What We Covered Yesterday...

- GPU computing w/ Thrust
- Parallel computing with OpenMP
  - Parallel regions
  - Work sharing
    - Parallel for
    - Parallel sections
    - Parallel tasks
  - Variable scoping
Today…

- Wrap up OpenMP
- Parallel computing with the Message Passing Interface (MPI)
OpenMP Performance Issues
Performance

- Easy to write OpenMP yet hard to write an efficient program

- Five main causes of poor performance:
  - Problem and/or implementation is sequential in nature
  - Too much data communication
    - False sharing
    - Processor affinity
  - Load imbalance
  - Synchronisation

Credit: Alan Real
Sequential Code

- Corollary to Amdahl’s law: Parallel code will never run faster than the parts which can only be executed in serial.

- Go back and understand whether you can approach the solution from a different perspective that exposes more parallelism.

- Thinking within the context of OpenMP
  - All code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential
  - This code should be as small as possible.
Communication

- On shared memory machines, which is where OpenMP operates, communication is “disguised” as increased memory access costs.
  - It takes longer to access data in main memory or another processor’s cache than it does from local cache.

- Memory accesses are expensive
  - ~100 cycles for a main memory access compared to 1-3 cycles for a flop.

- Unlike message passing, communication is spread throughout the program
  - Hard to analyse and monitor

Credit: Alan Real
Caches and Coherency

- Shared memory programming assumes that a shared variable has a unique value at a given time.

- Caching means that copies of a memory location may exist in multiple places.

- To avoid two processors caching different values of the same memory location, caches must be kept coherent.

- Coherency operations are usually performed on the cache lines in the level of cache closest to memory.
  - Level 3 cache on most systems these days.

Credit: Alan Real
What Does MESI Mean to You?
**MESI: Invalidation-Based Coherence Protocol**

- Cache lines have state bits.
- Data migrates between processor caches, state transitions maintain coherence.

**MESI** Protocol has four states: M: Modified, E: Exclusive, S: Shared, I: Invalid

1. Read “x”
   - Processor A’s Cache
     - I → E
     - “exclusive”
     - E → S
     - “shared”
   - Processor B’s Cache
     - I

2. Read “x”
   - Processor A’s Cache
     - E
   - Processor B’s Cache
     - I → S
     - “shared”

3. Write “x”
   - Processor A’s Cache
     - S → M
     - “modified/dirty”
   - Processor B’s Cache
     - S → I
     - “invalid”

Cache line was invalidated

[J. Marathe]→
Further simplify MESI to provide a simple illustration on next slide
  - Assume now that each cache line can exist in one of 3 states:
    - Exclusive: the only valid copy in any cache
    - Read-only: a valid copy but other caches may contain it
    - Invalid: out of date and cannot be used

Example of next slide a crude approximation of the complex reality
Important Concept: False Sharing

- Cache lines consist of several words of data
  - For instance, one cache line can store 8 double precision values

- What happens when two processors are both writing to different words on the same cache line?
  - Each write will invalidate the other processors copy
  - Lots of remote memory accesses

- Symptoms:
  - Poor speedup
  - High, non-deterministic numbers of cache misses
  - Mild, non-deterministic, unexpected load imbalance

Credit: Alan Real
double sum = 0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

#pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

#pragma omp atomic
    sum += sum_local[me];
}
Sometimes This Fixes It

- Reduce the frequency of false sharing by using thread-local copies of data.
  - The thread-local copy read and modified frequently
  - When complete, copy the result back to the data structure.

```
struct ThreadParams {
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; //Frequent read/write access variable
    unsigned long start;
    unsigned long end;
};

void threadFunc(void *parameter) {
    ThreadParams *p = (ThreadParams*)parameter;
    // local copy for read/write access variable
    unsigned long local_v = p->v;

    for (unsigned long local_dummy = p->start; local_dummy < p->end; local_dummy++) {
        // Functional computation, used to read/write the “v” member.
        // Keep reading/writing local_v instead
    }

    p->v = local_v; // Update shared data structure only once
}
```
Another Way to Fix This
[Kind of Ugly + Architecture Dependent]

- When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary.
- If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line.

```c
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;

    // expand to 64 bytes to avoid false-sharing
    // (4 unsigned long variables + 12 padding)*4 = 64
    int padding[12];
};
__declspec(align(64)) struct ThreadParams Array[10];
```

Credit: Intel
Data Affinity

- Data is cached on the processors which access it
  - Must reuse cached data as much as possible.

- Write code with good *data affinity*:
  - Ensure the same thread accesses the same subset of program data as much as possible

- Danger: you might solve this problem but instead introduce false sharing

Credit: Alan Real
Data affinity example

!$OMP DO PRIVATE(i)
do j=1,n
  do i=1,n
    a(i,j)= i+j
  end do
end do
!$OMP DO SCHEDULE(STATIC,16)
!$OMP& PRIVATE(i)
do j=1,n
  do i=1,j
    b(j)=b(j)+a(i,j)
  end do
end do

1st 16 j’s OK – rest are cache misses!

Credit: Alan Real
Load Imbalance

- Load imbalance can arise from both communication and computation

- Worth experimenting with different scheduling options
  - Static, dynamic, guided
  - Experiment with the chunk size
Synchronisation

- Barriers can be very expensive
  - Typically 1000s cycles to synchronise 40 threads

- Avoid barriers via:
  - *Careful* use of the NOWAIT clause
  - Parallelise at the outermost level possible
    - May require re-ordering of loops +/- indexes
  - Choice of CRITICAL / ATOMIC / lock routines may impact performance

Credit: Alan Real
Attractive Features of OpenMP

- Freedom to parallelize small parts of application, one hot-spot at a time
  - Can start with most time-critical parts and go from there

- Can implement complex algorithms

- Code size grows only modestly

- Expression of parallelism flows clearly, code is easy to read

- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - C/C++: http://openmp.org/mp-documents/OpenMP-4.0-C.pdf
End of OpenMP
Beginning of MPI
Acknowledgments

- Parts of MPI material covered draws on a set of slides made available by the Irish Centre for High-End Computing (ICHEC) - [www.ichec.ie](http://www.ichec.ie)
  - These slides will contain “ICHEC” at the bottom
  - In turn, the ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh

- Individual or institutions are acknowledged at the bottom of the slide, like
  
  [A. Jacobs]→
MPI: Textbooks, Further Reading…

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 - very good introduction.
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC
Memory shared among processors
- Typical scenario, on a budget: one node with four CPUs, each with 16 cores
- Typically one node systems; i.e., a workstation or a laptop

Example:
- Symmetric Multi-Processors (SMP) – like one node of Euler
  - Each processor has equal access to RAM

Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing

Two issues:
- Scales poorly with system size due to the need for cache coherence
- Challenging applications might need more memory than available on the typical multi-core node
Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS

- Information is passed between nodes using the network

- No cache coherence and no need for special cache coherency hardware
  - Parallelism accomplished by interplay between different *processes* and not threads

- Software development: more difficult to write programs for distributed memory systems since the programmer must keep track of memory usage

- Traditionally, this represents the hardware setup for big-iron computing
Overview of Large Multiprocessor Hardware Configurations

- Larger multiprocessors
  - Shared address space
    - Symmetric shared memory (SMP)
      - Examples: IBM eserver, SUN Sunfire
  - Distributed shared memory (DSM)
  - Commodity clusters: Beowulf and others
  - Custom cluster
    - Cache coherent: ccNUMA
      - SGI Origin/Altix
    - Non-cache coherent: Cray T3E, X1
    - Uniform cluster: IBM BlueGene
      - Constellation cluster of DSMs or SMPs
        - SGI Altix, ASC Purple

© 2007 Elsevier Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessy and Patterson, fourth edition
Euler
~ Hardware Configurations ~
Hardware Relevant in the Context of MPI
Two Components of Euler that are Important

- **CPU**: AMD Opteron 6274 Interlagos 2.2GHz
  - 16-Core Processor (four CPUs per node → 64 cores/node)
  - 8 x 2MB L2 Cache per CPU
  - 2 x 8MB L3 Cache per CPU
  - Thermal Design Power (TDP): 115W

- **HCA**: 40Gbps Mellanox Infiniband interconnect
  - Bandwidth comparable to PCIe2.0 x16 (~32Gbps), yet the latency is rather poor (~1microsecond)
  - Ends up being the bottleneck in cluster computing

- Quick remark: 100Gbps is available, 200 Gbps in 2016
N instances of the same program are launched for execution independently as N distinct processes
- Cache coherence here is a nonissue

Each process, which runs most often on a core, executes the same executable at roughly the same time
- Synchronization calls exist, useful to coordinate the execution of the MPI job

What differentiates processes is their rank: processes with different ranks do different things (“branching based on the process rank”)
- Very similar to GPU computing, where one thread did work based on its thread index
- Very similar to OpenMP function omp_get_thread_num()
The Message-Passing Model

- Each process has its own program counter and address space
- Message passing enables communication among processes that have separate address spaces
- Inter-process communication typically consists of
  - Synchronization, followed by…
  - … movement of data from one process’s address space to another’s
- Execution paradigm embraced in MPI: Single Program Multiple Data (SPMD)
The Message-Passing Programming Paradigm

- Sequential Programming Paradigm

  ![Sequential Programming Diagram](data-program-processor-memory)

- Message-Passing Programming Paradigm

  ![Message-Passing Programming Diagram](data-program-processor-network)

Distributed memory
Parallel processors
communication network
A process is a **software program** executing a task on a **processor**

In other words, each process in a message passing parallel job runs an instance of a **program**:
- Program written in a conventional sequential language, e.g., C, C++, Fortran
- The variables of each program have the same name but different locations (distributed memory) and **different data**
- Communicate via special send & receive routines (**message passing**)
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

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

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    }
    else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize();
    return 0;
}
Program Output

```
[negrut@euler04 CodeBits]$ mpiexec -np 8 ./a.out
I am a worker: euler04 (rank=1/7)
I am a worker: euler04 (rank=5/7)
I am a worker: euler04 (rank=6/7)
I am a worker: euler04 (rank=3/7)
I am a worker: euler04 (rank=4/7)
I am the master: euler04
I am a worker: euler04 (rank=2/7)
I am a worker: euler04 (rank=7/7)
[negrut@euler04 CodeBits]$  
[negrut@euler04 CodeBits]$  
```
Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - Some of these supercomputers might use GPU acceleration though
  - The program executed can also use OpenMP on a node

- Examples of architectures relying on MPI for HPC:
  - IBM Blue Gene L/P/Q (Argonne National Lab – “Mira”)
  - Cray supercomputers (Oakridge National Lab – “Titan”, also uses K20X GPUs)

- Widely used in Scientific Computing, MPI has FORTRAN, C, and C++ bindings
MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- One of the early common implementations: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - OpenMPI: joint effort of three or four groups (Los Alamos, Tennessee, Indiana University, Europe)
Where Can We Use Message Passing?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of Workstations
  - Even on shared memory systems
MPI vs. CUDA

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?
  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems

- In large scale computing the future likely to belong to heterogeneous architecture
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)
Parallel computing on the GPU, on multi-core chips, and multi-node (MPI) share one thing:

- Same code is executed but for different data
- There is some sort of ID that helps a thread/process figure out what data the thread/process needs to work on
  - In CUDA you have threadIdx and blockIdx
  - In OpenMP you have the thread ID
  - In MPI you have a process rank

There are exceptions from the rule (thread divergence, sections, etc.)
MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, newer edition available

```c
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 * Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
MPI, a Third Example: Approximating \( \pi \)

\[
\int_0^1 \frac{4}{1 + x^2} = 4 \cdot \tan^{-1}(1) = \pi
\]

Numerical Integration: Midpoint rule

\[
\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f((i - 0.5) \cdot h)
\]
MPI, a Third Example: Approximating $\pi$

- Use 4 MPI processes (rank 0 through 3)
- In the picture, $n=13$
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: 1,5,9,13
  - Rank 1: 2,6,10
  - Rank 2: 3,7,11
  - Rank 3: 4,8,12
- Each rank computes the area in its associated sub-intervals
- **MPIReduce** is used to sum the areas computed by each rank yielding final approximation to $\pi$
Code for Approximating $\pi$

```cpp
// MPI_PI.cpp : Defines the entry point for the console application.
//
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
}
```
```c
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) ";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        cout << "pi is approximately " << pi <<", Error is " << fabs(pi - PI25DT) << endl;
}
MPI_Finalize();
return 0;
```
Broadcast
[MPI function used in Example]

- A one-to-many communication.
Collective Communications

- Collective communication routines are higher level routines
- Several processes are involved at a time
- May allow optimized internal implementations, e.g., tree based algorithms
  - Require $O(\log(N))$ time as opposed to $O(N)$ for naïve implementation
Reduction Operations

[MPI function used in Example]

- Combine data from several processes to produce a single result
Barriers

[MPI function used in Example]

- Used implicitly or explicitly to synchronize processes
MPI, Practicalities
MPI on Euler
[Selecting MPI Distribution]

- What’s available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler

- To load OpenMPI environment variables:
  - (typically not needed, should be done automatically for you)

  ```bash
  $ module load mpi/gcc/openmpi
  ```
Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
- Adds in `-L`, `-l`, `-I`, etc. flags for MPI
- Passes any options to your native compiler (`gcc`)
- Very similar to what `nvcc` did for CUDA – it’s a compile driver…

```bash
$ mpicxx -o integrate_mpi integrate_mpi.cpp
```
Running MPI Code on Euler

```bash
mpiexec [-np #] [-machinefile file] <program> [<args>]
```

- **Number of processors.** Optional if using a machinefile.
- **List of hostnames to use.** Inside Torque, this file is at `$PBS_NODEFILE`.
- **Your program and its arguments**.

- The machinefile/nodefile is required for multi-node jobs with the version of OpenMPI on Euler.
- `-np` will be set automatically from the machinefile; can select lower, but not higher.
- See the `mpiexec` manpage for more options.
Example

euler $ qsub -I -l nodes=8:ppn=4:amd,walltime=5:00
qsub: waiting for job 15246.euler to start
qsub: job 15246.euler ready

euler07 $ cd $PBS_O_WORKDIR
euler07 $ mpiexec -machinefile $PBS_NODEFILE ./integrate_mpi
32 32.121040666358297 in 0.998202s

euler07 $ mpiexec -np 16 -machinefile $PBS_NODEFILE ./integrate_mpi
16 32.121040666359455 in 1.524001s

euler07 $ mpiexec -np 8  -machinefile $PBS_NODEFILE ./integrate_mpi
8 32.121040666359136 in 2.171963s

euler07 $ mpiexec -np 4  -machinefile $PBS_NODEFILE ./integrate_mpi
4 32.121040666360585 in 4.600204s

euler07 $ mpiexec -np 2  -machinefile $PBS_NODEFILE ./integrate_mpi
2 32.121040666366788 in 7.615060s

euler07 $ ./integrate_mpi
1 32.121040666353437 in 15.163330s
MPI Nuts and Bolts
The Rank & The Communicator
[As Facilitators for Data and Work Distribution]

- To communicate with each other MPI processes need identifiers: \textit{rank} = \textit{identifying number}

- Work distribution decisions are based on the \textit{rank}
  - Helps establish which process works on which data
  - Just like we had thread and block indices in CUDA

\begin{itemize}
  \item \texttt{myrank=0} \quad \texttt{data} \quad \texttt{program}
  \item \texttt{myrank=1} \quad \texttt{data} \quad \texttt{program}
  \item \texttt{myrank=2} \quad \texttt{data} \quad \texttt{program}
  \item \texttt{myrank=(size-1)} \quad \texttt{data} \quad \texttt{program}
\end{itemize}
Message Passing

- Messages are packets of data moving between different processes
- Necessary information for the message passing system:
  - sending process + receiving process \{ i.e., the two “ranks” \}
  - source location + destination location
  - source data type + destination data type
  - source data size + destination buffer size
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
Communicator  MPI_COMM_WORLD

- All processes of an MPI program are members of the default communicator MPI_COMM_WORLD

- MPI_COMM_WORLD is a predefined handle in mpi.h

- Each process has its own rank in a given communicator:
  - starting with 0
  - ending with (size-1)

- You can define a new communicator in case you find it useful
  - Use MPI_Comm_create call. Example creates the communicator DANS_COMM_WORLD

```c
MPI_Comm_create(MPI_COMM_WORLD, new_group, &DANS_COMM_WORLD);
```
MPI_Comm_create

- Synopsis

```c
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
```

- Input Parameters
  - `comm` - communicator (handle)
  - `group` - subset of the family of processes making up the `comm` (handle)

- Output Parameter
  - `newcomm` - new communicator (handle)
Point-to-Point Communication

- Simplest form of message passing

- One process sends a message to another process
  - `MPI_Send`
  - `MPI_Recv`

- Sends and receives can be
  - Blocking
  - Non-blocking
  - More on this shortly
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., DANS_COMM_WORLD
- Processes are identified by their ranks in the communicator
The Data Type

- A message contains a number of elements of some particular data type

- MPI data types:
  - Basic data type
  - Derived data types – more on this later

- Data type handles are used to describe the type of the data moved around

Example: message with 5 integers

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>654</td>
<td>96574</td>
<td>-12</td>
</tr>
<tr>
<td>MPI Datatype</td>
<td>C datatype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
count=5
int arr[5]
datatype=MPI_INT
```

[ICHEC]→
MPI_Send & MPI_Recv: The Eager and Rendezvous Flavors

- If you send small messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “eager mode”

- If you send a large amount of data, the sender function waits for the receiver to post a receive before sending the actual data of the message

- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send a small amount of data, the MPI implementation can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play though this trick if you attempt to move around a huge chunk of data
NOTE: Each implementation of MPI has a default value (which might change at run time) beyond which a larger MPI_Send stops acting “eager”
- The MPI standard doesn’t provide specifics
- You don’t know how large is too large…

Does it matter if it’s Eager or Rendezvous?
- In fact it does, sometimes the code can hang – example to come

Remark: In the message-passing paradigm for parallel programming you’ll always have to deal with the fact that the data that you send needs to “live” somewhere during the send-receive transaction
**MPI_Send & MPI_Recv: Blocking vs. Non-blocking**

- Moving away from the Eager vs. Rendezvous modes → they only concern the MPI_Send and MPI_Recv pair

- Messages can be sent with other vehicles than plain vanilla MPI_Send

- The class of send-receive operations can be classified based on whether they are blocking or non-blocking
  - Blocking send: upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since a copy of the data has been sent
  - Non-blocking: the send call returns immediately and there is no guarantee that the data has actually been transmitted upon return from send call
    - Take home message: before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed
Example: Send & Receive

Blocking Alternative: MPI_Send

- Several other blocking flavors exist, to be discussed later

- The problem with plain vanilla:
  - 1: when sending large messages, there is no overlap of compute & data movement
    - This is what we strived for when using "streams" in CUDA
  - 2: if not done properly, the processes executing the MPI code can hang

- There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above
Example: Send & Receive

Non-blocking Alternative: MPI_Isend

- If non-blocking, the data “lives” in your buffer – that’s why it’s not safe to change it since you don’t know when transaction was closed
  - This typically realized through a MPI_Isend
    - “I” stands for “immediate”

- NOTE: there is another way for providing a buffer region but this alternative is blocking
  - Realized through MPI_Bsend
    - “B” stands for “buffered”
  - The problem here is that *you* need to provide this additional buffer that stages the transfer
    - Interesting question: how large should *that* staging buffer be?
  - Adding another twist to the story: if you keep posting non-blocking sends that are not matched by corresponding “MPI_Recv” operations, you are going to overflow this staging buffer
The Mechanics of P2P Communication: Sending a Message

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- **buf** is the starting point of the message with **count** elements, each described with **datatype**

- **dest** is the rank of the destination process within the communicator **comm**

- **tag** is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The **tag** can be used to distinguish between different messages
  - Rarely used, typically for situations when things go wrong
The Mechanics of P2P Communication: Receiving a Message

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status *status)
```

- **buf/count/datatype** describe the receive buffer
- Receiving the message sent by process with rank **source** in **comm**
- Only messages with matching **tag** are received
- Envelope information is returned in the **MPI_Status** object **status**
MPI_Recv:
The Need for an MPI_Status Argument

- The MPI_Status object returned by the call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound
  - If multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation
The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – `source = MPI_ANY_SOURCE`
  - To receive from any tag – `tag = MPI_ANY_TAG`
  - Actual source and tag returned in receiver's `status` argument
The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI_RECV in status.

- status.MPI_SOURCE
- status.MPI_TAG
- count via MPI_Get_count()

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
```

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message data types must match
- Receiver’s buffer must be large enough
Blocking Type:
Communication Modes

- Send communication modes:
  - Synchronous send → MPI_SSEND
  - Buffered [asynchronous] send → MPI_BSEND
  - Standard send → MPI_SEND
  - Ready send → MPI_RSEND

- Receiving all modes → MPI_RECV
1) Synchronous Sending in MPI
2) Buffered Sending in MPI

- **Synchronous with MPI_Ssend**
  - In synchronous mode, a send will not complete until a matching receive is posted.
  - The sender has to wait for a receive to be posted
  - No buffering of data
  - Used for ensuring the code is healthy and doesn’t rely on buffering

- **Buffered with MPI_Bsend**
  - Send completes once message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet
3) Standard Sending in MPI
4) Ready Sending in MPI

- Standard with MPI_Send
  - Up to the MPI implementation to decide whether to do rendezvous or eager, for performance reasons
    - NOTE: If it does rendezvous, in fact the behavior is that of MPI_SSend
  - Very commonly used

- Ready with MPI_Rsend
  - Will work correctly *only* if the matching receive has been posted
  - Can be used to avoid handshake overhead when program is known to meet this condition
  - Rarely used, can cause major problems
## Cheat Sheet, Blocking Options

<table>
<thead>
<tr>
<th>Sender modes</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_SSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td>needs application-defined buffer to be declared with <strong>MPI_BUFFER_ATTACH</strong></td>
</tr>
<tr>
<td><strong>MPI_BSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classic</td>
<td>Standard send</td>
<td>Rendezvous or eager mode. Decided at run time</td>
</tr>
<tr>
<td><strong>MPI_SEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>Started right away. Will work out <strong>only</strong> if the matching receive is already posted!</td>
<td>Blindly do a send. Avoid, might cause unforeseen problems...</td>
</tr>
<tr>
<td><strong>MPI_RSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a the message (data) has arrived</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_RECV</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Most Important Issue: Deadlocking

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

Process 0

MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...

No Deadlock

Process 1

MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...

Deadlocking, Another Example

- MPI_Send can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data size > 127999 bytes
Data size < 128000 bytes

Deadlock

No Deadlock
Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair `MPI_Ssend` and `MPI_Recv` operations the right way:

```
Process 0
...
MPI_Ssend()
MPI_Recv()
...
```

```
Process 1
...
MPI_Recv()
MPI_Ssend()
...
```

- Conclusion: understand how the implementation works and what its pitfalls/limitations are
Example

- Always succeeds, even if no buffering is done

```c
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Will always deadlock, no matter the buffering mode

```c
if(rank==0)
{
    MPI_Recv(...);
    MPI_Send(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Only succeeds if message in at least one of the transactions is small enough and an “eager” mode is triggered

```c
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Send(...);
    MPI_Recv(...);
}
```
Concluding Remarks, Blocking Options

- Standard send (**MPI_SEND**)
  - minimal transfer time if lands in eager mode
  - blocks in rendezvous, switches to synchronous mode

- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting → idle time
  - high latency / best bandwidth

- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth

- Ready send (**MPI_RSEND**)
  - use **never**, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code
Technicalities, Loose Ends: More on the Buffered Send [1/2]

- Relies on the existence of a buffer, which is set up through a call:
  ```c
  int MPI_Buffer_attach(void* buffer, int size);
  ```

- A bsend is a **local** operation. It does not depend on the occurrence of a matching receive in order to complete.

- If a bsend operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete.

- Return from an `MPI_Bsend` does not guarantee the message was sent.

- Message remains in the buffer until a matching receive is posted.
Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space.

The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient.

MPI_Bsend lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data.

The MPI_Buffer_attach subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a process at any time.
Technicalities, Loose Ends: Message Order Preservation

- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
  - Messages do not overtake each other
  - True even for non-synchronous sends

- If both receives match both messages, then the order is preserved
Timing an MPI Job

```
int main()
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ....
    endtime = MPI_Wtime();
    printf("That took \%f seconds\n", endtime - starttime);
    return 0;
}
```

- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`
Non-Blocking Communication
Non-Blocking Communications: Motivation

- Overlap communication with execution (just like with CUDA):
  1. Initiate non-blocking communication
     - Returns Immediately
     - Routine name starting with MPI_...
  2. Do some other useful work
     - “latency hiding”
  3. Wait for non-blocking communication to complete
Non-blocking Send/Receive

- Syntax

```c
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

- `buf` - [in] initial address of send buffer (choice)
- `count` - [in] number of elements in send buffer (integer)
- `datatype` - [in] datatype of each send buffer element (handle)
- `dest` - [in] rank of destination (integer)
- `tag` - [in] message tag (integer)
- `comm` - [in] communicator (handle)
- `request` - [out] communication request (handle)

```c
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);
```
The Screenplay: Non-Blocking P2P Communication

- Non-blocking send

  MPI_Isend(...)  
  doing some other work  
  MPI_Wait(...)  

- Non-blocking receive

  MPI_Irecv(...)  
  doing some other work  
  MPI_Wait(...)  

/// = waiting until operation locally completed
Non-Blocking Send/Receive
Some Tools of the Trade

- Call returns immediately. Therefore, user must worry whether …
  - Data to be sent is out of the send buffer before trampling on the buffer
  - Data to be received has finished arriving before using the content of the buffer

- Tools that come in handy:
  - For sends and receives in flight
    - `MPI_Wait` – blocking - you go synchronous
    - `MPI_Test` – non-blocking - returns quickly with status information

- Check for existence of data to receive
  - Blocking: `MPI_Probe`
  - Non-blocking: `MPI_Iprobe`
Waiting for isend/ireceive to Complete

- Waiting on a single send
  ```c
  int MPI_Wait(MPI_Request *request, MPI_Status *status);
  ```

- Waiting on multiple sends (get status of all)
  - Till all complete, as a barrier
    ```c
    int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
    ```
  - Till at least one completes
    ```c
    int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status);
    ```
  - Helps manage progressive completions
    ```c
    int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses);
    ```
MPI_Test...

- Flag true means completed
  ```c
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
  ```

  ```c
  int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);
  ```

  ```c
  int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag,
                  MPI_Status *status);
  ```

- Like a non blocking MPI_Waitsome
  ```c
  int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices,
                   MPI_Status *statuses);
  ```
The Need for MPI_Probe and MPI_Iprobe

- The **MPI_PROBE** and **MPI_IPROBE** operations allow incoming messages to be checked for, without actually receiving them.

- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status).

- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.
Probe to Receive

- Probe yields incoming size

- Blocking Probe, wait till match
  ```c
  int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
  ```

- Non Blocking Probe, flag true if ready
  ```c
  int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
  ```
Two types of communication:

- Blocking:
  - Safe to change content of buffer holding on to data in the MPI send call
- Non-blocking:
  - Be careful with the data in the buffer, since you might step on/use it too soon

MPI provides four modes for these two types

- standard, synchronous, buffered, ready
Collective Actions
Collective Actions

- MPI actions involving a group of processes

- Must be called by all processes in a communicator

- All collective actions are blocking

- Types of Collective Actions (three of them):
  - Global Synchronization (barrier synchronization)
  - Global Communication (broadcast, scatter, gather, etc.)
  - Global Operations (sum, global maximum, etc.)
Barrier Synchronization

- Syntax:

  ```c
  int MPI_Barrier(MPI_Comm comm);
  ```

- `MPI_Barrier` not needed that often:
  - All synchronization is done automatically by the data communication
    - A process cannot continue before it has the data that it needs
  - If used for debugging
    - Remember to remove for production release
Communication Action: Broadcast

- Function prototype:

```c
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

![Diagram showing broadcast communication]

- Rank of the sending process (i.e., root process)
- Must be given identically by all processes
MPI_Bcast

A₀: any chunk of contiguous data described with MPI_Datatype and count
MPI_Bcast

```c
int MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
```

- **INOUT** : `buffer` (starting address, as usual)
- **IN** : `count` (number of entries in buffer)
- **IN** : `type` (can be user-defined)
- **IN** : `root` (rank of broadcast root)
- **IN** : `com` (communicator)

- Broadcasts message from `root` to all processes (including `root`)
- `com` and `root` must be identical on all processes
- On return, contents of `buffer` is copied to all processes in `com`
Example: MPI_Bcast

- Read a parameter file on a single processor and send data to all processes

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

int main(int argc, char **argv){
    int myRank, nprocs;
    float data = -1.0;
    FILE *file;

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

    if( myRank==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file ! = NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: \%f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: \%f\n", data);

    MPI_Finalize();
}
```
Example: MPI_Bcast

[Output]

[negrut@euler CodeBits]$ qsub -I -l nodes=8:ppn=4,walltime=5:00
qsub: waiting for job 16114.euler to start
qsub: job 16114.euler ready

[negrut@euler17 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler17 CodeBits]$ mpiexec -np 4 a.out

data before: -1.000000
data before: -1.000000
data before: -1.000000
data before: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
Communication Action: Gather

- Function Prototype

```c
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

e.g., root=1

Before gather:
- A
- B
- C
- D
- E

After gather:
- A
- B
- C
- D
- E

A → B → C → D → E
MPI_Gather

<table>
<thead>
<tr>
<th>processes</th>
<th>data</th>
<th>Gather</th>
<th>processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_0$</td>
<td></td>
<td></td>
<td>$A_0$</td>
</tr>
<tr>
<td>$A_1$</td>
<td></td>
<td></td>
<td>$A_1$</td>
</tr>
<tr>
<td>$A_2$</td>
<td></td>
<td></td>
<td>$A_2$</td>
</tr>
<tr>
<td>$A_3$</td>
<td></td>
<td></td>
<td>$A_3$</td>
</tr>
<tr>
<td>$A_4$</td>
<td></td>
<td></td>
<td>$A_4$</td>
</tr>
<tr>
<td>$A_5$</td>
<td></td>
<td></td>
<td>$A_5$</td>
</tr>
</tbody>
</table>
MPI_Gather

```c
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** `sendbuf`  (starting address of send buffer)
- **IN** `sendcount`  (number of elements in send buffer)
- **IN** `sendtype`  (type)
- **OUT** `recvbuf`  (address of receive buffer)
- **IN** `recvcount`  (n-elements for any single receive)
- **IN** `recvtype`  (data type of recv buffer elements)
- **IN** `root`  (rank of receiving process)
- **IN** `comm`  (communicator)
**MPI_Gather**

- Each process sends content of send buffer to the root process
- Root receives and stores in rank order

**Remarks:**
- Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
- `recvcount` on root indicates number of items received from each process, not total. This is a very common error

- Exercise: Sketch an implementation of `MPI_Gather` using only send and receive operations.
```c
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

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

    /* local array size on each proc = nlcl */
    data_loc = (float *) malloc(nlcl*sizeof(float));

    for (i = 0; i < nlcl; ++i) data_loc[i] = myRank;

    if (myRank == 0)
        data = (float *) malloc(nprocs*sizeof(float)*nlcl);

    MPI_Gather(data_loc, nlcl, MPI_FLOAT, data, nlcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0){
        for (i = 0; i < nlcl*nprocs; ++i){
            printf("%f\n", data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
Communication Action: Scatter

- Function prototype

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

e.g., root=1

before scatter

A
B
C
D
E

after scatter

A
B
C
D
E

[ICHEC]
MPI_Scatter

![Diagram of MPI Scatter]

Data (buffer) is distributed to processes A0 to A5.
MPI_Scatter

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** `sendbuf` (starting address of send buffer)
- **IN** `sendcount` (number of elements **sent to each process**)
- **IN** `sendtype` (type)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `recvcount` (n-elements **in receive buffer**)
- **IN** `recvtype` (data type of receive elements)
- **IN** `root` (rank of sending process)
- **IN** `comm` (communicator)
MPI_Scatter

- Inverse of MPI_Gather

- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter

- Remarks:
  - All arguments are significant on root, while on other processes only recvbuf, recvcount, recvtype, root, and comm are significant
```c
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

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

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ){
            for (int j = 0; j < n_lcl; ++j) printf("%f\n", data_l[j]);
        } 
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

This is interesting. Think what’s happening here...
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$
Putting Things in Perspective…

- Gather: you automatically create a serial array from a distributed one
- Scatter: you automatically create a distributed array from a serial one
Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- \( d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1} \)
  - \( d_i \) = data in process rank \( i \)
    - single variable, or
    - vector
  - \( \circ \) = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

- Floating point rounding may depend on usage of associative law:
  - \( (((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \ldots) \circ d_{s-2}) \circ d_{s-1}) \)
Example of Global Reduction

- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`.
- Assume root=0;

```c
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

- The result is only placed in `resultbuf` at the root process.
# Predefined Reduction Operation Handles

<table>
<thead>
<tr>
<th>Predefined operation handle</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of the maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of the minimum</td>
</tr>
</tbody>
</table>
MPI_Reduce

before MPI_REDUCE

- inbuf
- result

A B C D
E F G H
I J K L
M N O

root=1

after

A B C D
E F G H
I J K L
M N O

AoDoGoJoM

[ICHEC]→
Reduce Operation

Assumption: Rank 0 is the root

<table>
<thead>
<tr>
<th>processes</th>
<th>data (input buffer)</th>
<th>reduce</th>
<th>data (output buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td></td>
<td>A0+A1+A2</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td></td>
<td>B0+B1+B2</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td></td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
</tr>
<tr>
<td>C0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
</tr>
<tr>
<td>C1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
</tr>
<tr>
<td>C2</td>
</tr>
</tbody>
</table>

[A. Siegel]→
MPI_Reduce

```c
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)
MPI_Reduce example

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, 6, \text{MPI\_INT}, \text{MPI\_SUM}, 0, \text{MPI\_COMM\_WORLD})
\]

<table>
<thead>
<tr>
<th></th>
<th>sbuf</th>
<th>rbuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>3 4 2 8 12 1</td>
<td>11 16 20 22 24 18</td>
</tr>
<tr>
<td>P1</td>
<td>5 2 5 1 7 11</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2 4 4 10 4 5</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>1 6 9 3 1 1</td>
<td></td>
</tr>
</tbody>
</table>

11
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

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

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

Credit: Allan Snively
### MPI_Allreduce

<table>
<thead>
<tr>
<th>processes</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td></td>
</tr>
</tbody>
</table>

**Allreduce**

<table>
<thead>
<tr>
<th>data (buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1+A2</td>
</tr>
<tr>
<td>B0+B1+B2</td>
</tr>
<tr>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>

[Data (buffer)]

<table>
<thead>
<tr>
<th>Allreduce</th>
<th>data (buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
</tr>
<tr>
<td>A0+C1+C2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
#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 $0, \ldots, i$ (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

after

ABC DEF GHI JKL MNO

A AoD AoDoG AoDoGoJ AoDoGoJoM

done in parallel
Scan Operation

<table>
<thead>
<tr>
<th>Processes</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>B0</td>
<td>C0</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td></td>
</tr>
</tbody>
</table>

Scan process data (output buffer)

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1</td>
<td>B0+B1</td>
<td>C0+C1</td>
<td></td>
</tr>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
<td></td>
</tr>
</tbody>
</table>
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]→ 6 entries
MPI_Scan

```c
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
#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

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 significant 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 \( 0, \ldots, i-1 \) (inclusive).

- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for **MPI_REDUCE**.


**MPI_Exscan**

```c
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)
#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("
");
        }
        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

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:
  ```c
  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++;
    }
}

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...

```cpp
#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<floatgregate::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 sendrecv 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
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

<table>
<thead>
<tr>
<th>Primitive datatype 0</th>
<th>Displacement of 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive datatype 1</td>
<td>Displacement of 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Primitive datatype n-1</td>
<td>Displacement of n-1</td>
</tr>
</tbody>
</table>
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={\textbf{double}, 0}, \textbf{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 \{(\text{char, 0}), (\text{double, 8})\}? 
  Ans: 16

- Is this a valid type: \{(\text{double, 8}), (\text{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.
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

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

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

- **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

```c
int MPI_Type_size (MPI_Datatype datatype, int *size);
```
Committing Data Types

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

```c
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
Call to \texttt{MPI\_Type\_free} sets the value of an MPI data type to \texttt{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.
MPI_Type_contiguous

```c
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
```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 to send the last column
  ```
  int x[4][8];
  ```

Content of x:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>1006</td>
<td>1007</td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10002</td>
<td>10003</td>
<td>10004</td>
<td>10005</td>
<td>10006</td>
<td>10007</td>
</tr>
</tbody>
</table>

- 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**

- **oldtype**
  - 5 element stride
  - between blocks

- **newtype**
  - 3 elements per block
  - 2 blocks

- **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;
}
```

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

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 \( m \times n \) with \( n_g \) 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?

\[
\text{MPI_Type_vector}(\text{int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype});
\]

\[
\text{MPI_Type_vector}(m, n, n+2*ng, \text{MPIDOUBLE}, &\text{interior});
\]

\[
\text{MPI_Type_commit}(&\text{interior});
\]

\[
\text{MPI_Send}(\text{startPoint}, 1, \text{interior}, \text{dest}, \text{tag}, \text{MPI_COMM_WORLD});
\]

Ans:

\[
\text{MPI_Type_vector}\ (m, n, n+2*ng, \text{MPI_DOUBLE, } &\text{interior});
\]

\[
\text{MPI_Type_commit}\ (&\text{interior});
\]

\[
\text{MPI_Send}\ (\text{startPoint}, 1, \text{interior}, \text{dest}, \text{tag}, \text{MPI_COMM_WORLD});
\]
Type Map Example

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

- What is Type Map of newtype if defined as below?
  \[\text{MPI\_Type\_vector}(2,3,4,\text{oldtype}, &\text{newtype})\]

Ans:
\[
\{(\text{double, 0}), (\text{char, 8})\}, \{(\text{double, 16}), (\text{char, 24})\}, \{(\text{double, 32}), (\text{char, 40})\}, \{(\text{double, 64}), (\text{char, 72})\}, \{(\text{double, 80}), (\text{char, 88})\}, \{(\text{double, 96}), (\text{char, 104})\}\]
Exercise: MPI_Type_vector

- Express

\[ \text{MPI\_Type\_contiguous(count, old, \&new);} \]
...as a call to \text{MPI\_Type\_vector}

\[ \text{MPI\_Type\_vector(int count, int blocklength, int stride, MPI\_Datatype oldtype, MPI\_Datatype \*newtype);} \]

- Ans:

\[ \text{MPI\_Type\_vector (count, 1, 1, old, \&new);} \]
\[ \text{MPI\_Type\_vector (1, count, count, old, \&new);} \]
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
In some MPI implementations there are more than 300 MPI functions.

Not all of them part of the MPI standard though, some vendor specific.

Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code…
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 PDE Numerical Solution Utilities

- ODE Integrators
- Visualization
- Nonlinear Solvers, Unconstrained Minimization
- Interface
- Linear Solvers
- Preconditioners + Krylov Methods
- Grid Management
- Object-Oriented Matrices, Vectors, Indices
- Profiling Interface

Computation and Communication Kernels
- MPI, MPI-IO, BLAS, LAPACK
### PETSc Numerical Components

#### Nonlinear Solvers
- Newton-based Methods
- Line Search
- Trust Region
- Other

#### 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

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PETSc

PC

KSP

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

User code

PETSc code
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:
  \[ s = \text{Fraction of serial execution time that cannot be parallelized} \]
  \[ N = \text{Number of processors} \]

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

- 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

- 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
Most Important Two Things

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

- Hone your “computational thinking” skills
  - Understand the hardware and how it works