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

1　はじめに

複数のサイトや管理ドメインにまたがったクラスタ群を，多数のユーザによって共有する環境で，効率的で，長時間にわたって高信頼に動作を続ける並列計算の方式について研究を行っている [3, 1, 5, 4] .

2　GXP: 最低限の基盤と導入コストで高並列計算を行うツール

複数のクラスタや管理ドメインをまたがった計算機環境はファイルのアクセスやプロセス管理などにおいて，単一管理ドメインや SMP 環境と比較して不便であり，広域環境の導入コスト/導入利益を大きなものにしている．それらを解決する基盤ソフトウェアが多く研究･開発されているが，多くの場合，権限を持つ管理者による，多大なソフトウェアのインストール･設定の苦労が伴う．そして，複数管理ドメインでの環境構築は，管理者権限が分散しているために，欠けているソフトウェアのインストールや設定ミスの修復などに様々な人為的遅れが生ずる．

GXP[8, 7] は，最小限の基盤ソフトウェア (実質的には SSH のみ) で動作し，かつインストールが非常に簡単でしかも管理者権限を必要としないツールである．基本機能は多数の計算機に同時にログインして，一回のコマンド入力で多数の計算機上でプロセスを一斉に起動するという単純なものである．GXP はこれを SSH と Python という，Unix ではほとんどどこでも利用可能といってよい基盤 (のみ) の上で行うように設計されている．

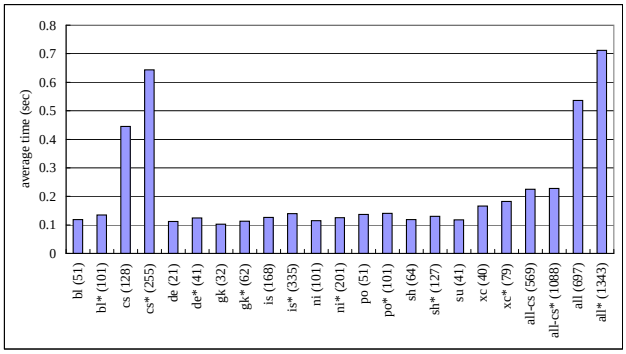


図 1: Average time of ‘echo -n’

GXP 自身のインストールは，一計算機上でのみ行えばよく，遠隔の計算機へは自動的にインストール (コピー) される．

プロセス起動は高速で，東京と徳島を含む 10 クラスタにまたがった環境で，1,000 プロセスを 200ms 程度で起動・終了する (図 1)．これはこれまでの類似のツール (MPD, dsh) よりもはるかに高速である (図 2)．図は，より小さな 180 プロセスでの比較実験であるが，この環境でも 10 倍以上高速である．比較対象の MPD は 1 クラスタ内のプロセス起動ツールであるために，1,000 プロセスでの比較は不可能であったが，可能であったとしてもその差はさらに広がる傾向が明らかに読み取れる．

この速度により GXP は多数の計算機上での仕事に快適な対話性をもたらす．また，ホストにまたがったプロセスの標準入出力を結合する機能により，限定された協調処理であれば，ネットワークプログラミングを一切行うことなく可能であり，我々はこれを用いて，クラスタをまたがって

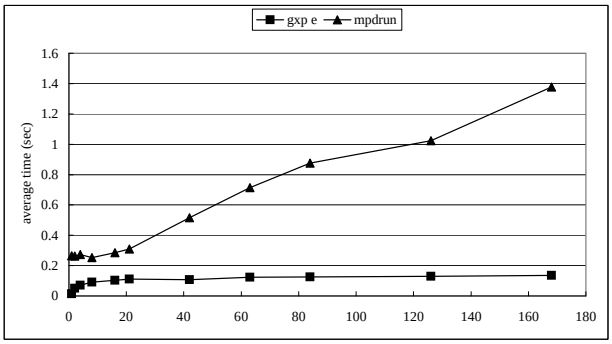


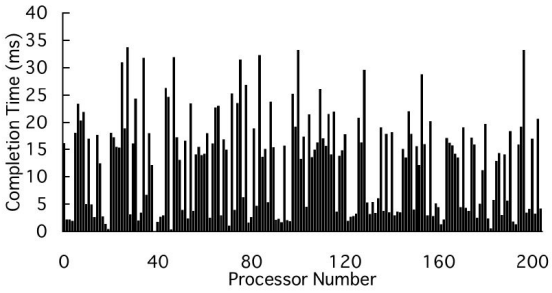
図 2: GXP e command and mpdrun

多数の逐次ジョブをディスパッチするタスクスケジューラを記述した．さらにこのタスクスケジューラを用い，大域知能辻井グループ，黒橋グループらと共同し，MEDLINE の 100 万アブストラクトの HPSG 構文解析，Web から収集した 5 億文の CFG 構文解析，などのテキスト処理を行った．いずれも複数のクラスタをまたがった 170 ノード/350 CPU 程度の CPU を用い，1-3 CPU 年の処理を 1.5 日-5 日程度で終了した．

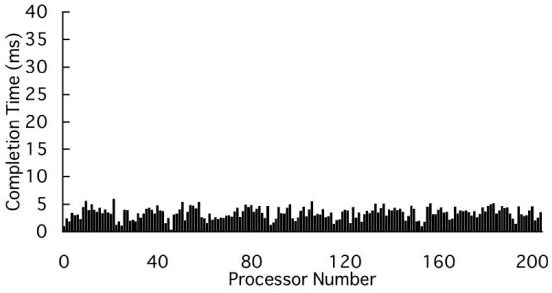
3 500 ノード/1000 CPU Grid Challenge 環境の構築

計算基盤に関する国内最大の会議 SACSIS 2005 (2005 5/18-20 筑波 http://www.hpcc.jp/sacsis/2005/) において併設企画として，Grid Challenge in SACSIS 2005 (http://www.hpcc.jp/sacsis/2005/grid-challenge/) という Grid プログラミング環境の構築と，そこでのプログラミングコンテストを企画した．東大 (本郷，柏)，東工大 (大岡山，すずかけ台)，徳島大，電通大，筑波大，産総研の 10 クラスタ，500 ノードを結び，960 CPU をコンテスト環境として提供した (表 1)．コンテストには 41 チーム，82 人が参加している．プログラミング環境として産総研がミドルウェア Ninf を提供し，我々は GXP を提供した．この試みは，この分野での若手人材育成，並列分散研究の実験場の提供による，中期的に見た研究の規模や質の大幅な向上，などの効果を期待しているものである．

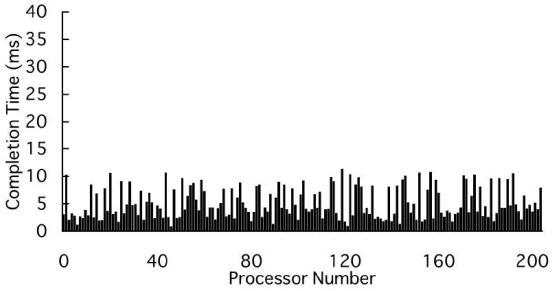
また，GXP はこの規模の環境でも問題なく動作することが確認され，問題データの作成，解答，環境の診断やトラブルシューティングなどに有効性が確認された．



(a) Topology-Unaware Implementation



(b) Topology-Aware Implementation



(c) Our Implementation

図 3: Broadcast over 201 processors in 3 clusters.

4 大域環境における自律的 spanning tree 構成

Local-area network (LAN) 用のメッセージパッシングシステムの集合通信は遅延が小さく且つ一

cluster	CPU type	nodes/CPUs	ネットワーク	所在地
bl	PIII 1.4GHz	50 / 100	100M	電通大
de	Xeon 2.4GHz	20 / 40	100M	筑波大
gk	PIII 1.4GHz	30 / 60	100M	東工大 (すずかけ台)
is	Xeon 2.4GHz	107 / 214	GigE	東大 (本郷)
ni	Athlon MP 2000+	100 / 200	100M	東工大 (大岡山)
po	Athlon MP 2000+	50 / 100	100M	徳島大
sh	Xeon 2.4GHz	63 / 126	GigE	東大 (柏)
su	Xeon 2.4GHz	40 / 40	GigE	東大 (本郷)
xc	Xeon 2.4GHz	40 / 80	GigE	産総研
total		500 / 960		

表 1: Grid Challenge 環境

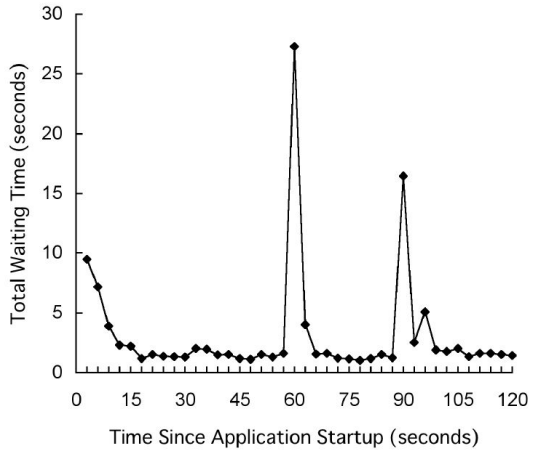


図 4: The total waiting time of all processors when the processors left (60 secs after start-up) and re-joined (90 seconds after start-up) during broadcasts.

様という仮定に基づいて作られているが，WAN ではプロセッサ間の遅延が大きく且つばらつきがあるので，全く性能が出ない．これまでに，WAN において効率良く集合通信を行う手法が提案されているが，それらの手法はトポロジーが事前に分かっており，プログラムの実行中には変わらない静的なものと仮定している．しかし，WAN における長時間に渡る大規模な計算では，用いる計算資源を動的に変えられることが不可欠である．なぜならば，他のユーザのために計算資源を解放したり，計算開始後に新たに使用可能になった計算資源を投入できたり，あるいは一部の計算資源が故障しても計算が続行できるような必要があるからである．

我々が提案する集合通信 [6] は，トポロジーが変化する環境でも効率良く行える．提案手法では，実行時に各プロセッサをルートとするスパニングツリーをプロセッサの数だけ動的に生成・維持し，これらのツリーを用いて集合通信を行う．トポロジーが変化している時にはツリーが崩れてしまうが，その場合には point-to-point 通信を用いることによってメッセージが全プロセッサに確実に届くようにする．この手法の特徴は以下の通りである．

- トポロジーが安定している時は効率が良い
- トポロジーが変化している時も確実に行える
- トポロジーの変化に迅速に対応できる

我々は提案手法を Phoenix メッセージパッシングライブラリの拡張として実装し，3 つの LAN にまたがる 201 プロセッサで評価実験を行った．スパニングツリーは 18 秒で収束したが，その間も集合通信は正しく行われた．そして，スパニングツリーの収束後にはブロードキャストがトポロジーを考慮しない実装に比べて 3 倍速く行え，トポロジーを事前に与えて生成した静的なツリーを用いた実装と比べても 2 倍以内の性能が出た．また，実行中に一部のプロセッサが参加・脱退してもブロードキャストは残りのプロセッサに伝わった (図

3)．スパニングツリーが再構築されるまで一時的にブロードキャストにかかる時間が長くなったが，その後再び効率良く行えるようになった (図 4)．

5 耐故障並列計算を支援する故障通知機構

広域の分散資源上で並列に動作するアプリケーションにとって、プロセスやノードの故障に動的に対処する機能を備えることが不可欠になってきている。そのような耐故障性を考慮した設計を行う際、どのプロセスが故障したかを迅速かつ正確に知ることが求められる。故障検知機構はそれを手助けするシステムであり、近年では数多くの効率的手法が提案されているが、自律性や効率性の面で問題があることが多い。

我々が提案する故障検知機構 [2] は、自律的に動作し、効率的に故障を全プロセスに通知する。具体的には、各プロセスがランダムに選択した k 個のプロセスに対して自分の監視を依頼することにより分散した監視体系を構築し、故障を検知したプロセスはその監視ネットワークを通じて効率的に全プロセスに通知するというものである。

故障が発生すると、監視体系も一時的に破壊されるが、各プロセスは自分を監視しているプロセスが k 個以上になるよう監視体系を自律的に修復していく。新たにプロセスが参入してきたときもそのプロセスが監視の依頼を行うだけでよく、この手法はプロセスの増減に対応することができる。

故障検知に要するプロセスあたりの平均 heartbeat メッセージ数は k 個なので、プロセス数が増加しても負荷はほぼ一定に抑えられる。我々の手法及び従来の代表的な自律的故障検知手法について、故障検知機構が動作している上でフィボナッチを実行したときのオーバヘッドの比較実験を行ったところ、我々の手法はプロセス数によらず高々２%程度に抑えられており、スケーラビリティが優れていることを示した（図 5）。既存手法の gossip も同様にスケーラビリティの高さが表れているが、この手法は故障検知までの時間が heartbeat 間隔の 30 倍程度に設定する必要があり、迅速に故障を検知することができないという問題がある。一方、我々の手法は heartbeat 間隔の数倍に設定すればよく、迅速な故障検知を実現することができる。

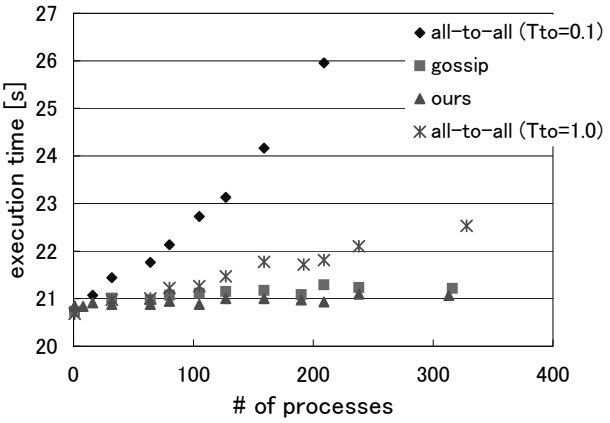


図 5: Execution time of a fibonacci program under failure detection services

また、本手法は故障検知の伝搬も効率的に行われる。動的に変化する任意のトポロジーにおいて情報を迅速に伝搬する手法としては、ルーティングプロトコルなどで用いられている flooding が一般的であるが、最悪の場合 $O(n^2)$ のメッセージが発生する。これでは故障情報伝搬時の負荷が大きくなり、伝搬効率も悪化させる恐れがある。そこで我々は前述した監視ネットワーク上に flooding を行うことにより、メッセージ数の削減及び伝搬効率の向上を実現する。本郷にある 108 台のクラスタと柏にある 65 台のクラスタの上で我々の故障検知機構を動作させ（heartbeat 間隔:0.1[s]、timeout 時間:2.0[s]、k=3）、その中の１プロセスを人為的に故障させたときの故障検知時間を図 6 に示す。172 プロセスでの情報伝搬は約 4.0 ミリ秒で達成され、このときの総メッセージ数は約 700 個に抑えられた。同様の実験を flooding で行ったところ、情報伝搬は約 8.0 ミリ秒、総メッセージ数は約 14000 個という結果となり、本手法はその約２倍の伝搬速度、約 20 分の 1 へのメッセージ数削減を達成できていることになる。

なお、この故障検知機構の枠組みは、我々が開発しているグリッド対応通信ライブラリの Phoenix

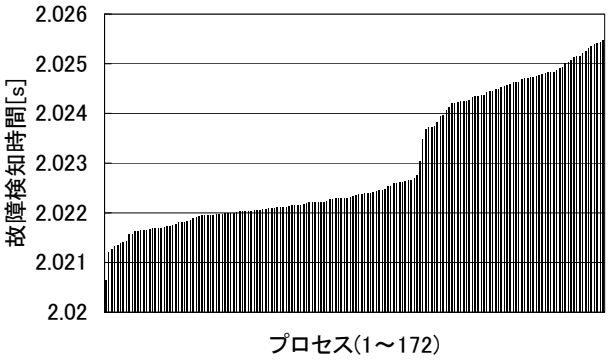


図 6: Failure detection time

にすでに組み込まれている。Phoenix では、ユーザプログラムが故障を知るための API が用意されており、実際にそれを利用して耐故障並列計算が記述されている。

参考文献

[1] Toshio Endo, Kenjiro Taura, Kenji Kaneda, and Akinori Yonezawa. High performance lu factorization for non-dedicated clusters. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'04)*, 2004.

[2] Yuuki Horita, Kenjiro Taura, and Takashi Chikayama. Phoenix プログラミングモデルにおける故障検知ライブラ リ. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing 2004 (SWoPP 2004)*, 2004.

[3] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Routing and resource discovery in phoenix grid-enabled message passing library. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'04)*, 2004.

[4] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. 接続を動的に制御するメッセージパッシングシステム. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing 2004 (SWoPP 2004)*, 2004.

[5] Hideo Saito, Kenjiro Taura, and Takashi Chikayama. Expedite: An operating system extension to support low-latency communication in non-dedicated clusters. In *Proceedings of Symposium on Advanced Computing Systems and Infrastructures (SACSIS) 2004*, 2004.

[6] Hideo Saito, Kenjiro Taura, and Takashi Chikayama. Phoenix プログラミングモデル用の集団通信. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing 2004 (SWoPP 2004)*, 2004.

[7] Kenjiro Taura. GXP homepage. http://www.logos.ic.i.u-tokyo.ac.jp/phoenix/gxp_quick_man_ja.shtml.

[8] Kenjiro Taura. GXP : An interactive shell for the grid environment. *2004 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA 2004)*, 2004.