初心者のためのクラスタ　III

クラスタにおける
並列アプリケーション

有限会社イワタシステムサポート　　岩田　進吉

2026年2月3日(火)

初心者のためのクラスタIII

並列アプリケーション

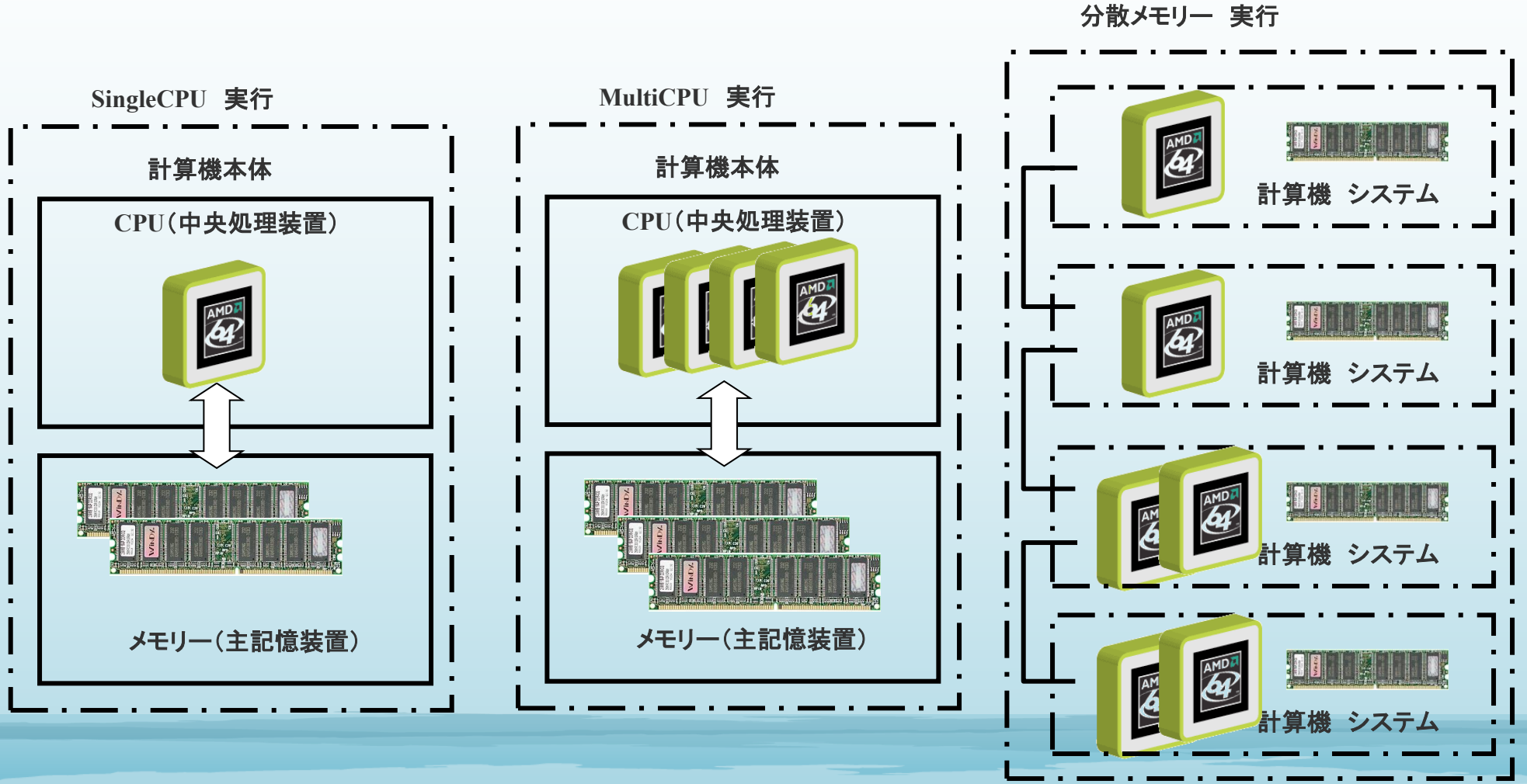
- メモリ共有アプリケーション
- 並列分散アプリケーション
- 並列アプリケーションに適したプログラミング言語
- 並列アプリケーションの作成方法
- 簡単な並列アプリケーションの紹介

HPC分野でのアプリケーションの特徴

クラスタシステムで使われるアプリケーションには主に以下の3種類があり、この中でMPIやSMP対応のアプリケーションがクラスタシステムで多く使われる。

- シングルCPU（コア）プログラム
- SMP（Symmetric MultiProcessing）プログラム
- MPI（Message Passing Interface）プログラム

アプリケーション実行時のCPUとメモリーの関係



並列アプリケーション

SMP（メモリ共有）アプリケーション

メモリ共有アプリケーションのイメージ

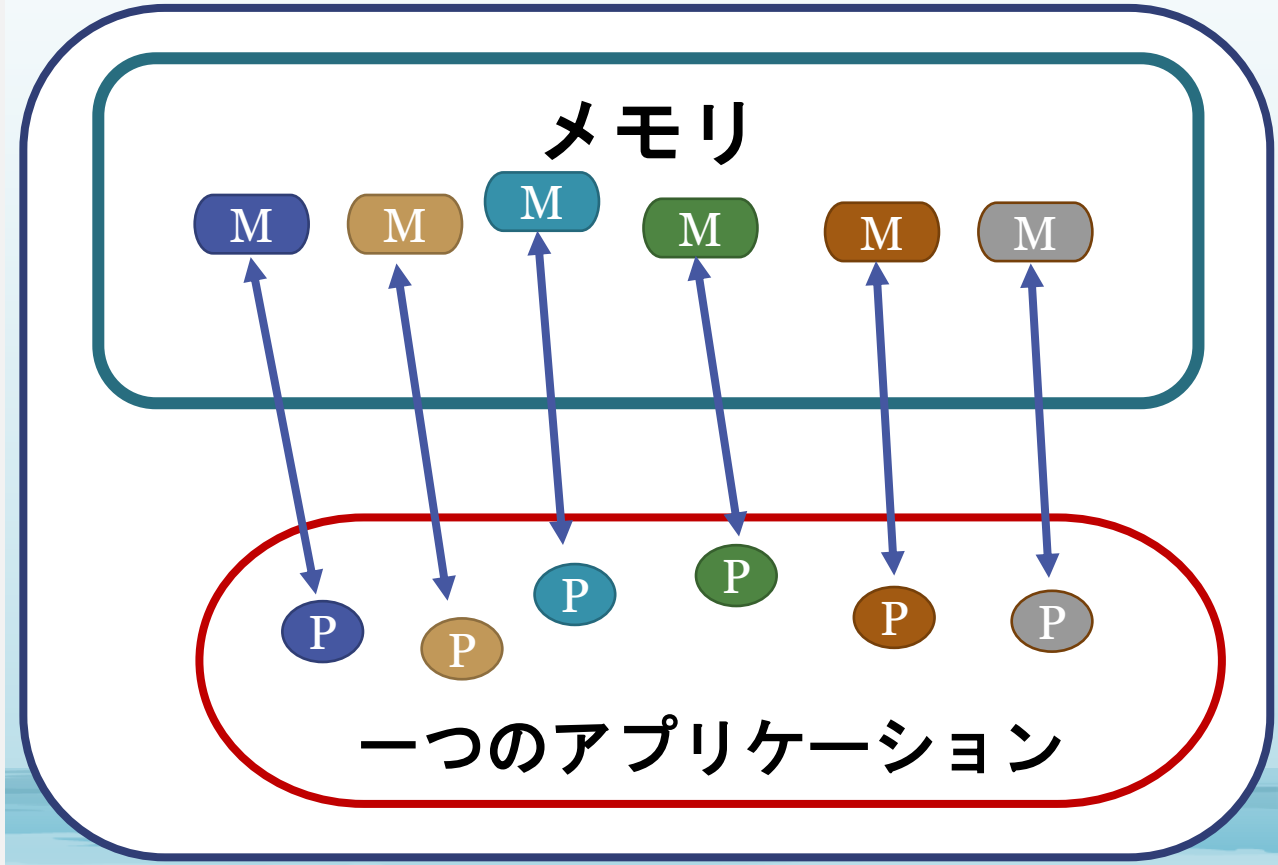
一つのアプリケーションが複数のコアを使って並列計算を行う

SMP【Symmetric Multiprocessing】対称型マルチプロセッシング

一つのコンピュータ上で一つのジョブを複数のコアを使って同じメモリ領域をアクセスしながら計算する手法。

以下の特徴を持つ

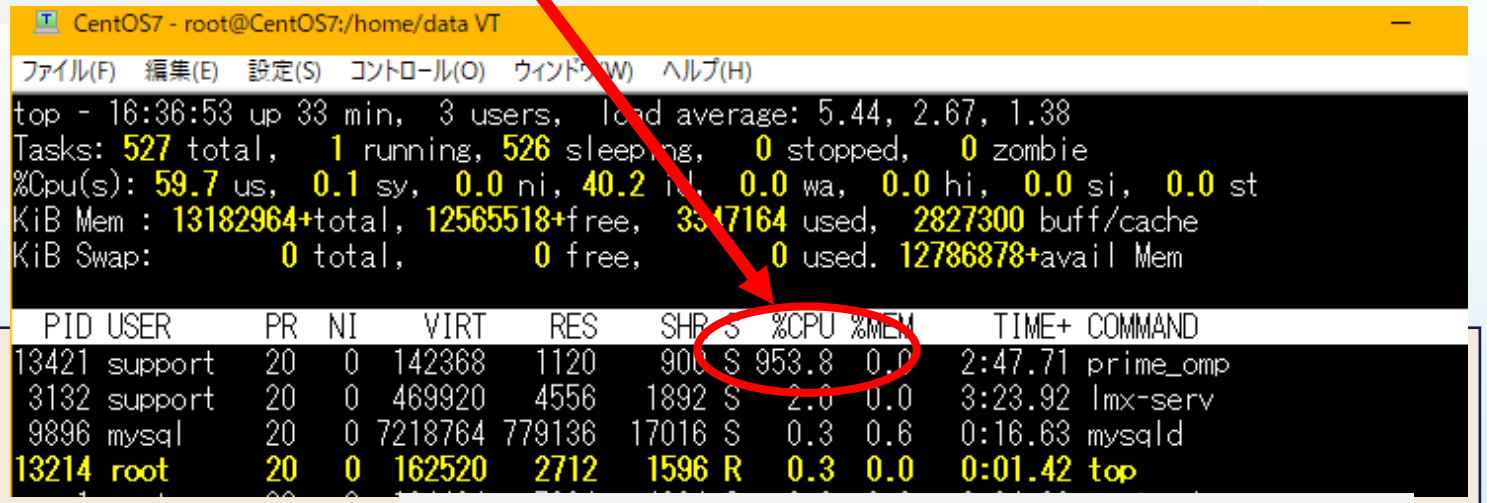
- ソフトウェア開発が容易
- メモリを共有するためにメモリのアクセス競合が発生する
- 並列度(利用するコア数）は8コア程度

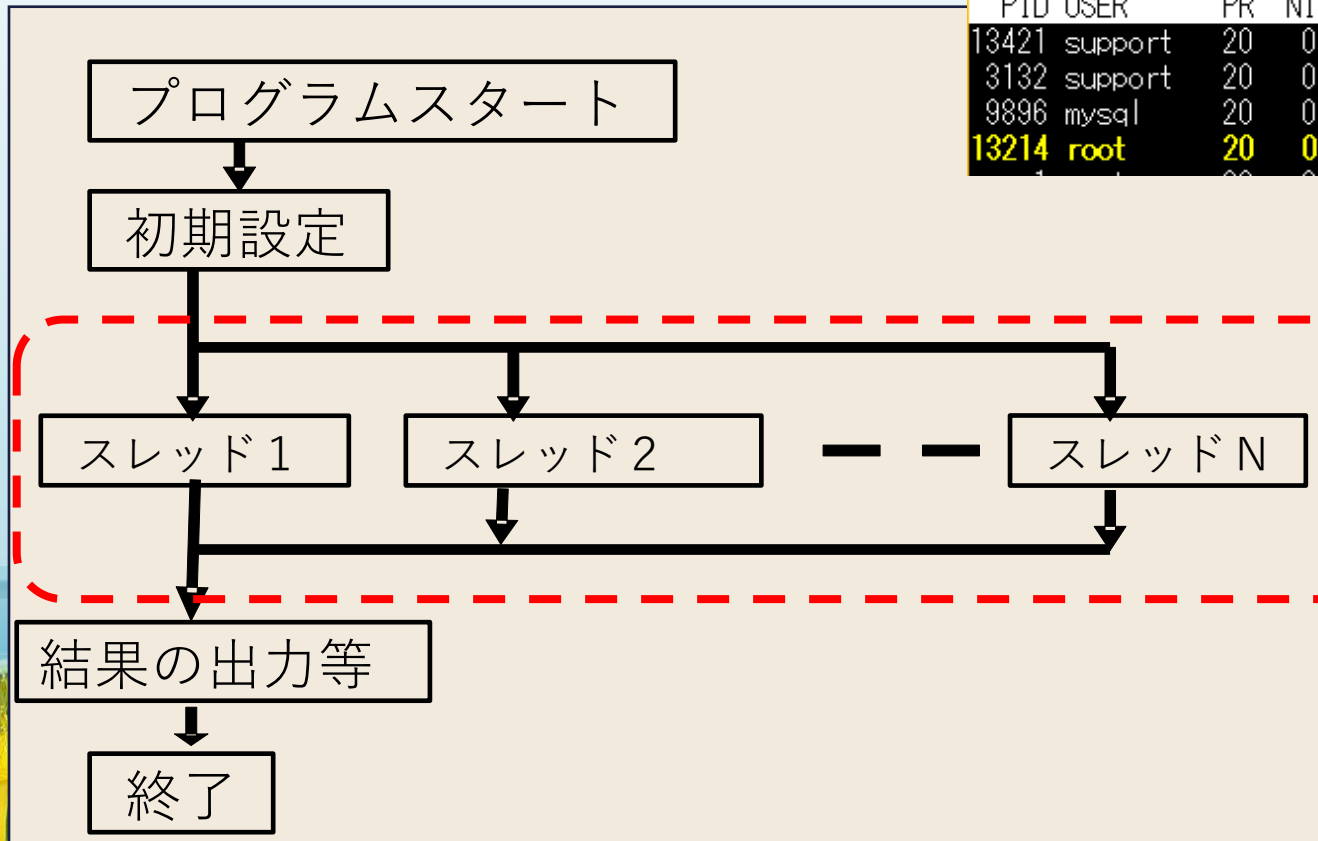


M:メモリ領域
P:　　各コア

参考：OpenMPによるSIMDアプリケーションの実行例

Topコマンドで見ると、以下の例の様にCPUの利用率が953%になって






```
print *,"単一処理部分"

!$omp parallel
      print *,"並列処理部分"
!$omp end parallel

print *,"単一処理部分"
```

並列アプリケーション

MPI（分散並列）アプリケーション

分散並列アプリケーションのイメージ

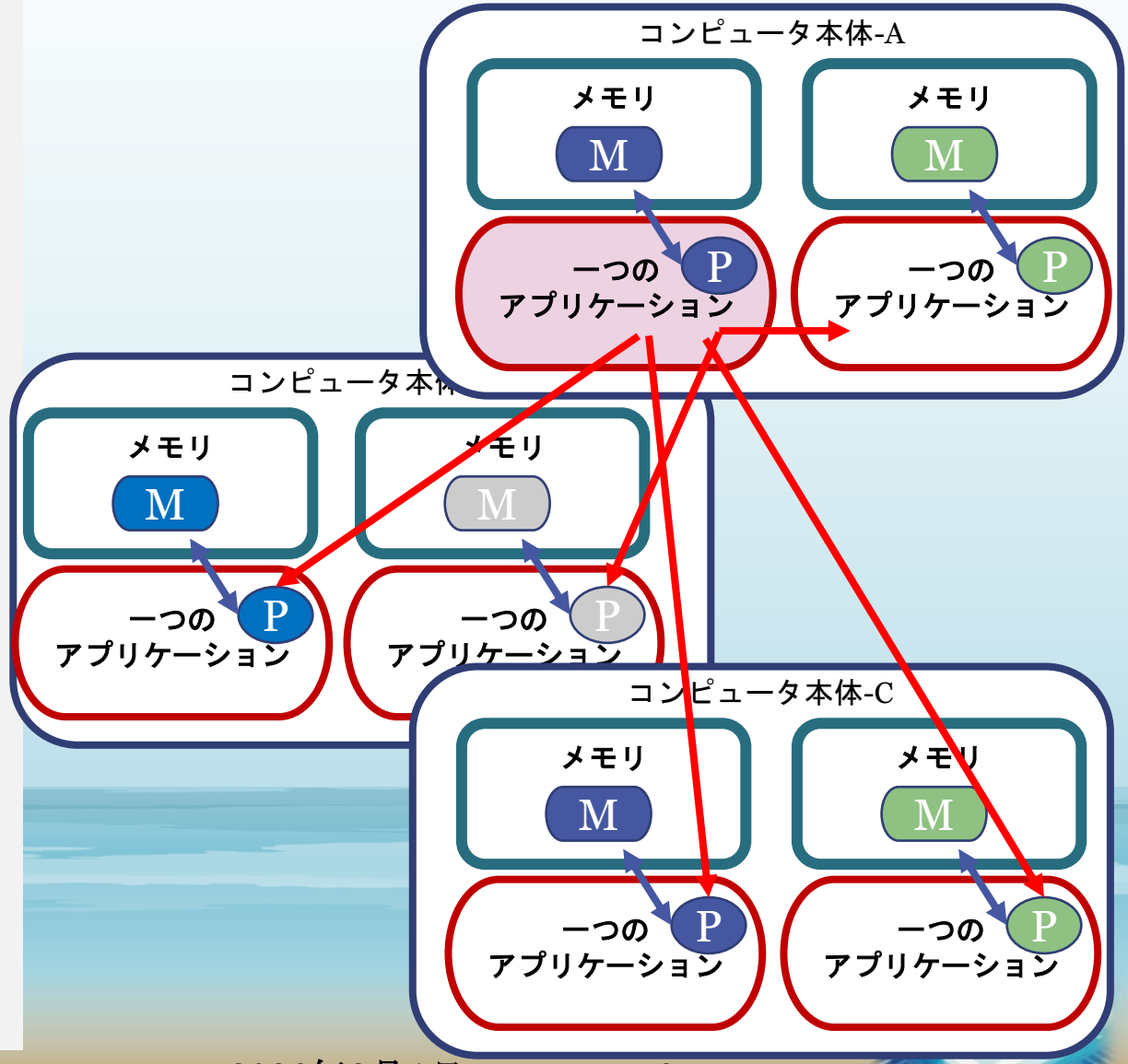
一つのコンピュータで起動を指示すると指定したノードにアプリケーションがコピーされ一緒に起動される。

MPI【Message Passing Interface】分散メモリ並列アプリケーション

一つもしくは複数ののコンピュータ上で一つのアプリケーションを複数のジョブとして起動し、複数のコアを使って別々のメモリ領域をアクセスしながら計算する手法。

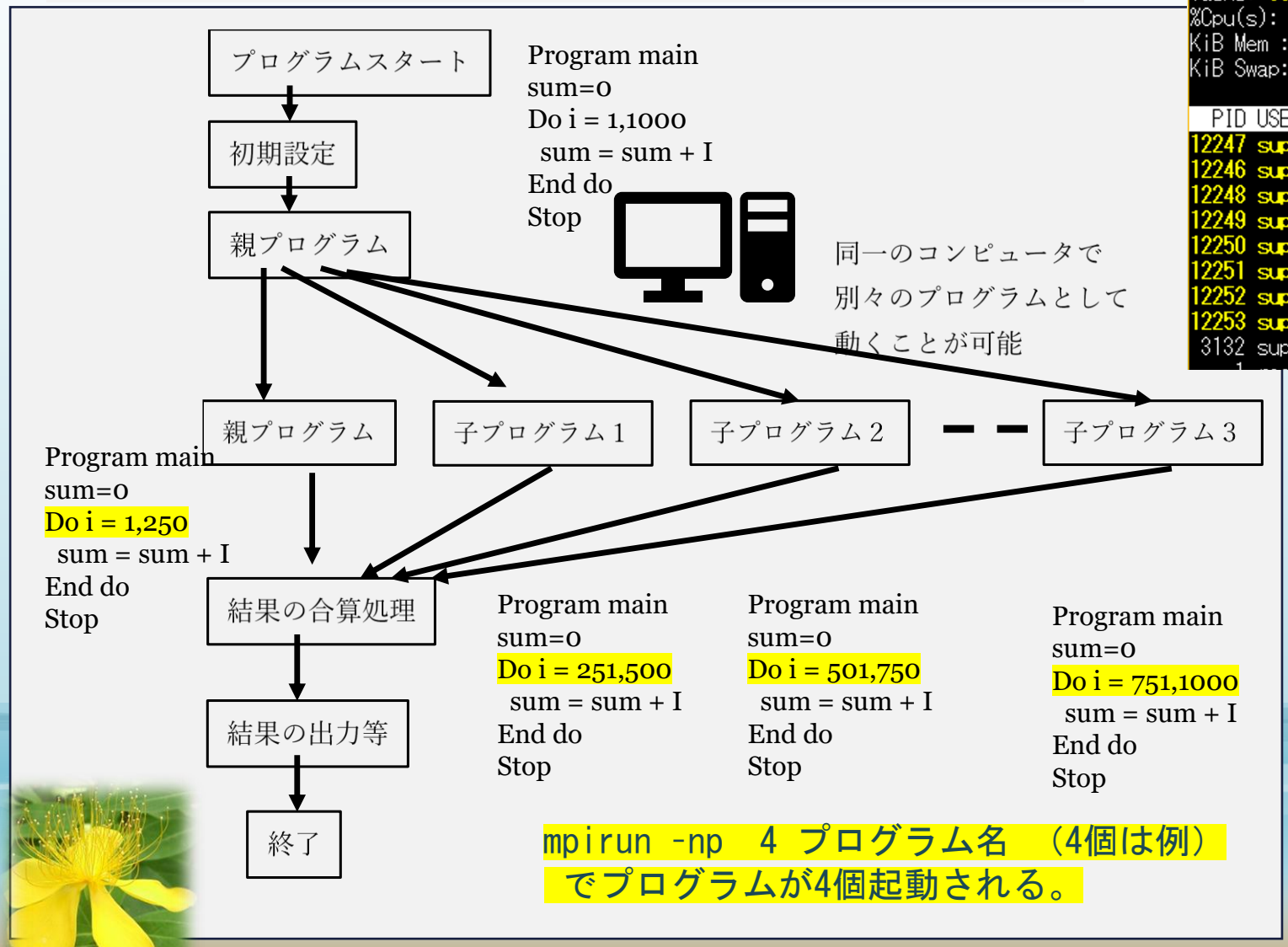
以下の特徴を持つ

- ソフトウェア開発が少々難しい
- メモリのアクセス競合しないので並列計算の性能が直線的に向上する
- 並列度は分野によるが流体計算等では数千コア並列でも性能が伸びる

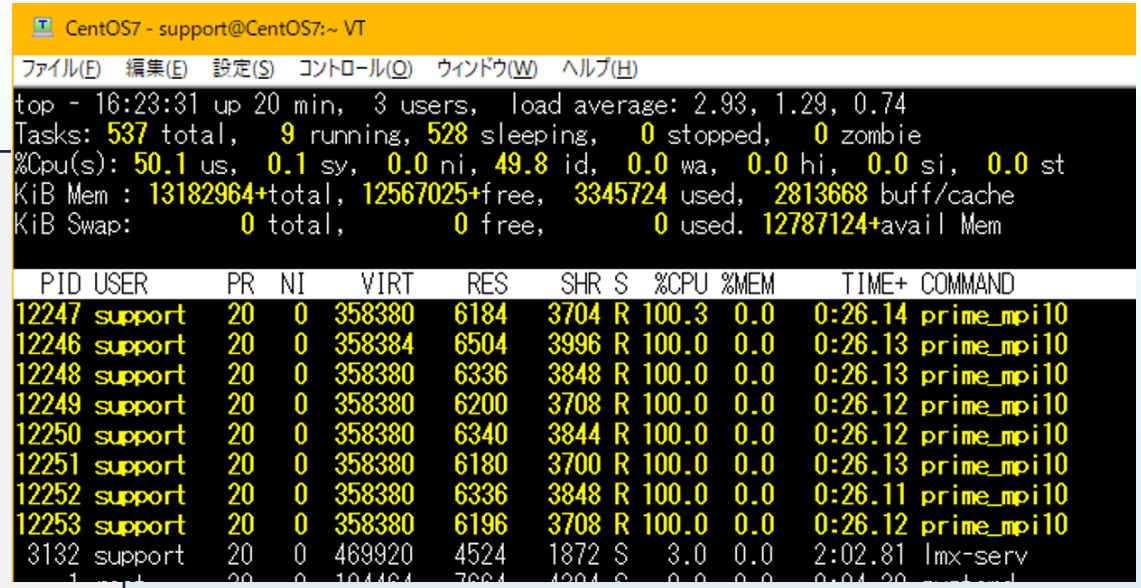


並列計算プログラム

並列計算におけるMPIの親プログラムから同一のコンピュータにジョブが展開されるイメージです。



mpirun -np 4 プログラム名 （4個は例）
でプログラムが4個起動される。



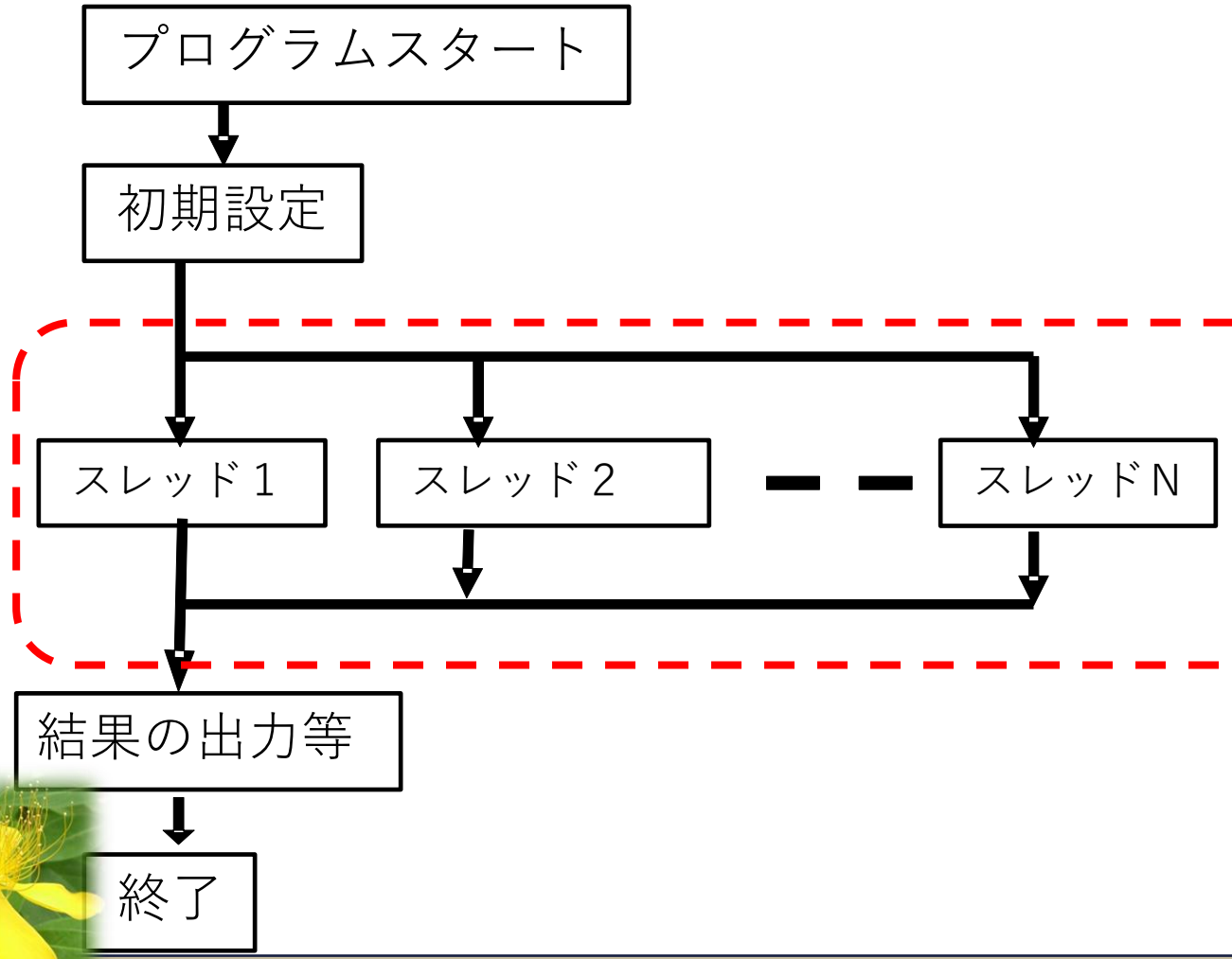
実行時の挙動

Topコマンドで見ると、以下の例の様にCPUの利用率が100%前後で8つ実行されている例

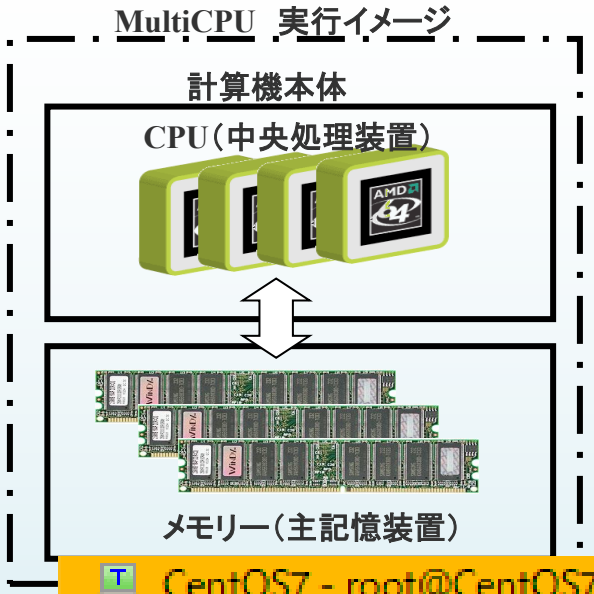
SMPアプリケーション

OpenMPは一つのプログラムがメモリを共有して複数の処理を同時に実行する仕組みで動く。コンピュータから見た時に動いているプログラムは一つであり、CPUの利用率が800%の様に100%以上の利用率になる。指定した数の各スレッドが、各CPUのコアを使って動いている。

各スレッドはメモリを同時にアクセスしているので並列性能を上げるのが難しい作りです。




```
print *,"単一処理部分"

!$omp parallel
      print *,"並列処理部分"
!$omp end parallel

print *,"単一処理部分"
```



実行時の挙動

Topコマンドで見ると、以下の例の様にCPUの利用率が953%になって

CentOS7 - root@CentOS7:/home/data VT

ファイル(F)　編集(E)　設定(S)　コントロール(O)　ウィンドウ(W)　ヘルプ(H)

```
top - 16:36:53 up 33 min,  3 users,  load average: 5.44, 2.67, 1.38
Tasks: 527 total,   1 running, 526 sleeping,   0 stopped,   0 zombie
%Cpu(s): 59.7 us,  0.1 sy,  0.0 ni, 40.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 13182964+total, 12565518+free,  3347164 used,  2827300 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 12786878+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13421 support   20   0  142368   1120    900 S 953.8  0.0   2:47.71 prime_omp
 3132 support   20   0  469920   4556   1892 S   2.0  0.0   3:23.92 lmx-serv
 9896 mysql     20   0 7218764 779136  17016 S   0.3  0.6   0:16.63 mysqld
13214 root      20   0  162520   2712   1596 R   0.3  0.0   0:01.42 top
```

4並列で実行した場合

```
$ export set OMP_NUM_THREADS=4
$ ./test

PRIME_NUMBER_OPENMP
  FORTRAN90/OpenMP version

  Number of processors available =  4
  Number of threads =               4
 単一処理部分
 並列処理部分
 並列処理部分
 並列処理部分
 並列処理部分
 単一処理部分
```

2並列で実行した場合

```
$ export set OMP_NUM_THREADS=2
$ ./test

PRIME_NUMBER_OPENMP
  FORTRAN90/OpenMP version

  Number of processors available =  4
  Number of threads =               2
 単一処理部分
 並列処理部分
 並列処理部分
 単一処理部分
```

並列処理部

OpenMPのポイントは

OpenMPを動かすために最低限必要なライブラリ

OpenMPを利用する為には、まずは

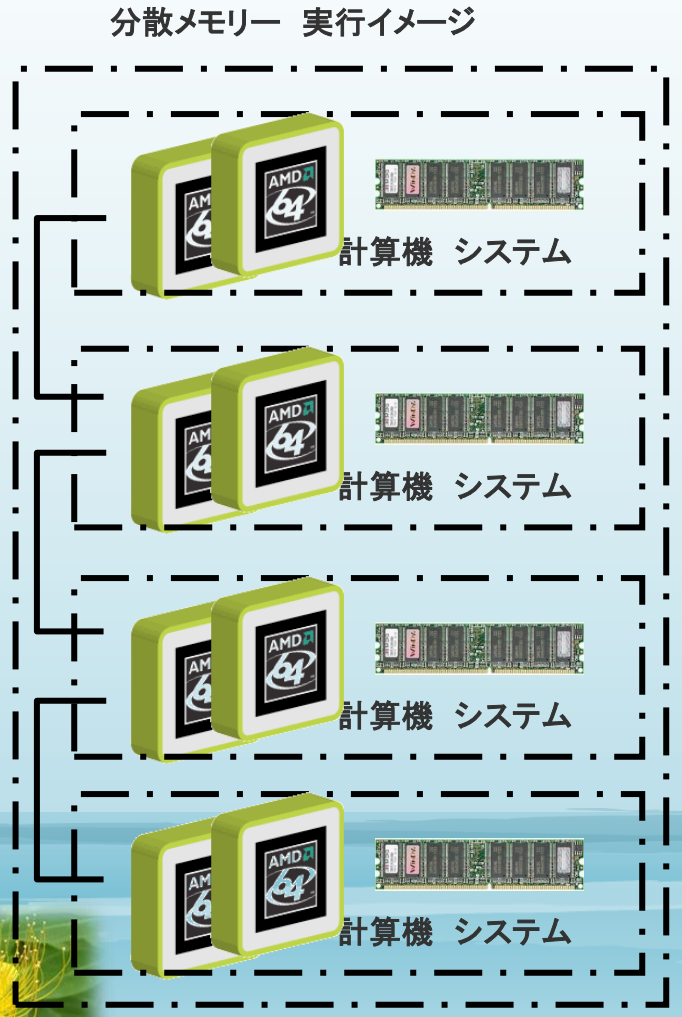
use omp_lib

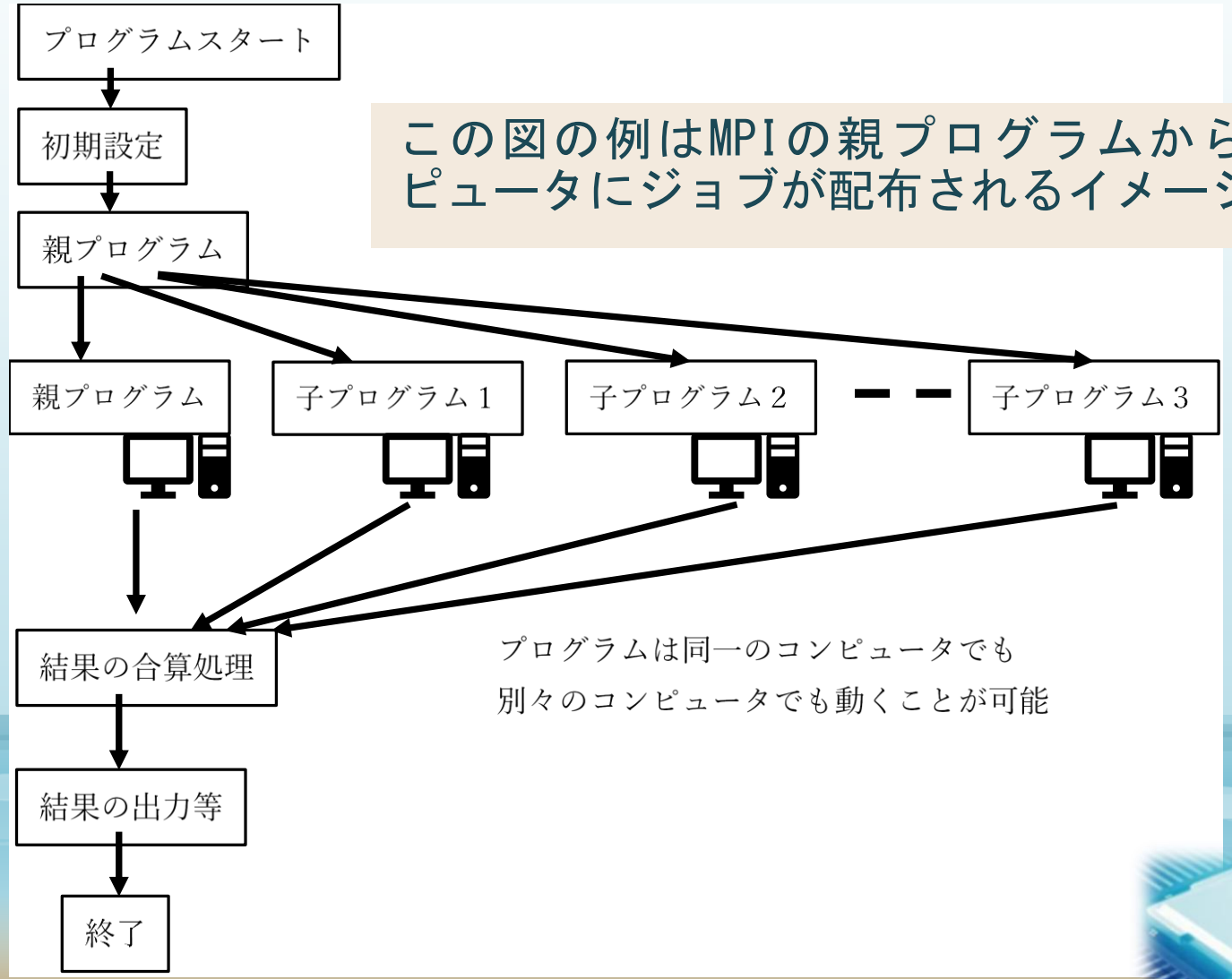
を呼出して必要なOpenMPのサブルーティン群を使えるようにします。これは　Fortran90以降で、それ以前のVersionでは　include　‘omp_lib.h’ で設定を行っておりました。

```
!$omp parallel                                  <———  並列区間を指示する
                                                <———  以下のdo文を並列実行するように指示
                                                      並列内でtotalを個別に掲載して最後に加算
!$omp do reduction ( + : total )

  do i = 1, n
    total = total + i
  end do

!omp end do                                     <———  並列のdo区間の最終を指示
!$omp end parallel                              <———  並列区間の最終を指示
```

MPIアプリケーション

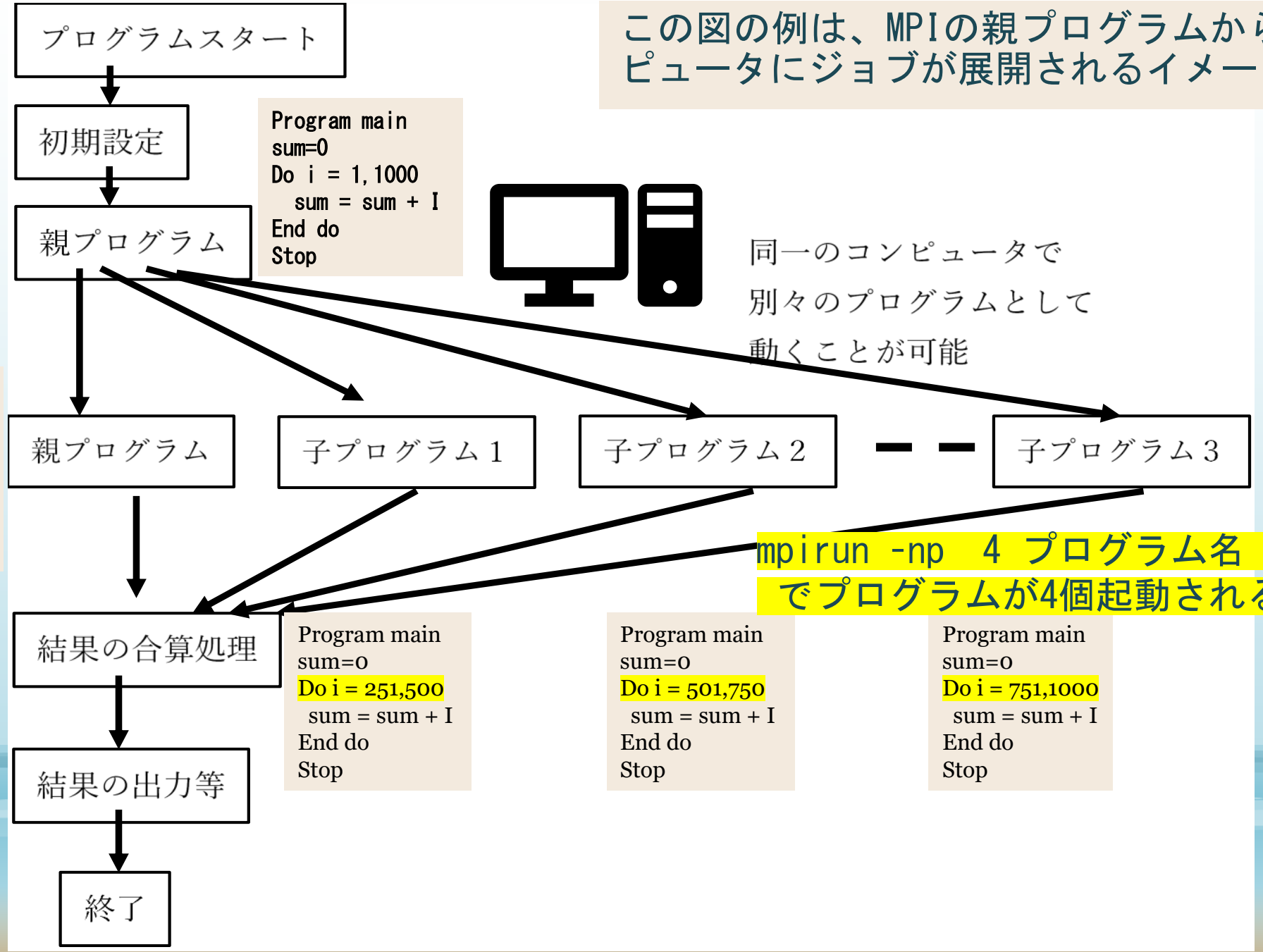
MPIライブラリは各プログラムが独立に動作します。よってプログラム単位ではCPUの利用率は、最大で100%になります。（コンピュータ内はOpenMP、コンピュータ間はMPIという構成もできますが、あまり聞きません。構築が少々、複雑になります）





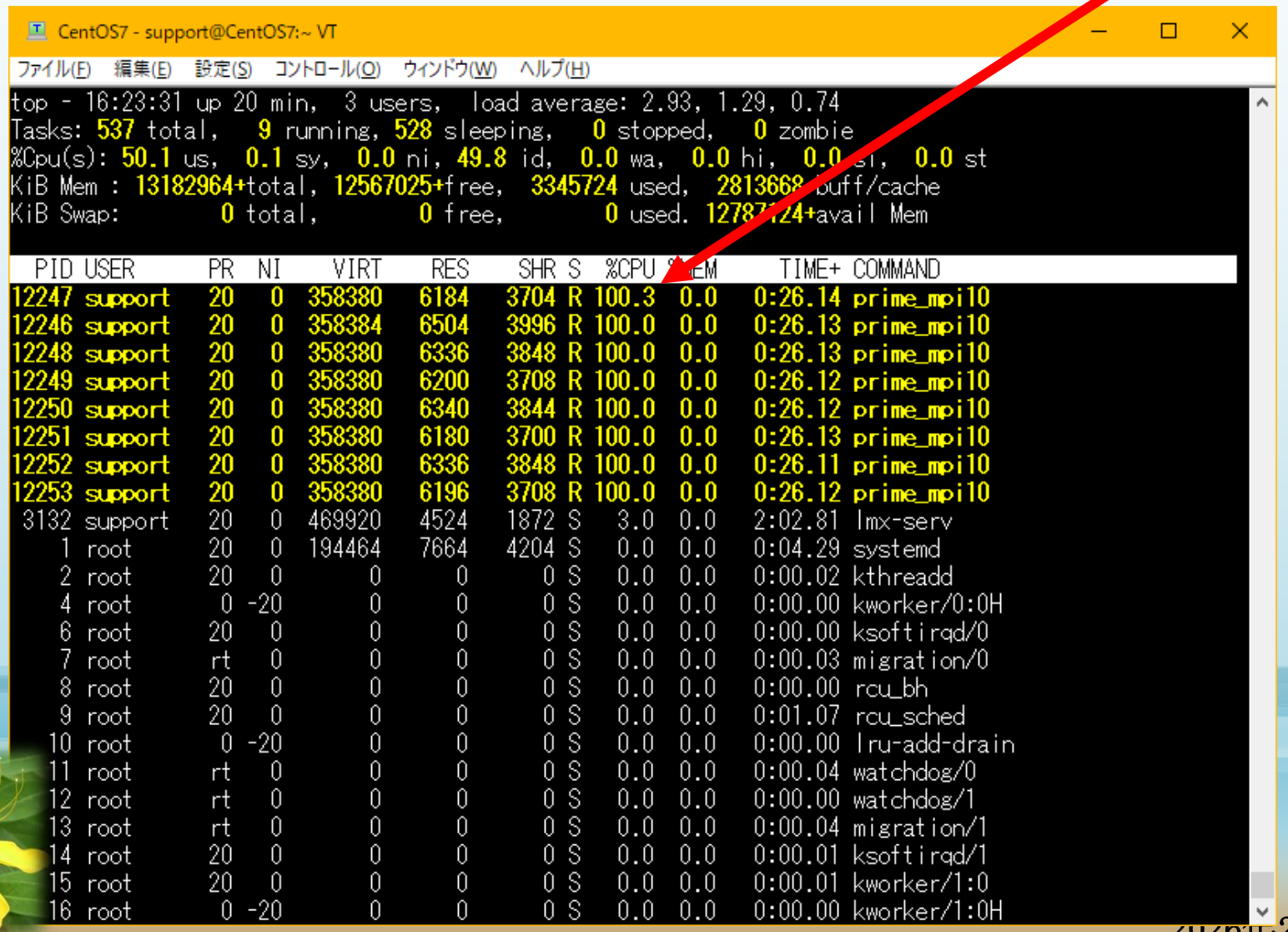
この図の例はMPIの親プログラムから各コンピュータにジョブが配布されるイメージです。

この図の例は、MPIの親プログラムから同一のコンピュータにジョブが展開されるイメージです。



実行時の挙動

Topコマンドで見ると、以下の例の様にCPUの利用率が100%前後で8つ実行されている例



OpenMPIのポイントは

```
!  Initialize MPI.
!
   call MPI_Init ( ierr )
!
!  Get this process's ID.
!
   call MPI_Comm_rank ( MPI_COMM_WORLD, MyId, ierr )
!
!  Find out how many processes are available.
!
   call MPI_Comm_size ( MPI_COMM_WORLD, process , ierr )
!
!  Terminate MPI.
!
   call MPI_Finalize ( ierr )
```

<——— 初期化処理、最初に必ず呼んでおく

<——— プロセスに付けられる固有の番号(ランク)を取得する
ここでは　MyId　としています。　0が親になります。

<——— 全てのプロセス数を取得します。　process　が該当

ここで主たる計算処理が行われる。

<——— 最終処理、最後に必ず呼んでおく

１～100までの加算での例

```
program main
!
  use mpi
!
!  Initialize MPI.
!
  call MPI_Init ( ierr )   <―――  初期化処理、最初に必ず呼んでおく
!                          <―――  プロセスに付けられる固有の番号(ランク)を取得する
!                                 ここでは  MyId  としています。  0が親になります。
  call MPI_Comm_rank ( MPI_COMM_WORLD, MyId, ierr )
  call MPI_Comm_size ( MPI_COMM_WORLD, process , ierr )<―――  全てのプロセス数を取得

  if ( MyId == 0 ) then
    write ( *, '(a)' ) '  An MPI example program'
  end if
```

ここで主たる計算処理が行われる。

```
  total = 0
    step  = 100/process
    istart= MyID * step + 1
    iend  = istart + step - 1

    do i = istart , iend
        total = total + i
    end do
```

アプリケーションの実行方法

```
print *,'Proc = ',process , '     MyID = ',MyId
  print *,'Total No = ', total, istart, iend

  call MPI_Reduce ( total, sumall, 1, MPI_INTEGER, MPI_SUM, 0, &
                    MPI_COMM_WORLD, ierr )
!  Terminate.
if ( MyId == 0 ) then
    write ( *, '(a)' ) ' '
    write ( *, '(a)' ) '  Normal end of execution.'
    write ( *, '(a,i20)' ) '              Summation is :',sumall
    write ( *, '(a)' ) ' '

    CALL system_clock(t2, t_rate, t_max)
    if ( t2 < t1 ) then
      diff = (t_max - t1) + t2 + 1
    else
      diff = t2 - t1
    endif
    WRITE(6,*) 'Time of operation was ', diff/dble(t_rate)
  end if
!  Terminate MPI.
call MPI_Finalize ( ierr )

  stop
end
```

```
   total = 0
     step  = 100/process
     istart= MyID * step + 1
     iend  = istart + step - 1

     do i = istart , iend
         total = total + i
     end do
```

```
$ mpif90 -o sum_mpi100 sum_mpi100.f90
support@HPG4:~/CAE/Sum_mpi
$ mpirun -np 4 ./sum_mpi100

  FORTRAN90/MPI version
  An MPI example program to print Message
                Process Number   =     4

Proc =           4     MyID =           0
Total No =                325           1       25
Proc =           4     MyID =           1
Total No =                950          26       50
Proc =           4     MyID =           2
Total No =               1575          51       75
Proc =           4     MyID =           3
Total No =               2200          76      100

 Normal end of execution.
           Summation is :              5050

 Time of operation was     1.01109730000000001E-002   sec
```

並列アプリケーションを作成できるプログラミング言語

1. Fortran

古くからある技術計算向き言語。今は教育でも教えていないようである。　機能は再帰機能も取り入れて進化。

2. C、C++

OS開発等に使われる高機能な言語、技術計算ではFortranに代わり使われるようになる。C++はオブジェクト指向。

3. Python

AI開発等に使われる言語でライブラリが豊富、SMP計算はライブラリ(numpy)が最適、MPIはライブラリ(mpi4py)を使用する。Pythonはオブジェクト指向。

クラウドをクラスタとして利用

クラスタとクラウドコンピューティング

クラウドとは、コンピューティングサービスをインターネット経由でサービスを提供します。

サービスは計算サーバー、データベースサーバー、アプリケーションサーバー、ストレージ、ネットワークなどがあります。

クラウドの中でHPCクラウドは大規模で高速な計算サービスを行うコンピュータの集合であり、基本的な要素はオンプレミスのクラスタの構成になります。

大きな違いはオンプレミスだと計算資源がわかり易いですが、AWSやAzureのような大規模なクラウドでは計算資源が判りづらくなっております。（資源とはCPUの周波数、タイプ、ネットワーク等）

FOCUSスパコンではHPCに特化している関係か、計算資源は非常にわかり易くなっております。

主なクラウド

- AWS（アマゾン　ウェブ　サービス）

  2006年にAmazon　EC2という仮想サーバーとAmazon　S3という仮想ストレージのサービスでクラウドサービスを開始。現在は非常に多くのサービスを提供している。HPCの分野で使うとするとAmazon　EC2　及び　AWS ParallelCluster になります。

- GCP（グーグル　クラウド　プラットホーム）

  2008年に　Google App Engineというアプリケーションの実行環境を提供開始したのが最初のサービス。HPC分野では　Compute　Engineを使ってOSのインストールからHPCシステムを構築できる。簡単な説明は以下のURLにあり、OSレベルの管理を知らないと対応が難しい。

  https://cloud.google.com/solutions/best-practices-for-using-mpi-on-compute-engine

➢ FOCUSスパコン

2011年に供用開始した日本の「京」「富岳」に関連した公立のクラウドサービス。

FOCUSスパコンは、スパコン利用企業層の拡大を目的に整備された産業利用向けの公的スーパーコンピュータ。システムはシミュレーション技術の活用による産業競争力強化のために幅広く利用することを目的とするほか、スーパーコンピュータ「京」および後継機「富岳」へのステップアップのためのテストベッドスパコンとしての役割を担っています。

クラウド利用での注意点

多くのクラウドがCPUの割当てで vCPU を使います。GCPの説明にありますが

ハイパースレッディングは、ノード上の物理コアごとに 2つの仮想コア（vCPU）を割当

多くの一般的なコンピューティング タスクや、大量の I/O を必要とするタスクの場合、ハイパー スレッディングによってアプリケーションのスループットを大幅に向上

計算依存型ジョブの場合、ハイパー スレッディングはアプリケーションの全体的なパフォーマンスを妨げる

ハイパー スレッディングを無効にすると、予測可能性に優れたパフォーマンスが可能になり、ジョブの時間を短縮可能

があり、vCPUについては注意が必要です。上記を十分理解の上、クラウドの設定をする必要があります。

【FOCUSスパコン】

マシンの詳しいスペックがわかるので、具体的に契約する場合は価格表を提示して頂けます。

https://www.j-focus.or.jp/focus/fee.html

以下は1時間単位での利用料金例、この他に1日単位、1ヶ月単位等あり、長期の方が安くなる。

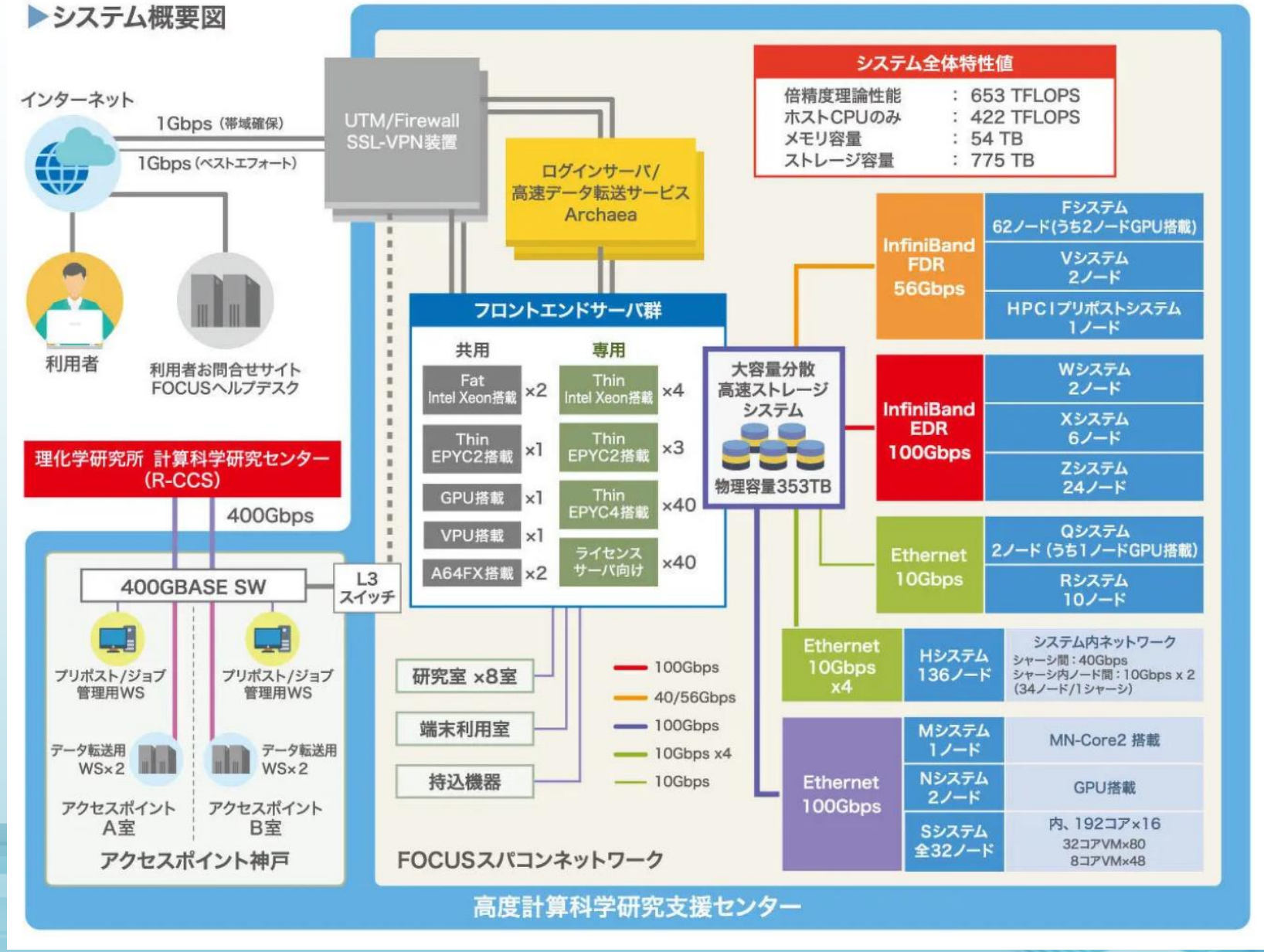
2. 共用計算資源の従量利用　初年度特典 無料枠対象

項目	ノード数	金額
Fシステム (GPU非搭載)	1～4ノード	300円
	5～8ノード	270円
	9～12ノード	240円
	13～16ノード	210円
	17～20ノード	180円
	21ノード以上	150円
Fシステム (GPU搭載)	1ノード以上	400円
Hシステム	1～4ノード	60円
	5～8ノード	55円
	9～12ノード	50円
	13～16ノード	45円
	17～20ノード	40円
	21ノード以上	35円
Rシステム	1～4ノード	300円
	5～8ノード	270円
	9ノード以上	240円
Sシステム（8コアVM）	1～4ノード	60円
	5～8ノード	55円
	9～12ノード	50円
	13～16ノード	45円
	17～20ノード	40円
	21ノード以上	35円
Sシステム（32コアVM）	1～4ノード	320円
	5～8ノード	288円
	9～12ノード	256円
	13～16ノード	224円
	17～20ノード	192円
	21ノード以上	160円

システム概要

各システムノード性能

	ノード数	ホストCPU、デバイス	コア数	メモリ(コアあたり)[GB]
Fシステム	60	Intel Xeon E5-2698v4 (2.2GHz) ×2基	40	128 (3.2)
	2	Intel Xeon E5-2698v4 (2.2GHz) ×2基	40	128 (3.2)
		NVIDIA Tesla P100/PCI ×1基	–	16 –
Hシステム	136	Intel Xeon D-1541 (2.1GHz) ×1基	8	64 (8)
Mシステム	1	Intel Xeon Platinum 8480+(2GHz) ×2基	112	1024 (9.1)
		MN-Core 2×8基	–	–(–)
Nシステム	2	AMD EPYC4 9354P(3.25GHz) ×1基	32	384 (12)
		NVIDIA H100 NVL 94GiB ×1基	–	94(–)
Qシステム	1	AMD EPYC 7713P (2GHz) ×1基	64	512 (8)
	1	AMD EPYC 7543 (2.8GHz) ×2基	64	512 (8)
		NVIDIA A100 ×2基	–	160 –
Rシステム	10	AMD EPYC 7543P (2.8GHz) ×1基	32	256 (8)
Sシステム	32	AMD EPYC 9654 (2.4GHz) ×2基	192	768 (-)



SMPアプリケーション

SMPアプリケーションの簡単な例

簡単な例：

```
      print *,"単一処理部分"

!$omp parallel                          ! Fork開始
      print *,"並列処理部分"
!$omp end parallel                      !Join開始

      print *,"単一処理部分"
```

プログラム

```
$ gfortran -o test -fopenmp Check_openMP.f90
```

コンパイル／リンク

4並列で実行した場合

```
$ export set OMP_NUM_THREADS=4
$ ./test

PRIME_NUMBER_OPENMP
  FORTRAN90/OpenMP version

  Number of processors available =  4
  Number of threads =               4
 単一処理部分
 並列処理部分
 並列処理部分
 並列処理部分
 並列処理部分
 単一処理部分
```

2並列で実行した場合

```
$ export set OMP_NUM_THREADS=2
$ ./test

PRIME_NUMBER_OPENMP
  FORTRAN90/OpenMP version

  Number of processors available =  4
  Number of threads =               2
 単一処理部分
 並列処理部分
 並列処理部分
 単一処理部分
```

並列処理部

簡単な例：

```
      n  = 12
      total = 0

!$omp parallel
!$omp do reduction ( + : total )
      do i=1,n
         total = total + i
         write(6,*) "Loop num: ",i,"total : ",total,"Thread N: ",omp_get_thread_num()
      end do
!$omp end do
!$omp end parallel
```

並列部

```
$ gfortran -o sum -fopenmp sum_openMP.f90
```

コンパイル／リンク

Reduction　の使い方？

Reductionは指定した変数が各スレッドで別々に計算されますが、それぞれの計算されたものが演算子を使って最終的に処理をされます。英語の辞書では　縮小、縮図　等の意味があるようです。

以下の場合は別々に計算された　total　の値の総和（＋）が並列処理が終了した時に全てを足したtotalとして返されます。

```
!$omp parallel
!$omp do reduction ( + : total )

  do i = 1, n
    total = total + i
  end do

!omp end do                    <———　並列のdo区間の最終を指示
!$omp end parallel             <———　並列区間の最終を指示
```

4並列で実行した場合

Totalは、0:6 + 1:15 + 2:24 + 3:33 で 78 になり、正しい1～12の結果を返します。

```
Number of processors available =        4
Number of threads =                     4
Loop num:          1 total :          1 Thread N:          0
Loop num:          4 total :          4 Thread N:          1
Loop num:          5 total :          9 Thread N:          1
Loop num:          6 total :         15 Thread N:          1
Loop num:         10 total :         10 Thread N:          3
Loop num:         11 total :         21 Thread N:          3
Loop num:         12 total :         33 Thread N:          3
Loop num:          2 total :          3 Thread N:          0
Loop num:          3 total :          6 Thread N:          0
Loop num:          7 total :          7 Thread N:          2
Loop num:          8 total :         15 Thread N:          2
Loop num:          9 total :         24 Thread N:          2
```

スレッド０

スレッド１

スレッド２

スレッド０

スレッド３

MPIアプリケーション

MPIアプリケーションの簡単な例

```
!  Initialize MPI.
!
   call MPI_Init ( ierr )                    <―――  初期化処理、最初に必ず呼んでおく

!
!  Get this process's ID.
!                                <―――  プロセスに付けられる固有の番号(ランク)を取得する
!                                        ここでは  MyId  としています。  0が親になります。
   call MPI_Comm_rank ( MPI_COMM_WORLD, MyId, ierr )
!
!  Find out how many processes are available.
!                                <―――  全てのプロセス数を取得します。  process  が該当
   call MPI_Comm_size ( MPI_COMM_WORLD, process , ierr )


!
!  Terminate MPI.
!
   call MPI_Finalize ( ierr )                <―――  最終処理、最後に必ず呼んでおく
```

前処理部分

```
program main
!
      use mpi
!
      IMPLICIT REAL*8(A-H,O-Z)
      INTEGER*8 i, n, ns, nn, istart, iend, t1, t2, t_rate, t_max, diff
      integer*4  totalProc, useProc,  p, ierr
!
!  Initialize MPI.
!
      call MPI_Init ( ierr )
!
!  Get this process's ID.
!
      call MPI_Comm_rank ( MPI_COMM_WORLD, id, ierr )
!
!  Find out how many processes are available.
!
      call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )

      if ( id == 0 ) then
!
        CALL system_clock(t1)
!
        write ( *, '(a)' ) ' '

        write ( *, '(a)' ) 'Pai_Calculation_with_MPI'
        write ( *, '(a)' ) '  FORTRAN90/MPI version'

        write ( *, '(a,i8)' ),'  The number of processes is ', p
        write ( *, '(a)' ) ' '
      end if
```

use mpi
  MPIライブラリの指定

call MPI_Init (ierr)
  MPI利用の初期化

call MPI_Comm_rank (MPI_COMM_WORLD, id, ierr)
  MPI内でのidを取得、　0から始まる

call MPI_Comm_size (MPI_COMM_WORLD, p, ierr)
  MPIの並列数の取得、　pが並列数

計算処理部分

```
      n  = 2000000
      n  = n*20000
      istart = id*(n/p) + 1
      iend = istart + n/p
      pai = 0.0d+0

      do i = istart, iend
         xlow = 2 * (i-1) + 1
         if ( mod(i,2) .eq. 0) then
           xup1 = -1.0d+0
         else
           xup1 =  1.0d+0
         end if
         cal  = xup1 / xlow
         pai  = pai + cal * 4.0d+0
      end do

 call MPI_Reduce ( pai, pai_reduce, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, &
                                MPI_COMM_WORLD, ierr )
```

400億回の繰返しを指定

各並列計算領域をistart、　iendで指定して計算をする

call MPI_Reduce (pai, pai_reduce, 1,
　　計算した結果を　pai　に渡し、ランク0で
　　その結果を総和する。

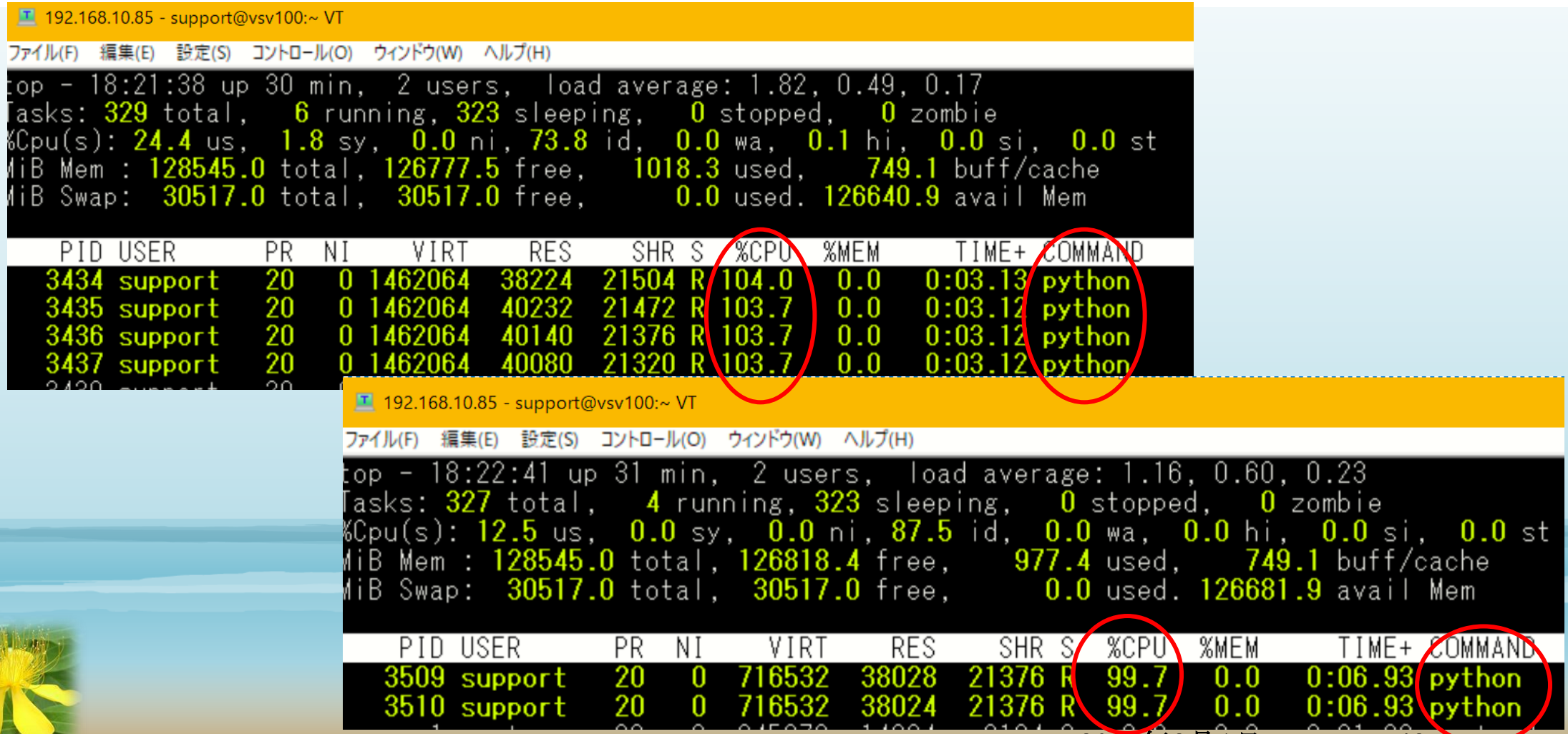
後処理部分

```
if ( id == 0 ) then
        WRITE(6,*)  'n=',n,',  pi=',pai
        CALL system_clock(t2, t_rate, t_max)
        if ( t2 < t1 ) then
          diff = (t_max - t1) + t2 + 1
        else
          diff = t2 - t1
        endif
        WRITE(6,*) 'Time of operation was ', diff/dble(t_rate) , ' sec'
      end if
!
!  Terminate MPI.
!
      call MPI_Finalize ( ierr )

      STOP
```

ランク0（親プロセス）で経過時間を求めて出力

【OSからの確認】

Windowsで並列計算の挙動を確認するのは難しいのでLinux（CentOS）のTopコマンドを使い、Linuxでシステムの動きを確認しました。Topコマンドの結果を見ると指定した並列数で動いているのが判ります。下記は４並列、２並列を指定した場合のTopコマンドの結果です。



ジョブ投入システム

slurmの利用例

slurm

Slurm（Simple Linux Utility for Resource Management）は、LinuxおよびUnix系のカーネルのためのフリーでオープンソースなジョブスケジューラーです。世界中の多くのスーパーコンピューターやコンピュータークラスタで使用されている。Slurmは3つの主要な機能を提供しています。

1. 計算を実行するユーザに対してリソース（コンピューターノード）への排他的・非排他的なアクセスを割り当てる機能。
2. 割り当てられたノードの集合上でのジョブの開始、実行、モニタリング（MPIなどの並列ジョブでよく使用される）を行う機能。
3. 待機中のジョブのキューを管理することで、リソースへの競合を解決する機能。

Slurmは、TOP500の約60%のスーパーコンピューターでワークロードマネージャーとして使用されています。

Slurmの構成

Slurmの構成は以下の様になっております。

主な機能を挙げると、

- 管理サーバー(slurmctld)

  ユーザーのジョブを受け取り，実行サーバー slurmd に実行を指示します。

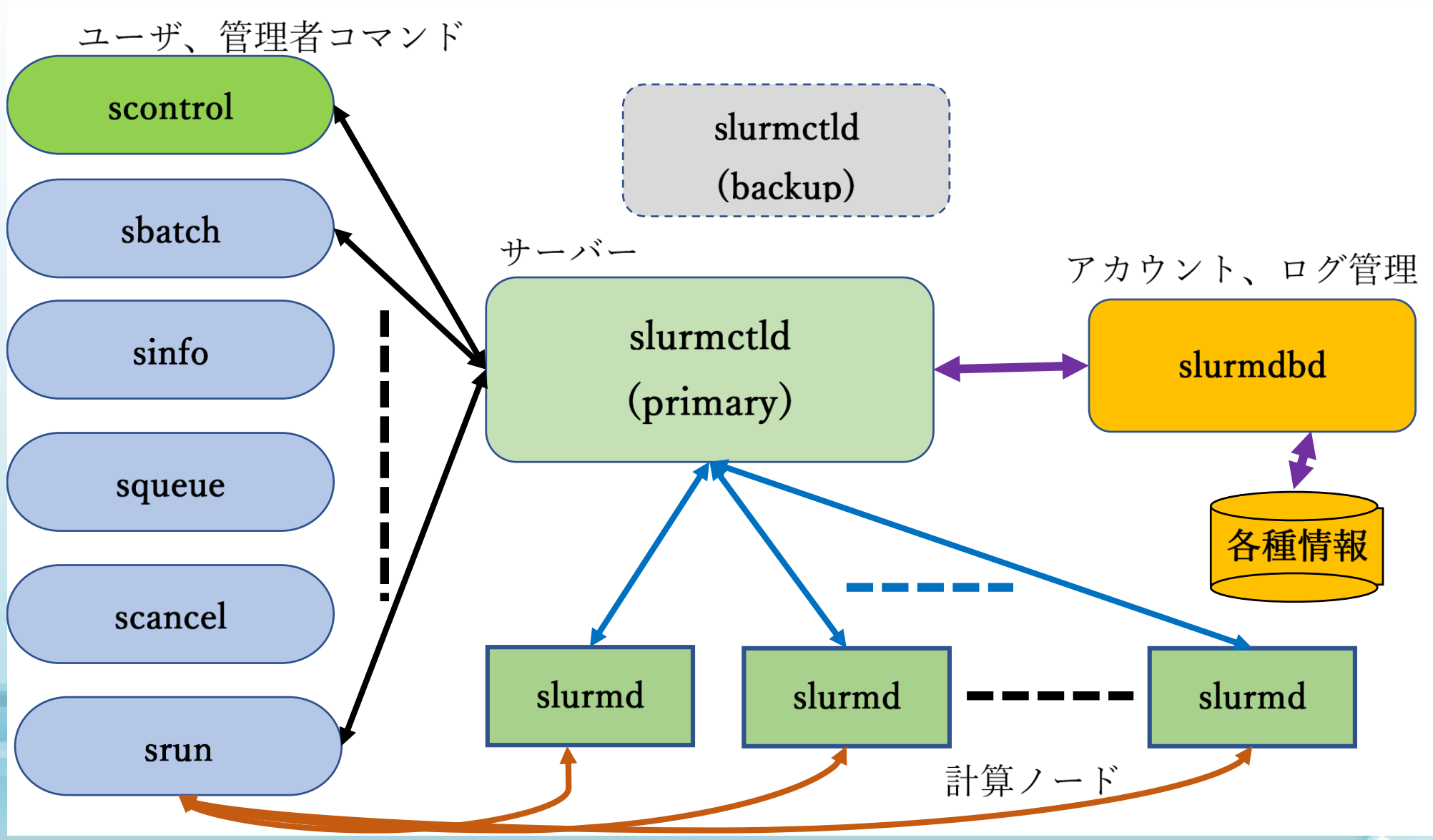
- 実行サーバー（slurmd）

  管理サーバーの指示を受けてユーザのジョブを実行します。

- データベースサーバー(slurmdbd)

  実行時間、ユーザ情報等の各種情報を保存します。

が基本になります。また信頼性を上げるためにBackUpサーバーを置いて,Primaryサーバーに障害が発生した場合、BackUpサーバーが機能を引き継ぐようにすることができます


ユーザ、管理者コマンド
scontrol
sbatch
sinfo
squeue
scancel
srun
slurmctld
(backup)
サーバー
slurmctld
(primary)
アカウント、ログ管理
slurmdbd
各種情報
slurmd
slurmd
slurmd
計算ノード

slurmの主なコマンド・オプション

- sbatch　[OPTIONS...] executable [args...]

-b　--begin=<time>　　例:　18:00:07

指定した時刻にジョブを起動する。

--begin=16:00

--begin=now+1hour

--begin=now+60　(seconds by default)

--begin=2010-01-20T12:34:00

-d　--dependency=<state:jobid>

指定したジョブが指定した状態になるまでジョブを待機させる。

after:job_id[:jobid...]

afterany:job_id[:jobid...]

afternotok:job_id[:jobid...]

afterok:job_id[:jobid...]

-e　--error=<filename>

エラーメッセージを出力するファイル名の指定

-D, --chdir=<directory>

作業ディレクトリの指定

-J　--job-name=<name>

　　ジョブ名の指定

--mem=<MB>

　　ノード辺りの割当てるメモリ量

--mem-per-cpu=<MB>

　　割当てたCPU当たりのメモリ量

-N<minodes[-maxnodes]>

　　ジョブに割当てるノード数

-n, --ntasks=<number>

　　起動するタスク数

-w, --nodelist=<naes>

　　ジョブを割り当てるホスト名

-o　--output=<name>

　　ジョブの標準出力先ファイル名

-p　--partition=<names>

　ジョブを実行するパーティション(キュー)名

-t　--time=<time>

　　実行時の経過時間の制限／単位は分

ファイル名に使えるシンボル

```
filename pattern
        %%      The character "%".
        %J      jobid.stepid of the running job. (e.g. "128.0")
        %j      jobid of the running job.
        %N      short hostname.
        %n      Node  identifier relative to current job
                  (e.g. "0" is the first node of the running job)
        %u      User name.
        %x      Job name.
        %t      task 識別子.

   sample
        job%J.out       job128.0.out
        job%4j.out      job0128.out             4桁でジョブ番号を表示
        job%j-%2t.out   job128-00.out, job128-01.out, ...
```

Slurmシェルスクリプトのサンプル

```
$ cat pai_calc_mpi.sh
#!/bin/sh
#SBATCH -J pai_calc
#SBATCH -o output_pai%j.log
#SBATCH -e errors_pai%j.log
#SBATCH -n 4
#SBATCH -p batch

mpifort -o pai_MPI pai_MPI.f90
date
mpirun -np 4  ./pai_MPI
date
```

自由な質疑(Q&A)