Recall: Parallel Computing Hardware

Multiprocessor:
- CPU
- Cache Memory
- Bus
- Primary Memory
- I/O Devices

Multicomputer:
- CPU
- Cache Memory
- Memory
- I/O Devices
- Interconnection Network

Multicore:
- Core
- Data Cache
- Instr. Cache
- TLB
- Shared Unified Cache
- Main Memory
- Level 1
- Level 2
Motivation

As easy as OpenMP is to use, it cannot solve all parallel computing problems:

- Potential bugs due to private vs. shared variables require constant diligence.
- While non-loop-level parallelism is possible, it is more difficult to implement.
- Limited to SMP architectures, a small/decreasing subset of modern parallel computers.
- SMP architectures have limited available memory; cannot handle very large problems.
- The high cost of SMP machines leads to a non-scalable cost/size model.

Alternately, distributed-memory parallelism has its own difficulties:

- All memory is private, so any data sharing must be performed explicitly.
- Theoretically infinite available memory through adding nodes, though still limited by available memory on individual nodes (often rather small).
- Low-level programming leads to incredible flexibility in parallel approaches, though requires more tedious implementation.
- Scalable cost/size model, depending on the network backbone.
- Retains many SMP bugs: data race conditions, synchronization points, dead-lock.
Distributed-Memory Model

- Assumes that the machine consists of a collection of processors, each with local memory.

- Each processor can access only the instructions/data stored in its own memory.

- The machine has an interconnection network that supports passing messages between processors: A may send a message containing some of its data to B.

- A user specifies a number of concurrent processes when program begins; the number of active processes typically remains constant throughout execution. However, dynamic parallelism is also possible.

- Every process executes the same program, though the flow of execution should depend on the processor’s unique ID number (e.g. “if (myid == 0) { }”).

- Each process performs computations on its local variables, then communicates with other processes, (and repeats), to eventually achieve the computed result.

- In this model, message passing performs two roles: to send/receive information, and to synchronize with one another.
Advantages

The distributed model has some distinct advantages over SMP programming:

- Programs can utilize MIMD computers and computers without a shared address space (the majority of parallel computers today).

- There is a clear distinction between faster, directly-accessible local memory and slower, indirectly-accessible remote memory.
  - Forces algorithm design to maximize local computation, minimize communication.
  - Programs also run well on SMP machines; each process owns private variables in the shared address space (making better use of cache), “communication” consists of copying variables in RAM.

- Distributed programs can be simpler to debug than SMP programs:
  - Since each processor controls its own memory, it is impossible for one process to overwrite a variable controlled by another process.
  - Programs can be written to execute deterministically; can aid in debugging.
A Brief History of MPI

- Vendors began producing distributed-memory computers in the late 1980s.
- As with SMP, each vendor provided its own set of message passing functions, which were not portable to other architectures.
- In 1989, researchers at Oak Ridge National Lab invented PVM (Parallel Virtual Machine). PVM was portable across platforms, and was publicly released in 1993.
- Meanwhile, the Center for Research on Parallel Computing sponsored workshops for experts in the field to determine a standard for distributed memory computing.
- Their result was MPI-1.0, which was drafted in 1994 (~14 years).
- MPI has been refined over time; the MPI-2 standard was adopted in 1997, the MPI-3 standard in 2012, and the MPI-3.1 standard in 2015.
- For a while, both PVM & MPI were widely used; however, MPI has surpassed PVM to become the most popular message-passing library for parallel programming.
- MPI is now available on nearly all commercial multicomputers. Free implementations exist for commodity clusters, workstations, and even laptops (pre-installed in OS X and most Linux).
What is MPI?

The MPI goal:

To demonstrate that users need not compromise among \textit{efficiency}, \textit{portability} and \textit{functionality} when writing parallel programs. Hence, users can write portable programs that take advantage of specialized hardware offered by vendors on specific platforms.

What is MPI?

- Addresses the message passing model for distributed-memory parallel computing.
- An attempt to collect the best features of many previous message passing systems, improve where appropriate, and then standardize them.
- It is a \textit{library}, i.e. a set of subroutines, functions and constants. Users write programs in a standard computing language (Fortran, C, C++), compile with an ordinary compiler, and link their code with an appropriate MPI library.
- It is a \textit{specification}, not an implementation. Each vendor's implementation may leverage proprietary hardware advances. Free MPI implementations may be downloaded from the internet (MPICH, Open-MPI, MVAPICH, LAM-MPI, …)
The MPI Model

Since concurrent processes have separate address spaces, we must *communicate* data.

- Occurs when part of one’s address space is copied into another’s address space.
- This is *cooperative*, and can typically only occur when both the first process executes a *send* operation, and the second process executes a *receive* operation.

A natural question arises: “what is the minimal information required for these functions?”

**Sender:**

- The *data* to communicate.
- The *amount of data* to communicate.
- The *destination* for the message.
- A message *tag* so that the receiver can know *which* message is arriving (if multiple messages are coming in).
- We could therefore have a call syntax of the form

  \[
  \text{send(address, length, destination, tag)}
  \]
Similarly, the receiving process must do some work on its end to receive the data appropriately.

Receiver:

- *Where* to put the incoming data:
  - The starting memory address,
  - The length of the buffer.
- The *identity* of the sender.
- The message *tag*.
- If the message is smaller than the area set aside in memory, we may need an output argument specifying the *actual message size*.
- We could therefore have a call syntax of the form
  
  \[
  \text{receive}(\text{address, length, source, tag, actual\_length})
  \]
The MPI Send/Receive

What is the MPI approach?

Describing the message (address, length):

- Some multicomputers are heterogeneous (different architecture, OS, etc.), so individual nodes could store numbers in different formats.
- Hence, just specifying a buffer length may result in one set of floating point numbers on the sender, and a different set for the receiver.
- MPI has therefore extended the buffer description slightly:
  
  \[(address, count, datatype)\]

  to specify:
  
  - Where the send/receive buffer begins in memory (address)
  - How many entries are in the buffer (count)
  - What these entries look like (datatype)

- MPI can then do any necessary translating from one architecture to another.
- For example, \((A, 300, MPI\_INT)\) corresponds to an array A holding 300 integers.
- MPI also allows specification of custom data types (though we won’t use those in this class)
The MPI Send/Receive

Separating Families of Messages (tag)

- The tag allows the programmer to deal with arrival of messages in an orderly fashion, even if the messages arrive in an unexpected order.
- Messages that arrive “of the wrong tag” are queued until the program is ready for them.
- MPI groups these tags with a context, which is allocated by the system at runtime in response to user requests. They are used along with tags for matching purposes.

Naming Processes:

- MPI processes belong to groups. If a group contains $p$ processors, then they are identified with unique integer “ranks,” from 0 to $(p - 1)$.
- All processes have a default group, but a programmer may create special groups if desired.

Communicators:

- The notions of context and group are combined in a single MPI communicator.
- The destination and source specified in sends and receives refer to the integer rank of the process in the group identified by the communicator.
**The MPI Send/Receive**

```c
int MPI_Send(void* address, const int count, MPI_Datatype dtype,
             int dest, int tag, MPI_comm comm)

(address, count, dtype) describe the message to be sent,
dest is the rank of the receiving processor in the communicator comm,
tag is an identifier used for message passing,
comm identifies the process communicator group,
Return value denotes whether the send completed successfully (MPI_SUCCESS) or not.
```

```c
int MPI_Recv(void* address, int count, MPI_Datatype dtype,
              int src, int tag, MPI_Comm comm, MPI_Status* stat)

(address, maxcount, dtype) describe the message receive buffer,
(tag, comm) are as above; an optional wildcard, matching any tag, is allowed,
src is the sender’s rank in the communicator comm, or is a wildcard matching any source,
stat holds information about the actual message size, source, and tag, if any were left unspecified above.
```
Fortran vs C/C++

MPI library calls are nearly identical between the Fortran and C/C++ APIs.

The primary difference is that in Fortran, the success/failure return flag is provided as the last argument to the function, that therefore has no return type.

Hence, the `MPI_Send()` and `MPI_Recv()` commands are modified for Fortran to be:

```c
MPI_Send(address, count, dtype, dest, tag, comm, ierr)
MPI_Recv(address, count, dtype, src, tag, comm, stat, ierr)
```

Where
- `address` is the first entry of the send/recv buffer.
- `count` and `dtype` are integer arguments specifying amount & type of data.
- `dest`, `src` and `tag` are integer arguments specifying message characteristics.
- `comm` is an integer specifying the communicator containing the source and destination processes.
- `stat` is an integer array of length `MPI_STATUS_SIZE` that contains information about the received message.
Other MPI Features

MPI does much more than just simple sends and receives.

**Collective communications:**
- Data movement to rearrange data among many processes (e.g. a broadcast)
- Collective computation (e.g. max, min, sum, logical .or., etc.)
- Implementations may utilize specific hardware optimizations to improve performance.

**Virtual topologies:**
- Can arrange processors in an application-specific topology for added convenience.
- Grids of processors (1D, 2D, 3D), and general processor graphs are supported.

**Debugging & Profiling:**
- MPI requires the provision of “hooks” that users may use to intercept MPI calls.
- Allows users to define debugging and profiling mechanisms (or use existing debuggers)

**Communication modes:**
- MPI enables *blocking* and *non-blocking* versions of the send and receive operations.
- These enable increased/decreased processor synchronization in a user’s program.
- Supports different *communication modes*, that change how MPI handles buffering and synchronization.
Other MPI Features

Library support:

- By structuring all messages through communicators, MPI allows creation of parallel libraries that are completely independent of user code, and that are interoperable with one another.
- As a result, there are numerous free, high-quality solver libraries that enable large-scale parallel computations.

Support for heterogeneous networks:

- By supplying internal translation between machine-dependent data types, MPI programs can run on networks of highly heterogeneous machines.
- These translations are performed in each MPI implementation, allowing vendors to optimize these conversions for a given architecture.

Processes vs processors:

- The MPI standard discusses processes, as opposed to processors.
- Some implementations allow multiple processes per processor core, others limit runs to a single process per core.
How Complicated is MPI?

- While MPI includes a large number of features, the number of ideas in MPI is relatively small, and many MPI functions may be built using only a small set.

- As with OpenMP, users may begin to write MPI programs with only a minimal set of indispensable functions, adding more functions as they gain experience.

- Other functions add flexibility (datatypes), robustness (nonblocking send/receive), modularity (communicators), convenience (collective operations, topologies), or efficiency (modes/one-sided communication).

- Users may begin with the following six base MPI functions:

<table>
<thead>
<tr>
<th>MPI Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init()</td>
<td>Initializes MPI</td>
</tr>
<tr>
<td>MPI_Comm_size()</td>
<td>Finds out the total number of processes</td>
</tr>
<tr>
<td>MPI_Comm_rank()</td>
<td>Finds out this processor’s ID</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>Sends a message</td>
</tr>
<tr>
<td>MPI_Recv()</td>
<td>Receives a message</td>
</tr>
<tr>
<td>MPI_Finalize()</td>
<td>Terminates MPI</td>
</tr>
</tbody>
</table>
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

// Example routine using the 6 basic MPI functions
int main(int argc, char* argv[]) {

    // local variables
    int nprocs, myid, num, p, tag, sender;
    MPI_Status status;

    // initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    // everyone sends (ID+1) to root with // tag mirroring sender proc ID
    if (myid != 0) {
        num = myid+1;
        MPI_Send(&num, 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
    }

    // root receives messages (in order) // and outputs each to screen
    if (myid == 0) {
        for (p=1; p<nprocs; p++) {
            // receive num from this proc
            MPI_Recv(&num, 1, MPI_INT, p,
                      MPI_ANY_TAG,
                      MPI_COMM_WORLD, &status);

            // get information about message
            tag = status.MPI_TAG;
            sender = status.MPI_SOURCE;

            // output information to stdout
            printf("recv %i from proc %i", num, p);
            printf(", tag = %i, sender = %i\n", tag, sender);
        }
    }

    // finalize MPI
    MPI_Finalize();

    return 0;
} // end main
A First MPI Program – Output

Running this program on 10 MPI processes generates the following output:

received value 2 from processor 1, tag = 1, sender = 1
received value 3 from processor 2, tag = 2, sender = 2
received value 4 from processor 3, tag = 3, sender = 3
received value 5 from processor 4, tag = 4, sender = 4
received value 6 from processor 5, tag = 5, sender = 5
received value 7 from processor 6, tag = 6, sender = 6
received value 8 from processor 7, tag = 7, sender = 7
received value 9 from processor 8, tag = 8, sender = 8
received value 10 from processor 9, tag = 9, sender = 9
General Definitions

**Nonblocking** – If the procedure may return before the operation completes, and before the user is allowed to re-use resources (such as buffers) specified in the call.

**Blocking** – If return from the procedure indicates the user is allowed to re-use resources specified in the call. Both `MPI_Send()` and `MPI_Recv()` are blocking.

**Local** – If completion of the procedure depends only on the local executing process. Such an operation does not require communication with other processes. The routines `MPI_Comm_size()` and `MPI_Comm_rank()` are local.

**Non-local** – If completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process. `MPI_Send()` and `MPI_Recv()` are of course non-local.

**Collective** – If all processes in a process group must invoke the procedure. Both `MPI_Init()` and `MPI_Finalize()` are collective routines.
<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C/C++ Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>(a single byte)</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(non-contiguous data)</td>
</tr>
</tbody>
</table>
# MPI Datatypes (C++/C)

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C/C++ Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>Pair of float &amp; int values</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>Pair of double &amp; int values</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>Pair of long &amp; int</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>Pair of int &amp; int</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>Pair of short &amp; int</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>Pair of long double &amp; int</td>
</tr>
</tbody>
</table>
### MPI Datatypes (Fortran)

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_INTEGER1</td>
<td>integer*1</td>
</tr>
<tr>
<td>MPI_INTEGER2</td>
<td>integer*2</td>
</tr>
<tr>
<td>MPI_INTEGER4</td>
<td>integer*4</td>
</tr>
<tr>
<td>MPI_INTEGER8</td>
<td>integer*8</td>
</tr>
<tr>
<td>MPI_INTEGER16</td>
<td>integer*16</td>
</tr>
<tr>
<td>MPI_2INTEGER</td>
<td>pair of integer</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>real</td>
</tr>
<tr>
<td>MPI_REAL4</td>
<td>real*4</td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td>real*8</td>
</tr>
<tr>
<td>MPI_REAL16</td>
<td>real*16</td>
</tr>
<tr>
<td>MPI_2REAL</td>
<td>pair of real</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>real*8</td>
</tr>
<tr>
<td>MPI_2DOUBLE_PRECISION</td>
<td>pair of real*8</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>complex</td>
</tr>
<tr>
<td>MPI_COMPLEX8</td>
<td>complex*4</td>
</tr>
<tr>
<td>MPI_COMPLEX16</td>
<td>complex*8</td>
</tr>
<tr>
<td>MPI_COMPLEX32</td>
<td>complex*16</td>
</tr>
<tr>
<td>MPI_2COMPLEX</td>
<td>pair of complex</td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>complex*8</td>
</tr>
<tr>
<td>MPI_2DOUBLE_COMPLEX</td>
<td>pair of complex*8</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>character(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>(a single byte)</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(non-contiguous data)</td>
</tr>
</tbody>
</table>
Additional Notes on our Example

MPI has a number of reserved names and constants (defined in mpi.h):

- **MPI_COMM_WORLD** – the default MPI communicator,
- **MPI_ANY_TAG** – wildcard tag, allowing receipt of any messages that may come,
- **MPI_ANY_SOURCE** – wildcard source, allowing receipt from any sender,
- **MPI_STATUS_SIZE** – the number of entries in the status array,
- **MPI<Tag>** – index in status array storing message tag information,
- **MPI_SOURCE** – index in status array storing message source information,
- **MPI_ERROR** – index in status array storing the error code for the received message,
- **MPI_SUCCESS** – return value for functions indicating successful completion,

To query the message length (count entries of type datatype) from the status:

```c
int MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)
```
MPI Timing & Performance Utilities

As with everything in this class, we would like an accurate timer. Like OpenMP, MPI provides convenient and portable timing functions:

```c
double MPI_Wtime()
```

- Returns a double-precision floating point number that is the time in seconds since some arbitrary point of time in the past.
- Use like a stopwatch by calling before and after a program segment, and subtracting.
- The time resolution of this function will be hardware-dependent.

```c
double MPI_Wtick()
```

- Returns the finest resolution of time increments available on the machine, i.e. it returns a floating-point number that is the time in seconds between successive ticks of the clock.

General comments:
- These functions are not synchronized between processes, i.e. you will get different values on each node/process, so they should only be used for timing each task separately.
- If you are using MPI, I suggest that you always use these timers.
Collective Computations

Collective computations are useful when all processors need to communicate toward a shared goal. Consider the following dot-product using the 6 basic MPI subroutines:

```c
#include <stdlib.h>
#include <stdio.h>
#include “mpi.h”

int main(int argc, char* argv[]) {

    // local variables
    int nprocs, myid, num, p, i, N;
    const int n=10000;
    double lsum=0.0, sum=0.0;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Status stat;

    // initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &nprocs);
    MPI_Comm_rank(comm, &myid);

    // set global vector length
    N = n*nprocs;

    // allocate vectors
    double a[n];  double b[n];

    // initialize values
    for (i=0; i<n; i++) {
        a[i] = (0.001*(myid*n+i+1)/N;
        b[i] = (0.001*(N-myid*n-i-1)/N;
    }

    // compute local dot product
    for (i=0; i<n; i++)  lsum+=a[i]*b[i];

    // communicate for overall dot product
    if (myid != 0) {
        MPI_Send(&lsum, 1, MPI_DOUBLE, 0, myid, comm);
    } else {
        sum = lsum;
        for (p=1; p<nprocs; p++) {
            MPI_Recv(&lsum, 1, MPI_DOUBLE, p, MPI_ANY_TAG, comm, &stat);
            sum += lsum;
        }
    }

    printf(“dot product = %g\n”,sum);
}

MPI_Finalize();
```
Collective Computations

The previous example works, but has a few key drawbacks:

- Only processor 0 ends up with the result,
- It is somewhat complicated for such a simple operation,
- The final reduction requires that the addition occur in increasing processor order.

MPI therefore has special routines to handle collective communications/computations:

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

- All processors in `comm` send the values from their `sendbuf` (of type `dtype`) to the root.
- The root processor receives the `count` values from each processor, performs `op` on them, and places the resulting `count` values in `recvbuf`.
- The return argument is as before: MPI_SUCCESS if the operation worked, otherwise failure.

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

- Just like `MPI_Reduce()`, but the result of the reduction is sent back to ALL processors in `comm` (not just `root`), who each receive the result in their `recvbuf`. 
The reduction buffer `recvbuf` can be a scalar (`count = 1`), an array (`count > 1`), or even a multi-dimensional array (`count >> 1`) as long as it's stored contiguously.

The send and receive buffers, `sendbuf` and `recvbuf`, *cannot be the same*.

MPI has many pre-defined operations (op) for collective computation:

<table>
<thead>
<tr>
<th>MPI op Name</th>
<th>Operation</th>
<th>Allowed Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_INTEGER, MPI_REAL,(MPI_COMPLEX (Fortran only) and variants</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td></td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td></td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Local or</td>
<td></td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or (xor)</td>
<td></td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
<td>MPI_INTEGER, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
<td></td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise xor</td>
<td></td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum value and location</td>
<td>MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT, MPI_LONG_DOUBLE_INT</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum value and location</td>
<td></td>
</tr>
</tbody>
</table>
Dot Product Simplified

Using the MPI_Reduce() function, we may simplify the previous dot product routine:

```c
#include <stdlib.h>
#include <stdio.h>
#include “mpi.h”

int main(int argc, char* argv[]) {

    // local variables
    int nprocs, myid, num, p, i, N;
    const int n=10000;
    double lsum=0.0, sum=0.0;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Status stat;

    // initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &nprocs);
    MPI_Comm_rank(comm, &myid);
    // set global vector length
    N = n*nprocs;
    // allocate vectors
    double a[n];  double b[n];

    // initialize values
    for (i=0; i<n; i++) {
        a[i] = (0.001*(myid*n+i+1)/N;
        b[i] = (0.001*(N-myid*n-i-1)/N;
    }

    // compute local dot product
    for (i=0; i<n; i++)  lsum+=a[i]*b[i];

    // communicate for overall dot product
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
                MPI_SUM, 0, comm);

    if (myid == 0)
        printf(“dot product = %g\n”,sum);

    MPI_Finalize();
}
```
Dot Product Demo

Convert our previous serial dot-product program to parallel:

- Start with a simple approach that allocates the full arrays for a and b on all processors, but they share computation in personal subintervals.
- Add in `MPI_Wtime()` and `MPI_Wtick()` instead of previous timers.
- Update approach so that processors only allocate local portions of arrays.
Collective Communications

In addition to collective computation routines, MPI has special collective communication routines for data movement between processors:

**Broadcast:**

- Data is broadcast from P0 to all other processors (P1, P2, P3).
- Each processor receives the same data (A).}

**Scatter/Gather:**

- Data is scattered from P0 to P1, P2, and P3.
- Data is gathered from P1, P2, and P3 to P0.
- Each processor receives a different data set (A, B, C, D).
In addition to collective computation routines, MPI has special collective communication routines for data movement between processors:

**Allgater:**

<table>
<thead>
<tr>
<th>Processors</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A</td>
</tr>
<tr>
<td>P1</td>
<td>B</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
</tr>
</tbody>
</table>

**All to All:**

<table>
<thead>
<tr>
<th>Processors</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A0</td>
</tr>
<tr>
<td>P1</td>
<td>B0</td>
</tr>
<tr>
<td>P2</td>
<td>C0</td>
</tr>
<tr>
<td>P3</td>
<td>D0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processors</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A1</td>
</tr>
<tr>
<td>P1</td>
<td>B1</td>
</tr>
<tr>
<td>P2</td>
<td>C1</td>
</tr>
<tr>
<td>P3</td>
<td>D1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processors</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A2</td>
</tr>
<tr>
<td>P1</td>
<td>B2</td>
</tr>
<tr>
<td>P2</td>
<td>C2</td>
</tr>
<tr>
<td>P3</td>
<td>D2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processors</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A3</td>
</tr>
<tr>
<td>P1</td>
<td>B3</td>
</tr>
<tr>
<td>P2</td>
<td>C3</td>
</tr>
<tr>
<td>P3</td>
<td>D3</td>
</tr>
</tbody>
</table>
Collective Communications

Function Prototypes (all are collective, nonlocal, and blocking):

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

int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)

int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)

int MPI_Allgather(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)

int MPI_AlltoAll(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
```
Computing $\pi$ Demo

In-class demo, parallelizing our previous $\pi$ computation program:
- Use `MPI_Bcast()` to send the total number of subintervals to each processor.
- Each processor computes their own portion of $\pi$.
- These local portions are then combined back together using `MPI_Reduce()`.
- Add in `MPI_Wtime()` and `MPI_Wtick()` instead of previous timers.
- Update approach to send final result out to all processes using `MPI_Allreduce()`.
Simple Error Handling

- There are numerous reasons to stop a program in the middle of the computation: illegal arguments, solver non-convergence, illegal values (NaN), etc.
- Parallel computing adds in a strange difficulty: if only one process encounters a fatal error, how can you get all of the processes to stop?
- MPI provides a simple subroutine that makes a “best attempt” to abort all tasks associated with a communicator.

Function prototype:

```c
int MPI_Abort(MPI_Comm comm, int errorcode)
```

where

- `comm` is the communicator whose processes should be stopped (e.g. MPI_COMM_WORLD).
- `errorcode` is an integer that should be returned to the calling process (the shell).
- the return value denotes successful completion of the subroutine (pointless?).
MPI Communicators

As mentioned earlier, MPI allows the notion of a *virtual topology*, wherein a communicator can arrange processes in an application-specific topology.

In addition, MPI allows more general communicators, that can create sub-groups of processes, split apart communicators into disjoint groups, etc.

Here, we consider *Cartesian* communicators, that group processes into either a 1D, 2D or 3D processor grid. Each topology will have the following attributes:

- **Dimension (1, 2 or 3?),**
- **Process layout* (how many total processes in each direction?),**
- **Process location (where is this process in the layout?),**
- **Periodicity (is the dimension periodic?).**
Cartesian Communicators

We can create a Cartesian MPI communicator with the routine:

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

- `comm_old` is the original communicator (e.g. MPI_COMM_WORLD) – input
- `ndims` denotes the process topology dimension – input
- `dims` denotes how many processes are in each direction – input array
- `periods` denotes whether each dimension is periodic – input array
- `reorder` flag allowing MPI to rearrange the processes ‘optimally’ – input
- `comm_cart` is the new Cartesian communicator – output

We may subsequently query a Cartesian communicator for parallelism information:

```c
int MPI_Cart_get(MPI_Comm comm, int maxdims, int* dims,
                  int* periods, int* coords)
```

- `comm` is the Cartesian communicator of dimension `maxdims` – both are input
- `dims` and `periods` are as above – output arrays
- `coords` denotes the location of this process in the grid – output array
Cartesian Communicators

MPI also provides mappings between the process ID and its location in the process grid.

The first routine, mapping from process ID to location, is

```c
int MPI_Cart_coords(MPI_Comm comm, int rank,
                     int maxdims, int* coords)
```

- `rank` denotes the process ID you wish to find – input
- `coords` holds the process location in the grid – output array (of length `maxdims`)

The second routine, that maps from location to process ID, is

```c
int MPI_Cart_rank(MPI_Comm comm, int* coords, int* rank)
```

- `coords` denotes the location in the process grid – input array
- `rank` holds the process ID in that location – output integer
Matrix-Vector Product Demo

In-class demo, parallelizing our previous serial matrix-vector product program:

Version 1:
- Store matrix on every process; only compute your own rows.
- Root node collects overall result using MPI_Gather().
- Add in simple error handling in case calls fail.

Version 2:
- Convert to a block row decomposition of the matrix.
- Update approach to only store local matrix rows.

Version 3:
- Convert to a block column decomposition of the matrix.
- Update approach to only store local matrix columns.
2D Laplace Residual Demo

Linear residual calculation demo for a 2D Laplace operator on a regular finite difference grid.

- Requires an 8-phase system for 2D communication to neighbors:
  - X-directional communication:
    - Evens send East, while odds receive West
    - Odds send West, evens receive East
    - Odds send East, evens send West
    - Evens send West, odds receive East
  - Y-directional communication:
    - Evens send North, odds receive South
    - Odds send South, evens receive North
    - Odds send North, evens receive South
    - Evens send South, odds receive North

- Once all data is available, perform the residual computation.

*Note:* We need 8 distinct phases to avoid deadlock due to an incorrect ordering of messages.
Collective Communications Revisited

- A simple deficiency in the previous Scatter, Gather, Allgather and AlltoAll routines is that they must send/receive the same number of items to/from, each process.

- MPI therefore includes five more advanced communication routines: MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv, MPI_Alltoallv

- These routines allow different amounts of data for each associated process.

**MPI_Gatherv()** and **MPI_Scatterv()**:
MPI_*v Details

Function prototypes:

```c
int MPI_Scatterv(void* sbf, int* scts, int* displs,
                  MPI_Datatype styp, void* rbf, int rct,
                  MPI_Datatype rtype, int root, MPI_Comm comm)

int MPI_Gatherv(void* sbf, int sct, MPI_Datatype styp, void* rbf,
                 int* rcts, int* displs, MPI_Datatype rtyp,
                 int root, MPI_Comm comm)

int MPI_Allgatherv(void* sbf, int sct, MPI_Datatype styp,
                   void* rbf, int* rcts, int* displs,
                   MPI_Datatype rtyp, MPI_Comm comm)

int MPI_Alltoallv(void* sbf, int* scts, int* sdispls,
                   MPI_Datatype styp, void* rbf, int* rcts,
                   int* rdispls, MPI_Datatype rtyp, MPI_Comm comm)
```

All arguments are as before, except for:

- `scts`, `rcts` are now integer arrays (with length the size of the communicator group), corresponding to the number of elements to *send to*, or *receive from*, each other process.

- `displs`, `sdispls`, and `rdispls` are integer arrays (with length the group size), that specify the displacement (relative to `sbf`) from which to take the outgoing data destined for each process, or to place the incoming data from each process.
Two-Way Communication

Our earlier send and receive operations allowed point-to-point communication, but were blocking, and required two phases if a process wants to both send and receive information.

For the latter scenario, we have the MPI routine:

```c
int MPI_Sendrecv(void* sbuf, int scount, MPI_Datatype stype,
                 int dest, int stag, void* rbuf, int rcount,
                 MPI_Datatype rtype, int source, int rtag,
                 MPI_Comm comm, MPI_Status stat)
```

- Executes a blocking send and receive operation, where a process wants to do both a send and receive at the same time.
- Only `rbuf`, `status` and the usual error flag are outputs, all the rest are inputs.
- Both the send and receive operations use the same communicator, but possibly different tags.
- The source and destination processes need not be the same.
- `sbuf` and `rbuf` may have different lengths and data types.
Blocking vs. Non-blocking

All of our communications so far have been *blocking*, i.e. the calling process must wait until the send/recv is finished before the program can move on.

- Complicated communication patterns require even more complicated code.
- A program’s execution can slow down to the speed of the communication network.

We may instead perform *non-blocking* send and receive operations:

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

- Executes a non-blocking send operation.
- `request` is an output that we can query later on to see if the send completed.

```c
int MPI_Irecv(void* buf, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm,
              MPI_Request* request)
```

- Executes a non-blocking receive operation.
- `request` is an output that we can query later on to see if the receive completed.
Non-blocking Communication

To determine whether the non-blocking communications have completed, we have:

```c
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

and

```c
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status)
```

- These check to see whether the send/receive associated with `request` has completed.
- The first blocks as it waits; the second returns `flag=1` if the operation has completed.

Alternatively, we may use a barrier to synchronize all processes at a given point in the code:

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

- All processes in the communicator wait at this point until others have caught up.
Non-blocking 2D Laplace Residual

Instead of our complicated 8-phase system for 2D communication to neighbors, we may communicate boundary information to neighbors in a simpler manner. Each process will:

- Allocate receive buffers for neighbor information,
- Open receive channels to fill these buffers, using `MPI_Irecv()`,
- Fill send buffers to send to neighbors,
- Send information to waiting neighbor processes, using `MPI_Isend()`,
- Wait for the information to arrive, using `MPI_Wait()`,
- Perform the residual computation.

Notes:

- We no longer have to worry about deadlock due to an incorrect ordering of messages.
- We could even do some computations while we wait for the boundary data to arrive. This overlapping of communication and computation is desirable, since codes can operate at speeds much faster than the communication network.
- We must be careful with the tags, since each process may receive multiple messages simultaneously, and we need to ensure that they end up in the right places.
Common MPI Errors

Some typical pitfalls that new (and experienced) MPI programmers often make:

- Doing things before `MPI_Init()` and `MPI_Finalize()`:
  - The MPI standard says nothing about what can happen before `MPI_Init()` and after `MPI_Finalize()`, not even how many processes are running.
  - Doing anything in your program during these periods is risky and may not be portable.

- Matching collective communications [e.g. `MPI_Bcast()`] with `MPI_Recv()`:
  - Often, one thinks of `MPI_Bcast()` as a “multiple send” operation, and the result should be “received” by all processes.
  - But `MPI_Bcast()` is *already* a collective communication, so any `MPI_Recv()` will hang.

- Forgetting to use a pointer to an object when required (especially on a send/recv buffer) instead of the object itself. Usually the compiler will catch this, but if it doesn't then it can be very difficult to find the error.
New Features (MPI v3 & v3.1)

The newest MPI standards include numerous advanced features, with the goal of improving performance of large-scale applications on modern parallel hardware:

- **Non-blocking collectives** – extension of existing collective communication & computation routines to support separate initiation/wait operations

- **User-defined reduction operations** – introduction of arbitrary user-defined operations

- **Threading support** – introduction of routines to better support hybrid MPI+threading calculations

- **More advanced communicators** – ability to create/spawn communicator groups, differentiate between "intracommunicator" and "intercommunicator" for separation of groups contributing to a calculation

- **One-sided communication** – one process may copy local data into another process' memory space, or retrieve values from other another process' memory, without needing the other processor to explicitly participate with a “Send” or “Receive”

- **Persistent communication requests** – if a communication with the same argument list is repeatedly executed within a computation, the communication can be optimized by first setting up a persistent communication "request" that will be reused in subsequent send/recv operations.

...
MPI v3 – Non-blocking collectives

The model is similar to before:

a) A nonblocking call initiates the collective operation, returning a request handle

b) Once initiated, all associated send buffers and input arguments should not be modified, and all associated receive buffers should not be accessed, until the collective operation completes

c) A separate completion call (e.g. MPI_Wait() or MPI_Test()) is required to verify that the operation has completed

Non-blocking collectives – MPI_Ibroadcast() MPI_Igather() MPI_Iscatter()
MPI_Iallgather() MPI_Ialltoall() MPI_Iallreduce() MPI_Ireduce() …

Note: multiple nonblocking collective operations can coexist, although all processes must call collective operations (blocking and nonblocking) in the same order (per communicator).

Nonblocking barrier synchronization (used to hide latency):

    int MPI_Ibarrier(MPI_Comm comm, MPI_Request* request)

This call notifies that a process has reached the barrier. The call returns immediately; the usual barrier semantics are enforced at the corresponding test/wait operation.
MPI v3 – User-defined reduction ops

You may now add to the set of allowable reduction operations:

```c
int MPI_Op_create(MPI_User_fn* ufn, int commute, MPI_Op* op)
```

- `op` – new operation that can be used in subsequent calls to reduce, scan, etc.
- `commute` – integer flag denoting whether operation is commutative (aids efficiency)
- `ufn` – the user-defined function (assumed to be associative); must have prototype
  ```
  void foo(void* ivec, void* iovec, int* len, MPI_Datatype* dtype)
  ```
  - `dtype` – handle to the data type passed to reduction
  - Should perform an operation of the form: (here ♠️ is the new operation)
    ```
    for (i=0; i<*len; i++)
      iovec[i] = iovec[i] ♠️ ivec[i];
    ```

To mark a user-defined reduction operation for deallocation:

```c
int MPI_Op_free(MPI_Op* op)
```
MPI v3 – Threading support

int MPI_Init_thread(int* argc, char*** argv, int required,
                     int* provided)

A replacement for MPI_Init(), this initializes the MPI thread environment.

- **required** – specifies the desired level of thread support:
  - MPI_THREAD_SINGLE – Only one thread will execute [standard MPI_Init()]
  - MPI_THREAD_FUNNELED – The process may be multi-threaded, but the application must ensure that only the main thread makes MPI calls
  - MPI_THREAD_SERIALIZED – The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time
  - MPI_THREAD_MULTIPLE – Multiple threads may call MPI, with no restrictions

- **provided** – return value with actual level of thread support (same values as above)

Other query functions may be used to ascertain threading support, master thread:

int MPI_Query_thread(int* provided)

int MPI_Is_thread_main(int* true_false_flag)
MPI v3 – [Intra/Inter]communicators

MPI3 now considers two types of communicators:

- **Intracommunicator** – an identifier for a single group of processes linked with a context. The “standard” communicator from MPI v1 and v2.

- **Intercommunicator** – identifies two distinct groups of processes that must communicate. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. Inter-communication also enables a dynamic MPI model where not all processes are preallocated at initialization time.


- E/A – `MPI_Comm_create()`, `MPI_Comm_dup()`, `MPI_Comm_idup()`, `MPI_Comm_dup_with_info()`, and `MPI_Comm_split()`

- A – `MPI_Comm_create_group()` and `MPI_Intercomm_merge()`

- E – `MPI_Intercomm_create()`
MPI v3 – Dynamic Parallelism

```c
int MPI_Comm_spawn(const char* cmd, char* argv[], int mxproc,
                   MPI_Info info, int root, MPI_Comm comm,
                   MPI_Comm* intercomm, int err_array[])
```

- **cmd** – name of program (in filesystem) to be spawned (input, string)
- **argv** – arguments to **cmd** (input, array of strings, terminate w/ NULL in C)
- **mxproc** – maximum number of processes to start (input, integer)
- **info** – key-value pairs telling runtime system where/how to start the processes (input)
- **root** – rank of process in which previous arguments are examined (input, integer)
- **comm** – intracommunicator containing group of spawning processes (input)
- **intercomm** – intercommunicator between original group and spawned group (output)
- **err_array** – one code per process (output, array of integers)

Tries to start **mxproc** copies of the program specified by **cmd**, establish communication and return an intercommunicator. The spawned processes “children” have their own **MPI_COMM_WORLD**, which is separate from that of the parents.

The “parent” intercommunicator for a spawned process may be obtained with

```c
int MPI_Comm_get_parent(MPI_Comm *parent)
```
Remote Memory Access (RMA) extends MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side.

- Remote write: `MPI_Put()`, `MPI_Rput()`
- Remote read: `MPI_Get()`, `MPI_Rget()`
- Remote update: `MPI_Accumulate()`, `MPI_Raccumulate()`
- Remote atomic swap operations: `MPI_Compare_and_swap()`
- Remote read and update: `MPI_Get_accumulate()`, `MPI_Rget_accumulate()`, and `MPI_Fetch_and_op()`

To enable, each process must specify a “window” in its memory to make accessible by remote processes. RMA ops can then be performed on “windowed” data by all processes in a group:

```c
int MPI_Win_create(void* base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win* win)
```

- `base` – initial address of window
- `size` – size of window in bytes (non-negative integer)
- `disp_unit` – local unit size for displacements, in bytes (positive integer)
- `info` – info arguments (no_locks, accumulate_ordering, accumulate_ops)
MPI v3 – Persistent Communicators

To create a persistent send or receive (non-blocking) communicator:

```c
int MPI_Send_init(const void* buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request* request)
```

```c
int MPI_Recv_init(void* buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request* request)
```

Communications are then initiated with one of:

```c
int MPI_Start(MPI_Request* request)
```

```c
int MPI_Startall(int count, MPI_Request request_array[])
```

As usual with non-blocking send/recv, communications are completed with one of:

```c
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

```c
int MPI_Waitall(int count, MPI_Request request_array[], MPI_Status status_array[])
```

```c
int MPI_Test(int count, int* flag, MPI_Status* status)
```
MPI v3 – Non-blocking Advances

To check whether a message has arrived before actually receiving the message (e.g. to query the message length, then allocate the receive buffer and call MPI_Irecv()):

```c
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int* flag, MPI_Status* status)
```

- Arguments match MPI_Recv(), except that flag == 1 if the message has arrived.
- Variants for blocking call, multi-threaded versions: MPI_Probe(), MPI_Mprobe(), MPI_Improbe(); “m” versions receive with MPI_Mrecv() & MPI_Imrecv()

If you wish to cancel a pending, nonblocking communication (send or recv), call:

```c
int MPI_Cancel(MPI_Request* request)
```

- The call is local and returns immediately, so it is still necessary to call MPI_Request_free(), MPI_Wait(), MPI_Test(), etc., to check that the request has actually been canceled.
- May be used to cancel a communication from a persistent request (only the active communication, not the entire persistent request).
Free Parallel Solver Software

One of the great benefits of MPI is the prevalence of portable parallel software libraries. Here are some [free] high-quality ones you may use in your research:

Dense linear solvers and eigenvalue solvers:

- ScaLAPACK – dense and banded linear solvers and eigenvalue analysis (netlib.org/scalapack) [Fortran77; callable from C++, C, Fortran]
- PLAPACK – dense matrix operations (www.cs.utexas.edu/~plapack) [C; callable from C++, Fortran]

Sparse/iterative linear/nonlinear solvers and eigenvalue solvers:

- SuperLU Dist – direct solvers for sparse linear systems (crd.lbl.gov/~xiaoye/SuperLU) [C; callable from C++, Fortran]
- HYPRE – iterative solvers for sparse linear systems (www.llnl.gov/CASC/linear_solvers) [C; callable from C++, Fortran]
- PARPACK – large-scale eigenvalue problems (www.caam.rice.edu/software/ARPACK) [Fortran77; callable from C++, C, Fortran]
Free Parallel Solver Software

Other:

- **SUNDIALS** – nonlinear, IVP, DAE solvers w/ sensitivities ([www.llnl.gov/casc/sundials](http://www.llnl.gov/casc/sundials)) [C; callable from C++, Fortran]
- **FFTW** – multi-dimensional parallel discrete Fourier transform ([www.fftw.org](http://www.fftw.org)) [C; callable from C++, Fortran]
- **ParMETIS** – graph partitioning meshing, sparse-matrix orderings ([www.cs.umn.edu/~metis](http://www.cs.umn.edu/~metis)) [C; callable from C++]
- **PHDF5** – parallel data input/output library ([www.hdfgroup.org](http://www.hdfgroup.org)) [C; callable from C++, Fortran]
- **mpiP** – mpi profiling library ([mpip.sourceforge.net](http://mpip.sourceforge.net)) [C; callable from C++, Fortran]
- **LAMMPS** – large-scale molecular dynamics simulator ([lammps.sandia.gov](http://lammps.sandia.gov)) [C++; callable from C, Fortran, Python]

Larger parallel packages (that include or can call many of the above software):

- **PETSc** – data structures & nonlinear/linear PDE solvers ([www.mcs.anl.gov/petsc](http://www.mcs.anl.gov/petsc)) [C; callable from C++, Fortran, Python]
- **Trilinos** – enabling technologies for complex multi-physics problems ([trilinos.sandia.gov](http://trilinos.sandia.gov)) [C++; interfaces in Fortran90+, Python]
Advection Equation

In-class demo of developing a parallel solver for a 2D system of advection equations:

- MPI parallelization of our earlier serial approach,
- periodic Cartesian communicator,
- Asynchronous communication, using `MPI_Waitall()`,
- New version using `MPI_Put()` and “windows”.
Hybrid MPI+OpenMP Chemistry

In-class demo of hybrid MPI+OpenMP chemistry solver:

- MPI parallelization of our earlier serial approach:
  - Initialize MPI in thread-safe manner via `MPI_Init_thread()`
  - Static/even work distribution among MPI tasks
  - Root process assigns temperatures via `MPI_Scatterv()`

- OpenMP parallelization within MPI tasks

- Exploration of hybrid parallelism performance (on KNL workstation):
  - Pure MPI
  - Pure OpenMP
  - Hybrid MPI+OpenMP