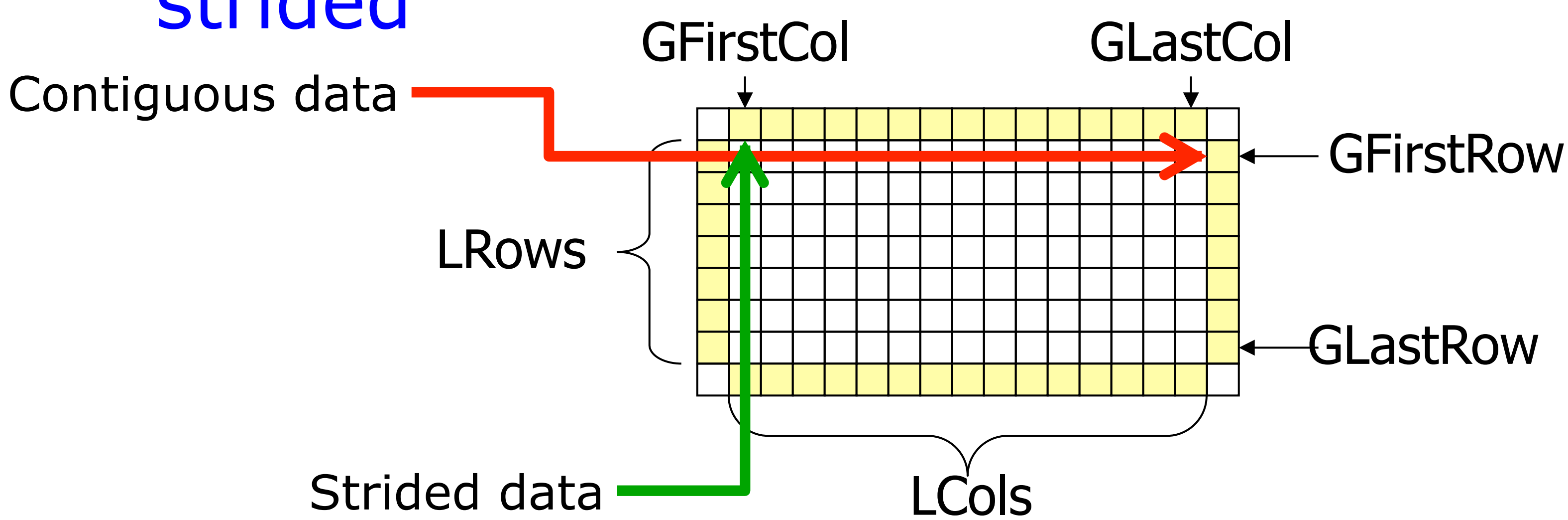
Lecture 27a: MPI Datatypes

William Gropp
www.cs.illinois.edu/~wgropp

Halo Exchange and Data Copies

- Simple analysis assume all data contiguous
  - ♦ In fact, for all but 1D decomposition, some data is contiguous, other strided



PARALLEL@ILLINOIS

Halo Exchange and Data Copies

- Common approach is to copy data to/from a temporary buffer
  - for (i=0; i<n; i++) temp[i] = a[i*nc];
- But the MPI implementation may need to copy the data from the buffer to special memory for sending and receiving
  - Depends on many details of the implementation and the interconnect design

Avoiding the Extra Copy

- MPI provides a way to efficiently and concisely define a non-contiguous pattern in memory
  - ♦ The MPI implementation may be able to avoid one memory copy by using this description
  - ♦ Note: What MPI permits, and what an implementation *may* do is not the same as what *will* happen.

MPI Datatypes

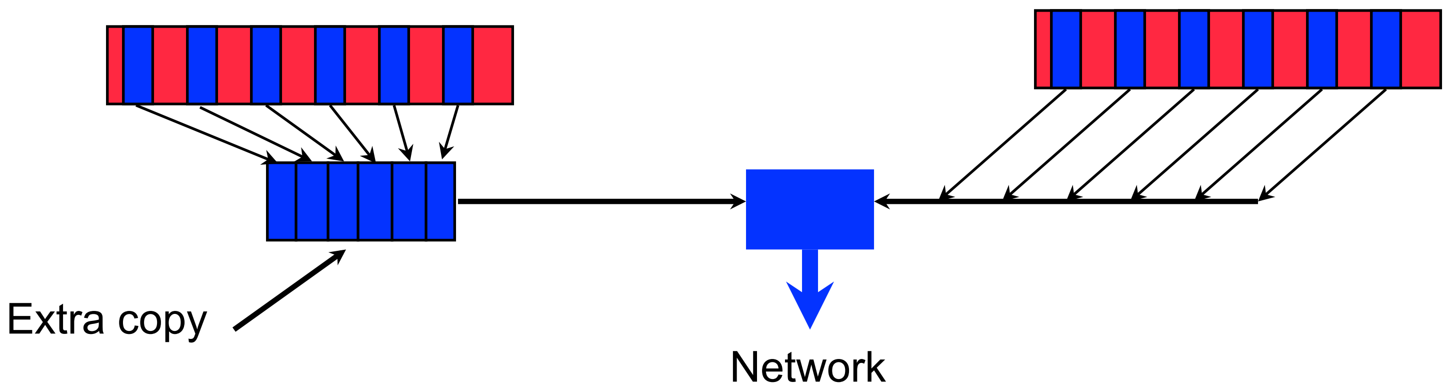
- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - can reduce memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available
- Specifying application-oriented layout of data on a file
  - can reduce system calls and physical disk I/O

Non-contiguous Datatypes

- Provided to *allow* MPI implementations to avoid copy



- MPI implementations handle with varying degrees of success
  - Strided copies of basic types likely to be best

Potential Performance Advantage in MPI Datatypes

- Handling non-contiguous data
- Assume must pack/unpack on each end
  - $cn + (s + r\,n) + cn = s + (2c + r)n$
- Can move directly
  - $s + r'\,n$
  - r' probably > r but < $(2c+r)$
- MPI implementation must copy data anyway (into network buffer or shared memory); having the datatype permits removing 2 copies

MPI Datatypes Have Been Available for Years

- Test system and software
  - ♦ System: 2.0 GHz Xeon
    - 1 Gbyte main memory
    - 512 Kbyte L2 cache
    - 1230.77 Mbyte/sec Stream benchmark result
  - ♦ Tests: MPI_Pack vs. hand coded packing
    - MPICH2 as of May 7, 2003
    - MPICH 1.2.5-1a
    - LAM 6.5.9
  - ♦ Unpack results are very similar
  - ♦ Data from 2003, EuroMPI/PVI: "Fast (and Reusable) Datatype Processing," Ross, Miller, Gropp

Performance

Test	Manual (MB/sec)	MPICH2 (%)	MPICH (%)	LAM (%)	Size (MB)	Extent (MB)
Contig	1,156.40	97.2	98.3	86.7	4	4
Struct Array	1,055.00	107.0	107.0	48.6	5.75	5.75
Vector	754.37	99.9	98.7	65.1	4	8
Struct Vector	746.04	100.0	4.9	19.0	4	8
Indexed	654.35	61.3	12.7	18.8	2	4
3D Face, XY	1,807.91	99.5	97.0	63.0	0.25	0.25
3D Face, XZ	1,244.52	99.5	97.3	79.8	0.25	63.75
3D Face, YZ	111.85	100.0	100.0	57.4	0.25	64

- Struct vector is similar to the struct example
  - Convenient way to describe N element vector
- Indexed test shows necessity of indexed node processing (though we should still do better!)
- Clear need for loop reordering in 3D YZ test
- Current implementations somewhat better but still somewhat limited; see "Micro-Applications for Communication Data Access Patterns and MPI Datatypes," Schneider, Gerstenberger, and Hoefler

Datatype Abstractions

- Standard Unix abstraction is “block of contiguous bytes” (e.g., readv, writev)
- MPI specifies datatypes recursively as
  - count of (type,offset)
    where offset may be relative or absolute

Working With MPI Datatypes

- An MPI datatype defines a *type signature*:
  - sequence of pairs: (basic type,offset)
  - An integer at offset 0, followed by another integer at offset 8, followed by a double at offset 16 is
    - (integer,0), (integer,4), (double,16)
  - Offsets need not be increasing:
    - (integer,64),(double,0)
- An MPI datatype has an extent and a size
  - *size* is the number of bytes of the datatype
  - *extent* controls how a datatype is used with the *count* field in a send and similar MPI operations
  - extent is a misleading name

What Does Extent Do?

- Consider
  MPI_Send(buf, count, datatype, …)
- What actually gets sent?
- MPI defines this as sending the same data as
  do i=0,count-1
  MPI_Send(buf(1+i*extent(datatype)),1,
  datatype,…)
  (buf is a byte type like integer*1)
- extent is used to decide where to send from (or where to receive to in MPI_Recv) for count > 1
- Normally, this is right after the last byte used for (i-1)

Changing the Extent

- MPI provides the routine MPI_Type_create_resized for changing the extent and the lower bound of a datatype
  - ♦ This doesn't change the *size*, just how MPI decides what addresses in memory to use in offseting one datatype from another.
- Usage:
  MPI_Type_create_resized(oldtype,
  lowerbound, extent, newtype)
- Except in weird cases, lowerbound should be zero.

Sending Rows of a Matrix

- From Fortran, assume you want to send a row of the matrix
  A(n,m),
  that is, A(row,j), for j=1,…, m
- A(row,j) is not adjacent in memory to A(row,j+1)
- One solution: send each element separately:

```
Do j=1,m
   Call MPI_Send( A(row,j), 1, MPI_DOUBLE_PRECSION, …)
```

- Why not? (Hint: What is the cost?)

MPI Type vector

- Create a single datatype representing elements separated by a constant distance (*stride*) in memory
  - m items, separated by a stride of n:
  - call MPI_Type_vector(m, 1, n, &
    MPI_DOUBLE_PRECISION, newtype, &
    ierr)
    call MPI_Type_commit(newtype, ierr)
  - Type_commit required before using a type in an MPI communication operation.
- Then send one instance of this type
  MPI_Send(a(row,1), 1, newtype, ….)

Test your understanding of Extent

- How do you send 2 rows of the matrix? Can you do this:
  MPI_Send(a(row,1),2,newtype,…)
- Hint: Extent(newtype) is distance from the first to last byte of the type
  - Last byte is a(row,m)
- Hint: What is the first location of A that is sent after the first row?

Sending with MPI_Vector

- Extent(newtype) is ((m-1)*n+1)*sizeof(double)
  - Last element sent is A(row,m)
- do i=0,1
  call MPI_Send(buf(1+i*extent(datatype)),1,&
  datatype,…)

  becomes
- call MPI_Send(A(row,1:m),…) (i=0)
  call MPI_Send(A(row+1,m:2m-1),…) (i=1)
- The second step is *not*
  call MPI_Send(A(row+1,1:m),…)
- **Note:** Do not use A(row,1:m) in MPI programs; it is used here as a shorthand for A(row,k) for k=1,m
  - With the MPI_F08 module, it *may* be possible to use array sections.

Solutions for Vectors

- MPI_Type_vector is for very specific uses
  - rarely makes sense to use count other than 1 with a vector type
- To send two rows, simply change the blockcount:

```
call MPI_Type_vector( m, 2, n, &
    MPI_DOUBLE_PRECISION, newtype, &
      ierr )
```

- Stride is still relative to basic type

Sending Vectors of Different Sizes

- How would you send A(i,2:m) and A(i+1,3:m) with a single MPI datatype?
  - Allow "count" to select the number of columns, as in
    call MPI_Send(A(i,2),m-1,type,…)
    call MPI_Send(A(i+1,3),m-2,type,…)
- Hint: Use an extent of n elements

Striding Type

- Create a type with an extent of a column of the array:
  - Integer (kind=MPI_ADDRESS_KIND)extent
    extent = n*8
    Call MPI_Type_create_resized(&
     MPI_DOUBLE_PRECISION, 0, extent, &
     newtype, ierr)
- Then
  MPI_Send(A(i,2),m-1,newtype,…)
  sends the elements A(i,2:m)

Test Your Understanding of Datatypes

- Write a program that sends rows of a matrix from one processor to another. Use both MPI_Type_vector and MPI_Type_create resized methods
  - Which is most efficient?
  - Which is easier to use?
- **Hard but interesting**: Write a program that sends a matrix from one processor to another. Arrange the datatypes so that the matrix is received in transposed order
  - A(i,j) on sender arrives in A(j,i) on receiver

Realities of MPI Datatypes

- Performance depends on quality of implementation
  - Not all patterns well optimized
- Example:
  - Gather for unstructured grid, 4 elements at each point. Compare:
    - Manual packing
    - MPI_Type_create_indexed_block (contiguous)
    - MPI_Type_create_indexed_block

Manual Packing

- ```
  for(int i = 0; i < slst->xlen; i++) {
  int i0 = bcsr->c * slst->isx[i];
  int i1 = bcsr->c * i;
  for(int j = 0; j < bcsr->c; j++)
      xsend[ i1 + j ] = x[ i0 + j ];
  }
  ```
  ```

MPI_Type_create_indexed_block

- ```
  MPI_Type_contiguous(bcsr->c,
        MPI_DOUBLE, &type2);
  MPI_Type_commit(&type2);
  int *sdisp = slst->isx + slst->isn[i];
  int slen = slst->isn[i+1] - slst->isn[i];
  MPI_Type_create_indexed_block(
      slen, 1, sdisp, type2, &newtype);
  MPI_Type_commit(&newtype);
  ```
- Note each block is one instance of a contiguous type of 4 doubles
  ```

MPI_Type_create_indexed_block (version 2)

- MPI_Type_create_indexed_block(
  slen, 4, sdispb4,
  MPI_DOUBLE, &newtype);
  MPI_Type_commit(&newtype);
- Sdisp array scaled by 4 from previous slide
- Note each block is 4 instances of one double

Notes On Datatypes for Gather

- Manual packing may force an extra move of data
  - MPI implementation may need to move data internally; the user pack operation is an (semantically) unnecessary move
- Both versions using MPI_Type_create_indexed_block *should* be equivalent
  - They are functionally – they describe the same data to move
  - They are not in performance (depending on the MPI implementation)
  - On Blue Waters, the 3rd form is the fastest of the three; the second is quite slow

Questions for Discussion

- Where might you use datatypes in your application?
- Why does MPI have so many different datatype constructors? Why not just use the Unix iov?
  - Hint: What is a performance model for using iovs? Compare that to an MPI vector or block-indexed type.