ハッシングに基づく大規模探索問題の耐故障分散処理手法

横山大作[†] 田浦健次朗[††] 近山隆[†]

多数の計算機を長時間用いる大規模分散計算を行うためには，一部の計算機の故障に際しても全体の計算が破綻なく動き続ける，という耐故障性を実現することが特に重要である．このような要件を満たしつつ，組合せ最適化問題などに代表される大規模探索問題を解くための分散計算フレームワークとして，マスタ・ワーカ構成，ワークスティーリングなどの手法が提案され，適用されてきた．探索問題は一般に，親問題の解が多数の子問題の探索結果によって決まる，という再帰的な構造で表現されるが，多数の親問題間で同一の子問題を共有し，ある子問題の結果が多くの親問題に必要とされるような種類の探索問題も多い．ゲーム木探索などがその代表例である．前述のマスタ・ワーカ構成などを用いると，同一の子問題を多数重複して無駄に計算してしまうことになりがちであり，探索効率に悪影響を及ぼしてしまう．このような重複計算をさけるための手法として，ハッシングに基づく分散探索手法が提案されている．我々は，失われた子問題を再実行することでこの手法に耐故障性を付加し，大規模探索に適用することを試みた．本論文では，重複する子問題を持つような探索問題の構造をモデル化し，マスタ・ワーカなどの手法と提案手法との，探索効率および故障発生時の回復に必要となるコストについて，シミュレーションによって比較を行い，提案手法の特質を明らかにする．また，実問題に対して本手法を適用し，実システムでの性能評価を行う．

A Hash-based Fault-tolerant Distributed Processing for Large-scale Search Problems

Daisaku Yokoyama,[†] Kenjiro Taura[††] and Takashi Chikayama[†]

For large-scale distributed processing of time-consuming computation with many computers, failures of some of the nodes should not cause failure of the whole computation. For this purpose, several fault-tolerant computation frameworks, such as master-worker and work-stealing methods, have been proposed and actually used. Search problems are generally defined recursively: A subproblem is solved using the results of its own child subproblems. Most practical search problems have subproblems shared as children of two or more subproblems. Game tree search is a typical example. With the master-worker framework, many subproblems are likely to be solved repeatedly leading to inefficiency. A search algorithm based on distributed hash table has been developed to eliminate such duplicated computation. Our proposal adds fault-tolerance to the algorithm through recomputation of subproblem results lost with faults. In this paper, a model of search problems with shared subproblems is formalized, and the proposed framework are compared with other methods such as one based on the master-worker framework in search efficiencies of cases with and without faults. The performance on real computers for some practical search problems is also shown.

1. はじめに

1.1 耐故障計算フレームワークの必要性

近年，マルチコア CPU の一般化，PC グリッドの普及など，並列計算環境はますます身近なものになってきているが，その環境を生かした並列プログラミングはそれほど普及しているとはいえない．これは，並列プログラミングの難しさに大きな原因がある．最も基礎的な手法で並列プログラムを書くためには，共有メモリ，メッセージパッシングなどの逐次実行とは異なる何らかのプログラミングモデルを新たに学習し，通信や同期といった並列計算特有の処理を多数加える必要があるが，これは面倒でありバグが入りやすく，かつデバッグも難しいものとなってしまう．並列計算環境の多様さもまたプログラミングを困難にする．

このような問題に対してはいくつかの方策がある．まず，並列計算向きの新しいプログラミング言語を用いるという方法があるが，これは一般のプログラマに

† 東京大学大学院新領域創成科学研究科
Graduate School of Frontier Sciences, The University of Tokyo
†† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

とって障壁が高い．そこで，従来の言語を使いつつ並列プログラムを簡単に書くために，並列化ライブラリや並列化フレームワークを用いるという手法がある．並列化フレームワークは，構造が単純な計算についてはよく研究され，成果をあげてきている．たとえば，科学技術計算においてよく用いられる規則性のある配列計算は比較的容易に並列化することができ，高性能化もしやすいため，OpenMP [1] などのフレームワークとして研究されてきた．しかし，より複雑な構造を持つ問題については，まだ研究が不十分な点も多い．

また，並列化フレームワークの基本的な機能は，データ分割，通信，同期など，並列アルゴリズム特有の処理を隠蔽することであるが，大量の計算機を長時間使って大規模な計算を行うためには，動的な計算機構成変更への対応や耐故障性といった機能が必要となる．同時に参加する計算機数が増えれば増えるほど，また計算にかかる時間が伸びれば伸びるほど，構成を変更する必要が生じたり故障が発生しやすくなったりする．なお，故障とは予告なく計算機構成が変更される場合，というとらえ方をすれば，耐故障性は参加計算機の動的再構成の極端な場合であると考えてもよい．

1.2 部分問題の計算結果を共有する問題

前述のように，規則的な配列計算などの構造が単純な並列計算については並列化フレームワークが整備されつつあるが，より複雑な問題領域においてはいまだ不十分である．そこで，本研究では，木状の依存関係を持ち部分問題を共有するような問題に対して，並列化フレームワークを提案する．

対象としている問題は，ある部分問題 t を解くために，その部分問題の子問題が再帰的に定義され，子問題の結果を使って t の結果を計算するようなものである．つまり，Divide-and-conquer によって計算できる構造を持つような問題である．構造を示す擬似コードを**図 1** に示す．Divide-and-conquer の構造を持つ計算は，子問題の計算を並列に計算することにより高速化できる．さらに，この子問題が，複数の異なる親問題を解くために必要とされる，すなわち部分問題を共有するような構造が存在するような問題領域がある．このとき，メモ化を行い，部分問題の重複計算を回避することで計算の効率を向上させることが可能である．このような問題の例としては，組合せ最適化問題やゲーム木などの探索問題があり，現実世界の問題として広い応用範囲を持つ．

図 2 は，15 パズルで部分問題の計算結果再利用を行ったときの探索効率の変化を示したグラフである．15 パズルを反復深化 A* [2] で解くプログラムを作り，

```
compute(Task t)
{
  for each (Child_Task c) {
    child_result = compute(c);
  }
  wait for all child_result;
  compute result_of_t
      from child_results;
  return result_of_t;
}
```

図 1 対象問題の構造を表す擬似コード
Fig. 1 Pseudo code of the target problem.

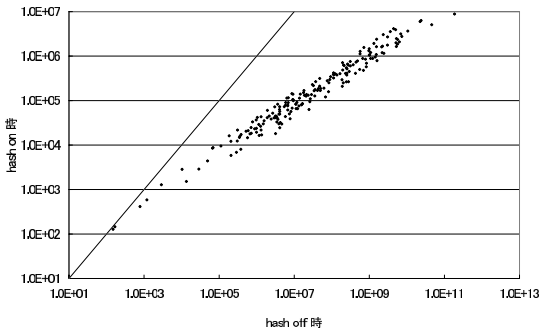

図 2 15 パズルでの部分問題共有の効果
Fig. 2 Effects of reusing partial results in the 15 puzzle problem.

ランダムに生成した様々な問題について，探索結果が出るまでに調べた盤面の数をプロットした．横軸は部分問題の計算結果再利用を行わなかったときの盤面数，縦軸は再利用を行ったときの盤面数を表している．部分問題の計算結果の再利用を行うことで，探索に必要な計算量が大きく削減されていることが見て取れる．特に，問題の規模が大きくなればなるほど必要な計算量の違いも大きくなり，大規模な問題を解くためには部分問題の計算結果再利用をしなければならないことが分かる．

一般に探索問題の計算量は，子問題の平均分枝数 b と探索深さ d を用いて，b^d と表される．部分問題の再利用を行うと，この平均分枝数 b が b' へと小さくなると期待できるため，問題規模が大きくなるにつれて大きく計算量が削減されることになる．

1.3 本論文の提案

従来研究において，Divide-and-conquer の構造を持つ問題領域の並列計算を行うためには，マスタ・ワーカ方式，Work-stealing 方式などが提案されてきた．こ

れらの手法は耐故障性も実現可能であるが，メモ化についてはまったく考慮していない．本論文の対象とする，メモ化が可能な問題領域に対して適用することは可能ではあるが，計算効率が大きく悪化してしまう．

また，メモ化について考慮した方式としては transposition table driven work scheduling [3] が提案されているが，これは計算機の参加・脱退などの構成変更や，耐故障性については考慮されていない．

そこで，我々は，Divide-and-conquer 型の構造を持ちメモ化が行えるような問題領域を対象とし，計算効率を落とさずに耐故障性を実現できるような分散計算方式として，分散ハッシュテーブルを用いた計算方式を提案する．

本論文では，メモ化が可能な問題において，従来手法の 1 つであるマスタ・ワーカ方式による耐故障分散計算ではどのように計算性能が悪化するのかをシミュレーションによって明らかにし，提案手法のシミュレーションとの比較で提案手法の有用性を示す．また，提案手法の耐故障性がどのような性能で実現できるのか，シミュレーションと試験実装で評価を行い，その有効性を示す．

2. 耐故障計算フレームワークの提案

2.1 フレームワーク概要

提案する耐故障性を持つ並列探索フレームワークの概要を以下に示す．なお，以降では並列計算に参加している計算機を「ノード」と呼ぶ．また，フレームワークで扱う，ある部分問題を計算し結果を求めるというひとかたまりの仕事を「タスク」と呼ぶ．

- タスクはある入力に対してある結果を出力する関数である．タスクは再帰的に子のタスクを生成し，その結果を用いてもとのタスクの結果を求める．
- タスクは入力に対して出力が一意に定まるような性質を持たなければならない．また，タスクの副作用が出力結果に影響を与えてはならない．
  今適用を考えている問題は，共通する部分問題を発見し，以前にその部分問題の計算結果が求められていたならば，その結果を再利用することで計算の効率を上げることが可能なものである．同一の入力に対しても計算のたびに結果が異なるようなタスクでは，再利用は不可能である．
- タスクは入力によって一意に定められる．入力からはハッシュ値を計算することができるものとし，そのハッシュ値に従ってタスクの計算を担当するノードが定まる．ノードは担当するタスクハッシュ値の範囲を持っており，ハッシュ値全体がすべてのノードによって，重複なく抜けなく分割されるようにシステムが動作する．
  これは，分散ハッシュテーブルと同様の考え方である．
- 動的に参加計算機を増減させる際には，ノードの担当範囲割当てを変更することで並列計算への新しい計算機の追加，あるいは脱退が可能になる．
  総タスク数がノード数に対して十分多く設定されていれば，ノードに含まれるタスクの量は担当範囲の広さに比例すると考えられる．よって，負荷の均衡を図って計算効率を上げるためには，担当範囲の広さをノードの計算力の割合に見合ったものにしなければならない．また同時に，担当範囲割当ての変更がなるべく局所的になるようにして，必要な通信量を抑えることも必要になると考えられる．現在の担当範囲割当て変更ポリシは，ランダムに担当範囲の分割を要求し，脱退時には隣の範囲を受け持つノードが範囲を併合する，というごく簡単なものであり，このような条件を満たしてはいない．今後の検討課題としたい．
- タスクは結果を必要としている親タスクを記憶している．タスクの計算結果が出たところで，タスクは自分の親タスクすべてに結果を返す．
- タスクは現在計算に必要な子タスクを追跡し続ける．子タスクが故障により失われたことが判明したとき，タスクが子タスクを再び生成，再実行する．
  子タスクが失われたことは，システムのハートビートモニタなど何らかの方法で検出する必要があるが，最も素朴に実現するには親タスクが子タスクに対して定期的にメッセージを送り続ければよい．
  また，すべてのタスクの親であるルートタスクは，生成に必要な情報とそのハッシュ値を計算に参加するすべてのノードが保持しておく．ルートが存在するはずの領域を担当しているノードにルートタスクがないとき，ルートタスクが失われたと考えて，そのノードがルートタスクを生成し，後は他の失われたタスクと同様に子タスクの再実行を行う．

また，対応する故障のモデルは，ある時点においてあるノードでの実行が停止してしまう，という場合を想定する．通信において最低限必要な要件は，ハッシュ値によって指定される宛先に対して，メッセージがいつかは届く，たまにメッセージが失われることもあるが再送すればそのうちに 1 つはメッセージが届く，という条件を満たしていればよい．

2.2 アルゴリズム概要

各ノードは計算途中のタスクの集合と，計算が終

わったタスクの結果の集合を保持している．タスクは実行可能状態と結果待ち状態の 2 状態をとる．ノード間では，親タスクが子タスクに送る query メッセージと，子タスクが親タスクに結果を送る answer メッセージの 2 種類のメッセージが通信される．

ノードは，実行可能タスクが存在する場合，それを 1 つ選び実行する．タスクは実行を終えると，子タスクの結果待ち状態か，タスク自身の結果が定まって終了するかのいずれかの状態になる．結果が定まった場合はそのタスクの結果を親タスクに返答し，計算結果をノード内に記憶された結果集合に追加して，タスクを消す．

query メッセージが到着したとき，結果集合にその答えが存在したら，すぐにその結果を返答する．結果がない場合，計算途中のタスク集合にそのタスクが含まれていないならば生成し，計算途中であったならばそのタスクの親タスクとして追加登録する．

answer メッセージが到着したとき，結果待ちタスクに対応するタスクが存在しないならば何もしない．これは故障発生時やメッセージの欠落などに起因するものであり，無視しても計算全体には影響しない．結果待ちタスクに対応するものがあったとき，タスクに子の結果を登録し，必要とする子の結果がそろったならば実行可能状態にする．

何らかの手段でノードの故障が検出できたとき，まず残ったノードで分担範囲の割当てをし直し，分散ハッシュテーブルに欠けた部分がないようにする．その後，各々のノードにおいて，計算中のタスクのなかで失われたと思われる範囲に存在する子タスクの結果を待っているものを探し，query メッセージを送って子タスクの結果が必要であることを伝える．

メッセージが欠落するかもしれない場合は，一定期間ごとに結果を待っている子タスクに対し query メッセージを送る．これにより，メッセージが欠落していた場合でも，あるいは子タスクが故障により失われていた場合でも，いつかは結果が得られるようになる．

2.3 フレームワークの API

このフレームワークを使うために，ユーザは

- User_key
- User_result
- User_task

の 3 つのデータ型を定義しなければならない．

User_key はタスクの入力を示すデータ型であり，15 パズルの盤面のように，タスクを一意に決めるものである．比較を行うための関数と，$[0, HASH\_MAX)$ の範囲のハッシュ値を返す関数をユーザは定義しなければならない．

User_result はタスクの出力を示すデータ型である．

User_task は User_key を使って User_result を計算するために必要なデータを保持するデータ型である．ユーザは計算の途中結果などを自由にここに保存しておくことができる．

これら 3 つのデータ型は，通信のためにシリアライズ/アンシリアライズ関数を定義しておく必要がある．

タスクの計算を行う際，フレームワークはユーザ定義のタスク計算用関数 do_partial_task に，User_key と User_task，およびタスクがどこまで計算を終えているかを示すシーケンシャルナンバを渡して呼び出す．シーケンシャルナンバは，ユーザ定義フレームワークが do_partial_task から帰ってくるたびに 1 ずつ増える数字であり，その User_task の計算がどこまで進んだかを示すものとして使用できる．ユーザは do_partial_task 内で，子タスクを生成してすべての結果がそろうまで待つ，どれか 1 つの子タスクの結果が得られるまで待つ，親タスクに結果を返す，のいずれかを行う．

また，ルートタスクの User_key を返す関数，ルートの結果が求められたときにフレームワークが呼び出す関数も定義しておく必要がある．

2.4 実用性を高めるための要素

A*アルゴリズムなどを実現する場合，タスクの User_key は同じだが，許容コストの上界など計算に必要なパラメータが異なるような計算を行いたくなる場合がある．このパラメータも User_key の一部だと考え，パラメータが異なれば違うタスクであるととらえることもできるが，パラメータを User_key とは別扱いすることで部分問題の結果を再利用することが可能な場合がある．このような問題を解くために，User_cond というユーザ定義のデータ型を用意し，do_partial_task 関数に渡すことができるようにした．この場合，タスクの計算結果は何らかの意味で単調にある終端状態へと変化していくものでなければならない．また，query メッセージには User_key と User_cond が含まれることになり，ある User_cond に対して現在得られている User_result が十分要求を満たしているかどうかを判断する関数をユーザが定義する必要がある．要求を満たしている場合はそのまま現在の User_result を用い，満たしていない場合は新しい User_cond でタスクを再実行することになる．

また，あらかじめ各ノードに 1 つずつコピーされるようなユーザ定義のデータ構造や，タスクのスケジューリングのヒントなど，利便性を向上させたり計

算効率を高めるために様々な部品を提供することが考えられる．これらは今後の検討課題としたい．

3. シミュレーションによる評価

実装を行う前に，今回提案する並列フレームワークの特性をシミュレーションによって明らかにすることを試みた．

3.1 シミュレーションのためのタスク生成モデル

本フレームワークが対象とする，部分問題を共有する構造の問題をシミュレートするためには，重複する部分問題に出会う様子をうまく表現できなければならない．そこで，ある問題領域でのタスク生成をモデル化し，その領域での共通部分問題出現の様子をパラメータを用いて表現できるようにすることを試みた．

図 3 は，15 パズルを解く A*探索プログラムにおいて，部分問題の記憶を用いることで平均分枝数がどれだけ削減されたか，を示している．様々な問題について，探索木の中で解までの深さが等しい部分問題群が生成した子問題が，記憶表によってどれだけ削減されたかを測定した．グラフの横軸は，子問題の解までの残り深さが示されている．グラフを見ると，ほぼすべての深さにおいて均一な割合で，重複した部分問題が生成されていることが分かる．

また，ある部分問題の子問題が共有されるのは，何らかの意味で近傍にある部分問題の間であることが多く，共有の様子には局所性があることも重要な事実である．15 パズルの例でいえば，似た局面の子問題が同一局面になるわけであり，大きくかけ離れた局面どうしの間では部分問題の共有は起きにくい．

このような振舞いを表現するため，タスク生成を以下のようにモデル化することにした．

タスクは (k, d) という 2 つの整数で定義される．k は $[0, MAX)$ の区間の整数，d はタスクの大きさを表す整数である．(k, d) は $(k_c, d-1)$ という子タスクを生む．k は d によってとりうる値が変化する．タスクはすべて m 個の子タスクを持つとする．ルートタスクの子タスクの k は，$[0, MAX)$ 上に均等にとられた m 個の点の値のみとることができる．つまり，i を整数として $k = MAX \times i/m$ という値のみとることができる．m は d が 1 減るごとに b 倍される．つまり大きさ d のタスクの m を m_d とすると，$m_{d-1} = bm_d$ である．この場合，ある大きさのタスク数の増え方は m の累乗根ではなく，およそ b の累乗個に抑えられるため，部分問題の共有が発生しつつ，全体としては平均分枝数 b の構造を持つタスクがシミュレートできる．さらに，子タスクの k_c は親タスクの k の近傍 m 個の値をとるものとする．k が近いタスクは「似ている」タスクに相当し，似ているタスクの子タスクは同一のタスクになる可能性が高くなると期待される．

図 4 は，このようなモデルでタスクを生成したときのタスクの大きさごとの子問題削減率を，図 3 と同様にグラフ化したものである．実際の 15 パズルの問題と同様の，ほぼ一定の子問題削減率になるようなタスクが生成されていることが分かる．

以上のようなモデルを用いてタスク生成をシミュレートし，提案手法の特性について検証を試みる．

3.2 マスタ・ワーカモデル

ここで，マスタ・ワーカ型の並列処理についても比較対象としてシミュレーションを行うことにする．マスタ・ワーカ型の並列処理は，単純な並列化手法であるが，負荷分散，スケーラビリティ，耐故障という，並列化フレームワークが持つべき特性を比較的簡単に実現することができる．耐故障性は，ワーカに送った仕事をマスタが覚えておき，ワーカが故障した場合にはそのタスクを別のワーカに送り直せばよい．

マスタ・ワーカによる並列化方式の重要な特徴として，ワーカどうしは通信を行わないという点がある．

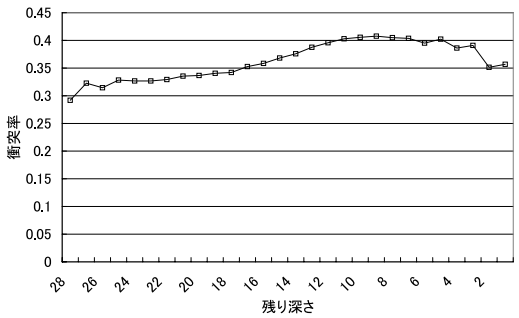

図 3 15 パズルでの子問題削減率
Fig. 3 Subproblem reduction ratio in the 15 puzzle problem.

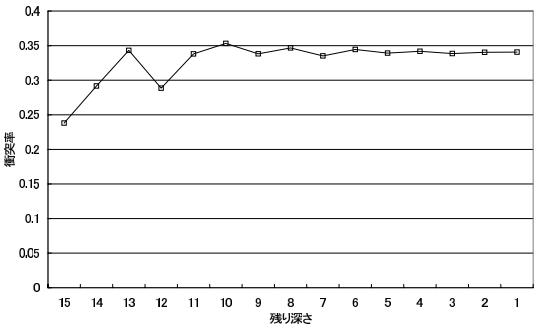

図 4 シミュレーションで用いたモデルの子問題削減率
Fig. 4 Subproblem reduction ratio in the problem of the simulation.

今回のフレームワークのターゲットである，部分問題が共有されているという問題領域は，ワーカどうしの通信がなされないと，同一の部分問題を複数のワーカで重複して計算してしまうという，無駄な計算コストが発生することが予想される．もちろん，マスタ内ではハッシュ表と同様の仕組みを用いて重複する部分問題を把握しているし，ワーカ内部についても同様であるが，ワーカ間にまたがって共有されている部分問題については無駄な計算は避けられない．これにより，どの程度計算効率が落ちるかが実用性をはかるうえで重要になる．

3.3 シミュレーションによる比較実験

3.3.1 シミュレーションの設定

以上のようなマスタ・ワーカ方式について，提案する分散ハッシュによる並列化手法と，同一のタスク生成モデルを用いてシミュレーションによって比較を試みた．

シミュレーションのパラメータは，タスクが子タスクを生成するのに 1 クロック，子タスクの結果がすべてそろった時点からタスクの結果を計算するのに 1 クロック，ノード間の通信に 1000 クロック，ノード内の通信に 1 クロック（すなわち，次のクロックには通信が届いている）と設定した．末端のタスクに関しては，途中のタスクとは異なり計算にある程度時間がかかるものとして，10 クロック必要な場合と 1000 クロック必要な場合の 2 通りについてシミュレーションを行った．

3.3.2 故障なしの場合

図 5 と**図 6** は，深さ 18 になるまで子タスクを生成し，そこで末端タスクを呼ぶような同一の問題について，故障が起きなかったときの台数効果を示したものである．図 5 が末端タスクの実行時間 10 クロック，図 6 が末端タスクの実行時間 1000 クロックの場合である．グラフ中，hash という系列が提案する分散ハッシュを用いた方式，mw から始まる系列がマスタ・ワーカ方式である．マスタ・ワーカ方式では，ルートのタスクからある深さまでの子タスクをマスタが計算し，その深さの子タスクを暇になったワーカに順次割り当て，ワーカがその深さ以降の子タスクの計算を行う，という計算方式をとるものとする．mw の後の数字は，ワーカに配られるタスクが末端タスクまで残り深さいくつのものであるかを示しており，これはすなわちワーカに送られる仕事の単位の大きさを表している．数字が大きいほどワーカに送られた仕事は大きいことになる．ワーカに送る仕事の単位が大きくなるほど少ない通信量で並列計算ができ，またワーカがマスタにタスクを要求する通信時間のオーバヘッドも少なくなるが，負荷の偏りが生じやすくなる．また，マスタ・ワーカ方式におけるノード数はワーカの数を示している．縦軸は，1 台の計算機で実行した場合の速度を 1 としたときの速度向上率を示している．mw，hash の 1 台での実行時間は等しい．図 5 のノード数が少ない領域において，mw が 1 以下の速度向上を示しているのは，問題がマスタとワーカ 1 台ずつに分割された時点で，分割される前（全体で 1 台のノードを使って計算した場合）より計算時間がかかるようになったことを示している．これは，ワーカがマスタにタスクを要求するときの通信遅延に起因している．

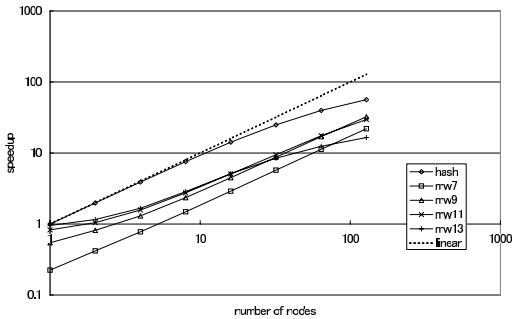

図 5 シミュレーションでの台数効果（問題サイズ 18，末端 10 クロック）

Fig. 5 Simulated speedups (problem size: 18, leaf size: 10).

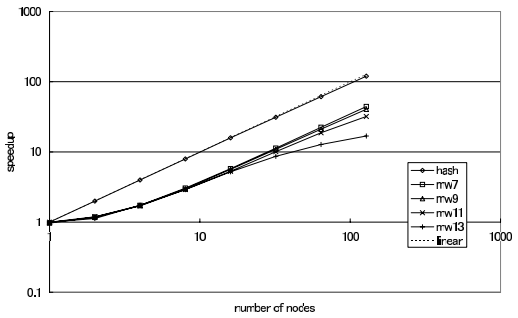

図 6 シミュレーションでの台数効果（問題サイズ 18，末端 1000 クロック）

Fig. 6 Simulated speedups (problem size: 18, leaf size: 1000).

全体的な傾向として，提案手法は良好な台数効果を出しているといえる．特に，通信と末端タスクの仕事量がほぼ等しいような設定（末端 1000 クロック）では，ほぼ完全な台数効果を得られている．マスタ・ワーカ型も，ノード数が多くなってからの台数効果は直線的に伸びている．これはつまり，ノード数に比例した形で速度が向上しており，どこかにボトルネックが生じることによる速度向上率の鈍化は生じていない，と

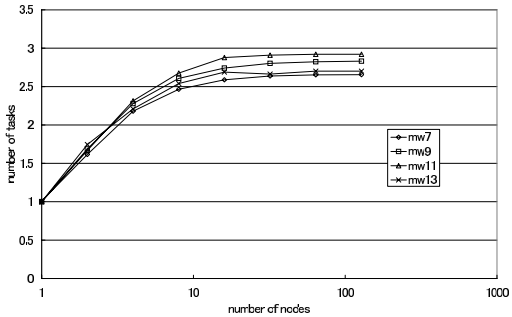

図 7 総タスク数
Fig. 7 Total numbers of tasks.

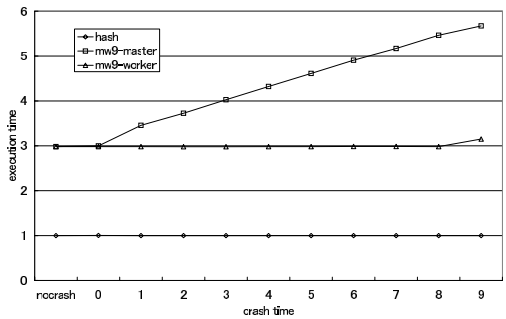

図 8 故障時の計算時間変化（128 並列，問題サイズ 18，末端 1000 クロック）
Fig. 8 Computation times when a failure takes place (128 parallel execution, problem size 18, leaf size 1000).

いうことである．ただし，速度向上率が一定の割合で低下していることになる．この原因は，マスタ・ワーカ型にしたことによる無駄なタスク生成に起因する．**図 7** は最終的に計算されたタスクの総数を示したものである．1 ノードで計算を行った際に必要だったタスク総数を 1 として，そこからの比をプロットしている．提案手法はノード数を変更してもタスク数は変化しない．マスタ・ワーカでノード数を増やしていくと，ワーカ間で重複したタスクを計算してしまうため，大きく総タスク数が増えていることが分かる．ワーカに与える仕事の大きさにはそれほど関係なく，最終的に 2.5 倍から 3 倍程度の探索タスク数で飽和することも見て取れる．これは，タスクの依存関係が，均等に部分問題の衝突が起きるような構造をしているため，どの深さでマスタとワーカの仕事を分割しても失われる共通部分問題の率が一定であるからであると考えられる．

3.3.3 故障ありの場合

次に，故障が発生する場合の評価を行う．シミュレーションにおける故障の設定は，以下のとおりである．

故障したノードは，故障が発生した時点から 30000 クロックの間（通信にかかる時間の 30 倍の間），動作をやめる．その間，他のノードの動作に影響はない．故障ノードに対する通信はすべて通信路にとどまったままの状態になる．30000 クロック経過後，故障ノードは内部に保持していた実行途中のタスク，および実行終了したタスク結果をすべて消去した状態で，再び動作を始める．つまり，まったく新たな代替ノードが新たに計算に参加した，というシナリオである．また，故障終了と同時に，通信路に存在している故障ノードへの通信は，すべて破棄される．

サイズ 18 の問題について，128 ノードで動作中，1 ノードを様々なタイミングで故障させたときの計算時間の変化を**図 8** に示す．前述の故障なしの場合の実験結果を参考にして，末端の仕事が 1000 クロック，マスタ・ワーカでの並列動作時，ワーカに分割するタスクの大きさは 9 という設定を用いている．横軸は故障させるタイミングの違いを表している．nocrash が故障なしの場合の計算時間，この故障なしの計算時間を 10 等分し，最初の時点（つまり 0 クロック目）で故障を起こした場合の計算時間を横軸の 0 で，次の時点で故障を起こした場合を 1 で，というように表している．縦軸は計算時間を表すが，分散ハッシュ方式で故障なしの時間を 1 として正規化したものとなっている．各系列は，分散ハッシュによる方式（hash），マスタ・ワーカ方式で，マスタが故障した場合（mw9-master），マスタ・ワーカ方式で，ワーカが故障した場合（mw9-worker），をそれぞれ表している．hash，mw9-worker は，故障させるノードを変更して実験を行ったが，故障させるノードの違いはほとんど結果に影響してこなかったので，ここでは適当なノードを故障させた代表的な結果を示している．

まず分かるのは，分散ハッシュ方式，マスタ・ワーカ方式ともに，適当なノードが故障してもまったくといっていいほど計算時間には影響しない，ということである．マスタ・ワーカ方式で，計算時間の後の方で故障が発生すると，若干トータルの計算時間が伸びてはいるが，全体的には故障時間はほぼ隠蔽されてしまう．

また，マスタノードが故障した場合は計算時間に大きく影響することも分かる．これは当然の結果で，マスタノードが故障復帰後，ワーカに割り当てたタスクの結果をすべて忘れて，新たにタスクを一から割り当て直すので，故障復帰時点よりもう一度最初から探索をやり直すのとほとんど変わらない．図 8 での計算時間の伸びは，最後の方の時点でマスタの故障が起きた場合，故障なしの場合のほぼ倍の計算時間がかかることを示しており，この考え方を裏付けている．

ここから，マスタ・ワーカ型の並列化は，耐故障性を

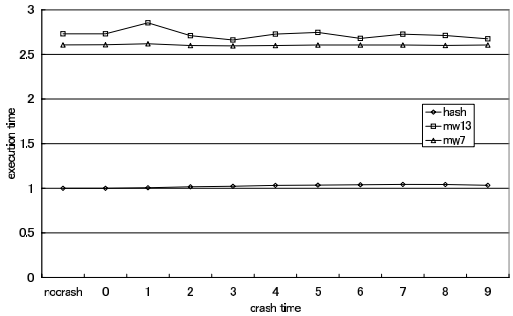

図 9 故障時の計算時間変化（8 並列，問題サイズ 18，末端 1000 クロック）

Fig. 9 Computation times when a failure takes place (8 parallel execution, problem size 18, leaf size 1000).

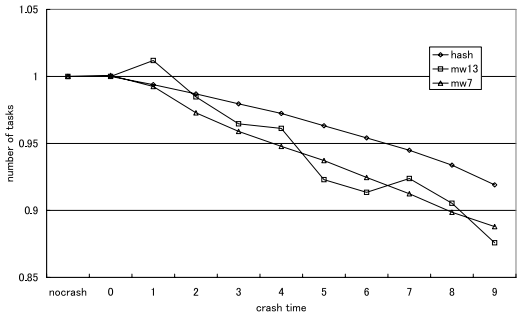

図 10 故障時の最終的なタスク数の変化（8 並列，問題サイズ 18，末端 1000 クロック）

Fig. 10 Total number of tasks when a failure takes place (8 parallel execution, problem size 18, leaf size 1000).

実現するうえではマスタの故障がきわめて大きな悪影響を与えることが分かる．今回提案する分散ハッシュを用いる並列化手法は，マスタノードのような single point of failure が存在しない，という点で有利であると考えられる．

以上のように，末端タスクの計算時間が大きい場合は故障時間がほとんど隠蔽されてしまった．末端タスクの計算時間を 10 クロックとした場合も同様の傾向だったが，分散ハッシュ表方式では静的に負荷分散を決めているため，1 ノードが故障していた時間がそのまま計算時間全体を伸ばす結果になった．このことから，分散ハッシュ表方式では何らかの動的負荷分散を必要とすることが明らかとなった．

次に，より多くの部分問題の記憶が失われる場合を想定して，8 ノードで計算を分担しているときに 1 ノードが故障する場合について，同様のシミュレーションを行った．問題サイズは前の実験と同じ 18，末端の仕事のサイズは 1000 クロックとしている．計算時間の変化を**図 9** に示す．マスタ・ワーカ方式で故障ノードはワーカのみとし，マスタの故障は今回は記していない．代わりに，マスタ・ワーカでのワーカの仕事の大きさを変えた場合について実験を行っている．図 9 から読み取れるように，どちらの方式においても，故障は全体の計算時間にそれほど影響を与えていない．分散ハッシュを用いたものが若干先ほどの 128 並列時よりは遅延が目立つが，ほぼ故障の影響なしといえる程度である．また，ワーカのタスクサイズ 13 のときにやや計算時間の変化が目立つ．この原因は，ワーカタスクサイズが大きいほど，故障後のタスク再割当てが変化したとき与える影響が大きくなる点にあると考えられる．

図 10 に，計算終了時に記憶されていた最終的なタスク数の変化を示す．縦軸は hash，mw13，mw7 の

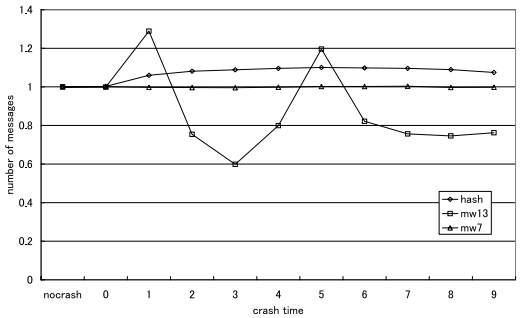

図 11 故障時のメッセージ数の変化（8 並列，問題サイズ 18，末端 1000 クロック）

Fig. 11 Number of messages when a failure takes plece (8 parallel execution, problem size 18, leaf size 1000).

それぞれの方式で故障が起きなかったときの最終的なタスク数を 1 として正規化してある．

マスタ・ワーカ方式はどちらも，ほぼ同様の傾きで最終的なタスク数が減少している．計算の最終盤で故障が発生したときにはおよそ 1/8 のタスクが記憶から失われている．これは，1 ノードの故障によってその時点の全体の 1/8 の記憶が失われる，と考えれば納得できる．これに対し，分散ハッシュ方式ではタスク数の減少割合が少ない．これはつまり，故障によって一時的に 1/8 の記憶が失われたものの，他のノードによって，計算に本当に必要なタスクが回復されていることを示している．このような回復プロセスが，全体の計算時間に影響を与えることなく実現されていることが重要であり，提案手法の大きな特長となると考えられる．

また，この実験において通信路に流れたメッセージ数の変化を**図 11** に示す．メッセージ数はそれぞれの方式での故障なしの場合を 1 として正規化して表示してある．分散ハッシュを用いる方法では，故障からの回復のために 1 割程度の余計な通信が発生すること

が分かる．もちろん，元々のメッセージ数は分散ハッシュとマスタ・ワーカで大きく異なり，分散ハッシュでは 1,400,000 個程度，マスタ・ワーカではどちらもおよそ 18,000 個程度のメッセージ数であった．つまり分散ハッシュの方法は 100 倍ほどメッセージ数が多いことになる．

3.4 シミュレーション実験のまとめ

シミュレーションの結果，提案する分散ハッシュを用いた並列計算手法は，末端のタスクの仕事量を通信にかかるコストと同等程度に調整しておけば，高い効率で並列計算ができることが示された．また，故障発生時にも，single point of failure を持たないという大きな利点があり，故障の影響をよく隠蔽して，全体の計算時間を遅くすることなく並列計算を継続することが可能である．故障時には，失われた部分問題の計算結果のうち，再利用しなければならない結果のみを，計算時間に影響を与えることなく回復することが可能であった．このとき，通信量は失われた部分問題の割合程度増加するが，計算対象の問題が大規模化し，参加する計算機の数が増えれば，通信量の増加分は少なくなっていくと考えられる．

比較対象としたマスタ・ワーカ方式は，通信量が少ない，動的負荷分散が容易，という特性を持ったまま耐故障性を実現することができ，シミュレーションの結果，故障の影響をほとんど完全に隠蔽することができた．ただし，マスタノードの故障に対しては弱いという欠点がある．また，ワーカ間のタスクの依存関係を完全に無視した並列実行を行うため，マスタ・ワーカ方式にした時点ですでに計算にかかる時間が長くなっており，結果的には提案手法の数倍の計算時間がかかるようになっている．

提案手法で問題なのは，故障により引き起こされた負荷の不均等を解消する動的負荷分散機構がない点である．マスタ・ワーカ方式のように，仕事がなくなったノードが別のノードのタスクを横取りするというワークスティーリングの手法を適用できれば，強力な動的負荷分散機構として働くと考えられるが，どのように実現すればよいかは今後検討する必要がある．

4. 試験実装と評価

シミュレーション結果を基に，提案する並列化フレームワークの試験実装を行った．

実装するシステムは C++の API を持つものとし，内部で Phoenix[4] を通信ライブラリとして使用している．

また，耐故障性の実現のため，実行可能なタスクが

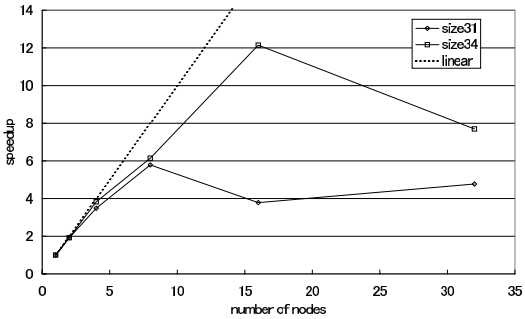

図 12 試験実装の台数効果
Fig. 12 Speedups of the experimental implementation.

10 秒以上現れなかったとき，故障を疑って query メッセージを子タスクに送るという実装をとった．故障が発生していない場合にも無駄な通信を行ってしまう可能性や，故障が発生しているにもかかわらずなかなかそれに気づかない可能性もあるが，試験実装としてこのような簡易的な手法を用いている．

実験はシミュレーションと同じ，15 パズルの反復深化 A*アルゴリズムを用いた探索プログラムによって行った．実験環境は，Xeon 2.4 GHz dual，2 GB memory の PC からなるクラスタであり，ネットワークは 1 Gbit ether で接続されている．まず，このフレームワークを用いる前のプログラムの探索時間と，フレームワークを用いて 1CPU で通信なしで実行したときの探索時間を比較すると，平均して 3.5 倍程度の探索時間が必要になっていた．もちろんこのオーバヘッドは，フレームワークを用いるときの末端タスクの粒度によって異なってくるものである．末端タスクの粒度設定は以後の実験を通して同じである．

図 12 は問題サイズの異なる 2 つの問題について，計算に参加しているノード数を変えながら台数効果を測定したものである．ここでの問題サイズは，与えられた盤面を解くための最短手順の長さである．測定は 3 回ずつ行い，最短の実行時間を採用した．特に台数が多くなってきたときに実行時間のばらつきが激しかったためである．これは，探索に用いているアルゴリズムが探索順序のコントロールをしていないため，通信タイミングなどの偶然性によって大きく探索木の形が変化しているためであると考えられる．問題サイズが小さい（size31）ときは 8 台程度まで，より大きいときは 16 台程度まで比較的良好な速度向上をしているが，それ以降は台数を増やしても速度が遅くなる結果になっている．

これは，タスク数の不足と個々のタスクの仕事量の不均一に大きな原因があると考えられる．A*アルゴリズムでは，探索の最後の方ではごく少数の局面のみを

表 1 故障発生時の探索時間 (sec)
Table 1 Computation times when a failure takes place (sec).

問題	ノード数	故障なし	故障発生時
size31	2	50.0	65.9〜92.9
size34	8	89.3	73.8〜144

探索していることが多く，一部のタスクの仕事量が他のタスクに比較して非常に大きくなりがちである．これを防ぐためには，何らかの動的な負荷分散を採用することが必要になると思われる．また，通信量が多くなったときのシステムの耐性も大きな問題である．タスクの粒度を小さくし，より多くのタスクを扱わせることで負荷の不均一は緩和されると思われたが，今回の実装では通信が多くなることによる弊害の方がよりはっきりと現れて，速度向上が得られない結果となった．同一の通信先へのメッセージをまとめるなどの手法を用いて，通信量を削減する必要があると思われる．

次に，故障を発生させた場合の速度低下を測定した．結果を**表 1** に示す．表 1 の 1 行目は，問題のサイズ 31，ノード数 2 台で探索中，ランダムに 1 台のプロセスを停止させ，3 秒後に新たなプロセスを立ちあげる実験の結果を表している．実験は 5 回行い，「故障なし」の欄の数字は 5 回の平均探索時間,「故障発生時」の欄は 5 回の実験の最短と最長の探索時間を示している．故障発生時の実行時間は大きくばらつくためこのような表記をとっている．問題サイズ 31，ノード数 2 台では，故障が発生するとおよそ 1.3 倍から 1.8 倍程度の速度低下が発生したことが見て取れる．サイズ 34 での故障発生時，故障なしの場合より計算時間が早くなった場合もあったが，これは一度だけ起きたものであり，おそらく探索木の形が大きく変化してしまったために偶然発生したものと思われる．ほとんどの場合は故障時に 110 秒程度の計算時間であった．このような大幅な速度低下が起きた原因は，故障発生が周囲のノードに伝わるのが遅くなる可能性があるという，現在の耐故障性実装方式の問題点に依ると考えられる．

5. 関 連 研 究

5.1 耐故障性を実現した研究

5.1.1 チェックポインティング

耐故障性を実現するための 1 つの手段として，メモリや IO の情報を定期的にディスクに出力し，故障時には以前保存したメモリイメージを復元して計算の途中状態まで回復する，チェックポインティングという手法があり，広く研究されてきた．計算機内部のメモリだけでなく，外部との通信も含めて一貫性を保ったまま途中状態に復帰できるようなチェックポイントを作成しようとするとオーバヘッドも大きくなるが，ユーザに故障を意識させることなく，耐故障性が得られるという点で強力な手法であるといえる．チェックポインティングは Condor [5] や Cactus [6] などの並列計算システムにも使われてきた．

しかし，チェックポインティングは基本的に，故障前の構成とまったく同じ構成を後から再現することを目的としており，ノードが 1 台壊れたときにそのまま，1 台少ないままで動き続けるといったことはできない．提案手法と比較して，参加計算機構成の変更に対応できない，というのは大きな問題である．

ところで，提案手法においても，個々のノードにおいて独立にチェックポインティングを用い，ノードの信頼性を向上させるということは十分意味があると考えられる．故障発生時に計算の途中状態まで復帰できれば，親タスクからの再計算要求に対して最初から再計算する必要がなく，故障の影響を小さくすることができる．

5.1.2 Embarrassingly parallel

完全にタスクが独立した計算として分割され，互いに依存関係がないような Embarrassingly parallel と呼ばれる問題では，きわめて大規模な計算でも並列計算することは可能であり，実用的なシステムも実現されてきた．たとえば，SETI@home [7] プロジェクトはこのような並列計算の最も大規模な成功例である．SETI@home のように，ボランティアベースの PC を多数用い，依存関係のないタスクをマスタ・ワーカ型で並列計算する，という手法を並列計算フレームワーク化したものが BOINC [8] であり，現在同様なプロジェクトに多数用いられている．

このような問題の耐故障計算のためには，マスタ・ワーカ型計算を行い，信頼性の高いマスタノードがワーカに配布したタスクを覚えておき，ワーカから結果が帰ってこないときは別のワーカに再配布する，という方式をとればよい．このようにして耐故障性は簡単に実現できるが，本提案のような問題に対しては計算効率が落ちる可能性があることはこれまで述べてきたとおりである．

また，Google map-reduce [9] は，規則的なデータ構造に対して，map と reduce という 2 種類の方法でユーザが記述した関数を並列に適用できるようなフレームワークである．シンプルでありながら適用範囲の広いフレームワークではあるが，対象問題は部分問題間に依存のない，単純なものに限られている．

5.1.3 Divide and Conquer

本提案とほぼ同様の，タスクが再帰的に子タスクを生成するような依存関係を持つ，Divide-and-Conquer 型の計算モデルを対象にした並列計算フレームワークとして，Cilk [10] があげられる．Cilk は work-stealing によって動的に負荷分散を行い，効率の良い並列計算を簡単に実現できる．しかし，共有メモリを前提にしたフレームワークであり，動的な構成変更や耐故障性については考慮されていなかった．

同様の計算モデルを対象に，分散計算や耐故障性などを実現したものに，Atlas [11] や Satin [12] がある．特に Satin では，ノードが故障したときそのノードが保持していたタスクの子タスクが失われることを防ぐ，global result table を導入するなどして，故障発生時の計算効率の改善を図っている [13]．

これらのシステムは部分問題が共有されている問題を対象にしていない．1.2 節で述べたように，部分問題を共有するような問題においては，部分問題の計算結果を再利用することで計算効率が大きく向上する．Satin は global result table を導入して重複する再計算を減らそうとしているが，我々の提案手法はより積極的にその方針を推し進める方向でフレームワーク設計を行っているともいえる．これらのシステムが実現している work stealing による強力な動的負荷分散機構を，提案手法に生かしていく方法を考えることも重要だと思われる．

5.2 分散計算のための研究

5.2.1 Distributed Hash Table

P2P による情報保持，情報検索のための手法として，Distributed Hash Table が提案され，広く研究されている．DHT の実現法として，chord [14]，pastry [15]，tapestry [16] などの手法が研究されてきた．

これらの手法は，多数の計算機が参加するネットワーク構造において，いかにハッシュテーブルのような一定の構造を保ち，あるハッシュ値に相当する資源を発見するか，ということを実現するものであるが，現在のところ，大規模な分散ストレージとしての応用用途を想定した研究がほとんどである．本提案では，部分問題の計算結果をハッシュテーブルに保持することで，新しい領域への応用を試みようとしている．

提案手法が通信に求める最も緩やかな要件は，通信先にいつかはパケットが届く，というものであるので，DHT の維持やその上での通信手法など，既存の研究で得られた知見は提案手法の様々な箇所に役立つと考えられる．

5.2.2 Phoenix

Phoenix [4] は，動的な構成変更への対応や耐故障性を持つ，並列計算用通信ライブラリである．問題の部分部分をある決まった範囲内に存在する仮想的な多数の計算機群に割り付け，仮想計算機の範囲を実計算機にマッピングすることで並列計算を行い，そのマッピングを変更することで動的な構成変更を可能にする．これは，分散ハッシュテーブルと同様に仮想化された資源を提供しているともいえる．Phoenix はメッセージパッシングモデルを提供し，通信のみを保証する低レベルな層の並列計算ライブラリである．

Phoenix は並列計算のための通信を提供するため，P2P より密な結合網を利用し，レイテンシやバンド幅などの面において高い性能を得ることができる．また，計算モデルが分散ハッシュテーブルの考え方と相性が良いため，今回はこのライブラリを用いて実装を行った．スケーラビリティの面では P2P での各種手法が有利であるので，これらの手法との使い分けなども今後検討する必要があると考えられる．

5.3 本提案と同じ問題領域を対象とする研究

5.3.1 DHT Driven Scheduling

ゲーム木探索の分野においては，本手法がとった，部分問題の計算結果を再利用しつつ並列計算する手法として，transposition table driven work scheduling [3] がすでに提案され，高い効率で並列探索が可能であることが示されている．これは，本提案と同じ問題領域を対象にしていると考えてよい．

ただし，この手法は耐故障性については考慮していない．子タスクの再実行による耐故障性を付加し，より広い範囲の問題に適用できるようなフレームワークとして整理することは重要であると考えられる．

6. まとめと展望

我々は，再帰的な依存関係を持ち，重複する部分問題が多く含まれるような問題を対象にした，耐故障性のある並列計算フレームワークの構築方法を提案した．部分問題の計算結果を分散ハッシュ表と同様の考え方で管理し，故障が発生した際には親タスクが子タスクを再実行することで，計算続行に必要な部分問題の計算を復元し，重複した部分問題は最大限発見して効率良く計算を行うことができる．このような対象問題の依存関係を表現するモデルを提案し，そのモデルを用いたシミュレーションを通して，提案する並列計算フレームワークは，大きく効率を低下させることなく実現することが可能であることを示した．また，提案するフレームワークの試験実装を行い，15 パズルの A*

探索アルゴリズムを題材に，並列実行の性能や実現された耐故障性の性能検討を行った．その結果，適切にパラメータを調整すれば良い並列実行性能を示すことができるが，動的負荷分散機構や通信量削減が必要であること，また故障検知とその復帰手法はさらに検討が必要であること，などの知見を得ることができた．

今後は，フレームワークの設計検討と実装を進め，多くの実問題に適用することでフレームワークの適用範囲の広さを検証したいと考えている．また，多くの実問題を検討することで，シミュレーションのモデルの妥当性についても検証を行いたい．さらに，より大規模な問題，より広域に広がった計算環境にも対応できるような実装手法について検討を進めていきたいと考えている．

謝辞 本研究の一部は科学研究費補助金若手研究（B）17700051「非定型問題を大規模分散環境で解くためのプログラミング環境に関する研究」において実施された．

参 考 文 献

1) OpenMP: OpenMP Sites. http://www.openmp.org/
2) Korf, R.E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence*, Vol.27, No.1, pp.97–109 (1985).
3) Romein, J.W., Plaat, A., Bal, H.E. and Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Search, *AAAI/IAAI*, pp.725–731 (1999).
4) Taura, K., Endo, T., Kaneda, K. and Yonezawa, A.: Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources, *PPoPP* (2003).
5) Litzkow, M., Livny, M. and Mutka, M.: Condor — A Hunter of Idle Workstations, *Proc. 8th International Conference of Distributed Computing Systems* (1988).
6) Allen, G., Benger, W., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E. and Shalf, J.: The Cactus Code: A Problem Solving Environment for the Grid, *HPDC*, pp.253+ (2000).
7) SETI@home: SETI@home Sites. http://setiathome.berkeley.edu/
8) Anderson, D.P.: BOINC: A System for Public-Resource Computing and Storage, *5th IEEE/ACM International Workshop on Grid Computing* (2004).
9) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: 6th Symposium on Operating System Design and Implementation* (2004).
10) Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *Journal of Parallel and Distributed Computing*, Vol.37, No.1, pp.55–69 (1996).
11) Baldeschwieler, J., Blumofe, R. and Brewer, E.: ATLAS: An Infrastructure for Global Computing (1996).
12) van Nieuwpoort, R.V., Kielmann, T. and Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications, *ACM SIGPLAN Notices*, Vol.36, No.7, pp.34–43 (2001).
13) Wrzesinska, G., van Nieuwport, R.V., Maassen, J., Kielmann, T. and Bal, H.E.: Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments, *Accepted for publication in International Journal of High Performance Applications* (2005).
14) Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications, *Proc. 2001 ACM SIGCOMM Conference*, pp.149–160 (2001).
15) Rowstron, A. and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, *IFIP/ACM International Conference on Distributed Systems Platforms* (*Middleware*) (2001).
16) Zhao, B.Y., Kubiatowicz, J.D. and Joseph, A.D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing, Technical Report UCB/CSD-01-1141, UC Berkeley (2001).

(平成 18 年 7 月 4 日受付)
(平成 18 年 12 月 6 日採録)

横山 大作（正会員）

1974 年生．2000 年東京大学大学院工学研究科情報工学専攻修士課程修了．2002 年同博士課程中退．同年より東京大学新領域創成科学研究科助手．2006 年同大学より博士号取得．博士（科学）．並列計算量理論，並列プログラミングライブラリ，およびゲームプログラミングに関する研究に従事．日本ソフトウェア科学会，IEEE-CS 各会員．

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）．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講師．2002 年より同助教授．並列・分散処理，プログラム言語に興味を持つ．ACM，IEEE，日本ソフトウェア科学会各会員．

近山　隆（正会員）

1953 年生．1982 年 3 月東京大学大学院工学系研究科情報工学専門課程博士課程修了．同年 4 月富士通株式会社入社．同年 6 月（財）新世代コンピュータ技術開発機構に出向．1995 年 4 月東京大学工学系研究科助教授，同教授を経て，1999 年 4 月より東京大学新領域創成科学研究科教授．日本ソフトウェア科学会，人工知能学会，ACM，Association for Logic Programming 各会員．プログラム言語と処理系，並列分散処理，機械学習に興味を持つ．