Data types

• So far we’ve assumed that messages are contiguous 1-dimensional arrays
• The element types have been restricted to built in types like `float`, `int`, `char`
• But users generally require a richer set of types
  – structs
  – “every k\textsuperscript{th} element of a vector”
• MPI provides a data type mechanism to enable us to work with such types
Data types in MPI

- MPI encodes the meaning of user-defined types with a special set of functions
- The type system is limited
- No support for
  - pointer-based data structures
  - callbacks

The basics of MPI data types

- **Create** an `MPI_Datatype` object

```
MPI_Datatype new_type_t
MPI_Type_vector(nblks, blkLen, stride, elt_t, &new_type_t);
```

- **Commit** the data type, allowing MPI to make some internal changes that may improve performance

```
MPI_Type_commit(&new_type_t);
```

- **Communicate** the data using the committed type

```
MPI_Send(ptr, n, new_type_t, dest, tag, comm)
```
Derived types

- MPI provides derived data types, e.g. for `struct`
- We need to describe
  - The number of elements in the `struct`
  - The type of each element
- From this information we can determine the displacement from the start of the `struct`, where each element begins
  \{(t_0,d_0), (t_1,d_1),..., (t_{n-1},d_{n-1})\}
- Members may be built-in or previously defined MPI types, but not pointers

An example of a struct

- Consider
  ```
  struct x {float a; float b; int c;}
  ```
- There are three members
  - The first member (a) is of type MPI_FLOAT
  - The second member (b) is of type MPI_FLOAT
  - The third member (c) is of type MPI_CHAR
- We can’t say what the offsets of the members are from the start of the struct because C doesn’t guarantee that members are stored contiguously
The API

\textbf{MPI\_Type\_struct}( \text{int count, int block\_lengths[ ], MPI\_Aint displacements[ ], MPI\_Datatype typelist[ ], MPI\_Datatype* new\_mpi\_t);}

Count – number of members in the struct
block\_lengths[ ] - number of entries in each member (why is this needed?)
displacements[ ] - offset of each member, not an int but an MPI provided type
typelist[ ] - type of each member

Building a struct type

\textbf{MPI\_Type\_struct}( \text{int count, int block\_lengths[ ], MPI\_Aint displacements[ ], MPI\_Datatype typelist[ ], MPI\_Datatype* new\_mpi\_t);}

Consider \texttt{struct x { float a; float b; int c;}}
Count = 3
block\_lengths[0:2] = 1
typelist[ ] = \{MPI\_FLOAT, MPI\_FLOAT, MPI\_CHAR\}
displacements[0] = 0
displacements [0:2] are computed using \texttt{offsetof( )}
Computing the displacement

- Address arithmetic involving struct members is not legal since C doesn’t guarantee that members are stored contiguously.
- We can’t compute displacements using subtraction.

```c
typedef struct { float a; float b; int n} Ts;
struct Ts S;
displacements[0] = 0
displacements[1] = &S.b - &S.a
```

Using offsetof()

- Use the offsetof() macro defined in ANSI standard C.
  [URL](http://www.lysator.liu.se/c/rat/d1.html#offsetof-4-1-5)
- MPI defines MPI_Address(), but this is unnecessary in C.

```c
typedef struct { float a; float b; int n} Ts;
struct Ts S;
displacements[0] = 0
displacements[1] = offsetof(S,b);
displacements[2] = offsetof(S,c);
```
An alternative to types

- We can copy the data into a buffer
- Packing a heterogeneous struct ourselves can lead to surprises if we are moving across machine boundaries
  ```c
  struct {int x; float y; double z;}
  ```
- MPI provides functions to support more elaborate types, and to support message packing and unpacking, but won’t discuss these

Matrix Multiplication
Matrix Multiplication

- Given two *conforming* matrices A and B, form the matrix product $A \times B$.
- Second dimension of A must equal first dimension of B:
  - A is $m \times n$
  - B is $n \times p$
- Let’s assume that the matrices are square:
  - $n \times n$
- Operation count is $O(n^3)$

Matrix multiply algorithm

```plaintext
function MM(Matrix A, Matrix B, Matrix C)
  for i := 0 to n – 1
    for j := 0 to n – 1 do
      C[i, j] = 0;
      for k := 0 to n - 1
        C[i, j] += A[i, k] * B[k, j];
      end for
    end for
end MM
```
Parallel matrix multiplication

- Assume $p$ is a perfect square
- Each processor gets an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ chunk of data
- Organize processors into rows and columns
- Process rank is an ordered pair of integers
- Assume that we have an efficient serial matrix multiply

A simple parallel algorithm

- Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element
- Thus, $A[i,k] \times B[k,j]$ is a matrix multiply in $C[i,j] += A[i,k] \times B[k,j]$
A simple parallel algorithm

- Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element.
- Thus, $A[i,k] \times B[k,j]$ is matrix multiply in $C[i,j] += A[i,k] \times B[k,j]$.
Cost

- Each processor performs $n^3/p$ multiply-adds
- But a significant amount of communication is needed to collect a row and a column of data onto each processor
- Each processor broadcasts a chunk of data of size $n^2/p$ within a row and a column of $\sqrt{p}$ processors
- Disruptive - distributes all the data in one big step
- High memory overhead
  - needs $2\sqrt{p}$ times the storage needed to hold A & B

Observation

- In the broadcast algorithm each processor multiplies two skinny matrices of size $n^2/\sqrt{p}$
- But we can form the same product by computing $\sqrt{p}$ separate matrix multiplies involving $n^2/p \times n^2/p$ matrices and accumulating partial results

$$\text{for } k := 0 \text{ to } n - 1$$

$$C[i, j] += A[i, k] \times B[k, j];$$
A more efficient algorithm

• Take advantage of the organization of the processors into rows and columns
• Move data incrementally in \( \sqrt{p} \) phases, using smaller pieces than with the broadcast approach
• Circulate each chunk of data among processors within a row or column
• In effect we are using a ring broadcast algorithm
• Buffering requirements are \( O(1) \)
A more efficient algorithm

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases, using smaller pieces than with the broadcast approach
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$

Canon’s algorithm

- Based on the above approach
- A slight reformulation to make things work out
- Consider iteration $i=1, j=2$:

**Canon’s algorithm**

- \( C[1,2] += A[1,k] \times B[k,2] \), for \( k = 0, 1, 2 \)
- We want \( A[1,0] \) and \( B[0,2] \) to reside on the same processor initially
- We then shift each row and column so that the next pair of values \( A[1,1] \) and \( B[1,2] \) line up on the same processor
- And so on with \( A[1,2] \) and \( B[2,2] \)

---

The steps of Canon’s algorithm

- This works out if we *pre-skew* the matrices
Skewing the matrices

- Canon’s algorithm requires that we *pre-skew* the matrices
- Shift each row $i$ by $i$ columns to the left using sends and receives
- Communication wraps around
- Do the same for each column

Shift and multiply

- Takes $\sqrt{p}$ steps
- Circularly shift
  - each row by 1 column to the left
  - each column by 1 row to the left
- Each processor forms the product of the two local matrices adding into the accumulated sum

\[
C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2)
\]
Cost of Canon’s algorithm – Pre skewing

\[
\text{forall } i=0 \text{ to } \sqrt{p} - 1 \\
\quad \text{CShift-left } A[i; :] \text{ by } i \quad \text{// } T = a + \beta n^2 / p
\]

\[
\text{forall } j=0 \text{ to } \sqrt{p} - 1 \\
\quad \text{Cshift-up } B[: , j] \text{ by } j \quad \text{// } T = a + \beta n^2 / p
\]

Cost of computational loop

\[
\text{for } k=0 \text{ to } \sqrt{p} - 1 \\
\quad \text{forall } i=0 \text{ to } \sqrt{p} - 1 \text{ and } j=0 \text{ to } \sqrt{p} - 1 \\
\quad \quad \text{C}[i,j] += A[i,j] \times B[i,j] \quad \text{// } T = 2 \times n^3 / p^{3/2}
\]

\[
\text{forall } i=0 \text{ to } \sqrt{p} - 1 \\
\quad \text{CShift-leftA}[i; :] \text{ by } 1 \quad \text{// } T = a + \beta n^2 / p
\]

\[
\text{forall } j=0 \text{ to } \sqrt{p} - 1 \\
\quad \text{Cshift-up } B[: , j] \text{ by } 1 \quad \text{// } T = a + \beta n^2 / p
\]

end for
Cost of Canon’s algorithm

- \[ T_p = \frac{2n^3}{p} + 4\sqrt{p}a + 4\beta n^2/\sqrt{p} \]
- \[ E_p = \frac{2n^3}{pT_p} = \frac{1}{1 + 2\alpha (\sqrt{p/n})^3 + 2\beta \sqrt{p/n}} \]
  \[ = \frac{1}{1 + O(\sqrt{p/n})} \]

- Drawbacks
  - Extra added storage for shifting the matrices
  - Awkward if \( P \) is not a perfect square,
  - \( A \) and \( B \) are not square, and not evenly divisible by \( \sqrt{P} \)

- Alternative: the SUMMA algorithm
  www.netlib.org/lapack/lawns/lawn96.ps

MPI Communicators

- MPI Communicators provide a way of hiding internal behavior of a library written using MPI
- If we call a library routine, we don’t want the message passing activity in the library to interfere with our program
- A communicator specifies a name space called a Communication Domain
- Messages remain within their communication domain
Implementing Cannon’s algorithm

- Cannon’s algorithm provides a good motivation for using MPI communication domains
- Communication domains simplify the code, by specifying subsets of processes that may communicate
- We may structure the sets in any way we like
- Each processor may be a member of more than one communication domain
- We will define new sets of communicators that naturally reflect the structure of communication along rows and columns

Splitting communicators

- We can create a set of communicators, one for each row and column of the geometry
- Each process computes a key based on its rank
- We then group processes together that have the same key
- Each process has a rank relative to the new communicator
- If a process is a member of several communicators, it will have a rank within each one
Splitting communicators for Cannon’s algorithm

- In Cannon’s algorithm, each processes needs to communicate with process within its row and column
- Let’s create a communicator for each row and one for each column
- Consider a grouping of processors by row
  \[ \text{key} = \text{myid div } \sqrt{P} \]
- Thus, if P=9, then
  - Processes 0, 1, 2 are in one communicator because they share the same value of key (0)
  - Processes 3, 4, 5 are in another (1)
  - Processes 6, 7, 8 are in a third (2)

MPI support

- **MPI_Comm_split( )** is the workhorse
  
  ```
  MPI_Comm_split(MPI_Comm comm, int splitKey, int rankKey, MPI_Comm* newComm);
  ```

- A collective call
- Each process receives a new communicator, which it shares in common with other processes having the same key value
Comm_split

\[
\text{MPI\_Comm\_split}(\text{MPI\_Comm \_comm},
\text{int \_splitKey},
\text{int \_rankKey},
\text{MPI\_Comm \_newComm});
\]

- Each process receives a unique rank within its respective communicator according to the value of \text{rankKey}
- The relative values of the ranks follows the ordering of the rankKeys across the processes
- I.e. if A give a rank key of 1, and B a rank key of 10, then A’s rank < B’s rank

More on Comm_split

\[
\text{MPI\_Comm\_split}(\text{MPI\_Comm \_comm},
\text{int \_splitKey},
\text{int \_rankKey},
\text{MPI\_Comm \_newComm});
\]

- Ranks are assigned arbitrarily among processes sharing the same \text{rankKey} value
- It is also possible to exclude a process from a communicator, by passing the constant \text{MPI\_UNDEFINED} as the \text{splitKey}
- A special \text{MPI\_COMM\_NULL} communicator will be returned
Splitting into rows

**MPI_Comm** rowComm;

int myRow = myid / √P;

**MPI_Comm_split** (MPI_COMM_WORLD, myRow, myid, &rowComm);

A ring shift

**MPI_Comm_rank**(rowComm,&myidRing);

**MPI_Comm_size**(rowComm,&nodesRing);

int l = myrow, X = …, XR;

int next = (myidRng + 1 ) % nodesRing;

**MPI_Send**(&X,1,MPI_INT,next,0,rowComm);

**MPI_Recv**(&XR,1,MPI_INT,MPI_ANY_SOURCE,0,rowComm,&status);
Other kinds of communication domains

• Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar
• But we can accomplish a good deal of what we want using the splitting method

The Code

• Cartesian grids permit us to work in different coordinate systems such that the rank is no longer a scalar
• But we can accomplish a good deal of what we want using the splitting method
• We use another routine MPI_Sendrecv_replace() to simplify the coding
• Sends then receives a message using one buffer
• Code listing on separate handout
MPI_Sendrecv_replace()

- Sends then receives a message using one buffer

```c
int MPI_Sendrecv_replace( void *buf,
   int count,   MPI_Datatype datatype,
   int dest,    int sendtag,
   int source,  int recvtag,
   MPI_Comm comm
MPI_Status *status )
```