大域分散並列コンピューティング

田浦健次朗

情報理工学系研究科電子情報学専攻

概 要

広域環境で，ノード数の増減を伴いながら並列計算を継続的に実行するプログラミングモデル Phoenix の設計と実装を行っている．今年度の成果報告として，Phoenix の設計と応用記述の方法論について執筆した論文 [4] の要約について述べる．また，今年度主として取り組んだ，資源発見と経路表構築，ならびに Phoenix を利用した，非占有環境で頑健に高性能を示す高速数値計算アルゴリズム (LU 分解) について述べる．

1 今年度の主な成果

非占有環境で頑強な性能を発揮する高速数値計算 (LU 分解): Phoenix の特徴である動的なノードの増減可能性を生かして，共有されたクラスタ環境 (非占有環境) で高性能を発揮する LU 分解アルゴリズムの設計と実装を行った [1]．環境の占有時には，128CPU の Xeon 2.4GHz クラスタにおいて，行列サイズ (一辺) 30,720 において 127GFlops を達成し，同じ環境での Atlas + HPL とほぼ同じ性能を得た．また，非占有時には Atlas + HPL を上回る性能を得た．また，計算途中に動的にノードを増加させ，負荷の割り当てを実行時に修正することによって，実際に追加したノードの分だけ性能が向上することも確かめられた．この結果により，非占有環境であいた資源を獲得し，混雑した資源を解放しながら高性能を維持し続ける，今後の環境で有望な，動的な高性能アルゴリズムの例を示すことができた．

Grid 用通信システムにおける高速な資源発見と経路構築: Phoenix の使い勝手と実装を大規模なスケールにおいて大幅に改善した [3]．Grid 環境における通信を簡便に行うためには，第一に，firewall や NAT などの制限を乗り越えてアプリケーションレベルでメッセージの経路付けを行う仕組みが必要である．また，現在典型的な大規模環境においては，静的なノード名を持たないノード (DHCP で設定されたノード)，初期故障ノード，計算途中に追加されるノード，動的にノードを選択するスケジューラ，などの存在により，ユーザが計算に参加するノードを事前に記述し，参加ノードに正確に伝えることが困難になってきている．そこで通信システムは計算が始まってから参加ノードを発見し，それらの情報を参加ノードに伝える仕組み (資源発見) を備える必要がある．そしてそのようにして発見された資源間で接続を確立し，経路構築を行う必要がある．我々のシステムは，多数のノード間で，ネットワーク設定上許されたペアすべての間で接続を行い，かつ構築された接続の上で任意ノード間のメッセージを経路づけする．我々がこのシステムの初期実装を行った結果，既知の経路表構築アルゴリズム (DSDV) を安直に適用しただけでは，経路更新のためのトラフィックが大量に発生し，50 を超えるノード数では経路表の構築に多大 ($> 1,000$ 秒) の時間がかかることがわかった．我々はこの原因を探求して，400 プロセスを超えても，事前に接続情報を (静的に) 記述した場合の数倍程度の時間 (30 秒程度) ですべての最短経路を求めることに成功した．また，Phoenix システムでは元々計算途中のノード増減を許しているために，完全に安定する以前の経路表でも多くのメッセージは配送でき，今回の実験でも 10 秒程度で 90% 以上のノード対について，経路が求まっていることが判明した．

また，上記以外の関連した研究として，

- 多数のノードへ並列にファイルコピーを行うアルゴリズムとシステム [2]
- Grid 環境で並列にコマンドを投入するシェル

に関する研究を行った．前者は 100 を超えるノードに大きなファイルを効率的にコピーするためのアルゴリズム

で，ネットワークの遅延を測定しながら動的に適切な経路を求めていく適応的なアルゴリズムである．大きなファイルのコピーという操作は分散システムを設定するのに頻繁に用いられる．実験により，150 ノード程度で，ほぼ最適な経路を構築することが確かめられている．

後者は SSH と Perl という，わずかなインストールベースのみを仮定して，簡単に 100 を超えるノードに対して並列にコマンドを投入することができるツールである．初期状態としてはローカルノードに対する SSH 公開鍵を各ノードが保持していることだけを仮定し，その状態から自動的に各ノードにインストールされ，コマンドの投入が行えるようになる．Globus などの大掛かりなソフトウェアを利用することなく簡単に，安全なコマンドの投入を，大量のノードに対して行うことができるユーザレベルのツールである．

2 Phoenix システムの設計と Phoenix による応用記述

2.1 動機

Phoenix システムは，今後広がりを見せるであろう，広域分散環境における高性能計算をサポートすることを第一の目標にして設計された，メッセージパッシングシステム (通信ライブラリ) である．これまでの通常のメッセージパッシングライブラリ (たとえば MPI) と異なり，計算に参加するノード数を増減させながら，ひとつの並列アルゴリズムを継続させることが可能で，かつ，通常のメッセージパッシングモデルとの連続性を保っている．計算に参加するノード数を増減させながら行う並列計算の形態として，今日非常に一般的なものにサーバ-クライアント型，ないしマスター-ワーカ型と呼ばれるパラダイムがあるが，それらと異なり，任意のノード間の通信 (メッセージ送受信) をサポートしている．これによって，サーバ (マスタ) のボトルネックに悩まされない実装が可能であり，従来 MPI などで記述されてきた高性能計算を自然にサポートすることができる．

Phoenix の実装は，広域分散環境のような，firewall や NAT によって TCP レベルでの接続性に制限がなされている環境において動作し，かつ高性能を達成している．具体的には，(1) LAN 内のスイッチで接続されている場合の様に，TCP レイヤでの接続性に制限がない限り，極力下位レイヤを直接用いる (1 ホップで通信を行う), (2) 一般には，許されている接続上で最短路を求めて，任意のノード対間の通信を経路付けする．また，静的 (永続的) な名前を持たないノードを容易に計算に参加させることができる．それは，各ノードが事前に，計算に参加する他の全ノードの名前を (設定ファイルなどによって) 知らされなくても良い，という資源発見機構によって可能になる．

2.2 基本的な設計

メッセージパッシングアルゴリズムにおいてノードの増減をサポートする際に直面する困難は，プロセッサ間がノード名 (通常は番号) を用いて通信するという，基本的な事実である．つまり，計算に参加するノードに逐次的な番号が割り当てられ，多くの並列アルゴリズムで，データ分割や負荷分散などの本質的な目的のために，計算に参加するノード番号の集合がアルゴリズム中不変であることを仮定している．たとえば良く用いられる配列のブロック分割では，配列 A の i 番目の要素 $A[i]$ をノード番号 $\lfloor \frac{iP}{n} \rfloor$ に割り当てる (P はプロセッサ数で，n は配列の要素数)．この状況で，あるノードが計算から脱退しようとすると，そのノードが持つデータを別のノードに移送し，それ以降そのノード番号に対して送られたメッセージを新しいノードに転送しなければならない．このデータ移送は，移送に関与しないノードからは透明に行われる必要がある．つまり，データの移送に伴ってノード番号とデータの間のマッピングが変化するのはプログラマの負担を増大させるため，望ましくなく，仮にそれを受け入れたとしてもマッピングの変化の通知のタイミングなど，複雑な設計上の問題を引き起こす．

Phoenix では，アプリケーションが通信のために (メッセージのあて先として) 用いるノード番号と，実際の物理的なノード名を分離し，それらの間のマッピングを動的に変化させることでこの問題に対処する．アプリケーションは，参加ノード数や集合によらない，**仮想ノード名**を用いて通信を行う．これは任意に (現在は 62 bit 以内の整数) 広大な空間を用いてよく，通常データ分割の最小単位の数だけの空間を用いる．たとえば n 要素の配列 (のみ) を分割する場合，自然な方法は $[0, n)$ という仮想ノード名を用いることである．

メッセージは仮想ノード名をあて先とする．その仮想ノードをどの物理的なノードが現在受け持っているか，は Phoenix API の呼び出しによって決まる．具体的には，

- ph_assume(S)
- ph_release(S)

という二つの API がある．S は仮想ノード名の集合で，それぞれ呼び出したノードが S の担当を開始する，終了する，という意味を持つ．Phoenix は，明示的にある仮想ノードを担当しているノードにのみ，その仮想ノード宛のメッセージを配達 (届ける) することを保証する．

2.3 Phoenix を用いた並列アルゴリズムの記述

Phoenix を用いて並列アルゴリズムを記述する場合の基本的な考え方は，仮想ノードとデータの間のマッピングは通常不変とし，物理ノードの増減時には，仮想ノードと物理ノード間のマッピングを (それに関与しない物理ノードからは透明に) 変更することで対処する，というものである．より一般的にはノードの増減のあるなしにかかわらず，データを物理ノード間で移送する際に，それにともなって対応する仮想ノードの担当も移送し，仮想ノードとデータのマッピングを不変に保つ，ということである．この考え方で，ノード参加・脱退に伴う必然的なデータの移送のほか，動的負荷分散のためのデータの移送なども，統一的に対処することができる．短く言い換えれば，Phoenix は整数 (仮想ノード番号) をオブジェクトの ID とした，migration 透明なメッセージ配達システムということもできる．

例として n 要素の配列を分割する場合，要素 $A[i]$ を仮想ノード i にマッピングするという規則を (プログラマが) 確立する．仮想ノードと物理ノードのマッピングは上述した API を用いて確立する．要素 $A[i]$ を物理ノード p から q に，何らかの理由 (例: 物理ノード p が脱退したい，物理ノード q が新しく参加した，負荷分散を調整したい) で移送したい場合，以下の手順をとる．

- p は ph_release({i}) を実行し，仮想ノード i の担当を終了する．
- p は要素 $A[i]$ を q に送信する．
- q は $A[i]$ を受け取り次第，ph_assume({i}) を実行し，仮想ノード i の担当を開始する．

p, q 以外のノードはこれを知ることなく計算を続けてよい．特に仮想ノード i に送られたメッセージは常に「データ $A[i]$ を保持する物理ノード」に送られている．

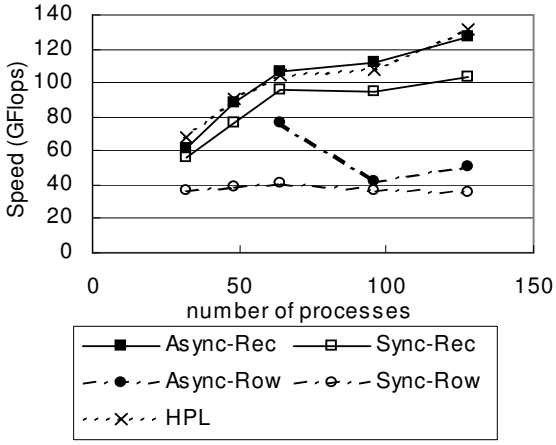

図 1: 固定ノード，占有状態での性能．

3 これまでの性能評価結果の要約

これまでに Phoenix を用いて，LU 分解，クイックソートなどのアプリケーションをノード数を増減させながら動作させることに成功しており，ここでは 1 節で述べた，並列 LU 分解の結果を要約する．このアルゴリズムは非占有環境で頑健な性能を発揮することを目指して設計されている．基本的な考え方は，通常のバリア同期を多用したアルゴリズムではなく，必要なデータがそろい次第，ブロックごとに計算を開始する，ということである．それに加え，実行時にプロセッサが追加したときに，負荷分散と通信コストを両立させるデータ分割法を採用している．

図 1 は固定台数の占有環境における性能を種々のアルゴリズムで比較したものである．環境は dual Xeon 2.4GHz をノードとする 69 ノードからなる Gibabit Ethernet で結合されたクラスタ (128CPU までを利用) である．CPU あたりのピーク性能は 4.8GFlops である．グラフ中，Async-Rec が上で述べた非同期アルゴリズムである．比較対象として，高性能な Linpack 実装としてよく用いられる HPL (BLAS ライブラリとして，自動性能チューン機能を備えた Atlas を利用) を用いたところ，グラフからわかるように，ほぼ匹敵する性能を得ている．我々が得た性能は Supercomputer top500Web サイト (http://www.top500.org/) などに掲載されている，類似のハードウェア構成の結果と比べると 50% 程度であり，その差異はまだ判明していないが，現時点では HPL と Phoenix が我々の環境でほぼ同等の性能を得ていることを重要視している．そして，差異の原因がわかり次第，両者の性能向上に貢献するであろうことを期待している．

図 2 の左は非占有環境での性能低下を，我々のアルゴリ

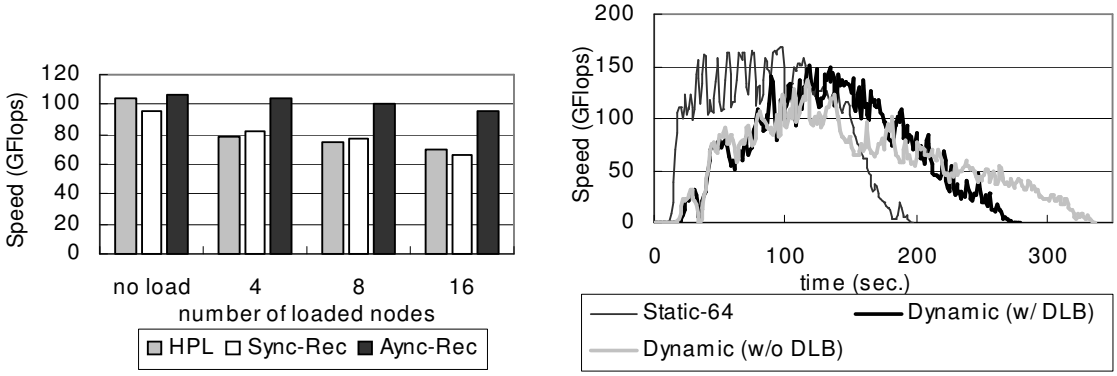

図 2: 一部のノードに負荷をかけた場合の性能低下の比較 (左) 台数の実行時増加に伴う性能向上 (右)

ズムと HPL で比較している．64CPU を用いている．横軸の数字は，占有されていない (負荷をかけた)CPU 数を示しており，そのような CPU では，約 40%の CPU share が本来の LU 分解プロセスに割り当てられるように調節されている．また，負荷をかけるノードはランダムに選び，10 秒ごとにランダムに変化させている．グラフからわかるように，HPL がわずかなノードにおける負荷に敏感に性能低下を示すのに対し，我々の非同期なアルゴリズムははるかに頑健な性能を示している．

最後に図 2 の右はノード数を実行途中で増やした際の，速度の時間変化をあらわしている．Static-64 のラベルがついているグラフが，最初から 64 ノードが投入されている場合，Dynamic は，最初に 16 台で開始させた後，プログラムの初期化後 (データ配置終了後) に 48 台のノードを追加した場合の性能である．w/DLB は，ノード追加後も負荷の調整を行った場合，w/o DLB は行わない場合である．w/DLB の場合，性能が Static-64 の場合と比べて，同等な性能に達するのに時間はかかるものの，最終的にはほぼ同等な性能に達していることが読み取れる．

4 まとめ

近年のネットワークのバンド幅向上の勢いは目覚しく，CPU の速度向上のそれを凌駕している．今日の一般的な分散環境に見られる，通信バンド幅の不足による実行可能な並列アプリケーションの制限は，早晩実質的に消え失せることが決定しているといっても良い．将来現れる，バンド幅に富んだ環境では，より密な協調を必要とする並列アプリケーションの実行を Grid 上でも実行できるようになることが期待できる．しかしそれを実現するためには，これまでのような占有された環境でしかまともに動作しない並列アルゴリズムや，プログラミングモデルからの脱却を図る必要がある．遅延に強いアルゴリズムの設計がまず必須であり，さらに，より動的な資源選択 (遊休資源を実行時に取得して活用したり，負荷の高い資源を実行時に計算から脱退させたりする) をサポートする並列アルゴリズム記述の枠組み，そのためのミドルウェアの完備が不可欠である．我々の結果は，高いバンド幅が利用可能な環境ではそれが，LU 分解のような伝統的な HPC アプリケーションでも可能なことを示している．

参考文献

[1] Toshio Endo, Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. High performance lu factorization for non-dedicated clusters. In *IEEE/ACM Symposium on Cluster Computing and Grid (CCGrid)*, 2004. (to apper).

[2] Takashi Hoshino, Kenjiro Taura, and Takashi Chikayama. An adaptive file distribution algorithm for wide area network. *Journal of Parallel and Distributed Computing Practices*, 2004. (to appear).

[3] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Routing and resource discovery in phoenix grid-enabled message passing system. In *IEEE/ACM Symposium on Cluster Computing and Grid (CCGrid)*, 2004. (to appear).

[4] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix : a parallel programming model for accommodating dynamically joining/leaving resources. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.