Wisteria/BDEC-01（Odyssey）における OpenMP によるプログラミング入門（その 2）

中島研吾[a]，笠井良浩[b]，坂口吉生[b]

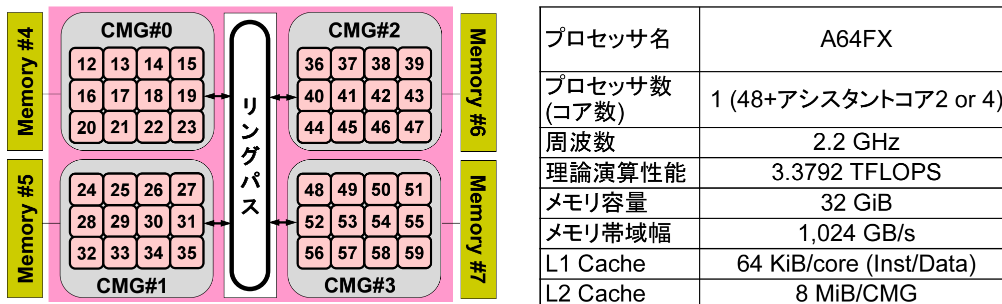
(a)東京大学情報基盤センター，(b)富士通株式会社

本稿では，前号（その 1）に引き続き，Wisteria/BDEC-01（Odyssey）上での最適化について，説明する。プログラム類は Wisteria/BDEC-01 の **`/work/share/ompw/ompw.tar`** から取得できるので，興味ある方は試してみられると良い。

1. Odyssey での実行

（1）A64FX プロセッサ

Odyssey の計算ノードは，A64FX プロセッサであり，図 1 に示すように 48 個のコアから構成されている。12 コアが CMG（Core Memory Group）を構成しており，各 CMG に HBM2 による高速メモリ（8GiB）が搭載されている [1]。いわゆる NUMA アーキテクチャであるため，OpenMP による並列化は各 CMG 単位で実施し，OpenMP/MPI ハイブリッド並列プログラミングでは，ノード当たり 4 つの MPI プロセスを立ち上げる，ことが推奨されているが，本稿では敢えて，1 ノード，48 コアに対して OpenMP 並列化を適用する。図 1 に示すように，メモリ，コアの番号は各 CMG において，CMG#0（メモリ：#4，コア：#12-#23），CMG#1（メモリ：#5，コア：#24-#35），CMG#2（メモリ：#6，コア：#36-#47），CMG#3（メモリ：#7，コア：#48-#59）



プロセッサ名	A64FX
プロセッサ数 (コア数)	1 (48+アシスタントコア2 or 4)
周波数	2.2 GHz
理論演算性能	3.3792 TFLOPS
メモリ容量	32 GiB
メモリ帯域幅	1,024 GB/s
L1 Cache	64 KiB/core (Inst/Data)
L2 Cache	8 MiB/CMG

図 1　A64FX プロセッサの構成と諸元［1］

（2）C コンパイラ

Odyssey では C コンパイラとして「trad モード」，「clang モード」の 2 種類がある（表 1 参照）。デフォルトは trad モードである。使用する機能や性能によって使いわける必要があるが，本稿のケースの場合，問題規模が比較的小さい場合は「clang モード」の方が高速であったため，上記のサンプルプログラムでは基本的に「clang モード」を使用している。後述するように，最適化を施し，問題規模を大きくした場合には，同等，もしくは「trad モード」の方が高速である場合もある。

表1　C コンパイラ：2 種類のモード

trad モード （-Nnoclang オプション） （デフォルト）	• 「京」および PRIMEHPC FX100 以前のシステム向け富士通コンパイラをベースとする。 • trad モードは，従来の富士通コンパイラとの互換を重視する場合に適している。 • サポートしている仕様は, C89/C99/C11, OpenMP 3.1/OpenMP 4.5（一部） • オプション省略時（デフォルト）は，-Nnoclang オプション適用
clang モード （-Nclang オプション）	• オープンソースソフトウェアである Clang/LLVM コンパイラをベースとする。 • clang モードは，最新言語仕様を使用したプログラムや，オープンソースソフトウェアを翻訳する場合に適している。 • サポートしている仕様は，C89/C99/C11，　OpenMP 4.5/OpenMP 5.0（一部)

（3）コンパイル・実行

まず，図 1 の CMG＃0 のみ 12 コアを使ったケースを実行してみよう。詳細は［2］をご覧いただくとして，ここでは簡単に概要について紹介する。

```
>$ cp /work/share/ompw/ompw.tar .
>$ tar xvf ompw.tar
>$ cd ompw            ! (このディレクトリを以下<$O-ompw>と呼ぶ)
>$ module load fj     ! (ログインしたら必ずこれをタイプする)
>$ make -f makec      ! (Fortan であれば make -f makef)
>$ cd run

"INPUT.DAT", "c12.sh", "f12.sh"を修正する

>$ pjsub c12.sh       ! (Fortran であれば psub f12.sh)
```

図 2　ファイルのコピー，コンパイル，実行［2］

C 言語の場合，**make -f makc-org** とすると（2）で述べた「trad」モードになる。コンパイルオプションは，下記のようになっている。:

- **makec：-Kfast,openmp -Nclang -msve-vector-bits=512**
- **makec-org：-Kfast,openmp**

-msve-vector-bits=512 は，SIMD 長を 512 に固定するオプションである。Fortran では SIMD 長は 512 に固定されているが，clang では可変長 SIMD がデフォルトとなっている。SIMD 長を 512 に固定することによって，PCG 法の計算の大部分を占める行列ベクトル積部が高速化されたため，本オプションを採用することとしている。

図 3 は実行制御ファイル **<$O-ompw>/run/INPUT.DAT** の例である。ここではまず，メッシュ数=2,097,152（$=128^3$）の場合を実施する。

図 4 はジョブスクリプト **<$O-ompw>/run/c12.sh** の例である。Fortran 用の **<$O-ompw>/run/f12.sh** も **solc0** が **solf0** となっている以外は同様である。NUMA アーキテクチャにおいて，できるだけローカルなメモリを使用して効率良く計算を実行するために［3］,

numactl -l を使用する。また使用する CMG を指定するために**-C** でコア番号，**-m** でメモリ番号（図 1）を指定することもできるが，結果的に計算時間への影響はほとんどない［2］。環境変数 **XOS_MMM_L_PAGING_POLICY** については複数 CMG 対して OpenMP を適用する場合には「**demand**」を指定することが推奨されている。詳細は［2］を参照されたい。

```
128 128 128                          NX NY NZ        !X,Y,Z 方向のメッシュ数
1.00e-0  1.00e-00 1.00e-00           DX/DY/DZ        !各メッシュの 3 辺の長さ
1.0e-08                              EPSICCG !収束判定値（10-8 を使用）
```

図 3　メッシュ数=128^3 の場合の実行制御ファイル**<$O-ompw>/run/INPUT.DAT** の例［2］

```
#!/bin/sh
#PJM -N “c12”                                !ジョブ名称（省略可）
#PJM -L rscgrp=debug-o               !実行キュー名（Resource Group）
#PJM -L node=1                               !ノード数（原則＝1）
#PJM --omp thread=12                         !スレッド数（1-48）
#PJM -L elapse=00:15:00                      !実行時間
#PJM -g gxYZ                                 !グループ名（財布）
#PJM -j
#PJM -e err                                  !エラー出力ファイル
#PJM -o c12.lst                              !標準出力ファイル

module load fj
export OMP_NUM_THREADS=12                    !スレッド数（--omp thread=XX と同じ数）
export XOS_MMM_L_PAGING_POLICY=demand:demand:demand

numactl -l ./solc0
numactl -C 12-23 -m 4 ./solc0
```

図4　12 コア･1-CMG を使用する場合のジョブスクリプト**<$O-ompw>/run/c12.sh** の例[2]

また，**solver_PCG.c** のインタフェースが，［4］のコードと比べて若干変更となっている。図 5 が変更を施した冒頭部で，配列 W へのアクセス効率向上のために，配列 W に対して連続領域を確保するように変更した。また係数行列に関連したポインタに restrict 型修飾子を適用することによって，最適化の促進を図っている。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <math.h>
#include <omp.h>
#include "solver_PCG.h"

extern int
solve_PCG(int N, int *restrict indexLU, int *restrict itemLU,
          double *restrict D, double *restrict B, double *restrict X,
          double *restrict AMAT, double EPS, int *restrict ITR, int *restrict IER)
{
       double VAL, BNRM2, WVAL, SW, RHO, BETA, RHO1, C1, DNRM2, ALPHA, ERR;
       double Stime, Etime;
       int i, j, ic, ip, L, ip1, N3;

 double (*restrict W)[N] = (double (*)[N])malloc(4*sizeof(double[N]));
       if(W == NULL) {
              fprintf(stderr, "Error: %s¥n", strerror(errno));
       return -1;
       }
```

図 5　**solver_PCG.c** の変更部分，係数行列に関連したポインタに restrict 型修飾子の適用と，配列 W に対して連続領域の確保

（4）実行例

表 2 は，図 3 に示す NX=NY=NZ=128 の場合にコア数（スレッド数）を 1 から 12 まで変化させた場合の PCG 法ソルバーの計算時間，並列化効率の値を示す。計算を 5 回実行して，最速時間を採用している（以下同様に測定している）。コア数（スレッド数）を増加させると，計算粒度（Granularity）の低下，メモリの実行性能が飽和するため，効率は低下するが，12 コア使用時に 75%程度の並列化効率が達成されている。

表 2　PCG 法ソルバーの計算時間（NX=NY=NZ=128）（1～12 コア）（Fortran）

Thread #	sec	Speed-up	Parallel Efficiency (%)
1	50.27	1.00	100.00
2	25.24	1.99	99.60
4	12.98	3.87	96.86
6	9.24	5.44	90.73
8	7.27	6.92	86.50
12	5.27	9.54	79.49

表 3　PCG 法ソルバーの計算時間（NX=NY=NZ=128）（12~48 コア）（Fortran）

Thread #	sec	Speed-up	Parallel Efficiency (%)
12	5.27	12.00	100.00
24	2.78	22.72	94.68
36	1.95	32.49	90.24
48	1.60	39.54	82.38

表 3 は CMG の数を増やして，最大 48 コアまで使用した場合である。12 コア（1-CMG）の場合の計算性能を 12.0 としてある。48 コア使用時の並列化効率は，12 コアの場合を基準とすると 80%程度となっている。スレッド数を変化させるためには，図 4 のジョブスクリプトの中の「**#PJM --omp thread=XY**」の「**XY**」の値を変化させれば良いが，**<$O-ompw>/run/**には，各スレッド数に対して，**cXY.sh**，**fXY.sh**（XY=01,02,04,06,08,12,24,36,48）が用意されている。

2. 複数 CMG での性能向上：First Touch Data Placement

表 3 に示すように，48 コア，すなわち 4-CMG を使用した場合の並列化効率は 12 コア，1-CMG の場合を基準とすると 80%程度である。これは決して悪い数字とは言えないが，更なる高速化を試みよう。

NUMA アーキテクチャでは，プログラムにおいて変数や配列を宣言した時点では，物理的メモリ上に記憶領域は確保されず，ある変数を最初にアクセスしたコア（の属する CMG）のローカルメモリ上に，その変数の記憶領域（ページ）が確保される［3]。

これを **First Touch Data Placement（First Touch）**［3］と呼び，配列の初期化手順により得られる性能が大幅に変化する場合があるため，注意が必要である。例えばある配列を初期化する場合，特に指定しなければ 0 番の CMG で初期化が行われるため，記憶領域は 0 番 CMG のローカルメモリ上に確保される。したがって，他の CMG でこの配列のデータをアクセスする場合には，必ず 0 番 CMG のメモリにアクセスする必要があるため，高い性能を得ることは困難である。

配列の初期化を，実際の計算の手順にしたがって OpenMP を使って並列に実施すれば，実際に

計算を担当する CMG のメモリにその配列の担当部分の記憶領域が確保され，より効率的に計算を実施することができる。1-CMG しか使用しない場合はこのような配慮は不要であり，例えば OpenMP/MPI ハイブリッドで 1-CMG 当りに 1 つの MPI プロセスを使用する場合も同様である。

1. で使用したプログラム類は「**<$O-ompw>/src-c0**（実行形式は **solc0**）」，「**<$O-ompw>/src-f0**（同 **solf0**）」であるが，ここで述べた「First Touch」を適用したプログラム類は，「**<$O-ompw>/src-c1**（同 **solc1**）」，「**<$O-ompw>/src-f1**（同 **solf1**）」にある。プログラムの変更箇所は，係数行列を生成する **poi_gen.c**，**poi_gen.f** である。変更は配列の初期化の部分のみである。

図 6，図 7 に **src-c0**，**src-f0** との相違点を示す。**src-c1**，**src-f1** では配列初期化部分が計算実行時と同様に OpenMP によって並列化されている部分のみが異なっている。表 4 に First Touch 適用の効果を示す。48 コアで約 10%の性能改善が得られている。実行にあたっては，**<$O-ompw>/run/**に，**c1_XY.sh**，**f1_XY.sh**（XY=12,24,36,48）が用意されているので，それらを適宜編集して使用すると良い。

表 4　PCG 法ソルバー計算時間（NX=NY=NZ=128）（12~48 コア）（Fortran），First Touch の効果

	Thread #	sec	Speed-up	Parallel Efficiency (%)
src-f0	12	5.27	12.00	100.00
	24	2.78	22.72	94.68
	36	1.95	32.49	90.24
	48	1.60	39.54	82.38
src-f1	12	5.26	12.00	100.00
	24	2.70	23.39	97.45
	36	1.86	33.83	93.96
	48	1.44	43.77	91.18

```
[XYZ@wisteria01 run]$ cd ../src-c0
[XYZ@wisteria01 src-f0]$ diff poi_gen.c ../src-c1/poi_gen.c
25,29c25,31
        for (i = 0; i <ICELTOT ; i++) {
                BFORCE[i]=0.0;
                D[i]   =0.0;
                PHI[i]=0.0;
                INLU[i] = 0;
        }
---
#pragma omp parallel for private (i)
        for (i = 0; i <ICELTOT ; i++) {
                BFORCE[i]=0.0;
                D[i]   =0.0;
                PHI[i]=0.0;
                INLU[i] = 0;
        }
```

src-c0

src-c1

```
        for (i = 0; i <=ICELTOT ; i++) {
                indexLU[i] = 0;
        }
---
#pragma omp parallel for private (i)
        for (i = 0; i <=ICELTOT ; i++) {
                indexLU[i] = 0;
        }
```

src-c0

src-c1

```
        for(i=0; i<ICELTOT; i++) {
          for(j=indexLU[i]; j<indexLU[i+1]; j++) {
            itemLU[j]=0;
              AMAT[j]=0.0;
          }
        }
---
#pragma omp parallel for private (i,j)
        for(i=0; i<ICELTOT; i++) {
          for(j=indexLU[i]; j<indexLU[i+1]; j++) {
            itemLU[j]=0;
              AMAT[j]=0.0;
          }
        }
```

src-c0

src-c1

図 6　First Touch を適用するためのプログラム変更（**<$O-ompw>/src-c0/poi_gen.c, <$O-ompw>/src-c1/poi_gen.c**）

```
[XYZ@wisteria01 run]$ cd ../src-f0
[XYZ@wisteria01 src-f0]$ diff poi_gen.f ../src-f1/poi_gen.f
25,29c25,31
<         PHI    = 0.d0
<         BFORCE= 0.d0
<           D    = 0.d0
<
<         INLU= 0
---
> !$omp parallel do private (icel)
>         do icel= 1, ICELTOT
>           PHI    (icel)= 0.d0
>           BFORCE(icel)= 0.d0
>           D      (icel)= 0.d0
>           INLU   (icel)= 0
>         enddo
71,72c73,75
<         indexLU= 0
<
<         do icel= 1, ICELTOT
<           indexLU(icel)= INLU(icel)
<         enddo
---
>         indexLU(0)= 0
>
> !$omp parallel do private (icel)
>         do icel= 1, ICELTOT
>           indexLU(icel)= INLU(icel)
>         enddo
85,86c88,94
<         itemLU= 0
<           AMAT= 0.d0
---
> !$omp parallel do private (icel,k)
>         do icel= 1, ICELTOT
>         do k= indexLU(icel-1)+1, indexLU(icel)
>           itemLU(k)= 0
>           AMAT   (k)= 0.d0
>         enddo
>         enddo
```


図 7　First Touch を適用するためのプログラム変更（**<$O-ompw>/src-f0/poi_gen.f, <$O-ompw>/src-f1/poi_gen.f**）

3. 疎行列格納形式の効果

図 8 は，CG 法で疎行列ベクトル積 $A\ p= q$ の計算を実行している部分で，**<$O-ompw>/src-c0,src-c1** 及び**<$O-ompw>/src-f0,src-f1** では，前号記事でも述べたようにこのように CRS（Compressed Row Storage）形式［5］と呼ばれる疎行列格納形式を採用している：

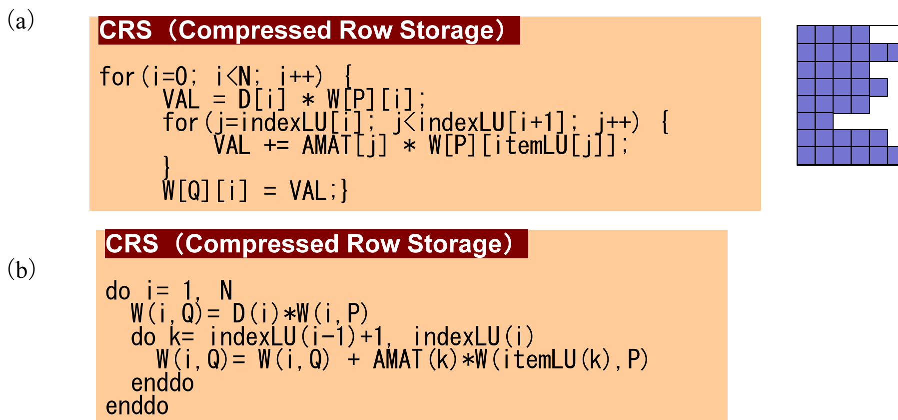


図 8　CRS 形式（Compressed Row Storage）［5］による疎行列格納方法，疎行列ベクトル積実装，（a）C 言語（**<$O-ompw>/src-c0,src-c1**）（b）Fortran（**<$O-ompw>/src-f0,src-f1**）

疎行列ベクトル積の特徴は，右辺に現れる **W[P][itemLU[j]]**，**W(itemLU(k),P)** が間接参照を含むため，memory-bound なプロセスとなっていることである。CRS 形式はメモリ，計算量の削減には効果的であるが，メモリアクセス効率は必ずしも良くない。Ellpack-Itpack（ELL）形式［5］は各行における非零非対角成分数は最大非零非対角成分数に固定する方法で（図 9），実際に非零非対角成分が存在しない部分は係数=0 として計算する。計算量，必要記憶容量ともに CRS 形式と比較してやや増加するが，高いメモリアクセス効率が得られるため，計算時間としては大幅に短縮できる場合がある［6］。非ゼロ非対角成分が存在しない列に対しては，ダミーの列番号を参照し，係数行列は 0 として扱う（図 9）。

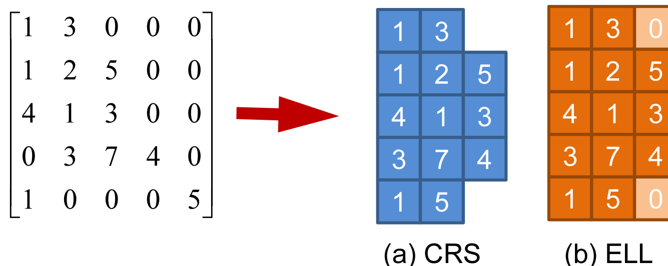


図 9　疎行列の格納形式（a）CRS（Compressed Row Storage），（b）ELL（Ellpack-Itpack）［5］

本稿で扱うアプリケーションは，疎行列の非ゼロ非対角成分の数が最大 6 であるため，ELL 形式を容易に適用することができる。非ゼロ非対角成分数が行によって大幅に変動する場合は，よりフレキシブルな Sliced-ELL 形式［7］が使用される場合もある［6］。

図 10 は，図 8 に示す疎行列ベクトル積を ELL 形式で記述したものである。C 言語の場合は，図 11 に示すような実装が考えられるが，A64FX では非常に計算速度が遅いため，図 10（a）の

ような実装を採用している。Intel Xeon プロセッサでは，図 10（a）と図 11 の実装は同じ効率が得られる。ダミー列番号の設定方法等の実装の詳細については，プログラム本体（**<$O-ompw>/src-c2**（実行形式は **solc2**），**<$O-ompw>/src-c2**（同 **solf2**））及び講習会資料［2］を参照されたい。

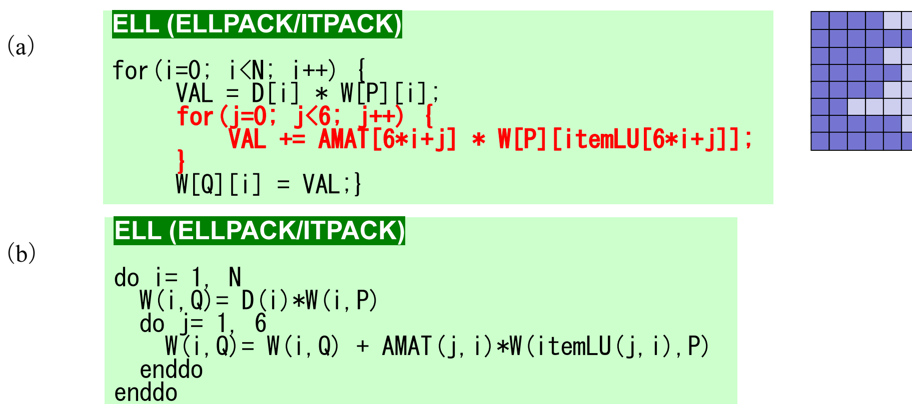


図 10　ELL 形式（Ellpack-Itpack）［5］による疎行列格納方法，疎行列ベクトル積実装，（a）C 言語（**<$O-ompw>/src-c2**）（b）Fortran（**<$O-ompw>/src-f2**）

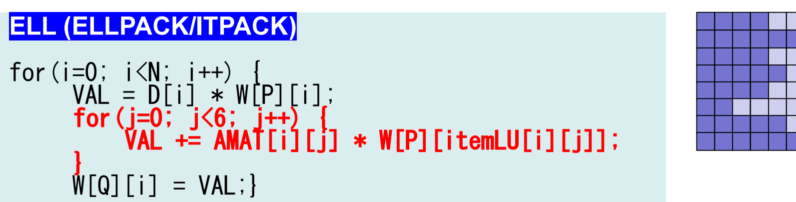


図 11　ELL 形式（Ellpack-Itpack）［5］による疎行列格納方法，疎行列ベクトル積実装，図 8（a）の拡張としては自然な実装であるが，本稿ではこの方法は採用していない

4. ループオーバーヘッド削減の効果

OpenMP の実行モデルは，fork-join モデル［8］と呼ばれるもので，通常はマスタースレッドによるシリアル実行であるが，OpenMP 指示文（directive）によりマルチスレッドが生成し，並列実行を実施するものである。

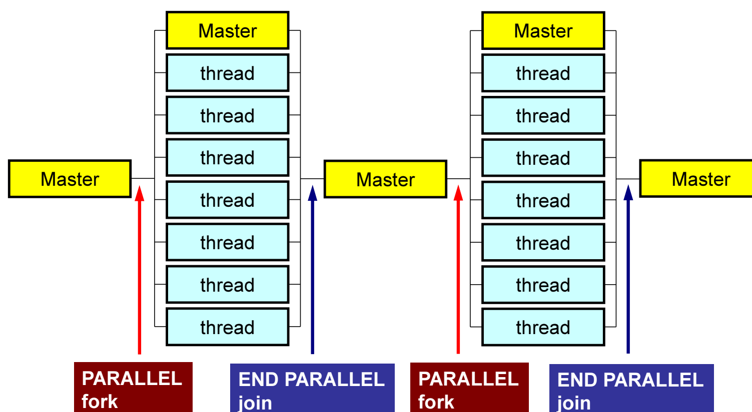


図 12　OpenMP の実行モデル：fork-join モデル［8］

共役勾配法の各ステートメントのほとんどは各ベクトルに対するループであり，これまで紹介したプログラムでは，各ループに OpenMP 指示文を挿入している。図 13 は CG 法の後半部分の実装例（CRS 形式）であり，各ループに「**#pragma omp parallel for**」，「**!$omp parallel do**」が挿入されている。「**#pragma omp parallel**」，「**!$omp parallel**」により，図 12 に示すような fork-join が各ステートメントごとに生じるため，これがオーバーヘッドとなる可能性がある。本章で紹介する新たな実装（**<$O-ompw>/src-c3**（実行形式は **solc3**），**<$O-ompw>/src-c3**（同 **solc3**））は ELL 法による実装（**<$O-ompw>/src-c2**，**<$O-ompw>/src-c2**）に対して，「**#pragma omp parallel**」，「**!$omp parallel**」の呼び出しを各反復で一回とし，各ループに対しては「**#pragma omp for**」，「**!$omp do**」を適用するものである（図 14）。

(a)

```
#pragma omp parallel for private (i,VAL,j)
  for(i=0; i<N; i++) {
      VAL = D[i] * W[P][i];
      for(j=indexLU[i]; j<indexLU[i+1]; j++) {
          VAL += AMAT[j] * W[P][itemLU[j]];
      }
      W[Q][i] = VAL;
  }

C1 = 0.0;
#pragma omp parallel for private (i) reduction(+:C1)
  for(i=0; i<N; i++) {
      C1 += W[P][i] * W[Q][i];
  }
  ALPHA = RHO / C1;

#pragma omp parallel for private (i)
  for(i=0; i<N; i++) {
      X[i]    += ALPHA * W[P][i];
      W[R][i] -= ALPHA * W[Q][i];
  }


DNRM2 = 0.0;
#pragma omp parallel for private (i) reduction(+:DNRM2)
  for(i=0; i<N; i++) {
      DNRM2 += W[R][i]*W[R][i];
  }

  ERR = sqrt(DNRM2/BNRM2);
...
```

(b)

```
!$omp parallel do private(i,VAL,k)
      do i= 1, N
        VAL= D(i)*W(i,P)
        do k= indexLU(i-1)+1, indexLU(i)
          VAL= VAL + AMAT(k)*W(itemLU(k),P)
        enddo
        W(i,Q)= VAL
      enddo

      C1= 0.d0
!$omp parallel do private(i) reduction(+:C1)
      do i= 1, N
        C1= C1 + W(i,P)*W(i,Q)
      enddo

      ALPHA= RHO / C1

!$omp parallel do private(i)
      do i= 1, N
        X(i)  = X(i)   + ALPHA * W(i,P)
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo

      DNRM2= 0.d0
!$omp parallel do private(i) reduction(+:DNRM2)
      do i= 1, N
        DNRM2= DNRM2 + W(i,R)**2
      enddo

      ERR = dsqrt(DNRM2/BNRM2)...
```

```
Compute r^(0)= b-[A]x^(0)
for i= 1, 2, …
    solve [M]z^(i-1)= r^(i-1)
    ρ_(i-1)= r^(i-1) z^(i-1)
    if i=1
      p^(1)= z^(0)
     else
      β_(i-1)= ρ_(i-1)/ρ_(i-2)
      p^(i)= z^(i-1) + β_(i-1) p^(i-1)
    endif
    q^(i)= [A]p^(i)
    α_i = ρ_(i-1)/p^(i)q^(i)
    x^(i)= x^(i-1) + α_i p^(i)
    r^(i)= r^(i-1) - α_i q^(i)
    check convergence |r|
end
```

図 13　CG 法の後半部分の実装（CRS 形式）（a）C 言語（**<$O-ompw>/src-c0,src-c1**），（b）Fortran（**<$O-ompw>/src-f0,src-f1**）

5. 計算結果（First-Touch，行列格納形式，ループオーバーヘッド削減の効果）

3.，4.に示した各実装例については，**c2_48.sh**，**c3_48.sh**，**f2_48.sh**，**f3_48.sh** を使用して実行することができる。これらは，全て 48 コアを使用した場合であるが，ジョブスクリプト内でコア数を変更して実行することも可能である。表 5 に 48 コアを使用した場合の PCG 法の計算時間を示す。First-Touch の効果は C 言語でより大きい。ループオーバーヘッド削減の効果は Frotran では 5%程度，C 言語（clang）ではむしろ遅くなっているが，C 言語（trad）では大きな効果が認められ，sol-c3 のレベルでは clang と trad の差はほとんど無くなっている。

表 5　PCG 法ソルバーの計算時間（sec.）（NX=NY=NZ=128）（48 コア）

	Fortran	C（clang）	C（trad）
初期設定（sol-c0, sol-f0）	1.671	1.564	2.354
＋First-Touch（sol-c1, sol-f1）	1.480	1.122	1.720
＋ELL（sol-c2, sol-f2）	0.747	0.809	1.127
＋omp-parallel 削減（sol-c3, sol-f3）	0.707	0.834	0.854

(a)

```
      ITR= N
      Stime= omp_get_wtime()

      do L= 1, ITR
!$omp parallel private(i,k,VAL)

!$omp do
      do i= 1, N
        W(i,Z)= W(i,R)*W(i,DD)
      enddo

      RHO= 0.d0
!$omp do reduction(+:RHO)
      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo

      if ( L.eq.1 ) then
!$omp do
        do i= 1, N
          W(i,P)= W(i,Z)
        enddo
       else
         BETA= RHO / RHO1
!$omp do
         do i= 1, N
           W(i,P)= W(i,Z) + BETA*W(i,P)
         enddo
      endif

!$omp do
      do i= 1, N
        VAL= D(i)*W(i,P)
        do k= indexLU(i-1)+1, indexLU(i)
          VAL= VAL + AMAT(k)*W(itemLU(k),P)
        enddo
        W(i,Q)= VAL
      enddo

      C1= 0.d0
!$omp do reduction(+:C1)
      do i= 1, N
        C1= C1 + W(i,P)*W(i,Q)
      enddo

      ALPHA= RHO / C1

!$omp do
      do i= 1, N
        X(i)   = X(i)    + ALPHA * W(i,P)
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo

      DNRM2= 0.d0
!$omp do reduction(+:DNRM2)
      do i= 1, N
        DNRM2= DNRM2 + W(i,R)**2
      enddo

!$omp end parallel
      ERR = dsqrt(DNRM2/BNRM2)...
```

(b)

```
  *ITR = N;
  Stime = omp_get_wtime();

  for(L=0; L<(*ITR); L++) {
#pragma omp parallel private (i,j,VAL){

#pragma omp for
    for(i=0; i<N; i++) {
      W[Z][i] = W[R][i]*W[DD][i];
    }

    RHO = 0.0;
#pragma omp for reduction(+:RHO)
    for(i=0; i<N; i++) {
      RHO += W[R][i] * W[Z][i];
    }

if(L == 0) {
#pragma omp for
    for(i=0; i<N; i++) {
      W[P][i] = W[Z][i];
    }
    } else {
     BETA = RHO / RHO1;
#pragma omp for
    for(i=0; i<N; i++) {
      W[P][i] = W[Z][i] + BETA * W[P][i];
    }
}

#pragma omp for
  for(i=0; i<N; i++) {
      VAL = D[i] * W[P][i];
      for(j=0; j<6; j++) {
          VAL += AMAT[6*i+j] *
W[P][itemLU[6*i+j]];
      }
      W[Q][i] = VAL;
  }

  C1 = 0.0;
#pragma omp for reduction(+:C1)
  for(i=0; i<N; i++) {
      C1 += W[P][i] * W[Q][i];
  }
  ALPHA = RHO / C1;

#pragma omp for
  for(i=0; i<N; i++) {
      X[i]     += ALPHA * W[P][i];
      W[R][i] -= ALPHA * W[Q][i];
  }

  DNRM2 = 0.0;
#pragma omp for reduction(+:DNRM2)
  for(i=0; i<N; i++) {
      DNRM2 += W[R][i]*W[R][i];
  }

}

ERR = sqrt(DNRM2/BNRM2);
```

図 14　ループオーバーヘッドを削減した実装例（a）C 言語（**<$O-ompw>/src-c3**），（b）Fortran（**<$O-ompw>/src-f3**）

今回は，Odyssey 上での実行方法，First Touch Data Placement（First Touch）・疎行列格納形式・ループオーバーヘッド削減の効果について述べた。次回（恐らく最終回）は更なる最適化，詳細プロファイラによる性能測定法について紹介する。

（第 3 回（7 月号）へ続く）

参　考　文　献

[1] Wisteria/BDEC-01（「計算・データ・学習」融合スーパーコンピュータシステム）:https://www.cc.utokyo.ac.jp/supercomputer/wisteria

[2] OpenMPによるマルチコア・メニィコア並列プログラミング入門（Wisteria/BDEC-01（Odyssey，A64FX搭載），http://nkl.cc.u-tokyo.ac.jp/seminars/multicore2021/

[3] Mattson, T.G., Sanders, B.A., Massingill, B.L., Patterns for Parallel Programming, Software Patterns Series (SPS), Addison-Wesley, 2005

[4] P3D関連資料
- ソースコード等：http://nkl.cc.u-tokyo.ac.jp/files/fvm.tar
- 解説資料（Fortran）：http://nkl.cc.u-tokyo.ac.jp/seminars/multicore2021/omp-f-01.pdf
- 解説資料（C）：http://nkl.cc.u-tokyo.ac.jp/seminars/multicore2021/omp-c-01.pdf

[5] Saad, Y.: Iterative Methods for Sparse Linear Systems Second Edition, SIAM, 2003

[6] Nakajima, K., Optimization of Serial and Parallel Communications for Parallel Geometric Multigrid Method, Proceedings of the 20th IEEE International Conference for Parallel and Distributed Systems (ICPADS 2014) 25-32, Hsin-Chu, Taiwan, 2014

[7] Monakov, A., Lokhmotov, A., Avetisyan, A., Automatically tuning sparse matrix-vector multiplication for GPU architectures, Lecture Notes in Computer Science 5952, 112-125, 2010

[8] OpenMP ARB（Architecture Review Board）：https://www.openmp.org/