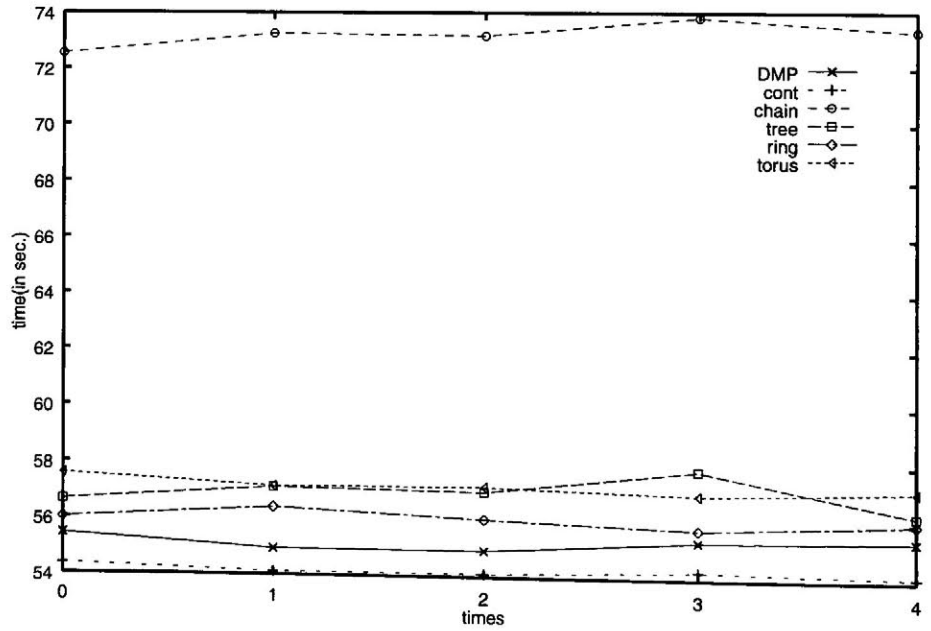


図3.14　SORの実行結果

以上の結果より，DMP版DSEはTCP版DSEにおいて完全網接続した場合と比べて若干速度が低下してしまうことが分かった．しかしながら，ポート数を制限した他の接続網に比べた場合には，DMP版DSEの方が高速であることが確認できた．なお，完全網接続は，OSの制限を越えるような大規模なクラスタ（通常のUNIXの場合は32プロセッサ）では不可能であるため，このような場合には他の仮想的な接続網を利用せざるおえない．また，規模が大きくなると，プロセッサ間の距離が長くなるため，経由するプロセッサの数が増え，これにより通信速度がさらに低下すると考えられる．このような，大規模クラスタを考えた場合，プロセッサ数によらず直接通信を実現するプロトコルが速度面で有利になることは明らかである．

3.9 まとめ

本章では，DSEで大規模実験を行う場合に問題となっていたTCP接続制限の問題に対する解決策としてUDPを用いた独自のプロトコルを実装することを提案し，設計および実装を行った結果について述べた．このプロトコルはユーザレベルのプロセスとして実装されており，UDPを用いることで1つのポートで複数の仮想プロセッサからのメッセージの受信を可能にする．これにより，従来のDSEで問題になっていた接続制限の問題を，UNIXシステムに手を加えることなく環境を構築できるというDSEの利点を損なうことなく解決する．このようにDMPは，ポートというシステムリソースの消費を大幅に削減する．

また，実験の結果から，実装したプロトコルはTCP版DSEで完全網接続した場合には劣るものの，(ポート数を制限した)それ以外の接続を行うのに比べれば十分高速であることが分かった．なお，完全網接続は，DSEで大規模なクラスタを構成しようとした場合にはOSリソースの問題から不可能となる．この場合，TCP版DSEでは他の仮想ネットワークによる接続を行わなければならない．また，大規模なクラスタになればなるほど，トーラス網などの仮想ネットワークでは直接接続されていない計算機間の距離が長くなり，通信速度は実験結果に比べさらに低下する．したがって，大規模なクラスタを構成する場合に，実装したプロトコルはパフォーマンス的により有利になると考えられる．加えて，利用ポート数の削減は，今後展開が予定のポートの自動割り当ての仕組みや，複数のユーザがDSEを同時利用する場合に発生していたポートの衝突などの問題を解消する可能性があり，ユーザの便利性を考えるとさらに有用であると考えられる．

第4章

オブジェクト指向プログラミング環境 (PPElib)

4.1 概論

これまで，DSEのプログラミングは，DSElibとよぶ共有メモリのアクセスやプロセスの起動，同期処理などを含んだC言語用のライブラリを用いて記述していた．しかし，このライブラリは，DSEの提供する基本的な機能を呼び出すライブラリであり，このようなライブラリを用いての並列アプリケーションのプログラミングでは，ユーザの並列処理に対する熟練度(スキル)を必要とし，(逐次)プログラミングはできるが並列処理について知識が浅いユーザが同期処理などを適切に挿入し，正しく動作するプログラムを記述するのは難しかった．

そこで，本研究では，これまでに記述されたDSE向け並列アプリケーションから，多くのアプリケーションが共通して持っているプログラミングスタイルを抽出し，これを基にユーザのプログラムのスタイルを制限することで，ユーザの記述性を向上させる試みを行った．

新たに作成したライブラリは，抽出したプログラミングのスタイルの共通点をプログラミングモデルとしてユーザに提供する．具体的には，オブジェクト指向言語C++の特徴であるクラス定義の機能により，基本となるモデルを提供し，ユーザはこの基本となるモデルクラスを継承して，必要な部分だけ記述する．このように，クラス継承を用いてプログラミングさせることで，ユーザが提供するモデルからはずれたプログラミングを行うことを難しくし，結果としてユーザにプログラミングモデルに沿ったプログラミングを行わせることができる．

このような，C++言語のクラス定義と継承という特徴を利用して，ユーザにプログラミングのための基本構造を提供するものとして，WindowsやMacintoshなどでグラフィカルユーザインタフェースを作成するために容易された，MFC(Microsoft Foundation Calss Library)やPowerPlantなどのフレームワークと呼ばれるものがある．これらフレームワークでは，ウィンドウやダイアログ，ボタンといったGUIの基本となるパーツがクラスとして用意されている．ユーザは，これらのクラスを継承して，独自部分だけを記述する(例えば，ボタンがクリックされた場合の動作など)ことでプログラミングを行う．このようにすることによって，これらのフレームワークはユーザがフレームワークから外れたプログラムを記述することを防ぎ，また，GUIの作成経験の少ないユーザでも，容易にプログラミング可能なプログラミング環境を提供する．

本研究の目的の1つは，ユーザに対して並列アプリケーション記述のためのフレームワークとなるクラスライブラリを提供することであり，これによりユーザの記述性を向上させることである．

また，本研究の他の目的として，シームレスな並列処理環境[44][45][46]の提供がある．近年，ネットワーク形態および計算機の構成が非常に多岐にわたるようになってきた．例えば，高性能なPCや2または4プロセッサを搭載したSMPなどの低価格化，および急速な性能向上により，ネットワークに接続されたPCクラスタの場合，それぞれの計算機の性能が異なることも珍しくなくなっている．このような環境で並列アプリケーションを実行する場合，プログラマは個々の計算機がどのくらいのCPU性能で何プロセッサ搭載しているかなどの性能をあらかじめ把握しておくか，何らかの負荷分散機能を並列アプリケーションに組み込んでおかなければ効率のよい実行は望めない．この性能の差異に加えマルチユーザ環境で計算機の負荷変動が発生する場合は，これらを考慮したプログラミングはさらに難しくなる．このように，既存の計算機クラスタを利用した並列処理環境で効率のよい並列アプリケーションを作成することは，以前にもまして難しくなってきている．

本研究は，計算機の性能差や，ノード間の複数レベルの並列性，および動的に変化するシステムの状態をクラスライブラリうまく取り扱い，ユーザのプログラミングの負担を軽減すると共に，高い並列実行性能を得ようという試みである．

4.2 オブジェクト指向プログラミングライブラリ(PPElib)

本研究で設計したオブジェクト指向プログラミングライブラリ(Parallel Programing Environment and Library, PPElib)は，アプリケーション，タスク，軽量プロセス(LWP)の3層構造からなるプログラミングモデルをユーザに提供する．本節では，PPElibの提供するプログラミングモデルと，そのインタフェースについて説明する．

4.2.1 プログラミングモデル

PPElibの提供するプログラミングモデルでは，大きくアプリケーションとタスクの2つから構成される．なお，この2つはC++言語のクラスとして提供される．また，タスクは内部にLWPを含み，プログラムはアプリケーション，タスク，LWPで3層構造をとる．

・アプリケーション(クラス)

　プログラム全体の流れを管理する．並列実行のための環境の初期化や，タスクの実行制御を行う．

・タスク(クラス)

　並列実行される部分プログラムを管理する．タスクでは，各仮想プロセッサ上に割り当てられ実行されるLWPの仮想プロセッサへの割り当て，および割り当てたLWPの処理終了時の次LWPの自動割り当てや，タスク終了時の同期処理を行う．

図4.1は，ライブラリが提供するプログラムのモデルである．図4.1のように，アプリケーションでは，タスクの実行順序を記述し，実行時には，アプリケーションクラスで指定された順番でタスクが呼び出され実行される．また，タスクは，内部にLWPを持ち，これがプロセッサに動的に割り当てられていく．なお，このモデルでは，同一タスク中に含まれるLWP間には順序依存がないことが前提である．しがたって，同一タスク中に含まれるLWPは，どのような順番で実行されても正しく実行できる処理でなければならない．この制限により，タスククラス内のLWPアロケータで，仮想プロセッサへのLWPの割り当てを自由に設定できるようになり，環境に合わせたLWP割り当て処理が可能になる．

ライブラリでは，アプリケーションとタスクはクラスとして提供されているため，

ユーザはこれらのクラスを雛形としてプログラミングを行う．具体的には，アプリケーションクラスおよびタスククラスをそれぞれ継承して，ユーザ独自のアプリケーションクラスおよびタスククラスを作成する．

なお，本クラスライブラリでは，プロセス割り当ての部分についてもユーザのメソッド上書きにより，ユーザにより指定可能である．これを利用すれば，例えば，特定の仮想プロセッサにあるLWPを割り付けるといった操作も可能である．

このように，継承を用いたプログラミングでは，ユーザのアプリケーション作成の枠組みを提供するだけでなく，必要であればLWP割り当てなどの細かな部分まで変更できる柔軟性を提供する．

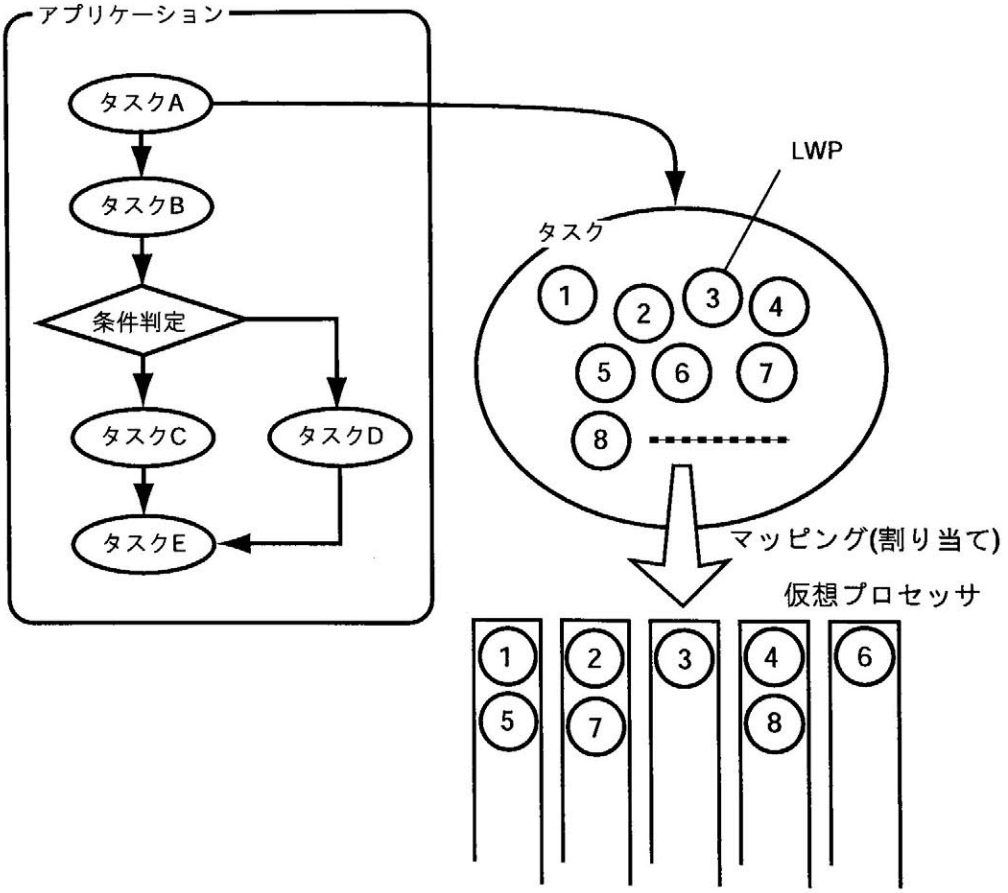


図 4.1　プログラムモデル

本節の最後に，クラスライブラリの提供するクラス一覧を示す（表4.1）．図のように，PPElibは，PPE_Application, PPE_Task, PPE_Memory, PPE_FetchAndAddクラスの大きく

4つのクラスからなる．また，PPE_TaskクラスのProcessメソッドはLWPの処理を記述するためのメソッドであり，派生したクラスでこのメソッドを上書きすることでLWP処理を記述する．

表4.1　クラスライブラリのクラス一覧

クラス名	メソッド名	機能
PPE_Application	Run	並列アプリケーションを実行する
	main	並列アプリケーションのメイン処理を記述する
	RegisterProcess	タスクIDを登録する
PPE_Task	Run	タスクを実行する
	BeforeTask	タスク実行前に実行される
	AfterTask	タスク実行後に実行される
	Process	LWPの処理を記述する
	BeforeProcess	各プロセッサでLWP実行前に一度だけ実行される
	AfterProcess	各プロセッサでLWP実行後に一度だけ実行される
	SetNumOfProcess	LWP数をセットする
	GetNumOfProcess	LWP数を取得する
	DoSerial	排他実行を実現する
PPE_Memory	Read	共有メモリの読み込みを行う
	Write	共有メモリの書き出しを行う
PPE_FetchAndAdd	Init	Fetch&Add用変数の初期化
	FetchAndAdd	Fetch&Addの実行

4.2.2 プログラミングインタフェース

本節では，実際のプログラミングの例を示しながらPPElibのプログラミングスタイルとPPElibを用いたアプリケーション作成手順について具体的に説明する．

4.2.2.1 アプリケーション

PPElibを使ってアプリケーションを作成する場合には，まず，ライブラリのアプリケーションクラス(PPE_Applicationクラス)を継承したクラスを作成する．図4.2は，ユーザのアプリケーション(myAppクラス)の例である．図中のRegisterProcessメソッドは，プロセスマネージャ(PPE_ProcessManager)にタスクIDとタスク生成メソッドの組を登録するメソッドである．このライブラリでは，並列アプリケーション内で利用するすべてのタスクをプロセスマネージャに登録する必要があり，この登録をアプリケーションクラ

スのRegisterProcessメソッドで行う．なお，タスクIDは，アプリケーション内で一意なIDでなければならず，異なるタスクに同じIDをつけてはならない．

また，mainメソッドは，並列アプリケーションのメイン関数となる部分であり，ここには，タスクの生成や条件分岐などを記述する．このmainメソッドの実行は，１つのプロセッサ(ホストプロセッサ)で実行される．したがって，このメソッドの実行は逐次処理になり，この部分には処理の流れを記述することになる．

```
class myApp : public PPE_Application {
public:
  void RegisterProcess();
  void main();
};

void myApp::RegisterProcess()
{
  PPE_ProcessManager::Register(myTaskID, myTask::Create);
}

void myApp::main()
{
  myTask theTask;
 :
 :
  theTask.Run();
 :
 :
}
```

図 4.2　myAppクラスの定義例

図4.3に，プログラムのメイン関数(main())を示す．このように，メイン関数ではアプリケーションオブジェクト(theApp)を生成し，Runメソッドを呼び出すだけとなる．Runメソッドが呼び出されると，アプリケーションオブジェクトは並列処理環境(DSE,pthreadなど)の初期化を行い，その後mainメソッドの呼び出しを行う．

```
main()
{
  myApp theApp;

  theApp.Run();
}
```

図 4.3　メイン関数の例

以上，アプリケーションクラスには逐次処理部を記述する．なお，並列処理部は以下で説明するタスククラスを用いて作成する．このように，並列処理部と逐次処理部を２つのクラスに分けたことにより，プログラム全体の流れが見やすくなる．

4.2.2.2 タスク

タスククラスでは，並列実行可能な多数のLWPから構成される処理を簡単に記述できるように設計した．したがって，処理をこのような構成にマッピングすれば，簡単にプログラミングが可能となる．

図4.4は，タスククラス(PPE_Taskクラス)を継承して，ユーザのタスク(myTaskクラス)を生成した例である．図のように，作成したタスククラスでは，最低限３つのメソッドを定義する必要がある．１つめは，コンストラクタで，ここでは，タスクIDの設定を行う．タスクIDは前述したように一意なIDであり，ライブラリではこれを用いてタスクを識別する．２つめは，Createメソッドで，このメソッドはこのタスクのオブジェクト(myTaskオブジェクト)を生成するためのメソッドである．このメソッドは，他プロセッサでタスクオブジェクトのコピーを作成するためのメソッドであり，ライブラリ内の制御のために利用されるメソッドである．このメソッドの処理では，例えば，図4.4のようにnewメソッドにより自分自身を作成する．

```
class myTask : public PPE_Task {
public:
  myTask() : PPE_Task() { RegID = myTaskID; }
  static myTask* Create() { return(new myTask); }
  void Process();
};
```

図4.4　タスククラスの定義例

3つめはProcessメソッドであり，このメソッドには仮想プロセッサ上に割り当てられ実行されるLWPの処理を記述する．Processメソッドとしては例えば図4.5のようになる．Processメソッド中のGetNumOfProcessメソッドおよびデータメンバmCurrentProcessは，それぞれLWP数を取得するメソッドと現在のLWPのプロセスIDを保持するデータメンバである．タスクに含まれるLWPには0からはじまるプロセスIDが割り当てられ

ており，タスククラスでは自動的にLWPにプロセスIDを付加し，Processメソッドを呼び出す．Processメソッドでは，このプロセスIDをデータメンバmCurrentProcessにより調べ，プロセスIDに一致した処理を行う．図では，LWP数とプロセスIDを用いて処理を決定し，処理を実行している．このように，処理をLWP数で割って分割しておけば，プロセスIDにより処理する部分が特定できる．

```
void myTask::Process()
{
  int max = GetNumOfProcess();
  int n = mCurrentProcess;

  // nとmaxを用いて処理を決定し，実行する
}
```

図 4.5　Processメソッドの定義例

また，タスククラスには，タスクの実行前（並列動作前）と後（同期後）の処理をBeforeTaskおよびAfterTaskメソッドとして記述することができる．例えば，BeforeTaskメソッドで，LWP数の設定や各種パラメータの初期化などを行い，AfterTaskメソッドで実行後の結果の収集などを行うことができる．加えて，タスククラスには，各プロセッサでLWPが実行される前に一度だけ呼び出されるBeforeProcessおよび，プロセッサ上でのLWP実行がすべて終了した時点で呼び出されるAfterProcessメソッドが用意されている．わかりやすくいえば，これらのメソッドは，各プロセッサでタスク処理（LWP実行）が行われる前と終了時に呼び出されるメソッドである．したがって，このメソッドで確保されたメモリなどは，同じプロセッサで実行されるすべてのLWPから共通してアクセスすることができる．これらのメソッドを利用すれば，各プロセッサがLWPを実行する前に行う初期化（ローカルメモリの確保や，共有メモリからローカルメモリへのデータコピーなど）や，終了処理（確保したメモリの解放）などを行うことができる．

以上のように，タスククラスを利用したプログラミングでは，並列実行可能な部分についてさらに独立実行可能な処理に分け，この独立な処理をLWPとしてナンバリングしておく．これができれば，タスククラスの機能により，各プロセッサへの動的なLWPの割り当てと実行や，各プロセッサでの処理が完了した時の同期処理などは自動的に行わ

れるため，ユーザがこれらの処理を記述する必要はない．このようにプログラムを記述すればタスククラス内ルーチンで同期処理が自動的に行われるため，プロセス制御や同期制御のミスによるバグの発生を抑えることができ，プログラミングが容易になる．

4.2.2.3 ユーザによるタスク内での排他制御

前述したように，タスク内に含まれるLWPはタスククラスにより自動的に割り当てられ実行されるため，依存関係のない独立した処理でなければならない．これでは，LWP間で共有変数を持つプログラムは，このライブラリではプログラミングすることができない．これを解決するために，タスククラスにDoSerialメソッドを用意した．DoSerialメソッドは，引数として指定されたメソッド（関数）の実行は，ただ1つのLWPだけであることを保証するメソッドである．このメソッドを経由して呼び出されたメソッド（関数）で共有変数へのアクセスを行えば，共有変数へアクセスしているLWPが唯一1つであることが保証できる．PPElibでは，このメソッドを提供することでLWP間の排他制御を実現する．

4.2.2.4 共有メモリ

PPElibでは，基本的にRead/Writeメソッドによる共有メモリのアクセスを提供している．この方式では，ユーザはReadおよびWriteメソッドを用いて共有メモリの内容をローカルメモリにコピーするか，またはローカルメモリの内容を共有メモリにコピーする．

図4.6は，PPElibの共有メモリクラスを利用した共有メモリアクセスの例である．共有メモリクラスでは，基本的には共有メモリ全体にアクセスできるように図4.6 2行目のようなアクセスを行う．また，図4.6 4行目のように，共有メモリの一部分にアクセスすることも可能である．この実装では，共有メモリはプログラム全体から参照可能なバッファまたはファイルのようなイメージとなる．

```
1: // (1) int型で共有メモリ全体にアクセス
2: PPE_Memory<int> theMemA;
3: // (2) 共有メモリのアドレス100(offset)から1024個のint型データにアクセス
4: PPE_Memory<int> theMemB(100, 1024);
5: // 読み出し，書き込み
6: theMemA.Read(100, sizeof(int) * 10, a);
7: theMemA.Write(100, sizeof(int) * 10, a);
```

図 4.6　共有メモリクラスの利用例

4.3 他環境(HPC++Lib)との比較

本節では，PPElibの記述性の向上について説明するために，同じく継承を利用して並列プログラミングを行うHPC++Libとの比較を行う．２章で説明したように，HPC++Libはスレッドなどがクラスとして用意されており，これを継承したプログラミングを行う．ここではHPC++Libのサンプルプログラム[28]をPPELibで記述し直すことで，両者で同じアプリケーションを用意し，記述に対する比較を行った．つまり，比較したプログラムはPPElibに有利な設計ではなく，HPC++Lib向けに記述されたものである．

4.3.1 基本プログラム（１）

１つめのプログラムはHPC++Libのサンプルプログラム Helloである．このプログラムは，HPCXX_Threadクラスの基本的な利用方法を説明するシンプルなプログラムである．このプログラムでは，スレッドを生成して，"hello" という文字列を表示し，終了する．図4.7はHPC++Libのプログラム，図4.8はこれをPPElibで記述し直したプログラムである．図4.7 16行目のsleep(5)は，スレッドが生成されて起動後，終了するまで適当な時間待つためのウエイトである．図4.7のようにHPC++Libのプログラムでは，HPCxx_Threadクラスを継承することにより，スレッド(MyThread)を作成する．このプログラミングスタイルは，PPElibのプログラミングスタイルと非常に似ている．しかしながら，HPC++Libではスレッドの生成はユーザにより行う必要があるが，PPElibではスレッドに相当するLWPの生成はタスククラスにより自動的に行われるため，スレッド数が多くなった場合はPPElibの方がスレッド管理が楽になりプログラミングが行いやすい．

この例では，スレッド(LWP)を１つしか生成しないこと，およびPPElibではアプリケーションもクラス継承によりプログラミングしなければならないことなどがり，コードサイズはPPElibを用いて記述した方が長くなる．以上のように，生成するスレッド数が少

ないプログラムではPPElibの方がプログラミングに必要なコード量も多くなる. これは, PPElib が, 多くの LWP を含んだプログラムの記述を前提にしているためである.

```
 1: class MyThread : public HPCxx_Thread {
 2:    char *x;
 3:                                        HPCxx_Threadを継承
 4:    public:
 5:      MyThread(char *y): x(y),HPCxx_Thread(){}
 6:       void run() { cout << x << endl << flush; }
 7: };
 8:
 9: int main(int argc, char **argv)
10: {
11:    HPCxx_Group *g;                          スレッド生成
12:    hpcxx_init(argc, argv, g);
13:
14:    MyThread *t1 = new MyThread("hello\n");
15:    t1->start();
16:    sleep(5);    <-------------------- ウェイト処理
17:
18:    cout << "DONE" << endl;
19:    return hpcxx_exit(g);
20: }
```

図 4.7　Hello.cc(HPC++Lib による記述)

```
 1: #define dID_myTask 1000                    タスククラスの定義
 2: class myTask: public PPE_Task {
 3: public:
 4:   myTask() : PPE_Task() { RegID = dID_myTask; }
 5:   static myTask* Create() { return(new myTask); }
 6:   void Process() {cout << "hello\n" << flush; }
 7: };
 8:
 9: class myApp : public PPE_Application {
10: public:
11:   void main();
12:   void RegisterProcess();
13: };
14:                                  アプリケーションクラスの定義
15: void myApp::RegisterProcess()
16: {
17:   PPE_ProcessManager::Register(dID_myTask,
    (PPE_RegFunction)myTask::Create);
18: }
19:                                  アプリケーションクラスのmainメソッド
20: void myApp::main()
21: {
22:   myTask theTask;
23:   theTask.SetNumOfProcess(1);
24:   theTask.Run();
25:   cout << "DONE" << endl;
26: }
27:
28: main()
29: {
30:   myApp theApp;
31:   theApp.Run();
32: }
```

図 4.8　Hello.cc(PPElib による記述)

4.3.2 基本プログラム（2）

2つめのプログラムはHPC++Libのサンプルプログラム Sample である．このプログラムは，HPCxx_Thread クラスの基本的な利用方法を説明するサンプルプログラムで，いくつかのスレッドを生成し，並列に実行する．また，このプログラムはセマフォを利用したスレッドの終了同期を行っている．図 4.9 は HPC++Lib のプログラム，図 4.10 はこれを PPElib で記述し直したプログラムである．この例では，2つのプログラムの行数はほとんど同じであった．以下，2つのプログラムを比較しながら説明を行う．

図 4.9 の HPC++Lib の例では，10行目から 18行目までがスレッドの行う処理の部分である．PPELib でこれに対応するのは，図 4.10 の 6 ～ 11 行目の部分である．なお，この

部分のプログラムは両者でほとんど同じである．大きく異なるのは，図4.9の21～30行目の部分である．この部分は，スレッドをn個生成して（25行目），それぞれのスレッドを開始（26行目）し，すべてのスレッドの終了を28行目のセマフォのwaitにより待つ処理を行っている．PPElibでは，これに対応する部分はライブラリ内に含まれており，自動的に行われるため，PPElibのプログラムでは図4.10の31行目でのLWP数の設定と，37行目のRunメソッドによるタスクの起動だけとなる．

このように，複数のスレッドを生成し，生成したスレッドの同期をとるようなプログラムでは，スレッドの制御の部分がライブラリに含まれるためにユーザのプログラミングの負担は小さくなる．

```
 1: class MyThread: public HPCxx_Thread{
 2:   HPCxx_CSem *c;
 3:   int my_num;
 4: public:
 5:   MyThread( HPCxx_CSem * s , int m ):  HPCxx_Thread(){
 6:     my_num = m;
 7:     c = s;
 8:     printf("constructor complete\n");
 9:   }
10:   void run(){
11:     printf("hi from thread %d\n", my_num);
12:     double a = 1;
13:     for(int i = 0; i < 10000; i++)
14:       for(int j = 0; j < 1000; j++) a += sqrt((double) 1.0*j);
15:     printf("done with thinking in %d\n", my_num);
16:     c->incr();
17:     printf("done with write in %d\n", my_num);
18:   }
19: };
```

スレッドの処理

```
20:
21: int foo(int n){
22:   HPCxx_CSem c(n);
23:   MyThread *t[100];
24:   for(int i = 0; i < n; i++){
25:     t[i] = new MyThread(&c, i);
26:     t[i]->start();
27:   }
28:   c.wait();
29:   return 0;
30: }
31:
```

複数のスレッドの起動

```
32: int main(int argc, char *argv[]){
33:   HPCxx_Group *g;
34:   hpcxx_init(argc, argv, g);
35:   int x;
36:   cout << "num threads:" ;
37:   cin >> x;
38:   cout << "doing " << x << "threads.";
39:   for(int i = 0; i < 10; i++){
40:     double a;
41:     cout << "*** iteration " << i << " ***" << endl;
42:     foo(x);
43:     for(int i = 0; i < 100000; i++)
44:       for(int j = 0; j < 1000; j++) a += sqrt((double) 1.0*j);
45:   }
46:   printf("done with read\n");
47:   hpcxx_exit(g);
48:   return(0);
49: } /* main() */
```

図4.9　Simple.cc(HPC++Libによる記述)

```
 1: #define dID_myTask 1000
 2: class myTask: public PPE_Task {
 3: public:
 4:   myTask() : PPE_Task() { RegID = dID_myTask; }
 5:   static myTask* Create() { return(new myTask); }
 6:   void Process() {
 7:     cout << "Hi from thread " << mCurrentProcess << endl;
 8:     double a = 1.0;
 9:     for(int j = 0; j < 1000; j++) a += sqrt(1.0 * j);
10:     cout << "done with thinking in " << mCurrentProcess << endl;
11:   }
12: };
13:
14: class myApp : public PPE_Application {
15:   int  num;
16: public:
17:   void SetNum(int n) { num = n; };
18:   void main();
19:   void RegisterProcess();
20: };
21:
22: void myApp::RegisterProcess()
23: {
24:   PPE_ProcessManager::Register(dID_myTask,
25:           (PPE_RegFunction)myTask::Create);
26: }
27:
28: void myApp::main()
29: {
30:   myTask theTask;
31:   theTask.SetNumOfProcess(num);
32:
33:   cout << "using " << num << " threads.\n";
34:   for(int i = 0; i < num; i++) {
35:     double a = 1.0;
36:     cout << "*** iteration" << i << "***" << endl;
37:     theTask.Run();
38:     for(int j = 0; j < 1000; j++) a += sqrt(1.0 * j);
39:   }
40:   cout << "DONE" << endl;
41: }
42:
43: main(int argc, char** argv)
44: {
45:   myApp theApp;
46:   int n;
47:   if (argc > 1) n = atoi(argv[1]);
48:   if (n == 0) n = 1;
49:   theApp.SetNum(n);
50:   theApp.Run();
51: }
```

スレッドの処理

複数のスレッドの起動

図4.10　Simple.cc(PPElibによる記述)

4.3.3 同期処理（１）

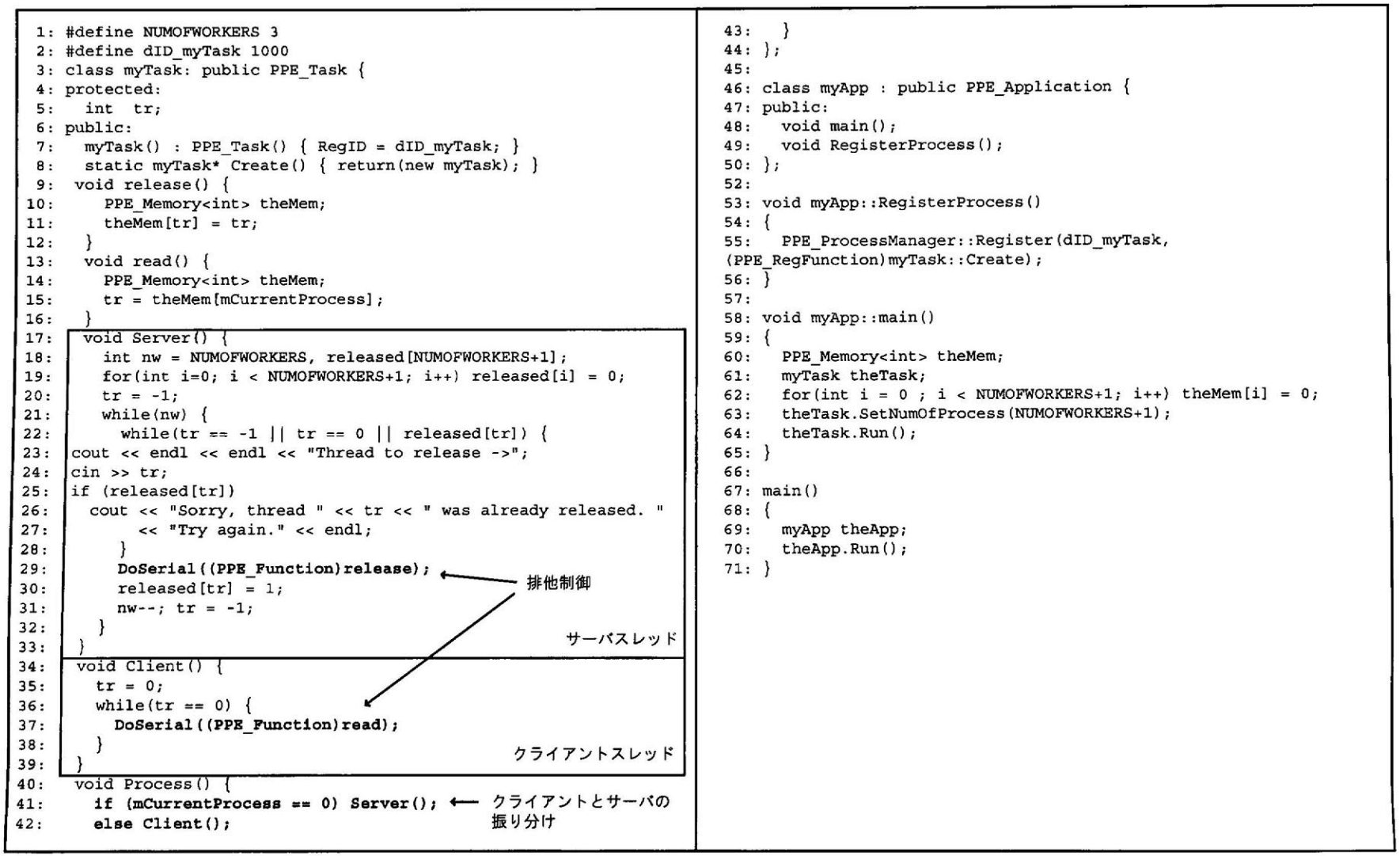
３つめのプログラムは，HPC++Libのサンプルプログラム Sync である．このプログラムでは，複数のWorkerスレッドがサーバスレッドと同期を行いながらデータのやりとりを行う．図4.11はHPC++Libのサンプルプログラムのソースリストである．図4.12は，PPElibでこのプログラムを書き直したものである．なお，図4.11のリストではプログラムの一部が省略されていおり，図4.12のすべてのプログラムを含んでいる．このため，HPC++Libに比べてかなりコード量が多く見える．

すべてのLWPが独立した処理ということがプログラミングの前提であるPPElibで，クライアント・サーバモデルのプログラムを記述するのは若干難しい．しかし，タスクのProcessメソッド（図4.12 40～43行目）のようにプロセスIDが0番のものをサーバスレッドとすることで，クライアント・サーバモデルのプログラムを記述することは可能である．また，サーバとワーカーの両者でのデータのやりとりには，タスクの持つ排他制御機構であるDoSerialメソッドを利用した関数の呼び出しを利用している（図4.12 29,37行目）．

以上のように，本プログラミングライブラリでは，サーバ・クライアントモデルのように，役割（処理）の異なるLWPを含んだタスクも記述することが可能である．また，DoSerialメソッドにより共有変数へのアクセスの排他制御も実現可能である．

```
 1: class Worker: public HPCxx_Thread {
 2:   HPCxx_Sync<int> &s;
 3:   int myID;
 4:   public:
 5:      Worker(HPCxx_Sync &s_, int i): s(s_), myID(i),
HPCxx_Thread()
 6:      {
 7:         cout << "Thread " << myID << " init complete" << endl;
 8:      }
 9:      void run()
10:      {
11:        int x;
12:        x=s;
13:        cout << "Thread " << myID <<
" thanks you for your support " << endl;
14:      }
15: };
16:
17: int main(int argc, char **argv)
18: {
19:   int tr=-1, nw=NUMWORKERS, far[NUMWORKERS];
20:   HPCxx_Group *g;
21:   HPCxx_Sync<int> sar[NUMWORKERS];
22:
23:   hpcxx_init(argc, argv, g);
24:   for(int i=0; istart();
25:     far[i]=0;
26:   }
27:   while(nw) {
28:     while(tr==-1 || far[tr]) {
29:       cout << endl << endl << endl << "Thread to release ->";
30:       cin >> tr;
31:     }
32:     sar[tr]=tr;
33:     far[tr]=1;
34:     tr=-1;
35:     nw--;
36:   }
37:   hpcxx_exit(g);
38:   return 0;
39: }
```

図4.11　Sync.cc(HPC++Libによる記述)




```
 1: #define NUMOFWORKERS 3
 2: #define dID_myTask 1000
 3: class myTask: public PPE_Task {
 4: protected:
 5:   int  tr;
 6: public:
 7:   myTask() : PPE_Task() { RegID = dID_myTask; }
 8:   static myTask* Create() { return(new myTask); }
 9:  void release() {
10:     PPE_Memory<int> theMem;
11:     theMem[tr] = tr;
12:   }
13:   void read() {
14:     PPE_Memory<int> theMem;
15:     tr = theMem[mCurrentProcess];
16:   }
17:  void Server() {
18:    int nw = NUMOFWORKERS, released[NUMOFWORKERS+1];
19:    for(int i=0; i < NUMOFWORKERS+1; i++) released[i] = 0;
20:    tr = -1;
21:    while(nw) {
22:      while(tr == -1 || tr == 0 || released[tr]) {
23: cout << endl << endl << "Thread to release ->";
24: cin >> tr;
25: if (released[tr])
26:   cout << "Sorry, thread " << tr << " was already released. "
27:        << "Try again." << endl;
28:      }
29:      DoSerial((PPE_Function)release);
30:      released[tr] = 1;
31:      nw--; tr = -1;
32:    }
33:  }
34:  void Client() {
35:    tr = 0;
36:    while(tr == 0) {
37:      DoSerial((PPE_Function)read);
38:    }
39:  }
40:  void Process() {
41:    if (mCurrentProcess == 0) Server();
42:    else Client();
43:   }
44: };
45:
46: class myApp : public PPE_Application {
47: public:
48:   void main();
49:   void RegisterProcess();
50: };
52:
53: void myApp::RegisterProcess()
54: {
55:   PPE_ProcessManager::Register(dID_myTask,
(PPE_RegFunction)myTask::Create);
56: }
57:
58: void myApp::main()
59: {
60:   PPE_Memory<int> theMem;
61:   myTask theTask;
62:   for(int i = 0 ; i < NUMOFWORKERS+1; i++) theMem[i] = 0;
63:   theTask.SetNumOfProcess(NUMOFWORKERS+1);
64:   theTask.Run();
65: }
66:
67: main()
68: {
69:   myApp theApp;
70:   theApp.Run();
71: }
```

図4.12　Sync.cc(PPElibによる記述)

4.3.4 同期処理（２）

同期処理の２つめのプログラムは，4.3.3の同期プログラムに加えて，各々のスレッドが，一定時間sleepしながら共有変数を書き換えるというものである．このプログラムも，クライアント・サーバーモデルのアプリケーションであり，HPC++Libの場合，main関数（メインスレッド）がサーバとなる．図4.13はHPC++Libのプログラム，図4.14はこれをPPElibで書き直したプログラムである．4.3.3節の同期の例と同じで，PPElibはクライアント・サーバーモデルのプログラムを記述する目的で設計されていないため，若干コード量が多くなる傾向がある．加えて，プロセスIDが0のLWPをサーバーとして振り分けるなどのプログラム上のテクニックが必要になる（図4.14 41～44行目）．しかしながら，図4.13の18～31行目に相当するスレッドの生成部を記述しなくてよいという利点があり，この点においては，HPC++Libよりプログラミングが容易となっている．

以上のように，サーバ・クライアントモデルのようにスレッド間で同期を頻繁に行うプログラムの記述はPPElibで行うことができるが，ライブラリの提供するモデルにマッチしていないためコード量は増加してしまう．コード量を見るとHPC++Libに比べて記述の面で不利だが，それでもスレッドの生成や終了同期についてユーザが記述しなくてよいという利点がある．

```
 1: class MyThread: public HPCxx_Thread{
 2:     HPCxx_SyncQ<int> *sync;
 3:     int my_num;
 4:     public:
 5:        MyThread( HPCxx_SyncQ *s , int m ):  HPCxx_Thread(){
 6:          my_num = m;
 7:          sync = s;
 8:          cout << "constructor complete" << endl;
 9:        }
10:        void run(){
11:           cout << "thread " << my_num << " sleeping..." << endl;
12:           sleep((10-my_num));
13:           *sync=my_num;
14:           cout << "thread " << my_num << " inserted value : "
<< my_num << endl;
15:        }
16: };
17:
18: int foo(int n) {
19:     HPCxx_SyncQ<int> s;
20:     MyThread *t[100];
21:     int temp;
22:     for(int i = 0; i < n; i++){
23:       t[i] = new MyThread(&s, i);
24:       t[i]->start();
25:     }
26:     for(i=0; i<n; i++) {
27:       temp=s;
28:       cout << "foo() recevied value " << temp << endl;
29:     }
30:     return 0;
31: }
32:                                          複数のスレッドを生成する処理
33: int main(int argc, char *argv[]){
34:     HPCxx_Group *g;
35:     hpcxx_init(argc, argv, g);
36:     int x;
37:     cout << "num threads:" ;
38:     cin >> x;
39:     cout << "doing " << x << "threads.";
40:     foo(x);
41:     printf("done with read\n");
42:     printf("DONE\n");
43:     hpcxx_exit(g);
44:     return(0);
45: } /* main() */
```

図4.13　SyncQ.cc(HPC++Libによる記述)

```
 1: #define dID_myTask 1000
 2: class myTask: public PPE_Task {
 3: protected:
 4:   int  num;
 5:   int* read;
 6: public:
 7:   myTask() : PPE_Task() { RegID = dID_myTask; }
 8:   static myTask* Create() { return(new myTask); }
 9:   void Read() {
10:     PPE_Memory<int> theMem;
11:     int max = GetNumOfProcess();
12:     for(int i=1; i < max; i++) {
13:       int val = theMem[i];
14:       if (val != 0&&read[i] == 0) {
15:  cout << "Received value " << val << endl << flush;
16:  num--;
17:  read[i] = 1;
18:       }
19:     }
20:   }
21:   void Server() {
22:     num = GetNumOfProcess() - 1;
23:     read = new int[num+1];
24:     for(int i=0; i < num+1; i++) read[i] = 0;
25:     while(num) {
26:       DoSerial((PPE_Function)Read);
27:     }
28:   }
29:   void Write() {
30:     PPE_Memory<int> theMem;
31:     int my_num = mCurrentProcess;
32:     cout << "LWP " << my_num << " sleeping..." << endl << flush;
33:     sleep((10 - my_num));
34:     theMem[my_num] = my_num;
35:     cout << "LWP " << my_num << " inserted value : "
36:          << my_num << endl << flush;
37:   }
38:   void Client() {
39:     DoSerial((PPE_Function)Write);
40:   }
41:   void Process() {
42:     if (mCurrentProcess == 0) Server();
43:     else Client();
44:   }
45: };
46:
47: class myApp : public PPE_Application {
48: public:
49:   void main();
50:   void RegisterProcess();
51: };
52:
53: void myApp::RegisterProcess()
54: {
55:   PPE_ProcessManager::Register(dID_myTask,
(PPE_RegFunction)myTask::Create);
56: }
57:
58: void myApp::main()
59: {
60:   PPE_Memory<int> theMem;
61:   myTask theTask;
62:   int num;
63:   cout << "enter number of LWPs: ";
64:   cin >> num;
65:   cout << "using " << num << " threads." << endl << flush;
66:   for(int i = 0 ; i < num+1; i++) theMem[i] = 0;
67:   theTask.SetNumOfProcess(num+1);
68:   theTask.Run();
69: }
70:
71: main()
72: {
73:   myApp theApp;
74:   theApp.Run();
75: }
```

クライアントとサーバの振り分け

図4.14　SyncQ.cc(PPElibによる記述)

4.3.5 排他制御

最後のサンプルプログラムは，HPC++Libの排他制御（HPCxx_Mutexクラス）を利用したものである．図4.15はHPC++Libのソースプログラム，図4.16はこれをPPElibでコーディングし直したものである．HPC++Libでは，排他制御のためにHPCxx_Mutexクラスが用意されており，クリティカルセクションの前後でlockおよびunlockメソッドを実行することで排他的な実行を実現している（図4.15 15～18行目）．なお，このサンプルでも図4.15の25～38行で，複数のスレッドを生成し，実行し，終了同期を行っている．PPElibでは，排他制御の機能をタスククラスのDoSerialメソッドとして用意しているため，このプログラムは比較的簡単にPPElibを用いて記述することができる．なお，具体的に排他制御を行っている部分は図4.16の14行目の部分である．

```
 1: class MyThread : public HPCxx_Thread {
 2:   HPCxx_CSem *sem;
 3:   HPCxx_Mutex l;
 4:   int &sharedVal;
 5:   int myId;
 6:
 7: public:
 8:   MyThread(int id, HPCxx_CSem *_sem, int &sv ):
 9:     myId(id), sem(_sem), sharedVal(sv), HPCxx_Thread(){}
10:   void run() {
11:
12:     sleep((unsigned)3);
13:     cout << "Thread " << myId << " waiting for my turn..."
<< endl;
14:                          ← ロック処理
15:     l.lock(); // critical section
16:     cout << "Thread " << myId << " has critical section!"
<< endl;
17:     sharedVal+=myId;
18:     l.unlock(); ← アンロック処理
19:
20:     sem->incr();
21:     cout << "Thread " << myId << " added in my value (exiting)."
 << endl;
22:   }
23: };
24:                                   複数のスレッドの生成処理
25: int foo(int n){
26:   HPCxx_CSem c(n);
27:   MyThread *t[100];
28:   int total=0;
29:
30:   for(int i = 0; i < n; i++){
31:     t[i] = new MyThread(i, &c, total);
32:     t[i]->start();
33:   }
34:   c.wait();
35:   cout << " Total of thread IDs is : " << total << endl << flush;
36:   cout << "DONE" << endl << flush;
37:   return 0;
38: }
39:
40:
41: int main(int argc, char **argv)
42: {
43:   HPCxx_Group *g;
44:   hpcxx_init(argc, argv, g);
45:   int n;
46:
47:   cout << "Number of threads : " << endl << flush;
48:   cin >> n;
49:   foo(n);
50:   return hpcxx_exit(g);
51: }
```

図4.15　Mutex.cc(HPC++Libによる記述)

```
 1: #define dID_myTask 1000
 2: class myTask: public PPE_Task {
 3: public:
 4:   myTask() : PPE_Task() { RegID = dID_myTask; }
 5:   static myTask* Create() { return(new myTask); }
 6:   void CriticalSection() {
 7:     PPE_Memory<int> theMem;
 8:     theMem[0] = theMem[0] + mCurrentProcess;
 9:   }
10:   void Process() {
11:     sleep((unsigned)3);
12:     cout << "LWP " << mCurrentProcess <<
" waiting for my turn..."
13:   << endl << flush;
14:     DoSerial((PPE_Function)CriticalSection);
15:     cout << "LWP " << mCurrentProcess <<
" added in my value (exiting)."
16:          << endl << flush;
17:   }
18: };
19:
20: class myApp : public PPE_Application {
21: public:
22:   void main();
23:   void RegisterProcess();
24: };
25:
26: void myApp::RegisterProcess()
27: {
28:   PPE_ProcessManager::Register(dID_myTask,
(PPE_RegFunction)myTask::Create);
29: }
30:
31: void myApp::main()
32: {
33:   PPE_Memory<int> theMem;
34:   myTask theTask;
35:   int num, total = 0;
36:   cout << "enter number of LWPs: " ;
37:   cin >> num;
38:   //  SetNumOfThread(num);  for PPEPlib
39:   theTask.SetNumOfProcess(num);
40:   theTask.Run();
41:   cout << " Total of thread IDs is : " << (int)theMem[0]
<< endl << flush;
42:   cout << "DONE" << endl;
43: }
44:
45: main()
46: {
47:   myApp theApp;
48:   theApp.Run();
49: }
```

排他制御

図4.16　Mutex.cc(PPElibによる記述)

以上，ここでは記述の似たHPC++Libと提案するPPElibを比較することで，PPElibの記述性と利点について述べた．なお，ここで比較に利用したサンプルプログラムはHPC++Lib向けに用意されたプログラムであるため，特に同期処理のプログラムなどPPElibでは記述しにくいものもがあったが，基本的なHPC++Libのサンプルは PPElibで再現することができた．このことから，よりプログラムのモデル制約の強いPPElibでもHPC++Libと同様のプログラムを記述できることを示すことができた．また，複数スレッドの生成とスレッドの終了同期などの部分についてはPPElibを用いた方がよりスマートに記述できることが確認できた．

以下では，最初からPPElib向けてプログラミングしたプログラムを例に，実際のPPElibを使ったアプリケーションの作成についてさらに詳しく説明する．

4.4 ライブラリを用いたプログラミング

本節では，具体的なPPElibを用いたプログラミングについて，行列演算を行うプログラム(martix)と，オセロゲーム(othello)を例にして説明を行う．

4.4.1 行列演算

行列演算のプログラムでは，行列積を求めるタスクの作成を例に説明を行う．このタスクの生成では，各行に対する演算を1つのLWPとした．したがって，LWPの数は行列の行数となる．例えば，300 × 300の行列であれば総LWP数は300である．

ここで作成した行列演算タスククラスは，アプリケーションクラスからは図4.17のように利用する．図4.17の例では，11行目のタスクの実行により行列aと行列aを掛け合わ（a × a）が並列に実行され，行列cに格納される．このように，アプリケーションクラスでは，タスククラス(Matrixクラス)が並列実行されることをい意識する必要はない．なお，このクラスでは，引数として行列を渡すために引数付きのRunメソッドを追加している．以下，このMatrixクラスの中身について詳しく説明する．

```
 1: void myApp::main()
 2: {
 3:   Matrix theMat(dSize);
 4:   double a[dSize][dSize];
 5:   double c[dSize][dSize];
 6:
 7:   memset(a, 0, dSize * dSize * sizeof(double));
 8:   a[0][0] = 1;  a[0][1] = 2;  a[0][2] = 3;
 9:   a[1][0] = 4;  a[1][1] = 5;  a[1][2] = 6;
10:   a[2][0] = 7;  a[2][1] = 8;  a[2][2] = 9;
11:   theMat.Run(&c[0][0], &a[0][0], &a[0][0]); // C = A x B
12: }
```

図 4.17　Matrix クラスの利用例

クラス定義

図 4.18 は，Matrix クラスのクラス宣言部である．1 行目は，PPElib のタスク (PPE_Task) を継承して Matrix クラスが定義されていることを示している．また，2～7 行目はタスククラス内のデータメンバの定義である．このクラスのプログラムで重要なのポイントは，16 行目で引数付きの Run メソッドが定義されていることと，17,18 行目に BeforeProcess,AfterProcess メソッドが定義されていることである．

```
 1: class Matrix : public PPE_Task {
 2: private:
 3:   PPE_Memory<char>* mMem;
 4:   PPE_Memory<char>* mA;
 5:   double*       mB;
 6:   int           mSize;
 7:   double*       mRes;
 8: public:
 9:   PPE_MatrixMul() : PPE_Task() { RegID = dID_MatrixMul; }
10:   PPE_MatrixMul(int inSize);
11:   ~PPE_MatrixMul() { delete mMem; }
12:   static PPE_MatrixMul* Create();
13:   void SetData(double* inA, double* inB);
14:   void GetResult(double* outC);
15:   void Run();
16:   void Run(double* outC, double* inA, double* inB);
17:   void BeforeProcess();
18:   void AfterProcess();
18:   void Process();
19: };
```

図 4.18　Matrix クラスの宣言部

RUN メソッド

図 4.19 は，引数付きの Run メソッドのプログラムリストである．このメソッドでは，引数として渡されたデータの共有メモリへの書き込み (3～5 行目)，LWP 数を行列の列

数に設定（6行目），タスククラスのRunメソッドの呼び出し（7行目），および実行後の共有メモリからの行列データの読み出し（8行目）を行っている．このように，このメソッドでは，タスクの実行（Runメソッドの呼び出し）前後のインタフェースが記述されている．

```
1: void Matrix::Run(double* outC, double* inA, double* inB)
2: {
3:   int theSize = sizeof(double) * mSize * mSize;
4:   mMem->Write(0, theSize, (char*)inA);
5:   mMem->Write(theSize, theSize, (char*)inB);
6:   SetNumOfProcess(mSize);
7:   PPE_Task::Run();
8:   mMem->Read(theSize * 2 , theSize, outC);
9: }
```

図 4.19　Run(double* outC, double* inA, double* inB)のプログラムリスト

BeforeProcess/AfterProcessメソッド

また，各プロセッサでのLWP実行の前後に実行されるBeforeProcessおよびAfterProcessメソッドは図4.20のような処理を行っている．BeforeProcessでは共有メモリのオブジェクトを生成し，行列の片方(mB)を共有メモリからローカルメモリへ転送している（9行目）．また，結果を格納するためのメモリ(mRes)を確保している（10行目）．このようにBeforeProcessで共有メモリからローカルメモリへデータを転送しておけば，LWPではローカルメモリに対してデータアクセスすればよくなるため，実行効率を高めることができる．また，AfterProcessでは，BeforeProcessで確保したメモリを解放している．このように，BeforeProcessおよびAfterProcessには，各プロセッサでLWPを実行前後の処理を記述できる．

```
 1: void PPE_MatrixMul::BeforeProcess()
 2: {
 3:   int theSize;
 4:   mSize = GetNumOfProcess();
 5:   theSize = mSize * mSize * sizeof(double);
 6:   mMem = new PPE_Memory<char>(theSize * 3);
 7:   mA   = new PPE_Memory<char>(theSize * 3);
 8:   mB   = new double[mSize * mSize];
 9:   mMem->Read(theSize, theSize, mB);
10:   mRes = new double[mSize];
11: }
12:
13: void PPE_MatrixMul::AfterProcess()
14: {
15:   delete mMem;
16:   delete mA;
17:   delete[] mB;
18:   delete[] mRes;
19: }
```

図 4.20　BeforeProcess および AfterProcess メソッドのプログラムリスト

Process メソッド

次に，図 4.21 に，Process メソッドを示す．Process メソッドでは，共有メモリから必要な部分行列の読み出し（10～11 行目）と theCurrent(=mCurrentProcess)で指定される行の乗算を行い（12～18 行目），結果を共有メモリの書き戻している（20～21 行目）．

以上のように，LWP の処理を記述する Process メソッドでは，プロセス ID で指定される行の演算を行い，結果を書き戻している．なお，このメソッドでは行列 B は BeforeProcess メソッドで既に読み込まれているため，ローカルメモリに格納されたデータ(mB)をアクセスしている．

```
 1: void Matrix::Process()
 2: {
 3:   int  theCurrent = mCurrentProcess;
 4:   int  theSize = mSize;
 5:   int  theDSize = mSize * mSize;
 6:   double* theA;
 7:   double  theRes;
 8:
 9:   theA = new double[theSize];                    部分行列の読み出し
10:   mA->Read(dCalc_XY(0, theCurrent) * sizeof(double),
11:     theSize * sizeof(double), theA);
12:   for(int i = 0; i < theSize; i++) {
13:     theRes = 0;
14:     for(int j = 0; j < theSize; j++) {
15:       theRes += theA[j] * mB[dCalc_XY(i, j)];
16:     }
17:     mRes[i] = theRes;
18:   }
19:                                                  結果の書き戻し
20:   mMem->Write(dCalcC_XY(0, theCurrent)*sizeof(double),
21:         theSize * sizeof(double), (char*)mRes);
22:   delete[] theA;
23: }
```

図 4.21　Process メソッドのプログラムリスト

4.4.2 オセロゲーム

アルゴリズム

PPElib を用いた２つめのアプリケーション例は，オセロゲームである．オセロゲームアプリケーションでは，コンピュータvs.コンピュータで対戦を行うオセロゲームの作成を例に，PPElib を用いた並列アプリケーションの作成について説明を行う．

オセロゲームは，典型的な探索プログラムである．ここでのプログラミングには，チェスなどのゲームでもよく利用されている α - β 法を利用した．以下，オセロゲームのアルゴリズムについて簡単に説明する．

オセロゲームの思考ルーチンでは，まず，盤面のある場所に自分が駒を打ったと仮定する．この時，駒を打つことによって何個の駒がひっくり返るか演算し，評価関数を用いてその場所に打ったときの利益を演算する．本プログラムでは，評価関数として以下の関数を利用している．

$$f = \max(f'(x,y), f'(x,y) = \sum_{t=1}^{8} g(x,y,z) + c(x,y)$$

g(x,y,t)は，方向t（図4.22参照）に対して盤面の評価値の合計値であり，c(x,y)は，盤面の(x,y)の位置の評価値である．ここでいう，評価値とは盤面のマス目に対してつけられている値で，例えば図4.23のようなものである．g(x,y,t)としては，例えば図4.22の方向8に対して図4.23の評価値をもとに求めたとすると，

$$g(4,4,8) = c(5,5) + c(6,6) = 0 + 10 = 10$$

となる．プログラムでは，この計算を全8方向に対して行う．つまり，f(x,y)は，盤上の(x,y)に駒をおいた場合の置いた石と取れた石の評価値の合計になる．したがって，fはこの合計のうち最も大きい値を得られるものとなり，結果として最も大きい値（価値の高い）(x,y)に駒を置くことになる．

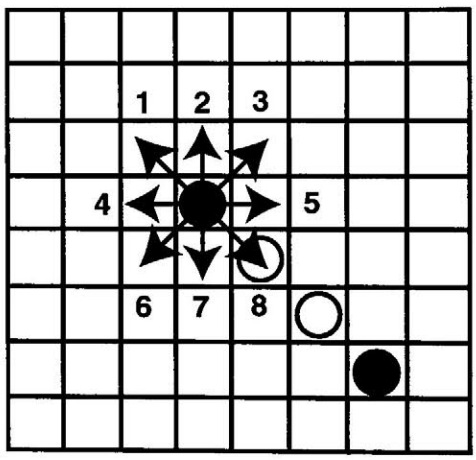


図4.22　方向t

99	-30	20	10	10	20	-30	99
-30	-50	-1	-1	-1	-1	-50	-30
20	-1	10	1	1	10	-1	20
10	-1	1	0	0	1	-1	10
10	-1	1	0	0	1	-1	10
20	-1	10	1	1	10	-1	20
-30	-50	-1	-1	-1	-1	-50	-30
99	-30	20	10	10	20	-30	99

図 4.23　評価値の例

これが，「深さ１（１手読み)」の場合のオセロゲームのアルゴリズムである．深さ２の場合は，自分が駒を置いた局面で同じ評価を今度は相手の立場で行う．そして，相手の得られる利益（最大の評価値）を自分の利益から差し引くことで，利益の修正を行う．深さｎの思考を行うオセロゲームでは，この盤面の評価を再帰的にｎ回繰り返す．このため，深さが深くなれば深くなるほど，オセロゲームの演算量は大きくなる．

以上が作成したプログラムのオセロゲームの基本的なアルゴリズムである．作成したプログラムでは，この処理を先手と後手が交互に行い両者がおける場所が無くなった時点で１回のゲームが終了する．なお，プログラムではこの操作を先手と後手を入れ替えて計２回行う．以下，プログラムのソースコードを用いて，並列処理化に関する部分に重点を置いて説明を行う．

アプリケーションクラスのmainメソッド

本プログラムでは，上記に説明した駒を置くための処理（思考）部分をタスクとした．アプリケーションクラスでは，このタスクを，駒をおく都度呼び出している．オセロゲームのマス目の数は６４マスである．したがって，１回の対戦で両者で６４回のタスクの呼び出しが行われる．

図4.24は，このオセロゲームのアプリケーションクラス(OthelloAppクラス)のmainメソッドのプログラムリストの一部である．このプログラムでは14～15行目で各種変数の初期化，16～38行目で実際のゲームの処理，39行目で勝ち負けの判定のための駒のカウントを行っている．この中で並列実行が行われるのは，18行目と28行目のオセロの思考を行うタスククラス（PutOneクラス）のオブジェクトthePutのThinkingメソッドの部分である．このプログラムでは，Thinkingメソッドの実行が終了すると，次に駒を置

く位置が変数theX, theYに返されるので, (theX, theY)で示されるマス目に駒を置く処理が行われる（22～23行目，32～33行目）．なお，theXに-1が返された場合には，置ける場所が無かったということでパスの処理が行われる．

以下，Thinkingメソッドの処理について詳しく説明を行う．

```
 1: void OthelloApp::main()
 2: {
 3:   PutOne theThink;
 4:   int    theResult[2][2] = {0,0,0,0};
 5:   int    n, thePass, theDir;
 6:   char   theBoard[10][10]; // ボードの配置データ
 7:   PutOne thePut;
 8:   short  theX, theY;
 9:
10:   printf("%s 対 %s の対戦を開始します. \n", mPlayer[0].name, mPlayer[1].name);
11:   //
12:   // 先手 Player1 後手 Player2
13:   //                                                          初期化
14:   InitBoard(theBoard);
15:   PrintBoard(theBoard);
16:   for(n = thePass = 0; n < 60; ) {
17:     // 先手
18:     thePut.Thinking(&theX, &theY, 2, theBoard, 0, n);  // 並列実行部
19:     if (theX == -1) {
20:       thePass++;
21:     } else {
22:       theDir = CanPut(theX, theY, 2, theBoard);
23:       Put(theX, theY, 2, theDir, theBoard);
24:       thePass = 0; n++;
25:     }
26:     PrintBoard(theBoard);
27:     // 後手
28:     thePut.Thinking(&theX, &theY, 1, theBoard, 1, n);  // 並列実行部
29:     if (theX == -1) {
30:       thePass++;
31:     } else {
32:       theDir = CanPut(theX, theY, 1, theBoard);
33:       Put(theX, theY, 1, theDir, theBoard);
34:       thePass = 0; n++;
35:     }
36:     PrintBoard(theBoard);                    勝ち負け判定
37:     if (thePass >= 2) break;
38:   }
39:   CountResult(theResult[0], theBoard);
40:
41:   //
42:   // 先手 Player2 後手 Player1
43:   //
44:       :
45:       :
46:       :
47: }
```

図 4.24　OthelloApp クラスの main メソッドのプログラムリスト

タスククラスのThinkingメソッド

図4.25は，PutOneクラスのThinkingメソッドのプログラムリストである．Thinkingメソッドでは，Parameter構造体にLWPが利用する各種パラメータの値を設定し，これをRunメソッドの引数として渡すことでLWPに渡している．また，Runメソッドの実行前にSetNumOfProcessメソッドでLWP数を64に設定している（33行目）．ここで呼び出されるRunメソッドでは，先に説明した1手分の探索処理を行っている．Runメソッドによる探索の並列実行が完了すると，並列実行によって得られた各マス目の評価値を集計し，評価値が最大になるマス目を調べている（36～43行目）．

```
 1: //  outX, outY ... 駒を置く場所
 2: //  inColor     ... 駒の色(黒＝2,白＝1)
 3: //  inBoard     ... 現在の盤面
 4: //  inThink     ... 思考用データ
 5: //  inNum       ... 現在の手数
 6: int  PutOne::Thinking(short* outX, short* outY,
 7:          short inColor, char inBoard[10][10],
 8:          int  inPlayer, short inNum)
 9: {
10:   int  i, n;
11:   char theMaxdepth;
12:   Parameter theParam;
13:   ThinkF* theThink = &gOthelloApp->mPlayer[inPlayer];
14:   short theMax;
15:   PPE_Memory<int> theCost;
16:
17:   *outX = *outY = -1; // 初期設定
18:
19:   // どの思考データを利用するかTiming変数により決定する
20:   for(i = n = 0; i < 2; i++) {
21:     if (theThink->timing[i] <= inNum) n = i + 1;
22:   }
23:   // 読む深さの修正(終盤用)
24:   theMaxdepth = (theThink->depth[n] > (61 - inNum)) ?
25:   61 - n : theThink->depth[n];
26:   // パラメータ設定
27:   theParam.depth    = 1;
28:   theParam.maxdepth = theMaxdepth;
29:   theParam.color    = inColor;
30:   theParam.n        = inPlayer;        LWP数の設定
31:   theParam.use      = n;
32:   memcpy(theParam.board, inBoard, sizeof(char)*100);
33:   SetNumOfProcess(64);
34:   Run((char*)&theParam, sizeof(struct Parameter));
35:   // 計算結果集計
36:   theMax = (short)-32767;
37:   for(i = 0; i < 64; i++) {
38:     if (theMax < (short)theCost[i].Val()) {
39:       theMax = theCost[i].Val();
40:       *outX = i % 8;
41:       *outY = i / 8;
42:     }
43:   }
44:   return(theMax);                        最大の評価値の検索
45: }
```

図 4.25　PutOne クラスの Thinking メソッドのプログラムリスト

以上のように，オセロゲームにつては，PPElibが提供するプログラミングモデルで自然な記述を行うことができる．また，あらかじめ用意されたクラスを継承して記述することで,並列実行に必要となるLWPのプロセッサへの割り当てやバリア同期に代表される同期処理を記述することなくプログラミングを行うことができる．このように，本ライブラリを用いることで２章のDSEのプログラムサンプルで必要になっていた並列プロ

グラミング記述に伴うバリア同期などの煩雑な部分の記述を省略することができ，ユーザの同期操作挿入ミスによるデットロックに代表される並列処理特有のバグの発生を抑えることができる．

4.5 DSE への PPElib の実装

本節では，PPElib の DSE 上での実装方法について説明する．DSE では，複数の仮想プロセッサが異なる UNIX プロセス上で実行されている．また，複数のワークステーションまたは PC を用いて環境が実現されるため，すべての仮想プロセッサで LWP を実行するためには，LWP 処理を含むオブジェクト（タスクオブジェクト）を他の計算機で実行する必要がある．したがって，ライブラリの実行のためには，何らかの遠隔オブジェクト生成手段が必要になる．このような，遠隔オブジェクト生成手段としては，Java RMI や HORB[33]などで利用されている Proxy を利用する方法がある．しかし，PPElib では，Peoxy が実現する遠隔メソッド呼び出しの機構は必要ない．そこで，DSE に対する PPElib の実装では「DSE での並列アプリケーション実行では，アプリケーション実行時にすべての仮想プロセッサで同一プログラムが起動される」という特徴を利用して，より簡易な遠隔オブジェクト生成方法をとった．

DSE からのアプリケーションの起動と初期化

図 4.26 は，PPElib を用いた作成したアプリケーションが DSE 上でどのように実行されるかを示したものである．DSE のコンソールからアプリケーションのファイル名が入力されると，最初に PE0（ホストプロセッサ）で，アプリケーションが起動され main 関数が実行される(1)．main 関数では，アプリケーションオブジェクト（myApp）の生成が行われ，このオブジェクトの Run メソッドを呼び出す．Run メソッドでは，まず，DSE の初期化と各種変数の初期化を行う(2)．次に，RegisterProcess メソッドが呼び出され，この後main メソッドが呼び出されユーザの記述した並列アプリケーションに制御が移される．なお，PE0 で DSE の初期化が行われると，他の PE でもアプリケーションが起動され，同様にアプリケーションオブジェクトが生成され，Run メソッドが呼び出され RegisterProcess メソッドが呼び出される(3)．しかしながら，他の PE ではアプリケーションオブジェクトの main メソッドの呼び出しは行われず，プロセス起動待ち状態になる(4)．

以上が，アプリケーションが起動された直後の動作である．この動作が完了した状態では，すべてのPE上でRegisterProcessメソッドが実行されているため，すべてのPEでタスクIDとタスク生成メソッドの登録が完了していることになる．つまり，この時点でタスクIDの受け渡しだけで，他のPEでタスク生成が可能な状態になっていることになる．

アプリケーションの実行とタスク生成

PE0で実行されているアプリケーションオブジェクトでタスクオブジェクトのRunメソッドが呼び出されると，PE0から他のPEのプロセスマネージャに対してタスクIDを引数としてタスク生成メソッドの起動要求が送られる(5)．この要求を受けた各PEでは，プロセスマネージャが呼び出され，タスクIDに対応するタスクオブジェクトが生成される(6)．このとき，PE0で実行されていたタスクオブジェクトのデータメンバのうち，基本的なものがコピーされる．そして，タスクオブジェクト内のLWPアロケータが呼び出されLWPの割り当てと実行が開始される．PE0はLWPの実行終了待ち状態になりLWPの実行が完了するのを待つ(7)．各PEによるLWPの実行がすべて完了すると，再びプロセスマネージャに制御が移り同期処理（バリア同期処理）が行われる(8)．同期後，PE0以外では，生成したタスクオブジェクトが削除され，再びプロセス待ち状態に遷移する．また，PE0では，タスク実行完了待ちの同期から抜け，再びユーザプログラムに制御が移動する(9)．

以上のように，PPElibではDSEの特徴を利用したリモートでのタスクオブジェクトの生成が行われる．このように遠隔オブジェクト生成を行うことで，Proxyに比べオーバヘッドを削減することができ，またDSE側への機能拡張なしにPPElibを実装することが可能となった．

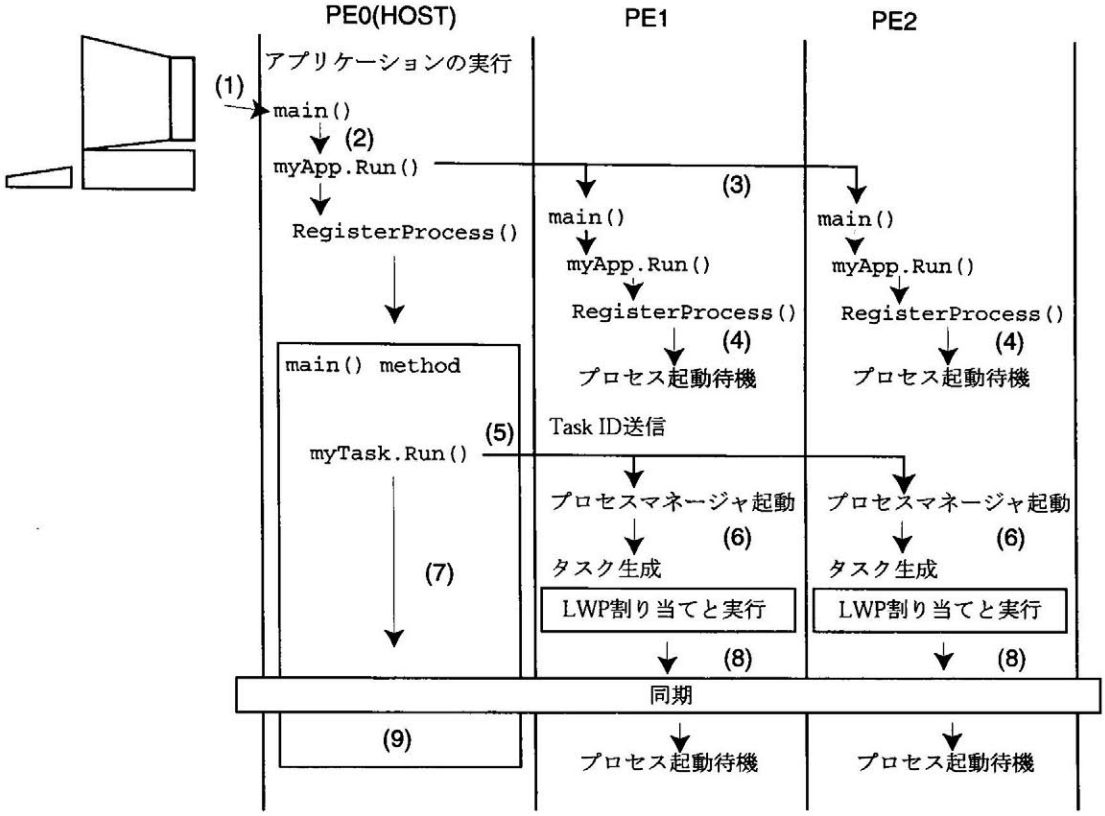


図 4.26　DSE での PPElib アプリケーションの実行イメージ

4.6 まとめ

本節では，オブジェクト指向の並列プログラミングライブラリ PPElib のプログラミングモデルについて説明した．また，HPC++Lib のサンプルプログラムとの比較およびいくつかのアプリケーションを PPElib を用いて作成することで，PPElib によるプログラミングについて説明した．さらに PPElib の DSE の実装について説明した．

現在，PPElib は，DSE および UNIX の pthread ライブラリ上で動作しており，また逐次版も用意されているため，逐次環境から並列環境への移植性も高いライブラリとなっている．以降の章では，パフォーマンス向上のための PPElib の内部（プロセス割り当ておよびキャッシング）の実装について述べる．

第５章

PPElibにおける動的プロセス割り当て機構

5.1 概要

４章で説明したように，PPElib ではLWP の自動的な割り当て機構を備えている．この割り当て機構では，アプリケーションの実行時に動的にLWP を割り当てている．一般的に，プロセス割り当て機構は，大きく静的割り当てと動的割り当ての２つに分類される．以下，PPElib のプロセス割り当てについて述べる前に，２つの割り当て手法の概要，利点および欠点について説明する．

5.1.1 静的プロセス割り当て

静的プロセス割り当ては，プログラム実行前にプロセスの割り当てを決定する方式である．このような方式としては，例えば，プログラム中にプロセス割り当てをあらかじめ記述しておく方式や，実行時にパラメータファイルなどで渡す方法がある．静的プロセス割り当て方式の場合，各プロセッサへ等分にプロセスを割り当てたり，計算機の性能に応じて割り当て数を決定したりする．

この方式では，プログラマが直接割り当てを指定することができるため，プログラムの性質に合わせて各プロセッサへの割り当てを決定することができるという利点がある．また，実行時には指定されたとおりにプロセッサへのプロセス割り当てを行えばよいため，プロセス割り当てのための負荷が小さいという利点もある．

しかしながら，各プロセスの処理量がデータに依存して変化するなど，あらかじめ各プロセスの処理量が分からないプログラムの場合には効率よく実行できるようにプロセスを割り当てることが難しいという問題がある．また，例えばマルチユーザ環境のよう

に外的要因により計算機の負荷が動的に変化する場合もプロセス割り当てを効率よく行うことができない.

以上のように，静的プロセス割り当て方式は，実行前にプロセス割り当てが決まっているため実行時の割り当て処理オーバヘッドが小さいという利点があるが，ダイナミックに処理量が変化するアプリケーションやダイナミックに計算機の負荷が変動する場合に対応できないという欠点がある.

5.1.2 動的プロセス割り当て

動的プロセス割り当ては，プログラム実行中にプロセスの割り当てを決定する方式である. このような方式としては，例えば，アイドルな（空いた）プロセッサ群をプロセッサプールと呼ばれるリストに登録しておき，プロセッサプールに登録されているプロセッサにプロセスを割り当てていく方式などがある.

この方式の利点は，以下の２つである.

(1) 処理量の異なるプロセスを，空いているプロセッサに効率よく割り当てられる
(2) 計算機の動的な負荷変動に対応してプロセッサ割り当てができる

この動的プロセス割り当てでは，静的プロセス割り当てで問題となっていた前述の2つの問題に対応できる. しかしながら，このような動的割り当てを実現するには，実行時にプロセッサ間でプロセッサの状態を交換する必要があるため，静的プロセス割り当てに比べてプロセス割り当てに伴う処理オーバヘッドが大きくなる傾向にある.

以上のように，動的プロセス割り当て方式は，実行するアプリケーションのプロセスや実行環境のダイナミックな変化に対応できるが，静的割り当てに比べ状態情報の交換のための処理が必要になる.

5.2 PPElib のプロセス割り当て機構

PPElib では，標準のプロセス割り当て方式として動的プロセス割り当て方式を採用している. PPElib の割り当て機構は，ライブラリの機能の一部として実装されているため，アプリケーションの性質によらず効率のよい実行を実現しなければならない. ダイナミックな変化に対応できない静的な割り当てではアプリケーションによらない効率実行

を実現できないため,PPElibのデフォルトの割り当て機構として動的割り当てを用いた.しかしながら,この割り当て機構についても,継承したユーザタスク内で上書きすることで柔軟に変更することが可能である.極端な例としては,ユーザが割り当て機構を変更すれば,静的な割り当てを実現することも可能である.

DCTアプリケーションでの動的プロセス割り当て

ここでは,PPElibが標準で用意している動的プロセス割り当て方式について説明を行う.図5.1は,画像に対してDCTを行うアプリケーションに対する動的割り当ての動作の例である.この例のアプリケーションでは,画像は8×8ピクセルのブロックに分割され,分割されたそれぞれのブロックがLWPとしてプロセッサに割り当てられて実行される.したがって,PPElibのプログラミングスタイルに沿って記述する場合には,画像全体のDCT処理がタスク,8×8の画像ブロックに対する処理がLWPとなる.このアプリケーションでは,図5.1のように各ブロックに0からnまでの番号を振り,これをプロセスIDをすることで処理するブロックとLWPのプロセスIDを1対1に対応させている.したがって,ユーザの記述したProcess メソッドでは,データメンバmCurrenProcess でプロセスIDを調べ,対応したブロックのDCT処理を行うことになる.以下,このアプリケーションでの動的プロセス割り当てについて図5.1を参照しながら説明を行う.

基本的に,PPElibによるLWPの割り当てと実行は各プロセッサに対してFCFS(First Come, First Served) で行われる.また,各プロセッサではLWPの複数同時実行は行わず,LWPの処理終了後に次のLWPを割り当てる.したがって,PPElibでは空いたプロセッサにLWPを割り当てることにより動的プロセス割り当てによる負荷分散を実現している.図5.1の例では,画像のDCT処理を3つのプロセッサを用いて実行している.また,プロセッサの速度はPE3>PE1>PE2 の順であり,PE3はプロセッサの負荷が動的に変化しているとする.図では,実行開始時にはそれぞれのプロセッサに0から2のLWPが割り当てられ実行が開始される.ここでは,PE3のLWP実行の終了が一番最初に終了するので,新たにPE3にプロセスID=3のLWPが割り当てられる.続いて実行が終了するのはPE1で,PE1にプロセスID=4のLWPが割り当てられ,実行される.このように,PPElibの動的割り当てでは,先に処理が終了したプロセッサにLWPが割り当てられることになる.このような,割り当て方針であるため,PE3のように動的に負荷が変動するプロセッサにも適当にLWPが割り当てられることになる.

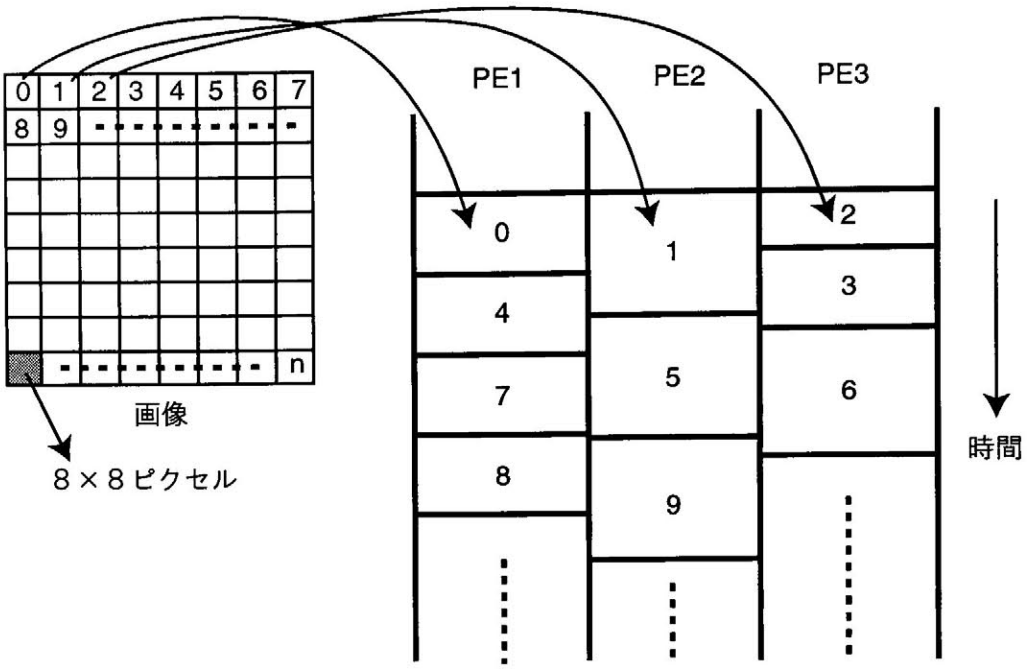


図 5.1　プロセス割り当て例

割り当て方式詳細

この割り当てを実現するために, PPElib では処理を終えたプロセッサ側がホストプロセッサにLWPをリクエストする方式をとった. 図5.2は, 各プロセッサがホストプロセッサへLWP をリクエストする処理の概念図である. LWP の割り当てを要求するプロセッサは, ホストプロセッサに要求(Request) を送る. 要求を受け取ったホストプロセッサでは, LWP 割り当て用変数(割り当てるプロセス ID)を読み出し, 読み出した値に 1 加算を行うとともに読み出した値を要求したプロセッサへ応答(Answer) を返す. ホストプロセッサでは, この要求から応答までの処理を不可分な処理として実行することで, 各プロセッサが重複したプロセス ID を獲得することを防いでいる.

応答を受け取ったプロセッサでは, 全LWP 数と受け取ったプロセスIDの比較を行う. もし, 受け取ったプロセス ID が全 LWP 数よりも大きな場合, 各プロセッサではすべての LWP の処理が完了したとして, LWP 割り当て処理を終了する.

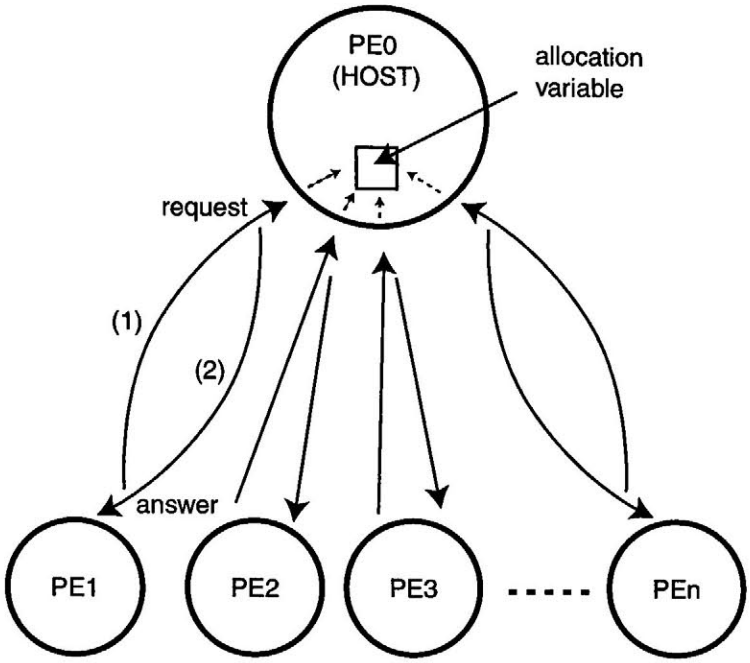


図 5.2　プロセス割り当て概念図

以上のように，PPElib の動的プロセス割り当てでは，空いたプロセッサがホストプロセッサへLWP を要求する．また，ホストプロセッサでは，要求から応答までの処理を不可分な処理として実行する．これにより，各プロセッサへの重複しないLWP割り当てを実現している．また，プロセス割り当ての終了確認を各プロセッサ側で行うことで，割り当て負荷の集中するホストプロセッサ側の処理を軽減している．

加えて，ライブラリによる動的プロセス割り当ての実現は，ユーザの記述なしでの動的プロセス割り当てを可能にした．さらに，ライブラリによるプロセス割り当ては，プロセッサの増減に対して再コンパイルなしの適応を可能にする．

5.3 実装

前節で説明した動的プロセス割り当て機構のDSE への実装は, DSE の提供する共有メモリに対する FAA(フェッチ・アンド・アッド, Fetch&Add) 操作を利用した．FAA操作は，共有メモリの読み出しと，読み出した値への加算，および新しい値の書き戻しを不可分な処理として実現する DSE の共有メモリ操作である．このFAA操作において，加算する値に 1 を設定することで，前節で説明したホストプロセッサでの処理が実現できる．

このように, DSE の持つFAA操作を利用することでプロセス割り当て機構を要求と要求への応答の２つのメッセージ交換で実現できた．また，FAA処理で加算する値を大きくすることで, 一度のプロセス割り当て要求操作で複数のLWPを一度に割り当てることも可能となる．これを利用すれば, SMP クラスタへの一括したプロセス割り当てを実現することもできるようになる．

5.4 評価実験

評価実験では,（1）クラスライブラに実装したプロセス割り当てアルゴリズムの有効性を評価するための実験と,（2）プロセス割り当て性能へのホストプロセッサの影響を調査するための実験の２つの実験をおこなった．以下では，これらの２つの実験について説明を行う．

5.4.1 実験環境

動的プロセス割り当ての評価実験では, 性能の異なる4台のPC/AT互換機を利用した．表5.1 に，各計算機の性能を示す．表中の速度比は，AMD K6-2を1.0 とした時の各計算機の相対的な速度であり，この値が大きいほど高速なことを意味する．なお，この速度比の計算には，以降の実験で利用した400 × 400 の行列積の演算を逐次実行した結果を利用した．また，計算機間を接続するネットワークとしては，100Base-T のFast イーサネットを利用し，各計算機は同じスイッチングハブに接続した．なお，OS としてLinux（kernel 2.0.26）を, コンパイラとしてgcc(2.7.2.3）を利用した．

以降の説明では，それぞれの計算機をCPU のクロック(300Mhz, 166Mhz, 266Mhz, 450Mhz) で呼ぶこととする．

表 5.1　実験に用いた計算機の性能一覧

CPU	Clock (Mhz)	Memory (byte)	速度比
AMD K6-2	300	64M	1.0
MMX Pentium	166	64M	0.3
PentiumII	266	64M	2.8
PentiumII	450	64M	4.3

5.4.2 実験 1

1つめの実験では，PPElib に実装した動的プロセス割り当て機構を評価するために，すべての仮想プロセッサに対して同じ数のLWPを割り当てる静的スケジューリング機構をPPElib に実装し，動的プロセス割り当て機構を実装したPPElib との比較実験を行った．比較に用いたアプリケーションは，すべてのLWPの計算粒度が等しい，行列積を演算するアプリケーション(以下，Matrix と呼ぶ)と，LWP により計算粒度が異なるオセロゲーム(以下，Othello と呼ぶ)の2つである．なお，本実験では，Matrixの行列サイズが300 × 300 と400 ×400 の場合と，オセロゲームの思考の深さ(探索木の深さ)が4手と5手の場合について測定を行った．また，ホストプロセッサは，300Mhz 上に配置した．300Mhz は，ホストプロセッサ以外にDSE の仮想プロセッサを実行するため，計2つのDSE の仮想プロセッサ（ホストプロセッサと仮想プロセッサ）を実行する．一方，他の166Mhz,266Mhz, 450Mhz は，1つの仮想プロセッサのみ実行している．しかし，ホストプロセッサはタスクの並列実行に参加しないため，実際に並列実行に参加するプロセッサの数は各計算機1つずつとなる．

プロセッサ数を変更しての実行

図5.3-5.6 は，実行を行った結果のグラフである．グラフでは，横軸に並列実行に利用した計算機を，縦軸に処理時間をとっている．横軸は実行に利用した計算機を表しており，例えば”300+166Mhz ”と書かれている場合は，実行が300Mhz と166Mhz を利用して並列実行を行ったことを意味している．これらの結果より，ほとんどの場合において動的割り当て(dynamic) の方が静的割り当て(static) に比べて実行に要した時間が短く，また，パフォーマンスが安定していることが分かる．特に，166Mhz が仮想プロセッサとして参加した場合には，static では極端にパフォーマンスが低下しているのに比べ，dynamic ではほとんど速度が低下していない．これは，処理能力に応じてLWP を割り当てる動的プロセス割り当て機構がうまく働いている結果である．

以上のように，性能差のある計算機を用いた並列実行では動的割り当ての効果が大きなことが確認できた．なお，図5.3 のオセロの4手読みの場合のみstatic とdynamic が同程度の処理時間となっているが，これについては後で詳しく説明を行う．

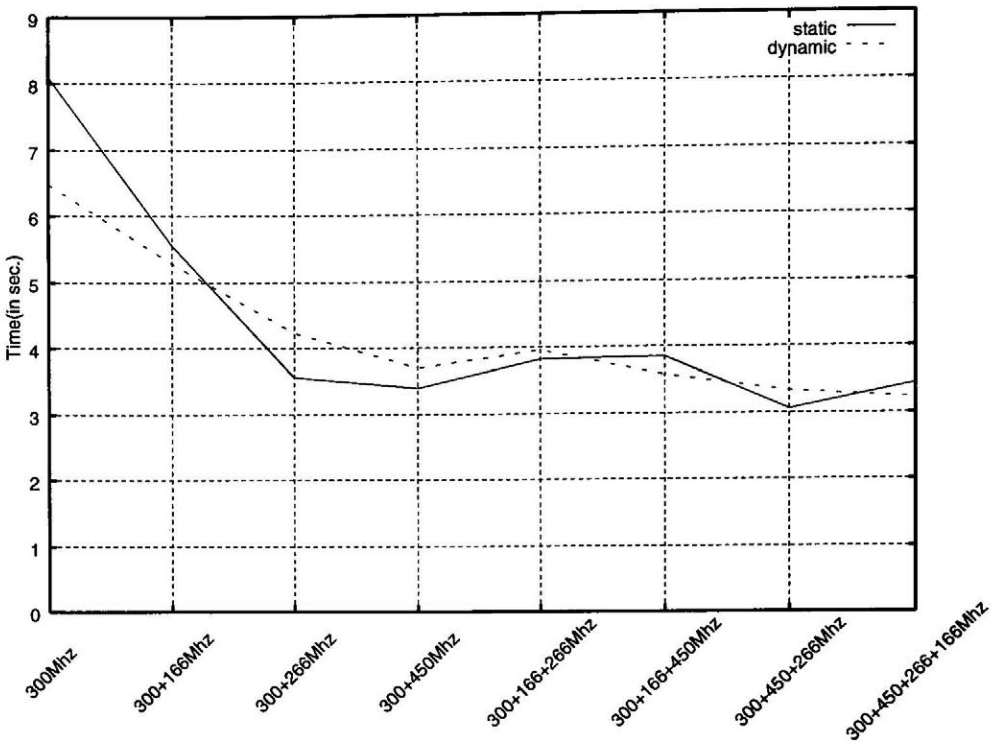


図 5.3　Othell(4 手読み)の実行結果

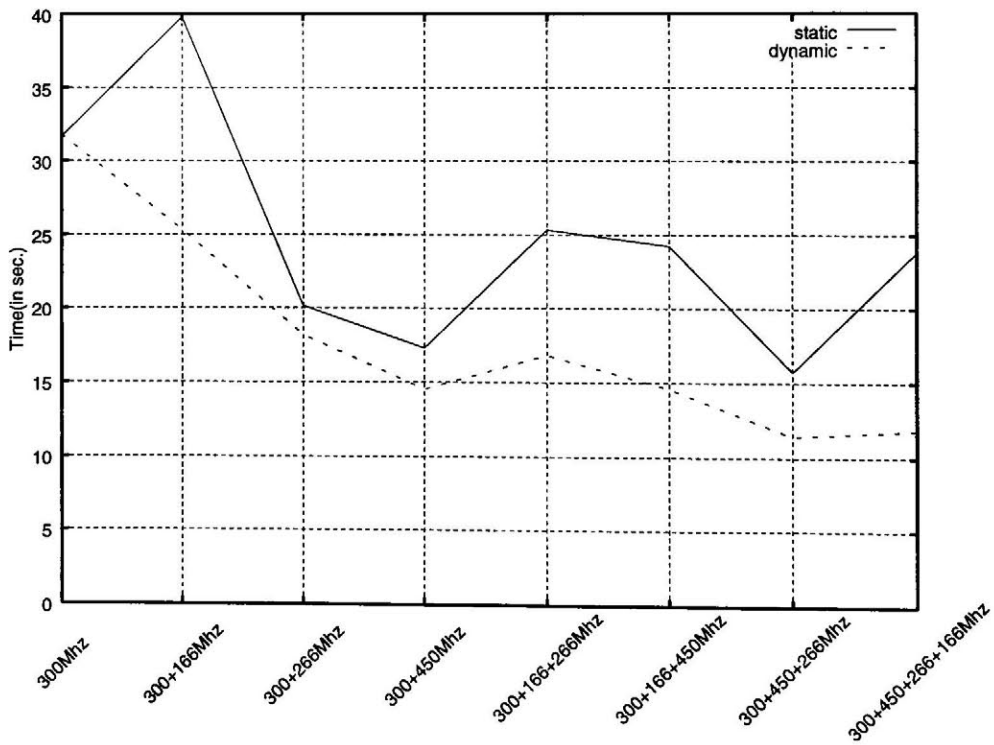


図 5.4　Othell(5 手読み)の実行結果

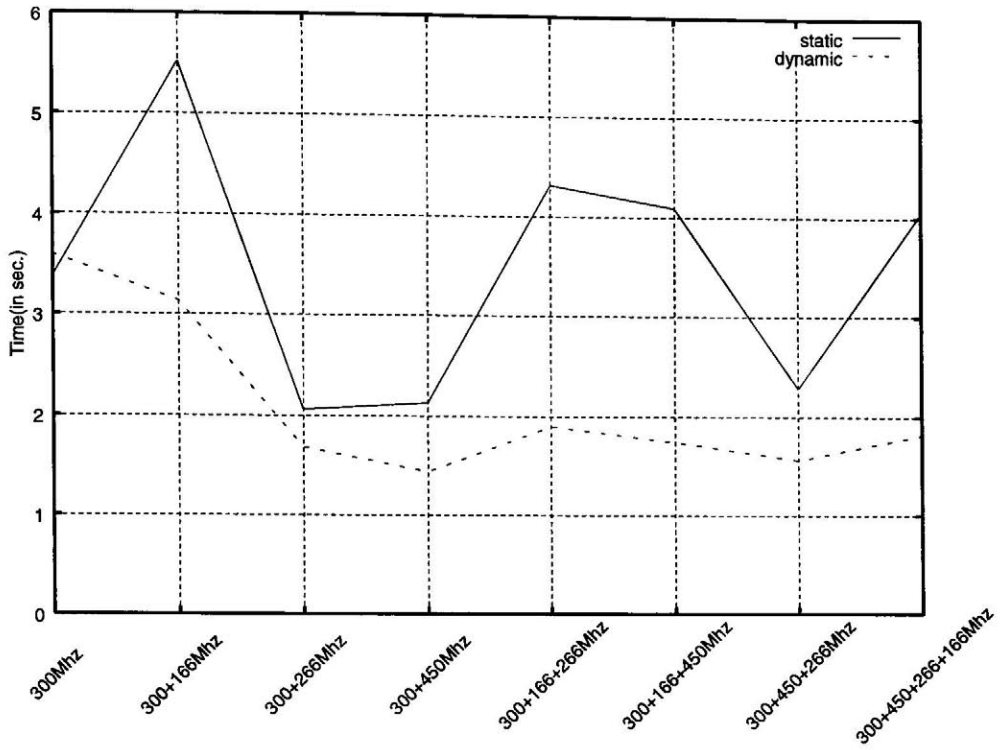


図 5.5　Matrix(300× 300)の実行結果

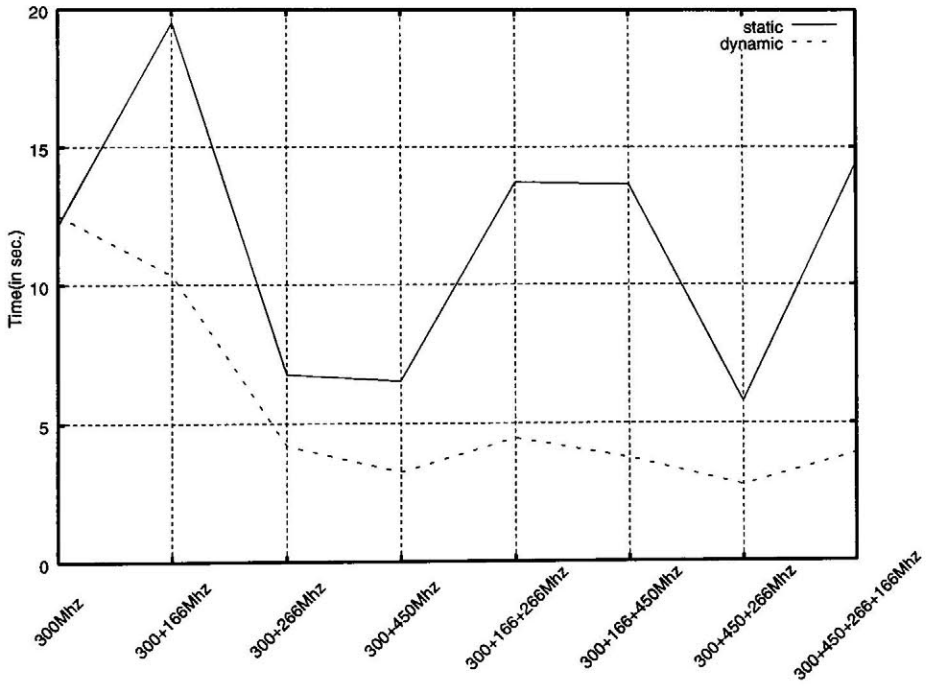


図 5.6　Matrix(400× 400)の実行結果

各仮想プロセッサへのLWPの割り当て数

次に,4つの計算機すべてを利用して並列実行した場合の,各仮想プロセッサへのLWP割り当て数について図5.7, 5.8 を利用しながら説明を行う.図5.7, 5.8 は,各仮想プロセッサに割り当てられたLWP数のグラフである.グラフにおいて,横軸は仮想プロセッサを,縦軸は割り当てられたLWP数を示している.このグラフより,dynamic ではほとんどの場合においても高速なプロセッサに多くのLWPが割り当てられていることが分かる.しかしながら,オセロの場合は,266Mhz と300Mhz では266Mhz の方が処理能力が高いにも関わらず,300Mhz により多くのLWPが割り当てられている.この理由は以下のとおりである.前述したように,オセロでは,駒を盤面に置くためのコスト計算を１つのLWPとしている.このように１手打つ処理を１つのLWPとしているため,手を計算するために生成されるLWP数は盤面のサイズ（８マス×８マス）と同じ６４個である.オセロゲームの処理では,駒を置くことができなければ探索は行われないたいためすぐにLWPの実行が終了する.また,オセロゲームでは,盤面の状態によって多少は変化するものの,ほとんどの場合,駒を置くことができるマス目は数個に限られるため,多くのLWPはほとんど計算を行わずにすぐに終了することになる.このため,LWPの実行は一瞬で終了し,LWP割り当てのための通信が頻繁に行われることになる.このとき,ホストプロセッサと同一の計算機上で実行される300Mhz の方がホストプロセッサとの通信遅延が小さいため,266Mhz に比べ多くのLWPが割り当てられる.なお,この傾向は,LWPの計算粒度がより小さい4手読み(othello4) の場合に顕著に表れている.また,グラフからは計算粒度が比較的大きなMatrixでは,ほぼ計算機の能力に応じて割り当てが行われてることも確認できる.

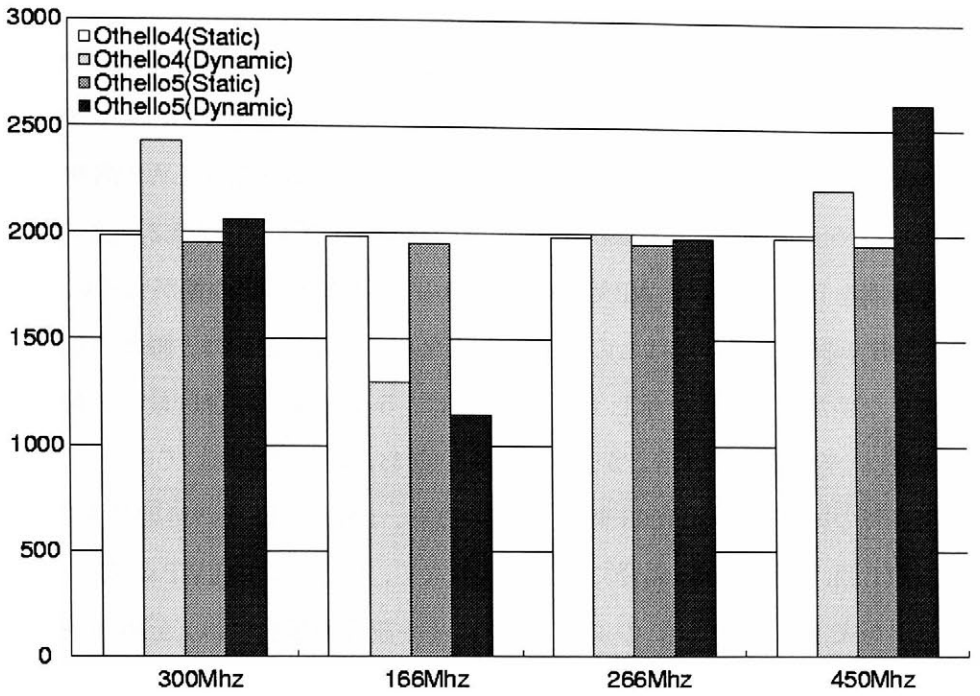


図 5.7　LWP 割り当て数(othello)

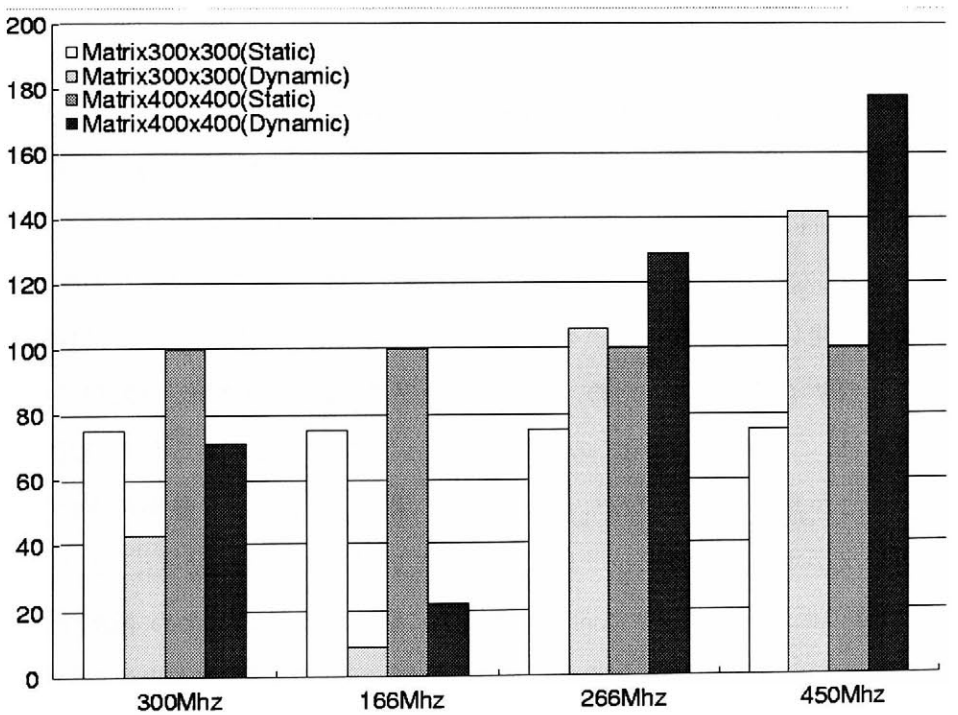


図 5.8　LWP 割り当て数(matrix)

動的プロセス割り当ての通信と実行時間の詳細

次に，表5.2 ～5.5 に，動的LWP 割り当てに関するより詳細な測定データを示す．これら表では，オセロおよび行列積のアプリケーション実行時の，各プロセッサの処理LWP 数，平均・最大・最小LWP 処理時間，平均・最大・最小LWP 割り当て待ち時間，平均・最小・最大通信発生間隔および通信回数を示している．表5.2，5.3 のオセロのLWP 処理時間を見ると，最大LWP処理時間と最小LWP処理時間の差が大きいことが分かる．特に，５手読みの場合にはこの差は大きくなっている．これは，個々のLWP の処理時間に大きな差があることを示している．例えば，５手読みの場合は最大と最小LWP処理時間の差は，約200 ～370 倍とかなり大きい．なお，表には記載していないが，オセロゲーム実行時の総LWP 数約7500 個のLWP のうち，10msec 以上の処理時間のものは４手読みで111 個，５手読みの場合で967 個と少なく，ほとんどのLWP の実行が10msec 以内で処理が完了していた．このように，特に４手読みの場合には，全体の98.5 %のLWP が10msec 以内に終了するため，ホストプロセッサへのLWP の割り当て要求の間隔が短くなる．このため，ホストプロセッサでのLWP 割り当てが頻繁に衝突し，この結果として割り当てオーバヘッドが増加する．これが，４手読みの場合に，静的な割り当てに比べて速度向上が得られにくい原因となっている．なお，これは，平均プロセス割り当て待ち時間の標準偏差からも確認することができる．

一方，行列積の結果を見ると，300 ×300 の場合でも，LWP の平均処理時間はオセロゲームに比べ十分大きく，また最小LWP 処理時間も大きいことがわかる．このため，行列積の場合には，動的スケジューリングのためのアクセス衝突は少なく抑えられれている．しかしながら，特に400 ×400 の行列積の演算では，300Mhz の平均プロセス割り当て待ち時間が大きくなっていることが確認できる．これは，ホストプロセッサ上の共有メモリのサイズが小さいため，行列のデータの一部が300Mhz の仮想プロセッサに割り当てられ，このデータに全ての仮想プロセッサがアクセスするため処理速度が低下していることが原因である．なお，300 ×300 の場合には，データ量が小さいためすべての行列データがホストプロセッサに配置されている．しかしながら，ホストプロセッサと300Mhz の仮想プロセッサは同じ計算機上に配置されているため，配列データアクセスの影響を少なからず受けると考えられる．なお，行列積演算のLWP 処理時間がばらついてしまう大きな理由は，この共有メモリ上のデータへのアクセスのオーバヘッドである．

表 5.2　オセロゲーム(4 手読み)の実行結果詳細（単位 msec.)

	動的				静的			
	300Mhz	166Mhz	266Mhz	450Mhz	300Mhz	166Mhz	266Mhz	450Mhz
処理LWP数	2427	1296	1998	2215	1984	1984	1984	1984
平均LWP処理時間	1.06 (1.66)	1.80 (2.78)	0.84 (1.46)	0.70 (1.23)	- -	- -	- -	- -
最小LWP処理時間	0.25	0.52	0.22	0.17	-	-	-	-
最大LWP処理時間	14.32	16.71	12.71	13.13	-	-	-	-
平均LWP割り当て待ち時間	0.70 (0.31)	1.39 (1.21)	1.30 (1.25)	1.25 (1.20)	- -	- -	- -	- -
最小LWP割り当て待ち時間	0.29	0.51	0.30	0.27	-	-	-	-
最大LWP割り当て待ち時間	4.32	13.49	13.03	12.20	-	-	-	-
平均通信発生間隔	0.57 (1.04)	0.99 (1.75)	0.68 (1.17)	0.62 (1.04)	1.55 (5.01)	1.55 (3.41)	1.55 (6.71)	1.55 (7.10)
最小通信発生間隔	0.12	0.24	0.10	0.07	0.08	0.17	0.07	0.06
最大通信発生間隔	48.49	32.93	18.73	13.34	75.77	52.00	104.99	100.31
通信回数（回）	8028	4635	6741	7392	2483	2483	2483	2483

表 5.3　オセロゲーム(5 手読み)の実行結果詳細（単位 msec.)

	動的				静的			
	300Mhz	166Mhz	266Mhz	450Mhz	300Mhz	166Mhz	266Mhz	450Mhz
処理LWP数	2060	1144	1978	2818	1950	1950	1950	1950
平均LWP処理時間	5.45 (15.9)	9.11 (30.44)	5.25 (15.53)	3.99 (12.46)	- -	- -	- -	- -
最小LWP処理時間	0.25	0.53	0.22	0.17	-	-	-	-
最大LWP処理時間	150.59	232.02	140.31	171.55	-	-	-	-
平均LWP割り当て待ち時間	0.53 (0.78)	1.20 (1.85)	0.97 (1.87)	0.79 (1.29)	- -	- -	- -	- -
最小LWP割り当て待ち時間	0.29	0.52	0.31	0.27	-	-	-	-
最大LWP割り当て待ち時間	6.30	24.00	33.26	22.64	-	-	-	-
平均通信発生間隔	1.84 (9.06)	3.14 (17.16)	1.97 (9.01)	1.53 (7.23)	9.84 (53.44)	9.84 (33.95)	9.84 (59.04)	9.84 (66.21)
最小通信発生間隔	0.12	0.24	0.10	0.08	0.08	0.17	0.07	0.06
最大通信発生間隔	148.53	231.69	139.32	170.87	986.55	290.80	1108.79	1135.54
通信回数（回）	6921	4147	6675	8595	2443	2443	2443	2443

表 5.4　行列積(300× 300)の実行結果詳細（単位 msec.）

	動的				静的			
	300Mhz	166Mhz	266Mhz	450Mhz	300Mhz	166Mhz	266Mhz	450Mhz
処理LWP数	43	9	106	142	75	75	75	75
平均LWP処理時間	18.16 (1.66)	72.96 (73.48)	6.99 (8.54)	5.14 (14.19)	- -	- -	- -	- -
最小LWP処理時間	15.20	32.92	5.72	3.72	-	-	-	-
最大LWP処理時間	20.62	234.79	93.91	173.04	-	-	-	-
平均LWP割り当て待ち時間	0.38 (0.12)	0.67 (0.20)	0.48 (0.17)	0.47 (0.17)	- -	- -	- -	- -
最小LWP割り当て待ち時間	0.29	0.57	0.38	0.32	-	-	-	-
最大LWP割り当て待ち時間	0.82	1.21	1.30	1.32	-	-	-	-
平均通信発生間隔	4.41 (19.78)	8.24 (39.47)	3.05 (19.42)	2.68 (16.37)	8.66 (24.27)	12.21 (37.57)	11.48 (97.98)	17.23 (187.79)
最小通信発生間隔	0.03	0.10	0.04	0.03	0.03	0.13	0.03	0.03
最大通信発生間隔	236.24	413.37	280.19	190.60	327.05	390.81	1790.40	2921.66
通信回数（回）	444	238	643	732	486	345	367	245

表 5.5　行列積(300× 300)の実行結果詳細（単位 msec.）

	動的				静的			
	300Mhz	166Mhz	266Mhz	450Mhz	300Mhz	166Mhz	266Mhz	450Mhz
処理LWP数	71	22	129	178	100	100	100	100
平均LWP処理時間	35.95 (2.62)	117.96 (80.30)	19.15 (74.35)	8.78 (11.31)	- -	- -	- -	- -
最小LWP処理時間	32.32	92.70	11.25	7.54	-	-	-	-
最大LWP処理時間	39.55	470.03	849.51	156.66	-	-	-	-
平均LWP割り当て待ち時間	2.78 (20.62)	1.65 (4.38)	0.67 (1.75)	0.50 (0.16)	- -	- -	- -	- -
最小LWP割り当て待ち時間	0.29	0.57	0.38	0.29	-	-	-	-
最大LWP割り当て待ち時間	175.33	21.67	19.90	1.55	-	-	-	-
平均通信発生間隔	6.70 (36.40)	12.26 (60.33)	5.63 (46.35)	4.88 (40.72)	18.98 (48.58)	26.70 (65.61)	25.24 (315.32)	38.01 (523.88)
最小通信発生間隔	0.04	0.11	0.04	0.04	0.04	0.15	0.04	0.04
最大通信発生間隔	836.94	1012.18	1040.06	884.11	830.12	994.00	6833.02	9342.81
通信回数（回）	664	361	786	912	641	455	482	320

図5.9は，表5.3～5.6までの平均通信発生間隔をグラフ化したものである．これを見ると，動的割り当てでは静的割り当てに比べて短い間隔で通信が発生していることが分かる．特に，通信頻度が激しいオセロの５手読みの場合には約2msec に一度の間隔で通信が発生しており，これは静的割り当てに比べて約３～４倍の頻度である．また，表5.3～5.6の通信回数を見ると，すべての場合において動的割り当てでは静的割り当てに比べ２倍以上の通信を行っていることも分かる．

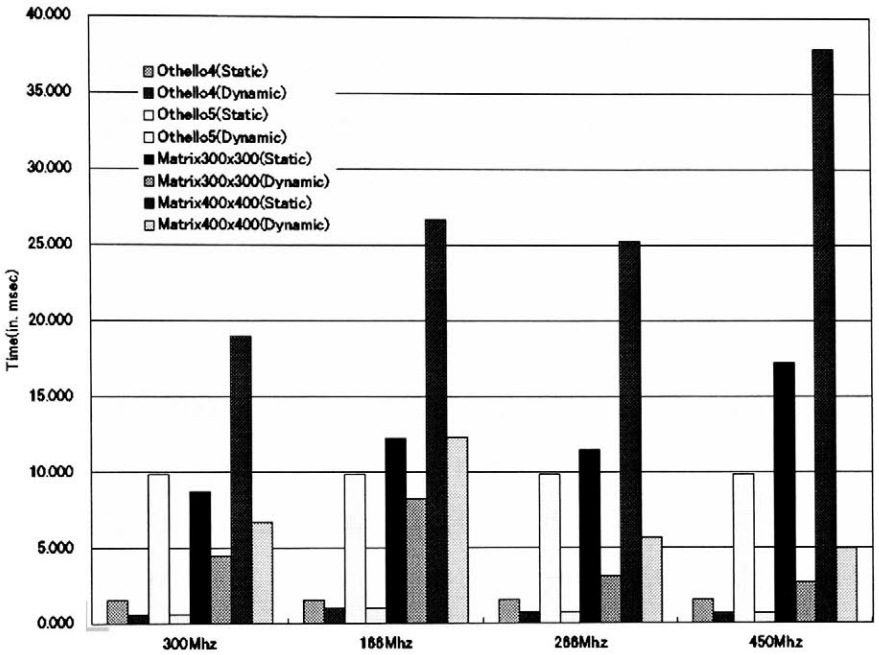


図 5.9　平均通信発生間隔

以上のように，動的割り当てにより通信間隔が短くなること，および通信量が大きく増加することが確認できた．しかしながら，動的割り当てと静的割り当ての実行時間を比べた場合には，動的割り当ての方が短い時間で処理を完了していることから，計算機の性能が不均等であったり，他のユーザにより計算機の負荷が変動するような環境では動的割り当てにより効率の良い実行が可能になることが確認できた．

5.4.3 実験２

２つめの実験では，ホストプロセッサを配置する計算機の性能による割り当て性能への影響を調査するために，ホストプロセッサを300Mhz から166Mhz に変更して実験を

行った．なお，この実験には８クイーンや巡回セールスマン問題などと同じ探索問題である騎士巡回問題（Knight Tour）を利用した．このアプリケーションを選んだ理由は，このアプリケーションはLWPを多数含み，LWP割り当てを頻繁に行うため，ホストプロセッサの割り当てが頻繁に発生し，割り当ての性能の差が出やすいと考えたからである．

図5.10は，Knight Tourの実行結果のグラフである．グラフでは，横軸に利用した計算機を，縦軸に処理に要した時間をとっている．結果より，ホストプロセッサがどちらの場合でも，静的割り当てに比べた場合には短い時間で処理が完了していることが分かる．なお，ホストプロセッサが166Mhzの場合で仮想プロセッサが一台(300Mhz)の場合に静的およびdynamic(300Mhz)に比べて処理時間が長くなっている理由は，dynamic(166Mhz)の場合にはホストプロセッサと仮想プロセッサを実行する計算機が異なるため，同じ計算機上で２つを実行するdynamic(300Mhz)やstaticに比べて通信オーバヘッドが大きくなることが原因である．

dynamic(300Mhz)とdynamic(166Mhz)を比較すると，全体的にdynamic(166Mhz)の方が処理時間が長い．これは，ホストプロセッサの変更の影響が少なからず処理時間に表れていることを示している．これをより詳しく調べるために，より詳細な情報を収集した．表5.7は，実行結果の詳細データである．これを見ると，166Mhzがホストプロセッサの場合は，300Mhzがホストプロセッサの場合に比べて平均LWP処理時間が大きくなっていることが分かる．また，平均LWP処理時間の標準偏差から，LWP処理時間のばらつきも大きくなっていることが確認できる．この原因は，ホストプロセッサが変更されたことで，共有メモリへのデータ配置が変わってしまい，LWPがアクセスするデータが300Mhzから166Mhz上へ移動したことによる，通信処理時間の増加であると考えられる．

LWP割り当て待ち時間の方に注目すると，166Mhzの方が300Mhzに比べて時間が長くなっており，また標準偏差も大きくなっている．つまり，ホストプロセッサでの割り当て処理が遅くなったため，割り当て要求の衝突が頻繁に発生するようになり，処理時間がばらついたと考えられる．

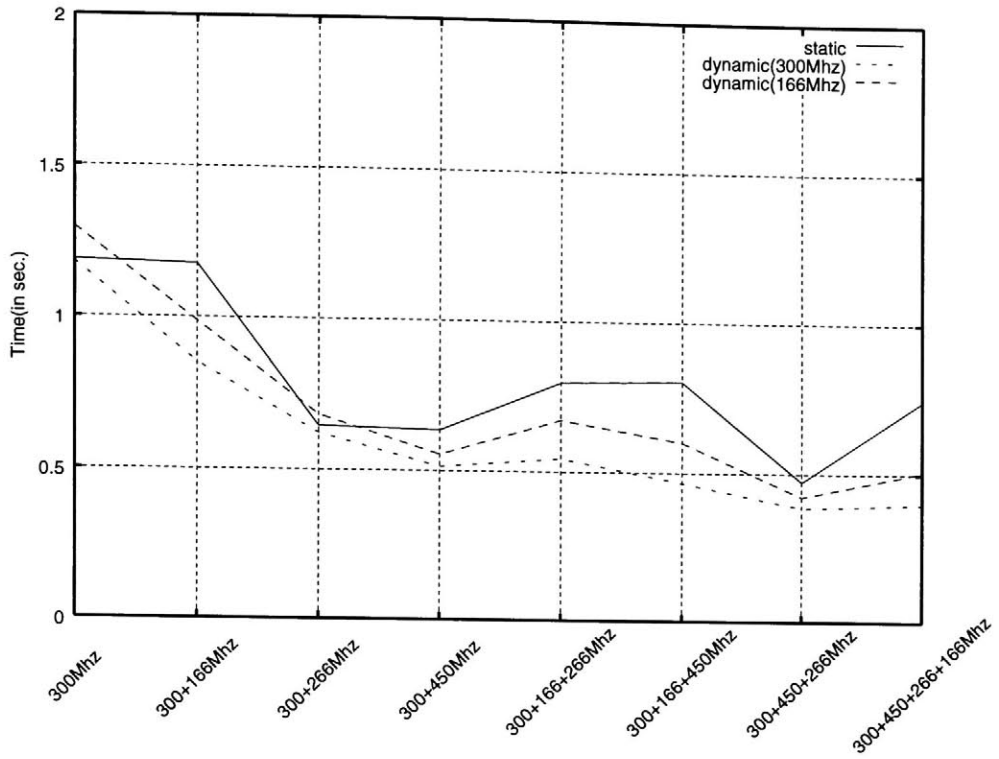


図 5.10　Knight Tourの実行結果

表 5.7　Knight Tourの実行結果詳細（単位 msec.）

	300Mhz				166Mhz			
	300Mhz	166Mhz	266Mhz	450Mhz	300Mhz	166Mhz	266Mhz	450Mhz
処理LWP数	12	11	20	37	11	10	21	38
平均LWP処理時間	26.46 (18.73)	24.93 (16.99)	13.58 (9.34)	7.26 (5.05)	34.41 (44.92)	39.37 (30.77)	20.78 (20.50)	8.91 (8.49)
最小LWP処理時間	6.45	11.58	0.80	1.45	6.16	17.72	0.11	1.72
最大LWP処理時間	59.75	66.94	29.29	17.69	165.40	96.72	98.05	41.13
平均LWP割り当て待ち時間	0.32 (0.06)	0.61 (0.09)	0.36 (0.06)	0.31 (0.05)	0.65 (0.10)	1.65 (0.89)	1.72 (3.27)	1.45 (2.96)
最小LWP割り当て待ち時間	0.28	0.53	0.32	0.27	0.55	0.85	0.54	0.49
最大LWP割り当て待ち時間	0.46	0.79	0.59	0.52	0.84	3.80	14.34	16.01

今回の実験では共有メモリ配置の変化による影響も含まれたため，実行結果がそのまま割り当ての変化によるものであるとはいえない．しかしながら，LWP割り当て時間を見るとホストプロセッサの割り当て処理速度の差による影響が少なくないことが確認できた．

5.5 まとめ

本章では，ライブラリによる動的なLWP割り当て機構について説明し，この機構の性能評価の結果を示した．評価実験の結果，DSE の実行環境として想定される処理速度が不均一な計算機環境下で効果的な割り当てが行われることが確認できた．さらに，動的割り当てに必要な変数を保持するホストプロセッサの変更が，実行速度に影響することから，ホストプロセッサの速度の差が無視できないことが確認できた．

このように，実装した動的割り当て機構は効果的ではあるが，ホストプロセッサの性能による影響が予想以上に大きいことがわかった．これらの問題は，ライブラリによる割り当て変数を持つ計算機の自動選択や，いくつかのプロセッサをまとめたプロセッサグループに一括してLWPを割り当てる階層的な割り当て機構の追加などにより，割り当て変数へのアクセスの分散を行うことで解決できる．

第6章

PPELibにおける共有メモリキャッシング機構

6.1 概要

PPElibでは，現在普及しているイーサネット(10Mbps, 100Mbps) 程度のLAN環境で十分な並列実行性能が達成できるように，共有メモリアクセスの方式としてread/write関数により共有メモリの内容を一度プロセッサローカルなメモリに転送して処理を行い，処理後に再び共有メモリへ書き戻すという，I/Oインタフェース方式を用いている．この方法では，ユーザがプログラム上で共有メモリのアクセスを一括して行うことができるため，分散環境での実行性能を上げやすいという利点がある．しかしながら，この方法はユーザに共有メモリアクセスの効率化および共有メモリからローカルメモリへのデータコピーの管理を強いるため，ユーザのプログラミングの利便性を考慮した場合は，共有メモリに対してローカルメモリと同様にアクセスできるメモリR/W方式に比べ劣っている．しかしながら，このようなR/W方式による効率の良い共有メモリアクセスを実現するには，ネットワークの遅延を隠蔽するキャッシュ機構が必須となる．そこで，本研究では，PPElibにキャッシング機構を組み込むことで，R/W方式による効率の良い共有メモリアクセスの実現を試みた．

本章では，メモリR/W方式による共有メモリアクセスの高速化を目的としてPPElibに実装したソフトウェアキャッシュ機構について説明する．このような，ソフトウェアでキャッシュを実現したものとしては，TreadMark[16][17][18][19] やLemuria[38] などがある．これらでは，共有メモリをページまたはセグメントと呼ばれる連続したメモリブ

ロックを単位としてキャッシングを行っている．また，クラスタ環境でDSMシステムを実現する多くのシステムでは，一貫性保証のためにLazy Releasse Consistency(LRC)　モデル[20]などに基づくキャッシュコヒーレントプロトコルを実装している[34][35][36][37]．

しかし，このようなキャッシュ制御に利用されるページまたはセグメントという単位は，実行するアプリケーションのメモリアクセスの特徴とは関係が少ない．このため，アプリケーションによっては，無駄なデータが大量にキャッシングされてしまい，効率の良いキャッシングが行われない場合がある．例えば，画像の垂直方向へのフィルタリング処理などは，このようなアプリケーションの典型である．

そこで，実装した共有メモリのキャッシング機構では，PPElib で記述されたプログラムの性質を利用してデータのアクセスパターンの履歴を取り，これを利用してキャッシングする領域を決定することで，アプリケーションに合わせたキャッシングを行う．また，この方式では，キャッシング領域として連続したアドレス空間だけでは無く，一定ステップ毎に数バイトずつキャッシュするといったことが可能なことが特徴の１つである．

本節では，分散共有メモリでのコヒーレンス制御の方式の１つである RC(Release Consistency) モデルについて簡単に説明する．続いて，PPElib に実装した適応バッファリングについて説明すると共に，評価実験の結果について述べる．

6.2 キャッシュモデル

一般的に，分散共有メモリのキャッシングでは，キャッシュのミスヒットなどが発生する度に数多くのメッセージを交換する必要ある．特に，共有メモリへの書き込みの場合には，一貫性を保つために他のプロセッサでキャッシングされているデータの無効化や更新を行う必要があり，このため多数のプロセッサ間でメッセージを交換しなければならない．

また，他プロセッサが読み出すデータと異なるデータを書き込んだ場合でも，同じページ（キャッシュライン）内のデータであればキャッシングされたページが無効化または更新する必要があるため，無効化であれば再び共有メモリからの読み出しが，更新であれば更新のための通信が必要になる．この例の場合は，読み出しと書き込みが異なるアドレスであるため，本来はキャッシュの無効化や更新は必要ないにも関わらず処理が行われていることになる．このように実際にはデータの共有化は行われていないにも

関わらずキャッシュのバッファ単位によりあたかも共有されているように処理されることをfalse-sharing と呼ぶ. もし, このfalse-sharing による無駄な処理を減らすことができれば, コヒーレンス処理のためのメッセージ量を削減することができる. これが, Sequential Consistency やRelease Consistency といったRelaxed Consistency モデルの基本的な考え方である.

6.2.1 RC モデル

Release Consistency モデルは, 同期操作を獲得(aquire) と解放(release) に分け, 同期操作の性質を考えて共有メモリアクセスの順序制約を緩和するものである. 例えば, 2 つのプロセッサP1 とP2 が存在し, 図6.1 の順序で処理が行われた場合を考える. なお, release とaquire のペアは矢印の方向に同期が成立している.

ここで, (1), (3) の書き込み(write) と (4) の読み出し(read) はの順序は, 同期操作により制御されている. したがって, (4) の読み出しでは, (3) で書き込まれたデータが読み出されることになる. 一方, (2) の読み出しは順序が決まっていないため(1), (3) のどちらの結果を読み出すかは決まらない. このため, (2) では(1) または(3) の書き込みのどちらを読み出すか分からない. しかし, 同期操作が行われていないことを考えると, (2) の読み出しではどちらのデータを読み出しても意味上は問題がないはずである.

つまり, (1), (3) の書き込みはrelease が行われる前までに完了していれば良いことになる. このような性質を利用して, キャッシュの更新操作をrelease 時まで遅らせることができる. さらに, 同期処理をaquire とrelease のペアに分離した場合, P1 がrelease した後でも, 対応するaquire を行っていないプロセッサでは書き込み結果の反映を行う必要はない.

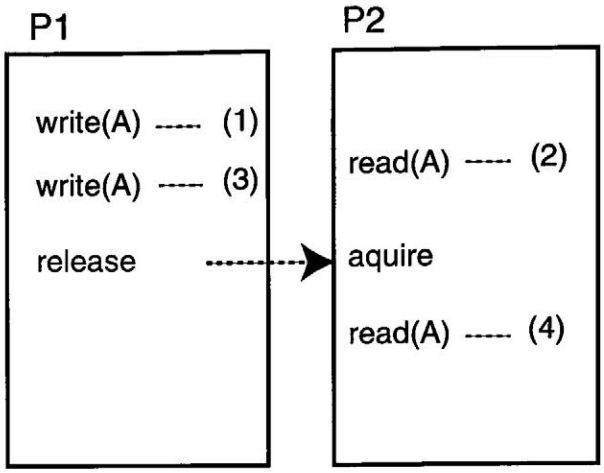


図 6.1　RCモデルの例

以上のように同期操作を遅らせることで, false-sharing が生じている場合のキャッシュの無駄な更新処理を削減でき, これによりコヒーレント処理のオーバヘッドを削減することが, 上記のようなRelaxed Consistency　モデルの目的である.

RC モデルでは, 同期処理をrelase とaquire に分類し, 書き込み操作とrelease , aquire と読み出し操作の順序関係を分けて考えることで, false-sharing による無駄なキャッシュの更新操作による通信の増加を緩和する.

6.2.2 PPElib のバッファリング制御機構

PPElib では, タスククラスがLWP の割り当てと実行, およびLWP の同期操作を行っている. タスクは, タスクの開始(各プロセッサへのLWP の割り当て開始)と, タスク終了(すべてのLWPの実行が完了したときの同期処理)時に, 内部で同期処理を行っている. PPElib ではLWPは互いにデータ依存関係のない処理に限定されているため, この同期処理をそれぞれaquire , release と考えると, タスクの実行に関してRCモデルに基づいたメモリコンシステンシモデルが適応できる. また, LWP内で排他的制御を行うために用意されたSynchronized メソッドも, タスククラス内部ではユーザの指定したメソッドが呼び出される前後で同期処理が挿入されているので, これについてもRC モデルに基づいたメモリコンシステンシモデルが適応できる.

以上により, キャッシュのコヒーレント制御は, タスクの終了時とSynchronized メソッドの前後で適切に行えばよいことになる. ここでは, このバッファ制御としてキャッ

シュ全体のフラッシュを用いた．このフラッシュ操作は，各プロセッサ上で独立して行うことができるため，無効化(invalidate) や更新(update) を行う場合と比べて非常に少ないプロセッサ間の通信で実現することができる．通常，キャッシュ全体のフラッシュを行うと，同じデータを以降のプログラムで利用する場合に再び共有メモリへアクセスしなければならないため，無駄な通信が生じてしまう．しかしながら，PPElib ではタスクの終了は1つの意味のあるプログラム実行の終了であるため，ここからメモリアクセスパターンが変化することも多く，フラッシュによる影響は小さい．また，以降で述べる適応バッファリングとの組み合わせにより総バッファサイズも小さく抑えられているため，フラッシュによる影響はさらに小さくなっている．

図6.2は，PPElib で記述されたプログラムのどの位置でキャッシュのフラッシュ行われるかを示したものである．PPElib では，キャッシュのフラッシュは図6.2 に示すように，プロセッサへのLWP の割り当て前と，タスクの終了時に挿入した．また，LWP 内のSynchronized メソッドでは，メソッドが呼び出された時に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の挿入により，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無く，比較的軽い処理でバッファ制御が実現可能となった．

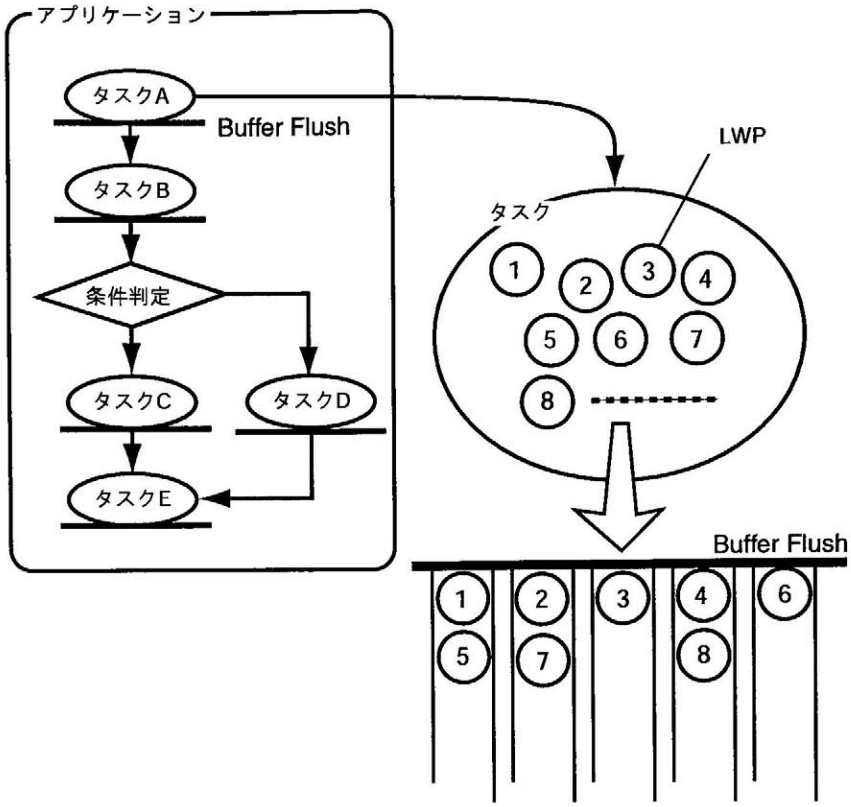


図 6.2　バッファフラッシュ挿入部

6.3 適応バッファリング

PPElib では, RC モデルに基づくキャッシュの制御に加え, キャッシュする領域を動的に変化させていくという適応バッファリング機構を実装した. 適応バッファリングは, タスクが意味のある 1 まとまりの処理であり, それに含まれるLWPの処理は似ているというPPElib の特徴を利用して効率の良いキャッシングを行う方式である.

例えば, 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. 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 を用いて作成する場合, DCT, Q, IQ, IDCT といった処理をタスククラスを継承して作成し, アプリケーションクラスでこれらのタスクを逐次実行することになる. このとき, 各タスクの最小処理単位をMB とすれば, LWP は 1 つのMB に対する処理となる. このようにプログラミングすると, タスク内の各LWP の処理はほとんど同じになり, 各LWP の処理の差異は, 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 の処

理の差異は，処理するデータの先頭位置（先頭アドレス）の違いだけとなる．

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[48][49]，ウエーブレット変換[47] 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×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，各プロセッサには何個かのLWPが割り当てられ実行されることになる．したがって，プロセッサで実行したLWPの共有メモリのアクセスパターンを収集しておき，これを利用して次に実行されるLWPのアクセスパターンを予測し，キャッシングを行えば，バッファのヒット率を向上させることが可能となる．バッファのヒット率が向上すれば，結果として共有メモリへのアクセス回数を削減でき，これにより実行速度の向上が望める．

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法であり，これにより少ない総バッファサイズで効率の良いバッファリングと，共有メモリへのアクセス回数の削減を図るものである．なお，このようにアクセスパターンに合わせてバッファリングを変更するものとしてはHuら[19]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替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，アクセスパターンの指定をプログラム中に記述する．一方，本方式では，このアクセスパターンの指定を予測処理に基づいて自動的の行うため，ユーザがパターンを指定する必要はない．

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．

6.3.1 DSEへのプリミティブの追加

先に述べたように，画像処理では，水平，垂直ラインやN×Mの矩形ブロックに対して処理が行われることが多い．したがって，画像処理に対するアクセス領域を予測して一括してキャッシングする場合には，メモリから上記のような領域のデータを読み出す必要がある．しかしながら，通常の共有メモリ読み出し(read) では，連続したメモリ領域

を読み出すことしかできないため，例えばある画像領域を読み出そうとした場合には，各ライン毎に読み出しを行うか，または，必要のない領域を含めて一括して読み出すかを行わなければならない．前者の場合は共有メモリアクセスが複数回行われるためメッセージ数が増加してしまうし，後者の場合は無駄な領域の転送を含むためデータ転送量の増加に伴い通信時間が長くなってしまう．そこで，このような矩形領域を無駄なく転送できるように，DSE側に矩形領域を一括読み出しするためのプリミティブを追加した．

図6.3は，追加したプリミティブで読み出せる共有メモリの領域の例を示したものである．図のように，追加したプリミティブでは，メモリの先頭アドレス(start)，1ラインの長さ(rowsize)，次の読み出し開始位置までのステップ(step)，読み出す長さ(len)をパラメータとして与えることで，任意の矩形領域の読み出しを可能にする．例えば，start = 0, len = 64 , rowsize = 8 , step = 640 と指定すれば，640 ×480 ピクセルの画像の座標(0,0)から8 ×8 ピクセルの矩形領域を一度の共有メモリ読み出しで行うことが可能である．

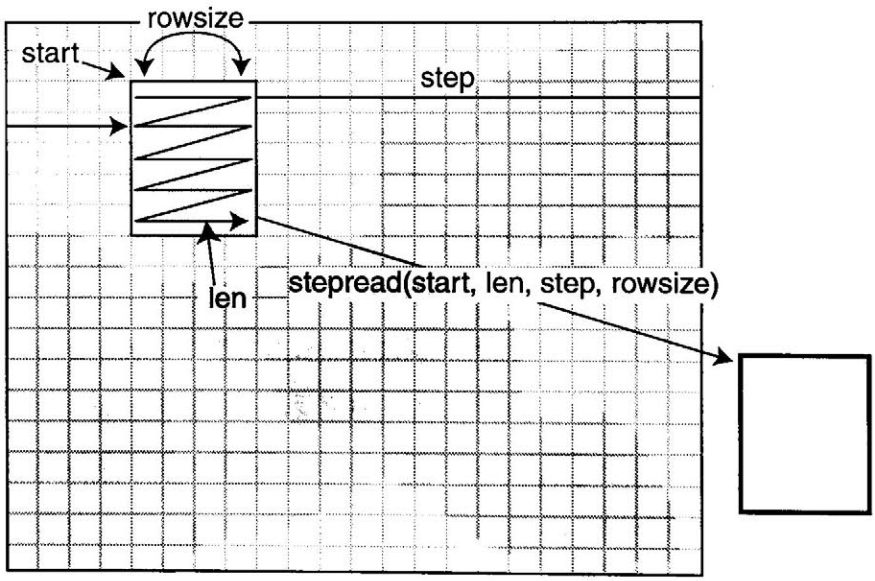


図 6.3　2次元メモリアクセス

適応バッファリングの実装では，効率の良い共有メモリの矩形領域の読み出しを実現するためにstepread プリミティブをDSE へ実装した．

6.3.2 アクセスパターン収集処理

適応バッファリングでは，実行中のLWPの共有メモリへのアクセスパターンを収集し，このログをもとにして次のLWPのメモリアクセスを予測する．この予測結果から共有メモリ中の矩形エリアを読み込みためには，先ほどのstepread プリミティブと同様にstart，len，step，rowsize の情報が必要になる．しかしながら，一度に読み出すサイズをキャッシュのラインサイズに固定した場合には，len の情報は必要ない．したがって，アクセス予測の結果として必要なパラメータは，開始アドレス(start)，間隔(step)，連続読み出しサイズ(rowsize) の３つとなる．

そこで，予測のためのログ収集では，これらのパラメータを予測結果として得るために，アクセス間隔を示すSTEP と，rowsize に対応するデータ長LENと，あるパターンに一致したアクセスが何回行われたかを記録する．なお，このアクセス回数としては，パターンにマッチしたアクセスを行った回数(COUNTER A) と，プログラム中での読み出し発生回数(COUNTER B) の２つを個別に用意し，それぞれカウントした．これらのカウンタの違いについては後で詳しく説明するが，２つカウンタを用意した理由は，ログ生成時に近傍へのアクセスを１つのアクセスとしてまとめる処理によりパターンにマッチしたアクセスの回数と実際のプログラム中での読み出し回数が異なってくるためである．例えば，２バイト間隔に４回アクセスした場合は，このまとめる処理により８バイトのアクセスを行ったとして１つのパターンにまとめられる．この場合は，COUNTER A は1に，COUNTER B は4になる．

表6.1 は，以上で説明したログ収集される情報をまとめたものである．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１つ前のアクセスの先頭アドレスを示している．また，以降の説明では，PLen を１つ前のアクセスデータ長，PEA を１つ前のアクセスの末尾アドレスとする．

表 6.1 ログ情報

項目名	説明
STEP	直前のアクセス位置からの相対的なアドレス STEP = SA -PSa
LEN	データ長 LEN = EA - SA
COUNTER A	このパターンに一致した回数
COUNTER B	プログラムからの読み込み(READ命令発行)回数

画像のフィルタリング処理などでは，ある画素を中心として上下左右の画素に交互にアクセスするケースがある．この場合，単純にSTEP とLENだけを記録するだけでは，実際には連続したアドレスへのアクセスであるにも関わらず，離れたアドレスへのアクセスとして分割して記録されてしまう．例えば，x[-1]+x[0]+ x[1]とx[0]+x[-1]+x[1]は，同じ連続領域にアクセスするにも関わらず，単純にログ収集を行った場合には前者は連続アクセスとして，後者は３つのアクセスとして記録されてしまう．

そこで，ログの収集にあたり，直前のアクセスと現在のアクセス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つのアクセスをまとめる処理を行った．

$$SA - PSa + PLen \leq \delta$$
$$SA - Len + PSa \leq \delta$$

この条件により，直前のアクセスと今回のアクセスの間で前後にδだけ離れている場合でも１つのアクセスとしてまとめられる．なお，実装ではδ =64 とした．

以上が，アクセスパターン収集の説明である．次節では，収集したアクセスログを利用した予測処理について説明する．

6.3.3 予測処理

適応バッファリングの予測処理では,LWPがアクセスしたメモリ範囲が以下の条件を満たす場合には，ログを利用した予測処理は行わず，通常の連続領域に対するキャッシングが行われる．なお，式中のTSa はLWP がアクセスした最小アドレス，TLa は最大アドレス，BufSize はキャッシュのバッファのサイズである．

$$TLa - Tsa \leq BufSize$$

この式は,LWPがアクセスしたメモリの範囲がキャッシュのサイズ以下であることを意味しており，この場合はアクセスされた領域全体がすべてバッファに入りきるため，適応バッファリングを行う必要はない．したがって，実際の予測処理はこの条件が成立しない場合に限って行われる．

予測処理の最初のステップでは，全てのログエントリ (ei)から以下の条件を満たすものを検索する．なお，式中のCbi はエントリ(ei)のCOUNTER B を，Cai はCOUNTER A を示している．

$$E1 = \{ei : Cbi, \{Cbi\} = \max\{Cbi, \ldots, Cbn\}\}$$

上の条件により，共有メモリの読み込みが最も多く行われたパターンが選択されることになる．続いて，上記の条件により選択されたエントリの中から以下の条件を満たすエントリを検索する．

$$E2 = \{ei : ei \in E1, Cai = \max\{Cai, \ldots, Can\}\}$$

上の条件は，E1 の中からさらに，アクセスパターンに一致する回数が最も多いものを選択する操作になる．以上により，LWP が読み込み命令を発行した回数が最も多く，最も多くの回数マッチしたパターンが選択される．なお，E2 に含まれるエントリが複数ある場合は，先に見つかったエントリを選択する．

6.3.4 予測ミスのペナルティ削減

適応バッファリングは，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の良いバッファリングを行うことができるよう設計している．しかし，ターゲット以外のアプリケーションでは，適応バッファリングにより逆にヒット率が低下するケースが考えられる．そこで，適応バッファリングでの予測ミスによるペナルティを小さくするために，予測ミスが発生した場合には通常の連続領域のキャッシングに切り替える機構を実装した．

図6.4は，適応バッファリングと通常のバッファリングの切り替えを行うための状態遷移を示したものである．図6.4 中のUP，DN，EQ は，前回のLWP 実行時のキャッシュヒット率と現在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)を表している．タスクが実行された直後はStatus1 の状態であり，以降LWPの実行毎にヒット率の比較を行い，それに応じてStatus 間の移動が行われる．この結果，Status 0 およびStatus 1 の状態である場合には通常の連続領域に対するバッファリング(Liner buffering) が，Status 2 およびStatus 3 の状態である場合には適応バッファリング(Adaptive buffering) が行われる．なお，隣合った状態間でしか状態遷移が行われないため，Status0 またはStatus3 の状態の場合には，次にキャッシュのミスヒットが多発した場合でもバッファリング方式が切り替えられないことになる．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切り替えが多発する状態を防いでいる．

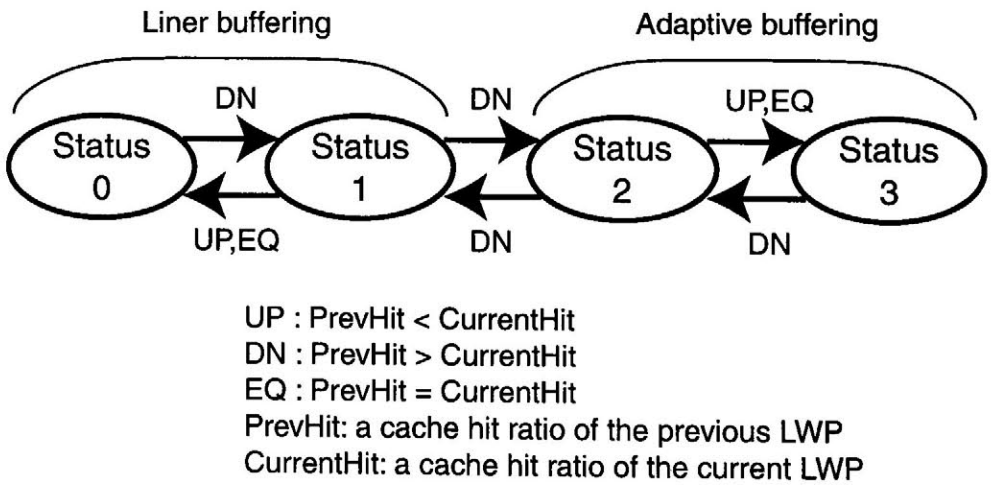


図 6.4　バッファリング方式切り替え

6.3.5 PPElib への実装

PPElib への適応バッファリングの実装は, 以下のような処理をタスククラスと共有メモリクラスに挿入することにより行った.

・共有メモリクラスでのread 処理

共有メモリクラスのread 処理では, 読み出されたデータがキャッシュ内に存在するかどうかが判定され, 存在しない場合は指定されたバッファリング方式に従って共有メモリからのデータ読み出しが行われる. また, 共有メモリ読み出しのログ収集を同時に行う.

・共有メモリクラスでの write 処理

共有メモリクラスのwrite処理では, 書き込み予約バッファに書き込まれたデータが保存される. なお, 予約バッファがいっぱいになるか, 同期の解放 (release) 発生時には, 予約バッファの内容を更新するために共有メモリに対して実際の書き込みがおこなわれる. このように, write 処理では, RC モデルに基づき共有メモリへの実際の書き込みはrelease 発生まで遅らされる.

・タスククラスの LWP 割り当て処理

タスククラスでのLWP 割り当て処理では, LWP の実行の前に収集されたログを使った予測処理が行われ, バッファリング方式と領域を示すパラメータが決定される.

・タスククラスの終了同期処理

タスククラスの終了同期処理では, 同期の解放処理が行われる.

・タスククラスの LWP 実行前処理

タスククラスの LWP 実行前処理では同期の獲得(aquire) 処理が行われ, キャッシュの内容がフラッシュされる.

以上の処理をPPElib に実装することで, RC モデルに基づいたキャッシュのコヒーレント制御と適応バッファリングを実装した.

6.4 評価実験

適応バッファリングの評価実験では，適応バッファリングのターゲットである画像処理アプリケーションとしてウェーブレット変換を行うアプリケーションと，適応バッファリングが有効に機能しないアプリケーションとしてKnight Tour を用意し，実験を行った．以下，実験環境および２つの評価実験の結果について述べる．

6.4.1 実験環境

適応バッファリングの実験では，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 450Mhz, 128Mbyte, FreeBSD)　５台を利用した．なお，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ホストプロセッサであり，このPC はタスクの実行(並列実行)には参加しない．したがって，並列実行は最大で4 台のPC で行われる．

なお，この環境で共有メモリの読み込み処理を行った場合，１バイトのデータの読み込みで270usec ，2キロバイトのデータの場合で560usec であった．

6.4.2 ウェーブレット変換

実験の結果の前にウェーブレット変換を行うアプリケーションの処理内容について説明を行う．図6.5 は，PPElib で作成したウェーブレット変換を行うアプリケーションの構成である．図のように，このアプリケーションは，水平方向の走査を行うタスク(Horizontal Filter) と，垂直方向への走査を行うタスク(Vertical Filter) の２つのタスクから構成される．また，アプリケーションクラスでは，処理する画像のファイルからの読み出し後，水平，垂直の順にタスクを実行し，実行後に変換画像をファイルに保存している．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) を施している．この処理を行うことにより，画像の低周波成分と高周波成分が分離される．この処理を水平／垂直の双方に施した結果の画像が，図6.6 であり，画像の左上にLL成分が，それぞれの部分にはLH，HL，HH 成分が分離される．なお，実験では，640 × 480 ピクセルの画像を利用している．

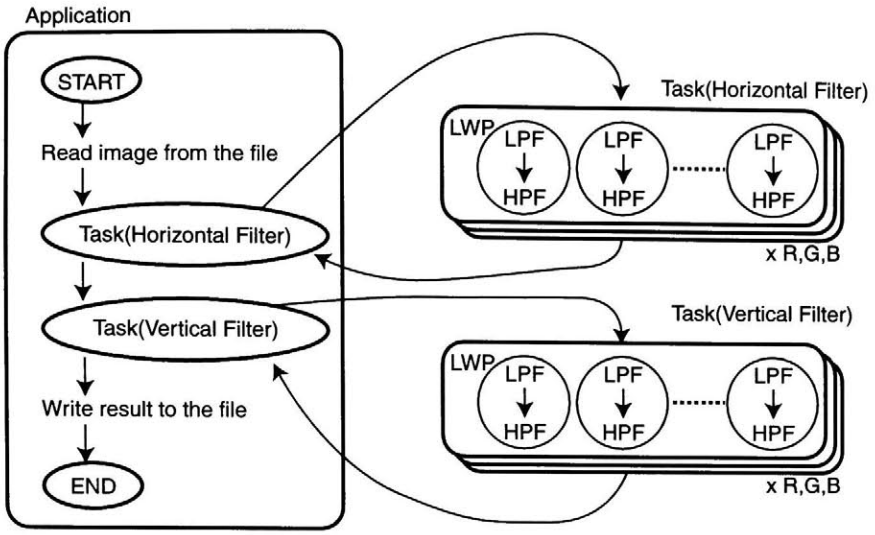


図 6.5　ウェーブレット変換アプリケーションのソフトウェア構成

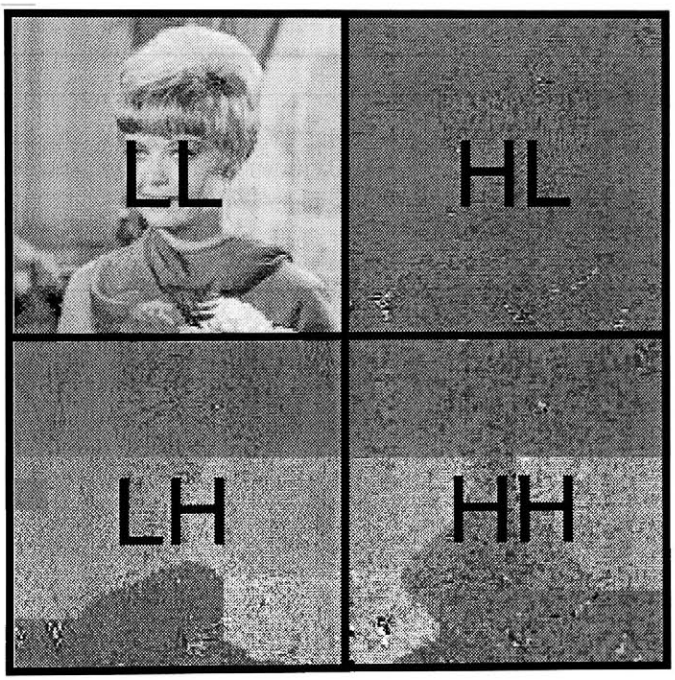


図 6.6　ウェーブレット変換後の画像例

ウェーブレット変換のフィルタ処理

図6.7 は，それぞれのLWP のフィルタ処理の演算内容を示したものである．図のように，ウェーブレット変換では，水平方向には(x,y)を中心としてx±1またはx±2へのアクセスが行われる．また，垂直方法については(x,y)を中心としてy±1またはy±2への

アクセスが行われる．水平方向のアクセスは，横方向へのアクセスであり連続し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も効果的なキャッシングが可能である．一方，垂直方向へのアクセスは，画像サイズが640 × 480 ピクセルの場合で640 ピクセル単位で離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無駄なデータが多く含まれてしまい，効率の良いバッファリングができない．なお，実際のウェーブレット変換アプリケーションでは，ウェーブレット変換処理の前にピクセルデータをdouble の形式に変換するため，640 ピクセルは640 ×sizeof(double) バイト離れたアクセスとなる．

Horizontal Filter

```
LPF     Value = H0[0]*(img[x-2][y] + img[x+2][y]) +
                H0[1]*(img[x-1][y] + img[x+1][y]) +
                H0[2]*img[x][y];
HPF     Value = H1[0]*(img[x-1][y] + img[x+1][y]) +
                H1[0]*img[x][y];
```

Vertical Filter

```
LPF     Value = H0[0]*(img[x][y-2] + img[x][y+2]) +
                H0[1]*(img[x][y-1] + img[x][y+1]) +
                H0[2]*img[x][y];
HPF     Value = H1[0]*(img[x][y-1] + img[x][y+1]) +
                H1[0]*img[x][y];
```

図 6.7　LPF/HPFの演算

以下，このアプリケーションを利用した実験の結果について述べる．

6.4.2.1 実験 1

ウェーブレット変換アプリケーションを用いた１つめの実験では，バッファリングを行わないもの(必要データを常にネットワークを経由して取得するもの)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，適応バッファリング(Adaptive Buffering, AB) を行うものの３つを比較した．なお，これらのバッファリング制御はすべてPPElib 内に組み込んだ形で実装したので，両者のアプリケーション側のプログラムは同一である．プログラムの変更はライブラリ側だけに施している．

図6.8は，ウェーブレット変換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(buffer size)，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(speed-up ratio) であり，1以上であればバッファリングなしに比べて高速であることを意味する．図6.8の結果より，ABはバッファサイズが2Kバイトの時に最も効果的で，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では最大で1.57倍しか速度向上が得られていない．NBで速度向上が得られない理由としては，垂直方向のフィルタリング時にバッファリングが効果的に行われていないことが考えられる．これを確認するために，バッファリングを行う場合のキャッシュのヒット率を調査した．

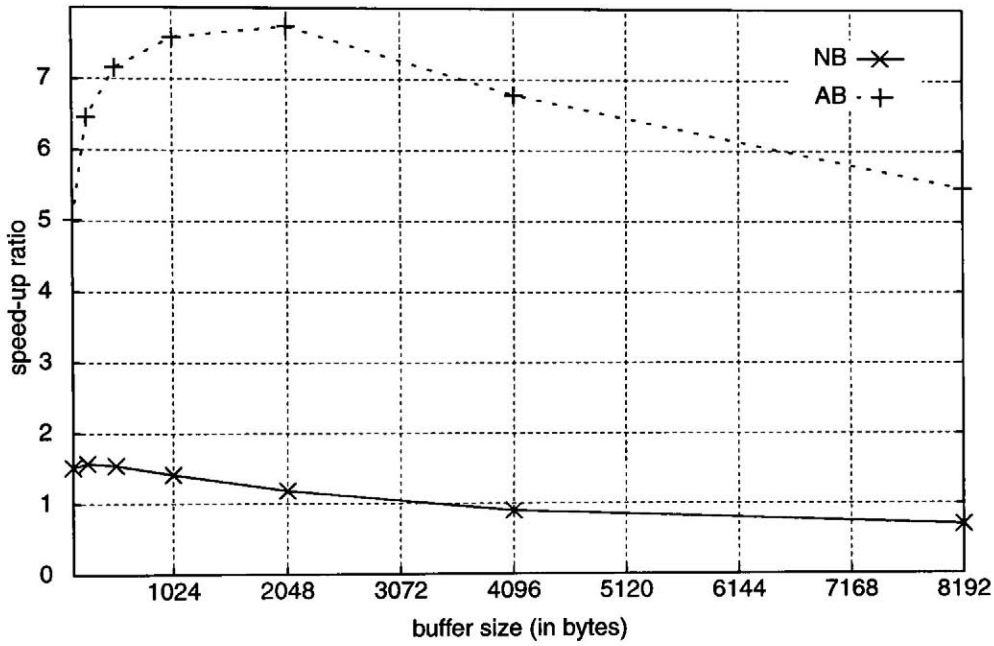


図6.8 ウェーブレット変換アプリケーションの実行結果

図6.9は，バッファリング時のキャッシュのヒット率を示したものである．これから，NBのヒット率はおおよそ58%から68%前後であることが分かる．これは，先に説明したように水平方向へのフィルタリングでは，ほぼ100%のヒット率が得られるのに対し，垂直方向のフィルタリングでは離れたアドレスへのアクセスとなるためにNBではミスヒットが多発してしまい，結果としてヒット率が低くなってしまうからである．これに比べて，ABの場合はバッファサイズが128バイトの場合であっても90%以上のヒット率が実現されている．これは，このアプリケーションではABの矩形バッファリングがうまく機能していることを意味している．

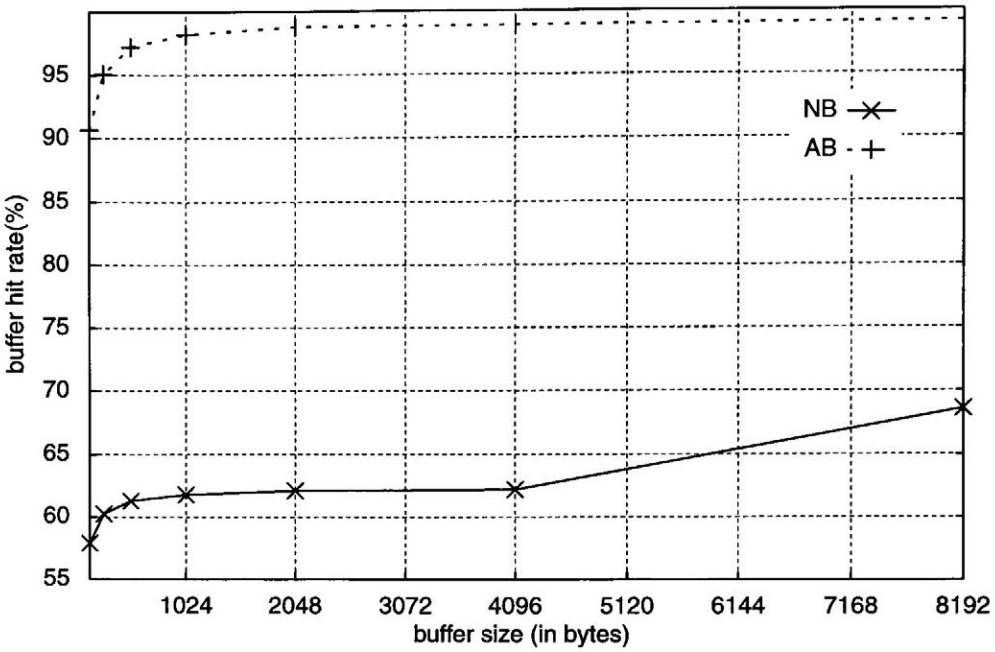


図 6.9 ウェーブレット変換アプリケーションのキャッシュのヒット率

なお，一定以上バッファサイズが大きくなった場合に速度向上が得られなくなる理由は，読み込みサイズが大きくなると読み込み時間が長くなるにも関わらず，無駄なデータ（処理に利用されないデータ）まで読み込んでしまい，結果として無駄な読み込みによるオーバヘッドが増加してしまうからである．

6.4.2.2 実験 2

ウェーブレット変換アプリケーションを用いた 2 つめの実験では，ページベースでキャッシングを行うLazy Release Consistency(LRC)　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の無効化を行う．また，実験では，バッファリングの単位(ページサイズ)は DSE の共有メモリページサイズと同じ 4K バイトとした．なお，LRC の場合はページ数が1024 ページ(RC(1024)) と，256 ページ(RC(256)) のものを用意し，ABは実験 1 で効果の高かった1Kバイト(AB(1K))と2Kバイト(AB(2K))のものを用意し比較を行った．ページサイズが4Kバイトであることより，RC(1024) とRC(256) の総キャッシュサイズはそれぞれ4Mバイトと 1Mバイトであり，ABの場合は総キャッシュサイズは1K と 2K で

ある．なお，LRC のページ置換アルゴリズムにはLRUを採用している．

図6.10 は，実験の結果のグラフである．グラフでは，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), 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以下のとおりである．垂直方向へのフィルタ処理では，各ライン毎に1ピクセルしか処理しないにもかかわらず，キャッシュには4Kバイトの連続したデータが格納されてしまう．RC(1024) の場合は，総バッファサイズが画像全体のサイズよりも大きいので，画像全体がキャッシングできるため，キャッシュのフラッシュは発生しない．このため，全ての画像がキャッシングされ，共有メモリへのアクセスなしに処理を進めることが可能である．しかし，RC(256) の場合は画像全体を納めるだけの十分な総バッファサイズが無いため，バッファが足りなくなり最初に読み出したデータ(画像上部のデータ)がフラッシュされてしまう．このため，頻繁にミスヒットが発生することになり，このキャッシュ更新処理により速度向上が得られない．

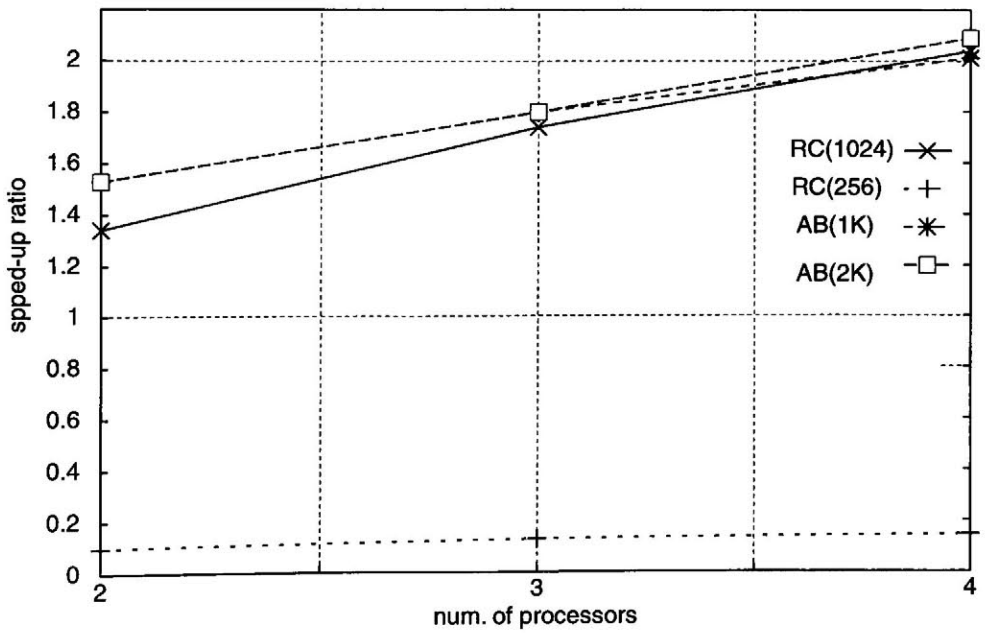


図 6.10 AB 対 RC(ウェーブレット変換アプリケーション)

実験２の結果より，ウェーブレット変換アプリケーションにおいては，ABは1Kバイトまたは2Kバイトという少ない総バッファサイズで，4Mバイトのバッファサイズを持つRC と同等の性能が得られていることが分かる．

6.4.3 巡回騎士問題(Knight Tour)

巡回騎士問題(Knight Tour) のアプリケーションは，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，逐次処理で５手目までの盤面の状態をすべて作成し，それ以降の探索処理を並列に実行する．このアプリケーションでは５手目以降の探索をタスクとし，続きの探索の１つ１つをLWP に割り当てている．したがって，アプリケーションでは５手目まで展開したデータをすべて共有メモリに書き出し，LWPがこれから必要な部分だけを取り出して処理を行っている．共有メモリアクセスパターンを見ると，このプログラムは，通常のバッファリングが有利であり，適応バッファリングの予測機構は効果的に働かない．本実験では，このように適応バッファリングが機能しない場合の評価を行った．なお，実験には実験１と同様にAB(1K)，AB(2K)，RC(1024) ，RC(256) を利用した．

実験結果

図6.11 は，実験の結果のグラフである．この実験の場合，適応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，通常の連続アドレスに対するバッファリングが行われていた．したがって，RC とABの差はバッファサイズの差だけとなる．実験から，RCとABでは若干ABの方が良いという結果が得られた．この理由は，RCの場合のキャッシュメモリと管理部の初期化がライブラリ内部で行われるため，アプリケーション実行時間にこれらの初期化処理時間が含まれてしまうためである．なお，このアプリケーションでは，両者はほぼ変わらない結果となったが，バッファサイズの差がより表に表れるアプリケーションであれば，バッファサイズの大きいRC の方が実行効率が良くなることが考えられる．しかしながら，これはバッファサイズの問題であるので，ABの総バッファサイズ(バッファサイズとバッファ数)をRCと同じ条件にすれば速度は変わらないと考えられる．

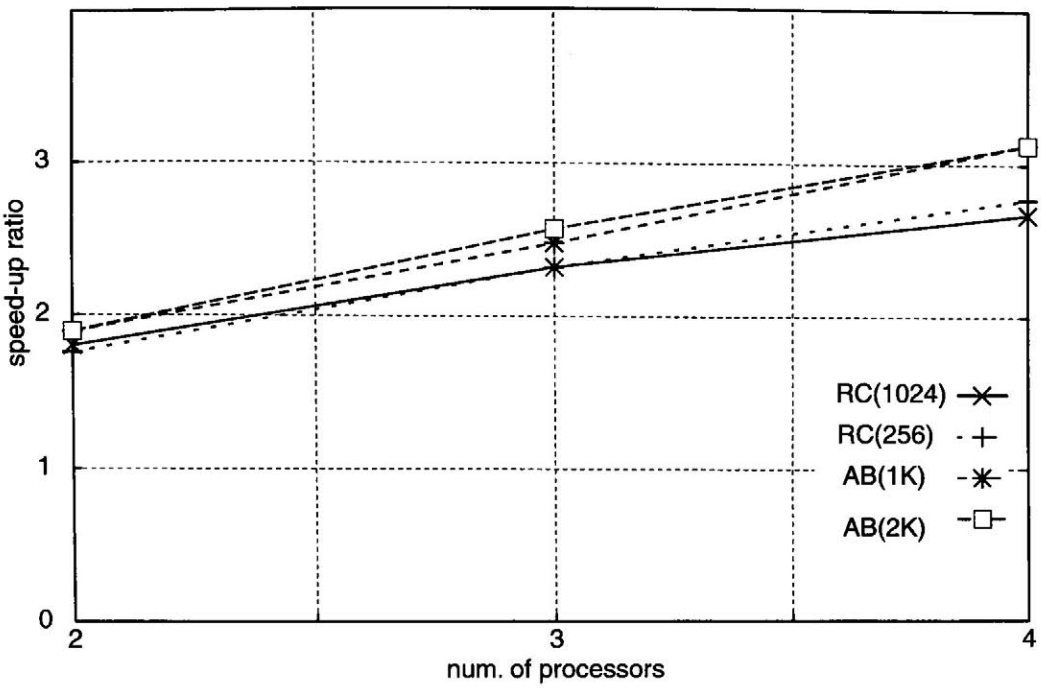


図 6.11 AB 対 RC(Knight Tourアプリケーション)

以上のように，本実験においては，適用バッファリングのペナルティ削減処理がうまく機能し，RC と変わらないパフォーマンスを得られることが確認できた.

6.5 まとめ

本章では，PPElib に組み込んだ適応バッファリングについて説明を行った．適応バッファリングは，PPElib のクラスの特性を利用してLWPの共有メモリアクセスパターンを予測し，予測に合わせたキャッシングをすることで効率のよいキャッシングを実現する．これは，特に画像処理アプリケーションなどのように規則性を持った共有メモリアクセスを行うもの有効であり，また，ライブラリに組み込むことで，ユーザのプログラムの変更なしに自動的なバッファリングを実現する．このように，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されていることが，このキャッシング機構の特徴である．本章では，この適応バッファリングの説明と，評価実験の結果を示した.

評価実験の結果より，ウェーブレット変換などの画像処理アプリケーションにおいては，適応バッファリングは少ないバッファサイズでLRCモデルに基づくページベースの

キャッシング機構と同様の効果が得られることを示した．また，適応バッファリングが効果的に働かない場合でも，LRCと同等のバッファリング性能を維持できることを示した．これらの結果よりライブラリに組み込んだ適応バッファリングの効果が確認できた．

第７章

おわりに

本論文では，分散共有メモリモデルに基づくクラスタコンピューティング環境DSEのDSEの通信処理とプログラム実行時のプロセス割り当て等の制御のの改善について述べた．

通信処理の改善では，通信プロトコルをTCPからUDPに変更し，UDPの上にユーザレベルの独自のプロトコルを実装することで，DSEが必要とするUNIXのポート資源を大幅に削減することができた．また，同一ポートで複数のプロセッサとの通信を実現したことで，接続情報ファイルとよばれる，DSEのプロセッサ要素間の接続形態を記述するファイルをシンプルにすることができ，接続情報ファイル記述の負担を軽減できた．加えて，少ないポート数での通信の実現は，複数ユーザでDSEを利用する場合のポートの衝突などの問題を少なくする．また，評価実験から，UDPによる通信はTCPと同程度のパフォーマンスであることが確認できた．通信処理において，さらにユーザの便利性を向上する手段としては，デーモンプロセス等による自動ポート割り当てなどがある．これまでは，大量のポートを消費するため，デーモンによるポート管理が難しかったが，本研究の改良によりデーモンによるポート管理が容易になる．

２つめのプログラムの実行制御の改善では，オブジェクト指向言語の特徴であるクラス機構のうち，特に「継承」を利用してプログラミングするオブジェクト指向並列プログラミングライブラリPPElibを提案し，これを用いた性能向上を目指した．PPElibでは，あらかじめ用意したクラスを雛形としてプログラムを作成するため，ユーザがライブラリが用意するプログラミングのスタイルから離れたプログラミングを行うことを抑制する．また，プログラミングスタイルを制限することで，ライブラリ側に並列処理特有の処理であるプロセス割り当てや同期処理などを持つことができ，ユーザがこれらを記述する必要がないようにした．これらにより，ユーザの並列プログラムの記述性の向上さ

せると同時に，システム側でアプリケーション実行時の制御を細かく行えるようにした．本論文では，まず，このプログラミングライブラリのプログラムの記述性についてHPC++Libのサンプルプログラムの移植およびいくつかの並列アプリケーションの記述を通して論じた．さらに，並列プログラムの効率の良い実行を目指して実装した，ライブラリによる動的プロセス割り当て機構および，共有メモリのキャッシング機構について述べた．また，評価実験の結果から，これらの機構の効果を確認した．なお，PPElibは，ビジュアルな並列プログラミング環境を視野に入れて設計を行ったライブラリであり，今後の研究としてはPPElibを利用したビジュアルな並列プログラミング環境の構築などが考えられる．このような，環境を構築すればユーザのプログラミング環境をよりいっそう改善できると考えている．

論文および口頭発表

1) 手塚 忠則，末吉 敏則，有田 五次郎 : 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 , *情報処理学会論文誌* , Vol. 41, No. 5, 1470-1479, 2000.

2) T. Tezuka, B. Apduhan, T. Sueyoshi and I. Arita : Evaluation and Analysis of a Dynamic Allocation Scheme on a C++-based Parallel Programming Library, In Proc. of *4th International Conference/Exhibition on High Performance Computing in Aias-Pacific Region*, Vol. II, 915-922, 2000.

3) 手塚 忠則，末吉 敏則，有田 五次郎 : UDP を利用した DSE 向けプロトコルの設計と実装 , *情報処理学会 マルチメディア通信と分散処理ワークショップ予稿集* , 79-84, 1999.

4) 手塚 忠則，末吉 敏則，有田 五次郎 : クラス継承による並列プログラミングライブラリの設計と実装 , *並列/分散/協調処理に関するサマーワークショップ(SWoPP'99)*, 1999.

5) 手塚 忠則，末吉 敏則，有田 五次郎 : 同期操作を隠蔽した並列プログラミングライブラリの実装と評価 , *情報処理学会 プログラミング研究会(PRO)- 並列・分散処理特集 及び一般* -, Jun, 1999.

6) 山本 孝，手塚 忠則，B. Apduhan，有田 五次郎 : Java ベース広域並列処理環境の構築 , *情報処理学会 プログラミング研究会(PRO)- 並列・分散処理特集 及び一般* -, Jun, 1999.

7) 江口 真，手塚 忠則，末吉 敏則，有田 五次郎 : UDP を用いたクラスタコンピューティング向け通信プロトコル設計のための予備実験, *情報処理学会 マルチメディア通信と分散処理ワークショップ予稿集* , 203-207, 1998.

参考論文

8) B. Apduhan, T. Sueyoshi, T. Tezuka, Y. Ohnishi and I. Arita : Reconfigurable Multiprocessor Simulation Environment on a Distributed Procesing System, In Proc. of *6th International Joint*

Workshop on computer communications, 283-290, 1991.

9) B. Apduhan, T. Sueyoshi, Y. Namiuti, T.Tezuka, T. Fujiki and I. Arita : Experiments and Analysis of a Reconfigurable Multiprocessor Simulation on a Distributed Environment, *平成３年度電気関係学会九州支部連合大会論文集*, 1991.

10) 藤木 健士，手塚 忠則，大西 淑雅，B. Apduhan, 末吉 敏則, 有田 五次郎 : 分散処理システムを用いた可変構造型並列計算機シミュレーション環境, *情報処理学会第４３回全国大会*, 1991.

11) B. Apduhan, T. Sueyoshi, Y. Namiuti, T. Tezuka, T. Fujiki and I. Arita : Experiments and Analysis Toward Distributed Supercomputing on a Distributed Workstation Environment, In Proc. of *1991 International Symposium on Supercomputing*, 183-190, 1991 ; or *SUPERCOMPUTER*, vol. VIII, no. 6, 1991.

12) B. Apduhan, T. Sueyoshi, Y. Namiuti, T. Tezuka and I. Arita : Experiments of a Reconfigurable Multiprocessor Simulation on a Distributed Environment : In Proc. of *1992 IEEE International Phenix Conference on Computers and Communications*, 539-546, 1992.

13) B. Apduhan, T. Sueypshi, T. Tezuka and I. Arita : The Effect of Communication Processing in Network Supercomputing Environment, In Proc. of *7th Internatioanal Joint Workshop on Computer Communications*, 373-380, 1992.

14) 了戒 清，手塚 忠則，B. Apduhan，末吉 敏則，有田 五次郎 : 分散スーパコンピューティング環境における通信処理の影響, *平成４年度電気関係学会九州支部大会論文集*, 1992.

15) 手塚 忠則，了戒 清，B. Apduhan，末吉 敏則，有田 五次郎 : 分散処理システムを利用した並列処理環境における通信処理の影響，*情報処理学会第４５回全国大会論文集*, 1992.

16) T. Tezuka, K. Ryokai, B. Apduhan and T. Sueyoshi : Implementation and Evaluation of a Distributed Supercomputing Environment on a Cluster of Workstations, In Proc. of *International Conference on Parallel And Distributed System*, 1992.

17) 手塚 忠則，了戒 清，末吉 敏則 : 分散処理システムを利用した並列処理環境における通信処理の影響, *情報処理学会 マルチメディア通信と分散処理ワークショップ予稿集*, 1993.

18) 池田 淳，手塚 忠則，平島 毅，志水 郁二 : 色空間の変形を用いた色彩調整方式，*テレビジョン学会年次大会予稿集*，1995.

19) 手塚 忠則，池田 淳，平島 毅，井上 由紀子，志水 邦二：簡易操作による色彩調整技術の開発，*映像情報 Industrial*, Vol. 28, 産業開発機構, 10-14, 1996.
20) 米澤 友則，手塚 忠則，九郎丸 俊一，東島 勝義，孝橋 靖雄，西指 真納，松尾　昌俊，藤本 仁：テレビ会議システムにおけるノイズ除去フィルタに関する一検討，*画像符号化シンポジウム予稿集*，電気情報通信学会，77-78, 1997.
21) 手塚 忠則：色空間の変形による色彩調整方式 ～より直感的な色彩調整を可能にする技術の開発～，*画像ラボ*, Vol.9, No.12, 日本工業出版, 1-5, 1998.
22) 手塚 忠則，池田 淳，平島 毅，井上 由紀子，志水 邦二：色空間の変形による色彩調整方式，*情報処理学会論文誌*, Vol. 39, No. 3, 585-592, 1998.

参考文献

[1] Sunderam, V. S.: PVM: A Framework for Parallel Distributed Computing, Concurrency: *Practice and Experience*, vol. 2, no. 4, pp. 315-339, 1990.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88-95, June 1993.

[3] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166-175, April 1993.

[4] PVM ホームページ, *http://www.epm.ornl.gov/pvm/pvm_home.html*

[5] MPI Forum.: MPI: A message-passing interface standard., *International Jounal of Supercomputer Applications and High Performance Computing*, vol. 8, 1994.

[6] HPICH homepage, *http://www.mcs.anl.gov/mpi/mpich/index.html*

[7] Paul Pierce. The NX/2 operating system. In Proceedings of *the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384-390. ACM Press, 1988.

[8] Parasoft Corporation, Pasadena, CA. Express User's Guide, version 3.2.5 edition, 1992.

[9] nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990.

[10] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portableparallel programming and their implementation on the Intel iPSC/2. *Parallel Com-puting*, 15, 1990.

[11] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable pro-gramming with the parmacs message{passing library., *Parallel Computing, Special issue on message passing interfaces*

[12] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991.

[13] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992.

[14] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, *Argonne National Laboratory*, March 1993.

[15] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instrumented communication library. Technical Report TM-11616, *Oak Ridge National Laboratory*, October 1990.

[16] C.Amza , A.L.Cox, S.Dwarkadas , P.Keleher , H.Lu, R.Rajamony , W.Yu and W.Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer*, vol.29, no.2, pp.18-28, 1996.

[17] P. Keleher, A.L. Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of *the Winter94 Usenix Conference*, 115-131, 1994.

[18] H. Lu. Message Passing Versus Distributed Shared Memory on Network of Workstations. In Proc. of *Supercomputing'95*, 1995.

[19] Y.C. Hu, A. Cox and W. Zwaenepoel: Improving Fine-Grained Irregular Shared-Memory Benchmarks by Data Recording , *http://www.cs.rice.edu/~willy/TreadMarks/papers.htm*

[20] K. Gharachorloo, D. Lenoski, J. Laudon, P.Gibboms, A.Gupta and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In Proc. of *the 17th Annual International Symposium on Computer Architecture*, 15-26, 1990.

[21] P. Kelheher, A. L. Cox and W. Zwaenepoel. Lezy release consistency for software distributed shared memor. In Proc. of *the 19th Annual International Symposium on Computer Architecture*, 13-21, 1992.

[22] J.B. Carter, J.K. Bennett and W. Zwaenepoel. Implementation and performance of Munin. In Proc. of *the 13th ACM Symposium on Operationg Systems Princeples*, 152-164, 1991.

[23] M. Matuda, M. Sato and Y. Ishikawa: OBPLib: An Object-Oriented Parallel Library and its Preliminary Performance. *2nd France-Japan Workshop Object Based Parallel and Distributed Computing (OBPDC'97)*, 1997.

[24] Y. Ishikawa: Multi Thread Template Library - MPC++ Version 2.0 Level0 Document - ; RWC Tecchnical Report TR-96012, *http://www.rwcp.or.jp/lab/pdslab/papers*, 1996,

[25] T.J. Willoams, J.V.W. Reynders and W.F. Humphery: POOMA User Guide, *http://www.acl.lang.gov/pooma/doc/userguide*

[26] The HPC++ working group: HPC++ Whitepapers and Draft Working Documents, 1996.

[27] D. Gammon, P.Beckman and E. Johnson: HPC++ Documentation, *http://www.extreme.indiana.edu/hpc++/docs*

[28] http://www.extreme.indiana.edu/hpc++/docs/Examples/thread-examples/

[29] High Performance Fortran Forum: High Performance Fortran language specification, version 1.0, Technical Report CRPC-TR92225, *Center for Research on Parallel Computation*, Rice University, 1992.

[30] J. B Carter, J.K. Bennet, W. Zwaenepoel: Implementation and performance of Munin, In Proc. of *the 13th ACM Symposium on Operation System Principles*, pp. 152-164, 1991.

[31] OpenMP コンソーシアム: OpenMP, *http://www.openmp.org*

[32] Michael Afagan: Quick Java Reference, *Que Corporation*, 1997.

[33] Hirono, S.: HORB: Distributed Execution of Java Programs, Worldwide Computing and Its Applications, *Springer Lecture Notes in Computer Science*, pp. 29-42, 1997.

[34] K. Li, P. Hudak: Memory Coherency in Shared Virtual Memory Systems, *ACM Transaction Computer System*, Vol.7, No.4, pp. 321-359, 1989.

[35] T. Matsumoto , J. Niwa , K. Hiraki: Compiler-Assisted Distributed Shared Memory Scheme Using Memory-Based Communication Facilities, In Proc of *the 1998 PDPTA*, vol. 2, July 1998.

[36] L. Iftode, C. Dubincki, E.W. Felten and K. Li: Improving Release-Consistent Shared Virtual Memory using Automatic Update, In Proc. of *the 2nd HPCA*, Feb. 1996.

[37] 田中 清史, 松本 尚, 平木 敬: 軽いハードウェアによる分散共有メモリ機構, *情報処理学会論文誌*, pp. 2025-2036, vol. 40, no. 5, May 1999.

[38] 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, *電気情報通信学会論文誌 D-I*, Vol.82, No.3, pp. 457-466, 1999.

[39] 村井 純, 楠木 博之 訳: 第２版 TCP/IP によるネットワーク構築 Vol.1, *共立出版株式会社*, pp. 130-138, 1991.

[40] Perloff, M. and Reiss, K.: Improvements to TCP performance in high-speed ATM networks, *Communication of the ACM*, vol. 38, no. 2, pp. 633-641, 1995.

[41] 小口 正人, 新谷 隆彦, 田村 孝之, 喜連川優: ATM結合PC クラスタのTCP 再送機構の解析と並列データマイニングの性能向上, 情報処理学会研究報告DPS87-37, OS77-37, pp.215-220, 1998.

[42] 松田元彦, 石川裕, 佐藤三久: C++ テンプレートを使ったデータ並列ライブラリの効率化手法, *情報処理学会論文誌*, vol. 38, no. 9, 1997.

[43] 湯浅太一, 中村通晃, 中田登志之編: はじめての並列プログラミン, 共立出版, 1998.

[44] 末広謙二，松浦健一郎，菊地賢太郎，村井均，妹尾義樹: 異機種間並列分散システム向けプログラミング環境, *情報処理学会研究報告 99-HPC-77*, pp. 77-82, 1999.

[45] 石川，佐藤，工藤，島田: 分散環境におけるシームレスコンピューティングシステムの構想, *SWOPP-97 (CPSY)*, *電子情報通信学会*, pp. 83-90, 1997.

[46] Y. Ishikawa, M. Sato, T. Kudoh, Y. Akiyama and J. Shinmada, Towards a Seamless Pallel Computing System on Distributed Environments, *SWOPP-97(CPSY), IEICE*, pp. 83-90, 1997.

[47] Charles K. Chui: An Introduction to Wavelets, *ACADEMIC PRESS, INC.*, 1992.

[48] 尾上守男ほか (編): *画像処理ハンドブック*, 昭晃堂(1987).

[49] 応用物理学会光学懇話会: 色の性質と技術, 朝倉書店, 1986.

[50] 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, March 1997.

[51] 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.

[52] B. Apduhan, T. Sueyoshi, T. Tezuka and I. Arita: The Effect of Communication Processing in Network Supercomputing Environment., *Proc. of 7th International Joint Workshop on Computer Communications*, pp.373-380, 1992.

[53] T. Tezuka, K. Ryokai, B. Apduhan and T. Sueyoshi: Implementation and Evaluation of a Distributed Supercomputing Envionment on a Clustor of Workstations, In Proc. of *1992 International Conference on Parallel And Distributed System*, pp. 58-65, Dec. 1992.

[54] 手塚 忠則， 了戒 清, 末吉 敏則: 分散システムを利用した並列処理環境における通信処理の影響, *情報処理学会「マルチメディア通信と分散処理」ワークショップ論文集*, pp. 249-256, 1993.

[55] 江口 真，手塚 忠則, 末吉 敏則，有田 五次郎: UDP を用いたクラスタコンピューティング向け通信プロトコル設計のための予備実験, *情報処理学会「マルチメディア通信と分散処理」ワークショップ論文集*, pp. 203-207, 1998.

[56] 手塚忠則，末吉敏則，有田五次郎: 同期操作を隠蔽した並列プログラミングライブラリの実装と評価, *情報処理学会 プログラミング研究会 1 月発表*

[57] 有田五次郎, 末吉敏則他: FIFO キューを同期手段とする並列プログラム(I)(II)(III) 情報処理学会論文誌, vol. 24, No.2,6, 1983.