

2020/09/30 13:30-14:30 オンライン

# OpenMP入門

[http://www.hpccmc.osaka-u.ac.jp/lec\\_ws/20200930/](http://www.hpccmc.osaka-u.ac.jp/lec_ws/20200930/)

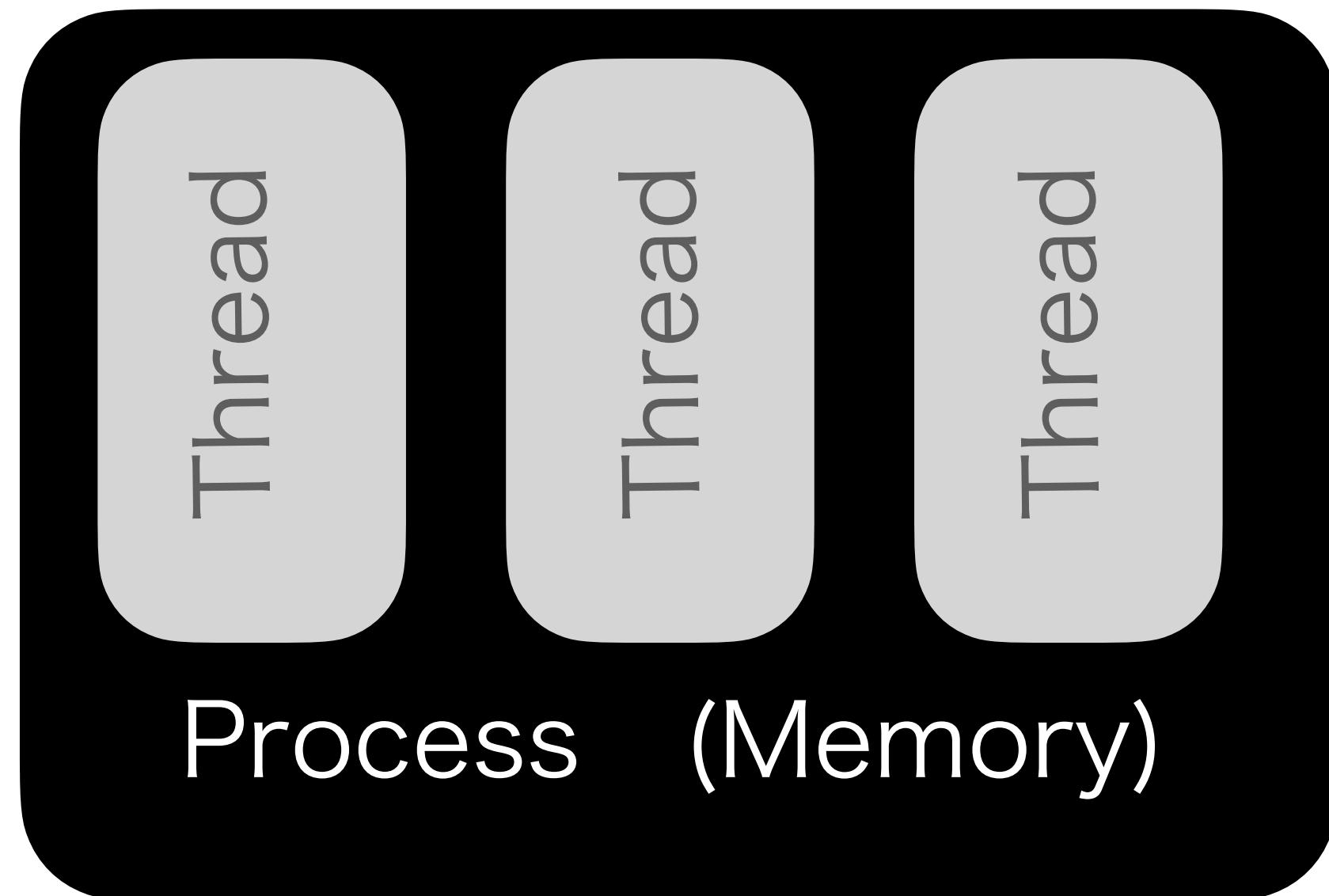
大阪大学サイバーメディアセンター  
吉野 元

# この講習会の内容

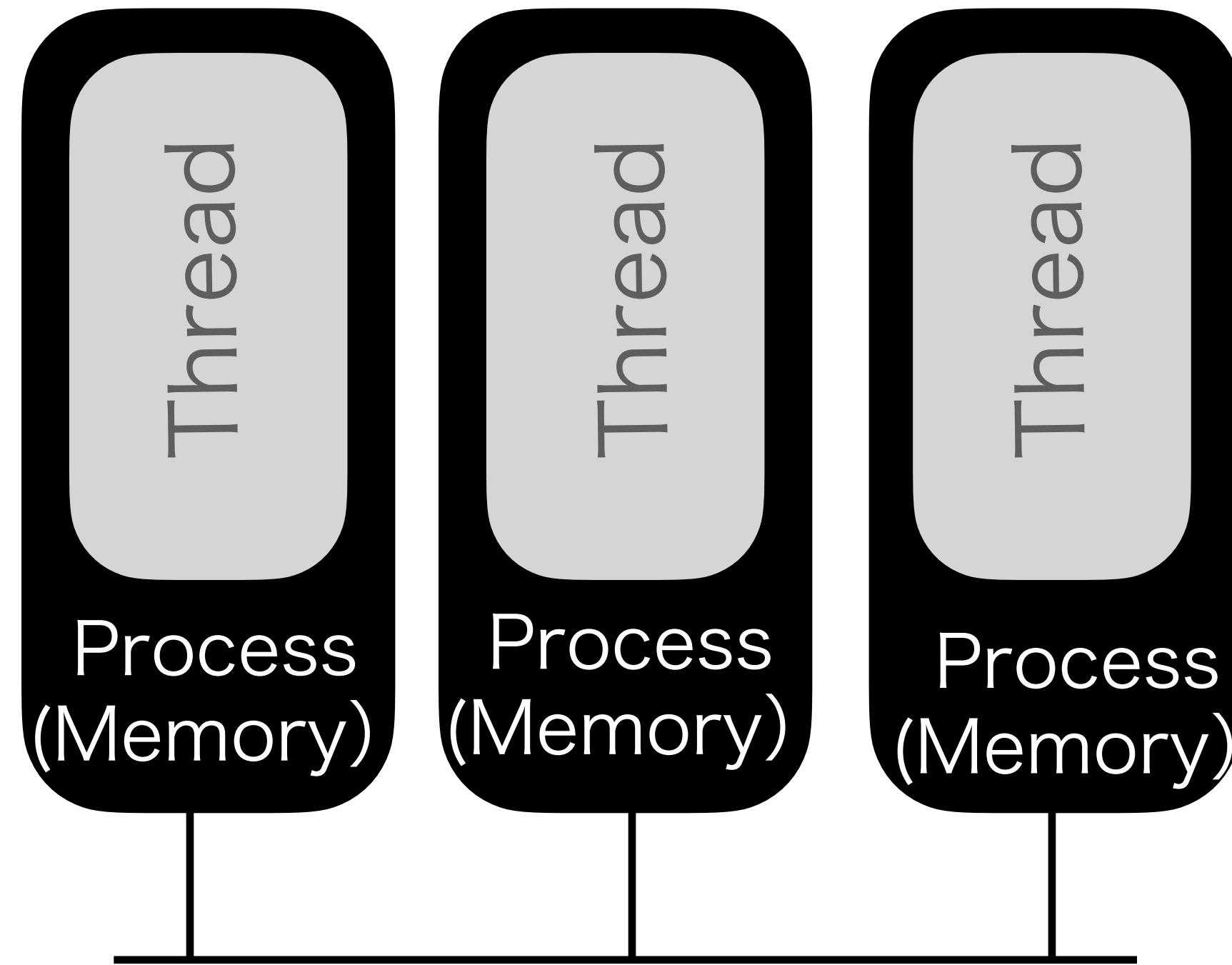
- ・ 導入
- ・ 演習：サンプルプログラムのコンパイル、実行、実行時間の計測
- ・ まとめとヒント

# OpenMPとは？

OpenMP/自動並列化



MPI (Message Passing Interface)



プロセス間通信

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)を作り並列に計算する。メモリーは共有。

# OpenMPが使えるところ

- SX-ACE (1node/4core 60GB)- ベクトル化、MPIとの組み合わせも可  
[http://www.hpc.cmc.osaka-u.ac.jp/system/manual/sx-ace/openmp\\_sxace/](http://www.hpc.cmc.osaka-u.ac.jp/system/manual/sx-ace/openmp_sxace/)
- OCTOPUS (1node/24core 190GB) - MPIとの組み合わせも可  
<http://www.hpc.cmc.osaka-u.ac.jp/system/manual/octopus-use/cpu-openmp/>
- 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。。

- 自動並列化、OPENMPは簡単に試せる  
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）

## SX-ACEでの説明

また、ノード時間は使用したノード数を元に計算します。ノード内の使用率によって減額はいたしません。  
1ノード内で使用するコアを限定しても、ノード時間は1ノード中の全コアを使用したものとして扱います。

## OCTOPUSでの説明

1ノード時間=1ノードを使った計算を1時間行う資源量となります。

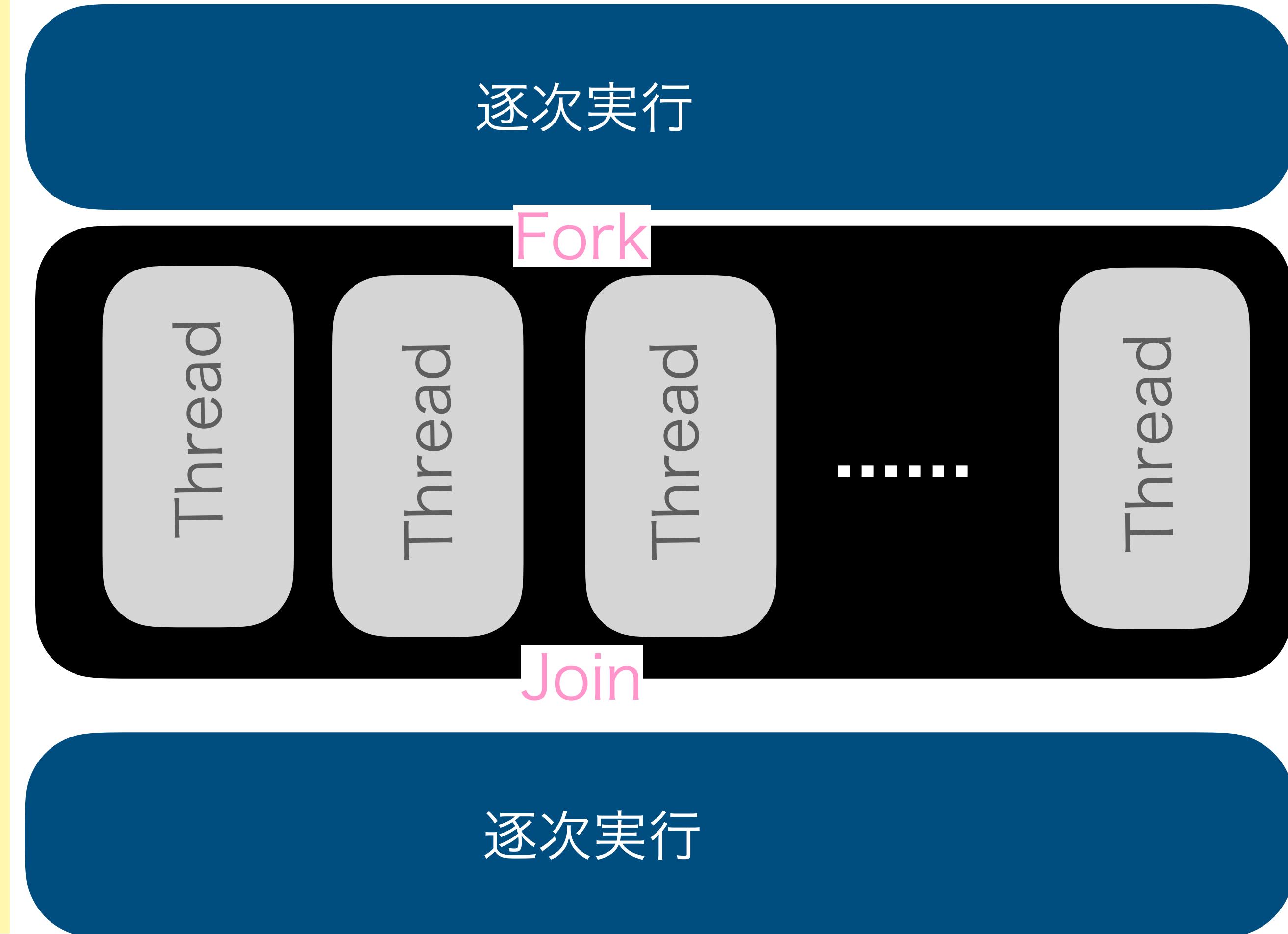


# プログラムはどう書くか？

FORTRAN  
の場合

`!$ omp parallel`

`!$ omp end parallel`



Cの場合

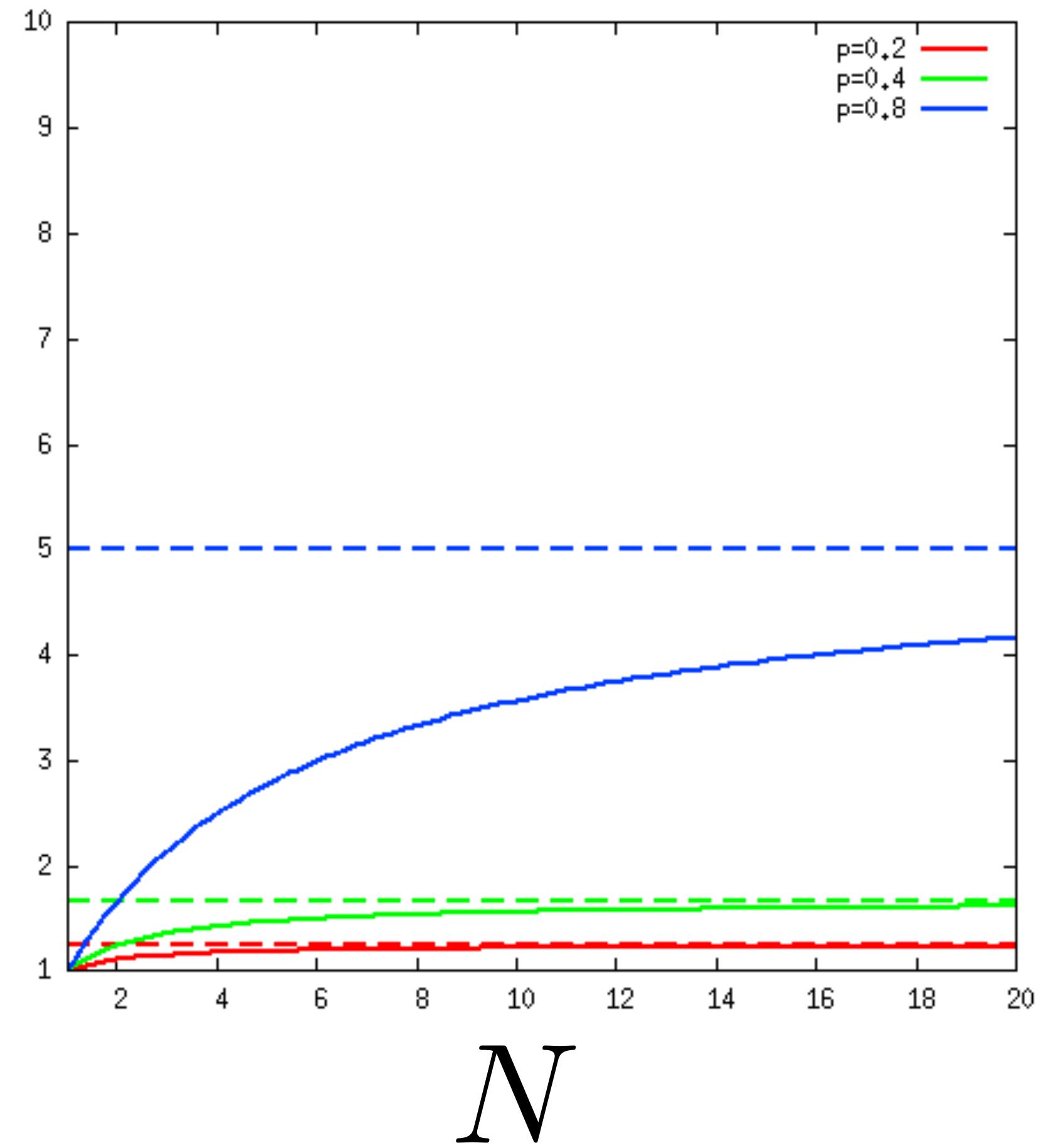
`#pragma parallel {`

`}`

# Amdahlの法則

並列化効率 =

$$\frac{1}{1 - p + \frac{p}{N}}$$



並列化できる部分  
の計算量

$p$

$N = 5$

並列化できない部分の計算量

$1 - p$

$N = 10$



# 演習

- OCTOPUS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 /octfs/apl/kosyu/OpenMP/C.tar

Fortran /octfs/apl/kosyu/OpenMP/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/reaction\_diffusion

# (1) hello world

```
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
```

fortran

```
# 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;
}
```

C

## コンパイル

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

以下は同じ

```
v6a022@octopus03[122]% cat helloworld_omp.sh  
#!/bin/bash  
#PBS -q LECTURE (講習会用) (DBGがデバッグに便利)  
#PBS -y 332 (講習会用)  
#PBS -l cpunum_job=24  
#PBS -v OMP_NUM_THREADS=10  
#PBS -l elapstim_req=00:10:00  
cd $PBS_O_WORKDIR  
. ./a_helloworld_omp.out > helloworld_omp.log  
v6a022@octopus03[117)% qsub helloworld_omp.sh  
Request 872555.oct submitted to queue: LECTURE.
```

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。

```
v6a022@octopus03[121)% qstat  
RequestID      ReqName   UserName Queue      Pri  STT S    Memory          CPU   Elapse R H M Jobs  
-----  -----  -----  -----  ---  -  -----  -----  -----  -----  -----  -----  
872555.oct      helloworld v6a022  LECTURE-C      0  QUE -    0.00B    0.00      0 Y Y Y  1
```

## 結果

```
v6a022@octopus03[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

```
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

```
# 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

```
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
```

```
# 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;
}
```

# 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
```

スレッドごとに  
tmpを書き換えてしまうので  
ダメ

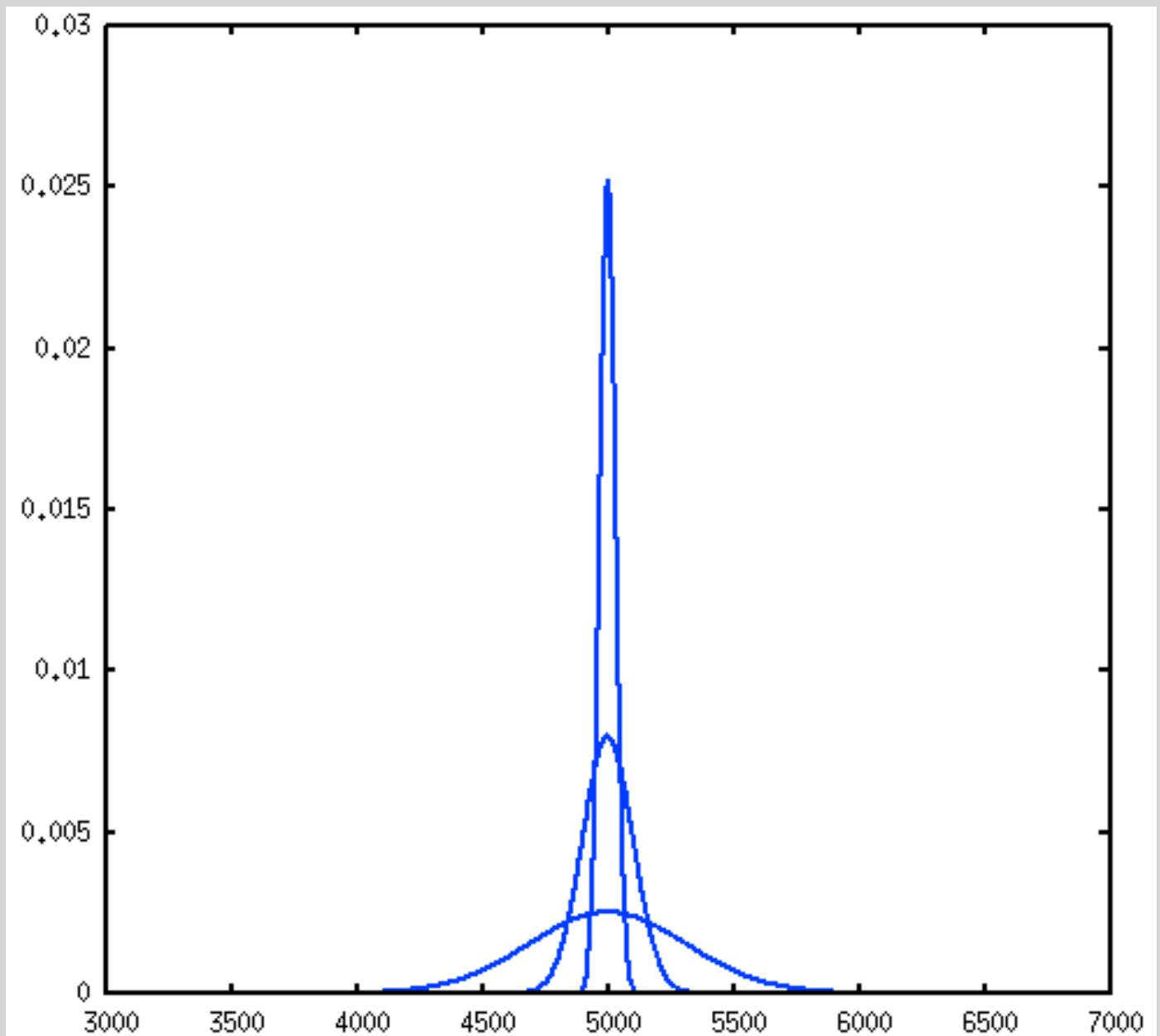
```
!$omp do private(tmp)
do i=1,1000
  tmp = some_function(i)
  a(i) = tmp;
nend do
 !$omp end do
```

```
#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

```
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
```

```
#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]);
    };

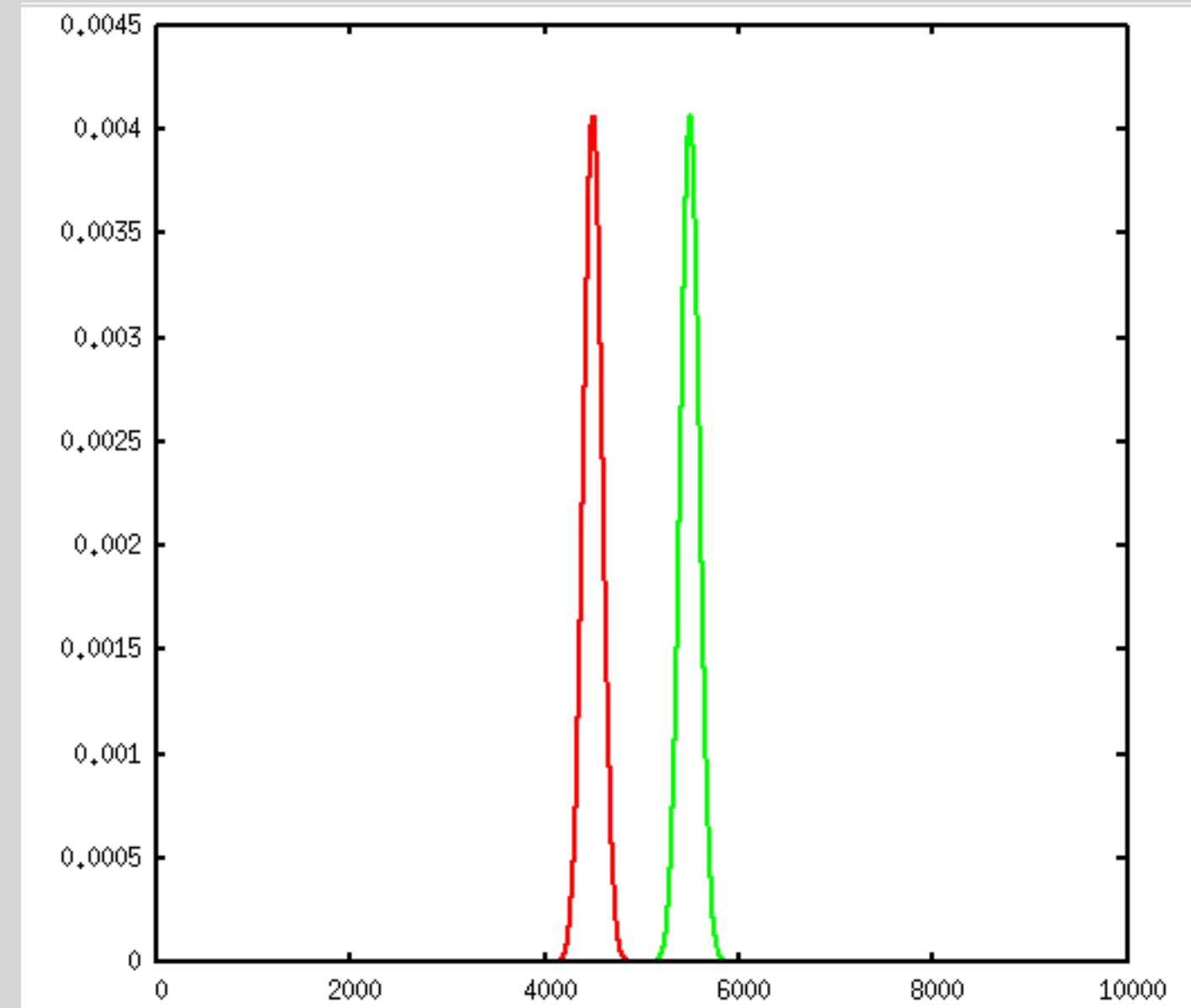
    return 0;
}
```

# (5) reaction\_diffusion

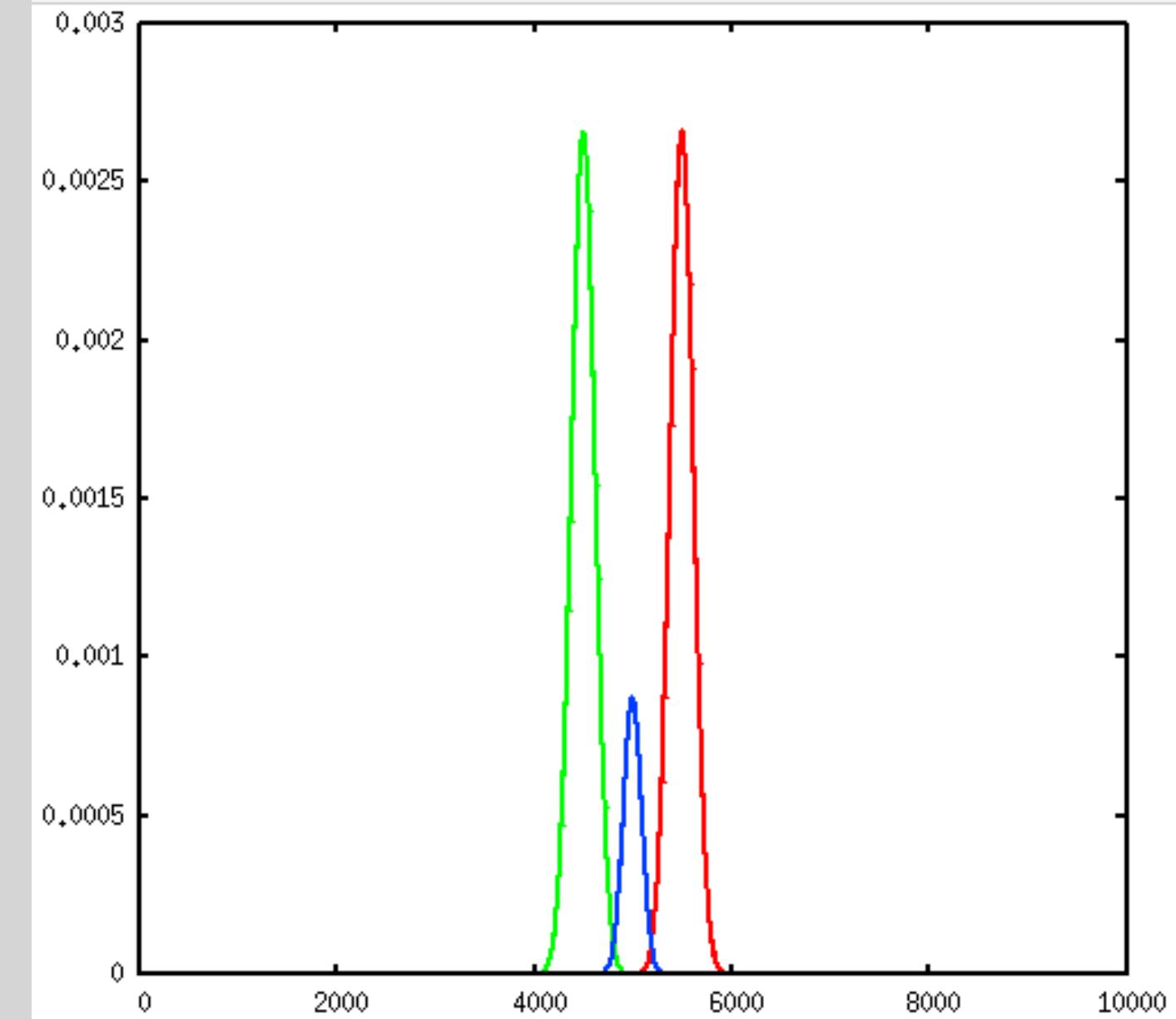
ループ内の計算量が増えた場合

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

反応拡散



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



# 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ができる。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。
- さらに効率を上げるために、OpenMPでさらに工夫するよりも、ベクトル化(SX-ACE), MPI(SX-ACE,OCTOPUS)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かもしれない。(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。(HPCっぽくないが、構わない。)