＜特集：炉物理研究への PC クラスタの利用・並列計算の実践的アプローチ＞

並列計算の実践的アプローチ

原子燃料工業(株) 巽　雅洋、山本　章夫
tatsumi@nfi.co.jp, a-yama@nfi.co.jp

1 はじめに

さて、ここまでお読みになった皆さんは、並列計算の基礎的な知識を得られているかと思います。また、これまでに並列計算を実際にやった経験のある方もおられるでしょう。そこで、本章ではもう少し実践的な話題を取り扱ってみたいと思います。より具体的なイメージを掴んでもらうために、本章は以下の 3 部構成となっています。

①既存コードの並列化のための実践的アプローチ
②ケーススタディ１：既存コードの並列化（VARIANT コードの並列化）
③ケーススタディ２：並列コードの新規設計と実装(SCOPE2 の開発)

最初に、既存の計算コードを並列化したい場合に何を考慮すべきかについて考えていきましょう。これを踏まえたうえで、筆者の一人が過去におこなった決定論的輸送計算コードの並列化について、最小限の手間で最大限の効果を得るために行った方法について述べていきます。また最後に、より高度な話題として、最初から並列計算を前提として開発した炉心計算コード SCOPE2 の設計・実装について述べていきます。

2 既存コードの並列化のための実践的アプローチ

あなたは計算コードを並列化したいと思ったことがあるでしょうか？　あるとしたらどんなときでしょうか？　漠然ともっと早く計算結果が欲しいとか、純粋に並列計算の技術について知りたいと、そこには様々な動機があることでしょう。しかし、やはりもっとも多い動機は、計算時間の短縮だと思います。特に、既存コードを目の前にして、その計算時間の長さにため息が出てくる場合も少なくないと思います。では、並列化のためには、どういう着目で何を実施すれば良いのでしょうか？　具体的に見ていきましょう。

2.1 計算コードの解析

「敵を知るには、まず己を知る」が戦いの心得であるならば、「並列化するには、まず構造を知る」が並列化の心得です。計算コードは、入力処理や初期化、計算部分、ファイル書き出し、帳票出力等の機能ブロックに分かれていて、各ブロックはサブルーチンや関数[1]やいくつか組み合わさって構成されているでしょう。これらがどのように組み合わさって機能しているか、できるだけ詳細に理解しておく必要があります。この際に、データがどの

[1] Fortran 言語ではサブルーチン、C 言語では関数。

ように受け渡されているかに注意しておくと理解が早いと思います。計算コード解析の古典的な手法としては、プログラムリストを印刷して、ひたすら「コードを読む」という方法が挙げられます。これはもっとも単純ですが実は最も効果的であり、この方法に勝るものは無いと筆者は考えています[2]。 全ての部分の詳細までを「読む」必要はありませんが、全体の流れはできるだけつかんでおいた方が良いでしょう。計算コードを「読む」際には、プログラム解析ツールを用いると便利です。たとえば、floppy[3]や flow[4] は Fortran プログラムの「流れ図」や「クロスリファレンス」を作成してくれます。クロスリファレンスとは、データがどのサブルーチンで使われているかという一覧を作成してくれるものです。また、大型計算機では、Fortran コンパイラがこのような解析情報を出力してくれる場合がありますので、利用すれば良いでしょう。

2.2 ホットスポットを探す

さて、禅問答のようになってしまいますが、そもそも「なぜ並列化したい」のでしょうか？　その答えはおそらく「計算時間がかかるから」でしょう[5]。では、計算コードの「どこ」に時間がかかっているのでしょうか？これを同定することを、プロファイリング(profiling)と呼びます。計算コードをプロファイリングするには、プロファイリング・オプションをコンパイル時に指定してやる必要があります。一般的には、”-p”か”-pg”のどちらかです[6]。注意する点は、リンク時にも同様にプロファイリング・オプションを指定する必要があります。

プロファイリング・オプションを指定されてコンパイルされた計算コードには、どのサブルーチンが呼ばれたかを記録するコードが自動的に挿入され、計算終了時に集計情報をファイルに書き出してくれます。このファイルの内容を、人眼が分かる形式に変換してくれるのが、プロファイラ[7]です。プロファイラは、どのサブルーチンが何回呼ばれたか、一回あたりの実行時間やトータルの実行時間がどの程度を集計して報告してくれます[8]。

プロファイラを用いることで、図 1 のように計算コードの実行時間の内訳を定量化することができます。たいていの場合は、図 1 での(A)のように、計算コードのサブルーチンのうち一つか二つで計算時間がかかっている状況になっていると思います。

---

[2] 異論があるかもしれませんが、少なくとも筆者らが解析する際にはそうするでしょう。上手くプログラムが書けるようになるかは、読んだプログラムの量に（たぶん）比例します。
[3] http://www.netlib.org/floppy/
[4] http://www.netlib.org/flow/
[5]「そこに並列化されていない計算コードがあるから」という答えもあります!?
[6] HP 社製コンパイラは少し特殊で、”+I”や”-G”を用います。
[7] ”prof”あるいは”gprof” が一般的。gprof を用いる場合には、コンパイルオプションとして”-pg”を指定する必要があることが多い。
[8] 中には、GUI で視覚的に表示してくれるものもあります。例えば、Intel の VTune など。

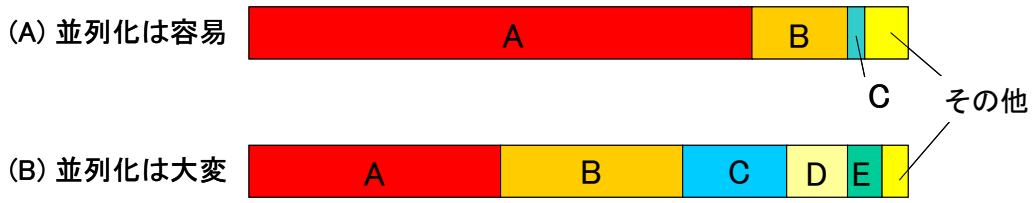


図 1　計算コードのどこで時間がかかっているか？

(A)の場合には、サブルーチン A とサブルーチン B を並列化することで、コード全体の大部分を並列化できることになります。しかしながら、(B)の場合には、(A)の場合と同様の並列効果を得るためには、サブルーチン A から D まで並列化する必要があり、「大変」です。このように、プロファイリングを行うことにより、対象とする計算コードが並列化に向いているかどうかを大まかに判断することができます。ほぼ同一の計算時間であるサブルーチンが数多くある計算コードの場合では、並列化に手間がかかり、また高い並列効率を達成するのは一般的に難しいと考えた方が良いでしょう。

ここでの議論は、「並列化しようとするコードがそもそも並列化に向いているのか？」という疑問について、計算時間の観点から捉えたものです。１箇所あるいは２箇所程度のサブルーチンにおいて計算時間が多く取られている場合、すなわち少数の「ホットスポット」が存在する場合には、高い並列効率を達成することは比較的容易でしょう。しかしながらそうでない場合、すなわちそれほど時間がかからないサブルーチンが多数ある場合には、高い並列効率を求めることは困難でしょう[9]。そういう場合には、少し見方を変えてみると良いかと思います。例えば、少し消極的ですが計算機の速度向上を待つという方法[10]もありますし、コンパイラの自動並列化や自動ベクトル化機能を用いて「お手軽に」高速化するといったまったく別のアプローチもあります。前者は明らかですが、後者もなかなか興味深いテーマであります。これらについては、付録をご参照ください。

2.4　計算アルゴリズムと並列化手法の選択

計算アルゴリズムには、並列計算に「並列化に向かないもの」、「一部制限があるが並列化可能なもの」、「完全に適しているもの」等いろいろあります。これを理解することにより、並列計算に向いていないアルゴリズムに基づく計算コードを一生懸命に並列化しようと努力する…といったことを回避できるでしょう。また、新規に並列計算コードを開発する際には、基礎となる計算アルゴリズムの選定がスムーズに行えるでしょう。では、どういうものが並列計算に「完全に適している」のでしょうか？　一言で言うと、「計算順序の

9 第 1 章のアムダールの法則(Amdahl's law)を参照。
10 ムーアの法則(Moore's law)によると、これまで約 18 ヶ月で半導体の集積度が 2 倍になってきています。これにより、計算速度も約 2 倍になると期待できます。なお、ムーアの法則のトレンドは、今後少なくとも 10 年間は続くと言われています。

依存性がないか、あったとしても最終的に結果に影響がないため無視できる」が、並列計算に適しているアルゴリズムである条件です。では、「計算順序の依存性」とは何でしょうか？　以下では、有限差分計算を例に具体例を幾つか考えてみましょう。

(1) 連続的スウィープを行う計算アルゴリズムの並列化

有限差分法では、あるメッシュの物理量を計算する際には、隣のメッシュからの情報が必要となります。炉物理計算における例として、中性子束を未知数とした解法を考えてみましょう[11]。メッシュ間の相互作用から大局的な中性子束分布を求めます。一般的には反復的に計算を行うわけですが、この際のメッシュ・スウィープ[12]の仕方が問題となります。多くの差分法コードでは、図 2 のようにある方向に連続的にスウィープして各メッシュでの計算をおこないます。たとえば、左から右にスウィープしていく場合、左隣のメッシュにおける最新の計算結果が直ぐに反映されるので、計算効率としては高いものであるといえます[13]。

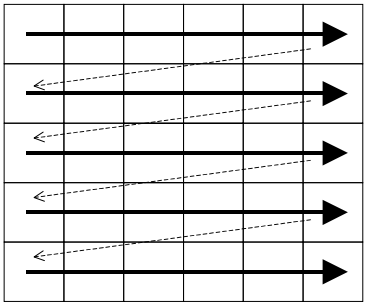


図 2　差分法解放におけるメッシュの連続的スウィープの例

しかしながら、実はこの方法は、並列計算にはあまり向いていないのです。というのも、あるメッシュを計算するためには、ひとつ左のメッシュでの計算が終了している必要があるからです。この前提条件は並列計算の場合には致命的とも言えます。なぜならば、計算順序に暗黙の依存性が発生し、となりのプロセッサの計算が終わらないと自分は計算できないという本末転倒な事態となってしまうからです（図 3）単一プロセッサとまったく同じ計算手順を並列計算に持ち込んだとすると、通信オーバーヘッド等もあることから、単一プロセッサ時よりもより多くの計算時間が必要となることでしょう。

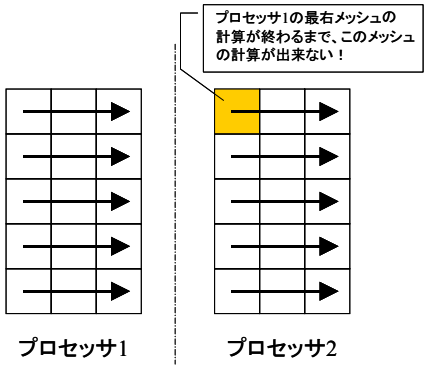


図 3 連続スウィープ方式の並列化における計算順序依存性の問題

これを解決する根本的な方法には、計算順序の依存性を無視するしかありません。つまり、プロセッサ 2 も自分の担当する部分でどんどん計算を進めていき、両プロセッサの計算が終わった時点でプロセッサ

[11]一般的な拡散コード(CITATION 等)や Sn コードはこの範疇に当てはまります。
[12] 計算をある方向にずっと進めていくことを sweep（スウィープ）といいます。ホウキで地面を掃く、あの感じです。
[13] 行列解法におけるガウス・ザイデル法に相当します。

間の境界条件を交換するわけです。こうすることにより無駄な待ち時間はなくなりますが、単一プロセッサ時と同一の収束過程をとりません。このことによる収束性への影響は定量的には判断できませんが、定性的には収束性が若干悪化すると考えられます[14]。また、単一プロセッサと計算結果が微妙に異なる可能性があります[15]。

(2) 色分けされたメッシュでスウィープを行う計算アルゴリズムの並列化

先ほどの例のように連続的にスウィープするのとは対照的なものとして、いわゆる Red/Black スウィープが挙げられます。これは、部分中性子流を未知数とする解法で良く使われています[16]。Red/Black スウィープでは、図 4 のようにメッシュを「赤」と「黒」に色分けします[17]。それぞれの色が互い違いに配置されている点が重要です。例えば、「赤」メッシュの計算時には、隣接する「黒」メッシュからの放出部分中性子流を取り出し、「赤」メッシュへの入射部分中性子流として用います。この入射中性子流から、メッシュ内の応答を計算することにより、最終的に「赤」メッシュからの放出中性子流が計算できます。これを、次のステップである「黒」メッシュを対象とした計算時に用いるわけです。このように、「赤」「黒」メッシュの計算を交互に行うことにより、体系全体の分布を求めるという方法です。この方法は、収束性が先の方法に比べて若干悪化しますが、適当な加速法を用いることによって克服できます。それよりもむしろ、並列計算への親和性の方が重要であると言えます。

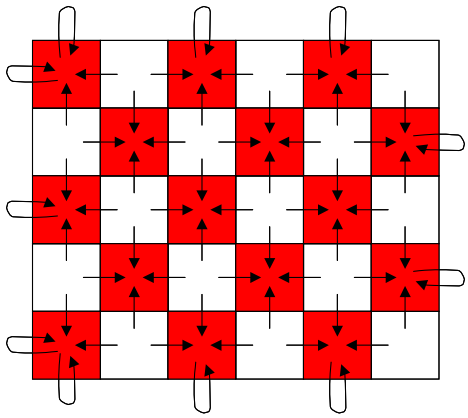


図 4　Red/Black スイープでのメッシュの色分け（矢印は、Red 計算時の中性子流データの参照を示す）

Red/Black スウィープ解法を並列化するのはとても簡単です。まず領域をプロセッサの数だけ分割し、それぞれを各プロセッサに割り当てます。各プロセッサは、自分の担当とする領域でスウィープを行います。その際に、プロセッサの境界（以下、プロセッサ境界と呼ぶことにします）で、プロセッサ間で

---

14 なぜならば、プロセッサ境界において他プロセッサから得られたデータは、いわば「前ステップ」における計算結果であるからです。3 章のポアッソン解法の並列化でも、この影響が出ています。

15 このようなデメリットが受け入れられない場合には、そのコードの並列化をあきらめざるを得ないでしょう。

16 炉心計算コードとしては、原研で開発された MOSLA-Light や、筆者らが開発した SCOPE2 等があります。また、筆者の一人が学生時代に作成した CCCP コード MICA も Red/Black スイープで体系計算を行っています。これは、後に別の学生によって並列化されました。

17 赤と黒は、おそらくルーレットの色分けから来ていると思います。

通信を行って情報を交換しておきます（図 5）。具体的には、次の①から④の手順で順次行い計算します。

① プロセッサ境界における黒メッシュの放出部分中性子流を通信にて交換
② 各プロセッサの赤メッシュの計算を実施
③ プロセッサ境界における赤メッシュの放出部分中性子流を通信にて交換
④ 各プロセッサの黒メッシュの計算を実施

これにより、逐次計算と全く同一の手順で計算を行うことができ、原理的には並列計算でも逐次計算と同一の収束過程を経ることができます[18]。

上記のアルゴリズムは、プロセッサ間通信が内側反復内部に存在するため、頻繁に通信を行う「細粒度アルゴリズム」となっています。そのため、メッシュ計算部分にそれほど時間がかからない「軽い」計算の場合には[19]、並列パフォーマンスはそれほど期待できないでしょう。

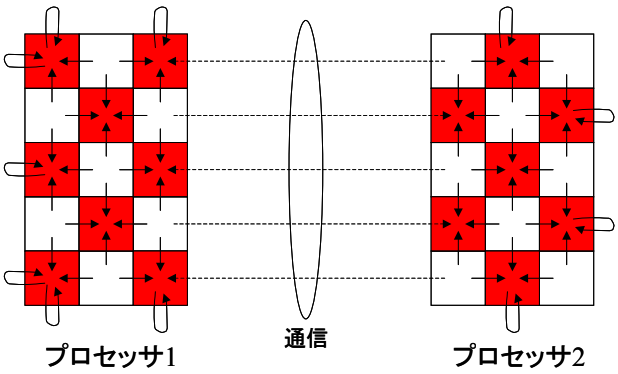


図 5　Red/Black スイープの並列化

2.5　まとめ

既存コードを並列化する際に、考慮すべき事柄について述べてきました。エッセンスを以下にまとめると、次のようになります。

- 基本解法（特にスウィープ法）がどんなものか把握し、その上で対象コードが並列化しやすいかどうかを判断する。
- プログラムの構造を理解し、多くの計算時間が費やされている「ホットスポット」を探す。そして、並列化する意義があるかどうかを判断する。

---

[18] 厳密に言うと、計算順序が変ることによる丸め誤差等で、若干結果が変る可能性があります。これは、被加算変数には倍精度(double)型を用いる等の工夫により解決できます。
[19] 例えば、2 群粗メッシュ拡散計算等

3. ケーススタディ１：既存コードの並列化（VARIANT コードの並列化）

次のステップとして、既存コードの並列化について考えてみましょう。題材としては、Red/Black スウィープに基づく輸送計算コード VARIANT の並列化です[20]。

3.1 ホットスポットを探す

このコードは、入力処理、応答行列計算、中性子輸送計算ソルバー、帳票出力の 4 つの機能ブロックに分類することができます。プロファイラで実行時間の内訳を計測したところ、応答行列計算と中性子束計算ソルバーの部分で計算時間が掛っている事がわかりました。そこで、それらの部分について詳しくコードの「解析」を行っていきました。

3.2 データ構造と処理部分の内容把握

応答行列のデータ構造は比較的簡単なもので、以下のようなものでした。

```
dimension　coef(nntype, kg)
```

つまり、ユニークなノードタイプ[21]毎、エネルギー群毎にデータが格納されていました。また、計算部分は図 6 に示すようなノードタイプによるループ構造となっていました。

```
do ntyp=1, nntype                              ! loop by unique node type
      Do calculation for coefficients
enddo
```

図 6　応答行列計算部のループ構造

一方、輸送計算ソルバー部分データ構造はもう少し複雑でした。軸方向の部分中性子流に関する配列の構造を調べると、図 7 に示すようなテクニックが使われていました。部分中性子流データの保持の仕方として、上部プレーンからの入射中性子流（I1), 上部プレーンへの放出中性子流(O1)、下部プレーンからの入射中性子流(I2)、下部プレーンへの放出中性子流(O2)の順番でデータが並んでいました[22]。

---

[20] このあたりのエピソードは第 1 章をご覧ください。
[21] 燃料の組成等で決定されます。
[22] 最初これが理解できたときは、複雑なパズルが解けたような感じがしたのを今でも覚えています。

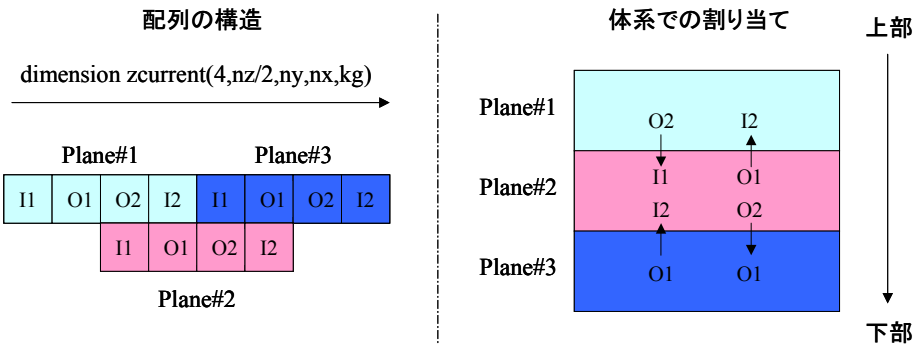


図 7　軸方向の部分中性子流のデータ構造と体系での割り当て

この構造の利点は、データへのポインタを二つだけずらすと次のプレーンにおける情報となることです。図 7 では、あたかも多次元配列であるかのように宣言していますが、実際には 1 次元配列を切り売りする、いわゆる variable dimension によるデータ管理がなされていました。このようにデータポインタへの位置をずらす方法を採用することにより、計算カーネル部分は軸方向の計算と径方向の計算が、図 8 のループ構造のようにで分離されていました[23]。

```
do kz=1, nplane                          ! loop over all axial planes
    Do partial current sweeps for this plane   (X,Y-direction)
enddo
```

図 8　輸送計算カーネルの軸方向ループ構造

3.3 コードの並列化

VARIANT コードの並列化を行う際、短時間で効率よく行うために、次の三つの基本方針を採りました。これらの理由は説明するまでも無いでしょう。

① オリジナルのデータ構造は変更しない
② 計算コードの改変は最小にする
③ 計算結果は並列化によって変化しない

先ほどのようなループ構造を、最小限の労力で並列化する際には、「責任行列」（responsible matrix）[24]と呼ばれるものを導入すると簡単に行えます。これは、担当するプロセッサ番号

[23]「上手い！」とは思いましたが、今後コード開発を行う際には、Variable dimension のテクニックはできれば使いたくないものです。
[24] これは、人から教わった用語ですが、一般的ではない可能性が高いです。

が入った行列のことで、例えば、先ほどの応答行列に関しては次のような感じで値が格納されています。

parameter (nntype=12)
dimension ircproc(nntype)

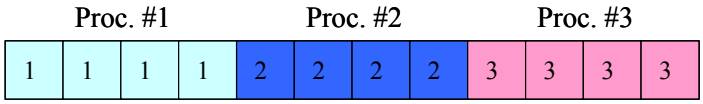


図 9　応答行列計算向けの「責任行列」の内容
(3 プロセッサによる並列計算向け)

この「責任行列」と自分のプロセッサ番号を比較することにより、そのプロセッサが処理を行う「責任」があるかどうかを判断するわけです。

```
do ntyp=1, nntype                          ! loop by unique node type
    if( ircproc(ntyp).eq.myid) then        ! processor is responsible for this node
        Do calculation for coefficients
    endif
enddo
```

図 10　並列化後の応答行列計算ループ(反転部分を追加)

輸送計算カーネル部分の軸方向に関するループは、図 11 のようになります。

```
do kz=1, nplane                    ! loop over all axial planes
  i                                i                r this plane
    Do partial current sweeps for this plane   (X,Y-direction)
  endif
enddo
Exchange partial currents between processors
```

図 11　並列化後の輸送計算カーネルの軸方向ループ(反転部分を追加)

先ほどと同様に、自分の担当プレーンについてのみ計算を行い、その後プロセッサ間で部分中性子流のデータ通信を行う部分を追加しています。参考までに、付録に 8 プレーンを 4 プロセッサに分割した場合について検討した際のメモを示しておきます。

このように、「責任行列」を用いると、オリジナルのデータ構造やプログラム構造をほとんど変えることなく、コードを並列化することができます。なお、責任行列の計算には次の二通りの方法が考えられます。

① マスタープロセッサが代表して作成を行い、それを各プロセッサに分配する。
② 同一のアルゴリズムを用いて、各プロセッサにおいて個別に作成する。

通常は、単純なアルゴリズムによって計算すると思われるので②で十分でしょう[25]。

3.4 まとめ

既存の計算コードを、「手っ取り早く」並列化したい場合には、オリジナルのデータ構造とプログラム・ループ構造を保存することが重要です。つまり、プログラムの改変を最小限にとどめておき（最小限の労力)、並列化を得る訳です（最大の効果)。その際に、「責任行列」というものを用いれば、比較的簡単に並列化できることを紹介しました。

4. ケーススタディ２：並列コードの新規設計と実装(SCOPE2 の開発)

では、いよいよ山場です。これから並列計算コードを開発しようと思っている方に役立つ情報を提供することを目標に進めていきたいと思います。題材として、筆者らが開発した炉心計算コード SCOPE2 を取り上げます。

4.1 初めに

現行の設計計算では、粗メッシュ体系（PWR では集合体幅の半分程度、BWR では集合体幅程度）において 2 群拡散ノード法を適用することが一般的です。一方 SCOPE2 では、燃料格子幅程度のメッシュ体系にて多群・輸送理論・ノード法を適用しています。そのため、計算量・必要メモリ容量は従来に比べて格段に多くなことが予想されたため、開発当初から並列計算機で実行することを前提に設計されています。

話は SCOPE2 コードの開発に着手する前の、2000 年晩秋にさかのぼります。前身である SCOPE コード（以下では、区別するために SCOPE1 と記します）は、汎用解析コードとして、ほぼ完成の域に達していました。そこで、SCOPE1 の開発で得られた経験を元に、次世代炉心設計コードの開発計画を立てました。そのときのグランドデザインは次のようなものでした。

[25]何らかのランダム処理を用いて決定する場合には、①を用いる必要があります。

- 3 次元 pin-by-pin 体系における 9 群 SP3 輸送理論に基づく差分法（後に、同体系における SP3 輸送理論に基づく近代ノード法に格上げ)。
- PWR 炉心設計に必要な全ての機能を搭載（フィードバック、燃焼、ボロンサーチ等、制御棒取り扱いモデル（Cusping モデル）及びグリッドモデルの考慮、リスタート計算、リロード計算（燃料再配置))。
- 並列計算機上で実行し、1 サイクル分の設計燃焼計算を一晩(8 時間)以内に完了すること。

これは、きわめて挑戦的なものでありました。特に心配であったのが計算時間でした。というのも、詳細メッシュ化、多群化、輸送計算の導入等で概算でも 1000 倍程度の計算負荷がかかることが予想されたからです。これを元に、現行手法を用いた場合の計算時間からおおよその計算時間を評価したところ、一晩というのは非常に厳しいということが徐々に明らかになってきました。そこで、並列計算に親和性が高いと同時に、非常に効率よく計算できる輸送ノード法を導出し、１群合成拡散加速等の加速法と組み合わせる方法を開発しました。これにより、高速で安定した計算が可能となった訳です。現在では、3 次元 pin-by-pin 体系における 9 群 SP3 輸送ノード法に基づく設計燃焼計算は、16 台の計算機を用いて数時間程度で完了します。計算理論については本特集の範囲を超えていますので、章末に示す文献をご参照ください。

以下においては、SCOPE2 の並列計算モデル着目し、高効率を達成するためのポイントについて述べていきたいと思います。また、実際の並列パフォーマンスについてもご紹介します。

4.2 並列化モデルと計算体系の取り扱い

SCOPE2 の並列計算モデルを一言で表すと、「2 次元領域分割を用いた細粒度 Red/Black スウィープ並列計算アルゴリズム」となります。実は、第 2 節(2)で述べた Red/Black スウィープアルゴリズムと全く同様のものが採用されています。これにより、計算結果がプロセッサ数等により異なることはありませんが、内側反復の段階で頻繁にプロセッサ間通信を行う必要があり、パフォーマンスの悪化が気になります。しかし、SCOPE1 の開発の段階で、ノンブロッキング通信を用いて効率良く通信を行えば、そこそこの性能向上が出ることが分かっていました。さらに、SCOPE2 では高い並列効率を達成するために、次の工夫を行っています。

・階層型分割による詳細メッシュ体系の構築
・Even-Mesh Red/Black スウィープ法
・通信遅延隠蔽モデルの採用

では、それぞれについて具体的に触れましょう。

4.2.1 階層型分割による詳細メッシュ体系の構築

一つ目の工夫は、体系のメッシュ分割法とその取り扱いに関するものです。3 次元詳細メッシュ体系をプログラム内で取り扱う場合、一般的な方法としては次のような取り扱いが挙げられます。例えば、中性子束を表す flux という配列は、3 次元体系における多群近似では次のように定義できるでしょう。

```
dimension flux(kg, x, y, z)
```

ここでは、これを「方法 1」と呼ぶことにします。方法 1 は、逐次計算しか考慮しないのであれば、シンプルで取り扱いやすいものとなります。また、うまくプログラムすれば長いベクトル長を持つベクトル計算アルゴリズムも適用することができるでしょう。しかし、領域分割法による並列アルゴリズムを適用する場合には、あまり効率が良いとは言えません。その理由は後に譲るとして、ではどうするのが良いのでしょうか？　それは、図 12 のように階層的に体系を細分化していく方法です。つまり、炉心は幾つかの集合体に、集合体は幾つかのブロックに分割されます。そのブロックの中を更に詳細メッシュに分割するといった具合です。このような体系の取り扱いを方法 2 と呼ぶことにします。

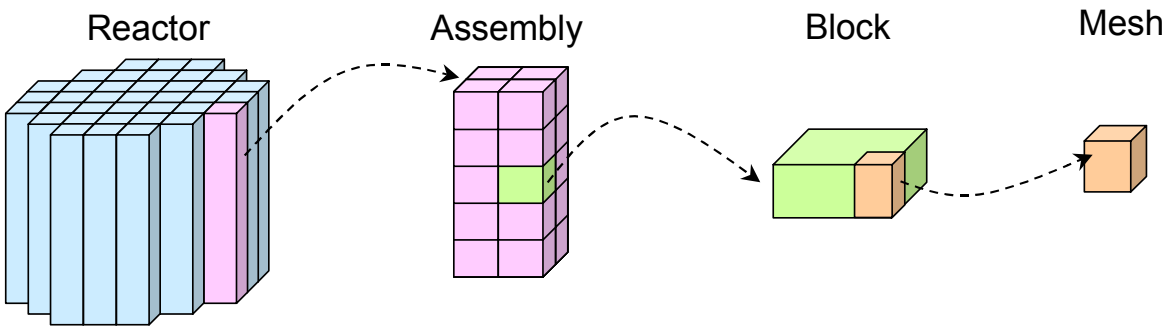


図 12　SCOPE2 における計算体系の階層的分割方法

これをプログラム的に取り扱うにはどうすれば良いのでしょうか？　SCOPE2 は内部的にオブジェクトとして取り扱っています[26]が、例えば Fortran では次のようにすることで解決できるでしょう

```
dimension flux(kg, x, y, blk)
```

[26] SCOPE2 では、テンプレート機能を多用したクラスライブラリを構築し、それを用いてブロックを表現しています。これにより、高速性と保守性を同時に実現しています。

先ほどの例と見比べてみてください。何も変ってないように見えますか？　表面上は変っていなくても、データ確保の観点からは大きく異なっています。実際に領域分割することを考えてみると良く理解できるでしょう。いま、2 次元体系を歪な形で分割する事を考えます[27]。この際に、方法 1 と方法 2 でどの様になるかを見てみましょう。

図 13 をご覧ください。まず、単一プロセッサ時の取り扱いについて考えます。この体系は、x,方向に 500,y 方向に 300 のメッシュに分割され、全体で 1500 の要素を持っています。いま、ある群の中性子束を flux という配列で表現したとすると、方法 1 では、flux(x,y)という形式で参照できます。一方、方法 2 では、まずブロックを決定し、その中のメッシュを指定するという形で flux(x,y,blk)として参照できます。方法 2 では、参照の為のインデックスが一つ増えるため、単一プロセッサ時にはやや煩雑とも言えます[28]。

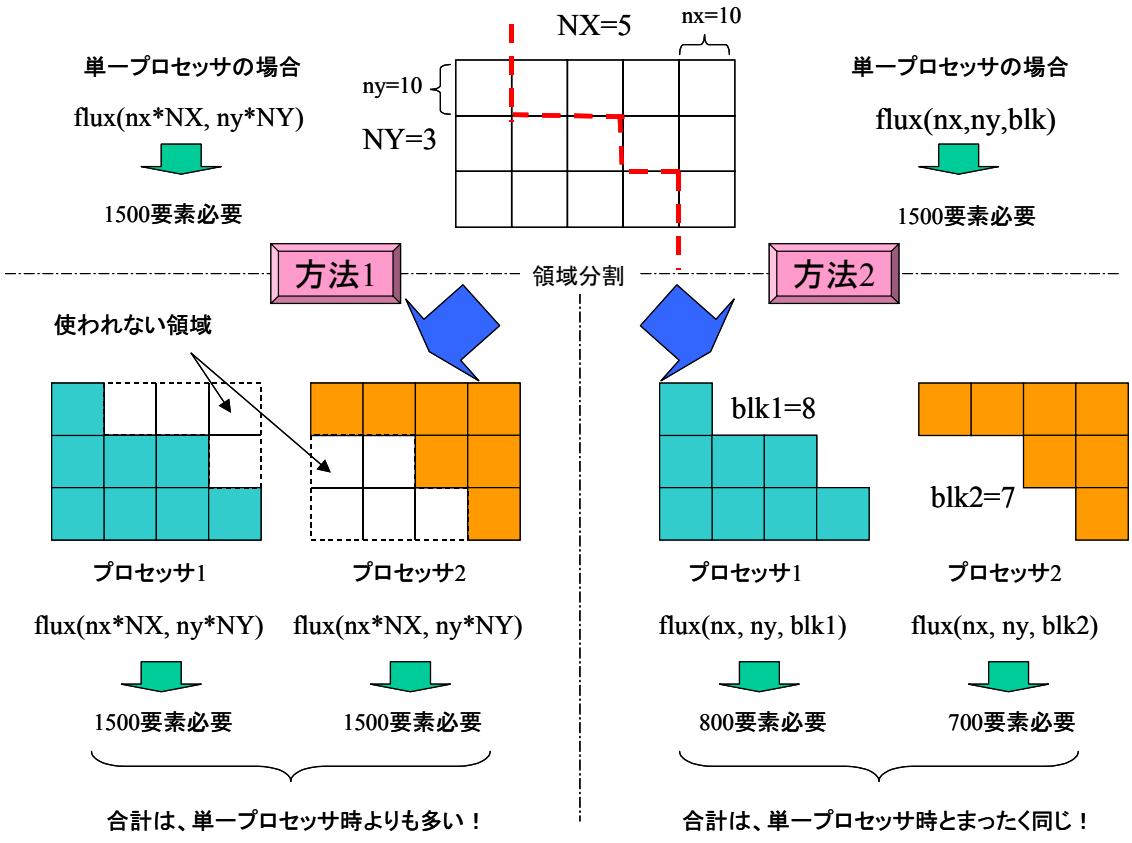


図 13　領域分割に適するデータ構造

[27] 歪な形で分割することを題材に取り上げた理由は、もう少し後になれば理解できます。
[28] ただし単一プロセッサ時にもおいても、炉心外部の領域において計算をスキップさせたい場合では、方法 2 による取り扱いは有効と言えます。

方法 1 では、並列計算時に各プロセッサで用意する配列に無駄な部分ができてしまう可能性が出てきます。最悪の場合では、各プロセッサにおいて全体系を表す配列を用意する必要があります。これは明らかに無駄ですし、計算速度の観点からも不利となります[29]。一方、方法 2 では配列要素も必要最低限ですむことは、図 13 から明らかです。各プロセッサで必要となった配列の合計サイズは、常に全体系を表す配列のサイズに等しくなります。これは、複数プロセッサで計算することによって、単一プロセッサでは解けないような大規模問題も解くことができると言うことを意味します[30]。このように、並列計算時では各プロセッサが最小限のメモリ使用量となるような体系分割法が重要となります。

さて、その気になれば軸方向にも不規則な形状でブロックを分割することができます。しかしながら、SCOPE2 ではブロックの分割は径方向にのみ行う、2 次元領域分割法を採用しています。これは熱水力フィードバック計算において、エンタルピ上昇から減速材温度の計算を、下部ブロックから上部ブロックに向かって計算をする必要があるためです。もし軸方向にも分割すると、計算順序に依存性が発生し並列効率を低下させることになります。

さて、高効率を達成するために、領域分割を行う際に考慮する事柄が幾つかあります。ここで重要な条件を 3 つ挙げておきます。

① 各プロセッサにおける計算量が均等であること
② プロセッサ間の境界部分の面積を最小とすること
③ お互いに接続されるプロセッサ数を最小かつ均等とすること

条件①は、負荷分散の観点から明らかですが、条件③もかなり重要です。これは、「一つのプロセッサだけが多くのプロセッサと通信する」となると、そこの部分だけに処理時間がかかってしまうからです。そこで、SCOPE2 では、各プロセッサにおけるプロセッサ間接続数を、可能な限り均等となるようにしています。このような条件を満たす方法の一つとして、SCOPE2 では径方向を放射状に分割し、体系を「ショートケーキ型」に分割する方法を採用しています[31]。図 14 には、3 ループ型全炉心体系を 16 プロセッサで計算する場合の分割方法の一例を示します。なお、図中において太線で示されている部分が燃料集合体で、その外側は水反射体領域となっています。

---

[29] メモリ参照の連続性が失われると、キャッシュミスによる大きなペナルティを受けます。
[30] Linux ベースの PC クラスタ(IA32 アーキテクチャー)では、ユーザープロセスが使用できる最大メモリ空間は 2GB に限られているため、それ以上のメモリを必要とする問題は単一プロセッサでは取り扱うことができません。
[31] 燃料領域と、反射体領域とでは若干ですが計算負荷に差異があるため、各プロセッサにおいて、燃料領域と反射体領域のブロック数の比が同一となるようにしています。

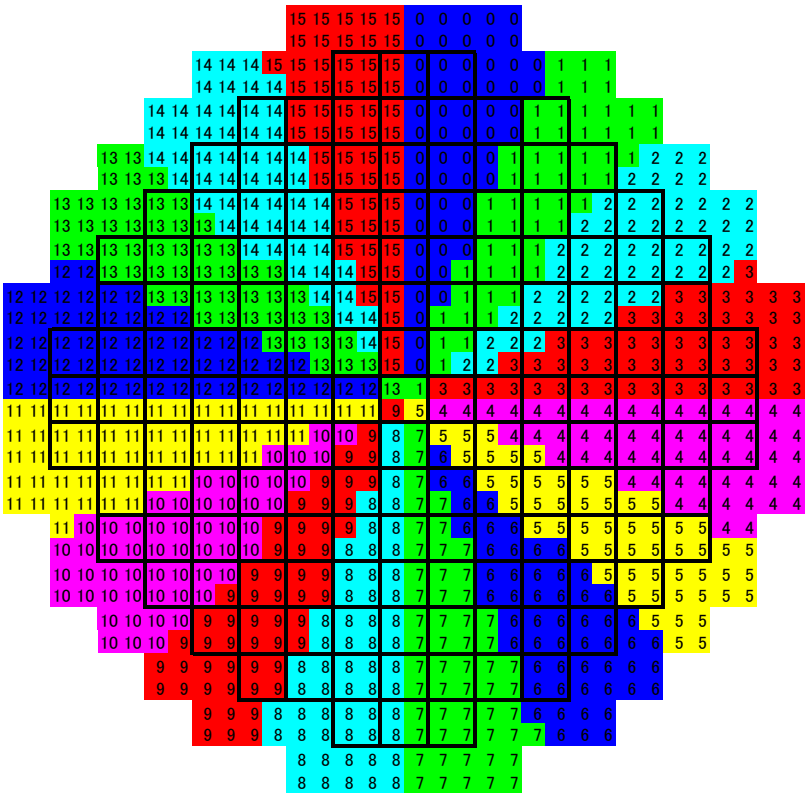


図 14　3 ループ炉心体系の分割方法
(数字は割り当てられるプロセッサ番号)

4.2.2 Even-Mesh Red/Black スウィープ法

二つ目の工夫は、ブロック内の詳細メッシュ分割法に関するものです。まず、結論を先に述べましょう。それは、「ブロック内の詳細メッシュ分割は、各方向について必ず偶数とする」ということです。SCOPE2 では、燃料集合体を径方向に 2x2 のブロックに分割しています。その中を 10x10 程度のメッシュに分割しています。例えば、17x17 型燃料集合体[32]の場合では、図 15 のような分割方法を採用しています。ブロック周辺部では、特に燃料集合体の外周部分では、中性子束が急激に変化することが予想されるため、メッシュ幅を細かくしています。それはさておき、ここで重要なことはメッシュ分割数を偶数としていることです。これを、筆者らは Even-Mesh Red/Black 法と呼んでいます。

[32] 燃料棒が 17 本×17 本の束状にまとめられています。ただし、そのうち 25 本は、制御棒案内管(24 本)と核計装管(1 本)となっています。

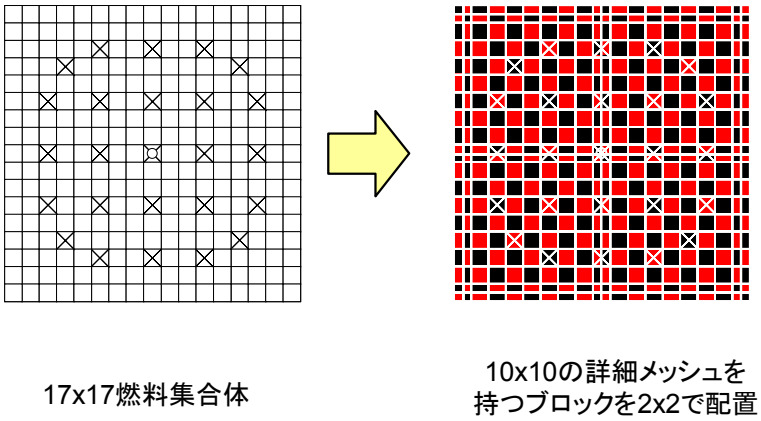


図 15　Even-Mesh Red/Black スウィープ法におけるメッシュ分割例

Even-Mesh Red/Black 法の利点は、ブロック内を偶数に分割することで、ブロックの原点（左上）のメッシュの色が固定できることです。これにより、図 16 に示すように計算や通信アルゴリズムを大幅に単純化できます。というのも、中性子流データのやりとりの部分で様々な場合分けが不要となり[33]、処理を大幅に単純化できるからです。これは最終的に、パフォーマンスとコードの保守性[34]の向上につながります。

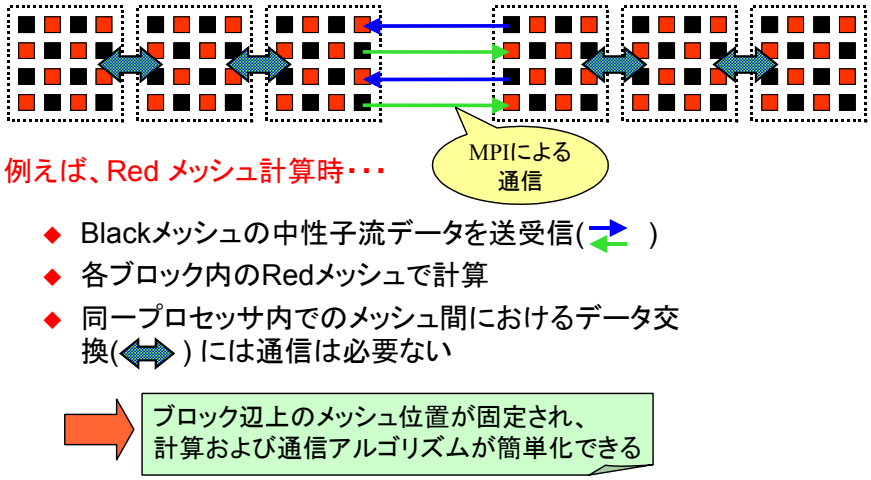


図 16　Even-Mesh Red/Black スウィープ法の利点

[33] SCOPE1 の実装では、この場合分け処理に大変苦労しました。
[34] SCOPE2 では、高速計算と保守性の両方を強く意識して設計・実装しています。

4.2.3 通信遅延隠蔽モデル

三つ目の工夫は、いわゆる”latency-hidden communication algorithm”です。極端に言うと、通信時間を見かけ上無くしてしまう方法です。果たしてそのような事ができるのでしょうか？

少し脱線しますが、何故通信に時間がかかるといけないのでしょうか？　それは、本来は「計算」を加速したいのであって、そのための手段として通信があるわけです。つまり、通信に必要な時間は、本来の「計算」に必要な時間とは全く関係が無く、全く無駄ということになります。それを無くしてやりたいというわけです。

さて、本論に戻ります。結論から述べると、ノンブロッキング通信と計算するブロックの順番を調整することによって、通信時間を見かけ上なくすことができます。基本的なアイデアは、「通信と計算を同時に行う」という極めて単純なものです。図 17 をご覧ください。前節で説明したブロックは、二種類に分類することができます。一つはプロセッサ境界に面しており、計算する際に他プロセッサとの通信が必要なもの。もう一つは、同一プロセッサ内のデータのみで計算できるものです。

まず初めに、データ通信が必要なブロックに関して着目します。交換するべきデータの準備を行い、ノンブロッキング通信命令を発行します（図 17 のステップ 1)。ノンブロッキング通信命令の発行後、制御は直ちに戻ってきます[35]ので、その後計算を開始します。この際、通信は平行して行われると考えてください[36]。このときは、通信が必要ないブロックに対して計算を行っていきます。Even-Mesh Red/Black スウィープ法では、任意のブロックから計算を行えるということを利用しています(同ステップ 2)。 通常は、計算時間の方が長いので不要かもしれませんが、念のため通信が終了していることを確認します（同ステップ 3)。最後に、先ほどプロセッサ間で交換したデータを用いて、プロセッサ境界におけるブロックにおいて計算を行います（同ステップ 4)。

このような方法をとることにより、通信に必要な処理時間を最小限にすることができます。実際には、通信時間は完全には無くなりませんが、ある程度の効果は期待できます。ただし、プロセッサ数が多くなると、計算順序の調整等のオーバーヘッドにより、その効果は小さくなったり、逆に普通に処理するよりも時間が必要となったりする場合がありますので、注意深く実装する必要があります。なお、SCOPE2 においては、最大で数%並列効率が向上することを確認しています[37]。

---

[35] これが「ノンブロッキング」と呼ばれる所以です。

[36]筆者も詳細は分かりませんが、ネットワーク機器が CPU の邪魔をすることなしに通信していると考えれば良いようです。専用の並列計算機では、通信用専用のプロセッサが搭載されています。

[37] たかが数%という意見もありますが、こういう地道な努力の積み重ねで高効率が達成できるわけです。

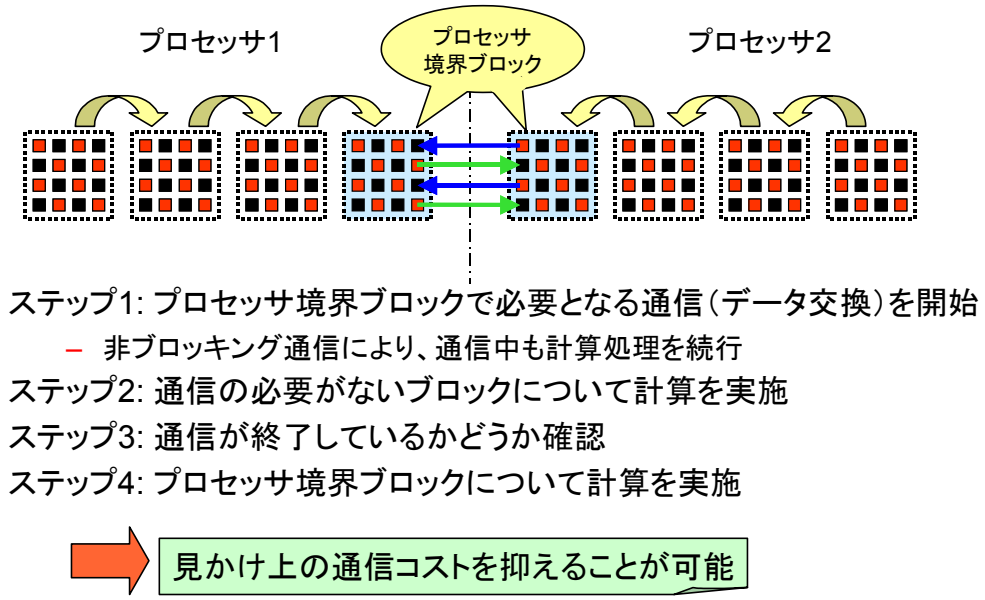


図 17　Even-Mesh Red/Black スウィープ法の通信遅延低減への応用

4.3 SCOPE2 計算の全体フロー

SCOPE2 では、幾つかの機能ブロックに分割されています。具体的には、以下のものが挙げられます。

① 初期化とユーザ入力等の処理（領域分割等も含む）
② 詳細メッシュ計算
③ 1 群粗メッシュ拡散計算による加速
④ 熱水力フィードバック計算（ボロンフィードバック計算を含む）
⑤ 断面積合成
⑥ 燃焼計算
⑦ リスタートファイルの読み書き

これらの中で、③から⑤の部分で大部分の計算時間が費やされています。その部分の計算フローについて詳しく見てみましょう。図 18 をご覧下さい。最も外側に、ボロン反復に関する反復計算ループがあります。その中には、詳細メッシュ体系に関する外側反復のループがあります。そこでは、まずフィードバック計算により群定数を決定し、中性子源を更新します。内側反復により詳細メッシュ体系での中性子束を決定します。全てのエネルギー群について内側反復計算を行った後、求まった中性子束を用いて 1 群の粗メッシュ体系向け平均群定数と補正係数を計算し、1 群粗メッシュ拡散計算を行います。得られた粗メッシュ解を用いて詳細メッシュ解を加速し、補正された中性子束を用いて体系の実効増倍率

を計算します。これらの一連の計算を、実効増倍率、臨界ボロン濃度、出力分布が収束するまで繰り返し行います。

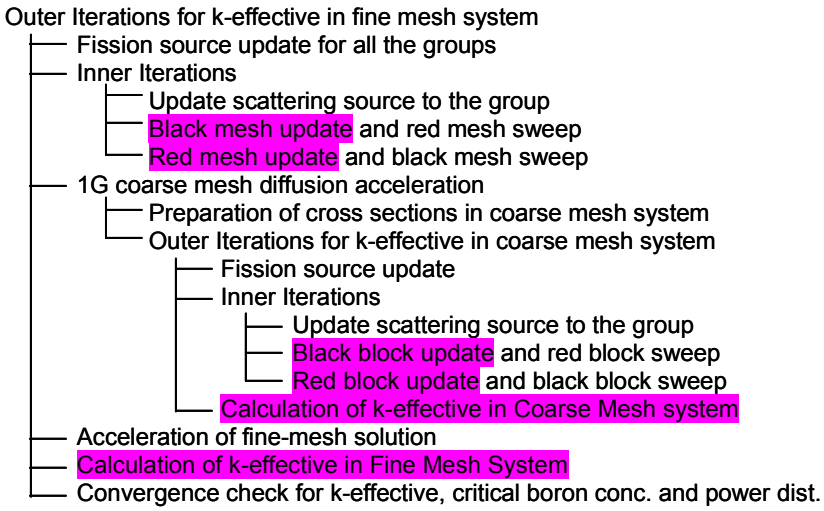


図 18　SCOPE2 における反復計算の処理フロー

この一連の計算の中で通信が必要となるのは、内側反復内における部分中性子流の交換部分と、実効増倍率の計算部分です。これらは、詳細メッシュ体系と粗メッシュ体系の両方に当てはまります。これらについては、頻繁にプロセッサ間通信を行うため、高効率を達成するために、これまでに述べたように様々な工夫をこらしています。なお、SCOPE2 では、詳細メッシュと粗メッシュで処理コードを共有化しており、計算の高速性、通信の効率性、コードの保守性の向上に寄与しています[38]。

実効断面積の合成部分や燃焼計算部分においては、プロセッサ間の情報交換が必要なく、各ブロックは完全に独立して処理を実施することができるため、並列効率はほぼ 100%となります[39]。それから意外に忘れられがちなのが、ファイルアクセスです。SCOPE2 では、1 ステート当たりのリスタートファイルが数百 MB と非常に巨大な為、その取り扱いにも工夫を凝らしています。具体的には、MPI-2 規格の一部で並列ファイルアクセスの為の API である MPI-I/O を用いています。

[38] C++のテンプレート機能を活用したオブジェクトクラス設計となっています。
[39] 各プロセッサが取り扱う問題サイズが小さくなるにつれ、メモリ参照の局所化によるキャッシュの利用効率の向上から、100%以上の効率が出るいわゆる「スーパーリニアー」も起こる可能性があります。

4.4　NFI における PC クラスタ

NFI では、1998 年の春頃から PC クラスタの運用を行っています。この話について少し述べたいと思います。

1998 年当時、クラスタを組める願ってもないチャンスがやってきました。事務用に使っていた PC がリース切れになり、新しい PC システムとの入れ替えがあったのです。リース会社に返却するまでに約 1 ヶ月間の猶予がありました。その間に 12 台の PC をネットワークで接続し、クラスタに仕立て上げたのです。その当時の様子を図 19 に示します。

12 台の PC で構成されるクラスタ

各 PC の構成
- プロセッサ: Pentium-90 MHz
- メモリ: 128M バイト
- カーネル: Linux-2.0
- ネットワーク: Ethernet(10Mbps)

ネットワークスイッチ
- Ethernet スイッチ

図 19　NFI における 第 1 世代 PC クラスタ

そのスペックは、リースアップしたものですから、当時としても貧弱なものでした。しかし、並列計算の実験環境としては十分なものでした。なんと言っても、初期投資額がほとんど不要[40]だったのが最大のメリットでした。このクラスタを用いて、SCOPE1 の並列計算に関する実験を行っていました。結局、格安でリースの延長ができることがわかり、約 1 年間このシステムを用いました。このシステムの構築の際には、一台だけリファレンスシステムを作成し、そのディスクイメージを dd コマンドでクローニングする[41]という、かなり原始的な方法で設定を行いました。

99 年頃、大きな体系を取り扱うにつれ、もっと速い計算機が必要と感じられてきました。そこで、当時ほぼ最速のシステムを、パーツ毎に購入し、自分たちで組み立てました[42]。途

[40] 購入したのは、Ethernet スイッチぐらいでした。当時は、10Mbps のスイッチでも結構しましたが、潰しが効くということで認めてもらいました。

[41] dd if=/dev/had of=/dev/hdc bs=512　　といった方法で、ディスクのクローニングを行うことができます。

[42] CPU、マザーボード、メモリ等を別々に注文して、全て組み立てました。

中何回かアップグレードし、最終的に図 20 に示すクラスタとなりました。このクラスタの構築の際にも、当初は dd コマンドを用いたハードディスクのクローニングを行って、クライアントのコピーを作成していました。しかし、途中から LUI[43]と呼ばれるネットワークインストール・ユーティリティーを用いた完全自動インストールが行えるようにしましたことで、システム構成の変更等は劇的に楽になりました。

8 台のクライアントと 1 台のサーバで構成される
PC クラスタ
サーバ:
　プロセッサ Pentium-III　1GHz
　メモリ: 1024M バイト
各 PC の構成
　プロセッサ: Pentium-III　450 MHz×2
　メモリ: 1024M バイト
　カーネル: Linux-2.2 (smp)
　ネットワーク: Fast Ethernet(100Mbps)
ネットワークスイッチ
　Fast Ethernet スイッチ

図 20　　NFI における第 2 世代の PC クラスタの構成

さて、第 2 世代のクラスタも約 3 年間の間にすっかり陳腐化してしまいました。ちょうどその頃、SCOPE2 の開発もいよいよ佳境に入り、そろそろ実炉心体系を対象とした検証計算の段階に入ってきました。こうなると、大規模なクラスタが必要だということで、ほぼ最新・最速のクラスタを構築することとなりました[44]。この第 3 世代のクラスタは、第 2 世代のクラスタを取り込み、合計で 35 プロセッサの構成となっています。インストール作業も LUI により、完全自動で行い、数日のうちに完成しました。とあるセミナーで聞いた GNF-J の中氏のクラスタ構築に関する講演[45]に触発され、ラックに収めてすっきりと配線しました。これらのクラスタは、24 時間空調の効いたマシンルームで稼働していますが、

[43] 現在はもうメンテナンスされていないようです。次世代版としては、OSCAR というものが開発されています。(http://oscar.sourceforge.net/)
[44] 実際には、コストパフォーマンスを考慮し、少しだけ CPU のクロックスピードが低いものを選択しました。（当時最速は 2.4GHz だったが、2.0GHz を採用した。CPU の価格は 2 倍程度違ったと記憶しています。）
[45] 第 2 章を参照。

それでもかなりの熱量が発生します[46]。また、電力使用量も結構なもので、クラスタの増設に際して電源工事も行う必要がありました。第 2 章でも触れられていますが、排熱と電源確保は十分に検討してからクラスタの導入を行う必要があると思います。

<table>
<tr>
<td>Pentium-3　450MHz × 16
Pentium-4　2GHz × (16+3)
Pentium-4　2.4GHz×8
各計算ノードは FastEthernet(100Mbps)スイッチで接続。スイッチとファイルサーバは、GigabitEthernet(1000Mbps)で接続</td>
<td>  </td>
</tr>
<tr>
<td>  </td>
<td>  </td>
</tr>
</table>

図 21　　第 3 世代の PC クラスタの構成

4.5 SCOPE2 のパフォーマンス

本節では、SCOPE2 の並列計算のパフォーマンスについて述べていきます。

3 ループ PWR 炉心の解析を対象に、二通りのケースを設定しました。一つは比較的小規模な例として、図 21 に示すような 1/4 炉心体系で 170×170×26 メッシュを設定しています。この程度だと、なんとか 1 プロセッサで計算ができるため[47]、逐次計算に対するスケーラビリティーのチェックに用いました。二つ目は、図 21 の 4 倍規模である全炉心体系で、全メッシュ数は 340×340×26 となります。ちなみに、全炉心体系において 9 群輸送ノー

[46] 意外に PC の電源ユニットからの発熱が大きく、困りものです。
[47] 必要としたメモリは 1GB 強でした。クラスタのうち、2GB のメモリを搭載しているマシンで実行しました。

ド法を適用した場合に、全体で 4.5GB 程度のメモリが必要だったため、6 台で計算した場合を基準としてスケーラビリティーの測定を行いました。

計算は、予測子/修正子法[48]に基づく 12 ステップの燃焼計算です。従って、22 回の臨界ボロン濃度サーチ計算を行っています。図 22 にプロセッサ数と計算時間の関係を示します。プロセッサ数の増加に伴って、ほぼ全てのセクションにおいて計算時間が減少しています。ファイル書き込みも、MPI-I/O による並列アクセスの効果が現れ、処理時間が減少しています。なお、全炉心体系においては、ディスク書き込み速度が律速となり、プロセッサ数の増加に対して横ばいになっていますが十分なパフォーマンスが出ています[49]。

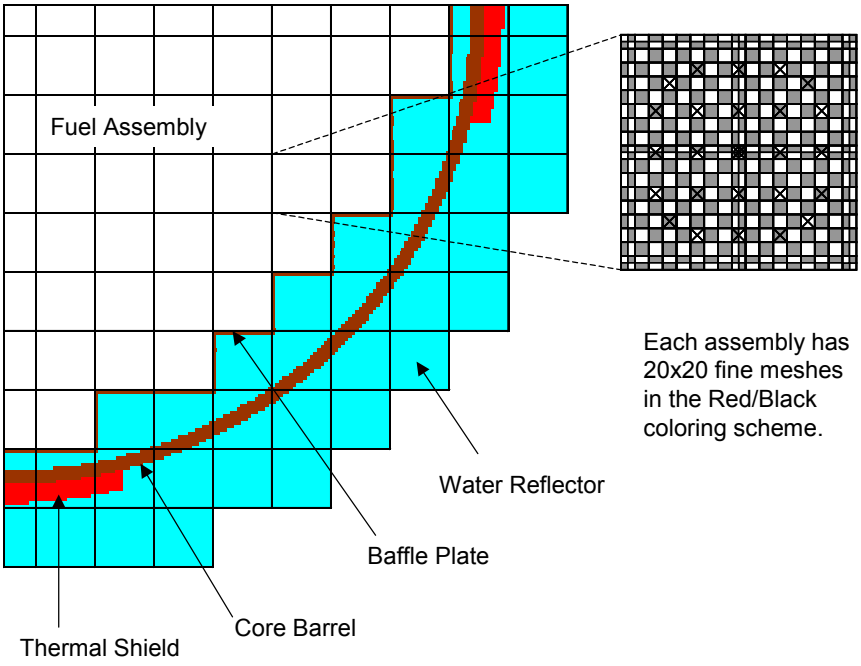


図 21　パフォーマンス測定時における計算体系(1/4 炉心)

[48] Predictor/Corrector 法

[49] ファイルサーバに接続されている RAID5 ディスクアレイへの書き込み速度が律速となっているようです。（書き込み速度は約 25MB/sec 程度）

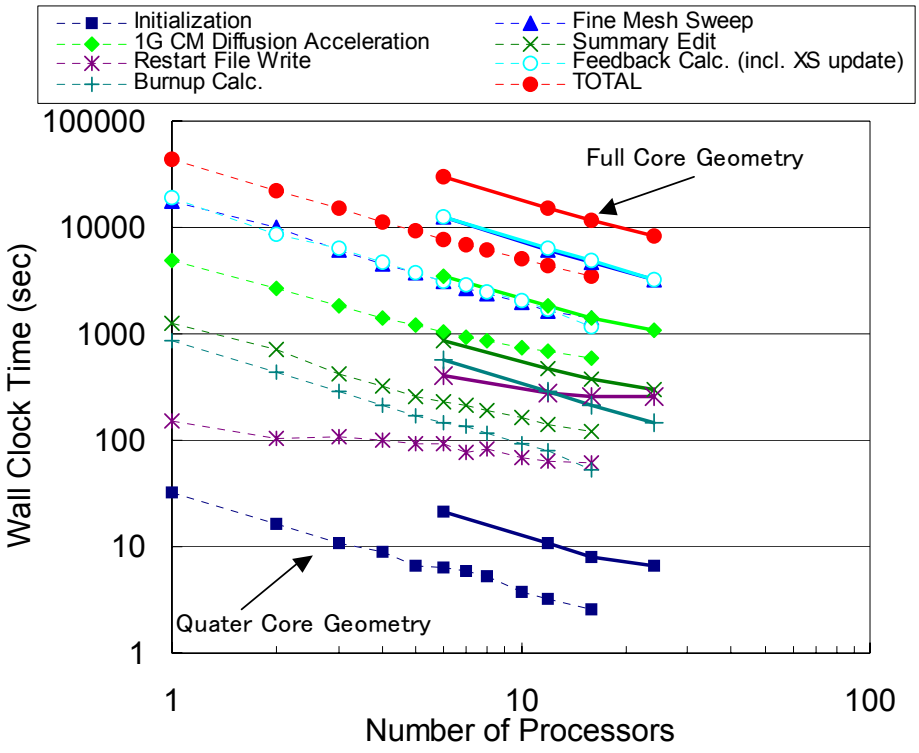


図 22　プロセッサ数の増加と処理時間の減少

さて、全体のパフォーマンスを決定づける重要部分が、90%以上計算時間を費やす詳細メッシュ計算部分とフィードバック計算と断面積合成部分です。これらは図 23 に示すように、ほぼ完全なスピードアップとなっています。にもかかわらず、全体のスピードアップは完全には線形ではありません。これは、約 10%程度の処理時間を費やす 1 群拡散加速計算の線形性が悪いためです。1 群拡散加速計算は「軽い」計算のため、計算時間に対する通信時間の割合が大きくなるため、並列性能が悪化するためです。ただし、この程度の悪化では特に問題とはならないでしょう。外挿による評価では、64 台使用時

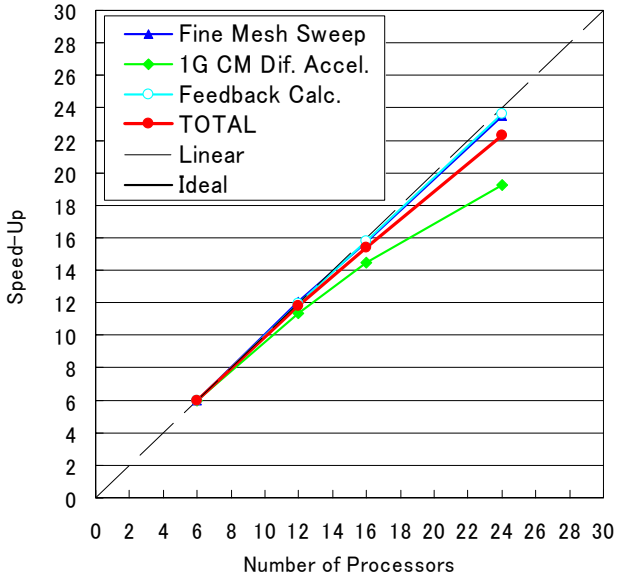


図 23　全炉心体系での設計燃焼計算におけるスピードアップ

に 60 倍のスピードアップが期待できると予測しています。

全体の計算時間としては、3 次元 pin-by-pin 体系において、9 群輸送ノード法計算に基づく設計燃焼計算が、24 台を用いた場合に約 2.2 時間で完了しています。従って、設計時に打ち立てた目標は十分にクリアできました。ただし、今後このモデルを過渡解析に拡張することを考えると、まだまだ高速化が必要であると考えています。

4.6　まとめ

SCOPE2 は、並列計算においても逐次計算とまったく同一の計算結果を与えます。これは、QA 上の観点から極めて重要なことです。これは内側反復内におけるプロセッサ間通信を行う、細粒度の並列計算アルゴリズムにより実現されています。通常、細流度アルゴリズムでは高いパフォーマンスを達成することは難しいのですが、SCOPE2 の場合には計算負荷が適度に高いため、高い並列効率を達成することができました。

5 おわりに

さて、並列計算の実際ということで 3 部構成にて述べてきましたが、いかがでしたでしょうか？　特に、最後の部分に関しては、これから本格的に並列計算コードを書いてみたい方向けに、やや詳しく説明したつもりです。本章が、今後の並列コード開発に役立てば幸いです。

SCOPE1, SCOPE2 関連論文

1. M. Tatsumi and A. Yamamoto, “Object-Oriented Three-Dimensional Fine-Mesh Transport Calculation on Parallel/Distributed Environments for Advanced Reactor Core Analyses,” *Nucl. Sci. Eng.*, 190, 141 (2002).
2. M. Tatsumi and A. Yamamoto, “SCOPE2: Object-Oriented Parallel Code for Multi-Group Diffusion/Transport Calculations in Three-Dimensional Fine-Mesh Reactor Core Geometry,” *Proc. Int. Conf. on Physics of Reactors (PHYSOR2002)*, Seoul, Korea, Oct., 2002, p. 12A-01 (2002).
3. M. Tatsumi and A. Yamamoto, “Advanced PWR Core Calculation Based on Multi-group Nodal-transport Method in Three-dimensional Pin-by-Pin Geometry,” submitted to *J. Nucl. Sci. Technol.*
4. M. Tatsumi and A. Yamamoto, “Performance of a Fine-Grained Parallel Model for Multi-Group Nodal-Transport Calculation in Three-Dimensional Pin-by-Pin Reactor Geometry,” submitted to *Int. Conf. Supercomputing in Nuclear Application (SNA’2003)*.

付録

1. データ・パラレル

データ・パラレルとは、ループ内のデータ構造に着目した並列化方法で、明示的なプロセッサ間の通信というものはありません。もう少し踏み込んで言うと、共有メモリ型のマルチプロセッサ計算機(SMP)で有効な並列化手法です。たとえば、次の do ループを考えて見ましょう。

```
do   i=1, 10000
    x(i) = y(i) + z(i)
enddo
```

この場合は、ループをいくつかのスレッド[50]に分割し、各々のスレッドにおいて個別の範囲を担当して処理すれば良いでしょう。たとえば、2 プロセッサで処理する場合には、スレッド 1 では 1 から 5000 の範囲を、スレッド 2 では 5001 から 10000 の範囲について担当するといった感じです。スレッド 1 はプロセッサ 1 で、スレッド 2 はプロセッサ 2 で同時に動作すると[51]、処理時間は半分で済みます。ひとつのループをいくつかのスレッドに分割する、いわゆるマルチスレッド化は、コンパイラが自動的に行ったり[52]、指示命令(directive)をソースコードに埋め込んでコンパイラに半自動的に行うことができます[53]。

このような、ループ内部のデータ操作を分割し、マルチスレッド化によって並列化する方法を「データ・パラレル」と呼んでいます。データ・パラレルによる高速化は、ベクトル化されたプログラムに向いています。というのも、ベクトル化されたプログラムは先ほどのようなループ構造が多数見られるからです。また、ベクトル長が長い場合には、データ分割された後のスレッドあたりの仕事量が大きくなるため、パフォーマンスが出やすくなります。

ループ構造に気をつけることは、シングルプロセッサシステムでも有益です。特に、Pentium-4 に代表される IA32 アーキテクチャーでは、SSE(Streaming SIMD Extension)

---

50 タスクの最小構成単位。ジョブ（プロセス）はひとつ以上のスレッドから構成される。複数のプロセッサがある場合、マルチスレッド化されているプログラムはより短い時間で処理できる可能性がある。

51 実際にはスレッド 1 が必ずしもプロセッサ 1 で動作する保障はありません。ある瞬間にはプロセッサ 2 で動作するかもしれません。一部の OS では、スレッドとプロセッサの対応付けが可能だそうです。このことはキャッシュの有効利用の観点から有益ですがそれほど気にする必要はないでしょう。

52 PGI Fortran や Intel Fortran では自動的にループを複数スレッドに分割することが出来ます。

53 OpenMP と呼ばれる規格があります。PGI Fortran や Intel Fortran では OpenMP によるスレッド化が可能です。

や Hyper-Threading と呼ばれる技術が導入され、これらとデータパラレルモデルは非常に親和性が高いからです。これらの詳細は本論とかけ離れるため、これ以上の議論はここでは行わないことにします[54]。

2. 参考情報

(a) 並列ライブラリ

並列化が行われているライブラリは幾つか開発されています。これから並列計算コードを「できるだけ手間を省いて」作りたいという場合、一度チェックしてみることをおすすめします。

ScaLAPACK: http://www.netlib.org/scalapack/scalapack_home.html
PETSc: http://www-unix.mcs.anl.gov/petsc/petsc.html
POOMA: http://www.acl.lanl.gov/POOMA/

(b) 教科書等

・PC クラスタ超入門　http://mikilab.doshisha.ac.jp/dia/smpp/cluster2000/
同志社大学の三木先生が主催する、超並列計算研究会で行われた講習会のテキスト。内容は多岐にわたり、かつ非常に充実している必見の資料。

・W. Gropp 他、”Using MPI”, “Using MPI-2”　MIT Press
筆者らは MPI 標準策定の中心メンバー。MPI の基礎から応用までこの 2 冊で全て分かる。MPI 使いのバイブル。”Using MPI-2”の邦訳が最近出版されたようだが、できれば原書で読む事をお勧めする。

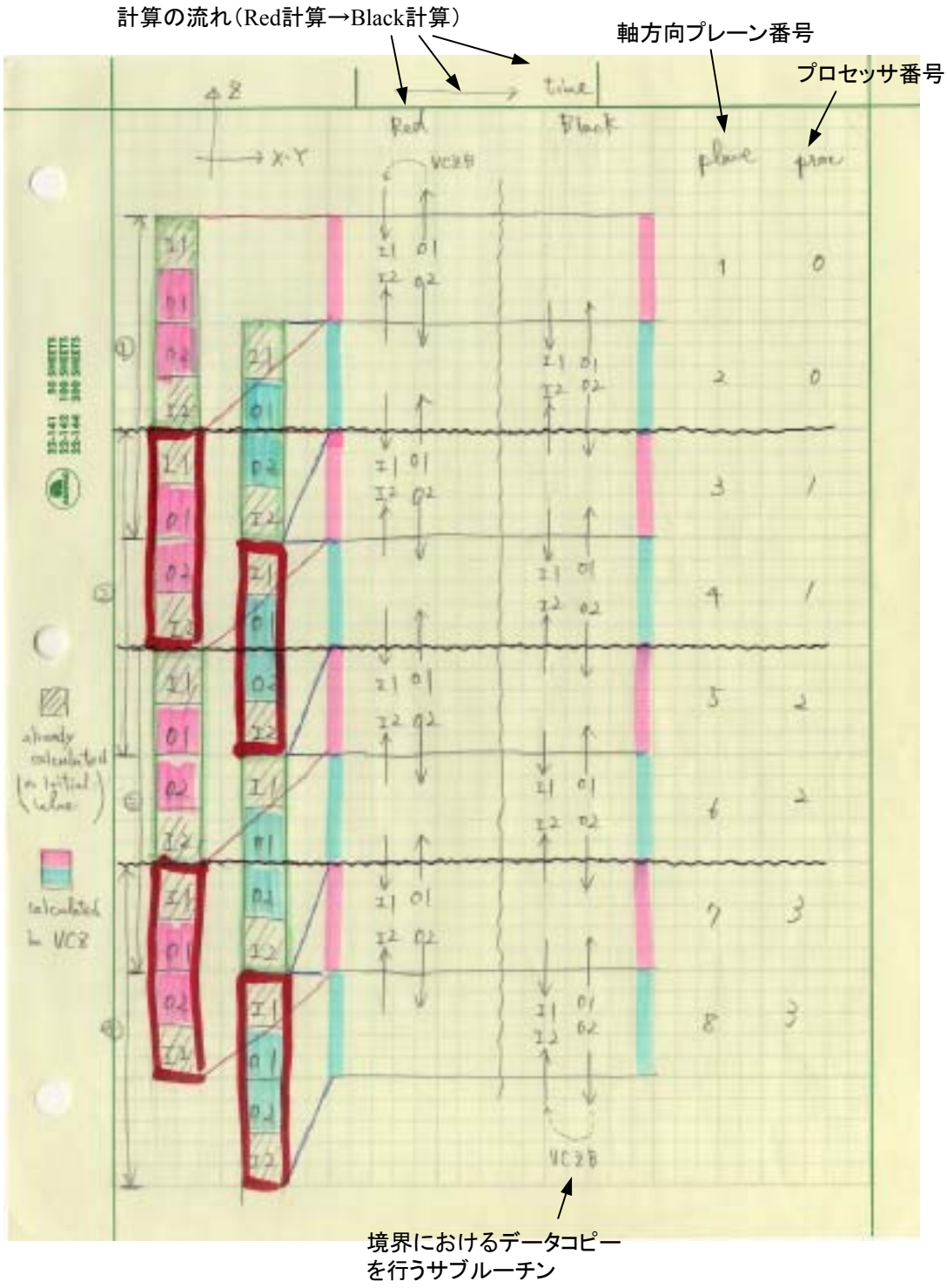
・ Ian Foster, Designing and Building Parallel Programs, Addison Wesley, 1995.
並列計算の基礎から応用に至るまでわかりやすく書かれている。オンライン文書が公開されている。http://www-unix.mcs.anl.gov/dbpp/

・ OpenMP に関するチュートリアル http://phase.etl.go.jp/Omni/openmp-tutorial.pdf
新情報処理開発機構つくば研究センタの佐藤三久氏によるチュートリアル。これで、OpenMP の概要が理解できるはず。

---

[54] 個人的には非常に重要だと認識していますが、かなりマニアックな領域になってきます。

3. VARIANT コードの中性子輸送カーネルの並列化におけるデータ構造



軸方向には、Red プレーン(奇数番号)と、Black プレーン（偶数番号）があり、実際には交互に計算されている。