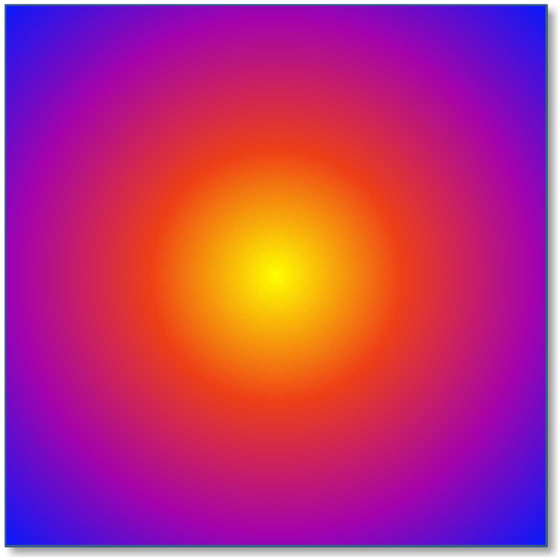
CSE 613: Parallel Programming

Lectures 28 – 29
(Distributed Memory Algorithms: Dense Matrices)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2015

2D Heat Diffusion

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



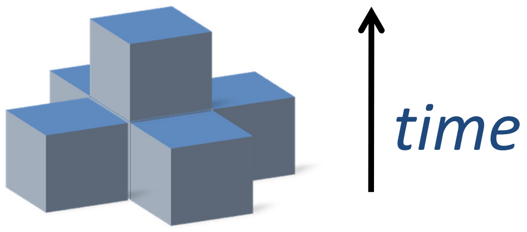
Heat Equation

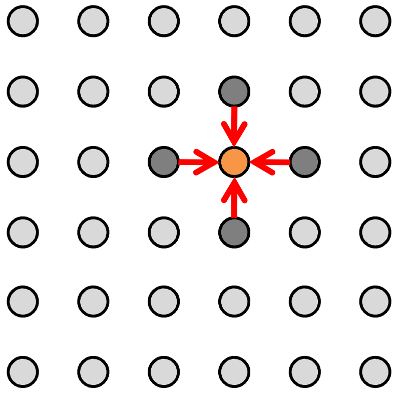
$$\frac{\partial h}{\partial t} = \alpha \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right), \quad \alpha = \text{thermal diffusivity}$$

Update Equation (on a discrete grid)

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

2D 5-point Stencil



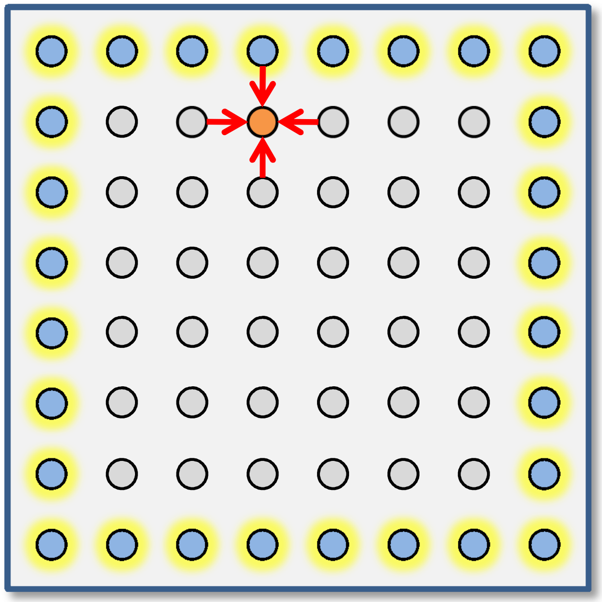


Standard Serial Implementation

Implementation Tricks

– Reuse storage for odd and even time steps

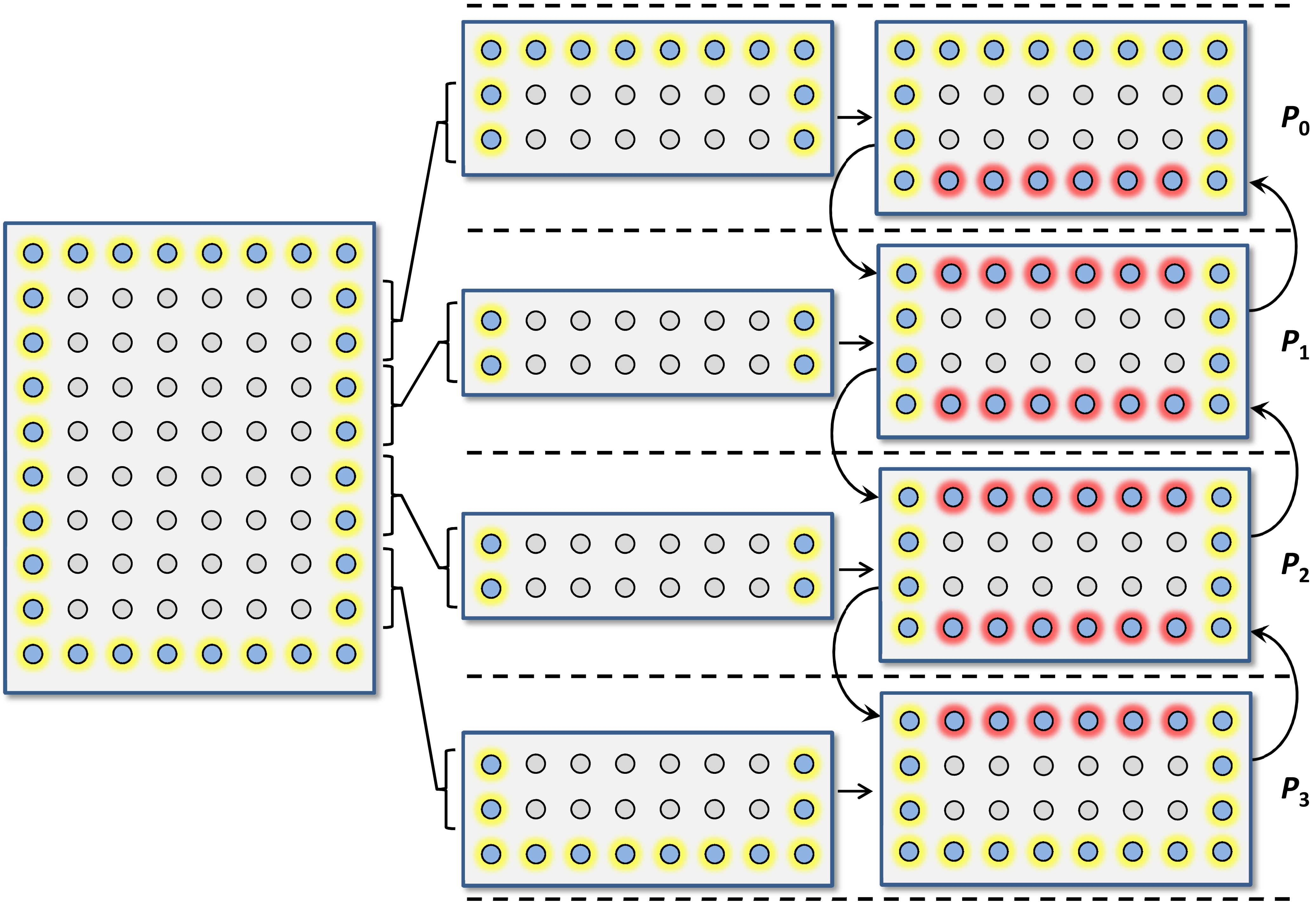
– Keep a halo of ghost cells around the array with boundary values



```
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

downward send and upward receive

```
#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];
}
```

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

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

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

```
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 $P_{i,j}$, $1 \le j \le n$. So that row can be broadcast to the group $\{P_{i,1}, P_{i,2}, \dots, P_{i,n}\}$ of size n.

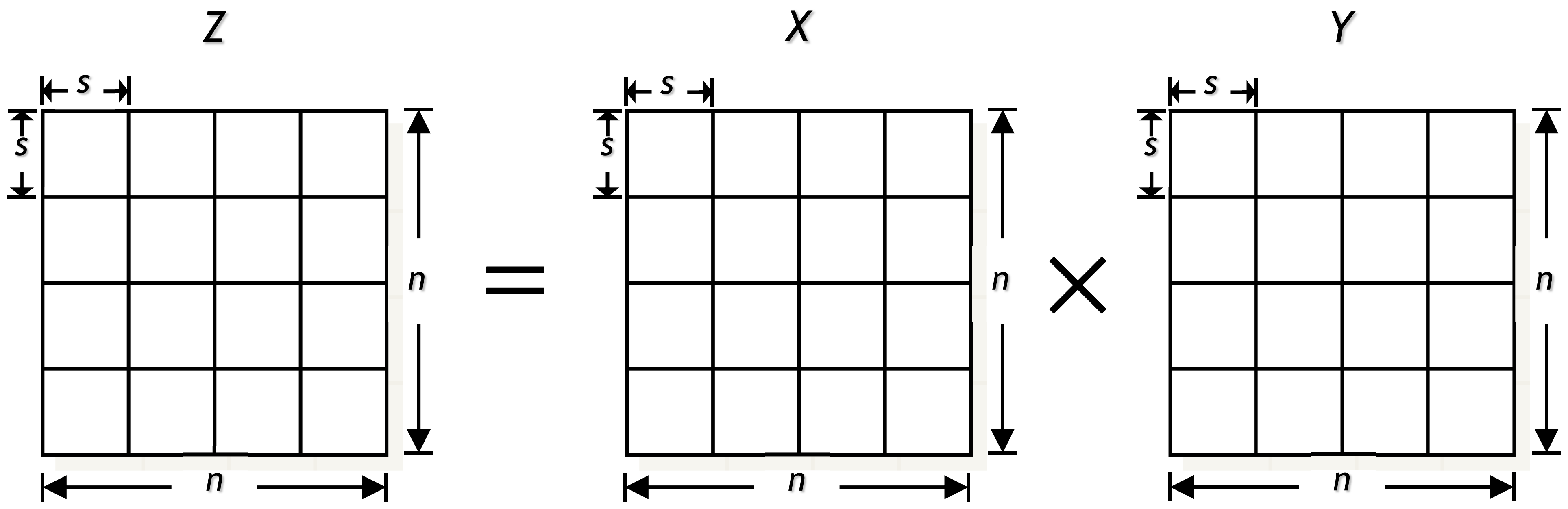
Similarly, for other rows of X, and all columns of Y.

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.

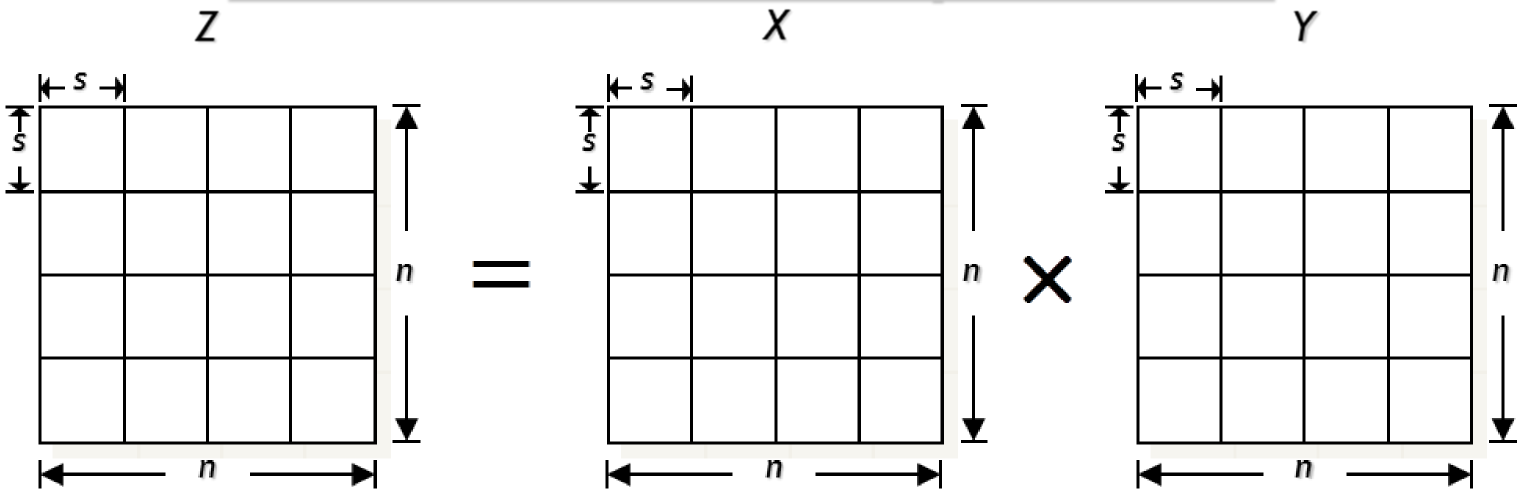
Hence, $t_{comm} = 2n(t_s + nt_w)\log n \ + n^2(t_s + t_w)$.

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 $k = 1, 2, \ldots, \frac{n}{s}$ to each processor P_{ij}.

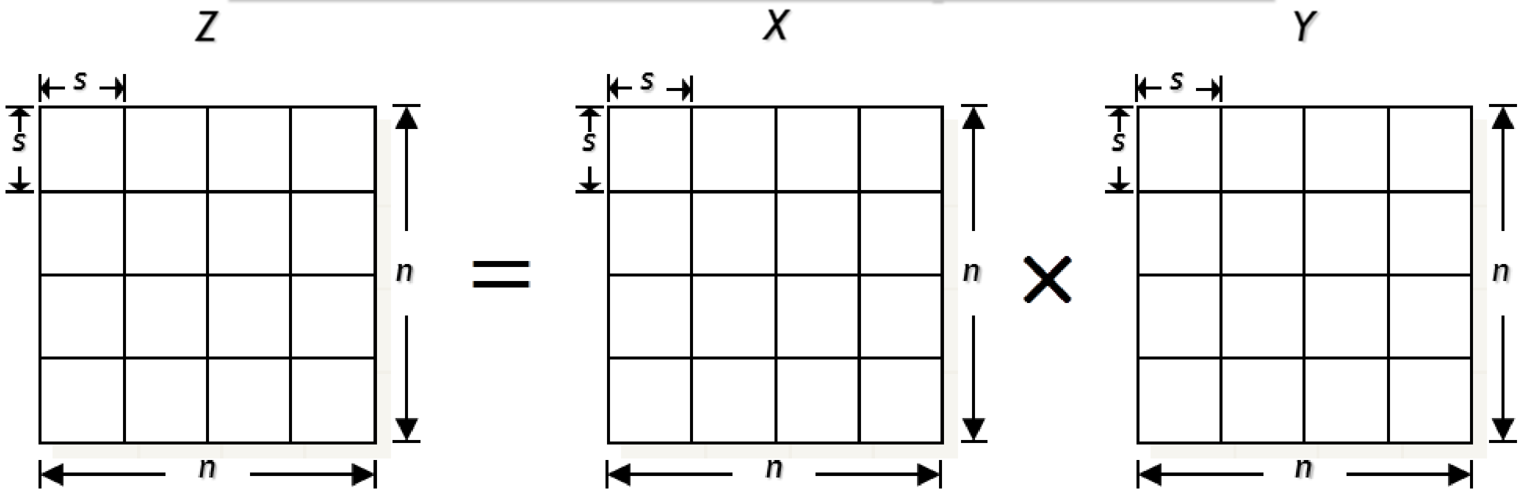
Thus $t_{comp} = \frac{n}{s}(2s^3 + s^2) = \mathrm{O}(ns^2)$,

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

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

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

Block Matrix Multiplication



Now consider one-to-group broadcasting.

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

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 ← Z + X · 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,

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

$$= \Theta(n^3) \qquad \text{[MT Case 1]}$$

Communication cost is too high!

Recursive Matrix Multiplication

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

   1. if n = 1 then Z ← Z + X · 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,

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

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

Parallel running time,

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

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

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

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

Cannon's Algorithm

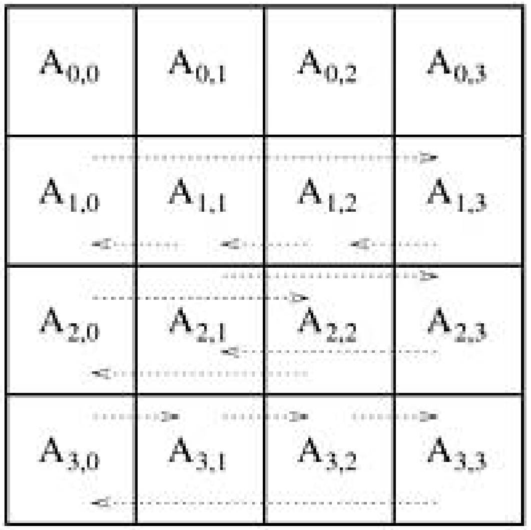
We decompose each matrix into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each.

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}.

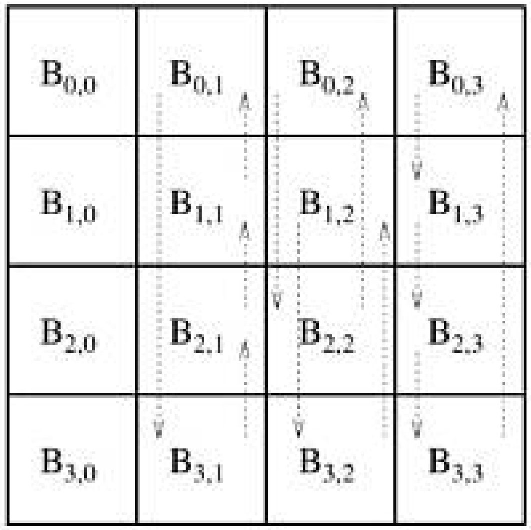
We rotate block row i of A to the left by i positions, and block column j of B upward by j positions.

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



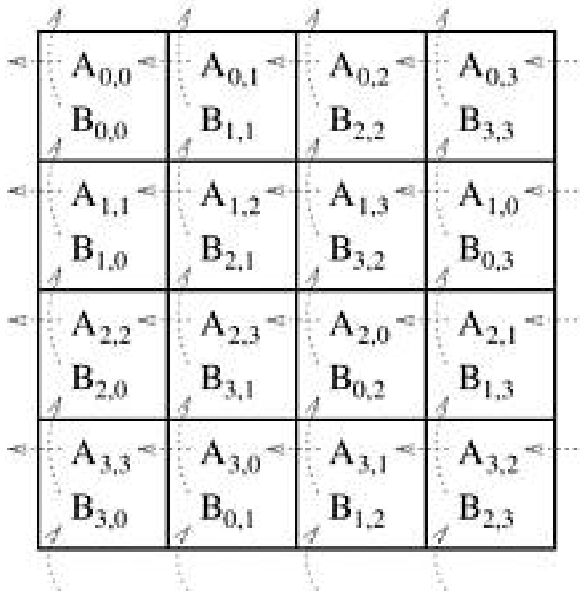
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$

(a) Initial alignment of A



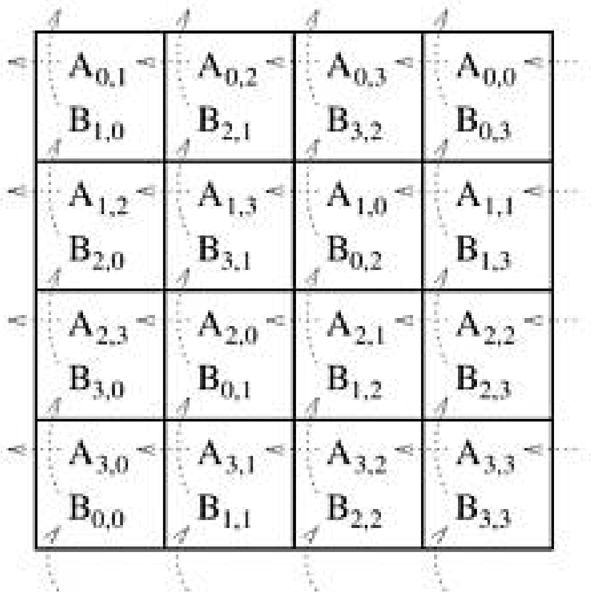
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

(b) Initial alignment of B



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

(c) A and B after initial alignment



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

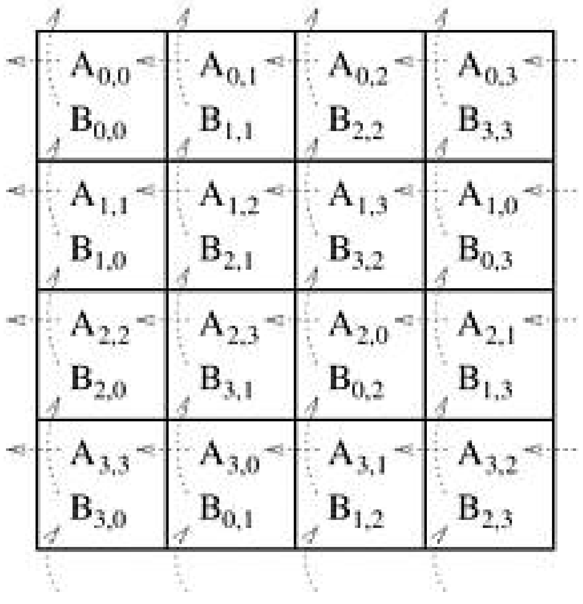
(d) Submatrix locations after first shift

Cannon's Algorithm

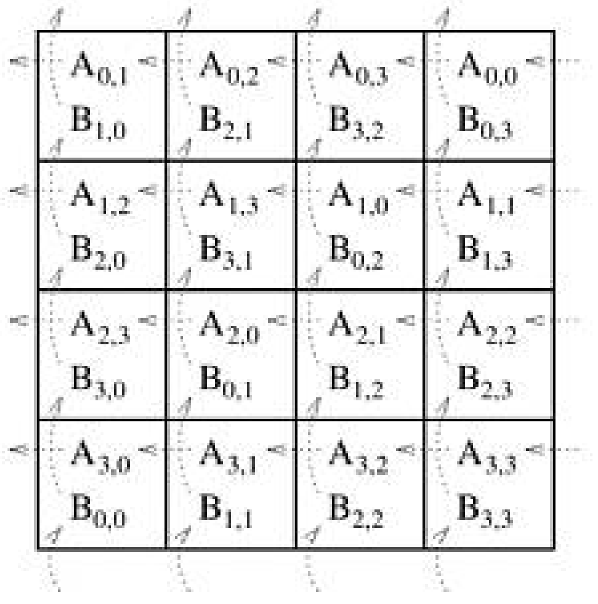
P_{ij} now holds $A_{i,j+i}$ and $B_{i+j,j}$.

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

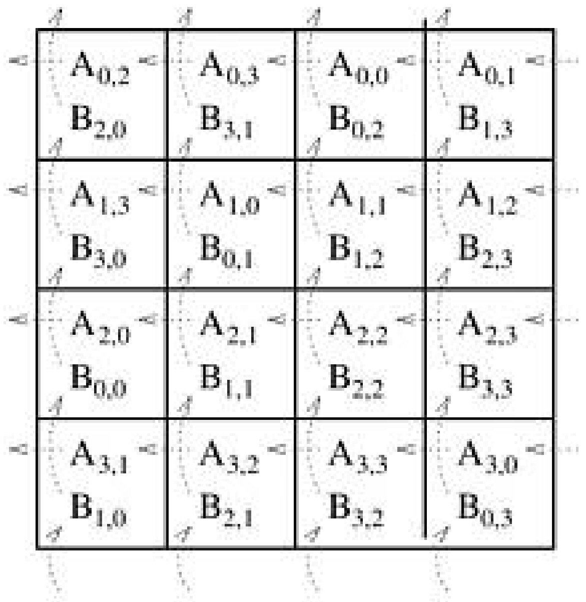
Then in each of the next $\sqrt{p} - 1$ steps, each block row of A is rotated to the left by 1 position, and each block column of B is rotated upward by 1 position. Each P_{ij} adds the product of its current submatrices to $C_{i,j}$.



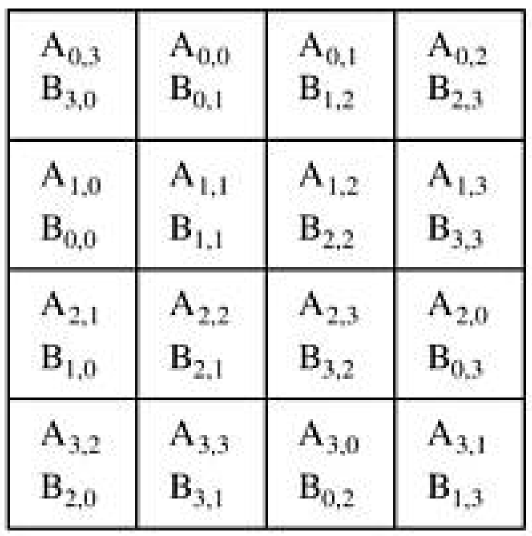
(c) A and B after initial alignment



(d) Submatrix locations after first shift



(e) Submatrix locations after second shift



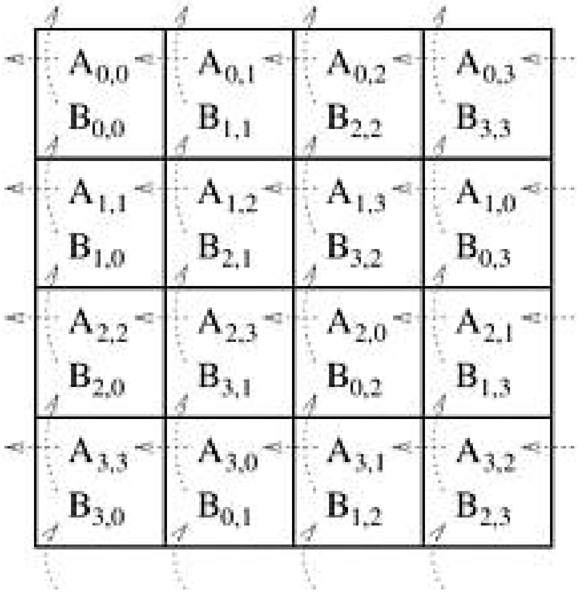
(f) Submatrix locations after third shift

Cannon's Algorithm

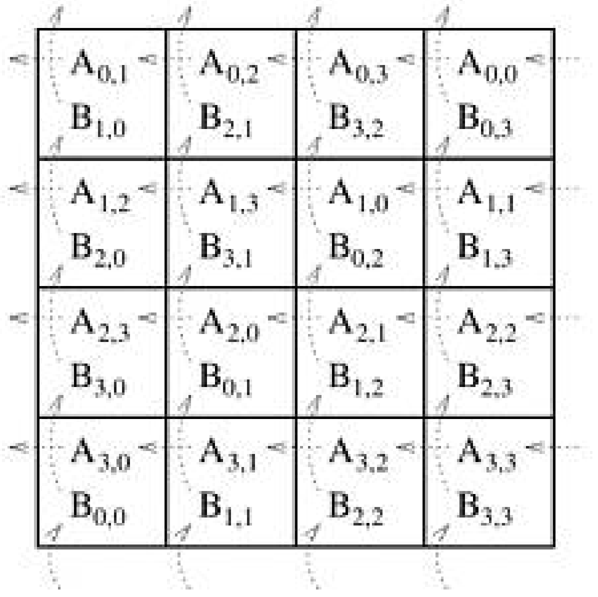
Initial arrangement makes $\sqrt{p}-1$ block rotations of A and B, and one block matrix multiplication per processor.

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

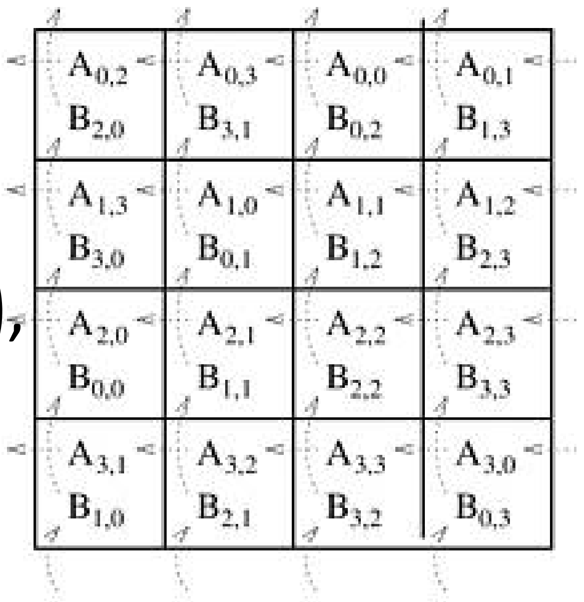
$$t_{comp} = 2\sqrt{p}\left(\frac{n}{\sqrt{p}}\right)^3 = \mathrm{O}\left(\frac{n^3}{p}\right),$$
$$t_{comm} = 4(\sqrt{p}-1) \times \left(t_s + \left(\frac{n}{\sqrt{p}}\right)^2 t_w\right).$$



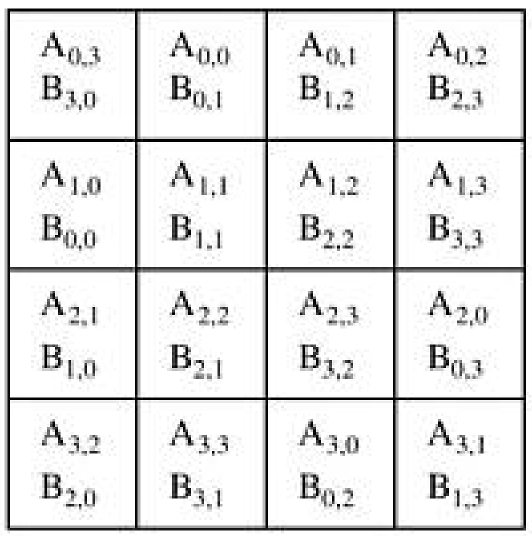
(c) A and B after initial alignment



(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 $\sqrt{p}$.

Floyd-Warshall's All-Pairs Shortest Paths

Let $G = (V, E, w)$ be a weighted directed graph with vertex set $V = \{v_1, v_2, \dots, v_n\}$, edge set E, and weight function w.

The weight of edge $(v_i, v_j) \in E$ is given by $w(v_i, v_j)$.

We construct an $n \times n$ matrix A as follows:

$$A(i,j) = a_{ij} = \begin{cases} 0, & if\ i = j, \\ \infty, & if\left(v_i, v_j\right) \notin E, \\ w\left(v_i, v_j\right), & otherwise. \end{cases}$$

Floyd-Warshall's algorithm takes matrix A as input, and returns another $n \times n$ matrix D as output with

$$D(i,j) = d_{ij} = \text{shortest distance from } v_i \text{ to } v_j \text{ in } G.$$

Floyd-Warshall's All-Pairs Shortest Paths

```
FW-APSP( A, n )
    1. $D^{(0)} \leftarrow A$
    2. for $k \leftarrow 1$ to $n$ do
    3.     for $i \leftarrow 1$ to $n$ do
    4.         for $j \leftarrow 1$ to $n$ do
    5.             $d_{i,j}^{(k)} \leftarrow min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$
    6. return $D^{(n)}$
```

– can be solved using only $\Theta(n^2)$ extra space, e.g., using only two $n \times n$ matrices for storing the values of D

– can be solved in-place in A

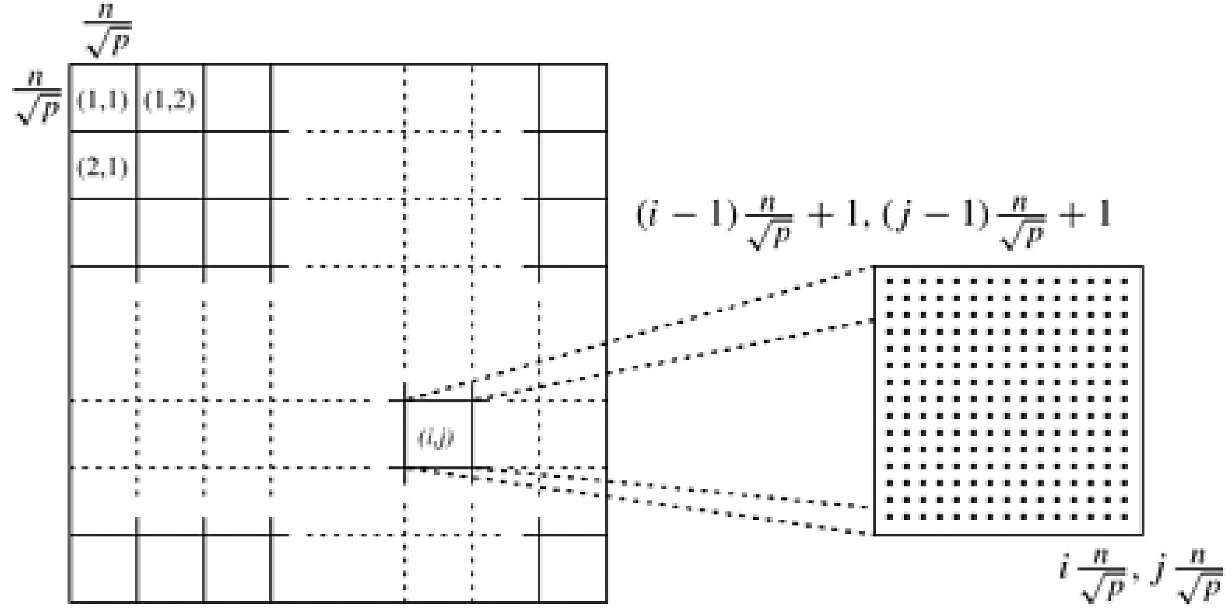
– serial running time is $\Theta(n^3)$

Distributed Memory Implementation

Let p be the number of processing nodes.

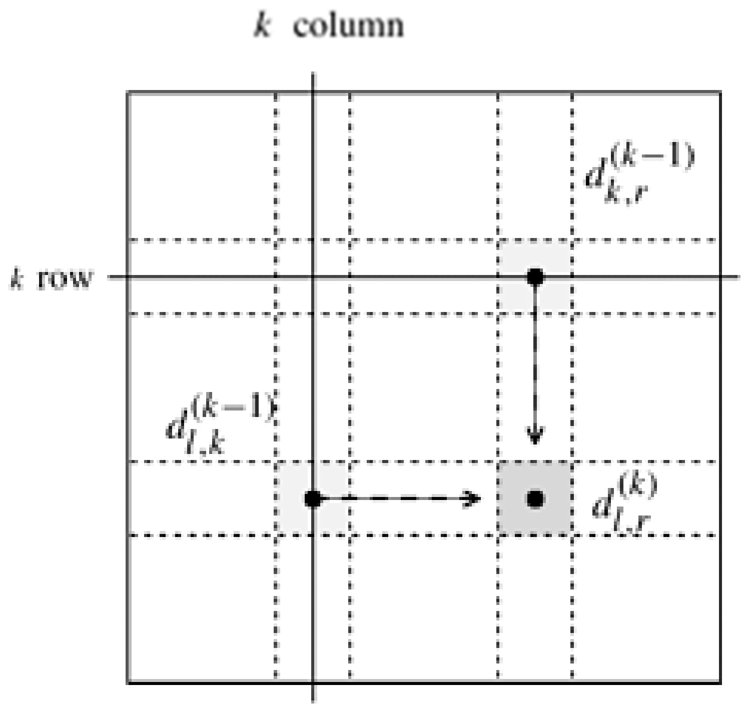
We divide $D^{(k)}$ into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each.

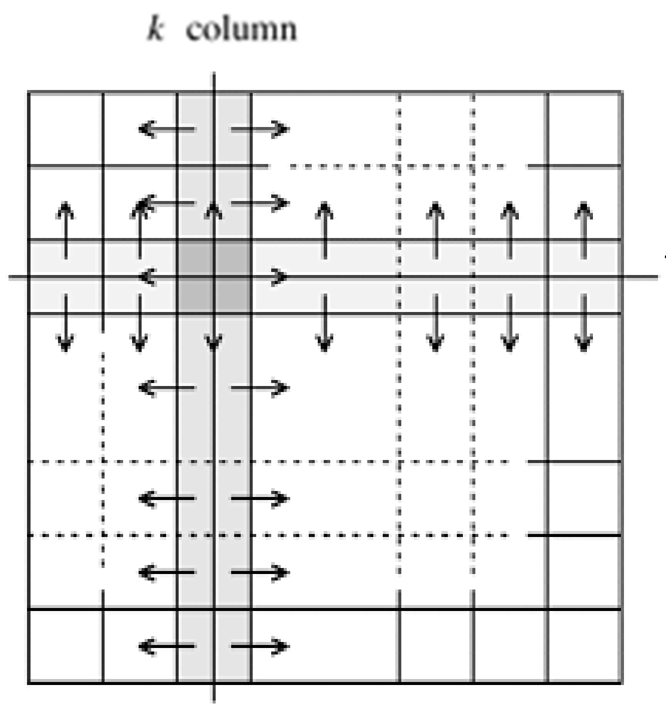
We assign block (i, j) to processor $P_{i,j}$ for $1 \leq i, j \leq \sqrt{p}$.


$\frac{n}{\sqrt{p}}$
$\frac{n}{\sqrt{p}}$
(1,1)
(1,2)
(2,1)
(i,j)
$(i-1)\frac{n}{\sqrt{p}}+1, (j-1)\frac{n}{\sqrt{p}}+1$
$i\frac{n}{\sqrt{p}}, j\frac{n}{\sqrt{p}}$

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed Memory Implementation



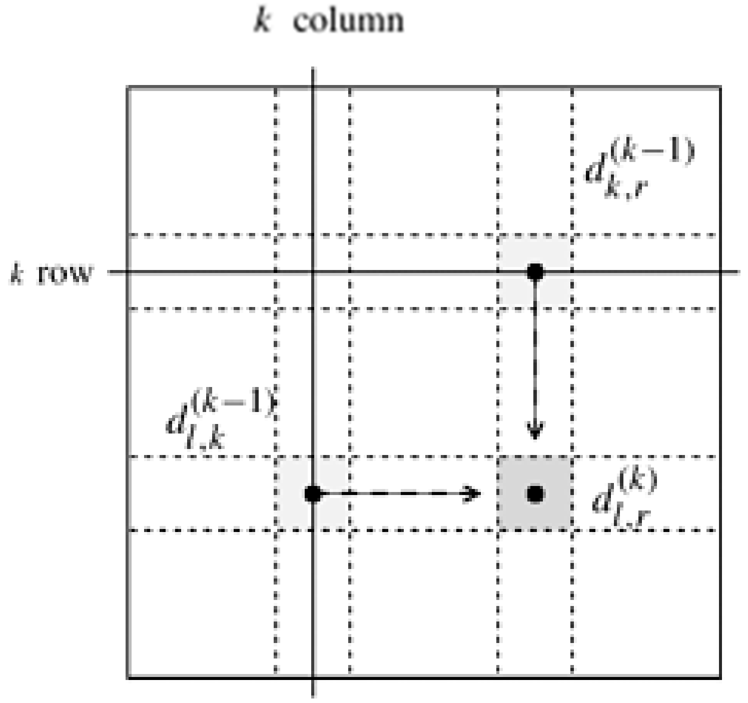


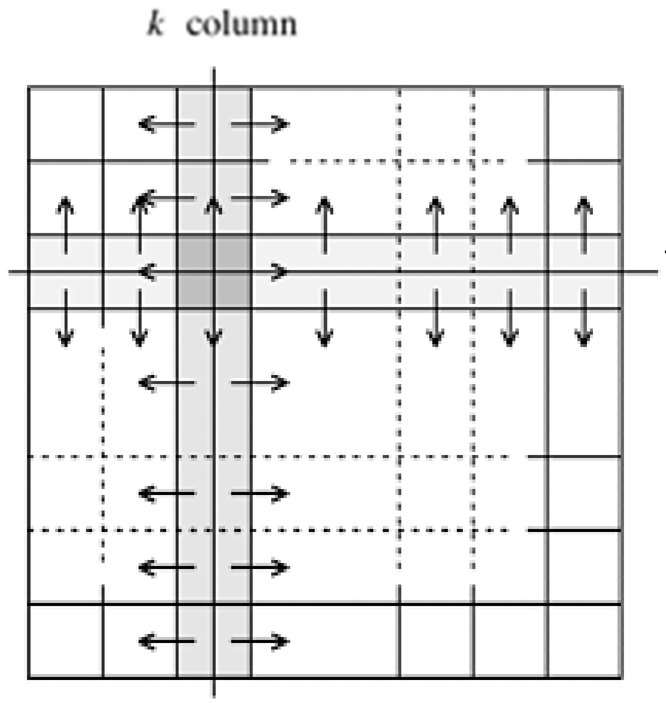
Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

During the computation of $D^{(k)}$ each processor $P_{i,j}$ requires

- a segment (of length $\frac{n}{\sqrt{p}}$) from row k of $D^{(k-1)}$ which belongs to a processor in block column j
- a segment (of length $\frac{n}{\sqrt{p}}$) from column k of $D^{(k-1)}$ which belongs to a processor in block row i

Distributed Memory Implementation





Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

After the computation of $D^{(k-1)}$ if processor $P_{i,j}$

- contains a segment from row k of $D^{(k-1)}$, it broadcasts that segment to all processors in block column j
- contains a segment from column k of $D^{(k-1)}$, it broadcasts that segment to all processors in block row i

Distributed Memory Implementation

```
FW-APSP-2D-Block ( $D^{(0)}$ )
1.  for  $k \leftarrow 1$  to  $n$  do
2.      parallel: each node $P_{i,j}$ does the following:
3.          if it contains a segment of row $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{*,j}$
4.          if it contains a segment of column $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{i,*}$
5.          waits until all nodes receive the needed segments ( global sync )
6.          computes its part of the $D^{(k)}$ matrix
```

In each iteration of the for loop (assuming t_s and t_w to be constants)

- **Line 3:** communication complexity $= \Theta\left(\frac{n}{\sqrt{p}} \log \sqrt{p}\right)$ (why?)
- **Line 4:** communication complexity $= \Theta\left(\frac{n}{\sqrt{p}} \log \sqrt{p}\right)$ (why?)
- **Line 5:** communication complexity $= \Theta(\log p)$ (sync)
- **Line 6:** computation complexity $= \Theta(n^2/p)$

Distributed Memory Implementation

```
FW-APSP-2D-Block ( $D^{(0)}$ )
1.  for $k \leftarrow 1$ to $n$ do
2.      parallel: each node $P_{i,j}$ does the following:
3.          if it contains a segment of row $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{*,j}$
4.          if it contains a segment of column $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{i,*}$
5.          waits until all nodes receive the needed segments ( global sync )
6.          computes its part of the $D^{(k)}$ matrix
```

Overall:

$$t_{comm} = \Theta\left(n \times \frac{n}{\sqrt{p}} \log p\right) = \Theta\left(\frac{n^2}{\sqrt{p}} \log p\right)$$

and $t_{comp} = \Theta\left(n \times \frac{n^2}{p}\right) = \Theta\left(\frac{n^3}{p}\right)$

Hence, $T_p = t_{comp} + t_{comm} = \Theta\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \log p\right)$

Improved Distributed Memory Implementation

```
FW-APSP-2D-Block ( $D^{(0)}$ )
1.  for $k \leftarrow 1$ to $n$ do
2.      parallel: each node $P_{i,j}$ does the following:
3.          if it contains a segment of row $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{*,j}$
4.          if it contains a segment of column $k$ of $D^{(k-1)}$,
            broadcasts that segment to nodes $P_{i,*}$
5.          waits until all nodes receive the needed segments ( global sync )
6.          computes its part of the $D^{(k)}$ matrix
```

The global synchronization in line 5 can be removed without affecting the correctness of the algorithm.

The trick is to use *pipelining*.

Pipelined 2D Block Mapping FW-APSP

```
FW-APSP-Pipelined-2D-Block ( $D^{(0)}$ )

1.  for $k \leftarrow 1$ to $n$ do
2.      parallel: each node $P_{i,j}$ does the following:
3.          if it contains a segment of row $k$ of $D^{(k-1)}$, sends that
            segment to nodes $P_{i-1,j}$ ( if $i > 1$ ) and $P_{i+1,j}$ ( if $i < \sqrt{p}$ )
4.          if it contains a segment of column $k$ of $D^{(k-1)}$, sends that
            segment to nodes $P_{i,j-1}$ ( if $j > 1$ ) and $P_{i,j+1}$ ( if $j < \sqrt{p}$ )
5.          waits only until it receives the two segments it needs
6.          computes its part of the $D^{(k)}$ matrix, and at any point if it
            receives data from any direction it stores them locally, and
            forwards them in the opposite direction
```

After the computation of row 1 & col 1, all relevant segments of $D^{(1)}$ reach $P_{\sqrt{p},\sqrt{p}}$ after $\Theta\left(\left(n/\sqrt{p}\right) \times \sqrt{p}\right) = \Theta(n)$ time units. (how?)

Successive rows & cols follow after time $\Theta(n^2/p)$ in pipelined mode.

Hence, $P_{\sqrt{p},\sqrt{p}}$ completes computation in time $\Theta(n^3/p) + \Theta(n)$.

Pipelined 2D Block Mapping FW-APSP

```
FW-APSP-Pipelined-2D-Block( $D^{(0)}$ )

1. for $k \leftarrow 1$ to $n$ do
2.     parallel: each node $P_{i,j}$ does the following:
3.         if it contains a segment of row $k$ of $D^{(k-1)}$, sends that
           segment to nodes $P_{i-1,j}$ ( if $i > 1$ ) and $P_{i+1,j}$ ( if $i < \sqrt{p}$ )
4.         if it contains a segment of column $k$ of $D^{(k-1)}$, sends that
           segment to nodes $P_{i,j-1}$ ( if $j > 1$ ) and $P_{i,j+1}$ ( if $j < \sqrt{p}$ )
5.         waits only until it receives the two segments it needs
6.         computes its part of the $D^{(k)}$ matrix, and at any point if it
           receives data from any direction it stores them locally, and
           forwards them in the opposite direction
```

When $P_{\sqrt{p},\sqrt{p}}$ completes iteration $n-1$, it sends the relevant values of row n and column n to other nodes.

These values reach $P_{1,1}$ in time $\Theta(n)$.

Hence, $T_p = t_{comp} + t_{comm} = \Theta\left(\frac{n^3}{p}\right) + \Theta(n) = \Theta\left(\frac{n^3}{p} + n\right)$