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

2009－ARC－182 (25)
2009－HPC－119 (25)
2009／2／28

プロセス間共通メモリイメージを考慮したマイグレーション最適化

山崎翔平[†,†††] 遠藤敏夫[†,†††] 松岡聡[†,††,†††]

マシンの性能向上に伴いジョブが大規模化する一方、様々な動機からジョブを動的にマイグレーションさせる要求がある。動機の中にはアプリケーション性能向上も含まれており、このケースではマイグレーション先およびタイミングは動的に決定する必要がある。また、そのためにもマイグレーションコストの大幅な削減が必要である。本論文では、プロセス間のメモリイメージの類似性を活用したマイグレーション時間の最適化手法を提案する。本手法は、マイグレーションに必要なデータ転送サイズを削減するだけでなく、実装方法において高い並列性を持つために、ノード数に対して高いスケーラビリティを持つ。本論文の実験では、アプリケーションの問題サイズ、構成ノード数、といったパラメータを変化させたときに、アプリケーションのマイグレーションにかかる時間や、そのマイグレーションプロセスの内訳にどのような変化が表れるかを調べた。

Migration Optimization Accounting for Similarity of Process Images

Shohei Yamasaki,[†,†††] Toshio Endo[†,†††]
and Satoshi Matsuoka[†,††,†††]

Demands for migration of large scale jobs are getting stronger on large scale systems for several reasons. For example, jobs may be migrated to different machines to avoid machine maintainance or performance degradation. In many cases, destination and timing should be determined dynamically. For reduction of migration costs of large scale jobs, this work presents an optimization method that utilizes similarity among memory images of parallel processes. In addition to reducing amount of communication, this method has high scalability, since it creates differences of images in parallel. With this method, we evaluated migration costs on a real cluster in detail, with several problem sizes and the number of nodes.

1. はじめに

複数のプロセスの集合体である大規模ジョブを動的に配備する要求がある。その動機として、1) マシンの運用/保守、2) アプリケーションの利用継続性、3) 大規模環境上でのフォールト発生に対する対故障性、4) アプリケーションの性能向上、などが挙げられる。アプリケーションの利用継続性とは、計算プラットフォームの利用時間を、プログラムの実行時間が上回った場合でも、それまでの計算内容を保持し、他のプラットフォーム上でプログラムの実行を再開することを指す。プログラムの実行時間は、実行するまで分からない事がよくあるため、このような機能があれば非常に便利である。このような機能なしでは、それまで膨大な時間をかけて行っていた計算が水の泡になってしまう。

マイグレーションは使い方によっては、アプリケーションの性能を向上させる可能性を持つ。関連する研究として、プロセスを実行するマシンを動的に再配置することでMPIプログラムの性能を向上させる研究[1]や、2) 仮想マシンのマイグレーション機能を活用することでリモートデータへのアクセスを最適化する研究がある[2]。前者では、アプリケーション特性や、マシンに関する物理的な情報 (マシンのスペック、マシン間のネットワーク性能) を考慮し、利用可能な資源のうち最適な資源上でアプリケーションを実行する。また、資源の利用状況、サブミットされるジョブに応じて、資源の再分配も考慮する。後者では、マイグレーションするべきか否かを決定する性能評価モデルを構築しており、これは最短経路問題に帰着したアルゴリズムを採用する。

このようなマイグレーションの要求がある一方、ジョブのサイズやマシン性能の増加傾向に伴い、マイグレーションコストも増加する。例えば、東京工業大学のTSUBAME[3]から他拠点へ1000ノードを同時にマイグレーションをする際、1ノード当たり使用メモリが10GB、クラスタ間のバンド幅を10Gbpsとした場合、単純な計算から8000秒程度マイグレーションに時間がかかることが分かる。

† 東京工業大学
†† 国立情報学研究所
††† JST, CREST

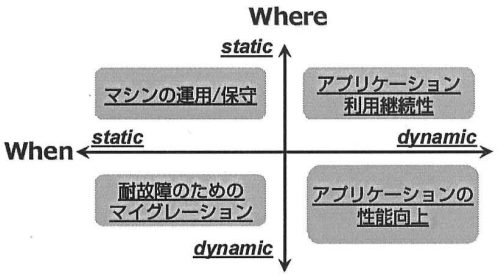

図 1　マイグレーションの分類

2. 提案手法

2.1 提案手法の概要

問題点を解決する要件として、1) トータルのマイグレーション時間の削減に効果的であること、と 2)When/Where において共に dynamic な場合でも対応できること、を挙げる。大規模なジョブにおいて、マイグレーションが要求される動機として 1) マシンの運用/保守、2) アプリケーションの利用持続性、3) 対故障性、4) アプリケーション性能向上、の 4 点を挙げたが、これらはどのタイミングでマイグレーションをするか (When) という観点と、マイグレーションする先はどこか (Where) という観点で、分類することができる。また、これらの軸は、さらにあらかじめ決定されている場合 (static)、アプリケーション実行時に動的に決定される場合 (dynamic) という観点でも分類することができる。これらを考慮した分類を、図 1 に示す。アプリケーション性能向上のためのマイグレーションを実現するには、When/Where ともに dynamic である必要があるため、我々はこれを要件として考慮する。

各サイトの代表ノード間で、メタ情報を交換することで、When/Where において動的にマイグレーション先を決定することができる。メタ情報には、マシンの利用状況や、各マシンのスペックを含む。今後、代表ノード間の通信によって、マイグレーション要求が発生した時に、動的にそのマイグレート先が決定されているものと仮定する。本論文は、マイグレーション時間を最適化することに焦点を当てており、マイグレーション先を決定するアルゴリズムに関しては言及しない。

本論文では、大規模ジョブを構成するプロセス間のメモリイメージの類似性を活用することでマイグレーション時間の最適化を提案する。本手法の特徴は、以下の 2 点である。

(1)　類似性を活用することにより、データ転送サイズを削減すること

(2)　並列性の高い実装方法により、全体のマイグレーション時間の最適化を行うこと

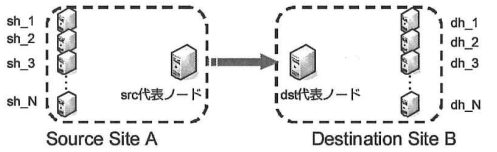

図 2　問題の設定

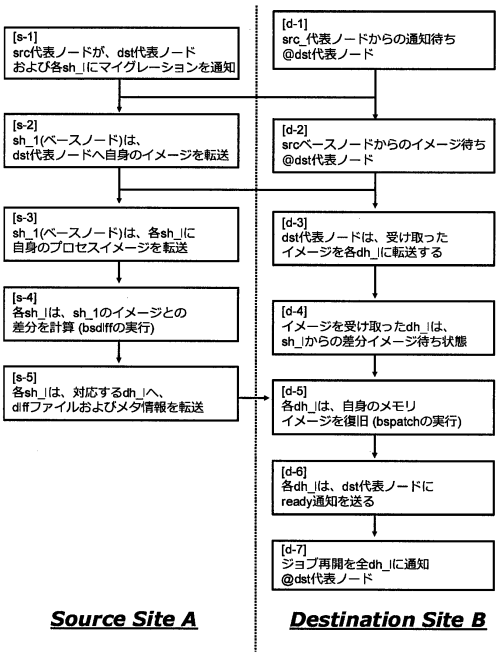

図 3　提案手法のフロー概念図

2.2 提案手法のフロー

前述の通り、マイグレーション先は決定されていると仮定し、あるサイト A からサイト B への大規模ジョブのマイグレーションを想定する。また、簡単のため、ジョブは同一サイト内のノードにより構成されるとする。sh_i は、ジョブを構成するサイト A 内 i 番目のノードを意味し、dh_i は、sh_i のマイグレーション先のノードを意味する。$mem(node)$ を node のプロセスメモリイメージとした場合、問題は「全ての $mem(sh\_i)$ を、dh_i へ移動するまでの時間をいかに最小化するか」と定義される。

図 3 は、本手法の全体のフローを示す。まず、マイグレーション先の代表ノード (dst 代表ノード) は、マイグレーション元の代表ノード (src 代表ノード) からのマイグレーション要求通知の待ち状態にある。マイグレーション元のサイトでは、src 代表ノードがマイグレーション先の dst 代表ノード、およびサイト内の各 sh_i にたいしてマイグレーション要求を通知する。この際、src 代表ノードは、1)sh_i と dh_i の対応表、

2) ベースとなるノードの指定 (*sh_i* のいずれかとなる)、3) マイグレーション元や先の代表ノードの通信に必要となる情報 (IP アドレス、port 番号など)、の3点の情報も通知メッセージに付加する。ベースとなるノードは、マイグレーション先の dst 代表ノードへ自身のプロセスイメージを転送する。それを受け取ったマイグレーション先の dst 代表ノードは、各 *dh_i* へ受け取ったイメージを転送し、各 *dh_i* は、イメージを受け取ると、対応する *sh_i* からの通信待ち状態となる。ベースノードは、サイト内の *sh_i* に自身のプロセスイメージを転送する。ベースノードのプロセスイメージを受け取った *sh_i* は、自身のプロセスイメージとの差分を計算し差分イメージおよび合成時に必要なメタ情報生成し、それらを *dh_i* へと転送する。マイグレーション先の *dh_i* は、差分イメージとメタ情報を受け取ると、元のイメージを復旧する。復旧が完了すると、マイグレーション先の dst 代表ノードへと ready メッセージを送る。dst 代表ノードは、全ての *dh_i* から ready メッセージを受け取ると、ジョブの再開を全 *dh_i* に通知する。

以上のプロセスで用いた特筆すべきユーティリティについて説明をする。マイグレーション元であるサイト A において、ベースノードから各 *sh_i* に分配する際には、dolly+[4] を用いる。これは、クラスタネットワーク用のアプリケーションレベルマルチキャストのツールである。リングトポロジを構成し、受信、書き込み、送信をパイプライニングすることで、ノード数に対して O(1) の転送時間を実現する。マイグレーション元のサイト A において、各 *sh_i* がベースノードとの差分を計算する際には、bsdiff[5] を用いる。また、マイグレーション先のサイト B において、各 *dh_i* が自身のメモリイメージを復旧する際には、bspatch[5] を用いる。これらは、バイナリファイル用のパッチを作成したり、適用するためのツールである。

3. プロセスイメージの類似性

本手法は、プロセス間のメモリイメージの高い類似性を要求するが、実際のアプリケーションにどれほどの共通部分があるのかを計測する必要がある。アプリケーションとして、1)HPL と 2)POVRAY-MPI[6] を対象に予備実験を行った。HPL は、連立方程式の解を求めるベンチマーク MPI プログラムである。POVRAY は、3 次元グラフィックスで、物体の座標データや、光源や視点の位置などの環境に関するデータを計算して画像を描画する Ray Tracing プログラムである。予備実験では、この POVRAY(version 3.1g) に MPI 実行用のパッチを当てた POVRAY-MPI を使用する。HPL では、プロセス間で割り振られる行列データが全く異なるために、共通部分が少ないと予測し、対象アプリケーションとして選択した。また、POVRAY においては、オブジェクト等のプロセス間での共通データが多く、共通部分が多いと予測し、対象アプリケーションとして選択した。以下では、まずメモリイメージの取得方法について説明し、次に各アプリケーションの予備実験の概要および結果について述べる。

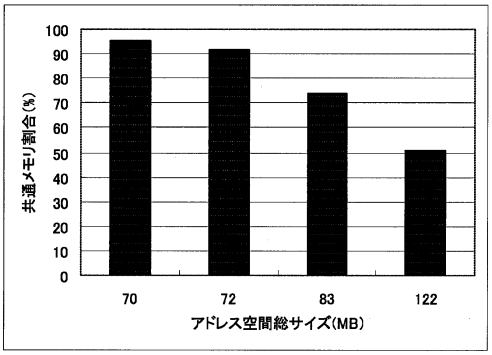

図 4 HPL におけるプロセス間の共通メモリ割合

3.1 メモリイメージの取得方法

作成したダンプ関数を対象アプリケーションのソースにダンプ関数を挿入し、メモリイメージをファイルに出力する。ダンプ関数では、"/proc/⟨process ID⟩/maps"から使用メモリの番地を取得し、メモリの中身をファイルに出力する。本論文において、ダンプ関数を挿入する場所は固定する。具体的には、HPL の場合では、LU 分解計算の直後にしており、POVRAY-MPI の場合では、レンダリング計算の直後とする。

出力されたプロセス間のダンプファイルの類似性を解析する。類似性の計算では、比較の対象となるデータサイズの粒度によって結果が異なるが、予備実験では、粒度として 4 バイト単位の比較を行う。

3.2 予備実験

予備実験は、4 ノード上で、上述した 2 種類のアプリケーションを動作させた。パラメータは、1 ノードあたりのプロセスメモリイメージのサイズとする。具体的には、HPL の場合、設定ファイル (HPL.dat) の N の値を 1280, 2560, 3840, 5120 の 4 種類に変化させた。POVRAY では、sample に付属する chess2.pov ファイルを用いて、出力ファイルサイズの値 (ピクセル) を、12800 * 9600, 14400 * 10800, 17600 * 13200, 19200 * 14400 の 4 種類に変化させた。

評価環境では、東京工業大学、松岡研究室の PrestoIII クラスタを利用した。CPU は Dual COre Opteron270 2.0GHz を使用し、メモリサイズは 2GB、OS は Linux 2.6.18-6-686 を用いた。使用した MPI は、mpich のバージョン 1.2.7p1 である。

3.3 予備実験結果

HPL では、ヒープ領域 (行列データ) の類似性が低いため、問題サイズが大きくなるほど全体の類似性も

メモリ領域の用途	使用サイズ[KB]	共通サイズ[KB](使用サイズに対する割合)	使用サイズの総イメージサイズに対する割合
xhpl	594	594 (100%)	0.5 %
heap(1)	33,555	33,180 (99%)	27 %
heap(2)	52,482	24 (0.05%)	43 %
/SYSV00000000	33,554	26,231 (78%)	27 %

図 5 HPL におけるプロセスイメージの主な内訳

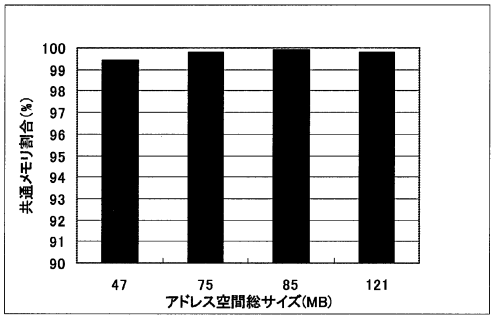

図 6 POVRAY-MPI におけるプロセス間の共通メモリ割合

メモリ領域の用途	使用サイズ[KB]	共通サイズ[KB](使用サイズに対する割合)	使用サイズの総イメージサイズに対する割合
mpi-x-povray	1,012	1,012 (100%)	0.8 %
heap(1)	83,710	83,520 (99%)	69 %
heap(2)	8.20	7.96 (97%)	0 %
/SYSV00000000	33,555	33,554 (99%)	28 %

図 7 POVRAY-MPI におけるプロセスイメージの主な内訳

低くなることが分かった (図 4)。また、問題サイズが最大の時 (総メモリイメージサイズ=122MB, N=5120) のメモリイメージの主な内訳は、図 4 の通りである。heap(2) の部分が、計算時に配布される行列データが入った領域だと考えられるが、そのプロセス間類似性がほとんどない。ちなみに、heap(2) 以外の領域では、たとえ問題サイズを大きくしても、変化することがなかった。問題サイズを大きくすることは、この行列データを格納する領域が大きくなることを意味するため、結果として全体の類似性も小さくなる。

POVRAY-MPI では、ヒープ領域の類似性が高いため、問題サイズが大きくしても、全体の高い類似性が保たれることが分かる (図 6)。また、問題サイズが最大の時 (出力ファイルサイズが 19200*14400) のメモリイメージの主な内訳は、図 6 の通りである。今回は heap(1) が問題サイズを大きくしたときに連動して拡大する領域であった。問題サイズが変化しても、heap(1) 以外の領域では変化は見られなかった。heap(1) の領域では、99 %という高い類似性があるため、問題サイズを大きくしても、全体としての類似性が高いまま保たれる。

これらの結果より、POVRAY-MPI のようなプロセス間で共通のオブジェクトを持つアプリケーションでは、問題サイズが大きくなったとしても高い類似性を保つことができ、本手法の対象となることがわかった。

4. 評価実験

4.1 評価実験の設定

本論文の実験では、1) アプリケーションの種類、2) アプリケーションの問題サイズ、3) ジョブの構成ノード数、といった 3 つのパラメータを変化させたときに、提案手法とごく単純な手法において、アプリケーションのマイグレーションにかかる時間や、そのマイグレーションプロセスの内訳にどのような変化が表れるかを調べる。アプリケーションの種類は、予備実験と同様に HPL と POVRAY-MPI の両アプリケーションにて実験する予定であったが、HPL に関しては実験データが揃わなかったため、本論文では POVRAY-MPI についてのみ言及させていただく。また、アプリケーションの問題サイズは、予備実験と同様に chess2.pov を用い、出力ファイルサイズの値 (ピクセル) を、12800 * 9600, 14400 * 10800, 17600 * 13200, 19200 * 14400 の 4 種類に変化させることで再現した。ジョブの構成ノード数は、4, 8, 16, 32 の 4 通りを考慮した。また、単純な手法とは、各 sh_i が、自身で対応する dh_i へイメージを転送する手法を指す。

2.2 章にて説明した提案のマイグレーション最適化手法のフローは、1) ベースノードが各 sh_i に自身のイメージを転送するフェーズ、2) 各 sh_i において差分イメージを計算するフェーズ、3) 各 sh_i が、対応する dh_i へ差分イメージを転送するフェーズ、といった 3 つのフェーズでは全ての sh_i の最大値を取るようにし、ファイル転送を伴わない通信のためのメッセージ交換の時間を無視した場合、以下のシーケンシャルなプロセスとして理解することができる。これは、マイグレーションの実行時間の内訳を知る際に、必要なプロセス分解となる。

(1) **base_transfer**: ベースノードは、自身のメモリイメージを dst 代表ノードへ転送する
(2) **dolly_transfer**: ベースノードは、サイト内の各 sh_i に自身のイメージを分配する
(3) **bsdiff**: 各 sh_i は、ベースノードのイメージとの差分を計算する
(4) **diff_transfer**: 各 sh_i は、対応する dh_i へ差

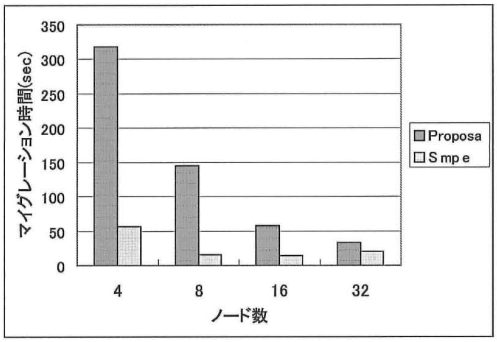

図 8　問題サイズ 12800 * 9600 に固定した時の POVRAY マイグレーション時間

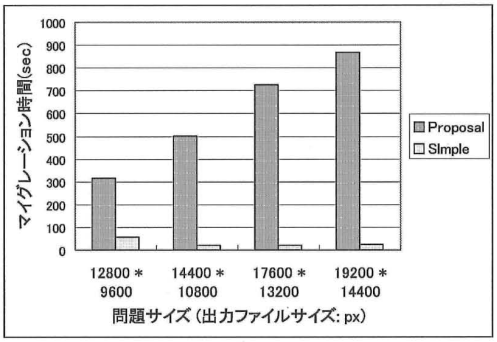

図 10　ノード数 4 に固定した時の POVRAY マイグレーション時間

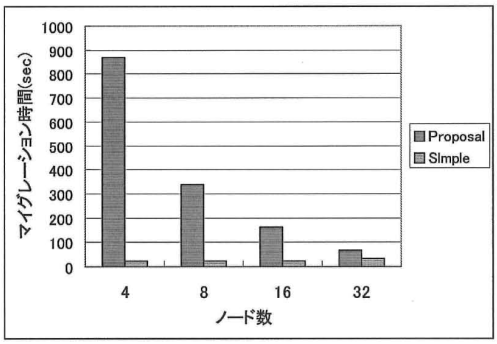

図 9　問題サイズ 19200 * 14400 に固定した時の POVRAY マイグレーション時間

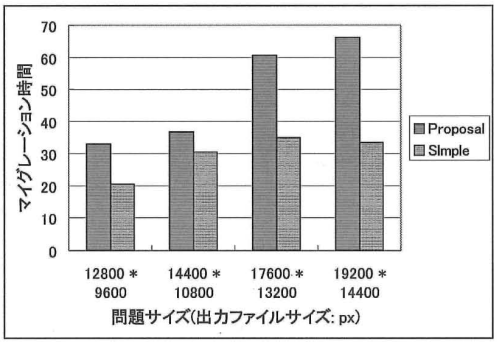

図 11　ノード数 32 に固定した時の POVRAY マイグレーション時間

分イメージを転送する

(5)　**bspatch**: 各 *sh_i* は、転送された差分イメージとベースノードのイメージから、元のイメージを復旧する

サイト A としては、東京工業大学、松岡研究室の PrestoIII クラスタを用いた。また、サイト B としては、InTrigger[7] の本郷キャンパス拠点を利用した。サイト A として、使用したマシンのスペックおよびネットワーク性能等は、予備実験で用いたものと同様である。サイト B として、使用したマシンには 2 種類ある。1 つ目は、CPU は Pentium M 1.8GHz、メモリは 1GB を搭載する。2 つ目は、CPU は Core2Duo 2.33GHz、メモリは 4GB を搭載する。両マシンともに、OS は Linux 2.6.18 で、ギガビット・イーサーネットに接続されている。

4.2　実験結果

問題サイズを変化させたときの挙動の理解として、図 8 と図 9 を比較から、問題サイズがより小さい方が提案手法の性能が単純手法に対してより近づくことが分かる。

ノード数を変化させたときの挙動の理解として、図

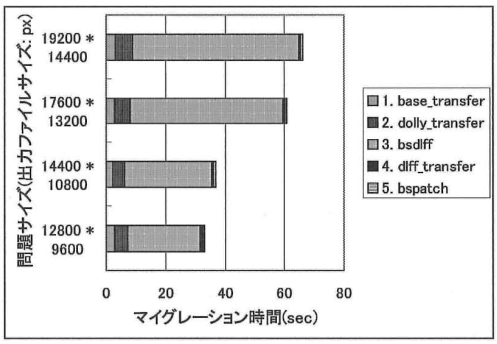

図 12　ノード数 32 に固定した時のマイグレーション内訳

10 と図 11 の比較から、構成プロセス数が多い方が本手法は有効であることが考えられる。

また、問題サイズが最小のパターンと、ノード数が最大のパターンにおいて、マイグレーション工程をより分解して性能を評価した (図 12)．これまでに得られた知見に加えて、差分イメージ計算が全体のボトルネックとなることが観測されるため、このフェーズの

最適化が課題であることが分かる．

5. 関連研究

時間的局所性を活用することで、保存すべきアドレス空間のサイズを削減する手法として、Incremental Checkpointing が挙げられる．チェックポイントの際に保存するデータは、1) プロセスのアドレス空間、および 2) カーネルやシステムが管理するプロセスステート情報である．また、この手法では複数回ステートを保存する必要があるが、dirty bit が立っているデータだけを新たに保存するため、その場での保存すべきデータ量を削減することができる．本手法は、時間的局所性ではなく、プロセス間の類似性に注目した手法である．また、Incremental Checkpointing の概念をマイグレーションに応用することは可能であるが、あらかじめマイグレーション先を固定しなければならないため、で述べた、When/Where において dynamic という要件を満たすことができない．

6. 今後の課題

今後の課題として大きく 3 点ある．

1 つ目は、評価実験のパラメータを増やし、より詳細に解析することである．加えるパラメータとしては、1) アプリケーションのタイプ、2) ダンプのタイミング、3) ノード数を考えている．アプリケーションのタイプとして HPL, POVRAY-MPI だけではなく、NAS Parallel benchmark や、プロセス間の類似性が 100 %の恣意的なプログラムでの結果を調べることが考えている．ダンプのタイミングによって当然プロセス間の共通部分は異なるため、ダンプのタイミングによって、本手法の性能が変わる．今後は、各アプリケーションのフローを整理し、いくつかのタイミングを比較するつもりである．また、今回の実験のパラメータでは本手法が単純な手法に比べて有効であることが示せなかったが、ノード数を増やすことで、本手法が単純な手法の性能を上回る可能性が十分にある．まずは、よりノード数を増やした実験を行うことで、その挙動を調べる必要がある．

2 つ目は、モデル化による本手法の有効性の予測である．上述した通り、ノード数が増えた時に、統計的手法を用いて性能モデルを構築することにより、パラメータをどう変化させた時に、本手法が有効になるのかを示唆できればと考えている．

3 つ目は、本手法のアイディアを実装することである．本手法では、マイグレーション元のプロセスのイメージを、他サイトにて復元すればジョブが再開すると仮定してるが、ここには多くの問題がある．並列ジョブのチェックポイント機構等の調査を進めながら、必要な技術的な問題を明らかにする必要がある．

7. まとめ

本論文では、複数プロセスによって構成されるジョブのマイグレーションの最適化手法として、プロセス間のメモリイメージの類似性に注目する方法を提案した．本手法は、総転送データサイズを削減するだけでなく、ベースとなるプロセスイメージと各プロセスの差分イメージをそれぞれのノード上で並列に計算するために、ノード数に対してスケーラブルである．評価実験の限定的なパラメータの範囲では、単純な手法と比べて、本手法の有効性を示すことが難しかったが、1) 構成プロセス数はなるべく多い方が良いことや、2) プロセス上のメモリイメージはなるべく小さくなるようにする方が良い、といった重要な知見を得ることができた．ノード数をさらに増やし、ジョブの規模を拡大したときに、本手法の有効性が証明される可能性が十分に高いことがわかった．

謝辞 本研究の一部は科学研究費補助金特定領域研究 (18049028) および JST-CREST「ULP-HPC: 次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング」の補助による．

参考文献

1) 立薗真樹, 中田秀基, 松岡聡: 仮想計算機を用いたグリッド上での MPI 実行環境, 先進的計算基盤システムシンポジウム (SACSIS2006) , pp.525-532, May 2006. (2006).
2) 佐藤賢斗, 佐藤仁, 松岡聡: 仮想クラスタを用いた Data-Intensive Application 実行環境の性能モデル構築と最適化, 情報処理学会研究報告, Vol.2008-HPC-111 (SWoPP2008), pp. 25–30 (2008).
3) 遠藤敏夫, 松岡聡, 橘爪信明, 長坂真路: ヘテロ型スーパーコンピュータ TSUBAME の Linpack による性能評価, ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2007), pp. 33-40, (2007).
4) : Dolly+. http://corvus.kek.jp/manabe/pcf/dolly/index.htm.
5) Percival, C.: Naive Differences of Executable Code, Technical report, Computing Lab, Oxford. University (2003).
6) Fava, A., Fava, E. and Bertozzi, M.: Mpipov: a parallel implementation of povray based on mpi, in: Proc. Euro PVM/MPI '99, Lecture Notes in Computer Science, Springer-Verlag, pp. 426–433 (1999).
7) 斎藤秀雄, et al.: InTrigger : 柔軟な構成変化を考慮した多拠点に渡る分散計算機環境, 情報処理学会研究報告, Vol. 2007-HPC-111 (SWoPP2007), No. 80, pp. 237–242 (2007).