東京大学
THE UNIVERSITY OF TOKYO

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

Communication-Computation Overlapping in Parallel Iterative Solvers

Wisteria
BDEC-01

Kengo Nakajima
Information Technology Center
The University of Tokyo

Background

- Significant communication overhead of parallel preconditioned iterative solvers under massively parallel environment

Communication/Synchronization Avoiding/Reducing/Hiding for Parallel Preconditioned Krylov Iterative Methods

- Dot Products
  - Pipelined Methods [Ghysels et al. 2014]
  - Gropp's Algorithm
  - Utilization of Asynchronous Collective communications (e.g. MPI_Iallreduce) supported in MPI-3 for hiding such overhead [Horikoshi KN et al. IXPUG at HPC Asia 2022]
- SpMV
  - Overlapping of Comp. & Comm. (CC-Overlapping)
  - + Dynamic Loop Scheduling
  - Matrix Powers Kernels [Demmel et al. 2008]

```
Compute r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
     else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i-1)
    endif
    q(i)= [A]p(i)
    αi  = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |r|
end
```

- Dot Products
- Matrix Vector Multiplication
- Exercise

Hiding Overhead by Collective Comm. in Krylov Iterative Solvers

- Dot Products in Krylov Iterative Solvers
  - MPI_Allreduce: Collective Communications
  - Large overhead with many nodes
- Pipelined CG [Ghysels et al. 2014]
  - Utilization of asynchronous collective communications (e.g. MPI_Iallreduce) supported in MPI-3 for hiding such overhead.
  - Algorithm is kept, but order of computations is changed
  - [Reference] P. Ghysels et al., Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm, Parallel Computing 40, 2014
  - When I visited LBNL in September 2013, Dr. Ghysels asked me to evaluate his idea in my parallel multigrid solvers

4 Algorithms [Ghysels et al. 2014]

- Alg.1 Original Preconditioned CG
- Alg.2 Chronopoulos/Gear
  - 2 dot products are combined in a single reduction
- Alg.3 Pipelined CG (MPI_Iallreduce)
- Alg.4 Gropp's asynchronous CG (MPI_Iallreduce)

- Algorithm itself is not different from the original one
  - Recurrence Relations: 漸化式
  - Order of computation changed -> Rounding errors are propagated differently
    - Convergence may be affected (not happened in my case)
    - update of r= b-Ax needed at every 50 iterations (original paper)

Original Preconditioned CG (Alg.1)

Original Preconditioned CG (Alg.1)

1: $r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad p_0 := u_0$
2: **for** $i = 0, \ldots$ **do**
3: $\quad s := Ap_i$
4: $\quad \alpha := (r_i, u_i) / (s, p_i)$
5: $\quad x_{i+1} := x_i + \alpha p_i$
6: $\quad r_{i+1} := r_i - \alpha s$
7: $\quad u_{i+1} := M^{-1}r_{i+1}$
8: $\quad \beta := (r_{i+1}, u_{i+1}) / (r_i, u_i)$
9: $\quad p_{i+1} := u_{i+1} + \beta p_i$
10: **end for**

$$s_i = Ap_i$$
$$x_{i+1} = x_i + \alpha_i p_i$$
$$r_{i+1} = b - Ax_{i+1} = b - Ax_i - \alpha_i Ap_i$$
$$= r_i - \alpha_i Ap_i = r_i - \alpha_i s_i$$

Chronopoulos/Gear CG (Alg.2)

2 dot products are combined into a single reduction s-step algorithm with s=1

Chronopoulos/Gear CG (Alg.2)

1: $r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad w_0 := Au_0$
2: $\alpha_0 := (r_0, u_0) / (w_0, u_0); \quad \beta_0 := 0; \quad \gamma_0 := (r_0, u_0)$
3: **for** $i = 0, \ldots$ **do**
4: $\quad p_i := u_i + \beta_i p_{i-1}$
5: $\quad s_i := w_i + \beta_i s_{i-1}$
6: $\quad x_{i+1} := x_i + \alpha_i p_i$
7: $\quad r_{i+1} := r_i - \alpha_i s_i$
8: $\quad u_{i+1} := M^{-1} r_{i+1}$
9: $\quad w_{i+1} := Au_{i+1}$
10: $\quad \gamma_{i+1} := (r_{i+1}, u_{i+1})$
11: $\quad \delta := (w_{i+1}, u_{i+1})$
12: $\quad \beta_{i+1} := \gamma_{i+1} / \gamma_i$
13: $\quad \alpha_{i+1} := \gamma_{i+1} / (\delta - \beta_{i+1}\gamma_{i+1}/\alpha_i)$
14: **end for**

$$s_i = Au_i + \beta_i s_{i-1},\ p_i = u_i + \beta_i p_i \Leftrightarrow s_i = Ap_i$$

$$x_{i+1} = x_i + \alpha_i p_i$$

$$r_{i+1} = b - Ax_{i+1} = b - Ax_i - \alpha_i Ap_i$$
$$= r_i - \alpha_i Ap_i = r_i - \alpha_i s_i$$

- 2 dot products combined

- $s_i = Ap_i$ is not computed explicitly: by recurrence

Pipelined Chronopoulos/Gear (No Preconditioning)

Pipelined Chronopoulos/Gear (No Precond.)

1: $r_0 := b - Ax_0$; $w_0 := Ar_0$
2: **for** $i = 0, \ldots$ **do**
3: $\gamma_i := (r_i, r_i)$
4: $\delta := (w_i, r_i)$
5: $q_i := Aw_i$
6: **if** $i > 0$ **then**
7: $\beta_i := \gamma_i / \gamma_{i-1}$; $\alpha_i := \gamma_i / (\delta - \beta_i \gamma_i / \alpha_{i-1})$
8: **else**
9: $\beta_i := 0$; $\alpha_i := \gamma_i / \delta$
10: **end if**
11: $z_i := q_i + \beta_i z_{i-1}$
12: $s_i := w_i + \beta_i s_{i-1}$
13: $p_i := r_i + \beta_i p_{i-1}$
14: $x_{i+1} := x_i + \alpha_i p_i$
15: $r_{i+1} := r_i - \alpha_i s_i$
16: $w_{i+1} := w_i - \alpha_i z_i$
17: **end for**

- Global synchronization of dot products are overlapped with SpMV

$$u_i = r_i,\ w_i = Au_i = Ar_i$$

$$Ar_{i+1} = Ar_i - \alpha_i As_i \Rightarrow w_{i+1} = w_i - \alpha_i As_i$$

$$As_i = Aw_i + \beta_i As_{i-1} \Rightarrow z_i \left(= As_i = A^2 p_i\right) = Aw_i + \beta_i z_{i-1}$$

$$q_i = Aw_i = A^2 r_i$$

$$= r_i - \alpha_i Ap_i = r_i - \alpha_i s_i$$

Preconditioned Pipelined CG (Alg.3)

Preconditioned Pipelined CG (Alg.3)

1: $r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad w_0 := Au_0$
2: **for** $i = 0, \ldots$ **do**
3: $\quad \gamma_i := (r_i, u_i)$
4: $\quad \delta := (w_i, u_i)$
5: $\quad m_i := M^{-1}w_i$
6: $\quad n_i := Am_i$
7: $\quad$ **if** $i > 0$ **then**
8: $\quad\quad \beta_i := \gamma_i/\gamma_{i-1}; \quad \alpha_i := \gamma_i/\left(\delta - \beta_i\gamma_i/\alpha_{i-1}\right)$
9: $\quad$ **else**
10: $\quad\quad \beta_i := 0; \quad \alpha_i := \gamma_i/\delta$
11: $\quad$ **end if**
12: $\quad z_i := n_i + \beta_i z_{i-1}$
13: $\quad q_i := m_i + \beta_i q_{i-1}$
14: $\quad s_i := w_i + \beta_i s_{i-1}$
15: $\quad p_i := u_i + \beta_i p_{i-1}$
16: $\quad x_{i+1} := x_i + \alpha_i p_i$
17: $\quad r_{i+1} := r_i - \alpha_i s_i$
18: $\quad u_{i+1} := u_i - \alpha_i q_i$
19: $\quad w_{i+1} := w_i - \alpha_i z_i$
20: **end for**

- Global synchronization of dot products are overlapped with SpMV and Preconditioning

$$\gamma_i = (u_i, u_i)_M = (Mu_i, u_i) = (r_i, u_i)$$

$$\delta_i = \left(M^{-1}Au_i, u_i\right)_M = (Au_i, u_i) = (w_i, u_i)$$

$$M^{-1}r_{i+1} = M^{-1}r_i - \alpha_i M^{-1}s_i \Rightarrow u_{i+1} = u_i - \alpha_i q_i \quad \left(q_i = M^{-1}s_i\right)$$

$$M^{-1}s_{i+1} = M^{-1}w_i + \beta_i M^{-1}s_i \Rightarrow q_{i+1} = M^{-1}w_i + \beta_i q_i$$

$$Au_{i+1} = Au_i - \alpha_i Aq_i \Rightarrow w_{i+1} = w_i - \alpha_i Aq_i$$

$$Aq_i = AM^{-1}w_i + \beta_i Aq_{i-1} \Rightarrow z_i = Am_i + \beta_i z_{i-1}$$

$$\left(m_i = M^{-1}w_i = M^{-1}Au_i = M^{-1}AM^{-1}r_i,\ z_i = Aq_i\right)$$

Gropp's Asynchronous CG (Alg.4) Smaller Computations than Alg.3

Gropp's Asynchronous CG (Alg.4)

1: $r_0 := b - Ax_0$; $u_0 := M^{-1}r_0$; $p_0 := u_0$; $s_0 := Ap_0$; $\gamma_0 := (r_0, u_0)$
2: **for** $i = 0, \ldots$ **do**
3: $\delta := (p_i, s_i)$
4: $q_i := M^{-1}s_i$
5: $\alpha_i := \gamma_i/\delta$
6: $x_{i+1} := x_i + \alpha_i p_i$
7: $r_{i+1} := r_i - \alpha_i s_i$
8: $u_{i+1} := u_i - \alpha_i q_i$
9: $\gamma_{i+1} := (r_{i+1}, u_{i+1})$
10: $w_{i+1} := Au_{i+1}$
11: $\beta_{i+1} := \gamma_{i+1}/\gamma_i$
12: $p_{i+1} := u_{i+1} + \beta_{i+1}p_i$
13: $s_{i+1} := w_{i+1} + \beta_{i+1}s_i$
14: **end for**

- Definition of δ is different from that of Alg.3
- Global synchronization of dot products are overlapped with SpMV and Preconditioning

W. Gropp, Update on Libraries for Blue Waters.
http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf
Presentation Material (not a paper, article)

Implementation (Alg.4)

```
!C
!C +--------------+
!C | Delta= (p,s) |
!C +--------------+
!C===
      DL0= 0.d0
!$omp parallel do private(i) reduction(+:DL0)
      do i= 1, 3*N
        DL0= DL0 + P(i)*S(i)
      enddo
      call MPI_Iallreduce (DL0, Delta, 1, MPI_DOUBLE_PRECISION,           &
     &                     MPI_SUM, MPI_COMM_WORLD, req1, ierr)
!C===

!C
!C +----------------+
!C | {q}= [Minv]{s} |
!C +----------------+
!C===

 (Preconditioing)

!C===

      call MPI_Wait (req1, sta1, ierr)
```

Gropp's Asynchronous CG (Alg.4)

1: $r_0 := b - Ax_0$; $u_0 := M^{-1}r_0$; $p_0 := u_0$; $s_0 := Ap_0$; $\gamma_0 := (r_0, u_0)$
2: **for** $i = 0, \ldots$ **do**
3: $\delta := (p_i, s_i)$
4: $q_i := M^{-1}s_i$
5: $\alpha_i := \gamma_i/\delta$
6: $x_{i+1} := x_i + \alpha_i p_i$
7: $r_{i+1} := r_i - \alpha_i s_i$
8: $u_{i+1} := u_i - \alpha_i q_i$
9: $\gamma_{i+1} := (r_{i+1}, u_{i+1})$
10: $w_{i+1} := Au_{i+1}$
11: $\beta_{i+1} := \gamma_{i+1}/\gamma_i$
12: $p_{i+1} := u_{i+1} + \beta_{i+1}p_i$
13: $s_{i+1} := w_{i+1} + \beta_{i+1}s_i$
14: **end for**

Pipelined CR (Conjugate Residuals)

Preconditioned Pipelined CR

1: $r_0 := b - Ax_0$; $u_0 := M^{-1}r_0$; $w_0 := Au_0$
2: **for** $i = 0, \ldots$ **do**
3: $m_i := M^{-1}w_i$
4: $\gamma_i := (w_i, u_i)$
5: $\delta := (m_i, w_i)$
6: $n_i := Am_i$
7: **if** $i > 0$ **then**
8: $\beta_i := \gamma_i/\gamma_{i-1}$; $\alpha_i := \gamma_i/(\delta - \beta_i\gamma_i/\alpha_{i-1})$
9: **else**
10: $\beta_i := 0$; $\alpha_i := \gamma_i/\delta$
11: **end if**
12: $z_i := n_i + \beta_i z_{i-1}$
13: $q_i := m_i + \beta_i q_{i-1}$
14: $p_i := u_i + \beta_i p_{i-1}$
15: $x_{i+1} := x_i + \alpha_i p_i$
16: $u_{i+1} := u_i - \alpha_i q_i$
17: $w_{i+1} := w_i - \alpha_i z_i$
18: **end for**

- Global synchronization of dot products are overlapped with SpMV

Amount of Computations

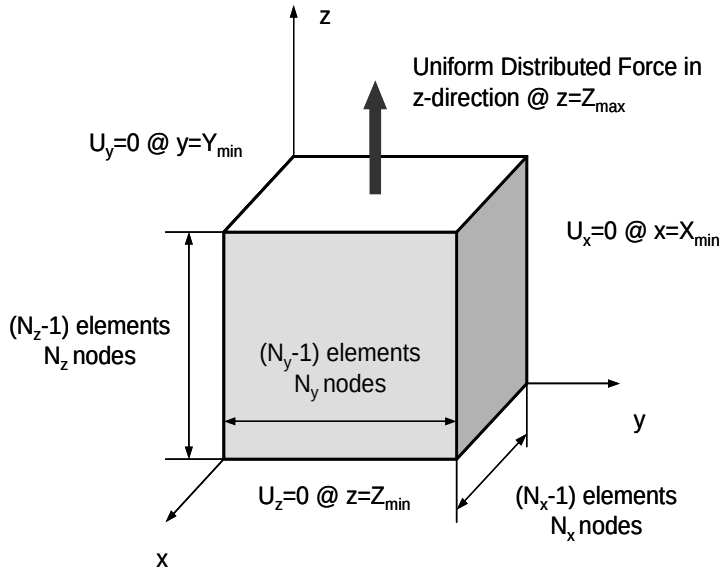
DAXPY could very smaller than (sophisticated) preconditioning

		SpMV	Precond.	Dot Prod.	DAXPY
1	Original CG	1	1	2+1	3
2	Chronopoulos/ Gear	1	1	2+1	4
3	Pipelined CG	1	1	2+1	8
4	Grppp's Algorithm	1	1	2+1	5
	Pipelined CR	1	1	2+1	6

+1 for residual norm

GeoFEM/Cube

- Parallel FEM Code (& Benchmarks)
- 3D-Static-Elastic-Linear (Solid Mechanics)
- Performance of Parallel Precond. Iterative Solvers
  - 3D Tri-linear Elements
  - SPD matrices: CG solver
  - Fortran90+MPI+OpenMP
  - Distributed Data Structure
  - Localized SGS Preconditioning
    - Symmetric Gauss-Seidel, Block Jacobi
    - Additive Schwartz Domain Decomposition
  - Reordering by CM-RCM: RCM
  - MPI, OpenMP, OpenMP/MPI Hybrid



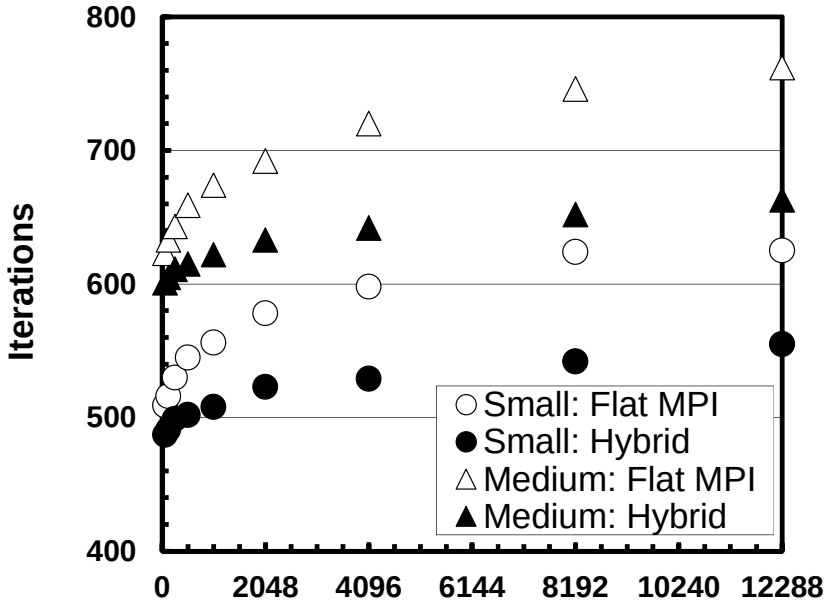
Results on Reedbush-U (RDB)

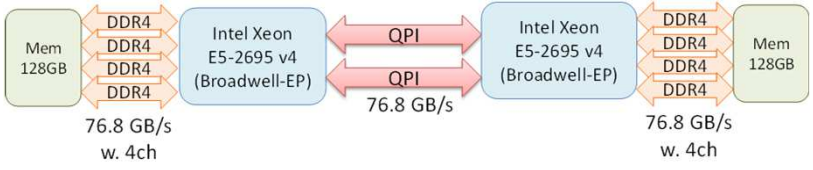
- 4 Types of Algorithms
  - Alg.1 Original Preconditioned CG
  - Alg.2 Chronopoulos/Gear
  - Alg.3 Pipelined CG (MPI_Allreduce, MPI_Iallreduce,)
  - Alg.4 Gropp's asynchronous CG (MPI_Allreduce, MPI_Iallreduce)

- Flat MPI, OpenMP/MPI Hybrid with Reordering
- Platform
  - Integrated Supercomputer System for Data Analyses & Scientific Simulations (Reedbush-U)
  - Intel Broadwell-EP 18 cores x 2 sockets x 420 nodes
  - Intel Fortran + Intel MPI
  - 16 of 18 cores/socket, up to 384 nodes (= 768 sockets, 12,288 cores)

Results: Number of Iterations

- Strong Scaling
- Small
  - 256×128×144 nodes (=4,718,592)
  - 14,155,776 DOF
  - at 384 nodes (12,288 cores)
    - 8×8×6＝384 nodes/core
    - 1,152 DOF/core
- Medium
  - 256×128×288 nodes (=9,437,184)
  - 28,311,552自由度



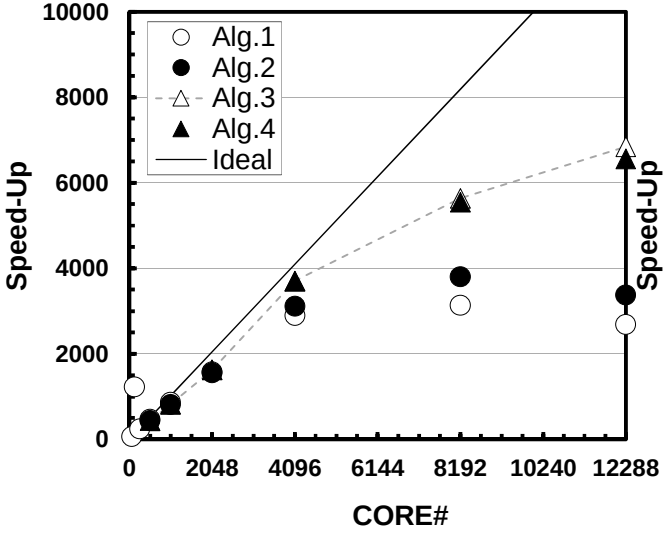


Results: Speed-Up

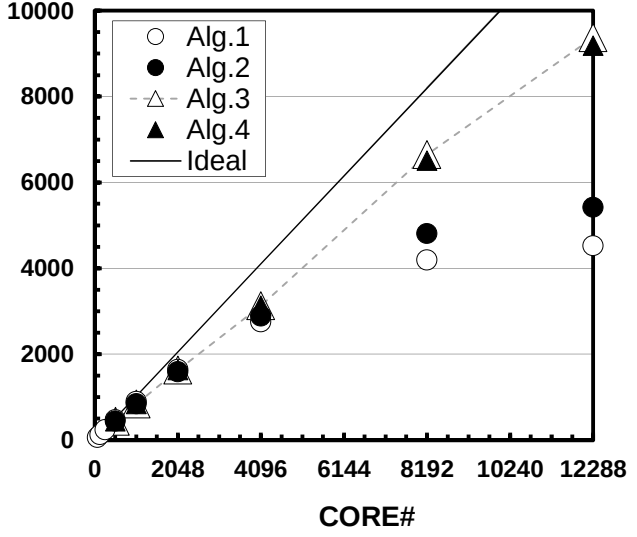
Performance of 2 nodes of Flat MPI = 64.0 (4 sockets, 64 cores)

Alg.1	Original PCG
Alg.2	Chronopoulos/Gear
Alg.3	Pipelined CG
Alg.4	Gropp's CG

Small



Medium

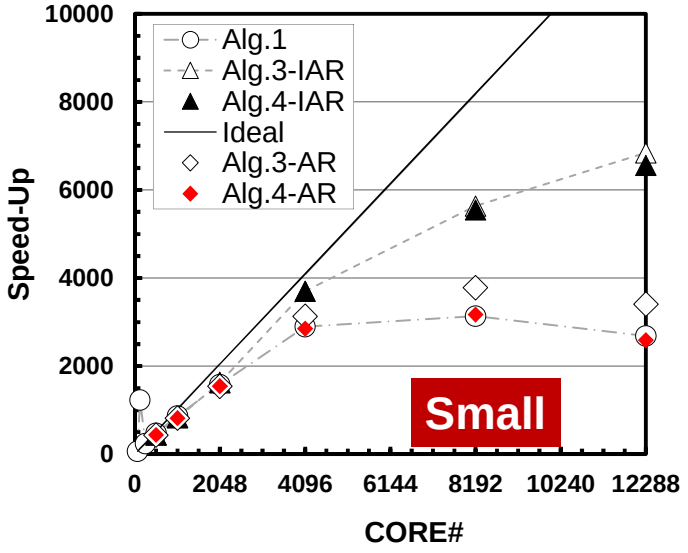


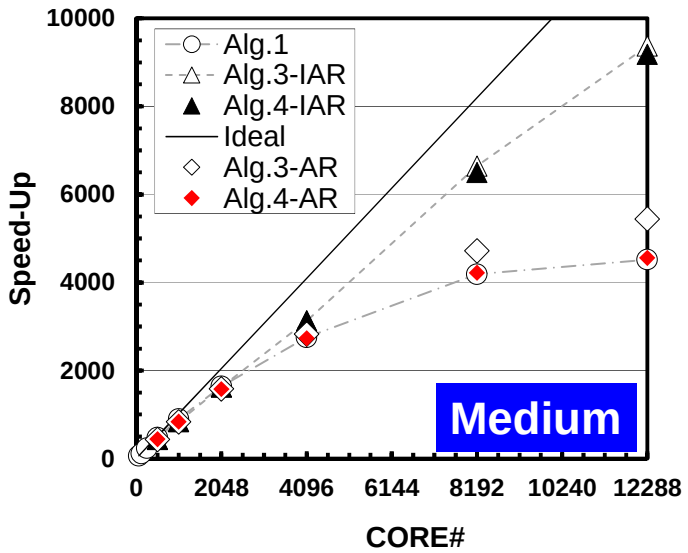
Allreduce vs. Iallreduce for Hybrid

Performance of 2 nodes of Flat MPI = 64.0 (4 sockets, 64 cores)

IAR: MPI_Iallreduce
AR : MPI_Allreduce

Alg.1	Original PCG
Alg.3	Pipelined CG
Alg.4	Gropp's CG





Current Status (March 2019)

- Results were obtained in July 2016, just after installment of Reedbush-U (RBU) Cluster
- Intel® Fortran Compiler/Intel® MPI Library have been improved. Performance of MPI_Allreduce is much better now
  - Therefore these results are not reproduced now ☺
- Same Situations on Oakforest-PACS (OFP)
  - Generally, it is difficult to observe effects of asynchronous collective communications without HW support (e.g. Communication Assist Cores on Fujitsu's FX100)
- In Intel® MPI Library 2019, performance of Asynchronous Progress Thread has been significantly improved.
  - Therefore, Comm.-Comp. Overlapping will be improved by Asynchronous Progress

What to be done for utilization of “Asynchronous Progress”

- (Basically) No changes in the Source Code
- Just use Intel® MPI Library 2019 at Runtime
  - No differences are found in compiling by 2018 or 2019
- Just add statements which activate “Asynchronous Progress”
  - Example of 2T/core
  - 64 of 68 cores on each node are used for computations
  - Unused cores are like “communication assist cores”

```
export I_MPI_FABRICS=shm:ofi
export I_MPI_ASYNC_PROGRESS=on
export I_MPI_ASYNC_PROGRESS_PIN="0,1,66,67,68,69,134,135"
```

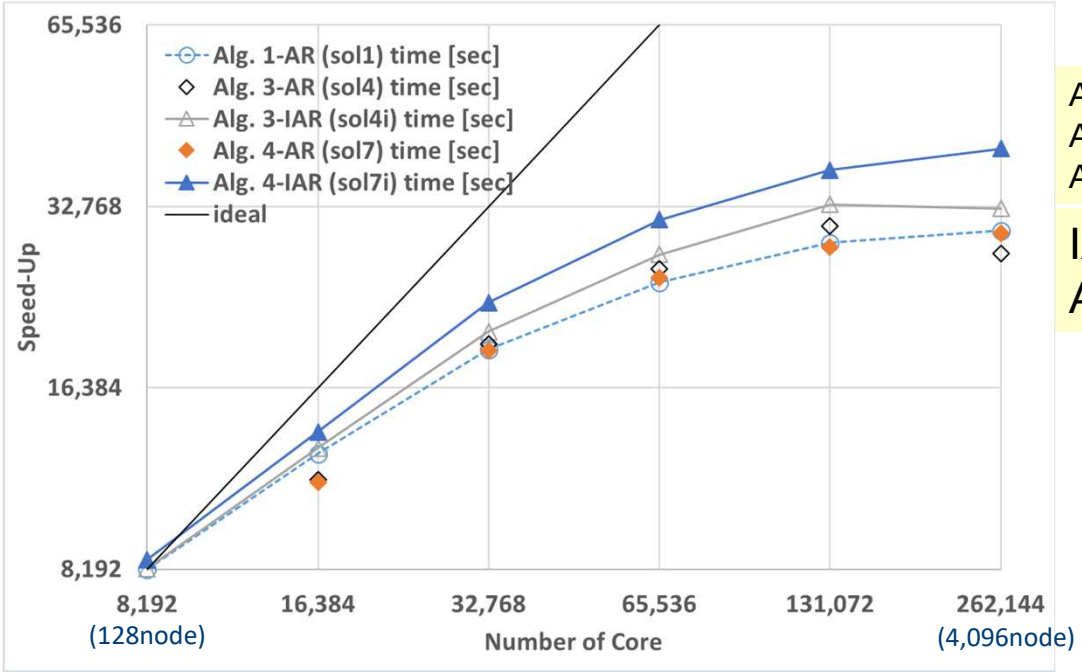
Effect of I_MPI_ASYNC_PROGRESS Intel Parallel Studio 2019 (March 2019)

128^3 nodes (6,291,456 DOFs)
64 nodes of OFP, HB 16x8 (2T/core), Time for ICCG
Fastest case in 5 runs, Flat-MCDRAM Only, Time (sec)

	2019 ASYNC_PROGRESS =on	2019 ASYNC_PROGRESS =off	2018	2019 ASYNC_PROGRESS=on 128 nodes
Alg.1	1.861	2.010	1.987	**1.764**
Alg.2	1.741	1.907	1.912	1.627
Alg.3-AR	1.772	1.944	1.953	1.647
Alg.3-IAR	**1.715**	1.989	**2.366**	1.558
Alg.4-AR	1.834	1.989	1.993	1.712
Alg.4-IAR	**1.618**	2.021	**2.204**	**1.434**

Effect of Asynchronous Progress (MPI_Iallreduce)

Strong Scaling from 128 to 4096 node on Oakforest-PACS. MCDRAM only.
100,663,296 DOFs (3x512x256x256). Fastest case in 5 runs. Per iter. Perf.
Performance of 128 node = 8,192 (128 node * 64 core)



Alg.1	Original PCG
Alg.3	Pipelined CG (1 sync)
Alg.4	Gropp's CG (2 sync)

IAR: MPI_Iallreduce
AR : MPI_Allreduce

IAR vs. AR

Speedup	Alg. 3	Alg. 4
8,192	1.03	1.06
16,384	1.13	1.21
32,768	1.05	1.20
65,536	1.06	1.25
131,072	1.08	1.34
262,144	1.19	1.38

Current Situation (January 2024)

- MPI_Iallreduce is available on Wisteria/BDEC-01 (Odyssey), while its implementation is same as that of MPI_Allreduce.
  - Therefore, performance of MPI_Iallreduce and MPI_Allrecuce is (almost) same
- Asynchronous Progress on Intel Cluster
  - We re-evaluated the cases in the previous page in February 2022 (2 years later).
    - No differences between MPI_Iallreduce & MPI_Allreduce
    - Performance of MPI has been much improved in 2 years
  - After that, we have no chances for evaluating the code on big clusters with Intel Xeon CPUs

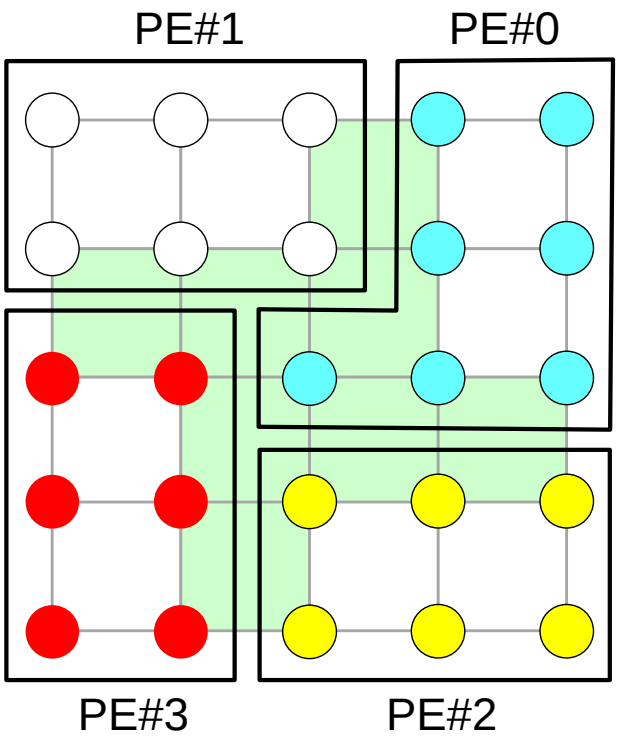
- Dot Products
- **Matrix Vector Multiplication**
- Exercise

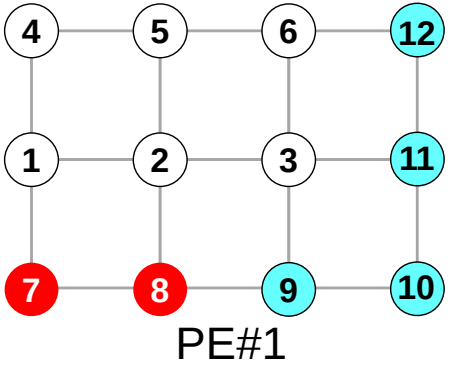
Communication Avoiding/Reducing Algorithms for Sparse Linear Solvers utilizing Matrix Powers Kernel

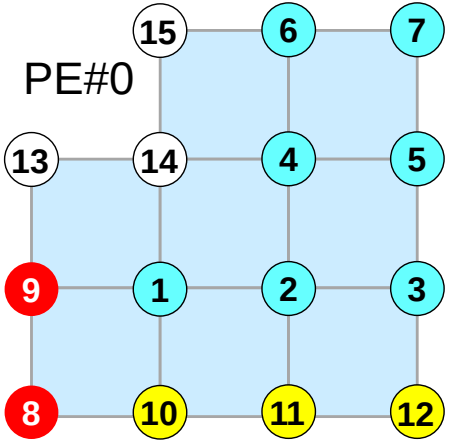
- Matrix Powers Kernel: Ax, A^2x, A^3x ...
- Krylov Iterative Method without Preconditioning [Demmel et al. 2008]
- *s-step* method
  - Just one P2P communication for each Mat-Vec during *s* iterations. Convergence may become unstable for large *s*.
- Communication Avoiding ILU0 (CA-ILU0) [Moufawad & Grigori, 2013]
  - First attempt to CA preconditioning
  - Nested dissection reordering for limited geometries (2D FDM)
- Generally, it is difficult to apply Matrix Powers Kernel to preconditioned iterative solvers

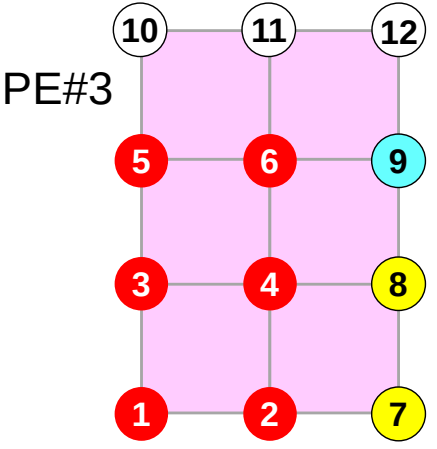
- Dot Products
- **Matrix Vector Multiplication: CC-Overlapping**
- Exercise

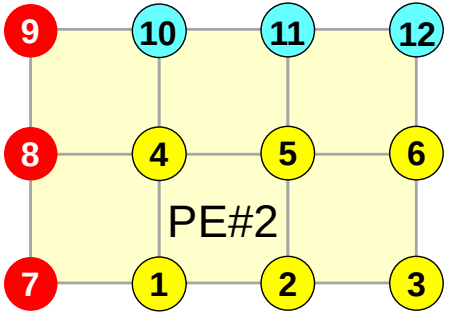
Parallel FEM: Distributed Data Structure







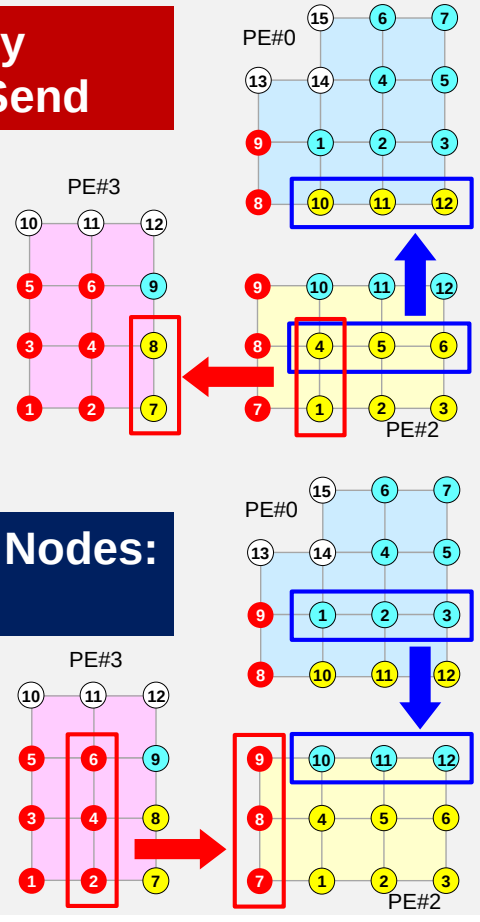




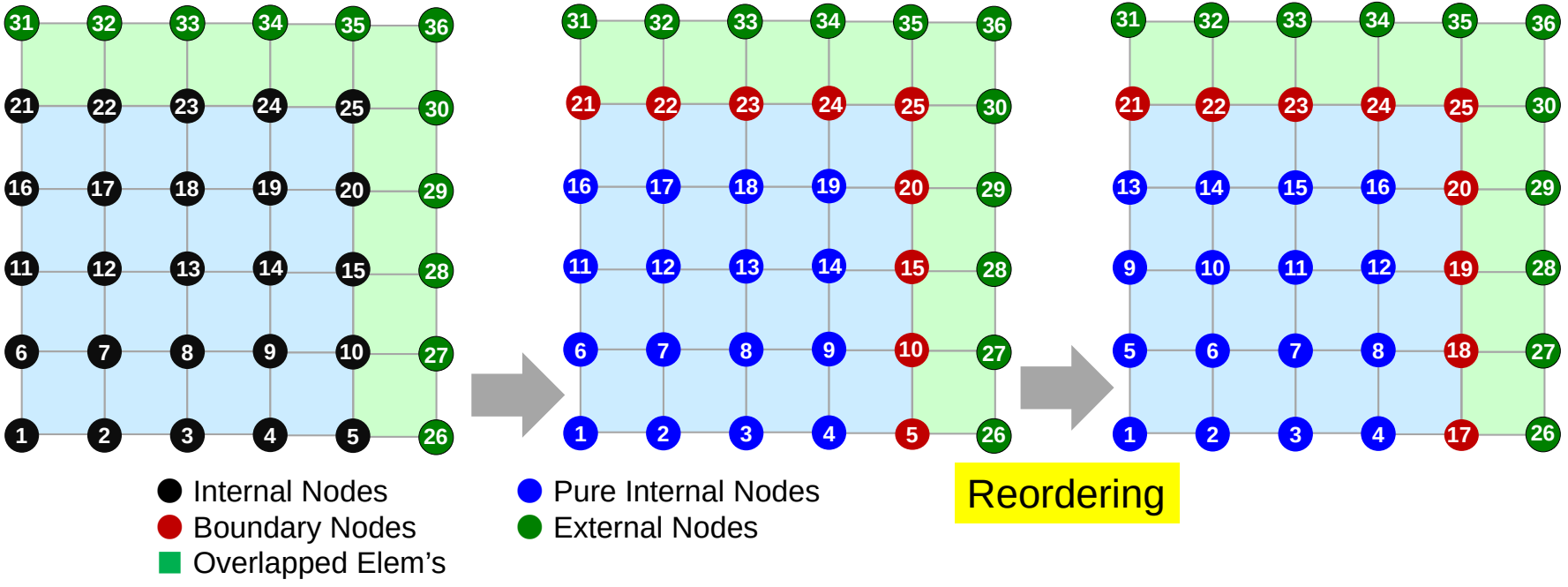
```
      allocate (stat_send(MPI_STATUS_SIZE,2*NEIBPETOT))
      allocate (request_send(2*NEIBPETOT))

!$omp parallel do private (neib,k,kk)
      do neib= 1, NEIBPETOT
        do k= export_index(neib-1)+1, export_index(neib)
          kk= export_item(k)
          SENDbuf(k)= W(kk,P)
        enddo
      enddo

      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_Isend (SENDbuf(is), len_s, MPI_DOUBLE_PRECISION,      &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,               &
     &                  request_send(neib), ierr)
      enddo

      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_Irecv (W(ir+N,P), len_r, MPI_DOUBLE_PRECISION,        &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,               &
     &                  request_send(neib+NEIBPETOT), ierr)
      enddo

      call MPI_Waitall (2*NEIBPETOT, request_send, stat_send, ierr)
```

Boundary Nodes: Send

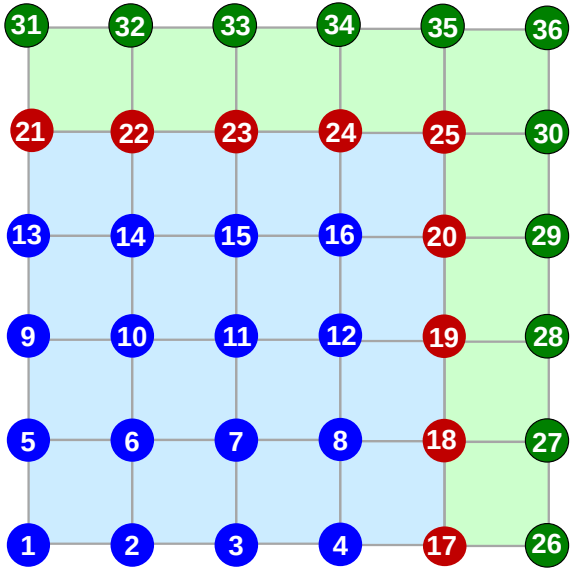
External Nodes: Recv



Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]



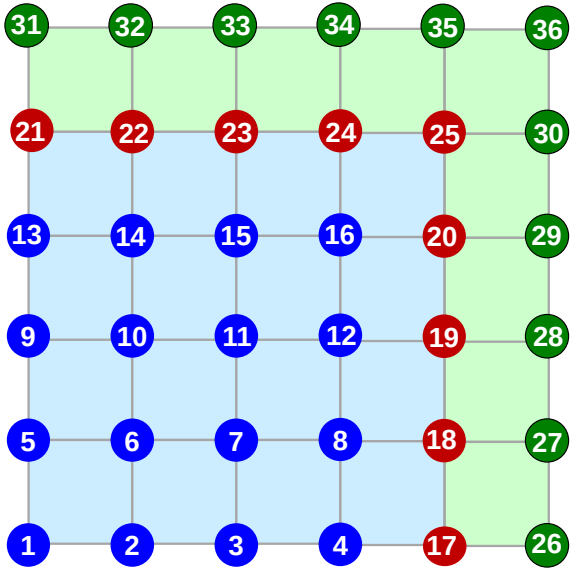
Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]



Original

```
call MPI_Isend
call MPI_Irecv
call MPI_Waitall

!$omp parallel do (…)
  do i= 1, N
    (computing)
  enddo
```

Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]



CC-Overlapping: Static Loop Scheduling

```
call MPI_Isend
call MPI_Irecv
```

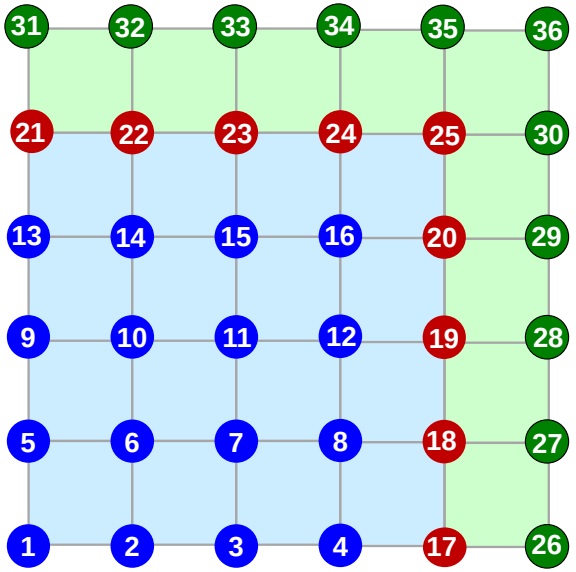
```
!$omp parallel do (…)
  do i= 1, Ninn
    (computing)
  enddo
call MPI_Waitall
```

Pure Internal Nodes

```
!$omp parallel do (…)
  do i= Ninn+1, Nall
    (computing)
  enddo
```

Boundary Nodes

Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]




```
$omp parallel (…)
$omp master
  call MPI_Isend
  call MPI_Irecv
  call MPI_Waitall
$omp end master
```

Master
Halo Comm.

```
!$omp do schedule (dynamic,200)
  do i= 1, Ninn
    (computing)
  enddo
```

Dynamic
Pure Internal Nodes

```
!$omp do
  do i= Ninn+1, Nall
    (computing)
  enddo
!$omp end parallel
```

Static
Boundary Nodes

CC-Overlapping: Dynamic Scheduling

Dynamic Loop Scheduling (1/2)

- CC-Overlapping in HALO Exchange
  - HALO exchange including sending buffer copy is done by the *master* thread
  - Dynamic loop scheduling is applied to the computations for *pure* internal nodes/meshes
  - The computations for pure internal nodes/meshes starts without master thread, while the master thread is doing communications.
  - The master thread can join the computations for pure internal nodes/meshes after completion of the communication.
- There are four different loop scheduling types (*kinds*) (*static, dynamic, guided, auto*), and the optional parameter (*chunk*) must be a positive integer:

C: #pragma omp parallel for schedule (kind [, chunk])
Fortran: !$omp parallel do schedule (kind [,chunk])

Dynamic Loop Scheduling

- “dynamic”
- “!$omp master～!$omp end master”

```
!$omp parallel private (...)

!$omp master              Communication is done by the master thread (#0)
      call MPI_Irecv (...)
      call MPI_Isend (...)
      call MPI_WAITALL (...)
!$omp end master

!C                          The master thread can join computing of internal
!C-- Pure Internal Nodes    nodes after the completion of communication

!$omp do schedule (dynamic,200)    Chunk Size= 200
      do j= 1, Ninn
        (…)
      enddo
!C
!C-- Boundary Nodes          Computing for boundary nodes are by all threads

!$omp do                     default: !$omp do schedule (static)
      do j= Ninn+1, N
        (…)
      enddo

!$omp end parallel
```

Ina, T., Asahi, Y., Idomura, Y., Development of optimization of stencil calculation on Tera-flops many-core architecture, IPSJ SIG Technical Reports 2015-HPC-152-10, 2015 (in Japanese)

OpenMP: Loop Scheduling

```
!$omp parallel do schedule (kind, [chunk])
!$omp do schedule (kind, [chunk])
```

```
#pragma parallel for schedule (kind, [chunk])
#pragma for schedule (kind, [chunk])
```

kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the OMP_SCHEDULE environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

Dynamic Loop Scheduling (2/2)

- The kind “*static*” is the default, and loops are divided into equal-sized chunks (or as equal as possible)
  - By default, chunk size is loop-count/number-of-threads.

- If the kind “*dynamic*” is applied, the internal work queue is used for giving a chunk-sized block of loop iterations to each thread.
  - When operations of a thread have finished, that retrieves the next block of loop iterations from the top of the work queue.
  - The chunk size is equal to 1 by default.
  - Extra overhead for scheduling is involved for this type of scheduling.

Dynamic Loop Scheduling (2/2)

- The kind “*static*” is the default, and loops are divided into equal-sized chunks (or as equal as possible)
  - By default, chunk size is loop-count/number-of-threads.

- If the kind “*dynamic*” is applied, the internal work queue is used for giving a chunk-sized block of loop iterations to each thread.
  - When operations of a thread have finished, that retrieves the next block of loop iterations from the top of the work queue.
  - The chunk size is equal to 1 by default.
  - Extra overhead for scheduling is involved for this type of scheduling.

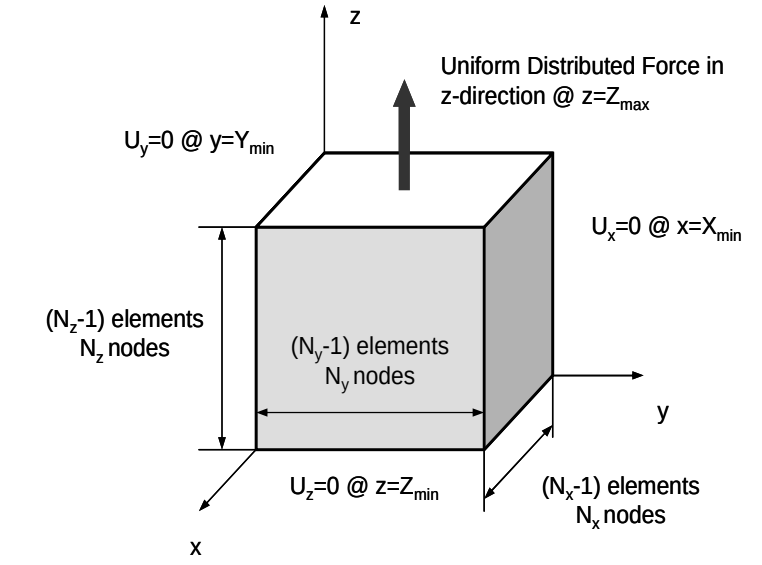
Chunk Size for Dynamic Loop Scheduling

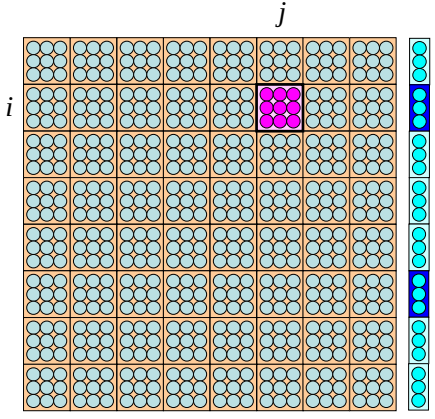
- Loop Cycle for checking completion of works by the master thread
- Small Chunk Size
  - ✓ Good for efficient utilization of resources
  - ✓ Overhead
- Large Chunk Size
  - ✓ Small Overhead, but possibility of long idle time of the mater thread
- Optimum Chunk Size
  - ✓ Problem Size, Hardware, Compiler etc.
  - ✓ ~O(10^3) for Odyssey: Problems should be large enough

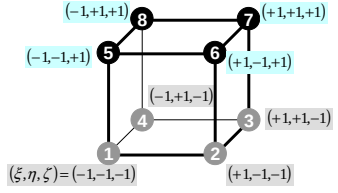
```
!C                              The master thread can join computing of internal
!C-- Pure Internal Nodes        nodes after the completion of communication

!$omp do schedule (dynamic,200)     Chunk Size= 200
      do j= 1, Ninn
        (…)
      enddo
```

GeoFEM/Cube

- Parallel FEM Code (& Benchmarks)
- 3D-Static-Elastic-Linear (Solid Mechanics)
- Performance of Parallel Preconditioned Iterative Solvers
  - 3D Tri-linear Hexahedral Elements
  - Block Diagonal LU + CG
    - Dynamic Loop Scheduling in SpMV
  - Fortran90+MPI+OpenMP
  - Distributed Data Structure
  - MPI, OpenMP, OpenMP/MPI Hybrid
  - Block CRS Format
- Odyssey (up to 2,048 nodes)







$$\begin{bmatrix} a_{i_e j_e 11} & a_{i_e j_e 12} & a_{i_e j_e 13} \\ a_{i_e j_e 21} & a_{i_e j_e 22} & a_{i_e j_e 23} \\ a_{i_e j_e 31} & a_{i_e j_e 32} & a_{i_e j_e 33} \end{bmatrix} (i_e, j_e = 1 \ldots 8)$$

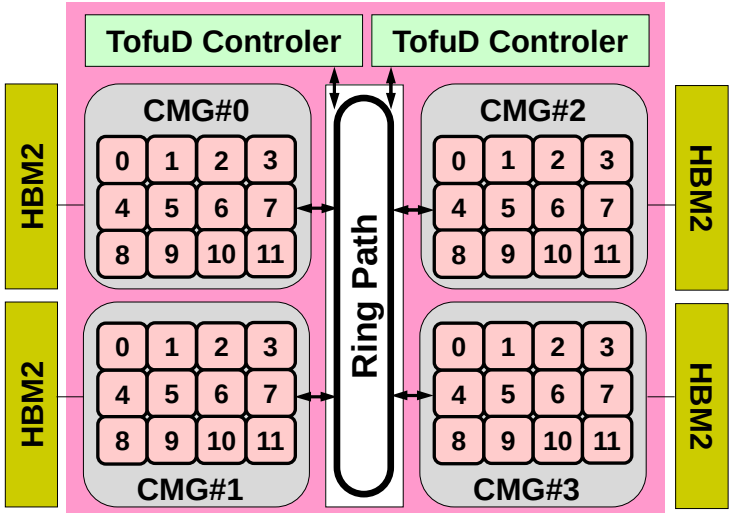
Computational Environment

- Wisteria/BDEC-01 (Odyssey)
  - Fujitsu PRIMEHPC FX1000, A64FX
    - Commercial version of “Fugaku”
  - 25.9 PF, 7,680 nodes (20 racks)
  - #40 TOP500 (May 2024)

Name of System	Wisteria/BDEC-01 (Odyssey)
CPU	Fujitsu A64FX（2.2GHz）
Core #	48
Peak Performance (GFLOPS)	**3,379**
Memory Capacity (GB)	32
Memory Bandwidth (GB/sec) STREAM Triad	**840+**
Compiler	Fujitsu FCC
Network	Tofu-D, 6D Torus

A64FX Processor

Name	A64FX
Core #	48-Cores + 2/4-Assistant Cores
Frequency	2.2 GHz
Peak Performance	3.3792 TFLOPS
Memory Capacity	32 GB
Peak Memory Bandwidth	1,024 GB/s

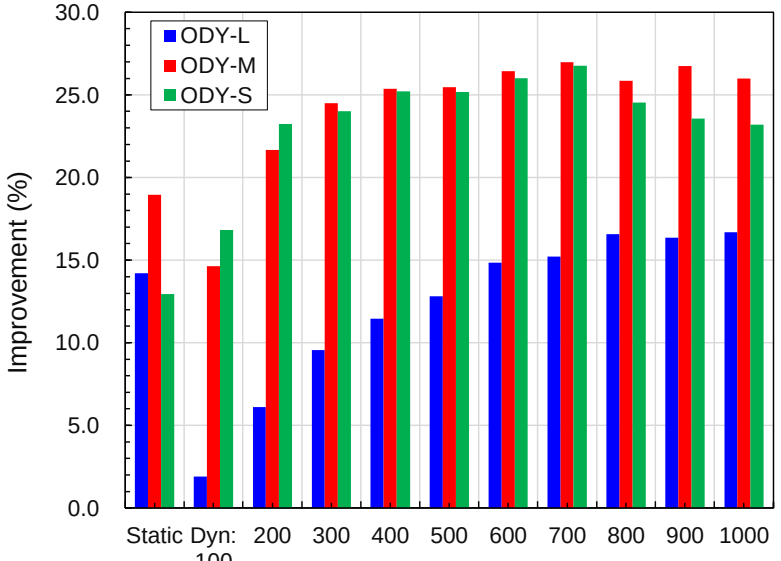


- 4 CMG (Core Memory Group), 12-Cores/CMG
- NUMA Architecture (Non-Uniform Memory Access)
- Recommended Programming Model
  - Hybrid 12 × 4
  - 12-OpenMP Threads, 1-MPI Process for each CMG
  - 4-MPI Proc's on each Node

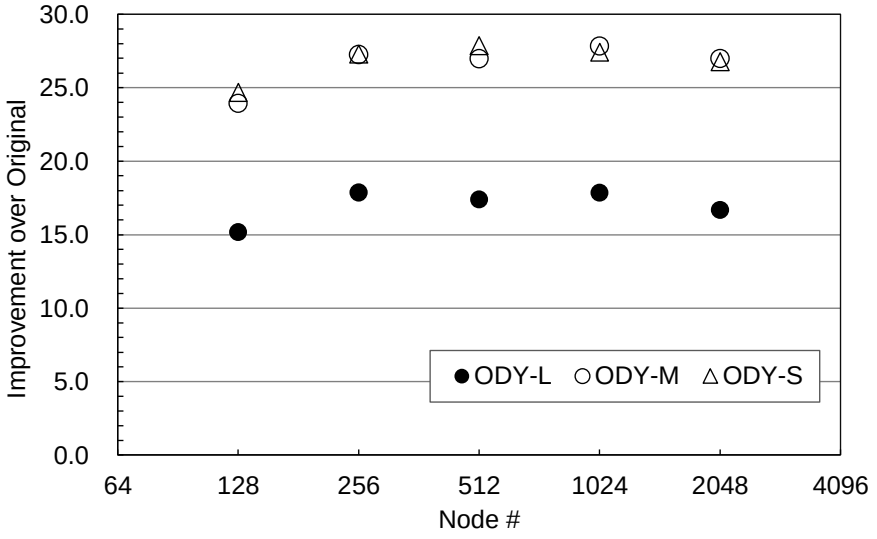
Weak Scaling: up to 2,048 Nodes
Improvement over "Original" of HB 12x4

S: 96x 96x 48x3 DOF/node (= 1,327,104)
M:128x128x 64x3 DOF/node (= 3,145,728)
L: 200x200x100x3 DOF/node (=12,000,000)

2,048 nodes

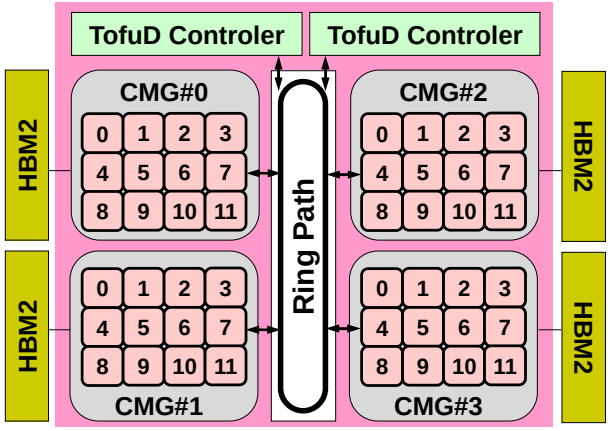


Best Cases for Each Node #



- Dot Products
- Matrix Vector Multiplication
- **Exercise**
  - **Fortran codes only**

Overview

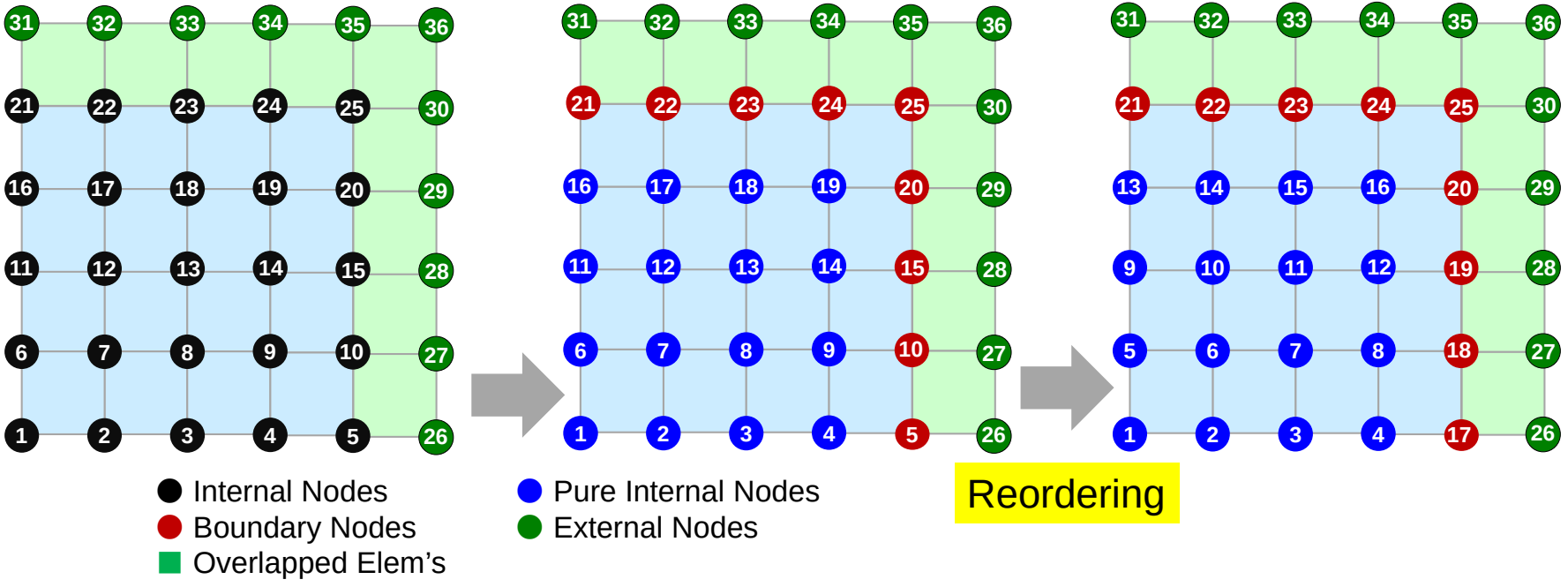


- You can experience the effects of CC-Overlapping using the following two types of code by 3D parallel FEM
  - Heat Conduction (in this class)
  - Solid Mechanics
  - HB 12x4
- Meshes are automatically generated in the code
- General Observations
  - 12-nodes are too small for experiencing the effects
  - Computational complexity of Solid Mechanics for each node in SpMV is 9-time that of Heat Conduction
  - If the problem size is small, loops for Heat Conduction might be too small for hiding communications

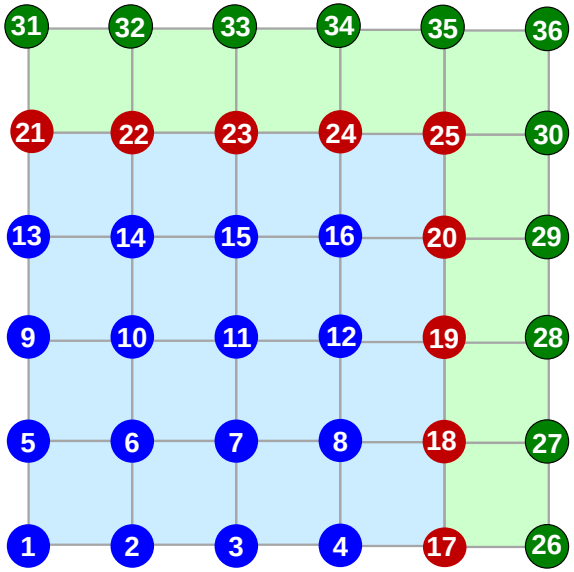
Preparation (Odyssey)

```
>$ cd /work/gt36/t36XYZ/pFEM
>$ cp /work/gt00/z30088/pFEM/cco.tar .
>$ tar xvf cco.tar

>$ cd CCoverlapping

>$ cd heat
>$ make
>$ ls ../run/sol*
   sol0 sol1 sol2 sol3

>$ cd ../solid
>$ make
>$ ls ../run/sol*
   sol0 sol1 sol2 sol3
```

Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]



Communication-Computation Overlapping (CC-Overlapping) for SpMV [KN ICPP 2017]




```
call MPI_Isend
call MPI_Irecv
call MPI_Waitall

!$omp parallel do (…)
  do i= 1, N
    (computing)
  enddo
```

Original

```
call MPI_Isend
call MPI_Irecv
```

```
!$omp parallel do (…)
  do i= 1, Ninn
    (computing)
  enddo
```

Pure Internal Nodes

```
call MPI_Waitall
```

```
!$omp parallel do (…)
  do i= Ninn+1, Nall
    (computing)
  enddo
```

Boundary Nodes

CC-Overlapping: Static Scheduling

```
$omp parallel (…)
$omp master
  call MPI_Isend
  call MPI_Irecv
  call MPI_Waitall
$omp end master
```

Master
Halo Comm.

```
!$omp do schedule (…)
  do i= 1, Ninn
    (computing)
  enddo
```

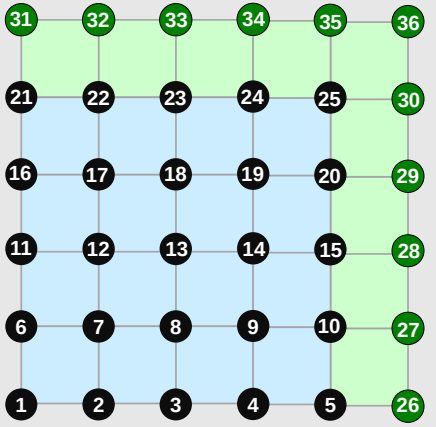
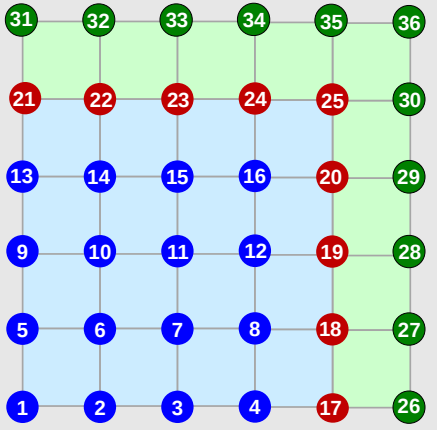
Dynamic
Pure Internal Nodes

```
!$omp do
  do i= Ninn+1, Nall
    (computing)
  enddo
```

Static
Boundary Nodes

```
!$omp end parallel
```

CC-Overlapping: Dynamic Scheduling

sol0	Original Code	
sol1	Static Scheduling without Reordering	
sol2	Static Scheduling with Reordering	
sol3	Dynamic Scheduling (with Reordering)	

mesh.inp (fixed name)

```
256 256 192  npx, npy, npz
  4   4   3  ndx, ndy, ndz
 12   1      PEsmpTOT, FTflag
 50          ITERmax
```

- **npx/npy/npz:** Number of nodes in each direction
- **ndx/ndy/ndz:** Number of division in each direction
  (**npx/ndx, npy/ndy, npz/ndz**): divisible
  ndx x **ndy** x **ndz**= Total # of MPI Processes
- **PEsmpTOT:** Number of Thread for each MPI Process
  4x4x3= 48 processes for 12 nodes
- **Ftflag:** (unused)
- **ITERmax:** Total Number of CG Iterations

[A]{p}={q} in “sol0”

```
      call SOLVER_SEND_RECV                                             &
     &   ( N, NP, NEIBPETOT, NEIBPE, IMPORT_INDEX, IMPORT_ITEM,         &
     &     EXPORT_INDEX, EXPORT_ITEM, WS, WR, WW(1,P), my_rank)

!$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
```

[A]{p}={q} in “sol1”

```
!$omp parallel do
      do i= 1, N
        do k= index(i-1)+1, index(i)
          if (item(k).gt.N) BN(i)= 1
        enddo
      enddo
```

```
      do neib= 1, NEIBPETOT
        istart= IMPORT_INDEX(neib-1)
        inum  = IMPORT_INDEX(neib  ) - istart
        call MPI_IRECV ()
      enddo

      do neib= 1, NEIBPETOT
        istart= EXPORT_INDEX(neib-1)
        inum  = EXPORT_INDEX(neib  ) - istart
!$omp parallel do private (ii)
        do k= istart+1, istart+inum
            ii   = EXPORT_ITEM(k)
          WS(k)= WW(ii,P)
        enddo
        call MPI_ISEND ()
      enddo
```

```
!C— Pure Inner Nodes

!$omp parallel do private(j,k,i,WVAL)
      do j= 1, N
      if (BN(j).eq.0) then
        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
      endif
      enddo
      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)

!C— Boundary Nodes

!$omp parallel do private(j,k,i,WVAL)
      do j= 1, N
      if (BN(j).eq.1) then
        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
      endif
      enddo
```

[A]{p}={q} in “sol2” Ninn: Total # of “Pure Internal” Nodes

```
      do neib= 1, NEIBPETOT
        istart= IMPORT_INDEX(neib-1)
        inum  = IMPORT_INDEX(neib  ) - istart
        call MPI_IRECV ()
      enddo

      do neib= 1, NEIBPETOT
        istart= EXPORT_INDEX(neib-1)
        inum  = EXPORT_INDEX(neib  ) - istart
!$omp parallel do private (ii)
        do k= istart+1, istart+inum
          ii   = EXPORT_ITEM(k)
          WS(k)= WW(ii,P)
        enddo
        call MPI_ISEND ()
      enddo
```

```
!C== Pure Inner Nodes

!$omp parallel do private(j,k,i,WVAL)
      do j= 1, Ninn
        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
      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)

!C== Boundary Nodes

!$omp parallel do private(j,k,i,WVAL)
      do j= Ninn+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
```

[A]{p}={q} in “sol3”

```
!$omp parallel private(j,k,i,WVAL,neib,istart,inum, ierr)

!$omp master
 do neib= 1, NEIBPETOT
        istart= IMPORT_INDEX(neib-1)
        inum  = IMPORT_INDEX(neib  ) - istart
        call MPI_IRECV ()
      enddo

      do neib= 1, NEIBPETOT
        istart= EXPORT_INDEX(neib-1)
        inum  = EXPORT_INDEX(neib  ) - istart
        do k= istart+1, istart+inum
            ii   = EXPORT_ITEM(k)
           WS(k)= WW(ii,P)
        enddo
        call MPI_ISEND ()
      enddo
      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)
!$omp end master
```

```
!C— Pure Inner Nodes

!$omp do schedule (runtime)
      do j= 1, Ninn
        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— Boundary Nodes

!$omp do
      do j= Ninn+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

!$omp end parallel
```

[A]{p}={q} in “sol3”

```
!$omp parallel private(j,k,i,WVAL,neib,istart,inum,  ierr)

!$omp master
 do neib= 1, NEIBPETOT
        istart= IMPORT_INDEX(neib-1)
        inum  = IMPORT_INDEX(neib  ) - istart
        call MPI_IRECV ()
      enddo

      do neib= 1, NEIBPETOT
        istart= EXPORT_INDEX(neib-1)
        inum  = EXPORT_INDEX(neib  ) - istart
        do k= istart+1, istart+inum
          ii    = EXPORT_ITEM(k)
          WS(k)= WW(ii,P)
        enddo
        call MPI_ISEND ()
      enddo
      call MPI_WAITALL (2*NEIBP
!$omp end master
```

This loop is not parallelized

```
!C-- Pure Inner Nodes

!$omp do schedule (runtime)
      do j= 1, Ninn
        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-- Boundary Nodes

!$omp do
      do j= Ninn+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

!$omp end parallel
```

How to Run

```
>$ cd /work/gt36/t36XYZ/pFEM/Ccoverlappnig

>$ cd heat/run
>$ (modify “mesh.inp”, “g1.sh”, “g3.sh”)
>$ pjsub g1.sh
>$ pjsub g3.sh

>$ cd ../../solid/run
>$ (modify “mesh.inp”, “g1.sh”, “g3.sh”)
>$ pjsub g1.sh
>$ pjsub g3.sh
```

“g1.sh” for “sol0/sol1/sol2”

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture6-o
#PJM -L node=12
#PJM --mpi proc=48
#PJM --omp thread=12
#PJM -L elapse=00:15:00
#PJM -g gt36
#PJM -j
#PJM -e err
#PJM -o test_422.lst

module load fj
module load fjmpi
export OMP_NUM_THREADS=12
export XOS_MMM_L_PAGING_POLICY=demand:demand:demand

mpiexec ./sol0
mpiexec ./sol0
mpiexec ./sol0
mpiexec ./sol0
mpiexec ./sol0
mpiexec ./sol1
mpiexec ./sol1
mpiexec ./sol1
mpiexec ./sol1
mpiexec ./sol1
mpiexec ./sol2
mpiexec ./sol2
mpiexec ./sol2
mpiexec ./sol2
mpiexec ./sol2

rm wk*.*
```

“g3.sh” for “sol3”

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture6-o
#PJM -L node=12
#PJM --mpi proc=48
#PJM --omp thread=12
#PJM -L elapse=00:15:00
#PJM -g gt36
#PJM -j
#PJM -e err
#PJM -o test_422.lst

module load fj
module load fjmpi
export OMP_NUM_THREADS=12
export XOS_MMM_L_PAGING_POLICY=demand:demand:demand

export OMP_SCHEDULE="dynamic,500"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,1000"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,1500"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,2000"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,2500"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,3000"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,3500"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,4000"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,4500"
mpiexec ./sol3
export OMP_SCHEDULE="dynamic,5000"
mpiexec ./sol3

rm wk*.*
```

Output of “Heat Conduction”

```
 256       256       192                                   npx, npy, npz
   4         4         3                                   ndx, ndy, ndz
  12                                                       PEsmpTOT

*** matrix conn.      4.319429E+00 sec.
*** matrix ass.       2.607580E-01 sec.

    1    1.377489E+01
    2    1.370239E+01
    3    1.362989E+01
    4    1.355739E+01
    5    1.348489E+01
    6    1.341239E+01
(…)
  187    2.854326E-01
  188    2.113718E-01
  189    1.356350E-01
  190    5.126520E-02
  191    1.212828E-14
*** real  COMP.      5.084291E-01 sec.                    Computation Time for CG

       0       1    1.824050E+04                          Reference Result at Origin

jwe0002i stop  * normal termination
```

Output of “Solid Mechanics”

```
 256       256       192                                 npx, npy, npz
   4         4         3                                 ndx, ndy, ndz
  12                                                     PEsmpTOT
   1
### NORMAL

color number:         0
    3.762594E-01    1.344097E-02
  262144   -1.280253E+01   -1.280253E+01    4.802166E+01
   50    1.035398E+00

elapsed      50    6.838770E-03                        Computation Time/Iteration

### min/max/ave    3.419377E-01    3.419435E-01    3.419396E-01
```

Results (Small is Good)

12-nodes, 48-processes (4x4x3), 12-threads/proc.

Problem Size (Total) Problem Size (MPI Proc.)		256x256x192 64x64x64	512x512x384 128x128x128	1,024x512x384 256x128x128
Heat Conduction	sol0	3.522×10^{-1}	$4.064 \times 10^{+0}$	$8.619 \times 10^{+0}$
	sol1	4.619×10^{-1}	$5.126 \times 10^{+0}$	$1.013 \times 10^{+1}$
	sol2	4.791×10^{-1}	$4.062 \times 10^{+0}$	$8.246 \times 10^{+0}$
	sol3 (best)	4.423×10^{-1}	$4.088 \times 10^{+0}$	$8.284 \times 10^{+0}$
Solid Mechanics	sol0	8.106×10^{-3}	5.504×10^{-2}	-
	sol1	8.483×10^{-3}	5.784×10^{-2}	-
	sol2	6.801×10^{-3}	4.977×10^{-2}	-
	sol3 (best)	6.653×10^{-3}	5.011×10^{-2}	-

Future Work

- Applying to CC-Overlapping to more complicated preconditioning methods.