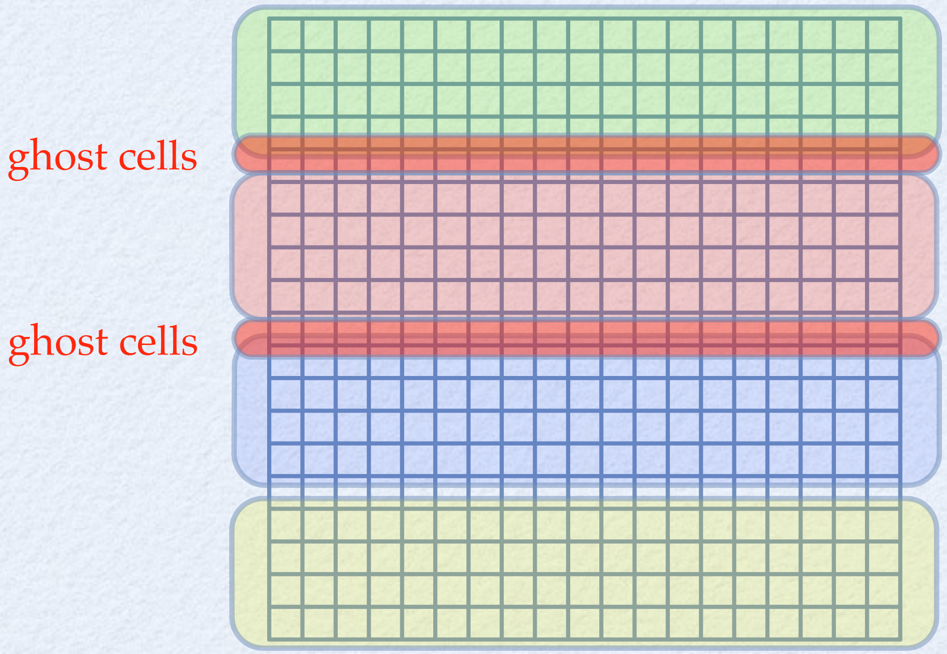
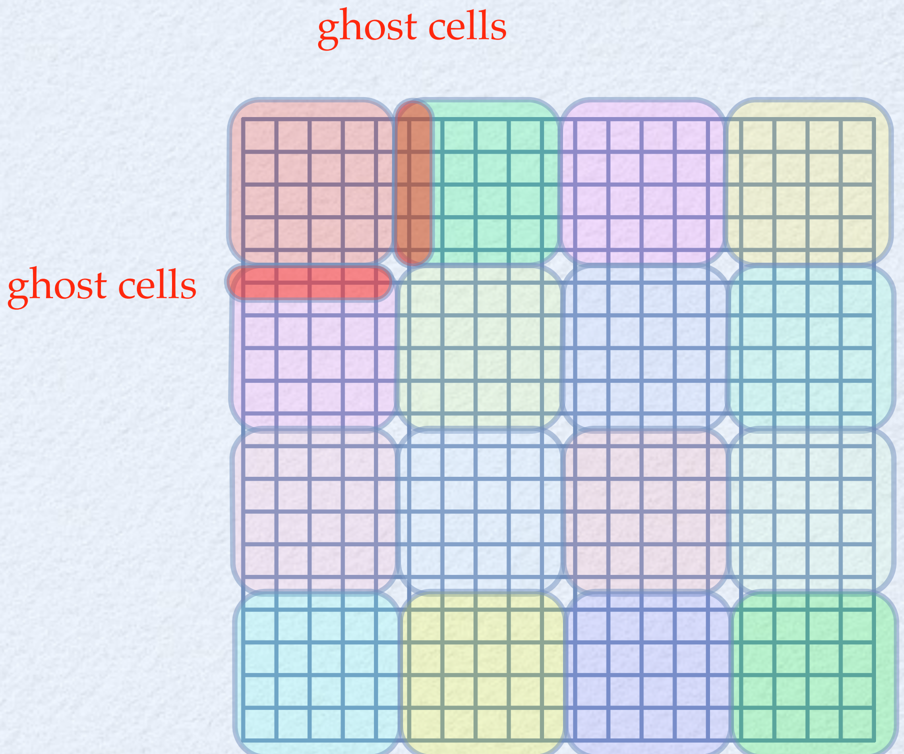
HPCSE II

Advanced MPI

Recall domain decomposition and ghost cells

- How do we best exchange boundary values with the neighboring ranks?
  - In 1D it was just a single number and was easy
  - Sometimes we might be lucky and they could be contiguous arrays
- What shall we do in the general case?
  - pack them into buffers?
  - or just describe to MPI where they are in memory?





Recall sending the parameters

- We had three options, none was ideal
  - three individual broadcasts: wasteful since three communications
  - packing it into a buffer: wasteful since it involves copying
  - sending the struct bitwise: dangerous since it assumes homogeneity
- What we want to do here and for the ghost cells is to send non-contiguous or heterogeneous data without copying.

Broadcast

- MPI provides a collective broadcast operation

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
// broadcast the data from the root rank to all others
```

- We can use this to broadcast the data

```
// and then broadcast the parameters to the other ranks
 MPI_Bcast(&a, 1, MPI_DOUBLE,0, MPI_COMM_WORLD);
 MPI_Bcast(&b, 1, MPI_DOUBLE,0, MPI_COMM_WORLD);
 MPI_Bcast(&nsteps, 1, MPI_INT,0, MPI_COMM_WORLD);
```

- This is inefficient since we use three broadcasts.
- We will later pack all parameters into one buffer and broadcast that buffer.

Sending it bitwise

- The dangerous solution: pack it all into a struct and send it bitwise
- This assumes a homogeneous machine with identical integer and floating point formats.

```
// define a struct for the parameters
struct parms {
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  int nsteps; // number of subintervals for integration
};

parms p;

// read the parameters on the master rank
if (rank==0);
  std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms as bytes - warning, not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```

Packing data into a buffer

- Pack the input data, broadcast it and unpack

```
// create a buffer and pack the values.
// first get the size for the buffer and allocate a buffer
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT,    MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// pack the values into the buffer on the master
if (rank==0) {
  int pos=0;
  MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  assert ( pos <= buffer_size );
}

// broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
assert ( pos <= buffer_size );

// and finally delete the buffer
delete[] buffer;
```

Recall sending the parameters

- We had three options, none was ideal
  - three individual broadcasts: wasteful since three communications
  - packing it into a buffer: wasteful since it involves copying
  - sending the struct bitwise: dangerous since it assumes homogeneity
- What we want to do here and for the ghost cells is to send non-contiguous or heterogeneous data without copying.
- The solution are MPI datatypes: describe your data layout to MPI and MPI uses that information in the communication.

```
struct parms {
  double a;
  double b;
  int nsteps;
};
```

type	count	offset
MPI_DOUBLE	2	0
MPI_INT	1	16

Building an MPI data type

- The most general function is MPI_Type_create_struct, taking numbers, offsets and types

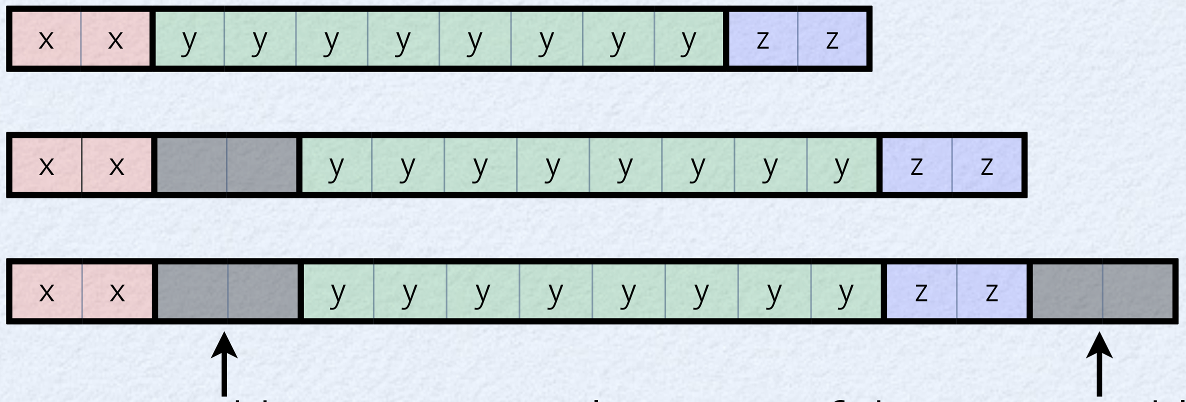
```
// define a struct for the parameters
struct parms {
  double a;             // lower bound of integration
  double b;             // upper bound of integration
  int nsteps; // number of subintervals for integration
};

// describe this struct through sizes, offsets and types
// and create an MPI data type
// still dangerous since it assumes that we know any potential padding
MPI_Datatype parms_t;
int          blocklens[2]  =  {2,1};
MPI_Aint     offsets[2]    =  {0,2*sizeof(double)};
MPI_Datatype types[2]      =  {MPI_DOUBLE, MPI_INT};
MPI_Type_create_struct(2, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t); // finish building the type

parms p;

// read the parameters on the master rank
if (rank==0);
  std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms now using our type
MPI_Bcast(&p, 1, parms_t, 0, MPI_COMM_WORLD);

// and now free the type
MPI_Type_free(&parms_t);
```

Alignment and padding

- This code was dangerous since we assumed that we know how the compiler lays out a struct in memory.

```
struct parms {
  short  x;
  double y;
  short  z;
};
```

- We might be wrong due to padding and alignment. Consider the following three examples of how this could be stored in memory



Padding to ensure alignment of the next variable

Safer way of using MPI_Type_create_struct

- To get the right offsets and size we take actual addresses
  - use MPI_Get_address to convert pointers to integers
  - specify lower bound and upper bound of the struct

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

parms p;

// describe the struct through sizes, offsets and types
// the safe way getting addresses

MPI_Aint p_lb, p_a, p_nsteps, p_ub;
MPI_Get_address(&p,        &p_lb);      // start of the struct is the lower bound
MPI_Get_address(&p.a,      &p_a);       // address of the first double
MPI_Get_address(&p.nsteps, &p_nsteps); // address of the integer
MPI_Get_address(&p+1,      &p_ub);      // start of the next struct is the upper bound

int          blocklens[] =  {0, 2, 1, 0};
MPI_Datatype types[]     =  {MPI_LB, MPI_DOUBLE, MPI_INT, MPI_UB};
MPI_Aint     offsets[]   =  {0, p_a-p_lb, p_nsteps-p_lb, p_ub-p_lb};

MPI_Datatype parms_t;
MPI_Type_create_struct(4, blocklens, offsets, types,&parms_t);
MPI_Type_commit(&parms_t);
```

MPI_Type_create_struct

- The declaration of the functions used in the previous examples

```
int MPI_Type_create_struct(int count, int blocklengths[], MPI_Aint offsets[],
                           MPI_Datatype types[], MPI_Datatype *newtype)
// builds an MPI data type for a data type for a general data structure given by
// types, counts (blocklengths) and their offsets relative to the start of the data structure

int MPI_Get_address(void *location, MPI_Aint *address)
// converts a pointer to the integer type used internally by MPI to store pointers

int MPI_Type_commit(MPI_Datatype *datatype)
// commits the data type: finished building it. It can now be used.

int MPI_Type_free(MPI_Datatype *datatype)
// frees the data type, releasing any allocated memory
```

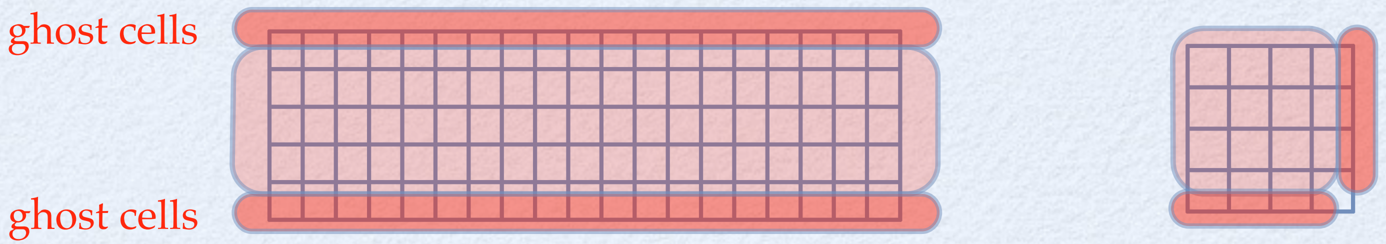
- We can use MPI_Type_create_struct to send the contents of linked lists
  - we view the whole memory as a huge struct from which we send some select data
  - thus give absolute addresses as offsets
  - pass **MPI_BOTTOM** as the buffer pointer in communication to indicate that the type uses absolute addresses

Receiving a list into a vector

```
if(num==0) {
  // receive data into a vector and print it
  std::vector<int> data(10);
  MPI_Status status;
  MPI_Recv(&data[0], 10, MPI_INT, 1, 42, MPI_COMM_WORLD, &status);
  for (int i=0; i < data.size(); ++i)
    std:: cout << data[i] << "\n";
}
else {
  // fill a list with the numbers 0-9 and send it
  std::list<int> data;
  for (int i=0; i<10; ++i)
    data.push_back(i);

  std::vector<MPI_Datatype> types(10,MPI_INT);
  std::vector<int>          blocklens(10,1);
  std::vector<MPI_Aint>     offsets;

  for (int& x : data) {
    MPI_Aint address;
    MPI_Get_address(&x, &address); // use absolute addresses
    offsets.push_back(address);
  }

  MPI_Datatype list_type;
  MPI_Type_create_struct(10, &(blocklens[0]), &offsets[0], &types[0] ,&list_type);
  MPI_Type_commit(&list_type);

  MPI_Send(MPI_BOTTOM, 1, list_type, 0, 42, MPI_COMM_WORLD);

  MPI_Type_free(&list_type);
}
```

MPI data types for ghost cells

- The ghost cells in a 2D array or column and rows in a matrix can be described as strided vectors



```
int MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype *new_type_p)
// build an MPI datatype for a contiguous array

int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype old_type, MPI_Datatype *newtype_p)
// build an MPI datatype for a vector array of blocklength contiguous entries that are
// spaced at a given stride. stride is like the leading dimension in BLAS and specifies
// the distance between blocks

int MPI_Type_create_hvector(int count, int blocklen,  MPI_Aint stride,
                            MPI_Datatype old_type, MPI_Datatype *newtype_p)
// like MPI_Type_vector but now the stride is given in bytes
```

Row and column data types

- We can use this to create data types for rows and columns of a matrix, and similarly for slices of an array

```
hpc12::matrix<double,hpc12::column_major> a(4,4);

MPI_Datatype row, col;

MPI_Type_contiguous(4,       MPI_DOUBLE, &col);
MPI_Type_vector    (4, 1, 4, MPI_DOUBLE, &row);

MPI_Type_commit(&row);
MPI_Type_commit(&col);

// use them
// ...
// and finally free them

MPI_Type_free(&row);
MPI_Type_free(&col);
```

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Subarrays

- More general is the creation of subarrays, especially for boundary layers and ghost cells

```
int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[],
                             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
// build an MPI datatype for a subarray of a larger array:
//    ndims:    number of dimensions
//    sizes:    extent of the full array in each dimension
//    subsizes: extent of the subarray in each dimension
//    starts:   starting index of the subarray
//    order:    array storage order, can be either of MPI_ORDER_C or MPI_ORDER_FORTRAN
```

- Use it for the 2D diffusion equation parallelized using MPI

Indexed data types

- Finally, we want to send just some elements of an array. For example, send some particles to a different cell list.

```
int MPI_Type_indexed(int count, int blocklens[], int indices[],
                     MPI_Datatype old_type, MPI_Datatype *newtype)
// build an MPI datatype selecting specific entries from a contiguous array. Starting a
// at each of the given indices a number of elements given in the corresponding entry
// of blocklens is chosen.

int MPI_Type_create_hindexed(int count, int blocklens[], MPI_Aint displacements[],
                             MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but now instead of indices the displacement in bytes from the
// start of the array is specified

int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[],
                                  MPI_Datatype oldtype, MPI_Datatype *newtype)
// same as MPI_Type_indexed but with constant sized blocks
```

Parallelizing your codes with MPI

- Let us now discuss how to parallelize the various codes you wrote so far and what MPI features to use:
  - Monte Carlo simulations
  - Partial differential equations
  - N-body codes with long range forces
  - N-body codes with short range forces
  - Linear algebra

- How would you do the following with MPI datatypes?
  - send some particles from one node to another? (hint: particles1.cpp)
  - send just the positions of some particles to another node? (hint: particles2.cpp)
  - send the positions of one particle to another node and receive them separately as x, y, and z coordinates?

Groups and Communicators

- Imagine we want to split a computation into individual tasks that run on subsets of the ranks:
  - do multiple integrations at the same time
  - operate on rows or columns of a matrix
  - operate on slices of a 3D mesh
- We want to split the ranks into groups and build a new communicator for each group
- We can then do collective operations within a group instead of within all ranks

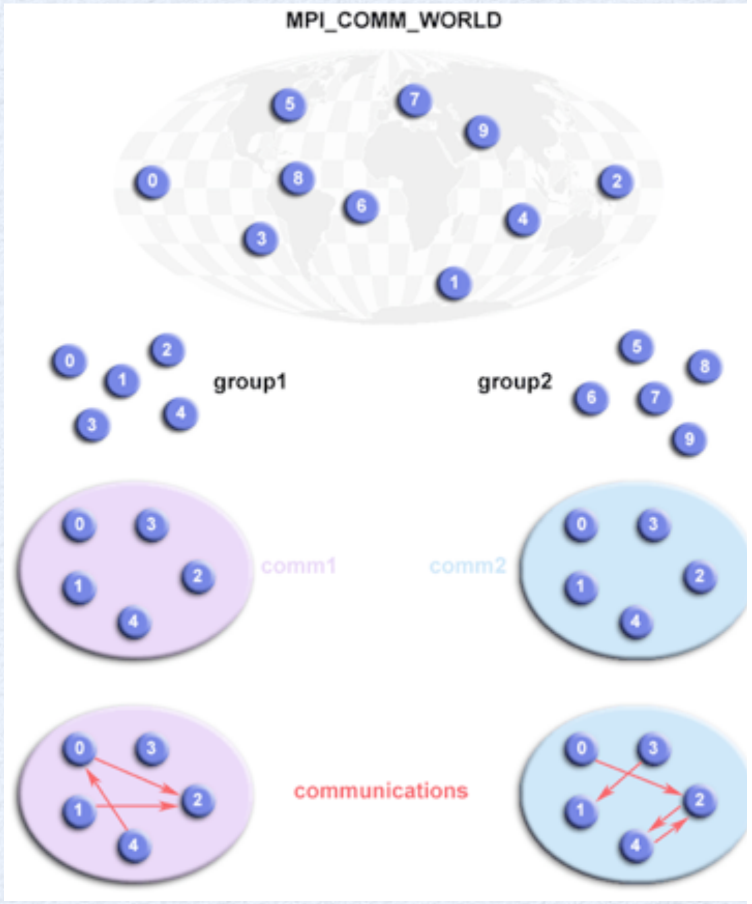


Image © CSCS

Simpson using a communicator

- Simpson integration by MPI using a communicator that might be other than MPI_COMM_WORLD

```
double parallel_simpson(MPI_Comm comm, parms p)
{
  // get the rank and size for the current communicator
  int size;
  int rank;
  MPI_Comm_size(comm,&size);
  MPI_Comm_rank(comm,&rank);

  // integrate just one part on each rank
  double delta = (p.b-p.a)/size;
  double result = simpson(func,p.a+rank*delta,p.a+(rank+1)*delta,p.nsteps/size);

  //  collect the results to all ranks
  MPI_Allreduce(MPI_IN_PLACE, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
  return result;
}
```

Three Simpson integrations at once

```
int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // we want to do three integrals at once
  parms p[3];
  ...

  // split the ranks into three groups
  int which = rank % 3;
  MPI_Comm comm;
  MPI_Comm_split(MPI_COMM_WORLD, which, rank, &comm);

  // do the integration in each group
  double result = parallel_simpson(comm,p[which]);

  // only the master for each group prints
  int grouprank;
  MPI_Comm_rank(comm, &grouprank);
  if (grouprank==0)
    std::cout << "Integration " << which << " results in " << result << std::endl;

  //  free the type and the new communicator
  MPI_Comm_free(&comm);

  MPI_Finalize();
  return 0;
}
```

Creating and destroying communicators

- The most important functions for communicators

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )

int MPI_Comm_size( MPI_Comm comm, int *size )

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
// compares two communicators to test is they are the same, i.e. they have the same ranks
// in the same order

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
// duplicates a communicator.
// this is a collective communication that needs to be called by all ranks.

int MPI_Comm_free(MPI_Comm *comm)
// frees a communicator

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
// splits a communicator into subcommunicators.
// ranks with the same color are grouped together and sorted within each group by key.
// this is a collective communication that needs to be called by all ranks.

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
// creates a new communicator based on group that is a subgroup of the ranks in comm.
// this function allows more flexible creation of subcommunicators than MPI_Comm_split.
// this is a collective communication that needs to be called by all ranks.
```

Working with groups (1)

- There are **many** useful functions for group creation

```
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
// are similar to the corresponding communicator functions

int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
// translates ranks between group: given a set of ranks1 in group1 it sets their ranks in group2
// in the array ranks2, or sets them to MPI_UNDEFINED if no correspondence exists

int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
// extracts the group from a communicator

int MPI_Group_free(MPI_Group *group)

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
// newgroup is the union, intersection, or difference of the given groups
```

Working with groups (2)

- selectively choosing ranks

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
// create a newgroup containing only the given ranks of a group

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
// create a newgroup containing all except the given ranks of a group

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing only the given ranges of ranks of a group

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)
// create a newgroup containing all except the given ranges of ranks of a group
```

- ranges are given as triples (first, last, stride), and a range includes the ranks

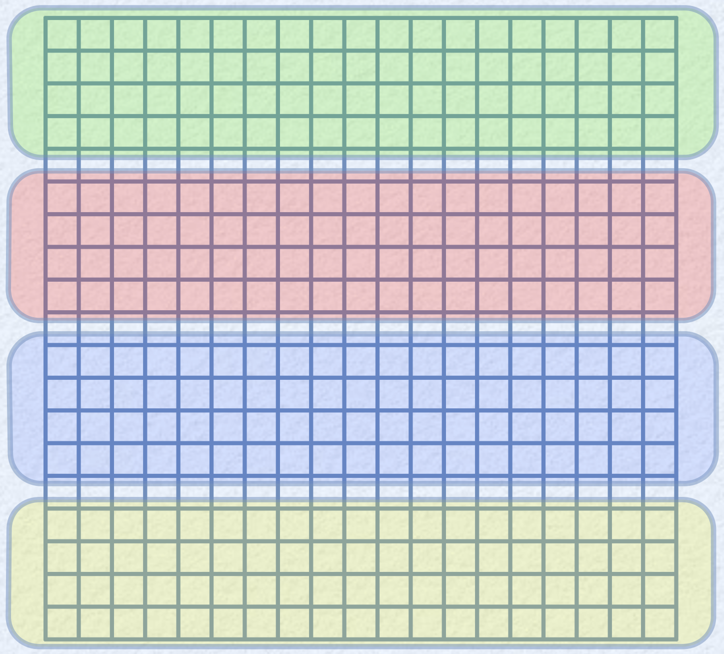
$$\text{first}, \text{first} + \text{stride}, \text{first} + 2\,\text{stride}, \ldots, \text{first} + \left\lfloor \frac{\text{last} - \text{first}}{\text{stride}} \right\rfloor \text{stride}$$

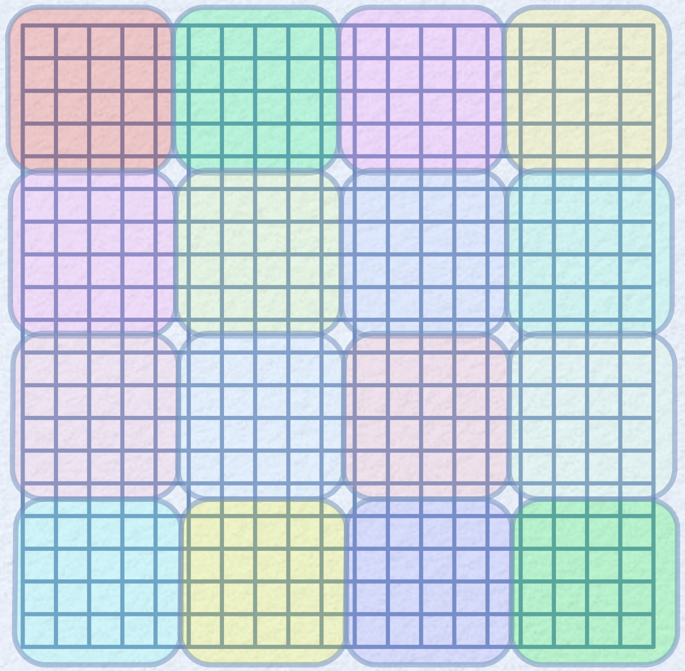
Using groups

- Discussion: how would you use groups for cell lists?
  - where do they make sense?
  - how do you create them?
  - how many groups do we need?

Finding the rank of the neighbor

- Easy in a one-dimensional layout
- Harder in two and more dimensions
- Even harder on irregular meshes





- MPI topologies are the solution to easily finding neighbors

MPI topologies

- A (virtual) topology describes the “connectivity” of MPI processes in a communicator. There may be no relation between the physical network and the process topology.
- Two main types
  - **Cartesian topology**: each process is “connected” to its neighbors in a virtual grid. Nodes are labeled by cartesian indices, boundaries can be cyclic (periodic).
  - **Graph topology:** an arbitrary connection graph
- Topologies are essentially a simple graph library built into MPI

Cartesian topologies: MPI_Cart_create

- To work with a regular mesh with row-major ordering we create a cartesian communicator

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                    int reorder, MPI_Comm *comm_cart)
```

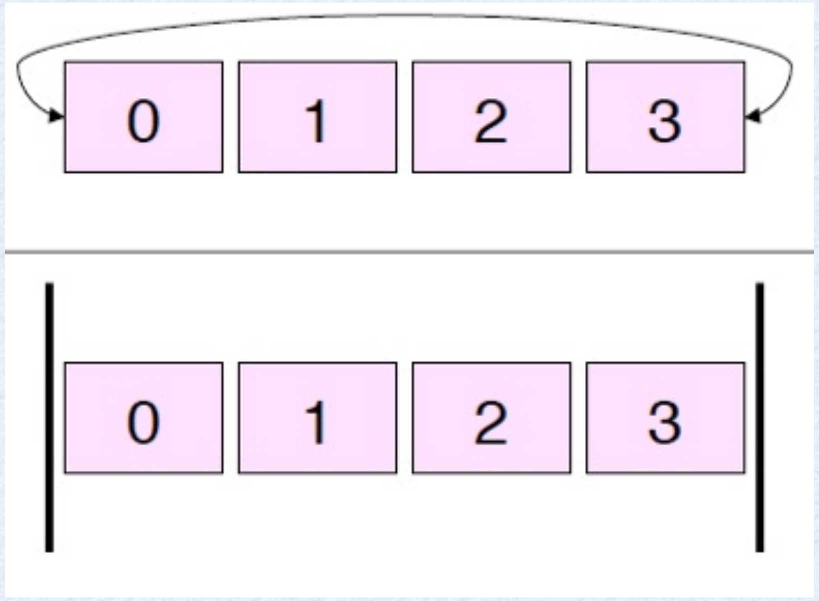
  - **comm_old**: the original communicator
  - **ndims**: number of dimensions
  - **dims**: integer array specifying the number of processes in each dimension
  - **periods**: integer array of boolean values whether the grid is periodic in that dimension
  - **reorder**: boolean flag whether the processes may be reordered
  - **comm_cart**: a new cartesian grid communicator

- To get an automatic splitting into approximately equal counts in each dimension use

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
// fills in the dims array to be the best fit of arranging nnodes ranks to
// form an ndims dimensional array
```

Periodic boundary conditions

int periods[] = {true}



$\text{left}_0 = 3$

$\text{right}_3 = 0$

int periods[] = {false}

left_0 = MPI_PROC_NULL

right_3 = MPI_PROC_NULL

Creating a cartesian communicator

```
int main(int argc, char** argv)
{
  // now initialize MPI and get information about the number of processes

  MPI_Init(&argc,&argv);

  int size;
  int rank;
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  int nums[3] = {0,0,0};
  int periodic[3] = {false, false, false};

  // split the nodes automatically
  MPI_Dims_create(size, 3, nums);

  if (rank==0)
    std::cout << "We create a " << nums[0] << "x" << nums[1] << "x" << nums[2] << " arrangement.\n";

  // now everyone creates a a cartesian topology
  MPI_Comm cart_comm;
  MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true, &cart_comm);

  MPI_Comm_free(&cart_comm);
  MPI_Finalize();
 }
```

The most important one: MPI_Cart_shift

- The neighbors are obtained by

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displacement, int *source, int *dest)
// gives the ranks shifted in the dimension given by direction by a certain displacement, where the
// sign of displacement indicates the direction.
// It returns both the source rank from which the current rank can be reached by that shift
// and the dest rank that is reached from the current rank by that shift.
```

- Example in 3D:

```
int left, right, bottom, top, front, back, newrank;

MPI_Comm_rank(cart_comm,&newrank);

MPI_Cart_shift(cart_comm, 0, 1, &left, &right);
MPI_Cart_shift(cart_comm, 1, 1, &bottom, &top);
MPI_Cart_shift(cart_comm, 2, 1, &front, &back);

std::cout << "Rank " << rank << " has new rank " << newrank << " and neighbors "
          << left << ", " << right << ", "
          << top << ", " << bottom << ", "
          << front << ", " << back << std::endl;
```

Functions for cartesian topologies

- Get number of dimensions

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

- Get the cartesian topology information

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)
// retrieves information about the cartesian topology associated with a communicator.
// The arrays are allocated with maxdims dimensions. dims and periods are the numbers used
// when creating the topology. coords are the dimensions of the current rank.
```

- Get the rank of a given coordinate

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

- Get the coordinates of a given rank

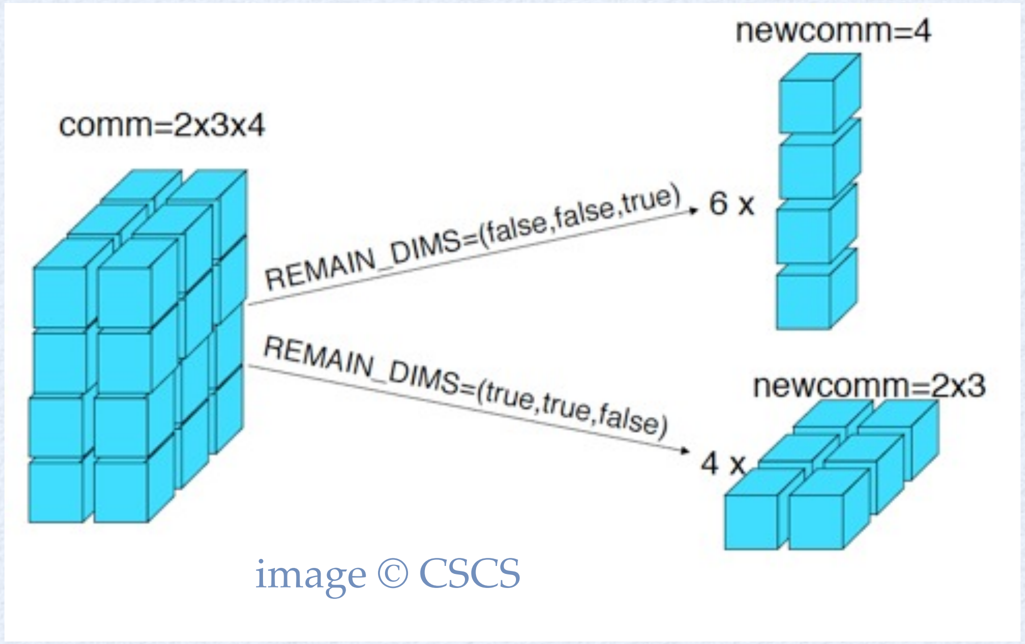
```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

Subgrids of cartesian topologies

- One can split the cartesian communicator into sub grid communicators for columns, rows, planes, ...

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *comm_new)
```

- The remain_dims array specifies whether to keep the processes along a direction joined in a group (true) or split them (false)



Graph topologies

- MPI contains another type of topology: a **graph topology**, which is not limited to a regular mesh.
  - Arbitrary number of neighbors for each rank
  - Useful for unstructured grids

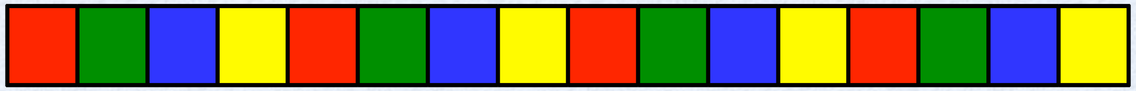
- This is essentially a graph library. In C++ we can use nicer C++ graph libraries for the same functionality.

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

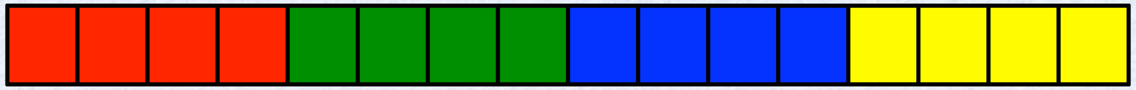
Distributed vector storage

- Cyclic distribution



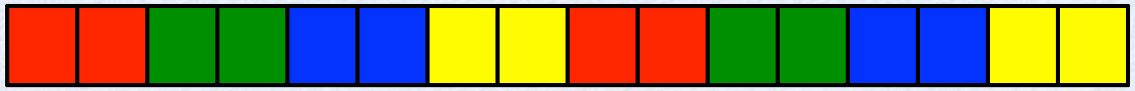
  - element *i* stored on rank $\mathrm{rank}(i) = i \bmod P$
  - local index of element *i* is $\mathrm{local}(i) = \lfloor i / P \rfloor$

- Block distribution



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

- Block-cyclic distribution



A distributed vector

```cpp
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);
}
```

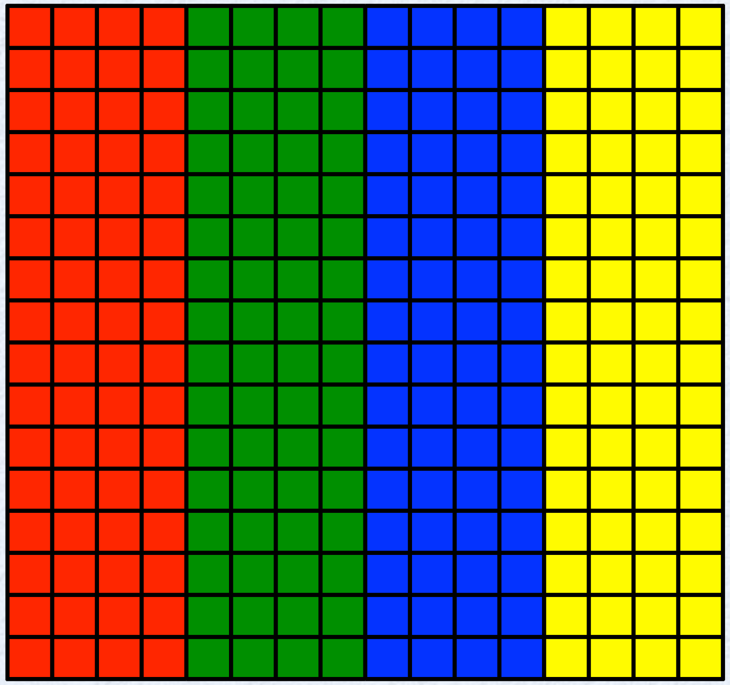
```
inline void daxpy(double alpha, dvector<double>& x, dvector<double>& y)
{
  // just scale and add the local block
  int size = x.size();
  daxpy_(size, alpha, x.data(), 1, y.data(), 1);
}
```

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

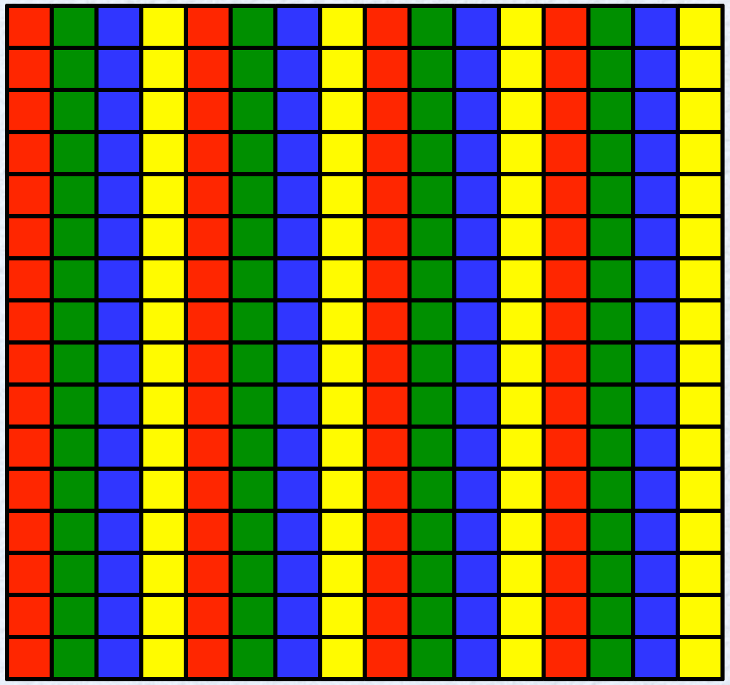
```
inline double ddot(dvector<double>& x, dvector<double>& y)
{
  assert(x.size() ==  y.size());
  int size = x.size();
  // get the local dot product
  double result = ddot_(size, x.data(), 1, y.data(), 1);

  // 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

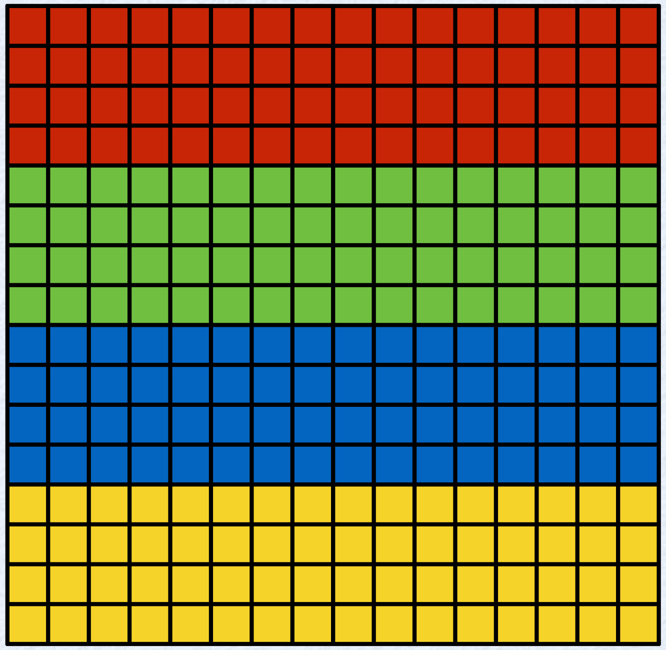


- Cyclic column distribution

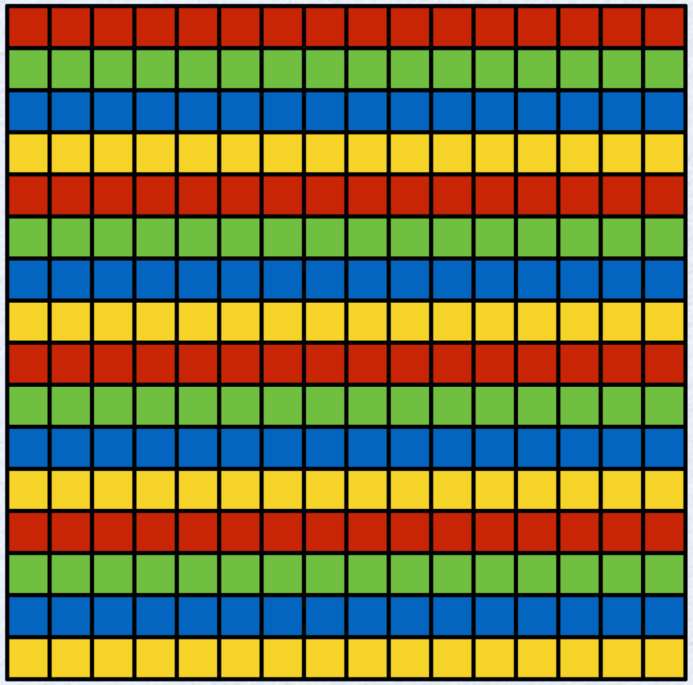


Distributed matrix storage (2)

- Block row distribution

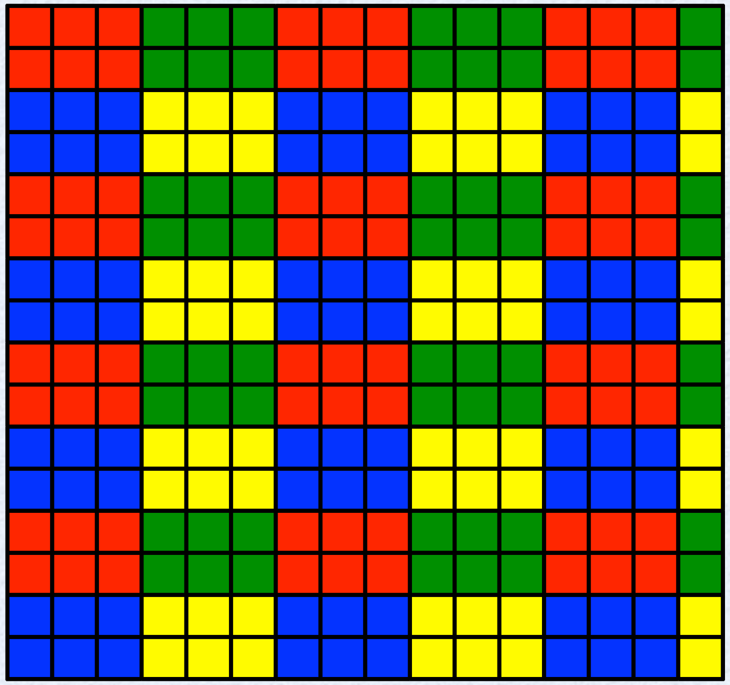


- Cyclic row distribution

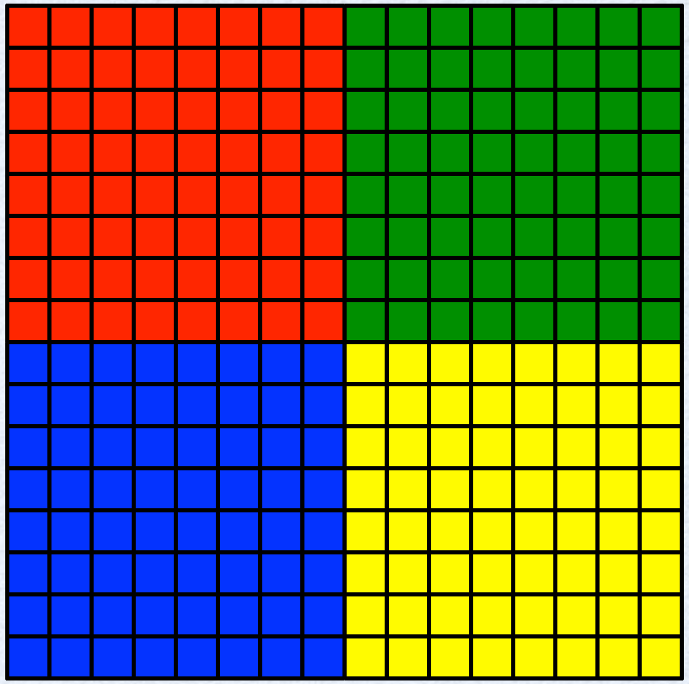


Distributed matrix storage (3)

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



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



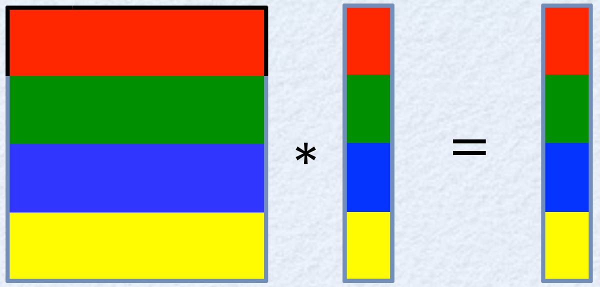
Distributed matrix operations

- 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

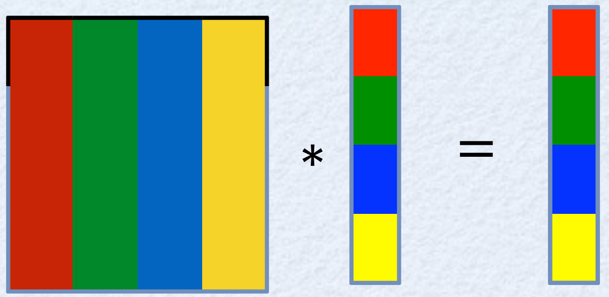
- Block-row distribution
  - Gather all parts of *x* locally
  - and then perform the local multiplications



```
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

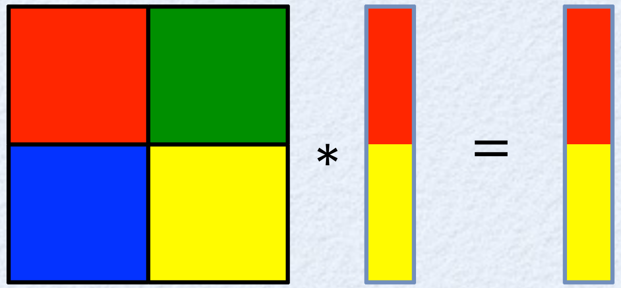


```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 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);
```

Sparse matrix-vector multiplication

- It's the same as for dense matrices but let us consider the communication requirements of the various versions
  - vector dimension N
  - sparsity a => aN nonzeros per row or column
  - number of ranks p => block size $b=N/p$

- **Block row** distributions needs to gather vector to every rank: N numbers collected to every rank

- **Block column** distribution can send a sparse result vector: $b\,a\,N$ numbers sent from every rank

- When do we send less data? Block column uses less if $aN < p$.
- But we send sparse data => overhead. Conclusion: you need to time it.

Recall the matrix multiplications

- We had three versions, neither of which scaled very well
  - Two versions were a loop over matrix-vector products (_GEMV)

```
for(unsigned int i=0; i < m; ++i)
    for(unsigned int j=0; j < n; ++j)
        for(unsigned int k=0; k < l; ++k)
            c(i,j) += a(i,k) * b(k,j);
```

matrix B multiplied from left by a row of A

```
for(unsigned int j=0; j < n; ++j)
    for(unsigned int i=0; i < m; ++i)
        for(unsigned int k=0; k < l; ++k)
            c(i,j) += a(i,k) * b(k,j);
```

matrix A multiplied from right by a column of B

  - The third was a loop over outer products of vectors (_GER)

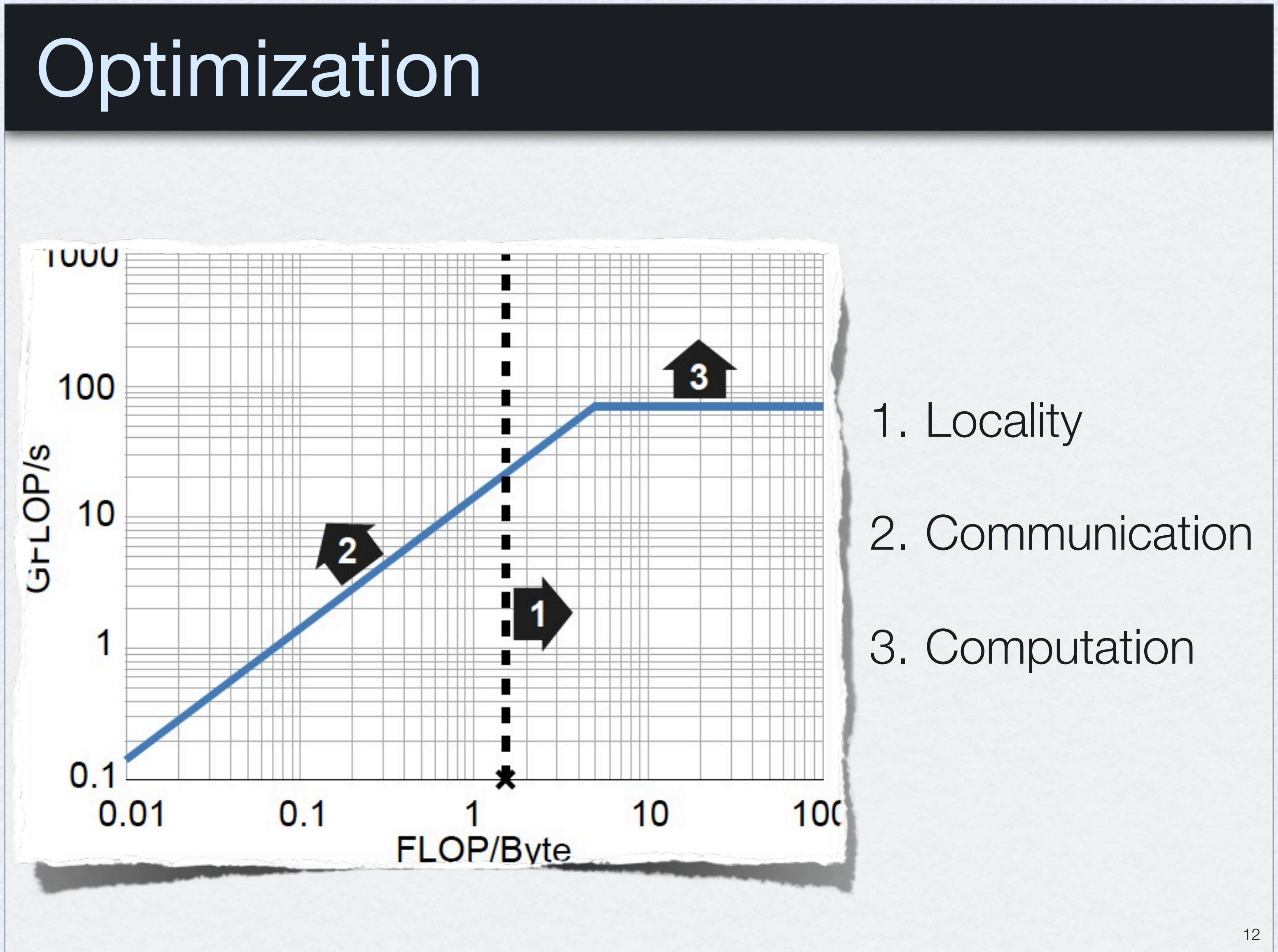
```
for(unsigned int k=0; k < l; ++k)
    for(unsigned int i=0; i < m; ++i)
        for(unsigned int j=0; j < n; ++j)
            c(i,j) += a(i,k) * b(k,j);
```

outer product of a column of A and row of B

  - These BLAS-2 operations perform *N* operations for *N* data accesses and are thus limited by memory bandwidth

Recall the roofline model

- We need to make more computations per byte that we load from memory



1. Locality
2. Communication
3. Computation

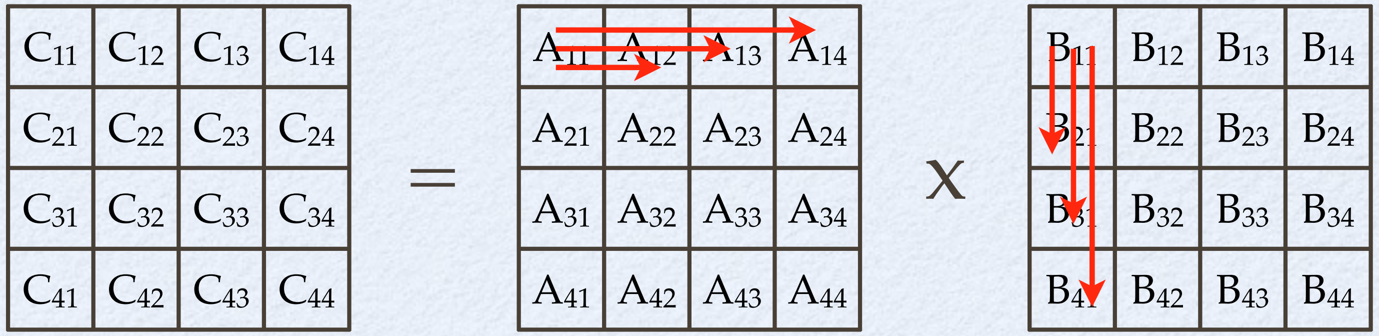
Blocking of matrix multiplies

- 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

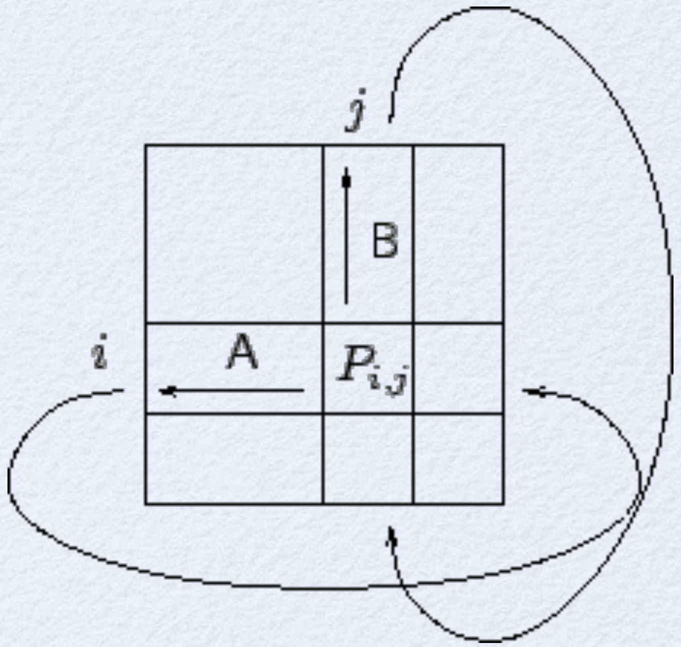
- This was not optimal yet!
  - We need memory for a whole block-row and block-column: $N \times N / \sqrt{p}$ instead of just a block of size $N \times N / p$
  - We cannot overlay computation and communication
- Solution: don't gather all data at first but shift the blocks A_{ij} and B_{ij} through 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)

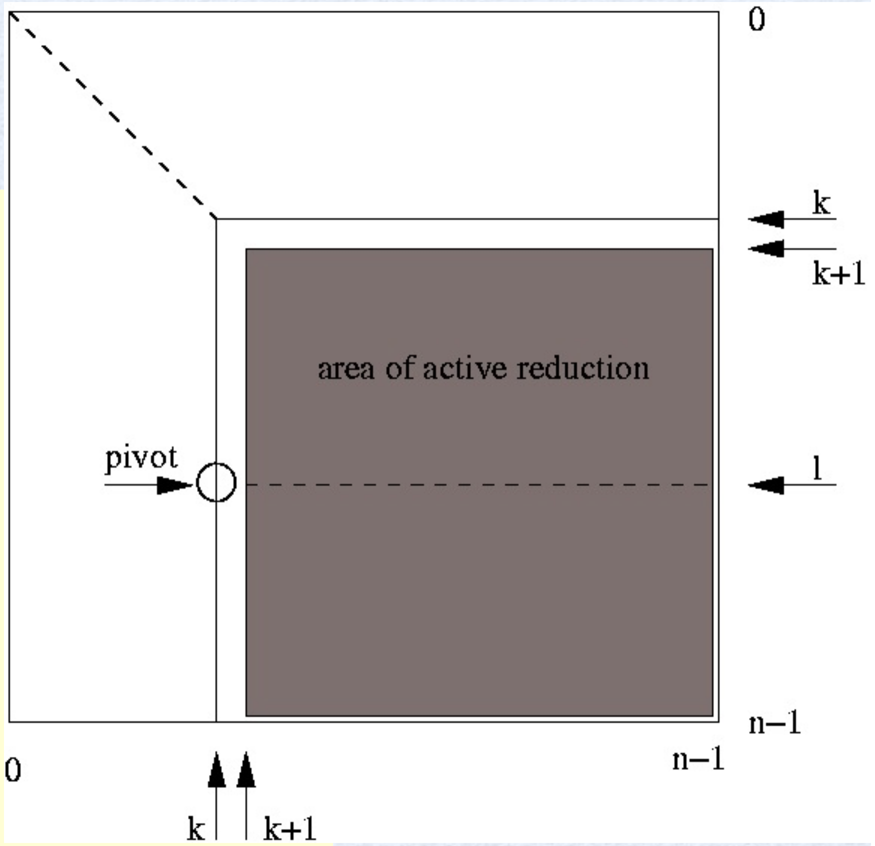
- Split the matrix into blocks like before on a *q* x *q* array
- Align the blocks so that we can start multiplying the right blocks locally:
  - move the *i*-th block row of *A* *i* blocks to the left
  - move the *j*-th block column of *B* *j* blocks up
- Repeat *q* times:
  - Multiply the local blocks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

Performance of the LU decomposition

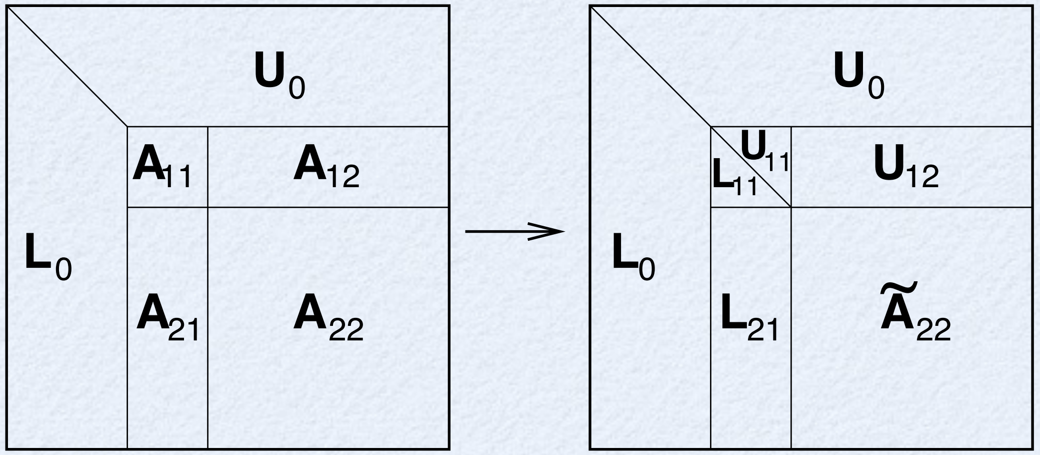
- Our previous LU code only did BLAS-1 and BLAS-2 operations
- Not enough computation per memory access



```
for(int k=0; k < a.num_rows()-1; k++){
  // 1. find the index of the largest element
  //    in column k starting at row k
  int nk = n-k;
  int l = idamax(nk,&a(k,k),one) + k;
  pivot.push_back(l); // and save it
  assert( a(l,k) != 0.0); // error if pivot is zero

  // 2. swap rows l and k, starting at column k
  dswap(nk,&a(l,k),lda,&a(k,k),lda);

  // 3. scale the column k below row k by the inverse
  // negative pivot element, to store L in the lower part
  double t = -1./a(k,k);
  int nkm1 = n-k-1;
  dscal(nkm1,t,&a(k+1,k),one);

  // 4. add the scaled k-th row to all rows in the lower right corner
  double alpha=1.;
  dger_(nkm1,nkm1,alpha,&a(k+1,k),one,&a(k,k+1),lda,&a(k+1,k+1),lda);
}
```

Blocked LU decomposition

- Do Gaussian elimination on multiple columns/rows at once



$$P\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$
$$= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}$$

1. Gaussian elimination with column pivoting

$$P\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11}$$

2. Apply *P* (row interchanges) to

$$L_0 \text{ and } \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

3. Forward substitution

$$U_{12} = L_{11}^{-1} A_{12}$$

4. Update the rest of the matrix

the computationally intensive part became DGEMM! --->
$$\tilde{A}_{22} = A_{22} - L_{21}U_{12}$$

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	—