High Performance Computing for Science and Engineering II

9.3.2020
High-Throughput and High-Performance Computing

Lecturer: Dr. Sergio Martin
Today's Lecture

A Recap on HPC and UQ
- Why Uncertainty Quantification and Optimization?
- High Throughput vs. High Performance Computing

Advanced MPI Topics
- MPI Custom Datatypes
- Communication Topologies and Groups
- Examples
Course Contents

- Methods for Uncertainty Quantification and Optimization
- HPCSE II
- High Performance Computing
Why Quantify Uncertainty?

For Decision Making. Example: Hurricane.
Why Quantify Uncertainty?

**Medicine:** Designing better drugs and treatments for cancer patients.

---

**Measurements**

- Measurements
- Drug administration

---

**Robust Predictions**

- Measurements
- Expected prediction
- 95% credible interval
- 99% credible interval

Why Optimize?

Example: Shaping wings for reduced drag.

Source: State-of-the-art in aerodynamic shape optimisation methods. N. Skinner, H. Zare-Behtash
Why Optimize?

Improving medical devices for diagnosis.

Methodology: Bayesian Inference

- **Experimental Data** (i.e., Physical Observations)
- **Computational Model** (e.g., MPI-Based hydrodynamics solver)
- **Statistical Assumptions** (e.g., Model parameters)

\[ d = f(x \mid \vartheta) + \epsilon \]
\[ \epsilon \sim \mathcal{N}(0, \sigma_n) \]

Applying Bayes’ Theorem

\[ p(\vartheta \mid d) = \frac{p(d \mid \vartheta) p(\vartheta)}{p(d)} \]

Posterior Distribution of Parameters

Bayesian Inference: Evidence-based knowledge about the physical reality.
Inference Example: Heated Plate

Given:
A square metal plate with 3 sources of heat underneath it.

We have: ~10 temperature measurements at different locations

Can we infer the (x,y) locations of the 3 heat sources?
Inferring Parameters

**Experiment**

**Data:**
Temperature Measurements

**Model:**
2D Heat Equation (MPI)

**Assumption:**
\[ f(c_i) = -e^{-(x-x_i)^2+(y-y_i)^2} \]
Stochastic Approach to UQ

Better understanding of the system and models uncertainties.

UQ Methods
- CMA-ES
- Markov-Chain Monte-Carlo
- TMCMC

Experimental Measurements

Improved Initial Parameter Guess

Discretize / Approximate

Mathematical Model

Computed Results

Compare

Physical System
Stochastic Sampling

Inferring Parameters Probability Distributions.

Parameter Space

Annealing to the real Distribution
Stochastic Optimization

Minimizing a two-parameter function.

Parameters to Optimize

Global Minimum

Local Minima

Initial Guess
Extreme Computational Demands

Physical Model
Row of two posts with periodic boundary conditions.

Computational Model

Statistical Model
Optimization of post configuration over ~50 RBC types.
Extreme Computational Demands

Computational Demands Estimation:

GPU-Time per Evaluation: \(~7 \text{ hours}\)
50 Optimization Experiments \times 400 \text{ Evaluations} = 60,000 \text{ Model Evaluations}

Total usage: \(~140,000 \text{ Node Hours}\)

This represents 100\% \text{ Piz Daint for a whole day!}
High-Throughput vs. High-Performance

Fact I: Stochastic UQ methods alleviate the curse of dimensionality yet need still many model runs.

Fact II: For problems with many parameters, the number of model evaluations can be vast.

Our Goals:
- Develop fast and scalable computational models
- Ways to execute many evaluations given limited time and computational resources.

Our Tools:

**High-Throughput Computing**
Efficiently running many tasks for long-periods of time, using many computational resources

In this course, you’ll learn about:
- Korali, UPC++

**High Performance Computing**
Build scalable algorithms that run as fast as possible in short periods of time (FLOPs)

In this course, you’ll learn about:
- Advanced MPI Topics, CUDA,
  Communication-Tolerant Programming
High-Throughput & High-Performance

Running Experiments Sequentially:

- Average Efficiency 73.9%

Running Experiments Dynamically:

- Average Efficiency 97.8%
Advanced MPI Topics: Topologies
Quick Review: [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)

**Point-to-Point Communication**

Message Exchange

- **Processor 1**
  - Process A
    - Application SEND data
    - System buffer

- **Processor 2**
  - Process B
    - Application RECV data
    - System buffer

Path of a message buffered at the receiving process

**Collective Communication**

- Broadcast
- Scatter
- Gather
- Reduction

Processor 1

Processor 2
MPI Communicators

So far we have always handled communication among all ranks as part of the same communicator: `MPI_COMM_WORLD`.

Suppose, however, we need to communicate only among subsets of the ranks:

- We want to split the ranks into groups and build a new communicator for each group,
- We can then do communication operations within a group instead of within all ranks.

**Example:** Communicate among ranks in the same row or column in a matrix multiplication algorithm.
Example: Structured Grids

Structured Grid Stencil Solver

- Iteratively approaches a solution.

2D Grid

Node

Traditional Decomposition

1 Process (Rank) per Core.

Ranks Exchange Halo (Boundary) Cells

Rank 0

Rank 1

Rank 2

Rank 3
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 Provides an easy way to find neighbors in cartesian grids: Topologies
MPI Topologies

A connectivity graph-building utility for MPI Ranks.

- 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.

Source: [mathworld.wolfram.com/GridGraph.html](http://mathworld.wolfram.com/GridGraph.html)
Source: [think-like-a-git.net/](http://think-like-a-git.net/)
Cartesian Topology

A 2D Cartesian Topology:

Boundaries can be cyclic (periodic):

\[\text{left}_0 = 3 \quad \text{right}_3 = 0\]
Creating an MPI Cartesian Topology

```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.

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)
```

Here is a table showing the function calls and the expected dimensions:

<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>

This is a Collective Operation.
Creating a 3D-Grid Topology

Example: 3D Cartesian Grid

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

    int size;
    MPI_Comm_size(&size);

    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 cartesian topology
    MPI_Cart_create(MPI_COMM_WORLD, 3, nums, periodic, true, &cart_comm);
    MPI_Comm_free(&cart_comm);
    MPI_Finalize();
}
```
Finding Neighbors

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

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

```c
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.

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

```c
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```
Rationale: Get the coordinates of a given rank.
MPI Vector Datatypes
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.

```c
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.

```c
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 bytes.

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

Rationale: build an MPI datatype for a contiguous array.
Example: Exchange Boundaries of a Square Domain

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

```c
int NCols = 4;
int NRows = 4;
double* a[NCols * NRows];

MPI_Datatype rowType, colType;
MPI_Type_contiguous(NCols, MPI_DOUBLE, &colType);
MPI_Type_vector(NRows, 1, NCols, MPI_DOUBLE, &rowType);
MPI_Type_commit(&rowType);
MPI_Type_commit(&colType);
```

// Perform Work and Communication
```c
do_some_work();
MPI_Sendrecv(NCols, colType, ...);
MPI_Sendrecv(NRows, rowType, ...);
```

```c
MPI_Type_free(&row);
MPI_Type_free(&col);
```
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
Multidimensional Arrays (II)

MPI_Type_create_subarray((sizes, subsizes, starts, ...))

int sizes    = { N, N, N }
int subsizes = { N, 1, N }
int starts   = { 0, 0, 0 }

int sizes    = { N, N, N }
int subsizes = { N, 1, N }
int starts   = { 0, N-1, 0 }

// Perform Work and Communication
MPI_Sendrecv(1, topFaceType,    topNeighbor, ...);
MPI_Sendrecv(1, bottomFaceType, bottomNeighbor, ...);
Multidimensional Arrays (III)

MPI_Type_create_subarray(sizes, subsizes, starts)

```c
int sizes    = { N, N, N }
int subsizes = { 1, N, N }
int starts   = { 0, 0, 0 }
```

```c
int sizes    = { N, N, N }
int subsizes = { 1, N, N }
int starts   = { N-1, 0, 0 }
```
Multidimensional Arrays (III)

MPI_Type_create_subarray(sizes, subsizes, starts)

```c
int sizes    = { N, N, N }    
int subsizes = { N, N, 1 }    
int starts   = { 0, 0, 0 }    
```

```c
int sizes    = { N, N, N }    
int subsizes = { N, N, 1 }    
int starts   = { 0, 0, N-1 }   
```
MPI Struct Datatypes
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/
Riemann Sums

\[ y = x^2 \]

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);
}
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

```c
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);
}
```
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.
Distributing Parameters (Byte-wise Comm)

// Defining a struct for the parameters

```c
struct parms {
    double a; // lower bound of integration
    double b; // upper bound of integration
    int n; // number of subintervals
};
```

```c
Not Portable: Requires all ranks to have same architecture
```

```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
}
```

```c
// 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:

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

MPI_Aint offsets[3] = {0,
                        sizeof(short),
                        sizeof(short)+sizeof(double)};
```

E.g., PS3
E.g., Intel x86
32-bit integer

Little-endian

Memory

0A0B0C0D

Big-endian

Memory

0A0B0C0D

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:
- Describe your data layout to MPI and use it as an MPI datatype.

```c
struct parms {
    double a;
    double b;
    int nsteps;
};
```

<table>
<thead>
<tr>
<th>type</th>
<th>count</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DOUBLE</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>
## MPI Struct Type

### 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.

### `MPI_Type_commit`

```c
int MPI_Type_commit(MPI_Datatype *datatype)
```

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

### `MPI_Type_free`

```c
int MPI_Type_free(MPI_Datatype *datatype)
```

**Rationale:** Frees the data type, releasing any allocated memory.
// 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
};

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);
Distributing Parameters (with MPI Structs)

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

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

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?
```

```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);
```