ME964
High Performance Computing for Engineering Applications

Parallel Computing with MPI

Wrap Up, Collective Communications
MPI Derived Types

April 19, 2012

ME964 UW-Madison

> "As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications."
> Dave Parnas

Before We Get Started…

- Last lecture
  - MPI Send/Receive, the non-blocking flavor (MPI_Isend)
  - Collective communications: MPI_Broadcast, MPI_Reduce, MPI_Scatter

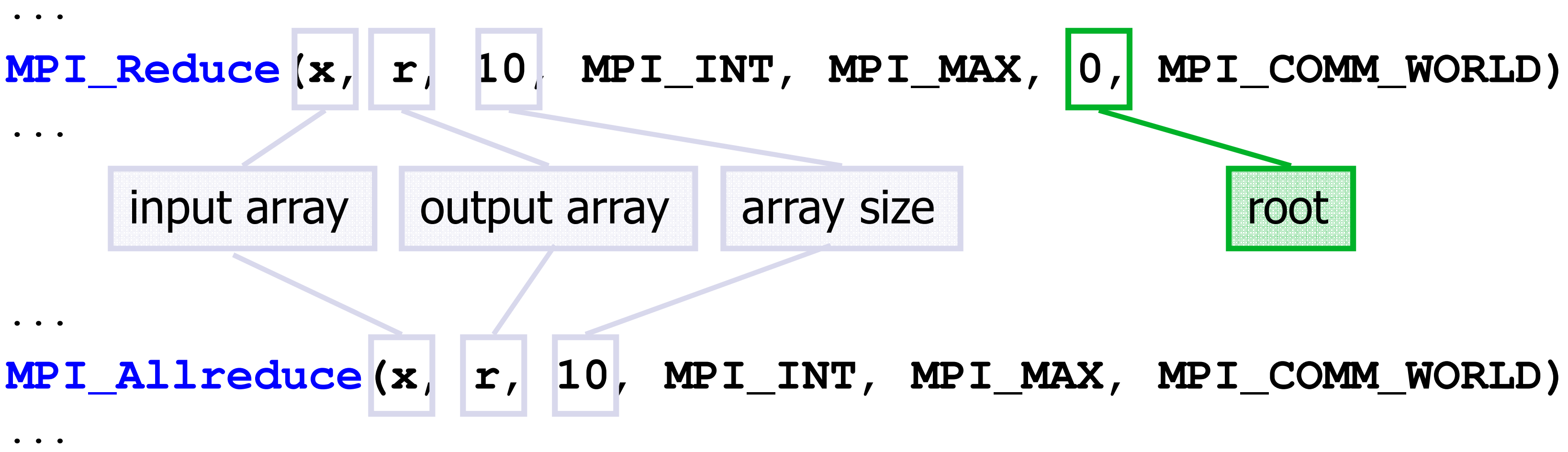
- Today
  - Wrap up, MPI collective communications
  - MPI Derived Types (Handling complex data)

- Other issues
  - Assignment 11 due Sunday, April 22 at 11:59 pm

MPI_Reduce, MPI_Allreduce

- `MPI_Reduce`: result is sent out to the root
  - The operation is applied element-wise for each element of the input arrays on each processor

- `MPI_Allreduce`: result is sent out to everyone



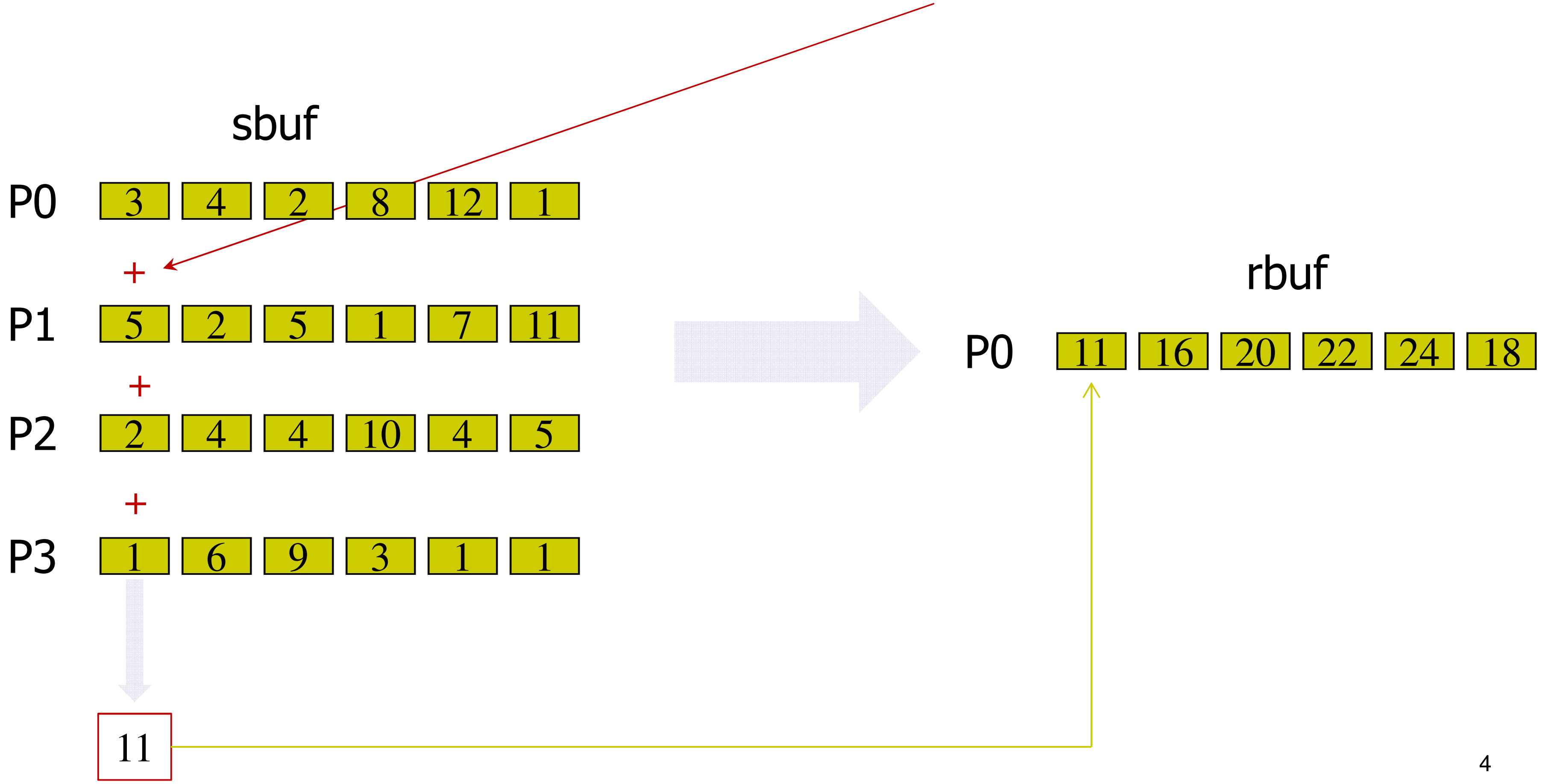
```
...
MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
...
```

```
...
MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD)
...
```

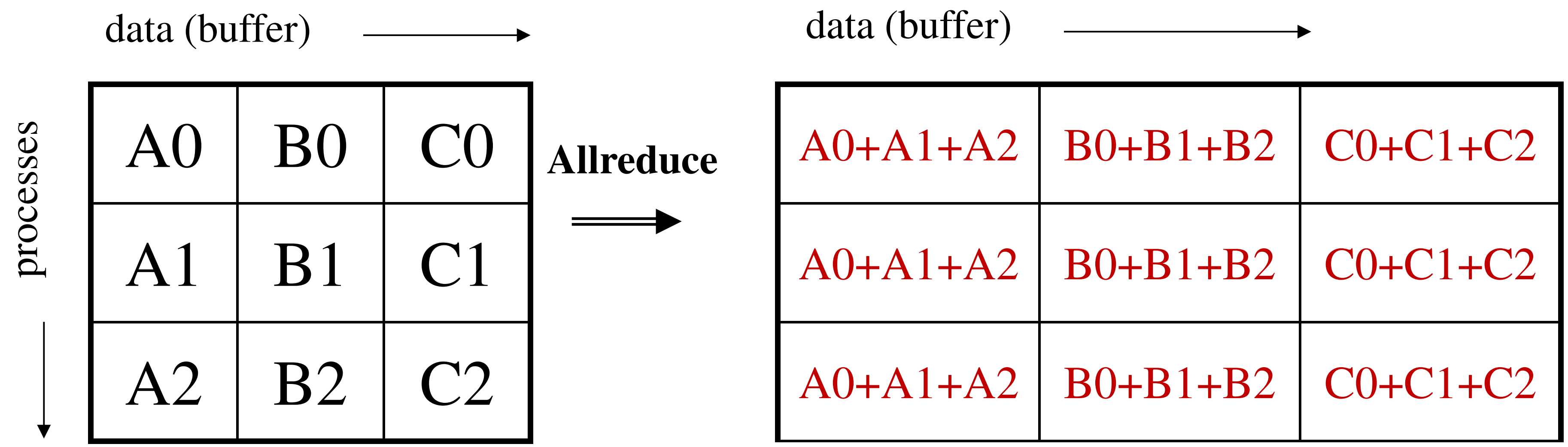
Credit: Allan Snavely

MPI_Reduce example

```
MPI_Reduce(sbuf,rbuf,6,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
```



MPI_Allreduce



data (buffer) →

processes ↓

A0	B0	C0
A1	B1	C1
A2	B2	C2

Allreduce

data (buffer) →

A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2
A0+A1+A2	B0+B1+B2	C0+C1+C2

[A. Siegel]→

MPI_Allreduce

```
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

IN	`sendbuf`	(address of send buffer)
OUT	`recvbuf`	(address of receive buffer)
IN	`count`	(number of elements in send buffer)
IN	`datatype`	(data type of elements in send buffer)
IN	`op`	(reduce operation)
IN	`comm`	(communicator)

Example: MPI_Allreduce

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int my_rank, nprocs, gsum, gmax, gmin, data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    data_l = my_rank;

    MPI_Allreduce(&data_l, &gsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmin, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("gsum: %d, gmax: %d  gmin:%d\n", gsum,gmax,gmin);
    MPI_Finalize();
}
```

Example: MPI_Allreduce

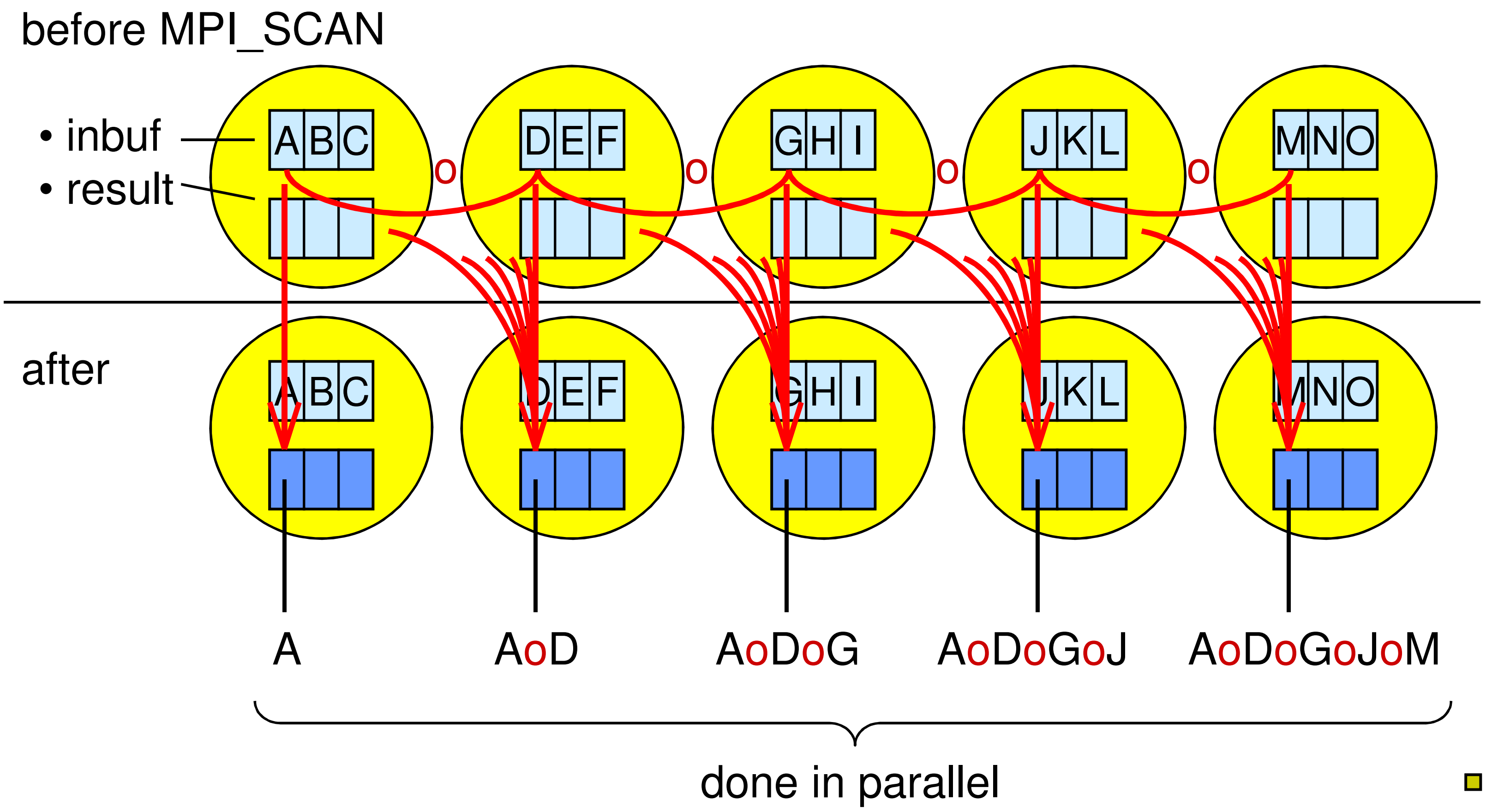
[Output]

```
[negrut@euler24 CodeBits]$ mpiexec  -np 10 me964.exe
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
[negrut@euler24 CodeBits]$
```

MPI_SCAN

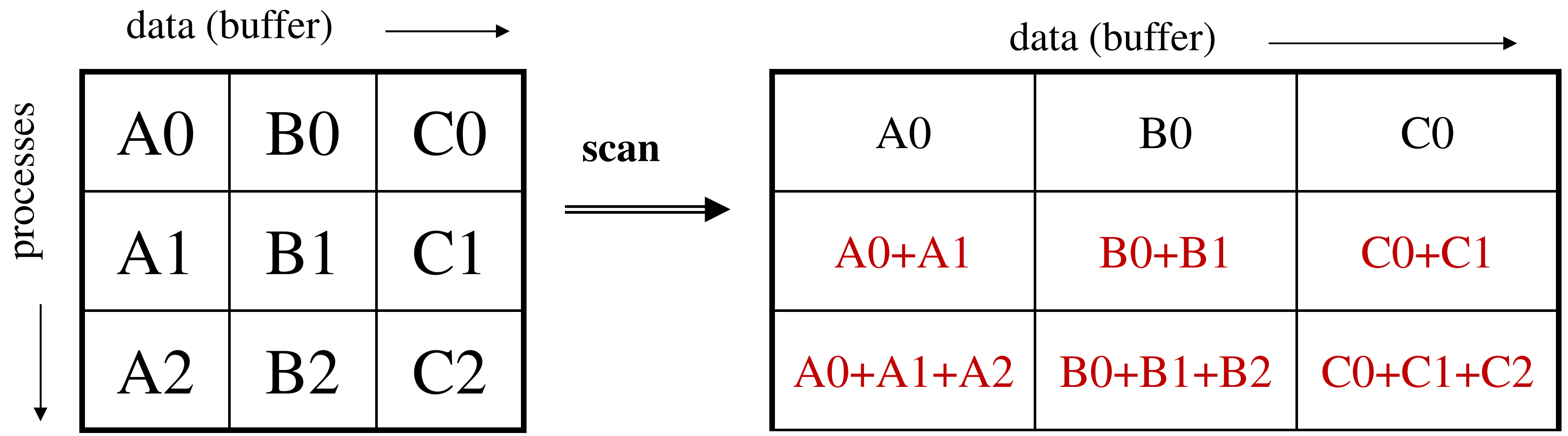
- Used to perform a prefix reduction on data distributed across a comm
- The operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i` (inclusive)
- The type of operations supported, their semantics, and the constraints on send and receive buffers are as for `MPI_REDUCE`

MPI_SCAN



[ICHEC]→

Scan Operation



data (buffer) →

processes ↓

A0	B0	C0
A1	B1	C1
A2	B2	C2

scan

data (buffer) →

A0	B0	C0
A0+A1	B0+B1	C0+C1
A0+A1+A2	B0+B1+B2	C0+C1+C2

[A. Siegel]→

MPI_Scan: Prefix reduction

- Process i receives data reduced on process `0` through `i`

sbuf

P0	3	4	2	8	12	1
P1	5	2	5	1	7	11
P2	2	4	4	10	4	5
P3	1	6	9	3	1	1

rbuf

P0	3	4	2	8	12	1
P1	8	6	7	9	19	12
P2	10	10	11	19	23	17
P3	11	16	12	22	24	18

```
MPI_Scan(sbuf,rbuf,6,MPI_INT,MPI_SUM,MPI_COMM_WORLD)
```

[A. Snavely]→

MPI_Scan

```
int MPI_Scan (void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

IN	`sendbuf`	(address of send buffer)
OUT	`recvbuf`	(address of receive buffer)
IN	`count`	(number of elements in send buffer)
IN	`datatype`	(data type of elements in send buffer)
IN	`op`	(reduce operation)
IN	`comm`	(communicator)

- Note: `count` refers to total number of elements that will be received into receive buffer after operation is complete

[A. Siegel]→

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs,i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n){
        if( myRank == n ) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d.  Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Scan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Example: MPI_Scan

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me964.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me964.exe
Process 0. Entry: 0.  Value: 0
Process 0. Entry: 1.  Value: 0

Process 1. Entry: 0.  Value: 1
Process 1. Entry: 1.  Value: 2

Process 2. Entry: 0.  Value: 2
Process 2. Entry: 1.  Value: 4

Process 3. Entry: 0.  Value: 3
Process 3. Entry: 1.  Value: 6
```

```
Post Scan - Content on Process: 0
Entry: 0. Scan Val: 0
Entry: 1. Scan Val: 0

Post Scan - Content on Process: 1
Entry: 0. Scan Val: 1
Entry: 1. Scan Val: 2

Post Scan - Content on Process: 2
Entry: 0. Scan Val: 3
Entry: 1. Scan Val: 6

Post Scan - Content on Process: 3
Entry: 0. Scan Val: 6
Entry: 1. Scan Val: 12
[negrut@euler26 CodeBits]$
```

MPI_Exscan

- `MPI_Exscan` is like `MPI_Scan`, except that the contribution from the calling process is not included in the result at the calling process (it is contributed to the subsequent processes)
- The value in `recvbuf` on the process with rank `0` is undefined, and `recvbuf` is not signficant on process `0`
- The value in `recvbuf` on the process with rank 1 is defined as the value in `sendbuf` on the process with rank `0`
- For processes with rank `i > 1`, the operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i-1` (inclusive)
- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for `MPI_REDUCE`

MPI_Exscan

```
int MPI_Exscan (void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

IN	`sendbuf`	(address of send buffer)
OUT	`recvbuf`	(address of receive buffer)
IN	`count`	(number of elements in send buffer)
IN	`datatype`	(data type of elements in send buffer)
IN	`op`	(reduce operation)
IN	`comm`	(communicator)

[A. Siegel]→

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs,i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n){
        if( myRank == n ) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d.  Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Exscan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Example: MPI_Exscan

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me964.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me964.exe
Process 0. Entry: 0.  Value: 0
Process 0. Entry: 1.  Value: 0

Process 1. Entry: 0.  Value: 1
Process 1. Entry: 1.  Value: 2

Process 2. Entry: 0.  Value: 2
Process 2. Entry: 1.  Value: 4

Process 3. Entry: 0.  Value: 3
Process 3. Entry: 1.  Value: 6
```

```
Post Scan - Content on Process: 0
Entry: 0. Scan Val: 321045752
Entry: 1. Scan Val: 32593

Post Scan - Content on Process: 1
Entry: 0. Scan Val: 0
Entry: 1. Scan Val: 0

Post Scan - Content on Process: 2
Entry: 0. Scan Val: 1
Entry: 1. Scan Val: 2

Post Scan - Content on Process: 3
Entry: 0. Scan Val: 3
Entry: 1. Scan Val: 6
[negrut@euler26 CodeBits]$
```

User-Defined Reduction Operations

- Operator handles
  - Predefined – see table of last lecture: MPI_SUM, MPI_MAX, etc.
  - User-defined
- User-defined operation ■:
  - Should be associative
  - User-defined function must perform the operation vector_A ■ vector_B
- Registering a user-defined reduction function:

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

- `commute` tells the MPI library whether `func` is commutative or not

Example:

Norm 1 of a Vector

```
#include <mpi.h>
#include <stdio>
#include <math.h>

void oneNorm(float *in, float *inout, int *len,
             MPI_Datatype *type)
{
 int i;
 for (i=0; i<*len; i++) {
    *inout = fabs(*in) + fabs(*inout);      /* one-norm */
    in++;
    inout++;
   }
}
```

Continues here…

```
int main(int argc, char* argv[]) {
  int root=0, p, myid;
  float sendbuf, recvbuf;
  MPI_Op myop;

  int commutes=1;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  //create the operator...
  MPI_Op_create(onenorm, commune, &myop);

  //get some fake data used to make the point...
  sendbuf = myid*(-1)^myid;
  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Reduce (&sendbuf, &recvbuf, 1, MPI_FLOAT, myop, root, MPI_COMM_WORLD);
  if( myid == root )
             printf("The operation yields %f\n", recvbuf);
  MPI_Finalize();
  return 0;
}
```

`thrust` code more simple...

```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

template <typename T> struct absval {
    __host__ __device__
        T operator()(const T& x) const {
            return fabs(x);
    }
};

int main(void)
{
    // initialize host array
    float x[4] = {1.0, -2.0, 3.0, -4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    absval<float> unary_op;
    float res = thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, 0.f, thrust::plus<float>());

    std::cout << res << std::endl;
    return 0;
}
```

MPI Derived Types

[Describing Non-contiguous and Heterogeneous Data]

The Relevant Question

- The relevant question that we want to be able to answer?
  - "What's in your buffer?"
- Communication mechanisms discussed so far allow send/recv of a contiguous buffer of identical elements of predefined data types
- Often want to send non-homogenous elements (structure) or chunks that are not contiguous in memory
- MPI enables you to define derived data types to answer the question "What's in your buffer?"

MPI Datatypes

- MPI Primitive Datatypes
  - `MPI_Int`, `MPI_Float`, `MPI_INTEGER`, etc.

- Derived Data types - can be constructed by four methods:
    - contiguous
    - vector
    - indexed
    - struct
  - Can be subsequently used in all point-to-point and collective communication

- The motivation: create your own types to suit your needs
  - More convenient
  - More efficient

Type Maps

[Jargon]

- A derived data type specifies two things:
  - A sequence of primitive data types
  - A sequence of integer (byte) displacements, measured from the beginning of the buffer
- Displacements are not required to be positive, distinct, or in increasing order (however, negative displacements will precede the buffer)
- Order of items need not coincide with their order in memory, and an item may appear more than once

Type Map

Primitive datatype 0	Displacement of 0
Primitive datatype 1	Displacement of 1
…	…
Primitve datatype n-1	Displacement of n-1

Map Type, Examples

- Assume that Type = {(double, 0), (char, 8)} where doubles have to be strictly aligned at addresses that are multiples of 8. What is the extent of this data type?

Ans: 16

- What is extent of type {(char, 0), (double, 8)}?

Ans: 16

- Is this a valid type: {(double, 8), (char, 0)}?

Ans: yes, since order does not matter

Example

- What is Type Map of `MPI_INT`, `MPI_DOUBLE`, etc.?
  - {(int,0)}
  - {(double, 0)}
  - Etc.

[A. Siegel]→

Type Signature

[Jargon]

- The sequence of primitive data types (i.e. displacements ignored) is the type signature of the data type

- Example: a type map of

  {(double,0),(int,8),(char, 12)}

- …has a type signature of

  {`double`, `int`, `char`}

Extent

[Jargon]

- Extent: distance, in bytes, from beginning to end of type

- More specifically, the extent of a data type is defined as:

  … the span from the first byte to the last byte occupied by entries in this data type (rounded up to satisfy alignment requirements)

- Example:
  - Type={(**double**,0),(**char**,8)} i.e. offsets of 0 and 8 respectively.
  - Now assume that doubles are aligned strictly at addresses that are multiples of 8
  - extent = 16 (9 rounds to next multiple of 8, which is where the next double would land)

Data Type Interrogators

`MPI_Aint` - C type that
holds any valid address

```
int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent);
```

- `datatype` - primitive or derived `datatype`
- `extent` - returns extent of `datatype` in bytes

```
int MPI_Type_size (MPI_Datatype datatype, int *size);
```

- `datatype` - primitive or derived `datatype`
- `size` - returns size in bytes of the entries in the *type signature* of `datatype`
  - Gaps don’t contribute to size
  - This is the total size of the data in a message that would be created with this datatype
  - Entries that occur multiple times in the datatype are counted with their multiplicity

Committing Data Types

- Each derived data type constructor returns an *uncommited* data type. Think of commit process as a compilation of data type description into efficient internal form

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

- Required for any derived data type before it can be used in communication

- Subsequently can use in any function call where an `MPI_Datatype` is specified

MPI_Type_free

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

- Call to `MPI_Type_free` sets the value of data type to `MPI_DATATYPE_NULL`
- Data types that were derived from the defined data type are unaffected.

MPI Type-Definition Functions

[“constructors”]

- `MPI_Type_Contiguous`: a replication of data type into contiguous locations
- `MPI_Type_vector`: replication of data type into locations that consist of equally spaced blocks
- `MPI_Type_create_hvector`: like vector, but successive blocks are not multiple of base type extent
- `MPI_Type_indexed`: non-contiguous data layout where displacements between successive blocks need not be equal
- `MPI_Type_create_struct`: most general – each block may consist of replications of different data types

  - The inconsistent naming convention is unfortunate but carries no deeper meaning. It is a compatibility issue between old and new version of MPI.

[A. Siegel]→

MPI_Type_contiguous

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

- IN count (replication count)
- IN oldtype (base data type)
- OUT newtype (handle to new data type)

- Creates a new type which is simply a replication of old type into contiguous locations

[A. Siegel]→

```c
#include <stdio.h>
#include<mpi.h>
/* !!! Should be run with at least four processes !!! */
int main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;
        int y;
        int z;
    } point;
    MPI_Datatype ptype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if( rank==3 ){
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    }
    else if( rank==1 ) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d) \n",rank,point.x,point.y,point.z);
    }
    MPI_Type_free(&ptype);
    MPI_Finalize();
    return 0;
}
```

Example: MPI_Type_contiguous

[Output]

```
[negrut@euler24 CodeBits]$ mpiexec  -np 10 me964.exe
P:1 received coords are (15,23,6)
[negrut@euler24 CodeBits]$
```