学位論文

マスタ・ワーカ型並列プログラムを
高速に実行するためのコンパイラ支援に関する研究

平成16年12月

水谷 泰治

内容梗概

分散メモリ型並列計算環境（分散メモリ環境）における並列プログラムの開発を容易にするために，高級並列プログラム言語で記述したプログラムを分散メモリ環境で実行可能なプログラムへ変換する並列化コンパイラが開発されている．これらのコンパイラの中には，並列再帰を扱えるものがある．並列再帰とは複数の独立な再帰処理を並列実行することをいう．並列再帰を扱えるコンパイラは，再帰処理の並列化方式として，負荷分散を用いない方式，あるいは動的負荷分散が可能なマスタ・ワーカ（MW）方式の一方を採用している．並列再帰プログラムを高速に実行するためには，MW 方式による負荷分散の効果とオーバヘッドのトレードオフを考慮して並列化方式を選び，その並列化方式を扱うコンパイラを用いて並列再帰プログラムを開発する必要がある．また，MW 型並列プログラムを高速に実行するためには，マスタの過負荷による性能低下を生じさせない実行パラメータ（ワーカ数，マスタ数，タスクの一括割当数）の値を検出し，その値を用いて実行する必要がある．様々な分散メモリ環境において並列再帰プログラムを高速に実行するためには，この開発と検出を効率良く行うことが重要である．

本論文では，この開発と検出の支援を目的とし，R1：並列化方式を指定できるコンパイラによる，並列再帰プログラムの開発の支援，および R2：MW 型並列プログラムの性能予測による，高速実行できる実行パラメータ値の検出の支援に取り組む．R1 は，様々な並列化方式に基づく並列再帰プログラムを 1 つの並列プログラムから生成可能とすることで，並列再帰プログラムの開発を支援する．また，R2 は，様々な実行パラメータ値における MW 型並列プログラムの性能を予測することで，開発者による高速実行できる実行パラメータ値の指定を支援する．

まず R1 では，どの並列化方式（MW方式か負荷分散を用いない方式）が並列再帰プログラムの実行時間を短くできるかを，多くの場合，開発者自身が予想できる点に着目する．そして，再帰処理の並列化方式をプログラム開発者が指定できる並列化コンパイラを提案し，実装した．また，実験によりその有用性を示した．実験では，並列化方式および並列化条件（過剰な並列化を抑制するための条件）の違いによってそれぞれ最大 25% および最大 77%の性能差が生じることを確認した．また，並列化方式および並列化条件の指定方針について考察し，その方針に従うことで実行時間を短くできることを示した．さらに，MW 方式におけるマスタ数の指定によって実行時間を短くできることも示した．以上より，並列化方式および並列化条件を開発者が指定できることの重要性を確認した．

次に R2 では，マスタの過負荷時において従来手法が高精度に性能予測できない原因を分析し，性能予測の高精度化ための考慮点として，(D1）並列計算モデルの利用による予測オーバヘッドの低減, (D2）並列計算モデルの拡張によるマスタの通信オーバヘッドのモデル化，および（D3）プログラム実行時に決まる挙動の再現の 3 点を示した．また，これらの考慮点に基づく性能予測の評価実験を行った結果，従来手法ではマスタが過負荷時の予測誤差は最大 42%であるのに対し，考慮点に基づく性能予測では最大 10%の予測誤差であることを示した．この結果により，考慮点の重要性を確認した．

さらに，R2 における性能予測を高速に行うための手法を提案した．提案手法では，一部のタスクの実行時間から残りのタスクの実行時間を線形補間によって推測することで，直接実行部分を削減し，性能予測の高速化を実現する．また，この推測において，高い予測精度を維持するために，タスクの割当順に従って個々のタスクの実行時間も正確に推測することが重要であることを示した．さらに，実験によって，提案手法の予測速度は従来手法より 1.7 倍以上であり，実測との予測誤差は 7%以下であることを示した．この結果より，提案手法は MW 型並列プログラムを高速に実行できる実行パラメータ値の検出に有用であることがわかった．

本研究の成果は，アルゴリズムを構築する上で重要な技法である再帰を容易に並列化でき，開発者に対して並列プログラムの設計の幅を広げた点で有用といえる．また，多様化が進む近年の様々な分散メモリ環境において， MW 型並列プログラムを高速に実行できる実行パラメータ値の高速かつ高精度な検出が可能となった点で有用といえる．

目 次

図 目 次

表 目 次

第1章

序論

1.1 背景

高性能計算のための計算環境として，分散メモリ型並列計算環境（分散メモリ環境）が普及している．世界中の高性能計算機の演算性能を順位付けしている TOP500[38] の報告によると，2004 年 11 月時点において上位 500 のほとんどの高性能計算機は分散メモリ型である．また，複数のパーソナルコンピュータ（PC）を高速なネットワークで接続して構成する PC クラスタ [50] や，ネットワークで接続された計算機の遊休状態を利用して高性能計算を実現するデスクトップグリッド [14, 21] のように，価格性能比に優れた分散メモリ環境が注目されており，分散メモリ環境の利用は現在も広まっている．これに伴い，分散メモリ環境の多様化が進んでいる．

分散メモリ環境においては，メッセージ通信 [18, 24] を用いて並列プログラムを作成することが多い．メッセージ通信プログラムでは，プロセッサ毎の処理を明示的に記述し，プロセッサ間のデータのやりとりはメッセージの通信によって実現する．メッセージ通信プログラムは，プログラムの動作を細かく記述できるので，プログラム開発者の技量によっては高性能な並列プログラムを開発できる．しかし，メッセージ通信プログラムは，並列計算環境の演算性能や通信性能などの特性を考慮し，適切に記述しなければ逐次プログラムよりも性能が低下する可能性がある．また，計算ノードの性能が不均一な並列計算環境においては，性能特性の考慮がより複雑になり，性能の良い並列プログラムの開発がより煩雑になる．

性能の良い並列プログラムの開発を容易にするために，従来の逐次プログラム言語を拡張した高級並列プログラム言語で記述したプログラムからメッセージ通信プログラムへ変換する並列化コンパイラが研究されてきた [10, 28, 34, 44]． 図 1.1 に，高級並列プログラム言語によるプログラム記述の例を示す．ここで，図 1.1 の 3 行目の par 構文は，その内部の計算を並列に実行することを示す．高級並列プログラム言語では，プロセッサ毎の処理の明示的な記述やプロセッサ間の通信を意識せず，高い抽象度で並列プログラムを記述できる．

並列化コンパイラの研究が進むにしたがい，並列再帰を扱える並列化コンパイラが登場してきた [20, 27, 56]．並列再帰とは複数の独立な再帰処理を並列実行することをいう．再帰はアルゴリズムを構築する上で重要な技法であるため，並列実行によって再帰を高速に実行することは並列プログラムを開発する上で有用である．例えば，ある問題を複数の部分問題に分割しそれぞれの部分問題の解を統合することで全体の解を得る分割統治法 [30] と呼ばれるアルゴリズムは，一般に複数の独立な再帰呼出によって記述することが多い．並列再帰を扱う並列化コンパイラは，再帰処理の並列化方式として負荷分散を用いない方式，あるいは動的負荷分散が可

```
1: A[0] = 100;
2: B[0] = A[0];
3: par i = 1 to 100 do
4:     A[i] = B[i-1] + A[i-1];
```

図 1.1: 高級並列プログラム言語によるプログラム記述の例

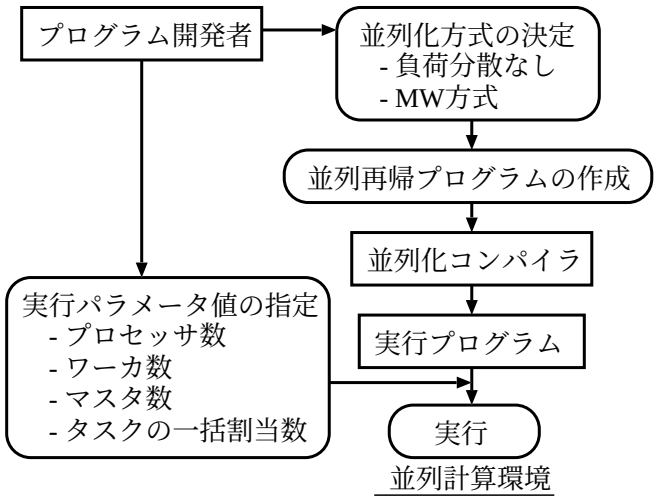


図 1.2: 並列化コンパイラを用いた並列再帰プログラムの開発と実行の流れ

能なマスタ・ワーカ（MW）方式 [24] の一方を用いている． MW 方式とは，使用可能なプロセッサ群を，マスタと呼ばれるグループとワーカと呼ばれるグループに分割し，マスタは並列処理する仕事（タスク）の生成およびワーカへのタスクの割当を担当し，ワーカは割り当てられたタスクを処理する方式である．

図 1.2 に，並列化コンパイラを用いた並列再帰プログラムの開発の流れを示す．まず，開発者は再帰処理の並列化方式を決定する．そして，決定した並列化方式を扱うことができる並列化コンパイラを選び，そのコンパイラが扱う高級並列プログラム言語を用いて並列再帰プログラムを作成する．そして，並列化コンパイラによって作成したプログラムを分散メモリ環境で実行可能なメッセージ通信プログラムに変換する．最後に，プログラムの実行開始時に，開発者は並列化方式に依存するパラメータ（MW 方式の場合はワーカ数，マスタ数，およびワーカへのタスクの一括割当数，負荷分散を用いない方式の場合はプロセッサ数）を指定し，プログラムを実行する．

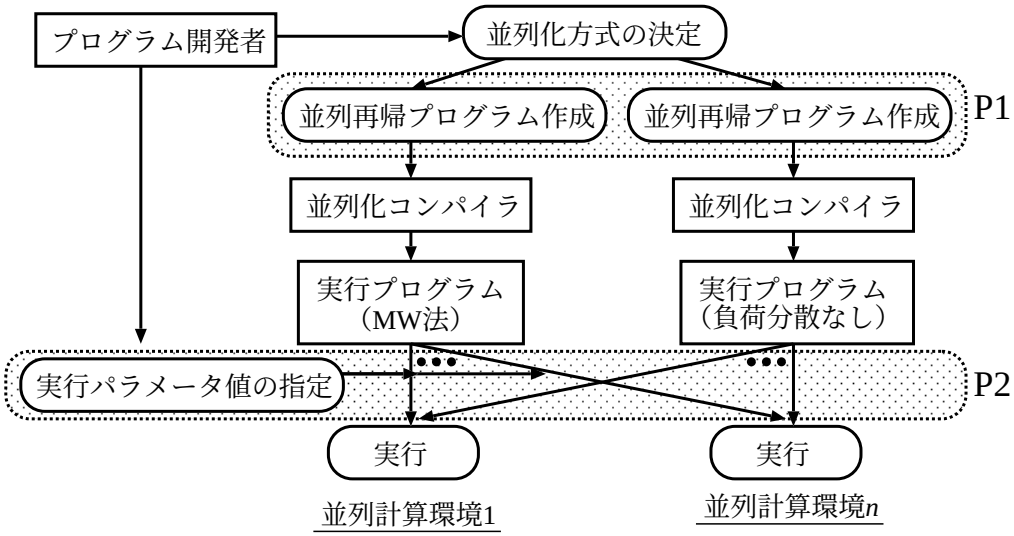


図 1.3: 様々な並列計算環境における並列再帰プログラムの開発と実行の流れ

1.2 動機と目的

近年，PC クラスタやグリッドの普及によって並列計算環境の構築が容易になり，並列計算環境の多様化が進んでいる．並列計算環境の多様化に伴い開発済の並列再帰プログラムを別の並列計算環境で実行したいという要求がある．しかし，この要求に対して，以下の 2 つの問題点がある．

P1. 並列再帰プログラムの再開発に要する時間の増加

P2. 高速実行できる実行パラメータ値の検出に要する時間の増加

図 1.3 に，並列計算環境の多様化による開発の流れの変化を示す．ある並列計算環境において実行時間を短くできる並列化方式が，別の並列計算環境においても実行時間を短くできるとは限らない．実行時間を短くできる並列化方式の決定においては，MW 方式による動的負荷分散の効果とオーバヘッドのトレードオフを考慮する必要がある．同じ再帰アルゴリズムを扱う場合でも，並列計算環境の違いより，実行時間を短くできる並列化方式が異なる可能性がある．このような場合，実行時間を短くできる並列化方式への変更する必要がある．すなわち，並列化コンパイラを変更し，そのコンパイラ用に並列再帰プログラムを再開発する必要がある（P1）．また，MW プログラムを高速実行できる実行パラメータ値も並列計算環境毎に異なる．不適切な実行パラメータ値で MW プログラムを実行すると，マスタが過負荷になり，MW プログラムの性能低下を引き起こす．高速実行できる実行パラメータは，並列計算環境の演算性能，通信性能，およびアプリケーションに依存し，解析的に求めることは容易ではない．したがって，高速実行できる実行パラメータ値の検出は，実際にいろいろな実行パラメータ値を与えてプログラムを実行するという，試行錯誤によることが多く，検出のコストが大きくなる（P2）．

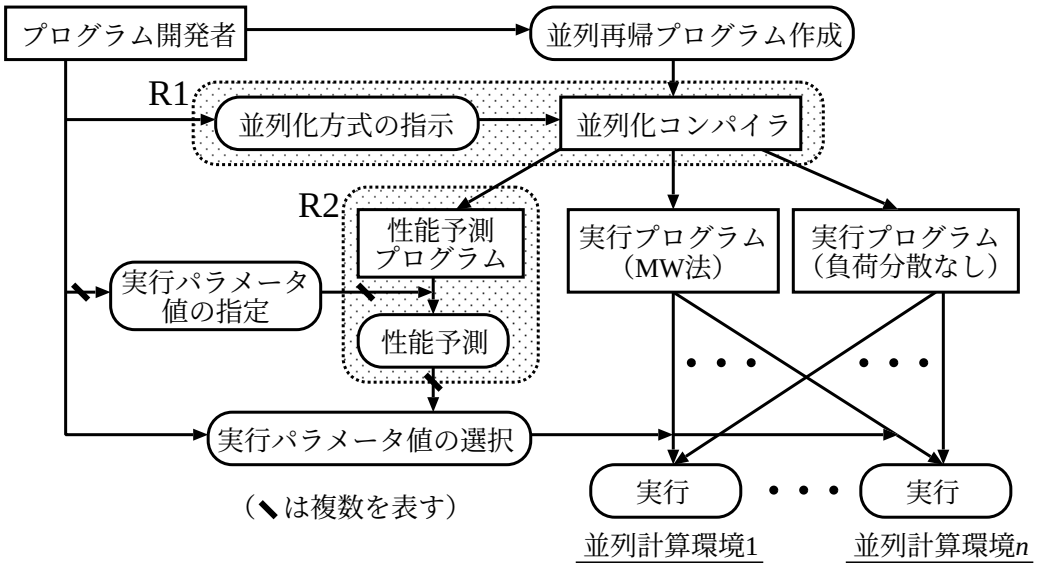


図 1.4: R1 および R2 による並列再帰プログラムの開発と実行の流れ

本研究の目的は，コンパイラ支援によって問題点 P1 および P2 を解決し，並列プログラムの開発者に対して，並列再帰プログラムの開発と，MW プログラムを高速実行できる実行パラメータ値の検出の支援を目的とする．この目的を実現するために，以下の 2 つに取り組んだ．

R1. 並列化方式を指定できるコンパイラによる，並列再帰プログラムの開発

R2. MW プログラムの性能予測による，高速実行できる実行パラメータ値の検出

図 1.4 に，R1 および R2 による並列再帰プログラムの開発の流れを示す．まず，開発者は高級並列プログラム言語で並列再帰プログラムを作成する．そして，開発者は並列化コンパイラに対して並列再帰プログラムの並列化方式を指定する．その指定を基に，並列化コンパイラは，並列再帰プログラムを実行プログラムへ変換する（R1）．異なる並列化方式に基づく実行プログラムを生成したい場合は，並列化コンパイラへの指定を変更するだけでよく，プログラムの作り直しは必要としない．一般に，開発者は，自身が開発する再帰アルゴリズムに対してどの並列化方式が実行時間を短くできるかを予想できる場合が多い．したがって，並列化方式を指定できる並列化コンパイラによって，高速実行できる並列再帰プログラムを容易に開発できる．これは，並列プログラムの設計の幅が広がることにつながり，並列プログラムの開発者にとって重要である．また，並列化コンパイラが MW プログラムを生成する際，その MW プログラムの性能を予測するプログラム（性能予測プログラム）も生成する．性能予測プログラムは，実行パラメータ値（ワーカ数，マスタ数，およびタスクの一括割当数）を入力とし，その実行パラメータ値で MW プログラムを実行したときの性能を高精度かつ高速に予測し，出力する（R2）．この性能予測プログラムを用いて，開発者は様々な実行パラメータ値に対する予測性能を列挙し，それらの中から最も実行時間を短くできる実行パラメータ値を選択して MW プログラムを

実行する．これにより，試行錯誤に頼らず，実行時間を短くできる実行パラメータ値の検出を高速に行える．

1.3 成果

本論文での成果を以下に要約する．

R1. 並列化方式を指定できるコンパイラによる，並列再帰プログラムの開発

再帰処理の並列化方式（MW 方式または負荷分散を用いない方式）をプログラム開発者が指定できる並列化コンパイラを提案および実装し，実験によりその有用性を示した．一般に，プログラム開発者は，自身が開発する再帰プログラムの動作を理解しており，どの並列化方式が実行時間をより短くできるかを予想できる場合が多い．この点に着目し，提案手法では開発者に再帰処理の並列化方式を指定させ，並列化コンパイラは指定された並列化方式に基づくプログラムを生成する．また，提案手法では，並列プログラムの過剰な並列化を抑制するための条件（並列化条件）および MW 方式におけるマスタ数も指定する．実験により，並列化方式および並列化条件の違いによるプログラムの性能の違いを調べたところ，並列化方式および並列化条件の違いによって，それぞれ最大 25%および最大 77%の性能差を確認した．また，並列化方式および並列化条件の指定方針について考察し，その方針に従うことで実行時間を短くできることを示した．さらに，マスタ数の指定によって実行時間を短くできることも示した．以上より，並列化方式および並列化条件を開発者が指定できることの重要性を確認した．

R2. MW プログラムの性能予測による，高速実行できる実行パラメータ値の検出

性能予測の精度および速度のそれぞれの観点から，以下に示す 2 項目について成果を得た．

- 従来の性能予測手法ではマスタが過負荷時に高精度に性能予測できない原因を分析し， MW プログラムに対する高精度な性能予測のための考慮点を提示した．また，実験によって，考慮点の重要性を示した． MW プログラムの高精度の予測を実現するために，マスタにおける性能ボトルネックを表現できるように拡張した並列計算モデルを用いて通信時間を予測する．また，実験によって，64 台の CPU をもつ PC クラスタにおける MW プログラムの実行時間を良い精度で予測できることを示した．実験結果は，実測実行時間と比較して，従来の並列計算モデルを用いた予測実行時間との誤差は 最大 42%であるのに対し，拡張したモデル用いた予測実行時間との誤差は 10%以下であった．この結果より，提示した考慮点は MW プログラムの性能を高精度な予測するために重要であることがわかった．
- MW プログラムの実行時間を高速に予測するための手法を提案し，その有用性を確認した．提案手法は，MW プログラム全体の直接実行を避けることで，予測に要する時間を短縮する．これを実現するために，まず，MW プログラムを部分的に実行することで一部のタスクのみの実行時間を測定し，線形補間によってその実行時間

から残りのタスクの実行時間を推測する．次に，MW 法の動作のシミュレーションによって各タスクを処理するプロセッサを決め，そのプロセッサ上でのタスクの実行時間を推定する．実験の結果，提案手法は MW プログラム全体の直接実行に基づく従来手法と比べ，同程度の予測精度を保ちつつ，1.7 倍以上高速に予測できた．

1.4 本論文の構成

本論文の構成を以下に示す．まず，2 章では，プログラムの並列化方式をコンパイラに指示する手法を提案する．また，負荷の分布傾向が異なるいくつかの問題に対して提案手法を適用し，評価することで，提案手法の有用性を示す．次に，3 章では，MW プログラムの性能を高精度に予測するための考慮点を提示する．また，実験によって考慮点の重要性を示す．そして，4 章では，MW プログラムの性能予測を高速に行う手法を提案し，その有用性を評価する．最後に，5 章で本論文の結論と今後の課題を示す．

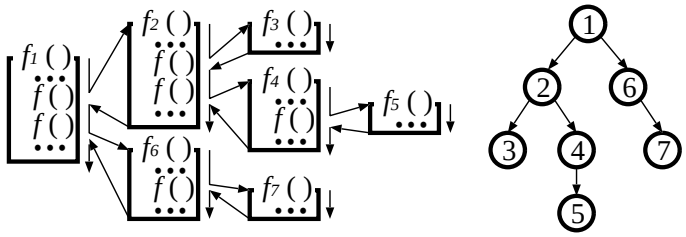
第2章

並列化方式を指定できるコンパイラによる並列再帰プログラムの開発支援

2.1 はじめに

近年，高級並列プログラム言語を用いて並列再帰を扱うことができる並列化コンパイラが開発されている [20, 27, 56]．これらのコンパイラでは再帰処理の並列化方式としてマスタ・ワーカ（MW）方式と呼ばれる方式を採用しているものが多い．MW 方式によって，並列実行される各再帰関数の計算量の不均等を緩和することができる（動的負荷分散）．しかし，計算量がほぼ均等となる再帰アルゴリズムに対しては，負荷を均等化するための処理がオーバヘッドとなり，実行性能が低下する原因となる．このような場合，負荷の均等化の処理を除いた単純な並列化方式の方が望ましい．既存のコンパイラでは，１つの並列化方式のみで処理するため，再帰アルゴリズムによっては十分な性能を得られない場合がある．そのため，各再帰アルゴリズムに対し効果的な並列化方式を適用することが重要である．負荷バランスは並列再帰アルゴリズムの動作等の要因によると考えられる．また，並列化方式に関わらず，過剰な並列化によって並列実行しない場合よりも実行性能が低下することがある．並列実行をするか否かは，再帰アルゴリズムの計算量，並列計算環境の計算能力，通信性能等に依存すると考えられる．しかし，これらの要因を並列再帰アルゴリズムのソースプログラムからコンパイラが機械的に解析し，最適な並列化方式および並列実行するか否かを判断することは容易ではない．

一方，一般に開発者は再帰アルゴリズムの動作を理解しており，どの並列化方式が並列再帰プログラムの実行時間を短くできるかを予想できる場合が多い．そのため，開発者が並列再帰の並列化方式を選択できることは有用である．また，過剰な並列化を抑制するための情報など，コンパイラによる機械的な解析が容易でない情報を，開発者が指定できることも重要である．開発者がもつ一般的な知識を基に性能の良い並列再帰プログラムを記述できることは，並列プログラムの設計の幅を広げることにつながり，並列プログラムの開発者にとって有用であると考える．そこで，本章では，実行性能の良い並列プログラムを生成するために有用な情報を，開発者が明示的にコンパイラに指定する手法を提案し，実験によってその有用性を確かめる．



(a) 制御の流れ (b) 再帰木

図 2.1: 再帰呼出の例

2.2 再帰処理の並列化手法

本節では，既存の並列化コンパイラ [20, 27, 35] が採用している並列再帰の実行方法を体系的に整理する．

2.2.1 再帰木と並列再帰

図 2.1(a) に，ある再帰関数 f を逐次実行したときの制御の流れを示す．再帰関数 f は最初の呼出も含め実行全体で 7 回呼び出されており，それぞれを $f_1, f_2, ..., f_7$ と表す．図 2.1(a) に対応する再帰呼出の概念図を図 2.1(b) に示す．図 2.1(a) の再帰関数 $f_1, f_2, ..., f_7$ は，図 2.1(b) の頂点 1,2,...,7 に対応する．矢印は再帰呼出を表す．このように再帰呼出を表した木を**再帰木** [20] と呼び，頂点の出次数の最大値を**分岐数**と呼ぶ．頂点 v と v の全子孫から成る部分木を $T(v)$ と表す． v に対応する再帰関数の全実行を $T(v)$ **に対応する処理**といい， $T(v)$ に対応する処理の計算量を $T(v)$ **の計算量**という．また，v **に対応する処理**は再帰関数中の再帰呼出以外の処理を表す． v の対応する処理から再帰呼出によって頂点 u に対応する処理を呼び出すをことを， v **から** u **を呼び出す**という．

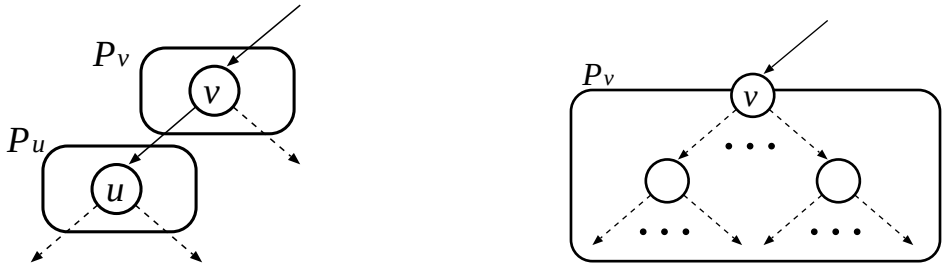
並列再帰とは，再帰関数中で直接呼び出す複数の再帰呼出の実行が独立（結果に依存関係が無い）な場合，それらを並列実行することをいう．図 2.1(b) を例にとると， $T(2)$ と $T(6)$ の処理が独立な場合，これらを並列実行することをいう．

2.2.2 再帰木へのプロセッサ割当方法

並列再帰呼出を行う場合，どのようにプロセッサを使用して並列実行するかを考える必要がある．頂点 v または部分木 $T(v)$ に対応する処理をプロセッサ p が実行することを， v または $T(v)$ **に** p **を割り当てる**という．プロセッサ割当方法として，プロセッサ集合を頂点に割り当てる方法（**頂点割当方式**，図 2.2(a)）と部分木に割り当てる方法（**部分木割当方式**，図 2.2(b)）が考えられる．

プロセッサの割当に関して以下の記号を定義する．

- $P(v)$: 頂点 v に割り当てるプロセッサ集合．



(a) 頂点割当方式　　(b) 部分木割当方式

図 2.2: 再帰木へのプロセッサ割当

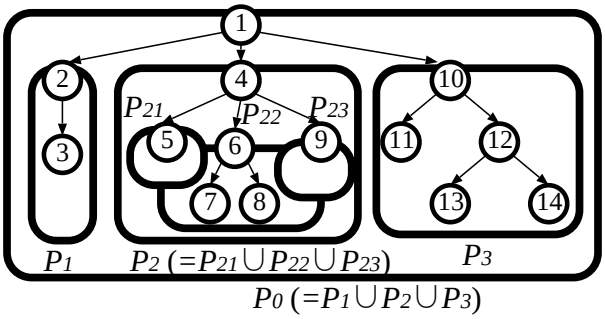


図 2.3: 呼出時割当型

- $P(T(v))$: 部分木 $T(v)$ に割り当てるプロセッサ集合.
- $|P|$: プロセッサ集合 P の要素数.

頂点割当方式では，頂点 v が呼び出された時点で v にプロセッサ集合 P_v を割り当てる． P_v は v に対応する処理を実行し，子頂点 u を呼び出した時点で，u にプロセッサ集合 P_u を割り当てる．P_u は一般に P_v とは異なる集合である．一方，部分木割当方式では部分木 $T(v)$ に対してプロセッサ集合 P を割り当てる．$T(v)$ の処理は，P のみを用いて行う．部分木割当方式は，どの時点で部分木にプロセッサ集合を割り当てるかによって，次の 2 通りに分類できる．

呼出時割当型 ある頂点 v から n 個の頂点 $v_1, ..., v_n$ を呼び出した時点で，$T(v_1), ..., T(v_n)$ にそれぞれプロセッサ集合 $P_1, ..., P_n$ を割り当てる．$P_i (1 \leq i \leq n)$ は，$P(T(v))$ を分割したものである（図 2.3）.

部分木数指定型 再帰木を幅優先で走査しながら部分木を得る．部分木数が指定された数に達した時点で，各部分木にプロセッサ集合を割り当てる．図 2.4 に部分木数を 6 に指定した例を示す.

一般に，頂点 v を呼び出した時点で $T(v)$ の計算量はわからない．部分木割当方式では，v を呼び出した時点で $P(T(v))$ を限定するため，$|P(T(v))|$ が $T(v)$ の計算量に対して不適切になる

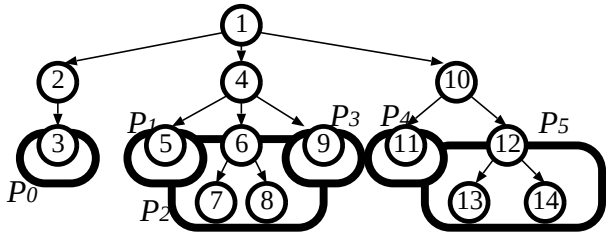


図 2.4: 部分木数指定型: 部分木数=6

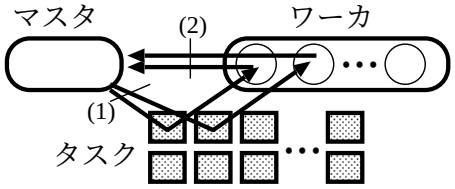


図 2.5: マスタ・ワーカ方式の概念図

可能性がある．頂点割当方式では，v を呼び出したときに， $P(v)$ を使用可能全プロセッサから選択するため，部分木割当方式に比べ，再帰木へのプロセッサの割当方に融通がきく．しかし，割当時には $P(v)$ 中のプロセッサは， $P(v)$ 以外のプロセッサが使用可能か否かの情報を把握しておく必要があり，その情報の交換を $P(v)$ と $P(v)$ 以外のプロセッサ間で行わなければいけない．一方，部分木割当方式では $T(v)$ の処理に使用できるプロセッサは $P(T(v))$ に限定されており， $P(T(v))$ 以外のプロセッサとの情報交換は必要ない．そのため，頂点割当方式に比べプロセッサの管理が軽減される．特に $|P(T(v))| = 1$ の場合，$T(v)$ の処理を逐次処理するため，並列化のオーバヘッドを無くすことができる．

全プロセッサ数が部分木数に満たない場合，1 つのプロセッサ集合が複数の部分木を担当する．

2.2.3 マスタ・ワーカ方式による動的負荷分散

並列再帰呼出によって並列計算する際，各プロセッサが実行する計算量（プロセッサの負荷）に偏りが生じる場合がある．そのため，他のプロセッサの計算が完了していないにも関わらず，あるプロセッサは計算が完了しアイドル（何も計算しない）状態となる場合がある．アイドル状態が多く発生すると全体の性能の低下につながる．このような場合，処理を完了したプロセッサを計算中の部分木のプロセッサ集合に含めることで，負荷の不均等を緩和することができる．このように，プログラム実行時に負荷の均等化を試みることを動的負荷分散と呼ぶ [17].

動的負荷分散方式の 1 つとして MW 方式 [5, 20, 23, 27] が知られている．図 2.5 に MW 方式の概念図を示す．この方式では，プロセッサをマスタ集合 $\mathcal{M}$ とワーカ集合 $\mathcal{W}$ に分割する．$\mathcal{M}$ はタスクの処理は行わず，$\mathcal{W}$ の状態管理と $\mathcal{W}$ へのタスクの割当を担当する． $\mathcal{W}$ に属するプロセッサ（ワーカ）はタスクの処理を担当する．プログラム実行時に，$\mathcal{M}$ は $\mathcal{W}$ から空き状態（タスクが割り当てられていない状態）のワーカ集合を選択し，タスクを割り当てる（図 2.5(1)）．

$\mathcal{W}$ に空き状態のワーカが存在しない場合，空き状態のワーカが現れるまで待つ．タスクを割り当てられたワーカは，処理の完了後にその旨を $\mathcal{M}$ に報告し（図 2.5(2)），空き状態に戻る．$\mathcal{M}$ は $\mathcal{W}$ の状態を把握しているため，ある処理を完了したワーカへ別の処理を再度割り当てることができる．

頂点割当方式では頂点を，部分木割当方式では部分木を，図 2.5 のタスクに対応させることで，MW 方式を適用可能である．

2.2.4 再帰アルゴリズムと並列化方式の適性

並列再帰の**並列化方式**とはあるプロセッサ割当方法と動的負荷分散の有無を組み合わせた実行方法を表す．頂点 $v_1, v_2, ..., v_k$ は兄弟の関係にあるとする．$T(v_i)(1 \leq i \leq k)$ の性質によって，並列再帰の効率的な並列化方式が異なる．

（C1）各 $T(v_i)$ の計算量が不均等，かつ v_i の呼出時に $T(v_i)$ の計算量を予測できない場合

（C2）各 $T(v_i)$ の計算量がほぼ均等，または v_i の呼出時に各 $T(v_i)$ の計算量を予測できる場合

（C1）の場合，負荷の均等化を行う動的負荷分散を用いた並列化方式が適する．（C2）の場合，各部分木への適切なプロセッサの投入量を予測できるので．動的負荷分散を適用しない並列化方式が適する．

2.2.5 並列再帰の実行時の問題点

上記の並列化方式に基づいたコンパイラを用いて並列再帰を実行する場合，以下の問題点が考えられる．

（E1）過剰な並列化による性能低下

並列再帰呼出においては次のオーバヘッドが生じる．

（E1-1） 割り当てるプロセッサ集合を決定する処理

（E1-2） 新たな頂点または部分木にプロセッサ集合を割り当てる処理

ある頂点 v から頂点 u を呼び出すとし，v または $T(v)$ に割り当てるプロセッサ集合を P_v，u または $T(u)$ に割り当てるプロセッサ集合を P_u とする．（E1-1）は P_u を決定する処理に対応する．（E1-2）は，P_v と P_u の間のメッセージ（制御信号，u の処理に必要なデータ）通信に対応する．

ある部分木 $T(v)$ の計算量が小さい場合，$T(v)$ の処理の中で（E1-1），（E1-2）の占める割合が大きくなる．そのため，（E1-1），（E1-2）を除き，$T(v)$ 内の各部分木または頂点に $P(T(v))$ を割り当てた方が早く処理を完了できる場合がある．

（E2）負荷バランスの予測が容易でない

各部分木の計算量の偏りを，コンパイラが並列再帰アルゴリズムのソースプログラム記

述から機械的に解析することは容易でない．そのため，効果的な並列再帰の並列化方式を自動的に選択することは難しい．

2.3 コンパイラに対する情報の指定

本研究では，問題（E1）および（E2）を解決するために，開発者がコード生成に有用な情報を指定できるコンパイラを提案する．コンパイラに指定する情報として，**並列化条件**を導入することで問題（E1）を解決する．また，2 種類の並列化方式を開発者が指定することで問題（E2）を解決する．開発者はソースプログラム中に並列化条件と並列化方式を宣言的に記述する（2.4 節）．

2.3.1 並列化条件の指定

並列化条件とは，頂点 v において新たに呼び出す部分木を，$P(v)$ のみで実行するか 2 節で述べた並列化方式で実行するかを決定する条件式である．図 2.3 を例にとると，頂点 4 において並列化条件を満たさない場合，$P_{21} = P_{22} = P_{23} = P_2$ とする．並列化条件によって，過剰な並列化による性能低下を避けることができる（2.5.1 節参照）．

2.3.2 並列化方式の指定

本研究では，（C1）に適する並列化方式として，動的負荷分散を適用した頂点割当方式（dynamic 方式と呼ぶ）を採用し，（C2）に適する並列化方式として，動的負荷分散を適用しない呼出時割当型の部分木割当方式（simple 方式と呼ぶ）を採用した．以下，各方式の概略を示す．詳細は文献 [17] に譲る．

simple 方式

再帰木に全プロセッサからなる集合 P_0 を割り当てる．再帰関数の全入力引数データは，P_0 に含まれる 1 台のプロセッサが保持する．以降，呼出時割当型の部分木割当方式に基づき，部分木毎にプロセッサ集合を分割していく．各頂点 v の処理は，$P(T(v))$ に属する 1 台のプロセッサ p が実行する．v への全入力引数データは p が保有する．再帰呼出時において，並列化条件を満たさない，または $|P(v)|=1$ の場合，v の各子頂点を根とする各部分木に対し，$P(v)$ を割り当てる．並列化条件を満たす場合，v の k 個の子を $v_1, v_2..., v_k$ とし，$T(v_1), T(v_2), ..., T(v_k)$ のそれぞれに，プロセッサ集合 $P_1, P_2, ..., P_k$ を割り当てる．$|P_i|(1 \leq i \leq k)$ は $P(T(v))$ を k 等分[1]した値である．$|P(T(v))| < k$ の場合，1 つのプロセッサ集合を複数の部分木に割り当てる．こ

[1] $T(v_1), T(v_2), ..., T(v_k)$ の計算量が均等ではないが，ばらつきを予測できる場合がある．計算量の比に応じて分割の比を調整することで性能向上を見込めるが，本論文では，実装の簡便化のためにこのような場合でも k 等分する．なお，計算量の比に応じた分割についての議論は，文献 [43] に譲る．

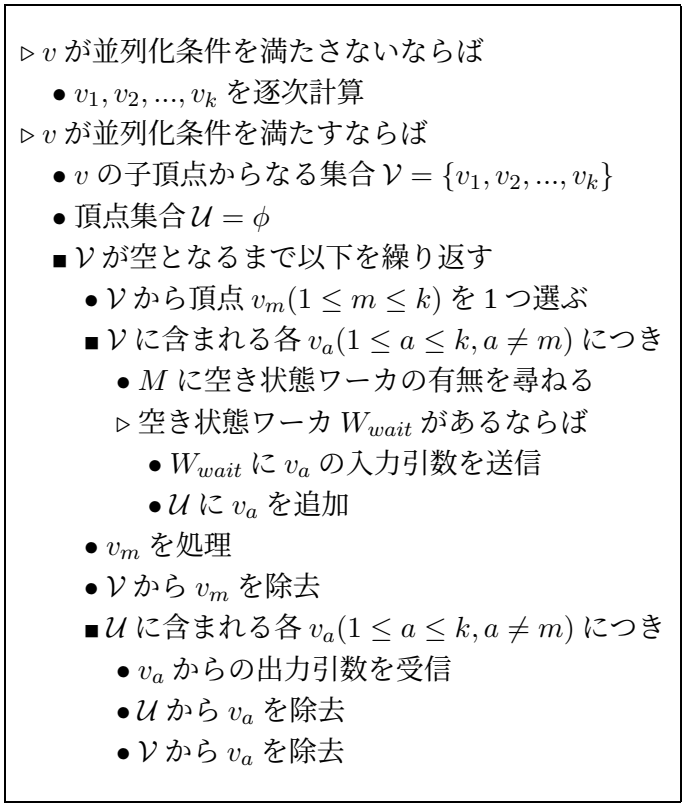


図 2.6: ワーカによる並列再帰呼出のアルゴリズム

の方式は，動的負荷分散を適用していないため，負荷の均等化のための処理が無く，プロセッサ割当の処理が単純であることが特長である．

dynamic 方式

MW 方式を適用した頂点割当方式に基づき，各頂点にプロセッサ集合を割り当てていく．本研究では，実装の簡略化のため，プロセッサ集合の要素数を 1 に限定した．すなわち，各頂点に対して 1 個のプロセッサを割り当てていく．全プロセッサの中から 1 台のマスタ M を選び，残りのプロセッサをワーカ集合 $\mathcal{W}$ とする．頂点 v に割り当てられるワーカを $W(v)$ と表記する．

最初の再帰呼出が行われるまでは，ある 1 台のプロセッサ p が並列再帰プログラムを実行する．プログラムの全データは，プログラムの実行開始時から p が保有する．最初に呼び出された再帰関数，すなわち再帰木の根には p を割り当てる．以後，p はワーカとして機能する．図 2.6 に，MW 方式による並列再帰の処理の流れを示す．頂点 v の子を $v_1, v_2, ..., v_k(k \geq 1)$ とする．各頂点 v では，並列化条件を満たさない場合，並列再帰呼出は行わず，$T(v_1), T(v_2), ..., T(v_k)$ の処理を $W(v)$ が逐次実行する．並列化条件を満たす場合，子頂点の 1 つ $v_m(1 \leq m \leq k)$ は $W(v)$ が担当する．残りの各子頂点それぞれについては，$W(v)$ が M を通し，空き状態のワーカ W_{wait} に v の子の処理を依頼する．頂点 $v_a(1 \leq a \leq k, a \neq m)$ の割当依頼が受理された場合，

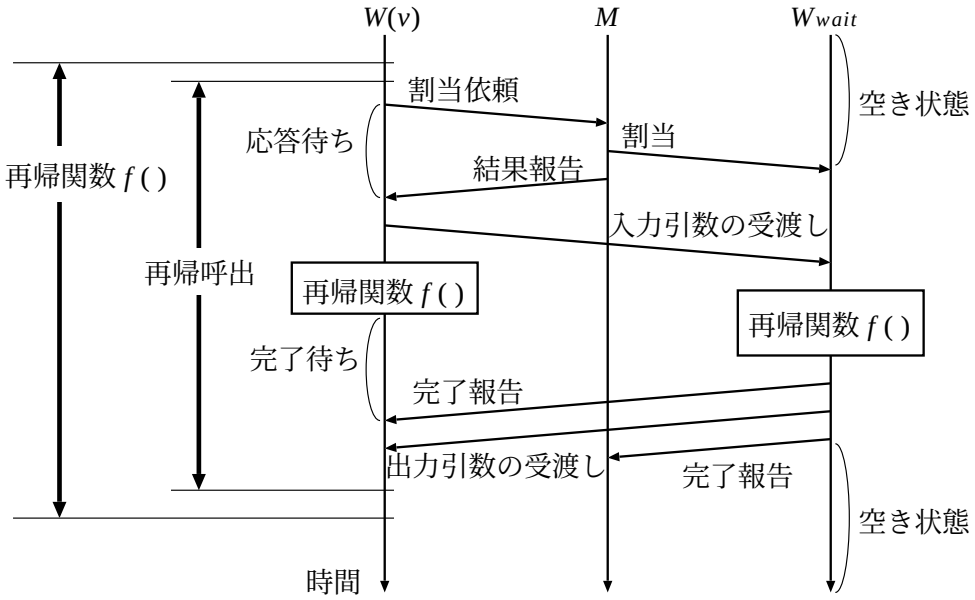


図 2.7: マスタ・ワーカ方式による並列再帰

W_{wait} に v_a の入力引数データを送信することで，W_{wait} の頂点 v_a への割当を完了する．空き状態のワーカが存在せず v_a の割当依頼が却下された場合，v_a への割当を先送りし，$W(v)$ は v_m の処理を実行する．v_m の完了後，以下を繰り返す．M に未処理の頂点 v_a の割当を依頼する．割当依頼が受理された場合は，他ワーカを v_a に割り当て，却下された場合は $W(v)$ が v_a を実行する．この操作を，全ての子頂点の処理を完了するまで繰り返す．図 2.7 に，再帰関数 f の実行における，並列化条件を満たす場合の $M, W(v), W_{wait}$ の動作を示す．

複数マスタの使用

dynamic 方式において $|\mathcal{W}|$ が大きい場合，M と $\mathcal{W}$ 間の通信頻度が高くなり，M に高負荷がかかる．そのため，$\mathcal{W}$ への応答が遅れ，プログラム全体の性能が低下する．このような場合，マスタ数を複数とし，マスタにかかる負荷を分散させる．

2.4 再帰処理に対する並列化方式と並列化条件の指定方法

並列再帰プログラムの記述には，1.1 節で述べた高級並列プログラム言語の 1 つである，Work-Time C 言語 [41] を用いる． Work-Time C 言語で並列再帰プログラムを記述し，コンパイラは開発者が指定した並列化方式に基づく中間プログラムを生成する．中間プログラムは通信ライブラリ MPI[48] を用いた C 言語で記述される．中間プログラムを既存の C コンパイラでコンパイルすることで，並列計算機上で動作する SPMD[54] 型プログラムを生成する．

従来の Work-Time C コンパイラ [56] では，並列再帰プログラムの記述はできるが，並列化方

```
recursive void MergeSort(int *ary, int n)
in:   ary, n;
out:  ary;
cond: n>4096;
mode: simple;
{
  int i, j, k, m, top[2], size[2], *b;
  if (n<=1) return;
  b = (int *)malloc(n*sizeof(int));
  m = (n-1)/2;
  top[0] = 0;   size[0] = m+1;
  top[1] = m+1; size[1] = n-(m+1);

  par x=0 to 1 do
    MergeSort(&ary[top[x]],size[x]);

  for (i=m+1; i>0; i--)
    b[i-1]=ary[i-1];
  for(j=m; j<n-1; j++)
    b[n-1+m-j]=ary[j+1];
  for(k=0; k<=n-1; k++)
    ary[k]=(b[i]<b[j])? b[i++]:b[j--];
  free(b);
}
```

図 2.8: Work-Time C 言語による並列再帰プログラム

式および並列化条件の指定はできず，再帰処理の並列化方式は負荷分散を用いない方式のみしか対応できない．本研究では，Work-Time C コンパイラに対して，いくつかの予約語を追加することで並列化方式および並列化条件の指定を可能とする．具体的には，並列再帰関数を表す recursive，再帰関数の入力引数および出力引数を表す in および out，並列化条件を表す cond，および並列化方式を表す mode を追加した．図 2.8 に，これらの予約語を用いた並列再帰プログラムの記述例を示す．この例では，再帰関数 MergeSort 中で並列再帰呼出を行っている．この例のように，Work-Time C 言語では par 文を用いて並列計算可能な部分を開発者が明示的に記述する．MergeSort においては，引数 ary を入力・出力の両方に用いている． 並列化条件は，予約語 cond に続いて C 言語における式の形で記述する．この式は，中間プログラムにおいて再帰関数中の再帰呼出の位置で評価される．MergeSort では式 n>4096 が真である場合のみ並列再帰呼出を行い，偽の場合には単一プロセッサで逐次実行する． 並列化方式

表 2.1: アルゴリズムの特徴

アルゴリズム	負荷	分岐数	予想実行方式
マージソート	均等 [2]	2	simple
クイックソート	不均等	2	dynamic
n 女王問題	不均等	n	dynamic

は，予約語 mode に続いて指定する． simple の場合には simple 方式，dynamic の場合には dynamic 方式を適用する．dynamic を指定する場合には,「mode： dynamic 2」のように，dynamic に続いて使用するマスタ数も指定可能である．省略した場合にはマスタ数は 1 と解釈する．

このように，Work-Time C 言語を用いることで，par 文による明示的な並列処理の指定以外は，逐次プログラムと同じ形で並列プログラムを記述できる．また，追加した予約語による並列化方式と並列化条件の指定は，再帰処理のプログラム記述を直接変更することはない．したがって，並列再帰プログラムの容易な記述が可能であると考える．

2.5 性能評価

再帰木の分岐数および負荷バランスの傾向が異なる 3 種類の再帰アルゴリズム（マージソート，クイックソート，n 女王問題）について性能評価を行う．ここで，ソーティングアルゴリズムを選択した理由は，逐次処理における代表的な再帰アルゴリズムの並列化による効果を調べるためである．また，ソーティングアルゴリズムとしてマージソートとクイックソートを選択した理由は，再帰処理の負荷バランスが異なる再帰アルゴリズムに対する並列化方式の違いによる性能変化を調べるためである． n 女王問題を選択した理由は，再帰木の分岐数が大きくなる再帰アルゴリズムの並列化の効果を調べるためである．

表 2.1 に．各アルゴリズムの特徴を示す．表中の予想実行方式は，2.5.3 節の指定方針に基づいて予想した並列化方式であり，開発者が mode:欄に指定する（2.4 節）．実験では，逐次処理において代表的な再帰アルゴリズムの並列再帰による性能変化を調べるためにソーティングアルゴリズムを用いた．ここで，並列化方式の違いによる並列再帰プログラムの性能の変化を調べるために，マージソートとクイックソートを用いた．また，再帰木の分岐数が大きくなる再帰アルゴリズムとして，n 女王問題を選択した． 表 2.1 に．各アルゴリズムの特徴を示す．表中の予測実行方式は， 2.5.3 節に基づいて，適すると予測した並列化方式であり，開発者が mode：欄に指定する（2.4 節）.

各アルゴリズムの入力データサイズを n で表す．マージソート，クイックソートでは n は配列要素数を表し，$n = 2^{22}$ である． n 女王問題では $n \times n$ の盤面サイズを表し，$n = 14$ である．本節では，マージソートおよびクイックソートについては，ある 1 つのランダムに並んだ入力データについて実行時間を測定した結果を示している．なお，クイックソートについては，多

[2]マージソートは，最後に呼び出される再帰処理の内容によっては負荷バランスが不均等となる可能性がある．本研究では負荷バランスが均等となる単純マージソート [33] を用いる．

表 2.2: 各アルゴリズムの並列化条件

アルゴリズム	並列化条件に使用する変数と意味
マージソート	c: 再帰呼出の入力配列の要素数
クイックソート	c: 再帰呼出の入力配列の要素数
n 女王問題	d: 再帰呼出の再帰木における根からの深さ

くのランダム系列の入力データに関して同様の結果を得ることができたので，1 種類の結果についてのみ示している．

性能評価基準として，以下に定義するプロセッサ p 台を用いたときの速度向上率 S_p を用いる．

$$S_p = \frac{\text{プロセッサ 1 台での実行時間}}{\text{プロセッサ } p \text{ 台での実行時間}}$$

S_p の値が大きいほど，プログラムの並列化による実行効率の向上が大きいことを表す．

並列計算環境としては，並列計算機 NEC Cenju-3 （PE:VR4400SC 75MHz 128 個，通信性能:40MB ／秒，メモリ:64MB ／ PE）を使用した．

以降では，実験結果と考察を示す．

2.5.1 並列化条件による性能の変化

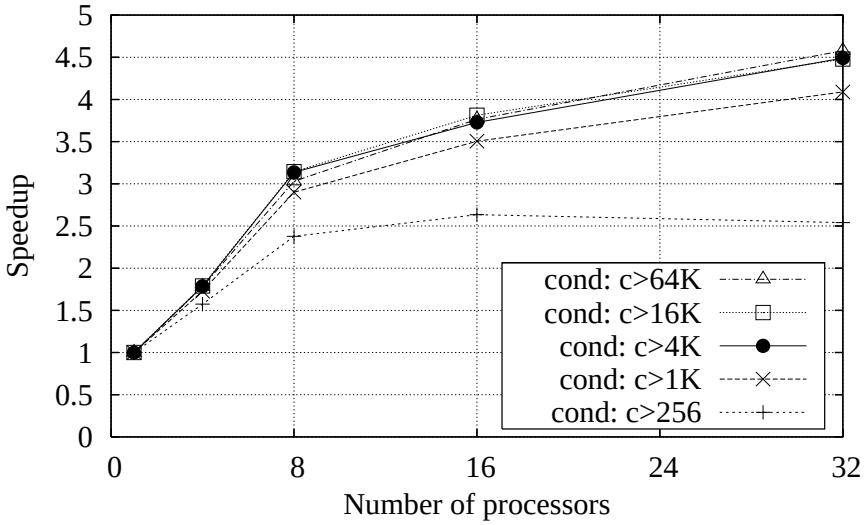
図 2.9 に，dynamic 方式を適用したクイックソートおよび n 女王問題のプログラムについて並列化条件毎の S_p の測定結果を示す．クイックソートの並列化条件は $c > y$（c:再帰呼出の入力配列要素数）であり，n 女王問題の並列化条件は $d < x$（d:再帰の深さ）である． n 女王問題のプロセッサ数 128 における実行時間の内訳（全ワーカの平均）の測定結果を図 2.10 に示す． 図 2.10 中の凡例は，図 2.7 で示した各部分を表す．

図 2.9(a) では，プロセッサ数 32 において，並列化条件 $c > 256$ の性能は $c >$ 64K の性能に対して 45%低下している．また，図 2.9(b) では，プロセッサ数 128 において，並列化条件 $d < 6$ の性能は $d < 4$ の性能に対して 77%低下している．このように，並列化条件は各プログラムの性能に対して大きな影響を与えている．図 2.10 では，n 女王問題では並列化条件を $d < 6$ とした場合，応答待ちおよび空き状態の時間が極めて大きい．これは過剰な並列化によって，並列実行する部分木数が膨大になり，マスタ M とワーカ間の通信が頻発し， M の負荷が大きくなり，M からの指令・応答が遅れるためである．

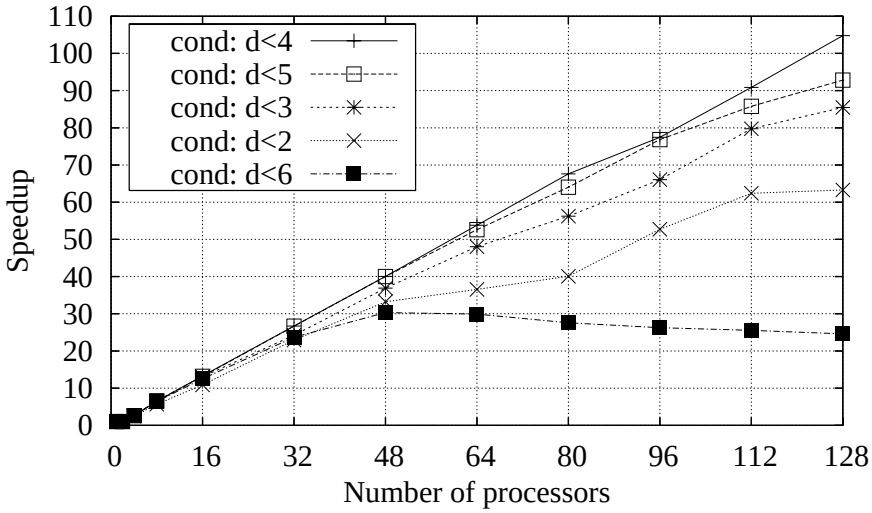
以上より，並列化条件の指定によって過剰な並列化を抑制できることは重要であることがわかる．

2.5.2 並列化方式の比較

図 2.11 に，各アルゴリズムに simple/dynamic 方式を指定してコンパイルし，各並列プログラムの S_p を測定した結果を示す．dynamic 方式において使用したマスタ数 m は 1 である．各ア



(a) クイックソート



(b) n 女王問題

図 2.9: 並列化条件による性能の違い

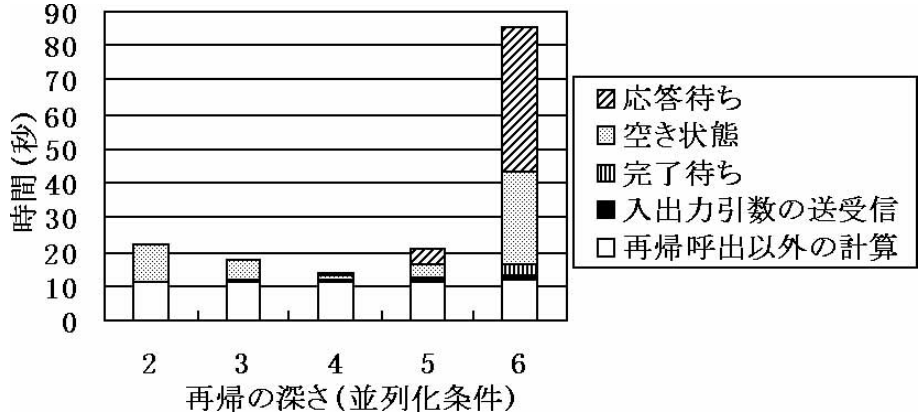


図 2.10: 実行時間の内訳:n 女王問題 (n=14, プロセッサ数 128, m=1)

ルゴリズムの並列化条件を表 2.2 に示す．図 2.11 では，並列化方式による性能の違いを示すため，両方式において同一の並列化条件を指定した．並列化条件は，2.5.4 節の予測に基づいて得られた並列化条件を利用した．

マージソート (図 2.11(a)) では， dynamic 方式より simple 方式の方が高い S_p を示している．マージソートの再帰木では各レベルの部分木の計算量がほぼ均等であるため，動的負荷分散の処理が無い simple 方式の方が，良い性能を示したと考えられる．

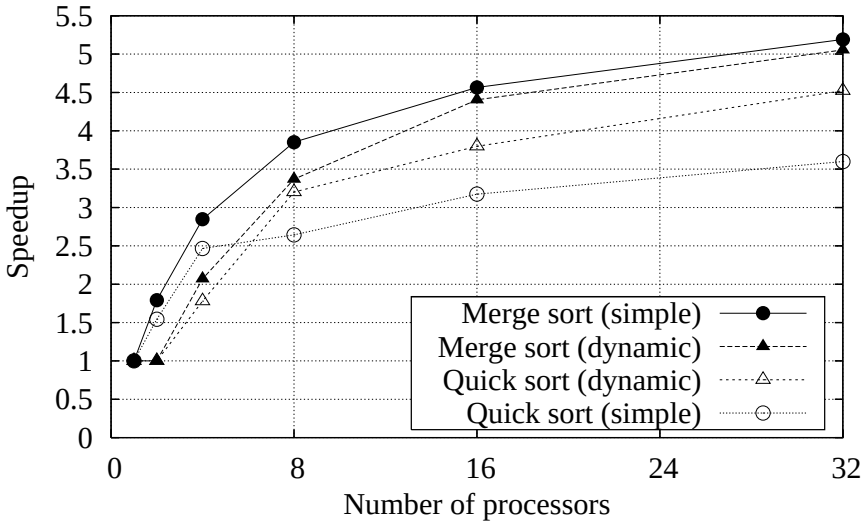
クイックソート (図 2.11(a)) と n 女王問題 (図 2.11(b)) は，プロセッサ数が 8 以上の場合，simple 方式より dynamic 方式の方が最大で 25%高い S_p を示している．これは動的負荷分散により各プロセッサの負荷が均等化されたためである（2.5.3 節）．プロセッサ数が 4 以下の場合，simple 方式と dynamic 方式の性能が逆転している． dynamic 方式におけるマスタは再帰関数自体の計算に参加できず，プロセッサ数が少ないほどマスタの計算不参加による影響が大きくなるためである（2.5.3 節）．

n 女王問題は分岐数が n と大きいため、他のアルゴリズムと比べて高い S_p が得られる．一般に，分岐数が大きいほど早い時刻に多くのプロセッサに処理を割り当てることができる．そのため各プロセッサが処理の割当を待つ時間は小さくなり実行性能が向上する．

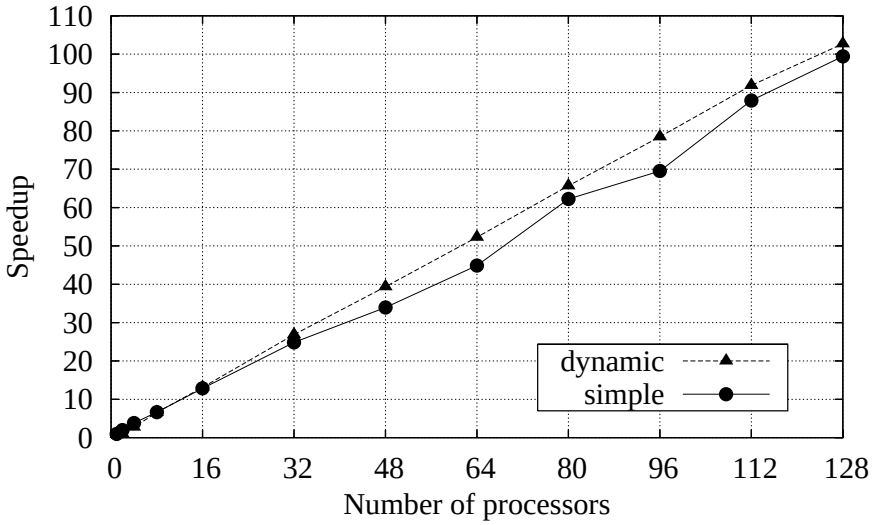
マスタの計算不参加の影響が大きい範囲（プロセッサ数 2,4）を除き，全てのアルゴリズムにおいて，表 2.1 の予測並列化方式の方が他方の手法より良い性能を示している．並列再帰アルゴリズムの特徴から，効果的な並列化方式が予測可能な場合がある．そのため，並列再帰アルゴリズムの並列化方式を選択・指定できることは有用である．

2.5.3 並列化方式の指定方針

クイックソート（プロセッサ数 16）における，各プロセッサの実行時間の内訳を図 2.12(a)（simple 方式）と図 2.12(b)（dynamic 方式，ワーカのみ）に示す．クイックソートの入力データは，図 2.11(a) で用いたものと同じである．simple 方式ではマスタが存在しないため，応答待ち時間は存在しない．simple 方式における空き状態とは， 2.3.2 節において，プログラム開始か

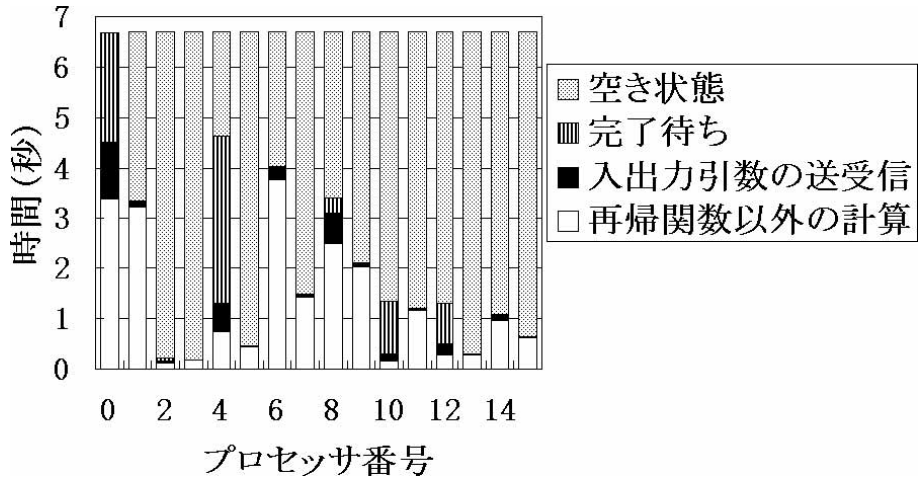


(a) マージソートとクイックソート（並列化条件: $c > 4$K，マスタ数: $m = 1$）

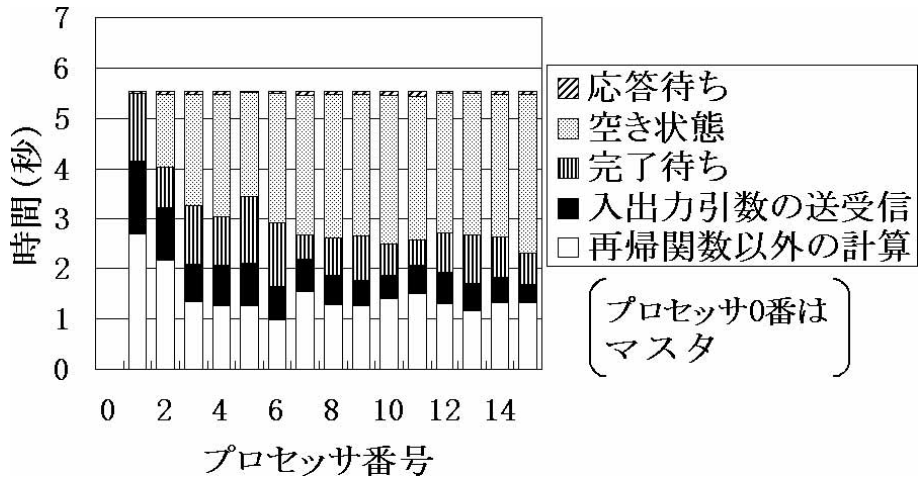


(b) n 女王問題（並列化条件: $d < 4$，マスタ数: $m = 1$ ）

図 2.11: 並列化方式の違いによる性能の変化



(a) simple 方式



(b) dynamic 方式

図 2.12: クイックソートの内訳時間

ら頂点処理の実行開始までの時間と，頂点処理が完了からプログラムが終了までの時間の和である．

図 2.12(a) では，各プロセッサにおいて再帰呼出以外の計算時間に大きな偏りがある．これは，各部分木の計算量が偏っているにも関わらず，simple 方式では部分木の処理を完了したプロセッサを再利用しないためである．一方図 2.12(b) では，再帰呼出以外の計算時間が全ワーカサにおいて分散され， simple 方式に比べて全体性能が良い．このように各部分木の計算量が不均等である再帰アルゴリズムに対しては，動的負荷分散を行う dynamic 方式が適する．

マージソートの場合，任意の 1 つのレベルにおいて各部分木の計算量はほぼ均等となる．そのため， simple 方式でも各プロセッサの計算量はほぼ均等となる．dynamic 方式でも各プロセッサの計算時間を均等化することは可能である．しかし割当依頼や応答待ちなど動的負荷分散に必要な処理がオーバヘッドとなり，simple 方式と比べ性能が低下する．このように，各部分木の計算量がほぼ均等である再帰アルゴリズムに対しては simple 方式が有効である．

使用プロセッサ数を k とすると，simple 方式では再帰関数自体の計算に使用するプロセッサは k 台であるのに対し， dynamic 方式（マスタ数 m）では $(k-m)$ 台である．使用プロセッサ数が少ないほど，マスタの計算不参加による影響が大きくなり， dynamic 方式は simple 方式よりも不利になる．

任意に与えられた再帰アルゴリズムに対して，ユーザが $(k-m)/k$ の値が小さい，すなわちマスタの計算不参加による影響が大きいと判断した場合，simple 方式を選択すればよい．$(k-m)/k$ の値が大きい，すなわちマスタの計算不参加による影響が小さいと判断した場合，再帰木の任意の 1 つのレベルにおいて，各部分木の計算量が均等であるか均等でないかに着目し，均等と予測できる場合は simple 方式を，均等と予測できない場合は dynamic 方式を選択すればよい．例えばマージソートでは，ソート対象の配列を均等に分割していくため，各部分木の計算量はほぼ均等になると予測できる．一方クイックソートの場合，ソート対象の配列の分割点は入力データに依存するため，各部分木の計算量は不均等になると予測できる．このように，アルゴリズムの性質から，各部分木の計算量のバランスを予測し，適する並列化方式を選択できると考えられる．

2.5.4 並列化条件の指定方針

不適切な並列化条件によってマスタ M が過負荷状態となり，全体性能が低下する可能性がある（2.5.1 節）．過剰な並列化を引き起こさない並列化条件の指定方針を，マスタの負荷の観点から示す．

記号を以下に定義する．t_m を M がワーカからの 1 個の割当依頼要求に応答するのに必要な時間，f を再帰木の分岐数，w を 1 個の M が管理するワーカ数，T_s を並列化条件を満たさない最大規模の部分木，t_s をワーカによる T_s の逐次実行時間とする（図 2.13）．t_m は再帰アルゴリズムによらず並列計算機の性能特性のみに依存する．

単位時間あたりに 1 つの M が処理できる最大要求数 C_p は $1/t_m$ である．単位時間あたりに 1 つの M が管理する全ワーカから M へ到達する要求数を C_r とする．並列再帰呼出が進み，全ワーカが並列化条件を満たす最小規模の部分木を担当する状況を考える． 1 個のワーカが 1 回

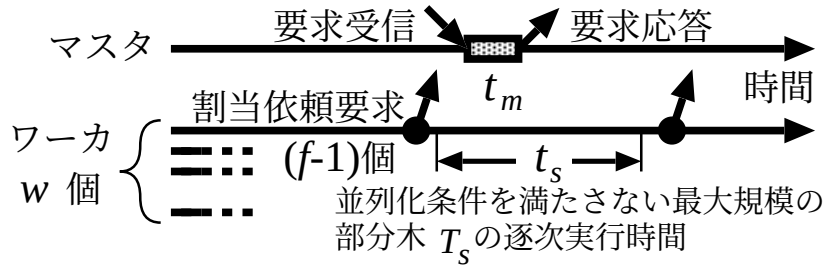


図 2.13: dynamic 方式におけるワーカ要求の処理

の並列再帰呼出で送信する要求数は高々$(f-1)$ である．ワーカは並列再帰呼出後，T_s を担当するので， t_s の間は次の並列再帰呼出は無い．並列再帰呼出時点の近傍を考えると，1 個のワーカが単位時間あたりに送信する要求数は $(f-1)/t_s$ である．これらより，並列再帰呼出時点の近傍の C_r は $(f-1)w/t_s$ と表せる．

M が過負荷状態にならないためには $C_r \leq C_p$ でなければならない．そこで，M が過負荷状態にならない条件として，次式 (2.1) の成立を目安とする．

$$(f-1)w/t_s \ \leq \ 1/t_m \tag{2.1}$$

式 (2.1) では t_s のみ並列化条件に依存する． f, w, t_m の各値を代入した式 (2.1) を満たすように t_s を選択し，その t_s に対応する並列化条件を選択すれば，M の過負荷による性能低下を避けることが可能と考えられる． t_s から並列化条件を導くことは容易でないため，予めいくつか並列化条件を選択し，対応する t_s を計測しておくものとする．

再帰木の部分木のみを切り出すことは容易ではないので， T_s は入力データ数 n を小さくした問題の再帰木で近似する．例えば，クイックソートの並列化条件を $c>y$ とした場合， T_s はデータ数 y のランダム系列を入力としたクイックソートの再帰木とする． n 女王問題の並列化条件を $d<x$ とした場合，T_s は $(n-x)$ 女王問題の再帰木とする． t_s は近似した T_s の逐次実行時間とする．

表 2.3 に n 女王問題 (n=14) とクイックソートにおける t_s の測定結果を示す．実験環境において t_m は 148［μs］であった． n 女王問題 (n=14) の分岐数 f は 14 である．使用プロセッサ数を 128（マスタ数 1）とし，式 (2.1) に f=14, t_m=148×10^{-6}, w=127 を代入すると， $0.244 \leq t_s$ となる．クイックソートの分岐数 f は 2 である．使用プロセッサ数を 32 （マスタ数 1）として同様に式 (2.1) に代入すると，$0.00458 \leq t_s$ となる．これらと表 2.3 より， n 女王問題では $d<4$ を，クイックソートでは $c>4$K を選べば，M の過負荷による性能低下を回避できる．この判定は，図 2.9 の結果とよく一致する．

2.5.5 マスタの複数化による性能向上

dynamic 方式の n 女王問題のプログラムについて，マスタ数 m の変化における S_p の測定結果を図 2.14 に示す．ここでは，マスタに高負荷をかけるため，不適切な並列化条件 $d<6$（2.5.1

表 2.3: 小さい入力データサイズにおける逐次実行時間

n 女王問題 (n=14)			クイックソート		
cond	T_sの n	t_s [s]	cond	T_sの n	t_s [s]
$d<2$	12	34.0	$c>64$K	64K	0.244
$d<3$	11	5.95	$c>16$K	16K	0.0551
$d<4$	10	1.03	$c>4$K	4K	0.0119
$d<5$	9	0.199	$c>1$K	1K	0.00279
$d<6$	8	0.0381	$c>256$	256	0.000627

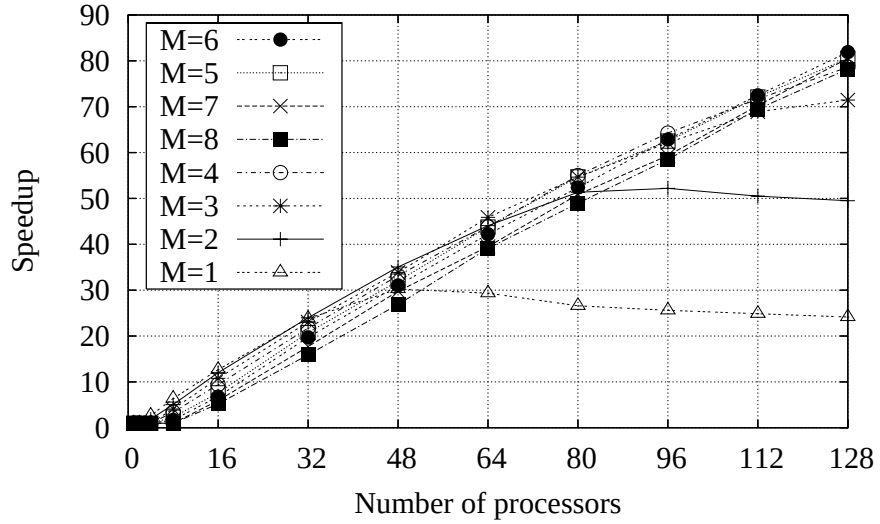


図 2.14: マスタ数 m による性能の変化: n 女王問題 (n=14, cond:d <6)

節）を使用した．プロセッサ数 128 におけるプログラム実行時間の内訳（全ワーカの平均）を測定した結果を図 2.15 に示す．図 2.15 から，マスタ数を増やすことで応答待ちおよび空き状態の時間を大きく削減していることがわかる．これは，複数マスタの利用により，マスタへの負荷が分散されるためである．

図 2.15 では $m=6$ の場合に最も良い性能を示している． m の増加により，マスタへの負荷を分散できる一方，マスタの計算不参加による影響も大きくなる．このトレードオフがバランスした点が，本実験では $m=6$ であったといえる．

適切なマスタ数は，2.5.4 節の式 (2.1) から予測できる．n 女王問題において並列化条件 $d<6$ とした場合，表 2.3 より t_s は 0.0381 秒である．使用プロセッサ数を 128 とすると，マスタ 1 台が担当するワーカ数 w は $(128-m)/m$ である．これらの値を用いて 2.5.4 節と同様に式 (2.1) に代入すると，$6.15 \leq m$ となる．これより，m は 6 付近の値を選べばよい．

このように，マスタへの負荷が大きいと予測できる場合，複数マスタの利用によりマスタへの負荷を分散し，全体性能の向上が可能である．

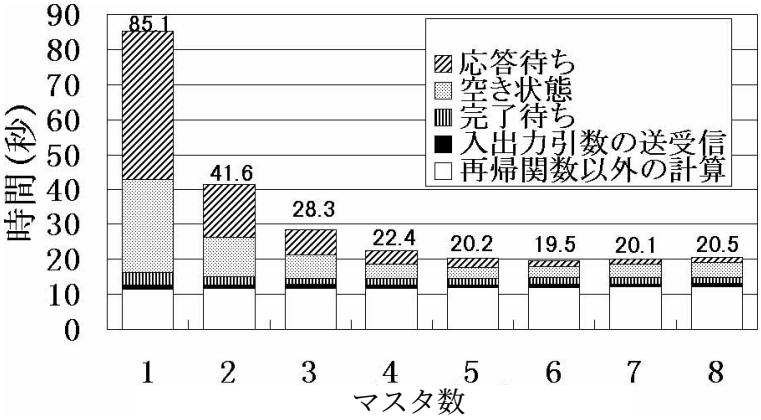


図 2.15: 実行時間の内訳（複数マスタ）: n 女王問題 (n=14, プロセッサ数 128, cond:$d<6$)

2.6 関連研究

並列再帰アルゴリズムの記述・実行が可能なコンパイラとして PRP[20] と Machiavelli[27] を紹介する．

PRP は，C 言語を拡張した言語で並列再帰プログラムを記述し，通信ライブラリ PVM を用いた並列プログラムに変換する．並列再帰の実行方法は，MW 方式を適用した部分木数指定型の部分木割当方式に基づく． PRP では，まずマスタが単独でプログラムを実行し，部分木を一定数（開発者が指定）になるまで生成する．そして，部分木数が一定数に達した時点で MW 方式による並列実行を開始する．PRP が生成するプログラムは，n 女王問題のように分岐数が多い場合，早い時刻に指定した部分木数に達し並列実行が可能になる．また， PRP が用いる MW 方式では， Work-Time C が用いる MW 方式のようなワーカとワーカの間の通信は発生せず，マスタとワーカの間のみで通信を行う．したがって， PRP では 図 2.7 におけるワーカの完了待ちは発生せず，Work-Time C によりも高性能に並列再帰プログラムを実行できる場合がある． 図 2.16 に，n 女王問題における Work-Time C と PRP の速度向上率を示す[3]．図 2.16 では， Work-Time C の速度向上率は PRP の速度向上率よりも 10%低い．しかし，クイックソートのように分岐数が小さく，分割に多くの時間を要する場合は，指定した部分木数に達するまでに多くの時間を要するため性能が低くなる可能性がある．一方，PRP は，クイックソートのように再帰関数への入力データ数がコンパイル時にわからないプログラムは機能制限により変換できない．また，PRP は並列化方式として MW 方式のみしか対応していないため，部分木の計算量が元々均等である場合にはオーバヘッドが避けられない．これに対して，Work-Time C では，PRP のような入力データ数に関する機能制限はなく，また，2.4 節で示したように容易に並列化方式を変更できる．以上より，Work-Time C は n 女王問題に対しては性能が 10%低いが，並列再帰プログラムの開発を容易でき，並列プログラムの開発者にとって並列プログラムの設計がしやすいという点で有用であると考える．

Machiavelli は C 言語ツールキットであり，通信ライブラリ MPI を用いて実装されている．

[3]図 2.16 の実験環境は，2.5 節で用いた実験環境とは異なり，3.5 節で用いた実験環境である．

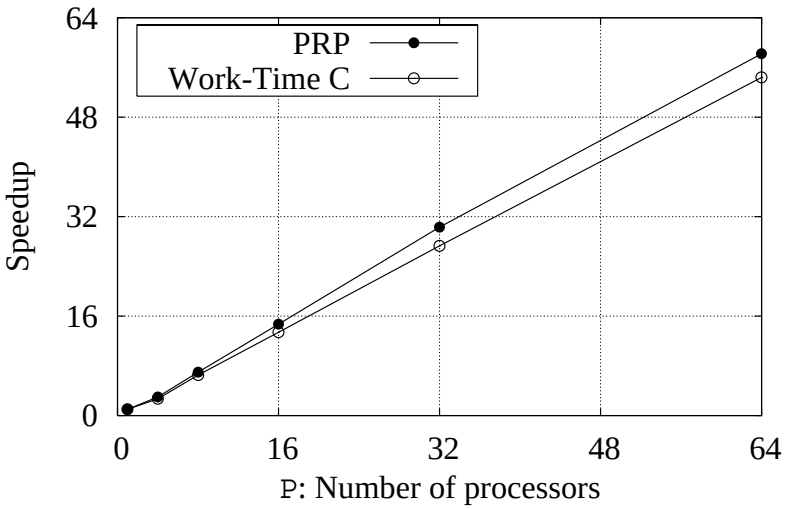


図 2.16: n 女王問題に対する Work-Time C と PRP の速度向上率

Machiavelli は，呼出時割当型の部分木割当方式と MW 方式を適用した頂点割当方式を併用する．まず，呼出時割当型の部分木割当方式に基づき，再帰木の部分木のプロセッサ集合を割り当てていく．プロセッサ集合のプロセッサ数は，再帰関数の入力引数のデータの大きさに比例する．再帰関数内では，そのプロセッサ集合でデータ並列計算が可能である．再帰的にプロセッサ集合を分割していき，集合要素数が 1 となるとそのプロセッサはワーカとして機能し，MW 方式を適用した頂点割当方式により動的負荷分散を行う． Machiavelli では，再帰関数内にデータ並列計算可能部分があるプログラムについては良好な性能の並列プログラムを記述可能である．しかし，クイックソートのようにデータ並列計算が容易でない部分が実行時間の大部分を占めるプログラムに対してはプロセッサ集合の分割処理がオーバヘッドとなる．

2.7 おわりに

本章では並列再帰の並列化方式を整理し，並列再帰アルゴリズムの効率的な実行に有用な情報を開発者が指定する手法を提案した．また，再帰アルゴリズムの特徴によって実行時間を短くできる並列化方式が異なることを示すことで，提案手法の有用性を示した．再帰はアルゴリズムを構築する上で重要な技法である．したがって，開発者がもつ一般的な知識を基に高性能な並列再帰プログラムを容易に記述できることは，並列プログラムの設計の幅を広げることにつながる．ゆえに，本章の成果は並列プログラムの開発者にとって有用であるといえる．

今後の課題として，開発者が指定できる並列化方式の種類の増加が挙げられる．例えば， 2.6 節 で示したように，本章で用いた並列化方式よりも他のコンパイラが用いる並列化方式の方が実行時間を短くできると判断できる場合， Work-Time C 言語で記述したプログラムをそのコンパイラ用の言語に変換することが考えられる．また，本章で採用した並列化方式においては，再帰関数に渡す引数の全データを，予め 1 つのプロセッサが保持している．そのため，並列計

算環境を利用する利点の 1 つである大量データの処理が行えない．データを最初から複数プロセッサに分散させた状態で並列再帰を実行する手法を考案することが重要な課題である．

第3章

マスタ・ワーカ型並列プログラムを高速実行できる実行パラメータ値を検出するための性能予測

3.1 はじめに

クラスタ計算 [13] およびグリッド計算 [21] に関する技術の発展に伴い，演算性能および通信性能が不均一な並列計算環境が増加している．このような並列計算環境に適したプログラミングパラダイムとして，マスタ・ワーカ（MW）方式がある．MW 方式は，使用可能なプロセッサ群を，マスタと呼ばれるグループとワーカと呼ばれるグループに分割し，マスタは並列処理する仕事（タスク）の生成およびワーカへのタスクの割当を担当し，ワーカは割り当てられたタスクを処理する手法である．MW 方式を用いることで，動的負荷分散を実現した高性能なアプリケーションを容易に開発できる．しかし，MW 方式はマスタ数に対してワーカ数が過剰である場合，マスタでの資源の競合によってプログラムの性能は低下する．MW プログラムを高速実行できるワーカ数を検出するための手段として，性能予測は有用である．性能予測とは，ある並列計算環境でプログラムを実行したときの性能（例えば実行時間）を，別の計算環境で予測することである．また，性能予測は，性能向上のための複雑な動作をする MW プログラムの開発の助けにもなる．

本章では，MW プログラムの性能解析を対象とした性能予測システムの構築を目的とする．本システムでは，単純な MW プログラムのみでなく，マスタ数の動的な調整や階層的な管理といった高度な MW プログラム [3, 16, 43, 52] も対象とする．本章ではメッセージ通信仕様 MPI[37] を用いた MW プログラムの性能を予測するために，以下の 3 点を導入した MW エミュレータ（MWE）を開発した．

D1： 並列計算モデルを用いた低オーバヘッドの性能予測

D2： 並列計算モデルの拡張によるマスタの通信オーバヘッドのモデル化

D3： プログラム実行時に決まる挙動の再現

D1 として MPI の通信をモデル化した LogGPS モデル [29] を使用した．また，D2 としてワーカからの到着メッセージの検査に要するマスタのオーバヘッドを並列計算モデルに追加した．さ

らに，D3 を実現するため，本章ではエミュレーションを用いた．MWE はプログラムの動作を忠実に再現するので，複雑な動作をする MW プログラムの性能評価が可能となる．

以降では，まず 3.2 節で関連研究について述べる．次に，3.3 節で MW プログラムの性能を予測するための考慮点について述べ，3.4 節で MWE の詳細について述べる．そして，3.5 節で MWE の適用実験を行う．最後に 3.6 節で本章のまとめを述べる．

3.2　関連研究

これまでに並列プログラムを対象とした多くの性能予測ツールが提案されている．MicroGrid [49] は，グリッドアプリケーションの複雑かつ動的な挙動を解析することを目的に，仮想的なグリッド環境を提供する． MicroGrid は，ネットワークの挙動を調べるために， TCP の輻輳制御などを再現できる詳細なネットワークシミュレーションを行う．しかし，詳細なネットワークシミュレーションは，通信がマスタに集中する MW プログラムにとって，予測できない大きなシミュレーションオーバヘッドを伴う．これは，マスタのタスク割当の動作を妨げ， MW プログラムの性能予測の精度を低下させる．したがって，MW プログラムの性能を精度よく予測するためには，オーバヘッドの小さい予測手法が適する．

オーバヘッドの小さい予測手法として，LogP モデル [15] に代表される並列計算モデルの利用がある．LogP モデルは以下の 4 つのパラメータによってメッセージ通信をモデル化する．

- L: 通信遅延．送信プロセッサから受信プロセッサまでのネットワーク転送に要する時間．
- o: 通信オーバヘッド．メッセージを送信または受信するときにプロセッサが占有される時間．この間プロセッサは他の処理を行うことはできない．
- g: メッセージを連続して送信もしくは受信するときの最短の時間間隔．
- P: プロセッサ数

また，LogGP モデル [4] は，以下のパラメータを LogP モデルに追加することで，長いメッセージの扱いを可能としたものである．

- G: 長いメッセージを送信するときの 1 バイトあたりの時間間隔．

LogGP モデルは，文献 [1, 46] において，ガウスの消去法などの並列プログラムに対して良い精度でその性能を予測することが示されている． LogGPS モデル [29] は，LogGP モデルに以下の 2 つのパラメータを追加することで，メッセージ通信仕様 MPI における複数の通信プロトコルの通信をモデル化する．

- s: 1 つのパケットで送信できるメッセージ長の上限値．
- S: 送信プロセッサが受信プロセッサと同期を必要とせずに送信できるメッセージ長の上限値．

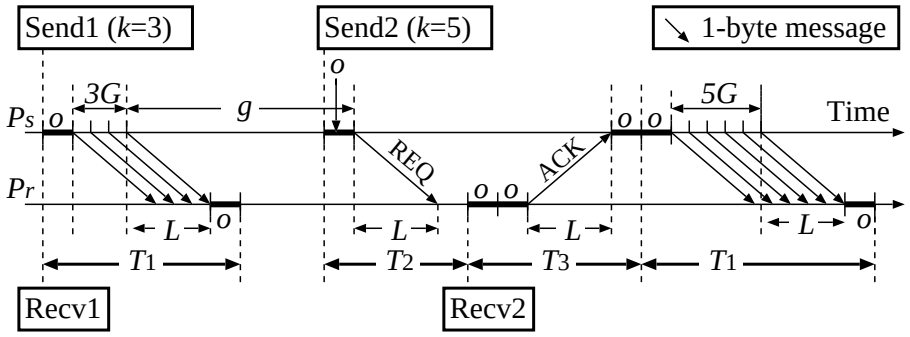


図 3.1: LogGPS モデルにおける 1 対 1 通信の例（$S = 4$）

図 3.1 に $S = 4$ としたときの LogGPS モデルにおける 2 つの通信（Send1 および Send2）の例を示す．ここで，k はメッセージ長を表す．$k \leq S$ のとき，プロセッサ P_s はプロセッサ P_r に T_1 のコストで非同期メッセージ（Send1）を送信する．$k > S$ のとき，P_s は P_r と同期をとった後にメッセージを送信する（Send2）．この場合の通信コストは，T_1 と同期コスト T_2 と T_3 の和となる．

一方，LoPC モデル [22] や LoGPC モデル [40] のように，資源の競合によるコスト C を追加したモデルも提案されている．LoPC モデルおよび LoGPC モデルは，それぞれプロセッサの競合およびネットワークの競合をモデル化する．

このように，LogP モデルに基づく多くの並列計算モデルが提案され，並列プログラムの性能を精度良く表現できることが示されている．しかし，これらのモデルでは MW プログラムへの適用と評価については言及していない．性能予測ツール CLUE[36] は，LogP と同等のモデルを用いて MW プログラムの性能を予測している．しかし，CLUE は SMP 構成の PC5 台から成る PC クラスタを用いて精度良く性能予測できることを示しているが，MW プログラムの最適なワーカ数の探索や複雑な MW プログラムの動作の解析ができるか否かは明らかにしていない．

3.3 マスタ・ワーカ型並列プログラムを高精度に性能予測するための考慮点

本節では，MW プログラムを高精度に性能予測するための考慮点について述べる．

3.3.1 並列計算モデルの利用による予測オーバヘッドの低減

MW プログラムの実行においては，マスタの性能がプログラム全体の性能を決定する．したがって， MW プログラムの性能を精度よく予測するためには，マスタの動作を正確に予測する必要がある．3.2 節の議論より，本研究では並列計算モデルを用いて性能予測する．使用する並列計算モデルとして LogGPS モデルを選択した．なぜならば， LogGPS モデルは文献 [29] において，MPI を用いて記述した並列プログラムの性能を良い精度で予測することが実証されてい

るためである．

本研究では，LoPC モデルや LoGPC モデルのように資源の競合を表現できる並列計算モデルは存在するが，競合を表現しない LogGPS モデルを選んだ．プログラムの性能に対するネットワーク競合の影響が小さい並列計算環境においては，LogGPS モデルは LoPC モデルや LoGPC モデルに対して 2 つの利点がある．1 つは，LoPC モデルや LoGPC モデルと比べてパラメータが単純であることである．パラメータが単純であるため，予測に要するオーバヘッドが小さく，MW プログラムの実行時に決まる挙動の再現（3.3.3 節）においてマスタの動作への影響を小さくできる．もう 1 つは，LogGPS モデルのパラメータはハードウェアのみにしか依存しないことである．LoPC モデルや LoGPC モデルでは，資源の競合に係るオーバヘッド C はハードウェアとソフトウェアの両方に依存する．例えば，メッセージの送受信率のようにソフトウェアに依存するパラメータを必要とする．複雑な MW プログラム [16, 52, 43, 3] においては，マスタを動的な増減によって通信パターンが変化するため，このようなパラメータを導くことは容易でない．

3.3.2　並列計算モデルの拡張によるマスタの通信オーバヘッドの表現

MW プログラムのスケーラビリティ解析においては，以下の I1 および I2 を検出することが重要である．

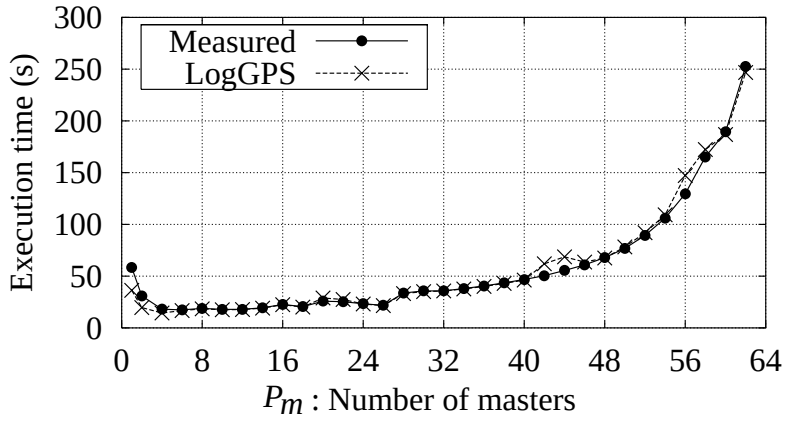
I1： 与えられたプロセッサ数に対して最も実行時間を短くできるマスタ数

I2： 与えられたタスク集合に対して最も実行時間を短くできるプロセッサ数

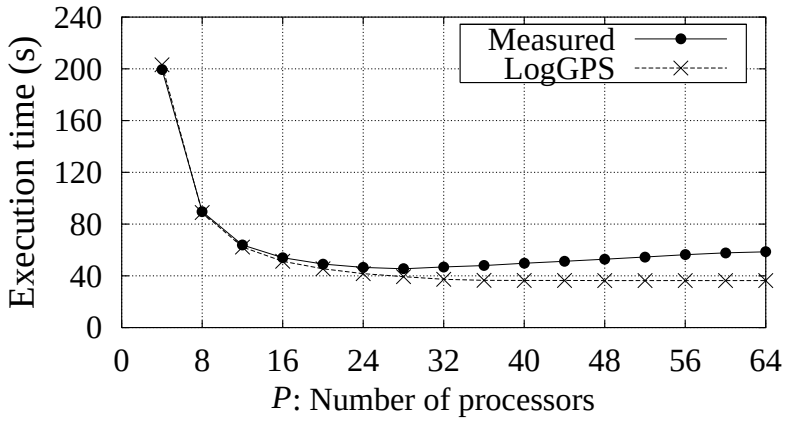
図 3.2 に，MW 方式に基づくマンデルブロ集合探索問題プログラムの実行時間を示す．ここで，P_m はマスタ数を表し，ワーカ数は $P - P_m$ である．図 3.2(a) が示すように，LogGPS モデルによって I1 を検出できる．一方，I2 を検出するためには，予測対象環境上のプロセッサ数 P を用いて求めた並列計算モデルのパラメータ値を使用しなければならない．そうでなければ，図 3.2(b) が示すように実測と予測の差は大きくなり，MW プログラムのスケーラビリティを調べることはできない．以降では，この差が生じる原因とこの問題の解決案について述べる．

図 3.2(b) において，P の増加とともに実測実行時間が増加する原因は，P の増加に伴うマスタとワーカの通信時間の増加である．表 3.1 に，3 つの MPI の実装についてイーサネット上での 1 バイトメッセージの往復時間（Round Trip Time, RTT）を示す．また，RTT の計測方法を 図 3.3 に示す．表 3.1 が示すように，全ての MPI の実装において P の増加とともに RTT も増加している．ここで，RTT の増加はネットワーク資源の競合によるものではないといえる．なぜならば，図 3.3 が示すように，RTT の計測は P によらずメッセージの送受信は 2 台のプロセッサ（P_1 および P_2）のみで行い，残りの $P-2$ 台のプロセッサは何もしないためである．図 3.2(b) における実測と予測の差は，LogGPS モデルがネットワーク競合によらない通信時間の増加を表現していないことに起因する．

MW プログラムのスケーラビリティを性能予測によって調べる場合，マスタとワーカの通信時間を正確に見積もることが重要である．これは，MW プログラムにおいては通信時間の見積



(a) $P = 64$



(b) $P_m = 1$

図 3.2: MW 方式に基づくマンデルブロ集合探索プログラムの実行時間

表 3.1: RTT_P：プロセッサ数 P における 1 バイトメッセージの往復時間

MPI implementation	RTT_P (μs) $P=2$	$P=16$	$P=64$	σ^*: Increasing rate (%)
MPICH [26]	144.0	152.1	185.8	29.0
LAM [12]	125.8	147.1	194.8	54.8
MPICH-SCore [42]	95.4	97.6	99.2	3.98

* $\sigma = (RTT_{64}/RTT_2 - 1) \cdot 100$

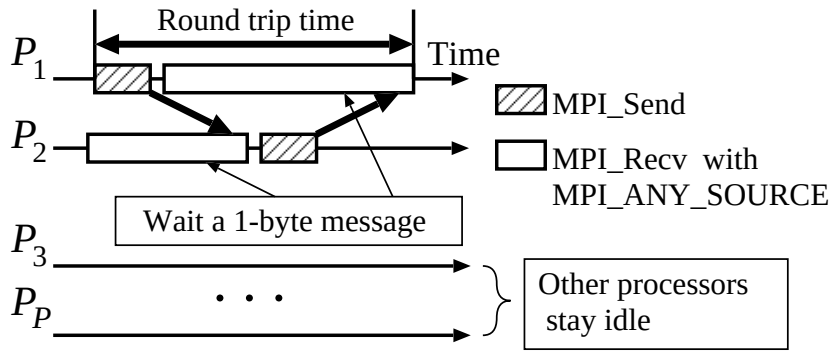


図 3.3: 通信の往復時間

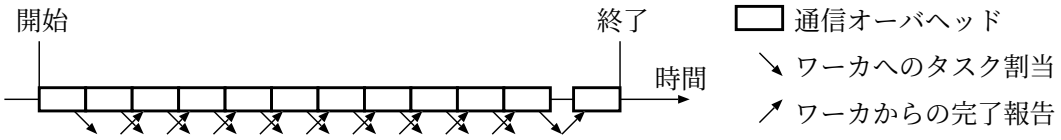


図 3.4: マスタの過負荷状態における実行状況

もりの誤差が，プログラム全体の予測性能に大きく影響するためである．図 3.4 に示すように，ワーカ数が過剰であるなどの要因によってマスタが過負荷状態にある場合，マスタは常にワーカと通信を行うこととなり，マスタの実行において通信オーバヘッドの占める割合が大きくなる．従来，LogGPS モデルのような並列計算モデルは，このような状態が発生しない並列プログラムを対象としていた．そこで本研究では，P の増加による RTT の増加を表現するために，LogGPS モデルが表す通信オーバヘッド o（$o = o' + Ok$）を P の線形関数に拡張する．すなわち，$o = o'_a + o'_b P + Ok$ と表す．ここで k はメッセージ長，o' と O は LogGPS モデルにおける定数，o'_a と o'_b は複数の小さい P を用いて測定した o' から求めた定数である（求め方は 3.5 節において述べる）．

線形表現した通信オーバヘッドは，マスタがワーカからのメッセージ到着を検査するオーバヘッドに対応するといえる．例えば，MPICH の `MPI_Recv` はその内部で `select` システムコールを呼び出しており， Linux と FreeBSD において `select` システムコールはメッセージの到着待ち状態にあるソケットから線形探索によって到着メッセージの有無を検査している．各プロセッサがもつ探索対象とのソケットの数は $P-1$ であるため，P の増加とともに RTT は線形に増加する．したがって，通信オーバヘッドの線形表現は妥当であるといえる．線形探索のオーバヘッドの有無は，通信ライブラリおよび OS の実装に依存する．しかし，多くの場合において線形探索のオーバヘッドは存在すると考えられる．なぜならば，一般に MW プログラムにおいて，マスタはいつ，どのワーカからメッセージが到着するかを知らず，マスタはメッセージの到着判定を全てのワーカに対して行う必要があるためである．したがって，通信オーバヘッドの線形表現は，多くの場合において有効であると考えられる．

3.3.3 動的に決まるプログラムの挙動の再現

性能向上のための複雑な処理を伴った複雑な MW プログラムの性能を精度よく予測するためには，プログラムの動的な挙動を正確に再現する必要がある．なぜならば，このような MW プログラムは，マスタ数の動的な増減など，実行状況に応じてプログラムの動作を変更し，動的負荷分散の効率を上げることでプログラムの性能を向上させているためである [3, 16, 52, 43].

本研究では，プログラム実行時に決まる挙動を再現するために，エミュレーションによる性能予測手法を採用した．プログラムの動作が静的に決定する並列プログラムに対しては，コンパイラによるソースコードの解析によってプログラムの実行を省略する手法が提案されている [2]．しかし，MW プログラムの動作は動的に決定するため，この手法を適用すると正確な性能予測ができなくなる．したがって，エミュレーションという手法を採用した．

また，エミュレーションは性能不均一な PC クラスタにおける性能予測にも有用である．本研究ではタスク割当を動的に決定する並列プログラムを対象としているので，静的な解析 [9] や実行履歴の解析 [4, 29, 46] に基づく手法では性能不均一な PC クラスタにおけるタスク割当を正確には再現できない．これは，性能不均一な PC クラスタにおける性能予測に対して致命的である．なぜならば，性能不均一な PC クラスタにおいて，タスクを割り当てるワーカが異なるということは，タスクの処理時間およびマスタとワーカ間の通信時間も異なるからである．すなわち，どのワーカへタスクが割り当てられるかがわからなければ，通信時間を正確には予測できない．したがって，プログラム実行に決まる挙動を正確に再現することは，MW プログラムの性能を高精度に予測する上で重要である．

3.4 考慮点に基づいた性能予測の流れ

本研究では，考慮点に基づく性能予測システム MWE を実装した．本節では，MWE による MW プログラムの性能予測の流れについて述べる． 図 3.5 に MWE による性能予測の手順を示す．MWE は入力として，(1) MWE ライブラリをリンクした実行プログラムおよび (2) 対象環境の性能情報の 2 つをもつ．(1) におけるライブラリのリンクは， 図 1.4 において並列化コンパイラが行う．(2) では LogGPS モデルのパラメータで表した通信性能と各ノードの相対的な演算速度比を記述する．このように，MWE では対象プログラムの修正を必要としない．性能予測を行いたいユーザは，生成されたプログラムに対して実行パラメータ値を指定して実行することで，指定した実行パラメータ値で MW プログラムを実行したときの予測実行時間を出力する．このようにして得た様々な実行パラメータ値に対する予測実行時間をユーザが比較することで，実行パラメータ値の検査を行う．

MWE は，プログラムの実行中に対象環境での実行をエミュレートする．図 3.6 にエミュレーションの例を示す．ここで，P_s は MPI_Send によって P_r に 1 バイトのメッセージを送信する．まず，P_s は，文献 [29] で述べられている通信コストの定義を用いて， `MPI_Send` の完了時刻 T_s とメッセージの到着時刻 T_a を求める．そして，P_s は T_a の値を送信メッセージとともに P_r に送信する．その後，P_s は T_s までアイドル状態となる．一方，P_r は受信メッセージから T_a の値を取り出し，`MPI_Recv` の呼出時刻 T'_r と T_a を用いて `MPI_Recv` の完了時刻 T_r を求める．

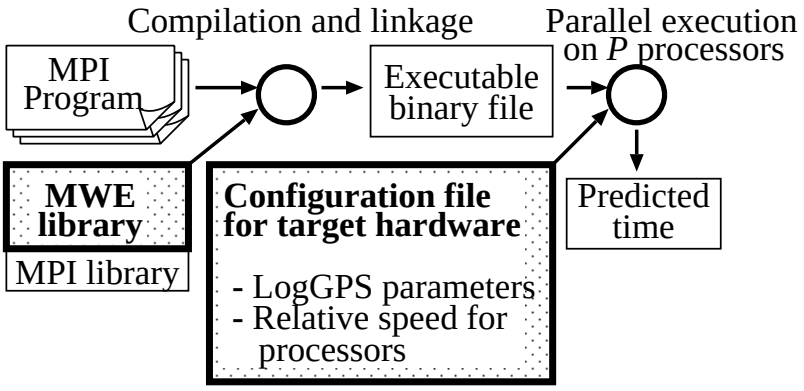


図 3.5: MWE による性能予測

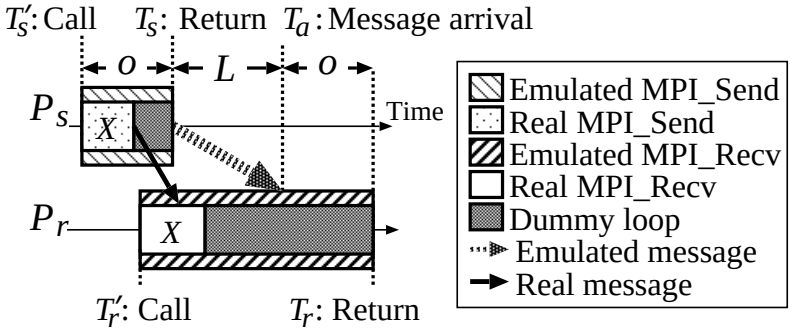


図 3.6: 1 バイト通信のエミュレーション

時刻 T_s と T_r において，P_s と P_r はそれぞれ `MPI_Send` と `MPI_Recv` を終了する．

図 3.6 に示すように，現在の MWE はエミュレーションにおいて予測オーバヘッド X を要する．X は完了時刻の算出コストと到着時刻の通信コストである．したがって，MWE は X を $T_s - T_s'$ や $T_r - T_r'$ に隠蔽するために性能予測の対象ネットワークよりも高速なネットワークを必要とする．ここで $T_s - T_s'$ と $T_r - T_r'$ は，それぞれエミュレートした `MPI_Send` と `MPI_Recv` のオーバヘッドを表す．X が $T_s - T_s'$ および $T_r - T_r'$ を超えたとき，以下の 2 つの問題が起こる．

(1) 後続のイベントが遅れる．この遅れは実際の実行とエミュレーションにおけるイベント発生のタイミングのずれによるものである．このとき，このずれを解消するためには，MWE は予測実行時間を調整する必要がある．

(2) MW プログラムの動的なタスク割当によって各プロセッサの通信数は異なる．したがって，イベント発生のずれはプロセッサ毎に異なる．メッセージの送信と到着の正しいタイミングを保持するためには，MWE は各プロセッサでずれを同期する必要がある．

性能予測において上述のネットワーク速度に関する制限は問題であるが，CLUE [36] で使用されている 2 つの手法を MWE に適用することでこの問題は解決できると考えられる．一方は

表 3.2: イーサネット上の MPICH における LogGPS モデルのパラメータ値

	L (μs)	G (μs)	o for `MPI_Send` (μs)	o for `MPI_Recv` (μs)
MWE	50.0	0.0268	$12.1 + 0.182P + 0.0708k$	$12.1 + 0.182P + 0.0722k$
LogGPS			$13.0 \quad + 0.0706k$	$13.0 \quad + 0.0723k$

仮想時刻の管理であり，他方は離散事象シミュレーションである．仮想時刻は実際の時間と独立に管理されるため，問題 (1) を解決できる．離散事象シミュレーションは全プロセッサ間で仮想時刻の同期をとるため，問題 (2) を解決できる．

3.5 適用実験

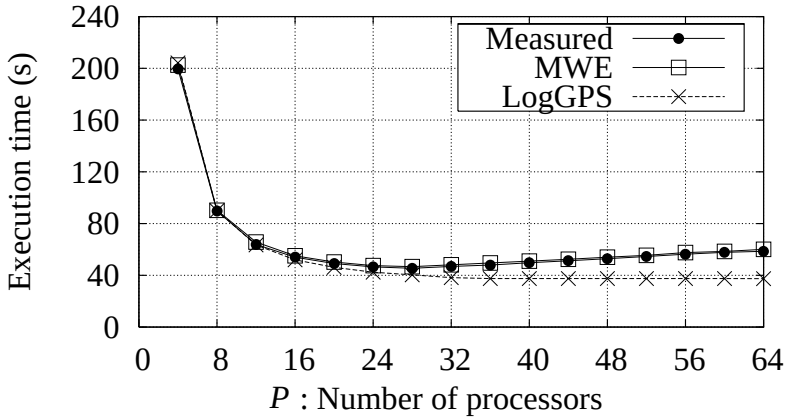
本節では，予測精度および予測オーバヘッドの観点から MWE を評価する．対象プログラムとしてマンデルブロ集合探索問題（MASE）と人工股関節の可動域計算（ROMS）[32] を用いた．両プログラムにおいて，各タスクは独立であり，各タスクの計算量は動的に決まる．MASE におけるタスクは，複素平面上の点がマンデルブロ集合に含まれるか否かの判定であり，ROMS におけるタスクは，3 次元回転角における大腿骨と人工股関節の衝突判定である．本実験では，単一マスタ（SI 方式），複数マスタ（ML 方式）および動的マスタ（DY 方式）の 3 つの実装を MASE と ROMS に適用した．ML と DY の詳細については後述する（3.5.3 節）．

実験環境は，64 台の PC から成る PC クラスタを用いた．各 PC は CPU として Pentium III 1GHz をもち，ネットワークはミリネット高速ネットワーク [11]（2Gb/s）とイーサネット (100Mb/s) で接続されている．本実験では，イーサネット上での MPICH プログラムを，ミリネット上で同数のプロセッサ数 P を用いた MPICH-SCore[42] プログラムでエミュレートした．

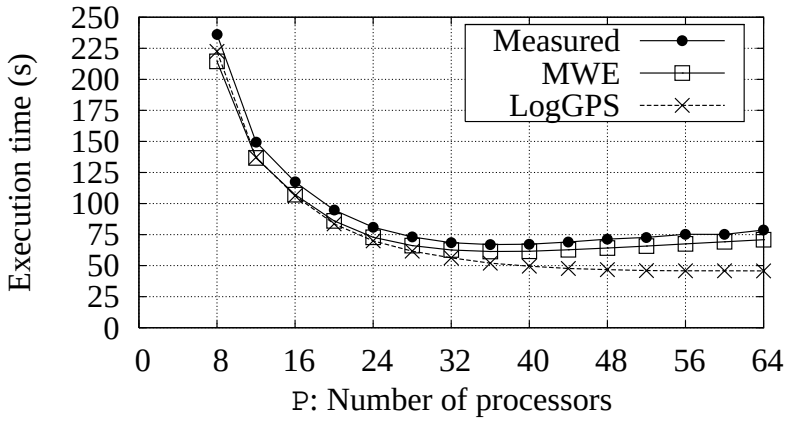
表 3.2 に，イーサネット上の MPICH において 8 台のプロセッサを用いて測定した LogGPS モデルのパラメータを示す．P と k はそれぞれプロセッサ数およびメッセージ長（バイト）を表す．MASE および ROMS では同期通信は発生しないため，ここでは非同期通信のためのパラメータを示す．近年の計算機では，g は o に含まれるため [29, 40]，g は記載していない．o'_a と o'_b を導出するため，まず $P = 2$ と $P = 8$ における LogGPS モデルの定数 o' を文献 [29] の方法にしたがって測定した．次に，変数 o'_a と o'_b に関する以下の連立方程式を解くことで，o'_a と o'_b の値を得る．

$$\begin{cases} o'_a + o'_b \cdot 2 &= 12.48 \quad (\text{マイクロ秒}), \\ o'_a + o'_b \cdot 8 &= 13.57 \quad (\text{マイクロ秒}). \end{cases}$$

表 3.2 において，P の項は o に対して無視できない大きさである．例えば，$k = 1$ において P が 2 から 71 に増加したとき，o は 2 倍になる．



(a) MASE-SI



(b) ROMS-SI

図 3.7: SI 方式における実測実行時間と予測実行時間

3.5.1　実行パラメータ値の変動に対する予測精度の評価

MW プログラムの性能飽和点に対する MWE の予測精度を評価するために，マスタが性能ボトルネックとなるタスクの粒度を用いて MW プログラムの性能を予測した．図 3.7 に SI 方式における実測と予測の実行時間を示す．MWE は，MASE および ROMS のそれぞれについて 3%および 10%以内の正確な予測を示している．また，MWE は性能飽和点も正確に予測している．MASE および ROMS の実測実行時間は，それぞれ $P = 28$ および $P = 36$ が最良の実行時間を示しており，MWE による予測実行時間もはその性能飽和点を示している．一方，LogGPS モデルは，MASE および ROMS の $P = 64$ において，実測との誤差がそれぞれ 38%および 42%と大きい．これは以下のように説明できる．MASE においてはマスタとワーカ間で 1,048,576 回の往復通信を行うため，マスタの実行には少なくとも $2o \cdot$ 1,048,576 マイクロ秒を要する．表 3.2 の o に $P = 8$ と $P = 64$ を代入したとき，それらの差は $1{,}048{,}576 \cdot 2 \cdot 0.182 \cdot (64 - 8)/10^6 = 21.4$ 秒である．この差は，$P = 64$ における実測実行時間と LogGPS の予測実行時間の差 21.2 秒に

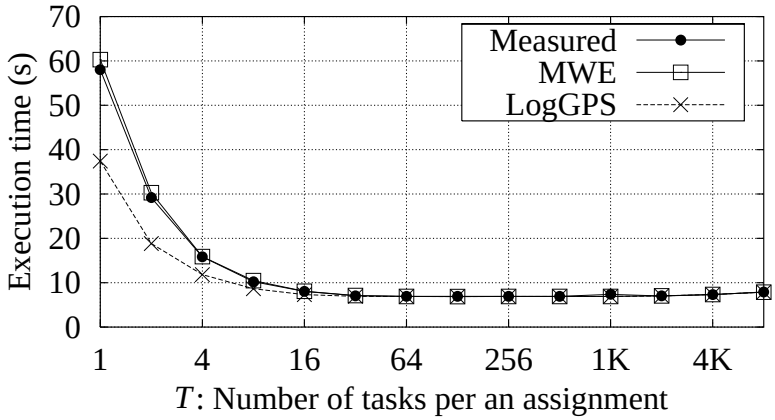


図 3.8: タスクの同時割当数の違いによる実測実行時間と予測実行時間（SI 方式の MASE）

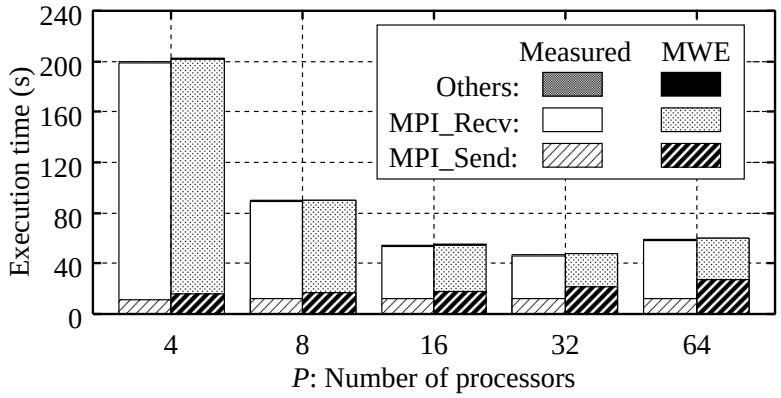
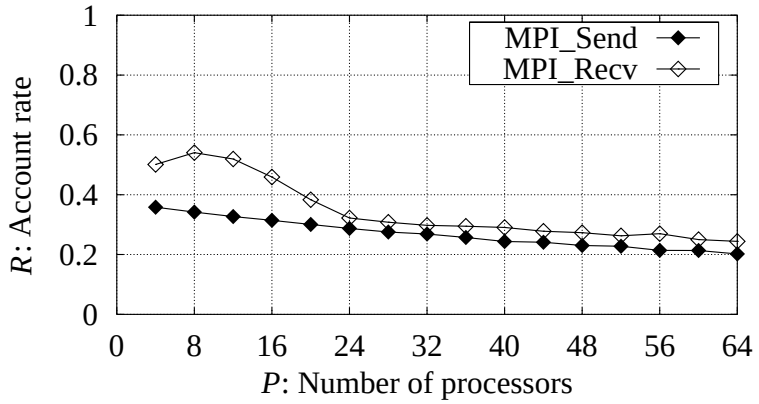


図 3.9: マスタの実行時間の内訳（SI 方式の MASE）

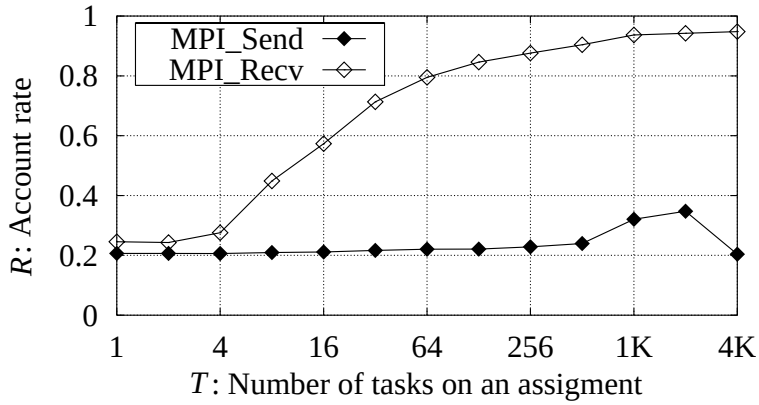
近い．このように，P が通信オーバヘッドに与える影響は大きい．したがって，MW プログラムのスケーラビリティ解析のためには，通信オーバヘッドを P について線形表現することが重要である．

次に，MASE の詳細について調べる． 図 3.8 に $P = 64$ においてタスクの同時割当数 T の変化による実測実行時間と予測実行時間を示す．ここで，マスタは 8 バイトの送信と $4T + 8$ バイトの受信をワーカと繰り返す．図 3.8 では，$T = 1$ において，LogGPS による予測と実測との差は大きい．一方，MWE に関しては，予測に用いた全ての T において実測と予測の差は小さく，予測誤差は最大 4%である． LogGPS による予測と実測の差は， T の増加に伴って小さくなっている．これは，マスタはワーカと 1,048,576/T 回の通信を行うことから，実測と予測の差は $21.4/T$ 秒と表せるためである．したがって，T の増加とともに実測と予測の差は小さくなり，LogGPS の予測精度は良くなる．

図 3.9 に，MASE におけるマスタの実行時間の内訳を示す．図 3.9 より， MWE は全ての P において精度よく合計実行時間を予測できるが， P の増加にともなって `MPI_Send` の時間を長



(a) $T = 1$



(b) $P = 64$

図 3.10: マスタにおける予測オーバヘッドの占有率（SI 方式の MASE）

く，MPI_Recv の時間を短く予測している．この理由は，提案手法で用いた並列計算モデルでは送信オーバヘッドと受信オーバヘッドは等しいと仮定しているためでる．この仮定は並列計算モデルを簡潔にするためのものであるが，実際の通信では送信オーバヘッドについては P に比例するコストはない．したがって，マスタにおいて MPI_Send と MPI_Recv の呼出回数が等しいような MW プログラムについては，MWE は精度良く性能予測できる．しかし，マスタが動的に生成および消滅するなどして，マスタにおける MPI_Send と MPI_Recv の呼出回数が異なる MW プログラムに対しては， MWE は精度よく予測できない可能性がある．この問題を解決するには，o'_a と o'_b のそれぞれを，送信用と受信用に区別すればよい．

3.5.2　予測オーバヘッドの評価

本節では MWE の予測オーバヘッドについて評価する．図 3.10 にマスタにおける予測オーバヘッドの占有率 R を示す．ここで，MPI_Send については $R = X/(T_s - T'_s)$ であり，MPI_Recv

については $R = X/(T_r - T'_r)$ である（各記号については図 3.6 を参照）．$R < 1$ のとき，エミュレートした通信命令が予測オーバヘッドを隠蔽する．すなわち，精度のよい性能予測をするために予測オーバヘッドは十分小さいといえる．図 3.10 より，R は 0.20 から 0.94 の範囲である．したがって，精度のよい予測を行うために予測オーバヘッドは十分小さい．

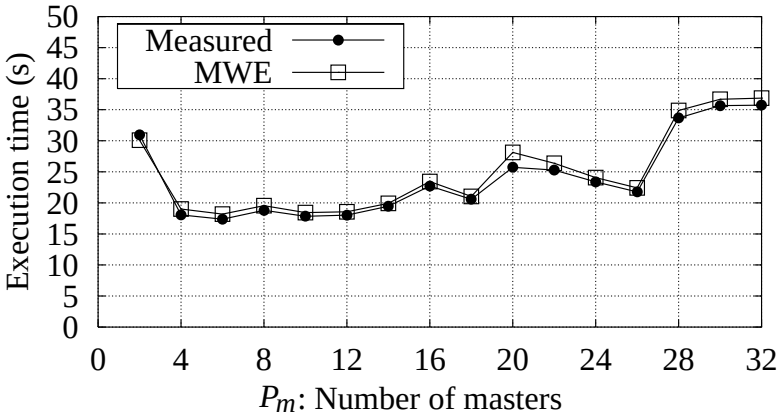
図 3.10(a) では P の減少とともに `MPI_Recv` （メッセージ長は $4T + 8$ バイト）の R は増加し，図 3.10(b) では T の増加とともに `MPI_Recv` の R は 1.0 に近付いている．この理由は `MPI_Recv` における同期時間 W が増加するためである． W は X と $T_r - T'_r$ の両者に含まれる．同期時間 W の間，マスタはワーカからのタスクの要求を待つ．本実験においては，1 回の `MPI_Recv` あたりの W の平均は， $T = 1$ かつ $P = 64$ において $W = 3.38$ マイクロ秒であり，$T = 4$K かつ $P = 64$ において $W = 12.9$ マイクロ秒であった．このように，マスタがワーカからの要求を待つとき，R は増加する．しかし，ミリネットのオーバヘッドがイーサネットのオーバヘッドより小さいとき， R は 1 を超えない．$P = 64$ における LogGPS モデルのパラメータを用いることで，メッセージ長 k が増加したとき，同期時間を無視した占有率 $R' = (X - W)/(T_r - T'_r - W) = (2.9 + 0.00767k)/(12.1 + 0.182 \cdot 64 + 0.0722k)$ は 0.11 に近付く．ここで，$X - W$ および $T_r - T'_r - W$ は，それぞれミリネットおよびイーサネットでの `MPI_Recv` のオーバヘッドを表す．

一方，マスタは全ての P と T において 8 バイトの送信を繰り返しているが，図 3.10(b) では $512 \leq T \leq 2$K かつ $P = 64$ において `MPI_Send` は不規則な曲線を示している．X の増加は R の増加を引き起こすため，この不規則な曲線は MPICH-SCore のフロー制御が働いたためであると考えられる．この場合，マスタは 8 バイトの送信バッファを確保するまでに，全ての $4T + 8$ バイトのメッセージを受信することになる．

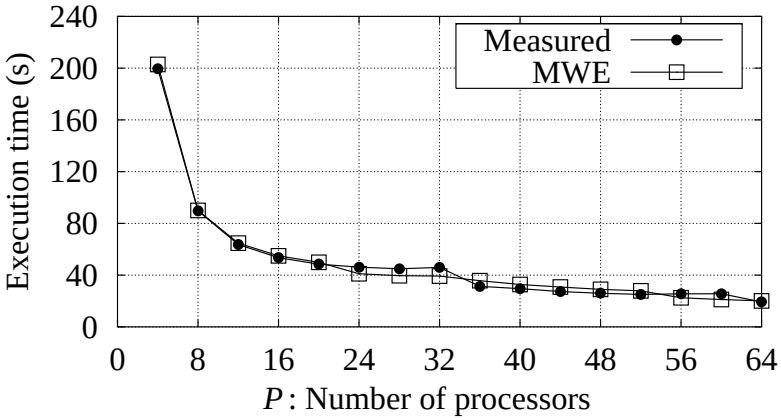
3.5.3 動的に決まる挙動の再現による効果の評価

ML 方式および DY 方式に基づく MW プログラムの性能を，MWE を用いて予測した．ML 方式では，P_m 個のマスタが P/P_m 個のワーカを管理する．ここで $2 \leq P_m \leq P/2$ である．DY 方式では，プログラムの実行中にマスタ数は変化する．DY 方式においては，各マスタは自身が管理するワーカ集合とタスク集合を二分割し，自身が管理するワーカの 1 つ w に分割したワーカ集合とタスク集合を割り当てることができる．w は，割り当てられた全てのタスクを処理し終えるまで，割り当てられたワーカ集合のマスタとして動作する．マスタは，自身が担当する全てのワーカの割当待ち時間の統計に基づいて，マスタを分割するか否かを決定する．本実験では，MWE が DY 方式の挙動を精度良く再現することを確かめるために，MASE を選択した．なぜならば，ROMS と比べて MASE はタスク 1 個あたりの実行時間が短いため，マスタにおけるワーカとの通信頻度が高くなりやすく，新しいマスタが生成される傾向が強いためである．

図 3.11(a) は $P = 64$ における ML 方式の実験結果を示す．図 3.11(a) より，実測実行時間は不規則な形状の性能曲線を示しているが，MWE の予測誤差は 9%以内であり，良い精度で予測できている．これは，MWE はエミュレーションによって実測とほぼ同じタスク割当を実現できるためである．例えば， $P = 16$ では不均等な負荷となるタスク分割によって実測実行時間は増加しているが， MWE はその増加を予測できる．次に，図 3.11(b) に DY 方式の結果を示す．



(a) ML: Multiple master implementation



(b) DY: Dynamic master implementation

図 3.11: $T = 1$ における MASE の実測実行時間と予測実行時間

図 3.11(b) より，MWE は実測実行時間を精度よく予測できている．また，図 3.11 の結果より，最適なマスタ数 P_m を検出できれば， ML 方式は DY 方式よりも良いの性能を示すことがわかる．このように，MWE は， MW プログラムの静的解析や実行履歴に基づく手法では容易にはわからない不規則な性能変化を精度良く予測できる．

最後に，$P = 64$ の DY 方式の実測と予測におけるタスクの分割のされ方を比較する．図 3.12 に MASE で求めたマンデルブロ集合の画像を示し，図 3.13 にマンデルブロ集合の画像において各マスタが担当した領域を表すタスク分割図を示す． 図 3.13 において，黒い水平線で分割した領域は， 1 つのマスタが担当したタスクの集合を表す．DY 方式では，ワーカの待ち時間をもとにマスタを増やすか否かを判断する．このとき，マスタの通信オーバヘッドはワーカの待ち時間に大きく影響する．また，タスク分割図は予測精度の正しさに影響される．ここで，予測精度の正しさとは（1）水平線の位置（新しいマスタを生成した場所）および（2）水平線の数（生成したマスタの数）の 2 点によって定義する．図 3.13(a) および (b) は，R_D においては

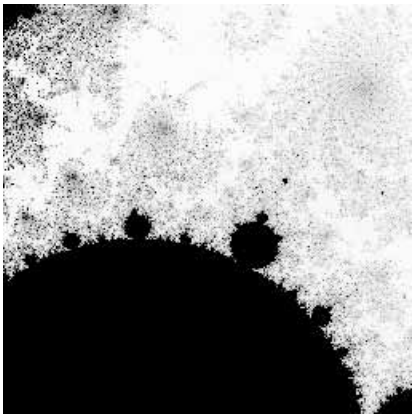
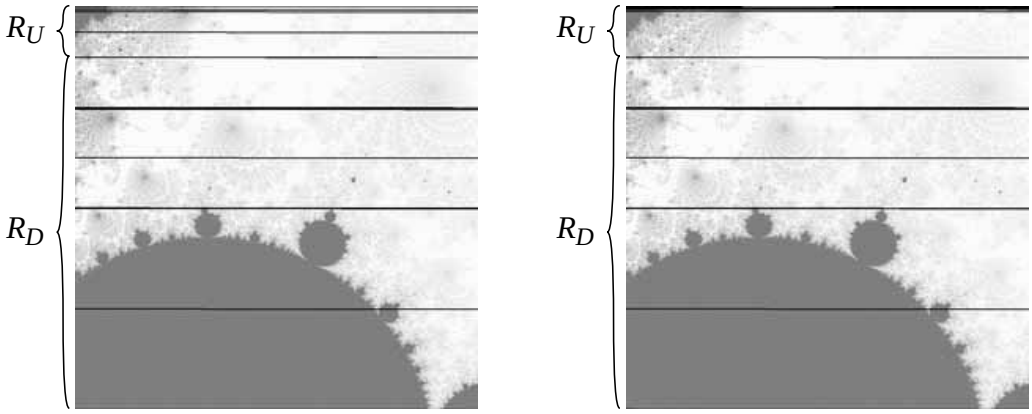


図 3.12: MASE で求めたマンデルブロ集合の画像



(a) Measured (b) MWE

図 3.13: DY 方式の MASE におけるタスク分割の比較

分割数は同じであるが，R_U においては分割数が異なっている．R_D において水平線の位置の差は，1024×1024 ピクセルの画像に対してわずかであり，水平線の数は同じである．一方 R_U では，MWE においてマスタが生成されないために水平線の数が違っている．しかし，この違いはマスタの分割の 4 段階目で起こっており，プログラムの性能への影響が大きい上位の段階では R_D と同様にマスタの分割を再現できている．以上の理由により，図 3.11(b) が示すように MWE は DY 方式の実行時間を精度良く予測できている．したがって，マスタの分割数は異なるが，実行時間を予測する上で図 3.13(a) および (b) は似ていると言える．

以上より，MWE は高度な MW プログラムの実行時間だけでなく，プログラム実行時に決まる挙動も再現でき，複雑な動作をする MW プログラムを評価する上で有用である．

3.6　おわりに

本章の貢献は，MW プログラムの高精度な性能予測を実現するために，以下の 3 点を示したことである．

(C1) MW プログラムの性能を精度良く予測するための 3 つの考慮点を示した．

D1： 並列計算モデルの利用による通信オーバヘッドの低減

D2： 並列計算モデルの拡張によるマスタの通信オーバヘッドのモデル化

D3： プログラム実行時に決まる挙動の再現

(C2) D1 および D2 の重要性を実験によって示した．実験結果は，64 台の PC から成る PC クラスタにおいて，10%以内の誤差で MW プログラムの性能を予測できた．また，MW プログラムのスケーラビリティ解析において，線形関数によってマスタの通信オーバヘッドを表現することの重要性を示した．

(C3) 複雑な動作をする MW プログラムへの適用実験により，D3 の重要性を示した．

本研究で用いた並列計算モデルは LogP モデルの拡張であるため，提案手法は他のメッセージ通信プログラムにも応用できる．特に，集合通信を用いることで特定のプロセッサに通信が集中するようなメッセージ通信プログラムには有効であると考える．

現在の MWE はエミュレーションに基くため，MWE を動作させるために予測の対象ネットワークよりも高速なネットワークが必要となる．今後の課題として，この点を改善し，遅いネットワークを使用して速いネットワークでの性能予測を可能とすることが挙げられる．

第4章

実行パラメータ値の高速な検出を目指したマスタ・ワーカ型並列プログラムの性能予測の高速化

4.1 はじめに

クラスタ計算技術およびグリッド計算技術の発展にともない，演算性能が不均一な計算ノードからなる並列計算環境が増加してきている．このような環境に適したプログラミングパラダイムとして，マスタ・ワーカ（MW）方式がある．MW 方式では計算ノード群をマスタとワーカに分類し，マスタは仕事（タスク）の生成とワーカへのタスク割当を担当し，ワーカは割り当てられたタスクを処理する．ワーカが処理結果をマスタへ返すと，マスタはそのワーカへ別のタスクを割り当てる．この動作を繰り返すことで，MW 方式ではワーカ間の計算負荷を動的に分散（動的負荷分散）する．MW 方式は実装が容易でありながら，動的負荷分散による高性能な計算を実現できるため，アプリケーションの並列化方式として広く利用されている [3, 7, 8, 31, 47, 53, 55]．しかし， MW 方式に基づく並列プログラム（**MW プログラム**）の性能は，マスタが管理するワーカの数やタスクの一括割当数といった要因（**実行パラメータ**）に依存する．実行パラメータの値が不適切な場合，マスタがボトルネックとなって MW プログラムの性能が低下する可能性がある．このような性能低下を引き起こさない実行パラメータ値を検出する手段として性能予測は有用である．性能予測によって様々な実行パラメータ値で MW プログラムを実行したときの性能を予測することで，対象としている実行環境において MW プログラムの実行時間を短くできる実行パラメータ値を検出することができる．

並列プログラムの性能を予測する既存の手法として，プログラムの動作を正確に再現することで高精度に予測する直接実行方式 [36, 39, 45] と，ソースプログラムからプログラムを動作を解析することで高速に予測する推測方式 [2, 51] がある．これらの手法は，MW プログラムに対しては高精度な予測と高速な予測を両立することはできない．直接実行方式では，性能予測のためにプログラム全体を実行するため，少なくともプログラムの実行時間と同等の時間を予測に要する．また，直接実行方式では予測対象のプログラムの全体を実行するため，予測が終了した時点でプログラムの計算結果を得ることができる．したがって，予測を終えた時点では予測によって得た適切な実行パラメータ値を使ってプログラムを実行する必要はなくなっている．一方，推測方式では，計算負荷をソースプログラムから静的に解析できないプログラムに

対しては，性能を精度良く予測できない．なぜならば，推測方式は，プログラムの一部の実行時間とソースプログラムの静的な解析から全体の実行時間を推測するためである．一般に，MW プログラムにおけるタスクの計算負荷は，ソースプログラムから静的に解析できないことが多い． なぜならば，MW 方式は，プロセッサ間の予測できない計算負荷の偏りを，動的負荷分散によって均等化するために広く用いられている手法であるためである．したがって，推測方式では MW プログラムの性能を精度良く予測できない．

本章では，MW プログラムに対して速度と精度を両立する性能予測手法を提案する．ここで，高速な予測とは，MW プログラムの実行時間よりも短い時間で予測することを表すものとする．提案手法では，直接実行方式に基づく性能予測において，直接実行するタスクの数を削減することで予測の高速化を図る．また，この削減の代わりに，直接実行した少数のタスクの実行時間からプログラム全体の実行時間を推測する．この推測では，線形補間を用いることで，タスクの割当順にしたがって個々のタスクの実行時間を推測する．これは精度の良い予測を実現するために重要である．

以降では，まず 4.2 節で既存の性能予測手法を述べる．次に， 4.3 節で提案する性能予測手法を述べる．その後，4.5 節で提案手法を予測速度と予測精度の観点から評価し，4.6 節で本章のまとめを述べる．

4.2　関連研究

これまでに，MW プログラムの性能予測に関していくつかの理論的な研究が行なわれている [6, 9, 25]．これらは MW プログラムを高速実行できるワーカ数の決定に有用であるが，全てのタスクの大きさが同じであったり，またはタスクの大きさが特定の分布に従うなど，現実のアプリケーションへの適用に関して厳しい制約を要する．

その他の手法として，並列プログラムの直接実行に基づく直接実行方式とソースプログラムの解析に基づく推測方式がある．MPI-SIM[45] は直接実行方式に基づいており，離散事象シミュレーションによってメッセージ通信仕様 MPI を用いた並列プログラム（MPI プログラム）の実行時間を予測する．この離散事象シミュレーションでは，MPI プログラムを実行しながら計算，送信および受信の 3 種類のイベントを取得し，これらのイベントに基づいて対象環境上での MPI プログラムの動作をシミュレートする．直接実行方式は，並列プログラムの計算および通信といった動作を詳細に再現することを目的としており，対象環境における並列プログラムの動作を詳細に再現できることが利点である．また，並列プログラムの動作を再現によって，並列プログラムの対象環境上での実行時間を精度良く予測できる．しかし，直接実行方式は，並列プログラムを直接実行するので，性能予測に少なくとも並列プログラムの実行時間と同等の時間を要する．本研究では，高速な性能予測を目指しているため，性能予測に多くの時間を要するという直接実行方式の性質は問題である．

一方，推測方式 [2, 51] は，プログラムを全く実行しない，あるいは一部しか実行しないので，直接実行方式よりも高速に性能を予測できる．推測方式では，ソースプログラムの静的な解析や直接実行したプログラムの一部の実行時間からプログラム全体の計算負荷を推測し，全体の

実行時間を算出する．例えば，文献 [2] の手法では，プログラムにおけるループの繰り返し 1 回分の実行時間を直接実行によって測定し，その実行時間とソースプログラムの解析によって得たループの繰り返し回数を乗算することで，ループ全体の実行時間を推測する．推測方式は，制御フローが静的に決まるようなプログラムに対して有用である．なぜならば，そのようなプログラムはソースプログラムから全体の計算負荷を推測しやすいためである．しかし，一般的に MW プログラムは 4.1 節 で述べたようにソースプログラムから計算負荷を予測することは容易でない．

このように，既存の直接実行方式と推測方式は，MW プログラムの高速な性能予測と高精度な性能予測を両立することができない．そこで，本研究では，MW プログラムの高速かつ高精度な性能予測を実現する性能予測手法を提案する．

4.3 マスタ・ワーカ型並列プログラムの性能予測に対する高速化手法

本節では，提案する性能予測手法について述べる．提案手法は以下に示す 2 つの手順から成る．まず，MW プログラムにおけるいくつかのタスクの実行時間を直接実行によって測定し，線形補間を用いてそれらのタスクの実行時間から残りの全タスクの実行時間を推測する．以降では，この直接実行を**部分実行**と呼ぶ．次に，推測したタスクの実行時間を用いて MW 方式の動作をシミュレートする．以降の節では，それぞれの詳細について述べる．

4.3.1 線形補間によるタスクの実行時間の推測

一般に，MW プログラムではタスクの計算負荷および処理の流れは動的に決まる．例えば，タスクの処理におけるループの繰り返し条件が計算結果に依存する場合，ループの繰り返し回数は動的に決まるといえる．また，マスタはタスクの処理が完了したワーカに新しいタスクを割り当てるので，タスクの計算負荷が動的に決まる場合，マスタが割り当てるワーカも動的に決まるといえる．したがって，既存の推測方式のように，ループの繰り返し 1 回の実行時間やソースコードの静的解析では，プログラム全体の実行時間を精度良く推測することは容易ではない．そこで，推測の精度を向上させるためには， MW プログラムのより多くの部分を直接実行し，推測に用いる情報を増やすことが考えられる．しかし，性能予測の高速化の観点からは，MW プログラムにおいて直接実行する部分をできる限り削減する必要がある．

これらの議論から，いくつかのタスクの実行時間を直接実行によって測定し，それらから残りの全タスクの実行時間を線形補間によって推測することを考える．一般に， MW プログラムの各タスクの計算負荷は動的に決まるため，タスクの実行時間を正確に推測することは容易でない．しかし，個々のタスクの実行時間は全く関係がないわけではなく，割当順序が近いタスクは計算負荷が似ることが多い．したがって，線形補間によって良い精度で推測できると考えられる．図 4.1 に，線形補間によるタスクの実行時間の推測例を示す．図 4.1 では，まず，タスク 1, 4, 7 および 10 の実行時間を直接実行によってを計測する．そして，残りの全タスクの実行時間を線形補間によって推測する．この推測によって，提案手法では全てのタスクを直接実行

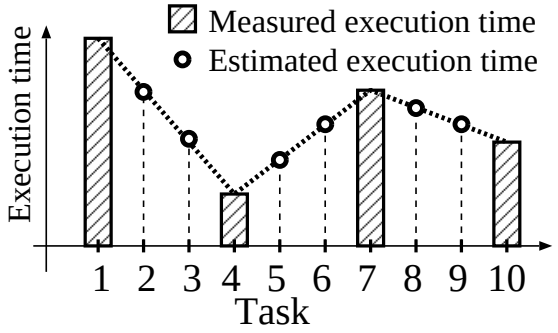


図 4.1: 線形補間によるタスクの実行時間の推測

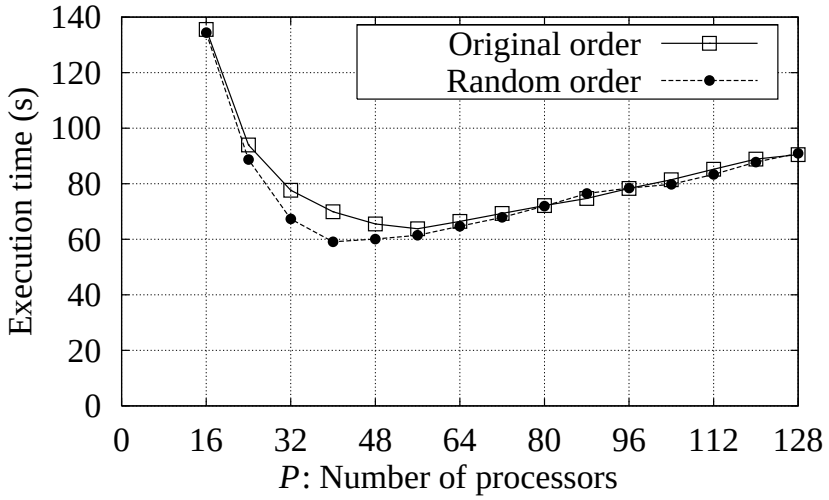


図 4.2: タスクの割当順序の違いによる性能の変化

することを避けている．

この推測において， MW プログラムの性能を精度良く予測するためには，全タスクの総実行時間だけでなく，マスタのタスク割当順序にしたがって個々のタスクの実行時間も正確に推測することが重要である．なぜならば，タスクの割当順序の違いにより，MW プログラムの性能が変化する可能性があるからである． 図 4.2 に，タスクの割当順序の違いによって MW プログラムの性能が変化する例を示す．図 4.2 では， MW プログラムにおいて元々の割当順序で実行したときの実行時間と，ランダムに割当順序を変更したときの実行時間を示している．図 4.2 では，$P = 40$ において，割当順序をランダムとしたときの実行時間が，元々の割当順序で実行したときの実行時間と比べて 15%短い．MW プログラムの性能予測において，この差は高速実行できるワーカ数の検出を妨げることにつながる．

このような差が生じる原因は，割り当てるタスクの順序に違いによって，ワーカの利用率が変化するためである．図 4.3 に MW 方式の動作を示す．ここで，図 4.3(a) はマスタが計算負荷

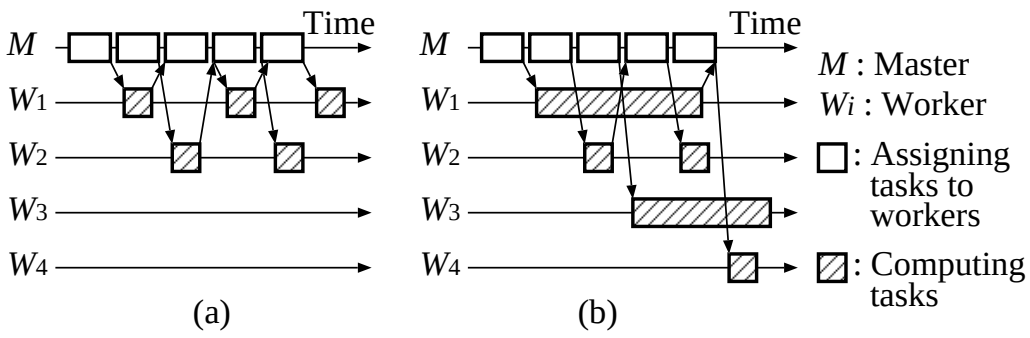


図 4.3: ワーカの利用効率

が小さいタスクのみをワーカに割り当てる場合を表し，図 4.3(b) は計算負荷が大きいタスクと小さいを交互に割り当てる場合を表す． 図 4.3(a) では，2 つのワーカ W_3 と W_4 はアイドル状態となっている．この理由は，タスクの計算負荷が小さく，マスタが W_3 と W_4 にタスクを割り当てるよりも前に W_1 と W_2 に再割当するためである．一方， 図 4.3(b) では，計算負荷の大きいタスクと小さいタスクを交互に割り当てることで，全てのワーカはタスクを処理している．このように，タスクの割当順序はワーカの利用率に影響する．以上より，タスク全体の合計実行時間だけでなく，タスクの割当順序にしたがって個々のタスクの実行時間も精度良く推測することが，MW プログラムの性能を精度良く予測するために重要である．線形補完による推測は，実行時間を測定するタスク数を増やすことで，タスク全体の合計実行時間と個々のタスクの実行時間を精度良く予測できると考える．

4.3.2 シミュレーションによる性能ボトルネックの表現

MW 方式における性能ボトルネックの 1 つとして，ワーカからのメッセージがマスタへ集中することが挙げられる．マスタへのメッセージの集中によってワーカへのタスク割当が遅れ，MW プログラムの性能が低下する．MW プログラムの性能を精度良く予測するためには，この性能ボトルネックによる性能低下を予測できなければならない．これを実現するために，提案手法では MW 方式の動作をシミュレーションによって再現する．以下では，シミュレータの動作について述べる．

シミュレータは，マスタが全てのタスクの処理結果を受信するまで以下の 3 つの動作を繰り返す．

(1) マスタは各ワーカへタスクを割り当てる．

(2) ワーカは，割り当てられたタスクの実行時間後にマスタへ処理結果を返す．

(3) マスタはワーカからの処理結果を受信し，そのワーカに対して未割当の別のタスクを割り当てる．

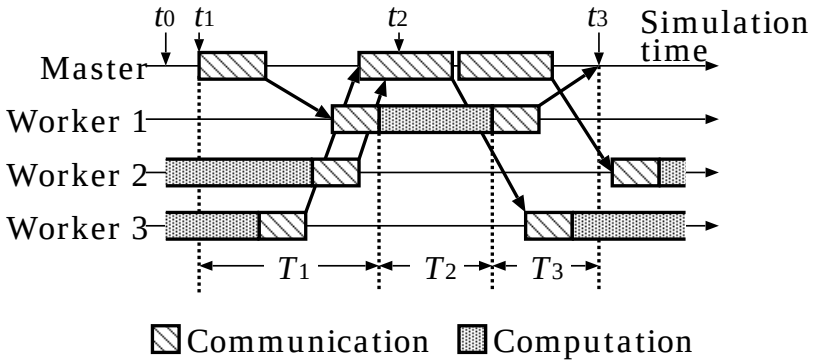


図 4.4: シミュレーションの例

マスタがあるワーカ W_1 へのタスク割当時に他のワーカ W_2 からの処理結果が届いた場合，W_2 へのタスク割り当ては W_1 の割り当てが完了するまで遅らせる．これにより，MW 方式のマスタにおける性能ボトルネックを再現する．シミュレータは，MW プログラムのタスク割当および実行時間を管理するために，シミュレータ内部に仮想時刻とメッセージキューをもつ．仮想時刻は，シミュレーションにおける MW プログラムの開始からの経過時間を表す．メッセージキューは，ワーカからマスタへの処理結果のメッセージが到着する時刻を格納する．このメッセージ到着時刻は，上述の動作（1）においてマスタがワーカへタスクを割り当てる際，そのタスクの実行時間とワーカとの通信時間との合計となる．通信時間は，後述する並列計算モデルによって計算する．シミュレータは，上述の各動作において，仮想時刻を適切に進めることで，MW プログラムの経過時間を管理する．

図 4.4 にシミュレーションの例を示す．この例では，仮想時刻が t_0 のときにキューには 2 つの到着時刻 t_1 および t_2 を格納している．まず，シミュレータはキューから最も早い到着時刻 t_1 を取り出し，仮想時刻を t_1 とする．このとき，シミュレータはタスクの割当に要する時間 T_1，タスクの実行時間 T_2 および結果の通信時間 T_3 を計算し，これらの合計からワーカからマスタへ処理結果が到着する時刻を求める．T_2 は，4.3.1 節で計測もしくは推測したタスクの実行時間である．高速な性能予測を実現するためには通信時間 T_1 および T_3 の見積りを高速に行う必要がある．そこで，3.3.1 節の議論より，並列計算モデルを用いて通信時間を見積もる．本章では，3 章で提案したマスタの通信オーバヘッドを表現するための拡張を適用した LogGP モデル [4] を使用する．

4.4　実行パラメータ値の検出の流れ

本節では，図 4.5 を用いて MW プログラムの実行パラメータ値を検出する流れを示す．まず，性能予測を行いたいユーザは，部分的にタスクを実行してそれらの実行時間を計測するように MW プログラムを修正する．この修正は，マスタがワーカへのタスク割当を抑制する処理と，タスクの処理に対して時間計測のための処理を付加するものであり， 図 1.4 の並列化コンパイ

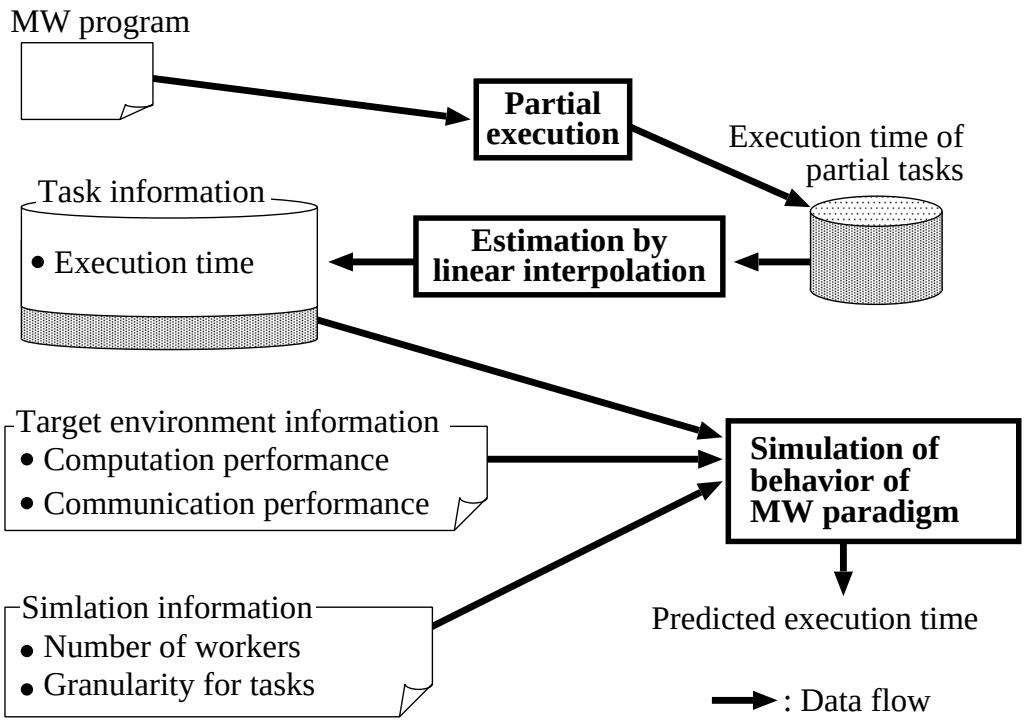


図 4.5: 性能予測の流れ

ラに容易に組込み可能と考えられる．次に，修正したプログラムを実行することで部分的なタスクの実行時間を得る．この実行時間から，線形補完によって残りの全タスクの実行時間を推測する．そして，MW 方式の動作をシミュレーションする．シミュレータには，タスク情報，対象環境情報およびシミュレーション情報の 3 つを入力として与える．タスク情報は，推測した全タスクの実行時間である．対象環境情報は，性能予測の対象とする計算環境（対象環境）の演算性能および通信性能を表す．ここで，演算性能は MW プログラムの部分実行に用いた計算機との演算速度比によって表す．また，通信性能は，LogGP モデルのパラメータによって表す．シミュレーション情報は，性能予測によって検査する実行パラメータ値を格納する．これらの入力をシミュレータに与えることで，対象環境において各実行パラメータ値を用いて実行したときの MW プログラムの予測実行時間を得る．ユーザは得られた予測実行時間を基に，実行パラメータ値を選択する．

4.5 評価実験

本節では，予測速度と予測精度の観点から提案手法を評価する．評価実験で用いる MW プログラムの問題として，マンデルブロ集合探索問題を用いた．この問題では各タスクの計算負荷は実行時の計算結果によって決まるため，従来の推測方式のようにソースプログラムの静的な解析に基づく性能予測手法では，精度良く予測することはできない．タスクは，複素平面上の

1 点に対するマンデルブロ集合への包含判定であり，タスクの総数は 1,048,576 である．

性能予測の対象環境として，64 台の PC からなる PC クラスタを用いた．各 PC はミリネット高速ネットワーク [11] およびイーサネットで接続されており，それぞれの通信バンド幅は 2Gb/s および 100Mb/s である．各 PC は CPU として Pentium III 1GHz を 2 台もち， PC クラスタ全体として 128 台の CPU をもつ．シミュレータは PC クラスタを構成する PC1 台を用いた． MW プログラムは，MPI を用いたメッセージ通信プログラムとして実装した．MPI の実装として，イーサネット上では MPICH[26] を用い，ミリネット上では MPICH-SCore[42] を用いた．

本実験では，部分実行するタスクの数（部分実行タスク数）の違いによる予測速度と予測精度への影響を調べるために，線形補完によって推測した 2 つのタスク集合 $S_{1\mathrm{K}}$ および S_{16} を用いた．$S_{1\mathrm{K}}$ および S_{16} は，実験に用いた MW プログラムと同じ 1,048,576 個のタスクからなり，それぞれ MW プログラムの 1,024 個および 16 個のタスクの実行時間から推測した．また，線形補間による予測精度の向上の効果を確認するために，タスク集合 S_{R} を用いた．S_{R} は，MW プログラムのタスクの実行時間と同じ確率分布の元でランダムに生成した 1,048,576 個のタスクからなる集合である．

4.5.1　予測速度

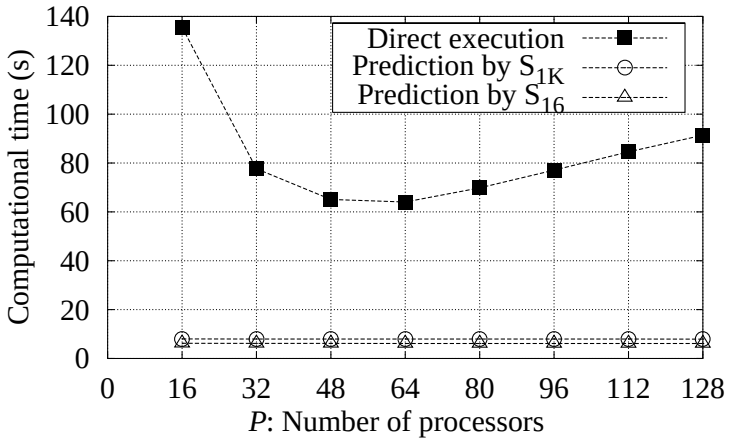
本節では，提案手法の予測速度について評価する．本実験では，予測に要した時間（予測時間）の比較対象として，MW プログラムの直接実行時間を用いた．この理由は，直接実行時間は，直接実行方式に基づく性能予測における予測時間の最小値と考えられるためである．予測時間は, (1) 部分実行, (2) 線形補間, (3) タスク情報のファイルへの入出力，および (4) MW 方式の動作シミュレーションの 4 点に要する時間の合計である．

図 4.6 に予測時間と直接実行時間を示す． 図 4.6 が示すように，予測時間は直接実行時間よりも短い．この実験において直接実行時間と予測時間の比率[1]の最小値は，イーサネット上の $P = 64$ において 8.0 であり，ミリネット上の $P = 128$ において 1.7 である．本研究では，並列プログラムの実行時間よりも短い時間で性能を予測することを目標としている．したがって，この結果より，提案手法による性能予測は高速であるといえる．

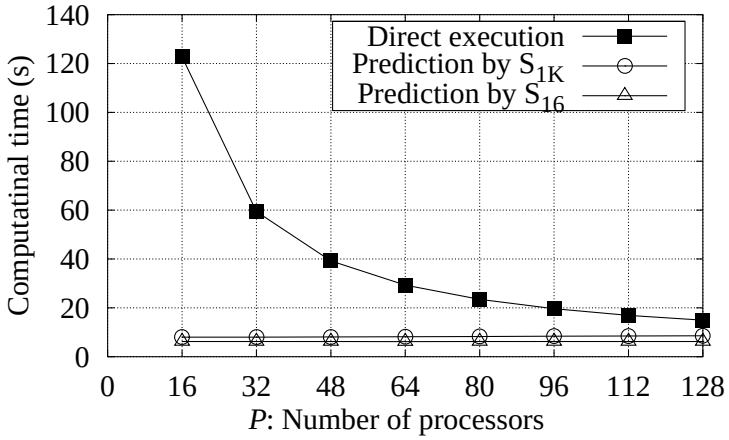
表 4.1 に予測時間の内訳を示す． 図 4.6 ではイーサネットおよびミリネットにおいて，直接実行時間の最小値はそれぞれ 63 秒および 17 秒であるのに対し，表 4.1 では部分的なタスクの直接実行に要する時間を 1.7 秒および 0.2 秒まで削減できている．この効果により，提案手法は直接実行時間よりも短い時間で性能予測が実現できている．表 4.1 では，線形補間を用いた推測に要する時間は小さく，線形補完のオーバヘッドは予測速度の低下にほとんど影響がないことがわかる．一方，ファイル入出力に要する時間が全体に占める割合は大きい．本研究では，シミュレータの実装を簡便化するためにファイルを介してタスク情報をシミュレータに与えたが，図 4.5 においてタスク情報をファイルへ出力せず，シミュレータに直接渡すことで省略できると考えられる．

部分実行に要する時間は，部分実行するタスクの数に依存する．部分実行するタスクの数を

[1] この比率は，$\frac{\text{直接実行時間}}{\text{予測時間}}$ で算出し，直接実行と比べて予測の方が何倍速いかを表す．



(a) イーサネット



(b) ミリネット

図 4.6: プログラムの予測時間と直接実行時間

表 4.1: 予測時間の内訳（単位：秒）

	P	部分実行	線形補間	ファイル入出力	シミュレーション		合計	
					イーサネット	ミリネット	イーサネット	ミリネット
S_{1K}	64	1.7	0.3	5.4	0.6	0.8	8.0	8.2
	128				0.6	1.2	8.0	8.6
S_{16}	64	0.02	0.2	5.4	0.6	0.6	6.2	6.2
	128				0.6	0.6	6.2	6.2

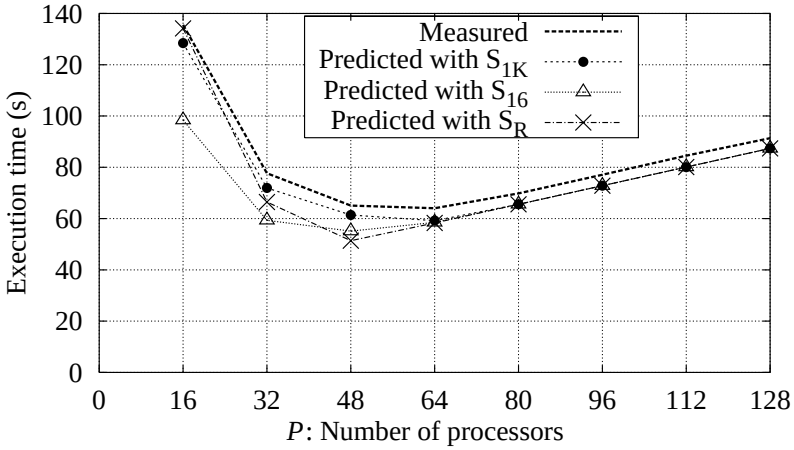
増やすことで予測精度の向上が期待できるが，部分実行に要する時間も増加する．したがって，予測時間を直接実行時間よりも短くするためには，部分実行するタスクの数を注意深く設定しなければならない．タスクの総数を N_a，使用可能な最大プロセッサ数を P とすると，予測時間を直接実行時間より短くするためには，部分実行するタスクの数は $N_a/(P-1)$ 以下である必要があると考える．ならぜならば，MW プログラムの実行において，マスタを除いた $P-1$ 台の各ワーカは平均的に $N_a/(P-1)$ 個のタスクを処理すると考えられるためである．本研究では部分実行は 1 つのプロセッサで行うので，部分実行するタスク数が $N_a/(P-1)$ 個以下であれば，部分実行時間が MW プログラムの実行時間よりも短くなると期待できる．本実験の場合，$N_a = 1{,}048{,}576$ および $P = 128$ である．したがって，部分実行するタスク数を 8,257 以下にすることで，部分実行に要する時間を直接実行時間よりも短くでき，高速な性能予測を実現できると期待できる．

4.5.2　予測精度

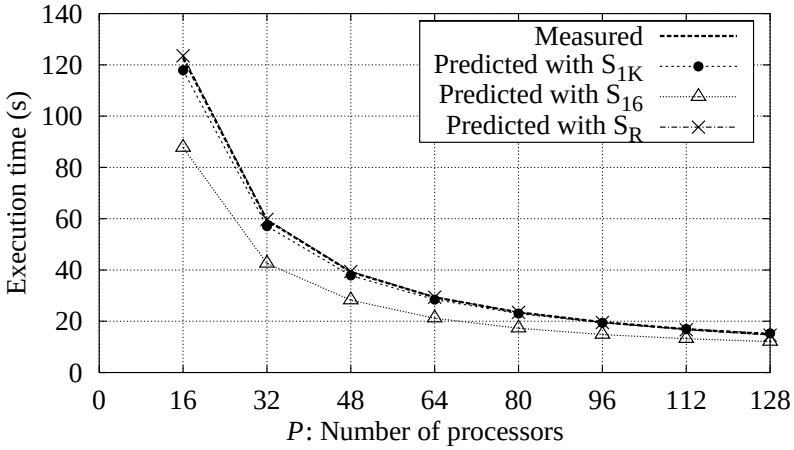
タスク集合 S_x を用いて予測した実行時間を E_x と表記する．図 4.7 に実測と予測の実行時間を示す．実測の実行時間と $E_{1\mathrm{K}}$ との最大誤差率は，イーサネット上の $P = 64$ において 7%，ミリネット上の $P = 32$ において 4%であった．この精度は，3.5.1 節で示した手法の予測誤差 3%に近く，提案手法は良い精度で予測できているといえる．

図 4.7(a) では，E_{R} は MW プログラムの性能飽和点を予測できていないが，$E_{1\mathrm{K}}$ は MW プログラムの性能飽和点を精度良く予測できている．これは，線形補間によって各タスクの実行時間を推測した効果である．4.3.1 節で述べたように，タスクの割当順序をランダムとした場合，ワーカが効率良く利用されることになり，MW プログラムの性能は向上する．この性能向上は，MW プログラムの精度の良い予測を阻害することになる．一方，図 4.7(b) では $E_{1\mathrm{K}}$ と E_{R} はともに同程度に良い精度で実行時間を予測している．これは，図 4.7(b) の $P \leq 128$ ではワーカが効率良く利用されており，S_{R} の使用によるワーカの利用効率向上の効果がほとんど無かったためである．図 4.7(b) の $P \leq 128$ でワーカが効率良く利用される理由は，ミリネットによる通信が高速であるため，マスタのタスク割当がイーサネット使用時と比べてより多くのワーカに迅速に行えるためである．以上より，S_{R} を使用することよる予測精度の低下は，ワーカの利用効率が低下する P の周辺，すなわち，MW プログラムの性能飽和点の付近で顕著になるといえる．したがって，MW プログラムの高速実行できるワーカ数を探索する場合，タスクの割当順序にしたがって個々のタスクの実行時間を正確に再現することが重要となる．

図 4.7 では，実測の実行時間と E_{16} の差はイーサネットおよびミリネットの両方において大きい．この差は，全タスクの総実行時間の差によるものである．実測実行時間と E_{16} との誤差は，イーサネットおよびミリネットにおいてそれぞれ 28%および 27%である．一方，実測の全タスクの総実行時間と S_{16} の総実行時間の誤差は 29% であり，上述の予測誤差 28%および 27%と近い値である．したがって，高い予測精度を実現するためには，推測したタスク集合の総実行時間を，実測の全タスクの総実行時間に近づける必要がある．そのための部分実行タスク数の選択指針を，4.5.3 節 で議論する．



(a) Ethernet



(b) Myrinet

図 4.7: マンデルブロ集合探索問題における実測実行時間と予測実行時間

表 4.2: ブートストラップ法による予測誤差の範囲の推測 ($N = 40$).

n: サンプルしたタスク数	$\mathcal{I}_e$: 予測誤差の範囲 (%)		R: 実測タスクと予測タスクの総実行時間の誤差 (%)
	最小	最大	
4	-70	81	-72
16	-41	46	-29
64	-19	17	-17
256	-10	11	-11
1K	-4.9	5.8	-4.2
4K	-2.9	2.6	-2.5
16K	-1.4	1.1	-1.7
64K	-0.7	0.4	-0.2

4.5.3　高精度な性能予測を実現できる部分実行タスク数の見積り

4.5.2 節で述べたように，部分実行タスク数は予測精度に大きく影響する．しかし，一般に MW プログラムの実行前には各タスクの実行時間の分布に関する情報は不明であるので，適切な部分実行タスク数を MW プログラムから解析的に求めることは容易でない．そこで，ブートストラップ法 [19] と呼ばれる統計的手法に基づいて，適切な部分実行タスク数を見積もる手法について述べる．

まず，ブートストラップ法によって全タスクの平均実行時間を見積もる（ブートストラップ法の詳細については後述する）．本研究で対象とする MW プログラムではタスクの総数は固定されているので，タスクの平均実行時間から MW プログラム全体の実行時間を計算できる．したがって，タスクの平均実行時間の誤差範囲から，MW プログラムの実行時間の誤差範囲を見積もることができる．以下にブートストラップ法の詳細を示す．

1. n 個のタスクをランダムに選び，それらの平均実行時間を測定する．
2. 1. の動作を N 回繰り返し，N 個の平均実行時間を得る．
3. 2. で得た N の実行時間を整列し，それらの両端 α%の要素を除外する．このとき，残った平均実行時間から成る区間を，$100 - 2\alpha$%信頼区間という．

ここで，3. の操作で残った平均実行時間の中央値を，全タスクの平均実行時間とみなす．さらに，その平均実行時間と信頼区間の最小値および最大値との誤差から成る区間を，予測誤差の誤差範囲 $\mathcal{I}_e$ とする． 中心極限定理より，n の増加に伴って信頼区間の幅は狭くなる．すなわち，$\mathcal{I}_e$ も狭くなる．したがって，n を序々に増やしながら $\mathcal{I}_e$ の幅を調べることで，予測誤差の範囲を見積もることができ，適切な部分実行タスク数を見積もることできる．表 4.2 に，本実験で用いた MW プログラムに対してブートストラップ法を適用した結果を示す．ここで，$N = 40$, $\alpha = 2.5\%$ としている． R は MW プログラムの実測総実行時間に対する予測総実行時間の比率

である．表 4.2 が示すように，n の増加に伴って $\mathcal{I}_e$ の幅は狭くなっている．表 4.2 中の全ての n において，$\mathcal{I}_e$ は R を含んでいるか，または R に近い値である．したがって，タスクの総実行時間の誤差を x%以下にしたい場合は，$\mathcal{I}_e$ が区間 $[-x, x]$ に含まれるような n を選択すればよい．例えば，総実行時間の誤差を 10%以下にしたい場合，表 4.2 より，部分実行タスク数を 1K 以上にすればよい．

4.6 おわりに

本章では，MW プログラムの性能予測の高速化手法を提案した．提案手法は，予測を高速に行うために，一部のタスクの実行時間から他のタスクの実行時間を推測することで直接実行するタスク数を削減する．また，この推測において，実際のタスクの総実行時間だけでなく，割当順序にしたがって個々のタスクの実行時間を正確に見積もることが予測精度を向上するために重要であることを示した．また，実験により提案手法の予測速度と予測精度について評価した．その結果，提案手法では，MW プログラムの直接実行と比べて 1.7 倍高速に予測でき，予測誤差は 7%以内であった．この結果より，提案手法は MW プログラムの高速かつ高精度な性能予測に対して有用であることがわかった．

今後の課題として，提案手法を並列化コンパイラに組み込むことで，本章では手動で生成した部分実行用の MW プログラムを，自動的に生成させることが挙げられる．また，4.5.3 節で触れた統計的手法に基づく部分実行タスク数の見積り手法を，並列化コンパイラに組み込むことが挙げられる．

第5章

結論

本論文では，分散メモリ環境における並列再帰プログラムの開発および MW 型並列プログラムを高速実行できる実行パラメータ値の検出に対する支援に取り組んだ．以下にそれぞれの成果をまとめる．

R1. **並列化方式を指定できるコンパイラによる，並列再帰プログラムの開発**

多くの場合，どの並列化方式が並列再帰プログラムの実行時間を短くできるかを開発者自身が予想できる点に着目し，再帰処理の並列化方式および並列化条件を指定できる並列化コンパイラを提案した．また，再帰アルゴリズムに適した並列化方式および並列化条件の指定方針を示した．負荷分布の特徴が異なるいくつかの再帰アルゴリズムに対して，提案手法を用いて並列プログラムを生成した結果，並列化方式および並列化条件の指定方針に従うことで，指定方針に従わない場合と比べてそれぞれ最大 25%および 77% の性能差を確認できた．これにより，並列化方式および並列化条件をプログラマが指定できることの重要性を確認できた．

R2. **MW プログラムの性能予測による，高速実行できる実行パラメータ値の検出**

MW プログラムの性能を高精度に予測するための考慮点として,（D1）並列計算モデルを用いた低オーバヘッドの性能予測,（D2）並列計算モデルの拡張によるマスタの通信オーバヘッドのモデル化,（D3）実行時に決まる挙動の再現の 3 点を示し，実験によってそれぞれの重要性を示した．実験結果は，64 台の CPU をもつ PC クラスタにおける MW プログラムの実行時間を予測したところ，従来の並列計算モデルでは 38%以上の予測誤差が生じるに対し，提案手法では 10%以内の予測誤差で予測できた．また，MW プログラムのスケーラビリティ解析においてマスタの通信オーバヘッドをプロセッサ数の線形関数とすることの重要性を示した．複雑な動作する MW プログラムへの適用実験により，D3 の重要性を示した．

さらに，MW プログラムの性能予測を高速に行うための手法を提案した．提案手法では，MW プログラムの直接実行を一部分のタスクのみに削減することで，性能予測を高速化する．この一部分のタスクを直接実行によって計測し，その実行時間から残りの全タスクの実行時間を推測する．このとき，タスクの総実行時間のみでなく，タスクの割当順にしたがって個々のタスクの実行時間も正確に予測することが高精度に性能予測するために重要であることを実験によって示した．実験の結果，提案手法は MW プログラム全体の直

接実行に基づく従来手法と比べ，同程度の予測精度を保ちつつ，1.7 倍以上高速に予測できた．

本研究の成果は，アルゴリズムを構築する上で重要な技法である再帰を容易に並列化でき，開発者に対して並列プログラムの設計の幅を広げた点で有用といえる．また，多様化が進む近年の様々な分散メモリ環境において， MW 型並列プログラムを高速に実行できる実行パラメータ値の高速かつ高精度な検出が可能となった点で有用といえる．

近年，クラスタ計算やグリッド計算に関する技術の発展により，並列計算はますます身近なものとなりつつある．しかしながら，これらの技術の発展により，並列計算環境の構造は複雑になり，性能の良い並列再帰プログラムの開発と実行が煩雑となってきている．また，並列計算環境の多様化によって，実行環境毎に高速実行できる実行パラメータ値を検出する必要が出てきている．特に，デスクトップグリッド環境のように計算ノードの性能が動的に変化する並列計算環境が普及してきているため，並列プログラムの実行開始時に最適な実行パラメータを高速に検出する技術は，並列プログラムの高速な実行を実現する上で重要といえる．本研究の成果は，このような現状を打破するために有用であるものと考える．

5.1 今後の課題

今後の課題として, (1) 本論文 3 章および 4 章の示した性能予測手法の並列化コンパイラへの組み込み, (2) 計算ノードの演算性能および通信性能が不均一な並列計算環境を対象とした提案手法の評価，および (3) デスクトップグリッド環境において実行パラメータを動的に切り替える手法の考案が挙げられる．近年注目されているグリッド環境は，ネットワークで接続された PC の遊休資源を利用することで，高性能計算を行うものである．各 PC の遊休状態は時々刻々と変化するため，提案手法によってプログラム実行開始時に発見した実行パラメータが，実行中に不適切なものとなる可能性がある．そこで，動的に変化する計算ノードの性能を監視しながら，その時々に応じて適切な実行パラメータに変更する機構を開発することで，常に効率的な実行を実現することができると考える．

謝辞

本研究の全過程において日頃からご指導，ご鞭撻を賜わりました大阪大学・萩原兼一教授に心より深謝いたします．本論文の執筆にあたり，多数の有益なご助言をいただきました大阪大学・増澤利光教授，同大学・松田秀雄教授，同大学・谷口健一教授に心より深謝いたします．

本研究を進めるにあたり，日頃から多くのご助言と励ましをいただいた大阪大学・藤本典幸助教授に深く感謝いたします．また，本研究を進めるにあたり，日頃から適切なご指導を賜わりました大阪大学・伊野文彦助手に深く感謝いたします．

また，研究活動を始めた当初より，研究を進める上での基本的な作業に関してご指導いただいたマイクロソフト・中島大輔氏に感謝いたします．本研究を進めるにあたって有益なご助言をいただき，また，研究を遂行するための計算機環境の維持にご尽力くださいました大阪大学・川崎康博氏に感謝いたします．本研究を進めるにあたり，研究環境の維持にお力添えいただいた大阪大学・置田真生氏，同大学・松井学氏，松下電器産業株式会社・大山寛郎氏，同社・竹内彰氏，株式会社東芝・神戸友樹氏に感謝いたします．本研究の実験を行うために並列計算機 NEC Cenju-3 を使用させていただきました日本電気株式会社に感謝いたします．

本研究を進めるにあたり，日頃より様々な御支援をいただきました，萩原研究室事務員・近藤充世さんに感謝いたします．また，日頃から本研究の細部に渡って多くの有益なご助言をいただきました萩原研究室の皆様に感謝いたします．

最後に，8 年にも及ぶ学生生活を支えてくださり，終始温かく見守ってくださいました故父，母，兄および義姉に心より感謝いたします．

参考文献

[1] Adve, V. S., Bagrodia, R., Browne, J. C., Deelman, E., Dube, A., Houstis, E. N., Rice, J. R., Sakellariou, R., Sundaram-Stukel, D. J., Teller, P. J. and Vernon, M. K.: POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems, *IEEE Trans. Software Engineering*, Vol. 26, No. 11, pp. 1027–1048 (2000).

[2] Adve, V. S., Bagrodia, R., Deelman, E. and Sakellariou, R.: Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures, *J. Parallel and Distributed Computing*, Vol. 62, No. 3, pp. 393–426 (2002).

[3] Aida, K. and Natsume, W.: Distributed Computing with Hierarchical Mater-worker Paradigm for Parallel Branch and Bound Algorighm, *Proc. 3rd IEEE/ACM Int'l Symp. Cluster Computing and the Grid (CCGrid'03)*, pp. 156–163 (2003).

[4] Alexandrov, A., Ionescu, M. F., Schauser, K. E. and Scheiman, C.: LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation, *J. Parallel and Distributed Computing*, Vol. 44, No. 1, pp. 71–79 (1997).

[5] Almasi, G. S. and Gottlieb, A.: *Highly parallel computing*, Benjamin-Cummings Publishing Company, Inc. (1994).

[6] Anglano, C.: Predicting Parallel Applications Performance on Non-dedicated Cluster Platforms, *Proc. 12th ACM Int'l Conf. Supercomputing (ICS'98)*, pp. 172–179 (1998).

[7] Babbar, M. and Minsker, B. S.: A Multiscale Master-Slave Parallel Genetic Algorithm with Application to Groundwater Remediation Design, *Proc. Genetic and Evolutionary Computation Conference (GECCO'2002)*, pp. 9–16 (2002).

[8] Basney, J., Raman, R. and Livny, M.: High Throughput Monte Carlo, *Proc. 9th SIAM Conference on Parallel Processing for Scientific Computing* (1999).

[9] Beaumont, O., Legrand, A. and Robert, Y.: The Master-Slave Paradigm with Heterogeneous Processors, *Proc. 3rd IEEE Int'l Conf. Cluster Computing (CLUSTER'01)*, pp. 419–426 (2001).

[10] Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J. and Zagha, M.: Implementation of a Portable Nested Data-Parallel Language, *J. Parallel and Distributed Computing*, Vol. 21, No. 1, pp. 4–14 (1994).

[11] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. and Su, W.-K.: Myrinet: A Gigabit-per-Second Local-Area Network, *IEEE Micro*, Vol. 15, No. 1, pp. 29–36 (1995).

[12] Burns, G., Daoud, R. and Vaigl, J.: LAM: An Open Cluster Environment for MPI, *Proc. Supercomputing Symp. (SS'94)*, `http://www.lam-mpi.org/`, pp. 379–386 (1994).

[13] Buyya, R.(ed.): *High Performance Cluster Computing*, Prentice Hall PTR, Englewood Cliffs, NJ (1999).

[14] Chien, A., Calder, B., Elbert, S. and Bhatia, K.: Entropia: architecture and performance of an enterprise desktop grid system, *J. Parallel and Distributed Computing*, Vol. 63, No. 5, pp. 597–610 (2003).

[15] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R. and von Eicken, T.: LogP: Towards a Realistic Model of Parallel Computation, *Proc. 4th ACM SIGPLAN Symp. Principles Practice of Parallel Programming (PPoPP'93)*, pp. 1–12 (1993).

[16] Czarnul, P., Tomko, K. and Krawczyk, H.: Dynamic Partitioning of the Divide-and-Conquer Scheme with Migration in PVM Environment, *Proc. 8th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'01)*, pp. 174–182 (2001).

[17] 中島大輔, 藤本典幸, 萩原兼一: 分割統治法アルゴリズムの効率的な並列化手法とそのコンパイラの実装, 情報処理学会研究会報告 2000-AL-71, pp. 25–32 (2000).

[18] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L. and White, A.(eds.): *Source Book of Parallel Computing*, Morgan Kaufmann, Aan Mateo, CA (2002).

[19] Efron, B.: Bootstrap Methods: Another Look at the Jackknife, *Annals of Statistics*, Vol. 7, No. 1, pp. 1–26 (1979).

[20] Eide, V.: Parallel Recursive Procedures. A manager/worker approach (1994). `http://www.ifi.uio.no/~arnem/PRP`.

[21] Foster, I. and Kesselman, C.(eds.): *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Mateo, CA (1998).

[22] Frank, M. I., Agarwal, A. and Vernon, M. K.: LoPC: Modeling Contention in Parallel Algorithms, *Proc. 6th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'97)*, pp. 276–287 (1997).

[23] Freisleben, B. and Kielmann, T.: Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs, *Computers and Artificial Intelligence*, Vol. 14, No. 6, pp. 579–596 (1995).

[24] Grama, A., Gupta, A., Karypis, G. and Kumar, V.: *Introduction to Parallel Computing*, Reading, MA: Addison-Wesley, 2nd edition (2003).

[25] Greenberg, A. G. and Wright, P. E.: Design and Analysis of Master/Slave Multiprocessors, *IEEE Trans. Computers*, Vol. 40, No. 8, pp. 963–976 (1991).

[26] Gropp, W., Lusk, E., Doss, N. and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol. 22, No. 6, pp. 789–828 (1996).

[27] Hardwick, J. C.: An Efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms, *Proc. 1st Int'l Workshop on High-Level Programming Models and Supportive Environments (HIPS1996)*, pp. 105–114 (1996).

[28] Hatcher, P. J., Quinn, M. J., Lapadula, A. J., Seevers, B. K., Anderson, R. J. and Jones, R. R.: Data-Parallel Programming on MIMD Computers, *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 377–383 (1991).

[29] Ino, F., Fujimoto, N. and Hagihara, K.: LogGPS: A Parallel Computational Model for Synchronization Analysis, *Proc. 8th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'01)*, pp. 133–142 (2001).

[30] JáJá, J.: *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, Inc. (1992).

[31] Kawasaki, Y., Ino, F., Mizutani, Y., Fujimoto, N., Sasama, T., Sato, Y., Sugano, N., Tamura, S. and Hagihara, K.: High-Performance Computing Service Over the Internet for Intraoperative Image Processing, *IEEE Trans. Information and Technology in Biomedicine*, Vol. 8, No. 1, pp. 36–46 (2004).

[32] Kawasaki, Y., Ino, F., Mizutani, Y., Fujimoto, N., Sasama, T., Sato, Y., Tamura, S. and Hagihara, K.: A High Performance Computing System for Medical Imaging in the Remote Operating Room, *Proc. 10th Int'l Conf. High Performance Computing (HiPC 2003)* (2003).

[33] 石畑清: アルゴリズムとデータ構造，岩波講座ソフトウェア科学 Vol.3, 岩波書店 (1989).

[34] Koelbel, C., Loveman, D., Schreiber, R., G. Steele, J. and Zosel, M.: *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA (1994).

[35] 小河原徹, 中島大輔, 藤本典幸, 萩原兼一: 分割統治法プログラムを並列実行するコンパイル手法の提案と評価, 情報処理学会研究会報告 1999-AL-66, pp. 73–80 (1999).

[36] Kvasnicka, D. F., Hlavacs, H. and Ueberhuber, C. W.: Simulating Parallel Program Performance with CLUE, *Proc. 2001 Int'l Symp. Performance Evaluation of Computer and Telecommuication Systems (SPECTS'01)*, pp. 140–149 (2001).

[37] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, *Int'l J. Supercomputer Applications and High Performance Computing*, Vol. 8, No. 3/4, pp. 159–416 (1994).

[38] Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H. D.: 24th Edition of TOP500 List of World's Fastest Supercomputers (2004).

[39] Mizutani, Y., Ino, F. and Hagihara, K.: Evaluation of Performance Prediction Method for Master/Slave Parallel Programs, *IEICE Trans. Information and Systems*, Vol. E87-D, No. 4 (2004).

[40] Moritz, C. A. and Frank, M. I.: LoGPC: Modeling Network Contention in Message-Passing Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol. 12, No. 4, pp. 404–415 (2001).

[41] 藤本典幸, 乾和弘, 前田昌也, 柘殖宗俊, 萩原兼一: ワーク・タイムモデルに基づく並列プログラミング言語 Work-Time C の提案と EWS 用コンパイラの実装, 日本ソフトウェア科学会第 13 回大会論文集, pp. 205–208 (1996).

[42] O'Carroll, F., Tezuka, H., Hori, A. and Ishikawa, Y.: The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network, *Proc. 12th ACM Int'l Conf. Supercomputing (ICS'98)*, `http://www.pccluster.org/`, pp. 243–250 (1998).

[43] 大山寛郎, 水谷泰治, 藤本典幸, 萩原兼一: プロセッサグループの動的分割による並列再帰プログラムの実現手法, 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG1（PRO13）, pp. 107–117 (2001).

[44] Pelagatti, S.: Compiling and supporting skeletons on MPP, *Proc. 3rd Working Conference on Massively Parallel Programming Models*, pp. 140–150 (1998).

[45] Prakash, S., Deelman, E. and Bagrodia, R.: Asynchronous Parallel Simulation of Parallel Programs, *IEEE Trans. Software Engineering*, Vol. 26, No. 5, pp. 385–400 (2000).

[46] Rugina, R. and Schauser, K. E.: Predicting the Running Times of Parallel Programs by Simulation, *Proc. 12th Int'l Parallel Processing Symp. (IPPS'98)* (1998).

[47] Sittig, D. F., Foulser, D., Carriero, N., McCorkle., G. and Miller, P. L.: A parallel computing approach to genetic sequence comparison: The master-worker paradigm with interworker communication, *Computers and Biomedical Research*, Vol. 24, No. 2, pp. 152–169 (1991).

[48] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. and Dongarra, J.: *MPI: The Complete Reference*, MIT Press (1996).

[49] Song, H. J., Liu, X., Jakobsen, D., Bhagwan, R., Zhang, X., Taura, K. and Chien, A.: The MicroGrid: a Scientific Tool for Modeling Computational Grids, *Proc. High Performance Networking and Computing Conf. (SC2000)* (2000).

[50] Sterling, T., Savarese, D., Becker, D. J., Dorband, J. E., Ranawake, U. A. and Packer, C. V.: BEOWULF: A Parallel Workstation for Scientific Computation, *Proc. 24th International Conference on Parallel Processing (ICPP'95)*, pp. 11–14 (1995).

[51] van Gemund, A. J. C.: Symbolic Performance Modeling of Parallel Systems, *IEEE Trans. Parallel and Distributed Systems*, Vol. 14, No. 2, pp. 154–165 (2003).

[52] van Nieuwpoort, R. V., Kielmann, T. and Bal, H. E.: Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications, *Proc. 8th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'01)*, pp. 34–43 (2001).

[53] Weber, D., Specialetti, M. and Barada, H.: Vidnet: Distributed processing environment for computer generated animation, *Software - Practice and Experience*, Vol. 26, No. 2, pp. 237–250 (1996).

[54] Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, Inc. (1996).

[55] 山本洋, 中田秀基, 下平英寿, 松岡聡: グリッド技術を用いた進化系統樹推定の並列化, 情報処理学会研究報告 2004-HPC-97, pp. 181–186 (2004).

[56] 岸部祥典, 小河原徹, 藤本典幸, 萩原兼一: SPMD プログラムを生成する Work-Time C 処理系の実現, 情報処理学会研究会報告 98-AL-60, pp. 57–64 (1998).

関連発表論文

学会誌掲載論文

1. Yasuharu Mizutani, Fumihiko Ino, and Kenichi Hagihara, "*Evaluation of Performance Prediction Method for Master/Slave Parallel Programs*", IEICE Transactions on Information and Systems, Vol. E87-D, No. 4, pp. 967–975, (2004-04)

2. 大山寛郎, 水谷泰治, 藤本典幸, 萩原兼一, "**プロセッサグループの動的分割による並列再帰プログラムの実現手法**", 情報処理学会論文誌: プログラミング, Vol. 43, No. SIG1(PRO13), pp. 107–117, (2002-01)

3. 水谷泰治, 中島大輔, 藤本典幸, 萩原兼一, "**並列再帰の実行方式をプログラマが選択可能なコンパイラの評価**", 電子情報通信学会論文誌, Vol. J84-D-I, No. 6, pp. 594–604, (2001-06)

国際会議発表論文

1. Yasuharu Mizutani, Fumihiko Ino, and Kenichi Hagihara, "*Fast Performance Prediction of Master-Slave Programs by Partial Task Execution*", In 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems (SEPADS 2005), Salzburg, Austria, (2005-02, 採録決定)

2. Yasuharu Mizutani, Fumihiko Ino, Yuki Kanbe, Masao Okita, and Kenichi Hagihara, "*Developing Performance Prediction Tools for Cluster Computing: Research Activities at Hokuriku IT Open Laboratory*", In Proceedings of the 1st International Symposium on Towards Peta-Bit Ultra-Networks (PBit 2003), pp. 169–175, Ishikawa, Japan, (2003-09)

3. Yasuharu Mizutani, Fumihiko Ino, and Kenichi Hagihara, "*An Emulation System for Predicting Master/Slave Program Performance*", In Proceedings of the 9th European Conference on Parallel Computing (Euro-Par 2003), Lecture Notes in Computer Science 2790, Springer-Verlag, pp. 135–140, Klagenfurt, Austria, (2003-08)

口頭発表論文

1. 水谷泰治, 伊野文彦, 萩原兼一, “**部分的なタスク実行によるマスタスレーブ型並列プログラムの高速な性能予測**”, 情報処理学会研究報告, 2004-HPC-98, pp. 13–18, (2004-04)

2. 水谷泰治, 藤本典幸, 萩原兼一, “**スケーラビリティを考慮した並列再帰の実行方式の提案**”, 情報処理学会研究報告, 2001-AL-77, pp. 49–56, (2001-03)