

大量のコンピューティングリソースを活用するための ソフトウェア基盤

横山大作

平成18年6月9日

概要

近年、並列・分散計算環境は急激に普及している。素子の発熱などの問題により単体計算機の性能向上が妨げられている現在、高性能な計算機を現実的なコスト性能比で構築するためには、多数の計算機を用いた並列計算を用いなければならない。また、計算機は急速にコモデティ化しているため、普及型 CPU を多数集めて構成した大規模計算機や、パーソナルコンピュータを一般的なネットワークで接続した PC クラスタといった構成が多く使われている。また、ネットワーク技術の発展により、複数のクラスタを結合させたグリッドや、個人の遊休計算機を使用するデスクトップグリッドなど、より広域に広がった分散計算環境も実用的な存在となってきた。さらに、単体 CPU 内においてもマルチコアプロセッサが一般的なものとして普及を始めている。このように、近年の計算機はさまざまなレベルの並列性を持ち、大量の計算リソースを一度に利用できる並列環境を利用することも比較的簡単になりつつある。

しかし、現在のソフトウェア開発においてこのような環境を十分に活用することは依然として難しい。これは、並列プログラムを作る際に、

- (1) いかに実行効率が良好な設計を行なえるか
- (2) いかに正しく実装するか

の二つの側面において、未解決な問題点が存在しているためである。

(1) 実行効率の良いプログラムを設計するためには、あるアルゴリズムがある環境ではどれくらいの時間で実行できるのか、を正確に把握するための計算量モデルが必要になってくる。これまでにいくつかの並列計算量モデルが提案されてはいるが、これらは単純過ぎて現実の計算環境を表現できていなかったり、特定の環境や特定のアルゴリズムに限定したモデルになってしまっていたりと、満足できるものにはなっていない。アルゴリズム設計において必要とされるのは、特定の環境における実行時間の精密な見積もりよりは、現在のさまざまな計算環境とこれから使用可能になるであろう計算環境とで、対象アルゴリズムがどのような振る舞いを示すのか、を定性的に示すようなモデルである。現在、シングルコア、マルチコア SMP からデスクトップグリッドまで、規模もネットワーク構成も大きく異なる多種の計算環境が利用可能であり、これら並列環境の構成方式のトレンドも日々変化している。このような多種多様な環境を、何らかのモデル化によって統一的に扱い、与えられたアルゴリズムが共通にどのような振る舞いを起こすのか、という実行性能予測を可能にするような計算量モデルが求められているのである。

そこで、計算コストは通信コストであるという基本理念の元に構築した新しい並列計算量モデル「アクセス計算量モデル」を提案し、その有効性を実証することを試みた。アクセス計算量モデルの提案では、単体計算機内部のメモリ階層から、LAN、インターネットといったグローバルなネットワークを用いた通信までを、統一的にかつ簡潔に表現し、ネットワークトポロジなど並列計算環境の具体的な構造に依存しない、一般性の高い計算モデルを構築しようと考えた。アクセス計算量モデルでは、メモリがある密度で広がる空間を考え、計算はメモリ上の任意の地点で、任意の密度で行えるとする。計算を行うものを計算実体と呼ぶことにするが、これはレジスタなどの記憶を持たず、メモリ上に現在いる「場所」と、プログラム中の実行箇所を示す「プログラムカウンタ」の状態のみを保持している。メモリ空間には何らかの距離の指標が定義されており、計算実体からメモリ上のある地点に存在するデータへアクセスする際には、その距離に従ったアクセスコストがかかるものとする。計算の過程は、演算に必要なデータを計算実体のある場所まで移動し、そこで演算を行い、演算結果を所定のメモリ位置に通信によって書き込む、という一連の流れで表現され

る。計算にかかるコストは、この通信にかかる時間コストのみであり、演算にはコストがかからないとする。計算自体は同一点でいくらでも行うことができるが、通信路には容量があり、通信路の混雑によって計算に必要なデータが届かないため、同一点でいくらでも多くの計算を同時に実行できるわけではない。

本論文では、このような概念に基づくモデルを提案した。より詳細に、1 次元のメモリ空間と、局所的な負荷が表現できる通信路からなるモデルを定め、そのようなモデル上でのプログラムを表現できる仮想機械を定義、設計した。また、よく知られた並列アルゴリズムである bitonic sort、merge sort、FFT について解析的に計算量を求め、実計算機上の実行結果を用いた評価を通して、モデルの妥当性と実用性を示した。

(2) バグのない並列プログラムを書くことは、それ自身難しいことである。並列計算環境はマルチコアやマルチプロセッサなどの共有メモリ型から PC クラスタ、ベクトル計算機やデスクトップグリッドまでさまざまな種類があり、それぞれの構成において計算性能を上げやすいプログラミングモデル、通信ライブラリが異なっている。ある環境である問題を並列に解きたいときにどのような計算モデルや通信ライブラリを選択するのか、あるいは複数の手法を組み合わせなければならぬのか、という決断はプログラマに深い知識と経験を要求する。また、別の計算環境が使えるようになったり、計算機の規模が大きくなったりという変化があったとき、プログラムの変更が必要になるようでは生産性も低く、バグも入りやすい。さまざまな環境で同一のプログラムが正しく動く、というプログラミング手法が求められるのである。さらに、大規模問題を解くためには長期間、多数の計算機を使う必要があるが、長期間同じ計算機を占有し続けられる状況は珍しく、多くの場合は使用可能な計算機が増減する。また、多数の計算機を使うとどこかで故障が発生する確率が高まるため、部分的な故障で計算全体が停止してしまうようなプログラムでは計算が進まない。よって、動的な計算資源の増減への対応、耐故障性の実現などが求められるのであるが、深い知識と多大な労力が必要となるため、プログラマが自分で実現することは難しい。

プログラマの負担を減らすための一つの方法に、並列計算フレームワークがある。ある問題領域に適用範囲を絞り、その問題に特有の並列化手法を専門家があらかじめ実装しておくことで、プログラマは並列化手法の詳細について悩むことなく並列プログラムを書くことができる。また、このフレームワークは並列環境の差異を隠蔽し、それぞれの環境で効率のよい実装を使い分けるなどして実装されているため、ひとつのプログラムで、さまざまな環境で効率よく計算することが可能になる。耐故障性の導入なども、問題領域を絞ることでプログラマへの負担を軽減した形で実現できる。

科学技術計算などの構造が単純な問題においてはこのようなフレームワークが存在し、高性能計算を簡単なものになっているが、より複雑な構造を持つ問題、例えば組合せ最適化やゲーム木探索などに代表される探索問題については、まだ研究が不十分な点も多い。探索問題は実社会においても応用範囲が広く、規模が大きくなると莫大な計算量を必要とするため、並列計算の技法を用いて実用的な時間でより大きな問題を解けるようにすることが強く求められている。探索問題において特に問題になるのは、共通の部分問題に依存した多くの部分問題が存在する点である。単純に並列化を行うと、これらの共通問題を別々の計算機で重複して計算してしまうため、無駄な計算が増えて探索性能の向上に結びつかない。このような共通部分問題を効率よく発見し、計算結果を再利用しながら並列計算を行うような手法が求められている。そこで、このような性質を持つ探索問題をターゲットとする並列計算フレームワークを設計・構築し、並列プログラム開発のための基盤技術とすることを試みた。

提案する並列探索手法は、親タスクが子タスクを再帰的に生成し、子タスクの計算結果を使って親タスクの結果を計算するような構造の問題を対象とし、分散ハッシュ表 (DHT) の考え方をを用い

て部分問題を管理する。対象とする問題領域では、部分問題は共通性が発見できるよう一意にコード化されるので、ハッシュ表を用いて共通部分問題を効率よく発見し、計算結果を再利用しながら探索を進める。DHT は近年研究が盛んであるが、多数の計算機でファイルなどの情報を保持するための手法として捉えられていることが多い。また、DHT の考え方を探索に用いる研究もあったが、分散システムの重要な特性である耐故障性能について考慮したものは存在していなかった。今回の提案では、故障時には失われた部分問題を必要とする親問題が子問題を再実行し、必要な探索途中結果を DHT 上に再構成する、という方法で耐故障性を実現する。

本論文では、提案する手法が、共通部分問題を含む問題領域において良い探索効率を与えること、故障が発生した際も性能悪化が少ないことを、既存の分散計算手法であるマスタワーカとの比較シミュレーションによって示した。また、提案する手法をフレームワークとして利用できるようにインタフェースを整理し、実装を行い、シンプルな探索問題を用いて評価を行った。

このような2つの領域の研究の両方がそろって初めて、現在身近になり、これからさらに普及し続けると考えられる並列計算環境を「活用」することができると考えている。本論文では、新しい並列計算量理論と、共通部分計算を持つ探索問題の大規模耐故障並列化フレームワークの提案を行うことで、並列プログラムをさまざまな環境で効率よく、正しく簡単に実装できるような基盤技術を拡充することができた。

目次

第 1 章	はじめに	7
1.1	大量のコンピューティングリソースの普及	7
1.2	ソフトウェア開発の困難さ	8
1.2.1	性能予測のための計算量モデルの欠如	8
1.2.2	実装上の困難	9
1.3	提案	10
1.4	本論文の構成	10
第 I 部	アクセス計算量モデルに基づく並列アルゴリズムの計算量予測	12
第 2 章	計算量モデルに求められる表現力	13
2.1	メモリ階層	13
2.2	多様な並列・分散計算環境	14
2.2.1	単一プロセッサ内の細粒度並列性	14
2.2.2	共有メモリ型マルチプロセッサ	16
2.2.3	クラスタ	18
2.2.4	GRID と desktop GRID	19
2.2.5	多様な並列計算環境の共通モデル化の必要	20
2.3	解析可能な単純さ	20
第 3 章	関連研究とその問題点	22
3.1	(P)RAM	22
3.2	HMM, UMH	22
3.3	LogP	23
3.4	メモリモデルと通信路モデルの組み合わせ	23
3.5	Coarse Grained Multicomputers, Bulk-synchronous Parallel	24
3.6	既存モデルの問題点	24
第 4 章	アクセス計算量モデルの提案	25
4.1	提案する計算モデルの目標	25
4.2	モデル概要	26
4.3	仮想機械	28
4.4	仮想機械シミュレータの設計と構築	29

第 5 章 通信路モデル	30
5.1 通信衝突モデル	30
5.2 通信負荷累積モデル	31
第 6 章 並列アルゴリズムの解析	35
6.1 diremption factor	35
6.2 bitonic sort	35
6.3 merge sort	38
6.4 FFT	40
6.5 解析のまとめ	42
6.6 クラスタ構成での通信路モデル化の評価	42
第 7 章 第 I 部のまとめと今後の展望	44
 第 II 部 多くの部分計算を共有する問題の耐故障計算フレームワーク	 45
第 8 章 背景	46
8.1 並列・分散計算環境の普及	46
8.2 耐故障計算フレームワークの必要性	47
8.3 部分計算を共有する問題	49
第 9 章 耐故障計算フレームワークの提案	53
9.1 フレームワークの方針概要	53
9.2 対象とする問題	53
9.2.1 樹状再帰構造	53
9.2.2 部分問題の決定性・副作用の排除	54
9.2.3 部分問題の発見のための仕組み	54
9.2.4 対象問題の探索の定式化	54
9.3 システムの必要要件	55
9.3.1 通信の要件	55
9.3.2 故障モデル	56
9.4 アルゴリズム概要	56
9.4.1 タスクの分担	56
9.4.2 タスク依存関係の保存	57
9.5 提案アルゴリズムの利点と欠点	58
9.6 適用問題を広げるための拡張	59
9.6.1 メモ化のための入力に関する適用問題拡張	59
9.6.2 A*アルゴリズムへの適用例	60
第 10 章 関連研究	63
10.1 チェックポイントニング	63
10.2 Embarrassingly parallel	63
10.3 Satin, Cilk	63
10.4 Distributed Hash Table	64

10.5 DHT driven scheduling	65
10.6 Phoenix	65
第 11 章 フレームワーク設計	66
11.1 各ノードのアルゴリズム	66
11.1.1 計算中	66
11.1.2 故障発見時	68
11.1.3 参加	69
11.1.4 脱退	69
11.2 フレームワークの API	69
11.2.1 基本 API	69
11.2.2 適用問題を広げるための拡張 API	70
11.2.3 ユーザコードの例	73
11.3 DHT 内部の Garbage Collection	76
11.4 探索効率向上のために今後拡張していくべき機能	77
第 12 章 シミュレーションによる評価	79
12.1 シミュレーションのためのタスク生成モデル	79
12.1.1 実アプリケーションのタスク生成	79
12.1.2 タスク生成モデル	79
12.2 マスタワーカモデル	81
12.3 シミュレーションによる比較実験	83
12.3.1 シミュレーションの設定	83
12.3.2 故障なしの場合	83
12.3.3 故障ありの場合	92
12.4 シミュレーション実験のまとめ	97
第 13 章 試験実装	99
13.1 試験実装の方式	99
13.2 試験実装を用いた実験	99
13.3 実装における性能向上への課題	101
第 14 章 第 II 部のまとめと今後の展望	102
第 15 章 終わりに	103
15.1 本論文のまとめ	103
15.2 今後の課題と展望	104
15.2.1 新しい計算量モデルについて	104
15.2.2 並列計算フレームワークについて	104

第1章 はじめに

1.1 大量のコンピューティングリソースの普及

近年の計算機環境において、速度向上のために並列性を用いることはもはや避けては通れない手法となってきた。過去数十年にわたり、プロセスの微細化、高クロック化によって単体計算機の速度は向上し続けてきたが、近年では、発熱などの問題によってその向上のペースが大きく妨げられている。演算素子自体のコスト、ならびに演算や冷却に必要な電力コストを考えた際、さらに高性能な計算機を現実的なコスト性能比で構築するためには、多数の計算機を用いた並列計算機としないといけないのはもはや自明である。

また、計算機は急速にコモデティ化している。パーソナルコンピュータは一般社会に広く普及し、大量に生産される CPU は競争を通じてより安価に、高性能に進歩し続けている。また、ネットワーク接続用機器についても同様であり、計算機間をつなぐ LAN のためのハードウェアは年々大容量化し、価格も低く抑えられている。これらのコモデティ部品を大量に用い、普及型 CPU を多数集めて構成した大規模計算機や、PC を一般的なネットワークで結合した PC クラスタのような並列環境を構築することで、高性能計算機を安価に構築することが可能になっている。現在、世界に存在する超高性能計算機の多くはこのようなクラスタ構成で実現されており、今後も同様の構成がますます支配的になるであろうと考えられる。また、PC クラスタの価格、及び維持コストが下がったため、比較的小規模の PC クラスタを研究室単位、部所単位で構築することが可能になり、このような小規模の並列環境は身近なものとなっている。

ネットワーク技術の向上により、より広域に物理的に広がった計算環境というものも実用的な存在となってきた。前述の大小さまざまなクラスタは世界中のあちこちに点在しているが、これらのクラスタ同士を一般的なネットワーク、例えばインターネットで結び、複数のクラスタを一体な並列計算機として使用する、グリッドと呼ばれる計算環境も急速に普及しつつある。また、個人の持つ単体 PC 同士をインターネットで結んで、多数の PC を一斉に使うことで大規模計算環境にしようという、デスクトップグリッドと呼ばれる考え方も生まれてきた。

ボランティアコンピューティングと呼ばれる枠組みでは、このようなデスクトップグリッドを用いており、プロジェクトに賛同する世界中のボランティアたちの計算機の遊休時間、例えばスクリーンセーバーが動いている時間などを利用して計算を行っている。ボランティアコンピューティングのプロジェクトは、電波天文台の信号処理やタンパク質の構造解析、様々なシミュレーションや数学的問題など、莫大な計算量が必要な問題を解くためにいくつも立ち上がっており、成果を上げてきた。

また、デスクトップグリッド的な考え方を企業内の遊休計算機に適用し、自社の計算資源として活用したり、他社に計算資源を必要に応じて売却したり、という使い方をしようという試みも始まっている。このようにデスクトップグリッドは使われ始めており、今後も一層発展すると考えられている。

さらに、単体 CPU 内においても並列化は進んでいる。プロセスルールの縮小により、CPU に大量のトランジスタ資源を使うことが可能になったものの、設計コストの増大、発熱が増大する問

題などにより単一 CPU での設計ではその資源が十分活用できない事態に陥りつつある。そこで、複数の CPU を 1 つのパッケージに納め、パッケージ全体での性能を向上させようというマルチコアプロセッサが提唱され、一般的な CPU として昨今いよいよ普及が始まった。今後はさらにこの流れが進み、より低価格な一般向け CPU に、より多くのコアが入っていくと考えられる。

このように、近年の計算機はさまざまなレベルの並列性を持ち、大量の計算リソースを一度に利用できる並列環境を利用することも比較的簡単になりつつある。そして、この流れは今後も続き、ソフトウェアはますます大量のコンピューティングリソースを使用可能になってゆくと考えられる。

1.2 ソフトウェア開発の困難さ

前述のように、大量の計算リソースを利用できる環境は整いつつある。しかし、現在のソフトウェア開発においてこのような環境を十分に活用することは依然として難しい。

これは、並列プログラムを作る際に、

- いかに実行効率が良好な設計を行なえるか
- いかに正しく実装するか

の二つの側面において、未解決な問題点が存在しているためである。

1.2.1 性能予測のための計算量モデルの欠如

実行効率の良いプログラムを設計するためには、あるアルゴリズムがある環境ではどれくらいの時間で実行できるのか、どのような箇所に計算時間がかかり、どのような箇所でもボトルネックを生じるのか、といったことを正確に把握するための計算量モデルが必要になってくる。これまでにいくつかの並列計算量モデルが提案されてはいるが、これらは単純過ぎて現実の計算環境を表現できていなかったり、特定の環境や特定のアルゴリズムに限定したモデルになってしまっていたりと、必ずしも満足できるものにはなっていない。

例えば、最も広く用いられている並列計算量モデルは PRAM (Parallel Random Access Memory モデル) である。これはアルゴリズムの実行時間を演算回数によって表現するモデルであり、極めて単純であるが故に広く使われてきたが、現実の計算機では演算回数が計算時間を表してはいない。以前の計算機、CPU とメモリの速度があまり乖離しておらず、メモリ階層がそれほど深くなかった頃の計算機であれば、PRAM モデルの仮定がよく適合したのであるが、現在の計算機は CPU の演算にかかる時間とメモリアクセス時間が大きく乖離しており、メモリ階層を考慮に入れない計算量モデルでの計算時間予測は現実と大きくかけ離れたものになってしまう。

現在の並列計算においては、シングルコア、マルチコア SMP からデスクトップグリッドまで、規模が大きく異なる多種の計算環境を利用することができる。クラスタ構成の計算機にしても、単体 CPU 速度とネットワーク速度の比やネットワークトポロジが異なった、実に多様な構成が存在する。さらに、これら並列環境の構成方式のトレンドは日々変化しており、モデルとなるような計算環境構成を定めることは難しい。並列アルゴリズム設計において、これら多様な構成のそれぞれについて特化したアルゴリズムを設計することは困難である。よって、これらの構成のうち多くの構成において共通に高い性能を発揮できるような、一般化された構成を対象としたアルゴリズムを設計しようとする。そのためには、一般化された構成での計算環境の振る舞いを正しく示すようなモデルが要求されることになる。

よって、このように規模も構成も大きく異なる多種多様な環境を、何らかのモデル化によって統一的に扱い、与えられたアルゴリズムの実行性能を予測する、という計算量モデルが求められているのである。

1.2.2 実装上の困難

多様な計算環境はまた、プログラミングそのものを困難なものにする。もともと、並列プログラムは複雑になりがちであり、同期、通信などさまざまな要素が絡んでバグが発生しやすい。さらに、SMP においてはマルチスレッド・マルチプロセスプログラミングを、クラスタにおいては MPI などのライブラリを、さらに GRID 環境においてはインターネット環境に適応した通信ライブラリを、といったように、プログラムに必要なライブラリをそれぞれ使い分けことが一般的であるが、それぞれのライブラリには固有の知識が必要であり、新しい知識を習得するために余分な労力が必要となる。時には、ライブラリの変更のためにプログラミングモデルまで変更を余儀なくされる場合すらあり、開発者には大きな負担を与えることになってしまう。このため、ライブラリ変更を行なう際にバグを作り込んでしまうことも多い。

また、開発段階では小規模な並列環境で実装、実験を行ない、次第に大規模な環境で実問題をターゲットとした計算を行なっていく、というソフトウェア開発の流れは多い。あるいは、計算機器が年々低廉化するため、ソフトウェアがそのままでも計算環境は次第に大規模なものに変更されていく、という場合も多い。このように、小規模な環境から大量のリソースを利用できる環境まで、シームレスに同じソフトウェアを動かしたいという要求は大きいと考えられる。しかし上述のように、個別の環境でライブラリを変更するとすると、このような連続的なリソース規模の拡大には対応しにくい。

このような問題に対する一つの解決策として、並列計算フレームワークを用いるという方法がある。種々の問題をまとめて、抽象的な、ある種の問題領域、ととらえる時、その問題領域全体で共通する並列化手法が存在することが多い。そこで、そのようなプログラムの並列化部分のみをフレームワークとして定式化し、並列処理をよく知る開発者があらかじめ実装して提供する。個々の問題固有知識を持つプログラマは、そのフレームワークに当てはめる形でプログラミングすることで、並列化に関する種々の知識を必要とすることなく、正しく簡単に並列環境を使用することができるのである。

このような並列化フレームワークは既にいくつか提案されている。例えば、科学技術計算に多く用いられてい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に、並列化フレームワークが研究されてきている。この分野では、例えば OpenMP[1] が有名である。

しかし、より複雑な構造を持つ問題に対しての並列化フレームワークについての研究は、まだ不十分な点も多い。組合せ最適化問題やゲーム木探索などに代表される探索問題は、大量のコンピューティングリソースを活用できるフレームワークを構築するために、まだまだ研究が必要な分野のひとつである。探索問題は配置設計や生産計画、配送計画や人工知能などの実問題に広く用いられている問題であり、問題サイズに対して必要リソースが爆発的に増大するため、大量のコンピューティングリソースを用いて、実用的な時間の範囲でなるべく大きな問題を解けるようにすることが常に求められている。ここで問題になるのが、共通する部分問題の扱いである。ある探索問題を、多くの部分問題からなる依存関係グラフとして表した時、ある部分問題に多くの部分問題が共通して依存している、逆に言うならば、多くの部分問題がそれぞれの結果を求めるためにある同一の部分問題を共有している、という構造を含む場合がある。このような種類の探索問題を並列に

解く時には、共有された部分問題を発見し、部分問題の結果を再利用することが探索性能の効率を大きく左右する要素となる。部分問題の共有を無視し、結果の再利用をあきらめるならば、部分問題間の依存関係が簡略化されて容易に並列化が可能になるが、同一の部分問題結果をあちこちで何度も計算してしまうことになり、結果、探索性能が上がらないことも多い。

以上で述べたように、多数の部分計算を共有する探索問題を対象としたソフトウェア開発において、並列化フレームワークは強く要望されるものとなっているが、その並列化手法や、耐故障性などの要件を満たす構成方法はまだ研究の余地があり、問題点を解決したフレームワークの構築が求められていると考えられる。

1.3 提案

前章までで、大規模なコンピューティングリソースが物理的には使用可能になっているにも関わらず、それを活用するようなソフトウェア開発が十分にはなされていないことを述べ、その原因は、「性能予測のための計算量モデルが欠如していること」と「並列プログラム開発を容易にするような基盤が不十分であること」に依る、と結論づけた。

本稿では、この2つの原因それぞれに対して改善策を提案し、大量のコンピューティングリソースを活用できるようなソフトウェア基盤を構築する。

計算量モデルの欠如の問題に対しては、計算コストは通信コストであるという基本理念の元に構築した新しい並列計算量モデル「アクセス計算量モデル」を提案し、その有効性を実証することを試みた。アクセス計算量モデルの提案では、単体計算機内部のメモリ階層から、LAN、インターネットといったグローバルなネットワークを用いた通信までを、統一的にかつ簡潔に表現し、ネットワークトポロジなど並列計算環境の具体的な構造に依存しない、一般性の高い計算モデルを構築しようと考えた。

また、並列プログラム開発のための基盤不足の問題に対しては、部分問題を共有するような探索問題をターゲットとする、耐故障性を有する大規模並列化プログラミングフレームワークを設計・構築することを試みた。対象としている問題は、探索問題などに多く見られる、再帰的な依存関係のあるものであり、重複する部分問題計算を多く含んでいるため、以前計算した同一の部分問題の計算結果を再利用することで、大幅に計算効率が改善されるような特徴を持つ。このような問題を、分散ハッシュテーブルの考え方を用いて並列計算することができるようなフレームワークを提案した。耐故障性については、失われた部分問題を再実行により求め直すことで、必要最小限の再計算によって実現することを目指した。

1.4 本論文の構成

本論文の構成は以下の通りである。

大きく二部構成を取り、第I部では新しい並列計算量モデルであるアクセス計算量モデルを提案する。2章では計算量モデルの要件についてまとめ、3章では既存の計算量モデルの紹介とその問題点を明らかにする。4章でアクセス計算量モデルの概要を、5章ではモデル中の重要な要素である通信路に関してのモデル化の方法を述べる。6章ではいくつかのよく知られた並列アルゴリズムに対してアクセス計算量モデルを用いた解析を行ない、実験結果と比較することでこのモデルの有効性を示す。7章でアクセス計算量モデルに関するまとめを述べる。

次に、第 II 部では多くの部分計算を共有する問題の耐故障計算フレームワークを提案する。8 章では現在の計算環境上で求められている並列化フレームワークについて、要件と本研究で対象とする問題領域について述べ、9 章で提案するフレームワークの概要をまとめる。10 章では関連する個別技術について紹介し、耐故障性をもつ並列化フレームワークにはどのような個別技術の考え方を使得いけばいいのか、その特質と問題点についてまとめる。11 章では提案手法の設計についての詳細をまとめ、12 章ではシミュレーションによる提案手法の評価を、13 章では試験実装による提案手法の評価を行う。14 章で提案した耐故障並列化フレームワークに関するまとめを述べる。

最後に、第 15 章で、大量のコンピューティングリソースを活用するという目的に関する本研究のまとめを行う。

第I部

アクセス計算量モデルに基づく並列アルゴリズムの計算量予測

第2章 計算量モデルに求められる表現力

計算量モデルとは、あるアルゴリズムがある環境ではどれくらいの時間で実行できるのか、の予測を行ない、その指標を提示するものである。このモデルを用いることで、複数のアルゴリズムの実行効率を比較し、どのような場合にどのようなアルゴリズムを用いればよいのか、というソフトウェア設計の指針を得ることができる。

計算量モデルは、現実の計算環境を何らかの意味で抽象化し、その振舞いを精度良く予測できるものでなければならない。現実の並列計算環境の振舞いを予測しにくいものになっている要因に、メモリ階層の存在と並列計算環境の多様さが挙げられる。

2.1 メモリ階層

近年、計算機アーキテクチャは、少量の速いメモリと大量の遅いメモリが存在するという階層構造を持ち、しかもそのメモリ階層は年々深くなり続けている。これは、メモリ素子の速度向上がCPUの速度向上に追いつかないため、およびCPUとメモリの物理的な距離による本質的な限界が存在するため、などに大きく起因している。

メモリ素子の速度を向上させるためには、多量の電力が必要な高速で複雑な回路を用いなければならない。しかし、CPUの速度が向上すると、それに相当したペースで必要な記憶容量も増えることが求められる。ある時間内でできる仕事が増えれば、それに見合った仕事を記憶しておくための容量が必要になるわけである。となると、消費電力の大きい高速メモリを用いては、電力や廃熱などの問題から現実的な範囲で使える部品になり得ない。その結果、大容量のメインメモリに用いられる素子はアクセス速度が低いものにならざるを得ない。もちろんこのままでは計算性能に悪影響が出るため、高速なメモリ素子を用いて少量のキャッシュメモリをCPU内部に構築し、レイテンシを隠蔽することを狙う設計がなされる。

また、メインメモリとCPUは通常別々のチップとして実現され、帯域の限定されたバスなどを用いて結合される。これは、設計の複雑さや熱密度などの設計上の制約、歩留まりや必要なプロセスルールの違いなどの製造上の制約、部品の再利用性や構成自由度の必要性などの商業的制約、といった様々な理由から、ほぼ今後も変えられない設計方針であると考えられる。このような条件下では、CPUとメインメモリの間には物理的にある程度長い距離が存在してしまう。高速、広帯域の通信路を長い距離で安定して実現するには、電気的な問題などの様々な課題を克服しなくてはならず、現実的にはCPUの速度向上と比較して転送インタフェースの高速化のペースは落ちてしまう。特に、転送帯域幅は比較的容易に大きくすることができるが、転送にかかるレイテンシは物理的制約から発生する限界があり、あまり短くすることができない。このレイテンシの方が、プログラムを高速に実行するためには特に厳しい障害として現れてくることが多い。

これらの原因から、CPU内部のレジスタ、L1キャッシュメモリ、L2メモリ、メインメモリという、いくつかのレベルに分かれた階層的なメモリ構造が一般的になっている。そして、CPUの速度向上のペースにメインメモリの速度向上が追いつけないことから、このメモリ階層は年々深化

し、今後もさらに深化していくと考えられているのである。

現在の計算機において、メモリ階層がどのようにプログラム実行に影響を与えているか、簡単なベンチマークプログラムを用いて示すことにする。

図 2.1 は SPARC III 1.2GHz, 8CPU の共有メモリ計算機において配列の要素へランダムにアクセスする際の 1 要素あたりのアクセス時間を測定したものである。余計な演算なしで配列にランダムにアクセスするために、配列上のデータにあらかじめランダムなアクセス順序を記憶させることにした。つまり、連続したメモリ領域の要素がランダムに並べ替えられた 1 本のリストとしてつながっており、リストをたどって全要素を読むことで、全メモリ領域をランダムに 1 回ずつ読むことができるようになっている。この手順を繰り返して実行時間を測定することで、さまざまな大きさの配列のデータを読む際のメモリアクセスコストのみを得ようとした。系列 1 は 1 スレッドで、2-から始まる系列は 2 スレッドで、のように複数スレッドで実験を行ない、複数スレッドでも全要素数 N は 1 スレッドと変わらず、各スレッドが（スレッド数 P として） N/P 個の要素を読む。-disjunct の系列は、 N の連続メモリ領域を N/P 個ずつの連続領域に分け、各スレッドはそれぞれの部分領域の中だけでランダムアクセスをした場合で、-mixed の系列は各スレッドが N の中からランダムに N/P 個のアクセスをした場合となる。また、横軸は要素数の log をとったものである。図からわかる通り、要素数が増えるとアクセス時間は 1000 倍近くまで増大する。

また、図 2.2 は Opteron 1.4GHz, 4CPU の共有メモリ計算機において同じ実験を行った結果である。アクセス時間が上昇するデータ要素数やその上昇の仕方はアーキテクチャによって異なるが、やはり要素数が増えるに従って、数百倍のアクセス時間が必要になってしまうことがわかる。

このように、現在の計算機において、メモリアクセスのコストはデータの置いてある場所ごとに大きく異なり、アルゴリズムの実行性能は、どのようにデータを配置し、どの部分のメモリを使うか、によって大きく異なることになる。

2.2 多様な並列・分散計算環境

これまで計算機は、計算内部に包含されるさまざまな並列性を抽出し、それを利用して高速化を図ることを続けてきた。プログラム中には、人間が把握しやすい仕事のまとまりのレベル、すなわちタスクやプロセス、スレッドといわれる大きなレベルの並列性から、一つ一つの演算に依存性がないという細かいレベルの並列性まで、さまざまなレベルの並列計算可能性が存在している。

2.2.1 単一プロセッサ内の細粒度並列性

細粒度レベルの並列性を抽出するために、細粒度並列言語やその処理系が多く研究されてきた。例えば並列論理型言語やデータフロー言語は、分割できない最小の演算を一つの単位とし、その入力データの依存性情報を用いることで演算の順序を管理し、依存関係のない演算は全て論理的には並行して実行できるものとして、実際に演算器を複数用いて並列実行する、という動作を行う。これらの言語では、プログラマは並列性を明示的に意識しなくても、プログラムに存在する全ての並列性を自動的に抽出することができ、多数の計算リソースを用いてプログラムを大幅に高速化することができる。しかし、これらの言語は一般のプログラマが親しんでいる手続き型プログラミングモデルとは大きく異なるプログラミングモデルに立脚しており、意味論的にも文法的にも違いが大きいため、それほど一般化はしなかった。

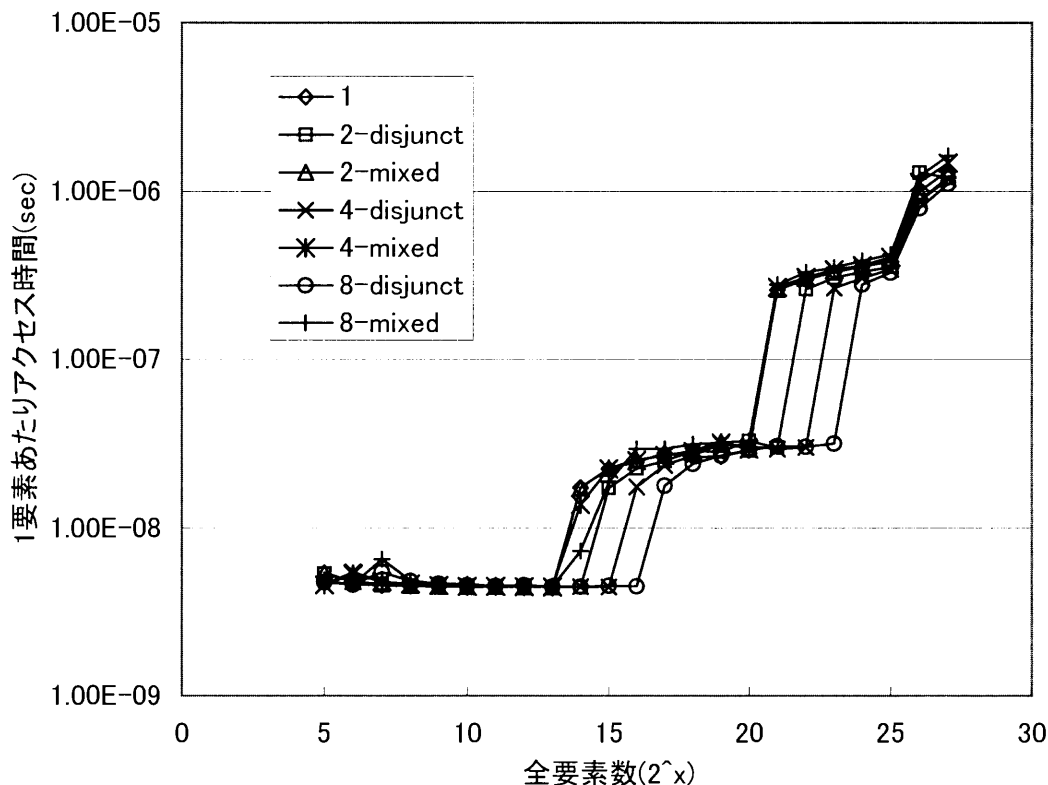


図 2.1: ランダムアクセステスト (SPARC)

しかし、細粒度レベルの並列性を抽出することが計算の高速化に役立つことは確実である。これをプログラム言語レベルではなくアーキテクチャレベルで実現するものとして、CPU 内部のスーパースケラ機構 [2] が研究され、発展してきた。

スーパースケラは、CPU 内部に複数の演算器を用意して、プログラムとして与えられた命令列をその複数の演算器にうまく割り振り、同時に複数の演算を行うことで高速化を図るための機構である。CPU の演算命令は、演算に必要なデータをメモリやレジスタから取得し、結果を指定されたメモリやレジスタに格納する、という動作を行うわけであるが、命令をいったん待ち行列に待避し、必要なデータがそろった命令から順次演算器に送るというアウトオブオーダー実行を行うことで、複数の演算が同時に実行可能になったときに、複数の演算器を用いて並列実行が可能になる、という手法である。もちろん、命令列中で前方にある命令の出力が後続命令の入力になる場合など、データには依存関係があるため、この依存関係を追跡し、データを出力する命令が実行を終えてから後続の命令を実行する、ということができなければならない。

スーパースケラは CPU の高速化のために積極的に取り入れられ、現在使用されている汎用プロセッサはほぼ確実にこの技術を用いている。同様の効果を得るための手法に VLIW (Very Long Instruction Word) [3] と呼ばれる機構がある。VLIW は、コンパイラによる静的解析やファームウェアレベルのコード変換プログラムを用いて、与えられた命令列中の並行性を抽出し、並列実行できる命令列を複数組み合わせ合わせた超長命令へと変換する。プロセッサ内の実行ユニットは一つの超長命令を単位時間で一気に実行するので、もとの命令列中の複数の命令が同時に実行されることになり、速度向上が図られる。スーパースケラと比較すると、超長命令にした段階で命令列の依存性

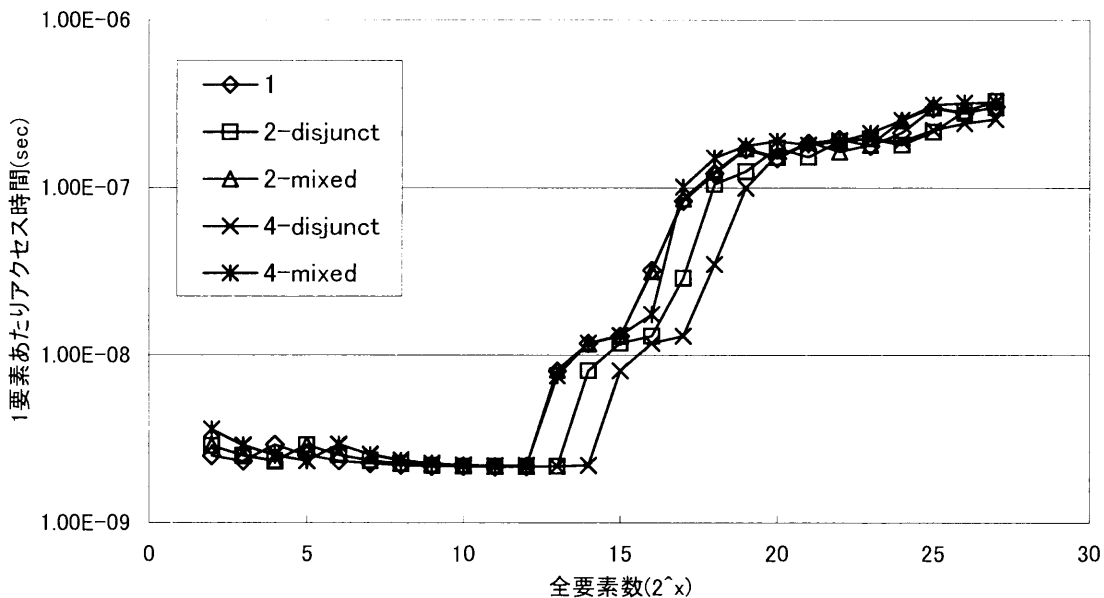


図 2.2: ランダムアクセステスト (Opteron)

の解析が終わっているため、依存性を追跡するハードウェアが必要ないというコスト的利点があるが、コンパイラやコード変換プログラムに対する要求が大きくなり、また静的に依存性が解析できないような並列性は抽出できないという欠点も持つ。

スーパースケーラは汎用プロセッサのアーキテクチャとして極めて広く普及したが、データフローマシンや VLIW の考え方も基本的には同種の並列性を追求したものであり、それぞれの技術が様々な複合されて現実のプロセッサに実装されている。例えば、近年の Intel のアーキテクチャは、互換性に必要な旧来の命令セット x86 命令を入力としているが、内部ではそれをより細かい μ OP と呼ばれる命令に分解し、さらに μ OP 列の中で依存性のないものを組み合わせて実行するという μ OP fusion という手法を用いており、これは VLIW の考え方を内部実装として取り入れたものとも言える。

このように、近年のプロセッサは内部的には複数の演算器を持っており、逐次実行用の命令列を実行させたとしても、内部的には並列性を抽出して並列実行しているのである。

2.2.2 共有メモリ型マルチプロセッサ

細粒度並列性よりやや大きいレベルの並列性に、スレッドレベル並列性がある。これは、プログラムの一つの命令列をスレッドと呼び、そのスレッドが複数同時に実行されるような場合である。スレッドレベル並列性は自動抽出の研究も行われてはいるが、現在のところ、プログラムレベルで人間が記述することで実現されている場合がほとんどである。

スレッドレベル並列性を用いてプログラムを高速化するために用いられるアーキテクチャは、何らかの意味でメモリを共有した複数のプロセッサが、同時に複数のスレッドを実行する、という形態を取る。よく使われているのが共有メモリ型マルチプロセッサアーキテクチャである。全てのプロセッサからメモリが均質に、対称的に見える場合を SMP (Symmetric Multiprocessing)、そう

ではなく近いメモリと遠いメモリの区別がある場合を NUMA (Non-Uniform Memory Access) と呼ぶ。

共有メモリ型アーキテクチャを用いたマルチスレッドプログラミングは、後述するプロセスレベルでの並列プログラムと比較すると、スレッド間でデータが簡単に共有でき、通信を明示的に書かなくてよい点や、スレッド間のデータ転送速度が大きい点が利点となる。しかし、通信を明示的に書かない故に、発見しにくいバグが入り込んでしまいがちであり、どちらがプログラミングスタイルとして簡単かは状況によって判断が分かれる。

近年、一般的な汎用プロセッサにおいても共有メモリ型アーキテクチャを意識した設計が行われ、2個から数個程度のプロセッサを持つマルチプロセッサのコンピュータが低廉化し、一般化してきている。現在では、サーバ用途に市販されているコンピュータはデュアルプロセッサ構成を取ることが多い。設置面積、周辺回路の消費電力、プロセッサの価格性能比を考慮すると、現在はデュアルプロセッサ程度が比較的良好なバランスの構成となっている、ということである。このような、バランスがよいと判断される構成は、技術の進展に伴って変化していくものではあるが、今後も複数プロセッサをまとめて一つのコンピュータとして扱うマルチプロセッサ構成は広く使われていくと考えられる。

また、一つのプロセッサ内に複数のプロセッサコアを収め、一つのプロセッサパッケージだけで共有メモリ型マルチプロセッサを構築できる、マルチコア技術も進展してきており、近年では一般的な汎用プロセッサにも導入され始めている。消費電力を抑えるために比較的性能の低いプロセッサを用いることの多いノートパソコンのようなコンピュータにおいてすら、2つのコアを持つデュアルコアプロセッサを搭載するようになりつつある。現在では2個から4個程度のコアを一つのパッケージに収める構成が採用されている場合が多いが、さらに多くのコアを集積することを目指して研究や製品開発が行われ続けている。

半導体の製造においては、「集積できるトランジスタの密度が3年で4倍に増大する」というムーアの法則 [5] が昔から広く語られている。これは経験則にすぎず特に根拠があるわけではないが、業界全体の技術目標的存在になっており、現在まではほぼこの法則を満たすペースで半導体の集積度が向上してきた。図 2.3 は Intel の市販汎用プロセッサに集積されているトランジスタ量の年代による変化を示しており、このムーアの法則をほぼ満たすようなペースでトランジスタ量が増えていることが読み取れる。

このように、半導体の集積密度は年々向上してきている。その結果利用可能になったより多くの半導体を使う際、より複雑な演算回路を作って速度向上を図る方法もあるが、用いるトランジスタの量の増加割合に対してプログラム実行速度が上がらないという問題、複雑な回路は熱密度が高い回路になりがちなため、廃熱の限界にぶつかってしまうという問題、さらに、設計から検証まであらゆる工程での作業が複雑になり、世代交代が激しい計算機業界で必要な素早い製品開発の妨げになってしまうという問題などがあり、次第にうまくいかないアプローチ方法となっている。そこで、一つのコアの設計はある程度の規模に抑えて、複数のコアを同じチップ上に配置して使う、マルチコアのアプローチ法が一般的になってきたのである。

マルチコアのアーキテクチャは、多くの場合共有メモリ型マルチプロセッサとして扱えるような構成を取る。Intel、AMD、IBM などの現行の汎用プロセッサは2個から4個程度のコアを集積し、その数程度のスレッドが並列実行できる性能を持っているが、今後より多くのコアを集積して、並列実行できるスレッドの数を数十以上にしていくという計画も研究されている。また、SUN の現行のプロセッサには8個のコアを集積したものもある。このプロセッサは、1個のコアが4スレッドを時分割で並行して扱えるように設計されているため、ユーザからは32スレッドが同時に実行されるように見える。このように、数十スレッドレベルの並列性を簡単に利用できるようになる時

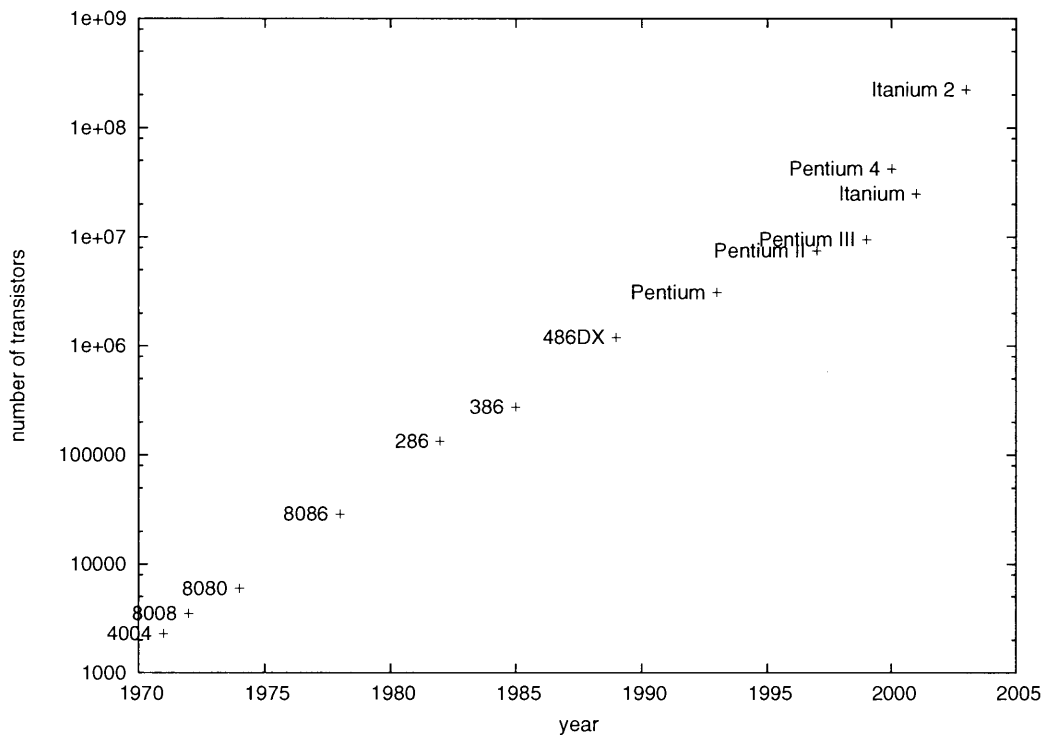


図 2.3: Intel の汎用プロセッサの規模とムーアの法則 [4]

代も、そう遠くないと考えられる。

また、多くのマルチコアは同一のコアを複数集めたものであるが、異種のコアを集めたヘテロジニアスな構成のマルチコアチップも開発されている。SONY、IBM、東芝の開発した Cell プロセッサは、機能の高いコアを一つと、ベクトル演算に特化したコアを 8 つ、一つのチップに集積している。汎用コアと機能特化したコアを組み合わせるという設計は他にも研究されており、共有メモリ型であっても不均一な構成となる場合も今後増えてくると思われる。なお、Cell の場合は共有メモリ型ではなく、それぞれのコアに独立したメモリを持ち、コア間を高速にメモリ転送できる仕組みを持つような構成になっている。

共有メモリ型は多くのアルゴリズムで高い性能を引き出しやすいが、大規模な並列計算機を作ることは難しい。大規模になればなるほど、それぞれのコアから見たメモリの一貫性を保つようにするための機構が複雑になり、メモリアクセスの性能が落ちてしまうからである。しかし、小規模な共有メモリ型並列計算環境は次第に身近なものとなりつつあり、今後、規模を拡大したり、構成を多様化したりしつつもさらに普及していくと考えられるのである。

2.2.3 クラスタ

パーソナルコンピュータとインターネットが急激に発達した結果、汎用プロセッサやネットワーク用機器は急激に低廉化、高機能化し続けている。これを利用し、一般的なプロセッサを乗せたパーソナルコンピュータ的な計算機を、一般的なネットワークでつないだ、クラスタと呼ばれる構成の並列計算環境が普及してきている。クラスタ構成の計算機は、独立したメモリを持つプロセッ

サが複数あり、その間を通信路がつかないものとしてユーザに認識される。

このような環境では、スレッドレベルよりはもう少し大きい粒度の並列性を用いていく必要がある。典型的には、独立したメモリ空間を持つプロセスが多数存在し、プロセスが互いに通信し合っ
て並列計算を進めていく、というプロセスレベルの並列度を利用した並列計算のスタイルが用いら
れることになる。

クラスタの利点は、そのスケーラビリティの高さ、コストパフォーマンスの良さにある。共有メ
モリ型の並列計算機は、前述のように規模が大きくなるにつれて急速にメモリ管理の複雑さが増
し、その結果価格も急速に高くなる。しかしクラスタ構成の場合、それぞれの計算機は独立したも
のであり、特に複雑な機構を追加する必要がない。結果、規模が増大しても性能あたりのコストが
それほど増加せず、高いスケーラビリティを実現できる。加えて、計算機もネットワーク機器も一
般に普及したものであるため、もともとコストパフォーマンスが高い部品を用いることができる。

クラスタは構成の自由度が高く、数台程度のパーソナルコンピュータを LAN で結合したような
ものから、数百台から数万台のプロセッサを結合したような大規模なものまで、様々な構成の計算
環境が作られている。数台程度の規模のクラスタは、部署単位でも十分構築が可能であり、身近な
並列環境として今後さらに普及していくと考えられる。

数万台のプロセッサを結合するようなクラスタは、スーパーコンピュータといわれる規模の計算環
境に相当するが、やはり一般的なネットワークを用いるよりは、より高速、広帯域なネットワーク
によって構築されることが多い。しかしそのような場合でも、規格化された部品を用いて高いコス
トパフォーマンスを維持したまま大規模化することが可能である。数年前まではスーパーコンピュ
ータは専用開発されたプロセッサを使い、専用開発されたネットワークを用いて結合されていた
が、近年のスーパーコンピュータはほとんどがこのクラスタ構成を取っている。

クラスタ構成は、計算能力だけでなくメモリやディスク等の I/O のスケーラビリティが高いこと
も大きな利点である。メモリもディスクも、それぞれのノードが独立して I/O できるものが接続
されているため、容量も I/O バンド幅もノード数を増やすに従って増加させることが可能である。

クラスタ構成の問題点のひとつは、アルゴリズムをうまく設計しないと並列化しても高速化が
図れない場合があることにある。通信の帯域幅も遅延も共有メモリ型に比べて性能が落ちるため、
それを隠蔽するようアルゴリズムを工夫する必要がある。また、通信を明示的にプログラムに記述
しなければならない点も、プログラミングを難しくする要素の一つとなる。多数の計算機を一度に
使うことになるため、管理などの手間が増えることも問題と言える。また、構成が自由であるが故
に、不均一な構成になる場合もあり負荷分散などの観点からプログラミングが難しくなることも多
い。導入時期によってプロセッサやネットワークが部分的に高機能化していたり、複数のクラスタ
を一体運用したりということもよく起きる状況であり、そのような不均一性はプログラマにとって
問題を難しくする要因になってしまう。

このように、クラスタ構成はコストパフォーマンスの高い並列計算環境として広く使われ始めて
おり、様々なスケール、様々な構成で今後もさらに普及していくと考えられるのである。

2.2.4 GRID と desktop GRID

GRID と呼ばれる構造は、複数のスーパーコンピュータをインターネットなどの広域ネットワーク
を用いて相互接続し、一体運用して大規模な並列環境を実現しようとするものである。もともと
は、電力網 (Power GRID) とのアナロジーで提唱された。ユーザが電気を使う際にはただ GRID
に接続すれば規格化された方法で電力が使え、背後の電力システム、発電所や変電所などの設備を
意識する必要がない、という電力網のように、計算力を提供する大規模なスーパーコンピュータを背

後に持つものの、ユーザはそれを意識することなく統一的に使うことができる、という環境を作ろうとしたものである。

スーパーコンピュータは並列計算機である場合が多いが、それをさらに複数結合することで、不均一で大きな並列計算環境を扱うことになり、ユーザに必要な労力も増えるが、大規模な計算能力を確保することができる強力なシステムでもある。ユーザの労力を軽減するような手法やシステムの研究が盛んであり、今後も発展していくものと考えられる。

また、少数の大型計算機を接続するのではなく、パーソナルコンピュータのような計算機を多数、インターネットなどを用いて接続して一体運用する方式もあり、desktop GRID と呼ばれている。この場合、それぞれの計算機が使われていない遊休時間を利用して並列計算をさせるような方式が多い。大量な計算力を極めて低いコストで使用可能になる反面、ネットワークの性能の低さ、各計算機の幅広い不均一性、利用可能資源が時間とともに大きく変化するダイナミックさや個々の計算機の信頼性の低さなど、並列計算に困難な側面も多い。しかし、SETI@home などいくつかの学術的プロジェクトでは大きな成功を収め、企業が保有する多数の計算機の遊休時間を有効利用する試みも広がるなど、今後も発展していくであろうことが予想され、その技術の研究も盛んである。

2.2.5 多様な並列計算環境の共通モデル化の必要

このように、プログラムには様々なレベルの並列性が存在し、それぞれの並列性を用いて計算を進める多種多様な並列計算環境が研究され、構築されてきている。また、それぞれの並列環境は単独で用いられるだけではなく、組み合わせられて使われることがほとんどである。例えば、クラスタに用いられる計算ノードはマルチコアなどの共有メモリ型並列計算機であることが多く、スレッドレベルとプロセスレベルの並列性をどちらも利用した並列計算を要求されることが多い。さらに、これらの並列環境は今後ますます発展し、大規模化しつつ普及していくことが予想されている。

このような状況で並列アルゴリズムを考える際、ひとつの標準的な並列計算環境を想定することは難しい。現実の並列環境はあまりに多種多様であるし、その構成のトレンドも技術の発展によって日々変化し続けていくものだからである。しかし、並列計算の発展のためには並列アルゴリズムを工夫することは欠かせない。特に近年、クラスタに見られるようにネットワークの性能が相対的に低くなっており、広域に広がった大規模な計算資源を使わなければならない場合には、並列アルゴリズムを研究して環境に対応できる計算手法を用いなければいくら計算資源をつぎ込んでも並列計算の性能が向上しない結果になってしまう。

よって、これらの並列環境を統一的に表現できるような計算環境モデルを構築し、その上で並列アルゴリズムの性能を議論していくことが必要になる。このような計算環境モデルを構築するためには、様々なレベルの並列環境を俯瞰し、何らかの意味で共通の性質を見いだして、計算に必要な時間を統一的に表現できるような枠組みを提案しなければならないのである。

2.3 解析可能な単純さ

計算量モデルにおいて必要なのは、もちろんその計算量予測の正確性である。実際の環境での計算時間をうまく表現できないようでは、計算量モデルを使う意味がない。

しかし、正確さの追求だけが目的ならば、計算量モデルをどんどん現実の計算機の構成に近づけ、現実の計算機構成を表現するためのパラメタを無制限に追加していけば、より現実に適した計算時間予測が可能になっていくはずである。この方針を最後まで推し進めると、結局現実の計算

機構成をそのままシミュレートすればよいことになってしまう。これでは特定の計算環境に特化した結果となってしまう、モデルを構築してアルゴリズムの普遍的な性能を評価しようという目的から逸脱してしまう。

そこで、計算量モデルにはもう一つ、できる限り単純でなければならない、という条件が課せられる。予測の正確さとモデルの単純さはトレードオフの関係にあり、その適切な組合せが求められることになる。

実用的な計算量モデルを目指すならば、あるアルゴリズムの計算量が手で解析的に求められる程度にモデルの表現が単純である必要がある。

第3章 関連研究とその問題点

3.1 (P)RAM

計算量モデルの中で、最も有名で広く使われてきたものは、Random Access Memory Model (RAM モデル)[6] である。このモデルは、計算機は演算器とメモリを持ち、すべてのメモリが演算器からある単位時間でアクセスできる、という構造を持っていると想定する。あるアルゴリズムにおいて、メモリ上のデータを単位時間で演算器に移動し、演算を行い、再びメモリに戻す、という手順の繰り返しを何回繰り返すのか、がそのアルゴリズムの計算量を与える、というモデル化がなされている。つまり、演算回数のみがアルゴリズムの計算量を規定する、と単純化したモデルである。

また、並列計算の計算量モデルとして、この拡張である Parallel Random Access Memory Model (PRAM モデル) がよく用いられてきた。PRAM モデルは、演算器が複数個、単一のメモリに接続されており、どの演算器もメモリのデータに対して任意に単位時間でアクセスできる、とするモデルである。このモデルでの計算量は、同時に実行できる演算を考慮しつつ、演算回数を数えあげるによって求められる。

このモデルの利点は、その単純性の高さにある。複雑なアルゴリズムであっても、そのアルゴリズムの演算回数を数えるという手法は容易なものであり、解析的に計算量を求めることが可能になるのである。アルゴリズム設計において、解析的に計算量が求められることはきわめて重要な利点となる。また、1990 年代初頭あたりまでの計算機の構造は RAM モデルと類似しており、演算器は接続されたメモリの全ての要素に、単位時間ないしはそれに近い時間でアクセスできることが多かった。そのため、RAM モデルでの計算量予測は現実の計算機の振る舞いとよく一致し、このモデルは有用なものとなった。

しかし、2.1 章で述べたように、近年の計算機アーキテクチャは、少量の速いメモリと大量の遅いメモリが存在するという階層構造を持ち、しかもそのメモリ階層は年々深くなり続けている。このため、モデルでの計算コスト見積りと、現実の計算機での実行結果との乖離が大きなものとなってしまう、RAM モデルはもはや適用できないのが現実である。

図 2.1 からわかる通り、要素数が増えるとアクセス時間は 1000 倍近くまで増大する。この振舞いは、とても RAM モデルでは表現できない。

3.2 HMM, UMH

RAM モデルとは異なり、メモリへのアクセスコストが一律ではないとするモデルもいくつか提案されてきた。Hierarchical Memory Model (HMM) [7] や Uniform Memory Hierarchy Model (UMH) [8] がそれである。HMM は、メモリのアクセスコストがアドレス x に対して $f(x)$ で与えられるというものであり、メモリの中でアドレスの小さい部分は CPU に近く、速くアクセスすることができるが、大量のメモリを使おうと思うと次第に CPU から遠い、遅いメモリを使うように

なる、ということを表示できるようにしたモデルである。UMH は、大きさやレイテンシが再帰的に決められた一連のメモリモジュール群というものを考え、それらがバスで相互に接続されたものがメモリである、とするモデルである。つまり、CPU の近くには小さくて速いメモリモジュールがあり、次第に大きくて遅いモジュールがつながっていく、という構造である。

HMM でのアクセスコスト関数 $f(x)$ については、 $f(x) = \log(x)$ を用いると多くのアーキテクチャにおける振舞いを説明しやすいと結論づけられてきた。この関数を適用することは、UMH において、大きさが定数倍になりアクセス速度が一定量だけ遅くなるようなメモリモジュールが連なっていると考えた場合と同様のモデル化であるといえる。また、HMM において、連続したブロックを対象とした通信が高速に行えるようなブロック転送の概念を加えたモデル [9] も提案されている。

これらのモデルはアルゴリズムの解析に成果を挙げてきたが、並列アルゴリズムは対象としていなかった。

3.3 LogP

並列計算のモデルでは、LogP[10] が良く知られている。これは、計算機間の通信路を、通信レイテンシ L 、通信発生時に生じるオーバーヘッド時間 o 、最小メッセージ間隔 (バンド幅の逆数に相当) g 、の 3 つのパラメタによって表現し、プロセッサ数 P の計算機がこの均一な通信路によって相互接続されている、とするモデルである。しかし、現在普及しつつある大規模な分散計算環境では、各計算機間の通信路が均一ではなく、遅延が大きいプロセッサとそうでないプロセッサ、すなわちプロセッサの「遠い近い」が計算時間に無視できない影響を与えてしまうため、このモデルも現実との乖離が大きくなりつつある。LogP の発展モデルとして、さらにパラメタを付け加え、詳細に通信路特性を記述しようとする試みもいくつか見られる。例えばメッセージ送受信時間を g のみではなく、メッセージ長に比例する部分と定数部分に分けた一次関数として表現した LogGP モデル [11] や、 o と g をメッセージ長 m の関数として表現する PLogP モデル [12]、さらにメッセージ長に応じて LogP と LogGP を使い分ける LogGPC[13] など、さまざまな拡張が提案されてきた。しかし、これらはすべて通信路のモデル化に留まっている。

3.4 メモリモデルと通信路モデルの組み合わせ

HMM などのメモリモデルと LogP などの通信路モデルを組み合わせた並列計算モデルがいくつか提案されてきた。

P-HMM[14] は HMM モデルがネットワークで接続されたものとして並列計算機をとらえている。すなわち、複数の CPU がネットワークで結合されており、そのそれぞれの CPU に HMM モデルのメモリがつながっている、というモデルである。CPU 間の通信は最もアクセスレイテンシの小さい、CPU に近い部分のメモリのみで可能である。ネットワークは密結合であり、レイテンシの違いは特に考慮されていない。また、HMM にブロック転送を加えたモデルで同様の並列化を行った P-BT[14] も提案されている。これらはいわば、CPU に最も近い部分のメモリが PRAM になっているようなモデルであると考えられ、通信路モデルが単純すぎて現実をよく反映していない。

また、UMH においては、複数の UMH モデルのメモリ階層が、ある大きさ以上のメモリブロック以上で共有されているような構造が提案されている [8]。例えば、SMP 構成の計算機の場合、複数の CPU のキャッシュに相当するメモリブロックはそれぞれの CPU で独立に存在しているが、メ

インメモリに相当するブロックからは全ての CPU 間で共有されているので、このブロックにより小さいメモリブロックが複数結合されている、とモデル化される。このブロックより大きいメモリブロック群は全て共有される。クラスタ構成の分散計算機の場合は、メインメモリより大きいブロックで共有が起きていると考える。これはある大きさ以上のメモリブロックが PRAM になっていると考えてもよく、やはり通信路が単純すぎるといえる。

メモリモデルとして HMM や UMH を、ネットワークモデルとして LogP を採用し、それらを組み合わせた LogP-HMM や LogP-UMH というモデルも提案されている [15]。PRAM モデルを仮定するよりは現実を表現できるようになった反面、モデルが複雑になり解析が難しくなったという欠点もある。

3.5 Coarse Grained Multicomputers, Bulk-synchronous Parallel

プログラミングモデルをある程度限定し、計算と通信のコストの両者による影響を考えようとした、Coarse Grained Multicomputers (CGM) [16] や、Bulk-synchronous Parallel (BSP) [17] といった並列計算モデルもある。これらは、ある程度の量のメモリをローカルに持つプロセッサが、何らかのトポロジを持つネットワークで相互に接続されている、と計算環境のモデル化を行ない、その上で「ローカルな計算フェーズに必要なコスト」と「通信フェーズに必要なコスト」を別々に考え、加算することによって全体の計算量を把握しようというモデルである。それぞれのフェーズは計算機全体で同期されており、計算と通信はオーバーラップされて実行される。

また、BSP を基本に、通信路の振舞いや計算機性能が場所によって異なるような計算機環境をモデル化する Heterogeneous BSP [18] も提案されている。計算フェーズ、通信フェーズが、並列計算全体で同期している BSP モデルであるため、通信が最も遅いところに揃うことで通信路の構成に起因する振舞いをとらえようとしている。ただし、このモデルでの通信時間はプロセッサ入口のバンド幅と送受信するデータ量によってのみ決定されるというシンプルなものであり、通信路の構造、通信の局所性などは考慮していない。

3.6 既存モデルの問題点

これらのモデルのように、単体計算機内の計算と計算機間の通信とを別々に考慮する場合、どうしてもモデルの組合せによる複雑さが生じる。また、単体の CPU 内のメモリ階層による影響を表現するモデルと、CPU 間の通信路を精密に表現するような別々のモデルを組み合わせるという複雑な手順を取って、ある特定の計算環境を精度よく表現することが可能になったとしても、単体の CPU の構成が違ふ、あるいはネットワークトポロジが違ふ環境を考えようとすると、モデルの再設定、及び計算コストの評価のやり直しが必要となってしまう。計算機構成のトレンドは年々変化しており、トレンドが変化するたびにモデルを変更しなければならないのでは、アルゴリズムの設計に対する指針としては使いにくい。

さらに、非常に多数の計算機が結合した環境で、すべての通信路を独立に扱うことはもはや困難になる場合などもある。このように、並列計算環境を「複数の計算機がネットワークでつながったもの」としてとらえることは、モデルの複雑化を招き、またこれから大いに発展すると思われる大規模分散環境に対応できないアプローチであると考えられる。

第4章 アクセス計算量モデルの提案

4.1 提案する計算モデルの目標

これまでの計算量理論は、演算にかかる時間の総和、もしくは、演算と通信にかかる時間の総和であるという立場に立っていた。しかし、「計算のコストとは、演算にかかる時間にあるのではなく、演算に必要なデータを取得し、演算結果を格納するという通信過程こそ存在する」という基本理念の方が、より正確に現状を把握しており、より簡潔に計算量の本質を表現しているのではないかと考えた。メモリ階層はデータへのアクセス遅延時間の違いが生み出すものであり、ネットワークトポロジの違いを考慮すべきなのも、他の計算機にあるデータへのアクセス遅延時間の違いが無視できないからである。

この基本理念をもとに、「計算機の内側と外側の区別、単体計算機内の計算とネットワーク上の通信の区別をなくし、それらを統合した1つの仮想計算機として表現したモデル」によって、新しい計算量モデルを構築することを考えた。このモデルではすべてのメモリ間に距離 x が定義できるものとし、それぞれのメモリ間の通信コストが「距離による単純な関数 $f(x)$ 」で表されるものと仮定する。この $f(x)$ によって、任意の単体計算機内部のメモリ階層、任意のトポロジによるネットワークのふるまいを、統一的にかつ簡潔に表現し得るのではないかと、というのが我々の主張である。これにより、アルゴリズム設計の際、計算コスト評価の労力が著しく緩和される。

このモデルは、特定の計算機環境における計算時間を極めて精密に予測するというよりは、多くの並列計算機環境で共通した振舞いをよく予測できるような、一般性の高い指標を目指している。図 2.2 は、図 2.1 と同じ実験を Opteron 1.4GHz, 4 CPU の計算機で実行した結果である。メモリアクセス時間の増大はキャッシュなどの構造に依っているもので、構造の異なる2つの環境では振舞いもかなり違っているが、全般的な増大の仕方に共通点もある。さらに、この実験はローカルメモリの大きさで制限されているが、ネットワークでつながった別の計算機のメモリを使ってランダムアクセス実験を行えば、このグラフの先もやはりアクセス時間は増大していくと予想される。その際も、計算機のネットワーク性能やネットワークトポロジの違いによって、細かい振舞いは環境によって変化する。しかし、環境に依存しない増大の仕方を、ローカルメモリからネットワーク越しのメモリの範囲まで全て、 $f(x)$ という単純な関数で表現するよう割り切ってしまう、というのがこのモデルの基本的な方針である。

Thomas rauber らの Locality measure[19] は、メモリアクセス履歴を用いた簡単な指標で、並列アルゴリズムの実行環境に依存しない実行性能を表現しようとするものであるが、本モデルも同じ方向を指向している。ただし、Locality measure がメモリアクセスの履歴から間接的な定性的要素を取り出しているのに対し、本モデルはより直接的に、計算機の構造を意識した指標を作ろうとしている。

このような並列計算コストのモデルを、「アクセス計算量」と名付けることにした。以下、このアクセス計算量の概念と、仮想機械のアーキテクチャの設計、仮想機械語の設計を示す。また、いくつかの並列アルゴリズムへの計算量解析への適用を通してモデルの妥当性と適用可能性について示す。

4.2 モデル概要

アクセス計算量とは、以下のようなモデルに従う計算量である。

- メモリはある密度で連続的に（現在のところ 1 次元で）分布したものであるとする。
- 計算する資源はメモリ上の至る所に存在している。メモリ上には「計算実体」と呼ばれる、計算を行なう「もの」が存在し、メモリ上の好きな地点で計算を行なうことができる。
- プログラムの実行時間はメモリアクセスの時間が支配的であり、演算そのものにかかる時間は無視できるとする。ただし、演算のためには必ずメモリ上からデータを持ってくる必要があるので、見掛け上無限に速い演算ができるわけではない。
- メモリへのアクセスには「格納された場所」から「計算する場所」までの「距離」に従ったコストがかかる。 x だけ離れた場所のメモリにアクセスするときは、指令からある時間 $f(x)$ だけたった後にアクセスが行われる。 x の点でのアクセスにはある一定の時間 $l, l > 0$ がかかる。さらに、読んだデータを送り返す時間または ack を返す時間が $f(x)$ だけかかるため、メモリアクセス命令全体では $2f(x) + l$ だけの時間がかかる。

ack を返さない書き込み命令はない。

- $f(x)$ は単調増加関数であり、 $f(0) = 0$ が成り立つ。また、 $0 < x < y$ としたとき $f(y) < f(x) + f(y - x)$ が成り立つ、すなわち上に凸な関数である。これはつまり、どこかで通信を中継することで速いアクセスが可能になるということはない、ということを示している。
- メモリアクセスそのものの衝突は考えない。メモリアクセスの際に使われる通信路が混雑することで、アクセス競合の振る舞いを捉えようとする。これは、データを近傍に並べるのではなく、適宜分散させて計算した方が速い場合があることを反映させようという狙いである。
- 計算実体は「プログラムカウンタ」と「実行場所」のみを持っており、PC を進めつつ、メモリ中で実行場所を移動させて計算を行う。計算実体自身にレジスタのようなメモリはない。
- 命令を取ってくる時のコストは考えない。
- 計算の実行場所が移るときもメモリアクセスと同じ時間コスト $f(x) + l$ がかかる。メモリアクセスを計算実体が追い越すようなことはできない。
- 通信路にはある有限の容量があり、容量を越えた通信についてはペナルティが追加された時間コストがかかるものとする。これにより、計算主体自身は任意の並列性で存在できたとしても、計算全体は通信路の混雑により速度向上に制限がかかる。

今、1CPU しかない普通の計算機をアクセス計算量モデルで表現すると、1 次元のメモリ上に計算実体が 1 つだけ存在する状態となる。計算実体の直近のメモリは、アクセスする際の距離が小さい、読み書きのレイテンシが最も短い場所であり、現実の計算機ではレジスタにあたる。その隣には、少し距離が大きくなった L1 キャッシュに相当するメモリがあり、以後段々遠くに L2 キャッシュ、メインメモリが置かれていると考えられる。現実の計算機ではレジスタ、L1、L2 キャッシュ、メインメモリとレイテンシが増大するに従って容量は大きくなる。このレイテンシと容量に合わせて $f(x)$ を定義すれば、現実の計算機のアクセスコストを模倣することができる。一般的には、レイテンシと容量の変化の度合は容量の方が累乗的に増大する傾向にあり、HMM の研究によると

$f(x) = \log(x)$ 程度が適切だと言われている。もちろん、近年はレイテンシの増大が激しくなっているため、 $f(x)$ をどのように設定したらよいかはまだ研究の余地がある。

現実の計算機では、メインメモリにアクセスするとアクセス対象の近傍のメモリがキャッシュメモリに移され、以後のアクセスが高速になるが、アクセス計算量モデルではキャッシュはアルゴリズム設計者が明示的に使用しなければならない。つまり、このモデルの仮想機械が持つブロック転送の機能を用いて遠いメモリから近くのメモリへとデータを移し、近くのメモリを使って計算を行なう、という動作を明示的に行なわせる必要がある。ただ、このモデルでは計算実体が自由に場所を移せるため、データを取ってくる代わりに、計算実体自身をデータの近くへと移すことでもキャッシュと同じ効果が得られる。

次に、SMP マシンをこのモデルで表現すると、メモリ上にプロセッサ数だけの計算実体が置かれた状態になる。図 2.1 と図 2.2 の実験を例に、SMP マシンの動作がこのモデルでどのように表されるかを説明する。

図 2.1、図 2.2 の実験は、連続したメモリ領域の要素が 1 本のリストとしてつながっており、リストをたどって全要素を読むことで全メモリ領域をランダムに 1 回ずつ読む、ということを繰り返すものである。系列 1 は 1 スレッドで、2-から始まる系列は 2 スレッドで、のように複数スレッドで実験を行ない、複数スレッドでも全要素数 N は 1 スレッドと変わらず、各スレッドが (スレッド数 P として) N/P 個の要素を読む。-disjunct の系列は、 N の連続メモリ領域を N/P 個ずつの連続領域に分け、各スレッドはそれぞれの部分領域の中だけでランダムアクセスをした場合で、-mixed の系列は各スレッドが N の中からランダムに N/P 個のアクセスをした場合となる。

1 スレッドの場合、わざわざレイテンシの大きいメモリを使う必要はないので、計算実体は N の要素のどこかに居る。アクセス対象メモリまでの平均距離は $N/2$ である。-disjunct の場合、計算実体は N/P の要素のどこかに居るのが自然だろう。この時、計算実体からアクセス対象メモリまでの平均距離は $N/2P$ である。それ以外は 1 スレッド実行の時と違いがない。そのため、図を見るとわかる通り、-disjunct のアクセス時間は要素数 $1/P$ の 1 スレッド実行時のグラフときれいに同じ形をしている。一方、-mixed の場合、計算実体の位置は-disjunct と同じでも、アクセス対象までの平均距離は 1 スレッド実行と同じ $N/2$ となる。そのため、-mixed のグラフは 1 スレッド実行のグラフときれいに重なる。このように、このモデルは並列実行の様子を比較的に直接的な形で把握できることを目指している。

ところで、メモリが 1 次元であることによる現実との乖離ももちろん存在する。SMP のモデル化で、3 個並んだ計算実体が互いに相手の近くのメモリを読む時端の 2 つの間の方がレイテンシが大きくなることになるが、これは現実計算機では考えにくい。クラスタのような分散環境でも同様である。また、分散環境の場合、ネットワークトポロジによっては、計算機間に複数の通信経路がある場合や、通信方向によってレイテンシに差がある場合などもありうる。このような現実を、1 次元のメモリモデルでは扱うことができない。しかし、例えばメモリと通信路を 2 次元以上に拡張し、2 地点間の距離と通信経路を適切に定めることができれば、このような現実も扱うことが可能になる。ただし、多次元の拡張を行なうと距離の定義が難しく、解析が困難になると予測される。多次元にすることで「迂回する」通信路を作ることができるのも、通信路の混雑の解析を困難にする。そもそも、一般的なネットワークは何次元あれば表現できるのかもよくわからない。以上のような理由で、現状のアクセス計算量モデルでは、メモリと通信路は 1 次元であると仮定する。多次元への拡張は今後の課題としたい。

4.3 仮想機械

これまで述べてきた計算モデルは、次のような命令セットを持つ仮想機械であると定義できる。以下の説明で、`<input>`、`<output>` は即値及びダイレクト、インダイレクトのアドレッシングモードをとるメモリ内容である。なお、アドレスは現在の計算実体が存在している場所からの相対位置を指定する。

各々の命令は現在の実行場所で行われ、実行後 PC を 1 つ進める。

演算・メモリ

- (`<operation-name>` `<input1>` `<input2>` `<output>`)
 $input_1 \cdot input_2 \rightarrow output$ という演算を行なう。
`<operation-name>` = add, sub, 等の基本演算
- (copy `<src>` `<dest>`)
 $src \rightarrow dest$ というメモリ内容コピーを行なう。メモリ上に定数を書き込みたい時もこの命令を使う (`src` が即値になる)。ブロック転送も可能。

実行場所の制御

- (next_place `<next-place>`)
次の実行場所を指示。`<next-place>` は (place `<diff>`) であり、現在の場所から相対的に記述。

制御

- (jump `<program-point>`)
無条件ジャンプ。`<program-point>` の命令を実行。
- (branch `<input>` `<program-point>`)
`<input>` の場所のメモリの値によって分岐。
- (fork `<program-point>`)
別の計算実体を作成し、`<program-point>` の命令を、現在の実行場所で行わせる。自らは次の命令を現在の実行場所で行う。
- (compare_and_swap `<src>` `<org>` `<update>` `<pp>`)
`<src>` 場所のメモリを読み、`<org-value>` と一致したら `<update-value>` で置き換え、`<program-point>` の計算を続ける。`<org-value>` と一致しなかったら `<org-value>` を `<src>` 場所の内容で置き換え、not taken で下に抜ける。
- (vanish)
計算実体が消える。

これらの命令セットを用いてアルゴリズムを記述することで、全ての計算に必要なデータの配置とその間の通信を明記することになり、アクセス計算量モデルに基づいた計算量を求めることができる。

4.4 仮想機械シミュレータの設計と構築

この仮想機械語プログラムを解釈実行し、その計算の振舞いを把握できるシミュレータを設計[20]、実装した。これを用い、複雑な並列アルゴリズムに対しても、その振舞いを把握することが可能になった。

仮想機械シミュレータは、与えられた仮想機械語の命令列を読み込み、順に解釈実行していく。「現在の実行場所」と「命令列中のどの命令を実行しているか」のみを記憶している計算実体を複数シミュレートし、それぞれの計算実体が命令の解釈実行を行う。各命令の実行は、(1) オペランドのフェッチ、(2) 計算、(3) 結果の格納、の3ステージを繰り返すことで行われる。命令は現在の実行場所にて実行され、(1) と (3) におけるメモリアクセスでは、実行場所から目的のメモリ上の場所までパケットが移動し、データの取得/格納が行なわれる様子をシミュレートする。パケットの衝突や負荷の集中などは、5章で述べる通信路モデルに従ってシミュレートされ、各命令の実行時間や通信路の振る舞いを知ることができる。

第5章 通信路モデル

5.1 通信衝突モデル

上述のモデルでは、並列実行時の計算の振舞いは通信路によって大きく規定される。計算と同様、通信路の混雑の局所性も計算量を見積もる際に大きな影響を与える、という立場から、我々のモデルでは全ての通信をパケットととらえ、パケットの動きを追跡することで通信路の振舞いを把握することにした。以下に、この「複層の通信路モデル」の概要を示す。

- メモリが1次元であるモデルの場合、通信路は正方向行き/負方向行きの2つが独立して存在する。正方向と負方向に送られるパケット相互は衝突しない。
- 通信路は層状に積み重なっている。通信の衝突は各層内でのみ発生する。各層の通信速度は、底の方ほど遅い。通信は一番底の通信路からスタートし、一定時間毎に隣の層へと移動していく。この時、世界全体で共有されたパルスが存在すると仮定している。一定時間毎に世界全体でパルスが発生し、そのパルスに従って通信路が一斉に状態を変える。
- 各通信路の速度は $f(x)$ によって規定される。例えば $f(x) = O(\log(x))$ の場合、高さ h の通信層の速度を $v(h)$ 、 k を1より大きい適切な定数として、 $v(h) = kv(h-1)$ とすればよい。(図 5.1 参照)
- 通信は、現在使っている通信層の速度 $v(h)$ 、データ長 n に対して、 $(n+1)v(h)$ に相当する広さの領域を占有する。1はパケットが持つ定数のコストを示しており、 n が大きいパケットを用いるほど通信路の利用効率が上がることを表現している。
- パケットは任意のデータ長を一度に送ることができる。複数ワードを送信する際は、送信するデータが置かれた場所を次々にパケットが訪問し、次第に長いパケットに成長していく。データを受信する側でも同様に次第にパケットが短くなりながらメモリ上に書き込みが行なわれる。(図 5.2 参照)
- 次の単位時間に占有すべき領域が空いていない時、パケットは空きを待つ。空きが足りない場合はできる限り進もうとする。2つのパケットが衝突している時は、底の通信路へ降りていく(パケットが減る方向の)パケットを優先させる。(図 5.3 参照)

この通信路モデルは、いわばキャッシュメモリからメインメモリへと連なるメモリバス、さらに Local Area Network、Wide Area Network へという階層的な物理的通信路のモデル化を狙ったものである。一番底の層が CPU 内部にあるレジスタやキャッシュとのバス、次の層がメインメモリのためのバスで、その上に順に LAN、WAN と連なっている、という意識である。例えば、LAN 経由で隣の計算機と通信をする際、データはメモリからメモリバス、LAN を通り、相手の計算機のメモリバスへと移行する。LAN の層でパケットの衝突が起きた時、複数の計算機の持つ広い範囲のメモリ領域の上の通信層が影響を受けていることになる。モデル上で、上に位置する層の方が

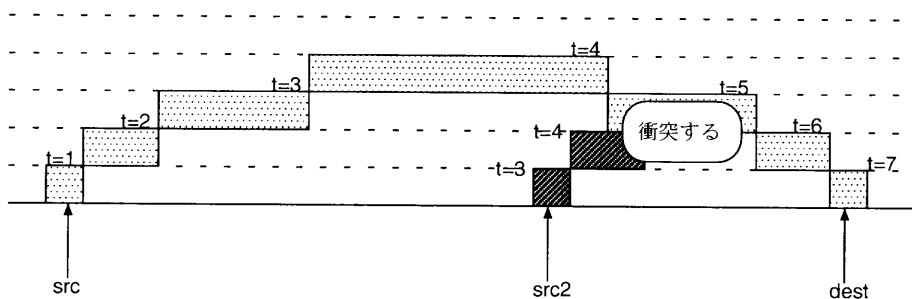


図 5.1: 通信路と衝突

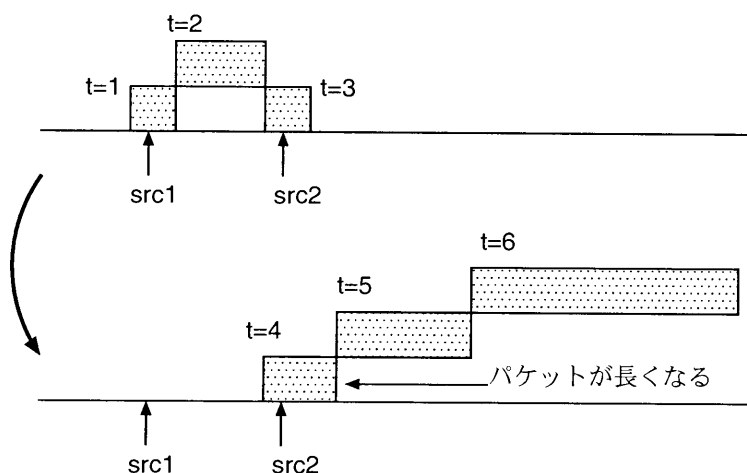


図 5.2: 2ワードを1パケットにまとめて送信する場合

速度が速いが1つのパケットがより広範囲に影響を及ぼすようになる、という設定はこのような状況表現しようとしたものである。ただし、LANの通信層が輻輳を起こしている時でも、単体計算機内の（キャッシュ）メモリアクセスは影響を受けない。この現実が、モデル上では、上の層の通信衝突が下の層には影響を与えないという設定で表現されている。このような層構造は、CPU内部のメモリバスでも、大規模なネットワーク環境でも、共通して見られる構造であると考えている。

5.2 通信負荷累積モデル

これまでに、アクセス計算モデルに基づく仮想機械を設計(4.3章)し、このモデルでの計算の様子を把握できるシミュレータを構築した。また、この仮想機械上でのプログラミング手法を提案し、高水準言語の設計と実装を行い[21]、いくつかのアルゴリズムをシミュレータ上で実行することで計算モデルの妥当性や有効性を検討してきた。その結果、複層の通信路モデルは現実のシステムを確かに良く表現するが、解析的に計算量を求めるにはやや複雑すぎることもわかった。そこで、通信路の混雑をより単純にモデル化した「通信負荷累積モデル」を作成し、解析的手法による計算量の見積もりを可能にしようと試みた。

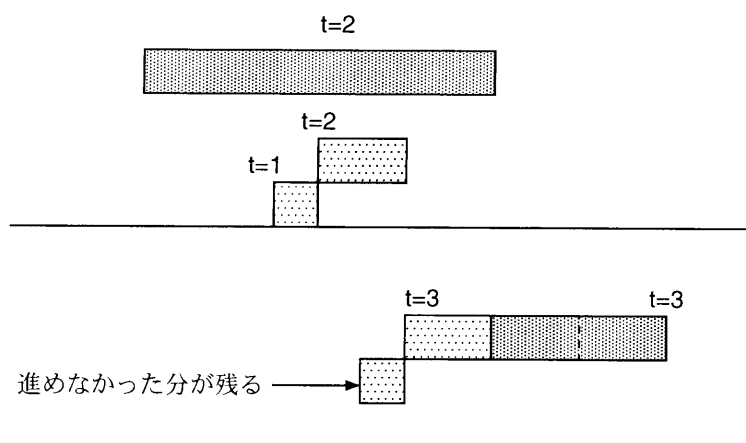


図 5.3: 通信路がふさがっている場合

通信負荷累積モデルでも、通信を担うパケットの各時点での速度、及びそのパケットが影響を与える範囲は複層の通信路モデルと同じである。結果、距離 x だけ離れた 2 地点間の通信は少なくとも $f(x)$ だけの時間がかかる。2 つのモデルは、通信の集中による通信路の混雑の表現についてのみ異なっている。

パケットは現在占めている通信路の領域にある負荷を与える。複数のパケットが同じ領域に存在する時、その領域にはそれぞれの負荷を加算した負荷が与えられる。通信路には決まった容量が定められており、ある領域の負荷が容量を越えた時、その時刻の通信路全体に渡って混雑による通信遅延が起きるものとする。

パケットは、速度に応じて負荷を与える範囲が変わる。単位量の通信パケットは全体としてはある一定の負荷を持ち、ある時刻での地点 x での負荷を $l(x)$ とすると、

$$\int l(x)dx = constant \quad (5.1)$$

が成り立つ。つまり、速度が上がると広い範囲に負荷を与えるが、その場合ある地点に与える負荷の値は小さくなる。このような $l(x)$ の形はさまざまなものが考えられ得るが、以降の議論では、ある時刻のパケットの速度を v とした時、 v の範囲に均一に $1/v$ の負荷が与えられる、という単純なモデルを採用する。

図 5.4 に、1 つのパケットのみが存在する時各時刻において通信路に与えられる負荷を示す。パケットの速度、存在するメモリ上の位置は図 5.1 と同様に変化しているが、通信路に層は存在せず、1 つの通信路に与える負荷 (load) が速度に応じて変化している。

図 5.5 は、2 つのパケットが衝突する様子を示している。 $t = 0$ で 1 つ目、 $t = 2$ で 2 つ目のパケットが発生し、ともに右に向かって進んでいる。 $t = 2, 3$ では 2 つのパケットが通信路の同じ領域に存在するため、2 つの負荷を加算したものがその時刻の通信路負荷となる。 $t = 2$ で、2 つの負荷の合計がある定められた通信路容量 *threshold* を越えるため、この時刻の通信路中全ての通信はペナルティを受けて遅くなる。ペナルティは、負荷 l が容量 T を越えた時、通信は通常の l/T 倍の時間がかかるというモデルを想定している。

このような通信路は、「複層になった通信路モデル」と比較して、現実の通信路の特性をいくつか無視している。例えば、複数のパケットの負荷を単純に加算してある地点の通信路負荷を求めているため、LAN 内の通信が輻輳を起こすとローカルメモリの読み書きが遅くなる、というモデル

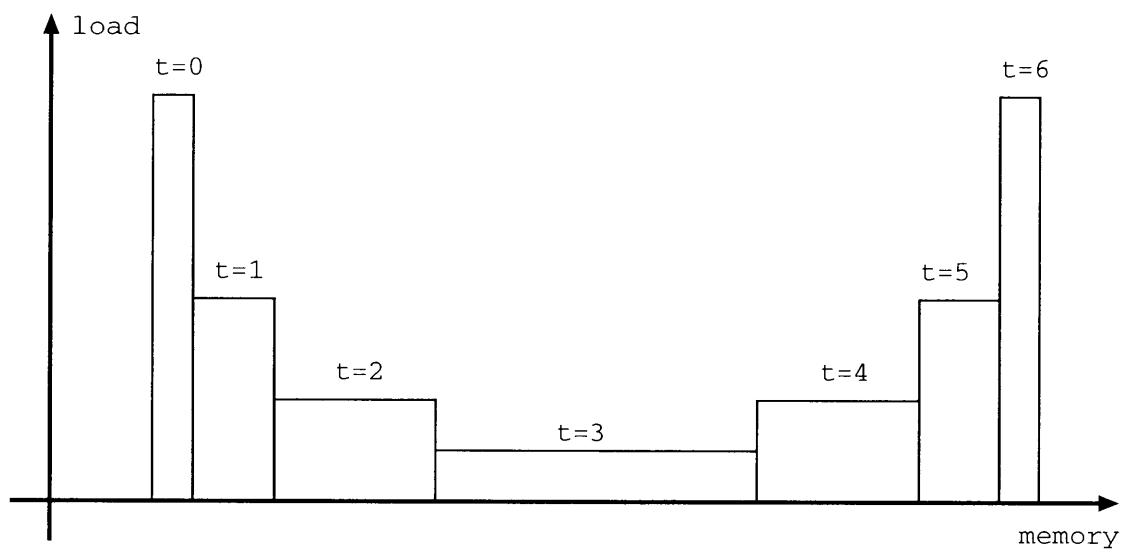


図 5.4: 通信負荷累積モデルでの 1 つのバケットの負荷

であると言える。しかし、このような単純化によって、複数の通信が同時に行なわれる時の通信路の振舞いを簡単に表現でき、解析的手法で計算量を求めることが可能になると期待される。

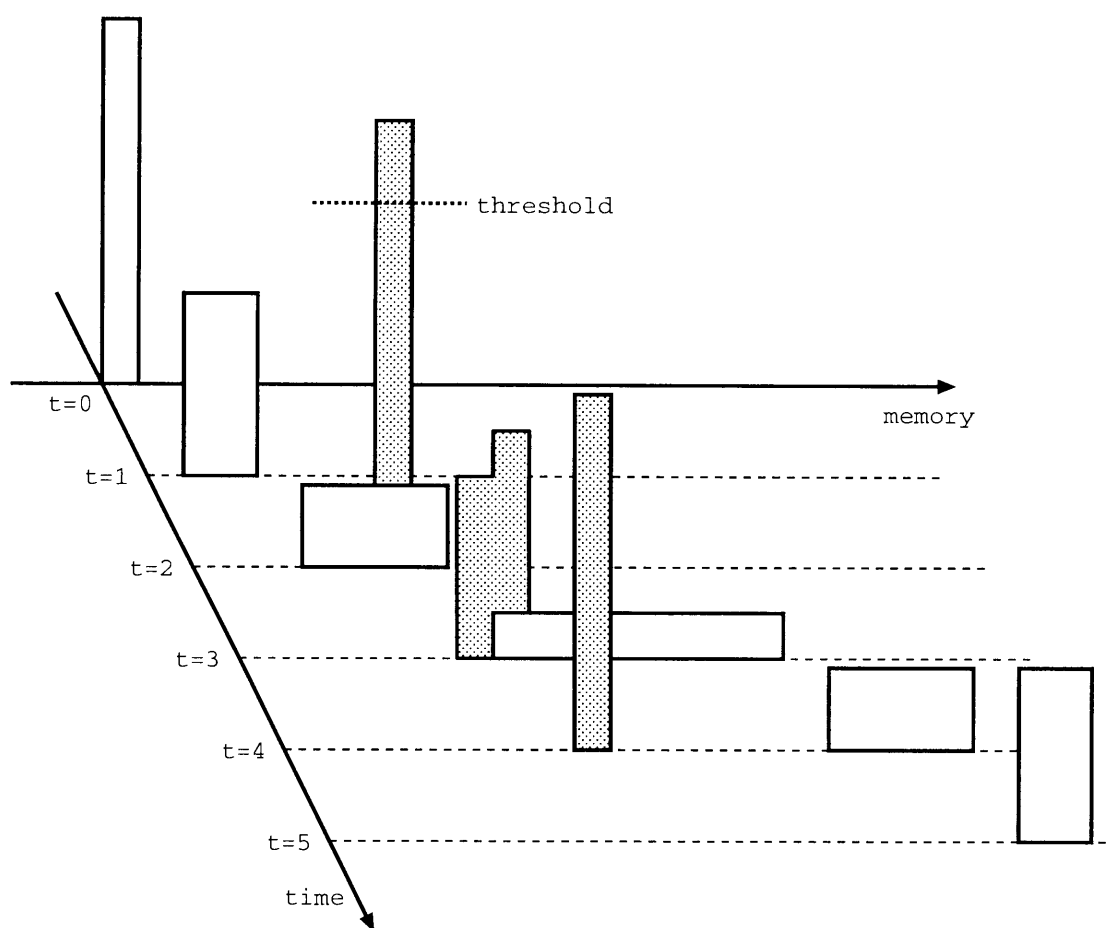


図 5.5: 通信負荷累積モデルでの複数のパケットの負荷

第6章 並列アルゴリズムの解析

通信負荷累積モデルを用いると、通信路の局所的な混雑を意識しながらも、通信による影響を解析的に求められる程度にはモデルを単純化することができると考えられる。

そこで、このモデルがアルゴリズム解析に十分適用できる程度に単純であることを示すために、よく知られた並列アルゴリズムである

- bitonic sort
- merge sort
- FFT

の3種類のアルゴリズムについて、アクセス計算量モデルを用いて計算量を解析的に求めることを試みた。

また、その解析結果が実際の計算環境の実行状況をどの程度正確に反映しているかについても、考察する必要がある。そこで、いくつかの共有メモリ型計算機を用いてマルチスレッド型の並列プログラムを実行し、それぞれのアルゴリズムの実計算機での振る舞いを測定して、求められた計算量予測がどの程度実際と合致するかを評価することにした。

6.1 diremption factor

実機での並列プログラムの実行時間と理論予測の実行時間の一致の度合いを測るために、本論文では diremption factor という尺度を考えることにした。

diremption factor は、アルゴリズムの入力データサイズが2倍になるときに理論予測の実行時間の増加する割合と、実際の実行時間の増加する割合の比を取ったものである。つまり、入力データサイズを 2^k としたとき、実測された実行時間を m_k 、理論予測された実行時間を p_k として、diremption factor は

$$(m_k/m_{k-1})/(p_k/p_{k-1})$$

と表されるものであるとする。

もし、理論予測が実測時間の増加の仕方と一致していれば、diremption factor の値は1へと収束する。

以下の評価では、この diremption factor を用いていくことにする。

6.2 bitonic sort

bitonic sort アルゴリズムは、sorting network の一種である。規則的な比較演算を全データに対して繰り返すアルゴリズムであり、比較を行なう要素の対がデータに関わらず固定されているため、並列計算の適用も比較的容易である。

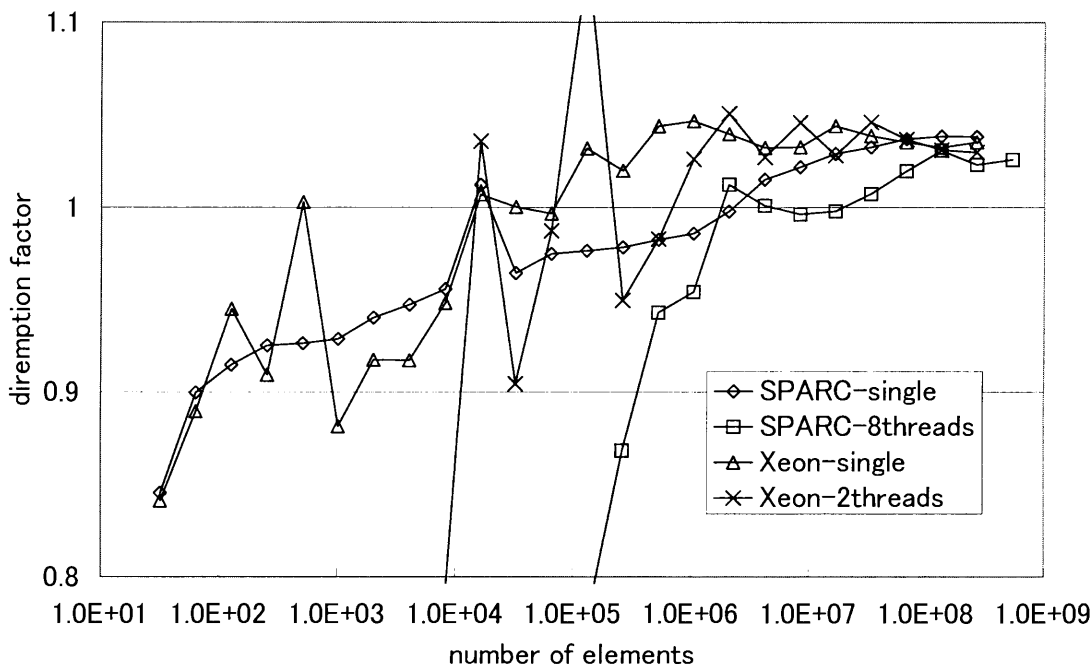


図 6.1: bitonic sort における実計算機と RAM モデルの計算量比較

n 要素の bitonic sort は $(n/2) \times (\log n(\log n + 1)/2)$ 回の比較演算を含むため、RAM モデルにおいて計算量は $O(n(\log n)^2)$ と表される。また、並列計算の際には $n/2$ 回の比較操作を並列に実行できるため、並列度 P のとき $O(n(\log n)^2/P)$ 、最高で $O((\log n)^2)$ の計算量となる。

実計算機上での実験結果を図 6.1 に示す。計算環境は

- Ultra SPARC III Cu 1.2GHz, 8 CPU
- Pentium4 Xeon 2.4GHz, 2 CPU

である。いずれも pthread ライブラリを用い、共有メモリを使って並列化を行なっている。4つのグラフ系列は、2つのアーキテクチャそれぞれで single thread 実行時と 8(2)thread 実行時の2通りの計算時間を示している。横軸はデータ数 n の対数表示、縦軸は第 6.1 章で示した diremption factor を示している。

図 6.1 に示される通り、データの量が大きくなるにつれて diremption factor が 1 より大きな値に収束している。これはつまり、実測された実行時間の方が、理論予測より急速に増大していていることを示している。つまり、RAM による単純なモデル化では把握できないメモリ階層などの要素が現実の計算時間を押し上げており、RAM モデルの計算量と現実との乖離が起きているのである。

次に、「アクセス計算量モデル」による解析を行なう。

アクセス計算量モデルでは、データの配置と計算実体の位置が計算量を決定する。実験に用いたプログラムでは、データは最初に連続領域に配置され、そのまま配置を変更することなく同じ連続領域でソートされる。このとき、 n 要素 bitonic sort に対して一度呼ばれる n 要素 bitonic merge は、 $i = 0$ から $n/2 - 1$ まで i を順に変えつつ、 i 番目と $i + n/2$ 番目要素の比較を行なう。

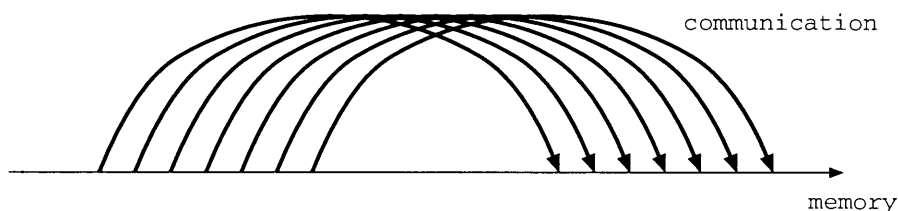


図 6.2: 連続した点からの同一方向への一斉通信

これをアクセス計算量モデルで解析する時、データ配置は元プログラムの配置に従えばよいが、計算実体の場所は通常のプログラムには概念が存在しないため、解析者が適切に定める必要がある。今回の場合、要素に順番にアクセスするため、キャッシュメモリにデータが乗っていることが多いと考えられる。解析をより正確にするためには、次に比較に使う2つの要素付近のメモリを計算実体そばにブロック転送し、キャッシュの効果を再現する方がよいが、今回は比較に使う片方の要素の上に計算実体が移動することで、簡易的にキャッシュの効果を再現することにした。この場合、もう片方の要素はやや遠い位置に存在することになるが、アクセスの際の距離が常に一定になるため、それほどひどいメモリアクセスコストを生じず、「順番に行なうアクセス」の振舞いを正しく把握できるのではないかと考えた。また、メモリの距離 x に対するアクセスコストは HMM の結果より $\log x$ と定めた。

このとき、 n 要素 bitonic sort に対して一度呼ばれる n 要素 bitonic merge 内での比較演算は距離 $n/2$ のメモリアクセスを行なっているので、 $O(\log n)$ の遅延が起きることになる。これを再帰的に適用していくと、bitonic sort 全体の計算量は $O(n(\log n)^3)$ となる。

また、 P プロセッサでの並列実行時には $n/2P$ だけ離れた箇所でも同じ距離の通信が並行して行なわれる。最も通信路が混雑するのは $P = n/2$ の時であり、連続した P 個のメモリ上の点から、 P だけ離れた点への通信が一斉に行なわれることになる。(図 6.2 参照)

このような場合について、通信路の「通信負荷累積モデル」を用いて解析する。図 6.2 の状況において、任意の地点 x の通信路の負荷は、同一の負荷の形をしたパッケージが距離 1 ずつずれながら加算されたものであり、1つのパッケージのある時刻の負荷を $l(x)$ として、

$$\sum^d l(x - d)$$

で表される。これは、1つのパッケージの持つ負荷の総量が一定であるという制限(式 5.1)より、一定値であるといえる。つまり、複数のパッケージが生まれていたとしても、あらゆる地点の負荷は単一のパッケージのみが存在した時の負荷を越えないことになり、このような通信パターンは通信路を混雑させない、と解析されることになる。

よって、並列計算時でも通信路の混雑は起きず、計算量は $O(n(\log n)^3/P)$ 、最高で $O((\log n)^3)$ となる。

先の実験の実行時間を、アクセス計算量モデルの理論計算量と比較したものを図 6.3 に示す。縦軸・横軸等は図 6.1 と同じ意味である。図 6.1 の RAM モデルとは異なり、要素数の大きい領域において、diremption factor の値が 1 に収束していつている。つまり、理論値が実測値の増加の様子をうまく表現できていると言え、アクセス計算量モデルは bitonic sort の計算量をうまく見積もれたといえる。

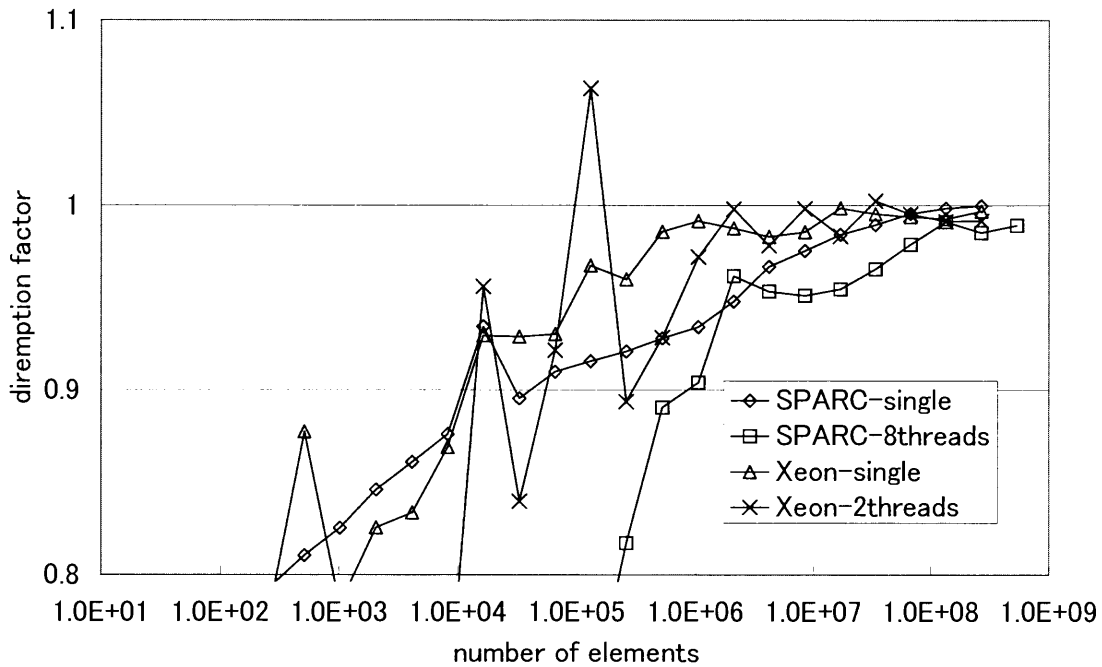


図 6.3: bitonic sort における実計算機とアクセス計算量モデルの計算量比較

6.3 merge sort

merge sort は並列化手法を決定的に決めることができるアルゴリズムであり、解析は容易である。 n 個の入力要素があったとき、それを $n/2$ 個の 2 つの集合に分割し、それぞれを再帰的に merge sort し、できあがった 2 つのソートされた列について、先頭から順番に一つずつ要素を比較し、より前方に来るべき要素を順番にコピーすることで、最終的な n 個のソートされた列を得られる、というアルゴリズムである。RAM モデルで考えると、 $n/2$ 個の要素からなる 2 つのソート列を 1 つの n 要素のソート列にするために n 回の比較演算が必要となり、これが再帰的に $\log n$ 回繰り返されるため、全体では $O(n \log n)$ という計算量となる。また並列化すると、最も多くのプロセッサを使った場合でもソートされた列のマージにかかる $O(n)$ の時間を並列化することはできないため、全体でも $O(n)$ の計算量となる。

次に、アクセス計算量モデルでの計算量を考える。まず、サイズ n の連続した入力データに対し、隣に同じ大きさのバッファを用意し、その 2 つのメモリ領域を交互に使ってソートを行なうものとする。通信は 2 つのソートされた部分領域の要素 (2 領域合わせた要素数を k とする) を順々に比較する時、及び n だけ離れたバッファに比較後の結果をコピーする時の両方で起きる。部分領域間の通信より結果コピー時の距離 n の通信コストが支配的であるので、結果コピー時の通信が最も遅くなるような場合を考えると、これは k 要素全てがソートされていて、全ての要素が図 6.2 のような距離 n の通信を行なう時だと考えられる。もし、図 6.2 の通信のどこかの要素の順序が入れ替わっていると、その部分の通信は「距離 n を 2 回」から「 $n-1$ と $n+1$ の距離の通信を 1 回ずつ」へと変化する。この時、 $2f(n)$ と $f(n-1) + f(n+1)$ の通信遅延を比較すると、 $f(x)$ が上に凸な関数であるので $2f(n) > f(n-1) + f(n+1)$ であり、順序を入れ換えた方が通信時間が短

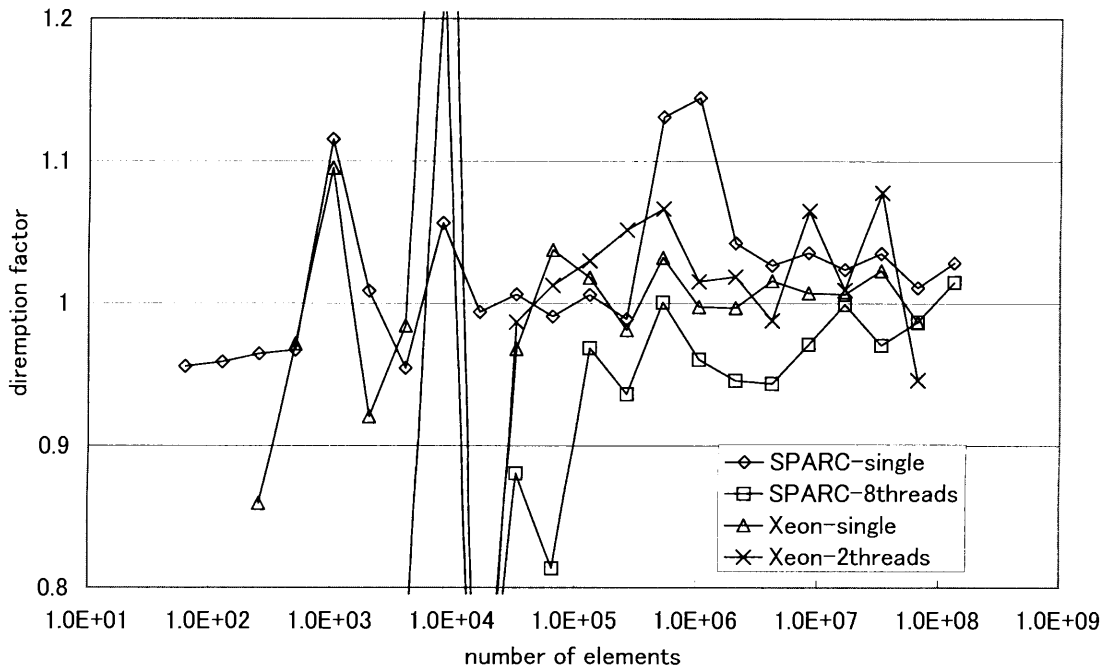


図 6.4: merge sort における実計算機と RAM モデルの計算量比較

くなる。よって、最も遅くなるようなアクセスパターンは図 6.2 のように全ての要素が距離 n の通信をする時で、その時の通信コストは $k \log n$ である。

この時、ソートされた領域の比較には $2 \sum_{i=1}^{k/2} \log i = k \log k - 2(k-1)$ の通信が必要となり、 k 要素の merge sort の時間 $T(k)$ についての漸化式（全体の要素数は n ）を書くと

$$T(k) = 2T(k/2) + k \log n + k \log k - 2(k-1)$$

となる。これを解くと計算量は $O(n(\log n)^2)$ となる。

異なったデータ配置でのアルゴリズム、例えば入力データと作業用バッファを一要素ずつ交互に配置する方式も考えられるが、この配置の場合でもアクセス計算量モデルでの計算量は変わらなかった。

また、並列化について考えると、bitonic sort と同様の考え方で通信路は混雑しないと結論づけられる。そのため、十分多いプロセッサが存在する場合は $O(n \log n)$ の計算量となる。

実計算機上での実験結果を各モデルと比較したものを図 6.4、図 6.5 に示す。merge sort の場合、SPARC 1CPU、Xeon 2CPU の構成での RAM モデルの diremption factor は、1 より大きい値に収束しているように見えるが、Xeon 1CPU の構成では値が 1 に収束しているように見える。一方、アクセス計算量モデルでの予測との比を見ると、SPARC 1CPU や Xeon 2CPU では要素数の大きい領域で 1 に収束する傾向にあり、理論予測が実際の実行時間と一致しているように見えるが、一般的に diremption factor が 1 より小さい値に収束していると言える。つまり、本実験のプログラムは、RAM モデルの $O(n \log n)$ と、アクセス計算量モデルの $O(n(\log n)^2)$ の間の辺りの計算量を持ち、アクセス計算量モデルによる計算量と完全には一致していない。

これは、2つの領域をそれぞれ連続アクセスし、その後でその領域2つを連続したものとしてア

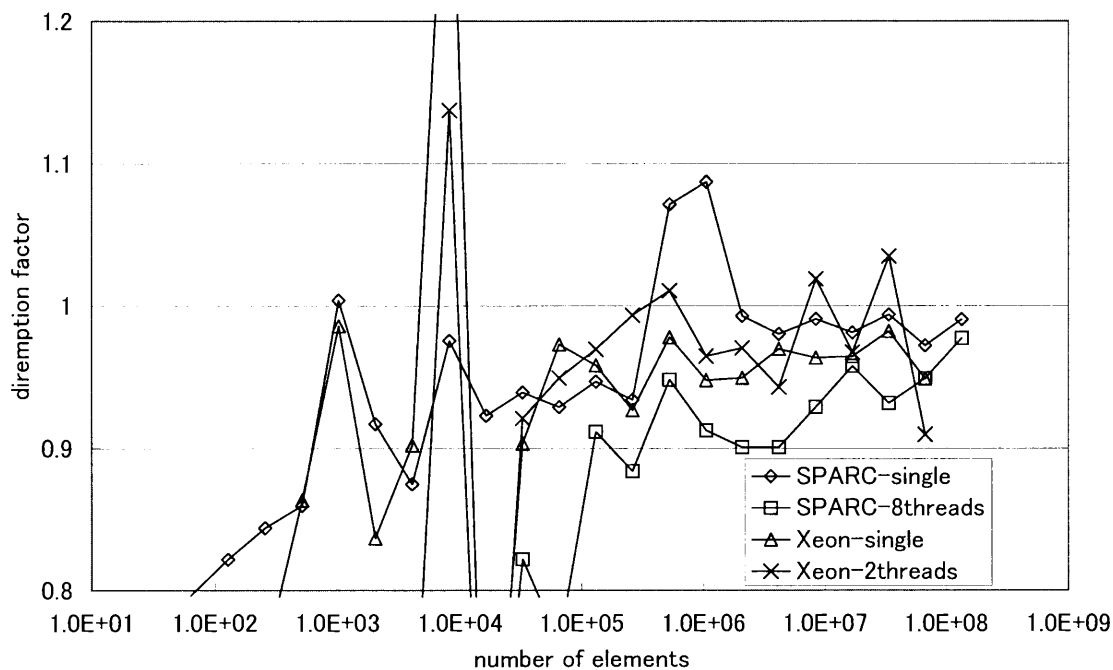


図 6.5: merge sort における実計算機とアクセス計算量モデルの計算量比較

クセスする、という merge sort プログラムのメモリアクセスパターンが、実計算機のキャッシュの仕組みにうまく当たるのに対し、アクセス計算量モデルで解析を行なう際、明示的にキャッシュを使うようなアルゴリズムを採用しなかったためであると考えられる。

6.4 FFT

FFT については、メモリアクセスのローカルリティを高めるためのさまざまな手法が知られている。また、演算数を減らすための手法も多い。しかし、工夫を行なっても演算数のオーダーそのものは変化しないこと、また演算数を減らす手法は並列化した時の解析が面倒になることを考慮し、今回は一番単純な形の FFT、つまり n 入力の FFT を、「バタフライ回路 → 2 個の $n/2$ 入力 FFT → シャッフル回路」と計算する方式について解析を行なう。

RAM モデルでの計算量を考えると、バタフライ回路とシャッフル回路での積和演算、データ移動演算回数は $O(n)$ 回であり、それが再帰的に $\log n$ 回繰り返される構造であるため、全体では $O(n \log n)$ という計算量になる。また、十分にプロセッサが存在する時は、全てのバタフライ演算、シャッフル演算を並列して実行することができるため、この部分の計算量が $O(1)$ になり、 $\log n$ 回の繰り返しを考慮して全体では $O(\log n)$ という計算量になる。

アクセス計算量モデルでは、 n 個の入力データは連続したメモリ上に置かれ、隣接した同じ大きさの作業領域を使って書き換えられるものとする。全体が n 要素の FFT で、 k 要素の部分 FFT を考える時、バタフライ回路の通信は $\log(k/2)$ の遅延が k 回、作業領域への書き込みが遅延 $\log(n)$ を k 回、シャッフル回路の通信コストは作業領域から入力データ領域への移動なので $k \log(n)$ であ

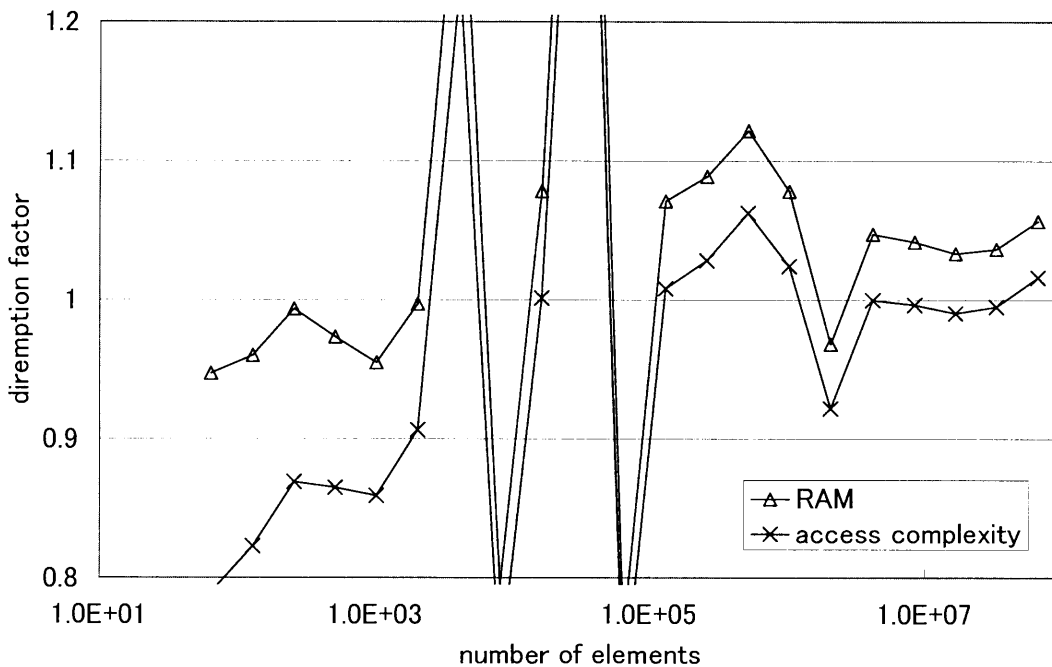


図 6.6: 実計算機と各モデルの計算量比較

る。よって、計算時間の漸化式は

$$T(k) = 2T(k/2) + k \log(k/2) + 2k \log(n)$$

となり、これを解いて全体の計算量は $O(n(\log n)^2)$ となる。また、並列計算時は bitonic sort と同様の議論で通信路が混雑しないことが示され、十分にプロセッサが存在する場合で $O((\log n)^2)$ となる。

簡単な実験を行なったものとモデルとの比較を図 6.6 に示す。これは厳密な FFT ではなく、バタフライで三角関数を掛けるべきところを全ての要素に同一の定数を掛けている。これは、通常ライブラリの三角関数はその部分だけで複雑な計算が必要となり、プログラム全体の実行の中で三角関数計算部分が律速になってしまうため、FFT アルゴリズムの計算量そのものを測定しているとは言えなくなってしまうからである。実用的な FFT の場合は三角関数も表を引くなどして高速化する必要があるのだが、どの程度のメモリを使って高速化するか、もアクセス計算量モデルでの解析では重要な要素となってくるため、今回の解析ではその部分を無視し、定数時間がかかるものとして測定することにした。参考程度の実験であるので、並列化も行なっておらず、実験環境も少し異なって Opteron 1.4GHz である。RAM モデル理論値による diremption factor とアクセス計算量モデル理論値による diremption factor との 2 つの系列を示してある。図 6.6 に示されている通り、RAM モデルでは diremption factor が 1 より大きい値に収束しており、実計算時間が理論値より遅くなっているが、アクセス計算量モデルでは diremption factor の値が 1 に収束しており、正しく実行時間を把握できていると言える。

merge sort と異なり、今回の単純な FFT の場合はシャッフル操作の部分がほとんどランダムアクセスで、実計算機ではキャッシュをうまく使えない。そのため、今回のキャッシュを意識しない

表 6.1: アクセス計算量モデルによる解析結果 (括弧内は並列性が限界まで利用できるときの計算量)

アルゴリズム	アクセス計算量モデル	(並列性の限界)	PRAM モデル	(並列性の限界)
bitonic sort	$O(n(\log n)^3/P)$	$(O((\log n)^3))$	$O(n(\log n)^2/P)$	$(O((\log n)^2))$
merge sort	$O(n(\log n)^2/P)$	$(O(n \log n))$	$O(n \log n/P)$	$(O(n))$
FFT	$O(n(\log n)^2/P)$	$(O((\log n)^2))$	$O(n \log n/P)$	$(O(\log n))$

解析方法でもうまく計算量が見積もれたと考えられる。

6.5 解析のまとめ

この章では、よく知られた並列アルゴリズムである bitonic sort、merge sort、FFT の 3 種類について、提案するアクセス計算量モデルを用いて解析的に計算量を求めようと試みた。その結果、通信路の混雑の様子を単純化した通信負荷累積モデルを用いれば、アクセス計算量モデルは十分単純化され、これらのようなアルゴリズムにおいても解析を行うことが可能であることが示された。それぞれの解析結果をまとめると、表 6.1 のようになる。提案しているアクセス計算量モデルでの解析結果には、RAM モデルでは解析できていなかったメモリ階層の影響を示す計算コスト $\log n$ が現れている。

この解析結果を、実際の共有メモリ型並列計算機上に実装したプログラムを用いて、diremption factor という指標により比較評価した。その結果、bitonic sort と FFT では、アクセス計算量モデルが実際のプログラム実行をうまく表現していることが示された。

しかし、merge sort アルゴリズムについては、実際のプログラム実行は RAM モデルとアクセス計算量モデルの予測の間あたりの計算量であるように測定された。これは、merge sort アルゴリズムを実装したプログラムが、キャッシュメモリの機構が自然と活用されてしまうような書き方であったことに起因すると考えられる。bitonic sort と FFT のプログラムでは、キャッシュメモリ機構がそれほど活用されないものであったため、今回の解析がうまく現実を表現できたが、merge sort においては、アクセス計算量モデルでもキャッシュを明示的に活用するようなアルゴリズムを用いて評価する必要があると考えられる。

また、FFT の検証では、最も単純なアルゴリズムを用い、実プログラムでは三角関数の計算時間を考慮に入れないなどの単純化を行っている。今後、さらに精密化した評価を行う必要があると考えられる。

6.6 クラスタ構成での通信路モデル化の評価

6.2 章の bitonic sort の共有メモリによる並列計算では、効率はそれほど落ちることなく並列化が行なわれている。これはアクセス計算量モデルの計算量とも合致しているが、通信路のバンド幅がもっと狭い、ネットワーク接続された分散計算機のような環境では同様の結果となるか、についても確認する必要がある。

そこで、bitonic sort で起きる通信パターン、同じ距離の通信が複数密集して起きる図 6.2 のような状態を、PC クラスタ上で再現した実験を行なった。実験環境は Xeon 2.4GHz dual、64 node のクラスタであり、各計算ノードの NIC は 1Gbit-ether である。この計算ノードが、32 台ずつ 1

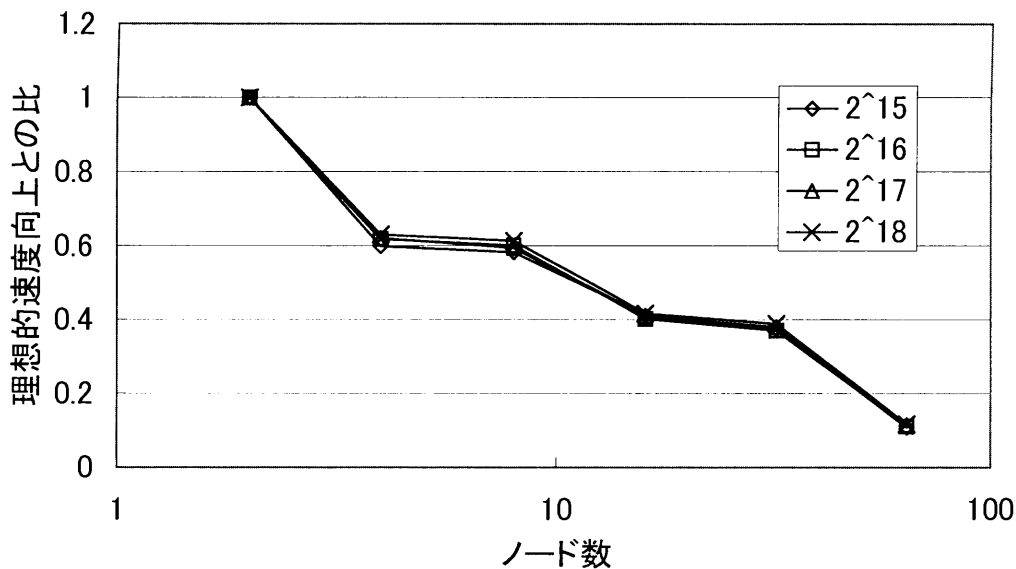


図 6.7: 同時通信数と通信速度

つのネットワークスイッチにつながれており、2つのネットワークスイッチ間は4本の1Gbit-etherをtrunkして結合されている。計算ノード台数が32台までの実験は1つのネットワークスイッチのみを使用している。通信ライブラリとしてはphoenix[22]を用いた。全体のデータサイズを一定とし、通信を行なうノードの数を変化させて通信時間を測定した。モデルでは台数に比例した速度向上が得られると期待される。結果を図6.7に示す。グラフの系列は全体のデータサイズの違いを示しているが、結果は4系列ともほとんど変わらない。横軸はノード数、縦軸は期待される理想速度向上と実際の通信時間との比を示している。図6.7からもわかる通り、ノード数が増えるに従って速度向上率が落ちる結果となっている。

つまり、ネットワークの帯域幅が十分に大きい環境では現在の通信負荷累積モデルで精度良く振舞いを予測できるが、帯域幅が小さいところは現在のモデルではうまく説明できていないと言える。通信路を全て統一的に表現することを目指し、通信路のモデル化についてはさらに検討を続ける必要がある。

第7章 第I部のまとめと今後の展望

第I部では、計算のコストの本質は演算ではなく通信にこそ存在するという基本概念から、アルゴリズムの計算量を、均一に広がったメモリ上に存在するデータ間の通信の遅延の総和であるとする、アクセス計算量モデルを提案した。このモデルを用いることで、単一計算機内のメモリ階層から計算機間のネットワーク遅延の差異までを統一的に、かつ簡潔に記述することができる。このモデルは十分簡潔なものであり、並列アルゴリズムの計算量を解析的に求めることができることを、merge sort、FFT のアルゴリズムで検証した。また、bitonic sort アルゴリズムの実計算機上での実行と比較することで、従来の PRAM モデルが表現しきれなかった振舞いを、このモデルがよく表現できていることが示された。FFT の簡単なプログラムでもこのモデルの予測が実験とよく一致した。

ただし、merge sort の解析結果が実計算機と必ずしも一致しなかったことからわかる通り、キャッシュを明示的に操作するアルゴリズムで解析しなければならない場合も多いと考えられる。さらに、通信路のモデル化には、帯域幅が十分でない環境において、通信の輻輳を必ずしも十分に表現しきれていないという側面もある。これらを今後の検討課題としたい。また、より多くのアルゴリズムやより大規模なアルゴリズムに適用することで、このモデルによる計算量解析の妥当性と適用可能性を検証していくことも、今後取り組むべき課題である。

第II部

多くの部分計算を共有する問題の耐故障計算フレームワーク

第8章 背景

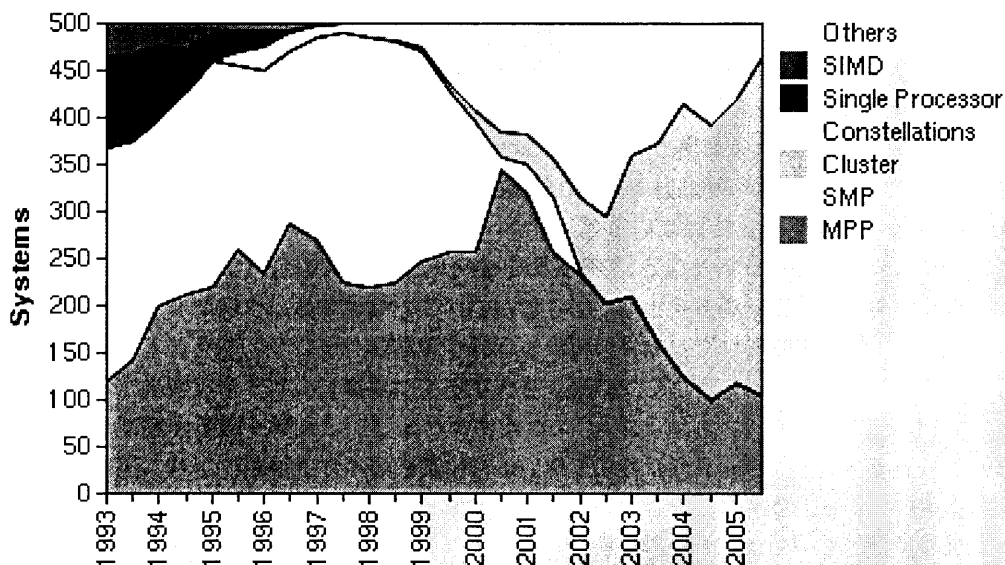
8.1 並列・分散計算環境の普及

近年、並列・分散計算環境は急激に普及してきた。これは、単体計算機のパフォーマンス向上の鈍化と計算機の低コスト化によって引き起こされている。

単体計算機のパフォーマンスは、素子の動作速度向上に伴ってこれまで大きく向上してきたが、近年では発熱に関する問題などから素子の動作速度向上が鈍化しつつある。一方、半導体は年々微細化し続けており、「集積度が3年で4倍になる」という「ムーアの法則」[5]が語られ続けてきた。ムーアの法則自体は経験則にすぎないが、コンピュータ業界全体の技術研究目標として設定されることも多いため、今までのところ結果的にこの法則にほぼ従った形で集積度は向上してきている。この半導体の微細化により集積できるようになった大量のトランジスタを用い、複雑な構成の論理回路を使う様々な手法を構築することで単体計算機のパフォーマンス向上を計ってきたのではあるが、そのような手法の利用も次第に困難さを増してきている。現在主流のプログラミングモデルでは命令列の平行性を十分抽出できない点、第I部で述べたようなメモリ階層の深化に伴い、メモリアクセスにかかる時間が大きくなりすぎてCPUの実行効率が上がらない点、などが主な原因となり、設計の工夫による高速化も難しくなりつつあるのである。また、複雑な構成によって設計が難しくなっていることも問題である。CPUは速いペースで設計更新を続けていかなければ市場競争に勝ち残れないが、複雑な構成は設計、検証、大量生産時の歩留まり向上にかかる時間を長くしてしまい、競争力を大きく下げることになってしまう。

そこで、複数のCPUコアを同時に使用することでパフォーマンスを向上させる、というアプローチが重要度を増すことになった。素子の微細化の結果複数のコアを現実的な面積に納めることが可能になっており、設計の困難さも軽減される。このような方針に基づいたマルチコアのCPUは次第に一般的になってきており、現在は市販されるパーソナルコンピュータやノートパソコンについても、2つ程度のコアを持つCPUを搭載した物がごくありふれた製品として売られるようになってきた。今後はさらにコアの数が増え、数個のコアを含むマルチコアから数十個以上のコアを含むメニコアへと発展していくことが予想されている。

また、計算機のコモデティ化と低コスト化が進んだことも計算機構成に大きく影響を与えている。高速、大規模な計算機を構築する際に、専用のハードウェアを特別に設計するのではなく、コモデティ化した一般的なプロセッサやネットワークを用いた方がコストパフォーマンス比が良好になることが多くなったのである。一般的なハードウェアは専用ハードウェアに比べて性能が劣ることが多いが、大量生産により一般的なハードウェアの方が価格が低いため、より多くのプロセッサを並べることで全体の性能を高くすることができる。一般的な技術は技術更新が早く、技術的成熟度も高いため、設計から稼働までの期間が短くなることも期待できる。このような理由から、近年構築される超大規模な計算機は、一般的なプロセッサを多数並べて構成したものがほとんどとなっている。例えば、世界のスーパーコンピュータのランキングであるTop500 [23]において、2005年11月のランキングで上位を占めているBlueGene、ASC PurpleやAltixなどは、一般に市販されているプロセッサそのものか極めて近いものを多数接続して構成されているし、5位のThunderbirdは



09/11/2005

<http://www.top500.org/>

図 8.1: Top500 での計算機構成の変遷 (2005 年まで)

市販されている PC サーバの集合体である。また、上位に入っているシステムはこの数年間、プロセッサ数を単純に増やしていくことで性能を向上させ、上位にとどまり続けていることも注目すべき点である。構成を変えないまま規模を拡大していくことができるのも、一般的なプロセッサを一般的なネットワークでつないだ構成を用いる際の大きな利点である。図 8.1 は Top500 リストにおける計算機構成の変遷を示している。Cluster で示されている構成のほとんどは市販品の PC サーバを接続した PC クラスタであるが、近年急激に増加していることが読み取れる。

スーパーコンピュータの世界で PC クラスタが使われるのと同様の理由で、より小規模な計算サーバにおいても PC クラスタ構成は多く用いられている。パーソナルコンピュータは年々低廉化しており、数台から 100 台程度の PC を並べてクラスタ構成にした計算機は、研究室単位や一般企業の部署単位程度の規模でも十分導入可能になっている。

このように、単体計算機の世界での速度向上ペースの鈍化に対抗するために作られたマルチコアによる並列計算環境や、低廉化した計算機を多数接続することでコストパフォーマンスを向上させようとするクラスタ構成の分散計算環境が広く普及し始めており、今後もさらに大きな規模、さらに多くの場面で活用されていくと予想される。

8.2 耐故障計算フレームワークの必要性

並列計算環境は身近なものになっている一方で、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なければならない。その上で、逐次プログラムの適切な箇所に、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になってしまう。

また、並列計算環境の多様さもプログラミングを難しくしている。前述したように、並列計算環境はマルチコアプロセッサによる共有メモリからクラスタ構成まで、様々な構成が存在している。共有メモリマシンであってもメモリアクセスにかかるコストが不均一な NUMA 構成が存在したり、クラスタにおいてはネットワーク速度やトポロジといった構成が大きく異なっていたりと、性能に関するパラメタも考慮すれば並列計算環境は実に多種多様なものになってしまう。しかし、例えば共有メモリマシンならば共有メモリを用いたスレッドプログラミングを使うと効率がよいし、高速なネットワークで結合されたクラスタならば MPI などのメッセージパッシングライブラリを、インターネット接続でしかつながっていないならばソケットを使った通信による分散プログラミングを、というように、それぞれの環境で効率よく扱いやすいプログラミング手法が異なっている。このため、並列処理では実行環境によって大きく異なるプログラムを作らざるを得ず、プログラマに必要な知識も手間も増大してしまうのである。

このような問題に対してはいくつかの方策がある。まず、並列計算向きのプログラミング言語を用いるという方法がある。もともと並列性を意識し、並列実行に伴うバグが生じにくいような計算モデルを持つ言語、例えば並列拡張された LISP や並列論理型言語 KLIC[24] などを用いれば、バグのない並列プログラムを書くことはずいぶん簡単になる。また、このような言語は通信や同期をプログラマから隠蔽したり、言語レベルでサポートしたりしているため、プログラマに必要とされる知識は少なくなり、また実行環境の違いも処理系レベルで吸収することができる。しかしこの方法は、プログラマが慣れ親しんだプログラミング言語から大きくかけ離れた言語を使わなければならない、という欠点を持つ。多くのプログラマは C や Java などの手続き型言語の知識しかなく、新しい言語や計算モデルを習得するには大きな労力が必要となるため、あまりこの方法は広まってはいない。

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。行列演算のように、対象となる問題がわかりやすい形で切り分けられ、並列化アルゴリズムが決まっているような場合は、その問題を解くための並列化ライブラリを使えばよい。このような並列化ライブラリとしては BLAS (Basic Linear Algebra Subprograms) [25] や LAPACK (Linear Algebra PACKage) [26] など様々なものが開発され、アーキテクチャごとに最適化されて使われている。しかし、ライブラリが作れるほどアルゴリズムが確定していないような問題領域も存在する。そのような種類の問題に対しても、抽象的なある種の問題領域ととらえて考えると、並列化のための計算分割、通信、同期などの方法が共通化されることがある。そのような場合には、並列化に関わる部分のみを抽出し、並列化フレームワークとして定式化することが可能である。このフレームワークを、並列処理をよく知る開発者があらかじめ実装しておけば、個々の問題固有知識を持つプログラマは、そのフレームワークに当てはめる形でプログラミングすることで、並列化に関する種々の知識を必要とすることなく、正しく簡単に並列環境を使用することができるのである。

このような並列化フレームワークはすでにいくつか提案されている。特に、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に、多くの並列化フレームワークが研究されてきている。例えば、OpenMP[1] は共有メモリ上での配列に対する演算を行うためのフレームワークであり、C や Fortran などの手続き型言語に簡単なプラグマを用いて指示文を加えることで、配列の分割配置や演算、同期などを行ってくれ

るものである。このようなフレームワークは、構造が単純な計算についてはよく研究され、成果を上げてきている。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。

並列化フレームワークには、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が要求されていたのであるが、並列計算環境が多様化することで、それら以外に要求される事項も次第に増えてきた。

例えば、動的な計算機構成変更への対応要求が挙げられる。近年、PC クラスタなどの並列計算環境が身近な物になってくると同時に、計算機の利用形態も多様化してきた。以前のように、計算機センターで限られた利用者のみが限られた時間だけを占有して使えるようにコントロールしてあるような利用形態だけではなく、ある部署の管理する計算機を一時的に借用して自分の管理する計算機とともに使ったり、時々借用先の部署のプログラムが走るので計算機を譲り渡す必要が生じたり、といったように、使用する計算リソースを頻繁に切り替えて使うような利用形態もよく見られるようになった。遊休計算機を集めて計算を行う、デスクトップ GRID と呼ばれる環境は、このような動的に増減する計算機構成の極端な例であると言える。計算リソースが増減するたびに計算を停止し、パラメータを変更して実行を再開するというのでは手間もかかり効率も良くない。よって、計算が行われている途中に、参加する計算機の増減に対応して負荷の再配置などを行うような仕組みが、強く求められるようになってきている。

また、故障に対する耐性も強く求められている要件の一つである。現在多く用いられている PC クラスタのような構成では、個々の計算ノードの信頼性はそれほど高くはないため、参加台数が増えたと特に頻繁に、どこかのノードが故障している状態になる。故障でなくても、他の部署と共有しているような環境では、連絡が不十分なまま計算リソースを突然取り上げられたりすることもあり得る。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、これらの事故は起きる可能性が高くなる。よって、耐故障性、つまり、一部の計算機が故障しても全体の計算は止まることなく実行され続けるという仕組みがより強く望まれるようになってきているのである。

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良い。

以上で述べてきたように、現在の並列計算環境を十分活用した並列プログラムを簡単に作成できるようにするためには、より多くの問題領域で使える、動的構成変更への対応や耐故障性を持つ並列計算フレームワークが求められているのである。

8.3 部分計算を共有する問題

前述したように、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まだまだ不十分である。そこで、本研究では、木状の依存関係を持ち部分計算を共有するようなタスクに対して、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。構造を示す擬似コードを図 8.2 に示す。ここで、複数の子問題の計算には依存関係がない場合を想定しており、子問題の計算を並列に実行することで計算の高速化を図ろうとしている。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。

```

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 8.2: 対象問題の構造を表す擬似コード

例えば、15 パズルを A* アルゴリズムで探索して解くことを考えてみる。ある盤面が問題として与えられたとき、そこから一手ランダムに動かした盤面を作り、それらの盤面からゴールへ向かう部分問題を子問題として生成する。子問題の全てを解けば、その結果の中で最も良い結果、つまり最も早くゴールへ到達できるような子問題の盤面へ向かうように、現在の盤面での解を定めることができる。ここで、部分問題は盤面によって定められることになるが、ある盤面から複数の経路をたどって同一の盤面に到着することがある。これが、部分問題が複数の親問題から共有されている状態である。同一の盤面に到着したことを認識せず、別々の部分問題として探索することも可能であるが、その場合は部分問題を複数回計算することになってしまい、探索の効率が極めて悪くなる。

図 8.3 は、実際に 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* で解くプログラムを作り、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、縦軸は部分結果を再利用した時の盤面数を表している。部分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。縦軸、横軸はともに対数軸であるので、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が極めて大きくなっていくことがこのグラフからわかり、大規模な問題を解くためには部分計算の再利用をしなければならないといえる。

一般に探索問題の計算量は、ある局面から調べなければならない次の局面数を表す平均分枝数 b 、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、ある割合ですでに計算された同一の局面が現れると考えられるため、この平均分枝数 b がその割合で小さくなると期待できる。部分問題の再利用を行ったときの平均分枝数を b' とすると、探索に必要な計算量は、再利用を行わなかったときの

$$\left(\frac{b'}{b}\right)^d$$

倍になり、 d が大きくなるにつれて、すなわち問題規模が大きくなるにつれて、その差が広がることになる。

このような振る舞いを示す問題は多い。例えば、ゲーム木探索が挙げられる。図 8.4 は、コンピュータ将棋プレイヤーが様々な盤面において探索によって手を決定するまでの計算量を表したものである。ここで実験に使用しているのはコンピュータ将棋プレイヤー「激指」[27] であり、探索の際に調べた盤面の数を同様にプロットしている。やはり、部分問題の共有を行うことで探索効率が大

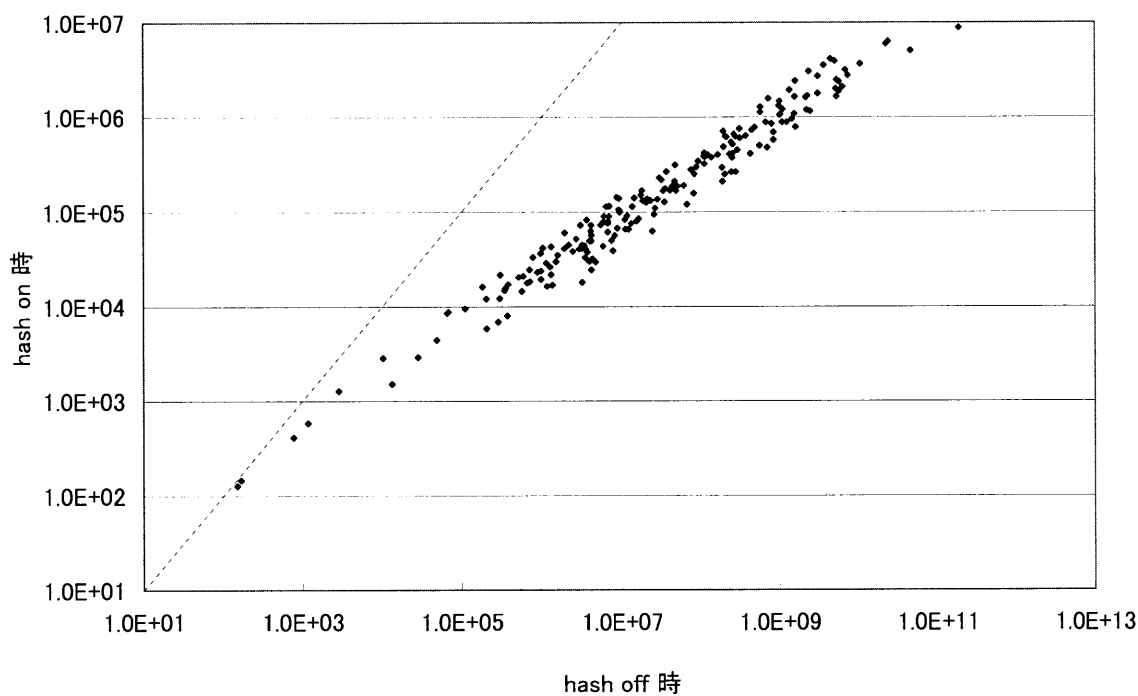


図 8.3: 15 パズルでの部分問題共有の効果

きく改善されていることがわかる。この場合、計算量が大きくなったところでの探索の効率化が 15 パズルほど明確ではない局面もあるが、これはメモリ量の制約によって局面の記憶がそれ以上できなくなっていることに起因している。

このように、部分問題を共有するような問題は多く、それらを解く際には部分問題の結果を再利用することが効率のために重要である。

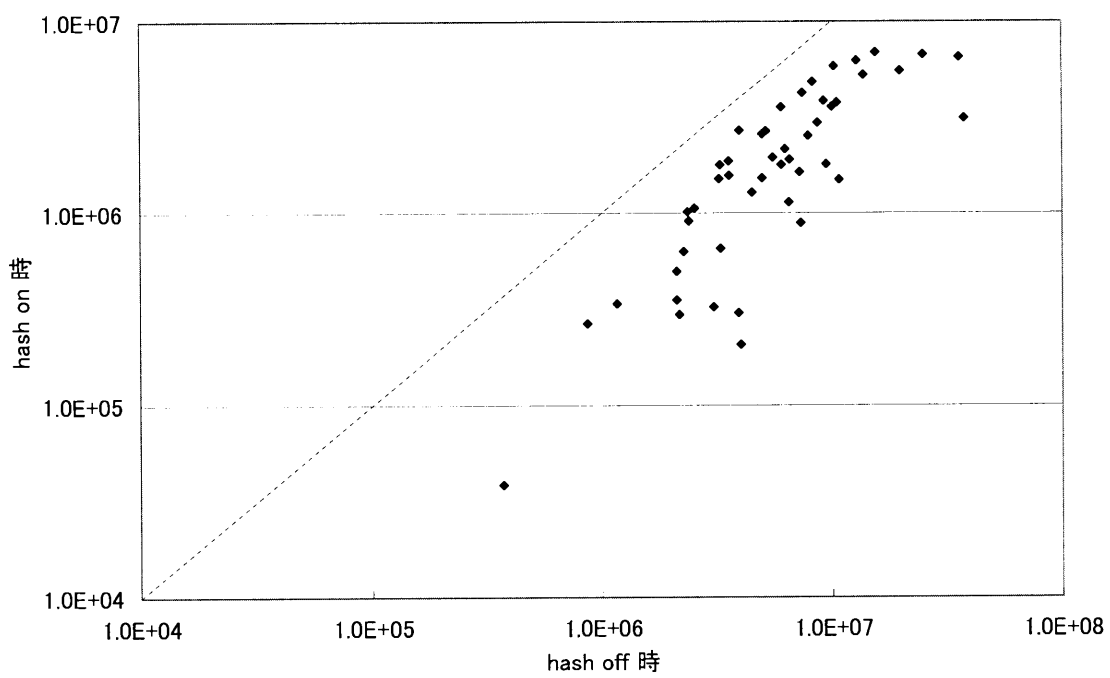


図 8.4: 激指での部分問題共有の効果

第9章 耐故障計算フレームワークの提案

本論文では、前述のような、部分問題を共有する問題を対象に、耐故障性を持つ並列探索フレームワークを提案する。

以下、提案するフレームワークの概要を示す。

9.1 フレームワークの方針概要

提案手法は、分散ハッシュテーブルの考え方をういて分散計算を行うことを基本的な概念としている。

分散ハッシュテーブルは多数の計算機に存在する資源を発見・利用するために提案されてきた手法であり、ネットワーク管理を司る中心となる計算機がいない、またはその寄与が低い、Peer to Peer と呼ばれる分散されたネットワーク網構造の上で、広く研究されてきている。

これまでの研究では、分散ハッシュテーブルはディスクなどの上のデータ資源を共有するための枠組みとして捉えられていることがほとんどであった。この場合、ある参加ノードは自分の受け持ちの範囲に存在するディスク上のデータ資源について責任を持ち、保持と検索要求への返答を行うことになっていた。しかし、分散ハッシュの枠組みに乗せる資源が静的なデータのみである必要はない。そこで、提案手法では、参加ノードは自分の受け持ちの範囲に存在する部分問題の計算について責任を持ち、必要な計算を行い、その結果を保持して、要求があればその部分問題の結果を返答する、という役割を果たす。

つまり、静的なデータ資源のみではなく、動的な計算そのものを分散ハッシュテーブルによって管理し、巨大な計算を多数の計算機によって分散して保持し続け、結果を求めよう、というのが本論文の提案手法となる。

9.2 対象とする問題

本フレームワークが対象とする問題は、

- 分割統治法が可能なタスク生成が行われ、
- メモ化が可能

なものである。

以下、詳細に対象問題の性質を示す。

9.2.1 樹状再帰構造

部分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。部分タスクは再帰的に子の部分タスクを生成し、その結果を用いてもとの部分タスクの結果を求めるとい、樹状再帰の構造を

持つ。親問題をより小さいサイズの子問題に分割し、それらの結果を用いて親問題の解を求める、という分割統治法による計算アルゴリズムであり、複数の子問題の計算を並行して行うことによって計算を高速化できる。

例えば 15 パズルの探索の場合、ある盤面の状態を入力とし、そこからパズルが解かれた状態に至るまでの最短経路を返すものが、15 パズルを解くためのタスクである。もともとの 15 パズルの問題は与えられた盤面 R から解までの最短経路を求めるというものであるが、これが最初に存在する唯一のタスク、ルートタスクである。ルートタスクは、その盤面 R から一手進めた盤面群 S_i について、それぞれの盤面から解までの最短経路を全て求めれば、その中で最良の解、最短の経路を与えるような盤面に向かう一手を、もともとの盤面 R での解の一手として採用すればよいとわかる。つまり、ルートタスク $t(R)$ の結果が、子タスク $t(S_i)$ によって求まることになる。このように、15 パズルでは、盤面 x に対して最短経路を求めるタスク $t(x)$ が定められており、問題全体はタスクが再帰的に生成されるような構造を持っている。

フレームワークが対象とするのは、このような問題である。

9.2.2 部分問題の決定性・副作用の排除

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、共通する部分タスクを発見し、以前にその部分タスクの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度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。また、タスク実行時に何か副作用が起きるとすると、同一の入力に対するタスクが 2 回実行されたときと、結果の再利用により片方のタスクが実行されなかったときとで、副作用の起きる回数が異なってしまう。もし副作用がタスクの出力結果に何らかの影響を与えたとしたら、再利用により問題の結果が変化してしまうことになる。

9.2.3 部分問題の発見のための仕組み

タスクは入力によって一意に決定され、その入力の値を使って他のタスクからアクセスされるものとする。つまり、親タスクが子タスクを生成する際、子タスクはその入力の値によって一意に定まる。ある部分タスクの結果が必要なときには、そのタスクの入力の値を指定することで、全ての部分タスクから必要な結果へアクセスすることが可能になる。

図 9.1 に、対象となる問題の擬似コードの例と依存関係を示す。部分問題は関数 $f()$ に与えられる入力によって一意に指定され、別々の親問題から結果を要求されることになる。

対象となる問題は共通する部分問題が存在するようなものであるので、その共通部分問題を発見するためのこのような仕組みは必ず必要であり、特殊な要件ではない。

また、その入力からはハッシュ値を計算することができるものとする。これは分散ハッシュテーブルによるシステム実装のために必要な要件となる。

9.2.4 対象問題の探索の定式化

本フレームワークは、以上のような問題を対象とする。本フレームワークの探索アルゴリズムを擬似コードで示すと、図 9.2 のようになる。 $f(x)$ を計算するとき、フレームワークはメモ化によ

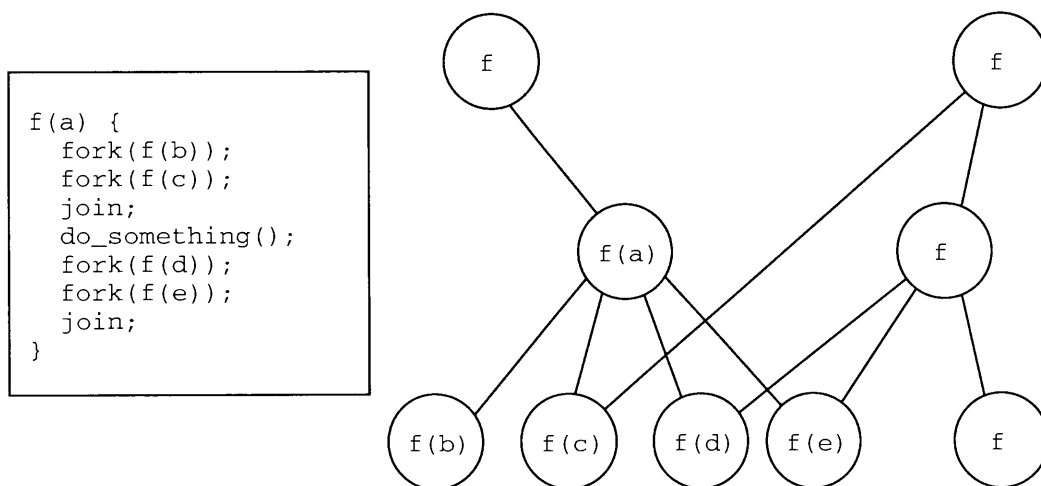


図 9.1: 対象となる部分問題の依存関係

```

memoize-f (x) {
  if (previously-computed(x)) {
    return result-of-f(x);
  } else {
    r = f(x); /* f の中で分割統治法により並列探索される */
    store-result(x, r);
    return r;
  }
}

```

図 9.2: 対象問題の探索アルゴリズム

る探索の効率化を行うため、`memoize-f(x)` のようなアルゴリズムで計算を行う。樹状再帰構造を持つ `f(x)` は、図 9.1 のように分割統治法により並列に探索される。

9.3 システムの必要要件

9.3.1 通信の要件

本手法で必要な通信は、ある範囲内の整数値のキーを持つメッセージが、その整数を割り当てられている計算ノードに到着する、という条件さえ満たしていればよい。これは、分散ハッシュテーブルの基本となる通信機能であり、さまざまな通信手法が研究されている中でも、共通して提供されている機能である。

この通信要件は緩やかなものである。例えば、通信は FIFO を仮定せず、メッセージの到着順序は前後する可能性がある。メッセージは、必ずしも最適でないルーティング経路を通り、「いつかは到着する」という条件のみが満たされており、それ以上の時間的な制約は存在していない。もち

ろん、高性能通信路を用いることができれば計算も高性能にすることが可能になるように実装することが必要であるが、Peer to Peer のような信頼性の低いネットワークを用いる場合も考慮に入ると、この程度緩やかな条件の通信でシステムを設計しておく必要があると考えられる。

さらに、故障するノードがあることを考慮に入れると、メッセージが失われる場合も許容する必要がある。提案するフレームワークの設計においては、正常動作するノードがある限り、何度かメッセージを送信すればいつかはその一部のメッセージが配達される、つまり永久にメッセージが届かなくなるような宛先が存在しない、という程度の緩やかな制約さえ満たしていれば、フレームワークの動作が可能であるように設計を行った。

フレームワークの実装においては、メッセージに ack 返答待ちタイムアウトを設けて、自動的にメッセージを再送信するような通信システムを使うことも可能であるし、そのような回復機構がない場合でも、アルゴリズム自体に必要なメッセージの再送信メカニズムが含まれるよう設計を行っている。

9.3.2 故障モデル

提案システムが対応する故障のモデルは、ある時点においてある計算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。

Byzantine Failure と呼ばれる、故障ノードの動作が不定になってしまうような故障状態は、分散アルゴリズムによって効率よく扱うことは難しく、実用的な分散計算フレームワークを構築する際には考慮するのは現実的とは言えない。よって、最も単純な故障モデルである、ある時点からノードの動作が停止し、そのまま復活しないという場合のみを考えることにする。

9.4 アルゴリズム概要

9.4.1 タスクの分担

タスクは、入力によって一意に定められるが、入力からはある決められた範囲のハッシュ値が計算できる。そのハッシュ値に従ってタスクを担当する計算ノードが定まる。

図 9.3 はタスクが複数のノードに分担される様子を示したものである。図の上部はもともとのタスクの依存関係を示している。それぞれのタスクはハッシュ値を持っている。

計算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、ハッシュ値の範囲全体が全ての計算ノードによって、重複もなく抜けもなく分割されるようにシステムが動作し続ける。図の下部はハッシュ値の範囲が PC1 から PC3 までの 3 台の計算ノードで分割されている様子を示している。上部のタスク木は、その呼び出し関係は保たれたまま、ハッシュ値に従って下部のように PC1 から PC3 までに分担されることになる。

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、計算ノードの担当範囲割り当てを変更することで、新しい計算ノードの追加、脱退が可能になる。

計算自体は、計算ノードの担当範囲内にあるタスクから、まだ計算を始めていないものを選択し、そのタスクを計算、子タスクの結果が必要になったらその結果を待っている状態にあることを記憶し、再びまだ計算が始まっていないタスクを選んで計算を始める、ということを繰り返すことによって進行する。

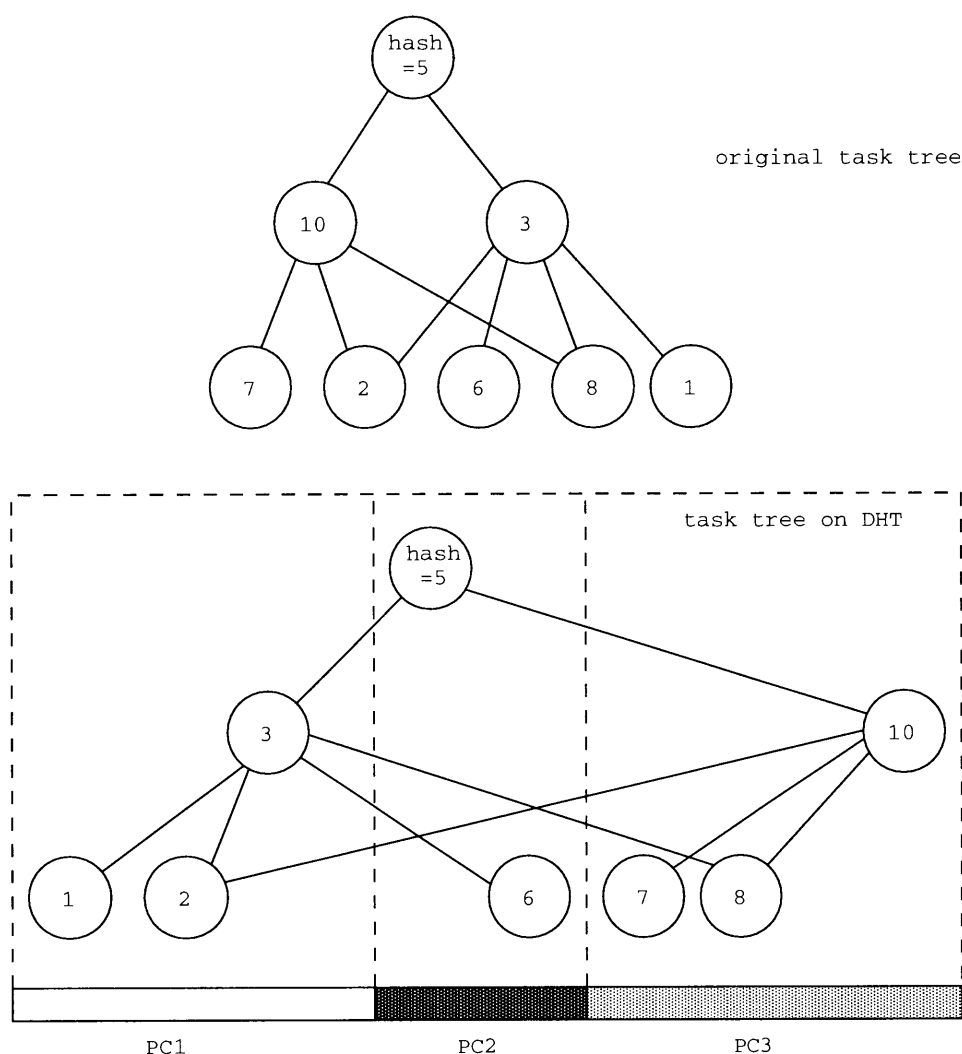


図 9.3: 分散ハッシュによるタスクの分担

9.4.2 タスク依存関係の保存

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。タスクが生成されたときには、必ずその値を必要とする親タスクが存在すると言えるので、これを記憶している。また、タスクの計算途中に未知の親タスクから結果を要求されることがある。これは、その部分タスクが共通部分タスクとなっていることを示しており、タスクはその親も記憶に加える。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全てに結果を返す。

また、それと同時に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を検出するための方法には、システムのハートビートモニタなどを用いて担当計算ノードが故障したことを検出する方法と、そのような故障検知システムを用いない方法がある。後者の場合、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るという方法によっても子タスクの再生成は可能である。つまり、子タスクの結果が必要な間は定期的にメッセー

ジを送り続け、子タスク側で重複した要求を無視するような動作を行えば、子タスクが本当に失われていたときに再生成されるようになる。これは非常にシンプルな方法ではあるが、通信量が増えるため計算効率に影響を与えられると考えられる。何らかの故障検知システムが使用可能かどうか、故障率がどの程度かなどの条件によって適切な方式は異なり、実装の際に考慮すべき点になると考えられる。

また、全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全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計算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、その計算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。

9.5 提案アルゴリズムの利点と欠点

提案アルゴリズムは、重複する部分計算をできる限り発見し、それを最大限に利用して高速化を図ることを目的としたものであり、第 8.3 章でも見たとおり、大規模な問題を解くためにはこれが極めて重要な利点となる。

よりシンプルな手法、例えばマスタワーカ型の並列計算によって、同様の問題を耐故障性を保ったまま解くことは容易である。この場合、マスタ内部で重複計算を検出し、それぞれのワーカ内部でも重複計算を検出することは可能であるので、検出できないのはワーカ間の重複計算のみとなる。このデメリットがある程度限定的であれば、マスタワーカ型の並列計算でも探索は可能なことになる。

しかし、マスタのメモリには限界があるため、マスタが検出できる重複部分計算の量にはそれによって決まる限界がある。問題全体のサイズが大きくなり、マスタに対してワーカの担当する部分計算が大きくなると、ワーカ間で見つけられずに無駄に計算される重複計算が増え、全体の効率が落ちることが考えられる。これを避けるためにはマスタワーカを複数段に再帰的に適用する方法が考えられるが、いずれにせよマスタの検出できる重複計算に限界があることには変わりはない。

これに対し、提案手法にはマスタが存在せず、検出できる重複部分計算の量、すなわち保持できる部分計算の量が参加計算ノードの増加に伴ってそのまま増加する。よって、問題が大規模になっても、計算ノードをそれに合わせて追加していけば、探索効率を下げることなく問題を解くことが可能になると期待される。

また他にも、耐故障性の面においては、マスタワーカのような single point of failure が存在しないという利点もある。

一方、提案手法の欠点となるのは通信量に関する問題である。本手法は全ての部分計算が生まれるときに通信が必要となるため、タスク数以上の数のメッセージが通信されることになる。これに対しマスタワーカ方式では、マスタとワーカに分割される境界部分に存在するごく一部の部分問題だけが通信を必要とし、マスタ、ワーカの仕事量に対して通信量が少なく抑えられる。

提案手法のもともとの理念は、過去に実行したことがある部分問題を、再計算する代わりに、外部の計算ノードにメモしてある計算結果を通信によって取り寄せることで、結果を素早く得ようというものであった。よって、通信にかかるコストが再計算にかかるコストと比較して十分小さくなるように末端のタスクの粒度が調節されていれば、提案手法は計算を高速化できることになる。通信にかかるコストは一定ではなく、メッセージ数が増えるに従って輻輳などの理由によりコストが増大していくと考えられる。適正な粒度を考えるとときには、このような通信コストの増大も考慮に入れておく必要がある。

通常のメモ化:

```
memoize-f (x) {  
  if (previously-computed(x)) {  
    return result-of-f(x);  
  } else {  
    r = f(x);  
    store-result(x, r);  
    return r;  
  }  
}
```

拡張されたメモ化:

```
memoize-f (x, c) {  
  if (previously-computed(x)) {  
    /* 記憶した計算結果が c という条件下でも使えるか判断する */  
    if (enough-result(c)) {  
      return result-of-f(x);  
    }  
  } else {  
    r = f(x);  
    store-result(x, r, c); /* c という条件下での計算結果であると記憶する */  
    return r;  
  }  
}
```

図 9.4: 拡張されたメモ化

9.6 適用問題を広げるための拡張

9.6.1 メモ化のための入力に関する適用問題拡張

A*アルゴリズムなどを実現する場合、部分タスクを決定する入力と同じだが、許容コストの上界など計算に必要なパラメタが異なるような計算を行いたくなる場合がある。このパラメタも入力の一部だと考え、パラメタが異なれば違う部分タスクであると捉えることもできるが、パラメタを入力とは別扱いすることで部分計算の結果を再利用することが可能な場合がある。これはいわば、メモ化の時に記憶する入力の他に、メモする計算結果を指定するために「許容コストの上界」という入力を用いる、という拡張されたメモ化を提供しているとも言える。

図 9.4 に拡張されたメモ化アルゴリズムの概念を示す。条件を示す入力 c が新たに加えられており、入力 x に対して記憶した計算結果が c の下で使えるかどうかを判定し、使えるようならばそのメモを使う、というアルゴリズムになっている。メモ化により記憶しているのは、1 つの入力 x に対し、結果 r と条件を示す入力 c の組を 1 つだけである。 r と c の組を計算するたびに全て記憶していれば、それは x と c の両者をメモ化のための入力として使用している事になり、分離した意味

```

a_star_search(Board& b, int cost)
{
    if (is_leaf(b)) {
        return found_result;
    }
    if (cost + H > threshold) return false;

    foreach(next_board = next_regal_board(b)) {
        a_star_search(next_board, cost + 1);
        if (found) return found_result;
    }
    table_store(b, cost);
    return false;
}

```

図 9.5: タスク指定の拡張が必要なプログラム例

が薄れる。このため、記憶しているものは最後に計算した時の (r,c) である。このアルゴリズムがうまく適用できるためには、問題の計算結果 r と条件 c の組が、何らかの意味で順序関係を持つような多数の組からなっており、より優越する側の計算結果を優越しない側の計算結果としても使用することが可能であるような性質を持っていなければならない。いわば、部分タスクの計算結果が「荒い解」から「精密な解」までさまざまな条件下で求められるようなものになっていて、荒い解が必要な時に精密な解の方を使っても差し支えないような、そんな問題でなければならない。

このような問題設定は、A*やゲーム木探索問題によく見られる。ゲーム木探索では、許容される計算リソース内でできるだけ正確な探索結果を求めたいとするような問題設定がなされるため、必要とするものより精密な部分解が求まっていればそれを再利用することは差し支えない場合が多い。

その他のアルゴリズムについても、より精密な解へと探索を進めて行くような問題の定式化ができれば、このフレームワークが適用可能になる場合もあると考えられる。

9.6.2 A*アルゴリズムへの適用例

15 パズルのプログラムを例に、どのような問題への適用が可能になるのかを説明する。

図 9.5 は、反復深化 A*法のプログラムの骨組みとなるコードである。15 パズルを念頭に考えると、この関数は盤面 b と初期状態からの距離 $cost$ を受け取り、 $threshold$ で定められた閾値以内の深さで解が存在するかどうかを求めるものである。現在の盤面において、初期状態からの $cost$ と、ゴール状態へのコストの下限見積もり H の和が閾値を越えていれば、探索をあきらめる。

この場合、 $cost$ が図 9.4 の入力 c に相当するものとなる。 a_star_search 関数が提案手法のタスクに相当すると考えたとき、入力は盤面 b と初期状態からの距離 $cost$ である。この両方ともをメモ化のための入力であると考え、初期状態からの移動距離が異なるだけで別のタスクとして扱われてしまうことになる。しかし、より $cost$ が低い状態、すなわちより制限が緩い状態で探索

```

result = table_read(b);
if (enough_result(result, parent_cost)) return result;

START:
cost = min(parent_cost);

a_star_search(Board& b, int cost)
{
    if (is_leaf(b)) {
        return found_result;
    }
    if (cost + H > threshold) return false;

    foreach(next_board = next_regal_board(b)) {
        a_star_search(next_board, cost + 1);
        if (found) return found_result;
    }
    table_store(b, cost);

    return false;
}

if (!enough_result(parent_cost)) goto START;

```

図 9.6: 本フレームワークでの動作の擬似コード

を行った結果がすでに求められていたならば、より制限が厳しい状態では同じ結果を再利用することができるはずである。このような場合に、盤面 b のみをメモ化のための入力とし、 $cost$ を条件 c という扱いの違うものに変更することで、この結果の再利用を可能にすることができる。

図 9.6 は、図 9.5 のプログラムを提案するフレームワーク上に移行したときの骨組みの擬似コードである。

まず、`table_read()` でこれまでに計算した部分計算結果がないか調べ、もし、親タスクの条件 `parent_cost` で十分な解が得られていたならば、その解を使おうとする。過去の解が不十分だったとき、親タスクの条件の中で最も厳しい条件を `cost` として採用し、図 9.5 と同じ `a_star_search()` 関数を呼ぶ。得られた結果について、全ての親の条件に対して十分な結果が得られたかどうかを調べる。十分な結果であると判断される場合は親にその結果を返す。しかし、もし不十分だと判断される場合は、条件を設定し直して再度 `a_star_search()` 関数を呼び直す。という動作を繰り返す。

このように、 x の他に c という分離された入力を用いることで、A*アルゴリズムで重複するタスクを検出し、部分計算結果を再利用することが可能になる。

なお、現実には 15 パズルを解くためには探索木のループを検出するためのやや複雑な機構が必要になる。これに対応するためには、新たな親ノードが加わったときにループを検出し、必要に応じ

てループ以降の探索をあきらめるなどの動作を行う必要がある。これは問題ごとに対応が異なるため、親ノードの追加時にフレームワークから呼ばれるユーザ定義関数を用意し、その内部に適切な処理を記述してもらうことにした。

第10章 関連研究

10.1 チェックポインティング

耐故障性を実現するためのひとつの手段として、メモリや I/O 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 [28]。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はオーバー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[29] や Cactus[30] などの並列計算システムにも使われてきた。

しかし、チェックポイントは基本的に、故障前の構成と全く同じ構成を後から再現することを目的としており、計算ノードが 1 台壊れたときにそのまま、1 台少ないまま動き続けるといったことはできない。

提案手法においても、個々の計算ノードにおいて独立にチェックポイントを用い、計算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。故障時にチェックポイントを用いて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。

10.2 Embarrassingly parallel

完全に部分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場合には、極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。例えば、SETI@home[31] 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 を多数使い、依存関係のない部分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[32] であり、現在同様なプロジェクトに多数用いられている。

また、Google map-reduce[33] 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。

10.3 Satin, Cilk

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[34] が挙げられる。Cilk は work-stealing によって動的に負荷分散を行い、効率のよ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[35] や Satin[36] がある。特に Satin では、計算ノードが故障した時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 [37]。

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。第 8.3 章で述べたように、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言える。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。

また、Satin の発展系である Satin++[38] では、より部分計算間での情報共有を意識したフレームワークを提案している。Satin では Divide-and-Conquer 型のタスクを対象としていたが、Satin++ では Divide-and-Share というキーワードを使い、対象となる問題領域を広げようとしている。Divide-and-Share とは、基本的に Divide-and-Conquer 型の依存関係を持つようなタスクを対象としているが、タスク間で共有できるようなオブジェクトを導入し、他の計算ノードで変更されたそのオブジェクトの更新が終了するまで待つ、などの機能を導入することで、タスク間の情報共有をしやすくしたものである。

Satin++ では、分枝限定法を例に Divide-and-Share の有効性を示している。分枝限定法の場合、探索途中での暫定解の情報を並列計算ノード間で共有することができれば、枝刈りの効率が大きく改善されて探索時間が大幅に短縮される。このように、比較的少量の限定された情報を共有することで計算速度を向上させられるような問題はいくつも存在する。

ただし、Satin++ の枠組みでは、共有データは全ての計算ノードに存在するデータであるとされており、その結果、あまり巨大な共有データは扱うことができない。Satin++ の思想は、少量に限定された共有データを用いることで、計算効率を向上させようというものである。しかし、分枝限定法においても、一部が確定した暫定解のコストを、その確定部分が共通である部分問題に限って適用して枝刈りをする、という手法を取ることができ、これは、グローバルな暫定解による枝刈りよりもさらに制約条件が厳しくなるため、より効率の良い探索ができる可能性が高い。このような情報共有は、提案している分散ハッシュテーブルによる方式を用いれば、実現することが可能である。提案手法は、全てのタスクが共有されることを前提としており、全ての計算ノードの記憶容量の総和分だけ共有データを保持することが可能である。つまり、提案手法は Satin++ より広い範囲の情報共有が必要な問題を念頭に置き、より共有する情報の多い計算を実現しようとしている。

10.4 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[39], pastry[40], tapestry[41] 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、問題の部分計算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。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。

10.5 DHT driven scheduling

ゲーム木探索の分野においては、本手法がとった、部分計算を再利用しつつ並列計算する手法として、transposition table driven work scheduling[42] がすでに提案され、クラスタ構成などの並列環境において高い効率で並列探索が可能であることが示されている。

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、重要な要素であると考えられる。

また、本手法は再実行によって耐故障性を実現しようとするものであるが、ゲーム木探索においては、部分計算の再現性がない場合も多い。実際のゲーム木探索においては、さまざまなヒューリスティックを使って探索順序をコントロールし、計算量が爆発することを防いでいるため、探索順序によってある盤面の評価結果が変化してしまうことがよくある。しかし、トランスポジションテーブルを使った探索を行う場合、以前十分な計算リソースを使って探索した結果があれば再利用する、という方式を採っているならば、このような部分問題の再現性のなさ、それに伴う探索木の変化の影響を受けることは避けられない。そのため、部分問題の探索結果に一貫性がない場合でも問題全体として見れば解に影響を受けないような工夫、もしくは解への影響が小さくなるような工夫を行って、これに対処してあることが多い。

このように、ある部分問題の計算結果に再現性がない場合でも、その影響を受けにくくするような工夫をすることが可能な問題領域は多いと考えられる。部分問題の計算結果が厳密な意味で一意に定まらなくても、例えば何らかの意味で「より厳密な解」「より緩やかな解」のような順序関係を付けることができ、解がより厳密なものへと単調に変化しているようなものであれば、本手法の枠組みはそのまま適用できると考えられる。

10.6 Phoenix

Phoenix[22] 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。問題の部分部分がある決まった範囲にある仮想ノードに割り付け、仮想ノード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や相性が良いため、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。

第11章 フレームワーク設計

11.1 各ノードのアルゴリズム

11.1.1 計算中

図 11.1 は各計算ノードの通常時の動作を示した擬似コードである。以下、動作を説明する。

各計算ノードは計算途中の部分タスクの集合と、計算が終わった部分タスクの結果の集合を保持している。部分タスクは実行可能状態と結果待ち状態の 2 状態をとる。計算ノード間では、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。

計算ノードは、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、結果を計算ノードの集合に追加して、タスクは消える。子タスクの結果待ち状態に移行した時は、計算途中部分タスク集合にそのタスクを戻す。そしていずれの場合も、実行可能タスクの選択へと戻り、この手順を繰り返す。

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つたとき、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。

タスクの計算結果集合 R

実行可能タスク集合 T_{exec}

結果待ちタスク集合 T_{wait}

メッセージ: query, answer の 2 種

Loop:

```
if ( $T_{exec}$  が空でない) {
     $T_{exec}$  からタスク  $t$  を取り出す ( $t$  は計算途中の状態にある)
    (わかっている子タスクの結果などをもとに)  $t$  の計算を続行する
    if (結果が求まった) {
         $t$  の全ての親タスク  $t_{parent}$  に対し、answer メッセージで結果を返す
         $R$  に結果を登録する
    } else {
        子タスク  $t_{children}$  を生成する
        全ての  $t_{children}$  に対し、query メッセージを投げる
         $t$  を  $T_{wait}$  に入れる
    }
}
if (query メッセージを受け取った) {
    if ( $R$  に結果が存在) {
        結果を answer メッセージを使って返す
    } else {
        対応するタスク  $t$  を生成して  $T_{exec}$  に追加
    }
}
if (answer メッセージを受け取った) {
    if ( $T_{wait}$  に対応するタスク  $t$  が存在) {
         $t$  に子タスクの結果を登録する
        if ( $t$  が必要としていた子タスクの結果がそろった) {
             $t$  を  $T_{exec}$  に移動する
        }
    } else {
        メッセージを無視する
    }
}
goto Loop;
```

図 11.1: 計算中のアルゴリズム

タスクの計算結果集合 R
実行可能タスク集合 T_{exec}
結果待ちタスク集合 T_{wait}
メッセージ: query, answer の 2 種

故障発見 (失われた ID の領域 F):
DHT を再構成する (F をどこかの計算ノードが担当する)
forall (T_{wait} 中の t) {
 if (t の待つ t_{child} が F に存在) {
 $t_{child} \rightarrow$ query メッセージを投げる
 }
}

図 11.2: 故障発見時のアルゴリズム

11.1.2 故障発見時

図 11.2 に故障発見時のアルゴリズムを示す。

何らかの手段で計算ノードの故障が検出できたとき、まず残った計算ノードで分担範囲の割り当てをし直し、分散ハッシュテーブルに欠けた部分がないようにする。この手順は様々なものが研究されているが、どのような手順であってもここでは問題ない。

その後、おのおのの計算ノードにおいて、計算中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。query メッセージは、故障した計算ノードの範囲を引き継いだ新たな計算ノードに届き、そこで必要な部分計算を再計算することができる。再計算は、図 11.1 の通常時のアルゴリズムで query メッセージを受け取った時の動作で実現される。

故障した ID 範囲 F に属していた計算途中タスク T_{exec} は全て query メッセージによって再実行されるが、 T_{exec} の子タスクは故障していない計算ノードに存在している可能性が高く、子タスクの結果や途中状態が保存されていると考えられる。このため、あまり計算時間を必要とせずに失われたタスクを回復することが可能であると期待される。

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。これはまた、故障検出機能が必ずしも正確に故障を検出できなかった場合の復旧にも役立つ手法である。

また、今回はノード故障はある時点で動作が停止してしまうものを想定したが、スナップショット技術などを用いると、故障の少し手前のノードの状態を再現することが可能な場合を考えることができる。この場合、故障した範囲を新たに分担したノードに、故障ノードが少し前の時間までに保持していた部分計算の途中状態や結果を送り、それを新たな分担ノードの保持する部分計算にマージして、途中結果の消失部分を減らすことが可能になると考えられる。

11.1.3 参加

新たな計算ノードが参加する際、DHT の制約を維持したままどこかの計算ノードの担当領域の一部を分けてもらい、自分も DHT の一部になる。計算ノードが DHT の一部に参加した後で、元々その領域を担当していた計算ノードは、その領域に割り振られている計算途中のタスクや計算結果を、新しい計算ノードに転送する。これは計算の途中結果をなるべく保持したまま構成変更することで、失われる途中結果をなるべく少なくしようとするものであり、元々の計算ノードが持っていた情報を全て新しいノードに移動してから新しいノードが計算を開始すれば、計算の途中結果は全く失われない。一方、この途中結果の転送通信が行われなかったとしても、問題全体の計算の正当性には影響はない。故障時に情報が失われたのと同じ状態になるだけであるので、耐故障性を実現するための機構が働けば、失われた途中計算のうち必要なものだけを再計算して、全体の計算を正しく続けることができる。つまり、この計算途中結果の転送は、計算の性能を向上させるためにのみ必要な手順であるので、転送によって得られる高速化効果と、転送に必要な通信データ量や通信に裂かれる CPU 資源のコスト、失われた途中結果の再実行によって増える余分なコスト、および耐故障のためのシステムが働くことによるコストなどを比較して、適切な程度の情報のみ転送できるように実装時点で工夫する必要がある。

11.1.4 脱退

計算ノードの脱退時も、参加時とほぼ同じような手順を取る。すなわち、DHT の制約を保ったままハッシュ表の担当領域を別のどこかの計算ノードに委譲し、どの領域も担当していない状態にする。この手法はやはり様々なものが考えられるが、どのような手法を用いても動作の正当性には影響しない。

その後、以前自分が担当していた領域を委譲した計算ノードに対して、それまで自分が持っていた部分計算の途中経過や結果などを転送する。これも、参加時と同じように計算効率を上げるためにのみ必要な手順であり、転送がなくても計算全体の正当性には影響しない。どの程度の情報を転送するかは実装時の問題となる。

11.2 フレームワークの API

11.2.1 基本 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、部分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。ユーザが定義すべき型とそのインタフェースを、図 11.3 に示す。ここでは、C++ のクラスとしてユーザ型を定義した場合を示している。なお、User_cond 型は第 9.6 章で説明した、適用問題を広げるための拡張に必要なデータ型である。

このユーザ定義型を使って問題を解くために、ユーザはアルゴリズムをある決まったインタフェースに当てはめて書く必要がある。ユーザプログラムを実装するためのインタフェースを図 11.4 に示す。

これらのインタフェースのうち、特に重要なのは、部分問題の解の計算を担当する do_partial_task 関数である。部分タスクの計算を行う際、フレームワークはユーザ定義の部分タスク計算用関数 do_partial_task に、User_key と User_task、および部分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ーを渡して呼び出す。シーケンシャルナンバーは、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全ての結果がそろそろまで待つ、どれかひと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。

11.2.2 適用問題を広げるための拡張 API

第 9.6 章で述べたような問題にこのフレームワークを適用するために、User_key とは別に 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。User_cond はメモ化のための入力ではなく、それ以前に計算されていた User_result が再利用可能かどうかを判断するための入力となる。図 9.5 に示した A* アルゴリズムでは Board が User_key、cost が User_cond に相当するものとなる。

query メッセージには User_key と User_cond が含まれることになる。ユーザは、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、すなわち図 11.4 の enough_result() を定義する必要がある。つまり、現在の解が十分満足できる質であるかを評価する関数が必要、ということである。要求を満たしている場合はそのまま現在の User_result が answer メッセージを用いて返され、満たしていない場合は新しい User_cond で部分タスクを再実行することになる。

なお、図 11.4 の need_reply_parent() は、A* アルゴリズムを実装する際に探索木のループ検出が必要になった時に必要となる API である。新しい親タスクから query メッセージが届いた時、この関数がフレームワークから呼ばれる。ユーザはこの関数内でループ検出を行い、必要ならば親タスクの一部に answer を返すことができる。

```

class Tkg_user_key
{
public:
    void serialize(Tkg_msg_buf& buf) const;
    void unserialize(const Tkg_msg_buf& buf);
    bool operator<(const Tkg_user_key& r);
    bool operator==(const Tkg_user_key& r);
    tkg_task_key_t key() const;
    Tkg_user_key();
};

class Tkg_user_cond
{
public:
    Tkg_user_cond();
    void serialize(Tkg_msg_buf& buf) const;
    void unserialize(const Tkg_msg_buf& buf);
    bool operator==(const Tkg_user_cond& r);
};

class Tkg_user_result
{
public:
    void serialize(Tkg_msg_buf& buf) const;
    void unserialize(const Tkg_msg_buf& buf);
    bool operator==(const Tkg_user_result& r);
    Tkg_user_result();
};

class Tkg_user_task
{
public:
    Tkg_user_task();
    Tkg_user_task(const Tkg_user_key& k);
};

```

図 11.3: ユーザ定義型に必要なインタフェース

```

class Tkg_user_worker
{
public:
    typedef enum {WAIT_ONE, WAIT_ALL, NEXT_PART, FIRST_PART,
                  FINISH} part_result_t;
    static part_result_t do_partial_task(Tkg_system_func& sfunc,
                                         int part_id);
    static part_result_t join_each_sub(Tkg_system_func& sfunc, int part_id,
                                       const Child_state_t& child_result);
    static part_result_t join_all_sub(Tkg_system_func& sfunc, int part_id,
                                       const std::list<Child_state_t>& child_results);

    static Tkg_user_key root_task_key();
    static Tkg_user_task create_root_task();
    static Tkg_user_cond create_root_cond();
    static void finish_root_task(const Tkg_user_task &utask,
                                  const Tkg_user_result& result);

    /* 適用問題を広げるための拡張 API */
    static bool enough_result(const Tkg_user_key& key,
                              const Tkg_user_cond& cond,
                              const Tkg_user_result& result);

    /* ループ検出機構用 */
    static int need_reply_parent(const Parent_state_t &l,
                                  const Parent_state_t &r,
                                  Tkg_user_result* reply_result);
};

```

図 11.4: ユーザプログラムを実装するためのインタフェース

```

int fib(int k)
{
    if (k < 3) {
        return 1;
    }

    fork r1 = fib(k - 1);
    fork r2 = fib(k - 2);

    wait_all_result;

    return r1 + r2;
}

```

図 11.5: 例とするフィボナッチ関数の並列化概念の擬似コード

11.2.3 ユーザコードの例

以下、フィボナッチ関数を例に取り、ユーザの記述するコードについて説明する。

図 11.5 は、フィボナッチ関数をどのように並列計算するかを擬似コードによって表したものである。fib(k) は、fib(k-1) と fib(k-2) を並列に計算させ、その全ての結果を待つ。全ての結果が出たらその結果を足し、fib(k) の結果として返す。

このプログラムを、本フレームワークに実装すると図 11.6 のようになる。fib 関数の中身は do_partial_task 関数にほぼ全て記述される。最初、シーケンス番号 part_id が 0 の状態で do_partial_task 関数の実行が始まる。入力は sfunc というデータ型にまとめて渡されるので、そこから計算に必要な入力 k を取り出し、もし k が 3 未満であれば、その部分問題の解を 1 に設定して、解が求まったことを示す FINISH を返して終わる。

そうでなければ、fib(k-1) と fib(k-2) に相当する部分問題を作るため、k-1 と k-2 に相当する User_key を作成し、fork_sub というサポート関数によってシステムに並列実行を命じる。全ての子問題の結果を待つので、それを示す WAIT_ALL という値を返して終わる。なお、フィボナッチ関数では User_cond に対応するものは存在しないので、ここでは特に意味を持たないダミーの User_cond が用いられている。

```

Tkg_user_worker::part_result_t
Tkg_user_worker::do_partial_task(Tkg_system_func& sfunc, int part_id)
{
    int k = sfunc.key().k_;

    switch (part_id) {
    case 0: {
        if (k < 3) {
            Tkg_user_result r(1);
            sfunc.set_result(r);
            return FINISH;
        }

        Tkg_user_key child_key1(k - 1);
        Tkg_user_key child_key2(k - 2);
        Tkg_user_cond child_cond;
        sfunc.fork_sub(child_key1, child_cond);
        sfunc.fork_sub(child_key2, child_cond);
        return WAIT_ALL;
    }
    case 1: {
        return FINISH;
    }
    default: {
        assert(0);
        return FINISH;
    }
    }
}

```

図 11.6: フィボナッチ関数のフレームワークへの適用

```

Tkg_user_worker::part_result_t
Tkg_user_worker::join_all_sub(Tkg_system_func& sfunc, int part_id,
    const list<Child_state_t>& child_results)
{
    int ret = 0;
    for (list<Child_state_t>::const_iterator it = child_results.begin();
        it != child_results.end(); it++) {
        ret += (*it).result.r_;
    }
    Tkg_user_result ret_r(ret);
    sfunc.set_result(ret_r);
    return NEXT_PART;
}

```

図 11.7: 結果を待つためのコード部分

子問題の解が全て集まると、`join_all_sub` というユーザ定義関数が呼ばれる。`join_all_sub` の内容を図 11.7 に示す。

`join_all_sub` には、全ての子問題の解が `child_results` というリストで返ってくる。そこで、全ての子問題の解の和を求め、部分問題の解として `set_result` を用いて設定し、次のシーケンスナンバーの実行に進む。

すると、図 11.6 の `do_partial_task` 関数が、`part_id` の値が 1 増やされた状態で再度呼ばれる。フィボナッチ関数の場合、これ以上仕事をする必要がないので、解が求まったことを示す `FINISH` を返して終了する。

もし、全ての子問題の解を待つ必要がない場合は、`join_all_sub` の代わりに `join_each_sub` が呼ばれる。ユーザは其中で暫定解を更新したりといった処理を行うことができる。

以上がフィボナッチ関数の計算を本フレームワークで計算するときの実装の中心になる部分である。

ユーザは他に、図 11.4 にあるように、

- ルートとなるタスクの `User_key` を返す関数 `root_task_key()`
- ルートとなるタスクを作るための関数 `create_root_task()`
- 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 `finish_root_task()`

についても定義しておく必要がある。

また、ユーザ定義型を通信するために、シリアライズ、アンシリアライズのための関数をユーザは定義しておく必要がある。その書き方の例を図 11.8 に示す。ここでは `User_key` に `int`, `double`, `long long` の変数 `i_`, `j_`, `k_` が含まれていたものとする。シリアライズ関数で通信用バッファに `i_`, `j_`, `k_` の順に書き込んだので、アンシリアライズ関数ではバッファから同じ順序で読み込むことで、`User_key` の変数を通信できる。

```

void
Tkg_user_key::serialize(Tkg_msg_buf& buf) const
{
    buf.write<int>(i_);
    buf.write<double>(j_);
    buf.write<long long>(k_);
}

void
Tkg_user_key::unserialize(const Tkg_msg_buf& buf)
{
    i_ = buf.read<int, int>();
    j_ = buf.read<double, double>();
    k_ = buf.read<long long, long long>();
}

```

図 11.8: ユーザ定義型でのシリアルライズ・アンシリアルライズの例

11.3 DHT 内部の Garbage Collection

DHT に保持された部分計算結果や計算途中状態は、メモリが許す限り保存し続けていれば、全ての重複計算を検出して最も効率のよい計算が行える。しかし、問題の規模が大きくなってくると、計算ノードの記憶に全ての情報を保持し続けられない場合も出てくると考えられる。そのような場合には、DHT 内部でもう必要ないと考えられる情報を削除する一種の Garbage Collection を行う必要がある。

まず、何も考えずに情報を消去してよいのは、すでに解いてしまった部分計算の結果である。これは、その計算結果を必要としていた親タスクには一度は結果を返答してしまったものであり、将来同じ部分計算を要求されたときのために保持している情報にすぎない。もしこの結果情報を失ってしまっても、後から本当にその情報が必要になったならば、再度実行すればよい。よって、この部分計算結果については、何も考えずに全て消去してしまっても計算全体の正当性には影響を与えない。

ただし、計算効率には大きく影響を与えられとされる。もちろん適用している問題領域にも大きく依存するが、どのような計算結果を残してどれを消すのか、再計算する際に必要となる計算量や、最後に結果が参照されたのはいつ頃であったのか、などの特徴を用いて、計算効率を下げない情報消去の仕方を考える必要がある。これは実装の際に考えるべき問題である。

次に、計算途中のタスクの情報についてであるが、これも部分的には消去してしまってもかまわない。耐故障性があるため、本当に必要なタスクは再度実行されるからである。ただし、むやみに消去すると全体の計算がいつまでたっても終わらない状態に陥る可能性がある。そのため、以下に示すような GC アルゴリズムを用いる必要がある。

GC のために、各タスクには、そのタスクが問題のタスク木のどの位置に存在しているのか、を示すシグネチャ情報が追加される。シグネチャは、 $[s_1 s_2 \dots s_n]$ という整数列によって表現される。

シグネチャ内の整数 s_i は、ルートタスクからの深さ i のタスクの、 s_i 番目の子タスクから生成されたタスクである、ということを示している。つまり、親タスクのシグネチャが $[s_1 s_2 \dots s_n]$ であったとき、その j 番目の子タスクのシグネチャは $[s_1 s_2 \dots s_n j]$ となる。

シグネチャは、深さ優先で探索したときの順序に相当する順序関係を定義することができる。つまり、複数のシグネチャを比較して、どちらが深さ優先探索で早く探索されるべきタスクであるかという順序づけが行える。以下、早く探索される方のシグネチャを「小さい」という大小関係で表現する

タスクは、生成されたときにシグネチャを付けられ、また、親タスクから結果を必要とされるたびにシグネチャを更新される。複数の親タスクの子供になっているタスクの場合、最も小さいシグネチャの親から自分のシグネチャを決定することになる。もし、新たな親タスクが現れるなどして自分のシグネチャが変更された場合、自分の子タスクにもその変更を通知して、子タスクたちのシグネチャも変更が必要かどうかを判定させなければならない。

タスクの結果が定まったときには、タスクは計算途中ではなくなるのでシグネチャは消える。もし、一部の子タスクの結果を待たないまま自分の結果が定まったときには、結果がもう必要なくなったことをその子タスクたちに通知する。通知を受けた子タスクは、必要ならば自分のシグネチャを更新する。

このような動作を行い、タスクのシグネチャを正しく保持し続ける。

GC の際には、保持しているタスクのうちシグネチャが大きなものから削除を行う。シグネチャが大きなものは、並列性を引き出すために深さ優先探索の順序に先行して探索されているタスクであり、問題領域によっては投機的な、将来必要なくなるかもしれないタスクだからである。

実装時には、必ずしもシグネチャが大きなものから順番に削除していく必要はない可能性もある。タスクの予想計算時間、これまでにつぎ込んだ計算リソースなどの条件から、削除に適したタスクを選択する手法を検討する必要がある。ただし、シグネチャが最も小さいものに関しては、これを削除してしまってはならない。

タスクを深さ優先で探索したときに必要なスタックの深さ、つまりルートから最も遠いタスクの深さを d とすると、計算ノードに d だけのメモリがあって、シグネチャが小さい方から順に d 個のタスクを残していれば、深さ優先探索の順序での計算は必ず完了する。このような動作をさせるために、シグネチャが小さいもののいくつかに関しては、決して削除してしまってはならないのである。

このようなアルゴリズムにより、計算がいつかは終了するという条件を保ったまま、タスクの GC を行うことができる。

11.4 探索効率向上のために今後拡張していくべき機能

探索効率を向上させるために、いくつかの拡張機能が考えられる。

グローバルな情報 定数表など、全ての計算ノードで必要となるグローバルな情報を簡単に扱いたいという要求がある。そこで、ある決まったノードで作成された後、計算が始まる前にあらかじめ全ての計算ノードに1つずつコピーされるようなユーザ定義のデータ構造をサポートし、これをグローバルな情報としてユーザが使えるようにすることが考えられる。

タスクのスケジューリングやキャンセル 複数の部分タスクについて、どちらを先に計算したらよいかは問題領域に依存する知識を深く活用しなければ決定できない。つまり、タスクのスケ

ジューリングはユーザがコントロールした方が効率の良い計算ができると考えられる。このような観点から、タスクの探索順序のヒントをユーザが簡単に与えられるような枠組みを、例えば優先度を指定できるようにするなどの方法を用いて実現することが必要になる。

また、並列化によって大きな速度向上を得るためには、投機的な実行と、必要なくなったタスクのキャンセル機構が欠かせない。投機的実行はしばしばスーパーリニアな速度向上すら生むことがあり、探索順序のヒューリスティックの信頼性があまり高くないような問題領域では重要な手法となる。投機実行が無駄になると判明したとき、計算リソースが無限に余っている場合はそのまま投機実行を続けさせても問題はないが、実際には、必要がない実行をキャンセルし、必要なタスクの計算に移行させる必要がある。

そこで、このような投機実行、実行キャンセルの仕組みも今後必要になってくると考えられる。

このように、利便性を向上したり計算効率を高めるためにさまざま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。

第12章 シミュレーションによる評価

実装を行なう前に、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。

12.1 シミュレーションのためのタスク生成モデル

12.1.1 実アプリケーションのタスク生成

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、重複する部分計算に出会う様子をうまく表現できなければならない。図 8.3 や図 8.4 で示したように、共通部分問題の出現の様子は問題領域によって異なってくる。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 12.1 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、探索木の中でルートから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。削減率の平均をとる際には、ルートからの深さではなく、(見つかった解の深さ - ルートからの深さ) が等しい部分問題をまとめて、その平均をとることにした。これはいわば、探索の残り深さであり、子問題の大きさがほぼ等しい物どうしの平均をとっていると考えて良い。グラフの横軸には、この残り深さが示されている。なお、ルートからの深さが 2 以下の子問題については平均から除外してある。例えば深さ 1 の部分問題は決して同一のものが生じないように、ルートに近い部分問題は他の深さとは大きく異なった性質を示すと考えられるからである。グラフを見ると、ほぼ全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わかる。

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同士の間では部分問題の共有は起きにくい。

12.1.2 タスク生成モデル

このような実アプリケーションの振る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) という二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 によって取り得る値が変化する。ルートタスクの子タスクの k は、 $[0, MAX)$ 上に均等に取られた m 個の点の値のみ取ることができる。つまり、 i を整数として $k = MAX \times i/m$ という値のみ取ることができる。 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、 $m_{d-1} = b m_d$ であ

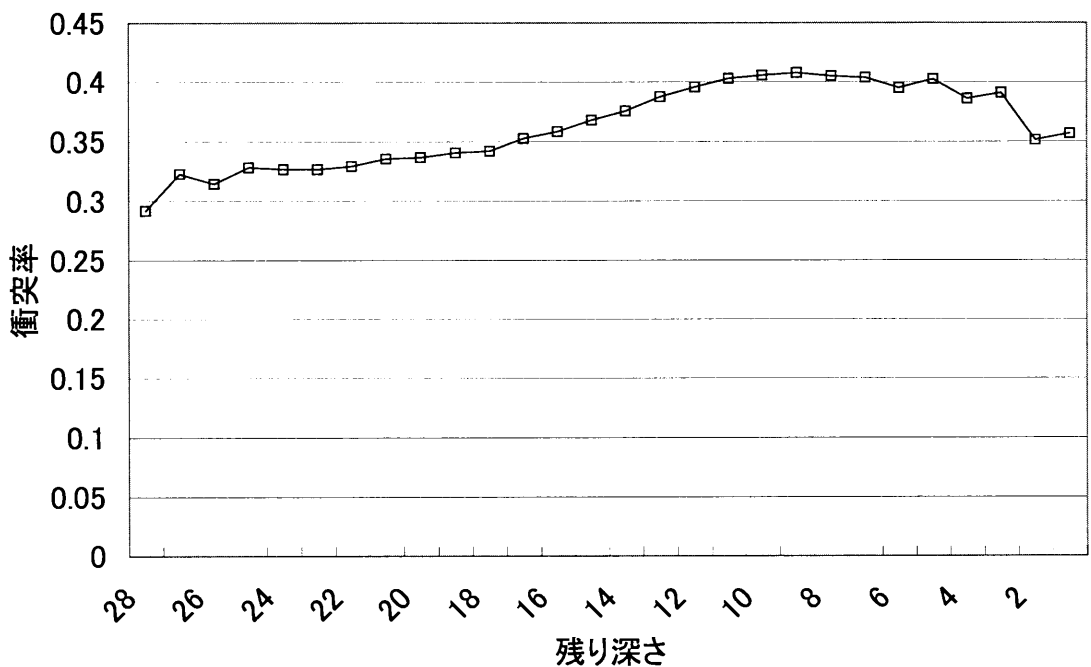


図 12.1: 15 パズルでの子問題削減率

る。ある大きさのタスク数はおよそ b^* 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 個の値を取る物とする。 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。

以下、15 パズルをこのモデルでシミュレートする場合を例に取る。ルートタスクは $(MAX/2, d)$ で表される。15 パズルの子タスクの平均生成数は 3 であると実験から求められたので、ルートタスクの子タスク $(k_c, d-1)$ の k_c は m 個の候補点からランダムに 3 個選んだものとする。なお、ここでは $m = 8$ とした。図 12.1 の結果より、子タスクの衝突率はおよそ 0.3、つまり平均分枝数は 2 であるので、次の子タスクの候補点は $8 \times 2 = 16$ 個になる。 $(k_c, d-1)$ の子タスクの k は、 k_c の近傍の $m (= 8)$ 個の点からランダムに 3 個選択することになる。これを繰り返してタスクを生成していく。

図 12.2 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 12.1 と同様にグラフ化した物である。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わかる。なお、図 12.1 と同様にルートからの深さが 2 以下の部分タスクについてはグラフ化していない。

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。

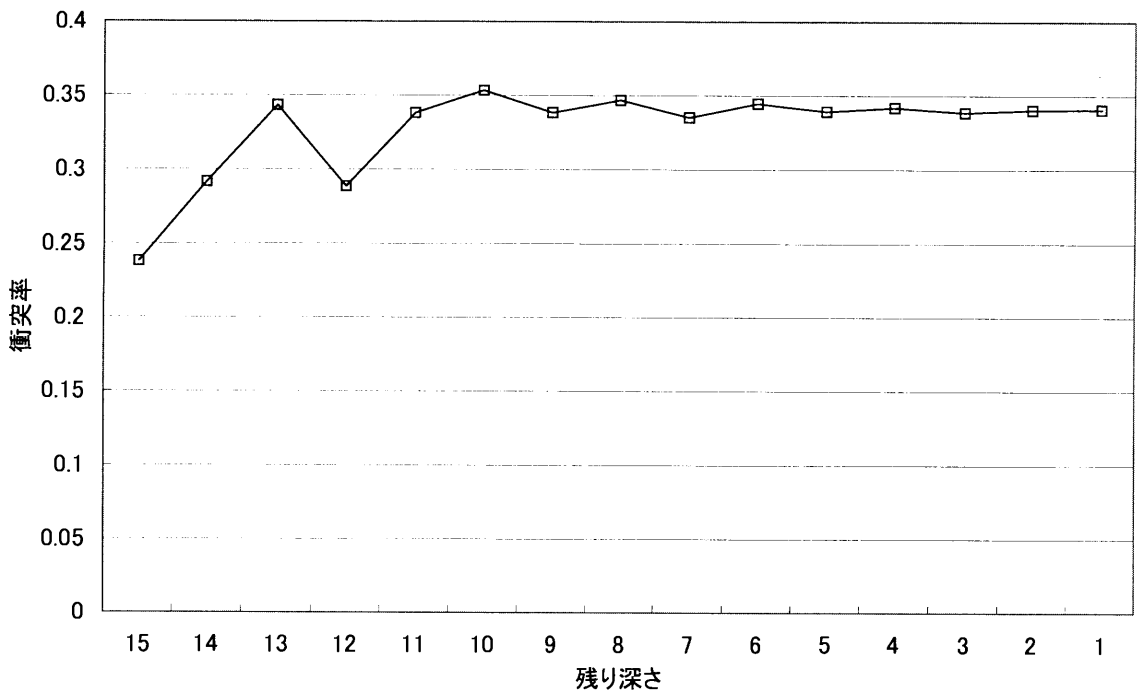


図 12.2: シミュレーションで用いたモデルの子問題削減率

12.2 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。マスタワーカ型の動作は、1 台のマスタノードがルート近くのタスクを実行し、ある一定の大きさの多くの子タスクを準備する。残りの計算ノードはワーカノードとなり、マスタノードに対して子タスクを要求し、その計算結果を返す、ということを繰り返す。暇になったワーカノードが仕事を要求するため、動的な負荷分散が実現される。スケーラビリティについては、マスタに対して通信が集中してしまうためあまり良いとはいえないが、ワーカに送る仕事の粒度を大きくすることで実用的には通信の集中による問題が起きない程度に調整することは可能である。また、マスタワーカの構造を多段に繰り返すことでスケーラビリティを上げることを目指した研究も存在する。耐故障性は、ワーカに送った仕事をマスタが覚えておき、ワーカが故障したことを何らかの方法で確認したら、失われたタスクを別のワーカに送り直すことで比較的簡単に実現される。ただし、マスタが故障した場合は、新たなマスタを作り、マスタの持っていた情報を再構成しなくてはならないため、ワーカの故障より重大な障害になると考えられる。

マスタワーカによる並列化方式の重要な特徴として、ワーカ同士は通信を行わないという点がある。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、ワーカ同士の通信がなされないと、同一の部分タスク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、マスタ内部ではハッシュ表と同様の仕組みを用いて重複する部分計算を把握することが可能であるし、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。

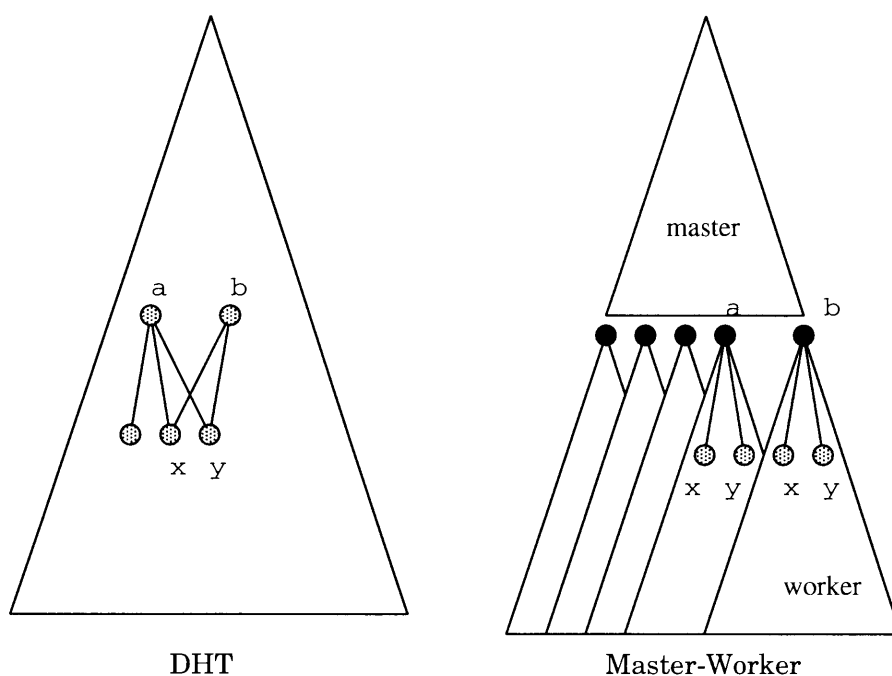


図 12.3: 分散ハッシュテーブル方式とマスタワーカ方式

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上で重要になる。

図 12.3 は、シミュレーションで使用するマスタワーカ方式での分散計算の方法を示したものである。図の左側は提案手法である分散ハッシュテーブルを用いた方法でのタスク木を示している。全ての部分計算はハッシュ表を用いて管理されているため、重複する部分計算は全て発見され、無駄に計算されることはない。

マスタワーカ型の分散計算をシミュレートする際、生成されるタスクは提案手法の場合と同じモデルで生成されるものとされ、また、マスタノード内、ワーカ内のタスクはハッシュ表を用いて管理される。マスタワーカ型の動作は、まずマスタノードがルートのタスクを実行し、ある決まった深さまでのタスクを生成する。図の a、b のタスクがそのようなある決まった深さのタスクであり、マスタはこれらのタスクを生成して、ワーカからの通信を待つ。ワーカは、実行できるタスクが存在しないときに通信を行い、マスタから実行すべきタスクを取ってきて、以降のタスク (図の x、y のタスク) をワーカ内で実行する。マスタから受け取ったタスクの結果が計算されたら、その結果をマスタに送り、新たなタスクを要求する。

分散ハッシュ方式では x、y のタスクの結果は複数の親タスク a、b に必要とされていることがわかり、ただ一度ずつ計算されるが、マスタワーカ方式では x、y の重複が発見できないため、a を計算するワーカと b を計算するワーカとで別々に複数回計算されてしまうことになる。ワーカの数が少ないときには、偶然 a と b を同じワーカが担当する場合があります、そのようなときには x、y の計算は一度ずつですむが、ワーカの数が増えたとこの偶然が生じる確率が下がり、タスクの無駄な重複計算が増えることになる。

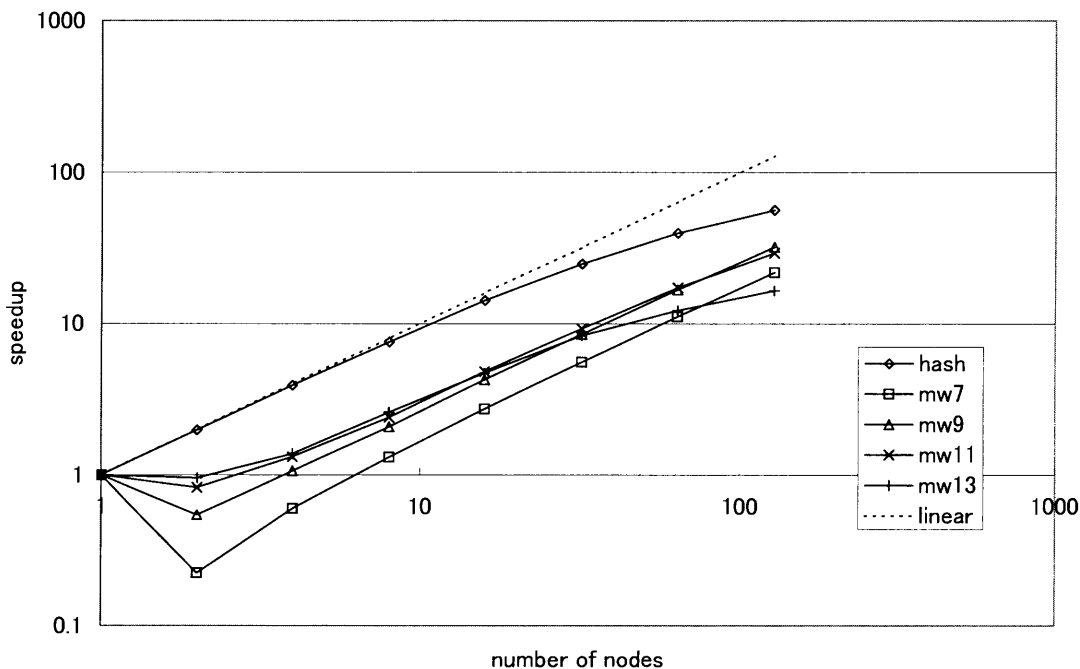


図 12.4: 台数効果 (問題サイズ 18, 末端 10 クロック)

12.3 シミュレーションによる比較実験

12.3.1 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、子タスクの結果が全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。

12.3.2 故障なしの場合

システムの台数効果

図 12.4 と 12.5 は、深さ 18 の同一タスクについて、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 12.4 が末端タスクの実行時間 10 クロック、図 12.5 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。mw の後の数字は、ワーカに配るタスクの大きさを示している。つまり、mw7 は残り深さ 7 までマスタが担当し、残り深さ 7 になったところでワーカに配る場合である。ワーカに配るタスク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

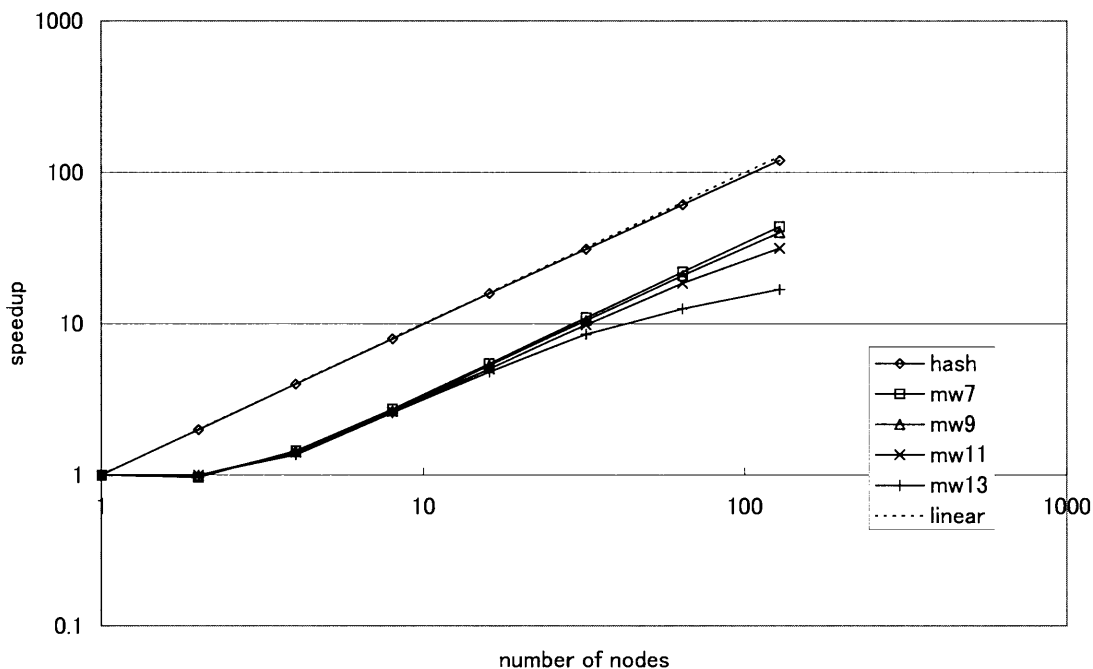


図 12.5: 台数効果 (問題サイズ 18, 末端 1000 クロック)

力がマスタにタスクを要求する通信時間のオーバーヘッドも少なくなるが、負荷の偏りが生じやすくなる。

この実験で、マスタワーカ方式におけるノード数はマスタとワーカを合計したノード数を示している。つまり、マスタワーカでノード数 2 という設定はマスタ 1 台とワーカ 1 台という設定でシミュレーションを行っている。そのため、ノード数の少ない領域では hash と比較するときに台数的にやや不公平な比較になってしまうかもしれないが、台数が多い領域ではほぼ影響は無視できると考えている。

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。

重複タスク検出による計算効率の改善効果

速度向上率が低下した原因は、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 12.6 と図 12.7 は最終的に計算された部分タスクの総数を示したものである。マスタワーカ型で 1 ノードで実行したときのタスク数を 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せず、マスタワーカの 1 ノードでの実行と全く同じタスク数である。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

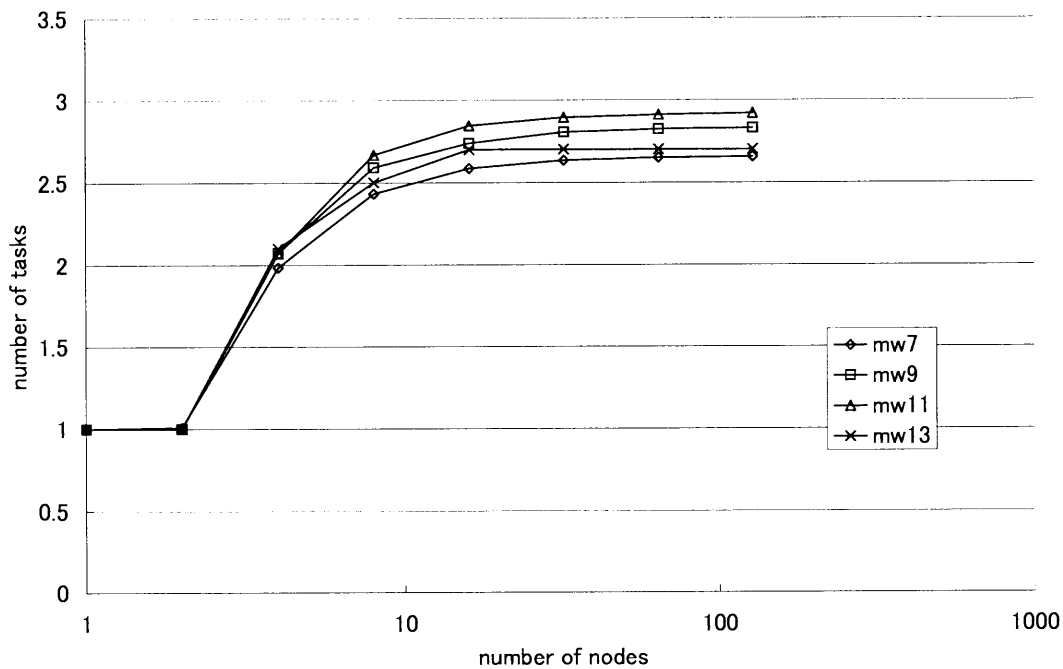


図 12.6: 総タスク数 (問題サイズ 18, 末端 10 クロック)

ため、大きく総タスク数が増えていることがわかる。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。

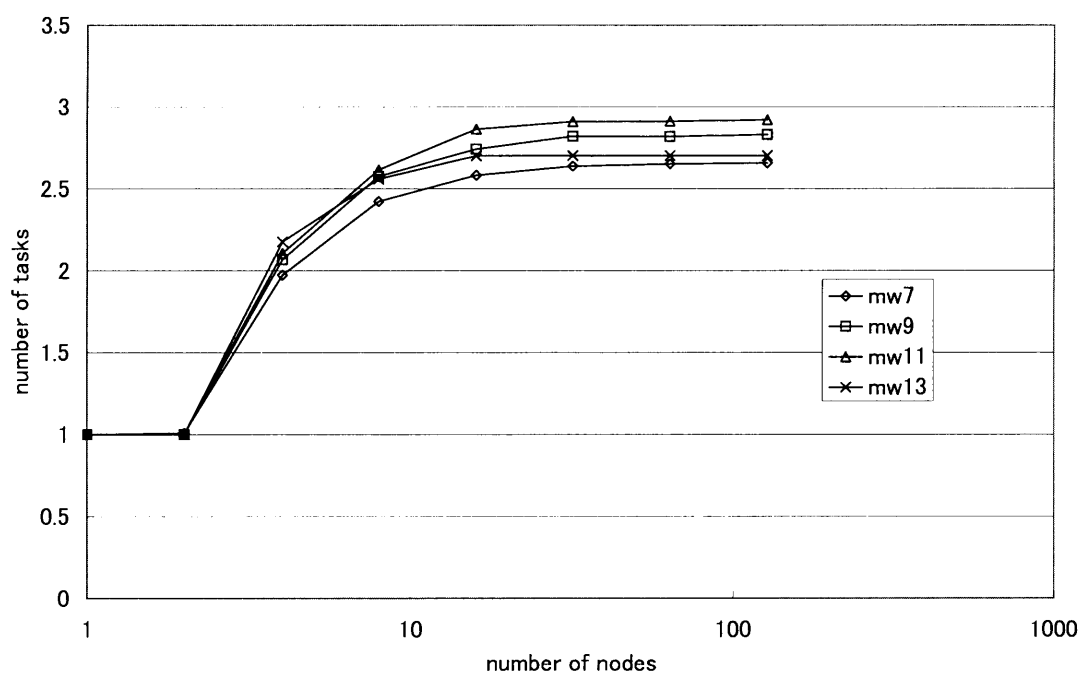


図 12.7: 総タスク数 (問題サイズ 18, 末端 1000 クロック)

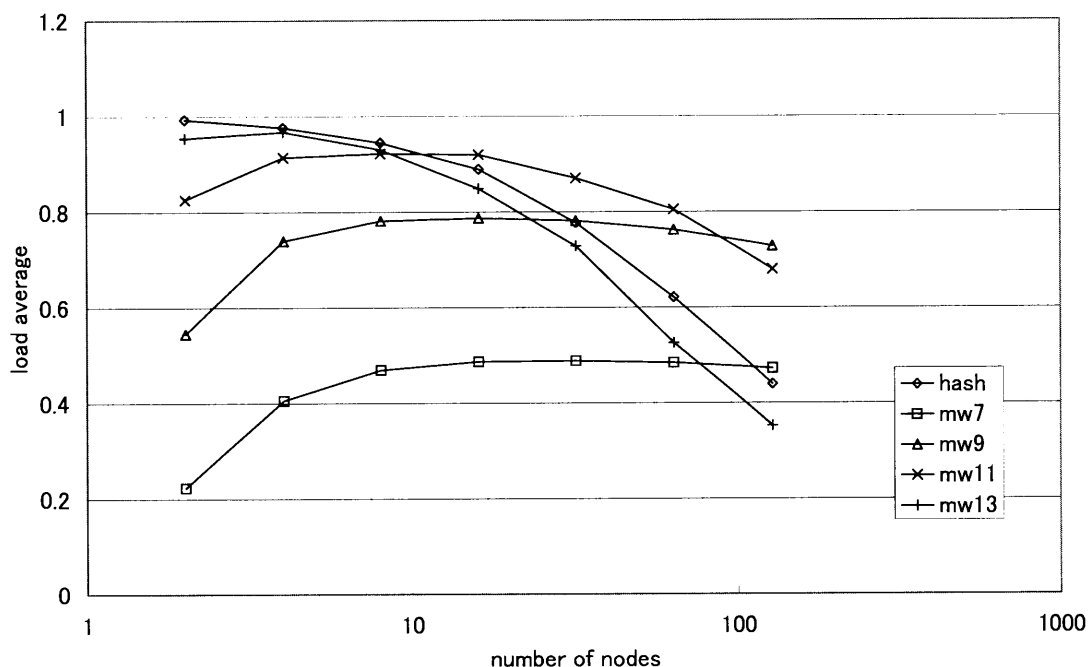


図 12.8: 各ノードの平均仕事率 (問題サイズ 18, 末端 10 クロック)

タスク粒度の変化の影響

総タスク数の増加の影響がある程度で飽和するとき、速度向上率には、ワーカのタスク粒度の違いによる通信ペナルティと負荷不均等が影響してくると考えられる。

そのことが図 12.8 と図 12.9 から読み取れる。図 12.8 と図 12.9 は、各ノードの動作時間中で、実際に計算をしている時間の比を示したものである。まず、末端が 10 クロックの場合、特にワーカの仕事の大きさが 7 の場合の 2 ノードでの実行を見ると顕著のように、ワーカが仕事をしている時間が少なくなってしまう。これは、配られるタスクの通信/計算比が悪いからである。ワーカに配られる深さ 7 のタスクは、平均分枝数 2 ならばおよそ 2^7 個のタスクからなり、末端が 10 クロックなのでだいたい 1280 クロック程度で仕事が終わってしまう。一方、ワーカからのタスク要求とマスタからのタスク配布にはノード間を往復する通信時間の 2000 クロックが必要になるため、ワーカに配られる仕事量に対して、仕事要求にかかる通信時間が明らかに長すぎると考えられる。ワーカノード数を増やすと次第に計算している時間の率が上昇するが、これは無駄な計算を始めることによる見かけ上の仕事率向上である。ワーカの仕事の大きさが 7 のとき、図 12.6 から読み取るとおよそ 2.5 倍程度の無駄な仕事をするようになるが、ワーカ仕事率の向上もおよそ 2.5 倍であり、これに対応している。

ワーカに配る仕事の大きさを 7 から 9、11 と大きくしていくと、仕事率は向上し、総合的に計算にかかる時間も短縮される。末端の仕事にかかる時間を大きくしても (図 12.9) 同じことがいえる。しかし、ワーカのノード数を増やすと次第に仕事時間率が低くなっていく。マスタワーカ型の場合は、ワーカのタスクの大きさを大きくすると仕事時間率が悪くなることから、これは負荷の不均等によるものであると考えられる。一方、分散ハッシュを用いる方は、末端のタスクの大きさが通信時間と同じ程度である場合 (図 12.9) は、参加ノード数を増やしても効率が悪化しないが、末

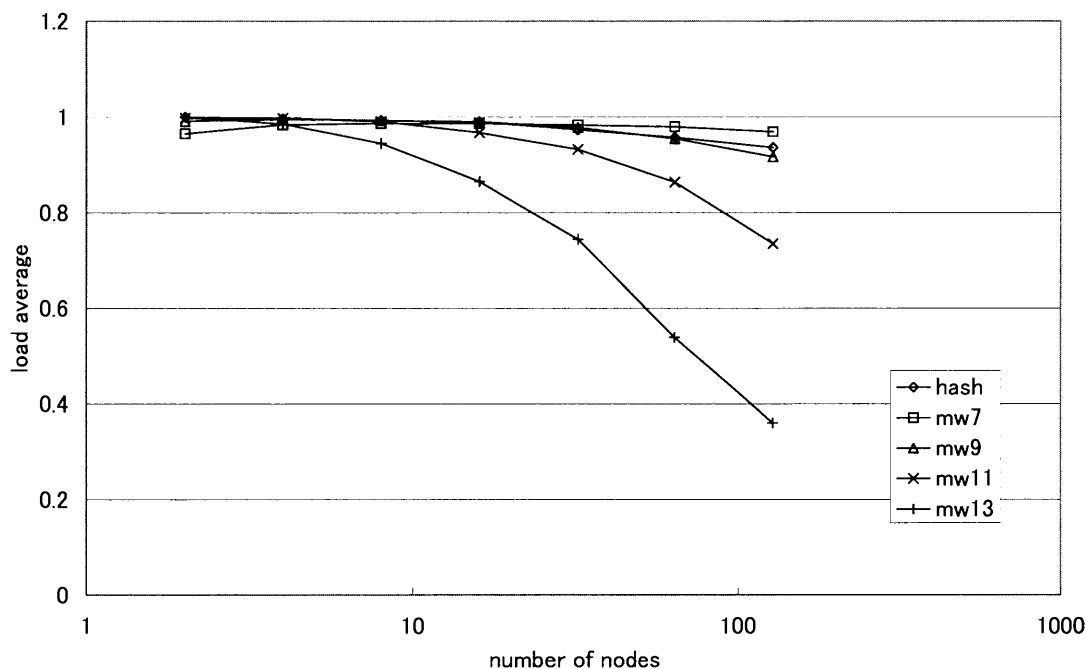


図 12.9: 各ノードの平均仕事率 (問題サイズ 18, 末端 1000 クロック)

端のタスクの大きさが通信より小さい場合 (図 12.8) は効率悪化が見られる。負荷分布は末端タスクの大きさの違いに関係なく同じであることから、この場合、効率悪化は通信待ちの時間が増えていくことに起因していると考えられる。

問題のサイズを 15 へと小さくした場合の台数効果を同様に図 12.10、図 12.11 に示す。無駄な仕事の増加の仕方は図 12.12、図 12.13 に示す。問題サイズ 18 の場合とほぼ同じ傾向を示している。仕事時間の率は図 12.14、図 12.15 に示すとおりであり、末端の仕事量が小さいとき、ノード数が増えると分散ハッシュ方式の仕事時間率がさらに低くなってしまっている。結果、末端タスクが小さく参加ノードが多いとき (図 12.10)、総合的な計算時間としてはマスタワーカ方式の方が早くなっている。

しかし、末端タスクの大きさを通信時間と同程度にとっておけば、同じ依存関係を持つ問題でも図 12.11 のようにリニアな速度向上を見せ、マスタワーカよりはるかに早い計算速度を得ることができる。

このことから、問題のサイズが小さくなくても、末端タスクの計算コストがノード間の通信にかかるコスト程度に調整されていれば、分散ハッシュを用いる提案方式は十分効率よく動作する、と期待できることが判明した。

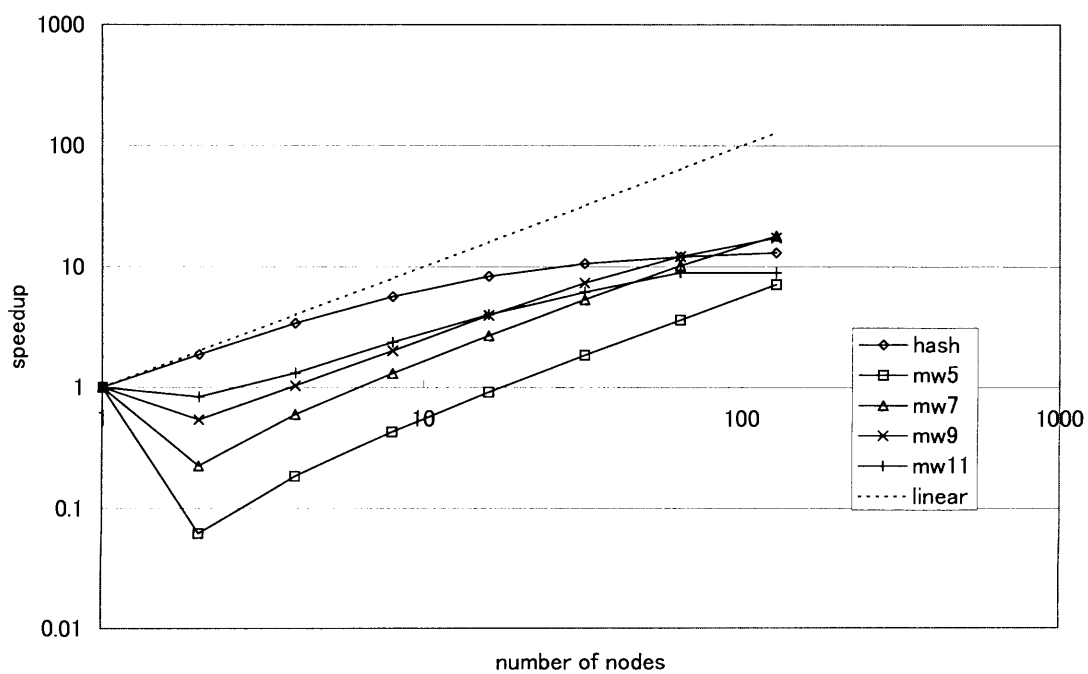


図 12.10: 台数効果 (問題サイズ 15, 末端 10 クロック)

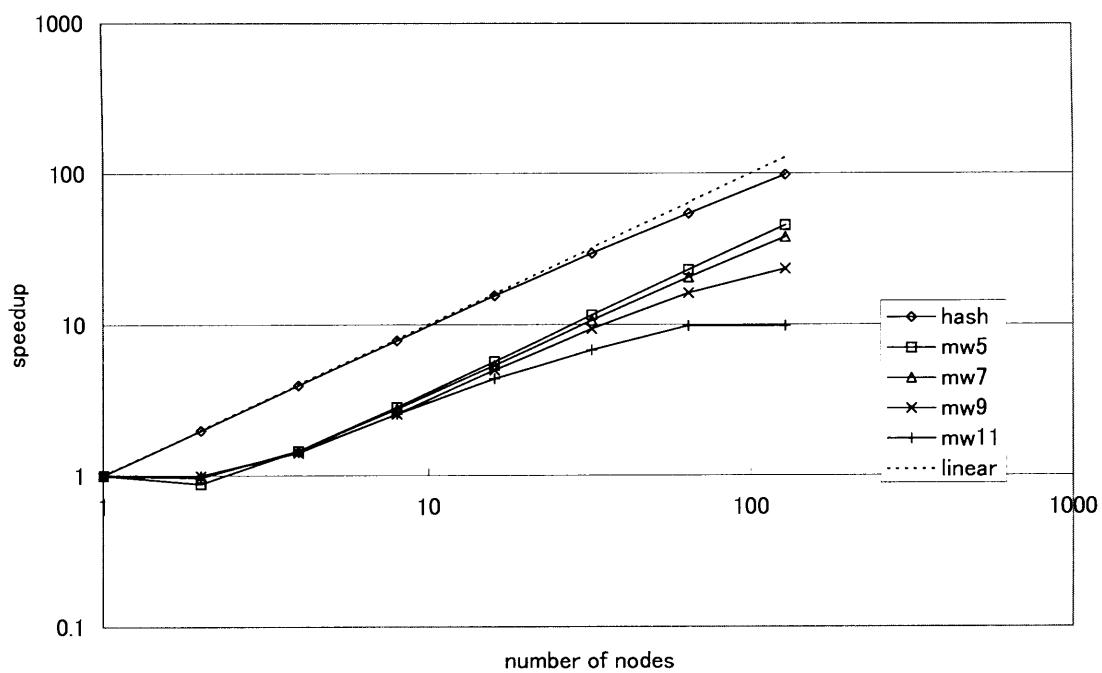


図 12.11: 台数効果 (問題サイズ 15, 末端 1000 クロック)

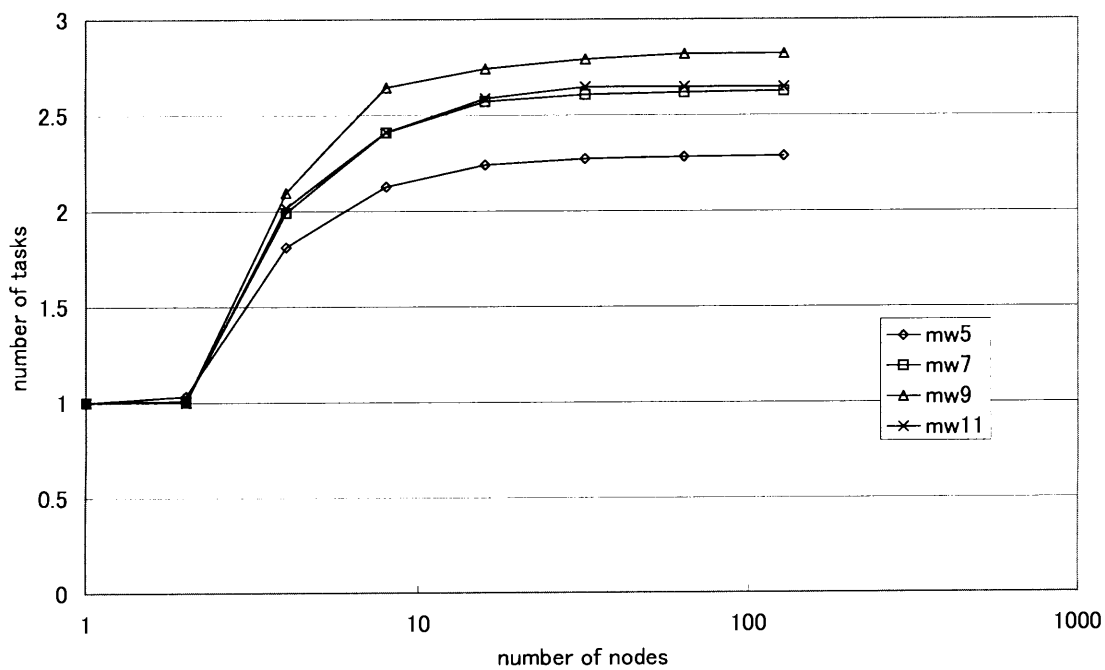


図 12.12: 総タスク数 (問題サイズ 15, 末端 10 クロック)

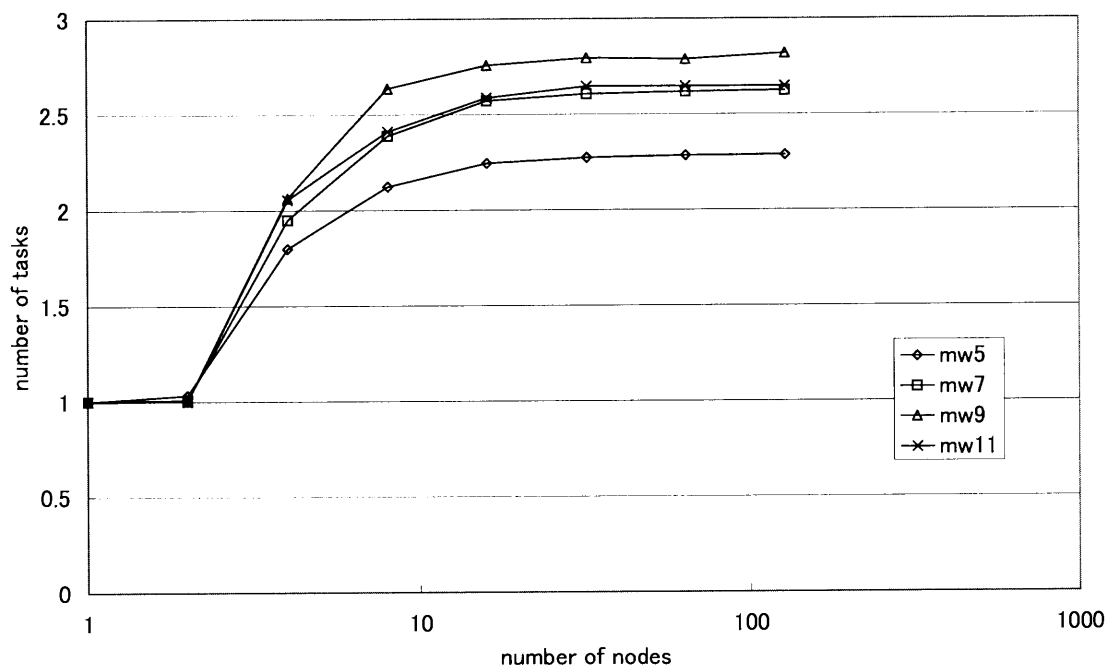


図 12.13: 総タスク数 (問題サイズ 15, 末端 1000 クロック)

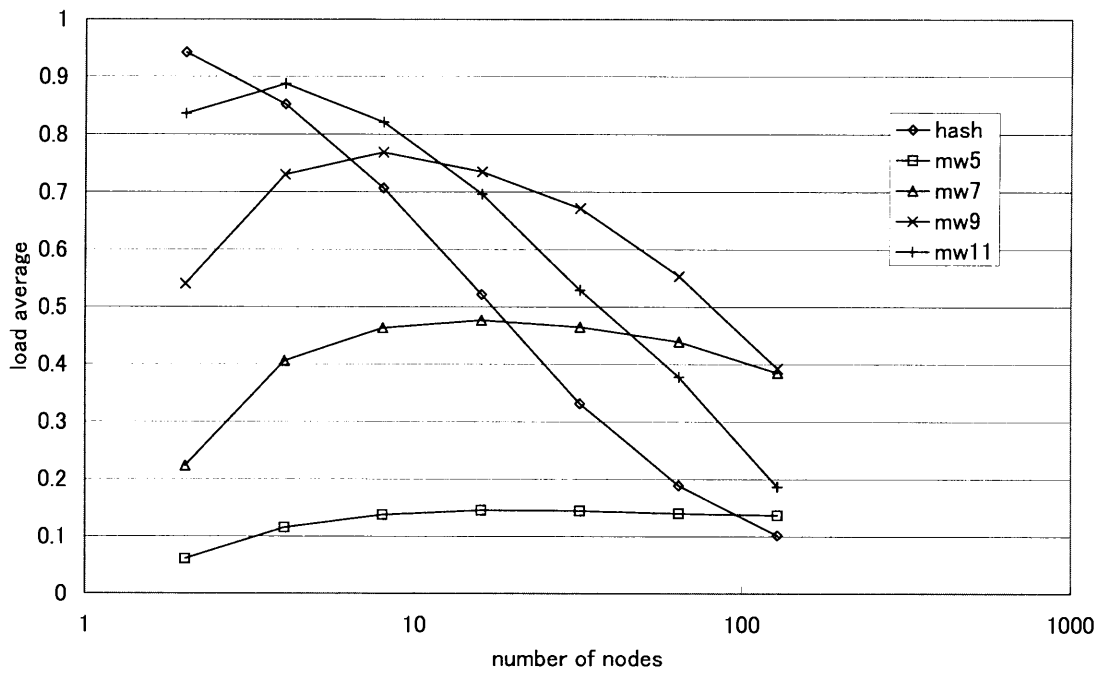


図 12.14: 各ノードの平均仕事率 (問題サイズ 15, 末端 10 クロック)

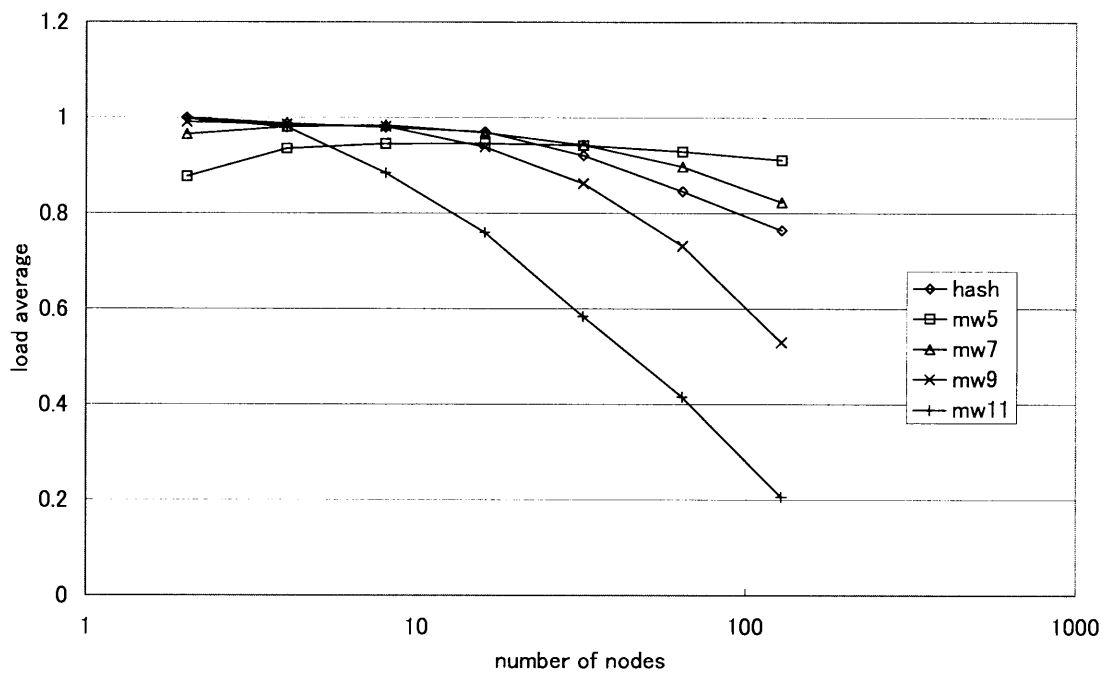


図 12.15: 各ノードの平均仕事率 (問題サイズ 15, 末端 1000 クロック)

12.3.3 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、以下の通りである。

故障ノードは、故障した時点から 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全て通信路に留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全て消去した状態で、再び動作を始める。つまり、全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、全て破棄される。

計算ノードの故障割合が小さい場合

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 12.16 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。横軸はクラッシュさせるタイミングの違いを表している。nocrash がクラッシュなしの場合の計算時間、このクラッシュなしの計算時間を 10 等分し、最初の時点 (つまり 0 クロック目) でクラッシュを起こした場合の計算時間を横軸の 0 で、次の時点でクラッシュを起こした場合を 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クラッシュなしの時間を 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、マスタがクラッシュした場合 (mw9-master)、マスタワーカ方式で、ワーカがクラッシュした場合 (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もので、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。

まずわかるのは、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全くと言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わかる。これは当然の結果で、マスタノードが故障から回復した後、ワーカに割り当てたタスクの結果を全て忘れて、新たにタスクを一から割り当て直すので、故障回復時点よりもう一度最初から探索をやり直すのと同程度変わらない。図 12.16 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。

マスタノードが故障から回復した際、ワーカノードにどのタスクを割り当てていたかを何らかの方法でワーカから教えてもらい、それをもとにすでに割り当てていたタスクを同じワーカに割り当てるような仕組みを作れば、このようなマスタノード故障時の大きな遅延は防げるかもしれない。ただし、このような仕組みを単純に設計すると、故障からの回復時にマスタに通信が一気に集中することになり、現実的ではなくなると考えられる。このような仕組みを効率よく実現するには、ワーカの持つタスク全てをマスタが知るという作業を、通信を時間的に分散させるか階層的にするかなどの何らかの方法を用いて、通信を集中させないようにしなければならない。比較対象として複雑なアルゴリズムを用いても、その妥当性の検証が難しくなるだけなので、ここではそのような

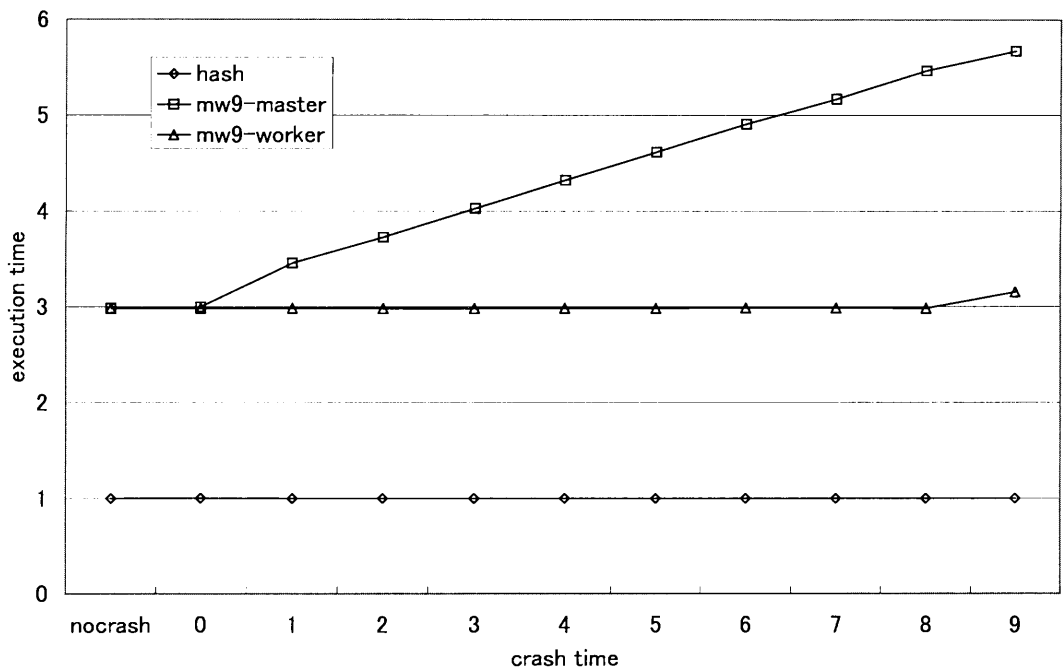


図 12.16: 故障時の計算時間変化 (128 並列、問題サイズ 18, 末端 1000 クロック)

工夫は行わず、ワーカノードへの割り当てをマスタが全て忘れて再計算するというアプローチを採用している。

ここから、マスタスレーブ型の並列化は、耐故障性を実現する上ではマスタの故障が極めて大きな悪影響を与えることがわかる。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。

前述のように、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ため、末端タスクの計算時間を 10 クロックとした場合の同様な実験の結果を、図 12.17 に示す。この場合、分散ハッシュを用いる時に故障すると、故障の発生時点にかかわらずほぼ一定の時間だけ計算時間が伸びるという結果になる。これは、今回のシミュレーションでは分散ハッシュの方に動的負荷分散の仕組みを入れていないためであると考えられる。故障なしの場合の計算時間が 54000 クロック程度、故障ありの場合の計算時間が最長で 88000 クロック程度であり、計算時間の伸びは、故障の持続時間の 30000 クロックにほぼ等しい。よってこの実験では、故障が発生したノードが復帰時に、ほとんどペナルティを受けることなく、計算の続きを開始することができた、とも読み取れる。

また、マスタワーカ方式については、計算の後のほうでワーカが故障した場合に計算時間が長くなる影響が起きることがよりはっきりとした。マスタノードの故障時に大きな影響を受ける点は前の実験と同様である。

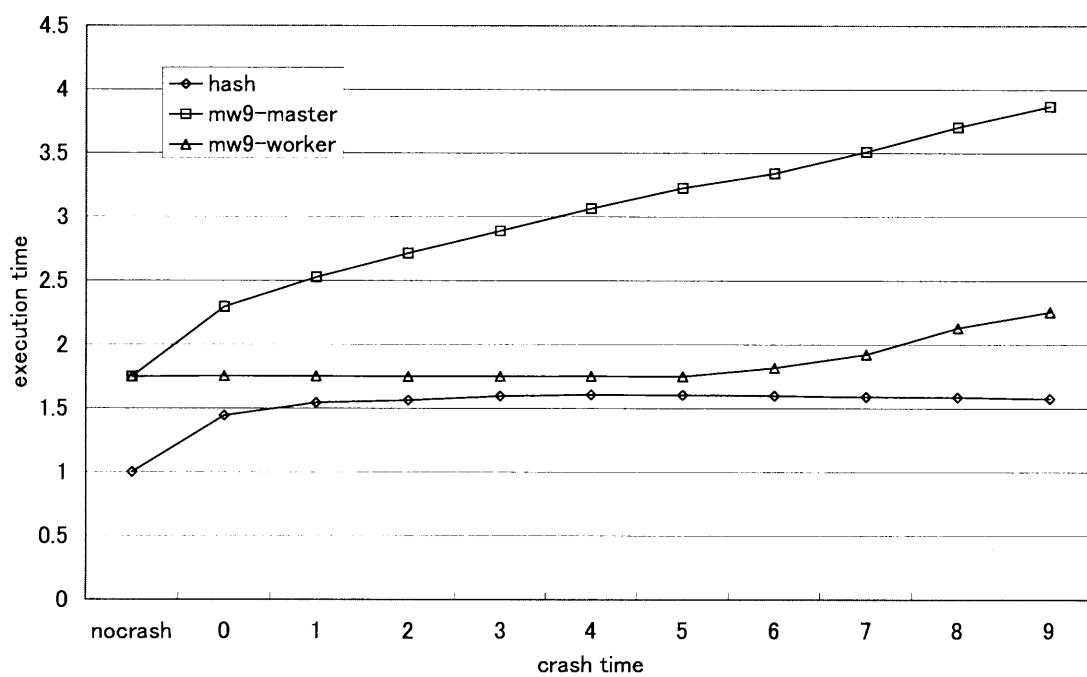


図 12.17: 故障時の計算時間変化 (128 並列、問題サイズ 18, 末端 10 クロック)

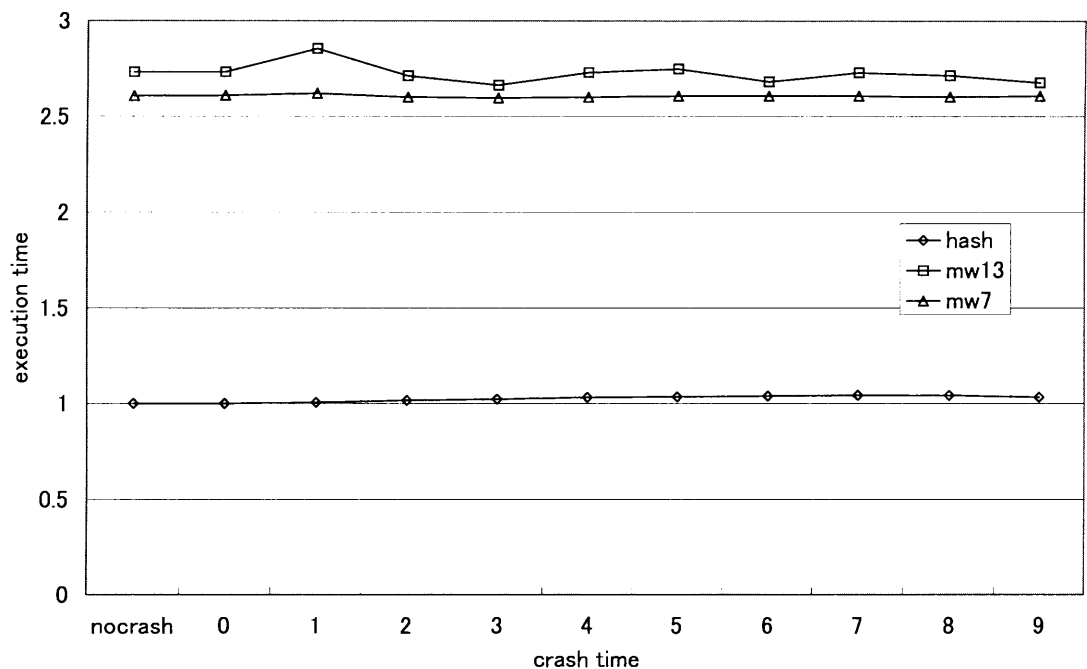


図 12.18: 故障時の計算時間変化 (8 並列、問題サイズ 18, 末端 1000 クロック)

計算ノードの故障割合が大きい場合

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 12.18 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。なお、並列度の変化の影響が大きくなるので、こちらの実験のマスタワーカ方式では、マスタノード 1 とワーカノード 8 の、計 9 ノードを用いて並列計算を行っているという設定でシミュレートした。図 12.18 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言える程度である。また、ワーカのタスクサイズ 13 の時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り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 12.19 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。実際のタスク数は、分散ハッシュ使用時に 51700 個程度、マスタワーカでワーカへの仕事サイズ 13 の時に 1300000 個程度、ワーカへの仕事サイズ 7 の時に 1270000 個程度となっている。ワーカのタスクサイズが 13 の時は、7 の時と比較して全体のタスク数が不安定に増減していることがわかる。これは、一部のワーカの故障によってワーカへの負荷割り当てが変更されるときには、偶然よく似たタスクを受け持っていたワーカなら部分タスクの結果再利用が可能になり、計算時間の短縮が図れるが、逆に運が悪ければ計算時間が大きく伸びる、という偶然に左右される部分があり、

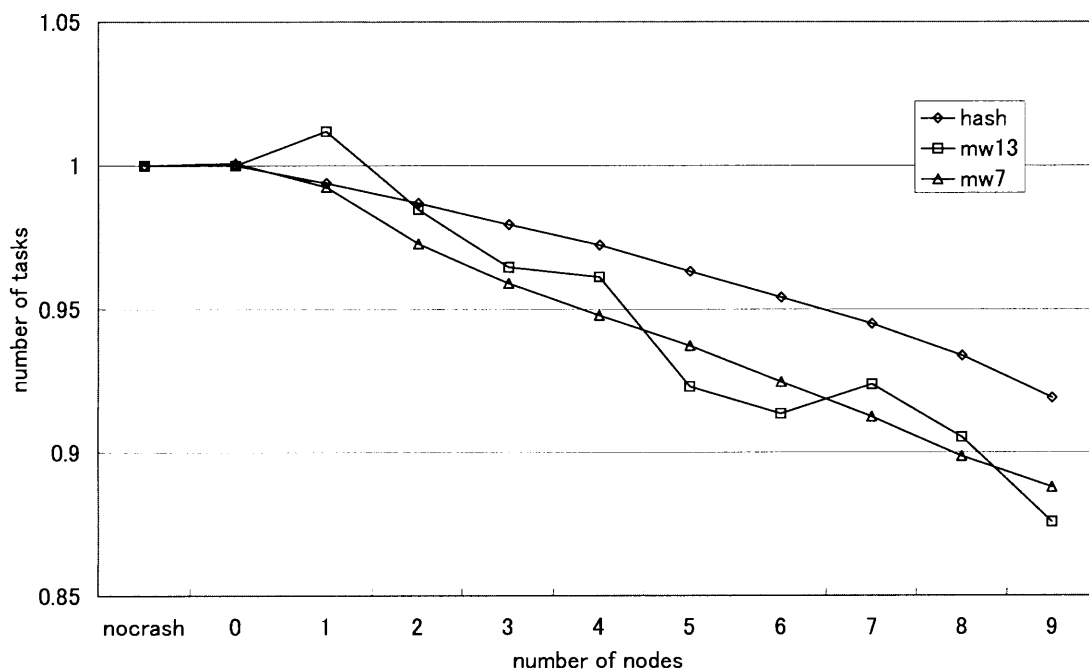


図 12.19: 故障時の最終的なタスク数の変化 (8 並列、問題サイズ 18, 末端 1000 クロック)

ワーカへのタスクサイズが大きいと、この偶然がより強く働くためであると考えられる。なぜなら、ワーカへのタスクサイズが大きい方が、似たタスクが見つかったときの計算時間削減が大きいからである。

ところで、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部分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 12.20 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わかる。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 1400000 個程度、マスタワーカではどちらもおよそ 18000 個程度のメッセージ数であった。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。ただし、もともと分散ハッシュの大量なメッセージ数に耐えられるように実装されたシステムであれば、1 割程度のメッセージ数増加ならそれほど問題にならないとも考えられる。

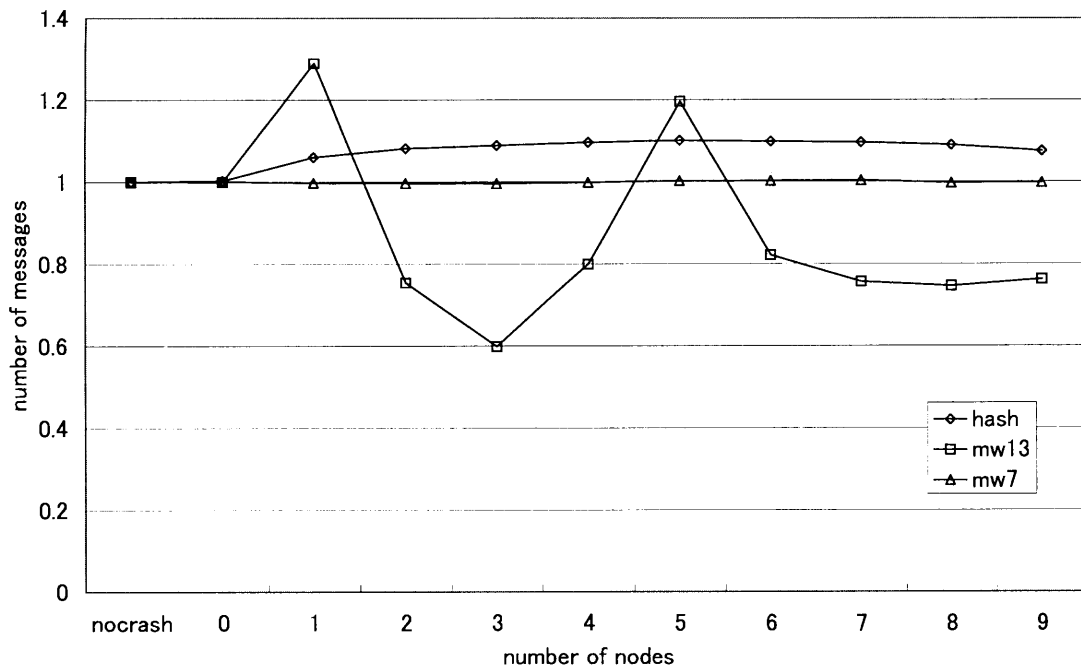


図 12.20: 故障時のメッセージ数の変化 (8 並列、問題サイズ 18, 末端 1000 クロック)

12.4 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、失われた部分計算の結果のうち、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計算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、ワーカ間の部分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、仕事がなくなった計算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。

また、今回のシミュレーションでは提案手法、マスタワーカ方式のどちらも、各計算ノードは全

体の計算が終了するまで、自らが担当した全ての部分問題の記憶を保持し続けていた。これは、提案手法にとっては自然な設定だが、マスタワーカにとってはやや不自然である。マスタワーカ方式では、ワーカに配られるタスク間の依存関係を検出することを最初からあきらめているのであるから、ワーカの記憶は配られたタスクの計算結果が出て、その結果をマスタに返した時点で消去してしまっても問題ないとも考えられる。一方で、提案手法に必要な記憶容量は、計算ノードが担当するハッシュ値の範囲の部分問題の数に比例するので、参加する計算ノードが増加すれば、問題全体に対して各計算ノードの保持しなければならない記憶の割合は低下していく。よって、全ての記憶を保持し続けるというアプローチが実用上非現実的かどうかはさらに検討しなければならない問題である。また、分散ハッシュを用いた手法でメモリが足りないときに、必要がないと思われる記憶を判定して消去するガーベジコレクションアルゴリズムも提案しており、これを用いた場合の振る舞いも検証する必要がある。以上から、各計算ノードやマスタノードが記憶できる容量に何らかの制限を与え、その中での性能比較を行う必要があると考えられる。これは、今後の課題としたい。

第13章 試験実装

13.1 試験実装の方式

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[22]を通信ライブラリとして使用している。

参加、脱退時における担当範囲割り当て変更ポリシーは、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計算ノードが範囲を併合する、というごく簡単なものを実装した。実験においては、開始時点では完全に均等に担当範囲を分割された状態で実行を始めることにした。

また、耐故障性の実現のため、実行可能なタスクが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。

13.2 試験実装を用いた実験

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 2.4GHz dual、2GB memory、64ノードのPCからなるクラスタであり、ネットワークは1Gbit etherで接続されている。この計算ノードが、32台ずつ1つのネットワークスイッチにつながれており、2つのネットワークスイッチ間は4本の1Gbit-etherをtrunkして結合されている。計算ノード台数が32台までの実験は1つのネットワークスイッチのみを使用している。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ーヘッドは、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13.1は解の長さの異なる2つの問題について、計算ノードの数を変えながら台数効果を測定したものである。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

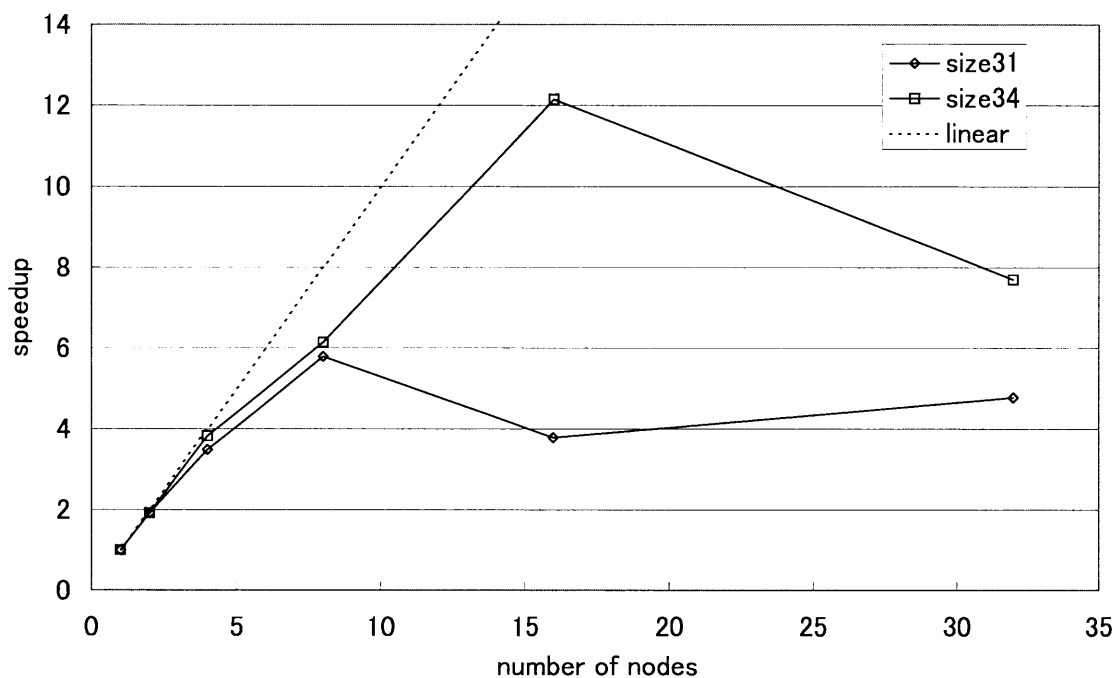


図 13.1: 試験実装の台数効果

表 13.1: 故障発生時の探索時間 (sec)

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

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 13.1 に示す。表 13.1 の 1 行目は、問題のサイズ 31、計算ノード 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上げる実験の結果を表しており、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を示している。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、故障発生が周囲の計算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。

13.3 実装における性能向上への課題

今回実験に用いた 15 パズルの A*探索アルゴリズムでは、探索されるノードが強く限定されるため、末端タスクで展開されるノード数に大きな偏りが発生し、末端タスクの計算粒度が大きく異なることになり、それによる各計算ノードの負荷の偏りが強く見られた。これに対応するためには、何らかの意味で動的負荷分散を実現する必要があると考えられる。

総タスク数が計算ノード数に対して十分多く設定されていれば、計算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、担当範囲の広さを計算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。ただし、これだけでは前述のようなタスクの計算粒度の違いには対応しきれない場合も考えられる。十分タスク数が多ければ、粒度の違いも平均的には均一化されると考えられるが、可能であるならばそのような粒度の大きな違いにも対処できるような負荷分散機構を検討していく必要がある。

また同時に、担当範囲割り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。これは、ハッシュという考え方とは相反するが、緩やかに局所性を保ったまま、問題全体的には均一にランダムにタスクが分布するようなハッシュ方式を考えられれば、それを利用して計算の効率を上げることが可能になると考えられる。ターゲットとなるアプリケーションにおいてこのようなハッシュ方式をうまく構築し、それに合わせたフレームワーク側の計算効率化手法を検討していくことは、価値があると考えている。

現在の仮実装の担当範囲割り当て変更ポリシーは、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計算ノードが範囲を併合する、というごく簡単なものであり、性能向上のためには検討すべき課題は多い。今後、さらに研究を続けていく必要がある。

第14章 第II部のまとめと今後の展望

第II部では、再帰的な依存関係を持ち、重複する部分計算が多く発生す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。部分計算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計算を復元し、重複した部分計算は最大限発見して効率よ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パズルの反復深化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、粒度を適切設定すれば良い並列実行性能を示すことができるとわかった。粒度の設定が必要なのは、粒度が大きいときに負荷の偏りが生じてしまうことと、粒度が小さいときに通信メッセージ数が多くなりすぎることがあり、それらが計算性能を引き下げてしまうからである。これらに対応するためには、動的負荷分散機構や通信量削減が必要であると考えられる。

また、故障検知とその回復手法の設計は正しく動作することが示された。ただし、実装面では、故障回復時の並列計算性能を向上させるためにさらに検討が必要であると考えられる。

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。

第15章 終わりに

15.1 本論文のまとめ

本論文では、大規模な並列計算環境を活用するためのソフトウェア基盤を拡充することを目指した。マルチコアプロセッサやPC クラスタなど、並列計算環境は現在普及が始まっており、今後もさらに大規模に、さらに身近に利用可能になっていくと期待される。この大量のコンピューティングリソースを「活用」できるようなソフトウェアを作るためには、

- 正しいプログラムをどう書くか
- 効率の良いプログラムをどう書くか

の両方を解決しなければならない。

本論文では、第I部において、新しい並列計算量モデルであるアクセス計算量モデルを提案し、効率の良いプログラムをどのように書くか、という問題に対しての改善を試みた。

ここでは、計算の本質は通信にあり、計算量は演算に必要なデータを通信するときの通信コストによって決定される、という基本理念に立つ新しい計算量モデルを提案した。この計算量モデルを用いることで、単体計算機の内部のメモリ階層も、計算機間のネットワーク構成も全て、単一のメモリと通信路のモデルによって簡潔に表現され、様々な実行環境でのアルゴリズムの動作が統一的に示されるようになる。bitonic sort, merge sort, FFT の3種の並列アルゴリズムをこの計算量モデルで解析し、このモデルが十分簡潔で計算量を解析的手法で得ることも可能であることを示した。また、bitonic sort の解析結果を実計算機上での実行と比較することで、従来のPRAMモデルが表現し切れていなかった振舞いをこのモデルがよく表現できていることが示された。FFTのアルゴリズムでも同様に計算量予測が現実とよく一致したが、merge sort の解析結果は実計算機とは必ずしも一致しなかった。

また、第II部では多くの部分計算を共有する問題の耐故障計算フレームワークを提案し、正しいプログラムをどう書くか、という問題に対しての改善を試みた。

ここでは、分散ハッシュテーブルを用いた並列計算手法を導入し、故障時、失われた部分問題を親問題が再実行することで計算を復元する、という耐故障性を実現しようと試みた。シミュレーションの結果、代表的な分散方式であるマスタワーカ方式と比較して、提案手法は無駄な探索を削減することが可能であり、その性質を保ったままで耐故障性を実現できることが示された。また、提案するフレームワークの試験実装を行い、反復深化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、適切に粒度を設定すれば良い並列実行性能を示すことができ、故障検知とその回復手法が正しく動作することが示された。また、動的負荷分散機構や通信量削減が必要であること、故障復旧時の並列計算性能を向上させるためにはさらに実装面での検討が必要であること、などの知見を得ることができた。

15.2 今後の課題と展望

15.2.1 新しい計算量モデルについて

多様な並列計算環境を統一的に扱う計算量モデルの考え方は、今後さらに重要性を増してくると考えられる。計算資源は今後ますます並列性を増し、並列アルゴリズムを使うことが一般的になってくると、環境ごとに最適なアルゴリズムを追求するより、様々な環境で適応的に動作して高い性能を出せるようなアルゴリズムが必要になってくると思われる。このようなアルゴリズムの設計においては、現在実現されている計算環境から未来の環境も含めて、多様な環境における性能を検討する必要がある。通信を計算の本質であると捉えるアプローチは、どのような環境であっても無視できない、論理的にも物理的にも避けられない要素をコスト計算の基準においたという意味で、今後も方針としては正しくあり続けるのではないかと考えている。

このような基本理念に立つとき、難しいのは通信路の統一的なモデルの形成である。本論文では通信パケットの移動をイメージしたやや詳細なモデルと、パケットを局所的な負荷のみで表現するというやや単純化したモデルの2つを提案し、それぞれ特性などを議論してきた。その上で、アルゴリズム解析のために必要な単純化の度合いについての知見を得たり、現実のプログラムの実行時間とのある程度の予測一致などの成果を得られた。しかし、より大規模な分散環境や、通信路がそれほど高機能でない場合などについて、やや提案した通信路のモデル化が現実をよく表現できない場合も見られた。

特に問題になっていたのは、通信路の混雑具合の表現であった。実際、並列アルゴリズムを設計する際に、通信の輻輳をいかに起こさないかは重要な設計目標であり、実際の計算環境でも輻輳が起きにくいような通信路をいかに確保するかが構築の際の注意点となっている。このことから、通信路の混雑具合をある程度一般的に、しかし現実的に即して表現できるようなモデルは重要であると考えられる。そのようなモデルを提案し、計算量理論として体系化することが今後の課題になってくると考えられる。

また、キャッシュメモリの機構が自動的に効率よく働いてしまうようなプログラムについて、本論文では十分な検証ができなかったことも課題である。提案モデルではキャッシュメモリ機構はアルゴリズムの中で明示的に使い方を記述しなければならないという立場を取っているが、これが手軽さを損なっているという問題点がある。しかし、キャッシュを意識したアルゴリズムを用いた解析も可能ではあるので、今後、そのようなアルゴリズムによる解析を通して、本モデルの評価を続けていきたいと考えている。

15.2.2 並列計算フレームワークについて

普及する多様な並列・分散計算環境の上でのプログラミングを、簡単に、再利用性の高い形で実現するためには、並列化ライブラリや並列計算フレームワークによるプログラミングスタイルが今後ますます必要になってくると考えられる。より多くの問題領域に対して、このような基盤ソフトウェアを構築して、より多くの人に並列計算環境による大きな計算力を使用してもらえようになれば、実社会の多くの局面で、より高度な情報処理を用いたより進んだサービスや問題解決が図れると考えられる。

本論文では、耐故障性を持つ大規模分散フレームワークの設計手法について主に提案をし、試験実装を用いてその性能面の評価を行った。今後は、より詳細な実装を行い、多くの問題を用いて、より大規模な計算環境の上での性能評価を行っていく必要がある。

本格実装の際にまず検討しなくてはならないのは、耐故障性の実現手法の詳細である。今回は専用の故障発見機構を用いず、個々の計算ノードのもつ情報のみで、比較的保守的に故障状態からの回復を図ろうとする方式を採用したために、故障発生時の回復性能がそれほど高くないものとなった。しかし、故障発見の手法は様々なものが検討されており、本論文の実装に用いた Phoenix 通信ライブラリの上でも、性能を落とさずに故障を発見できる手法が提案されている [43]。これらの手法を利用することで、より高性能な故障回復機構を実装することを試みる必要がある。ただし、より大規模な分散環境になった場合には、それに適応した故障発見機構を研究する必要があるかもしれない。実装では、ある程度環境に特化した手法を使い分けつつ、その組合せでより高性能なライブラリを構築することを試みるべきであろう。どのような故障検出を用いても、問題全体の計算の正当性には影響がないので、このような実装による工夫は比較的やりやすいと思われるのも、本フレームワークの特徴と言える。

また、計算の性能向上のためには、実装上さまざまな工夫を行う必要があると考えられるが、最も大きな問題としては動的な負荷分散の手法を確立する必要がある。ハッシュ値によって担当する計算ノードを決定する手法を取っているため、個々のタスクの移動はそれほど自由ではなく、一般的な work-stealing などの手法は適用しにくい。問題が十分大きく、粒度も適切で十分な量のタスクが存在している場合には、負荷分散は計算ノードの分担範囲の適切な分割をいかに行うか、という問題を考えることで実現される。これは難しい問題であり、分散ハッシュテーブルでも研究中の分野である。ヘテロな計算ノードの存在や、一部の計算ノードの負荷の高まりによる遅延への対応などを考え始めると、さらに多くの問題が存在しており、今後研究を続けていく必要があると考えられる。

さらに、今回実験に用いた A* アルゴリズムでは、末端タスクで展開されるノード数に大きな偏りがあったために、末端タスクの計算粒度が大きく異なり、それによる負荷の偏りも強く見られた。このような問題に対処するためには、計算ノードの分担領域を適切に分割するだけでは不十分であるとも考えられる。先に述べた計算ノード間で負荷の不均衡が生じる場合と同様の現象でもあるが、均質なクラスタを占有して計算を行っていても、適用問題によってはこのような負荷の偏りが生じるわけであり、これに対処できる何らかの動的負荷分散手法を提案していく必要があると考えられる。

また、通信の効率化も性能向上には欠かせないと言える。本フレームワークの並列化手法をルービックキューブの A* 探索に適用した実装 [44] では、同一の宛先への通信をまとめて一つの大きなメッセージとして送信し、扱うメッセージ数を減らすことで、計算性能が大きく向上していた。本フレームワークでは全てのタスクが他の計算ノードから共有可能であるため、どうしても論理的なメッセージ数は多くなってしまう。実装においてはこの論理的メッセージ数の多さを軽減し、メッセージ処理にかかる時間や通信路の負荷を抑えることが必要になる。同一の宛先への通信を一つにまとめるためには、相手の担当範囲を把握している必要があるが、これはメッセージのルーティングアルゴリズムによっては正確に把握するのが難しい場合もある。そのため、実装においては、ルーティングアルゴリズムにある程度特化したまとめ送り機構を実現したり、相手の範囲の把握が不正確であっても、過去の通信状況や部分的な情報を用いて、メッセージ通信の効率を上げられるような手法を提案したり、といった様々なアプローチ手法を用いることが必要であると考えられる。

他にも、DHT 上でブロードキャストを効率よく行うなどの通信の効率化も必要である。ブロードキャストは本システムでは計算ノードの管理に用いられているが、故障検出など様々な箇所で用いられることも考えられ、このような複雑な通信を効率的に行う手法を確立することも大切であると考えられる。

提案手法をライブラリとして多くの人に使ってもらうためには、使いやすさの面についてもう少

し検討する余地があると考えている。特に、現在のインタフェース実装ではもとのユーザプログラムからの変更が大きく、あまりプログラムが書きやすいとは言えない。これは、将来は、より通常のプログラム言語に近い見た目のプログラミングが行えるよう、ユーザから見えるインタフェースを設計し、プリプロセッサなどによってコード変換を行って、現在の実装のような形で実行を行おうとしているからである。Satin などではやはりこのようなアプローチを取っており、ユーザは通常のプログラムに近い形で最小限の並列計算指示を与え、それを実行システムが Java のバイトコード変換技術を用いて並列計算用ライブラリと結合、実際の並列計算を行う、という実行の流れになっている。このような使いやすいインタフェースを設計し、実装でサポートする必要があると考えている。

また、全ての計算ノードで共有されたグローバルな情報の扱いや投機的計算への対応など、ユーザのプログラミングがより簡単になるような追加機能についても設計、実装していく必要がある。

そして、より多くの問題について、より大規模に、より長期間の計算を行うことで、提案手法の設計の妥当性や実装の安定性を評価していくことが必要であると考えている。

謝辞

近山隆教授には本研究の開始当初より多大なご指導、ご助言をいただきました。心より感謝いたします。

田浦健次朗助教授には特に並列化フレームワークの設計と実装について、多くのご助言をいただきました。心より感謝いたします。

最後に、研究に関する活発な議論の場と心地好い環境を提供していただいた近山・田浦研究室の皆様には感謝いたします。ありがとうございました。

なお、本研究の一部は文部科学省科学研究費特定領域研究「ITの深化の基盤を拓く情報学研究」、ならびに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」の一環として実施された。

参考文献

- [1] OpenMP Architecture Review Board. OpenMP sites. <http://www.openmp.org/>.
- [2] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pp. 272–282, New York, NY, USA, 1989. ACM Press.
- [3] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pp. 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.
- [4] Intel Corporation. Intel high-performance consumer desktop microprocessor timeline.
- [5] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.
- [6] A. V. Aho, J. E. Hopcroft, and J. E. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [7] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing*, pp. 305–314, May 1987.
- [8] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, Vol. 12, No. 2/3, pp. 72–109, 1994.
- [9] A. Aggarwal, A.K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS 87)*, pp. 204–216, 1987.
- [10] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *PPoPP*, pp. 1–12, 1993.
- [11] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, Vol. 44, No. 1, pp. 71–79, 1997.
- [12] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. *Lecture Notes in Computer Science*, Vol. 1800, pp. 1176–1178, 2000.

- [13] C. A. Moritz and M. I. Frank. LoGPC: Modeling network contention in message-passing programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 4, pp. 404–415, April 2001.
- [14] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. Technical Report CS-1993-02, Department of Computer Science, Duke University, 1993.
- [15] Zhiyong Li, Peter H. Mills, and John H. Reif. Models and resource metrics for parallel and distributed computation. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, pp. 51–60, Hawaii, 1995.
- [16] Frank K. H. A. Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *Intl. Journal of Computational Geometry and Applications*, Vol. 6, No. 3, pp. 379–400, 1996.
- [17] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM archive*, Vol. 33, No. 8, pp. 103–111, 1990.
- [18] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. *LNCS*, Vol. 1800, pp. 102–108, 2000.
- [19] Thomas Rauber and Gudula Rünger. Program-based locality measures for scientific computing. In *IPDPS*, p. 164, 2003.
- [20] 横山大作, 近山隆. 計算連続体モデルに基づく仮想機械の仕様. ソフトウェア学会第19回大会, 2002.
- [21] 渡邊誠也, 横山大作, 近山隆, 小宮常康, 湯浅太一. メモリ上の配置を意識する並列処理向き高水準機械語の設計と実装. ソフトウェア学会第20回大会, 2003.
- [22] Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *PPoPP*, 2003.
- [23] TOP500 Project. TOP500 supercomputer sites. <http://www.top500.org/>.
- [24] Takashi Chikayama. KLIC: A portable parallel implementation of a concurrent logic programming language. In *Parallel Symbolic Languages and Systems*, Vol. 1068 of *LNCS*, pp. 286–294. Springer-Verlag, 1995.
- [25] BLAS (Basic Linear Algebra Subprograms) web page. <http://www.netlib.org/blas/>.
- [26] LAPACK (Linear Algebra PACKage) web page. <http://www.netlib.org/lapack/>.
- [27] 横山大作, 鶴岡慶雅, 丸山孝志, 近山隆. コンピュータ将棋プレイヤ「激指」. 平成13年度夏のプログラミング・シンポジウム, August 2001.
- [28] T. Anderson and P. Lee. *Fault Tolerance, Principles and practice*. Prentice Hall International, 1981.

- [29] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.
- [30] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andre Merzky, Thomas Radke, Edward Seidel, and John Shalf. The cactus code: A problem solving environment for the grid. In *HPDC*, pp. 253+, 2000.
- [31] SETI@home sites. <http://setiathome.berkeley.edu/>.
- [32] David P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.
- [33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.
- [34] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, Vol. 37, No. 1, pp. 55–69, 1996.
- [35] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.
- [36] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *ACM SIGPLAN Notices*, Vol. 36, No. 7, pp. 34–43, 2001.
- [37] Gosia Wrzesinska, Rob V. van Nieuwpoort, 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.
- [38] Gosia Wrzesinska, Jason Maassen, Kees Verstoep, and Henri E. Bal. Satin++: Divide-and-share on the grid. Submitted for publication, December 2005.
- [39] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pp. 149–160, 2001.
- [40] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP / ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.
- [41] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.
- [42] 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.

- [43] Yuuki Horita, Kenjiro Taura, and Takashi Chikayama. A scalable and efficient self-organizing failure detector for grid applications. In *Grid 2005 - 6th IEEE/ACM International Workshop on Grid Computing*, pp. 202–210, Seattle, Nov 2005.
- [44] 野澤康文, 横山大作, 近山隆. 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. 第58回情報処理学会・プログラミング研究会, 2006.