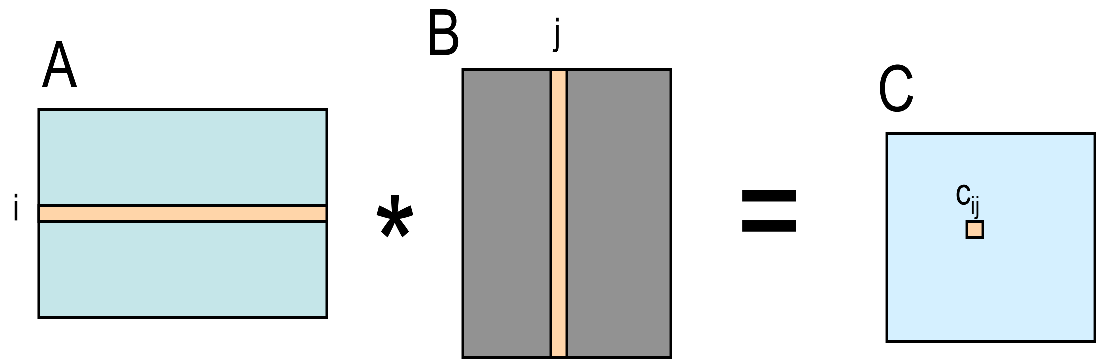
Matrix multiplication

- A is a $n*l$ matrix with elements a_{ij}
  B is a $l*m$ matrix with elements b_{ij}
  C is a $n*m$ matrix with elements c_{ij}
  - the product $C = A \times B$ is computed as the dot product of row i in A with column j in B

$$c_{i,j} = \sum_{k=0}^{l-1} a_{ik} b_{kj}$$



Sequential matrix multiplication

- The sequential code to multiply two matrices *A (of size n*l)* and *B* (of size *l*m*) is

```
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    C[i][j] = 0.0;
    for (k=0; k<l; k++) {
      C[i][j] += A[i][k]*B[k][j];
    }
  }
}
```

- Time complexity is $O(n^3)$
  - for each element in *C* we do *l* multiplications and additions

Memory access in matrix multiplication

- A straight forward implementation of matrix multiplication is very slow because it accesses memory inefficiently
  - accesses to matrices *A* and *C* are efficient, but accesses to *B* are done with a stride equal to the row length *m*

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++) {
    C[i][j] = 0.0;
    for (k=0; k<l; k++
      C[i][j] += A[i][k]*B[k][j];
  }
```

- In C and C++, elements in a row are stored in consecutive memory locations
  - elements in a column are located far from each other in memory
  - we get a cache miss for every access of the *B* matrix
- Leads to inefficient cache memory utilization
  - the processor spends most of its time waiting for memory accesses

Memory allocation in C

- Matrices should be allocated as a consecutive block of memory
  - the whole matrix can then be sent in a single message without first copying it to a contiguous message buffer
  - accesses are more efficient, because the elements are located in consecutive memory positions
  - less cache misses, better use of automatic prefetching
- Modern processors use prefetching
  - when a regular memory access pattern is detected, the next cache line is automatically brought in to main memory before the data is actually needed
  - cache lines are typically of length 64 bytes
- Should arrange memory accesses to take advantage of the cache memory and the prefetch mechanism

Static memory allocation

- Static memory allocation
  - the elements in the matrix are stored in memory as a contiguous block in row-major order

```
const int N = 1000;
double X[N][N];
... use the matrix X
```

- We can also read in the value of *N*
  - the matrix must be declared in a scope where *N* is initialized
- Static memory is allocated on the stack
  - there is an upper limit on how large blocks of memory can be allocated on the stack

```
int N;
printf("Give N? ");
scanf("%d", &N);

double X[N][N];
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
         X[i][j] = i+j;
    }
}
```

Dynamic memory allocation

- In C, memory is dynamically allocated with the *malloc* (or *calloc*) system function

```
void * malloc(size_t SIZE)
void * calloc(size_t COUNT, size_t ELTSIZE)
```

  - calloc initializes each element to zero
- In C++ memory is dynamically allocated with *new*
  - Example: `int *v = new int[size];`
- Dynamic memory is allocated on the heap
  - there are no limits on the size of memory blocks, except the amount of memory available in the system
- Dynamic memory allocation can be a slow procedure
  - should not be called inside the innermost loops

Allocating a matrix

- Allocation as a one-dimensional array of size *rows*cols*

```
double *M;
M = (double *) malloc(rows*cols*sizeof(double));
/* Set matrix M to zero */
for (i=0; i<rows; i++)
     for (j=0; j<cols; j++)
          M[i*cols+j] = 0.0;
```

- Have to calculate the address expressions explicitly in the code
  – can also use a macro definition to do the address calculation

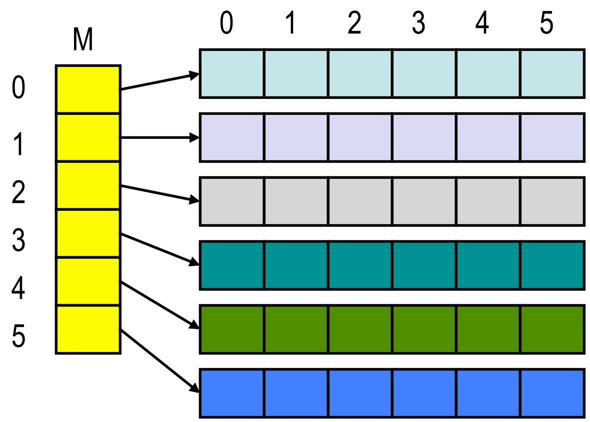
```
#define MAT(i,j) (M[i*cols+j]);
. . .
for (i=0; i<rows; i++)
     for (j=0; j<cols; j++)
          MAT(i,j) = 0.0;
```

Allocating a 2D matrix

- Allocation as a two-dimensional array, one row at a time

```
double **M;
M = (double **) malloc(rows*sizeof(double *));
for (i=0; i<rows; i++)
     M[i] = (double *) malloc(cols*sizeof(double));
. . .
for (i=0; i<rows; i++)
     for (j=0; j<cols; j++)
          M[i][j] = 0.0;
```

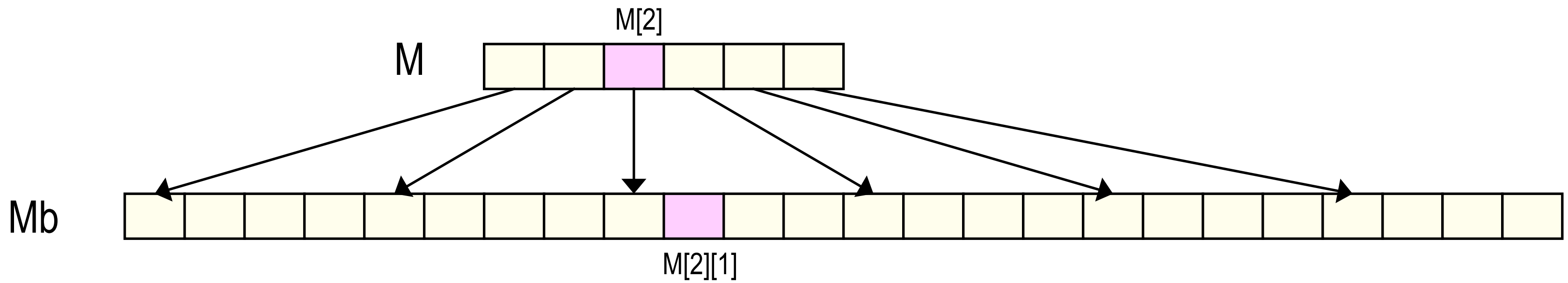
- Gives a non-contiguous allocation
  - each row is separately allocated and can be placed anywhere in memory
  - can not send the matrix to another process without copying it to a contiguous message buffer



Allocating a contiguous 2D matrix

- Alternative method using a contiguous block of memory

```
double **M;     /* Row pointers */
double *Mb;     /* Where data will be stored */
M = (double **) malloc(rows*sizeof(double *));
Mb = (double *) malloc(rows*cols*sizeof(double));
/* Initialize pointers to rows in the matrix */
for (i=0; i<rows; i++)
     M[i] = Mb + i*rows
. . .
for (i=0; i<rows; i++)
     for (j=0; j<cols; j++)
          M[i][j] = 0.0;
```



Parallel matrix multiplication

- We present a parallel algorithm for matrix multiplication based on block decomposition
  - each process computes a rectangular block (a sub-matrix) of the result
  - suitable for an implementation on distributed memory
- All the elements in the result matrix $c_{i,j}$ can be computed in parallel
  - to compute $c_{i,j}$ we only need to read row i of A and column j of B

- There are also methods based on row decomposition
  - each process computes a number of rows of the result matrix
  - suitable for a shared memory implementation, since all processes need access to the whole B matrix

Parallel matrix multiplication with Fox's algorithm

- For simplicity we assume that
  - the matrices are square and of order n (i.e., they are n x n matrices)
  - the number of processes is n^2
- The processes are arranged in a 2-dimensional grid
- Each process is assigned one element of the matrix
  - process (i,j) (= process with rank $i*n+j$) has elements a_{ij}, b_{ij} and c_{ij}
  - we will later in the agglomeration stage modify the algorithm so that each process operates on a square block of elements (a submatrix)
- We assume that the data is already distributed among the processes

Fox's algorithm

- The algorithm proceeds in *n* stages
  - one stage for each term in the dot product of row *i* and column *j*

    $c_{ij} = a_{i0}*b_{0j} + a_{i1}*b_{1j} + \ldots + a_{i,n-1}*b_{n-1,j}$

- Algorithm for process *(i,j)*
  - **Stage 0:** $c_{ij} = a_{ii} * b_{ij}$
    - multiply the diagonal entry of *A* in the own row by the own element of *B*
  - **Stage 1:** $c_{ij} \mathrel{+}= a_{i,i+1} * b_{i+1,j}$
    - multiply the element one step to the right of the diagonal (in the own row) in *A* by the element one step below the own element of *B*

  . . .

  - **Stage *k*:** $c_{ij} \mathrel{+}= a_{i,i+k} * b_{i+k,j}$
    - multiply the element *k* columns to the right of the diagonal of *A* by the element *k* rows below the own element of *B*

Fox's algorithm (cont.)

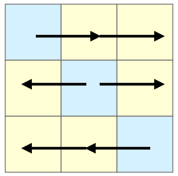
- Row and column subscripts are calculated *modulo n*
  - **Stage *k*:** $k' = (i+k) \bmod n$; c_{ij} += $a_{i,k'} * b_{k',j}$
- The dot product c_{ij} will be computed in the order
  - $a_{ii}*b_{ij} + a_{i,i+1}*b_{i+1,j} + \ldots + a_{i,n-1}*b_{n-1,j} + a_{i0}*b_{0j} + \ldots + a_{i,i-1}*b_{i-1,j}$
- Each process has to get the elements $a_{i,k'}$ and $b_{k',j}$ from the other processes in row *i* and column *j*
  - broadcast $a_{i,k'}$ to all processes in row *i*
  - calculate $a_{i,k'} * b_{k',j}$
  - shift the value of *b* one step up in column *j* (cyclically)

Illustration

- Stage *k*:
  - broadcast the element of *A k' steps to the right of the diagonal* to all processes in the same row
  - multiply the received element of *A* with the current element of *B*
  - shift the element of *B* up in the same column (circularly)

Broadcast A		Calculation in process (*i,j*)	Shift B

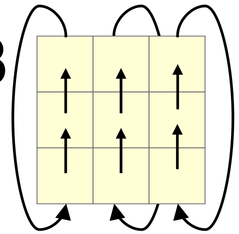
stage 0



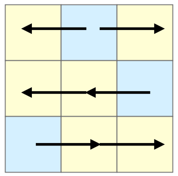
a_{00}
a_{11}
a_{22}

$c_{00} \mathrel{+}= a_{00}b_{00}$ $\quad c_{01} \mathrel{+}= a_{00}b_{01}$ $\quad c_{02} \mathrel{+}= a_{00}b_{02}$
$c_{10} \mathrel{+}= a_{11}b_{10}$ $\quad c_{11} \mathrel{+}= a_{11}b_{11}$ $\quad c_{12} \mathrel{+}= a_{11}b_{12}$
$c_{20} \mathrel{+}= a_{22}b_{20}$ $\quad c_{21} \mathrel{+}= a_{22}b_{21}$ $\quad c_{22} \mathrel{+}= a_{22}b_{22}$

B



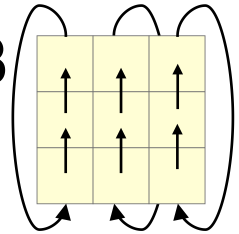
stage 1



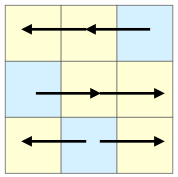
a_{01}
a_{12}
a_{20}

$c_{00} \mathrel{+}= a_{01}b_{10}$ $\quad c_{01} \mathrel{+}= a_{01}b_{11}$ $\quad c_{02} \mathrel{+}= a_{01}b_{12}$
$c_{10} \mathrel{+}= a_{12}b_{20}$ $\quad c_{11} \mathrel{+}= a_{12}b_{21}$ $\quad c_{12} \mathrel{+}= a_{12}b_{22}$
$c_{20} \mathrel{+}= a_{20}b_{00}$ $\quad c_{21} \mathrel{+}= a_{20}b_{01}$ $\quad c_{22} \mathrel{+}= a_{20}b_{02}$

B



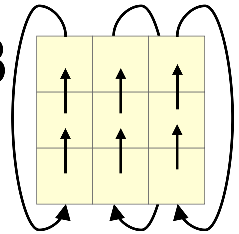
stage 2



a_{02}
a_{10}
a_{21}

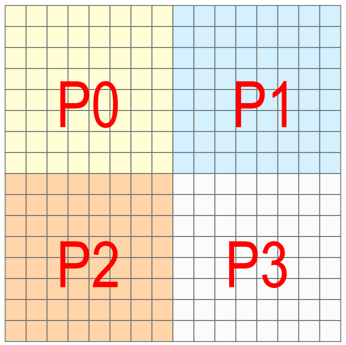
$c_{00} \mathrel{+}= a_{02}b_{20}$ $\quad c_{01} \mathrel{+}= a_{02}b_{21}$ $\quad c_{02} \mathrel{+}= a_{02}b_{22}$
$c_{10} \mathrel{+}= a_{10}b_{00}$ $\quad c_{11} \mathrel{+}= a_{10}b_{01}$ $\quad c_{12} \mathrel{+}= a_{10}b_{02}$
$c_{20} \mathrel{+}= a_{21}b_{10}$ $\quad c_{21} \mathrel{+}= a_{21}b_{11}$ $\quad c_{22} \mathrel{+}= a_{21}b_{12}$

B



Block decomposition

- Instead of giving one process one element we give each process a square block of elements
  - the matrices are of size *n* x *n*
  - we use a square grid of *p* processes where the number of rows and columns, $\sqrt{p}$, evenly divides *n*
- Each process stores a submatrix of size *n'* x *n'*, where $n' = n/\sqrt{p}$
- Example:
  - 16 x 16 matrix (*n*=16)
  - 4 processes arranged as a 2x2 grid (*p* = 4)
  - each process stores a 8 x 8 submatrix (*n'* = 8)

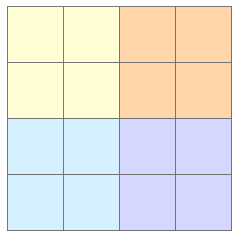


Submatrices

- Let the matrix A_{ij} be the $n' * n'$ submatrix of A whose first entry is $a_{i*n', j*n'}$
  - process (i,j) is assigned submatrices A_{ij}, B_{ij} and C_{ij}
  - submatrices can be multiplied as scalar elements, just use matrix multiplication instead of scalar multiplication
- Example:
  - 4x4 matrix divided among 4 processes
  - $n = p = 4$, $n' = 4 / \sqrt{4} = 2$

$$A_{0,0} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \qquad A_{0,1} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$$

$$A_{1,0} = \begin{pmatrix} a_{20} & a_{21} \\ a_{310} & a_{31} \end{pmatrix} \qquad A_{1,1} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$$



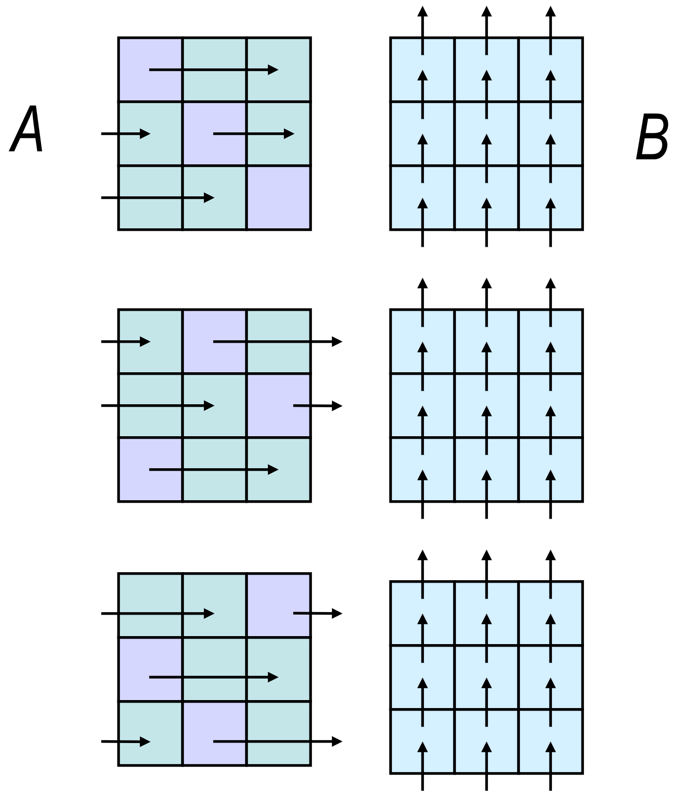
Fox's algorithm with block decomposition

- A_{ij}, B_{ij} and C_{ij} denotes submatrices as defined on previous slide
- Algorithm for process *(i,j):*

```
q = sqrt(p);
dest = ((i-1) mod q, j);  /* Process one step up */
source = (i+1) mod q, j); /* Process one step below */

for (stage=0; stage<q; stage++) {
  k_prime=(i+stage) mod q;
  Broadcast A[i,k_prime] to the processes in row i;
  C[i,j] += A[i,k_prime]*B[k_prime,j];  /* Multiply */
  Send B[k_prime,j] to dest;
  Receive B[(k_prime+1) mod q,j] from source;
}
```

Example: 3x3 processes



- Broadcast diagonal blocks of *A* to processes in the same row
- Multiply submatrices of *A* and *B* into *C*
- Shift blocks of B cyclically up one step

- Broadcast blocks one step to the right of the diagonal in *A* to processes in the same row
- Multiply submatrices *A* and *B* into *C*
- Shift blocks of B cyclically up one step

- Broadcast blocks two steps to the right of the diagonal in *A* to processes in the same row
- Multiply submatrices *A* and *B* into *C*
- Shift blocks of B cyclically up one step

Code for Fox's algorithm

```
dest = (my_row+q-1)%q;     /* Destination for circular shift in columns */
source = (my_row+1)%q;     /* Source for circular shift in columns */

/* Allocate storage for temporary local matrix */
tmp = (float *) malloc(sizeof(float)*N_local*N_local);

settozero(C_local, N_local);  /* Set the result matrix to zero */

for (stage=0; stage<q; stage++) {
  bcast_root = (my_row+stage)%q;     /* Process that does the broadcast */
  if (bcast_root == my_col) {
    /* Send to all other processes in the same row */
    MPI_Bcast(A_local, N_local*N_local, MPI_FLOAT, bcast_root, row_comm);
    /* Multiply the submatrices */
    matrixmult(A_local, B_local, C_local, N_local);
  } else {
    /* Receive submatrix of A from the process that broadcasts */
    MPI_Bcast(tmp, N_local*N_local, MPI_FLOAT, bcast_root, row_comm);
    /* Multiply it with own submatrix B_local */
    matrixmult(tmp, B_local, C_local, N_local);
  }
  /* Send submatrix of B up and receive a new from below */
  MPI_Sendrecv_replace(B_local, N_local*N_local, MPI_FLOAT, dest,
                       datatag, source, datatag, col_comm, &status);
}
```

Implementing Fox' s algorithm

- The implementation will contain the following steps
  - initialise MPI
  - check that we have a square number of processes (4, 9, 16, 25, ...)
  - read the size of the matrices *N*
  - check that the number of elements in the matrices is evenly divisible by the square root of the number of processes
  - allocate memory for the input matrices and the local sub-matrices
  - read in the input matrices *A* and *B* from files
  - create a 2-dimensional process grid
  - create communicators for rows and columns in the process grid
  - distribute the matrices *A* and *B* to the processes so that each process gets its own submatrices *A_local* and *B_local*
  - do the matrix multiplication with Fox's algorithm
  - collect the submatrices *C_local* from each process into a result-matrix *C*
  - write the result to a file
  - compare the result with a known correct result (available in a file)