

**CSE 590: Special Topics Course
(Supercomputing)**

**Lecture 6
(Analyzing Distributed Memory Algorithms)**

**Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2012**

2D Heat Diffusion

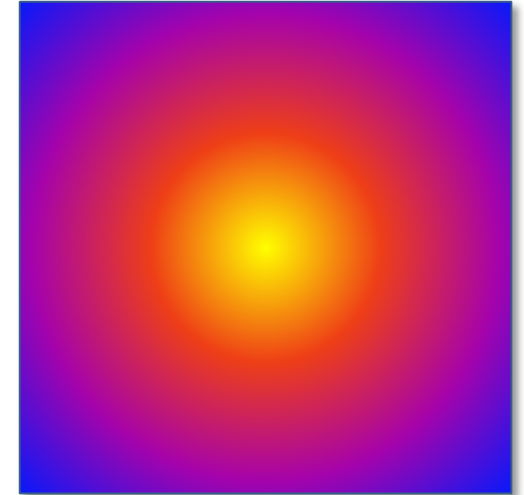
Let $h_t(x, y)$ be the heat at point (x, y) at time t .

Heat Equation

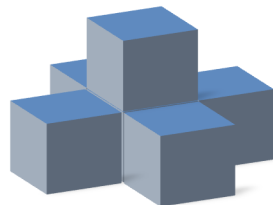
$$\frac{\partial h}{\partial t} = \alpha \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right), \quad \alpha = \text{thermal diffusivity}$$

Update Equation (on a discrete grid)

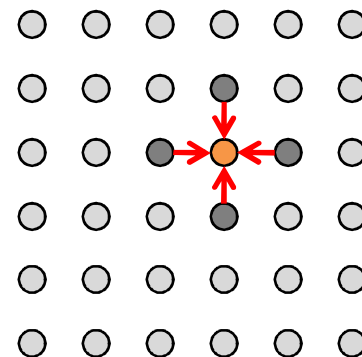
$$\begin{aligned} h_{t+1}(x, y) = & h_t(x, y) \\ & + c_x(h_t(x + 1, y) - 2h_t(x, y) + h_t(x - 1, y)) \\ & + c_y(h_t(x, y + 1) - 2h_t(x, y) + h_t(x, y - 1)) \end{aligned}$$



2D 5-point Stencil



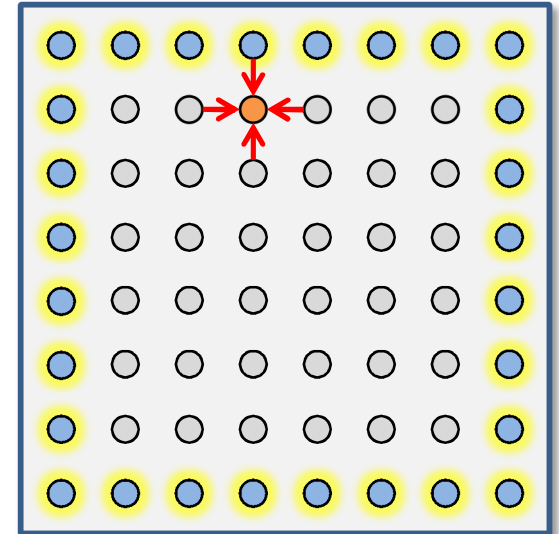
↑
time



Standard Serial Implementation

Implementation Tricks

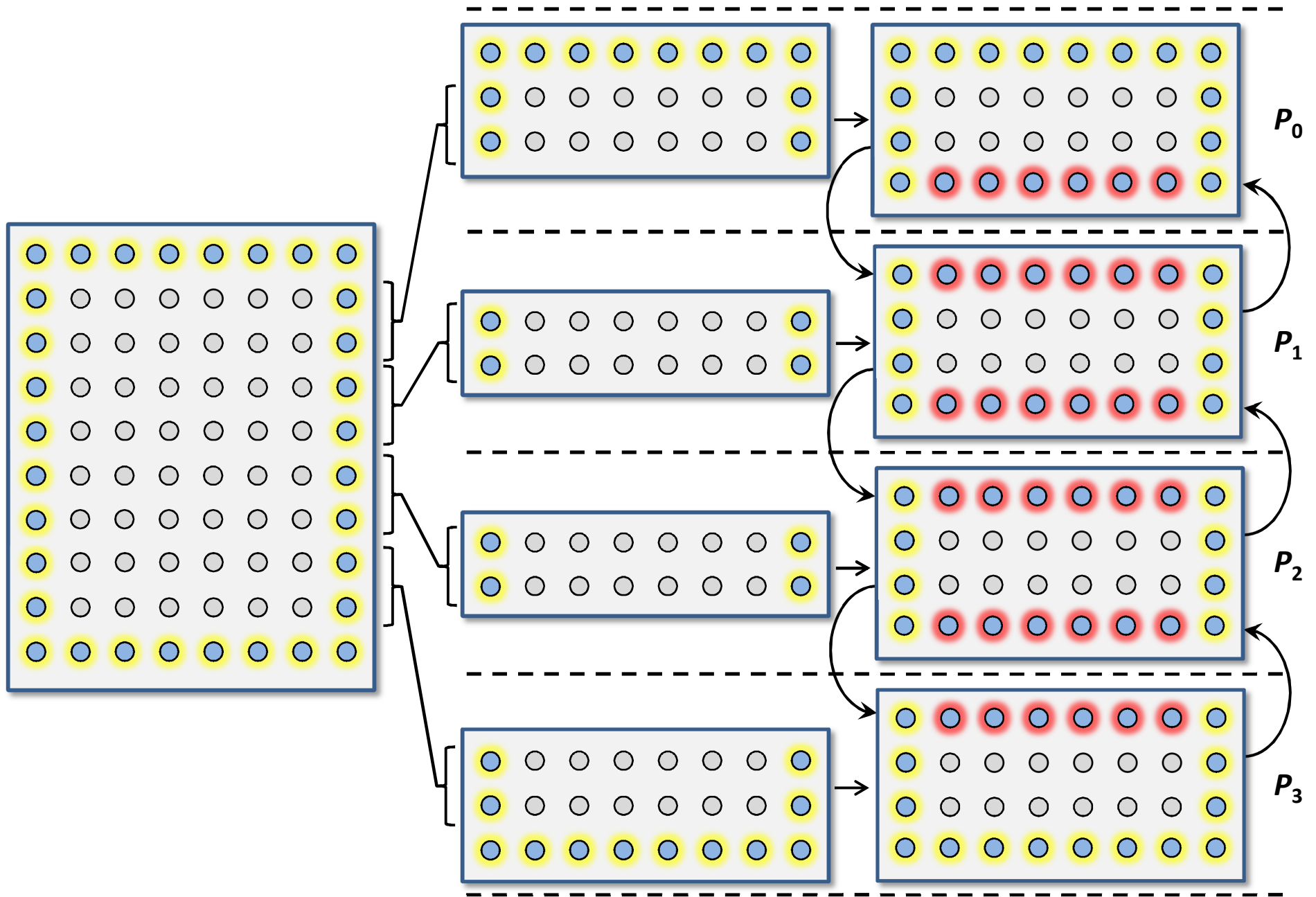
- Reuse storage for odd and even time steps
- Keep a halo of ghost cells around the array with boundary values



```
for ( int t = 0; t < T; ++t )
{
  for ( int x = 1; x <= X; ++x )
    for ( int y = 1; y <= Y; ++y )
      g[x][y] = h[x][y]
        + cx * ( h[x+1][y] - 2 * h[x][y] + h[x-1][y] )
        + cy * ( h[x][y+1] - 2 * h[x][y] + h[x][y-1] );

  for ( int x = 1; x <= X; ++x )
    for ( int y = 1; y <= Y; ++y )
      h[x][y] = g[x][y];
}
```

One Way of Parallelization



MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```

leave enough space
for ghost cells

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```

downward send and upward receive

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 ) { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                     MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 ) MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 ) MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```

upward send and
downward receive

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y; ++y )
            h[x][y] = g[x][y];
}
```

in addition to the ghost rows exclude
the two outermost interior rows

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

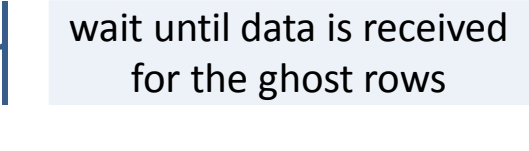
    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```



wait until data is received
for the ghost rows

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

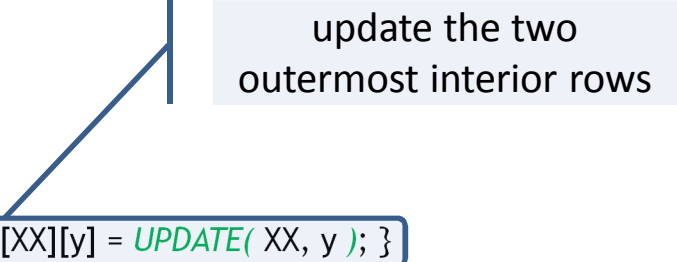
    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )   MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )   MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```



update the two
outermost interior rows

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```

wait until sending data is complete
so that h can be overwritten

MPI Implementation of 2D Heat Diffusion

```
#define UPDATE( u, v ) ( h[u][v] + cx * ( h[u+1][v] - 2* h[u][v] + h[u-1][v] ) + cy * ( h[u][v+1] - 2* h[u][v] + h[u][v-1] ) )
... ..
... ..
MPI_FLOAT h[ XX + 2 ][ Y + 2 ], g[ XX + 2 ][ Y + 2 ];
MPI_Status stat;
MPI_Request sendreq[ 2 ], recvreq[ 2 ];
... ..
... ..
for ( int t = 0; t < T; ++t )
{
    if ( myrank < p - 1 ) { MPI_Isend( h[ XX ], Y, MPI_FLOAT, myrank + 1, 2 * t, MPI_COMM_WORLD , & sendreq[ 0 ] );
                          MPI_Irecv( h[ XX + 1 ], Y, MPI_FLOAT, myrank + 1, 2 * t + 1, MPI_COMM_WORLD , & recvreq[ 0 ] ); }

    if ( myrank > 0 )     { MPI_Isend( h[ 1 ], Y, MPI_FLOAT, myrank - 1, 2 * t + 1, MPI_COMM_WORLD , & sendreq[ 1 ] );
                          MPI_Irecv( h[ 0 ], Y, MPI_FLOAT, myrank - 1, 2 * t , MPI_COMM_WORLD , & recvreq[ 1 ] ); }

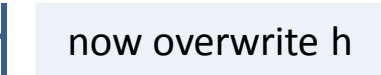
    for ( int x = 2; x < XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            g[x][y] = UPDATE( x, y );

    if ( myrank < p - 1 ) MPI_Wait( &recvreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &recvreq[ 1 ], &stat );

    for ( int y = 1; y <= Y ; ++y ) { g[1][y] = UPDATE( 1, y ); g[XX][y] = UPDATE( XX, y ); }

    if ( myrank < p - 1 ) MPI_Wait( &sendreq[ 0 ], &stat );
    if ( myrank > 0 )    MPI_Wait( &sendreq[ 1 ], &stat );

    for ( int x = 1; x <= XX; ++x )
        for ( int y = 1; y <= Y ; ++y )
            h[x][y] = g[x][y];
}
```



now overwrite h

Analysis of the MPI Implementation of Heat Diffusion

Let the dimension of the 2D grid be $n_X \times n_Y$, and suppose we execute n_T time steps. Let p be the number of processors, and suppose the grid is decomposed along X direction.

The computation cost in each time step is clearly $\frac{n_X n_Y}{p}$. Hence, the total computation cost, $t_{comp} = \frac{n_T n_X n_Y}{p}$.

All processors except processors 0 and $p - 1$ send two rows and receive two rows each in every time step. Processors 0 and $p - 1$ send and receive only one row each. Hence, the total communication cost, $t_{comm} = 4n_T(t_s + n_Y t_w)$, where t_s is the startup time of a message and t_w is the per-word transfer time.

Thus $T_p = t_{comp} + t_{comm} = \frac{n_T n_X n_Y}{p} + 4n_T(t_s + n_Y t_w)$,

and $T_1 = n_T n_X n_Y$.

Naïve Matrix Multiplication

$$z_{ij} = \sum_{k=1}^n x_{ik} y_{kj}$$

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Iter-MM(X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n *do*
2. *for* $j \leftarrow 1$ *to* n *do*
3. *for* $k \leftarrow 1$ *to* n *do*
4. $z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj}$

Naïve Matrix Multiplication

$$\boxed{z_{ij} = \sum_{k=1}^n x_{ik} y_{kj}}$$
$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Suppose we have $p = n \times n$ processors, and processor P_{ij} is responsible for computing z_{ij} .

One master processor initially holds both X and Y , and sends all x_{ik} and y_{kj} for $k = 1, 2, \dots, n$ to each processor P_{ij} . One-to-all Broadcast is a bad idea as each processor requires a different part of the input.

Each P_{ij} computes z_{ij} and sends back to master.

Thus $t_{comp} = 2n$, and $t_{comm} = n^2(t_s + 2nt_w) + n^2(t_s + t_w)$.

Hence, $T_p = t_{comp} + t_{comm} = 2n + n^2(2t_s + t_w + 2nt_w)$.

Total work, $T_1 = 2n^3$.

Naïve Matrix Multiplication

$$\boxed{z_{ij} = \sum_{k=1}^n x_{ik} y_{kj}}$$
$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Observe that row i of X will be required by all $P_{i,j}$, $1 \leq j \leq n$. So that row can be broadcast to the group $\{P_{i,1}, P_{i,2}, \dots, P_{i,n}\}$ of size n .

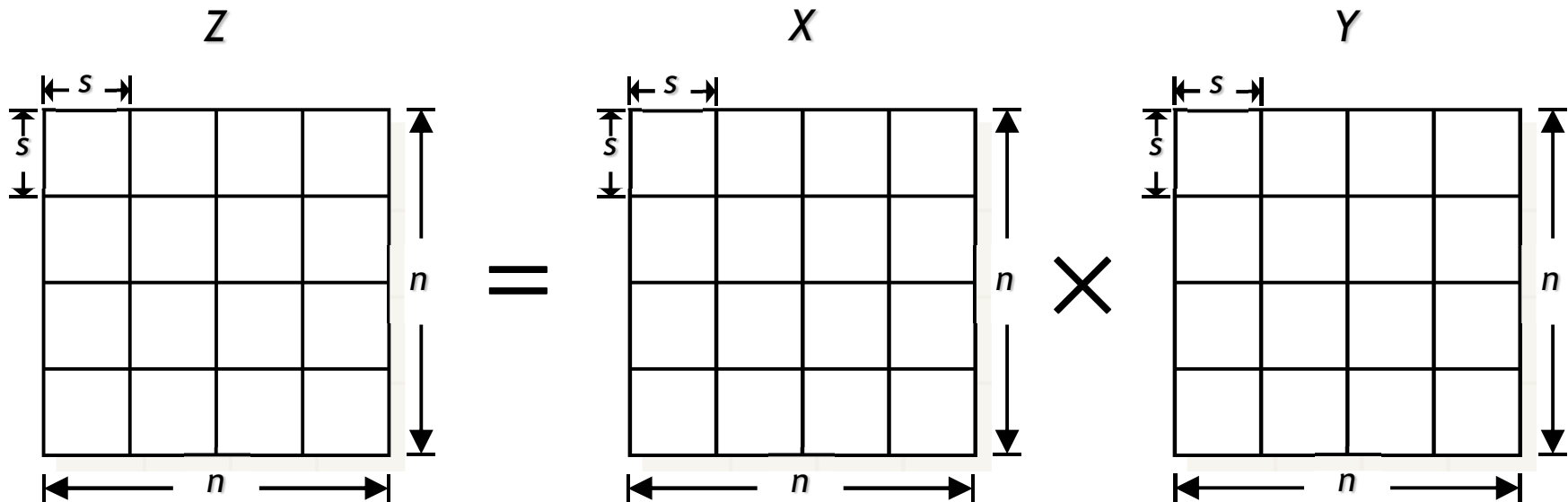
Similarly, for other rows of X , and all columns of Y .

The communication complexity of broadcasting m units of data to a group of size n is $(t_s + mt_w) \log n$.

As before, each P_{ij} computes z_{ij} and sends back to master.

Hence, $t_{comm} = 2n(t_s + nt_w) \log n + n^2(t_s + t_w)$.

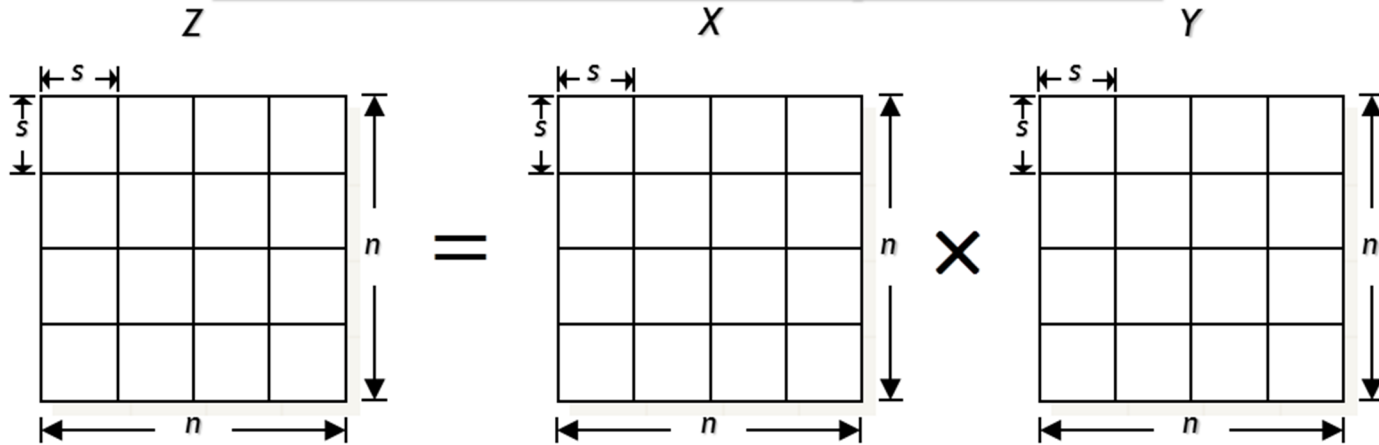
Block Matrix Multiplication



Block-MM(X, Y, Z, n)

1. *for* $i \leftarrow 1$ *to* n / s *do*
2. *for* $j \leftarrow 1$ *to* n / s *do*
3. *for* $k \leftarrow 1$ *to* n / s *do*
4. *Iter-MM*($X_{ik}, Y_{kj}, Z_{ij}, s$)

Block Matrix Multiplication



Suppose $p = \frac{n}{s} \times \frac{n}{s}$, and processor P_{ij} computes block Z_{ij} .

One master processor initially holds both X and Y , and sends all blocks X_{ik} and Y_{kj} for $k = 1, 2, \dots, \frac{n}{s}$ to each processor P_{ij} .

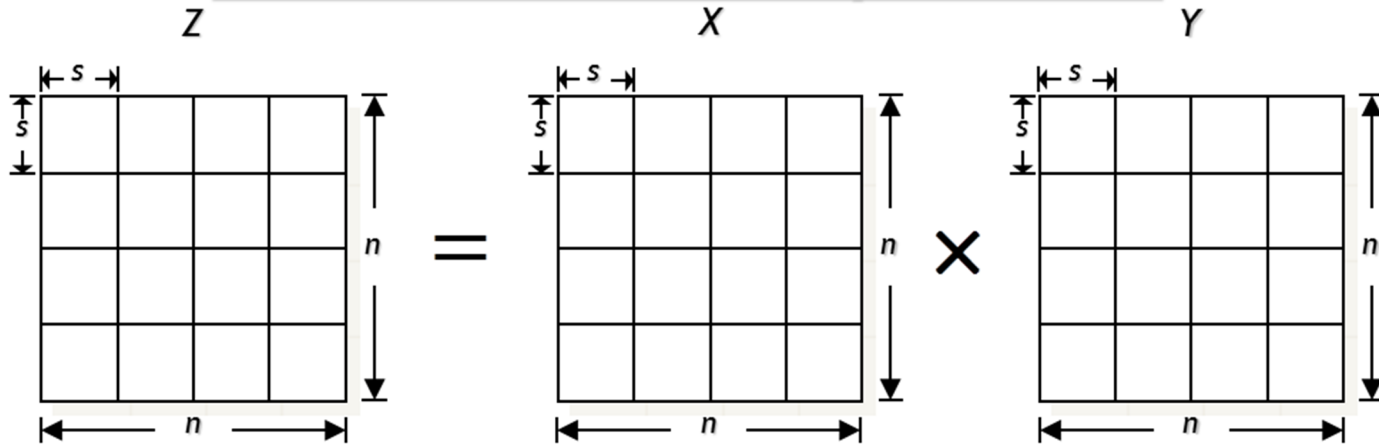
Thus $t_{comp} = \frac{n}{s} (2s^3 + s^2) = O(ns^2)$,

and $t_{comm} = \left(\frac{n}{s}\right)^2 (2(t_s + nst_w) + (t_s + s^2t_w))$. (w/o broadcast)

For $s = \sqrt{n}$, $t_{comp} = O(n^2)$, and $t_{comm} = O(nt_s + n^{2.5}t_w)$

For $s = n^{\frac{2}{3}}$, $t_{comp} = O(n^{2+\frac{1}{3}})$, and $t_{comm} = O(n^{\frac{2}{3}}t_s + n^{2+\frac{1}{3}}t_w)$

Block Matrix Multiplication



Now consider one-to-group broadcasting.

Block row i of X , i.e., blocks X_{ik} for $k = 1, 2, \dots, \frac{n}{s}$, will be required by $\frac{n}{s}$ different processors, i.e., processors P_{ij} for $j = 1, 2, \dots, \frac{n}{s}$.

Similarly, for other block rows of X , and all block columns of Y .

As before, each P_{ij} computes block Z_{ij} and sends back to master.

Hence, $t_{comm} = \frac{n}{s} (t_s + nst_w) \log \left(\frac{n}{s} \right) + \left(\frac{n}{s} \right)^2 (t_s + s^2 t_w)$.

Recursive Matrix Multiplication

Par-Rec-MM (X, Y, Z, n)

1. *if* $n = 1$ *then* $Z \leftarrow Z + X \cdot Y$

2. *else*

3. *in parallel do*

Par-Rec-MM ($X_{11}, Y_{11}, Z_{11}, n / 2$)

Par-Rec-MM ($X_{11}, Y_{12}, Z_{12}, n / 2$)

Par-Rec-MM ($X_{21}, Y_{11}, Z_{21}, n / 2$)

Par-Rec-MM ($X_{21}, Y_{12}, Z_{22}, n / 2$)

end do

4. *in parallel do*

Par-Rec-MM ($X_{12}, Y_{21}, Z_{11}, n / 2$)

Par-Rec-MM ($X_{12}, Y_{22}, Z_{12}, n / 2$)

Par-Rec-MM ($X_{22}, Y_{21}, Z_{21}, n / 2$)

Par-Rec-MM ($X_{22}, Y_{22}, Z_{22}, n / 2$)

end do

Assuming t_s and t_w are constants,

$$t_{comm}(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8t_{comm}\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} \end{cases}$$
$$= \Theta(n^3) \quad [\text{MT Case 1}]$$

Communication cost is too high!

Recursive Matrix Multiplication

Par-Rec-MM (X, Y, Z, n)

1. *if* $n = 1$ *then* $Z \leftarrow Z + X \cdot Y$

2. *else*

3. *in parallel do*

Par-Rec-MM (X_{11} , Y_{11} , Z_{11} , $n / 2$)

Par-Rec-MM (X_{11} , Y_{12} , Z_{12} , $n / 2$)

Par-Rec-MM (X_{21} , Y_{11} , Z_{21} , $n / 2$)

Par-Rec-MM (X_{21} , Y_{12} , Z_{22} , $n / 2$)

end do

4. *in parallel do*

Par-Rec-MM (X_{12} , Y_{21} , Z_{11} , $n / 2$)

Par-Rec-MM (X_{12} , Y_{22} , Z_{12} , $n / 2$)

Par-Rec-MM (X_{22} , Y_{21} , Z_{21} , $n / 2$)

Par-Rec-MM (X_{22} , Y_{22} , Z_{22} , $n / 2$)

end do

But with a $s \times s$ base case,

$$t_{comm}(n) = \begin{cases} \Theta(1), & \text{if } n \leq s, \\ 8t_{comm}\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} \end{cases}$$

$$= \Theta\left(\frac{n^3}{s}\right)$$

Parallel running time,

$$t_{comp}(n) = \Theta\left(\frac{n^3}{p} + ns^2\right) \quad (\text{how?})$$

For $s = n^{\frac{2}{3}}$,

$$t_{comp} = O\left(\frac{n^3}{p} + n^{2+\frac{1}{3}}\right),$$

$$\text{and } t_{comm} = O\left(n^{2+\frac{1}{3}}\right)$$

Cannon's Algorithm

We decompose each matrix into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each.

We number the processors from $P_{0,0}$ to $P_{\sqrt{p}-1, \sqrt{p}-1}$.

Initially, P_{ij} holds A_{ij} and B_{ij} .

We rotate block row i of A to the left by i positions, and block column j of B upward by j positions.

So, P_{ij} now holds $A_{i, j+i}$ and $B_{i+j, j}$.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

(a) Initial alignment of A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

(b) Initial alignment of B

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{2,2}$	$A_{2,3}$	$A_{2,0}$	$A_{2,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{3,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$

(c) A and B after initial alignment

$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{1,2}$	$A_{1,3}$	$A_{1,0}$	$A_{1,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{2,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$

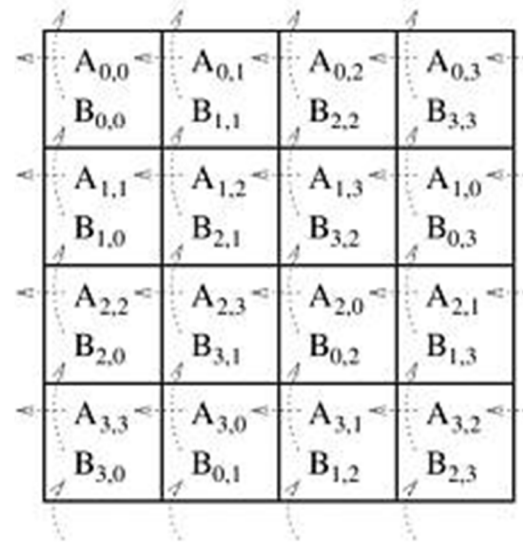
(d) Submatrix locations after first shift

Cannon's Algorithm

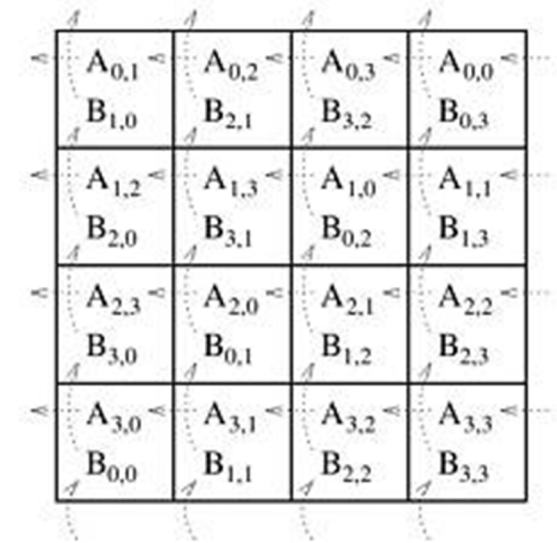
P_{ij} now holds $A_{i,j+i}$ and $B_{i+j,j}$.

P_{ij} multiplies these two submatrices, and adds the result to $C_{i,j}$.

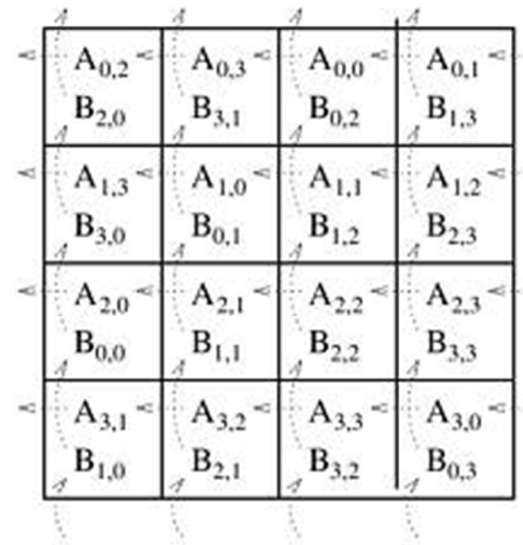
Then in each of the next $\sqrt{p} - 1$ steps, each block row of A is rotated to the left by 1 position, and each block column of B is rotated upward by 1 position. Each P_{ij} adds the product of its current submatrices to $C_{i,j}$.



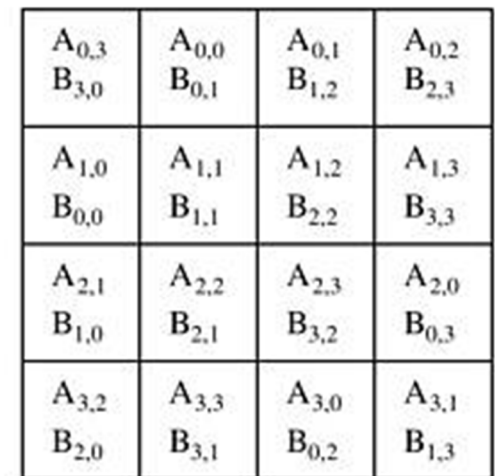
(c) A and B after initial alignment



(d) Submatrix locations after first shift



(e) Submatrix locations after second shift



(f) Submatrix locations after third shift

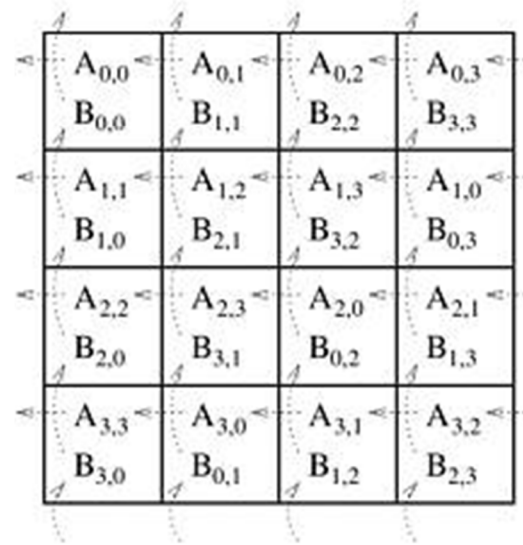
Cannon's Algorithm

Initial arrangement makes $\sqrt{p} - 1$ block rotations of A and B , and one block matrix multiplication per processor.

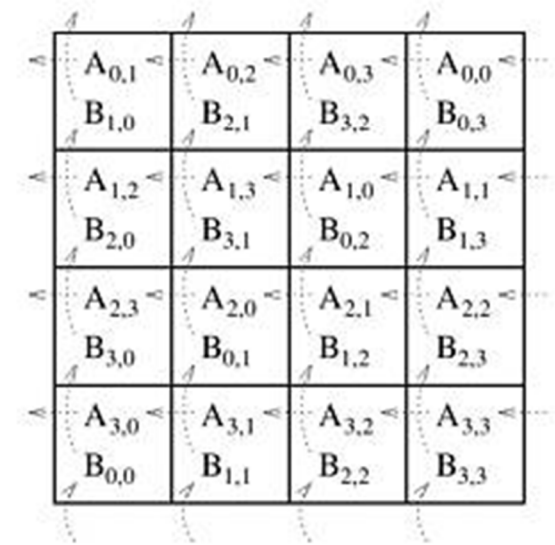
In each of the next $\sqrt{p} - 1$ steps, each processor performs one block matrix multiplication, and sends and receives one block each.

$$t_{comp} = 2\sqrt{p} \left(\frac{n}{\sqrt{p}}\right)^3 = O\left(\frac{n^3}{p}\right),$$

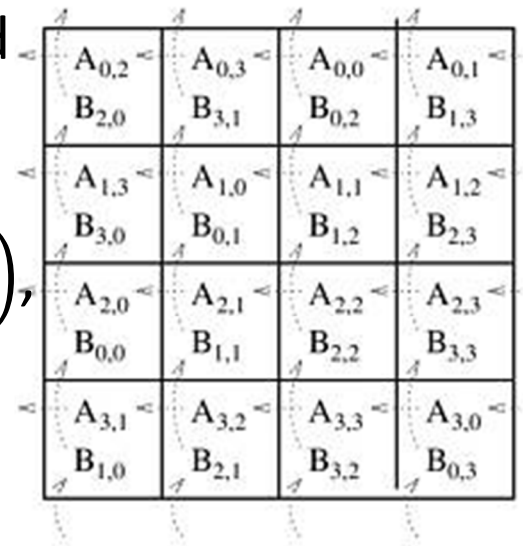
$$t_{comm} = 4(\sqrt{p} - 1) \times \left(t_s + \left(\frac{n}{\sqrt{p}}\right)^2 t_w\right).$$



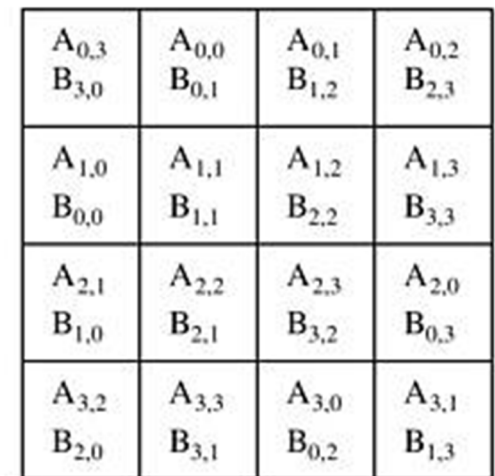
(c) A and B after initial alignment



(d) Submatrix locations after first shift



(e) Submatrix locations after second shift



(f) Submatrix locations after third shift

Cannon's Algorithm

What if initially, one master processor (say, $P_{0,0}$) holds all data (i.e., matrices A and B), and the same processor wants to collect the entire output matrix (i.e., C) at the end?

Processor $P_{0,0}$ initially sends $A_{i,j}$ and $B_{i,j}$ to processor $P_{i,j}$, and at the end processor $P_{i,j}$ sends back $C_{i,j}$ to $P_{0,0}$.

Since there are p processors, and each submatrix has size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, the additional communication complexity:

$$3p \times \left(t_s + \left(\frac{n}{\sqrt{p}} \right)^2 t_w \right) = 3(pt_s + n^2 t_w).$$

So, the communication complexity increases by a factor of \sqrt{p} .