

# **3D Parallel FEM (IV)**

## **(OpenMP + MPI) Hybrid Parallel Programming Model**

Kengo Nakajima  
Information Technology Center

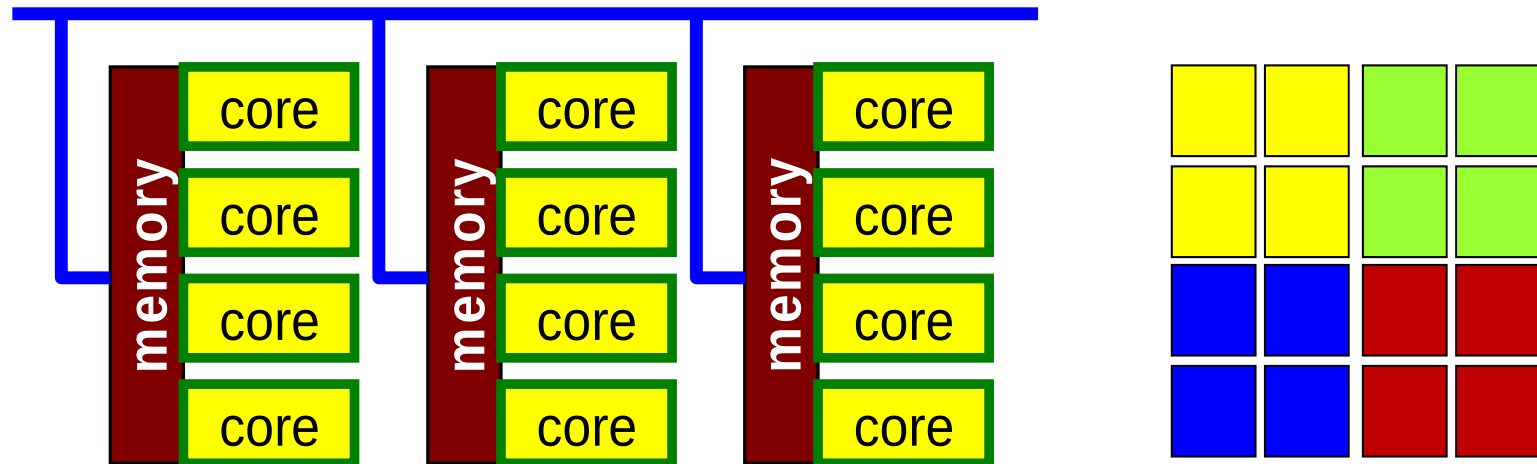
Technical & Scientific Computing II (4820-1028)  
Seminar on Computer Science II (4810-1205)  
Hybrid Distributed Parallel Computing (3747-111)

# Hybrid Parallel Programming Model

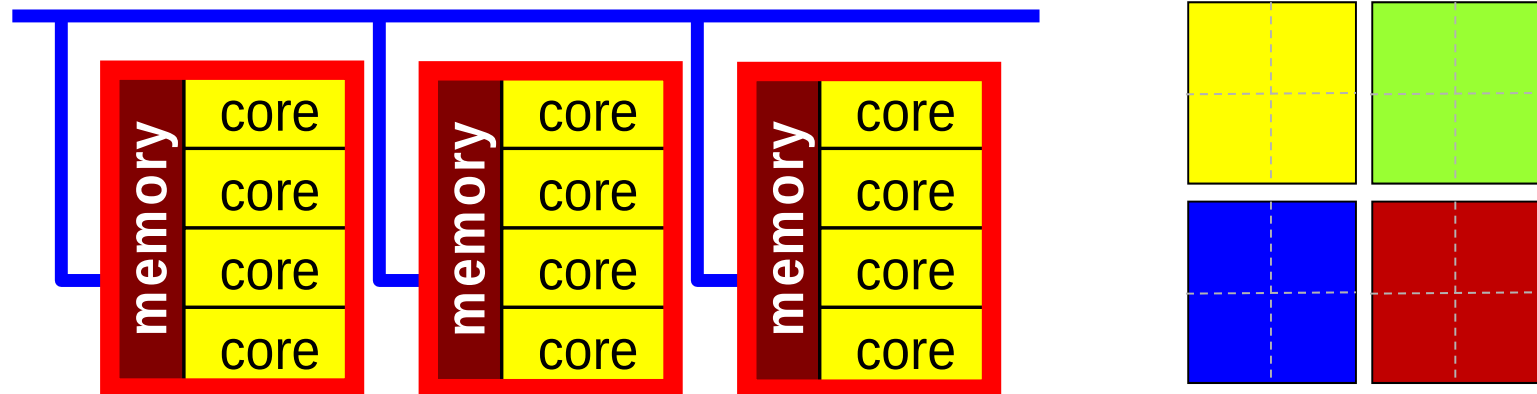
- Message Passing (e.g. MPI) + Multi Threading (e.g. OpenMP, CUDA, OpenCL, OpenACC etc.)
- In K computer and FX10, hybrid parallel programming is recommended
  - MPI + Automatic Parallelization by Fujitsu's Compiler
    - Personally, I do not like to call this “hybrid” !!!
- Expectations for Hybrid
  - Number of MPI processes (and sub-domains) to be reduced
  - $O(10^8-10^9)$ -way MPI might not scale in Exascale Systems
  - Easily extended to Heterogeneous Architectures
    - CPU+GPU, CPU+Manycores (e.g. Intel MIC/Xeon Phi)
    - MPI+X: OpenMP, OpenACC, CUDA, OpenCL

# Flat MPI vs. Hybrid

## Flat-MPI: Each Core -> Independent



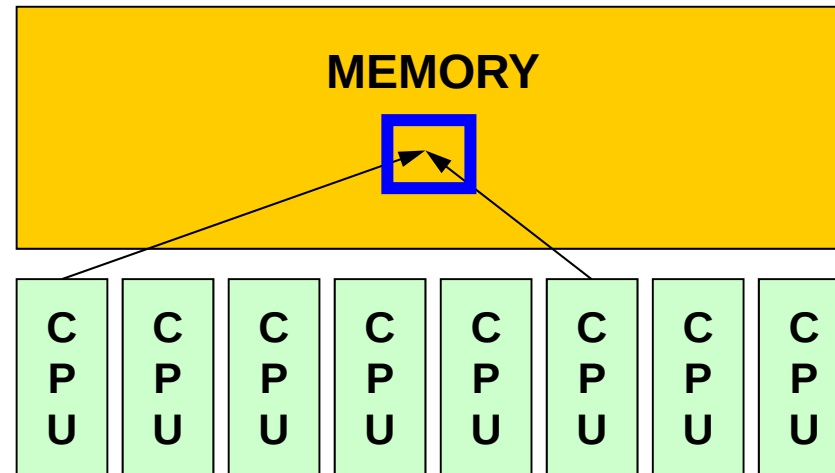
## Hybrid: Hierarchical Structure



# Background

- Multicore/Manycore Processors
  - Low power consumption, Various types of programming models
- OpenMP
  - Directive based, (seems to be) easy
  - Many books
- Data Dependency (S1/S2 Semester)
  - Conflict of reading from/writing to memory
  - Appropriate reordering of data is needed for “consistent” parallel computing
  - NO detailed information in OpenMP books: very complicated
- OpenMP/MPI Hybrid Parallel Programming Model for Multicore/Manycore Clusters

# SMP



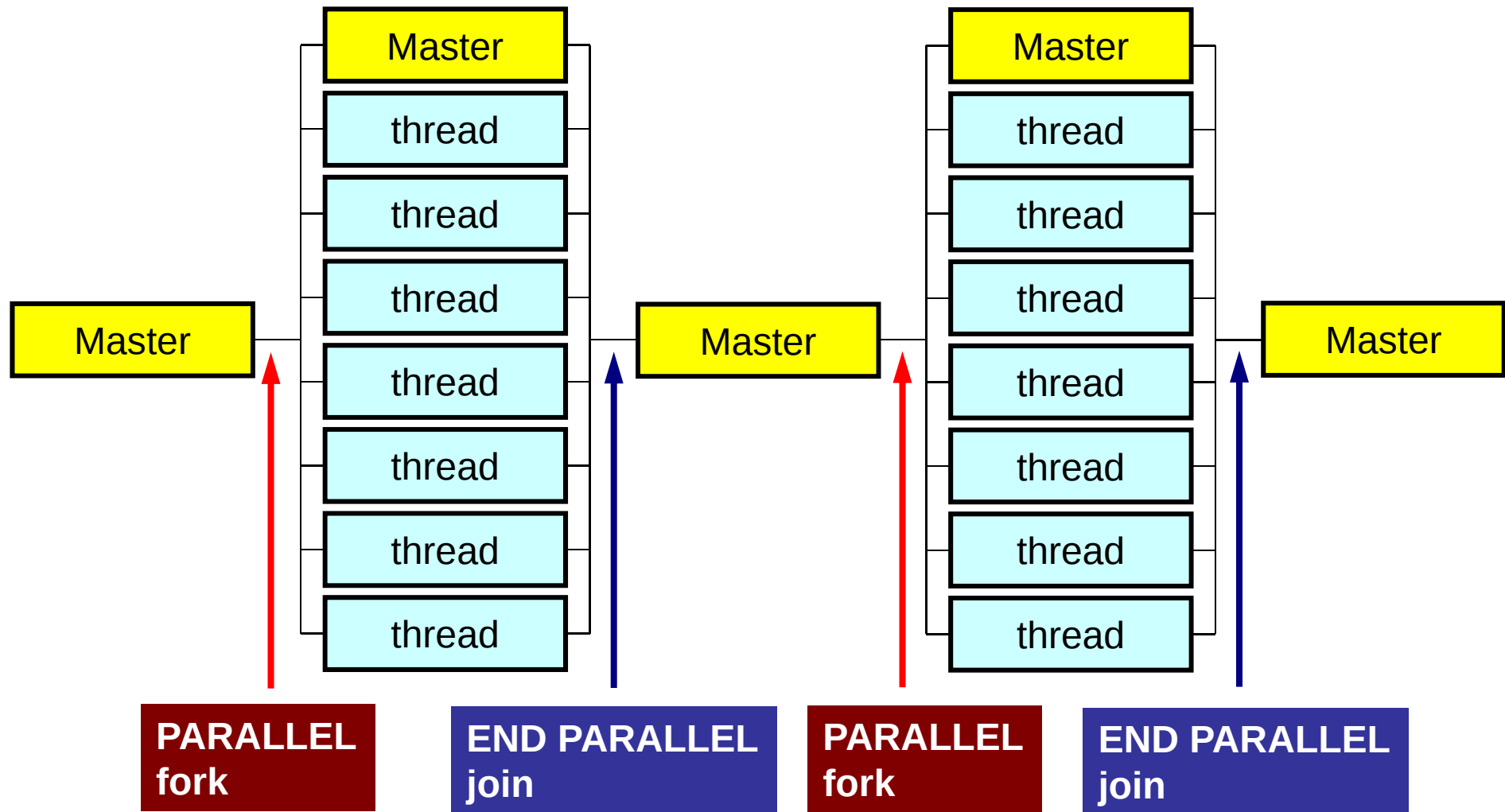
- SMP
  - Symmetric Multi Processors
  - Multiple CPU's (cores) share a single memory space

# What is OpenMP ?

<http://www.openmp.org>

- An API for multi-platform shared-memory parallel programming in C/C++ and Fortran
  - Current version: 4.0
- Background
  - Merger of Cray and SGI in 1996
  - ASCI project (DOE) started
- C/C++ version and Fortran version have been separately developed until ver.2.5.
- Fork-Join Parallel Execution Model
- Users have to specify everything by directives.
  - Nothing happen, if there are no directives

# Fork-Join Parallel Execution Model



# Number of Threads

- **OMP\_NUM\_THREADS**

- How to change ?

- `bash(.bashrc)`

- `export OMP_NUM_THREADS=8`

- `csch(.cschrc)`

- `setenv OMP_NUM_THREADS 8`

# Information about OpenMP

- OpenMP Architecture Review Board (ARB)
  - <http://www.openmp.org>
- References
  - Chandra, R. et al.「Parallel Programming in OpenMP」(Morgan Kaufmann)
  - Quinn, M.J.「Parallel Programming in C with MPI and OpenMP」(McGrawHill)
  - Mattson, T.G. et al.「Patterns for Parallel Programming」(Addison Wesley)
  - 牛島「OpenMPによる並列プログラミングと数値計算法」(丸善)
  - Chapman, B. et al.「Using OpenMP」(MIT Press)
- Japanese Version of OpenMP 3.0 Spec. (Fujitsu etc.)
  - <http://www.openmp.org/mp-documents/OpenMP30spec-ja.pdf>

# Features of OpenMP

- Directives
  - Loops right after the directives are parallelized.
  - If the compiler does not support OpenMP, directives are considered as just comments.

# OpenMP/Directives

## Array Operations

### Simple Substitution

```
!$omp parallel do
  do i= 1, NP
    W(i, 1)= 0. d0
    W(i, 2)= 0. d0
  enddo
!$omp end parallel do
```

### DAXPY

```
!$omp parallel do
  do i= 1, NP
    Y(i)= ALPHA*X(i) + Y(i)
  enddo
!$omp end parallel do
```

### Dot Products

```
!$omp parallel do private(iS, iE, i)
!$omp&      reduction(+:RHO)
  do ip= 1, PEsmptOT
    iS= STACKmcG(ip-1) + 1
    iE= STACKmcG(ip )
    do i= iS, iE
      RHO= RHO + W(i, R)*W(i, Z)
    enddo
  enddo
!$omp end parallel do
```

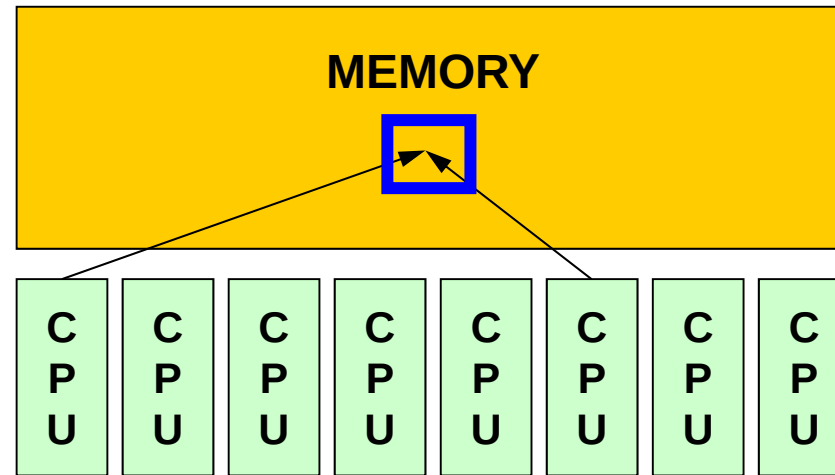
# OpenMP/Direceives Matrix/Vector Products

```
!$omp parallel do private(ip, iS, iE, i, j)
  do ip= 1, PEsmptOT
    iS= STACKmcG(ip-1) + 1
    iE= STACKmcG(ip )
    do i= iS, iE
      W(i, Q)= D(i)*W(i, P)
      do j= 1, INL(i)
        W(i, Q)= W(i, Q) + W(IAL(j, i), P)
      enddo
      do j= 1, INU(i)
        W(i, Q)= W(i, Q) + W(IAU(j, i), P)
      enddo
    enddo
  enddo
!$omp end parallel do
```

# Features of OpenMP

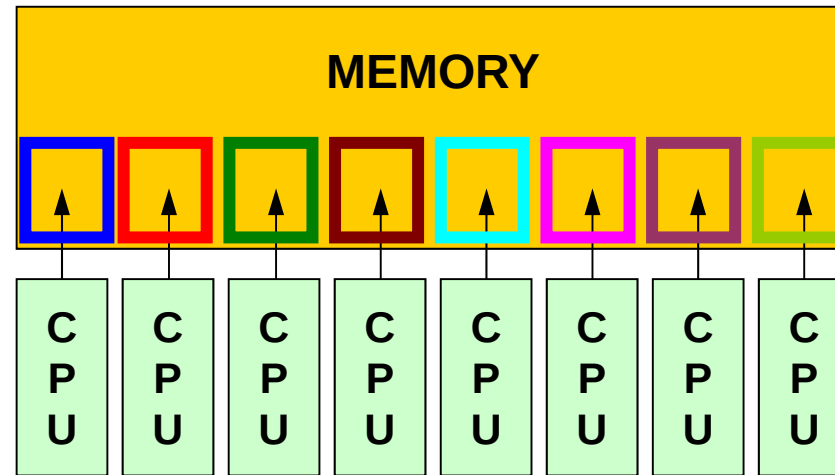
- Directives
  - Loops right after the directives are parallelized.
  - If the compiler does not support OpenMP, directives are considered as just comments.
- Nothing happen without explicit directives
  - Different from “automatic parallelization/vectorization”
  - Something wrong may happen by un-proper way of usage
  - Data configuration, ordering etc. are done under users’ responsibility
- “Threads” are created according to the number of cores on the node
  - Thread: “Process” in MPI
  - Generally, “# threads = # cores”: Xeon Phi supports 4 threads per core (Hyper Multithreading)

# Memory Contention: メモリ競合



- During a complicated process, multiple threads may simultaneously try to update the data in same address on the memory.
  - e.g.: Multiple cores update a single component of an array.
  - This situation is possible.
  - Answers may change compared to serial cases with a single core (thread).

# Memory Contention (cont.)



- In this lecture, no such case does not happen by reordering etc.
  - In OpenMP, users are responsible for such issues (e.g. proper data configuration, reordering etc.)
- Generally speaking, performance per core reduces as number of used cores (thread number) increases.
  - Memory access performance: STREAM

# Features of OpenMP (cont.)

- “!omp parallel do”-“!omp end parallel do”
- Global (Shared) Variables, Private Variables
  - Default: Global (Shared)
  - Dot Products: reduction

```
!$omp parallel do private(iS, iE, i)  
!$omp&                reduction(+:RH0)  
    do ip= 1, PEsmptOT  
        iS= STACKmcG(ip-1) + 1  
        iE= STACKmcG(ip )  
        do i= iS, iE  
            RH0= RH0 + W(i, R)*W(i, Z)  
        enddo  
    enddo  
!$omp end parallel do
```

W(:, :), R, Z, PEsmptOT  
global (shared)

# FORTRAN & C

```
use omp_lib
...
!$omp parallel do shared(n, x, y) private(i)
    do i= 1, n
        x(i)= x(i) + y(i)
    enddo
!$ omp end parallel do
```

```
#include <omp.h>
{
    #pragma omp parallel for default(none) shared(n, x, y) private(i)

    for (i=0; i<n; i++)
        x[i] += y[i];
}
```

# In this class ...

- There are many capabilities of OpenMP.
- In this class, only several functions are shown for parallelization of parallel FEM.

# First things to be done (after OpenMP 3.0)

- `use omp_lib` Fortran
- `#include <omp.h>` C

# OpenMP Directives (Fortran)

```
sentinel directive_name [clause[[,] clause]...]
```

- NO distinctions between upper and lower cases.
- sentinel
  - Fortran: !\$OMP, C\$OMP, \*\$OMP
    - !\$OMP only for free format
  - Continuation Lines (Same rule as that of Fortran compiler is applied)
    - Example for !\$OMP PARALLEL DO SHARED (A, B, C)

```
!$OMP PARALLEL DO  
!$OMP+SHARED (A,B,C)
```

```
!$OMP PARALLEL DO &  
!$OMP SHARED (A,B,C)
```

# OpenMP Directives (C)

```
#pragma omp directive_name [clause[[,] clause]...]
```

- “\” for continuation lines
- Only lower case (except names of variables)

```
#pragma omp parallel for shared (a,b,c)
```

# PARALLEL DO

```
!$OMP PARALLEL DO[clause[[,] clause] ... ]  
    (do_loop)  
!$OMP END PARALLEL DO
```

```
#pragma parallel for [clause[[,] clause] ... ]  
    (for_loop)
```

- Parallerize DO/for Loops
- Examples of “clause”
  - PRIVATE(list)
  - SHARED(list)
  - DEFAULT(PRIVATE|SHARED|NONE)
  - REDUCTION({operation|intrinsic}: list)

# REDUCTION

```
REDUCTION ({operator|instinsic}: list)
```

```
reduction ({operator|instinsic}: list)
```

- Similar to “MPI\_Reduce”
- Operator
  - +, \*, -, .AND., .OR., .EQV., .NEQV.
- Intrinsic
  - MAX, MIN, IAND, IOR, IEQR

# Example-1: A Simple Loop

```
!$OMP PARALLEL DO  
  do i= 1, N  
    B(i)= (A(i) + B(i)) * 0.50  
  enddo  
!$OMP END PARALLEL DO
```

- Default status of loop variables (“i” in this case) is private. Therefore, explicit declaration is not needed.
- “END PARALLEL DO” is not required
  - In C, there are no definitions of “end parallel do”

# Example-1: REDUCTION

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+:A,B)  
  do i= 1, N  
    call WORK (Alocal, Blocal)  
    A= A + Alocal  
    B= B + Blocal  
  enddo  
!$OMP END PARALLEL DO
```

- “END PARALLEL DO” is not required

# Functions which can be used with OpenMP

| Name                                                                                                           | Functions        |
|----------------------------------------------------------------------------------------------------------------|------------------|
| <code>int omp_get_num_threads (void)</code>                                                                    | Total Thread #   |
| <code>int omp_get_thread_num (void)</code>                                                                     | Thread ID        |
| <code>double omp_get_wtime (void)</code>                                                                       | = MPI_Wtime      |
| <code>void omp_set_num_threads (int num_threads)</code><br>call <code>omp_set_num_threads (num_threads)</code> | Setting Thread # |

# OpenMP for Dot Products

```
VAL= 0. d0  
do i= 1, N  
    VAL= VAL + W(i, R) * W(i, Z)  
enddo
```

# OpenMP for Dot Products

```
VAL= 0. d0  
do i= 1, N  
  VAL= VAL + W(i, R) * W(i, Z)  
enddo
```



```
VAL= 0. d0  
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:VAL)  
do i= 1, N  
  VAL= VAL + W(i, R) * W(i, Z)  
enddo  
!$OMP END PARALLEL DO
```

Directives are just inserted.

# OpenMP for Dot Products

```
VAL= 0. d0
do i= 1, N
  VAL= VAL + W(i, R) * W(i, Z)
enddo
```



```
VAL= 0. d0
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:VAL)
do i= 1, N
  VAL= VAL + W(i, R) * W(i, Z)
enddo
!$OMP END PARALLEL DO
```

Directives are just inserted.



```
VAL= 0. d0
!$OMP PARALLEL DO PRIVATE(ip, i) REDUCTION(+:VAL)
do ip= 1, PEsmptOT
  do i= index(ip-1)+1, index(ip)
    VAL= VAL + W(i, R) * W(i, Z)
  enddo
enddo
!$OMP END PARALLEL DO
```

Multiple Loop

**PEsmptOT**: Number of threads

Additional array **INDEX ( : )** is needed.

Efficiency is not necessarily good, but users can specify thread for each component of data.

# OpenMP for Dot Products

```
VAL= 0. d0
!$OMP PARALLEL DO PRIVATE(ip, i) REDUCTION(+:VAL)
do ip= 1, PEsmptOT
  do i= index(ip-1)+1, index(ip)
    VAL= VAL + W(i,R) * W(i,Z)
  enddo
enddo
!$OMP END PARALLEL DO
```

Multiple Loop

**PEsmptOT**: Number of threads

Additional array **INDEX ( : )** is needed.

Efficiency is not necessarily good, but users can specify thread for each component of data.

e.g.: N=100, PEsmptOT=4

```
INDEX(0)= 0
INDEX(1)= 25
INDEX(2)= 50
INDEX(3)= 75
INDEX(4)= 100
```

NOT good for GPU's

# Matrix-Vector Multiply

```
do i = 1, N
  VAL= D(i)*W(i, P)
  do k= indexL(i-1)+1, indexL(i)
    VAL= VAL + AL(k)*W(itemL(k), P)
  enddo
  do k= indexU(i-1)+1, indexU(i)
    VAL= VAL + AU(k)*W(itemU(k), P)
  enddo
  W(i, Q)= VAL
enddo
```

# Matrix-Vector Multiply

```

!$omp parallel do private(ip, i, VAL, k)
  do ip= 1, PEsmptOT
    do i = INDEX(ip-1)+1, INDEX(ip)
      VAL= D(i)*W(i, P)
      do k= indexL(i-1)+1, indexL(i)
        VAL= VAL + AL(k)*W(itemL(k), P)
      enddo
      do k= indexU(i-1)+1, indexU(i)
        VAL= VAL + AU(k)*W(itemU(k), P)
      enddo
      W(i, Q)= VAL
    enddo
  enddo
!$omp end parallel do

```

# Matrix-Vector Multiply: Other Approach

This is rather better for GPU and (very) many-core architectures: simpler structure of loops

```
!$omp parallel do private(i, VAL, k)
  do i = 1, N
    VAL= D(i)*W(i, P)
    do k= indexL(i-1)+1, indexL(i)
      VAL= VAL + AL(k)*W(itemL(k), P)
    enddo
    do k= indexU(i-1)+1, indexU(i)
      VAL= VAL + AU(k)*W(itemU(k), P)
    enddo
    W(i, Q)= VAL
  enddo
!$omp end parallel do
```

# omp parallel (do)

- Each “omp parallel-omp end parallel” pair starts & stops threads: fork-join
- If you have many loops, these operations on threads could be overhead
- omp parallel + omp do/omp for

```
!$omp parallel ...
```

```
!$omp do  
    do i= 1, N
```

```
...
```

```
!$omp do  
    do i= 1, N
```

```
...
```

```
!$omp end parallel 必須
```

```
#pragma omp parallel ...
```

```
#pragma omp for {
```

```
...
```

```
#pragma omp for {
```

# Exercise !!

- Apply multi-threading by OpenMP on parallel FEM code using MPI
  - CG Solver (solver\_CG, solver\_SR)
  - Matrix Assembling (mat\_ass\_main, mat\_ass\_bc)
- Hybrid parallel programming model
- Evaluate the effects of
  - Problem size, parallel programming model, thread #

# OpenMP (Only Solver) (F·C)

```
>$ cd <$O-TOP>/pfem3d/src1
```

```
>$ make
```

```
>$ cd ../run
```

```
>$ ls sol1
```

```
sol1
```

```
>$ cd ../pmesh
```

```
<Parallel Mesh Generation>
```

```
>$ cd ../run
```

```
<modify go1.sh>
```

```
>$ pjsub go1.sh
```

# Makefile (Fortran)

```

F90      = mpiifort
F90LINKER = $(F90)
LIB_DIR  =
INC_DIR  =
OPTFLAGS = -O3 -xCORE-AVX2 -align array32byte -qopenmp
FFLAGS   = $(OPTFLAGS)
FLIBS    =
F90LFLAGS=
#
TARGET   = ../run/sol1
default: $(TARGET)
OBJS     =
pfem_util.o ...

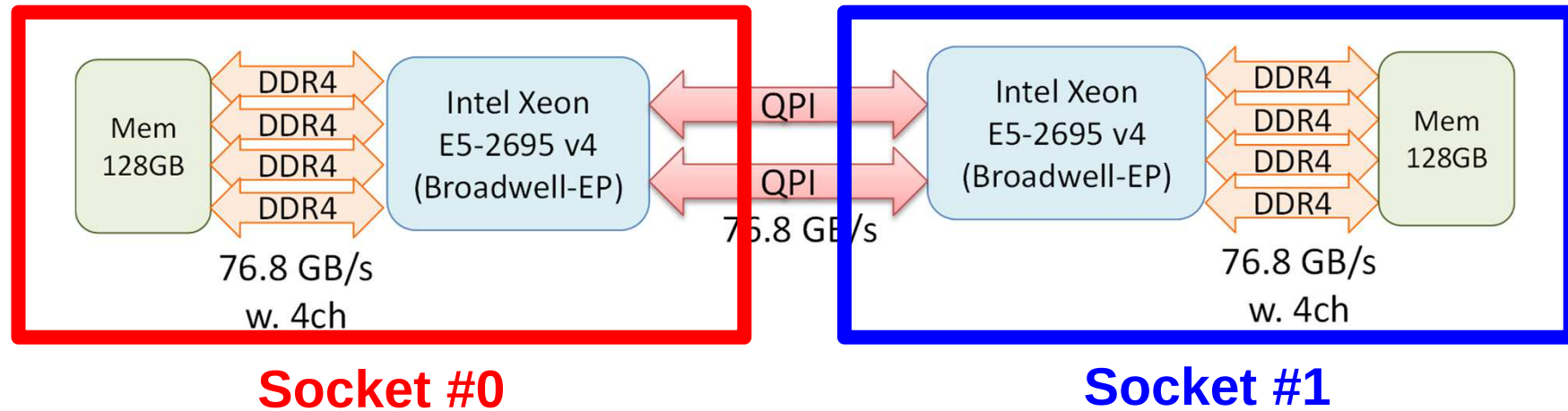
$(TARGET): $(OBJS)
            $(F90LINKER) $(OPTFLAGS) -o $(TARGET) $(OBJS) $(F90LFLAGS)
clean:
            /bin/rm -f *.o $(TARGET) *~ *.mod
.f.o:
            $(F90) $(FFLAGS) $(INC_DIR) -c  $*.f
.f90.o:
            $(F90) $(FFLAGS) $(INC_DIR) -c  $*.f90
.SUFFIXES: .f90 .f

```

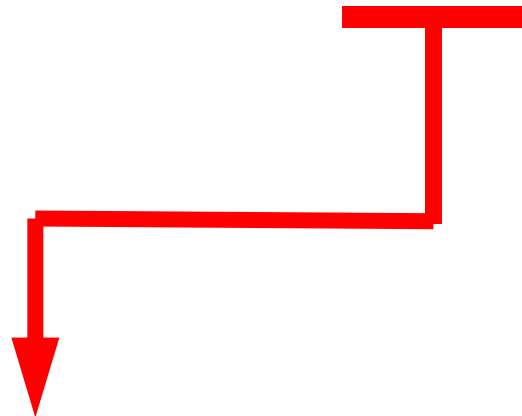
# Makefile(C)

```
CC      = mpiicc
LIB_DIR=
INC_DIR=
OPTFLAGS= -O3 -xCORE-AVX2 -align -qopenmp
LIBS =
LFLAGS=
#
TARGET = ../run/sol1
default: $(TARGET)
OBJS =\
    test1.o\...

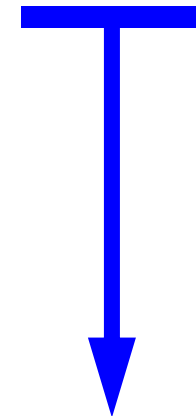
$(TARGET):  $(OBJS)
            $(CC) $(OPTFLAGS) -o $@ $(OBJS) $(LFLAGS)
.c.o:
            $(CC) $(OPTFLAGS) -c  $*.c
clean:
            /bin/rm -f *.o $(TARGET) *~ *.mod
```



**HB M x N**



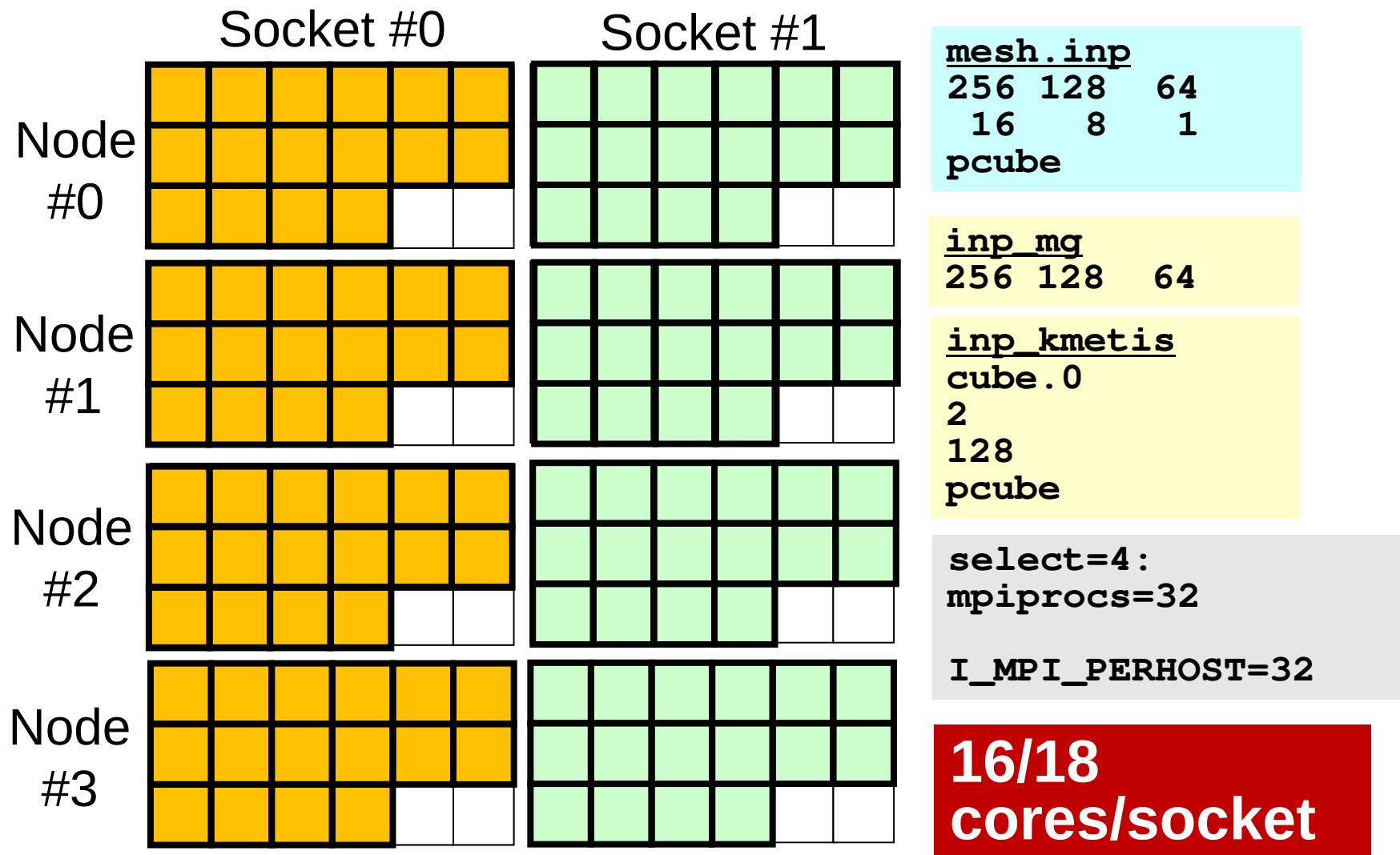
Number of OpenMP threads  
per a single MPI process



Number of MPI process  
per a single "socket"

# 4-nodes/8-sockets: 128 MPI process's

## Flat MPI, 32 MPI process's/Node



# Flat MPI: 16 MPI Processes/Socket

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=32    Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

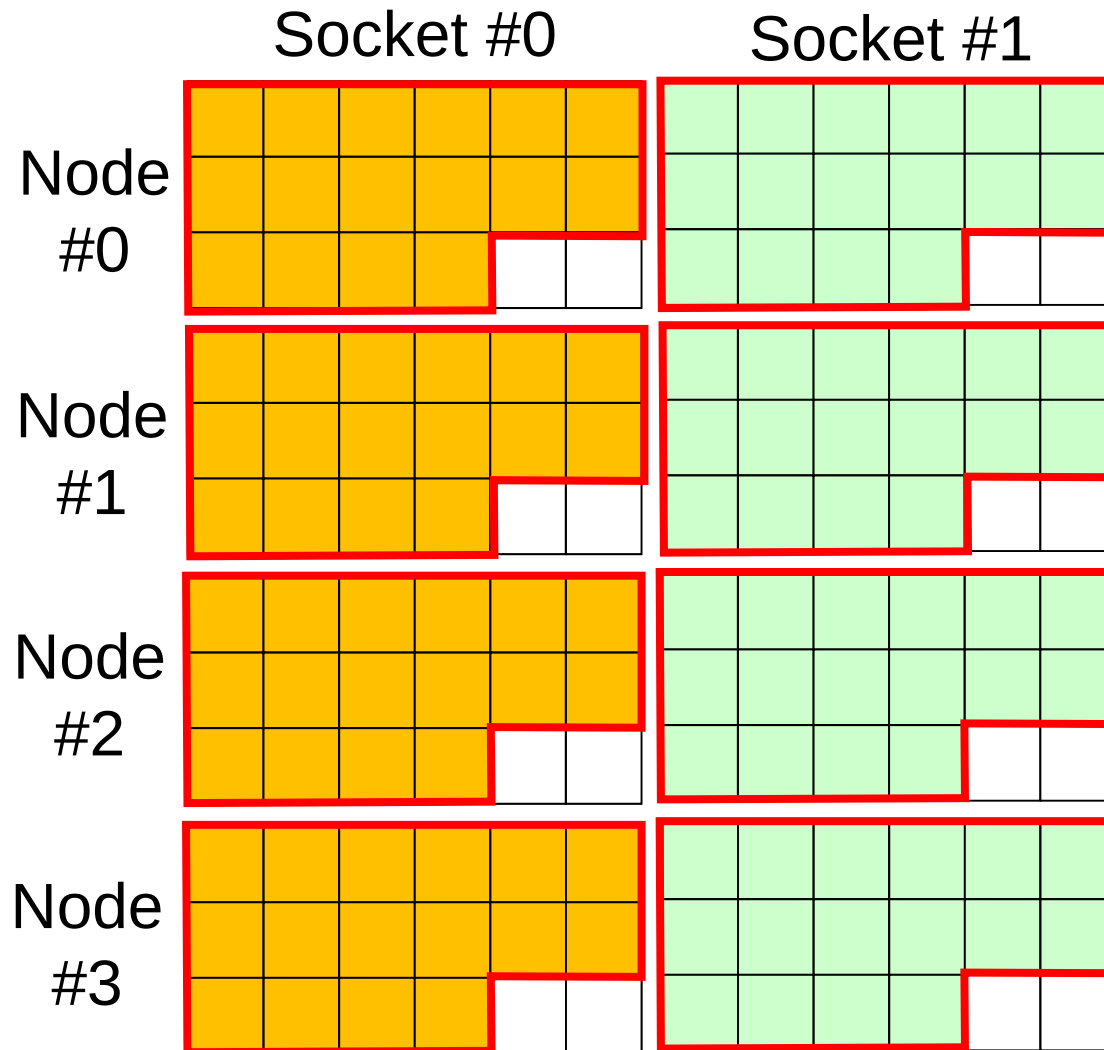
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=32          MPI Proc.#/Node

mpirun ./impimap.sh ./sol
```

# 4-nodes: 16-threads x 8 MPI process's

## HB 16x1, 2 MPI process's/Node



```
mesh.inp
256 128 64
  4  2  1
pcube
```

```
inp_mg
256 128 64
```

```
inp_kmetis
cube.0
2
8
pcube
```

```
select=4:
mpiprocs=2
```

```
I_MPI_PERHOST=2
OMP_NUM_THREADS=16
```

**16/18  
cores/socket**

# HB 16x1

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=2      Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

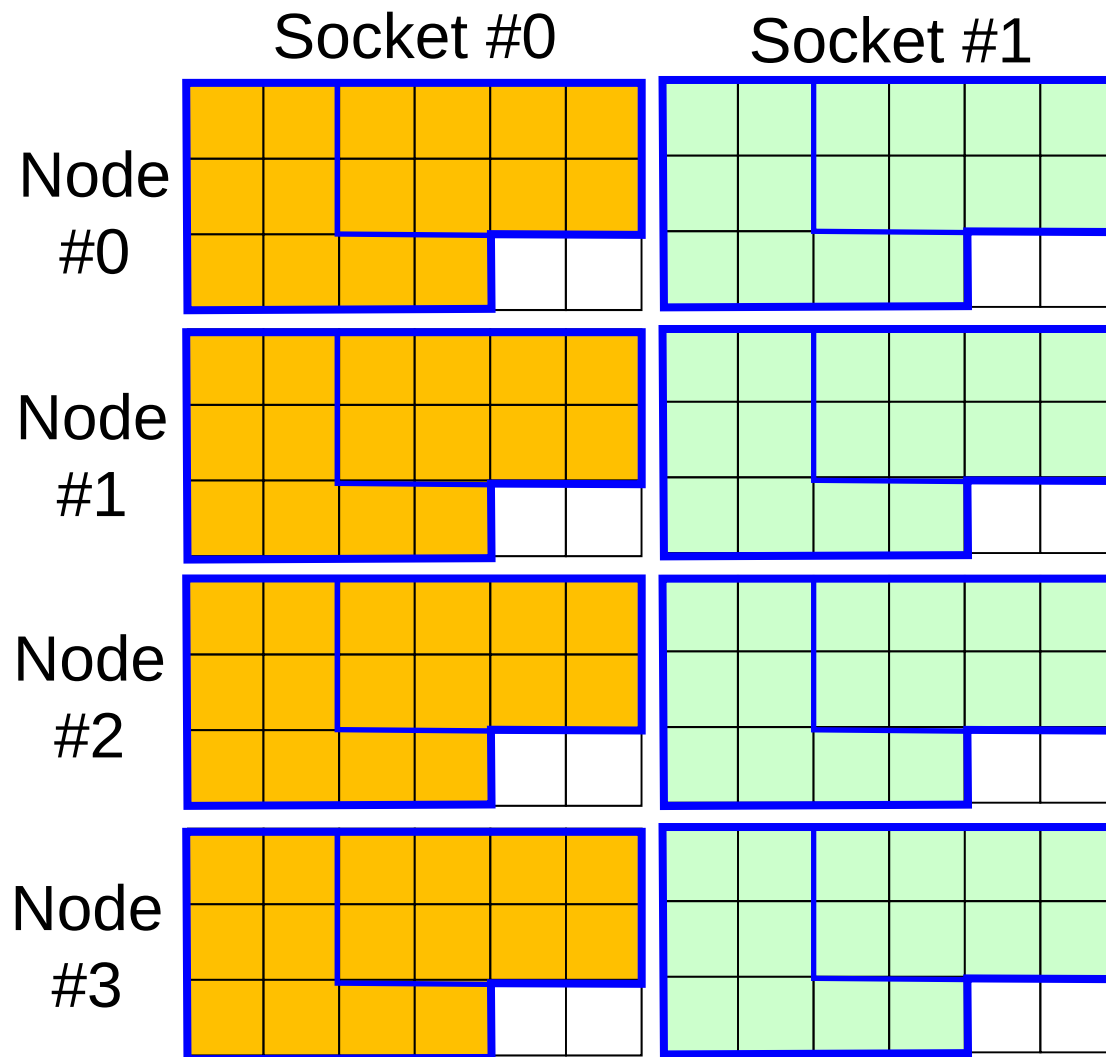
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=16      Thread#/MPI Process
export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=2        MPI Proc.#/Node

mpirun ./impimap.sh ./sol1
```

# 4-nodes: 8-threads x 16 MPI process's

## HB 8x2, 4 MPI process's/Node



```
mesh.inp
256 128 64
  4  4  1
pcube
```

```
inp_mg
256 128 64
```

```
inp_kmetis
cube.0
2
16
pcube
```

```
select=4:
mpiprocs=4
```

```
I_MPI_PERHOST=4
OMP_NUM_THREADS=8
```

**16/18  
cores/socket**

# HB 8x2

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=4      Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

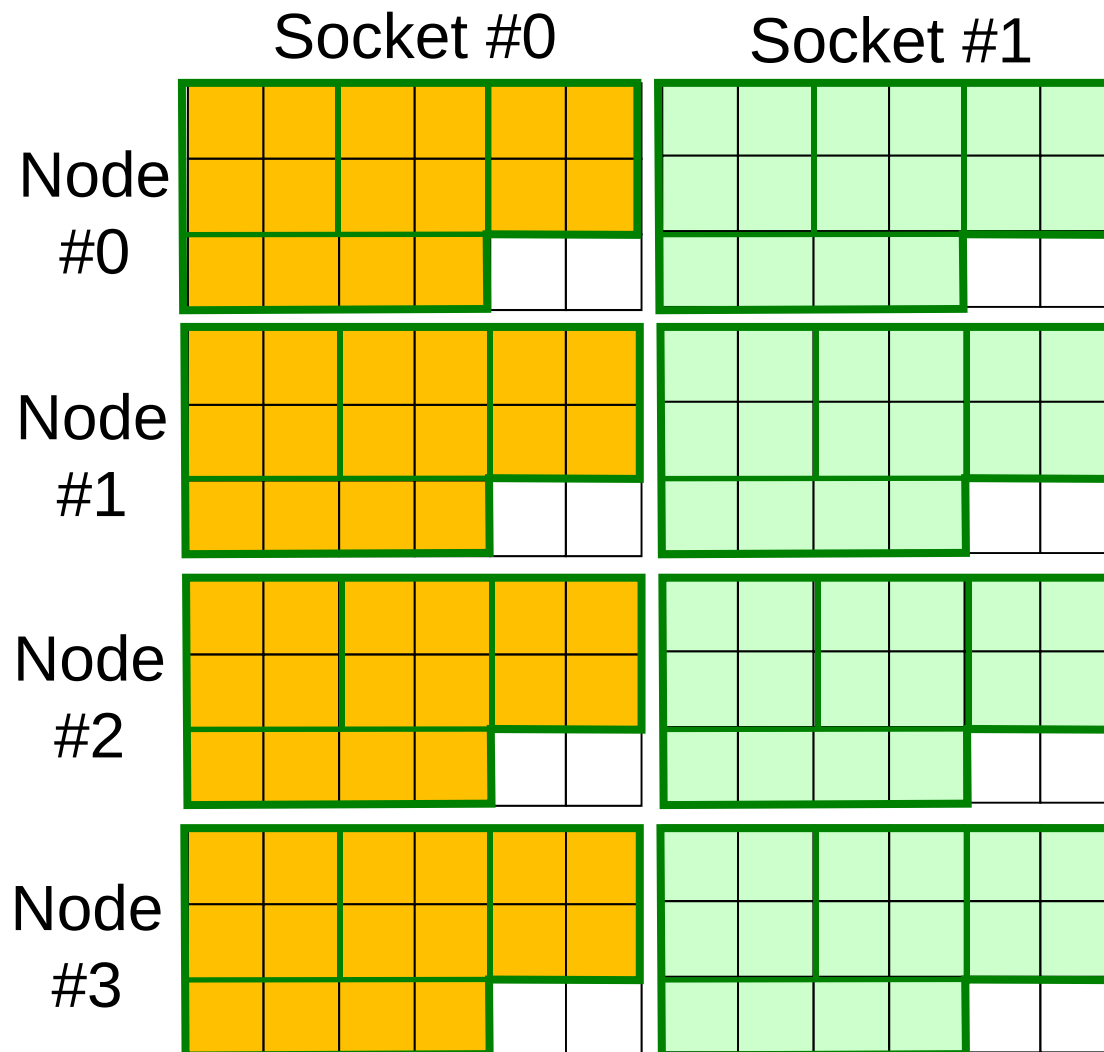
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=8          Thread#/MPI Process
export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=4           MPI Proc.#/Node

mpirun ./impimap.sh ./sol1
```

# 4-nodes: 4-threads x 32 MPI process's

## HB 4x4, 8 MPI process's/Node



```
mesh.inp
256 128 64
8 4 1
pcube
```

```
inp_mg
256 128 64
```

```
inp_kmetis
cube.0
2
32
pcube
```

```
select=4:
mpiprocs=8
```

```
I_MPI_PERHOST=8
OMP_NUM_THREADS=4
```

**16/18  
cores/socket**

# HB 4x4

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=8      Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

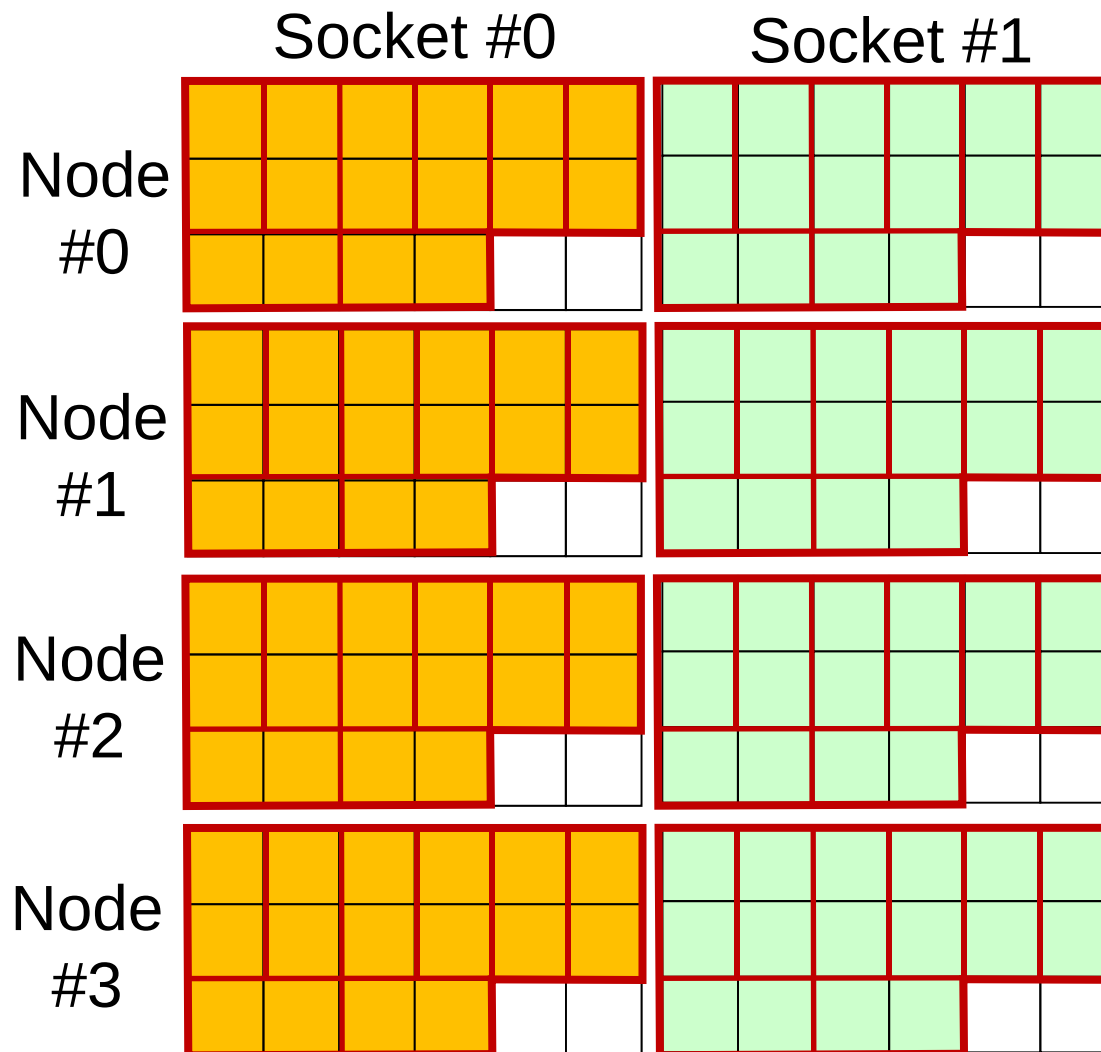
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=4          Thread#/MPI Process
export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=8           MPI Proc.#/Node

mpirun ./impimap.sh ./sol1
```

# 4-nodes: 2-threads x 64 MPI process's

## HB 2x8, 16 MPI process's/Node



```
mesh.inp
256 128 64
8 8 1
pcube
```

```
inp_mg
256 128 64
```

```
inp_kmetis
cube.0
2
64
pcube
```

```
select=4:
mpiprocs=16
```

```
I_MPI_PERHOST=16
OMP_NUM_THREADS=2
```

**16/18  
cores/socket**

# HB 2x8

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=16      Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

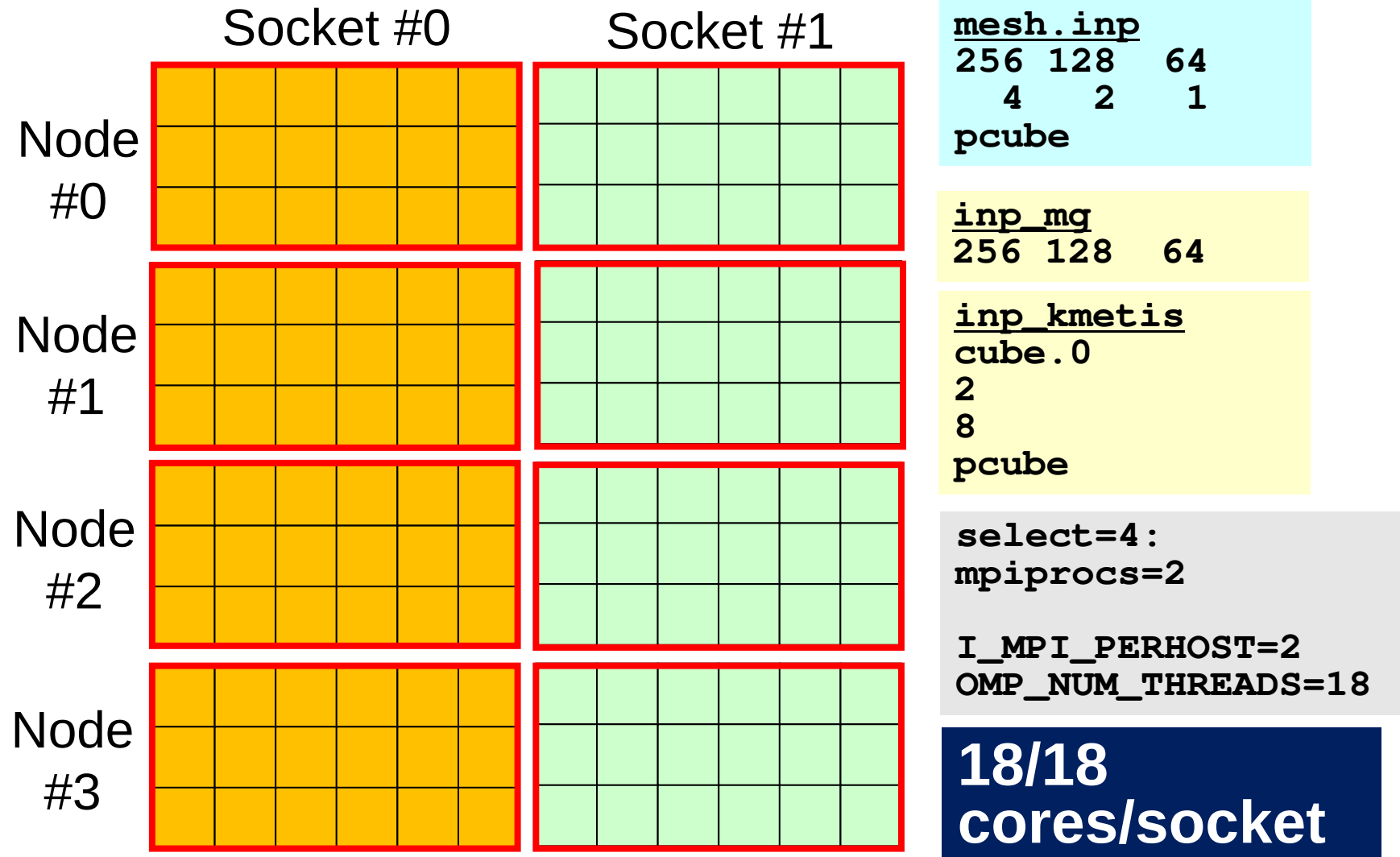
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=2           Thread#/MPI Process
export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=16           MPI Proc.#/Node

mpirun ./impimap.sh ./sol1
```

# 4-nodes: 18-threads x 8 MPI process's

## HB 18x1, 2 MPI process's/Node



# HB 18x1

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=2      Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

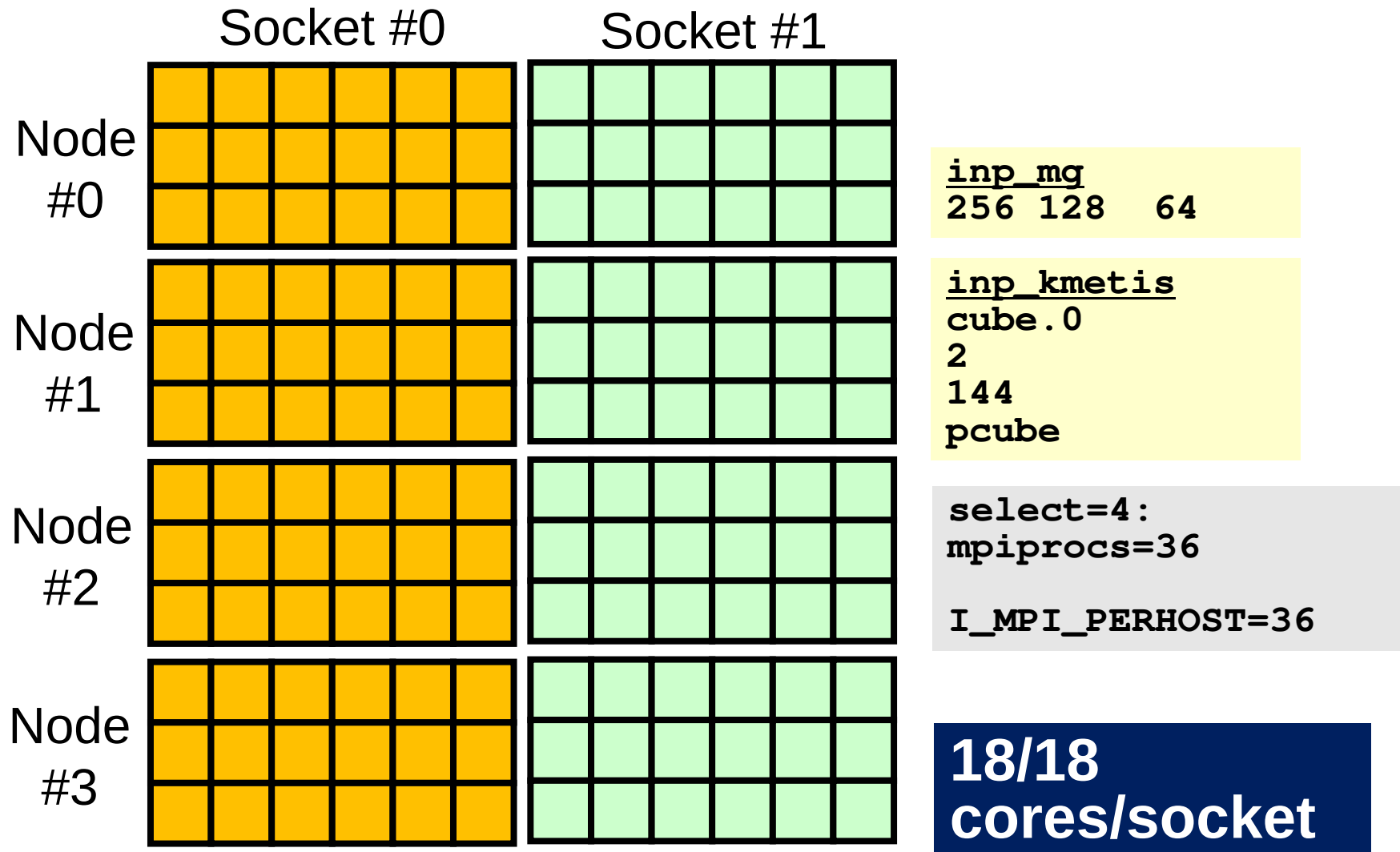
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=18      Thread#/MPI Process
export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=2        MPI Proc.#/Node

mpirun ./impimap.sh ./sol1
```

# 4-nodes/8-sockets: 144 MPI process's

## Flat MPI, 36 MPI process's/Node



# Flat MPI: 18 MPI Processes/Socket

```
#!/bin/sh
#PBS -q u-lecture
#PBS -N hybrid
#PBS -l select=4:mpiprocs=36    Node#, MPI Proc#/Node
#PBS -Wgroup_list=gt26
#PBS -l walltime=00:05:00
#PBS -e err
#PBS -o test.lst

cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

export I_MPI_PIN_DOMAIN=socket
export I_MPI_PERHOST=36          MPI Proc.#/Node

mpirun ./impimap.sh ./sol
```

# How to apply multi-threading

- CG Solver
  - Just insert OpenMP directives
  - ILU/IC preconditioning is much more difficult
- MAT\_ASS (mat\_ass\_main, mat\_ass\_bc)
  - Data Dependency
  - Avoid to accumulate contributions of multiple elements to a single node simultaneously (in parallel)
    - results may be changed
    - deadlock may occur
  - Coloring
    - Elements in a same color do not share a node
    - Parallel operations are possible for elements in each color
    - In this case, we need only 8 colors for 3D problems (4 colors for 2D problems)
    - Coloring part is very expensive: parallelization is difficult

# FORTRAN (solver\_CG)

```
!$omp parallel do private(i)
  do i= 1, N
    X(i) = X (i) + ALPHA * WW (i, P)
    WW (i, R) = WW (i, R) - ALPHA * WW (i, Q)
  enddo
```

```
DNRM20= 0. d0
!$omp parallel do private(i) reduction (+:DNRM20)
  do i= 1, N
    DNRM20= DNRM20 + WW (i, R)**2
  enddo
```

```
!$omp parallel do private(j, k, i, WVAL)
  do j= 1, N
    WVAL= D (j)*WW (j, P)
    do k= index(j-1)+1, index(j)
      i= item(k)
      WVAL= WVAL + AMAT (k)*WW (i, P)
    enddo
    WW (j, Q) = WVAL
  enddo
```

# C(solver\_CG)

```
#pragma omp parallel for private (i)
  for (i=0; i<N; i++) {
    X [i] += ALPHA *WW[P][i];
    WW[R][i] += -ALPHA *WW[Q][i];
  }
```

```
  DNRM20= 0. e0;
#pragma omp parallel for private (i) reduction (+:DNRM20)
  for (i=0; i<N; i++) {
    DNRM20+=WW[R][i]*WW[R][i];
  }
```

```
#pragma omp parallel for private (j, i, k, WVAL)
  for ( j=0; j<N; j++) {
    WVAL= D[j] * WW[P][j];
    for (k=indexLU[j]; k<indexLU[j+1]; k++) {
      i=itemLU[k];
      WVAL+= AMAT[k] * WW[P][i];
    }
    WW[Q][j]=WVAL;
```

# solver\_SR (send)

```

do neib= 1, NEIBPETOT
  istart= EXPORT_INDEX(neib-1)
  inum  = EXPORT_INDEX(neib ) - istart
!$omp parallel do private(k,ii)
  do k= istart+1, istart+inum
    ii  = EXPORT_ITEM(k)
    WS(k)= X(ii)
  enddo

  call MPI_Isend (WS(istart+1), inum, MPI_DOUBLE_PRECISION,      &
&                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(neib),  &
&                  ierr)
enddo

```

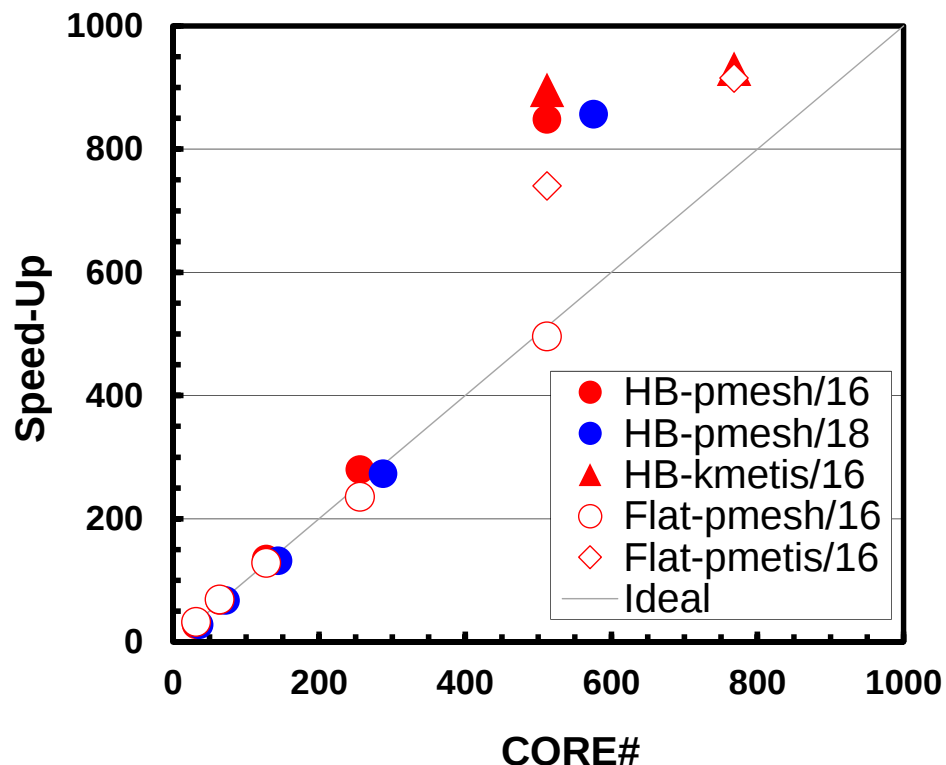
```

for( neib=1;neib<=NEIBPETOT;neib++) {
  istart=EXPORT_INDEX[neib-1];
  inum  =EXPORT_INDEX[neib]-istart;
#pragma omp parallel for private (k,ii)
  for( k=istart;k<istart+inum;k++) {
    ii= EXPORT_ITEM[k];
    WS[k]= X[ii-1];
  }
  MPI_Isend(&WS[istart], inum, MPI_DOUBLE,
            NEIBPE[neib-1], 0, MPI_COMM_WORLD, &req1[neib-1]);
}

```

# Example: Strong Scaling: Fortran

- $256 \times 128 \times 128$  nodes
    - 4,194,304 nodes, 4,112,895 elements
  - 32~864 cores, HB 16x1, HB 18x1, Flat MPI
  - Linear Solver
- Performance of Flat-pmesh/16 w/32 cores= 32.0**



```
256 128 128
  2   1   1
pcube
```

```
select=1:
mpiprocs=2
```

```
256 128 128
  2   1   2
pcube
```

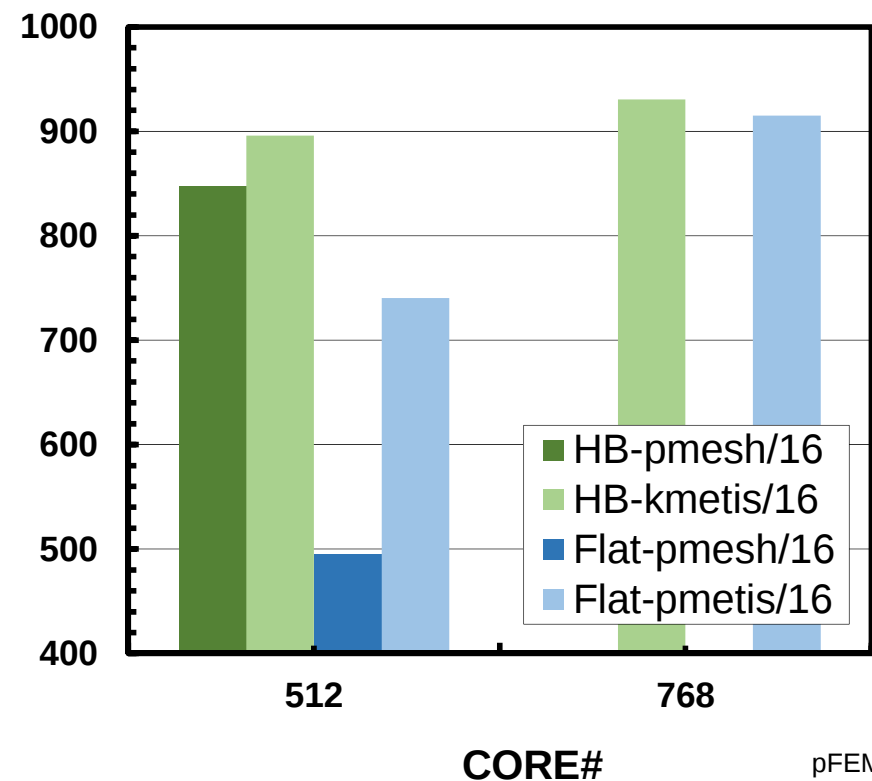
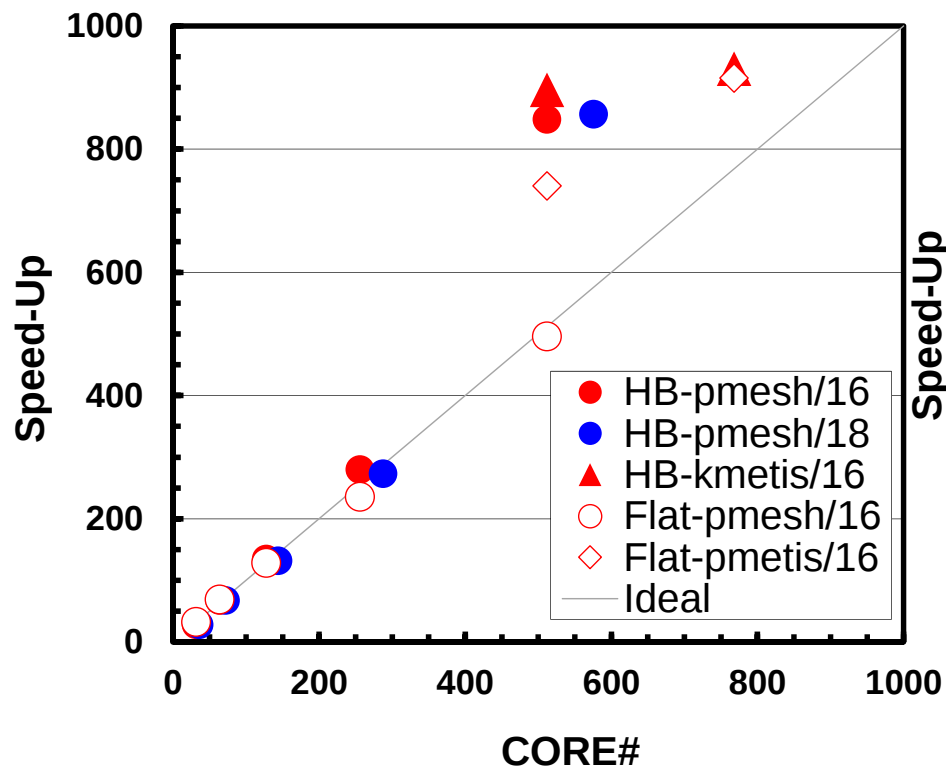
```
select=2:
mpiprocs=4
```

```
256 128 128
  4   2   2
pcube
```

```
select=8:
mpiprocs=16
```

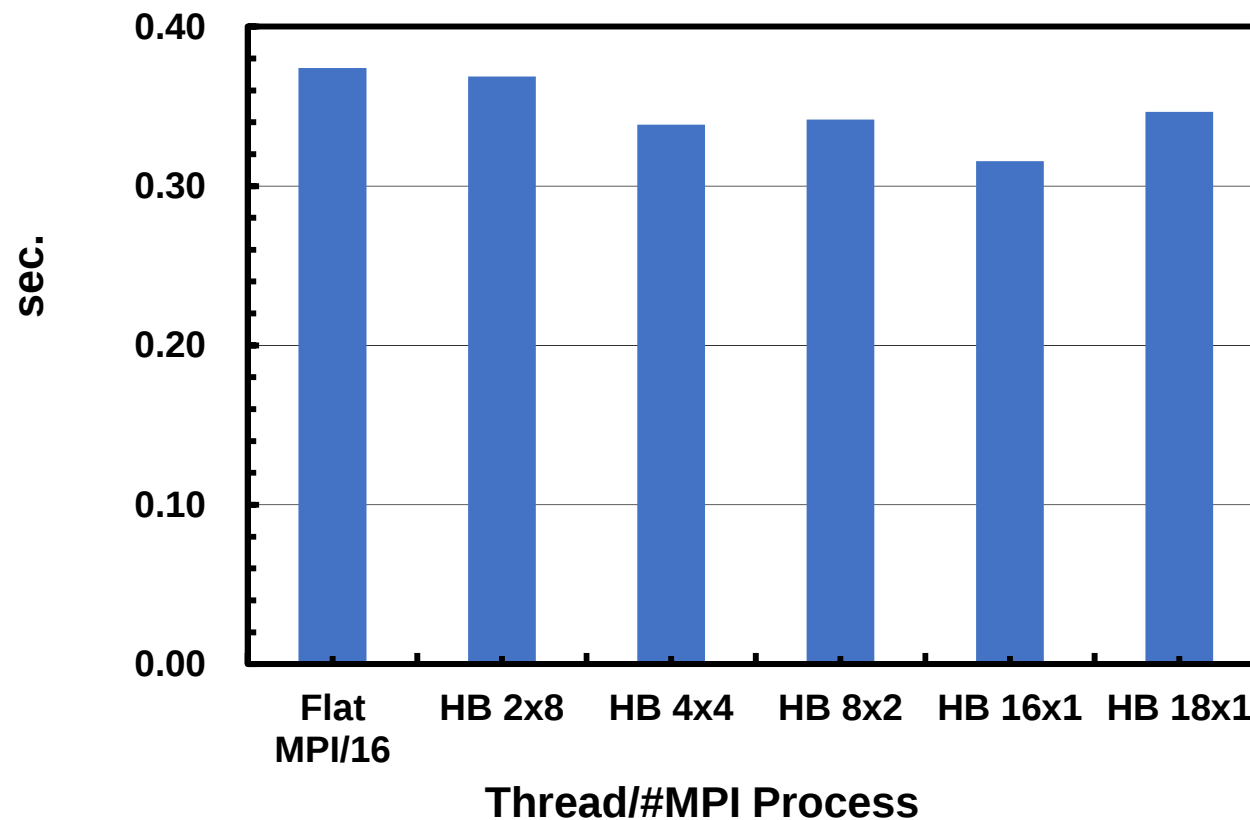
# Example: Strong Scaling: Fortran

- $256 \times 128 \times 128$  nodes
    - 4,194,304 nodes, 4,112,895 elements
  - 32~864 cores, HB 16x1, HB 18x1, Flat MPI
  - Linear Solver
- Performance of Flat-pmesh/16 w/32 cores= 32.0



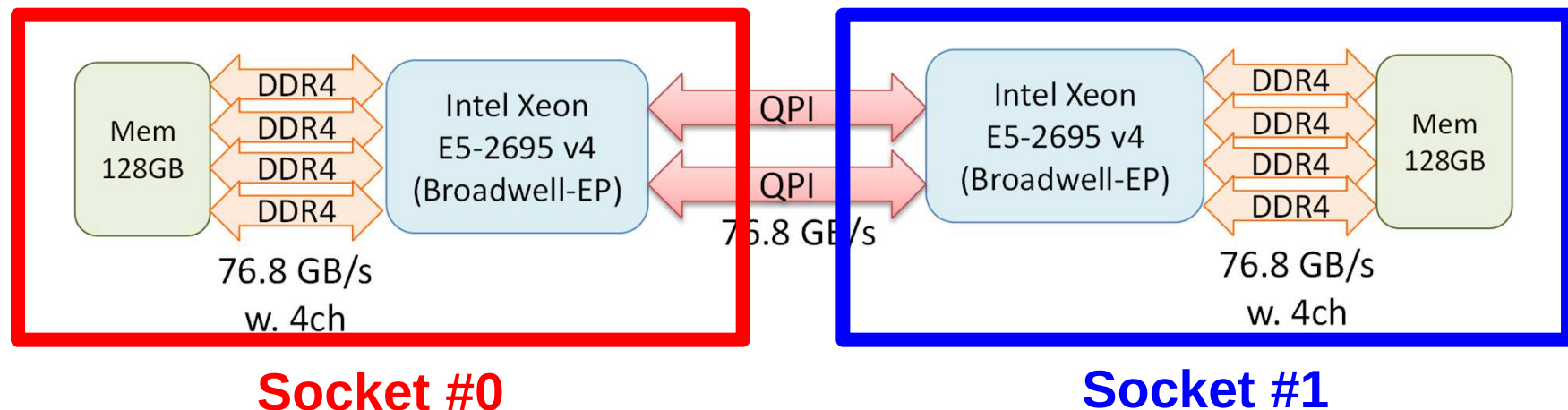
# Computation Time using 16 nodes

- kmetis
- Linear Solver



# Flat MPI vs. Hybrid

- Depends on applications, problem size, HW etc.
- Flat MPI is generally better for sparse linear solvers, if number of computing nodes is not so large.
  - Memory contention
- Hybrid becomes better, if number of computing nodes is larger.
  - Fewer number of MPI processes.
- 1 MPI Process/Node is possible: NUMA (A1/A2)

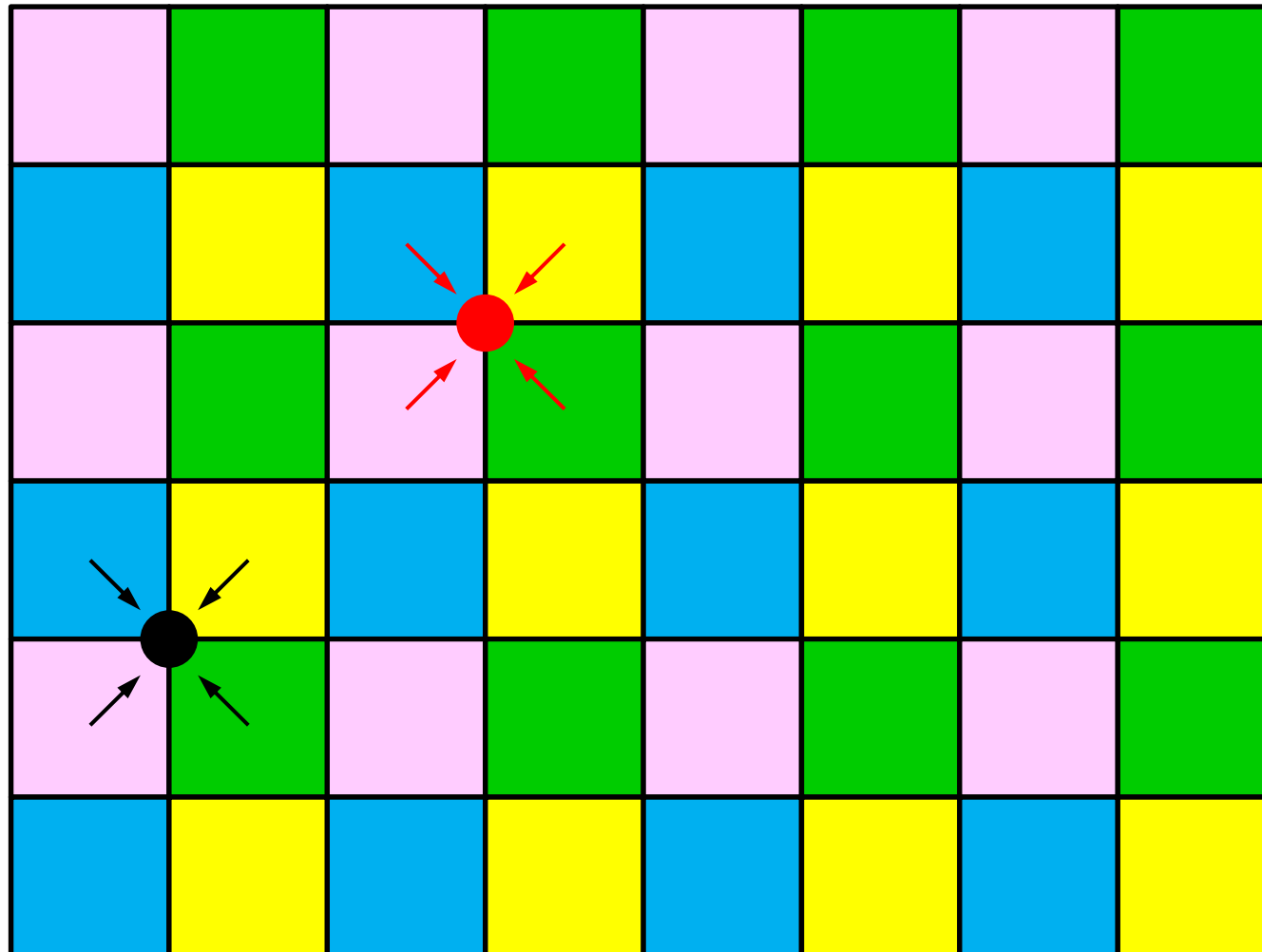


# How to apply multi-threading

- CG Solver
  - Just insert OpenMP directives
  - ILU/IC preconditioning is much more difficult
- MAT\_ASS (mat\_ass\_main, mat\_ass\_bc)
  - Data Dependency
  - Avoid to accumulate contributions of multiple elements to a single node simultaneously (in parallel)
    - results may be changed
    - deadlock may occur
  - Coloring
    - Elements in a same color do not share a node
    - Parallel operations are possible for elements in each color
    - In this case, we need only 8 colors for 3D problems (4 colors for 2D problems)
    - Coloring part is very expensive: parallelization is difficult

# Multi-Threading: Mat\_Ass

Parallel operations are possible for elements in same color (they are independent)



# Coloring (1/2)

```

allocate (ELMCOLORindex(0:NP))      Number of elements in each color
allocate (ELMCOLORitem (ICELTOT))   Element ID renumbered according to "color"
if (allocated (IWKX)) deallocate (IWKX)
allocate (IWKX(0:NP,3))

```

```

IWKX= 0
icou= 0
do icol= 1, NP
  do i= 1, NP
    IWKX(i,1)= 0
  enddo
  do icel= 1, ICELTOT
    if (IWKX(icel,2).eq.0) then
      in1= ICELNOD(icel,1)
      in2= ICELNOD(icel,2)
      in3= ICELNOD(icel,3)
      in4= ICELNOD(icel,4)
      in5= ICELNOD(icel,5)
      in6= ICELNOD(icel,6)
      in7= ICELNOD(icel,7)
      in8= ICELNOD(icel,8)

      ip1= IWKX(in1,1)
      ip2= IWKX(in2,1)
      ip3= IWKX(in3,1)
      ip4= IWKX(in4,1)
      ip5= IWKX(in5,1)
      ip6= IWKX(in6,1)
      ip7= IWKX(in7,1)
      ip8= IWKX(in8,1)
    end if
  enddo
enddo

```

# Coloring (2/2)

```

isum= ip1 + ip2 + ip3 + ip4 + ip5 + ip6 + ip7 + ip8
if (isum.eq.0) then      None of the nodes is accessed in same color
    icou= icou + 1
    IWKX(icol,3)= icou    (Current) number of elements in each color
    IWKX(icol,2)= icol
    ELMCOLORitem(icou)= icol  ID of icou-th element= icol

    IWKX(in1,1)= 1        These nodes on the same elements can not be
    IWKX(in2,1)= 1        accessed in same color
    IWKX(in3,1)= 1
    IWKX(in4,1)= 1
    IWKX(in5,1)= 1
    IWKX(in6,1)= 1
    IWKX(in7,1)= 1
    IWKX(in8,1)= 1
    if (icou.eq.ICELTOT) goto 100  until all elements are colored
endif
endif
enddo
enddo

100 continue
ELMCOLORtot= icol          Number of Colors
IWKX(0,3)= 0
IWKX(ELMCOLORtot,3)= ICELTOT

do icol= 0, ELMCOLORtot
    ELMCOLORindex(icol)= IWKX(icol,3)
enddo

```

# Multi-Threaded Matrix Assembling Procedure

```

do icol= 1, ELMCOLORtot
!$omp parallel do private (icel0, icel)
!$omp& private (in1, in2, in3, in4, in5, in6, in7, in8)
!$omp& private (nodLOCAL, ie, je, ip, jp, kk, iiS, iiE, k)
!$omp& private (DETJ, PNx, PNY, PNz, QVC, QV0, COEFij, coef, SHi)
!$omp& private (PNXi, PNYi, PNzi, PNxj, PNYj, PNzj, ipn, jpn, kpn)
!$omp& private (X1, X2, X3, X4, X5, X6, X7, X8)
!$omp& private (Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8)
!$omp& private (Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, CONDO)
do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
icel= ELMCOLORitem(icel0)
in1= ICELNOD(icel, 1)
in2= ICELNOD(icel, 2)
in3= ICELNOD(icel, 3)
in4= ICELNOD(icel, 4)
in5= ICELNOD(icel, 5)
in6= ICELNOD(icel, 6)
in7= ICELNOD(icel, 7)
in8= ICELNOD(icel, 8)

```

...