Introduction to Parallel and Distributed Processing

Custom Dataypes and I/O

Jaroslaw 'Jaric' Zola

http://www.jzola.org/

Partitioning Communication

- Often, it is advantageous to organize processors into logical groups, e.g. processors in the same row/column
- Each group can act "independently" of any other group (collectives)
- MPI provides mechanism to create new communicator from the existing one

Creating Communicator

```cpp
// ex01.cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int rank, size;

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

  int col = rank % 4;
  int row = rank >> 2;

  MPI_Comm col_comm;
  MPI_Comm_split(MPI_COMM_WORLD, col, rank, &col_comm);

  int nrank;
  MPI_Comm_rank(col_comm, &nrank);

  std::cout << rank << " " << row << " " << col << " "
            << nrank << std::endl;

  MPI_Comm_free(&col_comm);

  return MPI_Finalize();
} // main
```

Custom Datatypes

- Same but different: `sizeof(A)=12`, `sizeof(B)=8`

```
struct A {
  char c0;
  short int si0;
  int i0;
  char c1;
};
```

```
struct B {
  char c0;
  char c1;
  short int si0;
  int i0;
};
```

MPI Datatypes

- Abstractly, type represented via type map:
  $Type = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$
- Examples:
  - `MPI_DOUBLE`
    $\{(double, 0)\}$
  - `struct X { double d; char c; };`
    $\{(double, 0), (char, 8)\}$

Easiest Type Creator

- Replicate $count$ times old type
- Example: $old = \{(int, 0), (double, 8)\}$, with $count = 2$
  $new = \{(int, 0), (double, 8), (int, 16), (double, 24)\}$

```
MPI_Datatype MPI_INT2;
MPI_Type_contiguous(2, MPI_INT, &MPI_INT2);
MPI_Type_commit(&MPI_INT2);

if (rank == 0) {
  int tab[256];
  MPI_Send(tab, 1, MPI_INT2, 1, 11, MPI_COMM_WORLD);
} else if (rank == 1) {
  int buf[2];
  MPI_Status stat;
  MPI_Recv(buf, 1, MPI_INT2, 0, 11, MPI_COMM_WORLD, &stat);
}

MPI_Type_free(&MPI_INT2);
```

Brute-force Approach

- Consider each type as a stream of bytes:

```
struct X {
  char c;
  double d;
};

MPI_Datatype MPI_X;
MPI_Type_contiguous(sizeof(X), MPI_BYTE, &MPI_X);
MPI_Type_commit(&MPI_X);
// ...
MPI_Type_free(&MPI_X);
```

Data Blocks

- Stack together blocks of data with some stride
- Example, type describing column of a matrix:

```
int n = 2;
std::vector<int> A{0, 1, 2, 3};

MPI_Datatype MPI_COL;
MPI_Type_vector(n, 1, n, MPI_INT, &MPI_COL);
MPI_Type_commit(&MPI_COL);

if (rank == 0) {
  MPI_Send(A.data(), 1, MPI_COL, 1, 11, MPI_COMM_WORLD);
} else if (rank == 1) {
  std::vector<int> b(n);
  MPI_Status stat;
  MPI_Recv(b.data(), n, MPI_INT, 0, 11, MPI_COMM_WORLD, &stat);
}

MPI_Type_free(&MPI_COL);
```

Parallel I/O

- After computing and communication, forgotten element
- What if your input data consists of 10-100 TB?
- If sequential I/O becomes serious performance bottleneck (Amdahl's law)

I/O – Nasty Business

- Lack of really good hardware/software solutions
- Hard to scale
- Can go wrong in many places: client code, OS level, metadata server, storage node, network

MPI I/O

- Provides middle-level abstraction for I/O on top of the FS
- Hides low-level details
- But still requires seeks and data views

Basic MPI I/O Write

```
// ex05.cpp
#include <vector>
#include <mpi.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_File fh;
  MPI_File_open(MPI_COMM_WORLD, argv[1], MPI_MODE_CREATE|MPI_MODE_WRONLY,
                MPI_INFO_NULL, &fh);

  MPI_Status stat;
  std::vector<int> buf(8, rank);

  MPI_File_seek(fh, rank * buf.size() * sizeof(int), MPI_SEEK_SET);
  MPI_File_write_all(fh, buf.data(), buf.size(), MPI_INT, &stat);

  MPI_File_close(&fh);

  return MPI_Finalize();
} // main
```

Basic MPI I/O Read

```
// ex06.cpp
#include <vector>
#include <mpi.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

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

  MPI_File fh;
  MPI_File_open(MPI_COMM_WORLD, argv[1], MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

  MPI_Offset fs;
  MPI_File_get_size(fh, &fs);

  MPI_Status stat;

  int bsz = fs / size;
  std::vector<int> buf(bsz / sizeof(int));

  MPI_File_seek(fh, rank * bsz, MPI_SEEK_SET);
  MPI_File_read_all(fh, buf.data(), buf.size(), MPI_INT, &stat);

  MPI_File_close(&fh);

  return MPI_Finalize();
} // main
```

More Complex Reading

```cpp
// ex07.cpp
#include <vector>
#include <mpi.h>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

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

  MPI_Datatype MPI_COL;
  MPI_Type_vector(8, 1, 8, MPI_INT, &MPI_COL);
  MPI_Type_commit(&MPI_COL);

  MPI_File fh;
  MPI_File_open(MPI_COMM_WORLD, argv[1], MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

  MPI_Status stat;
  std::vector<int> buf(8);

  MPI_File_set_view(fh, rank * sizeof(int), MPI_INT, MPI_COL, "native", MPI_INFO_NULL);
  MPI_File_read_all(fh, buf.data(), 8, MPI_INT, &stat);

  MPI_Type_free(&MPI_COL);
  MPI_File_close(&fh);

  return MPI_Finalize();
} // main
```

Reading "Plain" Text

- Plain text is horrible for parallel computing
- No fixed-size separation between records, etc.
- Possible approaches:
  - Preprocess to binary format (e.g. index + data)
  - Preprocessor can be in OpenMP
  - Or:
  - Split file evenly at bytes level, read in parallel
  - Parse on each processor, communicate truncated lines

For Fun

- Implement 2D matrix-vector product with parallel I/O and custom communicators.