論文

分散処理環境における数値シミュレーションの静的負荷分散手法

Static load-balancing for distributed processing of numerical simulations

山下 真史
(豊橋技術科学大学 大学院 工学研究科 知識情報工学専攻)

連絡先
山下 真史
〒 441–8580 豊橋市天伯町字雲雀ヶ丘
豊橋技術科学大学
TEL: (0532)46-0111
email: shinji@ich.tutkie.tut.ac.jp

概要

処理能力の異なる複数のプロセッサ (PE) からなる分散処理環境上で，通信時間と計算時間の双方を考慮して並列数値シミュレーションを静的に負荷分散する手法について述べる．本手法では組合せ最適化問題を解いて実行時間を最適化するため，計算負荷の過剰な分散によって実行時間が増加することを回避し，最適な PE を最適な数だけ自動的に選択して用いることができる．この問題を，(1) 各計算ブロックへの PE の分配と，(2)PE の処理能力に合わせた計算ブロックの分割，という 2 段階に分けて解決する．いずれの問題も計算が極めて困難なので，分枝限定法や近似アルゴリズムを利用して現実的な時間内で解を求める．シミュレーションによれば，ブロック数 8，プロセッサ数 32 という条件下で提案する近似アルゴリズムの誤差は最適解から約 9%であった．近似解の求解時間は現状の計算機でも数ミリ秒と充分実用的である．

Abstract

This paper describes a static load-balancing scheme for parallel numerical simulations on distributed computing environment, which usually has a variety of processing elements (PEs). Our scheme allocates the most adequate combination of PEs to minimize execution time by using combinatorial optimization technique, avoiding excessive use of remote processors which results in aggravation of execution time. This problem is solved by the following two steps: (1) PEs are distributed among computing blocks, (2) then each computing block is split for each PE to minimize processing time of the whole simulation, considering both of computation and communication. As this problem is a kind of combinatorial optimization which is very hard to solve, this paper also shows some algorithms which gives good approximation in reasonable time.

1. はじめに

近年，コモディティ製品を用いた分散処理がコスト性能比に優れた計算環境として注目されている．本研究では数値シミュレーション (特に偏微分方程式の数値的求解) を分散処理環境で最適に行うための方法について検討する．

一般に分散処理環境では並列計算機に比べ通信時間が大きいため，高い台数効果を得ることが難しい．多数の要素プロセッサ (PE) を使用すれば計算負荷を減らすことはできるが，PE 間通信の発生により全体の性能が制限される．さらに必要以上に多数の PE に計算負荷を分散すれば，逆に実行時間は最適な実行時間より悪くなる☆．実行時間を最小化するには，通信時間を考慮にいれて最適な負荷分散を行う必要がある．

計算時間と通信時間を考えて実行時間を最小化する問題は組合せ最適化問題としてモデル化できるが，一般に計算困難であることが知られている．さらに，分散処理環境は一般に不均一な PE で構成されているため，並列計算機 (PE が均一) の場合と比べて自由変数の数が著しく増大し，最適化が一層困難になる．本研究では分散計算環境において通信時間と計算時間を考慮して最適な静的負荷分散を行う方法を検討する．問題を組合せ最適化問題としてモデル化し，最適解を求める方法を示すとともに，精度が良く求解時間の短い近似アルゴリズムを示す．

2. 計算モデル

市川ら[1)]は，偏微分方程式 (PDE) の並列求解システム NSL を例にとって通信時間と計算時間の双方を考慮する静的負荷分散法を示した．本研究でも NSL のモデルを使って負荷分散の検討を行う．論文 1) では並列計算機の利用を前提としているため，PE の構成・性能が全て等しいと仮定しているが，本研究では PE の性能が不均一な分散処理環境について論ずる．本研究で扱う計算モデルは論文 1) の計算手順 1 を，分散環境に適応するよう拡張したものである．本章のモデルと論文 1) のモデルの最も大きな相違は，並列計算機の PE は区別の必要がないのに対し，分散環境では PE の能力が不均一なので全てのプロセッサを区別する必要があるという点である．

☆ 以後本論文ではこのような状況を “過剰な負荷分散” と呼ぶ.

2.1 NSL

NSL[2]ではPDEを差分式に変換して陽解法で解く．計算対象となる物理領域は，境界適合法とマルチブロック法によって，相互に接続された複数の矩形の計算領域 (ブロック) に写像される (**図 1**).

各ブロックは 2 次元に配列された格子点からなり，各格子点上では PDE に対応した差分式が計算される．差分式の計算には近隣の格子点のデータが必要であるが，NSL で採用している陽の差分式では時刻 t における各格子点の計算に相互のデータ依存性がなく，全て並列に実行することができる．従って，各ブロックを複数の要素プロセッサ (PE) に割り当てることで差分式の計算を並列に実行できる．ただし，各時間ステップの計算後には格子点の値を交換するために PE 間通信が発生する．論文 1) は，このような並列計算モデルに関して実行時間を最小にする静的負荷分散法を論じている．

2.2 負荷分散法の概要

本研究の動機は，分散処理環境を利用して低コストで多くの PE に負荷を分散し，計算の実行時間を短縮することである．例えば，夜間利用されていない PC を必要なだけ LAN 上で確保して計算を流すといった使い方を想定している．極端な例としては，Ninf[3]や NetSolve[4]が目指すような広域分散処理環境を考えても良い．このような場合，利用可能な PE 数は相当大きな数になりえるが，一方で過剰な負荷分散による実行時間の増大を防ぐための静的負荷分散法が必須となる．

以下，ブロック数を m，PE 数を n として，計算負荷を分散する方法を考える．なお本研究では論文 1) と同じく $m \leq n$ と仮定し，各ブロックに対して 1 つ以上の PE を割り当てる．ブロックを担当する PE が複数ある場合，ブロックを PE の数に分割して，それぞれの断片 (サブブロック) を PE に割り当てることにする (**図 2**).

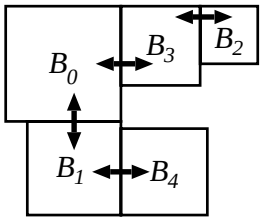

図 1　計算領域 (ブロック)

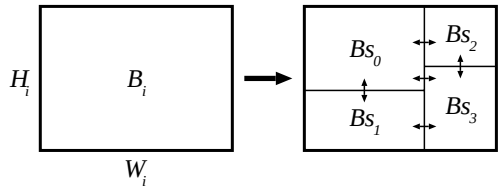

図 2　ブロックの分割

分割にあたってはサブブロックは必ず矩形にする．自由な形状に分割することを許すと，通信時間が増大したり並列処理ソフトウェアが複雑化する可能性がある．逆に矩形であればベクトル処理等による高速化も期待できる．同様な理由により，複数のブロックから同一の PE にサブブロックを割り当てることを禁じる．この制限によって負荷分散の質は若干損なわれるが，並列処理ソフトウェアの構造は単純になり，また通信時間や計算時間のモデル化が容易になるため負荷分散や最適化が精度良く行えると考えられる．このような制限のない一般的な負荷分散に関しては，本論文では範囲外とする．

本研究の目的である負荷分散問題を解くためには，まず部分問題として,「ブロックを担当する PE(1 つ以上) が決まったとき通信時間と計算時間の両方を考慮して処理時間が最小になるようにブロックを分割する方法」を求めなければならない．この部分問題については 3章で検討する．この部分問題が解決すれば，あとは「n 個の区別のあるプロセッサを m 個の区別のあるグループに分割する方法のうちから実行時間を最短にする組合せを選ぶ」という組合せ最適化問題に帰着される．この組合せ最適化問題の探索空間は非常に広く計算が困難である[5]．4章では，この最適化問題が分枝限定法[6][7]の採用により実用的な時間で解けることを示す．

また，精度が良く求解時間の短い近似アルゴリズムも示す．本研究では実際に最適化問題を解いているため，近似アルゴリズムの精度が最適解に対して絶対的・定量的に評価できる．

2.3　評価関数について

本研究の目的は，分散処理環境で数値シミュレーションの実行時間を最小化することである．したがって，本研究では実行時間 T を評価関数とする．既に述べたように，本論文の計算モデルでは各 PE がサブブロック 1 つを担当するので，対応する PE とサブブロックに同じ添字 i を付けて区別する．最も実行時間の長いサブブロックがクリティカルパスとなるため，PE 数を n とすれば

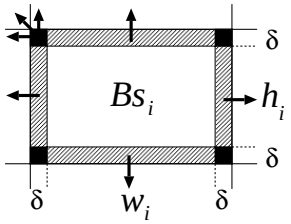

図 3　サブブロック

全体の実行時間 T は以下の式で表される．T_i は PE_i の実行時間である．

$$T = \max_i T_i \qquad (i = 0, 1, \ldots, n-1) \tag{1}$$

$$T_i = Ta_i + Tc_i \tag{2}$$

計算時間 Ta_i はサブブロック Bs_i の格子点数 Sa_i の一次関数であるとする．

$$Ta_i = Cta_i Sa_i + Dta_i \tag{3}$$

ここで Cta_i は格子点あたりの計算時間である．PE_i の処理速度が個別に違うので，Cta_i も PE 毎に異なる．Dta_i はサブブロックの計算による遅延を表す定数項である．

通信時間 Tc_i は通信量 Sc_i の一次関数と見積もれるので，

$$Tc_i = CtcSc_i + Cn_iDtc \tag{4}$$

となる．本論文では問題を単純化するために，小規模 PC クラスタなど均一なネットワークを持つ環境を仮定することとする．従って帯域幅 Ctc，通信による遅延 Dtc はすべての PE で等しいと仮定した．Cn_i はサブブロック Bs_i の PE 間通信の数で，サブブロックの配置によって異なる．例えば，図 2 の Bs_0 では同一ブロック内の通信が 3，他のブロック (上方向と左方向) への通信が 2 で $Cn_0 = 5$ となる．通信量 Sc_i は，サブブロックの縦横の格子点数をそれぞれ h_i, w_i とするとき (**図 3**)，以下の式で表される[1)]．

$$Sc_i = 2\delta\,(h_i + w_i + 2\delta) \tag{5}$$

3. ブロック内の負荷分散

本章では，単一のブロックを処理能力の異なる複数の PE に割り当てるためのブロック分割方法について検討する．このブロック分割は，図形的制約のある一種の組合せ最適化である．例えば図 2 のように分割するとき，各サブブロックの辺長は整数でなければならない (整数制約)．しかも各サブブロックをパズルのように組み合わせて元のブロックが構成できなければならない (図形的制約)．ブロック全体の実行時間を最小化するには，通信と計算を考慮して，各 PE

の実行時間の最大のものが最小となるようなブロック分割を求めれば良い．

しかしこのような最適化問題を解くのは極めて困難である．そこで本研究では，ヒューリスティックを用いた近似的分割法を検討する．近似的分割法としては以下の 5 つをとりあげるが，いずれも切断面が直線であると通信量が減るという直観に従って，計算量のバランスを取りながら再帰的に直線分割していく手法である．これは一般に RB (Recursive Bisection)[8)9)] と呼ばれる方法の一種である．これらのヒューリスティックの結果と，緩和問題を解いて求めた下界値を比較して，近似的分割法の精度を評価する．

3.1 近似的分割法

3.1.1 分割法 0 (Type0)

PE を 2 つのグループに分け，グループ全体の処理速度に比例した格子点数が分配されるよう，ブロックを直線的に二分する．このときブロックの縦方向と横方向のうち通信量の少ない方向に切る．これを 1 グループ 1PE になるまで再帰的に繰り返す．n 個の PE を 2 グループに分ける組合せは $2^n - 2$ 通りあるが，これらすべての組合せを調べる．

3.1.2 分割法 1 (Type1)

PE を $n/2$ 個ずつの 2 グループに分け，分割法 0 と同様にブロックを二分する．これを 1 グループ 1PE になるまで再帰的に繰り返す (**図 4**)．2 グループに分ける場合の組合せ ${}_nC_{\lfloor n/2 \rfloor}$ 通りあるが，すべてを調べる．分割例を**図 6** の Type1 に示す．1 回目から 4 回目までの分割線を，線の種類で図中に示した．

3.1.3 分割法 2 (Type2)

n 個の PE を $\lfloor \sqrt{n} \rfloor$ 個のグループに個数が均等になるように分ける．各グループの処理能力に応じて，$\lfloor \sqrt{n} \rfloor - 1$ 本の平行な直線でブロックを分割する．この時も縦横のうち通信量の少ない方向に切る．これ以後は，各グループに対して分割法 1 を適用する．ここでも，$\lfloor \sqrt{n} \rfloor$ 個のグループに分ける組合せをすべて調べる．分割例を図 6 の Type2 に示す．分割法 2 は同じ能力の PE が多数あるとき効果があると考えられる．

3.1.4 分割法 3 (Type3)

PE を処理能力が均等な 2 グループに分ける．このとき，**図 5** のように greedy 法を用いる．そして分割法 1 と同じようにブロックを直線分割し，1 グループ 1PE になるまで再帰的に繰り返す．2 グループに分けるとき全ての組合せを調べている分割法 1 に比べると精度は落ちるが，探索範囲を狭くすることでより

```
void Type1(int P[ ], int H, int W){
  /* H : ブロックの縦の格子点数 */
  /* W : ブロックの横の格子点数 */
  if(| P | == 1) return;

  PE を (⌊n/2⌋) 個と (n − ⌊n/2⌋) 個の 2 つのグループ Pa と Pb に分ける;
  1/Ctaa : Pa の 1/Cta の総和
  1/Ctab : Pb の 1/Cta の総和
  Ctaa = 1 / (1 + Ctaa / Ctab);

  if(H < W){
    Wa = floor (W / Ctaa);
    Wb = W - Wa;
    Ha = Hb = H;
  }
  else{
    Ha = floor (H / Ctaa);
    Hb = H - Ha;
    Wa = Wb = W;
  }

  Type1(Pa, Ha, Wa);
  Type1(Pb, Hb, Wb);
  return;
}
```

図 4　分割法 1(Type1)

短時間で解が得られると予測される．

3.1.5　分割法 4 (Type4)

分割法 3 と同じように greedy 法を用いて，PE を $\lfloor\sqrt{n}\rfloor$ 個のグループに処理能力が均等になるように分ける．そしてそれぞれのグループに対して分割法 3 を適用する．これも分割法 2 に比べ探索範囲が狭いため，短時間で解が得られることが予測される．

3.2　局所的負荷調整

近似的分割法では計算時間の均衡だけを考えて分割している．そのため PE の処理能力が大きいと，対応するサブブロックの格子点数も大きくなり，結果的に通信量も増加する．通信性能は PE の能力によらずネットワークで決まるので，計算時間が均等になるように分割すると速い PE ほど通信時間が大きくなって合計処理時間が平均より大きくなってしまう．そこで処理時間を元に近

```
void Type3(int P[ ], int H, int W){
  /* H : ブロックの縦の格子点数 */
  /* W : ブロックの横の格子点数 */
  if(| P | == 1) return;
  PE を 1/Cta で降順にソート;
  PSa = PSb = 0.0;
  for(i=0; i < | P |; i++){
    if(PSa <= PSb){
      PSa に P[i] の 1/Cta を加算;
      グループ Pa に P[i] を登録;
    }
    else{
      PSb に P[i] の 1/Cta を加算;
      グループ Pb に P[i] を登録;
    }
  }
  1/Ctaa : Pa の 1/Cta の総和
  1/Ctab : Pb の 1/Cta の総和
  Ctaa = 1 / (1 + Ctaa / Ctab);
  if(H < W){
    Wa = floor (W / Ctaa);
    Wb = W - Wa;
    Ha = Hb = H;
  }
  else{
    Ha = floor (H / Ctaa);
    Hb = H - Ha;
    Wa = Wb = W;
  }
  Type3(Pa, Ha, Wa);
  Type3(Pb, Hb, Wb);
  return;
}
```

図 5　分割法 3

似的分割を求めた後，大きなサブブロックから隣接するサブブロックに一部の格子点を移動して局所的負荷調整を行う (**図 7**).

ただし，**図 8** のようにサブブロックの境界の形によって格子点を移動できるサブブロックが限られる．例えば，図 2 の場合は，Bs_0と Bs_1，Bs_2と Bs_3の間でのみ格子点の移動が可能である．

この局所的負荷調整により，通信時間も考慮したブロック分割が行える．

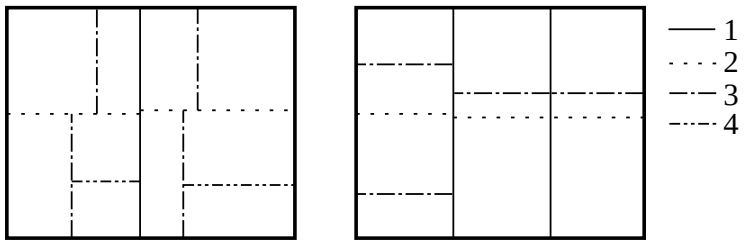

図 6　近似的分割法の分割例

do{
　各ブロックの中で実行時間が最大のサブブロック Bs_i を求める;
　Bs_i と隣接しているサブブロックの中で実行時間が最小のサブブロック Bs_j を求める;
　Bs_i と Bs_j の実行時間が均等になるように Bs_i の格子点を Bs_j に移動;
}while(移動できる格子点がある);

図 7　局所的負荷調整

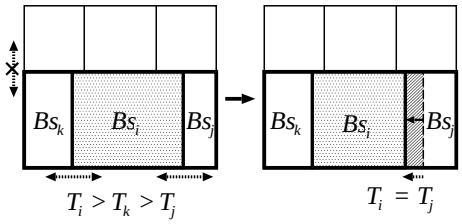

図 8　局所的負荷調整のイメージ

3.3　緩 和 問 題

近似的分割法の評価基準として，緩和問題を解いて実行時間の下界値を求める．緩和問題は以下のように定義する．

- サブブロックの辺長は整数で無くても良い．
- サブブロックの面積の和が元のブロックと等しければ良い．

一番目の条件で整数条件を外し，二番目で図形的制約を外す．この緩和問題では格子点数は減らない (計算時間は減らない) ので，通信時間を最小にすると実行時間が最小になる．従って，各サブブロックを正方形にして計算量に対する通信量を最小にし，さらに各プロセッサでの計算時間と通信時間の和が最小になるように格子点数を割り当てれば良い．この問題は非線形計画問題であるが，整数制約がなく評価関数が凸なので，簡単に解くことができる．

3.4　評 価 結 果

以上の近似的分割法の評価を行う．各精度は 3.3節の緩和問題を解いて得られ

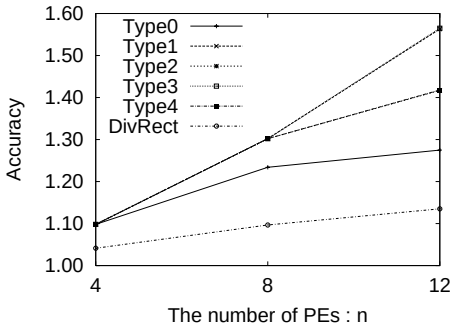

図 9　各 PE の能力が均一な場合の精度

た下界値を 1 として正規化したものを示す.

PE 数 n は 4 の倍数とし 4 から 4 ずつ増やした．その他のパラメータは**表 1** とした.

ブロック B_iの縦方向の格子点数 H_i，横方向の格子点数 W_iは，以下の条件に合わせて乱数で生成した．PE 数 n の 1 セットに対して 100 回の試行を行い，平均値をとって評価した.

$$10 \leq H_i, W_i \leq 200 \tag{6}$$

$$H_i, W_i \equiv 0 \pmod{10} \tag{7}$$

図 9 では，全ての PE で $Cta_i = 1$ とし，PE の能力が均一な場合の近似的分割法の精度を検証した．求解時間を**図 10** に示す．参考として過去の研究[1)]で並列計算機向けのブロック分割法 DivRect も同じパラメータで評価して示した．実行環境は**表 2** に示す通りである.

この場合，それぞれの分割法による精度の差はわずかである．PE 数が増えるにしたがって精度が悪くなっているのは，通信時間が全体の実行時間に占める割合が大きくなっているからである．実行時間は Type0 を除けばほとんど変化はない．これは PE が均一であることにより，PE を区別する必要がなくなり，組合せが 1 通りしかないためである．DivRect と精度を比較すると Type1,3 はほとんど変わらない．DivRect が他の分割法に比べて多少悪いのは，均等に分割することを目的に碁盤目のような切り方をするため，サブブロック間の通信数が多くからである.

図 11 では PE の能力が不均一な場合について検討した．PE の内訳を**表 3** に示す．求解時間を**図 12** に示す．Type0,1,2 は Type3,4 に比べ多くの組合せを調べているため，精度は良いが求解時間が大きい．ブロック分割は PE の割り当

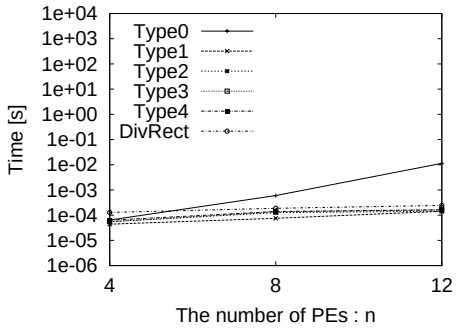

図 10　各 PE の能力が均一な場合の求解時間

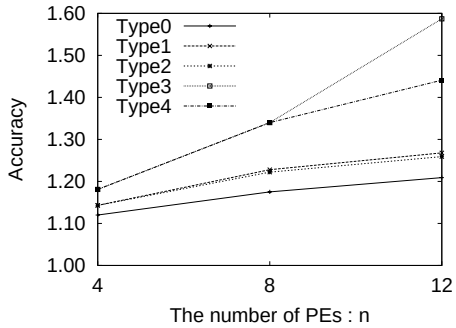

図 11　PE の能力が不均一な場合の精度

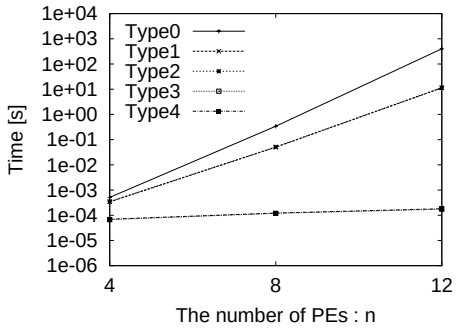

図 12　PE の能力が不均一な場合の求解時間

てごとに行うため，非常に多く呼び出される．その意味では，Type0,1,2 は求解時間が大きすぎて，実用的な手法ではない．一方，Type3,4 は求解時間が非常に短く，実用的である．

図 13 と**図 14** に局所的負荷調整を行った場合の精度を示す．PE の能力が均等な場合，局所的負荷調整を行っていない図 9 と比較しても大差はない．PE が

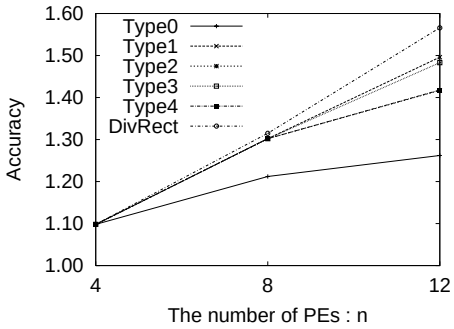

図 13　各 PE の能力が均一な場合の精度 (負荷調整あり)

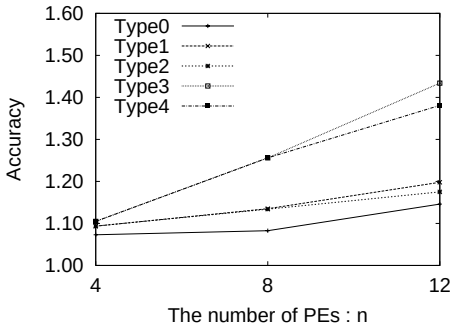

図 14　PE の能力が不均一な場合の精度 (負荷調整あり)

均一な場合は DivRect で碁盤目に切るように，それぞれのサブブロックの実行時間が均等になり，局所的負荷調整ができない場合が多いからである．

一方 PE の能力が不均一の場合は局所的負荷調整によって解の精度が大きく改善されている (図 14)．分割法 0,1,2 では 5%程度，3,4 では 10%程度改善されている．求解時間は局所的負荷調整を行っても元の数倍程度に収まっていて，分割法 3,4 は 1 ミリ秒未満と十分に短い時間である．よって，局所的負荷調整はすべての分割法で有効である．

これらの結果より，4 つのうちのどの分割法を採用するかが問題となる．前にも述べたように，Type0,1,2 は求解時間の点から実用的な手法とは言えず，採用できない．一方，求解時間が非常に短い Type3,4 の精度はともに n が大きくなるにつれ悪化していく．しかし，これは計算時間に比べ通信時間が支配的になっているためである．状況を単純に示すため，PE の能力が均一 $(Cta_i = 1(i = 0 \ldots n-1))$ である場合を検討してみる．**図 17** は，正方形 $(H = W = 100)$ のブロック B_i を Type0,Type3,Type4 の 3 つのアルゴリズムで分割したときの実行時間 T_iを

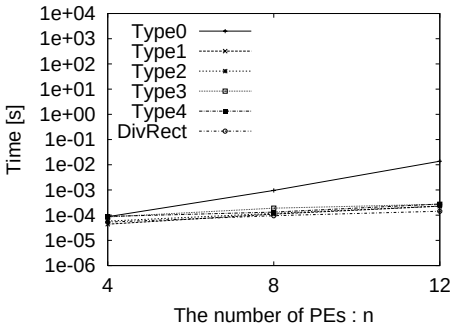

図 15　各 PE の能力が均一な場合の求解時間 (負荷調整あり)

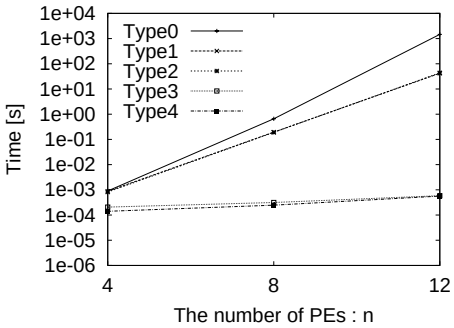

図 16　PE の能力が不均一な場合の求解時間 (負荷調整あり)

示している．図中の Lower は下界値である．Type0 は PE 数 n によらず安定した精度であるが，$n > 4$ では T_iを大きく改善することはない．n が増えても T_i が改善しないのは，既に通信時間が支配的になっているためで，PE 数を増やして計算時間を減らしても全体の実行時間にはほとんど影響しないからである (過剰な負荷分散状態)．一方，Type3 と Type4 は過剰な負荷分散状態では良い T_iを得られず精度を顕著に悪化させるが，PE 数が過剰でないうちは Type0 とほとんど差のない結果を与える．実際 $n = 5$ での精度は Type0 とほぼ同じで，そのときの誤差は下界から 20%程度に収まっている．

このように過剰な負荷分散を回避できれば，greedy な手法である Type3 や Type4 は実用的であると言える．そして過剰な負荷分散はこのあとの第 4章で述べるようなプロセッサ分配アルゴリズムを使えば回避できる．

これらの結果より，本研究では分割法 4 に局所的負荷調整を加えたものをブロック分割法として採用することにする．

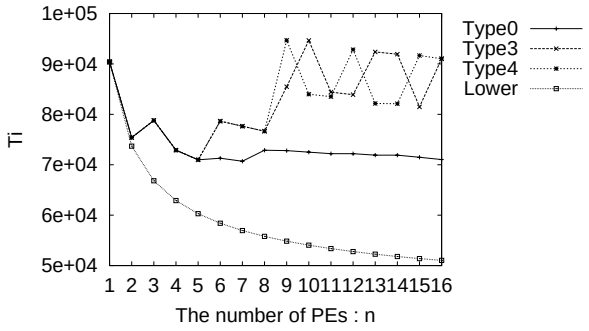

図 17　ブロック分割と実行時間 T_i

4. ブロック間の負荷分散

本研究では各ブロックへの最適な PE グループの割り当てと最適なブロック分割を行って，実行時間を最小にすることを目的としているが，第 3章で述べ

表 1　パラメータ一覧

名前	値	名前	値
Ctc	100	Dtc	10000
Dta_i	0.5	δ	1

表 2　実行環境

CPU	Intel Pentium-II 400MHz
主記憶	256MB
OS	FreeBSD 3.1R
コンパイラ	gcc 2.7.2.1

表 3　PE の内訳

Cta_i	台数
1.00	$n/4$
0.50	$n/4$
0.33	$n/4$
0.25	$n/4$

たように，最適なブロック分割を求めるのは困難である．そのため，第3章の近似的分割法Type4をブロック分割に用いて，その上で実行時間を最小にすることを考える．

4.1 分枝限定法

前にも述べたように，この問題の探索空間は膨大であるため，すべての組合せを調べることは現実的でない．そこで分枝限定法[6)7)]を用いて探索空間を制限する．

暫定解を$\bar{T}$，最適解をτ，第3章で用いた下界値をT_{lower}とすると，

$$\bar{T} \geq \tau \geq T_{lower} \tag{8}$$

が成り立つ．既に3.3節で下界が求まっているので，T_{lower}にはその下界値を用いればよい．ある組合せに対してT_{lower}を計算し，その値が暫定解$\bar{T}$より大きいときは，その組合せで最適解が得られる可能性はない．また，目的関数Tは式(1)により求められるので，あるサブブロックの実行時間T_iが暫定解より悪くなった時点で他のサブブロックの計算を打ち切る．もし$\bar{T}$よりよい実行可能解を得られれば，$\bar{T}$を更新する．こうして動的に探索範囲を狭めることができる．

4.2 プロセッサ分配の近似アルゴリズム

分枝限定法では，暫定解が悪いと探索空間が広くなり，暫定解の更新がますます難しくなる．そのため探索の初期から比較的最適解に近い暫定解を用いることが，探索時間を短縮する鍵となる．このため精度の良い近似アルゴリズムが必須となる．近似アルゴリズムでは短時間で実行可能な解を求めることが必須で，解の質がよく，精度保証があることが望ましい．

本研究ではヒューリスティックによる近似アルゴリズムを用いる．精度保証はないが，短時間で実行可能な解が求まる．この近似アルゴリズムと局所探索によって得られた実行可能解を分枝限定の暫定解の初期値とする．近似アルゴリズムの評価は第5章で行う．

4.2.1 近似アルゴリズム 1

次に各ブロックの計算量と各PEの性能の不均一性を考慮し，計算量の多いブロックから順に速いPEを割り当てていく方法を考える．初めに，ブロックを格子点数で降順に，PEをCtaで昇順にソートする．ブロックB_iの格子点が全体の格子点に占める割合をRB_i，PE_iの処理速度が全体の処理速度(総和)に占める割合をRPE_iは，

```
void Approx1(){
  各ブロックのシェア RBを計算し降順ソート;
  各 PE のシェア RPEを計算し降順ソート;
  for(i=0,j=0; j < n; j++) {
   /* Biに PEjを割り当てる */
   RBi = RBi − RPEj;
   グループ Piに PEjを登録;
   /* ブロックと PE の残りの数を比較 */
   /* Biの残りの割合を調べる */
   if ((m − i == n − j)||(RBi ≤ 0))
     /* 次のブロックへ */
     i++;
  }
  return;
}
```

図 18　近似アルゴリズム 1

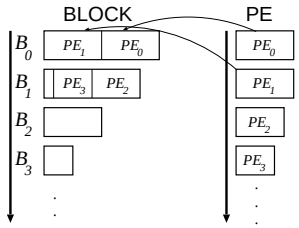

図 19　近似アルゴリズム 1 のイメージ

$$RB_i = H_i W_i / \sum_{j=0}^{m-1} H_j W_j \tag{9}$$

$$RPE_i = \frac{1}{Cta_i} \Big/ \sum_{j=0}^{n-1} \frac{1}{Cta_j} \tag{10}$$

となる．ブロックの割合 RB_iの大きい順に**図 18** の手順で PE を割り当てる．**図 19** にそのイメージを示す．図 19 の長方形の大きさが RB_i, RPE_iの大きさを表しており，RB_iを RPE_iで上から順に埋めていく．これで各ブロックに最低 1 個の PE を割り当てる．これを近似アルゴリズム 1(Approx1) とする．

4.2.2　近似アルゴリズム 2

近似アルゴリズム 1 に多少の変更を加えたものを考える．近似アルゴリズム 1 と同じように，RB_i, RPE_iを計算し，降順にソートする．そして**図 20** の手順でサブブロックに PE を割り当てる．**図 21** にイメージを示す．近似アルゴリズ

```
void Approx2(){
  int nb=m; /* PE が 1 つも割り当てられていないブロックの数 */   RB, RPE を計算し降順
ソート;
  for(i=0,j=0; j < n; j++) {
    /* B_i に PE_j を割り当てる */
    RB_i = RB_i − RPE_j;
    グループ P_i に PE_j を登録;
    nb を更新;
    /* PE の残りの数と nb を比較 */
    if ((n − j − 1) != nb) {
      /* 次のブロックを探す */
      RB が最大のブロック B_k を探す;
      i = k;
    }
    else {
      nb 個の残っているブロックに 1 つずつ PE を割り当てる;
      break;
    }
  }
  return;
}
```

図 20　近似アルゴリズム 2

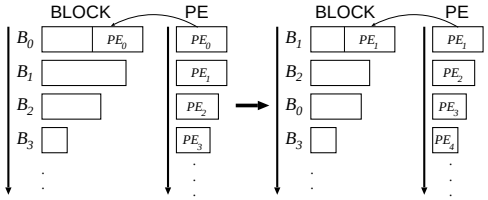

図 21　近似アルゴリズム 2 のイメージ

ム 1 との違いは，1 つの PE を割り当てた時点で RB_iを更新し，ブロックをその RB_iで降順ソートする点である．これで各ブロックに最低 1 個の PE を割り当てる．これを近似アルゴリズム 2(Approx2) とする．

4.2.3　近似アルゴリズム 3

近似アルゴリズム 2 では必ず全ての PE をブロックに割り当てるため，過剰な負荷分散によって良い近似解が求まらない可能性がある．そこで，近似アルゴリズム 2 によってまず各ブロックに PE を割り当てる．そのときブロック B_i に割り当てられた PE の集合を P_iとする．そして B_iに対して P_iの空集合を除く部分集合全てに関して分割を行い，最良の分割を採用する (**図 22**)．この方法

```
void Approx3(){
  /* Approx2 で各ブロックに PE を割り当てる */
  Approx2();
  T[i] : Approx2 によって得たブロック B_i の実行時間 (暫定解)
  P_i : Approx2 によって得た各ブロックの PE 集合
  P3_i : 暫定解が得られてときの P_i

  for(i=0; i < m; i++){
    for(P_i の全部分集合){
      3.3節の下界値 T0 を計算;
      if(T0 < T[i]){
        ブロック分割をして実行時間 TT を計算;
        if(TT < T[i]){
          /* 暫定解の更新 */
          T[i] = T;
          P3_i に部分集合を格納;
        }
      }
    }
  }
  return;
}
```

図 22　近似アルゴリズム 3

では近似アルゴリズム 2 の $2^{|P_i|}-1$ 倍の時間がかかるが，通信遅延が大きい場合も過剰な負荷分散を避けることができると期待される．これを近似アルゴリズム 3(Approx3) とする．

4.2.4　近似アルゴリズム 4

近似アルゴリズム 3 は過剰な負荷分散を防ぐので，精度の良い近似解が得られる．しかし近似アルゴリズム 2 で与えられた全ての部分集合 P_i を探索するため，探索に時間がかかる．そこで，過剰な負荷分散を防ぎ，なおかつより高速に近似解を得られるようなアルゴリズムを検討した．

初めにブロックを格子点数で降順ソートし，各ブロックの実行時間 T を∞にする．そして T が最大のブロックに残存の PE のうち最速のものを割り当てる．T が改善されればブロックを T で降順ソートし同じような割り当てを繰り返す．T が改善されない場合は max_k 回までなら許し，それでも改善されない場合はそのブロックへそれ以上の PE の割り当てを禁止する．すべてのブロックが PE 割り当て禁止になるまでこれを繰り返す (**図 23**)．本研究では max_k を測定の

```
void Approx4(){
  double T[m]; /* 各ブロックの暫定解 */
  int jun[m]; /* 暫定解 T_0で降順ソートした結果*/
  int max_k = 4; /* 改悪を許す回数 */
  int k = 0; /* 改悪の回数 */
  int num = 0; /* 割り当て禁止のブロック数 */
  ブロックを格子点数で降順ソート;
  for(i=0; i < m; i++){
  jun[i] = i;
  T[i] = ∞;
  temp[i] = 0;
  }
  do{
    Tが (num + 1) 番目に大きいブロックの番号を blkとする;
    グループ P_blkに残存中で最速の PE を登録;
    3.3節の下界値 T0 を計算;
    T0 < T[blk] ならばブロック分割をして TTを計算;
    if(TT < T[blk]){
      /* 暫定解を更新してブロックを並び替える */
      T[blk] = T;
      k = 0;
      P_blkを temp_blkに記憶;
      Tで降順ソートに並び替える;
    }
    else{
      if(k > max_k || 残存の PE 数 == 0){
        /* B_blkへの PE の割り当てを禁止する */
        P_blkを temp_blkの状態に戻す;
        num++;
      }
      else{
        k++;
      }
    }
  }while(num < m);
  return;
}
```

図 23　Approx4

結果より 4 とした．これを近似アルゴリズム 4(Approx4) とする．

4.3　再帰的近傍探索

これらの近似アルゴリズムに対して再帰的近傍探索を行う．本研究では PE の割り当てに対する近傍として以下のような 2 つの近傍を考える．

- あるブロックに割り当てられている PE を他のブロックへ移動する (1-近傍)
- あるブロックに割り当てられている PE と他のブロックの PE を交換する (2-近傍)

1-近傍ではブロックに割り当てる PE の数が変わってしまうので，解に与える影響は大きい．そのため各ブロックの実行時間 T_iのバランスがある程度とれている場合にはそれ以上の解の改善が困難で，バランスがとれていないときには有効であると考えられる．

2-近傍は PE の数に変化はなく，解に与える影響は小さい．そのため，T_iのバランスがある程度とれているときに，少しずつ解を改善するのには有効であるが，バランスがとれていないときには，PE 数固定という足かせがあるため，局所解に落ちやすいと考えられる．

この 2 つの近傍を調べ良い方を採用すれば，各ブロックの実行時間のバランスによらず，常に解の改善が可能であると予測される．本研究では，各イテレーションで 1-近傍と 2-近傍を調べ，その中で最も解の改善されたものを採用し，解が改善されなくなるまで再帰的に繰り返すという再帰的 1-2-近傍探索 (Local12) を行う．

5. 評　　価

5.1 評 価 方 法

数値シミュレーションにより本研究で提案した手法を評価する．ブロック数 m，PE 数 n，PE の処理速度 Cta_iとブロックサイズ H_i, W_iを変えながらシミュレーションを行い，分枝限定法で求めたブロック分割と，近似アルゴリズムで求めた分割結果の比較を行った．

ブロック数は $m = 4, 8$ の 2 通りとし，PE 数 n は 4 の倍数として，4 から 4 つずつ増やした．それ以外のパラメータは3章と同じく表 1 の通りである．PE の内訳も 3章と同じで表 3 の通りである．

ブロック B_iの縦方向の格子点数 H_i，横方向の格子点数 W_i(図 2) は，以下の条件に合わせて乱数で生成した．最適化問題の解はデータ依存であるため，各 (m, n) について 100 回の試行を行い平均値をとって評価する．

$$10 \leq H_i, W_i \leq 200 \tag{11}$$

$$H_i, W_i \equiv 0 \pmod{10} \tag{12}$$

このときの求解時間も測定して，100 回の試行の平均で評価する．求解に用

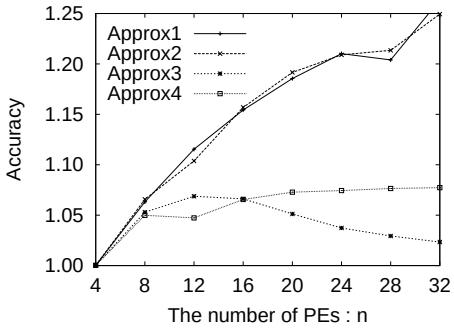

図 24　近似アルゴリズムの精度 (m=4)

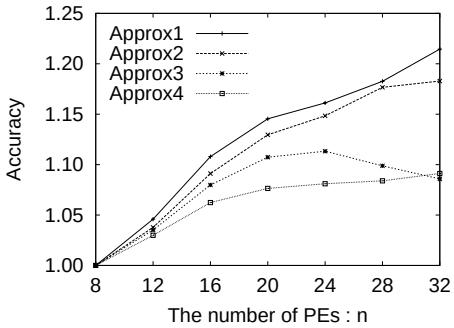

図 25　近似アルゴリズムの精度 (m=8)

いる計算環境は表 2 に示した通りである．

5.2　近似アルゴリズムの評価

第 4.2節で述べた近似アルゴリズムを，分枝限定法で求めた最適解と比較して評価する．ただし各ブロック内での負荷分散には，第 3章の分割法 4+局所的負荷調整による解を用いている．最適解を 1 として正規化した各近似アルゴリズムの精度を**図 24**，**図 25** に示す．

Approx1,2 と Approx3,4 とでは精度にかなり差があり，n が大きくなるとそれが顕著になる．Approx1,2 はともに全ての PE を使い切るため，n が大きくなると過剰な負荷分散が起こるためである．その点，Approx3,4 は PE の使用を制限しているため，過剰な負荷分散が起こりにくく，最適解から 10%以内の誤差で近似解を得ることができる．Approx3 と Approx4 を比較してみると，実行時間に対し計算時間が支配的になる ($m = 4, n \leq 16$) や $m = 8$ のようなケースでは Approx4 が，そうでないケースでは Approx3 が良い精度を得ている．計算時間が支配的になるようなケースでは，Approx3 は Approx2 と同じような PE

の分配をすることになってしまう．通信時間が支配的になるようなケースでは，両者とも PE の使用を制限するが，Approx3 の方が多くの組合せを調べているため，Approx4 よりも良い結果が得られている．

$n = 4, n = 8$ のときの求解時間を**図 26**, **図 27** にそれぞれ示す．図中の Optimize は最適化に要した時間を示している．

m に関わらず Approx3 の求解時間は大きく，n が大きくなるに従って増大している．n が増えると，Approx2 で各ブロックに割り当てられる PE 数も増えるため，それだけ P_iの部分集合も大きくなる．一方 Approx4 の場合，PE を 1 つずつ増やしていき，実行時間が悪くなるところで打ち切るので，n が増えても各ブロックに割り当てる PE 数はあるところで頭打ちになる．そのため，求解時間はそれほど増大しない．最適化時間は n の増加によって指数的に増加し，$n > 32$ では数 100 秒から数 1000 秒と到底実用的とは言いがたい．それに比べると，近似アルゴリズムは十分に高速であると言える．

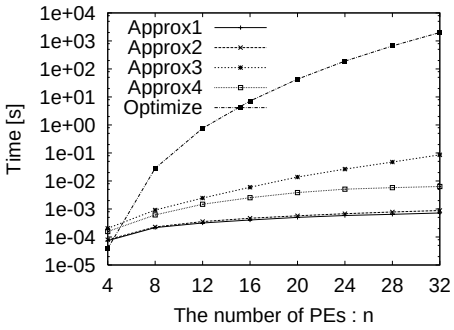

図 26　近似アルゴリズムの求解時間 (m=4)

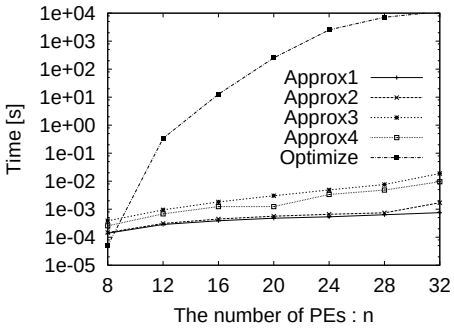

図 27　近似アルゴリズムの求解時間 (m=8)

5.3 再帰的 1-2-近傍探索の評価

再帰的 1-2-近傍探索 (Local12) による効果を示す．結果は最適解を 1 として正規化した各近似アルゴリズムの精度を**図 28**, **図 29** に示す．

近似解の精度があまり良くなかった Approx1,2 では，$n = 32$ で 10%程度と大きな効果が見られる．Approx3,4 でも数%改善されているが，n が大きくなるに従ってその効果は小さくなっている．$(m = 4, n < 12)$ や $(m = 8, n < 24)$ のような計算時間が支配的なケースでは Approx2+local12 の精度が Approx3+Local12 よりも良い．これは Local12 により，すべての PE を使って入れ替え，移動をしているためで，このような場合には効果がある．しかし，そうでない場合には PE 数を制限する Approx3 の方が良い．

これは $m \ll n$ になることにより通信時間が支配的になるため，過剰な負荷分散を回避できるの Approx3 が良くなる．$(m = 8, n \leq 28)$(図 29) の範囲では計算時間が支配的になっており，Approx3 の効果が見られていない．$(m = 8, n > 28)$

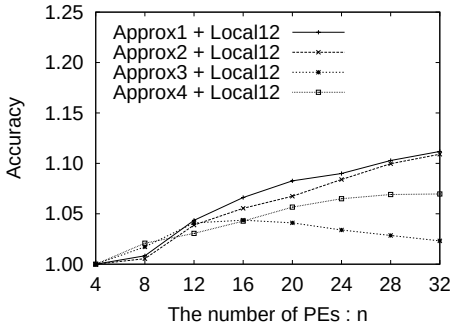

図 28　近似アルゴリズムの精度 (m=4,Local12 あり)

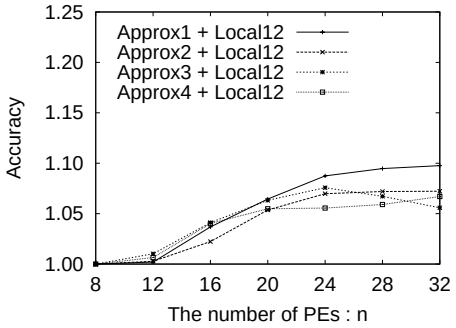

図 29　近似アルゴリズムの精度 (m=8,Local12 あり)

では $m = 4$ の場合と同じような傾向が出ると予測される．

Local12 を行った場合の求解時間を**図 30**, **図 31** に示す．

ここでは Approx3,4+Local12 の求解時間が Approx1,2+Local12 に比べ短くなっている．求解時間が逆転した理由は，これは Approx3,4 の近似解精度は Approx1,2 に比べ良いため，Local12 が早く停止するからである．また，Approx3,4 では PE の使用を制限することから，全ての PE を使い切る Approx1,2 に比べると探索する近傍も狭い範囲になるとも考えられる．Approx3 と Approx3+Local12 の求解時間を比較すると 10 倍から 100 倍程度になるが，$(m = 8, n = 32)$ でも 1 秒程度と，最適解の求解時間と比べると十分に高速である．

5.4 近似アルゴリズムの相対評価

これまでは，近似解を最適解と比較して評価を行った．しかし，$n > 32$ の範囲で最適解を求めることは求解時間的に困難であり，当然，近似解の精度を定量的に評価することはできない．そこで最適解が実用時間内で求められない

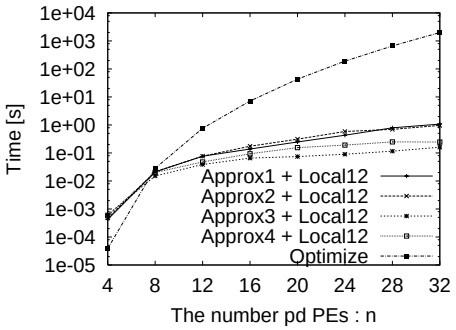

図 30　近似アルゴリズムの求解時間 (m=4,Local12 あり)

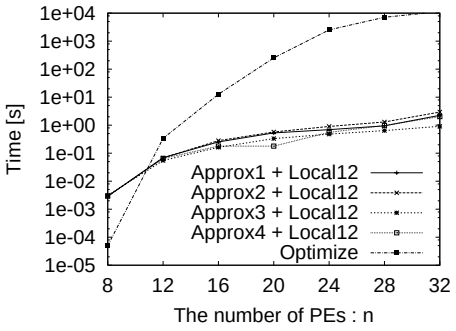

図 31　近似アルゴリズムの求解時間 (m=8,Local12 あり)

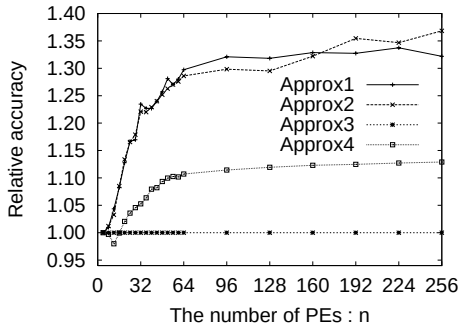

図 32　(n<=256,m=4) の近似アルゴリズムの相対評価

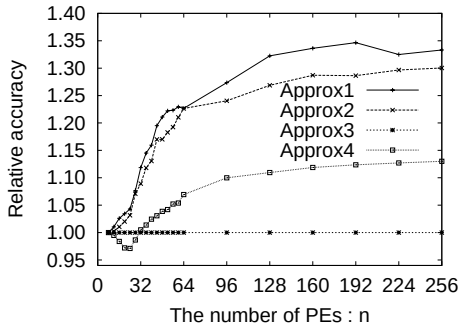

図 33　(n<=256,m=8) の近似アルゴリズムの相対評価

範囲の (m,n) に対して，近似アルゴリズムの相対評価を行った．ここでは最適解の代わりに，最も精度が良いと予測される Approx3 を 1 として正規化した．**図 32**, **図 33** に各近似アルゴリズムの相対精度を示す．最適解と比較した場合と同じように，Approx1,2 は n が大きくなるにつれ，Approx3 より悪くなっていく．$n \geq 160$ では，$m = 4$ の場合が 30%，$m = 8$ の場合が 20%以上の差が見られる．Approx4 もわずかながら n の増加によって悪くなっていく．こちらは $n = 256$ で約 10%の差がある．これは前に述べた最適解と比較したときと同じ傾向である．これによって，4 つの近似アルゴリズムを精度の面から見た場合，Approx3 が一番良い近似解を得ることがわかる．

次に再帰的 1-2-近傍探索を用いた結果を**図 34**, **図 35** に示す．Approx3+Local12 を見ると，n の増加に従って 1 に近付いていくことから，n が大きい，すなわち通信時間が支配的になるケースでは再帰的 1-2-近傍探索は効果がないことがわかる．これは Approx4+Local12 でも言える．Approx1,2 では Local12 による改善は見られるものの，Approx3,4+Local12 と比べると精度は悪い．

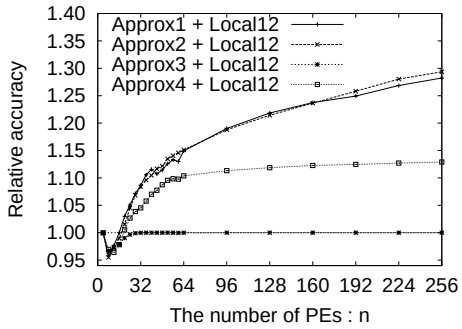

図 34　(n<=256,m=4) の近似アルゴリズムの相対評価 (Local12 あり)

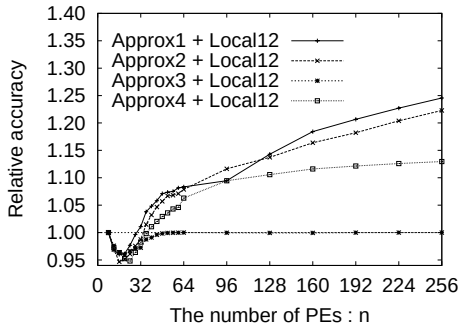

図 35　(n<=256,m=8) の近似アルゴリズムの相対評価 (Local12 あり)

近似アルゴリズムの求解時間を**図 36**, **図 37** に示す．Approx3 は n の増加によって求解時間も増大し，$(n = 256, m = 4)$ に至っては数千秒という時間がかかっている。これは図 26, 図 27 と同じように，n が増加に従って P_iの部分集合が大きくなるためである．一方，Approx4 は数ミリ秒で一定である．求解時間の観点からこの両者を比較した場合，ある程度短時間で近似解が得られる Approx4 の方が良いと言える．

Local12 を適用した場合の求解時間を**図 38**, **図 39** に示す．ここでも図 30, 図 31 と同じような傾向が出ている．Approx4 の求解時間は $(n = 256, m = 8)$ で 10ms 程度だったのが，Approx+Local12 で数秒程度まで増加している．Approx3 も同じように数秒が数 10 秒まで増加している．Approx1,2+Local12 は，Approx1,2 の近似解精度が悪いことと，近傍が広いため，$n = 256$ では数千秒の求解時間となっている．

これらの結果より次のことが言える．

- 通信時間が支配的になる $(m \ll n)$ 場合は PE 数を制限する方が良い．

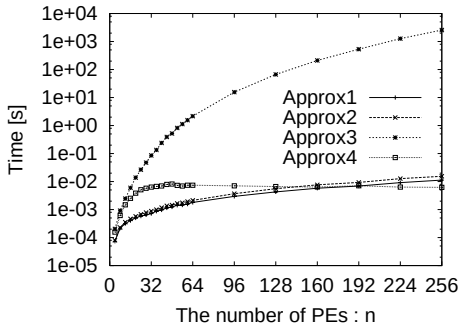

図 36　(n<=256,m=4) の近似アルゴリズムの求解時間

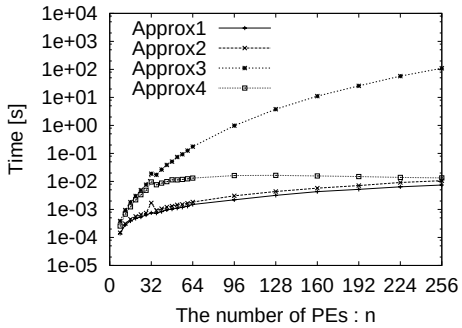

図 37　(n<=256,m=8) の近似アルゴリズムの求解時間

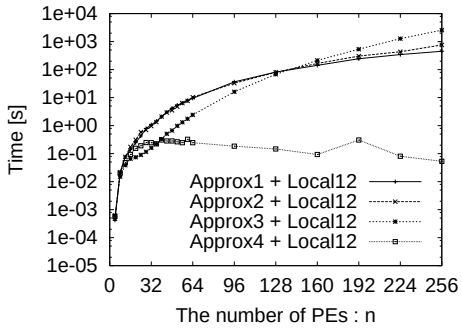

図 38　(n<=256,m=4) の近似アルゴリズムの求解時間 (Local12 あり)

- 再帰的近傍探索は計算時間が支配的になるほど効果的である.

再帰的近傍探索は，利用する PE 数を制限した Approx3,4 ではよいところで 5%程度の解の改善が見られたが，n が大きいところではほとんど効果が見られなかった．そのため，Approx3,4 に関しては再帰的近傍探索は行わなくても良いといえる．Approx3,4 の両者を比較すると精度の面では Approx3，求解時間

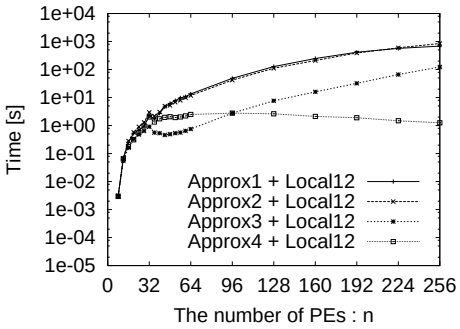

図 39 (n<=256,m=8) の近似アルゴリズムの求解時間 (Local12 あり)

の面では Approx4 が良い．しかし，Approx3 は n が大きいところでは求解時間が大きくなりすぎ，実用的ではない．一方の Approx4 は Approx3 に比べ精度は 10%程度劣るが，n によらず数 10ms という実用的な時間で近似解を得られる．そのため，本論文で提案した 4 つの近似アルゴリズムの中では Approx4 が最も良い近似アルゴリズムであると言える．

6. おわりに

本研究では分散処理環境における数値シミュレーションの静的負荷分散法について検討した．分散処理環境は最適化の対象として考えたとき自由度が非常に大きく，また通信などのモデル化が非常に難しい．今後，どのようなモデルが現実的であるか，またどのような条件で評価すべきなのか，実測を含めてさらに検討していく必要がある．

参 考 文 献

1) 市川周一, 川合隆光, 島田俊夫: 組合せ最適化による並列数値シミュレーションの静的負荷分散, 情報処理学会論文誌, Vol. 39, No. 6, pp. 1746–1756 (1998).

2) 川合隆光, 市川周一, 島田俊夫: 並列数値シミュレーション用高水準言語 NSL, 情報処理学会論文誌, Vol. 38, No. 5, pp. 1058–1067 (1997).

3) 中田秀基, 高木浩光, 松岡聡ほか: Ninf による広域分散並列計算, 並列処理シンポジウム JSPP '97, pp. 281–288 (1997).

4) Casanova, H. and Dongara, J.: NetSolve: A Network Server for Solving Computational Science Ploblems, Proceedings of Super Computing '96

(1996).

5) Liu, C. L.: 組合せ数学入門I, 共立出版 (1972).

6) 茨木俊秀: 組合せ最適化, 産業図書 (1983).

7) 坂和正敏: 数理計画法の基礎, 森北出版, chapter 3.4, pp. 111–132 (1999).

8) Fox, G.C., Williams, R.D. and Messina, P.C.: *Parallel Computing Works!*, Morgan Kaufmann, chapter 11.1.5 (1994).

9) Fox, G. C.: A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube, *Numerical Algorithms for Modern Parallel Computer Architectures* (Schultz, M.(ed.)), Springer-Verlag, pp. 37–62 (1988).

謝辞　本研究の一部は，(財) 電気通信普及財団・平成 10 年度研究助成，文部省科学研究費補助金・特定領域研究 (B) (2) 10205210 および奨励研究 (A) 11780211 によるものである．