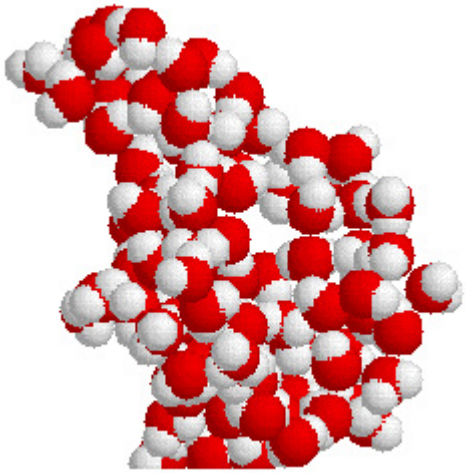
SX4 ベクトル化日記

増田 耕一(神戸大学大学院自然科学研究科環境科学専攻)
平田 和久(神戸大学大学院自然科学研究科地球惑星科学専攻)

△月×日

今日は私のプログラムをベクトル化する事になった。 ついでに、Fortran でのベクトル化は多いものの、C でのベクトル化は今まであまり なかったとの事でこのような体験談も書く事となった。そのプログラムとはMD(分子動力学)法を用いて水のクラスターアイスについて調べるものである。

********注1 分子動力学法とは?
具体的には、それぞれの粒子(この場合は、水分子)の運動方程式を数値積分して、時々刻々の粒子の位置と速度のデータを求め、これを解析して所要の量を計算して、系のマクロ(ミクロ)な性質、系の起こす現象の特徴等を調べる事が出来るコンピューターシミュレーションである。



MD 法を用いて作ったクラスター(水分子 100 個)

****

私の場合は、前 記した様にMD法を使って水分子100個程度のクラスターを作り、クラスター同士の衝突事象をシミュレーションしたいのだが、普通の計算機では1万回(クラスターが大体出来る位の計算)回すのにも大変な時間がかかってしまう。後々には水分子の数なども増やしてい

きたいので、ここはベクトル化をしてスパコンを用い計算を速くしようという事になったのである。
しかし、ベクトル化についてはおろか、スパコンを使ったこともないのでとりあえずは総合情報処理センターから、スパコン用の C 言語のプログラミングの手引を借りてきて、読んでみることにする。
それによるとベクトル化とは、

********注2 ベクトル化とは？
行列の行要素、列要素、あるいは対角要素など、規則的に並んだデータ列をベクトルデータという。また、ベクトルデータに対する演算を1度に実行することができる命令をベクトル命令という。これに対して、単一データ（スカラデータ）に対して演算を実行する命令をスカラ命令という。ここでスカラ命令をベクトル命令で置き換えることをベクトル化という。
例）

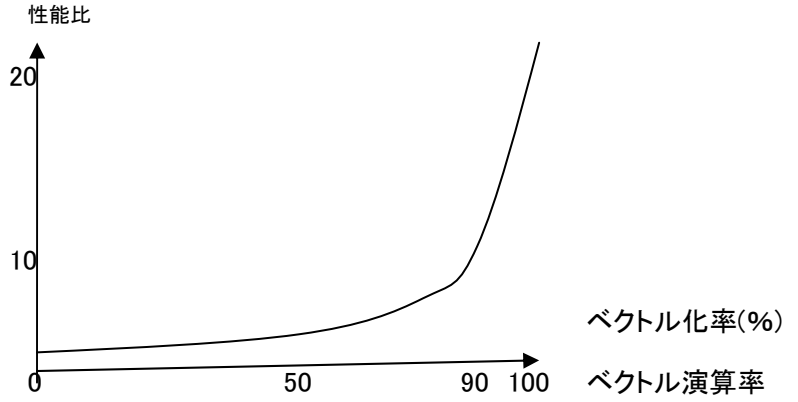
```
for(i=0; i<KOSUU; i++) {
a[i] = 2*i;
}
```

1）普通のスカラー命令だと上記の for ループは
a[0] = 0 ←1回ループが回る
a[1] = 2 ←1回ループが回る
a[2] = 4 ←1回ループが回る
……
……
a[KOSUU] = 2*KOSUU
と、個数回だけループが回るのに対して
2）ベクトル化すると、
a[0] = 0, a[1] = 2, a[2] = 4, a[3] = 6,
…… ←1回ループが回るだけで
…… 計算を実行
……
と、配列に要素を入れて、一度に計算を実行出来る
****

フムフム 普通は、配列をきって for 文で1回、1回まわしていくのをベクトル命令すれば、配列に一気に要素を全部ほうりこんでそれを一気に同時に計算してしまう。とりあえず「一気にやれ！型」だと理解する。さらに手引を読み進めると「アムダールの法則」というのがのっている。

********注3「アムダールの法則」とは？
ベクトル演算率、ベクトル化率には下図の様な関係（アムダールの法則）がある。ベクトル演算率、ベクトル化率が小さい時にはベクトル化による効果はほとんど期待できず、90%で効果が現れ始

め、95%以上で期待した性能が得られる。ベクトル演算率、ベクトル化率は95%以上を1つの目安にする。



なるほど95%か～。しかし、手引を読んでしまえば全てが出来たような気がするのが人の世の常である。とりあえずは友達のエヌ山に「俺も人生をベクトル化したいよ～」などとえらそうに言って、とりあえず今日は帰る事にした。

○ 月△日

さて、今日は実際にプログラムをベクトル化してみる事にした。 ここで私の行なうプログラムについて、概略は前述したが大まかな流れについて 説明すると

(1)実際に粒子間の距離を求め、粒子にかかる力を補間して出す。

↓

(2)粒子にかかる力から、粒子の次の位置を求める。

そして、(1)→(2)の過程を繰り返して、粒子の時々刻々の運動を見ていくプログラムである。 さて、ベクトル化であるが、まずは for 文のループの中で関数を呼び出している部分があるとベクトル化が出来ないという事でそこの手直しだ 。 呼び出している関数を for 文の中に書き出す、書き出す、書き出す、書き出す、書き出す…… (もしこの原稿をよみながら自分のプログラムベクトル化されてる方がいたら, ここで最後のページを参照してください) フーようやく全てを書き出した。コンパイルもうまくいったので、単純作業を終えた心地よさと共に一服。

******** 注4 どんな作業をしたのか？

```
double distance(double gensi[][3],double atom[][3],int j,int k) {
double kyori2;
kyori2 = (gensi[j][0]-atom[k][0])*(gensi[j][0]-atom[k][0])+
         (gensi[j][1]-atom[k][1])*(gensi[j][1]-atom[k][1])+
         (gensi[j][2]-atom[k][2])*(gensi[j][2]-atom[k][2]);
```

```
        return (kyori2);
}
for(j=0; j<KOSUU-1; j++) {
for(k=j+1; k<|KOSUU; k++) {
h1_h1= distance(h1,h1,j,k);
……
……
```

というのを、

```
for(j=0; j<KOSUU-1; j++) {
for(k=j+1;
k<KOSUU; k++){
h1_h1d = (h1[j][0]-h1[k][0])*(h1[j][0]-h1[k][0])+
        (h1[j][1]-h1[k][1])*(h1[j][1]-h1[k][1])+
        (h1[j][2]-h1[k][2])*(h1[j][2]-h1[k][2]);
        ……
        ……
```

という風に for ループの中に関数を書き出した。

****

次にそのプログラムを C-ANARYZER にかけてみる事にし た。これも総合情報処理センターから「C-ANARYZER／SX 利用の手引」というのを借りてきて読んでみると、これによってそのプログラムのベクトル化率や、関数の使用頻度といった情報がわかるらしい。早速その中の実行回数解析というのをやってみる。
実行回数解析というのは、マニュアルによると文の実行回数を数えるのと、文の実行回数と命令数をもとにプログラム、関数ごと、文ごとの実行コストを算出してくれる機能で、実行回数解析の処理は以下のように行なわれていくそうだ。

<STEP 1> ベクトル化／並列化情報を収集する。
<STEP 2> オブジェクトコード情報を収集する。
<STEP 3> プリプロセッサを実行して、測定のための測定用プログラムを作成する。
<STEP 4> C-ANARYZER／SX の実行時ルーチンをコンパイルしてオブジェクトコードを生成する。
<STEP 5> 測定用プログラムのオブジェクトと実行時ルーチンをリンクして、実行可能な測定用プログラムのオブジェクトコードを生成する。
<STEP 6> 測定用プログラムを実行して測定する。
<STEP 7> 測定結果を元にして解析リストを出力する。

うーん？？？まあとりあえずは、実行回数解析を行なう為マニュアルどおりに次の様に指定してコマンドラインから打ち込んでみる。

ca -ct -am -hvector,multi main.c force.c ……
ところが、何度やっても下の様に<STEP 5>で異常終了してしまう。
** C－ANALYZER／SXが起動されました。
<step1> ベクトル化／並列化情報を収集しています。
<step2> オブジェクトコード情報を収集しています。
<step3> プリプロセッサを実行しています。
<step4> 実行時ライブラリのオブジェクトコードを生成しています。
<step5> 測定用プログラムのオブジェクトコードを生成しています。
** 測定用プログラムのオブジェクトコードの生成処理が異常終了しました。
むぅ、しかしマニュアルと はありがたいもので「第１１章 困ったときは」というのがちゃんとある。
早速読んでみると、（3）<STEP 5>で異常終了の場合 というのまでちゃんとある。正にこの本はマニュアルのなかのマニュアル。そう、「KING OF MANUAL」だなあと感慨を抱きつつ読んでみると、<STEP 5>で異常終了する時には主に2つの間違いがありがちという事だ。
1 原始プログラムがきちんと指定されていない。
2 オプションの指定がただしくない。
どれどれとコマンドラインを確かめると、2つ間違いがあった。
（1）サブルーチンのプログラムの指定が1つ抜けている。
（2）コマンドラインのオプションにコンパイラオプションの -o と -lm の指定が されていない。
まるでマニュアルにあやつられている様な間違いっぷりだ。 これでは、まるで私が「KING OF MANUALMAN」ではないかと気づいて、 せめて次はもっと破天荒な間違いをしようと決心して正しく打ちなおしてみる。
ca -o -lm -ct -am -hvector,multi main.c force.c ……
** C－ANALYZER／SXが起動されました。
<step1>ベクトル化／並列化情報を収集しています。
<step2> オブジェクトコード情報を収集しています。
<step3> プリプロセッサを実行しています。
<step4> 実行時ライブラリのオブジェクトコードを生成しています。
<step5> 測定用プログラムのオブジェクトコ ードを生成しています。
<step6> 測定用プログラムを実行しています。
＊ PID＝15842
<step7> ポストプロセッサを実行しています。
** C－ANALYZER／SXが正常終了しました。

ふぅ、やればできるじゃないか。 結構、for 文への関数の書きだしは大変だったので、ベクトル化率50%は越えてるかな～と、淡い期待を抱きつつ、早速出来立ての、実行回数解析の結果のリストを見てみることにする。

```
*------------------------*
Summary List
*------------------------*
C-ANALYZER/SX Revision : Rev.062
Analyzed Date : Mon Dec 8 17:02:14 1997
ca Options : -ct -am Execution Information :
CPU Time = 0 : 00 ' 54 " 856
Total Execution Count = 53035811
Vector Information : Vectorization Ratio = 0.94
Parallel Information : Microtask Parallelization Ratio = 0.0

*-------------------------------------*
Function Summary List
*-------------------------------------*
```

Function	Atr.	Frequency	Exec Cost%	Vector Ratio
coulomb_force	GLOB	99	85.3	0.00
lj_force	GLOB	99	9.1	0.00
table	GLOB	1	2.0	0.00
main	GLOB	1	1.0	40.98

……

……

ベクトル化率、れ、れ、れ、れ、0.94？ 細かくリストを見てみると、table 以下のサブルーチンは Exec Cost（サブルーチンの実行コスト のプログラム全体の実行コ ストに対する比率）が 2.0 以下なので全体のベクトル化に対する寄与はほとんどない事がわかる。逆に上の2つのサブルーチン,coulomb_for ce（水素原子等にかかる力を計算する）とlj_force（酸素原子にかかる力を計算する）の2つだけで Exec Cost が足して 94.4 もあるのでこの2つのサブルーチンのベクトル化が重要な事がわかる。ちなみにこの ANARYZER をかけたプログラムは水分子が100個で計算を100回回すものなので、水分子の個数、計算の回数が増せば増すほど、この傾向は強まると思われる。

しかしながら、その重要な2つはサブルーチンベクトル化率がほとんどっていうか全くの0になっている。だが、先程行なわれた単純作業（関数の書き出し）の大半ももちろん、そのサブルーチンが占めていたはずだ。何故だ～ 夕闇の街に会いたくなる気分を、押し隠して、再びマニュアルを手にとる。 まずは、コンパイラオプションに

-pvctl,fullmsg

いうのをつけてみる。このオプションはベクトル化に関する全ての情報のリストをソースファイル名.O というファイルに出してくれるそうだ。そこで、-pvctl,fullmsg をつけて、コンパイルしてみると……出てきた出てきた。 そのうち、問題の2つのサブルーチン（coulomb_force,lj_force）の入ってるソースファイルの force2_tani.c の情報リストの force2_tani.O を見てみると、 ハッハーンなるほど、なるほど、謎はすべて解けた！

******** 注5 force2_tani.O はどんなもの？

SUPER-UX C/SX DATE: Mon Dec 8 17:41:49 1997 File Name : force2_tani.c

Vectorization Information List

Num.	Line Num.	Vectorizability Information
1	40	Partially vectorized by loop index j
		* Loop: Outer
		* Vectorized parts: 43
		* Iteration count: 99
		* Work vector: 1192 Bytes
2	43	Partially vectorized by loop index k
		* Loop: Innermost
		* Vectorized parts: 47--200,226
		* Assumed iteration count: 100
		*Index variables: k
		* Relative constants: CHARGE_M,CHARGE_H,j
		* Work vector: 21600 Byte
		* Loop transformed: Statement

214 DEP. assumed - value of relative constant is unknown : h1f 202

215 DEP. assumed - value of relative constant is unknown : h1f 203

……

……

上のリストの様に同じ for 文のループのなかでも、何行～何行がベクトル化していて、何行～何行がベクトル化してないか（その原因も）等細かい所まで教えてくれるリストである。

****

force2_tani.O によって、ベクトル化していない行には全て log が使われている事が、わかった。 再びマニュアルを読む、読む、読む。これかっ！ 次はコンパイラオプションに

-D_BUILTIN_

というのをつけてみる。このオプションは log などの関数を自動ベクトル化してくれるというものだ。これはいけるだろう、興奮をひた隠しながら C-ANARYZER にかけてみる。出てきたベクトル化率は……0.94！ ほわーい？何故だ～？ としばらく悩んだが、これは先程と同じ間違いで、ANARYZER をかける時にはコンパイラオプションの指定が必要なのを又、忘れていた。

ca -D_BUILTIN_ -o -lm -hvector,multi main.c ……
どうだろう？リストを見てみると、

```
*-------------------------*
Summary List
*-------------------------*
C-ANALYZER/SX Revision : Rev.062
Analyzed Date : Mon Dec 8 18:22:10 1997
ca Options : -ct -am
Execution Information : CPU Time = 0 : 00 ' 10 " 997
Total Execution Count = 53035811
Vector Information : Vectorization Ratio = 79.31
Parallel Information : Microtask Parallelization Ratio = 0.0
*-------------------------------------*
Function Summary List
*-------------------------------------*
```

Function	Atr.	Frequency	Exec Cost%	Vector Ratio%
coulomb_force	GLOB	99	82.9	81.33
lj_force	GLOB	99	10.1	99.79
table	GLOB	1	2.6	0.00
main	GLOB	1	1.3	42.13

……

……

うひゃー、上記のとおり全体のベクトル化率は 79.31%となってる！ しかも、注目すべき2つのサブルーチンも 81.33%,99.79%と、なかなかのものだ。水分子の個数、計算の回数が増えれば、全体のベクトル化率も更に上がるだろう。 しかし、coulomb_force も lj¥_force も大体同じ式なのに、結構ベクトル化率が違うのは、なぜだろう？ まあいい、とりあえず今日はもう遅いので帰るとしよう。 まだ俺と奴-スパコン-の戦いは、始まったばかりだ。又明日頑張ろう。

□月×日

今日は昨日の成果をためしてみる事にする。 あらかじめ、研究室で使ってる UNIX のパソコン（OS:FreeBSD, cpu:pentium）で計算を しておいて、同じ初期条件（水分子の位置の座標等）を使って、スパコンの方でも、同じ様に計算を回してみる。計算を回しながら、今までの長い道のりに思いを馳せ、今日でこの日記も終りかな～と感慨にふける。むっ？終ったようだ。流石に速い気がする。（実際は水分子100個で計算も100回位なら、研究室で使ってるパソコンでもそんなに速さは変わらないと思う。）早速、スパコンを使って出された値（100回回した後での水分子の位置や、運動エネルギーの値等）を先程研究室のパソコンで出しておいた値と比較してみる。おや？違あう～違あう～違あう～違あう～ ち・が・う・ぞ・お～ 危うくメルトダウンしそうな気分を押える為にちょっと一服。さて冷静な目でプログラムを見直してみるが、なかなか原因が判りそうにもないので、デバッグをかけてみる事にした。デバッガには dbx を使う。（詳しい使い方は man コマンドで見て下さい。） パソコンの方での計算にもデバッグをかけ、プログラム中の変数、配列について同じ数値が入っているかスパコンの方と比べてみる。 すると、またしても原子にかかる力の計算の所でおかしくなっている様だ。そこで、ベクトル化が原因かどうかを調べるために、自動ベクトル化を行なわない様にするオプションの

-hnovector

を付けてコンパイルして先程と同じ様にデバッグをかけて、配列や変数に入っている値を調べてみる。すると、見事にスパコンと研究室の値は一致した。って事はやっぱり、ベクトル化によって、配列や変数に入る値が変わってる事になる。 むぅ？変数か……変数ねぇ…… 私のプログラムにおいては、最終的に求める原子にかかる力については、配列に入れているが、それを求めるまでの計算中の原子間の距離等については、変数にしている。普通の計算中では for 文で 1 回、1 回まわるために計算中の、変数についても 1 回、1 回違う値が入る様になっているが、ベクトル化されると、for 文を一気に同時に計算してしまうため、変数にうまく数値が入っていないのだろうか？大体、文章で書いているうちに、自分でも何を言っているか、わけがわからなくなってきた。そこで、と りあえずは下に問題のプログラムを簡略化したのを書いて、実際に自分のプログラムの変数は、配列に書き直す事にした。

******** 注6 どんな事をしたのか？

例）

```
for(i=0; i<KOSUU; i++) {
a = i+1;
b = i*2;
c[i] = a+b;
}
```

上のプログラムの様に、計算中の数値は変数に入れていたのを、

例2）

```
for(i=0; i<KOSUU; i++) {
a[i] = i+1;
b[i] = i*2;
c[i] = a[i]+b[i];
}
```

という風に、計算中の数値も配列に入れ直した。

****

注6の様な短いプログラムなら良いのだが、実際のプログラムは、もっともっと長いので、これもひたすら、配列をきる、きる、きる、きる、きる、…… ふー疲れた。こういう作業をず一っとやっていると、だんだんこれで絶対いけてる様な気がしてくるから不思議なものだ。早速、デバッカにかけて研究室のパソコンの方の値と比べてみる。 ムフッよしよし！ピッタリー緒の値だ。しばしいい気分を満喫する。今度は肝心のベクトル化率の方を C-ANARYZER にかけて調べてみる。どうだろう？よぉぉぉぉし！全体のベクトル化率は、81.81%、力を計算する2つのサブルーチンの coulomb_force（水素等の原子にかかる力を計算する）は、83.74%で lj_fo rce（酸素原子にかかる力を計算する）は 99.54%である。いやーたいしたものだねぇー、とりあえず自分で自分をほめておく。しかし課題である 95%にはもう一声、何かが必要なようだ。まあそれは明日への課題としておこう。 マニュアルを読み、読み今日は帰ることにする。

×月〇日

さあて、今日も始めよう。きのう新しく書き直した問題の2つのサブルーチン（coulomb_force, lj_force）の入ってるソースファイルの force2_tani.c の情報リストの force2_tani.O を見てみる事にする。そして、実際どの部分がベクトル化されていないか調べてみた。すると以下の様な部分が、ベクトル化されていない事がわかった。

******** 注7 どんな部分？

例）

```
for(i=0; i<KOSUU; i++){
for(j=0; j<KOSUU; j++){
a[i][j] = i+j;
x[i] = x[i]+i;
x[j] = x[j]+j;
}
}
```

上記のプログラム例でいうと、a[1][0] の場合、i=1,j=0 の時のループが回るだけで入る数値がきまる. この様に 1 回のループで入る数値が決定する配列はベクトル化される。 x[1] の場合、i=1,j=0 のループだけでは入る数値がきまらず、ループ全体が回らないと配列に入る数値は決まらない.
この様に1回ループを回すだけでは入る数値が決まらない配列は、ベクトル化されない。

****

この様な部分があるせいで、coulomb_force のベクトル化率は、83.74%となっているんだろう。こういう時は、マニュアルを読むしかないのかなあと ぼやきつつもとりあえず読む、読む、読む、読む。うーんこれかなというのを見つける。以下に、マニュアルのその部分を書き出してみる。

１：データ依存関係

　あるループの繰り返しで計算されたデータが、ほかの繰り返しで使用される場合、そのループはデータ依存関係を持つといい、ベクトル化されない場合がある。

２：条件ベクトル化

　配列の依存関係がベクトル化可能かどうかがコンパイル時に不明の場合、そのループに対し依存関係がベクトル化可能としたコードとベクトル化不可能としたコードの2つを作っておき、実行時に、依存関係を定める変数の値をみて、 2つのコードのどちらを実行するかを選択する方法である。

３：ベクトル化指示オプション #pragma vdir nodep

　同一配列の要素がループ中に2度以上出現する場合、それらの定義・参照関係がベクトル化後も正しく保たれなければならない。この条件は、ベクトル化によって文の実行順序が変わる為、必要となる。ベクトル化指示行は、ループの直前に指定する。直後のループについて、ベクトル化してもデータの定義・参照関係が正しく保たれることを指示する。

？？？？うーんなんでマニュアルの日本語は読み難いんだろう、なんでモルツは泡までうまいんだろう？なんで…… いかんいかん、違う世界に行ってしまうところだった。とりあえず、注7のよう

なループは1番目のデータ依存関係になっているのだろうという事で、マニュアルどおりに、ベクトル化指示オプションの#pragma vdir nodep というのをループにつけてみる事にした。

******** 注8 どんな風につけるの？

```
for(i=0; i<KOSUU; i++) {
for(j=0; j<KOSUU; j++){
a[i][j] = i+j;
x[i] = x[i]+i;
x[j] = x[j]+j;
}
}
```

というプログラムにベクトル化指示オプションの#pragma vdir nodep をつける

```
for(i=0;i<KOSUU;i++){
#pragma vdir nodep
for(j=0;j<KOSUU;j++){
a[i][j]=i+j;
x[i]=x[i]+i;
x[j]=x[j]+j;
}
}
```

****

さてさて、うまくベクトル化されているだろうか？早速 C-ANARYZER にかけてみる。おっほーい！すばらしい、coulomb_force のベクトル化率は
92.37%となっている 。とうとうcoulomb_force のベクトル化率が90%を越えた！しかし、全体のベクトル化率は、88%位でやはりもう一声足りないようだ。 今日中にこれも 90¥%を越えてやるぅぅ。早速 coulomb_force の入ってるソースファイルの情報リストの force2_tani.O を見る事から始める。92.37%までにはなったがベクトル化されていない約8%の部 分を見つけようという訳だ。見てみると最後までベクトル化されないしつこい油汚れの様な部分は、以下のような、プログラムである事がわかった。

********* 注9 どんな部分？

```
for(i=0;i<KOSUU;i++){
#pragma vdir nodep
for(j=0;j<KOSUU;j++){
```

```
a[i][j]=i+j;
b[i][j]=2*a[i][j];
x[i]=x[i]+a[i][j]+b[i][j];←この部分！
x[j]=x[j]-a[i][j]-b[i][j]
}
}
```

****

なんでやろう？しかし、前々から書いてた様に同じような計算をしている lj_force ではず一っと99.54%とドカベン並の高打率を誇っている。同じ様な計算のはずなのに ……しかし見比べていくと、問題のベクトル化されない部分の計算が少し違うのに気づいた。coulomb_force の方では問題の配列に対して、幾つかの配列が足されているのに対して lj_force では問題の配列に1つしか配列が足されていないのだ。そこで、coulomb_force の方も足される配列を1つにまとめてみた。文章で書くとわかりにくいので、以下に例としてあげる。

********* 注10 何をしたのか

```
for(i=0;i<KOSUU;i++){
#pragma vdir nodep
for(j=0;j<KOSUU;j++){
a[i][j]=i+j;
b[i][j]=2*a[i][j];
x[i]=x[i]+a[i][j]+b[i][j];←問題の配列に幾つかの配列が足されている
x[j]=x[j]-a[i][j]-b[i][j]
}
}
```

こんなプログラムを

```
for(i=0;i<KOSUU;i++){
#pragma vdir nodep

for(j=0;j<KOSUU;j++){
a[i][j]=i+j;
b[i][j]=2*a[i][j];
c[i][j]=a[i][j]+b[i][j]
```

```
x[i]=x[i]+c[i][j];←問題の配列には1つだけ配列を足す
x[j]=x[j]-a[i][j]-b[i][j]
}
}
```

****

さて、これでかわっただろうか？再び、C-ANARYZER にかけてみる。

```
*-------------------------*
Summary List
*-------------------------*
C-ANALYZER/SX Revision : Rev.062
AnalyzedDate : Fri Nov 28 15:11:53 1997
ca Options : -ct -am
Execution Information : CPU Time = 0 : 00 ' 10 " 100
Total Execution Count = 61856811
Vector Information : Vectorization Ratio = 95.27 %
Parallel Information : Microtask
Parallelization Ratio = 0.0 %
*-------------------------------------*
Function Summary List
*-------------------------------------*
```

Function	Atr.	Frequency	Exec Cost%	Vector Ratio%
coulomb_force	GLOB	99	82.6	99.95
lj_force	GLOB	99	11.2	99.54
table	GLOB	1	2.3	0.00

……

……

……

おおぉぉぉぉぉぉ、すごいぞおおおぉぉぉぉぉ！ とうとう全体のベクトル化も 95%を越えた！やれば出来るじゃないか～。とりあえず今日はもう帰ろう、今夜は祝杯だぁ！

〇月〇日
祝杯の夜は明けたので、今までやってきたことを残った謎と共にまとめてみよう。

1. ループ内では、関数の呼び出しは行なわない。
　これは、まちがいなくあってる。
2. log などを使ってる場合には、それに必要なコンパイラオプションをつける。
　特に、-D_BUILTIN_は自動ベクトル化に有効なオプションである。他にも、場合場合によって必要なオプションがあると思うので、 いろいろためしてみて下さい。
3. ループ内での変数は配列になおす。
　これは、いつでもあってるとはいえないと思う。変数のままやっても、ベクトル化してしかも結果もおかしくない場合もあると思うが、うまくいかなかったり思ったようにベクトル化率があがらない時には、やってみる価値はあると思う。
4. データ依存関係を持つ配列を含むループには、 ベクトル化指示オプションの #pragma vdir nodep をつける。
　ベクトル化指示オプションや、#pragma vdir nodep については、色々あるので、これまた、いろいろためしてみて下さい。
5. データ依存関係を持つ配列に対して、幾つも配列や変数を足さない。
　データ依存関係を持つ配列に幾つかの配列や変数が足されていて、#pragma vdir nodep をつけてもベクトル化されない場合は、それら足された り、引かれている部分を1つにまとめたらベクトル化されると思う。

まだまだ、わからないところも多いが、ベクトル化率が 95%を越えるという目標は達成されたのでこの日記も終らせたいと思う。とりあえずは3日坊主で終らなくて良かったと思う今日このごろである。

追記
後日,判明したことを幾つか追加させてもらう。
1. ベクトル化をやるにあたって、最初の方で行なったループの中での関数の呼び出しについてですが、私は、ループの中に実際に関数を1つ1つ書き出していったが、あの作業は

-copp -Wo,"-e7"

という、コンパイラオプションで自動にしてくれるらしい。……あの作業はなんだったのだろうか.
2. スパコンでのデバッグはインタラクティブで行なったが、実際に計算を走らせる時は,ジョブを使

って計算してください。

3. 速報！ スパコンは本当に速いのか？

こんな作業をしてきたぶんだけ、本当にスパコンは速いのだろうか？文句ばっかり言って大したことないんちゃうん？という疑問は、当然頭をよぎる。 そこで実際に研究室のパソコンと同じ計算をさせて、実行速度を競ってみる事にした。その計算とは、私の MD のプログラムで 100 個の水分子で 10000 回計算を回すというものだ。

研究室の UNIX……OS： FreeBSD,cpu：pentium,クロック数：120MHz

スパコン ……SX-4/2C(zeus)

結果は研究室のパソコンは、3時間30分位かかるのに対して、スパコンの方は ……10分位！！！ いやいや、ベクトル化して良かった。

最後に参考文献を載せる。本文中でのベクトル化についてのマニュアルからの引用及び、「アムダールの法則」の図などは、全て[1],[2]から引用している。分子動力学について興味を持たれた方は、[3]を参考して下さい。

参考文献

[1]SX システムソフトウェア SUPER-UX C プログラミング の手引 (NEC, 1995)

[2]SX システムソフトウェア SUPER-UX C-ANALYZER/SX 利用の手引 (NEC, 1995)

[3]Koichi Masuda, Junko Takahasi, and Tadasi Mukai ``Sticking Probability and Mobility of a Hydrogen Atom on Icy Mantle of Dust Grains'' *Astron.Astrophys.*(1998 in press)