耐故障並列計算を支援する自律的な故障検知機構

堀田勇樹† 田浦健次朗† 近山隆†

本稿では，並列計算の耐故障性を支援する故障検知の自律的かつ効率的手法について提案する．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で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用いて効率的に全プロセスに通知する．この監視体系は一部接続性が制限されている場合でも自律的に構築され，複数故障に対して高い耐性を持ち，故障により破損しても各プロセスによって自律的に修復される．313 プロセスで動作させたときのオーバヘッドは，heartbeat 間隔を 0.1 秒に設定してもたかだか 2%程度に抑えられており，他の自律的故障検知手法と比較して効率性が高いことを示した．

Autonomous Failure Detection for Supporting Fault Tolerant Parallel Computation

Yuuki Horita,† Kenjiro Taura† and Takashi Chikayama†

In this paper, we propose an autonomous and efficient failure detection service for fault tolerant parallel computation. We dynamically create distributed monitoring relations among joining processes by each process's asking k random processes to monitor itself, and efficiently notify some failure information detected by some processes to all the proceses using the monitoring network. This system is autonomously constructed even if the connectivity is limited, has high tolerance to multiple failures, and is rebuilt autonomously. The overhead with 313 processes was at most around 2 percent even if the heartbeat interval was set to 0.1 second, and we showed that our approach was more efficient than other autonomous ones.

1. はじめに

耐故障性は大規模な科学技術計算にとって重要な要素の 1 つである．PC クラスタなどの大量の分散資源上で並列実行することで計算力を得ることが一般的になっている今日においては，ますますアプリケーションが故障を受けやすくなっており，特に信頼性の低いネットワークで接続された分散資源を利用するグリッド環境においては，ネットワーク障害の影響も無視できなくなっている．そのため，長時間にわたり並列実行するアプリケーションにおいては，耐故障性を考慮した設計が必要となってくる．

故障検知は，並列計算に耐故障性を持たせるためには不可欠なプロセスである．並列計算を記述する通信ライブラリの中には耐故障性を支援する枠組みを備えたものもあるが[1)~3)]，故障検知をオペレーティングシステム（OS）が通知してくれる通信エラーに依存していることが多い．しかし，これではノード自身の故障やネットワーク障害に対応することはできない．耐故障機構の機能を十分に発揮させるためには，故障の種類に依存しない一般的な故障検知手法を通信ライブラリに組み込むことが求められる．

既存の故障検知手法は数多く存在するが，資源管理システムを対象として設計されたものが多い[4),5)]．資源管理システムは，人間が資源情報を管理するために特定のプロセスに定期的に情報を収集させるものであり，それほど迅速さや精度を求められるものではない．それに対し，アプリケーションの耐故障性の支援を対象とした場合は，検知した故障情報は全プロセスが取得できなければならず，かつそれは迅速で精度の高いものでなければならない．そのため従来手法は，こうした要件に合致していないことが多く，そのまま転用するのは得策であるとはいえない．

そこで本稿では，並列計算の耐故障性を支援するにあたり故障検知機構に求められる要件を改めて検討し，それに適した故障検知アルゴリズムを提案する．各プロセスはトポロジ情報などの設定を必要とせず自律的に監視体系を構築し，故障発生時には一貫性のある故障情報を効率的に提供する．

† 東京大学
The University of Tokyo

本稿の構成は以下のようになっている．2 章で耐故障並列計算を支援している既存の通信ライブラリにおいて，現在用いられている故障検知手法の問題点について述べ，本研究の動機を明確にする．3 章で耐故障並列計算を支援するための故障検知機構が満たすべき要件について整理し，既存手法について検討する．4 章で我々の提案する自律的かつ効率的な故障検知手法のアルゴリズムを説明し，検証を行う．そして 5 章で実験・評価を行い，最後にまとめる．

2. 耐故障並列計算と故障検知機構

並列計算の耐故障性に関する研究は近年さかんに行われている．

FT-MPI[1] は，MPI を拡張してアプリケーションを構成するプロセスに故障が発生した場合に，MPI 関数の返すエラーや能動的に故障をチェックする関数を呼び出すことによって，他のプロセスが故障情報を取得できるようにしている．

MPICH-V2[2] では，非同期な checkpoint および pessimistic message logging を行うことにより，ユーザ透過に耐故障性を実現させている．

MPI/FT[3] は，プロセスを冗長化することによって耐故障性を実現している．coordinator と呼ばれる管理プロセスが，冗長プロセス間のメッセージの制御やログなどの管理を行い，プロセスが故障したら再起動させる．

このように並列計算に耐故障性を持たせる手法は多様であるが，いずれもそれを実現させるためには，故障発生後にどのノードのプロセスが故障したのかを知る必要がある．MPICH-V2 をはじめとするほとんどの耐故障通信ライブラリは，故障の検知に OS が返す通信エラーを用いている．しかし，それはプロセスが故障したノードの OS と通信できる場合に限って有効であり，ノード自体の故障やネットワーク障害には対応することができない．

一方 FT-MPI や MPI/FT は，一般的な故障検知手法である heartbeat メカニズムを用いて厳密な故障検知機能を提供している．heartbeat メカニズムとは，heartbeat と呼ばれるメッセージを定期的に送信してもらい，ある一定時間 heartbeat を受信しなかったら (i.e. timeout)，そのプロセスが故障したと判断するというもので，任意の障害を検知することができる．

FT-MPI には，watchdog と呼ばれる故障検知モジュールが用意されており，ユーザはオプションで利用することができるようになっている．この watchdog は各ノード上で起動し，定期的に全ノードと heartbeat を ping-pong することにより互いを監視しあう．そのため，各 watchdog の負荷はノード数の増加にともない大きくなり，スケーラビリティに欠ける．

MPI/FT では，各プロセスに heartbeat を送信させ，coordinator がすべてのプロセスを監視している．そして故障を検知したら，全体にブロードキャストして全プロセスに通知する．だが，この手法も FT-MPI の watchdog と同様に coordinator におけるスケーラビリティに問題があり，さらに coordinator が single point of failure となるという耐故障性の問題がある．

故障検知は，耐故障並列計算にとって基盤となる欠かせない要素である．しかし，既存の耐故障並列計算の枠組みに含まれている故障検知機構は，決して十分であるとはいえず，故障検知漏れや性能悪化を許してしまっているのが実状である．そこで我々は，そのような耐故障並列計算の枠組みが最大限に効果を発揮できるような故障検知機構を提供することを考える．

3. 要件の整理と従来手法の検討

耐故障並列計算を支援することを想定した故障検知機構は，資源管理システムのような用途と比較して，求められることがより厳格になる．そのため，資源管理システムの分野で効果を発揮する手法であっても，そのまま転用するのは難しい．この章では，耐故障並列計算を支援する故障検知機構に求められる要件を整理し，それをふまえて従来の手法について検討する．

3.1 要件の整理

故障検知機構が満たすべき特徴として，完全性・正確性・効率性が議論の対象となることが多い．しかし，完全性と正確性を同時に 100%満たすことが不可能であることが示されているように[6]，トレードオフの関係が絡みあっており，評価が難しい．また，耐故障並列計算の支援の観点から見た場合，これだけの要素では不十分だったり，必ずしも今回の要件と合致しなかったりする点も存在する．そこで我々は，以下に述べる自律性・一貫性・効率性という 3 要素に集約して，今回の故障検知機構が満たすべき要件を整理する．

3.1.1 自 律 性

故障検知機構も分散システムの一種なので故障の影響を受けることになるのだが，その故障を検知して他のアプリケーションに通知する役割を担う性質上，高い耐故障性を持ち故障発生下でも正常に動き続けなければならない．完全性とはこの耐故障性のうち故障検知漏れという観点から見たものであるといえるが，実用面から考えると完全性だけでなく正確性やその他の性質も含め機能的に正常に動作しているかどうかが重

要となる．また，システムに新たにプロセスが追加されることも考えられるため，故障などによるプロセスの減少だけでなく，増加に対しても柔軟に対応できる必要がある．このように故障検知機構には故障やプロセスの動的参入などの環境の動的変化に対して自律的に対応することが求められる．

その一方で，システムの自律的な構築も重要な要素となる．構成要素の配置やトポロジを手動で設定する必要があるシステムでは，実行環境が変わるたびに再設定する手間が発生する．これではノード数が増加し実行環境が複雑になればなるほど設定の手間が大きくなり，アプリケーションを扱いにくいものにしてしまう．それに対し，システムに参加するノード情報などの最低限の設定情報だけで，実行環境に応じて自動的にシステムが構築されるよう設計されていれば，ユーザにとって利用しやすいものとなる．

したがって故障検知機構には，開始時のシステム構築や実行時の環境の動的変化に対して人手を介さずに自動的に対応できる「自律性」が重要であると考えられる．

3.1.2 一 貫 性

故障検知における一貫性とは，故障検知機構が提供する故障情報が全プロセスで必ず一致していることを意味する．

故障が起きると計算の状態の一部が失われることになるのだが，それを残りのプロセスが何らかの形で復元し計算を再開させることが，故障から回復するために必要な処理となる．マスタ・ワーカのように各プロセスの計算が独立しているような並列計算では，失われた部分を再計算させることで容易に回復可能だが，一般的な並列計算ではプロセスどうしが依存しあっていることが多い．そのため，あらかじめ計算の状態を定期的に故障によって失われない形で保存しておき，故障が発生したときは全プロセスが一貫性のある正常状態までさかのぼることによって，計算を復旧させることになる．このときもしも 1 つでも足並みを乱すプロセスが存在すると，計算状態の一貫性が保てなくなり，計算が正常に進まなくなる．

したがって，故障検知機構から提供される故障情報は一貫性を保っていなければならない．極端な話，たとえその故障情報が誤ったものであっても，一貫性が満たされていれば，効率面での実害はあっても正常な状態に復帰できる可能性は高い（たとえば，故障とされたプロセスを本当に故障したものとして計算から切り離せば通常の故障と同様に扱える）．最も避けるべきなのは，故障情報が一部のプロセスに伝わらなかったり，一部のプロセスのみが誤って正常なプロセスを故障と判断してしまったりすることであり，その場合計算を再開不能状態にしてしまう恐れがある．

3.1.3 効 率 性

故障発生時に並列計算がより早く復旧するためには，故障を迅速に検知することが重要になる．この故障検知の効率性は，heartbeat 間隔や timeout を短くすることによって実現することができる．しかし，heartbeat 間隔を短くするほど，故障検知機構による負荷は大きくなり，本来の計算の性能に悪影響を及ぼす．また，timeout を短くすると故障情報の正確性が悪化する．このように効率性は，負荷や正確性の面でトレードオフになっており，それぞれを単独で評価することはできない．

そこで本稿では，故障検知の速度・負荷・正確性を統合的に評価する指標として，改めて「効率性」という用語を用いることにする．具体的には，3 要素のうち 2 つを固定して残った要素の数値が，「効率性」の質を定めるものということになる．たとえば負荷を基準にして効率性を測るのならば，故障検知速度および誤検知確率がある一定の値になるよう heartbeat 間隔と timeout のパラメータを定め，そのときの負荷を測定する．それが小さいほどその故障検知機構の効率性が高いということになる．

また，効率性はスケーラビリティの面から評価するべきである．プロセス数が少ないときはどのような手法であれ効率性にはさほど差は生まれない．重要なのはプロセス数が多いときにもどれだけ効率性を維持できるかであり，それが故障検知機構の性能の観点から見た総合的な評価となるといってよいだろう．

3.2 従来手法の検討

この節では，3.1 節で整理した要件をふまえて，従来手法について検討していく．

3.2.1 階 層 型

サーバ・クライアント型システムのように，すべてのプロセスの故障を監視するプロセスを用意すると，システムとしてシンプルになり実装が容易になる．また，発生するメッセージ数も $O(n)$（n：プロセス数）に抑えることができるという利点もある．しかし，この手法ではプロセス数の増加にともない監視プロセスに負荷が集中し，計算性能を悪化させてしまう恐れがある．

そこで，これを改善するために，監視プロセスを階層的に構成することにより負荷を分散させる方式がよく使用される．監視プロセスを適当な箇所に（ex. クラスタごとに）複数配置・分担させ，それらの監視お

よびプロセス情報の収集を行う管理プロセスを用意する．これにより，プロセス数が増加しても，監視プロセスの数を調整することによって，各監視プロセスにかかる負荷を小さく抑えられる．この手法は，トポロジに沿って監視プロセスを配置することにより，広域に分散したノード上で動作させせることが可能であり，管理プロセスに効率的にすべての情報を集約できるので，主に MDS[4)] や NWS[5)] などの資源管理システムに利用されている．

階層型は，特定のプロセスが故障情報を管理するため，その情報を全プロセスに漏れなく伝達できれば一貫性の要件を満たすことができる．また，特定のプロセスにのみ heartbeat を送信すればよいため，監視プロセスの配置を適切に行えば，メッセージ数を小さく抑えられ，高い効率性を実現することができる．

しかし，監視プロセスが故障すると対処できないという耐故障性の問題がある．また，実行環境が変わるたびに構成を手動で設定しなければならず，ノード数が増加するとそれが煩雑になり扱いにくい．したがって，自律性が満たされていないといえる．

3.2.2 Gossip 型

Gossip 型[7)] とは，各プロセスの heartbeat メッセージが最後にいつ送信されたかという情報（gossip list）を定期的にランダムな相手に送信しあい，共有することにより，噂が広まるようにして heartbeat を全体に伝える手法である．ある一定回数送信しあえば非常に高い確率で全体に情報が到達することが示されており，heartbeat が最後に送信された時点から一定回数時間情報が更新されなかったプロセスを故障と判断する．

この手法は，各プロセスの機能が対称的であるため，どのプロセスが故障しても同じように対処することができる．短時間に大量のプロセスが故障すると，heartbeat が全体にうまく行き渡らず誤検知してしまう恐れがあるが，ある程度の複数故障に対応することができ，高い耐故障性を得られる．また，プロセスの新規参入に関しては，新規プロセスを検知したら gossip list に加えて周りに送信することにより，heartbeat と同じように容易に全体に反映することができる．したがって，自律性において優れているといえる．

また，heartbeat が全体に伝達されるまでの回数はプロセス数が増加してもあまり変化しないため，ある故障検知速度において一定の正確性を維持するために必要なメッセージ数はほとんど変わらず，スケーラビリティが高い．ところが，少なくとも heartbeat を数十回送信しなければ全体に伝達されないため，故障と判断するまでの timeout が heartbeat 間隔と比較して長めに設定せざるをえないという問題があり，効率性のベースに若干難点がある．

さらに，各プロセスが独自に故障を判断するため，一部のプロセスが誤って故障と判断してしまうと故障情報が全体で一致しないという状態になる．よって一貫性の要求を満たしているとはいえない．

4. 耐故障並列計算を支援する故障検知機構

3.2 節で議論したように，代表的な従来手法はいずれも要件を満たしているとはいえない．そこで，我々は 3.1 節で整理した要件を満たす故障検知アルゴリズムを提案する．

我々の基本アルゴリズムを**図 1** に示す．基本構成としては，一部のプロセスで故障を検知し，故障情報を全体にブロードキャストするという形式をとる．Gossip 型のように，各プロセスに故障の判断を委ねる方式では，誤って判断してしまった場合の一貫性を保ちにくいためである．そこで以下では，このアルゴリズムを故障検知部分と故障情報伝達部分に分けて順に述べていき，検証を行う．

4.1 故障検知

本アルゴリズムでは各プロセスが監視体系を動的に構成する．まず各プロセスは，自分が接続を確立しているプロセス（以下，隣接プロセス：neighbors）の中からランダムに k 個選択し，自分の監視を依頼する（図 1 の Init）．それを受け取ったプロセスは，ただちに送信元に ack を返し heartbeat の監視を開始する．そしてある一定時間（$T_{hb}+T_{to}$）そのプロセスから heartbeat を受信しなかったら故障と判断する．基本的にはこれだけの単純なアルゴリズムである．各プロセスを監視しているプロセスが k 個存在することになるので，どのプロセスが故障しても同時に k 個の監視プロセスがすべて故障しない限り，必ずプロセスの一部が故障に気づくことになる．

ただし注意しなければならないのは，故障が発生すると，開始時に動的に作成した監視体系の一部も破壊されてしまうことである．たとえば $k=1$ のときにおいて自分の監視プロセスが故障してしまうと，自分を監視するプロセスが存在しないという状況になる．そこで，自分を監視しているプロセスが故障したら，先と同様にして新たに自分の監視を依頼して監視体系を修復する（ここで暗黙のうちに自分の監視プロセスの故障を知ることができることを仮定においているが，これはそのプロセスの監視プロセスが次節で述べるように故障情報を伝達してくれるためである）．これにより，定常状態では各プロセスはつねに k 個のプロ

At $\forall p \in P$

Init :

$S_i = S_m = S_h = \phi$

$\{q_1, ..., q_k\} \in_R neighbors(p)$

for $i = 0$ to k:

send $monitor_req(p)$ to q_i

add q_i to S_i

Recv $monitor_req(q)$ **:**

send $monitor_ack(p)$ to q

add q to S_m

Recv $monitor_ack(q)$ **:**

remove q from S_i

add q to S_h

Every T_{hb} **:**

send $heartbeat(p)$ to $\forall q \in S_h, S_i$

Recv $failure(p', q)$ **:**

T_{to} **has passed since** p **sent** $monitor_req(q)$ **where** $q \in S_i$ **:**

$T_{hb} + T_{to}$ **has passed since** p **received the last** $heartbeat(q)$ **where** $\exists q \in S_m$ **:**

if q is not seen as $failed$:

see q as $failed$

send $failure(p, q)$ to $\forall q' \in S_m | S_h$

if $q \in S_i$:

remove q from S_i

send $monitor_req(p)$ to q'

where $q' \in_R neighbors(p)$ and $q' \notin S_h, S_i$

if $q \in S_m$:

remove q from S_m

if $q \in S_h$:

remove q from S_h

send $monitor_req(p)$ to q'

where $q' \in_R neighbors(p)$ and $q' \notin S_h, S_i$

図 1 故障検知機構の基本アルゴリズム（P：全体のプロセス集合，S_i：監視を依頼中のプロセス集合，S_m：自分が監視しているプロセス集合，S_h：自分の監視プロセス集合，neighbors(p)：正常な p の隣接プロセス，k：監視の依頼数（$k \leq$ neighbors(p)），T_{hb}：heartbeat 間隔，T_{to}：timeout 時間）

Fig. 1 Fundamental algorithm of failure detection.

セスから監視されているという条件が満たされることになる．この条件をつねに満たすように各プロセスが自律的に動くというのが，本アルゴリズムの本質的な動作となる．

4.2 故障情報伝達

一貫性の要求から，故障情報は漏れなく全体に伝達される必要がある．

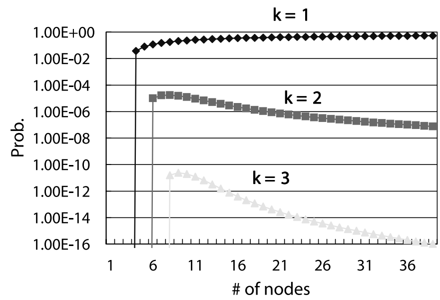

図 2 完全グラフにおける監視ネットワークの非連結確率

Fig. 2 Disconnectivity of monitoring network in complete graph.

全プロセスどうしが相互に接続しているアプリケーションにおいては，故障を検知したプロセスが全接続に対して故障情報を流すことによって容易に目的を達成できるが，一般的にはプロセスどうしが必ずしも直接接続されているとは限らない．そのため，故障を検知したプロセスから直接故障情報を届けられないプロセスに対して，他プロセスがその情報を中継する必要がある．これは flooding プロトコルを用いることによって実現することができるが，発生するメッセージ数は最悪の場合 $O(n^2)$ となり効率的とはいえない．一方，MPI で用いられているようなツリー状に送信するブロードキャストは，故障の発生下ではツリーが破壊されるため利用することができない．

我々は，故障情報の伝達に，前節で動的に生成した監視ネットワークを利用する．各プロセスが隣接プロセスと確立する監視関係をエッジと見なすと，その監視体系は 1 つのネットワークとして見ることができる．もしこの監視ネットワークが連結ならば，その上に故障情報を flooding しても全体に伝達されるはずである．その場合，メッセージ数はエッジの数とほぼ等しくなるので，$O(kn)$ に抑えることが可能となる．

そこで，監視ネットワークの連結性について評価を行う．ここでは簡単のため詳細は割愛するが，接続関係が完全グラフとなっている場合において各プロセスが非連結になる確率を計算した．**図 2** は，プロセスあたりの監視依頼数 k およびプロセス数 n を変化させたときの非連結確率の関係を表している．これを見ると，監視の依頼数が 2 つだけでも，非連結確率は 1.76×10^{-5}（$n = 8$）以下に抑えられていることが分かる．k が大きくなればなるほど非連結確率は減少していくわけだが，$k = 3$ の時点ではすでに 1.0×10^{-10} を下回っており，ほとんど 0 に等しいと考えてよい．つまり，各プロセスは自分の監視を少なくとも 2 プロセスに依頼すれば監視ネットワークは全体としてほぼ

```
At ∀p ∈ P
  Init :
    S_i = S_m = S_h = φ
    MS_n = neighbors(p) × k
    send monitor_req(p) to q ∈_R MS_n
    add q to S_i

  Recv monitor_req(q) :
    send monitor_ack(p, neighbors(p)) to q
    add q to S_m

  Recv monitor_ack(q, S_neigh) :
    remove q from S_i
    add q to S_h
    MS_n = MS_n − S_neigh − q × k
    if MS_n ≠ φ:
      send monitor_req(p) to q' ∈_R MS_n
      add q' to S_i
```

図 3 監視の依頼アルゴリズムの修正（S：セット，MS：マルチセット）
Fig. 3 Improvement of the algorithm.

確実に連結関係になり，故障による破損を考慮しても $k \geq 3$ に設定しておけば十分であるといえる．

ただし以上の議論は，クラスタ内のように各プロセスが直接接続できる場合に限り通用するものである．だが，一般的な環境では，クラスタ間をまたぐ接続はゲートウェイ間に限られていることが多く，完全グラフにはなりえない．このとき前節の監視プロセスをランダムに選択するというアルゴリズムをそのまま適用した場合，各ゲートウェイプロセスは自分の監視をすべて所属するクラスタ内に依頼してしまう可能性が高い．するとクラスタ間を結ぶ接続は互いに監視されないことになり，監視ネットワークが分断されてしまう．またこれは，ゲートウェイプロセスが故障してしまうと監視ネットワークの利用に限らず他クラスタのプロセスが故障を検知できないという別の問題もはらんでいる．

そこで，我々は先の基本アルゴリズムにおいて，監視の依頼部分に**図 3** のように修正を加える．

(1) 各プロセスは，マルチセット MS_n に隣接プロセスを k 個ずつセットする．
(2) この MS_n の中からランダムに選び監視を依頼するわけだが，その際に ack として依頼先の隣接プロセス集合の情報を送信してもらい，MS_n から引く．またこのとき監視の依頼先も MS_n の中から完全に消去しておく．
(3) もし MS_n が空でなかったら，再びその中からランダムに選んだプロセスに対して監視を依頼する．
(4) この過程を MS_n が空になるまで繰り返す．

このアルゴリズムを実行すると，最終的には自分のすべての隣接プロセスが，自分の監視プロセスかもしくは監視プロセスと少なくとも k 個隣接しているかのいずれかの状態に収束する．つまり，先の例でいえば，ゲートウェイプロセス間にも必ず監視の依頼が交わされることを示している．現実の計算機環境はほとんどの場合，全プロセス間で接続可能な部分完全グラフ（クリーク）がいくつも重ねあわさったものとして考えることができる．その意味でアルゴリズムを解釈し直すと，各クリークに k 本の監視のエッジを張るように動作するものであるといえる．よって，各クリークにおいて先に述べた完全グラフでの連結性の議論が成り立ち，k を適当な値に設定することで，監視ネットワークの連結確率を非常に高く保証することができる．

4.3 検 証

我々の手法は，各プロセスが等価的に動作するため，どのプロセスが故障しても同じように対処でき，監視プロセスがともに全滅しなければ故障を検知できるので，少なくとも k 以内の同時故障に耐えられる．監視ネットワークの断裂により故障情報が行き渡らない恐れがあるが，$k \geq 3$ に設定しておけばそれもほぼ無視できると考えられ，k を調節することにより非常に高い耐故障性を得られる．また，新規参入プロセスは監視の依頼を行うことで自動的に監視体系に加わることができるので，プロセスの動的増加にも対応できる．

一貫性は，監視ネットワークが連結している限り全プロセスに故障情報が伝達されるため満たされる．先にも述べたように，$k \geq 3$ に設定しておけば監視ネットワークが断裂する可能性は非常に低く抑えられるため，実用的に要件を満たしているといえる．

我々の故障検知機構では，通常時に発生する heartbeat 数は 1 プロセスあたり k 個である．つまりプロセス数が増加しても各プロセスにかかる負荷はほぼ一定になる．また，Gossip 型のように timeout を大きく設定する必要もないため，プロセス数によらず低負荷で迅速な故障検知が可能となる．

以上より我々の提案手法は，3.1 節の自律性・一貫性・効率性という要求を満たしていると考えられる．

3.2 節で 3 要求に対する特徴について既述した従来手法と FT-MPI・MPI/FT で用いられている故障検知手法（All-to-All，Server-Client）も加えて整理した表を**表 1** に示す．各プロセスが対称的で独立して動作する All-to-All や Gossip 型は自律性が高い一方，

表 1　各故障検知手法の特徴
Table 1　Features of failure detection protocols (All: All-to-All, S-C: Server-Client, Hier.: Hierarchical-style, Gossip: Gossip-style, Ours: Our approach).

Requirements	All	S-C	Hier.	Gossip	Ours
Autonomy	○	×	×	○	○
Consistency	×	○	○	×	○
Efficiency	×	×	○	△	○

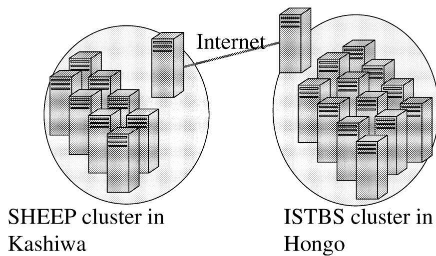

図 4　実験環境
Fig. 4　Experimental environment.

故障判断をそれぞれが独自に行うため一貫性は保たれない．それに対し，特定のプロセスに故障判断をさせる集中管理型の手法は，一貫性を保ちやすいものの，そのプロセスの耐故障性の問題やトポロジの手動設定が必要となり自律性が低い．我々の手法では，各プロセスの対称性・独立性を保ちつつ，監視プロセスが判断した故障情報を全体に反映させているため，自律性と一貫性を両立させることが可能になっている．

効率性は，heartbeat の負荷を分散している手法がスケーラビリティも高く優れている．しかし，Gossip 型に関しては 3.2 節で述べたように若干難点がある．

5. 実験・評価

提案手法に基づいた故障検知機構を実装し，構築時や故障発生時における連結性や故障検知の評価実験および効率性における比較実験を行った．実験環境には，東京大学本郷キャンパスにある 112 台のクラスタ（Xeon 2.4 GHz × 2CPU × 70 + Xeon 2.8 GHz × 2CPU × 42）と同柏キャンパスにある 65 台のクラスタ（Xeon 2.4 GHz × 2CPU × 65）を用いた．トポロジは**図 4** のようになっており，両クラスタはゲートウェイノード同士のみが接続できるものとする．

5.1 連結性の評価

本研究の手法において，監視体系の連結性は非常に重要な意味を持つ．万一連結性が損なわれると，故障検知機構が正常に機能せず，自律性・一貫性に支障が出るためである．そこで，システム構築時や故障発生時における監視体系の連結性について実験を行った．

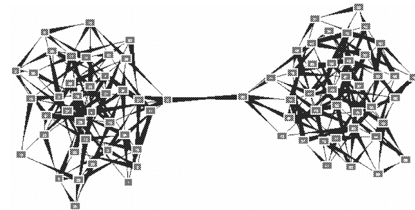
図 5　複数クラスタにおいて自律的に構築された監視体系
Fig. 5　Monitoring network autonomously constructed on multi-clusters.

本研究の手法に基づいて実装した故障検知機構を，$k = 3$ に設定して両クラスタでそれぞれ 40 プロセスずつ動作させたところ，**図 5** のような監視体系が構築された．図において，長方形はプロセス，それを結ぶエッジは監視関係（太い側のプロセスが細い側のプロセスを監視）を表しており，各プロセスに対して 3 本の監視が張られていることが見てとれる．また，クラスタ間を結ぶ接続間にも監視関係が築かれており，複数クラスタ環境でも連結性が維持されていることが確認できる．

ランダムに監視元を依頼するだけのアルゴリズムで動作させたところ，30 回の試行中 3 回しかクラスタ間で監視が張られることがなく，監視ネットワークが分離してしまった．それに対し，隣接プロセス情報を用いて連結性を考慮した手法では，必ずゲートウェイプロセスが互いに監視関係を築き，全体の連結性が保たれた．したがって，隣接プロセス情報を利用した手法を用いることにより，複数クラスタ環境のように接続が制限されている場合でも，手動でトポロジ情報を設定せず故障検知機構を自律的に構築することが可能になっている．

次に，本研究の手法に基づいて監視体系のグラフを生成するプログラムを作成し，ランダムにプロセスを F 個選んで同時に故障させたときの連結性についてシミュレーションを行った．プロセス数，監視依頼数 k を変化させて，各事例につき約 3.6×10^9 回の試行を行い非連結確率を求めたグラフが**図 6** となる．なお，グラフにおいてデータが存在していない部分は，3.6×10^9 回の試行のうち一度も非連結にならなかったことを表している．

いずれもプロセス数が増加するに従って非連結となる確率が下がっており，故障に強くなっていることが分かる．プロセス数が少ないときにやや非連結確率が高くなっているが，$k - 1$ 個以下の同時故障に関しては依然として無視できるほど小さく抑えられ，要求に応じて k を調節することで高い耐故障性が得られる

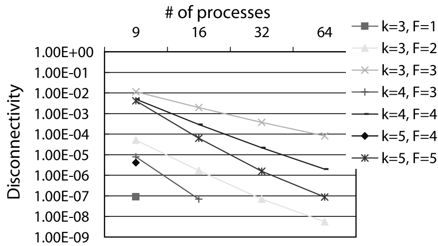

図 6　故障発生下での監視体系の非連結確率

Fig. 6　Disconnectivity of monitoring network with some failures.

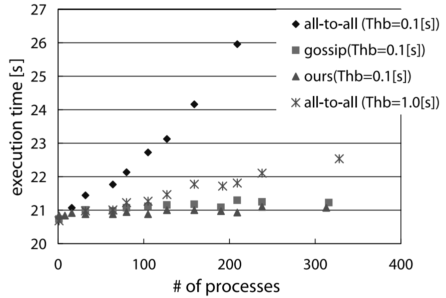

図 7　故障検知機構存在下のフィボナッチ実行時間

Fig. 7　Execution time of a Fibonacci program under failure detection services.

ことが確認できる．

本手法では，自分の監視プロセスの故障を検知すると，監視プロセスがつねに k 個存在するように新たに監視の依頼を行い，監視体系を自律的に修復する．そのため，故障検知に要する時間の間に k 個以上の複数故障が発生し監視ネットワークが分断しない限り，故障検知機構は正常に動作し続けることができる．また，実際にはプロセスの故障のほとんどが OS の返す通信エラーにより検知可能であり，自分の監視プロセスの故障を直接検知できるので，一時的に監視ネットワークが分断されたとしてもただちに連結状態になることが多い．したがって，実用的には非常に高い連結性が保たれ，自律性・一貫性を維持することができると考えられる．

5.2　効率性の評価

全対全で heartbeat を送信しあう手法（all-to-all），gossip プロトコルによる手法（gossip），我々の提案手法（ours($k=3$)）の 3 つの自律的故障検知手法について実装し，効率性に関する実験・評価を行った．各故障検知機構を別スレッドで動かしながらフィボナッチ（$n=45$，実行時間約 20 秒）の逐次プログラムを実行し，プロセス数や heartbeat 間隔 T_{hb} を変化させたときの実行時間を測定した．なお，測定にはつねに特定のノードを用い，測定するプログラムとは別にフィボナッチを走らせて SMP による効果を排除している．

結果を**図 7** に示す．まず all-to-all の結果を見ると，プロセス数が増加するに従って実行時間が長くなっていることが分かる．all-to-all は，各プロセスが T_{hb} ごとに全プロセスと送受信を行うため，$O(n)$ の負荷がかかる．それは T_{hb} が小さいほど大きくなるのだが，$T_{hb}=1$ [s] の時点ですでに無視できないオーバヘッドが発生しており，$n=159$ の時点で 5%を超えた．$T_{hb}=0.1$ [s] においては，$n=127$ の時点ですでに 10%にまで至り，もはや実用に耐えうるとはいいがたい．一方，gossip と ours はともに，$T_{hb}=0.1$ [s] に設定してもプロセス数によらずオーバヘッドは 2%程度に抑えられている．これは両者がスケーラビリティの点で優れていることを示している．

これを見ると，all-to-all と比較して gossip や我々の手法は効率性が高いように思える．確かに我々の手法においては，等しい T_{to} と T_{hb} を設定すれば，故障検知速度も正確性も等しくなるので，その議論は成り立つ．しかし，gossip においてはそのプロトコル上 T_{to} を長めに設定しておかなければならず，最低でも $T_{to}=30\times T_{hb}$ 程度は必要とする．また，$T_{hb}=0.1$ [s] においては，heartbeat の送信間隔が短くなるためプロトコルの同期的な動作が困難になり，逆に全体に伝わるまでにより多くのステップを要するようになった（$T_{to}=50\times T_{hb}$ でも誤検知する場合があった）．これを考えると，これ以上 heartbeat 間隔を短くすることは難しい．すると，gossip では故障検知までに，$T_{hb}=0.1$ [s]，$T_{to}=50\times T_{hb}$ としても，最短で 5.0 秒程度は要することになる．他の手法においては $T_{to}=1$ [s] 程度でも十分であったので，これは他の手法の $T_{hb}=4$ [s] のときとほぼ同じと考えてよい．$T_{hb}=4$ [s] とは all-to-all においてもオーバヘッドを小さく抑えられる範囲なので，gossip の方が効率性が高いとする先ほどの議論は成り立たないといえる．一方，ours と gossip を比較すると，ほぼ等しい負荷および誤検知が生じないパラメータ設定において ours の方が故障検知時間が短くなるため，我々の手法が gossip よりも効率性の点で優れていると考えられる．

したがって以上の議論より，我々の手法は従来の自律的故障検知手法よりも効率的であるといえる．

6. おわりに

本稿では，耐故障並列計算を支援する故障検知機構としての要件を整理し，それを満たした故障検知アルゴリズムを提案した．接続が一部制限されている環境

においても自律的に構築され，複数故障に対しても高い耐性を持っていることを示した．313 プロセスで動作させたときのオーバヘッドを測定してみたところ，heartbeat 間隔を 0.1 秒に設定した場合においてもたかだか 2%程度に抑えられており，他の自律的故障検知手法と比較しても効率性が高いことを示した．

今後の課題としてまずあげられるのは，ネットワークの分断への対応である．現時点では，ネットワークが分断されると，分断先のプロセスが到達不能になったことに気づかない恐れがあり，故障検知機構が正常に機能しない．

現在のアルゴリズムはプロセスの動的参入に対応することができるが，すでに参加していたプロセスは新たに監視の依頼を行わない．そのため，新規プロセスは他プロセスから監視を依頼されることがなく，プロセスの動的参入が頻繁に発生すると，プロセス間の不均衡が蓄積していき一部のプロセスへの負荷集中をもたらす可能性がある．したがって，環境の動的変化に応じて監視体系も柔軟に再構築する枠組みを備える必要があると考えられる．

参考文献

1) Fagg, G.E. and Dongarra, J.J.: Building and using a Fault Tolerant MPI implementation, *International Journal of High Performance Applications and Supercomputing*, Vol.18, No.3, pp.353–361 (2004).
2) Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P. and Magniette, F.: MPICH-V2: A Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging, *Proc. 2003 ACM/IEEE Conference on Supercomputing* (2003).
3) Batchu, R., Dandass, Y., Skjellum, A. and Beddhu, M.: MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware, *Cluster Computing*, Vol.7, No.4, pp.303–315 (2004).
4) Globus Project: MDS. http://www.globus.org/mds
5) Wolski, R., Spring, N.T. and Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, *Future Generation Computing Systems*, Vol.15, No.5–6, pp.757–768 (1999).
6) Chandra, T.D. and Toueg, S.: Unreliable failure detectors for reliable distributed systems, *J. ACM*, Vol.43, No.2, pp.225–267 (1996).
7) van Renesse, R., Minsky, Y. and Hayden, M.: A Gossip-Style Failure Detection Service, *Middleware '98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp.55–70 (1998).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 3 日採録)

堀田　勇樹（学生会員）

1981 年生．2004 年東京大学工学部電子情報工学科卒業．同年 4 月より東京大学大学院情報理工学系研究科電子情報学専攻修士課程在学中．

田浦健次朗（正会員）

1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）取得．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．

近山　　隆（正会員）

1953 年生．1977 年東京大学工学部計数工学科卒業．1982 年東京大学大学院情報工学専門課程修了，工学博士取得．同年より ICOT において第五世代コンピュータプロジェクトに参加．1995 年より東京大学に移り，現在同新領域創成科学研究科基盤情報学専攻教授．