平成 11 年度修士論文

領域依存知識の活用に重点を置いた並列組合せ最適化ライブラリ

指導教官　近山 隆 教授

平成 12 年 1 月 28 日提出

東京大学大学院 工学系研究科 情報工学専攻

86887　横山 大作

概要

組合せ最適化問題は並列計算に向いた性質を持っているため，莫大な計算量が必要となる大規模実問題を並列計算によって高速に解きたい，という要求は強い．しかし，既存の並列化ライブラリの多くは問題領域依存の知識が十分に活用できず，実問題の専門家たちにとって不満足なものとなっている．そこで，既存のライブラリのこの問題点を解決するために，並列組合せ最適化ライブラリ・パッケージ PopKern を構築した．PopKern は，問題領域依存知識の活用に重点をおいたライブラリであり，利用者の高度な高速化技法に基づく自由な記述を許し，かつ並列処理特有の専門知識を必要とする記述を行うことなく並列計算を行うことができる．本論文では，PopKern の設計，実装，評価を述べ，このライブラリの設計を記述の容易さと実装の効率の両面から検討する．

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 研究の背景

組合せ最適化問題は，スケジューリング，設計問題等，実社会において広い応用範囲を持ち，これらの問題を解く需要が高まっている．一般に組合せ最適化問題はNP困難であり，実社会で求められる大規模な問題を解くためには膨大な計算量が必要とされる．これらの問題の多くは，独立性の高い部分問題に分割しやすいという性質を持つため，並列計算の手段を適用することで，高速化，及び適用問題の大規模化が行えると期待される．また近年，その高いコストパフォーマンスから，並列計算の技術は急速に普及しつつある．大規模並列計算機は数値計算の主流となりつつあり，2～数プロセッサ程度の小規模な並列計算機やPCクラスタなど，安価に利用できる並列計算環境も身近なものとなっている．

しかしながら，組合せ最適化問題の並列計算はまだそれほど普及しているとは言えない．この原因は，並列プログラミングの困難さにある．通信，同期，負荷分散等，並列プログラミングでは様々な点についての専門知識が必要とされる．通信タイミングに起因するバグは，同一条件下での計算実行でも再現性がなく，デバッグ作業を非常に困難にする．

これらの問題を避けるべく設計された並列言語を用いれば，並列プログラミングは容易なものとなるが，プログラマに学習を強いることになり，また過去のリソースを生かすという意味においても不利となるため，並列言語はあまり一般化していないのが現状である．

そこで，従来の逐次言語上で並列化ライブラリを用いて並列プログラミングを行うという手法が有力なものになると考えられる．実際，いくつかのライブラリが研究されているが，実問題における組合せ最適化の専門家たちはこれらのライブラリを使わない．

これらのライブラリは特定の問題領域を想定していたり，高い探索効率が得られなかったりするためである．

組合せ最適化問題は，適用される問題領域に特化した専門知識を活用することによって探索効率が大きく向上する．探索量のオーダーが変わるほどのチューニングが行えることもしばしばあるため，問題領域依存の知識を使わないまま並列化したプログラムの性能は，問題領域専門に書かれた逐次プログラムのそれに劣ることも多い．こういった問題領域依存のチューニング技法は，汎用ライブラリに用意されたいくつかのパラメータを調整する，といった簡単な手段では実現できないことが多い．従って，既存の汎用ライブラリとは異なり，問題領域依存知識を活用できるような並列化ライブラリが求められているのである．

1.2 研究の目的

無数にある適用問題領域の全てに対してチューニングを行った汎用並列化ライブラリを構築することは現実的ではない．しかし，組合せ最適化アルゴリズムの並列化技法はある程度確立したものが存在する．従って，この技法を用いてプログラムを自動的に並列化してくれるライブラリを，問題対象領域に依存する部分の記述自由度を高くして構築することができれば，実問題における専門家にとって非常に有用なものとなる．

ここで，問題領域依存知識を活用したチューニングに最も必要な部分は，問題，解等のデータ構造であると考えられる．これらデータ構造，及びそれを扱う関数群は問題領域依存部分である．しかし，並列アルゴリズムによる探索のフレームワークの部分，探索をコントロールする部分は問題領域に依存しない．このことから，問題領域に依存しない並列アルゴリズム探索のアルゴリズム部分をライブラリとして提供し，そこに利用者が定義する問題領域依存部分をプラグインして，全体として並列組合せ探索が実現できるのではないか，というライブラリ設計が導かれる．そして，このような設計方針をとるならば，問題対象領域に依存した知識を活用した逐次プログラムの性能を落とすことはなく，専門家が実問題を解くために適用できると考えられるのである．

本論文では，このような設計方針のもとに構成された汎用並列組合せ最適化ライブラリ・パッケージであるPopKern (Parallel OPtimizer KERNel) について論じる．PopKernの設計，実装，評価を通して，問題領域依存知識を活用できるような並列組合せ最適化ライブラリの可能性を追求することが，本研究の目標である．

そして，本研究の目的は，問題領域依存知識を持つ実問題の専門家が，性能面での不満を抱くことなく並列プログラミング可能な並列化ライブラリを構築することで，組

合せ最適化問題の並列計算を一般化することにある．

1.3 本論文の構成

本論文の構成は，以下のようである．まず，第2章で，本研究の対象となる組合せ最適化問題の概要，その並列処理への取り組みと並列化ライブラリに関する既存研究，及びその問題点をまとめる．次に第3章では，構築したライブラリPopKernについて，その設計方針，構成，問題領域依存知識を活用することを目的としたライブラリ設計の詳細について述べ，基本設計に関する考察を行う．第4章では，PopKernの具体的な実装方式を示し，第5章で実装に関する性能評価を行う．最後に第6章でまとめを行い，これからの展望を示す．

第 2 章

組合せ最適化の並列処理

2.1 組合せ最適化問題

組合せ最適化問題とは，数理計画問題の 1 つの分野であり，「与えられた制約条件のもとで，最大の評価値が得られるような組合せを求める」という問題である．有名な問題の例としては，巡回セールスマン問題やナップサック問題が挙げられる．

巡回セールスマン問題 (Traveling Salesman Problem, TSP) とは，

> 複数の都市が配置された世界があり，セールスマンはその都市全てを 1 度づつ訪問しなければならない．全ての都市を回って再び最初の都市へと帰る経路を考えた際，その経路の長さがもっとも短くなるような都市訪問順を求めよ．

という問題である．都市全てを 1 度づつ訪問するということが都市の訪問順の組合せに対する制約条件であり，経路の長さが評価値になる．この場合，経路の長さが短いほど評価値が高い．

また，ナップサック問題とは，

> 決められた容積しかないナップサックに，様々な種類の品物を詰めるとき，どのような組合せにしたら詰め込んだ品物の値段が最大になるか，その組合せを求めよ

という問題である．ナップサックの容積が制約条件，評価値はナップサックに詰まった品物の値段の合計である．

これらの例のように，ある問題に対する解が何かの組合せで表され，制約を満たす様々な組合せの中で最も評価の高いものを探す，というのが組合せ最適化問題である．

なお，巡回セールスマン問題については，概要を付録 Aにまとめておく．本論文では，多くの個所で巡回セールスマン問題を用いて説明を行っている．

組合せ最適化問題は実社会で広い応用範囲を持つ．例えば，電気回路を設計する際，部品の配置できる場所がグリッド状に決まっており，部品のある端子とある端子は接続されている，という制約条件のもと，回路全体の配線長が最も短くなるような部品の配置はどのようなものであるか，という問題は組合せ最適化問題である．物流分野の問題で，複数の工場から複数の店へ品物を運ぶとき，どのように品物を振り分けたら輸送費が最小になるか，あるいは人員配置の問題で，どのように作業を割り振ったら全員の不満がなくなるか，等々，あらゆる領域において組合せ最適化問題は現れる．

組合せ最適化問題は，問題規模が大きくなると指数関数的に探索空間が広がってしまう．この様子は「組合せ的爆発」とも呼ばれ，規模の大きい組合せ最適化問題を解く時の大きな障害となっている．また，組合せ最適化問題に分類される多くの問題が，NP困難な問題であることが知られており，規模の大きな問題を効率よく解く一般的な手段はない．

このように，組合せ最適化問題を解くためには莫大な計算量が必要となる．しかし，実社会の問題において組合せ最適化問題を解きたい，という要求は強い．このため，組合せ最適化問題の実問題領域への応用は古くから研究され，その問題領域に特化した知識を用いた探索高速化の技法が確立されてきた．

組合せ最適化問題の最適解を求めるためには，基本的には可能な解を全て調べ，その中の最良解を発見するしかない．しかし，問題領域には固有の性質があり，うまくその知識を生かせば，全ての解を探索することなく最適解になりうる組合せのみを探索することが可能になり，高速な探索が実現される．

また，最適解を求めるためには莫大な時間がかかるため，準最適解，つまり最適解にできるだけ近い解を短時間で求める，という手法も盛んに研究されてきた．この手法でもやはり，問題領域依存の知識を用いた高速化技法が広く研究され，適用されている．

こうして現在，組合せ最適化問題は問題領域に依存する高速化技法とともに研究され，実問題領域で応用されている．しかし，扱うべき問題の規模は拡大の一途をたどっており，組合せ最適化問題を高速に効率良く解きたいという要求は常に存在している．

2.2 組合せ最適化の並列処理

組合せ最適化問題には，一般に，

独立した部分問題に分割しやすい

という性質，つまり，問題が簡単にいくつかの部分問題に分割でき，それら部分問題間の計算に依存性が少なく，計算途中に起きる通信が少なくてすむ，という性質がある．

例えば，組合せの際にある要素を「選択したか選択しなかったか」で，問題を2つの部分問題に自然に分割することができる．ナップサックの例で言うと，ある品物をナップサックに「入れた」場合と，「入れなかった」場合で，考慮すべき品物の種類が1個減った2つの部分問題に自然に分割される．しかも，この部分問題は完全に独立した世界であり，それぞれを解く際の計算に依存性はない．

この性質は，並列計算を適用することの有効性を示している．先の例で言うなら，2つの部分問題を独立した計算機で並行して解けば，部分問題を解く時間で問題全体を解くことができる．従って，組合せ最適化問題の並列解法による高速化の研究は，自然な流れとして生まれてきた．

ここで忘れてはならないのは，組合せ最適化問題は一般にNP困難な問題に属する，という点である．並列処理による高速化は，このNP困難性を解決するものではない．従って，並列処理というアプローチは，大規模な問題を解く時の本質的な解決を与えているわけではない．

しかし，並列処理を行うことで，実用的な時間内で解ける問題規模が拡大することもまた事実である．実際，多くの有名な組合せ最適化問題について，既知の最大規模の解は並列計算によって求められている．

これは，近年の並列環境の一般化という流れからの影響でもある．スーパーコンピュータを構築する際，すでに1つのプロセッサでの性能は技術限界に達しており，複数のプロセッサによる並列化，さらには数万台規模の超並列化を行って処理能力を増やすというアプローチを取らざるを得なくなっている．また，一般消費者向けのプロセッサのコストパフォーマンスが格段に向上しており，一般的なパソコンクラスのコンピュータをネットワークで接続したPCクラスタが，そのコストパフォーマンスの良さから普及し始めている．このように，高速な計算機を低いコストで得るためには，並列計算機を用いたほうが良い，という状況が生まれてきている．従って，莫大な計算量が必要とされる組合せ最適化問題を並列計算機上で解きたい，という欲求が強まっているのである．

ところで，並列化には2つの利点がある．すなわち，

- 計算の高速化
- 計算の大規模化

の2点である．

これまで述べてきた組合せ最適化の並列処理への期待，というのは高速な計算機を使うことでより早く解を求めたい，という要求を満たすことを目的としており，前者の利点に関するものである．ここで，後者の利点に関する並列化要求，というものも存在する．

並列化により計算の大規模化が可能になる，というのは，主にハードウェア的な制限によるものである．1 つのプロセッサで使えるメモリやディスク等のハードウェア資源の量には限界がある．メモリを大容量にしていくと，当然物理的なサイズが大きくなり，配線遅延などの物理的要因で高速化が妨げられる．このように，高速性と大容量性は相反する要求項目であり，メモリ等の容量には限界がある．この制限を越えるような，非常に大きなデータを扱うためには，専用のハードウェアを構築するしかない．しかし，複数のプロセッサを使ってうまく並列処理を行えば，この制限を越えることができる．1 つのプロセッサが持つメモリ量が 1GB なら，2 つのプロセッサで並列処理を行えば見掛け上 2GB のメモリを使えることになる．

このように，使える計算機資源を増大させ，可能な計算の規模を大きくする，という目的の並列処理もある．この目的を追求する動きは，計算機資源が潤沢にある大型計算機ではなく，計算機資源が比較的制限されたパソコンクラスのコンピュータによる計算を行いたい，という利用者に多い．

組合せ最適化問題では，多くのメモリが必要になるアルゴリズムがあり，その場合使えるメモリ量が制限になって解ける問題規模が限られることも多い．そこで，並列処理を行いたいという要求が現れるのである．

このような状況から，組合せ最適化問題の並列処理が研究され，成果をあげてきている．並列処理によって良い成果が得られた問題の例として，チェス・プレイヤである Deep Blue [8, 9] の概要を示す．

チェスのようなゲームは，盤面の状態が指し手によって変化していく，状態空間内の移動がゲームの進行であると考えられる．プレイヤは，この指し手によって移行可能な状態空間内の盤面状態を探索し，最も望ましい，つまり最も自分が優勢になる盤面へ移行することを目的とする．これは，組合せ最適化問題である．チェスのように，2 人が交互に指し手を選択するゲームの場合，それぞれのプレイヤが反対の目的を持つ評価関数を用いて探索を行い，指し手を選択する．このような探索の探索木を，ミニマックス木と呼ぶ．コンピュータによるゲームプレイヤを作るときには，このミニマックス木による探索問題の解を求めていることになる．

普通，ミニマックス木の探索を深くしていくと，より先の指し手まで読んで次の一手を指すことになり，ゲームプレイヤの強さが強くなる．しかし，組合せ最適化問題で

あるので，探索を深くすると可能な盤面状態が組み合わせ的爆発を起こすため，実用的な時間内での探索には限界がある．従って，より強いゲームプレイヤを作りたいという要求から，ミニマックス木の探索をより高速に行うことが常に求められている．

1997年，Deep Blueは，$\alpha\beta$法による探索木の枝刈り，チェスというゲームに特化したヒューリスティクスに基づく探索の効率化とともに，256個の専用プロセッサを使った並列探索を用いることで，チェスの世界チャンピオンであったカスパロフに勝つ，という優れた成果をあげた．

Deep Blueが用いた専用プロセッサは，チェスの盤面の評価，可能な指し手の生成，$\alpha\beta$探索のコントロールを実現しており，チップ自身が$\alpha\beta$探索を行える．256個のチップは汎用大型計算機に接続され，コントロールされる．並列化手法は，マスタ・スレーブ型である．浅いところの探索を，大型計算機上でソフトウェアを用いて実行し，ある深さ以上の探索は専用プロセッサを用いてハードウェアで行わせる．この時，ある深さまで展開した探索木のノードを並列化した専用プロセッサに次々に与えることで，並列化が行われている．つまり，マスタである汎用大型計算機がタスクを作り，スレーブである複数の専用プロセッサがそれを独立に並列実行する，という方式である．

このように，Deep Blueは専用ハードウェアを用いた大規模並列計算を行い，チェスの$\alpha\beta$探索に関して世界最速の処理能力を得た．他の問題領域においても，世界最速の処理能力を得ようとする場合には，大規模並列計算を行うことがごく普通の手法となっている．こうして，組合せ最適化問題の高速解法の研究は，自然に組合せ最適化問題の並列化への研究へとつながっていく．そして，並列化アルゴリズムやその実装等の並列化手法に関しては，すでに一定の成果が現れ始めているのである．

2.3 並列化ライブラリ

Deep Blueは，チェス専用ハードウェアを用いて組合せ最適化問題を並列化した．その結果，チェスという限られた問題領域に関する探索の限界を大きく広げた．また，$\alpha\beta$法による探索について，専用ハードウェアを作成すればこの程度の大きさの探索ができる，という一定の実現可能目標を示したとも言える．

これはすばらしい成果であるが，専用化の結果，利益を受けられる範囲は非常に狭い．そこで，このように専用化したものばかりではなく，もう少し一般性を保って並列化を行い，広い問題範囲で適用できるような研究も多くなされている．つまり，Deep Blueのようにチェスに限定せず，同じ$\alpha\beta$法を用いているオセロ，将棋等にも適用できるように$\alpha\beta$法を並列化する，という方向性を持った研究である．

このような研究の成果として，組合せ最適化アルゴリズムの並列化ライブラリ，というものがいくつか作られている．これは，ある組合せ最適化アルゴリズムについて，利用者が一部分をプログラムし，ライブラリと組み合わせると，ライブラリが自動的に並列化されたアルゴリズムで探索を行ってくれる，というものである．

並列化ライブラリの多くには，次のような利点がある．

- 利用者は，ごくわずかなコーディングを行うだけで，組合せ最適化アルゴリズムによる探索を行える．
- 利用者は，並列プログラミングに関する知識なしで並列計算を行うことができる．

前者は，組合せ最適化ライブラリとしての側面である．組合せ最適化アルゴリズムを実現するためのプログラミングを簡単に，かつ正確に行うことを目的としたものである．

後者は，並列化ライブラリとしての側面である．並列プログラミングは通信，同期，負荷の均等な分散など，専門知識を多く必要とする．デッドロックや通信タイミングに起因する再現性のないバグなどのせいで，デバッグも難しいことが多く，並列プログラムが動くようになるまでに必要な労力が大きい．このため，並列プログラミングはそれ自身難しいものである．この困難さを隠し，並列プログラミングの知識がない人でも，逐次プログラミングした部分をライブラリと組み合わせるだけで，自動的に並列計算が行われるようにするのが目的である．

当然，並列計算を行いたいという要求には前述の 2 つの目的，すなわち高速化と規模の拡大という目的があり，並列化ライブラリはこの目的を達成するために使われる．つまり，並列組合せ最適化ライブラリの目的は，利用者が，わずかな労力で，大規模な問題を高速に解くことにある．

このような組合せ最適化問題の並列化ライブラリはいくつか存在する．ZRAM[2] は分枝限定法，バックトラックサーチなど，木構造を基本とした探索法に関する並列化ライブラリのパッケージである．PGAPack [1] ，OO-Library [3] の 2 つのライブラリは，遺伝的アルゴリズムを並列化するためのライブラリである (なお，分枝限定法の詳細は 3.5.2章で，遺伝的アルゴリズムの詳細は 3.5.4章で述べる)．

ここでは，既存研究の例として，遺伝的アルゴリズムのライブラリである PGAPack について，その概要を示す．

PGAPack でのプログラム例を図 2.1に示す．

今，利用者は 3 つの実数を引数に持つ $f(x_1, x_2, x_3)$ という実数関数を最小化したいとする．この時，引数は$-10 \leq x_1 \leq 0, 0 \leq x_2 \leq 10, -10 \leq x_3 \leq 10$ という制限である

とする．この3引数のうちx_1, x_2を10ビット，x_3を20ビットで表して遺伝的アルゴリズムにより最適化することにする．これを実現するプログラムが図2.1である．

`main`関数では，`PGACreate`を用い，40ビットのバイナリ型遺伝子を持つ，最小化問題としてライブラリを初期化している．あとはセットアップ(`PGASetup`)，実行(`PGARun`)，終了(`PGADestroy`)を行うだけである．`PGARun`では，遺伝子の評価関数として`grayfunc`なる利用者定義関数を用いることを指定している．

利用者定義関数`grayfunc`内では，最初の`PGAGetRealFromGrayCode`で遺伝子表現の0ビットめから9ビットめまでを-10から0までの実数に変換し，`x1`としている．以下同様に`x2`，`x3`を遺伝子表現から実数表現に戻し，目的の関数`f(x1, x2, x3)`の値を求めて評価値として返している．なお，`PGAGetRealFromGrayCode`はライブラリが提供する関数であり，グレイ符号で遺伝子内に表現されている実数を実数表現に戻すものである．グレイ符号とは，数字が1増えるたびに，符号化された表現上では1ビットだけ変化が起きるような符号化方式であり，近傍の値が遺伝子表現上も近傍に存在するという意味で遺伝子表現に向いている．

このように，非常に簡単な指定を行うだけで，ライブラリに計算をまかせられる．もちろん，このままでは多くの遺伝的アルゴリズム上のパラメータがデフォルト値のままであり，最善解への収束速度が早くない可能性もあるので，遺伝子の選択条件，交叉関数，次世代への生き残り戦略等を問題領域に合わせて設定する必要がある．しかし，基本的にはPGAPackを使うことで「少ない労力で遺伝的アルゴリズムを実現する」ことが可能になっている．

しかも，このプログラムにはどこにも並列を意識した記述がない．しかし，PGAPackはこのプログラムを使って遺伝的アルゴリズムの並列計算を行うことができる．PGAPackは，マスタ・スレーブ型の並列化を行っている．遺伝子集合全体を1つのマスタPEが保持し，そこから選択した遺伝子を複数のスレーブPEに分担させて評価，交叉等の演算を並列に実行する，という方式である．これにより，1PEでの実行と同じ探索結果が，おそらく早い時間で求まることが期待できる．

このように，PGAPackを使うことで「並列プログラミングの専門知識がない利用者でも並列計算が行える」ことが実現されている．

```
#include "pgapack.h"
double grayfunc(PGAContext *ctx, int p, int pop);
double f(double x1, double x2, double x3);
int main(int argc, char **argv)
{
  PGAContext *ctx;
  ctx = PGACreate(&argc, argv,
                  PGA_DATATYPE_BINARY, 40, PGA_MINIMIZE);
  PGASetUp(ctx);
  PGARun(ctx, grayfunc);
  PGADestroy(ctx);
  return;
}

double grayfunc(PGAContext *ctx, int p, int pop)
{
  double x1, x2, x3,v;
  x1 = PGAGetRealFromGrayCode(ctx, p, pop,  0,  9, -10.,  0.);
  x2 = PGAGetRealFromGrayCode(ctx, p, pop, 10, 19,   0., 10.);
  x3 = PGAGetRealFromGrayCode(ctx, p, pop, 20, 39, -10., 10.);
  v = f(x1, x2, x3);
  return (v);
}
```

図 2.1: PGAPack プログラム例

2.4 既存ライブラリの問題点

PGAPack のような既存の並列組合せ最適化ライブラリを用いることで，利用者は，わずかな労力で並列化された組合せ最適化アルゴリズムを実現することができる．しかし，実問題を以前から専門に解いている専門家たちはこのようなライブラリを使わない．これは，探索性能面に問題があるためである．

既存の並列化ライブラリの多くは，組合せ最適化ライブラリとしての側面，つまり，「組合せ最適化問題の専門知識がない初心者が，簡単にある解法のプログラムを構築できる」という面を重視している．

この側面を重視するために，既存ライブラリのほとんどは，プログラマが書くコードの量をできるだけ少なくすることを目標として作られている．つまり，組合せ最適化をできるだけ簡単に実行できるよう，利用者は少しの部分を定義するだけでよいように作られている．しかし，大規模な実問題を解く際には，このようなライブラリでは問題が起きる．

組合せ最適化問題を解く際には，適用される問題領域に特化した知識が存在し，その知識を生かすことによって探索効率が大きく向上する．設計問題等，以前から盛んに研究されてきた問題領域では，その問題に特化したチューニング技法が多く存在しており，チューニングによって探索量のオーダが変わるほどの効果がでることもしばしばである．よって，大規模複雑な実問題を解く際には，これらの問題領域依存知識が活用できないと，実用的に解ける問題の大きさがごく小さいものになってしまうのである．

例として，分枝限定法による探索を考えてみる．探索効率をあげる技法としてまず挙げられるのは問題や解のデータ表現方法の工夫である．例えば，探索途中の解の表現を，探索木上での親に相当する解との差分として表現することは，メモリの節約や評価値計算の高速化といった利点をもたらす．他にも，部分問題が再利用できるようなデータ構造を作ったり，メモリ上の配置を工夫したりと，データ表現では様々なチューニングが考えられる．また，探索途中の解候補の集合をどのように保持するか，そしてどのような順番で解候補を選んで探索を進めていくか，といった工夫も考えられる．枝刈りを行うときも，類似局面が後から現れた時に枝刈りの情報が生かせるような機構をつけたり，統計情報をとって方針決定に生かしたりといった工夫がありうる．ワーキングセット縮小のために探索途中の解集合全体を対象に一度に枝を刈るのか，逐次に枝を刈っていくのか，その選択や枝刈りのタイミングも問題領域によって異なる．さらに，入力データに応じて探索戦略を変化させることも考えられる．

このように，適用される問題領域に特化したチューニングの可能性は多岐にわたる．

しかし，既存のライブラリでは，問題や解のデータ構造の自由度が低かったり，枝刈りの過程が利用者から見えなかったり，探索途中の解候補集合を全てライブラリが管理して，その試行順序が固定されていたり，または複数の方法から選ぶだけであったりと，このようなチューニングの自由度が低いものであることが多い．

実際のライブラリを例にあげてみる．2.3章で既存研究の例にあげたPGAPackでは，段階的にプログラムのチューニングができるよう，デフォルトで使われている関数を順次利用者定義に差し換える形でより高度なチューニングに対応している．例えば，最初はデフォルトの選択ルールを用いていたが，それを利用者が定義した選択ルーチンに差し換える，さらに交叉演算を書き換える，と次第に利用者定義の部分を増やしていき，最終的には遺伝子のデータ型を定義するところまで利用者が扱える．こうして，かなりの部分を利用者が定義することができるが，まだ不十分な点がある．それは，遺伝子集合のデータ型を利用者から隠蔽している点である．

PGAPackでは，基本的に2つの遺伝子集合領域を用意し，一方の領域に存在している古い遺伝子が，一部はそのまま，一部は交叉演算や突然変異を終えて，もう一方の領域へコピーされることで世代交代を実現している．この枠組みを保持しようとするため，遺伝子そのもののデータ型は利用者定義が可能だが，配列に遺伝子が保持され，コピーを繰り返されるという方式を変えることはできない．従って，例えば遺伝子集合をリスト構造で保持する，といった問題領域依存知識の活用による高速化が完全には行えないのである．

このように，既存のライブラリでは問題領域依存のチューニングを完全には行えない．それらのライブラリがある程度特定された問題領域や非常に包括的な問題領域を想定して，その中で高速な探索方式を提供しているため，想定と異なる問題領域でのチューニングは困難になるのである．

これは，並列化ライブラリの目的として，組合せ最適化問題の知識がない人でも簡単にプログラミングできる」ことに重点をおくことに起因している．今，本当に並列処理による速度向上を望んでいるのは，実問題領域の組合せ最適化の専門家たちであり，この専門家たちのために，組合せ最適化問題の知識がある人が自由にプログラミングできる」ようなライブラリが望まれているのである．

第 3 章

並列組合せ最適化ライブラリ PopKern

3.1 PopKern の設計方針

第 2章で述べたように，既存の並列化ライブラリでは問題領域依存知識が十分に活用できないことが問題となっている．並列組合せ最適化ライブラリ PopKern (Parallel OPtimizer KERNel) は，この問題を解決すべく設計され，構築されたものである．

PopKern の目標は，

- 組合せ最適化アルゴリズムの，問題領域依存部分は全て利用者が定義できる
- 利用者は並列プログラミングについての知識無しにプログラミングできる

という 2 点である．これらの目標は，次のようなライブラリならば達成できる．

問題領域依存知識を活用したチューニングに最も必要な部分は，問題，解等のデータ構造であると考えられ，これらデータ構造，及びそれを扱う関数群は問題領域依存部分である．しかし，並列アルゴリズムによる探索のフレームワークの部分，探索をコントロールする部分は問題領域に依存しない．よって，問題領域に依存しない並列アルゴリズム探索のアルゴリズム部分をコアとして提供し，そこに利用者が定義する問題領域依存部分をプラグインして，全体として並列組合せ探索が実現できるようなライブラリを構築すれば，前述の目標は達成できるのである．

これらの目標は，一般の並列組合せ最適化ライブラリの目標とやや異なっている．2.4 章で述べたように，一般のライブラリは，組合せ最適化ライブラリとしての側面が強く，「組合せ最適化問題の専門知識がない初心者が，簡単にある解法のプログラムを構築でき，それを並列実行できる」ことを目的としている．しかし，PopKern は，組合せ最適化ライブラリとしての側面は，問題領域に依存しないフレームワークを提供するだけで

あり，その上で並列化ライブラリとしての側面を最大限に発揮しようとしている．つまり，PopKern は，「組合せ最適化問題の専門知識を持つ問題領域専門家が，ある解法のフレームワークのもとで自由にプログラミングでき」，しかも「並列を意識することなく逐次プログラミングしたものが，ライブラリによって並列計算され，高速化，問題の大規模化が可能になる」ことを目標にしているのである．

このような目標を達成するため，PopKern の設計方針を以下のように定めた．まず，問題領域に依存する最大の部分は問題，解等のデータ構造である．PopKern ではこのデータ構造に対する制限をなくし，完全に利用者定義にした．よって，問題や解を操作するサブルーチンも必然的に利用者定義となった．解の集合等のデータ構造も同様に考えた．

一方，PopKern が提供すべき，問題領域に依存しない部分については，探索アルゴリズムにしたがって探索を制御するために必要な部分，として考え，PopKern のカーネルは基本的に探索を制御する役割だけを持つこととした．例えば分枝限定法では，現時点での最良解，上界，下界等を保持し，探索途中の解の選択・分枝・枝刈りの繰り返しというアルゴリズムの流れを実現する機能だけを PopKern に持たせる，という方針で設計した．しかし，並列化ライブラリとしての側面では，通信や同期，負荷分散といった並列処理に関する部分を利用者が極力意識しないように設計を行った．PopKern が想定する利用者は，並列計算の知識が必ずしも十分ではない特定問題領域の逐次プログラミングの専門家たちだからである．

このような観点から，PopKern ライブラリが提供する機能と利用者定義部分との切り分けにあたり，利用者の問題領域依存のプログラミング自由度をできるだけ高くするように設計を行った．PopKern の基本構成，基本的な処理の流れ，そして各アルゴリズムでの具体的な利用者定義部分の切り分けについて，以下に述べることにする．

3.2 構成

PopKern の構成を図 3.1に示す．

PopKern は次の 2 つの機能部分からなる．

組合せ最適化ライブラリ PopKern は代表的な組合せ最適化アルゴリズムを提供する．これらのアルゴリズムは，問題領域独立部分と問題領域依存部分に分けることができる．問題領域独立部分は PopKern の組合せ最適化ライブラリ部分に組み込まれている．問題や解候補の型表現等，問題領域依存部分は PopKern には定義され

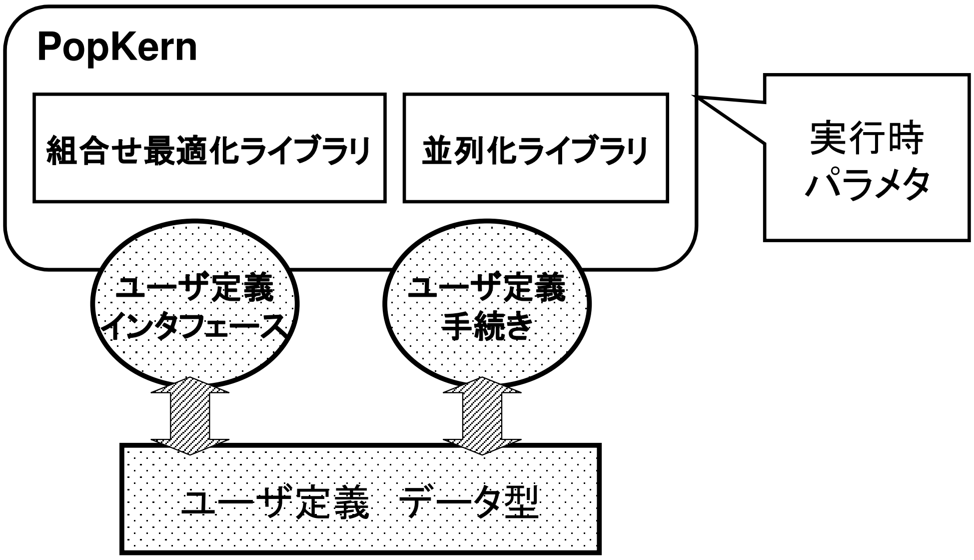

図 3.1: PopKern の構成

ていない．この問題領域依存部分を，PopKern の組合せ最適化ライブラリ部分にプラグインすることで，並列組合せ最適化アルゴリズムの動作が実現される．

PopKern のカーネル部分は，問題領域に依存する利用者定義のデータ型をブラックボックスとして扱い，利用者が定義したサブルーチンを介してデータを扱う．利用者の定義した，データ型定義及びそのデータに直接アクセスするアルゴリズム中で必要なサブルーチンを，PopKern のシステムと一緒にコンパイル，リンクして実行用バイナリを作成する．これらのサブルーチンを記述する際には，問題領域に関する専門知識を使い，これまでの通り逐次プログラミングを行えばよく，並列計算に関する専門知識は必要としない．

PopKern は現在，次の 4 種類の並列組合せ最適化アルゴリズムを提供している．

- 山登り法
- 分枝限定法
- 焼き鈍し法
- 遺伝的アルゴリズム

分枝限定法は全ての組み合わせを探索して最良の解を得るためのアルゴリズムであり，その他の 3 つはヒューリスティックに基づいてできるだけよい解を探索するためのアルゴリズムである．

並列化ライブラリ PopKern の並列化ライブラリ部分は，プロセッサ間の通信，同期，及び負荷分散を実現する．組合せ最適化ライブラリ部がこれらの機能を使って並列計算を行う．利用者がこれらの機能を直接使うことはなく，並列計算技法に関して考慮する必要はない．PopKern のカーネル部が，利用者定義データのプロセッサ間移動や同期を自動的に行う．PopKern は利用者定義データの内部構造を関知しないので，通信の際には利用者が定義したシリアライズルーチン（バイト列への変換ルーチン）を用いる．

3.3 並列化アルゴリズムに共通する処理

並列組合せ最適化アルゴリズムを実現する際，共通して必要となる基本的な手続きがある．すなわち，問題を読み込み，全ての PE に問題の情報を分配し，終了条件が満たされたら探索された最良解を保存して正常終了する，という手続きは必須なものである．また，並列アルゴリズムは普通，全 PE についての現時点の最適解のコストを把握していることを要求する．そのため，各 PE は全 PE についての現時点の最適解のコストを適宜更新しつつ保持しなければならない．

これらの，並列組合せ最適化アルゴリズムに共通して必要となる機能を実現する部分について，以下にその動作を示す．

1. 問題をファイルから読み込み，問題データを全ての PE(Processing Element) に配布する．

2. それぞれの PE で，指定されたアルゴリズムに基づき求解を始める．各 PE はその PE 内で探索された解候補のうち，最良となるものを保持している．これを局所最良解候補と呼ぶ．探索が進むにつれ，より良質の解候補が見つかり，局所最良解候補は更新されていく．

3. PopKern の実行時には，全ての PE を統括するグローバルマネージャが 1 つ作られる．グローバルマネージャは，システム全体の解候補のうち最良となるものを，大域最良解候補として保持している．局所最良解候補が更新されると，この解候補はグローバルマネージャに送られ，そこで大域最良解候補の更新が行われる．

4. 大域最良解候補が更新されると，グローバルマネージャはその解を全 PE にブロードキャストする．ローカルマネージャはその情報を受け，局所最良解候補と比較，良質な解の方を局所最良解候補として保持する．つまり，局所最良解候補は大域最良解候補のキャッシュのように振る舞う．

5. 最良解候補が更新されたとき，PopKern は利用者が定義した終了チェックルーチンを呼ぶ．最良解候補の質が十分満足のいくものであった場合，その時点で探索を打ち切ることができる．また，同時に表示ルーチンを呼ぶこともできるので，局所最良解候補や大域最良解候補の改善の様子を追うこともできる．

6. 終了指示があると，グローバルマネージャに保持された大域最良解候補を出力して探索を終了する．上に述べた終了チェックルーチン，及び実行時引数に与える実行制限時間によって終了の条件を決めることができる．

以上が PopKern の基本的な動作の流れとなる．

ローカルマネージャが保持する局所最良解候補は，グローバルマネージャが保持する大域最良解候補をキャッシュする形になっているが，その同期方式はラフなものであり，大域最良解候補との一致を常に保証するものではない．しかし，並列化アルゴリズムは通常，それほどの一貫性を要求しない．大域最良解の一貫性を保証するためのコストがかかりすぎ，並列効果に悪影響を与えるためである．従って，ここに示したようなラフなキャッシュ方式でも問題は生じない．なお，適切な実装を行えば，ある PE で最良解が更新され，それが大域最良解の変更を引き起こし，全 PE の局所最良解が変更されるまでの時間的遅れはそれほど大きなものではない．

3.4 共通する利用者定義部分

前述の基本動作を得るため，利用者は 4 つのアルゴリズム共通に以下の部分を定義する．利用者は，C 言語で逐次処理として利用者定義部分のプログラミングを行えばよい．

問題/解の型定義 PopKern は問題や解がどのようなデータ構造で表現されているか関知しない．利用者はこのデータ構造を C 言語の型として定義する．“問題” 構造には，計算途中で変化しない，組合せ最適化の対象となる問題に関する情報を格納する．“解” 構造には，計算途中で更新されていく，可能な解を表現する情報を格納する．2 次元ユークリッド距離の巡回セールスマン問題の単純な解法を例としてあげると，問題の構造には巡回すべき都市の座標を，解の構造には都市の巡回順序を示した数列を，それぞれ格納するようにすればよい (図 3.2)．

なお，実際には，分枝限定法 (Branch and Bound 法) では `popkern_bb_problem` のように，利用者が定義すべきデータ型やサブルーチン名は使用するアルゴリズムによって若干変化するが，本論文中の例示ではこの違いについては記述しない．

```
struct city_coordinates {
  double x, y;
};
typedef struct euclid_tsp_problem {
  int num_cities;
  struct city_coordinates *coords;
} popkern_problem;

typedef int *popkern_solution;
```

図 3.2: TSP のための問題/解記述データ型定義例

問題入力/解出力 計算を始める前に，PopKern は問題データを読み込まなければならない．利用者はこのためのサブルーチンを定義する必要がある．同様に，最終的な解を出力するサブルーチンも利用者定義である．読み込み/書き出しのための入出力のプリミティブは並列計算環境を意識する必要があるため，PopKern のシステムが管理している．

例として，巡回セールスマン問題を対象にして，問題記述ファイルの最初に都市数，続いて各都市について x 座標, y 座標の値が連続して記されているとすると，図 3.3のような定義をすればよい．解出力ルーチンも同様である．プログラム中の `popkern_error` は PopKern が提供するエラー表示用プリミティブであり，詳細は後述する．

問題/解のシリアライズ/アンシリアライズ 問題や解のデータ構造は，並列計算のために PE 間を必要に応じて移動する．しかし，PopKern はそれらのデータがどのように表現されているか知らない．従って利用者は，利用者定義の表現形式から PopKern システムが扱えるバイト列の形式へと変換するサブルーチンを定義しなければならない．これを，シリアライズと呼ぶ．逆に，バイト列表現から利用者定義表現へと変換するサブルーチン（アンシリアライズ）も定義する．PopKern システムは，このバイト列表現を使って，利用者定義データ構造の PE 間通信や同期機構を利用者に意識させないように実現する．

```
void
popkern_read_problem(FILE *file, popkern_problem *problem)
{
  int k;
  if (fscanf(file, "%ld", &problem->num_cities) != 1)
    popkern_error("Problem specification file format error");
  problem->coords =
    calloc(problem->num_cities, sizeof(struct city_coordinates));
  if (problem->coords == NULL)
    popkern_error("Couldn’t allocate memory "
                  "in reading problem spec");
  for (k = 0; k < problem->num_cities; k++) {
    if (fscanf(file, "%lf%lf",
               &problem->coords[k].x, &problem->coords[k].y) != 2)
      popkern_error("Problem specification file format error");
  }
}
```

図 3.3: TSP のための問題入力ルーチン定義例

前に例示した巡回セールスマン問題による問題データ型のシリアライズには，図3.4のようにコーディングすれば良い．popkern_send_long，popkern_send_doubleはシリアライズのためのプリミティブである．都市数，各都市の x, y 座標の順にシステム提供のプリミテイブを使ってシリアライズが行われている．アンシリアライズの際は，シリアライズした順にプリミティブを使うことで，バイト列表現からデータを取り出せる．図 3.5にアンシリアライズの例を示す．

利用者定義データ構造のコピー/解放ルーチン PopKern は利用者定義データがどのようなデータ構造をとっているか関知しない．従って，探索途中の最良解を保存したり，より良質の解が見つかったため以前の解を破棄したりするときのため，利用者定義データ構造のコピーやメモリ解放のためのルーチンを利用者が定義する必要がある．

```
void
popkern_encode_problem(popkern_problem *problem)
{
  int k;
  popkern_send_long(problem->num_cities);
  for (k = 0; k<problem->num_cities; k++){
    popkern_send_double(problem->coords[k].x);
    popkern_send_double(problem->coords[k].y);
  }
}
```

図 3.4: TSP のための問題シリアライズルーチン定義例

`popkern_problem` や`popkern_solution` のコピーの際，PopKern はメモリイメージをそのままコピーする．従って，これらのデータ型の中に全てのデータが格納されていれば，コピー/解放ルーチンは特に定義する必要がなく，用意されている何もしないダミールーチンを使えば良い．内部でポインタを用いて外部のデータ構造を参照している場合，例えばツリー構造をとっていたり，例示しているプログラムのように外部にデータをアロケートしている場合は，正しくコピー/解放が行えるルーチンを定義する必要がある．例示してきた巡回セールスマン問題の解データのコピールーチン，及び解データの解放ルーチンの例を，図 3.6，図 3.7に示す．

```
void
popkern_decode_problem(popkern_problem *problem)
{
  int k;
  problem->num_cities = popkern_receive_long();
  problem->coords =
    (struct city_coordinates *)
      calloc(problem->num_cities, sizeof(struct city_coordinates));
  if (problem->coords == NULL)
    popkern_error("Couldn’t allocate memory "
                  "in reading problem spec");
  for (k = 0; k < problem->num_cities; k++){
    problem->coords[k].x = popkern_receive_double();
    problem->coords[k].y = popkern_receive_double();
  }
}
```

図 3.5: TSP のための問題アンシリアライズルーチン定義例

```
void
popkern_finish_copy(popkern_solution *solution,
                    const popkern_problem *problem)
{
  int *copied =
    (int *)calloc(problem->num_cities, sizeof(int));
  if (copied == NULL)
    popkern_error("Couldn’t allocate memory "
                  "in popkern_finish_copy()");
  memcpy(copied, *solution, problem->num_cities*sizeof(int));
  *solution = copied;
}
```

図 3.6: TSP のための解データコピールーチン定義例

```
void
popkern_free_solution(popkern_solution *solution,
                      const popkern_problem *problem)
{
  free(*solution);
}
```

図 3.7: TSP のための解データ解放ルーチン定義例

解候補のコスト関数 解候補の質を`double`型で示す評価関数を作成する．PopKernで扱う問題は全て「最小化問題」とし，評価値が小さいほど望ましい解であるとする．

この他，解候補の可視化ルーチン，終了チェックルーチン，等を定義する．解候補の可視化ルーチンは，探索状況を把握するためのものであり，必要がなければデフォルトのダミールーチンで代用できる．

ところで，これらの利用者定義部分は，シリアライズ/アンシリアライズルーチンを除いて，組合せ最適化アルゴリズムを逐次プログラミングする際に必須となる部分ばかりである．従って，利用者がPopKernを利用するために必要となる余分な手間は，シリアライズに関する部分のみである．

さて，これらの利用者定義部分を記述するために，PopKernは以下のようなプリミティブを提供する．

シリアライズ/アンシリアライズプリミティブ これらのプリミティブは，利用者がプログラム中で使用している1つの数値と，PopKernのカーネルが用意したメモリ上のバイト列との相互変換を行うものである．

図3.4の`popkern_send_long`がシリアライズ用プリミティブである．これにより，1個の`long`型整数をバイト列に変更し，図3.5の`popkern_receive_long`で再び`long`型整数として受け取っている．PopKernが通信用のバッファとして用意したメモリは利用者はまったく意識しない．利用者が意識しなければならないのは，シリアライズプリミティブを呼んだ「順番」である．バイト表現はシリアライズプリミティブが呼ばれた順番に従って生成されるので，アンシリアライズルーチンで同じ順番でプリミティブを呼ばないと正しくアンシリアライズできない．

また，`double`型浮動小数点数用のプリミティブも用意されている．

エラー処理プリミティブ 実行制御用のプリミティブであり，図3.3にある`popkern_error`が相当する．`popkern_error`は，引数に与えた文字列をコンソールに表示し，ライブラリの実行を中断する．

メッセージ表示プリミティブ このプリミティブは並列実行環境下でも使えるコンソールへのメッセージ出力を提供する．デバッグ，簡単な動作確認等に自由に使える．

これらのプリミティブを用いて，利用者は並列環境を意識することなく，シリアライズ/アンシリアライズ操作やコンソールの利用を行うことができる．

3.5 最適化アルゴリズムの並列化

組合せ最適化には代表的なアルゴリズムがいくつか存在する．そして，それぞれのアルゴリズムについて，並列化への研究が行われている．

現在のPopKernは，代表的な4種類の組合せ最適化アルゴリズムについて，比較的効率が良くしかもライブラリの枠組みへの親和性が良い並列化手法を提供している．

この節では，PopKernが提供している組合せ最適化アルゴリズムについて，その概要，並列化のための手法，及び利用者が定義する部分を順に説明する．また，まだ実装されていないが提供しうるアルゴリズムについても述べる．

なお，これらの組合せ最適化アルゴリズムについては，[18, 13]に詳しい．

3.5.1 山登り法

山登り法の概要

山登り法とは，組合せ最適化問題で確率的に最良解を求める手法のうち，最も簡単な方法である．

性質の良い問題の場合，最良解を少し改変した解，つまり最良解の近傍の解はやはりそれなりに質の良い解が存在すると期待される．すなわち，解空間の評価値は最良解を頂点とした「山」のように広がっていると考えられる．従って，最良解の近くの「山肌」に相当する地点から，斜面の傾きを調べてより高い方へ，つまり山の頂点へ向かって探索を進めていけば，最良解に到達することができると考えられる．これが山登り法の考え方である．

疑似コードによるアルゴリズムの概要を図3.8に示す．今，ある暫定解が与えられたとき，その解をある手順で変更した複数の解を生成する．これは，解が存在する空間中，元の暫定解の近傍に位置するものである．この近傍解のうち，最も評価値が改善されたものを選びだして次の暫定解とし，再び近傍解を作成，探索を繰り返す．近傍を探索しても暫定解が改善されなくなったときは，その暫定解が山の頂点であるので，これが準最適解である．もしこの暫定解が不十分なものである場合は，初期暫定解をまったく別のところに設定し直し，再び山登り法を繰り返す．

山登り法には，局所最適解に陥る欠点がある．一般に，解空間の評価関数分布はなだらかな1つの山になっているわけではなく，あちこちに局所的な山や谷がある，でこぼことしたものになっていることが多い．山登り法では，常に山頂へ向かって探索を行うので，局所的な山頂を見つけたら，それ以上良質の解を発見することはできず，再び

```
山登り法 アルゴリズム
出力: 準最適解
begin
  1. repeat
       (a) 探索開始点となる初期暫定解を生成する;
       (b) repeat
             暫定解の近傍を探索，評価値が高くなる方へ暫定解を変更;
           until 暫定解が改善されなくなった;
     until 終了条件;
end.
```

図 3.8: 山登り法の概要

初期解を設定し直して別の山頂を探すしかない (図 3.9参照)．

このように山登り法は探索空間が狭く，単独で用いてもそれほど効率よく解が見つけられないことが多い．しかし，暫定解の近傍を高速に探索し，局所的な最適解をすばやく見つけることができる．よって，ヒューリスティクスに基づいて暫定解を改善しながら探索を進めるアルゴリズムと組み合わせると，暫定解をより早く局所最適解に収束させることができるため，効率が大きく向上する．PopKern では，このような組み合わせによる使用を考え，並列山登り法を提供している．

並列化手法

山登り法の並列化は，多数の地点から山登りを独立して行う，という手法が取られることが多い．PopKern もこの方式をとり，各 PE に異なる初期解を設定し，山登りを各 PE で独立して行う．山登りで局所最適解に収束した場合は，改めて初期解を設定し直し，山登りを繰り返す．

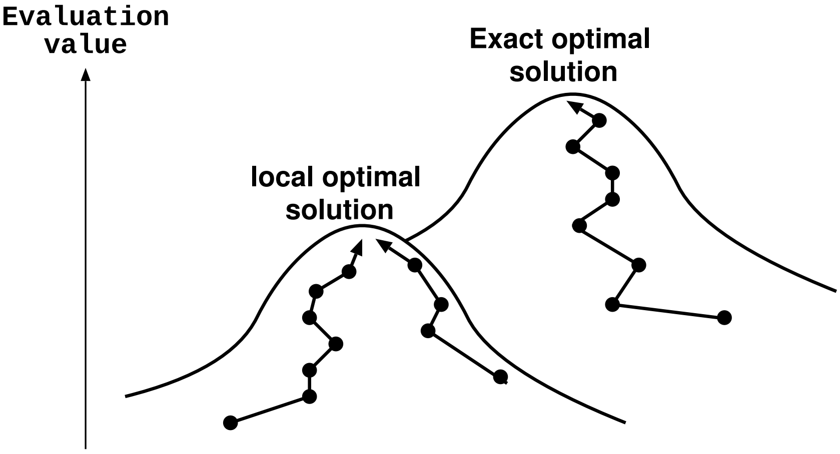

図 3.9: 山登り法による探索

利用者定義部分

利用者は，ランダムに初期解を生成するサブルーチン，暫定解の近傍を探索し，改善された解を返すサブルーチンを定義する．

3.5.2 分枝限定法

分枝限定法の概要

組合せ最適化問題の最適解を求めるためには，受理可能な全ての解を列挙し，探索しなければならない．このような列挙は普通莫大な計算コストが必要になるが，最適解である可能性がない解の探索をを順次取りやめる，すなわち探索木の枝刈りを行えば効率が向上する．これが分枝限定法の考え方である．

疑似コードによるアルゴリズムの概要を図 3.10に示す．基本的に，探索木における探索途中のノードの集合について，ノード集合中から 1 つのノードを選択し，下界チェックを行い，可能ならば分枝/枝刈りを行う，という操作の繰り返しで実現されている．

```
分枝限定法 アルゴリズム
出力: 最適解
begin
  1. 初期暫定解を計算する;
  2. 探索のルートとなるノードを作成する;
     $\mathcal{L} = \{($ルートノード$)\}$;
  3. while ($\mathcal{L} \neq \emptyset$) do
     begin
       (a) ノード $P \in \mathcal{L}$ を選び，$\mathcal{L} = \mathcal{L} \setminus \{P\}$ とする;
       (b) $P$中の実行可能解の計算，ならびに下界値評価を行う;
       (c) if $P$の中の実行可能解が見つかっていた then
             実行可能解を暫定解と比較，コストの低い方を新暫定解に;
       (d) if (下界値 $<$ 暫定解コスト) then $P$を展開し，$\mathcal{L}$ に追加 (分枝);
       (e) $\mathcal{L}$ 中の不要ノード，及び $P$ を削除 (枝刈り);
     end;
end.
```

図 3.10: 分枝限定法の概要

並列化手法

このアルゴリズムを並列に実行するためには，ノード集合を適当に PE 間で分担して探索を行うことが基本になる．しかし，ノード集合は探索中にその中身が大きく変化し，探索開始点で静的に分割しても，計算量を均等にすることは難しい．例えば，探索木のルートにあたるノードが分枝した時点で，各枝を各 PE に均等に分割しても，探索途中に枝刈りが行われるため，探索に必要な計算量は均等にならない．

これを避けるためには，探索の途中で適切にノードを移動し，それぞれの PE の負荷を均等化する，動的負荷分散が必要になる．PopKern も，動的負荷分散機構を提供している．PopKern は，各 PE が現時点で探索すべきノード集合を持っているか否かを把握するグローバルマネージャを作る．ある PE が，自分の持っていたノード集合を全て探索し尽くしたとき，この PE はグローバルマネージャにノードを要求する．グローバルマネージャは，まだノードがある PE に対し，持っているノード集合を分割し，ノードを要求していた PE に送るよう命令する．このようなアルゴリズムで動的負荷分散が実現される．

問題領域依存知識を重視するための戦略

分枝限定法では，探索途中のノード集合から 1 つのノードを選択する際の戦略によって，深さ優先探索や最良優先探索といった探索方法の違いが発生する．これら代表的な探索方法をライブラリとして提供することも可能であるが，深さ優先探索と最良優先探索を組み合わせ，場合によって使い分ける，といった戦略も考えられるため，利用者の問題領域依存知識を重視するという観点からは，探索方法までライブラリ側で規定してしまうことは問題である．そこで，PopKern ではこのノード集合の中身を利用者定義とし，その振る舞いを関知しないものとした．

また，ノード集合中の不要ノードを削除する際も，必ず不要ノードを削除するのか，暫定解がある程度更新されてからまとめて削除するのか，といった問題領域依存知識による戦略の違いが考えられるため，PopKern では枝刈りのタイミングも規定しないことにした．具体的な利用者定義部分の規定は後述する．

さて，並列化手法の項で述べた通り，分枝限定法の並列化のためには探索途中のノード集合を適切に分割し，各 PE に割り振る必要がある．この時，探索のスケジューリングが定まっていれば，PE 間を移動するノードを適切に選択することで並列化の効率が向上できる．例えば，深さ優先探索の場合は，探索の局所性を生かすためルートノードに近いノードを優先的に移動させ，最良優先探索では現在の探索とできるだけ独立した，しかし質の良い解が期待できるノードを優先的に移動させる，といった戦略が考えられ

る．このように，深さ優先探索の場合と最良優先探索の場合とではノード集合の効率的な分担方法が異なってくるし，一般的なものとは違う，問題領域依存の分担方法というものも考えられる．

しかし，PopKern は分枝限定法でのノード集合の構造や振る舞いを規定せず，利用者の自由度を最大限生かせることを設計方針としている．従って，これら探索戦略に依存したノード集合分割は提供することができない．そこで，PopKern ではこれらの移動するノードの性質を規定せず，利用者が適当に分割したノード集合を PE 間で移動させるものとした．

このような方針だと，利用者が不適切なノード集合分割を行った場合，並列化の効率が大きく低下する恐れがある．しかし，どのようなノード集合分割を行ったとしても，少なくとも分枝限定法は正しく実行される．正しく並列動作が行える段階まで達していれば，並列プログラミングの知識がない利用者でも，実験などの手段で気軽にノード分割の手法を適正化する気になるであろう．

このような理由から，PopKern では探索途中のノード集合の中身を利用者定義とし，並列動作のために必要なノード集合の分割，及び結合も利用者定義とすることとした．

利用者定義部分

前述したように，PopKern はノード集合を利用者定義のデータ型とし，ノード集合に対して操作を加えている．

従って，利用者は展開途中のノード集合のデータ構造を定義する．さらに，このノード集合に対して，分枝限定法に必要な操作を行うルーチン群を定義する．定義すべきサブルーチンは，基本的に図 3.10中に示した手続きと対応している．すなわち，初期暫定解の計算，ルートノードからなる初期ノード集合の作成，などを利用者定義する．

さて，前述した通り，枝刈りのタイミングやその方法などは問題領域依存知識が活用できる部分なので，この知識を生かせるように利用者定義ルーチン部分を切り分ける必要がある．図 3.10中の 3のループにおいて，まず，(3a) と (3b) の手続きを 1 つの利用者定義関数とする．(3c) と (3d) の制御部分をライブラリが受け持ち，暫定解の更新，保持等をライブラリ側が行う．その後に実行される (3d) と (3e) の部分，すなわち分枝と枝刈を行う部分をやはり一体として利用者定義関数とする．

分枝限定法を実現するための利用者定義部分は，以上のような枠組みで定義されている．

また，並列動作のために，ノード集合を適当に分割するルーチン，ノード集合をシリアライズ/アンシリアライズするルーチン，並びに移動してきたノード集合を PE に

```
焼き鈍し法 アルゴリズム
出力: 準最適解
begin
    1. 乱数に従って初期暫定解 $i$ を計算;
    2. 温度 $T$ を初期化;
    3. while 終了条件が満たされない do;
    begin
        (a) 暫定解 $i$ をランダムに変更した解 $j$ を生成，
            暫定解 $i$ との評価値の変化 $\Delta E$ を求める;
        (b) if $\Delta E < 0$ (改善) then $i$ := $j$;
            else if $P\left(\exp\left(-\frac{\Delta E}{T}\right)\right)$ then $i$ := $j$;
        (c) $T$ をスケジュールに合わせて下げる
    end;
end.
```

図 3.11: 焼き鈍し法の概要

元々存在していたノード集合と結合させるルーチンを定義する必要がある．元々の分枝限定法の逐次プログラムに存在しなかった部分は，これらの並列動作用のルーチンのみである．ここで，これら並列動作用ルーチンの記述のためには，並列処理に関する知識はまったく必要なく，逐次処理として記述を行うだけで良いことが重要な点である．つまり，利用者は並列プログラミングを意識することなく，本来の探索プログラムにごく少量の逐次ルーチンを追加記述するだけで，並列計算が実行できることになる．

3.5.3 焼き鈍し法

焼き鈍し法の概要

焼き鈍し法とは，確率的に解候補を改善していく手法の 1 つである．「焼き鈍す」という名前は，高温から徐々に温度を下げることによってエネルギー順位の低いきれいな

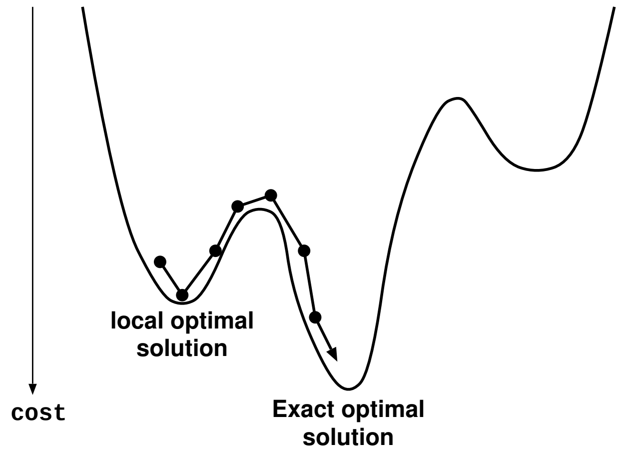

図 3.12: 焼き鈍し法による局所最適解からの脱出

結晶を得る方法をシミュレートすることに由来している．概要を図 3.11に示す．基本的な動作は焼き鈍し法と似ており，暫定解の近傍を調べ，より評価値の良い側へ暫定解を変化させる操作を繰り返すことで良質の解を求める．この時，解の評価値が悪くなる方向への移行を確率的に認めている点が山登り法と異なる点である．

コスト Eの解 x を，近傍のコスト E'の解 x'へ移行しようとしたとき，その移行が受理されるかどうかは，

$$P(E,E',T)=\begin{cases}1 & (E'\leq E)\\ e^{-(E'-E)/T} & (E'>E)\end{cases}$$

なる確率 $P(E,E',T)$ によって決定される．つまり，コストが減少する，改善方向への推移は無条件に認められるが，コストが増大し解が改悪される方向への推移は，温度と呼ばれる Tが大きいほど受理されやすくなる．このように，改悪方向への移行が確率的に認められることから，図 3.12のように局所最適解からの脱出が可能になるのである．

焼き鈍し法では，この温度 Tを最初は高くしておき，次第に下げていくことで，局所最適解に捕らわれることを避けて大域最適解を見つけようとする．この時，現時点の温度 T_k，次のステップの温度 T_{k+1}として，

$$T_{k+1}=\frac{T_k}{\log k}$$

という式より遅い速度で温度を下げていくとき，得られる解が最適解となることが証明されている．しかし，この温度スケジューリングでは解の収束が極めて遅いので，実用

上は

$$T_{k+1} = \alpha T_k$$

というスケジューリングが用いられることが多い (αは 0.8 ~ 1 程度) .

局所最適解が求まり，それが不十分であったときには，もう一度温度を上げて解の改善をやり直す．温度 Tをどのように設定するか，温度の下げ方をどのように行うか，そして再度温度を上げるときにどの程度まで上げるか，ということは一般にうまく求める方法がなく，経験や実験に基づいて決定されている．

並列化手法

焼き鈍し法を並列化する際，単純に各 PE が独立して逐次の焼き鈍し法を繰り返したのでは，良い性能は得られない．並列焼き鈍し法は，同時に複数の解が解空間を移動しながら最適解を求める方式であると言うことができる．この時，複数の解の中でも良い解は温度が低い状態でより厳密な解を探索し，悪い解は温度を上げて局所解から脱出させ，より見込みのある解へと早く移動させる，ということをしないと，各 PE が独立して逐次焼き鈍し法を行う場合と変わらない．

各 PE が逐次焼き鈍し法のアルゴリズムで探索を行いながら，このような要求を満たすためには，一定間隔ごとに全 PE の解の評価値や温度情報を集め，その情報をもとに全 PE の温度設定や暫定解を変更するという重い処理が必要になったりする．

また，逐次焼き鈍し法のアルゴリズムで，温度を徐々に下げる時のスケジュールは探索結果の解の質や探索速度に関わる重要なものであるが，これを適切なものに設定することは一般に難しい．

これらの問題点を回避するアルゴリズムとして，温度並列焼き鈍し法 [10] が提案されている．温度並列焼き鈍し法では，複数の PE はそれぞれ異なった温度を担当し，常に一定温度で独立して解の改善を続けている．そして，一定間隔ごとに隣り合った温度の解を比較し，確率的に解の入れ替えを行う (図 3.13 参照) .

いま，隣り合う温度 T ，T'において，解のコストが E ，E'であったとする．この時，解の入れ替えが起きる確率 $P(T, E, T', E')$ は

$$P(T, E, T', E') = \begin{cases} 1 & if \quad \Delta T \cdot \Delta E < 0 \\ \exp\left(-\frac{\Delta T \cdot \Delta E}{TT'}\right) & otherwise \end{cases}$$

$$where \quad \Delta T = T - T',\ \Delta E = E - E'.$$

と表される．つまり，温度が高い方の解の評価値の方が良ければ無条件に解を交換し，そうでないときは確率的に解交換が行われる．解は異なる温度間を移動しながら改善さ

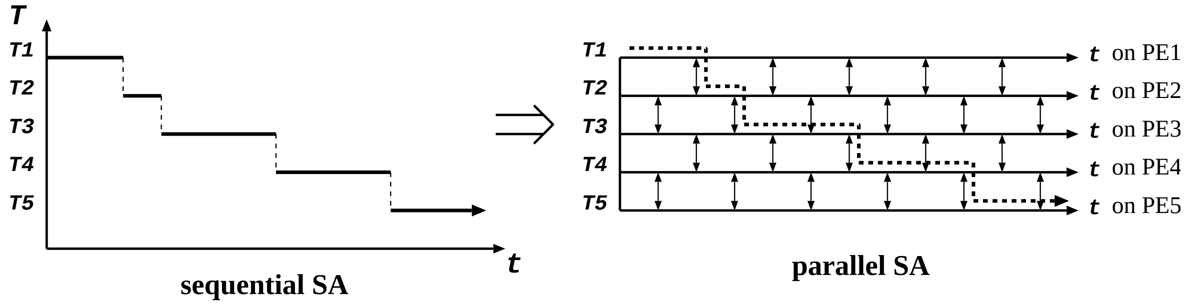

図 3.13: 温度並列焼き鈍し法

れ，並行して探索されている全暫定解のうち，良質の解が次第に低い温度帯へと移動していくと期待できる．つまり，プロセッサの温度分布は，逐次焼き鈍し法での時間軸方向の温度変化に相当する．解の質によって温度を調整することは必要なく，温度管理が自動化される点がこの温度並列焼き鈍し法の特長となる．

また，温度並列焼き鈍し法は時間的に一様なアルゴリズムである．つまり，いつの時点で探索を中止しても，並列系全体の最良解は最低温度で改善の処理が行われており，最良解近傍の局所最適解をできる限り求めようとしている．またどの時点においても，並列系全体で見てあまり質の良くない解は高温域で改善処理され，広い範囲で解を探索している．これに対し，逐次焼き鈍し法を基本にした方式，つまり温度を漸減させその後低温から高温に再設定することを試みるアルゴリズムの場合，探索中止時点によっては，系全体の最良解が最低温以外で改善処理されていたり，質の良くない解が最低温で改善されていたりと，一様ではない．このように，いつ探索をやめてもその時点での最善解付近での局所最適解が，すなわち大域最適解が探索されている状態にある，という点も特長の一つである．さらに，PE 間の通信は解が交換される時だけ，それも隣のプロセッサとだけ通信を行えば良く，並列実行時の通信量も低く押さえられている．

PopKern は，これらの特長を持つ温度並列焼き鈍し法を採用している．

利用者定義部分

温度並列焼き鈍し法は，基本的に一定温度での逐次焼き鈍し法での探索を繰り返すので，利用者は，ランダムな初期解候補の生成と解候補をランダムに変更するサブルーチンを定義する．

温度並列アルゴリズムを採用したために必要となるのは，PE の温度分布を定める関数と，隣の PE と解を交換する周期を設定する関数である．これらは，逐次焼き鈍し法

```
遺伝的アルゴリズム
出力: 準最適解
begin
    1. 時刻 $t := 0$;
    2. 初期遺伝子集合 $G(0)$ を生成;
    3. while 終了条件が満たされない do;
       begin
          (a) $G(t)$ の各個体 (遺伝子) の適応度を計算;
          (b) 適応度を参考に，$G(t)$ から個体を選択;
          (c) 選択した個体を交叉交配し，次世代の個体を生成する;
          (d) 選択した個体を突然変異させる;
          (e) $G(t+1)$ を作成;
          (f) $t := t + 1$;
       end;
end.
```

図 3.14: 遺伝的アルゴリズムの概要

での時間方向の温度変化関数を参考に作成すれば良い．

並列化のために必要になるのは，シリアライズ/アンシリアライズ部分である．これは 4 つの最適化アルゴリズムに共通するルーチンだけで良い．

3.5.4 遺伝的アルゴリズム

遺伝的アルゴリズムの概要

遺伝的アルゴリズムは，生物学的進化を模したシミュレーションを行うことでより優れた解を得ようという手法である．つまり，解を遺伝子と対応させ，コストの低い解は優秀な遺伝子であるため生き残りやすいという環境のもとで，遺伝子の交配，突然変

異を繰り返してより優秀な遺伝子が残ることを期待するものである．

遺伝的アルゴリズムの概要を図 3.14に示す．まず，組合せ最適化問題の 1 つの解は，1 つの「遺伝子」にコード化される．遺伝子とは，対応する解を一意に定めることのできる表現形式で，生物の遺伝子と同様，2 つの親遺伝子を交配することで子供の遺伝子が生成できるもののことである．交配の際には，やはり生物の遺伝子の場合と同様，親遺伝子の性質の一部が子供遺伝子に伝わり，子遺伝子は 2 つの親遺伝子の両方の性質を持つような手法が取られる．この時，遺伝子が表現している解についても同じ性質を持つことが必要である．つまり，交配によってできた子の解は，2 つの親の解両方の特徴を備えた解でなければならない．これは，生物学的進化のシミュレーションのために必要な要件である．生物学的な進化は，両親の良い性質を受け継いでより良い性質を持つようになった子供が産まれ，生き残っていくことによって起きる．これと同じことが，親子の遺伝子，さらにそれらが表現している親子の解の間で起きなければならない．従って，親の解の性質を子供の解が受け継ぐように，遺伝子の表現や解への対応を定めなければならない．

遺伝子は一般に，生物学的遺伝子とのアナロジーで，解を記号列として表現していることが多い．しかしもちろん，遺伝子を必ず記号列で表現する必要はなく，特殊なデータ構造，例えばリスト構造や木構造であってもまったく問題ない．遺伝子が，前述した条件，すなわち

- 解を一意に定めることのできる表現であること
- 親の解の性質が子の解に受け継がれるような交配手段が用意されること

を満たしている限り，どのような表現形式でも遺伝子になりうる．

遺伝的アルゴリズムでは，遺伝子の「適応度」，つまり世代交代での遺伝子の生き残りやすさを表現している解の評価値と対応させている．つまり，表現している解の質が高く評価値が高いほど，遺伝子は生き残りやすい良い遺伝子だ，ということになる．そして，この適応度が最大の遺伝子を探索することが，最高の評価値を持つ解を探索することに相当し，遺伝的アルゴリズムの目的となる．

さて，図 3.14では，最初に初期遺伝子集合 $G(0)$ を生成している．これはランダムに生成した実行可能解をコード化したものでも良いし，何らかのヒューリスティクスに基づく初期解でも良い．ただし，ヒューリスティクスに基づく場合は，後述する遺伝子集合の「多様性」を確保できるような配慮が必要となる．こうしてできた遺伝子集合が，自然淘汰を模したシミュレーションに基づいて世代交代していく．

自然淘汰説では，環境への適応度が高い個体が生き残り，子孫を残すことで進化が起きるとされる．これをシミュレーションするため，まず遺伝子集合内の個体の適応度を計算し，次世代に子孫 (や自分自身) を残せる個体をこの適応度に従って選択する．選択戦略には，適応度に比例した確率で遺伝子を決定するルーレット戦略，ランダムに選んだ遺伝子のうち適応度の高いものを残すトーナメント選択戦略，遺伝子集合中で高い適応度を持つ遺伝子を優先的にそのまま残すエリート保存戦略等，様々なものがある．

次に，選択された 2 つの遺伝子を親とし，それらを交配させることで子の遺伝子を生成する．多く用いられている，記号列による遺伝子表現の場合の交配時には，親遺伝子である記号列をいくつかの部分に分割し一部を入れ替える，交叉演算と呼ばれる操作が行われる．交叉演算もまた，一点交叉，多点交叉など様々な種類がある．これにより，子は両親の遺伝子を受け継ぐ遺伝子となる．

さらに，一定の確率で，遺伝子の一部をランダムに変更する操作が加えられる．これが突然変異に相当する．両親からの交叉による遺伝が大局的な探索を実現しているのに対し，突然変異は局所的変動によるランダムな探索を提供していることに相当する．なお，交配や突然変異によって，実現不能な解に相当する遺伝子が生まれることがあるが，これは「致死遺伝子」と呼ばれ，最低の適応度が与えられるため，次の世代で淘汰されることになる．

こうして次の世代の遺伝子集合が生成され，この世代交代のステップを繰り返すことで，遺伝子集合は，適応度の高い良質の解が含まれる集合へと進化していく．

遺伝的アルゴリズムで注意しなければならないのは，遺伝子集合の多様性を保つ，という点である．遺伝子集合がある 1 種類の遺伝子で埋め尽くされてしまった場合，どのように交配を行ってもそれ以上改善された遺伝子は生まれない．この場合，突然変異による改善を期待するしかないが，ほとんどの場合は一様化した遺伝子に淘汰されてしまう．1 種類の遺伝子に埋め尽くされる，という極端な状況でなくとも，同一の傾向を持った遺伝子ばかりになるだけでも進化は阻害される．また，現在の遺伝子中ではそれほど良い適応値を示していない遺伝子の一部分であっても，別の遺伝子と交配することによって初めて高い適応値を示すようになることもある．従って，進化を続けるためには必ずしも評価値の高いものばかりでなく，様々な可能性のある遺伝子を集合内にとどめておかねばならない．遺伝子集合の多様性を保つためには，選択戦略や淘汰戦略，遺伝子表現や交配演算等，あらゆる部分で注意する必要がある．ここで，遺伝子集合のサイズを大きくする，ということは遺伝子の多様性を保つために非常に有効な手段である．しかし，遺伝子集合を余り大きくしすぎると，世代交代の手間が大きくなったり，多様性を重視しすぎるあまり良質の解への収束速度が遅くなったりという弊害も出るため，

気をつける必要がある．

同様に確率的に準最適解を探索する焼き鈍し法との違いは，遺伝的アルゴリズムの方は多様性のある状態を保った集団により探索が進行する，という性格を持つことである．また，遺伝的アルゴリズムには，集団による大局的探索能力はある反面，局所探索能力が弱いので，最適解近傍での収束が遅いという欠点がある．そのため，探索の終盤で山登り法等の局所探索能力の高いアルゴリズムと組み合わせて探索することが有効な手段となる．

並列化手法

基本的に，遺伝的アルゴリズムの並列化は，遺伝子集合を分割しそれらを PE に分担させて逐次遺伝的アルゴリズムを実行することで行われている．その際，

- 1 つの遺伝子集合を複数の PE で分担するという立場
- 複数の遺伝子集合を複数の PE で分担するという立場

という 2 つの方向性がある．

前者の場合は，基本的にモデルは逐次遺伝的アルゴリズムと変わらず，交配，突然変異等の操作を複数の PE で行う事によって並列動作させようという方式である．通信量の軽減，同期処理の軽減等の工夫は様々にあるが，基本的な立場は 1 つの遺伝子集合を共有するというものである．

後者の場合は，逐次遺伝的アルゴリズムとモデルが異なり，島モデルと呼ばれる．この名前は，海で隔てられた複数の島での進化との類似性に由来している．

島モデルでは，遺伝子集合はいくつかに分割され，それぞれの部分が独立して世代交代を進める．これは，外界からの影響のない島での生物の進化に相当する．孤島での進化は，遺伝子集合のサイズが小さいので，高速に良質の解へと収束していく反面，遺伝子の多様性を失って進化がすぐに停止してしまう恐れがある．そこで，ある一定世代周期ごとに，遺伝子の一部が他の遺伝子集合へと移動し，元々その島に存在していた遺伝子と混合される．これはちょうど，動物が島の間を移住したことに相当する．こうして，定期的に遺伝子の移動と混合，淘汰が行われることで，遺伝子集合の多様性は保たれ，進化が持続するのである (図 3.15参照)．

このように，島モデルでの基本的な立場は，複数の遺伝子集合が遺伝的アルゴリズムに基づいて探索を行っており，それらの相互情報交換によって全体としてより大きな探索が進む，というものである．この方式の場合，基本的に 1 つの遺伝子集合を 1 つの PE が担当すれば効率の良い実行が期待できる．

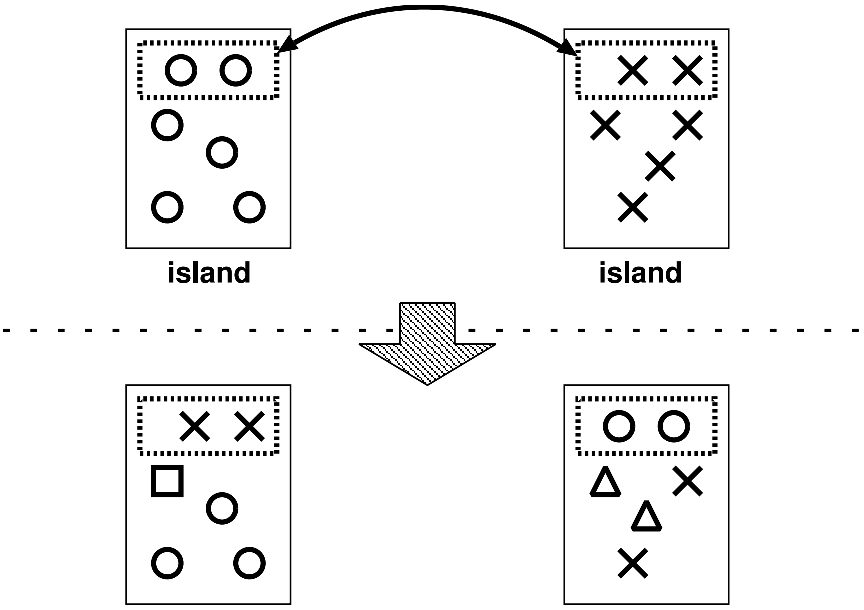

図 3.15: 島モデル

1 つの遺伝子集合を用いるモデルの場合，逐次遺伝的アルゴリズムとモデルが同じため，探索の結果は逐次時と同じ解が得られるはずである，という利点がある．しかし，PE 間で共有する情報が多いため，世代交代のたびに変化した情報を全てのプロセッサが送りあったり，世代交代のたびに同期処理が必要になったりと，並列効率を妨げる要因が大きい．一方，島モデルの場合は基本的に PE 内で世代交代が行われ，通信は一定期間ごとに移住を行う島の間でのみ行えば良く，並列効率が上げやすいモデルであると言える．そこで，PopKern では島モデルを採用する．

問題領域依存知識を重視するための戦略

遺伝的アルゴリズムは，遺伝子集合に対する選択操作，交配操作，そして次世代の遺伝子集合の生成操作等によって成立しており，前述した通り，これらはそれぞれ様々な方式が知られている．ある程度の方式をライブラリとして提供することは可能であるが，利用者の意図する方式を完全に実現させるためには，これらの操作をライブラリ側で用意すべきではないと考えられる．また，問題領域依存の知識を活用して遺伝子集合のデータ構造に工夫を加えることで，これらの操作の高速化が実現できることも考えられる．例えば，複数の観点からの適応度のソートが必要な場合，遺伝子へのポインタをソートして複数保持する，等の複雑な構造を取ることもできる．

このような理由から，PopKernでは遺伝子集合のデータ型は利用者定義の部分とした．遺伝子集合のデータ型が利用者定義であるので，選択等の操作もまた利用者定義とした．

利用者定義部分

前述したように，PopKernは遺伝子集合を利用者定義のデータ型とし，遺伝子集合に対して操作を加えている．

従って，利用者は遺伝子集合のデータ構造を定義する．さらに，この遺伝子集合に対して，遺伝的アルゴリズムに必要な操作を行うルーチン群を定義する．

また利用者は，図3.14中に示した手続きを定義するのであるが，図中(3a)~(3e)の部分の操作が全て利用者定義となるので，これらを全てまとめ，遺伝子集合を次の世代に変更するルーチン，と一体化して利用者定義とすることにした．

また，島モデルを用いて並列遺伝的アルゴリズムを実現するために，

- PE間の遺伝子移動が起きる世代間隔を定めるルーチン
- 移動する遺伝子集合を選択し，元の遺伝子集合と分割するルーチン
- 移動する遺伝子集合のシリアライズ/アンシリアライズルーチン
- 他のPEから移動してきた遺伝子集合を，元の集合と混合するルーチン

を定義する必要がある．遺伝子集合の分割，及び混合には様々な戦略があるため，これらの操作は利用者定義とするべきである．

これらは逐次遺伝的アルゴリズムのプログラミングの時には必要がなかった物であるが，移動が起きる世代間隔の決定，移動する遺伝子集合の選択及び混合のルーチンは，島モデルを採用する場合には必須の部分であり，また島モデルは比較的一般的なモデルなので，問題領域依存の知識を十分に持った利用者にとって，これらの部分を定義することはそれほどの労力にはならないと考えられる．よって，PopKernを使うことによって余分に定義しなくてはならないのは，シリアライズ/アンシリアライズのルーチンのみである．

3.5.5 その他のアルゴリズム

現在PopKernが提供しているのはこれまでに述べた4種類のアルゴリズムである．しかし，当然のことながら，これだけのアルゴリズムで全ての組合せ最適化問題に対応

できる，というわけではない．例えば，ゲームに用いられるミニマックス木の探索は，現在のPopKernでは行うことができない．しかし，この探索ができない理由はPopKernの基本設計にあるのではなく，まだミニマックス木探索を実装していないという，単にそれだけの理由である．

ミニマックス木の探索は，普通$\alpha\beta$探索というアルゴリズムを用い，その並列化手法も広く研究されている．利用者が持つ問題領域依存知識を最大限生かせるようなライブラリを設計するためには，分枝限定法の場合のように，探索途中のノード集合を利用者定義とし，探索を行うという枠組みが良いと考えられる．しかし，効率の良い並列$\alpha\beta$探索のためには，ノードの探索深さ，探索順序に関する情報を必要とする並列アルゴリズムが多い．従って，利用者に所定の条件を満たすノード集合を分割してもらったり，ノードの情報をもらったりするルーチンを定義してもらう必要があるかもしれない．

多少利用者に余分な労力をかける必要があるかもしれないにせよ，ミニマックス木の探索は現在のPopKernの枠組みの中で実現することができる．つまり，利用者の問題依存知識による設計が可能な部分は逐次プログラミングの知識だけを用いて逐次プログラミングを行い，それを探索アルゴリズムの制御系にプラグインすることで探索を行う，という現在の枠組みを変えることなく，ミニマックス木の探索をPopKernに組み込むことができる．

また，タブーサーチというアルゴリズムも良く使われている．これは，暫定解の近傍を順次山登りのように探索することで準最適解を求めるためのヒューリスティックに基づいたアルゴリズムであり，探索した暫定解を「タブーリスト」として保持し，このリストを使って局所最適解に陥ることを避けるという手法を用いている．これも，山登り法と同様の枠組みで，PopKernに組み込むことができると考えられる．

このように，現在提供しているもの以外のアルゴリズムでも，PopKernの枠組み内で提供することができ，PopKernは組合せ最適化問題の汎用なライブラリとしての機能を十分備えていると言える．

現在未実装のアルゴリズムのうち，適用可能問題領域が大きく広がるミニマックス木の探索は，実現したい機能の1つであり，現在は設計の段階にある．

3.6 設計の考察とまとめ

PopKernの設計の前提は，問題領域依存知識の活用に重点を置いた並列化ライブラリを作成する，というものである．そのため，基本的な設計方針として以下のように考えた．すなわち，並列組合せ最適化アルゴリズムには問題領域依存のデータ型，及びそ

の操作部分と，操作をコントロールすることで組合せ最適化アルゴリズムを実現する問題領域非依存の部分とが存在している．その問題領域依存の部分だけを切り出して利用者に逐次プログラミングしてもらい，ライブラリにプラグインすることで所定のアルゴリズムによる探索を実現できるはずである，という考え方である．

この章で述べてきた通り，PopKern の設計にあたっては，利用者定義部分とライブラリ提供部分の切り分けを注意深く検討してきた．その結果，このライブラリの設計は，ある並列組合せ最適化アルゴリズムのもとで考えられる問題領域依存知識によるチューニングを，利用者が最大限実現できるものとなった．

ここで，利用者が自由に記述できる部分が大きいということは，利用者に対する負担が大きい，ということにつながる，と考えられうる．事実，PopKern は一般のライブラリと比較して，利用者が最低限定義すべき部分が大きい．

しかし，実際には利用者の負担はそれほど大きなものではない．第 3.4章で述べたように，アルゴリズムに独立な共通部分での利用者定義では，利用者の余分な負担はシリアライズ/アンシリアライズに関するルーチンを用意する部分のみである．また，第 3.5 章で述べたように，それぞれの最適化アルゴリズムについての利用者の余分な負担も，基本的にはシリアライズに関する部分のみである．遺伝的アルゴリズムの場合のように，並列化アルゴリズムを用いることによる利用者定義部分の増加も多少はあるが，これも並列化アルゴリズムをプログラミングする際には必ず設計しなければならない部分であるので，それほど負担が増えたとは言えない．

さて，現在の PopKern が持つ組合せ最適化問題探索能力の限界は，

- 提供アルゴリズムの不足による，適用可能問題領域の制限
- PopKern の基本設計に関わる制限

の 2 つに分けられる．

前者の例は，$\alpha\beta$ 探索を提供していないため，ミニマックス木の探索に適用できない，という問題である．これは 3.5.5章で述べたように，多くのアルゴリズムは PopKern の基本設計の枠組みで実現できるので，必要な並列化アルゴリズムを順次増やしていくことで解決できる限界である．

これに対し，後者の限界を解決することは難しい．この例として，PE 間で利用者がある データ構造を共有する，という場合を考える．PopKern は，通信，同期の機構を利用者から隠蔽している．そのため，利用者定義のデータ構造の一部が別の PE にあり，利用者が自分自身の作業ルーチン内でその別の PE の部分の内容を参照する，という事ができない．これを可能にするためには，通信のプリミティブを利用者に提供すればよ

いが，並列プログラミングの知識がない利用者を対象としたライブラリ，という当初の目標から外れたものになる．

共有する部分のデータを PopKern に登録してもらい，それに対する排他的なアクセスプリミティブを提供する，という方法もあるが，並列動作を意識させないライブラリ，という大前提を満たすためには，利用者定義のデータ構造と PopKern が管理するデータ構造との設計上の切り分けをやり直す必要があるであろう．

例えば，分枝限定法のデータ構造で言うなら，現在の設計のままだと，PE 内でだけ参照されるノード集合のデータ構造と，共有されているノードのデータ構造との 2 種類を作ることになるが，利用者は常に操作対象ノードが共有されているかそうでないかを意識しなければならない．これはライブラリの目的に反する．この場合は「1 つのノード」を利用者定義データ型，その木構造は PopKern が把握するものとして，常に「1 つのノード」へのアクセスを PopKern 経由で行うことにすれば，通信や同期は利用者から隠蔽されて，並列動作を意識することなくライブラリを使えると考えられる．もちろん，この場合利用者の問題領域依存の知識を活用できる部分が少なくなり，PopKern のもうひとつの大前提である，問題領域依存知識の重視」が満たされなくなる．

このように，PopKern の基本設計に関わる制限も確かに存在する．しかし，今述べたようにこれらの制限は PopKern の目標とする条件から発生したものであり，設定したライブラリの目的を追求するための基本設計としては，妥当なものであったと考えられる．

第 4 章

実装

4.1　並列処理系

PopKern は，並行並列論理型言語 KL1[17] のポータブルな処理系である KLIC[5] 上に実装されている．

KLIC は KL1 のプログラムを C プログラムに変換して実行する．このため，適切に最適化された C コンパイラが存在する計算機上ならば，どこでも高い効率で動作することができる．また，KLIC の並列実装は，共有メモリ，PVM，MPI 等，多くの通信路上への実装が行われており，大規模並列計算機から，小規模な共有メモリ並列計算機，PC クラスタ等，多くの並列計算環境で実行できる．従って，KLIC 上に実装された PopKern も多くの並列計算環境で動作することができる．

KLIC は “ジェネリック・オブジェクト” と呼ばれるデータ型を扱うことができる．これは，論理型プログラミングの世界に新たなデータ型を加えるための枠組みである．この枠組みを使うことで，手続き的逐次プログラミングを宣言的並行プログラミング中に簡単に導入することができる．PopKern では，利用者が定義した C 言語による逐次プログラミングのデータ構造やその操作ルーチンを，ジェネリック・オブジェクトによって KL1 のデータ型に導入している．利用者定義部分は全てジェネリック・オブジェクトにカプセル化され，PopKern システムは完全に KLIC のデータ型として扱っている．従って，通信や同期は KLIC のシステムにより，他の KLIC のデータと同様に扱われている．

4.2　基本構造

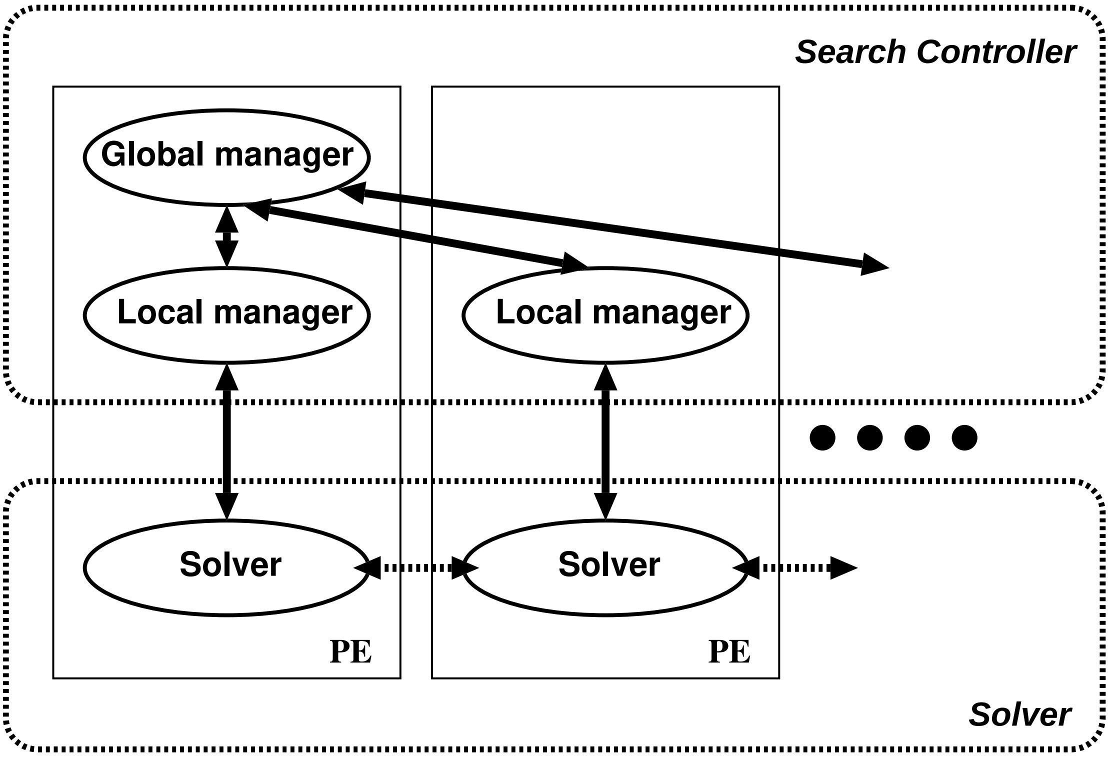

図 4.1: 実装の基本要素

PopKern は複数の組合せ最適化アルゴリズムを提供するライブラリ・パッケージである．従って実装の際には，アルゴリズム非依存の部分を共通化し，そこに各々のアルゴリズムによる探索部分を組み合わせることで，各種の組合せ最適化アルゴリズムによる探索を実現するよう，実装方式を定めた．

図 4.1に実装の基本構造を示す．大きく「探索コントロール部」と「ソルバ」に分けられており，「探索コントロール部」がアルゴリズム非依存の部分である．

探索コントロール部は，1 つのグローバルマネージャ，複数のローカルマネージャによって構成されている．これらは基本的に大域最適解の更新，保持を行う．また，探索の停止を指示したり，システム終了時に最適解の保存を行うのもこの部分の役目である．

ソルバは，探索コントロール部の指示のもと，実際に所定のアルゴリズムによって探索を行う部分である．この部分がアルゴリズムごとに変更されている．ソルバはソルバ同士及びローカルマネージャとだけ通信ができる．

このような実装法を用いることにより，将来新たなアルゴリズムを実装する際の労力を小さく押さえることができると考えられる．また，PE を複数の群に分け，複数の

アルゴリズムを使って同時に探索する，という方式を実現する際にも効果があると思われる．例えば，焼き鈍し法のようなヒューリスティックに基づいたアルゴリズムで準最適解を早く求め，その情報を使いながら分枝限定法を行うことで枝刈りの効率をあげる，といった方式である．さらに，個々の並列計算機のネットワークトポロジに最適化する際の労力を押さえることもできる．これは後述する．

これらの構成要素は，KLIC の提供する「プライオリティシステム」によって管理されている．つまり，実行の優先度が最も高いのはグローバルマネージャであり，その次にローカルマネージャ，最後にソルバという順に優先度が設定されており，このシステムによってコントロール部がソルバをコントロールできるのである．

4.3 グローバル/ローカルマネージャ

3.3章で述べたように，PopKern は 1 つのグローバルマネージャと複数のローカルマネージャを作る．ローカルマネージャからの通信路はグローバルマネージャにのみつながっており，ローカルマネージャ同士では通信できない．一方，グローバルマネージャからは全てのローカルマネージャへというブロードキャスト通信ができる．

実際の並列計算機は，様々な通信ネットワークトポロジをとっている．従って，通信路を設定するときにはその計算機にあったネットワーク構成を取ったほうが高速な通信ができる．しかし基本的に，この通信路を行き交うのは最良解情報のみであるので，この通信路は帯域をそれほど必要としない．従って，グローバル/ローカルマネージャ間の通信路の構造はそれほど気にしなくても良いと考えている．

このグローバルマネージャとローカルマネージャの役割は，基本的に

- 探索のコントロール，表示
- 全 PE についての現時点の最適解のコストを把握すること

の 2 つである．

探索のコントロールについては，採用した並列化アルゴリズムが，実行時のコントロールをあまり必要としないものばかりであるため，探索開始・終了のコントロールをする程度の役割しかない．具体的には，探索時間の管理をグローバルマネージャが行い，設定した探索時間が過ぎると，終了命令を放送，ローカルマネージャ経由でソルバを停止させる．また，最良解が更新された時点で，利用者定義の終了判定ルーチンを呼び，終了指示があった場合は同様にシステムを終了させる．システム終了時には，グローバル

マネージャは最良解を保持しているローカルマネージャを把握しているので，そのローカルマネージャから最終的な解を受け取り，出力する．

探索状況の表示は，大域最良解，局所最良解が更新されるたびに必要かどうか判定される．ファイルI/Oやコンソールはグローバルマネージャが一括管理しているので，表示の際にはグローバルマネージャまで解が送られることになる．

現時点の最良解の更新・保持は，ローカルマネージャの保持する最良解が更新されたらグローバルマネージャに送信，グローバルマネージャが持つ大域最良解が更新されたら全ローカルマネージャに放送，というアルゴリズムで行われている．この際，ローカルマネージャの同期等は行われていないので，最良解の情報が全ローカルマネージャに通知されるまでにはタイムラグがある．

4.4 各アルゴリズムのソルバ

ソルバは基本的にローカルマネージャからの指示によって動作する．グローバルマネージャへの直接通信はできない．

アルゴリズムによっては，ソルバ同士が通信路を持ち，通信しあって探索することがある．この時の通信ネットワークの張り方はローカルマネージャとグローバルマネージャの持つネットワークとはまったく独立に設定することができ，実際異なったネットワークになっている．

ソルバ間の通信は，比較的広い帯域を必要とする場合がある．特に，遺伝的アルゴリズムの場合は遺伝子集合の移動が起きるため，通信量が多くなりがちである．そのため，ソルバ間の通信は実際の並列計算機に合ったネットワークトポロジを取ったほうが性能向上につながると思われる．今回の設計では，ソルバのネットワーク構成が探索コントロール部に影響を与えないので，簡単にネットワークの最適化が行えると考えられる．

4.4.1 山登り法

山登り法のソルバは単純に山登りを繰り返すだけである．ソルバ間通信路は存在しない．

4.4.2 分枝限定法

分枝限定法の場合，動的負荷分散機構が必要となる．3.5.2章で述べたように，負荷分散はノード集合の分割，結合によって行われるが，このノード集合の管理をマスタ・

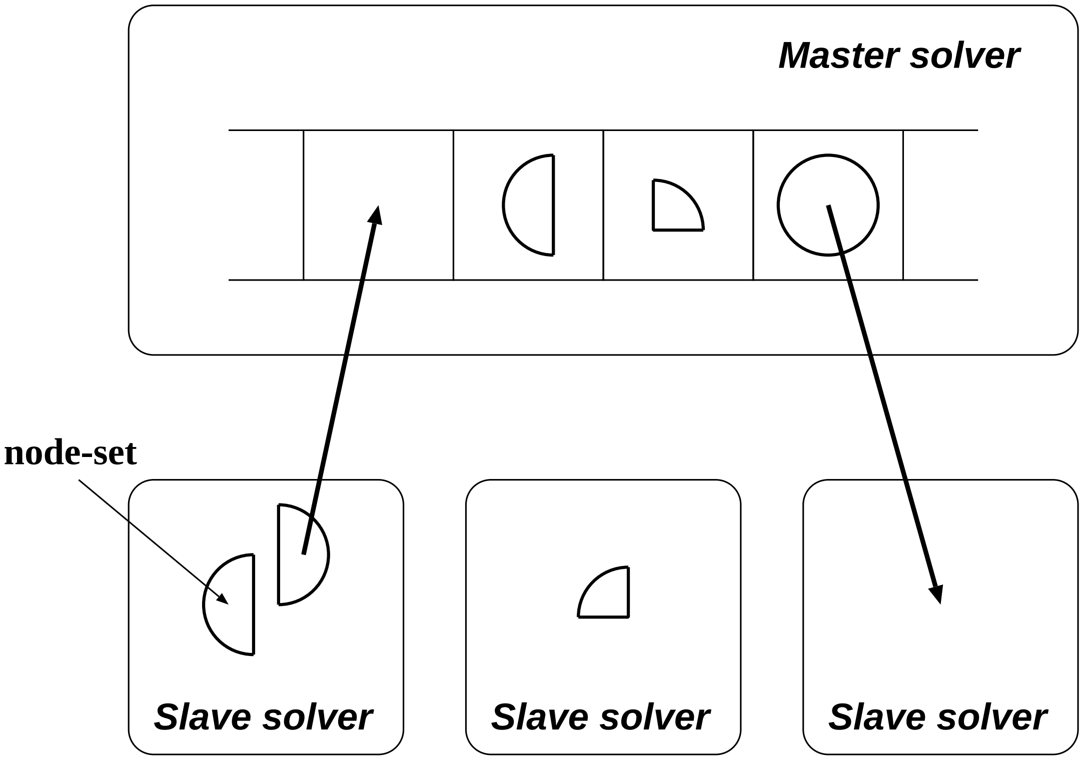

図 4.2: 分枝限定法のノード集合管理方式

スレーブ方式の一種で行っている．

図 4.2にノード集合の管理方式を示す．まず，ルートノードを含むノード集合を 1 つのスレーブソルバに割り当て，分枝限定法による探索を始める．探索中のマスタ/スレーブソルバの動きは，次のようになる．

1. スレーブソルバは，自分が保持するノード集合が空になると，マスターソルバにそのことを通知し，待機状態になる．
2. マスターソルバは，ノード集合をためておくキューを持っており，そのキューの中に未探索のノード集合が存在していれば，待機状態に入っているスレーブソルバにノード集合を送信する．キューの中にノード集合がない場合は，ノード分割要求を全てのスレーブソルバに対してブロードキャストする．
3. ノード分割要求を受けたスレーブソルバは，自分の保持するノード集合の分割を試みる．ノード集合が分割できたら，マスターソルバに送信する．

このようにして探索中の負荷分散が行われる．全てのスレーブソルバが待機状態に入り，

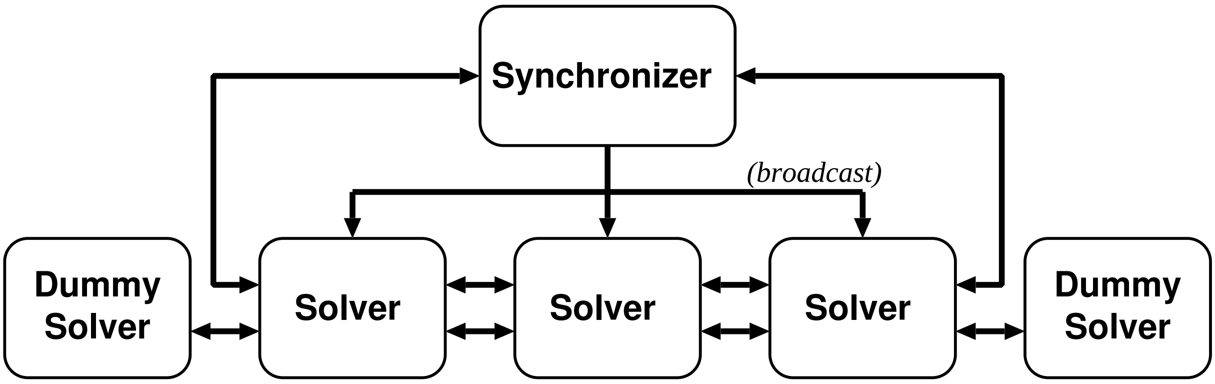

図 4.3: 焼き鈍し法のソルバ構成

マスターソルバにも未探索ノード集合がなくなると探索終了である．

なお，現在は実装の関係でソルバと探索コントロール部が一体となったシステムとなっている．

4.4.3 焼き鈍し法

温度並列焼き鈍し法では，1 つのソルバが 1 つの温度帯を担当している．従って，利用者が指定した温度帯の数だけソルバが生成される．それに伴って，グローバルマネージャとの通信担当であるローカルマネージャもソルバと同じ数だけ，つまり温度帯の数だけ生成される．このソルバが，各 PE に均等に割り当てられる．

ソルバは隣の温度帯を担当しているソルバと通信路を持っている．よって，ソルバ全体では温度順に一列に通信路がつながった構造を持つ．この，隣の温度帯との通信路は 2 本あり，1 本は解を交換するため，もう 1 本は同期のために使われる．解交換用の通信路はその両端がダミーソルバにつながっており，同期のための通信路はその両端を同期マネージャに接続している．同期マネージャはソルバへのブロードキャスト用通信路を持っている．

それぞれのソルバは設定された解交換周期まで，独立して一定温度による焼き鈍しを続ける．解交換周期に達すると，同期のための通信路を接続し，待機状態に入る．全てのソルバが解交換周期に達すると，同期のための通信路が完全に接続され，同期マネージャが同期を確認することができる．(この同期確認手順は並列論理型言語に特徴的なアルゴリズムであり，論理変数の単一化を利用している．) 同期マネージャは，同期を確認後，解交換命令をソルバに向けてブロードキャストする．解交換命令を受けてソルバは隣の温度のソルバと通信，アルゴリズムに従って解を交換する．この時，ダミーソ

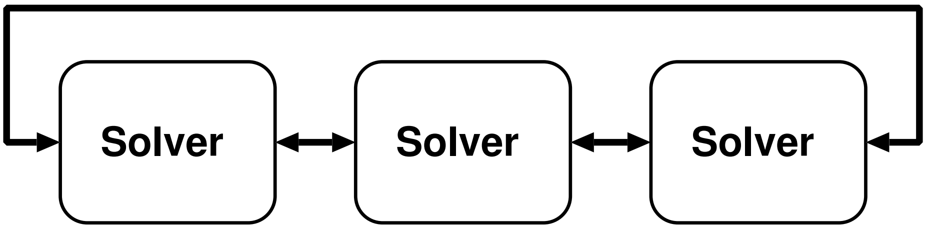

図 4.4: 遺伝的アルゴリズムのソルバ構成

ルバは「まったく解を交換しないソルバ」として働く．

4.4.4 遺伝的アルゴリズム

遺伝的アルゴリズムでは，1PE に 1 つずつソルバが割り当てられ，それぞれ 1 つの遺伝子集合，いわゆる「島」を担当する．それぞれのソルバは隣の PE のソルバと通信路で接続されている．両端のソルバはお互いつながっているので，全体としてソルバはリング状につながっていることになる．これは，島同士の結合に対称性を持たせるためである．

ソルバは指定された遺伝子交換周期だけ独立して世代交代を行い，隣のソルバと遺伝子集合を交換する．この時，全体の同期は行っていない．これは，少しでも同期を減らすことによって性能を向上させようとしたためである．

第 5 章

評価

5.1 評価環境

以降の評価は，SUN Ultra Enterprise E/4000 (Ultra SPARC 166MHz × 10，メモリ 1280MB) 上で行った．KLIC システムのバージョンは 3.003 である．KLIC には，様々な低レベル PE 間通信手段のための実装が存在するが，今回の評価は全て，通信路として共有メモリを用いたメッセージパッシングモデルの実装を使っている．

5.2 巡回セールスマン問題による評価

巡回セールスマン問題を解くプログラムを使って，PopKern ライブラリの評価を行った．

巡回セールスマン問題は様々な研究が行われ，高速な探索法が多数発表されている [15] が，本評価では非常に単純な探索方法を用いていることに留意されたい．

5.2.1 逐次オーバーヘッド

山登り法を使った巡回セールスマン問題ソルバーに関して，最適解の管理，探索のコントロール等の PopKern が提供している機能を，全て C 言語で逐次記述したプログラムを用意した．

連続して訪問する 4 都市の全ての組み合わせに対し，訪問順を変更した経路を生成し，経路長が短くなるならばその訪問順を変更する (図 5.1) という方式で山登り法における暫定解の近傍探索を行っている．

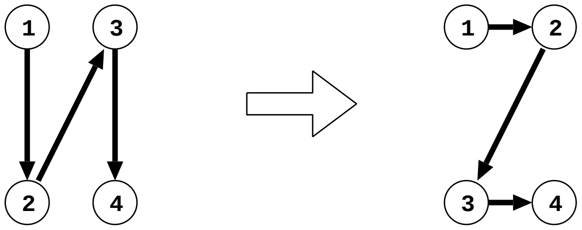

図 5.1: 山登り法での探索 – 隣接 4 都市の順序入れ替え

表 5.1: Popkern システムの実行時間オーバーヘッド

逐次プログラム実行時間 (sec)	Popkern 実行時間 (sec)
45.18	45.62

この逐次版の山登り法プログラムと，PopKern によって並列化されたプログラムの逐次実行時の性能を比較することで，PopKern の基本的なオーバーヘッドが求められる．

性能比較は，同一問題，同一乱数列を用いて，ある同一解が発見されるまでの時間を比較することにより行った．実験は 5 回行い，その平均実行時間を求めた．結果を表 5.1にまとめる．

この結果から PopKern システムの逐次オーバーヘッドを計算すると約 1%となり，オーバーヘッドは十分小さいと言える．

5.2.2 分枝限定法の並列効果

評価プログラム

評価に用いたプログラムは，未訪問の都市のそれぞれから最近傍にある都市までの距離の総和を求め，これと現時点での経路の距離との合計を現時点の部分解の上界値とし，分枝限定法の上界評価を行うことで探索の枝刈りを行っている．また，探索ノードのスケジューリングは，深さ優先である．別の PE へ移動するために探索ノード集合を分割する際には，未探索のノードのうち探索が最も進んでいない (すなわち一番浅い) ものを移動することとした．ただし，探索木の末端から一定の閾値以下の深さしかないノードは移動しないこととし，探索の粒度を一定以上に保っている．なお，評価に使用する問題は，TSPLIB[14] より選択した．

表 5.2: 分枝限定法の探索時間 (msec)

PE 数	A12	A14	A16	B14
1	15018	196845	2367016	2341858
2	3995	72985	843257	381374
3	2651	46992	513681	226957
4	3017	38249	319242	256176
5	2676	39546	310873	188527
6	2721	42324	311250	190846
7	2740	39937	312089	183997
8	2655	41027	321554	185942

評価

表 5.2に，全ての経路を探索して最良解を見つけるまでの実行時間を示す．A12, A14, A16, B14 は異なった都市配置の巡回セールスマン問題を表している．A12 は 12 都市，のように末尾の数字は都市数であり，A から始まるものは TSPLIB の問題 bayg29 の先頭の都市を必要な数だけ残したもの，B14 は TSPLIB の問題 burma14 となっている．

図 5.2は実行時間の台数効果である．探索にかかった実行時間について，1PE での実行時間と複数 PE での実行時間の比を台数効果とした．

図 5.2に示されるように，このテストプログラムは非常に高い台数効果が得られている．4 ~ 5PE 以下の領域では，スーパーリニアな速度向上が 4 つの問題全てにおいて認められる．これは，逐次実行時のチューニングが不十分で，探索木のノードの展開順が適切なものでなかったためと考えられる．ノードの展開順は枝刈りの効率に大きな影響を与える．PE 数が増えることで，より質のよい解が早い段階で見つかり，枝刈りが効率よく行われたため，探索時間が大幅に短縮されたと考えられる．これは，図 5.3によって確認される．図 5.3は，探索時に展開されたノード数の変化を表している．1PE での探索時の展開ノード数を 1 とし，比をとって示されている．B14 のように大きな台数効果が得られている問題は，1 ~ 3PE の領域で展開されたノード数も大きく減っており，高速化の原因が枝刈りによるものであることが読み取れる．ただし，探索木のノードの展開順を常に適切なものに設定することは難しいため，今回の評価プログラムの速度向上も十分意味のある並列効果であると言える．

さて，図 5.2において，PE 数を増やしていくと台数効果は 4 ~ 5PE でほぼ最大とな

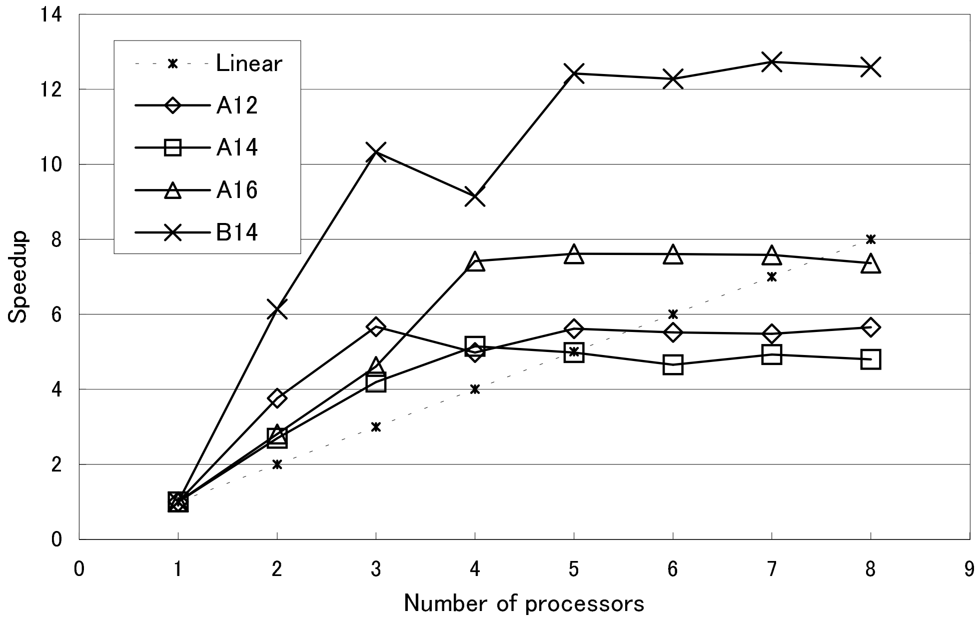

図 5.2: 分枝限定法の台数効果

り，それ以降は横ばいを続ける．並列探索の場合，PE 数を増やすと，逐次実行では枝刈りされていたノードが投機的に展開されてしまうという，無駄な投機実行が増えて並列効果が落ちることが多い．しかし，今回の評価プログラムでは，図 5.3に示されるように PE 数の多い領域でも展開ノード数は増加していない．従って，この領域での並列効果の低下は，PE 間通信の増加による通信オーバーヘッドの増加や，負荷のバランスが崩れてアイドリングする PE ができることなどに起因していると考えられる．

この考察が確認できるのが図 5.4である．図 5.4は，単位時間あたりに展開されたノード数の変化を，1PE での展開ノード数を 1 として示している．全ての PE が 100%の効率で働き続ければ，この単位時間あたり展開ノード数はリニアに増加することが期待される．この図を見ると，どの問題においても単位時間あたりのノード展開能力は 4～5PE 程度で最高となり，以降は横ばいとなったり性能が悪化したりという傾向にある．よって，5PE 程度以上の PE 数では CPU 能力をフルに活用できない CPU が現れてきている，と確認できる．

また，図 5.4を図 5.2と比較すると，総実行時間の台数効果が 3PE で最大となった A12 の問題においても，単位時間あたりのノード展開能力は 5PE の方が高い．この原因は，次のように考えられる．つまり，A12 では問題サイズが小さいため，枝刈りに必

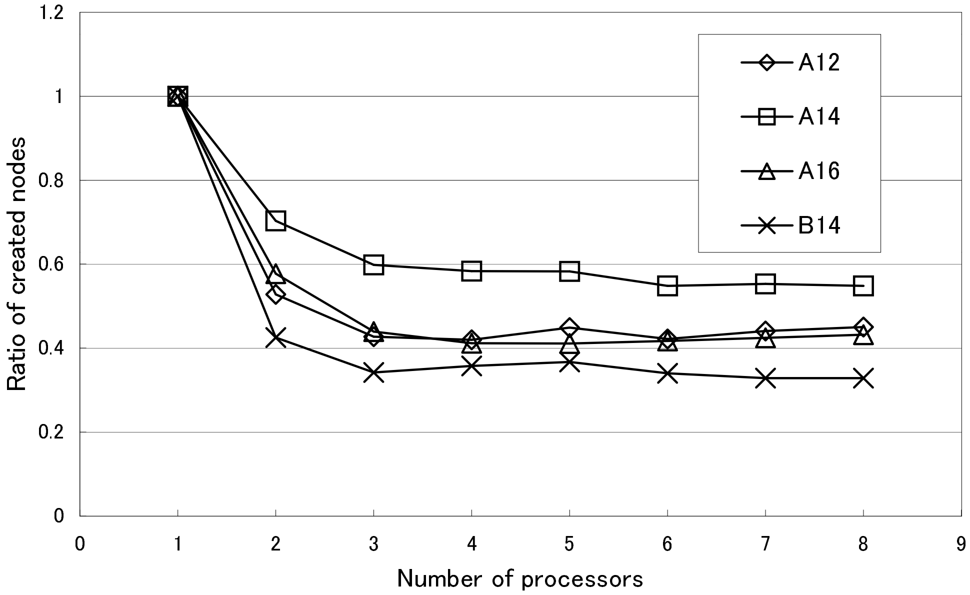

図 5.3: 展開されたノード数の変化

要な探索途中の最良解の情報を PE 間で通信する際のレイテンシが，実行時間に対して相対的に大きなものになる．結果，最良解の共有が探索の進行に比べて遅れるため，無駄なノードを展開することになり，3PE での実行の方がそれ以上の PE を投入する場合より早くなったと考えられるのである．

ところで，今回評価対象とした問題は，その探索空間の規模が非常に大きく異なっているにも関わらず，どれもほぼ同様の PE 数で並列化効率が最大となっている．これは，今回評価に用いたプログラムの並列計算の粒度が問題に関わらず一定であったことに起因すると考えられる．並列実行時の効率は，負荷のアンバランスが起きずかつ通信のオーバーヘッドが大きくならないように探索を分割したときに最大になる．これは，並列計算の粒度，すなわちノード集合の分割単位の大きさによって決定される．現在，粒度は利用者がノード集合分割方法によって定めている．粒度は，使用可能プロセッサ数，計算速度と通信速度の比によって適切に調整できると考えられるので，PopKern システムが利用者に，これらの要素によって計算される何らかの粒度の目安となる情報を提供することができれば，より効率のよい戦略を利用者がとる大きな手助けとなると考えられる．

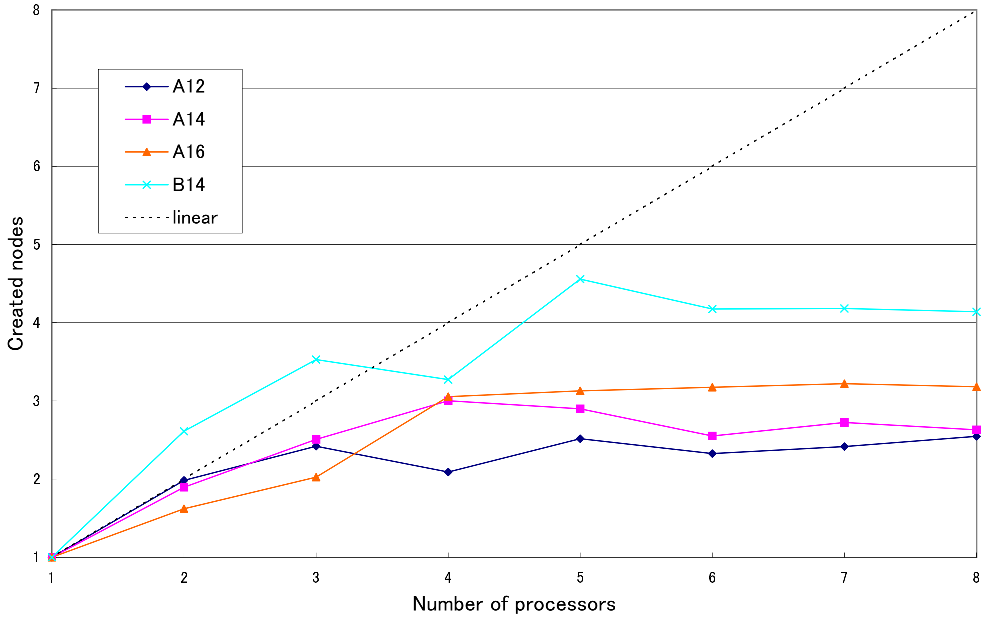

図 5.4: 単位時間当たりに展開されたノード数の変化

5.2.3 焼き鈍し法の並列効果

評価プログラム

評価プログラムでは，ランダムに選択した 2 都市の訪問順を入れ替える (図 5.5参照) ことで，暫定解の近傍の解を生成し，焼き鈍し法を行う．使用 PE 数に関わらず，常に 32 の温度帯を使って温度並列焼き鈍し法を行っている．PE 数 1 の場合は 1PE で 32 温度分の焼き鈍しを，PE 数 8 の場合は 1PE あたり 4 温度分の焼き鈍しを行う，という方式である．隣の温度帯との解の交換周期は暫定解 1000 変更毎とした．また，図 5.1で示した探索による山登り法も併用している．つまり，ランダムに生成した近傍の解が一定回数だけ連続して不受理とされると，山登り探索を試みるのである．

なお，評価結果として示している解改善速度はそれほど速いものではない．これは，ランダムな近傍の解の生成や山登り探索に用いている手法の探索範囲が非常に狭いためであることに留意されたい．

評価

図 5.6は，焼き鈍し法の探索で解がどのように改善されていったかという過程を示すため，その時点で発見されている最良解のコスト (巡回セールスマン問題では巡回経路

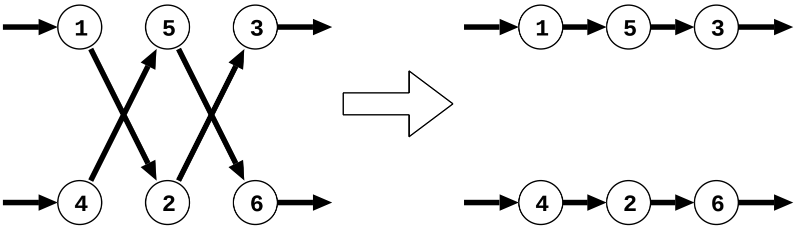

図 5.5: 焼き鈍し法での探索 – 2 都市の訪問順序入れ替え

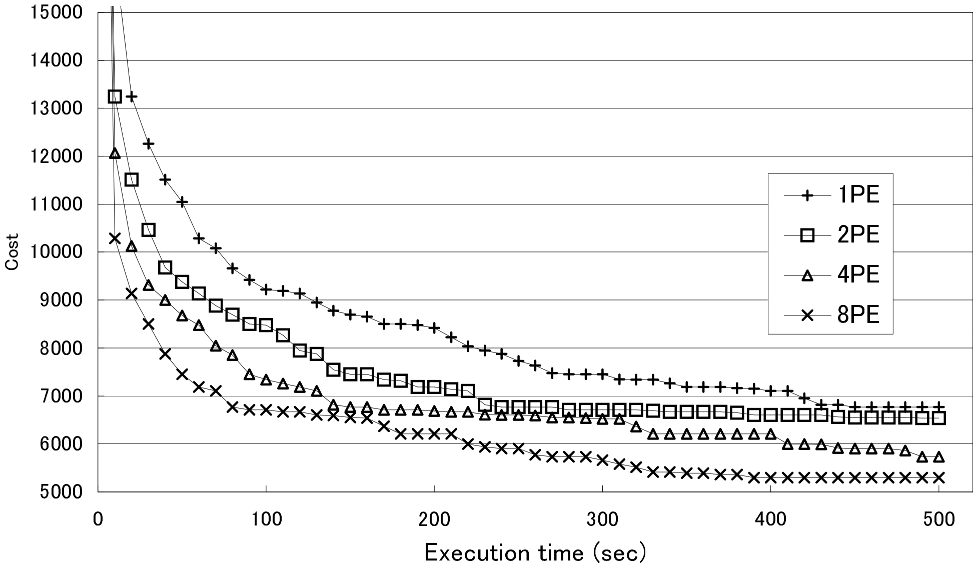

図 5.6: 焼き鈍し法の解候補改善過程

長) を，実行 PE 数ごとにプロットしたものである．使用した問題は TSPLIB の a280(都市数 280) である．

図 5.6に示される通り，解の改善は PE 数が増えるほどより早い時点でコストの低い解が見つかっており，並列計算の効果が現れている．解のコストが 10000 を切った時間，9000 を切った時間... を図 5.6から読み取り，その時間の比をプロットして並列効果を見たのが，図 5.7である．どの時点のグラフも高い速度向上を示しており，8PE でおよそ 6 倍程度の速度向上が見られる．

現在の焼き鈍し法の実装では，PE 数に関わらず並列温度数を一定にして並列実行を行うので，ある解が得られるまでに行われる計算量は，PE 数を変化させても変化しないと期待できる．従って，この解の改善課程からも速度向上を見積もることができると

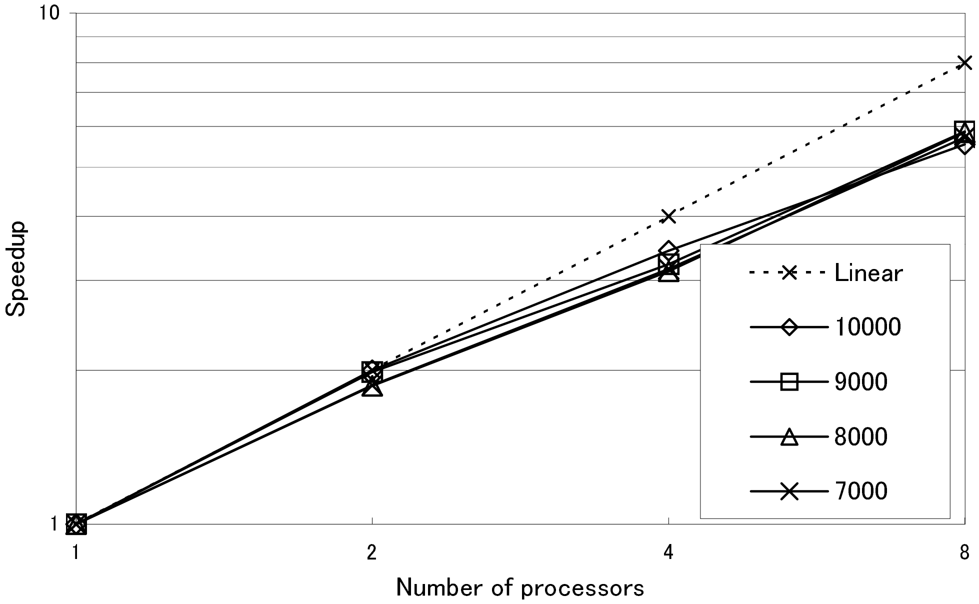

図 5.7: 焼き鈍し法の速度向上

考えられるが，ここでは，より計算量を正確に表す指標として，単位時間あたりの暫定解近傍探索回数を求め，全温度帯についてその総和をとって，評価することにした．1 回の暫定解の近傍探索操作がほぼ同じ計算量ならば，この指標はライブラリ全体の単位時間あたり計算量を正しく示すと考えられる．結果を図 5.8に示す．

解の改善課程と同様，8PE で 6 倍程度の計算が行われており，高い台数効果が得られている．リニアな速度向上に達しなかった残りの部分は，並列動作によって生じる通信オーバーヘッドや負荷のアンバランスといった部分による速度低下であると考えられる．特に，3PE 及び 5～7PE での実行時に単位時間あたり計算量が落ちているのは，32 温度帯を PE 数でうまく割りきれなかったため，負荷にアンバランスが生じているためである．

結論として，PopKern の並列焼き鈍し法ライブラリは，十分な性能を出したと言える．

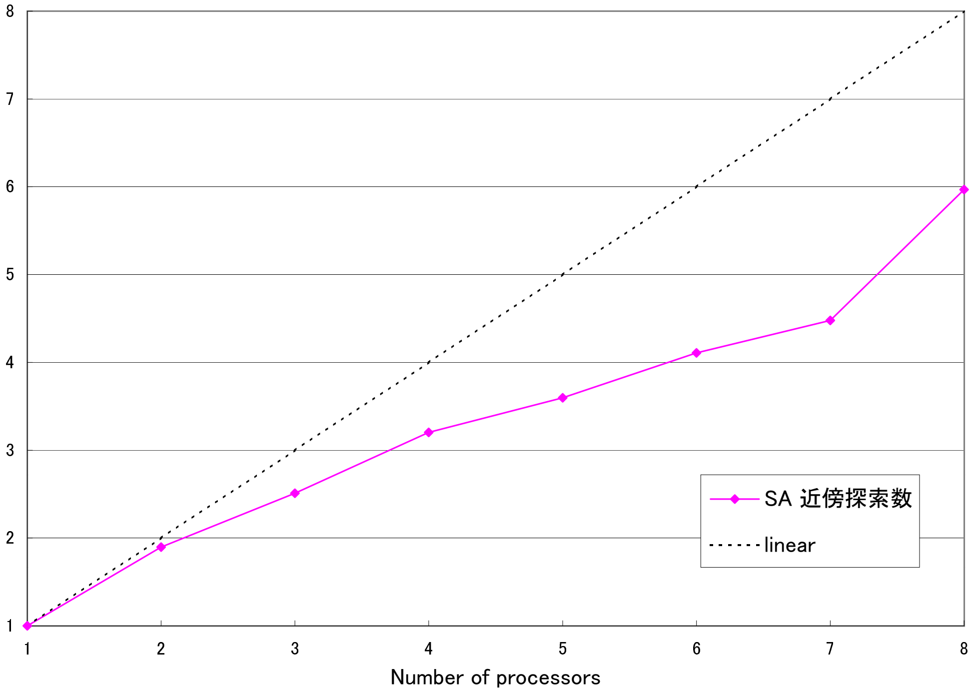

図 5.8: 焼き鈍し法における単位時間あたりの近傍探索回数

5.2.4 遺伝的アルゴリズムの並列効果

評価プログラム

評価プログラムでは，巡回セールスマン問題を遺伝子表現するための手法として，Grefenstette らによる方法 [7] を用いた．以下に簡単にその手法を説明する．

1 から Nまでの都市をある適当な順序で並べたリスト Wを用意しておく．今，ある巡回経路が $(c_1 \ldots c_N)$ という順序で都市を訪問するとき，i 番目に訪問する都市 c_iが，未訪問都市のリスト $W - \{c_1, \ldots, c_{i-1}\}$ の何番めの都市であるかを調べ，これを l_iとする．l_iは $1 \leq l_i \leq N - i + 1$ の範囲の整数となる．このリスト $L = (l_1 \ldots l_N)$ を遺伝子として，遺伝的アルゴリズムを実行する．

この方法を用いると，遺伝子は常に実現可能な正当な巡回経路と 1 対 1 に対応し，2 つの遺伝子をある長さの点で切り取り入れ替えるという単純な交叉が行える．また，突然変異についても，遺伝子中のある要素 l_iを $l'_i \in \{1, \ldots, N - i + 1\}$ で置き換えることで実現できる．

今回の評価プログラムでは，この方法を用い，1 点交叉による交叉演算と 1 要素の

突然変異を行った．また，隣の PE へと移住する遺伝子は，遺伝子集合からランダムに 16 個を選び，それらが無条件に隣の PE に受け入れられるものとした．

ここで，遺伝子集合のサイズの異なる 2 つの方式を評価することにした．「個別プールサイズ一定」方式と「プールサイズ合計一定」方式の 2 種類である．

「個別プールサイズ一定」方式は，1PE あたりの遺伝子集合のサイズを一定とする方式である．つまり，PE 数が増えるとそれに比例して遺伝子集合のサイズが大きくなることになる．これは，一定のメモリを持つ複数の PC をネットワークでつないだ，PC クラスタ上での使用を想定した評価に相当する．ここでは 1PE あたり 64 の遺伝子を扱うことにした．また，1 回の世代交代で交叉を行う遺伝子数，及び突然変異が起きる遺伝子数はそれぞれの PE で一定とした．

「プールサイズ合計一定」方式は，全 PE の遺伝子集合の合計サイズを一定にする方式である．PE 数が増えると，遺伝子集合は PE 数で均等に分割される．これは，共有メモリを持つ並列計算機上での使用を想定した評価に相当する．ここでは遺伝子集合の合計サイズを 512 とした．従って 8PE で並列動作すると 1PE あたり 64 の遺伝子を扱うことになる．また，1 回の世代交代で交叉を行う遺伝子数，及び突然変異が起きる遺伝子数の全 PE での合計も一定とする．つまり，1PE では 8PE での動作と比較して，8 倍の遺伝子集合中，8 倍の遺伝子を交叉演算し，8 倍の遺伝子を突然変異させる．

なお，巡回セールスマン問題に対応した遺伝子表現や，交配する際の演算の工夫は広く研究されている [19] が，ここでは比較的単純な手法を用いて評価を行っている．結果に見られる解の改善速度が遅いのはこのためであることに留意されたい．

個別プールサイズ一定方式による評価

個別プールサイズ一定方式について，遺伝的アルゴリズムの探索での解の改善課程をプロットしたものを図 5.9に示す．評価問題は焼き鈍し法の時と同じ，TSPLIB の a280(都市数 280) である．

図 5.9を見ると，PE 数を増やすほどよりコストの低い解が早い段階で見つかっていることが読み取れる．今回の評価プログラムでは，PE 数に比例して遺伝子集合の大きさが大きくなっているため，この高速化は「遺伝子集合の拡大により探索経路の多様性が高まり，より広範囲な探索が行えるようになった」効果と，「交配，突然変異操作を並列に行うことによる高速化」の両面の影響から達成されたと考えられる．並列動作させることにより生じる通信のオーバーヘッド等は，この評価プログラムではあまり支配的でなかったと言える．

焼き鈍し法の評価と同様に，解の評価値が 22000 を切った時点，20000 を切った時

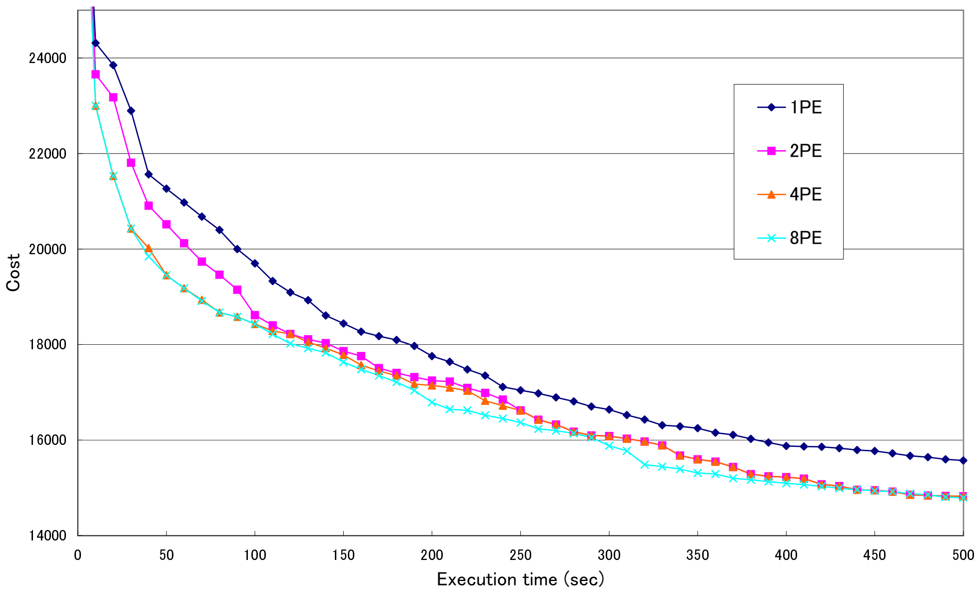

図 5.9: 遺伝的アルゴリズム (個別プールサイズ一定方式) の解候補改善過程

点... を図 5.9から読み取り，その時間の比をプロットしたものを図 5.10に示す．8PE の実行時で，最大 2.3 倍程度の速度向上が見られる．

この評価プログラムの場合，遺伝子集合の大きさが変化しているため，総計算量も使用する PE 数によって変化しており，理論上どのような速度向上が解改善過程に現れるはずであるか，は一概に言えない．そこで，単位時間あたりの計算量を正しく見積もるための指標として，全ての PE について単位時間あたりの世代交代回数の総和を取ることにした．1PE あたりの遺伝子集合の大きさが一定であるので，1PE の世代交代にかかる計算コストがほぼ等しいと考えられ，この指標で単位時間あたりに実行された計算量を見積もることができる．

この指標をプロットしたものを図 5.11に示す．ほぼ理想的な並列効果が現れている．このことから，並列動作時のシステムのオーバーヘッド等は十分小さく，PopKern は十分な性能を示したと言える．

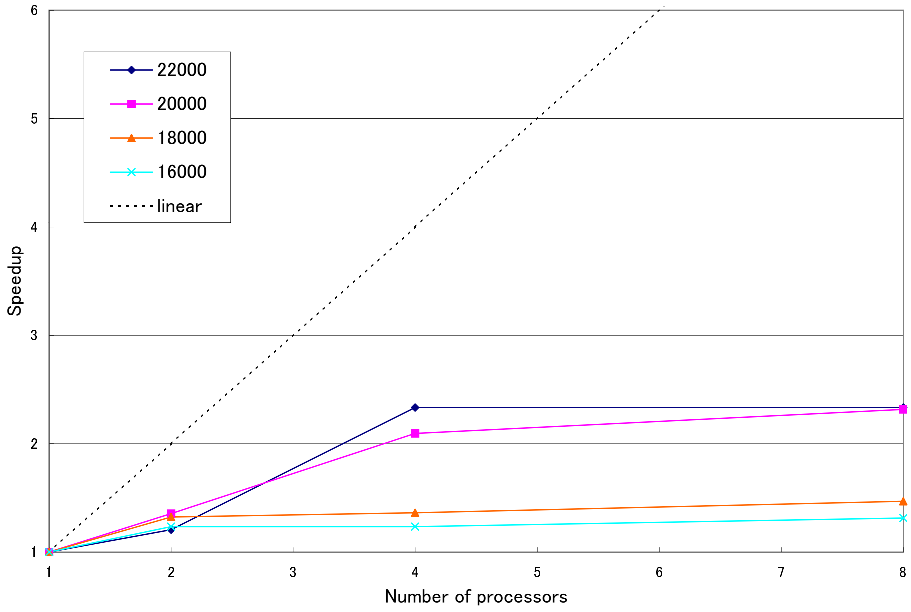

図 5.10: 遺伝的アルゴリズム (個別プールサイズ一定方式) の速度向上

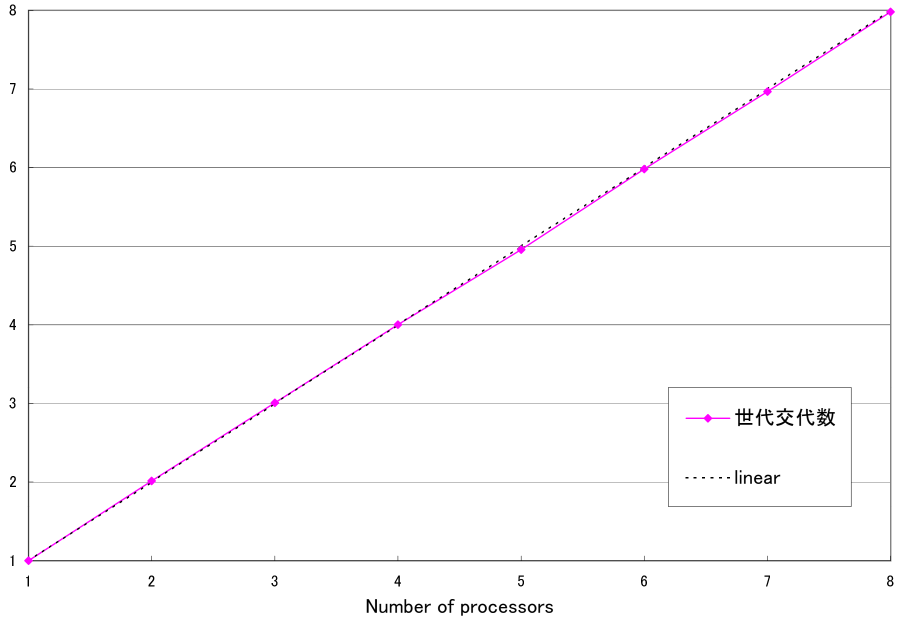

図 5.11: 個別プールサイズ一定方式 GA の単位時間あたり世代交代数

プールサイズ合計一定方式による評価

プールサイズ合計一定方式について，遺伝的アルゴリズムの探索での解の改善課程をプロットしたものを図5.12に示す．評価問題は焼き鈍し法の時と同じ，TSPLIBのa280(都市数280)である．やはりPE数を増やすほど解改善がすばやく行われるようになることが読み取れる．

個別プールサイズ一定方式での評価と同様に，あるコストの解が発見されるまでの時間を図5.12から読み，その速度向上をプロットしたものを図5.13に示す．非常に高い速度向上効果がみられる．

この評価プログラムの場合，遺伝子集合の大きさや交叉演算の対象となった遺伝子数，突然変異が起きた遺伝子数は使用するPE数が変化しても一定である．従って，PE数を変化させても総計算量はほぼ変化しない．しかし，交叉演算の対象となる遺伝子の選び方が変化したり，島モデルを取ることによる遺伝子の移動の仕方が変化するため，速度向上が解改善課程にどのように現れるはずであるかはやはり一概に言えない．そこで，個別プールサイズ一定方式と同様に，単位時間あたりの世代交代回数を計算量の指標として測定した．この方式の場合，遺伝子集合の全サイズが一定であるので，全てのPEの世代交代回数の平均を取れば，単位時間あたりに実行された計算量を正しく見積もることができる．

この指標をプロットしたものを図5.14に示す．やはりほぼ理想的な並列効果が見られている．なお，PE数3，及びPE数5~7の場合にリニアな速度向上とずれを生じているのは，交叉を起こす遺伝子数や突然変異を起こす遺伝子数がPE数でうまく割りきれず，負荷のアンバランスが生じるからである．

この通り，プールサイズ合計一定方式の評価でも，本システムは十分な性能を示した．

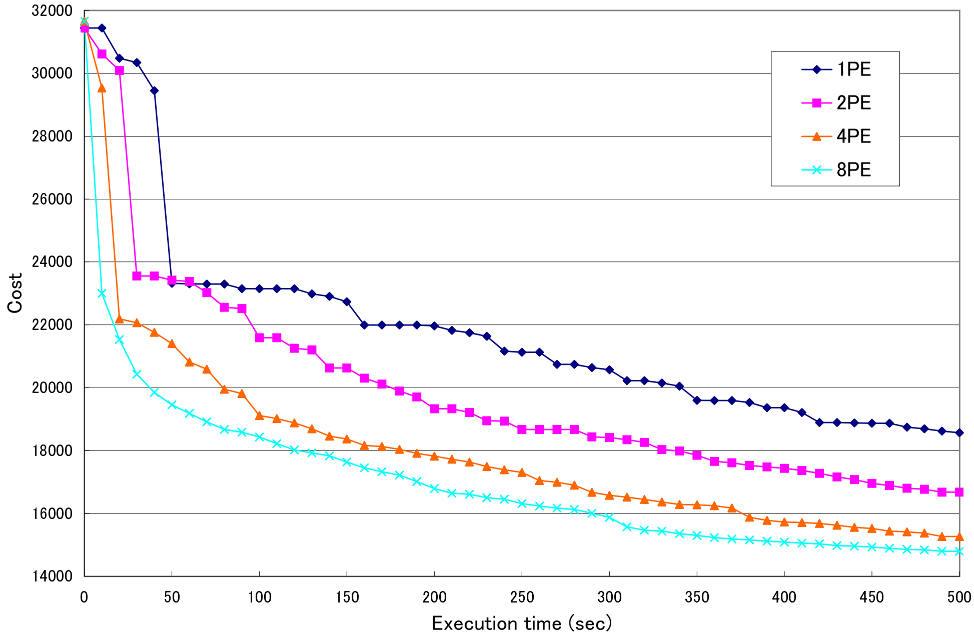

図 5.12: 遺伝的アルゴリズム (プールサイズ合計一定方式) の解候補改善過程

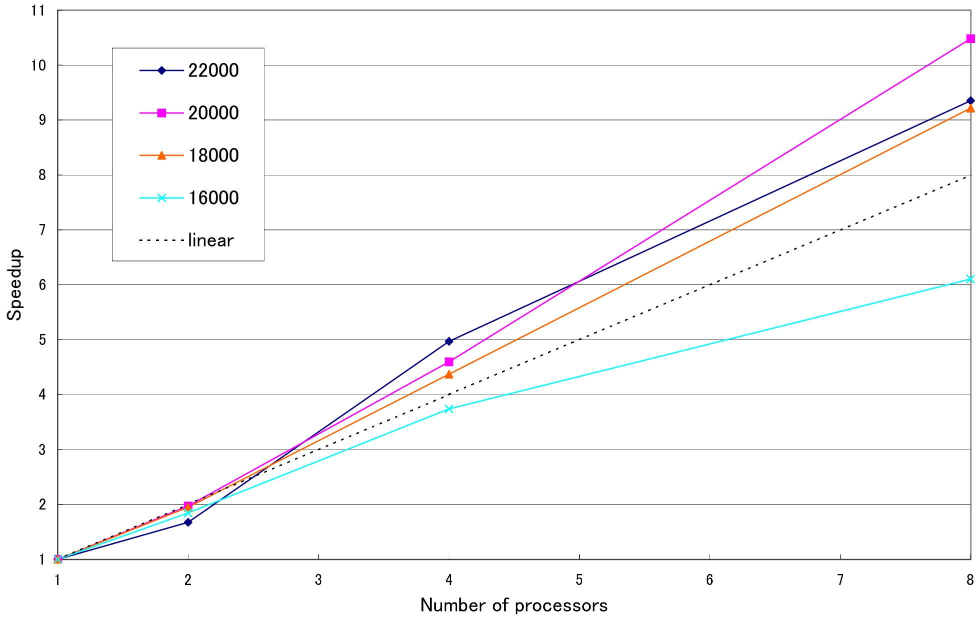

図 5.13: 遺伝的アルゴリズム (プールサイズ合計一定方式) の速度向上

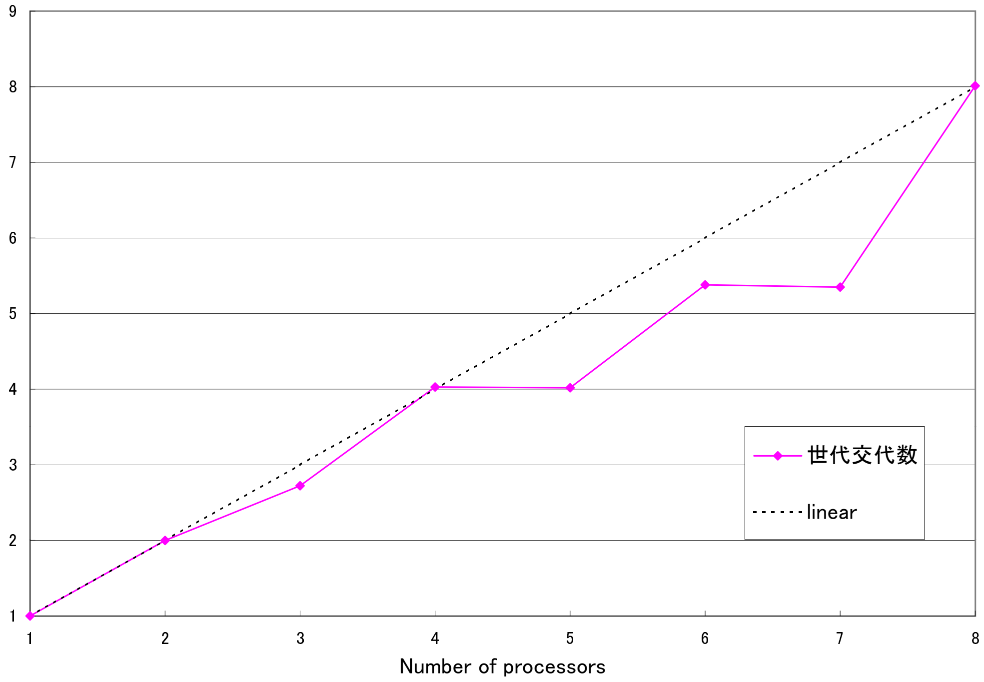

図 5.14: プールサイズ合計一定方式 GA の単位時間あたり世代交代数

5.3　2 次割当問題による分枝限定法の評価

本パッケージの想定する，問題領域依存のチューニングを十分に行ったプログラムの並列化による評価として，品野，藤江による 2 次割当問題求解プログラム [20] を用いることとした．

2 次割当問題とは，2 つの $n \times n$ 行列 $\boldsymbol{F} = (F_{ij}), \boldsymbol{D} = (D_{ij})$ が与えられた時，

$$\min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \sum_{j=1}^{n} F_{\pi(i)\pi(j)} D_{ij}$$

を求めるものである．ここで，$\mathcal{S}_n$ は $\{1, ...n\}$ 上の置換集合である．

評価プログラム

今回，分枝限定法によって厳密解を求めるプログラムを PopKern 上に移植した．解法の概略は，以下の通り．

- 初期解

  Taillard[16] によって提案された robust taboo search を用いた．

- 下界値の計算

  Gilmore-Lawler bound[6, 12] を用いた．すなわち，$i, j = 1, \ldots, n$ に対し，L_{ij}を

  $$L_{ij} \equiv \min_{\pi \in \mathcal{S}_n, \pi(j)=i} \sum_{k=1}^{n} F_{i\pi(k)} D_{jk}$$

  と計算して求め，行列 $\boldsymbol{L} = (L_{ij})$ に関する線形割当て問題の最適値を求める．この時，L_{ij}の計算において割当て問題を解く必要はなく，Gilmore-Lawler bound の計算は高速に行えることが知られている．

- 分枝方法

  固定されていない添字 i を選び，「$\pi(i) = 1, \pi(i) = 2, \ldots, \pi(i) = n$」または「$\pi(1) = i, \pi(2) = i, \ldots, \pi(n) = i$」と n 個の子問題を生成する．この時，行列 $\boldsymbol{L}$に関する割当て問題の最適解の情報や，行列 $\boldsymbol{F}$，$\boldsymbol{D}$の対称性等の情報を利用して無駄な分枝を省くことができる．

- 子問題の探索

  深さ優先探索を適用する．

3500 行程度の元プログラムの移植作業は約 7 人日を要した．

表 5.3: 2次割当問題，分枝限定法の探索時間 (msec)

PE数	nug15	nug16a	nug16b
1	228764	924185	179199
2	117257	460035	92800
3	77763	308686	62408
4	59737	233242	47763
5	48213	188345	38951
6	40552	157178	32949
7	35383	135253	28590
8	30991	118942	25418
9	28134	106033	23316
10	26070	96880	21352

評価

実行時間の測定は，QAPLIB[4] のベンチマーク問題を用いて行った．評価環境は巡回セールスマン問題の評価時と同一である．10CPUを用いたとき，nug15 (n=15) を最速11秒 (対称性による枝刈りを行った場合) で，nug16a (n=16) を最速96秒 (対称性を考慮しない場合) で解くことができた．探索時間を表5.3に示す．この探索時間は，全て対称性による枝刈りを行わない場合の結果である．また，1PEでの実行時間との比を図5.15に示す．ほぼ理想的な速度向上を示している．

2次割当問題は広く研究されており，ヒューリスティクスに基づく初期解の質は非常に良い．この評価プログラムに採用された robust taboo search が導出する初期解も非常に良質で，今回評価に用いた3つの問題では全て初期解がそのまま最良解であった．このため，分枝限定法の探索途中では，枝刈りができる枝は完全に枝刈りされ，無駄な探索がまったく発生しない．これは，どのように探索ノードの分割を行っても同じである．従って，この評価プログラムでは並列動作による無駄な探索が起きないので，並列化に伴うオーバーヘッドがなく，理想的な負荷分散が行われれば，PE数に比例した速度向上が得られるはずである．よって，速度向上が理論値に達していない部分は，システムの通信オーバーヘッドや負荷のアンバランスに起因する部分のみである．図5.15に見られるように，速度向上は理論値に非常に近く，オーバーヘッドは十分小さいと言える．

また，3つの問題規模の大きさは nug16a > nug15 > nug16b という順になっている

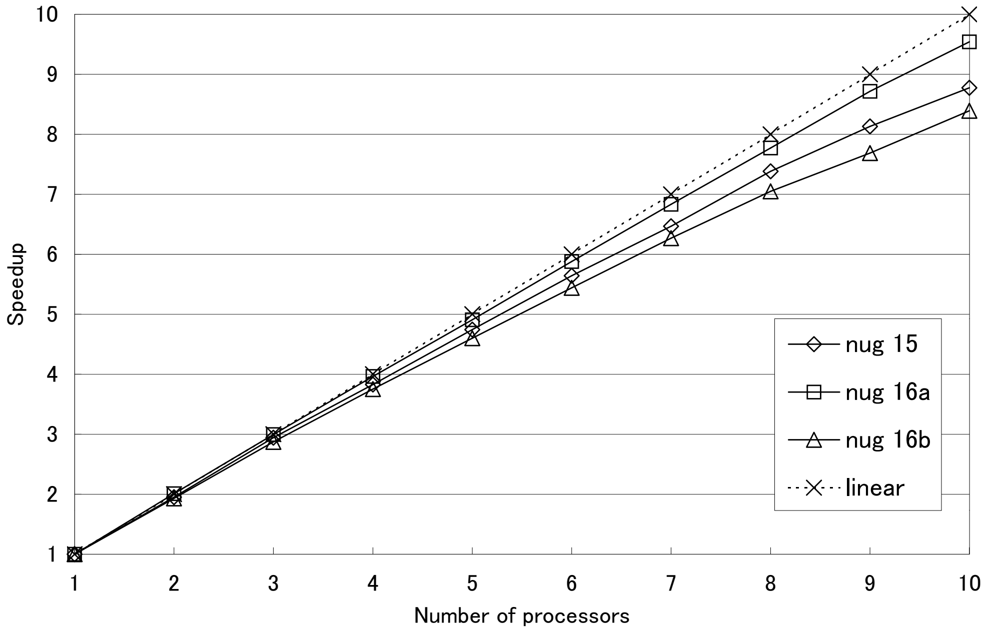

図 5.15: 2 次割当問題の速度向上

が，問題規模が大きいほど速度向上も高いものになっている．これは，負荷のアンバランスがより生じにくいためと考えられる．よって，十分大きな問題を探索対象にする際には，このシステムはほぼ理想的な速度向上をもたらせると期待できる．

5.4 評価のまとめ

巡回セールスマン問題による各アルゴリズムの評価が示す通り，PopKern の逐次オーバーヘッドはごく小さい．しかもその並列性能は高く，全てのアルゴリズムにおいてほぼ理想的な並列処理が行われた．このことから，PopKern の基本性能は高く，利用者が並列化ライブラリとして使用する時に適切なプログラミングを行えば十分な性能が得られることが確認された．また，問題領域依存知識を十分に活用したプログラムである 2 次割り当て問題による評価において，並列処理による速度向上もほぼ理想的であり，大規模な問題に対しても問題領域依存知識を活かして PopKern が適用できることが示された．

今回の実装では並列処理系として KLIC を使っており，そのためある程度のオーバーヘッドが予想されていたが，実際にはごくわずかなオーバーヘッドで収まっている．こ

れは，実行時間のほとんどの部分が利用者が定義したプログラム部分であることによる．また，並列化アルゴリズムのフレームワークの部分だけを提供するという今回の設計でも，並列実行による性能向上は十分に得られている．よってこれらの評価より，今回提案したライブラリの設計，すなわち，問題領域依存の利用者定義データ構造を問題領域独立な並列化アルゴリズムのフレームワークにプラグインする，というライブラリ構造は，問題領域依存知識の活用を重視しながらかつ高い並列実行効率を得られる枠組みとして適したものである，と結論することができる．

また，2次割り当て問題を解くプログラムの移植作業について，プログラムの大きさに対してPopKernへの移植にかかった作業量はそれほど大きくなかった．これにより，今回提案したライブラリの設計は，問題領域依存部分の記述の自由度を下げることなく，しかも記述しやすい枠組みを持っていると言え，基本設計の妥当さが確かめられた．

第 6 章

おわりに

6.1　まとめ

本論文では，最初に，既存の並列組合せ最適化ライブラリが問題領域依存の知識を十分に生かせるような能力を持っていないことを論じた．この問題に対し，探索のために必要となるデータ構造やそれを扱うサブルーチン群を利用者定義とし，代表的な組合せ最適化アルゴリズムの並列実行をコントロールするカーネル部に利用者定義部分をプラグインする，という設計を行うことで，高いチューニング能力を持つ並列化ライブラリが実現できると考えた．そこで，このような設計による並列ライブラリ・パッケージである PopKern を構築し，その設計と実装の効率について評価を行った．その結果，PopKern は高い並列計算性能を示し，問題領域依存知識を使ったチューニングが十分なされたプログラムでも対応できることが示された．結論として，提案したライブラリの設計は，「領域依存知識の活用に重点をおいた並列組合せ最適化ライブラリ」として，利用しやすく高い実装効率が得られるものであった．

6.2　展望

PopKern は広い問題領域への応用が可能な柔軟な構造を持つ．今後，より広大な探索空間，より複雑な制約条件，そしてより効率的な問題領域依存のチューニング技法を持つような工業的実問題への応用をはかり，システムの本格的評価を行いたいと考えている．

PopKern の発展として，2 つの方向性がある．1 つは利用者の使い勝手を向上させる方向である．解の探索状況を表示し，対話的に探索をコントロールできるような GUI の

装備等を考えている．もう1つは適用問題領域の拡大である．例えば，ゲームプログラミングで必要とされるミニマックス木の探索アルゴリズムを提供することを考えている．

謝辞

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

高瀬亮氏には計算機ハードウェアに関して広範なご助言をいただきました．大内拓実氏，鶴岡慶雅氏には組合せ最適化問題，特にミニマックス探索について貴重なご意見をいただきました．吉川隆英氏には並列計算の実装に関して貴重なご助言をいただきました．ここに深く感謝致します．

三浦直人氏，Feurtey Franck 氏とは，同輩として刺激しあい，支えあって研究生活を進めてきました．研究生活に貴重な活力を与えてくださった両氏に感謝致します．

最後に，研究に関する活発な議論の場と心地好い環境を提供していただいた近山研究室の皆様に感謝致します．ありがとうございました．

なお，本研究の一部は情報処理振興事業協会の先端的情報化推進基盤整備事業「組合せ最適化問題のための汎用並列処理パッケージの技術開発」プロジェクトにおいて実施されました．

参考文献

[1] Argonne National Laboratory. PGAPack parallel genetic algorithm library, 1998. http://www-fp.mcs.anl.gov/ccst/research/reports_pre1998/comp_bio/stalk/pgapack.html.

[2] Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jurg Nievergelt. The parallel search bench ZRAM and its applications. In *Annals of Operations Research: 90*, pp. 45–63. Baltzer Science Publisher, 1999.

[3] Marian Bubak, Waldemar Ciesla, and Krzysztof Sowa. Object-oriented library of parallel genetic algorithms and its implementation on workstations and hp/convex exemplar. In *Proc. Int. Conf. High Performance Computing and Networking*, Vol. 1225 of *LNCS*, pp. 514–523. Springer-Verlag, 1997.

[4] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. QAPLIB — a quadratic assignment problem library. *Journal of Global Optimization*, Vol. 10, pp. 391–403, 1997. http://www.imm.dtu.dk/˜sk/qaplib/.

[5] Takashi Chikayama. KLIC: A portable parallel implementation of a concurrent logic programming language. In *Parallel Symbolic Languages and Systems*, Vol. 1068 of *LNCS*, pp. 286–294. Springer-Verlag, 1995.

[6] P. C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the Society for Industrial and Applied Mathematics*, Vol. 10, No. 2, pp. 305–313, June 1962.

[7] John J. Grefenstette, Rajeev Gopal, Brian J. Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and*

their Applications, pp. 160–168, Pittsburgh, PA, July 1985. Lawrence Erlbaum Associates.

[8] Feng-hsiung Hsu, Murray S. Campbell, and A. Joseph Hoane, Jr. Deep Blue system overview. In ACM, editor, *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, Conference Proceedings of the International Conference on Supercomputing 1995; 9th, pp. 240–244, New York, NY 10036, USA, 1995. ACM Press.

[9] IBM Corporation. Deep Blue official web page. http://www.research.ibm.com/deepblue/.

[10] Kouichi Kimura and Kazuo Taki. Time–homogeneous parallel annealing algorithm. Technical Report 673, ICOT, 1991.

[11] KLIC Association. KLIC official web page. http://www.klic.org/.

[12] E. L. Lawler. The quadratic assignment problem. *Management Science*, Vol. 9, pp. 586–599, 1963.

[13] Panos M. Pardalos, Leonidas S. Pitsoulis, T. Mavridou, and Mauricio G. C. Resende. Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and GRASP. In *Parallel Algorithms for Irregularly Structured Problems*, Vol. 980 of *LNCS*, pp. 317–331. Springer-Verlag, 1995.

[14] Gerhard Reinelt. TSPLIB — A traveling salesman problem library. *ORSA Journal on Computing*, Vol. 3, No. 4, pp. 376–384, 1991. http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/.

[15] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*, Vol. 840 of *LNCS*. Springer-Verlag, 1994.

[16] Éric D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, Vol. 17, No. 4–5, pp. 443–455, July 1991.

[17] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, December 1990.

[18] 石塚満. 知識の表現と高速推論. 丸善株式会社, 1996.

[19] 三宮 信夫 他(システム制御情報学会編). 遺伝的アルゴリズムと最適化. 朝倉書店, 1998.

[20] 品野勇治, 藤江哲也. PUBBによるPCクラスタ環境における並列分枝限定法. 京都大学数理解析研 講究録 特定領域研究(B)新しいパラダイムとしてのアルゴリズム工学 :計算困難問題への挑戦 『アルゴリズム工学』平成11年度全体会議　講演集.

付録 A

巡回セールスマン問題

巡回セールスマン問題 (Traveling Salesman Problem, TSP) とは，組合せ最適化問題の代表的なものであり，

> 複数の都市が配置された世界があり，セールスマンはその都市全てを 1 度づつ訪問しなければならない．全ての都市を回って再び最初の都市へと帰る経路を考えた際，その経路の長さがもっとも短くなるような都市訪問順を求めよ．

という問題である．

巡回セールスマン問題には，都市間の距離の定義によって様々なバリエーションがあるが，代表的なのは対称なユークリッド距離を用いたものである．都市間の距離が対称である，というのは，都市 a から都市 b へと移動するときの距離と，b から a への距離とが等しい，という条件である．巡回セールスマン問題の場合，対称な問題より非対称な問題の方が難しい場合が多い．

また，ユークリッド距離以外を用いる問題もある．ユークリッド距離とは幾何学的性質が異なってくるため，同じ都市配置でも問題の性質は全く違ったものになる．

巡回セールスマン問題は単純でわかりやすい問題であり，かつ応用範囲が広い．少し制限を加えれば，実社会の回路配置問題や物流に関する問題に応用できる．このことから，巡回セールスマン問題の高速解法は古くから研究されてきた．

TSPLIB[14] は，良く知られた巡回セールスマン問題とその既知の最良解を集めたライブラリである．ここに納められた問題に関しては，都市数が数千程度の問題までは厳密最適解が求められており，それ以上になると準最適解が知られているだけである．

付録 B

PopKern の使用法

B.1 インストール

PopKern のインストールには，KLIC が必要である．現時点での KLIC 最新版 (3.3 版) は共有メモリ，PVM，MPI の 3 つの通信環境をサポートしており，PVM や MPI を使う場合は前もってインストールしておく必要がある．KLIC のダウンロード，インストール方法等は KLIC 公式ウェブページ [11] を参照されたい．

PopKern のインストールは，インストールするディレクトリ等を Makefile に指定して，make コマンドで行う．

B.2 プログラミングとコンパイル

利用者はまず，利用者定義のデータ型を定義したファイルを作る．ファイル名は `popkern_hc_user.h` である．(以降の説明に現れるファイル名等は山登り法の場合である．山登り法の時は `hc`，分枝限定法は `bb`，焼き鈍し法は `sa`，遺伝的アルゴリズムは `ga` という識別子が入る．)

次に，利用者が定義すべき関数群を定義する．ファイル名は自由である．ここでは `foo.c` とする．`foo.c` の先頭で，PopKern が提供する関数群，及び利用者が定義すべき関数群を宣言した `popkern_hc.h` をインクルードする．ちなみに，利用者が定義すべき関数群の宣言は，PopKern がインストールされたディレクトリ内の `popkern_hc_user.h` にある．

プログラミング後，

```
% /(popkern_installed_dir)/popkern hc (cflags) (srcs)
```

でコンパイルが行われる．

例えば，PopKern をインストールしたディレクトリにパスが通っており，利用者定義ファイルが `foo.c` のみ，libX11 を使用，最適化オプションに`-O2` を使うとき，

```
% popkern hc -O2 -lX11 foo.c
```

とすると実行バイナリ `a.out` ができる．

B.3 実行

実行バイナリ `a.out` の実行の際，与えるコマンドライン引数が 2 種類にわかれる．

```
% ./a.out (KLIC_args) -- (PopKern_args)
```

のように，`--` の前に KLIC のコマンドライン引数を，後ろに PopKern の引数をつける．

KLIC のコマンドラインスイッチは様々であるが，基本的には，

`-p <N>`: 実行プロセス数 (`N` プロセス)

を使うだけで十分であろう．その他 KLIC システム自身の詳細は，KLIC パッケージ内のマニュアル並びに KLIC 公式ウェブページを参照されたい．

PopKern のコマンドラインスイッチは，以下の通り．

`-i <filename>`: 問題入力ファイル名

`-o <filename>`: 解出力ファイル名

`-u <string>` : ユーザ定義引数 (ユーザ定義関数に渡される)

`-l <time>` : 制限時間 (分枝限定法では無意味)

`-t <N>` : 焼き鈍し法の温度数 (他のアルゴリズムでは無意味)

例えば，プロセス数 8，制限時間 30 秒で解を探索するときには，

```
% ./a.out -p 8 -- -i problem.txt -o solution.txt -l 30
```

とする．