数理解析研究所講究録
945巻 1996年 130-140

完全分散型並列分枝限定法システムのアーキテクチャと負荷分散

東京理科大学　品野　勇治　SHINANO Yuji
東京理科大学　平林　隆一　HIRABAYASHI Ryuichi

1. はじめに

分枝限定法はクラスNPに属する組合せ最適化問題を解くための代表的な解法である. 特定の問題の特徴や性質を利用した下界値計算法や分枝変数の選択を工夫することで, 各種問題毎に解ける問題規模は拡大してきている. しかし, 分枝限定法を用いても問題規模がある規模以上になると, 現実的な時間内に解けるインスタンスの数は極端に少なくなることが知られている.

一般的な枠組みとしての分枝限定法は, 十分に研究され列挙法に基づく計算原理として確立している. 並列分枝限定法は, 多数のプロセッサを利用し並列に解空間を列挙することで, 少しでも現実的な時間内に解けるインスタンスを増やすための試みである.

計算量が問題規模に対して指数的に増加する組合せ最適化問題に並列処理を適用し, 数倍の計算時間の短縮が可能となっても, 本質的に解ける問題規模を拡大するとは考え難い. しかし, 一定規模の問題までは必ず解けるような解法は, クラスNPに属する問題では開発できないと考えられている. 規模の大きな問題でも, 少数のインスタンスに限れば分枝限定法によって, 現実的な時間内に解かれている. 並列処理の適用は, この現実的な時間内に解けるインスタンスの量を少しでも拡大するための手段である.

本研究では, 筆者らが開発し実験をしてきたマスター・スレーブ型並列分枝限定法システムで得た知見に基づき, 完全分散型並列分枝限定法システムのアーキテクチャと負荷分散の仕組みについて提案する.

2. マスター・スレーブ型並列分枝限定法システム

並列分枝限定法が実際に実装され, 数値実験が実施され始めたのは1980年代後半からである[3]. 当初, 実装されたものは, 実装の容易性によりマスター・スレーブ型のものが多い. 筆者らも, マスター・スレーブ型のシステムを構築した. 筆者らが構築したシステムは, それまでに多かった特定の問題のための並列分枝限定法ではなく, 汎用ツールとして実装した. 開発したマスター・スレーブ型の分枝限定法並列化ツール[2] は, ワークステーション群上に実装され, 以下の特徴を持った.

- 並列分枝限定法のための汎用ツールであり, 特定の問題に依存しない並列分枝限定法のスケルトンを提供する.
- 割り込み機構を利用した迅速な限定操作を実現する.
- 深さ優先探索と下界値優先探索を組み合わせたハイブリッド探索を実装している.

マスター・スレーブ型分枝限定法並列化ツールにより, 比較的簡単に各種問題に対する並列分枝限定法を記述できた. そこで, 巡回セールスマン問題, 整数計画問題, 輸送制約付き枝巡回路問題を実装し, 数値実験を実施した. それぞれの実験結果において, 多数の超線形加速が観測され, 顕著な並列化の効果が示された.

しかし, マスター・スレーブ型の実装では, 子問題群を管理するための記憶容量は依然1台の計算機の限界を超えない. したがって, 計算時間が短縮されたとしても, これまでに解けなかったインスタンスを解くことに対して貢献するものではなかった.

3. 完全分散型並列分枝限定法システムの設計思想

以下の設計思想に基づいて開発された.

3.1. 分枝限定法の枠組みを提供する

並列分枝限定法の実装を, 本質的に問題固有のルーチンのみを記述することで実現する. 特に, 各問題毎の実装を行うユーザの観点からは, 逐次の分枝限定法としての動きが, 明確に意識できるスケルトンを提供する.

3.2. 並列分枝限定法実行環境と下界値計算法など各問題固有のルーチンの開発環境を分離する

マスター・スレーブ型のツールでは，問題固有のアルゴリズムの実装に際して，複数プロセスが動作する環境でデバッグを行う必要があり，依然開発は困難なものであった．そこで，問題固有の下界値計算等のルーチンの開発は，並列動作環境とは独立に逐次処理の分枝限定法として開発できる環境を提供する．問題固有のルーチンのデバッグが完了した時点で，再コンパイル程度の手間で並列分枝限定法の動作環境への移行を可能にする．

3.3. プロセッサの演算能力だけでなく，メモリの有効利用も図る

並列分枝限定法では，計算途中に保持すべき子問題の数が，逐次処理する分枝限定法と比較して増加する傾向がある．現実に解けるインスタンスを増やすためには，計算途中に保持すべき子問題群を管理できるだけのメモリ容量の確保が必要である．マスター・スレーブ型では，スレーブが割り当てられたプロセッサ側のメモリには子問題群を保持しないため，スレーブ側のメモリの有効利用が図られていない．完全分散型では，各プロセッサがローカルに子問題群を保持し，プロセッサの数を増やすことで，計算途中に保持できる子問題数を増やせるようにする．

3.4. 特定の計算機環境に依存せずに動作する

並列計算を行うための計算機環境を選ばないシステムであれば，実際に並列計算を行える環境の入手が容易となる．今日では，標準のメッセージ・パッシング・ライブラリを使用することで，特定の計算機環境に依存しない並列処理システムの構築が可能である．

3.5. 動作環境を並列分枝限定法の実行を停止することなく，動的に変更できる

近年，コンピュータ・ネットワークは，飛躍的に拡大してきている．日時を指定したり，誰も使用していない間という条件のもとで，計算機環境を入手できれば使用可能なプロセッサ数の増加が望める．これまでに解けなかったインスタンスを解くという観点からは，利用可能なプロセッサをできる限り入手し，並列分枝限定法を実行する必要がある．

また，効率の観点からも，計算に必要なだけの資源を動的に追加・削除できることのメリットは大きい．このことは，分枝限定法を数台のプロセッサで実行させ，必要に応じて，プロセッサを追加しながら解くことを可能にする．さらに，特定の計算機環境に依存しないシステムであれば，必要に応じて超並列計算機の追加をも可能となる．

3.6. 1000 個以上のプロセッサを使用した場合でも，負荷分散が適当に機能する

完全分散型のシステムでは，マスター・スレーブ型でボトルネックとなった，マスターへの処理の集中や，子問題を管理するためのメモリ容量不足は解消される．一方，マスター・スレーブ型での限定操作が有効に機能したのは，子問題群を集中管理することで，解の探索規則を完全に制御できたことによる効果が大きい．

完全分散型のシステムでは，各計算機がローカルに子問題群を保持するため，ある計算機では子問題の枯渇や，あふれが生じることもあり，解の探索規則を完全に制御することはできない．また，枯渇やあふれに対応するように，子問題をプロセッサへの割り当てる負荷分散の仕組みも複雑になる．さらに，限定操作が機能するタイミングのずれも，マスター・スレーブ型と比較すると大きくなる．そのため，一般に完全分散型の並列分枝限定法の実装では，マスター・スレーブ型ほど顕著な計算時間の短縮は得られていない．

しかし，これまでに解けなかったインスタンスを解くという観点からは，完全分散型のシステムが不可欠である．完全分散型システムでの効果は，負荷分散の仕組みに依存する．しかも，効果は実装と数値実験によってのみ検証される．よって，1000 個以上のプロセッサを使用しても，負荷分散が適当に機能するシステムを目標とし，現実的には負荷分散の仕組みを実装する部分の独立性を保ち，負荷分散の仕組みを変更可能なシステムを構築する．

4. 完全分散型並列分枝限定法システムのアーキテクチャと負荷分散

4. 1. システムの概要

上述した設計思想に基づいたシステムの実装を，C++言語とPVM(Parallel Virtual Machine)[1]を使用して行った．基底クラスを本ツールが提供し，各問題固有部分は解法毎に派生クラスとして記述する．よって，ツールにより提供される部分から，問題固有部分の記述が明確に分離された．現在の動作確認はワークステーション群上で行っているが，標準メッセージ・パッシング・ライブラリ PVM の利用により，計算機環境からも独立したシステムとして実装されている．さらに，負荷分散の機構は，別タスクの機能として分離することで，改善の余地を残している．

4. 2. アーキテクチャ

アーキテクチャを抽象的な構成を示すオブジェクト構成と，具体的に非同期で動作するプログラムの構成であるタスク構成の側面から説明する．

4. 2. 1. オブジェクト構成

システム構成を規定する主要オブジェクトは以下である．

① Problem Manager オブジェクト

分枝限定法が実行されている過程における問題とその状態を抽象化したオブジェクトである．このオブジェクトは，分枝限定法実行中の問題の状態として，未分枝の全ての子問題（子問題の入れ物を抽象化したSubproblem Pool オブジェクト内に保持される），暫定解，暫定値を保持している．また，このオブジェクトに対しては，以下のような演算子が定義されている．

- ●getIncumbentValue:暫定値を取り出す．
- ●getSubproblem:子問題を取り出す．
- ●putSubproblem:子問題を追加する．
- ●putSolution:解を追加する．

② Load Balancer オブジェクト

負荷分散の仕組みを抽象化したオブジェクトである．このオブジェクトは分枝限定法の汎用の枠組み外に存在し，負荷分散のための意志決定を行う．

③ Solver オブジェクト

分枝限定法の解法を抽象化したオブジェクトである．分枝限定法の枠組みのスケルトンは，このオブジェクトの solve 演算子の実装部に存在する．

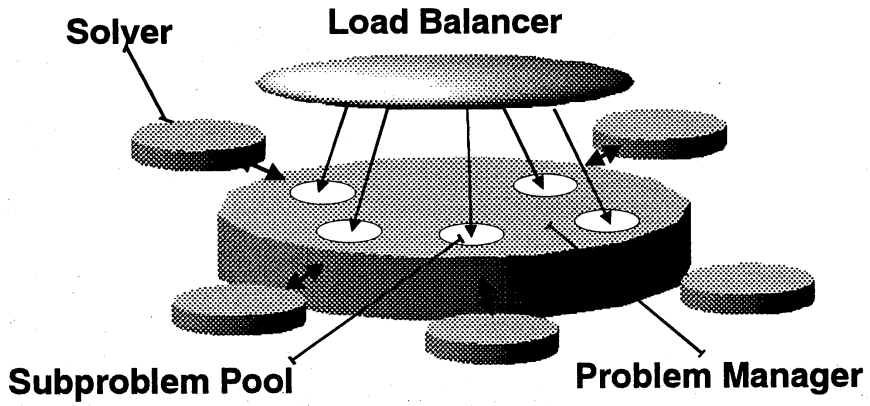

図 1. オブジェクト構成

図 1にこれらのオブジェクトの関係を示す．Solver オブジェクトは使用するプロセッサ数に対応して存在する．Problem Manager オブジェクト内には，Solver オブジェクトと１対１に対応するSubproblem

Pool オブジェクトがある．Load Balancer オブジェクトは，Subproblem Pool が枯渇したり，あふれたりすることを避けるように，子問題の移動元，異動先の Subproblem Pool を決定し，その移動の指示を与える．実際の移動は Problem Manager 内で起こり Load Balancer オブジェクトを経由することはない．

このようなオブジェクト構成を導入する理由は，設計思想の3.2.で示した実行環境を開発環境から分離するためである．開発環境を逐次処理の分枝限定法のスケルトンとして提供するには，分枝限定法のスケルトンから並列化に伴う処理部分を独立させる必要がある．スケルトンは，主に Solver オブジェクトの solve 演算子の実装部によって提供される．上記のようなオブジェクト構成を持つことで，Solver オブジェクトから並列化に関する部分は完全に分離される．なぜなら，Solver オブジェクトは，対応する Subproblem Pool とデータを受け渡すのではなく，Problem Manager に対してデータを受け渡す．したがって，Subproblem Pool を１つしか持たず，Load Balancer と交信を行わない逐次処理用の Problem Manager を用意することで逐次処理用の開発環境が提供できる．この構成により，Solver オブジェクト内に実装される問題固有部分のプログラム・コードは，１行も変更せずに逐次処理，並列処理の動作環境の両方で動作可能となる．

上記ののオブジェクト間を以下のオブジェクトがやり取りされることで，並列分枝限定法が実行される．

① InitData オブジェクト

分枝限定法の開始前に設定すべき全ての初期データを抽象化したオブジェクトである．以下のような演算子が定義されている．

●getProblemType:解くべき問題のタイプ（最大化，最小化）を取り出す．

●getInitialSolution:初期解を取り出す．

●getRootProblem:分枝木でのルートに位置する問題を取り出す．

② Subproblem オブジェクト

子問題を抽象化したオブジェクトである．並列化でのデータ転送量を縮小するため，子問題固有に変更される内容だけを保持し，実際の子問題の表現は InitData オブジェクトとの共用によって実現する．

③ Solution オブジェクト

解を抽象化したオブジェクトである．並列化に伴うデータ転送量を縮小するため，InitData オブジェクトの内容を共用することによって解を表現するのに十分な内容だけを保持する．

Load Balancer オブジェクトを除く，上記全てのオブジェクトは基底クラスがツールによって提供される．また，Problem Manager については，機械的なコード生成により派生クラスが定義できるので，マクロ処理によりコードを自動生成する．よって，ツールのユーザは，その他のクラスの派生クラスを記述することにより，ユーザ固有の解法に関する実装を行う．たとえば，TSP の場合には，Solver クラスから派生する TSPSolver，InitData クラスから派生する TSPInitData，Subproblem クラスから派生する TSPSubproblem クラス，Solution クラスから派生する TSPSolution クラスを記述する．

4. 2. 2. タスク構成

以下の４つのタスクにより構成される．

① Problem Manager タスク

② Load Balancer タスク

③ Solver Task タスク（対応する Load Balancer タスクと同じプロセッサ上で動作する）

これらは，主要オブジェクトと対応しているわけではない．図2に，タスク構成をオブジェクト構成と関

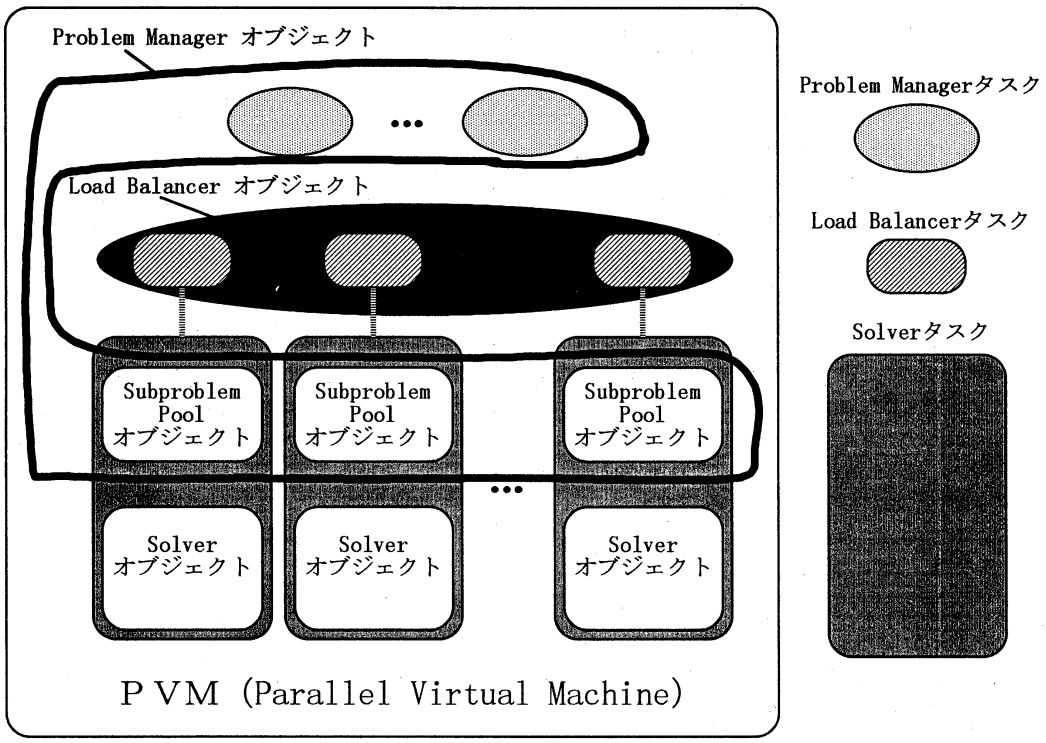

図2.タスク構成

連付けて示す．複数のタスクで構成されるProblem Managerオブジェクトと Load Balancerオブジェクトをタスク構成の観点から述べる．

Problem Managerオブジェクトは，複数のタスクで構成される1つのオブジェクトとして実装される．Problem Managerタスクは，暫定解に関する情報のみを管理し，子問題群は全てSolverタスクで管理される（ただし，ルートになる問題のみは，Problem Managerタスクが保持する）．また，子問題群の受け渡しもSolverタスク間で行われ，受け渡しの実行に際してProblem Managerタスクの介入はない．つまり，並列分枝限定法は，完全に等価に分散しているSolverタスクが協調しながら実行される．

ここで，協調を助けるのがオブジェクト構成で述べた Load Balancer オブジェクトである．Load Balancerオブジェクトも，複数のLoad Balancerタスクで構成される1つのオブジェクトとして機能する．各Load Balancerタスクは，計算を実行中に存在するSolverタスクと1対1に対応して存在する．Load Balancerタスクは，，周りのLoad Balancerタスクと協調し，負荷分散に関する意思決定を行う．このように，負荷分散の意思決定機構を完全に分離することで，実装した負荷分散が適当に機能しない場合には，新たな仕組みの実装が可能となる．

最後に，逐次処理の開発環境を生成する方法について述べる．開発環境として，逐次処理の分枝限定法のスケルトンを提供する時には，Load Balancerタスクは全く不要であるので取り除く．さらに，Problem Managerタスクもタスクとして分離せずSolverタスクへ吸収する．以上の操作で，開発するほとんどのルーチンを変更することなく，逐次処理の分枝限定法の実行ファイルを構成できる．

4.3. 負荷分散の概要

負荷分散を行うために，Subproblem Poolに関する以下のパラメタを起動時に与える．

●Subproblem Pool標準サイズ

1つのSolverタスク内に保持される標準子問題数である．Solverタスクが保持するSubproblem Poolサイズが一定でない環境では，子問題数の比較はこの値から正規化された値で行われる．

●Subproblem Poolサイズ

実際にSolverタスクを動作させるプロセッサ上に実装されたメモリ容量から見積もられる保持可能な子問題数である．この値により，Subproblem Poolのあふれを検出する．

●Threshold値

ここに指定される値以下となったら，子問題の枯渇が予想されることを意味する閾値である．この値以下となると，子問題の送り手を探す処理が始まる．

●Start Calculation(%)

Subproblem Poolにあふれが生じると，Subproblem Pool内の子問題を，他のSolverタスクへ転送する．あふれを生じたSolverタスクは計算を停止するが，Subproblem Poolの使用率がここに指定される比率を下回ると，転送は続けられるが分枝限定法の計算は再開される．

まず，Solverタスク間の通信について述べる．現在実装されている負荷分散は，基本的に上記の値と動作している全てのSolverタスクが保持している子問題数から計算された，1つのSolver当たり保持する平均子問題数によって行われる．1つのSolverタスクによって保持されている子問題数がThreshold値を下回ったとき，対応するLoad Balancerタスクは，周りのLoad Balancerタスクと交信し，送信元となるSolverタスクを探索する．送信元が決まると送信元から子問題を平均子問題数に達するまで受信し続ける．また，1つのSolverタスクによって保持されている子問題数が，Subproblem Poolサイズを越えてあふれを生じたとき，そのSolverタスクに対応するLoad Balancerタスクは，周りのLoad Balancerタスクと交信し子問題の受信先となるSolverタスクを探索する．受信先が決まると子問題数が平均子問題数に達するまで，子問題を送り続ける．その間，最初はSolverタスクでの計算は停止するが，Start Calculationで示す比率を下回って保持する子問題が減少すると，子問題の送信を行いながら計算を再開する．図3に，これらの関係を示す．

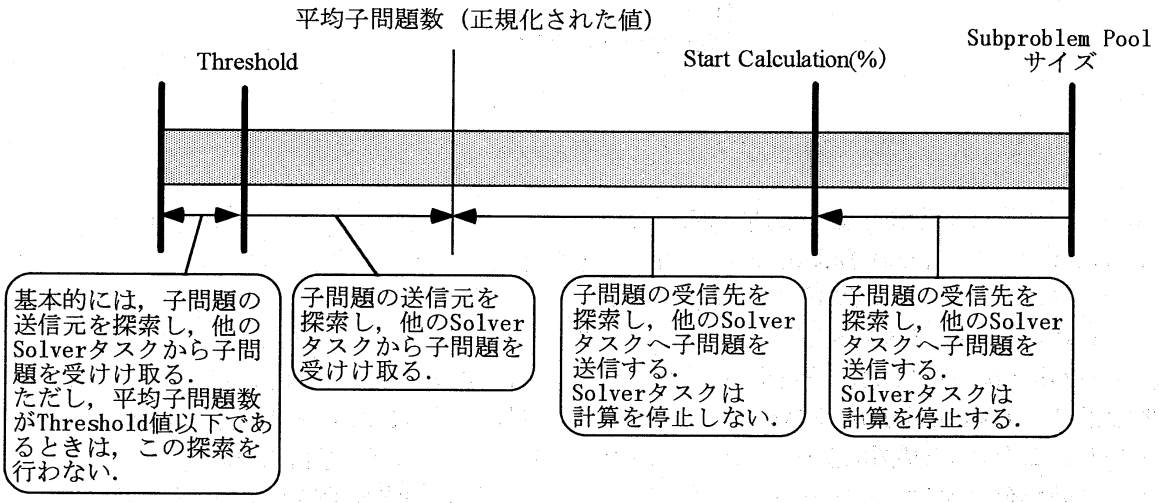

図3. 負荷分散のためのパラメタ

特に，子問題の送受信が必要となり，Load BalancerタスクがSolverタスクの探索を始める契機が，Threshold値を下回った時と，Subproblem Poolにあふれが生じたときだけであることが大切である．このことは，子問題数がThreshold値を越えて定常状態になると，他のSolverからの送信要求が来ない限

り，あふれを生じるまで積極的な子問題の送受信は起こさないことを意味する．つまり，実装している負荷分散は，基本的には子問題が枯渇したSolverタスク側から要求による負荷分散と考えられる．

次に，SolverタスクとLoad Balancerタスク間の通信について述べる．上述したパラメタを保持するのは，Load Balancerタスクである．Solverタスクは，分枝限定法の計算に専念し，1つの子問題の評価が終了する毎に，Load BalancerタスクへSubproblem Poolについての情報を伝達する．Load Balancerタスクは，その情報から判断し，SolverタスクのSubproblem Poolの状態を管理する．必要があれば，周りのLoad Balancerタスクと交信し，Solverタスクに対して子問題の送信を要求する．図4に，この通信の様子を示す．

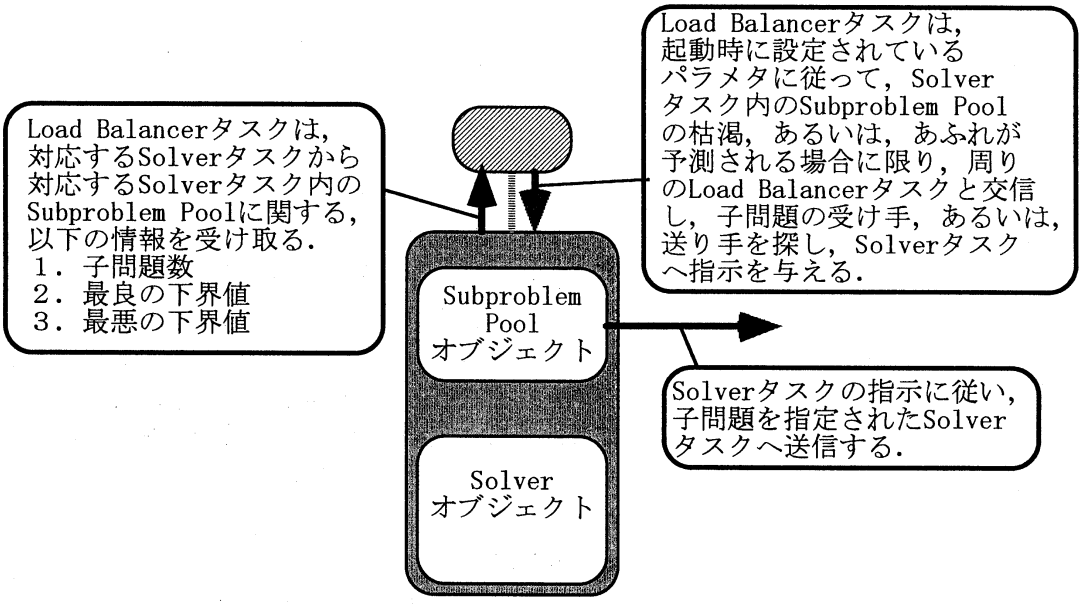

図4. Solverタスクと Load Balancerタスク間の通信

Solverタスク間の送受信では，子問題の転送が行われるので1回の送受信におけるデータ量は多い．しかし，SolverタスクとLoad Balancerタスク間の1回の送受信でのデータ量は極めて少なく，かつ，このデータ転送は同じプロセッサ内なので高速である．よって，分枝限定法で1つの子問題の評価が終了する毎のLoad Balancerタスクへのデータ転送は，全体としてのパフォーマンスは下げないと考えられる．また，Load Balancrタスクは，子問題が枯渇する前に送信元を探索し，この探索はSolverタスクとは非同期に動作するLoad Balancerタスク間で行われるため，探索の間もSolverタスクでの計算が停止することはない．このような構成により，全体としては高いスケーラビリィティを持ち1000個以上のプロセッサにより処理を行ってもパフォーマンスの低下しない完全分散型並列分枝限定法の実現が期待される．

4.4. Expanding Load Balancing Groupによる負荷分散の概念

われわれは，Expanding Load Balancing Groupによる負荷分散を提案する．上記の負荷分散の仕組みでは，全部のSolverタスクが保持している子問題の数から，1つのSolver当たりの子問題数を計算している．この部分は，明らかにボトルネックとなる．また，送信元，受信先のSolverは，データ転送を考えると遠くない方が好ましい．ここでの距離は，データ転送時の応答時間で定義するのが自然である．そこで，一定時間に応答があったSolverをまとめてGroup化し，その中での負荷分散をローカルに行い，

そこでの負荷分散に際して，送信元，受信先が見つからない場合に限り，より応答時間の長かったSolverタスクをまとめてGroupを拡張しながら，負荷分散を行うのがExpanding Load Balancing Groupによる負荷分散である．

Expanding Load Balancing Groupによる負荷分散を適用するため，4.3.で述べた負荷分散での子問題数の平均は，Expanding Load Balancing Group内での子問題数の平均として計算される．したがって，全てのSolverタスクの子問題数を収集することなく，送受信先を決定する．これは，全ての並列処理技術での最優先課題である局所化を意味する．Subproblem Poolの枯渇が予測されるときのSolverに対する，具体的な送信元Solver探索の手順を以下に記述する．

Step1. 要素数が0のLoad Balancing Groupを形成する．全Solverタスクに対応するLoad Balancerタスクに対して，メッセージをブロードキャストする．

Step2. 応答のあったLoad Balancerタスクを Load Balancing Group の要素として追加し，Load Balancing Groupを拡張する．すでに，動作中のSolverタスクが全てLoad Balancing Group内に入っていて，拡張不可なら停止（適当なSolverタスクが発見できなかった）．

Step3. Load Balancing Group内で1 Solverタスク当たりの平均子問題数を求める．

Step4. Load Balancing Groupで，現在送受信中でないSolverタスクを候補とする．

Step5. 候補の中で，最大の子問題数を保持するSolverタスクの子問題数が平均子問題数を越えていたなら，それを送信元として選択し，停止．そうでないなら，Step2.へ．

あふれに対しても手順は同様である．ただし，最小の子問題数を保持するSolverタスクを受信先として選ぶ．

局所的に，このような負荷分散を適用することで，全体として適当な負荷分散が行われることが期待される．局所的なLoad Balancing Groupが拡大する様子を図5に示す．

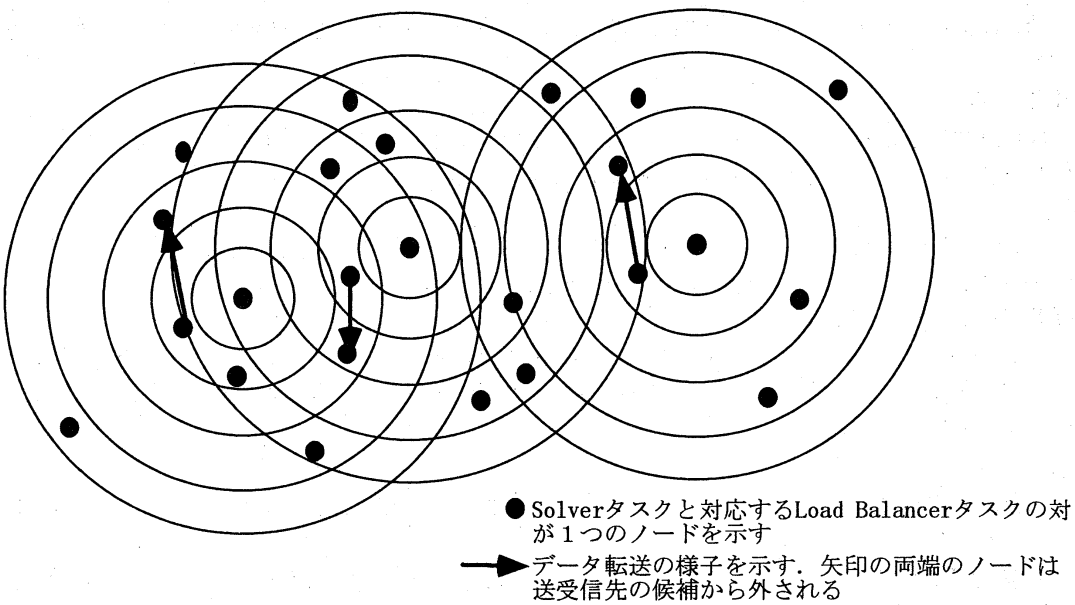

図5. Load Balancing Groupが拡大する様子

4.5. 限定操作の実装

本稿で示すアーキテクチャでは，全てのSolverタスクが子問題群の一部を保持する．したがって，1つのSolverタスクで暫定解の更新が起こると，全てのSolverタスクでSubproblem Poolの更新が必要となる．暫定解そのものは，Problem Managerタスクへ送信され，Problem Managerタスクが全体での暫

定解を保持する．限定操作は，Load Balancer オブジェクト(Load Balancer タスク群)への暫定値更新要求によって実現する．この要求は，暫定値を伴う全ての Load Balancer タスクへのブロードキャストである．

暫定値更新要求を受け取った Load Balancer タスクは，Solver タスクに割り込み暫定値を更新する．あるいは，１つの子問題の評価が終了したタイミングで暫定値の更新を行う．どちらの処理をするかは，起動時のパラメタで指定する．

このような限定操作の実装により，限定操作も等価な Solver タスクと Load Balancer タスクの組みによって実現される．Problem Manager タスクは，単に全体での暫定解を保持するだけである．

4.6. 負荷分散の詳細

Load Balancer タスクは，Solver タスクからの情報により，Solver タスクの状態を以下のように意識する．

- ●INITIALIZING
  Solver タスクが起動してから，子問題を受信できる状態になるまでの間の状態である．
- ●INITIAL_CHARGEING
  Solver タスクが起動してから，あるいは，一端 IDLE の状態になってから，子問題の受信を開始し，Threshold 値まで子問題が蓄えられる間の状態である．
- ●STABLE
  Threshold 値に達した後，他の要求を受けずにいる安定した状態である．
- ●REQUIRED_PUTTING
  他の枯渇が予測される Solver タスクの送信要求を受付け，子問題を送信中である．
- ●GETTING
  枯渇が予測されるため，他の Solver タスクから子問題を受信している．
- ●REQUIRED_GETTING
  Subproblem Pool にあふれを生じた他の Solver タスクから，受信要求を受付け，子問題を受信中である．
- ●PUTTING
  Subproblem Pool に子問題のあふれが生じ，他の Solver タスクへ子問題を送信中である．ただし，Solver タスクでは，分枝限定法の計算は続行されている．
- ●FULL
  Subproblem Pool に子問題のあふれが生じ，他の Solver タスクへ子問題を送信中，あるいは，受信先を探索中である．Solver タスクでの分枝限定法の計算は停止している．
- ●TERMINATING
  Subproblem Pool の枯渇が予測されるが，他の Solver タスクの保持している子問題数の平均が Threshold 値よりも少ないので，分枝限定法の計算そのものが終了状態にあると認識して，送信元の探索を打ち切った．
- ●IDLING
  Subproblem Pool は空である．また，送信元を探索したが送信元となる Solver タスクはなかった．
- ●KILLED
  強制的に Solver タスクの終了が要求されたために，子問題の受信先 Solver タスクを探索中，あるいは，すでに探索は完了し子問題を送信中である．
- ●EXIT_SOLVER
  Solver タスクの終了を表す．

以上の状態は，Load Balancer タスクが保持しており，Solver タスク自身はステータスを持たない．これらの状態は，Solver タスクからの情報と，周りの Load Balancer タスクからの要求によって，図 6のように状態遷移する．特に，限定操作と，Solver タスクの強制終了を実現するために状態遷移が複雑になる．

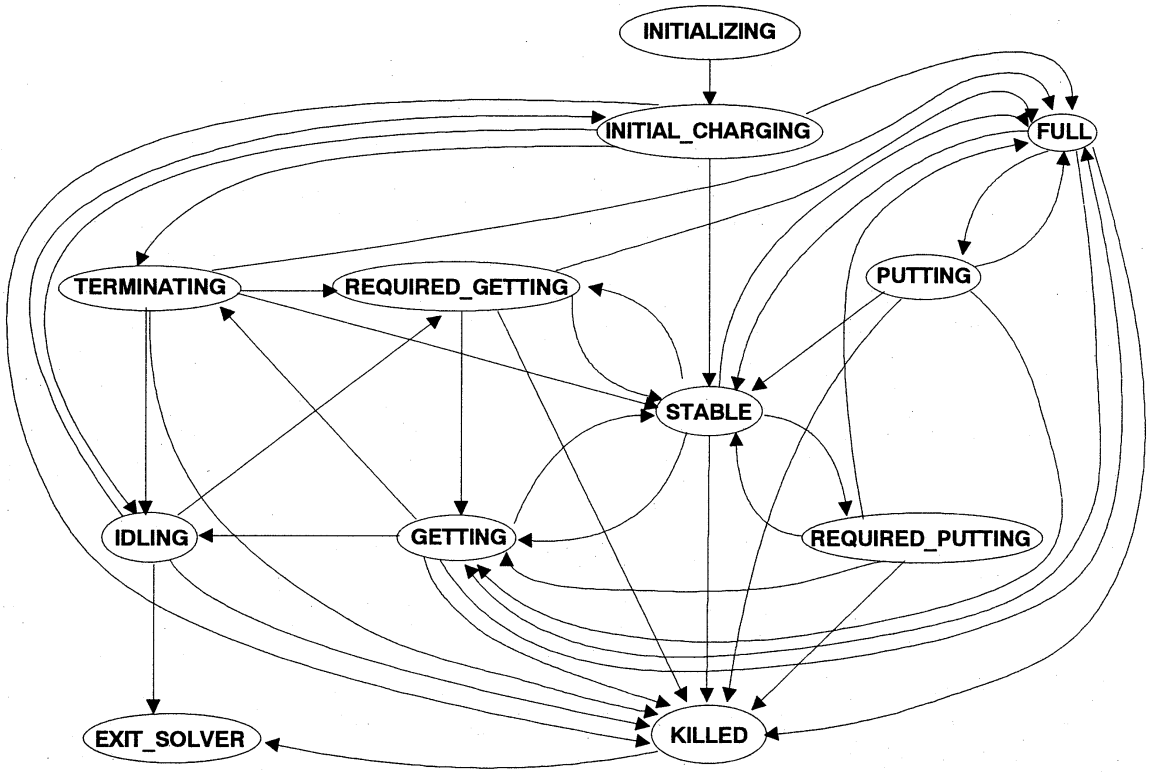

図6. Solver タスクの状態遷移

さらに，Load Balancer タスクは，Load Balancer タスク自身の状態として以下のステータスをもつ．

●SINGLE_LOAD_BALANCER_CHARGING
　Solver タスクが１つのみで，子問題を生成している．

●FREE
　Load Balancer タスクとしての処理を行っていない．

●LOCKING_TO_SEARCH_SENDER
　送信元となる Solver タスクを探索する前処理として，Load Balancing Group を形成するための情報収集中である．

●SEARCHING_TO_SENDER
　Load Balancing Group 中から，送信元となる Solver タスクを探索中である．

●MY_SOLVER_RECEIVING
　対応する Solver タスクは，子問題を受信中である．

●LOCKING_TO_SEARCH_RECEIVER
　受信先となる Solver タスクを探索する前処理として，Load Balancing Group を形成するための情報収集中である．

●SEARCHING_TO_RECEIVER
　Load Balancing Group 中から，受信先となる Solver タスクを探索中である．

●MY_SOLVER_SENDING
　対応する Solver タスクは，子問題を送信中である．

●EXITING_BALANCER
　Load Balancer タスクは終了しようとしている．

これらの状態も，Solverタスクからの情報と，周りのLoad Balancerタスクからの要求によって，図7のように状態遷移する．

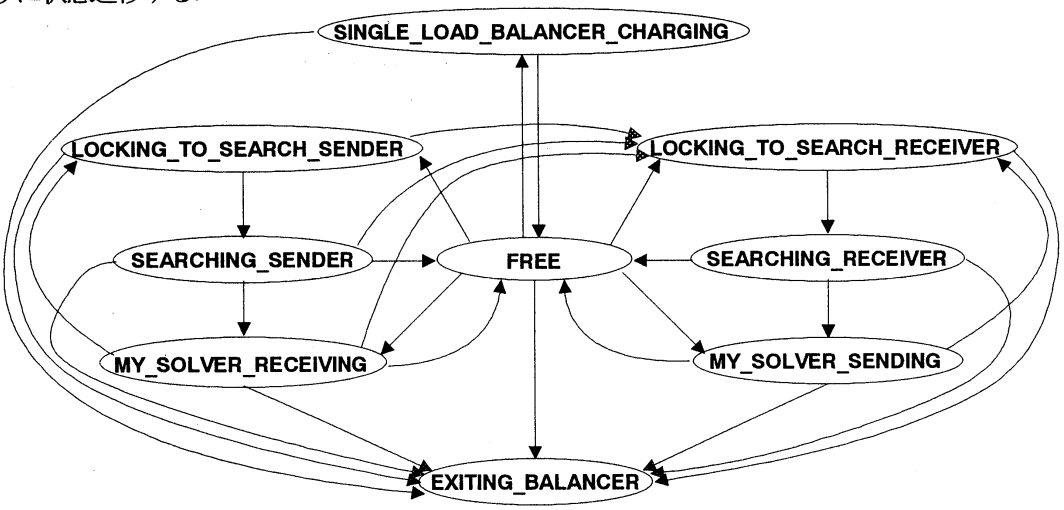

図7. Load Balancerタスクの状態遷移

4. 7. Problem Managerタスクの分散

Problem Managerタスクも複数動作させることが可能である．Problem Managerタスクは，基本的に元問題のデータと，暫定解の保持するだけである．しかし，Solverタスク初期化時に，元問題のデータをSolverタスクへ設定する必要があり，この転送はProblem Managerタスクによって行われる．元問題のデータ転送は，Solverタスクが初期化されるときに1度だけであるが，データ量は多い．Problem Managerタスクを複数起動すると，応答時間で定義される距離の一番近いProblem Managerタスクから元問題を受け取れるようになる．また，実行中にProblem Managerタスクが動作していたプロセッサの障害によって，それまでに，求められていた暫定解が消滅することを避けるためのバックアップ・タスクとして複数動作させることも考えられる．

5. おわりに

本稿では，開発中の完全分散型並列分枝限定法システムの設計思想と，その実装がどのようになされているかについて述べた．特に負荷分散に関して詳細に述べたが，現実にプログラムするには，さらに，データ送受信時の遅延に対応するための考慮が必要である．現在プログラムはデバッグ段階にあり，デバッグ終了後，代表的な組合せ最適化問題を例としたパフォーマンス・チェックが不可欠である．特に，何台程度のプロセッサを使用したときまで，計算時間の短縮が可能であるかのチェックは不可欠である．

参考文献

[1]A.Geist, A.Beguelin, J.Dongarra, W.Jiang, R.Manchek and V.Sunderam, "PVM:Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing," MIT Press, 1994.

[2]Y.Shinano,M.Higaki,R.Hirabayashi, "A Generalized Utility for Parallel Branch and Bound," Proc. Of 7th IEEE Symposium on Parallel and Distributed Processing, Texas, 1995.

[3]H.W.J.M.Trienekens, "Parallel Branch and Bound Algorithms," Doctoral Thesis, Erasmus Universiteit Rotterdam, 1990.