ppOpen-HPC:
Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT).

# ppOpen-AT

ver. 1.0.0

# コンパイルオプション　ガイド

## License



This software is one of the results of the JST CREST ``ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT)" project.

## Change History

2016.Mar. Version 1.0.0 is released.

# ppOpen-AT プリプロセッサのオプションについて

2016/03/27

利用方法、オプションの設定についてまとめてあります。

## 利用方法

コマンドラインから oat をオプションとファイル名を指定して実行します。ファイル名はスペースを空けて複数を並べて記述出来ます。

oat ［オプション］ ファイル名…

## 機能説明

ppOpenAT は、ソースコード中の!OAT$または#pragma OAT で始まる部分を解釈して変換したソースコードを作成します。

Fortran の場合は !OAT$でＣの場合は#pragma OAT で始まる部分が対象になります。OAT フォルダに指定したファイル名から生成された複数のファイルが出力されます。

入力ファイル（指定ソースファイル数＋１）

| |
|---|
| Source.f　変換元になるソースファイル。複数ファイルを指定可能。 |
| OAT.h OAT.h ファイル生成時に参照される元になる OAT.h ファイル。 |

出力ファイル（指定ソースファイル数＋６）

| |
|---|
| OATLog.txt OAT 実行時のログ出力ファイル |
| OAT/OAT_Source.f ソースファイルを元にして作成されるファイル。 |
| OAT/OAT_InstallRoutines.f インストールリージョン用に作成されるファイル。 |
| OAT/OAT_StaticRoutines.f スタティクリージョン用に作成されるファイル。 |
| OAT/OAT_DynamicRoutines.f ダイナミックリージョン用に作成されるファイル。 |
| OAT/OAT_ControlRoutines.f OAT のコントロール用に作成されるファイル。 |
| OAT/OAT.h OAT.h に変数定義が追加されたヘッダファイル |

Source.f は、変換元になるソースファイルです。この中の !OAT$で指定した部分が変換されます。複数のファイルを並べて指定出来ます。

対象になるファイルの拡張子は、.f90,c とそれ以外の３種類です。

.f90 の場合は自由形式の Fortran、.c の場合はＣ言語、それ以外の拡張子は固定形式の

Fortran として扱われます。ただし、オプションで-free や-fixed を付けた場合は拡張子と無関係に自由形式の Fortran と固定形式の Fortran として扱われます。

OAT.h OAT.h ファイル生成時に参照される元になる OAT.h ファイルです。引数としての指定は不要です。常に最初のソースファイルと同じフォルダにある OAT.h ファイルが参照されます。

自由形式の Fortran と固定形式の Fortan、C 言語で、それぞれ異なる OAT.h の指定が必要です。プログラムで使用するパラメータ等の記述も可能です。

OAT が有効になった時に OAT_INSTALL 等の定数の参照が発生します。OAT 関連のコードの記述がある部分に対して OAT.h の内容が参照できるように元になるソースコードへの include 'OAT.h'を手動で追加して下さい。Cの場合は #include "OAT.h"になります。

出力ファイルの構造

出力ファイルのプログラム構造は対象となるソースコードの種類で変わります。固定形式の Fortran を基本として、Fortran90 と C で異なる構造での出力になっています。

固定形式の Fortran

固定形式の Fortran の場合は元のソースに include 文が追加された OAT_ソースが作成され１つのソースとして扱われます。コンパイル対象も OAT_ソース名のファイルになり、OAT_ControlRoutines ファイル等の個別コンパイルは不要です。

OAT/OAT_Source.f の終了部分に以下の内容を追加

```
include 'OAT_ControlRoutines.f'
include 'OAT_DynamicRoutines.f'
include 'OAT_InstallRoutines.f'
include 'OAT_StaticRoutines.f'
```

自由形式の Fortran

自由形式の Fortran の場合は、module と use を使って結合されます。各ファイルの先頭に module 文が付けられて、OAT_ソースからは use で参照されます。

コンパイルは OAT_ControlRoutines 等を-c オプションをつけてそれぞれをコンパイルしてモジュールファイルを作成してから、.o を結合して構築します。以下に構造を示します。

コンパイルもモジュールの階層順に合わせて InstallRoutines 等から先に行う必要があります。

OAT/OAT_Source.f90 の先頭部分

```
      Program main
            use ppohAT_ControlRoutines
            use ppohAT_InstallRoutines
            use ppohAT_StaticRoutines
            use ppohAT_DynamicRoutines
```

OAT/OAT_ControlRutions.f90 の先頭部分

```
      module ppohAT_ControlRoutines
            use ppohAT_InstallRoutines
            use ppohAT_StaticRoutines
            use ppohAT_DynamicRoutines
            implicit none
            public
            contains
!
!     各種サブルーチン
!     === OAT_ATset
!     ================================================================
      subroutine OAT_ATset(OAT_TYPE, OAT_Routines)
      integer     OAT_TYPE
      character*8 OAT_Routines

      include 'OAT.h'
```

OAT/OAT_InstallRutions.f90 の先頭部分

```
      module ppohAT_InstallRoutines
            public
            contains
      subroutine OAT_InstallMyUnroll(n, dst, src, iusw1)
```

OAT/OAT_InstallRutions.f90 の先頭部分

```
module ppohAT_StaticRoutines
    public
    contains
subroutine OAT_StaticMyUnroll(n, dst, src, iusw1)
```

OAT/OAT_DynamicRutions.f90 の先頭部分

```
module ppohAT_DyamicRoutines
    public
    contains
subroutine OAT_DynamicMyUnroll(n, dst, src, iusw1)
```

モジュールの階層構造

```
Program Main ( main でない場合もあります)
    Module ppohAT_ControlRoutines
        Module ppohAT_InstallRoutines
        Module ppohAT_StaticRoutines.f
        Module ppohAT_DynamicRoutines
```

モジュールを参照するコードを含む場合

元のソースコードが use でモジュールを参照している場合に生成されるコードについて説明します。

現状では、ソースコードの Module 内に use があった場合に InstallRoutines 等に use として追加する形になっています。Module 内以外の use に関しては対象外としています。Program Main 内の use は対象外にしています（ppohFDM_AT_1.0.0 等を参照）。

use として追加したい場合は、入力ソースファイル内の Module 内に use として追加して下さい。また、OAT から見えているファイルは全ソースでなく引数として与えれたソースファイルのみが対象になるため、手動による調整が必要になる場合もあります。

元の SrcCode.f90 （Mousel m1 が Use m2 で参照している場合）

```
Module m1
Use m2
Contains
```

OAT/OAT_InstallRutions.f90 の先頭部分に use を追加

```
module ppohAT_InstallRoutines
    use m2  ！追加された use
    public
    contains
subroutine OAT_InstallMyUnroll(n, dst, src, iusw1)
```

C 言語の場合

C 言語の場合、生成されたＯＡＴの複数のファイルを Include する形になっています。固定形式 Fortran の場合と近い形です。また、先頭部分には、内部で使用される関数のプロトタイプ宣言のための “OAT_Routines.h”が生成され include され、その後に内部で使用される変数の宣言が行われます。

コンパイルについては OAT_Source.c をコンパイルすることになります。

OAT/OAT_Source.c の先頭部分

```
#include <stdio.h>
#include "OAT.h"

#include "OAT_Routines.h"   // プロトタイプ宣言
int OAT_iusw1_MyList;       // 内部で使われる変数宣言
```

OAT/OAT_Source.c の終了部分に追加

```
#include "OAT_ControlRoutines.c"
#include "OAT_DynamicRoutines.c"
#include "OAT_InstallRoutines.c"
#include "OAT_StaticRoutines.c"
```

AT 領域内でのサブルーチン呼び出し

OAT リージョン内からサブルーチンを呼ぶ場合には、InstallRoutins 等の各 OAT のコードから、OAT_Source への呼び出しが必要となります。

例えば、Select リージョンから２つのサブルーチンを呼び出した場合には、単純に変換を行った場合には以下のような呼び出しになります。

OAT_Source.f90 からの InstallRoutines への呼び出し部分

```
program    main
use ppohAT_ControlRoutines
use ppohAT_InstallRoutines
use ppohAT_StaticRoutines
use ppohAT_DynamicRoutines

call OAT_InstallSelectMatMul(A,B,C,N,iusw1_SelectMatMul);  ! 呼び出し
```

OAT_InstallRoutines.f90 で呼ばれる部分

```
module ppohAT_InstallRoutines
    public
    contains

subroutine OAT_InstallSelectMatMul(A, B, C, N, iusw1)
select case(iusw1)
 case(1)
    call SelectSub1(A, B, C, N)    ! Souce.f90 側のサブルーチン呼び出し
 case(2)
    call SelectSub2(A, B, C, N)    ! Souce.f90 側のサブルーチン呼び出し
end select
```

ここで、SelectSub1,SelectSub2 は、OAT_Source.f90 内にあるため OAT_InstallRoutines からは見えないため呼び出しが失敗します。この問題を解決するために Source.f90 内の SelectSub1,SelectSub2 の内容を OAT_InstallRoutines 内に別名で複写して呼び出す形になっています。

以下のように _OAT を付けた名前のサブルーチンを作成して InstallRoutines 内部での呼び出しとしてあります。対象リージョンが Static や Dynamic の場合には、それぞれの Routines.f90 に配置されます。

ただし、自由形式 Fortran の場合は名前が重複するため異なるリージョンに複数を割り当てることは出来ません。固定形式 Fortran の場合には Include で１つのソースコードとして結合しているため同じ名前での呼び出しが可能です。

OAT_InstallRoutines.f90 で呼ばれる部分

```
        module ppohAT_InstallRoutines
            public
            contains

        subroutine OAT_InstallSelectMatMul(A, B, C, N, iusw1)
        select case(iusw1)
         case(1)
            call SelectSub1_OAT(A, B, C, N)    ！名前を変えて複写したサブルーチン
          case(2)
            call SelectSub2_OAT(A, B, C, N)    ！名前を変えて複写したサブルーチン
        end select
!
！名前を変えて複写して作成したサブルーチン
!
        subroutine SelectSub1_OAT(A, B, C, N)
        integer N
        real*8   A(N,N), B(N,N), C(N,N)

        real*8   da1, da2
        real*8   dc

        ctmp = "fusionloopSub1"
        call OAT_SetParm_OAT(1,ctmp,N,iusw1_fusionloopSub1)
        call OAT_InstallfusionloopSub1(N,C,A,B,iusw1_fusionloopSub1)
!!oat$ install LoopFusionSplit region start
!!oat$ name fusionloopSub1
!         do i=1, N
!            do j=1, N
!               do k=1, N
!                 C(i,j) = C(i,j) + A(i,k) * B(k,j)
!               enddo
!            enddo
!         enddo
!!oat$ install LoopFusionSplit region end
```

```
        return
        end subroutine SelectSub1_OAT



        subroutine SelectSub2_OAT(A, B, C, N)
        integer N
        real*8   A(N,N), B(N,N), C(N,N)

        real*8   da1, da2
        real*8   dc

        ctmp = "fusionloopSub2"
        call OAT_SetParm_OAT(1,ctmp,N,iusw1_fusionloopSub2)
        call OAT_InstallfusionloopSub2(N,C,A,B,iusw1_fusionloopSub2)
!!oat$ install LoopFusionSplit region start
!!oat$ name fusionloopSub2
!           i=1
!           do j=1, N
!              do k=1, N
!                C(i,j) = C(i,j) + A(i,k) * B(k,j)
!              enddo
!           enddo
!!oat$ install LoopFusionSplit region end

        return
        end subroutine SelectSub2_OAT
```

リージョン記述の概要

リージョン記述は、以下のように行われます。

```
!OAT$ リージョングループ　リージョン機能 region start
    対象コード
!OAT$ リージョングループ　リージョン機能 region end
```

リージョングループは以下の３つを選択します。３つの指定によって、出力先のファイ

ルと動作が変わります。

Install インストール。出力先は InstallRoutines になります。
Static スタティク。出力先は StaticRoutines になります。
Dynamic ダイナミック。出力先は DynamicRoutines になります。

リージョン記述部分のコードの変化
　指定されたリージョン部分は OAT_パラメータセットサブルーチンと OAT_呼び出しコード選択サブルーチンの２つの呼び出しに変換されます。

OAT_Source.f のリージョン部分（パラメータセットと選択コード呼び出しに変換）

```
    Call OAT_SetParam(1,Name,N,iusw1)　！　パラメータセットの呼び出し
    Call OAT_InstallSelectSub(N,A,iusw1)　！　パラメータによる選択コードの呼び出し
```

OAT_ControlRoutines.f の OAT_SetParam によるパラメータの読込
　種類と名前とベースパラメータから、パラメータを求めて isw 変数にセットします。
　通常、パラメータはデータファイルから読み込んで決定されます。 ただし、リージョン種類がダイナミックの場合はデータファイルからの読込でなく、呼び出しごとに計算されます。

```
!       === OAT_SetParm
!       ============================================================
        subroutine OAT_SetParm(OAT_TYPE, OAT_Routine, n_bpset, isw)
        integer OAT_TYPE　！リージョングループ番号
        character*85 OAT_Routine ! リージョン名
        integer n_bpset , isw　！ベースパラメータとパラメータ変数

        include 'OAT.h'

!       ファイルからのパラメータ読込みコード
```

OAT_InstallRoutines.f の OAT_InstallSelectSub による選択コードの呼び出し
　パラメータ変数 iusw1 の値によって選択されたコードが実行されます。
　この例の場合はサブルーチン呼び出しのため別名の_OAT に複写したサブルーチンが呼

ばれます。呼び出すサブルーチンも InstallRoutines.f に複写されます。

```
      subroutine OAT_InstallSelectMatMul2( N, A, iusw1)
      real*8 A(N,N),
      integer N
      integer iusw1

      select case(iusw1)
        case(1)
          call Select1_OAT(A,N)      ！パラメータによって選択されるコード
        case(2)
          call Select2_OAT(A, N)     ！パラメータによって選択されるコード
      end select

      return
      end
!
! 名前を _OAT に変更して複写したサブルーチン 1
!
      subroutine Select1_OAT(A, N)
      integer N
      real*8   A(N,N),
          ！実行コード
      return
      end
!
! 名前を _OAT に変更して複写したサブルーチン 2
!
      subroutine Select2_OAT(A, B, C, N)
      integer N
      real*8   A(N,N),
          ！実行コード
      return
      end
```

リージョン機能

リージョン機能をリージョングループ名の後に指定して、リージョンの動作を指定します。以下の１２種類の機能があります。

1. Variable 変数の値を変化させます。
2. Select 複数のコードの選択をします。選択はサブリージョンで指定します。
3. Unroll ループを展開します。
4. LoopFusion ループを併合します。
5. LoopSplit ループを分解します。
6. LoopFusionSplit　ループの併合と分解をします。
7. RotationOrder 実行順を入れ替えます。
8. List 指定した変数を順番に変化させます。
9. VariableD　指示文の中の値を変化させます。
10. ListD 指示文の中の指定した変数を順番に変化させます。
11. Replace 指定した文字列と一致する行の一部を置き換えます。
12. GWV Gang/Worker/Vector の値を順番に変化させます。

## オプション

以下のオプションが使用可能です。オプションを ABC 順に説明します。

-cc=pgi

allocate 記述の対象を pgi に指定します。条件が一致したリージョンの先頭と最後に以下の行が挿入されます。

```
#pragma acc region
{ // #pragma allocate region start
・・・
} // #pragma allocate region end.
```

-cc= omp-cuda

allocate 記述の対象を omp-cuda に指定します。条件が一致したリージョンの先頭と最後に以下の行が挿入されます。

```
#pragma OMPCUDA gpu region
{ // #pragma allocate region start.
・・・
} // #pragma allocate region end.
```

-debug on

デバッグ情報の出力状態を変更します。on の前には空白が必要です。

デバッグ on が設定された場合には以下の行が挿入されます。

```
if (OAT_DEBUG .ge. 1)then
  print *, "N=",iloop_n, "BestSw=",iBestSW1
endif
```

-eecnt

オプションとして指定することで以下の行を挿入または置換えます。

```
Integer OAT_Eecntl_NextIndex
logical OAT_Eecntl_Continue
・・・
call OAT_Eecntl_Init(F1,ケース数)
  iBestSw1 = 0
  do while ( .true.)
    iusw1 = OAT_Eecntl_NextIndex()
    if (iusw1 < 1 ) exit
・・・
    iloop_inner = 0
    do while (OAT_Eecntl_Continue())
・・・
      iloop_inner = iloop_inner + 1
    end do
・・・
    t_all = bt / iloop_inner
    call OAT_Eecntl_Repperf(iusw1, t_all)
・・・
    if (iBestSw1 == 0) then
      dBestTime1 = t_all
      iBestSw1 = iusw1
・・・
    call OAT_Eecntl_Fin
```

-free

オプションとして指定することで拡張子と無関係に自由形式 Fortran として扱います。入力ファイル名に対応した出力ファイルの拡張子は元のままですが、それ以外の拡張子は.f90 になります。

-fixed

オプションとして指定することで拡張子と無関係に固定形式 Fortran として扱います。入力ファイル名に対応した出力ファイルの拡張子は元のままですが、それ以外の拡張子は.f になります。

-gcforce

オプションとして指定することでコード生成エラーが発生しても以後の生成を継続して行います。

-insert_module_head = <対象モジュールを含むファイル名>

指定したファイルに対して use 文を SOUBROUTINE 内でなくモジュールの先頭部分に挿入します。挿入位置は、MODULE 文の後のコメントと USE 文以外の文の前になります。サブルーチン呼び出しの場合には、サブルーチン名_OAT を使うようになったため不要になりました。指定がなくても SOUBROUTINE 内でなくモジュールの先頭部分に USE が置かれます。

-mpi

オプションとして指定する事で MPI に対応した以下のコードが複数の箇所に挿入されます。一部のコードは Fitting が有効の場合のみ挿入されます。

```
Use mpi
・・・
call MPI_BCAST(OAT_EXEC_Env,8,MPI_CHARACTER,0,MPI_COMM_WORLD,ierr)
・・・
call MPI_BCAST(OAT_EXEC_Env,8,MPI_CHARACTER,0,MPI_COMM_WORLD,ierr)
・・・
call MPI_BCAST(a_lsm,(OATLSM_MAX_M+1)*OATLSM_MAX_NPARM,
  MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
・・・
call MPI_BCAST(isw,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
・・・
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
・・・
```

```
call MPI_ALLREDUCE(t_all, bt, 1, MPI_DOUBLE_PRECISION,MPI_MAX,
  MPI_COMM_WORLD, ierr)
・・・
```

-nompi

オプションとして指定することで call の後が MPI で始まるサブルーチン名の呼び出しの行を削除します。

```
call MPI_BCAST( ・・・  等の行が削除されます。
```

-omp

このオプションは、現在使われていません。指定がなくても OAT_Wtime = omp_get_wtime()を使用しています。ただし、-mpi オプションがあればそちらが優先されたコードになります。

-omp_outer

このオプションは、現在使われていません。指定がなくても OAT_Wtime = omp_get_wtime()を使用しています。ただし、-mpi オプションがあればそちらが優先されたコードになります。

-omp_inner

オプションに-omp_inner を指定する事で、OMP Threads 中からの OAT_ATexec 呼び出し対応したコードが生成されます。以下のコードが複数の箇所に挿入されます。

```
integer   oat_mythread_num
common   /OAT_OMPval/oat_mythread_num
!$omp threadprivate(/OAT_OMPval/)
・・・
  if (oat_mythread_num .eq. 0) then
・・・
    !$omp flush(OAT_Routines,OAT_DYNAMICTUNE)
    !$omp flush(iusw1_チューニングリージョン名)
  endif
!$omp barrier
・・・
!$omp flush(isw)
・・・
endif
```

```
!$omp barrier
```

-stime_and_etime start_time stop_time

オプションとして指定する事で時間測定の開始と終了で使用される関数を指定出来ます。スペースで区切られた start_time が開始時に呼ばれるサブルーチン名で、その次の引数の stop_time が終了時に呼ばれるサブルーチン名になります。

-time TimeFuncName

オプションとして指定する事で時間測定に使用される関数を指定出来ます。スペースで区切られた TimeFuneName 名のサブルーチンが実行時間の所得に使用されます。

- visualization on

結果のビジュアル表示を変更します。on の前には空白が必要です。

## 現状の ver1.0.0 のドキュメント（英文）に記載のない部分

List 指定した変数を順番に変化させます。

!OAT$ list(j,k) with (1,2) (2,3) (3,4) (4,5)のように指定することで変数 j と k の値が順番に変化します。With の後のリストは文字として扱われるため数字だけでなく(a1,b1)等の記述も可能です。

```
!OAT$ static list(j,k) region start
!OAT$ name MyList
!OAT$ list(j,k) with (1,2) (2,3) (3,4) (4,5)
        do i=1,n
            dst(i) = src(i) * dble(j)
            dst(i) = dst(i) * dble(k)
        enddo
!OAT$ static variable region end
```

```
        subroutine OAT_StaticMyList(n, dst, src, j, k, iusw1)
        integer n
        double precision dst(n), src(n)
        integer j, k
        integer iusw1

        integer i

        select case(iusw1)
          case(1)
            do i=1,n
            dst(i) = src(i) * dble(1)
            dst(i) = dst(i) * dble(2)
        enddo
          case(2)
            do i=1,n
            dst(i) = src(i) * dble(2)
            dst(i) = dst(i) * dble(3)
        enddo
          case(3)
            do i=1,n
```

```
            dst(i) = src(i) * dble(3)
            dst(i) = dst(i) * dble(4)
        enddo
          case(4)
            do i=1,n
            dst(i) = src(i) * dble(4)
            dst(i) = dst(i) * dble(5)
        enddo
        end select

        return
        end
```

VariableD 指示文の中の値を変化させます。

Fortran でのコメントや C での#Pragma の中の値を変化させます。指示文の中の一致する変数の値を順番に変化させたコードを生成します。from 1 to 5 step 2 等のように step を指定することも出来ます。

```
!oat$ static variableD(col) region start
!oat$ name variable_in_directive
!$acc kernels
!oat$ variedD(col) from 1 to 2
!$acc loop collapse(col)
        do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
!oat$ static variableD region end
```

```
        subroutine OAT_Staticvariable_in_directive(N, C, A, B, iusw1)
        integer N
        double precision C(N,N), A(N,N), B(N,N)
        integer iusw1
```

```
        integer i, j, k

        select case(iusw1)
          case(1)
!$acc kernels
!$acc loop collapse(1)
             do i=1, N
            do j=1, N
               do k=1, N
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
               enddo
            enddo
        enddo
!$acc end kernels
          case(2)
!$acc kernels
!$acc loop collapse(2)
             do i=1, N
            do j=1, N
               do k=1, N
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
               enddo
            enddo
        enddo
!$acc end kernels
        end select

        return
        end
```

ListD 指示文の中の指定した変数を順番に変化させます。

Fortran でのコメントや C での#Pragma の中の値を変化させます。指示文の中の一致する変数の値を順番に変化させたコードを生成します。

```
!oat$ static listD(G,V) region start
!oat$ name variable_in_directive
!$acc kernels
!oat$ listD(G,V) with (15,32) (15,64) (30,64) (60,128)
!$acc loop gang(G) vector(V)
        do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
!oat$ static listD region end
```

```
        subroutine OAT_Staticvariable_in_directive(N, C, A, B, iusw1)
        integer N
        double precision C(N,N), A(N,N), B(N,N)
        integer iusw1

        integer i, j, k

        select case(iusw1)
           case(1)
!$acc kernels
!$acc loop gang(15) vector(32)
              do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
           case(2)
!$acc kernels
```

```
!$acc loop gang(15) vector(64)
              do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
           case(3)
!$acc kernels
!$acc loop gang(30) vector(64)
              do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
           case(4)
!$acc kernels
!$acc loop gang(60) vector(128)
              do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
        end select

        return
        end
```

Replace 指定した文字列と一致する行の一部を置き換えます。文字列が一致する行があれば、一致しない中間部分を指定した内容に順番に置き換えたコードを生成します。Replace には対象を指示文とするか指示文以外にするかの区別はありません。

```
!OAT$ static replace region start
!OAT$ name prefetch_2
!OAT$ target !dir$ prefetch P:1:
!OAT$ target !dir$ prefetch Q:1:
!OAT$ target !dir$ prefetch D:1:
!OAT$ target !dir$ prefetch AU:1:
!OAT$ target !dir$ prefetch itemU:1:

!OAT$ replace (2) (3) (4) (5) (6) (8) (16)

!$omp parallel private(ic,ip,ip1,ip2,i,k,ip0,ib0)
        do ic= 1, NCOLORtot
          if (ic.eq.1) then
!$omp do
!dir$ prefetch P:1:2
!dir$ prefetch Q:1:2
!dir$ prefetch D:1:2
!$omp simd

!OAT$ static replace region end
```

```
        select case(iusw1)
          case(1)
!$omp parallel private(ic,ip,ip1,ip2,i,k,ip0,ib0)
            do ic= 1, NCOLORtot
          if (ic.eq.1) then
!$omp do
!dir$ prefetch P:1:2
!dir$ prefetch Q:1:2
!dir$ prefetch D:1:2
!$omp simd

          case(2)
```

```
!$omp parallel private(ic,ip,ip1,ip2,i,k,ip0,ib0)
              do ic= 1, NCOLORtot
            if (ic.eq.1) then
!$omp do
!dir$ prefetch P:1:3
!dir$ prefetch Q:1:3
!dir$ prefetch D:1:3
!$omp simd
```

GWV Gang/Worker/Vector の値を順番に変化させます。!oat$ GWV-list label で指定した Label と一致する!oat$ GWV-target label が見つかった場合に、その次の行が対象になります。自然数の場合はその形状のサイズに使う、0 の場合はその形状を使わない、その他（文字無しを含む）の場合はデフォルト設定になります。

```
!oat$ static GWV region start
!oat$ name GWV
!oat$ GWV-list label (32,2,32) (20,4,64) (,x,0) (-,0,64)
!$acc kernels
!oat$ GWV-target label
!$acc loop gang(30) worker(2) vector(64)
        do i=1, N
            do j=1, N
                do k=1, N
                    C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
            enddo
        enddo
!$acc end kernels
!oat$ static GWV region end
```

```
        subroutine OAT_StaticGWV(N, C, A, B, iusw1)
        integer N
        double precision C(N,N), A(N,N), B(N,N)
        integer iusw1
```

```
        integer i, j, k

        select case(iusw1)
          case(1)
!$acc kernels
!$acc loop gang(32) worker(2) vector(32)
            do i=1, N
            do j=1, N
              do k=1, N
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
              enddo
            enddo
        enddo
!$acc end kernels
          case(2)
!$acc kernels
!$acc loop gang(20) worker(4) vector(64)
            do i=1, N
            do j=1, N
              do k=1, N
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
              enddo
            enddo
        enddo
!$acc end kernels
          case(3)
!$acc kernels
!$acc loop gang worker
            do i=1, N
            do j=1, N
              do k=1, N
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
              enddo
            enddo
        enddo
```

```
!$acc end kernels
          case(4)
!$acc kernels
!$acc loop gang vector(64)
             do i=1, N
            do j=1, N
               do k=1, N
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
               enddo
            enddo
         enddo
!$acc end kernels
         end select

         return
         end
```

OAT/OAT.h ファイルについて

生成される OAT/OAT.h は、ソースファイルと同じフォルダにある OAT.h ファイルの後ろに以下のテキストが追加されたファイルになります。

リージョン用変数の開始を示すコメント

```
!          === AT region variables
```

オプションに-omp_inner が設定されている場合に追加

```
integer   oat_mythread_num
common   /OAT_OMPval/oat_mythread_num
!$omp threadprivate(/OAT_OMPval/)
```

各リージョンの iusw1 変数（以下の例はリージョンが 2 つの場合）

```
integer iusw1_リージョン名 1
integer iusw1_リージョン名 2
common /OAT_ATswitches/ iusw1_リージョン名 1 , iusw1_リージョン名 2
```

各リージョンの iusw1_flag 変数（以下の例はリージョンが２つの場合）

```
integer iusw1_リージョン名 1_flag
integer iusw1_リージョン名 2_flag
common /OAT_ATswitches/ iusw1_リージョン名１_flag, iusw1_リージョン名２_flag
```

!OAT$ allocate (auto)　記述時の動作

ABCLibScript における GPU 実行ディレクティブ機能 平成 22 年 12 月 24 日 によって追加された機能です。その時点でＣ言語のみが対象でしたが Fortran についても同じ出力が行われます。現状では!$acc や!$omp でなく#pragma の出力になっています。Select と Unroll の２種類のリージョンに対応しています。

!OAT$ allocate (auto)

-cc=オプションで pgi または omp-cuda が指定されていれば、case 数を２倍にして通常コードとリージョンの前後にコードを追加した２種類のコードを生成します。

- !OAT$ allocate (cpu)

  通常コードが生成されます。

- !OAT$ allocate (gpu)

  -cc=オプションで pgi または omp-cuda が指定されていれば、リージョンの前後にコードを追加した２種類のコードを生成します。

-cc=pgi の場合に追加されるコード

```
#pragma acc region
{ // #pragma allocate region start
・・・
} // #pragma allocate region end.
```

-cc= omp-cuda の場合に追加されるコード

```
#pragma OMPCUDA gpu region
{ // #pragma allocate region start.
・・・
} // #pragma allocate region end.
```

元となる仕様

- ディレクティブ　#pragma ABCLib allocate (auto)　記述時の動作

■オリジナルコード

```
#pragma ABCLib install unroll (i,j,k) region start
#pragma ABCLib name MyMatMul
#pragma ABCLib varied (i,j,k) from 1 to 4
#pragma ABCLib allocate (auto)
#pragma omp parallel for
        do k = 1,n1
         do i = 1,n3
          c(i,k) = 0.0
          do j = 1,n2
           c(i,k) = c(i,k) + a(i,j) * b(j,k)
          enddo
         enddo
        enddo
#pragma ABCLib install unroll (i,j,k) region end
```

以上の例に対し、通常の ABCLibScript の生成コード（4×4×4＝64 通り）に加え、以下のコメントを追加したものを、64 通り生成する機能。

■アンローリングなしのコード例１（オプション “-PGI” 設定時）

```
#pragma acc region
        do k = 1,n1
         do i = 1,n3
          c(i,k) = 0.0
          do j = 1,n2
           c(i,k) = c(i,k) + a(i,j) * b(j,k)
          enddo
         enddo
        enddo
```

■アンローリングなしのコード例２（オプション “-OMP-CUDA” 設定時）

```
#pragma OMP-CUDA acc region
```

```
#pragma omp parallel for
        do k = 1,n1
         do i = 1,n3
          c(i,k) = 0.0
          do j = 1,n2
           c(i,k) = c(i,k) + a(i,j) * b(j,k)
          enddo
         enddo
        enddo
```

２．「#pragma ABCLib allocate」の書式

追加機能に関するディレクティブの書式は以下である。

```
#pragma ABCLib allocate (<Object>)
      <Object> := {CPU | GPU | auto }
          CPU : CPU 上での実行
          GPU : GPU 上での実行
          auto: CPU 上での実行、GPU 上での実行を試し、最も高速な環境で実行させる
```

引数が GPU のとき、１節のコード例１、もしくは、コード例２の処理がなされる。

引数が auto のとき、従来の処理＋GPU 指定時の処理の２種のコードが混合されて自動生成される。CPU 指定時は、従来の処理のみとする。

なお、１節のコード例１、および、コード例２の切り替えオプションについて、次節で定める。

３．オプションの指定

ABCLibCodeGen に実行時オプションとして指定する、1 節のコード例１、もしくは、コード例２の切り替えオプションの書式は以下である。

```
-CC=<Object>
          <Object> := {   PGI   | OMP-CUDA   }
```

利用例:

```
$ ABCLibCodeGen   -CC=PGI   MatMat.f
```

自動チューニング制御変数

OAT_Exec 関数がコールされた時、OAT_EXEC 環境変数の中身が 1 であるとき自動チューニングを行います。それ以外で OAT_ATEXEC 環境変数の中身が 1 である時も、自動チューニングを行います。それ以外は自動チューニングを行わずに戻ります。これらの処理は OAT_ATexec 関数内の最初の処理として行っています。

元となる仕様

●自動チューニング制御変数の導入

・環境変数の導入

旧 ABCLib_ATexec 関数がコールされたとき、
以下の環境変数の中身が 1 であるときに、自動チューニングを
行う。それ以外は、自動チューニングを行わずに戻る。

OAT_EXEC

・AT 実行を制御する予約語

以下の変数の中身が 1 であるとき、旧 ABCLib_ATexec 関数がコール
されるとき、自動チューニングを行う。それ以外は、自動チューニング
を行わずに戻る。

OAT_ATEXEC

・環境変数 OAT_EXEC と予約語変数 OAT_ATEXEC の
自動チューニング実行に関する優先度は、
環境変数 OAT_ATEXEC のほうが高い。

複数ファイルに記載された AT 領域については、引数に並べた複数のファイルの順番で読込処理を行っています。OAT.h,OAT_ControlRoutines.f90 等のファイルは複数のファイルがあっても 1 つにまとめられます。複数ファイルごとの OAT_元ファイル名.f90 について複数ファイルが作成されます。

元となる仕様

●複数ファイルに記載された AT 領域について

・ppOpenAT のプリプロセッサ起動時のカレントディレクトリ以下の
すべてのファイルについて、ppOpen-AT のプラグマ記述を
探索して処理ができること。
・登録については、見つけた順番に登録すること。
・自動チューニングの実行順序については、見つけた AT 領域の順番に行うこと。

なお本仕様では、AT 領域のプログラム上の深さ、つまり、別ファイルの
プログラムにおいて AT 領域が入れ子になっているときに、プログラム上
の入れ子構造について考えなくて良い。

環境変数 OAT_PATH にパスが設定されている場合、その指定パスとカレントディレクトリの双方のファイルについて処理が行われます。処理がパスを変えて２回行われる事になります。

元となる仕様

●処理を行うプログラムの指定方法

以下の環境変数にパスが設定されている場合、その指定パスにおかれた
ファイルと、カレントディレクトリにあるファイルの双方について
処理をすること。

OAT_PATH

!OAT$ call OAT_BPSet(“N”)もしくは#pragma OAT OAT_BPset("N")等の BP の指定をしない場合はエラーになります。!OAT$ call OAT_BPset("none")　もしくは#pragma OAT OAT_BPset("none")でのＢＰを設定しない指定になります。

元となる仕様

●基本パラメタ指定について

!OAT$ call OAT_BPset("N")

もしくは

#pragma OAT OAT_BPset("N")

などの、BP の指定をしない場合は、エラーで止まる仕様とすること。

また、BP を設定しない場合については、以下の指定法を拡張する。

!OAT$ call OAT_BPset("none")

もしくは

#pragma OAT OAT_BPset("none")

!OAT$ bind ATexec arguments start と!OAT$ bind ATexec arguments end による呼び出し時の引数の置換を行います。

1） !OAT$ bind ATexec arguments start が見つかったら、!OAT$ bind ATexec arguments end までの間を引数変換用の定義範囲とします。

2） 定義範囲内の !OAT$ については、元の変数名=新しい変数名 の置換え定義として扱うため、通常の !OAT$ コマンドでの動作や出力は行われません。

3） ATexec 呼び出しがあった場合に、その直前の !OAT$ bind ATexec arguments start を探して、引数の変数に関して定義されている変数名への置き換えを行って出力します。

元となる仕様

■ATexec 関数で、プリプロセッサにより自動生成される引数の宣言のための指示子の拡張

例)

```
!OAT$ bind ATexec arguments start
!OAT$ V = VZ
!OAT$ DXV = DXVX
!OAT$ DYV = DYVY
!OAT$ DZV = DZVZ
```

!OAT$ bind ATexec arguments end

○動作

上記の　!OAT$ bind ATexec arguments start ～!OAT$ bind ATexec arguments end

で囲まれた変数について、OAT_ATexec 中のオリジナルの変数名を、記述のとおり置き換える

○置き換え前：

```
call
OAT_ATexec(OAT_INSTALL,OAT_InstallRoutines,NZ00,NZ01,NY00,NY01,NX00,NX0
1,NX0,NX1,N¥
Y0,NY1,NZ0,NZ1,DX,NZ,NY,NX,DXV,V,DY,DY&
      &V,DZ,DZV)
```

○置き換え後：

```
call
OAT_ATexec(OAT_INSTALL,OAT_InstallRoutines,NZ00,NZ01,NY00,NY01,NX00,NX0
1,NX0,NX1,N¥
Y0,NY1,NZ0,NZ1,DX,NZ,NY,NX,DXVX,VZ,DY,DY&
      &VY,DZ,DZVZ)
-----
```

!OAT$ RotationOrder による式順の置き換え

!OAT$ RotationOrder sub region start と!OAT$ RotationOrder sub region end の間の式順を置き換えたコードとそのままのコードの２種類のコードを生成します。RotationOrder sub region は２つ以上必要です。１つめの sub region の行が ＡＢＣの３行で２つめの sub region の行が ａｂｃ の３行だった場合には、ＡＢＣａｂｃ の並びと ＡａＢｂＣｃ の並びの２種類の出力が生成されます。ケース数は RotationOrder sub region がない場合の２倍になります。

元となる仕様

- 式順入れ替えのための副指定子

以下の仕様を持つ、式順入れ替えのための副指定子の機能を実装すること。

```
!OAT$ RotationOrder sub region start
  式の並び
!OAT$ RotationOrder sub region end
```

なお、上記の副指定子はＡＴ領域内に 2 つ以上記載されている場合に、有効となる。
たとえば、以下の例では

```
!OAT$ RotationOrder sub region start
  A
  B
  C
!OAT$ RotationOrder sub region end

!OAT$ RotationOrder sub region start
  a
  b
  c
!OAT$ RotationOrder sub region end
```

想定される出力は以下となる。

```
A
a
B
b
C
c
```

以上を、RotationOrder 副指定子を記載する。

RotationOrder 副指定子は、基のコードそのままと、基のコードに RotationOrder 副指定子で指定される最適化の 2 種を調べるための専用の指定子 RatationOrder 指定子の機能を提供すること、すなわち、以下のように記載できること。

```
!OAT$ install RotationOrder region start
!OAT$ name RotationTest
!OAT$ debug (pp)
```

```
!OAT$ RotationOrder sub region start
  A
  B
  C
!OAT$ RotationOrder sub region end

!OAT$ RotationOrder sub region start
  a
  b
  c
!OAT$ RotationOrder sub region end
!OAT$ install RotationOrder region end
```

RotationOrder 副指定子は、任意のループ中にも記載できること。

RotationOrder 副指定子は、LoopFusion 指定子、LoopSplit 指定子、LoopFusionSplit 指定子の内部にも記載できること。この場合、内部の演算について、基のコードに対して指定される処理（例えば LoopFusion）による候補に加え、RotationOrder 副指定子による変更された内部演算について、指定される処理を施すこと。したがって、通常の指定子の候補に加えて、2 倍の候補を生成することになる。

例えば、以下の例では

```
!OAT$ install LoopFusion region start
!OAT$ name ppohFDMpassing_velocity
!OAT$ debug (pp)
    do k = NZ00, NZ01
       do j = NY00, NY01
          do i = NX00, NX01
             ! Effective Density
!OAT$ RotationOrder sub region start
             ROX = 2.0_PN/( DEN(I,J,K) + DEN(I+1,J,K) )
             ROY = 2.0_PN/( DEN(I,J,K) + DEN(I,J+1,K) )
             ROZ = 2.0_PN/( DEN(I,J,K) + DEN(I,J,K+1) )
!OAT$ RotationOrder sub region end
!OAT$ RotationOrder sub region start
             VX(I,J,K) = VX(I,J,K) &
```

```
                        + ( DXSXX(I,J,K)+DYSXY(I,J,K)+DZSXZ(I,J,K) )*ROX*DT
            VY(I,J,K) = VY(I,J,K) &
                        + ( DXSXY(I,J,K)+DYSYY(I,J,K)+DZSYZ(I,J,K) )*ROY*DT
            VZ(I,J,K) = VZ(I,J,K) &
                        + ( DXSXZ(I,J,K)+DYSYZ(I,J,K)+DZSZZ(I,J,K) )*ROZ*DT
!OAT$ RotationOrder sub region end
          end do
       end do
    end do
!OAT$ install LoopFusion region end
```

以上の元のコードに対して、ループ融合処理を行う候補の生成に加えて、以下のコードに対するループ融合処理を行った候補の生成もしなくてはいけない。

```
do k = NZ00, NZ01
       do j = NY00, NY01
          do i = NX00, NX01
             ! Effective Density
             ROX = 2.0_PN/( DEN(I,J,K) + DEN(I+1,J,K) )
             VX(I,J,K) = VX(I,J,K) &
                       + ( DXSXX(I,J,K)+DYSXY(I,J,K)+DZSXZ(I,J,K) )*ROX*DT
             ROY = 2.0_PN/( DEN(I,J,K) + DEN(I,J+1,K) )
             VY(I,J,K) = VY(I,J,K) &
                       + ( DXSXY(I,J,K)+DYSYY(I,J,K)+DZSYZ(I,J,K) )*ROY*DT
             ROZ = 2.0_PN/( DEN(I,J,K) + DEN(I,J,K+1) )
             VZ(I,J,K) = VZ(I,J,K) &
                       + ( DXSXZ(I,J,K)+DYSYZ(I,J,K)+DZSZZ(I,J,K) )*ROZ*DT
          end do
       end do
    end do
```

AT 領域のチューニングパラメタ変数を大域変数化しています。これらの変数の定義を追加した OAT.h を生成します。OAT.h の最後に変数定義を追加した OAT.h を生成する形になっています。

元となる仕様

●ＡＴ領域が呼ばれるたびに、ファイルからチューニングパラメタを読み出すオーハーヘッドが大きい。呼ばれた最初の１回のみ、ファイルから読み出すのを、デフォルトとする。

※ただし、実行時最適化のためにこれを無効にする機能は後日、必要になる可能性あり。

そのため、チューニングパラメタ変数を大域変数化する必要あり。

実装

1）OAT.h 中に、ファイル読み出しフラグを自動設定

AT 領域名＋"_flag" の integer 型

例）

```
integer iusw1_ppohFDMupdate_stress_flag
integer iusw1_ppohFDMupdate_sponge_flag
```

以上に加えて、パラメタ変数も OAT.h 中に宣言を自動生成

```
integer iusw1_ppohFDMupdate_stress
integer iusw1_ppohFDMupdate_sponge
```

以下、common 変数とする

```
!       === AT region variables
        integer iusw1_ppohFDMupdate_stress
        integer iusw1_ppohFDMupdate_sponge
         common /OAT_ATswitches/iusw1_ppohFDMupdate_stress,                &
        &                       iusw1_ppohFDMupdate_sponge
        integer iusw1_ppohFDMupdate_stress_flag
        integer iusw1_ppohFDMupdate_sponge_flag
         common /OAT_ATswitchFlags/iusw1_ppohFDMupdate_stress_flag,        &
        &                          iusw1_ppohFDMupdate_sponge_flag
```

2）subroutine OAT_ATset 中で 0 に初期化

例）

```
    iusw1_ppohFDMupdate_stress_flag = 0
```

3）subroutine OAT_SetParm 中で、上記フラグが"1"

でない場合は、ファイル読み出して、内容を"1"に変更。

その上で、データを読み出す。

AT ルーチンはモジュール化して利用されます。通常は SOUBROUTINE ごとに use が挿入されますが、-insert_module_head=<対象モジュールを含むファイル名>が指定された

ファイルに対しては use 文を SOUBROUTINE 内でなくモジュールの先頭部分に挿入します。

元となる仕様

●AT ルーチンのモジュール化
以下の ppOpen-AT ルーチンはモジュール化して利用
OAT_ControlRoutines.f90
　モジュール名：ppohAT_ControlRoutines
OAT_InstallRoutines.f90
　モジュール名：ppohAT_InstallRoutines
OAT_StaticRoutines.f90
　モジュール名：ppohAT_StaticRoutines
OAT_DynamicRoutines.f90
　モジュール名：ppohAT_DynamicRoutines
　以上に伴い、AT 領域中で使われている use モジュールを、AT_InstallRoutines.f90 中でも use 宣言しないと、変数などの引継ぎができなくなる。
　また、ＡＴ領域内で使われて宣言されている変数は、呼び出し先でコメントアウトする必要がある。
　例)

```
!    integer :: i, j, k
!    real(PN) :: ROX, ROY, ROZ
```

ＡＴ領域の切り替え確認用の表示は､デバックレベルを 2 以上にしています｡また､ATexec 時に AT 領域名が表示されます。

元となる仕様

●debug 用のプリントレベル
　ＡＴ領域の切り替え確認用の表示は、デバックレベルを 2 以上にする。
　　例)

```
    if (OAT_DEBUG .ge. 2)then
          print *, 'oat_myid: ',oat_myid
          print                *,                'Install                Routine:
ppohFDMupdate_stress=',iusw1_ppohFDMupdate_stress
        endif
```

●ATexec 時に、どのＡＴを行っているかわかるようにするため、

```
  AT 領域名を表示するようにする。
例)
    OAT_ControlRoutines.f90 中の記述
    if (OAT_DEBUG .ge. 1)then
       if (oat_myid .eq. 0) then
          print *, "AT region: ppohFDMupdate_stress"
        endif
    endif
```

ATのログをファイル名　チューニングパラメタ名+"TuneLog.dat"のファイルに落とします。

元となる仕様

```
●AT ログ出力機能
  AT のログを、ファイルに落とす。
  ファイル名は、チューニングパラメタ名+"TuneLog.dat"
  例)
  OAT_ControlRoutines.f90 中
    subroutine OAT_ATexec 中に
    以下のファイル操作を記入
-----
    if (oat_myid .eq. 0) then
        open(12, status = 'replace', &
     &     file = 'OAT_InstallppohFDMupdate_stressTuneLog.dat', &
     &     action = 'write', pad= 'yes')
    endif
    ...
    if (oat_myid .eq. 0) then
            write (12,"(A)") "AT region: ppohFDMupdate_stress"
    endif
    ...
    if (oat_myid .eq. 0) then
     write(12, "(A,I6,A,I6,A,F9.4,A)") "N=",iloop_n, " / iusw1=", iusw1, " : ",t_all,
" [sec.]"
    endif
    ...
    if (oat_myid .eq. 0) then
```

```
            write(12, "(A,I6,A,I6)") "N=",iloop_n, " BestSw=",iBestSW1
    endif
    ...
    if (oat_myid .eq. 0) then
        close(12, status = 'keep')
    endif
----
```

以上に加え、12 番ファイルに、書き込みとファイル終了操作を追加。

オプションに-omp_inner を指定する事で、OMP Threads 中からの OAT_ATexec 呼び出し対応したコードが生成されます。現状の生成コードは元となる仕様だけでなく、実行可能なサンプルコードを元にして作成されています。

元となる仕様

●OMP Threads 中からの OAT_ATexec 呼び出し対応

１）

ppOhBEM では、OMP スレッド中から OAT_ATexec を呼び出す必要がある。

そこで、OAT.h 中に記載される、スレッド番号を保持する変数

oat_mythread_num

を宣言する。

なおこの際、common ブロックは OAT_OMPval とし、OAT_pval を Threadprivate 化する。

```
------
          integer  oat_mythread_num
          common  /OAT_OMPval/oat_mythread_num
!$omp threadprivate(/OAT_OMPval/)
------
```

２）

上記変数適用は、OAT_ATexec、OAT_ATset、OAT_SetParm、内で利用され、0 番スレッドしか、自動生成される主要な AT 関数は利用されない。

利用は、以下のようになる

```
----
  if (oat_mythread_num .eq. 0) then
    ....
  endif
```

```
!$omp barrier
```

---

３）OAT_ATexec 中の common 変数 OAT_ATEXEC_FLAG は

　スレッド 0 番しか正しい値を持たないが、終了後、各スレッドで値を参照して終了処理をすることがある。そのため、値を flush する OMP 指示文を入れる。

---

```
     if (oat_mythread_num .eq. 0) then
      OAT_ATEXEC_FLAG = 1
      if (oat_myid .eq. 0) then
         call getenv("OAT_EXEC",OAT_EXEC_Env)
      endif
      call MPI_BCAST(OAT_EXEC_Env,8,MPI_CHARACTER,0,MPI_COMM_WORLD,ierr)
      if(TRIM(OAT_EXEC_Env) .ne. "")then
        if(TRIM(OAT_EXEC_Env) .ne. "1")then
          OAT_ATEXEC_FLAG = 0
         endif
      else
        if (oat_myid .eq. 0) then
           call getenv("OAT_ATEXEC",OAT_EXEC_Env)
        endif
        call MPI_BCAST(OAT_EXEC_Env,8,MPI_CHARACTER,0,MPI_COMM_WORLD,ierr)
        if(TRIM(OAT_EXEC_Env) .ne. "")then
          if(TRIM(OAT_EXEC_Env) .ne. "1")then
            OAT_ATEXEC_FLAG = 0
            return
          endif
        endif
      endif
!$omp flush(OAT_ATEXEC_FLAG)
      endif
!$omp barrier
      if (OAT_ATEXEC_FLAG .eq. 0) return
```

---

以上、同様に、OAT_ATset,　OAT_SetParm 中の主要変数も flush する。

４）

以上の処理は、oat のオプション ″-omp_inner″ が指定されるとき、処理される。

※ オプションなし、-mpi 指定の時は、従来通り

※ ″-omp_inner″ および ″-mpi -omp_inner″ 指定の時、上記の処理を行う

---

以上