3.3.2. MPI-IO の実アプリへの適用

ー3 次元構造格子のベクトルデータプログラム MPI-IO 化ー

富士通株式会社　　杉崎 由典

1. はじめに

MPI で並列化されたプログラムの入出力処理は、多くの場合で split file 処理方式と呼ばれる処理方式を用いている。Split file 処理方式は、プロセス毎に 1 つのファイルを生成する処理方式である。そのため、プロセス数が増えると生成されるファイル数もその分多くなる。

HPC においては、並列数は 100 や 1000 を超えることも珍しくないが、並列数分のファイルを入出力処理で扱う必要が生じると、ファイル管理が困難になることや、ファイルシステムへの負荷が増大する恐れがある。

一方、並列 IO 処理である MPI-IO 機能を用いることによって、複数プロセス実行であっても生成されるファイル数を 1 つに抑えることが可能である。

そこで、この MPI-IO 機能を用い、生成されるファイル数を 1 つに抑えることを実アプリで検証した。実アプリは、3 次元構造格子のベクトルデータプログラムを用いた。

2. ファイル転送方式

split file 方式と MPI-IO 方式の違いを提示する。

2.1. split file 方式

プロセス毎にファイルを生成する split file 方式について説明する。

以下の図の様に、split file 方式は、各プロセスがそれぞれファイルを生成する。（○はプロセス、□はファイルを表す）

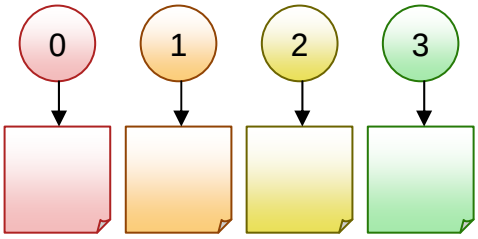

図1. split file 方式概略

```
sprintf(filename, “%s.%03d", BASENAME, this_rank);
unlink(filename);

fd = open(filename, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd, (void *)buf, bufsize);
fsync(fd);
close(fd);
```

図2. プログラム　Split file 方式

以下に split file 方式の特徴を挙げる。

・一般的な POSIX-IO で実現できるため、わかりやすいプログラムで記述できる。

・ファイルがプロセス数個できるため、プロセス増加に伴い、ファイルシステムに負荷がかかる。

2.2. MPI-IO 方式

複数のプロセスがあってもファイルは 1 つだけを生成する MPI-IO 方式について説明する。

以下の図の様に、MPI-IO 方式は、各プロセスがそれぞれ 1 つのファイルを分割して、各プロセスに割り当てられた部分に対して転送を行う。生成されたファイルは 1 つとして見える。(○はプロセス、□はファイルを現す)

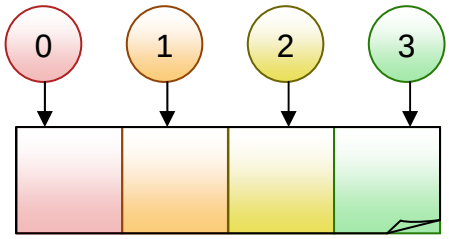

図3. MPI-IO 方式概略

```
MPI_FILE_DELETE(fname);

MPI_File_open(MPI_COMM_WORLD, filename,
    MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, myrank * bufsize, MPI_SEEK_SET);
MPI_File_write(fh, buf, elements, MPI_DOUBLE, &status);
MPI_File_sync(fh);
MPI_File_close(&fh);
```

図4. プログラム　MPI-IO 方式

以下に MPI-IO 方式の特徴を挙げる。

・ユーザーはプロセスの同期を取ることなく、一つのファイルを作成することができる。
・多重書き込みをサポートする高速なネットワーク共有ファイルシステムが前提となる。

3. 検証プログラム

今回検証対象としたのは 3 次元構造格子のベクトルデータプログラムである。

3.1. プログラム構造

このプログラムのプログラム構造と IO 処理部の特徴を以下に示す。

・プログラム構造
- 8 並列プログラム
- プロセスを 2×2×2 の 3 次元に分割(2×2×2 のカルテシアン空間)
- IO は各プロセス毎に実施
- IO 処理は OUTP,OUTR で実施

・IO 処理
- write 文(DO 型並び)による配列の出力
- 配列のサイズは、各プロセスで異なる。

例：配列 U の場合　宣言サイズ U(63,63,243)
rank=0 U(60,60,210,3)
rank=1 U(60,60,211,3)
rank=2 U(60,61,210,3)
rank=3 U(60,61,211,3)
rank=4 U(61,60,210,3)
rank=5 U(61,60,211,3)
rank=6 U(61,61,210,3)
rank=7 U(61,61,211,3)

3.2. MPI-IO 対象ファイル

3 次元カルテシアン空間に分けられ、それぞれの次元を 000 または 001 で現すことで、下記の 5 ファイルが対象となる。

MPI-IO 対象ファイル

fort_xxx_yyy_zzz.20	xxx は 000 または 001,yyy は 000 または 001,zzz は 000 または 001
fort_xxx_yyy_zzz.21	
fort_xxx_yyy_zzz.22	
fort_xxx_yyy_zzz.23	
fort_xxx_yyy_zzz.50	

3.2. MPI-IO 化イメージ

元のプログラムで生成されるファイルと MPI-IO 化後のファイルのイメージを以下に示す。

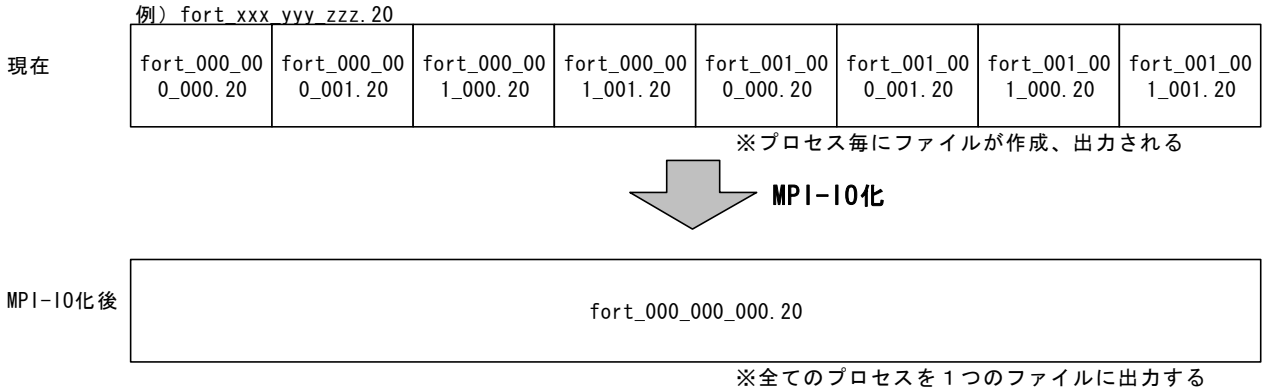

図5. MPI-IO 化イメージ

4. MPI-IO 適用方法

今回実施した MPI-IO 化の方法を示す。

4.1. IO 部の変形

以下の 2 つの方法がある。

・MPI-IO 化前に IO 部を変形

- 複数 write 文をまとめて以下の様に変換

```
write(20) (((U(I1,I2,I3,1),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
write(20) (((U(I1,I2,I3,2),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
write(20) (((U(I1,I2,I3,3),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
    ↓
write(20) U
```

・IO 部変換しない場合

- 各プロセスのインデックス 3 次元分を MPI_allgather で各プロセスに分配
- 上記 3 次元分のインデックスから、自 rank の offset 値を算出
  ただし、この方法は offset 計算が煩雑になる。上記の配列 U のケースに適用するためには、U(…,1),U(…,2),U(…,3)それぞれで offset 計算しなければならない。

上記理由から、IO 部をまとめて write する方法を採用した。

4.2. MPI-IO 化

元ソースから MPI-IO 化するためのポイントと、その変更方法を以下に示す。例として write のケースを挙げた。

以下の手順で MPI-IO 化を行った。

- OPEN 文、CLOSE 文、WRITE 文をそれぞれ、MPI_FILE_OPEN、MPI_FILE_CLOSE、MPI_FILE_WRITE_AT に変換する。
- MPI_FILE_WRITE_AT の引数にある、ファイル先頭からのオフセット値の計算処理を追加する。

■ 元ソース

```
SUBROUTINE OUTP
      …

OPEN(20,FILE=FN,STATUS='UNKNOWN',FORM=
'UNFORMATTED', ACCESS='APPEND')
      WRITE(FN,'(3A)') 'fort',FNP,'.21'

      WRITE(20) NTC,TM
      WRITE(20)
(((U(I1,I2,I3,1),I1=1,MYI1MAX),I2=1,MYI2MAX),
     &
I3=1,MYI3MAX)
      WRITE(20)
(((U(I1,I2,I3,2),I1=1,MYI1MAX),I2=1,MYI2MAX),
     &
I3=1,MYI3MAX)
      WRITE(20)
(((U(I1,I2,I3,3),I1=1,MYI1MAX),I2=1,MYI2MAX),
     &
I3=1,MYI3MAX)
      WRITE(20)
(((P(I1,I2,I3),I1=1,MYI1MAX),I2=1,MYI2MAX),
     &
I3=1,MYI3MAX)
      WRITE(20)
(((DIV(I1,I2,I3),I1=1,MYI1MAX),I2=1,MYI2MAX),
     &
I3=1,MYI3MAX)
      …
      CLOSE(20)
      RETURN
      END
```

■ MPI-IO 化後

```
SUBROUTINE OUTP
      …
      CALL  MPI_FILE_OPEN(MPI_COMM_SELF,FN,MPI_MODE_CREATE
     +
     & MPI_MODE_WRONLY,MPI_INFO_NULL,20,IERROR)
      WRITE(FN,'(3A)') 'fort',FNP,'.21'
      ntc_size=4
      offset=ntc_size*myrank
      CALL MPI_FILE_WRITE_AT(20,offset,NTC,ntc_size,MPI_INTEGER,
     & status,IERROR)
      tm_size=8
      offset=ntc_size*(myrank+1)+tm_size*myrank
      CALL MPI_FILE_WRITE_AT(20,offset,TM,tm_size,MPI_REAL8,
     & status,IERROR)
      u_size=md1*md2*md3*3*8
      offset=(ntc_size+tm_size)*(myrank+1)+u_size*myrank
      CALL MPI_FILE_WRITE_AT(20,offset,U,u_size,MPI_BYTE,
     & status,IERROR)
      p_size=md1*md2*md3*8
      offset=(ntc_size+tm_size+u_size)*(myrank+1)+p_size*myrank
      CALL MPI_FILE_WRITE_AT(20,offset,P,p_size,MPI_BYTE,
     & status,IERROR)
      div_size=md1*md2*md3*8

offset=(ntc_size+tm_size+u_size+p_size)*(myrank+1)+div_size*myrank
      CALL MPI_FILE_WRITE_AT(20,offset,DIV,div_size,MPI_BYTE,
     & status,IERROR)
      CALL MPI_FILE_CLOSE(20,IERROR)
      RETURN
      END
```

オフセット計

明示的オフセット使用

図6.　MPI-IO 化実施例

4.3. 出力ファイル

MPI-IO 化前と MPI-IO 化後の出力ファイルのサイズを以下に挙げる。例としてファイル装置番号 20 番のケースを挙げた。

・MPI-IO 化前(write(20))

サイズ(Byte)	ファイル名
30240060	fort_000_000_000.20
30384060	fort_000_000_001.20
30744060	fort_000_001_000.20
30890460	fort_000_001_001.20
30744060	fort_001_000_000.20
30890460	fort_001_000_001.20
31256460	fort_001_001_000.20
31405300	fort_001_001_001.20
246554920	合計

・MPI-IO 化後(write(20))

サイズ(Byte)　ファイル名

277766496　fort_000_000_000.20

5. 実施状況

MPI-IO 化の実施状況及び結果を以下に示す。

・一般ユーザーが書き換えすることを想定

- Fortran プログラミングの経験があり、MPI の経験がない者が実施。
- MPI-IO 化の手順のみを提示。（本稿の 2.1 split file 方式,2.2 MPI-IO 方式の説明レベル）

・期間

- 約 10 日間で実施。（一日 4 時間）
  プログラムの解析で約 3 日間。
  MPI-IO 化とデバッグまでを約 7 日間。（ファイルが出力されるまで）

・状況

- 長時間ジョブとなるため、演算部の回数を縮小して実施。
  500step を 1 ステップで実施。IO 部には影響なし。
- プログラミング(MPI-IO への書き換え)は順当に実施できた。
  手順が判れば比較的簡単。
- MPI_FILE_SYNC はなし。
- IO 処理が集約されているプログラムであったため、修正箇所を限定して作業できた。

6. 評価

実施結果から MPI-IO 化のポイントについて抽出した。以下に示す。

・MPI 化の移行工数

- 変換する MPI-IO の関数が判れば、通常の作業工数で書き換え可能。
- 予想よりは難しくない感触。

・MPI-IO 化のポイント

- 書き換え方法の明確化（指針）
- IO 対象を全配列化（全配列対象に書き換え）
- オフセット計算を間違えない（注意・対策・仕組み）

・注意点／課題

- オフセット計算が間違い易い
  対策：デバッグを考慮しながら作業。(デバッグ出力を挿入)
- デバッグ方法の検討

本稿では、１つのアプリを題材として MPI-IO 化を実施し、MPI-IO 化の手順や MPI-IO 化のポイントを抽出した。MPI-IO 化のポイントを押さえれば、それ程難しくなく通常の作業工数で書き換えが可能であることが判った。ただし、デバッグ方法は検討課題である。

7. 参考文献

- Message Passing Interface Forum　　http://www.mpi-forum.org/

以上