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

2004－ARC－157　(9)
2004－HPC－ 97　(9)
2004／3／1

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

矢 澤 慶 樹[†]　　堀口 進[†]

マルチユーザ・マルチプログラミングの並列計算機環境において，同一 PE 上で複数のジョブに属すプロセスがオーバラップして実行された場合，処理速度の著しい低下が発生する．この遅延は並列ジョブ全体に影響を与える．この状況を改善する方法として，実行中に各ノードの性能を監視し，動的に使用 PE を移動する MPI プログラムを作成した．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なったところ，遅延回避に有効であることが明らかになった．また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わかった．

A Performance Evaluation for a Dyamically Migrating MPI Program

Yoshiki Yazawa[†] and Susumu Horiguchi[†]

On a multi-user and multi-programming parallel computer, an over wrapped execution of processes which belong to several parallel jobs degrades performance severely. Even a delay of execution time on a PE will influence on the entire parallel jobs. To avoid this, we implemented a dynamically migrating MPI program, which considers background load of PEs and examined its performance. We obtained that the migrating MPI program in a combination of solver for Laplacian equation archives certain performance improvement against non-migrating program. The overhead of process migration is also examined, and it was found that its influence can be ignored.

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ジョブが投入され，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ジョブ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．競合によって並列ジョブを構成する部分プロセスが遅延すると並列ジョブ全体が影響を受ける．

マルチユーザ・マルチプログラミングの並列計算機では，各ノードの資源利用は，多くの場合ユーザの協調的な運用に任されており，ユーザは各ノード上で自分のジョブが他のユーザのジョブと重なり合うことがないようにノードの監視と投入先の決定を行っている．投入先を制御するキューシステムが備わっていることも多いが，投入時のノード状態を反映して投入先を決定するものがほとんどであり，計算中の負荷変動には対応できず，更なる利用増加に対応することは困難である．

このため，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ジョブ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が，従来はプロセス状態の転送にファイルを経由するなどオーバヘッドが大きく，有効性が疑問視されていた．

本稿では，マルチユーザ大規模並列計算機での処理時間遅延を解決する手法として，使用 PE を動的に変更する MPI プログラムを提案する．この方法では，プロセス移動時に内部状態の転送が完全にネットワーク上で行なわれるため，高い性能が期待できる．実装したアプリケーションであるラプラス方程式の求解プログラムとその性能評価について報告する．

2. 使用 PE を動的に変更するプログラム

2.1 プロセス競合とプロセスマイグレーション

同一ノード上でのプロセスの競合による処理時間の遅延を回避する方法として，従来，並列処理の高速化を目的に研究されてきた動的負荷分散手法を応用することが可能である．

並列アプリケーションでよく用いられる方法は，マ

† 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

スタ・スレーブ型の並列プログラム実装を行い，実行中に各ノードの進捗を見ながらマスタがスレーブに割り当てる仕事量を変化させる方法である．この方法はワークプールと呼ばれる．

もう一つのアプローチとして，ノード負荷の監視を行い，負荷の軽いノードが見出された場合，動作中のプロセスを中断し，負荷の軽いノードに移動してから実行を再開する方法がある．この方法はプロセスマイグレーションと呼ばれる．

OS の上で起動したプロセスは，実行とともに内部状態が変化していくため，プロセスの移動には内部状態の完全なコピーと状態の復元が必要になる． しかし計算機上の各種リソースに対する要求に関しては，固有のデバイスへのハンドルやディスクリプタといった形で資源にバウンドされていくため，移動が難しく，完全なプロセスマイグレーションには，OS のサポート[1] が不可欠である．

しかし商用の計算機の場合，OS に変更を加えることは困難であることから，多少の限定を加えて OS 外でミドルウェアやライブラリのサポートでプロセスマイグレーションを実現している[2]．この場合，プロセスの内部状態の保存はファイル経由で行なわれることが多く，オーバヘッドの原因となっている．

2.2 MPI プログラムのマイグレーション

MPI のようなメッセージパッシングを利用した並列アプリケーションでは通信が不可欠であり，プロセスの移動後に通信のリダイレクトが必要となる． また，MPI-1 では実行中のプロセス生成ができないという問題があり，これも解決する必要がある．

MPI プログラムを対象としたプロセスマイグレーションには MPI-TM[3] CoCheck[4] 等の研究があるが，これは MPI 実装自体にプロセスマイグレーション機能を追加するものであった．

現在利用可能な MPI を用いて，オーバヘッドの少ないプロセスマイグレーションを実現するために，筆者らは次のようなアプローチを取った．

- 通信先をランクで直接指定するのではなく，プロセスの実体を示す rank と，計算領域を示す area を対応づけるテーブルを用意し，正しい通信相手に通信がリダイレクトする
- MPI-1 規格に準拠した MPI 実装では，プロセス生成はできないため，並列ジョブの起動時に冗長プロセスを作っておく．この場合，冗長プロセスは MPI_Barrier() 関数を使って必要になるまで休眠させておく．
- バリア同期を使って全てのプロセスを同期させ，通信コンテキストを アプリケーション処理のための通信とプロセスマイグレーションに必要な通信とで切替える

以上のようにすれば MPI アプリケーション内でプロセスの移動を実現することが可能となる．

3. 実　　装

この章では実装した MPI プログラムの詳細について説明する．

3.1 計算ループとアプリケーション

作成したプログラムでは計算ループで扱うアプリケーションとしてラプラス方程式の差分法による求解を実装している．この部分のコードは SCore に関する参考文献[5] に掲載されているものを利用している．

このアプリケーションは，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なう．

PE には全格子点領域を縦方向に分割した領域が割り当てられるが，領域の境界では，他 PE に割り当てられた領域との値の交換が必要で，この部分で通信が必要になる．

3.2 通信コンテキストの切替

MPI による並列プログラムで，アプリケーションの処理とは別の動作をさせる場合，必要な通信はアプリケーション用のコンテキストとは分離しなければならない．これを実現するために，実装したプログラムでは，一定の条件で各プロセスがバリア同期を呼び，以後の通信は別コンテキストとして扱うモデルを用いている．

作成したプログラムでは計算ループのループカウンタを用いて，これが一定の値になったら chkpt() という関数を呼び出す実装になっている．

chkpt() はマイグレーション動作を全て提供する関数で，この関数がリターンすると，呼び出し元は単純に実行を再開するだけで以後の処理を実行できる．

冗長プロセスとして起動されたプロセスは，計算ループの入口でこの chkpt() を呼ぶことでアプリケーション処理をすることなく待機状態に入る．

3.3 chkpt() におけるマイグレーションの処理

chkpt() 内で行なわれるマイグレーション処理について，処理中の通信を図 1 に示す．図では rank1 のプロセスが高負荷のノードで動いており，rank3 の休止状態の冗長プロセスへ移動を行なう．

全てのプロセスが chkpt() を呼ぶとこのルーチンの入口にある MPI_Barrier() が成立する．ここからマイ

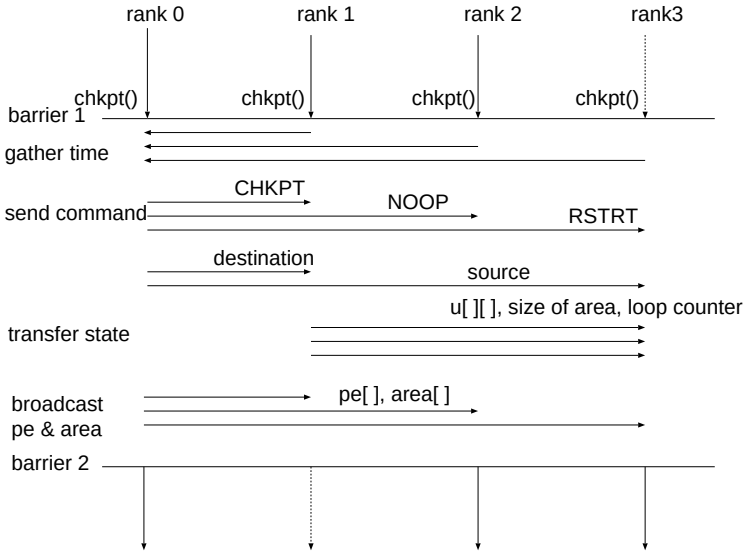

図 1 マイグレーション中の通信

グレーション処理の完了するまでの全ての通信はマイグレーションのための手順に則っていると仮定される．

1 回目のバリア成功後，rank0 のプロセスはコントローラとしての動作を開始する．同時に rank0 以外のプロセスはコントローラに向かってループ測定区間の実行時間を送信する．コントローラはこの時間から最大のものと最小のものを選び出し，これらを各々マイグレーション元である source，マイグレーション先である destination と選定する．

rank0 以外のワーカプロセスは実行時間の送信を終えると，コントローラからのコマンド受信状態で待機する．コントローラは source に CHKPT，destination に RSTRT，その他のプロセスに NOOP というコマンドを送信する．

CHKPT を受信したプロセスである source は，続いて destination の情報をコントローラから受け取る．RSTRT を受信したプロセスである destination は，同様に source の情報を受け取る．

次にこの情報を用いて，source と destination の間でマイグレーションに必要なデータを転送する通信が行なわれる．

一方，マイグレーションの手続きによってプログラム開始時とは PE の担当領域が変わっているため，コントローラは全プロセスに向かって各々の PE と新しい担当領域の組合せを格納したテーブルをブロードキャストする．

以上の全ての手続きが終った後，全てのプロセスは再び MPI_Barrier() を呼ぶ．ここで通信コンテキストがチェックポイント動作から通常のアプリケーション動作に戻り，以後は並列計算に必要なアプリケーション通信を行なう．

3.4 プロセス移動のトリガ

現在の実装では，プロセス移動のために chkpt() が呼び出されるのはループカウンタの値が 200 になった時で，背景負荷の測定はそれまでの計算ループの通信を除く累積実行時間である．ラプラス方程式のソルバでは通信が同期的に行なわれるため，通信を含む実行時間では負荷の測定ができない．

トリガ条件は計算ループ内に記述されており，条件式を書き換えるだけで変更可能である．

4. 性 能 評 価

実装したラプラス方程式ソルバプログラムを IBM SP2（図 2）システム上で実行し，性能を評価した．この計算機は，64 台の PowerPC 604e ノードと 4 台の Power3 ノードからなる並列計算機で，今回は PowerPC ノードを利用した．PowerPC ノードは 1 ノードあたり PowerPC 604e 332MHz を 4 個搭載した SMP 構成となっているが，並列ジョブのプロセスは，1 ノードあたり 1 つ投入されるように設定した．

プロセス数 4 および 8 の場合について，背景負荷あり／なし，マイグレーションあり／なしについて実行時間を調べた．背景負荷の与え方は，使用ノードの 1 つに CPU インテンシブな数値積分プログラムを走らせることで行なった． また，メッセージ交換に利用されるネットワークは SP Switch と 100Base-TX

Ethernet の双方についてそれぞれ測定した．

マイグレーションありの実行の場合は計算実行ノードの他に冗長ノードを 1 つ確保し，計算途中でタスクを移動する．

図 2 IBM SP2

4.1 背景負荷がない場合の並列ラプラス方程式ソルバの性能

図 3 は背景負荷がない場合のラプラスソルバの演算部分の実行時間である．図中のラベルで MIG はマイグレーションありを，SP は SP Switch を，Ether は Ethernet を表す．

この解法では各 PE に割り当てられた領域の境界で隣接する PE との値の交換が必要になるため，PE 数が増えると通信も多くなるが，この PE 数ではほぼ線形に近い速度向上が得られている．

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よって費やされる時間は平均 0.27 秒であった．

4.2 背景負荷がある場合の並列ラプラス方程式ソルバの性能

図 4 は背景負荷がある場合の並列ラプラスソルバの演算部分の実行時間である．

ここでは 2 番目の PE に背景負荷としてプロセスを 4 つ投入している．4 つである理由は前述のようにノードが 4CPU の SMP であるためである．

図からマイグレーションによって実行時間が大幅に改善することが分かる．マイグレーションしない MPI プログラムの場合，1PE の実行の遅延が並列ジョブ全体に影響しているが，マイグレーション可能なプログラムの場合，この遅延を免れ，背景負荷のない場合に近い実行時間を実現している．

マイグレーションするプログラムの実行時間が背景負荷なしの場合の実行時間より大きい理由は，移動元の決定に演算ループの 200 回の実行時間を使用しているためで，この部分が低速なノードの影響を受けている．

移動元の決定にループ何回分の実行時間を利用するかは議論のあるところである．

ループ 1 回の実行時間が長いプログラムや，比較的低速なプロセッサで 1 回あたりの実行時間が長い場合はループ回数を小さくして性能向上を図ることができる．一方，ループ 1 回の処理が軽かったり，プロセッサが高速である場合はループ回数を大きくしなければ正確な検出ができない．

今回の実装は全体で 1 回だけマイグレーションを行なうものであるが，複数回移動するように拡張することは容易である．この場合，検出した内容とその後のマイグレーションによる実行結果を追跡比較してサンプリングするループ回数を最適化することが考えられる．

4.3 バリア同期を用いたマイグレーションに関する議論

今回評価したプログラムでは，計算ループの実行回数をトリガにしてマイグレーションを行なっている．そしてマイグレーション動作の前後にはバリア同期を使って全プロセスを同期させている．これは全プロセスを通常の計算のためのデータ送受信からマイグレーション処理のための通信に切替えるためである．

この方法では，この間，全ての並列プロセスは計算処理を止めてマイグレーションの完了を待つ．これはマイグレーションするプロセス以外についても行なわれるので，この間の計算処理能力が損なわれる．

割り込み駆動型のマイグレーション機構を実装すれば，全プロセスにわたる同期を排除することができる．MPI の通信モデルでは，スレッドを利用したプログラミングにより，受信専用スレッドと共有変数を利用して疑似的に受信割り込み機構を実現することが可能と考えられるが，現在の MPI 実装の多くはこれに必要なスレッドサポートレベルである MPI_THREAD_MULTIPLE を提供していない[6] ため，実現が困難である．

今回の実装では，以上のような理由からバリア同期によるマイグレーションを選択しているが，実験結果から，マイグレーションに費やされる時間はごく僅か（8PE による実行で 95.8 秒中の 0.42 秒程度）であり，この間全プロセスが実行を停止しても影響は無視し得る．したがってバリア同期によるマイグレーション実

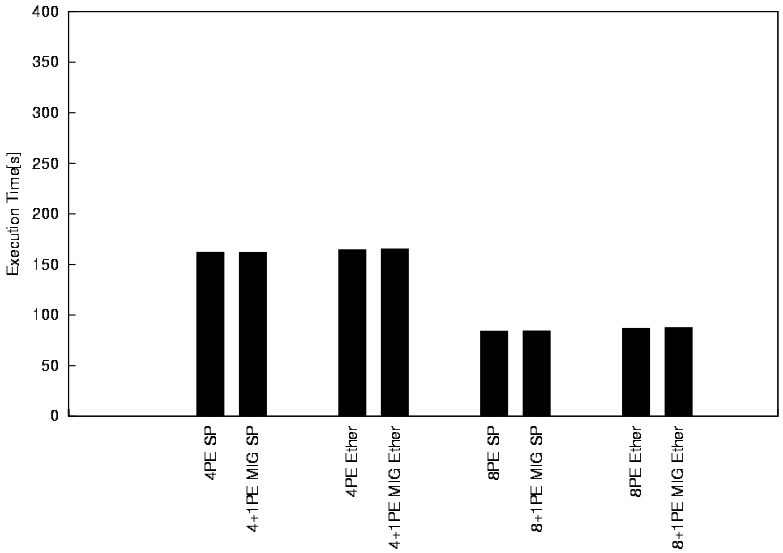

図 3 背景負荷がない場合の実行時間

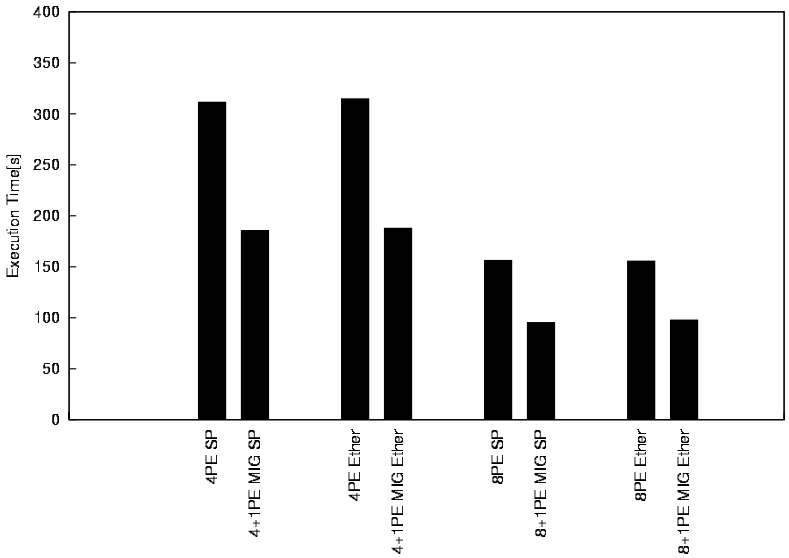

図 4 背景負荷がある場合の実行時間

装でも性能上全く問題がないと言える．

4.4 ネットワーク性能の影響

SP2 では通信用ネットワークデバイスとして帯域 100Mbit/s の 100Base-TX Ethernet と帯域 1200Mbit/s の SP Switch が利用可能である．今回の性能評価ではこれらの各々について性能を比較したが，並列計算時間（通信を含む）は最大で 3 秒程度しか違いがなく，ネットワーク性能はあまり性能に影響を及ぼさなかった．

次にマイグレーションに関して考察する．マイグレーション時にはプロセスが計算処理に使っている内部データを全てマイグレーション先のノードに転送しなければならない．評価に用いたプログラムでは，格子点として 8 バイトの double が 1024×1024 個アロケートされており，データ空間は 8MB になる．今回の実装では，移動元と移動先のノード間でこの格子点の値全てと領域サイズ，ループカウンタを転送しているが，転送時間はネットワークデバイスとは無関係に 0.27〜0.42 秒程度であり，マイグレーションの頻度を考えるとこれ以上の最適化を行なう必要性はないと思

われる．また，格子点の情報を必要な範囲に限定したとしても，現状ではネットワークの帯域に無関係に時間が決まっているために性能向上は得られないと思われる．

5. 結　　論

本稿では，ノードでのプロセス競合による処理時間遅延問題を解決するために，負荷の重いノードで処理中のプロセスを，負荷の軽いノードへ移動する MPI プログラムを実装し，性能評価を行なった．

性能評価によって，

- 背景負荷による遅延に関してマイグレーションによって得られる性能向上は大きい
- 通常の MPI によって実装可能なバリア同期によるマイグレーション機構でも実用上問題ない性能が得られる

ことが明らかになった．

従来，プロセスマイグレーションは得られる性能向上に比べてオーバヘッドが大きいという主張がされてきた[7] が，マイグレーションにファイルを経由しない本手法ではオーバヘッドは小さく，またマルチプログラミング環境で，プロセス競合による並列ジョブの遅延回避という観点からは極めて有効であることが分かった．

もちろんリソースマネージャや SMP などの方法でプロセス競合が起こらないようにするアプローチが有効であることには変わりがないが，それでも競合は起こり得る．この場合，アプリケーション内にこれを回避するコードがあれば，かなり事態は改善する．

今回の実験を行なった SP2 は大規模なシステムであるが，小規模なクラスタも数多く用いられており，このような計算機では本手法のようなアプローチは有用である．

今回のプログラムで，マイグレートなしのラプラス方程式ソルバが 100 行以下で記述できるのに対し，マイグレーション関連のコードは 500 行程度になる．しかしマイグレーション部分は chkpt() という関数の呼び出しにカプセル化されており，これをライブラリ化することが可能である．現在のコードは関数インタフェースがラプラス方程式ソルバの内部データに依存しているが，この部分を構造体等で抽象化して汎用性を持たせる作業に着手している．

筆者らは，少ないプログラミング負担でマルチユーザ並列計算機でのプロセス競合の問題を解決するという観点から，ライブラリやシステムサポートによるプロセスマイグレーションの研究を進めている[8]．

参 考 文 献

1) 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.
2) M. Litzkow, M.Livny, M.Mutka, "Condor - A Hunter of Idle Workstations", Proceeding of IEEE 8th International Conference on Distributed Computing Systems, 1988.
3) Jonathan Robinson et.al, "A Task Migration Implementation of the Message-Passing Interface", Proceedings of the High Performance Distributed Computing (HPDC'96), 1996.
4) G. Stellber, "CoCheck: Checkpointing and Process Migration for MPI", Proceeding of IPPS '96, 1996.
5) 石川 裕 et.al, "Linux で並列処理をしよう", 共立出版, 2002.
6) W. Gropp, E. Lusk, R. Thakur, "Using MPI-2 Advanced Features of the Message-Passing Interface", MIT Press, 1999.
7) Eager, D., Lazowska, E., and Zahorjan, J., "The Limited Performance Benefits of Migrating Active Processes for Load Sharing", In Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS), May 1988.
8) 矢澤慶樹, 堀口 進, "ユーザレベル通信ライブラリによるプロセスマイグレーション", Proceedings of FIT2002, pp183-184, Sep. 2002.
9) Peter S. Pacheco, "Parallel Programming with MPI", Morgan Kaufmann Publishers, 1997.
10) Casas, J., Clark, D. L., Konuru, R., Otto, S. W., Prouty, R. M., and Walpole, J, "MPVM: A Migration Transparent Version of PVM", Computing Systems 8, 2 (Spring), 1995.