Introduction to Parallel Programming for Multicore/Manycore Clusters

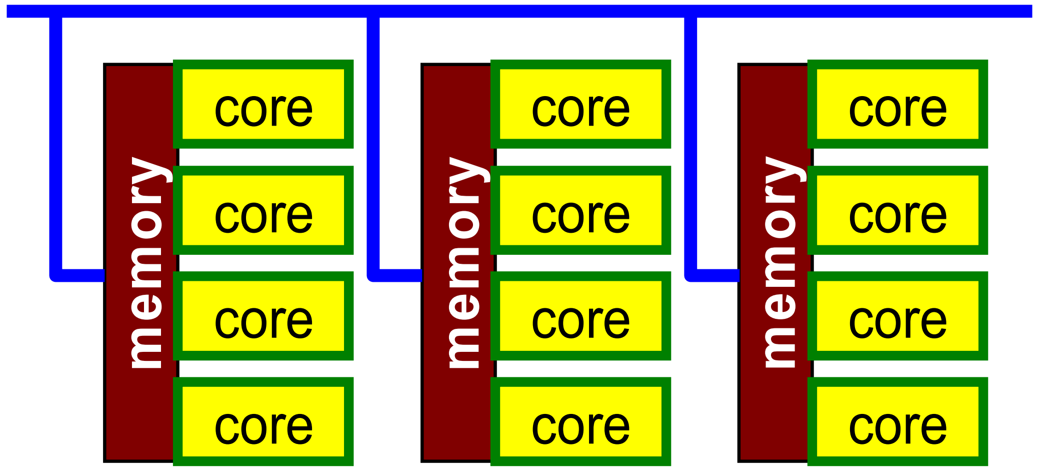
Part II-2: Parallel FVM using OpenMP

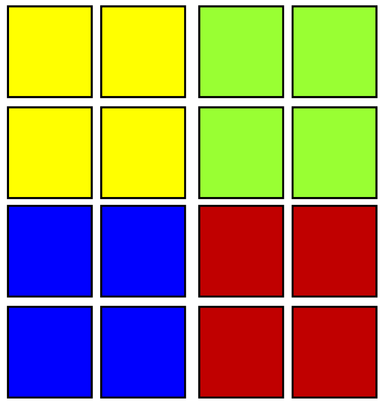
Kengo Nakajima
Information Technology Center
The University of Tokyo

- OpenMP
- Parallel Version of the Code by OpenMP
- STREAM

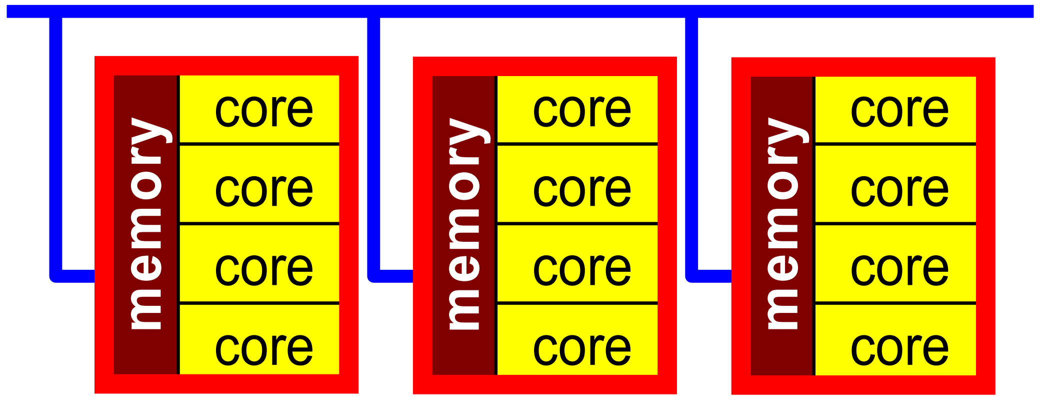
Flat MPI vs. Hybrid

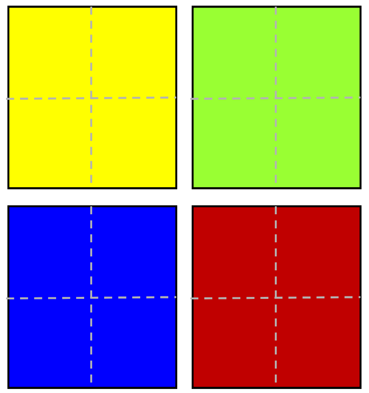
Flat-MPI：Each Core -> Independent

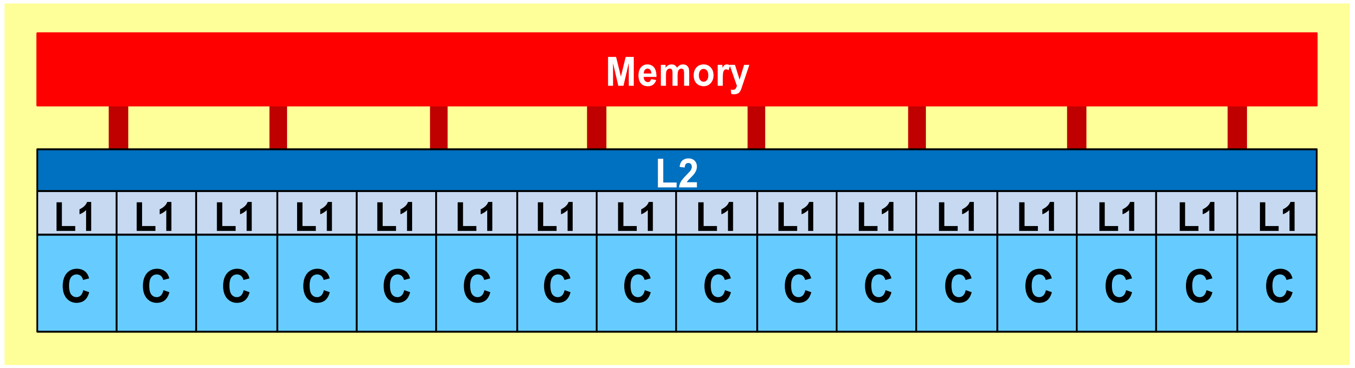




Hybrid：Hierarchal Structure







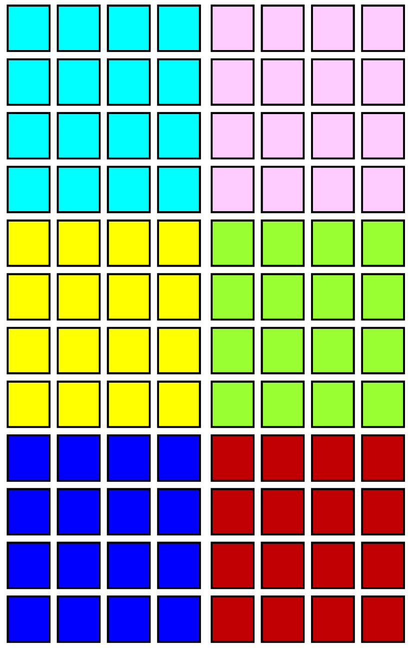
HB M x N

Number of OpenMP threads per a single MPI process

Number of MPI process per a single node

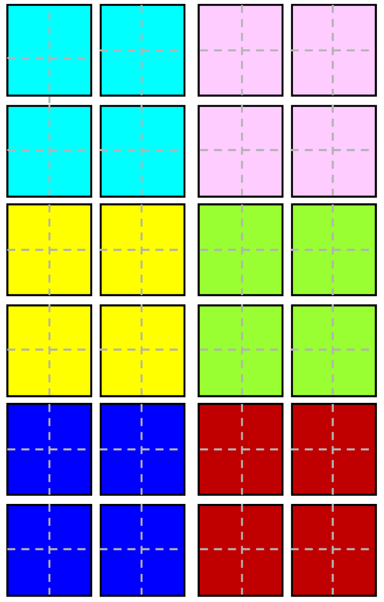
Size of data for each MPI process varies according to HB MxN

example: 6 nodes, 96 cores



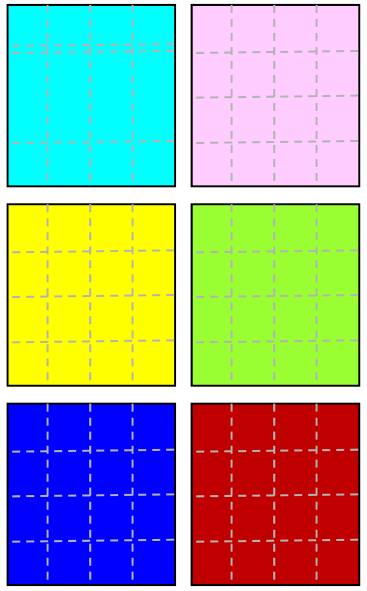
Flat MPI

```
128 192 64
  8  12  1
pcube
```



HB 4x4

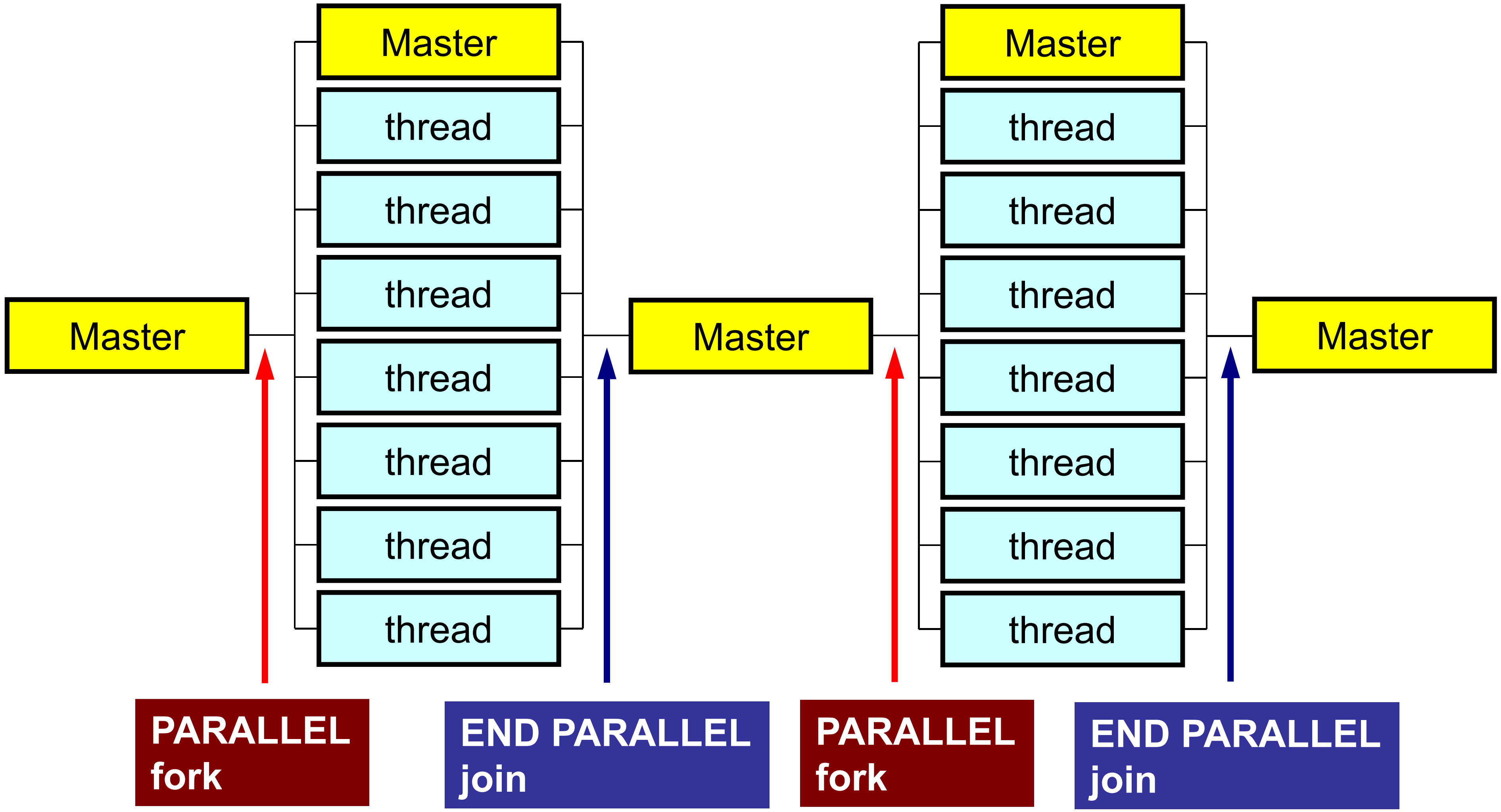
```
128 192 64
  4   6  1
pcube
```



HB 16x1

```
128 192 64
  2   3  1
pcube
```

Fork-Join Parallel Execution Model



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

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

DAXPY

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

Dot Products

```
!$omp parallel do private(i)
!$omp&               reduction(+:RHO)
      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo
!$omp end parallel do
```

OpenMP/Direceives Matrix/Vector Products

```
!$omp parallel do private(i,j)
      do i= 1, N
        W(i,Q)= D(i)*W(i,P)
        do j= indexLU(i-1)+1, indexLU(i)
          W(i,Q)= W(i,Q) + AMAT(j)*W(itemLU(j),P)
        enddo
      enddo
!$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)

Features of OpenMP (cont.)

- “for” loops with “#pragma omp parallel for”
- Global (Shared) Variables, Private Variables
  - Default: Global (Shared)
  - Dot Products: reduction

```
!$omp parallel do private(i)
!$omp&            reduction(+:RHO)
      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo
!$omp end parallel do
```

W(:,:), R, Z
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];
}
```

OpenMP Directives (Fortran)

```
sentinel directive_name [clause[[,] clause]…]
```

- NO distinctions between upper and lower cases.
- sentinel
  - Fortran: !OMP, COMP, *$OMP
    - !$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 in OpenMP

functions	description
int omp_get_num_threads (void)	Thread #
int omp_get_thread_num (void)	Thread ID
double omp_get_wtime (void)	Timer
void omp_set_num_threads (int num_threads) call omp_set_num_threads (num_threads)	Specifying 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
```

Matrix-Vector Multiply

```
do i = 1, N
  VAL= D(i)*W(i,P)
  do k= indexLU(i-1)+1, indexLU(i)
    VAL= VAL + AMAT(k)*W(itemLU(k),P)
  enddo
  W(i,Q)= VAL
enddo
```

Matrix-Vector Multiply

```
!$omp parallel do private(ip,i,VAL,k)
      do ip= 1, PEsmpTOT
        do i = INDEX(ip-1)+1, INDEX(ip)
          VAL= D(i)*W(i,P)
          do k= indexLU(i-1)+1, indexLU(i)
            VAL= VAL + AMAT(k)*W(itemLU(k),P)
          enddo
          W(i,Q)= VAL
        enddo
      enddo
!$omp end parallel do
```

Matrix-Vector Multiply: Other Approach

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

```
!$omp parallel do private(i,VAL,k)
  do i = 1, N
    VAL= D(i)*W(i,P)
    do k= indexLU(i-1)+1, indexLU(i)
      VAL= VAL + AMAT(k)*W(itemLU(k),P)
    enddo
    W(i,Q)= VAL
  enddo
!$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 {
```

- OpenMP
- **Parallel Version of the Code by OpenMP**
- STREAM

Target for Parallelization

- FVM code
- Preconditioned CG solver: PCG
  - Diagonal Scaling, Point Jacobi
- NO sample code.
- Please develop the parallel code by yourself

Preconditioned Conjugate Gradient Method（PCG）

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)} z^{(i-1)}$
    if i=1
      $p^{(1)}= z^{(0)}$
     else
      $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
      $p^{(i)}= p^{(i-1)} + \beta_{i-1} z^{(i-1)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence |r|
end
```

Solving the following equation:

$$\{z\}=[M]^{-1}\{r\}$$

"Approximate Inverse Matrix"

$$[M]^{-1}\approx[A]^{-1}, \quad [M]\approx[A]$$

Ultimate Preconditioning:
Inverse Matrix

$$[M]^{-1}=[A]^{-1}, \quad [M]=[A]$$

Diagonal Scaling: Simple but weak

$$[M]^{-1}=[D]^{-1}, \quad [M]=[D]$$

Diagonal Scaling, Point-Jacobi

$$[M]=\begin{bmatrix} D_1 & 0 & \dots & 0 & 0 \\ 0 & D_2 & & 0 & 0 \\ \dots & & \dots & & \dots \\ 0 & 0 & & D_{N-1} & 0 \\ 0 & 0 & \dots & 0 & D_N \end{bmatrix}$$

- `solve [M]z`$^{(i-1)}$`= r`$^{(i-1)}$ is very easy.
- Provides fast convergence for simple problems.

Files on Oakleaf-FX (1/2)

```
>$ cd

>$ cp /home/z30088/omp/omp2-c.tar .
>$ cp /home/z30088/omp/omp2-f.tar .

>$ tar xvf omp2-c.tar
>$ tar xvf omp2-f.tar

>$ cd multicore
```

<u>Confirm Directories:</u>

```
omp2  stream

<$O-omp2>, <$O-stream>
```

Files on Oakleaf-FX (2/2)

```
>$ cd <$O-omp2>
>$ cd src

>$ make

>$ cd ../run
>$ pjsub go.sh
```

<$O-omp2>/src/Makefile

NOT for parallel computing

```
F90          = frtpx                : Compiler
F90OPTFLAGS= -Kfast                 : Optimization
F90FLAGS =$(F90OPTFLAGS)

.SUFFIXES:
.SUFFIXES: .o .f .f90 .c
#
.f90.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
.f.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
#
OBJS = ¥
solver_PCG.o rcm.o struct.o pcg.o ¥
boundary_cell.o cell_metrics.o ¥
input.o main.o poi_gen.o pointer_init.o outucd.o

TARGET = ../run/sol                 : Exec File

all: $(TARGET)

$(TARGET): $(OBJS)
        $(F90) $(F90FLAGS) -o $(TARGET) ¥
        $(OBJS) ¥
        $(F90FLAGS)
clean:
        rm -f *.o $(TARGET) *.mod *~ PI*
```

Running Job

- Batch Jobs
  - Only batch jobs are allowed.
  - Interactive executions of jobs are not allowed.
- How to run
  - writing job script
  - submitting job
  - checking job status
  - checking results
- Utilization of computational resources
  - 1-node (16 cores) is occupied by each job.
  - Your node is not shared by other jobs.

Job Script

- **`<$O-omp>/run/go.sh`**
- Scheduling + Shell Script

```
#!/bin/sh
#PJM -L “node=1“                 Number of Nodes
#PJM -L “elapse=00:10:00“        Computation Time
#PJM -L “rscgrp=lecture7“        Name of “QUEUE”
#PJM -g “gt17“                   Group Name (Wallet)
#PJM -j
#PJM -o “test.lst“               Standard Output

./sol                            Execs
```

Available QUEUE's

- Following 2 queues are available.
- 1 Tofu (12 nodes) can be used
  - `lecture`
    - 12 nodes (192 cores), 15 min., **valid until March 30 08:30**
    - Shared by all "educational" users
  - `lecture7`
    - 12 nodes (192 cores), 15 min., active during class time (09:00-17:00)
    - **More jobs (compared to `lecture`) can be processed up on availability.**
    - **Just during until Feb.23 17:00**
- **Please use "`lecture`" after Feb.24 !!**

Submitting & Checking Jobs

Submitting Jobs	`pjsub SCRIPT NAME`
Checking status of jobs	`pjstat`
Deleting/aborting	`pjdel JOB ID`
Checking status of queues	`pjstat --rsc`
Detailed info. of queues	`pjstat --rsc –x`
Number of running jobs	`pjstat --rsc –b`
Limitation of submission	`pjstat --limit`

```
[z30088@oakleaf-fx-6 S2-ref]$ pjstat

Oakleaf-FX scheduled stop time: 2012/09/28(Fri) 09:00:00 (Remain: 31days 20:01:46)

JOB_ID     JOB_NAME   STATUS  PROJECT  RSCGROUP  START_DATE        ELAPSE         TOKEN NODE:COORD
334730     go.sh      RUNNING gt61     lecture   08/27 12:58:08    00:00:05         0.0 1
```

solve_PCG (1/3)

```
      do i= 1, N
        X(i)   = 0.d0
        W(i,2)= 0.0D0
        W(i,3)= 0.0D0
        W(i,DD)= 1.d0/D(i)
      enddo
...
      ITR= N

      do L= 1, ITR
!C
!C +----------------+
!C | {z}= [Minv]{r} |
!C +----------------+
!C===
      do i= 1, N
        W(i,Z)= W(i,R)*W(i,DD)
      enddo
!C===

!C
!C +-------------+
!C | RHO= {r}{z} |
!C +-------------+
!C===
      RHO= 0.d0
      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo
!C===
```

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
    if i=1
      $p^{(1)}= z^{(0)}$
     else
      $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
      $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i\ = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence $|r|$
end
```

solve_PCG (2/3)

```
!C
!C +-------------------------------+
!C | {p} = {z} if      ITER=1     |
!C | BETA= RHO / RHO1  otherwise  |
!C +-------------------------------+
!C===
      if ( L.eq.1 ) then
        do i= 1, N
          W(i,P)= W(i,Z)
        enddo
       else
         BETA= RHO / RHO1
         do i= 1, N
           W(i,P)= W(i,Z) + BETA*W(i,P)
         enddo
      endif
!C===

!C
!C +-------------+
!C | {q}= [A]{p} |
!C +-------------+
!C===
      do i= 1, N
        VAL= D(i)*W(i,P)
        do k= indexLU(i-1)+1, indexLU(i)
          VAL= VAL + AMAY(k)*W(itemLU(k),P)
        enddo
        W(i,Q)= VAL
      enddo
!C===
```

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
     solve $[M]z^{(i-1)}= r^{(i-1)}$
     $\rho_{i-1}= r^{(i-1)} z^{(i-1)}$
     if i=1
        $p^{(1)}= z^{(0)}$
      else
        $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
        $p^{(i)}= z^{(i-1)} + \beta_{i-1} p^{(i)}$
     endif
     $q^{(i)}= [A]p^{(i)}$
     $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
     $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
     $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
     check convergence |r|
end
```

solve_PCG (3/3)

```
!C
!C +-----------------------+
!C | ALPHA= RHO / {p}{q}   |
!C +-----------------------+
!C===
      C1= 0.d0
      do i= 1, N
        C1= C1 + W(i,P)*W(i,Q)
      enddo
      ALPHA= RHO / C1
!C===
!C +------------------------+
!C | {x}= {x} + ALPHA*{p}  |
!C | {r}= {r} - ALPHA*{q}  |
!C +------------------------+
!C===
      do i= 1, N
        X(i)   = X(i)    + ALPHA * W(i,P)
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo
      DNRM2= 0.d0
      do i= 1, N
        DNRM2= DNRM2 + W(i,R)**2
      enddo
!C===
      ERR = dsqrt(DNRM2/BNRM2)
      if (ERR .lt. EPS) then
        IER = 0
        goto 900
       else
        RHO1 = RHO
      endif

      enddo
      IER = 1
```

r= b-[A]x
DNRM2=|r|^2
BNRM2=|b|^2

ERR= |r|/|b|

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)} z^{(i-1)}$
    if i=1
      $p^{(1)}= z^{(0)}$
     else
      $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
      $p^{(i)}= z^{(i-1)} + \beta_{i-1} p^{(i)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence |r|
end
```

Parallelization by OpenMP

- Focusing on “solver_PCG.c” (solve_PCG)
- Just insert OpenMP directives

```
>$ cd <$O-omp2>
>$ cd ex
(modify files)

>$ make
>$ cd ../run
>$ pjsub g.sh
```

```
#!/bin/sh
#PJM -L "node=1"
#PJM -L "elapse=00:10:00"
#PJM -L "rscgrp=lecture7"
#PJM -g "gt17"
#PJM -j
#PJM -o "100_08_002.lst"

export OMP_NUM_THREADS=8      1-16
./sol0
```

<$O-omp2>/ex/Makefile

parallel computing by OpenMP

```
F90          = frtpx              : Compiler
F90OPTFLAGS= -Kfast,openmp : Optimization + OpenMP
F90FLAGS =$(F90OPTFLAGS)

.SUFFIXES:
.SUFFIXES: .o .f .f90 .c
#
.f90.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
.f.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
#
OBJS = ¥
solver_PCG.o rcm.o struct.o pcg.o ¥
boundary_cell.o cell_metrics.o ¥
input.o main.o poi_gen.o pointer_init.o outucd.o

TARGET = ../run/sol0          : Exec File

all: $(TARGET)

$(TARGET): $(OBJS)
        $(F90) $(F90FLAGS) -o $(TARGET) ¥
        $(OBJS) ¥
        $(F90FLAGS)
clean:
        rm -f *.o $(TARGET) *.mod *~ PI*
```

solve_PCG (1/5)

parallel computing by OpenMP

```
 module solver_PCG
 contains

 subroutine solve_PCG                                                  &
&         ( N, NPLU, indexLU, itemLU, D, B, X,                         &
&            AMAT, EPS, ITR, IER, N2)

 use omp_lib
 implicit REAL*8 (A-H,O-Z)
 integer :: N, NL, NU, N2

 real(kind=8), dimension(N)   :: D, B, X

 real(kind=8), dimension(NPL) :: AMAT

 integer, dimension(0:N) :: indexLU
 integer, dimension(NPLU):: itemLU

 real(kind=8), dimension(:,:), allocatable :: W

 integer, parameter ::  R= 1
 integer, parameter ::  Z= 2
 integer, parameter ::  Q= 2
 integer, parameter ::  P= 3
 integer, parameter :: DD= 4
```

solve_PCG (2/5)

```
!$omp parallel do private(i)
      do i= 1, N
        X(i)   = 0.d0
        W(i,2)= 0.0D0
        W(i,3)= 0.0D0
        W(i,DD)= 1.d0/D(i)
      enddo

!$omp parallel do private(i,VAL,j)
      do i= 1, N
        VAL= D(i)*X(i)
        do k= indexLU(i-1)+1, indexLU(i)
          VAL= VAL + AMAT(k)*X(itemLU(k))
        enddo
        W(i,R)= B(i) - VAL
      enddo

      BNRM2= 0.0D0
!$omp parallel do private(i) reduction(+:BNRM2)
      do i= 1, N
        BNRM2 = BNRM2 + B(i)  **2
      enddo
```

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)} z^{(i-1)}$
    if i=1
        $p^{(1)}= z^{(0)}$
      else
        $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
        $p^{(i)}= z^{(i-1)} + \beta_{i-1} p^{(i)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence |r|
end
```

solve_PCG (3/5)

```
      ITR= N
      Stime= omp_get_wtime()

      do L= 1, ITR

!$omp parallel do private(i)
      do i= 1, N
        W(i,Z)= W(i,R)*W(i,DD)
      enddo

      RHO= 0.d0
!$omp parallel do private(i) reduction(+:RHO)
      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo

      if ( L.eq.1 ) then
!$omp parallel do private(i)
        do i= 1, N
          W(i,P)= W(i,Z)
        enddo
       else
         BETA= RHO / RHO1
!$omp parallel do private(i)
         do i= 1, N
           W(i,P)= W(i,Z) + BETA*W(i,P)
         enddo
      endif
```

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
    if i=1
      $p^{(1)}= z^{(0)}$
     else
      $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
      $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i\ = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence |r|
end
```

solve_PCG (4/5)

```
!$omp parallel do private(i,VAL,j)
      do i= 1, N
        VAL= D(i)*W(i,P)
        do k= indexLU(i-1)+1, indexLU(i)
          VAL= VAL + AMAT(k)*W(itemLU(k),P)
        enddo
        W(i,Q)= VAL
      enddo

      C1= 0.d0
!$omp parallel do private(i) reduction(+:C1)
      do i= 1, N
        C1= C1 + W(i,P)*W(i,Q)
      enddo

      ALPHA= RHO / C1

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

      DNRM2= 0.d0
!$omp parallel do private(ip,i) reduction(+:DNRM2)
      do i= 1, N
        DNRM2= DNRM2 + W(i,R)**2
      enddo

      ERR = dsqrt(DNRM2/BNRM2)...
```

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
     solve $[M]z^{(i-1)}= r^{(i-1)}$
     $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
     if i=1
       $p^{(1)}= z^{(0)}$
      else
       $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
       $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i)}$
     endif
     $q^{(i)}= [A]p^{(i)}$
     $\alpha_i\ = \rho_{i-1}/p^{(i)}q^{(i)}$
     $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
     $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
     check convergence |r|
end
```

solve_PCG (5/5)

```
      Stime = omp_get_wtime()

       do L= 1, ITR
...
          if (ERR .lt. EPS) then
            IER = 0
            goto 900
           else
            RHO1 = RHO
          endif

       enddo
       IER = 1
  900  continue
       Etime= omp_get_wtime()

       write (*,'(i5,2(1pe16.6))') L, ERR
       write (*,'(1pe16.6, a)') Etime-Stime, ' sec. (solver)'

       ITR= L
       deallocate (W)

       return
       end
```

Elapsed Time= Etime - Stime

Etime-Stime

NX=NY=NZ=100, 10^6 DOF

```
#!/bin/sh
#PJM -L "node=1"
#PJM -L "elapse=00:10:00"
#PJM -L "rscgrp=lecture7"
#PJM -g "gt17"
#PJM -j
#PJM -o "100_08_002.lst"

export OMP_NUM_THREADS=M      M=1,4 8,16
./sol
```

M	sec.	Speed-Up
1	33.7	1.00
4	8.73	3.86
8	4.95	6.80
16	3.30	10.20

Exercises

- Develop your own program by inserting OpenMP directives !

- Effect of problem size (NX, NY, NZ)
- Effect of Thread # (OMP_NUM_THREADS: 1-16)

- OpenMP
- Login to FX10
- Parallel Version of the Code by OpenMP
- **STREAM**

Why less than 16x ?

- Memory Contention
- Performance of memory per each thread decreases if number of threads on each node increases
- Sparse Matrix Solver: Memory-Bound
  - Effect of this decreasing is more significant
- Problem size is not so larger

Sparse/Dense Matrices

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    Y(i)= Y(i) + AMAT(k)*X(item(k))
  enddo
enddo
```

```
do j= 1, N
  do i= 1, N
    Y(j)= Y(j) + A(i,j)*X(i)
  enddo
enddo
```

- “X” in RHS
  - Dense: continuous on memory, easy to utilize cache
  - Sparse: continuity is not assured, difficult to utilize cache
    - more “memory-bound”

GeoFEM Benchmark

ICCG in FEM for Solid Mechanics

	SR11K/J2	SR16K/M1	T2K	FX10	京
Core #/Node	16	32	16	16	8
Peak Performance (GFLOPS)	147.2	980.5	147.2	236.5	128.0
STREAM Triad (GB/s)	101.0	264.2	20.0	64.7	43.3
B/F	0.686	0.269	0.136	0.274	0.338
GeoFEM (GFLOPS)	19.0	72.7	4.69	16.0	11.0
% to Peak	12.9	7.41	3.18	6.77	8.59
LLC/core (MB)	18.0	4.00	2.00	0.75	0.75

<u>Sparse Linear Solver: Memory-Bound</u>

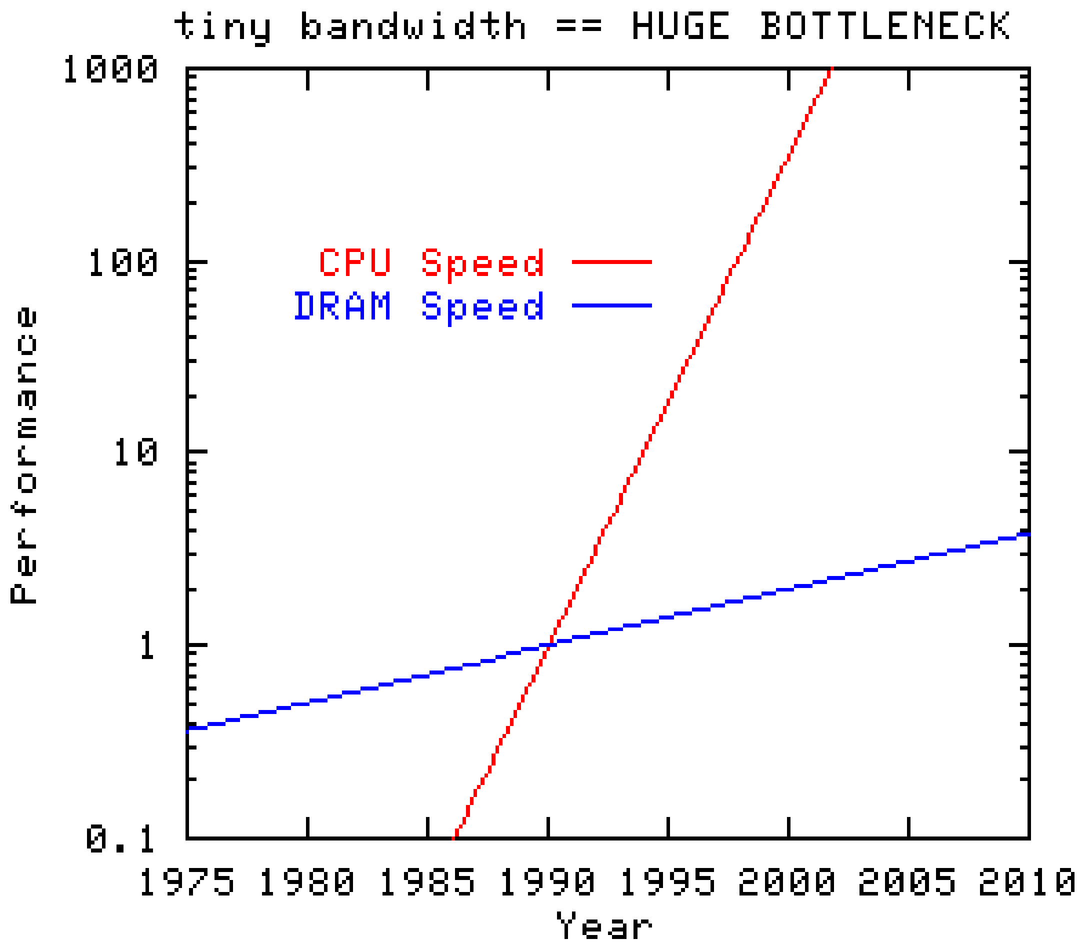
STREAM benchmark

http://www.cs.virginia.edu/stream/

- Benchmarks for Memory Bandwidth
  - Copy: c(i)= a(i)
  - Scale: c(i)= s*b(i)
  - Add: c(i)= a(i) + b(i)
  - Triad: c(i)= a(i) + s*b(i)

```
---------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
---------------------------------------------
 Number of processors =            16
 Array size =    2000000
 Offset     =          0
 The total memory requirement is     732.4 MB
(     45.8MB/task)
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
 ----------------------------------------------------
 ----------------------------------------------------
Function      Rate (MB/s)  Avg time   Min time  Max time
Copy:       18334.1898     0.0280      0.0279      0.0280
Scale:      18035.1690     0.0284      0.0284      0.0285
Add:        18649.4455     0.0412      0.0412      0.0413
Triad:      19603.8455     0.0394      0.0392      0.0398
```

Gap between performance of CPU and Memory



http://www.cs.virginia.edu/stream/

OpenMP version of STREAM

```
>$ cd <$O-stream>
>$ pjsub go.sh
```

- http://www.cs.virginia.edu/stream/
- C, Fortran, MPI, OpenMP etc.

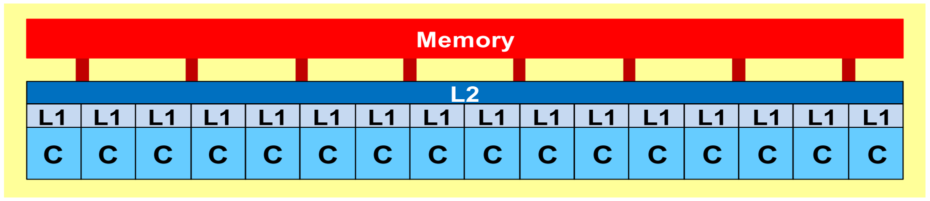
go.sh

```
#!/bin/sh
#PJM -L "rscgrp=lecture7"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j

export PATH=…
export LD_LIBRARY_PATH=…
export PARALLEL=16
export OMP_NUM_THREADS=16       Number of threads (1-16)

./stream.out > 16-01.lst 2>&1   Name of output file
```

Results of Triad

<$O-stream>/stream/*.lst

Peak is 85.3 GB/sec., 75%



Thread #	MB/sec.	Speed-up
1	8606.14	1.00
2	16918.81	1.97
4	34170.72	3.97
8	59505.92	6.91
16	64714.32	7.52

Exercises

- Running the code
- Try various number of threads (1-16)
- MPI-version and Single PE version are available
  - Fortran, C
  - Web-site of STREAM