チューニング技法入門
2012年6月1日版

理化学研究所　情報基盤センター
青山幸也

はじめに

先日、あるベンチマークプログラムで、ソルバー（連立一次方程式を解く）の部分を 2 行ほど修正したところ、計算時間が 8 時間からわずか 6 分に短縮され、パフォーマンスが何と 80 倍も向上しました。これは極端な例ですが、一般に、同じ計算をするプログラムでも、コーディングの仕方によってパフォーマンスが大きく異なることがあります。

既存のプログラムを修正してパフォーマンスを向上させることを、チューニングと呼びます。チューニングによってパフォーマンスが仮に 3 割向上したとすれば、それは、3 割性能の高いマシンを使用しているのと同じことになるわけです。

私は日ごろ、お客様の作成したプログラムを見る機会がありますが、チューニングの余地があるプログラムが多く見受けられます。この原因の一つとして、チューニングに関する適切なガイドがないことが挙げられます。メーカーが提供する Fortran のマニュアルにもチューニング技法の説明がありますが、アセンブラコードのレベルで検討しなければならない技法や、チューニングを行うことによってプログラムが見にくくなるような技法なども紹介されており、特に初・中級のユーザーにとっては難しすぎると言えます。

この現状を踏まえ、本ガイドでは、ワークステーションを使用する初・中級ユーザーを対象とし、パフォーマンスを向上させるための基本的なチューニング技法をまとめてみました。これらのチューニング技法は、勿論、プログラムを新規に作成する場合にも適用することができます。

なお、本書の内容に関する責任は負いかねますのでご了承願います。

1995 年　青山　幸也

本書は主にスカラー計算機を対象としたチューニングのガイドで、初版を作成してから早いもので約 8 年経ちました。初版は筆者がコンピュータ会社に勤務していたときにユーザー教育用として作成しました。私事で恐縮ですが、2002 年秋に転職したのを機に、書名を改め、特定のマシンに固有の部分を削除し、その他の部分も加筆修正する事にしました。このような経緯のため、内容に一部整合性がとれていない部分があるかと思いますが、今後少しずつ改善していきたいと思います。

2003 年　青山　幸也

当資料は、日本アイ・ビー・エム株式会社の承諾を得て、「(虎の巻シリーズ 1) チューニング技法虎の巻」を利用し、理化学研究所の責任で作成したものです。

本書は不定期に加筆修正しており、最新版は `http://accc.riken.jp/HPC/training.html` にあります。

目 次

第1章　チューニングの基礎

チューニングの目的は、経過時間や CPU 時間を短縮することです。本章ではまず、経過時間、CPU 時間、I/O 時間の定義について整理し、次に、チューニングの全体的な手順について説明します。

注意

- 本書の内容はときどき追加、更新しております。最新版は下記からダウンロード可能です。
  `http://accc.riken.jp/HPC/training/text.html`
- 本書内の「〜はマシン環境に依存します」という記述は、その動作がマシンのハードウェア、OS、コンパイラなどの種類によって異なる可能性がある事を意味します。
- 本書のプログラム例は Fortran で書かれています。C 言語を使用していて Fortran をご存じない方は以下の点に注意して下さい。
  - 本書のプログラム例の中で特に宣言をしていない変数のデータ形は、先頭文字が `I`, `J`, `K`, `L`, `M`, `N` で始まっている変数は整数、それ以外の変数は実数です。
  - 実数が単精度 (4 バイト) か倍精度 (8 バイト) かは、プログラム内に「`IMPLICIT REAL*8(A-H,O-Z)`」が指定されているかどうかによって以下のように決まります。

`IMPLICIT REAL*8(A-H,O-Z)`	なし	あり
`REAL A`	単精度	単精度
`REAL*8 B`	倍精度	倍精度
`DIMENSION C(10)`	単精度	倍精度
何も宣言されていない変数 `D`	単精度	倍精度

  - 本書のプログラムは原則として大文字で書かれていますが、Fortran では大文字と小文字の区別がないので、どちらで書いても（あるいは混在しても）構いません。例えば変数 `AA`, `Aa`, `aA`, `aa` は同一です。
  - Fortran では配列は `1` から始まります。
    【Fortran】`REAL A(2)`: `A(1)`, `A(2)`
    【C 言語】`float a[2]`: `a[0]`, `a[1]`
  - 多次元配列がメモリー上に配置される順番は、Fortran では左の添字が先に動く順番になります。
    【Fortran】`REAL A(2,2)`: `A(1,1)`, `A(2,1)`, `A(1,2)`, `A(2,2)`
    【C 言語】`float a[2][2]`: `a[0][0]`, `a[0][1]`, `a[1][0]`, `a[1][1]`

1-1　経過時間と CPU 時間

経過時間

ジョブの開始から終了までの時間を経過時間 (Elapsed Time) と呼びます。他のジョブが動いていると CPU 待ちの時間が発生するため、当然ながら経過時間は遅くなります。本書では、他のジョブが動いていない環境を想定します。

CPU 時間

CPU 時間は以下の 2 種類に分類されます。通常、システム CPU 時間はわずかなので無視してかまいません。なお、メーカーによってシステム CPU 時間の定義が若干異なるので、異なるメーカーのマシンで CPU 時間を厳密に比較する場合は『ユーザー CPU 時間 + システム CPU 時間』で比較して下さい。

- ユーザー CPU 時間
  ユーザープログラム自体が消費した CPU 時間です。
- システム CPU 時間
  入出力割り込み処理やページフォールトの処理など、オペレーティングシステムが消費した CPU 時間です。I/O 処理が多いプログラムの場合、システム CPU 時間が多くなります。

I/O 時間

他のジョブが流れていない状態でパフォーマンスを測定した場合、[経過時間] − [CPU 時間] = [I/O 時間] となります。正確には、I/O の際にも多少 CPU 時間がかかります。ただし I/O の方法によっては、かなり CPU 時間がかかることもあります。詳細は 6 章を参照して下さい。

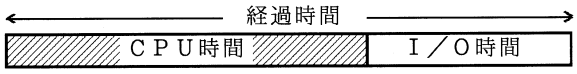


図 1-1-1

I/O 時間は以下の 2 種類に分類されます。

- ユーザープログラムによる I/O 時間
  ユーザープログラム自体が、ファイルや画面に対して入出力を行う時間です。
- ページングによる I/O 時間
  例えば 256 MB のメモリーを必要とするロードモジュール `a.out` を、メモリーが 128 MB のマシンで実行する場合、ページングスペース（ディスク）とメモリー間でロードモジュールの入出力が発生します。これをページングまたはスワッピングと呼びます (詳細は 6 章で説明します)。

1-2　ホットスポットとチューニング

ホットスポットとは

プログラム内で、CPU 時間を特に多く消費する（コストがかかるとも言います）限られた部分のことを本書ではホットスポットと呼びます。科学技術計算の場合、連立一次方程式、固有値、高速フーリエ変換などの、いわゆるソルバーの部分がホットスポットになることが多いですが、プログラムによってはホットスポットが顕著に現れないこともあります。

プログラムの構造の観点では、図 1-2-1 のように 100 回反復する 3 重ループがあった場合、処理 1、処理 2、処理 3 はそれぞれ 100 回、10000 回、1000000 回実行されます。従って多重ループでは、最も内側の部分が全体のほとんどを占め、ホットスポットになります。逆に処理 1 や処理 2 の部分は全体から見るとわずかなので無視してもかまいません。

```
    :
DO K = 1, 100
  処理1  ⬅ 100回
  DO J = 1, 100
    処理2  ⬅ 10000回
    DO I = 1, 100
      処理3  ⬅ 1000000回
    ENDDO
  ENDDO
ENDDO
    :
```

図 1-2-1

サブルーチン A〜E から構成されるプログラムを実行し、各サブルーチンのプログラム全体に対する CPU 時間の割合が図 1-2-2 のようになっていたとします。この例ではサブルーチン A がホットスポットになっています。サブルーチン A は全 CPU 時間の 70% を消費しているので、チューニングによってこれをもし半分にできれば全体でパフォーマンスは 35% 向上します。一方、サブルーチン A 以外の全てのサブルーチンを徹底的にチューニングし、計算時間を仮にゼロにしたとしても、パフォーマンスは全体で 30% しか向上しません。このように、ホットスポットの部分のみをチューニングする、つまり、最小の努力で最大の効果を上げるのが効率のよいチューニングであるといえます。

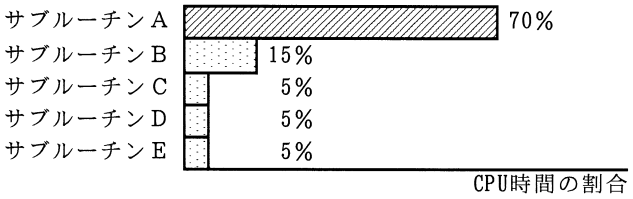


図 1-2-2

チューニングの手順

チューニングのおおまかな手順は以下のようになります。詳細な手順については各チューニング技法の説明を終了した後、8 章で説明します。

(1) ホットスポットの特定 (→ 3 章)
　↓
(2) ホットスポットに対してチューニング (→ 4〜7 章)
　↓
(3) 結果が正しいこととパフォーマンスが向上したかどうかを確認
　〓

第2章　コンパイルオプション

　パフォーマンスに関するコンパイルオプションを指定したことがないという人がたまにいますが、オプションを指定しないと通常パフォーマンスが悪くなるので、指定しないというのは論外です。本章では、パフォーマンスに関係のある主なオプションと、それ以外で知っていると便利なオプションについて説明します。

2-1　コンパイル・リンク・実行方法

コンパイル・リンク・実行方法

コンパイル・リンク・実行方法は、各メーカーのコンパイラによって異なりますので、本書では架空のマシンを想定し、一般的と思われる内容のみを説明します。

Fortran では、一般にコンパイルのコマンドは『f77』です。ソースプログラムの入ったファイル名が sample.f の場合、以下の①のようにコンパイル・リンクし、②のように実行します。①で、最適化のためのコンパイルオプションを指定しますが、これについては 2-2 節で説明します。

作成されるロードモジュール名のデフォルトは a.out ですが、ロードモジュール名を例えば test にしたい場合、③のようにコンパイル・リンクし、④のように実行します。

C 言語では、一般にコンパイルのコマンド名は『cc』です。ソースプログラムの入ったファイル名が sample.c の場合、⑤のようにコンパイル・リンクします。

```
f77 (コンパイルオプション) sample.f ↵          ··· ①
a.out ↵                                       ··· ②
f77 (コンパイルオプション) -o test sample.f ↵  ··· ③
test ↵                                        ··· ④
cc (コンパイルオプション) sample.c ↵           ··· ⑤
```

文法エラー

文法エラーがあるプログラムをコンパイルすると、エラー箇所（ソースプログラム内の行番号）とともに、そのエラーの重要度（例えば重大なエラー、コンパイラが修正できるエラー、警告メッセージなどの区別）が表示されます。重要度が低いと判定されたエラーでも、重大なエラーを誘発する可能性もありますので、念のためエラーの原因を確認することをお勧めします。

他社マシンから移植したときに発生するコンパイルエラー

同じ Fortran プログラムが、あるメーカーのコンパイラではエラーとならず、他のメーカーのコンパイラではエラーとなる事もあります。これは以下のような場合に発生します。

(1) コンパイラによってエラーチェックの厳しさが異なる。
(2) 特定のメーカーに固有の Fortran 文法の拡張機能を使用している。
(3) 組込関数の仕様（例えばタイマールーチン）が異なる。

(1) の例として、図 2-1-1 (1) の①のように配列 A を宣言し、サブルーチン側で②のように配列 A の先頭アドレスを合わせ、③のように A(1) 以外の値を参照や更新した場合、コンパイラによっては（確認のために）警告メッセージが出る事があります。

また、図 2-1-1 (2) の④のように、1 つの COMMON 文の中で 4 バイト変数 (I) の後に 8 バイト変数 (A) が並んでいる場合、コンパイラによっては警告メッセージが出る事があります。このような場合は、⑤のように 8 バイトの変数 (A) を先に書いて下さい。

```
PROGRAM MAIN
DIMENSION A(100)  ··· ①
CALL SUB(A)
END
```

```
SUBROUTINE SUB(A)
DIMENSION A(1)    ··· ②
A(100) = 1.0      ··· ③
END
```

図 2-1-1 (1)

```
  :
REAL*8 A
COMMON/COM/I,A  ··· ④
  :
```

```
  :
REAL*8 A
COMMON/COM/A,I  ··· ⑤
  :
```

図 2-1-1 (2)

2-2　パフォーマンスに関するコンパイルオプション

最適化のためのコンパイルオプション

コンパイラがパフォーマンスを向上させる目的で、プログラムの演算順序を変更したり、無駄な計算を省いてコンパイルを行うことを最適化と呼びます。本節では最適化のためのコンパイルオプションについて説明します。

一般に、最適化のオプションは『-O』(Optimize の意) というオプションが多く、-O1, -O2, -O3 と数字が大きくなるにつれて最適化のレベルが高くなります。なお、-O オプションの具体的な指定方法や意味は各コンパイラによって異なるので、詳細はマニュアルを参照して下さい。

以下の例では通常①が最も遅く、③が最も速くなりますが、プログラムによっては最適化レベルが低い方が速くなることもまれにあります。なお、最適化を行うとコンパイル時間がかかるので、プログラムの文法エラーを修正している段階では最適化オプションを指定しない方が効率的です。

```
f77     sample.f ↵  ①  (最適化を行わない)
f77 -01 sample.f ↵  ②  (低いレベルの最適化)
f77 -02 sample.f ↵  ③  (高いレベルの最適化)
```

最適化オプションを何も指定しないとどうなるか

上記①のように最適化オプションを何も指定しない場合、コンパイラの種類によって以下のように取扱いが異なります。マニュアルを確認して下さい。

(1) 最適化を全く行わず、パフォーマンスが悪くなる。
(2) デフォルトで、標準的な最適化オプションが指定される。
(3) デフォルトで、最も高度な最適化オプションが指定される（以下に示すように、この最適化オプションは非常に危険なので、もっと安全な最適化オプションに変更した方がよいでしょう)。

最適化のレベルを高くした場合の副作用

最適化のレベルを高くすると、計算結果が変わってしまう場合があります。図 2-2-1 (1) の④では、X=0.0 のとき「B/X」がゼロ除算例外になるのを防ぐため、二重線の IF 文が付加されています。ところで「B/X」の演算は DO ループの反復とは関係がなく、各反復で毎回同じ計算を行う必要はありません。そこで最適化のレベルが高い場合、コンパイラは演算 B/X を内部的に図 2-2-1 (2) の⑤のように DO ループの直前に移動します。ところが⑤では二重線の IF 文がないため、X=0.0 の場合はゼロ除算例外になってしまいます。

```
   :
DO I=1,N
  IF (X/=0.0) A(I) = B/X ④
ENDDO
   :
```

図 2-2-1 (1)

→

```
   :
TEMP = B/X   ··· ⑤
DO I=1,N
  IF (X/=0.0) A(I) = TEMP
ENDDO
   :
```

図 2-2-1 (2)

最適化のレベルを高くしすぎると、上記のように計算結果が変わる以外に、演算順序の変更によって計算精度が悪くなったり、場合によっては複雑な最適化処理をするためにコンパイラのバグ（誤ったコードを生成してしまう）が発生する恐れもあります。よく、1 つの入力データを実行し、結果が合っているから大丈夫だという人がいますが、その入力データを実行したときには通過せず、別の入力データで通過するプログラム内の経路にバグが発生している可能性もあり、正当性の確認にはなりません。

折角計算速度が速くなっても結果がおかしくては全く意味がないので、なるべく危険は避け、最適化のレベルはあまり高くせず、「そこそこ」速くてかつ安全な、言い換えると多くのユーザーが使用する、「枯れた」（バグがほぼなくなっている）レベルの最適化オプションにとどめる事を強くお勧めします。

2-3　計算結果がおかしい場合の考慮点

実定数の指定方法

プログラム内で円周率 3.1415 . . . などの実定数を指定する場合の考慮点について述べます。まず Fortran の仕様を説明します。図 2-3-1 (1) の①〜③で、左辺の変数 (PAI1〜PAI3) はいずれも倍精度ですが、右辺の定数は①と②は単精度、③は倍精度として扱われます。

このプログラムを実行すると、図 2-3-1 (2) に示すように、①，②は1，2のようになってしまいます（数値はコンパイラによって多少異なります)。①と②は単精度として扱われるので、図 2-3-1 (3) の A は、単精度で表現できて A に最も近い値 B にいったん変化した後、左辺の倍精度変数に代入するときに、倍精度で表現できて B に最も近い値 C に変化し、その値が1と2で書き出されます。一方③は倍精度として扱われるので、倍精度で表現できて A に最も近い値 D に変化し、その値が3で書き出されます。

単精度で指定した定数を自動的に倍精度に変換する (①，②を自動的に③に変換する) 機能を持っているコンパイラもあります。例えば以下の (1) (2) のように、デフォルトはコンパイラによって異なりますので、マニュアルで確認して下さい。異なるメーカーのマシンで計算結果が異なる場合、このデフォルトの違いが原因である事がよくあります。なお、プログラムを他のマシンに移植したときの可搬性の点からは、コンパイラの自動変換機能を使うよりも、単精度の定数を手で明示的に倍精度に変更した方がよいと思います。

(1) デフォルトでは、上記のように Fortran の仕様に従う。コンパイラによっては、デフォルトを (2) に変更するコンパイラオプションが提供されている場合があります

(2) デフォルトでは、単精度定数を自動的に倍精度定数に変換する。このとき「DATA A/1.1/」のように、倍精度の変数 A に対して DATA 文で単精度の定数を指定した場合は、倍精度に変換しないコンパイラもあります。

```
  :
REAL*8 PAI1,PAI2,PAI3  倍精度
PAI1 = 1.1111111111111111     ①単精度
PAI2 = 1.11111111111111110    ②単精度
PAI3 = 1.11111111111111110    ③倍精度
PRINT *,'PAI1 = ',PAI1          1
PRINT *,'PAI2 = ',PAI2          2
PRINT *,'PAI3 = ',PAI3          3
  :
```

図 2-3-1 (1)

```
PAI1 = 1.11111116409301758   1
PAI2 = 1.11111116409301758   2
PAI3 = 1.11111111111111116   3
```

図 2-3-1 (2)

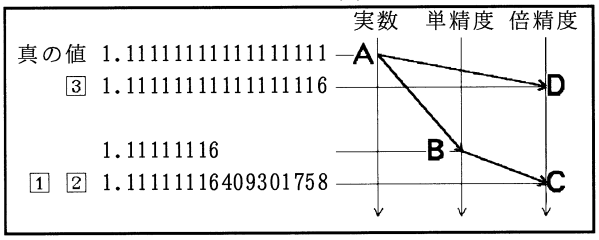


図 2-3-1 (3)

実定数を引数に指定する場合の考慮点

図 2-3-2 (1) では、サブルーチンコールの実引数①，②に直接定数を指定していますが、①は「D0」が付いていないので、前述のように単精度扱いとなります。ところがサブルーチンの仮引数 A，B，C は倍精度で宣言しているので、プログラムを実行すると、A には正しい値が渡らず、図 2-3-2 (2) に示すように結果がおかしくなります。この場合、②のように「D0」をつけるか、または③のように倍精度変数で指定するようにしてください。数値計算ライブラリーのサブルーチンの引数に直接定数を指定する場合も同じ注意が必要です (7-1 節の「考慮点」参照)。同様に、④のように実引数に倍精度の定数を指定し、サブルーチンの仮引数 X は単精度で宣言している場合 (つまり①の逆のケース) も結果がおかしくなります。

```
PROGRAM MAIN
REAL*8 TEMP    ③
TEMP = 1.0D0   ③
CALL SUB(1.0,1.0D0,TEMP,1.0D0)
END        ①   ②   ③   ④
```

```
SUBROUTINE SUB(A,B,C,X)
REAL*8 A,B,C
REAL   X
PRINT *,'A = ',A
PRINT *,'B = ',B
PRINT *,'C = ',C
PRINT *,'X = ',X
END
```

図 2-3-2 (1)

```
A = 1.424050322466053-306  ✖
B = 1.000000000000000      ○
C = 1.000000000000000      ○
X = 0.000000000e+00        ✖
```

図 2-3-2 (2)

Fortran の文法違反 (1)

サブルーチンやファンクション内で同じ変数を異なる名前で使用した場合、その変数を更新するのは Fortran の文法違反なので、避けて下さい。なお、コンパイラの種類や最適化のレベルによっては、結果が正しくなることもありますが、いずれにしても危険なので避けて下さい。

以下に例を示します。図 2-3-3 (1)〜(3) の各例はいずれも、メインルーチンの変数 M を、サブルーチン SUB で 2 つの変数 M, N として使用しています。

- 図 2-3-3 (1) のメインルーチンでは、CALL SUB の実引数に M を 2 回指定し、サブルーチン SUB では仮引数 M, N として使用しています。
- 図 2-3-3 (2) のメインルーチンでは、EQUIVALENCE で結合した M と N を CALL SUB の実引数に指定し、サブルーチン SUB では仮引数 M, N として使用しています。
- 図 2-3-3 (3) のメインルーチンでは、M を CALL SUB の実引数と COMMON の両方に指定し、サブルーチン SUB では仮引数を M、COMMON の変数を N として使用しています。

各プログラムの①で M に 1 を代入すると、N も M と同一なので 1 となり、②で M と N を合計すると「K = 2」となるはずです。ところが③で書き出すと「K = 1」が表示されることがあります。

```
M = 0
CALL SUB(M,M)
END
SUBROUTINE SUB(M,N)
M = 1                ①
K = M + N            ②
PRINT *,'K =',K      ③
END
```

図 2-3-3 (1)

```
EQUIVALENCE(M,N)
M = 0
CALL SUB(M,N)
END
SUBROUTINE SUB(M,N)
M = 1                ①
K = M + N            ②
PRINT *,'K =',K      ③
END
```

図 2-3-3 (2)

```
COMMON/COM/M
M = 0
CALL SUB(M)
END
SUBROUTINE SUB(M)
COMMON/COM/N
M = 1                ①
K = M + N            ②
PRINT *,'K =',K      ③
END
```

図 2-3-3 (3)

Fortran の文法違反 (2)

サブルーチンコールの実引数が定数か、あるいは演算子またはカッコを含む式の場合、サブルーチン側の対応する仮引数に値を代入するのは Fortran の文法違反なので、避けて下さい。なお、コンパイラの種類や最適化のレベルによっては、たまたま結果が正しくなることもありますが、いずれにしても危険なので避けて下さい。

以下に例を示します。図 2-3-4 の①で実引数に定数「1」を設定し、対応するサブルーチン SUB の仮引数 N を②で更新すると、メモリ上で定数「1」の入っている場所が「999」に書き替えられてしまいます。このため、③で変数 M に「1」を設定して④で書き出すと、「M = 1」ではなく「M = 999」が表示されてしまいます。

同様に、図 2-3-5 の⑤で実引数に演算子 (+) を含む式「M+1」を設定し、対応するサブルーチン SUB の仮引数 N を⑥で更新し⑦で書き出すと、「M+1 = 6」ではなく「M+1 = 999」が表示されてしまいます。

```
PROGRAM MAIN
CALL SUB(1)          ①
M = 1                ③
PRINT *,'M =',M      ④
END
SUBROUTINE SUB(N)
N = 999              ②
END
```

➡ M = 999

図 2-3-4

```
PROGRAM MAIN
M = 5
CALL SUB(M+1)        ⑤
PRINT *,'M   =',M
PRINT *,'M+1 =',M+1  ⑦
PRINT *,'M+2 =',M+2
END
SUBROUTINE SUB(N)
N = 999              ⑥
END
```

➡

```
M   = 5
M+1 = 999
M+2 = 7
```

図 2-3-5

変数のゼロクリアについて

コンパイラによっては、ジョブの実行が開始した時点で変数がゼロクリアされている保証がない場合があります。変数にゼロが入っているとみなしていきなり参照すると、プログラムが誤作動する可能性がありますので注意して下さい。

入力ファイル内のタブ文字

入力ファイルのデータにタブ文字が含まれている場合、コンパイラによっては、タブ文字を一文字として認識してしまうため、プログラムで想定した入力形式と桁がずれて計算結果がおかしくなります。

この問題が発生し、入力ファイル内のタブ文字をスペース文字に変換したい場合、untab コマンド (付録 A 参照) を使用します（ただしこのコマンドがサポートされているかどうかはマシン環境に依存します)。

コンパイラのバグ

前節でも説明しましたが、コンパイルオプションで指定する最適化レベルを高くし過ぎると、結果がおかしくなる場合があるので注意して下さい。

また、コンパイラのバージョンが上がった直後は一般にコンパイラのバグが発生しやすいので、バージョンアップ後、ある程度期間が経ち、枯れてから導入した方が良いかも知れません。また、コンパイラを導入後に発見されたバグを後で修復することが可能な場合は、修復するようにして下さい。

どう見てもプログラムは正しいはずなのに計算結果がおかしい場合、コンパイラのバグが原因である可能性があります。このような場合、以下の手順で問題箇所を絞り込みます。

(1) まず、図 2-3-6 (1) のケース 1 に示すように、コンパイルオプションの最適化レベルを最低にして実行して下さい。それでも計算結果がおかしい場合はプログラムのロジックエラーの可能性が高く、計算結果が正しい場合はコンパイラの最適化に伴うバグの可能性があります。
後者の場合、ケース 2、ケース 3 に示すように最適化レベルを少しずつ上げ、どのレベルからおかしくなるかを調べます。この例ではコンパイルオプション O2 を指定するとおかしくなりました。

(2) 次に、コンパイルオプション O2 を指定するとおかしくなるのはどのサブルーチンなのかを調べます。プログラム内に A, B, C, D, E, F, G, H の 8 個のサブルーチンが含まれていて、問題のサブルーチンは 1 つだけだとします。まず図 2-3-6 (2) のケース 4 に示すように、サブルーチンを A〜D と E〜H の 2 つのファイルに分け、前者は最適化オプションなしで、後者は問題のコンパイルオプション O2 を指定してコンパイルします。実行すると計算結果はエラーとなったので、問題のサブルーチンは E F G H のいずれかであることが分かります。
次に、ケース 5 に示すように E F は最適化オプションなし、G H は O2 を指定してコンパイル／実行すると、今度は結果が OK となりました。このことから問題のサブルーチンは E F のいずれかであることが分かります。
最後に、ケース 6 に示すように E は最適化オプションなし、F は O2 を指定してコンパイル／実行するとエラーとなりました。このことからコンパイルオプション O2 を指定するとおかしくなるサブルーチンは F であることが分かりました。

このように、問題のサブルーチンが 1 つだけの場合、2 分法を使用すると 3 回のテストでそのサブルーチンを特定することができます。さらに、サブルーチン内のどの DO ループやステートメントがおかしくなるかを特定することもできますが、説明は省略します。

ﾃｽﾄｹｰｽ	ｹｰｽ1	ｹｰｽ2	ｹｰｽ3
ｺﾝﾊﾟｲﾙｵﾌﾟｼｮﾝ	な し	01	02
計 算 結 果	OK	OK	ｴﾗｰ

図 2-3-6 (1)

ﾃｽﾄｹｰｽ	ｹｰｽ4		ｹｰｽ5		ｹｰｽ6	
ｻﾌﾞﾙｰﾁﾝ	ABCD	EFGH	abcdEF	GH	abcdEgh	F
ｺﾝﾊﾟｲﾙｵﾌﾟｼｮﾝ	な し	02	な し	02	な し	02
計 算 結 果	ｴﾗｰ		OK		ｴﾗｰ	

図 2-3-6 (2)

2-4　デバッグに関するコンパイルオプション

デバッグ時に役に立つオプション

コンパイラによっては、通常のコンパイル／リンクではエラーにならない、サブルーチンコール間の矛盾や、サブルーチンコールの引数の不一致などをチェックするコンパイルオプションが提供されている場合があります。プログラムのデバッグ中にどうしてもバグが見つからない場合、このオプションを指定するとバグが見つかることがあります。

ただし、このオプションを指定すると、通常パフォーマンスが低下するので、デバッグ時にのみ使用します。

- 図 2-4-1 (1) では、同じサブルーチンコールの引数のデータ型が、一方では単精度に、他方では倍精度になっています。
- 図 2-4-1 (2) では、同じサブルーチンコールの引数の数が、一方では 1 つ、他方では 2 つになっています。
- 図 2-4-1 (3) では、引数のデータ型が、サブルーチンをコールする側では単精度に、サブルーチン側では倍精度になっています。
- 図 2-4-1 (4) では、引数の数が、サブルーチンをコールする側では 1 つ、サブルーチン側では 2 つになっています。

コンパイル／リンクした場合、通常このような矛盾は検出されませんが（コンパイラの種類によっても異なります）、矛盾をチェックするためのオプションを指定すると、矛盾箇所と理由を示すメッセージが表示されます。

```
PROGRAM MAIN
REAL*4 A
REAL*8 B
CALL SUB(A)
CALL SUB(B)
END
```

図 2-4-1 (1)

```
PROGRAM MAIN
REAL*4 A,B
CALL SUB(A)
CALL SUB(A,B)
END
```

図 2-4-1 (2)

```
PROGRAM MAIN
REAL*4 A
CALL SUB(A)
END
SUBROUTINE SUB(A)
REAL*8 A
  :
END
```

図 2-4-1 (3)

```
PROGRAM MAIN
REAL*4 A
CALL SUB(A)
END
SUBROUTINE SUB(A,B)
REAL*4 A,B
  :
END
```

図 2-4-1 (4)

演算例外

実数の演算例外には次のような種類があります。

- 浮動小数点オーバーフロー
- 浮動小数点アンダーフロー
- 浮動小数点ゼロ除算
- 浮動小数点無効演算 ($\infty - \infty$, $\infty \div \infty$, $0 \div 0$, $\infty \times 0$, SQRT(-1.0) など)

プログラムの実行中に上記のような演算例外が発生した場合の（デフォルトの）対処方法は、マシン環境によって以下のように異なります。

(2) の場合、対処方法を (1) に変更するコンパイルオプションが提供されている場合もあります。パフォーマンスがさほど低下しないのであれば、そのコンパイルオプションを指定した方が安全です。

(1) メッセージを表示して異常終了
(2) メッセージを表示せず、計算をそのまま続行

dbx と -g オプション

図 2-4-2 (1) のプログラムは、← のステートメントで配列 A(10) の範囲を越えた部分をアクセスするため、ジョブは『異常終了発生』を意味するメッセージを表示し、core ファイルを作成して異常終了します（マシン環境によっては異常終了しない場合もあります)。

異常終了した場合、マシン環境によって、異常終了した箇所（ステートメント番号）を表示する場合と、表示しない場合があります。表示しない場合、dbx というデバッガーが使用可能であれば、下記のようにして異常終了した箇所を調べることができます。なお、dbx がサポートされていない場合、dbx に相当する他のデバッガーが提供されている場合もあります。

- 図 2-4-2 (2) の①で -g オプションをつけてプログラムを再びコンパイルし（マシン環境によって異なります)、②で実行します。すると③のメッセージを表示してジョブは異常終了し、core ファイルが作成されます。なお、-g オプションをつけるとパフォーマンスが低下する可能性があるので、デバッグのときにのみ使用して下さい。
- ④で dbx を開始します。このときロードモジュール名 (a.out) と core ファイル名 (core) を指定します。すると⑤と⑥のメッセージが表示されます。⑤はファイル sample.f の 7 行目にある、サブルーチン sub 内のステートメントで異常終了したことを示し、⑥は異常終了したステートメントが『A(100000) = 1.0』であることを示します。
- 異常終了したステートメントを含むサブルーチンをコールしている上位のサブルーチン名を調べたい場合、⑦のように where コマンドを実行します。⑧は異常終了したステートメントが含まれるサブルーチンは、test() (PROGRAM TEST のこと) 内の、ファイル sample.f の 3 行目からコールされたことを示します。
- 最後に⑨で dbx を終了します。

```
1   PROGRAM TEST
2   DIMENSION A(10)
3   CALL SUB(A)
4   END
5   SUBROUTINE SUB(A)
6   DIMENSION A(*)
7   A(100000) = 1.0 ←
8   END
```

図 2-4-2 (1)　sample.f

```
f77 -g (コンパイルオプション) sample.f ↵  ··· ①
a.out ↵                                   ··· ②
異常終了発生                              ··· ③
dbx a.out core ↵                          ··· ④
dbx Version 3.1.
Type 'help' for help.
[using memory image in core]

segmentation violation in sub at line 7 in file "sample.f" ··· ⑤
    7         A(100000) = 1.0             ··· ⑥
(dbx) where ↵                             ··· ⑦
sub(a = (...)), line 7 in "sample.f"
test(), line 3 in "sample.f"              ··· ⑧
(dbx) quit ↵                              ··· ⑨
```

図 2-4-2 (2)

2-5　環境変数

装置番号とファイル名の結合

図 2-5-1 (1) の①のように、プログラム内にファイル名 (outfile) を明示的に指定した場合、ファイル名を変更するたびに再コンパイルしなければならず面倒です。コンパイラによっては、図 2-5-1 (2) の②に示すように、プログラム内では装置番号で記述し、実行時に③に示すように装置番号とファイル名を関係付ける環境変数が提供されてる場合もあります。なお③は架空の例で、指定方法はコンパイラによって異なりますので、マニュアルを参照して下さい。

```
  :
OPEN(10,FILE="outfile") ①
WRITE(10) A
  :
```

図 2-5-1 (1)

```
  :
OPEN(10)                ②
WRITE(10) A
  :
```

```
export FILE10=outfile  ↵ ③
```

図 2-5-1 (2)

2-6　Fortran 90

Fortran 90 は 1994 年に規格化され、現在ではほとんどのコンパイラで使用できます。通常、既存の Fortran 77 のプログラムに、Fortran 90 のステートメントを共存して使用することができます。

本節では、Fortran 90 の新機能のうち、知っておくと便利ないくつかの機能について説明します。Fortran 90 の詳細は [1]〜[7] などを参照して下さい。

なお、Fortran 90 のプログラム形式には Fortran 77 と同じ固定形式 (1 行が 72 桁で 7〜72 桁目にステートメントを記述) と自由形式 (1 行が 132 桁で任意の位置から記述できる) がありますが、本書では固定形式で記述します。またコンパイラコマンドはコンパイラによって異なりますが、固定形式の場合は例えば「f77」(ただし通常 Fortran 90 も使えます)、自由形式の場合は例えば「f90」となります。

コメント行

Fortran 77 では、図 2-6-1 の①に示すように 1 桁目に「C」(または「*」) を書くと、その行はコメント行になりましたが、Fortran 90 では新たに②の「!」が追加されました。「!」は③のように一般のステートメントの後ろに書くこともでき、この場合は「!」より右側の部分がコメントとなります。

なお、Fortran 90 とは関係ありませんが、分かりやすさのために 1 行ブランクを空ける場合、④では先頭に「C」が付いていますが、⑤のように「C」を付けなくても構いません。

```
C  comment line                  ①
!  comment line                  ②
      A = 0.0    ! zero clear   ③
C    (空白行)                    ④
     (空白行)                    ⑤
```

図 2-6-1

IF 文の条件式

Fortran 90 では、IF 文の条件式は図 2-6-2 の左側の記述方法に加え、右側の記述方法も可能になりました。右側の記述法の方が大小関係が直感的に分かりやすいと思います。図 2-6-3 に例を示します。

Fortran77	Fortran90
IF (A .LT. B)	IF (A < B)
IF (A .LE. B)	IF (A <= B)
IF (A .EQ. B)	IF (A == B)
IF (A .NE. B)	IF (A /= B)
IF (A .GT. B)	IF (A > B)
IF (A .GE. B)	IF (A >= B)

図 2-6-2

```
    :
IF (A<B) THEN
  :
ELSEIF (A==B) THEN
  :
ELSE
  :
ENDIF
    :
```

図 2-6-3

SELECT CASE 文

図 2-6-4 (1) では、整数変数 K の値によって 4 つに分岐しています。このプログラムは、図 2-6-4 (2) に示すように、Fortran 90 の SELECT CASE 文を使用して書くこともできます。図 2-6-4 (2) は図 2-6-4 (1) と同じ意味を持ち、変数 K が 2 以下ならば①を、3，5，7 のいずれかならば②を、8〜10 または 12 以上ならば③を、それ以外ならば④を実行します。

いくつか補足します。

- K の部分には、整数式、論理式、または文字式を指定することができます。
- ①〜④の各部分には、複数のステートメントを記述することができます。
- CASE 文の順序は任意です。従って、「CASE DEFAULT」が最後以外の場所にあっても構いません。

この例のように、1 つの条件式で複数の分岐を行ない、条件式が複雑になる場合、IF〜ELSEIF〜ELSE〜ENDIF を使用するよりも、SELECT CASE 文の方が、条件式が書きやすいのでプログラムが簡単になります。

```
    :
IF      (K<=2) THEN
    ①
ELSEIF (K==3.OR.K==5.OR.K==7) THEN
    ②
ELSEIF ((8<=K.AND.K<=10).OR.(12<=K)) THEN
    ③
ELSE
    ④
ENDIF
    :
```

図 2-6-4 (1)

=

```
    :
SELECT CASE(K)
CASE(:2)
    ①
CASE(3,5,7)
    ②
CASE(8:10,12:)
    ③
CASE DEFAULT
    ④
END SELECT
    :
```

図 2-6-4 (2)

DO 文／DO WHILE 文

- 図 2-6-5 (1) の上図の下線のように DO ループの文番号を付けるのが面倒な場合、図 2-6-5 (1) の下図のように記述することができます。多重ループの場合は図 2-6-5 (2) のようになります。
- 図 2-6-5 (3) のように反復回数を記述しない DO ループは無限ループを意味します。
- 図 2-6-5 (4) は DO WHILE 文で、①の下線部が真の間は②，③を実行し、偽になると②，③を実行せずに④に移動します。実行結果を図 2-6-5 (4) の下図に示します。

```
    :
    DO 123 I=1,N
      A(I) = B(I)
123 CONTINUE
    :
```

↓

```
    :
    DO I=1,N
      A(I) = B(I)
    ENDDO
    :
```

図 2-6-5 (1)

```
DO K=1,N
  DO J=1,N
    DO I=1,N
      :
    ENDDO
  ENDDO
ENDDO
```

図 2-6-5 (2)

```
DO
  :
ENDDO
```

図 2-6-5 (3)

```
 I = 1
DO WHILE (I<3)      ①
  PRINT *,'I = ',I  ②
  I = I + 1         ③
ENDDO
PRINT *,'END'       ④
```

↓

```
I = 1
I = 2
END
```

図 2-6-5 (4)

DO 文／IF 文の名前

図 2-6-8 (1) に示すように、DO ループ内に含まれるステートメントが多く、DO 文と ENDDO 文の対応が分かりにくい場合、①, 1の下線部のように DO 文と ENDDO 文に名前を付ける事ができます。①を指定した場合、1の指定は必須です。①の名前の後には「:」を付けます。名前の 1 文字目に数字を指定することはできません。②と2についても同様です。

IF 文、SELECT CASE 文、WHERE 文にも名前を付けることができます。IF 文と SELECT CASE 文の例を図 2-6-8 (2) (3) に示します。③, ④を指定した場合、3, 4の指定は必須ですが、[3], [4] の指定は任意です。

```
OUTER: DO K=1,N      ①
  DO J=1,N
    INNER: DO I=1,N  ②
      :
      : (ｽﾃｰﾄﾒﾝﾄが多い)
      :
    ENDDO INNER      2
  ENDDO
ENDDO OUTER          1
```

図 2-6-8 (1)

```
CHECK: IF(I==1) THEN     ③
  :
ELSEIF(I==2) THEN CHECK [3]
  :
ELSE CHECK               [3]
  :
ENDIF CHECK              3
```

図 2-6-8 (2)

```
CHECK: SELECT CASE(I) ④
CASE(1) CHECK         [4]
    :
CASE(2) CHECK         [4]
    :
CASE DEFAULT CHECK    [4]
    :
END SELECT CHECK      4
```

図 2-6-8 (3)

IMPLICIT NONE

Fortran では、変数を宣言文で宣言せずに使用すると、先頭文字が「I, J, K, L, M, N」で始まっている変数は整数、それ以外の変数は単精度の実数として、自動的に宣言が行われます。

図 2-6-9 (1) の①で、変数名 A を誤って AA としても、AA は単精度の実数として自動的に宣言されるため、コンパイルエラーにはならず（コンパイラによってはウォーニングが出ます)、誤りに気付きません。

図 2-6-9 (2) の②のように Fortran 90 の「IMPLICIT NONE」を指定した場合、上記で説明した自動的な宣言は行われず、全ての変数を明示的に宣言する必要があります。従って図 2-6-9 (2) は「変数 AA が宣言されていない」というコンパイルエラーとなり、誤りに気付くことができます。

```
SUBROUTINE SUB(A)
REAL A
AA = 1.0 ⬅ 間違い ①
END
```

図 2-6-9 (1)

```
SUBROUTINE SUB(A)
IMPLICIT NONE        ②
REAL A
AA = 1.0 ⬅ 間違い
END
```

図 2-6-9 (2)

CYCLE 文／EXIT 文

- 図 2-6-10 の③の IF 文が真の場合 CYCLE 文が実行され、(④を実行せずに) ①に戻り、DO ループの次の反復を実行します。実行結果を下段に示します。
- 図 2-6-11 の③の IF 文が真の場合 EXIT 文が実行され、(④および DO ループの次以降の反復は実行せずに) DO ループは終了し、DO ループの次のステートメントである⑤を実行します。実行結果を下段に示します。
- 図 2-6-12 のように、無限ループを終了する場合も EXIT 文を使用することができます。①が真の場合 EXIT 文が実行され、無限ループの次のステートメントである②を実行します。

```
DO I=1,3                  ①←
  PRINT *,'A: I = ',I     ②
  IF (I==2) CYCLE         ③
  PRINT *,'B: I = ',I     ④
ENDDO
PRINT *,'END'
```

```
A: I = 1
B: I = 1
A: I = 2
A: I = 3
B: I = 3
END
```

図 2-6-10

```
DO I=1,3                  ①
  PRINT *,'A: I = ',I     ②
  IF (I==2) EXIT          ③
  PRINT *,'B: I = ',I     ④
ENDDO
PRINT *,'END'             ⑤
```

```
A: I = 1
B: I = 1
A: I = 2
END
```

図 2-6-11

```
DO
    :
  IF (～) EXIT ①
    :
ENDDO
  :                       ②←
```

図 2-6-12

- 図 2-6-13 のように多重ループ内に EXIT 文が含まれている場合、①が真ならば EXIT 文が実行され、内側の DO ループの次のステートメントである②に移動します。
- 多重ループ内の EXIT 文から、(例えば) 外側の DO ループの次のステートメントに移動したい場合、次のようにします。まず図 2-6-14 の二重線に示すように、外側の DO ループに名前を付けます。①が真の場合 EXIT 文が実行され、そこに指定した名前の DO ループの次のステートメントである②に移動します。

```
DO J=1,2
  DO I=1,3
    IF (I==2) EXIT ①
  ENDDO
    :                     ②←
ENDDO
  :
```

図 2-6-13

```
OUTER: DO J=1,2
  DO I=1,3
    IF (I==2) EXIT OUTER ①
  ENDDO
    :
ENDDO OUTER
  :                       ②←
```

図 2-6-14

Fotran 90 のその他の機能

- C 言語のように構造体やポインターを使用することができます。
- 以下のように、1 行に複数のステートメントを『;』で区切って書くことができます。

```
  :
A = 1.0; B = 2.0; C = 3.0
  :
```

MODULE 文／USE 文

MODULE 文／USE 文は、複数のルーチン（メインルーチン、サブルーチン、ファンクション）間で変数や配列を共有したい場合に便利な機能です。なお、MODULE 文にはその他の機能もありますが、説明は省略します。

図 2-6-15 の①に示すように、MODULE～END で囲まれた部分に、複数のルーチン間で共有したい変数や配列を宣言し、MODULE 文に任意の名前 (本例では「COM」) を付けます。

②，③に示すように、任意のルーチンの USE 文で MODULE 文の名前「COM」を指定すると、そのルーチンからは①内の変数や配列を参照／更新することができます。本例では MAIN と SUB1 からは①の変数を参照／更新することができますが、SUB2 からはできません。

MODULE 文は図 2-6-16 の⑤，⑥に示すように複数指定することができます。この例では、サブルーチン SUB1 は⑤と⑥の変数を参照／更新することができ、SUB2 は⑤の変数のみを参照／更新することができます。

MODULE 文／USE 文に関していくつか補足します。

- ①の部分は、プログラムと同じファイルに入れる事ができます。ただし、①はファイル内の最初の USE 文 (本例では②) よりも前に置かなければなりません（コンパイラまたはコンパイルオプションによっては後ろでもよい場合もあります)。従って①はプログラムの先頭に置くのがよいでしょう。
- ③，④に示すように、USE 文は各ルーチン（メインルーチン、サブルーチン、ファンクション）の一番上の行（コメント行を除く）に書く必要があります。
- MODULE 文／USE 文と INCLUDE ファイル／INCLUDE 文の機能は一見似ていますが、本質的に異なる点があります。MODULE 文／USE 文では、図 2-6-17 に示すように⑧から⑦を参照／更新できますが、⑦と⑧はあくまで異なる世界です。従って⑦で指定されている「IMPLICIT REAL*8」（注）は⑦内でのみ有効で、⑧内には適用されず、配列 B(2) は単精度になります。⑧で「IMPLICIT REAL*8」が必要な場合は USE 文の後に別途指定して下さい。
  一方 INCLUDE ファイル／INCLUDE 文では、図 2-6-18 (1) (2) に示すように、⑨の部分が⑩にはめ込まれるので、「IMPLICIT REAL*8」は⑨だけでなく⑪でも有効になり、配列 B(2) は倍精度になります。なお、⑦はプログラムと同じファイル内に入れることができますが、⑨は別のファイルにしなければなりません。
- MODULE 文、USE 文を使用するプログラムが複数ファイルから構成されている場合、デフォルトではコンパイル／リンクがうまくいかないことがあります。対処方法はコンパイラのマニュアルを参照して下さい。

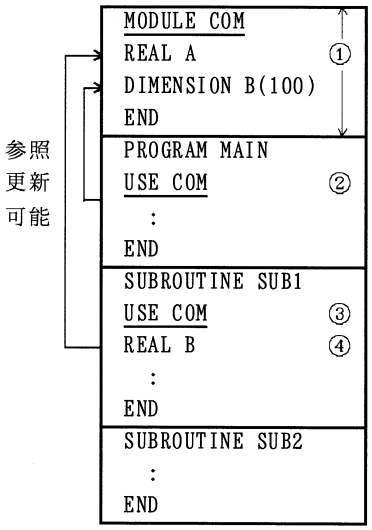


図 2-6-15

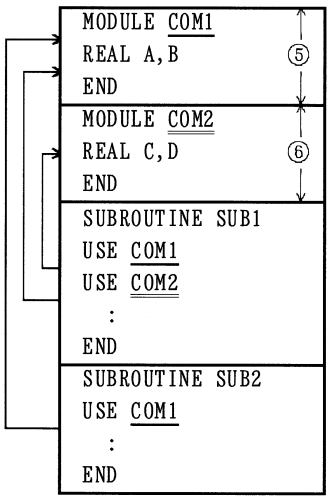


図 2-6-16

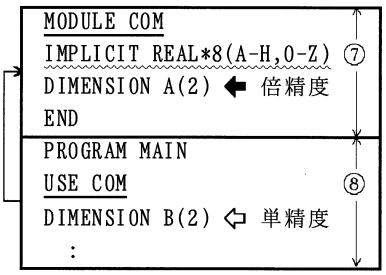


図 2-6-17

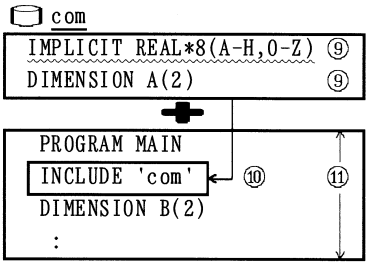


図 2-6-18 (1)

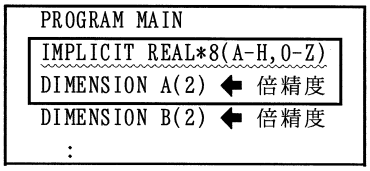


図 2-6-18 (2)

（注）Fortran では、宣言されていない変数／配列、DIMENSION 文で宣言した配列は、デフォルトでは、I，J，K，L，M，N で始まる変数／配列は整数、A～H，O～Z で始まる変数／配列は単精度の実数になります。「IMPLICIT REAL*8(A-H,O-Z)」を指定した場合は、A～H，O～Z で始まる変数／配列は倍精度の実数になります。

メモリーの動的割り当て（割り付け）機能

プログラムによっては、配列の大きさをコンパイル時に明示的に指定するのではなく、実行時に、例えば読み込んだ入力データの大きさに従って、必要最小限の大きさの配列を割り当てたい場合があります。このときメモリーの動的割り当て機能 (C 言語の関数 malloc に相当) を使用します。この機能で、一時的に使用する配列の割り当てと解放をこまめに繰り返すことにより、メモリー量を大幅に減らせる場合があります。

- 図 2-6-19 の①で、ALLOCATABLE 文を使用して、単精度実数 (REAL) の配列 A と B をプログラムの実行時に動的に割り当てる宣言をします。先頭の「REAL,～」は、配列が倍精度実数の場合は「REAL*8,～」、整数の場合は「INTEGER,～」となります。また配列が 1 次元の場合は A(:)、2 次元の場合は B(:,:) のように指定します。
- ②で例えば標準入力から配列 A と B の大きさ N を読み込みます。
- 読み込んだ N を使用して③の ALLOCATE 文を実行すると、この時点で初めて配列 A と B が実際にメモリー上に割り当てられます。以後は配列 A と B を通常の配列と同じように使用することができます。なお ALLOCATE 文で指定する配列の大きさは、標準入力から読み込んだ以外の値を使用してももちろん構いません。
- ③の「STAT=」の指定は任意です。指定した場合、③の ALLOCATE 文が終了すると、「STAT=」で指定した整定数 (本例では IERR) にリターンコードが戻ります。通常、正常終了したらゼロ、異常終了したらゼロ以外の値が戻りますが、この値はマシン環境によって異なる可能性があるので、マニュアルを参照して下さい。
- ③を実行し、メモリーが足りなくて配列を割り当てることができなかった場合、マシン環境によって、エラーメッセージを表示して異常終了する場合と、エラーメッセージを表示しない場合があります。後者の場合、③で「STAT=」を指定し、④でリターンコードをチェックして、エラーの場合はエラーメッセージを表示してプログラムを停止させることができます。
- 例えば（大きな）配列 A と B が不要となり、新規に（大きな）別の配列 C を確保したいような場合、配列 A と B を⑤の DEALLOCATE 文で解放することができます。もちろん解放せずにプログラムを終了しても構いません。DEALLOCATE 文の場合も、⑤, ⑥のように「STAT=」でリターンコードをチェックすることができます（任意）。
- 動的に割り当てた配列 A を引数でサブルーチンに渡す場合は図 2-6-20 のようにします。
  受け渡したい配列が多い場合、引数に書くのは面倒なので、図 2-6-21 (1) のように COMMON 文を使用することがあります。ところが図 2-6-21 (2) に示すように、COMMON 文に指定する配列を動的に割り当てることはできません。しかし前述の MODULE 文の中であれば ALLOCATABLE 文を指定できるので、図 2-6-21 (3) のようにすれば動的割り当てした配列を引数に使わずにサブルーチンに渡すことができます。

```
   :
REAL,ALLOCATABLE::A(:),B(:,:)      ①
READ(5,*) N                        ②
ALLOCATE(A(N),B(N,N),STAT=IERR)    ③
IF (IERR/=0) THEN                  ↑
  PRINT *,'ALLOCATION ERROR'       ④
  STOP                             ↓
ENDIF
   :
DEALLOCATE(A,B,STAT=IERR)          ⑤
(④と同じ)                          ⑥
```

図 2-6-19

```
PROGRAM MAIN
REAL,ALLOCATABLE::A(:)
READ(5,*) N
ALLOCATE(A(N),STAT=～)
CALL SUB(A,N)
  :
SUBROUTINE SUB(A,N)
DIMENSION A(N)
  :
```

図 2-6-20

```
PROGRAM MAIN
DIMENSION A(100)
COMMON/COM/A
CALL SUB
  :
SUBROUTINE SUB
DIMENSION A(100)
COMMON/COM/A
  :
```

図 2-6-21 (1)

```
   :
REAL,ALLOCATABLE::A(:)
COMMON/COM/A
   :
```

図 2-6-21 (2)　×　不可

```
MODULE COM
REAL,ALLOCATABLE::A(:)
END
PROGRAM MAIN
USE COM
READ(5,*) N
ALLOCATE(A(N),STAT=～)
CALL SUB
  :
SUBROUTINE SUB
USE COM
  :
```

図 2-6-21 (3)

第3章　パフォーマンス測定方法

本章では、ジョブ全体やプログラムの一部分のパフォーマンスを測定する方法、およびホットスポットの特定方法について説明します。

3-1　ジョブ全体のパフォーマンス測定方法

time コマンドの使用方法

ジョブ全体の経過時間と CPU 時間の測定は、UNIX や Linux で提供されている time コマンドを使用します。図 3-1-1 のプログラムを図 3-1-2 の②で実行すると、①に示すプログラムからの出力が③に表示され、time コマンドの結果が④〜⑥に表示されます。

- ④　経過時間：ジョブの開始から終了までにかかった時間。
- ⑤　ユーザー CPU 時間：プログラムが CPU を使用した時間。
- ⑥　システム CPU 時間：I/O などの際に（内部的に）システムコールなどにかかった時間

このうちシステム CPU 時間は、I/O が著しく多いジョブ以外ではごくわずかなので、通常は無視して構いません。

```
PROGRAM MAIN
PRINT *,'TEST' ①
END
```

図 3-1-1

```
time a.out ↵  ②

 TEST              ③  プログラムからの出力

real    0m0.12s   ④  経過時間
user    0m0.01s   ⑤  ユーザーCPU時間
sys     0m0.01s   ⑥  システムCPU時間
```

図 3-1-2

上記の例では、プログラムからの出力と time コマンドの結果はディスプレイに表示されましたが、ファイル、あるいはディスプレイとファイルの両方に書き出すこともできます。なお、time コマンドの出力形式は UNIX や Linux のシェル環境によって若干異なります。

以上をまとめると図 3-1-3 のようになります (本例で経過時間は 1 分 11 秒 11、ユーザー CPU 時間は 1 分とします)。図中の出力先の〓はディスプレイ、〓はファイル (ファイル名は outlist) を表します。

シェル	timeコマンド	出力先	表示形式
bash ksh	time a.out ↵	〓	real　1m11.11s　経過時間 user　1m00.00s　ﾕｰｻﾞｰCPU時間 sys　0m0.01s　ｼｽﾃﾑCPU時間
	(time a.out) > outlist 2>&1 ↵	〓	
	(time a.out) 2>&1 | tee outlist ↵	〓	
	(time a.out) > outlist 2>&1 ↵ tail -f outlist ↵ (別の画面で)	〓 〓	
csh tcsh	time a.out ↵	〓	60.0u 0.0s 1:11 85% 0+0k 0+0io 0pf+0w └経過時間 └ｼｽﾃﾑCPU時間 └ﾕｰｻﾞｰCPU時間
	(time a.out) > outlist ↵	〓	
	(time a.out) | tee outlist ↵	〓	
	(time a.out) > outlist ↵ tail -f outlist ↵ (別の画面で)	〓 〓	

図 3-1-3

ジョブ全体のパフォーマンスを測定する場合の注意

- 経過時間を測定する場合、他のジョブが流れていないことを確認して下さい。また CPU 時間のみを測定する場合も、なるべく他のジョブが流れていない状態で測定した方がよいでしょう。
- リモートのファイルサーバーに対して I/O を行うと I/O 時間が多くかかるので、I/O が多いジョブの場合、なるべくローカルディスクに対して I/O を行って下さい。

3-2　プログラムの一部分のパフォーマンス測定方法

プログラムの一部分の経過時間や CPU 時間を測定する場合、タイマールーチンを使用します。経過時間測定用と CPU 時間測定用のタイマールーチンがありますので、用途に応じて選択して下さい。

経過時間／CPU 時間の測定方法 (Fortran)

Fortran の場合、具体的な使用方法はマシン環境によって異なりますので、マニュアルなどを参照して下さい。以下に一般的な使用方法と注意点を示します。なお、Fortran 90 の規格に「SYSTEM_CLOCK」（経過時間測定用）が、Fortran 95 の規格に「CPU_TIME」(CPU 時間測定用) が、追加されたそうです。

- 図 3-2-1 と図 3-2-2 に示すように、測定したい部分の前後 (②と③) にタイマールーチンを挿入します。タイマールーチンには以下の 2 種類があります。
  - タイマールーチンが呼ばれた時点の時刻を返すタイプ。図 3-2-1 の④の下線部が測定時間になります。
  - 1 つ前にそのタイマールーチンが呼ばれた時刻からの時間を返すタイプ。図 3-2-2 の⑤の下線部が測定時間になります。
- タイマールーチンから戻る値の型（倍精度実数、単精度実数、整数）に応じて、①で変数 (本例では TIM1, TIM2, TIM) を宣言して下さい。なお、タイマールーチンがサブルーチン形式 (CALL xxx) ではなく関数形式の場合、①の下線部に示すように、タイマールーチン名の宣言が必要な場合もあります（マニュアルの例参照)。
- 単位が「秒」でない (例えば 1/100 秒) タイマールーチンもありますので、④または⑤で調整して下さい。

```
   :
REAL*8 TIM1,TIM2,(タイマールーチン)  ①
TIM1 = (タイマールーチン)            ②
      測定したい部分
TIM2 = (タイマールーチン)            ③
PRINT *,'TIME = ',TIM2-TIM1,'SEC'    ④
   :
```

図 3-2-1

```
   :
REAL*8 TIM,(タイマールーチン)        ①
TIM = (タイマールーチン)             ②
      測定したい部分
TIM = (タイマールーチン)             ③
PRINT *,'TIME = ',TIM,'SEC'          ⑤
   :
```

図 3-2-2

経過時間／CPU 時間の測定方法 (C 言語)

C 言語で経過時間を測定する方法を図 3-2-3 に示します。①の 2 重線で C 言語の関数を使用して経過時間を求めた後、倍精度の秒に変換します。測定したい部分の前後に②と③を挿入し、④で両者の差を求めて経過時間を表示します。CPU 時間を測定する場合は図 3-2-4 のようになります。他に clock、times などの関数でも時間を測定することができますが、説明は省略します。

```
#include<sys/time.h>
#include<stdio.h>
double gettimeofday_sec() {
struct timeval tv;
gettimeofday(&tv, NULL);                          ①
return tv.tv_sec + (double)tv.tv_usec*1e-6;
}
main(){
double elp1,elp2;
elp1 = gettimeofday_sec();                        ②
 経過時間を測定したい部分
elp2 = gettimeofday_sec();                        ③
printf("ELAPSED TIME(sec) = %.6f¥n",elp2-elp1);
}                                                 ④
```

図 3-2-3　経過時間

```
#include<sys/time.h>
#include<sys/resource.h>
#include<stdio.h>
double getrusage_sec() {
struct rusage RU;
getrusage(RUSAGE_SELF, &RU);                      ①
return RU.ru_utime.tv_sec
       + (double)RU.ru_utime.tv_usec*1e-6;
}
main(){
double cpu1,cpu2;
cpu1 = getrusage_sec();                           ②
 CPU時間を測定したい部分
cpu2 = getrusage_sec();                           ③
printf("CPU TIME(sec) = %.6f¥n",cpu2-cpu1);
}                                                 ④
```

図 3-2-4　CPU 時間

3-3　パフォーマンス測定の考慮点

人工的なプログラムでのパフォーマンス測定

図 3-3-1 の⑦のように人工的なプログラムのパフォーマンスを測定する場合、以下の点に注意します。

- 測定したい④の部分の前後に、②，⑤のタイマールーチンを挿入して計算時間 (通常は CPU 時間) を測定します。計算時間が短すぎるときは、③の DO ループを付加して計算時間を長くします。
- マシン環境によっては、1 回目のサブルーチンコールだけ、計算時間が若干遅くなる場合があります。そのときは、①で一度ダミーのサブルーチンコールを行ってから実際の測定を行います。また、同じジョブを複数回実行すると、計算時間が若干変動することがありますので、何回か実行した方がよいでしょう。
- (⑥の下線部がない場合)、コンパイラの種類と最適化のレベルによっては、コンパイラが、⑦で計算した配列 A の値を、その後の計算で使用していないことを見抜き、⑦を不要な計算とみなして削除することがあります。すると計算時間はゼロ秒になって、正しい計算時間を測定することができません。このような場合、⑥の下線部を追加して、配列 A を使用しているように見せかければ、コンパイラは⑦を削除せず、正しい計算時間を測定できるようになります。

```
PARAMETER(N=10000000)
REAL*8 A(N),B(N),CPU1,CPU2
DO I=1,N
  B(I) = I
ENDDO
CALL SUB(A,B,N)                    ①
CPU1 = タイマールーチン            ②
DO K=1,10                          ③
  CALL SUB(A,B,N)                  ④
ENDDO
CPU2 = タイマールーチン            ⑤
PRINT *,'CPU = ',CPU2-CPU1,A(1)    ⑥
END
```

```
SUBROUTINE SUB(A,B,N)
REAL*8 A(N),B(N)
DO I=1,N
  A(I) = B(I)**2.1                 ⑦
ENDDO
END
```

図 3-3-1

コンパイラによるループ順序の自動入れ替え

スカラー計算機（パソコンやワークステーション）の場合、図 3-3-2 (1) の 2 重ループは 4 章で説明するキャッシュミスが発生してパフォーマンスが低下する可能性があるため、コンパイラの種類や最適化のレベルによっては図 3-3-2 (2) のように DO ループの順番を自動的に入れ替えることがあります。

図 3-3-2 (1) の状態で CPU 時間を測定したい場合のように、何らかの理由で 2 重ループの順番を入れ替えたくない場合は、図 3-3-2 (3) の③，④のようにダミーのサブルーチンをコールすれば、コンパイラは一般に自動的に入れ替えを行わなくなります。

```
   :
DO I=1,M
  DO J=1,N
    A(I,J) = ～
  ENDDO
ENDDO
   :
```

図 3-3-2 (1)

→

```
   :
DO J=1,N
  DO I=1,M
    A(I,J) = ～
  ENDDO
ENDDO
   :
```

図 3-3-2 (2)

```
   :
DO I=1,M
  CALL DUMMY(A,M,N)                ③
  DO J=1,N
    A(I,J) = ～
  ENDDO
ENDDO
   :
```

```
SUBROUTINE DUMMY(A,M,N)
DIMENSION A(M,N)                   ④
END
```

図 3-3-2 (3)

MFLOPS（メガフロップス）値の求め方

マシン性能を示す指標として MFLOPS（メガフロップス）値、あるいは GFLOPS（ギガフロップス）値を使用することがあります。実行したジョブの MFLOPS 値を知ることができるマシン環境もありますが、できないマシン環境の場合でも、MFLOPS 値を調べたい箇所のロジックが簡単であれば、次のようにして調べることができます。

1 秒間に 1000000 回の浮動小数点演算 (実数の $+ - \times \div$) を実行した場合を 1 MFLOPS といいます。図 3-3-3 では乗算と加算をそれぞれ 3000 回実行しているので、この DO ループにかかった CPU 時間を t 秒とすると、

$$\text{MFLOPS 値} = (3000 \times 2) \div t \div 1000000$$

となります。なお、整数の計算のみを行っている場合は、上記の定義から MFLOPS 値はゼロとなります。

```
   :
DO I = 1, 3000
  A(I) = B(I)*C(I) + D(I)
ENDDO
   :
```

図 3-3-3

3-4　ホットスポットの特定方法

3-4-1　ホットスポットを特定するためのツール

ホットスポットを特定するためのツール（プロファイラまたはアナライザと呼びます）として、UNIX や Linux では一般に次の 2 つが提供されており、Fortran と C で使用することができます。ただしマシン環境によっては提供されていない場合もあります。

- prof: サブルーチンごとに、CPU 時間（および比率）とコールされた回数が表示されます。
- gprof: prof と同様の表示の他に、各サブルーチンがどのサブルーチンから何回コールされ、どのサブルーチンを何回コールしたかが表示されます。

上記のツールは、「どのサブルーチンに時間がかかっているか」しか分かりません。ホットスポットのサブルーチン内の、さらにどの DO ループがホットスポットなのかを知りたい場合は、3-2 節で説明したタイマールーチンを各 DO ループの前後に挿入して調べて下さい。

また、マシン環境によっては、以下のような情報の分かるツールが提供されている場合もありますので、マニュアルを参照して下さい。

- DO ループ、あるいはステートメントにかかっている時間や通過回数
- DO ループの反復回数
- IF 文の真と偽の割合

次節以降で、図 3-4-1 のプログラム (sample.f というファイルに入っているとします) を例に、各ツールの使用方法および、作成される報告書の見方について説明します。なお、図 3-4-1 中の下線はサブルーチンがコールされた回数、二重線は「←」のステートメントが実行された回数を示します。

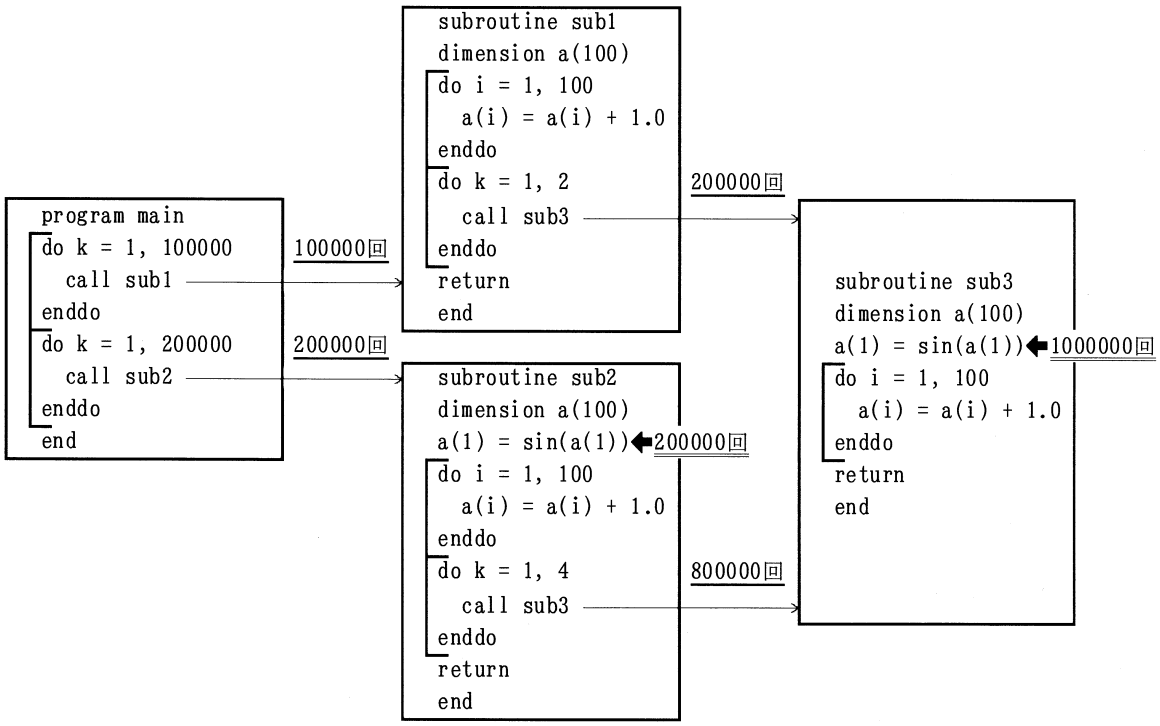


図 3-4-1

3-4-2　prof コマンドの使用方法

使用方法

使用方法はマシン環境によって異なるので一例を示します。

- ①のように、コンパイルオプション -p、および任意の最適化オプションをつけてコンパイルします。なお、-p オプションをつけるとパフォーマンスが低下する可能性があるので、prof で分析するときにのみ使用して下さい。
- ②のようにジョブを実行します。ジョブが終了すると mon.out というバイナリーファイルが作成されます。
- ③のように prof コマンドを実行すると、mon.out から報告書が作成されます。そのままではディスプレイに表示されてしまうので、例えば outlist というファイルに書き出します。ロードモジュール名が a.out のときは、③の a.out は省略可能です。なお、prof コマンドの詳細は、man コマンドで調べることができます。

```
f77 -p (コンパイルオプション) sample.f ↵  ･･･ ①
a.out ↵                                   ･･･ ②
prof a.out > outlist ↵                    ･･･ ③
```

作成される報告書

prof で作成される報告書の例を図 3-4-2 に示します。

- サブルーチン名が、CPU 時間の多い順に一番左の欄に表示されます。なお、下位の方にプログラムに含まれないサブルーチン (catopen など) が表示されますが、これらは内部的にコールされるシステムルーチンなので、無視してかまいません。
- __mcount というサブルーチンが上位に現れますが、このルーチンの CPU 時間は、prof の測定用ルーチンが稼動した時間で、本来は存在しない時間なので無視してかまいません。
- 図 3-4-1 に示すように、この例で組込関数 sin はサブルーチン sub2 と sub3 で使用されていますが、図 3-4-2 では _sin として独立した CPU 時間になっており、sub2 と sub3 の CPU 時間には含まれないので注意して下さい。
- 組込関数 sin がどのサブルーチンから何回コールされたのかを知りたい場合、次節に示す gprof を使用して下さい。

prof の報告書から以下の点を調べます。

- CPU 時間の比率が多い（ホットスポットの）サブルーチンを調べます。本例では sub3、sub2 の順にホットスポットになっているので、これらのサブルーチンをチューニングの対象とします。
- 他のルーチンと比べて、コールされた回数が非常に多いサブルーチンがないか調べます (なお、コールされた回数が多過ぎると、prof の内部的なカウンターがオーバーフローして、負の大きな値が表示されることがあります)。コール回数が非常に多いルーチンは、インライン展開 (コールする側のルーチンに埋め込むこと：5-3 節参照) するとパフォーマンスが向上する場合があります。

サブルーチン名	CPU時間の比率	CPU時間	CPU時間の累計	コールされた回数	1回のコールあたりの平均CPU時間
Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.sub3	64.2	35.17	35.17	1000000	0.0352
.sub2	12.6	6.88	42.05	200000	0.0344
._sin	8.7	4.76	46.81	1200000	0.0040
.__mcount	8.5	4.63	51.44		
.sub1	6.0	3.26	54.70	100000	0.0326
.main	0.1	0.06	54.76	1	60.
.catopen	0.0	0.00	54.76	1	0.
以 下 省 略					

図 3-4-2

3-4-3　gprof コマンドの使用方法

gprof は prof と同様の報告書に加え、各サブルーチンがどのサブルーチンから何回コールされ、どのサブルーチンを何回コールしているか等の報告書を作成します。従って、prof と gprof の両方を使用できる場合は、gprof を使用した方がよいでしょう。

使用方法

使用方法はマシン環境によって異なるので一例を示します。

- ①のように、コンパイルオプション -pg、および任意の最適化オプションをつけてコンパイルします。なお、-pg オプションをつけるとパフォーマンスが低下する可能性があるので、gprof で分析するときにのみ使用して下さい。
- ②のようにジョブを実行します。ジョブが終了すると gmon.out というバイナリーファイルが作成されます。
- ③のように gprof コマンドを実行すると、gmon.out から報告書が作成されます。そのままではディスプレイに出てしまうので、例えば outlist というファイルに書き出します。ロードモジュール名が a.out のときは、③の a.out は省略可能です。なお、gprof コマンドの詳細は、man コマンドで調べることができます。

```
f77 -pg（コンパイルオプション）sample.f ↵   ･･･ ①
a.out ↵                                      ･･･ ②
gprof a.out > outlist ↵                      ･･･ ③
```

作成される報告書 (1)

gprof で作成される図 3-4-3 の報告書は図 3-4-2 とほぼ同様なので、説明は省略します。

(*1) 1 回のコールあたりの平均 CPU 時間（そのサブルーチンのみ）

(*2) 1 回のコールあたりの平均 CPU 時間（そのサブルーチンと、それより下位の全サブルーチン）

```
CPU時間   CPU時間          コール
の比率    の累計  CPU時間 された回数  (*1)     (*2)   サブルーチン名
 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
58.3      36.00    36.00  1000000     0.04     0.04  .sub3 [4]
16.4      46.10    10.10                             .__mcount [6]
12.2      53.62     7.52   200000     0.04     0.20  .sub2 [3]
 6.2      57.44     3.82   100000     0.04     0.12  .sub1 [5]
 5.7      60.97     3.53  1200000     0.00     0.00  ._sin [7]
 0.8      61.49     0.52                             ._getcall2 [8]
 0.3      61.66     0.17                             ._getcall [9]
 0.1      61.75     0.09        1    90.00 50960.00  .main [1]
 0.0      61.75     0.00        3     0.00     0.00  .splay [10]
 0.0      61.75     0.00        2     0.00     0.00  .catclose [11]
 0.0      61.75     0.00        2     0.00     0.00  .free [12]
 0.0      61.75     0.00        2     0.00     0.00  .free_y [13]
 0.0      61.75     0.00        1     0.00     0.00  .catopen [14]
 0.0      61.75     0.00        1     0.00     0.00  .exit [15]
 0.0      61.75     0.00        1     0.00     0.00  .moncontrol [16]
        以 下 省 略
```

図 3-4-3

作成される報告書 (2)

gprof のもう 1 つの報告書を図 3-4-4 に示し、この図のまとめを図 3-4-5 に示します。図 3-4-4 と図 3-4-5 内の数字は対応しています。

図 3-4-4 内の [1]，[2]，... の数字は、各ルーチンを区別するために gprof が付けた識別子で、例えば [1] は main を表します。以下に報告書の主な部分を説明します。

- 例えば〓で囲んだ部分の右端の欄のサブルーチン名を見ると、「sub3」が左に寄っています。これは、〓で囲んだ部分がサブルーチン sub3 を中心とした情報であることを意味します。
- 〓内の⑩より上にある 8 と 9 の 2 行（二重線で示します）は、sub3 をコールしているルーチンに関する情報です。8 は sub3 が sub1 から 200000 回コールされ、そのとき sub3 で CPU 時間が合計 7.20 秒かかったことを意味し、9 は sub3 が sub2 から 800000 回コールされ、そのとき sub3 で CPU 時間が合計 28.80 秒かかったことを示します。
- 〓内の⑩（太線で示します）は sub3 そのものに関する情報で、sub3 が合計 1000000 回 (= 200000 + 800000 回) コールされ、CPU 時間が合計 36.00 秒 (= 7.20 + 28.80 秒) かかったことを示します。
- 〓内の⑩より下にある (11)（波線で示します）は、sub3 からコールしているルーチンに関する情報です。(11) は sub3 が組込関数 sin を 1000000 回コールし、そのとき sin で CPU 時間が合計 2.94 秒かかったことを示します。

他の部分の見方も同様です。

図 3-4-3 で sin の CPU 時間は 3.53 秒、比率は 5.7% でした。例えばこの sin がどのサブルーチンに含まれる sin なのかを知りたい場合、本報告書を使用します。図 3-4-4 の 15，16 を見ると、sub3 に含まれる sin が CPU 時間を多く (2.94 秒) 使用していることが分かります。

また報告書は、プログラム内に含まれるサブルーチン間の関連図を作成する場合にも便利です。関連図は、プログラムを MPI で並列化する際、どの部分を並列化するかを検討する場合などに使用します。

```
                                     called/total       parents
index  %time    self descendents      called+self    name    index
                                     called/total       children

                0.09        50.87         1/1          .__start [2]
[1]     82.5    0.09        50.87           1          .main [1]       ◆ ① mainの情報
                7.52        31.74    200000/200000     .sub2 [3]       ⇦ (2) mainからｺｰﾙするﾙｰﾁﾝ
                3.82         7.79    100000/100000     .sub1 [5]       ⇦ (3) mainからｺｰﾙするﾙｰﾁﾝ
-----------------------------------------------
6.6s                                                   <spontaneous>
[2]     82.5    0.00        50.96                     .__start [2]
                0.09        50.87         1/1          .main [1]
-----------------------------------------------
                7.52        31.74    200000/200000     .main [1]       ← 4 sub2をｺｰﾙするﾙｰﾁﾝ
[3]     63.6    7.52        31.74    200000           .sub2 [3]       ◆ ⑤ sub2の情報
               28.80         2.35    800000/1000000    .sub3 [4]       ⇦ (6) sub2からｺｰﾙするﾙｰﾁﾝ
                0.59         0.00    200000/1200000    ._sin [7]       ⇦ (7) sub2からｺｰﾙするﾙｰﾁﾝ
-----------------------------------------------
                7.20         0.59    200000/1000000    .sub1 [5]       ← 8 sub3をｺｰﾙするﾙｰﾁﾝ
               28.80         2.35    800000/1000000    .sub2 [3]       ← 9 sub3をｺｰﾙするﾙｰﾁﾝ
[4]     63.1   36.00         2.94   1000000           .sub3 [4]       ◆ ⑩ sub3の情報
                2.94         0.00   1000000/1200000    ._sin [7]       ⇦ (11) sub3からｺｰﾙするﾙｰﾁﾝ
-----------------------------------------------
                3.82         7.79    100000/100000     .main [1]       ← 12 sub1をｺｰﾙするﾙｰﾁﾝ
[5]     18.8    3.82         7.79    100000           .sub1 [5]       ◆ ⑬ sub1の情報
                7.20         0.59    200000/1000000    .sub3 [4]       ⇦ (14) sub1からｺｰﾙするﾙｰﾁﾝ
-----------------------------------------------
6.6s                                                   <spontaneous>
[6]     16.4   10.10         0.00                     .__mcount [6]
-----------------------------------------------
                0.59         0.00    200000/1200000    .sub2 [3]       ← 15 sinをｺｰﾙするﾙｰﾁﾝ
                2.94         0.00   1000000/1200000    .sub3 [4]       ← 16 sinをｺｰﾙするﾙｰﾁﾝ
[7]      5.7    3.53         0.00   1200000           ._sin [7]       ◆ ⑰ sinの情報
-----------------------------------------------
        以 下 省 略
```

図 3-4-4

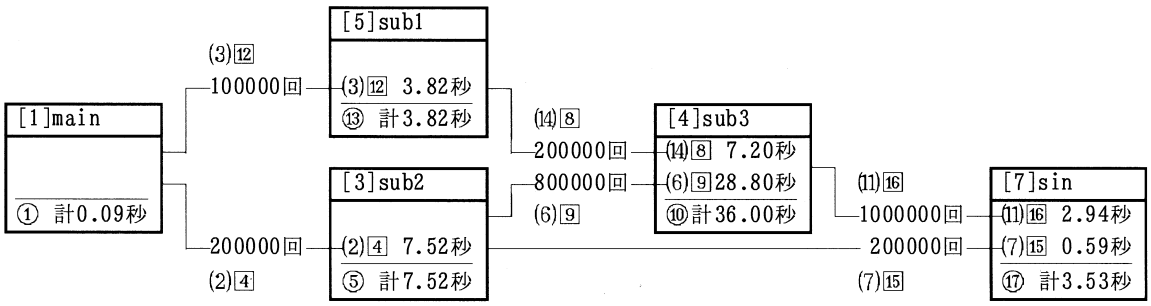


図 3-4-5

第4章　キャッシュチューニング

「キャッシュ」という名前は知っていても、どういう機能を持つのかよく分からないという人がいます。しかしキャッシュはパフォーマンスに大きく影響するので、チューニングでは非常に重要です。本章では、キャッシュを効率よく利用するためのプログラム上の考慮点について説明します。

4-1　キャッシュとは

本章で説明するキャッシュの考慮点は、スカラー計算機（ワークステーションやパソコン）を対象とし、スーパーコンピュータ（ベクトル計算機）は対象としないので注意して下さい。

プログラムのチューニングを行う場合、キャッシュそのものの動作もある程度は知っている方が融通がきくので、本節ではまずキャッシュの構造について簡単に説明します。ただし、本書はハードウェアの解説書ではないため、下記の説明には不正確な部分があることをご了承下さい。

記憶装置の階層構造

プログラムの実行を開始すると、図 4-1-1 (1) に示すように、ロードモジュール a.out 内の機械語命令（図中の「命」）とデータ（図中の「デ」）はメモリーにロードされます。次に、機械語命令は命令レジスターに転送されて命令の解読が行われ、データはデータレジスターに転送されて演算が行われます。

ここで、なぜメモリーとレジスターの 2 段構成になっているのかを考えてみましょう。図 4-1-1 (1) でレジスターに使用される材料はメモリーに使用される材料よりも高速（処理速度が速い）なので、例えば図 4-1-1 (2) のように高速な材料のみでメモリーを作成してしまえば、（極論すると）レジスターが不要となります。ところが高速な材料は値段が高いため、図 4-1-1 (2) では価格が著しく高くなってしまいます。

そこで、価格はなるべく低く抑え、しかも高速な材料を使用したのと同等に近いスピードを出すため、メモリーとレジスターの 2 段構成になっているのです。メモリーは低速／安価な材料で作成して大容量とし、レジスターは高速／高価な材料で作成して小容量とします。そしてデータや機械語命令はいったん全てメモリーに入れておき、そのうち重要な部分（現在処理中の部分）のみをレジスターに入れて高速に処理します。

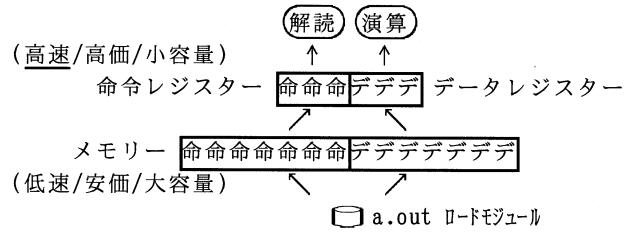


図 4-1-1 (1)

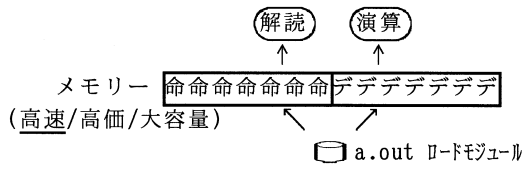


図 4-1-1 (2)

次にキャッシュについて説明します。多くの計算機では、図 4-1-1 (3) に示すように、メモリーとレジスターの間にキャッシュと呼ばれる装置が入っています。キャッシュはメモリーとレジスターの中間的な性質を持ち、メモリー内の機械語命令やデータのうち、必要性の高い部分がキャッシュに入り、さらに必要性の高い部分がレジスターに入ります。

キャッシュは、レジスターと同様に、機械語命令が入る命令キャッシュとデータが入るデータキャッシュに分かれています。このうちチューニングに関係があるのは主にデータキャッシュなので、本章では以後点線で囲んだ部分のデータの動きについて説明し、命令キャッシュについては 5-3 節で簡単に紹介します。

余談ですが、ディスクやテープはさらに値段が安くて低速なので、それらを含めると計算機の記憶装置は図 4-1-1 (3) のようなピラミッド型の階層構造になっており、必要性の高いデータほど上に入ります。日常生活でも必要性の高いものは身近に置き、必要性が低くめったに使用しないものは物置などに入れておきますが、図 4-1-1 (3) の階層構造はそれとよく似ています。

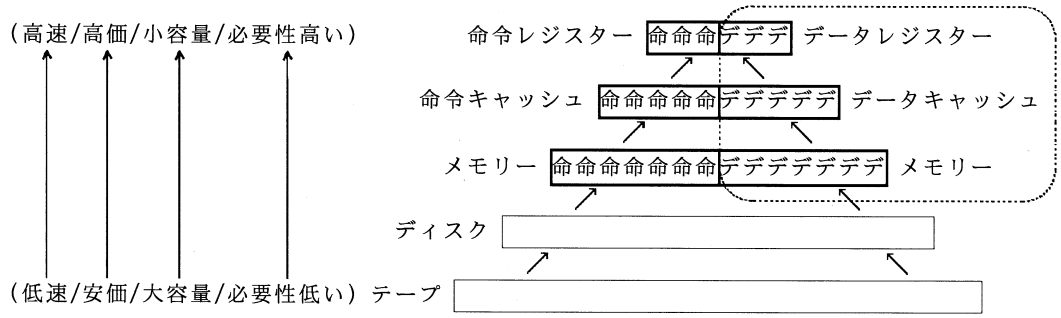


図 4-1-1 (3)

レジスター ↔ キャッシュ ↔ メモリー間のデータの転送

図 4-1-1 (3) の点線で囲んだ部分の動作について、図 4-1-2 (1) のプログラムを例に説明します。簡単のため、実際の装置よりもサイズを大幅に縮小した図 4-1-2 (2) の簡易モデルで説明します。

- 図 4-1-2 (1) のプログラムを開始すると、配列 A(1)～A(36) がメモリー上に連続して配置されます。図では配列 A は左上の端から開始していますが、どこから開始するかはそのときの状況によって異なります。
- この簡易モデルではキャッシュの段数は 2 段です。各段は〓で囲んだ小区画（キャッシュラインまたはキャッシュ線）に分かれています。またメモリー内も、同様に〓で囲んだ小区画に分かれています。
- 図 4-1-2 (1) の DO ループの反復が I=1 のとき、A(1) が計算に必要となり、A(1) はまずメモリーからキャッシュに転送されます。このとき①に示すように、A(1) だけでなく、A(1) が所属する〓で囲んだ小区画に含まれるすべての要素 (A(1)～A(3)) がまとめてキャッシュに転送されます。転送されたデータは 1 のキャッシュ線の 1 段目か 2 段目のいずれか空いている方に入り、2 または 3 に入ることはできません。(理解を深めるためにもう一つ例を挙げます。例えば A(33) が必要になった場合は、A(33) が所属する〓で囲んだ小区画に含まれる全ての要素 (A(31)～A(33)) がまとめて 2 のキャッシュ線のいずれか空いている方に転送されます。)
- 次に②に示すように、A(1) がキャッシュからレジスターに転送されて計算が行われ、計算後の A(1) はキャッシュ内の A(1) に戻されます。この時点ではキャッシュ内の A(1) には計算後の値が、メモリー内の A(1) には計算前の値が入っています。前述のように、キャッシュはメモリーよりも高速なので、②は①より高速に転送されます。なお、本例ではレジスターは 1 つしかありませんが、実際にはもっとたくさんあります。
- DO ループの反復が I=2 となり、A(2) が計算に必要となります。A(2) は既にキャッシュに入っているので、メモリーからではなく、キャッシュからレジスターに転送されます (③)。次に I=3 となり、A(3) も同様にキャッシュからレジスターに転送されます (④)。
- 次に DO ループの反復が I=4 となり、A(4) はキャッシュ上にないので A(4)～A(6) がキャッシュ線 2 の 1 段目にまとめて転送され、処理が行われます。同様に A(7)～A(9) がキャッシュ線 3 の 1 段目に転送されます。この時点でキャッシュの 1 段目は満杯になります。同様に、A(10)～A(18) がキャッシュの 2 段目の 1～3 に転送され、キャッシュの 2 段目も満杯になります。この時点での状態を図 4-1-2 (3) に示します。
- 次に A(19) が必要となり、A(19)～A(21) がキャッシュに転送されますが、キャッシュ線 1 は既に 2 段とも満杯になっています。この場合、古い方のデータ (本例では 1 段目の A(1)～A(3)) がキャッシュから追い出され (⑤)、そこに A(19)～A(21) が転送されます (⑥)。なお、追い出されたデータはメモリー内にある値と異なっていればメモリーに戻されます。以後、同様の処理が行われます。

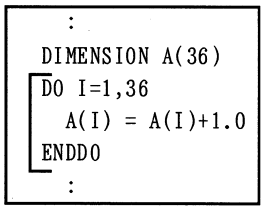


図 4-1-2 (1)

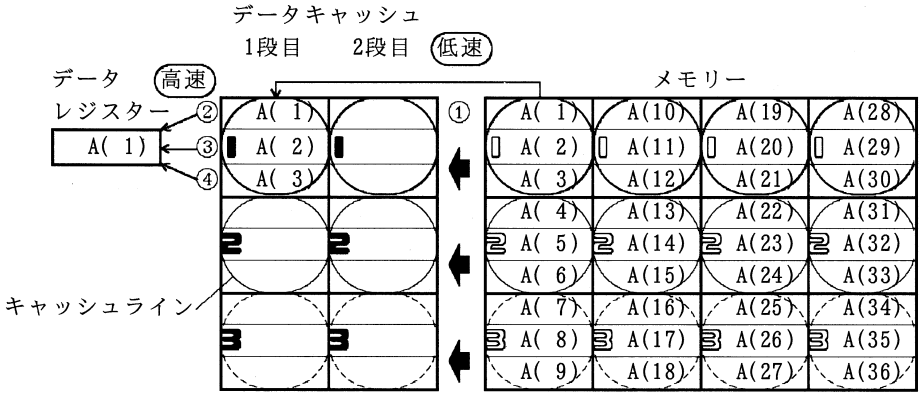


図 4-1-2 (2)

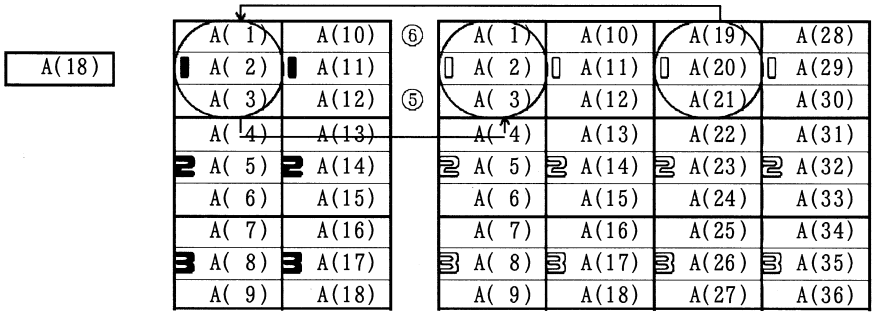


図 4-1-2 (3)

4-2　キャッシュミスを少なくするための考慮点

キャッシュミスとは

計算に必要なデータがその時点でキャッシュに入っていないことをキャッシュミスと呼びます。キャッシュミスが発生すると、メモリーからキャッシュへデータを転送せざるを得なくなりますが、図 4-1-2 (2) の①は②と比べて転送速度が遅いため、パフォーマンスは低下します。このため良好なパフォーマンスを得るためにはなるべくキャッシュミスを少なくする必要があります。もちろんデータは最初キャッシュには入っておらずメモリーのみに入っているため、キャッシュミスをゼロにすることはできませんが、一度キャッシュに入ったデータをできるだけ有効利用するのがポイントになります。

1 次元配列 (1 重ループ) とキャッシュミス

まず 1 次元配列 (1 重ループ) の場合を説明します。図 4-2-1 (1) のプログラムを実行したときのメモリーの様子を図 4-2-1 (2) に示します。○と□内の数字は各要素がアクセスされる順序を示します。ある要素と、その次に処理する要素の、メモリー上での距離をストライドと呼びます。図 4-2-1 (1) (2) ではストライドは 1 です。ストライドが 4 の例を図 4-2-2 (1) (2) に示します。

それではこの 2 つの例でどちらがキャッシュミスが少ない（パフォーマンスが良い）でしょうか？　まずストライドが 1 の図 4-2-1 (1) (2) では、A(1) がアクセスされたときにキャッシュミスが発生し、A(2) と A(3) では発生しません。次に A(4) でキャッシュミスが発生し、A(5) と A(6) では発生しません。このように、□の要素でキャッシュミスが発生し、○の要素では発生しません。

一方ストライドが 4 の図 4-2-2 (1) (2) では、A(1) がアクセスされたときにキャッシュミスが発生し、A(5) でまたキャッシュミス、A(9) でまたキャッシュミスと、全ての要素で必ずキャッシュミスが発生します。言いかえると、せっかく例えば A(1) とともにキャッシュに入った A(2)、A(3) はその後使用されておらず、キャッシュが有効利用されていないと言うことができます。なお、本例ではストライドがキャッシュ線の長さよりも長いため、必ずキャッシュミスが発生しました。ストライドがキャッシュ線より短い場合には、必ず発生する訳ではありませんが、ストライドが長くなるにつれてキャッシュミスの頻度が次第に高くなります。

以上をまとめると、1 次元配列 (1 重ループ) の場合の考慮点は次のようになります。なお、実際のプログラムでは、図 4-2-2 (1) のように 1 次元配列でストライドが大きい DO ループは、高速フーリエ変換など特別な場合以外には少なく、キャッシュミスが問題になるのは次節で説明する多次元配列（多重ループ）の場合です。

1 次元配列 (1 重ループ) の場合、キャッシュミスを少なくするためには、ストライドを短くすること。

```
 DIMENSION A(36)
 DO I=1,36
   A(I) = A(I)+1.0
 ENDDO
   :
```

図 4-2-1 (1)　○　1 次元、ストライド 1

```
 DIMENSION A(36)
 DO I=1,36,4
   A(I) = A(I)+1.0
 ENDDO
   :
```

図 4-2-2 (1)　×　1 次元、ストライド 4

ｽﾄﾗｲﾄﾞ1

①A(1)	⑩A(10)	⑲A(19)	㉘A(28)
②A(2)	⑪A(11)	⑳A(20)	㉙A(29)
③A(3)	⑫A(12)	㉑A(21)	㉚A(30)
④A(4)	⑬A(13)	㉒A(22)	㉛A(31)
⑤A(5)	⑭A(14)	㉓A(23)	㉜A(32)
⑥A(6)	⑮A(15)	㉔A(24)	㉝A(33)
⑦A(7)	⑯A(16)	㉕A(25)	㉞A(34)
⑧A(8)	⑰A(17)	㉖A(26)	㉟A(35)
⑨A(9)	⑱A(18)	㉗A(27)	㊱A(36)

図 4-2-1 (2)

ｽﾄﾗｲﾄﾞ4

①A(1)	A(10)	A(19)	A(28)
A(2)	A(11)	A(20)	⑧A(29)
A(3)	A(12)	⑥A(21)	A(30)
A(4)	④A(13)	A(22)	A(31)
②A(5)	A(14)	A(23)	A(32)
A(6)	A(15)	A(24)	⑨A(33)
A(7)	A(16)	⑦A(25)	A(34)
A(8)	⑤A(17)	A(26)	A(35)
③A(9)	A(18)	A(27)	A(36)

図 4-2-2 (2)

【Fortran の場合】多次元配列（多重ループ）とキャッシュミス

本書では、2 次元配列 (例えば A(4,9)) を図示する場合、図 4-2-5（一番下の図なので注意）に示すように、1 次元目を縦方向、2 次元目を横方向に描きます。この図は配列そのものの図で、メモリーの図ではないことに注意して下さい。

Fortran の場合、配列 A(4,9) は、(簡易モデルの) メモリー上では、図 4-2-3 (2) または図 4-2-4 (2) に示すように、A(1,1), A(2,1), A(3,1), ... のように左側の添字が先に動く順番に配置されます (3 次元以上の配列でも同様)。図 4-2-3 (2)、図 4-2-4 (2)、図 4-2-5 の矢印は同じ要素の部分を示します。図 4-2-5（配列の図）でメモリー上の要素の並びを知りたい場合は、図の矢印に示すように縦方向に並んでいると考えて下さい。

さて、図 4-2-3 (1) と図 4-2-4 (1) の 2 重ループは同じ計算をしていますが、ループの順番が逆になっています。2 重ループを実行したとき、メモリー上の各要素がアクセスされる順序を、図 4-2-3 (2) と図 4-2-4 (2) の□と○の数字で示します。図 4-2-3 (2) はストライドが 1 で、図 4-2-4 (2) はストライドが 4 でアクセスされており、それぞれ前のページの図 4-2-1 (2)、図 4-2-2 (2) と全く同じ順序で処理が行われています。従って前ページの説明と同じ理由で、□の要素でキャッシュミスが発生し、○の要素では発生しないので、図 4-2-3 (1) (2) の方がキャッシュミスは少なくなります。すなわち、Fortran では、2 次元配列がメモリー上に、左側の添字が先に動く順番に配置されているので、内側のループを配列の左側の添字で反復させればストライドが 1 になり、キャッシュミスが少なくなります。なお、図 4-2-4 (1) (2) でストライドが 4 になっているのは、配列 A(4,9) の 1 次元目の大きさが 4 だからです。

以上をまとめると、多次元配列（多重ループ）の場合の考慮点 (Fortran の場合) は以下のようになります。

【重要】多次元配列（多重ループ）の場合、キャッシュミスを少なくするためには、Fortran では、図 4-2-3 (1) に示すように、内側のループを配列の左側の添字で反復させること。

```
  DIMENSION A(4,9)
  DO J=1,9
    DO I=1,4
      A(I,J) = A(I,J)+1.0
    ENDDO
  ENDDO
    :
```

図 4-2-3 (1)　○　2 次元、ストライド 1

ｽﾄﾗｲﾄ1	[1]A(1,1)	[10]A(2,3)	[19]A(3,5)	[28]A(4,7)
	②A(2,1)	⑪A(3,3)	⑳A(4,5)	㉙A(1,8)
	③A(3,1)	⑫A(4,3)	㉑A(1,6)	㉚A(2,8)
	[4]A(4,1)	[13]A(1,4)	[22]A(2,6)	[31]A(3,8)
	⑤A(1,2)	⑭A(2,4)	㉓A(3,6)	㉜A(4,8)
	⑥A(2,2)	⑮A(3,4)	㉔A(4,6)	㉝A(1,9)
	[7]A(3,2)	[16]A(4,4)	[25]A(1,7)	[34]A(2,9)
	⑧A(4,2)	⑰A(1,5)	㉖A(2,7)	㉟A(3,9)
	⑨A(1,3)	⑱A(2,5)	㉗A(3,7)	㊱A(4,9)

図 4-2-3 (2)　メモリーの図 (Fortran)

```
  DIMENSION A(4,9)
  DO I=1,4
    DO J=1,9
      A(I,J) = A(I,J)+1.0
    ENDDO
  ENDDO
    :
```

図 4-2-4 (1)　×　2 次元、ストライド 4

	[1]A(1,1)	A(2,3)	A(3,5)	A(4,7)
	[10]A(2,1)	A(3,3)	A(4,5)	[8]A(1,8)
ｽﾄﾗｲﾄ4	A(3,1)	A(4,3)	[6]A(1,6)	A(2,8)
	A(4,1)	[4]A(1,4)	A(2,6)	A(3,8)
	[2]A(1,2)	A(2,4)	A(3,6)	A(4,8)
	[11]A(2,2)	A(3,4)	A(4,6)	[9]A(1,9)
	A(3,2)	A(4,4)	[7]A(1,7)	A(2,9)
	A(4,2)	[5]A(1,5)	A(2,7)	A(3,9)
	[3]A(1,3)	A(2,5)	A(3,7)	A(4,9)

図 4-2-4 (2)　メモリーの図 (Fortran) ([12]以降は省略)

	J→ 2次元目								
I	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)	A(1,6)	A(1,7)	A(1,8)	A(1,9)
↓	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)	A(2,6)	A(2,7)	A(2,8)	A(2,9)
1次元目	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)	A(3,6)	A(3,7)	A(3,8)	A(3,9)
	A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)	A(4,6)	A(4,7)	A(4,8)	A(4,9)

図 4-2-5　配列の図 (Fortran)

【C 言語の場合】多次元配列（多重ループ）とキャッシュミス

C 言語の場合は、Fortran と結論が逆になります。本書の以後の節でも Fortran の例で説明しますので、C 言語のユーザーの方も、まず前ページを読んで下さい。以下では Fortran との相違点のみ簡単に説明します。

本書では、2 次元配列 (例えば a[9][4]) を図示する場合、図 4-2-8（一番下の図）に示すように、1 次元目を縦方向、2 次元目を横方向に描きます。これは配列そのものの図で、メモリーの図ではありません。

C 言語の場合、配列 a[9][4] は、（簡易モデルの）メモリー上では、図 4-2-6 (2) に示すように、a[0][0], a[0][1], a[0][2], ... のように右側の添字が先に動く順番に配置されます。従って、図 4-2-8（配列の図）でメモリー上の要素の並びを知りたい場合は、図の矢印に示すように横方向に並んでいると考えて下さい。

図 4-2-6 (1) と図 4-2-7 (1) を実行すると、メモリー上の各要素がアクセスされる順序は、図 4-2-6 (2) と図 4-2-7 (2) の□と○の数字のようになります。2 ページ前で説明したように、□の要素でキャッシュミスが発生し、○の要素では発生しないので、図 4-2-7 (1) (2) の方がキャッシュミスは少なくなります。

すなわち、C 言語では、2 次元配列がメモリー上に、右側の添字が先に動く順番に配置されているので、内側のループを配列の右側の添字で反復させればストライドが 1 になり、キャッシュミスが少なくなります。

以上をまとめると、多次元配列（多重ループ）の場合の考慮点 (C 言語の場合) は以下のようになります。

【重要】多次元配列（多重ループ）の場合、キャッシュミスを少なくするためには、C 言語では、図 4-2-7 (1) に示すように、内側のループを配列の右側の添字で反復させること。

```
float a[9][4];
for (j=0;j<4;j++) {
  for (i=0;i<9;i++) {
    a[i][j] = a[i][j]+1.0f;
  }
}
  :
```

図 4-2-6 (1)　×　2 次元、ストライド 4

```
float a[9][4];
for (i=0;i<9;i++) {
  for (j=0;j<4;j++) {
    a[i][j] = a[i][j]+1.0f;
  }
}
  :
```

図 4-2-7 (1)　○　2 次元、ストライド 1

ｽﾄﾗｲﾄ4

1 a[0][0]	a[2][1]	a[4][2]	a[6][3]
10 a[0][1]	a[2][2]	a[4][3]	8 a[7][0]
a[0][2]	a[2][3]	6 a[5][0]	a[7][1]
a[0][3]	4 a[3][0]	a[5][1]	a[7][2]
2 a[1][0]	a[3][1]	a[5][2]	a[7][3]
11 a[1][1]	a[3][2]	a[5][3]	9 a[8][0]
a[1][2]	a[3][3]	7 a[6][0]	a[8][1]
a[1][3]	5 a[4][0]	a[6][1]	a[8][2]
3 a[2][0]	a[4][1]	a[6][2]	a[8][3]

図 4-2-6 (2)　メモリーの図 (C 言語) (12以降は省略)

ｽﾄﾗｲﾄ1

1 a[0][0]	10 a[2][1]	19 a[4][2]	28 a[6][3]
② a[0][1]	⑪ a[2][2]	⑳ a[4][3]	㉙ a[7][0]
③ a[0][2]	⑫ a[2][3]	㉑ a[5][0]	㉚ a[7][1]
4 a[0][3]	13 a[3][0]	22 a[5][1]	31 a[7][2]
⑤ a[1][0]	⑭ a[3][1]	㉓ a[5][2]	㉜ a[7][3]
⑥ a[1][1]	⑮ a[3][2]	㉔ a[5][3]	㉝ a[8][0]
7 a[1][2]	16 a[3][3]	25 a[6][0]	34 a[8][1]
⑧ a[1][3]	⑰ a[4][0]	㉖ a[6][1]	㉟ a[8][2]
⑨ a[2][0]	⑱ a[4][1]	㉗ a[6][2]	㊱ a[8][3]

図 4-2-7 (2)　メモリーの図 (C 言語)

j →　2次元目

i ↓ 1次元目

~~a[0][0]~~	~~a[0][1]~~	~~a[0][2]~~	a[0][3]
~~a[1][0]~~	~~a[1][1]~~	~~a[1][2]~~	a[1][3]
~~a[2][0]~~	~~a[2][1]~~	~~a[2][2]~~ →	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
a[5][0]	a[5][1]	a[5][2]	a[5][3]
a[6][0]	a[6][1]	a[6][2]	a[6][3]
a[7][0]	a[7][1]	a[7][2]	a[7][3]
a[8][0]	a[8][1]	a[8][2]	a[8][3]

図 4-2-8　配列の図 (C 言語)

4-3　キャッシュミスに関する補足

キャッシュに関していくつか補足します。

- 実行したジョブでキャッシュミスが多発しているかどうかの情報は、一般には取得できませんが、取得できるマシン環境もあります。
- 図 4-3-1 (1) (前述の図 4-2-2 (1) と同じ) はストライド 4 がなのでキャッシュミスを起こしやすいですが、図 4-3-1 (2) (3) も同様にストライドが 4 なので、キャッシュミスを起こしやすくなります。

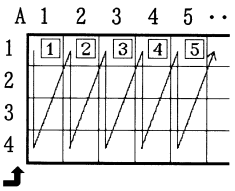
```
DIMENSION A(36)
DO I=1,36,4
  A(I) = A(I)+1.0
ENDDO
  :
```

図 4-3-1 (1)　×

```
DIMENSION A(36)
DO I=1,9
  A(I*4-3) = A(I*4-3)+1.0
ENDDO
  :
```

図 4-3-1 (2)　×

```
DIMENSION A(4,9)
DO J=1,9
  A(1,J) = A(1,J)+1.0
ENDDO          ↳イチ
  :
```



配列Aの図

図 4-3-1 (3)　×

図 4-3-2 (1) は配列 A の右側の添字がループ反復になっているので、一見キャッシュミスが発生するように見えますが、図 4-3-2 (2) と等価（内側のループを展開）なのでキャッシュミスはあまり発生しません。

図 4-3-3 (1) は配列 A の左側の添字がループ反復になっているので、一見キャッシュミスがあまり発生しないように見えますが、図 4-3-3 (2) と等価（内側のループを展開）なのでキャッシュミスが発生します。

```
DIMENSION A(4,9)
DO J=1,9
  A(1,J)=A(1,J)+1.0
    :      :
  A(4,J)=A(4,J)+1.0
ENDDO
  :
```

図 4-3-2 (1)　○

```
DIMENSION A(4,9)
DO J=1,9
  DO I=1,4
    A(I,J)=A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-2 (2)　○

```
DIMENSION A(4,9)
DO I=1,4
  A(I,1)=A(I,1)+1.0
    :      :
  A(I,9)=A(I,9)+1.0
ENDDO
  :
```

図 4-3-3 (1)　×

```
DIMENSION A(4,9)
DO I=1,4
  DO J=1,9
    A(I,J)=A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-3 (2)　×

- 本章で説明しているのはキャッシュを使用しているスカラー計算機（ワークステーションやパソコン）での考慮点です。スーパーコンピュータ（ベクトル計算機）では、キャッシュとは異なる機構を使用しているので、多重ループの順番が逆になっていてもあまり関係ありません（なお、キャッシュミスの考慮が必要になる機種もあるようなので、マニュアルなどを参照して下さい）。
  多重ループの場合、ベクトル計算機では内側のループがベクトル化されますが、ベクトル長（ループの反復回数）の長いループを内側のループにすると速度が速くなります。従って図 4-3-4 (1) よりも図 4-3-4 (2) の方が速度は速くなります。
  ただし、同じベクトル長の場合は、（マシンによりますが）図 4-3-4 (2) より、ループの順番が正しい図 4-3-4 (3) の方が速度が速くなるようです。また、（マシンによりますが）ベクトル化されないループではキャッシュが使用され、本章で述べた考慮が必要になる場合もあります。

```
DIMENSION A(10,100000)
DO J=1,100000
  DO I=1,10 ←ベクトル化
    A(I,J) = A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-4 (1)　×　ループの順番が正しいベクトル長が短い

```
DIMENSION A(10,100000)
DO I=1,10
  DO J=1,100000 ←ベクトル化
    A(I,J) = A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-4 (2)　○　ループの順番が逆ベクトル長が長い

```
DIMENSION AA(100000,10)
DO J=1,10
  DO I=1,100000 ←ベクトル化
    AA(I,J) = AA(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-4 (3)　○　ループの順番が正しいベクトル長が長い

- コンパイラによっては、最適化のコンパイルオプションを高くした場合、図 4-3-5 (1) のようにキャッシュミスを起こしやすい順番で書いた多重ループを、自動的に図 4-3-5 (2) のようにキャッシュミスを起こしにくい順番に入れ替えてくれる場合があります。しかし、以下のような理由から、コンパイラに任せるよりも、自分でホットスポットの DO ループを調べ、ループの順番が逆になっていたら手作業で入れ替えることをお勧めします。
  (1) DO ループ内の構造が複雑な場合は、コンパイラは入れ替えない可能性がある。
  (2) あるコンパイラで自動的に入れ替えても、将来他社のコンパイラに変更したとき自動的に入れ替えるとは限らない。
  (3) 最適化オプションのレベルを高くすると、コンパイラのバグ (2-2 節参照) など他の副作用を生じる可能性がある。

```
   :
DO I=1,N
  DO J=1,N
    A(I,J)=A(I,J)+1.0
  ENDDO
ENDDO
   :
```

図 4-3-5 (1)　×

→

```
   :
DO J=1,N
  DO I=1,N
    A(I,J)=A(I,J)+1.0
  ENDDO
ENDDO
   :
```

図 4-3-5 (2)　○

- 本章では、データキャッシュを図 4-3-6 (1) の簡易モデルで説明しています。実際のキャッシュはマシンによって構造や大きさが異なりますので、ここでは例を 1 つだけ紹介します。図 4-3-6 (2) のデータキャッシュは合計 64 K バイト、段数が 4 段、各段のキャッシュ線は 128 個、各キャッシュ線の大きさは 128 バイトです。1 つのキャッシュ線には、整数 (4 バイト) または単精度実数 (4 バイト) ならば 128 ÷ 4 = 32 個、倍精度実数 (8 バイト) ならば 128 ÷ 8 = 16 個の要素が入ります。従って、単精度のプログラムは倍精度のプログラムよりキャッシュが 2 倍あると考えることもできます（ただし、一般に科学技術計算では、計算精度の点から多くのプログラムが倍精度です)。
  また、最近のマシンでは、データキャッシュは 1 つでなく、図 4-3-6 (3) の斜線部のように複数備えているマシンもあります (図で例えれば L1 はレベル 1 の意)。
  ただし本書のレベルでは、キャッシュの物理的な大きさや構成を知る必要はなく、本章で述べた点に注意するだけでよいでしょう。

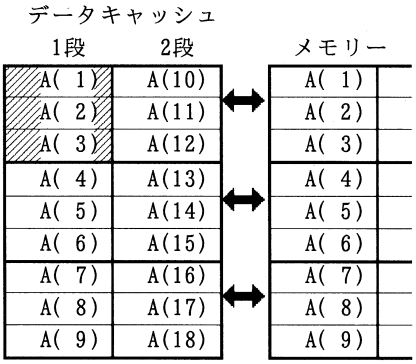


図 4-3-6 (1)　簡易モデル

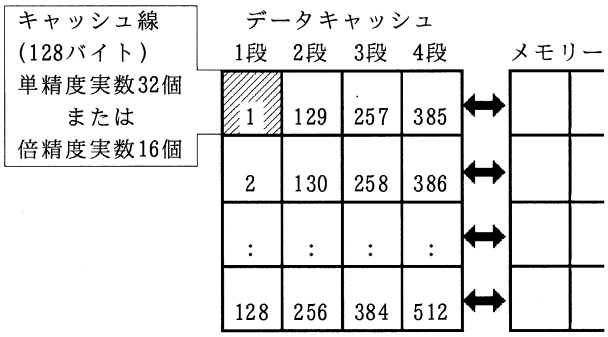


図 4-3-6 (2)　実際のキャッシュ例 (64 K バイト)

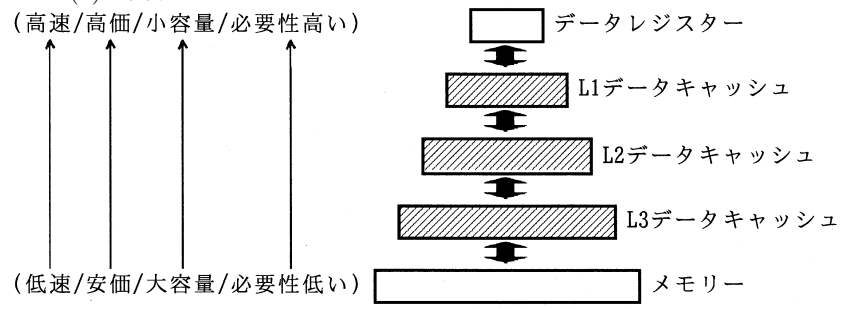


図 4-3-6 (3)

- 図 4-3-7 (1) (図 4-2-4 (1) と同一) では、二重ループの順番が反対になっているため、キャッシュミスが発生すると説明しました。これをもう少し詳しく検討してみます。外側のループが I=1 のとき、メモリー上の①, ②, ..., ⑨がアクセスされますが、ストライドが 4 でキャッシュ線より長いので、キャッシュミスが発生します。⑨のアクセスが終了した時点で、キャッシュにはメモリー上の着色した部分が入っています。次に外側のループが I=2 となり、⑩が必要になります。⑩は①がアクセスされたときに一度キャッシュに入りましたが、その後別の要素が入って追い出されたため、再度メモリーから転送する必要があり、再びキャッシュミスが発生します。以後の要素でも同様にキャッシュミスが発生します。
  図 4-3-7 (1) では内側のループの反復数は「9」でしたが、これを「4」に減らした図 4-3-8 (1) の動作を検討します。外側のループが I=1 のとき、メモリー上の①, ②, ③, ④がアクセスされ、キャッシュミスが発生します。次に外側のループが I=2 となり、⑤が必要になります。ところが図 4-3-7 (2) と違い、⑤はまだキャッシュに残っているためキャッシュミスは発生しません。以後の要素でも (⑦と⑯を除いて) 同様にキャッシュミスは発生しません。結局、図 4-3-8 (3) の□の部分のみでキャッシュミスが発生します。これを二重ループの順番が正しい図 4-3-9 (1) (2) と比較すると、キャッシュミスの頻度は同じになっています。
  以上をまとめると、多重ループの順番を逆にしても、内側のループの反復数が少なければ、キャッシュミスはあまり発生しません (実際には、ループの順番が正しい場合よりも若干遅くなることもあります)。また本例は簡易モデルなので、内側のループの「少ない反復数」を「4」としましたが、実際のマシンではもっと大きな値です (例えば「50」程度ですが、試行錯誤で決定して下さい)。4-5 節で説明するブロック化という技法では、この性質を利用してキャッシュミスの回数を減らします。しかし通常は余計なことを考えず、内側のループは配列の左側の添字 (Fortran の場合) で反復させて下さい。

```
DIMENSION A(4,9)
DO I=1,4
  DO J=1,9
    A(I,J) = A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-7 (1)　×二重ループの順番が反対内側のループの反復数が多い

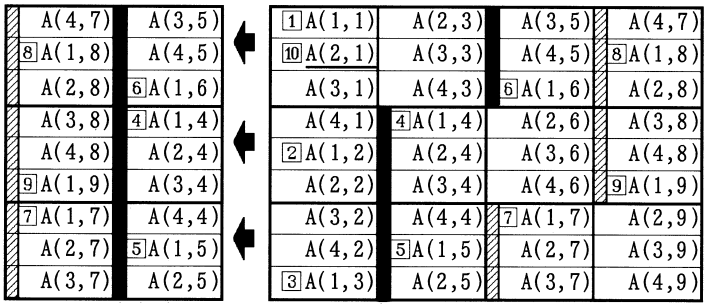


図 4-3-7 (2)　(I=2, J=1 の開始時点)

```
DIMENSION A(4,9)
DO I=1,4
  DO J=1,4
    A(I,J) = A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-8 (1)　○二重ループの順番が反対内側のループの反復数が少ない

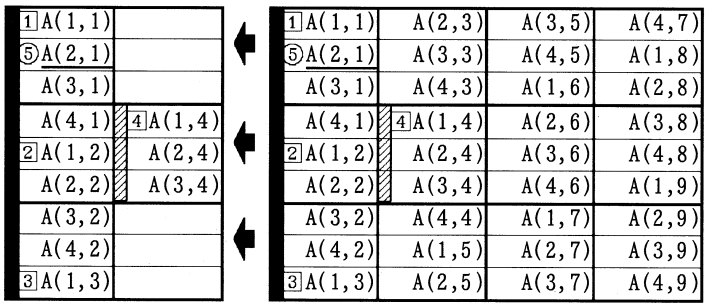


図 4-3-8 (2)　(I=2, J=1 の開始時点)

```
DIMENSION A(4,9)
DO J=1,4
  DO I=1,4
    A(I,J) = A(I,J)+1.0
  ENDDO
ENDDO
  :
```

図 4-3-9 (1)　○二重ループの順番が正しい内側のループの反復数が少ない

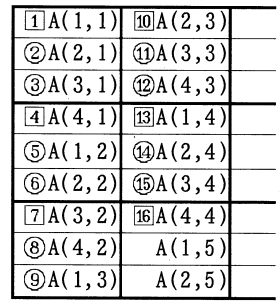


図 4-3-9 (2)

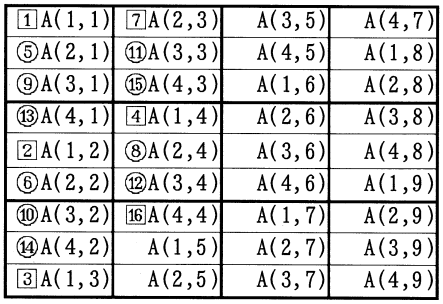


図 4-3-8 (3)

4-4　ストライドが 1 でキャッシュミスが発生する例

ストライドが 1 なのにキャッシュミスが発生する例を紹介します。

DO ループ内で使用している配列数が多い場合

再び簡易モデルで説明します。今までの例では、DO ループ内に含まれている配列は 1 つでしたが、通常は図 4-4-1 (1) のように複数の配列が含まれています。この例ではループ反復が I=1,4 と連続しているので、一見キャッシュミスが（最低限しか）発生しないように思われます。動作を調べると、図 4-4-1 (2) に示すように DO ループの反復 I=1 のとき①, ②, ..., ⑨がアクセスされ、この時点でキャッシュにはメモリー内の黒く着色した部分が入っています。次に反復 I=2 となり⑩が必要になりますが、⑩は一度キャッシュに入った後追い出されています。このため⑩を再度メモリーから転送する必要があり、キャッシュミスが発生します。

本例では DO ループの反復は連続しているので、同じ配列内の要素間のストライド (例えば A(1) と A(2) の距離) は確かに 1 ですが、使用する配列が複数あるため、実数のストライド (例えば①と②の距離) は 4 になっています。このような場合、DO ループに含まれる配列の数が多いと、上述のようにキャッシュミスが発生してしまいます。

ここで、図 4-4-2 (1) のように DO ループを適当に分割すると、1 つの DO ループ内の配列の数が少なくなります。すると図 4-4-2 (2) に示すように、1 つ目の DO ループの反復 I=1 のとき①, ②, ..., ⑤がアクセスされ、キャッシュにはメモリー内の着色した部分が入り、次に反復 I=2 となって⑥が必要になりますが、⑥はキャッシュに入っているためキャッシュミスは（最低限しか）発生しません。2 つ目の DO ループについても同様にキャッシュミスは（最低限しか）発生しません。

以上をまとめると、DO ループに含まれる配列数が多い場合、DO ループの反復が連続でもキャッシュミスが発生することがあります。このとき DO ループを（論理的に可能ならば）適当に分割して配列数を少なくすると、キャッシュミスを低減できる可能性があります。ただし DO ループを分割すると最適化が抑止され、却って遅くなることもあります。

前述のように、通常のマシン環境では、実行したジョブでキャッシュミスが多発しているかどうかを知る方法はありません。実用上は、例えば計算量がほぼ同じループがいくつか並んでいて、そのうち 1 つだけ妙に計算時間がかかっているような場合、そのループに配列がたくさん含まれているのであれば、DO ループを（可能であれば）適当に分割し、パフォーマンスが向上するかどうかを試行錯誤で試すしかありません。

```
    :
  DIMENSION A(4),B(4),C(4),D(4),E(4)
  DIMENSION W(4),X(4),Y(4),Z(4)
  DO I=1,4
    E(I) = A(I)+B(I)+C(I)+D(I)+E(I)
    Z(I) = W(I)+X(I)+Y(I)+Z(I)
  ENDDO
    :
```

図 4-4-1 (1)　×　ストライド 1、配列が多い

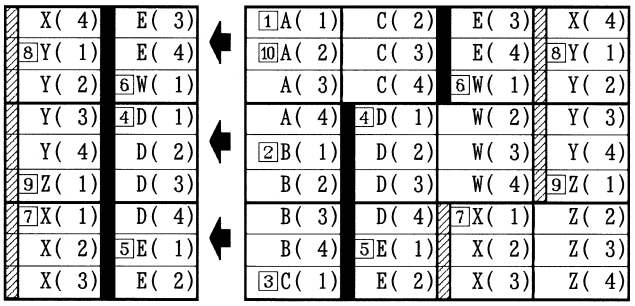


図 4-4-1 (2)

```
    :
  DIMENSION A(4),B(4),C(4),D(4),E(4)
  DIMENSION W(4),X(4),Y(4),Z(4)
  DO I=1,4
    E(I) = A(I)+B(I)+C(I)+D(I)+E(I)
  ENDDO
  DO I=1,4
    Z(I) = W(I)+X(I)+Y(I)+Z(I)
  ENDDO
    :
```

図 4-4-2 (1)　○　4-4-1 (1) の DO ループを分割

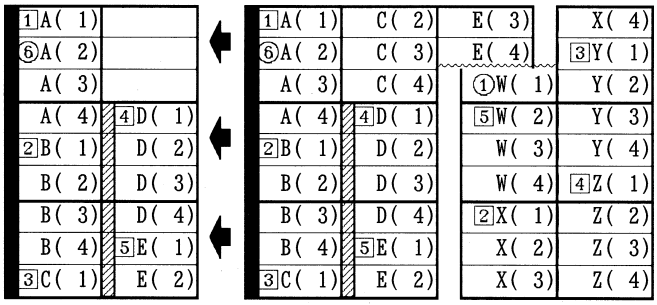


図 4-4-2 (2)

キャッシュ線の競合

日常生活では 4, 9, 13 などを不吉な数字とみなす場合がありますが、キャッシュミスの観点では 2 のベキ乗 (1024, 2048 など) が不吉な（パフォーマンスが低下する）値になります。この理由を以下で説明します。図 4-4-3 (2) に示すように、キャッシュの段数が 2 段、各段のキャッシュ線の数が 2、1 つのキャッシュ線に入る要素数が 4 の簡易モデルで説明します。

図 4-4-3 (1) のプログラムはストライドが 1 で配列数も少ないので、今までの説明ではキャッシュミスは発生しにくいはずですが、実際には全ての要素でキャッシュミスが発生します。以下に理由を説明します。

- I=1 のとき、A(1), B(1), C(1) が順に必要となります。まず A(1) と B(1) がキャッシュに転送されます。この時点での状態を図 4-4-3 (2) に示します。
- 次に C(1) が必要となります。ところが C(1)〜C(4) が入ることのできる「上段の」キャッシュ線は既に 2 段ともふさがっています（「下段の」キャッシュ線は空いていますが、入ることは出来ません）。このため①に示すように、C(1)〜C(4) はキャッシュ線内のデータのうち古い方の A(1)〜A(4) を上書きし、その結果、図 4-4-3 (3) のようになります。
- 次に I=2 となり、A(2)、B(2)、C(2) が順に必要となります。A(2) は図 4-4-3 (2) の時点ではキャッシュに入っていましたが、上記のように直前に上書きされてしまい、既にキャッシュには存在しません。そのため図 4-4-3 (3) の②に示すように、A(1)〜A(4) が再びメモリーから転送され、キャッシュ線内のデータのうち古い方の B(1)〜B(4) が上書きします。このとき発生するキャッシュミスは、A(2) がキャッシュに残っていた場合は発生しないので、余分なキャッシュミスということになります。
- 次に B(2) が必要となりますが、B(2) も上記のように直前に上書きされており、既にキャッシュには存在しないため、B(1)〜B(4) が再びメモリーから転送され、余分なキャッシュミスが発生します。以後も同様で、全ての要素がアクセスされるたびに必ず余分なキャッシュミスが発生してしまい、パフォーマンスは最悪となります。

この例では、配列 A、B、C の大きさ (8) が、キャッシュの 1 段の大きさとちょうど一致しています。このため、例えば DO ループが I=1 のときに使用される 3 つの要素 A(1)、B(1)、C(1) が、同じキャッシュ線の、全く同じ位置に入ります。ところがキャッシュの段数が 2 段なので、入ることができる上段のキャッシュ線は 2 つしかなく、また下段のキャッシュ線は空いていても入ることはできません。このため A(1)、B(1)、C(1) が 3 つ同時にキャッシュに入ることができず、1 つの要素（と近隣の要素）がキャッシュに入ると他がキャッシュから追い出されてしまうという悪循環が発生します。これを本書ではキャッシュ線の競合と呼びます。

```
DIMENSION A(8),B(8),C(8)
DO I=1,8
  C(I) = A(I) + B(I) + C(I)
ENDDO
  :
```

図 4-4-3 (1)　×

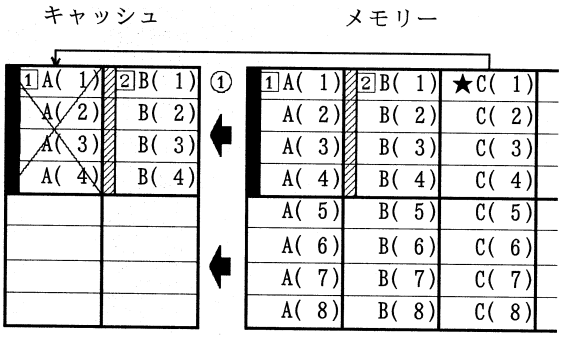


図 4-4-3 (2)

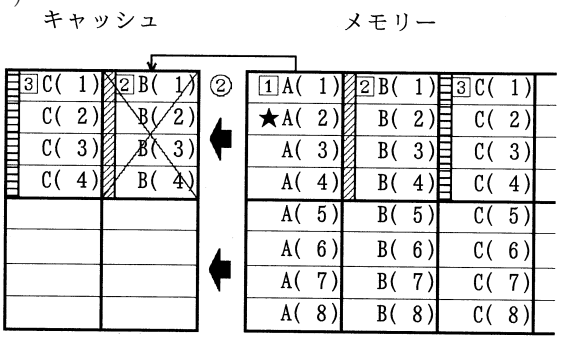


図 4-4-3 (3)

図 4-4-3 (1) の競合は、A(1)、B(1)、C(1) のうち、上段（下段も同様）のキャッシュ線に入る要素を、(キャッシュが 2 段だとすると) 2 個以下にすれば回避することができます。この回避方法を説明します。

図 4-4-4 (1) では各配列の大きさを 2 だけ大きくしており、メモリー上では図 4-4-4 (2) のように配置されます。図中の D は、A(1), B(1), C(1) が 3 つとも同じキャッシュ線に含まれないように位置をずらすためのダミー要素を示し、計算では使用しません。

A(1), B(1), C(1) がアクセスされた直後、キャッシュは図 4-4-4 (2) の状態になります。ダミー要素のおかげで、C(1)〜C(4) は「下段の」キャッシュ線に入ったことに注意して下さい。そして次に A(2) が必要になりますが、A(2) はキャッシュに残っているため、図 4-4-3 (2) (3) と違い余分なキャッシュミスは発生しません。

また図 4-4-4 (3) のように、配列 A, B, C の大きさは変えずに、間にダミー配列 D1, D2 を挿入しても、同じ効果があります。配列にダミー要素を付加したりダミー配列を挿入することを、パディングすると言います。

```
DIMENSION A(8+2),B(8+2),C(8+2)
DO I=1,8
  C(I) = A(I) + B(I) + C(I)
ENDDO
  :
```

図 4-4-4 (1)　○　ダミー要素を付加

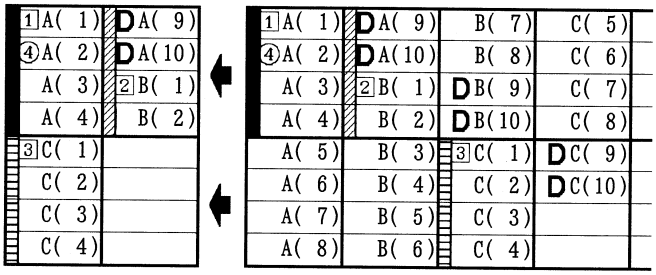


図 4-4-4 (2)

```
DIMENSION A(8),B(8),C(8)
DIMENSION D1(2),D2(2)
COMMON/COM/A,D1,B,D2,C
  :
```

図 4-4-4 (3)　○　ダミー配列を挿入

図 4-4-3 (1) のように各配列がキャッシュの 1 段とちょうど同じ大きさになることは現実にはほとんどありませんが、これ以外にもキャッシュ線の競合が発生する場合があります。図 4-4-5 (1) (2) では、各配列の大きさがキャッシュの 1 段の大きさのちょうど 2 倍になっているため、計算に同時に使用される、例えば A(1)、B(1)、C(1) が同じキャッシュ線の同じ位置■にそろってしまい、キャッシュ線の競合が発生します。

また図 4-4-6 (1) (2) では、配列の数は 1 つですが、計算に同時に使用される、例えば A(1,1)、A(1,2)、A(1,3) が同じキャッシュ線の同じ位置■にそろってしまい、キャッシュ線の競合が発生します。

このようにキャッシュ線の競合は、DO ループに含まれている配列の大きさが、キャッシュの 1 段の大きさ、およびその倍数（または約数）と同じ場合に発生する可能性があります。キャッシュの 1 段の大きさは 2 のベキ乗なので、言いかえると「キャッシュ線の競合は、配列の大きさが 2 のベキ乗の場合に発生する可能性がある」と言い換えることができます。

```
DIMENSION A(16),B(16),C(16)
DO I=1,16
  C(I) = A(I) + B(I) + C(I)
ENDDO
  :
```

図 4-4-5 (1)　×

■A(1)	A(9)	■B(1)	B(9)	■C(1)	C(9)
A(2)	A(10)	B(2)	B(10)	C(2)	C(10)
A(3)	A(11)	B(3)	B(11)	C(3)	C(11)
A(4)	A(12)	B(4)	B(12)	C(4)	C(12)
A(5)	A(13)	B(5)	B(13)	C(5)	C(13)
A(6)	A(14)	B(6)	B(14)	C(6)	C(14)
A(7)	A(15)	B(7)	B(15)	C(7)	C(15)
A(8)	A(16)	B(8)	B(16)	C(8)	C(16)

図 4-4-5 (2)

```
DIMENSION A(16,16)
DO J=2,15
  DO I=1,16
    A(I,J) = A(I,J-1) + A(I,J) + A(I,J+1)
  ENDDO
ENDDO
  :
```

図 4-4-6 (1)　×

■A(1,1)	A(9,1)	■A(1,2)	A(9,2)	■A(1,3)
A(2,1)	A(10,1)	A(2,2)	A(10,2)	A(2,3)
A(3,1)	A(11,1)	A(3,2)	A(11,2)	A(3,3)
A(4,1)	A(12,1)	A(4,2)	A(12,2)	A(4,3)
A(5,1)	A(13,1)	A(5,2)	A(13,2)	A(5,3)
A(6,1)	A(14,1)	A(6,2)	A(14,2)	A(6,3)
A(7,1)	A(15,1)	A(7,2)	A(15,2)	A(7,3)
A(8,1)	A(16,1)	A(8,2)	A(16,2)	A(8,3)

図 4-4-6 (2)

実際の大きさのキャッシュに対するプログラム修正例を示します。図 4-4-7 (2) の左図は、ある計算機の実際のキャッシュで、段数が 2 段、各段のキャッシュ線の数は 128、各キャッシュ線の大きさは 128 バイトです。以後倍精度実数 (8 バイト) を想定します。倍精度実数が、1 つのキャッシュ線に $128 \div 8 = 16$ 個入り、キャッシュの 1 つの段に $16 \times 128 = 2048$ 個入ります。

図 4-4-7 (1) では、2 次元配列 A の大きさ (2048×2048) が 2 のベキ乗です。1 次元目の大きさがキャッシュの 1 段の大きさと同じなので、メモリー内に、図 4-4-7 (2) の右図のように配置されます。従って、計算でほぼ同時に使用される、例えば A(1,1)，A(1,2)，A(1,3) の要素（図の〓）が同じキャッシュ線に入るため、キャッシュ線の競合が発生します。

図 4-4-8 (1) では、配列 A の 1 次元目に対し、1 つのキャッシュ線と同じ大きさ (16) のダミー要素（図の〓の部分）を付加したため、図 4-4-8 (2) に示すように、A(1,1)，A(1,2)，A(1,3) は異なるキャッシュ線に分散して入り、キャッシュ線の競合はなくなります。

なお、本例では、ダミー要素の大きさは 8 以上ならば、キャッシュ線の競合はなくなります。たとえば 8 の場合、図 4-4-8 (3) のようになり、A(1,1)，A(1,2) は同じキャッシュ線に入りますが、キャッシュの段数が 2 段なので、キャッシュ線の競合は発生しません。

```
REAL*8 A(2048,2048)
DO J=2,2047
  DO I=1,2048
    A(I,J) = A(I,J-1) + A(I,J) + A(I,J+1)
  ENDDO
ENDDO
  :
```

図 4-4-7 (1)

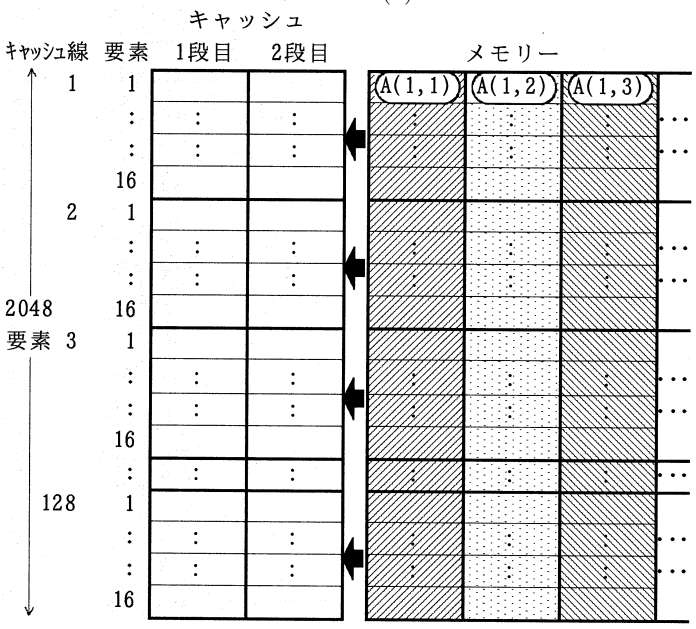


図 4-4-7 (2)

```
REAL*8 A(2048+16,2048)
DO J=2,2047
  DO I=1,2048
    A(I,J) = A(I,J-1) + A(I,J) + A(I,J+1)
  ENDDO
ENDDO
  :
```

図 4-4-8 (1)

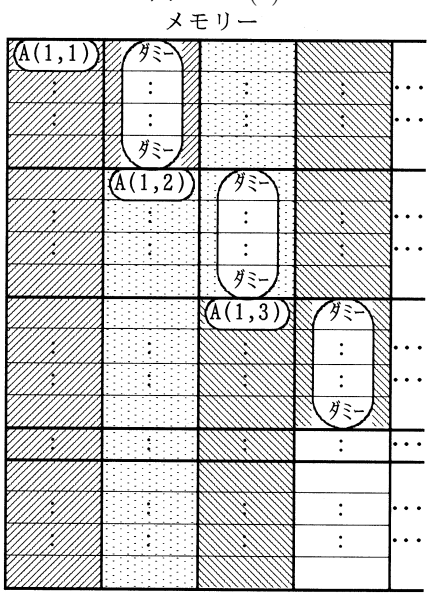


図 4-4-8 (2)

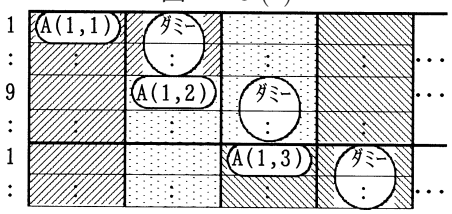


図 4-4-8 (3)

図 4-4-9 (1) では、2 次元配列 A，B，C の大きさ (4096 × 4096) が 2 のベキ乗です。4096 × 4096 はキャッシュの 1 段の大きさ (2048) で割りきれるので、メモリー内に、図 4-4-9 (2) の右図のように配置されます。従って、計算にほぼ同時に使用される、例えば A(1,1)，B(1,1)，C(1,1) の要素（図の〓）が同じキャッシュ線に入るため、キャッシュ線の競合が発生します。

この場合、配列 A，B，C の 1 次元目または 2 次元目のどちらか一方にダミー要素を n 個付加しても、全要素数 $(4096 + n) \times 4096$ 個がキャッシュの 1 段全体の大きさ (2048) で割りきれるので、同じ問題が発生します。

1，2 次元目の両方にダミー要素を付加する方法もありますが、ここではダミー配列を使用してパディングする方法を説明します。図 4-4-10 (1) では、配列 A，B，C の間に、1 つのキャッシュ線と同じ大きさ (16) のダミー配列 D1 と D2（図の〓の部分）を挿入したため、図 4-4-10 (2) に示すように、A(1,1)，B(1,1)，C(1,1) は異なるキャッシュ線に分散して入り、キャッシュ線の競合はなくなります。本例の場合、前の例と同様に、ダミー配列の大きさは 8 以上でキャッシュ線の競合はなくなります。

なお、図 4-4-10 (1) の①で COMMON 文を使用したのは、メモリー上で配列 A，D1，B，D2，C が確実にこの順に配置されるようにするためです。

```
REAL*8 A(4096,4096),B(4096,4096),C(4096,4096)
DO J=1,4096
  DO I=1,4096
    C(I,J) = A(I,J) + B(I,J) + C(I,J)
  ENDDO
ENDDO
  :
```

図 4-4-9 (1)

```
REAL*8 A(4096,4096),B(4096,4096),C(4096,4096)
REAL*8 D1(16),D2(16)
COMMON/COM/A,D1,B,D2,C                    ①
DO J=1,4096
  DO I=1,4096
    C(I,J) = A(I,J) + B(I,J) + C(I,J)
  ENDDO
ENDDO
  :
```

図 4-4-10 (1)

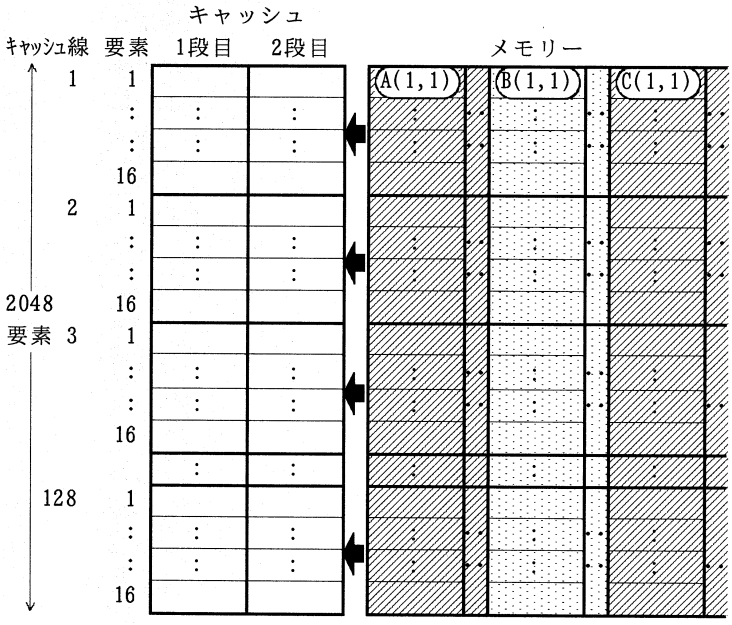


図 4-4-9 (2)

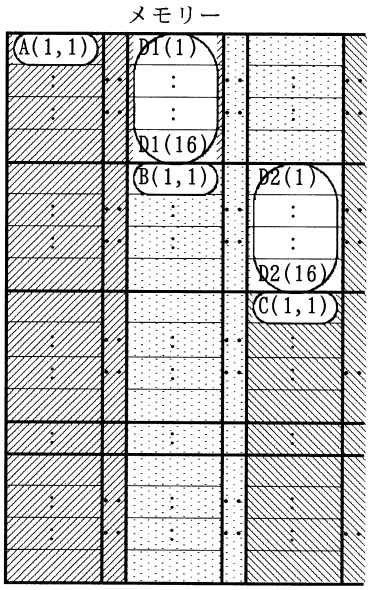


図 4-4-10 (2)

前述のように、通常のマシン環境では、実行したジョブでキャッシュミスが多発しているかどうかを知る方法はありません。従って実用上は、ホットスポットの DO ループ内の主要な配列（例えば流体計算なら流速や圧力）の大きさが 2 のベキ乗になっている場合、キャッシュ線の競合によって速度が低下している可能性があるため、上記の方法でパディングを試みるのがよいでしょう。

4-5　キャッシュチューニング

行と列の入れ替え

図 4-5-1 (1) で、配列 B は 2 重ループの反復が反対なのでキャッシュミスが発生します。この場合、配列 B の次元を逆にすればキャッシュミスを低減できますが、何らかの理由で逆にできないとします。

この場合、図 4-5-1 (2) のように、②で配列 B の値を転置して一時配列 BB に入れ、①では一時配列 BB を使用するようにすれば、①ではストライドが 1 になりキャッシュミスを低減させることができます。②の処理が新規に追加され、しかもキャッシュミスが発生しますが、①が 10000 回実行されており、配列 B のキャッシュミス低減による速度向上の方が大きいため、一般に図 4-5-1 (2) の方が良いパフォーマンスが得られます。

本例のように、一見無駄なことを行い、結果的にパフォーマンスを向上させる、いわば「肉を切らせて骨を断つ」というやり方はチューニングの 1 つのテクニックです。

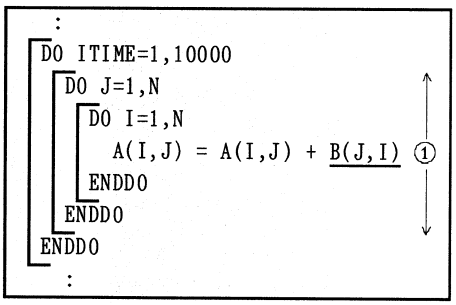


図 4-5-1 (1)　×

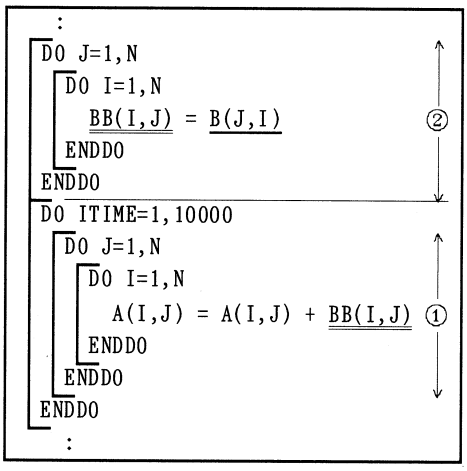


図 4-5-1 (2)　○

収集／拡散

【例 1】図 4-5-2 (1) (2) では、サブルーチン SUB が 500 回コールされ、DO ループのストライドが 100 なのでキャッシュミスが発生します。そこで図 4-5-2 (3) (4) のように、配列 A のデータのうち計算で使用するデータのみを一時配列 AA にあらかじめ 1 回だけ収集し、サブルーチン SUB での計算は一時配列 AA を用いてストライド 1 で行い、最後に一時配列 AA のデータを配列 A に 1 回だけ拡散します。

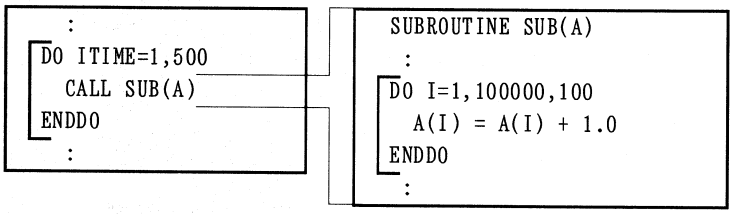


図 4-5-2 (1)　×

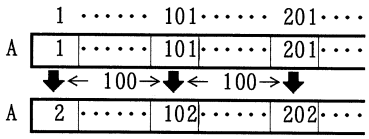


図 4-5-2 (2)

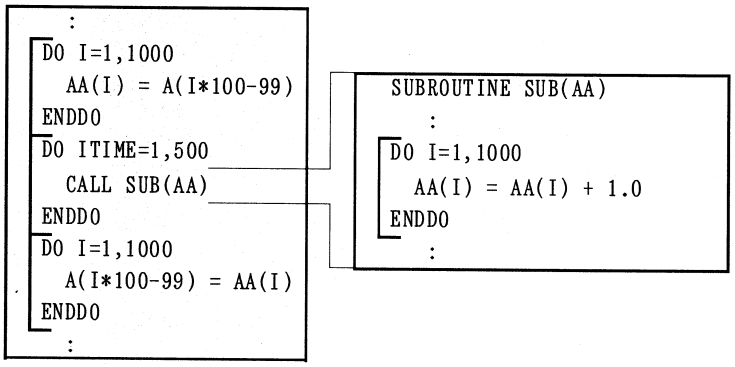


図 4-5-2 (3)　○

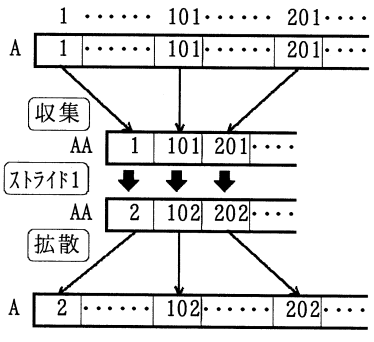


図 4-5-2 (4)

【例 2】図 4-5-3 (1) (2) では配列 IND を使用して間接アドレス指定を行っています。ここで配列 IND 内の値は全て異なるとします。要素をメモリー上を行ったり来たりしてアクセスするため、ストライドが大きくなりキャッシュミスが発生します。そこで図 4-5-3 (3) (4) のように収集と拡散を行ってストライドを 1 にします。

```
    :
  DO ITIME=1,500
    CALL SUB(A,IND)
  ENDDO
    :
```

```
    SUBROUTINE SUB(A,IND)
      :
    DO I=1,1000
      A(IND(I)) = A(IND(I)) + 1.0
    ENDDO
      :
```

図 4-5-3 (1)　×

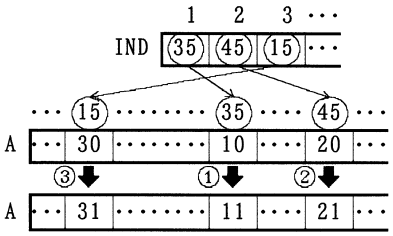


図 4-5-3 (2)

↓

```
    :
  DO I=1,1000
    AA(I) = A(IND(I))
  ENDDO
  DO ITIME=1,500
    CALL SUB(AA)
  ENDDO
  DO I=1,1000
    A(IND(I)) = AA(I)
  ENDDO
    :
```

```
    SUBROUTINE SUB(AA)
      :
    DO I=1,1000
      AA(I) = AA(I) + 1.0
    ENDDO
      :
```

図 4-5-3 (3)　○

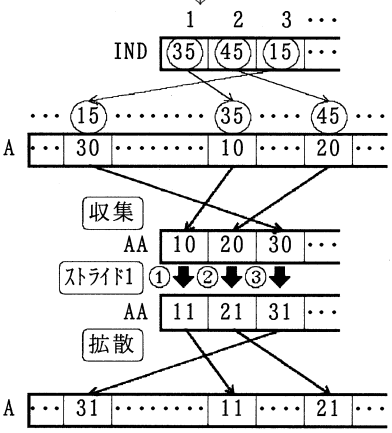


図 4-5-3 (4)

【例 3】図 4-5-4 (1) (2) も【例 2】と同様ケースですが、配列 IND 内に同一の値が複数存在する点が異なります (本例では「35」)。この場合、図 4-5-4 (3) (4) のように拡散を行ってストライドを 1 にします。図 4-5-4 (3) の下線部が図 4-5-3 (3) と異なります。

```
    :
  DO ITIME=1,500
    CALL SUB(A,IND)
  ENDDO
    :
```

```
    SUBROUTINE SUB(A,IND)
      :
    DO I=1,1000
      A(IND(I)) = A(IND(I)) + 1.0
    ENDDO
      :
```

図 4-5-4 (1)　×

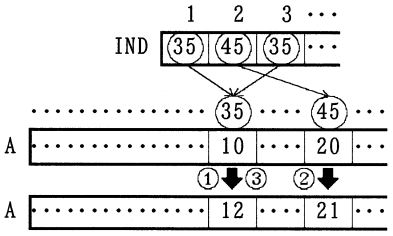


図 4-5-4 (2)

↓

```
    :
  DO I=1,1000
    AA(I) = 0.0
  ENDDO
  DO ITIME=1,500
    CALL SUB(AA)
  ENDDO
  DO I=1,1000
    A(IND(I)) = A(IND(I))+AA(I)
  ENDDO
    :
```

```
    SUBROUTINE SUB(AA)
      :
    DO I=1,1000
      AA(I) = AA(I) + 1.0
    ENDDO
      :
```

図 4-5-4 (3)　○

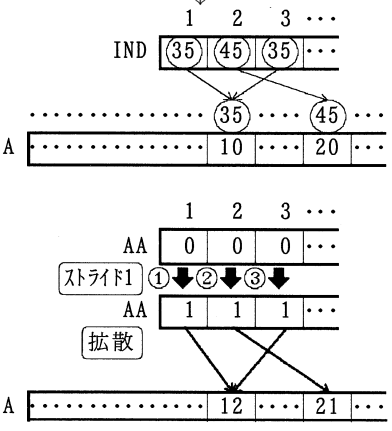


図 4-5-4 (4)

ブロック化

図 4-5-5 (1) で、配列 A はキャッシュミスがあまり発生しませんが、配列 B は 2 重ループの反復が反対なのでキャッシュミスが発生します。B の次元を逆にすればキャッシュミスを減らすことができますが、何らかの理由で逆にすることができない場合、ブロック化（ストリップ・マイニング、またはタイニングとも言います）という方法で、配列の次元を逆にせずにキャッシュミスを減らすことができます。なお、この方法は元のプログラムと計算順序が変わるので、順序が変わってもロジックが変わらない場合にのみ適用可能です。

前述の簡易モデルを想定します。配列 A と B の「配列の図」を図 4-5-5 (2) に示します。例えば a12 は配列の A(1,2) を表し、矢印は要素が処理される順序を示します。本書では配列の 1 次元目を縦方向、2 次元目を横方向で表すので、図 4-5-5 (2) の各要素は、メモリー上では図の縦方向 (Fortran の場合) に並んでいます。

ブロック化したプログラムを図 4-5-6 (1) に、各要素の処理順序を図 4-5-6 (2) に示します。配列 A と B を 3 × 3 の小さいブロックに分割し、対応するブロック (例えば A11 と B11、A21 と B12) 内の要素を連続に計算します。

配列 A は、図 4-5-5 (2) ではストライドが 1 です。図 4-5-6 (2) では、例えば a11〜a31 はストライドが 1 ですが、a31 と a12 の間はストライドが離れているため、図 4-5-5 (2) より若干キャッシュミスが増えると思われます。

配列 B は、図 4-5-5 (2) ではキャッシュミスが発生しました。ところで 4-3 節の最後の項目で、「多重ループの順番を逆にしても、内側のループの反復数が少なければ、キャッシュミスはあまり発生しない」と説明しました。図 4-5-6 (1) では、最も内側のループの反復数は少ない (簡易モデルなので「3」) ので、配列 B に対して多重ループの順番が逆になっていても、キャッシュミスはあまり発生しません。

これについてもう少し詳しく説明します。配列 B の「配列の図」を図 4-5-7 (1) に、「メモリーの図」（ただし簡易モデル）を図 4-5-7 (2) に示します。図 4-5-6 (1) を実行すると、図 4-5-7 (1) の左上の 9 個の要素が番号順に処理されます。このとき図 4-5-7 (2) に示すように、①〜③はキャッシュミスが発生しますが、④〜⑨は①〜③と共にキャッシュに入るため、以後キャッシュミスは発生しません。

```
    :
DO J=1,6
  DO I=1,6
    A(I,J) = B(J,I)
  ENDDO
ENDDO
    :
```

図 4-5-5 (1)

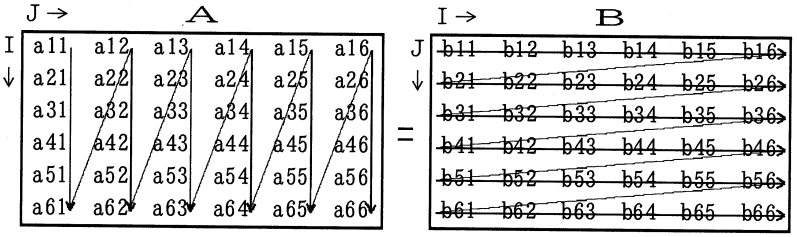


図 4-5-5 (2)　配列の図（ブロック化前）

```
    :
DO JJ=1,6,3
  DO II=1,6,3
    DO J=JJ,JJ+2
      DO I=II,II+2
        A(I,J) = B(J,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
    :
```

図 4-5-6 (1)

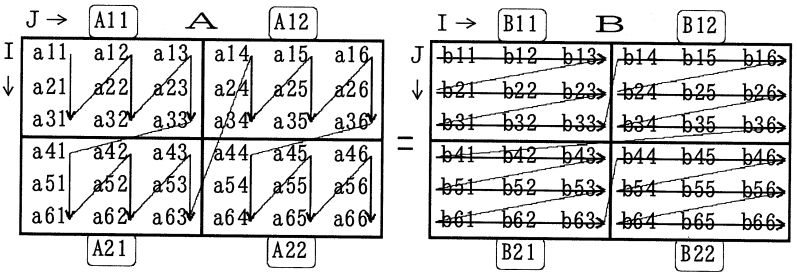


図 4-5-6 (2)　配列の図（ブロック化後）

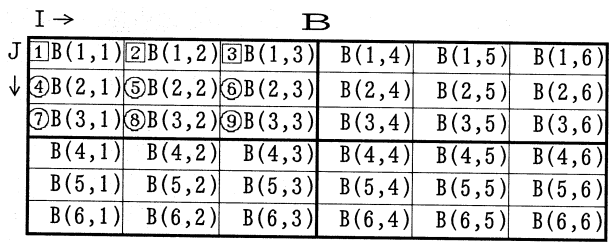


図 4-5-7 (1)　配列 B の配列の図

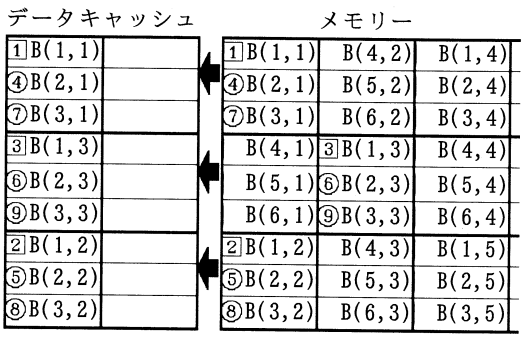


図 4-5-7 (2)　配列 B のメモリーの図

図 4-5-6 (1) (2) では、最も内側のループの「少ない反復数」を「3」としましたが、実際のマシン環境ではもっと大きな値です (例えば「50」程度ですが、試行錯誤で決定して下さい)。図 4-5-8 のプログラムを、「少ない反復数」を「50」としてブロック化したプログラムと、配列 A, B 内の要素の処理順序を、下記の【例 1】に示します。配列の各次元の要素数 (IMAX と JMAX) が「50」で割りきれない場合に、ループ反復の I と J が配列の上限を越えるのを防ぐため、組込関数 MIN を使用して下線部の処理を行っています。

```
    :
  DO J=1,JMAX
    DO I=1,IMAX
      A(I,J) = B(J,I)
    ENDDO
  ENDDO
    :
```

図 4-5-8

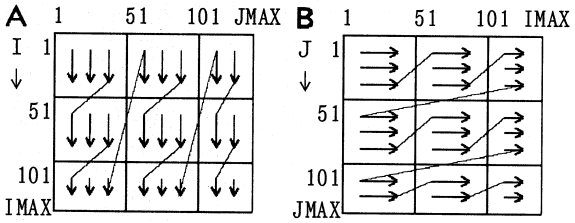
本例は【例 2】～【例 4】のようにブロック化することもできます。どれが一番速いかは、DO ループ内の計算内容とマシン環境に依存します。

ブロック化は、行列乗算などで有効です。数値計算ライブラリー (7 章参照) で提供されている行列乗算は、一般に専門家がブロック化などのチューニングを行っています。それらのライブラリーを使用する場合は、ユーザーが自分でブロック化する必要はありません。

ブロック化は、非常に大規模な連立一次方程式を解くため行列がメモリーに入りきらず、作業用にディスクを使用しなければならないような場合に、ディスクの I/O を少なくする目的でも用いられます (参考文献 [9] 参照)。

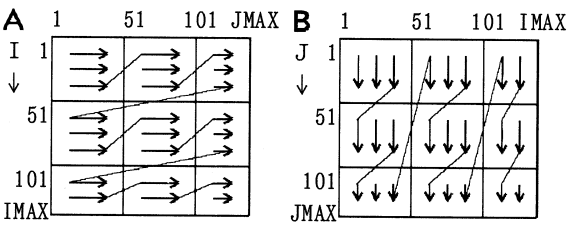
【例1】

```
    :
  DO JJ=1,JMAX,50
    DO II=1,IMAX,50
      DO J=JJ,MIN(JJ+49,JMAX)
        DO I=II,MIN(II+49,IMAX)
          A(I,J) = B(J,I)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
    :
```



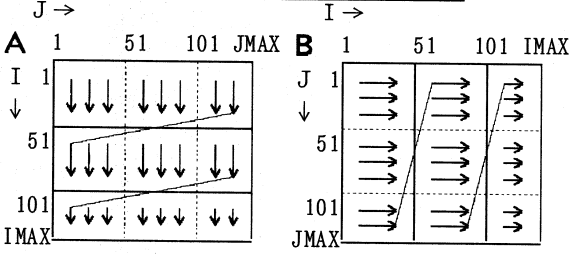
【例2】

```
    :
  DO II=1,IMAX,50
    DO JJ=1,JMAX,50
      DO I=II,MIN(II+49,IMAX)
        DO J=JJ,MIN(JJ+49,JMAX)
          A(I,J) = B(J,I)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
    :
```



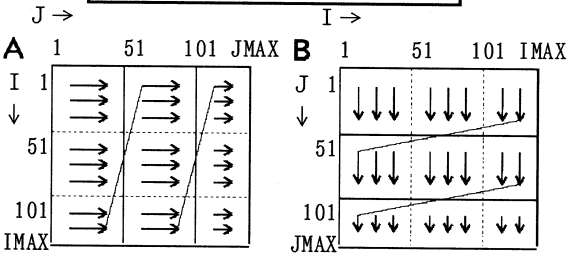
【例3】

```
    :
  DO II=1,IMAX,50
    DO J=1,JMAX
      DO I=II,MIN(II+49,IMAX)
        A(I,J) = B(J,I)
      ENDDO
    ENDDO
  ENDDO
    :
```



【例4】

```
    :
  DO JJ=1,JMAX,50
    DO I=1,IMAX
      DO J=JJ,MIN(JJ+49,JMAX)
        A(I,J) = B(J,I)
      ENDDO
    ENDDO
  ENDDO
    :
```



第5章　その他のチューニング

本章では、キャッシュチューニング以外の各種のチューニング技法について説明します。本書の中では4章と並んで重要な章です。

5-1　除算と組込関数の削減

計算時間のかかる演算の代表が除算と組込関数です。これらの演算はパフォーマンスにとっては天敵なので、ホットスポットで使用している場合、以下のような方法で削減できないか検討してみて下さい。

除算の乗算化

四則演算のうち、除算は加算、減算、乗算よりも CPU 時間がかかります。なお、Fortran の組込関数 MOD（剰余の計算）も、内部的に除算を使用するので、同様に CPU 時間がかかります。

除算は乗算に置き換えるのが 1 つのチューニング方法です。以下に例を示します。ただし、コンパイル時に最適化オプションが指定されている場合、これらのチューニングをコンパイラが自動的に行う場合があります。従って、チューニング後にパフォーマンスを測定し、速度が速くなった場合のみ、そのチューニングを採用して下さい（速度が速くならない場合は、コンパイラが自動的に行ったと考えられます）。

また、除算を乗算に置き換えると、桁落ちや丸め誤差の影響で計算結果が若干変わることがありますので、誤差が計算結果に影響しない程度なのかどうかも確認して下さい。なお、倍精度で計算すれば誤差は少なくなります。

```
   :
DO I=1,N
  X(I) = X(I)/2.5D0
ENDDO
   :
```

➡

```
   :
DO I=1,N
  X(I) = X(I)*0.4D0
ENDDO
   :
```

図 5-1-1

```
   :
DO I=1,N
  X(I) = X(I)/Y
ENDDO
   :
```

➡

```
   :
 YY = 1.0D0/Y
DO I=1,N
  X(I) = X(I)*YY
ENDDO
   :
```

図 5-1-2

```
   :
DO I=1,N
  B(I) = A(I)/B(I)/C(I)
ENDDO
   :
```

➡

```
   :
DO I=1,N
  B(I) = A(I)/( B(I)*C(I) )
ENDDO
   :
```

図 5-1-3

```
   :
DO I=1,N
  A(I) = B(I)/C(I)
  X(I) = Y(I)/Z(I)
ENDDO
   :
```

➡

```
   :
DO I=1,N
  TEMP = 1.0D0/(C(I)*Z(I))
  A(I) = B(I)*Z(I)*TEMP
  X(I) = Y(I)*C(I)*TEMP
ENDDO
   :
```

図 5-1-4　通分

```
  :
IF (A(I)/B(I) > 2.0D0) THEN
  :   (ただしB(I)＞0の場合)
```

➡

```
  :
IF (A(I) > 2.0D0*B(I)) THEN
  :
```

図 5-1-5

組込関数の削除

SIN, EXP などの初等組込関数は、ステートメントとしては 1 行でも、内部的にテーラー展開され、多くの演算から構成されているので、当然ながら計算時間がかかります。従ってホットスポットでは無造作に組込関数を使用せず、なるべく組込関数の数を減らすようにして下さい。

計算式の段階で式を整理したり、以下のように数学公式を用いて、組込関数の数を減らせる場合があります。なお、以下の (1)〜(3) は計算精度が変わるので注意して下さい。

マシン環境によっては、組込関数の高速版が提供されている場合もあります。これは純正の組込関数より速度を速くする代わりに精度を若干犠牲にしていますが、精度に敏感なプログラムでなければ問題にならないことが多いので、組込関数がホットスポットになっているプログラムでは使用を検討してみて下さい。

(1) $e^x \cdot e^y \rightarrow e^{x+y}$

(2) $\log_a m + \log_a n \rightarrow \log_a m \cdot n$

(3) $\sin\theta\cos\theta \rightarrow 0.5 \times \sin 2\theta$

(4) (R > 0 の場合) `IF (SQRT(X*X+Y*Y) < R)〜` → `IF (X*X+Y*Y < R*R)〜`

累乗の計算

以下の例も、計算精度が変わるので注意して下さい。

- 累乗の計算 B = A^3 の速度を図 5-1-6 の各ケースで比較すると、①は「真面目に」テーラー展開で計算するので本質的に計算時間がかかりますが、④は単なる乗算を 2 回行うだけなので①よりはるかに高速です。②と③は数学的には④と同じですが、内部的に①と④のいずれの方法（またはその他の方法）で計算するかはマシン環境に依存します。また①と④のどちらになるかが、変数 A が実数の場合と複素数の場合で変わったり、累乗の乗算 (A^n の n) の値によって変わったりするマシン環境もあります。
- 平方根 B = A$^{1/2}$ を計算する場合、図 5-1-7 (1) の⑤と⑥は数学的には同じですが、⑥の組込関数 SQRT は（特別なハードウェア機構を用いるなどで）高速に計算するマシン環境もあります。一方⑤は①と同じ方法で計算する場合は遅いですが、(⑥が速い場合) コンパイラが内部的に⑥に変換していれば速くなります。同様に、⑦を⑧にすると速くなる場合もあります。
- 図 5-1-7 (2) の⑨と⑩も数学的には同じですが、⑤より⑥の方が速い場合には差が出ます。⑨は①と同様に本質的に遅いですが、⑩は単なる乗算と、(⑥が速い場合) 高速な⑥を使用するので⑨よりも速くなります。⑨は⑪や⑫のように書くこともでき、⑨〜⑫のどれが一番速いかはマシン環境に依存します。この方法が適用できるのは B = A$^{n.5}$ や B = A$^{n/2}$ (n は整数) のような場合です。
- 立方根 B = A$^{1/3}$ を計算する場合、図 5-1-8 (1) の⑬と⑭は数学的には同じですが、⑤、⑥の場合と同様に、マシン環境によっては⑭の方が速くなります。⑭の方が速い場合、図 5-1-8 (2) の⑮のような B = A$^{n/3}$ (n は整数) の計算は、⑯〜⑲のように高速化することができます。

```
✖ B = A**3.1 ①
？ B = A**3.0 ②
？ B = A**3   ③
○ B = A*A*A  ④
```

図 5-1-6

```
×？ B = A**0.5          ⑤
○？ B = SQRT(A)         ⑥
×？ B = A**0.25         ⑦
○？ B = SQRT(SQRT(A))   ⑧
```

図 5-1-7 (1)

→

```
×？ B = A**2.5              ⑨
○？ B = A*A*SQRT(A)         ⑩
○？ B = SQRT(A**5)          ⑪
○？ B = SQRT(A*A*A*A*A)     ⑫
```

図 5-1-7 (2)

```
×？ B = A**(1.0/3.0)   ⑬
○？ B = CBRT(A)        ⑭
```

図 5-1-8 (1)

→

```
×？ B = A**(5.0/3.0)       ⑮
○？ B = A*CBRT(A**2)       ⑯
○？ B = A*CBRT(A*A)        ⑰
○？ B = CBRT(A**5)         ⑱
○？ B = CBRT(A*A*A*A*A)    ⑲
```

図 5-1-8 (2)

多項式の計算

以下の (1) のような多項式の計算は、(2) にすると計算量が減ります。ただし、計算精度が変わるので注意して下さい。

(1) $Y = 2X^4 + 3X^3 + 4X^2 + 5X + 6$

(2) $Y = (((2X + 3)X + 4)X + 5)X + 6$

組込関数と等価な Fortran コード

例えば図 5-1-9 (1)〜(4) の下線部のような組込関数は、図のように同じ機能を Fortran で簡単に作成することができます。どちらが速いかはマシン環境によって異なりますので、ホットスポットで下記のような演算を多用する場合は、試しに両方測定して比較すると良いでしょう。また組込関数を Fortran に置き換えた場合、図 5-1-10 に示すように周辺の計算が簡単化されて無駄な計算がなくなり、高速になる場合もあります。

なお、図 5-1-9 (4) に示す最大値、あるいは最小値の計算は、7 章で紹介する数値計算ライブラリーで高速なルーチンが提供されている場合があります。

```
B = ANINT(A)
  :
```

⇕

```
  :
IF (A > 0.0) THEN
  ITEMP = A + 0.5
ELSE
  ITEMP = A - 0.5
ENDIF
B = ITEMP
  :
```

図 5-1-9 (1)

```
K = NINT(A)
  :
```

⇕

```
  :
IF (A > 0.0) THEN
  K = A + 0.5
ELSE
  K = A - 0.5
ENDIF
  :
```

図 5-1-9 (2)

```
C = SIGN(A,B)
  :
```

⇕

```
  :
IF (B >= 0.0) THEN
  C =  ABS(A)
ELSE
  C = -ABS(A)
ENDIF
  :
```

図 5-1-9 (3)

```
  :
XMAX = 0.0
DO I = 1, N
  XMAX = MAX(XMAX,X(I))
ENDDO
  :
```

⇔

```
  :
XMAX = 0.0
DO I = 1, N
  IF (X(I)>XMAX) XMAX = X(I)
ENDDO
  :
```

図 5-1-9 (4)

```
C = SIGN(1.0,B)
A = (C+1.0)*D
  :
```

⇓ 図5-1-9(3)参照

```
  :
IF (B >= 0.0) THEN
  C =  ABS(1.0)
ELSE
  C = -ABS(1.0)
ENDIF
A = (C+1.0)*D
  :
```

⇓ 整理すると

```
  :
IF (B >= 0.0) THEN
  A = 2.0*D
ELSE
  A = 0.0
ENDIF
  :
```

図 5-1-10

5-2　無駄な計算の削減

5-2-1　削減による高速化の実例

無駄な計算を行っている部分を削減することによって、何十倍も高速になるプログラムが（それほど多くはないですが）あります。その実例を 2 つ紹介します。

プログラム例 1

ある電磁場解析のプログラムは、元々計算時間が約 20 分かかっていました。gprof (3-4-3 節参照) を使用して分析したところ、図 5-2-1 (1) に示すように、タイムステップ・ループ内にある①と②の部分がホットスポットになっていました。この部分では、密行列の連立一次方程式 $Ax = b$ を、以下のように LU 分解で解きます。

- ①では、係数行列 A を L（上三角行列）と U（下三角行列）に分解します。
- ②では、①で分解した LU と右辺のベクトル b を使用して解ベクトル x を求めます。

この部分を、図 5-2-1 (2) に示すように数値計算ライブラリー (7 章参照) の LU 分解ルーチンに置き換えたところ、計算時間は 1 分に短縮されました。通常、数値計算ライブラリーに置き換えても、速度の向上はせいぜい数倍程度です。ところが①, ②では、LU 分解をクラウト法 (5-5-2 節参照) という、特にキャッシュミスが多発する方法で行っていました。これを数値計算ライブラリーに置き換え、その結果キャッシュミスが低減したため、約 20 倍も速くなりました。

さらに高速にするため、他のチューニングを検討しました。③と④のうち、計算時間の大半を③が占めています。ところがプログラムを検討したところ、タイムステップ・ループが反復しても、係数行列 A の値は変わらないことが分かりました。つまり③はタイムステップごとに毎回計算する必要はなく、最初に一度だけ行えばよい訳です。そこで図 5-2-1 (3) の⑤に示すように、タイムステップ・ループに入る前に一度だけ計算するように修正したところ、計算時間は最終的に 3 秒まで短縮しました。

なお、さらに検討したところ、$Ax = b$ の A は実際には密行列ではなく帯行列であることがわかりました。従って⑤, ⑥を帯行列用の数値計算ライブラリーに変更すれば、さらに高速になると思われます。

```
DO ITIME = 1, 100
    :
  LU＝A（ｸﾗｳﾄ法） ①
  x＝(LU)⁻¹b    ②
    :
ENDDO
```

図 5-2-1 (1)　20 分

→

```
DO ITIME = 1, 100
    :
  LU＝A （数値計算ﾗｲﾌﾞﾗﾘｰ） ③
  x＝(LU)⁻¹b                ④
    :       （数値計算ﾗｲﾌﾞﾗﾘｰ）
ENDDO
```

図 5-2-1 (2)　1 分

→

```
  LU＝A （数値計算ﾗｲﾌﾞﾗﾘｰ） ⑤
DO ITIME = 1, 100
    :
  x＝(LU)⁻¹b                ⑥
    :       （数値計算ﾗｲﾌﾞﾗﾘｰ）
ENDDO
```

図 5-2-1 (3)　3 秒

プログラム例 2

もう 1 つ実際のプログラム例を紹介します。ある原子核のプログラムで、計算したデータからヒストグラムを作成する部分が全体の 90% 以上を占めていました。その部分では図 5-2-2 (1) に示すように、配列 X に入っている、0〜99.9 までの値を持つ 1000 個のデータを分類し、0.0〜10.0、10.0〜20.0, ..., 90.0〜100.0 の各レンジに入る個数を求め、配列 IHIST（ヒストグラム）を作成します。

まず図 5-2-2 (2) に示すオリジナルプログラムの動作を以下に説明します。

- 例えば 1 つ目の要素 X(1) の値が 11.1 であるとします。図 5-2-2 (1) の「値のレンジ」を上から順に調べ、そのレンジに 11.1 が入っているかどうかをチェックします。これを①と②で行います。
- 11.1 は 10.0〜20.0 のレンジに入っており、そのときの IX の値は 2 なので、③でカウンター IHIST(2) の値を 1 だけアップします。
- ④で内側の DO ループから抜け、次の要素 X(2) の処理に移ります。

チューニング後の図 5-2-2 (3) では、1 つ目の要素 X(1)=11.1 に対する IHIST 内の位置 IX が 2 であることを、⑤の計算で一発で求めます (11.1*0.1+1=2.11、これを整数化して 2)。このチューニングによって、平均して 5 回反復していた①のループが不要となったため、プログラム全体の 90% を占めていたこの部分の計算量が 1/5 程度に減少しました。

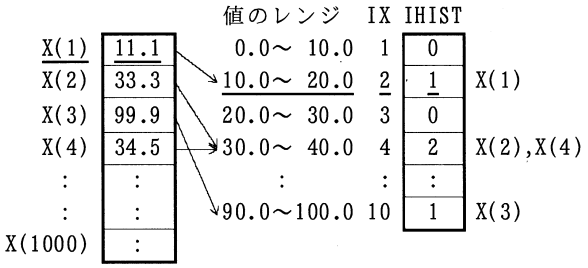


図 5-2-2 (1)

```
      REAL*8 X(1000)
      INTEGER IHIST(10)
        :
      DO I = 1, 1000 (1000個の各要素ごとに)
        DO IX = 1, 10 (10個のレンジを調べる) ①
          IF (X(I) < DFLOAT(IX)*10.0D0) THEN ②
            IHIST(IX) = IHIST(IX) + 1        ③
            GOTO 999                         ④
          ENDIF
        ENDDO
999     CONTINUE
      ENDDO
        :
```

図 5-2-2 (2)　オリジナルプログラム

→

```
      REAL*8 X(1000)
      INTEGER IHIST(10)
        :
      DO I = 1, 1000
        IX = X(I)*0.1D0 + 1          ⑤
        IHIST(IX) = IHIST(IX) + 1
      ENDDO
        :
```

図 5-2-2 (3)　チューニング後のプログラム

上記は原子核のプログラムですが、ホットスポットの図 5-2-2 (2) は原子核の計算とは関係がなく、単にヒストグラムを作成しているだけです。

このように、プログラムの機能として重要な部分とホットスポットは必ずしも一致しません。従って、機能として重要な部分が多分ホットスポットであろうと決めつけずに、必ずホットスポットを特定するツール (3-4 節参照) を使用して、どこが実際のホットスポットなのかを調べて下さい。

5-2-2　削減方法の基本

ホットスポット内の各計算部分を、「本当にその計算は必要なのか？」という観点で検討すると、意外と無駄な計算を行っている部分を発見できることがあります。本節では、無駄な計算を削減するための、一般的な方法を説明します。

図 5-2-3 (1) で、タイムステップ・ループ内から呼ばれるサブルーチン SUB1，SUB2 内の 3 つのループ（着色した部分）がホットスポットになっているとします。これらのループには、計算時間のかかる演算の代表である、除算と組込関数が含まれています。

前提として、①のスカラー変数 X，Y と、②の配列 Z の値は、タイムステップ・ループが反復しても不変だとします。

チューニングしたプログラムを図 5-2-3 (2) に示します。

- ①はタイム・ステップループが反復しても不変なので、1のようにタイムステップ・ループに入る前に一度だけ計算して結果をスカラー変数 TEMP1 に保管し、タイムステップ・ループ内では参照のみ行います。
  同様に②も2のように一度だけ計算しますが、このとき「2.0D0*」の部分も一緒に計算します。2では、結果を保管するために一時配列 TEMP2 が新規に必要になります。
  このように、同じ計算を何度も行っている部分を一度だけ計算して結果を保管し、後は参照のみ行なうようにすると、当然ながら、無駄な計算を削減することができます。
- ホットスポットのループはＩとＪの 2 重ループなので、ループ内の命令の各項は通常Ｉ と Ｊの 2 つの添字を持ちます。ところが③は内側のループ反復のＩしか添字を持たないので、外側の Ｊのループ反復で毎回計算する必要はありません。そこで3のように 2 重ループの外に出して 1 重ループで計算し、結果を一時配列 TEMP3 に保管します。③の計算回数は N × N でしたが、3は N 回で済むので計算回数は激減します (例えば N=100 なら 1/100 に減少します)。
  本例ではさらに、③と全く同じ計算を、2 つ目のループの (3) とサブルーチン SUB2 の [3] でも行っているので、一時配列 TEMP3 の計算結果をそのまま使用します。このように、計算時間のかかる全く同じ計算を、プログラム内の複数の箇所で行っているプログラムをときどき見かけます。
- ④は外側のループ反復のＪしか添字を持たないので、内側のＩのループ反復で毎回計算する必要はありません。そこで4のように内側のループの外に出します。
- ⑤，⑥の項は、Ｉ と Ｊの 2 つの添字を持ちますが、Ｉ と Ｊの部分を 2 つに分離できるので、③，④の場合と同様に、5，6のように修正します。
- ⑦は同一ループ内で同じ計算を 2 回行っています。そこで7のように一度だけ行うようにします。

なお、1，4，6，7のように一時配列が不要なチューニングは、ロジックが簡単であればコンパイラが自動的に行います。試しに手作業でチューニングしてみて、（計算結果が合っていて、かつ）効果があれば採用し、効果がなければコンパイラが自動的にチューニングしたと考えて下さい。

以上のチューニングにより、ホットスポットの 2 つのループの内側に含まれていた除算と組込関数は、7の組込関数 EXP のみとなりました。実際にこのようなチューニングでパフォーマンスが数倍向上するプログラムもあります。

```
PROGRAM MAIN
  :
DO ITIME=1,100 (タイムステップ・ループ)
    :
  CALL SUB1(～)
  CALL SUB2(～)
    :
ENDDO
  :
```

```
SUBROUTINE SUB1(～)
  :
DO J=1,N
  DO I=1,N     ①                ②
    P(I,J) = X/Y        + 2.0D0*SIN(Z(I,J))
&          + A(I)/B(I) + 3.0D0*COS(C(J)) ④
&              ③       + SQRT(D(I))/E(J)
  ENDDO                          ⑤      ⑥
ENDDO
DO J=1,N
  DO I=1,N       (3)              ⑦
    Q(I,J) = A(I)/B(I) + F(I,J)*EXP(G(I,J))
    R(I,J) =       4.0D0*F(I,J)*EXP(G(I,J))
  ENDDO                           ⑦
ENDDO
  :
```

```
SUBROUTINE SUB2(～)
  :
DO J=1,N
  DO I=1,N
    S(I,J) = A(I)/B(I)
  ENDDO         [3]
ENDDO
 :
```

図 5-2-3 (1)　チューニング前

```
PROGRAM MAIN
  :
TEMP1 = X/Y [1]
DO J=1,N
  DO I=1,N
    TEMP2(I,J) = 2.0D0*SIN(Z(I,J))
  ENDDO                  [2]
ENDDO
DO ITIME=1,100 (タイムステップ・ループ)
    :
  CALL SUB1(～)
  CALL SUB2(～)
    :
ENDDO
  :
```

```
SUBROUTINE SUB1(～)
  :
DO I=1,N
  TEMP3(I) = A(I)/B(I)  [3]
  TEMP5(I) = SQRT(D(I)) [5]
ENDDO
DO J=1,N
  TEMP4 = 3.0D0*COS(C(J)) [4]
  TEMP6 = 1.0D0/E(J)      [6]
  DO I=1,N
    P(I,J) = TEMP1    + TEMP2(I,J)
&          + TEMP3(I) + TEMP4
&                     + TEMP5(I)*TEMP6
  ENDDO
ENDDO
DO J=1,N
  DO I=1,N
    TEMP7  = F(I,J)*EXP(G(I,J)) [7]
    Q(I,J) = TEMP3(I) + TEMP7
    R(I,J) =      4.0D0*TEMP7
  ENDDO
ENDDO
  :
```

```
SUBROUTINE SUB2(～)
  :
DO J=1,N
  DO I=1,N
    S(I,J) = TEMP3(I)
  ENDDO
ENDDO
 :
```

図 5-2-3 (2)　チューニング後

5-2-3　削減例

- 以下の例では、一時変数を TEMP を消去することにより、計算が簡単になります。

```
    :
DO I=1,N
  IF (IFLAG(I)==1) THEN
    TEMP = 1.0
  ELSE
    TEMP = 0.0
  ENDIF
  A(I) = TEMP*B(I)*C(I)*D(I)
ENDDO
    :
```

図 5-2-4 (1)

→

```
    :
DO I=1,N
  IF (IFLAG(I)==1) THEN
    A(I) = B(I)*C(I)*D(I)
  ELSE
    A(I) = 0.0
  ENDIF
ENDDO
    :
```

図 5-2-4 (2)

- 図 5-2-5 (1) の指数関数 $e^{X(I)+Y(J)}$ の計算は、二重ループ内にあるため 100^2 (= 10000) 回行われます。これを $e^{X(I)+Y(J)} = e^{X(I)} \cdot e^{Y(J)}$ のように 2 つの関数に分解し、$e^{X(I)}$ を I のみのループ、$e^{Y(J)}$ を J のみのループで計算するようにすれば、図 5-2-5 (2) のように $e^{X(I)}$ と $e^{Y(J)}$ の計算回数はそれぞれ 100 回となり、計算回数は大幅に減少します。ただし計算の精度が変わります。

```
    :
DO J = 1, 100
  DO I = 1, 100
    A(I,J) = e^(X(I)+Y(J))
  ENDDO
ENDDO
    :
```

図 5-2-5 (1)

→

```
    :
DO I = 1, 100
  EI(I) = e^X(I)
ENDDO
DO J = 1, 100
  EJ = e^Y(J)
  DO I = 1, 100
    A(I,J) = EI(I)*EJ
  ENDDO
ENDDO
    :
```

図 5-2-5 (2)

- 図 5-2-6 (1) で、A の部分に含まれる命令数は少なく、B の部分に含まれる命令数は多いとします。②の IF 文が真になる確率が非常に高い場合、②より後の部分はほとんど実行されないため、ホットスポットは A のみとなります。このように、同一ループでも、各部のホットスポットの分布が偏っている場合があります。本例では、A で実行しても B で実行してもよい命令は、当然ながら B で実行する必要があります。①の計算結果が③のみで用いられるとすると、図 5-2-6 (2) のように①を②の後に移動すれば、①の実行される回数が減少します。このような DO ループは分子動力学法や個別要素法で現れます (5-5-6 節参照)。

```
      :
DO 100 I = 1, N
  WORK = B(I)/C(I)                ①
A (命令数が少ない)
  IF (A(I) < 0.0) GOTO 100        ②
    :
B (命令数が多い)
    :
  D(I) = WORK + 1.0               ③
100 CONTINUE
    :
```

図 5-2-6 (1)

→

```
      :
DO 100 I = 1, N
A (命令数が少ない)
  IF (A(I) < 0.0) GOTO 100        ②
  WORK = B(I)/C(I)                ①
    :
B (命令数が多い)
    :
  D(I) = WORK + 1.0               ③
100 CONTINUE 1
    :
```

図 5-2-6 (2)

- 有限要素法などの非構造格子を扱うプログラムで、図 5-2-8 (1) のように、各格子の面積を表す配列 AREA (体積の場合だと例えば配列 VOL) による除算を、タイムステップごとに毎回行うパターンが現れることがあります。タイムステップが反復しても格子の面積が変化しないのであれば、図 5-2-8 (2) のようにタイムステップに入る前に一度だけ除算を行なって結果を一次配列 XAREA に保存しておき、実際の計算では除算の代わりに XAREA を使用して乗算を行なうようにします。ただし除算を乗算に置き換えたことにより、計算結果が若干変わる可能性があります。

```
      :
    DO ITIME = 1, 100
         :
      DO J = 1, N
        DO I = 1, N
          A(I,J) = A(I,J)/AREA(I,J)
        ENDDO
      ENDDO
         :
    ENDDO
      :
```

図 5-2-8 (1)

→

```
      :
    DO J = 1, N
      DO I = 1, N
        XAREA(I,J) = 1.0D0/AREA(I,J)
      ENDDO
    ENDDO
    DO ITIME = 1, 100
         :
      DO J = 1, N
        DO I = 1, N
          A(I,J) = A(I,J)*XAREA(I,J)
        ENDDO
      ENDDO
         :
    ENDDO
      :
```

図 5-2-8 (2)

- 図 5-2-9 (1) の右辺にある 5 つの乗算では、配列 A と B の添字が全て同一です。そこで図 5-2-9 (2) のように、配列 A と B の乗算を一度だけ行なって一時配列 TEMP に保管し、実際の計算は TEMP を使用するようにすれば、乗算を削減することができます。

```
      :
    DO J=1,N
      DO I=1,M
        C(I,J) = A(I  ,J  )*B(I  ,J  )
     &         + A(I-1,J  )*B(I-1,J  )
     &         + A(I+1,J  )*B(I+1,J  )
     &         + A(I  ,J-1)*B(I  ,J-1)
     &         + A(I  ,J+1)*B(I  ,J+1)
      ENDDO
    ENDDO
      :
```

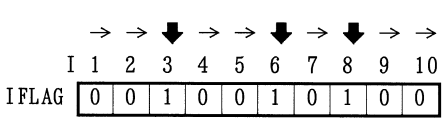
図 5-2-9 (1)

→

```
      :
    DO J=0,N+1
      DO I=0,M+1
        TEMP(I,J) = A(I,J)*B(I,J)
      ENDDO
    ENDDO
    DO J=1,N
      DO I=1,M
        C(I,J) = TEMP(I  ,J  )
     &         + TEMP(I-1,J  )
     &         + TEMP(I+1,J  )
     &         + TEMP(I  ,J-1)
     &         + TEMP(I  ,J+1)
      ENDDO
    ENDDO
      :
```

図 5-2-9 (2)

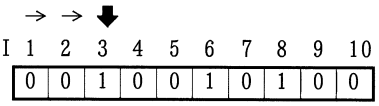
● 図 5-2-10 (1) に示すように配列 IFLAG に 0 または 1 が入っているとします。このプログラムでは、配列 IFLAG に 1 になっている要素があれば、スカラー変数 ITEMP に「999」をセットします。配列 IFLAG の全ての要素を調べていますが、1 になっている要素が 1 つでも見つかれば、それ以降の要素は調べる必要がありません。そこで図 5-2-10 (2) のように EXIT 文 (2-6 節参照) を追加すれば、当然ながら計算量は減少します。




```
ITEMP = 0
DO I=1,10
  IF (IFLAG(I)==1) THEN
    ITEMP = 999
  ENDIF
ENDDO
PRINT *,ITEMP
```

図 5-2-10 (1)

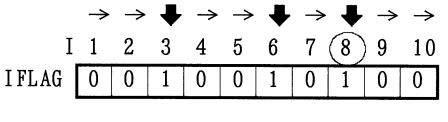
→




```
ITEMP = 0
DO I=1,10
  IF (IFLAG(I)==1) THEN
    ITEMP = 999
    EXIT
  ENDIF
ENDDO
PRINT *,ITEMP
```

図 5-2-10 (2)

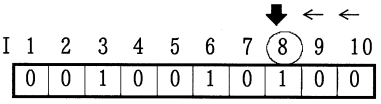
一方図 5-2-10 (3) のプログラムでは、配列 IFLAG の要素を調べ、1 になっている最後の要素の要素番号をスカラー変数 ITEMP にセットします。しかし番号の小さい要素から順に調べているため、全ての要素を調べる必要があります。そこで図 5-2-10 (4) のようにループを 10 から逆に反復させ、配列 IFLAG が 1 になっている最初の要素が見つかったら DO ループから EXIT するようにすれば、計算量は減少します。




```
ITEMP = 0
DO I=1,10
  IF (IFLAG(I)==1) THEN
    ITEMP = I
  ENDIF
ENDDO
PRINT *,ITEMP
```

図 5-2-10 (3)

→




```
ITEMP = 0
DO I=10,1,-1
  IF (IFLAG(I)==1) THEN
    ITEMP = I
    EXIT
  ENDIF
ENDDO
PRINT *,ITEMP
```

図 5-2-10 (4)

● 画像などのデータの平滑化を行うプログラムのチューニング例を説明します。この方法は計算領域の次元数が多いほど有効ですが、以下では簡単のため2次元で説明します。
オリジナルプログラムを図5-2-11 (1) に示します。図5-2-11 (2) に示すように、配列Bの各値（例えば★）は、配列Aの同じ位置とその周囲の計9点の値（▲, ▼, ■）を平均して求めます。ここで例えば着色した部分では同じ■の加算を図5-2-11 (2) (3) (4) の計3回行っており、これを1回に削減することができます。
チューニング後のプログラムと動作を図5-2-12 (1) (2) に示します。①では〓に示すように縦方向の3つの要素の組を加算し、2次元配列WORKに保管しておきます。そして②では〓に示すように横方向の3つの要素の組を加算します。これによって着色した部分の加算を1回にすることができました。
上記では配列WORKが2次元になりますが、これを（メモリーを節約して）1次元にする方法を図5-2-12 (3) (4) に示します。③では〓に示すように、横方向の3つの要素の組を加算し、1次元配列WORKに保管しておきます。そして④では〓に示すように縦方向の3つの要素を加算します。

```
    :
  DO J=2,4
    DO I=2,3
      TEMP = A(I-1,J-1)+A(I,J-1)+A(I+1,J-1)
     &     + A(I-1,J  )+A(I,J  )+A(I+1,J  )
     &     + A(I-1,J+1)+A(I,J+1)+A(I+1,J+1)
      B(I,J) = TEMP/9.0
    ENDDO
  ENDDO
    :
```

図 5-2-11 (1)

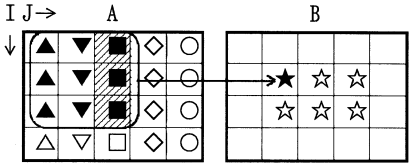


図 5-2-11 (2)

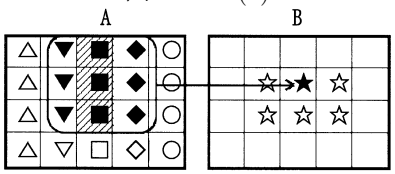


図 5-2-11 (3)

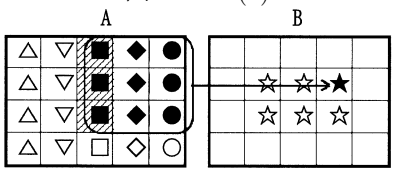


図 5-2-11 (4)

```
    :
  DO J=1,5
    DO I=2,3
      WORK(I,J) = A(I-1,J)+A(I,J)+A(I+1,J)     ①
    ENDDO
  ENDDO
  DO J=2,4
    DO I=2,3
      TEMP = WORK(I,J-1)+WORK(I,J)+WORK(I,J+1)②
      B(I,J) = TEMP/9.0
    ENDDO
  ENDDO
    :
```

図 5-2-12 (1)

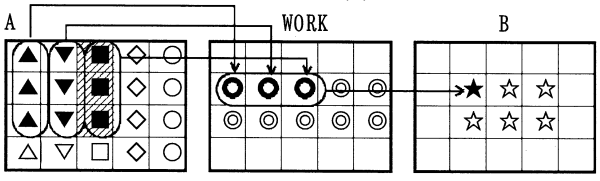


図 5-2-12 (2)

```
    :
  DO J=2,4
    DO I=1,4
      WORK(I) = A(I,J-1)+A(I,J)+A(I,J+1)③
    ENDDO
    DO I=2,3
      TEMP = WORK(I-1)+WORK(I)+WORK(I+1)④
      B(I,J) = TEMP/9.0
    ENDDO
  ENDDO
    :
```

図 5-2-12 (3)

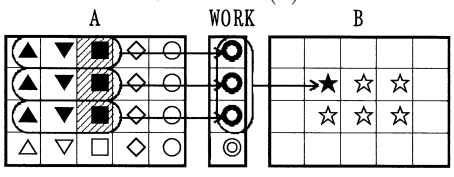


図 5-2-12 (4)

5-2-4　IF 文に関連するチューニング

5-2-4-1　IF 文の除去

IF 文は、IF 文自体の処理時間の他に、IF 文の前後で、コンパイラが行う最適化が抑止される可能性があるため、ホットスポットではなるべく（無駄な）IF 文を使用しないで下さい。

- 図 5-2-13 (1) の IF 文はループの反復とは無関係なので、図 5-2-13 (2) のように DO ループを IF 文の THEN 側と ELSE 側の 2 つに分割することによって、IF 文を DO ループの外へ出すことができます。

```
   :
DO I=1,N
  IF (X > 0.0) THEN
    A(I) = 1.0
  ELSE
    A(I) = 2.0
  ENDIF
ENDDO
   :
```

図 5-2-13 (1)

→

```
   :
IF (X > 0.0) THEN
  DO I=1,N
    A(I) = 1.0
  ENDDO
ELSE
  DO I=1,N
    A(I) = 2.0
  ENDDO
ENDIF
   :
```

図 5-2-13 (2)

- 図 5-2-14 (1) で、配列 IFLAG の各要素には「0」と「1」のいずれかが設定されているとします。このとき DO ループ内の IF 文は、図 5-2-14 (2) のようにすれば除去することができます。

	1	2	3	4	5	6	･･･	N
IFLAG	1	0	0	1	0	1	･･･	0

```
   :
DO I=1,N
  IF (IFLAG(I)==1) A(I) = A(I) + B(I)
ENDDO
   :
```

図 5-2-14 (1)

```
A(I) = A(I) + 0*B(I)  【IFLAG(I)=0のとき】
A(I) = A(I) + 1*B(I)  【IFLAG(I)=1のとき】
```

```
   :
DO I=1,N
  A(I) = A(I) + IFLAG(I)*B(I)
ENDDO
   :
```

図 5-2-14 (2)

- 図 5-2-15 (1) で、配列 IFLAG の各要素には「1」,「2」,「3」のいずれかが設定されているとします。このとき DO ループ内の IF 文は、図 5-2-15 (2) のようにすれば除去することができます。

```
   ;
ICOUNT1 = 0
ICOUNT2 = 0
DO I=1,N
  IF     (IFLAG(I)==1) THEN
    ICOUNT1 = ICOUNT1 + 1
  ELSEIF (IFLAG(I)==2) THEN
    ICOUNT2 = ICOUNT2 + 1
  ENDIF
ENDDO
   :
```

図 5-2-15 (1)

	1	2	3	4	5	6	7	8	9	10(=N)
IFLAG	2	1	3	2	1	2	2	3	1	3

	1	2	3
ICOUNT	3	4	3

→

```
   :
ICOUNT(1) = 0
ICOUNT(2) = 0
DO I=1,N
  ICOUNT(IFLAG(I)) = ICOUNT(IFLAG(I)) + 1
ENDDO
ICOUNT1 = ICOUNT(1)
ICOUNT2 = ICOUNT(2)
   :
```

図 5-2-15 (2)

● 図 5-2-16 (1) の IF 文は、図 5-2-16 (2) (3) のように計算部分を 3 つに分割すれば、除去することができます。同じ処理内容の DO ループを 2 つにしたくない場合は、図 5-2-16 (4) のように DO ループを 1 つにします (計算ロジックによっては、1 つにできない場合もあります)。

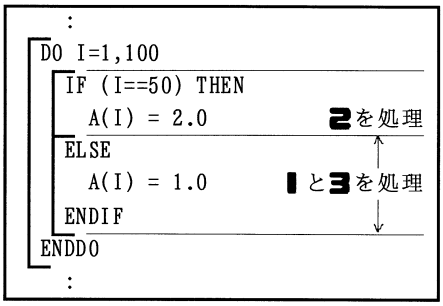


図 5-2-16 (1)

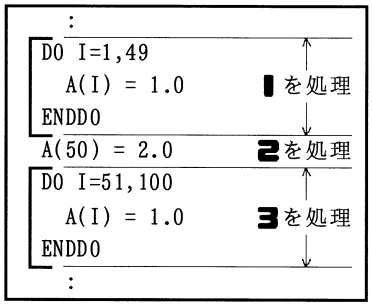


図 5-2-16 (2)

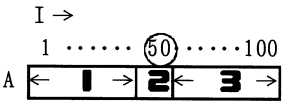


図 5-2-16 (3)

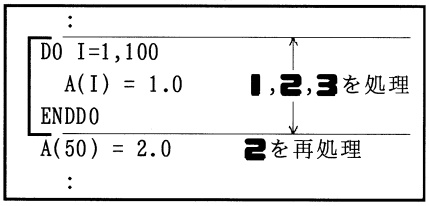


図 5-2-16 (4)

● 上記と同様の例として、図 5-2-17 (1) の IF 文 (その要素が行列の対角要素かどうかを判別する IF 文) は、図 5-2-17 (2) (3) のように内側のループを 3 つに分割すれば、除去することができます。同じ処理内容の DO ループを 2 つにしたくない場合は、図 5-2-17 (4) のように DO ループを 1 つにします (計算ロジックによっては、1 つにできない場合もあります)。図 5-2-17 (5) の IF 文も、同じ方法で除去することができます。

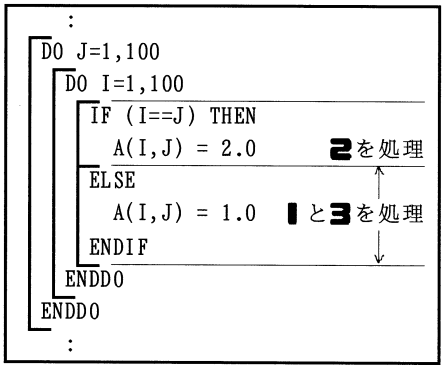


図 5-2-17 (1)

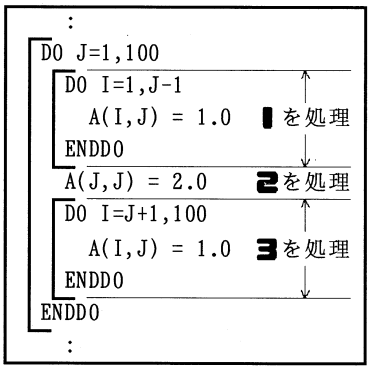


図 5-2-17 (2)

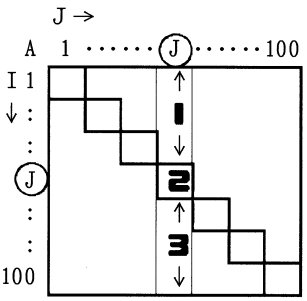


図 5-2-17 (3)

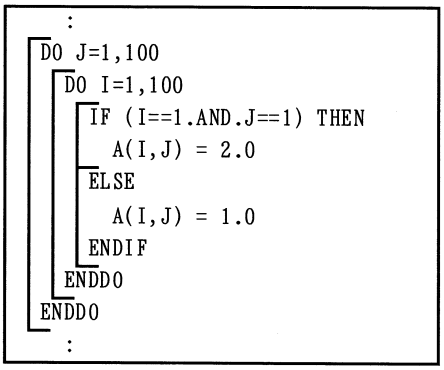


図 5-2-17 (5)

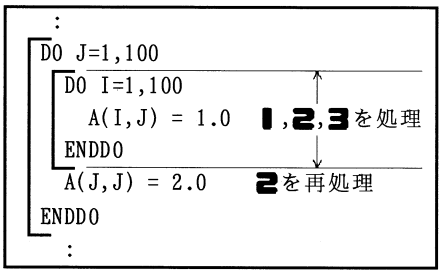


図 5-2-17 (4)

● 図 5-2-18 (1) では②の DO ループが 10000 回反復しますが、このうち③の IF 文の条件式 IFLAG が真になるのは I=3 と I=5 の 2 回のみであるとします (図 5-2-18 (3) 参照)。また IFLAG の値は①のタイムステップ・ループが反復しても不変であるとします。
これをチューニングした図 5-2-18 (2) では、タイムステップ・ループに入る前に④で一度だけ、③が真になるループ反復 (本例では I=3 と I=5) を調べ、これを示すようにテーブル ITABLE に保管しておきます。④が終了した後、カウンター ICOUNT には②が真になる要素数 (本例では 2) が入っています。本番の⑤では、③が真になる要素のみをテーブル ITABLE から抜き出して計算します。このチューニングによって、②で 10000 回反復していたループが⑤では 2 回に減少します。

```
   :
DO ITIME=1,10                    ①
   :
  DO I=1,10000                   ②
    IF (IFLAG(I)==1) THEN        ③
      A(I) = A(I) + 1.0
    ENDIF
  ENDDO
   :
ENDDO
   :
```

図 5-2-18 (1)

→

```
   :
ICOUNT = 0
DO I=1,10000
  IF (IFLAG(I)==1) THEN
    ICOUNT = ICOUNT + 1          ④
    ITABLE(ICOUNT) = I
  ENDIF
ENDDO
DO ITIME=1,10
   :
  DO II=1,ICOUNT ◀ 反復数は2回
    I = ITABLE(II)
    A(I) = A(I) + 1.0            ⑤
  ENDDO
   :
ENDDO
   :
```

図 5-2-18 (2)

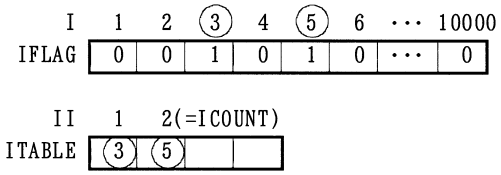


図 5-2-18 (3)

● 図 5-2-19 (1) では、I = 1, 1001, 2001, ... のとき、A(I) の値を書き出しています。下線の IF 文は、IF 文自体の存在に加え、IF 文の条件式で組込関数 MOD を使用しているため、計算時間がかかります。
DO ループを図 5-2-19 (2) のように 2 重ループに変更すると、IF 文を除去することができます。内側のループの反復 I が最大値の N を越えないように、下線部で組込関数 MIN を使用しています。

```
   :
DO I=1,N
  IF (MOD(I,1000)==1) PRINT *,A(I)
  A(I) = A(I) + 1.0
ENDDO
   :
```

図 5-2-19 (1)

→

```
   :
DO II=1,N,1000
  PRINT *,A(II)
  DO I=II,MIN(II+999,N)
    A(I) = A(I) + 1.0
  ENDDO
ENDDO
   :
```

図 5-2-19 (2)

- 図5-2-20 (1)では、図5-2-20 (3)のうち障害物以外の部分を計算します。障害物の判定を行う①のIF文は、図5-2-20 (2)のように1〜4の領域を別々に計算すれば除去することができます。

```
    :
DO J=1,N
  DO I=1,M
    IF ((ISTA<=I.AND.I<=IEND) .AND.
       (JSTA<=J.AND.J<=JEND)) THEN
    ELSE                          ①
      A(I,J) = A(I,J) + 1.0
    ENDIF
  ENDDO
ENDDO
    :
```

図 5-2-20 (1)

→

```
    :
DO K=1,4
  IF    (K==1) THEN
    I1=1;I2=M;J1=1;J2=JSTA-1          1の処理
  ELSEIF(K==2) THEN
    I1=1;I2=ISTA-1;J1=JSTA;J2=JEND    2の処理
  ELSEIF(K==3) THEN
    I1=IEND+1;I2=M;J1=JSTA;J2=JEND    3の処理
  ELSEIF(K==4) THEN
    I1=1;I2=M;J1=JEND+1;J2=N          4の処理
  ENDIF
  DO J=J1,J2
    DO I=I1,I2
      A(I,J) = A(I,J) + 1.0
    ENDDO
  ENDDO
ENDDO
    :
```

図 5-2-20 (2)

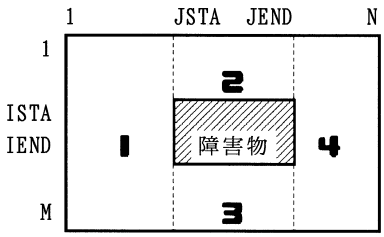


図 5-2-20 (3)

5-2-4-2　IF 文の順序の変更

AND 結合

1 つの IF 文内に複数の判定条件があって真になる割合が異なる場合、どの条件式を先にテストするかによって IF 文の計算量が変わります。

図 5-2-21 (1) のように IF 文の複数の条件式が AND 結合している場合、条件式は（一般に）左からテストされていき、条件が偽になった時点でそれより右の条件のテストは中止となります。図 5-2-21 (1) では左の条件が真になる割合が 9 割なので、全体の 9 割が右側の条件式も実行します。一方条件式の順序を逆にした図 5-2-21 (2) では、左の条件が真になる割合は 2 割なので、全体の 2 割のみが右側の条件式も実行します。従って、AND 結合の場合、図 5-2-21 (2) のように真の割合の低い条件を左で判定した方が、IF 文の計算量が少なくなります。

図 5-2-22 (1) (2) も全く同じ理由で、図 5-2-22 (2) の方が IF 文の計算量が少なくなります。

```
   :
DO I=1,1000
  IF (M(I)==1 .AND. K(I)==1) THEN
    :   ↳真の割合9割 ↳真の割合2割
  ENDIF              全体の9割がこちらの
ENDDO                条件式も実行する。
   :
```

図 5-2-21 (1)　×

→

```
   :
DO I=1,1000
  IF (K(I)==1 .AND. M(I)==1) THEN
    :   ↳真の割合2割 ↳真の割合9割
  ENDIF              全体の2割がこちらの
ENDDO                条件式も実行する。
   :
```

図 5-2-21 (2)　○

```
   :
DO I=1,1000
  IF (M(I)==1) THEN ➡真の割合9割
    IF (K(I)==1) THEN ➡真の割合2割
      :
    ENDIF
  ENDIF
ENDDO
   :
```

図 5-2-22 (1)　×

→

```
   :
DO I=1,1000
  IF (K(I)==1) THEN ➡真の割合2割
    IF (M(I)==1) THEN ➡真の割合9割
      :
    ENDIF
  ENDIF
ENDDO
   :
```

図 5-2-22 (2)　○

OR 結合

複数の条件式が OR 結合されている場合、図 5-2-23 (1) では全体の 2 割のみが左側の条件式のみで終了するのに対し、図 5-2-23 (2) では全体の 9 割が左側の条件式のみで終了するので、図 5-2-23 (2) のように真の割合の高い条件を左で判定した方が、IF 文の計算量が少なくなります。

```
   :
DO I=1,1000
  IF (K(I)==1 .OR. M(I)==1) THEN
    :   ↳真の割合2割 ↳真の割合9割
  ENDIF 全体の2割がこちらの
ENDDO   条件式のみで終了する。
   :
```

図 5-2-23 (1)　×

→

```
   :
DO I=1,1000
  IF (M(I)==1 .OR. K(I)==1) THEN
    :   ↳真の割合9割 ↳真の割合2割
  ENDIF 全体の9割がこちらの
ENDDO   条件式のみで終了する。
   :
```

図 5-2-23 (2)　○

多数の分岐先

図 5-2-24 (1) で、配列 M は、1 か 2 か 3 のいずれかの値になっているとします。図 5-2-24 (1) では、3 つの IF 文が全てテストされます。

一方 IF～ELSEIF～ENDIF を使用した図 5-2-24 (2) では、IF 文 (または ELSEIF 文) は（一般に）上から順にテストされ、条件が真になった時点でそれより下の ELSEIF 文は中止となります。従って全体の 1 割は一番上の条件式で終了するので、図 5-2-24 (1) よりも IF 文の計算量が少なくなります。

さらに図 5-2-24 (3) では、全体の 7 割は一番上の条件式で終了するので、IF 文の計算量はさらに少なくなります (前述の OR 結合と同様)。従って真の割合の高い条件を先にテストした方が、IF 文の計算量が少なくなります。

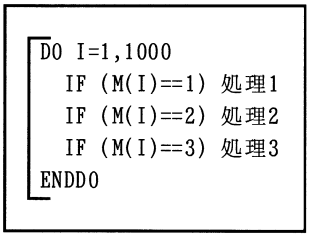


図 5-2-24 (1)　×

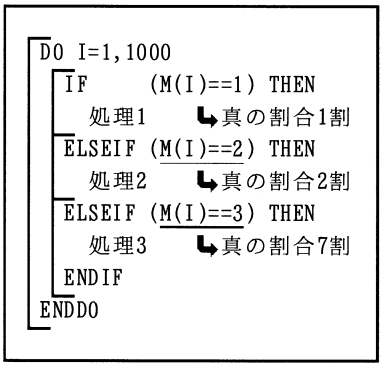


図 5-2-24 (2)　△

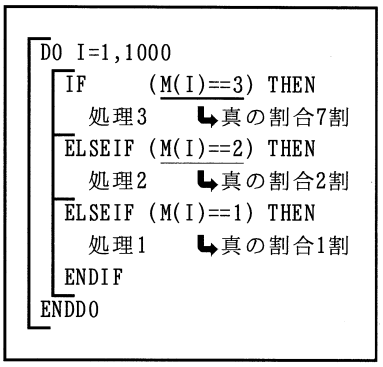


図 5-2-24 (3)　○

冗長な IF 文

以下のプログラムで、配列 A(I) の値はゼロ以上だとします。図 5-2-25 では、A(I) の値が 0～10 の場合、①と②の両方の IF 文が実行されます。一方図 5-2-26 では、A(I) の値が 0～5 のときは③の IF 文のみが実行されるため、IF 文の計算量が少なくなります（表参照）。

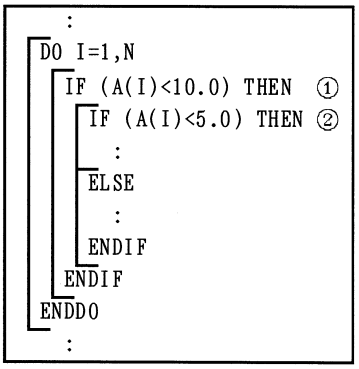


図 5-2-25

A(I)の値	図5-2-25	図5-2-26
0.0 ～ 5.0	① ②	③
5.0 ～10.0	① ②	③ ④

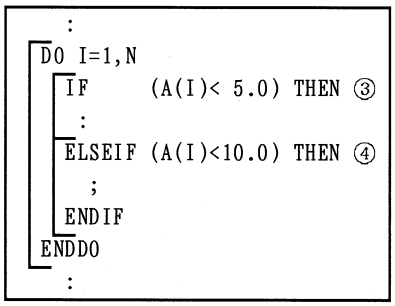


図 5-2-26

5-2-5　計算量のオーダーの減少

図 5-2-27 (1) の 2 重ループでは 100 × 100 = 10000 回の計算を行っています。これらを何らかの方法で図 5-2-27 (2) のように 1 重ループに変えることができれば、計算量は 1/100 となり、計算時間はほとんどゼロになります。つまり、あたりまえの話ですが、N^2 のオーダーの計算を N のオーダーの計算に減らすことができれば、計算時間は著しく減少します。また N が大きければ大きいほど、N^2 の計算は 2 乗で計算時間が増加してしまいます。このため、例えば N が小さいときは N^2 の計算部分がホットスポットでなかったのに、N を大きくした途端にホットスポットになってしまうという状況が発生します。

もちろん、本質的に N^2 の計算をしなければならない場合は仕方がないですが、本節で紹介するように、N の計算で済むのに N^2 の計算を行っているケースがあります。5-5-6 節でも同様の例を紹介します。

10000回の計算

```
    :
  N = 100
  DO J=1,N
    DO I=1,N
      :
    ENDDO
  ENDDO
    :
```

→

100回の計算

```
    :
  N = 100
  DO I=1,N
    :
  ENDDO
    :
```

図 5-2-27 (1)　　図 5-2-27 (2)

例 1

図 5-2-28 に示すように、1〜3の各要素内に①〜⑧の粒子が入っているとします。図 5-2-29 (2) に示す、各粒子番号とその粒子が所属する要素番号を対応付ける配列 IPART を使用して、各要素に所属する粒子の個数を調べ、配列 IELEM に入れるプログラムについて検討します。全要素数を NELEM、全粒子数を NPART とします。

オリジナル版の図 5-2-29 (1) のプログラムでは、図 5-2-29 (2) に示すように、[1] のループで J = 1 のとき、[2] のループを反復させて、各粒子が1の要素に所属しているかを全粒子について調べ、所属していれば配列 IELEM の1の要素の個数を 1 つ増やします。同様に J = 2 のとき2の要素に所属する粒子の個数を調べ、J = 3 のとき3の要素に所属する粒子の個数を調べます。

一方チューニング版の図 5-2-30 (1) のプログラムでは、図 5-2-30 (2) に示すように、[3] の粒子のループを反復させ、[4] でその粒子が所属する要素番号を調べ、配列 IELEM の該当する要素の個数を 1 つ増やします。

図 5-2-29 (1) の [1]，[2] の 2 重ループが、図 5-2-30 (1) では [3] の 1 重ループになったため、計算量は著しく減少します。

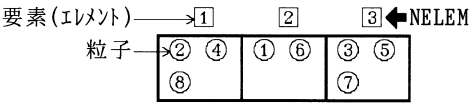


図 5-2-28

```
  PARAMETER(NELEM=3,NPART=8)
  INTEGER IPART(NPART),IELEM(NELEM))
    :
  DO J=1,NELEM                          [1]
    IELEM(J) = 0
    DO I=1,NPART                        [2]
      IF (J==IPART(I)) THEN
        IELEM(J) = IELEM(J) + 1
      ENDIF
    ENDDO
  ENDDO
    :
```

図 5-2-29 (1)

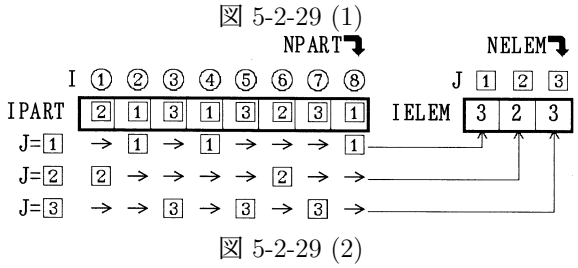


図 5-2-29 (2)

```
  PARAMETER(NELEM=3,NPART=8)
  INTEGER IPART(NPART),IELEM(NELEM)
    :
  DO J=1,NELEM
    IELEM(J) = 0
  ENDDO
  DO I=1,NPART                          [3]
    J = IPART(I)                        [4]
    IELEM(J) = IELEM(J) + 1
  ENDDO
    :
```

図 5-2-30 (1)

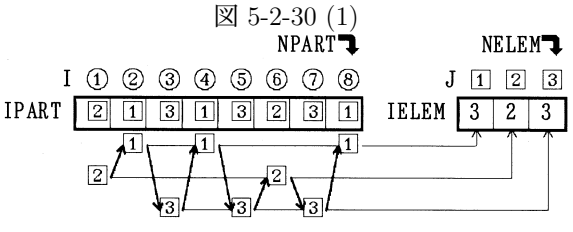


図 5-2-30 (2)

例 2

図 5-2-31 (1) の計算領域は[1]～[6]の要素と①～⑫の節点から構成されています。各要素の周囲の 4 つの節点には、左上から反時計回りに 1，2，3，4 の相対番号がついています。図 5-2-31 (2) に示すように、各要素とその相対番号に接続されている節点番号との対応を示す配列 IELEMENT があらかじめ作成されています。

本例のプログラムでは、配列 IELEMENT を使用して各要素の左隣、下隣、右隣、上隣の要素の番号を調べ、図 5-2-31 (3) に示す配列 ICONNECT を作成します。なお、図 5-2-31 (1) の計算領域は規則的になっていますが、不規則な（非構造格子）場合も同様です。

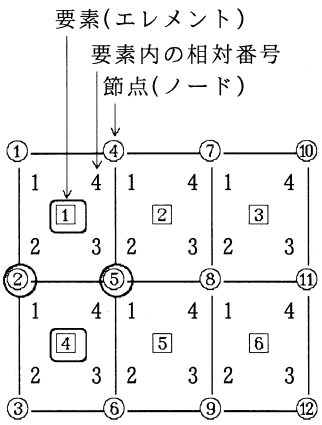


図 5-2-31 (1)

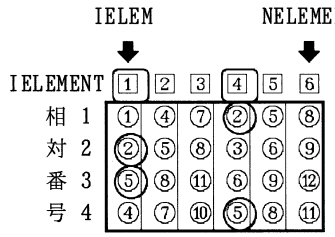


図 5-2-31 (2)

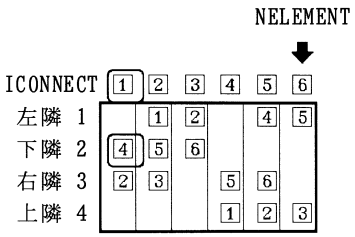


図 5-2-31 (3)

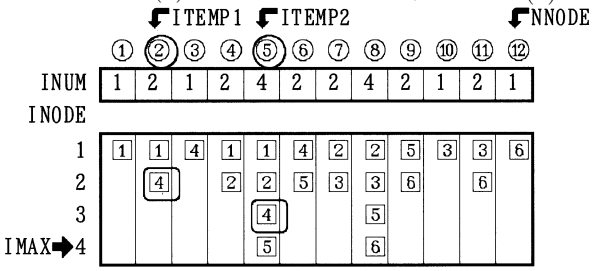


図 5-2-31 (4)

オリジナル版のプログラムを図 5-2-32 に示します。NELEMENT は全要素数、NNODE は全節点数を示します。

- 図 5-2-32 の (1) で、図 5-2-31 (3) の配列 ICONNECT をゼロクリアします。
- (2) の IELEM は隣接する要素を調べたい要素の番号、(3) の JELEM は IELEM に隣接しているかどうかを調べる相手の要素番号を示します。以下、IELEM＝[1]、JELEM＝[4] として説明します。
- (4) で自分の要素番号と相手の要素番号が同じときは、以後の処理をスキップします。
- (5) で配列 IELEMENT を使用して、[1]の要素に接続されている①，②，⑤，④の各節点が[4]の要素にも接続されているかどうかを調べます。本例では図 5-2-31 (2) に示すように、[1]の要素に接続されている②と⑤が[4]の要素にも接続されているため、②と⑤の ([1]の要素での) 相対番号である 2 と 3 を配列 IWORK に入れます。
- (5) の終了後、変数 ISUM には[1]と[4]の要素が共有する節点の数 (本例では 2) が入ります。
- (6) で[1]と[4]の要素が共有する節点の数が 2 の場合、[4]の要素は隣の要素であると判断されます。
- (7)～(10) で、[4]が上下左右のどの方向に隣接した要素であるかを調べます。図 5-2-31 (1) から分かるように、[1]の要素が隣の要素と共有する相対番号が 1 と 2 の場合は左隣、2 と 3 の場合は下隣、3 と 4 は場合の右隣、1 と 4 の場合は上隣の要素となります。本例では、上記下線に示すように配列 IWORK に 2 と 3 が入っているので (8) が真となり、相手の要素[4]は[1]の下隣の要素であると判断されます。
- (11) で相手の要素番号[4]を図 5-2-31 (3) の[1]の 2 番目（下隣なので）に代入します。

オリジナル版では、[1]～[6]の全要素間でお互いが互いに隣の要素になっているかどうかを調べるため、(2) と (3) は 2 重ループとなっており、計算量のオーダーは N^2 となります。

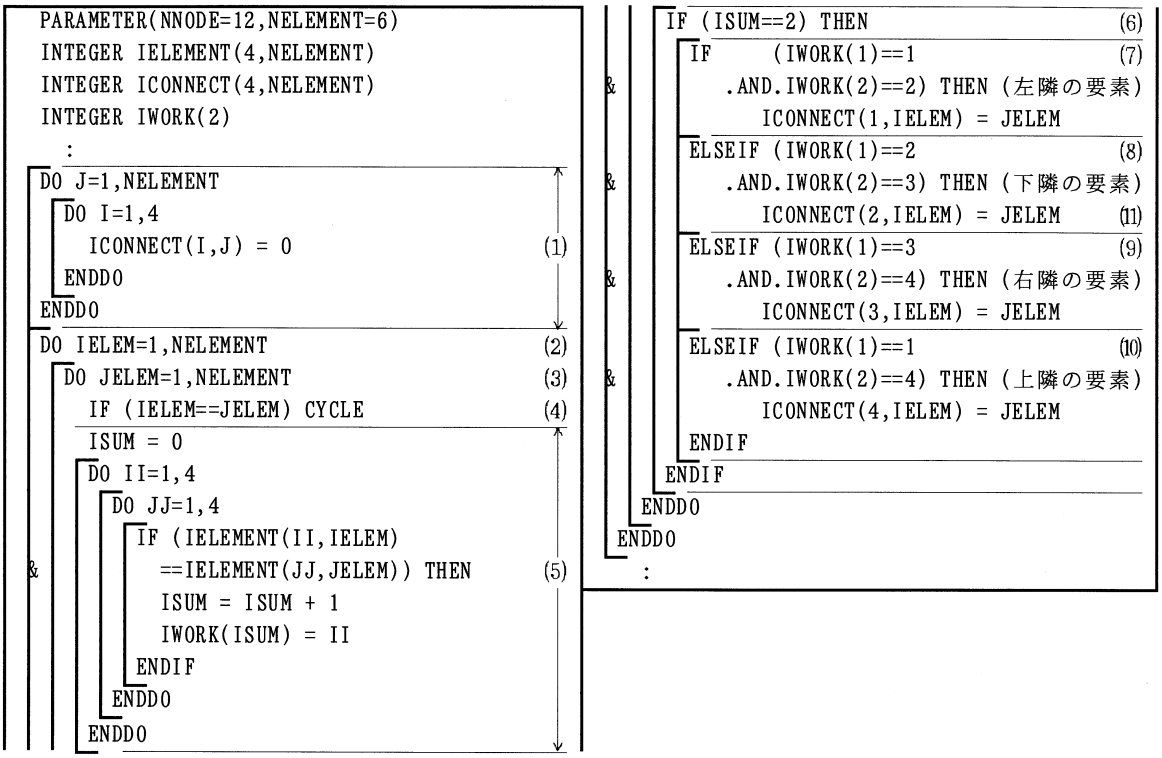


図 5-2-32

チューニング版のプログラムを図 5-2-33 に示します。

- まず図 5-2-31 (4) の下段に示す、各節点とその節点を所有する要素番号の対応を示す配列 INODE を作成します。配列 INODE の 1 次元目の大きさは、同じ節点を所有する要素数の最大値 (本例では 4) ですが、計算領域が不規則な場合は 4 より大きくなることもあるので、まずこの値を求めます。図 5-2-33 の [1] で図 5-2-31 (4) の上段に示す配列 INUM をゼロクリアし、[2] で配列 IELEMENT を使用して各節点を所有する要素数を求め、配列 INUM に入れます。
- [3] で配列 INUM 内の最大値 IMAX を求め、これを用いて [4] で配列 INODE を動的に確保します。
- [5] で配列 INUM を再びゼロクリアし、[6] で配列 INUM をカウンターとして使用して、図 5-2-31 (4) の下段に示すように、各節点ごとにその節点を所有する要素番号を配列 INODE に代入します。[6] の終了後、配列 INUM には再び各節点を所有する要素数が入ります。
- 以下、図 5-2-31 (1) で要素①の下隣の要素番号 (本例では④) を調べる場合を例に説明します。まず [7] で、隣接する要素を調べたい要素番号 IELEM = ① となります。
- [8] で K=1 のとき左隣、K=2 のとき下隣、K=3 のとき右隣、K=4 のとき上隣の要素を調べます。本例では下隣の要素を調べるため K=2 となり、[9] が実行され、変数 ITEMP1 と ITEMP2 に①の要素の相対番号 2, 3 の節点番号である②と⑤が入ります。
- 図 5-2-31 (1) から分かるように、①の要素の下側に要素が存在するとすれば、その要素は②と⑤の節点を所有するはずです。従って、図 5-2-31 (4) の②と⑤を所有する自分 (本例では①) 以外の要素から、共通する要素番号を探します。まず [10] で配列 INODE を使用して、ITEMP1 (本例では②) の節点を所有する要素番号である①を変数 JELEM に代入します。
- ①の要素は自分自身なので、[11] で以後の処理をスキップします。
- 次に [10] で変数 JELEM に④を代入します。[12] で、④の要素が ITEMP2 (本例では⑤) の節点を所有する要素の中にも含まれているかどうかを調べます。本例では含まれているので、④の要素は②と⑤の節点を所有し、要素①の下隣の要素であると判断されます。
- [13] で④を配列 ICONNECT に代入します。

チューニング版では、[7] のループに示すように計算量のオーダーは N となり、オリジナル版よりも計算量は著しく減少します。なお、要素①の隣が④だと判明したとき、要素④の隣も当然ながら①なので、④の INODE も同時に更新すれば計算量は半分になりますが、説明がややこしくなるのでこの機能は省略しました。

```
  PARAMETER(NNODE=12,NELEMENT=6)
  INTEGER IELEMENT(4,NELEMENT)
  INTEGER ICONNECT(4,NELEMENT)
  INTEGER,ALLOCATABLE::INODE(:,:)
  INTEGER INUM(NNODE)
    :
  DO J=1,NELEMENT
    DO I=1,4
      ICONNECT(I,J) = 0
    ENDDO
  ENDDO
  DO I=1,NNODE
    INUM(I) = 0                          [1]
  ENDDO
  DO J=1,NELEMENT
    DO I=1,4
      ITEMP = IELEMENT(I,J)              [2]
      INUM(ITEMP) = INUM(ITEMP)+1
    ENDDO
  ENDDO
  IMAX = 0
  DO I=1,NNODE
    IMAX = MAX(IMAX,INUM(I))           [3]
  ENDDO
  ALLOCATE(INODE(IMAX,NNODE))          [4]
  DO I=1,NNODE
    INUM(I) = 0                          [5]
  ENDDO
  DO J=1,NELEMENT
    DO I=1,4
      ITEMP = IELEMENT(I,J)
      INUM(ITEMP) = INUM(ITEMP)+1        [6]
      INODE(INUM(ITEMP),ITEMP) = J
    ENDDO
  ENDDO
```

```
  DO IELEM=1,NELEMENT                    [7]
    DO K=1,4                             [8]
      IF (K==1) THEN          (左隣の要素)
        ITEMP1 = IELEMENT(1,IELEM)
        ITEMP2 = IELEMENT(2,IELEM)
      ELSEIF (K==2) THEN      (下隣の要素)
        ITEMP1 = IELEMENT(2,IELEM)       [9]
        ITEMP2 = IELEMENT(3,IELEM)
      ELSEIF (K==3) THEN      (右隣の要素)
        ITEMP1 = IELEMENT(3,IELEM)
        ITEMP2 = IELEMENT(4,IELEM)
      ELSEIF (K==4) THEN      (上隣の要素)
        ITEMP1 = IELEMENT(1,IELEM)
        ITEMP2 = IELEMENT(4,IELEM)
      ENDIF
      DO J=1,INUM(ITEMP1)                [10]
        JELEM = INODE(J,ITEMP1)          [10]
        IF (IELEM==JELEM) CYCLE          [11]
        DO I=1,INUM(ITEMP2)              [12]
          IF (JELEM/=INODE(I,ITEMP2)) CYCLE [12]
          ICONNECT(K,IELEM) = JELEM      [13]
          GOTO 9
        ENDDO
      ENDDO
    9 CONTINUE
    ENDDO
  ENDDO
    :
```

図 5-2-33

例 3

本例はゲノム解析のクラスタリングという計算で現れた例です。なお、実際の計算では三角行列を扱い、計算が進むにつれて計算領域の大きさが少しずつ小さくなりますが、以下では説明を分かりやすくするため、計算領域は正方行列で、計算が進んでも大きさは変わらないとします。また更新する行、列と更新方法も実際とは異なります。

オリジナル版のプログラムを図 5-2-34 に、動作を図 5-2-35 (1)〜(4) に示します。

- まず図 5-2-34 の①のループ反復が ICOUNT=1 となります。図 5-2-35 (1) に示すように、配列 M の全要素の最小値 (本例では 0) を求め、その座標を ILOC, JLOC (本例では 4, 2) とします。これを図 5-2-34 の②で行います。なお、図 5-2-35 (1) のように最小値 (本例では 0) が複数ある場合は、一番最初に見つかった要素の座標を採用します。
- 図 5-2-35 (2) に示すように、最小値の要素が含まれる行と列の値を更新します。これを図 5-2-34 の③で行います。
- ①のループ反復が ICOUNT=2 となり、図 5-2-35 (3) に示すように、再び配列 M の全要素の最小値 (本例では 0) とその座標 ILOC, JLOC (本例では 2, 3) を求めます。
- 図 5-2-35 (4) に示すように、最小値の要素が含まれる行と列を更新します。以後同様に処理を行います。

このプログラムでは、2 次元配列 M の全要素から最小値を求めるため、④のループが二重ループになっており、計算量のオーダーは N^2 になります。

```
  PARAMETER(N=5)
  INTEGER M(N,N)
    :
┌─DO ICOUNT=1,100                          ①
│   ITEMP = M(1,1)
│   ILOC = 1
│   JLOC = 1
│  ┌─DO J=1,N                              ④
│  │ ┌─DO I=1,N                            ④
│  │ │ ┌─IF (M(I,J)<ITEMP) THEN            ②
│  │ │ │   ITEMP = M(I,J)
│  │ │ │   ILOC = I
│  │ │ │   JLOC = J
│  │ │ └─ENDIF
│  │ └─ENDDO
│  └─ENDDO
│  ┌─DO I=1,N
│  │   M(I,JLOC) を更新
│  └─ENDDO
│  ┌─DO J=1,N                              ③
│  │   M(ILOC,J) を更新
│  └─ENDDO
└─ENDDO
  END
```

図 5-2-34

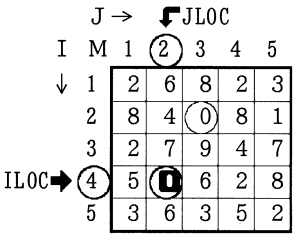


図 5-2-35 (1)

↓

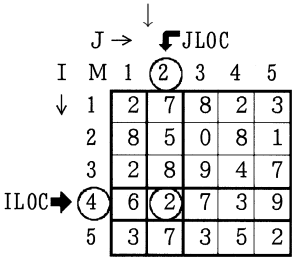


図 5-2-35 (2)

↓

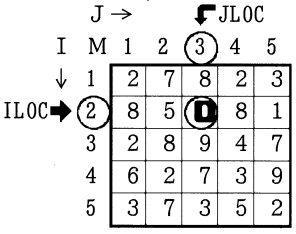


図 5-2-35 (3)

↓

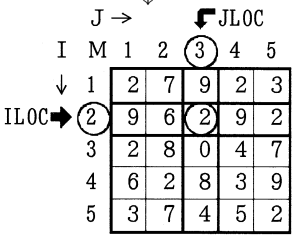


図 5-2-35 (4)

↓

チューニング版の動作の概要を説明します。計算を開始する前に、図 5-2-36 に示すように、配列 M の各列ごとの最小値（太い文字）を配列 IMIN に、最小値の I 方向の座標を配列 IPOS に保管します。そして計算が開始した後は、配列 M の値が更新されてその列の最小値が変わったときのみ、IMIN と IPOS を更新します。

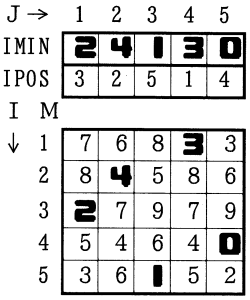


図 5-2-36

チューニング版のプログラムを図 5-2-38 (1) に、動作を図 5-2-38 (1)〜(3) に示します。オリジナル版と違い、配列 M の大きさは 10 × 10 であるとします。また説明を分かりやすくするため、各要素は簡単な値になっています。

- 計算に入る前に、図 5-2-38 (1) に示すように、配列 M の各列の最小値（太い数字）とその I 方向の座標を求め、配列 IMIN と IPOS に入ります。これを図 5-2-37 の[1]からコールする[2]のサブルーチン CHECK で行います。
- [3]のループ反復が ICOUNT=1 となり、図 5-2-38 (1) の配列 IMIN の中を調べて最小値 (本例では 1) と J 方向の座標 JLOC (本例では 4) を求め、さらに配列 IPOS を使用して I 方向の座標 LOC (本例では 5) を求めます。これを[4]で行います。
- 図 5-2-38 (2) に示すように、最小値の要素が含まれる行と列の値を更新します。これを[5]で行います。
- この更新によって、各列の最小値やその位置が変わった場合は配列 IMIN と IPOS の値を変更します。まず、図 5-2-38 (2) の 4 列目は、列内の全要素が更新されたので、改めてこの列内の最小値 (本例では 2) とその位置 ILOC (本例では 9) を求め、図 5-2-38 (3) の [1] のように更新します。これを[7]で行います。
- 図 5-2-38 (2) の 1〜3 列目では、元の最小値が 3、更新した値は 1 なので、更新した値が新しい最小値になり、その I 方向の座標は ILOC (本例では 5) になります。従って図 5-2-38 (3) の [2] のように更新します。これを[8]で行います。
- 図 5-2-38 (2) の 5〜7 列目では、元の最小値が 3、更新した値も 3 で同じ値です。このときは、オリジナル版のプログラムと同じように、2 つの「3」のうち最初に現れた（各列の上に位置する）要素を採用します。従って図 5-2-38 (3) の [3] のように更新します。これを[9]で行います。
- 図 5-2-38 (2) の 8〜10 列目では、元の最小値 3、更新した値は 5 です。従って 8 列目と 10 列目では各列の最小値は元の 3 のままで変わりません。しかし 9 列目では更新前に最小値 3 が入っていた部分を更新したので、改めてこの列の最小値 (本例では 4) と I 方向の座標 (本例では 8) を求める必要があります。従って図 5-2-38 (3) の [4] のように更新します。これを[10]で行います。以上で配列 IMIN と IPOS が正しい値に更新されました。
- [3]のループ反復が ICOUNT=2 となります。
- [4]で再び配列 IMIN の最小値を求めます。図 5-2-38 (3) のように最小値 (本例では 1) が複数あるときは、オリジナル版のプログラムと同じように、最小に現れた (IMIN の最も左の) 要素を採用します。
- 以後同様に処理を行います。

チューニング版では、[11]と[6]のループは 1 重ループになっており、計算量のオーダーは N^2 から N に減少しました。その結果、特に N が大きいときは、計算時間が著しく短縮します。

```
PARAMETER(N=5)
INTEGER M(N,N)
INTEGER IPOS(N),IMIN(N)
  :
DO J=1,N                                   1
  CALL CHECK(M,N,J,IMIN(J),IPOS(J))
ENDDO
DO ICOUNT=1,100                            3
  ITEMP = IMIN(1)
  JLOC = 1
  DO J=1,N                                 11
    IF (IMIN(J)<ITEMP) THEN
      ITEMP = IMIN(J)                      4
      JLOC = J
    ENDIF
  ENDDO
  ILOC= IPOS(JLOC)
  DO I=1,N
    M(I,JLOC) を更新
  ENDDO                                    5
  DO J=1,N
    M(ILOC,J) を更新
  ENDDO
  DO J=1,N                                 6
    IF (J==JLOC) THEN                      7
      CALL CHECK(M,N,J,IMIN(J),IPOS(J))    7
    ELSE
      IF (M(ILOC,J)<IMIN(J)) THEN
        IMIN(J) = M(ILOC,J)                8
        IPOS(J) = ILOC
      ELSEIF (M(ILOC,J)==IMIN(J)) THEN     9
        IPOS(J) = MIN(IPOS(J),ILOC)        9
      ELSEIF (ILOC==IPOS(J)) THEN          10
        CALL CHECK(M,N,J,IMIN(J),IPOS(J))  10
      ENDIF
    ENDIF
  ENDDO
ENDDO
END

SUBROUTINE CHECK(M,N,J,IMIN,IPOS)
INTEGER M(N,N)
IMIN = M(1,J)
IPOS = 1
DO I=1,N                                   2
  IF (M(I,J)<IMIN) THEN
    IMIN = M(I,J)
    IPOS = I
  ENDIF
ENDDO
END
```

図 5-2-37

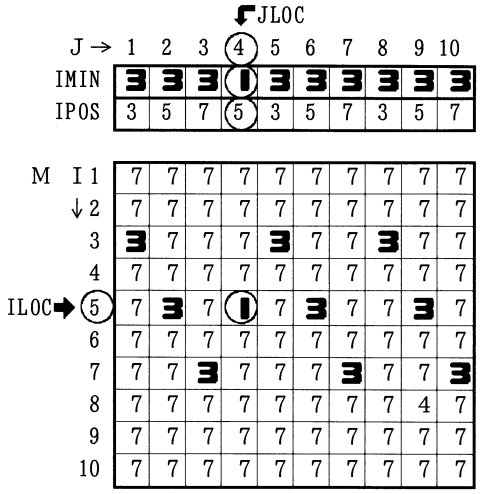


図 5-2-38 (1)

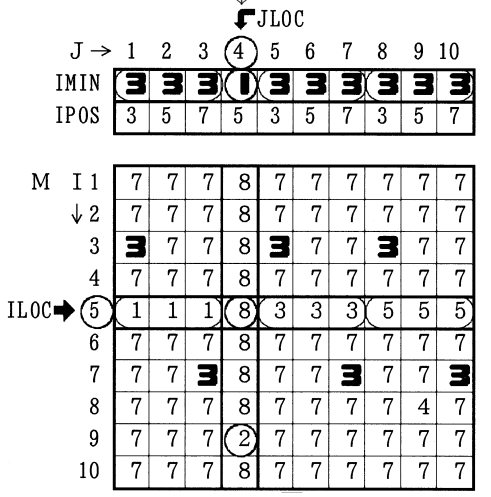


図 5-2-38 (2)

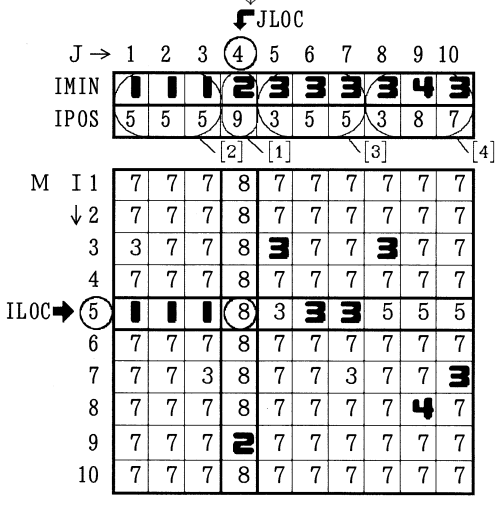


図 5-2-38 (3)

5-3　マシン環境に依存したチューニング

本節では、マシン環境（主にコンパイラによる最適化）に依存するチューニングを紹介します。従って、マシン環境によって効果がある場合とない場合があります。

複数の演算器の同時実行

加算、乗算などの演算器を複数持っているマシンでは、複数の演算器が同時に動くと速度が向上します。通常はこのことを意識する必要はありませんが、以下のような場合、パフォーマンスに影響する可能性があります（ただし、以下の例で効果が出るかどうかは、マシン環境によって異なります）。

- 図 5-3-1 (1) の配列 A(I) は、〓に示すように、次の反復の右辺で A(I-1) として参照されています。このような依存関係がある場合、右辺の A(I-1) の位置に注意する必要があります。I=1,2 の場合の式を書き下すと図 5-3-1 (2) のようになります。まず①が左から順に計算されて A(1) が求まり、求めた A(1) を使用して②が左から順に計算されます。すなわち A(1) に ↘ の依存関係があるため、①が終了してからでないと②を計算することができません。
  一方右辺の A(I-1) を式の最後に移動した図 5-3-1 (3) (4) では、①と[1]に依存関係がないため、①と[1]が複数の演算器で同時に計算され、速度が向上します。そして両方の計算が終了した後、②が計算されます。
  以上をまとめると、演算器を複数持っているマシンでは、本例の A(I-1) のように、最内側ループの反復に依存関係のある配列の項 (本例では A(I-1)) が式の右辺にある場合、その項は（試行錯誤で）式のなるべく後ろに置いた方が速くなることがあります。このようなパターンはガウスザイデル法や SOR 法などの反復解法で発生します。なお、図 5-3-1 (5) に 1 次元マルチカラー法の例を示しますが、この方法の場合は図 5-3-1 (6) に示すように各式間に依存関係がないため、A(I-1) の位置にかかわらず複数の演算器が同時に動きます。

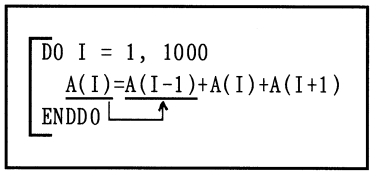


図 5-3-1 (1)

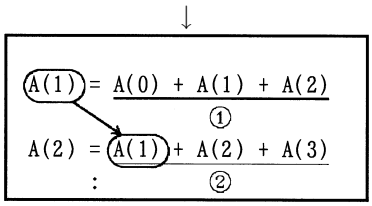


図 5-3-1 (2)

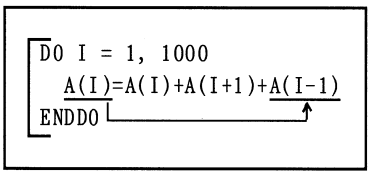


図 5-3-1 (3)

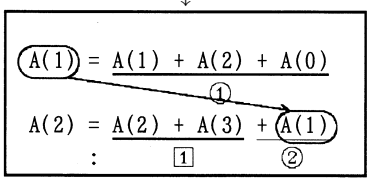


図 5-3-1 (4)

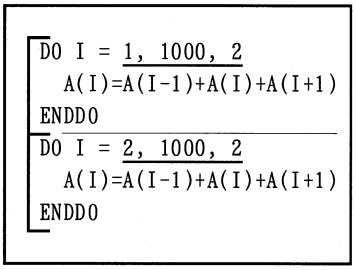


図 5-3-1 (5)

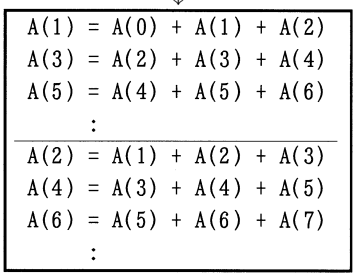


図 5-3-1 (6)

- 図 5-3-1 (7) では、①は左から順に計算されますが、図 5-3-1 (8) のようにカッコをつけると、①と[1]が 2 つの演算器で同時に行われるため、パフォーマンスが向上する可能性があります。

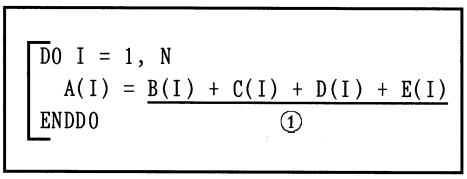


図 5-3-1 (7)

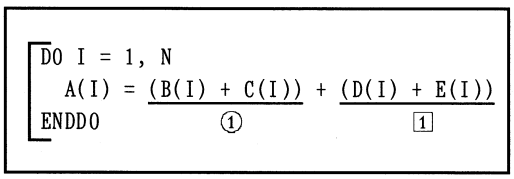


図 5-3-1 (8)

反復回数が少ない DO ループの展開

1 つの DO ループに到達するごとに、ループカウンターの初期設定と呼ばれるオーバーヘッドがかかります。図 5-3-2 (1) の上図では、□で囲んだ内側のループに 100000 回到達するので、100000 回のオーバーヘッドがかかります。この例では、内側のループの反復回数 (3 回) が明示的に指定されていて、かつ回数が少ないので、下図のようにループを展開することができます。これによって内側のループ自体がなくなるため、ループカウンターの初期設定のオーバーヘッドもなくなります。またこれにより、元のループの内部と外部の両方にまたがる最適化が促進される可能性があります。

図 5-3-2 (2) もループを展開した例です。ループを展開したことにより、①の命令が不要となりました。

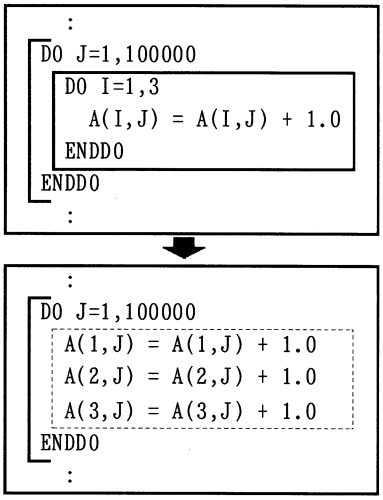


図 5-3-2 (1)

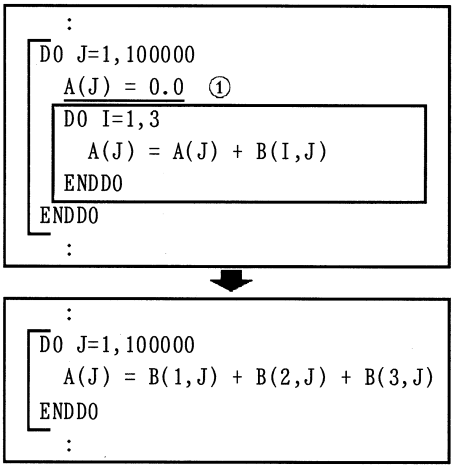


図 5-3-2 (2)

- 図 5-3-2 (3) で、外側のループを展開し、さらに変更したところ、3 × N 回実行していた①の計算が、②のように N 回に減少しました。このように、反復回数が明示的に指定されていて、かつ少ない DO ループを展開すると、チューニングの展望が開ける場合があります。

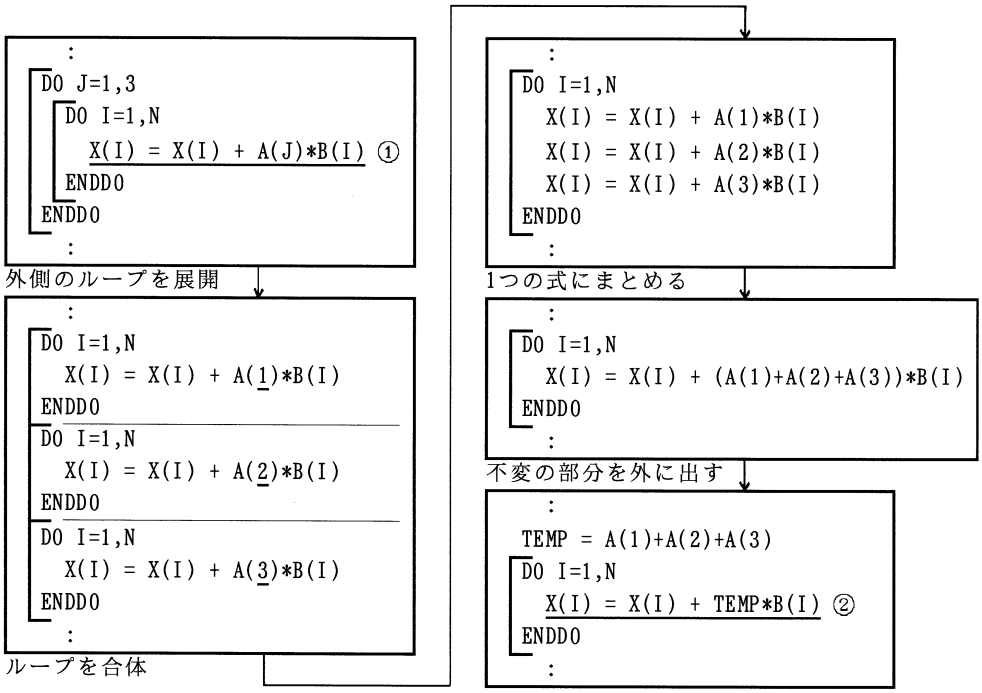


図 5-3-2 (3)

ループの分割と合体

図 5-3-3 (1) (2) のように、ループを分割した場合と合体（融合とも言います）した場合のどちらが速いか（または変わらないか）は、マシン環境やプログラムに依存するため何とも言えません。

- 分割した方が速くなる理由　ループ内に配列が多く、4-4 節で説明したキャッシュミスが発生している場合、分割すると 1 つのループ当たりの配列が少なくなって回避できる可能性があります。
- 合体した方が速くなる理由　コンパイラは各ループごとに最適化を行うため、分割した場合、以下の①と②は個別に最適化されますが、合体すると、③と④の全体にまたがる最適化が促進される可能性があります。また分割した場合、①でキャッシュに入った配列 A を、②で再びキャッシュに入れなければならない可能性がありますが、合体すると、④で再びキャッシュに入れる必要がないため、合体した方が速くなることがあります。なお、コンパイラが複数のループを自動的に合体する場合もあります。

  実用上は、ホットスポットの複数のループが論理的に分割／合体できる場合、試しに分割／合体し、パフォーマンスが変化するかどうか試してみて下さい。

```
    :
DO I = 1, N
  A(I) = B(I) + C(I)    ①
ENDDO
DO I = 1, N
  D(I) = A(I) + E(I)    ②
ENDDO
    :
```

↔

```
    :
DO I = 1, N
  A(I) = B(I) + C(I)    ③
  D(I) = A(I) + E(I)    ④
ENDDO
    :
```

図 5-3-3 (1)　分割　○／×

図 5-3-3 (2)　合体　×／○

スカラー変数と配列

DO ループ内で使用する一時変数を、図 5-3-4 (1) のように配列にした場合と、図 5-3-4 (2) のようにスカラー変数にした場合、どちらが速いか（あるいは差がないか）は、マシン環境やプログラムに依存するのでなんとも言えません。

また 3 種類の配列を、図 5-3-5 (1) (2) のように 1 つの 2 次元配列にした場合と、図 5-3-5 (3) のように 3 つの 1 次元配列にした場合も、どれが速いか（あるいは差がないか）は何とも言えません。

ホットスポットでこのようなケースに遭遇した場合は試行錯誤で試してみて下さい。

```
    :
DO I = 1, N
  T(1) = ～
  T(2) = ～
  T(3) = ～
  A(I) = T(1) + T(2) + T(3)
ENDDO
    :
```

↔

```
    :
DO I = 1, N
  T1 = ～
  T2 = ～
  T3 = ～
  A(I) = T1 + T2 + T3
ENDDO
    :
```

図 5-3-4 (1)

図 5-3-4 (2)

```
DIMENSION A(3,N)
    :
DO I = 1, N
  A(1,I) = ～
  A(2,I) = ～
  A(3,I) = ～
ENDDO
    :
```

↔

```
DIMENSION A(N,3)
    :
DO I = 1, N
  A(I,1) = ～
  A(I,2) = ～
  A(I,3) = ～
ENDDO
    :
```

↔

```
DIMENSION A1(N),A2(N),A3(N)
    :
DO I = 1, N
  A1(I) = ～
  A2(I) = ～
  A3(I) = ～
ENDDO
    :
```

図 5-3-5 (1)

図 5-3-5 (2)

図 5-3-5 (3)

インライン展開

図 5-3-6 (1) のように何度もコールされる行数の少ないサブルーチンやファンクションを、図 5-3-6 (2) のようにコールする側のルーチンに埋め込むことをインライン展開と呼びます。インライン展開を行うとパフォーマンスが向上する可能性があります。

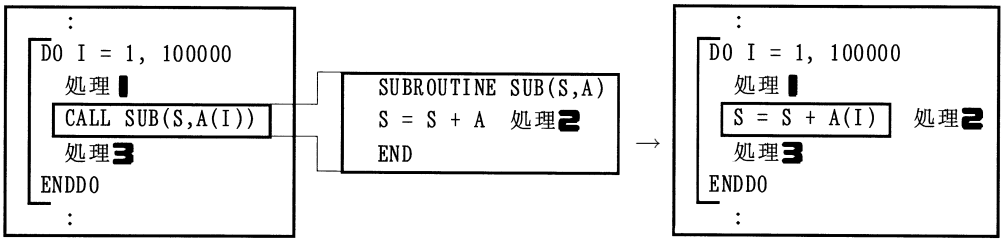


図 5-3-6 (1)　　図 5-3-6 (2)

パフォーマンスが向上する理由は主に次の 3 つですが、このうち (1) の効果が大きいようです。

(1) 図 5-3-6 (1) では処理 1，2，3 が分断されているため、コンパイラは各処理内での最適化を行いますが、図 5-3-6 (2) では処理 1，2，3 が連続するため、全体にまたがる最適化が行われる可能性があります。

(2) サブルーチンコールを行うと、内部的にレジスターの内容の保管などのオーバーヘッドがかかりますが、インライン展開するとサブルーチンコールがなくなるためオーバーヘッドもなくなります。

(3) データと同様に、機械語命令もメモリーから命令キャッシュに転送され、さらに命令レジスターに転送されます (4-1 節参照)。従って機械語命令も、実行される順序がメモリー上で連続している方が命令キャッシュミスが発生しにくくなります。インライン展開前の図 5-3-7 (1) では、メモリー上の機械語命令のうち①と②の間、および③と④の間が不連続になりますが、インライン展開後の図 5-3-7 (2) では、処理 1，2，3 の機械語命令がメモリー上で連続するので、命令キャッシュミスは発生しにくくなります。

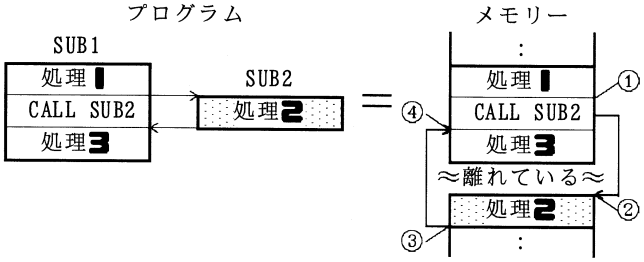


図 5-3-7 (1)　インライン展開前

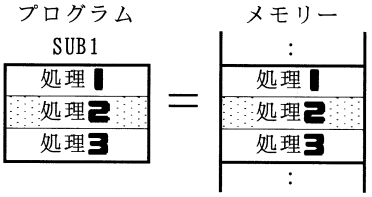


図 5-3-7 (2)　インライン展開後

インライン展開の手順を説明します。まず、コールされる回数が非常に多く、行数の少ないサブルーチンまたはファンクションを選択します。コールされる回数は prof コマンドや gprof コマンド (3-4 節参照) で調べることができます。

次に、選択したサブルーチンやファンクションをコールする側のルーチンに埋め込みます（後述するように、コンパイラで自動的に行う方法と、手作業で行う方法があります）。なお、プログラムの見やすさの観点からサブルーチンのままにしておきたい場合、（論理的に可能ならば）図 5-3-8 (1) (2) のように DO ループ命令をサブルーチン側に移す方法もあります。これは正確にはインライン展開ではありませんが、サブルーチンコールの回数が減少するため、同等の効果があります。

インライン展開が終了したら、パフォーマンスが向上したかどうか調べ、向上していれば採用します。インライン展開は、本当に効果のある、必要最小限のルーチンのみに対して行って下さい。

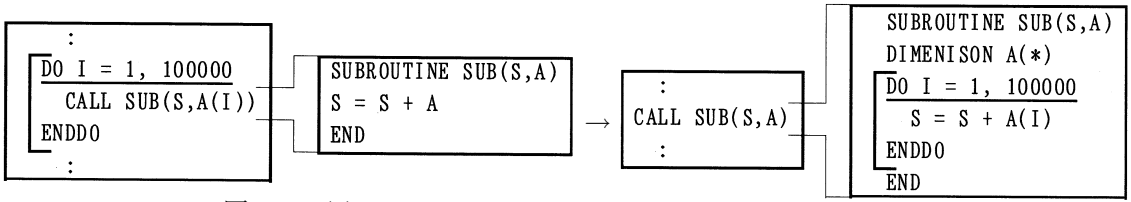


図 5-3-8 (1)　　図 5-3-8 (2)

インライン展開を自動的に行う機能を持つコンパイラもあります。自動で行うか、手作業で行うか、どちらも一長一短があります。自動で行う場合、プログラムが複雑な場合はコンパイラが誤って展開してしまう危険性 (2-2 節参照) があることと、他マシンに移植したときに同じ機能があるかどうか分からないという互換性の問題があります。

一方、手作業で行う場合の問題点ですが、例えば図 5-3-9 (1) のようにサブルーチン SUB が 2 回呼ばれていて、図 5-3-9 (2) のように両方とも手作業でインライン展開したとします。すると同じ「処理 A」が 2 箇所に現れてしまうため、プログラムの保全性が悪くなります。自動で行う場合は、サブルーチン自体は元の状態のままなので、この問題は発生しません。

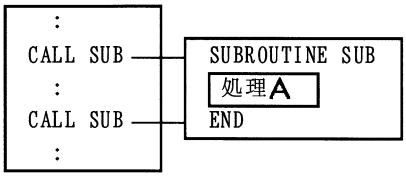


図 5-3-9 (1)

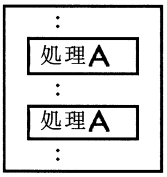
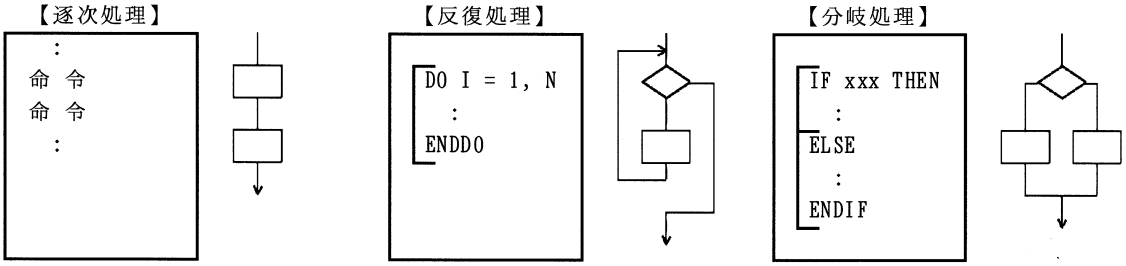


図 5-3-9 (2)

構造化プログラミング

プログラミングスタイルは人によってまさに千差万別です。整然としていて非常にわかり易いプログラムもあれば、GOTO 文を多用していて制御構造を把握するのが極めて困難な、いわゆるスパゲッティプログラムもあります。

さて、以下に示す 3 種類の要素を使用し、なるべく GOTO 文を使用しないプログラミングスタイルを構造化プログラミングと言います。構造化プログラミングのスタイルで書かれたプログラムは（普通の）人にとって分かりやすくなるとともに、コンパイラにとっても解析しやすくなるため、（一般に）最適化が促進されます。このため、ホットスポットではなるべく構造化プログラミングのスタイルで書くことをお勧めします。



構造化プログラミングによってパフォーマンスが向上する例を以下に示します（ただしマシン環境に依存します）。図 5-3-10 (1) と図 5-3-10 (2) は論理的に同じですが、構造化プログラミングのスタイルで書かれた図 5-3-10 (2) は（マシン環境によっては）最適化が促進され、パフォーマンスが向上します。なお、構造化プログラミングのスタイルで書いてもパフォーマンスが変わらないか、却って遅くなる場合もあります。

```
      :
    DO I = 1, 1000
      GOTO (10, 20, 30) M(I)
10    A(I) = A(I) + 1.0
      GOTO 40
20    A(I) = A(I) + 2.0
      GOTO 40
30    A(I) = A(I) + 3.0
40    CONTINUE
    ENDDO
      :
```

図 5-3-10 (1)

```
      :
DO I = 1, 1000
  IF     (M(I) == 1) THEN
    A(I) = A(I) + 1.0
  ELSEIF (M(I) == 2) THEN
    A(I) = A(I) + 2.0
  ELSEIF (M(I) == 3) THEN
    A(I) = A(I) + 3.0
  ENDIF
ENDDO
  :
```

図 5-3-10 (2)

5-4　ループアンローリング

前節と同じく、ループアンローリングもマシン環境に依存したチューニング方法ですが、前節が長くなりすぎたので節を改めます。なお、ループアンローリングは 4-5 節で紹介したブロック化と並んで重要なチューニング技法ですが、実際に使用する局面はそれほど多くはありません。

ループアンローリングとは

DO ループの反復回数を減らし、その代わりにループ内のステートメント数を増やすことをループアンローリングと呼びます。例えば図 5-4-1 (1) の DO ループを 2 段でアンローリングすると、図 5-4-1 (2) のようになります。ループ反復を 2 反復ごとにし、その代わりにステートメントを 2 行とし、配列の添字を「I」、「I+1」とします。同様に 3 段でアンローリングすると図 5-4-1 (3) のようになります。6 段でアンローリングした場合は、元の DO ループの反復回数が 6 回なので、図 5-4-1 (4) のようにループが展開されて消滅します。

```
   :
DO I = 1, 6
  A(I) = B(I)
ENDDO
   :
```

図 5-4-1 (1)　オリジナル

```
   :
DO I = 1, 6, 2
  A(I)   = B(I)
  A(I+1) = B(I+1)
ENDDO
   :
```

図 5-4-1 (2)　2 段

```
   :
DO I = 1, 6, 3
  A(I)   = B(I)
  A(I+1) = B(I+1)
  A(I+2) = B(I+2)
ENDDO
   :
```

図 5-4-1 (3)　3 段

```
   :
  A(1) = B(1)
  A(2) = B(2)
  A(3) = B(3)
  A(4) = B(4)
  A(5) = B(5)
  A(6) = B(6)
   :
```

図 5-4-1 (4)　6 段

図 5-4-2 (1) のように DO ループ内に含まれるステートメント数が 2 行以上の場合、2 段でアンローリングする方法は図 5-4-2 (2) と図 5-4-2 (3) の 2 通りあります。図 5-4-2 (3) の場合、DO ループ内で更新される変数 TEMP を TEMP1, TEMP2 (2 段でアンローリングする場合) のように区別する必要があります。図 5-4-2 (2) と図 5-4-2 (3) は最適化の影響でパフォーマンスに差が出る可能性があります。

```
   :
DO I = 1, 6
  TEMP = B(I)
  A(I) = TEMP
ENDDO
   :
```

図 5-4-2 (1)　オリジナル

```
   :
DO I = 1, 6, 2
  TEMP   = B(I)
  A(I)   = TEMP
  TEMP   = B(I+1)
  A(I+1) = TEMP
ENDDO
   :
```

図 5-4-2 (2)　2 段

```
   :
DO I = 1, 6, 2
  TEMP1  = B(I)
  TEMP2  = B(I+1)
  A(I)   = TEMP1
  A(I+1) = TEMP2
ENDDO
   :
```

図 5-4-2 (3)　2 段

図 5-4-3 (1) の下線部のようにループ反復に依存関係がある場合、注意が必要です。各図の右側に I=1 のときの具体的なステートメントを書き下します。図 5-4-3 (1) (2) では例えば T(2)=A(1)=T(1)=A(0)（これが正解）ですが、図 5-4-3 (3) では T(2)=A(1) となり、オリジナルと計算結果が変わってしまうので誤りです。

```
   :
DO I = 1, 6
  T(I) = A(I-1)      T(1)=A(0)
  A(I) = T(I)        A(1)=T(1)
ENDDO                T(2)=A(1)
   :                 A(2)=T(2)
                        :
```

図 5-4-3 (1)　オリジナル

```
   :
DO I = 1, 6, 2       T(1)=A(0)
  T(I)   = A(I-1)    A(1)=T(1)
  A(I)   = T(I)      T(2)=A(1)
  T(I+1) = A(I)      A(2)=T(2)
  A(I+1) = T(I+1)       :
ENDDO
   :
```

図 5-4-3 (2)　2 段　○

```
   :
DO I = 1, 6, 2       T(1)=A(0)
  T(I)   = A(I-1)    T(2)=A(1)
  T(I+1) = A(I)      A(1)=T(1)
  A(I)   = T(I)      A(2)=T(2)
  A(I+1) = T(I+1)       :
ENDDO
   :
```

図 5-4-3 (3)　2 段　×

アンローリングの種類

ループアンローリングを行う目的は主に以下の 2 つですが、本書では (1) についてのみ説明します。(2) は参考文献 [16] などを参照してください。

(1) (多重ループの) 外側のループのアンローリングによる、ロードとストアの削除
(2) (多重ループの) 内側のループのアンローリングによる、命令のオーバーラップ

4 章で説明したように、メモリーに入っているデータはいったんデータキャッシュ（以後キャッシュ）に入り、さらにデータレジスター（以後レジスター）に入って演算が行われます。4 章で説明したキャッシュチューニングは、なるべくキャッシュ内に入っているデータを使用し、キャッシュミス (図 5-4-4 の点線に示す矢印) を少なくするのが目的でした。

図 5-4-4 のキャッシュからレジスターへのデータの移動をロード、その反対をストアと呼びます。ロードやストアは演算を行うのと同程度に時間がかかります。外側のループのアンローリングは、なるべくレジスター内に入っているデータを使用し、ロードとストアの回数を少なくするのが目的で、キャッシュチューニングの一段上のレベルのチューニングになります。

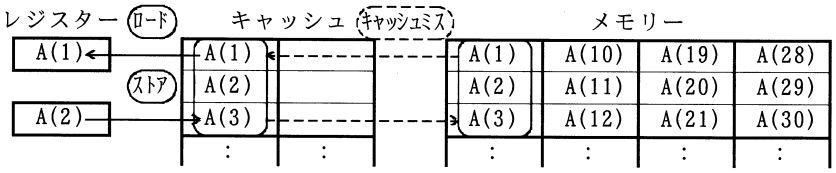


図 5-4-4

ロードとストアはどんなときに発生するのか

プログラム上でどのような場合にロードとストアが発生するかについて図 5-4-5 (1) で説明します。

- スカラー変数 D は、I=1, J=1 で初めて必要となった時点で、メモリーからキャッシュを経てレジスターにロードされます。D の値はループが反復しても不変なので、以後はレジスターに入ったままでロードとストアは発生せず、パフォーマンスには影響しません。
- 配列 C(J) は、I=1, J=1 で初めて必要となった時点でレジスターにロードされ、以後内側のループが反復している間はレジスターに入ったままでロードとストアは発生せず、パフォーマンスには影響しません。そして I=1, J=2 の時点で再びレジスターにロードされます。
- 配列 A と B は、内側のループの反復 I を添字に持っているため、内側のループが反復するたびに、その配列が式の右辺（実線の下線）にあればレジスターへのロード、左辺（二重線の下線）にあればキャッシュへのストアが行われます。

図 5-4-5 (2) のように同じ配列 A と B が DO ループ内に 2 回以上現れている場合、各配列は DO ループ内で最初に右辺に現れた時点でロードが行われ、最後に左辺に現れた時点でストアが行われます。それ以外の所（図中の点線）ではロードとストアは行われず、レジスターに入ったままになります。以下に示す外側のループのアンローリングでは、この性質を利用してロードとストアの回数を減らします。

```
    :
DO J = 1, 100
  DO I = 1, 100
    A(I,J) = A(I,J) + B(I) + C(J) + D
     ストア       ロード   ロード   1度だけロード
  ENDDO
ENDDO
    :
```

図 5-4-5 (1)

```
    :
DO J = 1, 100
  DO I = 1, 100
    A(I,J) = A(I,J) + B(I)
             ロード   ロード
    A(I,J) = A(I,J) - B(I)
     ストア
  ENDDO
ENDDO
    :
```

図 5-4-5 (2)

外側のループのアンローリング

図 5-4-6 (1) を 3 段でアンローリングした図 5-4-6 (2) を例に、修正手順を以下に示します。

- 図 5-4-6 (1) の配列 A と B のように、外側のループ反復 (本例では J) を添字に持っていない配列 (本例では A と B) がある場合、外側のループをアンローリングできる場合があります（三重以上のループでも同様)。
- 外側のループ反復①を③のように 3 反復ごとにし、②のステートメントを④〜⑥のように 3 行にします。
- ④〜⑥の配列の添字中にある J を、それぞれ J，J+1，J+2 とします。
- ①の DO ループの終値は 902 ですが、③の DO ループの終値を 902 にすると J は $J = 1, 4, 7, \ldots, 898, 901$ となり、J=901 のときに⑥の配列 C が C(I,903) となって 902 を越えてしまいます。これを防ぐため、図 5-4-6 (2) では DO ループを 2 つ作成し、最初のループでは J=1〜900 をアンローリングして計算し、2 つ目のループ⑦では残りの J=901 と 902 をアンローリングせずに計算します。

図 5-4-6 (1) (2) の下線は、実線がロード、二重線がストア、点線がロードとストアが不要な部分を示します。前述のように、図 5-4-6 (2) の点線の部分ではデータはレジスターに入ったままロードとストアが発生しないので、図 5-4-6 (1) と比べてパフォーマンスが向上します。なおプログラムが複雑な場合、コンパイラが点線部分のロードとストアが不要であることを認識できない可能性があるので、例えば図 5-4-6 (3) のように、ロードとストアが不要であることが明示的に分かるようにした方がよい場合もあります。

図 5-4-6 (1) では DO ループの反復回数が明示的に分かっていましたが、図 5-4-7 (1) のように反復回数が変数になっている DO ループを m 段 (⑧で m に値を設定します) でアンローリングする場合、図 5-4-7 (2) のようになります。最初のループはアンローリングして計算する主力のループ、2 つ目のループは残りの部分をアンローリングせずに計算するループです。⑨で最初のループの上限を一時変数 jtemp に求め、2 つ目のループは jtemp+1 から反復します。

```
   :
DO J = 1, 902                          ①
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)          ②
  ENDDO
ENDDO
   :
```

図 5-4-6 (1)

↓

```
   :
DO J = 1, 900, 3                       ③
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)          ④
    A(I) = A(I) + B(I)*C(I,J+1)        ⑤
    A(I) = A(I) + B(I)*C(I,J+2)        ⑥
  ENDDO
ENDDO
DO J = 901,902
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)          ⑦
  ENDDO
ENDDO
   :
```

図 5-4-6 (2)

＝

```
A(I) = A(I) + B(I)*(  C(I,J)
                    + C(I,J+1)
                    + C(I,J+2) )
```

図 5-4-6 (3)

```
   :
DO J = JMIN, JMAX
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)
  ENDDO
ENDDO
   :
```

図 5-4-7 (1)

↓

```
   :
m = 3                                  ⑧
jtemp = JMAX - MOD(JMAX-JMIN+1, m)     ⑨
DO J = JMIN, jtemp, m
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)      ← 1段
    A(I) = A(I) + B(I)*C(I,J+1)    ← 2段
         :                              :
    A(I) = A(I) + B(I)*C(I,J+m-1)  ← m段
  ENDDO
ENDDO
DO J = jtemp+1, JMAX
  DO I = 1, 1000
    A(I) = A(I) + B(I)*C(I,J)
  ENDDO
ENDDO
   :
```

図 5-4-7 (2)

外側のループのアンローリングの考慮点

外側のループのアンローリングは少々とっつきにくいですが、前述のように機械的に変更でき、修正は意外と簡単なので、適用できる局面が現れたら一度試してみて下さい。以下に考慮点を示します。

- 前述の図 5-4-3 (1) のようにループ反復に依存関係がある場合、図 5-4-3 (3) の方法でアンローリングすると、オリジナルと計算結果が変わってしまうので注意して下さい。
- アンローリングの段数を増やせば増やすほど、ロードとストアの回数が減るのでパフォーマンスもどんどんよくなるように思えます。しかし実際は（プログラムとマシン環境によりますが）4〜5 段程度が最もパフォーマンスがよく、段数をあまり増やすとパフォーマンスは逆に悪くなります。これはアンローリングの段数を増やすとデータレジスターを全部使いきってしまい、レジスターの内容がキャッシュに追い出されてしまうのが主な原因です。最適な段数はプログラムによって異なるので、試行錯誤で調整して下さい。なお、マシン環境によってはコンパイラが最適化によって自動的にアンローリングするため、手でアンローリングしても効果がない（または却って遅くなる）場合もあります。
- 3 重ループでは、外側の 2 つのループをアンローリングする場合もあります（例えば行列乗算）。ただし行列乗算は、数値計算ライブラリー (7 章参照) を使用するのが効率的です。

定数なのにロードが発生する DO ループ

外側のループのアンローリングとは関係ありませんが、ロードに関するチューニングを 1 つ紹介します。図 5-4-8 (1) は連立一次方程式の解法である LU 分解の前進消去のルーチンです。内側のループが反復すると左辺にある X(J+1)〜X(N) の各要素が更新されますが、これらは右辺の X(J) とは異なる要素です。従って右辺の X(J) は内側のループが反復しても値が変化しないので、本来ならば内側のループ内で初めて参照されたときにレジスターにロードされ、以降内側のループが反復している間はレジスターに入ったままになるはずです。

ところが、コンパイラは右辺の X(J) と左辺の X(I) が関係ないことを判断できないため、内側のループが反復するたびに X(J) がレジスターにロードされてしまい、パフォーマンスが低下します。このような場合、図 5-4-8 (2) に示すように、X(J) が内側の反復と関係のない定数であることが明示的に分かるようにすれば、内側のループ反復の最初に 1 回だけレジスターにロードされるようになり、パフォーマンスが向上します。

図 5-4-9 (1) に示す修正コレスキー分解の後退代入のルーチンも同様の例で、図 5-4-9 (2) の方がパフォーマンスが向上します。

```
   :
DO J = 1, N-1
  DO I = J+1, N
    X(I) = X(I) - A(I,J)*X(J)
  ENDDO
ENDDO
   :
```

図 5-4-8 (1)

→

```
   :
DO J = 1, N-1
  XJ = X(J)
  DO I = J+1, N
    X(I) = X(I) - A(I,J)*XJ
  ENDDO
ENDDO
   :
```

図 5-4-8 (2)

```
   :
DO J = N-1, 1, -1
  DO I = J+1, N
    X(J) = X(J) - A(I,J)*X(I)
  ENDDO
ENDDO
   :
```

図 5-4-9 (1)

→

```
   :
DO J = N-1, 1, -1
  XJ = X(J)
  DO I = J+1, N
    XJ = XJ - A(I,J)*X(I)
  ENDDO
  X(J) = XJ
ENDDO
   :
```

図 5-4-9 (2)

5-5　アルゴリズムの変更

ソルバーを高速なアルゴリズムに変更するのも広い意味でのチューニングと言えます。私はこの分野に関しては素人ですが、過去の経験からいくつかトピックを紹介します。

5-5-1　連立一次方程式

連立一次方程式の解法の種類

図 5-5-1 (1) は連立一次方程式 $Ax = b$ の係数行列 A を表します。空白部分の要素の値はゼロ、●, ■, ★の要素の値はゼロ以外だとします。連立一次方程式 $Ax = b$ の解法は大きく図 5-5-1 (2) のように分類されます（具体的な解法名は省略します)。このうち反復法は疎行列用の解法なので、×の部分の解法は存在しません。

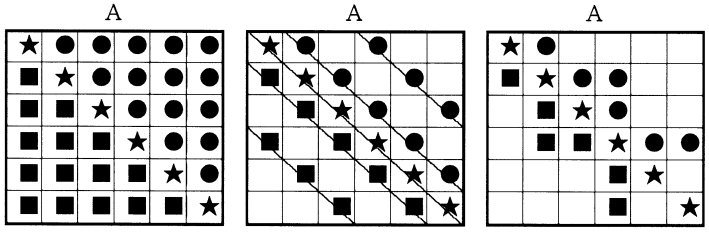


図 5-5-1 (1)

離散化の方法		境界要素法		差分法		有限要素法	
行列Aの種類		密行列		疎行列(帯行列)		不規則疎行列	
行列Aの対称性		対称	非対称	対称	非対称	対称	非対称
解法	直接法	○	○	○	○	○	○
	反復法	×		○	○	○	○

図 5-5-1 (2)

一般的な考慮点

- パフォーマンスの観点から、$Ax = b$ の行列 A が疎行列の場合は（密行列でなく）疎行列用の解法を、対称行列の場合は（非対称行列でなく）対称行列用の解法を使用して下さい。
- 一般に直接法では連立一次方程式 $Ax = b$ を以下のように 2 ステップに分けて解きますが、①が計算時間の大半を占めます。$Ax = b$ を何度も解く場合、もし A の値が不変ならば、①を 1 回だけ計算して得られた LU を保管しておき、②では保管した LU を使用すれば、①の計算は 1 回ですみます (5-2-1 節参照)。

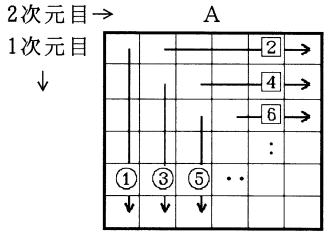
$$LU = A \quad \text{（分解）} \tag{5.1}$$

$$x = (LU)^{-1}b \quad \text{（求解）} \tag{5.2}$$

5-5-2　連立一次方程式（直接法）

クラウト法

密行列の連立一次方程式 $Ax = b$ の解法である LU 分解には、内積法、外積法、クラウト法などがあります。このうちクラウト法では、行列 A の行と列を以下の数字に示す順に交互に分解します。このうち行の分解 (②, ④, ⑥, ...) はストライドが長いため、キャッシュミスが多発し、パフォーマンスが低下します。



マルチフロンタル法

図 5-5-2 (1)〜(4) は、不規則疎行列の連立一次方程式 $Ax = b$ の係数行列 A（簡単のため対称行列とします）を表します。空白部分の要素の値はゼロ、●, ★の要素の値はゼロ以外だとします。

不規則疎行列の連立一次方程式の直接法として、一般にスカイライン法が使用されます。スカイライン法では、各列の対角要素（★）と、対角要素から一番離れたゼロ以外の要素（●）との間にある全ての要素 (図 5-5-2 (2) の着色した部分) を、途中の値がゼロであっても全てメモリー上に保持し、計算します。

スカイライン法は、図 5-5-2 (2) のようにバンド幅 (図中の ↕) が狭い場合、図 5-5-2 (1) のように行列全体を密行列として計算するよりもメモリーを節約でき、計算時間も短縮することができます。ところが図 5-5-2 (3) のようにバンド幅が広い場合、メモリーをあまり節約できず、計算時間もあまり短縮することができません。

これに対して、マルチフロンタル法という解法では、図 5-5-2 (4) に示すように、基本的にゼロ以外の要素のみをメモリー上に保持するため（これ以外に作業域が必要）、バンド幅が広い場合、スカイライン法と比べてメモリーを節約でき、計算時間も短縮することができます。

マルチフロンタル法の数値計算ライブラリーについては、参考文献 [28] を参照して下さい。

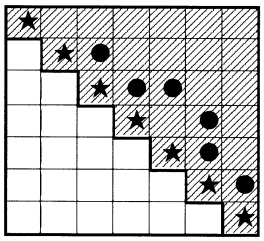


図 5-5-2 (1)　バンド幅が狭い密行列として計算

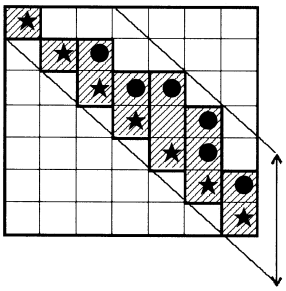


図 5-5-2 (2)　バンド幅が狭いスカイライン法

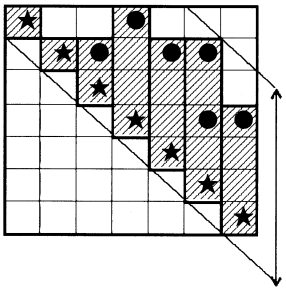


図 5-5-2 (3)　バンド幅が広いスカイライン法

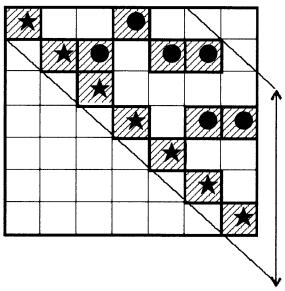


図 5-5-2 (4)　バンド幅が広いマルチフロンタル法

未知数を 1 つだけ求めたい場合

以下のように、連立一次方程式の未知数の数が例えば 1000 個で、そのうち 1 つの未知数 (例えば $\underline{x_{200}}$) のみ求めたいとします。

$$\begin{bmatrix} 5x_1 + \cdots + 8x_{200} + \cdots + 7x_{1000} = 3 \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \quad \vdots \\ 2x_1 + \cdots + 4x_{200} + \cdots + 6x_{1000} = 9 \end{bmatrix}$$

説明を簡単にするため、図 5-5-3 の「方法 1」に示すように、未知数は 3 個、そのうち求めたい未知数は $\underline{x_2}$ のみだとします。連立一次方程式 $Ax = b$ を LU 分解 (LU 分解自体の説明は省略します) で解く場合、以下の (1) (2) (3) の順に計算を行います。このうち (2) の計算は図 5-5-3「方法 1」の①，②，③の順に行い、(3) の計算は1，2，3の順に行います。

連立一次方程式 $Ax = b$ (A は係数行列、b は右辺のベクトル、x は解ベクトル)

(1) $A = LU$　分解 (L は左下三角行列、U は右上三角行列)

(2) $y = L^{-1}b$　求解（前進消去）(y は作業ベクトル)

(3) $x = U^{-1}y$　求解（後退代入）

- 求めたい未知数が $\underline{x_2}$ の場合、(1) (2) は全て計算する必要がありますが、(3) は図 5-5-3「方法 1」の2まで計算すれば $\underline{x_2}$ が求まるので、ここで計算を終了することができます。しかし冒頭の例のように、未知数が 1000 個で求めたい未知数が、$\underline{x_{200}}$ の場合、(3) では $\underline{x_{200}}$ を求める前に、$x_{1000}, x_{999}, \ldots, x_{201}$ と 800 個の不要な未知数を求める必要があり、効率的ではありません。
- 図 5-5-3 の「方法 2」では、連立一次方程式の x_1，$\underline{x_2}$，x_3 の順序を、$\underline{x_2}$ が最後になるように変えました。これによって、(3) で求める最初の未知数が、図 5-5-3「方法 2」の1に示すように $\underline{x_2}$ となり、他の未知数は求める必要がないので計算を終了します。この方法の場合、図から分かるように、行列 A 内の列の位置を変える必要があります。
- 何らかの理由で、行列 A 内の列の位置を変えたくない場合、図 5-5-3 の「方法 3」に示すように、元の連立一次方程式の最後に「$x_2 - x_4 = 0$」という式を追加します。これは「$x_2 = x_4$」と同じなので、求めた $\underline{x_4}$ が求めたい x_2 となります。(3) で求める最初の未知数が、図 5-5-3「方法 3」の1に示すように $\underline{x_4}$ となり、他の未知数は求める必要がないので計算を終了します。

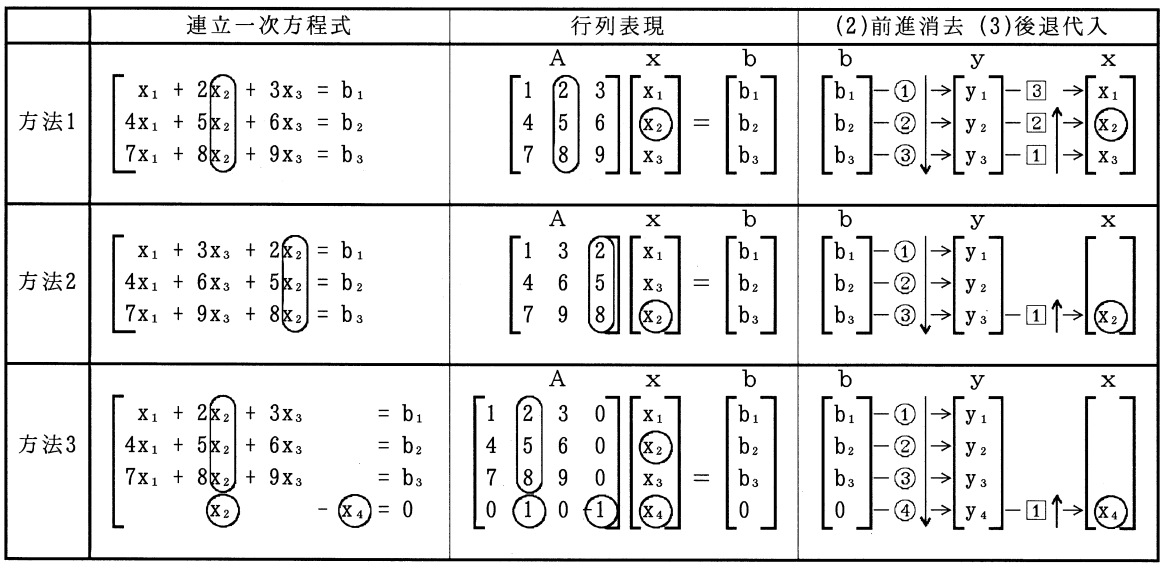


図 5-5-3

5-5-3　連立一次方程式（反復法）

超平面法とキャッシュミス

スーパーコンピュータ（ベクトル計算機）用にチューニングされたプログラムをスカラー計算機で実行すると、キャッシュミスが多発してパフォーマンスが悪くなることがあります。この理由について、帯行列の ICCG 法の例で説明します。

通常の計算方法／スカラー計算機

図 5-5-4 (1) の○は 2 次元差分法の計算領域の各要素を表します (㊢は境界条件を示します)。これらの要素に対して、スカラー計算機を使用して通常の ICCG 法で計算する場合、前進消去の部分の計算は○内の数字の順に行われます。各要素間には矢印に示すような依存関係があり、例えば②と④の値を用いて⑤を計算を行います。図から分かるように、この順序で計算した場合はストライドが 1 なので、キャッシュミスは（最低限しか）起こりません。

通常の計算順序／ベクトル計算機

次にベクトル計算機での計算方法を説明します。ベクトル計算機ではベクトル化して計算することによりパフォーマンスが向上します。ベクトル化とは、一言で言うと連続した複数の要素を一度に計算する方法であると考えて下さい。

もし図 5-5-4 (1) をベクトル化したとすると、図の長方形で囲んだ各要素 (例えば①，②，③) を一度に計算することになります。ところが矢印から分かるように、②は①の結果を使用し、③は②の結果を使用して計算を行う必要があるため、①〜③を一度に計算することはできません。従って、図 5-5-4 (1) の計算方法はベクトル化できません。

超平面法／ベクトル計算機

図 5-5-4 (2) で、例えば②，③の各値が確定すれば④，⑤，⑥の各要素を独立に計算でき、④，⑤，⑥の値が確定すれば、⑦，⑧，⑨の各要素を独立に計算することができます。従って図 5-5-4 (2) の斜め長方形を単位として、1，2，3，. . . の順に計算するのであれば、ベクトル化が可能となります。この方法を超平面法（ハイパープレーン法）と呼びます。

超平面法／スカラー計算機

図 5-5-4 (2) をスカラー計算機で計算する場合、要素を①，②，③，. . . の順に 1 つずつ計算します。ところが連続に計算する要素 (例えば④，⑤，⑥) 間のストライドはメモリー上では離れているため、この順序で計算するとキャッシュミスが多発してしまいます。

以上のことから、ICCG 法の計算をスカラー計算機で行う場合、ベクトル計算機用の図 5-5-4 (2) をそのまま使用するとキャッシュミスが多発するので、図 5-5-4 (1) のような計算順序で行って下さい。

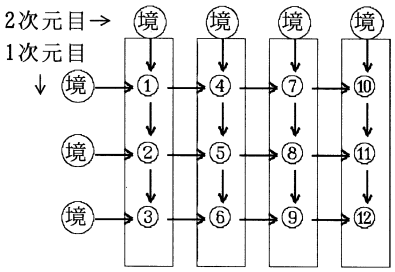


図 5-5-4 (1)　通常の ICCG 法

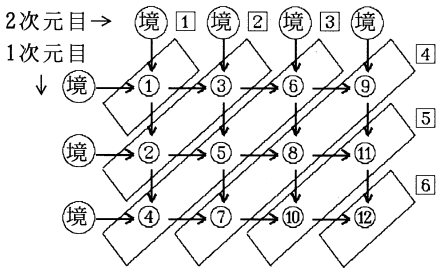


図 5-5-4 (2)　超平面法の ICCG 法

CG 法系統の解法の収束性

非対称疎行列の連立一次方程式 $Ax = b$ の反復解法には、ILUBCG、ILUCGS、ILU Bi-CGSTAB のようなアルゴリズムがありますが、それぞれ収束性が異なるため、パフォーマンスも異なります。各解法の詳細は参考文献 [18]〜[21] などを参照して下さい。

CG 法系統の解法の前処理のバグ

疎行列の連立一次方程式 $Ax = b$ の反復解法 (例えば CG 法) では、収束性を高めるために一般に前処理を行ないます。前処理の部分にバグがあったとすると、当然ながら収束性は改善されません。ところが解 x は正しい値が得られてしまうことがあるため、バグの存在が発覚しないことがあり注意が必要です。ICCG 法を例にその理由を説明します（ただし概念的な説明にとどめます)。

- 前処理なしの場合、図 5-5-5 (1) の①の形のまま連立一次方程式 $Ax = b$ の解 x を CG 法で求めます。
- 前処理付きの CG 法である ICCG 法では、まず図 5-5-5 (2) の②に示すように、行列 A に対して不完全コレスキー分解を行い、A の近似である A（実際には複数の行列）を求めます。次に①の両辺に左から A の逆行列 A^{-1} を掛けると③になります。③は①と数学的には等価なので、③を CG 法で解くと①と同じ（正しい）解 x が求まります。③の下線部は④と⑤に示すように単位行列 E に近いため、③は①よりも収束しやすくなります。
- 分解を行う②の部分にバグがあった場合、図 5-5-5 (3) の⑥に示すように、A は誤った $\mathsf{A}_{バグ}$ に分解されます。ところが①の両辺に、誤りである同じ $\mathsf{A}^{-1}_{バグ}$ を掛けるため、⑦は①と数学的に等価となり、⑦を CG 法で解くと①と同じ（正しい）解 x が求まります。ただし⑦の下線部は（一般に）単位行列に近くないので、収束性は向上しません。
- ①の両辺に左から A^{-1} を掛ける③の部分にバグがあるとします。ところが①の両辺に、誤りである同じ掛け算を行うため、⑨は①と数学的に等価になり、⑨を CG 法で解くと①と同じ（正しい）解 x が求まります。ただし⑦の下線部は（一般に）単位行列に近くないので、収束性は向上しません。

前処理付きの反復解法を使用していて何となく収束性が悪い場合、上記の部分のプログラムに誤りがないかチェックしてみて下さい。

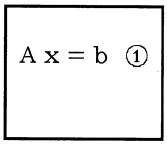


図 5-5-5 (1)　○

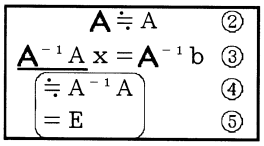


図 5-5-5 (2)　○

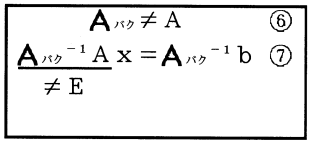


図 5-5-5 (3)　×

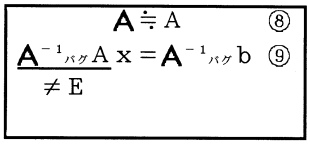


図 5-5-5 (4)　×

蛇足ですが、以前私が ICCG 法のプログラムを作成した際、図 5-5-6 (1) にすべき後退代入のルーチンを、間違えて図 5-5-6 (2) のようにしたのに正しい解 x が得られたため、不思議に思って調べたところ、上記が原因であることが分かりました。

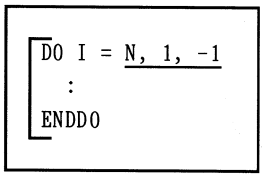


図 5-5-6 (1)　○

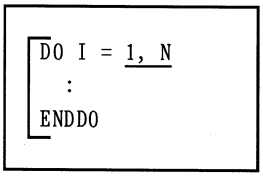


図 5-5-6 (2)　×

5-5-4　高速フーリエ変換

- 高速フーリエ変換の高速なフリーのライブラリーとして FFTW、FFTE があります (参考文献 [24] 参照)。
- 高速フーリエ変換で、要素数が 2 のベキ乗 (1024 や 2048 など) の場合、キャッシュ線の競合が発生している可能性があります。この場合、配列をダミー要素またはダミー配列でパディングすると速くなる可能性があります (詳細は 4-4 節の「■　キャッシュ線の競合」参照)。
- 高速フーリエ変換には一般に以下の 3 種類のルーチンがあり、実数を扱う (1) (2) は複素数を扱う (3) よりも計算量が半分になります。例えば入力系列が実数、出力系列が複素数の場合、入力系列の虚数部をゼロにして (3) のルーチンを使用しているプログラムをときどき見かけますが、上記のように (1) を使用する方が速くなります。

	入力系列	出力系列
(1)	実数	複素数
(2)	複素数	実数
(3)	複素数	複素数

- 図 5-5-7 の④は高速フーリエ変換の計算を行うサブルーチンで、下線の引数は規格化定数で倍精度だとします。規格化定数の具体的な値は①～③の下線部で指定しますが、①のように単精度の値を指定すると、(④では倍精度なので) 2-3 節で説明したようにサブルーチンにおかしな値が渡ってしまいます。規格化定数を倍精度で指定する必要がある場合は、②または③のように倍精度の値を指定して下さい。

```
  :
REAL*8 TEMP
CALL FFT(～,1.0,～)          ① ✖
CALL FFT(～,1.0D0,～)        ② ○
TEMP = 1.0D0                 ③
CALL FFT(～,TEMP,～)         ③ ○
  :
SUBROUTINE FFT(～,SCALE,～) ④
REAL*8 SCALE
  :
```

図 5-5-7

5-5-5　乱数

モンテカルロ法や分子動力学法などのプログラムでは、乱数を多用します。本節では、数値計算ライブラリーの乱数生成ルーチンを使用する場合の考慮点を説明します。数値計算ライブラリーの乱数生成ルーチンでは、一般に、1回のコールで、指定した個数の乱数を生成することができます。

乱数ルーチンの名前を「RANSU」だとします。図5-5-8 (1) のように1回のコールで1個だけ乱数を生成し、それを何度も行うよりも、図5-5-8 (2) のように1回のコールでまとめて乱数を生成した方が、サブルーチンコールのオーバーヘッドや乱数ルーチンの最適化の影響で、一般にパフォーマンスがよくなります。

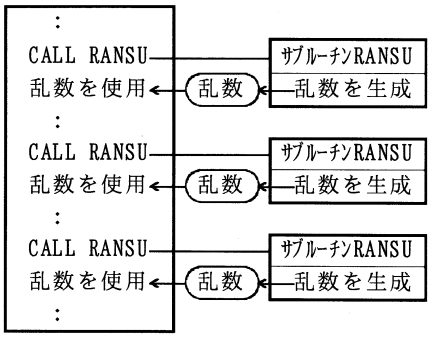


図 5-5-8 (1)

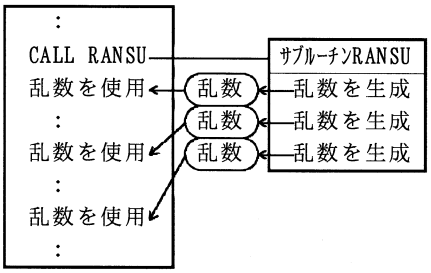


図 5-5-8 (2)

以下に具体的な方法を説明します。まず修正前のプログラムを図5-5-9 (1) に示します。①で数値計算ライブラリーの乱数生成ルーチンRANSUをコールして乱数を1個生成し、その乱数を②で使用します。

これを図5-5-8 (2) の方法で修正したプログラムを図5-5-9 (2) に示します。乱数生成ルーチンRANSUの1回のコールで10000個の乱数を生成するとします。

- 変数ICOUNTは、配列X(10000)内の乱数をどこまで使用したかを示すカウンターです。プログラムの最初に一度、③で、ICOUNTを10000（全部使用したという意味）にします。
- ④でIF文が真なので、⑤でサブルーチンRANSUをコールして乱数を10000個作成します。
- ⑥でカウンターICOUNTをゼロにリセットします。
- ⑦でカウンターICOUNTを1増やし、⑧で乱数を使用します。
- 以後、使用した乱数が10000個になるまでは、④のIF文は偽になります。使用した乱数が10000個になったら、④のIF文は真となり、再び⑤で乱数を10000個生成します。

```
  :
DO ITIME=1,100
  :
 CALL RANSU(R,1,～) 乱数を1個生成 ①
 乱数Rを使用する。                ②
  :
ENDDO
  :
```

図 5-5-9 (1)

```
 DIMENSION R(10000)
   :
 ICOUNT = 10000                          ③
   :
DO ITIME=1,100
  :
 IF (ICOUNT==10000) THEN                  ④
   CALL RANSU(R,10000,～) 乱数を10000個生成 ⑤
   ICOUNT = 0                            ⑥
 ENDIF
 ICOUNT = ICOUNT + 1                      ⑦
 乱数R(ICOUNT)を使用する。                ⑧
   :
ENDDO
   :
```

図 5-5-9 (2)

5-5-6　個別要素法／分子動力学法

粒子を扱う計算方法として、機械工学などで使用する個別要素法、計算化学などで使用する分子動力学法があります。両者は計算を行う対象は異なりますが、アルゴリズムの点では似ています。この計算方法は、5-2-5 節で紹介した例と同様に、チューニングによって計算量のオーダーを N^2 から N に減少させることができます。このチューニング方法は他の分野でも応用できる可能性があるので、本節で紹介します。

なお、この分野の先端ではもっと効率的な計算方法を使用しているかも知れませんので、文献などを参照して下さい。

固定境界（オリジナル版）

図 5-5-13 (1) に示す X 方向が 60.0、Y 方向が 70.0 の計算領域内に、直径 R=10.0 の粒子が N 個 (本例では①～⑩の 10 個) 飛んでおり、計算領域は固定境界で粒子は外に飛び出さないとします。

チューニング前のプログラムを図 5-5-11 に示します。このプログラムでは、全粒子のうち接触している粒子のペアを調べ、「接触している粒子間には力が働く」として力の計算を行います。

- 1は粒子①のループ、2は粒子ⓙのループで、粒子①が粒子ⓙと接触しているかどうかを調べます。例えば粒子①と②が接触しているかどうかを調べれば、粒子②と①を調べる必要はないので、2のループの範囲は I+1～N になっています。
- 3では、三平方の定理を用いて粒子①とⓙの中心点の間の距離 RIJ を求めます。配列 X(I), Y(I) は粒子①の中心点の X 方向と Y 方向の座標を示します
- 4で、粒子①と粒子ⓙの距離 RIJ と、粒子の直径 R（分子動力学法の場合はカットオフ距離）を比較し、粒子①とⓙが接触しているかどうかを調べます (図 5-5-10 (1) (2) 参照)。そして接触している場合は5で粒子間に働く力を計算します。
- RIJ と R を比較する代わりに、それぞれ二乗した RIJ^2 と R^2 を比較すれば、3の平方根の計算が不要になります。これを図 5-5-12 に示します。以後の説明では図 5-5-12 をオリジナル版とします。

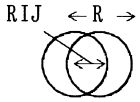


図 5-5-10 (1)　RIJ ≦ R の場合は接触

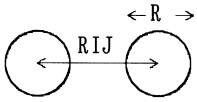


図 5-5-10 (2)　RIJ ＞ R の場合は非接触

```
      PROGRAM MAIN
      PARAMETER(N=10,R=10.0,SIZEX=60.0,SIZEY=70.0)
      DIMENSION X(N),Y(N)
        :
      DO I=1,N-1                                  1
        DO J=I+1,N                                2
          RIJ = SQRT((X(I)-X(J))**2               3
     &             + (Y(I)-Y(J))**2)
          IF (RIJ <= R) THEN                      4
            粒子ⓘⓙ間に働く力の計算                5
          ENDIF
        ENDDO
      ENDDO
      END
```

図 5-5-11

```
      PROGRAM MAIN
      PARAMETER(N=10,R=10.0,SIZEX=60.0,SIZEY=70.0)
      DIMENSION X(N),Y(N)
        :
      R2 = R*R
      DO I=1,N-1                                  1
        DO J=I+1,N                                2
          RIJ2 = (X(I)-X(J))**2                   3
     &         + (Y(I)-Y(J))**2
          IF (RIJ2 <= R2) THEN                    4
            粒子ⓘⓙ間に働く力の計算                5
          ENDIF
        ENDDO
      ENDDO
      END
```

A

図 5-5-12

5の「粒子① ⓙ間に働く力の計算」は（一般に）複雑ですが、全粒子のうち、接触している粒子の割合が極めて少ない (つまり図 5-5-12 の4が真になる確率が低い) 場合、一般に5ではなく、3と4の部分がホットスポットになります。このような場合、以下の方法で計算量を著しく減らすことができます。

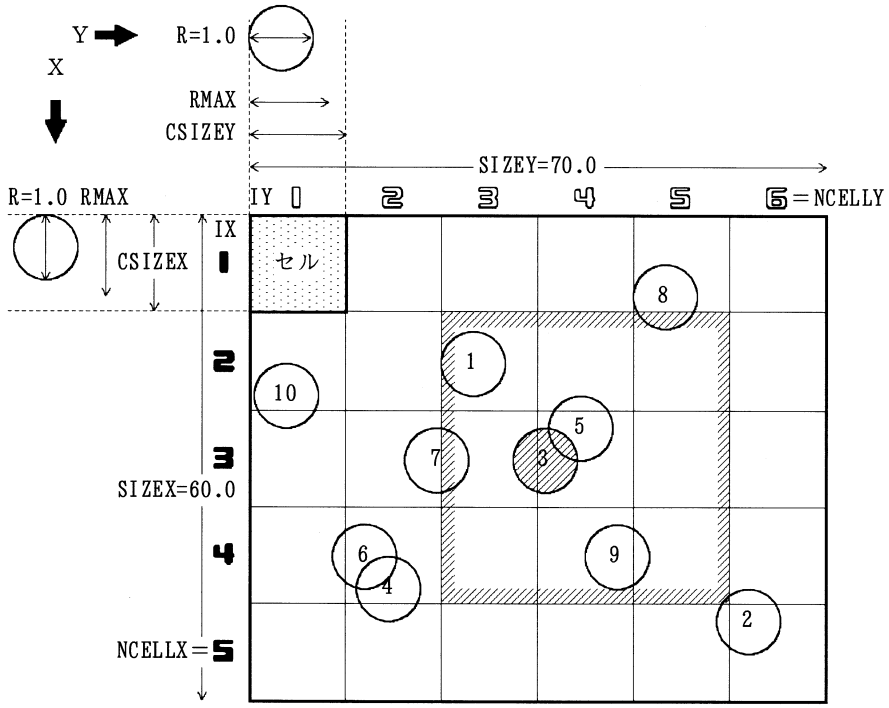


図 5-5-13 (1)

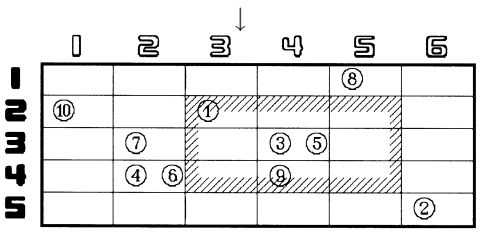


図 5-5-13 (2)

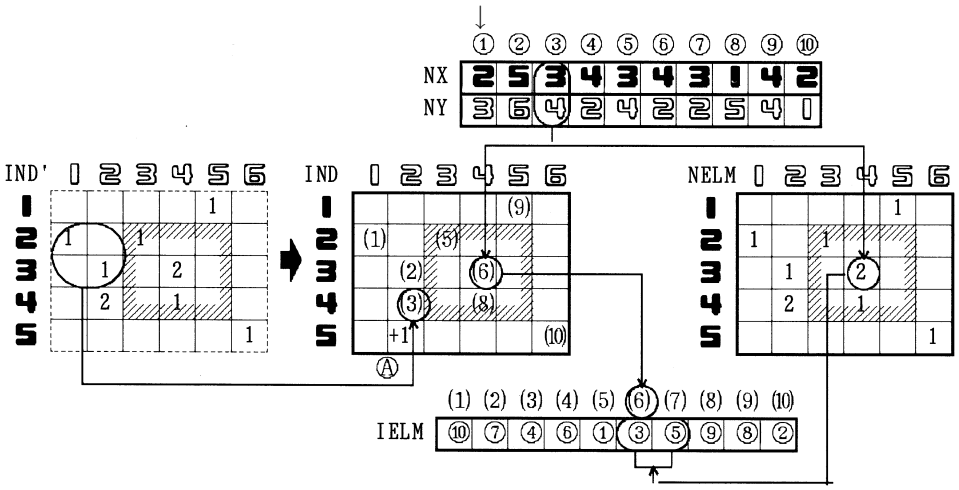


図 5-5-13 (3)

- NX(I): 粒子①が所属するセルの X 座標が入ります。
- NY(I): 粒子①が所属するセルの Y 座標が入ります。
- IELEM(I): セルごとの粒子番号が入ります。
- IND'(IX,IY): X, Y 座標が IX, IY のセルに含まれる粒子数が入り、後で IND(IX,IY) に変わります。
- IND(IX,IY): X, Y 座標が IX, IY のセルに含まれる粒子数が入っている、IELM(I) 内の先頭アドレス I が入ります。
- NELM(IX,IY): X, Y 座標が IX, IY のセルに含まれる粒子数が入ります (IND' と同じです)。

固定境界（チューニング版）

まずチューニング方法の概略を図 5-5-13 (1) で説明します。

- X 方向と Y 方向の長さがそれぞれ SIZEX，SIZEY の計算領域を、左上の着色した部分に示す大きさのセルで分割します。1 つのセルの X 方向の Y 方向は、粒子の直径 R よりも少し長くします。各セルの大きさは全て同じですが、1 つのセルの X 方向と Y 方向の長さは一般に異なります。
- 次に①〜⑩の各粒子がどのセルに所属するかを調べます。例えば粒子③は (3,4) のセルに所属します。
- 次に接触している粒子を調べます。例えば粒子③と⑦は比較的接近していますが、間に別のセル (3,3) がはさまっています。前述のようにセルの各辺は粒子の直径 R より少し長いので、③と⑦の粒子の中心間の距離 RIJ は粒子の直径 R よりも長く、③と⑦は接触していないことが保証されます (図 5-5-10 (2) 参照)。このことから、③の粒子の図 5-5-13 (1) の斜線で囲んだ範囲の外にある粒子とは接触していないことが保証されるので、斜線内に含まれる粒子 (①，⑤，⑨) のみと、接触しているかどうかのチェックをすればよいことになります。図 5-5-12 では粒子③は④〜⑩の全粒子との間で接触しているかどうかのチェックをしたので、チューニング版ではチェックする回数が大幅に減少します。

チューニング版のプログラムを図 5-5-14 に示し、以下で説明します。

- 前述のように、セルの各辺は粒子の直径 R より長くする必要があります。6で、粒子の直径 R に安全係数 (1 より少し大きな数：本例では 1.01) を掛けて RMAX とし、セルの一辺の長さは最低 RMAX 以上になるように、以下で計算します。
- 7でセルの X，Y 方向の個数 NCELLX，NCELLY を求めます (図 5-5-13 (1) 参照)。割りきれない場合は切り捨てになるので、セル数が少なくなる側、即ちセルの一辺の長さが粒子の直径 R より長くなる安全側になります。
- 図 5-5-13 (1) から分かるように、X 方向と Y 方向のセル数がどちらも 3 以下だと、接触しているかどうかをチェックする範囲が計算領域全体と等しくなってしまい、計算量は減少しないので、8でエラーとします。
- 9で 1 つのセルの X，Y 方向の長さ CSIZEX，CSIZEY を求めます (図 5-5-13 (1) 参照)。

図 5-5-13 (1) のうち、各セルとそこに所属する粒子番号の対応は図 5-5-13 (2) のようになり、これを実際にはメモリーを節約するため、図 5-5-13 (3) の 5 つの配列 (NX，NY，IND，NELM，IELM) で表現します (各配列の意味は図 5-5-13 (3) の下の説明を参照)。参考のため、粒子番号③に関連する部分を図 5-5-13 (3) の矢印で示します。

- セル数 NCELLX，NCELLY が決まったので、10で配列 NELM と IND を動的割振りで確保し、11で初期化します。なお、タイムステップ・ループで何度も接触計算を実行する場合、6〜10は一度実行するだけで構いません。
- 12で各粒子が所属するセルの X 方向と Y 方向の座標 (例えば粒子③が所属するセルの座標は (3,4)) を求め、配列 NX，NY に入れます。セルの座標が NCELLX，NCELLY を越えないように、組込関数 MIN を使用しています。なお、12の除算のパフォーマンスに問題がある場合は、9で CSIZEX，CSIZEY の逆数を求め、12の除算を乗算に変更しても構いません。
- 13で図 5-5-13 (3) の点線で囲んだ配列 IND’ を作成します（点線で囲んだのは後で中身が変わるからです)。IND’ には、各セルに含まれる全粒子数が入ります。
- 14で配列 IND’ の値を変更して配列 IND を作成します。図 5-5-13 (3) のⒶに示すように、例えば IND の (4,2) には、IND’ の (2,1) と (3,2) の値 (本例ではそれぞれ 1) の合計に 1 を加えた値が入ります。
- 15で各粒子が所属するセルの座標を調べ、17で (16は後述) その粒子の番号を配列 IELM に入れます。例えば粒子③はセル (3,4) に所属し、セル (3,4) 内の粒子は配列 IELM 内の (6) 番目 (= IND(3,4)) から順に入ります。またその時点で既に配列 IELM に入っている粒子数が配列 NELM(3,4) に入っており、この 2 つの値 (配列 IND と配列 NELM) を使用して、配列 IELM 内で粒子③を入れる位置を決定します。配列 NELM は16でカウンターとして使用され、15〜17のループが終了した時点で各セルに含まれる全粒子数 (IND’ と同一) が入ります。

以上で配列類が完成したので、これらを使用して接触している粒子を調べます。

- 18のループは粒子番号 I で反復し、まず19で粒子①が所属するセルの座標 (IX,IY) を求めます。
- 20の 2 重ループが反復すると、粒子①が所属するセルを中心とした 3×3 個のセル (例えば粒子が③の場合、図 5-5-13 (3) の斜線で囲んだ範囲のセル) が 1 つずつ順に確定します。

- 21のループが反復し、20で確定したセルに含まれる（相手の）粒子番号①が22で変数 J に入ります。
- チェックの重複を避けるため23で I<J の粒子のみに限定し、24で粒子①と粒子①が実際に接触しているかどうかチェックします (24の A は図 5-5-12 の A と同一です)。

いくつか補足します。

- チューニングの効果を検討します。粒子数を N とすると、図 5-5-12 では計算量のオーダーは N^2 になります。一方図 5-5-14 では、各ループの反復回数から判断して最も計算量の多いのは18〜24のループで、図 5-5-13 (1) の斜線で囲んだ部分に含まれる平均粒子数を n とすると、計算量のオーダーは $N \cdot n$ となります。従って図 5-5-12 と図 5-5-14 の計算量の比率は $N^2 : N \cdot n$、すなわち $N : n$ となり、全粒子のうち接触している粒子の割合が極めて少ない $N \gg n$ の場合、図 5-5-14 では計算量が著しく減少します。なお図 5-5-13 (1) から分かるように、粒子の直径 R が計算領域に比べて短いほどセルの一辺が短くなり、それによって計算領域のうち斜線で囲んだ部分の割合が少なくなるので（つまりチェックする粒子の数が減るので）、効果が高くなります。
- 例えば粒子③に対して粒子⑤，⑨が接触している場合、図 5-5-12 の5の計算は⑤，⑨の順番に行いますが、図 5-5-14 では必ずしもこの順にはなりません。5では（一般に）、接触している粒子間に働く力の合力を求めますが、上記の理由で力を加算する順序が変わった場合、力の合力の値は（一般に）桁落ちや丸め誤差などの影響で若干変わります。図 5-5-14 のプログラムの正当性を検証したい場合、図 5-5-12 と合力の値を比較するよりも、5で衝突した粒子番号①と①をファイルに書きだし、それを UNIX コマンドの「sort ファイル名 > ソート後のファイル名」でソートし、ソートしたファイル同志を diff コマンドで比較するのが良いでしょう。

```
   PROGRAM MAIN
   PARAMETER(N=10,R=10.0,SIZEX=60.0,SIZEY=70.0)
B  DIMENSION X(N),Y(N),IELM(N),NX(N),NY(N)
   INTEGER,ALLOCATABLE::NELM(:,:),IND(:,:)
     :
   R2 = R*R
C  RMAX = R*1.01                                  6
   NCELLX = SIZEX/RMAX                            7
   NCELLY = SIZEY/RMAX                            7
   IF (NCELLX<=3.AND.NCELLY<=3) THEN
     PRINT *,'===ERROR==='                        8
     STOP
   ENDIF
   CSIZEX = SIZEX/NCELLX                          9
   CSIZEY = SIZEY/NCELLY                          9
   ALLOCATE(NELM(NCELLX,NCELLY))                  10
   ALLOCATE(IND (NCELLX,NCELLY))                  10
   DO IY=1,NCELLY
     DO IX=1,NCELLX
       NELM(IX,IY) = 0                          11
       IND (IX,IY) = 0
     ENDDO
   ENDDO
   DO I=1,N
     NX(I) = MIN(INT(X(I)/CSIZEX+1),NCELLX)     12
     NY(I) = MIN(INT(Y(I)/CSIZEY+1),NCELLY)     12
     IX = NX(I)
     IY = NY(I)                                 13
     IND(IX,IY) = IND(IX,IY) + 1
D  ENDDO
```

```
   ISUM = 1
   DO IY=1,NCELLY
     DO IX=1,NCELLX
       IF (IND(IX,IY)/=0) THEN
         ITEMP = IND(IX,IY)
         IND(IX,IY) = ISUM                      14
         ISUM = ISUM + ITEMP
       ENDIF
     ENDDO
   ENDDO
   DO I=1,N
     IX = NX(I)                                 15
     IY = NY(I)                                 15
     NELM(IX,IY) = NELM(IX,IY) + 1              16
     IELM(IND(IX,IY)+NELM(IX,IY)-1) = I         17
   ENDDO
   DO I=1,N-1                                   18
     IX = NX(I)                                 19
     IY = NY(I)                                 19
     DO IIY=MAX(IY-1,1),MIN(IY+1,NCELLY)        20
       DO IIX=MAX(IX-1,1),MIN(IX+1,NCELLX)      20
         DO JJ=1,NELM(IIX,IIY)                  21
           J = IELM(IND(IIX,IIY)+JJ-1)          22
           IF (I<J) THEN                        23
             Aと同じ                            24
           ENDIF
         ENDDO
       ENDDO
     ENDDO
   ENDDO
   END
```

図 5-5-14

固定境界 (チューニング版 + セル縮小)

チューニング版の計算量をさらに減らす方法を紹介します。図 5-5-13 (1) の一部を拡大した図 5-5-15 (1) では (粒子⑤は省略)、セルの一辺 A が粒子の直径 R より長いので、前述のように粒子③と⑦の間にセルがはさまっていれば両粒子は接触していないことが保証されました。

図 5-5-15 (2) では、セルの一辺の長さは A の約 1/2 で、2 つのセルを合わせた長さ B が粒子の直径 R より長いので、粒子③と⑦の間にセルが 2 個はさまっていれば両粒子は接触していないことが保証されます。同様に図 5-5-15 (3) ではセルの一辺の長さが A の約 1/3 で、3 つのセルを合わせた長さ C が粒子の直径 R より長くなります。A，B，C がほぼ同じ長さだと仮定すると、図 5-5-15 (1) (2) (3) で粒子③の Y 方向の探索範囲は (1) $3 \times A >$ (2) $(5/2) \times B >$ (3) $(7/3) \times C$ で、(3) が最も少なくなります。また B は一般に A より少し短く（粒子の大きさにフィットしやすいので)、C は一般にさらに短くなるので、さらに差が広がります。

以上をまとめると、図 5-5-15 (2) (3) のように 1 つのセルの長さを A より短くした方が、接触しているかどうかを探索する範囲が狭くなるため、計算量が少なくなります。ただしセルの長さをあまり短くしすぎると、今度はセル情報を入れる配列を作成する部分 (図 5-5-14 の11〜17) で時間がかかってしまいます。上記チューニングを行ったプログラムを図 5-5-16 に示し、図 5-5-14 と異なる部分を下線で示します。例えば図 5-5-15 (2) のように 2 つのセルを合わせて粒子の直径 R より長くなる場合、25の IBLOCK を 2 とし、26で粒子の直径の 1/2 の長さを変数 RX に入れます。27と28では図 5-5-14 と同じ計算を行なうので、得られた NCELLX と NCELLY は粒子の直径 R の 1/2 より少し長いセルの X 方向と Y 方向の数となり、このセルを 2 つ (= IBLOCK) 合わせると粒子の直径 R より長くなります。接触しているかどうかを調べるセルの範囲は、30に示すように、調べたい粒子が所属するセルと、その前後 2 つ (= IBLOCK) のセルとなります。これに伴い29も変更します。

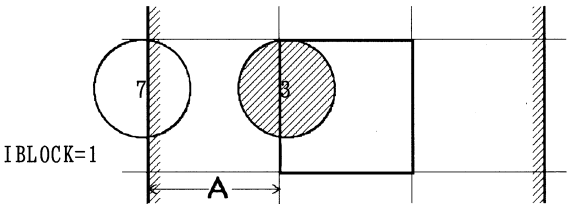


図 5-5-15 (1)

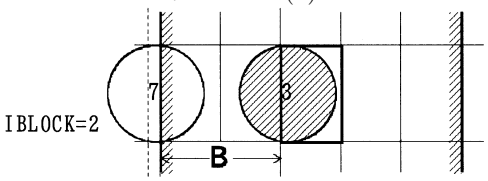


図 5-5-15 (2)

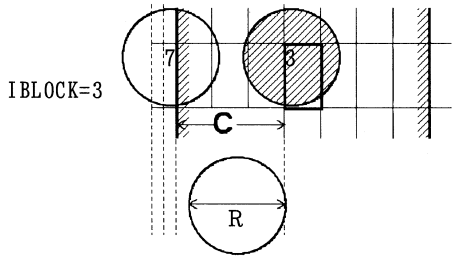


図 5-5-15 (3)

```
      PROGRAM MAIN
      Bと同じ
      R2 = R*R
      IBLOCK = 2                                    25
      RX = R/IBLOCK                                 26
      RMAX = RX*1.01                                27
      NCELLX = SIZEX/RMAX                           28
      NCELLY = SIZEY/RMAX                           28
      IF (NCELLX<=1+2*IBLOCK.AND.                   29
     &    NCELLY<=1+2*IBLOCK) THEN                  29
        PRINT *,'===ERROR==='
        STOP
      ENDIF
      Dと同じ
      DO I=1,N-1
        IX = NX(I)
        IY = NY(I)
        DO IIY=MAX(IY-IBLOCK,1),                    ↑
     &        MIN(IY+IBLOCK,NCELLY)                 30
          DO IIX=MAX(IX-IBLOCK,1),                  |
     &        MIN(IX+IBLOCK,NCELLX)                 ↓
            DO JJ=1,NELM(IIX,IIY)
              J = IELM(IND(IIX,IIY)+JJ-1)
              IF (I<J) THEN
                Aと同じ
              ENDIF
            ENDDO
          ENDDO
        ENDDO
      ENDDO
      END
```

図 5-5-16

周期境界（オリジナル版）

今までの例は粒子が計算領域の外に飛び出さない固定境界でしたが、周期境界にもセルを使用したチューニング方法を適用する事ができます。

まず周期境界プログラムのオリジナル版を説明します。図 5-5-17 の中央の着色した部分が着目する計算領域（セルではありません）で、この中に●，①，②，③の 4 つの粒子が含まれているとします。周期境界の場合、この計算領域の周囲（前後左右および斜め隣）に、同じ粒子を含む同じ計算領域が存在すると考えます。ただし処理するのはあくまで着色した計算領域内の粒子のみです。

例えば●と②の接触チェックを行う場合、●と②の距離よりも●と2との距離の方が近いときは、②は2の位置にあるとみなして接触チェックを行います。具体的には、●と②の Y 方向の距離の絶対値 (本例では 6) が計算領域の Y 方向の長さの半分 (10/2 = 5) より長い場合、②は2の位置にあると見なし、粒子間の距離の絶対値を 6 でなく 10 − 6 = 4 にして接触チェックを行います。X 方向も同様に処理を行い、結局①，②，③は1，2，3にあると見なして接触チェックを行なうことになります。

プログラムを図 5-5-18 に示します。31で粒子① ①間の X 方向と Y 方向の距離の絶対値 DISTX，DITY を求め、32で上記下線部の処理を行います。

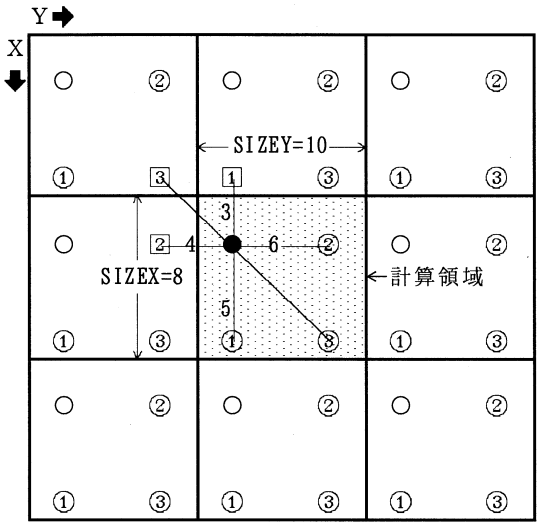


図 5-5-17

```
      PROGRAM MAIN
       PARAMETER(N=10,R=10.0,SIZEX=60.0,SIZEY=70.0)
       DIMENSION X(N),Y(N)
         :
       R2 = R*R
       DO I=1,N-1
         DO J=I+1,N
           DISTX = ABS(X(I)-X(J))                    31
           DISTY = ABS(Y(I)-Y(J))                    31
           IF (DISTX > SIZEX*0.5)                     ↑
     &       DISTX = SIZEX-DISTX                     32
           IF (DISTY > SIZEY*0.5)                     ↓
     &       DISTY = SIZEY-DISTY
           RIJ2 = DISTX**2+DISTY**2
           IF (RIJ2 <= R2) THEN
             粒子ⓘⓙ間に働く力の計算
           ENDIF
         ENDDO
       ENDDO
       END
```

図 5-5-18

周期境界（チューニング版）

周期境界のプログラムをセルを使ってチューニングする方法を説明します。図 5-5-19 (1) で外枠は対象とする計算領域、小さい四角はセルを表します。前述の固定境界の場合、左上の○のセルに含まれる粒子が接触チェックを行う相手のセルの範囲は○自身と■でした。周期境界の場合はこれらに加え、△，☆，◇の各セルも対象となりますが、これらのセルは実際には▲，★，◆です。従って、相手のセルとして例えば☆ (0,0) が選ばれた場合、★ (4,5) に置き換えればよいことになります。

図 5-5-19 (2) のように○が右下にある場合も同様に、相手のセルとして例えば☆ (5,6) が選ばれた場合は★ (1,1) に置き換えます。

上記の下線部をまとめると、セルの座標が以下の (1) の場合、「↓」の部分を (2) に置き換えればよいことが分かります。

```
                                X座標                          Y座標
(1) 置き換え前(34参照)   IIIX  0 1 2 3 4 5      IIIY  0 1 2 3 4 5 6
                              ↓       ↓              ↓         ↓
(2) 置き換え後(35参照)   IIX   4 1 2 3 4 1      IIY   5 1 2 3 4 5 1
```

図 5-5-18 を修正してセルを導入したプログラムを図 5-5-20 に示し、図 5-5-20 が図 5-5-14 と異なっている部分を下線で示します。34に示す相手のセルの X，Y 座標 IIIX，IIIY を、35で MOD を使用して上記 (1) (2) に示すように IIX，IIY に置き換えます。

周期境界の場合の注意点を述べます。図 5-5-19 (3) では計算領域の X 方向が短いため、X 方向のセル数が 2 つになっています。この場合、○のセルの粒子が接触チェックを行う相手のセルは○自身と■と◇で、このうち◇は実際には前述のように◆のセルに置き換えます。ところが◆と■は同一のセルなので、この部分のセルは 2 回（◇と■）チェックが行われてしまうことになります。この問題は X 方向または Y 方向のセル数が 2 個以下の場合に発生するので、誤作動を防ぐために33の下線部を追加しました。

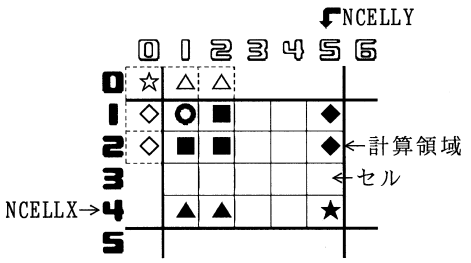


図 5-5-19 (1)

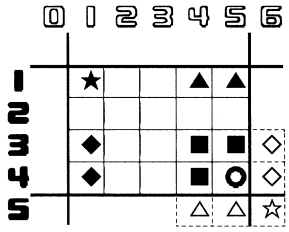


図 5-5-19 (2)

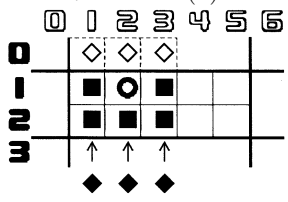


図 5-5-19 (3)

```
PROGRAM MAIN
   Bと同じ
   Cと同じ
IF ( NCELLX<=2 .OR. NCELLY<=2 .OR.                          33
&    (NCELLX<=3.AND.NCELLY<=3) ) THEN                       33
  PRINT *,'===ERROR==='
  STOP
ENDIF
   Dと同じ
DO I=1,N-1
  IX = NX(I)
  IY = NY(I)
  DO IIIY=IY-1,IY+1                                         34
    IIY = MOD(IIIY+NCELLY-1,NCELLY) + 1                     35
    DO IIIX=IX-1,IX+1                                       34
      IIX = MOD(IIIX+NCELLX-1,NCELLX) + 1                   35
      DO JJ=1,NELM(IIX,IIY)
        J = IELM(IND(IIX,IIY)+JJ-1)
        IF (I<J) THEN
          Eと同じ
        ENDIF
      ENDDO
    ENDDO
  ENDDO
ENDDO
END
```

図 5-5-20

第6章　I/Oチューニング

本章では、I/O の多いジョブで I/O 時間を減らすためのチューニング方法についてごく簡単に紹介します。

6-1　プログラムからの I/O

一般に科学技術計算のプログラムでは、経過時間の大部分を CPU 時間が占めますが、まれに I/O 時間の割合が非常に高いジョブがあります。本節では I/O に関する基本的な考慮点を説明します。

図 6-1-2 に、10000 × 10000 の配列 A をファイルに書き出した場合の、経過時間と CPU 時間（単位は秒）を示します（読み込みの場合も同様の結果となります）。なお、測定結果はマシン環境によって若干異なります。

[1] は「書式指定なし」、[2], [3] は「書式指定あり」で、[1] のほうが圧倒的に速くなります。この理由について説明します。プログラム内ではデータを 2 進数として取扱いますが、「書式指定なし」の場合は図 6-1-1 (1) に示すように 2 進データをそのままファイルに書き出します。一方「書式指定あり」の場合、図 6-1-1 (2) に示すように 2 進データを文字に変換するため CPU 時間がかかり、さらに出力ファイルの大きさも図 6-1-1 (1) より大きくなるため、物理的な I/O 時間も増大します。

従って、パフォーマンスの観点からは、人間が読む必要のある最終結果が入るファイル以外（作業ファイルなど）は書式指定なしで I/O を行って下さい。

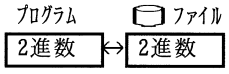


図 6-1-1 (1)　書式指定なし

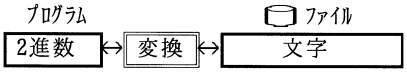


図 6-1-1 (2)　書式指定あり

[4] は [1] とほぼ同じ速度です。[4] とループの順番を逆にした [5] は、ストライドが長くキャッシュミスを起こすため (4-2 節参照)、[4] より遅くなります。[6] は圧倒的に遅くなります。

[1] ○書式指定なし
経過時間0.8 CPU時間0.1

```
DIMENSION A(M,N)
WRITE(10) A
  :
```

[4] ○書式指定なし/暗黙のDOループ
経過時間0.8 CPU時間0.1

```
DIMENSION A(M,N)
WRITE(10) ((A(I,J),I=1,M),J=1,N)
  :
```

[6] ✖書式指定なし/明示的なDOループ
経過時間20　CPU時間18

```
DIMENSION A(M,N)
DO J=1,N
  DO I=1,M
    WRITE(10) A(I,J)
  ENDDO
ENDDO
  :
```

[2] ✖デフォルトの書式指定
経過時間33　CPU時間31

```
DIMENSION A(M,N)
WRITE(10,*) A
  :
```

[5] ✖[4]とループの順番が逆
経過時間4.4 CPU時間3.8

```
DIMENSION A(M,N)
WRITE(10) ((A(I,J),J=1,N),I=1,M)
  :
```

[3] ✖書式指定あり
経過時間26　CPU時間24

```
      DIMENSION A(M,N)
      WRITE(10,999) A
999   FORMAT(E13.6)
        :
```

図 6-1-2

その他、いくつか考慮点を述べます。

- マシン環境によっては、I/O を高速化する特別な設定方法が提供されている場合もあります。
- 昔のメモリーの少ないマシン環境で開発されたプログラムは、メモリーの代わりにワークファイルを多用するため、I/O が多くなっていることがあります。現在使用しているマシンでメモリーが確保できるならば、ワークファイルでなくメモリーを使用するようにプログラムを書き直せば I/O を減らす事ができます。
- 大規模な連立一次方程式を解く場合のように、メモリーが足りずワークファイルを使用せざるを得ない場合、ワークファイルへ書き出す I/O 回数とデータ量をできるだけ減らすように、計算のアルゴリズムを工夫する必要があります。一度ワークファイルからメモリーに（一部の）データを読み込んだら、その部分の処理を出来るだけたくさん行なうようにするアルゴリズムが、参考文献 [9] などに紹介されています。

6-2　ページングによるI/O

図 6-2-1 のようにメモリーがロードモジュール（実行可能モジュール）よりも小さい場合、プログラムを実行すると動作は次のようになります（ただし動作はマシン環境によって異なります）。

- ジョブを実行すると、①に示すようにロードモジュールの一部 (本例では 1〜3) がメモリーにロードされます。
- 同時に、②に示すようにロードモジュール全体がディスク上のページング（またはスワッピング）スペースと呼ばれる領域に入ります。
- 実行中にメモリーに存在しない部分（本例では×）が必要になったとき、③に示すようにメモリー内であまり使用されていない部分 (本例では 1) がページングスペースに追い出され、④に示すように新規に必要になった 6 がページングスペースからメモリーにロードされます。これをページング（スワッピング）と呼びます。

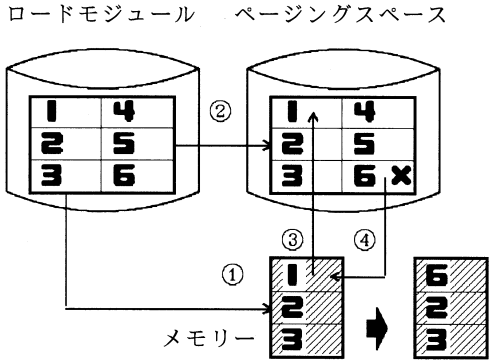


図 6-2-1

ページングはI/Oなので、CPU時間ではなくI/O時間に含まれます。ページングが発生しないジョブではCPU使用率は（他のジョブが存在しない場合）通常90数％になりますが、ページングが発生するとCPU使用率は低下してほとんどI/Oになり、パフォーマンスは著しく低下します。頻繁に使用して計算時間のかかるプログラムでページングが発生する場合は、なるべくメモリーの大きなマシンで実行するようにしてください。通常UNIXでは、実行中のジョブでページングが発生しているかどうかを調べる `vmstat` というコマンドが提供されています (付録 A 参照)。

なお、マシン環境によっては、ページングスペースよりも大きなメモリーを必要とするロードモジュールを実行すると、マシンがダウンすることがあるので注意して下さい。

第7章　数値計算ライブラリー

連立一次方程式やFFTなどのいわゆるソルバーがホットスポットになっている場合、その部分を数値計算ライブラリーのサブルーチンに置き換えるのが、（一般に）最も簡単でかつ効果の高いチューニングです。

7-1　数値計算ライブラリー

本章では数値計算ライブラリーの概要を簡単に紹介します。なお、5-5 節とも関連するので合わせて参照下さい。

種類

数値計算ライブラリーには次の種類があります。(2) と (3) は巻末の参考文献 [24] を参照して下さい。

(1) コンピュータメーカーが提供している数値計算ライブラリー
(2) ソフトウェア会社が販売している数値計算ライブラリー (NAG，IMSL など)
(3) フリーの数値計算ライブラリー (LAPACK，FFTW，FFTE など)

利点

数値計算ライブラリーには、一般に以下の利点があります（ただしライブラリーによって異なります)。

- 特に上記 (1) の数値計算ライブラリーは、そのメーカーのマシン性能を引き出すようにチューニングされているので、Fortran でコーディングした場合と比較して（通常）高速なパフォーマンスを得ることができます。
- ソルバーが複雑なアルゴリズムの場合、その部分のプログラムを 1 から作成すると、開発、テストおよびチューニングのワークロードがかかりますが、数値計算ライブラリーを使用すればこれらのワークロードをほぼゼロにすることができ、一般に品質も保証されています。
- 数値計算ライブラリーを使用すると、互換性の問題が生じるため（そのプログラムを異なるメーカーのマシンで実行できなくなる)、使用したくないという人もいますが、各数値計算ライブラリーで指定する引数はほぼ同じで（一般に）置き換えは容易なので、パフォーマンスが飛躍的に向上するのであれば数値計算ライブラリーを使用することをお勧めします。

考慮点

数値計算ライブラリーを使用する際の一般的な考慮点を説明します（ただしライブラリーによって異なります）。

- Fortran で作成したプログラムより必ず速くなるわけではなく、パフォーマンスが変わらないか却って遅くなる場合もあります。また同じサブルーチンでも、処理するデータ量によってパフォーマンスは左右されます。
- 一般に、部品のようなサブルーチンよりも、まとまった機能をもったサブルーチン（例えば行列乗算、連立一次方程式、固有値、高速フーリエ変換、乱数、ソートなど）の方が効果が出ます。
- 数値計算ライブラリーを使用する場合でも、配列のキャッシュミスに関して以下の注意が必要です。
  - ストライドがなるべく 1 になるようにデータを配置する (4-2 節参照)。
  - (特に FFT で) 配列の 1 次元目の大きさを 2 のベキ乗かそれに近い値にしない (4-4 節参照)。
- (例えば乱数生成ルーチンで) 1 回のコールで少量の演算を行いコール回数を多くするよりも、1 回のコールで大量の演算を行ないコール回数を少なくした方が、一般にパフォーマンスは向上します (5-5-5 節参照)。
- 引数に値を直接指定する場合、引数が倍精度ならば、『`CALL xxx(〜,1.0D0,〜)`』のように『`D0`』を付けて下さい (5-5-4 節参照)。例えば高速フーリエ変換の規格化定数や、数値計算ライブラリー `BLAS` に含まれる行列乗算 `DGEMM` の係数 ($C = \beta C + \alpha AB$ の α と β) などがこれに該当します。
- 一般に数値計算ライブラリーでは、配列と行列は必ずしも一致している必要はありません。つまり、図 7-1-1 のように、2 次元配列の一部分を行列（斜線で着色した部分）として指定をすることができます。
  サブルーチンコールの指定例と引数の意味を以下に示します。
  `CALL xxx(3,4,A(2,2),5,...)`
  - `3`, `4`: 行列の行数と列数。
  - `A(2,2)`: 配列内の、行列の先頭の位置。本例では★で示すように `A(2,2)` となります。行列が配列の先頭から始まっている場合は、`A(1,1)` を「`A`」と略記できます。
  - `5`: 配列の 1 次元目の大きさ。これを整合寸法と呼ぶこともあります。

整合寸法を指定する理由ですが、2 次元配列の場合、図 7-1-2 に示すようにデータはメモリー上で 1 次元目が先に動く順番 (Fortran の場合) に並んでいます (4-2 節参照)。行列の行数と列数を指定するだけでは★と●の距離が分からないので、引数で整合寸法を指定します。

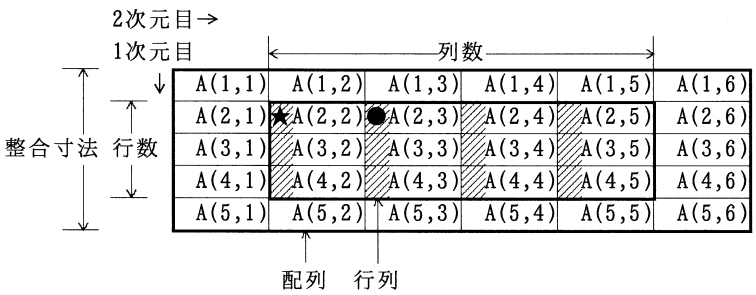


図 7-1-1　配列と行列の図

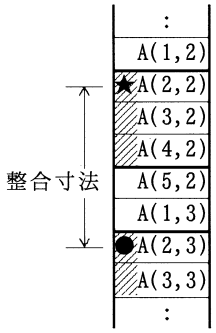


図 7-1-2　メモリー上の配置

第8章　チューニングの手順

前章までは個々のチューニング手法について紹介してきました。本章ではまとめとして、実際にチューニングを行う際の手順と考慮点について説明します。ただし紹介するのは私自身が実際に行っている方法なので、必ずしもこの通りに行う必要はありません。役に立つと思う部分を適当に取捨選択して自分に合った方法で行って下さい。

8-1　ホットスポットの特定

作業に入る前に

- 滅多に使用しないプログラムや元々計算時間が短いプログラムは、当然ながらチューニングによる効果よりもチューニング作業に要する時間の方が長くなってしまうので、チューニングする意味がありません。
- チューニングを行うとプログラムが分かりにくくなるので行いたくないという人がいますが、チューニングで修正するのはあくまで限定されたホットスポットのみで、それ以外の箇所は当然ながら修正する必要はありません。またチューニングを行うことにより、却ってプログラムが分かりやすくなる場合もあります。
- プログラム自体の作成、テストが完全に終了してからチューニング作業に入って下さい。

ホットスポット特定用の入力データ

- ホットスポットを特定する際に使用する入力データは、本番で使用する典型的なデータを使用して下さい。またプログラムがいくつかの異なる機能を持つ場合、当然ながら、使用頻度が高く計算時間のかかる機能を実行する入力データを使用して下さい。
- 実行時間の長いプログラムの場合、計算時間を短くして（後述）ホットスポットを特定してもかまいませんが、その場合、本来ホットスポットではない部分（例えば初期設定を行う部分や、入力データの読み込みを行う部分）がホットスポットになることがあるので、そのような部分は無視して下さい。
- 計算時間を短くする場合、計算領域（配列）を小さくする方法とタイムステップを短くする方法がありますが、前者は本来発生するはずのキャッシュミスがなくなるなど、ホットスポットの分布が変化してしまう可能性があるので、後者の方法をとって下さい。ただし、まれにタイムステップが小さい時点と大きくなった時点で動作が異なる（つまりホットスポットの分布が変わる）プログラムがあり、その場合はタイムステップが大きくなる時点まで測定する必要があります。

ホットスポットの特定

- 3-3 節で説明した方法でホットスポットを特定し、報告書を作成します。なお、ホットスポット特定用にプログラムをコンパイルする場合、最適化のコンパイルオプションも付けて下さい。
- ホットスポットの報告書、およびホットスポットになっているサブルーチンのソースリストを印刷します。全サブルーチンが 1 つのファイルに入っている場合、`fsplit` コマンド (付録 A 参照) で各サブルーチンを別ファイルにしてから印刷します。印刷したソースリストは以下のようにすると分かりやすくなります。

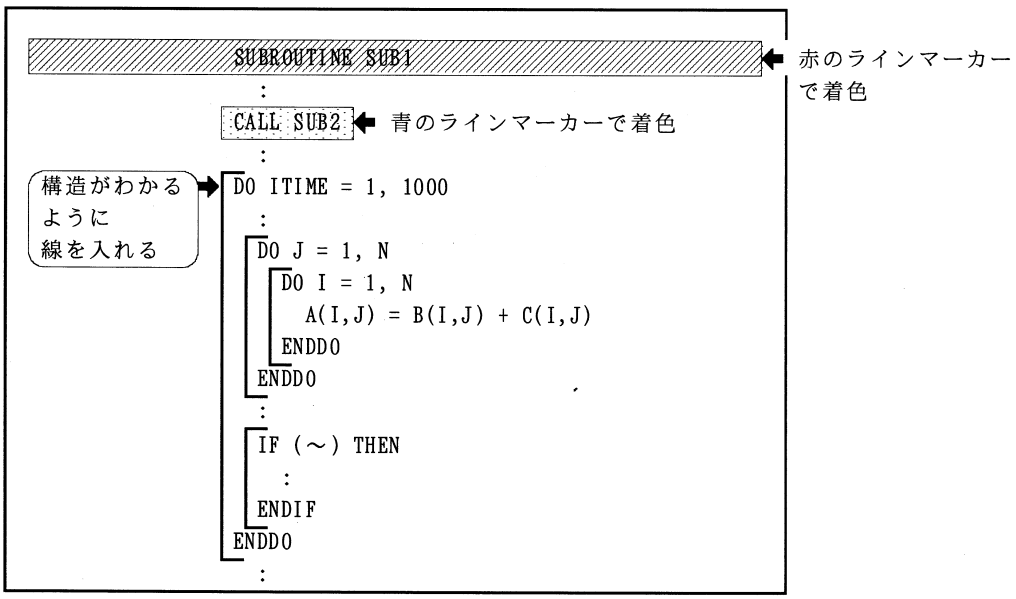


図 8-1-1

8-2　チューニング

チューニングしたプログラムのバージョン管理

チューニングする場合、オリジナル版のプログラム (以下 orig.f) に直接修正を加えずに、オリジナル版のプログラムはそのままの状態で保存しておいて下さい。また図 8-2-1 に示すように、オリジナル版のプログラムが例えば main，sub1，sub2，sub3 の 4 つのルーチンから構成されていて、サブルーチンごとに別ファイルになっている場合、以下のコマンドで 1 つのファイルに合体します（別ファイルになっていると、以下で説明するチューニング版のバージョン管理が面倒になるので)。

```
cat *.f > orig.f  ↵ ①
```

チューニングを行う場合、まず②に示すように ofig.f をコピーして tune1.f というファイルを作成し、例えば A の部分をチューニングし、後述するように計算結果とパフォーマンスの確認を行います。

さらに別の箇所をチューニングする場合、③に示すように tune1.f をコピーして tune2.f を作成し、例えば B の部分をチューニングします。このように 1 箇所チューニングするごとに別ファイルにして保存しておくと、途中で計算結果がおかしいことに気が付いた場合、どのバージョンからおかしくなっているかを調べることにより、エラーの箇所を迅速に発見することができます。

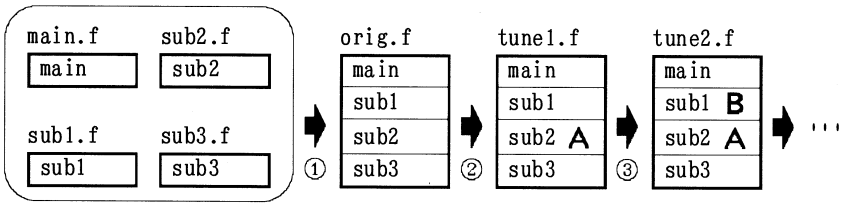


図 8-2-1

チューニング作業の効率化

チューニング作業を効率化するため、計算時間は以下のいずれかの方法で数分程度に短縮します。

- ホットスポットを特定する場合と同様に、タイムステップを短くする。
- チューニング箇所の前後に CPU 時間測定用のタイマールーチンを挿入し (3-2 節参照)、チューニング箇所が終了したら、CPU 時間と、チューニング箇所で計算した結果（全部または一部）を標準出力に書き出して STOP する。オリジナル版のプログラムでも同様の処理を行い、CPU 時間、計算結果を比較する。

チューニング方法の検討

- 最小の努力で最大の効果を挙げるため、ホットスポットの順にチューニング方法を検討します。
- 3〜7 章で多くのチューニング手法について紹介してきましたが、主な検討項目は以下の通りです。
  - まるごと数値計算ライブラリーのサブルーチンに置き換えることはできないか (→ 7 章参照)
  - 多重ループの順番が逆になっていないか (→ 4-2 節参照)
  - 除算や組込関数を削減できないか (→ 5-1 節参照)
  - ムダな計算を行っている箇所はないか (→ 5-2 節参照)
- ホットスポットの部分を、「本当にその計算は必要なのか？」という観点で検討すると、意外と無駄な計算を行っている部分を発見できることがあります。
- ホットスポットの部分を、試行錯誤で別の書き方（ロジックは同じ）に変えると、最適化などの影響で速くなることがあります。なお、講習会などで、「ある書き方と別の書き方ではどちらが速いか？」という質問を受けることがありますが、コンパイラの開発者でなければコンパイラが内部でどのような最適化を行っているのか分からないことと、マシン環境によってどちらが速いかが変わることもあるので、計算量が異なっているなど明らかな場合を除いては、「実際に測定してみないと分からない。」としか回答できません。

チューニング箇所の修正方法

チューニング版では、チューニングした箇所が一目で分かるようにしておきます。まず図 8-2-2 (1) のようなチューニング箇所を示す区切りの『ひな型』を、例えば @com というファイルに作成します。図 8-2-2 (2) のプログラムをチューニングして追加、変更、削除を行う場合、プログラム内に @com をコピーし、図 8-2-2 (3) のように修正します。これによって、どこをどのようにチューニングしたのかが一目で分かります。

Fortran で固定形式の場合、1 行あたりの最大桁数は 72 桁ですが、チューニングの最中にうっかり越えてしまうことがあるので、そのチェック用に図 8-2-2 (1) の桁数を 72 桁にしています。

図 8-2-2 (1) の先頭に含まれている「TUNE」（任意）の文字は、エディターでプログラム内のチューニング箇所を探索するときのキーワードとして使用することができます。また、あるチューニングに関係する部分を「TUNE1」、別のチューニングに関係する部分を「TUNE2」のように分類するのもよいでしょう。

@com

```
!TUNE************* 中略 ***************>>>
!----------------- 中略 -----------------
!<<<*************** 中略 ******************
```

72桁

図 8-2-2 (1)

```
      :
      DO I=1,N
C
        DO J=1,N
          A(I,J) = 1.0
        ENDDO
      ENDDO
      C = 2.0
        :
```

図 8-2-2 (2)

```
        :
!TUNE1************* 中略 ***************>>>   ┐
      B = 1.0                                  追加
!<<<*************** 中略 ******************   ┘
!TUNE1************* 中略 ***************>>>   ┐
!     DO I=1,N                                 │
C                                              変更前
!       DO J=1,N                               │
!----------------- 中略 -----------------       ┘
      DO J=1,N                                 変更後
        DO I=1,N                               │
!<<<*************** 中略 ******************   ┘
          A(I,J) = 1.0
        ENDDO
      ENDDO
!TUNE2************* 中略 ***************>>>   ┐
!     C = 2.0                                  削除
!<<<*************** 中略 ******************   ┘
        :
```

図 8-2-2 (3)

コンパイル／リンク／実行方法

図 8-2-3 に示すように、コンパイル／リンク／実行を連続して行うシェルを作成し、例えば @go という名前にします。②と③の『$1』は、このシェルを実行する際に外から与える引数です。例えば orig.f を実行する場合、④のようにすると『$1』に orig という文字が入り、②で orig.f がコンパイル／リンクされ、③で標準出力／標準エラー出力が orig.result に書き出され、同時に tee コマンドなので画面にも表示されます (3-1 節参照)。画面に表示される標準出力／標準エラー出力の量が多すぎる場合は tee コマンドを使用しない方がよいでしょう。

①で a.out を削除しているのは、②のコンパイル／リンクに失敗したときに、以前作成してディレクトリーに残っている古い a.out が③で実行されてしまうのを防ぐためです。

なおコンパイルオプションについてですが、チューニング作業中は標準的なオプションを指定し、チューニング終了後に他の（安全な）オプションをいくつか試すのがよいでしょう。2-2 節で述べたように、あまり高度な最適化を行うオプションは誤動作する可能性がありますので使用しない方が賢明です。

@go

```
rm a.out                                  ①
f77 (コンパイルオプション) $1.f           ②
(time a.out) 2>&1 | tee $1.result         ③
```

```
  :
[aoyama@cpu1]/u/aoyama: @go orig  ↵ ④ →orig.result
  :
```

図 8-2-3

結果の検証／効果の把握

- 多くの箇所を一度にチューニングすると、効果のあるチューニングと効果のない（または逆効果の）チューニングが混在してしまいます。必ず、1 箇所チューニングするごとにチューニングしたプログラムを実行し、計算結果が正しいかどうかと、パフォーマンスが向上したかどうかの両方を確認して下さい。どちらも OK ならばそのチューニングを採用し、プログラムをコピーして次の箇所のチューニングに移ります。
- 検証用のジョブの実行時間は短い（数分程度）方が効率的です。前節の「ホットスポット特定用の入力データ」で述べたように、一般にタイムステップを短くして実行時間を短縮します。
- 図 8-2-4 の①と②で、オリジナル版とチューニング版の標準出力／標準エラー出力ファイルを作成し、③で両者の計算結果とパフォーマンスを比較します。③で計算結果が比較しにくい場合は④のコマンドを使用します (付録 A 参照)。なおパフォーマンスは、チューニング箇所の前後にタイマーを挿入して測定しても構いません (3-2 節参照)。

```
[aoyama@cpu1]/u/aoyama: @go orig ↵   ①  (→ orig.resultが作成されます)
[aoyama@cpu1]/u/aoyama: @go tune1 ↵  ②  (→ tune1.resultが作成されます)
[aoyama@cpu1]/u/aoyama: diff orig.result tune1.result ↵  ③
1c1
<  Result = 123.45
---                    計算結果の比較
>  Result = 123.46
3,5c3,5
<  real  5m10.23s
<  user  5m8.52s
---                    パフォーマンスの比較
>  real  4m6.18s
>  user  4m4.79s
[aoyama@cpu1]/u/aoyama: sdiff -l -w 200 orig.result tune1.result > result ↵  ④
```

図 8-2-4

チューニング作業で役立つコマンド

- チューニングが進むにつれ、ディレクトリー内にファイルが tune1.f, tune2.f, ... と増えていき、現在作業を行っているファイルを探すのが大変になります。このような場合、付録 A (「ls コマンドの応用」) で紹介する「l（エル）」コマンド (UNIX の「ls -aFtrl」コマンドを「l」と命名) を使用すれば、図 8-2-5 の①の下線に示すように更新時刻の新しいファイルほど下に表示されるので、目的のファイルを探すのが簡単になります。
- 例えば orig.f と tune1.f のプログラムのどこが異なるのかを調べたい場合、以下の②のように sdiff コマンドを使用すると便利です (付録 A 参照)。
- チューニングした各バージョンで CPU 時間がどのように向上したかを見たい場合、以下の③のように「user」という単語をキーとして grep コマンドを使用します。

```
[aoyama@cpu1]/u/aoyama: l ↵  ①
-rw-r--r--   1 aoyama   staff       3423 Apr  1 13:17 orig.f
-rw-r--r--   1 aoyama   staff         52 Apr  1 13:21 orig.result
-rw-r--r--   1 aoyama   staff       3551 Apr  1 13:58 tune1.f
-rw-r--r--   1 aoyama   staff         52 Apr  1 14:07 tune1.result

[aoyama@cpu1]/u/aoyama: sdiff -l -w 160 orig.f tune1.f > result ↵  ②

[aoyama@cpu1]/u/aoyama: grep user *.result ↵  ③
orig.result:user  5m8.52s
tune1.result:user  4m4.79s
tune2.result:user  3m39.12s
```

図 8-2-5

付 録　便利なコマンド

本付録では、チューニングを行う際に知っていると便利な UNIX/Linux コマンドなどを紹介します。なお、コマンドとコマンドのオプションは、マシン環境によって異なる場合があります。

A　UNIX/Linux コマンド

man/info コマンド（オンラインマニュアル）

UNIX/Linux の各コマンドの機能や使い方を知りたい時、man コマンドや info コマンドを使用することができます。本節で紹介する各コマンドも、概要のみ説明しますので、詳細は man/info コマンド (または参考文献 [26] のようなコマンド集) で調べて下さい。

```
man　調べたいコマンド名 ↵
```

```
info　調べたいコマンド名 ↵
```

sdiff コマンド

図 A-1 (1) (2) のように、2 つのファイル ofig.f と tune.f の内容がわずかに異なっていて、どこが異なっているかを調べたい場合、通常図 A-1 (3) のように diff コマンドを使用します。しかし異なっている行が縦に並んで表示されるだけなので、プログラムのフローの中でどこが違っているのかを把握するのが困難です。

このような場合、図 A-1 (4) の①の sdiff コマンドを使用すると、両方のファイルの内容が横に並び、異なっている行に『|』『<』『>』が付加されます。①の「-w 160」で指定する数字（任意の値）は、表示される全桁数 (②) を示します。これを指定しないとデフォルトの桁数で表示されますが、一般に③の部分が Fortran 77 の最大桁数 (72 桁) より少なく表示され、プログラムの一部が欠けてしまうので、「-w 160」に適当な値を指定して下さい。

①では両ファイルとも全ての行が表示され、少々見にくいですが、④のように -l オプションを指定すると、左側のファイルは全ての行が表示され、右側のファイルは異なっている行のみが表示されます。一致している行では、⑤の部分は通常ブランクですが、「(」が表示されるマシン環境もあります。

sdiff コマンドは、オリジナル版とチューニング版プログラムの相違点を調べたり、計算結果の相違部分を調べたりする場合に便利です。

orig.f

```
PROGRAM MAIN
DIMENSION A(10)
INTEGER I
DO I = 1, 10
  A(I) = I
ENDDO
END
```

図 A-1 (1)

tune.f

```
PROGRAM MAIN
REAL*8 A(10)
DO I = 1, 10
  A(I) = I
ENDDO
PRINT *,A
END
```

図 A-1 (2)

```
diff orig.f tune.f ↵
2,3c2
<       DIMENSION A(10)
<       INTEGER I
---
>       REAL*8 A(10)
6a6
>       PRINT *,A
```

図 A-1 (3)

```
sdiff -w 160 orig.f tune.f > result ↵ ①
PROGRAM MAIN            PROGRAM MAIN
DIMENSION A(10)    |    REAL*8 A(10)
INTEGER I          <
DO I = 1, 10            DO I = 1, 10
  A(I) = I                A(I) = I
ENDDO                   ENDDO
                   >    PRINT *,A
END                     END
```

②

③　　③

```
sdiff -l -w 160 orig.f tune.f > result ↵ ④
PROGRAM MAIN            ⑤
DIMENSION A(10)    |    REAL*8 A(10)
INTEGER I          <
DO I = 1, 10
  A(I) = I
ENDDO
                   >    PRINT *,A
END
```

図 A-1 (4)

fsplit コマンド

本コマンドは、複数の Fortran ルーチン（メインまたはサブルーチン）が入っているファイルを、ルーチンごとに別ファイルに分割します。例えばホットスポットのサブルーチンのみを印刷したい場合、全ルーチンが 1 つのファイルに入っていると、その中からホットスポットのサブルーチンのみを抽出するのは面倒ですが、このコマンドを使用すると簡単に抽出することができます。

以下の①を実行すると、図 A-2 に示すように、ファイル sample.f の中の全ルーチンが分割され、サブルーチン名がファイル名になります。②を実行した場合はファイル sub1.f のみが作成されます。

なお、マシン環境によっては、プログラム内に Fortran 90 の命令（例えば module 文など）が含まれているとうまく分割できないことがあります。この場合、コマンドに Fortran 90 用のオプションが提供されていたり、f90split という別コマンドが提供されている場合もありますので、man コマンドで調べて下さい。

③は①の逆で、図 A-2 に示すように、ディレクトリー内の～.f という全てのファイルを（ファイル名の）アルファベット順に合体し、all.f というファイルを作成します。

```
fsplit         sample.f  ↵  ･･･ ①
fsplit -e sub1 sample.f  ↵  ･･･ ②
cat *.f > all.f  ↵          ･･･ ③
```

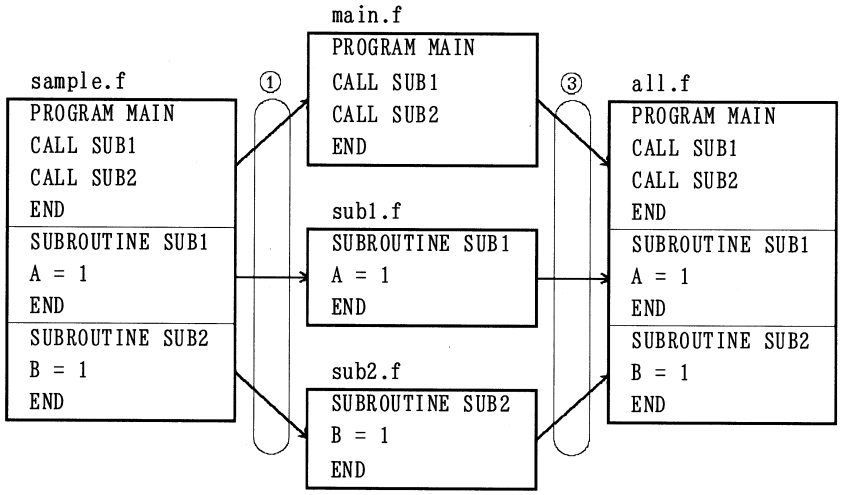


図 A-2

tr コマンド

例えば、Fortran で大文字と小文字が混在したプログラムを、どちらか一方に統一したい場合、tr コマンドを使用します。

①はファイル upper.f に入っている大文字を小文字に変換し、結果を lower.f に書き出します。②はファイル lower.f に入っている小文字を大文字に変換し、結果を upper.f に書き出します。

なお、以下の例で upper.f と lower.f を同じファイル名にすると、ファイルの中身が消えてしまうので注意して下さい。

```
tr '[A-Z]' '[a-z]' < upper.f > lower.f  ↵  ･･･ ①
tr '[a-z]' '[A-Z]' < lower.f > upper.f  ↵  ･･･ ②
```

grep/egrep コマンド

図 A-3 の①では、ファイル main.f 内の「CALL」という文字列を検索します。②では①に加え、「CALL」という文字列が含まれる行番号が表示されます。

③では現行ディレクトリー内にある『～.f』というファイル全て (本例では main.f と sub2.f) から「SUBROUTINE」という文字列と、その文字列が入っているファイル名を検索します。

④では複数の文字列「MAIN, CALL, SUBROUTINE, FUNCTION」を検索します。出力例のようにプログラムのサマリーリストを作成するときに便利です。

main.f

```
1 PROGRAM MAIN
2 CALL SUB1
3 CALL SUB2
4 END
5 SUBROUTINE SUB1
6 A = 1
7 END
```

sub2.f

```
1 SUBROUTINE SUB2
2 B = 1
3 END
```

```
grep CALL main.f ↵  ①
  CALL SUB1
  CALL SUB2

grep -n CALL main.f ↵  ②
  2:  CALL SUB1
  3:  CALL SUB2

grep SUBROUTINE *.f ↵  ③
  orig.f:   SUBROUTINE SUB1
  sub2.f:   SUBROUTINE SUB2

egrep -n "MAIN|CALL|SUBROUTINE|FUNCTION" *.f ↵  ④
  orig.f:  PROGRAM MAIN
  orig.f:  CALL SUB1
  orig.f:  CALL SUB2
  orig.f:  SUBROUTINE SUB1
  sub2.f:  SUBROUTINE SUB2
```

図 A-3

size コマンド

ロードモジュールを実行した時に必要となるメモリー量を表示します。以下の例ではロードモジュール a.out を実行するのに合計 19535 バイトのメモリーが必要になることを示します。なお、実行時に動的に取られる配列 (Fortran 90 の ALLOCATE 文や C 言語のローカル変数) の大きさは含まれません。

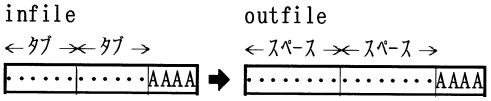
64 ビットモードで作成したロードモジュール用のオプションが用意されているマシン環境もあります。詳細は man コマンドで調べて下さい。

```
size a.out ↵
  a.out: 492 + 3199 + 385 + 660 + 13668 + 364 + 767 + 0 + 0 = 19535
```

expand コマンド

以下の命令を実行すると、ファイル infile 内に含まれる 2 つのタブ文字がスペース文字に置き換わり、ファイル outfile に出力されます。「-8」の数字（任意の値）は、1 つのタブ文字当たりに置き換わるスペース文字の文字数です。

```
expand -8 infile > outfile ↵
```



ls コマンドの応用

「ls -l」コマンドには以下のようなオプションがあります (図 A-4 参照)。

- 「ls -al」コマンドを実行すると、「.bashrc」のように先頭に「.」が付いているファイルも表示されます。
- 「ls -Fl」コマンドを実行すると、ディレクトリー名は先頭に「/』が付いて表示されます。
- 「ls -trl」コマンドを実行すると、現行ディレクトリー内のファイルが更新時刻の古い方から順番に表示されます。ディレクトリー内に多くのファイルが含まれる場合、最近更新されたファイルほど下に表示されるので、目的のファイルを探すのに便利です。

以上の機能を組み合わせると①の「ls -aFtrl」コマンドになります。これでは長くてキーインが面倒なので、例えばシェルが「bash」の場合、ホームディレクトリー内の「.bashrc」ファイルの中に、②のように例えば「l」という別名を付けておけば、「l」コマンドとして実行することができます。

```
ls -aFtrl (または「l」) ↵  ①

                                          古い
-rw-r--r--  1 aoyama  staff   323 Apr  2 11:02 | .profile
-rw-r--r--  1 aoyama  staff    31 Apr  2 11:10 | .kshrc
-rw-r--r--  1 aoyama  staff    43 Apr  2 11:37 | .exrc
drw-r--r--  2 aoyama  staff   512 Apr 10  9:21 | /data
-rw-r--r--  1 aoyama  staff  9341 Apr 10  9:22 | tune.f
                                          新しい
```

.bashrc

```
  :
alias l='ls -aFtrl'  ②
  :
```

図 A-4

容量の大きい順にファイル名を表示する方法

以下のコマンドを実行すると、現行のディレクトリーとそれ以下のディレクトリーに含まれる全てのファイルが、容量の大きい順に表示されます。

ディスク容量が少なくなり、自分のホームディレクトリー以下にある不要ファイルを削除するため、容量の大きい『ホットスポット』のファイルを探す場合に便利です。表示されたファイルの中に不要な物があれば、そのファイル名をマウスでカット・アンド・ペーストして「rm ファイル名 〓」とすれば、簡単に削除することができます。

sort コマンドに続くオプションは、「左から 7 番目のフィールドで逆順にソートする」という意味ですが、このオプションの指定方法はマシン環境によって異なりますので、「man sort 〓」で調べて下さい。

```
find ./ -ls | sort -rgk7 | more ↵   (ファイルサイズ(バイト))
                                          ↓ 大きい
30735 9164 -rw-r--r-- 1 aoyama  staff    9381677 | Jan 27 14:19 ./sample1/data1
84042 3972 -rw-r----- 1 aoyama  staff    4067328 | Feb  7 17:06 ./sample2/input.dat
 2363 3428 -rw-r--r-- 1 aoyama  staff    3506496 | Feb 19 18:40 ./sample3/source/test.f
30725 8712 -rw-r--r-- 1 aoyama  staff    2457514 | Jan 27 14:19 ./sample1/data2
                                          小さい
```

vmstat コマンド

vmstat コマンドを実行すると、システム内の各種情報が、以下の例では 2 秒ごとに 1 行ずつ表示されます。「pi」と「po」はシステム内で発生したページング（ページインとページアウト）の回数を示し、ページングが発生していない場合はゼロになります (6-2 節参照)。

「us」はシステム内の全プロセスのユーザー CPU 使用率 (%) を示します。CPU のみを使用するジョブを実行した場合はほぼ 100% となり、ページングや I/O が発生すると以下のように低下します。

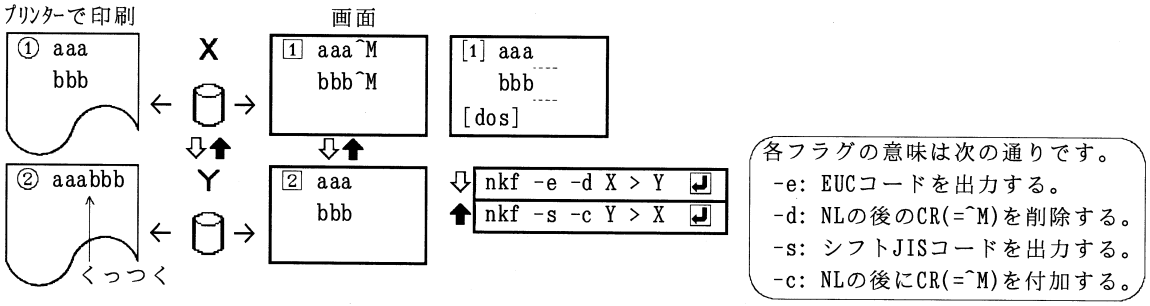
```
vmstat 2 ↵

procs     memory              page                faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b    avm    fre  re  pi  po  fr   sr  cy  in   sy  cs us sy id wa
 1  0 13052    787   0   0   0   0    0   0 112   67  20 83 17  0  0
 1  0 14118      2   0   0  92 136  276   1 465   67 150 29 17  0 53
 0  1 14400     68   0   1  77  24   28   0 414   67  42 15 12  0 74
 0  1 14400     61   2  17  24  24   49   0 267   67  58 11  9  0 48
                          以下省略
```

nkf コマンド

他のマシン環境から持ってきたファイル X を画面に表示した場合、図の[1]に示すように、各行の最後に「^M」という文字が表示されることがあります。なお、マシン環境によっては [1] のように、「^M」が点線部分に存在はしていても表示されない場合もあります (この場合、vi エディターでファイルを開くと、左下に [dos] という文字が表示されます)。

- ファイルから「^M」を取り除きたい場合は、↓に示すコマンドを実行します。
- 「^M」のついていないファイル Y をプリンターで印刷すると、②のように各行の文字がくっついてしまう場合があります。このときは、↑のコマンドでファイル X に変換してから印刷して下さい。



ps コマンド

ps コマンドはシステム内の各プロセスの状況を表示します。ps コマンドには各種のオプションがありますが、チューニングで実行ジョブの状況を調べる場合は以下の①がよいでしょう。ただしマシン環境によっては「au」の部分が他のパラメータになります。以下で「%CPU」は CPU 使用率 (%)、「%MEM」は実メモリー使用率 (%)、「STIME」はジョブの開始時刻、「TIME」は消費した CPU 時間（秒）を示します。

①のうち、たとえば USER が「aoyama」の行のみを表示したい場合は②のようにします。

```
ps au ↵ ①

USER        PID %CPU %MEM   SZ  RSS    TTY STAT    STIME  TIME COMMAND
suzuki     8711  0.0  2.0  284  556  pts/1 S    12:59:23  0:00 -ksh
aoyama     8851  0.0  1.0  112  164  hft/0 S    12:21:40  0:00 xinit
aoyama    10936 98.1 53.6   40   44  hft/0 S    12:59:38  4:17 a.out

ps au | grep aoyama ↵ ②
```

ファイルの各行に行番号を付ける方法

ファイルの各行に行番号を付加したい場合、下記のコマンドを使用します。例えば、コンパイル時の診断メッセージや最適化メッセージに行番号が含まれているので、プログラムに行番号を付けて印刷したい場合などに便利です。

```
infile                          outfile
      PROGRAM MAIN        ➡     0001:      PROGRAM MAIN
      PRINT *,'TEST'            0002:      PRINT *,'TEST'
      END                       0003:      END

  perl -ne 'printf("%04d:%s",$., $_)' infile > outfile ↵
```

シェルの環境設定

- 「bash」では、ログインするとホームディレクトリーの「.bashrc」が実行されます。この中に図 A-5 (1) の①を設定しておくと、コマンドプロンプトが A-5 (2) のように表示されます（▬はカーソルを示します）。
- 「.bashrc」に図 A-5 (1) の②を設定しておくと、コマンドのリトリーブ（以前キーインしたコマンドを、再度表示したり修正する機能）が使用可能になります。
  いくつかコマンドをキーインした後、まず「Esc キー」を押し、次に「K キー」を押すと、1 つ前にキーインしたコマンドが表示され、「J キー」を押すと 1 つ後にキーインしたコマンドが表示されます。以前実行したコマンドを修正して再利用する場合は、まず目的のコマンドを表示した後、カーソルを「L キー」で右に移動、「H キー」で左に移動させ、vi エディターと同じ要領で修正します。

.bashrc

```
    :
PS1="[`whoami`@`hostname`]"'$PWD: ';export PS1 ①
set -o vi                                      ②
    :
```

図 A-5 (1)

```
[aoyama@node1]/u/aoyama: ▮
  ↳ユーザー名 ↳ホスト名 ↳現行ディレクトリー名
```

図 A-5 (2)

vi エディター

- vi エディターで開いたファイルに含まれる、例えば「aaa」という文字列を「/aaa」コマンドで順に検索する場合、通常はファイル内の最後の「aaa」に到達すると、再びファイルの最初から検索されます (図 A-6 参照)。ファイルの最後に到達したらそこで終了したい場合（つまり図の点線の部分を行いたくない場合)、ホームディレクトリーに作成した「.exrc」というファイルに①を指定して下さい。
- 検索時に大文字と小文字を区別したくない場合 (つまり「/aaa」コマンドで、「aaa」、「AAA」、「aAa」など全ての文字列を検索したい場合)、②を指定して下さい。大文字と小文字が混在している Fortran プログラム内の文字列を検索する場合に有効です。

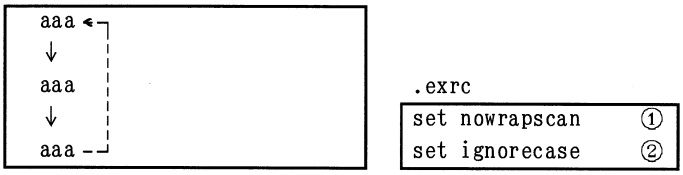


図 A-6

- vi エディターで図 A-7 (1) の各行に例えば「111」という文字を追加し、(5) にする場合を想定します（下線はカーソル位置を示します)。まず (1) の状態で「Esc キー」を押し、次に入力モードにして「111」とキーインし、再度「Esc キー」を押すと (2) になります。このとき 2 回の「Esc キー」の間にキーインしたコマンド (つまり「111」という文字の追加) が自動的に保存されます。
  保存した内容は、「.」をキーインすると再び実行することができます。例えば (3) のようにカーソルを「Enter キー」で 2 行目の先頭に移動して「.」をキーインすると、さきほど保存されたコマンドが再び実行されて (4) になります。以後 Enter キーと「.」を交互に繰り返すと (5) になります。
  ここでは文字を挿入する例を示しましたが、2 回の「Esc キー」の間にキーインする任意のコマンドを保存できるので、いろいろな応用が考えられます。

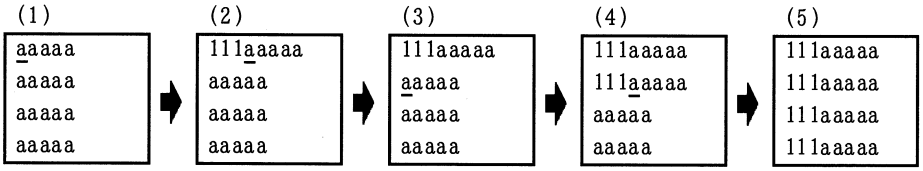


図 A-7

ftp

ファイル転送を行う ftp コマンドで、あるディレクトリー内のすべてのファイルを送信（または受信）する場合、まず①を実行し、次に②を実行します。* はワイルドカードで、例えばファイル内の xxx.f (xxx は任意の名前) というファイルのみを送信する場合は、③を実行します。

```
ftp> prompt ↵                                  ①
Interactive mode off.
ftp> mput *    ↵ (受信の場合はmget * ↵)        ②
ftp> mput *.f  ↵ (受信の場合はmget *.f ↵)      ③
```

付 録　参考文献

* がついている資料は英語で書かれています。
本書を改訂した影響で、文献の参照番号が不連続になっています。また本文から参照されていても下記に掲載されていない文献がある可能性もあります。ご了承下さい。

[0] 本書は http://accc.riken.jp/HPC/training.html に掲載されています。

[1] Fortran 90　—入門編—　小暮　仁　著（産業図書）

[2] Fortran 77 と Fortran 90　竹内則雄、平野廣和　著（森北出版）

[3] 実践 Fortran 90 プログラミング　田辺　誠、平山　博　著（共立出版）

[4] Fortran 90 入門　新井親夫（森北出版）

[5] bit 別冊　詳解 Fortran 90（共立出版）

[6] 入門 Fortran 90 実践プログラミング　東田幸樹、山本芳人、熊沢友信　著 (SOFT BANK)

[7] Fortran 90 プログラミング　冨田博之　著（培風館）

[8] コンピュータによる連立一次方程式の解法　—ベクトル計算機と並列計算機—　小国　力　訳（丸善）

[9] 岩波講座ソフトウェア科学 9　数値処理プログラミング　津田孝夫　著（岩波書店）

[14] スーパーコンピュータ　科学技術計算への適用　村田健郎、小国　力、唐木幸比古　著（丸善）
ベクトル計算機向けの数値計算法が紹介されています。

[15] *Performance of Various Computers Using Standard Linear Equations Software, Jack J. Dongarra
LINPACK のパフォーマンス資料です。

[16] RISC 超高速化プログラミング技法　寒川　光　著（共立出版）
各種チューニング技法とその原理が詳細に説明されています。

[17] ハイ・パフォーマンス・コンピューティング　RISC ワークステーションで最高のパフォーマンスを引き出すための方法　Kevin Dowd　著，久良知　真子　訳（オーム社）
RISC マシンにおける各種チューニング技法が紹介されています。

[18] 反復法の数理　藤野清次、張紹良　著（朝倉書店）

[19] 反復法 Templates　長谷川里美／長谷川秀彦／藤野清次　訳（朝倉書店）

[20] *CGSTAB: A more smoothly converging variant of CG-S, H. A. Van der Vorst and P. Sonneveld Delft University of Technology Delft, the Netherlands May 21, 1990

[21] *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing, Vol. 13 (1992), pp. 631–644, H. A. van der Vorst

[22] 行列計算ソフトウェア　WS、スーパーコン、並列計算機　小国　力　編著、村田健郎、三好俊郎、ドンガラ，J. J.、長谷川秀彦　著（丸善）

[23] 岩波講座　応用数学 [方法 2] 線形計算　森　正武、杉原正顕、室田一雄　著（岩波書店）

[24] 数値計算ライブラリー

- BLAS，LAPACK，ScaLAPACK など
  http://www.netlib.org/ で「Browse」をクリック
  以下の書籍に、LAPACK サブルーチンの使い方が説明されています。
  行列計算パッケージ LAPACK 利用の手引　小国　力　訳（丸善）
  以下に、LAPACK と ScaLAPACK に関する日本語のドキュメントがあります。
  http://phase.hpcc.jp/phase/lapack-j/
- PCP（離散化並列解法のための並列計算プラットフォーム）
  http://www.aist.go.jp/infobase/pcp/platform/index.html
- SSI（大規模シミュレーション向け基盤ソフトウェア）
  http://ssi.is.s.u-tokyo.ac.jp/
- NAG
  http://www.nag-j.co.jp/（日本ニューメリカルアルゴリズムグループ（株））
- IMSL
  http://www.vnij.co.jp/（日本ビジュアルニューメリックス（株））
- FFTW
  http://www.fftw.prg/ (高速フーリエ変換を高速に行う FFTW のサイト)
- FFTE
  http://www.ffte.jp/ (高速フーリエ変換を高速に行う FFTE のサイト)

[25] 学会／メーリングリスト／リンク集など
- PHASE (Parallel and HPC Application Software Exchange)
  http://phase.hpcc.jp/
  HPC 関係の各種リンクやメーリングリストのリンクがあります。
- SWoPP メーリングリスト
  http://www.hpcc.jp/swopp/ml_readme.html
- SofTeK HPC メーリングリスト
  http://www.softek.co.jp/
- HPC 関係のニュースを集めたサイトです。
  http://ytaka.at.infoseek.co.jp/

[26] UNIX コマンド集（他にもあると思います）
- 実用 UNIX ハンドブック　舟木　奨　著（ナツメ社）
- UNIX 活用ハンドブック　中西　隆　著（技術評論社）
- UNIX コマンド　ポケットリファレンス　中西　隆　著（技術評論社）
- 人に聞けない UNIX の使い方　アスキー書籍編集部　編（アスキー出版局）
- Linux コマンド　ポケットリファレンス　沓名亮典　平山智恵　著（技術評論社）

[27] HPC プログラミング　寒川　光、藤野清次、長島利夫、高橋大介　著（オーム社）

[28] 下記参考文献によると、疎行列の直接法（マルチフロンタル法およびその他の解法）の数値計算ライブラリーとして、下記のライブラリーが（有償または無償で）提供されています。
- 連立 1 次方程式の直接解法とソフトウェア　渡辺　善隆
  http://yebisu.cc.kyushu-u.ac.jp/~watanabe/RESERCH/MANUSCRIPT/KOHO/WSMP/index.html
- 直接解法による大規模疎行列に対する連立 1 次方程式ソルバー　渡辺　善隆
  http://yebisu.cc.kyushu-u.ac.jp/~watanabe/RESERCH/results.html
- 計算力学の常識 (第 21 話)（丸善）

BCSLIB-EXT, MA57, MUMPS, PARDISO, SPOOLES, SuperLU, SMS-MF, TAUCS, UMFPACK, WSMP, Oblio, SPRSBLKLLT, PSPASES

[29] 並列数値処理　—高速化と性能向上のために—　金田　康正　編著（コロナ社）

[30] プロセッサを支える技術　Hisa Ando　著（技術評論社）

コンピュータ設計の基礎　Hisa Ando　著（毎日コミュニケーションズ）

高性能コンピュータ技術の基礎　Hisa Ando　著（毎日コミュニケーションズ）