分散メモリ型並列処理環境における
効率的な並列性制御・負荷分散に関する研究

佐藤裕幸

システム情報工学研究科
筑波大学

2003年6月

目次

第1章　はじめに **1**

1.1　本研究の背景 1

1.2　本研究の目的 2

1.3　対象とするアーキテクチャと並列分割形態 3

1.4　本論文の構成 5

第2章　大規模並列処理環境における OS での効率的な並列性制御方式 **7**

2.1　緒言 7

2.2　プロセス 8

2.3　言語レベルで提供される並列実行管理・制御機能 11

2.4　資源木による資源管理・実行制御 13

2.4.1　タスク 14

2.4.2　資源木 14

2.4.3　OS 資源の管理方式 17

2.4.4　従来 OS との比較 18

2.5　システムの保護機構 19

2.5.1　問題点 19

2.5.2　保護フィルタ 20

2.6　プロセス・オーバヘッドの評価 23

2.7　結言 26

第3章　大規模並列処理環境向け負荷分散方式 **27**

3.1　緒言 27

3.2 従来の方式 29
3.2.1 MLB 方式と STB 方式 29
3.2.2 LLS 方式 30
3.3 世代別動的負荷分散方式 (LLS-G 方式) 31
3.3.1 基準情報の報告と均等化の間隔 31
3.3.2 基準情報 33
3.3.3 分配量の決定 34
3.4 負荷分散オーバヘッドの評価 36
3.4.1 評価項目 36
3.4.2 計測 38
3.4.3 考察 40
3.5 結言 44

第 4 章 効率的な並列性制御を行なうための言語サポート 45
4.1 緒言 45
4.2 設計方針 46
4.3 言語仕様 48
4.3.1 リモート・オブジェクト 48
4.3.2 同期機構 49
4.3.3 逐次 (排他) 性の保証 56
4.3.4 割り込み機能 57
4.4 他システムとの比較 60
4.5 実現方式 62
4.6 評価 63
4.6.1 プログラムサイズ 64
4.6.2 実行時間 65
4.7 結言 67

第 5 章 クラスタにおける効率的な並列性制御、負荷分散方式 –医用放射線量分布計算への応用 68

5.1 緒言 68
5.2 分散型並列処理支援ルーツ ParaJET 69
5.2.1 機能概要 70
5.2.2 負荷分散方式 71
5.2.3 他システムとの比較 73
5.3 医用放射線量分布計算への応用 73
5.3.1 医用放射線量分布計算 74
5.3.2 並列処理システム 77
5.3.3 並列処理方式 80
5.3.4 評価 83
5.4 結言 91

第 6 章 おわりに **93**
6.1 本論文のまとめ 93
6.2 研究の成果 95
6.3 今後の課題と展望 97

謝辞 **99**

参考文献 **101**

発表論文一覧 **106**

図 目 次

2.1 資源木とユーザ・タスクの構造 15
2.2 保護フィルタ付き通信 20
2.3 各バッファ・サイズの実行時間と PIMOS プロセスのオーバヘッド . . 24

3.1 仕事量と負荷の均等化の間隔 33
3.2 近傍の忙しさ 35
3.3 15 パズル (問題 D，問題 E) 37
3.4 LLS-G と STB の台数効果 40
3.5 LLS-G と STB の通信メッセージ数 42

4.1 オブジェクトとリモート・オブジェクト 49
4.2 *ANUFO* のプログラム例 50
4.3 sync 変数の記述例 1 (書き込み/読み出し) 53
4.4 sync 変数の記述例 2 (マージ) 54
4.5 stream 変数の記述例 56
4.6 *ANUFO* から C++へのトランスレート 63
4.7 台数効果 66

5.1 ParaJET の負荷分散方式 72
5.2 三次元 CT 画像 75
5.3 線量分布計算 76
5.4 システム構成 78
5.5 並列処理クラスタ 79
5.6 ソフトウェア構成 79

5.7 並列処理方式 82
5.8 計算モデル 85
5.9 線量分布計算の計算結果 85
5.10 DRR 計算の計算結果 85
5.11 線量分布計算の速度向上率 88
5.12 DRR 計算の速度向上率 88
5.13 並列処理方式の評価 89

表 目 次

3.1 LLS-G と STB の実行時間 39
3.2 LLS-G と STB のメッセージ数 41

4.1 13-Queen のプログラムサイズ 64
4.2 各 WS の性能比率 65
4.3 13-Queen の実行時間 65

5.1 実行時間（秒）とその内訳 86

第1章　はじめに

1.1　本研究の背景

近年，情報処理分野の発展は，CPU の高クロック化，メモリの大容量化，ネットワークの高速化に支えられ，目覚しいものがある．さらに，より高い計算パワーの要求に応えて，サーバ型の計算機においては，複数の CPU を共有メモリで搭載するものが当たり前のようになってきており，各ユーザのジョブのようなお互いに独立した処理については，ユーザが意識しなくても，並列に処理されるようになってきている．

一方，分散メモリ型の並列計算機に関しては，並列専用機の形態ではなく，コストパフォーマンスの観点から，通常のパーソナル・コンピュータ（パソコン：PC）を汎用のネットワークで接続したクラスタ型のものが多く利用されるようになってきた．さらにそのサイズも小型化が進み，高さ 4cm 程度の PC やボード（ブレード）型の PC がクラスタの構成要素として使用され，サイズがコンパクトになってきたため、千台以上の大規模な PC クラスタが構築されている．また，米国エネルギー省による ASCI 計画や日本の旧科学技術庁（現 文部科学省）による地球シミュレータ計画においては，数千台規模の並列処理環境が構築されている．さらに，近年，広域ネットワークが整備され高速化されたことにより，複数の遠隔地に存在する計算資源を 1 つの大きな並列処理環境として扱うグリッド・コンピューティングが注目を集めている．このように，分散メモリ型の並列処理環境は，ますます大規模化し，比較的単純な構成で単にプロセッサの数を増やすことにより，システム性能を向上させようとする傾向にある．また一方で，PC や汎用ネットワークの低価格化によ

り，ある程度の規模の PC クラスタであれば，安価に構築することが可能となり，分散メモリ型の並列処理環境が利用し易くなってきている．

このような並列処理環境を利用する目的は，高速化に他ならない．すなわち，利用者にとっては，プログラムが機能的に動作すれば良いといったものではなく，**高速に**動作しなければ意味がない．しかし，この並列処理環境の利用技術が未熟であれば，いくら構成プロセッサ数を増やしても，各プロセッサの負荷の不均衡やプロセッサ間の通信オーバヘッドなどが原因で，並列化効果（高速化）は直ぐに頭打ちになる場合が多い．すなわち，実際のプログラム性能がその処理環境が潜在的に持つピーク性能になかなか近づかないことが多々ある．また，コンパイラなどによる自動並列化の研究も様々な方面で行われているが，分散メモリ型の並列処理環境においては，実用レベルに達しているとは言い難い．このように，並列処理環境では利用技術を持たないと，その高速性能を引き出すことは難しい．これは，並列処理においては，負荷を各プロセッサに均等に分散したり，ぞれぞれ並列に動作する処理を制御しなければならないなど，逐次処理の世界では存在しなかった新たな技術が必要となるからである．

これまで，多くの並列処理に関する研究開発がなされてきた．特に旧通産省による国家プロジェクトとして，1980 年代では第五世代コンピュータ，1990 年代ではリアルワールドコンピューティングが行われてきた．これらのプロジェクトでは，並列処理アーキテクチャのみならず，並列言語処理系，並列オペレーティングシステム，負荷分散ミドルウェア，並列計算モデル・アルゴリズムの研究開発も成されてきた．このように，並列処理にとっては，並列計算機としての処理方式だけでなく，並列計算機の効率的な利用技術の研究も非常に重要視されてきた．

1.2　本研究の目的

並列処理を行うには，まず，処理を複数に分割し，その分割された個々の処理を各計算資源に分配し，各処理間で同期・通信を行いながら，計算を進めていく．従っ

て，並列処理環境の利用技術とは，処理負荷を各計算資源に分散させる負荷分散技術や並列に動作する各処理を管理・制御する並列性制御技術となる．そもそもの並列処理の目的である高速化のためには，これら負荷分散や並列性制御自体がオーバヘッドとならず，効率良く行わなければならない．

この並列性制御や負荷分散を実現するには，(1) オペレーティングシステムやミドルウェアにより，システム側でこれらの機能を提供する方法，(2) プログラミング言語により，ユーザ自身が並列性制御や負荷分散を行い易くするための機能を提供する方法，(3) ユーザ自身で行う方法が考えられる．この中で後者 2 つについては，ユーザ自身が並列性制御・負荷分散を行うのであるから，その効率性のみならず，実現容易性も重要となる．一般に，逐次処理環境において速度性能面で不満のある処理を並列処理環境に移行する．従って，並列性制御・負荷分散を行う上では，逐次処理環境からの移行性を考慮すべきである．すなわち，必要最小限の変更により並列処理環境に移行できるようにしたい．これは，これまでの逐次処理環境におけるプログラム財産を有効に活用するためだけでなく，近年の並列処理環境が逐次処理環境をネットワークで接続したクラスタの形態を採っているものが多いため，既存の（単一）プロセッサで効率良く動作するプログラムというのが重要となるからである．

以上の背景から，本研究の目的は，近年の並列処理環境の大規模化を鑑み，そのような環境においてもオーバヘッドを抑えた効率的な並列性制御・負荷分散方式を提案することである．また，並列処理環境の構成形態及び並列処理の適用範囲の拡大を考慮し，逐次処理環境からの移行容易性を考慮した方式を提案する．

1.3 対象とするアーキテクチャと並列分割形態

本研究で対象とする大規模並列処理環境とは，ネットワーク接続のマルチプロセッサにおいて，近くのプロセッサ同士の通信時間と遠くのプロセッサ同士の通信時間が異なるような環境である．プロセッサ数が多くなると，各プロセッサを接続する

ネットワークは階層構造にならざるを得ない．この階層を介したプロセッサ間通信が階層内のプロセッサ間通信に比べて，無視できない程のコスト高である場合，できるだけ遠くのプロセッサ同士の通信を抑えた方式を採用すべきである．本研究では，そのために，並列性制御や負荷分散において，集中して管理・制御せずに，分散管理を行う方式を提案する．具体的な規模としては，128 台 ~512 台程度であり，これは全体を 1 システムとして管理できるサイズである．最近の PC クラスタには千台を越えるものも多く存在するが，そのようなシステムにおいても，実際の運用ではシステムを複数に分けて，128 台 ~512 台程度で使用している場合が多く，このような規模を対象とすることは適用性が高いと考える．

また，逐次処理環境からの移行性を考慮した方式に関しては，百万円 ~ 2 百万円程度で容易に構築できる，16 台程度の PC クラスタを対象とする．このような環境においては，複雑な分散管理を考えるよりは，単純な構成を採り，逐次処理環境からの移行性を重視した方式を提案する．

OS やプログラミング言語機能は，並列分割形態や問題タイプに依存せず，汎用的な方式を採用すべきであるが，負荷分散においては，どのような応用にも適用可能な汎用的な方式を提案するのは非常に困難である．そこで，本研究の負荷分散方式の検討においては，大規模並列処理環境においてもスケーラブルに並列化効果が現れ，逐次処理環境からの移行性も容易である OR 並列型探索問題や計算領域分割による並列分割形態に対象を絞る．

このような形態の場合，最終的な計算結果の送信を除いて，並列に実行される各実行主体同士の通信は無いため，負荷の分散において粒度（各実行主体の実行時間）を考慮するのみで良く，各実行主体同士の関連性を考慮する必要がない．ただし，OR 並列型探索問題の場合は探索の各世代毎に実行主体数が異なり，領域分割の場合は分割された各領域のサイズと粒度（実行時間）が比例しない場合が多い．従って，実行の前に静的に負荷を分散させることは困難であり，実行過程の状況により動的に負荷を分散させる動的負荷分散方式を採用する必要がある．

1.4 本論文の構成

本論文は，以下のように構成される．

第2章では，大規模並列処理環境におけるオペレーティングシステム (OS) での効率的な並列性制御方式について述べる．非常に多くのプログラムが高並列に動作している大規模な並列処理環境において，その能力を最大限に引き出すためには，正常時の効率的な実行のみならず，異常事態が起きた際の備えが重要となる．このような処理を行うのは，OS の仕事であり，ここでは，第五世代コンピュータ・プロジェクトにおいて開発された並列推論マシン PIM 用の OS である PIMOS を例にして，大規模並列処理環境における OS での効率的な並列性制御方式を提案する．

第3章では，大規模並列処理環境向けの負荷分散方式について述べる．プロセッサ間の距離により通信時間に差異のあるような大規模な環境においては，負荷分散そのものが大きな負荷となる場合がある．ここでは，大規模並列処理環境でもスケーラビリティの面で並列化効果が高く，適用可能な応用分野も広い OR 並列型探索問題を主対象として，大域的な情報を用いずに近隣プロセッサの局所的な情報のみで均等化を行いながら自発的に負荷を分散する方式を提案する．

第4章では，効率的な並列性制御を行なうための言語サポートについて述べる．プログラミング言語レベルで，並列性制御機能を提供するためには，単純なデータ並列のような1階層の並列分割形態だけでなく，より複雑な様々な分割形態に対応する必要がある．また，性能チューニング等を考えると，その分割個所（並列実行の単位）をできるだけ容易に変更できるようにすべきである．ここでは，データアクセスの局所性等の並列実行可能性及び逐次処理環境からの移行性を考慮し，オブジェクト指向におけるオブジェクト群単位の並列性制御が容易に行え，更に既存の逐次プログラムからの変更を最小限に抑えた，C++言語をベースとした並列処理言語機能を提案する．

第5章では，逐次処理環境からの移行を容易にし，安価で容易に構築可能な比較的小規模なクラスタ環境における並列性制御及び負荷分散を支援する方式について

提案し，医用放射線量分布計算への適用例を示す．本応用の計算手法は，いわゆるレイトレーシングであり，その並列化は計算領域を分割することにより行える．ここでは，そのような比較的単純な並列処理形態に対応し，逐次処理環境におけるプログラムがほぼそのまま利用可能な負荷分散方式を提案する．さらに，より大きな並列化効果を上げるために，対象問題の特性を考慮した方式を加味する．

最後の第 6 章では，本論文のまとめを行う．

第2章　大規模並列処理環境における OS での効率的な並列性制御方式

2.1　緒言

非常に多くのプログラムが高並列に動作している大規模な並列処理環境において，その能力を最大限に引き出すためには，正常時の効率的な実行のみならず，異常事態が起きた際の備えが重要となる．例えば，ある個所で異常事態が発生した際に，他の（正常に動作している）個所にできるだけ影響を与えないように，異常個所のプログラムの実行を停止したり，再開できるようにしなければならない．このような処理を行うのは，オペレーティングシステム (OS) の仕事であり，大規模並列処理環境においては，高並列に動作しているユーザプログラムの OS による効率的な制御方式が，システム全体の性能を左右すると言って良い．

第五世代コンピュータ・プロジェクトにおいて，高度な知識情報処理を行うに当たって必要とされる計算能力を提供するために開発された並列推論マシン (PIM) [1, 2] や，並列ソフトウェアの研究を促進させるためのシステムであるマルチ PSI[3, 4] は，プロセッサが複数存在する並列マシンであるだけでなく，並列論理型言語 KL1（核言語第１版）をベースとしているため，ゴールと呼ばれる非常に多くの実行主体が並行に動作する大規模並列処理環境である．これら PIM やマルチ PSI などの並列推論マシン用の OS は，PIMOS(Parallel Inference Machine Operating System)[5, 6, 7, 8] と呼ばれている．

OSの主な仕事は，各入出力装置をユーザに共通のインタフェースで提供したり，CPU資源を管理し，各実行主体(UNIXではプロセス)にできるだけ公平に計算資源を提供し，ユーザの過ちや異常事態からシステム全体を保護するなど多岐に渡る．また，並列処理環境向けのOSでは，並列に動作するプログラムの計算資源を有効に利用し効率良く動作させるためのスケジューリング機構[9]，各プログラム間での共有データの排他制御のための同期機構[10]及び分散されたメモリを効率良く管理・制御するメモリ管理[11]も重要な仕事であり，多くの機関で研究開発が行われてきた．

PIMOSのように遠いプロセッサ間同士と近くのプロセッサ間同士の通信時間に差異があるような大規模な並列処理環境の場合，これらのOSの仕事のうち，システム全体の様々な計算資源を一括して効率良く管理・制御することが重要となってくる．その際，集中管理を避け，階層的に分散管理・制御する方式が効率面で得策であり，本章では，PIMOSを例にしてこの方式について述べる．

PIMOSは，並列論理型言語KL1により，言語レベルで提供されるプログラムの実行制御機能を利用して，入出力装置やユーザプログラムの管理・実行制御機能を実現している．また，ユーザの過ちがシステム全体に影響を及ぼさないような保護機能を提供している．

2.2 プロセス

ユーザ・プログラムが消費するCPU時間，メモリ，入出力装置などの資源をそれぞれのプログラムが並列に動作する環境の下で管理し，制御するのは，容易なことではない．例えば，OSがあるデータに対する処理を行っている最中に，そのデータの状態をユーザ・プログラムによって変更されることを防ぎたい場合がよくある．そのような場合，従来の逐次計算機上では，「OSがそのデータを処理している最中は，ユーザ・プログラムを動作させない」という実行順序によってデータの変更を防ぐことができた．しかし，大規模な並列環境では，並列性を犠牲にせずに，そのような実行順序の制御はできない．つまり，OSがそのデータを処理している最中に，全

てのユーザ・プログラムの実行を停止させることになり，これは明らかに並列性を犠牲にすることになる．従って，従来の逐次計算機と同様の方式を採用することはできない．

そこで PIMOS では,「プロセス」[12] と呼ばれるプログラミング・スタイルを導入することにより，この問題を解決している．このプロセスは，管理しているデータと通信用の変数 (ストリーム) を保持している．プロセスの外からこのデータにアクセスする場合は，このストリームにメッセージを送るという形でしか行えない．従って，そのストリームにアクセスすることができないプログラムは，このデータにアクセスすることはできない．つまり，データへのアクセス・パスを 1 つにすることで，複数のプログラムからの競合を防いでいる．

PIMOS が管理する全ての資源は，このプロセスを用いて，ストリームを介して通信することによって管理されている．つまり，このシステムでの資源へのアクセスは，従来のシステムのスーパバイザ・コールのような，資源を集中管理しているモジュールへの通信によって行われるのではなく，それぞれの資源を管理しているプロセスへストリームを介して通信することによって行われる．従って，PIMOS では，これらのストリームを管理することが，資源を管理することに相当している．

ストリームを介したメッセージの送受信は，具体的には，共有変数の具体化によって行われる．例えば，ユーザが，ある入力装置から文字列を読み込む場合は，以下のようなプログラムになる．

```
?- pimos(Req), user(Req).
user(Req) :-
    Req = [getb(N,String)|ReqT], ...
pimos([getb(N,String)|ReqT]) :-
    readFromKbd(N, KBDString),
    KBDString = String,
    pimos(ReqT).
```

ユーザとPIMOSとは，共有変数Reqをストリームとして利用することで通信を行う．ユーザは，そこにN文字読み込みたいと言う要求getb/2を送る．PIMOSは，そのメッセージを受け取ると，readFromKbd/2により必要なN文字分の文字列を読み込み，Stringを読み込まれた文字列に具体化することによりユーザに返す．そして，ReqTにより次の通信が行われる．

このpimos/1のようなプログラミング・スタイルをプロセスと呼んでおり，メッセージによってのみ，自分の管理している資源(この例ではキーボード)への操作を許している．

ユーザとPIMOSとの通信は，最初に与えられた唯一の共有変数(ストリーム)Reqを基に，共有変数を増やして行くことで進んで行く．例えば，ユーザが新しいウインドウを生成し，そこに対して入出力を行う場合は，以下のようになる．

```
?- pimos(Req), user(Req).
user(Req) :-
   Req = [create_window(Window)|ReqT],
   Window = [getb(N,String)|WinT],
          .....
```

新たに生成されたウインドウに対する入出力操作は，PIMOSから与えられた通信用変数Window及びそこから派生した変数を介してのみ行える．つまり，これらの変数を持っていない他のプログラムは，このウインドウにアクセスすることはできない．従って，PIMOSではこれらの通信用の変数をユーザに与えることが，資源にアクセスすることのできる権利(capability)を与えることに相当しており，この機構により，ユーザの不正なシステムへの介入を防いでいる．

2.3 言語レベルで提供される並列実行管理・制御機能

PIMOS の記述言語である KL1 は Flat GHC[13] を基にしているため，全てのゴールは AND 関係になっている．従って，このままでは，ユーザ・プログラムが (上記の例の user(Req) から呼ばれたサブ・ゴールの内の１つでも) 失敗した場合は，PIMOS も失敗し，システムダウンを起こしてしまう．これは，PIMOS とユーザ・プログラムが同じレベルになっているからである．しかし，ユーザ・プログラムと OS とは，オブジェクト・レベルのプログラムと，そのオブジェクト・プログラムを監視/制御するメタ・レベルのプログラムの関係になるべきである．

メタ・プログラミングの機能を実現する単純な方法には，インタプリタがあるが，実行効率面で大きな損失が生じてしまう．そのため，PIMOS の記述言語である KL1 では，このメタ・プログラミングの機能が言語レベルで組み込まれており，これを「荘園」と呼んでいる．荘園には，実行制御/監視機能, 資源管理機能, 例外処理機能が組み込まれている．

荘園は以下のような組込述語を用いて生成する．

```
execute(Goal, ExpMask, Control, Report)
```

ここで, 各引数は以下のような意味を持つ．

1. Goal
   荘園の中で実行すべきゴールを指定するための変数である．
2. ExpMask
   この荘園で，どのような例外を処理するかを決めるためのマスク・パターンである (詳しくは省略する).
3. Control
   荘園内の実行を制御したり，資源の割り当てを指定するためのストリームである．ここには，以下のような機能を持ったメッセージを流すことができる．
   - 荘園内の実行の開始/中断/再開/放棄

- 荘園内で消費できる計算資源の許容量の問い合わせ/追加
- 荘園内でこれまでに消費された計算資源量の問い合わせ

4. Report

   荘園内の実行状態が報告されるストリームである．ここには，以下のような2種類の報告が行われる．

   (a) 事象の報告

   荘園の実行にともなって生じた事象が報告され，これには以下のようなものがある．

   - 荘園内の実行の終了
   - 荘園内で消費できる計算資源の不足
   - 荘園内で発生した例外事象 (オーバフロー, ゴールの失敗など)

   (b) 受け取りの確認

   制御ストリームからのメッセージを受け取ったことを報告する．荘園の制御/監視は，制御用と報告用の2つのストリームによって行うので，この確認の報告がないと，それらのストリームに流れるメッセージ間の同期を取ることができない．例えば，制御ストリームから計算資源の許容量を追加した後，報告ストリームから計算資源の不足が報告された場合，

   - 資源を追加したがそれでも不足してしまったので，更に追加する必要がある．
   - 資源追加のメッセージが届く前に不足を検出したので，その後資源は追加され，今は追加する必要がない．

   のどちらか分からない．しかし，報告ストリームから資源追加メッセージを受け取ったという報告がなされれば，それと資源不足の報告との前後関係によって，上記のどちらであるかを知ることができる．

荘園内のゴール群は，荘園外とは独立したAND関係を成している．すなわち，荘園内での失敗は荘園内に閉じたものであり， 荘園外のゴールを巻き添えにすることはない．従って，概念上，荘園はインタプリタを機械語レベルで実現したものであると考えて良い．

また，荘園の中で更に荘園を作ることも可能である．この場合，外側の荘園に実行の中断を指令すれば，内側の荘園の実行も中断される．外側の荘園の実行を再開すれば，内側の荘園の実行も再開される．また，内側の荘園が使用できる資源は，外側の荘園から分け与えられるものである．つまり，内側の荘園が消費した資源は，外側の荘園も消費したものとして扱われる．

このような管理方式を取ることにより，メタ・レベルはオブジェクト・レベル以下にどのようにメタ/オブジェクトの階層構造があるかを意識することなく，全体として管理することが可能となっている．

この荘園の機能を使った，ユーザと PIMOS との通信は以下のようになる．

```
?- pimos(Req, Cnt, Rep),
   execute(user(Req), -1, Cnt, Rep).
```

PIMOS は，変数 `Cnt` によりユーザ・プログラムの実行を制御し，変数 `Rep` により実行の監視及び資源の管理を行う．このように荘園は，制御ストリームと報告ストリームをインタフェースとした，言語レベルで提供されているプロセスと考えることができる．

2.4　資源木による資源管理・実行制御

OS が管理すべき資源には，荘園で管理されるような CPU 時間やメモリのほかに，入出力装置などがある．前者のような資源を「言語定義資源」と呼び，後者のような資源を 「OS 定義資源」と呼んでいる．このように，PIMOS がユーザの消費する全ての資源を管理するためには，荘園の機能だけでは不十分であり，入出力装置などの OS 定義資源も管理する機構を導入しなければならない．

2.4.1 タスク

これらの言語定義資源や OS 定義資源の管理を行うために，PIMOS では,「タスク」と呼ばれる資源管理の単位を導入している．このタスクは，荘園の機能を用いて実現されており，荘園と同様に任意にネストすることができる．このタスク内で生成された子タスクも，親タスク内で使用されている資源の 1 つとして管理される．

ユーザがタスクを生成すると，タスク制御ストリームとタスク報告ストリームが返される．ユーザは，タスク制御ストリームにメッセージを送ることにより，タスク内の実行を制御したり，タスク内で使用されている資源に対して問い合わせることができる．また，タスク報告ストリームから知らされる情報により，タスクの実行状態を監視したり，例外処理を行なうことができる．

2.4.2 資源木

タスクの中で使用されている全ての資源 (言語定義資源及び OS 定義資源) は,「資源木」と呼ばれるストリームでお互いに結合されたプロセスの木構造によって，タスクの階層ごとに管理されている．

タスクの階層構造及び資源木の構造の例を図 2.1 に示す．この例では，3 つのタスクが存在しており，それぞれのタスクは以下のような OS 定義資源を使用している．

- 親タスク：1 つの入出力資源と「子タスク 1」と「子タスク 2」
- 子タスク 1：2 つの入出力資源
- 子タスク 2：1 つの入出力資源

この図の右側の部分を資源木と呼び，ユーザ・タスクとは別の領域に置かれる．資源木の各ノードは，タスクの階層にそって木構造になっており，各タスク内の資源

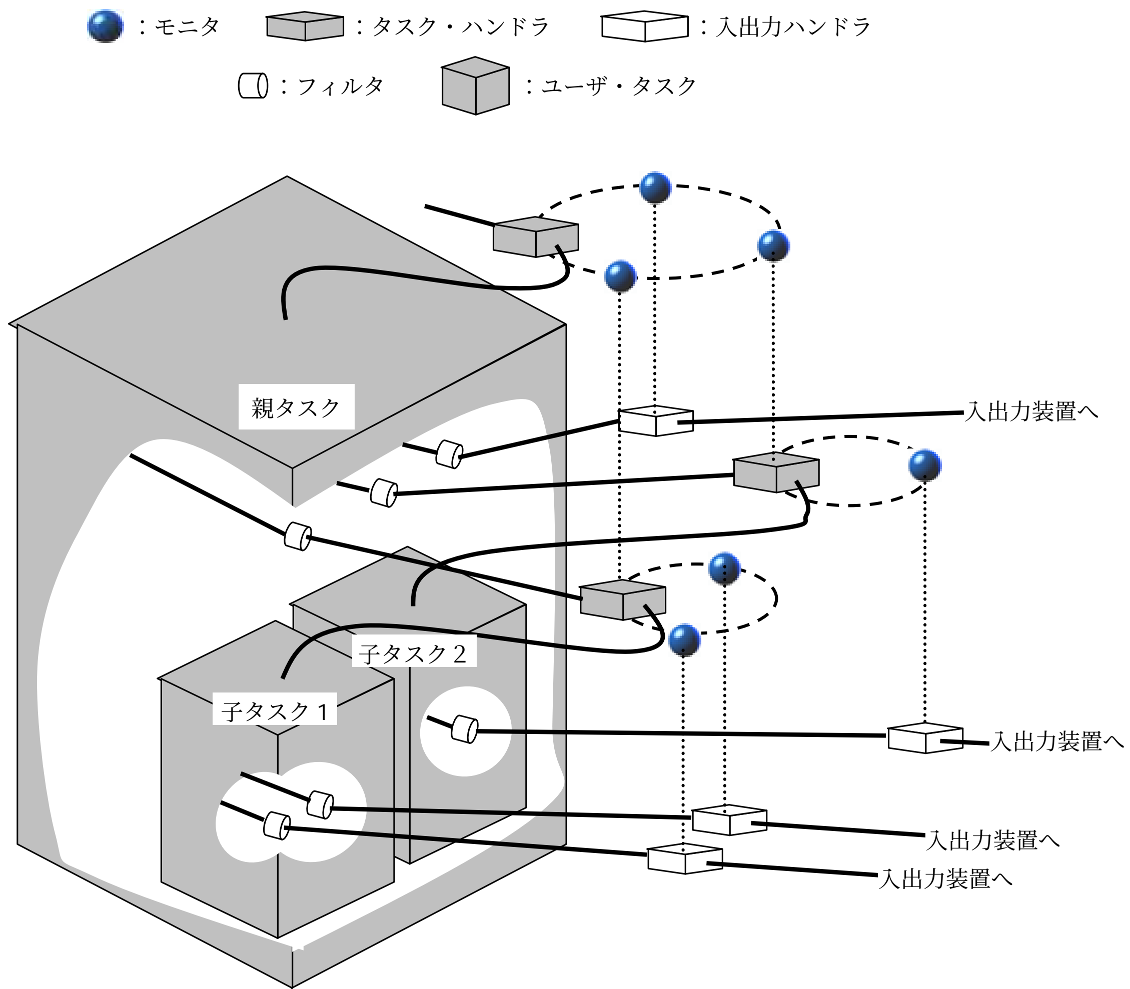


図 2.1: 資源木とユーザ・タスクの構造

は，それぞれが輪構造で結ばれている．以下に，資源木を構成する各要素について説明する．

1. タスク・ハンドラ
   タスク内の資源を管理しているプロセスであり，ユーザがそれらの資源を操作する時に，ここにメッセージが流れて来る．また，祖先タスクからもここで管理している資源に対するメッセージがモニタ (後述) を経由して流れて来ることもある．ここでは，タスクを実現している荘園の制御ストリームと報告ストリーム (図ではタスクの上につながる太実線) を保持することで，タスクの実行を制御し，実行状態を管理している．
   また，このタスク内で使用されている入出力装置や子タスクなどの OS 定義資源は，それぞれ対応するモニタを輪状に繋げること (これを資源ループと呼ぶ) で管理されている．これらの資源に対する操作メッセージは，モニタを通して子資源のハンドラに送られる．
2. 入出力ハンドラ
   入出力装置を資源としてユーザに見せるためのプロセスであり，一般の入出力要求に応じる機能と，この資源に対するメタな問い合わせ (例えば，どのような種類の資源なのかなど) に答える機能を持っている．前者のようなメッセージはユーザからフィルタを通して送られ，更に入出力装置に近いプロセス (デバイス・ドライバ) に再送される．また，後者のようなメッセージはモニタを通して送られ，このプロセスで処理される．
3. モニタ
   タスク・ハンドラからのメッセージを自分が保持しているハンドラに送るかどうかを決める．そのメカニズムについては，後述する．
4. フィルタ
   ユーザの誤りからシステム全体を保護するためのプロセスである．ユーザと PIMOS が通信する場合には，必ずこの種の保護フィルタが付けられる．この保護フィルタはユーザ・タスク上で実行されるが，プログラム・コードとしては PIMOS が提供するものである．なお，保護フィルタの詳細については，2.5 節で記述する．

2.4.3　OS 資源の管理方式

PIMOS が管理する総ての OS 定義資源には，各タスク内でユニークな ID (資源 ID) が付けられており，資源 ID 列によって特定の資源を指定することができる．例えば，

$$3, 4, 2$$

であれば，タスク内の 3 番目の資源内の 4 番目の資源内の 2 番目の資源を表わしている．

ユーザは，タスク制御ストリームを通して，メッセージを送ることで OS 定義資源に対する制御を行える．この時，資源 ID をメッセージに付けるとその資源に対する操作となる．モニタは，それぞれ自分が管理している OS 定義資源の ID を保持している．そして，自分の所に流れてきたメッセージに付いている ID と自分の ID を比べ，一致すればそのメッセージを (資源 ID 列の残りを付けて) 自分のハンドラに送り，一致しなければ隣のモニタに再送する．

タスク・ハンドラは，親モニタからメッセージを受け取ると，それが自分宛であれば自分で処理し，子孫資源宛であれば自分の資源ループに再送する．資源ループに送ったメッセージがもう一方のストリームからそのまま返ってきた場合は，指定した ID に相当する資源が存在しなかったことになり，エラーとして扱われる．

タスク・ハンドラは，タスクを実現している荘園の報告ストリームを監視している．そこから実行の終了または放棄が報告されると，そのタスク内で使用されていた未解放の全ての資源 (資源ループ) に対して，タスク・ハンドラが解放要求を送ることにより，資源を自動的に解放する．

OS 定義資源を解放する時には，モニタが自分の兄からのストリームと弟へのストリームをユニファイすることにより，輪の中から消滅する．

2.4.4　従来 OS との比較

従来の OS では，タスク (システムによってはプロセスと呼ぶこともある) は，1 つの平坦なテーブルで集中的に管理されている場合が多い．また，入出力装置への操作も，例えば,「タスク内の入出力資源テーブルの N 番目に登録されている出力装置に出力する」という形で行われることが多い．従って，これらの資源に対する操作を行うたびに管理テーブルへアクセスすることになる．

従来の逐次計算機システムの場合は，逐次的に実行されるため，この管理テーブルへのアクセス集中がそれほど問題になっていなかった．一方，並列計算機の場合は，独立した資源に対する操作を並列に行うことができるが，共有メモリ型の並列計算機の場合には，資源へのアクセス頻度などを考えると，集中管理によるネックがそれほど問題となっていない．

しかし，PIMOS が対象としている分散メモリ型でしかも並列度の高い並列計算機システムの場合は，集中管理によりプロセッサ間の通信量が増えてしまうので，並列性を犠牲にすることになる．そのため PIMOS では，資源のアクセスが,「資源へのストリームに直接メッセージを送る」という形になっており，独立した資源への操作は，それぞれ並列に行えるようになっている．

また,「タスク内で使用している全ての資源の状態を返す」といった処理の場合，タスク・ハンドラは，資源ループへその旨のメッセージを送るだけなので，すぐにユーザからの次のメッセージに対する処理を行える．またモニタは，そのメッセージを受け取ると，資源ハンドラにメッセージを送ると共に，次のモニタに同じメッセージを送る．このように，メッセージが各資源ハンドラに分散されるため，それぞれ独立して各資源の状態を返すことができる．

2.5 システムの保護機構

2.5.1 問題点

ユーザ・プログラムの実行をメタなレベルから監視する機構は，KL1 の荘園機能により行える．しかし，PIMOS とユーザの間での通信では，KL1 に個々のユニフィケーションを制御/監視する機能がないため，単純に通信を行ってしまうと，システムダウンに陥ることがある．

ユーザが指定した長さ (文字数) の文字列をキーボードから読み込む場合を考えてみる．

```
user(Req) :- true |
    Req = [getb(N,String)|ReqT], ...
pimos([getb(N,String)|ReqT], Cnt, Rep) :- true |
    readFromKbd(N, KBDString),
    KBDString = String,...          --- (a)
```

ユーザは，通信用ストリーム `Req` に入力要求メッセージ `getb/2` を送る．PIMOS はメッセージ `getb/2` を受け取ると，`readFromKbd/2` により必要な N 文字分の文字列を読み込む．この時，`readFromKbd/2` は，変数 `N` が具体化されるのを待つ．そして，変数 `String` を読み込まれた文字列 `KBDString` に具体化し，ユーザに返す．しかしこのプログラムには，以下のような 2 種類の問題点が含まれている．

1. 問題点 1

   ユーザが以下のように (誤って) 文字列が入るべき変数をアトムに具体化してしまったとする．

   ```
   String = foo                   --- (b)
   ```

   並列環境の下では，効率を落とさずにゴール (a) と (b) の実行順序を規定するこ

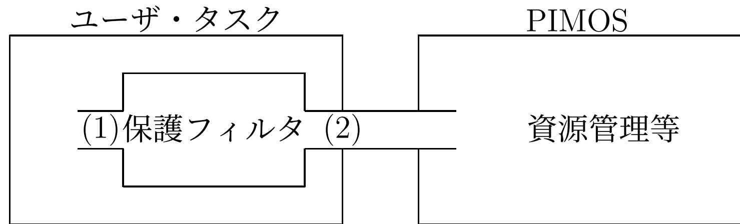


図 2.2: 保護フィルタ付き通信

とはできない．そのため，(a) が先に実行されれば (b) が失敗するし，(b) が先に実行されれば (a) が失敗する．前者の場合であればユーザ・タスク内で失敗が起きるだけであり，問題はないが，後者の場合だと PIMOS 側で失敗が起こるので，システムダウンに陥ってしまう．KL1 では，ユニフィケーションが成功するかどうかをチェックするような (テスト・アンド・ユニファイ) 機能を持っていない (もし，KL1 がテスト・アンド・ユニファイの機能を持った Concurrent Prolog[14] を基にしているのであれば，問題とはならない)．例えば，上記の `String` が未定義変数かどうかのチェックができたとしても，そのチェックとユニフィケーションとの間に，`String` が他の値に具体化されてしまう可能性があるからである．

2. 問題点 2

   タスク側が文字数設定用共有変数 `N` に値を設定するのを忘れたか，または設定する前にタスクの実行が放棄された場合，PIMOS 側の `N` の具体化を待つプロセス (`readFromKbd/2`) が永遠に待ち続け，デッドロックしてしまう (これは，たとえテスト・アンド・ユニファイの機能があっても問題となる)．

2.5.2 保護フィルタ

これらの問題点を解決するために，PIMOS とユーザ・タスク間のストリームに保護のためのフィルタ・プロセス (保護フィルタ) を設け，そのプロセスをユーザ・タスク内で実行することにした (図 2.2)．

タスクは PIMOS とつながっているストリームに直接メッセージを送らずに，保

護フィルタへのストリーム (図 2.2 の (1)) へメッセージを送る．このフィルタはユーザのメッセージを絶対に失敗しない形式に変換し，また，PIMOS 側で待ち続けないことが保証されるまで待ってから，PIMOS へのストリーム (図 2.2 の (2)) へ変換されたメッセージを送信する．

このように PIMOS 側のストリームをユーザに直接見せないため，ユーザの誤りが PIMOS 側へ波及するのを防ぐだけではなく，ユーザからの悪意を持った PIMOS への侵入を保護フィルタが監視して防ぐことができる．

この保護フィルタの具体的な仕事は以下のようなことである．

- ユーザが値を設定すべきデータ部分は，その値が設定されたことを確認してから PIMOS 側へ送る (問題点 2 の解決)．これにより，PIMOS にメッセージが送られた時には，それらの値は確定していることが分かる．ただし，ストリームの場合だけは，そのヘッド (第 1 要素) の具体化しか待たない．そうしないと，そのストリームを閉じるまで，全ての要求が PIMOS 側に流れないことになるからである．PIMOS では，総てのストリームを資源として管理しており，タスクの実行が放棄された時に資源の解放処理を行なうので，ストリームの場合は問題点-2 を考慮する必要はない．
- PIMOS 側から返されるデータ部分は，必ず未定義状態であるような別の変数を送る (問題点 1 の解決)．そして，PIMOS から値が返された後で，ユーザが指定した変数を返ってきた値に具体化する．これにより，PIMOS 内での値を返すユニフィケーションは，必ず未定義変数とのユニフィケーションとなり，失敗することはない．

例えば，上記のキーボードの例に対する保護フィルタ・プログラムは以下のようになる．

```
pfilter([getb(N,String)|T], OS) :- wait(N) |
    OS = [getb(N,OSString)|OST],
```

```
      waitAndUnify(OSString, String),
      pfilter(T, OST).
waitAndUnify(OSV, USERV) :- wait(OSV) |
      OSV = USERV.
```

保護フィルタ (pfilter/2) は，読み込む文字数 N に値が設定されるまで待ってから PIMOS へ要求を送信する．また，読み込まれた文字列が設定される変数 String の代わりに，未定義変数 OSString を PIMOS に送る．そして，その変数の値が設定されるのを待ってから，ユーザの変数 String へ文字列が設定される．従って，ユーザが変数 String を誤った値に具体化したとしても，ユーザ・タスク内で実行されるフィルタ (OSV=USERV) が失敗するだけで PIMOS には影響はない．

このフィルタは，ユーザ・プログラムを起動する時に，PIMOS により以下のように挿入される．

```
?- pimos(OSReq, Cnt, Rep),
   execute( ( user(Req),
              pfilter(Req, OSReq) ),
         -1, Cnt, Rep).
```

また，ストリーム Req から新たに派生した通信用変数のためのフィルタは，pfilter/2 の中で生成される．例えば，ウインドウを生成するメッセージのための保護フィルタのプログラムは，以下のようになる．

```
pfilter([create_win(Name,Win)|T], OS) :-
      wait(Name) |
  OS = [create_win(Name,OSWin)|OST],
  pfilter_win(Win, OSWin),
  pfilter(T, OST).
```

ユーザは新たに生成されたウインドウに対するメッセージを Win に送るが，それは PIMOS と直接つながれておらず，pfilter_win/2 により保護される．

この保護フィルタのプログラムは，OS-ユーザ間の通信プロトコルを定義すれば，機械的に生成することができる．PIMOS では，プロトコル定義言語を設け，プロトコル・コンパイラを用いて，保護フィルタを自動生成している．

2.6 プロセス・オーバヘッドの評価

PIMOS では，このように実行制御・管理のために，資源ハンドラや保護フィルタ等の管理プロセスをユーザ・プロセスとの間に置いている．これらの OS による実行制御・管理のためのプロセスは，ユーザ・プログラムの実行にとっては直接関係のないものであり，何らかのオーバヘッドが存在することが考えられる．そこで，ここでは，PIMOS で採用しているプロセスをストリームで結合した実行制御方式で，保護フィルタや資源ハンドラなどの各プロセスがどの程度のオーバヘッドになっているかを評価する．

一般に入出力は，1 文字ずつ行うのではなく，何文字かをバッファリングして行う方が，スループットが良くなる．PIMOS が扱うような並列計算機の場合も逐次計算機と同様にこのバッファリングの効果があり，PIMOS では，このバッファリングを行うユーティリティが用意されている．そこで今回は，ある固定のサイズの文字列をバッファリングのサイズを変えて入出力した時の実行時間の変化を

1. オリジナルの PIMOS(保護フィルタ, 入出力ハンドラ付き)
2. 保護フィルタを取った場合
3. 保護フィルタ及び入出力ハンドラを取った場合

の 3 種類について，測定した．

これらの測定を行う場合，実際の入出力装置 (例えばファイル) を使うと，それ

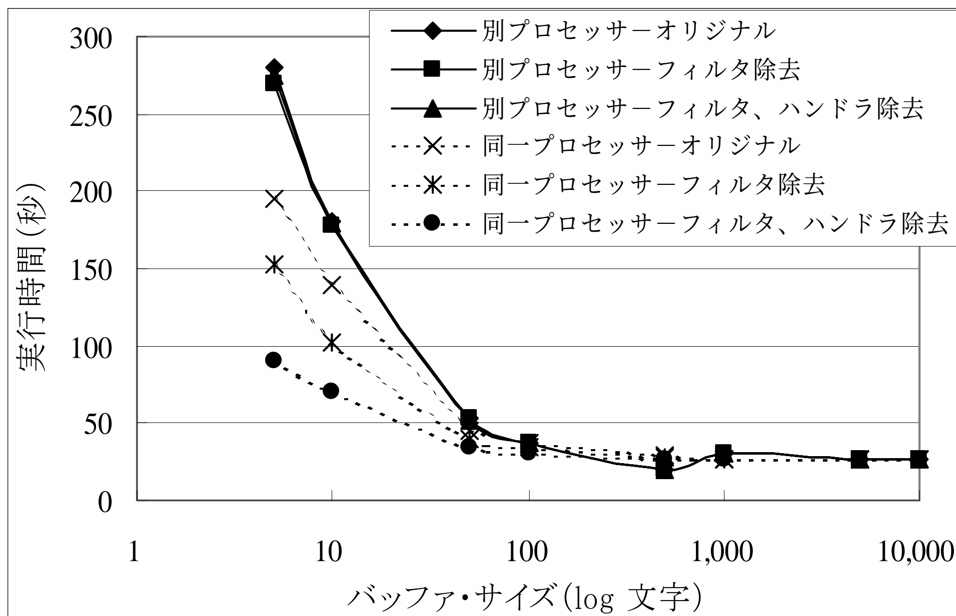


図 2.3: 各バッファ・サイズの実行時間と PIMOS プロセスのオーバヘッド

自体の遅さが目立ってしまうので，今回はダミーの入出力装置を KL1 で作成した．PIMOS での入出力装置は，プロセスとして見えるので，KL1 でそれを摸擬するものを作成するのは容易である．このダミー装置は，あるサイズの文字列を出力したり，指定したサイズの文字列を入力する機能を持っている．

このダミー装置は測定プログラムとは別プロセッサで実行させるのが実際の入出力装置に近い動きになる．しかし，これでは，通信オーバヘッドのために，PIMOS プロセスのオーバヘッドが隠れてしまう可能性がある．そこで，ダミー装置プロセスが測定プログラムと同一プロセッサに存在する場合についても測定した．

1 メガ文字 (1 文字は 2 バイト) の文字列を読み込む場合のバッファ・サイズを変化させた実行時間の推移を図 2.3 に示す．このグラフから以下のようなことが言える．

- ダミー装置が同一プロセッサにある場合には，バッファ・サイズ 1 キロ文字くらいまで保護フィルタや入出力ハンドラのオーバヘッドが出ているのに対し，

別プロセッサの場合は出ていない．これは，バッファ・サイズが 1 キロ文字くらいまではプロセッサ間の通信オーバヘッドが PIMOS プロセスのオーバヘッドを打ち消してしまうほど大きいからである．

- バッファ・サイズが 1 キロ文字より大きい場合は，ダミー装置プロセスが別プロセッサにあろうと同一プロセッサにあろうと，PIMOS プロセスのオーバヘッドが出ていないし，両者の実行時間がほぼ等しくなっている．これは，バッファ・サイズが 1 キロ文字より大きい場合には，ダミー装置の仕事量が PIMOS プロセスのオーバヘッドを打ち消してしまうほど大きいためである．

- ダミー装置プロセスが別プロセッサにある場合，バッファ・サイズが 500 文字近辺の処理時間が最小になっており，それを越えると (1,000 文字近辺で) 急激に処理時間が上がっている．今回の測定で使用した KL1 処理系では，プロセッサ間通信をパケットにより行っており，1 度にプロセス間で通信できるデータのサイズが，500 文字辺りになっている．それを越えると 1 つのバッファでも複数のパケット (マルチ・パケット) に分割されるため，ここで実行時間が不連続になっている．

以上の測定の結果，本来は PIMOS プロセスがオーバヘッドになっているが (同一プロセッサによる測定より)，実際の入出力においてはどのようなバッファ・サイズをとっても，PIMOS プロセスのオーバヘッドは無視できる程度であると言える．また，バッファリングのサイズは，マルチ・パケットになる寸前である 500 文字近辺が最適であることが分った．このことから，現在バッファリング・ユーティリティでは，バッファ・サイズの既定値をこの辺りに設定している．

2.7　結言

以上，ユーザ・タスクや入出力装置などの資源を資源木により階層的に分散管理・制御する方式を示し，保護フィルタにより，ユーザの誤りが OS やシステム全体に波及することや，悪意を持った OS への侵入を防ぐことを示した．また，これらの管理・実行制御をプロセスにより実現し，それがオーバヘッドとなっていないことを評価実験により示した．

PIMOS は，推論マシン用の OS で，論理型言語により記述されており，特殊なシステムのように思われる．しかし，ここで示したプロセスによる記述，階層的な資源の分散管理方式やシステム保護の方式は，一般の大規模並列処理環境に適用できると考えている．プロセスとストリームによる実現方式に関しては，現在の UNIX におけるプロセス（またはスレッド）とソケット通信に置き換えて考えることが出来る．また，大規模な並列処理環境を考えると，UNIX のように計算資源を集中管理することは処理効率の上で考えることはできず，ここで示した資源木のような階層的な分散管理が効率的な並列性制御に適していると言える．一般のシステムにおいては，ユニフィケーションの失敗は存在しないが，ウィルスを持ったプログラム侵入などのあるユーザの誤りを他サイトへ波及させないようにすることや悪意を持った侵入を防ぐことは非常に重要な機能である．UNIX のプロセスと何らかの通信路を用いて，ここで示した保護フィルタのような方式を実現することは可能であり，現在のファイアーウォールの実現手法は，それに類似していると考えることができる．

すなわち，PIMOS における並列性制御機能は，推論マシン上の論理型言語により記述しているという特殊な環境で実現されているが，そこで採用している制御方式は，一般の大規模並列処理環境に適用でき，効率的に作用すると考えている．

第3章　大規模並列処理環境向け負荷分散方式

3.1　緒言

大規模な並列処理環境において，負荷の動的な分散と均等化も，プロセッサ資源の有効利用の観点から非常に重要な問題の１つである．特にプロセッサ間の距離により通信時間に差異のあるような大規模並列環境では，負荷分散や均等化のオーバヘッドを出来るだけ抑え，均等化の際に大域的な情報を用いずに，近隣プロセッサの局所的な情報のみで作業が行えるのが，スケーラビリティという面で有効である．

また，どのような情報に基づいて負荷の分散を行えば各プロセッサの資源を有効利用したことになるのかは，そのシステムあるいはアーキテクチャごとの固有の問題であるが，一般的には CPU の稼働率をその対象として考えることが多い．すなわち，とりあえず CPU を遊休させず稼働率を 100%に維持する，という戦略である．このような戦略では，結果としてすべてのプロセッサでの実行時間が同じくらいになる，という効果はあるが,「均等化」をしているわけではないので，ある時間断面での各プロセッサの持つ負荷量は一般に不均等となり，ある特定のプロセッサでメモリのワーキングセットが大きくなりキャッシュ・メモリのミスヒットが多発する，といった現象が防止できない．

従来の方式の多くは単に CPU を遊休させないということを目標としているため，負荷分配の遅れの防止が困難であると同時に，自プロセッサが所有して実行すべき負荷の適正量を求める手段がなく，負荷の自発的な分配ができない，という問題点

がある．しかし，実用上はこのような戦略でも十分効果的な場合が多く，また，負荷の適正量を算出するために各プロセッサから得るべき情報が明確にできないこと，「均等化」のメカニズムをソフトウェアで実装した際のオーバヘッドが「均等化」の効果を上回る可能性が否定できないことなどにより，複雑な「均等化」のメカニズムを実際の問題に適用しようという努力はほとんど払われていない．

また，どのような並列分割形態にも有効な汎用的な負荷分散方式の考案は非常に困難であり，ある程度，対象とする分割形態を絞って，方式を検討すべきである．探索過程で複数の可能性を調べる問題は，それぞれの可能性の探索に依存関係がない場合が多く，並列に探索できるため，OR 並列型探索問題と呼ばれている．このような並列分割形態の問題は，大規模並列処理環境でもスケーラビリティの面で並列化効果が高く，適用可能な応用分野も広い．また一般に，探索過程におけるそれぞれの可能性の個数を予測することが困難であるため，実行前に負荷を分散させる静的負荷分散方式を採用することが出来ず，実行過程での状況に応じて負荷を分散させる動的負荷分散方式を採用せざるを得ない．ただし，この種の並列分割形態では，分割されたそれぞれの探索処理には依存関係がなく通信が発生しないため，どの探索をどのプロセッサに分配しても，通信オーバヘッドは発生しないという利点がある．

本章では，大規模並列処理環境においてこのような並列分割形態を主対象とし，負荷の動的な均等化問題に対する１つの解決策として，大域的な情報を用いずに均等化を行いながら自発的に負荷を分散し,「世代」という概念を取り入れることで均等化のオーバヘッドを小さく抑えることを可能とした「世代別の拡散型動的負荷分散方式」の提案を行う．

3.2 従来の方式

3.2.1 MLB 方式と STB 方式

マルチレベル動的負荷分散方式 (Multi-Level Dynamic Load Balancing : MLB 方式) [15] は，特定のプロセッサが，問題を互いに独立な仕事に分割する作業を行い，仕事を要求してきたプロセッサに対して割り付ける方式 (このような方式を『要求駆動型』の負荷分散方式と呼ぶ) である[1]．一般にこのような方式ではプロセッサの台数が多くなると仕事を供給するプロセッサがボトルネックになるが，MLB 方式では仕事の分割と供給を行うプロセッサを階層的に複数台割り付けることによってこのボトルネックの解消を行う．このため，階層の構成 (何層にするか，各層における仕事の供給を行うプロセッサを何台にするか，など) をユーザが問題ごとに調整する必要が出てくる．

また、スタック分割動的負荷分散方式 (Stack Splitting Dynamic Load Balancing: STB 方式) [16] は，MLB と同様に要求駆動型のメカニズムを用いながら，仕事の分割および供給を行うプロセッサを特定せず，どのプロセッサも仕事の分割および供給の対象とする方式である．各プロセッサは仕事の供給が受けられるまで，ある戦略に従い自分以外のすべてのプロセッサに順番に仕事を要求する．このため，MLB 方式のような階層化によるボトルネック対策は不要であるが，自分以外の全プロセッサに対して仕事を要求および供給する可能性があるため，大規模な並列処理環境においては局所性が失われ，通信距離や通信量が大きな問題となる可能性を持っている．

この 2 つの方式は，実際に幾つかの応用問題に適用されて相当の台数効果をあげているが，いずれの方式においても均等化に相当することは特に行っていない．

[1] MLB 方式では仕事を供給するプロセッサも他のプロセッサからの仕事の要求がないときには仕事を実行する．

3.2.2 LLS 方式

動的負荷浸透方式 (Dynamic Local Load Spreading Method: LLS 方式) [17] は筆者らが提案を行った，均等化メカニズムを含む動的かつ『要求駆動型でない』負荷分散方式である．

LLS 方式は，マルチプロセッサを構成する各プロセッサがおのおのの隣接するプロセッサ群 (これを「近傍」と呼ぶ) からの「忙しさ」の報告を受け，近傍の平均的な忙しさに対する自プロセッサの忙しさを動的に計算し，その結果に基づいて仕事の分配を行う方式である．仕事の分配は自プロセッサが近傍内で「比較的忙しい」と判断した場合に起こり，分配量は近傍の各プロセッサの「忙しさ」ができるだけ均等になるよう決定される．すべてのプロセッサが自分を中心とする近傍に対して同様の処理を行うことにより，プロセッサ全体の仕事量の均等化を行うことができる．

LLS 方式でいう「忙しさ」とはプロセッサがある個数 (ジョブキューにセンチネルを挟んで数えることからこれをセンチネル個数と呼んでいる) の部分問題の実行に費やす時間，すなわち,「センチネル個数の仕事の実行にどのくらい時間を要したか」を負荷均等化のための基準情報として扱っている．この基準情報は近傍からおのおののプロセッサの基準で自発的に報告されるので，この「時間」の正規化を報告された側のプロセッサの仕事の個数を用いて行う．これにより，本方式では，タイマが不要であると同時に，基準情報のデータ量が最小限に抑えられる[2]．

また，センチネル個数は，均等化の度に近傍での平均の忙しさに合わせられるので，近傍の忙しさに応じて均等化の間隔が自動的に調節されるという利点があるが，一方，センチネル個数の値によっては，基準情報の報告を行う回数が非常に増える可能性を持っており，同時に均等化の回数も増えるため，負荷分散オーバヘッドが大きくなるという問題点を持っている．

[2]実際にこの情報は「今センチネル個数の仕事を完了した」という通知のみでよく，ハードウェア上で信号線を一本使用するだけで実現できる，という利点がある．

3.3 世代別動的負荷分散方式 (LLS-G 方式)

MLB 方式と STB 方式ではパラメータが最適に調整された場合，性能差はほとんど見られず (64 台以下の場合) 線形に近い台数効果が得られることが報告されている [16]．一方，LLS 方式ではその性能は STB 方式の約 1/2 であり [17]，その原因として大きく以下の 3 つの要素が考えられる．

1. 均等化に用いる基準情報の報告を行う間隔が細かすぎ，オーバヘッドが大きい．
2. 均等化に用いる基準情報が不十分である．
3. 均等化の間隔が細かすぎ，仕事のたらいまわしが起こりやすくなっている．

そこで，これらの要素について考察を行い，世代別動的負荷分散方式 (Dynamic Local Load Spreading Method-Generation: LLS-G 方式) を新たに提案する．

3.3.1 基準情報の報告と均等化の間隔

LLS 方式では，基準情報報告の間隔は各プロセッサに設定されたセンチネル値に従う[3]．このセンチネル値の変更は近傍における各プロセッサの「忙しさ」が異なっている場合にのみ起こるので，たまたま小さなセンチネル値で均等化が行われてしまうと，それ以降，各プロセッサに十分な量の仕事がありかつ均等化された状態においても，(不要な) 基準情報の報告を細かい間隔で実施してしまう．また，均等化の間隔も基準情報の報告と同様に各プロセッサのセンチネル値に従うので，小さなセンチネル値で一度均等化に成功した場合には，以降不必要な均等化処理を行う回数が増えることになる．

一般にプログラムの実行は，負荷の均等化の立場から見たとき次の 3 つのフェーズに分けることができる．

[3]基準情報の報告はセンチネル個数の仕事の実行が完了したときに行われる．

プログラムの起動時: できるだけ早く問題を分散するために細かい間隔で均等化を実施する.

各プロセッサに十分たくさんの仕事があり，均等化がある程度できている時: 細かすぎないある程度の実施間隔を維持する.

プログラムの終了時: 全てのプロセッサができるだけ同時に終了できるように，細かい間隔で均等化を実施する.

このようなプログラム全体の実行状況は，問題の (例えば，OR 並列型探索問題を対象とするならば，探索木の) 形状からある程度予測することが可能である．つまり「問題 (探索木の) 形状」は，各深さにおける部分問題群がその子供として生成した部分問題群の総数に基づいた関数で表すことができるので，各深さにおける部分問題群ごとの均等化を実現することにより，プログラム全体の実行状況 (すなわち，現在起動時か終了時かあるいは中間か) をある程度考慮した均等化が可能である.

LLS 方式では，基準情報の報告と均等化の間隔およびそれらの回数については，センチネル個数を用いて制御することで仕事量の増減に関し追従性のよい値が得られると考えていた．しかし実際には基準情報の報告にかかるコストが予想外に大きいため，このコストを削減する必要がでてきた．そこで LLS-G 方式では LLS 方式のセンチネルを廃止し，基準情報の報告および均等化を「世代」別に行うことにした.

ここでいう「世代」は，探索木でいえばルートからの深さである．LLS-G 方式での均等化は各「世代」の実行が終了した時点で行う．このため，均等化は世代で一度しか行わない．したがって，ある世代の均等化はその親世代の基準情報を用いて行われ，次の均等化のタイミングはその世代の部分問題の量に依存する．図 3.1 に仕事量と負荷の均等化の間隔の例を示す．1 つの短冊は 1 世代を表し，短冊の横幅は均等化の間隔を表しており，仕事量が十分にある場合には均等化の間隔は大きくなる．このように世代ごとの仕事量の増減に応じて自動的に均等化の間隔を調整することで，ほぼ最適な均等化タイミングを得ることが可能である.

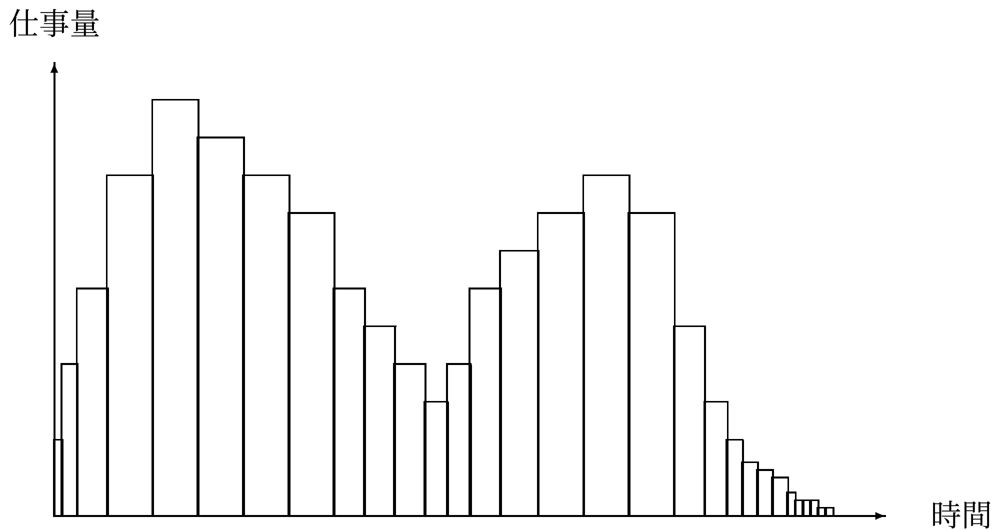


図 3.1: 仕事量と負荷の均等化の間隔

3.3.2 基準情報

LLS 方式では，基準情報の報告の頻度が高いため，できるだけ情報の内容を軽減しておく必要があった．LLS 方式での基準情報の内容は,『今，センチネル個数の仕事の実行が終わった』という信号のみであり，それがどのくらいの時間間隔と評価されるかは，その信号を受け取ったプロセッサの側にまかされていた．

一方，LLS-G 方式では，基準情報の報告回数が必要最小限に留められているため，もう少し詳細な情報の報告を行うことができる．しかし，センチネルを廃止したため，近傍での比較の基準となる数値を新たに設定する必要がでてきた．そこで LLS-G 方式では，次のような情報を用いた基準情報を算出する[4]．

- 親世代が実行した仕事の個数 (N_p)
- 親世代の仕事の実行を開始した時刻 (t_0)

[4]時刻を計測するために各プロセッサが独立したタイマを持つことを前提としている．

- 親世代の仕事の実行を終了した時刻 (t_1)
- 親世代の実行により生成された子世代の仕事の個数 (N_c)

これらの情報から，近傍のプロセッサ群における子世代の仕事の実行 (孫世代の生成) にかかると予測される時間 (T_g) を算出することができる．

$$T_g = (t_1 - t_0) \times \frac{N_c}{N_p}$$

LLS-G 方式における均等化は，ここで求められる T_g を基準情報とし，この値を近傍で平均化することで行う．

3.3.3 分配量の決定

基準情報 T_g は，子世代の仕事の実行にかかると予測される時間を示している．この値はプロセッサごとに異なっており，この値を各プロセッサの忙しさと定義することで，近傍の忙しさ (T_{ave}) は以下のように算出できる．

$$T_{ave} = \frac{1}{n}\sum_{i=1}^{n} T_{gi}$$

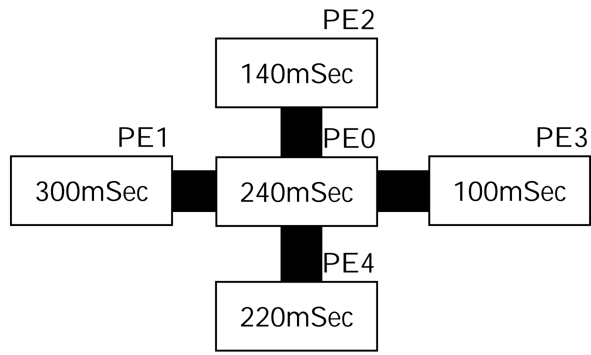
ただし n は近傍に属するプロセッサの台数である．

例えば，図 3.2(1) において，近傍の忙しさ T_{ave} は，

$$(240 + 300 + 140 + 100 + 220) \div 5 = 200$$

となる．この T_{ave} を用いて各プロセッサの忙しさを正規化すると，図 3.2(2) に示す値が得られる．これが近傍における各プロセッサの相対的な忙しさを表している．

図 3.2(2) でわかるように，PE0(プロセッサ 0) を中心とする近傍の忙しさ 100%に対し PE0 の相対的な忙しさは 120%であるため，PE0 は「近傍の他のプロセッサより 20%余計に仕事を抱えている」と考えることができる．このとき PE0 が分配を行



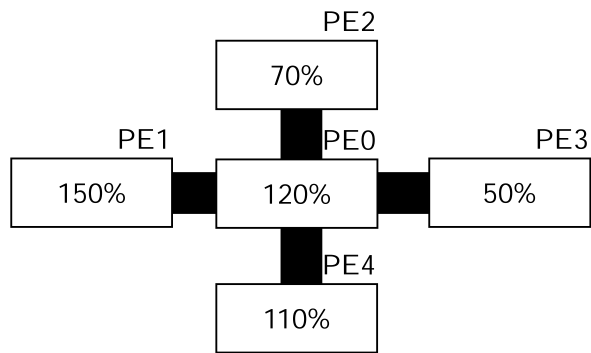


図 3.2: 近傍の忙しさ

う総仕事量 (N_{dist}) は，その近傍において親世代と同じペースで仕事の実行がなされた場合に，現在 PE0 が持っている子世代の仕事の個数 (N_c) のうち他のプロセッサと同一時間内には実行しきれないと予測される仕事量である．この値は次の計算式により，簡単に算出できる．

$$N_{dist} = N_c \times \frac{T_g - T_{ave}}{T_g} \qquad (\text{ただし}\ T_g > T_{ave})$$

この余剰すると予測される仕事量を平均よりも負荷が軽い (と予測される) プロセッサに対しておのおのの忙しさに応じて比例分配することで，子世代の仕事量の均等化を図る．例えば図 3.2 の例では，PE2 と PE3 に対して，おのおの

$$\begin{aligned} PE2 &: N_{dist} \times \tfrac{100-70}{(100-70)+(100-50)} = N_{dist} \times \tfrac{3}{8} \\ PE3 &: N_{dist} \times \tfrac{100-50}{(100-70)+(100-50)} = N_{dist} \times \tfrac{5}{8} \end{aligned}$$

ずつ仕事を分配することで次世代の仕事量の均等化を図ることができる．

一方，どのような規模（プロセッサ数）の並列処理環境においても，ある一定時間内ですべてのプロセッサに仕事を拡散できるようにすることを考えると，システム半径が大きくなり拡散を『早めたい』場合には，仕事の粘性 (拡散のしにくさ) を下げてやる必要がある．この粘性 D はシステム半径に応じて設定されるべきハードウェア上のパラメータであり，0 から 1 の間の値をとると考えている．近傍の忙しさの基準値 T_{ave} は，実際にはこの粘性 D を乗じて 100%よりも小さな値として計算されるので，分配すべき仕事量 N_{dist} も多めに計算されることになる．これは，1 ホップよりも遠隔にあるプロセッサ群の忙しさを隣接するプロセッサと同様であると予測していることにほかならない．

3.4 負荷分散オーバヘッドの評価

3.4.1 評価項目

LLS-G 方式を並列推論マシン PIM/m[18] 上に並列論理型言語 KL1 を用いて実装した．ここでは，LLS-G 方式が STB 方式に比べて，どの程度，負荷分散オーバヘッ

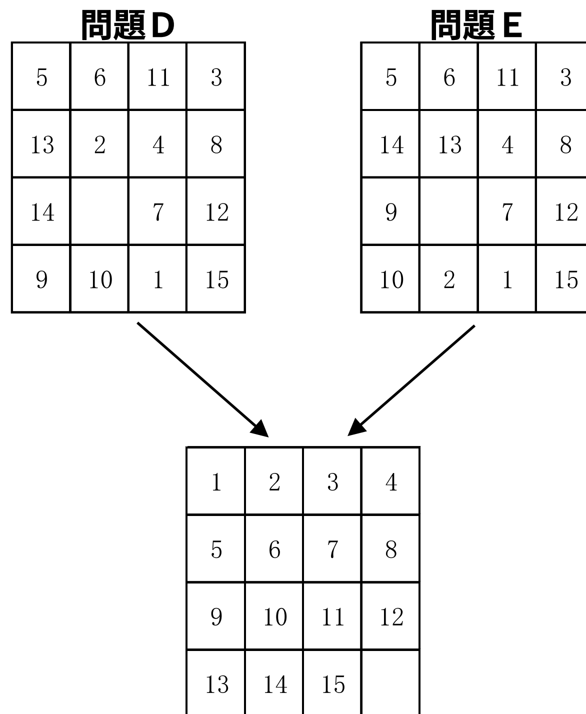


図 3.3: 15 パズル (問題 D，問題 E)

ドが軽減しているかを評価する．対象とした問題は，Iterative-Deepening A*アルゴリズムに基づく 15 パズル問題 [19] である．この種の問題及び並列分割形態では，並列に動作する個々の処理同士で通信がないため，どの処理をどのプロセッサに割り当てるかによる実行時間の違いはほとんど存在しない．また，実行の初期段階と終焉段階を除いて，中間段階においては各プロセッサに充分負荷が割り付けられているため，負荷分散方式の違いによる負荷のアンバランスは起き難く，負荷分散オーバヘッドの違いが全体の実行時間に反映されると考える．従って，ここでの負荷分散オーバヘッドの評価では，まずは実行時間及び台数効果をベースとした比較で行うことにする．

3.4.2 計測

ここでは，ある１つの初期状態から最終状態を導く 15 パズルについて文献 [19] に掲載された 2 種類の問題 (問題 D および E) を用いて計測を行った (図 3.3)．15 パズルは，15 枚のタイルのある状態から次にどのタイルを動かすかにより，幾つかの独立した状態を作るので，基本的な OR 並列型探索問題として考えることができる．

ここで計測に用いた問題 D および E は，タイルを 1 枚動かすことを 1 世代として，どの 1 枚を動かすかによって子世代にあたる異なった状態を作ると仮定した時，それぞれ，

問題 D 総状態数約 150 万，世代数 38，最大状態数約 17 万 (第 20 世代)

問題 E 総状態数約 590 万，世代数 42，最大状態数約 62 万 (第 23 世代)

となり，いずれの場合も，各世代の状態数の推移は最大状態数を持つ世代を中心とした山なりの形状を示す．

ユーザプログラムと LLS-G 方式を用いた負荷バランサの基本的な動作は，以下の通りである．

- ユーザプログラムは実行を 1 世代進める，すなわち，本 15 パズルの場合は，ある状態を与えられると，その続きとなる全ての「タイルを 1 枚動かした状態」を作って返す．
- LLS-G 方式を用いた負荷バランサは，各プロセッサごとにユーザプログラムを用いてある世代を得て，各世代ごとに他のプロセッサと負荷分散を行う．

本負荷バランサを他の問題に対して適用するユーザは,「1 世代」を定義し,「ある状態から 1 世代進めた状態を生成する」特定の名前の述語を設けるだけで良い[5]．前述

[5] このような負荷バランサとユーザプログラムの切り分けは，STB 方式 [16] で用いられているものである．

表 3.1: LLS-G と STB の実行時間

(単位：ミリ秒)

プロセッサ数	16	32	64	96	128	160	192	224	256
問題 D									
LLS-G	136,140	69,731	34,807	24,473	20,339	17,137	15,181	14,099	12,919
STB	121,594	63,143	33,243	24,297	19,394	17,265	16,258	15,400	15,276
問題 E									
LLS-G	539,895	271,696	151,238	98,405	75,227	58,863	49,865	44,594	38,710
STB	469,451	236,317	121,390	82,110	65,055	54,681	47,849	43,518	39,821

した各問題の「世代数」は探索木のルートからの距離を示す値であり，いわば,「論理的な世代」である．これに対し，実装上は各プロセッサにおける世代ごとの実行は同期せずに進むため，並列実行した場合の均等化の回数は世代数を超える数値となるし，各プロセッサが 1 世代と見なす仕事は，異なった世代の仕事の混在状態となる．

表 3.1 に各問題の実行時間を，また，16 台版の実行時間を基準とした台数効果を図 3.4 に示す.

表 3.1 からわかるように，マシンを構成するプロセッサ台数 (以下,「構成台数」と呼ぶ) が少ない場合には，LLS-G 方式は STB 方式より実行時間が遅いが，問題 D では 160 台以上の場合に，また，問題 E では 256 台の場合に実行時間の逆転が起こっている．このことから，構成台数がある程度多い場合には，LLS-G 方式の方が STB 方式よりも速くなる傾向にあると考えられる．また，図 3.4 で問題 D の台数効果をみると，STB 方式では構成台数が増えるとともに台数効果が頭打ちとなっているのに対し，LLS-G 方式では傾斜が緩やかになってはいるが，依然として上昇傾向を示していることがわかる．

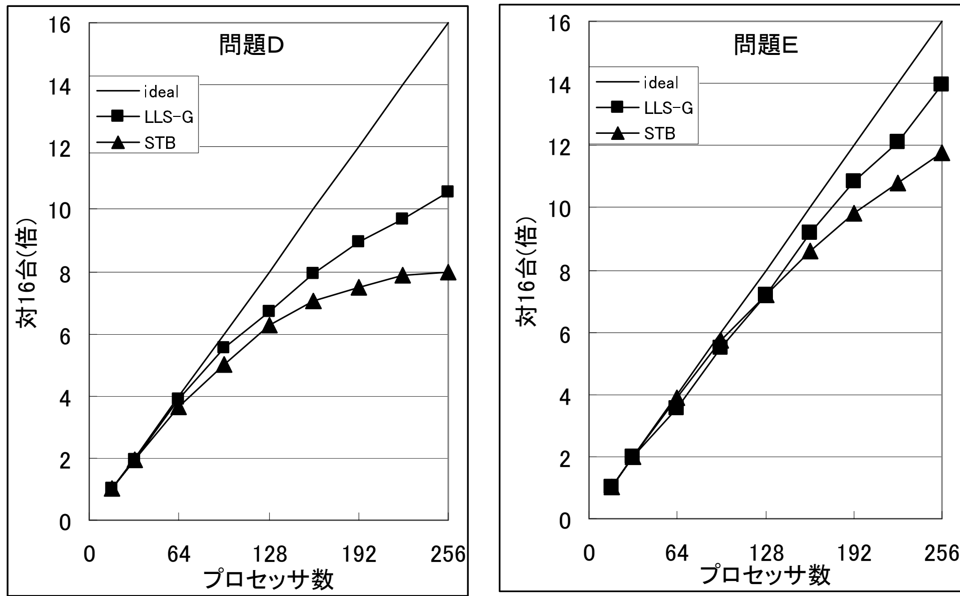


図 3.4: LLS-G と STB の台数効果

3.4.3 考察

LLS 方式の長所は，構成台数の増加による影響を受けにくいことにある．例えば，分散のコストを「仕事を貰う 1 回あたりのコスト」として構成台数の増加について考える．

LLS 方式では，1 台あたりの分散コストが構成台数の増加によらず各プロセッサが近傍と接続する手の本数で抑えられる．一方，STB 方式では分散のための論理的なネットワークが完全結合である必要があり，そのコストは (プロセッサ総数×システムの半径) のオーダになると考えられる．この 2 つのコストが交差する点 (点 P とする) が実際に構成台数が何台のところにあるかを知ることが，本研究における 1 つの大きな課題であった．

PIM/m 上の LLS 方式では，少なくとも 128 プロセッサまででではこの 2 つのコストが交差しないように思われた [17]．これは STB 方式が要求駆動型であり，1 回の

表 3.2: LLS-G と STB のメッセージ数

(単位：× 1,000 メッセージ)

プロセッサ数	4	8	16	32	64	96	128	160	192	224	256
問題 D											
LLS-G	1,387	2,073	2,026	2,976	2,915	3,135	3,482	3,928	3,842	4,264	4,737
STB	490	379	340	1,237	1,337	2,626	3,994	6,227	7,894	14,510	23,518
問題 E											
LLS-G	9,001	7,216	9,072	10,303	12,869	11,115	12,026	12,968	13,303	13,782	14,462
STB	1,667	1,136	1,266	1,745	3,890	5,751	7,451	8,729	11,251	14,868	18,938

負荷分散のためのメッセージ数が 1〜2 に抑えられているのに対し，LLS 方式では実行時間に依存する関数になってしまうため，構成台数に応じた大きな問題を扱い実行時間が伸びるほど負荷分散コストを引き上げているためと考えられた．そこでこのメッセージ数の削減を行った LLS-G 方式の提案を行った．

表 3.2 に，LLS-G 方式および STB 方式で実行時に実際に受信された全ての通信メッセージの数を示す．数値を見ただけでも，明らかに構成台数に対する増加率が異なっていることがわかる．

図 3.5 は，表 3.2 のメッセージ数を基に，1 ミリ秒当たりに 1 台のプロセッサが受信したメッセージの数を示したものである．特に問題 D の場合に LLS-G 方式と STB 方式の差が顕著である．STB 方式では構成台数に対してメッセージ数が線形程度かまたはそれ以上に伸びているのに対し，LLS-G 方式ではほぼ一定の値を保っているので，2 つのコストが逆転する点 P が，問題 D では 128 台のところに，問題 E では 224 台のところにあることが分かる．すなわち，LLS-G 方式では各プロセッサが分散に費やすメッセージ数は構成台数によらず一定であり，分散コスト $\simeq$ メッセージ数と考えると，より大規模な並列処理環境向きの動的負荷分散方式である，と言うことができる．実行時間の観点から見ても，分散コストの点からは明らかに STB 方式が有利である少ない構成台数の場合にも，LLS-G 方式では最悪 1.25 倍程度の実行時間で抑えることができ，かつ構成台数がある程度を超えると逆に実行時間が速くなる (256 台で 1.18 倍) ことから，構成台数によらず LLS-G 方式を用いる利点は十分にあると思われる．

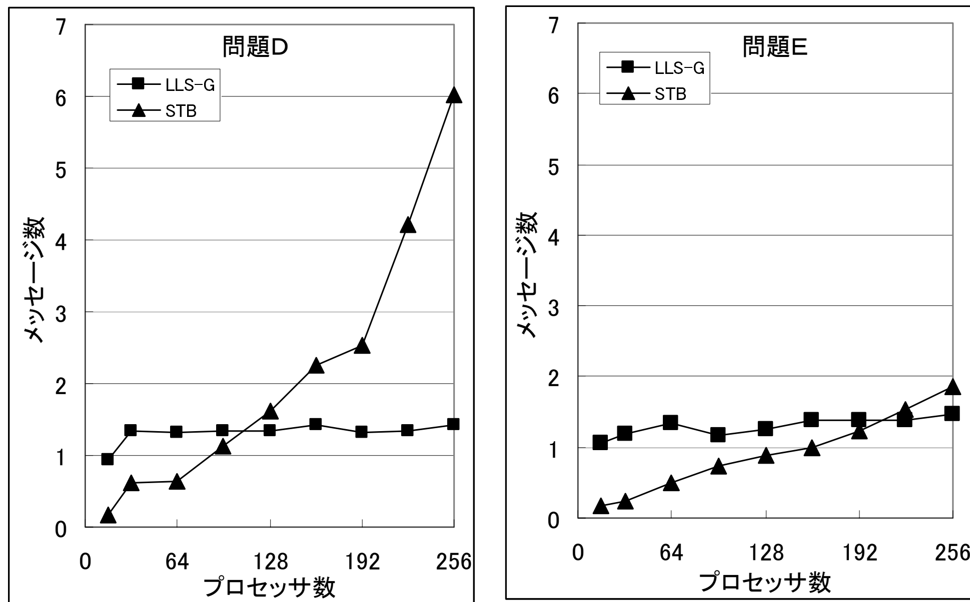


図 3.5: LLS-G と STB の通信メッセージ数

また，図 3.4，図 3.5 に共通の傾向として，問題 E に比べ問題 D の結果に STB 方式と LLS-G 方式の差が顕われやすい．この理由として，問題サイズ (これは 15 パズルの場合，探索木上の総状態数を指す) が考えられる．ある問題を並列実行する場合に，その実行時間を支配するのは「プロセッサあたりの仕事量[6]」と「負荷分散オーバヘッド」である．プロセッサ当たりの仕事量が少ない問題では，多い問題に比べて負荷分散オーバヘッドが支配項になりやすく，仕事の分散による実行時間の短縮の効果が現われにくい．

図 3.4 の問題 D における台数効果で，STB 方式の台数効果が LLS-G 方式の台数効果に比べて早く飽和するのは，プロセッサ当たりの仕事量と負荷分散オーバヘッドの実行時間に占める割合の逆転が，比較的少ない台数で起こるためであると考えられる．これは図 3.5 の STB 方式による問題 D の通信メッセージ数の増加率が LLS-G 方式に比べてかなり大きいことからも予測できる．これに対し問題 E では，プロセッサ当たりの仕事量が比較的多いため，仕事量と負荷分散オーバヘッドの割合の逆転

[6] ここでは，世代ごとに実行する仕事量を構成台数で除したものと考えられる．

が起こりにくく，基本的に高い台数効果を得やすい．このため負荷分散オーバヘッドの大小によらず，比較的良い台数効果を維持できると考えられる．

また，LLS-G 方式に関して各問題の台数効果と負荷分散オーバヘッドを見た時，通信メッセージ数がほぼ一定であるにもかかわらず台数効果の低下が見られる．これはプロセッサの稼働率の低下によるものであり,「負荷の分散状態が理想的な均質状態からどのくらい遠いか」を示しているものと考えられる．

別途，パフォーマンス・メータと呼ぶプロセッサの稼働率モニタ [20] により各プロセッサの稼働率を見てみると，実行の終了に近づいた時には，遊休プロセッサと稼働率 100%のプロセッサが同時に存在していた．このような状況に，完全な線形な台数効果を得ることのできない理由の一端があり，さらに理想的に負荷が均等化された状態が存在する可能性は十分に考えられる．より精度の高い立ち上がり，高い立ち下がりを得るためには，マシンごとの粘度 D を定式化するための尺度を確定し,「仕事の拡散しにくさ」を正確に定義可能とする必要がある．

いずれにしろ，考察を行ってきたように，LLS-G 方式は LLS 方式だけでなく STB 方式のような従来方式と比較しても，分散メモリ型の大規模な並列処理環境において，かなり有望な動的負荷分散方式であると言える．

また，計測に用いた並列推論マシン PIM/m においては，ネットワークを渡るメッセージの送受信に要する時間のうち，通信パケットのエンコード/デコードに要する時間がかなりの部分を占めており [21]，ネットワーク・ハードウェア上のホップ数の影響がほとんど現われない．換言すれば，メッセージの送受信時間に対してシステム半径が支配項となるようなタイプの大規模並列処理環境においては，局所情報のみにより負荷分散を行う LLS-G 方式の効果は一層期待できると考えられる．

3.5　結言

本章では，OR 並列型探索問題を主対象とした「世代別動的負荷浸透方式 (LLS-G 方式)」について述べ，並列推論マシン上でその方式の負荷分散オーバヘッドを評価した．

拡散型の動的負荷分散方式は，元来その負荷分散オーバヘッドの大きさから実現が難しいと考えられていたが,「世代」概念の導入により，より大規模な並列処理環境に対して有効な方式であることが実証できたと考えている．

「世代」という概念を導入するにあたり，対象問題を OR 並列型探索問題とすることで，問題に対する世代の効果の解析を行いやすくなった.「世代」は，元々このような問題自身の性質とは異なった概念である．しかし，OR 並列型でない問題，例えばデータ並列型の問題に対してもデータを適当に分割して，それぞれの分割領域を「世代」として扱う，などのように「世代」を想定したプログラミングは可能であり，本方式を適用することができると思われる．ただし，本方式では負荷均等化の精度が「親世代」と「子世代」の負荷量 (T_g) の増減の類似性に依存しているので，世代ごとに常に増加傾向と減少傾向が切り換わるような問題があるとすると,「負荷の予測」が外れ続け，仕事のたらいまわしが起きるだけで実行が全くなされない，ということも考えられる．

本方式は，大域的な情報を用いず，隣接するプロセッサの情報しか用いていないため，プロセッサ間同士に距離の違いが存在するような環境，すなわち，近くのプロセッサ間同士と遠くのプロセッサ間同士の通信時間がかなり異なるような並列処理環境において有効である．例えば，プロセッサ同士が物理的に離れた場所に存在するグリッド・コンピューティングのような形態に適しており，今後はそのような環境が増えることが予想されるため，本方式の適用可能範囲は増えると思われる．

第4章　効率的な並列性制御を行なうための言語サポート

4.1　緒言

並列処理環境においてプログラミングする上で重要な問題の１つに，既存プログラムからの継続性が挙げられる．これまで，逐次処理環境におけるプログラム財産は膨大なものがあり，並列処理環境においてもそれらをできるだけ利用したい．また，近年の並列処理環境は，プロセッサと汎用ネットワークの目覚しい性能向上を利用して，パソコンなどで利用される汎用のプロセッサをネットワークで接続したクラスタの形態を採っているものが多い．その意味でも，既存の（単一）プロセッサで効率良く動作するプログラムというのが重要となる．そのためには，単一プロセッサで効率良く動作する既存の逐次処理用のプログラミング言語を用い，それに最小限の並列性制御のための機能を追加する方法が考えられる．

更に，プログラミング言語レベルで並列性制御機能を提供するためには，単純なデータ並列のような１階層の並列分割形態だけでなく，より複雑な様々な分割形態に対応する必要がある．また，性能チューニング等を考えると，並列性制御を行う部分をできるだけ容易に変更できるのが望ましい．そのためには，その並列性制御部分を並列処理のためにわざわざ設けるのではなく，逐次処理の時から存在する単位をベースに記述できるのが良い．並列処理を行う単位には，OpenMP[22] のようにループをベースとする方法も考えられるが，データアクセスの局所性等の並列実行可能性を考慮すると，オブジェクト指向におけるオブジェクトを並列実行の単位とする方が得策と思われる．

本章では，以上のことを考慮して，オブジェク群単位の並列性制御が容易に行え，更に既存の逐次プログラムからの変更を最小限に抑えた，C++言語をベースとした並列処理言語 $\mathcal{ANUFO}$(A NORMA-base User-Friendly Object-Oriented Language and Programming System) [23] を提案する．

4.2 設計方針

これまでの並列プログラミングにおける問題点を解決すべく，並列プログラミング言語に必要とされる機能を考慮に入れて，以下のような方針に基づいて $\mathcal{ANUFO}$ の言語機能を設計した．

- 並列実行の単位を明確に記述できる
  並列プログラミング言語では，何らかの「並列に実行される単位」を記述できなければならない．この単位では，変数などへのアクセスの局所性を記述できるべきである．

  オブジェクト指向は生産性/保守性に優れており，様々な分野で広く採り入れられている．オブジェクトの内部状態は，オブジェクトの外から直接アクセスできないというデータアクセスの局所性があるため，オブジェクト単位での並列実行に適している．そこで $\mathcal{ANUFO}$ では，オブジェクトをベースに並列実行の単位を記述できるようにしている．

- 並列実行の粒度を容易に変更できる
  並列処理においては，使用する各プロセッサをできるだけ有効に利用できるようにするために，並列実行の単位の大きさ (粒度) を調整することがしばしばある．従って，粒度の変更はできるだけ容易にできなければならない．

  そのため $\mathcal{ANUFO}$ では，並列実行のための指示を最小限の記述で可能にしている．

- 既存 (逐次) 言語の延長として記述できる

並列処理のために新たなプログラミング言語やプログラミング手法を習得するのは大変である．そこで，並列プログラミング言語は既存の逐次型言語の最小限の拡張とし，これまでのプログラミング手法などをできるだけ流用できるようにすべきである．

そこで *ANUFO* では，現在オブジェクト指向言語として最も多く利用されている C++言語 [24] をベースにして，最小限の範囲で並列処理に必要な通信/同期機能を拡張している．これにより，逐次に処理されるようなオブジェクト内のプログラムは，既存の逐次プログラムをそのまま流用できる．

- 多くのシステムで利用できる
  これまでの並列処理記述には，ある並列マシン特有の機能を利用したものが多く見られた．*ANUFO* では，オブジェクト指向機能を用いて並列処理を記述するようにしたので，ある特定の並列マシンの機能に依存するような記述は不要である．従って，多くのタイプのマシンで変更することなしに動作可能な，すなわち，互換性のあるプログラムを記述することが可能である．

  また，このような互換性を保つために，*ANUFO* 処理系は特定のシステムの機能を使用しないで実装する必要があり，多くの場合システムに依存してしまうような通信機能やコンパイラは，多くのシステムをサポートしているフリーソフトウェアを利用することで，可搬性を向上させている．

  ANUFO プログラムは，トランスレータにより，通信/同期機能を提供する実行時ライブラリ呼び出しを伴う C++プログラムに変換され，Gnu C++コンパイラ (`g++`)[25] でコンパイルされる．従って，継承などのオブジェクト指向機能に対する処理は，C++コンパイラ及びそれに付随した C++ライブラリ (`libg++`) に完全に任せることができる．

4.3 言語仕様

ANUFO は，C++をベースとして，並列プログラミングに必要な並列実行の単位 (本章ではこの単位をスレッドと呼ぶ) の生成及びプロセッサへの割り付け機能, スレッド間の通信機能及び同期機構などが拡張されている．

4.3.1 リモート・オブジェクト

前述したように *ANUFO* における並列実行の単位 (スレッド) は，オブジェクトをベースとしている．これは，オブジェクト単位で変数 (メモリ) アクセスなどの局所性が存在し，オブジェクト単位で並列処理が適しているからである．

1 つのオブジェクトは，部品として複数の他オブジェクトを保持するなど，複数オブジェクトから構成されていたり，1 つのメンバ関数 (メソッド) が複数オブジェクトへのアクセスで処理されたりする場合が多々ある．つまり，オブジェクトには階層性があり，一言でオブジェクトと言ってもさまざまなサイズ (粒度) が考えられる．

そのため，1 つ 1 つのオブジェクトをスレッドとしてしまうと，この階層上の最も下層を単位とすることになる．しかし，並列処理環境によっては，CPU 性能対通信性能の比率から，個々のオブジェクトをスレッドとするには小さ過ぎる場合がある．従って，オブジェクト階層の任意のレベルで並列/逐次の区切りを指定できるようにすべきである．

そこで *ANUFO* では，オブジェクトに，通常の (C++) オブジェクトとリモート・オブジェクトを設けている．図 4.1 で,「スレッド 1」内のオブジェクトから見て,「スレッド 2」内のオブジェクトはリモートであり，その逆もリモートである．各スレッドの実行は逐次に行われ，それぞれのスレッドの実行は独立して並列に行われる．

スレッドの生成は，リモート・オブジェクトの生成により，暗黙に行われる．スレッドはあくまで実行環境であり，プログラム上の記述には存在しない．

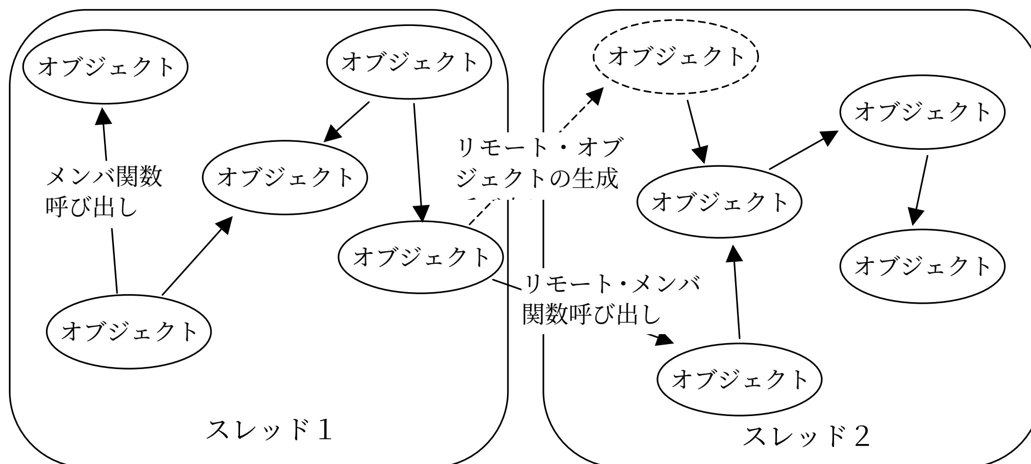


図 4.1: オブジェクトとリモート・オブジェクト

図 4.2 にプログラム例を示す．変数宣言時 (図中 16,17 行) 及びオブジェクトの生成時 (18,20 行) にクラス名の左にキーワード `remote` を付けることで，リモート・オブジェクトであることを指示する．20 行の `@i` は，リモート・オブジェクトを生成するプロセッサ番号[1]を `i` で指定している．

リモート・オブジェクトへのメンバ関数呼び出し (21 行) をリモート・メンバ関数呼び出しと呼ぶ．リモート・メンバ関数呼び出しはスレッド間の通信となるが，そのシンタックスは C++と全く同じである．従って，並列粒度の調整では，変数宣言時及びオブジェクト生成時に `remote` を付けるかどうかだけで行える．

4.3.2 同期機構

ANUFO の同期機構には，リモート・メンバ関数の実行待ち合わせによる方法と変数アクセスによる方法の 2 種類がある [28].

[1]*ANUFO* でプロセッサ指定は，全て番号 (整数値) で指定する．

```
 1 class Producer {
 2 public:
 3   Job* get_job();
 4 };
 5 class Consumer {
 6   void do_job(job*);
 7 public:
 8   void go(remote Producer* p) {
 9     job* j;
10     while( j=p->get_job() ) { // 同期型
11       do_job(j);
12     }
13   }
14 };
15 main() {
16   remote Producer* prod;      // 変数宣言
17   remote Consumer* cons;      // 変数宣言
18   prod = new remote Producer; // 生成
19   for( int i=0; i<N; i++ ) {
20     cons = new remote Consumer@i; // 生成
21     cons -> go(prod);   // 非同期呼び出し
22   }
23 }
```

図 4.2: $\mathcal{ANUFO}$ のプログラム例

4.3.2.1 関数呼び出しによる同期

リモート・オブジェクトへのメンバ関数呼び出しは，原則として非同期に実行される．つまり，リモートのメンバ関数の実行は，呼び出し側の実行と並列に実行される．21 行のリモート・メンバ関数呼び出しは main とは非同期に実行されるので，結果的に N 個の go(prod) が並列に実行されることになる．

また，同期呼び出し (リモート・メンバ関数の実行が終了するまで，呼び出し側の実行を中断すること) も可能である．

以上まとめると，以下のようになる．

非同期型

- 返り値を使用しない呼び出し
  例: `robj -> bar(...);`
  `func(...);`
- 返り値を同期機能付き変数 (後述) に代入する呼び出し
  例: `sync int *X = robj -> bar(...);`
  `func(...);`

同期型

- 返り値を使用する呼び出し
  例: `X = robj -> bar(...);`
  `func(...);`
- キーワード wait を前に付けた呼び出し
  例: `wait robj -> bar(...);`
  `func(...);`

bar(...) の全ての実行が終了するまで

(`bar(...)` 内から更にリモート・メンバ関数呼び出しがある場合も，それらの実行が全て終了するまで)，次の statement(`func(...)`) の実行に進まない．

この `wait` 機能により実行終了の検知ができるので，自らプログラムのスケジュールをするようなメタなプログラミングが可能となる．

4.3.2.2　同期機能付き変数

同期変数とは，未定義状態と定義状態を設け，未定義状態への読み込みは定義状態になるまで実行が停止するという機能を持った変数であり，sync 変数、stream 変数の 2 つである．

sync 変数

write once/multiple read 型の同期変数を sync 変数と呼ぶ．変数には未定義/定義の 2 つの状態があり，未定義の状態での読み出しは書き込みが起きる (これにより定義状態になる) まで中断し，定義状態での書き込みはエラーになる．それ以降の読み出しに対しては，常に同じ値を返す．sync 変数はいわゆる I-Structures[29] であるが，$\mathcal{ANUFO}$ では構造体ではなく，int などのアトミックなデータにもこの機能を取り入れている点が異なる．

sync 変数は，ある 1 人の生産者がデータを生成しそのデータを消費者が必要とする場合に，消費者では単に読み出すという記述だけで，データの生成が完了するまで待つことができる．

図 4.3 は，sync 変数の書き込み/読み出しの記述例とその実行手順である．

1. sync 変数 `var` を `sync int *var` で宣言し，`var = new sync int` で生成する．この時の sync 変数の生成はデフォルトプロセッサ 0(PE0) 上になされる．

```
class obj {
  int Y;
public :
  void f(sync int* X){   //sync 変数の受け取り
    Y = *X;              //var の読み出し要求が発生
    cout << Y;
        :
  }
};

main() {
  sync int *var;
  remote obj *A;
  A = new remote obj@PE1  //PE1 に並列オブジェクト obj を生成
  var = new sync int;     //sync 変数 var を生成
  A->f(var);              //sync 変数 var をオブジェクト A に渡す
      :
  *var = 1;               //var に 1 が書き込まれる
}
```

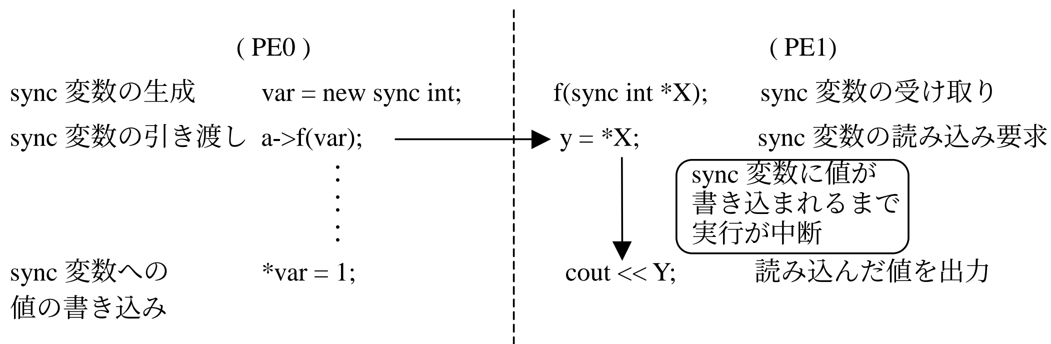


図 4.3: sync 変数の記述例 1 (書き込み/読み出し)

```
sync int *W, *X, *Y, *Z;
            //W,X,Y,Z は sync 変数
int V;
   :        //sync 変数の生成など
*W = 1;     //W に 1 を書き込む
merge(W,X); //W(定義状態) × X(未定義状態)
merge(Y,Z); //Y(未定義状態) × Z(未定義状態)
merge(X,Z); //X(定義状態) × Z(Y を参照)
V = *Y;     //V = 1 : Y を V に読み込む
```

図 4.4: sync 変数の記述例 2 (マージ)

2. リモート・オブジェクト obj が PE1 上に生成され，Y = *X で sync 変数 X(すなわち var) への読み出し要求が発生する．その時点でまだ PE0 で var が書き込まれていなければ，PE1 の読み出し要求は中断する．
3. PE0 で*var = 1 により sync 変数 var に 1 が書き込まれると，PE1 の読み出し要求が動作し始め，X に値 1 が読み出される．

これは共有した 1 つの sync 変数を用いる場合である．一方，これに対し並列実行される複数のスレッドにおいて，それぞれのスレッドで生成された sync 変数が，処理の途中で同一の同期タイミングであることが判明し，それらの sync 変数を同一の変数として扱いたくなる場合がある．このような場合のために，$\mathcal{ANUFO}$ では、マージと呼ばれる生成済みの 2 つの sync 変数を同一の変数として取り扱えるように変換する操作を提供している．ある 2 つの sync 変数に対し，マージ操作を行なうと，一方の変数に書き込んだデータをもう一方の変数から読み出すことができる．但し，2 変数が共に定義状態の場合は，たとえ同じ値を持っていてもマージできない．

図 4.4 は，sync 変数のマージの記述例である．2 つの sync 変数 X と Y のマージは merge(X,Y) と記述する．W に 1 を書き込んで，さらに W と X，Y と Z，X と Z をマージすることにより，W,X,Y,Z のどの変数からも 1 を読み出すことができる．

stream 変数

データを複数回書き込みでき，書き込まれた順に FIFO でデータが読める同期変数を stream 変数と呼ぶ．stream 変数に対する読み出しは，データがない場合に中断する．

stream 変数と sync 変数の機能の違いは，sync 変数への書き込みは 1 度だけで，その後は何度読んでも同じ値が読み出されるのに対して，stream 変数に対しては読み出しと同じ回数だけの書き込みが行なえ，(同じ値が書き込まれない限り) 全て異なる値が読み出される点である．

sync 変数は値を一度しか代入できなかったので，同期を取ることが中心であり，変数値そのものにあまり意味がなかったが，stream 変数はデータを何度も代入できるので，同期機構を簡便に記述できるだけでなく，データ通信にも利用できる．

stream 変数に対してもマージの機能を提供しており，sync 変数の場合と同様に，一旦マージされた 2 つの stream 変数はあたかも同一の変数のように振舞う．stream 変数同士のマージは，2 変数が未定義/定義状態すべての場合に可能である．

stream 変数 `X` を `stream int *X` で宣言し，`X = new stream int` で生成する．stream 変数への書き込み/読み出し/マージのシンタックスは sync 変数と同じである．

図 4.5 は，stream 変数の記述例である．コメント文の中では，stream 変数が保持する値の集合を [⋯] で示している．

共に stream 変数である X と Y をマージすることにより，結合集合が生成される．X,Y どちらに対する操作 (読み/書き) も，結合集合に対して行なわれる．一度，読み込まれた変数値は，結合集合から取り出される．従って，次の読み出しでは，集合上の異なる変数値が読める．

```
stream int *X,*Y; // stream変数X,Yを宣言
int Z;
   :              // stream変数の生成など
*X = 11, *X = 12; // Xに11,12を書き込む, X=[11,12]
*Y = 21, *Y = 22; // Yに21,22を書き込む, Y=[21,22]
merge(X,Y);       // XとYをマージ,       X=Y=[11,12,21,22]
*X = 13;          // Xに13を書き込む,     X=Y=[11,12,21,22,13]
*Y = 23;          // Yに23を書き込む,     X=Y=[11,12,21,22,13,23]
Z = *X;           // XをZに読み込む,      Z=11,X=Y=[12,21,22,13,23]
Z = *Y;           // YをZに読み込む,      Z=12,X=Y=[21,22,13,23]
```

図 4.5: stream 変数の記述例

4.3.3 逐次 (排他) 性の保証

並列プログラミングにおいて，あるデータを処理している最中に，そのデータを他スレッドから変更されたくないという状況がしばしば存在し，通常はセマフォなどを用いて排他制御を行う．

ANUFO においては，オブジェクト内の実行が逐次なので，オブジェクト内を実行している最中は，そのオブジェクトの状態を他スレッドにより変更されること及び覗かれることは決してない．従って，上記のような排他制御を容易に記述することができる．ただし，リモート・オブジェクト間の同期型のメンバ関数呼び出しがループしている (A が B を呼び，その中で A を呼んでいる) ような場合，簡単にデッドロックに陥ってしまうので，注意が必要である．これを回避するために，`const` メンバ関数 (オブジェクトの状態を変えないメンバ関数) だけは同時に複数実行可能とする仕様も考えられるが，実装方式の単純化のために導入しないことにした．

また，1 つのスレッドからの同じリモート・オブジェクトへの非同期メンバ関数呼び出しは，リモート側では必ず呼び出された順に実行されることを保証している．これにより，同じリモート・オブジェクトへの複数のメンバ関数呼び出しにおいて，

それぞれ返答を待つ必要がなくなっている．

4.3.4 割り込み機能

$\mathcal{ANUFO}$ では，オブジェクト内の実行が逐次に行われており，その実行中に他の並列に実行されているオブジェクトからリモート・メンバ関数呼び出しが起きても，オブジェクト内の (逐次の) 実行が終了するまで，その関数は起動されない．また，1つのオブジェクトへのリモート・メンバ関数呼び出しが複数行われた場合，それらはキューイングされ，順番に起動される．従って，このような $\mathcal{ANUFO}$ の実行機構では，並列に動作しているリモート・オブジェクトに対して緊急の処理を行いたくても不可能である．そこで，$\mathcal{ANUFO}$ では，現在実行中の処理を中断して，別の処理を割り込ませる機能が提供されている．

$\mathcal{ANUFO}$ では，オブジェクト間でのみ並列実行が可能であり，1つのオブジェクトに同時に複数からアクセスすることはできない．これにより，オブジェクトの状態を複数の処理が同時に変更しないことを保証している．従って，$\mathcal{ANUFO}$ のプログラムでは，(少なくとも) 自オブジェクトの状態は，並列に実行している他のオブジェクトから変更されることはなく，これを利用して排他制御の記述が可能となっている．

通常の割り込み機能では，実行中のプログラムを中断して別の処理を行わせるので，オブジェクトの状態を実行途中で変更できるようになり，これまでのように排他制御ができなくなってしまう．例えば，オブジェクトの状態が2つの変数で保持されているとすると，片方の変数だけを変更し，もう片方はまだ変更していない状態で割り込みが起きると，オブジェクトの正しくない状態を外部に公開してしまうことになる．

そこで、通常の割り込み機能に加え，オブジェクトの状態を正しく保ったまま緊急の処理を行える機能として，キューイングされているメンバ関数呼び出しの要求を全て追い越して，呼び出し要求キューの先頭に入れる (キューに割り込む) 機能も

提供している．

処理中の実行を中断して関数を呼び出す通常の割り込み機能を「緊急関数呼び出し」と呼び，キューの先頭に関数呼び出し要求を入れる機能を「特急関数呼び出し」と呼ぶ．また，これらの呼び出しにより起動される関数を「割り込み関数」と呼ぶ．

4.3.4.1　緊急関数呼び出し要求

リモート・オブジェクトで現在実行中の処理を中断して，割り込み関数を呼び出す．この割り込み関数の実行が終了したら，中断していた (割り込まれる前の) 処理が再開される．この呼び出しは，非同期型呼び出しである．つまり，割り込む (呼び出す) 側と割り込まれる側 (リモート・オブジェクト) は互いに並列 (並行) に動作しており，割り込む側は，割り込まれる側の割り込み関数の実行終了を待たずに，次の処理 (ステートメント) へ進む．

リモート・オブジェクトが何も実行しておらず中断状態の場合は，単に割り込み関数が呼び出されるだけで，その関数の実行が終了したら再び中断状態となる．

割り込み関数は，int 型の引数を 1 つ持ち，その引数によりどのような割り込み処理をするかを指示することができる (割り込み関数の定義方法に関しては後述する)．

[**シンタックス**]

```
リモート・オブジェクト変数 ==> int 型変数
```

シンタックスは，リモート・メンバ関数呼び出しに似ているが，呼び出す関数名は指定せず，引数 (上記の int 型変数) だけを記述する．関数名を指定しないのは，呼び出される関数 (割り込み関数) が一意に決まるからである．なお，上記の「int 型変数」の部分には，int 型の数値を直接記述しても良いし，int 型を返す関数呼び出

しを記述しても良い.

4.3.4.2　特急関数呼び出し要求

リモート・オブジェクトで現在実行中の関数の実行が完了した後，割り込み関数を呼び出す．従って，オブジェクトの状態を正しく保ったまま，割り込み関数を呼び出すことができる．この割り込み関数の実行が終了したら，キューに貯められている次のリモート・メンバ関数呼び出し要求が処理される．緊急関数呼び出しと同様に，非同期で呼び出される．また，リモート・オブジェクトが中断状態の場合は，緊急関数呼び出しと同様に単に割り込み関数が呼び出されるだけであり，その関数の実行が終了したら再び中断状態となる．

[**シンタックス**]

```
リモート・オブジェクト変数 => int 型変数
```

緊急関数呼び出しのシンタックスとほぼ同様で、==>の代わりに=>と記述する点が異なるだけである.

4.3.4.3　割り込み関数の定義

割り込み関数の呼び出し方は，緊急呼び出しと特急呼び出しで記述方法が異なっているが，呼び出される関数 (割り込み関数) の定義方法は 1 種類である．つまり割り込み関数は，呼び出し方法だけが緊急と特急の 2 種類になっており，どちらの呼び出し方をしても同じ関数が呼び出される．

割り込み関数の定義は，コンストラクタ[2]やデストラクタ[3]と同じように，クラス定義の中で行う．

[シンタックス]

```
class クラス名 {
    ......
    !クラス名(int ID) {
            // 割り込み関数の定義
    };
    ......
};
```

割り込み関数の定義では，!をクラス名の左に記述し，int 型の引数を 1 つ取る．割り込み関数の本体に記述できる内容は，通常のメンバ関数と全く同じである．

4.4 他システムとの比較

並列オブジェクト指向言語の研究はこれまで様々な方面で行われているが，特にC++をベースとした研究開発が米国の大学/研究機関を中心に多く見られる．以下に代表的なシステムを挙げ，*ANUFO* との違いに基づいて特徴付ける．

- Concurrent C++[30]
  Concurrent C++では，スレッド内に C++を記述できるというものであるが，スレッドの生成やスレッド間の通信が C++のオブジェクトの生成やメンバ関数呼び出しという形にはなっておらず，違う機構として組み込まれている．そ

[2]そのクラスのオブジェクトが生成される時に呼び出される関数．
[3]オブジェクトが消去される時に呼び出される関数．

のため，例えば，これまで1つのスレッドにしていた部分を2つに分割しようとするような粒度の調整を行う場合，分割されたスレッド同士の通信等を記述し直さなければならない．一方 $\mathcal{ANUFO}$ では，オブジェクトにキーワード `remote` を付ける程度で良い．

- Compositional C++[31]
  ブロック内の各ステートメントを並列に実行する par ブロックや並列ループである parfor 文などにより並列実行を行う．従って，C++言語の拡張ではあるが，オブジェクトをベースとした並列処理が行えない．その他に本システムでは，sync 変数と呼ばれる同期機構を持った通信のための変数が提供されており，これは $\mathcal{ANUFO}$ の同期機能付き変数とほぼ同じ機能を持っている．

- pC++[32]
  コレクションと呼ばれるオブジェクトの集合が定義でき，要素オブジェクトに対する関数呼び出しが並列に実行される．要素オブジェクトのプロセッサへのマッピングなどを指定することができ，HPF(High Performance Fortran)[33] の C++版と言える．並列マシン上に分散された各オブジェクトに，1つのプログラム (関数) を適用するというデータ並列に適したプログラミングが可能であるが，オブジェクト同士が複雑に呼び出し合うような並列プログラミング手法を取ることは困難である．

- Charm++[34]
  スレッド (本システムでは Chare と呼んでいる) 間の直接通信は，メンバ関数とは異なるエントリと呼ばれる機構で行われる．またその際に，通信データは，メッセージと呼ばれるオブジェクトに詰め込む．その他に，複数のプロセッサに分散される Branched chares や共有データを保持するための大域オブジェクトがデータの性質等によって5種類 (Read-Only, Write-Once, Accumulator, Monotonic, Distributed table) 提供されている．

- Mentat[35]
  オブジェクトに通常の C++のオブジェクトと Mentat オブジェクトがあり，Mentat オブジェクト同士は並列に実行される．Mentat オブジェクト同士の

通信はメンバ関数呼び出しで行われ，関数の返り値を使っている個所をシステムが検出し，自動的に関数呼び出しの待ち合わせ等の同期が行われる．このシステムの並列処理機構は*ANUFO*が目指すものに非常に近いが，リモート(Mentat)オブジェクトの指定をMentatではクラス定義時に行うのに対し，*ANUFO*ではオブジェクト生成時に行う点が異なっている．*ANUFO*においては，同じクラスをリモートにもローカルにも使用できるが，Mentatでは不可能である．

4.5　実現方式

*ANUFO*プログラムは，トランスレータによりC++プログラムに変換されるが，その変換方法について概説する．

4.3.1節で述べたように，リモート・オブジェクトの変数宣言時及びオブジェクトの生成時に，キーワード`remote`によりリモートであることを指定する．従って，トランスレート時に，オブジェクトの生成やメンバ関数呼び出しがリモート・オブジェクトに対してであることを知ることができる．リモート・オブジェクトへのメンバ関数呼び出しでは，その実引数の引き渡しをスレッド間通信で行わなければならない．この関数の各引数データを通信ライブラリで送信するオブジェクトをエンコーダ，通信ライブラリで受信するオブジェクトをデコーダと呼び，*ANUFO*上でのリモート・オブジェクト変数は，エンコーダ・オブジェクト変数に変換される(図4.6)．

エンコーダ・クラスには対象となるリモート・オブジェクトのクラスと同じメンバ関数が定義されており[4]，各メンバ関数では，引数を通信ライブラリで送信するようプログラムが生成される．また，デコーダ・クラスには，通信ライブラリで受信し，対応するメンバ関数を呼び出すようなプログラムが生成される．

[4]メンバ変数アクセス用の関数も追加される．

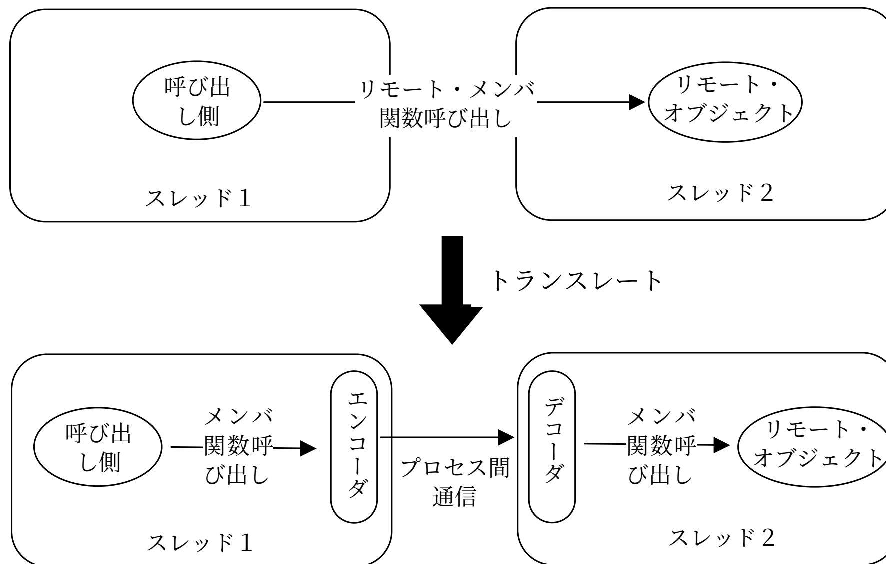


図 4.6: *ANUFO* から C++へのトランスレート

C++と通信ライブラリを用いた記述においても，通信においては，エンコーダとデコーダが行っているようなデータのシリアライズが必要となるが，トランスレータにより変換されたプログラムにおいては，直接通信する場合に比べて，エンコーダへのメンバ関数呼び出しとデコーダからのメンバ関数呼び出しが余分に実行されることになる．しかし，これらの呼び出しには，ループ等の複雑な処理が含まれないので，inline 展開などの C++コンパイラの最適化により，直接通信をした場合とほとんど変わらないオブジェクト・コードが生成されるはずであり，実行時オーバヘッドはほとんどないことも次節の評価で分かっている．

4.6　評価

現在，トランスレータ及び通信ライブラリとして PVM (Parallel Virtual Machine)[26, 27] を使用した版が，イーサネットで結合された WS クラスタ上で稼働している．そこで，13-Queen 問題を使って，*ANUFO* プログラムと C++(WS 間通信の部分は

PVMを使用した) プログラムを比較する.

13-Queenのプログラムは，図4.2のプログラム例に示したように，仕事を供給するProducerオブジェクトと仕事を実際に処理するWS台数分のConsumerオブジェクトから構成され，Consumerは仕事を処理し終ったらProducerに仕事の要求を行なう形で処理を進めていく．ここで，仕事とは，幾つか (13以下) のQueenを置いた盤面であり，各Consumerはその盤面を基にして13個のQueenを置けるパターンを全て求める.

*ANUFO*及びC++によるプログラムとも，Queenを2つ置いた状態の盤面を各プロセッサに順番に割り付けていく，静的負荷分散を採用している．どちらも全く同じ負荷分散方式を採っているので，記述言語による負荷のアンバランスの違いはない.

4.6.1 プログラムサイズ

表4.1に*ANUFO*プログラム, C++プログラム, *ANUFO*をトランスレートしたC++プログラムの行数及びそれらの比率を示す．C++プログラムでは，WS間通信で，通信バッファへのデータ転送などのPVMライブラリ関数呼び出しが記述されているが，*ANUFO*では，単なるメンバ関数呼び出しとなっている．そのため，約25%ほどC++のプログラムサイズが大きくなっている.

13-Queenプログラムは評価用の小さなプログラムであり，各オブジェクトも単純

表 4.1: 13-Queenのプログラムサイズ

ANUFO	C++	変換後のC++
175行	219行	398行
1.00	1.25	2.27

表 4.2: 各 WS の性能比率

種類	WS1	WS2	WS3,4	WS5,6	WS7,8
時間	40,006	73,800	90,092	100,120	185,086
比率	1.00	0.54	0.44	0.40	0.22

表 4.3: 13-Queen の実行時間

WS 数	1 台	2 台	4 台	8 台
$\mathcal{ANUFO}$	40,006	26,543	17,516	12,590
C++	39,602	25,947	17,017	12,277
$\mathcal{ANUFO}$/ C++	1.010	1.023	1.029	1.025

な呼び出し関係をしているのでこの程度の差であるが，実用レベルの大規模なプログラムになるとこの差はかなり大きくなり，それに伴う開発コスト (コーディング/デバッグ時間など) もかなり差が出ると思われる．

また，トランスレータが生成した C++プログラムは 2 倍強の大きさになっており，人手による C++プログラムに比べてもかなり大きくなっている．これは，使用されないリモート・メンバ関数呼び出しのためのコードまで生成しているからである．

4.6.2 実行時間

今回計測に用いた WS クラスタはすべて同じ種類のマシンではなく，幾つかの異なった種類のマシンから成る非均質な構成である．それぞれの WS の性能比率を知るために，13-Queen を各 WS で逐次実行した．マシンの種類毎の実行時間 (単位: ミリ秒) と最も速い WS1 との比率を表 4.2 に示す．

表 4.3 は，WS1 ∼ WS8 の WS を性能の良い順に 1, 2, 4, 8 台使用した場合のそれぞれの実行時間 (単位: ミリ秒) 及びそれぞれの台数における $\mathcal{ANUFO}$ と C++の実

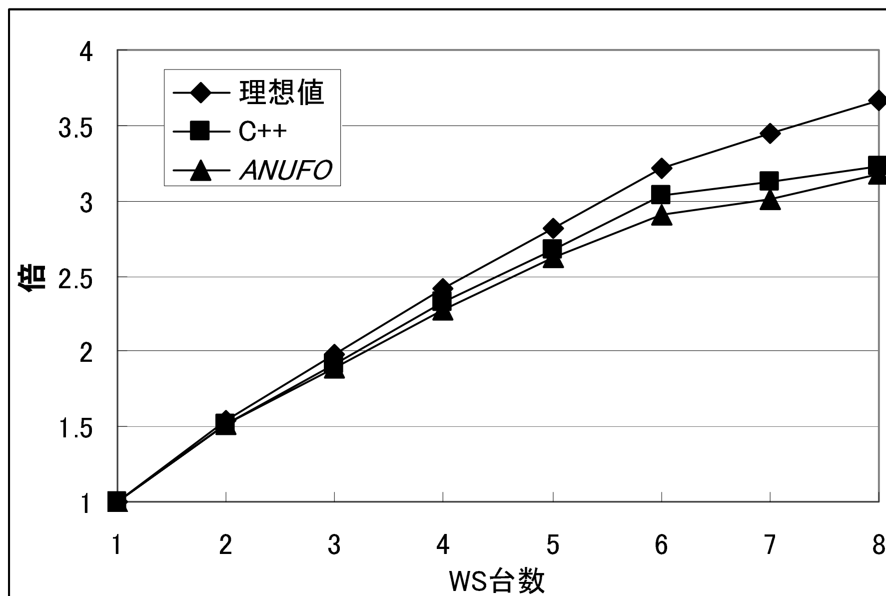


図 4.7: 台数効果

行時間比率を示している．各台数において，*ANUFO* が 1%~3%弱遅くなっているが，ほとんど無視できる程度の差と言えるであろう．通信の際に，*ANUFO* ではエンコーダ及びデコーダ・オブジェクトが間に介されるが，予想通りコンパイラの最適化によりその影響がほとんど現れていない．

図 4.7 は，1 台で実行した場合に対する実行時間比率 (台数効果) である．理想値は，非均質環境なので，表 4.2 で示した各 WS の性能比率の和で計算しており，直線にはなっていない．グラフから分かるように，*ANUFO* と C++はほとんど同じ傾向を示しており，*ANUFO* で記述することによる実行時のオーバヘッドはほとんど見受けられない．また，6 台くらいから理想値より離れていっているのは，台数が増えたことによる通信量の増加 (ネットワーク性能の問題) と Job の供給を行う Producer がネックになっていること (並列化手法の問題) が原因と考えられる．

以上，計測の結果，*ANUFO* で記述することによりプログラムサイズは小さくなり開発量が削減でき，実行時のオーバヘッドはほとんどないということが分かった．

4.7 結言

以上，既存の逐次処理環境からの移行性を考慮し，オブジェクト指向をベースとした効率的な並列性制御を可能とするプログラミング言語機能及びその実現方式を提案した．実験プログラムによる評価の結果，直接 C++と通信ライブラリ PVM で記述した場合に比べて，25%程度プログラム量が削減され，実行時オーバヘッドはほとんどないことを確認した．

本提案は C++言語をベースとしているが，リモート・オブジェクトの考え方は，同じくオブジェクト指向言語として多く利用されている JAVA にも適用できると考えている．

また今後は，共有メモリを持ったマルチプロセッサをネットワークで接続するハイブリッドなクラスタが主流となるであろう．その際にも，オブジェクト内並列は共有メモリを前提とした自動並列化コンパイラを用い，オブジェクト間並列は $\mathcal{ANUFO}$ を用いて記述する，といった利用方法が考えられる．

第5章　クラスタにおける効率的な並列性制御、負荷分散方式 –医用放射線量分布計算への応用

5.1　緒言

近年のパーソナルコンピュータ（PC）の低価格化とギガビット・イーサネットを代表とする汎用ネットワークの高速化により，PC やワークステーション（WS）を構成要素とする並列処理クラスタが，安価で拡張性の高い計算サーバとして注目されて久しい [36]．特に 16 台程度の PC クラスタであれば，百万円～ 2 百万円程度で容易に構築できるため，並列処理が身近になってきており，非常に処理時間を要する大規模計算だけでなく，これまで並列処理を適用するまでもないような分野にまで，安価に並列処理を適用することが可能となってきた．

また，並列計算を必要とする応用分野の内，計算領域 (範囲) を単純に複数に分割して並列化できるような場合は，分割されたそれぞれ並列に実行されるプログラム同士でほとんど同期・通信が不要なため，比較的容易に逐次処理環境からの移行が可能である．このように小規模な並列処理環境において，それぞれ並列実行されるプログラム同士に依存関係のない問題を並列処理する場合は，複雑な分散管理を考えるよりは，単純な構成を採り，逐次処理環境からの移行性を重視した方式を採用すべきと考える．

本章では，このような逐次処理環境からの移行を容易にし，比較的小規模なクラスタ環境において並列性制御及び負荷分散を支援する方式について述べる．さらに，放射線医療分野の応用問題を採り上げ，より大きな並列化効果を上げるために，汎用的な並列性制御や負荷分散方式だけでなく，対象問題の特性を考慮した方式を示す．

5.2　分散型並列処理支援ルーツ ParaJET

分散型並列処理支援ツール ParaJET(Parallel Job Execution Tool)[37] は，クラスタ環境全体を一つの並列システムと見做し，与えられた複数のジョブを各 WS や PC に自動分配して並列実行させることを目的としたツールである．ここでジョブとは，ユーザから見た仕事の最小単位のことで,「応用プログラムの実行」に相当する．従って ParaJET は,「互いに独立して実行できるジョブを多数実行させたい場合」に対して適用して効果を上げることができる．

具体的な例としては，LSI の論理シミュレーションが挙げられる．これは，設計した論理回路に様々なテストパターンを入力して回路の正しさを検証する，という処理であるが，個々のテストパターンの処理は完全に独立して実行できる．従って，それぞれのテストパターンの処理を一つのジョブとして ParaJET により並列実行させることにより，全体の実行時間を大幅に短縮することができる．

他に，例えば次のような例に対して適用可能である．

- 画像処理 (画像を分割してそれぞれのデータ処理をジョブとする).
- 暗号解読のアルゴリズム検証 (暗号解読の一つのキーに対する処理を一つのジョブとする).
- 最適パラメータ探索 (１つ１つのパラメータの組に対する処理を一つのジョブとする).

特に最後の最適パラメータ探索に関しては，実際に，人工衛星に使用される部品

や各種電機製品などの熱解析の設計に適用されている [38, 39, 40]．これらの熱設計においては，例えば，放熱フィンの本数，高さ，間隔などを探索パラメータとして，各プロセッサ (PC) で与えられたパラメータ値に基づいて評価点の温度を並列に計算し，所望の温度となるパラメータ値を求めている．

5.2.1 機能概要

ParaJET が提供する機能を以下に示す．

- ジョブが実行される各 WS や PC の負荷状況に応じて，ユーザの投入したジョブを動的に分散する．
- ジョブの定義は，ジョブ定義ファイルに記述する．各ジョブは，Shell に入力する 1 行と考えて良い．すなわち，プログラム名と引数列から成る 1 行をジョブの数だけ記述する．引数には，領域分割による並列処理であれば，領域を表す数値を記述し，最適パラメータ探索のような場合は，パラメータ値を記述する．
- 一定時間以内にジョブの実行が終了しない場合に，そのジョブを強制終了する．これは，ジョブの応用プログラムにエラーがあって暴走する可能性を考慮したものである．
- ジョブを任意の階層にグルーピングして扱うことが可能である．例えばテスト項目毎にジョブをグルーピングするなど，関連するジョブをグルーピングすることにより，ジョブの管理が簡単になることが期待される．また，次のようなことも可能である．
  - 任意のグループを負荷分散の単位とすることができる．これにより，ジョブの分配に伴う通信オーバーヘッドを軽減させることができる．
  - グループをエラー処理の単位とすることができる．具体的には，あるグループ内のジョブの実行がエラーとなった場合，以下のいずれかが選択

できる．

1. グループ内の残りのジョブの実行を放棄する．
2. グループ内の残りのジョブの実行を予定通り進める．

- 各負荷分散単位に計算資源の条件を指定し，条件を満たす WS や PC で実行させることができるようになっている．例えば,「メモリが 32MB 以上載っている WS で実行したい」などという指定が可能となる．
- 各 WS や PC ごとに，ある時間帯にのみジョブを実行させるとか，負荷平均[1]がある値よりも高くなると，新たなジョブを実行させないといった指定が可能である．

5.2.2 負荷分散方式

ParaJET が対象としているジョブは，実行時間が負荷分散制御のための処理時間に比べて十分長いことを前提としている．また，実行環境も安価で容易に構築可能な 16 プロセッサ程度の全体を 1 階層で管理できる環境を想定している．そのため，ParaJET の負荷分散方式は，以下のような比較的単純な要求駆動型の動的負荷分散を採用している．

Server マシンのジョブプールにジョブ群を貯めておき，実際にジョブを実行する各 Client マシンは実行するジョブがなくなったら（又はなくなりそうになったら），Server マシンにジョブの要求を行い，Server マシンはその要求によりジョブプールに貯めているジョブを 1 つ要求元の Client マシンに与える（図 5.1）．なお，Server マシンが Client マシンを兼ね，Server マシンは負荷分散処理を行っていない間は自らもジョブの実行を行っても良い．

なお，計算を行う各マシン (PC) 同士は，ネットワーク・ファイル・システム (NFS: Network File System) によりディスクを共有し，ジョブ起動の情報は共有ファイル

[1]一定時間内にカーネルの実行キューにあるプロセスの平均数.

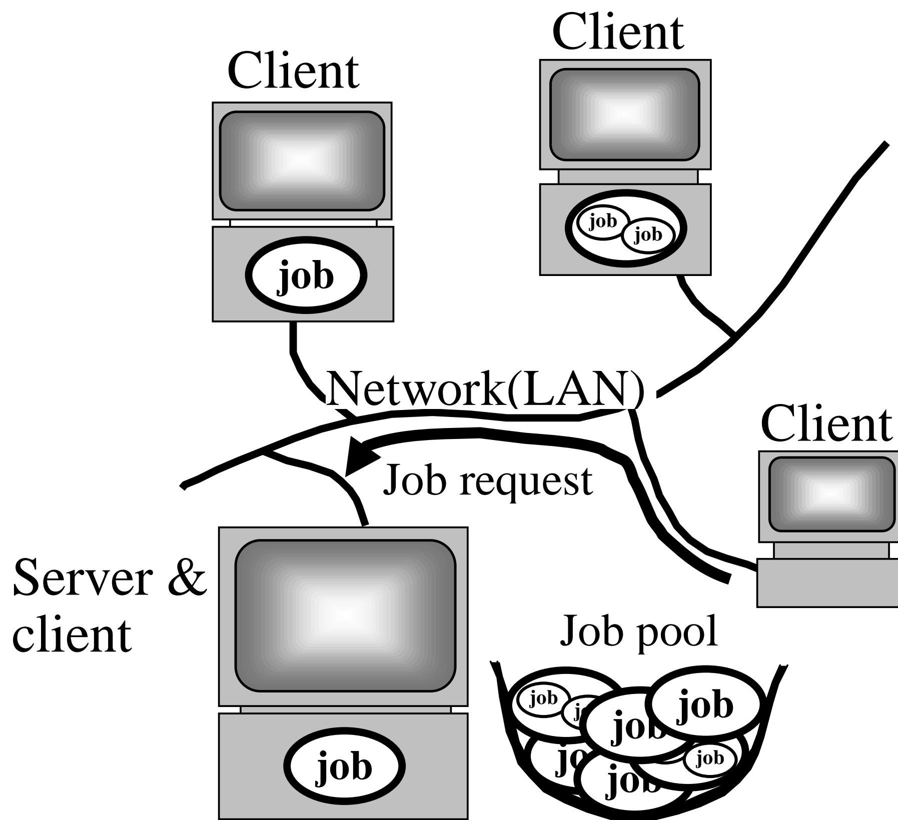


図 5.1: ParaJET の負荷分散方式

により受け渡ししている．また，各マシン間の通信には，多くの種類のマシン上で動作し広く利用されている通信ライブラリ PVM (Parallel Virtual Machine)[26, 27] を用いている．

5.2.3 他システムとの比較

ParaJET と同様に小規模なクラスタを対象とした負荷分散ツールは幾つか存在するが，実際に製品として多くのサイトで利用されているものに，プラットフォーム・コンピューティング社の LSF(Load Sharing Facility)[41, 42] がある．

ParaJET の LSF と比較した際の機能的な利点には，ジョブのグルーピングが挙げられるが，その他にプロセッサ間通信に PVM を使用していることを応用プログラムに公開している点がある．一般に，並列に動作する各ジョブ間で全くデータ交換が必要ないことは稀である．例えば，各ジョブで計算された結果を 1 箇所にマージするなどの処理が必要であり，その際には何らかの通信機構を利用しなければならない．

負荷分散ツールが応用プログラムに通信方法を公開していない場合，例えば，ツールとその上で動作する応用プログラムとの間でソケット・ポートの競合が生じる可能性がある．一方 ParaJET では，PVM を使用しているため，応用プログラムも同じ PVM を使用して通信することにより，そのような競合が生じることはない．

5.3 医用放射線量分布計算への応用

ParaJET の応用として，パラメータ探索を挙げたが，その他に比較的単純な領域分割による並列処理にも容易に適用可能である．ここでは，ParaJET を医療分野における放射線量分布計算に応用した例を示し，更に対象問題の特性を考慮した並列性制御方式について述べる．

5.3.1 医用放射線量分布計算

がん治療法の一つである粒子線治療は，ヘリウム，炭素，ネオンなどの重粒子線を患部に集中して照射し，周辺の正常細胞への影響を最小限に抑え，患者の早期社会復帰を可能とする非常に有効ながん治療方法である [43]．この粒子線を用いたがん照射治療を普及させるためには，遠隔地の複数の医療機関から重粒子線がん照射施設の計算サーバにより，照射施設のノウハウを利用して患者毎に最適な照射方法を得る治療計画を立案できることが望ましい．この治療計画では，効率的な治療を行うために，患者体内の線量分布を正確に計算しなければならない．しかし，従来の高精度な三次元の線量分布計算は処理時間がかかるため，治療計画の効率が悪いという問題点があった．そのため，治療計画の高速化の研究が幾つか行われてきた [44][45][46] が，これらの研究では，トランスピュータや CRY-YMP のような従来型の並列計算機や PentiumPro による特殊な自製の並列計算システムを用いており，コストパーフォーマンスや拡張性の面で問題があった．

線量分布は，患部周辺の平行横断面である CT 画像を複数枚（〜100 枚）取得することで得られる三次元情報（図 5.2）を基に計算する．1 枚の CT 画像は 512×512 ピクセルであり，1 ピクセル当たり 4 バイトで表現しているので，その三次元データ量は 0.5K ピクセル $\times$0.5K ピクセル $\times$100 枚 $\times$4 バイト=100M バイトと大量となる．線量分布計算では，この三次元化された CT 画像データや各種照射に関するデータを基にして各点の吸収線量を計算する．そして，計算結果は CT 画像と重ね合わせて表示するため，各線量を CT 画像データと同じ形式（座標系）の三次元データとして格納する．したがって，計算結果も 100M バイトのデータ量となる．このように，線量分布計算では，大量データを扱うので，並列化の際に，データの通信時間をいかにして抑えるかが課題となる．

線量分布の計算は，以下のような手順で行う．

- 照射される面を格子状に分割し，ビームを各格子への線（Ray）に分割する．図 5.3 では，3×3 の 9 つの Ray に分割している．ここで，コリメータ（Collimator）とは，ビームの横方向の広がりを調整するためのカメラの「絞り」のようなも

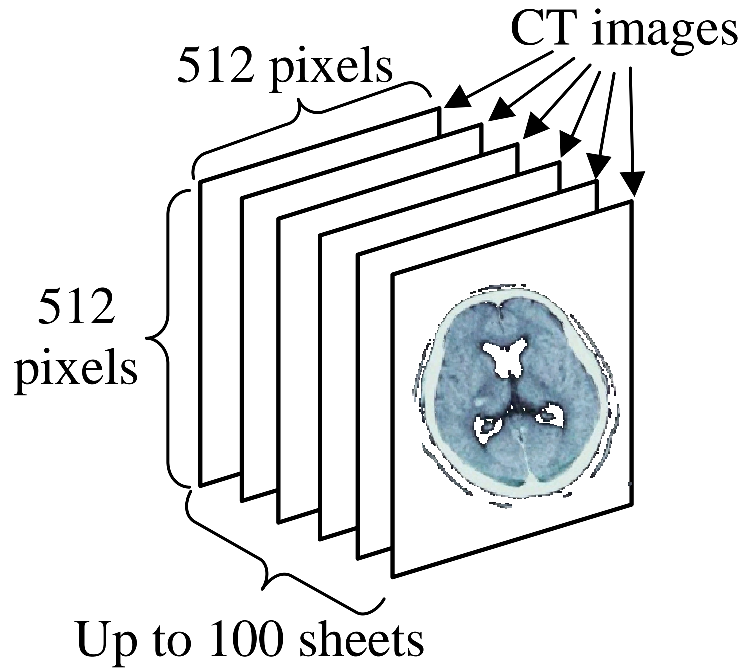


図 5.2: 三次元 CT 画像

のである．

- 各 Ray 毎に Ray 上に沿って一定間隔に計算点を決める（図 5.3 の星印）．
- 各計算点毎に，数式 (5.1)〜(5.3) により線量を算出する．計算結果を CT 画像と同じ座標系の形式に格納する際に，計算点座標と CT 画像座標が正確に一致しないため，その距離に応じて計算結果を線形補間して格納する．

$$D_p = DD_p \times OAR_p \tag{5.1}$$

計算点 p での線量値 D_p は，ビームの中心軸上の線量 DD_p と中心軸外の線量比率である OAR_p から算出される．

$$DD_p = BST(WEL_p) \times (\frac{SSD + WEL_p}{L_p})^2 \tag{5.2}$$

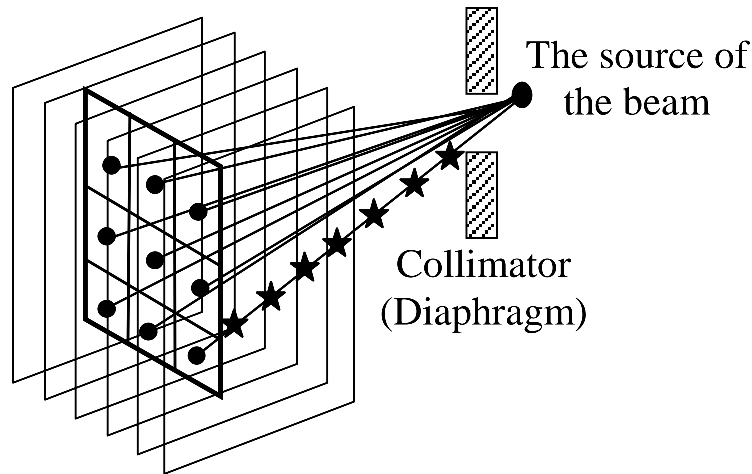


図 5.3: 線量分布計算

別途水中で実際に計測した線量値がテーブル BST に格納されており，計算点 p のビームの中心軸上の線量 DD_p は，線源と計算点 p との距離を水等価厚 WEL_p に変換して，水等価厚でテーブル BST を引くことで得ることができる．なお，SSD は線源から計測開始点までの距離で，L_p は線源から計算点 p までの距離であり，式 (5.2) の第 2 項はビームが逆二乗則で広がることによる粒子数の変化を補正するためのものである．

$$WEL_p = \sum_{i=1}^{p}(ED_i \times Step) \tag{5.3}$$

計算点 p までの水等価厚 WEL_p は，各計算点 i での CT 値から算出される電子密度 ED_i と計算ステップ幅 $Step$ の積分により算出される．

計算点 p での中心軸外の線量比率 OAR_p は，線源のビームサイズと計算点 p でのビームの進行方向に対して横方向の座標を基に算出されるが，その詳細は文献 [47] に記載されている．

なお放射線としては，陽子及び炭素を対象としている．この方式による線量分布計算は，通常の WS を用いて数十秒程度かかり，更にペンシルビーム法 [47] と呼ばれるより正確な結果が得られる方式では更に時間がかかる．

計算点毎の処理は，式 (5.3) により水等価厚の算出で積分しているので，同一 Ray 上の前計算点の（中間）結果が利用可能であり，逐次的になっている．一方，Ray 毎の計算は独立しているので Ray 束の単位で並列に処理できる．つまり，計算領域（Ray）を複数の部分領域に分割し，それぞれの分割された領域を各プロセッサで独立に計算し，計算結果を最後にマージするという並列化方法である．各部分領域（Ray）は，その位置により CT 画像を通過する距離が異なり，それにより必要な計算時間も異なるので，部分領域によって負荷が異なる．したがって，並列化においては，各プロセッサ間で負荷をいかにして均等化するかということも課題となる．

また，治療計画においては，照射線源から照射体を見て撮影したような画像を CT 画像を基に再構成する DRR（Digitally Reconstructed Radiograph：ディジタル再構成 X 線撮影）画像の生成も処理時間がかかるので，線量分布計算と同様の方法で並列化した．

5.3.2　並列処理システム

治療計画の計算処理システムは，10 台の PC を汎用のネットワークで接続した，いわゆる PC クラスタである（図 5.4）．各 PC の CPU は Alpha21164A 600MHz であり，10 台の内 1 台がサーバとして使用される．サーバには 1GB のメモリが搭載され，それ以外のマシンには 512MB 搭載されている．各マシンは，100Mbps のイーサネット・スイッチで接続されている．将来，１ Gbps のイーサネットに拡張する予定である．このクラスタが，遠隔地の医療機関からギガビットレベルの高速なネットワークを介して利用されることを想定している．

図 5.5 に並列処理クラスタの実物写真を示す．このクラスタは，２セットのデスクトップ型の PC を 5 台積み重ねたものから成り，1 台数十万円の PC とネットワーク機器などで，総計数百万円という安価かつ高性能で拡張性のある並列処理環境となっている．

図 5.6 に，並列処理クラスタのソフトウェア構成を示す．オペレーティングシス

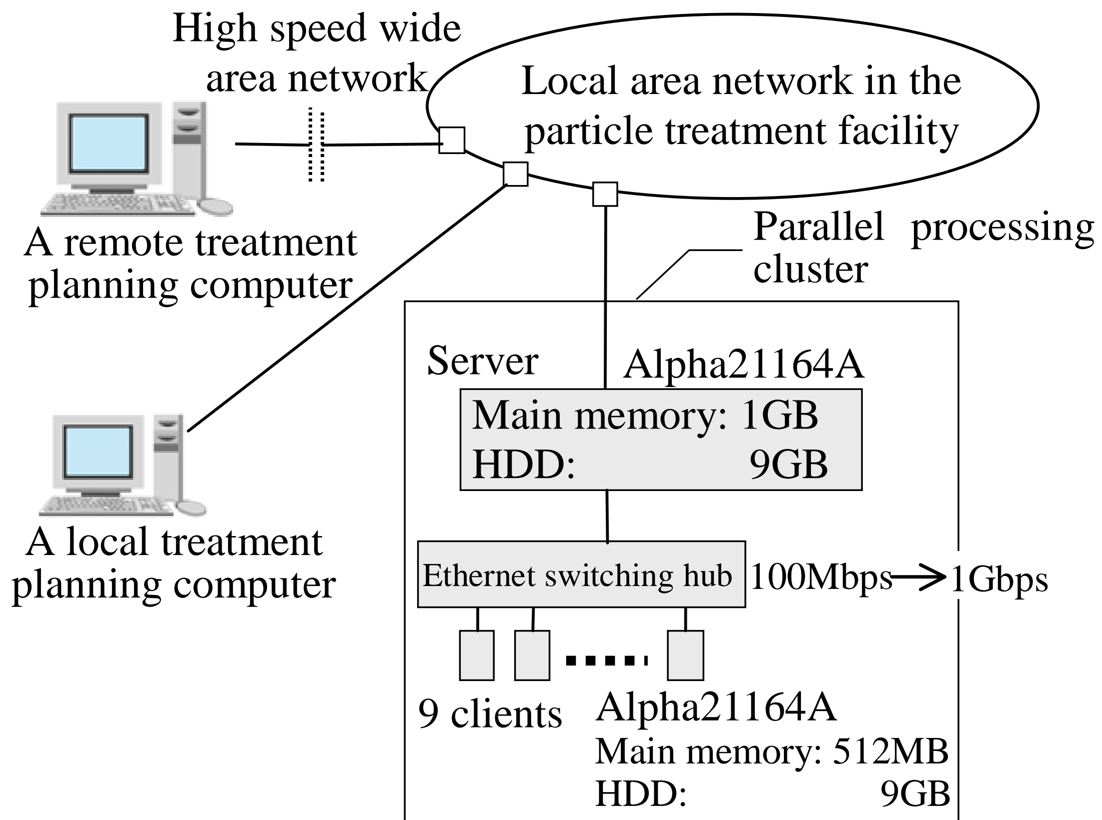


図 5.4: システム構成

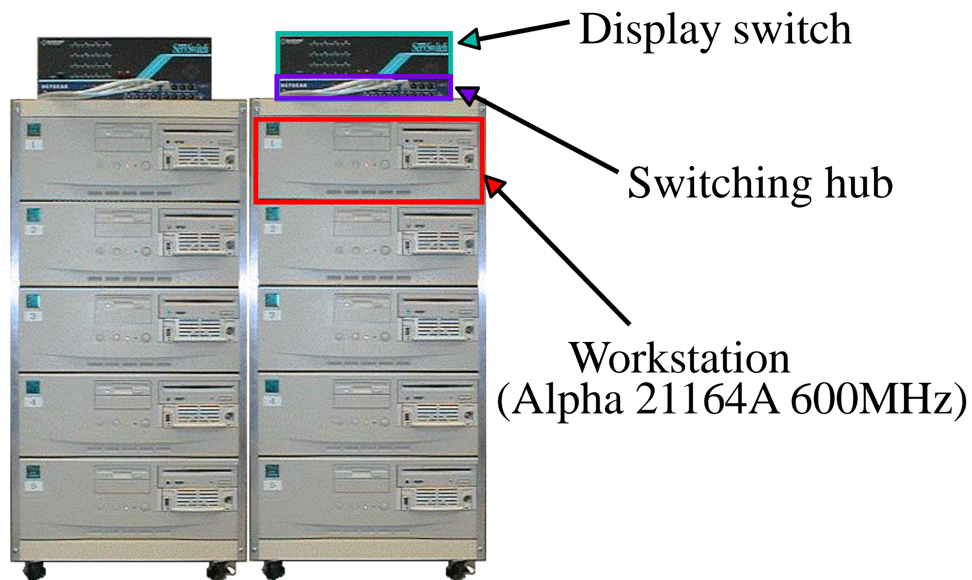


図 5.5: 並列処理クラスタ

Application: Dose/DRR calculation

Load balancing tool: ParaJET

Communication library: PVM

Operating system: Linux
(Redhat 5.2 kernel 2.0.35)

Hardware: Alpha WS cluster

図 5.6: ソフトウェア構成

テムはLinuxを用い，プロセッサ（PC）間の通信ライブラリは，PVMを用いている．また，ParaJETを用いて，動的に負荷分散を行う．

5.3.3 並列処理方式

線量分布計算及びDRR計算の並列処理方式を検討する際に，以下のことを考慮しなければならない．

- データ転送がネックとなることが明白なので，できるだけ転送データ量を削減するよう努める．
- 領域毎の負荷のアンバランスの解消とデータ転送と計算実行をできるだけ同時に行えるように，領域をプロセッサ台数以上に分割し，各プロセッサに複数の領域を担当させるようにする．

以上のことから，線量分布計算は，以下のように処理する．

(1). 各クライアントが計算結果をサーバへ転送する際には，データを圧縮して転送する．Rayの座標系とCT画像及び計算結果データの座標系は異なる，すなわち，Ray上で連続する領域はCT画像データでは連続した領域とはならないので，各クライアントにCT画像を分割して渡したり，結果データを分割した状態で保持することはできない．そのため，全領域の結果データを走査して線量値が0である部分を転送しないようにすることで，転送量を削減（圧縮）する．また，計算結果はfloat型の4バイトデータとして保持しているが，ユーザの要求仕様として計算結果の精度は2バイト程度でも良いとされているので，2バイトに圧縮して転送する．具体的には，C言語において，float型の計算結果に0xFFFFを乗じてunsigned short int型に変換することで4バイトから2バイトに圧縮している．計算された線量値の範囲は0.0〜1.0で，値の分布の偏りもないので，この圧縮方法で大きな桁落ち等の問題が発生することはない．

後述の評価実験で用いる計算モデルでは，全データ量は100Mバイトと非常に大きいが，照射領域に相当するデータ量は，コリメータ直径40mmとマージン40mmから約9.8Mバイト（$(40+40)^2 \div 256^2 \times 100$ M）となる．更に線量値を2バイトに圧縮して通信することで，通信データ量は4.9Mバイトとなる．

(2). Ray上の計算領域はその位置により計算時間が異なっている．そこで，計算領域をプロセッサ台数以上に細かく分割して，各プロセッサで複数領域を担当するようにする．これとParaJETの動的負荷分散機能により，各計算領域毎の計算時間のアンバランスを解消する．

(3). 線量分布計算プログラムは，起動時にCTデータの入力，コリメータや補償体などの照射機器情報の初期化を行っている．線量分布計算プログラムを単純に計算領域を引数として起動する逐次プログラムとして，ParaJETを使って負荷分散を行うと，1つのプロセッサが複数の領域を担当する場合，負荷が分散されるたびに初期化をしてしまうことになる．そこで，線量分布計算プログラムは，各プロセッサで起動し初期化した後，通信により計算領域を受け取り，計算が終了してもプログラムは終了せず再び計算領域を通信により受け取るという方法を採る．そして，このプログラムに計算領域を与えるプログラムをParaJETにより負荷分散する．これにより，初期化は1回だけで済む．
また，各プロセッサで計算した結果をサーバで収集しなければならないが，サーバでは線量分布計算プログラムとデータ収集プログラム間でのネットワークを経由しないプロセス間通信となる．しかし，データ量が多い場合はその通信時間も無視できない．そこで，サーバ上の線量分布計算プログラムはデータ収集プログラムを兼ねることにし，線量分布計算の合間にデータ収集を行うようにする．これにより，サーバ上では，線量分布計算プログラムとデータ収集プログラムのデータのやり取りが不要となる．

この方式による逐次プログラムからの変更点は，通信により計算領域を受け取り，それに従って計算を開始する部分と，各プロセッサから計算結果を収集する部分である．その変更ソース量は，200ライン程度であり，線量分布計算の全体のプログラム量（約20Kライン）に対して，1%程度と軽微な変更で済んでいる．

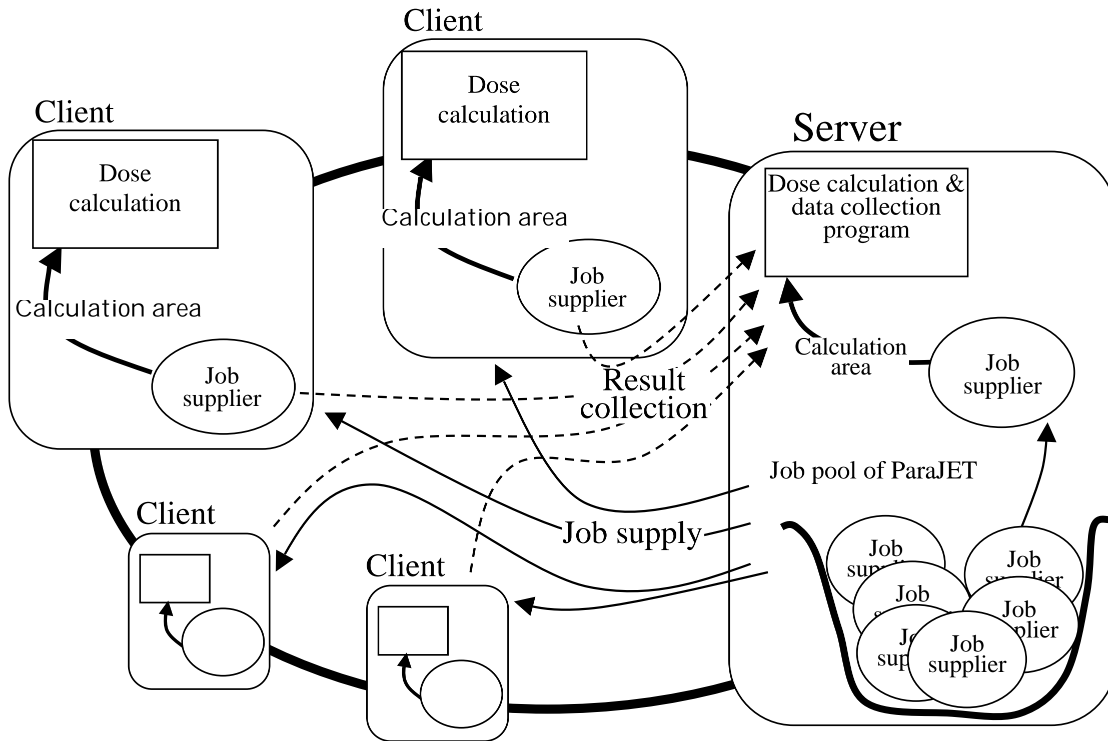


図 5.7: 並列処理方式

具体的な並列処理方式は，以下のようになる（図 5.7）．

- 予め ParaJET 及び PVM デーモンを起動しておく．
- 本システムの起動プログラムが，サーバ及びクライアント上に線量分布計算デーモン（Dose calculation）を起動し，ParaJET のジョブプール（Job pool of ParaJET）に，計算領域の分割数分の Job Supplier を格納する．なお，計算領域の分割数はプロセッサ数の定数倍とする．
- 線量分布計算デーモンは，計算領域（計算開始の Ray 番号と担当する Ray の本数）を PVM 通信により渡されると，その領域の線量分布計算を行うプログラムである．ただし，サーバでは，このプログラムはデータ収集プログラム（data collection）を兼ねている．つまり，線量分布計算の合間に定期的にクラ

イアントから結果データが送られてきたかをポーリングして，データ収集を行う．

- 各 Job Supplier は，担当する計算領域（Ray の開始番号と担当する Ray の本数）をそれぞれ保持しており，それらを自プロセッサ上の線量分布計算デーモンに渡すプログラムである．このプログラムは，線量分布計算デーモンに担当する計算領域を通信ライブラリ PVM により通信して渡し，線量分布計算が終了したら PVM 通信によりその通知を受け終了する．したがって，線量分布計算を行っている間（実際には処理を全く行わないが）生きているプログラムである．このプログラムを ParaJET のジョブプールに格納することで，ParaJET によって 5.2.2 節で示した方式により，アイドル状態の（仕事のない又は終了した）プロセッサ上に割り付けられる（Job supply）．そして，計算領域（Calculation area）がそのプロセッサ上の線量分布計算デーモンに渡される．

- 各プロセッサ上の線量分布計算デーモンは，計算領域を受け取るとその領域の線量分布計算を行い，計算結果をサーバへ送信する（Result collection）．その際，計算した領域のみデータ転送し，線量値も 4 バイトから 2 バイトに圧縮して転送する．

- サーバ上のデータ収集プログラムは，分割数分の計算結果を受け取ると，全クライアントの線量分布計算デーモンに終了を通知すると共に，結果をディスクに書き込んで終了する．

5.3.4 評価

ここで示した並列性制御方式及び負荷分散方式を評価するために，擬似的な計算モデルを用いて実験を行った．

5.3.4.1　計算モデル

線量分布計算　この実験で用いた線量分布計算の計算モデルは，256mm × 256mm × 200mm の水で満たされた直方体に直径 40mm の球（球の媒質も水）があり，線源側に直径 40mm のコリメータと球の表面に均一線量を照射するために半径 20mm の半球状にくり貫かれた補償体（放射線吸収体）がある（図 5.8）．計算の基となる CT データ及び線量分布結果は，512 × 512 × 100 ピクセルであり，1 ピクセル 4 バイトで表現しているので，そのデータ・サイズは 100M バイトとなる．このモデルでの線量分布の計算結果の二次元表現は，図 5.9 のようになる（実際にはカラー表示）．治療計画においては，このような等線量分布表示は，患部の CT 画像と重ね合せて表示される．

DRR 計算　DRR 計算の計算モデルは，線量分布計算の計算モデルとほとんど同じであるが，コリメータや補償体は DRR 計算にとって不要なので削除してあり，また，照射方向により計算結果が変わるようにターゲットを半円柱（かまぼこ状）にしている．このモデルでの DRR の計算結果は図 5.10 のようになり，上部が薄く下部が厚いかまぼこ状になっているのが分かる．この DRR 計算の結果は 512 × 512 の二次元データなので，そのサイズは 1M バイトと線量分布計算に比べて小さい．

5.3.4.2　実行時間と速度向上率

逐次実行と 10 台で並列実行したそれぞれの実行時間とその内訳を表 5.1 に示す．なお，これらの実行における計算領域の分割数は，線量計算が 20 で，DRR 計算が 40 である．ここで，“Input” とは CT 画像をファイルから読込む時間であり，“Output” とは計算結果をファイルに書出す時間である．また，“Init.” とは照射領域のサイズを算出する初期化処理であり，“Comm.” は通信時間，“Comp.” は計算時間である．なお，この通信時間は，計算結果をクライアントからサーバへ送信する時間だけであり，その他の並列制御のための通信時間は個々に計測できないため，“Comp.” の

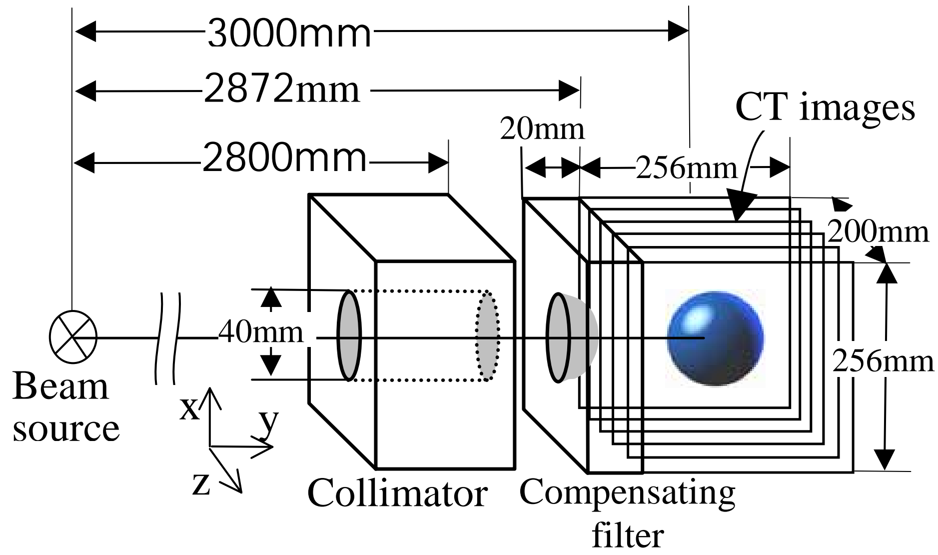


図 5.8: 計算モデル

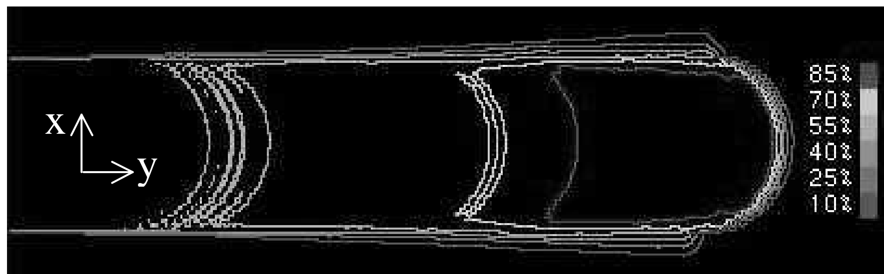


図 5.9: 線量分布計算の計算結果

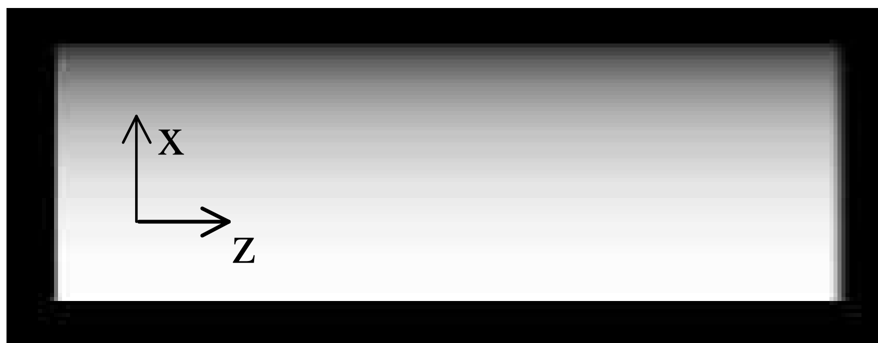


図 5.10: DRR 計算の計算結果

表 5.1: 実行時間（秒）とその内訳

		Dose calculation			DRR calculation		
		Seq.	10 Para.	Ratio	Seq.	10 Para.	Ratio
(1)	Input	1.20	1.20		1.20	1.20	
(2)	Output	1.30	1.30		0.04	0.04	
(3)	Init.	1.90	1.90		0.00	0.00	
(4)	Comm.	0.00	0.70		0.00	0.18	
(5)	Comp.	19.36	2.01	9.63	51.16	5.43	9.42
(6)	Total	23.76	7.11	3.34	52.40	6.85	7.65
(7)	(4)+(5)	19.36	2.71	7.14	51.16	5.61	9.12

方に含まれている．

これらの時間の中で，“Input” と “Init.” は全プロセッサで実行され，“Output” はサーバでのみ実行され，“Comm.” と “Comp.” のみが並列に実行される．そのため，“Comm.” と “Comp.” 以外は，逐次実行と並列実行で同じ時間になっている．

DRR 計算の実行時間が線量分布計算よりかなり長くなっているのは，DRR 計算では CT 画像の全領域を計算対象としているのに対して，線量分布計算では照射領域のみを計算対象としているからである．

線量分布計算では，逐次で 23.76 秒かかっていた処理が 10 台並列実行で 7.11 秒になっているが，これは全実行時間であり，計算（“Comp.”）だけを見ると 2 秒である．したがって，通信時間を除いた速度向上率（並列実行による効果）は 9.63 倍になり，これは使用プロセッサ台数にほぼ等しいと言ってよい．DRR 計算では，逐次で 52.40 秒かかっていた処理が 6.85 秒になっており，速度向上率は 9.42 倍である．これもほぼ使用台数に等しくなっている．したがって，動的負荷分散によるオーバヘッドはほとんどないと言ってよい．また，サーバが各クライアントに最初に計算領域を割り付ける部分が逐次化されているが，計測結果からは少なくとも 10 台程度

までであれば，そのオーバヘッドはほとんどないと言える．

実際の医療現場では，線源や照射機器情報を調整しながら何度か計算を繰り返す利用形態が多い．つまり，初期化以降（“Comp.” 及び”Comm.”）が繰り返されると考えてよく，この部分の高速化が重要となる．図5.11及び図5.12にこの部分の使用プロセッサ台数毎の並列実行効果（逐次実行を基準にした速度向上率）を示す．両者とも，台数が増えるにしたがって線形に高速化されており，スケーラブルであると言える．すなわち，プロセッサ台数を増やせば更に高速化できる．DRR計算では，通信時間を含めた10台実行での速度向上率は9.12倍でありかなり良く，線量分布計算では7.14倍でDRR計算ほど良くない．これは，DRR計算の通信時間が計算時間の3%であるのに対し，線量分布計算では35%もあり，線量分布計算の結果のデータ量がDRR計算に比べてかなり多いため，通信時間が大きくなっているのが原因であると思われる．

いずれにせよ，実際の医療現場で何度も繰り返し行われている処理が，これまで線量分布計算で約20秒，DRR計算で約50秒かかっていたものが，3秒 ~6秒と端末の前で待っていられる時間まで短縮されたことは，非常に有意義であると言える．

5.3.4.3　並列処理方式の評価

5.3.3節で述べた並列処理方式(1)~(3)の効果を評価するために，以下の3種類の線量分布計算の実行時間を計測した．

(a). 5.3.3節の(1)を評価するために，計算結果を圧縮せずに転送する．全データ（100Mバイト）を転送する場合は，各クライアント毎に8秒以上かかり明らかに遅くなるので計測していない．

(b). 5.3.3節の(2)を評価するために，計算領域を使用プロセッサ台数と同じ10分割して実行する．

(c). 5.3.3節の(3)を評価するために，サーバ上のデータ収集プログラムを計算プロセスとは別のプロセスとして実行し，更に計算プロセスは計算領域を受け取る

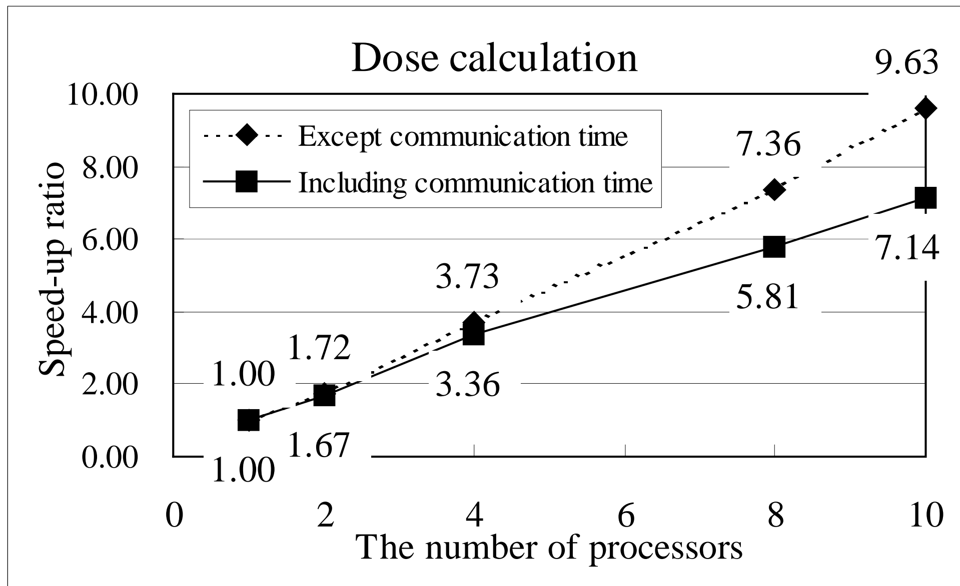


図 5.11: 線量分布計算の速度向上率

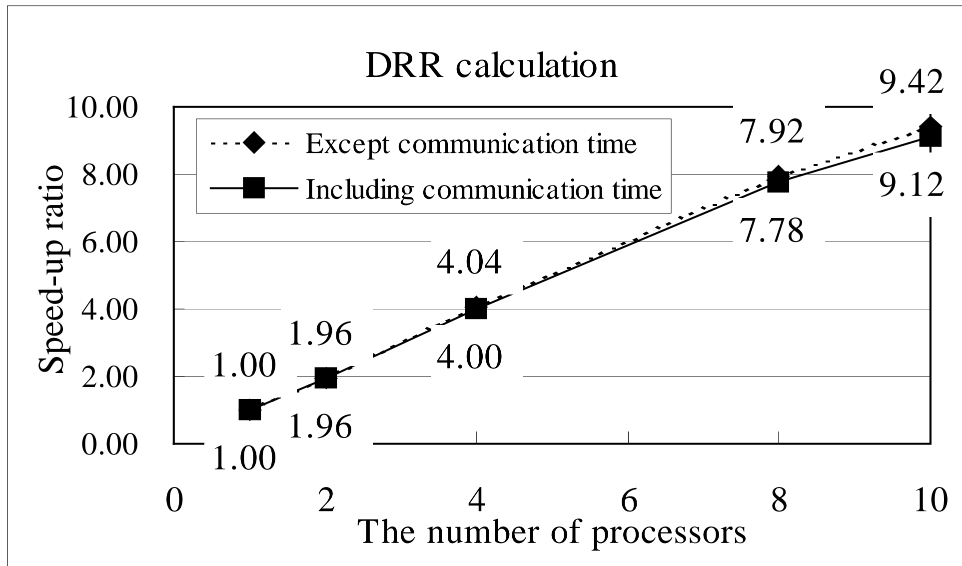


図 5.12: DRR 計算の速度向上率

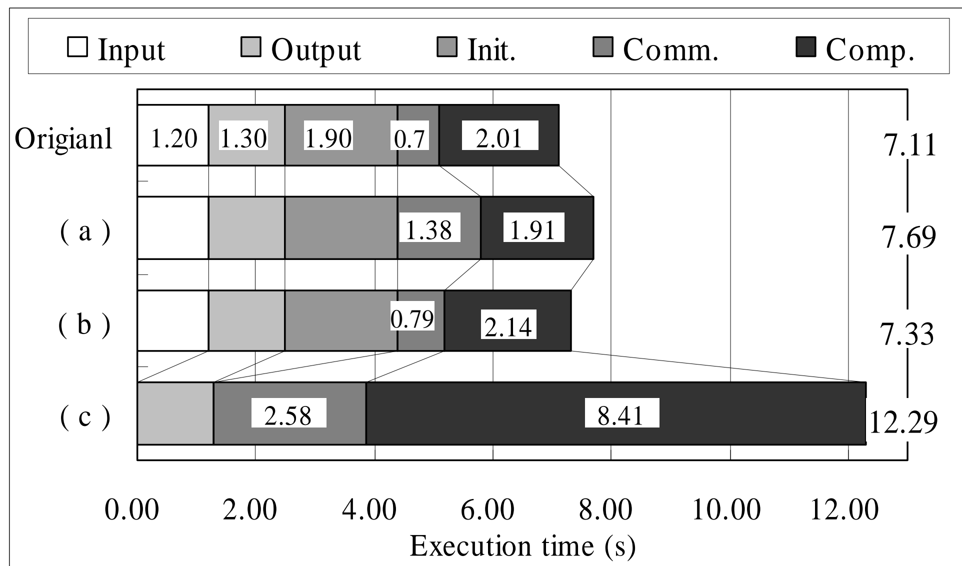


図 5.13: 並列処理方式の評価

たびに初期化を行うようにする．

図 5.13 に計測結果を示し，以下で各項目について考察する．

- 通信時のデータ圧縮の効果
  “Original” と (a) を比較すると，データ通信時にデータ要素を 4 バイトから 2 バイトに圧縮・伸長することにより，通信時間はほぼ半分の 0.70 秒短縮されており，データ圧縮の効果は十分出ていると言える．“Comp.” が約 100 ミリ秒ほど “Original” の方が大きいのは，“Original” ではクライアントでのデータ圧縮の時間が “Comp.” に含まれているからである．なお，サーバでのデータ伸長の時間は “Comm.” に含まれている．

- 各プロセッサでの複数領域担当の効果
  “Original” と (b) の全実行時間を比較すると，220 ミリ秒ほど “Original” の方が短くなっており，各プロセッサで複数領域を担当することにより，各領域毎の計算時間のアンバランス解消の効果が出ていると言える．なお，分割数を

30, 40 の場合も計測したが，20 の場合が最も結果が良かった．この分割数は，並列処理環境の計算性能と通信性能の関係で決まると考えられるが，今回の並列処理環境では，DRR 計算を含めた計測結果から，1 粒度の実行が 1 秒程度の場合が最も良かったと言える．我々の並列処理方式では，この並列粒度を調整できるようにしている点が，1 つの特長として挙げられる．

- 線量分布計算のデーモン化の効果
(c) の計測では，入力（Input）と初期化（Init.）は “Comp.” に含まれる．“Original” と (c) の全実行時間を比較すると，5.18 秒ほど “Original” の方が短くなっており，クライアントでの初期化を 1 回にし，サーバで線量分布計算プログラムとデータ収集プログラム間のデータのやり取りをなくすことの効果が出ていると言える．(c) は計算領域を 20 分割にしているので，各プロセッサで平均 2 領域担当することになり，初期化も 2 回行っている．2 回の初期化で 5.18 秒を要しているので，計算領域を更に小さく分割すると実行時間は更に大きくなると思われる．また，(c) では “Comm.” が非常に時間を要しているが，これは純粋に通信の時間ではなくサーバとクライアントで通信中もサーバ上の別プロセスで線量分布計算を行っているため，その時間も通信時間として計測されてしまったのが原因であると考えられる．

以上のことから，並列処理方式で示した (1)〜(3) の効果を確認することができた．

5.3.4.4　他方式との比較

ここで，文献 [44] の並列処理方式との比較を行う．

[44] では，Ray を分割して並列化するのではなく，三次元 CT 画像の方を分割して並列化している．その理由は，Ray を分割する方式では，CT 画像を全てのプロセッサで保持する必要があるからと述べている．我々は，パラメータを変えながら何度も繰り返される処理部分を高速化したいと考えている．一旦 CT 画像を各プロセッサに転送してしまえば，何度も繰り返される処理においては，CT 画像の転送は不

要である．

また，[44] では，プロセッサの境界で Ray 上の計算点の計算結果を隣接プロセッサに送信する必要があり，従来の逐次方式からの変更も大きくなる．一方，我々の方式では，1 本の Ray 上の計算は従来の逐次方式（プログラム）をそのまま利用できるという利点がある．また，計算結果は最後に 1 度だけ圧縮して送信すれば，それほど大きなオーバヘッドになっていないことが，計測結果から確認できた．

次に，[44] でも我々と同様に 1 プロセッサに複数領域を割り付けているが，静的に割り付けているだけである．すなわち，[44] では各プロセッサの負荷は確率的には均等になるが，その保証はない．一方我々のシステムでは，動的負荷分散を行っているので，各プロセッサの負荷は静的負荷分散に比べて均等になり，計測の結果その動的負荷分散のオーバヘッドはほとんどないことを確認した．

5.4 結言

逐次処理環境からの移行を容易にし，安価で容易に構築可能な比較的小規模なクラスタ環境における並列性制御及び負荷分散を支援する方式について述べ，医用放射線量分布計算への適用例を示した．この並列化は計算領域を分割することにより行い，データ圧縮を行うことによりプロセッサ間通信時間を抑え，また，各 PC に複数領域を担当させることで負荷が均等化されていることを確認した．評価実験の結果，実際の医療現場で何度も繰り返し行われている部分が，線量分布計算では 10 台で 7 倍以上，より通信量の少ない DRR 計算では 9 倍以上の高速化を確認することができた．これは，これまで数十秒かかっていた処理が，3 秒 ~6 秒と端末の前で待っていられる時間まで短縮されたことになり，医療現場に与えるインパクトは大きいと言える．

本章では，医用放射線量分布計算を応用問題として採り上げたが，計算過程においてはある程度の計算精度が必要であるが，最終結果としては 2 バイト程度で構わ

ないという応用問題は多く存在すると考えられる．しかし，通常の計算機の場合は，2 バイトの浮動小数点演算は提供されないため，4 バイトデータを用いるが，通信時においては通信量が半分になるため，2 バイトに圧縮する効果は大きい．少量データを頻繁に何度も通信するような問題の場合には，複数の通信を 1 つにまとめて通信回数を削減する方が効率的であるが，本研究で挙げた大量データを 1 度に通信するような問題の場合は，通信におけるデータ型として 2 バイトデータを提供し，4 バイトデータとの変換（圧縮・伸長）機能を用意することで，通信量を削減することは非常に有効であると思われる．

第6章　おわりに

6.1　本論文のまとめ

本研究の第一の目的は，特に大規模な並列処理環境においてもオーバヘッドを抑えた効率的な並列性制御・負荷分散方式を提案することであった．また，第二の目的は，逐次処理環境からの移行性を重視した並列性制御・負荷分散方式を提案することであった．

第2章では，大規模並列処理環境におけるOSでの効率的な並列性制御方式について述べた．非常に多くのプログラムが高並列に動作している大規模な並列処理環境において，その能力を最大限に引き出すためには，正常時の効率的な実行のみならず，異常事態が起きた際の備えが重要となる．このような処理を行うのは，オペレーティングシステム(OS)の仕事であり，大規模並列処理環境においては，高並列に動作しているユーザプログラムのOSによる効率的な制御方式が，システム全体の性能を左右すると言って良い．ここでは，第五世代コンピュータ・プロジェクトにおいて開発された並列推論マシンPIM用のOSであるPIMOSを例にして，ユーザ・タスクや入出力装置などの資源を資源木により階層的に分散管理・制御する方式を示し，保護フィルタにより，ユーザの誤りがOSやシステム全体に波及することや，悪意を持ったOSへの侵入を防ぐことを示した．また，これらの管理・実行制御をプロセスと呼ばれるプログラミング・スタイルにより実現し，それがオーバヘッドとなっていないことを評価実験により示した．

第3章では，大規模並列処理環境向けの負荷分散方式について述べた．大規模並列処理環境における負荷分散で考慮すべき点は，対象となるプロセッサ数が多いた

め，負荷分散そのものが大きな負荷とならないようにすることである．特に，プロセッサ間の距離により通信時間に差異のあるような場合は，負荷分散の際に大域的な情報を用いずに，近隣プロセッサの局所的な情報のみで作業が行えるのが良い．また，どのような並列分割形態にも有効な汎用的な負荷分散方式の考案は非常に困難であり，ある程度，対象とする分割形態を絞って方式を検討した．ここでは，大規模並列処理環境でもスケーラビリティの面で並列化効果が高く，適用可能な応用分野も広いOR並列型探索問題を主対象として，大域的な情報を用いずに均等化を行いながら自発的に負荷を分散する方式を示し,「世代」という概念を取り入れることで均等化のオーバヘッドを小さく抑えることを可能とした．この世代は，OR並列型探索問題では探索木を一段辿ることに相当し，データ並列型問題ではそれぞれの分割された領域を世代として扱うことができるので，多くのタイプの問題に適用可能な方式である．

第4章では，効率的な並列性制御を行なうための言語サポートについて述べた．並列処理環境においてプログラミングする上で重要な問題の１つに，既存プログラムからの継続性が挙げられる．これは，これまでの逐次処理環境におけるプログラム財産を有効に活用するためだけでなく，近年の並列処理環境が逐次処理環境をネットワークで接続したクラスタの形態を採っているものが多いため，既存の（単一）プロセッサで効率良く動作するプログラムというのが重要となるからである．更に，プログラミング言語レベルで並列性制御機能を提供するためには，単純なデータ並列のような１階層の並列分割形態だけでなく，より複雑な様々な分割形態に対応する必要がある．また，性能チューニング等を考えると，並列性制御を行う部分をできるだけ容易に変更できるようにすべきである．ここでは，データアクセスの局所性等の並列実行可能性を考慮し，オブジェクト指向におけるオブジェクト群単位の並列性制御が容易に行え，更に既存の逐次プログラムからの変更を最小限に抑えた，C++言語をベースとした並列処理言語機能を提案した．実験プログラムによる評価の結果，直接C++と通信ライブラリ関数で記述した場合に比べて，25%程度プログラム量が削減され，実行時オーバヘッドはほとんどないことを確認した．

第5章では，逐次処理環境からの移行を容易にし，安価で容易に構築可能な比較

的小規模なクラスタ環境において並列性制御及び負荷分散を支援する方式について述べ，医用放射線量分布計算への適用例を示した．本応用の計算手法は，いわゆるレイトレーシングであり，その並列化は計算領域を分割することにより行った．ここでは，そのような単純な並列処理形態に対応し，逐次処理環境におけるプログラムがほぼそのまま利用可能な負荷分散方式を提案した．さらに，より大きな並列化効果を上げるために，対象問題の特性を考慮した方式を加味した．評価実験の結果，実際の医療現場で何度も繰り返し行われている部分が，線量分布計算では 10 台で 7 倍以上，より通信量の少ない DRR 計算では 9 倍以上の高速化を確認することができた．これは，これまで数十秒かかっていた処理が，3 秒 ∼6 秒と端末の前で待っていられる時間まで短縮されたことになり，医療現場に与えるインパクトは大きいと言える．

6.2 研究の成果

本研究の成果は以下の通りである．

まず，OS による並列性制御及びミドルウェアによる負荷分散に関しては，大規模並列処理環境においてもオーバヘッドを抑え，効率的な一方式を提案できたと考える．これらの方式は，並列推論マシン PIM 上，並列論理型言語 KL1 という，現在ではある種の特殊な環境で実装・評価したものである．しかし,「世代別動的負荷浸透方式 (LLS-G 方式)」に関しては，その特殊性に全く依存しておらず，大域的な情報を用いず，隣接するプロセッサの情報しか用いていないため，大規模な並列処理環境一般に共通した，近くのプロセッサ同士と遠くのプロセッサ同士の通信時間がかなり異なるような環境において広く有効であると考えている．例えば，プロセッサ同士が物理的に離れた場所に存在するグリッド・コンピューティングのような形態に適しており，今後はそのような環境が増えることが予想されるため，本方式の適用可能範囲は増えると思われる．

前者の OS による並列性制御方式に関しては，論理型言語特有の「プロセス」「ス

トリーム」と呼ばれるプログラミング・スタイルにより，KL1 が提供する「荘園」の機能を用いて実現している．荘園の機能は，現在の UNIX などの逐次処理環境のOS が提供しているプロセス[1]制御の機能そのものであり，プロセスとストリームによる実現方式に関しては，UNIX におけるプロセス（またはスレッド）とソケット通信（またはそれに代わるプロセス間通信）に置き換えて考えることができる．従って，本研究で提案した資源木による階層的な分散管理方式は，現在の大規模並列処理環境においても実現可能である．少なくとも，プロセッサ間の距離により通信時間に差異のあるような大規模な並列処理環境においては，逐次処理環境のように計算資源を集中管理することは処理効率の上で考えることはできず，ここで示した資源木のような階層的な分散管理が効率的な並列性制御に適していると言える．ただし，現在の環境においては，1 つのプロセッサ内では資源の集中管理を行い，複数のプロセッサにまたがる部分において，資源木の方式を採用しないと効率的にならないであろう．

次に，逐次処理環境からの移行性を考慮した並列性制御を行うための言語サポートに関しては，逐次型言語と通信ライブラリによるプログラミングに比べて記述量を抑え，制御オーバヘッドの少ない機能を提案できた．これまで並列処理言語は様々なものが研究され提案されてきているが，なかなか standard にはなっていない．これは，それぞれの言語により，並列化のための仕掛けが異なっており，それを習得する手間が大きいことと言語機能だけでなく，デバッグや性能チューニングのための機能も提供しなければ使い物にならないことが原因であると思われる．従って，並列性記述のみに着目して機能を設計するのではなく，これまで慣れ親しんだ逐次処理環境からの移行性を考慮して機能を設計したことは，使い易さや既存ツールが利用できるという面で利点であった．また，本提案は C++言語をベースとしていたが，ここで提案したリモート・オブジェクトの考え方は，同じくオブジェクト指向言語として多く利用されている JAVA にも適用できると考えている．

最後に，クラスタにおける並列性制御、負荷分散方式に関しては，データ並列のような比較的単純な並列処理形態を汎用的にサポートする並列性制御及び負荷分散

[1] これは UNIX におけるプロセスであり，KL1 のプロセスとは異なる．

を支援する方式を提案し，更に対象問題の特性を考慮した方式を加味することにより，より大きな並列化効果を上げることができた．ここでは，データ通信によるオーバヘッドを抑えるために，ユーザとして必要な計算精度を考慮して，データ圧縮を採り入れた．本応用と同様に，計算過程においてはある程度の計算精度が必要であるが，最終結果としては 2 バイト程度で構わないという応用問題は多く存在すると考えられる．しかし，通常の計算機の場合は，2 バイトの浮動小数点演算は提供されないため，4 バイトデータを用いるが，通信時においては通信量が半分になるため，2 バイトに圧縮する効果は大きい．少量データを頻繁に何度も通信するような問題の場合には，複数の通信を 1 つにまとめて通信回数を削減する方が効率的であるが，本研究で挙げた大量データを 1 度に通信するような問題の場合は，通信におけるデータ型として 2 バイトデータを提供し，4 バイトデータとの変換（圧縮・伸長）機能を用意することで，通信量を削減することは非常に有効であると思われる．

6.3 今後の課題と展望

本研究に残された今後の課題と展望を以下に示す．

まず第一に，本研究で提案した大規模並列処理環境における並列性制御及び負荷分散方式に関しては，その実装・評価が並列推論マシン上であり，現在となっては特殊な環境であることから，現在利用可能な大規模な PC クラスタ上での評価が挙げられる．既に，この環境でも提案した方式は実現可能であること及びその環境に即した改良点を示しているので，実装・評価は環境さえ揃えば可能と考えている．

第二に，応用問題の特性を考慮した制御方式を提案したが，その適用範囲を広げるためには，ここで採用した方式の汎用化が今後の課題となる．先に示したように，通信時のデータ圧縮に関しては，適用可能分野が多いと予想され，汎用化も可能と考えている．また，ここで提案した負荷分散ツールに関しては，負荷（処理）の分配，負荷の要求と実行の機能を提供しているが，計算結果を 1 つにまとめる作業を行わなければならない場合が多いので，負荷の分割，分配，結果の収集をユーザが

行い，負荷の要求，実行，結果の送信の機能のみを提供する方が，容易に適用できる範囲が広がると思われる．

最後に，本研究においては，分散メモリ型の並列処理環境に絞って議論してきたが，最近は，構成要素である各ノードが共有メモリ型のマルチプロセッサ構成である場合も多くなってきている．すなわち，複数の共有メモリ型並列処理マシンをネットワークで接続した構成をしている．このような環境では，共有メモリ環境での並列処理と分散メモリ環境での並列処理をうまく切り分ける必要がある．共有メモリ環境での並列性制御は，並列化コンパイラ等による自動化に任せるという方法も考えられるが，本研究成果を活用して，細粒度と疎粒度の並列性をうまく記述できる仕掛けを入れて，共有メモリと分散メモリのハイブリッドな並列性制御を行う新たな方式を検討していきたいと考えている．

このように今後も様々な並列処理環境が現れてくる可能性が高く，その応用範囲も広がっていくことが期待されるので，今後，制御ソフトウェアの役割がますます重要となってくることは間違いなく，この分野において研究すべき課題もまだまだ多く残されていると言って良い．

謝辞

本研究をまとめるに際し，ご指導を賜りました筑波大学 電子・情報工学系 佐藤三久教授に厚く御礼を申し上げます．また，本論文を作成する際に、有益なご教示を賜りました筑波大学 電子・情報工学系 田中二郎教授に深く感謝申し上げます．また，学生時代に研究の基礎をご指導賜りました法政大学 情報科学部 中田育男教授 (元筑波大学 電子・情報工学系) に深く感謝申し上げます．さらに，最終的な論文を仕上げる際に貴重なご意見を頂いた筑波大学 電子・情報工学系 和田耕一教授，朴泰祐助教授，加藤和彦助教授に深く感謝いたします．

三菱電機 (株) においては、本研究をまとめる機会を与えて頂いた　情報技術総合研究所前所長　小野修一氏，同所長　肥塚裕至氏，同光・電波部門統括　石田修己氏，同知的財産センター長 (元電子システム技術部長)　小菅義夫氏，同電子システム技術部長　平井俊之氏に感謝いたします．特に小菅義夫氏には，この機会を得るために暖かいご支援を頂いたことを感謝いたします．また，本研究全般に深く係わり，永年，筆者の上司として多くのことをご指導頂いた開発本部開発業務部企画グループ・マネージャ　中島克人氏に感謝いたします．

本研究遂行の初期段階では，(財) 新世代コンピュータ技術開発機構において，(財) 日本情報処理開発協会 参与 内田俊一氏 (当時 ICOT 研究部長，後に研究所長)，東京大学 近山隆教授 (当時 ICOT 第二研究室長)，神戸大学 瀧和男教授 (現エイ・アイ・エル (株) 社長，当時 ICOT 第一研究室長)，沖電気工業 (株) 宮崎敏彦氏 (当時 ICOT 研究員) に，暖かいご指導とご援助を頂いた．ここに深く感謝申し上げます．また，研究遂行に際しては，九州大学 越村三幸氏 (当時 ICOT 研究員)，(株) 三菱総合研究所 藤瀬哲朗氏，同 松尾正浩氏, 沖電気工業 (株) 和田久美子氏にご協力頂き，

感謝いたします.

三菱電機 (株) においては，筆者の妻でもある佐藤令子 (元同社情報技術総合研究所)，古市昌一氏 (同社鎌倉製作所)，川上かおり氏 (同社情報技術総合研究所)，白石將氏 (同)，青山功氏 (同社鎌倉製作所)，浅見廣愛氏 (同社情報技術総合研究所)，中川隆文氏 (同社先端技術総合研究所)，依田潔氏 (同)，坂本豪信氏 (電力・産業システム事業所)，東誠一氏 (同) にご協力を頂いた．ここに深く感謝申し上げます.

その他，本研究は，多数の方々のご指導，ご協力の下になされたものです．ここに謹んで御礼申し上げます.

最後に，筆者を学問の道へ導いてくれた父 美幸，母 千代子，そして，研究を行う間，筆者の健康に配慮し，心の支えとなってくれ，かつては家庭内で技術議論を行い研究遂行に協力してくれた妻 令子と，時には邪魔をされることもあったが，いつも笑顔で励ましてくれた二人の息子 柾幸，旺幸に心から感謝します.

参考文献

[1] A.Goto and S.Uchida: ”Toward a High Performance Parallel Inference Machine - The Intermediate Stage Plan of PIM -”, Technical Report TR-201, ICOT, 1986.

[2] 後藤厚宏, 杉江衛, 服部彰, 伊藤徳義, 内田俊一:「並列推論マシン PIM-中期構想-」, 情報処理学会第 33 回全国大会,3B-5, pp.91-92, 1986.

[3] K.Taki:”The parallel software research and development tool: Multi-PSI system”, Technical Report TR-237, ICOT, 1986.

[4] 瀧和男, 木村康則, 横田実, 近山隆, 内田俊一:「Multi-PSI システムの概要」, 情報処理学会第 32 回全国大会, 3Q-8, pp.102-102, 1986.

[5] T.Chikayama, H.Sato, T.Miyazaki:”Overview of the Parallel Inference Machine Operating System (PIMOS)”, In proc. of FGCS’88, pp.230-251, 1988.

[6] 佐藤裕幸, 近山隆, 杉野栄二, 瀧和男:「PIMOS の概要ー並列推論マシン用オペレーティング・システムの構築ー」, 情報処理学会第 34 回全国大会, 2P-8, pp.127-128, 1987.

[7] 佐藤裕幸, 越村三幸, 近山隆, 瀧和男:「並列論理型 OS − PIMOS」, 情報処理学会第 35 回全国大会, 4D-3, pp.259-260, 1987.

[8] 佐藤裕幸, 越村三幸, 近山隆, 藤瀬哲朗, 松尾正浩, 和田久美子: 「PIMOS の資源管理方式」, 情報処理学会論文誌 第 30 巻 第 12 号, pp.1646-1655, 1989.

[9] C.McCann, R.Vaswani and J.Zahorjan:"A Dynamic Processor Allocation Policy for Multiprogramed Shared-Memory Multiprocessors", ACM Trans. Comput. Syst., Vol.11, No.2, pp.146-178, 1993.

[10] B.Lim and A.Agarwal:"Waiting Algorithms for Synchronization in Large-Scale Multiprocessors", ACM Trans. Comput. Syst., Vol.11, No.3, pp.253-294, 1993.

[11] R.P.LaRowe, C.S.Ellis and M.A.Holliday:"Evaluation of NUMA Memory Management Through Modeling and Measurements", IEEE Trans. Parallel and Distributed Systems, Vol.3, No.6, pp.686-701, 1992.

[12] E.Shapiro and A.Takeuchi:"Object Oriented Programming in Concurrent Prolog, New Generation Computing", Springer Verlag Vol.1, No.1, pp.25-48, 1983.

[13] K.Ueda:"Guarded Horn Clauses", Technical Report TR-103, ICOT, 1985.

[14] E.Shapiro and A.Takeuchi:"A Subset of Concurrent Prolog and Its Interpreter", Technical Report TR-003, ICOT, 1983.

[15] 古市昌一, 瀧和男, 市吉伸行 :「疎結合並列計算機上での OR 並列問題に適した動的負荷分散方式とその評価」, KL1 プログラミングワークショップ '90, pp.1-9, 1994.

[16] 古市昌一, 中島克人, 中島浩, 市吉伸行 :「スタック分割動的負荷分散方式とマルチ PSI 上での評価」, 1991 年並列/分散/協調処理に関する『大沼』サマー・ワークショップ, コンピュータシステム研究会, pp.33-40, 1991.

[17] 佐藤令子, 佐藤裕幸,: 「疎結合型マルチプロセッサ上の拡散型負荷分散の一方式」, 電子情報通信学会コンピュータ研究会 (CPSY) 技術報告 CPSY92-9, pp.1-8, 1992.

[18] H.Nakashima, K.Nakajima, S.Kondo, Y.Takeda, Y.Inamura, S.Onishi and K.Masuda:"Architecture and Implementation of PIM/m", In proc. of FGCS'92, pp.425-435, 1992.

[19] 和田正寛, 市吉伸行, 六沢一昭 :「Iterative Deepening A * アルゴリズムのスタック分割動的負荷分散方式による並列化と並列推論マシン PIM/m 上の性能評価」, 1992 年並列/分散/協調処理に関する『日向灘』サマー・ワークショップ, プログラミング–言語・基礎・実践–研究会, pp.195-202, 1992.

[20] S.Aikawa, M.Kamiko, H.Kubo, F.Matsuzawa and T.Chikayama:"ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems", In proc. of FGCS'92, pp.286-293, 1992.

[21] K.Nakajima:"Distributed Implementation of KL1 on the Multi-PSI", Implementation of Distributed Prolog, John Wiley & Sons, pp.311-332, 1992.

[22] http://www.openmp.org.

[23] 佐藤裕幸, 川上かおり:「並列 C++言語システム ANUFO の並列処理機構」 電子情報通信学会コンピュータ研究会 (CPSY) 技術研究報告 CPSY94-83, pp.81-88, 1994.

[24] M.A.Ellis and B.Stroustrup:"The Annotated C++ Reference Manual", Addison-Wesley, 1990.

[25] R.M.Stallman:"Using and Porting GNU CC for version 2.6", Free Software Foundation, 1994.

[26] Computer Science & Mathematics Division Oak Ridge National Lab.:"PVM: Parallel virtual machine", `http://www.csm.ornl.gov/pvm/`, 2000.

[27] A.Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek and V.Sunderam:"PVM 3.0 User's Guide and Reference Manual", Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1993.

[28] 川上かおり, 佐藤裕幸:「並列 C++言語システム ANUFO の同期機構」 電子情報通信学会コンピュータ研究会 (CPSY) 技術研究報告 CPSY94-83, pp.89-96, 1994.

[29] R.S.Nikhil and K.K.Pingali:"I-Structure: Data Structures for Parallel Computing", ACM Trans. on Prog. Lang and Sys., Vol.11, No.4, pp.598-632, 1989.

[30] N.H.Gehani and W.D.Roome:"Concurrent C, CONCURRENT PROGRAMMING", AT&T Bell Laboratories, 1989.

[31] K.Mani Chandy and Carl Kesselman:"Compositional C++: Compositional parallel programming", Technical Report Caltech-CS-TR-92-13, Dept. of Computer Science, California Institute of Technology, 1992.

[32] F.Bodin, Irisa, P.Beckman, D.Gannon, S.Narayana and S.Yang:"Distributed pC++: Basic Ideas for an Object Parallel Language", 1993.

[33] High Performance Fortran Forum:"High Performance Fortran Language Specification", Version 1.0, 1993.

[34] L.V.Kale and S.Krishnan:"CHARM++: A Portable Concurrent Object Oriented System Based On C++", 1993.

[35] Andrew S.Grimshaw:"Easy-to-use Object Oriented Parallel Processing with Mentat", Technical Report CS-92-32, Dept. of Computer Science, University of Virginia, Charlottesville, 1992.

[36] 緑川博子編,:「特集：計算機クラスタ」, 情報処理, vol.39, no.11, pp.1072-1100, 1998.

[37] 白石將, 佐藤裕幸, 中島克人: 「分散型並列処理支援ツール ParaJET」, 電子情報通信学会コンピュータ研究会 (CPSY) 技術報告 CPSY96-60, pp.23-30, 1996.

[38] 佐藤裕幸, 川上かおり, 白石將, 中島克人, 後藤明広: 「分散型並列パラメタサーベイ支援ツール ParaVEY」, 情報処理学会第 55 回全国大会, 2G-04, pp.1-60-61, 1997.

[39] 佐藤裕幸, 永田真也, 中島克人, 福島康之:「分散型並列パラメータサーベイ支援ツール ParaVEY の熱解析への適用」, 情報処理学会第 58 回全国大会, 3D-02, pp.1-3-4, 1999.

[40] 青山功, 羽下誠司, 佐藤裕幸:「応答曲面法を適用した最適パラメータ探索」, 情報処理学会第 64 回全国大会, 3W-01, pp.1-27-28, 2002.

[41] Platform Computing Corp.: 「LSF ユーザーガイド第 1 版」, ダイキン工業株式会社, 1995.

[42] J.Wang, S.Zhou, K.Ahmed and W.Long: "LSBATCH: A Distributed Load Sharing Batch System", Tech. rep. CSRL-286, Univ. of Toronto, 1993.

[43] 辻井博彦:「重粒子放射線によるがん治療の現状」, バイオメカニズム学会誌, vol.21, no.3, pp.108-112, 1997.

[44] F.U.Rosenberger, J.W.Matthews, G.C.Johns, R.E.Drzymala and J.A.Purdy:"Use of transputers for real time dose calculation and presentation for three-dimensional radiation treatment planning", Int. J. Radiat. Oncol. Bool. Phys., vol.25, no.4, pp.709-719, 1993.

[45] J.G.Rosenman, E.L.Chaney, T.J.Cullip, J.R.Symon, V.L.Chi, H.Fuchs and D.S.Stevenson: "VISTANET: Interactive real-time calculation and display of 3-dimensinal radiation dose: an application of gigabit networking", Int. J. Radiat. Oncol. Bool. Phys., vol.25, no.1, pp.123-129, 1993.

[46] Lawrence Livermore National Lab.: "PEREGRINE - Advancing the field of radiation treatment for cancer", http://www-phys.llnl.gov/peregrine/index.html , 2000.

[47] L.Hong, M.Goitein, M.Bucciolini, R.Comiskey, B.Gottschalk, S.Rosenthal, C.Serago and M. Urie: "A pencil beam algorithm for proton dose calculations", Phys. Med. Biol. no.41, pp.1305-1330, 1996.

発表論文一覧

論文誌

[1] 佐藤裕幸, 越村三幸, 近山隆, 藤瀬哲朗, 松尾正浩, 和田久美子: 「PIMOS の資源管理方式」, 情報処理学会論文誌 第 30 巻 第 12 号, pp.1646-1655, 1989.

[2] 佐藤裕幸, 青山功, 浅見廣愛, 中島克人, 坂本豪信, 土谷昌晴: 「遺伝的アルゴリズムを用いた 2 段階方式による粒子線治療装置スケジューリングシステム」, 電子情報通信学会論文誌 D-I 第 J83-D-I 巻 第 10 号, pp.1033-1042, 2000.

[3] 佐藤裕幸, 中川隆文, 依田潔, 中島克人, 坂本豪信, 遠藤真広: 「ワークステーション・クラスタを用いた放射線治療計画の高速化」, 電子情報通信学会論文誌 D-I 第 J85-D-I 巻 第 2 号, pp.184-192, 2002.

[4] 佐藤令子, 佐藤裕幸, 中島克人, 田中千代治: 「疎結合型マルチプロセッサ上の拡散型動的負荷分散方式–LLS-G 方式–」, 情報処理学会論文誌 第 35 巻 第 4 号, pp.571-580, 1994.

[5] 飯塚剛, 佐藤裕幸, 落合真一, 伊藤隆弘, 斎藤成一: 「通信遅延を低減したプロセッサ間通信方式の提案」, 情報処理学会 HPS 論文誌 第 1 巻 第 2 号, pp.28-38, 2000.

[6] 青山功, 佐藤裕幸, 中島克人:「介護サービススケジューリング問題への遺伝的アルゴリズムおよびタブーサーチの適用とその比較」, Journal of Operations Research Society of Japan, 第 44 巻 第 3 号, P.262-280, 2001.

[7] 和泉秀幸, 中島克人, 佐藤裕幸: 「SAR 画像再生処理の高速化–キャッシュアクセスを考慮したコーナーターンの 1 改善法」, 情報処理学会論文誌 第 44 巻 第 6 号, pp.1525-1537, 2003.

国際会議論文

[8] H.Sato, M.Kawamura:”Towards a Parallel Database Management System”, 日米AIシンポジウム’89, 1989.

[9] H.Sato, I.Aoyama, H.Asami, K.Nakajima, H.Sakamoto, M.Tsuchiya:”A Particle Treatment Scheduling System using Two Genetic Algorithms”, In proc. of Applied Informatcs ’99, pp.346-348, 1999.

[10] H.Sato, T.Nakagawa, K.Yoda, K.Nakajima, H.Sakamoto, M.Endo:”SPEED-UP OF RADIATION TREATMENT PLANNING USING A LINUX ALPHA CLUSTER”, In proc. of 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2000), Vol. 1, pp.1-6, 2000.

査読付国内会議

[11] 佐藤裕幸, 越村三幸, 近山隆, 藤瀬哲朗, 松尾正浩, 和田久美子: 「PIMOSの資源管理方式」, 並列処理シンポジウムJSPP’89, pp.267-274, 1989.

[12] 寿崎かすみ, 佐藤裕幸, 杉村領一, 赤坂宏二, 瀧和男, 山崎重一郎, 弘田直人:「マルチPSIにおける並列構文解析プログラムPAXの実現および評価」, 並列処理シンポジウムJSPP’89, pp.343-350, 1989.

[13] 佐藤令子, 佐藤裕幸, 中島克人:「疎結合型マルチプロセッサ上の拡散型動的負荷分散方式=LLS-G方式=」, 並列処理シンポジウムJSPP’93, pp.363-369, 1993.

口頭発表

[14] 佐藤裕幸, 太細孝, 溝口徹夫: 「漢字Prologのデータベースインタフェースの実現について」, 情報処理学会第27回全国大会, 4D-7, pp.1147-1148, 1983.

[15] 佐藤裕幸, 太細孝, 溝口徹夫: 「コピー方式によるProlog処理系の実現」, 情報処理学会第28回全国大会, 2G-8, pp.1037-1038, 1984.

[16] 佐藤裕幸, 市吉伸行, 太細孝, 宮崎敏彦, 竹内彰一: 「Concurrent Prologのシーケンシャルインプリメンテーション –deep binding方式による多環 境の実現–」,

日本シフトウェア科学会第 1 回大会, 3D-3, pp.299-302, 1984.

[17] 佐藤裕幸, 渡辺久晃, 堀敦史, 上田尚純, 近山隆: 「SIMPOS のシステムトレーサ」, 情報処理学会第 29 回全国大会, 4E-10, pp.303-304, 1984.

[18] 佐藤裕幸, 堀敦史, 溝口徹夫, 坂井公:「SIMPOS のプログラミング・システムーエディター」, 情報処理学会第 30 回全国大会, 4E-3, pp.299-300, 1985.

[19] 坂井公, 堀敦史, 佐藤裕幸:「SIMPOS のプログラミング・システムートランスデューサー」, 情報処理学会第 30 回全国大会, 4E-4, pp.301-302, 1985.

[20] 佐藤裕幸, 坂井公:「演算子文法用汎用構造エディタ」, 第 27 回プログラミングシンポジウム, pp.63-73, 1986.

[21] 佐藤裕幸, 近山隆, 石橋弘義, 吉田かおる, 内田俊一: 「SIMPOS（逐次型推論マシン PSI の OS） –ユーザ・インタフェースとその特徴–」, 情報処理学会マルチメディア通信と分散処理 研究報告 86-MDP-31, pp.1-8, 1986.

[22] 佐藤裕幸, 白須裕之, 堀敦史:「SIMPOS のプログラミング環境 –テキスト・エディタ Pmacs–」, 情報処理学会第 33 回全国大会, 4D-2, pp.409-410, 1986.

[23] 佐藤裕幸, 近山隆, 杉野栄二, 瀧和男:「PIMOS の概要ー並列推論マシン用オペレーティング・システムの構築ー」, 情報処理学会第 34 回全国大会, 2P-8, pp.127-128, 1987.

[24] 佐藤裕幸, 越村三幸, 近山隆, 瀧和男:「並列論理型 OS − PIMOS」, 情報処理学会第 35 回全国大会, 4D-3, pp.259-260, 1987.

[25] 堀敦史, 近山隆, 瀧和男, 佐藤裕幸, 佐藤令子, 和田久美子: 「PIMOS の入出力管理」, 情報処理学会第 36 回全国大会, 2D-2, pp.273-274, 1988.

[26] 佐藤裕幸, 越村三幸, 近山隆, 藤瀬哲朗, 松尾正浩, 和田久美子: 「PIMOS の設計方針」, 情報処理学会第 37 回全国大会, 5P-1, pp.247-248, 1988.

[27] 佐藤裕幸, 星田昌紀, 近山隆, 藤瀬哲郎:「PIMOS の評価」, 情報処理学会第 39 回全国大会, 2P-7, pp.1189-1190, 1989.

[28] 佐藤裕幸:「並列自然言語構文解析システム PAX の改良」, KL1 プログラミングワークショップ'90, 1990.

[29] 佐藤令子, 佐藤裕幸,: 「疎結合型マルチプロセッサ上の拡散型負荷分散の一方式」, 電子情報通信学会コンピュータ研究会 (CPSY) 技術報告 CPSY92-9, pp.1-8, 1992.

[30] 河村元夫, 佐藤裕幸, 横田一正:「並列データベース管理システム Kappa-P の概要」, 情報処理学会第 45 回全国大会, 5R-3, pp.4-173-174, 1992.

[31] 永沼和智, 佐藤裕幸, 河村元夫:「Kappa-P の並列問い合わせ処理」, 情報処理学会第 45 回全国大会, 5R-4, pp.4-175-176, 1992.

[32] 坂下之子, 合田光宏, 佐藤裕幸, 河村元夫:「Kappa-P のネームサーバ機能」, 情報処理学会第 45 回全国大会, 5R-5, pp.4-177-178, 1992.

[33] 椿野宣行, 佐藤裕幸, 河村元夫:「Kappa-P のトランザクション制御」, 情報処理学会第 45 回全国大会, 5R-6, pp.4-179-180, 1992.

[34] 川村達, 佐藤裕幸, 河村元夫:「Kappa-P のアンネスト／ネスト処理」, 情報処理学会第 45 回全国大会, 5R-8, pp.4-183-184, 1992.

[35] 中嶋かおり, 佐藤裕幸, 河村元夫:「Kappa-P の単一レコードアクセス機能」, 情報処理学会第 45 回全国大会, 5R-9, pp.4-185-186, 1992.

[36] 佐藤裕幸, 川上かおり:「並列 C++言語システム ANUFO の並列処理機構」 電子情報通信学会コンピュータ研究会 (CPSY) 技術研究報告 CPSY94-83, pp.81-88, 1994.

[37] 川上かおり, 佐藤裕幸:「並列 C++言語システム ANUFO の同期機構」 電子情報通信学会コンピュータ研究会 (CPSY) 技術研究報告 CPSY94-83, pp.89-96, 1994.

[38] 白石將, 佐藤裕幸, 中島克人:「WS クラスタ上の並列ジョブ支援ツールの試作」, 情報処理学会第 51 回全国大会, 3P-7, pp.6-117-118, 1995.

[39] 白石將, 佐藤裕幸, 中島克人: 「分散型並列処理支援ツール ParaJET」, 電子情報通信学会コンピュータ研究会 (CPSY) 技術報告 CPSY96-60, pp.23-30, 1996.

[40] 青山功, 佐藤裕幸, 中島克人:「ニューラルネットワークを用いた訪問看護スケジューリングシステム」, 情報処理学会第 54 回全国大会, 2M-07, pp.2-317-318, 1997.

[41] 川上かおり, 中島克人, 佐藤裕幸, 瀧寛和, 後藤明広:「GA を用いた LSI マルチワイヤリング最適設計ツールの実装」, 情報処理学会第 55 回全国大会, 4AG-8, pp.2-483-484, 1997.

[42] 佐藤裕幸, 川上かおり, 白石將, 中島克人, 後藤明広: 「分散型並列パラメタサーベイ支援ツール ParaVEY」, 情報処理学会第 55 回全国大会, 2G-04, pp.1-60-61, 1997.

[43] 川上かおり, 後藤明広, 瀧寛和, 中島克人, 佐藤裕幸:「LSI パッケージリードフレーム設計への GA の適用」, 情報処理学会数理モデル化と問題解決研究会 人工生命とその応用シンポジウム論文集, pp.87-94, 1997.

[44] 佐藤裕幸, 青山功, 浅見廣愛, 中島克人, 坂本豪信, 土谷昌晴, 菅靖則:「粒子線治療装置スケジューリングシステムーシステム概要ー」, 情報処理学会第 56 回全国大会, 4W-01, pp.2-377-378, 1998.

[45] 青山功, 佐藤裕幸, 浅見廣愛, 坂本豪信, 土谷昌晴, 菅靖則:「粒子線治療装置スケジューリングシステムー治療日スケジュールー」, 情報処理学会第 56 回全国大会, 4W-02, pp.2-379-380, 1998.

[46] 浅見廣愛, 佐藤裕幸, 青山功, 坂本豪信, 土谷昌晴, 菅靖則:「粒子線治療装置スケジューリングシステムー治療順スケジュールー」, 情報処理学会第 56 回全国大会, 4W-03, pp.2-381-382, 1998.

[47] 青山功, 佐藤裕幸, 浅見廣愛, 土谷昌晴, 坂本豪信:「粒子線治療装置スケジュールへの GA の適用　治療日スケジュール」, 情報処理学会数理モデル化と問題解決研究会研究報告, No.21-008, pp.43-48, 1998.

[48] 浅見廣愛, 佐藤裕幸, 青山功, 坂本豪信, 土谷昌晴,:「粒子線治療装置スケジュールへの GA の適用　治療順スケジュール」, 情報処理学会数理モデル化と問題解決研究会研究報告, No.21-009, pp.49-54, 1998.

[49] 佐藤裕幸, 永田真也, 中島克人, 福島康之:「分散型並列パラメータサーベイ支援ツール ParaVEY の熱解析への適用」, 情報処理学会第 58 回全国大会, 3D-02, pp.1-3-4, 1999.

[50] 中島克人, 森伯郎, 佐藤裕幸, 高橋勝己, 浅見廣愛, 水上雄介, 飯田全広, 新留勝広:「FPGA ベース並列マシン RASH の概要」, 情報処理学会第 58 回全国大会, 1H-08, pp.1-141-142, 1999.

[51] 浅見廣愛, 佐藤裕幸, 飯田全広, 森伯郎, 中島克人:「FPGA ベース並列マシン RASH のシステム機能と構成」, 情報処理学会第 58 回全国大会, 1H-09, pp.1-143-144, 1999.

[52] 飯田全広, 水上雄介, 高橋勝己, 浅見廣愛, 佐藤裕幸:「FPGA による並列暗号解析装置の構成 (1)-DES 暗号等の鍵探索-」, 情報処理学会第 58 回全国大会, 5N-08, pp.3-337-338, 1999.

[53] 佐藤裕幸, 中川隆文, 依田潔, 中島克人, 室井克信, 坂本豪信, 土谷昌晴, 遠藤真広:「ワークステーションクラスタを用いた放射線治療計画の高速化ーシステム概要と予備評価実験ー」, 情報処理学会第 59 回全国大会, 1M-07, pp.1-107-108, 1999.

[54] 中川隆文, 依田潔, 中島克人, 佐藤裕幸, 室井克信, 坂本豪信, 土谷昌晴, 遠藤真広:「並列計算機を用いた治療計画の高速処理システムの設計」, 第 77 回日本医学放射線物理学会大会資料, 日本医学放射線物理学会, No.60, pp.78, 1999.

[55] 青山 功, 佐藤裕幸:「介護サービススケジューリングへの GA の適用」, 情報処理学会数理モデル化と問題解決研究会研究報告, No. 26-002, pp.5-8, 1999.

[56] 佐藤裕幸, 中川隆文, 依田潔, 中島克人, 室井克信, 坂本豪信, 遠藤真広:「Linux Alpha クラスタを用いた放射線治療計画の高速化」, 第 7 回" ハイパーフォーマンスコンピューティングとアーキテクチャの評価" に関する北海道ワークショップ (HOKKE-2000), 情報処理学会研究報告 (2000-ARC-137), pp.143-148, 2000.

[57] 佐藤裕幸, 中川隆文, 依田潔, 中島克人, 坂本豪信, 遠藤真広:「ワークステーションクラスタを用いた放射線治療計画の高速化ー線量分布計算と再構成 CT 画像生成の評価実験ー」, 情報処理学会第 61 回全国大会, 2D-03, pp.1-5-6, 2000.

[58] 浅見廣愛, 飯田全広, 中島克人, 森伯郎, 佐藤裕幸, 高橋勝己:「FPGA ベース並列マシン RASH における TMTO 法暗号解析の実装 (1)〜実装手法〜」, 情報処理学会第 62 回全国大会, 2S-07, pp.3-309-310, 2001.

[59] 佐藤裕幸, 中島克人:「多目標追尾アルゴリズム航跡型 MHT の並列化ー解候補生成の並列化とその評価ー」, 2001 年並列／分散／協調処理に関する『沖縄』サマー・ワークショップ (SwoPP「沖縄」2001), 情報処理学会研究報告 (2001-HPC-87), pp.55-60, 2001.

[60] 青山功, 中島克人, 佐藤裕幸:「人工衛星運用スケジューリングへの遺伝的アルゴリズムの適用」, 数理モデル化と問題解決研究会研究報告, No.036-005, pp.17-20, 2001.

[61] 今井照久, 竹内秀樹, 高根沢眞紀, 和泉秀幸, 佐藤裕幸:「SAR 画像の超解像処理 (非線形帯域拡張法) の並列化」情報処理学会第 63 回全国大会, 5H-6, pp.1-11-12, 2001.

[62] 和泉秀幸, 佐々木和司, 佐藤裕幸:「SAR 画像再生処理の並列化～改善したコーナーターン法の適用評価～」, 情報処理学会第 63 回全国大会, 5H-7, pp.1-13-14, 2001.

[63] 青山功, 田中秀俊, 白石將, 川上かおり, 佐藤裕幸:「多峰性関数に対する最適化アルゴリズムの探索性能比較」, 情報処理学会第 63 回全国大会, 2P-4, pp.2-87-88, 2001.

[64] 佐藤裕幸, 浅見廣愛, 中島克人, 森伯郎, 飯田全広, 山田美和, 澤田一郎, 角田淳一:「FPGA ベース並列マシン RASH によるパスワード解析」, 情報処理学会第 63 回全国大会, 6K-7, pp.1-55-56, 2001.

[65] 浅見 廣愛, 飯田 全広, 佐藤裕幸, 中島克人, 森伯郎:「FPGA ベース並列マシン RASH の改良検討-RASH2 のハードウェア構成-」, 情報処理学会第 64 回全国大会, 1ZB-01, pp.1-4-5, 2002.

[66] 和泉秀幸, 竹内秀樹, 佐藤裕幸:「SAR 超解像処理の並列化検討」, 情報処理学会第 64 回全国大会, 4ZB-6, pp.1-87-88, 2002.

[67] 今井照久, 水野政治, 金子智巳, 佐藤裕幸:「リアルタイムシステムの高速インターコネクトに関する検討ー Fibre Channel 及び Myrinet の基本性能評価ー」, 情報処理学会第 64 回全国大会, 3ZB-02, pp.1-65-66, 2002.

[68] 青山功, 羽下誠司, 佐藤裕幸:「応答曲面法を適用した最適パラメータ探索」, 情報処理学会第 64 回全国大会, 3W-01, pp.1-27-28, 2002.

[69] 和泉秀幸, 竹内秀樹, 佐藤裕幸:「画像分割による MUSIC 超解像処理の高速化」, 第 1 回 情報科学技術フォーラム (FIT2002), A-14, pp.27-28, 2002.

[70] 和泉秀幸, 竹内秀樹, 佐藤裕幸:「MUSIC 超解像処理の高速化～画質評価法の提案～」, 電子情報通信学会宇宙・航行エレクトロニクス研究会 (SANE) 技術研究報告 SANE2002-85, pp.53-58, 2003.

[71] 浅見廣愛, 山岸陽, 今井照久, 滝本哲也, 天野一彦, 中川雅博, 佐藤裕幸:「画像処理プラットフォーム RASH-IP の構成」, 第 10 回”ハイパーフォーマンスコンピューティングとアーキテクチャの評価”に関する北海道ワークショップ (HOKKE-2003), 情報処理学会研究報告 (2003-ARC-152), pp.139-143, 2003.

[72] 佐藤裕幸, 中川隆文, 東誠一, 遠藤真広:「ワークステーションクラスタを用いた放射線治療計画の高速化ービーム設計計算と通信評価実験ー」, 情報処理学会

第 65 回全国大会, 5F-2, pp.1-25-26, 2003.
[73] 浅見廣愛, 山岸陽, 今井照久, 滝本哲也, 天野一彦, 中川雅博, 佐藤裕幸:「画像処理プラットフォーム RASH-IP(1) ーハードウェア構成ー」, 情報処理学会第 65 回全国大会, 5G-1, pp.1-49-50, 2003.
[74] 今井照久, 滝本哲也, 浅見廣愛, 山岸陽, 天野一彦, 中川雅博, 佐藤裕幸:「画像処理プラットフォーム RASH-IP(2) ー画像処理の手法ー」, 情報処理学会第 65 回全国大会, 5G-2, pp.1-51-52, 2003.