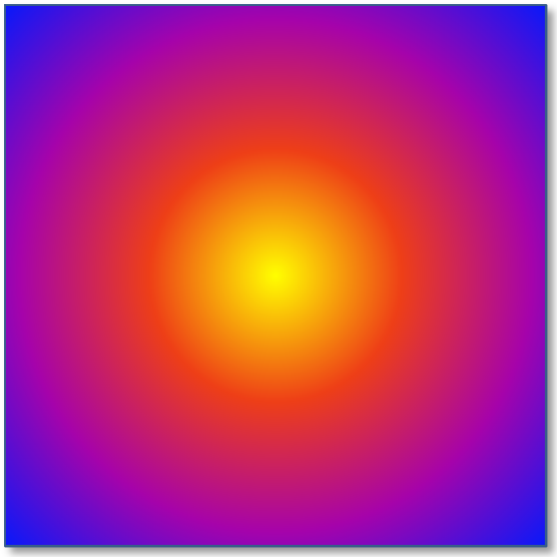
CSE 590: Special Topics Course
(Supercomputing)

Lecture 7
(Analyzing Distributed Memory Algorithms)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2016

2D Heat Diffusion

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



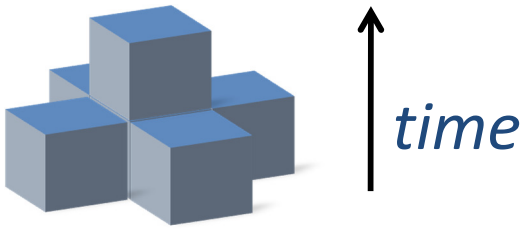
Heat Equation

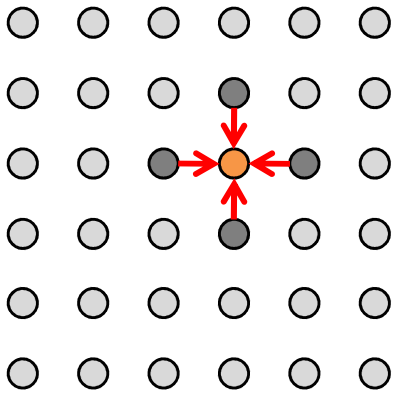
$$\frac{\partial h}{\partial t} = \alpha\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right), \quad \alpha = \text{thermal diffusivity}$$

Update Equation (on a discrete grid)

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

2D 5-point Stencil

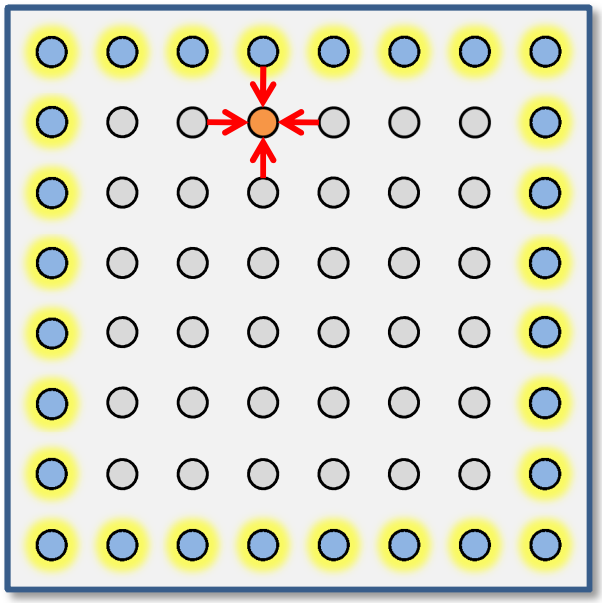




Standard Serial Implementation

Implementation Tricks

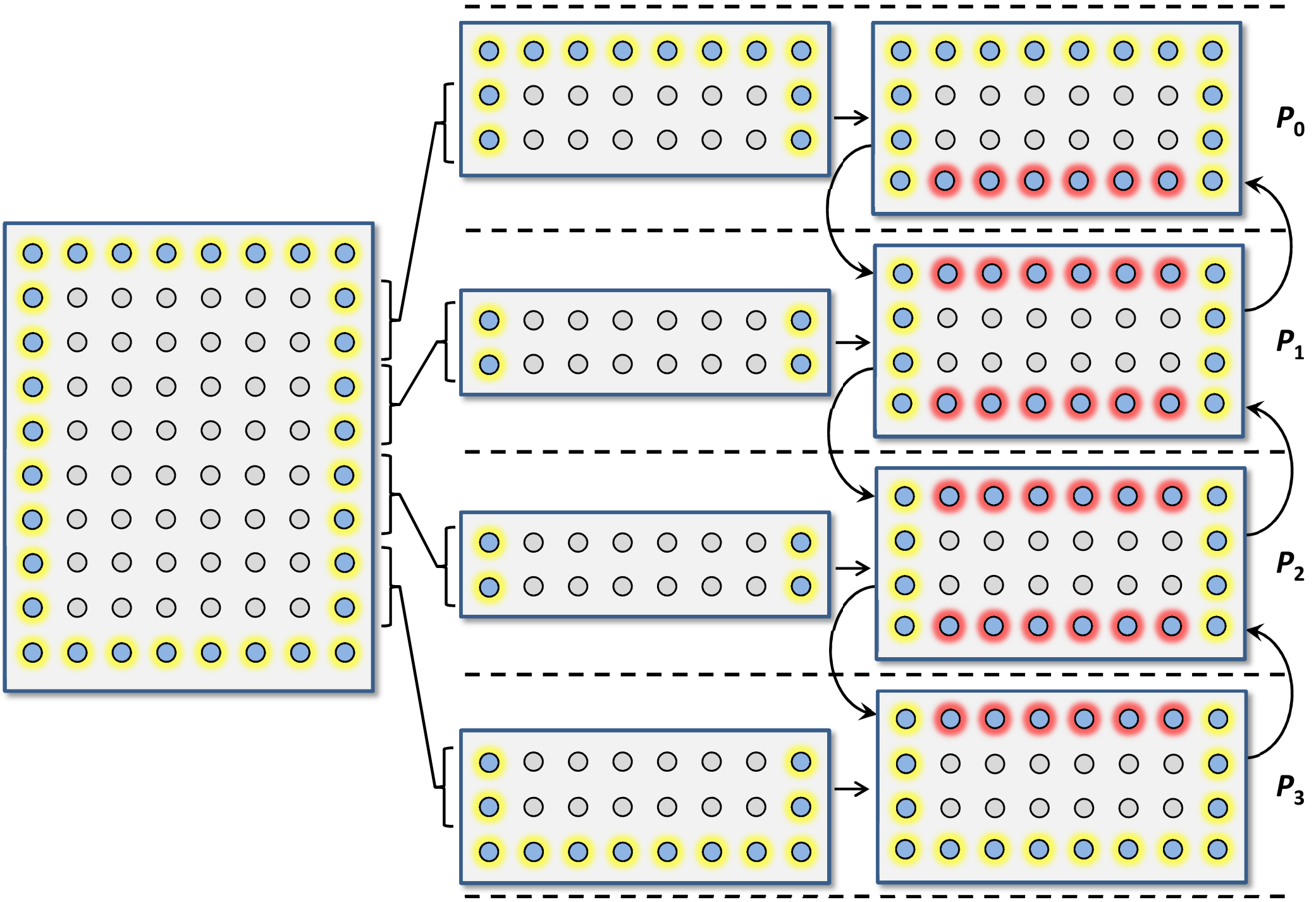
- Reuse storage for odd and even time steps
- Keep a halo of ghost cells around the array with boundary values



```
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

```
#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 ← 1 to n do
2.      for j ← 1 to n do
3.          for k ← 1 to n do
4.              z_ij ← z_ij + x_ik × y_kj
```

Naïve Matrix Multiplication

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

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

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 $k = 1, 2, \ldots, n$ 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

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

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

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 $\{P_{i,1}, P_{i,2}, \ldots, P_{i,n}\}$ of size n.

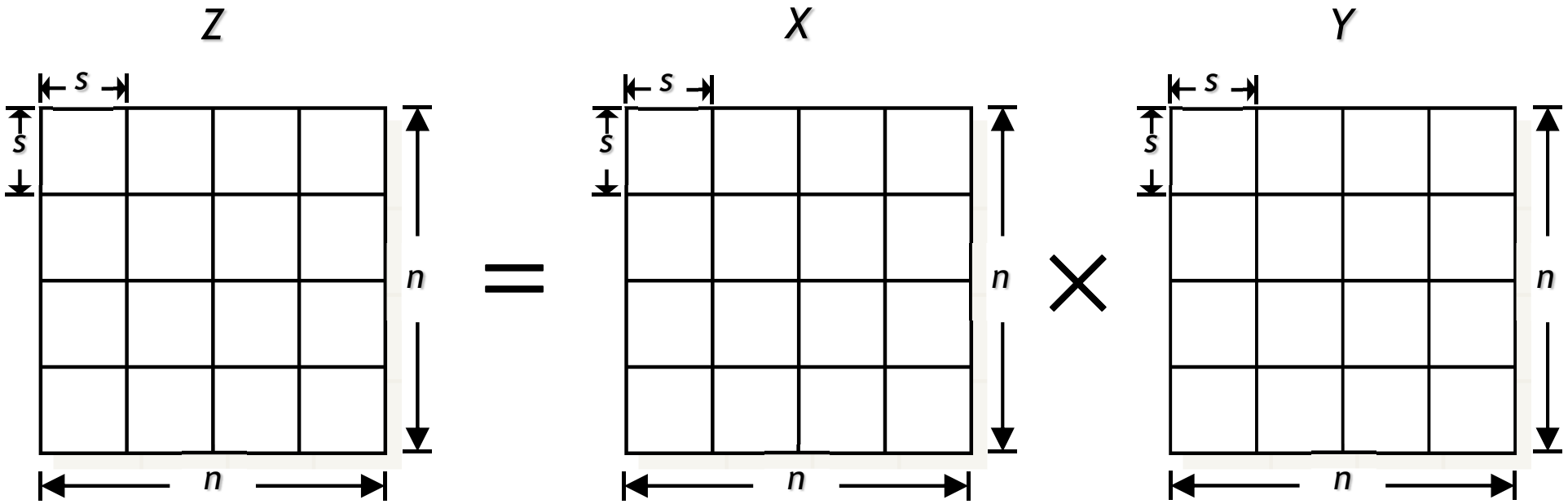
Similarly, for other rows of X, and all columns of Y.

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.

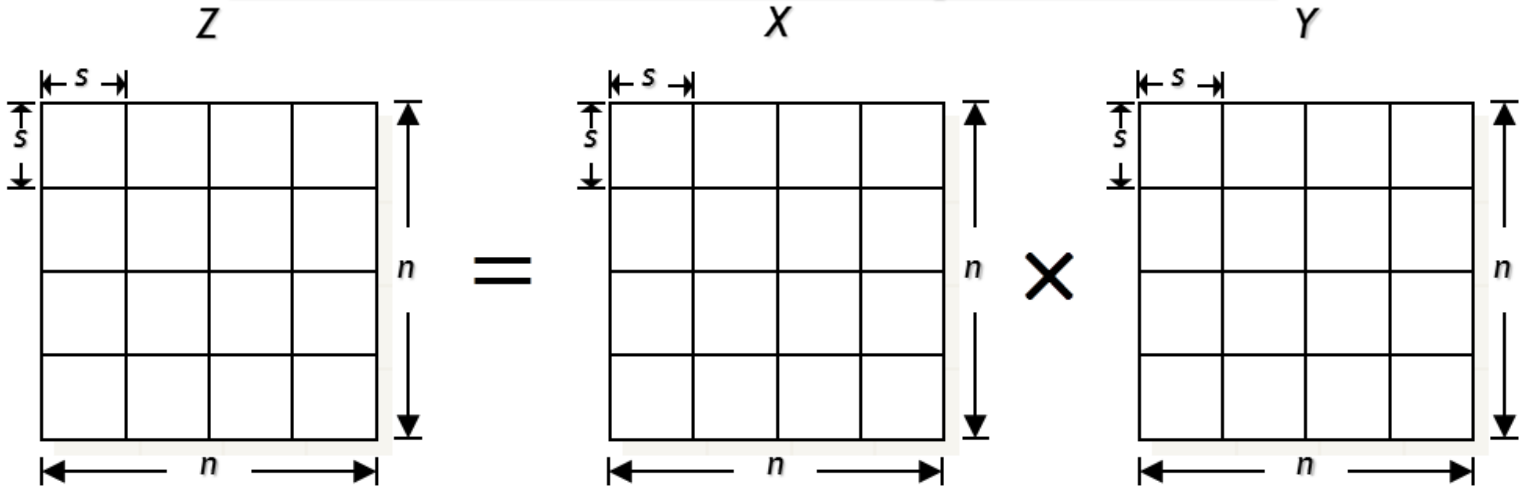
Hence, $t_{comm} = 2n(t_s + nt_w) \log n \ + n^2(t_s + t_w)$.

Block Matrix Multiplication




```
Block-MM( X, Y, Z, n )
1.  for i ← 1 to n / s do
2.      for j ← 1 to n / s do
3.          for k ← 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}.

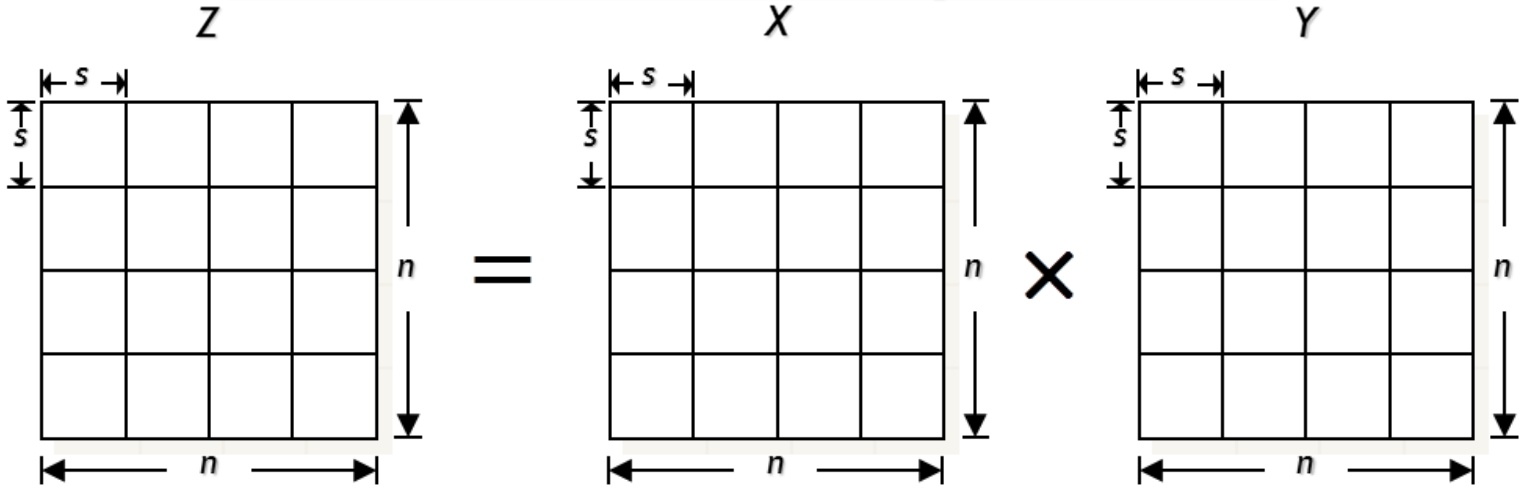
Thus $t_{comp} = \frac{n}{s}\left(2s^3 + s^2\right) = \mathrm{O}\left(ns^2\right)$,

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

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

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

Cannon's Algorithm

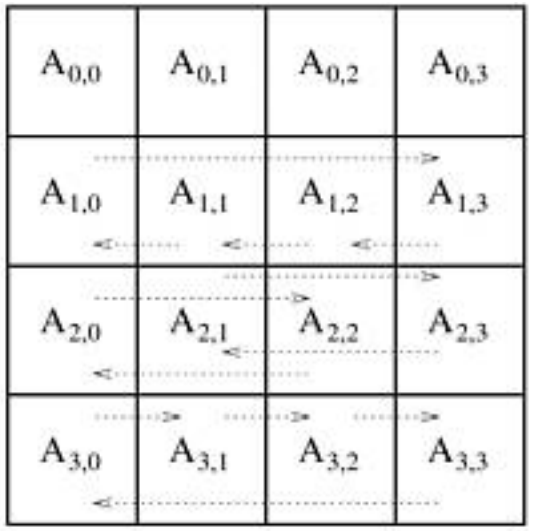
We decompose each matrix into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each.

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

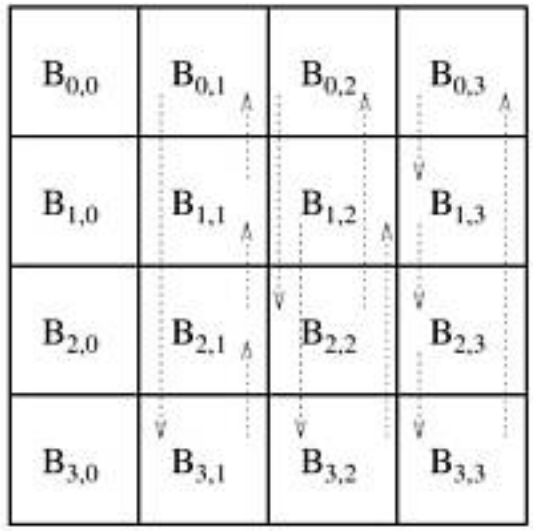
Initially, P_{ij} holds A_{ij} and B_{ij}.

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

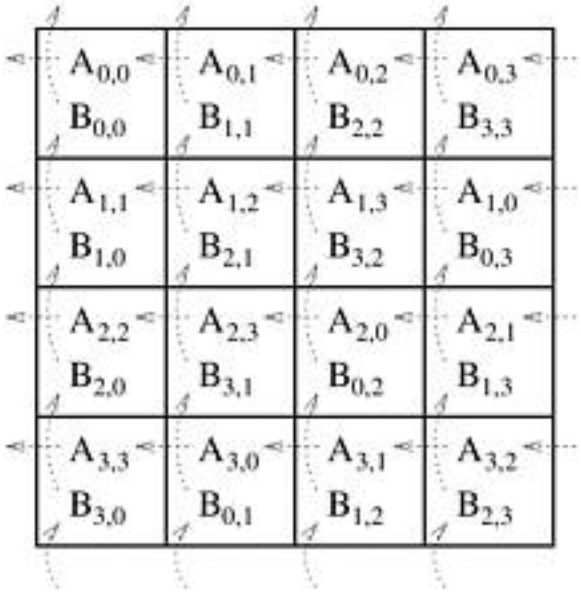
So, P_{ij} now holds $A_{i,j+i}$ and $B_{i+j,j}$.



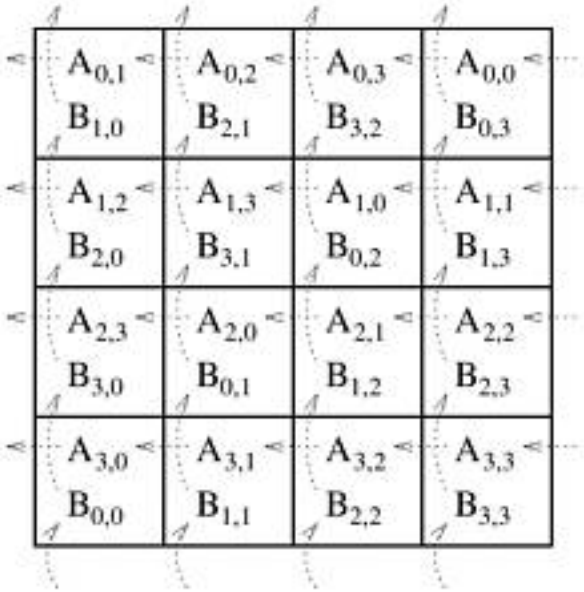
(a) Initial alignment of A



(b) Initial alignment of B



(c) A and B after initial alignment



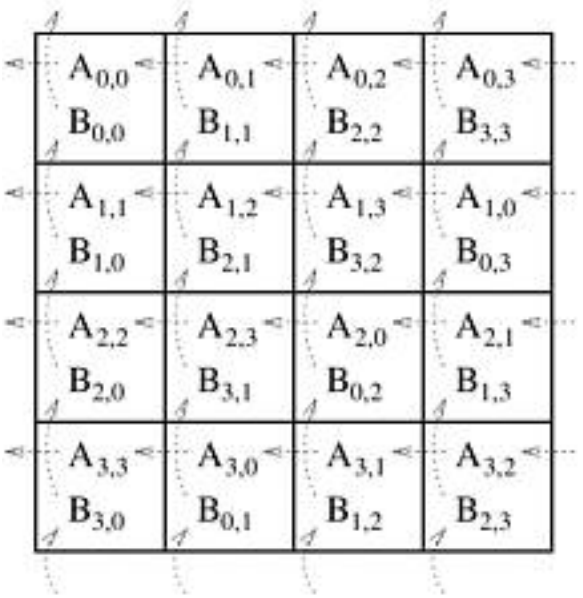
(d) Submatrix locations after first shift

Cannon's Algorithm

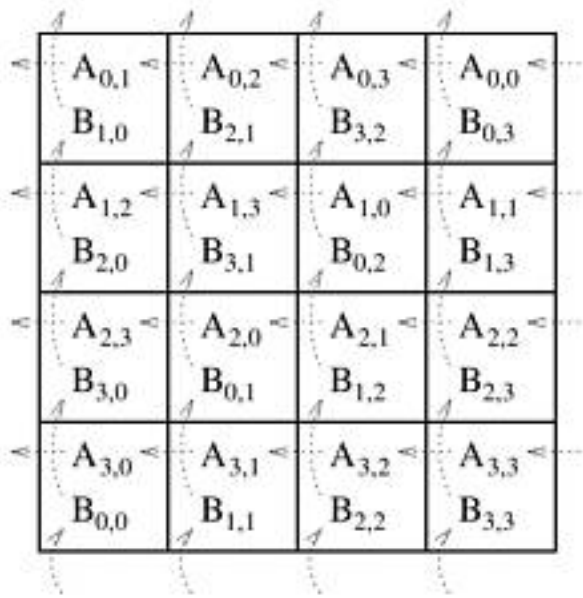
P_{ij} now holds $A_{i,j+i}$ and $B_{i+j,j}$.

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

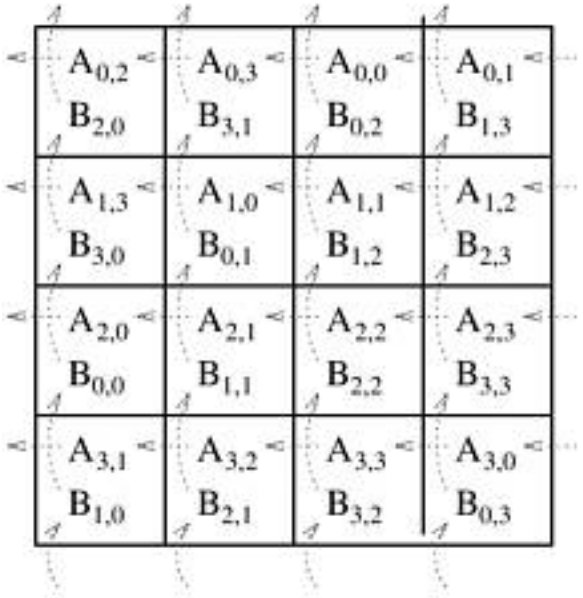
Then in each of the next $\sqrt{p}-1$ steps, each block row of A is rotated to the left by 1 position, and each block column of B is rotated upward by 1 position. Each P_{ij} adds the product of its current submatrices to $C_{i,j}$.



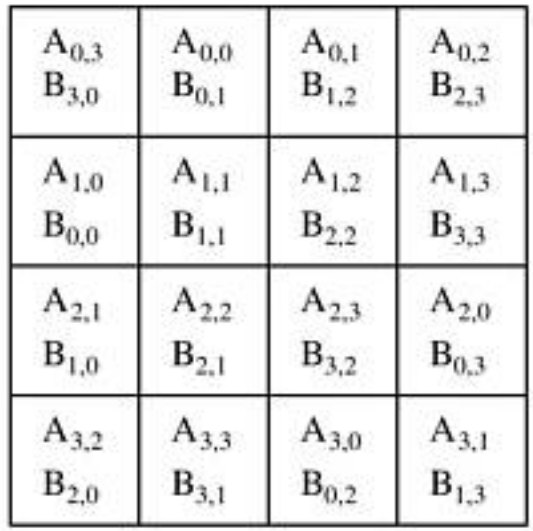
(c) A and B after initial alignment



(d) Submatrix locations after first shift



(e) Submatrix locations after second shift



(f) Submatrix locations after third shift

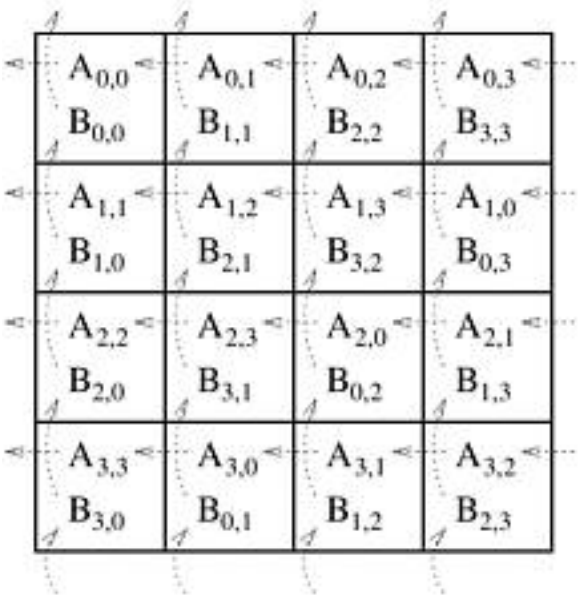
Cannon's Algorithm

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

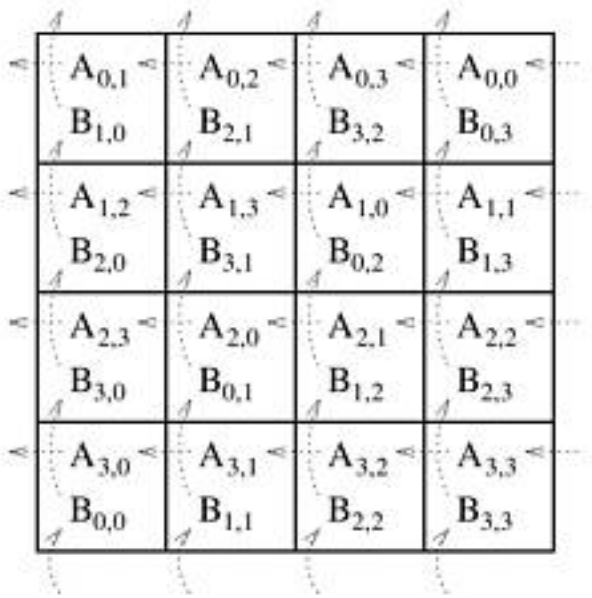
In each of the next $\sqrt{p} - 1$ steps, each processor performs one block matrix multiplication, and sends and receives one block each.

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

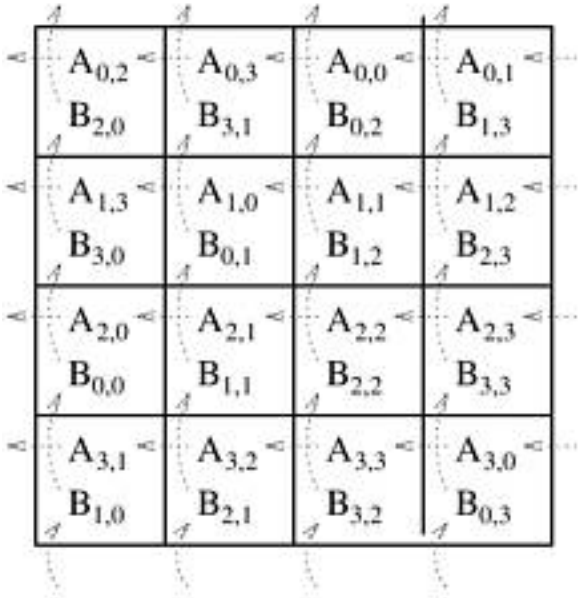
$$t_{comm} = 4\left(\sqrt{p} - 1\right) \times \left(t_s + \left(\frac{n}{\sqrt{p}}\right)^2 t_w\right).$$



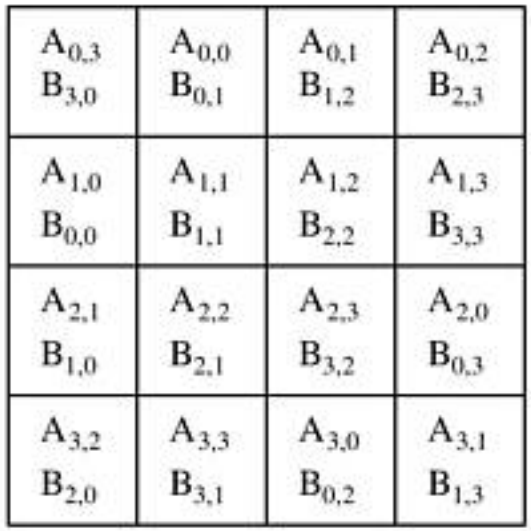
(c) A and B after initial alignment



(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:

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

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 $V = \{v_1, v_2, \ldots, v_n\}$, 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:

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

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) ← A
2. for k ← 1 to n do
3.     for i ← 1 to n do
4.         for j ← 1 to n do
5.             d_{i,j}^(k) ← min{ d_{i,j}^(k−1) , d_{i,k}^(k−1) + d_{k,j}^(k−1) }
6. return D^(n)
```

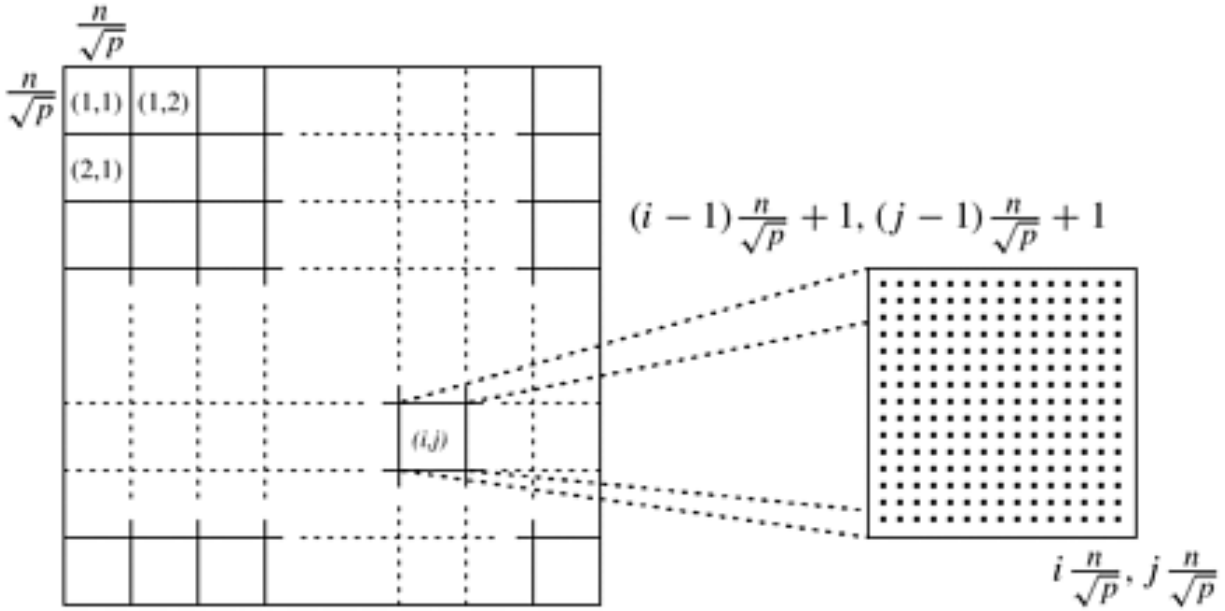
- can be solved using only $\Theta(n^2)$ extra space, e.g., using only two $n \times n$ matrices for storing the values of D
- can be solved in-place in A
- serial running time is $\Theta(n^3)$

Distributed Memory Implementation

Let p be the number of processing nodes.

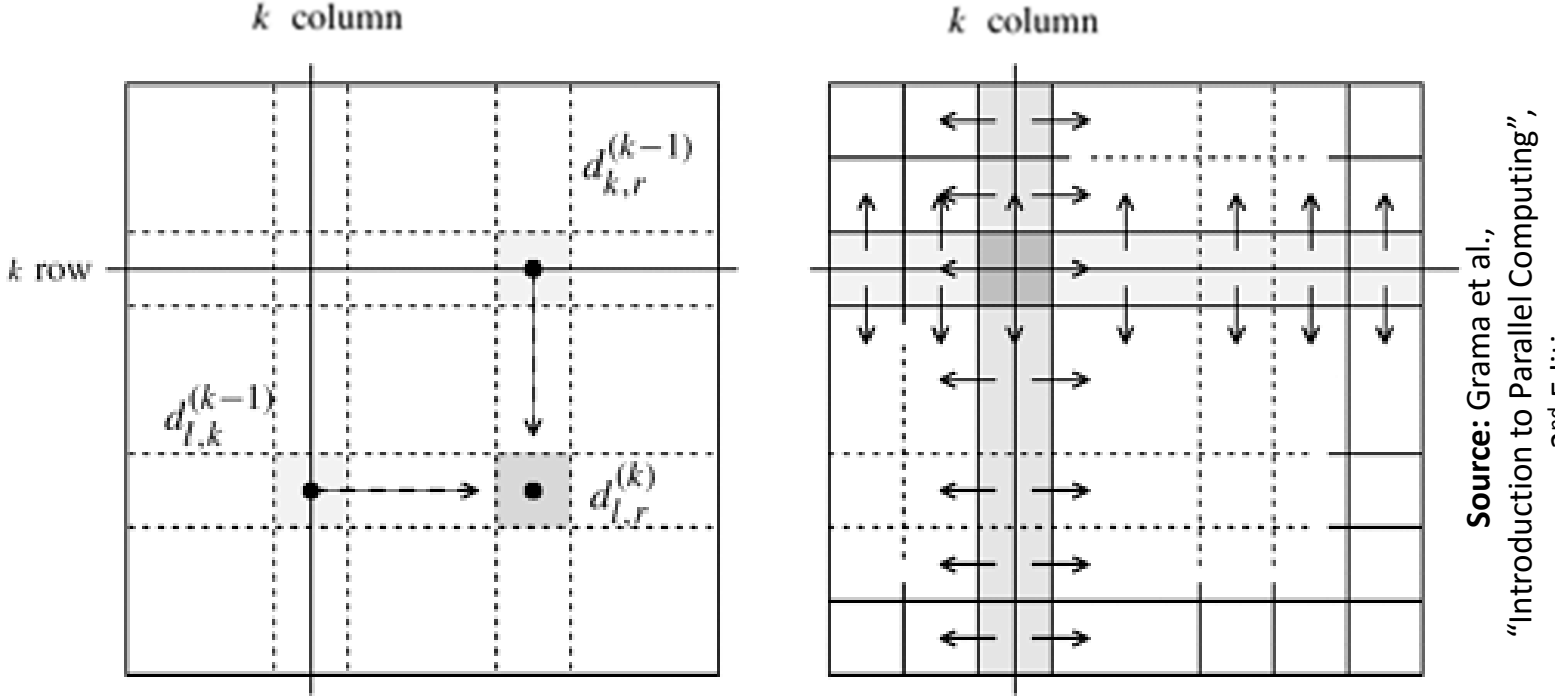
We divide $D^{(k)}$ into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each.

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



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

Distributed Memory Implementation

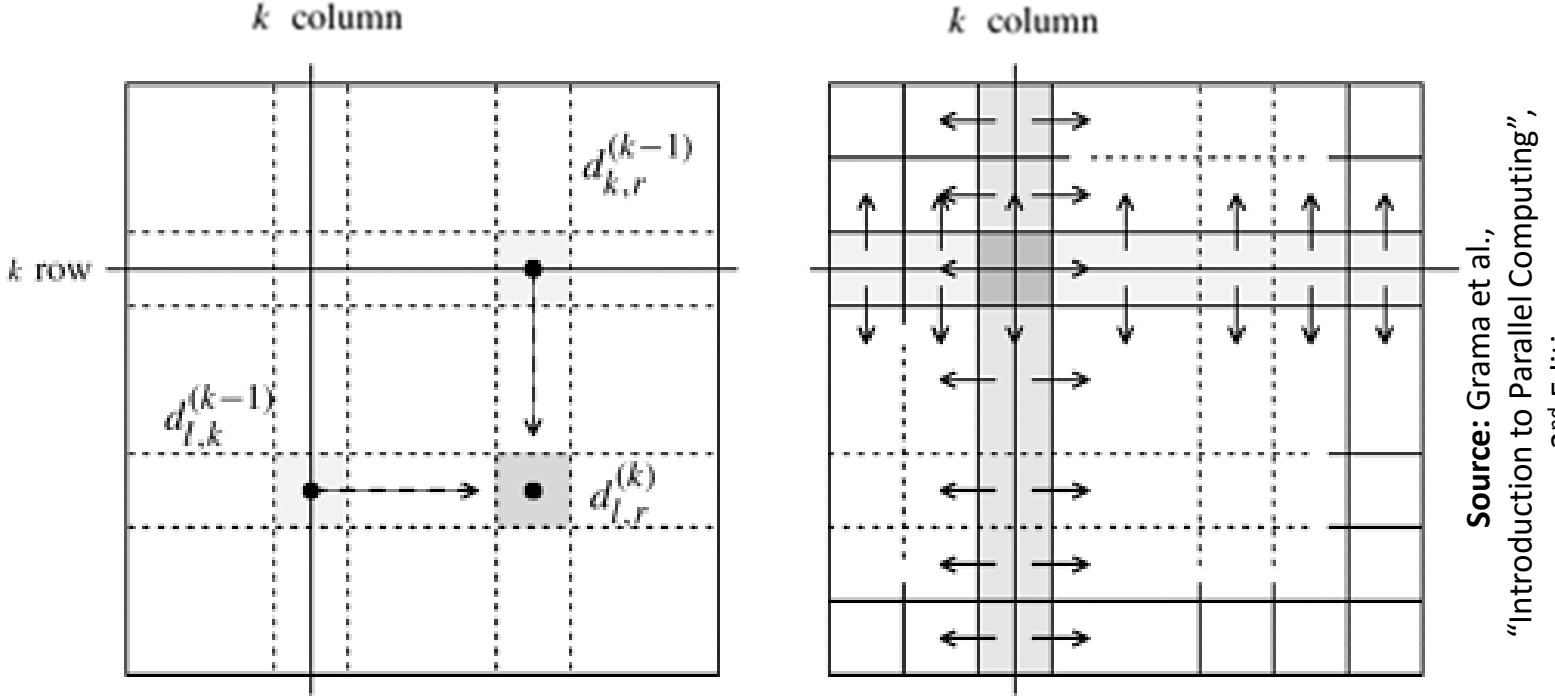


Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

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 $\Theta\left(n^2/p\right)$ in pipelined mode.

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

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

Parallel QuickSort: A Shared-Memory Version

Input: An array $A[\, q : r \,]$ of distinct elements.

Output: Elements of $A[\, q : r \,]$ sorted in increasing order of value.

```
Par-Randomized-Looping-QuickSort ( A[ q : r ] )
 1.  m ← r − q + 1
 2.  if m > 1 then
 3.      k ← 0
 4.      while max{ r − k, k − q } > 3m / 4 do
 5.          select a random element x from A[ q : r ]
 6.          k ← Par-Partition ( A[ q : r ], x )
 7.      spawn Par-Randomized-Looping-QuickSort ( A[ q : k − 1 ] )
 8.      Par-Randomized-Looping-QuickSort ( A[ k + 1 : r ] )
 9.      sync
```

Parallel QuickSort: Distributed-Memory Version

Input: An array $A[\, q : r \,]$ of distinct elements distributed among processing nodes P_s, P_{s+1}, …, P_t such that each nodes contains between $\alpha/2$ and 2α elements, where $\alpha = n / p$ = orig #elems / orig #nodes.

Output: Elements of $A[\, q : r \,]$ sorted in increasing order of value distributed among the nodes in the following order: P_s, P_{s+1}, …, P_t.

Distributed-Randomized-Looping-QuickSort ($A[\, q : r \,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\, q : r \,]$ locally on P_s using serial quicksort
2. *else*
3. $m \leftarrow r - q + 1$, $k \leftarrow 0$
4. *while* $\max\{ r - k, k - q \} > 3m / 4$ *do*
5. select a random element x from $A[\, q : r \,]$
6. $k \leftarrow$ *Distributed-Rank* ($A[\, q : r \,]$, x , s, t)
7. Find an i, and redistribute $A[\, q : r \,]$ among the nodes as evenly as possible such that
   (a) all elements $\le x$ are stored among nodes P_s to P_i ,
   (b) all elements $> x$ are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer $\alpha/2$ or more than 2α elements
7. *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\, q : k \,]$, α, s, i)
   Distributed-Randomized-Looping-QuickSort ($A[\, k + 1 : r \,]$, α, $i + 1$, t)

Distributed QuickSort: Example

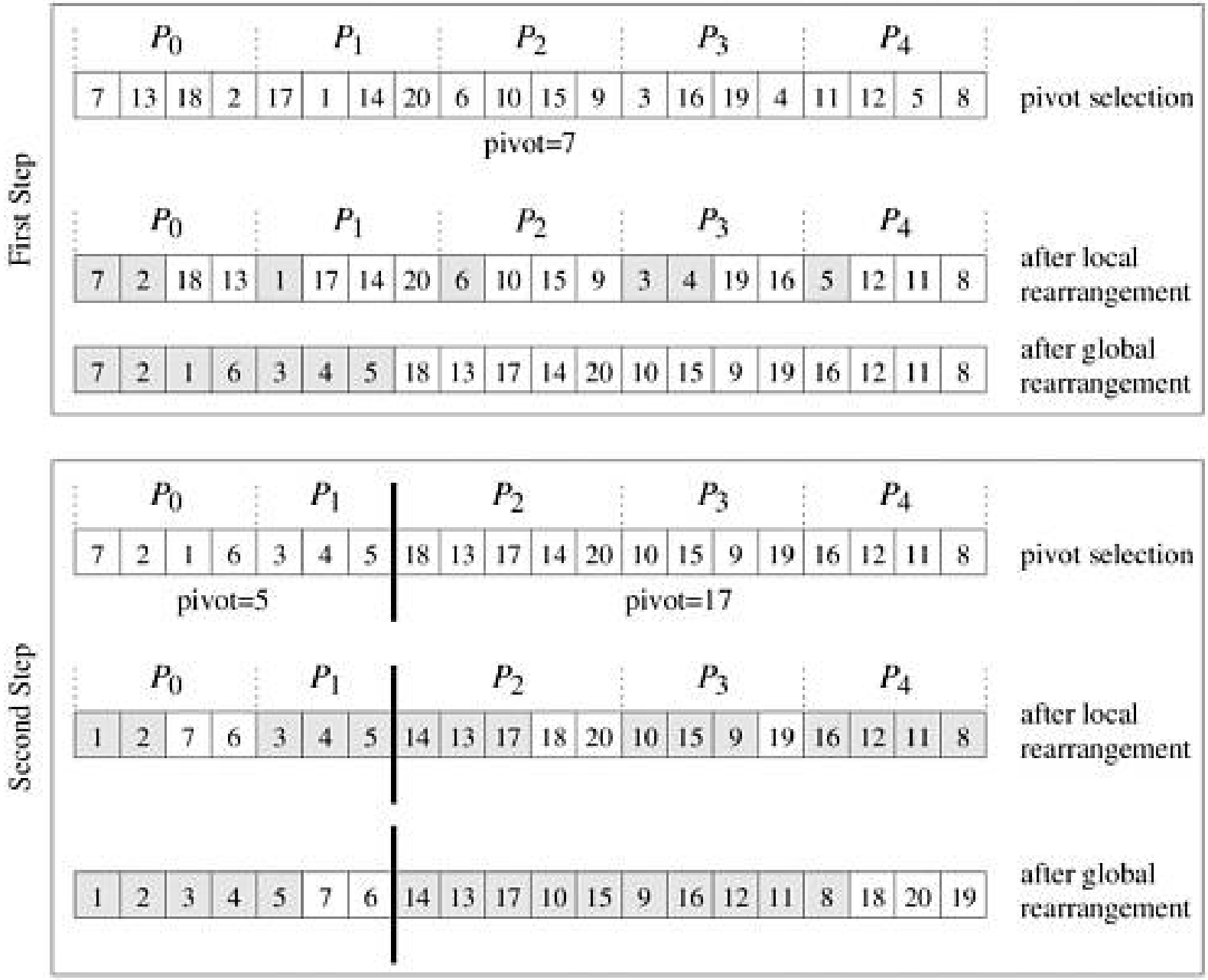


Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed QuickSort: Example

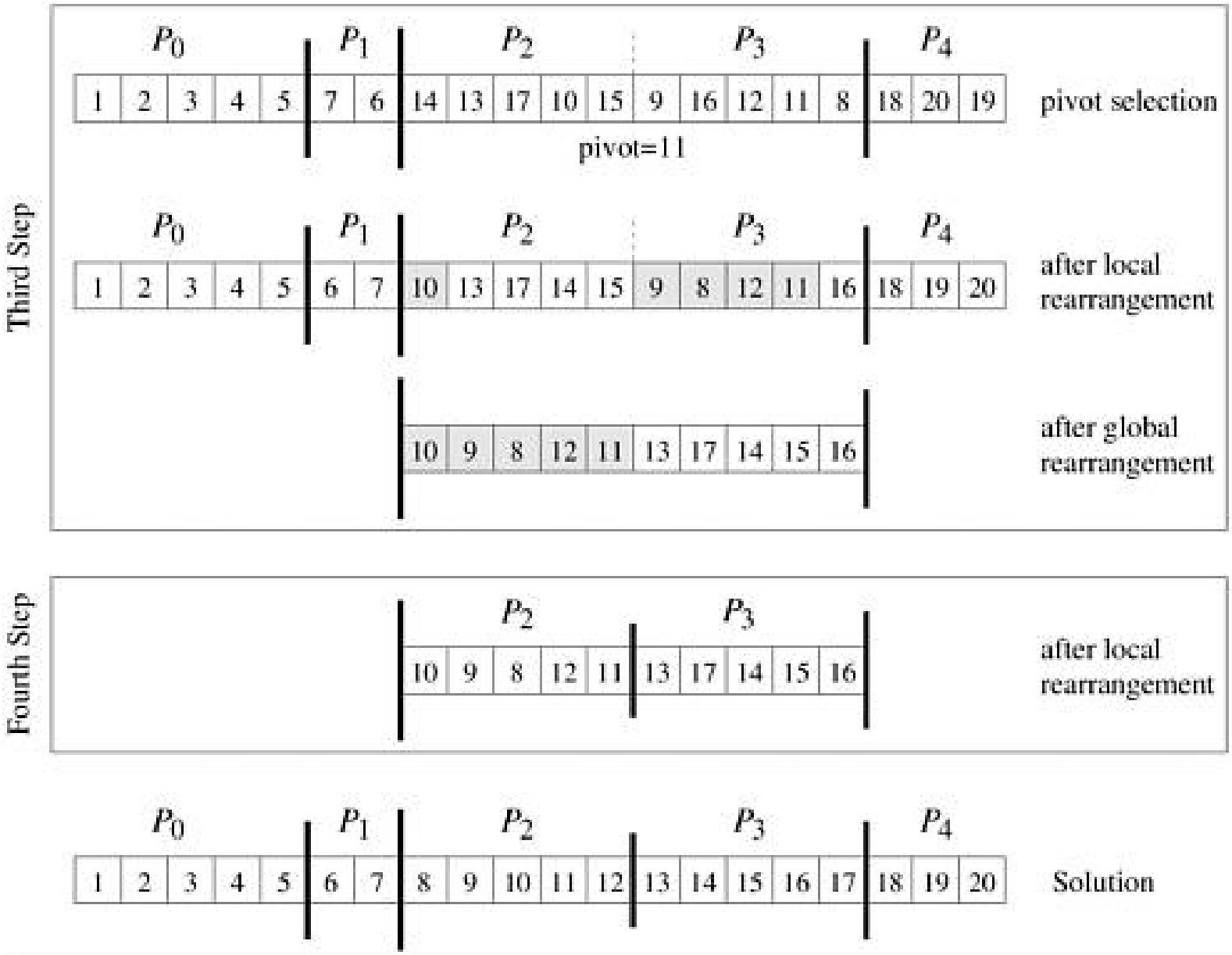


Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed QuickSort: Distributed Rank & Partition

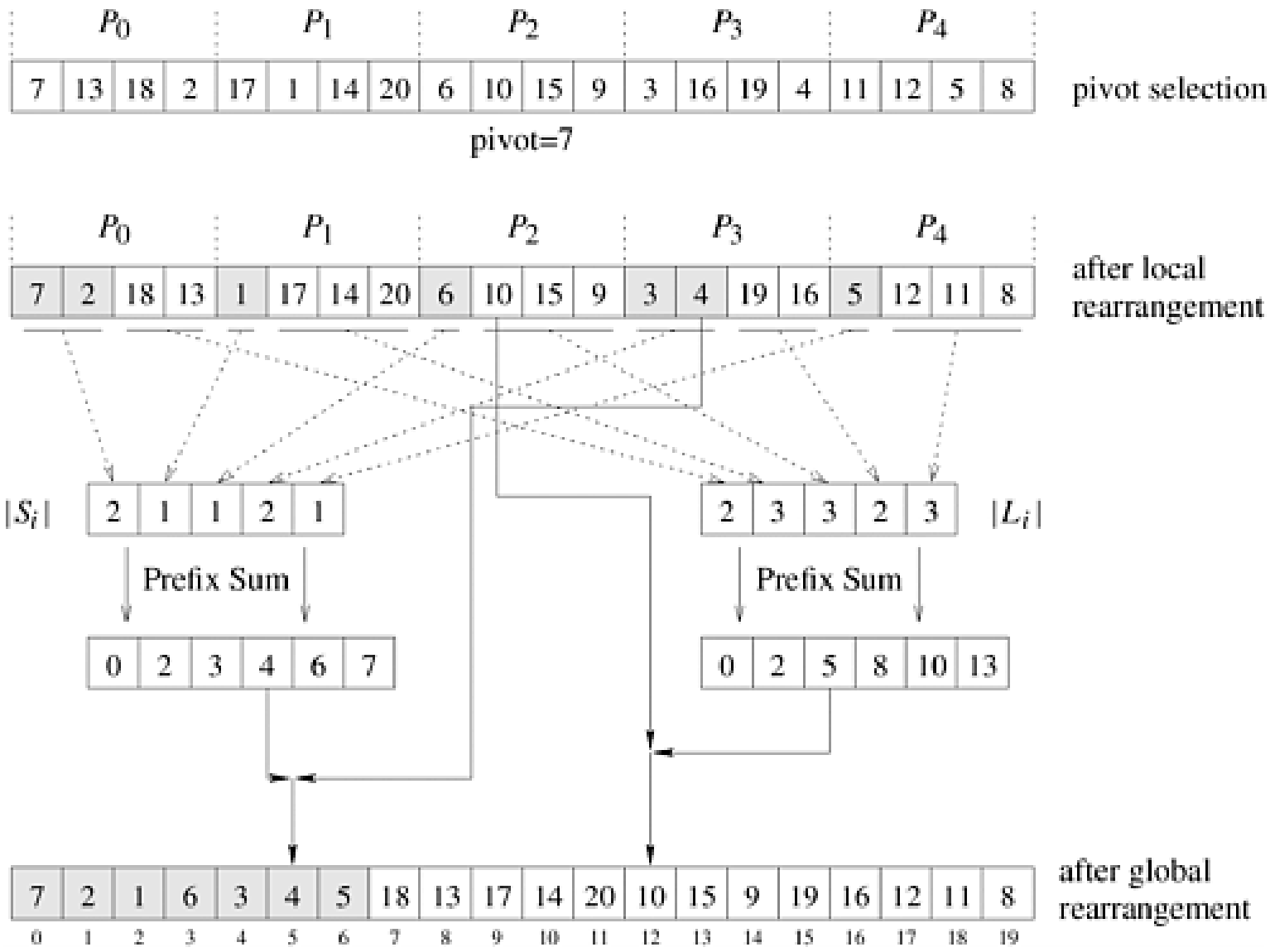


Image Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\, q : r \,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\, q : r \,]$ locally on P_s using serial quicksort
2. *else*
3. $\quad m \leftarrow r - q + 1$, $\quad k \leftarrow 0$
4. $\quad$ *while* $\max\{\, r - k,\ k - q \,\} > 3m / 4$ *do*
5. $\quad\quad$ select a random element x from $A[\, q : r \,]$
6. $\quad\quad k \leftarrow$ *Distributed-Rank* ($A[\, q : r \,]$, x , s, t)
7. $\quad$ Find an i, and redistribute $A[\, q : r \,]$ among the nodes as evenly as possible such that
   (a) all elements $\leq x$ are stored among nodes P_s to P_i ,
   (b) all elements $> x$ are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer than $\alpha/2$ or more than 2α elements
8. $\quad$ *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\, q : k \,]$, α, s, i)
   $\quad\quad\quad$ *Distributed-Randomized-Looping-QuickSort* ($A[\, k + 1 : r \,]$, α, $i + 1$, t)

Lines 5-6 (assuming t_s and t_w to be constants)

- communication complexity $= O(p + \log p)$ (why?)
- computation complexity $= O\left(\frac{n}{p}\right)$ (why?)
- overall $= O\left(\frac{n}{p} + p + \log p\right)$

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\, q : r \,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\, q : r \,]$ locally on P_s using serial quicksort
2. *else*
3. $m \leftarrow r - q + 1$, $k \leftarrow 0$
4. *while* $\max\{ r - k, k - q \} > 3m / 4$ *do*
5. select a random element x from $A[\, q : r \,]$
6. $k \leftarrow$ *Distributed-Rank* ($A[\, q : r \,]$, x , s, t)
7. Find an i, and redistribute $A[\, q : r \,]$ among the nodes as evenly as possible such that
   (*a*) all elements $\leq x$ are stored among nodes P_s to P_i ,
   (*b*) all elements $> x$ are stored among nodes P_{i+1} to P_t , and
   (*c*) no node stores fewer than $\alpha/2$ or more than 2α elements
8. *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\, q : k \,]$, α, s, i)
   Distributed-Randomized-Looping-QuickSort ($A[\, k + 1 : r \,]$, α, $i + 1$, t)

Line 7 (assuming t_s and t_w to be constants)

- communication complexity $= O\left(p + \log p + \frac{n}{p}\right)$ (why?)
- computation complexity $= O(1)$ (why?)
- overall $= O\left(\frac{n}{p} + p + \log p\right)$

Distributed QuickSort

Distributed-Randomized-Looping-QuickSort ($A[\, q : r \,]$, α, s, t)

1. *if* $s = t$ *then* sort $A[\, q : r \,]$ locally on P_s using serial quicksort
2. *else*
3. $\quad m \leftarrow r - q + 1$, $k \leftarrow 0$
4. $\quad$ *while* $\max\{ r - k, k - q \} > 3m / 4$ *do*
5. $\quad\quad$ select a random element x from $A[\, q : r \,]$
6. $\quad\quad k \leftarrow$ *Distributed-Rank* ($A[\, q : r \,]$, x , s, t)
7. $\quad$ Find an i, and redistribute $A[\, q : r \,]$ among the nodes as evenly as possible such that
   (a) all elements $\le x$ are stored among nodes P_s to P_i ,
   (b) all elements $> x$ are stored among nodes P_{i+1} to P_t , and
   (c) no node stores fewer $\alpha/2$ or more than 2α elements
8. $\quad$ *parallel:* *Distributed-Randomized-Looping-QuickSort* ($A[\, q : k \,]$, α, s, i)
   $\quad\quad$ *Distributed-Randomized-Looping-QuickSort* ($A[\, k + 1 : r \,]$, α, $i + 1$, t)

Depth of the shared-memory version is $O(\log n)$ w.h.p.

Same bound applies to the distributed-memory version.

Hence, $T_p = O\left(\left(\frac{n}{p} + p + \log p\right)\log n\right) = O\left(\frac{n \log n}{p} + p \log n\right)$ (w.h.p.)

Distributed Sample Sort

Task: Sort n distinct keys using p processing nodes.

Steps:

1. **Initial Distribution:** The master node scatters the n keys among p processing nodes as evenly as possible.
2. **Pivot Selection:** Each node sorts its local keys, and selects $q-1$ evenly spaced keys from its sorted sequence. The master node gathers these *local pivots* from all nodes, locally sorts those $p(q-1)$ keys, selects $p-1$ evenly spaced global pivots from them, and broadcasts them to all nodes.
3. **Local Bucketing:** Each node inserts the global pivots into its local sorted sequence using binary search, and thus divides the keys among p buckets.
4. **Distribute Local Buckets:** For $1 \leq i \leq p$, each node sends bucket i to node i.
5. **Local Sort:** Each node locally sorts the elements it received in step 4.
6. **Final Collection:** The master node collects all sorted keys from all nodes, and for $1 \leq i < p$, places all keys from node i ahead of all keys from node $i+1$.

Bound on Bucket Sizes

Theorem: If each node selects $q - 1$ evenly spaced keys in step 2, then no node will sort more than are $\frac{n}{p} + \frac{n}{q}$ keys (in the worst case) in step 5.

Proof: Homework.

Analyzing Distributed Sample Sort

Steps: (assuming $q = \Theta(p)$, and t_s and t_w constants)

1. **Initial Distribution:** $\mathrm{O}\left(\log p + \frac{n}{p}(p-1)\right) = \mathrm{O}(n + \log p)$ **[comm: scatter]**

2. **Pivot Selection:** $\mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + pq\log(pq)\right) = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p\right)$ **[comp: sort]**

   $\mathrm{O}(\log p + (q-1)(p-1) + (p-1)\log p) = \mathrm{O}(p^2)$ **[comm: gather, broadcast]**

3. **Local Bucketing:** $\mathrm{O}\left((p-1)\log\frac{n}{p}\right) = \mathrm{O}(p\log n)$ **[comp: binary search]**

4. **Distribute Local Buckets:** $\mathrm{O}\left(\frac{n}{p} + \left(\frac{n}{p} + \frac{n}{q}\right)\right) = \mathrm{O}\left(\frac{n}{p}\right)$ **[comm: send, receive]**

5. **Local Sort:** $\mathrm{O}\left(\left(\frac{n}{p} + \frac{n}{q}\right)\log\left(\frac{n}{p} + \frac{n}{q}\right)\right) = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p}\right)$ **[comp: sort]**

6. **Final Collection:** $\mathrm{O}\left((p-1)\left(\frac{n}{p} + \frac{n}{q}\right)\right) = \mathrm{O}(n)$ **[comm: receive]**

Analyzing Distributed Sample Sort

Overall:

$$t_{comp} = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

$$t_{comm} = \mathrm{O}\left(n + p^2\right)$$

$$T_p = t_{comp} + t_{comm} = \mathrm{O}\left(n + \frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

Overall (excluding steps 1 and 6):

$$t_{comp} = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$

$$t_{comm} = \mathrm{O}\left(\frac{n}{p} + p^2\right)$$

$$T_p = t_{comp} + t_{comm} = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + p^2\log p + p\log n\right)$$