並列数値シミュレーションの
静的負荷分散法の拡張について

Enhancements on Static Load-balancing Scheme
for Parallel Numerical Simulations

知識情報工学専攻　931714　藤村 佳克

1　はじめに

物理現象のシミュレーションなど科学技術計算では大規模な計算が必要とされる．大規模計算分野では計算の高速化に対する要求は果てしない．

並列処理は，複数の演算装置を同時に動作させることにより計算速度の向上を目指す方法である．並列処理を用いた数値シミュレーションの高速化に関しては，現在までに様々な研究が行われてきた．プロセッサ台数に応じた速度向上を得るためには適切な負荷分散が必須である．適切な負荷分散を行うためには計算時間とプロセッサ間で発生する通信を考慮する必要がある．不適切な負荷分散を行うと通信によるオーバヘッドにより実行時間が増大してしまう恐れがある．負荷分散の手法としては，コンパイル時に行う静的負荷分散と実行時に行う動的負荷分散がある．負荷の性質が予めわかっている場合は静的負荷分散が有効である．本研究では並列数値シミュレーションの静的負荷分散について考える．

計算時間と通信時間を考慮してマルチプロセッサ上で実行時間を最小化する研究は以前から行われ，既に幾つもヒューリスティック・アルゴリズムが提案されている [1]．しかしこれらはいずれも固定されたタスクグラフに関して静的負荷分散を行うためのものである．一方本研究の目的は，本質的にデータ並列性のある問題に対して，データと計算の自動分割を行って全体の実行時間を最小化することである．

市川ら [2] は並列偏微分方程式求解システム NSL の静的負荷分散問題を組合せ最適化問題として定式化し，分枝限定法を用いて実用的に最適解を求める方法を示した．文献 [2] での静的負荷分散は，計算領域を複数のブロックに分割し，計算量と通信量を考慮して各ブロックに適切な数のプロセッサを割り当てることによって実行時間を最小化するというものである．この方法ではブロック数 m とプロセッサ数 n に関して $m \ll n$ の関係が満たされないと適切な負荷分散を行うことができない．この制限を緩和するため仮想プロセッサを用いる方式も提案された [3] が性能が改善可能であることが示されただけで，具体的・実用的な解決法は未解決のままであった．

本研究では文献 [2] の手法で適切な負荷分散が困難となる $m \geq n$ の場合について，静的負荷分散問題を各プロセッサに１つ以上のブロックを割り当てる一種の箱詰め問題 [4] としてモデル化し，分枝限定法 [4][5] を用いて最適解を求める方法を示す．この最適化問題の探索空間は $O(n^m)$ であるため，規模が大きな問題では実用時間内に最適解を求めることが不可能になる．そこで本研究では短時間で精度の良い近似解が求まる近似アルゴリズムを合わせて検討する．実用時間内に最適解が求まる小規模な問題に対しては分枝限定法を用いて最適解を求め，その最適解を基準として

近似解の精度を定量的に評価した．実用時間内に最適解が求まらない大規模な問題に対しては，最良の近似解との相対評価を行いその精度を評価した．最適解，近似解の求解時間についても測定を行った．

本研究ではプロセッサに対してブロックを割り当てる方法を検討するが，ブロックの計算量自体に大きな不均衡があれば，適切な負荷分散を行っても割り当て結果に必ず不均衡が残る．文献 [1] 等の問題設定であればここで問題は終ってしまうが，本研究ではさらにデータの自動分割 [2] を併用して負荷を平準化し，全体の実行時間を最小化することを目的とする．本論文ではその最終目的への 1 ステップとして，静的負荷分散を箱詰め問題としてモデル化し解法を検討する．

2　計算モデル

2.1　NSL の概要

数値シミュレーションにおける多くの問題は偏微分方程式として定式化される．これは自然現象の多くが近接作用を基本原理としているためである．NSL は時間に関して 1 階，空間に関して 2 階までの 2 次元偏微分方程式を並列に解くためのものである [6]．ユーザが解くべき偏微分方程式と計算領域を記述すると適切な並列処理プログラムを自動生成する．偏微分方程式は差分式に変換して陽解法で解く．陽解法は陰解法より高いデータレベル並列性を持つため並列処理に適していると考えられる．

自動並列化を目的とした研究は他にもあるが，NSL はマルチブロック法と境界適合法を用いた点において他のものとは異なっている．境界適合法は複雑な形状をもつ物理領域を座標変換によって単純な短形の計算領域に写像する．マルチブロック法の適用により，写像が困難な複雑な物理領域を複数の計算ブロックからなる計算領域に写像することができる．マルチブロック法，境界適合法の適用により各計算ブロックが短形になり，並列化やベクトル化による性能向上が容易になる．従来の研究では物理領域の形状が不規則だと負荷の均衡化が困難であった．

2.2　計算ブロック

NSL の計算領域は m 個の並列に処理されるブロック $B_j (j = 0, ..., m-1)$ から構成されている．各ブロックは相互に接続されているが，ここでは最も基本的な接続関係として，図 1のような木構造で接続されているとする．図内の矢印はブロック間での通信を表している．

各ブロックは 2 次元に配列された格子点で構成されており，各格子点

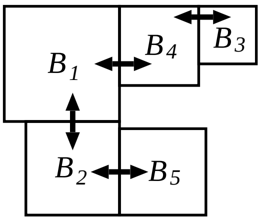

図 1: 計算ブロック

では解くべき偏微分方程式に対応した差分式を陽解法で計算する．中心差分を採用する場合，各格子点の計算には座標軸に沿って正負両側の格子点の値が必要となる．陽の差分式では時刻 t の値を時刻 $t-1$ のデータから計算するので，時刻 t における各格子点の計算には相互のデータ依存性がなく並列に実行することができる．従ってブロックを並列計算機の各要素プロセッサに割り当てれば，各プロセッサは並列に差分式を計算することができる．

各時間ステップの計算後には，各格子点のデータを交換する必要がある．このデータ交換のためにブロックの辺縁部 (図 1の矢印部) でプロセッサ間通信が発生する．ただし同一プロセッサ内でのデータ交換時間は無視できる程度に小さいと仮定する．発生する通信量は隣接しているブロック間の隣接部の格子点数に比例すると考える．

2.3 要素プロセッサ

並列計算機では，一般に各プロセッサの能力は均一である．そこで文献 [2] ではプロセッサの能力が均一な場合を扱っていた．しかし本研究では条件を一般化し，要素プロセッサの能力が不均一であっても良いとする．これによって，小規模 PC クラスタなどの分散環境に関しても提案アルゴリズムが有効か否か検証できる．

2.4 計算手順

計算手順は文献 [2] の計算手順 1 と同様なものを用いる．各要素プロセッサ上での計算は，割り当てられた計算ブロックの計算を行った後，プロセッサ間で発生する通信を計算する．

2.5 評価関数

評価関数としてはシミュレーションの 1 時間ステップの実行時間を用いる．本研究の目的は，m 個のブロックを n 個のプロセッサに割り当てる方法 ($O(n^m)$ 通り) の中から，全体の処理時間 T を最小にする割り当てを探し出すことである．

ある割り当てを選んだ時，全体の処理時間 T は各プロセッサ P_i での処理時間 T_i のうち最大のもので決まるので，

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

と表される．

T_i は，P_i が担当しているブロックの処理時間の和で求まる．P_i が担当するブロック B_j の処理時間は計算時間 Ta_j と通信時間 Tc_j の和で求まるので，T_i は，

$$T_i = \sum_{B_j \in G_i} (Ta_j + Tc_j) \tag{2}$$

と表される．ここで，G_i は P_i が担当しているブロックの集合を表している．

計算時間 Ta_j はブロック内の格子点数 Sa_j の一次関数で近似できると考えられるので，以下の式でモデル化される．

$$Ta_j = Cta_i Sa_j + Dta_i \tag{3}$$

ここで Cta_i はプロセッサ P_i が一つの格子点での計算に要する時間，Dta_i は遅延時間である．

B_j と隣接しているブロックを B_l と置く．B_j と B_l 間の通信時間 $Tc_{j,l}$ が通信量 $Sc_{j,l}$ の一次関数で近似できるとすると，B_j を P_i に割り当てた時の通信時間 Tc_j は以下の式で表される．

$$Tc_{j,l} = CtcSc_{j,l} + Dtc \tag{4}$$

$$Tc_j = \sum_{B_l \notin G_i} Tc_{j,l} \tag{5}$$

ここで，Ctc は格子点 1 つあたりの通信時間，Dtc は通信の遅延時間である．Cta_i, Dta_i, Ctc, Dtc はいずれも計算機やプログラムの実装に依存して変化するので，測定によって決定する必要がある．

2.6 NSL の静的負荷分散の拡張

本研究で考える，m 個のブロックを n 個のプロセッサに割り当てる方法の中から T を最小にする割り当てを探し出す問題は，一種の箱詰め問題

である．この最適化問題の探索空間は，m ,n が大きくなると組合せの数が増大し最適解探索に要する時間が指数関数的に増大してしまう．そこで探索空間を制限するために組合せ最適化問題にしばしば適用される分枝限定法 [4][5] を用いる．

分枝限定法は図 2 のような探索木を作成し，現在得られている暫定解よりも良い解を得る可能性のある枝を探索し，可能性のない部分木は早い時期に切り落とすという操作を繰り返すことにより探索空間を制限し，短時間で最適解を見つけ出す方法である．枝を切り落とすためには下限値を用いる．最適解が短時間で見つかるかどうかは精度の良い初期暫定解の発見，どのような下限値を用いるかに大きく左右される．精度の良い初期暫定解を利用すると探索初期から探索空間がある程度制限されるため最適解の探索に要する時間が短縮される．

本研究ではヒューリスティックを用いた近似アルゴリズムにより初期暫定解を求める．探索木の探索には暫定解を用いて分枝を制限していくため，また記憶量の点で有利なため，深さ優先探索を用いる．枝の限定操作は部分木 Sに対する制約条件を緩和して下限値 $L(S)$ を求め，その値と現在の暫定解$\hat{T}$を比較することにより行う．

$$L(S) \geq \hat{T} \tag{6}$$

が成立すれば，その部分木を探索しても現在の暫定解よりも良い解が求まる可能性がないということなのでその枝は切り落とす．

図 2は図 1の計算ブロックを 2 台のプロセッサに割り当てる場合の分枝図である．() 内の数字は枝の限定操作を行わずに探索を行った場合の探索木の探索順序を表している．図内の各ノードはある割り当て時点においてどのブロックをどのプロセッサに割り当てるかを表している．例えば (1) のノードはブロック B_1をプロセッサ P_0に割り当てることを表している．ノード (5) の時点ではP_0に B_1, B_2 ,B_5 ,B_4 ,B_3が割り当てられていることになる．図内の$\bar{g}$はまだ割り当て先が決まっていないブロックの集合である．ブロックのプロセッサへの割り当ては早期に暫定解に近付くようにブロックの大きいものから行う．そのためブロックは大きい順に並べておく．図 1の計算ブロックでは B_1, B_2 ,B_5 ,B_4 ,B_3という順番でプロセッサに割り当てる．各割り当て時点では，まだ割り当て先が決まっていないブロックの中で一番大きなブロックをどのプロセッサに割り当てるかを考えるので，各ノードからの分枝はプロセッサ数分だけ分枝する．図 2では 2 台のプロセッサに割り当てることを考えているので各ノードからの分枝数は 2 である．図内の d は探索木上での各ノードの深さを表している．枝の限定操作のための下限値 $L(S)$ の計算は，まだ割り当て先が決まってい

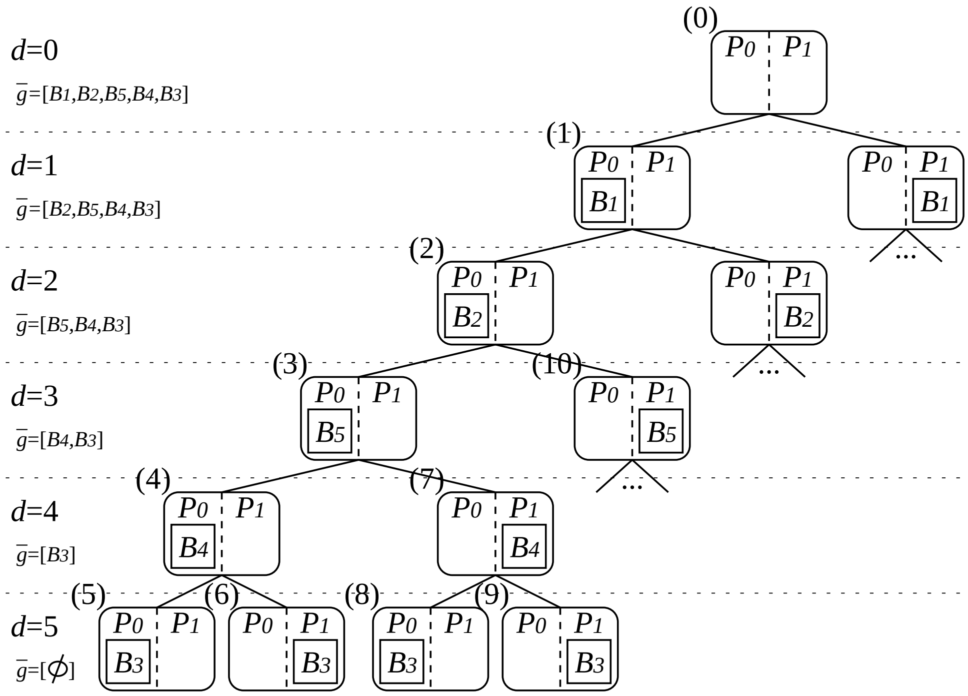

図 2: 図 1 の計算ブロックに対する分枝図

ないブロックの計算量だけを考慮し図形制約は除去する．下限値 $L(S)$ は以下の式によって計算する．

$$L(S) = \left(\sum_{B_j \in \bar{g}} Ta_j + \sum_{i \in n} \tilde{T}_i \right) / n \tag{7}$$

$\tilde{T}_i$は，まだ全ての割り当てが終わっていない段階でのプロセッサ P_iの処理時間である．既に P_iに割り当てられているブロックの集合を g_i，まだ割り当て先が決まっていないブロックの集合$\bar{g}_i$ とすると，$\tilde{T}_i$の見積り値は以下のようになる．

$$\tilde{T}_i \quad = \quad \sum_{B_j \in g_i} \left(Ta_j + \sum_{B_l \notin g_i \cup \bar{g}} Tc_{j,l} \right) \tag{8}$$

2.7　既存の研究

負荷を適切にプロセッサに割り当てて実行時間の最小化を図るという問題は“実行時間最小マルチプロセッサ・スケジューリング問題”と呼ば

れ，長期にわたって研究が行われている．しかしこの問題は一般的に難しい最適化問題であり，種々の制約条件を付加してもそのほとんどがNP困難となる．最適化アルゴリズムも開発されているが，非常に厳しい制約条件が付加されている．Hu[7]は，タスクの処理時間が全て1で等しいという制約下で，タスクの先行制約がツリー状の場合の最適なアルゴリズムを開発した．Coffman・Graham[8]は，タスクの処理時間が全て1で等しいという制約下で，タスクの先行制約が任意，プロセッサ数が2台という場合の最適なアルゴリズムを開発した．しかし一般的な問題に近付けるために制約を少しでも緩めるとNP困難問題となり，Coffman・Grahamのアルゴリズムを適用しても最適解が求まる保障はない．

NP困難となる問題を解くためのアルゴリズムもShirazi，WangとPathak[9]，WuとGajski[10]，SihとLee[11]，YangとGerasoulis[12]，KwokとAhmad[13]などにより開発されている．しかしどのアルゴリズムもヒューリスティックな静的スケジューリングアルゴリズムであり最適解が求まる保障はない．これに対し，笠原ら[14][15]はNP困難問題となる一般的な問題に対して最適化アルゴリズムを開発した．このアルゴリズムは各タスクの処理時間が異なり，先行制約が任意，プロセッサ数が任意という問題に対しても有効であり実用性の高いスケジューリングアルゴリズムである．しかしこれらの研究はいずれも固定されたタスクグラフに対して静的負荷分散を行うためのものであり，本研究の目的とは異なっている．本研究で考える与えられた計算ブロックをプロセッサに割り当てて，実行時間の最小化を図るという研究は一見既存の研究と同様なもののように見えるが，本研究の最終目的はデータ並列性のある問題に対して箱詰めとデータ分割を併用して実行時間の最小化を図ることである．

3 近似アルゴリズム

本研究では分枝限定法を用いて最適化問題の探索空間を制限し，最適解を求める．2.6節で述べたように精度の良い初期暫定解を近似アルゴリズムを用いて求める．本研究ではブロックの計算時間と通信時間を考慮した5通りの近似アルゴリズムを提案し，その精度を検討し，さらに近似アルゴリズムによる初期暫定解を出発点とした再帰的近傍探索(Local)も試みる．複数の近似アルゴリズムに近傍探索を適用して，その中で最良の近似値を選択する手法(BestEffort)も検討した．

図1の計算ブロックに対して各アルゴリズムを適用した結果を図5，7，9，11，13に示す．ここでは単純化のため，プロセッサ能力は全て同一としている．横軸は時間で，各近似アルゴリズムで得られる全体の処理時間

Tも示してある．図中の数字 (i,j) はブロック B_i, B_j間の通信である．

以下の近似アルゴリズムでは，ブロックを順番に選び出し，評価関数を用いて割り当て先のプロセッサを決定する．探索などは行わず，必要があれば事後に再帰的近傍探索で改良することとする．これらの近似アルゴリズムで問題となるのは，まだ割り当てられていないブロックが残っている段階で，ブロックの割り当て先を判断する (そのための評価関数が必要) ということである．まだ割り当てが終っていない段階での Tの見積り値を，以下のように$\tilde{T}$ と表現する．

$$\tilde{T} = \max_i \tilde{T}_i \tag{9}$$

3.1 近似アルゴリズム 1 (Approx1)

ブロックの計算時間だけを考慮して割り当てを行う．図 3 に Approx1 のアルゴリズムを示す．まずブロック B_jを格子点数 Sa_jに従って降順ソートし (図 4)，その順番にプロセッサへの割り当てを決める．割り当て先には，B_jを割り当てた場合の累積計算時間が最小となるプロセッサを選ぶ．B_jを P_iに割り当てる場合の評価関数を $f_1(i,j)$ とすると，$f_1(i,j)$ は以下の式で表される．

```
void approx1(m,n,incumbent)
int m,n; /* m はブロック数，n はプロセッサ数 */
double *incumbent; /* 暫定解 */
{
  int i,j;
  double $f_1$(i,j);

  ブロックを降順ソート;
  for(j=1;j<=m;j++){
    for(i=0;i<n;i++){
      仮に Bj を Pi に割り当ててみる;
      $f_1$(i,j) を計算;
    }
    $f_1$(i,j) が最小となる Pi に Bj を割り当てる;
  }
  T を計算;
  *incumbent=T;
}
```

図 3: Approx1 のアルゴリズム

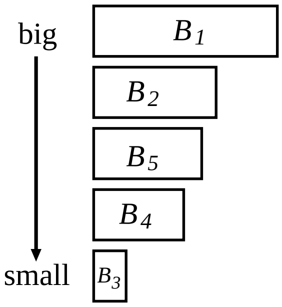

図 4: 降順ソート 1

$$f_1(i,j) = Ta_j + \sum_{B_k \in g_i} Ta_k \tag{10}$$

Ta_jはプロセッサの処理能力 Cta_iに依存して変るので，Ta_jの項も i に依存して変化する．B_jを割り当てる際には全ての P_iに関してこの $f_1(i,j)$ を計算し，$f_1(i,j)$ を最小にする P_iを選んで B_jを割り当てる (B_jを g_iに入れる)．ただし式の第 2 項の総和は過去の割り当ての結果なので，プロセッサ毎に現状を覚えておけば毎回計算する必要はない．図 5は，図 1の計算ブロックに対して Approx1 を適用した結果である．

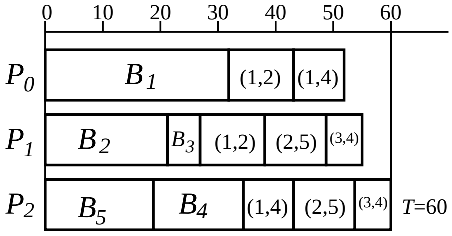

図 5: 近似アルゴリズム 1 の例

3.2 近似アルゴリズム 2 (Approx2)

ブロックの計算時間と通信時間を考慮し割り当てを行う．通信時間に関しては，既に他のプロセッサに割り当てられているブロックへの通信だけを考慮し，割り当て未定のブロックに関しては考慮しない．

図 6に Approx2 のアルゴリズムを示す．まず Approx1 と同じように B_jを降順ソートし，その順で割り当てを決めていく．B_jを P_iに割り当てる場合の評価関数 $f_2(i,j)$ は以下の式で表される．$f_2(i,j)$ を最小にする P_i を選んで B_jを割り当てる．図 7 が図 1の計算ブロックに対して Approx2 を適用した結果である．

$$f_2(i,j) = \tilde{T}_i + Ta_j + \sum_{B_l \notin g_i \cup \bar{g}} Tc_{j,l} \tag{11}$$

```
void approx2(m,n,incumbent)
int m,n; /* m はブロックク数，n はプロセッサ数*/
double *incumbent; /* 暫定解 */
{
  int i,j;
  double $f_2$(i,j);

  ブロックを降順ソート;
  for(j=1;j<=m;j++){
    for(i=0;i<n;i++){
      仮に Bj を Pi に割り当てて見る;
      $f_2$(i,j) を計算;
    }
    $f_2$(i,j) が最小となる Pi に Bj を割り当てる;
  }
  *incumbent=$f_2$(i,j);
}
```

図 6: Approx2 のアルゴリズム

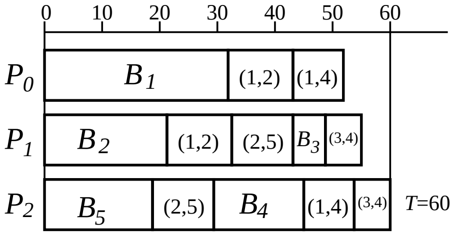

図 7: 近似アルゴリズム 2 の例

3.3　近似アルゴリズム 3 (Approx3)

Approx1，2 では各プロセッサでの処理時間$\tilde{T}_i$しか考慮していなかったが，Approx3 では全体の処理時間$\tilde{T}$を考慮して割り当てを行う．B_jを P_iに割り当てた場合，通信時間は P_iだけでなく B_jの接続先 B_lを担当しているプロセッサにも発生するので，$\tilde{T}$も変わる可能性があるからである．

図 8に Approx3 のアルゴリズムを示す．まず Approx1 と同じように B_jを降順ソートし，ブロックを順に取り出して割り当ててゆく．ブロックを割り当てた時，$\tilde{T}$が最小となるプロセッサを選んでブロックを割り当てる．$\tilde{T}$が最小となる割り当て候補が複数ある場合は，$\tilde{T}_i$が最小となるような P_iを選ぶ．

```
void approx3,4(m,n,incumbent)
int m,n; /* m はブロック数，n はプロセッサ数 */
double *incumbent; /* 暫定解 */
{
  int i,j;
  double $\tilde{T}$,$\tilde{T}i$;

  ブロックを降順ソート;
  for(j=1;j<=m;j++){
    for(i=0;i<n;i++){
      仮に Bj を Pi に割り当てて見る;
      $\tilde{T}$，$\tilde{T}i$を計算;
    }
    if($\tilde{T}$が最小となる候補が複数ある)
      $\tilde{T}i$が最小となる Pi に Bj を割り当てる;
    else
      $\tilde{T}$が最小となる Pi に Bj を割り当てる;
  }
  *incumbent=$\tilde{T}$;
}
```

図 8: Approx3,4 のアルゴリズム

図 9 が図 1の計算ブロックに対して Approx3 を適用した結果である．ブロック B_3のようにブロックの計算時間より通信時間が多い場合に有効に働く．Approx2 では T_iしか考慮してないので B_3を P_1に割り当ててしまっている．そのため P_2との間に通信が発生し T増加の原因となっている．Approx3 を適用することにより図 9のように Tを短縮することができる．

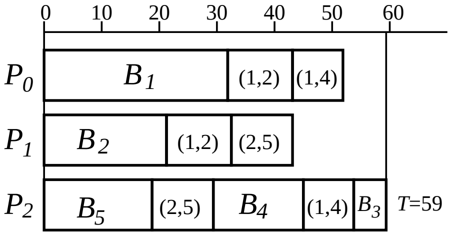

図 9: 近似アルゴリズム 3 の例

3.4 近似アルゴリズム 4 (Approx4)

Approx1，2，3 でのブロックの降順ソートではブロックの大きさしか考慮してなかったが，ブロックで発生する通信も考慮し降順ソートを行う(図 10)．図 10から分かるように B_4と B_5の順番が入れ替わっている．

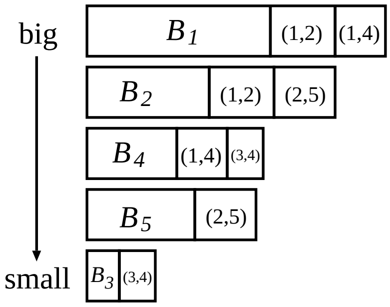

図 10: 降順ソート 2

プロセッサへの割り当てはApprox3 での割り当てと同じ手法を使用する．図 8に Approx4 のアルゴリズムを示している．図 11が図 1の計算ブロックに対して Approx4 を適用した結果である．発生する通信も考慮してブロックの割り当て順を決めることにより，通信が多く発生して割り当て時にネックとなるブロックを早く割り当てることができる．

3.5 近似アルゴリズム 5 (Approx5)

Approx1，2，3，4 はブロックの割り当て順を決めてからプロセッサを選択していたが，Approx5 ではプロセッサを選択してから割り当てるブロックを決める．

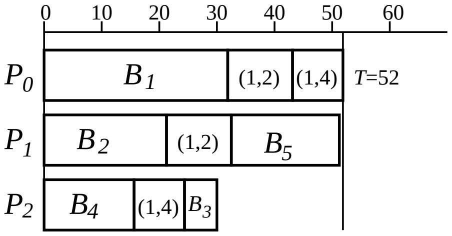

図 11: 近似アルゴリズム 4 の例

図 12に Approx5 のアルゴリズムを示す．まず，割り当てるプロセッサを決めるために，まだ割り当てられてないブロックの中で Sa_jが最大のブロック B_hを選び，B_hを割り当てた場合に$\tilde{T}_i$が最小となるプロセッサ P_iを選択する．要素プロセッサの能力が同じであれば，この部分は単に一番$\tilde{T}_i$の小さいプロセッサを選ぶだけである．プロセッサ能力が均一でない場合，$\tilde{T}_i$が小さくても遅いプロセッサを選んでブロックを割り当ててしまうと，そこがボトルネックになる可能性がある．そこで計算量をたよりに，この方法で能力の割に空いているプロセッサを選び出す．

選択した P_iに対して，各ブロック B_jに対応する優先度 Q_jを計算する．そして Q_jが最大のブロックを P_iに割り当てる．Q_jは以下の式で計算する．

$$\begin{aligned} Q_j &= Cta_i Sa_j + Dta_i \\ &+ \sum_{B_l \in g_i} (Cts Sc_{j,l} + Dts) \\ &- \sum_{B_l \notin \bar{g} \cup g_i} (Cts Sc_{j,l} + Dts) \end{aligned} \tag{12}$$

Q_jは以下の点を考慮し計算している．

- Sa_jの大きなブロックは優先度を上げる．
- 接続先が P_i上にあれば優先度を上げる．
- 接続先が P_iの外にあれば優先度を下げる．
- 割り当て未定のブロックは考えない．

Q_jはブロックを一つ割り当てるごとに再計算する．図 13 が図 1の計算ブロックに対して Approx5 を適用した結果である．他のアルゴリズムでは 2 番目に割り当てられていたブロック B_2が Approx5 では 4 番目に割り当てられる．これは B_2が Sa_jは大きいが発生する通信が多いため Q_jが小さくなるからである．

```
void approx5(m,n,incumbent)
int m,n; /* m はブロック数，n はプロセッサ数 */
double *incumbent; /* 暫定解 */
{
  double $\tilde{T}$,$\tilde{T}i$;

  while(全てのブロックが割り当てられていない){
    まだ割り当てられてないブロックの中で一番大きい Bh を選択;
    for(i=0;i<n;i++){
      仮に Bh を Pi に割り当てて見る;
      $\tilde{T}i$を計算;
    }
    $\tilde{T}i$が最小となる Pi を選択;
    for(j=1;j<=m;j++){
      if(Bj はまだ割り当てられていない)
        Pi に対して，Bj の Qj を計算;
    }
    Qj が最大の Bj を Pi に割り当てる;
    $\tilde{T}$を計算;
  }
*incumbent=$\tilde{T}$;
}
```

図 12: 近似アルゴリズム 5

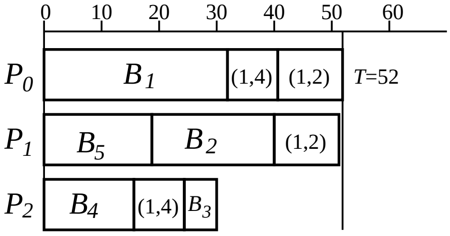

図 13: 近似アルゴリズム 5 の例

3.6 再帰的近傍探索 (Local)

近似アルゴリズムで得られたブロックの割り当てを改善する方法を考える．ブロックを2つ取り出して入れ換えた場合，Tが小さくなるかどうか調べ，小さくなるならばその入れ換えを採用する．さらにTが小さくなる入れ換えがないか，再帰的にこの手続きを繰り返す．どのように入れ替えを行ってもTが改良されなくなったら，入れ替えを終了する．

Localのアルゴリズムは図14に示す．プロセッサ上の2つのブロックB_i, B_jを取り出して入れ換えた場合の全体の処理時間$T_{i,j}$を計算する．2つのブロックの入れ換えの全パターン($m(m-1)/2$通り)について計算する．現在の暫定解$incumbent$に対して

$$\min_{i,j} T_{i,j} < incumbent \tag{13}$$

が成り立つならば，暫定解を$\min_{i,j} T_{i,j}$に更新する．暫定解が更新されたならば，さらに$incumbent$が小さくなる入れ換えがないか再帰的に探索する．暫定解が更新されなくなると入れ換えは終了する．

```
void local(m,n,incumbent)
int m,n; /* m はブロック数，n はプロセッサ数 */
double *incumbent; /* 暫定解 */
{
  int i,j;

  do{
    for(i=m;i>1;i--){
      for(j=i-1;j>=1;j--){
        if($B_i, B_j$ は同一プロセッサ上でない){
            仮に $B_i, B_j$ をプロセッサ上で入れ換える;
            $T_{i,j}$ を計算;
        }
      }
    }
    if($\min_{i,j} T_{i,j}$ < *incumbent){
      $\min_{i,j} T_{i,j}$ となる組み合わせで $B_i, B_j$ をプロセッサ上で入れ換える;
      *incumbent=$\min_{i,j} T_{i,j}$; /* 暫定解の更新 */
    }
  }while(暫定解が更新された);
}
```

図 14: Local のアルゴリズム

4　評価結果

プロセッサ能力が均一な場合と不均一な場合についてブロック数を変えながらシミュレーションを行い，近似解の精度を測定した．各ブロックは正方形とし，各ブロックの格子点数は100から10000までの範囲でランダムに決めた．プロセッサ数nは4とした．$m \leq 32$での結果は分枝限定法で求めた最適解を1として正規化した．ブロック数が32以上では実行時間が長くなり最適解を求めることが困難なので，$m \leq 256$の範囲で最良の近似解Best Effortを1とした相対評価を行った．近似解の求解時間の測定も行った．Approx$i$+Localは近似アルゴリズムiに再帰的局所探索を適用した結果である．optimizationは最適解の求解時間である．最適化問題の解はデータに依存するので，各ブロック数で20回の試行を行ない，その平均値で評価を行なっている．

4.1　プロセッサ能力が均一な場合

各パラメータは表1のとおりである．これらのパラメータは実装依存であるが，本研究では特定の実装によらず一般的と思われる値を設定した．プロセッサが均一という条件なので，Cta_i, Dta_iは全てのプロセッサで同じであるとした．

表1: パラメーター覧

名前	値	名前	値
Cta_i	1.0	Cts	20.0
Dta_i	0.1	Dts	0.1

4.1.1　近似解の精度

$m \leq 32$で，最適解を1とした場合の近似解の精度が図15，16である．図15，16を比較することにより，LocalによってApprox1～4の精度が一定の値まで改善されているのを確認できる．Approx5の精度も他の近似アルゴリズムほど大きくないがLocalにより改善されている．Approx5に対してLocalの効果が小さいのは，Approx5が他の近似アルゴリズムに比べ比較的精度の高い近似解を求めることができ，早く局所最適解に落ちるからである．ブロック数が32以下ではApprox5，Approx5+Local，Best Effortの解は最適解から誤差10%以下である．

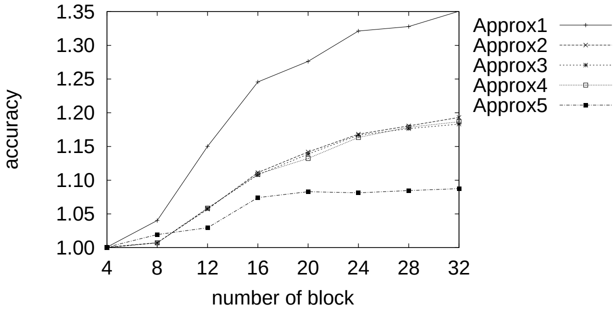

図 15: 近似解の精度 1(プロセッサ均一)

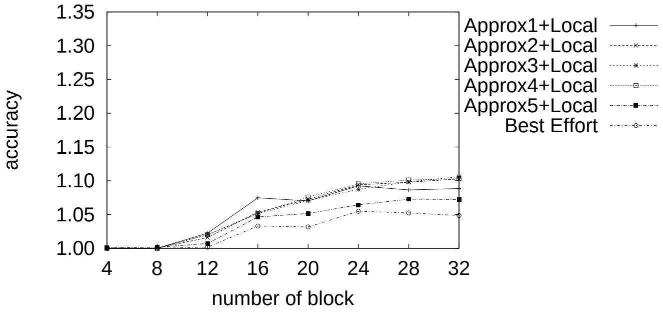

図 16: 近似解の精度 2(プロセッサ均一)

$m \leq 256$ で，最良の近似解 Best Effort を 1 として相対評価を行った結果が図 17，18である．図 17，18よりブロック数が増加しても Local が有効に働いていることを確認できる．Approx5，Approx5+Local はブロック数が増加しても他の近似アルゴリズムに比べ高い精度を示している．

Approx5 が他の近似アルゴリズムより精度が高いのは，Approx1～4 が降順ソート 1, 降順ソート 2 によりブロックの割り当て順序を固定するのに対して，Approx5 では優先度 Q_j を繰り返し計算しながら動的にブロック選択を行っているためである．

4.1.2 近似解の求解時間

図 19，20は近似解の求解時間を測定したものである．Approx5 の求解時間が最も短い．Approx5+Local は他の近似アルゴリズムに Local を適用したものと比べ，短い時間で解を求めることができている．Approx1

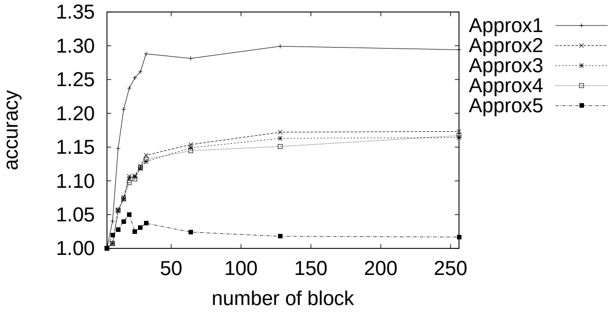

図 17: 近似解の相対評価 1(プロセッサ均一)

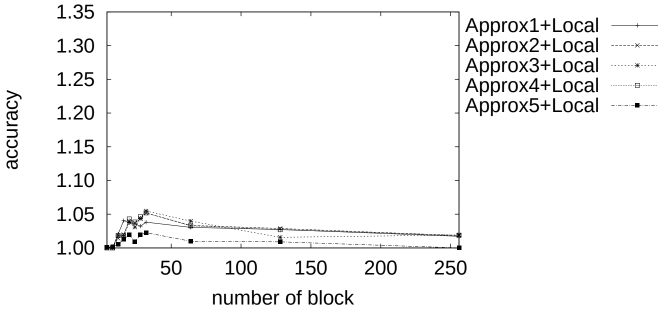

図 18: 近似解の相対評価 2(プロセッサ均一)

~4 に対しての Local に時間がかかるのは，元々の近似解の精度が悪いためである．Best Effort は複数の近似アルゴリズムに Local を適用するので当然一番時間がかかる．最適解の求解時間と近似解の求解時間を比較することにより，近似解が最適解の求解時間に比べ遥かに短い時間で求まっていることを確認できる．この結果より求解時間の面からも Approx5，Approx5+Local は有効な近似アルゴリズムであると考えられる．しかし，分枝限定法の初期暫定解としては Best Effort を用いるのが最適である．近似解の求解時間は最適解の求解時間に比べれば遥かに小さいので，少しでもよい初期暫定解を用いて探索空間を制限し，最適解の求解時間を短縮することが望ましい．

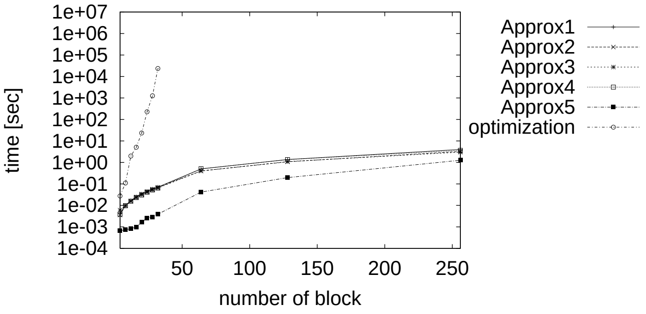

図 19: 近似解の求解時間 1(プロセッサ均一)

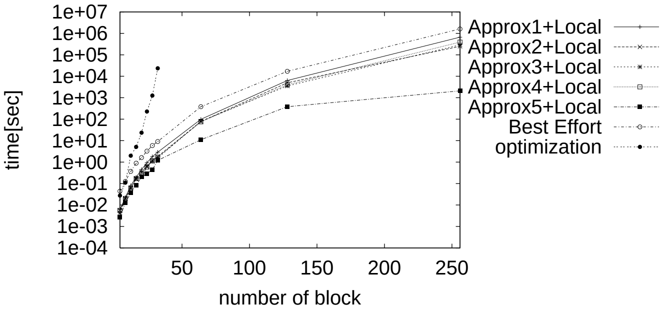

図 20: 近似解の求解時間 2(プロセッサ均一)

4.2 プロセッサ能力が不均一な場合

各パラメータは表 2のとおりである．プロセッサの能力 Cta_iに差がつけてある．Dta_iは全てのプロセッサで同一であるとした．

4.2.1 近似解の精度

$m \leq 32$ で，最適解を 1 とした場合の近似解の精度が図 21，22である．均一な場合と同様に Approx5，Approx5+Local，Best Effort の精度が良く，不均一な場合でもこれらの近似アルゴリズムは有効であると考えられる．均一な場合と比べ全体的に精度が上がっているのは，プロセッサ能力にばらつきがあるため，処理時間がかかるブロックは処理能力が高いプロセッサへ割り当てるという割り当てが可能となっているためである．

表 2: パラメーター覧

名前	値	名前	値
Cta_0	1.0	Cts	20.0
Cta_1	2.0	Dts	0.1
Cta_2	3.0		
Cta_3	4.0		
Dta_i	0.1		

Local は不均一な場合でも有効に働いている．

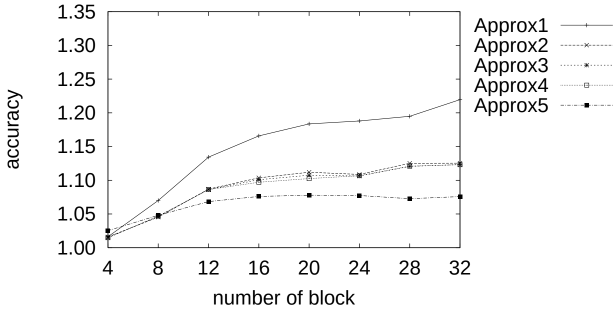

図 21: 近似解の精度 1(プロセッサ不均一)

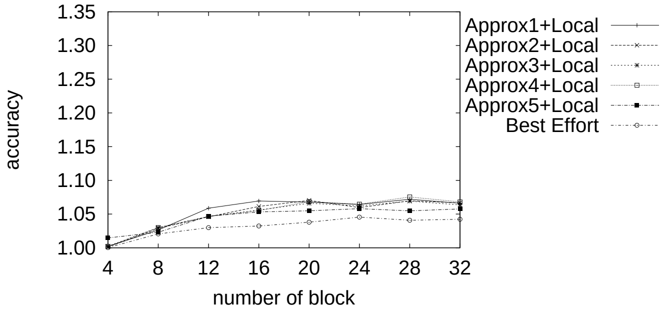

図 22: 近似解の精度 2(プロセッサ不均一)

$m \leq 256$ で，最良の近似解 Best Effort を 1 として相対評価を行った結果が図 23，24である．ブロック数が増加しても均一な場合と同様に Local

は有効に働いている．Approx5，Approx5+Local の精度の高さも変わらない．

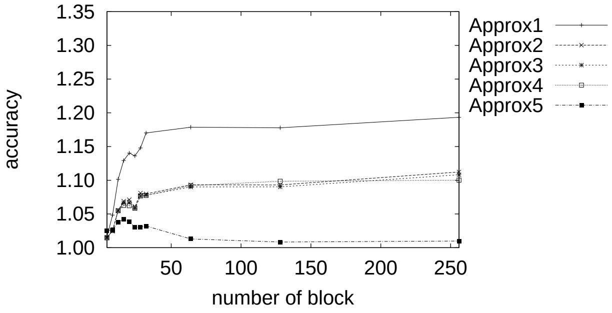

図 23: 近似解の相対評価 1(プロセッサ不均一)

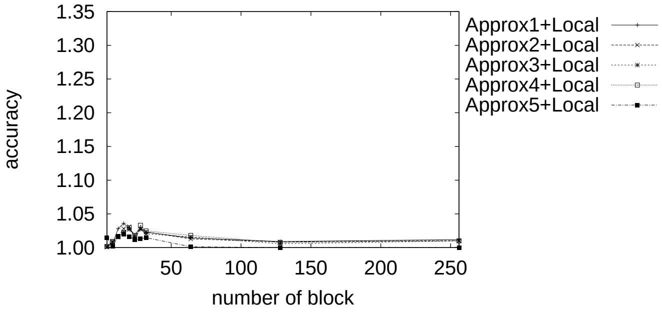

図 24: 近似解の相対評価 2(プロセッサ不均一)

4.2.2 近似解の求解時間

図 25，26は近似解の求解時間である．均一な場合と同様に Approx5 の求解時間が最も短く，Approx5+Local が他の近似アルゴリズムに Local を適用したものよりも短い時間で解を求めることができる．近似解の求解時間は均一な場合と同様に最適解の求解時間と比較することにより，近似解が最適解よりも遥かに短い時間で求まっていることを確認することができる．この結果よりプロセッサ能力に関係なく Approx5，Approx5+Local が有効な近似アルゴリズムであることを確認できた．

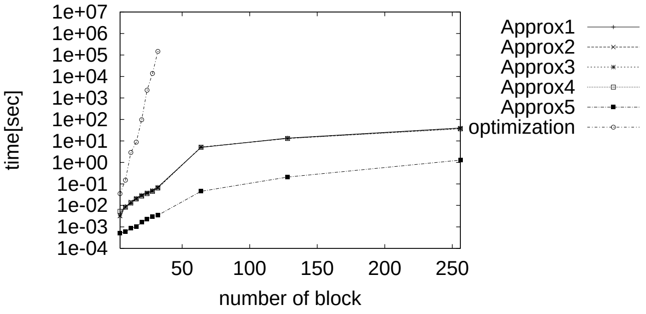

図 25: 近似解の求解時間 1(プロセッサ不均一)

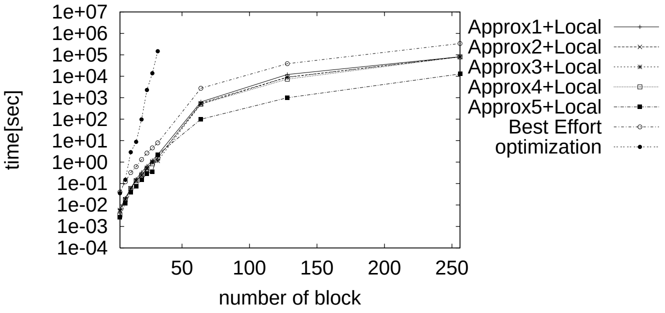

図 26: 近似解の求解時間 2(プロセッサ不均一)

5 おわりに

本研究では並列数値シミュレーションを一種の箱詰め問題としてモデル化し，実行時間を最小化する静的負荷分散法を示した．短時間で精度の良い近似解を与える近似アルゴリズムを提案し，分枝限定法を用いて求めた最適解と比較して近似解の精度を定量的に評価した．本研究で提案した Approx5，Approx5+Local，Best Effort は近似解の精度，求解時間の点で充分有効であると考えられる．ただし，実際の実行時間は実装依存のパラメータによるので，実機上で実際に確認する必要がある．

本研究の箱詰めモデルでは，$m \geq n$ の条件を満たしていても各ブロックの計算量が大きく異なる場合には良い静的負荷分散を行うことができない．例えば図27では，1 つの巨大なブロックが全体のボトルネックとなっ

ている．このような場合は，文献 [2] のようなブロックを分割する手法が必須となる．全ての割り当てが終わった後，ボトルネックとなっているブロックを分割し他のプロセッサに割り当てて実行時間の短縮を図るという割り当て手法を考える必要がある．

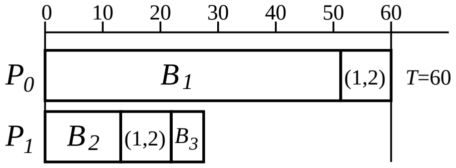

図 27: データ分割が必要になる例

本研究の最終目的は，m ,n 間にどのような関係があっても適切な負荷分散を行うことである．そのためにはブロック分割と箱詰めを同時に行う実用的最適化手法を開発する必要がある．今回の結果を基に，データの自動分割とブロックの割り当てを融合した手法を検討してゆきたい．

参考文献

[1] 笠原博徳: 並列処理技術, コロナ社, chapter 3.3, pp. 148–162 (1991).

[2] 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. 4, pp. 1746–1756 (1998).

[3] Ichikawa, S., Kawai, T. and Shimada, T.: Enhanced Optimization Scheme for Parallel PDE Solver of NSL, 並列処理シンポジウム JSPP ’98, p. 143 (1998).

[4] 茨木俊秀: 組合せ最適化, 産業図書 (1983).

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

[6] 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 (1997).

[7] Hu, T.: Parallel sequencing and assembly line problem, *Oper. Res.*, Vol. 9, pp. 841–848 (1961).

[8] Coffman, E. and Graham, R.: Optimal Scheduling for two-processor systems, *Acta. Informatica*, Vol. 1, pp. 200–213 (1972).

[9] Shirazi, B., Wang, M. and Pathak, G.: Analysis and Evaluation of Heuristic Methods for Static Task Scheduling, *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 222–232 (1990).

[10] Wu, M. and Gajski, D.: Hypertool: A Programming Aid for Message-Passing Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 330–343 (1990).

[11] Sih, G. and Lee, E.: A Compile-time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, pp. 175–187 (1993).

[12] Yang, T. and Gerasoulis, A.: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 9, pp. 951–967 (1994).

[13] Kwok, Y. and Ahmad, I.: Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 5, pp. 506–521 (1996).

[14] Kasahara, H. and Narita, S.: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Transactions on computers*, Vol. c-33, No. 11 (1984).

[15] 飛田高雄, 笠原博徳: 実用的並列最適化マルチプロセッサスケジューリングアルゴリズム PDF/IHS の大規模問題への適用と性能評価, 情報処理学会並列処理シンポジウム JSPP'99 論文集, pp. 31–37 (1998).