Vol.2009-HPC-121 No.8
2009/8/4

並列アプリケーションの性能を損なわないポーリング型のモニタリング

鴨志田　良和[†1]　田浦　健次朗[†1]

同期が多い並列処理の実行時間は, 計算機上で動作している OS やデーモンの影響を受けて, 大きく遅延することがある. 1 秒に 2 回, システム情報を収集するのに 20 msec 程度かかるモニタリングシステムを動作させている場合, 256 ノード (4096 プロセス) で 10 秒間に 4000 回程度の All-reduce 処理を行うプログラムの実行時間が, モニタリングシステムがない場合と比較して 7 倍にもなる. このような遅延を削減するためには, (1) 各ノードのモニタリングデーモンを同期してスケジュールさせたり, (2) モニタリングの処理自体を大幅に軽量化したりすることが必要となる. 我々は, このような最適化の結果, 256 ノードの場合にモニタリングシステムを共存させたときの並列プログラムの実行速度を, 元の 1.2 倍程度に抑えることができることを実験により確かめた.

System Monitoring without Harming Performance of Parallel Applications

Yoshikazu Kamoshida[†1] and Kenjiro Taura[†1]

Parallel applications which perform collective communications causing synchronizations of processes many times are often delayed a lot by seemingly randomly scheduled background processes. Running time of a parallel program which does 4000 All-reduce communications in 10 seconds is multiplied by 7 when a monitoring process, which runs 20 msec per each 500 msec, is running on each node of the 256-node parallel environment. To reduce such delays, optimizations such as (1) simultaneous scheduling of monitoring daemons spreading many nodes and (2) substantial reducing of monitoring load are required. We developed an optimized version of monitoring system and ran it on the 256-node environment along with a parallel workload. As a result, we confirmed that delay of runtime with the optimized monitoring system is less than 20% of an original runtime.

1. はじめに

計算機上で実行されているアプリケーションの性能 (実行時間) は, その計算機上の OS やデーモンプロセスなどの動作などに影響されることがある. この影響は, ノード数が多い場合や, 実行するアプリケーションが同期処理を多用している場合に, より深刻になる. 実際どのような影響があるのかを端的に示したのが, 図 **1** である. この図に示されるように, どこかの CPU に, デーモンプロセスなどの計算に直接関係しないプロセスがスケジュールされる (Noise) と, 同期をしている処理全体が遅くなる. OS 等が原因で定期的に実行されるべきアプリケーションの同期処理の時間がずれることから, このような現象は OS jitter とも呼ばれる[4),15)]. ノード数が増えるほど, このようなデーモンプロセスの影響によって同期が遅れる可能性は高くなるし, 図 1 の灰色の領域で表わされる, 無駄な待ち時間も増加することになる. このような, 一部のノードで動いたプロセスが, 並列アプリケーション全体に影響を与える現象は, Cascading Effect とも呼ばれている[8)].

モニタリングシステムを実装するときにも, このことに十分注意する必要がある. 本稿で述べるモニタリングシステムは, 各計算機で適当な時間間隔でシステム情報をポーリングして取得するシステムを指す. モニタリングシステムが取得する情報は, システム全体の統計情報など, 取得にほとんど負荷がかからないものが多いが, より詳細な情報を取得することができれば, そのことで得られる利点も多い. たとえば, より短い時間間隔で情報を取得することにより短期的な負荷の偏りを把握することができたり, 不定形なアプリケーションの動作を把握するするために, 計算機上で実行されているプロセスごとの統計情報を活用したり, Autopilot[12)] などのソフトウェアのように, アプリケーションにフックをかけて, より詳細なパフォーマンス情報を取得して解析することにより, アプリケーションのチューニングに活用したりと, さまざまな用途に活用することが可能である. 多くのクラスタシステムで用いられている Ganglia[10)] でも, そのような詳細情報を取得するために, C や Python を使用してモニタリングシステムの機能を拡張するための API を提供している. このように, モニタリングシステムがより多くの仕事をすることが必要になる場合には, モニタリング用のデーモンプロセスが並列アプリケーションに与える影響が, より甚大なものになると考えられる.

†1 東京大学
The University of Tokyo

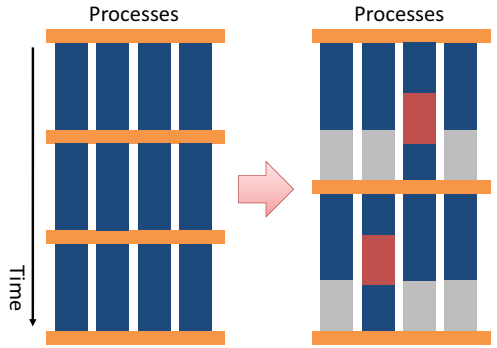

図 1　同期が並列アプリに与える影響

モニタリングの分野では, モニタリングシステムがモニタされる側のシステムに与える影響を intrusiveness と呼ぶ. 我々は, 豊かな情報を収集しつつ, intrusiveness がより低いモニタリングを行うための研究を行っている. この 2 者を両立するためには, モニタリングプロセスがシステムやアプリケーションにどのような影響を与えるかを把握することが重要である. 本稿は, 上に述べた背景のもと, モニタリングが並列アプリケーションに与える影響を減らす可能性があるいくつかの方法について, 実環境で評価を行い, その結果を報告するものである.

本稿は, 以降次のように構成される. まず 2 章でモニタリングシステムが並列アプリケーションに与える影響の大きさを示すための予備的実験についてその概要と結果を述べ, 次に 3 章でアプリケーションの同期処理をなるべく遅らせないための方法について考察し, 2 種類の最適化の方法を提案する. これに続き, 4 章で先に述べた最適化の方法を評価する. そして 5 章で関連研究との比較検討を行い, 6 章でまとめを行う.

2. 予備的実験

我々はまず, モニタリングプロセスが並列アプリケーションに, 実際にどの程度の影響を与えるのかを調べるための予備的実験を行った. モニタリング用のプロセス (pinfo) は, VGXP[9] で用いているシステム情報取得用のプロセスと同様のものを使用した. pinfo は, 定期的に実行され, ロードアベレージやメモリ使用量などのシステムの統計情報と, ノード上で動作している各プロセスの使用 CPU 時間などの統計情報を取得する. このプロセスをノードあたり 1 つ起動する. 本稿で行われたすべての実験では, 0.5 秒に 1 回の頻度でこの情報取得処理を実行した.

2.1 アプリケーションの概要

実験に使用したアプリケーションは, 不均質な物性値分布を有する立方体形状における三次元静的弾性問題を, 並列有限要素法 (Finite-Element Method，FEM) によって解く[16] もので, 連立一次方程式の解法には SGS(Symmetric Gauss-Seidel)[11] を前処理手法とし, 共役勾配法 (Conjugate Gradient，CG) 法を使用している.

アプリケーション内では以下に述べるような通信が行われる. 階層型領域間境界分割 (HID) によって各自の担当領域を分担したプロセスは, 繰り返しごとに, 隣接領域を担当するプロセスと境界部分のデータを交換する. このために 1 対 1 通信による送受信が 7 から 8 回行われる. その後, 全プロセスで `MPI_Allreduce` を 3 回実行する. この処理を収束するまで, 実験で与えた入力データでは 1250 回程度繰り返している. 全体の同期を伴う Allreduce 処理は, ベクトル同士の内積計算のために必要な処理であり, この頻繁なな同期処理があるため, デーモンプロセスの動作が実行時間に大きな影響を与えることが予想される.

2.2 実行環境の概要

実験は東京大学の HA8000 クラスタシステム上で行った. HA8000 クラスタシステムは, 東京大学がサービスを提供するスーパーコンピュータシステムで, 筑波大学，東京大学，京都大学の 3 大学で定められた「T2K オープンスパコン仕様」[14] に基づき日立製作所が製作した 952 ノード，約 15,000 コア，ピーク性能 140TFLOPS のクラスタ型コンピュータシステムである[13]. 各ノードは AMD quad-core Opteron(2.3GHz)4 ソケット，合計 16 コアから構成されており (図 **2**)，ノードあたりの記憶容量は 32GB(一部 128GB) である. HA8000 クラスタシステムは図 **3** に示すように，内部でクラスタ群に分かれている. 各クラスタ群内のノード間は Myrinet-10G(1 リンクあたり 1.25GB/sec ×双方向) で接続されている. ノード A 群は各ノード 4 本 (5.00GB/sec ×双方向)，ノード B 群は 2 本 (2.50GB/sec ×双方向) である. 本章の実験では, ノード A 群のうちの 8 ノード (合計 128 コア) から 64 ノード (合計 1024 コア) を使用した. アプリケーションのコンパイルには日立製専用コンパイラ (FORTRAN90) を使用した. 各ノード 16 コアを全て使用した.

2.3 予備的実験の結果

図 **4** の BASE が, ノード数を 8 ノード (128 プロセス) から 64 ノード (1024 プロセス) まで変化させて, アプリケーションを実行した時の実行時間である. 一方, PINFO は, このアプリケーションと同時に, pinfo を 0.5 秒に 1 回の頻度で実行した結果である. 図 **5** には,

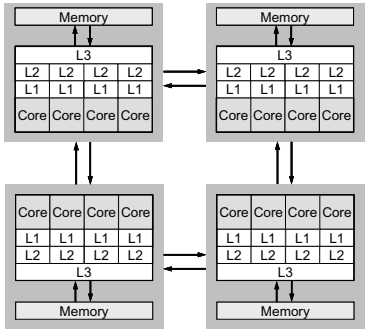

図 2　HA8000 クラスタシステムの, 各ノードの CPU. AMD Quad-core Opteron プロセッサ (2.3GHz) 4 ソケットから構成される.

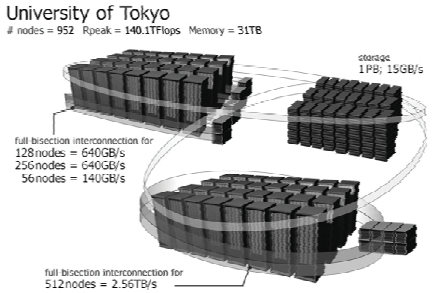

図 3　HA8000 クラスタシステムの全体図

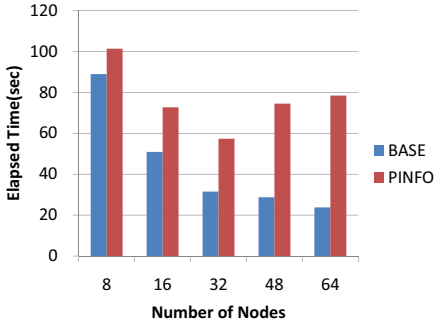

図 4　モニタリングプロセスの有無によるアプリケーション実行時間の違い
BASE/PINFO はモニタリングプロセスがない/ある場合

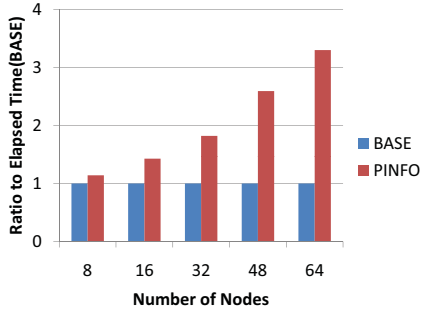

図 5　モニタリングプロセスの有無によるアプリケーション実行時間の違い
(BASE に対する PINFO の割合)

PINFO の BASE に対する割合を示した. このように定期的にシステム情報を取得するようなデーモンが動作しているだけで, アプリケーションの実行時間が大きく遅延することが分かる. しかも, 8 ノードの場合は数%の遅延にとどまるのに対して, 64 ノードの場合は元の 3 倍以上の時間がかかることからも分かるように, モニタリングプロセスが与える影響は, ノード数が増加するに従って, より大きなものになる.

3. 同期の待ち時間を減らす方法

予備的実験で得られた結果が示すように, モニタリングプロセスが実験で用いた並列アプリケーションに与える影響は, 無視できない大きさである. 本章では, このようなことが起きた原因とそれを防ぐ方法を考察する. 同期による遅延を減らすための基本的なアイデアは, モニタリング用のプロセスを,

(1)　なるべく同時に動くようにする
(2)　なるべく一度当たりの実行時間が短くなるようにする

という, 2 つのことである.

以下, 予備的実験の考察を行った後, それぞれの項目について述べる.

3.1 実行時間遅延の原因

実行時間の遅延を, 計算を行うアプリケーションと計算以外のプロセスを次のようにモデリングすることで見積もることにする. まず, アプリケーションは, 一定時間の計算をした後に同期を行う処理を繰り返すものとする. また, 計算以外のプロセスは, 各ノードでランダムなタイミングでスタートし, 一定間隔でスケジュールされ, 一定時間実行されるものであるとする. この場合, アプリケーションの実行時間の遅延が図 1 に示したように, 計算以外のプロセスが CPU にスケジュールされることのみに起因しているとすると, 計算時間の遅延は, 以下のそれぞれの積として表すことができる.

(1)　同期処理を行った回数
(2)　計算以外のプロセスが 1 回スケジュールされるたびに実行される時間
(3)　計算以外のプロセスが単位時間にスケジュールされる頻度
(4)　アプリケーションが 1 回の繰り返しにかかる時間
(5)　ノード数

上記の (2) を pinfo 単体での実測値に基づき 15(msec), (3) を 2 回/sec とし, 残りを pinfo なしでアプリケーションを実行したとき (BASE) の結果から決定し, 理論的な遅延時間を計算したものと, 実際に観測された遅延時間を図 **8** に示す.

理論値と実測値がおおむね一致しているため, 上に述べた現象が, 遅延の主な原因であると考えられる. モニタリングプロセスがスケジュールされる頻度を減少させずに, アプリケーションの遅延を改善する方法は, 2 つ考えられる. まずはランダムなタイミングでスタートせず, 各ノードのモニタリングプロセスが, 同期的に同じ時間にスケジュールされるように

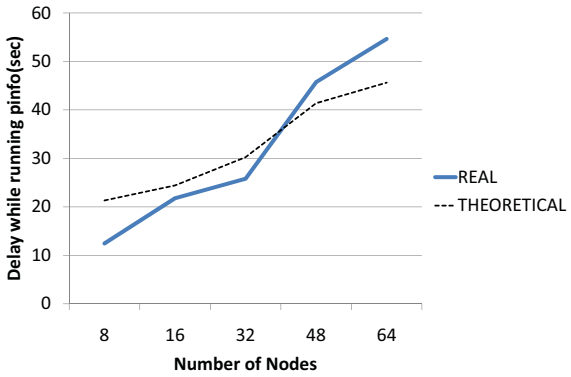

図 6　モニタリングプロセスによるアプリケーションの遅延時間

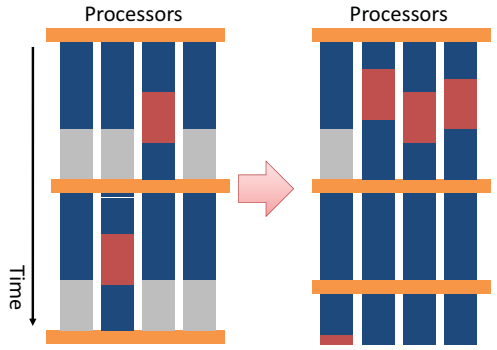

図 7　モニタリングプロセスの同期によるアプリケーション遅延の削減

すれば, アプリケーションの 1 度あたりの繰り返しでプロセスが遅延する確率が下がり, 影響を少なくすることが可能である (図 **7**). また, 遅延の要因の一つである, モニタリングプロセスが 1 回スケジュールされるたびに実行される時間を減らすことでも, 遅延時間を削減することが可能である.

3.2　タイミングをそろえる

アプリケーションの同期処理が遅延する現象は, あるひとつの CPU の処理の遅れが, その完了を待機している他の多数のプロセスに影響を与えることで, より深刻になる. 逆に, ある CPU の処理が遅れたときに, 他の CPU でも遅延が発生していれば, 相対的に影響は小さくなると言える. このために, モニタリングプロセスをノード間で同時にスケジュールさせることを検討する. ここで考えるモニタリングプロセスは, 一定時間ごとに情報収集を実行するものであるから, 最初の情報収集のタイミングがすべてのノードで揃っていれば, その後もすべてのノードで同時にプロセスが実行されると期待できる.

具体的には以下のように動作させる.

- 動作間隔を $I = 0.5(sec)$ とする.
- まず, すべてのノードでほぼ同時に現在時刻を取得して, ノード n でのこの結果を S_n とする.
- 情報取得プロセスは, 1 回の情報取得を行った後, $fmod(t_n - S_n, I) == 0$ となるまで sleep する.

ほぼ同時に現在時刻を取得するためには, MPI プログラムで各ノード 1 プロセスずつを含むコミュニケータを作成して MPI_Barrier を実行した後に, 各ノードのローカル時刻を取得することにした. 各ノードで MPI_Barrier から復帰した直後で同時にローカル時刻を取得しても, 厳密には, 物理的に同じ時刻を得られるわけではない. 通信遅延の分布が既知の場合に利用することができる Probabilistic Clock Synchronization[1] のような時刻同期プロトコルも提案されているが, アプリケーションの繰り返しにかかる時間や, モニタリングプロセスが情報取得にかかる時間と比べると, この時刻のずれは無視できる程度であると考え, 今回は上記の厳密でない方法を用いることとした.

また, クロックの進む速度の違いなどの理由により, 単に初期時刻をずらすだけでは並列処理のグローバル時刻を表すには不十分であるという報告[3] もあるが, 本稿で実行するアプリケーションの実行時間は短いため, このことは性能に大きな影響を与えないと考えている.

3.3　モニタリングプロセスの軽量化

モニタリング用の情報取得を行うプログラム pinfo が 1 回の情報収集で実行される時間は, HA8000 クラスタシステムで 200 程度のプロセスが動いている標準的な状態では, 15 msec から 20 msec である. この大部分は, 計算機上で動いている, プロセスごとの統計情報を取得する部分である. この処理では, `/proc` ファイルシステムから各プロセスの情報を取得する (これが, ps コマンドなどでも行われている最も普通の方法である) が, プロセスの情報は, プロセスごとに別々のファイルで提供されるため, プロセスの数だけ `open, read, close` システムコールが実行される. 実際の情報取得にかかる時間は少ないのだが, このシステムコールのオーバーヘッドのために, プロセス数にほぼ比例する実行時間がかかり, 1 度の情

報取得あたり, 10 msec 台の時間を必要としている.

この, プロセスごとの情報取得の負荷を減らすための方法として, 情報取得頻度をプロセスごとに動的に変更する方法と, Linux カーネルの taskstats インタフェースを利用する方法の 2 つについて述べる.

3.3.1 情報取得頻度の動的な変更

モニタリングの負荷の多くはプロセスごとの情報を取得する部分にある. そこで, 不必要な, あるいは, 影響の少ないプロセスについて, 情報取得頻度を下げることで, 1 度の情報取得でスキャンするプロセスの数を減らし, その結果として情報取得にかかる時間を短縮することを目指す. たとえば, カーネルデーモンの一部は, ほとんど CPU 時間を消費せずに眠っているものが多い. これらのプロセスについての情報取得頻度を減らすことを考える. この方法の効果を確かめるため, 以下の条件の両方が成立する場合には, プロセス情報の取得を行わないという変更を, pinfo に加えることにした.

- プロセス起動後から現在までの CPU 使用率が α を下回る
- 前回の情報取得時から新たなプロセスが生成されていない

後者の条件が満たされた場合は, プロセス一覧を取得するための処理 (`/proc` のディレクトリエントリから, 数字のみを含むディレクトリを取り出す処理) が不要になるため, その分の高速化も期待できる. 実験に用いたシステムでは, 10 秒に 1 回程度, システムが提供しているシステム監視用のシェルスクリプトが実行されるため, そのタイミングではすべてのプロセスがスキャンされることになる. また, 閾値 α には 10%を設定した. これにより, 後者の条件が満たされた場合はほぼ, 実行中のアプリケーションのみの情報が取得されることになる.

上記の変更を pinfo に加えることにより, 1 度あたりの情報取得時間は, 一部のプロセスのみをスキャンした場合は 2 msec から 3 msec 程度, 全プロセスのスキャンをした場合は 20 msec から 30 msec 程度の時間となった.

3.3.2 taskstats インタフェース利用

上記の方法は情報の正確性をある程度犠牲にして情報収集処理を高速化する方法であったが, Linux のプロセスアカウンティング機能の一部を使用することで, 情報の正確性を変えることなく高速化することが可能である. Linux カーネルソース内のドキュメントのひとつである, `Documentation/accounting/taskstats.txt` によると, Linux Kernel の taskstats インタフェースは, netlink ソケット (Linux カーネルとユーザ空間の通信を, ソケット通信を用いて行うインタフェース) を利用して, プロセス ID を指定してリクエストを送り, そのプロセスに関する情報をレスポンスとして受け取るという機能を提供している. このインタフェースを利用することにより, プロセスごとにファイルをオープンする必要が最小限に抑えられるので, モニタリングプロセスの CPU 使用時間を大幅に少なくすることができる. (ただし, プロセスの一覧を取得する部分は依然として`/proc` ファイルシステムに頼る必要がある.)

taskstats インタフェースから得られる情報は, カーネルのバージョンにより異なるが, このうち, プロセスが使用した CPU 時間に関する情報については, 広く利用されているカーネルバージョン 2.6.18 以降で取得可能である. このため, 実験ではプロセスごとの CPU 時間を取得することにした.

この taskstats インタフェースを利用してプロセス情報取得プログラムを記述することにより, 1 回の情報取得にかかる時間は 1 msec から 3 msec 程度となり, これまでの 10 分の 1 程度に抑えられることが確認できた.

4. 実　　験

上記の最適化を適用したモニタリングシステムを作成し, 並列計算への影響を評価した. 評価の際に使用したプログラムは, 予備実験で使用したアプリケーションをより単純化したものを用いた. 具体的には, 各プロセスが約 25ms 程度ローカルな計算をした後, 全プロセスで Allreduce を行う処理を 4096 回繰り返すものとした.

このプログラムを, HA8000 クラスタシステムの A 群の, 8 ノードから 256 ノードまでを使用して実行した時の時間を図 **9** に示す.

グラフで, BASE はモニタリングプロセスがない場合, PINFO は最適化前の pinfo を使用した場合, SYNC_ONCE は 3.2 で述べたタイミングをそろえる最適化を行った場合, SYNC_ALWAYS は計算に関わらない 1 ノードで同期用プロセスを立ち上げて, 各モニタリングプロセスがこの同期用プロセスとソケット接続を行い, 同期用プロセスが定期的にメッセージを送るのを待ってから情報取得を行った場合, OPT1 は pinfo に 3.3.1 で述べた「情報取得頻度の動的な変更」の最適化を行った場合, OPT2 は pinfo に 3.3.2 で述べた「taskstats インタフェース利用」の最適化を行った場合である. OPT1_SYNC_ONCE 等は, OPT1 と SYNC_ONCE の組み合わせである.

256 ノードの場合は, 実験のために割り当てた時間の制約により, 一部のケースしか試すことができなかった.

最適化前の pinfo を使用した場合, 128 ノード, あるいは 256 ノード使用時は, 実行時間が

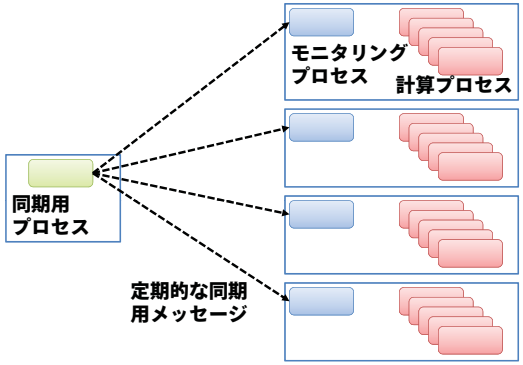

図 8　SYNC_ALWAYS 設定時の通信イメージ

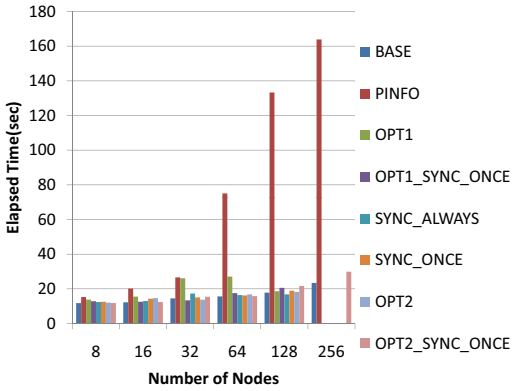

図 9　テストプログラムの実行時間

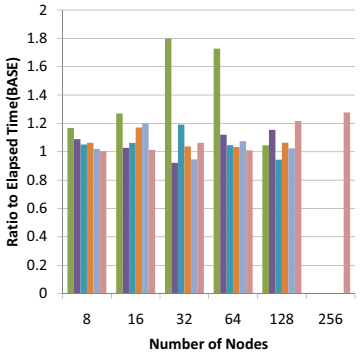

図 10　テストプログラムの実行時間 (BASE に対する割合)

元の 7 倍を超えており, 影響が甚大であることが分かる. 図 **10** には PINFO を除く各ケースについて, BASE に対する割合を示したものである. OPT1 のみでは, 32 ノードの場合に実行時間が元の約 1.8 倍になるなど, 大きな影響が出る場合もあることが分かる. これは, 頻度は減るものの, OPT1 では 20 msec 程度の遅延がランダムに発生することがあるためであると考えられる. OPT1 以外では, 最適化を施すと, 遅延への影響は多くても 20%程度に抑えられていることが分かった. しかしながら, 最適化前の pinfo と比べると大きく削減できたものの, 20%の実行時間増加というのは決して無視できる程度のものではない.

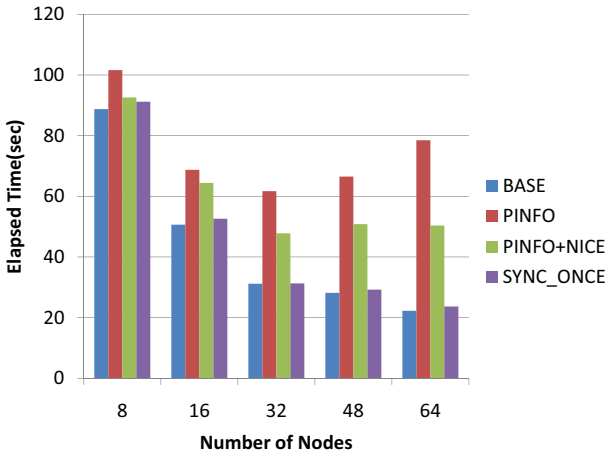

図 11　アプリケーションの実行時間

また, 毎回同期を行う場合でも, 最初に 1 度だけ同期を行う場合でも, この実験は, 少ないノードの BASE ケースが 10 秒程度で終わる非常に短いものだったため, 実行時間に大きな違いはないことが分かった. またこの実験だけでは, 観測結果に揺れが大きく, ノード数の増加に従って遅延への影響が大きくなる現象は PINFO 以外では明確に表れたとは言えなかった.

次に, 予備実験で用いたアプリケーションを, 最適化した pinfo と同時に実行した場合の結果を図 **11** に示す. PINFO+NICE は最適化前の pinfo を nice をかけて実行したときの結果である. テストプログラムの OPT1 の時と同様, 単に nice をかけて優先度を下げるだけでは遅延削減の効果は限定的であることが分かる. ノード数が増えるほど, nice の効果は少なくなる. また, SYNC_ONCE 最適化だけでも 64 ノード時の遅延は約 6% に削減できていることが観察された.

5. 関連研究

並列動作するプロセスが同時にスケジューリングされない場合に, パフォーマンスに影響がでることについてはSMPシステムでの並列計算の研究で古くから指摘されていた[5]. OS jitterが並列処理に与える影響は, ノード数が多い分散並列環境ではより深刻な問題になるため, ASCI-Whiteのように, OSのスケジューラに変更を加えて解決しようとしている[8]ものもある. また, Gang scheduling[2][7]のように複数の並列プロセスのスケジューリングの問題としても同様の研究が行われている. また, 計算以外のプロセスが与える影響を確率モデルとしてモデル化して解析する試み[15]も行われている. また, BlueGene/L等の, jitterの影響が少ないシステムでアプリケーションを実行したログを元に, 他のシステムでの性能予測を行う枠組みについての研究[6]も行われている.

我々の研究では, 運用中のクラスタシステムへの導入しやすさを考慮して, モニタリングシステムをユーザプロセスとして実装しつつ, 低いintrusivenessと詳細な情報収集をいかに両立するかを課題としている. 上に述べた先行研究で提案されているモデル化手法やデータの解析手法を参考にしつつ, 現実にスパコンセンター等で利用されているアプリケーションと効率的に共存できるような, モニタリングシステムの設計と実装を目指していく予定である.

6. おわりに

本稿では, モニタリングシステムが周期的な実行が並列プログラムに与える負荷についてのデータを, 256ノード, 4096並列規模の実環境で収集し, その基礎的なデータ解析の結果を報告した. その結果, 何も考慮しないでモニタリングプロセスを実行場合は, 実行時間が7倍にもなってしまうところを, 各ノードのモニタリングプロセスの実行を同期させたり, 個々の処理の最適化を行うことで, 実行時間をモニタリングシステムが動作していない場合と同じか20%程度の遅延に抑えることができることを示した.

今後は, 収集したデータの解析をさらに深く行い, 遅延が発生する際の詳細な仕組みを明らかにする. また, 他のOSプロセスや, Ganglia等の他のモニタリングシステムの動作の特性についても調査を行う. また, 実験で観測された, 実行時間のゆれについてもより詳細な解析を行う. また, 3.3.1において述べた動的な情報取得頻度の変更は, 実際にはこれまでのプロセスの動作の履歴等から, 情報取得をしないことによる誤差を評価し, 誤差が一定の範囲内に収まるように行うべきである. このような改良を行い, 低い負荷とデータの信頼性の高さを両立するような方法についても検討を行う予定である.

謝辞 本研究の一部は文部科学省次世代IT基盤構築のための研究開発「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」および文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新IT基盤研究プラットフォームの構築」の支援を受けている.

参考文献

1) Alari, G. and Ciuffoletti, A.: Implementing a Probabilistic Clock Synchronization Algorithm, *Real-Time Syst.*, Vol.13, No.1, pp.25–46 (1997).
2) Barbosa, F.A., Silva, B. and Scherson, I.D.: Concurrent Gang: Towards a Flexible and Scalable Gang Scheduler, *In Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing* (1999).
3) Becker, D., Rabenseifner, R. and Wolf, F.: Implications of non-constant clock drifts for the timestamps of concurrent events, *Cluster Computing, 2008 IEEE International Conference on*, pp.59–68 (2008).
4) Beckman, P., Iskra, K., Yoshii, K. and Coghlan, S.: The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale, *Cluster Computing, 2006 IEEE International Conference on*, pp.1–12 (2006).
5) Burger, D. C., Hyder, R. S., Miller, B. P. and Wood, D. A.: Paging tradeoffs in distributed-shared-memory multiprocessors, *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, Los Alamitos, CA, USA, IEEE Computer Society Press, pp.590–599 (1994).
6) De, P., Kothari, R. and Mann, V.: A trace-driven emulation framework to predict scalability of large clusters in presence of OS Jitter, *Cluster Computing, 2008 IEEE International Conference on*, pp.232–241 (2008).
7) Hori, A., Tezuka, H. and Ishikawa, Y.: Overhead Analysis of Preemptive Gang Scheduling, *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, London, UK, Springer-Verlag, pp.217–230 (1998).
8) Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., Maskell, B., Tomlinson, P. and Roberts, M.: Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System, *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, IEEE Computer Society, p.10 (2003).
9) Kamoshida, Y. and Taura, K.: Scalable Data Gathering for Real-time Monitoring Systems on Distributed Computing, *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2008)*, Lyon, France, pp. 425–432 (2008).

Vol.2009-HPC-121 No.8
2009/8/4

10) Massie, M.L., Chun, B.N. and Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation, and Experience, *Parallel Computing*, Vol.30, No.7 (2004).
11) Nakajima, K.: The Impact of Parallel Programming Models on the Linear Algebra Performance for Finite Element Simulations, *Lecture Notes in Computer Science 4395*, pp.334–348 (2007).
12) Ribler, R.L., Vetter, J.S., Simitci, H. and Reed, D.A.: Autopilot: Adaptive Control of Distributed Applications, *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, IEEE Computer Society, p.172 (1998).
13) T2K オープンスパコン (東大)：http://www.cc.u-tokyo.ac.jp.
14) The T2K Open Supercomputer Alliance: http://www.open-supercomputer.org.
15) Tsafrir, D., Etsion, Y., Feitelson, D.G. and Kirkpatrick, S.: System noise, OS clock ticks, and fine-grained parallel applications, *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, New York, NY, USA, ACM, pp. 303–312 (2005).
16) 中島研吾：マルチコアクラスタにおける有限要素法アプリケーションのための階層型領域間境界分割に基づく並列前処理手法，情報処理学会 研究報告 (HPC-119-18)，pp. 103–108 (2009).