3D Heat Transfer Problem

Overview

1. 2D/3D Heat Conduction equation
2. Finite Volume Method
3. TDMA (Tri-Diagonal Matrix) Interative Solver
4. Serial/MPI/OpenMP + Hybrid version
5. Scalability results from test runs

Sample problem : 2D Heat Conduction

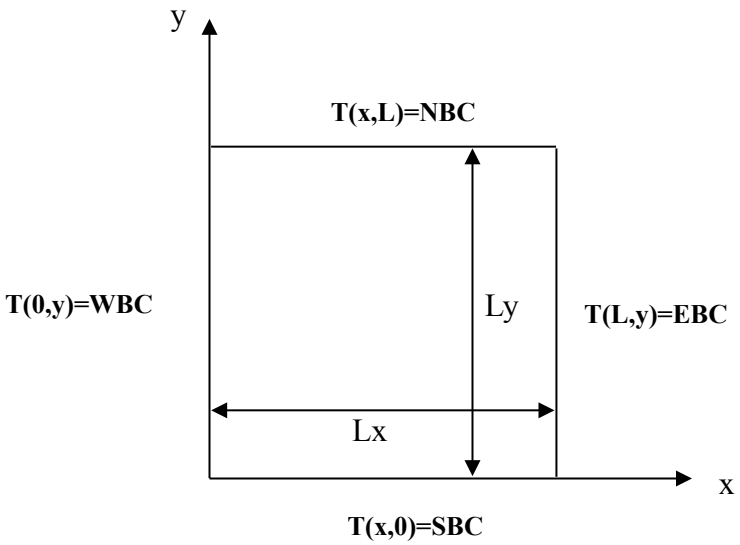
Governing Equation

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

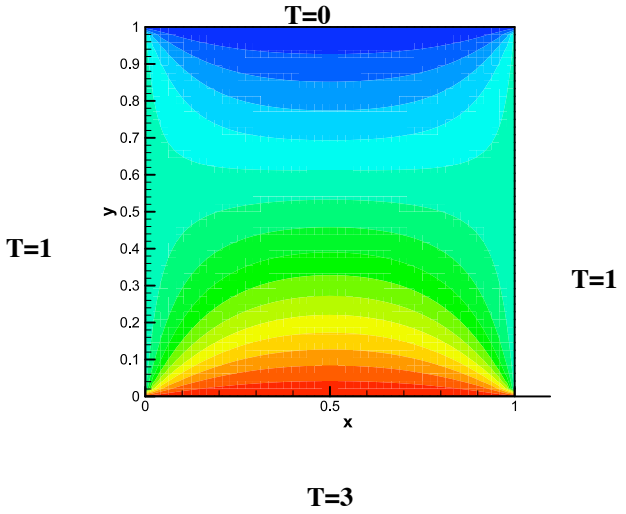
T : Temperature

t : Time

α : Thermal expansion coefficient



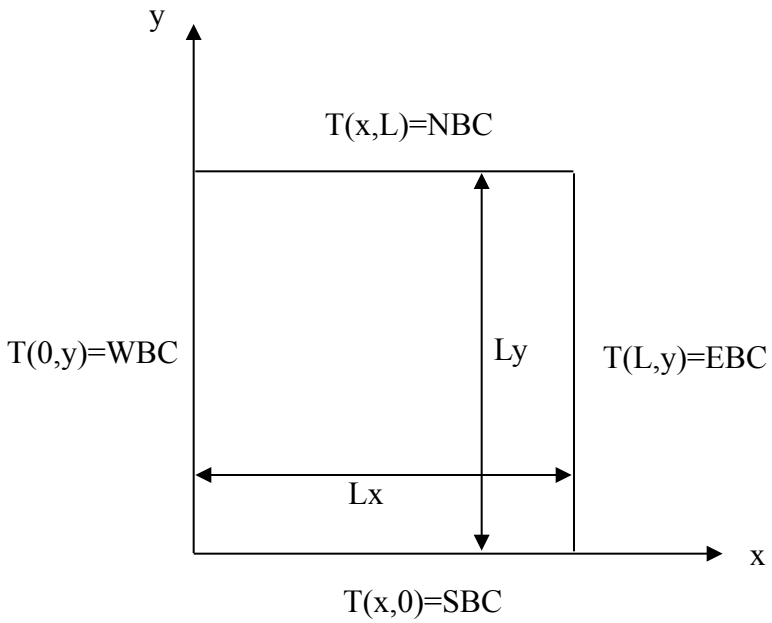
Ex)



Solution by Finite Volume Method

Governing Equation

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) + q \ , \quad \alpha = \frac{k}{\rho C}$$



T : Temperature

t : Time

k : Thermal conductivity

C : Heat Capacity

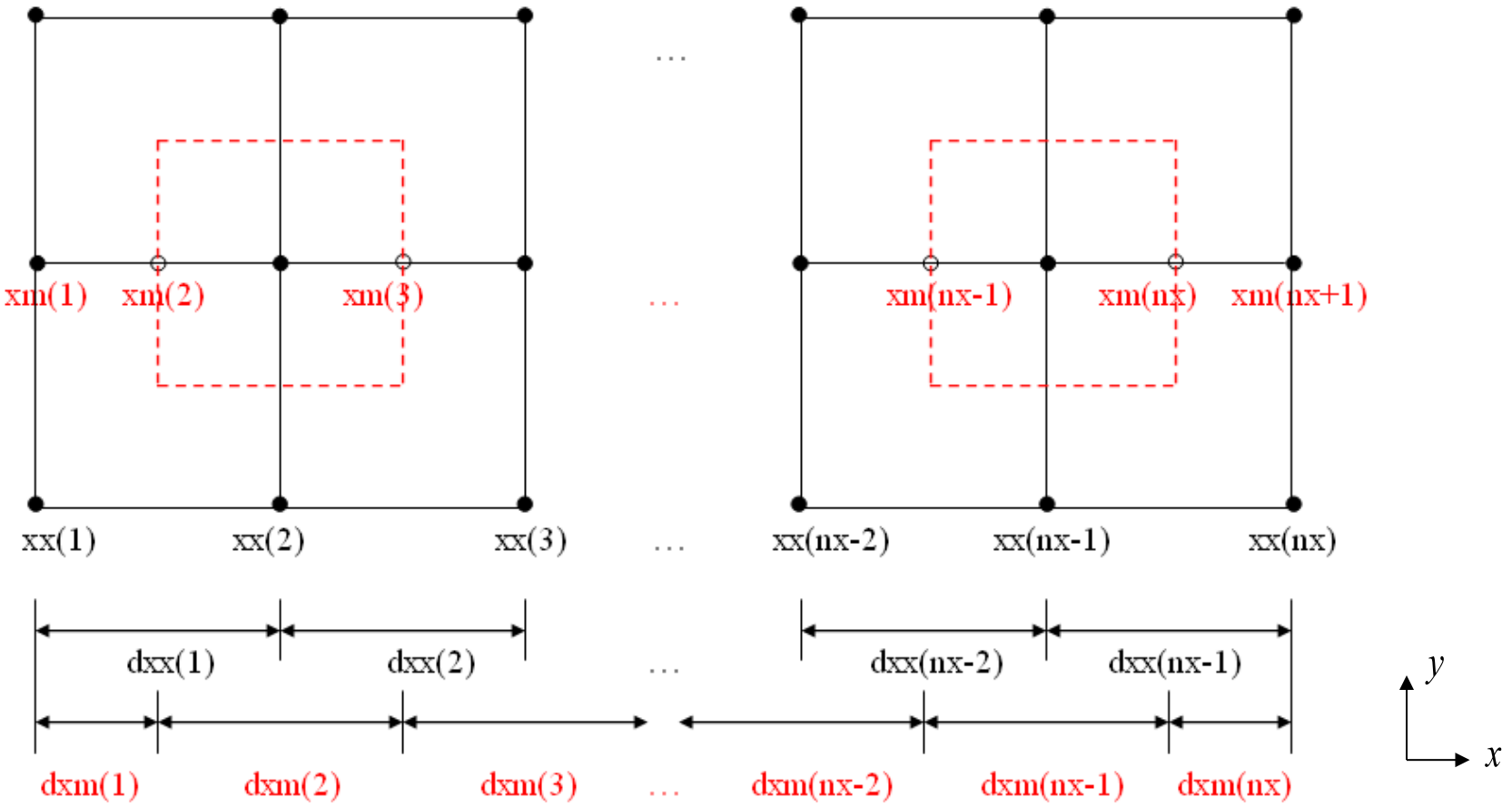
ρ : Density

α : Thermal expansion coefficient

q : Heat generation rate

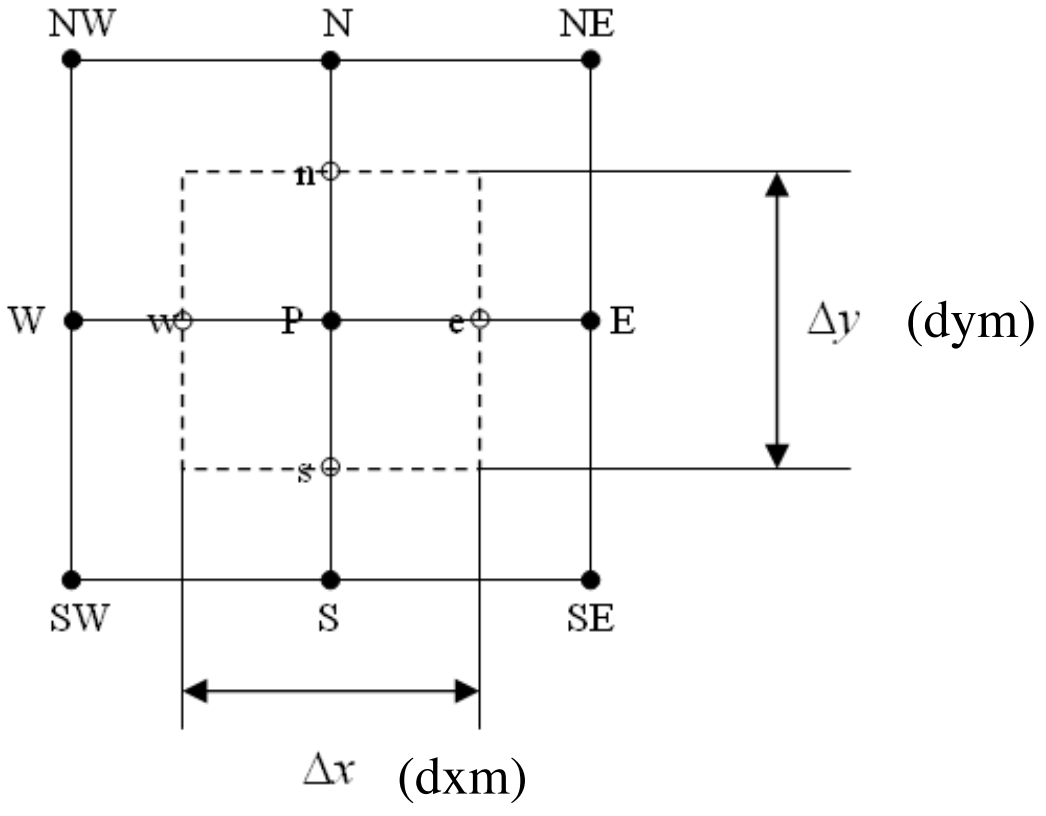
Solution by Finite Volume Method

Coordinate notations



Solution by Finite Volume Method

Discretization



Solution by Finite Volume Method

Discretization

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) + q$$

$$\int_t^{t+\Delta t}\int_s^n\int_w^e \left[\frac{\partial T}{\partial t}\right] dx\,dy\,dt = \Delta x \Delta y \left(T_P - T_P^{old}\right)$$

$$\int_t^{t+\Delta t}\int_s^n\int_w^e \alpha\left[\frac{\partial}{\partial x}\left(\frac{\partial T}{\partial x}\right)\right] dx\,dy\,dt$$

$$= \alpha\int_t^{t+\Delta t} \Delta y\left[\frac{(T_E - T_P)}{(\delta x)_e} - \frac{(T_P - T_W)}{(\delta x)_w}\right] dt$$

$$= \alpha\Delta y\Delta t\left[\frac{(T_E - T_P)}{(\delta x)_e} - \frac{(T_P - T_W)}{(\delta x)_w}\right]$$

$$\int_t^{t+\Delta t}\int_s^n\int_w^e \alpha\left[\frac{\partial}{\partial y}\left(\frac{\partial T}{\partial y}\right)\right] dx\,dy\,dt$$

$$= \alpha\int_t^{t+\Delta t} \Delta x\left[\frac{(T_N - T_P)}{(\delta y)_n} - \frac{(T_P - T_S)}{(\delta y)_s}\right] dt$$

$$= \alpha\Delta x\Delta t\left[\frac{(T_N - T_P)}{(\delta y)_n} - \frac{(T_P - T_S)}{(\delta y)_s}\right]$$

$$\int_t^{t+\Delta t}\int_s^n\int_w^e [q]\,dx\,dy\,dt = \Delta x \Delta y \Delta t\,(q)$$

Solution by Finite Volume Method

Discretization

$$\left[\frac{\Delta x\Delta y}{\Delta t}+\frac{\alpha\Delta y}{(\delta x)_e}+\frac{\alpha\Delta y}{(\delta x)_w}+\frac{\alpha\Delta x}{(\delta y)_n}+\frac{\alpha\Delta x}{(\delta y)_s}\right]T_P$$

$$=\frac{\alpha\Delta y}{(\delta x)_e}T_E+\frac{\alpha\Delta y}{(\delta x)_w}T_W+\frac{\alpha\Delta x}{(\delta y)_n}T_N+\frac{\alpha\Delta x}{(\delta y)_s}T_S+\left(\frac{\Delta x\Delta y}{\Delta t}T_P^{old}\right)+\Delta x\Delta y(q)$$

$$a_P T_P = a_E T_E + a_W T_W + a_N T_N + a_S T_S + S_T$$

$$a_p^{old}=\frac{\Delta x\Delta y}{\Delta t} \qquad a_E=\frac{\alpha\Delta y}{(\delta x)_e} \qquad a_W=\frac{\alpha\Delta y}{(\delta x)_w} \qquad a_N=\frac{\alpha\Delta x}{(\delta y)_n} \qquad a_S=\frac{\alpha\Delta x}{(\delta y)_s}$$

$$a_P = a_E + a_W + a_N + a_S + a_p^{old} \qquad S_T = a_p^{old}T_P^{old} + \Delta x\Delta y(q)$$

Solution by Finite Volume Method

Iterative Solver (TDMA)

- The tridiagonal matrix algorithm (TDMA), also known as the Thomas algorithm, is a simplified form of Gaussian elimination that can be used to solve tridiagonal systems of equations. A tridiagonal system may be written as

$$a_i T_i = b_i T_{i+1} + c_i T_{i-1} + d_i \quad \longleftrightarrow \quad a_P T_P = a_N T_N + a_S T_S + \left(S_t + a_E T_E + a_W T_W\right)$$

where, $c_1 = 0$ and $b_n = 0$. In matrix form, this system is written as

$$\begin{bmatrix} a_1 & -b_1 & & & & & 0 \\ -c_2 & a_2 & -b_2 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & -c_{N-1} & a_{N-1} & -b_{N-1} \\ 0 & & & & & -c_N & a_N \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ \\ \vdots \\ T_{N-1} \\ T_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \\ \vdots \\ d_{N-1} \\ d_N \end{bmatrix}$$

Solution by Finite Volume Method

Iterative Solver (TDMA)

$$a_i T_i = b_i T_{i+1} + c_i T_{i-1} + d_i \qquad (1)$$

Step 1 : Forward substitution

Let, $P_1 = \frac{b_1}{a_1} \quad Q_1 = \frac{d_1}{a_1}$

Assumption, $T_{i-1} = P_{i-1} T_i + Q_{i-1} \qquad (2)$

We obtain the following equations as substitute (2) for (1)

$$P_i = \frac{b_i}{a_i - c_i P_{i-1}} \quad Q_i = \frac{d_i + c_i Q_{i-1}}{a_i - c_i P_{i-1}} \qquad (3)$$

From the above matrix,

$$a_N T_N = c_N T_{N-1} + d_N \qquad (4)$$

In case of i=N from eq.(2)

$$T_{N-1} = P_{N-1} T_N + Q_{N-1} \qquad (5)$$

From (4) and (5),

$$T_N = \frac{d_N + c_N Q_{N-1}}{a_N - c_N P_{N-1}}$$

Therefore,

$$T_N = Q_N$$

Step 2 : Backward substitution

From (2), we can find all of the solutions

Solution by Finite Volume Method

Iterative Solver (TDMA)

From, $T_1 = B.C. \qquad a_1 = 1,\ b_1 = 0,\ c_1 = 0,\ d_1 = T_1$

Therefore, $P_1 = \frac{b_1}{a_1} = 0, \qquad Q_1 = \frac{d_1}{a_1} = T_1$

Do i=2,N-1

$$d_i = S_{T,i} + a_E T_{i+1} + a_W T_{i-1}$$

$$P_i = \frac{b_i}{a_i - c_i P_{i-1}} \qquad Q_i = \frac{d_i + c_i Q_{i-1}}{a_i - c_i P_{i-1}}$$

ENDDO

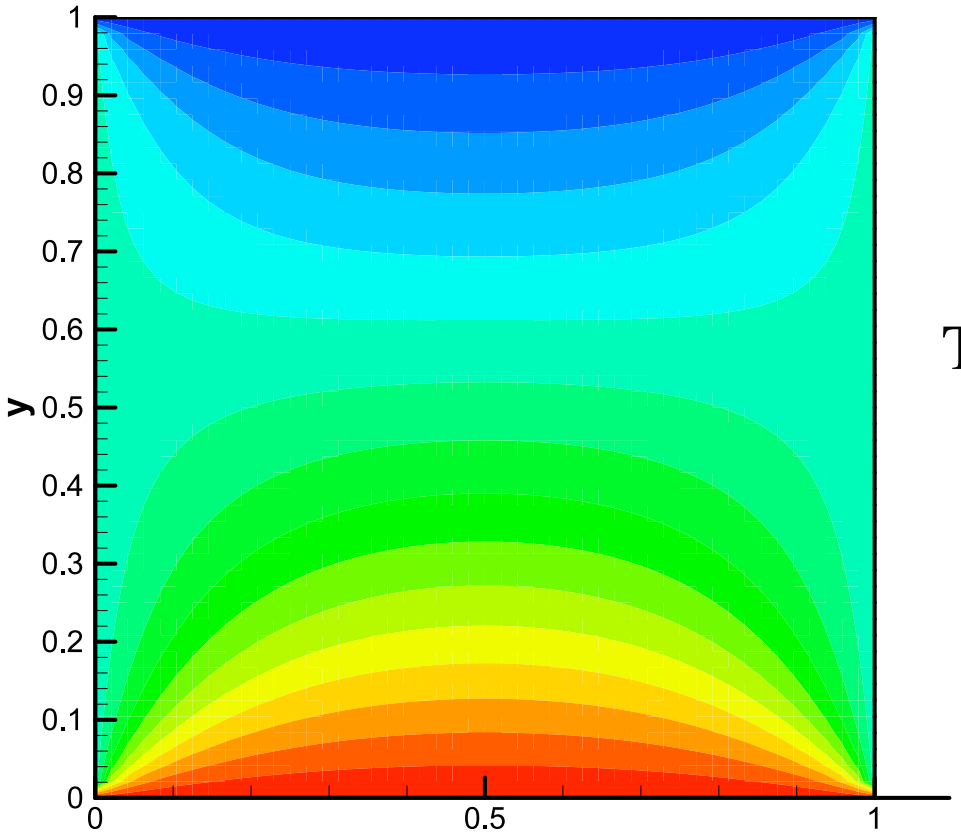
Do i=N-1,2

$$T_i = P_i T_{i+1} + Q_i$$

ENDDO

T=0

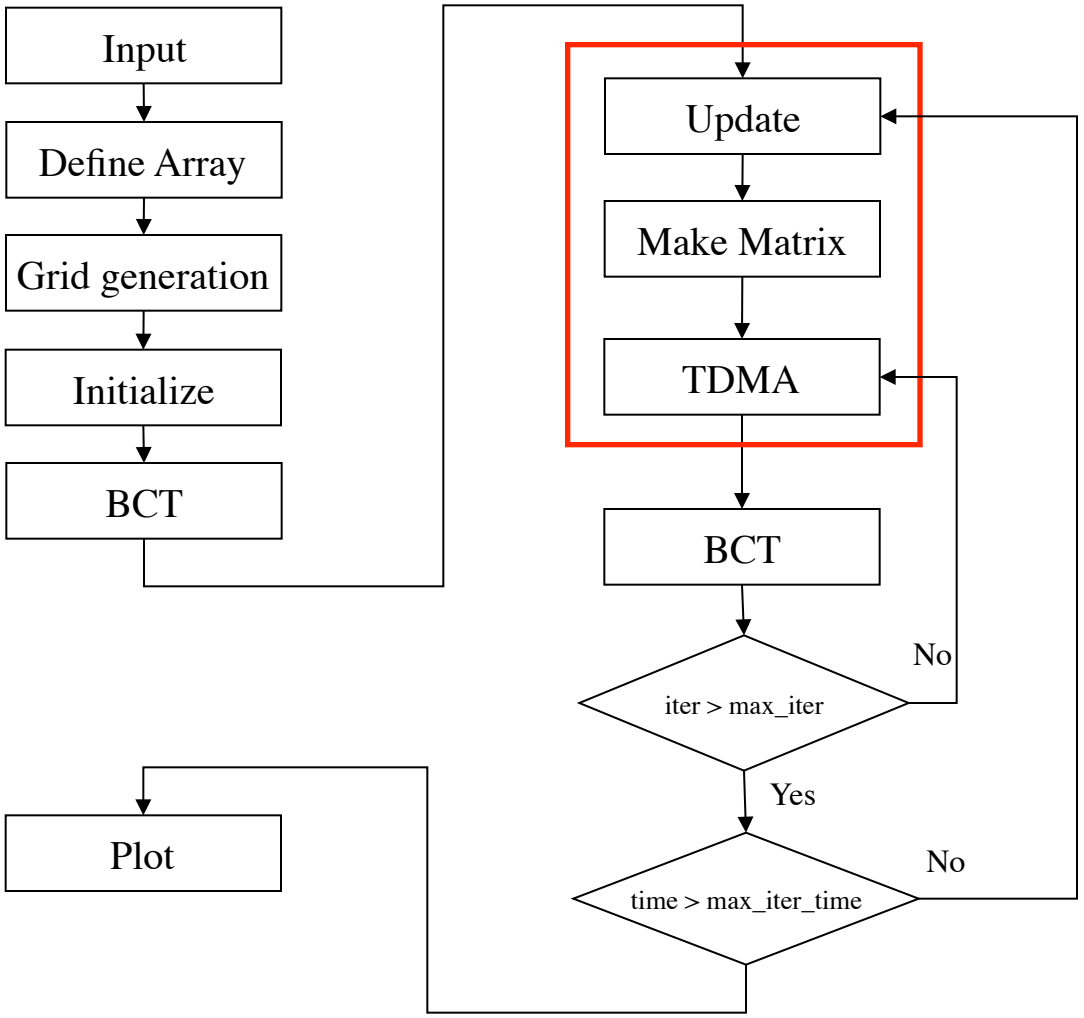
T=1



T=1

T=3

Code Flow Chart



5. Parallelization

- **The Very Basic!**

```
Program Hello
INCLUDE "mpif.h"
INTEGER :: ierr

CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NP_tot,ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MyRank,ierr)

Print *, "NP_tot=",NP_tot,"MyRank=",MyRank,"Hello World"

CALL MPI_FINALIZE(ierr)

End Program Hello
```

Parallelization (C version)

- **The Very Basic!**

```
/*Program Hello*/
#include "mpi.h"

int main(int *argc, int** argv)
{
   int ierr,NP_tot,MyRank;

   ierr=MPI_Init(argc,argv);
   ierr=MPI_Comm_size(MPI_COMM_WORLD,&NP_tot);
   ierr=MPI_Comm_size(MPI_COMM_WORLD,&MyRank);


 printf("NP_tot=%d  MyRank=%d Hello World",NP_tot,MyRank);

   ierr=MPI_Finalize();

   return(0);
}
```

Main.f90

- INCLUDE "mpif.h"
  - ✓ Variable declaration
- CALL MPI_INIT(ierr)
- CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPtotal,ierr)
- CALL MPI_COMM_RANK(MPI_COMM_WORLD,MyPE,ierr)
  - ✓ Def_Array (decomposition)
  - ✓ Setup MPI datatypes
  - ✓ Actual coding work…
  - ✓ Call TDMA
  - ✓ Call Exchange (Communication Routine)
  - ✓ Plot
- CALL MPI_FINALIZE(ierr)
  - ✓ End of program

Main.c

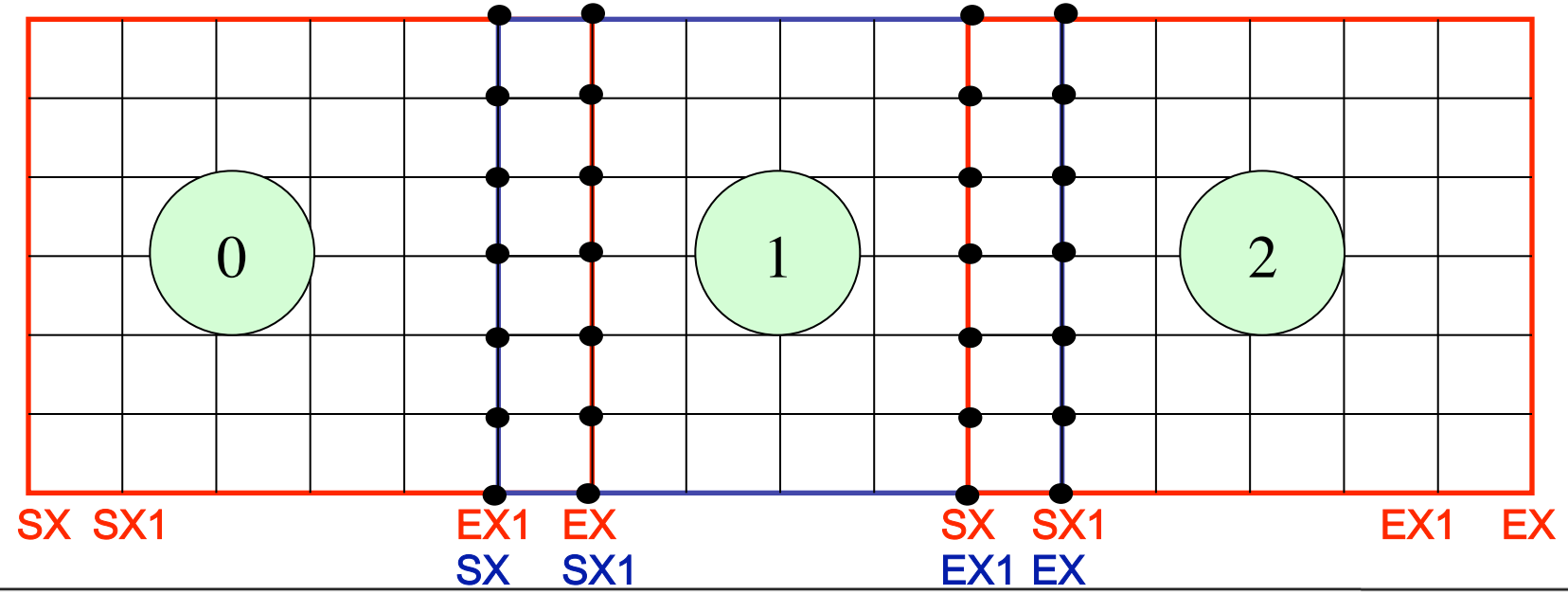
- #include "mpi.h"
  - ✓ Variable declaration
- ierr=MPI_INIT(argc,argv);
- ierr=MPI_Comm_size(MPI_COMM_WORLD,NPtotal);
- ierr=MPI_Comm_rank(MPI_COMM_WORLD,MyPE) ;
  - ✓ Def_Array (decomposition)
  - ✓ Setup MPI datatypes
  - ✓ Actual coding work…
  - ✓ Call TDMA
  - ✓ Call Exchange (Communication Routine)
  - ✓ Plot
- ierr=MPI_Finalize();
  - ✓ End of program

Domain Decomposition

- Data parallelization: Extensibility, Portability
- Divide computational domain into many sub-domains based on number of processors
- Solves the same problem on the sub-domians, need to transfer the b.c. information of overlapping boundary area
- Requires communication between the subdomains in every time step
- Major parallelization method in CFD applications
- For a good scalability, requires careful algorithm implementation.

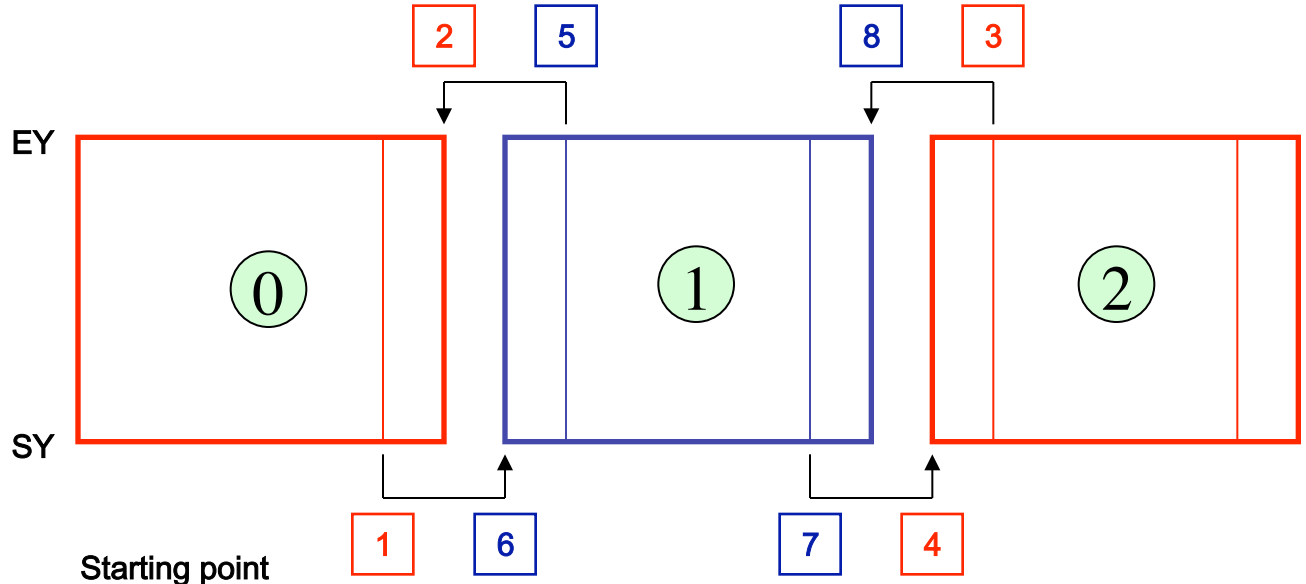
Def_Array.f90

```
INTEGER :: TMP_X
TMP_X = (Nx + (2*Nptotal-2))/Nptotal
SX1 = (myPE*(TMP_X-2))+2
EX1 = SX1 + TMP_X -3
IF(myPE==Nptotal-1) EX1 = Nx-1
SX = SX1-1
EX = EX1+1
```



Exchange.f90

- **Send/IRecv Communication**



Starting point

myPE=0

1 MPI_SEND (Temp_s(sy), Count, …, myPE+1, …)♪

2 MPI_IRECV(Temp_r(sy), Count, …, myPE+1, …)♪

3 MPI_SEND (Temp_s(sy), Count, …, myPE-1, …)♪

4 MPI_IRECV(Temp_r(sy), Count, …, myPE-1, …)♪

myPE=NPtotal-1

myPE=Else

5 MPI_SEND (Temp_s(sy), Count, …, myPE-1, …)♪

6 MPI_IRECV(Temp_r(sy), Count, …, myPE-1, …)♪

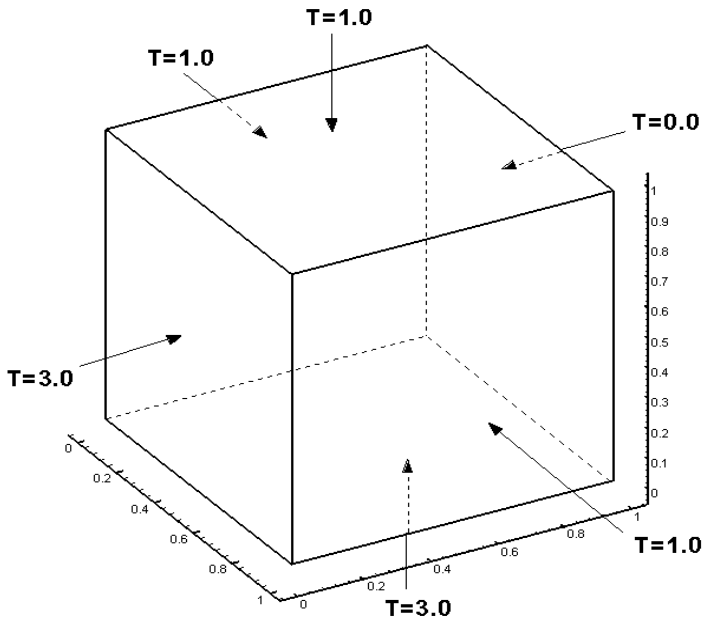
7 MPI_SEND (Temp_s(sy), Count, …, myPE+1, …)♪

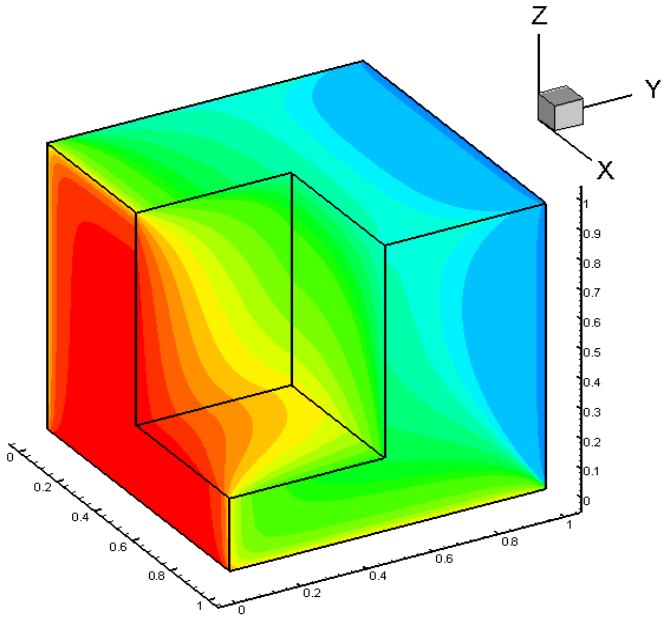
8 MPI_IRECV(Temp_r(sy), Count, …, myPE+1, …)♪

Count = EY – SY + 1

3D Heat Conduction Model

- **Solving heat conduction equation** $\frac{\partial T}{\partial t} = k\nabla^2 T$ **by TDMA (Tri-Diagonal Matrix Algorithm)**



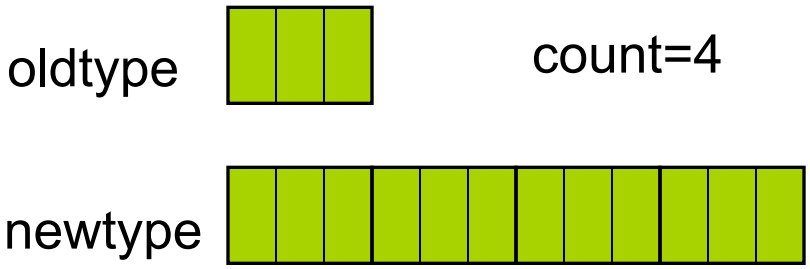


Extensions in The Code

- Cartesian coordinate index extension for 3D mesh
- Extra works in main.f90 & exchange.f90
- Virtual (Cartesian) topology implementation
- MPI datatype setup in main.f90
- Same OpenMP implementation for TDMA routine
- MPI_Sendrecv (blocking) communication in exchange.f90
- The other processes are the same as the 2-D code

Contiguous Datatype

MPI_TYPE_CONTIGUOUS(count,oldtype,newtype)		
IN	count	replication count
IN	oldtype	old datatype
OUT	Newtype	new datatype

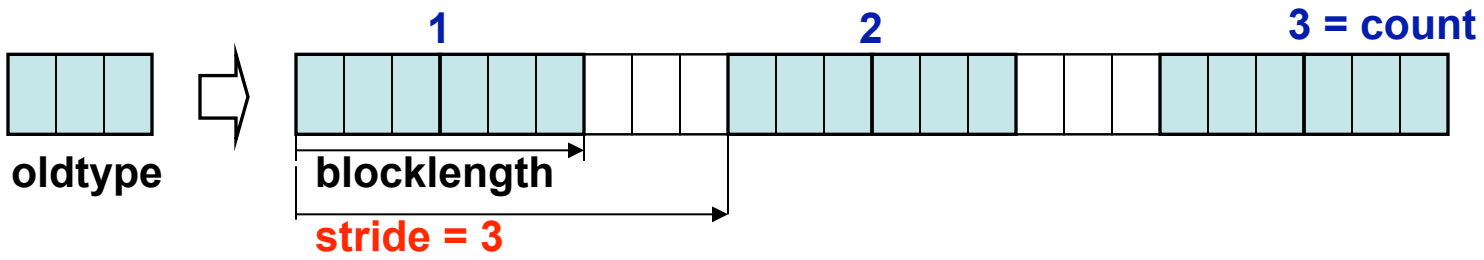


- **Simply specifies the new datatype that consists of `count` contiguous elements.**
- **Each of the elements has type `oldtype`.**

Vector Datatype

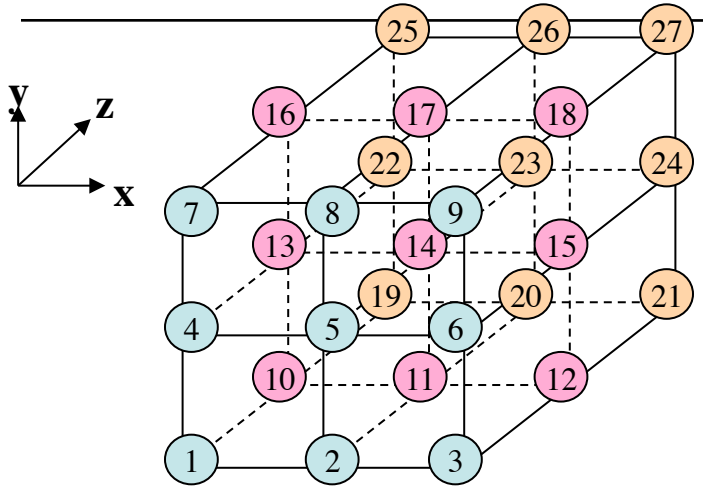
```
MPI_TYPE_VECTOR(count,blocklength,stride,oldtype,newtype)
```

IN	Count	Number of blocks
IN	Blocklength	Number of elements in each block
IN	Stride	spacing between start of each block, measured as number of elements
IN	oldtype	old datatype
OUT	Newtype	new datatype

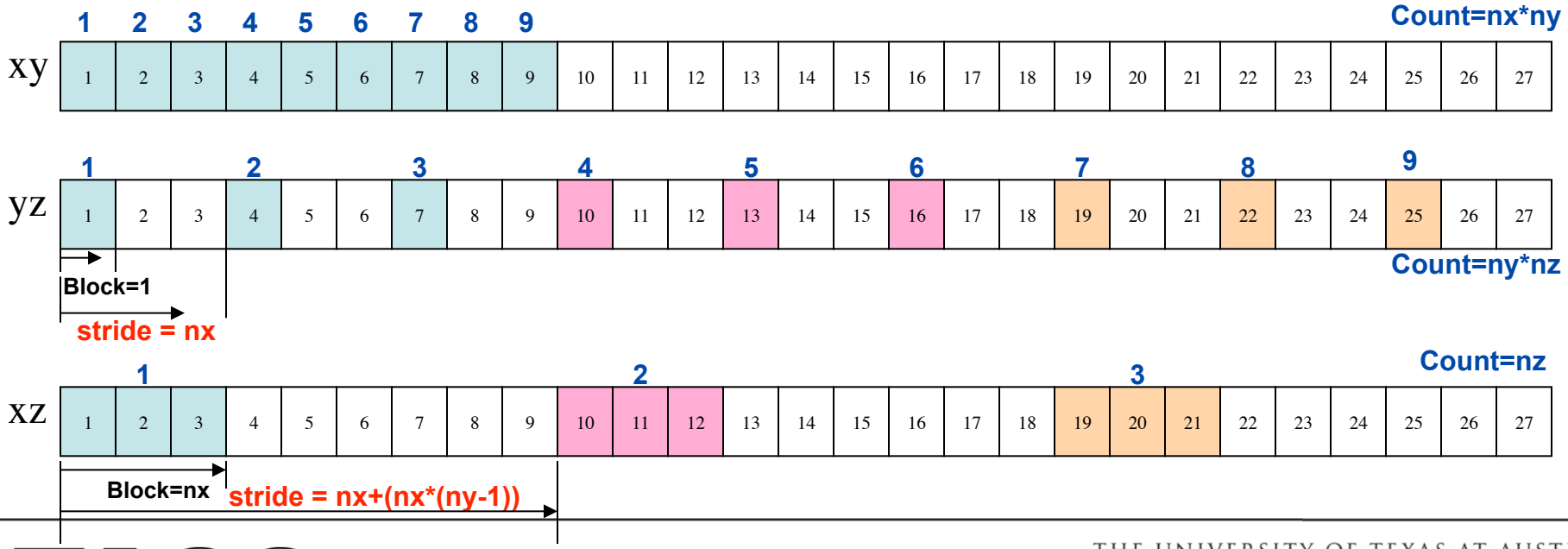


- Allows replication of a datatype into locations that consists of equally spaced blocks.
- Each block is obtained by concatenating the same number of copies of the old datatype.
- The spacing between the blocks is a multiple of the extent of the old datatype.

Matrix-Datatype Mapping



XY plane	YZ plane	XZ plane
Contiguous	Vector	Vector
Count = nx*ny	Count = ny*nz	Count = nz
	Blocklength = 1	Blocklength = nx
	Stride = nx	Stride = nx+(nx*(ny-1))

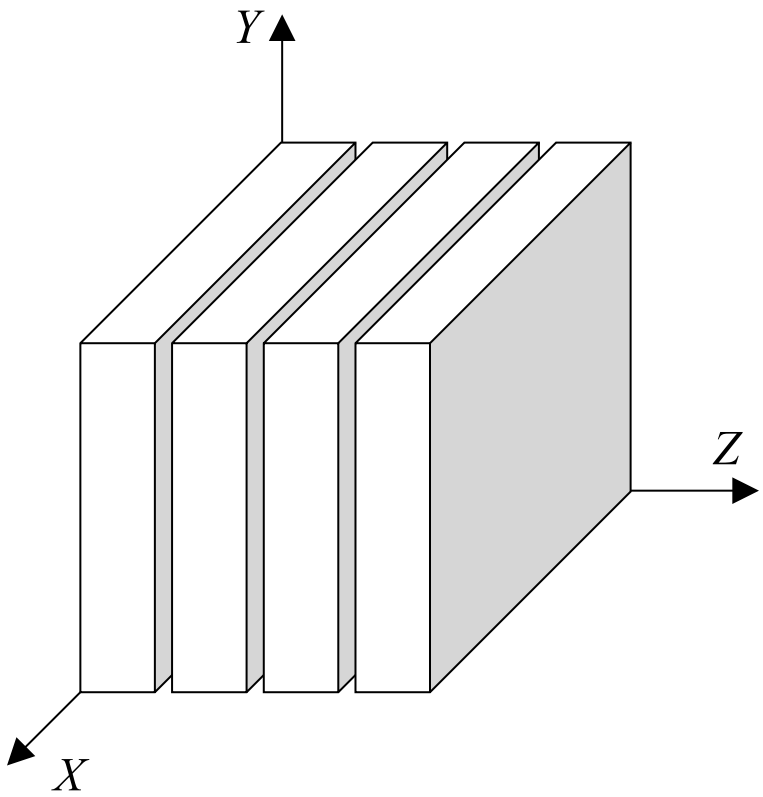


Recall: MPI_SendRecv

MPI_SendRecv
(sendbuf,sendcount,sendtype,dest,sendtag,
recvbuf,recvcount,recvtype,source,recvtag,
comm,status)

- Useful for communications patterns where each node both sends and receives messages (two-way communication).
- Executes a blocking send & receive operation
- Both function use the same communicator, but have distinct tag argument

3D Code Parallelization



- 1-D parallel method using domain-decompostion (Pipeline).
- Easy to implement.
- Hard to scale-up.
- Finding sweet spot is key for performance.
- Large data size - resulting communication overload when using with large number of processors.

main.f90 (1-D Parallel)

```
!-------------------------------------------------------------
! MPI Cartesian Coordinate Communicator
!-------------------------------------------------------------
CALL MPI_CART_CREATE
 (MPI_COMM_WORLD, NDIMS, DIMS, PERIODIC,REORDER,CommZ,ierr)
CALL MPI_COMM_RANK (CommZ, myPE,ierr)
CALL MPI_CART_COORDS (CommZ, myPE, NDIMS,CRDS,ierr)

CALL MPI_CART_SHIFT (CommZ,0,1,PEb,PEt,ierr)

!-------------------------------------------------------------
! MPI Datatype creation
!-------------------------------------------------------------
 CALL MPI_TYPE_CONTIGUOUS (Nx*Ny,MPI_DOUBLE_PRECISION,XY_p,ierr)

 CALL MPI_TYPE_COMMIT(XY_p,ierr)
```

Main.C (1-D Parallel)

```
//-------------------------------------------------------------
//MPI Cartesian Coordinate Communicator
//-------------------------------------------------------------
ierr=MPI_Cart_create
 (MPI_COMM_WORLD, NDIMS, DIMS, PERIODIC,REORDER,CommZ);
 ierr=MPI_Comm_rank(CommZ, myPE);
 ierr=MPI_Cart_coords (CommZ, myPE, NDIMS,CRDS);

ierr=MPI_Cart_shift (CommZ,0,1,PEb,PEt);

//-------------------------------------------------------------
//MPI Datatype creation
//-------------------------------------------------------------
 ierr=MPI_Type_contiguous (Nx*Ny,MPI_DOUBLE,XY_p);

 ierr=MPI_Type_commit(XY_p);
```

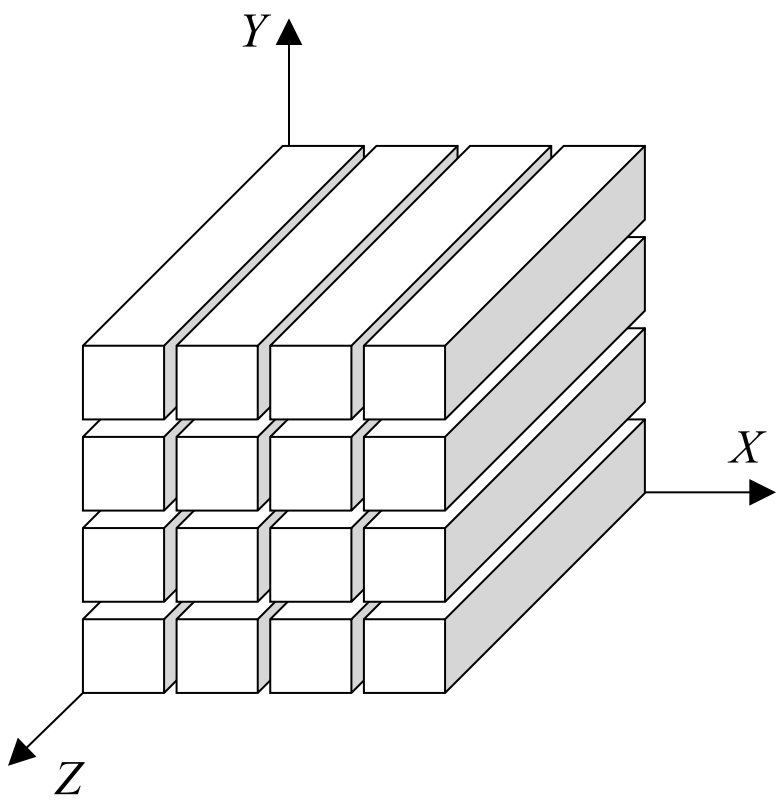
exchange.f90 (1-D Parallel)

```
!-----------------------------------------------------------------------
      SUBROUTINE EXCHANGE(VAL)
!-----------------------------------------------------------------------
      USE COMDAT_SHARED
      IMPLICIT NONE
      SAVE
!-----------------------------------------------------------------------
      INCLUDE 'mpif.h'
!-----------------------------------------------------------------------

      DOUBLE PRECISION,INTENT(inout) :: VAL(SX:EX,SY:EY,SZ:EZ)

      INTEGER :: status(MPI_STATUS_SIZE), K
!-----------------------------------------------------------------------
! TOP & BOTTOM

      CALL MPI_Sendrecv (val(SX, SY, EZ1),1,XY_p,PEt,1, &
                         val(SX, SY, SZ),1,XY_p,PEb,1,commZ,status,ierr)

      CALL MPI_Sendrecv (val(SX, SY, SZ1),1,XY_p,PEb,2, &
                         val(SX, SY, EZ),1,XY_p,PEt,2,commZ,status,ierr)
```

Exchange.C (1-D Parallel)

```
#include “mpi.h”
//-------------------------------------------------------------------------------
      void EXCHANGE(double ***val, MPI_Comm commZ)
{

      int *status = (int) malloc (MPI_STATUS_SIZE*sizeof(int));
//-------------------------------------------------------------------
//TOP & BOTTOM

      ierr=MPI_Sendrecv (&val[SX][SY][EZ1], 1,XY_p,PEt, 1, &val[SX][SY][SZ], 1,XY_p,PEb,
      1,commZ,status);

      ierr=MPI_Sendrecv (&val[SX][SY][SZ1],1,XY_p,PEb,2, &val[SX][SY][EZ], 1,XY_p,PEt,
      2,commZ,status);
}
```

2-D Parallelization (Sweep)



- Easier to scale-up with large number of processors than 1-D parallelization.
- Stronger scalability & performance efficiency with larger number of CPUs.
- Need to be very careful with sweeping direction.
- Should investigate application's characteristics before implementation.

main.f90 (2-D Parallel)

```
CALL MPI_CART_CREATE
 (MPI_COMM_WORLD, NDIMS, DIMS, PERIODIC,REORDER,CommXY,ierr)
CALL MPI_COMM_RANK (CommXY,myPE,ierr)
CALL MPI_CART_COORDS (CommXY,myPE,NDIMS,CRDS,ierr)

CALL MPI_CART_SHIFT (CommXY,1,1,PEw,PEe,ierr)
CALL MPI_CART_SHIFT (CommXY,0,1,PEs,PEn,ierr)
!-------------------------------------------------------
! MPI Datatype creation
!-------------------------------------------------------
CALL MPI_TYPE_VECTOR
 (cnt_yz,block_yz,strd_yz,MPI_DOUBLE_PRECISION,YZ_p,ierr)
CALL MPI_TYPE_COMMIT (YZ_p,ierr)

CALL MPI_TYPE_VECTOR
 (cnt_xz,block_xz,strd_xz,MPI_DOUBLE_PRECISION,XZ_p,ierr)
CALL MPI_TYE_COMMIT (XZ_p,ierr)
```

Main.C (2-D Parallel)

```
ierr=MPI_Cart_create
 (MPI_COMM_WORLD, NDIMS, DIMS, PERIODIC,REORDER,CommXY);
ierr=MPI_Comm_rank(CommXY,myPE);
ierr=MPI_Cart_coords (CommXY,myPE,NDIMS,CRDS);

ierr=MPI_Cart_shift(CommXY,1,1,PEw,PEe);
ierr=MPI_Cart_shift (CommXY,0,1,PEs,PEn);
//-----------------------------------------------------------
// MPI Datatype creation
//-----------------------------------------------------------
ierr=MPI_Type_vector
 (cnt_yz,block_yz,strd_yz,MPI_DOUBLE_PRECISION,YZ_p);
ierr=MPI_Type_commit (YZ_p);

ierr=MPI_Type_vector
 (cnt_xz,block_xz,strd_xz,MPI_DOUBLE_PRECISION,XZ_p);
ierr=MPI_Type_commit (XZ_p);
```

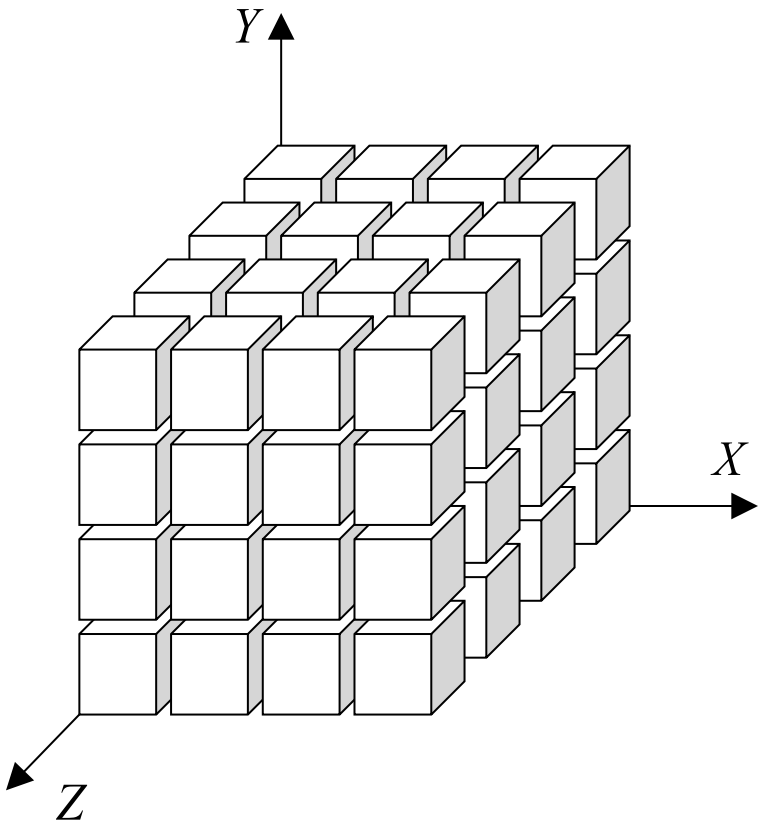
exchange.f90 (2-D Parallel)

```
!-----------------------------------------------------------------------
      INCLUDE 'mpif.h'
!-----------------------------------------------------------------------
      DOUBLE PRECISION,INTENT(inout) :: VAL(SX:EX,SY:EY,SZ:EZ)

      INTEGER :: status(MPI_STATUS_SIZE)

!-----------------------------------------------------------------------
!---LEFT&RIGHT
      CALL MPI_Sendrecv(val(EX1, SY, SZ),1,YZ_p,PEe,2, &
                        val(SX , SY, SZ),1,YZ_p,PEw,2,commXY,status,ierr)

      CALL MPI_Sendrecv(val(SX1, SY, SZ),1,YZ_p,PEw,3, &
                        val(EX , SY, SZ),1,YZ_p,PEe,3,commXY,status,ierr)

!---NORTH&SOUTH

      CALL MPI_Sendrecv(val(SX, EY1, SZ),1,XZ_p,PEn,0, &
                        val(SX, SY,  SZ),1,XZ_p,PEs,0,commXY,status,ierr)

       CALL MPI_Sendrecv(val(SX, SY1, SZ),1,XZ_p,PEs,1, &
                         val(SX, EY,  SZ),1,XZ_p,PEn,1,commXY,status,ierr)
```

Exchange.C (2-D Parallel)

```
//-----------------------------------------------------------------------
      #include “mpif.h”
//-----------------------------------------------------------------------

 void EXCHANGE(double ***val, MPI_Comm commZ)
{
      int *status = (int) malloc (MPI_STATUS_SIZE*sizeof(int));

//-----------------------------------------------------------------------
//---LEFT&RIGHT

      ierr=MPI_Sendrecv(&val[EX1][SY][SZ],1,YZ_p,PEe,2, &val[SX ][ SY][ SZ],1,YZ_p,PEw,
      2,commXY,status);
      ierr=MPI_Sendrecv(&val[SX1][ SY][ SZ],1,YZ_p,PEw,3, &val[EX ][ SY] [SZ],1,YZ_p,PEe,
      3,commXY,status);

//---NORTH&SOUTH

      ierr=MPI_Sendrecv(&val[SX][ EY1][ SZ],1,XZ_p,PEn,0, &val[SX][ SY][  SZ],1,XZ_p,PEs,
      0,commXY,status);
      ierr=MPI_Sendrecv(&val[SX][ SY1][ SZ],1,XZ_p,PEs,1, &val[SX][ EY]  [SZ],1,XZ_p,PEn,
      1,commXY,status);
}
```

3-D Parallelization (Cubic)



- Superior scalability over the other two cases.
- Requires more than 8 processors minimum.
- Increasing communication load (but with smaller data saize)
- Super-linear speedup is possible when data size is small enough – sitting on cache.

main.f90 (3-D Parallel)

```
CALL MPI_CART_CREATE (MPI_COMM_WORLD,…,commXYZ,ierr)
CALL MPI_COMM_RANK (CommXYZ,myPE,ierr)
CALL MPI_CART_COORDS (CommXYZ,myPE,NDIMS,CRDS,ierr)

CALL MPI_CART_SHIFT (CommXYZ,2,1,PEw,PEe,ierr)
CALL MPI_CART_SHIFT (CommXYZ,1,1,PEs,PEn,ierr)
CALL MPI_CART_SHIFT (CommXYZ,0,1,PEb,PEt,ierr)
!-------------------------------------------------------
CALL MPI_TYPE_VECTOR
     (cnt_yz,block_yz,strd_yz,MPI_DOUBLE_PRECISION,YZ_p,ierr)
CALL_MPI_TYPE_COMMIT (YZ_p,ierr)

CALL MPI_TYPE_VECTOR
     (cnt_xz,block_xz,strd_xz,MPI_DOUBLE_PRECISION,XZ_p,ierr)
CALL MPI_TYEP_COMMIT (XZ_p,ierr)

 CALL MPI_TYPE_CONTIGUOUS (cnt_xy,MPI_DOUBLE_PRECISION,XY_p,ierr)
 CALL MPI_TYPE_COMMIT (XY_p,ierr)
```

Main.C (3-D Parallel)

```
ierr= MPI_Cart_create (MPI_COMM_WORLD,…,commXYZ);
ierr= MPI_COMM_RANK (CommXYZ,myPE);
ierr= MPI_CART_COORDS (CommXYZ,myPE,NDIMS,CRDS);

ierr= MPI_Cart_shift (CommXYZ,2,1,PEw,PEe);
ierr=MPI_Cart_shift (CommXYZ,1,1,PEs,PEn );
ierr= MPI_Cart_shift (CommXYZ,0,1,PEb,PEt );
//-------------------------------------------------------
ierr=MPI_Type_vector (cnt_yz,block_yz,strd_yz,MPI_DOUBLE_PRECISION,YZ_p);
ierr=MPI_Type_commit (YZ_p);

ierr=MPI_Type_vector(cnt_xz,block_xz,strd_xz,MPI_DOUBLE_PRECISION,XZ_p);
ierr=MPI_Type_commit ( XZ_p);

 ierr=MPI_Type_contiguous(cnt_xy,MPI_DOUBLE_PRECISION,XY_p);
 ierr=MPI_Type_commit (XY_p);
```

exchange.f90 (3-D Parallel)

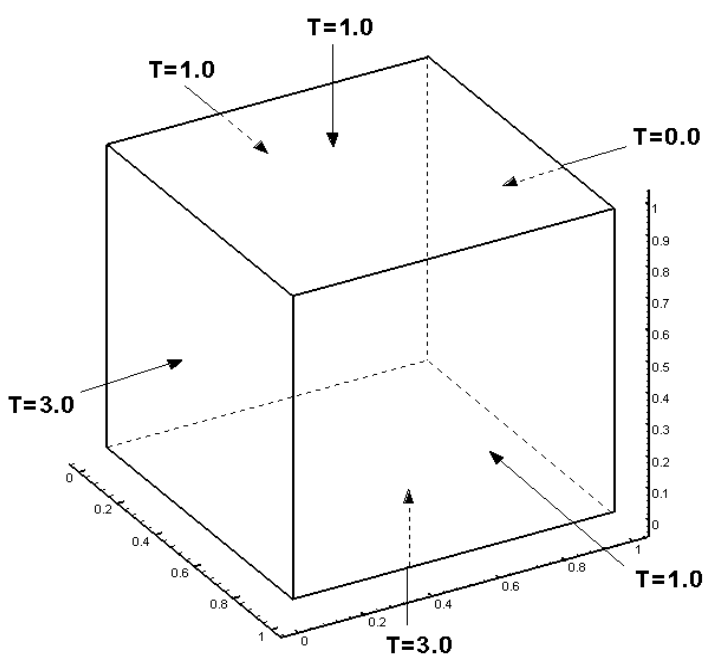
```
!-----------------------------------------------------------------------
      INCLUDE 'mpif.h'
      DOUBLE PRECISION,INTENT(inout) :: VAL(SX:EX,SY:EY,SZ:EZ)
      INTEGER :: status(MPI_STATUS_SIZE)
!-----------------------------------------------------------------------
!---LEFT&RIGHT (Y-Z plane)
      CALL MPI_Sendrecv(val(EX1, SY, SZ),1,YZ_p,PEe,2, &
                        val(SX , SY, SZ),1,YZ_p,PEw,2,commXYZ,status,ierr)

      CALL MPI_Sendrecv(val(SX1, SY, SZ),1,YZ_p,PEw,3, &
                        val(EX , SY, SZ),1,YZ_p,PEe,3,commXYZ,status,ierr)
!---NORTH&SOUTH (X-Z plane)
      CALL MPI_Sendrecv(val(SX, EY1, SZ),1,XZ_p,PEn,0, &
                        val(SX, SY,  SZ),1,XZ_p,PEs,0,commXYZ,status,ierr)

      CALL MPI_Sendrecv(val(SX, SY1, SZ),1,XZ_p,PEs,1, &
                        val(SX, EY,  SZ),1,XZ_p,PEn,1,commXYZ,status,ierr)
!----TOP&BOTTOM (X-Y plane)
      CALL MPI_Sendrecv(val(SX, SY, EZ1),1,XY_p,PEt,4, &
                          val(SX, SY, SZ),1,XY_p,PEb,4,commXYZ,status,ierr)

      CALL MPI_Sendrecv(val(SX, SY, SZ1),1,XY_p,PEb,5, &
                          val(SX, SY, EZ),1,XY_p,PEt,5,commXYZ,status,ierr)
```

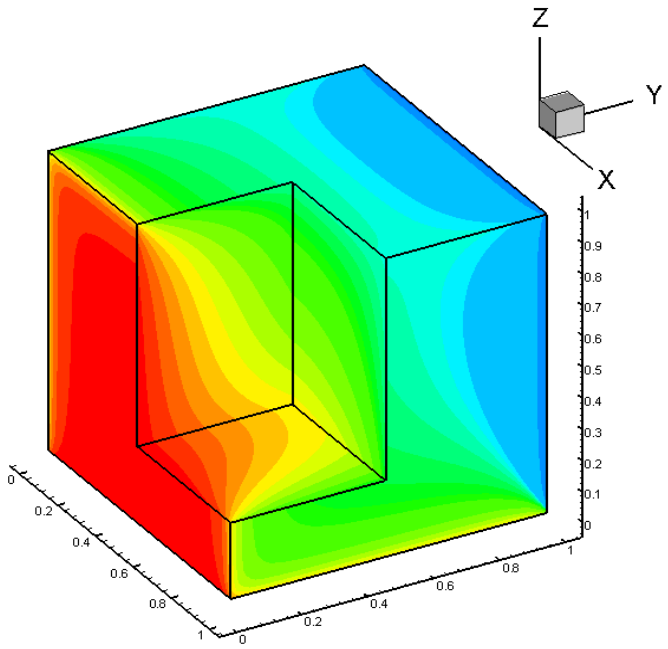
Exchange.C (3-D Parallel)

```
//---------------------------------------------------------------------
      #include "mpi.h"

      void EXCHANGE(double ***val, MPI_Comm commZ)
      {
         int *status = (int) malloc (MPI_STATUS_SIZE*sizeof(int));
//---------------------------------------------------------------------
//---LEFT&RIGHT (Y-Z plane)
      ierr=MPI_Sendrecv(&val[EX1][ SY][ SZ],1,YZ_p,PEe,2, &val[SX ][ SY][ SZ],1,YZ_p,PEw,2,commXYZ,status);
      ierr=MPI_Sendrecv(&val[SX1][ SY][ SZ],1,YZ_p,PEw,3, &val[EX ][ SY][ SZ],1,YZ_p,PEe,3,commXYZ,status);

//---NORTH&SOUTH (X-Z plane)
      ierr=MPI_Sendrecv(&val[SX][ EY1][SZ],1,XZ_p,PEn,0, &val[SX][ SY][  SZ],1,XZ_p,PEs,0,commXYZ,status);
      ierr=MPI_Sendrecv(&val[SX][ SY1][ SZ],1,XZ_p,PEs,1, &val[SX][ EY][  SZ],1,XZ_p,PEn,1,commXYZ,status);

//----TOP&BOTTOM (X-Y plane)
      ierr=MPI_Sendrecv(&val[SX][ SY][ EZ1],1,XY_p,PEt,4, &val[SX][ SY][ SZ],1,XY_p,PEb,4,commXYZ,status);
      ierr=MPI_Sendrecv(&val[SX][ SY][ SZ1],1,XY_p,PEb,5, &val[SX][ SY][ EZ],1,XY_p,PEt,5,commXYZ,status);
```

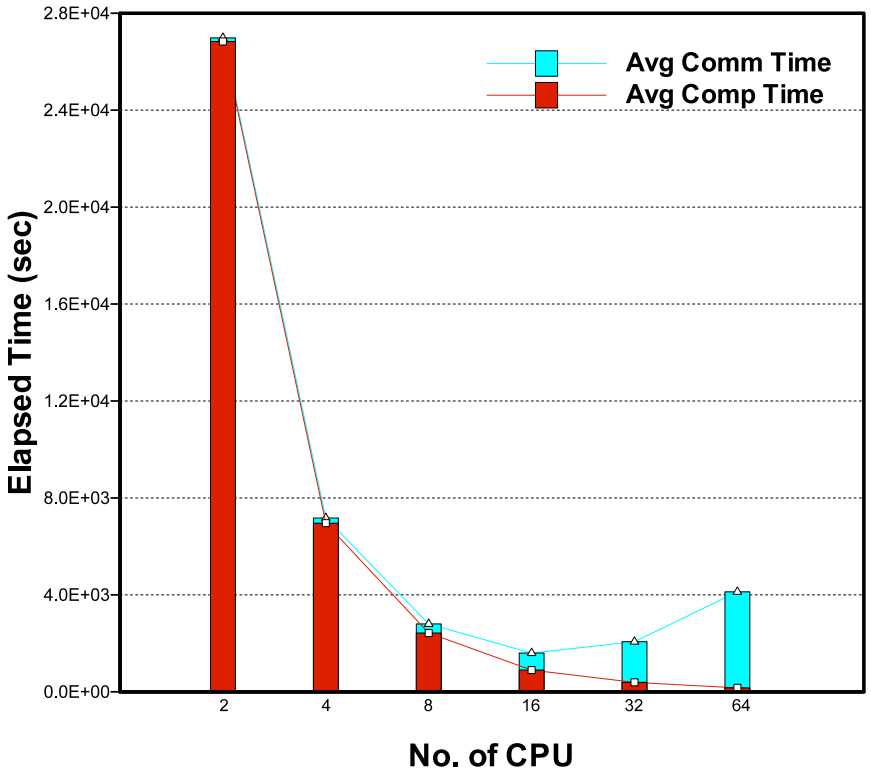
Scalability on 3D Heat Conduction Model

- 256^3 mesh size (max. grid size for one local node)
- Runs up to 10,000 time steps, 100 iteration at each time step



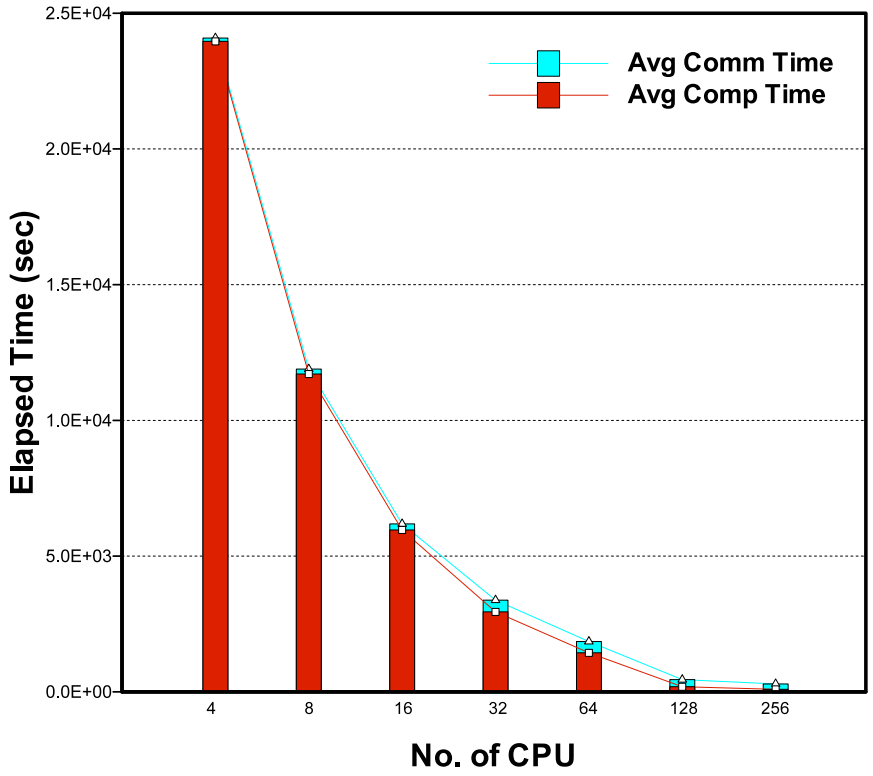


Scalability : 1-D



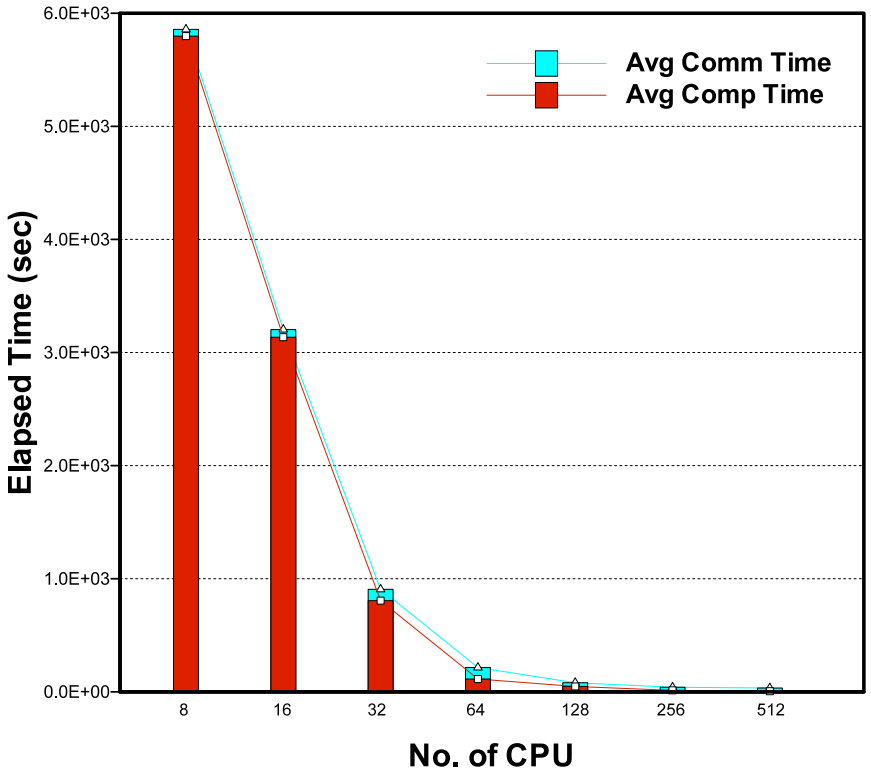
- Good Scalability up to small number of processors (16)
- After the choke point, communication overhead becomes dominant.
- Performance degrades with large number of processors

Scalability: 2-D



- Strong Scalability up to large number of processors
- Actual runtime larger than 1-D case in the case of small number of processors
- Need to be careful with the direction/coordination of domain decomposition.

Scalability: 3-D



- Superior scalability behavior over the other two cases
- No choke point observed up to 512 processors
- Communication overhead ignorable compared to total runtime.

Summary

- 1-D decomposition is OK for small application size, but has communication overhead problem when the size increases
- 2-D shows strong scaling behavior, but need to be careful when apply due to influences from numerical solvers' characteristics.
- 3-D demonstrates superior scalability over the other two, observed superlinear case due to hardware architecture.
- There is no one-size-fits-all magic solution. In order to get the best scalability & application performance, the MPI algorithms, application characteristics, and hardware architectures should be in harmony for the best possible solution.