［大規模科学計算システム］ 高速化推進研究活動報告第 5 号より転載

スーパーコンピュータ SX-9 の高速化

スーパーコンピューティング研究部　江川隆輔　岡部公起
情報部情報基盤課　伊藤英一　小野敏　山下毅
日本電気株式会社　撫佐昭裕　神山典　小久保達信
吉村健二　遠藤清隆　小沢実希
坂本英顕　金野浩伸　坂口祐太
NEC システムテクノロジー株式会社　曽我隆

4.1 SX-9 の特徴

SX-9 は高速ベクトルプロセッサと大規模主記憶装置を有するベクトル型スーパーコンピュータである. 東北大学サイバーサイエンスセンターでは 18 ノードの SX-9 を導入しており, そのうち 16 ノードがノード間接続装置(以下, IXS)に接続されたマルチノードシステムである. 表 4.1-1 に SX-9 の主要諸元を, 表 4.1-2 に SX-9 マルチノードシステムの主要諸元を示す.

表 4.1-1 SX-9 主要諸元

項目			諸元
最大ベクトル演算性能			1.6TFLOPS
CPU 数			16
CPU	レジスタ	ベクトルレジスタ	144KB
		ベクトルマスクレジスタ	256bit×16
		スカラレジスタ	64bit×128
	データ形式	固定小数点	32/64bit
		浮動小数点	32/64/128bit
		*128bit はスカラ命令のみサポート	IEEE
		論理	64bit
	ベクトル演算パイプライン		5 種類×8 セット
	ADB		256KB
	スカラ演算パイプライン		1 セット
	キャッシュ		命令:32KB
			オペランド:32KB
主記憶装置	容量		1TB
	最大データ転送能力		4TB/s

表 4.1.2 SX-9 マルチノードシステム主要諸元

項目	諸元
ノード数	16
最大ベクトル演算性能	26.2TFLOPS
CPU 数	256
主記憶装置容量	16TB
IXS 転送性能(片方向)	128GB/s (ノード当たり) 2TB/s (システム全体)

(1) CPU の特徴

SX-9 に搭載されている CPU は, 単一チップで 102.4GFLOPS のベクトル演算性能を有するベクトル

プロセッサである. 図 4.1-1 に SX-9 のプロセッサ構成を示す. CPU 内部はベクトルユニットとスカラユニットに分かれており, ベクトルユニットは 8 つのベクトルパイプラインセットで構成されている. 各ベクトルパイプラインセットはマスク演算器, 論理演算器, 乗算器×2, 加算器×2, 除算器/平方根演算器を有している. また, SX-9 から MAX/MIN 関数がハードウェア命令化され, 最大値・最小値の計算時間が短縮されている.

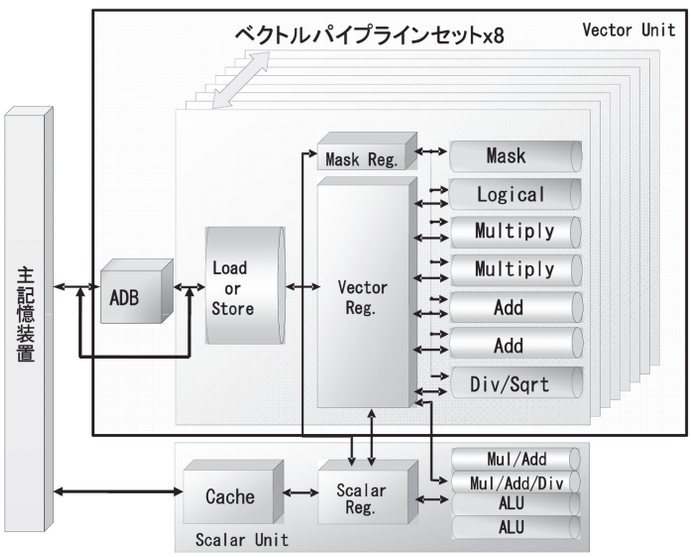

図 4.1-1 SX-9 プロセッサの構成図

(2) メモリシステムの特徴

SX-9 は, CPU と主記憶装置(MMU)間の距離が論理的に等しい共有メモリ型計算機である. 図 4.1-2 に SX-9 のメモリ構成を示す. SX-9 は CPU と主記憶装置間にネットワークルータ(以下, RTR)という内部スイッチング機構を配することで, 広いメモリバンド幅と大規模共有メモリを実現している. また, 主記憶装置はノードあたり 32,768 個のメモリバンクで構成されている. バンクのアクセスに関しては SX-7C までの SX シリーズでは一度に 1 演算要素(8Byte)を処理していたが, SX-9 から 2 演算要素(16Byte)を処理するように強化し, ロード・ストアの処理時間の短縮を図っている.

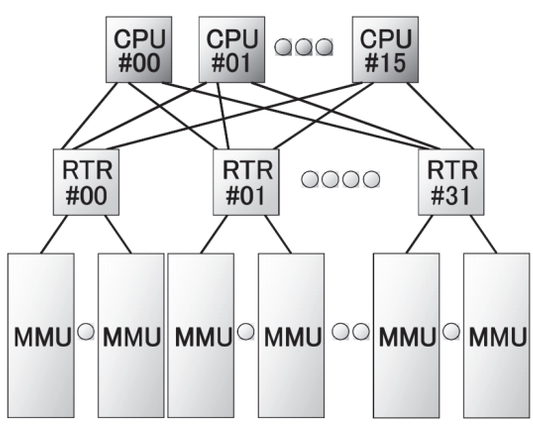

図 4.1-2 SX-9 のメモリ構成図

SX-9 では CPU－主記憶装置間に ADB（Assignable Data Buffer）と呼ばれるベクトルデータを選択的にバッファする機構を有している．図 4.1-3 に ADB の概念図を示す．ADB は CPU－主記憶装置間よりメモリバンド幅が広く，メモリレイテンシ（遅延）が短いという特徴を持っている．表 4.1-3 に SX-9 の主記憶装置と ADB のデータ供給性能を示す．データ供給性能は，演算性能あたりのデータ供給量を表す指標としてベクトル演算性能あたりのメモリバンド幅（Byte/FLOP）である．

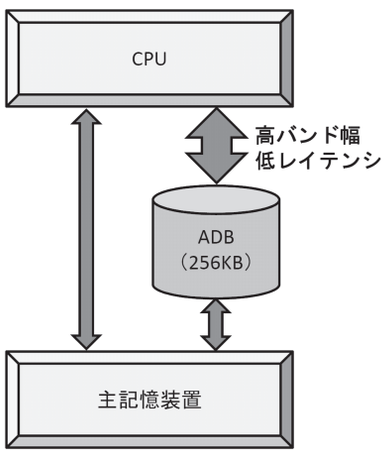

図 4.1-3 ADB の概念図

表 4.1-3 SX-9 の主記憶装置と ADB のデータ供給性能

	CPU-主記憶装置間	CPU-ADB 間
バンド幅	256.0GB/s	409.6GB/s
ベクトル演算性能あたりのバンド幅	2.5Byte/FLOP	4.0Byte/FLOP

(3) ソフトウェアの特徴

① コンパイラ

SX-9 用コンパイラとして FORTRAN90/SX と C++/SX が利用可能である．FORTRAN90/SX は ISO/IEC 1539-1:1997，ISO/IEC1539:1991 に準拠している．C++/SX は ISO/IEC 9899:1999，ISO/IEC 14882:1998 に準拠している．

FORTRAN90/SX，C++/SX は自動ベクトル化機能，自動並列化機能を有しており，ユーザは意識することなく，ベクトル化，並列化することができる．

② MPI ライブラリ

SX-9 用 MPI ライブラリは MPI-1.2 および MPI-2 に準拠した機能を提供している．

③ 科学技術計算ライブラリ

SX-9 用ライブラリとして，科学技術計算ライブラリ ASL，数値計算ライブラリ MathKeisan が利用可能である．

4.2 ベクトル処理による高速化

(1) ベクトル処理の概要

科学技術計算プログラムの多くは特定の DO ループに実行の大部分が集中し，DO ループ内の配

列データ(ベクトル)に対し繰り返し演算が行われている. この規則性に着目して高速化を図った方式がベクトル処理である. ベクトル処理はDOループによる配列データへの繰り返し演算を, ベクトルユニットを用いて一括に実行するものである. 一方, スカラ処理は繰り返し演算を逐次的に実行するものである.

(2) ベクトル化率

ベクトル型スーパーコンピュータではプログラム中のベクトル処理可能な部分を高速に実行している. 図 4.2-1 に同一のプログラムをスカラ処理する場合とベクトル処理する場合の実行時間に関する概念図を示す.

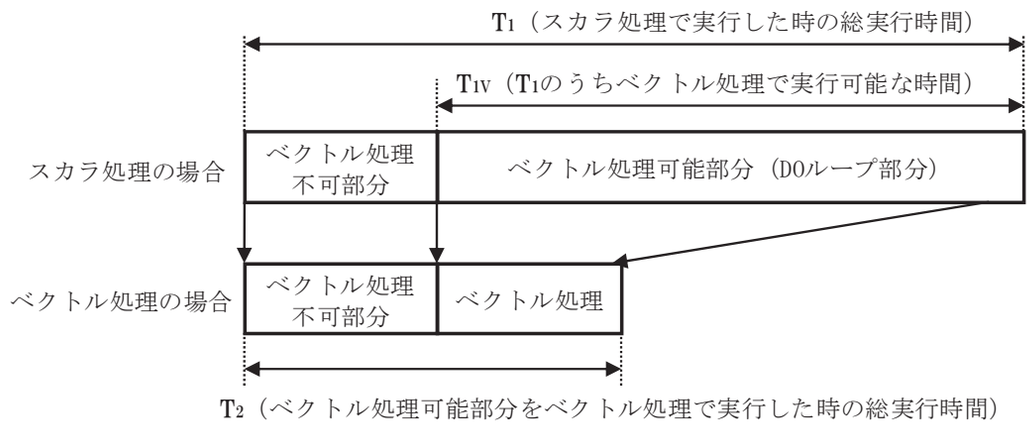

図 4.2-1　ベクトル処理による実行時間短縮のイメージ

ここで, あるプログラムをスカラ処理で実行する場合の総実行時間を T_1, そのプログラムでベクトル処理が可能な部分の実行時間を T_{1V} とすると, ベクトル化率 α は以下の式で定義される.

$$\text{ベクトル化率}\quad \alpha = \frac{T_{1V}}{T_1}$$

また, そのプログラムをベクトル処理する場合の性能向上比 P は, スカラ処理性能とベクトル処理性能の比を β とすると,

$$\text{性能向上比}\quad P = \frac{1}{(1-\alpha)+\frac{\alpha}{\beta}}$$

と表される.

この式から, ベクトル化率と性能向上比の関係は図 4.2-2 のようになる(アムダールの法則). この図からベクトル化率 80%程度では大きな性能向上は見られず, ベクトル化率 90%を超えたあたりから急激に性能が向上していることがわかる. ベクトル型スーパーコンピュータで高い実効性能を得るためにはベクトル化率を 100%にできる限り近付ける必要がある.

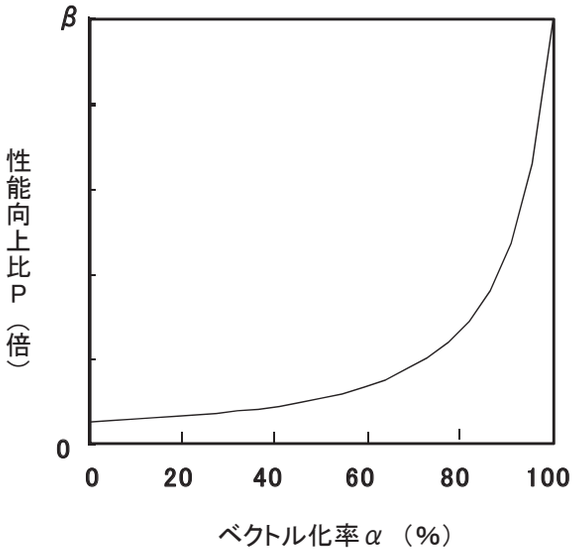

図 4.2-2 ベクトル化率と性能向上比（アムダールの法則）

ベクトル化率を求めるため，プログラムをスカラ実行する場合とベクトル実行する場合のそれぞれの実行時間が必要である．しかし，これらの実行時間は同時に得られないのでベクトル化率の算出は困難である．そのため SX-9 ではベクトル化率の代わりにベクトル演算率をベクトル化の指標としている．後述の PROGINF や FTRACE などの性能解析機能に出力される情報はベクトル演算率である．ベクトル演算率は，プログラムで処理される全演算要素数に対するベクトル演算命令で処理される演算要素数の割合で算出し，ほぼベクトル化率とみなせる指標である．

(3) ベクトル長

ベクトル処理を効率的に行う上で重要な指標にベクトル長がある．ベクトル長はベクトル化対象の DO ループの繰り返し回数のことであり，効率良くベクトル処理を行うためには，できるだけループ長を長くする必要がある．

一般に命令を発行してから結果が返ってくるまでに遅延時間が存在する．この遅延時間を立ち上がり時間と呼ぶ．図 4.2-3 にベクトル処理における立ち上がり時間および演算時間の概念図を示す．図 4.2-4 にベクトル長と立ち上がり時間の関係を示す．立ち上がり時間はベクトル長が異なる場合でも一定である．よって総演算量が等しい場合，短ベクトル長処理を繰り返すよりも長ベクトル長処理を一括して行う方が実行時間を短くすることができる．このように高速化を図るためには，DO ループのベクトル長を十分に確保し，演算時間に対する立ち上がり時間の占める割合を低くすることが重要である．

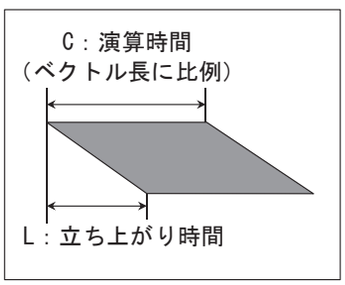

図 4.2-3　立ち上がり時間

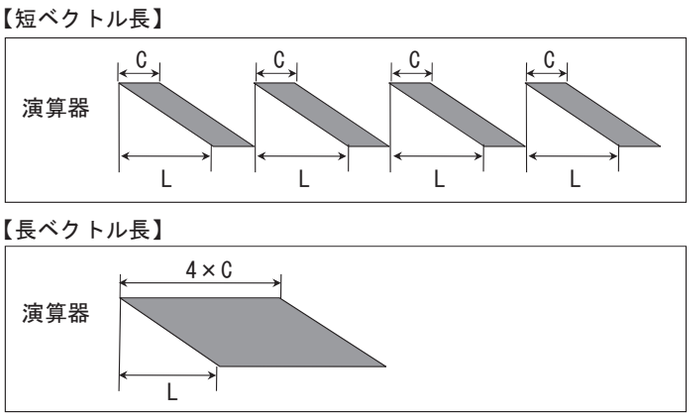

図 4.2-4　ベクトル長と立ち上がり時間

(4)　メモリアクセス

ベクトルユニットはベクトル処理によって複数の演算要素を一括で処理することが可能である．この処理を高速に行うためには，主記憶装置からベクトルユニットに対して，データを滞りなく供給することが必要である．

データ供給の遅延の要因にバンクコンフリクトがある．バンクコンフリクトは CPU ポート競合とメモリネットワーク競合の 2 つに分類される．CPU ポート競合は CPU 内における同一のポートにロード・ストアが集中した時に発生する競合を指す．メモリネットワーク競合は同一のメモリバンクへのアクセスで発生する競合や CPU と主記憶装置間の経路上で発生する競合を指す．バンクコンフリクトが発生するとデータ供給能力が低下し，高速化の妨げの要因になる．バンクコンフリクトを回避するためには，メモリアクセスの間隔が偶数となることを避け，可能ならば連続アクセスとすることや，ループのアンロールを行うなどしてメモリアクセス回数を減らすことが重要である．

また，メモリアクセスの効率化では，前述の ADB の活用があげられる．ADB を利用することで主記憶装置からのデータ供給性能を補うことや，メモリレイテンシを短くすることが可能となり性能向上を図れる場合がある．後述のチューニング例 4.3.8(4)を参考に ADB の利用を試みて頂きたい．

4.3 高速化技法

4.3.1 多重 DO ループの一重化による高速化

(1)　多重 DO ループの一重化の概要

多重 DO ループの一重化とは，図 4.3-1 のような多重構造の DO ループを，図 4.3-2 のように一重の DO ループに書き換えることをいう．図 4.3-1 では，内側の DO ループはベクトル長 10 でベクトル処理され，外側の DO ループはこのベクトル処理が 20 回実行されるので，ベクトル処理の立ち上がり時間は 20 回分必要となる(4.2.(3) 図 4.2-4 参照)．一方，図 4.3-2 では，ベクトル長 200 のベクトル処理を1回だけ実行すればよいので，立ち上がり時間は1回分だけですみ，高速化することができる．ここで，配列 a(ij,1)，b(ij,1)，c(ij,1)は一重化された DO ループで，元の配列 a(i,j)，b(i,j),c(i,j)を連続してアクセスするために書きかえたものである．

多重 DO ループの一重化は，ここで示した例のように，内側の DO ループの長さが短いときに特に効果がある．

```
!多重 DO ループ
real,dimension(10,20)::a,b,c
do j=1,20
  do i=1,10
    a(i,j)=b(i,j)+c(i,j)
  enddo
enddo
```

図 4.3-1 多重 DO ループの例

```
!図 4.3-1 の多重 DO ループの一重化
do ij = 1, 10*20
  a(ij,1) = b(ij,1) + c(ij,1)
enddo
```

図 4.3-2 多重 DO ループの一重化の例

(2) コンパイラによる DO ループの一重化

多重 DO ループは，コンパイラによって自動的に一重化される場合がある．4.3.1.(1)であげた図 4.3-1 から図 4.3-2 への変形は，コンパイラが自動的に行うものである．コンパイラが変換可能なのは，多重ループが以下の条件を満たす場合である．

① 密な多重 DO ループ(注)である
② 指標変数を含む添字式が，その配列の次元の下限から上限までの値をとる
③ 各 DO ループの指標変数は，ループ中に現れる配列要素の添字中に左から連続して同じ形式かつ同じ順序で現れている
④ DO ループ中に現れる配列要素の各添字式は異なるループの指標変数を含まない
⑤ 一重化しても定義・引用の順序関係が正しく保たれる
⑥ ループ中に現れるすべての配列はポインタでも形状引継ぎ配列でもない

(注)密な多重 DO ループとは，直接の入れ子関係をなす DO ループのうち，外側ループの DO 文と内側ループの DO 文の間および内側ループの ENDDO 文と外側ループの ENDDO 文の間に，実行文が現れないもの(およびそれと等価な構造の多重ループ)．

(3) 自動的に一重化されない多重 DO ループのチューニング

DO ループ中にポインタや形状引継ぎ配列がある場合は上述⑥の条件を満たさないため，コンパイラによる自動的なループの一重化は行われない．このような場合，ループはコンパイル時にオプション「-pvctl collapse」(注 1)を指定することにより一重化することができる．

この他，自動的に一重化されない例として，図 4.3-3 のように DO ループの範囲の外に冗長な領域を持つ配列が含まれる場合をあげられる．ループの指標変数が配列 work の下限から上限までの値をとらず，上述②の条件を満たさないため，ループの一重化は行われない．図 4.3-4 のように配列 work の冗長な領域をなくすことで，コンパイラにより自動的に一重化されるようになる．なお，図 4.3-3 と図 4.3-4 は，コンパイラによる編集リスト(注 2)の形式で記載している．

```
!オリジナル
42:              subroutine sample_f(a,b,work,p)
43:                implicit none
44:                include 'parameter.h'
45:                real*8,intent(in)  :: a(imax,jmax,kmax,3)
46:                real*8,intent(in)  :: b(imax,jmax,kmax,3)
47:                real*8,intent(in)  :: work(imax+1,jmax+1,kmax+1,3)
48:                real*8,intent(out) :: p(imax,jmax,kmax)
49:                integer :: i,j,k
50:
51: +------>       do k=1,kmax
52: |+----->         do j=1,jmax
53: ||V---->           do i=1,imax
54: |||                 p(i,j,k)=work(i,j,k,1)*(a(i,j,k,1)+b(i,j,k,1)) &
55: |||                         +work(i,j,k,2)*(a(i,j,k,2)+b(i,j,k,2)) &
56: |||                         +work(i,j,k,3)*(a(i,j,k,3)+b(i,j,k,3))
57: ||V----           enddo
58: |+-----         enddo
59: +------       enddo
```

図 4.3-3　多重 DO ループが一重化されないコードの例

```
!チューニング
42:              subroutine sample_f(a,b,work,p)
43:                implicit none
44:                include 'parameter.h'
45:                real*8,intent(in)  :: a(imax,jmax,kmax,3)
46:                real*8,intent(in)  :: b(imax,jmax,kmax,3)
47:                real*8,intent(in)  :: work(imax,jmax,kmax,3)
48:                real*8,intent(out) :: p(imax,jmax,kmax)
49:                integer :: i,j,k
50:
51: W------>       do k=1,kmax
52: |*----->         do j=1,jmax
53: ||*---->           do i=1,imax
54: |||                 p(i,j,k)=work(i,j,k,1)*(a(i,j,k,1)+b(i,j,k,1)) &
55: |||                         +work(i,j,k,2)*(a(i,j,k,2)+b(i,j,k,2)) &
56: |||                         +work(i,j,k,3)*(a(i,j,k,3)+b(i,j,k,3))
57: ||*----           enddo
58: |*-----         enddo
59: W------       enddo
```

図 4.3-4　多重 DO ループが一重化されないコードのチューニング例

(注 1)「-pvctl collapse」は，次元数が 2 以上の配列式，または多重 DO ループ中に現われる形状引き継ぎ配列，ポインタ配列の全要素がメモリ上の連続領域にあると仮定し，一重化を行うことを指示するコンパイルオプションである．

(注 2)編集リストとは，ソースコードと共に，ループと配列式のベクトル化情報，手続きのインライン展開情報，ループと配列式の並列化情報を出力する機能である．以下に編集リストに用いられる記号の意味を示す．

V：ベクトル化されたループ・配列式
W：一重化されてベクトル化されたループ・配列式
P：並列化されたループ
Y：並列化され，かつベクトル化されたループ
I：インライン展開された手続き
+：最適化されなかったループ

(4) 性能評価

(3)の図 4.3-3, 図 4.3-4 で示したチューニング前後のコードについて, 問題サイズを「imax= 50, jmax=200, kmax=1000」として, SX-9 1CPU を用いて性能評価を行う. SX-9 では, 性能情報として PROGINF(注 1)と FTRACE(注 2)がある. PROGINF からはプログラム全体の性能情報が取得でき, FTRACE からは手続(サブルーチンや関数)ごとの性能情報を取得することができる.

① PROGINF

図 4.3-5, 図 4.3-6 はチューニング前後の PROGINF である. 実効性能を比較するため, MFLOPS(1 秒間に実行された浮動小数点演算数を 100 万単位で示した値)の項目を見る. チューニング前は8,578MFLOPSであるのに対し, チューニング後には20,063MFLOPSとなっており, 実効性能が 2.33 倍向上していることが分かる. また, A.V.Length(平均ベクトル長(注 3))の項目を見ると, チューニング前は 49.9 であるのに対し, チューニング後は 252.3 となっている. チューニングにより多重 DO ループが一重化されることで, コード全体の平均ベクトル長が長くなっていることを確認できる.

```
    ******  Program Information  ******
Real Time (sec)                :            4.699699
User Time (sec)                :            4.672104
Sys  Time (sec)                :            0.023863
Vector Time (sec)              :            4.667600
Inst. Count                    :         3615133579.
V. Inst. Count                 :         1806406030.
V. Element Count               :        90320127249.
FLOP Count                     :        40080201114.
MOPS                           :        19718.922095
MFLOPS                         :         8578.619208
A. V. Length                   :           49.999904
V. Op. Ratio (%)               :           98.036742
Memory Size (MB)               :          960.031250
MIPS                           :          773.769929
I-Cache (sec)                  :            0.000384
O-Cache (sec)                  :            0.000478
Bank Conflict Time
  CPU Port Conf. (sec)         :            0.053537
  Memory Network Conf. (sec)   :            3.597606
```

図 4.3-5 チューニング前の PROGINF

```
    ******  Program Information  ******
Real Time (sec)                :            2.028015
User Time (sec)                :            1.997679
Sys  Time (sec)                :            0.025952
Vector Time (sec)              :            1.993263
Inst. Count                    :          695335689.
V. Inst. Count                 :          357973030.
V. Element Count               :        90320127249.
FLOP Count                     :        40080201166.
MOPS                           :        45381.410080
MFLOPS                         :        20063.384140
A. V. Length                   :          252.309866
V. Op. Ratio (%)               :           99.627871
Memory Size (MB)               :          960.031250
MIPS                           :          348.071782
I-Cache (sec)                  :            0.000325
O-Cache (sec)                  :            0.000416
Bank Conflict Time
  CPU Port Conf. (sec)         :            0.010343
  Memory Network Conf. (sec)   :            0.779060
```

図 4.3-6 チューニング後の PROGINF

② FTRACE

図 4.3-7, 図 4.3-8 はチューニング前後の FTRACE の情報である. ①PROGINF と同様に, 実効性能と平均ベクトル長を確認することができるが, FTRACE ではこれらの情報を手続(サブルーチンや関数)単位で取得することができる.

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
sample_f	500	4.638(99.5)	9.276	19794.6	8624.8	98.04	50.0	4.637	0.000	0.000	0.051	3.577
tp_sample	1	0.022(0.5)	22.060	14752.3	3635.5	98.33	50.0	0.022	0.000	0.000	0.000	0.015
total	501	4.660(100.0)	9.301	19770.7	8601.2	98.04	50.0	4.659	0.000	0.000	0.051	3.593

図 4.3-7 チューニング前の FTRACE 情報

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
sample_f	500	1.959(98.9)	3.918	46114.9	20420.2	99.63	256.0	1.959	0.000	0.000	0.011	0.715
tp_sample	1	0.022(1.1)	22.036	14749.9	3639.4	98.45	50.0	0.022	0.000	0.000	0.000	0.015
total	501	1.981(100.0)	3.954	45765.9	20233.5	99.63	252.3	1.980	0.000	0.000	0.011	0.730

図 4.3-8 チューニング後の FTRACE 情報

(注 1)プログラム実行解析情報:プログラム全体の性能情報が標準エラー出力ファイルに出力される. 以下に PROGINF 機能で取得可能な情報を示す.

Real Time	:経過時間(プログラム実行に要した時間)
User Time	:ユーザ時間(プログラム実行に要した CPU 時間の内, ユーザルーチンが実行に要した時間)
Sys Time	:システム時間(プログラム実行に要した CPU 時間の内, システムルーチンが実行に要した時間)
Vector Time	:ベクトル命令実行時間
Inst. Count	:全命令実行数
V. Inst. Count	:ベクトル命令実行数
V. Element Count	:ベクトル命令で実行された演算要素の個数
FLOP Count	:実行された浮動小数点演算の個数
MOPS	:1 秒間に実行された演算数を 100 万単位で示した値
MFLOPS	:1 秒間に実行された浮動小数点演算数を 100 万単位で示した値
A. V. Length	:平均ベクトル長(注 3 参照)
V. Op. Ratio	:全実行命令の個数の内, ベクトル命令の占める割合
Memory Size	:メモリ使用量
MIPS	:1 秒間に実行された命令数を 100 万単位で示した値
I-Cache	:命令キャッシュミスにより発生した合計時間
O-Cache	:オペランドキャッシュミスにより発生した合計時間
Bank Conflict Time	:バンクコンフリクト時間
CPU Port Conf.	:CPU ポート競合時間
Memory Network Conf.	:メモリネットワーク競合時間

（注 2）性能解析情報：コンパイル時にオプション「-ftrace」を指定することで，手続（サブルーチンや関数）ごとの性能解析情報を採取することができ，プログラム実行後に解析情報が標準出力ファイルに出力される．特に，チューニング対象とする手続を選択する際に活用すると良い．以下に FTRACE 機能で取得可能な情報を示す．

PROC.NAME	：手続名
FREQUENCY	：手続の呼び出し回数
EXCLUSIVE TIME	：手続の実行に要した占有の CPU 時間と，手続全体の実行に要した CPU 時間に対する比率
AVER.TIME	：手続の 1 回の実行に要した平均 CPU 時間
MOPS	：1 秒間に実行された演算数を 100 万単位で示した値
MFLOPS	：1 秒間に実行された浮動小数点演算数を 100 万単位で示した値
V.OP RATIO	：ベクトル演算率
AVER V.LEN	：平均ベクトル長（注 3 参照）
VECTOR TIME	：ベクトル命令実行時間
I-CACHE MISS	：命令キャッシュミスにより発生した合計時間
O-CACHE MISS	：オペランドキャッシュミスにより発生した合計時間
BANK CONFLICT	：バンクコンフリクト時間
CPU PORT	： CPU ポート競合時間
NETWORK	： メモリネットワーク競合時間

（注 3）SX-9 の性能情報の一つに，平均ベクトル長がある．SX-9 は一つのベクトル命令で最大 256 組の演算要素を処理することができるので，例えば DO ループの長さが 1000 の場合，256，256，256，232 という 4 つの処理単位に DO ループを分割して処理する．この場合，平均ベクトル長は 250 になる．

4.3.2 多重 DO ループのアンローリングによる高速化

(1) 多重 DO ループのアンローリングの概要

DO ループのアンローリングとは，ループ本体の演算を n 倍にし，繰り返し数を 1/n にする処理である．このとき n をアンローリングの段数という．

図 4.3-9 の外側の j のループについて，2 段のアンローリングをしたのが図 4.3-10 である．外側の j のループは増分 2 とし，内側の i のループ内に j+1 の演算を追加して，内側ループの演算を 2 倍にしている．これにより，図 4.3-10 の①の **x(i)**のストアと②の **x(i)**のロードが不要になる．つまり，図 4.3-10 は図 4.3-11 のように変形できる．よって，多重 DO ループ全体で考えると，x(i)のロード回数，ストア回数がアンローリング前の半分になる．

多重 DO ループのアンローリングでは，一般に，ベクトル化されない外側ループについてアンローリングを行うことで性能向上が期待できる．特にロード・ストア回数が減少するような場合には，大きな効果が得られることが多い．

```
!多重 DO ループ
do  j=1,100
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)
    enddo
enddo
```

図 4.3-9　多重 DO ループの例

```
!図 4.3-9 の多重 DO ループを，外側ループについて 2 段でアンローリング
do  j=1,99,2
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)        ・・・①
        x(i)=x(i)+a(i,j+1)*b(j+1)    ・・・②
  enddo
enddo
```

図 4.3-10　外側 DO ループのアンローリング後

```
!図 4.3-10 の x(i)のロード・ストアを削減した結果
do  j=1,99,2
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)+a(i,j+1)*b(j+1)
  enddo
enddo
```

図 4.3-11　x(i)のロード・ストア削減の結果

(2)　差分コードのアンローリング

図 4.3-12 は，差分コードのチューニング例である．多重 DO ループにおいてコンパイラが外側ループのアンローリングを行うように，コンパイラ指示行 OUTERUNROLL(注 1)を挿入している．

```
!チューニング
 :
47: +------>      do k=2,kmax-1
48: |             !CDIR OUTERUNROLL=4
49: |+----->        do j=2,jmax-1
50: ||V---->           do i=2,imax-1
51: |||                 s0=c(i,j,k,1)*(a(i+1,j+1,k)-a(i+1,j-1,k) &
52: |||                               -a(i-1,j+1,k)+a(i-1,j-1,k)) &
53: |||                   +c(i,j,k,2)*(a(i,j+1,k+1)-a(i,j-1,k+1) &
54: |||                               -a(i,j+1,k-1)+a(i,j-1,k-1)) &
55: |||                   +c(i,j,k,3)*(a(i+1,j,k+1)-a(i-1,j,k+1) &
56: |||                               -a(i+1,j,k-1)+a(i-1,j,k-1))
57: |||                 ss=s0*b(i,j,k)
58: |||                 p(i,j,k)=a(i,j,k)+omega*ss
59: ||V----           enddo
60: |+-----         enddo
61: +------       enddo
 :
```

図 4.3-12　差分コードのチューニング例

図 4.3-13 は図 4.3-12 のチューニング例について，アンローリング部分の変形結果を変形リスト(注 2)より抜粋したものである．

図 4.3-13 アンローリング後の変形リスト

この 4 段のアンローリングの例では，配列 a，b，c は全部で 68 個の要素がロードされるが，そのうち配列 a は 8 個の要素が 2 回参照される(注 3)．重複する部分は再度ロードされることが無いため，DO ループ内のロード回数が 68 回から 60 回に減る．同様に，8 段でアンローリングした場合は，配列 a，b，c は全部で 136 個の要素がロードされるが，そのうち配列 a は 24 個の要素が 2 回参照されるため，DO ループ内のロード回数が 136 回から 112 回に減る．

(注 1)直後の外側 DO ループに対し，強制的に n 段のアンローリングを行うことを指定する指示行である．これを指定した場合，コンパイラはループ内のデータの依存関係をチェックせずにアンローリングを行う．なお OUTERUNROLL の段数は，指定した値以下の 2 のべき乗の値となる．

(注 2)変形リスト:コンパイラが行った最適化の結果，実際に生成されたオブジェクトコードの構造をソースコードのイメージで表示したリストである．

(注 3)図 4.3-13 では，変形リストを抜き出して行番号を追記し，2 回参照される配列がわかりやすいよう，配列ごとに形式を変えて強調表記している．例えば，行番号 4 と 10 の矢印で示した a(i,j+2,k)の配列は **a(i,j+2,k)**と表記している．

(3) 性能評価

(2)の図 4.3-12，図 4.3-13 で示しているチューニング前後のコードについて，問題サイズを「imax=256, jmax=256, kmax=256」とし，SX-9 1CPU を用いて性能評価を行う．表 4.3-1 が性能測定結果である．4 段でアンローリングする場合には，チューニング前に比べて 1.16 倍，8 段でアンローリングする場合には，1.30 倍性能向上していることが分かる．

表 4.3-1 アンローリングの性能比較

	性能
チューニング前	16.29GFLOPS
アンローリング後(4 段)	18.91GFLOPS
アンローリング後(8 段)	20.21GFLOPS

(4)　アンローリング段数による性能の違い

(3)での評価結果からもわかるとおり，アンローリング段数によって性能に差が生じている．図 4.3-14 は，(2)の差分コードについて，段数を 2，4，8，16，32 と変えて性能測定している結果である．

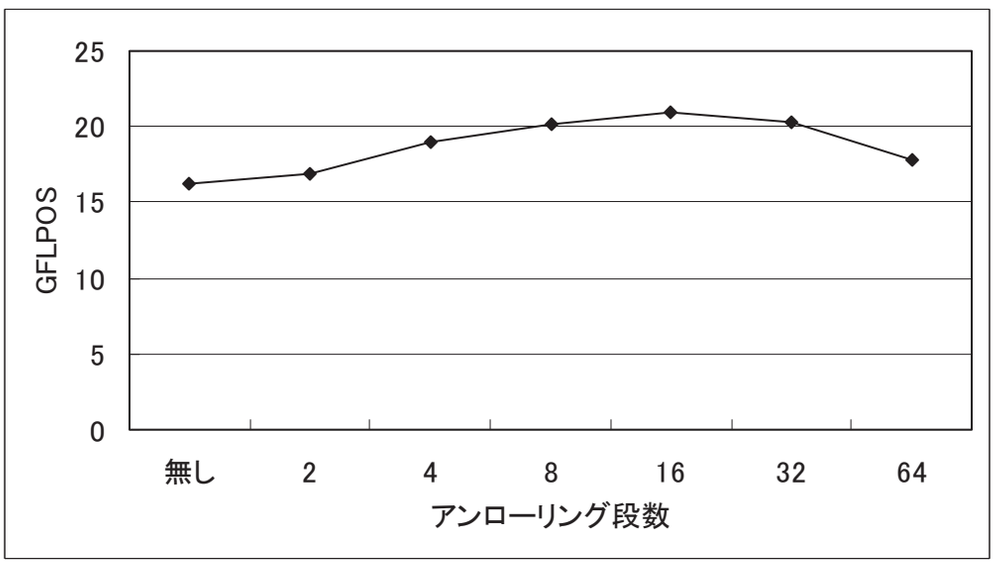

図 4.3-14　アンローリング段数による性能差

一般に，アンローリング段数を増やすと性能が向上する．しかし，段数が増えるに伴いループ中の中間変数が増加する．そのため段数を増やしすぎると途中で中間変数を保存するレジスタが足りなくなり，性能が劣化する．図 4.3-14 のとおり，(2)の差分コードの例では，16 段でアンローリングすると最も性能が高くなる．最適なアンローリング段数はコードにより異なる．

(5)　アンローリングの自動/手動に関する補足

外側の DO ループをアンローリングすることで，内側の DO ループのベクトル演算パイプラインを有効に利用できる場合や，ロード・ストア回数を減らす効果がある場合，コンパイラは自動的にアンローリングを行う．コンパイラによって自動的にアンローリングされる場合でも，デフォルトの段数（通常 4 段）以外で実行すると更に性能向上が得られる場合がある．多重ループでは，ロード・ストア回数が最も削減される次元でアンロールするのが有効である．

4.3.3　IF 文を含む DO ループのチューニング

(1)　IF 文を含む DO ループのベクトル処理の概要

図 4.3-15 のように IF 文を含む DO ループは，つぎのように処理される．

①　条件式「flag(n).eq.1」の真偽に応じて，マスクベクトルを作成

②　条件式の真偽にかかわらず，b(n)+c(n) を演算

③　マスクベクトルが真である要素のみ，b(n)+c(n)の演算結果を a(n)に代入

このように，IF 文を含まない DO ループに比べ，条件分岐のための処理が加わるので，分岐の数が多いと性能劣化の要因となる．

```
real,dimension(nsp)::a,b,c
integer::flag(nsp)

do n=1,nsp
  if(flag(n).eq.1)then
    a(n)=b(n)+c(n)
  endif
enddo
```

図 4.3-15 IF 文を含んだ DO ループの例

なお, IF 文を含んでいても, 図 4.3-16 のようにループの繰り返しの間, 条件式の結果が不変の場合, コンパイラは IF 文を DO ループの外に出してベクトル化するので, IF 文のオーバヘッドは発生しない.

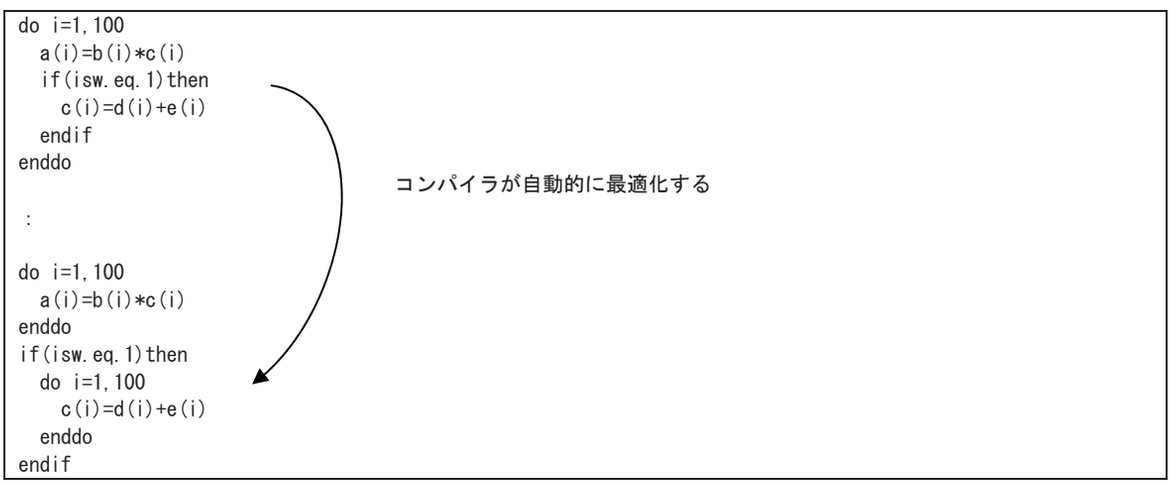

図 4.3-16 IF 文が DO ループの外に展開される変形イメージ

(2) マスクベクトルを用いてベクトル化しているループの性能

図 4.3-17 の DO ループは, マスクベクトルを用いてベクトル化するので TRUE, FALSE のどちらの場合も演算が実行される. 演算結果は, マスクベクトルを用いて TRUE に対応する要素のみ代入される. このため, flag_v の真の割合(以下, 真率)に関係なく, DO ループの処理時間はほぼ一定になる.

```
do n=1,nsp
  if(flag_v(n) .eq. 1) then
    tmpx3(n)=(tmpx1(n)-tmpx2(n))*2.0d0
    tmpy3(n)=(tmpy1(n)-tmpy2(n))*2.0d0
    tmpz3(n)=(tmpz1(n)-tmpz2(n))*2.0d0
  endif
enddo
```

図 4.3-17 一般的な IF 文を含む DO ループ

図 4.3-18 は, 以下の 3 パターンで図 4.3-17 の DO ループを SX-9 1CPU で実行するときの性能値である. それぞれの実行時間がほとんど変わらないことがわかる.

- Rfv_vt :flag_v の値が全て TRUE の場合
- Rfv_vf :flag_v の値が全て FALSE の場合
- Rfv_50 :flag_v の値がランダムに TRUE/FALSE となる場合

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
Rfv_vt	30	0.757(0.3)	25.237	48061.8	15954.8	99.59	256.0	0.757	0.000	0.000	0.006	0.327
Rfv_vf	30	0.755(0.3)	25.174	48182.2	15994.8	99.59	256.0	0.755	0.000	0.000	0.000	0.327
Rfv_50	30	0.758(0.3)	25.262	48015.4	15939.4	99.59	256.0	0.758	0.000	0.000	0.014	0.329

図 4.3-18　FTRACE 情報の一覧

(3)　事例1:IF 文をループの外に移動するチューニング

図 4.3-19 の多重ループは，TRUE 側の演算量が FALSE 側の演算量よりも大きい IF 文の例である．マスクベクトルを用いるベクトル化は，条件式の結果に関わらず，どちらの演算も実行しているため，本来必要な演算数より多くなる．以下に，ループの入れ替えによって IF 文をループの外に移動するチューニングの例を示す．

```
!オリジナル
153: +------>     do k=1,nz
154: |+----->       do j=1,ny
155: ||V---->         do i=1,nx
156: |||               if(lc(i,k).eq.0) then
157: |||                 p(1,i,j,k) = ff*va(1,i,j,k)*(f1*vb(i,j,k)+gv10) + d1*c(i,j,k)
158: |||                 p(2,i,j,k) = ff*va(2,i,j,k)*(f2*vb(i,j,k)+gv10) + d2*c(i,j,k)
159: |||                 p(3,i,j,k) = ff*va(3,i,j,k)*(f3*vb(i,j,k)+gv10) + d3*c(i,j,k)
160: |||               else
161: |||                 p(1,i,j,k) = d1*c(i,j,k)
162: |||                 p(2,i,j,k) = d2*c(i,j,k)
163: |||                 p(3,i,j,k) = d3*c(i,j,k)
164: |||               endif
165: ||V----         enddo
166: |+-----       enddo
167: +------     enddo
```

図 4.3-19　TRUE 側の演算量が FALSE 側の演算量よりも大きい IF 文のコード

この IF 文の条件式「lc(i,k).eq.0」はループ変数 j に依存しておらず，ループ内の配列に依存関係がない．このため DO ループ j を IF 文の内側に移動できる．

チューニング後のコードを図 4.3-20 に示す．IF 文は DO ループ j よりも外側にあり，マスクベクトル処理とならない．このチューニングで IF 文の条件式により内側の DO ループが選択され，真の時だけ実行される．

```
!チューニング
153: +------>     do k=1,nz
154: |+----->         do i=1,nx
155: ||                 if(lc(i,k).eq.0) then
156: ||V---->       do j=1,ny
157: |||                 p(1,i,j,k) = ff*va(1,i,j,k)*(f1*vb(i,j,k)+gv10) + d1*c(i,j,k)
158: |||                 p(2,i,j,k) = ff*va(2,i,j,k)*(f2*vb(i,j,k)+gv10) + d2*c(i,j,k)
159: |||                 p(3,i,j,k) = ff*va(3,i,j,k)*(f3*vb(i,j,k)+gv10) + d3*c(i,j,k)
160: ||V----        enddo
161: ||                 else
162: ||V---->       do j=1,ny
163: |||                 p(1,i,j,k) = d1*c(i,j,k)
164: |||                 p(2,i,j,k) = d2*c(i,j,k)
165: |||                 p(3,i,j,k) = d3*c(i,j,k)
166: ||V----        enddo
167: ||                endif
168: |+-----         enddo
169: +------     enddo
```

図 4.3-20　IF 文をループの外に移動したコード

チューニング前後の各ケースについてSX-9 1CPUを用いて性能評価を行う. 問題サイズは「nx=256, ny=256, nz=128」としている. IF 文の条件式の真率による評価を行うため, チューニング前後で真率をそれぞれ以下に設定する.

- Res_vt : IF 文の条件式が全て TRUE の場合
- Res_vf : IF 文の条件式が全て FALSE の場合
- Res_25 : IF 文の条件式が真率 25%となる場合
- Res_50 : IF 文の条件式が真率 50%となる場合
- Res_75 : IF 文の条件式が真率 75%となる場合

```
!オリジナル
PROC.NAME  FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS    MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT
                      TIME[sec](  %)    [msec]                    RATIO V.LEN     TIME    MISS    MISS CPU PORT  NETWORK

Res_vt           30    0.145( 11.8)     4.836  67836.6  36428.7 99.74 256.0    0.145   0.000   0.000    0.053    0.035
Res_vf           30    0.145( 11.8)     4.827  67955.3  36492.4 99.74 256.0    0.145   0.000   0.000    0.052    0.037
Res_25           30    0.163( 13.2)     5.421  60511.5  32495.0 99.74 256.0    0.163   0.000   0.000    0.067    0.054
Res_50           30    0.172( 13.9)     5.719  57358.6  30801.9 99.74 256.0    0.172   0.000   0.000    0.072    0.062
Res_75           30    0.163( 13.2)     5.427  60447.5  32460.7 99.74 256.0    0.163   0.000   0.000    0.068    0.049
```

図 4.3-21 チューニング前の FTRACE 情報

```
!チューニング
PROC.NAME  FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS    MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT
                      TIME[sec](  %)    [msec]                    RATIO V.LEN     TIME    MISS    MISS CPU PORT  NETWORK

Res_vt           30    0.123( 13.9)     4.116  61321.9  36687.8 99.71 256.0    0.123   0.000   0.000    0.064    0.044
Res_vf           30    0.095( 10.6)     3.151  18804.9   7987.7 99.11 256.0    0.094   0.000   0.000    0.045    0.060
Res_25           30    0.105( 11.8)     3.514  30567.0  16091.8 99.46 256.0    0.105   0.000   0.000    0.050    0.057
Res_50           30    0.115( 13.0)     3.844  40563.3  22932.7 99.60 256.0    0.115   0.000   0.000    0.054    0.055
Res_75           30    0.120( 13.4)     3.983  51256.7  30021.8 99.67 256.0    0.119   0.000   0.000    0.059    0.049
```

図 4.3-22 チューニング後の FTRACE 情報

チューニング前は, 実行時間にわずかな差が出ているが, 全ての場合で全浮動小数点演算数(MFLOPS×時間)の変化がない. このことから, IF 文の条件式の真率に関係なく TRUE/FALSE 両方の演算がなされていることがわかる.

チューニング後は, 全浮動小数点演算数は IF 文の条件式の真率よって増減している. 演算密度の違いで MFLOPS 値が低下している場合もあるが, どの場合でも処理時間の短縮が確認できる.

(4) 事例2:IF 文を他の処理に置き換えるチューニング

図 4.3-23 の例は, 粒子コードに見られる周期的境界条件で位置情報の補正を行うコードである. DO ループ中に多数の IF 文があるため, 多くのマスク付きベクトル演算を行い, 演算時間が長くなっている. この根本的な解決として IF 文に代わる処理への置き換えを実施する.

```
!チューニング前
54: V------>      do n=1,nsp
55: |               if(pa(n,1) < 0.0d0) pa(n,1)=pa(n,1)+xdim
56: |               if(pa(n,1) >= xdim) pa(n,1)=pa(n,1)-xdim
57: |               if(pa(n,2) < 0.0d0) pa(n,2)=pa(n,2)+ydim
58: |               if(pa(n,2) >= ydim) pa(n,2)=pa(n,2)-ydim
59: |               if(pa(n,3) < 0.0d0) pa(n,3)=pa(n,3)+zdim
60: |               if(pa(n,3) >= zdim) pa(n,3)=pa(n,3)-zdim
61: V------       enddo
```

図 4.3-23 IF 文による周期的境界条件のコード

図 4.3-23 のコードは, 位置情報が各方向の最小値から最大値の範囲に収まっているかを, IF 文の

条件式で判定している. 収まっていない場合は, 位置情報に各辺の長さを加減して周期的境界条件を満たす補正がなされる.

図 4.3-24 にチューニング後のコードを示す. ここでは, mod 関数を使用して周期的境界条件を満たす同等な補正を行っている. 全要素一律に各辺の長さを加算し, その長さで割った余りを求めることで周期的境界条件が満たされ, IF 文による場合分けが不要となる. ただし, 周期内の条件にある位置は, mod 関数による補正計算が行われるため, チューニング前のコードと比べて若干の誤差が生じる. どちらのコードもループ長は同じである.

```
!チューニング後
54: V------>      do n=1,nsp
55: |               pa(n,1)=dmod(pa(n,1)+xdim,xdim)
56: |               pa(n,2)=dmod(pa(n,2)+ydim,ydim)
57: |               pa(n,3)=dmod(pa(n,3)+zdim,zdim)
58: V------       enddo
```

図 4.3-24　mod 関数による周期的境界条件のコード

図 4.3-23 と図 4.3-24 の各場合について SX-9 1CPU を用いて性能評価を行う. 問題サイズは「nx=256, ny=128, nz=64, ns=32」としている. 要素数 nsp は nx, ny, nz, ns の積となり, 各方向の最大値を実数型変数 xdim, ydim, zdim で与えている.

```
PROC.NAME  FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT
                      TIME[sec](  %)    [msec]                   RATIO V.LEN     TIME    MISS    MISS CPU PORT  NETWORK

sample_f          30     9.557( 85.8)   318.582   6327.7   1895.8 99.87 256.0     9.557   0.000   0.000    0.021    8.658
```

図 4.3-25　チューニング前の FTRACE 情報

```
PROC.NAME  FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT
                      TIME[sec](  %)    [msec]                   RATIO V.LEN     TIME    MISS    MISS CPU PORT  NETWORK

sample_f          30     1.720( 51.5)    57.345  52707.6  17554.0 99.91 256.0     1.720   0.000   0.000    0.004    0.029
```

図 4.3-26　チューニング後の FTRACE 情報

図 4.3-25 と図 4.3-26 がそれぞれチューニング前後の性能値で, ベクトル演算率やベクトル長に大きな差異は見られない. チューニング前は配列 pa のロード・ストア回数が多く, 条件分岐のオーバヘッドで待ち時間も発生し, 「BANK CONFLICT/NETWORK」が大きくなっている. チューニング後は mod 関数を用い IF 文の条件分岐が取り除かれたことで, ベクトル演算の密度が上がり MOPS および MFLOPS の値が向上している.

4.3.4 DO ループ内の作業配列の削除による高速化

(1) 作業配列の削除の概要

一般にプログラムは入力データと出力結果の他にも, 途中経過を作業配列に保存する場合がある. しかし, 不必要な作業配列の利用はメモリ負荷を増加させ, 演算性能を低下させることになる.

図 4.3-27 の例は, ループ内でのみ使用する途中の演算結果 a(n)+b(n)を作業配列 work(n)に保存している. この途中の演算結果をその後の処理で利用することがなければ, 作業配列 work(n)を使わずスカラ変数に書き換え, 作業配列への保存を回避できる. このようにすることで, 途中の演算結果は作業レジスタに保存され, メモリ負荷が下がり演算性能が向上する.

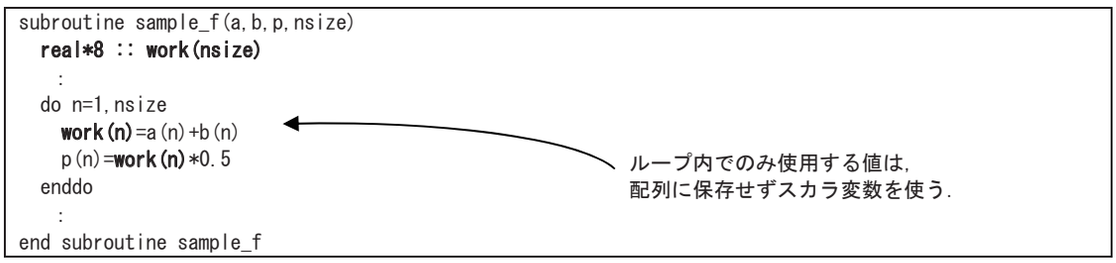

図 4.3-27 作業配列からスカラ変数に書き換えることができる例

(2) 作業配列を削減するチューニングの事例

図 4.3-28 のサブルーチン sample_f1, sample_f2 は, プログラム中の複数の箇所で同じ演算となる部分を括りだし, 共通部品としてサブルーチンにしたものである. このプログラムは, 3 次元配列のデータを 1 次元作業配列にコピーしサブルーチン sample_f1 へ渡し, サブルーチン sample_f1, sample_f2 で計算し, 1 次元作業配列の結果を 3 次元配列に書き戻している.

```
!オリジナル
 34: |+----->       do k=1,kmax
 35: ||+---->       do j=1,jmax
 36: |||
 37: |||V--->         do i=1,imax
 38: ||||              wi(1,i)=a(1,i,j,k)
 39: ||||              wi(2,i)=a(2,i,j,k)
 40: ||||              wi(3,i)=a(3,i,j,k)
 41: ||||              wv(i)  =b(i,j,k)
 42: |||V---           enddo
 43: |||
 44: |||               call sample_f1(imax,wi,c ,wt)
 45: |||               call sample_f2(imax,wv,wt,wo)
 46: |||
 47: |||V--->         do i=1,imax
 48: ||||              p(i,j,k)=wo(i)
 49: |||V---           enddo
 50: |||
 51: ||+----        enddo
 52: |+-----        enddo

 85:             subroutine sample_f1(lmax,wi,wc,wt)
 :
 97: V------>     do l=2,lmax-1
 98: |              wt(l)=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
 99: |                   +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
100: |                   +wc(3)*(wi(3,l+1)-wi(3,l-1))
101: V------       enddo

106:            subroutine sample_f2(lmax,wv,wt,wo)
 :
118: V------>     do l=2,lmax-1
119: |              wo(l)=wt(l)*wv(l)
120: V------       enddo
```

図 4.3-28 作業配列を削減前のコード

図 4.3-29 のチューニングは, 作業配列 wi,wv と元の入力配列 a,b の 1 列が同じ値を参照していることに着目する. この場合, 作業配列 wi,wv を使わないで配列 a,b の 1 列を直接引き渡すことができる. 同様に引数の作業配列 wo と出力配列 p の 1 列が同じ値を更新しているので, 出力データも作業配列 wo を使わないで配列 p へ直接出力できる. このようにサブルーチン呼び出し前後に作業配列 wi, wv, wo を使用しないでデータを引き渡すことができるため, 作業配列へのメモリアクセス回数が削減できる.

```
!チューニング①
 58: |+-----> do k=1,kmax
 59: ||+----> do j=1,jmax
 60: |||
 61: |||        call sample_f1(imax,a(1,1,j,k),c ,wt      )
 62: |||        call sample_f2(imax,b(1,j,k)  ,wt,p(1,j,k))
 63: |||
 64: ||+----   enddo
 65: |+-----   enddo

 85:          subroutine sample_f1(lmax,wi,wc,wt)
 :
 97: V------>   do l=2,lmax-1
 98: |            wt(l)=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
 99: |                 +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
100: |                 +wc(3)*(wi(3,l+1)-wi(3,l-1))
101: V------    enddo

106:         subroutine sample_f2(lmax,wv,wt,wo)
 :
118: V------>   do l=2,lmax-1
119: |            wo(l)=wt(l)*wv(l)
120: V------    enddo
```

図 4.3-29 ［第 1 段階］入力データと出力データを引数にする修正

図 4.3-30 の更なるチューニングは，サブルーチン sample_f1，sample_f2 を 1 つのサブルーチン sample_f にまとめ，これらのサブルーチンそれぞれにあった DO ループを融合している．このループ融合によって，sample_f1 から sample_f2 に受け渡している途中の結果を，作業配列 wt を使わずに演算処理できる．

```
!チューニング②
 71: |+-----> do k=1,kmax
 72: ||+----> do j=1,jmax
 73: |||
 74: |||        call sample_f(imax,a(1,1,j,k),b(1,j,k),c,p(1,j,k))
 75: |||
 76: ||+----   enddo
 77: |+-----   enddo

127:         subroutine sample_f(lmax,wi,wv,wc,wo)
 :
141: V------>   do l=2,lmax-1
142: |            s=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
143: |             +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
144: |             +wc(3)*(wi(3,l+1)-wi(3,l-1))
145: |            wo(l)=s*wv(l)
146: V------    enddo
```

図 4.3-30 ［第 2 段階］サブルーチンにまたがる 2 つの DO ループを融合する修正

(3) 性能評価

チューニング前後の各場合について SX-9 1CPU を用いて性能評価を行う．(2)で示したコードを用い，問題サイズは「imax=1000, jmax=50, kmax=50」としている．図 4.3-31 は，チューニング前後の性能値である．

- case_000：図 4.3-28 チューニング前の結果
- case_001：図 4.3-29 チューニング［第 1 段階］での結果

- case_002:図 4.3-30 チューニング[第 2 段階]での結果

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
case_000	1000	5.841(100.0)	5.841	12542.9	3844.5	98.83	249.7	5.840	0.000	0.000	1.862	4.740
case_001	1000	1.821(100.0)	1.821	26312.6	12331.1	98.93	249.5	1.820	0.000	0.000	0.520	1.145
case_002	1000	1.064(100.0)	1.064	40150.0	21097.7	99.26	249.5	1.064	0.000	0.000	0.516	0.423

図 4.3-31 FTRACE 情報一覧

case_000→case_001→case_002 とチューニングの段階を経て,「BANK CONFLICT/ NETWORK」の時間が短縮されている. これは作業配列へのデータ転送が減っているためである. データ転送が削減されメモリ負荷が下がったことで, 結果的に演算器の待ち時間が減少し, MFLOPS 値が向上している.

4.3.5 組み込み関数 MAX/MIN の利用による高速化

(1) 組み込み関数 MAX/MIN の利用による高速化の概要

最大値・最小値を求める計算はマクロ演算を利用してベクトル化され, 図 4.3-32 に示すようにコンパイル診断メッセージに「Macro operation Max/Min」と表示される. また, 4.1 節で述べたとおり, SX-9 から MAX/MIN 関数がハードウェア命令化され, 最大値・最小値を求める計算時間の短縮が行われる. 本節では, 最大値を求めるコードでベクトル化される場合とベクトル化されない場合を示し, さらにハードウェア命令が使われた場合と, 使われない場合の性能の違いを示す.

```
    61   vec   (  26): Macro operation Max/Min.

 :(※中略)

57: W------>      do k=1,kmax
58: |*----->        do j=1,jmax
59: ||*---->          do i=1,imax
60: |||                 tmp=r1(i,j,k)*r2(i,j,k)
61: |||                 s=max(s,tmp)
62: ||*----           enddo
63: |*-----         enddo
64: W------       enddo
```

行番号 61 番で最大値・最小値のマクロ演算を適用するメッセージ

図 4.3-32 MAX/MIN 関数利用の例とコンパイル診断メッセージ

(2) 最大値を求めるコードの事例

次に示す図は, 3 種類の最大値を求めるコードで, 以下の特徴がある.

- コード①(図 4.3-33)は, 組み込み関数 MAX を利用して最大値を求める
- コード②(図 4.3-34)は, 大小関係を IF 文で判断し, 大きい値で変数 s を更新し最大値を求める
- コード③(図 4.3-35)は, ループの 1 回目で初期値を設定し, 2 回目以降のループで最大値を求める

コード①は, 組み込み関数 MAX を利用しているので SX-9 のハードウェア命令が使われベクトル化される. コード②はコンパイラが最大値を求める処理と認識できるのでベクトル化される. コード③は条件分岐が複雑であり, コンパイラが最大値を求める処理と認識できないのでベクトル化されない.

```
!コード①
131:              subroutine sample_b1(r1,r2,s)
:
143:                s=-999d0
144: V------>       do i=1,imax*jmax*kmax
145: |                tmp=r1(i)*r2(i)
146: |                s=max(s,tmp)
147: V------        enddo
```

図 4.3-33　最大値を求めるコード①

```
!コード②
154:              subroutine sample_b2(r1,r2,s)
:
166:                s=-999d0
167: V------>       do i=1,imax*jmax*kmax
168: |                tmp=r1(i)*r2(i)
169: |                if(tmp .gt. s) s=tmp
170: V------        enddo
```

図 4.3-34　最大値を求めるコード②

```
!コード③
177:              subroutine sample_b3(r1,r2,s)
:
190:                lflag=.true.
191: +------>       do i=1,imax*jmax*kmax
192: |                tmp=r1(i)*r2(i)
193: |                if(lflag) then
194: |                  s=tmp
195: |                  lflag=.false.
196: |                else
197: |                  s=max(s,tmp)
198: |                endif
199: +------        enddo
```

図 4.3-35　最大値を求めるコード③

(3)　性能評価

各コードを SX-9 1CPU を用いて性能評価を行う. 問題サイズは「imax*jmax*kmax=256^3」で十分に長いループ長としている.

表 4.3-2 3 種類コードの性能

	性能値
コード①	20,999MFLOPS
コード②	11,373MFLOPS
コード③	93MFLOPS

コード①は MAX/MIN 関数でベクトル化され SX-9 で強化されたハードウェア命令が使われるため, 高い性能を示している. コード②はベクトル化されるが SX-9 のハードウェア命令が使われてないため, コード①と比較して半分程度の性能である. コード③はベクトル化されていないので性能が出ていない. SX-9 で最大値・最小値を求める場合, 高い性能を得るには MAX/MIN 関数を使用してベクトル化することが必要である.

4.3.6 科学技術計算ライブラリASLの利用による高速化

(1) 科学技術計算ライブラリ利用の概要

数値計算などで利用している連立 1 次方程式, 固有値・固有ベクトル, 高速フーリエ変換(FFT), 特殊関数, 乱数などの機能は計算ライブラリとして準備されている. このような計算ライブラリをユーザが作成することはできるが, ハードウェアの性能を引き出すために複雑な最適化が必要であり, 高い性能を得るのは容易ではない. また一般の計算ライブラリは, 特定のハードウェアに向けての最適化は通常行われておらず, SX-9 の最大性能を得ることは難しい. SX-9 には科学技術計算ライブラリ ASL があり, これは広範な分野の数値シミュレーションプログラムの作成を強力に支援する計算ライブラリである. ASL はハードウェアの性能を最大限に引き出す最適化が行われており, ユーザはこれらのライブラリを使うことにより, SX-9 の性能を引き出すことが容易に行える.

これから科学技術計算ライブラリ ASL の FFT とフリーソフトウェアの FFT ライブラリとの性能を比較する. 比較対象のフリーソフトウェアは, NETLIB の FFTPACK, および筑波大学高橋大介氏作成の FFTE 4.1 としている. それぞれの FFT ライブラリは, 以下の URL からダウンロードできる.

FFTPACK	http://www.netlib.org/fftpack/
FFTE	http://www.ffte.jp/

(2) ASL と FFTPACK の性能評価

ここでは1次元複素数フーリエ変換を ASL のサブルーチン JFCOSI/JFCOSF と FFTPACK のサブルーチン ZFFTI/ZFFTF/ZFFTB を用いて比較する. それぞれの評価コードを図 4.3-36 と図 4.3-37 に示す. これらサブルーチンは周期2πの1周期を n 等分した n 個の複素数データが与えられたとき, n 個のデータに対する離散フーリエ変換(順変換と逆変換)を行う.

```
! 1次元複素数 FFT (ASL) の評価コード
    integer, parameter :: ld=2**26
    complex(kind=8) :: c(ld),trigs((3*ld)/2),wc(ld)
    integer :: ifax(128)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=11,26
     n=2**ii
     do isw=1,-1,-2
       do i=1,n
        c(i) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*i))
       enddo

       call JFCOSI(n,ifax,trigs,ierr)
       call clock(t1)
       call JFCOSF(n,c,ld,isw,ifax,trigs,wc,ierr)
       call clock(t2)
       tm=(t2-t1)
       perf(isw)=5.d0*n*log(dble(n))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) n,perf(1),perf(-1)
    enddo
    end
```

ASL の FFT サブルーチン

図 4.3-36 1 次元複素数 FFT(ASL)の評価コード

```
! 1 次元複素数 FFT (FFTPACK) の評価コード
    integer, parameter :: ld=2**26
    complex(kind=8) :: w(ld*2+15),cx(ld)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=11,26
     n=2**ii
     do isw=1,-1,-2
       do i=1,n
        cx(i) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*i))
       enddo

       call ZFFTI (n,w)
       call clock(t1)
       if(isw.eq.1)then
        call ZFFTF (n,cx,w)
       else
        call ZFFTB (n,cx,w)
       endif
       call clock(t2)
       tm=(t2-t1)
       perf(isw)=5.d0*n*log(dble(n))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) n,perf(1),perf(-1)
    enddo
    end
```

FFTPACK のサブルーチン

図 4.3-37　1 次元複素数 FFT（FFTPACK）の評価コード

ここで性能測定プログラムに使う評価コードについて説明する. この評価コードでは, 必要なデータ領域を確保し, FFT の入力データおよびワーク領域の初期化を行って1次元複素数フーリエ変換のサブルーチンを呼び出している. 実行時間の計測は, フーリエ変換のサブルーチンを呼び出す前後で, タイマールーチン（CLOCK）を挟んで行っている.

つぎに性能の算出方法について説明する. FFT の代表的な計算方法である Cooley-Tukey のアルゴリズムからデータ数 n の1次元複素数 FFT の演算数を数え上げると, 次式で表わされる.

$$演算数＝5n\times\log_2(n)$$

この Cooley-Tukey のアルゴリズムの演算数は基数が 2 の場合の結果であるが, 基数が 2 以外でも大きく違わないため, 演算数を求めるには良い近似値となる. この方法で FFT の演算数を求め, 実行時間で除する事で性能(GFLOPS 値)を算出している.

図 4.3-38 は, データ数を 2^{11} (=2,048) から 2^{26}(=67,108,864)まで 2 のべき乗で増加させた時の FFT の性能である. 図中, fwd は順変換を bwd は逆変換を意味している.

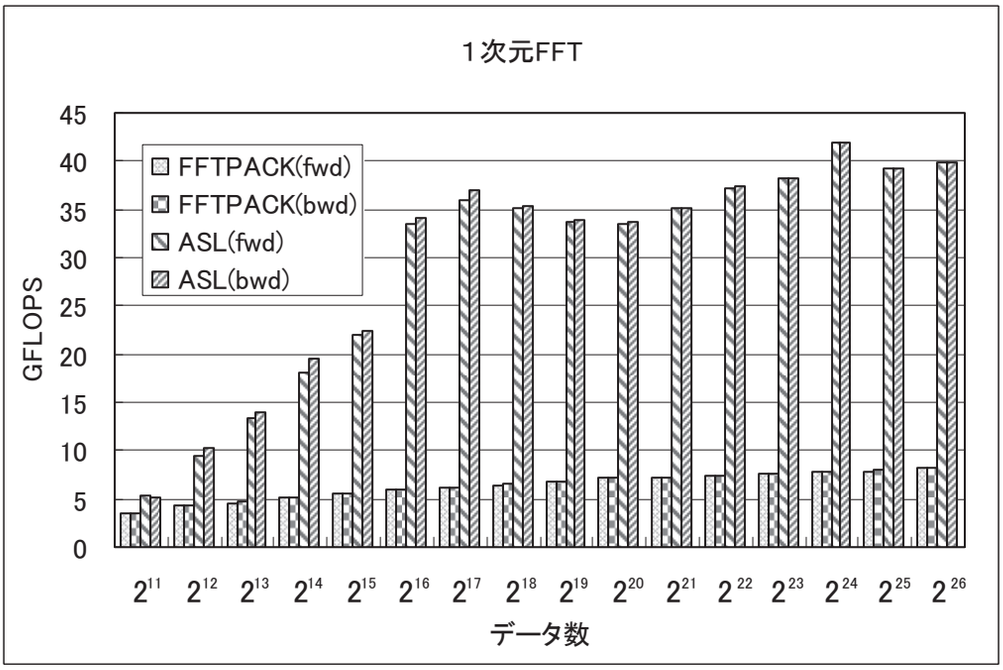

図 4.3-38 1 次元複素数 FFT の性能比較

FFTPACK は，データ数を増加しても大きな性能向上が得られない． ASL は，データ数を増加すると 2^{17}(=131,072)で 36GFLOPS と性能が向上し，最大性能は 2^{24}(=16,777,216)で 41GFLOPS となる．ASL と FFTPACK の性能を比較すると，データ数が大きいところで約 5 倍 ASL が高速である．

(3) ASL と FFTE の性能評価

ここでは2次元複素数フーリエ変換を ASL のサブルーチン ZFC2FB/ZFC2BF と，FFTE はベクトル用のサブルーチン ZFFT2D を用いて行っている(注 1)．それぞれの評価コードを，図 4.3-39 と図 4.3-40 に示す．これらのサブルーチンは，2次元の各次元のデータで離散フーリエ変換(順変換と逆変換)を行う．

```
! ２次元複素数 FFT （ASL） の評価コード
    integer, parameter :: ld=10241
    complex(kind=8) :: c(ld*ld),trigs(2*ld),wc(ld*ld)
    integer :: ifax(40)
    real(kind=8):: t1,t2,tm,perf(-1:1)
    do ii=1024,10240,1024
    do isw=1,-1,-2
      nx=ii
      ny=ii
      lx=nx+1
      ly=ny+1
      call init(c,lx,ly,nx,ny)

      call ZFC2FB(nx,ny,c,lx,ly,0,ifax,trigs,wc,ierr)
      call clock(t1)
      call ZFC2BF(nx,ny,c,lx,ly,isw,ifax,trigs,wc,ierr)
      call clock(t2)
      tm=(t2-t1)
      perf(isw)=5.d0*nx*ny*(log(dble(nx))+log(dble(ny)))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) ii,perf(1),perf(-1)
    enddo
    end
    subroutine init(c,lx,ly,nx,ny)
    complex(kind=8) :: c(lx,ly)
    do j=1,ny
     do i=1,nx
      c(i,j) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*j))
     enddo
    enddo
    end
```

ASL の FFT サブルーチン

図 4.3-39　2 次元複素数 FFT(ASL)の評価コード

```
! ２次元複素数 FFT （FFTE） の評価コード
    integer, parameter :: ld=10240
    complex(kind=8) :: a(ld*ld)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=1024,10240,1024
     if(mod(ii,7).eq.0) cycle
     do isw=1,-1,-2
      nx=ii
      ny=ii
      call init(a,nx,nx,nx,ny)

      call ZFFT2D(a,nx,ny,0)
      call clock(t1)
      call ZFFT2D(a,nx,ny,-isw)
      call clock(t2)
      tm=(t2-t1)
      perf(isw)=5.d0*nx*ny*(log(dble(nx))+log(dble(ny)))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) ii,perf(1),perf(-1)
    enddo
    end
    subroutine init(c,lx,ly,nx,ny)
    complex(kind=8) :: c(lx,ly)
    do j=1,ny
     do i=1,nx
      c(i,j) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*j))
     enddo
    enddo
    end
```

FFTE のサブルーチン

図 4.3-40　2 次元複素数 FFT(FFTE)の評価コード

図 4.3-41 は 2 次元データの各次元とも 1,024 の増分で 1,024×1,024 から 10,240×10,240 までデータ数を増加させた時の FFT の性能である. ただし, FFTE はデータ数に 2, 3, 5 以外の約数が入ると計算できないため, 約数に 7 が入るデータ数は除外している(注 2). 実行時間の計測は, (2)と同様, フーリエ変換のサブルーチンを呼び出す前後で行い, タイマールーチンを呼ぶ前に FFT の入力データとワーク領域の初期化を行っている. 演算数は1次元の FFT での算出方法を2次元の FFT に拡張し, 性能を求めている.

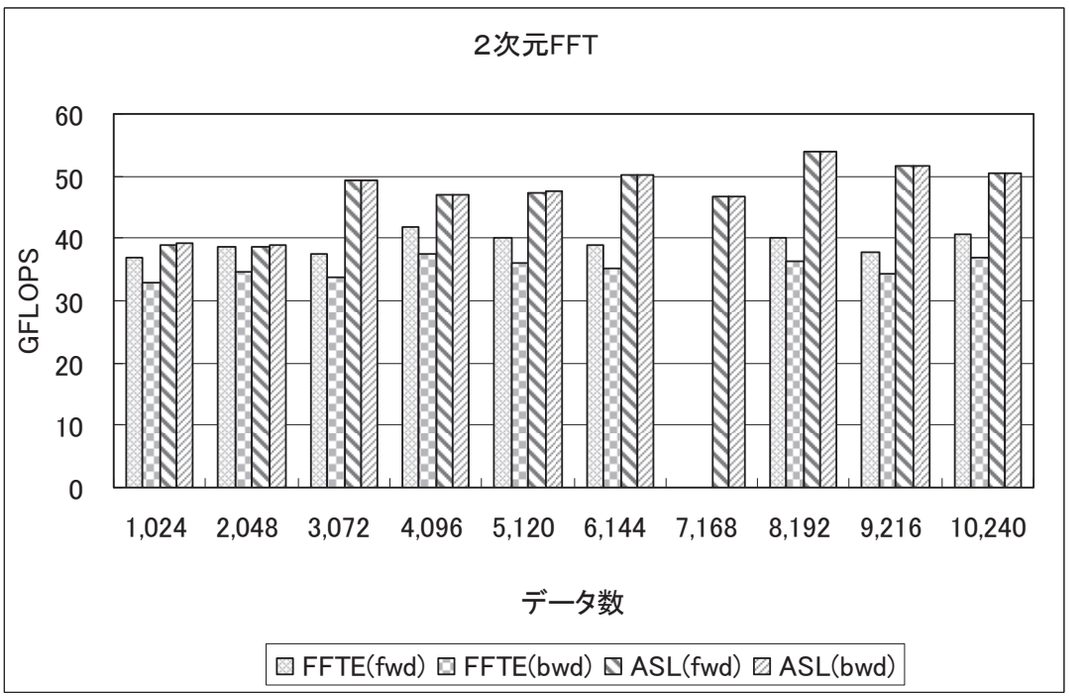

図 4.3-41 2 次元複素数 FFT の性能比較

FFTE はデータ数により 32～42GFLOPS の性能である. ASL はデータ数が 8,192×8,192 のとき, 最大性能の 54GFLOPS となる. ASL と FFTE の性能を比較すると 1.1～1.4 倍 ASL が高速である.

(注 1) FFTE は, ベクトル用およびスカラ用に最適化された FFT のサブルーチンが用意されている. それぞれのプラットフォームに適しているサブルーチンを利用すること.

(注 2) FFTE は, データ数に 2,3,5 以外の約数が入っていると FFT の計算ができない仕様である. 例えば 7,168 は約数に 7 が入り計算できない. 一方 ASL の FFT は, データ数に約数の制約はなく任意のデータ数で FFT の計算ができる.

(4) 性能評価のまとめ

科学技術計算ライブラリ ASL の FFT サブルーチンとフリーソフトウェアの FFT ライブラリである FFTPACK, FFTE との間で性能を比較している. その結果, ASL の FFT サブルーチンが高い性能を示している. FFTE にはベクトル用に最適化されている FFT サブルーチンが用意されており, 実行効率が 30%～40%と比較的よい性能が得られているが, ASL の FFT サブルーチンは実行効率が最大で 50%を超え, より高速である.

SX で FFT の計算を行う場合は, SX のハードウェア向けに高度な最適化が施されている科学技術計

算ライブラリASLを使うことで，フリーソフトウェアのFFTライブラリを上回る性能が引き出せることがわかる．

4.3.7 リストベクトルを含む DO ループの最適化

(1) リストベクトルの概要

リストベクトルとは，図 4.3-42 に示すように間接参照される配列 IX にしたがって，配列 A の参照または更新をベクトル処理で行うものである．図 4.3-43 に図 4.3-42 に示したコードのメモリアクセスパターンを示す．リストベクトルを用いることで，連続でないランダムなメモリアクセスをベクトル処理として行うことができる．リストベクトルはDOループ中のIF文の除去や，ベクトル化できないデータ参照関係を回避する平面法などに活用されている．平面法については，「高速化活動報告第 4 号」を参照のこと．

```
!参照
do  i=1,n
  T(i)=A(IX(i))
enddo

!更新
do  i=1,n
  A(IX(i))=T(i)
enddo
```

図 4.3-42 リストベクトルのコード

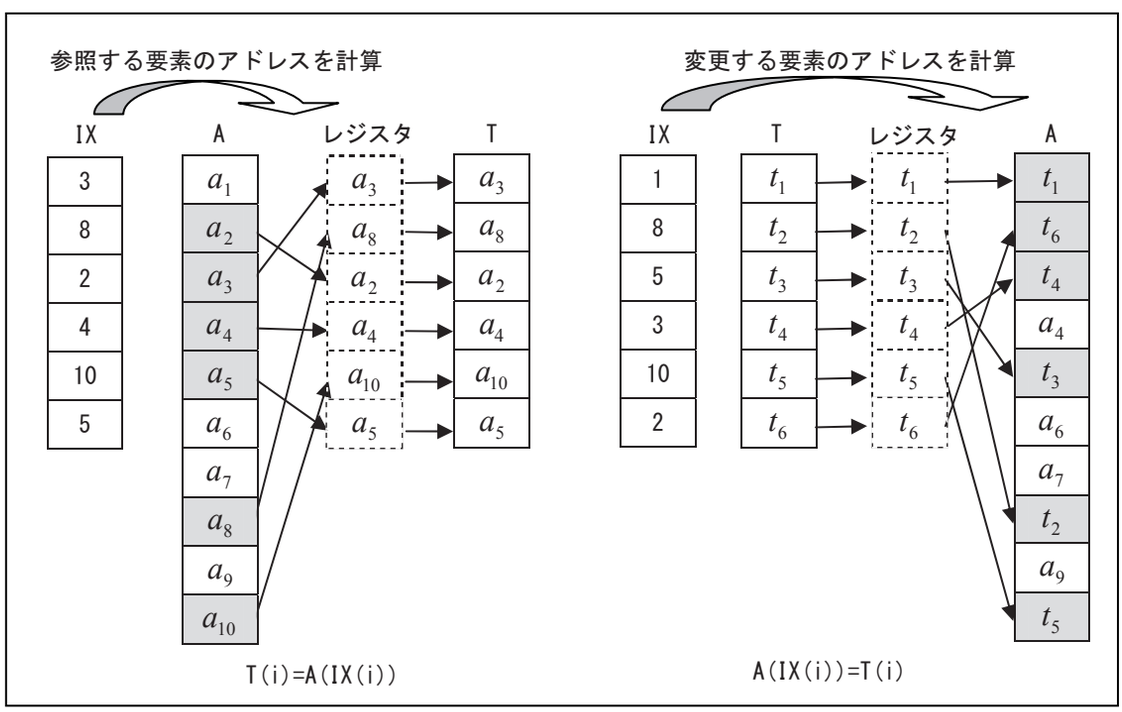

図 4.3-43 リストベクトルのメモリアクセス

(2) 事例1:リスト参照のメモリレイテンシの隠ぺい

リストベクトルによる参照や更新が同一 DO ループ内に複数回現れるプログラムの高速化の手法について説明する．

図 4.3-44 にリストベクトルによる参照や更新が同一 DO ループ内に複数回現れるコード(注)を示す．

間接参照 n2 による配列 p の参照は，間接参照 n1 による同配列 p への更新の完了を待つためメモリレイテンシ（遅延）が表面化し，演算性能を発揮できない．そこで配列 p のリストベクトルによる参照と更新のタイミングをずらすため，一度別の作業配列に連続データとして保存する．

```
121: +------>       do icol=1,ncol
122: |              !cdir nodep
123: |V----->         do edge=esep(icol),esep(icol+1)-1
124: ||                 n1=e2n(edge,1)
125: ||                 n2=e2n(edge,2)
 :
141: ||                 x1=u(n1,1)
142: ||                 y1=u(n1,2)
143: ||                 x2=u(n2,1)
144: ||                 y2=u(n2,2)
 :
161: ||                 !userfunc はインライン化によりベクトル化されるユーザ定義関数
162: ||                 v=userfunc(x1,y1,z1,x2)
 :
191: ||                 p(n1)=p(n1)+v
192: ||                 p(n2)=p(n2)-v
 :
209: |V-----          enddo
210: +------        enddo
```

図 4.3-44 リストベクトルが同一 DO ループに複数回現れるコード

図 4.3-45 に作業配列を使って最適化するコードを示す．先ず配列 p を間接参照 n1,n2 により参照し，それぞれの作業配列 pn1,pn2 に連続データとして保存する．次に作業配列 pn1,pn2 を使って連続アクセスのデータで演算する．最後に演算結果を，作業配列から元の配列 p へ書き戻している．このようにすることで，配列 p の間接参照による参照を更新の前にまとめて行い，更新完了の待ちによる間接参照時のメモリレイテンシの影響を少なくできる．

```
121: +------>       do icol=1,ncol
122: |              ! リストベクトルによる参照処理の集約
123: |V----->         do edge=egsep(icol),egsep(icol+1)-1
124: ||                 n1=e2n(edge,1)
125: ||                 n2=e2n(edge,2)
 :
141: ||                 x1_a(edge)=u(n1,1)
142: ||                 y1_a(edge)=u(n1,2)
143: ||                 x2_a(edge)=u(n2,1)
144: ||                 y2_a(edge)=u(n2,2)
 :
161: ||                 pn1(edge)=p(n1)
162: ||                 pn2(edge)=p(n2)
 :
179: |V-----          enddo
180: |              ! 連続ベクトルによる演算
181: |V----->         do edge=egsep(icol),egsep(icol+1)-1
182: ||                 v=userfunc(x1_a(edge),y1_a(edge),x2_a(edge),y2_a(edge))
 :
211: ||                 pn1(edge)=pn1(edge)+v
212: ||                 pn2(edge)=pn2(edge)-v
 :
229: |V-----          enddo
230: |              ! リストベクトルによる更新処理の集約
231: |              !cdir nodep
232: |V----->         do edge=egsep(icol),egsep(icol+1)-1
233: ||                 n1=e2n(edge,1)
234: ||                 n2=e2n(edge,2)
 :
250: ||                 p(n1)=pn1(edge)
251: ||                 p(n2)=pn2(edge)
 :
268: |V-----          enddo
269: +------       enddo
```

図 4.3-45　作業配列を使った演算コード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-46 に FTRACE の情報を示す．org がチューニング前のコード，tune がチューニング後のコードを表している．配列 p の間接参照を更新より先に一括して行っていることで，メモリレイテンシを隠ぺいでき，図 4.3-46 で示されるようにメモリネットワーク競合時間（NETWORK）が半減し，CPU 時間比にして 1.95 倍に性能が向上している．

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
org	100	24.849(12.8)	248.493	6508.8	1817.5	99.95	250.1	24.843	0.001	0.001	2.039	18.972
tune	100	12.707(6.5)	127.074	16339.5	3554.2	99.80	250.1	12.640	0.016	0.012	2.301	8.792

図 4.3-46　チューニング前後の FTRACE 情報

（注）本コードは，間接参照 n1, n2 がそれぞれ最内の DO ループの中で重複する値を取らないことと，間接参照 n1 と n2 の間にも値の重複がないことを想定している．いずれも重複がある場合，ベクトル化によりデータの不整合が起こる．コンパイラはデータ依存性を判断できないため，図 4.3-44 の 122 行目および図 4.3-45 の 231 行目に指示行「!cdir nodep」を追加し，コンパイラに重複がないことを指示することでベクトル化を行っている．

(3) 事例2:配列の次元の入れ替えによるリストベクトルの回避

リストベクトル処理はメモリアクセスのレイテンシを隠ぺいすることが難しく，メモリを連続にアクセスする処理と比べて，処理時間を多く必要とし実効性能が低下する．

図 4.3-47 にリストベクトルのコードを示す．最内の i の DO ループの配列 ain はリスト il2，il1，ir1，ir2 により間接参照され，リストベクトルとなっている．

```
68: +------>            do k=1,kmax
69: |+----->              do j=1,jmax
70: ||V---->                do i=2,imax
71: |||                       il2=ilir(i,1)
72: |||                       il1=ilir(i,2)
73: |||                       ir1=ilir(i,3)
74: |||                       ir2=ilir(i,4)
75: |||                       stbc=wstbc(i)
76: |||                       dqr0=ain(i,j,k)
77: |||                       dqr1=ain(ir1,j,k)
78: |||                       dqr2=ain(ir2,j,k)
79: |||                       dql0=ain(i-1,j,k)
80: |||                       dql1=ain(il1,j,k)
81: |||                       dql2=ain(il2,j,k)
```

図 4.3-47 リストベクトル処理のコード

配列 ain に対して，i,j,k の次元から j,k,i の次元へ入れ替えを行うとともに，i の DO ループを最外に移動させる．図 4.3-48 に配列の次元を入れ替えたコードを示す．この変更により，配列 ain の参照は j,k の DO ループに対し連続ベクトルとなり，リストベクトルが回避される．加えてこの変更により，添え字 j が配列の下限から上限までの値を取ることで j,k の DO ループが一重化でき，ベクトル長が長くなる（注）．

```
68: V----->            do i=2,imax
69: |                    il2=ilir(i,1)
70: |                    il1=ilir(i,2)
71: |                    ir1=ilir(i,3)
72: |                    ir2=ilir(i,4)
73: |                    stbc=wstbc(i)
74: |W---->              do k=1,kmax
75: ||*--->                do j=1,jmax
76: |||                      dqr0=ain(j,k,i)
77: |||                      dqr1=ain(j,k,ir1)
78: |||                      dqr2=ain(j,k,ir2)
79: |||                      dql0=ain(j,k,i-1)
80: |||                      dql1=ain(j,k,il1)
81: |||                      dql2=ain(j,k,il2)
```

図 4.3-48 配列の次元を入れ替えたコード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-49 に FTRACE の情報を示す． org がリストベクトル処理主体のコード，tune が配列の次元を入れ替えたコードを表している．リストベクトルの回避とベクトル長が長くなることで，メモリネットワーク競合時間が大幅に削減され，CPU 時間比にして 11.41 倍に性能が向上している．

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
org	1	33.964(30.1)	33963.691	7920.3	1748.9	99.55	99.0	33.964	0.000	0.000	1.875	29.967
tune	1	2.977(2.6)	2976.662	46796.1	19955.2	99.50	250.0	2.976	0.000	0.000	0.010	1.418

図 4.3-49 チューニング前後の FTRACE 情報

(注)図 4.3-48 のコードにおける j,k の DO ループは，ループ長 jmax×kmax の一重ループとして処理される．例えば ain(j,k,i)への参照は，1～jmax×kmax の値を取るループ変数 jk を用いることで ain(jk,1,i)への参照と同様に処理される．

(4)　事例3：IF 文の変更によるリストベクトルの回避

図 4.3-50 のコードは，22 行目の IF 文で決定される変数 j を用いて配列 c を間接参照しているため，リストベクトルとなる(注)．この場合，IF 文の処理内容を変更することでリストベクトルを回避できる．

```
21: V------>        do i=1,n
22: |                 if ( x(i) .ge. 0.0 ) then
23: |                   j=2
24: |                 else
25: |                   j=1
26: |                 endif
27: |                 z1(i)=z1(i)+y1(i)*c(1,j)
28: |                 z2(i)=z2(i)+y2(i)*c(2,j)
29: V------         enddo
```

図 4.3-50　IF 文で決定した変数を用いて間接参照を行うコード

図 4.3-51 にリストベクトルの回避コードを示す．22 行目以降の IF 文を配列 c の値を直接参照し，新たに導入した作業変数 c1,c2 に格納する．これらの作業変数 c1,c2 を用いた 29 行目以降の演算は連続ベクトルとなるため，リストベクトルが回避される．加えて，IF 文の処理は配列 c の値を定数としてレジスタ上で利用できるため，メモリアクセスも大幅に低減される．

```
21: V------>        do i=1,n
22: |                 if ( x(i) .ge. 0.0 ) then
23: |                   c1=c(1,2)
24: |                   c2=c(2,2)
25: |                 else
26: |                   c1=c(1,1)
27: |                   c2=c(2,1)
28: |                 endif
29: |                 z1(i)=z1(i)+y1(i)*c1
30: |                 z2(i)=z2(i)+y2(i)*c2
31: V------         enddo
```

図 4.3-51　IF 文内で配列を直接参照するコード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-52 に FTRACE の情報を示す．org がリストベクトルのコード，tune がリストベクトル回避コードを表している．リストベクトルの回避とメモリアクセスの低減によりメモリネットワーク競合時間が大幅に削減され，CPU 時間比にして 9.90 倍に性能が向上している．

PROC.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK
org	1	22.664(27.6)	22663.970	8405.8	1764.9	99.73	255.8	22.664	0.000	0.000	1.275	20.002
tune	1	2.287(2.8)	2286.695	70192.4	17492.5	99.68	255.8	2.287	0.000	0.000	0.023	0.693

図 4.3-52　チューニング前後の FTRACE 情報

(注)ループ変数 i の値それぞれに応じた変数 j の値は，ベクトルレジスタもしくはコンパイラが自動生成する作業配列に，一度ベクトルデータとして格納される．そのため j による配列 c の間接参照は，この格納されたベクトルデータを元にリストベクトルとして処理される．

4.3.8 SOR 法のベクトル化（Red Black 法）

(1) SOR 法の概要

SOR 法（Successive Over-Relaxation method）とは，n 元連立方程式 Ax=b を反復法で解く手法の一つである．図 4.3-53 に示すとおり，ある要素 p(i,j,k)の計算には近傍の要素 p(i-1,j,k)，p(i+1,j,k)，p(i,j-1,k)，p(i,j+1,k)，p(i,j,k-1)，p(i,j,k+1)が必要な計算を行う．

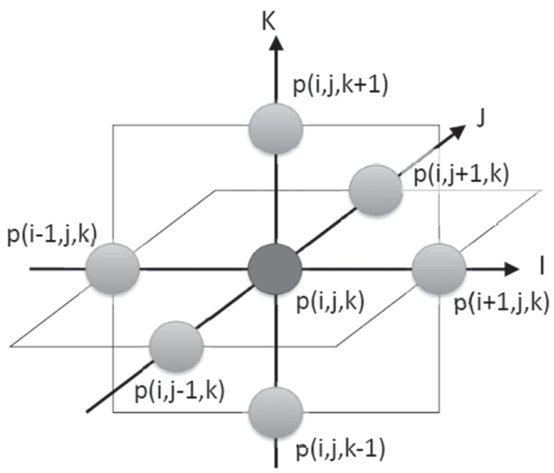

図 4.3-53 SOR 法計算の関係図

図 4.3-54 に SOR 法のコードを示す．p(i,j,k)の値を計算する際に，p(i-1,j,k)の要素を参照する必要がある．ベクトル化の対象となる最内の DO ループでは，p(i-1,j,k)の値は 1 回前で定義されており（注 1），この定義・参照関係がベクトル化を阻害する依存関係になる．そのためベクトル化できず（注 2），SX-9 で高い実効性能を得ることができない．

（注 1）p(i+1,j,k)は参照された後で値が更新されるので，ベクトル化を阻害する要因にはならない．

（注 2）図 4.3-54 の例のように，コンパイラはベクトル化可能な部分のみベクトル化を行う．これは部分ベクトル化と呼ばれ編集リストでは「S」で示される．

```
115  +-----> do k = n_bnd + 1, n_bnd + n_cell
116  |+----->  do j = n_bnd + 1, n_bnd + n_cell
117  ||V--->   do i = n_bnd + 1, n_bnd + n_cell
 :
141  |||  S      p_a = c0
142  |||  S  .   * ( p(i-1,j  ,k  ) * xm + p(i+1,j  ,k  ) * xp
143  |||  S  .     + p(i  ,j-1,k  ) * ym + p(i  ,j+1,k  ) * yp
144  |||  S  .     + p(i  ,j  ,k-1) * zm + p(i  ,j  ,k+1) * zp
145  |||  S  .     - rhs * ds * ds )
 :
148  |||  S      p(i,j,k) = p(i,j,k) + cnst_p * ( p_a - p0 )
149  ||V---     end do
150  |+----    end do
151  +-----   end do
```

図 4.3-54 SOR 法のコード

(2) Red Black 法

定義・参照の依存関係が存在することでベクトル化ができない DO ループに対して，依存関係が生じない順序で配列要素をアクセスすることによりベクトル化を可能にする手法の一つに Red Black 法が

ある. 図 4.3-55 に 3 次元の Red Black 法の概念図を示す. 配列要素を赤・黒と順番に色付けを行い, 赤の要素の計算と黒の要素の計算を行うループを分けることにより, ベクトル化を阻害する依存関係を排除できる.

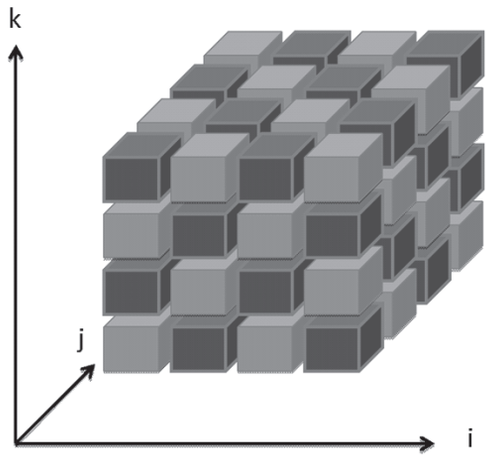

図 4.3-55 Red Black 法の概念図

3 次元の場合, 配列のアクセスは以下のように行われる(各次元のサイズが 8 の場合).

赤のループ:p(1,1,1),p(3,1,1),…,p(7,1,1),p(2,2,1),p(4,2,1),….

黒のループ:p(2,1,1),p(4,1,1),…,p(8,1,1),p(1,2,1),p(3,2,1),….

図 4.3-56 に Red Black 法のコードを示す. 最内 DO ループは上記に示した配列のアクセスのとおり増分 2 で繰り返される. 先頭要素から始まるループが赤のループ, 後続の先頭要素+1 から始まるループが黒のループに該当する. j, k の DO ループも増分 2 のアクセスになるように IF 文で処理を分岐している.

```
113 +-----> 　　do k = n_bnd + 1, n_bnd + n_cell
114 |+-----> 　　do j = n_bnd + 1, n_bnd + n_cell
115 ||　　　　　if(mod(k,2).eq.0.and.mod(j,2).eq.0.or.
116 ||　　+　　mod(k,2).eq.1.and.mod(j,2).eq.1) then
117 ||V---> 　　do i = n_bnd + 1, n_bnd + n_cell,2
 :
133 |||　　　　　p_a = c0
134 |||　　.　　* ( p(i-1,j  ,k  ) * xm + p(i+1,j  ,k  ) * xp
135 |||　　.　　  + p(i  ,j-1,k  ) * ym + p(i  ,j+1,k  ) * yp
136 |||　　.　　  + p(i  ,j  ,k-1) * zm + p(i  ,j  ,k+1) * zp
137 |||　　.　　  - rhs * ds * ds )
138 |||　　　　　p(i,j,k) = p(i,j,k) + cnst_p * ( p_a - p0 )
139 ||V---　　　end do
140 ||　　　　　else if(mod(k,2).eq.1.and.mod(j,2).eq.0.or.
141 ||　　+　　mod(k,2).eq.0.and.mod(j,2).eq.1) then
142 ||V---> 　　do i = n_bnd + 2, n_bnd + n_cell,2
 :
```

赤のループの実行

黒のループの実行

図 4.3-56 Red Black 法のコード

(3)　DO ループの一重化

図 4.3-56 に示したコードにおいて, 最内の DO ループがベクトル化の対象になるため, ベクトル長は

i の DO ループの半分のループ長になる(増分 2 でアクセスする)ため，十分なベクトル長が得られない．そこで，2 つのループを含めて 3 重ループを一重化することで，ベクトル長の拡大を図る．この例では，各次元に計算を行わない袖要素を含んでいるため，単純にDOループの一重化ができない．そこで予め計算を行う要素の場合を「1」，行わない要素の場合を「0」とするマスクテーブルを用意しておき，IF 文の判定で真になる場合のみ計算を実行するようにする．図 4.3-57 にDOループの一重化を行うコードを示す．コンパイラが配列pの依存関係を解析できないので，コンパイラ指示行NODEPを指定して，配列 p のアクセスに重なりがないことを明示する．

```
277 +-----> do icolor = 1, 2
278 |     !cdir nodep(p)
279 |V----> do ijk=icolor,max_cell*max_cell*max_cell,2
280 ||       if(matbl(ijk,1,1,icolor).eq.1.or.
281 ||    +     matbl(ijk,1,1,icolor+2).eq.1 ) then
 :
297 ||         p_a = c0
298 ||    .    * ( p(ijk-1,                1,1) * xm + p(ijk+1,                1,1) * xp
299 ||    .      + p(ijk-max_cell,         1,1) * ym + p(ijk+max_cell,         1,1) * yp
300 ||    .      + p(ijk-max_cell*max_cell,1,1) * zm + p(ijk+max_cell*max_cell,1,1) * zp
301 ||    .      - rhs * ds * ds )
 :
304 ||         p(ijk,1,1) = p(ijk,1,1) + cnst_p * ( p_a - p0 )
305 ||       endif
306 |V----   end do
307 +-----   end do
```

マスクテーブル配列 matbl の値が真の場合，ループの処理が実行される

図 4.3-57 マスクテーブルを使用するコード

(4) ADB の利用

SOR 法の計算は，4.3.8(1)で示しているように，ある要素の計算に近傍の 6 つの要素を使用するステンシル計算である．各要素は異なる繰り返しにおける計算で再び参照される．ON_ADB 指示行で，これらの要素を格納する配列 p を ADB に乗せることを指示する．これにより，1 回目のロード命令はメモリから行われるが，2 回目以降のロード命令は ADB から行われることになる．この結果，メモリのアクセス回数を削減することができ，処理効率の向上が期待される．図 4.3-58 に ON_ADB 指示行を指定したコードを示す．

```
277 +-----> do icolor = 1, 2
278 |     !cdir nodep(p)
279 |     !cdir on_adb(p)
280 |V----> do ijk=icolor,max_cell*max_cell*max_cell,2
281 ||       if(matbl(ijk,1,1,icolor).eq.1.or.
282 ||    +     matbl(ijk,1,1,icolor+2).eq.1 ) then
 :
298 ||         p_a = c0
299 ||    .    * ( p(ijk-1,                1,1) * xm + p(ijk+1,                1,1) * xp
300 ||    .      + p(ijk-max_cell,         1,1) * ym + p(ijk+max_cell,         1,1) * yp
301 ||    .      + p(ijk-max_cell*max_cell,1,1) * zm + p(ijk+max_cell*max_cell,1,1) * zp
302 ||    .      - rhs * ds * ds )
 :
305 ||         p(ijk,1,1) = p(ijk,1,1) + cnst_p * ( p_a - p0 )
306 ||       endif
307 |V----   end do
308 +-----   end do
```

図 4.3-58 ON_ADB 指示行の挿入

(5) 性能評価

SX-9 1CPU を用いて性能評価を行う. 評価を行うプログラムは以下のとおりである.

① SOR 法を最適化行わず記述したプログラム

② Red Black 法を用いてベクトル化を行ったプログラム

③ ②に対してマスクテーブルを用いてループの一重化を行ったプログラム

④ ③に対して再利用性の高い配列 p を ADB に乗せることを指示したプログラム

表 4.3-3 に, 各次元のループ長が 64 の場合の性能特性を示す.

表 4.3-3 性能特性

プログラム	ベクトル演算率(%)	平均ベクトル長	実効性能(MFLOPS)
①	51.86	64.0	189.5
②	90.11	54.4	970.1
③	99.59	255.9	8,247.3
④	99.59	255.9	18,940.6

①のプログラムは, ベクトル化できない DO ループ構造であるため, コンパイラは部分的にしかベクトル化を行わず, ベクトル演算率は 51.9%となり 200MFLOPS に満たない性能値となる. ②の 3 次元ループのままの Red Black 法のプログラムでは, ベクトル化は行えるものの, 平均ベクトル長が短いことと, ベクトルループを外側で何度も繰り返すため, 高い実効性能が得られない. ③の 3 次元ループに対してマスクテーブルを用いて一重化したプログラムでは, 平均ベクトル長が 255.9 となり, 8.3GFLOPS の性能値となる. さらに④の ADB に再利用性の高い配列要素を乗せるプログラムでは, メモリへのアクセス回数を削減することができ, ③と比較して 2 倍近い性能向上が得られる.

図 4.3-59 に, それぞれのプログラムで次元サイズを変更した場合の性能グラフを示す. すべてのケースにおいて, ④が最も高い性能であることが分かる. ベクトル型プロセッサの SX-9 では高いベクトル演算率, 十分な平均ベクトル長だけでなく, メモリ負荷を軽減する最適化が有効であることが示される. 特に再利用性の高い要素を ADB に載せることで実効性能の向上が期待される.

Red Black 法は, 元の SOR 法と計算順序が異なるため, 結果に差異が生じる. そのため SOR 法のような反復計算において, 収束までの繰り返し回数が元の SOR 法の計算より増える場合がある. 極端に収束回数が増加する場合は, Red Black 法による高速化の効果が相殺される可能性があるので注意されたい.

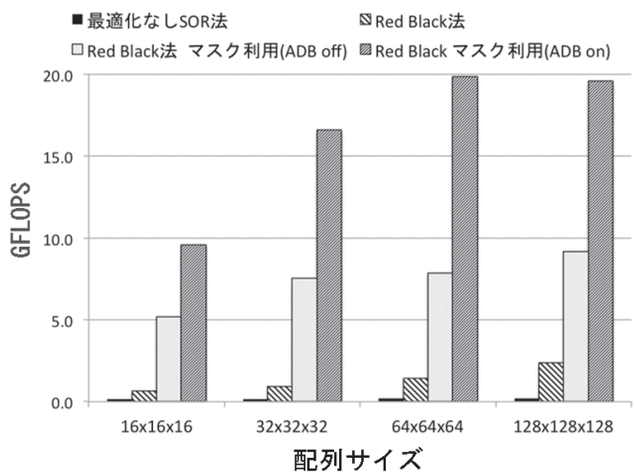

図 4.3-59 配列サイズと性能値