3D Parallel FEM (IV) (OpenMP + MPI) Hybrid Parallel Programming Model

Kengo Nakajima
Information Technology Center

Programming for Parallel Computing (616-2057)
Seminar on Advanced Computing (616-4009)
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
HB \(\text{M} \times \text{N}\)

- Number of OpenMP threads per a single MPI process
- Number of MPI processes per a single node
Size (and number) of local data changes according to parallel programming model

example: 6 nodes, 96 cores
Batch Script (1/2)
Env. Var.: OMP_NUM_THREADS

Flat MPI

#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=96"

mpiexec ./sol

rm wk.*

Hybrid 16 x 1

#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=6"

export OMP_NUM_THREADS=16
mpiexec ./sol

rm wk.*
Batch Script (2/2)
Env. Var.: OMP_NUM_THREADS

**Hybrid 4 × 4**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=24"

export OMP_NUM_THREADS=4
mpiexec ./sol

rm wk.*
```

**Hybrid 8 × 2**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=12"

export OMP_NUM_THREADS=8
mpiexec ./sol

rm wk.*
```
Background

• Multicore/Manycore Processors
  – Low power consumption, Various types of programming models

• OpenMP
  – Directive based, (seems to be) easy
  – Many books

• Data Dependency (no classes this year)
  – 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**
  - 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

Master

thread

thread

thread

thread

thread

PARALLEL fork

END PARALLEL join

Master

thread

thread

thread

thread

thread

PARALLEL fork

END PARALLEL join

Master

thread

thread

thread

thread

thread

PARALLEL fork

END PARALLEL join
Number of Threads

• **OMP_NUM_THREADS**
  – How to change?
    • bash (.bashrc)
    • csh (.cshrc)

```bash
export OMP_NUM_THREADS=8
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.)
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

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

Dot Products

```c
!$omp parallel do private(iS,iE,i)
!$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
```

DAXPY

```c
!$omp parallel do
do i = 1, NP
   Y(i) = ALPHA*X(i) + Y(i)
enddo
!$omp end parallel do
```
OpenMP/Directives
Matrix/Vector Products

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

```c
!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
endo
dodo
!omp end parallel do
```

W(:,;), R, Z, PE$\text{mpTOT}$ global (shared)
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)

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

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

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

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

```c
 #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

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

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

- “END PARALLEL DO” is not required
# Functions which can be used with OpenMP

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int omp_get_num_threads (void)</td>
<td>Total Thread #</td>
</tr>
<tr>
<td>int omp_get_thread_num (void)</td>
<td>Thread ID</td>
</tr>
<tr>
<td>double omp_get_wtime (void)</td>
<td>= MPI_Wtime</td>
</tr>
<tr>
<td>void omp_set_num_threads (int num_threads)</td>
<td>Setting Thread #</td>
</tr>
<tr>
<td>call omp_set_num_threads (num_threads)</td>
<td></td>
</tr>
</tbody>
</table>
OpenMP for Dot Products

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

```
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
```

Directives are just inserted.

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

```plaintext
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**

<table>
<thead>
<tr>
<th>INDEX(0)</th>
<th>INDEX(1)</th>
<th>INDEX(2)</th>
<th>INDEX(3)</th>
<th>INDEX(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>50</td>
<td>75</td>
<td>100</td>
</tr>
</tbody>
</table>

**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
endo
Matrix-Vector Multiply

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

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

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

```verbatim
!$omp parallel ...

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

!$omp do
  do i= 1, N
...
!$omp end parallel

#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 #
```makefile
# Makefile (Fortran, C)

F90 = mpifrtpx
F90LINKER = $(F90)
LIB_DIR =
INC_DIR =
OPTFLAGS = -Kfast,openmp
FFLAGS = $(OPTFLAGS)
FLIBS =
F90LFLAGS =
TARGET = ../run/sol1default:
OBJS = pfem_util.o pfem_finalize.o output_ucd.o

$(TARGET): $(OBJS)
  $(F90LINKER) $(OPTFLAGS)
  $(FLIBS)
  -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

CC = mpifccpx
LIB_DIR=
INC_DIR=
OPTFLAGS= -Kfast,openmp
LIBS =
LFLAGS =
TARGET = ../run/sol1default:
OBJS = test1.o ...
  util.o

$(TARGET): $(OBJS)
  $(CC) $(OPTFLAGS) -o $@
$(OBJEX) $(LFLAGS)
  .c.o:
    $(CC) $(OPTFLAGS) -c
    $(F90) $(FFLAGS)
  $(INC_DIR) -c *.*.f
  .f90.o:
    $(F90) $(FFLAGS)
  $(INC_DIR) -c *.*.f90
  .SUFFIXES: .f90 .f

clean:
  /bin/rm -f *.o $(TARGET)
  *~ *.mod
```
HB \ M \times N

Number of OpenMP threads per a single MPI process

Number of MPI process per a single node
Distributed Local Meshes for Flat MPI, HB 4x4, and HB 16x1
example: 6 nodes, 96 cores
go0.sh (1/2): OMP_NUM_THREADS

**Flat MPI (go.sh)**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school"
#PJM -o "test.lst"
#PJM --mpi "proc=96"

mpiexec ./sol

rm wk.*
```

**Hybrid 16 × 1**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school"
#PJM -o "test.lst"
#PJM --mpi "proc=96"

export OMP_NUM_THREADS=16
mpiexec ./sol1

rm wk.*
```
go0.sh (2/2): OMP_NUM_THREADS

**Hybrid 4 × 4**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=24"

export OMP_NUM_THREADS=4
mpiexec ./sol1

rm wk.*
```

**Hybrid 8 × 2**

```bash
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=12"

export OMP_NUM_THREADS=8
mpiexec ./sol1

rm wk.*
```
Example: HB 16 × 1

go0.sh

#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=lecture1"
#PJM -g "gt91"
#PJM -o "test.lst"
#PJM --mpi "proc=6"

export OMP_NUM_THREADS=16
mpiexec ./sol1

rm wk.*
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)

```fortran
!$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;
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]);
}
# Results (1/2)

512 × 384 × 256 = 50,331,648 nodes
12 nodes, 192 cores
64³=262,144 nodes/core

<table>
<thead>
<tr>
<th>ndx, ndy, ndz (#MPI proc.)</th>
<th>Iter’s</th>
<th>sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat MPI</td>
<td>1240</td>
<td>73.9</td>
</tr>
<tr>
<td>HB 1 × 16</td>
<td>1240</td>
<td>73.6</td>
</tr>
<tr>
<td>HB 2 × 8</td>
<td>1240</td>
<td>78.8</td>
</tr>
<tr>
<td>HB 4 × 4</td>
<td>1240</td>
<td>80.3</td>
</tr>
<tr>
<td>HB 8 × 2</td>
<td>1240</td>
<td>81.1</td>
</tr>
<tr>
<td>HB 16 × 1</td>
<td>1240</td>
<td>81.9</td>
</tr>
</tbody>
</table>

-Kopenmpでコンパイル、OMP_NUM_THREADS=1

512 384 256 ndx ndy ndz pcube
Results (2/2)

512 \times 384 \times 256 = 50,331,648 nodes
12 nodes, 192 cores
64^3 = 262,144 nodes/core

<table>
<thead>
<tr>
<th>OMP_NUM_THREADS</th>
<th>sec.</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1056.2</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>592.5</td>
<td>1.78</td>
</tr>
<tr>
<td>4</td>
<td>289.8</td>
<td>3.64</td>
</tr>
<tr>
<td>8</td>
<td>148.1</td>
<td>7.13</td>
</tr>
<tr>
<td>12</td>
<td>103.6</td>
<td>10.19</td>
</tr>
<tr>
<td>16</td>
<td>81.9</td>
<td>12.90</td>
</tr>
<tr>
<td>Flat MPI, 1 proc./node</td>
<td>1082.4</td>
<td>-</td>
</tr>
</tbody>
</table>
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.
• Flat MPI is not realistic for Intel Xeon Phi/MIC with 240 threads/node
  – MPI process requires certain amount of memory.
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)
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
ico= 0
do icol= 1, NP
  do i= 1, NP
    IWKX(i,1)= 0
  enddo
doi cel= 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)
Coloring (2/2)

\[ isum = ip1 + ip2 + ip3 + ip4 + ip5 + ip6 + ip7 + ip8 \]

if (isum .eq. 0) then
  None of the nodes is accessed in the same color
  icou = icou + 1
  IWKX(icol,3) = icou
  (Current) number of elements in each color
  IWKX(icol,2) = icol
  ELMCOLORitem(icou) = icel
  ID of icou-th element = icel

  IWKX(in1,1) = 1
  These nodes on the same elements can not be
  IWKX(in2,1) = 1
  accessed in the 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. ICETOT) 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) = ICETOT

do icol = 0, ELMCOLORtot
  ELMCOLORindex(icol) = IWKX(icol,3)
enddo
Multi-Threaded Matrix Assembling Procedure

```fortran
    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, QVO, 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)
            ...
```