3.2.3. 流体解析から見る Fortran90 の構造体性能評価

宇宙航空研究開発機構　　高木 亮治

1. はじめに

Fortran90 では、構造体、動的配列、ポインターなど様々な便利な機能が追加され、ユーザーがプログラムを作成する際に選択の幅が広がりより便利になった。一方で、実際のアプリケーションプログラムを開発する際には、解析対象となる物理現象を記述する数学モデルやそれらを解析するための計算手法が内包する階層構造を反映したプログラムを作成できるかどうかは一つの重要な観点であると考えられる。Fortran90 で導入された構造体、モジュール、動的配列などの機能はデータ構造の階層化を容易に実現できるものとして非常に便利な機能であるが、従来の静的な配列などに比べて性能面でのデメリットが考えられる。ここでは広く一般の科学技術計算プログラムを作成する際の指針を得ることを最終的な目標として、例えば流体解析プログラムを実装する際に、性能も考慮するとどのような機能を利用してプログラムを実装すれば良いか、逆にこういう実装は駄目ということを確認する。

2. 流体解析プログラム（複合格子法）

流体解析プログラムはデータ構造の観点で整理すると幾つかの種類に分類できるが、ここではマルチブロック構造格子法を対象とする。マルチブロック構造格子は図 1 で示すように、構造格子を 1 つのブロックとして計算空間全体を複数のブロックで構成する手法である（図 2 に例を示す）。特徴として①各ブロックのサイズ、形状はばらばらである、②各ブロック間は非構造的に接合する、③各ブロック内は単一の構造格子である、といった点が挙げられる。マルチブロック構造格子法ではデータ構造に階層構造（①計算空間が複数のブロックで構成される、②各ブロックは構造格子となる、③各格子点で物理量が定義される）が存在する。

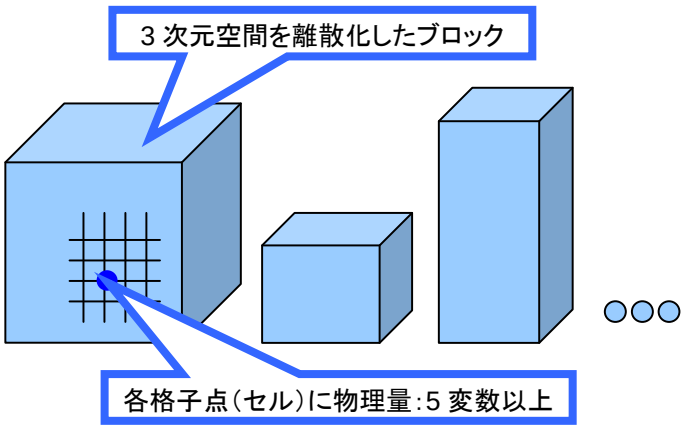

図 1.　マルチブロック構造格子のコンセプト

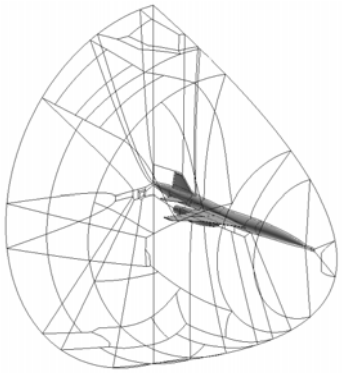

図 1.　マルチブロック構造格子の例

特に注意すべき点として各ブロックのサイズ、形状はばらばらであり、これらは解析対象の形状もしくは計算格子作成の都合で決められること、また、汎用的なプログラムとしたいという理由により配列の大きさ、形状は実行時に指定したいという要求があること、である。

3. プログラム概要

3.1 概要

流体解析プログラムでは 3 次元構造格子（I,J,K）で定義されるデータに対して I、J、K 方向それぞれのスイープを実施する。その際にスイープ方向の隣接点を利用した流束(f)の計算を実施し、元の配列(q）の値を更新する。ここでは簡単な場合として 1 次精度の流束（対流項、SHUS スキーム）の計算を考える。高次精度の流束の場合、ステンシル（計算に利用する隣接点の個数）が増加する。ブロック数は 21、各ブロックは 51x51x51 とする。プログラムの概略を以下に示す。

```
I 方向の f および dq の計算
 do j,k,n
 do i=1,imax-1
    f(i,j,k,n) = ...
 enddo
 do j,k,n
 do i=2,imax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i-1,j,k,n)
 enddo

J 方向の f および dq の計算
 do k,i,n
 do j=1,jmax-1
    f(i,j,k,n) = ...
 enddo
 do k,i,n
 do j=2,jmax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i,j-1,k,n)
 enddo

K 方向の f および dq の計算
 do i,j,n
 do k=1,kmax-1
    f(i,j,k,n) = ...
 enddo
 do i,j,n
 do k=2,kmax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i,j,k-1,n)
 enddo

q の更新
do i,j,k,n
   q(i,j,k,n) = q(i,j,k,n) + dq(i,j,k,n)
enddo
```

3.2 データ構造

マルチブロック構造格子を実装するデータ構造として以下の 7 パターンを考えた。

1 構造体を用いて階層構造を表現する場合：ブロック(M)/格子点(I,J,K)/物理量(N)

A：blk(:)%q(:,:,:,:)、インデックスは I,J,K,N（I=J=K=51,N=5）、動的配列%動的配列

B：blk(:)%q(:,:,:,:)、インデックスは N,I,J,K（I=J=K=51,N=5）、動的配列%動的配列

C：blk(:)%q(I,J,K,N)、動的配列%静的配列

D：blk(M)%q(I,J,K,N)、静的配列%静的配列

E：blk(:)%cell(:,:,:)%q(:)、動的配列%動的配列%動的配列

F：blk(:)%cell(:,:,:)%q(N)、動的配列%動的配列%静的配列

2 通常の配列を用いて階層構造を持たない場合：

G：q(I,J,K,N,M)、（I=J=K=51, N=5, M=21）、静的配列

3.3 ループ構造

計算のループとして表 1 で示す 4 種類（カーネル）を考えた。

表 1. カーネル（ループ構造）

カーネル 0	カーネル 1	カーネル 2	カーネル 3
スイープ方向にかかわらず、ループネストの順番を変えない。（インデックスの順番通り） do ndir=1,3 do n=1,5 do k のループ do j のループ do i のループ f の計算 enddo do n=1,5 do k のループ do j のループ do i のループ dq の計算 enddo enddo do n=1,5 do k のループ do j のループ do i のループ q の更新 enddo	カーネル 0 と同じだが、n のループを最内に。 do ndir=1,3 do k のループ do j のループ do i のループ do n＝1,5 f の計算 enddo do k のループ do j のループ do i のループ do n=1,5 dq の計算 enddo enddo do k のループ do j のループ do i のループ do n=1,5 q の更新 enddo	スイープの方向に合わせてループネストの順番を変える。 I 方向 do k=1,kmax do j=1,jmax do i=1,imax-1 f の計算 enddo do n=1,5 do i=2,imax-1 dq の計算 enddo enddo J 方向 do k=1,kmax do i=1,imax do j=1,jmax-1 f の計算 enddo do n=1,5 do j=2,jmax-1 dq の計算 enddo enddo K 方向 do j=1,jmax do i=1,imax do k=1,kmax-1 f の計算 enddo do n=1,5 do k=2,kmax-1 dq の計算 enddo enddo do n=1,5 do k=1,kmax do j=1,jmax do i=1,imax q の更新 enddo	カーネル 2 で n=1,5 のループを最内とする。 I 方向 do k=1,kmax do j=1,jmax do i=1,imax-1 f の計算 enddo do i=2,imax-1 do n=1,5 dq の計算 enddo enddo J 方向 do k=1,kmax do i=1,imax do j=1,jmax-1 f の計算 enddo do j=2,jmax-1 do n=1,5 dq の計算 enddo enddo K 方向 do j=1,jmax do i=1,imax do k=1,kmax-1 f の計算 enddo do k=2,kmax-1 do n=1,5 dq の計算 enddo enddo do k=1,kmax do j=1,jmax do i=1,imax do n=1,5 q の更新 enddo

4. 計測結果

4.1 計測環境

① PC（Intel Core2 Extreme (QX9650)、3.0GHz、X38、DDR3-1333）
　コンパイルオプション：-O5

② PC（pointer 属性を allocatable 属性に変更）
　コンパイルオプション：-O5

③ FX1（SPARC 64VII, 2.5GHz, visIMPACT なし）
　コンパイルオプション：-O5

④ FX1（SPARC 64VII, 2.5GHz, visIMPACT）
　コンパイルオプション：-O5 JAXA デフォルト

⑤ FX1
　コンパイルオプション：-O5 JAXA デフォルト –x200 –Karray_private –Kprefetch_strong,noalias=s –Ncalleralloc=2 –Kauto –Kthreadsafe –KNOFILTLD

⑥ FX1（pointer 属性を allocatable 属性に変更）
　コンパイルオプション：-O5 JAXA デフォルト –x200 –Karray_private –Kprefetch_strong,noalias=s –Ncalleralloc=2 –Kauto –Kthreadsafe –KNOFILTLD

JAXA デフォルト：-Kfast,ocl –Nrtrap –Ktl_trt –Kimpact –Kmfunc=2 –Kpreex,prefetch_model=FX1

4.2 計測結果

計測結果を表 2 に示す。PC に関しては連続して 10 回計測を行いその平均値を計測結果とした。FX1 に関しては 1 回の計測値である。念のため手動で数回計測を行ったが、計測値はその都度若干変わったが、傾向としては大きくは変動しなかった。

4.3 考察

・データ構造
　・A と G および B と G の比較から構造体を使っても大きなデメリットはない。
　・A と C、D の比較から動的配列よりも静的配列のほうが有利。
　・PC では pointer（①PC）と allocatable（②PC）の違いは小さい。（allocatable が若干有利？）
　・FX1 では allocatable（⑥FX1）よりも pointer（⑤FX1）が良い。（本計測を行った時点では富士通コンパイラの allocatable 属性に対する最適化が不十分であったためである）
　・C,D,G が他に比べて速い（そうでもないケースもあるが）ことから静的配列が有利。
　・E が最悪、F も全般的に駄目。でも PC では F の 3 が最速となった。原因は不明。
・カーネル
　・2,3 は駄目。
　・0 と 1 は大きな差がない（コンパイルリストを確認するとカーネル 0 では do n=1,5（最外側）のループで自動並列化されていた。ディレクティブを挿入することで do n=1,5 を除いて内側のループで自動並列化を行ったが、性能的には大差はなかった。）
・全般
　・PC と FX1 の比較では PC は比較的ケース間の性能差が小さいが FX1 は大きい。
　・FX1 のオプションの効果が大。

表 2. 計測結果

データ構造	カーネル	時間［sec］					
		①PC	②PC	③FX1	④FX1	⑤FX1	⑥FX1
A	0	1.052	0.924	1.171	1.114	0.377	4.543
	1	0.947	0.892	1.220	1.271	0.361	4.315
	2	1.380	1.205	2.211	2.382	2.098	7.786
	3	1.289	1.186	2.211	2.882	2.101	7.466
B	0	1.676	1.546	1.447	1.404	0.497	4.589
	1	0.915	0.860	1.241	1.297	0.346	4.173
	2	1.151	1.092	1.767	1.809	1.746	5.547
	3	0.789	0.869	1.774	1.751	1.746	5.243
C	0	0.810	0.830	0.856	0.789	0.296	0.879
	1	0.848	0.828	0.831	0.765	0.271	0.957
	2	0.949	1.051	2.108	2.246	2.015	2.094
	3	0.982	1.025	1.982	2.251	1.978	2.021
D	0	0.858	0.734	0.971	0.788	0.877	0.877
	1	0.820	0.857	0.974	0.771	0.936	0.935
	2	1.031	0.973	2.262	2.216	2.080	2.084
	3	0.989	0.983	2.341	2.244	2.085	2.086
E	0	4.703	4.594	13.268	13.249	6.808	15.692
	1	1.760	1.606	5.154	5.079	3.401	7.575
	2	2.829	2.556	4.052	6.092	3.593	8.590
	3	1.564	1.546	4.030	4.101	3.592	6.362
F	0	1.899	1.913	1.690	1.213	1.216	1.210
	1	0.890	0.881	1.089	1.011	1.013	0.987
	2	0.733	0.645	1.017	1.005	0.992	1.358
	3	0.662	0.626	1.009	1.005	0.982	0.923
G	0	0.795	0.783	0.984	0.792	0.312	0.296
	1	0.812	0.792	0.950	0.766	0.304	0.273
	2	1.053	1.042	2.347	2.229	2.089	2.081
	3	1.036	1.042	2.333	2.229	2.252	2.254

	最短時間より倍以上遅いケース

5. まとめ

・静的配列がやはり性能的に有利だが、構造体もループの回し方に気をつければそれほど大きなペナルティーはない。

・ループは常に同じ順番（インデックスの順番）でまわすべき。

以上