High Performance Computing for Science and Engineering II

6.5.2019 - Lecture 10: Advanced MPI Topics

Lecturer: Dr. Sergio Martin

Based on previous work by: P. Hadjidoukas and M. Troyer
Today's Class

Advanced MPI: Datatypes
  - Vector & Struct Types

Advanced MPI: Communicators
  - Groups & Cartesian Geometries

Putting it all together:
  - Cannon's Algorithm
MPI Custom Datatypes
MPI Review

Must Read: https://computing.llnl.gov/tutorials/mpi/

Point-to-Point Communication
Message Exchange

Collective Communication

Path of a message buffered at the receiving process
Example: Numerical Integration

\[ \int_a^b f(x) \, dx \approx \sum_{k=1}^{m} f(x^{[k]}) \Delta x \]

- **Riemann Sums**
- **Trapezoidal Rule**
- **Simpson’s Rule**

Source: https://www.value-at-risk.net/numerical-integration-in-one-dimension/
Increasing the number of rectangles

\[ y = x^2 \]

\[ \text{Area} \]

Number of rectangles

Source: By 09glasgow09 - Own work, CC BY-SA 3.0

Source: Dcoetzee - Own work. This diagram was created with Mathematica, CC0
int main(int argc, char** argv) {
    double a = read(); // lower bound of integration
    double b = read(); // upper bound of integration
    int n = read(); // number of subintervals

    double area = 0.0;
    double dx = (b-a)/n;

    for (int i = 0; i < n; i++) // Integration
    {
        double x = a + i*dx;
        area += f(x)*dx;
    }

    printf("Area: %f\n", area);
}
Riemann Sums Code (MPI)

```c
int main(int argc, char** argv) {
    MPI_Init(&argc,&argv);
    int rankCount, myRank;

    MPI_Comm_size(MPI_COMM_WORLD,&rankCount);
    MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

    double a, b;
    int n;

    if (myRank == 0) {
        a = read();  // lower bound of integration
        b = read();  // upper bound of integration
        n = read();  // number of subintervals
    }

    // Need to distribute these values to other ranks
    // We will see how later.
```
int myStart = myRank * (n/rankCount);
int myEnd = (myRank+1) * (n/rankCount);

double myArea = 0.0;
double dx = (b-a)/n; // Delta X = b-a / number of intervals

for (int i = start; i < end; i++) // Integration
{
    double x = a + i*dx;
    myArea += f(x)*dx;
}

double TotalArea = 0.0;
    MPI_Reduce(&area, &TotalArea, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) printf("Area: %f\n", TotalArea);
Distributing Parameters (Broadcast)

```c
if (myRank == 0)
{
    a = read();  // lower bound of integration
    b = read();  // upper bound of integration
    n = read();  // number of subintervals
}

// Use broadcasts to distribute parameters

MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT,     0, MPI_COMM_WORLD);
```

Inefficient: uses 3 broadcasts.
Latency-intensive.
// Defining a struct for the parameters
struct parms {
    double a; // lower bound of integration
    double b; // upper bound of integration
    int n;    // number of subintervals
};

parms p;
if (myRank == 0) {
    p.a = read(); // lower bound of integration
    p.b = read(); // upper bound of integration
    p.n = read(); // number of subintervals
}

// Broadcast the parameters as array of bytes
// Not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
Alignment and Padding

It is not safe to assume how the compiler/architecture lay data in memory. It may be wrong due to unforeseen alignment/padding and endianness. Consider the following data layouts for the structure above:

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

`MPI_Aint` offsets[3] = 

{0, 
sizeof(short),
sizeof(short)+sizeof(double)};

E.g., Intel x86
32-bit integer

```
0A0B0C0D
```

<table>
<thead>
<tr>
<th>Memory</th>
<th>Little-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 0D</td>
<td></td>
</tr>
<tr>
<td>a+1: OC</td>
<td></td>
</tr>
<tr>
<td>a+2: OB</td>
<td></td>
</tr>
<tr>
<td>a+3: 0A</td>
<td></td>
</tr>
</tbody>
</table>

E.g., PS3
32-bit integer

```
0A0B0C0D
```

<table>
<thead>
<tr>
<th>Memory</th>
<th>Big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 0A</td>
<td></td>
</tr>
<tr>
<td>a+1: 0B</td>
<td></td>
</tr>
<tr>
<td>a+2: OC</td>
<td></td>
</tr>
<tr>
<td>a+3: 0D</td>
<td></td>
</tr>
</tbody>
</table>

Padding to ensure alignment of the next variable
MPI_Pack / MPI_Unpack

Functions for packing non-contiguous data into contiguous buffers.

```c
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
              void *outbuf, int outcount, int *position, MPI_Comm comm)
```

**Rationale:** packs the data given as input into the `outbuf` buffer starting at a given position. `Outcount` indicates the buffer size and `position` gets updated to point to the first free byte after packing in the data.

```c
int MPI_Unpack(void *inbuf, int insize, int *position,
                void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

**Rationale:** unpacks data from the buffer starting at given `position` into the buffer `outbuf`. `position` is updated to point to the location after the last byte read.

```c
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

**Rationale:** returns in `size` an upper bound for the number of bytes needed to pack `incount` values of type `datatype`. This can be used to determine the required buffer size.
Distributing Parameters (Pack/Unpack)

```c
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 (myRank == 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(&n, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
}

MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

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);
```
Distributing Parameters Recap

We have analyzed 3 alternatives:
- Multiple Broadcasts: requires excessive latency.
- Bytewise struct passing: non-portable.
- MPI_Pack/Unpack: works, but requires extra-copying.

What we need:
- An efficient way to communicate structures or non-contiguous data.

Solution:
- **MPI datatypes**: Describe your data layout to MPI and uses it as an MPI datatype.
General Approach: `MPI_Type_create_struct`, describes any data layout.

```c
int MPI_Type_create_struct(int count, int blocklengths[], MPI_Aint offsets[], MPI_Datatype types[], MPI_Datatype *newtype)

Rationale: Builds a new MPI data type for a general data structure given by types, counts (blocklengths) and their offsets relative to the start of the data structure
```

```c
int MPI_Type_commit(MPI_Datatype *datatype)

Rationale: Commits the data type: finished building it. It can now be used.
```

```c
int MPI_Type_free(MPI_Datatype *datatype)

Rationale: Frees the data type, releasing any allocated memory.
```
/\begin{verbatim}
// Defining a struct for the parameters
struct params {
  double a; // lower bound of integration
  double b; // upper bound of integration
  int n;    // number of subintervals
};

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);
\end{verbatim}
Distributing Parameters (MPI Custom Type)

```c
parms p;
if (myRank == 0)
{
    p.a = read(); // lower bound of integration
    p.b = read(); // upper bound of integration
    p.n = read(); // number of subintervals
}

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

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

✓ Single Broadcast
✓ No Data Duplication
('? Portable
Safe usage of MPI_Type_create_struct

Solution: int MPI_Get_address(void *location, MPI_Aint *address)

Rationale: Converts a pointer to the correct (MPI Internal) offset representation

```c
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
```

```c
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);
```

✓ Portable
Structured Grid Stencil Solver

- Iteratively approaches a solution.

2D Grid

Node

Traditional Decomposition
1 Process (Rank) per Core.

Ranks Exchange Halo (Boundary) Cells
Grids: How do we exchange boundaries?

Contiguous vs. Strided Data

1D Grid

Contiguous Data = No Problem
Use native types: e.g., MPI_Double

2D Grid

Y-Axis Boundaries: Contiguous Data
X-Axis Boundaries: Strided Data
Non-Contiguous Data Strides

A common problem for communication in multi-dimensional grids. (Worse for 3D!)

Source: Pekka Paalanen
Strided/Vector Datatypes

Boundary elements in a multidimensional array (or matrix) can be described as strided vectors.

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Rationale: build an MPI datatype for a contiguous array.

Boundary elements in a multidimensional array (or matrix) can be described as strided vectors.

int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Rationale: Build an MPI datatype for a vector array of blocklength contiguous entries that are spaced at a given stride. This specifies the distance between blocks.

int MPI_Type_create_hvector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Rationale: MPI_Type_vector but now the stride is given in bytess.

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Rationale: build an MPI datatype for a contiguous array.
Example: Matrix

Let's create datatypes for the elements of a 4x4 matrix:

\[
\text{hpc12::matrix<double,hpc12::column_major>} \quad a(4,4);
\]

\[
\begin{align*}
\text{MPI Datatype} & \quad \text{rowType, colType;} \\
\text{MPI Type contiguous} & \quad (4, \text{MPI DOUBLE}, \&\text{colType}); \\
\text{MPI Type vector} & \quad (4, 1, 4, \text{MPI DOUBLE}, \&\text{rowType}); \\
\text{MPI Type commit} & \quad (\&\text{rowType}); \\
\text{MPI Type commit} & \quad (\&\text{colType}); \\
\end{align*}
\]

// Perform Work and Communication

\[
\begin{align*}
\text{MPI Type free} & \quad (\&\text{row}); \\
\text{MPI Type free} & \quad (\&\text{col});
\end{align*}
\]
MPI Communicators
MPI Communicator Groups

Suppose we need to run an application kernel on a 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.

Source: Blaise Barney
double parallel_riemann(MPI_Comm comm, double a, double b, int n) 
{
    int rankCount, myRank;
    MPI_Comm_size(comm, &rankCount);
    MPI_Comm_rank(comm, &myRank);

    // Do the work
    double area = 0.0;
    MPI_Reduce(&area, &myArea, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

    if (myRank == 0) return area;
}
Riemann Sums with Custom Communicators

Do 3 Riemann Sums at once, each with a third of the ranks.

```c
int main(int argc, char** argv)
{
    MPI_Init(&argc,&argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    parms p[3]; initialize(p, 3); // We want to do three integrals at once

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

    double result = parallel_riemann(comm, p[which]); // Integrate my group

    int grouprank;
    MPI_Comm_rank(comm, &grouprank);
    if (grouprank == 0) printResults(); // Only the master for each group prints

    MPI_Comm_free(&comm); // free the type and the new communicator
    MPI_Finalize();
}
```
Creating and Destroying Communicators

```c
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 if 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.
```
Connectivity of MPI Ranks

Determining the ranks of my neighbors:
- 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 Types:
- **Cartesian topology**: each process is “connected” to its neighbors in a virtual grid.
- **Graph Topology**: any arbitrary connection graph.

Topologies are just a graph library for MPI Ranks.
Cartesian Topology

A 2D Cartesian Topology:

Boundaries can be cyclic (periodic):

- left₀ = 3
- right₃ = 0

left₀ = MPI_PROC_NULL
right₃ = MPI_PROC_NULL
Creating an MPI Cartesian Topology (I)

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

- **comm_old**: The source communicator.
- **ndims**: Number of dimensions
- **dims**: Integer array specifying the number of processes in each dimension
- **periods**: Integer array of boolean values indicating whether the grid is periodic in that dimension
- **reorder**: Boolean flag indicating whether the processes may be reordered
- **comm_cart**: A new cartesian grid communicator.

This is a Collective Operation.

To help creating a grid with a fair node distribution in each dimension, use:

```c
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

<table>
<thead>
<tr>
<th>dims before call</th>
<th>function call</th>
<th>dims on return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(6, 2, dims)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(7, 2, dims)</td>
<td>(7,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(6, 3, dims)</td>
<td>(2,3,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(7, 3, dims)</td>
<td>erroneous call</td>
</tr>
</tbody>
</table>
```
Creating an MPI Cartesian Topology (II)

Example: 3D Cartesian Grid

```c
int main(int argc, char** argv)
{
    MPI_Init(&argc,&argv);

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

    MPI_Dims_create(size, 3, nums); // split the nodes automatically
    printf("Grid: (%d, %d, %d)\n", nums[0], nums[1], nums[2]);

    MPI_Comm cart_comm; // now everyone creates a a cartesian topology
    MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true, &cart_comm);
    MPI_Comm_free(&cart_comm);
    MPI_Finalize();
}
```
Let use MPI topologies to find the ranks nearest neighbors.

```c
int MPI_Cart_shift(MPI_Comm comm, int dir, int disp, int *source, int *dest)
```

**Rationale:** Gives the ranks shifted in the dimension given by direction by a certain displacement, where the sign of displacement indicates the direction.

```c
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);

printf("MyRank: %d -> NewRank: %d\n", rank, newrank);
printf("Left:  %d, Right:  %d\n", left,  right);
printf("Top:   %d, Bottom: %d\n", top,   bottom);
printf("Front: %d, Back:   %d\n", front, back);
```
Useful Cartesian Functions

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
Rationale: Get number of dimensions.

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

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
Rationale: Get the rank of a given coordinate.

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
Rationale: Get the coordinates of a given rank.
Putting it all together: Cannon's Algorithm
Cannon's Algorithm

Distributed Square Matrix-Matrix multiplication algorithm -> Highly-Scalable

Subdivides Matrices into n Submatrices

Square Matrix A

\[
\begin{array}{ccc}
A00 & A01 & A02 \\
A10 & A11 & A12 \\
A20 & A21 & A22
\end{array}
\]

Square Matrix B

\[
\begin{array}{ccc}
B00 & B01 & B02 \\
B10 & B11 & B12 \\
B20 & B21 & B22
\end{array}
\]

Square Matrix C

\[
\begin{array}{ccc}
\text{Rank 0} & \text{Rank 1} & \text{Rank 2} \\
\text{Rank 3} & \text{Rank 4} & \text{Rank 5} \\
\text{Rank 6} & \text{Rank 7} & \text{Rank 8}
\end{array}
\]

Each rank processes a submatrix product at a time. Finishes after $\sqrt{n}$ steps.
Cannon's Algorithm Computation

Step: Communicate submatrices and compute subA x SubC.

Number of blocks (p) = 9
Number of concurrent processes = 9

Source: Amin Jarrah, Abdel-Karim Al-Tamimi, Tala Albashir
Figure 6.1: Baseline Cannon2D algorithm where ranks shift the A and B submatrices along rows and columns of the processor geometry, in a ring topology.
Cannon's Code (I): Communicator

```c
void cannon(double *A, double *B, double *C, int N, MPI_Comm comm)
{
    int nums[2] = {n,n};
    int periodic[2] = {true, true};

    MPI_Comm newComm;
    MPI_Cart_create(comm, 2, nums, periodic, true, &newComm);

    int size, rank;
    MPI_Comm_size(newComm,&rankCount);
    MPI_Comm_rank(newComm,&myRank);

    int upRank, downRank, leftRank, rightRank;
    MPI_Cart_shift(newComm, 0, 1, &leftRank, &rightRank);
    MPI_Cart_shift(newComm, 1, 1, &downRank, &upRank);

    ... // Define Data Types and perform Cannon's algorithm

    MPI_Comm_free(&newComm);
    MPI_Finalize();
}
```
void cannon(double *A, double *B, double *C, int N, MPI_Comm comm) {

...

// Determine elements per side of submatrices
int p = sqrt(rankCount);
int n = N / p;

// Create MPI Datatypes
MPI_Datatype subMatrixType;
MPI_Type_contiguous(n*n, MPI_DOUBLE, &subMatrixType);
MPI_Type_commit(&subMatrixType);

// Initialize Submatrices
double subA = malloc(n*n); double subB = malloc(n*n); double subC = calloc(n*n);
MPI_Scatter(A -> subA, root = 0);
MPI_Scatter(B -> subA, root = 0);

...
}
void cannon(double *A, double *B, double *C, int N, MPI_Comm comm) {
    ...

    for (int step = 0; step < p; step++) {
        MPI_Irecv(tmpA, 1, subMatrixType, rightRank, newComm);
        MPI_Irecv(tmpB, 1, subMatrixType, downRank, newComm);

        MPI_Isend(A, 1, subMatrixType, leftRank, newComm);
        MPI_Isend(B, 1, subMatrixType, upRank, newComm);
        MPI_Waitall();

        swap(subA, tmpA); swap(subB, tmpB);
        cblas_dgemm(subA, subB, subC, n);
    }

    MPI_Gather(C <- subC, root = 0);
}
Example II:
Sending a Linked List
Example: Sending a linked list

We can use MPI_Type_create_struct to send the contents of linked lists:

Sender

```cpp
std::list<int> data;
for (int i=0; i<10; ++i) data.push_back(i);
```

```cpp
std::vector<MPI_Datatype> types(10, MPI_INT);
std::vector<int> blocklens(10, 1);
std::vector<MPI_Aint> offsets;
// How do we resolve the offsets?
```

Receiver

```cpp
std::vector<int> data(10);
MPI_Status status;
MPI_Recv(&data[0], 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
for (int i=0; i < data.size(); i++) printf("%d\n", data[i]);
```
Example: Sending a linked list

To send the linked list
1. View the whole memory as base struct which using absolute addresses as offsets.
2. Pass MPI_BOTTOM as the buffer pointer in communication to indicate absolute addressing.

```c
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);
```
Extra: Datatypes
Multidimensional Arrays

For multidimensional problems, **MPI_Type_create_subarray** is a more general solution.

```c
int MPI_Type_create_subarray(
    int ndims, int sizes[], int subsizes[], int starts[], int order,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**Rationale**: Builds 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
Indexed Datatypes

If you need to send a subset of the array, use indexed types:

```c
int MPI_Type_indexed(int count, int blocklens[], int indices[],
                     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**Rationale:** Builds an MPI datatype selecting specific entries from a contiguous array. Starting at each of the given indices a number of elements given in the corresponding entry of `blocklens` is chosen.

```c
int MPI_Type_create_hindexed(int count, int blocklens[], MPI_Aint
                              displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**Rationale:** Same as `MPI_Type_indexed` but now instead of indices the displacement in bytes from the start of the array is specified.
Extra: Groups
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
Choose ranks selectively:

```c
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 triplets of \{First, Last, Stride\}:

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