HPCSE II

Advanced MPI

Hybrid codes

- We finally want to combine all we learned so far
  - **SIMD** vectorization on a single core
  - **Multithreading** on a single node
  - **Message passing** between nodes
- Example for PDGEMM
  - Distribute the matrices over nodes
  - Split the matrix into smaller blocks on each node
  - Finally vectorize the in-cache multiplication of the smallest blocks
- There is a potential problem: is MPI communication thread-safe?
  - Your MPI library might not care about thread-safety and you thus cannot make concurrent MPI calls
  - It can be worse: MPI might use an incompatible threading library to implement asynchronous communication. Your code might crash if it tries to launch a thread

Using MPI in a multithreaded context

- You need to call a special initialization function to use MPI with threads instead of the standard MPI_Init:

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided )
// required is the threading support you desire
// provided is what the library supports and can be less
```

Level of thread support	Description
MPI_THREAD_SINGLE	only a single thread can execute
MPI_THREAD_FUNNELED	The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
MPI_THREAD_SERIALIZED	The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, with no restrictions.

Why use hybrid MPI?

- Less memory use since threads can share data
  - N-body codes: no need to duplicate particle positions of other threads
  - PDE codes: no need for ghost cells within a node
- “Easier” to program
  - MPI requires explicit communication
  - Using threading within a node we can keep the MPI communication at a more coarse grained level
- Performance advantages
  - use multi-threaded libraries on a node, e.g. multi-threaded BLAS libraries

Hybrid programming styles

- Many ways to combine MPI processes and threads
  - One MPI process per node
  - One MPI process per socket (avoids NUMA issues)
  - Multiple MPI processes per socket, each with threads

- Many ways to use threads
  - “vector mode”: communication regions done by one thread followed by parallel loops done by all threads. Similar to using vector instructions.
  - “task mode”: one or more thread are responsible for communication, others do computation

Hybrid integration example

- Use OpenMP in Simpson integration

```
inline double simpson(double (*f) (double), double a, double b, unsigned int N)
{
  double      h = (b-a)/N;
  double result = ( f(a) + 4*f(a+h/2) + f(b) ) / 2.0;
  #pragma omp parallel for reduction(+ : result)
  for ( unsigned int i = 1; i <= N-1; ++i )
        result += f(a+i*h) + 2*f(a+(i+0.5)*h);
    return result * h / 3.0;
}
```

- And check for thread support in MPI part

```
int main(int argc, char** argv)
{
  int provided;
  MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,&provided);
  // we need to be able to communicate at least from the main thread
  assert(provided >= MPI_THREAD_FUNNELED);

  ...

  double delta = (p.b-p.a)/size;
  double result = simpson(func,p.a+rank*delta,p.a+(rank+1)*delta,p.nsteps/size);
  MPI_Reduce(rank==0 ? MPI_IN_PLACE : &result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank==0)
    std::cout << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

Hybrid programming styles

- How do we spawn a special OpenMP thread for communication?

```
#include <omp.h>

int main()
{
  #pragma omp parallel num_threads(2)
  {
    if (omp_getthread_num()==0)
    {
      // do communication
      ...
    }
    else
    {
    #pragma omp parallel
      {
        // do parallel work with remaining threads
        ...
      }

    }
  }
}
```

Enabling hybrid MPI

- Many platforms require special linker or runtime options

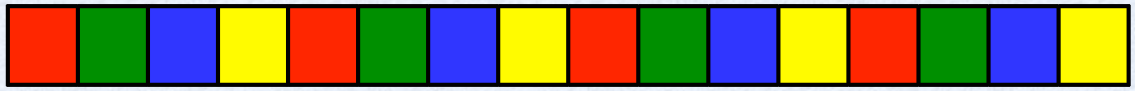
Platform	Enabling multithreaded MPI
Intel MPI	Compile and link with mpiicpc -mt_mpi
Cray	Set the environment variable MPICH_MAX_THREAD_SAFETY to one of single, funneled, serialized, multiple
MPAVICH2	Set environment variable MV2_ENABLE_AFFINITY=0
OpenMPI	——

Distributed linear algebra

- Distributed storage
- Dense linear algebra
  - vector operations and matrix additions
  - matrix-vector multiplication
  - matrix-matrix multiplication
  - LU factorization
- Sparse linear algebra
  - matrix-vector multiplication

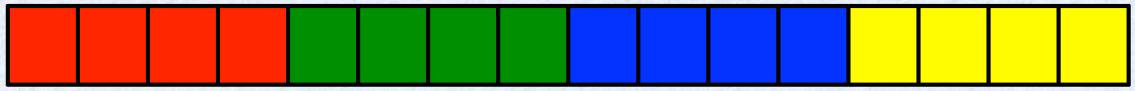
Distributed vector storage

- Cyclic distribution



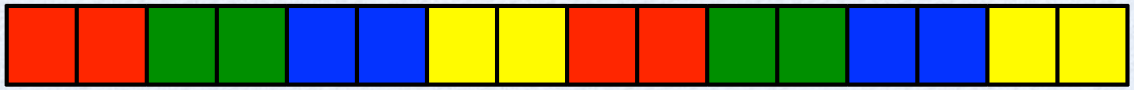
  - element *i* stored on rank $\text{rank}(i) = i \bmod P$
  - local index of element *i* is $\text{local}(i) = \lfloor i / P \rfloor$

- Block distribution



  - element *i* stored on rank $\text{rank}(i) = \lfloor i / b \rfloor$ where $b = \lceil N / P \rceil$
  - local index of element *i* is $\text{local}(i) = i \bmod b$

- Block-cyclic distribution



A distributed vector

```
template <typename T, typename Allocator = hpc12::aligned_allocator<T,64> >
class dvector : public std::vector<T,Allocator>
{
public:
  typedef T value_type;

  dvector(std::size_t n, MPI_Comm c = MPI_COMM_WORLD)
  : comm_(c)
  , global_size_(n)
  {
    int s;
    MPI_Comm_rank(comm_,&rank_);
    MPI_Comm_size(comm_,&s);
    // calculate the block size and resize the local block
    block_size_ = (global_size_+s-1)/s;
    if (rank_*block_size_ < global_size_);
      this->resize(std::min(block_size_,global_size_-rank_*block_size_));
  }

  value_type const* data() const { return this->empty() ? 0 : &this->front(); }
  value_type* data() { return this->empty() ? 0 : &this->front();}
  std::size_t global_size() const { return global_size_;}
  std::size_t offset() const { return rank_ * block_size_;}
  std::size_t block_size() const { return block_size_;}
  MPI_Comm& communicator() const { return comm_;}

private:
  mutable MPI_Comm comm_;
  int rank_;
  std::size_t global_size_;
  std::size_t block_size_;
};
```

Distributed vector operations

- _COPY, _SCAL, _AXPY can be done locally on each segment

```
inline void dscal(double alpha, dvector<double>& x)
{
  // just scale the local block
  int size = x.size();
  dscal_(size, alpha, x.data(), 1);
}
```

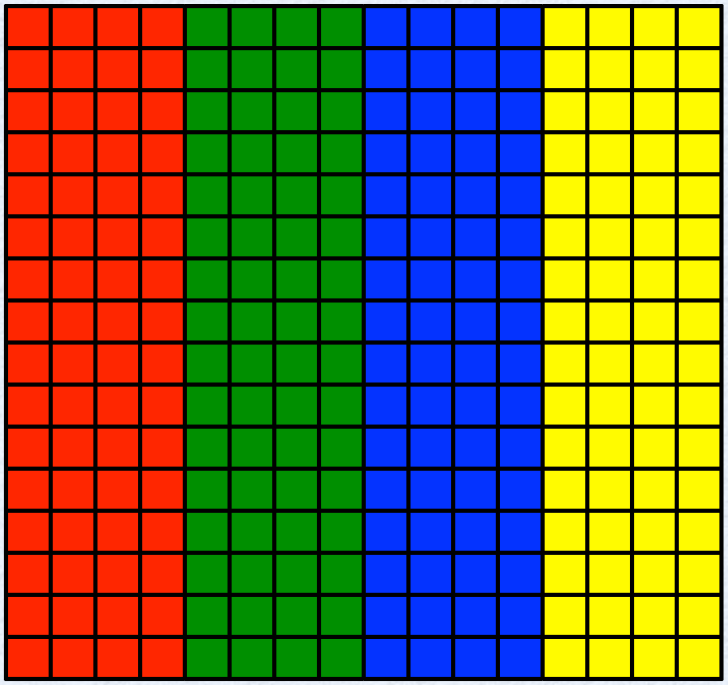
```
inline void daxpy(double alpha, dvector<double>& x, dvector<double>& y)
{
  // just scale and add the local block
  int size = x.size();
  daxpy_(size, alpha, x.data(), 1, y.data(), 1);
}
```

- _DOT can be done locally **followed by a reduction**

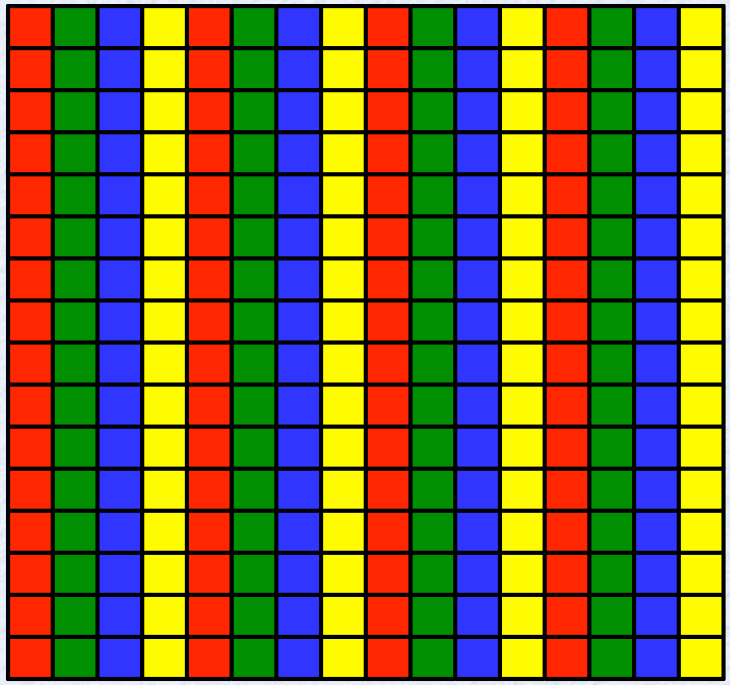
```
inline double ddot(dvector<double>& x, dvector<double>& y)
{
  assert(x.size() ==  y.size());
  int size = x.size();
  // get the local dot product
  double result = ddot_(size, x.data(), 1, y.data(), 1);

  // and perform a reduction
  MPI_Allreduce(MPI_IN_PLACE,&result,1,MPI_DOUBLE,MPI_SUM, x.communicator());
  return result;
}
```

Distributed matrix storage (1)

- Block column distribution

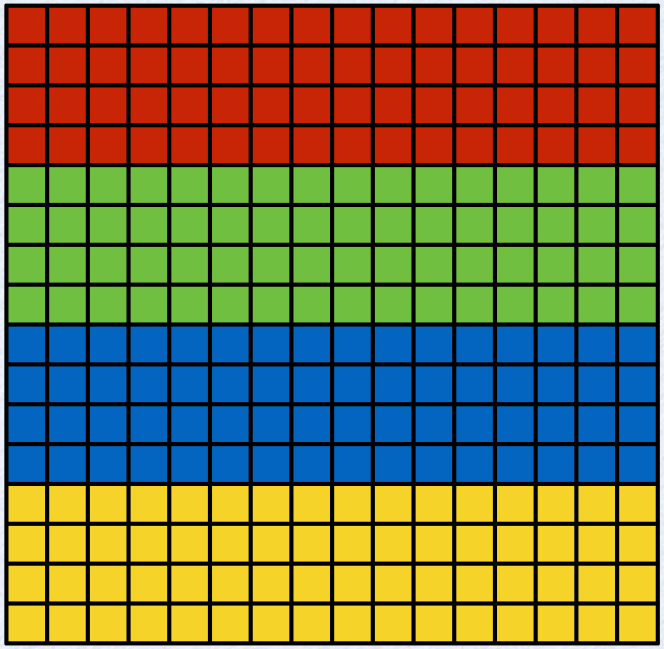


- Cyclic column distribution

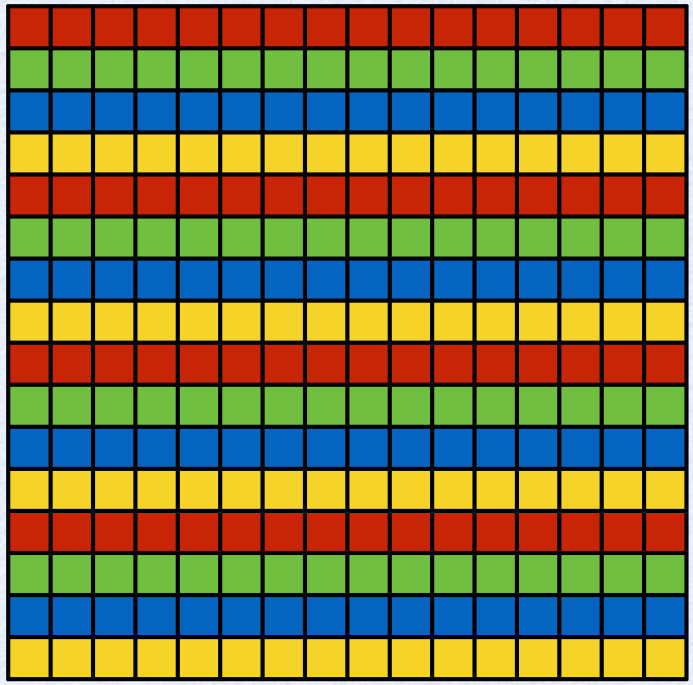


Distributed matrix storage (2)

- Block row distribution

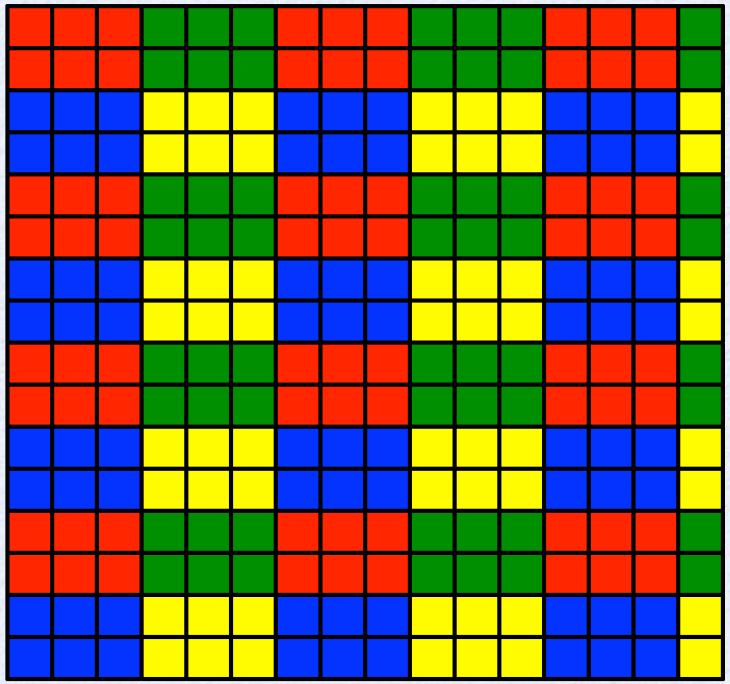


- Cyclic row distribution

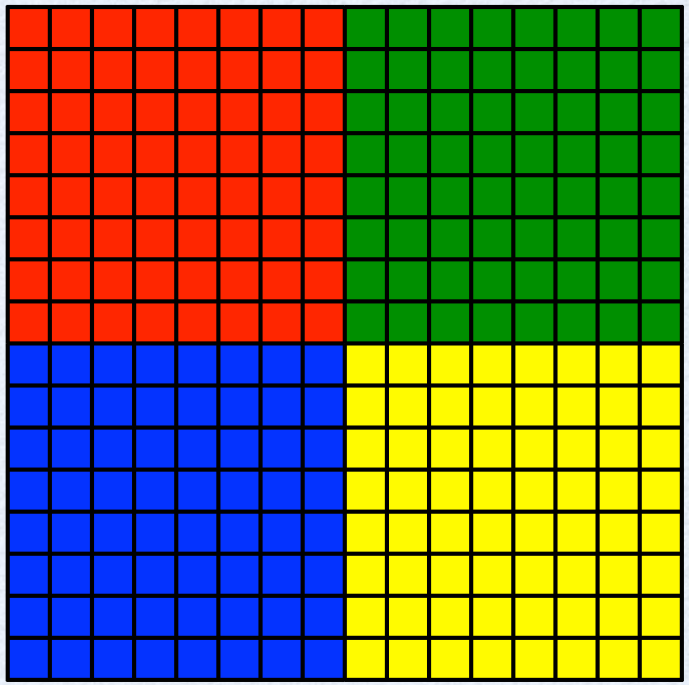


Distributed matrix storage (3)

- Block cyclic distribution
  - 3x2 blocks
  - 2x2 process array



- Block cyclic distribution used in our example codes
  - a single tiling
  - perfect fit



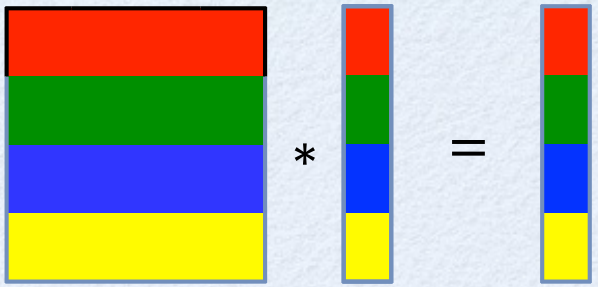
Distributed matrix operations

- We will now look at several matrix operations
  - matrix additions are the same as _AXPY!
  - matrix-vector multiplications
  - matrix-matrix multiplications
  - LU decomposition
  - sparse matrix-vector multiplications

- Being forced to make use of data locality and minimizing communication we will get better ideas for the matrix multiplication that will also help us for the multithreaded version

Parallel gemv version 1

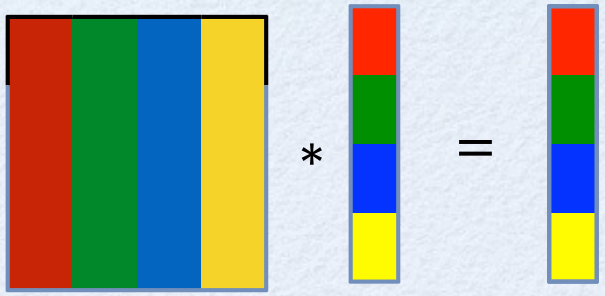
- Block-row distribution
  - Gather all parts of *x* locally
  - and then perform the local multiplications



```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// we want a simple size that can be divided evenly by the number
// of ranks to keep the code simple
int block_size = N/size;
assert(N % size ==0);

// block distribution of the vectors
std::vector<double> x(block_size), y(block_size);

// block row distribution for the matrix: keep only N/size rows
matrix_type A(block_size,N);

...

//Gather all pieces into a big vector and then do a multiplication
std::vector<double> fullx(N);
MPI_Allgather(&x[0], x.size(), MPI_DOUBLE, &fullx[0], x.size(), MPI_DOUBLE, MPI_COMM_WORLD);
dgemv(A,fullx,y);
```

Parallel gemv version 2

- Block-column distribution
  - Perform local multiplications
  - Add all parts (reduction)
  - Finally scatter the results

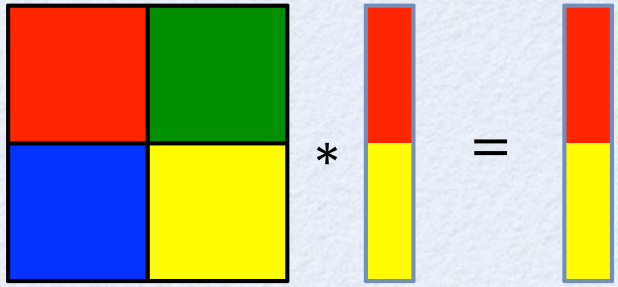


```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// we want a simple size that can be divided evenly by the number
// of ranks to keep the code simple
int block_size = N/size;
assert(N % size ==0);

// block distribution of the vectors
std::vector<double> x(block_size), y(block_size);

// block column distribution for the matrix: keep only N/size columns
matrix_type A(N,block_size);

...

// do a local multiplication, obtaining a full vector
// and then reduce-scatter the result
std::vector<double> fully(N);
dgemv(A,x,fully);
std::vector<int> recvcounts(size, block_size);
MPI_Reduce_scatter(&fully[0],&y[0], &recvcounts[0], MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Parallel gemv version 3

- Block-cyclic distribution on q x q array
  - store vector on diagonal blocks
  - broadcast x_j along column j
  - multiply
  - reduce y_i along row i

$$y_i = \sum_{j=0}^{q-1} A_{i,j} x_j$$



```
// do the multiplication:
// 1. broadcast along columns
MPI_Bcast(&x[0], x.size(), MPI_DOUBLE, col, col_comm);

// 2. do local multiplication
dgemv(A,x,y);

// 3. reduce along row
MPI_Reduce(row==col ? MPI_IN_PLACE : &y[0], &y[0], y.size(), MPI_DOUBLE, MPI_SUM, row, row_comm);
```

Now with communicator construction

```
int N=1024;
int num_blocks = std::sqrt(size);;
int block_size = N/std::sqrt(size);
assert(size = num_blocks * num_blocks);
assert(N % block_size == 0);

// build a cartesian topology
int periodic[2] = {true, true};
int extents[2] = {num_blocks, num_blocks};
MPI_Comm comm;
MPI_Cart_create(MPI_COMM_WORLD, 2, extents, periodic, true, &comm);

// get my row and column number
int coords[2];
MPI_Cart_coords(comm, rank, 2, coords);
int row = coords[0];
int col = coords[1];

// build communicators for rows and columns
MPI_Comm row_comm, col_comm, diag_comm;
MPI_Comm_split(comm,row,col,&row_comm);
MPI_Comm_split(comm,col,row,&col_comm);

// block distribution of the vectors on the diagonal
vector_type x(block_size), y(block_size);
...

// allocate a block of the matrix everywhere and fill it in
matrix_type A(block_size,block_size);
  ...

// do the multiplication:
MPI_Bcast(&x[0], x.size(), MPI_DOUBLE, col, col_comm);
dgemv(A,x,y);
MPI_Reduce(row==col ? MPI_IN_PLACE : &y[0], &y[0], y.size(), MPI_DOUBLE, MPI_SUM, row, row_comm);
```

Blocking of matrix multiplies

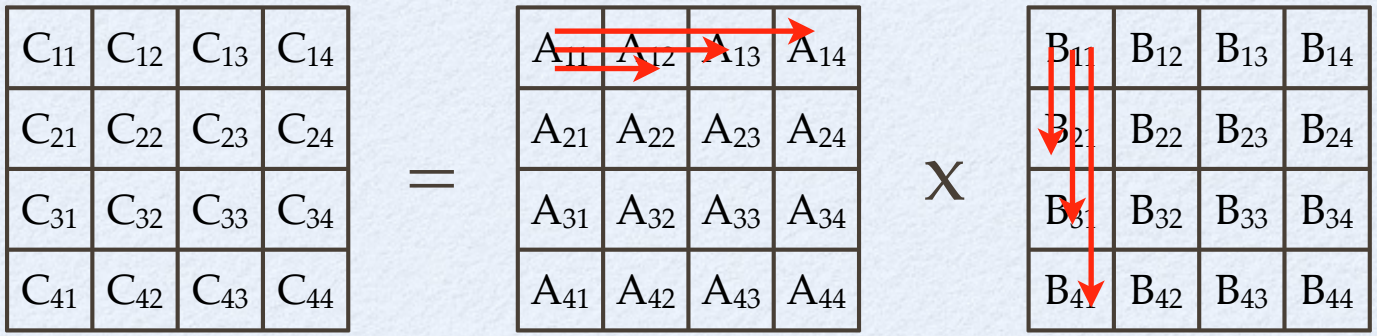
- The solution: block the operations and do *b* of these matrix-vector multiplications or vector-vector outer products at once. Data is then reused *b* times and thus we do *bN* operations for *N* memory accesses.

- Consider the various matrix multiplications we did:
  - **Block row** distribution required an all-gather of the full vector
    Block column distribution built a full-sized vector
    - We need the full matrix on one node!
    - might run out of memory!
    - lots of communication!

  - **Block cyclic** distribution needed memory only for a row or column
    - less memory requirements
    - less network traffic

Parallel matrix multiplication $C = A \times B$

- Block the matrices in a two-dimensional array layout



- Need to send data:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + A_{i3}B_{3j} + A_{i4}B_{4j}$$

  A_{ij} is needed on on all rows *i*

  B_{ij} is needed on on all columns *j*

- Do an all-gather along rows and columns, then calculate the local C_{ij}

Distributed matrix multiplication

```
// prepare row and column communicators like before
...

// allocate a block of the matrix everywhere and fill it in
matrix_type A(block_size,block_size);
matrix_type B(block_size,block_size);
matrix_type C(block_size,block_size);

for (int i=0; i<block_size; ++i)
  for (int j=0; j<block_size; ++j) {
    A(i,j) = i+j+(row+col)*block_size;
    B(i,j) = i+j+(row+col)*block_size;
    C(i,j) = 0.;
  }

// allocate working space for the block row of A
// and the block column of B

vector_type Arow(block_size*block_size*q);
vector_type Bcol(block_size*block_size*q);

// 1. gather rows and columns
MPI_Allgather(A.data(),block_size*block_size,MPI_DOUBLE,
              &Arow[0],block_size*block_size,MPI_DOUBLE,row_comm);
MPI_Allgather(B.data(),block_size*block_size,MPI_DOUBLE,
              &Bcol[0],block_size*block_size,MPI_DOUBLE,col_comm);

// 2. do all multiplications
for (int i=0; i<q; ++i)
  dgemm_('N','N',block_size,block_size,block_size,1.,
         &Arow[i*block_size*block_size],block_size,
         &Bcol[i*block_size*block_size],block_size,
         1., C.data(),block_size);
```

Better distributed matrix multiplies

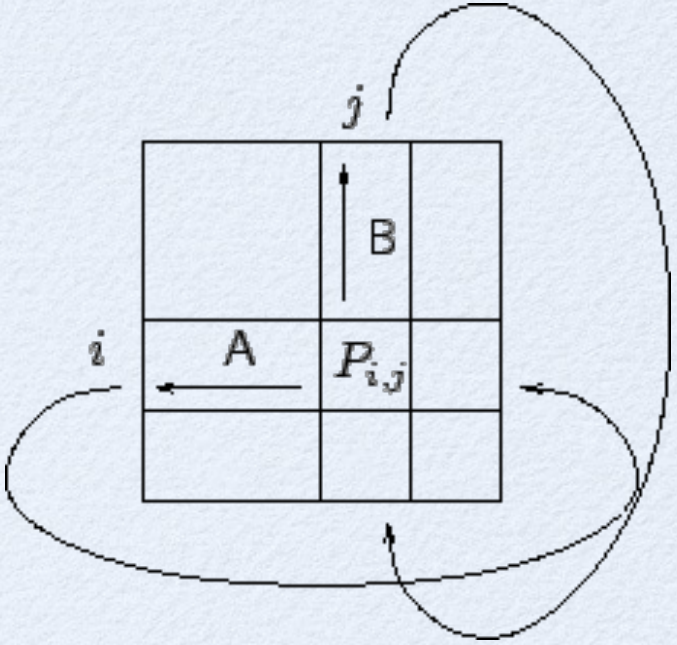
- This was not optimal yet!
  - We need memory for a whole block-row and block-column: $N \times N / \sqrt{p}$ instead of just a block of size $N \times N / p$
  - We cannot overlay computation and communication
- Solution: don't gather all data at first but shift the blocks A_{ij} and B_{ij} through the network, always having only one on each rank.
  - Naïve version: just broadcast one block after the other as it is needed
  - First such algorithm invented 1969 by Cannon.
  - Optimal algorithm invented 2011 by Solomonik and Demmel http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-10.pdf

Distributed matrix multiplication

```
// prepare row and column communicators like before
...

// allocate a block of the matrix everywhere and fill it in
matrix_type A(block_size,block_size);
matrix_type B(block_size,block_size);
matrix_type C(block_size,block_size);

for (int i=0; i<block_size; ++i)
  for (int j=0; j<block_size; ++j) {
    A(i,j) = i+j+(row+col)*block_size;
    B(i,j) = i+j+(row+col)*block_size;
  }

// allocate working space for the block row of A
// and the block column of B

matrix_type Atmp(block_size,block_size);
matrix_type Btmp(block_size,block_size);

// loop over all block
for (int i=0; i<q; ++i) {
  // 1. broadcast block along row and column
  if (i==col)
    Atmp=A;
  if (i==row)
    Btmp=B;
  MPI_Bcast(Atmp.data() ,block_size*block_size,MPI_DOUBLE,i,row_comm);
  MPI_Bcast(Btmp.data() ,block_size*block_size,MPI_DOUBLE,i,col_comm);

  // 2. do all multiplications
  dgemm_('N','N',block_size,block_size,block_size,1.,
         Atmp.data(),block_size,Btmp.data(),block_size,
         1., C.data(),block_size);
}
```

Cannon's algorithm (1969)

- Split the matrix into blocks like before on a q x q array
- Align the blocks so that we can start multiplying the right blocks locally:
  - move the i-th block row of A i blocks to the left
  - move the j-th block column of B j blocks up
- Repeat q times:
  - Multiply the local blocks
  - Shift A one block to the left
  - Shift B one block up



- Needs much less memory and communication
- Allows overlaying communication and computation
- Easy to implement using MPI_Sendrecv and cartesian communicators