Announcements

- Homework #5 due today at 11:59:59 PM
- Assignment #6
  - Implement sample sort
  - Will be posted on Wednesday

New print facility

- Installed in ~/.public/lib/PPF
- There is a new version of the arch file called
  
  arch.valkyrie_ppf

- Each module that uses the facility must
  
  #include "ptools_ppf.h"

- See the program test_print.c for usage examples

Summary of capabilities

- Compact output list sets of nodes with common output
  
  PPF_Print( MPI_COMM_WORLD, "Hello world" );

- Results in 0-3: Hello world

- Special %N specifier enables positioning of node information
  
  PPF_Print( MPI_COMM_WORLD, "Message from %N \n" );

- Results in Message from 0-3

- Can also have lists of nodes
  
  [0,2] Message from even numbered nodes
  [1,3] Message from odd numbered nodes
MPI 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 kth element of an array”
• 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
• But the type system is limited
• There is 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 support 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
• Members may be built-in or previously defined MPI types, but not pointers
Derived types

• Consider the struct
  
  ```c
  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.

Describing types

• A derived data type is a sequence of pairs
  
  \[(t_0, d_0), (t_1, d_1), \ldots, (t_n, d_n)\]

• Each \(t_i\) is an MPI data type

• Each \(d_i\) is displacement

• The extent of a datatype is defined to be the length from the first to last byte of the datatype, including any rounding upward needed to satisfy installation-dependent alignment requirements (compiler options can have an effect).

The API

```c
MPI_Type_struct(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

```
MPI_Type_struct(int count,
                int block_lengths[],
                MPI_Aint displacements[],
                MPI_Datatype typelist[],
                MPI_Datatype* new_mpi_t);
```

Count = 3
block_lengths[0:2] = 1
typelist[] = {MPI_FLOAT, MPI_FLOAT, MPI_CHAR}
displacements[0] = 0
displacements[0:2] are computed using

```
MPI_Address()
```

Computing displacements

- Address arithmetic involving struct members is not legal.
- Thus, we can’t compute displacements using subtraction.

```
typedef struct {float a; float b; int n} Ts;
struct Ts S;
displacements[0] = 0
displacements[1] = &S.b - &S.a; // 4
```

An alternative to types: copying

- Copy the data into a temporary buffer.
- `struct {int x; float y; double z};`
- MPI provides functions to support packing, but won’t discuss these.
- But a heterogeneous struct can lead to surprises if we are moving across machine boundaries.
Structural equivalence

- The recipient has flexibility to store incoming values according to a locally defined rule
- Process A can send a block of data in row major order to Process B, which can receive the data in column major order into a local data structure
- The only constraint is that the number of sent and received elements are the same

Specifying a Vector type

```cpp
MPI_Type_vector(
    int count=N,
    int blockLen=1,
    int stride=N,
    MPI_Datatype MPI_INT,
    MPI_Datatype &vec_t);
```

Vector type

- The addresses in a column of a 2D array are not contiguous
- When else do we need to deal with the pattern?

Copying a column

```cpp
if (myid == 0)
    MPI_Send(A, 1, vec_t, 1, 0, MPI_COMM_WORLD);
else
    MPI_Recv(&A[0][N-1], 1, vec_t, 0, 0, MPI_COMM_WORLD, &status);
```
Receiving data contiguously

• Takes advantage of structural equivalence

```
if (myid == 0)
   MPI_Send(A, 1, vec_t, 1, 0, MPI_COMM_WORLD);
else
   MPI_Recv(A, N, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
```

A more elaborate example

Transmit a 3 x 2 sub-block of an N x N array

```
MPI_Type_vector(
    int count = 2,
    int blockLen = 3,
    int stride = N,
    MPI_Datatype MPI_DOUBLE,
    MPI_Datatype &vec_t);
```

Another example

• What does this sequence of calls perform?

```
MPI_Type_vector(n,                     // blocks
    n,                     // blockLen
    n*n,                  // stride
    MPI_INT,
    &horiz_t);
...
MPI_Send(buff, 1,     horiz_t, dest, tag, comm)
MPI_Recv(buff,  n*n, MPI_INT, src, t ag,
          comm, &status);
```
Another example

- Copies a horizontal plane of a 3D array into a 2D buffer

```c
MPI_Type_vector(n, // blocks
    n, // blockLen
    n*n, // stride
    MPI_INT,
    &horiz_t);
...
MPI_Send(buff, 1, horiz_t, dest, tag, comm)
MPI_Recv(buff, n*n, MPI_INT, src, tag, comm, &status);
```

A word problem

- Reminiscent of block cyclic distributions
- Let’s transmit N blocks of 2 elements, that skip 4 elements between each block

```c
MPI_Type_vector(N, // blocks
    2, // blockLen
    6, // stride
    MPI_INT,
    &horiz_t);
```

A word problem

- Reminiscent of block cyclic distributions
- Let’s collect a block of 2 elements, that skips 6 elements between each block

```c
MPI_Type_vector(N, // blocks
    ?, // blockLen
    ?, // stride
    MPI_INT,
    &horiz_t);
```