2026/06/05 13:30-14:30 オンライン

OpenMP入門

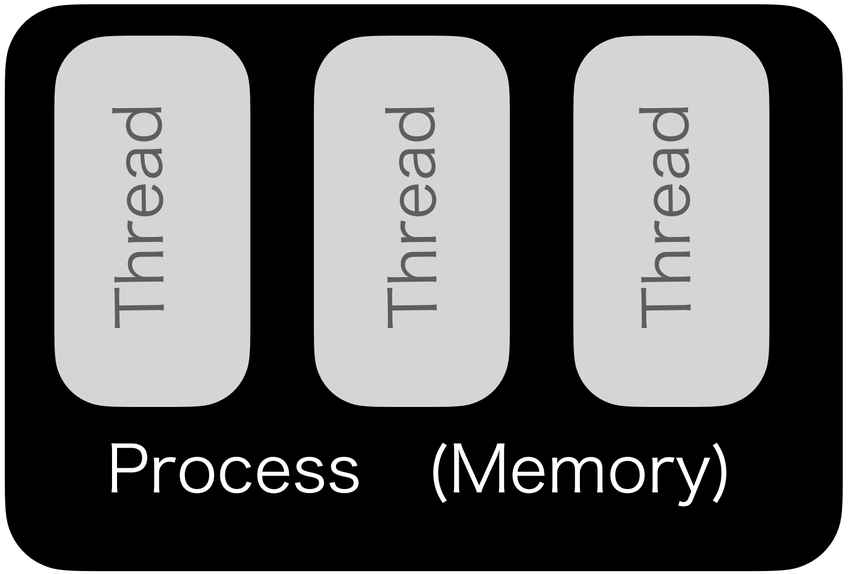
大阪大学D3センター
吉野　元

この講習会の内容

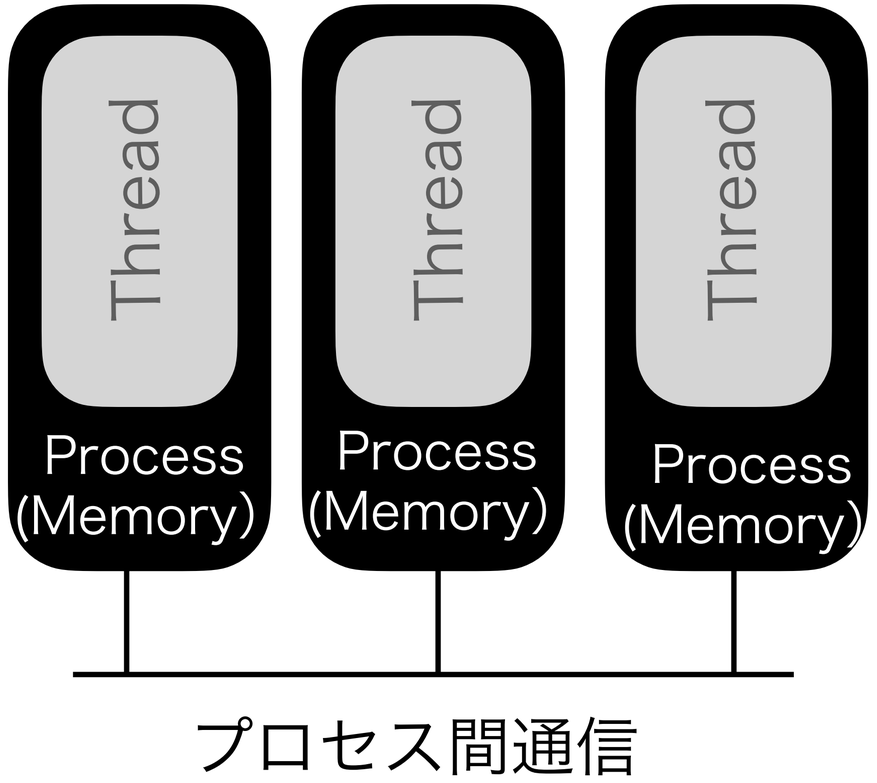
- 導入
- 演習：サンプルプログラムのコンパイル、実行、実行時間の計測
- まとめとヒント

OpenMPとは？

OpenMP/自動並列化



MPI (Message Passing Interface)



OpenMPのプロセス：一つのノード内の複数のコア上に一つずつスレッド(Thread)を作って並列に計算する。メモリーは共有。

OpenMPが使えるところ

- SQUID (汎用CPUノード群 38x2core/node GPUノード群 38x2 core(+8GPU)/node，ベクトルノード群 10x8core/node) http://www.hpc.cmc.osaka-u.ac.jp/squid/
- OCTOPUS (汎用CPUノード 140x 256 core/node) http://www.hpc.cmc.osaka-u.ac.jp/octopus2/
- お手元のマシン、例えばmacでも

ノード内並列できるのにしないのは
もったいない。。

消費ポイント = 使用ノード時間 × 消費係数 × 季節係数 × 燃料係数

http://www.hpc.cmc.osaka-u.ac.jp/system/manual/point/

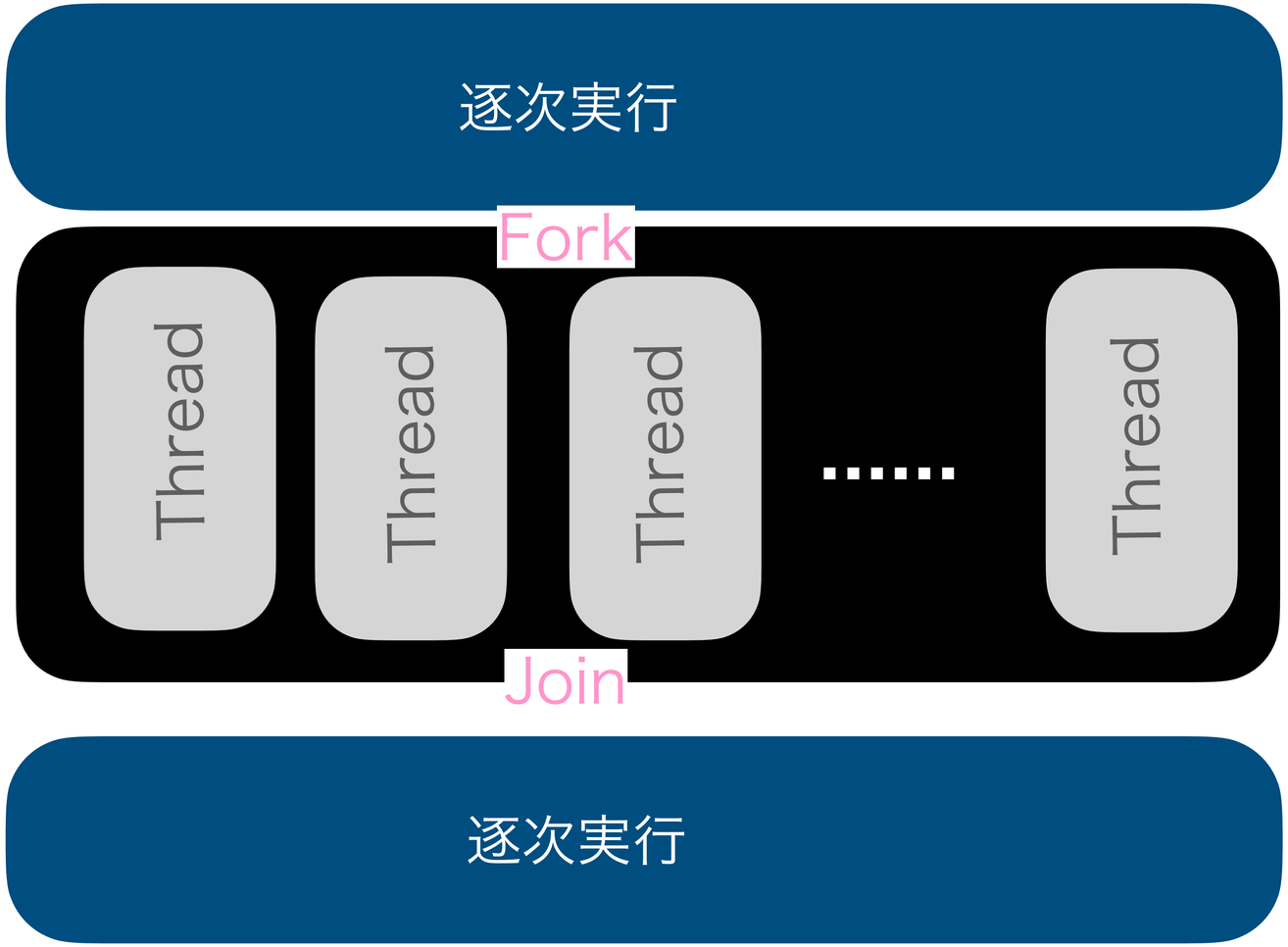
- 自動並列化、OPENMPはプログラムをほとんど変えずに簡単に試せるので試してみるべき
- MPIでもノード内並列はできる（難易度はあがる ） ベクトル化との組み合わせも可

プログラムはどう書くか？

FORTRAN
の場合

!$ omp parallel

!$ omp end parallel



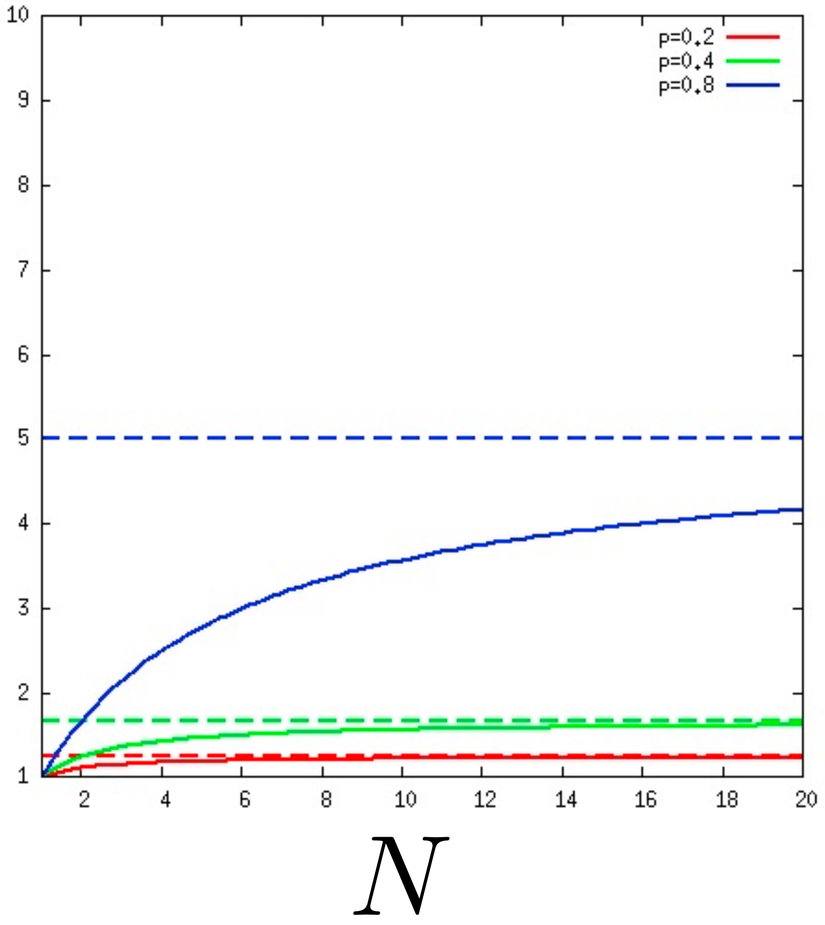
Cの場合

#pragma parallel {

}

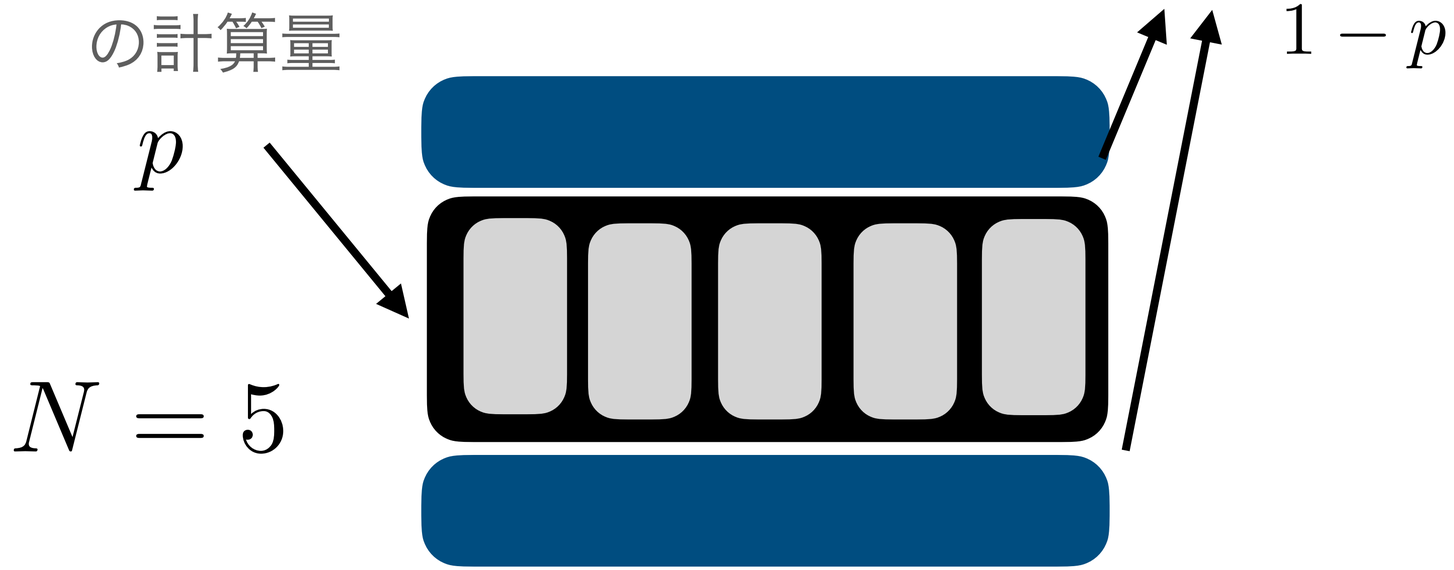
Amdahlの法則

並列化効率= $\dfrac{1}{1-p+\frac{p}{N}}$



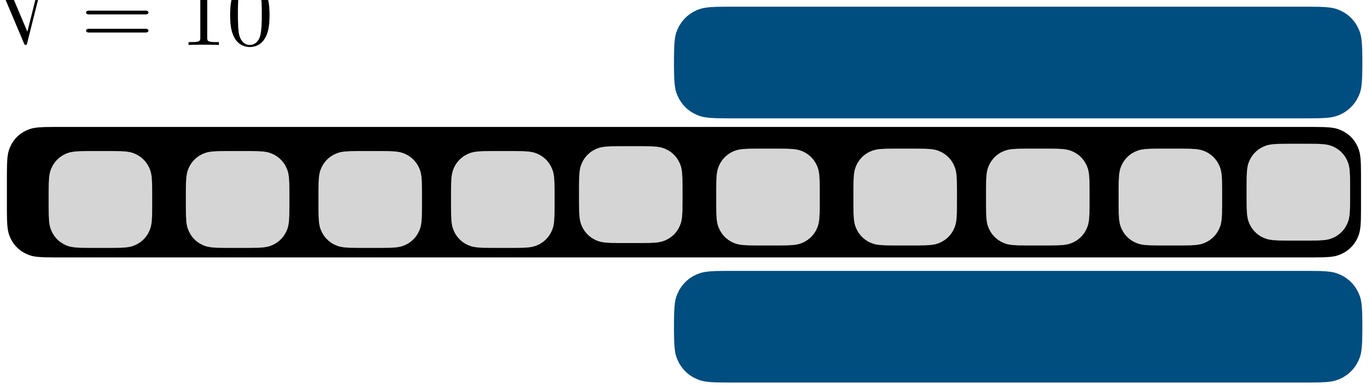
並列化できる部分の計算量 p

並列化できない部分の計算量 $1-p$



$N=5$

$N=10$



演習

- SQUIDへのlogin
- 以下のファイルを自分のディレクトリにコピーして展開(tar -xvf *.tar) してください。

  C言語　/system/lecture/OpenMP.2025/C.tar

  Fortran /system/lecture/OpenMP.2025/Fortran. tar

  中身は以下のサンプルプログラムとコンパイルスクリプト、jobスクリプトです。

- (1) hello_world
- (2) vector_sum
- (3) vector_inner_product
- (4) diffusion
- (5) reaction_diffusion

(1) hello world

fortran

```fortran
program main
  implicit none
 include "omp_lib.h"
  !$use omp_lib
  !$omp parallel
  print *, "HELLO WORLD mythread = ",omp_get_thread_num(),"(",omp_get_num_threads(),"threads)"
  !$omp end parallel
end
```

c

```c
# include <stdio.h>
# include <omp.h>

int main(){
  #pragma omp parallel
    {
    printf("HELLO WORLD mythread = %d %d threads\n",omp_get_thread_num(),omp_get_num_threads());
    }
    return 0;
}
```

コンパイル

```
 fortran      ifort  -O3 -qopenmp -qopt-report=2 -o a_helloworld_omp.out helloworld_omp.f90
   c          icc -O3 -qopenmp -qopt-report=2 -o a_helloworld_omp.out helloworld_omp.c
```

以下は同じ

```
v6a022@squidhpc3[122]% cat helloworld_omp.sh
#!/bin/bash
#PBS -q SQUID  (デバッグにはDBGが便利)
#PBS –group=kosyuXXX (講習会用のグループ名)
#PBS -l cpunum_job=10
#PBS -v OMP_NUM_THREADS=10
#PBS -l elapstim_req=00:10:00
module load BaseCPU/2021
cd $PBS_O_WORKDIR
./a_helloworld_omp.out > helloworld_omp.log
```

参考：http://www.hpc.cmc.osaka-u.ac.jp/system/manual/squid-use/jobscript/

ジョブスクリプト

環境変数　OMP_NUM_THREADSの値を変えると
並列化に使うスレッドの数が変わる。

CPUノードの場合

```
v6a022@squidhpc3[117]% qsub helloworld_omp.sh
Request 125187.sqd submitted to queue: SQUID.
```

```
v6a022@squidhpc3[121]% qstat
RequestID       ReqName  UserName Queue     Pri STT S   Memory       CPU   Elapse R H M Jobs
--------------- -------- -------- -------- ---- --- - -------- -------- -------- - - - ----
125189.sqd      hellowor v6a022   SC1          0 QUE -    0.00B     0.00        0 Y Y Y    1
```

結果

```
v6a022@squidhpc3[128]% cat helloworld_omp.log
 HELLO WORLD mythread =             0 (          10 threads)
 HELLO WORLD mythread =             2 (          10 threads)
 HELLO WORLD mythread =             3 (          10 threads)
 HELLO WORLD mythread =             6 (          10 threads)
 HELLO WORLD mythread =             9 (          10 threads)
 HELLO WORLD mythread =             1 (          10 threads)
 HELLO WORLD mythread =             5 (          10 threads)
 HELLO WORLD mythread =             4 (          10 threads)
 HELLO WORLD mythread =             7 (          10 threads)
 HELLO WORLD mythread =             8 (          10 threads)
```

(2) vector_sum

```fortran
program main
!$ include "omp_lib.h"
  integer,parameter :: num_size=100000

  real*8 :: st,en

  real*8 :: a(num_size),b(num_size),c(num_size)

  time40=etime(tarray0)

  call random_number(a)
  call random_number(b)

  st = omp_get_wtime()

  !$omp parallel do
  do i=1,num_size
     c(i)=a(i)+b(i)
  end do
  !$omp end parallel do

  en = omp_get_wtime()
  print *, "Elapsed time in second is:", en-st
end program main
```

fortran

```c
# include <stdio.h>
# include <omp.h>

int main(){
  const int num_size=1000;
  double a[num_size],b[num_size],c[num_size];
  int i;
  double st, en;

  for(i=0;i<num_size;i++){a[i]=rand()/pow(2,31);};
  for(i=0;i<num_size;i++){b[i]=rand()/pow(2,31);};

  st = omp_get_wtime();

  #pragma omp parallel for
  for(i=0;i<num_size;i++){
    c[i]=a[i]+b[i];
  };

  en = omp_get_wtime();
  printf("Elapsed time in second is: %f\n", en-st);
  return 0;
}
```

c

(3)vector_inner_product

総和の計算

```fortran
program main
!$ include "omp_lib.h"
  integer,parameter :: num_size=100000

  real*8 :: st,en
  real*8 :: a(num_size),b(num_size),s

  call random_number(a)
  call random_number(b)

  st = omp_get_wtime()

  s=0.d0
  !$omp parallel do reduction(+:s)
  do i=1,num_size
     s=s+a(i)*b(i)
  end do
  !$omp end parallel do

  en = omp_get_wtime()
  print *, "Elapsed time in second is:", en-st

end program main
```

fortran

```c
# include <stdio.h>
# include <omp.h>

int main(){
  const int num_size=100000;
  double a[num_size],b[num_size];
  int i;
  double st, en;
  double s;

  for(i=0;i<num_size;i++){a[i]=rand()/pow(2,31);};
  for(i=0;i<num_size;i++){b[i]=rand()/pow(2,31);};

  st = omp_get_wtime();

  s=0.0;
#pragma omp parallel for reduction(+:s)
  for(i=0;i<num_size;i++){
    s=s+a[i]*b[i];
  };

#pragma omp barrier

  en = omp_get_wtime();
  printf("Elapsed time in second is: %f\n", en-st);

    return 0;
}
```

c

Private変数/Shared変数

デフォルトではShared変数(スレッド間で共有)、各スレッドで持っていたい変数はPrivate変数として指定する必要あり。ただし、ループの変数は自動的にPrivate変数として見なされている。

```
!$omp do
do i=1,1000
   tmp = some_function(i)
   a(i) = tmp;
nend do
!$omp end do
```

```
!$omp do private(tmp)
do i=1,1000
   tmp = some_function(i)
   a(i) = tmp;
nend do
!$omp end do
```

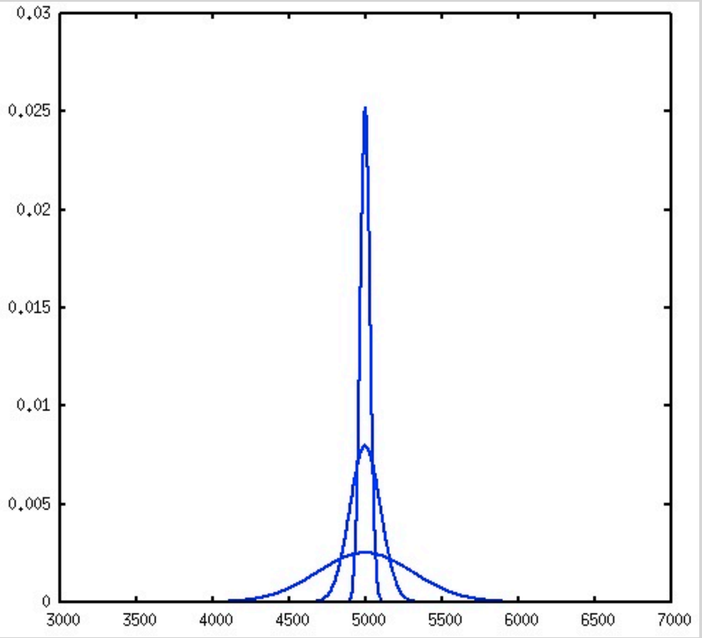
スレッドごとに
tmpを書き換えてしまうので
ダメ

```
#pragma omp for
for (i=0; i< 1000; i++) {
  tmp = some_function(i);
  a[i] = tmp;
 }
```

```
#pragma omp for private(tmp)
for (i=0; i< 1000; i++) {
  tmp = some_function(i);
  a[i] = tmp;
 }
```

(4) diffusion

1次元の拡散



スレッド生成するのは時間を消費するので
it のループの外側で並列化領域をとった。

変数itはPrivate変数とした

```fortran
program main
!$ include "omp_lib.h"
  integer,parameter :: num_size=10000
  integer,parameter :: no_time=10000

  real*8 :: st,en

  integer :: ip(0:num_size-1),im(0:num_size-1)

  real*8 :: p0(0:num_size-1),p(0:num_size-1)
  real*8 :: r

  integer :: it

  r=0.5d0

  do i=0,num_size-1
     ip(i)=mod(i+1          ,num_size)
     im(i)=mod(i-1+num_size,num_size)
  end do

  do i=0,num_size-1
     p0(i)=0.d0
  end do
  p0(num_size/2)=1.d0

  st = omp_get_wtime()

  !$omp parallel private(it)

  do it=1,no_time
     !$omp do
     do i=0,num_size-1
        p(i)=r*p0(ip(i))+(1.d0-r)*p0(im(i))
     end do
     !$omp end do
     !$omp do
     do i=0,num_size-1
        p0(i)=p(i)
     end do
     !$omp end do
  end do

  !$omp end parallel

  en = omp_get_wtime()
  write(*,*) "#Elapsed time in second is:", en-st

  do i=0,num_size-1
     write(*,*) i, p(i)
  end do

end program main
```

fortran

```c
# include <stdio.h>
# include <omp.h>

int main(){
  const int num_size=10000;
  const int no_time=10000;

  double p0[num_size],p[num_size];
  double r;

  int ip[num_size],im[num_size];

  int i,it;
  double st, en;

  r=0.5;

    for(i=0;i<num_size;i++){
      ip[i]=(i+1) % num_size;
      im[i]=(i-1+num_size) % num_size;
    };

    for(i=0;i<num_size;i++){
      p0[i]=0.0;
    };

  p0[  num_size/2]=1.0;

  st = omp_get_wtime();

#pragma omp parallel private(it)
  {

  for(it=0; it < no_time; it++){

#pragma omp for
    for(i=0;i<num_size;i++){
      p[i]=r*p0[ip[i]]+(1.0-r)*p0[im[i]];
    };

#pragma omp for
    for(i=0;i<num_size;i++){
      p0[i]=p[i];
    };
  };
  };

en = omp_get_wtime();
printf("#Elapsed time in second is: %f\n", en-st);

for(i=0;i<num_size;i++){
  printf("%d %f\n",i,p[i]);
 };

return 0;
}
```

c

diffusion_omp.optrpt

```c
# include <stdio.h>
# include <omp.h>

int main(){
  const int num_size=10000;
  const int no_time=20000;

  double p0[num_size],p[num_size];
  double r;

  int ip[num_size],im[num_size];

  int i,it;
  double st, en;

  r=0.5;

    for(i=0;i<num_size;i++){
      ip[i]=(i+1) % num_size;
      im[i]=(i-1+num_size) % num_size;
    };

    for(i=0;i<num_size;i++){
      p0[i]=0.0;
    };

  p0[  num_size/2]=0.5;
  p0[  num_size/2+1]=0.5;

  st = omp_get_wtime();

#pragma omp parallel private(it)
  {

  for(it=0; it < no_time; it++){

#pragma omp for
    for(i=0;i<num_size;i++){
      p[i]=r*p0[ip[i]]+(1.0-r)*p0[im[i]];
    };

#pragma omp for
    for(i=0;i<num_size;i++){
      p0[i]=p[i];
    };
  };
  };

en = omp_get_wtime();
printf("#Elapsed time in second is: %f\n", en-st);

for(i=0;i<num_size;i++){
  printf("%d %f\n",i,p[i]);
 };

return 0;
}
```

```
インテル(R) Advisor はベクトル化を支援するため、ソースコードで最適化
レポートメッセージを表示します。
詳細は、"https://software.intel.com/en-us/intel-advisor-xe" を参照してください。


    レポート: プロシージャー間の最適化 [ipo]

インライン展開オプション値:
  -inline-factor: 100
  -inline-min-size: 30
  -inline-max-size: 230
  -inline-max-total-size: 2000
  -inline-max-per-routine: 10000
  -inline-max-per-compile: 500000


最適化レポート開始: main()

    レポート: プロシージャー間の最適化 [ipo]

インライン展開レポート: (main()) [1] diffusion_omp.c(4,11)


    レポート: OpenMP* の最適化 [openmp]

OpenMP Construct at diffusion_omp.c(38,5)
リマーク #16204: SINGLE の OpenMP* マルチスレッド・コードが正常に生成されました。
OpenMP Construct at diffusion_omp.c(43,5)
リマーク #16204: SINGLE の OpenMP* マルチスレッド・コードが正常に生成されました。
OpenMP Construct at diffusion_omp.c(32,1)
リマーク #16201: OpenMP* 定義領域が並列化されました。

    レポート: ループの入れ子、ベクトル、自動並列化の最適化 [loop, vec, par]


ループの開始 diffusion_omp.c(18,5)
   リマーク #25084: 前処理ループの入れ子: ストアを外に移動しています。    [ diffusion_omp.c(18,24) ]
   リマーク #15335: ループ はベクトル化されませんでした: ベクトル化は可能ですが非効率です。オーバーライドするには vector always ディレクティブまたは -vec-threshold0 を使用してください。
ループの終了

ループの開始 diffusion_omp.c(23,5)
   リマーク #25084: 前処理ループの入れ子: ストアを外に移動しています。    [ diffusion_omp.c(23,24) ]
   リマーク #15300: ループがベクトル化されました。
ループの終了

ループの開始 diffusion_omp.c(52,1)
   リマーク #15344: ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。最初の依存関係を以下に示します。詳細については、レベル 5 のレポートを使用してください。
ループの終了

ループの開始 diffusion_omp.c(35,3)
   リマーク #15542: ループ はベクトル化されませんでした: 内部ループがすでにベクトル化されています。

   ループの開始 diffusion_omp.c(37,1)
      リマーク #15344: ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。最初の依存関係を以下に示します。詳細については、レベル 5 のレポートを使用してください。
      リマーク #15346: ベクトル依存関係: FLOW の依存関係が at (39:7) と at (39:7) の間に仮定されました。
      リマーク #25439: 剰余ありアンロール - 2
      リマーク #25456: ループで置換された配列参照スカラーの数: 1
   ループの終了

   ループの開始 diffusion_omp.c(37,1)
   <剰余>
      リマーク #25456: ループで置換された配列参照スカラーの数: 1
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <ベクトル化のピールループ, マルチバージョン v1>
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <マルチバージョン v1>
      リマーク #25228: データの依存関係のループをマルチバージョンにしました。
      リマーク #15300: ループがベクトル化されました。
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <代替アライメントでベクトル化されたループ, マルチバージョン v1>
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <ベクトル化の剰余ループ, マルチバージョン v1>
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <マルチバージョン v2>
      リマーク #15304: ループ はベクトル化されませんでした: マルチバージョンのベクトル化できないループ・インスタンスです。
      リマーク #25439: 剰余ありアンロール - 2
      リマーク #25456: ループで置換された配列参照スカラーの数: 1
   ループの終了

   ループの開始 diffusion_omp.c(42,1)
   <剰余, マルチバージョン v2>
      リマーク #25456: ループで置換された配列参照スカラーの数: 1
   ループの終了
ループの終了

    レポート: コード生成の最適化 [cg]

diffusion_omp.c(4,11):リマーク #34051: レジスター割り当て: [main] diffusion_omp.c:4

   ハードウェア・レジスター
       予約済み           :    2[ rsp rip]
       利用可能           :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
       呼び出し先セーブ   :    6[ rbx rbp r12-r15]
       割り当て済み       :   20[ rax rdx rcx rbx rsi rdi r8-r15 zmm0-zmm5]

   ルーチンの一時変数
       合計               :     305
           グローバル     :      92
           ローカル       :     213
       再生成             :     108
       退避               :       7

   ルーチンスタック
       変数               :      68 バイト*
           読み取り       :       8 [0.00e+00 ~ 0.0%]
           書き込み       :      17 [3.26e+00 ~ 3.3%]
       退避               :      96 バイト*
           読み取り       :      23 [3.13e+00 ~ 3.1%]
           書き込み       :      17 [3.46e-01 ~ 0.3%]

   注

       *オーバーラップしない変数と退避はスタック空間を共有できるため、
        合計スタックサイズはこれよりも小さくなる可能性があります。


===========================================================================
```

(5) reaction_diffusion

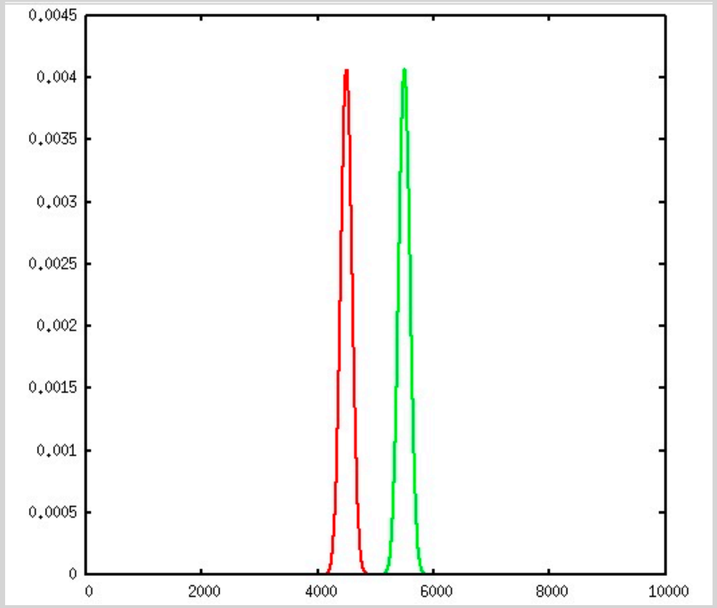
ループ内の計算量が増えた場合

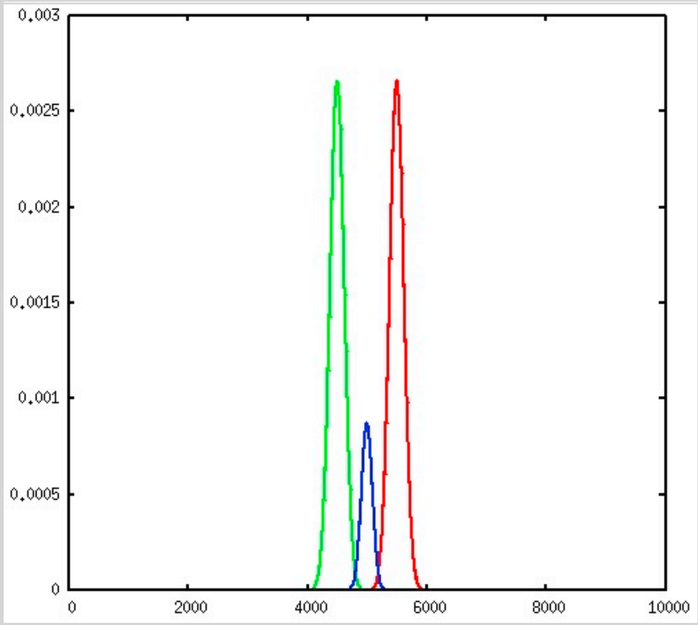
```
!$omp do
do i=0,num_size-1
   p(i)=r*p0(ip(i))+(1.d0-r)*p0(im(i))
end do
!$omp end do
```

```
!$omp do
do i=0,num_size-1
   pA(i)=rA*pA0(ip(i))+(1.d0-rA)*pA0(im(i))-alpha*PA0(i)*PB0(i)
   pB(i)=rB*pB0(ip(i))+(1.d0-rB)*pB0(im(i))-alpha*PA0(i)*PB0(i)
   pC(i)=rC*pC0(ip(i))+(1.d0-rC)*pC0(im(i))+alpha*PA0(i)*PB0(i)
end do
!$omp end do
```

反応拡散

$A + B \rightarrow C$





まとめとヒント

- この講習では、ループの並列化(ループ構文)を例として取り上げた。変数の定義参照関係に注意することなど、ベクトル化の場合と似ている。これ以外に、section構文(ループではないが、独立に実行できる仕事を複数スレッドに割り振る)もある。

- OpenMPの良さ：並列化前のプログラム(動作確認済み)に並列化を指示する行を付け加えるだけでできる。デバッグしやすい。(コンパイルのとき並列化のオプションを外せば並列化前と同じ) 定義参照関係、shared/privateの違いなどに注意しつつ、少ない修正(努力)で並列化できるところをやる。自動並列化（-parallel) だけでもやる価値あり。

- 1つのジョブで複数プログラムを実行する方法　ノード内のコアをなるべく使うために有用

- さらに効率を上げるためには、OpenMPでさらに工夫するよりも、ベクトル化(SQUID ベクトルノード), MPI(SQUID)と組み合わせることにし、そちらの方に注力した方が良い。(プログラム開発の労力と得られる効率のバランスの観点から)

例えばパラメータについてMPI並列化（trivialな並列化)するのは比較的楽で得られる効率は大きい。