Auto-tuning on NUMA and Many-core Environments with an FDM code

Takahiro Katagiri [i],
Satoshi Ohshima [ii], Masaharu Matsumoto [iii],

[i] Information Technology Center, Nagoya University, Japan
[ii] Research Institute for Information Technology, Kyusyu University, Japan
[iii] Graduate School of Information Science and Technology, The University of Tokyo, Japan

The Twelfth International Workshop on Automatic Performance Tuning
June 2, 2017, Buena Vista Palace Hotel, Orlando, Florida USA
Case-study of auto-tuning and optimization, 14:00 - 14:30
Chair: Toshiyuki Imamura, RIKEN AICS, Japan

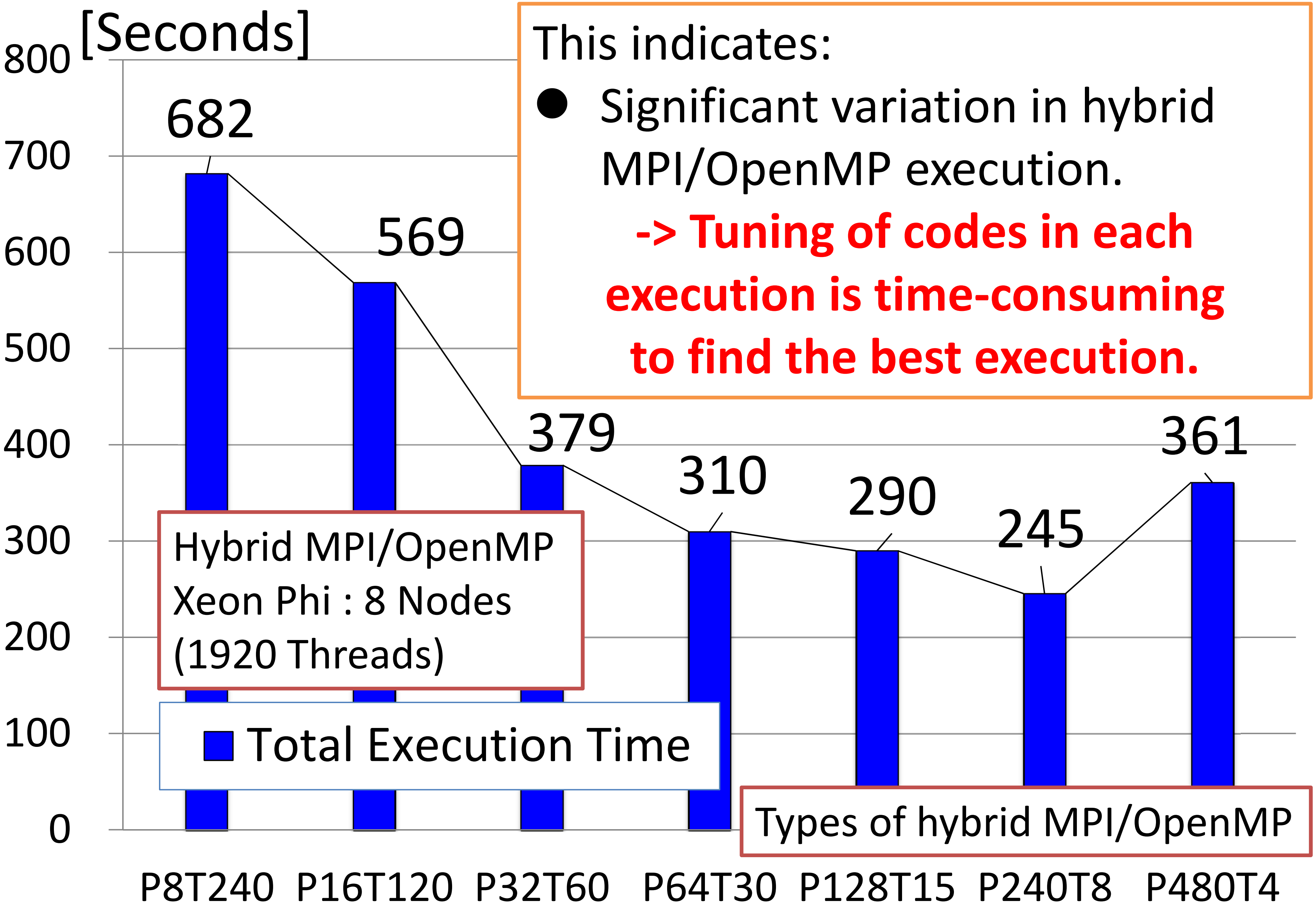
Outline

- Background
- An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
- Performance Evaluation with KNL (Knights Landing)
- Conclusion

Outline

- Background
- An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
- Performance Evaluation with KNL (Knights Landing)
- Conclusion

A Motivating Example (An simulation based on FDM)

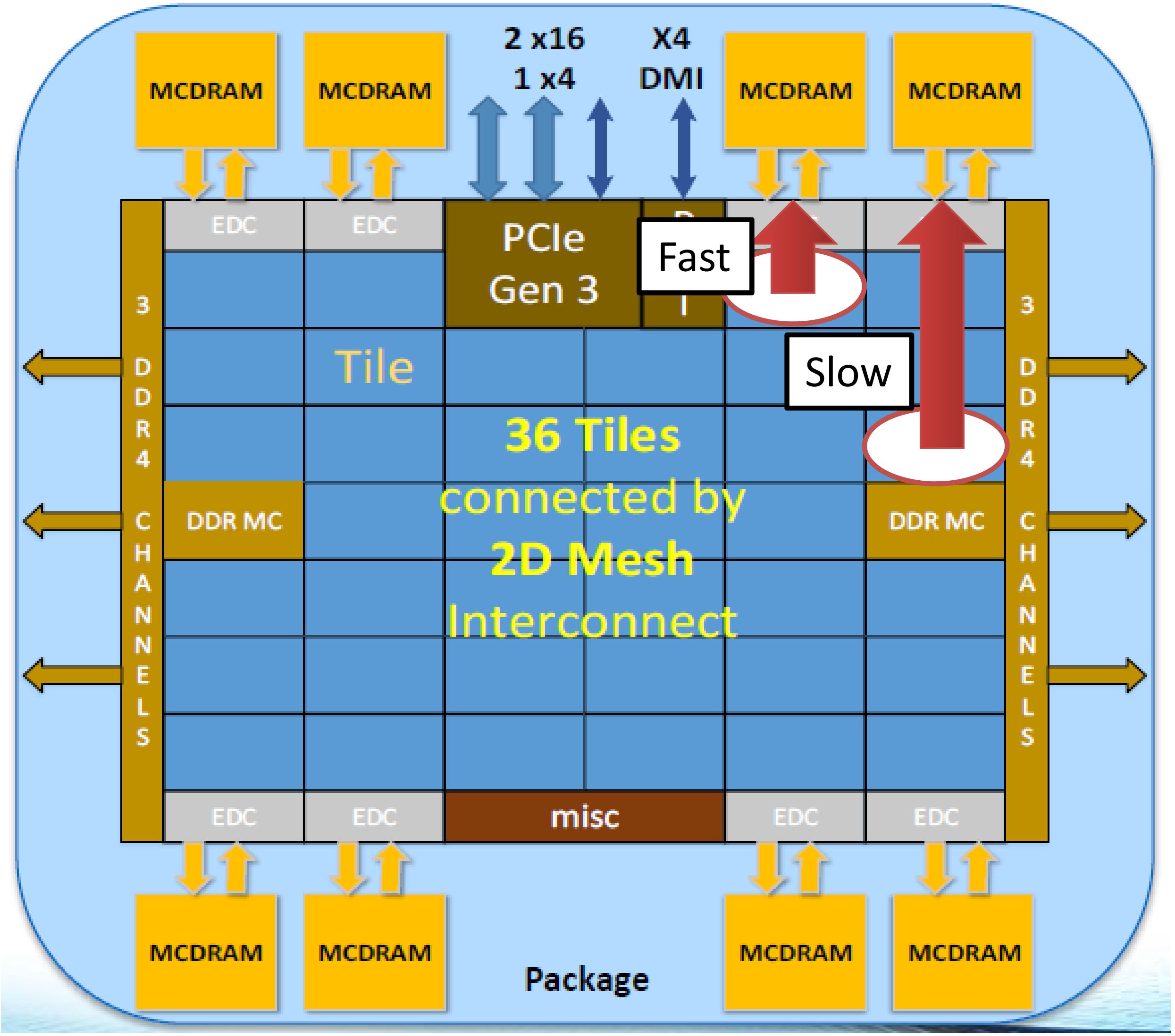


This indicates:

- Significant variation in hybrid MPI/OpenMP execution.

-> Tuning of codes in each execution is time-consuming to find the best execution.

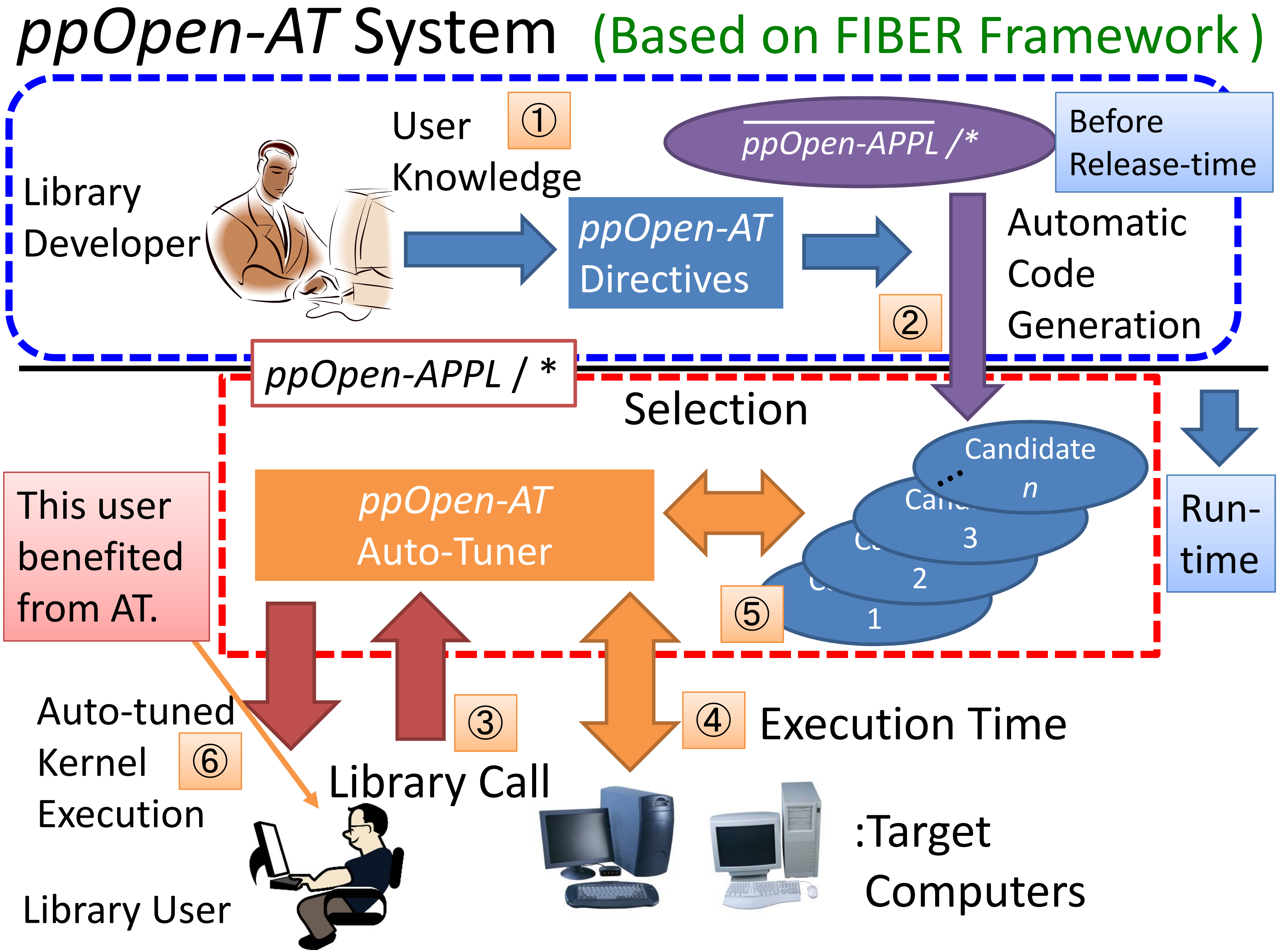
Current Many-core Architectures



Intel Xeon Phi (Knights Landing)

- Many Cores
  68 Physical Cores, Maximum 272 Threads, For each CPU
- Non Uniform Memory Access (NUMA)

Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.


ppOpen-AT System (Based on FIBER Framework)
Library Developer
User Knowledge
①
ppOpen-APPL /*
Before Release-time
ppOpen-AT Directives
②
Automatic Code Generation
ppOpen-APPL / *
Selection
Candidate n
3
2
1
ppOpen-AT Auto-Tuner
⑤
Run-time
This user benefited from AT.
Auto-tuned Kernel Execution
⑥
③
Library Call
④
Execution Time
:Target Computers
Library User

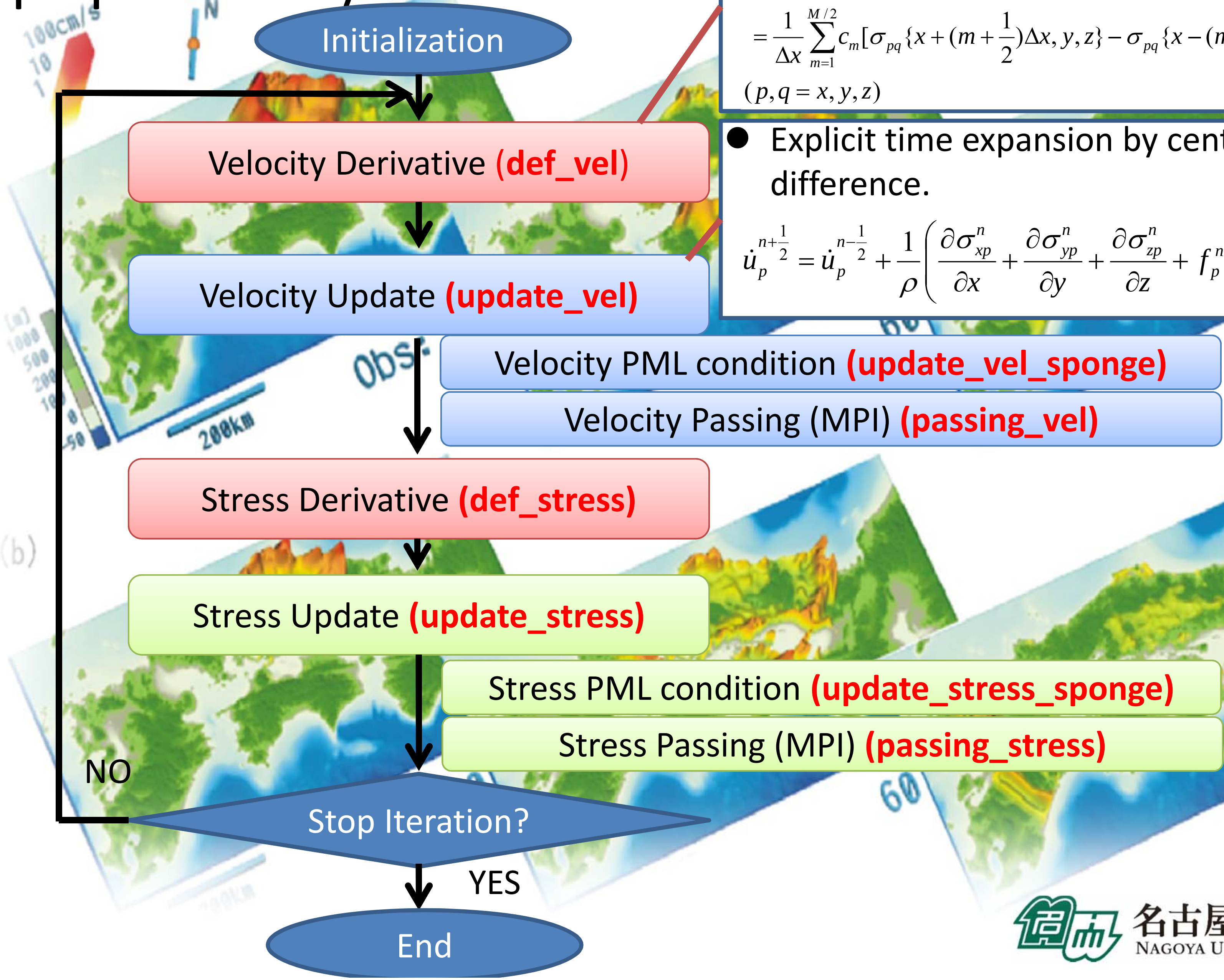
Outline

- Background
- An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
- Performance Evaluation with KNL (Knights Landing)
- Conclusion

Target Application

- **Seism3D**:
  Simulation for seismic wave analysis.
- Developed by Professor T. Furumura
  at the University of Tokyo.
  – The code is re-constructed as ppOpen-APPL/FDM.
- Finite Differential Method (FDM)
- 3D simulation
  – 3D arrays are allocated.
- Data type: Single Precision (real*4)

Flow Diagram of ppOpen-APPL/FDM



- Space difference by FDM.

$$\frac{d}{dx}\sigma_{pq}(x,y,z)$$

$$=\frac{1}{\Delta x}\sum_{m=1}^{M/2} c_m\left[\sigma_{pq}\{x+(m+\frac{1}{2})\Delta x,y,z\}-\sigma_{pq}\{x-(m-\frac{1}{2})\Delta x,y,z\}\right],$$

$$(p,q=x,y,z)$$

- Explicit time expansion by central difference.

$$\dot{u}_p^{n+\frac{1}{2}}=\dot{u}_p^{n-\frac{1}{2}}+\frac{1}{\rho}\left(\frac{\partial\sigma_{xp}^n}{\partial x}+\frac{\partial\sigma_{yp}^n}{\partial y}+\frac{\partial\sigma_{zp}^n}{\partial z}+f_p^n\right)\Delta t,(p=x,y,z)$$

名古屋大学 NAGOYA UNIVERSITY ITC

AT WITH LOOP TRANSFORMATION

Original Code

A Flow Dependency

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
  RL  = LAM (I,J,K)
  RM  = RIG (I,J,K)
  RM2 = RM + RM
  RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
  QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
  SXX (I,J,K) = ( SXX (I,J,K)+ (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
  SYY (I,J,K) = ( SYY (I,J,K)+ (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
  SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
  RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
  RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
  RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
  SXY (I,J,K) = ( SXY (I,J,K)   + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT) * QG
  SXZ (I,J,K) = ( SXZ (I,J,K)   + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT) * QG
  SYZ (I,J,K) = ( SYZ (I,J,K)    + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT) * QG
END DO
END DO
END DO
```

Loop Collapse – One dimensional

Merit: Loop length is huge.
This is good for OpenMP thread parallelism.

```
DO KK = 1, NZ * NY * NX
   K = (KK-1)/(NY*NX) + 1
   J = mod((KK-1)/NX,NY) + 1
   I = mod(KK-1,NX) + 1
   RL  = LAM (I,J,K)
   RM  = RIG (I,J,K)
   RM2 = RM + RM
   RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
   RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
   RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
   RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
   QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
   SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
   SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
   SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
   SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
   SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
   SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
```

Loop Collapse – Two dimensional

Merit: Loop length is huge.
This is good for OpenMP thread parallelism.

This I-loop enables us an opportunity of pre-fetching

```
DO KK = 1, NZ * NY
  K = (KK-1)/NY +  1
  J = mod(KK-1,NY) + 1
  DO I = 1, NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
    SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
  ENDDO
END DO
```

Loop Split with Re-Computation

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
   RL  = LAM (I,J,K)
   RM  = RIG (I,J,K)
   RM2 = RM + RM
   RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
   QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
   SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
   SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
   SZZ (I,J,K) = ( SZZ (I,J,K)   + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
ENDDO
DO I = 1, NX
   STMP1 = 1.0/RIG(I,J,K)
   STMP2 = 1.0/RIG(I+1,J,K)
   STMP4 = 1.0/RIG(I,J,K+1)
   STMP3 = STMP1 + STMP2
   RMAXY = 4.0/(STMP3  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
   RMAXZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I+1,J,K+1))
   RMAYZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I,J+1,K+1))
   QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
   SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
   SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
   SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
END DO
END DO
```

Re-computation is needed.
⇒Compilers do not apply it without directive.

Perfect Splitting

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
   RL  = LAM (I,J,K)
   RM  = RIG (I,J,K)
   RM2 = RM + RM
   RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
   QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
   SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
   SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
   SZZ (I,J,K) = ( SZZ (I,J,K)   + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
ENDDO;  ENDDO; ENDDO
```

Perfect Splitting

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
   STMP1 = 1.0/RIG(I,J,K)
   STMP2 = 1.0/RIG(I+1,J,K)
   STMP4 = 1.0/RIG(I,J,K+1)
   STMP3 = STMP1 + STMP2
   RMAXY = 4.0/(STMP3  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
   RMAXZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I+1,J,K+1))
   RMAYZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I,J+1,K+1))
   QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
   SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
   SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
   SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO;  END DO;  END DO;
```

ppOpen-AT Directives

: Loop Split & Collapse with data-flow dependence

```
!oat$ install LoopFusionSplit region start
!$omp parallel do private(k,j,i,STMP1,STMP2,STMP3,STMP4,RL,RM,RM2,RMAXY,RMAXZ,RMAYZ,RLTHETA,QG)
    DO K = 1, NZ
    DO J = 1, NY
    DO I = 1, NX
       RL  = LAM (I,J,K);   RM  = RIG (I,J,K);   RM2 = RM + RM
       RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
!oat$ SplitPointCopyDef  region  start
       QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
!oat$ SplitPointCopyDef  region  end
       SXX (I,J,K) = ( SXX (I,J,K) + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
       SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
       SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
!oat$ SplitPoint  (K, J, I)
       STMP1 = 1.0/RIG(I,J,K);  STMP2 = 1.0/RIG(I+1,J,K);  STMP4 = 1.0/RIG(I,J,K+1)
       STMP3 = STMP1 + STMP2
       RMAXY = 4.0/(STMP3 + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
       RMAXZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I+1,J,K+1))
       RMAYZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I,J+1,K+1))
!oat$ SplitPointCopyInsert
       SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
       SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
       SYZ (I,J,K) = ( SYZ (I,J,K) + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
    END DO;  END DO;  END DO
!$omp end parallel do
!oat$ install LoopFusionSplit region end
```

Specify Loop Split and Loop Fusion

Re-calculation is defined.

Loop Split Point

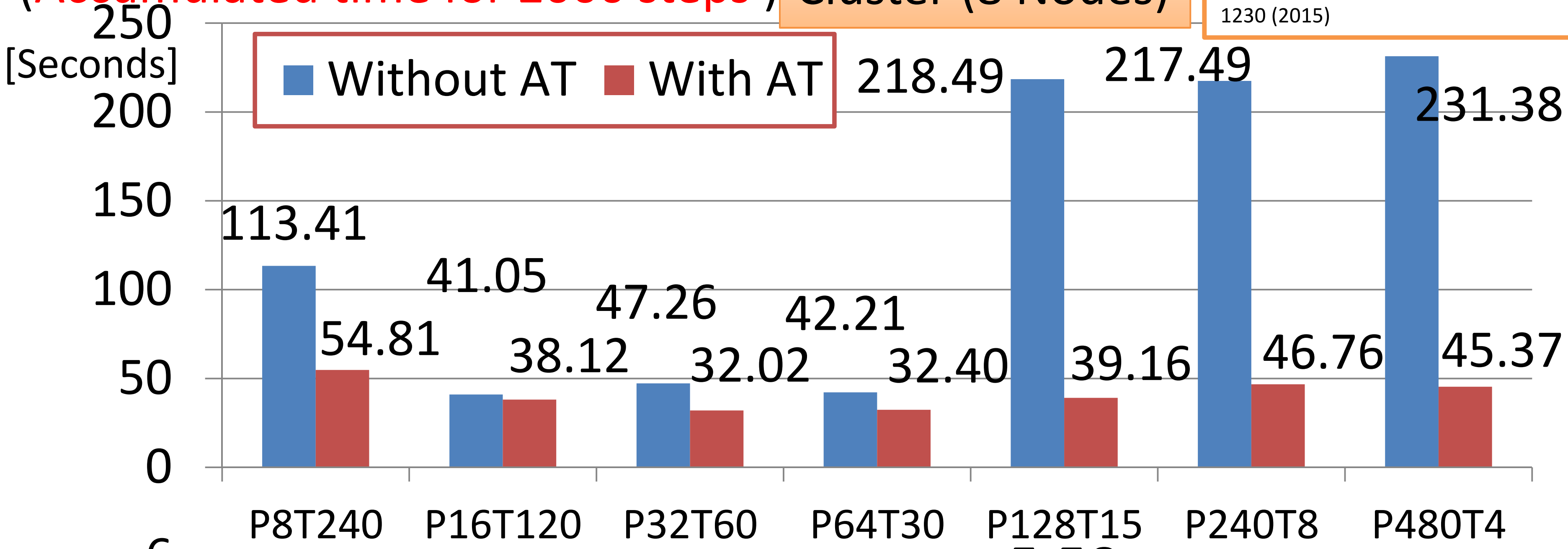
Using the re-calculation is defined.

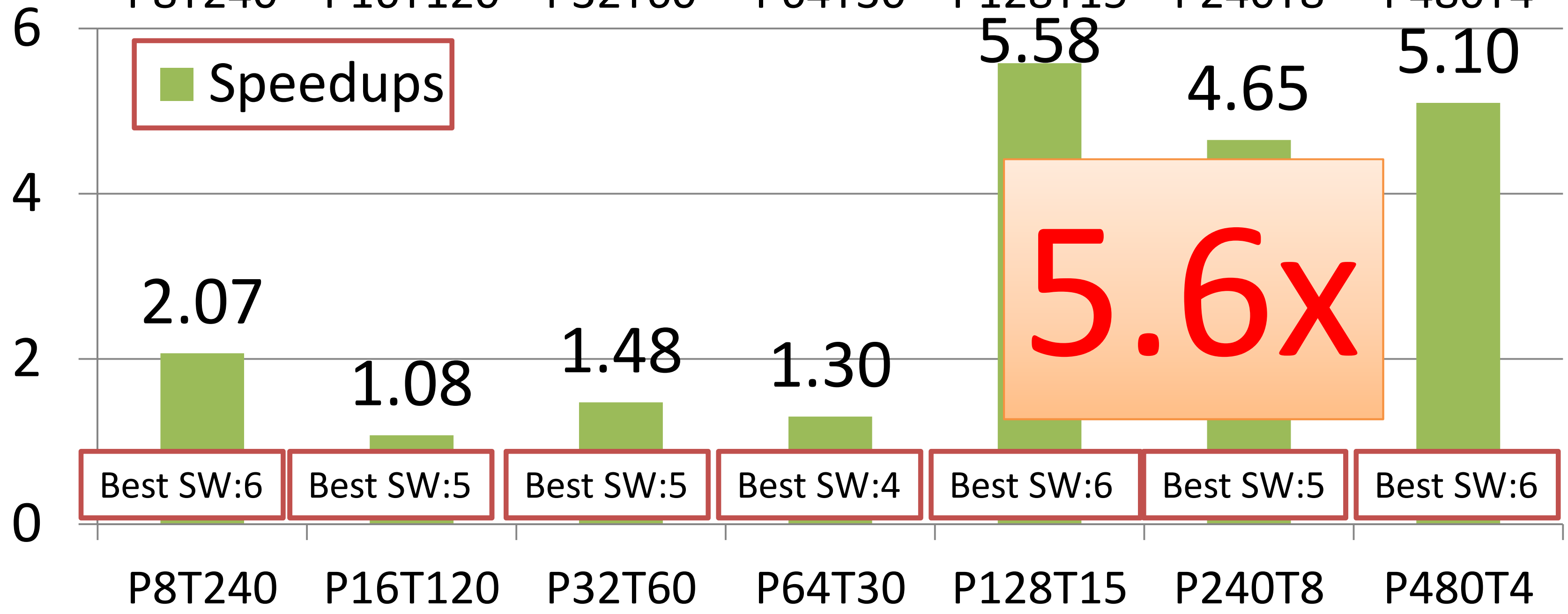
AT Effect (update_stress)

(Accumulated time for 2000 steps)

Xeon Phi (KNC) Cluster (8 Nodes)

T. Katagiri, S. Ohshima, M. Matsumoto: "Directive-based Auto-tuning for the Finite Difference Method on the Xeon Phi", Proc. of IPDPSW2015, pp.1221-1230 (2015)





AT WITH CODE SELECTION

Original Implementation (For Vector Machines)

Fourth-order accurate central-difference scheme for velocity. (**def_stress**)

```
call ppohFDM_pdiffx3_m4_OAT( VX,DXVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
call ppohFDM_pdiffy3_p4_OAT( VX,DYVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
call ppohFDM_pdiffz3_p4_OAT( VX,DZVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
call ppohFDM_pdiffy3_m4_OAT( VY,DYVY, NXP,NYP,NZP,NXP0,NXP1,NYP0,… )
call ppohFDM_pdiffx3_p4_OAT( VY,DXVY, NXP,NYP,NZP,NXP0,NXP1,NYP0,… )
call ppohFDM_pdiffz3_p4_OAT( VY,DZVY, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
call ppohFDM_pdiffx3_p4_OAT( VZ,DXVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
call ppohFDM_pdiffy3_p4_OAT( VZ,DYVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,… )
call ppohFDM_pdiffz3_m4_OAT( VZ,DZVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
```

```
 if( is_fs .or. is_nearfs ) then
       call ppohFDM_bc_vel_deriv( KFSZ,NIFS,NJFS,IFSX,IFSY,IFSZ,JFSX,JFSY,JFSZ )
end if
```

Process of model boundary.

```
call ppohFDM_update_stress(1, NXP, 1, NYP, 1, NZP)
```

Explicit time expansion by leap-frog scheme. (update_stress)

Original Implementation (For Vector Machines)

```
subroutine OAT_InstallppohFDMupdate_stress(..)
!$omp parallel do private(i,j,k,RL1,RM1,RM2,RLRM2,DXVX1,DYVY1,DZVZ1,…)
  do k = NZ00, NZ01
    do j = NY00, NY01
      do i = NX00, NX01
        RL1   = LAM (I,J,K); RM1   = RIG (I,J,K);  RM2   = RM1 + RM1; RLRM2 = RL1+RM2
        DXVX1 = DXVX(I,J,K);  DYVY1 = DYVY(I,J,K);  DZVZ1 = DZVZ(I,J,K)
        D3V3  = DXVX1 + DYVY1 + DZVZ1
        SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
        SYY (I,J,K) = SYY (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
        SZZ (I,J,K) = SZZ (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
        DXVYDYVX1 = DXVY(I,J,K)+DYVX(I,J,K);  DXVZDZVX1 = DXVZ(I,J,K)+DZVX(I,J,K)
        DYVZDZVY1 = DYVZ(I,J,K)+DZVY(I,J,K)
        SXY (I,J,K) = SXY (I,J,K) + RM1 * DXVYDYVX1 * DT
        SXZ (I,J,K) = SXZ (I,J,K) + RM1 * DXVZDZVX1 * DT
        SYZ (I,J,K) = SYZ (I,J,K) + RM1 * DYVZDZVY1 * DT
      end do
    end do
  end do
retuen
end
```

Explicit time expansion by leap-frog scheme. (update_stress)

Input and output for arrays in each call -> Increase of B/F ratio: ~1.7

The Code Variants (For Scalar Machines)

- Variant1 (IF-statements inside)
  - The followings include inside loop:
  1. Fourth-order accurate central-difference scheme for velocity.
  2. Process of model boundary.
  3. Explicit time expansion by leap-frog scheme.
- Variant2 (IF-free, but there is IF-statements inside loop for process of model boundary.)
  - To remove IF sentences from the variant1, the loops are reconstructed.
  - The order of computations is changed, but the result without round-off errors is same.
  - **[Main Loop]**
  1. Fourth-order accurate central-difference scheme for velocity.
  2. Explicit time expansion by leap-frog scheme.
  - **[Loop for process of model boundary]**
  1. Fourth-order accurate central-difference scheme for velocity.
  2. Process of model boundary.
  3. Explicit time expansion by leap-frog scheme.

Variant1 (For Scalar Machines)

Stress tensor of Sxx, Syy, Szz

```
 !$omp parallel do   private
(i,j,k,RL1,RM1,RM2,RLRM2,DXVX, …)
 do k_j=1, (NZ01-NZ00+1)*(NY01-NY00+1)
   k=(k_j-
   j=mod(
   do i = N
      RL1
      RM2
  !4th ord
     DXV
        - (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
     DYVY0 = (VY(I,J,K)  -VY(I,J-1,K))*C40/dy &
        - (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
     DZVZ0 = (VZ(I,J,K)  -VZ(I,J,K-1))*C40/dz &
        - (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
! truncate_diff_vel
! X dir
      if (idx==0) then
        if (i==1)then
          DXVX0 = ( VX(1,J,K) - 0.0_PN    )/ DX
        end if
        if (i==2) then
          DXVX0 = ( VX(2,J,K) - VX(1,J,K) )/ DX
         end if
      end if
      if( idx == IP-1 ) then
        if (i==NXP)then
    DXVX0 = ( VX(NXP,J,K) - VX(NXP-1,J,K) ) /  DX
        end if
```

Fourth-order accurate central-difference scheme for velocity.

```
! Y dir
      if( idy == 0 ) then ! Shallowmost
        if (j==1)then
          DYVY0 = ( VY(I,1,K) - 0.0_
        end if
        if (j==2)then
          DYVY0 = ( VY(I,2,K) - VY(
        end if
      end if
      if( idy == JP-1 ) then
        if (j==NYP)then
          DYVY0 = ( VY(I,NYP,K) - V
        end if
      end if
! Z dir
      if( idz == 0 ) then ! Shallowmost
        if (k==1)then
          DZVZ0 = ( VZ(I,J,1) - 0.0_
        end if
        if (k==2)then
          DZVZ0 = ( VZ(I,J,2) - VZ(
        end if
      end if
      if( idz == KP-1 ) then
        if (k==NZP)then
          DZVZ0 = ( VZ(I,J,NZP) -
        end if
      end if
```

Explicit time expansion by leap-frog scheme

```
DXVX1 = DXVX0;  DYVY1 = DYVY0
DZVZ1 = DZVZ0;  D3V3  = DXVX1 + DYVY1 + DZVZ1
  SXX (I,J,K) = SXX (I,J,K) &
    + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
  SYY (I,J,K) = SYY (I,J,K) &
    + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
  SZZ (I,J,K) = SZZ (I,J,K) &
    + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
   end do
 end do
!$omp end parallel do
```

☺B/F ratio is reduced to <u>0.4</u>

☹IF sentences inside – it is difficult to optimize code by compiler.

Process of model boundary.

Variant2 (IF-free)

Stress tensor of Sxx, Syy, Szz

```
!$omp parallel do   private(i,j,k,RL1,RM1,・・・)
do k_j=1, (NZ01-NZ00+1)*(NY01-NY00+1)
  k=(k_j-1)/(NY01-NY00+1)+NZ00
  j=mod((k_j-1),(NY01-NY00+1))+NY00
        = NX00, NX01
           = LAM (I,J,K); RM1   = RIG (I,J,K);
         2   = RM1 + RM1; RLRM2 = RL1+RM2
          rder diff (DXVX,DYVY,DZVZ)
          VX0 = (VX(I,J,K)  -VX(I-1,J,K))*C40/dx -  (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
          VY0 = (VY(I,J,K)  -VY(I,J-1,K))*C40/dy  - (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
      DZVZ0 = (VZ(I,J,K)  -VZ(I,J,K-1))*C40/dz  - (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
      DXVX1 = DXVX0; DYVY1 = DYVY0;
      DZVZ1 = DZVZ0;
      D3V3  = DXVX1 + DYVY1 + DZVZ1;
      SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2* (DZVZ1+DYVY1) ) * DT
      SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2* (DXVX1+DZVZ1) ) * DT
      SZZ (I,J,K) = SZZ (I,J,K)  + (RLRM2*(D3V3)-RM2* (DXVX1+DYVY1) ) * DT
    end do
end do
!$omp end parallel do
```

Fourth-order accurate central-difference scheme for velocity.

Explicit time expansion by leap-frog scheme.

☺Win-win between B/F ratio and optimization by compiler.

Variant2 (IF-free)

Loop for process of model boundary

```
! 2nd replace
if( is_fs .or. is_nearfs ) then
!$omp parallel do private(i,j,k,RL1,RM
do i=NX00,NX01
   do j=NY00,NY01
      do k = KFSZ(i,j)-1, KFSZ(i,j)+1, 2
         RL1   = LAM (I,J,K); RM1   = RIG
         RM2   = RM1 + RM1; RLRM2 = RL1+RM2
! 4th order diff
         DXVX0 = (VX(I,J,K)  -VX(I-1,J,K))*C40/dx &
              - (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
         DYVX0 = (VX(I,J+1,K)-VX(I,J,K)  )*C40/dy &
              - (VX(I,J+2,K)-VX(I,J-1,K))*C41/dy
         DXVY0 = (VY(I+1,J,K)-VY(I  ,J,K))*C40/dx &
             - (VY(I+2,J,K)-VY(I-1,J,K))*C41/dx
         DYVY0 = (VY(I,J,K)  -VY(I,J-1,K))*C40/dy &
             - (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
         DXVZ0 = (VZ(I+1,J,K)-VZ(I  ,J,K))*C40/dx &
             - (VZ(I+2,J,K)-VZ(I-1,J,K))*C41/dx
         DYVZ0 = (VZ(I,J+1,K)-VZ(I,J,K)  )*C40/dy &
             - (VZ(I,J+2,K)-VZ(I,J-1,K))*C41/dy
         DZVZ0 = (VZ(I,J,K)  -VZ(I,J,K-1))*C40/dz &
              - (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
```

Process of model boundary.

Fourth-order accurate central-difference scheme for velocity

```
! vel derive
   if (K==KFSZ(I,J)+1) then
      DZVX0 = ( VX(I,J,KFSZ(I,J)+2)-VX(I,J,KFSZ(I,J)+1) )/ DZ
      DZVY0 = ( VY(I,J,KFSZ(I,J)+2)-VY(I,J,KFSZ(I,J)+1) )/ DZ
   else if (K==KFSZ(I,J)-1) then
      DZVX0 = ( VX(I,J,KFSZ(I,J)  )-VX(I,J,KFSZ(I,J)-1) )/ DZ
      DZVY0 = ( VY(I,J,KFSZ(I,J)  )-VY(I,J,KFSZ(I,J)-1) )/ DZ
   end if
   DXVX1 = DXVX0
   DYVY1 = DYVY0
   DZVZ1 = DZVZ0
   D3V3  = DXVX1 + DYVY1 + DZVZ1
   DXVYDYVX1 = DXVY0+DYVX0
   DXVZDZVX1 = DXVZ0+DZVX0
   DYVZDZVY1 = DYVZ0+DZVY0
   if (K==KFSZ(I,J)+1)then
      KK=2
   else
      KK=1
   end if
   SXX (I,J,K) = SSXX (I,J,KK) &
      + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
   SYY (I,J,K) = SSYY (I,J,KK) &
      + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
   SZZ (I,J,K) = SSZZ (I,J,KK) &
      + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
   SXY (I,J,K) = SSXY (I,J,KK) + RM1 * DXVYDYVX1 * DT
   SXZ (I,J,K) = SSXZ (I,J,KK) + RM1 * DXVZDZVX1 * DT
   SYZ (I,J,K) = SSYZ (I,J,KK) + RM1 * DYVZDZVY1 * DT
      end do
   end do
end do
!$omp end parallel do
```

Explicit time expansion by leap-frog scheme.

Code Selection by ppOpen-AT and Hierarchical AT

Upper Code

With Select clause, code selection can be specified.

```
Program main
....
!OAT$ install select region start
!OAT$ name ppohFDMupdate_vel_select
!OAT$ select sub region start
call ppohFDM_pdiffx3_p4( SXX,DXSXX,NXP,NYP,NZP,....)
call ppohFDM_pdiffy3_p4( SYY,DYSYY, NXP,NYP,NZP,.....)
...
if( is_fs .or. is_nearfs ) then
  call ppohFDM_bc_stress_deriv( KFSZ,NIFS,NJFS,II
end if
call ppohFDM_update_vel    ( 1, NXP, 1, NYP, 1, NZ
!OAT$ select sub region end

!OAT$ select sub region start
Call ppohFDM_update_vel_Intel  ( 1, NXP, 1, NYP,
!OAT$ select sub region end

!OAT$ install select region end
```

Lower Code

```
subroutine ppohFDM_pdiffx3_p4(....)
....
!OAT$ install LoopFusion region start
....
```

```
subroutine ppohFDM_update_vel(....)
....
!OAT$ install LoopFusion region start
!OAT$ name ppohFDMupdate_vel
!OAT$ debug (pp)
!$omp parallel do private(i,j,k,ROX,ROY,ROZ)
do k = NZ00, NZ01
  do j = NY00, NY01
    do i = NX00, NX01
   .....
....
```

The Number of AT Candidates (ppOpen-APPL/FDM)

Kernel Names	AT Objects	The Number of Candidates
1. update_stress	・Loop Collapses and Splits ：8 Kinds ・Code Selections : 2 Kinds	**10**
2. update_vel	・Loop Collapses, Splits, and re-ordering of statements: ：6 Kinds ・Code Selections: 2 Kinds	**8**
3. update_stress_sponge	・Loop Collapses：3 Kinds	**3**
4. update_vel_sponge	・Loop Collapses：3 Kinds	**3**
5. ppohFDM_pdiffx3_p4	Kernel Names：def_update、def_vel ・Loop Collapses：3 Kinds	**3**
6. ppohFDM_pdiffx3_m4		**3**
7. ppohFDM_pdiffy3_p4		**3**
8. ppohFDM_pdiffy3_m4		**3**
9. ppohFDM_pdiffz3_p4		**3**
10.ppohFDM_pdiffz3_m4		**3**
11. ppohFDM_ps_pack	Data packing and unpacking ・Loop Collapses： 3 Kinds	**3**
12. ppohFDM_ps_unpack		**3**
13. ppohFDM_pv_pack		**3**
14. ppohFDM_pv_unpack		**3**

- Total：54 Kinds
- Hybrid MPI/OpenMP: 7 Kinds
- 54×7 = 378 Kinds

Outline

- Background
- An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
- Performance Evaluation with KNL (Knights Landing)
- Conclusion

Oakforest-PACS (ITC, U. Tokyo and CCS, Tsukuba U.) :Fujitsu PRIMERGY CX1640 M1

8,208 Nodes (558,144 Cores)

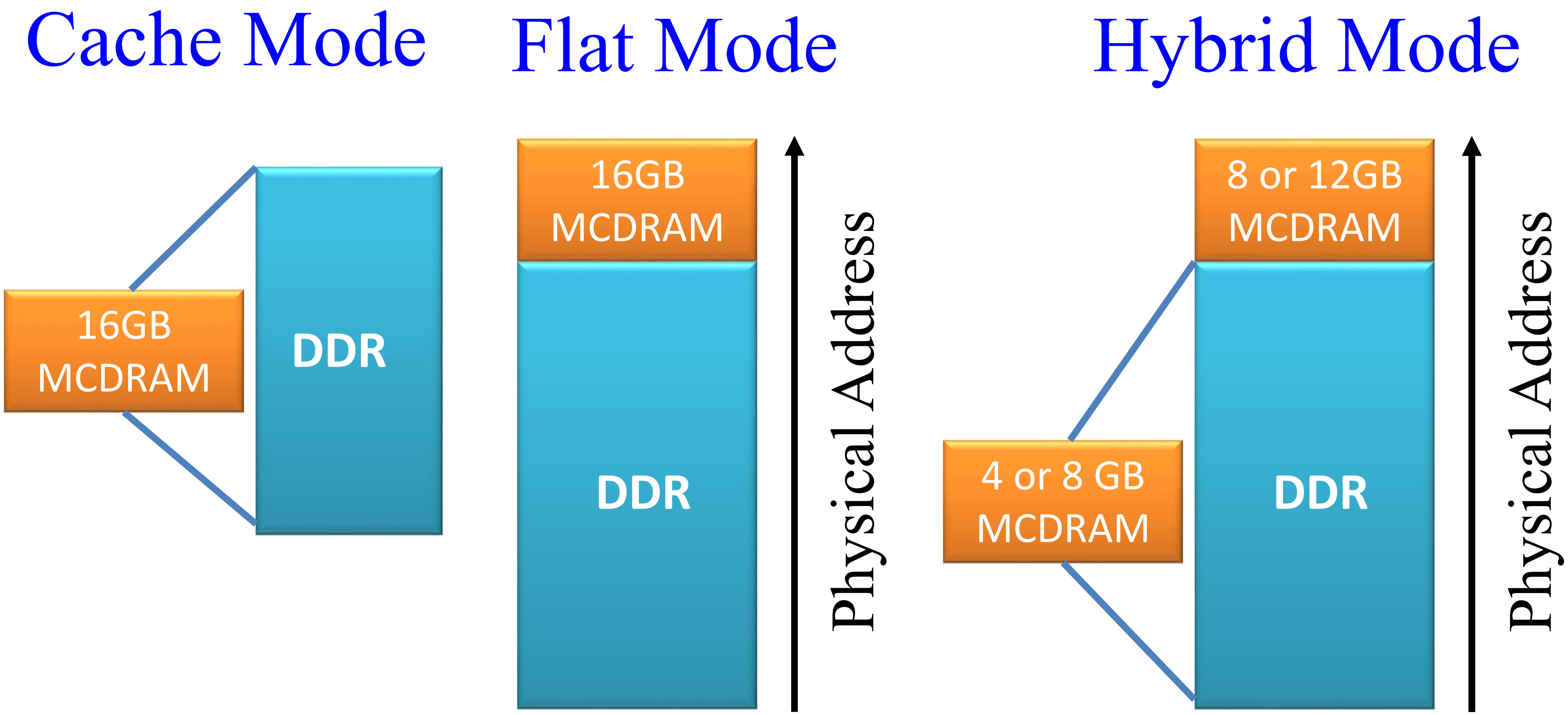
● Total Organization

item	Specification
Total Theoretical Peak	**25.004 PFLOPS**
Total Number of Nodes	8208
Total Memory Amount	897 TByte
Total Storage Amount	26 PB
Total Storage Bandwidth	500 GB/sec
Total Fast Cache Amount	940 TB
Total Fast Cache Bandwidth	1,560 GB/sec

● Node Configuration

CPU	Intel® Xeon Phi ™ 7250 (Knights Landing, KNL)
#Processors	1 (68 Physical Cores)
Frequency	1.4 GHz
Theoretical Peak for each Node	3.0464 Tflops (double) / 6.0928 Tflops (single)
Memory	**96 GB(DDR4)＋ 16 GB(MCDRAM)**
Inter Connect	Intel ® Omni-Path Network (100 Gbps)

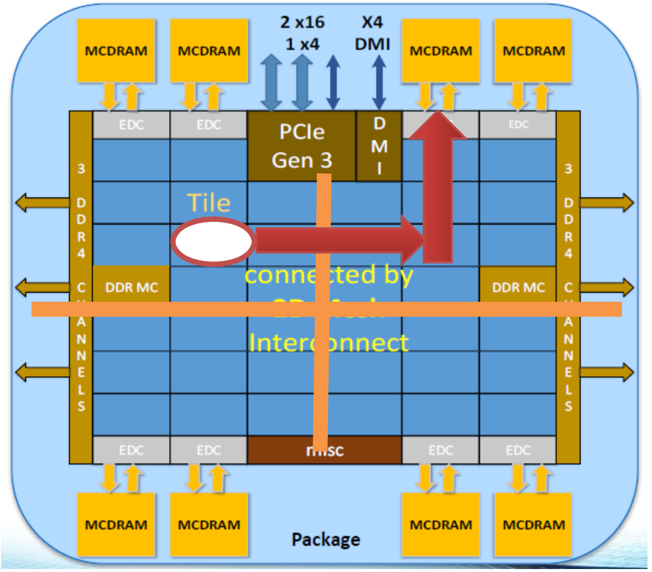
Execution Modes of KNL



Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.

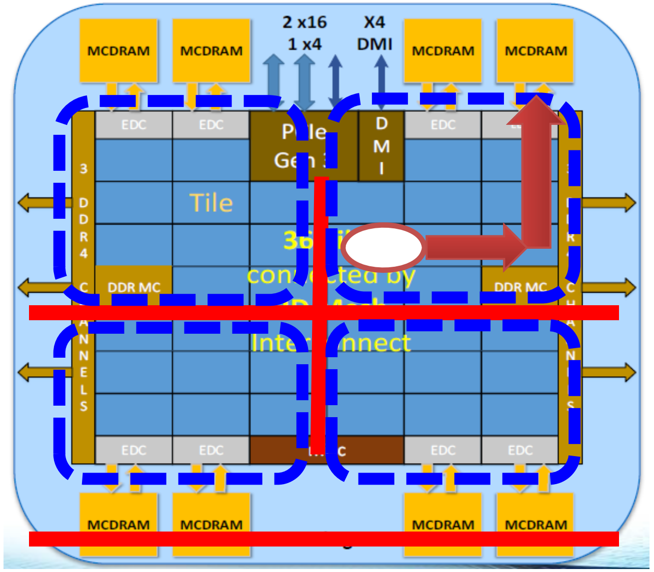
Mesh Modes of KNL

Quadrant



- Chip divided into **four virtual Quadrants.**
- Address hashed to a Directory in the same quadrant as the Memory.
- Affinity between the Directory and Memory.

Sub-NUMA Clustering (SNC)



- Each Quadrant (Cluster) exposed as **a separate NUMA domain** to OS.
- Looks analogous to 4-Socket Xeon Affinity between Tile, Directory and Memory.
- Local communication. Lowest latency of all modes.
- SW **needs to NUMA optimize** to get benefit.

Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.

Memory and Mesh Modes in This Experiments

1. FLAT-QUADRANT
   – Memory Mode: FLAT, Mesh Mode: QUADRANT
2. FLAT-SNC4
   – Memory Mode: FLAT, Mesh Mode: SNC with 4 clusters
3. CACHE-QUADRANT
   – Memory Mode: CACHE, Mesh Mode: QUADRANT
4. CACHE-SNC4
   – Memory Mode: CACHE, Mesh Mode: SNC with 4 clusters

- NUMA affinity is set by appropriate manner.
- Arrays set in DDR4 in CACHE mode.

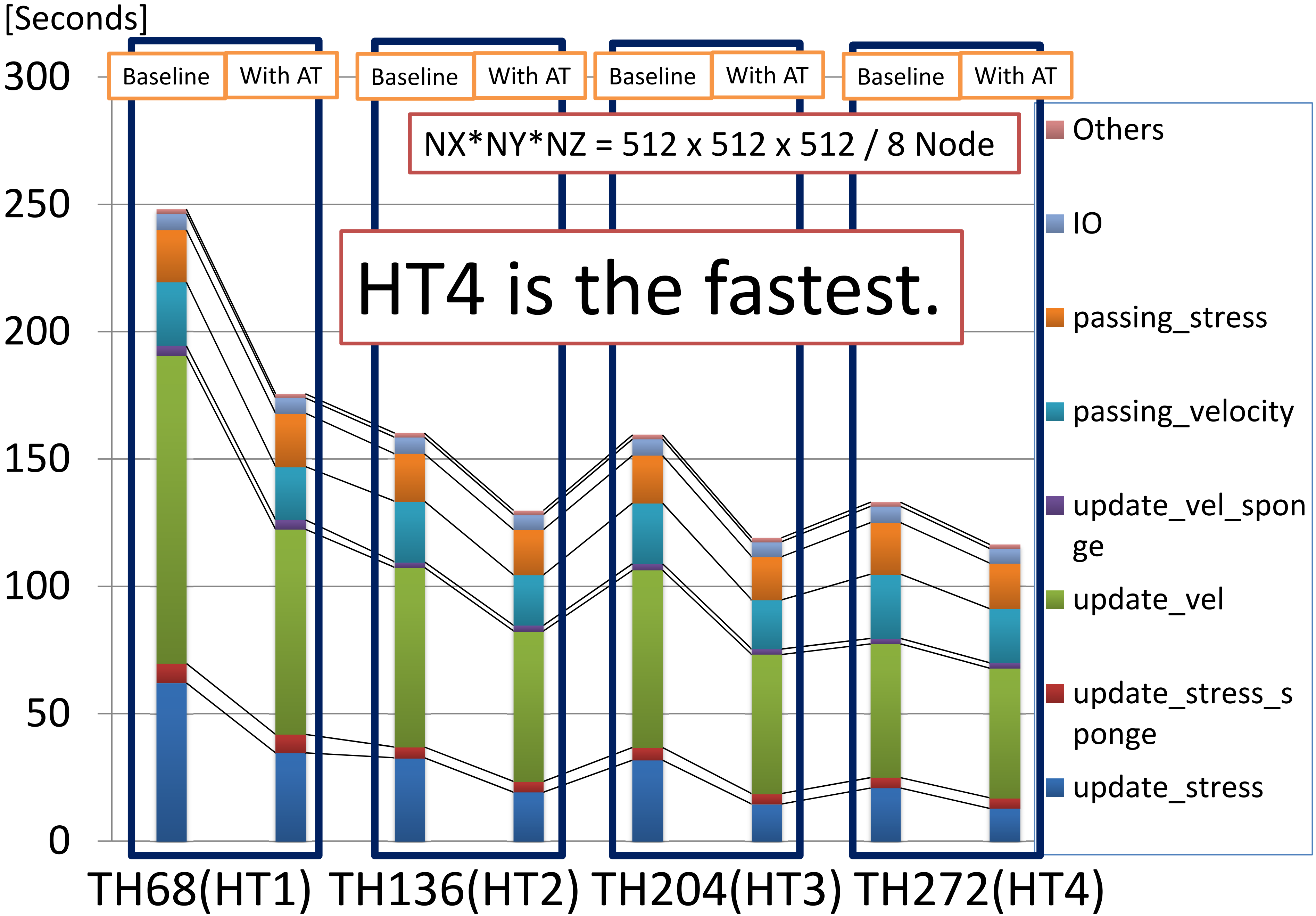
Execution Details

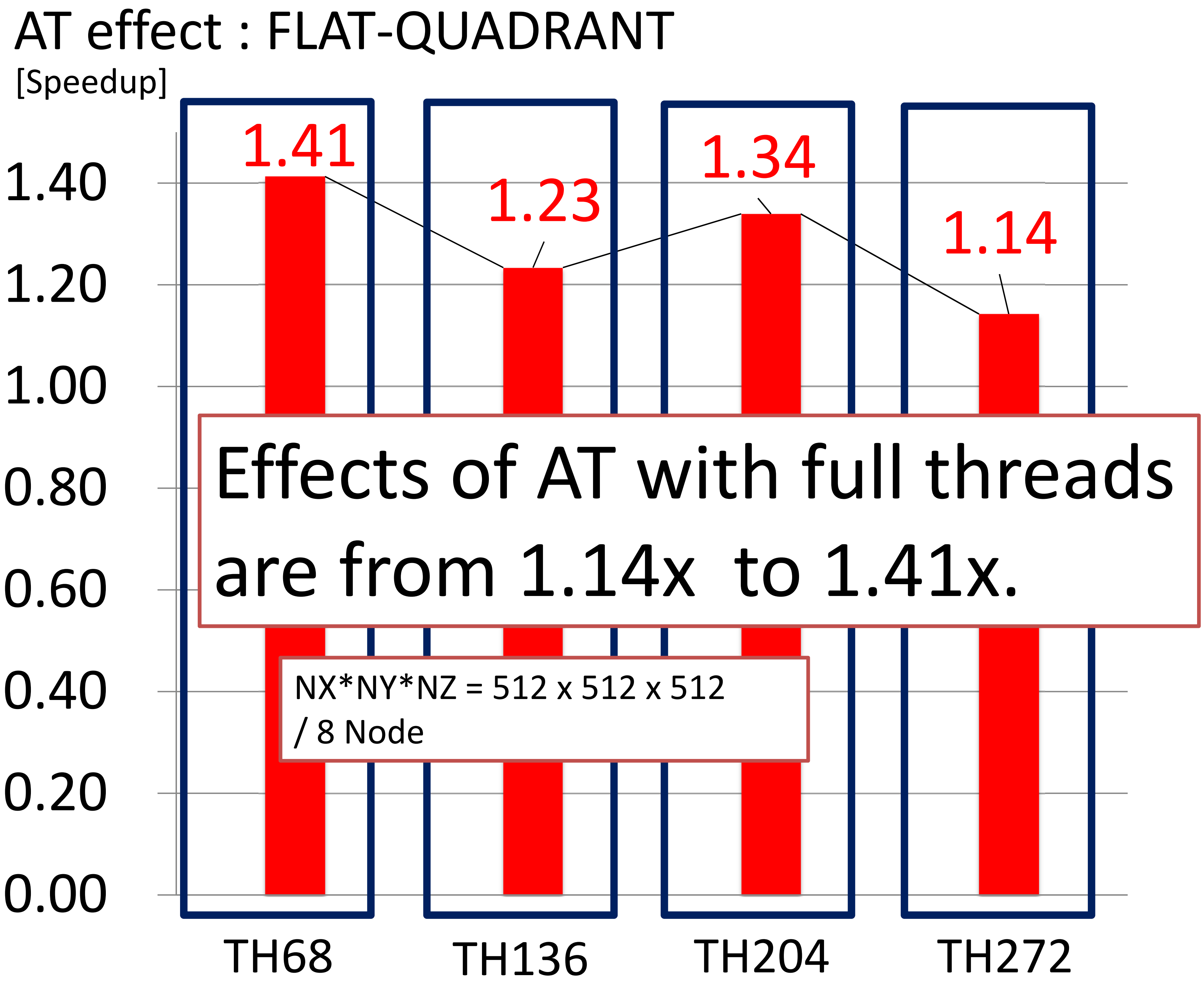
- ppOpen-APPL/FDM ver.0.2
- ppOpen-AT ver.0.2
- The number of time step: 2000 steps
- The number of nodes: 8 node
- Target Problem Size (Almost maximum size with 8 GB/node)
  - NX * NY * NZ = 512 x 512 x 512 / 8 Node
  - NX * NY * NZ = 256 * 256 * 256 / node (!= per MPI Process)
- Target MPI Processes and Threads on the Xeon Phi
  - 1 node of the KNL with 4 HTs (Hyper Threading)
    - **We checked performance of H1-H4, then the H4 is the fastest.**
  - P*X*T*Y*: *X* MPI Processes and *Y* Threads per process
  - P8T272 : Minimum Hybrid MPI-OpenMP execution for ppOpen-APPL/FDM, since it needs minimum 8 MPI Processes.
  - P16T136, P32T68, P64T34, P128T17, P256T8, P512T4, P1024T2
  - P2048T1: pure MPI
- The number of iterations for the kernels: 100

名古屋大学 NAGOYA UNIVERSITY ITC

EFFECT OF HYPER THREADINGS WITH FULL THREADS EXECUTION PER NODE

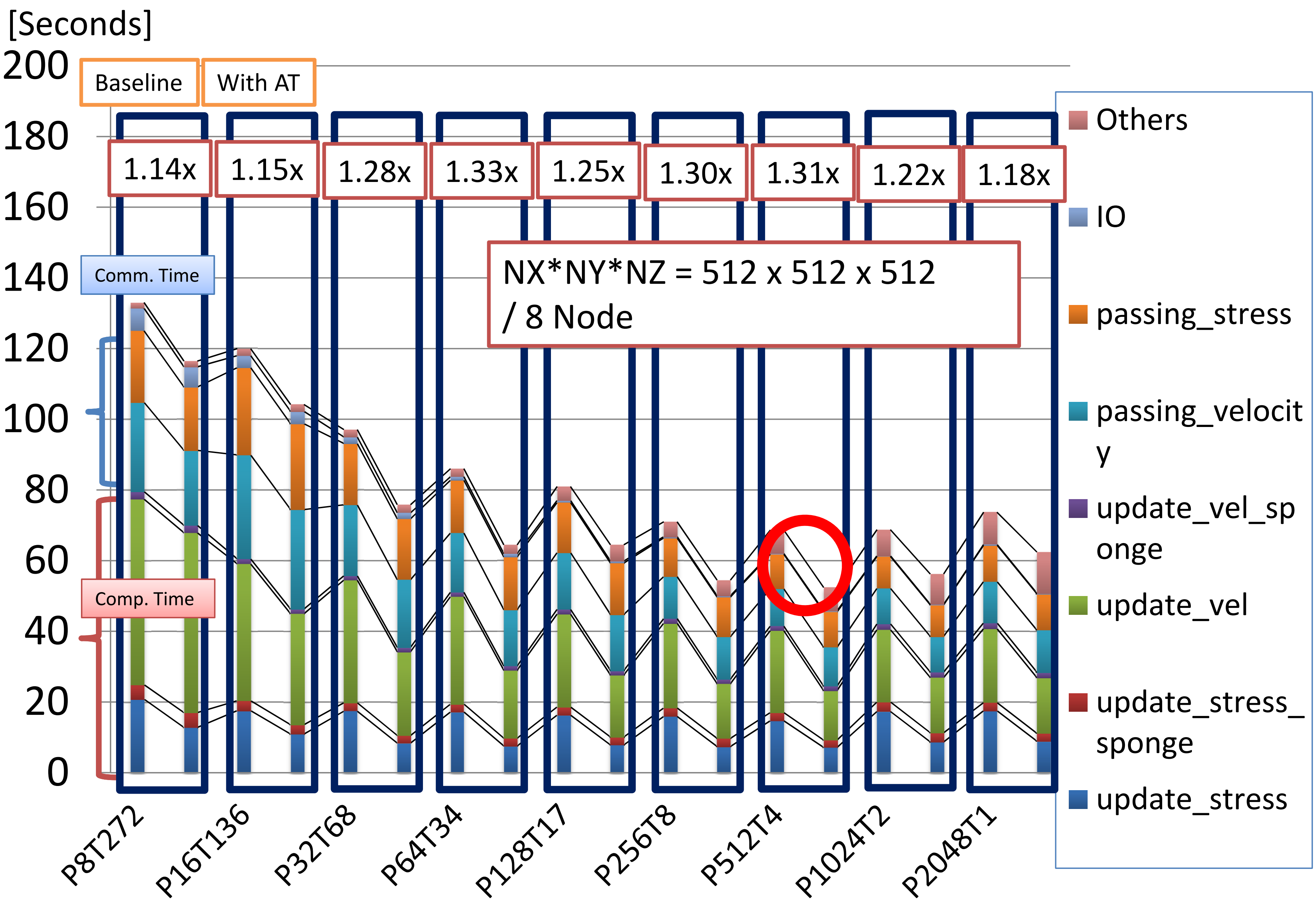
Whole Time（2000 Time Steps）:FLAT-QUADRANT



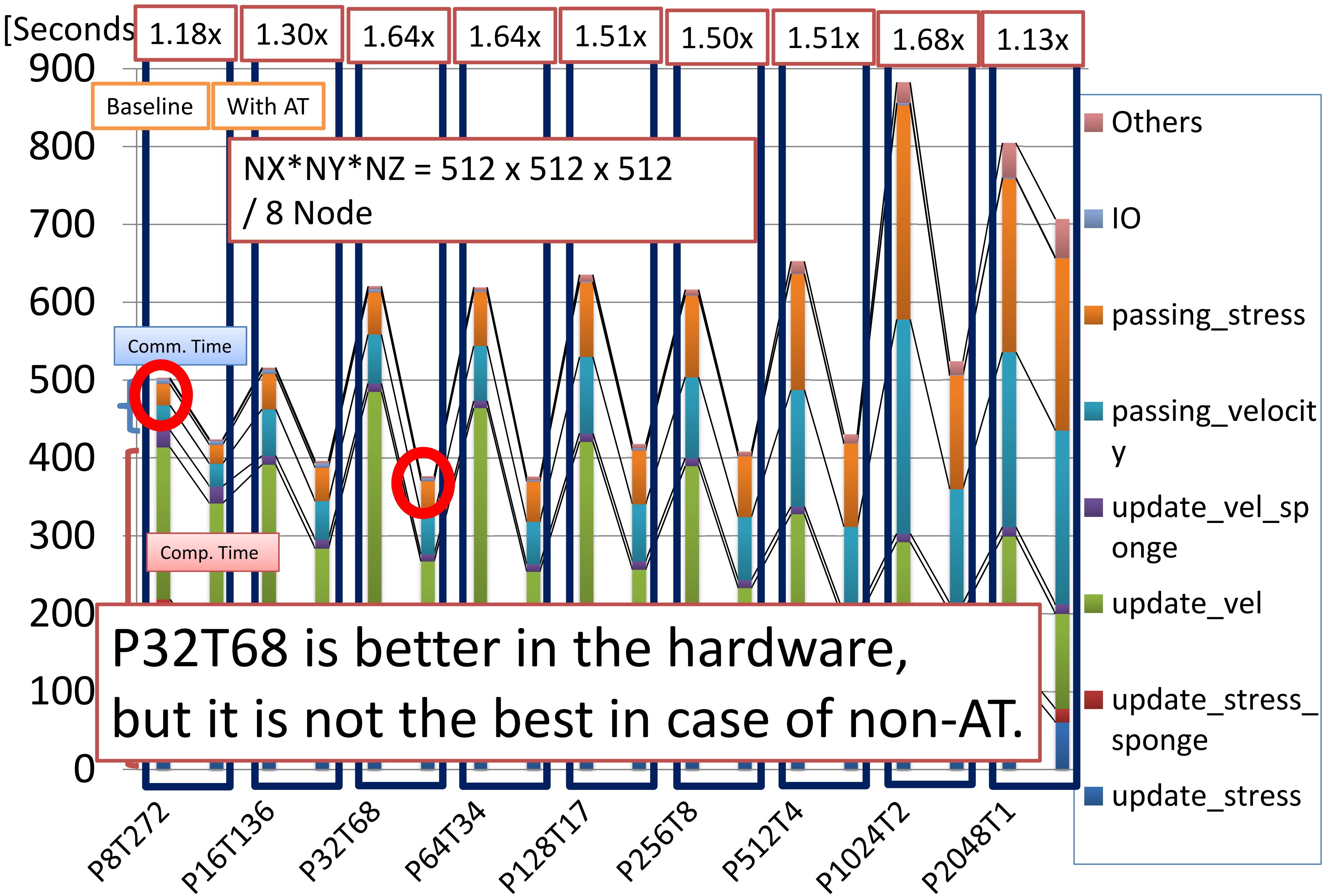

AT effect : FLAT-QUADRANT
[Speedup]
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
1.41
1.23
1.34
1.14
Effects of AT with full threads are from 1.14x to 1.41x.
NX*NY*NZ = 512 x 512 x 512
/ 8 Node
TH68
TH136
TH204
TH272

RESULTS OF HYBRID MPI / OPENMP WITH HT4

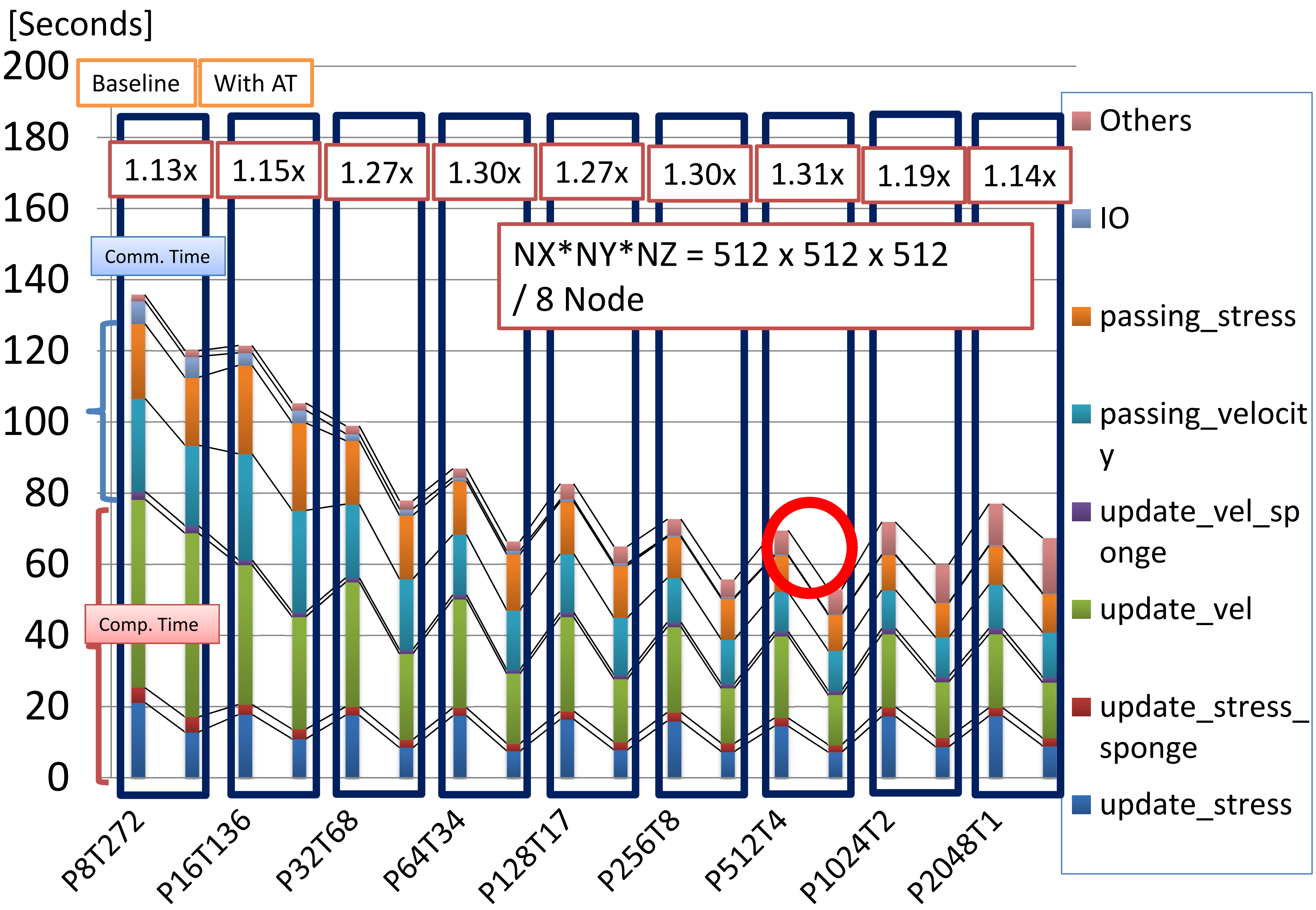
Whole Time（2000 Time Steps）:FLAT-QUADRANT



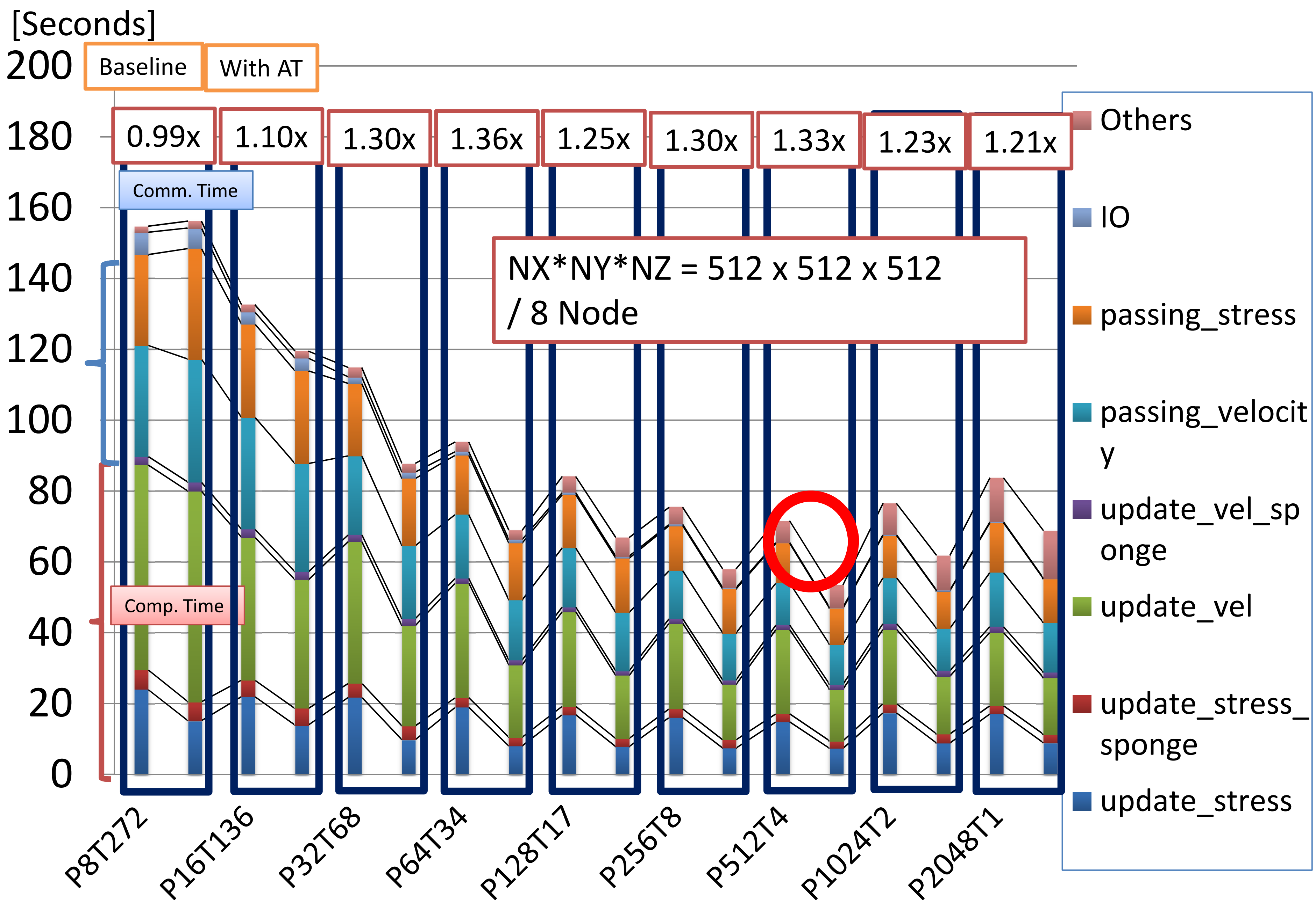
Whole Time（2000 Time Steps）:FLAT-SNC4



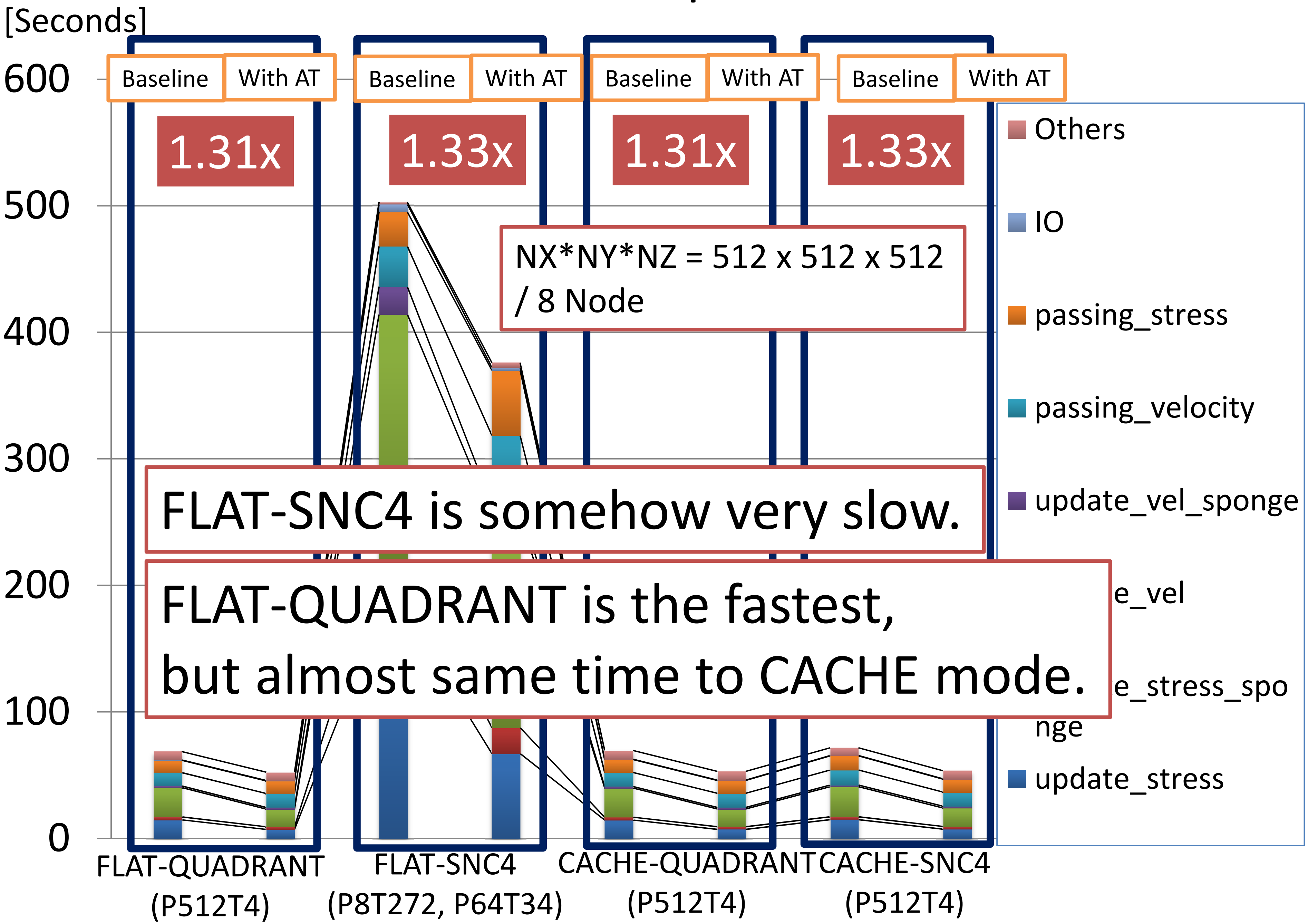
Whole Time（2000 Time Steps）:CACHE-QUADRANT



Whole Time（2000 Time Steps）:CACHE-SNC4

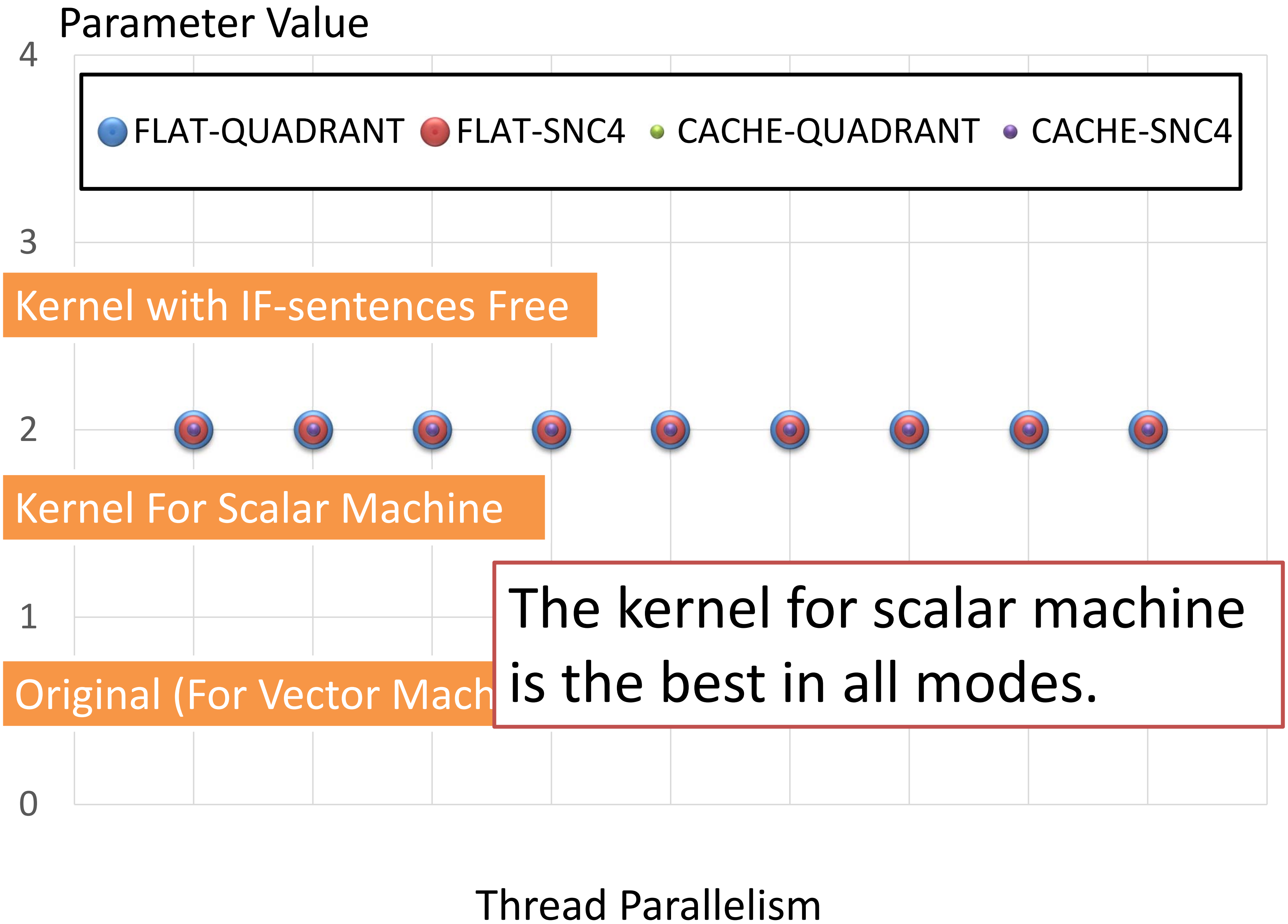


Whole Time（2000 Time Steps）:The Best



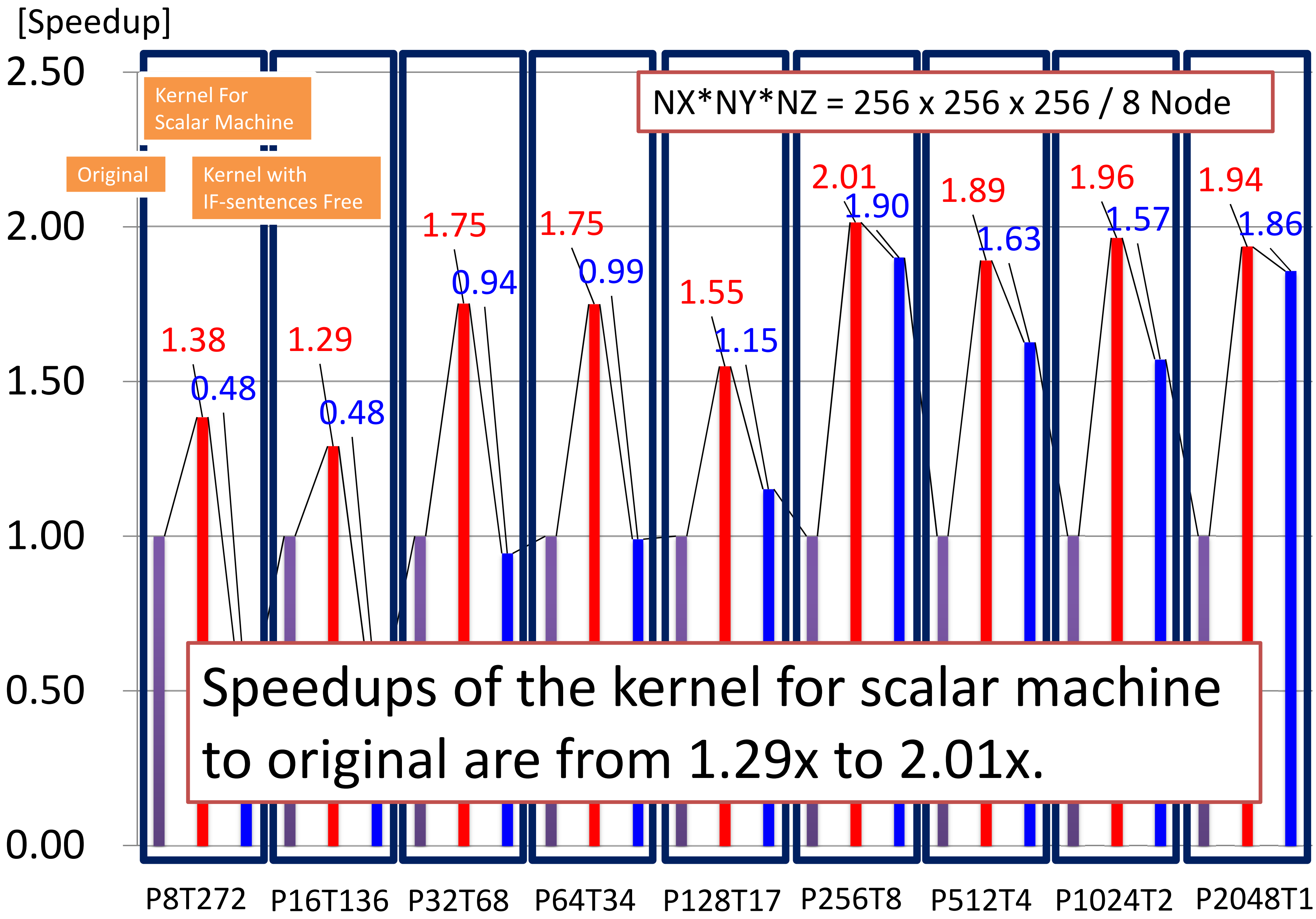
PARAMETER DISTRIBUTION

Parameter Distribution (update_stress) with 8 nodes



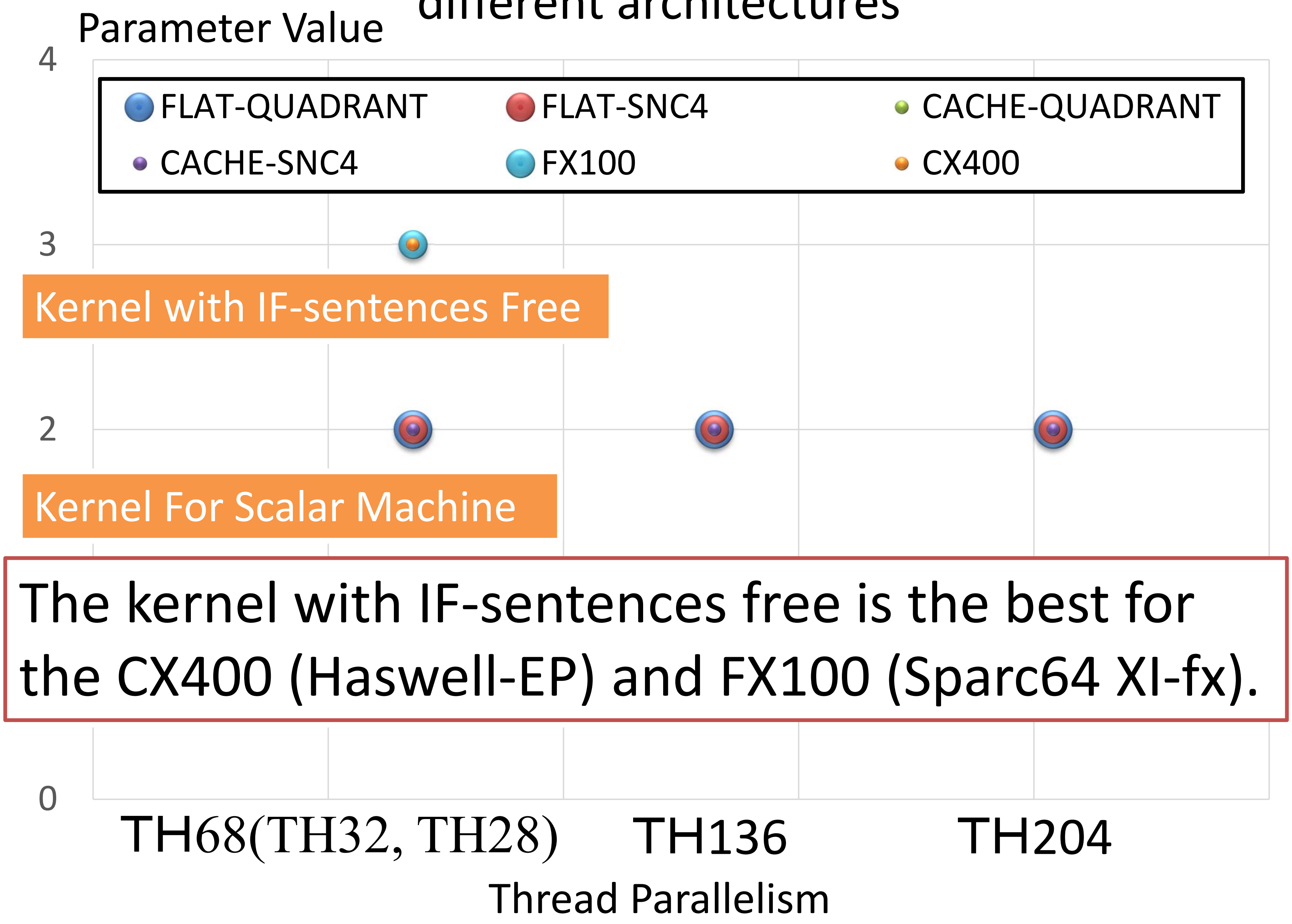
The kernel for scalar machine is the best in all modes.

Kernel speedup (update_stress):FLAT-QUADRANT

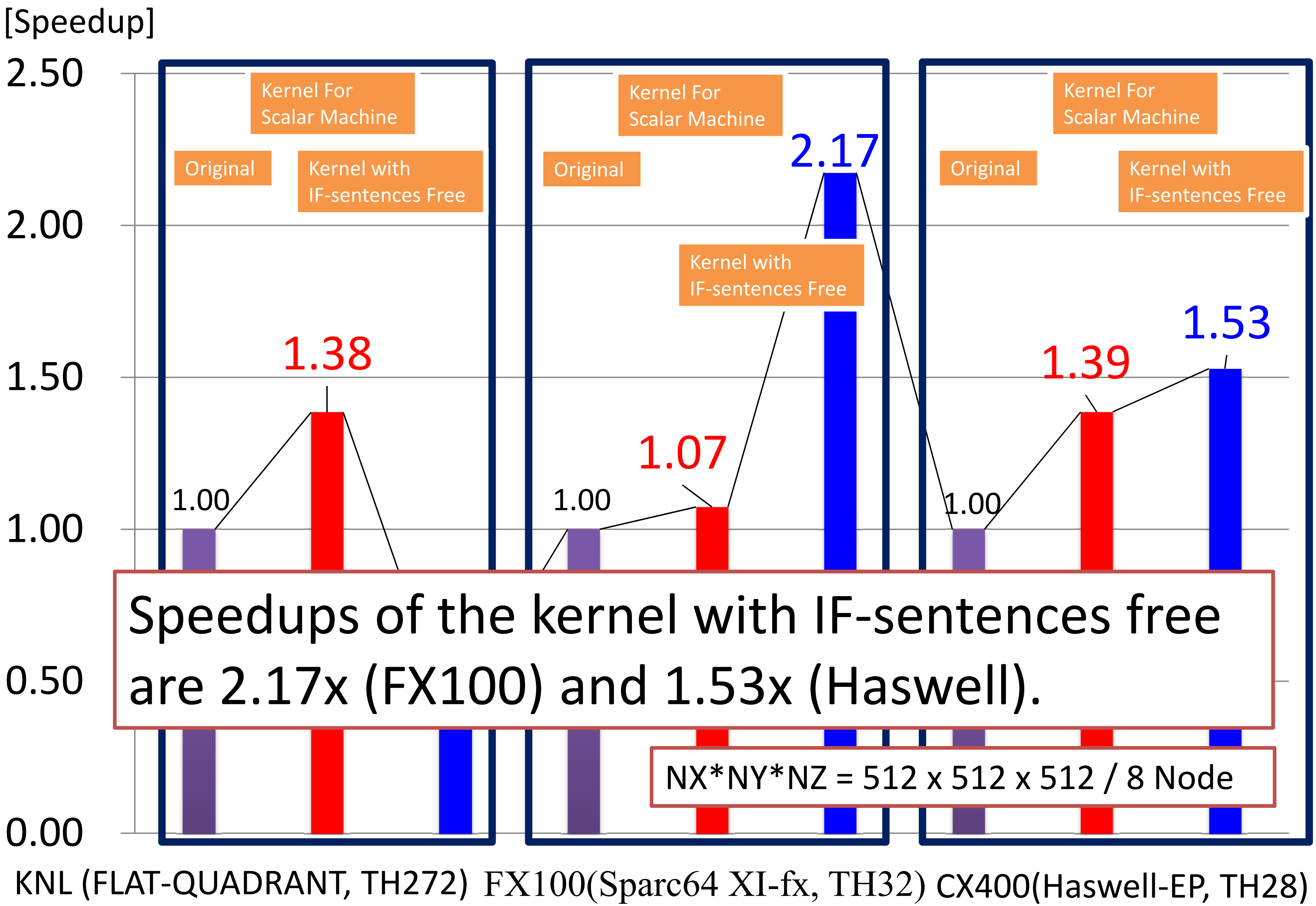


Speedups of the kernel for scalar machine to original are from 1.29x to 2.01x.

Parameter Distribution (update_stress) with 8 nodes in different architectures



Kernel speedup (update_stress, the Best)



Outline

- Background
- An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
- Performance Evaluation with KNL (Knights Landing)
- Conclusion

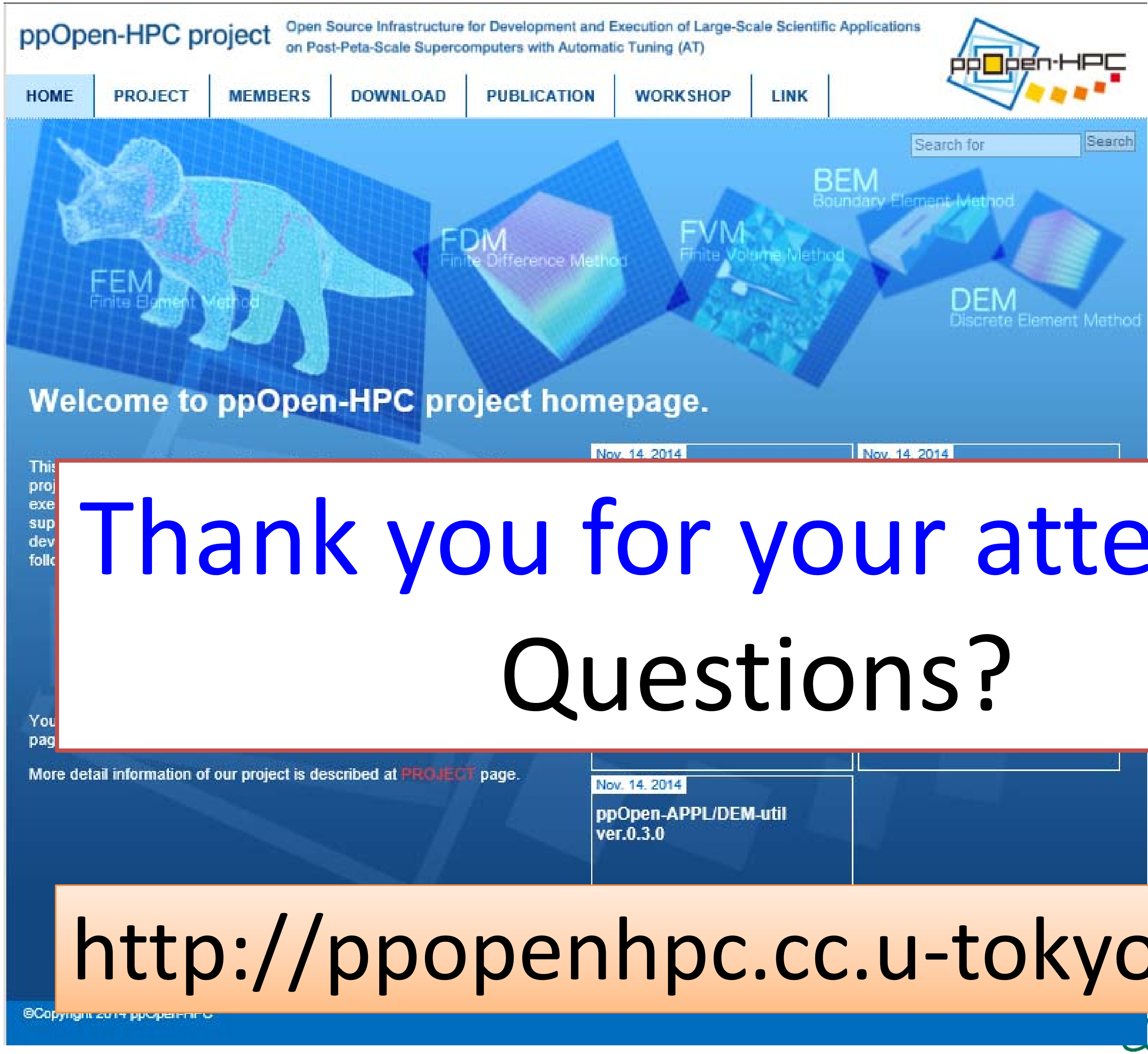
Conclusion Remarks

- We have checked performance of AT with an FDM code with KNL.
- CACHE mode is not slow with compared to FLAT mode.
- Sub-NUMA Clustering (SNC) mode in FLAT mode is somehow very slow.
  - We do not find the reason yet.
- We also observe big difference according to type of MPI/OpenMP executions by increasing the number of MPI processes.

  -> We still need to tune the type of MPI-OpenMP in advanced multicore CPU.

Future Work

- Find the reason that why FLAT-SNC4 is very slow.
- Find methodology (or novel implementations) to maximize performance w.r.t NUMA affinity in KNL.
- Implement a new AT facility (and model the performance) w.r.t NUMA affinity in KNL.



Thank you for your attention!

Questions?

http://ppopenhpc.cc.u-tokyo.ac.jp/