# 分散ハッシュ表に基づく大規模探索問題の耐故障並列化手法

野 澤 康 文[†] 横 山 大 作[†] 近 山 隆[†]

大規模な分散環境で長時間の計算を行う際には耐故障性が必要である。耐故障性を有する既存の並列計算システムとしてサーバ・クライアント方式に基づくものが挙げられるが、それで効率的に計算できるアプリケーションは限られる。組合せ最適化問題やゲーム木探索といった探索問題を効率良く並列化するには、計算ノード同士が通信して共通部分を発見し、その結果を再利用することが必要である。本稿では共通部分問題の結果を再利用し、かつ耐故障を実現する枠組みとして分散ハッシュ表を用いた手法を提案する。本手法では故障時に問題を再実行することで耐故障を実現するが、その際に既に求められている部分問題の解を効率的に再利用して故障からの迅速な復旧を試みている。本手法を実装し、故意に故障を生じさせる実験を通じて耐故障性の検証を行なった。その結果、均等に負荷分散がなされる状況ならば耐故障処理のオーバーヘッドが小さいことが確認された。

# A Fault Tolerant Parallelization Framework based on Distributed Hash Table for Large-scale Search Problems

Yasubumi Nozawa,[†] Daisaku Yokoyama[†]
and Takashi Chikayama[†]

Fault-tolerance is necessary for widely distributed, long-running, parallel computation. The server-client framework has been commonly used to solve many practical search problems demanding a large amount of computing resources. This framework, however, only can treat embarrassingly parallel problems that can be easily divided into mutually independent subproblems. Problems such as combinatorial optimization and game tree search are more difficult. For efficiency, shared subproblems should be computed only once reusing the already computed results. In this paper, we propose a framework of parallel computation for such problems using a distributed hash table. We tried to verify the fault tolerance of this framework through experiments of inserting artificial faults to some nodes. As long as the load is balanced, the overhead of this framework is small.

## 1. はじめに

近年、計算機の普及やネットワークの発達に伴い地理的に分散した多数の計算資源を連携させるグリッド・コンピューティングが盛んに研究されている。実用面でもデスクトップグリッドなど、遊休計算資源を用いたプロジェクトが数多く存在し、一定の成果を上げてきている。そのような広域分散環境では計算機が故障する可能性を考慮に入れなければならず、とくに参加する計算機の数が多く、また計算にかかる時間が長いほど、その可能性は大きく無視できなくなってくる。

耐故障性を有する既存の並列計算のフレームワークとしてはクライアントをサーバが集中的に管理するというサーバ・クライアント方式が挙げられる。しかしサーバ・クライアント方式では、あらかじめ計算するべきタスクが固定で、そのタスクも互いに依存関係のない部分問題に分割することが可能な問題でないと効率的に並列化することは難しい。

並列化が単純ではない問題の例として組合せ最適化問題やゲーム木探索といった探索問題がある。このような問題は親問題が子問題を生成し子問題の結果を利用して親問題の解を求めるという木状の依存関係を持つモデルで記述できる。この木構造には互いに異なる親問題から共通の子問題を派生するという共通部分問題がある場合が多い。よって単純に問題を分割しただけだと 2 台以上の計算ノードで共通部分問題を重複して計算してしまう。そこで効率的な探索のためには計算ノード同士が通信して共通部分問題を発見し、その結果を再利用することが必要になってくる。これまでの研究でも最短経路探索問題や、ゲーム木探索など、共通部分問題が多い問題ではその結果を再利用する

† 東京大学
University of Tokyo

ことが並列計算の速度向上につながるという報告がなされている[10)9)6)]。それらの枠組みは計算に参加するノードは基本的に他の全ての参加ノードと通信する可能性がある。そのため故障に対応することが難しくなり、それらの枠組みに追従するシステムでの大規模な並列化はまだなされていない。これら共通部分問題のある問題で耐故障を実現することはグリッドの応用範囲の拡大につながり、有用であると言える。

これらの背景をふまえ、本研究では上記のような並列化が単純ではないような問題群に対し、共通部分問題の結果を再利用し、かつ耐故障性を持つ枠組みを提案する。提案手法はTDS(Transpositon Table Driven Work Scheduling)[10)] という分散ハッシュの考え方に基づくタスクのスケジューリング方法に耐故障性を付加したものである。本手法では故障時に問題を再実行することで耐故障を実現するが、その際に既に求められている部分問題の解を効率的に再利用して故障からの迅速な復旧を試みている。本手法を実装し、故意に故障を生じさせる実験を通じて耐故障性の検証を行なった。

## 2. 関連研究

本稿では $A^*$ アルゴリズムを用いた木探索やゲーム木探索など、タスクの分割を再帰的に行うアプリケーションを対象としている。ここではそれらを並列化する枠組みの関連研究について述べる。

### 2.1 サーバ・クライアント方式

計算機の参加や脱退、故障を扱える単純な枠組みとしてサーバ・クライアント方式がある。これは計算を行う多数のクライアントノードと、それらの状態を集中管理するサーバから構成される。クライアントはサーバにタスクを要求し、サーバは要求に応じてタスクを生成し、割り当てる。クライアントは割り当てられたタスクを計算した後その結果をサーバに返す。通信はサーバとクライアントとの間でのみ行われ、クライアント同士は互いにその存在を知ることはない。サーバ・クライアント方式に基づく並列計算のフレームワークとしては Charlotte[2)] や MW[5)] などが挙げられる。

この方式の利点としてはクライアントの故障に柔軟に対応できるという点が挙げられる。サーバはクライアントに割り当てたタスクを覚えておき、そのうち一定時間経過しても結果が返ってこないものについては別のクライアントに再割り当てする。このようなタスクの再実行により耐故障が実現される。また、故障と同様に計算機の参加・脱退に対応することも容易である。

問題点としてはサーバに負荷が集中することが挙げられる。サーバは非常に多数のクライアントの情報と、それらに分配したタスクの情報を管理しなければならない。また通信もサーバに集中する。サーバがボトルネックにならないためにはクライアントとサーバとの通信量を抑える必要がある。従ってクライアントに分配されるタスクは粒度が大きく設定される必要がある。

$A^*$ アルゴリズムなどの木探索では木の深さや幅が一定ではなく、部分木の計算コストをサーバが見積もることは難しい。よって粒度の大きいタスクをクライアントに割り当てる場合、クライアントどうしの負荷分散に問題が生じる。負荷分散を達成するにはタスクの粒度を小さくし、クライアント・サーバ間のタスクの要求、割り当てをこまめに行う必要がある。しかしそれは前述の通り、サーバの負荷の集中につながる。

また、サーバ・クライアント方式では木探索における共通部分問題を扱うことが難しい。各クライアントは、自らが計算しているタスク木については共通部分を発見できるが、他のクライアントが計算しているタスクとの共通部分は発見できない。そのため、この方式で共通部分問題が多いような問題群を扱うのは不十分であると言える。

### 2.2 work-stealing

サーバ・クライアント方式では難しかった負荷分散を達成するための手法として work-stealing が挙げられる。これは計算ノードどうしでタスクをやり取りすることによって負荷分散を達成するというものである。各ノードは自分のタスクキューにタスクがある場合、そのタスクをキューから取り出し展開する。タスクを展開して新しいタスクができた場合、キューに挿入する。深さ優先探索の場合はキューの頭からタスクを挿入する。キューにタスクがない場合は他のノードにタスクを要求し、要求を受けたノードはタスクが余っていればそのうち粒度の大きいものを返送する。

work-stealing による並列化のフレームワークとしては共有メモリ環境を対象とした Cilk、その後継で分散メモリ環境へ拡張した Cilk-NOW[3)]、Atlas[1)]、さらには Javelin[8)] とその後継の JICOS[4)]、その他には Satin[12)] などがある。これらの研究の多くで、branch-and-bound のアプリケーションで優れた台数効果が得られることが示されている。また、耐故障性を実現しているものもある。

work-stealing を用いた場合の耐故障は基本的にはサーバ・クライアント方式と同様、失われたタスクの再実行によってなされる。各計算ノードはどのタスク

がどのノードにスチールされたかを記憶しておく。故障が発生した場合、その故障ノードにスチールされたタスクを再実行することにより耐故障を実現する。このとき、故障ノードからスチールされたタスクについては、異なる 2 つのノードで重複して計算されてしまうという問題が生じる。そのため、サーバ・クライアント方式に比べて work-stealing では故障時のペナルティは大きい。そのペナルティを軽減する手法を提案しているものとして Satin[12] がある。

work-stealing は負荷分散の手法としては有効だが、サーバ・クライアント方式と同様に木探索における共通部分問題の重複計算の回避を扱うことが難しい。work-stealing を用いて $A^*$ のアプリケーションを並列化し優れた台数効果を示している先行研究がいくつかあるが、共通部分問題の重複計算の回避については言及されていない。共通部分問題が少ないアプリケーションについてはそれで問題ないが、グリッドの適用範囲を広げるためにもそれを扱うことは重要である。

### 2.3 Transposition table Driven Scheduling

Transposition table Driven Scheduling(TDS)[10] は並列計算におけるタスクのスケジューリングアルゴリズムの一つである。この手法を用いてアプリケーションを並列化することによって共通部分問題の重複探索を避けることができる。

#### 2.3.1 Transposition Table

TDS を用いるか否かにかかわらず、木探索における共通部分問題の重複計算を避けるためには、以前計算した部分問題の結果を保存しておくハッシュテーブルが用いられる。このテーブルのことを transposition table という。タスクにハッシュ関数を適用して得られた値をキーとしてこのテーブルを参照することで、既に計算したタスクならばその計算結果を即座に得ることができる。

並列計算の場合、計算に参加しているノード間でテーブルをどのように保持するかが問題となる。各計算ノードがローカルにテーブルを参照・更新するだけでは、他の計算ノードが計算しているタスクとの重複を発見することができない。また、各計算ノードがどのタスクのエントリを保持するかをあらかじめ決めておくいう戦略も考えられるが、その場合テーブルの参照・更新に通信や同期のコストがかかる。

#### 2.3.2 TDS のアルゴリズム

TDS はテーブルの参照・更新時のコストを抑え、かつ共通部分問題の重複計算を避けることができる手法として提案された。TDS のアルゴリズムの概要は以下の通りである。

まず、各計算ノードが担当するタスク空間を計算開始前に決めておく。例えば区間 $[a, b)$ のハッシュ値のタスクはノード $A$ が担当し、区間 $[b, c)$ のハッシュ値のタスクはノード $B$ が担当するといった具合にタスク空間を割り当てる。そして、各計算ノードはタスクを展開するたびにその子タスクのハッシュ値を計算し、タスク空間の割り当てに従って担当計算ノードに転送する。このようにすることで同じ部分問題は常に同じ計算ノードが担当することになるので共通部分問題の重複計算が回避される。

このアルゴリズムではタスクを展開して子タスクを生成するたびに通信が行われる。しかしテーブルの更新はすべてローカルに行われ、通信の必要はない。なぜなら TDS ではタスクを計算するノードと、その結果の保存先のノードは常に同じだからである。タスクの重複計算を避けられるアルゴリズムの中では、TDS は通信量が小さいと言える。

#### 2.3.3 TDS の評価

先行研究においてルービックキューブ、15puzzle などのアプリケーションで TDS の評価がなされている[10]。評価は LAN 内の 128 プロセスで行われ、work-stealing を用いた並列化に比べて 2 倍から 13.7 倍の速度向上が達成されている。また、TDS を WAN の環境に拡張した実験も行われている[9]。クラスタ間の通信量を抑えるために TDS はクラスタ内のノード間でのみ行われ、クラスタ間の負荷分散は work stealing によって行うという手法がとられた。この実験では 4 クラスタでの計算時間が 1 クラスタでの計算時間に比べて 2.1-3.4 倍という結果が得られている。また、同じ環境で work stealing のみによる並列化と比較され、TDS の方が有効であることが示されている。

## 3. 提 案 手 法

サーバ・クライアント方式、および work stealing だけでは共通部分問題の重複計算を回避できないために十分な性能が得られないアプリケーションが存在する。そして transposition table を使って共通部分問題の重複計算を回避できる並列探索アルゴリズムとして TDS があることを述べた。

TDS に耐故障性を付加することができれば、従来の手法で大規模な並列化を断念していたそれらのアプリケーションも扱えるようになることが期待される。そこで本稿では TDS に耐故障性を付加する手法を提案する。

### 3.1 基本的なアイデア

サーバ・クライアント方式でプロセスが故障した場合タスク木上で局所的に部分木が失われるのに対し、TDS の場合は失われるタスクは分散され、タスク木は小さな穴がたくさん空いたような状態になる (図 1)。その状態でルートのタスクを再展開する。タスクの再計算においては、transposition table を参照することで既に求められているタスクの結果を即座に返すことができる。木のルートに近い部分のタスクのテーブルエントリが参照され、結果を返すことができれば、そのタスクから派生する部分木のノードは再計算されない。そのため、故障ノードが計算していたタスクのうち実際に再計算されるタスクは少ないと期待される。

**図 1** 故障後のタスク木

### 3.2 耐故障処理の手順

提案手法による TDS の耐故障処理は以下の 3 段階で行われる

(1) 故障検知
(2) 故障したノードが担当していたタスク空間の再割り当て
(3) ルートタスクの再実行

故障を常に正しく検知することは容易ではないが、ここではそれが可能であるとする。false positive も false negative も起こらないと仮定する。

故障後に問題となるのは、故障したノードが担当していたタスク空間が別のノードによって引き継がれなければならないことである。常にタスクが正しい担当ノードに転送されるという条件を満たしながらノードとタスク空間のマッピングを変更することは容易ではない。本研究ではそれを Phoenix Library[11] という並列計算プラットフォームを用いることで実現している。

タスク空間の再割り当てが終了次第、再割り当てされたノードはそのことをルートタスクの担当プロセスに知らせる。その後ルートタスクが再実行される。タスク木は前述のアイデアに従い、部分解の再利用によって迅速に復旧される。

なお、本手法は故障したノードは完全に停止するという fail-stop failure モデルに従うことと、通信でメッセージが失われないことを仮定する。

## 4. 実　　装

ルービックキューブをアプリケーションとして、提案する枠組みの実装を行った。実装したプログラムは並列計算プラットフォームである Phoenix Library[11] を利用している。

### 4.1 アプリケーション

アプリケーションとしてはルービックキューブを用いている。ゴール状態までの最短経路を反復深化 $A^*$ を用いて求めるものである。$A^*$ におけるヒューリスティック関数としては先行研究に倣ってパターンデータベース[7] を用いている。

transposition table にはタスク、探索の残り深さ、計算結果が保存される。探索の残り深さとは、反復深化 $A^*$ における探索の深さの境界と現在までの探索の深さとの差のことである。各ノードはタスクを計算する前にテーブルを参照し、該当するエントリを探す。エントリが見つかった場合、保存されている残り深さと現在の残り深さを比較し、前者が後者以上の場合、エントリに保存されている計算結果を再利用する。

TDS を実装するにあたり残り深さが小さいタスクについては、担当ノードに転送して既知の結果の再利用を期待するよりも自らローカルに計算するほうが速くなる。よって実装の際はある一定以下の残り深さのときはタスクを転送しないようにしている。その残り深さのことをここではリーフの粒度と呼ぶ。最適なリーフの粒度の設定はネットワークの速度やアプリケーションの共通部分問題の多さに依存する。リーフの粒度が大きすぎるとタスクの数が不足して負荷分散が十分になされない。また、ローカルに探索する部分木が大きくなってしまうので、他のノードが計算している部分との共通部分を発見する機会が少なくなってしまう。逆にタスクの粒度が小さすぎると通信量が多くなってしまう。実装の評価に際しては最適なリーフの粒度を実験によって求め、それを用いることにした。

各マシンは同時に実行可能なタスクが複数ある場合、深さ優先に (つまり残り深さの少なさを優先して) タスクを選択する。これは並列化された探索が全体として深さ優先に従うようにするためである。また、同じ深さのタスクが複数ある場合は先に到達したタスクが優先して展開される。

今回のアプリケーションでは、最適解が求められるまでの時間ではなく、反復深化 $A^*$ においてある深さ

までの探索がすべて終了するまでの時間を測定している。そうした理由は、最適解が求められるまでの時間は試行によってばらつきがあり、分散が十分小さくなるまで測定するのは難しいからである。

### 4.2 耐故障処理

実装に際して、計算ノードと担当タスク空間の対応関係を故障検知後に変更することが必要であった。それを扱うために、今回のプログラムは Phoenix Library の仮想ノード空間という機能を用いている。

Phoenix Library は基本的には並列計算モデルであるメッセージパッシングモデルを拡張したものである。メッセージパッシングモデルではメッセージの宛先として、IP アドレスなどのホスト名の代わりに整数値で表されるノード名を用いる。Phoenix が通常のメッセージパッシングと違うところは各ホストにノード名の集合が割り当てられることである。ノード名の集合のことを仮想ノード空間と呼ぶ。このようにノード名の集合を割り当てるという考え方は、計算途中に動的に資源を参加・脱退させる目的で生まれた。全体の仮想ノード空間は、区間 [0, L)(L は非常に大きな数) の整数で表され、各ノードにはその部分集合が割り当てられる。参加・脱退は仮想ノードをやり取りすることでなされるので、ノード名として指定できる空間は参加・脱退があっても変化しない。これによって資源の参加・脱退ができるアプリケーションのプログラミングを容易にしている。

物理ノードと担当タスク空間の対応関係は Phoenix の仮想ノード空間を用いて以下のように説明できる (図 2)。タスクはハッシュ関数によってハッシュ値に変換される。さらにそのハッシュ値に簡単な関数を施して Phoenix の仮想ノード空間にマッピングする。その仮想ノード空間に対応する物理ノードが、元のタスクの担当ノードとなる。つまりタスクの送信の際は、タスクのハッシュ値を計算、さらに仮想ノード名を計算し、それを宛先とする。物理ノード同士で均等に負荷分散がなされるために、仮想ノード空間は物理ノードの性能に比例した大きさに配分されることが望ましい。ハッシュ値空間から仮想ノード空間へのマッピングも負荷分散のため、均等になるようにする。

故障は Phoenix ランタイムライブラリによって検知される。故障検知は Heartbeat モニタリングによって行われる。これは監視される側が監視する側に対して生存を示すメッセージを送り続け、一定時間そのメッセージが監視する側に届かない場合に故障と見なすというものである。

故障により失われたタスク空間は、別のノードに

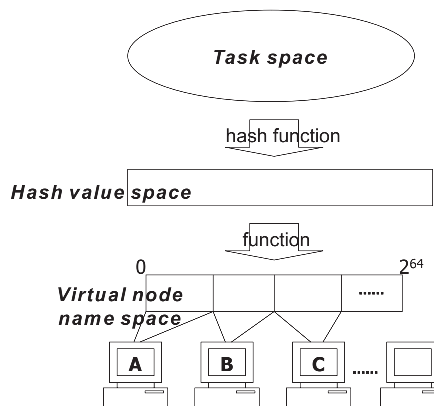


**図 2** タスク空間の割り当て

よって引き継がれる。タスク空間の割り当ての変更は Phoenix ランタイムによって全ての参加ノードに伝えられる。空間を引き継いだノードはルートタスクを持つノードに再計算要求を送信する。要求を受け取ったノードはルートタスクを再実行する。

## 5. 評　　価

実験は CPU Xeon 2.40GHz dual、メモリ 2GB のノード 65 台が Gbit イーサのネットワークで接続されている PC クラスタで行った。

### 5.1 リーフの粒度

まずは今回の実験環境に最も適したリーフの粒度を測定した。例えばリーフの粒度が 12 で反復深化 $A^*$ の探索の境界の値が 17 ならば、初期状態から深さ 5 までは TDS のスケジューリングに従ってタスクがノード間で転送され、それ以降は深さ 17 までローカルに探索が行われることを意味する。

ルービックキューブでゴールまでの深さが 18 以上である問題について、反復深化 $A^*$ のイテレーションが 17 の探索が終了するまでの時間を測定した。表 1 がその結果である。リーフの粒度が 11 の時がもっとも速く終了した。よって以降の評価でリーフの粒度は 11 に設定している。

**表 1** タスクの粒度ごとの探索終了時間

| リーフの粒度 | 探索終了時間 (sec) |
|---|---|
| 10 | 318.594 |
| 11 | 250.075 |
| 12 | 264.313 |
| 13 | 303.546 |
| 14 | 443.659 |
| 15 | 879.763 |

### 5.2 台数効果の評価

今回の実験は同じ性能のプロセッサによる並列化の

ため、均等な負荷分散のためにタスク空間は各プロセスで平等に割り当てた。そして解が深さ 18 以降に見つかるような 3 つの問題について、反復深化 $A^*$ のイテレーションが 17 の探索が終了するまでの時間を測定した。

図 3 に台数効果を示す。図 3 は Phoenix Library を使わないときの実装で 1 台で計算したときを基準とした台数効果である。3 つの問題とも 63 台でおよそ 48 倍の台数効果を示している。

図 4 は図 3 の問題 1 を 63 台で計算したときの負荷をとったものである。y 軸は 63 台がすべてタスクを計算しているときが 1 である。

なお、この図で負荷の部分の面積は、全面積の 77.6% であった。48/63=76.2%であるから、台数効果が出ていない原因はほぼ負荷分散で説明できる。残りの原因は通信や Phoenix Library ランタイムのオーバーヘッドであると予測できる。

負荷が分散されていないのは反復深化 $A^*$ のそれぞれの iteration の終盤である。20 秒経過後も負荷がないが、これは反復深化 $A^*$ の深さ 16 の iteration の終盤に該当する。そのフェイズではタスクの数が少なくなるため、アイドル状態になるノードが増えたと考えられる。

先行研究では TDS でルービックキューブを解くとほぼリニアな台数効果を示しているが、本研究の実装ではそれに及ばなかった。リニアな台数効果が得られなかった原因は負荷分散にあることがわかったが、均等な負荷分散ができなかった原因はリーフの粒度が大きかったからと考えている。今回の実装ではメッセージの生成、送受信などの処理のコストが大きかったためリーフの粒度を小さくしすぎると表 1 が示すように性能が失われた。そのコストがもっと小さければリーフの粒度をより小さくすることが可能となり、より多くの細かいタスクが生成されるため均等な負荷分散がなされたのではないかと考えている。

### 5.3 耐故障性の評価

耐故障性の実験は 63 台のノードで計算し、途中でランダムに選ばれた何台かを故障させ、それによって計算終了時間がどれだけ遅れるかを調べることで行った。問題は図 3 の問題 1 を用いた。これを 63 台で 10 回計算したときの平均終了時間はおよそ 250 秒であった。故障させる時間はその中間にあたる開始 125 秒後とした。また、故障させる台数は 1, 2, 4 台の 3 通りの場合について実験した。

故障したノードの担当タスク空間については (a) それまで計算に参加していなかった全く新しい計算ノードが引き継ぐ場合と、(b) 現在計算に参加している別のノードが引き継ぐ場合の 2 通りで実験を行った。

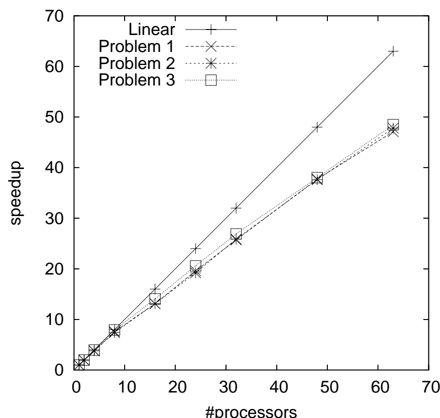



**図 3** 台数効果

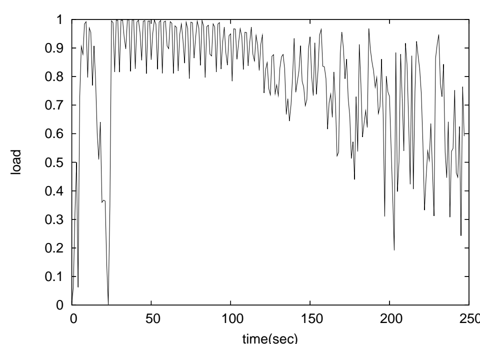



**図 4** 負荷分散 (故障なし)

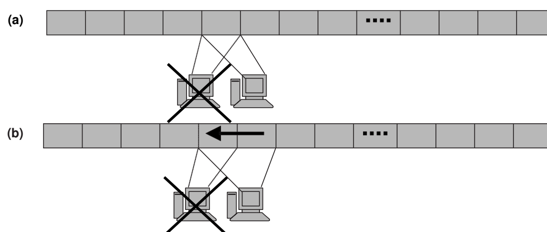



**図 5** 失われたタスク空間の引き継ぎ方

(b) については、故障した仮想ノード空間に隣接するノードが引き継ぐことにした。また、1 つのノードが 2 台分の仮想ノードを引き継がないようにするため、複数台を故障させる場合、隣接する 2 つの物理ノードは故障させないようにした。

故障する台数と (a), (b) のそれぞれの組合せについて 10 回ずつ実験を行った。

図 6 はそれぞれの故障の条件での計算終了時間の 10

回の平均値、最大値、最小値を示している。条件 (a) については故障がない場合に比べて終了時間にほとんど違いがないことが確認された。このことは、ルートタスクの再展開による再計算のオーバーヘッド自体は小さいことを示している。transposition table のデータの再利用が探索木のルートに近い部分で行われ、失われたタスクのうちリーフに近いものはほとんど再計算されなかったために、オーバーヘッドを小さく抑えられたと考えられる。

条件 (b) の 3 つのデータについては故障がない 250 秒に比べて大幅に遅れている。これは故障ノードのタスク空間を引き継いだノードがボトルネックとなるからである。

(a)(b) 共に 1, 2, 4 と故障する台数が多いほど終了時間が遅くなっている。故障が多いことはそれだけタスク木上のノードが多く失われることを意味する。失われるタスクが多いほど再計算されるべきタスクは多くなるのでオーバーヘッドは大きくなる。

(b) は特に故障の台数を増やすと計算が遅れる傾向が顕著である。(b) ではタスク空間を引き継いだノードがボトルネックとなり、そのノードに転送したタスクの結果待ちの親タスクが多くなる。そのため、(a) のように探索木のルート付近で transposition table のデータの再利用が行われる回数は少なくなる。よって失われたタスクのうちリーフに近いものが多く再計算され、遅くなったと考えられる。

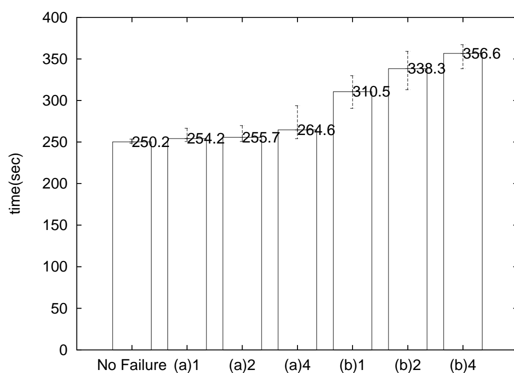



**図 6** 耐故障実験の結果

図 7 は 125 秒後に 4 台故障させたときの負荷分散の様子である。220 秒あたりから急に負荷が少なくなっていることが分かる。この時間になるとタスク空間を引き継いだ 4 つのノード以外はそれぞれ担当するタスクをほとんど計算し終わっていると考えられる。それに対してタスク空間を引き継いだノードは、基本的に従来の 2 倍のタスクを処理しなければならないのでボトルネックとなる。

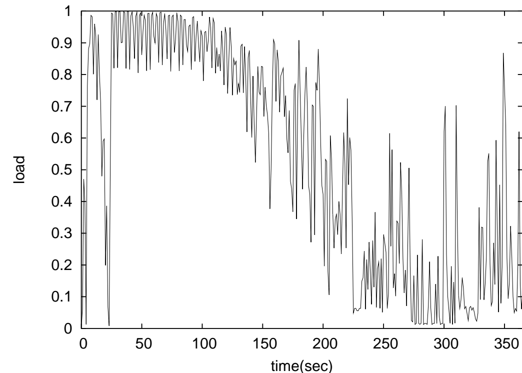



**図 7** 負荷分散 (条件 (b) で 4 台故障)

以上をまとめると、(a) の実験より、故障後も各計算ノード間で均等に負荷分散がなされるという理想的な状況ならば TDS で探索していても有効に耐故障処理がなされることが確認された。しかし (b) の実験が示す通り、故障したノードのタスク空間を別の 1 つのノードがそのまま引き継ぐというだけでは負荷分散ができず、結果的に効率が悪くなってしまう。そのことは動的負荷分散が必要であることを示している。

## 6. おわりに

本研究では共通部分問題の結果を再利用し、かつ耐故障および計算途中のノードの参加・脱退が可能な枠組みとして TDS を用いた手法を提案した。本手法は故障時に問題を再実行することで耐故障を実現し、その際に既に求められている部分問題の解を効率的に再利用して故障からの迅速な復旧を試みた。本手法を実装し、ルービックキューブをアプリケーションとして 63 台の PC クラスタで実験を行ったところ 48 倍の台数効果を得ることができた。また、故意に故障を生じさせる実験を行ない、本手法の耐故障性について検証を行なった。故障したノードの代わりに全く新しい計算ノードが即座に参加するという実験を通じて、そのような状況ならば耐故障処理のオーバーヘッドは極めて小さいということが確認できた。このことから、常に均等な負荷分散がなされるという理想的な状況ならば TDS と問題の再実行による耐故障は有効であることが示された。

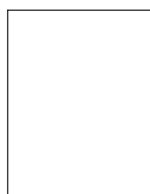
故障したノードの代わりに残った計算中のノードだけで耐故障処理をするという場合、負荷分散に課題が残った。今後の課題としては動的負荷分散を実現する

ことが挙げられる。

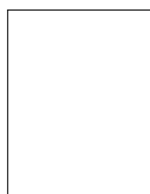
## 参考文献


1) J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing, 1996.
2) A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In Proc. of the 9th Int 'l Conf. on Parallel and Distributed Computing Systems (PDCS-96), 1996.
3) Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. pp. 133-147, 1997.
4) Peter Cappello and Christopher James Coakley. Jicos: A Java-Centric Networking Computing Service. In S.Q. Zheng, editor, Proc. 17th IASTED Int. Conf. Parallel and Distributed Computing and Systems, pp. 510 - 515, Nov. 2005.
5) Jean-Pierre Goux, Sanjeev Kulkarni, Jeff Linderoth, and Michael Yoder. An enabling framework for master-worker applications on the computational grid. In HPDC, pp. 43-50, 2000.
6) Akihiro Kishimoto. Transposition table driven scheduling for two-player games. Ms thesis, University of Alberta, Edmonton, Canada, 2002.
7) R. Korf. Finding optimal solutions to rubik 's cube using pattern databases. In Proceedings of the Workshop on Computer Games (W31) at IJCAI-97, pp. 21-26, Nagoya, Japan, 1997.
8) Michael O. Neary and Peter Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In Proc. ACM Java Grande/ISCOPE Conference, pp. 56 - 65, November 2002.
9) John W. Romein and Henri E. Bal. Wide-area transposition-driven scheduling. In HPDC, pp. 347-355, 2001.
10) John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. In AAAI/IAAI, pp. 725-731, 1999.
11) Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix : a parallel programming model for accommodating dynamically joining resources, 2003.
12) Gosia Wrzesinska, Rob V. van Nieuwport, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Fault-tolerant scheduling of fine-grained tasks in grid environments. Accepted for publication in International Journal of High Performance Applications, 2005.

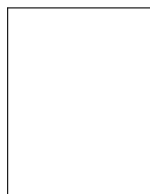

(平成 16 年 11 月 28 日受付)
(平成 17 年 2 月 4 日採録)



**野澤　康文**（学生会員）
.



**横山　大作**（正会員）
.



**近山　　隆**（正会員）
.