社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－115 (6)
2008／5／20

プロセスマイグレーション可能なMPIプログラムとその性能評価

矢澤 慶樹† 日比野 靖†

並列プログラムをマルチユーザ・マルチプログラミングの並列計算機で実行する場合，複数のプロセスが同一ノードで競合して実行されると並列プログラム全体で処理時間の大きな遅延が発生する．遅延を改善する方法として，実行中に各ノードの負荷を監視し，プロセスの移動を行うMPIプログラムの実装方法を提案する．プロセスの移動はMPIプログラムの内部だけで実現するため，高い可搬性が得られる．提案手法によるアプリケーションとしてラプラス方程式の求解を用いて性能評価を行ったところ，遅延の改善に有効であり，プロセスの移動によるオーバヘッドは無視できる程度であることが明らかになった．また提案手法を大規模アプリケーションに適用することを目指し，シミュレーションを行って設計上の指針となる知見を得た．

The Design and Performance Evaluation of the Process Migratable MPI Program

Yoshiki Yazawa† and Yasushi Hibino†

On a multi-user and multi-programming parallel computer, an over-wrapped execution of processes which belong to different parallel applications degrades performance severely. Even a delay of execution time on a node will influence on the entire parallel programs. To avoid this, we propose the portable design of process migratable MPI program which completes on MPI application layer. To evaluate the efficiency of the proposed method, we implemented a migratable Laplacian equation solver and compared with non-migratable solver. It reveals that the proposed method sufficiently reduces the delay caused by over-wrap and that the overhead of process migration is very low. Further more, aiming to apply the proposed method to large scale problems, we estimated the performance using our simulator under various conditions and obtained answers to the design considerations.

1. はじめに

近年，大規模な科学技術計算で広範に並列計算機が用いられており，高性能並列計算機に複数のユーザの並列タスクが投入され，同時に実行されることが多くなっている．このような環境では，異なるユーザの並列タスクの一部が同一ノードを取り合って競合的に動作することが少なくない．競合によって並列タスクを構成する部分プロセスが遅延すると並列タスク全体が影響を受けるため，同時に複数の利用者を許可する並列計算機では，部分プロセスの競合による並列タスク全体の処理時間遅延を回避する方法が必要である．

この目的で有望な方法にプロセスマイグレーションがある．プロセスマイグレーションとは，動作中のプログラムの実行を停止し，その内容を別の場所で復元することをいう．従来，プロセスマイグレーションではプロセス状態の転送にファイルを経由するなど処理オーバヘッドが大きいため，有効性が疑問視されていた[1)2)]．また，従来のプロセスマイグレーションシステムでは，プロセス状態の保存と復元にOSやミドルウェアのサポートを必要としていた[3)4)5)6)]．

並列計算機による並列処理ではMessage Passing Interface (MPI) が多く用いられているが，MPIプログラムのプロセスマイグレーションはより困難である．プロセスマイグレーションでは，プロセスの移動のために新たなプロセスを生成することが必要だが，現在広範に利用されているMPI-1ではMPIプログラム内から新たにプロセスを生成し，並列処理に組み込むことができない[7)]．またMPI-1プログラムでは，スレッドを用いて複数のコンテキストの通信を並行して行えない問題があり，計算処理中に通信で割り込んでマイグレーション動作を開始することができない．またMPIアプリケーションでは明示的なメッセージ通信が本質的に不可欠であるため，プロセスマイグレーションの後でプロセス間の通信を復元させなければならない．

MPIプログラムを対象としたプロセスマイグレー

† 北陸先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science, Japan Advanced Institute of Science and Technology

ション機構としてMPI-TM[5]，CoCheck[6]，MPI Process Swapping[8][9] 等の研究がある．これらは独自のMPI実装を行ったり，MPI実装に小規模な変更を加えて上記の問題を解決している．このようなアプローチは有効であるものの，ミドルウェアへの大規模な変更が必要であることから，商用の計算機での利用が難しいという問題点がある．

そこで本論文では，システムへの変更なしにMPIアプリケーションプログラムだけでプロセスマイグレーションを実現できる手法について提案し，この手法によるアプリケーションの性能改善を論じる．特に提案手法を大規模アプリケーションへ応用することを目指し，背景負荷の性質とマイグレーション動作の関係による性能をシミュレーションにより評価し，その結果を述べる．

2. プロセスマイグレーション可能なMPIプログラム

2.1 MPIプログラムでのプロセスマイグレーション

実行中のプロセスを別の計算機へ移動するためには，実行の中断，内部状態の保存と移動，移動先での状態の復元と実行の再開が必要である．またMPIアプリケーションでは実行中のプロセスの生成と追加，通信コンテキストに制限があるため，この点を考慮しなければならない．

そこで，並列計算処理の中断と別マシンでの処理再開に機能を絞り，通常のプロセスマイグレーションで考慮されるリソースの継続利用を省略し，並列処理で利用しているメモリ領域や変数のみを移動する．移動は通信だけを用いて行い，ファイルへの出力は行わない．プロセス生成の問題に対しては，プログラムの起動時にプロセスを冗長に確保しておき，プロセスの移動先としてこのプロセスを利用することで対応する．プロセス間通信の復元は，通信先を記録したテーブルを用意し，プロセス移動時にこのテーブルを更新し，ブロードキャストすることで行う．

MPI-1プログラムでは，スレッドを用いて複数のコンテキストの通信を並行して行えない問題があり，計算処理中に通信で割り込んでマイグレーション動作を開始することができない．そこで，定期的に全プロセスがバリア同期を行い，通信コンテキストを一斉に並列計算からプロセスマイグレーションに切り替えることでこの問題を解決する．

これらのプロセスマイグレーションに関連した処理は，chkpt() という関数に全てカプセル化し，この関数をループ中から定期的に呼び出す．

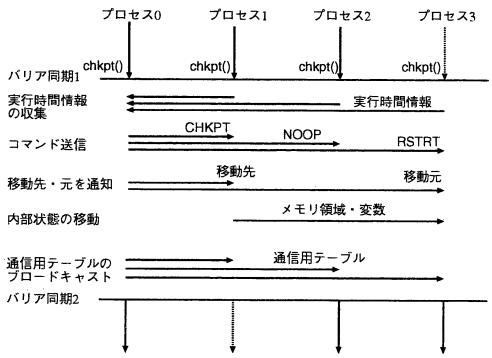

図1 chkpt() での処理

2.2 chkpt 関数

図1は高負荷ノードで動いているプロセス1から，休止状態のプロセス3へ処理を移動する場合のchkpt()関数の処理を示したものである．

全てのプロセスがchkpt()関数を呼ぶとバリア同期が成立する．この時点で通常の計算のための通信は全て終了していることが保証され，以後，chkpt()関数を抜けるまで通信はマイグレーションコンテキストで行われることが保証される．

アプリケーション内のプロセス一つがコントローラとなり，情報収集と移動条件の判断を行う．図中ではプロセス0がコントローラとなっている．コントローラでは，各プロセスから負荷情報を集め，移動元と移動先を選び，プロセスの移動条件を満たすか判断を行う．負荷情報はプロセスの進捗から求めたり，システムからload値を得るなどの方法で取得する．移動条件は，通常，移動後に現在よりも速く実行されることを目標としてプロセス数の比較によって決定する．移動条件を満たすノードが見つかった場合，コントローラはプロセスにコマンドと移動に必要な情報を送信する．移動の指示を受けたプロセス間でデータの転送が行われる．次いでコントローラから通信リダイレクトテーブルのブロードキャストが行われる．

リダイレクトテーブルを受信したプロセスは再びバリア同期を行い，同期成立後に通信コンテキストを切り替え，通常のアプリケーションコンテキストに復帰する．

3. 基本性能の解析

提案するプロセスマイグレーション手法を用いてMPIアプリケーションプログラムを実装し，プロセスマイグレーション動作の実証と基本性能の解析を

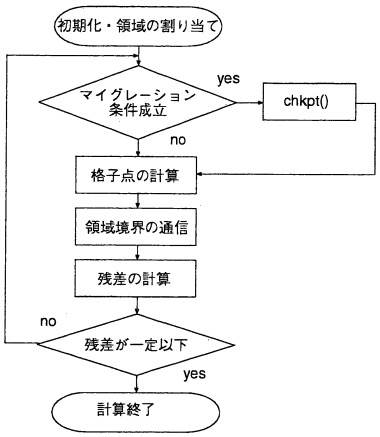

図 2　プログラムの処理の流れ

行った．

3.1　サンプルアプリケーション

提案手法を用いてラプラス方程式の差分法による反復解法を実装した．

このプログラムは，格子点に適当に与えた初期値から出発し，ある点の値を近傍 4 点の値から計算する．計算後，領域全体の残差を計算し，これが一定の閾値以下になるまで反復して値の改善を行う．並列解法では，全格子点領域を縦方向に分割した小領域を並列計算機の各ノードに割り当てる．今回の例では 1024×1024 の領域を 4 または 8 プロセスへ分割を行った．各小領域の境界では，他ノードに割り当てられた領域との値の交換が必要で，この部分で通信が発生する．計算部分のコードは文献[10] にある MPI-1 を用いた並列解法を利用している．

図 2 は実装したプログラムの処理の流れを示す．プログラムは起動後まず初期化とノードへの計算領域の割り当てを行う．このとき計算領域を割り当てないマイグレーション用スペアプロセスを確保する．ループでは計算領域の割り当てられたプロセスは各格子点の計算と境界での通信を行う．ループ中，マイグレーション開始条件が成立すると chkpt() 関数によるマイグレーション動作に切り替わる．このプログラムでは，開始条件はサンプルアプリケーションでは担当する計算領域がないか，ループ回数が 200 回としている．

3.2　実行環境と実験条件

実験には 64 台の PowerPC 604e ノードと 4 台の Power3 ノードを持つ IBM RS6000SP を用いた．実験で使用した PowerPC ノードは，1 ノードに動作クロック 332MHz の PowerPC 604e を 4 個搭載した SMP 構成となっている．

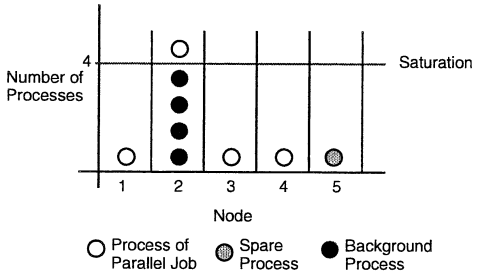

図 3　プロセス投入

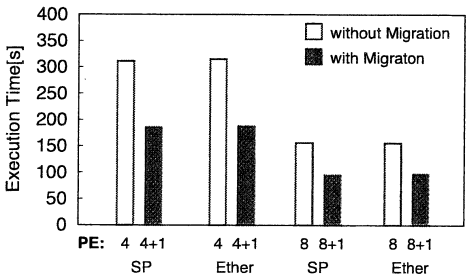

図 4　背景負荷がある場合の実行時間

MPI のプロセス間通信をネットワーク経由で行わせるため，MPI プログラムは各ノードあたり 1CPU を使用するように設定した．また通信はこの計算機で利用可能な SP Switch と 100Base-TX Ethernet を各々利用した．

3.3　プロセスマイグレーションの動作と効果

提案手法によるプロセスマイグレーション可能な MPI プログラムを，背景負荷のある環境で動作させ，プロセスマイグレーション動作を確認した．計算プロセス数は 4 または 8 で，使用ノード数はプロセスマイグレーションを行わない場合はプロセス数と同じ 4 または 8，プロセスマイグレーションを行う場合は 5 または 9 である．

背景負荷には数値積分プログラムを用いた．SMP 構成のノードの処理能力を飽和させるため，背景負荷を並列プログラムを実行するノードのうち 1 つに図 3 のように投入した．

実行時間を図 4 に示す．マイグレーションしない MPI プログラムの場合，1 ノードの実行の遅延によって並列プログラム全体に遅延が起きているが，マイグレーション可能な MPI プログラムではマイグレーションによって実行時間が大幅に改善している．

3.3.1　マイグレーションオーバヘッド

プロセスマイグレーションによるオーバヘッドを測

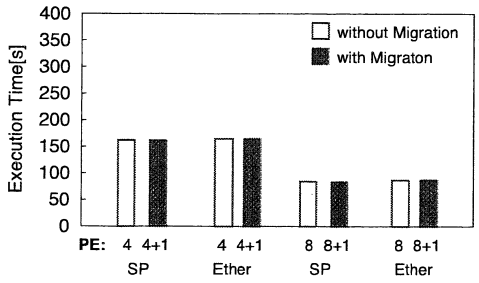

図 5 背景負荷がない場合の実行時間

定するため，提案手法によるプロセスマイグレーション可能な MPI プログラムと，マイグレーションなしの MPI プログラムについて，背景負荷を置かずに実行し，実行時間を測定した．

計算プロセス数は 4 プロセスと 8 プロセスで，使用ノード数はマイグレーションしない場合は 4 または 8，マイグレーションを行う場合は 5 または 9 である．

得られた実行時間を図 5 に示す．図中のラベルで SP は SP Switch を，Ether は Ethernet を表す．

プロセスマイグレーションするプログラムの実行時間はマイグレーションなしのものとほぼ変わらず，プロセスマイグレーションの時間的なオーバヘッドは非常に小さいことがわかる．プロセスマイグレーションに費す時間は SP Switch を使用した場合約 0.3 秒，Ethernet の場合約 0.8 秒であった．

3.4 考　　察

実験により，提案するプロセスマイグレーション手法が機能することが確認できた．スペアプロセス，リダイレクトテーブル，バリア同期を用いた通信コンテキスト切替えによって，MPI アプリケーションプログラムの内部だけで計算処理中のプロセスの内容をあらかじめ確保したプロセスに移動して実行を再開し，正しい結果を得ることができた．

マイグレーション時には内部データとして 8MB を転送しているが，転送に要する時間は SP Switch の場合 0.27 秒，100Base-TX の場合 0.73 秒となった．これは実行時間と比較して十分小さい．

提案手法では通信コンテキストを切替える必要から全プロセスで定期的にバリア同期を行う．このため，バリア同期が成立するまでの間，バリアに早く到達した負荷の軽いノードの計算能力に無駄が生じる．しかしマイグレーション開始までの間隔を適切に設定すれば，同期による影響は得られる遅延改善と比べて十分許容できる．そこでこの手法をさらに大規模な並列アプリケーションに応用することを検討することにした．

4. シミュレーションによる性能評価

4.1 目　　的

前章までの実アプリケーションによる実験では，背景負荷が一定である場合のプロセスマイグレーションの効果が示されたにすぎない．

実用的な大規模アプリケーションに提案マイグレーション手法を適用した場合の効果を見るためには，時間的に変動する背景負荷に対してマイグレーションの効果を見る必要がある．

このため，シミュレーションによって目的アプリケーションの実行時間，背景負荷の到着分布及びマイグレーション間隔に対する粒度を変化させてスペアノード数やプロセス配置及びマイグレーション頻度について適切な値を求める．

4.2 シミュレータの概要

シミュレータは複数のノードからなる並列計算機をモデル化したもので，並列プログラムの仕様記述と背景負荷の性質，マイグレーションの間隔，マイグレーションオーバヘッドを入力として取り，背景負荷と並列プログラムの実行を模擬し，終了時刻や履歴を出力する．

並列プログラムの仕様は使用ノード数，プロセスのノードへのマップ情報，アクティブプロセス数，開始時刻，実行時間からなる．背景負荷の性質は，平均到着間隔，平均処理時間で，これらの値から背景負荷がポアソン到着，指数サービス時間分布で生成される．

各ノードはキューを用いてプロセスを管理し，ラウンドロビンスケジューリングによるタイムシェアリングシステムを模擬する．ノードはタイムスライス毎にキューの先頭からプロセスを取り，与えられた実行時間からクオンタムを差し引いてキューの末尾に返す．複数のプロセスがキューにあれば，ラウンドロビンによって実行時間の遅延が発生する．

指定されたマイグレーション間隔毎にバリア同期が取られ，マイグレーション処理が行われる．マイグレーション処理では，負荷の高いノードで動くプロセスを負荷の低いノードへ移動する．移動はキュー間でプロセスを付け替えることで行う．指定されたオーバヘッド時間の間プロセスに与えられるクオンタムはマイグレーション処理に費やされる．マイグレーション処理が終了すると，再びバリア同期を経て通常の処理に戻る．

4.3 スペアノード数と実行時間の関係

使用ノード数 8，プロセス数 8 の並列タスク内で，アクティブプロセス数を変え，スペアに廻すノードを

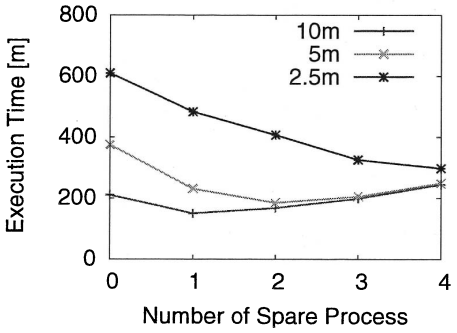

図 6　スペアプロセス数と実行時間

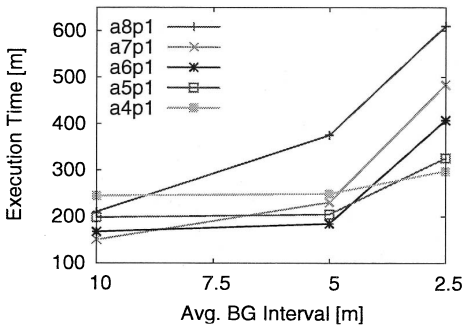

図 7　ノードあたり 1 プロセスの場合の実行時間

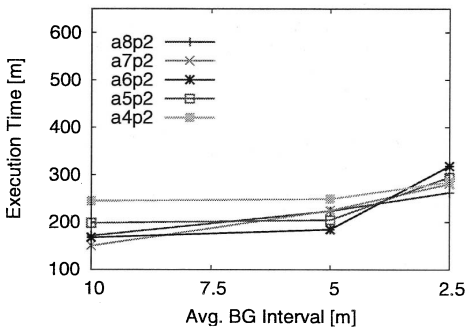

図 8　ノードあたり 2 プロセスの場合の実行時間

0 から 4 まで変化させ実行時間を求めた．マイグレーションを試行する間隔は 1 分毎で，並列タスク内のプロセスは 1 ノードあたり 1 つ配置する．

並列タスクの延べ実行時間は，120 分× 8 プロセス分で 960 分とする．またシミュレーションに使用する各定数は，実アプリケーションの実験から得られたものを使用する．

背景負荷の平均処理時間は 10 分で，背景負荷の平均到着間隔を 10 分，5 分，2.5 分と変化させた．ノード数は 8 なので，システム全体としてこれは利用率 $\rho = 1/8, 1/4, 1/2$ に相当する．

結果を図 6 に示す．横軸はスペアノード数，縦軸は実行時間である．背景負荷が大きくなるに従って実行時間を最短にするスペアプロセス数が多くなっている．一方，低負荷時にスペアプロセス数を多くしすぎると計算を行うプロセスが少なくなるため，性能低下が起きている．

4.4　1 ノードあたりのプロセス数と実行時間の関係

並列タスクのプロセスを 1 ノードあたり 2 つ以上実行することを許せば，計算プロセス数を多く保ったままプロセスマイグレーションを行える．これは各ノードに 1 つずつスペアプロセスを追加することで実現できる．

1 ノードあたり 1 プロセスのみ許可した場合の実行時間と，2 プロセスまでの重複を許した場合の実行時間をシミュレーションで比較した．並列タスクの延べ実行時間、マイグレーション間隔，背景負荷の条件は前節のシミュレーションと同様である．

結果を図 7，図 8 に示す．横軸は背景負荷の平均到着間隔，縦軸は並列タスクの実行時間である．パラメータとしてアクティブプロセス数を変えている．

結果から 1 ノードに 2 プロセスを収容することで，背景負荷が大きい場合の実行時間が大きく改善していることが分かる．

4.5　背景負荷の粒度とマイグレーション間隔による実行時間の変化

背景負荷の粒度を，ノード使用率 ($\rho = 1/8$) 一定のまま，到着間隔と平均処理時間を変えることで変化させる．各々に対してプロセスマイグレーションの間隔を 1, 2, 4, 8 分と変え，マイグレーション頻度に対する相対的な粒度毎に実行時間を求めた．

結果を図 9，図 10 に示す．縦軸は実行時間，横軸は背景負荷の到着間隔をマイグレーション間隔で割った相対的な粒度である．また，パラメータとしてアクティブプロセス数を変えている．

結果から，マイグレーションを行う間隔が背景負荷の平均到着間隔の 1/2 を越えると著しい性能低下が起きていることが分かる．

4.6　考　　察

背景負荷がある場合，スペアプロセスを置きプロセスマイグレーションを行うことで実行時間の改善が得られる．高負荷状況下ではスペアプロセスが多いほど実行時間の改善が得られるが，負荷に対してスペア数

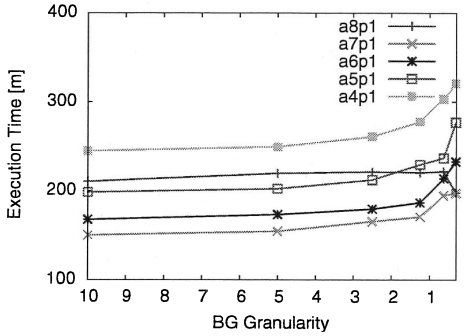

図 9　相対粒度と実行時間：1 ノードあたり 1 プロセス

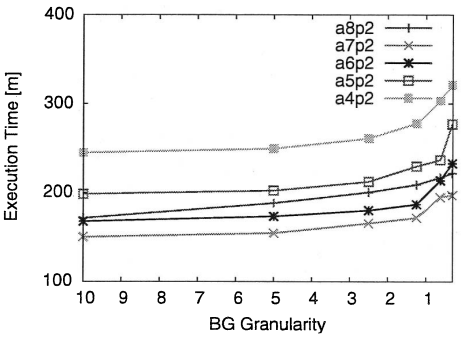

図 10　相対粒度と実行時間：1 ノードあたり 2 プロセス

を多く取りすぎると計算に使われるプロセスが減るため性能低下が起きる．負荷が比較的軽い場合はスペア 1 または 2 が適当である．

1 ノードに 2 プロセスを収容する効果は大きく，高負荷状況で明らかに 1 プロセスよりも有利であった．2 プロセス収容の場合，高負荷向けにアクティブプロセスを制限する必要がなく，低負荷から高負荷まで広く対応できることが分かった．

総合的に見てスペアプロセス数1，各ノード上のプロセス数上限 2 のような設計が有効で，低負荷から高負荷まで良好な性能改善が得られる．

マイグレーションを行う間隔は，背景負荷の平均到着間隔の半分程度にすると良好な性能が得られる．

5. 結　　論

本論文では，プロセス競合による処理時間遅延を解決するために，アプリケーション内でプロセスマイグレーション可能な MPI プログラムの実装方法を提案した．

提案手法によるプロセスマイグレーションは実現可能であり，良好に遅延を解消することを示した．

様々な背景負荷に対して，どのようなプロセス移動を行うと有効であるかをシミュレーションによって評価した．

参 考 文 献

1) Michael Litzkow, Miron Livny, and Matt Mutka, “Condor - A Hunter of Idle Workstations”, Proceeding of IEEE 8th International Conference on Distributed Computing Systems, 1988.

2) Derek L. Eager, Edward D. Lazowska, and John Zahorjan, “The Limited Performance Benefits of Migrating Active Processes for Load Sharing”, In Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS), May 1988.

3) Amnon Barak, Oren La'adan and Amnon Shiloh, “Scalable Cluster Computing with MOSIX for LINUX”, Proceeding of Linux Expo '99, pp. 95-100, Raleigh, N.C., May 1999.

4) Jeremy Casas, Dan Clark, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole, “MPVM: A Migration Transparent Version of PVM”, Computing Systems 8, 2 (Spring), 1995.

5) Jonathan Robinson, Samuel Russ, Bjorn Heckel, and Brian Flachs, “A Task Migration Implementation of the Message-Passing Interface”, Proceedings of the High Performance Distributed Computing (HPDC'96), 1996.

6) Georg Stellber, “CoCheck: Checkpointing and Process Migration for MPI”, Proceeding of IPPS '96, 1996.

7) Peter S. Pacheco, “Parallel Programming with MPI”, Morgan Kaufmann Publishers, 1997.

8) Otto Sievert and Henri Casanova, “MPI Process Swapping: Architecture and Experimental Verification', University of California Sand Diego Dept. of Computer Science and Engineering Technical Report CS2003-0735, 2003.

9) Otto Sievert and Henri Casanova, “A Simple MPI Process Swapping Architecture for Iterative Applications”, to appear in the International Journal of High Performance Computing Applications Fall issue 2004, 2004.

10) 石川 裕, 佐藤三久, 堀敦史, 住元真司, 原田 浩, 高橋俊行, “Linux で並列処理をしよう”, 共立出版, 2002.

11) 矢澤慶樹, 堀口 進, “並列計算機上で実行中に使用 PE を移動する MPI プログラムの性能評価”, 情報処理学会研究報告 2004-HPC-97 (9), 2004.