分散システムにおける高速なプロセス移送方式

Fast Process Migration Method for Distributed System

増田 峰義

平成 13 年 2 月 9 日

分散システムでの動的負荷分散を実現し，システム資源の利用効率を向上させる方法として，ノード間でのプロセス移送がある．プロセスの移送時，プロセスは実行を停止する．複数のプロセスを用いて並列にジョブを実行する場合，長期にわたるプロセスの停止は，協調するプロセスの実行を阻害し，ジョブ全体の実行効率を低下させる．そのため，プロセスの移送方式には，プロセスの停止時間を短縮することが求められる．本稿では，移送時のプロセス停止時間の短縮を目的とした，二つのプロセス移送方式について述べる．

一つ目の方式は，移送時にプロセスの受け入れ側で行なわれる，プロセスの生成処理を最適化した方式である．初期化済みプロセスのプールを事前に用意しておき，プールからプロセスをフェッチすることでプロセス生成を行なう．これにより，プロセス生成処理の大半を占める，データのアロケートおよび初期化処理を省略することができ，プロセス停止時間を短縮できる．プロセスの初期化およびプールへの返却は，プロセスの終了時あるいは移送時に行なわれる．この方式を，現在，我々が開発している分散 OS Colonia 上に実装し，移送時における OS のオーバーヘッドを測定した．その結果，移送先ノードでオーバヘッドを 36%低減できた．

もう一つの移送方式として，差分移送方式の提案を行なう．差分移送方式は，プロセスの実行に不可欠なスタック ・ページの転送を差分で行なう方式である．差分のみ転送するため，データの転送量を低減でき，プロセスの停止時間を短縮することができる．スタック差分の抽出には，いくつかの実装方法が考えられる．本稿では，twin/diff による方法，ハードウェア支援による方法，プログラミング言語に依存する方法について述べる．

Process migration between nodes, which realize efficiently use of computation resource, is a method for dynamic load balancing on distributed system. Process suspends, when migrated. If parallel job execution uses several processes, long time suspention of a process may prevent cooperative process from execution, and as a result, entire job execution go slow. So process migration method, which realizes short process suspention, is required. This paper shows two process migration methods aims at short process suspension.

One method optimizes process creation phase on destination node. Preparing pool of initialized processes, process creation can be done by fetch from pool. This optimization omits most of process creation steps such as data allocation and its initialization, and process suspention become short. When process finished or process is emigrated, process is initialized and retuned to process pool. We implemented this migration method to distributed Operating System Colonia which we now develope, and evaluate operating system overhead of process migration. As a result, overhead at destination node is reduced by 36 %.

This paper propose diff process migration as the another method. Diff process migration translate diff of stack page which is indispensable for process resuming. Decrease of data translation can reduce process suspention time. Creating diff of stack is implemented in several way. This paper discribes three method, twin/diff method, hardware support method, and programing language dependent method.

目 次

1 はじめに **2**

2 背景 **3**

2.1 プロセス移送の目的 3

2.1.1 公平性 3

2.1.2 効率性 3
2.2 従来型のプロセス移送方式 4
2.2.1 プロセスのコピーの生成 4
2.2.2 プロセスの状態情報 4
2.2.3 移送処理の流れ 4
2.3 移送方式の評価ポイント 5
2.3.1 評価ポイント 5
2.3.2 プロセス停止時間短縮 6
2.4 テリトリ ・スケジューリング 6
2.4.1 チケット 6
2.4.2 テリトリの形成 6
2.4.3 テリトリの大きさ 6
2.4.4 テリトリの拡大/縮小 7
2.4.5 テリトリ内の負荷分散 7
2.5 関連研究 7

3 コンピュータ ・コロニーにおけるユニット移送の実装 7
3.1 コンピュータ ・コロニー 8
3.1.1 Colonia の概要 8
3.1.2 ミッション-ユニット ・モデル . . 8
3.1.3 共有メモリ管理 9
3.1.4 GDS 9
3.1.5 データ構造 10
3.2 実装の方針 11
3.2.1 初期化処理の最適化 11
3.2.2 オンデマンド ・ページ転送 12
3.3 ユニット移送 12
3.3.1 移送元ノードでの処理 12
3.3.2 移送先ノードでの処理 12
3.3.3 レコードの pack/unpack 処理 . . 13
3.3.4 ローカルレコードの作成 13
3.3.5 ページ移送 14
3.4 評価 14
3.4.1 環境 14
3.4.2 結果 14

4 差分移送によるプロセス移送の高速化 15
4.1 差分移送の基本概念 15
4.1.1 プロセス ・ドメインの形成 15
4.1.2 実行プロセスの変更 16
4.1.3 差分移送方式 16
4.1.4 スケジューリング 16
4.1.5 二種類の移送方式 17
4.2 差分生成に関する考察 18
4.2.1 スタック差分 18
4.2.2 twin/diff 方式による差分抽出 . . 19
4.2.3 ハードウェア支援による差分抽出 20
4.3 プログラミング言語に依存する高速化 . . 21
4.3.1 トップ ・フレームの即時転送 . . 21
4.3.2 リターン時のトラップ 22
4.3.3 間接アクセスへの対処 22
4.4 今後の方針 22

5 まとめ 22

1 はじめに

最近，分散システムの形態をとりながら高い計算能力を持つ計算機システム，クラスタシステムが注目を集めている．安価な汎用品で構成できるクラスタシステムは，優れた価格性能比を実現する．そのため導入がしやすく，急速に普及しつつある．また，性能面においても，スーパーコンピューティング分野のランキングへも顔を出し始める [1] など，台頭著しい．このように，クラスタシステムは，High Performance Computing の分野で，その位置を確立しつつある．

クラスタシステムは，スーパーコンピューティング分野の流れと，分散コンピューティング分野の流れが融合してできた，新しい計算機システムである．スーパーコンピューティング分野の流れは，ピーク性能の追及を目的とし，分散コンピューティング分野の流れは，能率，快適性を目的とする．クラスタシステムの将来的な目標は，これら二つの流れの長所を併せ持つシステムを実現することである．これは，分散システムにおけるスーパーコンピューティングを目指すものである．このような理想的なシステムを実現するために，システムが満たすべき性質として以下のものを挙げる．

(1) マルチ ・ユーザ環境

ユーザの快適なシステム利用のためには，複数のユーザが同時にシステムを利用できる環境は必須である．現在のスーパーコンピュータの利用形態に見られる，バッチ処理，シングル ・ユーザ環境，空間分割による擬似マルチ ・ユーザ環境は，実行する場としての環境に比重を置いたものであり，開発を行う環境として快適とは言えない．

(2) SSI(Single System Image)

SSI とは分散システム全体を，一つの仮想的なコンピュータとして見せる技術である．この仮想コンピュータは，システムが持つ計算資源を総和した計算能力を持つスーパーコンピュータである．ネットワーク透過なファイルシステムなどは快適性を実現する．また，システム全体として持つ膨大な計算資源を統合し，高い性能を提供する．

(3) 公平な資源分配 (Fair Share)

マルチ ・ユーザ環境においては，計算資源の分配に公平性が必要となる．ユーザ間で，資源分配に不公平が生じるシステムは，快適とは言えない．

現在のところ，上記の性質を全て満たす分散システムは実現できていない．スーパーコンピューティング分野からのアプローチとしての種々のクラスタシステムにおいて，クラスタリングによる分散化のメリットは，ほとんど価格性能比のよさだけに留まっている．スーパーコンピュータの代替品として登場したクラスタの利用形態は，今のところスーパーコンピュータのそれと変わらない．これは，パフォーマンスを重視する考え方から，変動要素の多いマルチ ・ユーザ環境での利用に対して消極的なためである．逆に，分散コンピューティング分野からのアプローチにおいては，従来の分散システムの流れを汲むため，プロセスレベルのマルチ ・ユーザ環境はある程度実現されている．しかし，システム全体にわたるスケジューリングは不十分であり，システムが持つ計算機資源の有効利用は実現できていない．

これらのアプローチがうまくいかない原因は，計算資源を分配する機能が，公平性と効率性を両立できないためである．つまり，マルチ ・ユーザ環境を実現することで発生する，動的な負荷の変動に対応しきれず，ジョブに対する効率的な資源分配ができないのである．したがって，求められているのは，マルチ ・ユーザ環境での負荷変動に対して，柔軟かつ迅速に対応して，計算資源の分配を行う技術である．

分散システムでの計算資源の分配には，通常，ノード間でのプロセス移送が用いられる．プロセス移送とは，ノード間の負荷の不均衡を動的に緩和するために，負荷が高いノードから低いノードへプロセスを移送する技術である．

プロセス移送の研究対象としては，

- どのプロセスをどのノードへ移送するのか，を決める移送ポリシー，と
- 実際に移送を行なう移送メカニズム

の二つがある．移送ポリシーには，負荷の変動に対して柔軟に対応し，システム全体にわたるスケジューリングを行なうことが求められる．一方，移送メカニズムには，移送ポリシーでの決定を効率的に反映すること，つまり，ポリシーから見て使い手の良い道具であることが求められる．本稿では，このような移送メカニズムとして，二種類の移送方式の提案を行なう．

本稿の構成は以下の通りである．まず，2で，本研究の背景および関連研究について述べる．3では，一つ目の移送方式として，移送先での処理を最適化したプロセス移送方式について述べる．次に，4において，二つ目の移送方式である，差分によるプロセス移送方式について述べる．最後に，5でまとめを行なう．

2 背景

2.1 プロセス移送の目的

プロセス移送は，ノードの負荷に応じて，計算資源の動的な分配を行なうための機能である．資源の分配を行なう上で，留意すべき点は以下の二点である．

- マルチ ・ユーザ環境での，計算資源分配の公平性
- パフォーマンスの向上のための資源分配の効率性

2.1.1 公平性

マルチ ・ユーザ環境での計算資源の分配では，ユーザ間の公平性が保たれなければならない．ユーザは，例えば支払った課金に応じてシステムから計算資源を分配されジョブの実行にあてる．各ユーザに対して，資源を公平に分配することは，システムの責任である．

システムは，ユーザ間の公平性を維持するために，プロセスを移送を用いてユーザに対する資源割当てを調整する．あるユーザへの資源割当てが課金で定められた量を下回る場合，システムは，そのユーザに対してより多くの資源を割当てるために，ユーザのプロセスをより負荷の小さいノードへ移送する．

2.1.2 効率性

システムの持つ計算資源を有効に使用するために，プロセス移送を使う．システム内に負荷の低いノードが存

在すると，そのノードへプロセスを移送し，計算資源を無駄無く活用する．システム内の資源を有効に活用し，各ジョブに分配することで，ジョブを効率的に実行できる．

プロセスの生成時に，負荷の低いノードを選択してプロセスを生成することで，資源を有効利用する方法もある．しかし，この方法では，プロセス生成後の負荷の変化に対応できないため，動的な負荷の変化に対応するためには，プロセスの実行途中にプロセスを移送することが必要となる．

2.2 従来型のプロセス移送方式

ここでは，従来用いられてきた，一般的なプロセス移送方式の流れについて簡単に説明する．

2.2.1 プロセスのコピーの生成

プロセスの移送とは，あるノードで動作中のプロセスの実行を止め，別のノードで実行を再開させる処理である．ただ，移送とは言いながら，実際に行っている処理は，ネットワークを経由して，移送先ノードにおいてプロセスのコピーを生成することである．移送先ノードでプロセスのコピーを生成し，移送元ノードの古いプロセスを削除する．外からこの様子を眺めた場合，プロセスが移動しているように見える．

プロセスのコピーは，プロセスの状態情報から生成できる．プロセスの状態情報は，プロセスの本質である．プロセスから状態情報を取り出し，別のプロセスの状態情報を上書きしてやれば，同じ状態を持つプロセスのコピーを作成できる．したがって，移送先でプロセスのコピーを生成するには，移送元ノードから移送先ノードへプロセスの状態情報を転送してやればよい．移送先ノードでは，新規に生成したプロセスに転送されてきた状態情報を適用すれば，移送元ノードでのプロセスと全く同じプロセスを生成できる．

以上をまとめると，プロセスの移送手順は以下のようになる．

1. プロセス状態情報を取り出し
2. 状態情報を移送先ノードへ転送
3. 状態情報からコピープロセスを生成
4. 移送元ノードの，古いプロセスを削除

2.2.2 プロセスの状態情報

移送処理では，プロセスの状態を規定する全ての情報をプロセス状態情報として引出す必要がある．プロセス状態情報には，プロセスが動作中に行なう様々な活動の状態が記録される．プロセスの活動は多様であり，その中には他のプロセスとの通信，ファイル操作などの状態も含まれる．

プロセスの状態情報を以下にまとめる．

カーネル・リソース

システムが持つ，プロセス管理用のデータ構造である．古典的な UNIX では，プロセス構造体，ユーザ構造体にあたる．その他にファイルを使用している場合，そのディスクリプタなどが含まれる．

アドレス空間

スタック・ページ，ヒープ・ページなど，プロセスが使用しているメモリである．

通信状態

他のプロセスと通信をしている場合，通信の途中の状態，つまり通信バッファの内容などもプロセスの状態情報に含まれる．

2.2.3 移送処理の流れ

典型的なプロセス移送の実装例を以下に示す．移送の様子を図 1に示す．

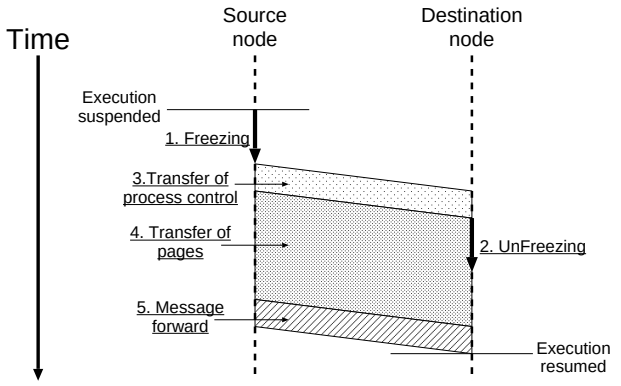

図 1: 移送処理の手順

① プロセスの凍結 (Freezing)

移送元ノードにおいて，プロセスのカーネル ·リソースを取り出し，一つのメッセージにまとめる．この処理をプロセスの凍結 (Freezing) と呼ぶ．また，以降，プロセス状態情報に変更が発生しないように処置する．まず，プロセスが実行されないように，実行キューから外す．また，外部からメッセージが届くと状態情報が変化するため，メッセージをペンディングしておくための準備をする，

② カーネル ·リソースの転送

Freezing 処理によって，一つのメッセージにまとめられたカーネル ·リソースを移送先ノードへ転送する．

③ プロセスの解凍 (UnFreezing)

移送先ノードにおいて，Freezing 処理と逆の処理を行ない，移送先ノードにおいて，プロセスを再開可能な状態にする．具体的には，ページテーブルの生成などの，移送されてきたプロセスが実行を再開するための準備を行なう．この処理をプロセスの解凍 (UnFreezing) 処理と呼ぶ．

④ ページの転送

プロセスが使用するページを移送先ノードへ転送する．ページの転送の方法については，以下に挙げるように，様々な方法が提案されている．

一括転送方式

プロセスが使用する全てのページを，移送時に転送する方式である [3]．移送先において，効率的にプロセスを実行できるが，転送するページの数に応じて，移送にかかる時間が増大するため，多数のページを使用するプロセスの移送に時間がかかる欠点を持つ．

プリページング方式

プロセスのワーキングセットを考慮に入れ，ページを一括して転送する方式である [4]．ただし，無駄なページ転送が発生する可能性がある．

デマンドページング方式

移送時に転送するページは小数に留め，残りのページは移送先ノードで必要になった際に，その都度転送する方式である [13]．無駄なページ転送は起きないが，移送先でのプロセス実行の効率は低下する．

⑤ メッセージのフォーワード

移送元ノードでペンディングしておいたメッセージを，移送先ノードへフォーワードする．

2.3 移送方式の評価ポイント

現在，色々なプロセス移送方式が提案されている．それらを互いに評価，比較するための指標を以下にまとめる．さらに，評価の項目の内，我々が重視する項目と，その理由について述べる．

2.3.1 評価ポイント

透過性

プロセス移送が透過であるとは，プロセスが移送されることで，移送されるプロセス自身，およびそのプロセスと関連のあるプロセスに影響が及ばないことである．具体的に満たすべき項目を以下に挙げる．

- プロセス移送中に発生する例外を処理するためのコード等を，一切プロセスに追加しない．
- プロセスの実行が位置透過性である．つまり，移送されるプロセスから見て，実行環境がどのノードでも同じであることである．移送元ノードと移送先ノードで，プロセスが異なる振舞いをしてはプロセス移送の透過性は実現できない．
- 移送の操作は，ユーザのプロセスから見えない．

効率

プロセス移送の効率は，重要な評価指標である．効率的にプロセス移送を行なえば，移送に伴うプロセス停止時間を短縮することができ，システム全体のスループットが向上する．効率を決定する要素を以下に示す．

- 移送プロセスの選択にかかる時間
- 移送先ノードの選択にかかる時間
- 移送先で，移送されてきたプロセスのための実行環境を整える時間
- プロセスのデータ転送にかかる時間

信頼性

プロセス移送中のノードの故障に対して，適切な処理を行なう信頼性が求められる．故障によって，移

送中のプロセスが失われたり，誤った情報を持つプロセスが他のプロセスに影響を与えることがあってはならない．

大規模なシステム，例えば地球シミュレータなどでは，信頼性を向上させるためにプロセス移送が行なわれる．このような目的で移送を行なう場合，プロセス移送自体の信頼性が重要となる．

2.3.2 プロセス停止時間短縮

並列プロセスの移送を設計する場合，2.3.1節で挙げた評価項目の内，効率の項目の，特に移送に伴うプロセス停止時間を可能な限り短縮するアプローチが最適であると考える．これは，移送されるプロセスの実行効率よりも，ジョブ全体の実行効率の向上を重視するためである．

一般に，並列プロセスのプログラミングでは，ロック機構，同期機構などを多用し，プロセス同士で共有するメモリ・オブジェクトの保護を行なう．そのため，ロック解放待ちや同期待ちなどで，プロセスが別のプロセスをブロックすることが頻繁に発生する．

他のプロセスをブロックしているプロセスを移送する場合，移送によるプロセス停止時間分だけブロックの解除が遅れる．そのため，プロセスの停止時間が，移送されるプロセスだけでなく，ブロックされている他のプロセスにまで反映される．したがって，移送による長時間のプロセス停止は，ジョブ全体の実行を阻害することになる．ゆえに，並列プロセスの移送の場合には，よりプロセス停止時間を短くする方式が適していると考える．

2.4 テリトリ・スケジューリング

我々は，公平性と効率性を両立する資源配分を実現するための，システム全域にわたるジョブ・スケジューリング方式として，テリトリ・スケジューリングを提案する．

システム内の各ジョブは，課金に応じてチケットをもらい，その実行環境として，チケットの多寡に応じた大きさのテリトリと呼ばれるノード集合を形成する．つまり，テリトリが大きいほど使用できる計算資源が多い．システムは，各テリトリの大きさを公平に保つことで公平性を，テリトリ内で動的な負荷分散を行なうことで効率性を実現する．以下では，チケットおよびテリトリについて詳述する．

2.4.1 チケット

テリトリについての説明を行なう前に，チケットについて述べる．チケットは，システムが計算資源を公平に分配するための指標である．ユーザは，課金に応じてシステムからチケットをもらう．ユーザは，もらったチケットを，更に各並列ジョブに対して割当てる．システムはジョブの持つチケットに応じて公平に計算資源を分配する．計算資源分配の偏りは，プロセスを移送することで是正される．

2.4.2 テリトリの形成

テリトリとは，システム内で動作している各並列ジョブが，ジョブごとに形成するノード集合のことである．並列ジョブは，並列実行可能な複数のプロセスからなる．各プロセスを異なるノードで実行することで，ジョブを並列に実行することができる．システムを構成するノードの内，同一のジョブに属するプロセスが動作しているノードの集合をテリトリと呼ぶ．

システムは，関連性の高いプロセス同士を，できるだけ近隣のノードへ配置する．これは実行効率を向上させるためである．例えば，二つのプロセスが同一メモリ領域にアクセスする場合，それらを近隣のノードへ配置することで，アクセス・レイテンシを低減できる．また，頻繁に通信を行うプロセス同士を，同一のノードに配置することで，通信時間の短縮およびネットワーク・トラフィックの低減を実現できる．

このように，同じジョブに属するプロセスは，互いに協調することが多いため物理的に近くのノードに配置され，異なるジョブのプロセスはできるだけ異なるノードに配置される．したがって，システム内に複数の並列ジョブが投入された場合，各ジョブごとに，物理的にまとまった領域にテリトリを形成する．この様子を図 2 に示す．

2.4.3 テリトリの大きさ

各ジョブのテリトリの大きさは，ジョブの持つチケットによって決まる．システムは，現在システムに投入されているジョブのチケットの総和に対する，各ジョブのチケットの割合に応じて，計算資源を分配する．したがって，割合が大きいジョブは大きなテリトリを形成し，小さなジョブのテリトリは小さくなる．ジョブが，割当てられた資源を使い切らない場合，そのジョブのテリトリの大きさは必要なだけのものとなる．残りの計算資源は，他のジョブに対して公平に割当てられ，他のジョブのテ

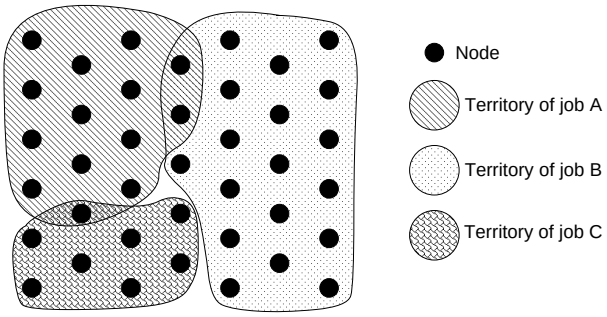

図 2: テリトリ

リトリが大きくなる．このように，システム内にアイドル ・ノードが存在しないようにする．

2.4.4 テリトリの拡大/縮小

テリトリの拡大/縮小は，次のような場合に発生する．

ジョブの生成/終了時

ジョブの生成/終了時に，大規模なテリトリの拡大/縮小が起きる．新しくジョブが生成された際，そのジョブが持つチケット相応のテリトリが形成される．同時に，その他のジョブのテリトリは縮小される．逆に，ジョブが終了した際には，それまでそのジョブが占めていたテリトリが持つ計算資源を，その他のジョブに対して公平に割当てる．

公平性の調整

テリトリの拡大/縮小は，ジョブの生成/終了時以外でも，資源分配の公平性の微調整を行なうために起きる．システムは，チケット以上に計算資源を消費したジョブのテリトリを縮小し，その他のジョブに割当てる．したがって，各テリトリの境界付近では，テリトリをまたがったプロセス移送が行なわれる．

2.4.5 テリトリ内の負荷分散

形成されたテリトリ内でジョブを効率的に実行するために，テリトリ内で動的な負荷分散を行なう．

テリトリ内で，各ノードの負荷は分散される．つまり，システムは，テリトリ内の各ノードの負荷ができるだけ等しくなるようにする．テリトリ内のノードの負荷が大きくなると，負荷の不均衡を緩和するために，プロセス移送を行なう．プロセスの移送は，テリトリ内のノードの負荷が均等になるまで続く．

2.5 関連研究

プロセスの移送に関する関連研究を紹介し，我々のプロセス移送に対する想定を明らかにする．

並列プロセス / 単独プロセス

我々は，単独プロセスの移送ではなく，他のプロセスと協調して動作する並列プロセスの移送を考える．単独プロセス移送の研究の一つに，LOCUS[3] でのプロセス移送がある．LOCUS での移送方式は，ページを全て移送時に転送するという，単純な方式であったため，プロセスの停止時間が長時間にわたるという欠点があった．プロセスの停止時間の短縮を狙ったプロセス移送の研究としては，V-System[4] でのプロセス移送がある．V-System での移送方式は，プロセスの実行と，ページの転送をオーバーラップさせることで，プロセス停止時間の短縮を図ったものである．並列プロセスであるスレッドの移送の研究には，MILLIPEDE[5]，Ariadne[6]，Active Thread[14] などがある．

カーネルレベル ・プロセス / ユーザレベル ・プロセス

前述の，スレッド移送の研究の内，MILLIPEDE はカーネルレベルのスレッド，Ariadne と Active Thread はユーザレベルのスレッドの移送の研究である．ユーザレベル ・スレッドの長所は，移植性の高さとコンテクスト ・スイッチの速さである．逆に，ページ ・フォールトやシステム ・コールの発行などで，カーネルのサービスを受ける間，全てのスレッドが停止してしまう．MILLIPEDE で採用されているカーネルレベル ・スレッドでは，スレッドごとにカーネルのサービスを受けられるので，他のスレッドの動作を阻害することはない．

我々は，カーネルレベルのプロセスの移送を想定している．

3 コンピュータ ・コロニーにおけるユニット移送の実装

本章では，2.2.3節で示した従来型のプロセス移送方式に対して，プロセス停止時間の短縮を目的とした改良を施した移送方式について述べる．この方式を，現在我々が開発しているコンピュータ ・コロニー上に実装し，移送処理における OS のオーバヘッドを評価した．

まず，分散 OS Colonia の概略と，従来型の移送方式が持つ欠点および改良の方針についてまとめる．次に，改良した方式の，分散 OS Colonia 上への実装について述べる．最後に，エミュレータ上で行った評価を示す．

3.1 コンピュータ・コロニー

3.1.1 Colonia の概要

我々は，並列処理による高速計算を実現し，かつ複数のユーザが気軽に自分のマシンのように自由に使える，価格性能比の高い計算機システムとして，コンピュータ・コロニーを開発している．コロニー (Colony) とは，群体を意味する．群体とは，複数の個体が集合して 1 個体のように生活する生物の状態のことである．同様に，コンピュータ・コロニーは，複数のマシンが集合して 1 つの大きな高性能マシンとなることを目標とする．

コンピュータ・コロニーは分散共有メモリを支援するハードウェアと，分散 OS Colonia からなる．コンピュータ・コロニーは，大きさの様々な計算機が高速ネットワークで接続された形の分散システムである．

コンピュータ・コロニーの構成モデルを図 3に示す．

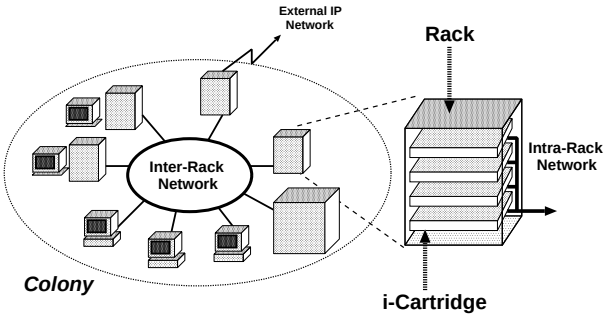

図 3: Computer Colony のモデル

システムの各ノードとなる計算機は，i-Cartridge と呼ばれるモジュール・カードと，それを収める **Rack** で構成される．i-Cartridge は，プロセッサとキャッシュ，主記憶およびネットワーク・インターフェイスを内蔵し，一つのコンピュータとして動作可能である．システムに i-Cartridge を追加することで，性能の増強が可能である．

i-Cartridge

コンピュータ・コロニーの最も重要な構成要素は，「**i-Cartridge**」(individual:個体) と呼ぶカートリッジである． i-Cartridge は，プロセッサとキャッシュ (それぞれ複数あってもよい)，それに主記憶，ネットワーク・インターフェイスを内蔵する．i-Cartridge は 1 つのコンピュータとして動作可能である．

Rack

Rack は，i-Cartridge を収める Bay，Rack 内の i-Cartridge 間を結ぶ **Intra-Rack Network**，システム内の Rack 間を結ぶ **Inter-Rack Network** へのインターフェイス，その他の I/O スロット，電源装置を内蔵する．デスクトップ・モデルでの i-Cartridge Bay の数は，1 ~ 4 個程度，サーバ・モデルでは数十 ~ 数千程度である．

3.1.2 ミッション-ユニット・モデル

コンピュータ・コロニー上で動作する分散 OS Colonia では，プログラミング・モデルとして，ミッション-ユニット・モデルを提供する．

ミッション-ユニット・モデルは，一般的な共有メモリ上の並列アクティビティモデルであるタスク-スレッド・モデルを，分散共有メモリ環境向けに拡張したモデルである．ミッションはタスクに，ユニットはスレッドに相当する．ユニットが複数ノードで動作することで，ミッションを並列実行する．システムは，ノードの負荷状況に応じて，ユニットの投入，移送を行い，負荷を動的に分散させる．移送に伴うユニットの位置の変更は，Global Directory Service(GDS) [12] が管理する．

このモデルでは，タスク-スレッド・モデルと同様に，プログラミング環境として共有メモリを提供する．ただし，分散環境下での共有領域の管理コストを低減するため，全アドレス空間ではなく，アドレス空間の一部を共有する．

ミッション-ユニット・モデルでは，以下の 3 つのアクティビティを定義する．

ミッション

タスク-スレッド・モデルのタスクに相当し，複数のユニットで構成される．ユニットが異なるノードで動作することでミッションが並列に実行される．ミッションは，システム内で一意な識別子 (ミッション ID) を持つ．

ユニット

タスク-スレッド・モデルのスレッドに相当し，並列実行の単位である．ユニットは，ミッション内で一

意な識別子 (ユニット ID) を持つ．また，プロセッサ割り当ての単位としてスレッドを持つ．

スレッド

プロセッサ割り当ての単位であり，プロセッサ ・コンテキストを持つ．スレッドはノード ・ローカルなアクティビティであり，ノード内で一意な識別子 (スレッド ID) を持つ．

3.1.3 共有メモリ管理

タスク-スレッド ・モデルでは，同一タスクに属するスレッドは全アドレス空間を共有するが，ミッション-ユニット ・モデルでは，アドレス空間を部分的に共有する．

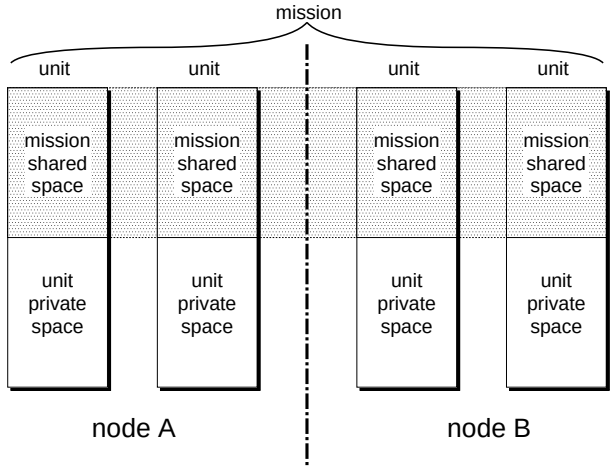

図 4: アドレス空間の構成

図 4のように，ユニットのアドレス空間を二つの領域に分ける．一つは，ユニット固有領域(unit private space)と呼び，スタックなどのユニットに固有なデータを配置する．もう一方の領域は，ミッション共有領域 (mission shared space) と呼び，同一ミッションに属する全てのユニットで共有する領域である．ミッション共有領域には，ユニットが共通に使用するデータ，およびコードなどを配置する．

アドレス空間を二つの領域に分けたのは，管理コストを軽減するためである．タスク-スレッド ・モデルでは，全てのスレッドはアドレス空間を共有する．そのため，スレッドが固有に使用する領域を確保する際にも，他のスレッドとの調整および，そのための通信が必要となる．ミッション-ユニット ・モデルでは，ユニット固有の領域を設けることで，このような不要なコストを削減する．

Colonia は，ミッション共有領域という形で，ユーザに対して共有メモリを提供する．共有領域はページ単位で管理され，Shared Virtual Memory(SVM) [11] を構成する．

共有領域の各ページには，それぞれホームページと呼ばれるページが存在し，ホームページをリモートノードのメインメモリへキャッシュしたページをコピーページと呼ぶ．また，ホームページが存在するノードを，そのページのホームノードと呼ぶ．リモートノードで，共有ページのリードミスが発生した場合，ホームページからデータをフェッチする．ホームノードには，コピーページを管理するディレクトリが存在し，共有ページへのライトは，ホームノードを経由して一貫性制御が行われる．

ホームページは，ノードではなくユニットに対して割り付けられる．割り付けられたユニットを，そのホームページのホームユニットと呼ぶ．ホームページは，ホームユニットと同じノードに存在する．したがって，ホームノード = ホームユニットが動作しているノードである．ページとホームユニットの関係は，ホームユニット ・テーブルによって記述され，GDS (3.1.4節参照) がその管理を行う．

3.1.4 GDS

GDS(Global Directory Service) は，ユニットの位置管理といった，システム ・グローバルなオブジェクトを管理するサービスである．GDS は，システム内の全てのユニットごとに，レコードと呼ばれる管理情報を作成し，ユニットを移送すると，レコードの内容を変更する．

ユニットの移送が行われ，移送先でユニットが継続実行可能になると，移送先ノードのシステム ・ユニットは GDS が持つレコードを移送ユニットの ID で検索し，レコード内の位置情報を変更する．ユニットの位置情報を知りたい場合も同様に GDS を ID で検索する．GDS は，システム内の任意のノードから参照することができる．

GDS が管理する情報は以下の通りである．

識別子 (ID)

ミッション，ユニットなどグローバルなアクティビティに与える，システム内で一意な識別子の管理を行う．

ユニットの位置管理

動的な負荷分散のため，ユニットは適宜移送される．

GDS は，ユニットのシステム内での位置情報を提供する．

ホーム ・マッピングテーブル

共有領域の各ページには，ホームユニットが割り付けられる．ページとユニットのマッピングの関係は，GDS が持つホームマッピングテールによって，各ミッションごとに管理される．

3.1.5 データ構造

Colonia のアクティビティ管理のためのデータ構造を図 5に示す．

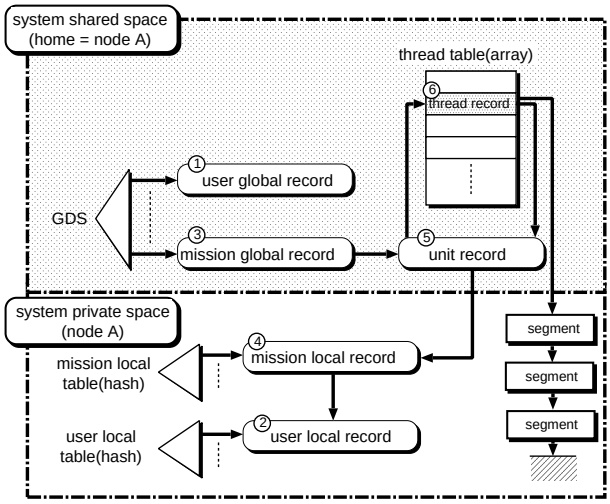

図 5: ミッション ・ユニットの管理構造

図の各四角は，データ構造体を表し，矢印はポインタによるリンクを表現している．図中下側の領域は，ノード A のカーネルが管理を行なうアドレス空間である．三角形のオブジェクトは，ハッシュなどの検索テーブルを表している．図中上側の領域は，各ノードごとのシステムのカーネルで共有するアドレス空間である．

ここで，ユーザ/ミッションの拡大/縮小の定義を行う．

ユーザの拡大

ユニットの移送もしくは新規作成により，これまでそのユーザのユニットが動作していなかったノードにそのユーザのユニットが生成される．

ユーザの縮小

ユーザの拡大と逆に，移送もしくは終了によりノードからそのユーザのユニットが全てなくなる．

ミッションの拡大

ユニットの移送もしくは新規作成により，これまでそのユニットのミッションが動作していなかったノードにミッションが生成される．

ミッションの縮小

ミッションの拡大とは逆に，移送もしくは終了によりノードからそのミッションのユニットがなくなる．

以下，図中 (1) ~ (6) のデータについて，ユーザ管理，ミッション管理，ユニット/スレッド管理に分けて説明する．

ユーザ管理

ユーザの管理データには，グローバルなものと，ノード ・ローカルなものがある．グローバルレコードは，ユーザがログインした時点で作成される静的なデータであり，ローカルレコードは，グローバルレコードを基に，各ノードの状況に応じて変化する動的なデータである．それぞれのレコードが格納する情報を以下に示す．グローバルレコードは，ユーザがシステムにログインした時点で生成され，ローカルレコードは，ユーザの拡大時に生成され，縮小時に解放される．グローバルレコードの取得には，GDS をユーザ ID で検索する．ローカルレコードの取得には，ユーザ ・ローカルテーブル (ハッシュ) をユーザ ID で検索する．

① ユーザ ・グローバルレコード:(184Byte)

- ユーザ名，グループ ID，ユーザが持つチケット情報など．

② ユーザ ・ローカルレコード:(96Byte)

- ノード内に存在するこのユーザのミッションが持つチケットの総量
- 上記ミッションの内，アクティブなミッション[1]が持つチケットの総量
- ユーザの CPU 使用状況の記録

[1] アクティブなユニットを持つミッションを指す．

ミッション管理

ミッション管理データにも，ユーザの管理データと同様，グローバルなものと，ノード ・ローカルなものがある．それぞれのレコードが格納する情報を以下に示す．

③ ミッション ・グローバルレコード:(20Byte)

- ホームユニット ・テーブルへのポインタ
- ミッションが持つチケット

④ ミッション ・ローカルレコード:(28Byte)

- ノード内に存在するこのミッションに属するユニットが持つチケットの総量
- 上記ユニットの内，アクティブなユニット[2]が持つチケットの総量
- ミッション共有領域のページテーブルへのポインタ

グローバルレコードは，ミッション生成時にシステム内で一つ生成される．ローカルレコードは，ミッションの拡大時に生成され，縮小時に解放される．グローバルレコードの取得には，GDS をミッション ID で検索する．ローカルレコードの取得には，ミッション ・ローカルテーブル (ハッシュ) をミッション ID で検索する．

ユニット/スレッド管理

ユニットおよびスレッドの管理情報は，ユニット ・レコード，スレッド ・レコードに格納される．ユニットの移送の際に，これらのレコードは移送元ノードから移送先ノードへ転送される．それぞれのレコードの内容を以下に示す．

⑤ ユニット ・レコード:(60Byte)

- ユニットが持つチケット，ユニットの状態など．

⑥ スレッド ・レコード:(228Byte)

- プライオリティなど，スケジューリングのための情報
- レジスタ ・コンテクスト
- 仮想アドレス空間管理のためのセグメント・テーブル．図 5に示すように，このテーブルはリスト構造である．

[2]アクティブなユニットとは，実行キューに繋がれているユニットを指す

3.2 実装の方針

ユニット移送の設計方針を以下にまとめる．

3.2.1 初期化処理の最適化

2.2節で述べたように，ユニットの移送の処理は，移送先ノードで新しくユニットを生成する処理である．

一般に，プロセスの生成には，メモリ ・オブジェクトのアロケートとその初期化処理が多く含まれ，ユニット生成処理中の大半の時間を占める．実際に，Colonia 上のプロセス生成処理では，約 70%が初期化処理で占められていた．したがって，初期化処理にかかるコストを低減することは，ユニット停止時間の短縮に有効であると言える．

初期化処理自体はキャッシュ ・ヒット率の悪い処理である．新規にアロケートした領域はたいていキャッシュ上に存在しないため，初期化の際にコールドミスが多発し，初期化に時間がかかる．

さらに，初期化したとしても，移送元ノードから送られてくるユニットの状態情報の内，即座に上書きされてしまうデータも存在する．このようなデータの初期化は無駄である．そこで，次のような最適化を行なう．

初期化済みオブジェクトのプール

ユニット生成処理の大半を占める初期化の処理コストを低減するために，メモリ ・オブジェクトのアロケートおよびその初期化処理を行なわず，事前に済ませておくことを考える．

初期化処理には，送られてきたユニット状態情報と無関係なものが多い．このような処理は，事前に済ませておく．初期化済みメモリ ・オブジェクトのプールを用意しておき，使用時には，単にフェッチするだけで済ます．初期化処理は，オブジェクトの返却時やブート時に行なう．例えば，ページ ・テーブルのエントリをクリアする処理などは，送信されてくるユニット状態情報と依存性がないため，移送開始前に行うことができる．このように，移送時に行わなければならない処理量を減らす．

初期化済みユニットのプール

メモリ ・オブジェクトの再利用する考えをさらに進め，ユニット自体の再利用を行なう．これにより，ユニット生成の処理の多くを省略する．

初期化済みメモリ ・オブジェクトを用意しておくように，初期化済みのユニット，空ユニットを用意しておく．空ユニットのプールを作成し，ユニット生成およびユニット移送時に，プール内から一つをフェッチする．移送の場合には，移送元ノードから転送されてきたユニット状態情報を，空ユニットのものの上に上書きする．

ユニットのプールはブート時などに用意する．終了したユニットや，移送後のユニットは削除されることなく，初期化されプールに返却される．

3.2.2 オンデマンド ・ページ転送

プロセスの実行に必要なページの転送を，移送時に一度に行なう場合，プロセスの停止時間が増大する．プロセスが使用しているページの全てが，移送先ノードで必要とは限らない [13] ため，移送時には，移送先で即座に必要となる最低限のページのみ転送し，残りのページは必要に応じて転送する方式 (オンデマンド ・ページング方式) を採用する．移送時に転送するページ数を制限することで，停止時間の短縮を図る．

通常，オンデマンド ・ページング方式でページの転送を行う場合，移送先でページが未転送ページの転送が必要となる度に，プロセスの実行がブロックされるため，プロセスの実行効率は悪くなる．しかし，移送する並列プロセスが共有メモリを持つ場合，例えばスレッドの場合には，単独プロセスほど実行効率は低下しないと予想される．これは，並列プロセスが共有メモリを持つ場合，移送先ノードに共有ページがあることを期待できるからである．

3.3 ユニット移送

本節では，前節で示したデータ構造をふまえて，移送時のデータ処理実装の詳細について述べる．

3.3.1 移送元ノードでの処理

移送元では，移送するユニットの実行を停止し，ユニットの状態情報を取り出し，移送先ノードへ転送するメッセージを作成する．また，ユニットのスタックページや，必要であればコードページの転送準備，移送中に届くメッセージのフォーワード設定を行なう．

主な処理内容を以下にまとめる．

1. detach 処理

   移送選択したユニットが実行中であれば停止し，実行キューから外す．

2. pack 処理

   ユニット ・レコードおよびスレッド ・レコードなど，ユニットの状態情報をメッセージに詰め移送先ノードへ転送する．

3. ページ転送

   スタックページ等，移送先ノードで即座に必要となるページを転送する．

3.3.2 移送先ノードでの処理

移送先ノードでの処理は，転送されてきたユニットを再開可能にするための処理である．基本的な処理内容を以下に示す．

1. ローカルレコードの作成

   移送先でユーザ/ミッションの拡大が起きた場合，ローカルレコードを作成する．

2. ユニットの生成

   空ユニットのプールから空ユニットを一つ獲得する．プールに空ユニットのストックが存在しない場合，新規にスレッドを生成する．

3. unpack 処理

   ユニットの状態情報を受信し，中からレコードを取り出し，生成したユニットのレコードを上書きする．

4. ページの受け入れ処理

   空きページ領域を確保し，転送されてきたページを格納する．

移送先ノードでの基本的な処理の流れは，上記の通りであるが，以下に示す条件によってはいくつかの処理を省くことができる．

ユーザの拡大が発生するか

発生しない場合，ユーザ ・ローカルレコードの生成処理を省くことができる．

ミッションの拡大が発生するか

発生しない場合，ミッション ・ローカルレコードの生成処理を省くことができる．

空ユニットがあるか

空ユニットがある場合，ユニットの生成処理を省くことができる．

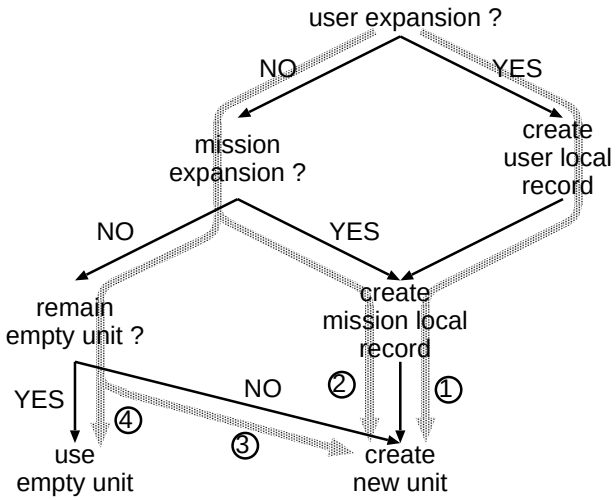

図 6: 移送先ノードでの処理手順

以上をまとめると，処理手順は，図 6に示すように 4 通りに分岐する．それぞれの場合での処理内容を以下に示す．

① 場合 1

ユーザの拡大が発生する場合，必ずミッションの拡大も発生する．また，空ユニットはミッションごとに管理されているため，ミッションの拡大が発生する場合，空ユニットも存在しない．したがって，ユニットの生成処理も行なう必要がある．

② 場合 2

ユーザの拡大は発生しないが，ミッションの拡大が発生する場合，つまり，ユーザの他のミッションは存在する場合である．ミッション ・ローカルレコードの生成および，場合 1 と同様の理由によりユニット生成処理も行なう．

③ 場合 3

ユーザ/ミッション共に拡大が発生しないが，空ユニットのストックが存在しない場合である．ローカルレコードの生成は行なわず，ユニットの生成処理のみ行なう．また，場合 3 の場合，ミッション共有領域の設定処理を行なう．ノード内でのミッション共有領域は，ユニットがページ ・テーブルを部分的に共有することで実現する．そのため，新規にユニットを生成する場合 3 では，既存のページ ・テーブルとの共有処理を行なう．

④ 場合 4

ユーザ/ミッション共に拡大が発生せず，空ユニットが存在する場合である．ユニットの生成処理も省くことができる．

3.3.3 レコードの pack/unpack 処理

レコードの pack 処理とは，移送元ノードにおいてユニットの状態情報，つまりユニット ・レコードおよびスレッド ・レコードを転送メッセージに詰める処理である．逆に unpack 処理は，転送メッセージからレコードを取り出す処理である．

pack/unpack 処理では，ポインタ ・データと，そうでないデータは区別して扱う．なぜなら，移送元ノードでのポインタは，移送先ノードでは意味をなさないデータだからである．

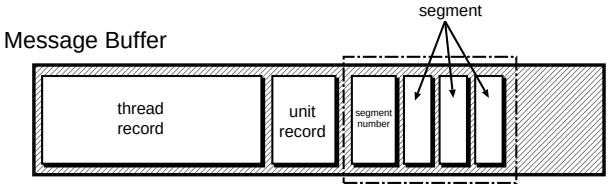

図 7: pack/unpack 処理

例えば，セグメント ・テーブルのようなリスト構造を pack 処理する場合，リスト ・データ転送用のコンテナ構造体を定義し，リストの各要素と共にリストの要素数も添えて転送する (図 7)．移送先ノードでの pack 処理の際には，コンテナからリストの各要素を取り出し，ポインタの貼り替えを行ないリストの再構築を行なう．

3.3.4 ローカルレコードの作成

移送先ノードの状況において，

- ユーザの拡大が発生する
- ミッションの拡大が発生する

場合，ユーザ/ミッション・ローカルレコードの作成を行う．当然，拡大が発生しない場合はレコードを作成する必要がなく，処理自体を省くことができ，移送コストを低減できる．

拡大が発生するかどうかは，ユーザ/ミッション・ローカルテーブル (ハッシュ) を検索することで判明する．検索して見つからない場合，拡大が発生する．

スケジューラのポリシーを工夫することで，これらの処理を省く最適化が考えられる．すなわち，長期的な戦略から，テリトリの周辺ノードに対して，あらかじめミッションを拡大させておく，これにより，拡大の発生によるローカル・レコードの作成処理を移送時の処理から確実に省くことができる．

3.3.5 ページ移送

ユニットの移送に伴い，移送先ノードへ転送する必要のあるページは以下の通りである．

- ユニット固有領域のページ
- ホーム・ユニットを担当しているホームページ

これらのページの転送方法として，Colonia では，移送時にはスタックなどの必要最低限のページのみ転送し，残りのページは Copy on reference 方式で転送する．

移送時の転送

移送時に転送するページは，参照頻度で選別する．すなわち，スタックが配置されている固有領域のページ，参照頻度が高い共有領域のページなどが最初に転送される．転送ページの選別には，TLB のエントリをチェックするなどの実装方法などが考えられる．

残りのページの転送

Colonia では，移送時には最低限のページしか転送しないため，転送しなければならないページが移送元に残る．これらのページは，以下の方法で移送先に転送される．

- 移送先で必要になった時に転送

  移送先で，未転送のページへアクセスが発生した場合，システム・ユニットが移送元へページの転送要求を出す．移送元のシステム・ユニットがこの要求を受信し，ページの転送を行う．ページ転送ごとに起動オーバーヘッドがかかるため，移送時に一括してページ転送を行う方式と比べて，トータルでの転送時間は長くなる．これを軽減するために，ページのプリフェッチといった技術が考えられる．

- 移送元のカーネルによる転送

  Colonia では，各ノードごとに，残されたページを転送するためのシステム・スレッドを起動しておく．このシステム・スレッドは，適宜スケジューリングされ，移送先へ残りのページを転送する．

3.4 評価

3.4.1 環境

Colonia のユニット移送機構をエミュレータ上に実装し，移送元および移送先での処理にかかる，OS によるソフトウェア・オーバヘッドを測定した．エミュレータの実行には Sun Microsystems 社の WS，Ultra-30(UltraSPARC-II 296MHz) を使用した．コンパイラには gcc，最適化オプションは -O2 を用いた．実行サイクル数の測定には，tick レジスタを用いた．

3.4.2 結果

評価結果を表 1に示す．数値はサイクル数である．なお，計算量の参考のために，キャッシュヒット率がほぼ 100%の理想値も併記した．() でくくった数値が理想値である．理想値は，同一処理を繰り返し行い，一回あたりの平均として算出した．

移送先ノードでの処理は，3.3節で示した順序 1 ~ 4 の各場合について評価を行なった．なお，移送ユニットのデータサイズは，ユニット・レコードとスレッド・レコードの他にセグメントを 16 個 (合計 448Byte) 持たせて，合計で 716Byte とした．

評価結果より，移送時のソフトウェア・オーバヘッドは，移送元ノードで約 11μs，移送先ノードで約 22μs ~ 約 14μs であった．この評価結果は，コールドなキャッシュ状態でのサイクル数なため，実際にはこの数値よりも小さくなることが期待できる．順序 3 ではページ・テーブルの共有処理を行なうため順序 2 よりもサイクル数がかかった．順序 4 の場合，プールからフェッチすることで，

表 1: サイクル数

移送元処理	共通			
(1) スレッド detach	1141 (124)			
(2) スレッド pack	1842 (593)			
(3) ユニット pack	334 (131)			
小計	3317 (848)			
移送先処理	**場合 1**	**場合 2**	**場合 3**	**場合 4**
(1) ユーザ ・ローカルレコード	898 (109)	442 (40)		
(2) ミッション ・ローカルレコード	1630 (112)		311 (41)	
(3) ユニット生成処理	1135 (173)			374 (42)
(4) スレッド unpack	2338 (637)			
(5) ユニット unpack	866 (134)			
(6) ミッション共有 ページ ・テーブル	-	-	1691 (366)	-
小計	6867 (1165)	6411 (1096)	6783 (1391)	4331 (894)

ユニット生成処理を省略し，他の場合に比べて 761 サイクル高速化することができた．

4 差分移送によるプロセス移送の高速化

本章では，プロセス移送時のデータ転送を差分で行なう移送方式について述べる．

4.1 差分移送の基本概念

4.1.1 プロセス ・ドメインの形成

プロセス ・ドメインとは，プロセスごとに形成される，プロセスの活動領域である．あるプロセスのプロセス ・ドメインに含まれるノードには，それぞれ一つづつ，そのプロセスのコピーが存在する．ある瞬間には，プロセスのコピーの内，ただ一つだけが動作する．このプロセスを実行プロセス (executing process) と呼ぶ．実行プロセス以外の，残りのプロセスのコピーをシャドウ ・プロセス (shadow process) と呼ぶ．シャドウ ・プロセスは全て実行を停止している．

プロセス ・ドメインは，プロセスを移送することで形成される．2.2節で述べたように，通常，移送元ノードのプロセスは，移送先ノードにプロセスのコピーを生成した後に削除されるが，プロセス ・ドメインの形成時には，プロセスを削除せずにシャドウ ・プロセスとして残しておく．図 8に，プロセス ・ドメインが形成される様子を示す．この図では，構成ノードが node A のみのプロセス ・ドメインが，プロセスを移送することで，node B にまで拡大される様子を示している．移送元である node A のプロセスは削除されず，シャドウ ・プロセスとして残る．

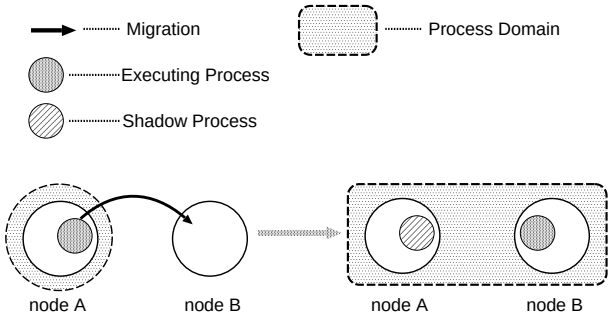

図 8: プロセス ・ドメインの形成

プロセス ・ドメインの形成過程を見ればわかるように，

シャドウ ・プロセスとは，プロセスの古い状態情報を保存したものである．シャドウ ・プロセスが生成された直後は，シャドウ ・プロセスと実行プロセスの状態情報は同じであるが，実行プロセスの方は，実行されることで状態情報が変化する．一方，シャドウ ・プロセスは停止状態であるため，状態情報は変化しない．

4.1.2 実行プロセスの変更

システムは，ノードの負荷変動に応じて，シャドウ ・プロセスとの間で実行プロセスの変更を行なう．プロセス ・ドメインが複数のノードで構成される場合，すなわち，一つの実行プロセスといくつかのシャドウ ・プロセスが存在する場合を考える．実行プロセスが動いているノードの負荷が高くなると，システムは実行プロセスの変更を始める．シャドウ ・プロセスの中から次の実行プロセスとなるプロセスを一つ選び，

- 選んだシャドウ ・プロセスを実行プロセスに変更
- 現在の実行プロセスをシャドウ ・プロセスに変更

を行なう．

シャドウ ・プロセスが実行プロセスとなるためには，シャドウ ・プロセスが保存している古い状態情報を，最新の状態情報へ更新してやる必要がある．最新状態への更新は差分で行なう．実行プロセス側からシャドウ ・プロセス側へ，状態情報の差分のみを転送し，差分とシャドウ ・プロセスの状態情報をマージし，最新の状態情報とする．

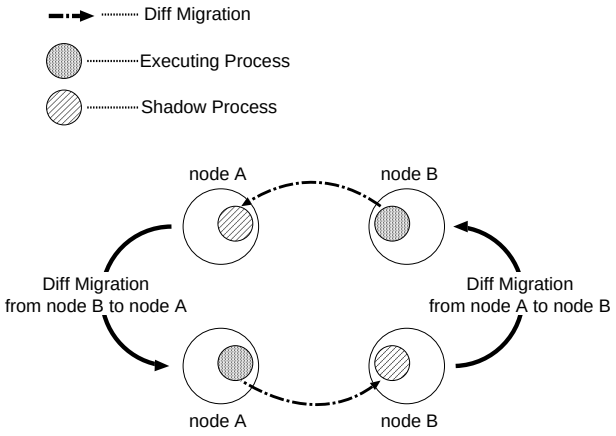

図 9: 実行プロセスの変更

実行プロセスの変更は，一種のプロセス移送であり，変更に伴うプロセス状態情報の更新作業は，差分で行なうことができる．したがって，この処理を差分移送と呼ぶ．差分移送によって，実行プロセスの変更する様子を図 9に示す．この図では，node A と node B の間で，交互に実行プロセスを変更している様子を表している．

4.1.3 差分移送方式

差分移送方式のメリットは，以下の二つの理由により，通常のプロセス移送方式に比べ，プロセス停止時間を短縮できることである．

プロセス生成処理の省略

移送差分方式では，移送先ノードにシャドウ ・プロセスが存在する．そのため，新規にプロセスを生成する必要がない．したがって，移送時にプロセス生成にかかる処理を行なわなくて済み，プロセスの停止時間を，通常のプロセス移送よりも短縮することができる．

データ転送量の低減

移送時に転送するデータは，シャドウ ・プロセスと実行プロセスとの状態情報の差分であるため，データ転送量を低減できる．これにより，データの転送にかかる時間が短かくなり，停止時間を更に短縮することができる．

更に，この移送方式の特徴として次のことが挙げられる．

一時的な移送に向いている

差分移送方式は，一時的に他のノードへプロセスを出す場合に有効である．もし他のノードでの実行が長期にわたる場合，プロセス状態情報の変更が大きくなり，データ転送量を減らすメリットが少なくなる．また，移送先ノードでの変更分の抽出，および移送元ノードでのマージ処理にかかるコストが大きくなり，通常の移送処理以上にプロセス停止時間が長くなる可能性もある．

4.1.4 スケジューリング

プロセス ・ドメインやテリトリは，スケジューラが移送先ノードを決定するための，有益な情報となる．なぜ

なら，移送にかかるコストおよび，移送後のプロセス実行効率に関して，ある程度の予測ができるからである．したがって，プロセスの移送先を決定するグローバル ・スケジューリングの設計には，これらのノード情報を考慮に入れる必要がある．

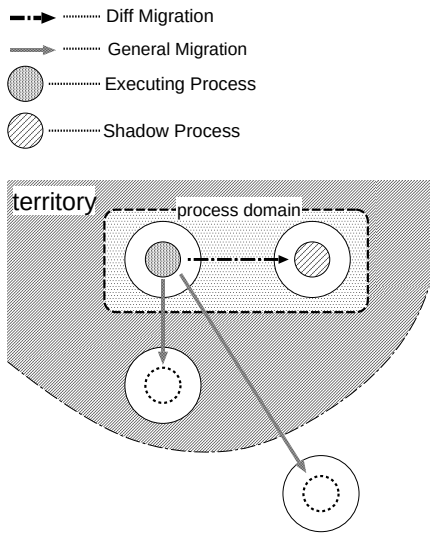

図 10: 移送先ノードの選択

図 10に示すように，移送先ノードの選択としては，

- プロセス ・ドメイン内のノード
- テリトリ内のノード
- テリトリ外のノード

が考えられる．それぞれの特徴を以下にまとめる．

プロセス ・ドメイン内での移送

移送方式は，シャドウ ・プロセスへの差分移送となるため，高速に移送できる．また，プロセス ・ドメインに属するノードは，一度はそのプロセスが実行されたことがあるノードである．したがって，ドメイン内のノードには，プロセスの実行に必要なページなどが残っている可能性があり，ドメイン外のノードに比べて，高いプロセス実行効率が期待できる．

プロセス ・ドメイン内での移送では，移送先ノードが限定されるため，ノード選択の自由度は低い．反面，選択の幅が狭い分，スケジューリング ・コストは低い．

ドメイン内のノードの負荷が全て高い場合，プロセス ・ドメインを形成するメリットを活かすことができない．このような事態を避けるために，シャドウ ・プロセスを，あらかじめ負荷の低いノードへ移送させておくなどの方法が考えられる．

テリトリ内への移送

周囲の状況などを考慮に入れ，テリトリ内でのプロセスの最適配置を行なう．他のプロセスとの協調関係などを考慮して移送ノードを決定する．ノード選択の自由度が高い分，スケジューリング ・コストは高くなる．

移送後にシャドウ ・プロセスを残す場合，プロセス ・ドメインが拡大される．再び元のノードへ戻る可能性が少ないと判断する場合，シャドウ ・プロセスは残さない．

テリトリ外への移送

テリトリ外への移送は，主に他のテリトリとの境界付近で行なわれる移送である．テリトリの拡大や，公平性の調整などに用いられる．

4.1.5 二種類の移送方式

システムは，プロセス移送時の他のプロセスの配置状況，これまで行なってきたプロセス移送の経過といった，その場その場の局面に合わせて，通常の移送方式と差分移送方式を使い分ける．

このような方法は，ただでさえ複雑な移送メカニズムが更に複雑になるデメリットがあるが，可能な限りプロセス停止時間を短縮し，計算資源を有効に活用するためには，有効なアプローチであると考える．

通常のプロセス移送方式は，以下に示す局面のように，再び元のノードへ戻る可能性が少ない場合に用いる．

ジョブの生成/終了時

ジョブの終了時，そのジョブのテリトリがなくなり，テリトリの空白地ができる．空白地は，残りのジョブに公平に分配され，それぞれのジョブのテリトリが拡大する．この時に発生する大規模なプロセス移送では，再び元のノードへプロセスが戻る可能性は低いため，通常のプロセス移送が用いられる．ジョブの生成時も同様である．

新方式の移送は，一時的なプロセスの移送に適する．このような局面として，以下のようなものが挙げられる．

公平性の微調整

計算資源分配の，ジョブ間での公平性を保つためにプロセス移送が行なわれる．公平性を保つための移送では，計算資源をもらいが少ないジョブのテリトリから，もらいの多いジョブのテリトリへプロセスを移送する．移送されたプロセスは，計算資源のもらいが公平になるまで実行され，その後，もとのテリトリに戻される．このような移送パターンは，新方式で実現するのに適している．

一時的な負荷の不均衡の解消

ノード間の負荷の不均衡には，一時的なものがある．例えば，並列プログラムで多用される同期処理の直後である．あるテリトリ内での負荷が均等な状態で，プロセス群が同期ポイントに向けて処理を行なっているとする．同期ポイントに至ったプロセスから次々と同期待ち状態になり，まだ動いているプロセスの数によりノード間で負荷の不均衡が発生する．しかし，同期ポイントを過ぎると，プロセスが一斉に起きため，この不均衡は一時的なものである．このような負荷の不均衡の緩和のために，プロセス移送を行なった場合，同期ポイント後にまた負荷の不均衡が発生するため，再度，負荷を緩和するための移送が必要となる．このような一時的な負荷の不均衡の緩和には，新方式での移送が適している．つまり，プロセスの動作は，同期ポイント前に別のノードへ移送され，同期ポイント直後に再び元のノードへ戻す．戻る際の移送は，通常の移送よりも高速に行なうことができる．

また，通常の移送方式と，新方式を組み合わせることもできる (図 11)．

テリトリの拡大

組み合わせての移送は，二回移送を行なうので，移送コスト自体は通常のプロセス移送に比べて大きくなる．しかし，対象プロセスを停止させることなく，移送先ノードでの実行準備を整えることができる，という利点がある．そのため，この組み合わせはテリトリの拡大の際などの場面で用いると効果的である．通常，テリトリが拡大される時，ノード内のチケット配分などの処理が伴なう．通常移送でテリトリが拡大した場合，この処理コストがプロセス停止時間に直接反映される．これを避けるために，シャドウ ・プロセスを移送させた後，そのプロセスを実行プロセスに変える．テリトリを拡大させるのは，シャドウ ・プロセスの移送であるので，プロセスの実行に影響が出ない．

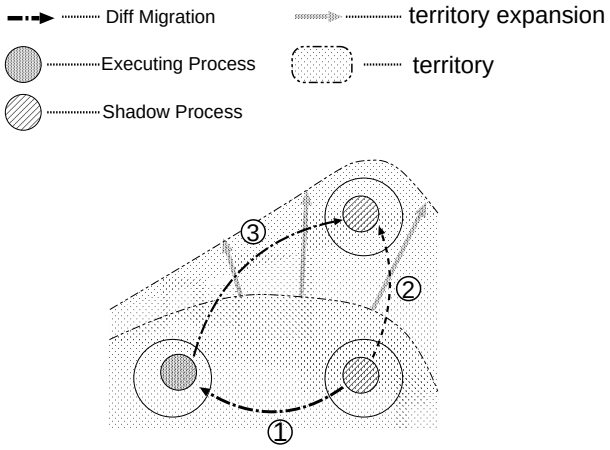

図 11: 両方式の組み合わせ

4.2 差分生成に関する考察

差分移送方式の差分の対象として，スタックを考えている．以下では，スタックを対象とする理由および，その差分抽出の実装方法をいくつか提案する．

4.2.1 スタック差分

スタック転送量の低減

プロセスの状態情報には，2.2節で示したように，いくつかのデータの集合である．差分移送では，これらのデータの内，スタックを差分をとる対象とし，スタックの転送量の削減することで，プロセス停止時間の短縮を図る．

プロセスの実行には，スタックが必須である．そのため，プロセス移送に伴うページ転送の方式として，必要なときにページ転送を行なうオンデマンド ・ページング方式を採用したとしても，スタック ・ページは即時移送する必要がある．したがって，スタック ・ページの転送にかかる時間は，移送時のプロセス停止時間に直接反映される．

3での，プロセス停止時間の短縮を目的として，移送元ノードでの UnFreezing 処理にかかる処理の最適化を行ない，プロセス停止時間を短縮することができた．しかし，これらの処理にかかる時間が短縮された結果，プロセス停止時間に占める，スタックの転送にかかる時間が目立つようになった．実際，Active Thread[14] では，スタックの転送時間がプロセス停止時間の大部分を占めている．

したがって，差分によるプロセス移送により，スタックの転送量を抑えることは，プロセス停止時間の短縮に有効であると考える．

スタックの構造

一般的なスタックの構造について説明する．スタックは，スタック ・フレームと呼ばれる，関数ごとに生成されるデータ構造が積上げられたものである．スタック ・フレームには，関数内のローカル変数の格納領域や，関数のリターン ・アドレス等のコンパイラが生成する管理データなどが含まれる．関数が call されると，スタックのトップに新しくスタック ・フレームが push され，関数から return すると pop される．現在実行中の関数に対応するスタック ・フレームを，Current Stack Frame と呼ぶ．

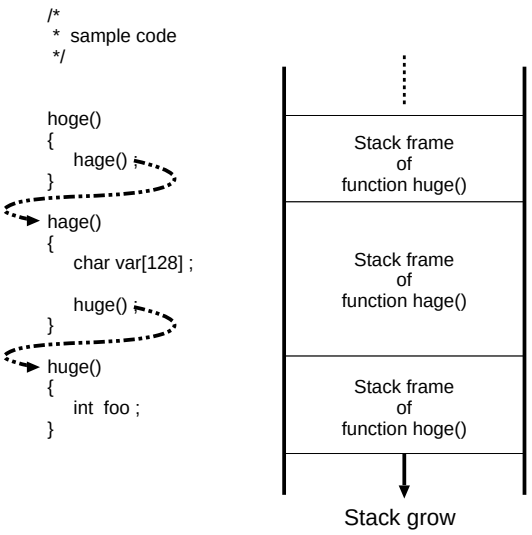

図 12: スタックの構造

関数呼び出しのパターンはプロセスに依存するものであるが，通常，ある程度の呼出しパターンを示すものと考える．すなわち，関数の呼出しパターンは，呼び出しばかりを繰り返した後，リターンばかりを繰り返すといったように激しく乱高下するものではなく，ある特定の関数をベースとして，関数の呼出し，リターンを行なう．これは，Register Window が想定する関数呼出しのパターンと同じものである．

基本的に，スタック ・フレームは Current Stack Frame である時に内容を変更される．そのため，プロセスがこのような関数呼出しのパターンを示す場合，スタックの底部に近いフレームほど，変更される頻度が小さくなる．つまり，スタックの差分をとった時に，残る可能性が少ない．

4.2.2 twin/diff 方式による差分抽出

スタックの差分を抽出する方法として，twin/diff [15],[16] を用いる方法について述べる．

twin/diff

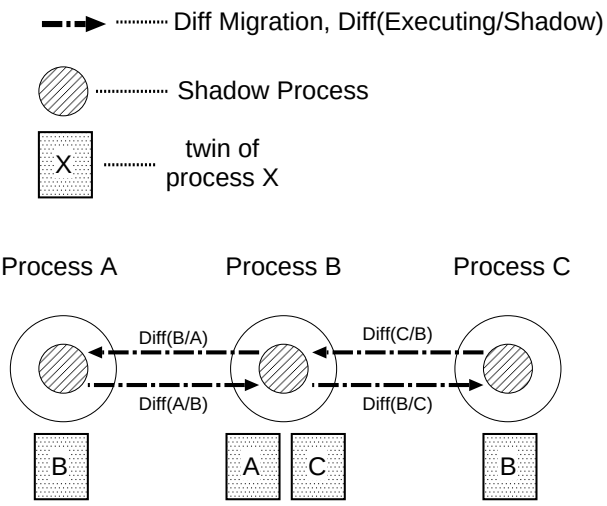

図 13: twin によるスタック差分移送

twin とは，スナップショットのことである．移送されてきた直後，つまり，シャドウ ・プロセスから実行プロセスに切り替わった直後のスタック ・ページの内容を，twin として保存しておく．再び他のノードへプロセスを移送する際，つまり，再び実行プロセスからシャドウ ・プロセスに切り替わる際には，保存しておいた twin と，現在のスタック ・ページの内容を比較して，差分 diff を作成する．

この様子を図 13に示す．図では，あるプロセスが三つのプロセス (Process A ~ C) を持ち，その内の一つが実行プロセスとして動作している．図では Process A と Process B，Process B と Process C の間で実行プロセ

スの切り替えが行なわれる．差分による移送であるため，Process A と Process C の間の切り替えはできない．差分移送を行なうためには，相手 Process の twin が必要となる．したがって，Process A と Process C は Process B の，Process B は Process A,C 両方の twin を保持する．

twin/diff の問題点

twin/diff の問題点として，twin を保存しておく領域が別途必要となることが挙げられる．twin のための領域は，シャドウ ・プロセスの数に比例して増える．

また，diff 生成にかかるコストに関する問題がある．ソフトウェアによって diff を生成する場合を考える．diff の生成には，twin と現在のスタックとをシーケンシャルに比較する必要がある．ところが，twin は実行プロセスの切り替え時にしか使用しないため，比較の際にはたいていメインメモリ上にあり，キャッシュへのヒットが期待できない．また，スタックの内容も，直前のプロセス実行でアクセスされなかった部分はキャッシュにヒットしない．そのため，比較処理中にコールド ・ミスが多発し，高速な処理ができない．したがって，diff 生成処理を高速に行なうためには，キャッシュ ・ミスすることを前提としたプログラムの最適化を行なう必要がある．

結局のところ，diff をとることが有効かどうかは，メインメモリとネットワークの転送能力で決まる．ネットワークがメインメモリと同等以上に高速であれば，diff などとらずに，そのまま転送すればよい．しかし，現在のところ，メインメモリの方がネットワークよりも高速なので，差分による移送は有効といえる．

コンパイラ支援

twin のスタック ・トップを，現在のスタック ・トップが下回った場合，twin 上部のスタック ・フレームが，フレームごと無効になる．無効なフレームを特定できれば，その部分に関して diff を生成する必要がなくなるため，その分 diff 生成を高速化できる．

コンパイラの支援により，twin 内の無効なスタック ・フレームを即座に確定することを考える．関数からリターンする際に，twin のスタック ・トップと現在のスタック ・トップを比較し，twin のスタック ・トップを下回れば，twin 内の有効なスタック ・トップの値を更新するコードを挿入する．これにより，プロセス実行中に，twin 内の有効なスタック ・トップの値を記録できる．

4.2.3 ハードウェア支援による差分抽出

ハードウェアのサポートにより，diff の生成を高速に行なうことを考える．RHiNET[17] のように，diff の生成をハードウェアに任せる方法もある．ここでは，コンピュータ ・コロニーの各ノードに搭載された，ネットワーク ・インターフェイス (NI) が提供する，キャッシュ ・コヒーレンス制御機能を利用する方法を示す．

B_map

NI は，分散共有メモリを提供するために，ノード間でブロック単位のキャッシュ ・コヒーレンス制御を行なう．NI は，コヒーレンス制御のために，B_map と呼ぶ，メインメモリの各ブロックの状態を記録したタグを持つ．

NI は，ハードウェアによって，ノード間での高速なキャッシュ ・コヒーレンス制御を行なう．システム ・バスに接続された NI は，システム ・バスを常にスヌープし，必要性を検出すると，コヒーレンス制御を行なう．NI は，バスに流れる物理アドレスを監視し，物理アドレスに該当するブロックの状態を B_map を引いて検出する．ブロックの状態が無効である，もしくは，共有状態のブロックに対する write 操作である場合，ノード間でキャッシュ ・コヒーレンス制御を開始する．

この，NI の機能を用いてスタックの変更ブロックを記録することを考える．つまり，スタック ・ページに属するブロックに対して write 操作であれば，B_map の該当タグの状態を変更する．

スタックの変更ブロックのみを記録する手順を以下に示す．

- 初期状態を設定

  プロセスが移送されてきた直後，そのプロセスのスタック ・ページをメイン ・メモリに格納する．その際，スタック ・ページに属するブロックに対応する，B_map のエントリの状態を全てクリアする．

- 物理アドレスを監視

  プロセスを実行させると，スタック ・ページに対する変更が行なわれる．NI はバスに流れる物理アドレスを監視し，スタック ・ページに属するブロックに対する write 操作を検出し，B_map の該当タグの状態を変更する．

- 差分の抽出時

  プロセスを際移送する際，その時の B_map の内容をもとに，スタックの差分を作成する．B_map のエントリは，変更されたブロックのみ状態が変更されている．これを元に差分を抽出する．

B_map を用いた差分抽出

前述の手順により，B_map を用いて，プロセスの実行中にスタック ・ページ内の変更ブロックの記録を行なう．移送時には，プロセスのスタック ・ページに対応する B_map の内容を解釈し，スタックの差分を作成する．

NI 上には，例外的に発生する，ハードウェアだけでは対応できない，複雑な処理を行なうためにプロトコル・プロセッサが搭載されている．B_map へのアクセスは，システム ・バスを介してメイン ・プロセッサから行なうよりも，NI 上のプロトコル ・プロセッサから行なう方が効率がよい．そこで，メイン ・プロセッサではなく，プロトコル ・プロセッサにスタック ・ページの差分生成処理を行なわせる．また，プロトコル ・プロセッサに差分生成処理を行うのと同時に，メイン ・プロセッサは他の移送のための処理を行なう．つまり，移送処理を並行してを行なうことができる．

以上をまとめると，実際に差分を抽出する手順は，以下のようになる．

1. Colonia は，移送プロセスのスタック ・ページのアドレスと共に，NI に差分生成命令コマンドを発行する．

2. Colonia からの命令を受け，NI はプロコル ・プロセッサに割込みをかける．プロトコル ・プロセッサが起動し，変更ブロックを特定するために，対応する B_map の内容を逐次解釈し，差分を生成する．差分をメッセージを移送先へ転送する．

3. 移送先ノードで，転送されてきた差分メッセージを受信する．移送先ノードでも，差分処理のために，移送元ノードと同様に NI 上のプロトコル ・プロセッサを起動させる．プロトコル ・プロセッサは，差分メッセージを解釈し，シャドウ ・プロセスのスタック ・ページに適用し，スタック ・ページは最新の状態に更新される．

4.3 プログラミング言語に依存する高速化

ここでは，今後の改良案として，java 言語などのように，変数の型がプロセスの実行時でも判別できる場合に，高速なプロセス移送を実現する方法について述べる．

スタック ・トップのフレームを含めて数フレームのみを即時転送し，残りは後で転送することを考える．したがって，転送するスタックのサイズが大きい場合でも，プロセス停止時間を短くできる．この方法は，差分移送に限らず通常の移送でも適用できる．

4.3.1 トップ ・フレームの即時転送

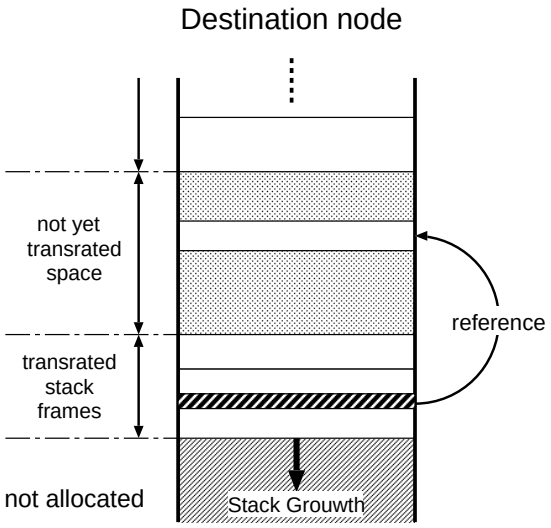

図 14: スタック ・フレームの即時転送

スタック ・トップのフレームを含めて数フレームのみを即時転送すると，移送先ノードで再構成されるスタックには，図 14に示すように未転送部分が生じるため，不完全なものとなる．この状態でプロセスを実行し，プロセスが未転送部分にアクセスした場合，未転送部分には不正なデータが格納されているため，プロセスが誤った動作をする可能性がある．したがって，プロセスが未転送部分にアクセスすることを防ぐ必要がある．

プロセスの，未転送部分へのアクセスは，関数がリターンする時，および参照渡しなどにより間接的にアクセスする場合が考えられる．それぞれに対して，以下に示す方法で対処する．

関数のリターン

関数がリターンする直前にトラップを発生させることで，OS に制御を移し，プロセスの実行を停止させる．OS はスタックの未転送部分を転送する．

間接アクセス

参照渡しでアクセスされる可能性のある部分については，その個所を特定し，移送時に転送しておく．

以下，それぞれについて詳しく述べる．

4.3.2 リターン時のトラップ

関数がリターンし，スタックの未転送部分にあるスタック ・フレームが Current Stack Frame となる直前に，プロセスにトラップを引き起こさせる．これには，リターン ・バリア [18] と同じ方法を採る．

ジョブのコンパイル時に，トラップを引き起こすコードを混ぜてリンクを行ない，ジョブの実行ファイルを生成する．未転送部分の直前のスタック ・フレームのリターン ・アドレスを，このトラップを起こすコードのアドレスに差し替える (図 15)．こうすることで，関数からリターンしようとすると，トラップ ・コードが実行され，自然な形で OS に制御が移る．OS は残りの未転送部分を転送する．

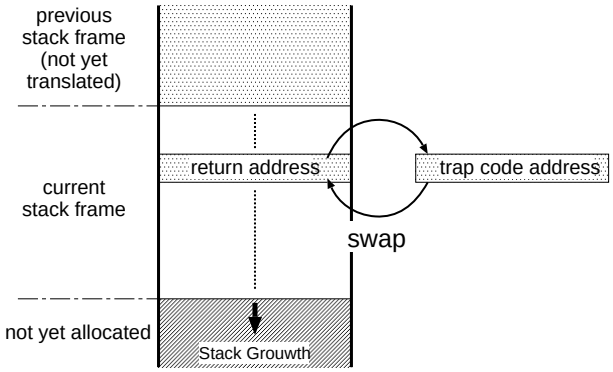

図 15: トラップ ・コードの挿入

4.3.3 間接アクセスへの対処

関数からリターンする場合以外にも，関数への参照渡し，もしくは大域変数を介したアクセスなどで，プロセスが未転送部分にアクセスする可能性がある (図 14)．これを防ぐために，間接的にアクセスされる可能性がある個所については，移送時に転送する．

フレーム内のデータ型が実行時でもわかる言語であれば，フレーム内のローカル変数が他のデータへの参照かどうかを判別できる．これを利用して，即時転送するスタック ・フレームの中身，および大域変数を調べ，未転送部分のフレーム内の変数を参照している場合，参照先の変数をそのまま，もしくはその差分も一緒に転送する．

4.4 今後の方針

本稿では，差分移送方式を提案するに留まり，その有効性を評価結果などで示すことができなかった．今後は，本章で挙げた様々な実装方法について，詳しく評価を行ない，差分移送方式の有効性を示したいと考えている．また，差分移送方式を考慮に入れた移送ポリシーの研究など，考察が不十分なところも多々あるため，今後はそれらの研究も並行して進めていきたいと考えている．

5 まとめ

本稿では，分散システムでの動的負荷分散を実現するため手段である，プロセス移送について述べた．並列ジョブのプロセスの移送方式を設計する場合，移送時のプロセス停止時間を短縮するアプローチが有効であると考え，二つの移送方式の提案を行なった．

一つは，移送先ノードでの処理を最適化し，プロセスの生成処理を高速化する方式である．初期化済みプロセスのプールを用意しておき，プロセス生成時にはプールからプロセスをフェッチするだけで済む．そのため，プロセス生成のコストがプロセス停止時間に直接反映されない．同様のプールを，プロセスを構成する各データ ・オブジェクトについても用意し，プールにプロセスのストックが存在しない場合でも，高速にプロセスを生成できる．この方式を分散 OS Colonia 上に実装し，エミュレータ上で OS のオーバヘッドの評価を行なった．

もう一つの移送方式として，差分移送方式を提案した．プロセス移送時のスタック転送量を低減し，転送コストを抑えることで，プロセス停止時間の短縮を図る．また，スタック差分を高速に抽出する方法として，いくつかの実装案を示した．本稿では，差分移送方式の提案に留まったため，今後の研究では，差分移送方式の有効性を，シミュレーションなどの方法で示したいと考えている．

謝辞

本研究の機会を与えて頂いた，本研究室の富田眞治教授に深甚な謝意を表します．

また，本研究に関して適宜御指導，御鞭撻を賜った森眞一郎助教授，五島正裕助手に深く感謝致します．

さらに，共同研究者である鳥崎 唯之氏，石川 智祥氏，鳥居 大祐氏をはじめとして，日頃様々な角度から助力して下さった京都大学大学院情報学研究科通信情報システ

ム専攻富田研究室の諸兄に心より感謝致します．

参考文献

[1] http://www.top500.org

[2] Kai Hwang, Hai Jin, Edward Chow, Choli Wang, Zhiwei Xu : Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space, IEEE Concurrency,pp.60-69,Jan-Mar,1999

[3] G. Popek and B. Walker, editors. The Locus Distributed System Architecture. M.I.T. Press, Cambridge, Massachusetts, 1985.

[4] Marvin N. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In Proceedings of the Tenth ACM Symposium on Operating Systems Principles, pages 2–12. ACM, December 1985.

[5] Itzkovitz, A., A. Schuster, and L. Wolfovich, Thread Migration and its Applications in Distributed Shared Memory Systems, . 1997, Technion IIT.

[6] E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. Software-Practice and Experience, 26(3):327–357, March 1996.

[7] 青木秀貴, 他 :共有メモリベースのシームレスな並列計算機環境を実現するオペレーティングシステムの構想, 情処研報,97-OS-34,1997

[8] 山添博史, 他 :並列アプリケーションを指向した分散システムコンピュータ ・コロニーの構想, 情処研報,97-OS-76,pp55-60,199

[9] 山添博史, 他 :分散共有メモリ上の OS における高速移送可能な並列アクティビティ,(SWoPP 下関)1997

[10] Damien Le Moal, 他 分散システムにおける Fair Share プライオリティスケジューラ, 情報研報 99-OS-80(SWoPP 下関'99),1999

[11] Kai Li, Paul Hudak :Memory Coherence in Shared Virtual Memory Systems, In the ACM Transactions on Computer Systems, Vol7, No.4, November 1989, pp321-359

[12] 増田 峰義, 他 分散 OS Colonia における共有メモリを利用した大域的ネーム ・サービス, 信学技報 99-CPSY-53(SWoPP 下関'99), pp.49-56

[13] Zayas, E. R. 1987. Attacking the process migration bottleneck. In 11th ACM Symposium on Operating Systems Principles (1987), pp.13–24

[14] B.Weissman, B.Gomes, J.W.Quittek, M.Holkamp, "Efficient Fine-Grain Thread Migration with Active Threads" 12th International Parallel Processing Symposium, March 1998, Orland.

[15] TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, Proceedings of the Winter 94 Usenix Conference, pp. 115-131, January 1994.

[16] Software DSM Protocols that Adapt between Single Writer and Multiple Writer, C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, Proceedings of the Third High Performance Computer Architecture Conference, pp. 261-271, February 1997.

[17] 工藤 知宏, 他:PC 間ネットワークによる共有アドレス空間を持つ並列処理システム. 情報処理学会研究報告 ARC, 第 21-21 巻,pp.121-126,1999.

[18] 湯淺 太一, 他:リターン ・バリア. 情報処理学会論文誌:プログラミング, Vol.41 No.SIG9(PRO 8),pp87-99,2000.