2004－HPC－99　(39)
2004／8／1

Phoenix プログラミングモデルにおける故障検知ライブラリ

堀田勇樹[†]　田浦健次朗[†]　近山隆[†]

グリッド環境で長時間に渡るアプリケーションを実行するには、耐故障性を考慮した設計が重要となる。本稿では、我々が開発しているグリッド環境対応のメッセージパッシングライブラリである Phoenix に故障検知機能を追加し、ユーザが耐故障並列計算を記述する枠組みを提供した。そして実際にこの Phoenix ライブラリを用いて耐故障並列 N-Queen を作成し、従来の Phoenix ライブラリを用いた耐故障処理をしない並列 N-Queen と性能比較をしたところ、オーバヘッドはプロセス数によらず高々１％程度に抑えられ、スケーラビリティの高さを示した。

A Fault Detection Library in Phoenix Programming Model

Yuuki Horita,[†] Kenjiro Taura[†] and Takashi Chikayama[†]

In Grid environments, fault tolerance is one of the most important elements for the long-running application. We implement a fault detection library into Phoenix, our message passing library supporting Grid computations, and provide a framework for users to describe fault tolerant parallel computations. We show that the overhead of fault detection is at most one percent regardless of the number of the processes, and evaluate its high scalability.

1. はじめに

高性能大規模計算における資源環境は、ノード数的にも地理的にも拡大傾向にある。数百から数千台規模の計算もすでに稀ではなく、高性能のネットワークが整備されたことにより、WAN を超えた並列計算（i.e. グリッド）も現実的になってきている。このような環境では、ノード数の増加に伴い、以前に増してノードやプロセスのクラッシュによる影響を受ける確率が上昇するとともに、地理的な分散によりネットワークの切断やネットワークトポロジーの変化など、LAN 内ではあまり考慮する必要がなかった要素もアプリケーションの弊害となってきている。そのため、耐故障性に対する重要性はますます高まっており、不測の故障にも対処できる信頼性の高い (reliable) アプリケーション設計が不可欠となってくると考えられる。

Phoenix[7)6)] とは我々が開発しているグリッド環境対応のメッセージパッシングライブラリである。Rank と呼ばれる唯一のプロセス識別子を介してメッセージの送受信を行う MPI モデルとは異なり、Phoenix モデルではプロセスを仮想ノード番号と呼ばれる比較的大きな識別子集合に重複なくマップさせ、仮想ノード番号を用いて送受信を行う。プロセス数や識別子が固定されている MPI モデルでは、プロセス数の増減に対応するのは困難である。一方 Phoenix モデルでは、識別子がプロセス数に依存しないため、マッピングを変更するだけで容易にプロセス増減に対応することが可能である。またもう一つ特筆すべき特徴としてルーティング機能をサポートしており、直接コネクションを張れない環境においても、下層でメッセージをフォワードしてくれるため、ユーザはネットワークトポロジーを意識せずプログラムを記述することができる。しかし、一方で耐故障性の支援はまだ十分とは言えず、現在の Phoenix ライブラリを用いて reliable なアプリケーションを記述するのは現段階では困難である。

そこで本研究では、Phoenix モデルにおいて耐故障アプリケーションを記述する際に有用な枠組みの提供を目的とする。本稿では、その第一歩として実装した故障検知ライブラリについて述べる。従来の Phoenix ライブラリと並列化した N-Queen を用いて性能を比較してみたところ、高々１％程度のオーバヘッドに抑えられていることがわかった。

本稿の構成は以下のようになっている。第２節でまず Phoenix モデルについて触れ、第３節で故障検知機能を Phoenix に組み込む際の要件を整理する。第４節で今回実装した故障検知手法を述べ、第５節ではその API とそれを利用した耐故障モデルを示す。第６節で、実際に作成した N-Queen の並列プログラムを用いて従来の Phoenix ライブラリとの性能比較など評価を行い、第６節で関連研究について言及し、最

† 東京大学
University of Tokyo

後にまとめと今後の課題について述べる。

2. Phoenix モデル

グリッドのような信頼性の低い環境で、多数のノードを用いて長時間計算を続けていると、ノードやネットワークの障害による計算ノードの喪失は避けられない。その一方で、共有計算資源を利用することが多いため、他のユーザの要望によって計算ノードを一部解放する必要が生じたり、あるいは空いている計算資源を追加したいという要求が生じたりするのも、ごく自然なことである。しかし、一般的な MPI モデルでは、計算開始時にプロセス数・識別子を固定しそれを元にプロセス間通信を行うため、そういったプロセスの増減に対応するには困難を要する。そこで考案されたのが Phoenix モデルである。

Phoenix モデルでは、まず仮想ノード番号と呼ばれるプロセス数よりも十分に大きい識別子集合を用意する。計算に参加するプロセスでこの仮想ノード番号の集合を重複なく所持することによって、ユーザは仮想ノード番号を用いてプロセス間通信を行うことが可能になる (図 1)。これによりプロセス数は識別子に依存せず、仮想ノードのマッピングを変えるだけで容易にプロセスの脱退・参入に対応することができる。

ただし注意しなければならないのは、プロセスの脱退・参入による仮想ノードのマッピング変更の際、一時的にどのプロセスもマップされていない仮想ノードが存在するということである。この時にその仮想ノード番号宛てに通信を行おうとした場合、目的地が存在しないことになる。Phoenix ライブラリではこの際のメッセージの喪失を防ぐため、目的地が存在しないときはそのメッセージをローカルに保持しておき、いずれその仮想ノードにマップされたプロセスが現れた時に送信されることになっている。これによって、安全にプロセスのマッピングを変更することが可能となる。

また、Phoenix ライブラリはルーティング機能もサポートしている。NAT やファイアウォールの傘下に位置するプロセスに対して、外部から直接コネクションを張ることはできない。ところが、本来ならばゲートウェイに位置するプロセスを介することで、互いに通信することは可能である。Phoenix ライブラリでは、仮想ノード番号を対象としたルーティングテーブルを動的に作成しており、直接通信できないプロセスに対しても下層で他プロセスを経由して伝達される。そのため、ユーザはネットワークトポロジーを意識することなくプログラムを記述することができる。

こうして Phoenix モデルでは、従来ではあまり考慮されていなかった環境の動的変化に対する一種のソリューションを提供しているわけだが、一方で耐故障性に対する支援は十分とは言えない。現在のところ計算中に一部のプロセスがクラッシュしても、他のプロセスはそれに全く気づくことはない。プロセスの故障が他プロセスに直接影響しないことは、耐故障性を実現する上での重要な要素であるが、周りが異変に気づかない限りは故障に対する処理を行うこともできない。そこで、我々は Phoenix ライブラリに故障検知機能を追加し、ユーザが耐故障並列計算を記述するための枠組みを提供する。

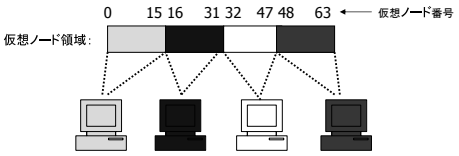

図 1 Phoenix model : 物理プロセスは仮想ノード集合にマップされる。この例では [0, 64) の仮想ノード集合に 4 プロセスがマップされている。例えば 36 番宛てにメッセージを送信すると、図の白の物理プロセスに伝達されることになる。

3. 要件の整理

ここで、Phoenix モデルにおいて故障検知機能を提供する際の要件について整理しておく。

3.1 故障検知手法の要件

故障検知手法として一般的に Heartbeat & Timeout メカニズムが用いられる。コネクション先のプロセスがクラッシュしたとき、ほとんどの場合は送受信を試みたときにエラーとして検出することができる。ところが、ネットワークの障害や相手がノードごとクラッシュした場合は、コネクションの切断に気づかないこともあり、運が悪いと永遠にやって来ないメッセージを待ち続けるということになる。それを防ぐために、定期的にメッセージを送信してもらいそれを timeout によって監視するという手法が用いられる。これは迅速に故障を検知するという効果もある。

しかし、このままではグリッド環境で適応するにあたって様々な問題を孕んでいる。[4] を参考にすると、グリッド環境における故障検知システムが満たすべき要件は以下の項目にまとめられると考えられる。

要件１ (メッセージ数の削減)

Heartbeat を全コネクションに対してそのまま適用すると、ノード数を n とすると 1 サイクルのメッセージ数は $O(n^2)$ となり、たちまちネットワーク上にパケットが氾濫してしまう。このネットワークの圧迫がアプリケーションの性能低下につながるので、メッセージ数をできる限り抑えることが重要である。

要件２（スケーラビリティ）

グリッド環境では大量の計算資源を用いることが多いため、効率の面でもオーバヘッドの面でもスケーラブルである必要がある。

要件３（低誤検知確率）

グリッド環境では、信頼性の低い WAN を経由するためにメッセージロスが発生する可能性が高く、またネットワークトポロジーも動的に変化するため、timeout が発生するまでにメッセージが届かないことが珍しくない。これが故障と判断されてしまうと、実際にはプロセスが正常であるにもかかわらず誤った情報がユーザ側に伝わり、耐故障アプリケーションは予期しない挙動を示す。耐故障性を支援する枠組みとして提供するためには、誤検知確率をできる限り小さくしなければならない。

要件４（耐故障性）

故障検知システム自体も、故障によって正しく機能しなくなる可能性がある。例えば、特定のノードが集中管理的に故障を監視していた場合、そのノードがクラッシュしてしまうと機能しなくなる。どのノード・プロセスが故障したときにも対処できるよう、全体として対称性を備えたシステムが望ましい。

要件５（プロセスの動的増減への対応）

アプリケーションによっては、プロセスが故障によって減少するだけでなく、意図的に計算から脱退したり新たに計算に加わったりする場合がある。そのように監視対象が動的に変化する状況にも対応できるようにする必要がある。

要件６（柔軟性）

ユーザのアプリケーションによって、故障検知に対する要求は異なると考えられる。従って、アプリケーションに応じてパラメータを変えられるような柔軟性も備えている必要がある。

3.2 Phoenix モデルの要求

通常は故障情報としてどのプロセスが故障したのかをユーザに提供してあげれば良いのに対し、Phoenix モデルでは仮想ノード番号を識別子としてプログラムを記述するために、どの仮想ノード番号が失われたかが重要になる。そのため、プロセスを監視するというよりも、仮想ノードのマッピング状況を監視することが要求される。

4. 故障検知手法

4.1 基本方針

Phoenix ライブラリは、ルーティングテーブルを動的に作成することによって、すでにある程度の故障検知能力を持っている。各プロセスは自分の周辺に変化があると自分のルーティング情報を更新し、DSDV という Flooding に良く似たアルゴリズムを効率化した手法を用いて全体に伝達する[5] ため、ノードやプロセスがクラッシュしてコネクションが切断されたことを検知できれば、その情報が直ちに全体のルーティングテーブルに反映される。しかも、このルーティングテーブルは仮想ノードを対象としたものであるため、到達可能かどうか見るだけでそのまま仮想ノードの異変に気づくことができる。

ところがこのままでは大きく２つの問題がある。まず、故障の発見は完全に送受信のエラーに依存してしまっているため、場合によっては故障を発見することができない恐れがある。２つ目に、ルーティング情報を見て到達不能だったからといって、必ずしもそれが故障によるものとは限らない。コネクションの切断がネットワークトポロジーの変化によるものだった場合、プロセス自体は正常であり、別の経路を見つけて再び到達可能になる可能性がある。さらに、動的なマッピング変更による一時的な仮想ノードの解放である恐れもある。これらは故障と判断してはならない事象であり、前節の要件３に反することになる。そこで、

- Heartbeat & Timeout メカニズムを組み込み、故障を確実に検知できるようにする
- ルーティングテーブルにおいてある一定時間以上到達不能の状態が続いたら故障と判断する

という手法を取る。以下の節ではそれぞれについてより詳しく述べていく。

4.2 Heartbeat & Timeout の組み込み

Phoenix のルーティング機構によって、プロセスの故障をどこかで検知すればそれを即座に全プロセスに伝達されることが保証されている。つまり、１プロセスさえ故障に気づけば良く、各プロセスは少なくとも１つ以上のプロセスに監視されていれば良い。

4.2.1 監視プロセスの設定

プロセスの監視は、特定のプロセスに任せるのが一般的で簡単な方法である。しかしこの場合、プロセス数の増加に伴い監視プロセスの負荷が高くなる上、監視プロセスの耐故障性は保証されていない (要件２、４の欠落)。また、監視体系を固定すると、環境の動的な変化にも適応できない (要件５の欠落)。

そこで、我々は各プロセスが自分を監視する隣接プロセス (直接コネクションを張れているプロセス) を指定するという方式を取る。具体的には、まず各プロセスはランダムに選んだ k 個の隣接プロセスに対して heartbeat を送信する。この際、次回送信するまでの時間 ($T_{interval}$) の情報をメッセージに含めておき、$T_{interval}$ 経過したら再び同じように heartbeat を

送信する。一方 heartbeat を受け取ったプロセスは、$T_{interval} + T_{timeout}$ 以内に次のメッセージが来るかどうかを監視する。もしもその時間内にメッセージがやってこなかったら、そのコネクションを破棄し、ルーティング情報を変更する。

この手法は、各プロセスは監視プロセスを自由に選択でき、監視プロセス数 k や heartbeat 間隔 ($T_{interval}$) などのパラメータも各プロセスごとに設定・変更できるので、環境の変化やアプリケーションに柔軟に対応することができる (要件５、６)。また、ランダムに監視プロセスを選ぶため負荷が分散され、プロセスあたりの監視数は総プロセス数に依存せず平均して k 個に抑えられる (要件２)。

4.2.2 耐故障性

しかし、このままではあるプロセスとその監視プロセスが同時に故障してしまうと、そのプロセスの故障を検知できない恐れがある。これを防ぐために、上記で選択されたコネクション以外でも、全て監視を行うようにする。ただし、この heartbeat 間隔 ($T_{insurance}$) は $T_{interval}$ に比べて非常に大きいものを想定している。なぜなら大半の故障は 4.2.1 で検知可能で、この監視は永遠に気づかないという最悪の事態を防ぐための言わば保険のようなものだからである。従って、これらの heartbeat は稀にしか送信されないので、メッセージ数としてはさほど影響を与えることはない。しかも、これによって任意の故障がいつかは必ず検知されることを保証することができる (要件４)。

4.2.3 全メッセージの利用

Phoenix ライブラリで作成したアプリケーションには、heartbeat 以外にも、ルーティングに関するメッセージやユーザによるメッセージが存在する。これらも本来であれば heartbeat の意味を持っているはずである。そこで、以下のように実装し、全てのメッセージを heartbeat として利用する。

- メッセージを送信したらそのコネクションの送信タイムスタンプを更新
- メッセージを受信したらそのコネクションの受信タイムスタンプを更新
- 送信タイムスタンプから $T_{interval}$(or $T_{insurance}$) 経過したら heartbeat 送信
- 受信タイムスタンプから $T_{interval}(orT_{insurance})+T_{timeout}$ 経過していたらコネクション破棄

これにより、無駄な heartbeat メッセージを省くことができる。

ここでメッセージ数について検証してみる。Phoenix ライブラリで発生するメッセージは大きく分けて、connect 関連、routing 関連、heartbeat 関連の３種類である。起動後しばらく経った環境として安定した状態を想定すると、connect 及び routing 関連のメッセージは発生しない。いずれも環境が変化した時にしか送信されないためである。heartbeat は、$T_{insurance} \gg T_{interval}$ とすると、単位時間当たりのメッセージ数はほぼ $(k*n)/T_{inteval}$ となり、k を小さくすることでメッセージ数を抑えることができる。実際 k は大きくても１桁台が実用的であると考えられるため、この主張は妥当である。そしてユーザメッセージの利用により実際のメッセージ数はさらに削減され、従って本手法は (要件１) を満たしていると言える。

4.3 Timeout による故障判断

ネットワークトポロジーの変化や動的なマッピング変更により一部の仮想ノードがルーティング情報において到達不能になるのは、ほとんどの場合一時的なものである。そのため、ある一定時間 (T_{broken}) 待っても到達不能の状態が続いた場合に故障と判断することで、それらと故障を切り分けることができる。

また、これは誤検知を防ぐ効果もある。メッセージロスなどによって heartbeat が時間通り到達せず、正常なプロセスを到達不能と判断してしまうことがある。そしてルーティング情報を更新し周りに伝達することになるのだが、そのプロセスと正常に通信を保っているプロセスが存在する限り、到達可能を示すルーティング情報が再び送られてくる。これにより、ルーティング情報において到達可能となり、この一連の挙動が T_{broken} 以内に完了すれば、誤って故障と判断することはない。すなわち、ルーティング情報を共通に保つことで全プロセスの合意を取る形になり、誤検知確率を下げていると言える (要件３)。

5. API と耐故障モデル

5.1 API

前節で述べたような故障検知機能をユーザが利用するために以下のような API を提供する。

- ph_msg_t ph_recv(int tag) : これは Phoenix ライブラリにおける受信関数で、tag が一致するメッセージを受信するとそのポインタを返す。これを故障が発生したら PH_MSG_ERROR が返るよう変更した。これにより、受信でブロックすることなく、エラーとして故障に気づくことができる。
- int ph_exist_broken_vps(ph_vps_t vps) : 仮想ノード集合 vps の中に故障が存在するかどうかを返す。
- ph_vps_t ph_get_broken_vps(ph_vps_t vps) : 仮想ノード集合 vps の中で故障している仮想ノード

集合を返す。存在しなかったら NULL を返す。

- void ph_set_*parameter*(double t) : 各種パラメータ (k, $T_{interval}$, $T_{insurance}$, $T_{timeout}$, T_{broken}) を変更する。

5.2 耐故障モデル

この API を利用してユーザがアプリケーションに耐故障性を持たせるには、次のようにエラーチェック・エラー処理の要領で耐故障処理を埋め込む形で行う。

```
/* メッセージ受信時 */
if((msg = ph_recv(tag)) == PH_MSG_ERROR){
  ph_vps_t broken_vps = ph_get_broken_vps();
  fault_handler(broken_vps); // 耐故障処理
}
...
/* 適宜故障チェック */
if(ph_exist_broken_vps(all_vps)){
  ph_vps_t broken_vps = ph_get_broken_vps();
  fault_handler(broken_vps); // 耐故障処理
}
```

基本的には、メッセージの受信時や、故障が発生すると無限ループに陥る恐れがあるような箇所に適宜耐故障処理を挿入する。これによって、いくつかのモデルのアプリケーションに対して、比較的容易に耐故障性を提供することができる。

5.2.1 Master-Worker モデル

Master-Worker モデルでは、他プロセスの故障情報を得られることで、容易に耐故障性を達成できる。Master はアプリケーションの状態を全て所持しているため、Worker が故障しても、割り当てていたタスクを復元することができる。すなわち、その失われたタスクを再び他の Worker に割り当て直すだけで良い。これだけでは Master の故障時に対応できないが、全ての情報を Master が管理していることから、Master が定期的に checkpoint を取っておくことによって、その問題も解決することができる。

5.2.2 Random-Stealing モデル

Master-Worker モデルよりもやや複雑になるが、Random-Stealing モデルでも同様に適応できる。Randam-Stealing とは、Master という固定プロセスからではなく、ランダムに選んだプロセスからタスクを分け与えてもらうという手法である。Master-Worker モデルでは、Master に全ての要求が集中するため Master の負荷が高くなってしまうのに対し、Random-Stealing モデルでは、要求が分散する上、タスクの再分配により上手く負荷分散され、効率的に計算を進めることができる。動的にタスク受け渡しの親子関係が生成される点で Master-Worker モデルとは異なるが、個々の関係で見ると子のタスクの情報を親が所持しているため、Master-Worker モデルと同じようにして耐故障性を実現することができる。

6. 実験と評価

作成したライブラリを評価するため、具体的に Random-Stealing モデルに従って作成した並列 N-Queen を実装し、性能測定などの実験を行った。実験環境としては、東京大学の本郷キャンパスにある 112 台 (CPU:Xeon2.4GHz×2×70+2.8GHz×2×42、RAM:2GB×112) からなるクラスタを用いた。なお、柏キャンパスにある 64 台のクラスタとつなぎグリッド環境でも動作することは確認してあるが、性能に他の要素が大きく影響し正しく評価できないため、本稿では示さない。

6.1 オーバヘッドの測定

表 1 故障検知ライブラリのパラメータ

k	$T_{interval}$	$T_{insurance}$	$T_{timeout}$	T_{broken}
2	5.0[s]	200.0[s]	5.0[s]	5.0[s]

従来の Phoenix ライブラリを用いた並列 N-Queen(Normal) と今回故障検知機能を加えた Phoenix ライブラリを用いて耐故障性を持たせた耐故障並列 N-Queen(FT) について、それぞれプロセス数を変えて計算時間を測定した。故障検知ライブラリのパラメータは (表 1) のように設定した。いずれも各ノードで必ず 2 プロセスずつ起動し SMP による効果を排除している。

表 2 N-Queen の性能比較

n	8	16	32	64	128	224
Problem	16	16	16	17	17	17
Normal[s]	252.9	126.9	67.5	280.7	147.5	89.2
FT[s]	255.7	128.3	65.87	283.5	148.5	89.4
pef.[%]	98.9	98.9	102.4	99.0	99.4	99.7

結果を (表 2) に示す。プロセス数によらずオーバヘッドが 1 %程度に抑えられていることがわかる。しかもプロセス数を増やすにつれて小さくなるので、我々の故障検知手法の負荷は小さくスケーラビリティも高いことを示している。

6.2 メッセージ数の評価

6.1 で用いた N-Queen を用いてメッセージについて調べた。パラメータには $T_{insurance}$ を 100[s] にした以外は同じ値を設定した。結果を (図 2) に示す。まず、ルーティングメッセージは開始直後に大量に発生するものの、計算中は全く送信されていないことが確認できる。また、1 プロセスの 5 [s] あたりの heartbeat 受信数も、理論どおり平均して 2 個前後に抑えられている。$T_{insurance}$ 経過後に残りのプロセスが heartbeat を送信する影響で一時的に増加するが、よく見ると本

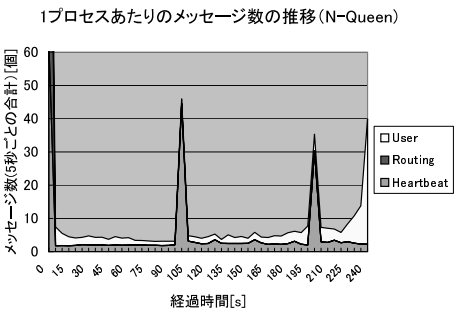

図 2 N-Queen のメッセージ分布

来送信されるべき heartbeat 数 (63 個) より少ないことがわかる。これはアプリケーションメッセージを利用することで、heartbeat 送信が先送りされたためであり、不必要な heartbeat を削減していることを示している。その結果として $T_{insurance}$ 経過以降若干増加は見られるものの、少ない heartbeat で全コネクションの監視を実現できていることがうかがえる。

7. 関 連 研 究

ライブラリレベルで耐故障性を支援する研究は数多く存在している。MPICH-V2[1] は、非同期な checkpoint 及び pessimistic message logging を行うことにより、ユーザ透過に耐故障性を実現する MPICH の拡張ライブラリである。大きなメッセージを用いて長時間実行するアプリケーションを対象としており、MPICH と比べ性能改善するなどの効果を挙げている。しかし、メッセージを頻繁に行うようなアプリケーションでは、messege logging のオーバヘッドが大きくなり、性能が著しく低下してしまうという難点がある。

MPI-FT[2] は、Master-Worker モデルに対する耐故障性を提供している。Observer と呼ばれる計算に参加しないプロセスを用意しておき、Master はそのプロセスに定期的に checkpoint しておく。Observer はプロセスの故障を検知すると、新しいプロセスを作成し故障プロセスの役割を引き継がせることによって、耐故障性を実現する。この手法は、余計にプロセスを必要とする上、checkpoint に大きなオーバヘッドが生じる。Observer が故障すると、耐故障性が失われてしまうという問題もある。

FT-MPI[3] では、他プロセスが故障すると通信関数を呼んだときにエラーを返すようにしたり、エラーハンドラを登録して復旧を半自動的に行うようにすることができる。考え方は本研究に近く、ユーザが耐故障アプリケーションを記述するための枠組みを提供するという方針を取っている。しかし、FT-MPI は TCP 接続を前提としており、グリッド環境における耐故障性の枠組みを提供しているとは言えない。

8. お わ り に

本稿では、Phoenix モデルにおける故障検知手法を提案し、Phoenix ライブラリに組み込んだ。並列化した N-Queen を用いて性能評価をしたところ、従来と比較してプロセス数によらず１%程度のオーバヘッドに抑えられていることがわかり、今回の故障検知手法がスケーラビリティに優れていることを示した。

しかし、現段階では耐故障性を実現できるアプリケーションは制限されてしまっている。本研究の耐故障モデルは、ユーザ透過に実現する耐故障モデルと比べてユーザの負担が大きい反面、オーバヘッドが小さいことに加え、ユーザが自らの要求に従って柔軟に耐故障性を実現できるという利点がある。この利点を生かすためにも、Phoenix モデルにおける耐故障性に対するさらなる支援の枠組みを提案・実装し対象とするアプリケーションを拡張するとともに、より信頼性の高いアプリケーション設計のモデルを模索していくつもりである。

参 考 文 献

1) Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. 2003.
2) Paraskevas Evripidou, Soulla Louca, and Neophytos Neophytou. Implicit Fault Tolerance : www2.cs.ucy.ac.cy/skevos/html/ft_for_mpi.html.
3) G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovsky, and J. Dongarra. Fault Tolerant Communication Library and Applications for High Performance Computing. October 2003.
4) Naohiro Hayashibara, Adel Cherif, and Takuya Katayama. Failure Detectors for Large-Scale Distributed Systems. In *SRDS*, 2002.
5) 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 2004)*.
6) Phoenix Project: www.logos.ic.i.u-tokyo.ac.jp/phoenix.
7) Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources(PPoPP2003). 2003.