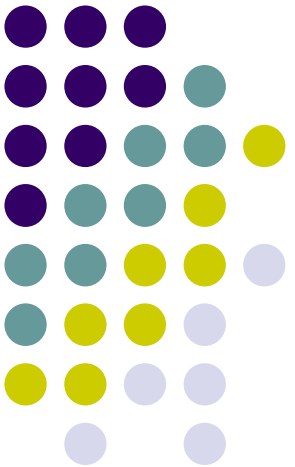
ME759
High Performance Computing for Engineering Applications

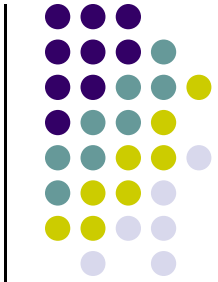
Parallel Computing with the Message Passing Interface (MPI)

November 6, 2013



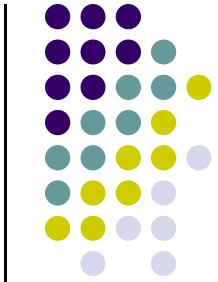
"Now this is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning."
-- Winston Churchill

Before We Get Started…



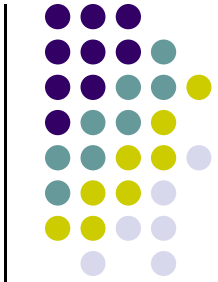
- Last time:
  - Wrap up point-to-point communication in MPI: non-blocking flavors
  - Collective action: barriers, communication, operations
- Today:
  - Collective action: operations
  - User defined types in MPI
  - Departing thoughts: CUDA, OpenMP, MPI
- Miscellaneous
  - No class on Friday. Time slot set aside for midterm exam
  - Midterm exam is Nov. 25 at 7:15 PM in room 1163ME
    - Review session on Monday, Nov 25 during regular class. Attend if you have questions
  - I will travel and miss four office hours: next week and subsequent week
    - I am checking my email on daily basis
  - Final Project Proposal due at 11:59 PM on Nov. 15

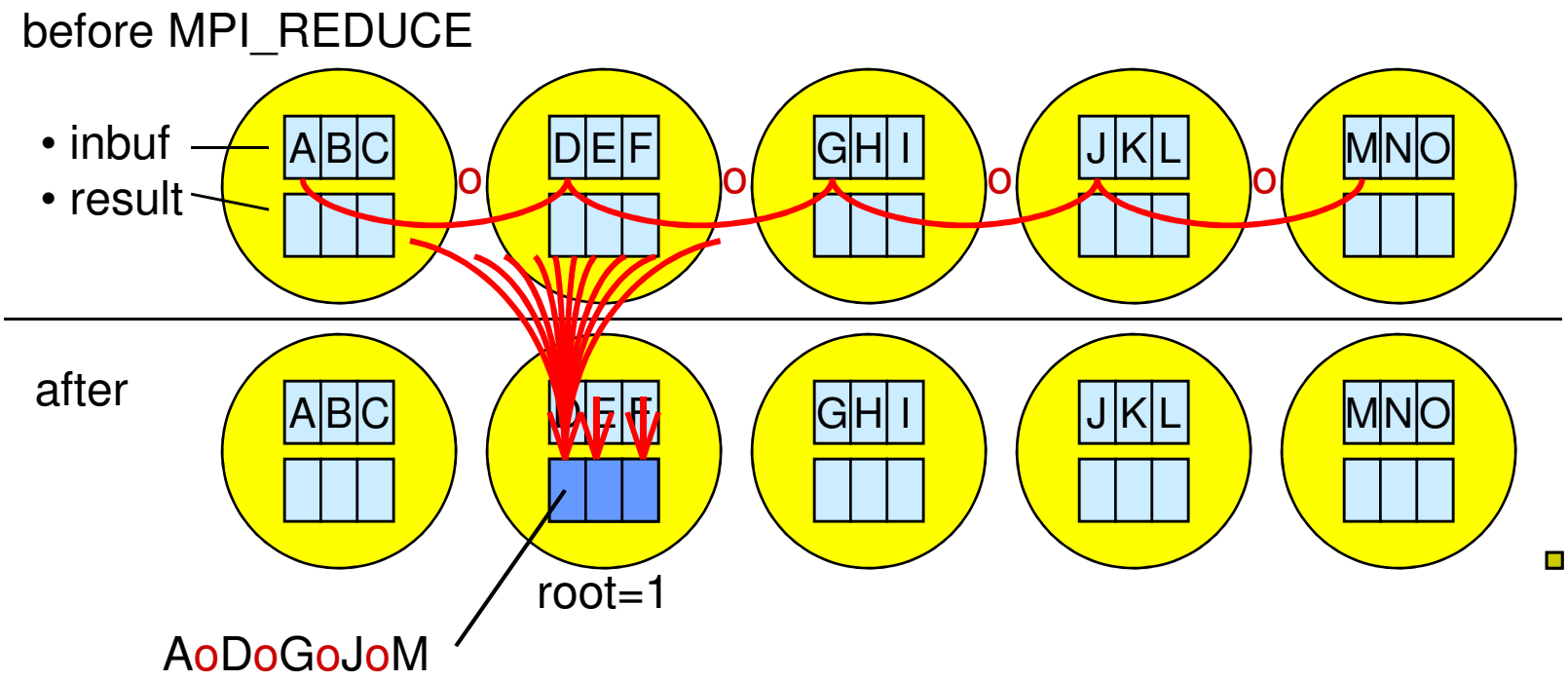
Midterm & Final Project Partitioning



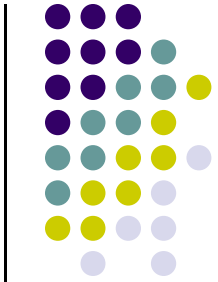
- If you are happy with your Midterm Project, it can become your Final Project
  - No midterm project report due then
- If not happy w/ your Midterm Project selection: November 15 provides the opportunity to wrap up and choose a different Final Project
  - Report should be detailed and follow rules spelled out in forum posting
- Nov 15: Final Project proposal should be uploaded
  - Do so even if you choose to continue Midterm Project
    - In this case simply upload a one liner stating this
  - If changing to new project, submit a proposal that details the work to be done
- For SPH default project: the student[s] w/ the fastest implementation will write a paper with Arman, Dan and another lab member

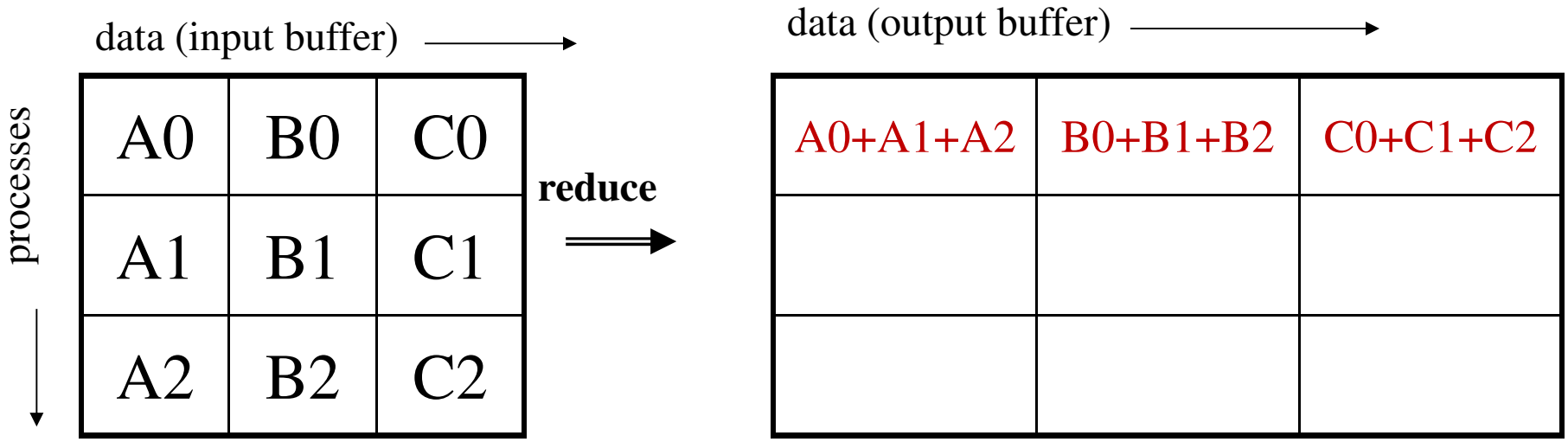
MPI_Reduce




before MPI_REDUCE
• inbuf
• result
A B C
o
D E F
o
G H I
o
J K L
o
M N O
after
A B C
G H I
J K L
M N O
root=1
AoDoGoJoM

Reduce Operation

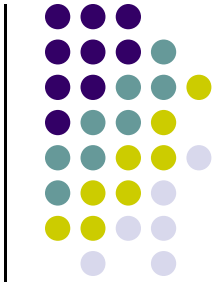




Assumption: Rank 0 is the root

[A. Siegel]→

MPI_Reduce

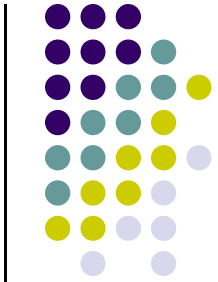


```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

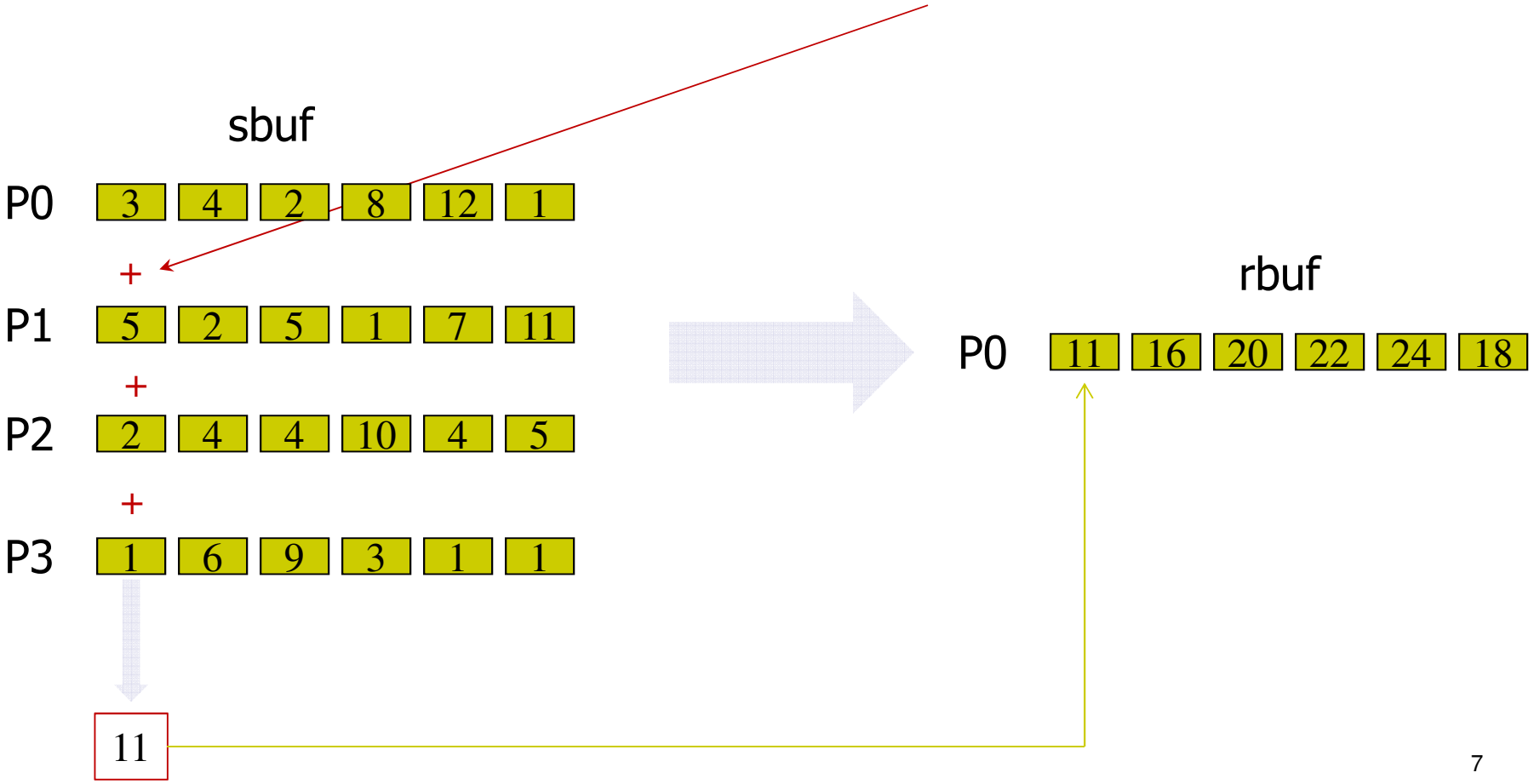
- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `root` (rank of root process)
- IN `comm` (communicator)

[A. Siegel]→

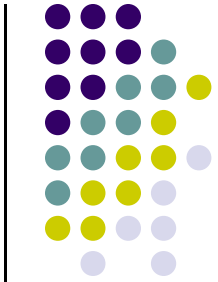
MPI_Reduce example



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



MPI_Reduce, MPI_Allreduce



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

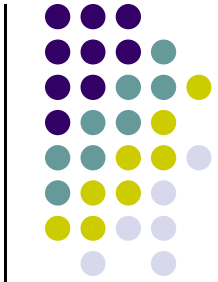
```
...
MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
...
```

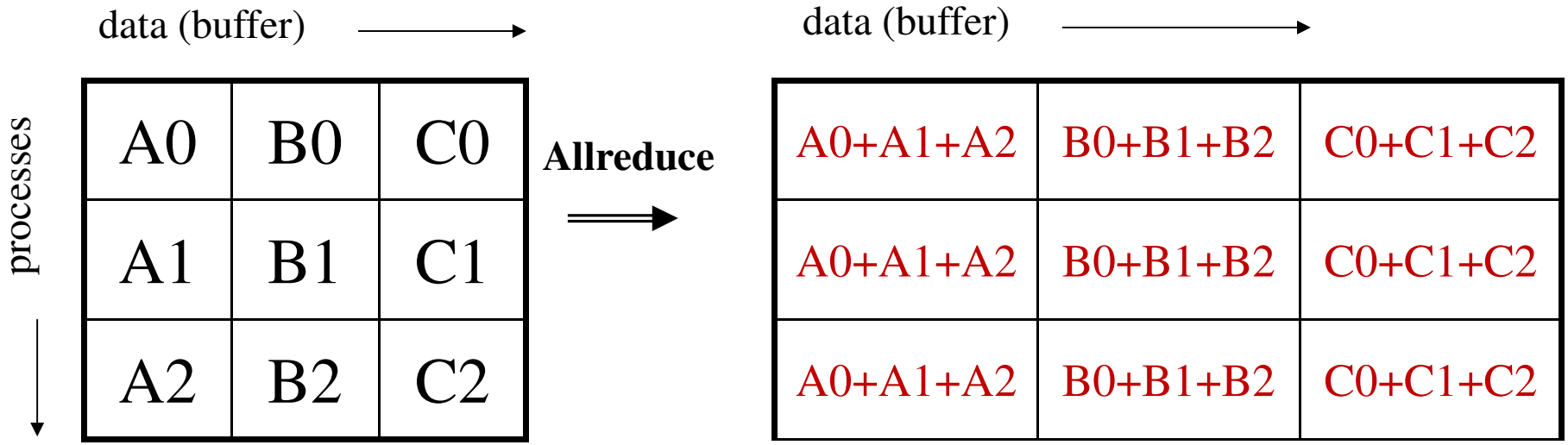
input array | output array | array size | root

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


Credit: Allan Snavely

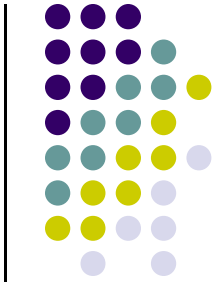
MPI_Allreduce





[A. Siegel]→

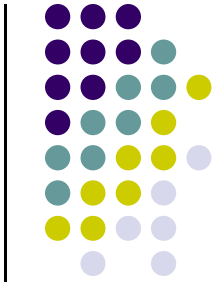
MPI_Allreduce



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

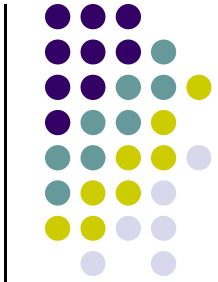
- IN sendbuf (address of send buffer)
- OUT recvbuf (address of receive buffer)
- IN count (number of elements in send buffer)
- IN datatype (data type of elements in send buffer)
- IN op (reduce operation)
- IN comm (communicator)

Example: MPI_Allreduce



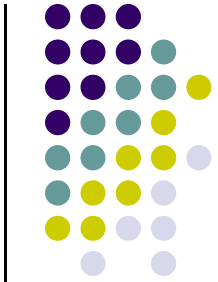
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

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]



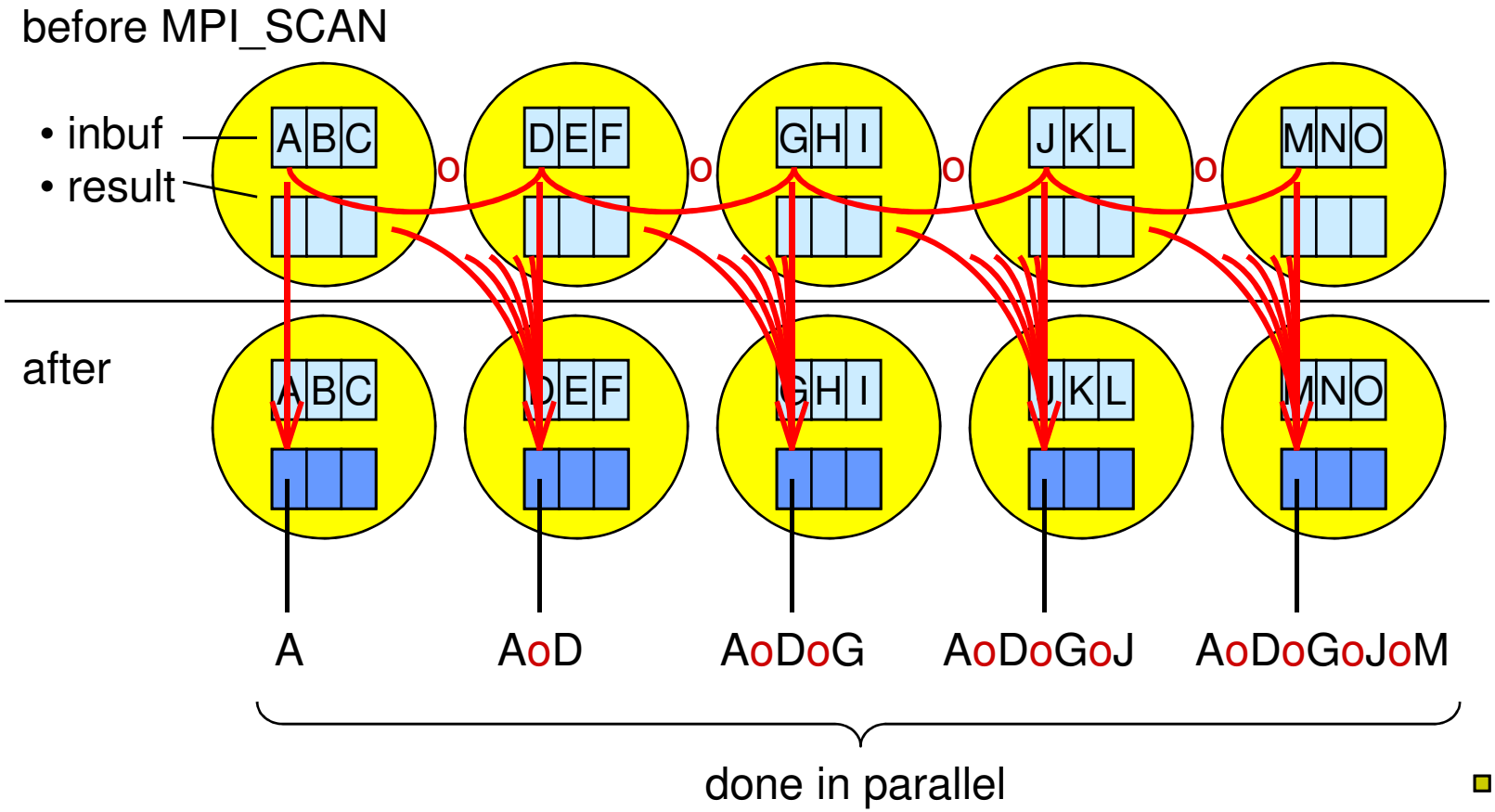
```
[negrut@euler24 CodeBits]$ mpiexec  -np 10 me759.exe
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
gsum: 45, gmax: 9  gmin:0
[negrut@euler24 CodeBits]$
```

MPI_SCAN

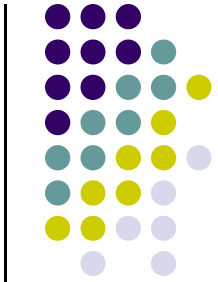


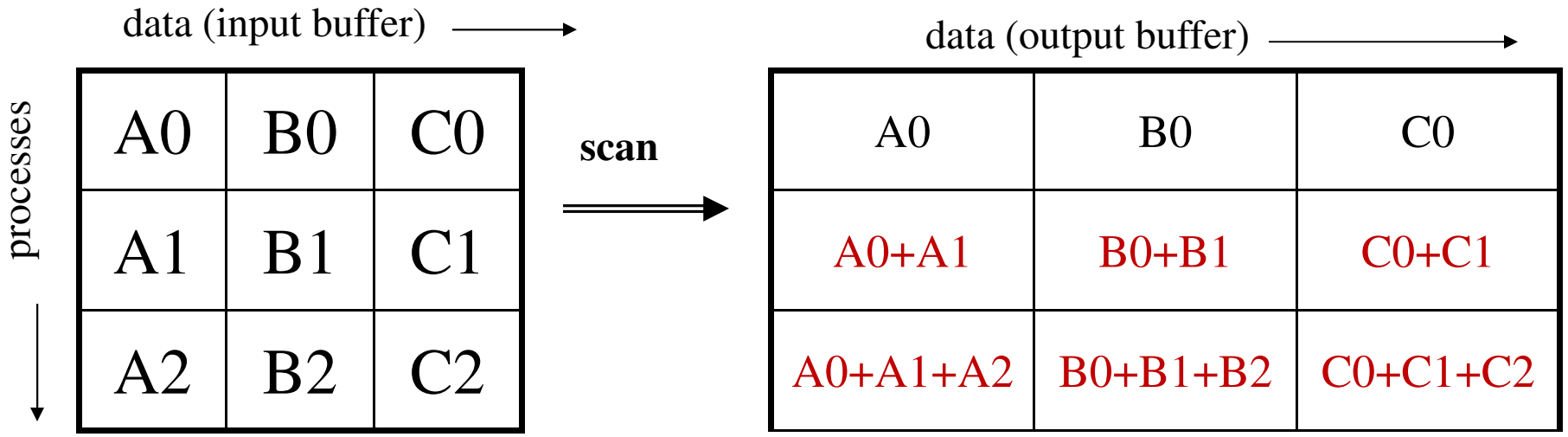
- Performs a prefix reduction on data distributed across a communicator
- The operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i` (inclusive)
- The type of operations supported, their semantics, and the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

MPI_SCAN


before MPI_SCAN
• inbuf
• result
A B C
D E F
G H I
J K L
M N O
o
o
o
o
after
A B C
D E F
G H I
J K L
M N O
A
AoD
AoDoG
AoDoGoJ
AoDoGoJoM
done in parallel

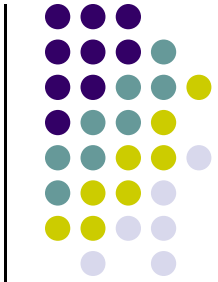
Scan Operation



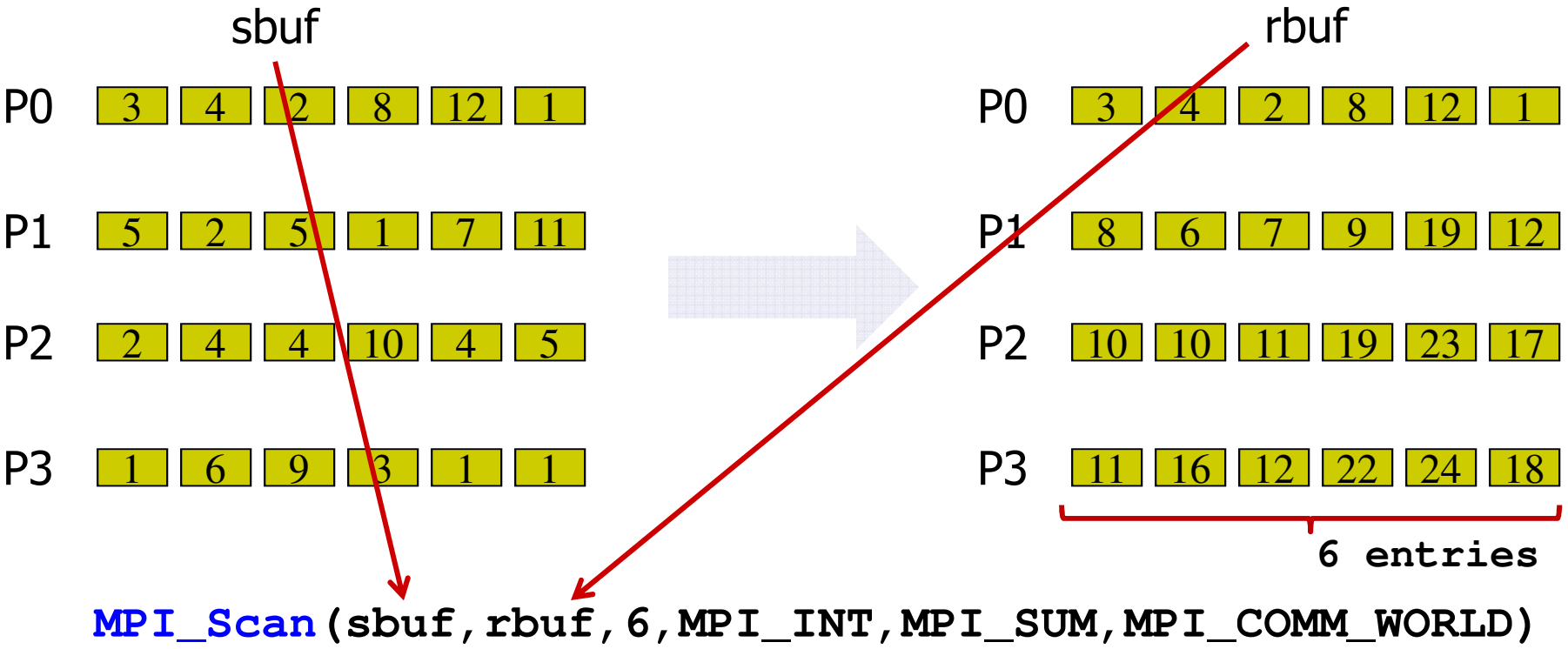


[A. Siegel]→

MPI_Scan: Prefix reduction



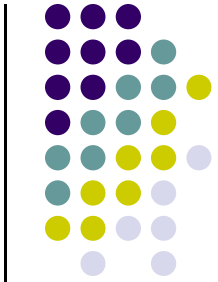
- Process i receives data reduced on process `0` through `i`




```
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);
```

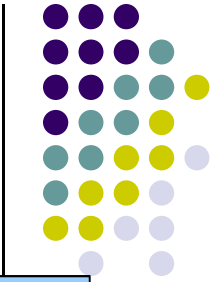
- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `comm` (communicator)

- 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();
    free(result); free(data_l);
    return 0;
}
```

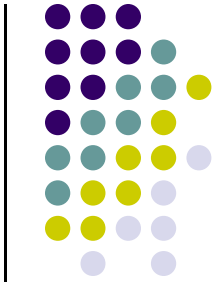
Example: MPI_Scan
[Output]



```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.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
```

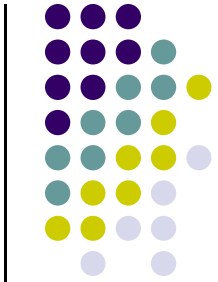
```
Post Scan - Content on Process: 0
Entry: 0. Scan Val: 0
Entry: 1. Scan Val: 0

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



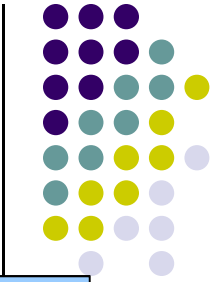
```
int MPI_Exscan (void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- 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]→

```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_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;
}
```

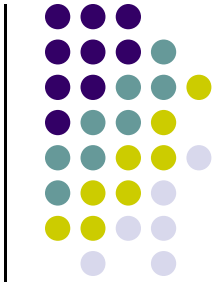
Example: MPI_Exscan
[Output]



```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.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”

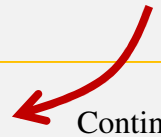
- Registering a user-defined reduction function:

```
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

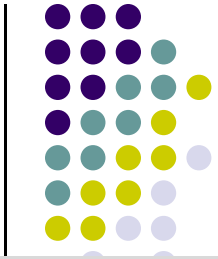
```c
#include <mpi.h>
#include <stdio>
#include <math.h>

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++;
   }
}
```



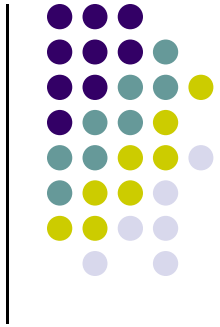
Continues here…

```c
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…



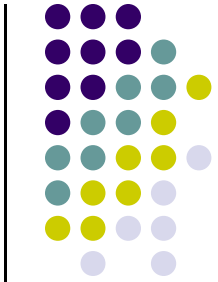
```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

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



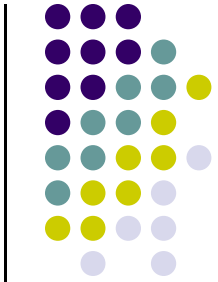
- The relevant question that we want to be able to answer?
  - "What's in your buffer?"

- 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

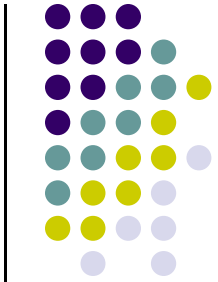


- MPI Primitive Datatypes
  - `MPI_Int`, `MPI_Float`, `MPI_INTEGER`, etc.

- 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

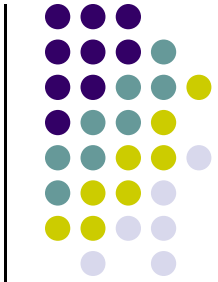
- The motivation: create your own types to suit your needs
  - More convenient
  - More efficient

Type Maps
[Jargon]



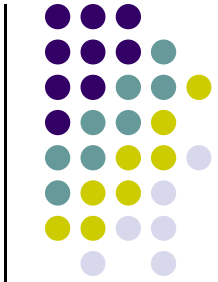
- A derived data type specifies two things:
  - A sequence of primitive data types
  - A sequence of integers that represent the byte displacements, measured from the beginning of the buffer
- Displacements are not required to be positive, distinct, or in increasing order (however, negative displacements 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

Extent
[Jargon]



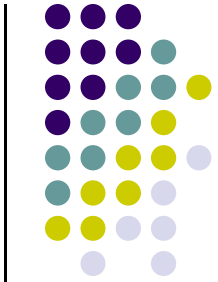
- Extent: distance, in bytes, from beginning to end of type

- 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)

Map Type, Examples



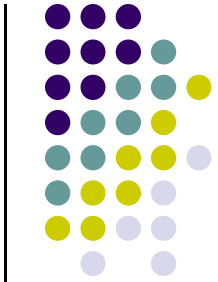
- What is extent of type {(char, 0), (double, 8)}?

  Ans: 16

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

  Ans: yes, since order does not matter

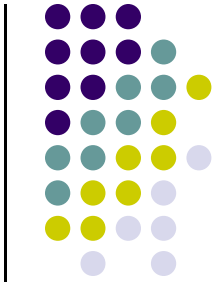
Example



- 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`}

Data Type Interrogators

MPI_Aint - C type that holds any valid address

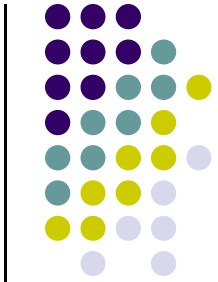
```
int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent);
```

- 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



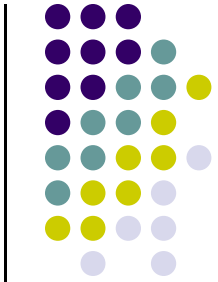
- Each derived data type constructor returns an *uncommited* data type. Think of commit process as a compilation of data type description into efficient internal form

```
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

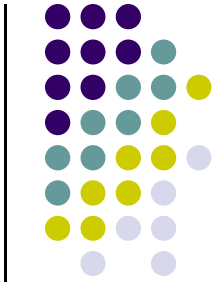
MPI_Type_free



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

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

MPI Type-Definition Functions
["constructors"]

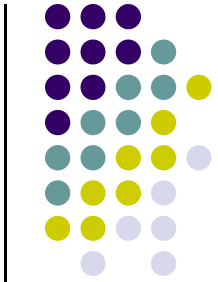


- `MPI_Type_Contiguous`: a replication of data type into contiguous locations
- `MPI_Type_vector`: replication of data type into locations that consist of equally spaced blocks
- `MPI_Type_create_hvector`: like vector, but successive blocks are not multiple of base type extent
- `MPI_Type_indexed`: non-contiguous 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);
```

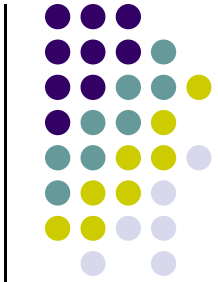
- IN count (replication count)
- IN oldtype (base data type)
- OUT newtype (handle to new data type)

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

[A. Siegel]→

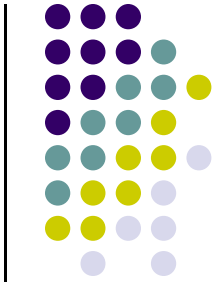
```c
#include <stdio.h>
#include<mpi.h>
/* !!! Should be run with at least four processes !!! */
int main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;
        int y;
        int z;
    } point;
    MPI_Datatype ptype;

    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 me759.exe
P:1 received coords are (15,23,6)
[negrut@euler24 CodeBits]$
```

Motivation: MPI_Type_vector



- Assume you have a 2D array of integers, and want send the last column

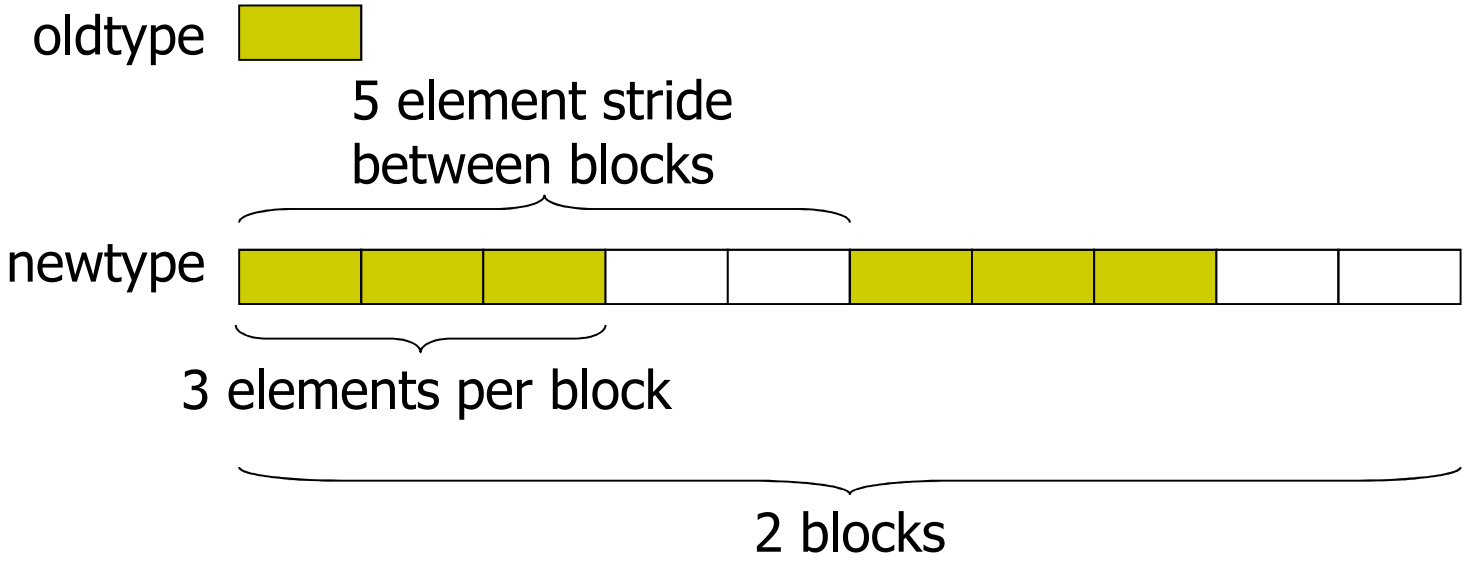
```
int x[4][8];
```

Content of x:

10	11	12	13	14	15	16	17
100	101	102	103	104	105	106	107
1000	1001	1002	1003	1004	1005	1006	1007
10000	10001	10002	10003	10004	10005	10006	10007

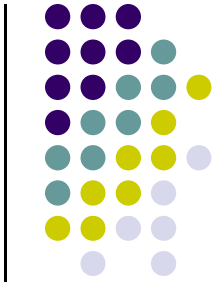
- There should be a way to say that I want to transfer integers, 4 of them, and they are stored in array x 8 integers apart (the stride)

MPI_Type_vector: Example



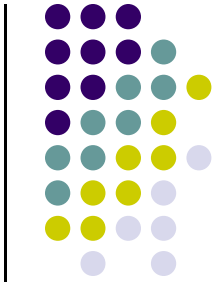
- count = 2
- blocklength = 3
- stride = 5

MPI_Type_vector



- `MPI_Type_vector` is a constructor that allows replication of a data type into locations that consist of equally spaced blocks.
- Each block is obtained by concatenating the same number of copies of the old data type
- Spacing between blocks is a multiple of the extent of the old data type
- One way to look at it:
  - You want some entries but don't care about other entries in an array
  - There is a repeatability to this pattern of "wanted" and "not wanted" entries

MPI_Type_vector

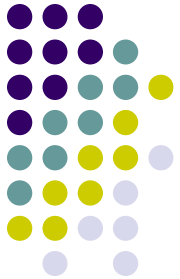


```
MPI_Type_vector (int count, int blocklength, int stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- IN `count` (number of blocks)
- IN `blocklength` (number of elements per block)
- IN `stride` (spacing between start of each block, measured as # elements)
- IN `oldtype` (base datatype)
- OUT `newtype` (handle to new type)

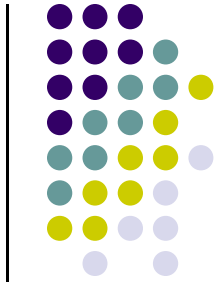
- Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of `oldtype` with a stride that is multiple of extent of old type

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

int main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;

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

    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);

    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }

    MPI_Type_free(&coltype);
    MPI_Finalize();
    return 0;
}
```




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

Example: MPI_Type_vector
[Output]

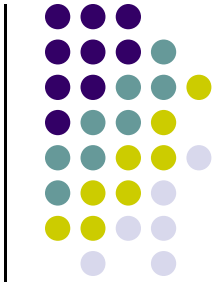


Content of x:

10	11	12	13	14	15	16	17
100	101	102	103	104	105	106	107
1000	1001	1002	1003	1004	1005	1006	1007
10000	10001	10002	10003	10004	10005	10006	10007

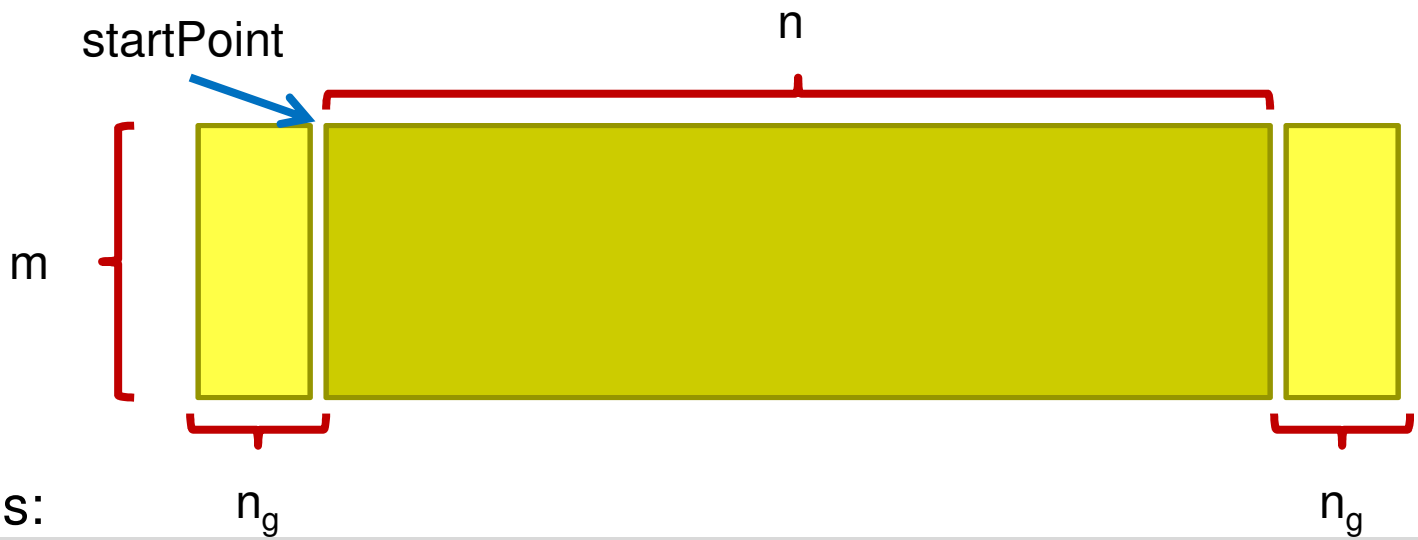
```
[negrut@euler19 CodeBits]$ mpiexec -np 12 me759.exe
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
[negrut@euler19 CodeBits]$
```

Example: MPI_Type_vector



- Given: Local 2D array of interior size **mxn** with $\mathbf{n_g}$ ghostcells at each edge
- You wish to send the interior (non ghostcell) portion of the array
- How would you describe the data type to do this in a single MPI call?

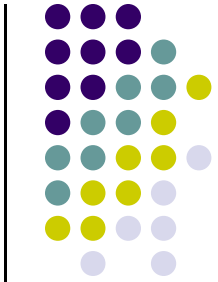
```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```



- Ans:

```
MPI_Type_vector (m, n, n+2*ng, MPI_DOUBLE, &interior);
MPI_Type_commit (&interior);
MPI_Send (startPoint, 1, interior, dest, tag, MPI_COMM_WORLD);
```

Type Map Example



- Start with `oldtype` for which
  Type Map = {(double, 0), (char, 8)}

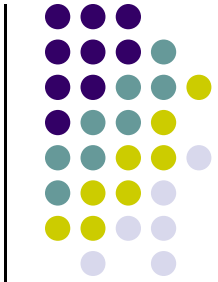
- What is Type Map of `newtype` if defined as below?
  `MPI_Type_vector`(2,3,4,`oldtype`,&`newtype`)

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

Ans:

{{(double, 0), (char, 8)}, {(double,16),(char,24) }, {(double,32),(char,40) },
{(double,**64**),(char,72), {(double,80),(char,88) }, {(double,96),(char,104)}}

Exercise: MPI_Type_vector



- Express

```
MPI_Type_contiguous(count, old, &new);
```

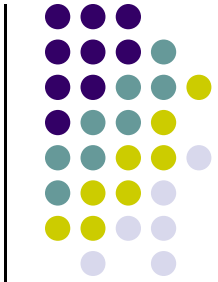
…as a call to `MPI_Type_vector`

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

- Ans:

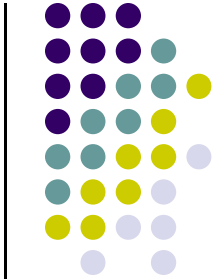
```
MPI_Type_vector (count, 1, 1, old, &new);
MPI_Type_vector (1, count, count, old, &new);
```

Outline



- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks

MPI – We're Scratching the Surface

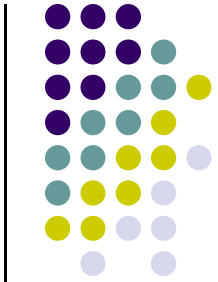


- In some MPI implementations there are more than 300 MPI functions
  - Not all of them part of the MPI standard though, some vendor specific

MPI_Abort, MPI_Accumulate, MPI_Add_error_class, MPI_Add_error_code, MPI_Add_error_string, MPI_Address, MPI_Allgather, MPI_Allgatherv, MPI_Alloc_mem, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Attr_delete, MPI_Attr_get, MPI_Attr_put, MPI_Barrier, MPI_Bcast, MPI_Bsend, MPI_Bsend_init, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Cancel, MPI_Cart_coords, MPI_Cart_create, MPI_Cart_get, MPI_Cart_map, MPI_Cart_rank, MPI_Cart_shift, MPI_Cart_sub, MPI_Cartdim_get, MPI_Comm_call_errhandler, MPI_Comm_compare, MPI_Comm_create, MPI_Comm_create_errhandler, MPI_Comm_create_keyval, MPI_Comm_delete_attr, MPI_Comm_dup, MPI_Comm_free, MPI_Comm_free_keyval, MPI_Comm_get_attr, MPI_Comm_get_errhandler, MPI_Comm_get_name, MPI_Comm_group, MPI_Comm_rank, MPI_Comm_remote_group, MPI_Comm_remote_size, MPI_Comm_set_attr, MPI_Comm_set_errhandler, MPI_Comm_set_name, MPI_Comm_size, MPI_Comm_split, MPI_Comm_test_inter, MPI_Dims_create, MPI_Errhandler_create, MPI_Errhandler_free, MPI_Errhandler_get, MPI_Errhandler_set, MPI_Error_class, MPI_Error_string, MPI_Exscan, MPI_File_call_errhandler, MPI_File_close, MPI_File_create_errhandler, MPI_File_delete, MPI_File_get_amode, MPI_File_get_atomicity, MPI_File_get_byte_offset, MPI_File_get_errhandler, MPI_File_get_group, MPI_File_get_info, MPI_File_get_position, MPI_File_get_position_shared, MPI_File_get_size, MPI_File_get_type_extent, MPI_File_get_view, MPI_File_iread, MPI_File_iread_at, MPI_File_iread_shared, MPI_File_iwrite, MPI_File_iwrite_at, MPI_File_iwrite_shared, MPI_File_open, MPI_File_preallocate, MPI_File_read, MPI_File_read_all, MPI_File_read_all_begin, MPI_File_read_all_end, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_at_all_begin, MPI_File_read_at_all_end, MPI_File_read_ordered, MPI_File_read_ordered_begin, MPI_File_read_ordered_end, MPI_File_read_shared, MPI_File_seek, MPI_File_seek_shared, MPI_File_set_atomicity, MPI_File_set_errhandler, MPI_File_set_info, MPI_File_set_size, MPI_File_set_view, MPI_File_sync, MPI_File_write, MPI_File_write_all, MPI_File_write_all_begin, MPI_File_write_all_end, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_at_all_begin, MPI_File_write_at_all_end, MPI_File_write_ordered, MPI_File_write_ordered_begin, MPI_File_write_ordered_end, MPI_File_write_shared, MPI_Finalize, MPI_Finalized, MPI_Free_mem, MPI_Gather, MPI_Gatherv, MPI_Get, MPI_Get_address, MPI_Get_count, MPI_Get_elements, MPI_Get_processor_name, MPI_Get_version, MPI_Graph_create, MPI_Graph_get, MPI_Graph_map, MPI_Graph_neighbors, MPI_Graph_neighbors_count, MPI_Graphdims_get, MPI_Grequest_complete, MPI_Grequest_start, MPI_Group_compare, MPI_Group_difference, MPI_Group_excl, MPI_Group_free, MPI_Group_incl, MPI_Group_intersection, MPI_Group_range_excl, MPI_Group_range_incl, MPI_Group_rank, MPI_Group_size, MPI_Group_translate_ranks, MPI_Group_union, MPI_Ibsend, MPI_Info_create, MPI_Info_delete, MPI_Info_dup, MPI_Info_free, MPI_Info_get, MPI_Info_get_nkeys, MPI_Info_get_nthkey, MPI_Info_get_valuelen, MPI_Info_set, MPI_Init, MPI_Init_thread, MPI_Initialized, MPI_Intercomm_create, MPI_Intercomm_merge, MPI_Iprobe, MPI_Irecv, MPI_Irsend, MPI_Is_thread_main, MPI_Isend, MPI_Issend, MPI_Keyval_create, MPI_Keyval_free, MPI_Op_create, MPI_Op_free, MPI_Pack, MPI_Pack_external, MPI_Pack_external_size, MPI_Pack_size, MPI_Pcontrol, MPI_Probe, MPI_Put, MPI_Query_thread, MPI_Recv, MPI_Recv_init, MPI_Reduce, MPI_Reduce_scatter, MPI_Register_datarep, MPI_Request_free, MPI_Request_get_status, MPI_Rsend, MPI_Rsend_init, MPI_Scan, MPI_Scatter, MPI_Scatterv, MPI_Send, MPI_Send_init, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend, MPI_Ssend_init, MPI_Start, MPI_Startall, MPI_Status_set_cancelled, MPI_Status_set_elements, MPI_Test, MPI_Test_cancelled, MPI_Testall, MPI_Testany, MPI_Testsome, MPI_Topo_test, MPI_Type_commit, MPI_Type_contiguous, MPI_Type_create_darray, MPI_Type_create_f90_complex, MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_hindexed, MPI_Type_create_hvector, MPI_Type_create_indexed_block, MPI_Type_create_keyval, MPI_Type_create_resized, MPI_Type_create_struct, MPI_Type_create_subarray, MPI_Type_delete_attr, MPI_Type_dup, MPI_Type_extent, MPI_Type_free, MPI_Type_free_keyval, MPI_Type_get_attr, MPI_Type_get_contents, MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_get_name, MPI_Type_get_true_extent, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_match_size, MPI_Type_set_attr, MPI_Type_set_name, MPI_Type_size, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, MPI_Unpack, MPI_Unpack_external, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome, MPI_Win_call_errhandler, MPI_Win_complete, MPI_Win_create, MPI_Win_create_errhandler, MPI_Win_create_keyval, MPI_Win_delete_attr, MPI_Win_fence, MPI_Win_free, MPI_Win_free_keyval, MPI_Win_get_attr, MPI_Win_get_errhandler, MPI_Win_get_group, MPI_Win_get_name, MPI_Win_lock, MPI_Win_post, MPI_Win_set_attr, MPI_Win_set_errhandler, MPI_Win_set_name, MPI_Win_start, MPI_Win_test, MPI_Win_unlock, MPI_Win_wait, MPI_Wtick, MPI_Wtime

- Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code…
  - MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize

The PETSc Library
[The message: Use libraries if available]

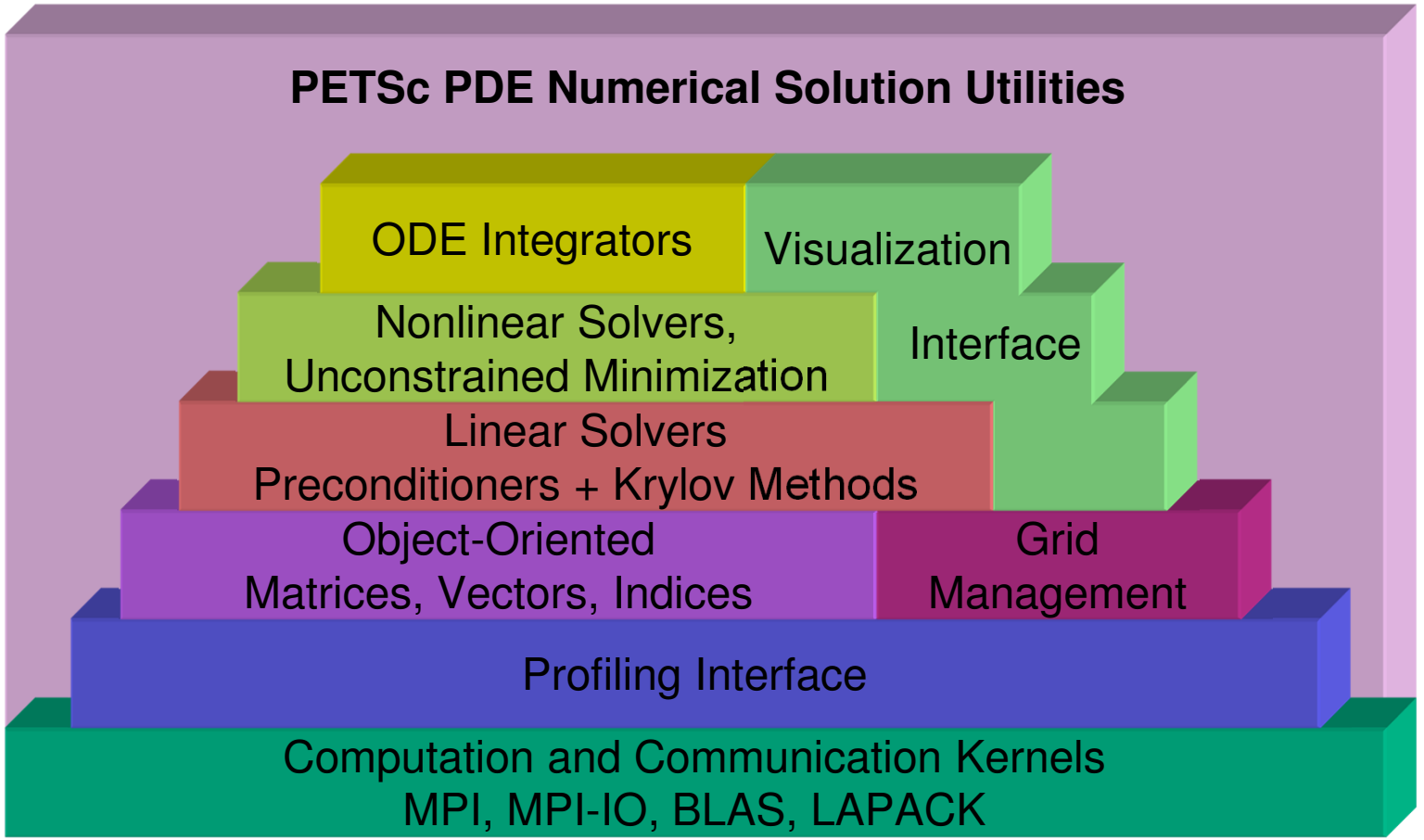


- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)
  - Open source, available for download at http://www.mcs.anl.gov/petsc/petsc-as/

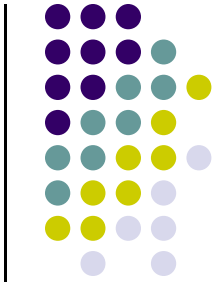
- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)

Structure of PETSc



PETSc Numerical Components



Nonlinear Solvers		
Newton-based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi-CG-STAB	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwartz	Block Jacobi	Jacobi	ILU	ICC	LU (Sequential only)	Others

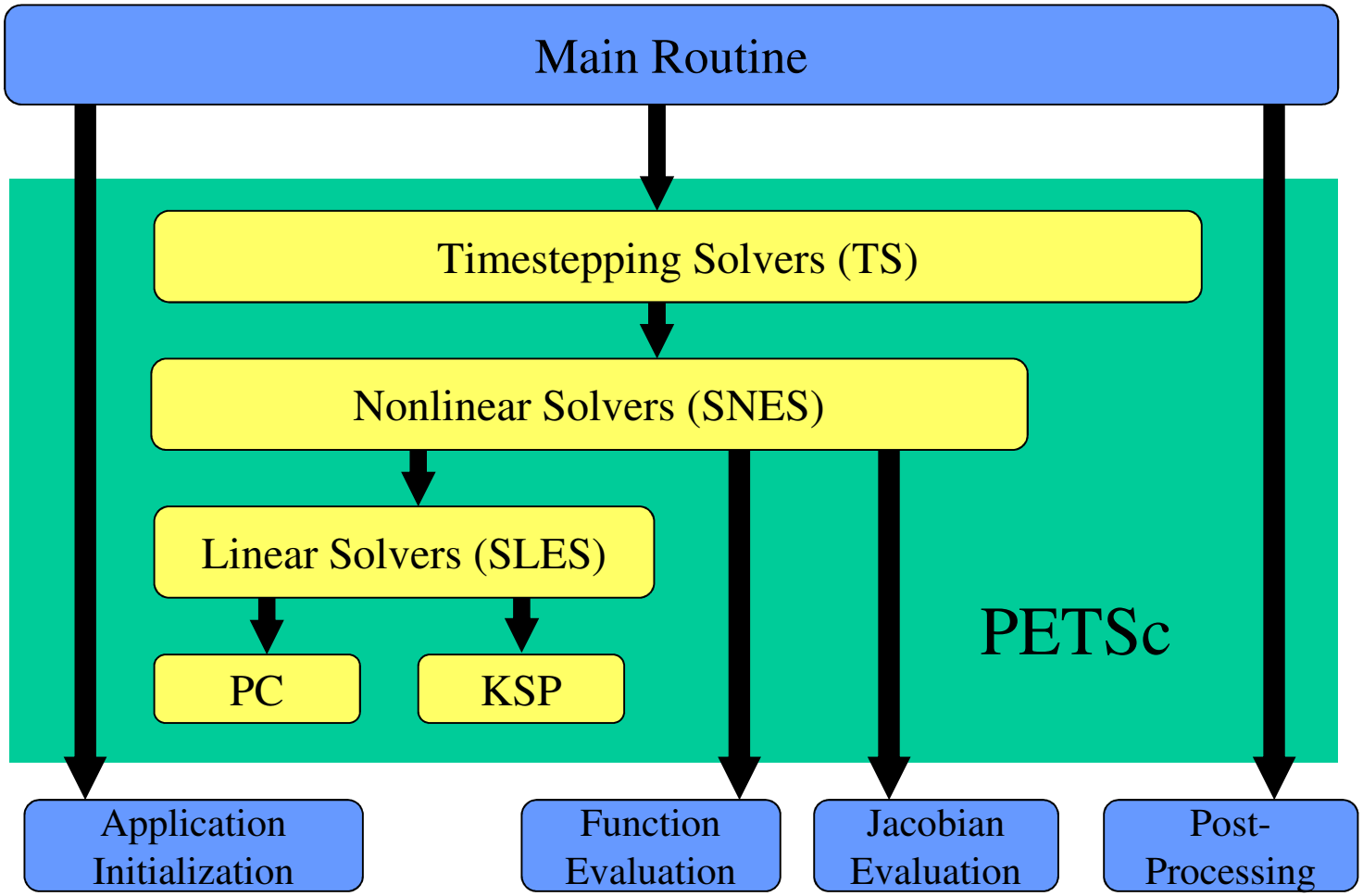
Matrices					
Compressed Sparse Row (AIJ)	Blocked Compressed Sparse Row (BAIJ)	Block Diagonal (BDIAG)	Dense	Matrix-free	Other

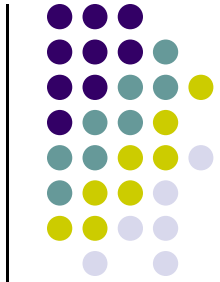
Distributed Arrays

Index Sets			
Indices	Block Indices	Stride	Other

Vectors

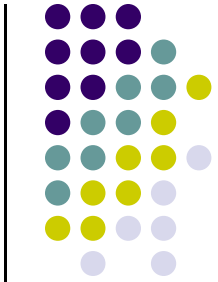
Flow Control for PDE Solution





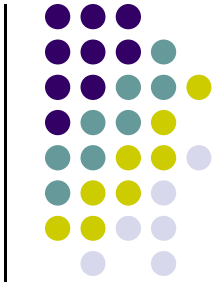
CUDA, OpenMP, MPI:
Putting Things in Perspective

Pros, CUDA



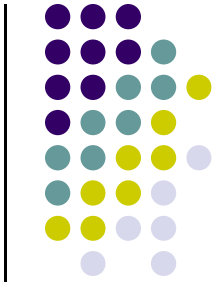
- Many remarkable success stories when the application targeted is data parallel and with high arithmetic intensity
  - One order of magnitude speed-ups are common
- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L got you six or seven years ago
- Ubiquitous
  - Present on more than 100 million computers today support CUDA
- Good productivity tools

Cons, CUDA



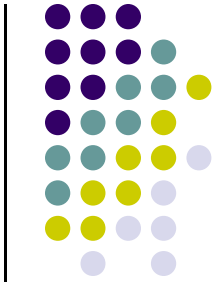
- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware
- Not that much device memory available – 6 GB is the most you get today
  - Getting around it requires moving data in and out of the device, which complicates the programming job
- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance and complicating the implementation task
- For true HPC, using CUDA in conjunction with MPI remains a challenge
  - Ongoing projects aimed at addressing this, but still…

What Would Be Nice…



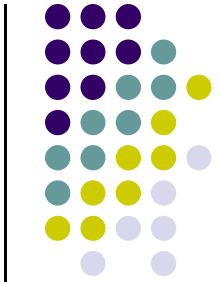
- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases
- Integrate CPU & GPU so that concept of global device memory disappears
- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming

Pros of OpenMP

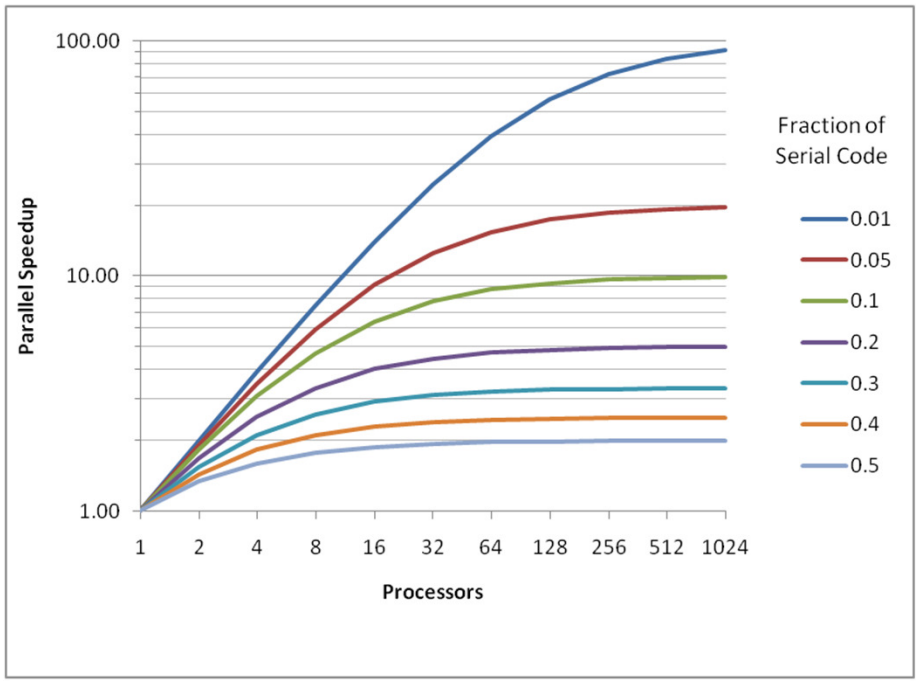


- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement
- Programming model is “serial-like”, thus conceptually simpler than message passing
- Compiler directives are generally simple and easy to use
- Legacy serial code does not need to be rewritten

Cons of OpenMP



- The model doesn't scale up all that well
- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl's Law prohibits substantial speedups



- Amdahl's Law:

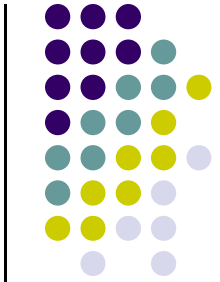
  s = Fraction of serial execution time that cannot be parallelized

  N = Number of processors

$$\text{Execution speedup: } = \frac{1}{s + \frac{1-s}{N}}$$

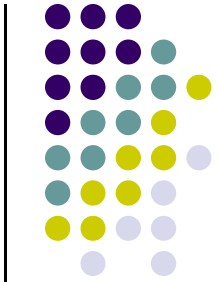
- If you have big loops that dominate execution time, these are ideal targets for OpenMP

Pros of MPI



- Good vendor support for the standard
  - It was great that the community converged upon a standard (something that can't be said about GPU computing)
- Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores
- Can be deployed both for distributed as well as shared memory architectures
- Today it is synonym with High Performance Computing
  - Provided a clear and relatively straightforward framework for reaching Petaflops grade computing

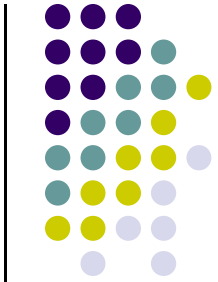
Cons of MPI



- The interconnect is Achilles' heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically worse though

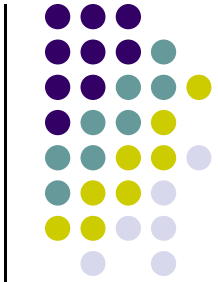
- Like CUDA, works well only for applications where you don't have to communicate all that much (high arithmetic intensity)

General Remarks on Parallel Computing



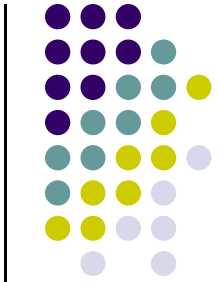
- Parallel Computing is and will be relevant at least for this decade
- Nonetheless, it continues to be challenging
  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it's a skill that is eventually acquired
    - Parallel Programming more difficult than programming for Sequential Computing
  - Productivity tools (debuggers, profilers, build solutions) more challenging to master
  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run

Skills I hope You Picked Up in ME759



- I think of these as items that you can add to your resume:
  - Basic understanding of hardware for parallel computing
  - Basic understanding of parallel execution models: SIMD, MIMD, etc.
  - CUDA programming
  - OpenMP Programming
  - MPI Programming
  - [Build management: **`Cmake`**]
  - Debugging: **`gdb`**, **`cuda-gdb`**, **`memcheck`**, **`cuda-memcheck`**
  - Profiling: **`nvvp`**

ME759:
Most Important Two Things



- Don’t move data around
  - Costly in terms of time and energy.

- Hone your “computational thinking” skills