平成 21 年度　修士論文

空間的局所性を考慮した
ユーザレベルメモリ管理手法

指導教員　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院 創成シミュレーション工学専攻
平成 20 年度入学 学籍番号 20413525
平成 22 年 2 月 4 日提出

久野 友紀

目 次

第 1 章

はじめに

近年，大規模なメモリ空間を有効に活用できるプラットフォームの必要性が高まりつつある．例えば，遺伝子の探索や画像や動画像を扱うマルチメディア処理，大規模データベースなど，大容量のデータを扱うアプリケーションは大きなメモリ空間が必要となる．また，シミュレーションや数値計算などの科学技術計算においても，前処理あるいは後処理において巨大メモリ空間が要求されることがある．

従来，このような巨大メモリ空間を提供できるコンピュータは共有メモリ型並列コンピュータしかなく，非常に高価であることから限られたユーザしか利用できなかった．しかし，このような大容量メモリを要求するアプリケーションの重要性の高まりを受けて，近年ではパーソナルコンピュータ (以下 PC) 向け 64 ビットマイクロプロセッサが広く普及するようになった．例えば，Intel や AMD の 64 ビットアーキテクチャでは，少なくとも 48 ビット以上の仮想アドレス空間をサポートしており，理論上では 256TB の仮想アドレス空間を扱うことができるということを意味する．

ところが，現代の PC に搭載可能なメモリは高々数 GB から数十 GB であるため，それより大きなメモリ空間を利用しようとすると，オペレーティングシステム (以下 OS) における仮想メモリと呼ばれる機能を使用する必要がある．仮想メモリはハードディスクドライブ (以下 HDD) などの補助記憶装置を用いることで仮想的に大規模なメモリ空間を実現できるが，HDD 等，アクセス速度がメモリに比べて非常に遅い補助記憶装置にメモリの内容を退避するメモリスワッピングを行う場合，性能の低下につながる．

現在，コモディティ[1]ハードウェアを基にしたクラスタ型並列計算機上で巨大なメモリ空間を利用するためには，MPI[1](Message Passing Interface) などを用いて分散メモリ環境で動作する並列プログラムを作成しなければならない．しかし，すべてのアプリケーションが分散メモリモデルに適合するわけではなく，さらに，分散メモリ環境におけるプログラミングは，特にコンピュータサイエンスを専門としない研究者にとって学習コストと実装コストが大きい．

コモディティハードウェアを基にしたクラスタ型並列計算機上で大規模メモリ空間を提供するためのアプローチとして，ソフトウェア分散共有メモリと遠隔メモリスワップシステムがある．ソフトウェア分散共有メモリは，大規模メモリ空間提供と並列実行環境提供の両立を目指している．ノード数の増大により並列性と利用可能なメモリ容量がともに増大するが，実際には共有メモリ領域の同期コストから台数効果は限定的である．結果として，並列性能の台数効果が頭打ちになる台数が利用可能なメモリ容量の最大値となる．従来の研究成果 [2][3] では，アプリケーションにも依存するが，台数効果は高々 8 台から 16 台程度までしかない．

ところで，ネットワークのスループットはメモリバンド幅に近づきつつあり，その差は今後さらに小さくなる可能性がある．そこで，ハードディスクよりスループット性能が高い高速ネットワークで接続された PC クラスタ上で，遠隔 (リモート) マシンのメモリをスワップデバイスとして用いることで，高性能かつ大容量なメモリ空間を実現する遠隔スワップメモリシステムが提案されている [4][5][6][7][8]．しかし，これらのシステムは，高速ネットワークやクラスタ導入にかかる経済面でのコストや，計算機におけるカーネルモジュールの実装コスト，さらにシステム全体構築においてネットワークや OS 等，多分野における知識が必要となり，一般ユーザが利用するには困難を要する．また，ネットワークレイテンシの問題も依然存在するといった状況である．

[1]品質，機能，形状，その他の属性が，標準化の進展，技術の発達，市場の発達，ライフサイクルの成熟化等の理由により安定的に均一化・共通（Common）化して，交換・代替が容易な普遍的（Universal）価値として確立する様.

一方，近年 NAND フラッシュメモリのコストが劇的に低下したことにより，NAND フラッシュメモリを PC のストレージとして使う SSD(Solid State Drive) と呼ばれる補助記憶装置に注目が集まっている [9]．ハードディスクのようにディスクを持たないため，読み取り装置 (ヘッド) をディスク上で移動させる時間 (シークタイム) や，目的のデータがヘッド位置まで回転してくるまでの待ち時間 (サーチタイム) がなく，データに対し高速にランダムアクセスすることができる．よって，今後 1 台の PC で SSD を利用して仮想メモリを元とした大規模メモリ空間利用のが高まる可能性がある．

そこで本研究では，将来的に HDD よりスループット性能が高い SSD を利用した仮想メモリ管理を仮定する．そのための前段階としてまず，仮想メモリにおいて補助記憶装置を利用することありきの問題における，空間的局所性を考慮したメモリ管理手法を提案する．提案手法では，断片的なメモリアクセスパターンに対応した整列データを生成することで連続的なメモリアクセスを可能とする．今回，補助記憶装置は提案手法の有効性をよりはっきり検証するために，HDD をスワップ領域として利用した．そして，行列積の演算処理において，提案手法を適用したユーザレベルメモリ管理手法での演算時間と，LinuxOS のスワップ機構を使用したときの演算時間とで比較し，提案手法の有効性を評価した．

本論文は以下のように構成される．2 章で大規模メモリ空間を実現する関連研究について述べ，3 章で連続メモリアクセスを実現するメモリ管理手法の詳細を示す．4 章で提案手法の OS のスワップ機能と比較することで性能評価と結果に対する考察を示し，最後に 5 章で本論文の成果をまとめる．

第 2 章

大規模メモリ空間を実現する方法と関連研究

本章では，大規模メモリ空間を実現する仮想メモリと関連研究について述べる．

大規模メモリ空間を実現するために，古くから現在でも用いられている技術として仮想メモリがある．補助記憶装置と OS のスワップ機能の組合せにより，物理メモリ容量が多くなったように見せる．本研究では，提案手法の比較対象として Linux OS のスワップ機能を用いて評価を行った．

また，複数の PC をネットワークで接続しクラスタを形成し，クラスタ全体のメモリを利用して仮想的に大容量のメモリ空間を提供する遠隔メモリスワップシステムがある．

2.1 仮想メモリ

仮想メモリは，計算機上に実装されているメモリ容量よりも大きな記憶領域を仮想的に提供する仕組みである．仮想記憶は，図 2–1 のようにメモリ管理ユニット (MMU) と呼ばれるハードウェアによって実現されている．本研究では，提案手法の実現方法として仮想メモリ管理の概念を用いた．

MMU は，CPU の要求するメモリアクセスを処理し，計算機のメモリ空間を管理するための機構である．MMU は情報テーブル TLB(Translation Look-aside Buffer) あるいはページテーブルと呼ばれるアドレス情報を管理するテーブルを持ち，このテーブルによって仮想アドレスを物理アドレスに変換する．本論文では以降，このテーブルをペー

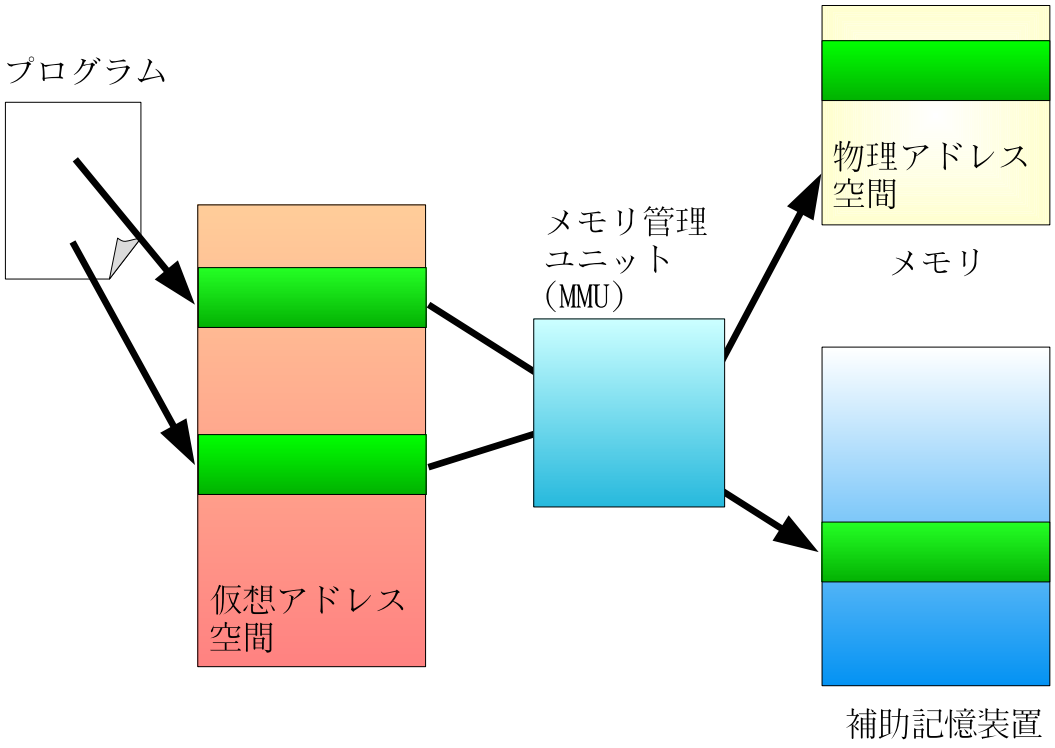

図 2–1　メモリ管理ユニットのアドレス変換機能

ジテーブルと称する．

ページテーブルの一つ一つは「エントリ」と呼ばれ，MMU はいくつかのエントリを持つ．一つのエントリは,「ページ」と呼ばれる固定長のメモリ領域のアドレスを変換対象とする．仮想アドレスは上位ビットのページ番号と下位ビットのオフセットに分けられる．MMU は，このページ番号をページテーブルから検索し，一致する変換情報から物理アドレスを取得し，その物理アドレスにオフセットを加算して実際の物理アドレスに変換する．

仮想記憶の実現には，次の操作が必要となる．

- スワップイン
  実行中のプログラムがある時点で必要となる領域 (プログラム領域とデータ領域) を，補助記憶装置から主記憶上に転送する操作．

- スワップアウト
  スワップイン時に補助記憶装置から主記憶上に転送する領域を確保するために，現在実行中のプログラムで必要としない領域を主記憶から補助記憶に転送する操作．

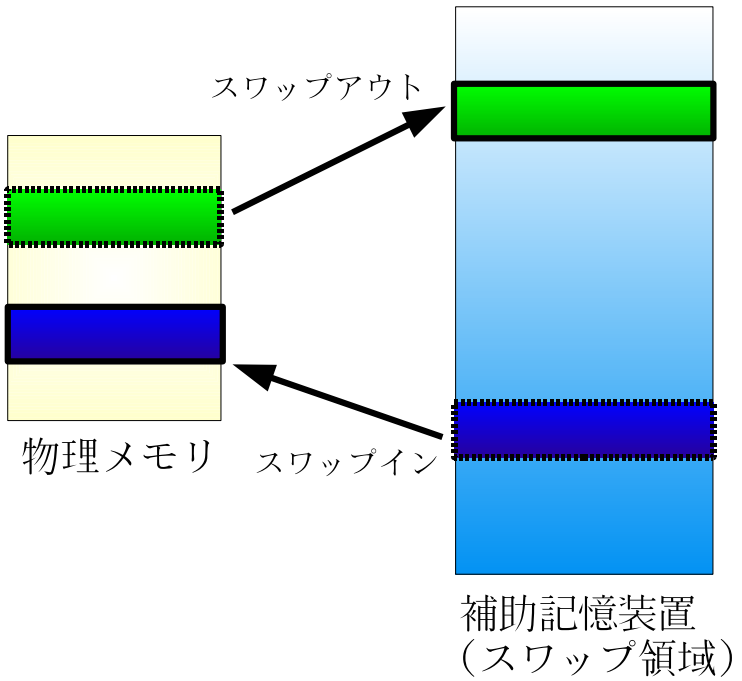

図 2–2　仮想メモリによって実現されたメモリ領域とスワップイン・スワップアウト

大きなプログラムや多数のプログラムを同時に実行する場合にPCに物理的に装着されているメモリ量を超える場合は，図2–2のように，メモリに入りきらない部分を補助記憶装置へ書き出しておき，必要に応じて優先度の低い部分を補助記憶装置に書き出しておく(スワップアウト)ことでメモリ領域を確保する．そして，必要な部分を確保したメモリ領域へ読み出す(スワップイン)ことで実行を続ける．これを繰り返すことにより，少ないメモリ量でも大容量のメモリがある場合と同じようにプログラムの実行を続けることができる．

2.1.1 LinuxOSのスワップ機構

Linuxの仮想メモリでは，基本的にページ単位(4KB:キロバイト)で処理が行われる．つまり，スワップイン・スワップアウトはページ単位で行われる．

スワップアウト処理は以下のような場合に起動される．

- バッファやページの割り当てに失敗したとき

- 利用可能なページの数が一定の閾値を下回ったとき(カーネルスレッドkswapdにより起動される．)

スワップアウトの対象となるメモリ上のページは，一般的に擬似的なLRUアルゴリズムを用いて決定される．Linuxでは，アクティブなページのリストと非アクティブなページのリストを管理している(図2–3)．ある一定の期間に連続してアクセスがあるとページはアクティブなリストに移され，逆にしばらくアクセスがないとページは非アクティブなリストに移される．スワップアウトが行われる時には，非アクティブなリストからページを取り出し，スワップ領域に移動される．

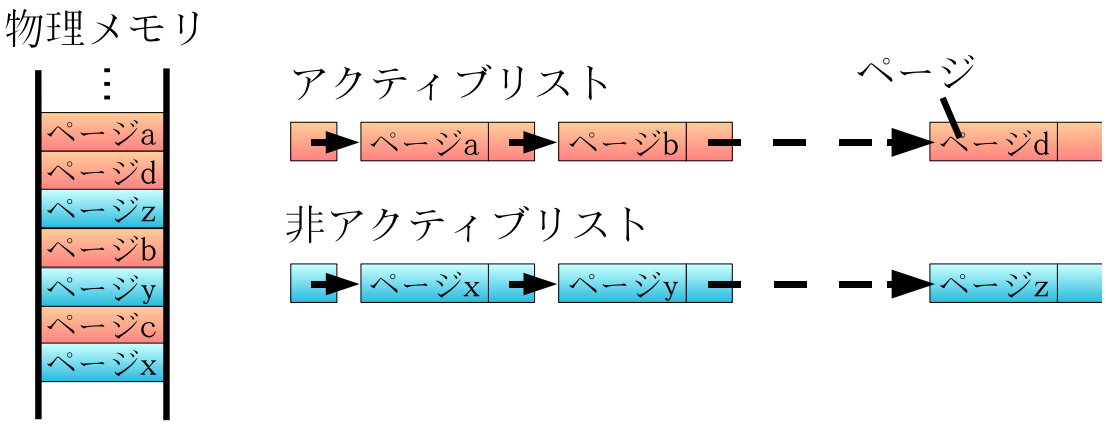

図 2–3　OS のスワップ機構における線形リストを利用したメモリ管理

スワップ領域上のどこに保存するかはスワップアウトの度にそれぞれのページについて計算され，連続したスワップアウト要求では，スワップ領域上でなるべく連続した位置に保存するようにする (図 2–4)．これには，HDD をスワップ領域とした場合にシーク時間を削減することと，関連したページをなるべく近くに集めるためという二つの理由がある．

スワップイン処理は，ユーザプロセスが既にスワップアウトされたページにアクセスしようとして，ページフォルトが発生した時にページフォルトハンドラから起動される．スワップインする時は，スワップ領域上で取得するべきページ以降の一定ページを連続してスワップインしようとする．具体的には，図 2–5 のように，カーネル内の変数 *page-cluster* を用いて $2^{page-cluster}$ ページ分だけ先読みされる．この先読みもスワップアウトと同様に，シーク時間の削減とページのクラスタ化を目指して実装されている．

スワップイン・スワップアウト要求は，ページ単位のブロックデバイス[1]への要求に変

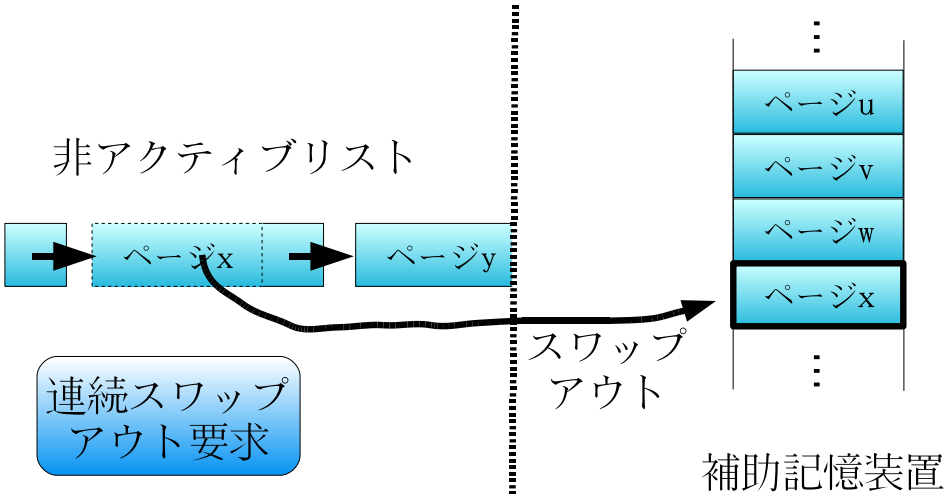

図 2–4　OS のスワップ機構におけるスワップアウト

換され，ブロックデバイスへのアクセスを抽象化する汎用ブロックレイヤを通過し，最終的にデバイスドライバが要求を処理する．汎用ブロックレイヤは，ブロックデバイスへの要求を一定時間ため込み，可能ならば要求のマージを行ってデバイスドライバへ処理を引き継ぐ役割を担っている．

デバイスドライバが転送を終えると，スワップアウトの場合，不要になったメモリ上のページが解放され，スワップインの場合，スワップデバイス上のページスロットが解放される．

以上が Linux のスワップ機能の処理の大まかな流れである．

[1]データの読み書きが，ある大きさのブロック単位（通常 512〜2048 バイト）でランダムに行えるデバイスのこと．具体例として，HDD や CD-ROM ドライブ，フロッピー・ディスク (FDD) などのディスク装置全般がブロック・デバイスとして挙げられる．

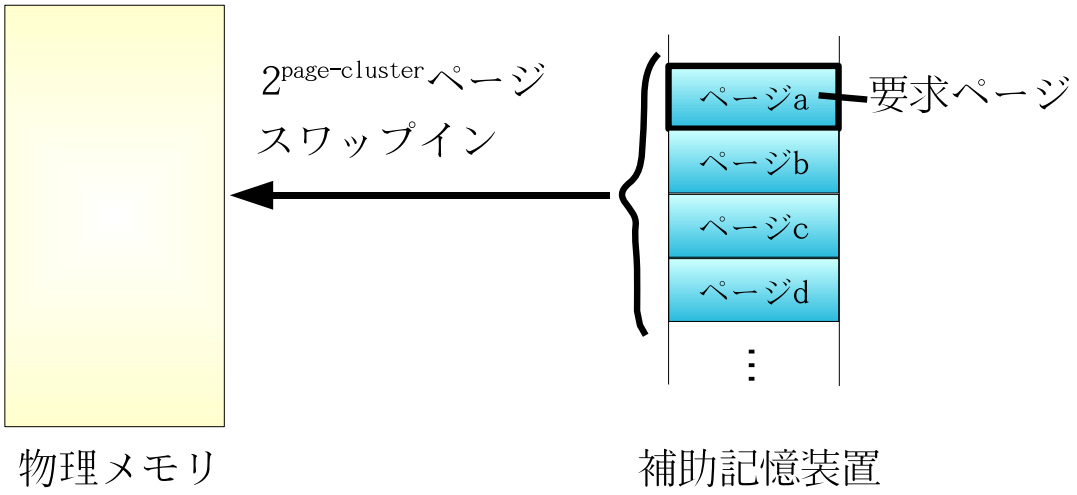

図 2–5　OS のスワップ機構におけるスワップイン (page-cluster=2 のとき)

2.1.2　断片的なメモリアクセス

本節では，行列積の演算処理を例に，LinuxOS のスワップ機構における物理メモリ内での断片的なメモリアクセスによって性能が低下する問題を挙げる．

計算機上で二次元配列 N x N の行列積を解く場合，$O(n^3)$ 時間必要であることが一般で知られているが，この計算量は，先で示した擬似コードのように行列同士の掛け算をするための反復処理によって引き起こされているものである．

二次元配列 N x N における行列積の擬似コード

```
#define N 100                        # 行列サイズ
Iterate i, j, k;                     # 反復処理のためのイテレータ
Array A[N][N], B[N][N], C[N][N];     # N x N の行列
for (i = 0; i < N; i++) {            # 反復処理
  for (j = 0; j < N; j++) {          # 反復処理
    tmp = 0.0;
    for (k = 0; k < N; k++) {        # 反復処理
      tmp += A(i, k) * B(k, j);
    }
    C(i, j) = tmp;
  }
}
```

- k に関する for 文では，反復回数が $O(n)$ であり，tmp+=A(i,k)*b(k,j); の計算量は $O(1)$ なので，全体として $O(n*1)$=$O(n)$
- j に関する for 文では，反復回数が $O(n)$ であり，反復の中の 1 回の計算量は $O(1)$+$O(n)$ = $O(n)$ なので，全体としては $O(n*n) = O(n^2)$
- i に関する for 文では，反復回数が $O(n)$ であり，反復の中の 1 回の計算量は $O(n^2)$ なので，全体としては $O(n*n^2) = O(n^3)$

となり，n 回の三重ループの演算処理の計算量は $O(n^3)$ ということになる．この構文から分かるように，k に関する for 文では，配列 A，B のメモリアクセスパターンは xy 座標系で表すと図 2–6 のようになる．

ここで，配列 A に対するアクセスパターンは x 方向，配列 B に対するアクセスパターンは y 方向へのアクセスとなる．二次元配列へのアクセスにおいて x，y 方向へのそれぞれのアクセスは，xy 座標系では連続な領域として表されるが，二次元配列をアドレス空間にマッピングした場合，y 方向へのアクセスは，図 2–7 の暗色の領域で示されるように断片的なアクセスとなる．

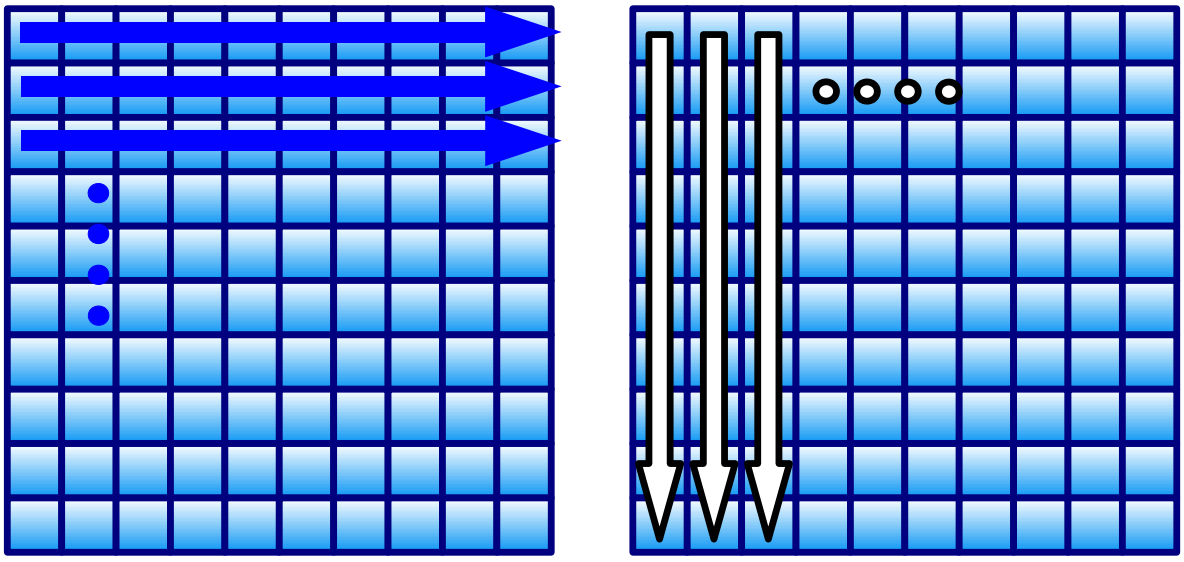

図 2–6　行列積の演算における各配列のアクセスパターン

二次元配列に対して先ほどの擬似コードを用いた行列積の演算を想定した場合，行列積の演算では，x 方向と y 方向のアクセスが交互に発生する．図 2–8 赤矢印のようにアクセスしたときに物理メモリ上にデータをページ単位でスワップする場合を想定する．今回，問題を簡単化に説明するために 1 ページを配列 4 個分として話を進める．

まず，はじめの要素にアクセスしたとき，物理メモリ上に仮想アドレス空間で連続となるページ 1 が存在するかを判別する．今回，物理メモリ上に該当ページ 1 が存在しないため，補助記憶装置からページ 1 をスワップインする．

続いて，次の要素にアクセスしたときも同様にページ 2 が物理メモリ上に存在しないため，スワップインが発生する．以下，同様の流れが続き，データにアクセスする度に

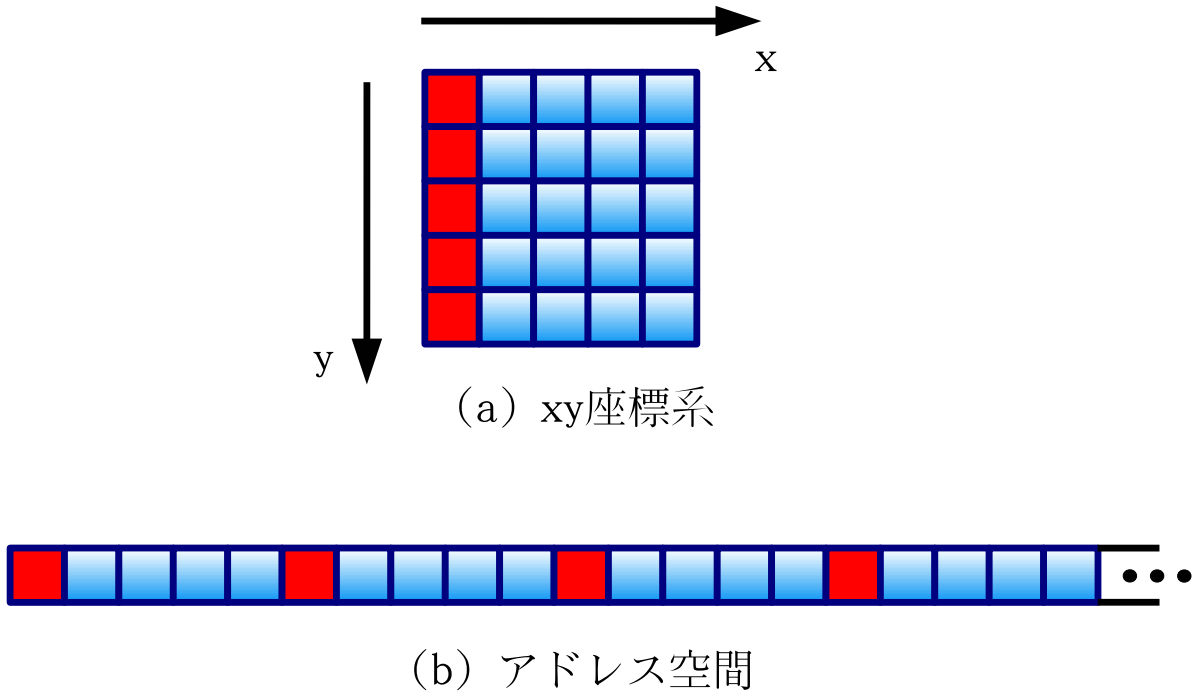

図 2–7　xy 座標系とアドレス空間で表現された二次元配列

スワップインが繰り返される．配列サイズを N とすると，これは理論上スワップ回数を *Swap*，配列全体のデータサイズを *Datasize*，ページサイズを *Pagesize*，データ構造のサイズを *Size of Structure* とすると，スワップ回数とデータサイズの関係は次のようになる．

$$Swap = Datasize/(Pagesize/SizeofStructure)$$

例えば，データ構造が double 型配列の場合，1 要素あたり 8 バイトなので，1 ページに入る要素数は 512 となるため，スワップ回数は最悪 512 倍変わることになる．今回の例のように y 方向へのアクセスのような断片的なメモリアクセスがあると，OS スワップ機構は y 方向への連続ページを管理していないため，連続にメモリアクセスすることができず，頻繁にスワップインが発生し，性能が低下する．

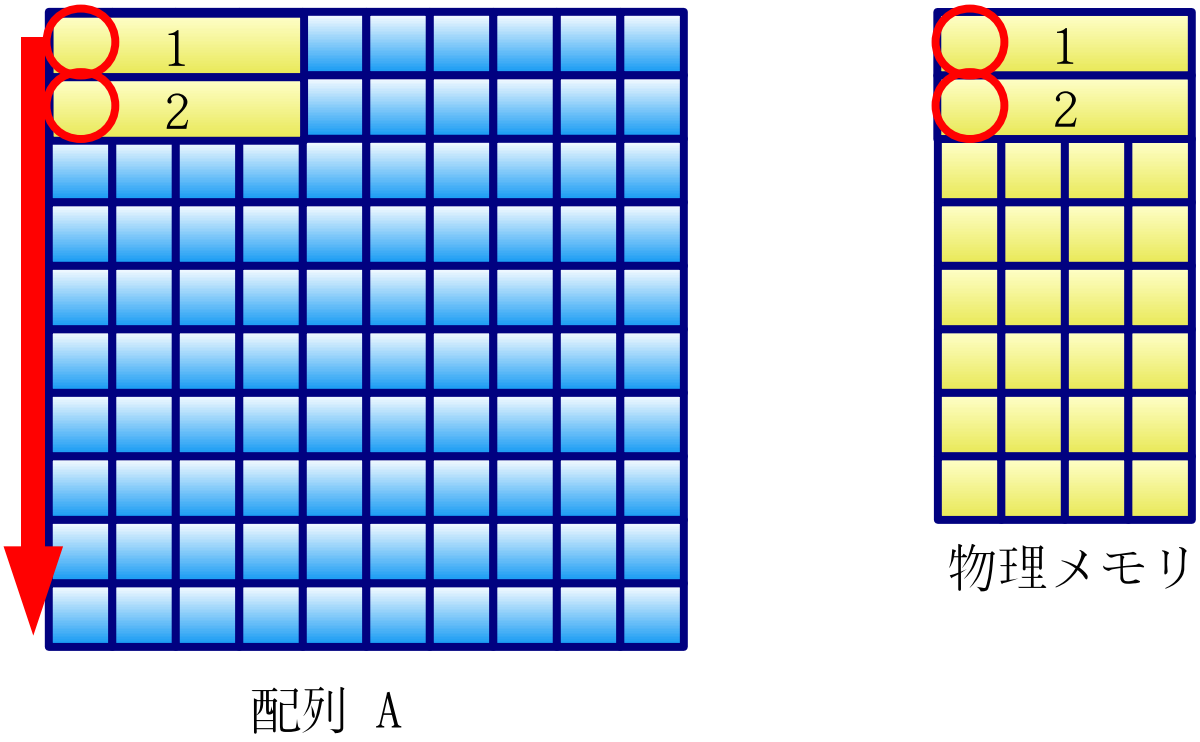

図 2–8　アクセスパターンに対してメモリにスワップインされるページ

2.2 遠隔メモリスワップシステム

遠隔メモリスワップシステムでは，ローカルマシンの物理メモリサイズに制限されることなく，クラスタの各計算機 (ノード) のリモートメモリを利用して仮想的に大容量のメモリ空間を提供する．遠隔メモリスワップの概念を図 2–9 に示す.

各ノードは独立した計算機であり，ネットワークで接続されてクラスタを構成する．ローカルマシンの OS はシステムのメモリが足りないことを検出すると，遠隔マシンのメモリ (リモートメモリ) へスワップアウトして空きメモリを確保する．遠隔メモリスワップシステムには，OS のスワップ機構を用いるシステムと用いないシステムが存在する.

大容量仮想メモリを提供する Teramem[4] では，OS のメモリスワップ機能を使用しない遠隔メモリスワップシステムで，Myrinet[10] や InfiniBand[11] といった高速ネットワークで接続された大規模クラスタにおける高性能計算向けの利用を想定している．Linux のスワップ機構と独立に実装することにより，遠隔メモリへのページ転送が最適化されているという特徴を持つ.

一方，比較的安価なネットワークを用いてクラスタを構成することで大容量メモリを実現する Nswap[5] では，クラスタ内で起こるメモリ使用量の不均衡に着目し，メモリに余裕があるノードがメモリ不足のノードに対して一部のメモリを分け与えることでクラスタ全体のメモリの有効利用を目指している．Nswap はスケーラビリティを重視しており，クラスタ全体のメモリ割り当てを集中管理するノードは存在せず，各ノードはクラスタ全体の厳密な状態を知らなくても動作する設計となっている．サーバの物理メモリが不足したときは，サーバ間でページのマイグレーションができるほか，各ノードはクライアントとしての振る舞いとサーバとしての振る舞いを動的に切り替えることができる.

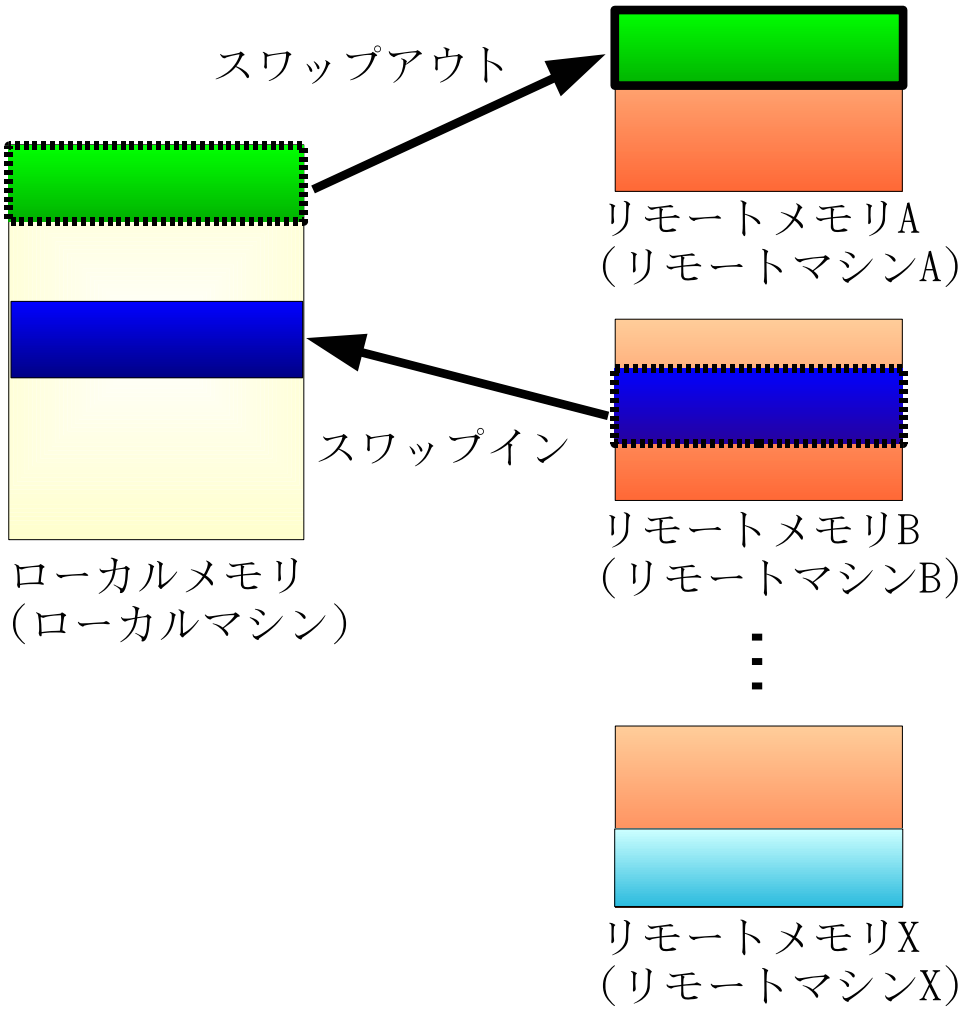

図 2–9　ローカルメモリとリモートメモリによって実現されたメモリ空間

第 3 章

空間的局所性を考慮したメモリ管理手法

本章では，行列積の演算処理を例に，断片的なメモリアクセスによって性能が低下する問題を解決するために，整列データを利用することで連続的なメモリアクセスを可能する方法を提案する．

3.1 空間的局所性を考慮した整列データを利用したスワップイン

先の節では，二次元配列に対し，列方向に順番にアクセスすると，アクセスの度にページをスワップインする必要があり，性能が低下する可能性を示した．

そこで，アクセスパターン毎に対応した整列させたデータを別データとして補助記憶装置に保存しておく．具体的には，図 3–1 のように，配列に対し①と②のようにアクセスしたときのデータの並び順番で整列させたデータを別々のファイルとして補助記憶装置に格納する．

そして，図 3–2 のように，あるアクセスパターンでメモリアクセスがあった場合に，先ほど説明したあらかじめ生成しておいたアクセスパターンに対応する整列データから該当ページをスワップインすることで連続メモリアクセスが可能となり，スワップ回数を抑制することができる．

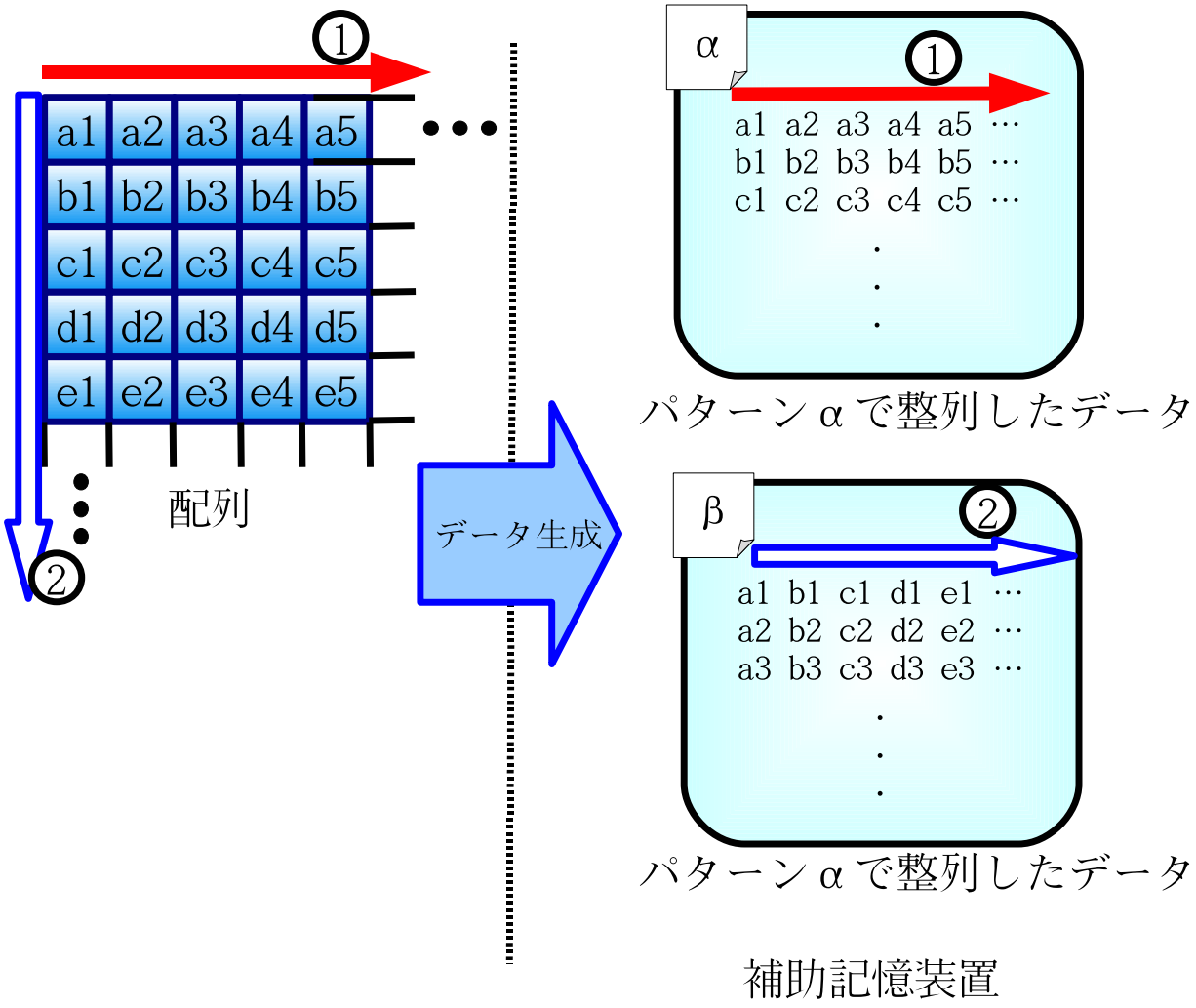

図 3–1　連続メモリアクセスを可能とする整列データ

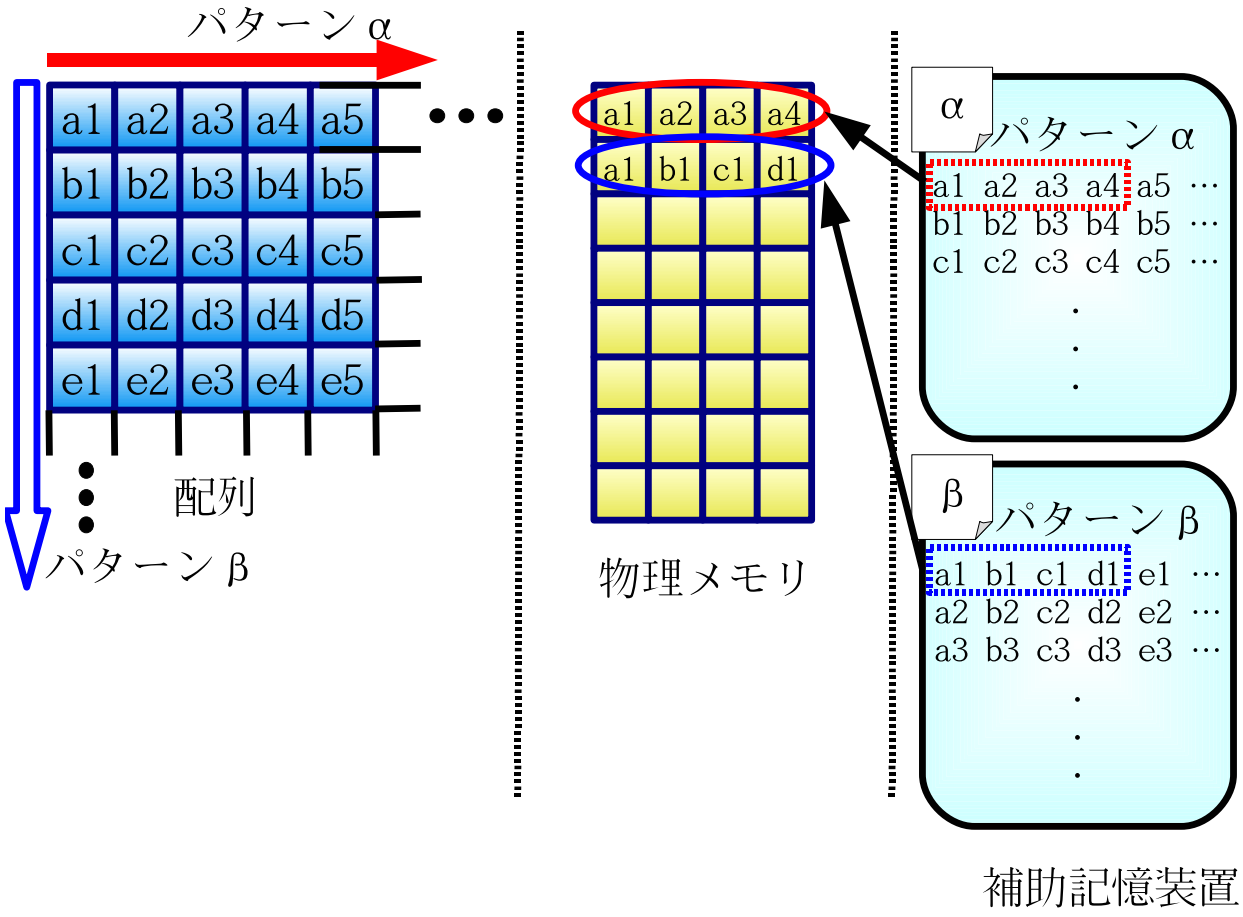

図 3–2　整列データによって実現する連続メモリアクセス

3.2　空間的局所性を考慮したメモリ管理手法

本節では，あるアクセスパターンでメモリアクセスがあった時の空間的局所性を考慮したメモリ管理方法について述べる．提案するメモリ管理手法では，仮想メモリの概念と同様に，該当データに対する仮想アドレスをページ部とオフセット部から構成される物理アドレスに変換し，ページテーブルとメモリ管理ユニットによってページを管理する (図 3–3).

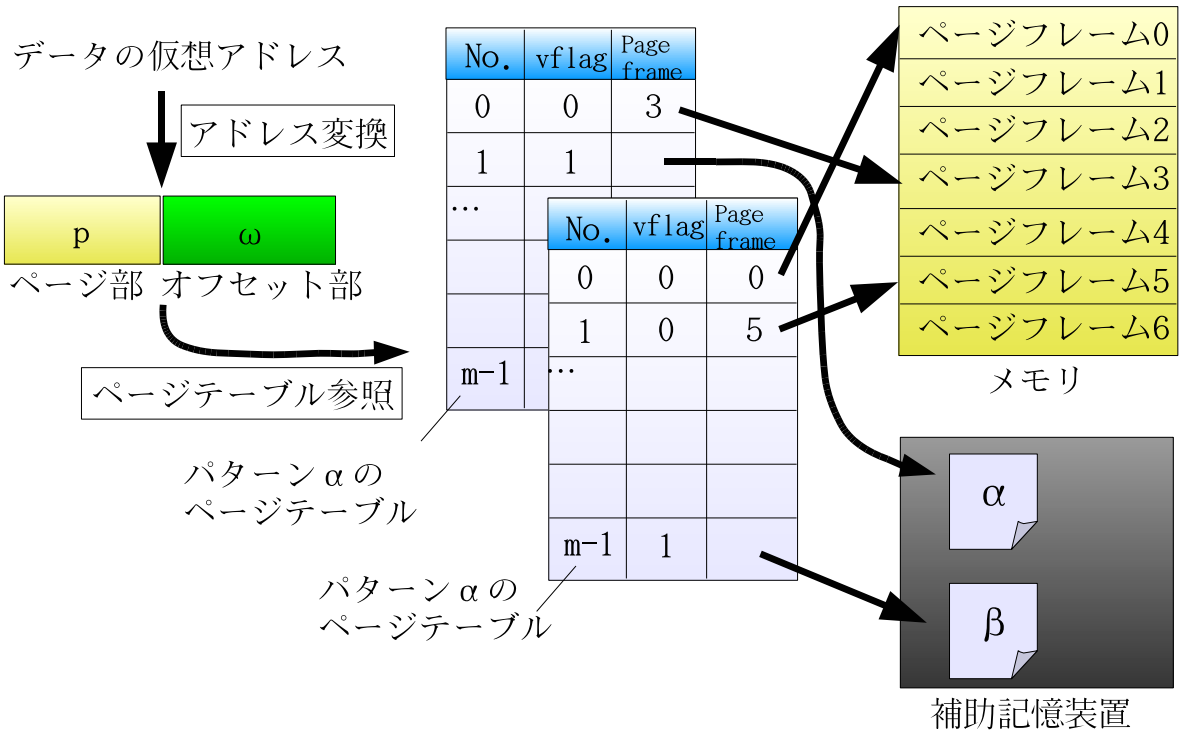

図 3–3　メモリ管理部の動作

本提案手法を適用するために，今回，本提案におけるメモリ管理手法ではメモリアク

セスパターンに対応した空間的局所性を考慮したメモリ管理を行うために，以下のような部位を実装した．

- 整列データ生成機構
  指示されたアクセスパターンに対応する整列データを生成する機構 (図 3–4)．

- アドレス変換部
  仮想アドレスを入力としたとき，アクセスパターンに対応する物理アドレスのページ番号とオフセット値を出力する機構 (図 3–5)．

- ページテーブル
  表 3–1 で表すような構造を持つページテーブル．ページ番号，v フラグ，ページフレーム番号と呼ばれるエントリから構成され，ページ番号からページフレーム番号へのマッピングを行う．ページ番号は仮想アドレスから変換された物理アドレスにおけるページ番号である．なお，ページテーブルはアクセスパターン毎に複数のテーブルが存在し，あるアクセスパターンでページへアクセスがあった場合，対応するページテーブルが参照される．

表 3–1　ページテーブル

No.	vflag	page frame
0	1	-
1	0	f1
2	0	f2
...	...	...

表 3–2　ページテーブルエントリと説明

エントリ	説明
No.	テーブルインデックス番号
vflag	該当ページが主記憶か補助記憶装置のどちらかに存在するかを示すフラグ
pageframe	主記憶上のページフレーム番号

- メモリ管理部
  ページテーブルを参照して，メモリと補助記憶装置とのページを管理する部位．

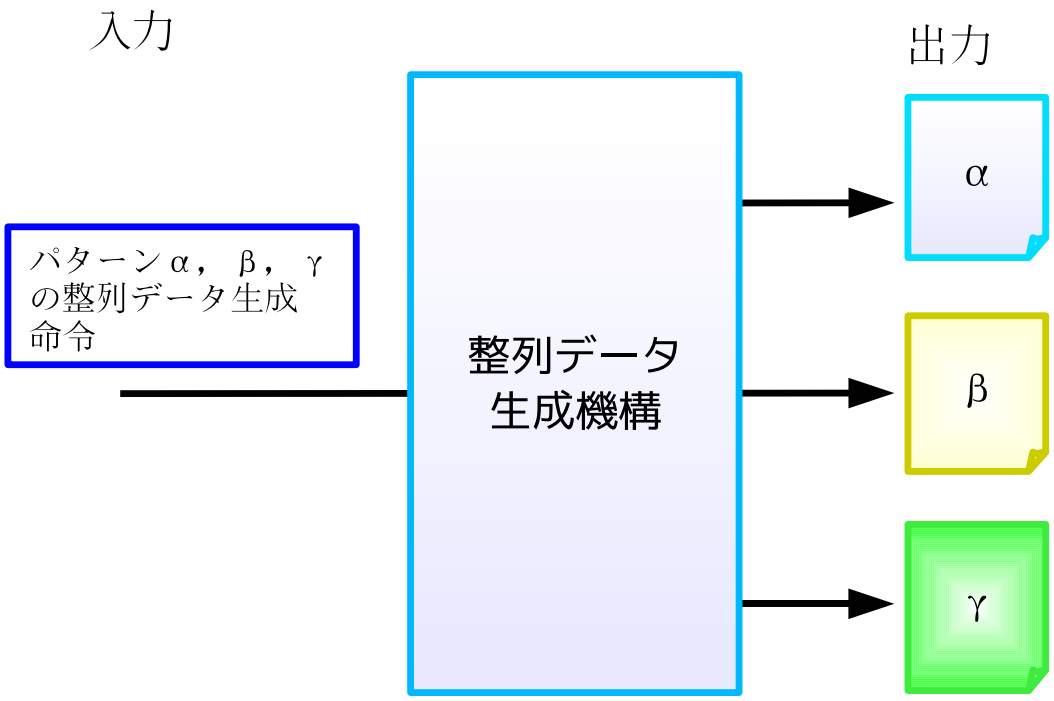

図 3–4　データ生成機構

3.2.1　メモリ管理アルゴリズム

本節では，前節で紹介した整列データ生成機構，アドレス変換部，メモリ管理部を用いてユーザレベルで実現したメモリ管理のアルゴリズムについて説明する．まず，あらかじめ整列データ生成機構において指示されたアクセスパターンに対応する整列データを用意しておく．そして，データに対してアクセスがあった場合，メモリアクセスパターンに対応するページ番号とオフセット値をアドレス変換機構によって変換する．そして図 3–3 のように，複数のページテーブルを用いてアクセスパターンに対応するページを管理する．

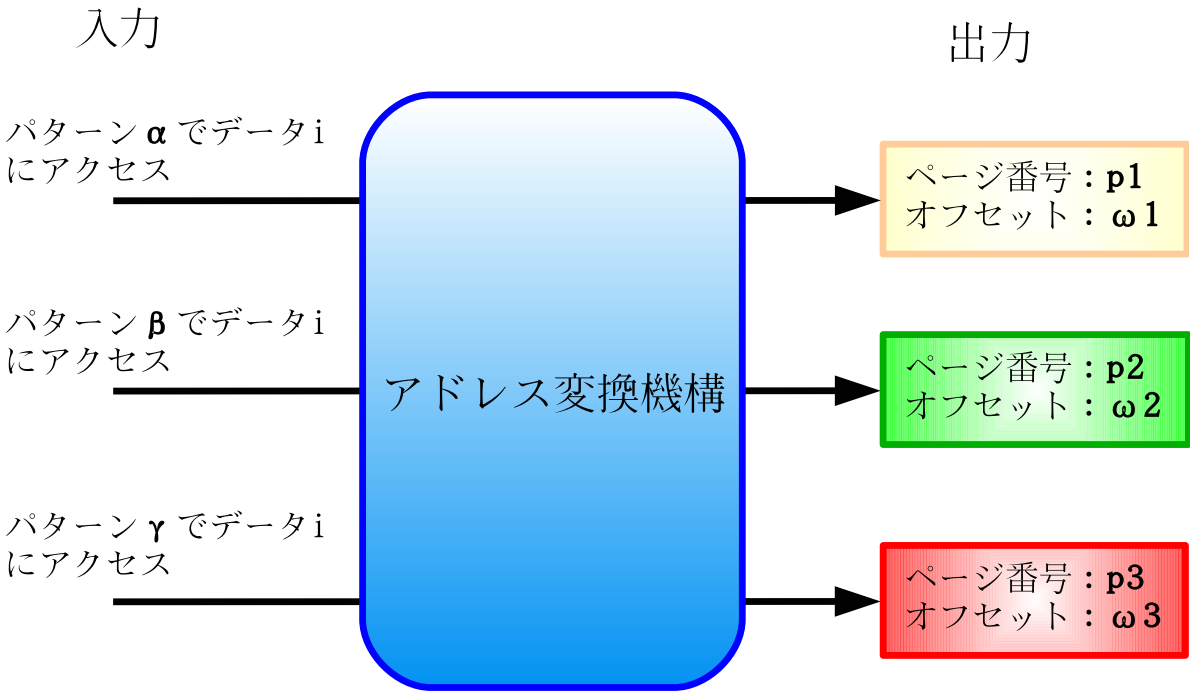

図 3–5　アドレス変換機構

図 3–6 に提案手法を適用したメモリ管理アルゴリズムを示す．図において色が塗られた部分が，提案手法を適用した場合の特徴となる部分となる．

まず，アクセスパターンに対応するページテーブルを参照する．そして，該当ページがメモリ上にあるか補助記憶上にあるかを vflag によって判別する．vflag = 0 の場合，該当ページがメモリ上に存在するため，ページ番号に対応するページフレーム番号からメモリ上のページフレームのオフセット値を得る．vflag=1 の場合，該当ページがメモリ上に存在しないため，メモリ上に該当ページをスワップインする必要がある．スワップインの際，メモリに空きがあるかどうかで必要に応じてメモリ上にあるページをスワッ

プ合うとさせる必要がある．

メモリに空きが存在する場合，空いている領域に対して該当ページを割り当てる．その後，割り当てられたページに対応するパターンのページテーブル該当ページを更新する．

メモリに空きが存在しない場合，メモリに空き領域を確保するためにスワップアウトした後，該当ページをスワップインする．そして，スワップアウトされたページフレームに対応するアクセスパターンに対応するページテーブルの該当ページを更新する．

次に，確保されたメモリ領域に対応するアクセスパターンの該当ページをスワップインする．そして，割り当てられたページに対応するパターンのページテーブル該当ページを更新する．今回の仕様ではページ内容が書き換えられないことを前提としているため，スワップアウトは起こらず，単に置き換えられる対象ページがスワップインするページに上書きされる．

3.2.2 スワップのタイミング

メモリを管理する上で，いつスワップするかという問題があるが，今回の実現方法における各スワップ処理のタイミングは次のようになる．

- スワップアウト
  スワップイン処理において物理メモリに空きページフレームが存在しないに発生する．

- スワップイン
  物理メモリ上に該当ページが存在しない場合に発生する．

3.2.3 スワップ先となる空きページフレームの選択

物理メモリに空きがある場合，ページフレームの中でもページフレーム番号が小さい順番でページが割り当てられる．物理メモリに空きがない場合，スワップアウト処理を行い物理メモリに空き領域を確保した後，ページが割り当てられる．

3.2.4 スワップアウト対象となるページ選択

今回の実現方法におけるスワップアウト対象ページの選択方法は，FIFO(FirstInFirstOut)の概念をもとにページを選択する．物理メモリに一番長時間存在するページが優先的にスワップアウト対象ページとなる．

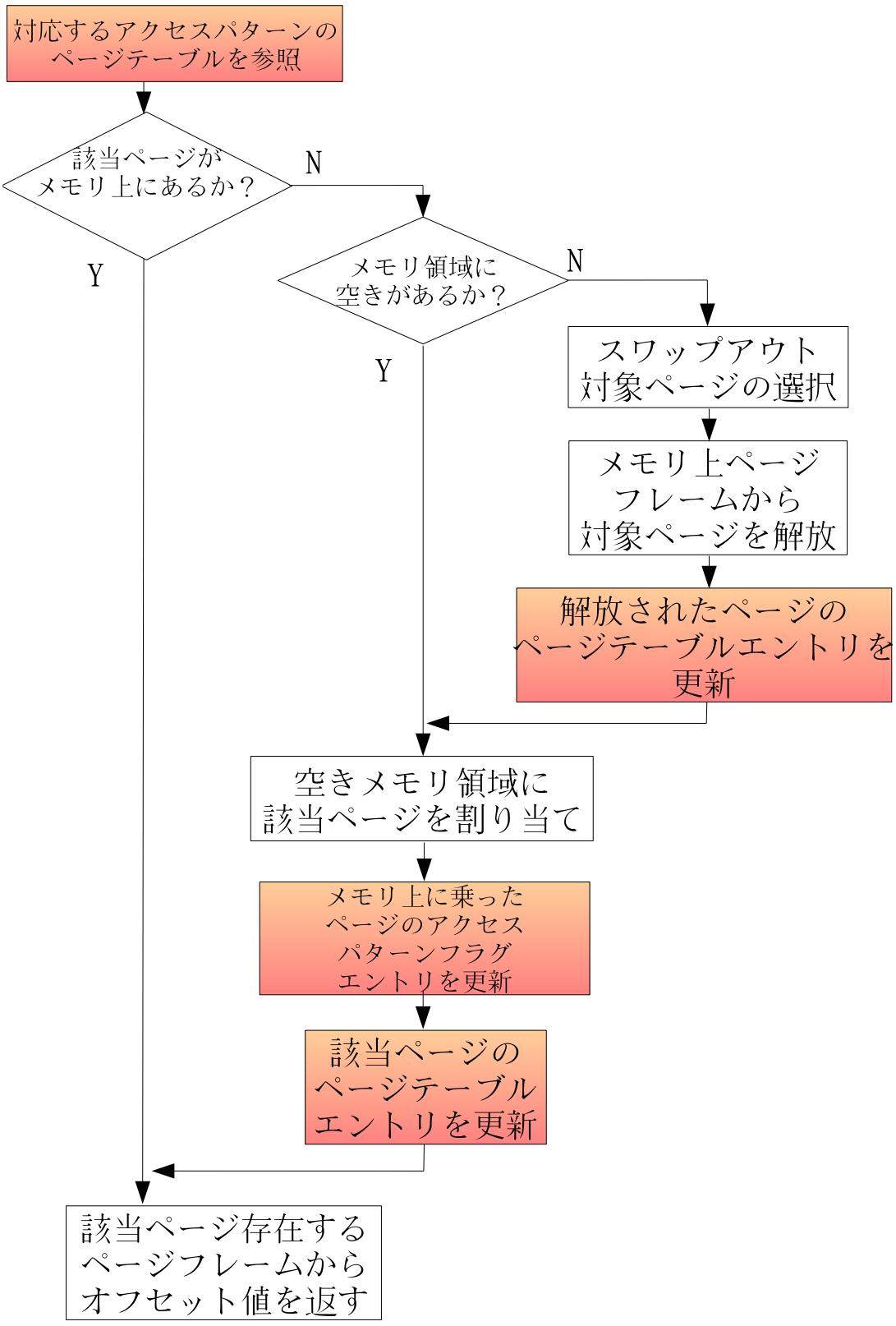

図 3–6　メモリ管理アルゴリズム

第 4 章

実験と評価

本章では提案手法の有効性を確かめるために実験を行った．評価対象は提案手法である空間的局所性を考慮したメモリ管理手法であり，比較対象は LinuxOS のスワップ機構を用いて評価を行った．はじめに実験環境を説明し，次に実験内容と評価結果，考察を示す.

4.1 評価環境

実験に用いた計算機とデバイスの仕様を表 4–1 に示す．LinuxOS のスワップデバイスは HDD を用い，4GB 分の容量をスワップ領域として確保した．HDD のアクセス性能は Linux ベンチマークソフト bonnie++によって読み込み/書き込み性能を計測した．LinuxOS のスワップ機能において実行時間を計測する際は，Linux kernel 機能にある cgroup によって利用できるメモリの大きさを制限して実験を行った．付録 A.1 に cgroup によるメモリ制限方法を示す．提案手法を適用したメモリ管理手法の設定値はメモリを 8MB、ページサイズはファイルシステムで用いられているブロック単位でもある 4KB をスワップ単位としてメモリ管理を行った．また，今回提案手法の有効性を確証させるために，キャッシュの効果を取り除いて (flush して) 実験を行った．付録 A.2 にキャッシュのフラッシュ方法を示す.

表 4–1　実験環境

CPU	名前	AMD Athron 64 x2 Dual Core Processor 4600+
	コア数	2 個
	クロック周波数	1GHz
	L1 キャッシュサイズ	128KB
	L2 キャッシュサイズ	512KB
メモリ	規格	DDR2-SDRAM
	メモリサイズ	2GB
OS	ディストリビューション	Ubuntu8.10
	Linux Kernel	2.6.27
	スワップ領域サイズ	8GB
	回転数	7200rpm
	容量	320GB
	接続	SATA 3.0Gb/s
	キャッシュ	16MB
	読み込み性能	72 MB/s
	書き込み性能	52 MB/s
ファイルシステム	ext3	

4.2　行列積の演算

提案手法の有効性を検証するために，行列積の演算プログラム実行時における LinuxOS のスワップ機構と提案手法の演算時間，プログラム全体の実行時間で評価を行った．行列積の演算プログラム内容は，double 型二次元配列 N x N の行列積の演算で，要素数をパラメータとする．各配列要素数で実験を試行した回数はそれぞれ 10 回であり，結果は試行回数 10 回の平均値を示す.

前提条件として，両手法ともに使用できるメモリを 8MB に設定した．このときの物理メモリとパラメータである二次元配列 A，B の要素数 N によるサイズは図 4–1 のよ

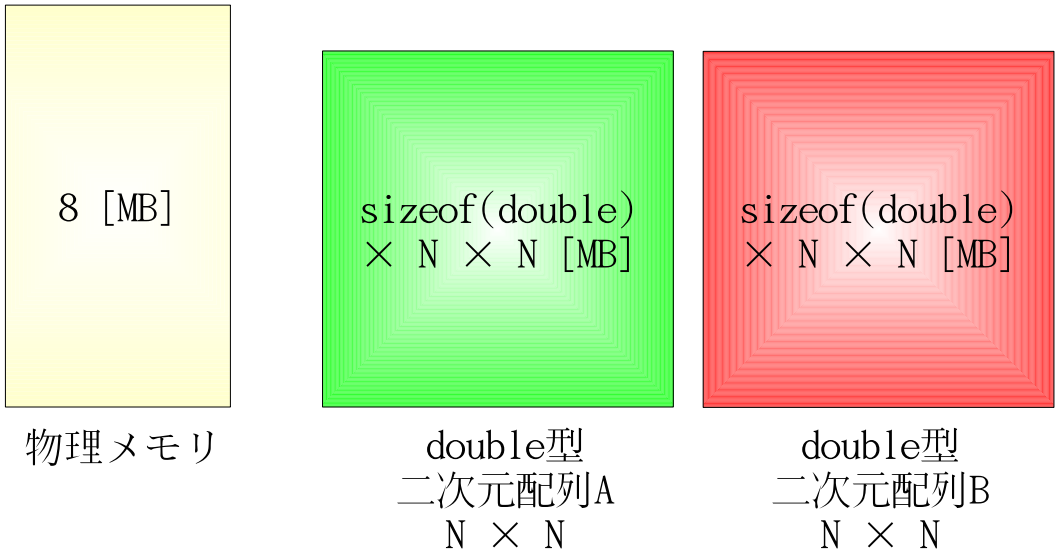

図 4–1　行列積の演算における物理メモリと演算データサイズの関係

うになる．LinuxOS のスワップ機構での実行時間計測時は cgroup 機能でメモリ使用量を制限した．提案手法では，静的に 8MB の領域を確保した double 型配列を宣言し，これをプログラム上での仮想的な物理メモリとして扱った．また，提案手法において，プログラム内にアクセスパターンが判別できる指示を含ませた．

演算時間やプログラム全体における実行時間の計測方法は，C 言語ライブラリ gettimeofday() によって計測を行った．また提案手法においては，プログラム実行開始時にあらかじめ行列積のメモリアクセスパターンに対応する整列データを生成し，補助記憶装置に格納しておく必要があり，そのための生成時間がオーバーヘッドとなり提案手法における性能低下を引き起こす可能性がある．よって，整列データ生成から完了までの時間も計測した．

結果を表 4–2，図 4–2 に示す．表 4–2 の数値をグラフ化したものが図 4–2 である．図 4–2 における横軸は配列の一辺における要素数 [個] で縦軸は時間 [s] である．それぞれの要素数において左のグラフが LinuxOS のスワップ機構の結果で，右のグラフが提案手法の結果である．LinuxOS のスワップ機構における要素数 832 x 832 からの詳細な数値は，表 4–2 を参照してもらいたい．また，提案手法における実行時間の内訳として，演算時間と整列データ生成時間があるが，下部のグラフが演算時間であり，上部が整列データ生成時間である．

表 4–2　行列積の演算結果

配列の要素数	LinuxOS のスワップ機構	提案手法	
	演算時間 [s]	演算時間 [s]	整列データ生成時間 [s]
512*512	7.71	13.02	1.79
768*768	26.01	46.77	3.64
832*832	784.03	112.26	4.19
896*896	950.92	142.10	4.97
960*960	990.08	178.46	5.16
1024*1024	1129.47	236.20	6.06

4.2.1　演算時間についての結果と考察

表??の演算結果より，行列サイズ 768 x 768 までは OS のスワップ機構の方が短時間で演算処理を終えているが，行列サイズが 832 x 832 以上であるとき，提案手法が LinuxOS のスワップ機構に対し短時間で演算する結果となった．

一方，提案手法では，アクセスパターンに対応した整列データからページをスワップインすることで，LinuxOS のスワップ機構に比べ短い実行時間で演算処理が実現されていることがわかる．

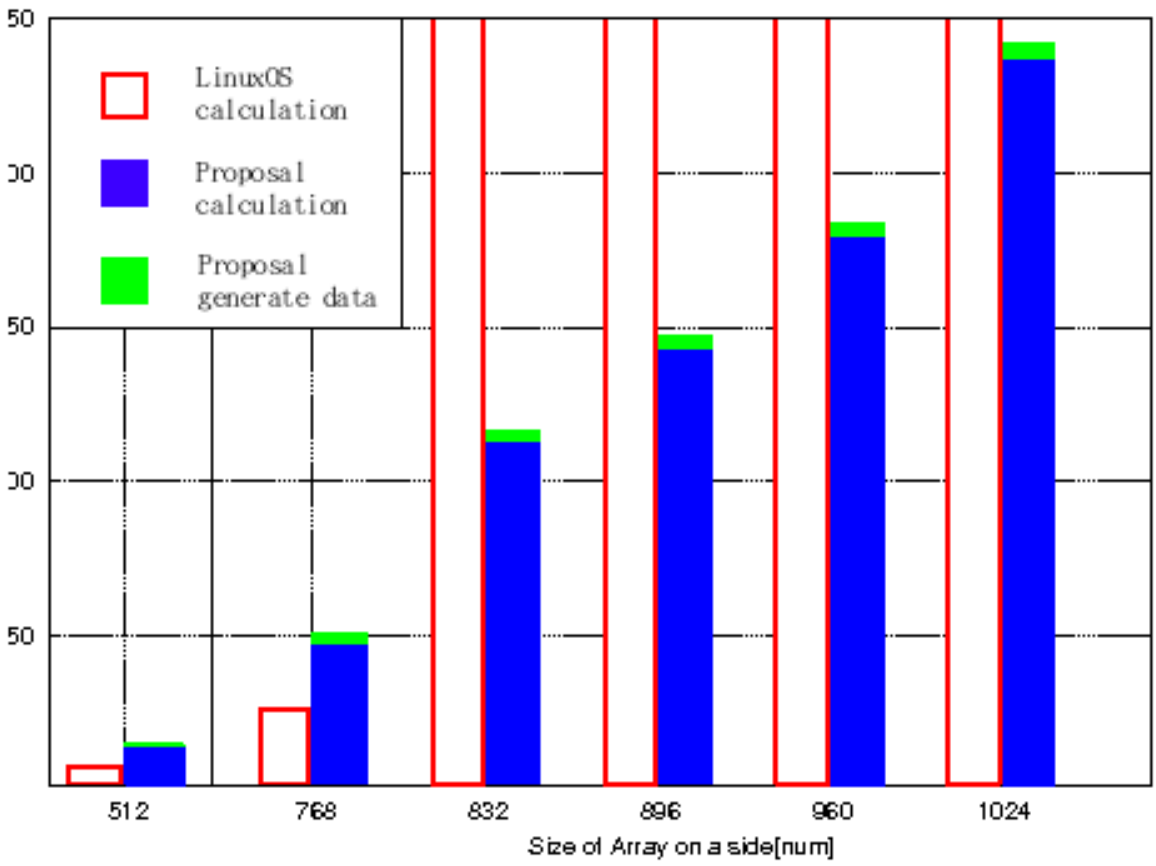

図 4–2　行列積の演算における実行時間の内訳

考察としては，LinuxOS のスワップ機構は行列サイズが 768 x 768 から 832 x 832 の間で急激にスワップ回数が増加したためと考えられる．提案手法では，連続メモリアクセスできた効果で余計なスワップ回数発生を抑制しており，これにより演算時間の延長が最小限に抑えられているとみれる．

以上，演算時間の結果より，提案手法の演算時間における有効性を確認した．

4.2.2 整列データ生成時間によるオーバーヘッドについての結果と考察

表 4–2，図 4–2 の提案手法における整列データ生成時間の結果より，整列データ生成時間は最大でも計算時間の 1 割程度のオーバーヘッドで済むことがわかった．提案手法ではプログラム上で指示されたアクセスパターンの数だけ整列データを生成するが，データサイズが大きくなったときの演算時間の増加分に比べると，整列データ生成時間の増加分はさほど演算処理に差し支える程の影響は与えていないことが分かる．

以上，整列データ生成時間の結果より，提案手法のオーバーヘッドの影響は演算時間に比べ微々たるものであることが分かった．

4.2.3 演算時間と整列データ生成時間を含めたプログラム全体の実行時間についての結果と考察

表 4–2 図 4–2 の LinuxOS のスワップ機構の全体の実行時間 (演算時間) と提案手法の全体の実行時間 (演算時間+整列データ生成時間) の結果より，行列サイズが 832 x 832 以上のとき，提案手法が LinuxOS のスワップ機構に比べ短時間でプログラムを実行終了することができた．

Linux の OS スワップ機構は，第 2 章 2.1.1 節でも説明したとおり，大まかに擬似 LRU アルゴリズム，ページクラスタリングという 2 つのアルゴリズムによってメモリ管理を行っている．今回，LinuxOS の演算時間が配列の要素数が大きくなる毎に大幅に性能低下したのは，ページクラスタリングによって余分なスワップ回数が発生したことに起因すると考えている．ページクラスタリングのカーネルパラメータである/proc/sys/vm/page-cluster の値を調査したところ,「3」であった．これはつまり，ディスクへのアクセス回数を示すページフォルトが起きた回数に対し，スワップ回数が $2^3 = 8$ 回であることから，データへアクセスしたときに該当ページが物理メモリ上に存在しなかった場合，最高で一度に 8 ページ分スワップインしていることもあるということである．このスワッ

プした総ページ数のうち，連続メモリアクセスに貢献するデータが含まれているかどうかにもよるが，今回の推測であるページクラスタリングによる影響と実験パラメータである要素数サイズ増加によって性能低下が引き起こされたと考えている．

今回，スワップ回数のデータを示していないことに加え，LinuxOS のスワップ機構が上記 2 つのアルゴリズム以外にも細かいチューニングパラメータによって管理されていることから，正確なスワップ回数を出すことができなかった．

以上から，LinuxOS のスワップ機構に比べ，提案手法が有効であることを確認した．

第 5 章

まとめ

大規模メモリ空間実現のために，物理メモリと補助記憶装置から構成されるような PC において，空間的局所性を考慮したユーザレベルメモリ管理手法を提案した．

行列積の演算の実験において，二次元配列のデータサイズが物理メモリ容量を越える場合，LinuxOS のスワップ機構に比べて実行時間における提案手法の有効性を確認した．また，提案手法適用によって引き起こされる整列データの生成時間によるオーバーヘッドも，演算時間に比べて 1 割に満たない程度の時間で抑えることを確認できた．さらに，演算時間と生成時間を含めたプログラム全体の実行時間においても，LinuxOS のスワップ機構に対し，提案手法が有効であることを示した．

今後の課題としては，行列積の演算以外の行列計算を提案手法を適用して計算したときの実行時間や特性を考察することである．また，現段階のメモリ管理手法における実装部分であるメモリ管理機構では，あるデータの更新において，補助記憶装置やメモリ上に存在する同一データとの一貫性を考慮していないため，データ同士に不整合が生じる場合がある．よって，ページテーブル，メモリ管理部，メモリ管理アルゴリズムに改良を加え，データの値が更新・変更された場合に本手法を適用したときの有効性を示すことが今後の課題として挙げられる．具体的には，あるページにアクセスした際，該当ページが読み込みできるか書き込みできるかを示すための制御フラグ用エントリを追加することで，ページに対するアクセス制御を実現する．また，あるページがスワップインしてからスワップアウトするまでに，そのページに対して一度でも更新があった時点で，更新があったことを示すための更新フラグ用エントリの追加である．このエントリ

により，スワップアウト対象となるページをあらためて補助記憶装置に書き戻す必要がなくなり，余分なディスクアクセス発生を抑制することが可能となる．さらにこの課題に関連し，近年高速な補助記憶媒体として注目されてきたSSDをスワップデバイスとして用いて，SSDの特性を活かしたメモリ管理手法を提案することが挙げられる．

第 6 章

謝辞

本論文は筆者が名古屋工業大学大学院工学研究科創成シミュレーション工学専攻博士前期課程に在籍中の研究成果をまとめたものです．同専攻教授松尾啓志先生には指導教官として本研究の遂行に関して終始，ご指導を戴きました．ここに深謝の意を表します．また同専攻準教授津邑公暁先生，情報工学専攻准教授齋藤彰一先生，創成シミュレーション工学専攻助教松井俊浩先生には本論文の細部においてご指導を戴きました．ここに感謝の意を表します．本専攻松尾・津邑研究室並びに情報工学専攻齋藤研究室の各位には研究遂行にあたり日頃より有益なご討論ご助言を戴きました．ここに謝意を表します．

参 考 文 献

[1] “MPI Documents”, *http://www.mpi-forum.org/docs/*.

[2] 松葉浩也, 石川裕：“動的アクセスパターン解析によるソフトウェア分散共有メモリ”, 先進的計算基盤システムシンポジウム SACSIS，情報処理学会, 2004.

[3] 城田祐介, 吉川克哉, 本多弘樹, 弓場敏嗣：“マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ”, 先進的計算基盤システムシンポジウム SACSIS，情報処理学会, 2003.

[4] 山本和典, 石川裕：“テラスケールコンピューティングのための遠隔スワップシステム Teramem”, 先進的計算基盤システムシンポジウム SACSIS, 07 2009.

[5] Tia Newhall 他：“Nswap:a network swapping module for linux clusters”, *Euro-Par Parallel Processing*, 2003.

[6] 後藤正徳, 佐藤充, 中島耕太, 久門耕一：“10GbEthernet 上での RDMA を用いた遠隔スワップメモリの実装”, CPSY，信学技報告 Vol.106 No.287,200.

[7] S.Liang, R.Noronha, D.K.Panda：“Swapping tp Remote Memory over InfiniBand : An Approach using a High Performance Network Block Device”, *IEEE Cluster Computing*, 2005.

[8] Pavel Mache: “Linux Network Block Device”, *http://nbd.sourceforge.net/*, 1997.

[9] 竹内健：“フラッシュメモリの最新技術動向：SSD への応用”, 情報処理学会研究報告, 09 2008.

[10] “Myri-10G Overview”, *http://www.myri.com/Myri-10G/overview/*.

[11] “Infiniband trade association”, *http://www.infinibandta.org/*.

[12] 緑川博子, 黒川原佳, 姫野龍太郎：“遠隔メモリを利用する分散大容量メモリシステム DLM の設計と 10Gb Ethernet における初期性能評価”, 情報処理学会論文誌, 12 2008.

[13] “LinuxKernelDocumentation::cgroup.txt”, *http://www.mjmwired.net/kernel/Documentation/cgroups.txt*, 10 2008.

付 録 A

実験時に用いた環境設定方法

A.1 cgroupによるメモリ使用量制限方法

本付録では，実験において用いたLinuxOSにおけるプロセスのメモリ制限の方法を示す．なお，この機能は，LinuxKernelバージョン2.6.27以降で利用可能ある．cgroupの詳細については，[13]を参考にしてもらいたい．例として，カレントディレクトリに存在するサブディレクトリtestを使ってメモリ使用量を制限する手順を説明する．

```
$ su root
$ mount -t cgroup none test -o memory
$ cd test
$ mkdir group
$ cd group
$ echo 4M >memory.limit_in_bytes
$ ./program
$ echo [PID] >tasks
```

まず，mountコマンドを使えるようにするためにrootでログインし，ディレクトリtest以下でcgroupの機能を有効にするためにマウントオプションタイプをcgroupに指定する．また，オプションをmemoryとすることで，cgroup機能によってメモリを制限するための各種設定ファイルが生成される．そして，マウントコマンド実行時にはマウントするデバイスとマウント先のマウントポイントを設定するが，マウントデバイスを指定せず(none)，マウントポイントをtestに指定する．次に，testディレクトリ以下に新た

にディレクトリ (説明では group) を作成する．cgroup 機能により，test 以下にサブディレクトリを作成することで，プロセスごとに異なるメモリ容量に制限することができる．そして，echo コマンド等によって memory.limit_in_bytes ファイルに制限メモリサイズを書き込む．最後に，メモリ制限をかけるプログラムを実行し，実行プログラムのプロセス ID(PID) を tasks ファイルに書き込むことで実行中のプログラムのメモリ使用量が制限される．(しかし，この方法ではプログラム実行後すぐに PID を取得し tasks に登録しなければならない．よって，実行時に演算部が始まる前にプログラムを中断して PID を得るなどしなければ，正確にメモリ使用量を制限できない．本実験では，プログラム内に別プロセスを起動してはじめに制限するプロセスの PID を取得することで完全にメモリ使用量を制限した．)

A.2 キャッシュフラッシュ方法

本付録では，実験において提案手法の効果をよりはっきり検証するためにキャッシュフラッシュを行った．以下にその方法について示す．LinuxOS では，/proc/sys/vm/drop_caches に書き込みを行うことで，クリーンなキャッシュ，ダーティオブジェクト，inode をメモリ上から外し，そのメモリを解放する．具体的には，/proc/sys/vm/drop_caches の値を 3 にすることで，ページキャッシュとダーティオブジェクトと inode を解放する．また，sync を実行することで，ディスク上のデータをメモリと同期させる．この 2 つのコマンドを実行することでキャッシュをフラッシュした．本研究における実験では試行前に，以下のコマンドによってキャッシュをフラッシュした．

```
$ sync
$ echo "3" > /proc/sys/vm/drop_caches
$ sync
```

まず 1 行目のコマンドでまずメモリとディスクを同期させ，2 行目のコマンド実行でメモ

リキャッシュが解放され，最後に3行目のコマンド実行によりディスク上のデータキャッシュが解放されることになる．