Introduction to Parallel Computing With MPI

Lecture 1: Introduction to Parallel Computing

Philip LePoudre
philip.lepoudre@usask.ca

Outline

• Intro to High Performance Computing
  – Motivation for parallel scientific computing
  – Basic computer architecture
  – Cluster computing

• Approaches to parallelizing a problem
  – Parallel programming models.
  – Problem decomposition.
  – Domain decomposition approach.
  – Parallel efficiency and communication overhead.

Short Course Overview

• Intention is to give an introduction to the basics of how to do parallel computing using Message Passing Interface (MPI)
  – Learning curve is fairly steep, but will try to cover the basics of all steps: access to the HPC resource, programming parallel codes, compiling and building the executable, submitting and managing parallel jobs, performance evaluation

• Hands-on practice – build and run some simple parallel codes

• Give direction for further learning
Motivation
• High Performance Computing (HPC) is an exciting new tool for solving large scale computational problems in science and engineering.
• Involves the nexus of Computer Science, Mathematics, and Physics/Engineering - a new field "Scientific Computing".

Computer Science
- Computer architecture and design
- Software tools and development
- Programming and algorithm design

Mathematics
- Statement of the math problem
- Development of numerical methods
- Verification

Physics/Engineering
- The science or design problem
- Modeling
- Interpreting and applying results

Scientific Computing

High Performance Computing
• HPC promotes interdisciplinary research.
  – HPC users come from a wide range of fields in engineering and the sciences
  – Biology, chemistry, physics, mathematics, computer science, engineering, etc.
• Simulation of physical processes as a research tool.
  – Ex. Computational Aeroacoustics analysis of fan noise (show animation)

High Performance Computing
• Challenges:
  – Managing the whole process from developing the algorithm/parallel code, running large scale simulations on HPC resources, to analyzing and publishing output.
  – Research groups probably need a team approach to develop large scale parallel codes.
  – Handling the complexity of the output. Large scale simulations typically involve large and complex data sets.
  – Visualization techniques can be used to explore the data.
Visualization Resources

- Westgrid Collaboration and Visualization Facility (WGCVF)
  - http://www.usask.ca/its/services/research_computing/wgcvf.php

- SRC 3-D Virtual Reality Centre

Basic Computer Architecture

- HPC has moved to a paradigm based on multiple processors operating in parallel.

- One way of classifying modern parallel computers is by memory configuration:
  - Shared memory computers – many CPUs share the same memory and address space.

  ![Shared Memory Diagram]

  - Distributed memory computers – each CPU has own memory. CPUs are connected by some network and may exchange messages.

  ![Distributed Memory Diagram]
Basic Computer Architecture

- Combinations of both architectures. Shared memory within a node. Message passing between nodes across the interconnect.

Cluster Computing

- Create a parallel computer by assembling many off-the-shelf computers/servers in parallel and joining together with a fast interconnect.
- Very cost effective since mass-produced computer parts are cheap.
- Today this is typically done using the 64-bit Intel Xeon or AMD Opteron processors in single/dual/quad/... core configurations.
- Interconnects: Myrinet, Infiniband, Gigabit Ethernet.

Socrates

- Head Node:
  - 2 Quad Core Intel Xeon processors
  - 8 GB RAM, 1 TB storage
- 8 Capacity nodes:
  - 2 Quad Core Intel Xeon processors, 2.5 GHz
  - 32 GB RAM, 146 GB local storage
- 28 Capacity nodes:
  - 2 Quad Core Intel Xeon processors, 2.5 GHz
  - 8 GB RAM, 250 GB local storage
- 1 Gigabit Ethernet private network
- [http://www.usask.ca/its/services/research_computing/socrates.php](http://www.usask.ca/its/services/research_computing/socrates.php)
Top500 List and Trends

• The top 500 supercomputers are catalogued twice a year and details are published at www.top500.org
• Computers are ranked using the High Performance LINPACK Benchmark (solving a dense matrix system), which gives a measure of the FLOPS performance of the machine.
• The current fastest computer (Nov. 2010) is “Tianhe-1A” at National Computing Centre, Tianjin, China running at 2.566 petaflops (one petaflops is 1 quadrillion floating point operations per second)
• Intel EM64T Xeon X56xx 2.93 GHz, 186368 cores in total, Power requirement – 4.04 MW!

Top500 List and Trends

MPP – Massively Parallel Processor (no remote memory access, tightly coupled)
SMP – Symmetric Multi-Processor (uniform memory access, single address space)

Top500 List and Trends
Parallel Programming Models

1. Multithreading in Shared Memory
   - Creation of multiple threads to divide program tasks (fork-join).
   - Data Parallelism: division of independent tasks among processors.
     • eg. Adding two vectors together – addition of vector elements can occur concurrently and can be easily divided among processors.

   - Dissection of loops.
     • Loop index divided among processors.

   - This model requires shared memory computers.
   - Standard is OpenMP.
   - Some compilers also support automatic parallelization of sequential codes.
   - Many details of the communication and synchronization are hidden from the user.
Parallel Programming Models

2. Message Passing
   - Creation of multiple processes, each with their own local data.
   - Common paradigm is Single Program Multiple Data (SPMD). Each process runs the same program, but with different data.

3. Hybrid
   - OpenMP on shared memory nodes, with MPI between nodes over the interconnect.

Problem Decomposition

- Functional (Task) Decomposition
  - The program or algorithm is partitioned according to the different tasks or functions that must be performed.
  - This is not a common approach, but may help conceptualize a way to parallelize a problem.
  - Ex. Climate Model Code

Each Model is a separate task and can be assigned to a different processor.

From "Designing and Building Parallel Programs", Ian Foster
Problem Decomposition

• Domain (Data) Decomposition
  – The data set is divided among processors, and each performs computations on their piece of the pie.
  – A natural decomposition for simulations on a numerical grid – subdivide the grid evenly among processors.
  – Works with SPMD paradigm – multiple instances of the same program work on different parts of the grid.
  – Message passing is required if non-local data values are needed in the computation.
  – This approach is probably the most common in parallel scientific computing.

Domain Decomposition Approach

• Example: Finite difference on a 2D Cartesian grid

Domain Decomposition Approach

• Divide the mesh into subdomains, say over 4 processes.
• Each process only stores its data in local memory
Domain Decomposition Approach

• At edge nodes the stencil will overlap into neighboring subdomain.
  – These values will have to be obtained from neighboring process through message passing
  – Allocate space in the local array for the “subdomain ghost point” values obtained from neighbors

Local data
Data that will be sent to neighbor
Subdomain ghost points obtained from neighbor

Parallel Efficiency

The basic principle of parallel efficiency is to balance the computational work as evenly as possible across all processors.

In the domain decomposition approach, this generally means dividing the grid as evenly as possible.
Parallel Efficiency

• Example: Domain decomposition on a mesh with a multi-time-step algorithm.
  – In Computational Aeroacoustics (CAA) calculations, a multi-time-step algorithm can be used to provide greater temporal resolution in certain regions of the mesh.
  – At these points the solution is computed more frequently in time, and therefore entails a greater amount of computational work.
  – Strategies for load balance:
    • Determine required size of each subdomain so that the total amount of computation work is the same for each process.
    • Equally divide up each section of the mesh with a different time step.

Parallel Overhead

• Parallelizing a sequential algorithm adds “parallel overhead” to the total execution time, which is made up of two main components:
  – Communication time. The time required for the message passing. Depends on the total amount of communication required by the algorithm, and the speed of the interconnect.
  – Idle time. Caused by poor load balance or lack of synchronization (e.g., a process waiting to receive a message from another process).

Parallel Overhead

• Communication time depends on two properties of the communication fabric:
  – Bandwidth – the rate data can be transferred across the interconnect (Mbit/s).
  – Latency – the time it takes to initiate a message (μs).
• Bandwidth generally increases with message size to the limiting value supported by the interconnection fabric.
• Bandwidth and latency can vary between different processors on computers with a mix of shared memory and distributed memory nodes.
Parallel Efficiency

Parallel efficiency is improved if messages are made as few as possible (reduce latency costs) and larger in size to improve bandwidth (pack all data to be sent into one message).

Parallel Efficiency

- Reducing communication time means:
  - Reducing the amount of communication required by the problem decomposition and algorithm.
    - Some algorithms are much easier to parallelize than others.
  - Avoiding unnecessary communication (only pass messages when absolutely required by the algorithm).
    - It is easy to inadvertently schedule more communication than is required.
  - Reducing the number of messages.
  - Optimizing the message size.
Parallel Efficiency

- Reducing idle time means:
  - Achieving a good load balance (even division of computational work).
  - Synchronizing the processes, i.e. avoiding unnecessary waiting for messages to send or receive.
  - SPMD codes generally have good synchronization if the data (work) is evenly divided, and communication performance is uniform between all processes.

Parallel Efficiency

- Example: Finite difference on 2D Cartesian grid

- To reduce the number of subdomain ghost points make the subdomains square (or as close to square as possible).

Parallel Efficiency

- In domain decomposition approach try to reduce the surface area to volume ratio of the subdomains as much as possible to reduce the parallel overhead.
- The more subdomains (processes) used to parallelized the mesh, the greater the number of ghost points.
  - This governs the scalability and speedup of this algorithm.
Outline

- Description and history of MPI.
- Implementations available.
- Basic code structure and MPI environment.
- Communicators.
- Handles.
- Point-to-point communication.
- Blocking, buffering.
- Communication patterns and avoiding deadlock.

What is MPI?

- A library specification for a Message Passing Interface, proposed as a standard.
- Includes the definition of a library of routines for creating parallel programs in Fortran, C, and C++
- Many different implementations of the library exist.
History of MPI

- Many different message passing systems existed in the 80's and early 90's.
- A standardization effort began in 1992, and led to the release of the MPI-1 specification in 1994.
- The effort involved over 80 people from 40 organizations, mainly in the US and Europe.
- Some revisions and extensions were introduced leading to the definition of MPI-2 in 1997. This is the current standard.
- The official version of the MPI standards are available at [http://www.mpi-forum.org](http://www.mpi-forum.org)

Why Use MPI?

- The de facto message passing standard used or available on virtually all parallel computers.
- Codes written using MPI are widely portable.
  - If written correctly to the standard, your source code should work well on most computers.
- MPI can be used on a wide range of different architectures, including distributed memory and shared memory computers, and even heterogeneous clusters.
- Can achieve high performance
  - MPI implementations can be optimized for a certain architecture.

Implementations Available

- Argonne National Laboratory provides the most popular free implementation: MPICH2
- Information on MPI at:
- Other major freeware version is OpenMPI (for Linux machines)
  - [http://www.open-mpi.org/](http://www.open-mpi.org/)
- Many vendor and commercial versions available:
  - Intel, IBM, SGI, Microsoft (MS MPI based on MPICH)
Scope of MPI

- Total of 160 routines covering:
  - Point-to-point communication
  - Collective operations
  - Process groups and communication domains
  - Process topologies
  - Environmental management and inquiry
  - Profiling interface
  - Etc.
- Language bindings given for Fortran (90), C, C++ (MPI-2)
- However, only 6 routines are required for the simplest message passing codes!

Basic Code Structure (Fortran)

```fortran
program basicMPIcode
  implicit none
  include "mpif.h" ! Include the MPI header file
  integer::ierr
  Other declaration statements
  Sequential code
  call MPI_INIT(ierr) ! Initialize MPI environment
  Other parallel code
  call MPI_FINALIZE(ierr) ! Terminate MPI env.
  Other sequential code
  stop
end program basicMPIcode
```

Header File

- “mpi.h” (C programs), or “mpif.h” (Fortran) header file must be included in the program declaration and any subroutine issuing MPI calls
  - Defines MPI constants, derived types and function declarations
- Fortran: can “use MPI” module if supported by the implementation
Initialization

- Must be called before any other MPI routine in the main program

\[ \text{MPI_INIT()} \]

- Specific language bindings:

  Fort: \( \text{MPI_INIT(ierr)} \)
  
  \[ \text{integer ierr} \]

  C: \( \text{MPI_Init(int *argc, char **argv)} \)

  C++: \( \text{void MPI::Init(int& argc, char**& argv)} \)

Finalize

- Finalizes a parallel computation, after which no MPI routines can be called.

\[ \text{MPI_FINALIZE()} \]

- Language bindings:

  Fort: \( \text{MPI_FINALIZE(ierr)} \)
  
  \[ \text{integer ierr} \]

  C: \( \text{MPI_Finalize(void)} \)

  C++: \( \text{void MPI::Init()} \)

Communicators

- In MPI land, processes are each given a unique identifying number from 0 to \((\# \text{processes} - 1)\), called the process rank, in the global communicator.
- A “communicator” is a communication domain – i.e. a list of processes. The global communicator contains all processes, and is accessed with the handle “\text{MPI_COMM_WORLD}”. 
Handles

- In MPI lingo a “handle” is the way users access opaque objects in the MPI environment.
  - `MPI_COMM_WORLD` is the handle for the global communicator (details of which are stored in the MPI system memory and are opaque to the user).
- Handles are typically used in function and procedure calls.
  - In Fortran, all handles are of type `INTEGER`.
  - In C/C++ a different handle type is defined for each category of objects.

Process Identification

- One of the most basic MPI operations is to have each process identify its own rank and the total number of processes in the global communicator.

```
MPI_COMM_RANK(comm, rank)
IN comm communicator (handle)
OUT rank (integer)

MPI_COMM_SIZE(comm, size)
IN comm communicator (handle)
OUT size (integer)
```

“Hello World” MPI program (C)

```c
#include <mpi.h> /* Include the MPI header file */
#include <stdio.h>
main (int argc, char *argv[]) {
    int ierr, pid, np;
    ierr = MPI_Init(&argc, &argv); /* Initialize MPI environment */
    MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get number of processes (np)*/
    MPI_Comm_rank(MPI_COMM_WORLD, &pid); /* Get local rank (pid) */
    printf("I am process: \%d \n", pid);
    MPI_Finalize(); /* Terminate MPI environment */
}
```
Point-to-Point Communication

• A message can be passed between two processes using the basic MPI send and receive routines.

• **SEND:**

  \[\text{MPI Send}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm})\]

  - **IN buf** initial address of send buffer (choice)
  - **IN count** number of entries to send (integer)
  - **IN datatype** datatype of each entry (handle)
  - **IN dest** rank of destination (integer)
  - **IN tag** message tag (integer)
  - **IN comm** communicator (handle)

Point-to-Point Communication

• **RECEIVE:**

  \[\text{MPI_recv}(\text{buf}, \text{count}, \text{datatype}, \text{source}, \text{tag}, \text{comm}, \text{status})\]

  - **OUT buf** initial address of receive buffer (choice)
  - **IN count** max number of entries to receive (integer)
  - **IN datatype** datatype of each entry (handle)
  - **IN source** rank of source (integer)
  - **IN tag** message tag (integer)
  - **IN comm** communicator (handle)
  - **OUT status** return status (status)

Communication Buffer

• The send buffer contains \text{count} successive entries of type \text{datatype}, i.e. it is a one-dimensional array will all the send values in sequential order in memory.

• Receive buffer has at least \text{count} consecutive memory spaces available of type \text{datatype}.

• The buffer argument to the routines is the address of the first element in the buffer array
  - \text{MPI Send}(\text{buffer}(1),...
MPI Datatypes

- **datatype** is the appropriate MPI type handle

<table>
<thead>
<tr>
<th>Fortran datatype</th>
<th>MPI datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>MPI_COMPLEX</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>MPI_CHARACTER</td>
</tr>
</tbody>
</table>

C datatype

<table>
<thead>
<tr>
<th>C datatype</th>
<th>MPI datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed int</td>
<td>MPI_INT</td>
</tr>
<tr>
<td>float</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
</tr>
<tr>
<td>signed char</td>
<td>MPI_CHAR</td>
</tr>
</tbody>
</table>

C++ datatype

<table>
<thead>
<tr>
<th>C++ datatype</th>
<th>MPI datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed int</td>
<td>MPI::INT</td>
</tr>
<tr>
<td>float</td>
<td>MPI::FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>MPI::DOUBLE</td>
</tr>
<tr>
<td>Complex&lt;float&gt;</td>
<td>MPI::COMPLEX</td>
</tr>
<tr>
<td>signed char</td>
<td>MPI::CHAR</td>
</tr>
</tbody>
</table>

Receive Status

- **status** returns information about the source, tag, error code, and can be used to find the receive count if any of this information is unknown.
- **status** must be created by the user:
  - In Fortran it is an array of integers of length MPI_STATUS_SIZE
  - In C it is a structure of type MPI_Status
  - In C++ it is an object of type MPI::Status
- To avoid the hassle you can specify MPI_STATUS_IGNORE as the status argument.
Blocking vs Non-blocking Routines

- MPI_SEND() and MPI_RECV() are blocking routines.
  - The MPI_SEND() routine does not return until the message has been copied out of the buffer, and the buffer can be safely overwritten.
  - The MPI_RECV() routine does not return until the buffer actually contains the contents of the message.
- These blocking routines may cause one of the sending or receiving processes to wait until the other process issues its matching communication call (if the processes are not closely synchronized).
- MPI also provides non-blocking point-to-point communication routines to avoid this problem.

Message Buffering

- MPI implementations may use message buffering – i.e. temporarily storing the message in a system buffer until the receive call executes.
- This allows the send call to return once the message has been copied into the system buffer.
- However, the MPI standard does not define the size of the system buffer (if any), so safe MPI programs should not depend on message buffering for proper execution.
Communication Patterns

Example: Finite difference on 2D Cartesian grid. For simplicity divide evenly over 2 processes. Both processes create subdomain ghost points at interface.

Field: $T(i,j)$

$\begin{array}{c}
\text{i} \\
\text{j}
\end{array}$

$\begin{array}{c}
0 \\
1
\end{array}$

Code to fill ghost point values

```fortran
integer::rank,ierr
integer, dimension(MPI_STATUS_SIZE)::status
real, dimension(7)::buffer
real, dimension(0:6,7)::T ! Compute nodes are T(1:5,1:7)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
if (rank .eq. 0) then
  buffer=T(5,1:7) ! Explicit packing of send buffer
  call MPI_SEND(buffer(1),7,MPI_REAL,1,0, &
                MPI_COMM_WORLD,ierr)
  call MPI_RECV(buffer(1),7,MPI_REAL,1,0, &
                MPI_COMM_WORLD,status,ierr)
  T(6,1:7)=buffer ! Unpack ghost point values
end if
if (rank .eq. 1) then
  buffer=T(1,1:7)
  call MPI_SEND(buffer(1),7,MPI_REAL,0,0, &
                MPI_COMM_WORLD,ierr)
  call MPI_RECV(buffer(1),7,MPI_REAL,0,0, &
                MPI_COMM_WORLD,status,ierr)
  T(0,1:7)=buffer ! Unpack ghost point values
end if
call MPI_FINALIZE(ierr)
```
Deadlocking

- What is wrong with the previous code?
  - Both processes issue a blocking send call before issuing a receive. One of two things may happen:
    - The program deadlocks because both processes are waiting for the message to send (and no matching receives have been posted).
    - The program relies on message buffering to complete properly. One or both messages are copied to the system buffer allowing send call(s) to complete, and subsequently receive call(s) to be made. This situation is unsafe because it requires that adequate system buffer space is available.

Communication Patterns

! A safe code to pass ghost point values. The order of send and receive is switched for processes

```fortran
integer::rank,ierr
integer, dimension(MPI_STATUS_SIZE)::status
real, dimension(7)::buffer

call MPI_INIT(ierr)

if (rank .eq. 0) then
  buffer=T(5,1:7) ! Explicit packing of send buffer
  call MPI_SEND(buffer(1),7,MPI_REAL,1,0, &
                 MPI_COMM_WORLD,status,ierr)
  T(6,1:7)=buffer ! Unpack ghost point values
end if
```

if (rank .eq. 1) then
  ! First issue the receive call, then send.
  call MPI_RECV(buffer(1),7,MPI_REAL,status,0, &
               MPI_COMM_WORLD,ierr)
  buffer=T(1,1:7)
  call MPI_SEND(buffer(1),7,MPI_REAL,status,0, &
               MPI_COMM_WORLD,ierr)
end if
```

```fortran
call MPI_FINALIZE(ierr)
```

MPI_SENDRECV()

- The communication pattern above is common enough that MPI offers special support.

```c
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)
```

- **IN** sendbuf initial address of send buffer (choice)
- **OUT** recvbuf initial address of receive buffer (choice)
- **IN** (send/recv)count number of entries in buf (integer)
- **IN** (send/recv)type datatype of each entry (handle)
- **IN** dest rank of destination (integer)
- **IN** source rank of source (integer)
- **IN** (send/recv)tag message tag (integer)
- **IN** comm communicator (handle)
- **OUT** status return status (status)

MPI_SENDRECV()

- MPI_SENDRECV() is guaranteed to be safe and will not deadlock.
- This routine can be used in a number of different situations where all processes in a communicator are issuing matching send and receive calls at the same time.
- A variant is MPI_SENDRECV_REPLACE() where the send buffer is replaced with the contents of the received message (send and receive messages of equal length).

Null Processes

- It is sometimes convenient to issue a send or receive call with a dummy source or destination. This can be achieved using the handle MPI_PROC_NULL in place of source or dest.
  - An example is the 2D Cartesian grid. In general each subdomain will be surrounded on all sides by subdomain ghost points and will have to communicate with 4 neighbors. However subdomains at the edges of the global grid will have external boundary conditions on some sides (no adjacent process to communicate with). In this case, to preserve a symmetric SPMD code structure, dummy communication calls can be made at external boundaries.
Outline

• Collective communication and communicators.
• Barrier synchronization.
• Group communication functions.
• Example: LU matrix decomposition.
• Group reduction operations.
• Example: vector inner product.

Collective Communication

• MPI provides a number of routines to facilitate communication among groups of processes.
  – All processes within the group participate in the communication.
• The global group of all processes is accessed with MPI_COMM_WORLD.
• Subgroups can be created using MPI’s group and communicator management routines.
  – A new communicator can be created for subgroups as well.
  – An intracommunicator is for communication within a group. An intercommunicator is for communication between different groups.
Barrier Synchronization

- A collective call which returns only after all processes in `comm` have entered the routine.
- Provides synchronization.
- May not be needed if other blocking communication calls provide adequate synchronization.
- Liberal use of this routine can degrade performance.

Group Communication Functions

- **Broadcast:**

  - Broadcasts data from the root processes to all processes within the group.

  ![Broadcast Diagram]

- **Scatter/Gather:**

  - Scatter is the inverse operation.

  ![Scatter/Gather Diagram]

  - Gather is equivalent to each process sending the contents of its send buffer to the root. The root receives all data concatenated in rank order.

Group Communication Functions

- **Allgather:**
  - Every process receives all send buffers concatenated in rank order.

```
<table>
<thead>
<tr>
<th>A_1</th>
<th>B_1</th>
<th>C_1</th>
<th>D_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>B_2</td>
<td>C_2</td>
<td>D_2</td>
</tr>
<tr>
<td>A_3</td>
<td>B_3</td>
<td>C_3</td>
<td>D_3</td>
</tr>
</tbody>
</table>
```

Group Communication Functions

- **Alltoall:**
  - `MPI_BCAST(buffer, count, datatype, root, comm)`
    - `INOUT buffer` initial address of buffer (choice)
    - `IN count` number of entries in buffer (integer)
    - `IN datatype` data type of buffer (handle)
    - `IN root` rank of broadcast root (integer)
    - `IN comm` communicator (handle)

  - `buffer` is broadcast from `root` to all other processes in `comm`.

```
<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_1</td>
<td>B_2</td>
<td>B_3</td>
</tr>
<tr>
<td>C_1</td>
<td>C_2</td>
<td>C_3</td>
</tr>
<tr>
<td>D_1</td>
<td>D_2</td>
<td>D_3</td>
</tr>
</tbody>
</table>
```

Broadcast

```c
MPI_BCAST(buffer, count, datatype, root, comm)
```

- `INOUT buffer` initial address of buffer (choice)
- `IN count` number of entries in buffer (integer)
- `IN datatype` data type of buffer (handle)
- `IN root` rank of broadcast root (integer)
- `IN comm` communicator (handle)

- `buffer` is broadcast from `root` to all other processes in `comm`. 

```
<table>
<thead>
<tr>
<th>A_1</th>
<th>B_1</th>
<th>C_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td>B_2</td>
<td>C_2</td>
</tr>
<tr>
<td>A_3</td>
<td>B_3</td>
<td>C_3</td>
</tr>
<tr>
<td>A_4</td>
<td>B_4</td>
<td>C_4</td>
</tr>
</tbody>
</table>
```
Gather

\[
\text{MPI\_GATHER}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \\
\text{recvcount}, \text{recvtype}, \text{root}, \text{comm})
\]

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

* `recvcount` is number of elements received from each process.

Vector Variants

- Vector variants of global communication functions are available.
- Vector variant allows a varying count of data from each process, i.e. `recvcount` is now an array.

GatherV

\[
\text{MPI\_GATHERV}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \\
\text{recvcounts}, \text{displs}, \text{recvtype}, \text{root}, \text{comm})
\]

- **IN** `sendbuf` initial address of send buffer (choice)
- **IN** `sendcount` number of elements in send buffer (integer)
- **IN** `sendtype` data type of send buffer (handle)
- **OUT** `recvbuf` initial address of receive buffer (choice)
- **IN** `recvcounts` integer array of counts (integer array)
- **IN** `displs` integer array of displacements (integer array)
- **IN** `recvtype` data type of receive buffer (handle)
- **IN** `root` rank of receiving process (integer)
- **IN** `comm` communicator (handle)

* `displs(i)` gives offsets (in terms of elements of `recvtype`) of data from process `i` in `recvbuf`. 
Example: LU Matrix Decomposition

- System of linear equations:
  \[ a_1 x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1 \]
  \[ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n = b_2 \]
  \[ \vdots \]
  \[ a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n = b_m \]

- Assemble into matrix system: \( Ax = b \)
- To solve for \( x \) use a "LU" decomposition of \( A \) (i.e. Gauss Elimination)

\[ LUx = b \]
\[ Ux = L^{-1} b = y \]

Example: LU Matrix Decomposition

- Solution procedure:
  1. LU decomposition: \( A = LU \)
  2. Forward solve for \( y \): \( Ly = b \)
  3. Backward solve for \( x \): \( Ux = y \)
- Code algorithm – do steps 1 and 2 together for efficiency. This essentially involves Gauss elimination on rows until \( A \) is overwritten by \( U \) and \( b \) is overwritten by \( y \). Then back solve for \( x \) which is easy to do with upper diagonal matrix.

Example: LU Matrix Decomposition

- Ex. First step – eliminate coefficients in first column below \( a_{11} \)
  \[ a_{12} x_2 + a_{13} x_3 + \cdots + a_{1n} x_n = b_1 \]
  \[ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n = b_2 \]

- Algorithm is
  \[ a^{(1)}_{ij} = a_{ij} - a_{i1} \cdot j^{(1)}_{ij} \]
  \[ j^{(1)}_{11} = \frac{a_{11}}{a_{11}} \]

- Result:
  \[ a_{12} x_2 + a_{13} x_3 + \cdots + a_{1n} x_n = b_1^{(1)} \]
  \[ 0 + a_{22}^{(1)} x_2 + \cdots + a_{2n}^{(1)} x_n = b_2^{(1)} \]
  \[ \vdots \]

Example: LU Matrix Decomposition

- System of linear equations:
  \[ a_1 x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1 \]
  \[ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n = b_2 \]
  \[ \vdots \]
  \[ a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n = b_m \]
Example: LU Matrix Decomposition

- Elimination continues until matrix is upper triangular.
- Sequential code for forward solve:
  ```
  do k=1,n-1
    do i=k+1,n
      l=a(i,k)/a(k,k) ! Assuming a(k,k) is non-zero
      do j=k+1,n
        a(i,j)=a(i,j)-l*a(k,j)
      end do
      b(i)=b(i)-l*b(k)
    end do
  end do
  
  A "saxpy" operation which is very common. Can use an optimized routine from numerical library.
  
  Need to know the contents of the kth row before all rows below can be updated.
  Process containing kth row needs to broadcast these values to everyone else, then saxpy operation can be done in parallel.
  ```
Example: LU Matrix Decomposition

```
cnt=1  ! cnt is incremented every time a local row is fully reduced
do k=1,n  ! Increment over all global rows
  if (k .eq. myrows(cnt)) then  ! myrows contains global row indices of local rows
tmp=a(cnt,1:n+1)  ! tmp is storage buffer for pivot row, a is local augmented matrix
call MPI_BCAST(tmp(1),n+1,MPI_DOUBLE_PRECISION,  
  rank,MPI_COMM_WORLD,ierr)  ! broadcast pivot row
  cnt=cnt+1
  else
    call MPI_BCAST(tmp(1),n+1,MPI_DOUBLE_PRECISION,  
      rowalloc(k),MPI_COMM_WORLD,ierr)  ! rowalloc has rank #  
  end if
  do i=cnt,numrows  ! numrows is total number of local rows
    l=a(i,k)/tmp(k)  ! Inner saxpy operation
    a(i,k)=a(i,k)-l*tmp(i)
  end do
end do
```

Example: LU Matrix Decomposition

- Block decomposition of rows is not very efficient because once top 3 rows are reduced, P₁ will idle for rest of computation. Then P₂ will idle, etc.

```
P₁  \begin{align*}  
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 
\end{align*}
P₂
nPₙ

\begin{align*}  
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m  
\end{align*}
```

Example: LU Matrix Decomposition

- An interleaved decomposition of rows would be more efficient because all processes will have work to do until the very end of the computation.

```
P₁  \begin{align*}  
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 
\end{align*}
P₂
nPₙ

\begin{align*}  
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m  
\end{align*}
```
Group Reduction Operations

- MPI predefined reduction operations (among others):

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</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_LOR</td>
<td>logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
</tr>
</tbody>
</table>

- User-defined operations are also supported.

Group Reduction Operations

- Reduce:

  data
  \[
  \begin{array}{ccc}
  A_1 & B_1 & C_1 \\
  A_2 & B_2 & C_2 \\
  \end{array}
  \]

  reduce
  \[
  \begin{array}{c}
  \text{(MPI\_SUM)}
  \end{array}
  \]

  data
  \[
  \begin{array}{ccc}
  A_1+A_1+A_1 & B_1+B_2+B_2 & C_1+C_2+C_2 \\
  \end{array}
  \]

  processes

- Root process receives the element-wise reduction.

Group Reduction Operations

- Allreduce:

  data
  \[
  \begin{array}{ccc}
  A_1 & B_1 & C_1 \\
  A_2 & B_2 & C_2 \\
  \end{array}
  \]

  allreduce
  \[
  \begin{array}{c}
  \text{(MPI\_SUM)}
  \end{array}
  \]

  data
  \[
  \begin{array}{ccc}
  A_1+A_1+A_1 & B_1+B_2+B_2 & C_1+C_2+C_2 \\
  A_1+A_1+A_1 & B_1+B_2+B_2 & C_1+C_2+C_2 \\
  A_1+A_1+A_1 & B_1+B_2+B_2 & C_1+C_2+C_2 \\
  \end{array}
  \]

  processes

- All processes receive element-wise reduction.
Group Reduction Operations

- Reduce-scatter:

<table>
<thead>
<tr>
<th>Processes</th>
<th>data</th>
<th>reduce-scatter</th>
<th>processes</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td>(MPI_SUM)</td>
<td>A&lt;sub&gt;i&lt;/sub&gt; + A&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td></td>
<td>B&lt;sub&gt;i&lt;/sub&gt; + B&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td></td>
<td>C&lt;sub&gt;i&lt;/sub&gt; + C&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- Reduction plus a scatter over processes.

- Scan:

<table>
<thead>
<tr>
<th>Processes</th>
<th>data</th>
<th>scan</th>
<th>processes</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td>(MPI_SUM)</td>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td></td>
<td>B&lt;sub&gt;i&lt;/sub&gt; + B&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>A&lt;sub&gt;i&lt;/sub&gt;</td>
<td>B&lt;sub&gt;i&lt;/sub&gt;</td>
<td>C&lt;sub&gt;i&lt;/sub&gt;</td>
<td></td>
<td>C&lt;sub&gt;i&lt;/sub&gt; + C&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Reduce

```c
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
```

- `sendbuf` initial address of send buffer (choice)
- `recvbuf` initial address of receive buffer (choice)
- `count` number of elements in send buffer (integer)
- `datatype` datatype of send buffer (handle)
- `op` reduce operation (handle)
- `root` rank of root process (integer)
- `comm` communicator (handle)

- `root` is the process which receives the output of the reduction operation.
- `recvbuf` is only significant on the root process.
- `datatype` must be compatible with `op`.
Example: Vector Dot Product

- Dot product c = a * b.
- Sequential code:
  ```fortran
  integer::i
  real::c
  real, dimension(1000)::a, b
  
  c=0.0
  do i=1,1000
    c=c+a(i)*b(i)
  end do
  ```

Example: Vector Dot Product (Parallel)

```fortran
integer::i,ierr
real::c,sum
real, dimension(n)::a,b ! n local elements. Distribute evenly for good load balance.

! Local sum
sum=0.0
do i=1,n
  sum=sum+a(i)*b(i)
end do

! Global sum
call MPI_REDUCE(sum,c,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)

! Only root has value of c. (or use MPI_ALLREDUCE)
```

MPI Routines/Bindings on Web

- [http://www.lam-mpi.org/tutorials/bindings/](http://www.lam-mpi.org/tutorials/bindings/)
- [http://www.nersc.gov/vendor_docs/ibm/pe/am106_mst02.html](http://www.nersc.gov/vendor_docs/ibm/pe/am106_mst02.html)
Introduction to Parallel Computing With MPI
Lecture 4: Parallel Performance Metrics and Optimization

Philip LePoudre
philip.lepoudre@usask.ca

Outline
• Speedup and efficiency.
• Performance measurement.
• MPI_WTIME
• Ping-Pong.
• MPI tracing.
• Tips for improving code performance.

Parallel Performance Metrics
• Parallel performance is measured in terms of speedup and efficiency.
• Speedup: 
  \[ S_{rel} = \frac{T_1}{T_p} \]  
  - \( T_1 \) - execution time on one processor
  - \( T_p \) - execution time on \( P \) processors
• This is speedup relative to the parallel code running on one process.
• A tougher metric is absolute speedup which is w.r.t. the fastest sequential algorithm.
  \[ S_{abs} = \frac{T_1}{T_{seq}} \]  
  - \( T_{seq} \) - execution time of fastest sequential algorithm
Parallel Performance Metrics

- Speedup shows the scalability of the parallel program, i.e., how effectively the program uses larger numbers of processors.
  - Scalability with fixed problem size: How many processes can I use to solve my problem while maintaining an acceptable level of efficiency? What is the maximum obtainable speedup?
  - Scalability with scaled problem size: As I increase the problem size, how many more processes can I use while maintaining the same efficiency?

- Efficiency is defined as
  \[ E = \frac{S}{P} \]
  - Efficiency generally decreases with increasing processor count. It is a measure of the parallel overhead (communication and idle time).

Speedup of a 3D LES code using a Multigrid solver. Symbols indicate different numbers of grid points. Results were obtained on a 32 node Beowulf cluster running IBM PIII-866 MHz processors with 100 Mbps Ethernet (2003).
Parallel Performance Metrics

Efficiency of 3D LES code versus number of grid points.

Note efficiency greater than 1 for large problem size and small P.

Parallel Performance Metrics

- Efficiency and speedup are functions of problem size.
- For a fixed number of processors, as the problem size is increased the ratio of computation time to communication time increases, hence efficiency increases.
  - Example: domain decomposition of 2D Cartesian mesh. As the total number of mesh points is increased (fixed # of processes) the ratio of subdomain ghost points to computation points decreases.

Speedup Anomalies

- Some programs may exhibit super-linear speedup over moderate numbers of processes.
- This is usually a cache effect:
  - As more processors are used, the amount of local data is reduced and a greater proportion of the data is stored in cache memory (fast access).
  - Fast memory access reduces computation time, which may more than offset parallel overhead.
Code Performance Measurement

• Experimental measurement of your code’s efficiency and speedup is necessary to demonstrate efficient use of resources.

• To get an accurate measurement consider:
  – Whether your algorithm is deterministic or nondeterministic. A nondeterministic algorithm (like a search algorithm) may have variable runtime, which nullifies speedup results.
  – Timer accuracy. MPI provides timing routines.
  – Startup and shutdown costs such as allocation of memory, read/write to file, etc. may vary significantly with the state of the system, so just time the main algorithm loop.

Code Performance Measurement

– Interference from other programs. On shared systems other users may compete for shared resources such as processors and network bandwidth. System functions may also consume resources.
  • If possible, specify that resources are dedicated to your job.
  – Random resource allocation. The job scheduler may randomly distribute cores over the system and communication costs between cores may vary (i.e. sockets vs. nodes).
    • If possible, select which part(s) of the machine you want your job to run on.

MPI Timing Function

MPI_WTIME()

double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()

double MPI::Wtime()

• Returns a floating point number of seconds, representing elapsed wall-clock time from some fixed time in the past (guaranteed not to change over the life of the process).
• Two calls to MPI_WTIME can be used to determine elapsed run time of your algorithm or loop.
MPI Timing Function

• Example:

{ double stime, etime;
  stime = MPI_Wtime();
  ...Your algorithm...
  etime = MPI_Wtime();
  printf("Run time is %f seconds\n", etime-stime);
}

MPI Timing

• Returns resolution of MPI_WTIME in seconds, i.e. the number of seconds between clock ticks (if clock is incremented every millisecond, result is 10^3).

Ping-Pong (Xing Cai, Dept. Informatics, U. of Oslo)

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

#define NUMBER_OF_TESTS 10

int main(int argc, char **argv)
{
  double *buf;
  int rank;
  double ti, t2, time;
  int i, j, k, nlep;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
    printf("Kind of time (sec): \%f\n", time / (double)NUMBER_OF_TESTS);
  for (i=1; i<=NUMBER_OF_TESTS; i++)
  { sleep = 1000;
    buf = (double *)malloc( n * sizeof(double) );
  }
Tracing MPI Programs

- Tracing a parallel program can reveal issues with communication/idle time and synchronization.
Tracing MPI Programs

• Many cluster tools packages contain software for tracing MPI routines. The output is a record of each MPI call and the timing of the call.
• MPICH2 comes with the MPI Parallel Environment (MPE) libraries for profiling MPI calls. The profiling libraries, an event viewer (Jumpshot), and other tools are available for download at

Tracing MPI Programs

• Submit job with –trace flag
  – mpiexec –trace myapp.exe
• An event log is generated which must be processed in order to view with Jumpshot (or other viewer).

Jumpshot
Improving Code Performance

- Reduce operation count as much as possible (especially in routines/loops which are called most frequently).
  - Four multiplies, three adds (7 floating pt ops):
    
    ```
    do i=1,1000
      result(i)=a*b(i)+a*c(i)+a*d(i)+a*e(i)
    end do
    ```

  - Three adds, one multiply (4 floating pt ops):
    
    ```
    do i=1,1000
      result(i)=a*(b(i)+c(i)+d(i)+e(i))
    end do
    ```

- Avoid divide and square root operations as much as possible.
  - IBM Power4 processor functional units: divides and square roots take about 30 times as long as multiplies/adds!
  - Repeated dividing inside loop:
    
    ```
    do i=1,1000
      result(i)=(b(i)+c(i)+d(i)+e(i))/a
    end do
    ```

  - Instead, multiply by the inverse
    
    ```
    a_inv=1.0/a
    do i=1,1000
      result(i)=(b(i)+c(i)+d(i)+e(i))*a_inv
    end do
    ```

- Use cache memory effectively. This generally means ordering computations to use consecutive values in memory.

  Memory lines are loaded into cache where they can be accessed more quickly. Your algorithm should make use of the values in the cache before reloading it. (Use a small stride through memory)
Improving Code Performance

- FORTRAN allocates column-major arrays, (columns stored consecutively in memory), while C uses row-major.
  - Poor memory use in FORTRAN program:
    ```fortran
    do i=1,1000
      do j=1,1000
        A(i,j)=c*B(i,j) ! going across a row, so large stride in memory (1000)
      end do
    end do
    ```
  - Instead, switch the loops:
    ```fortran
    do j=1,1000
      do i=1,1000
        A(i,j)=c*B(i,j) ! using consecutive memory locations
      end do
    end do
    ```

- Use compiler optimization!
- Consider using optimized math libraries for common operations instead of writing your own code.
  - BLAS, LAPACK, IMSL
  - Many compilers come with optimized math libraries.

Introduction to Parallel Computing With MPI

Lecture 5: Introduction to the MPI Library III

Philip LePoudre
philip.lepoudre@usask.ca
Outline

- Non-blocking point-to-point communication.
- Derived types.
- Communicators and groups.
- Process topologies.

Non-Blocking Point-to-Point

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of entries to send (integer)
IN datatype datatype of each entry (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request request handle (handle)

- This routine will return immediately, but buf may not be sent yet.
- request is used to test for completion.

Non-Blocking Point-to-Point

MPI_IRecv(buf, count, datatype, source, tag, comm, request)

OUT buf initial address of receive buffer (choice)
IN count number of entries to send (integer)
IN datatype datatype of each entry (handle)
IN source rank of source (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request request handle (handle)

- This routine will return immediately, but buf may not be filled with the receive message.
Completion Operations

MPI_TEST(request, flag, status)
INOUT request request handle (handle)
OUT flag true if operation completed (logical)
OUT status status object (Status)

- MPI_TEST returns flag = true if the communication operation (either send or receive) has completed.
- Upon return, request is set to MPI_REQUEST_NULL
  - C: request is of type MPI_Request
  - FORTRAN: request is of type INTEGER

Completion Operations

MPI_WAIT(request, status)
INOUT request request handle (handle)
OUT status status object (Status)

- This routine returns only when the communication indicated by request (either send or receive) has completed.
- Upon return, request is set to MPI_REQUEST_NULL

Non-Blocking Communication

real, dimension(100)::buffer
integer::ierr,request,status(MPI_STATUS_SIZE)
logical::flag
...
call MPI_ISEND(buffer(1),100,MPI_REAL,1,0,
  MPI_COMM_WORLD,request,ierr)
...! do other computation (but don’t touch buffer)
call MPI_TEST(request,flag,status,ierr)
if (.not. flag) then
  ...! continue with other computation
end if
! all done other computation so wait if send has not yet completed
call MPI_WAIT(request,status,ierr)
Derived Types

- User-defined derived types is a powerful feature in MPI.
- A derived type specifies a sequence of primitive types and a sequence of integer displacements.
- It is used to send a message located over non-contiguous memory locations.
  - The derived type is specified in place of the MPI datatype handle: call MPI_SEND(buf,count,DERIVED_TYPE,...)

Derived Type Constructor:

```c
MPI_TYPE_INDEXED(count,blocklengths,displacements,oldtype,newtype)
```

| IN  | count   | number of blocks (integer) |
| IN  | blocklengths | number of elements per block (integer array) |
| IN  | displacements | displacement for each block (integer array) |
| IN  | oldtype     | old datatype (handle) |
| OUT | newtype     | new datatype (handle) |

- Specifies a noncontiguous layout of data in memory
  - `blocklengths` contains size of each block (in elements of `oldtype`), and `displacements` contains location of each block (number of elements from start).

Derived Types

Locations in memory
(of size `oldtype`)

- `oldtype`
- `newtype` (of size `oldtype`)

- `count = 5`
- `blocklengths = (4,1,1,2,1)`
- `displacements = (0,5,7,10,13)`
Derived Types

- **oldtype** may be a primitive type (MPI_REAL, etc.), or another derived type.
- **newtype** is the handle that can be used in communication calls with the derived type.
  - C: **newtype** is of type MPI_Datatype
  - FORTRAN: **newtype** is of type INTEGER
- If all blocks are the same length (e.g., 1), then can use routine MPI_TYPE_CREATE_INDEXED_BLOCK()

---

Derived Types

- **newtype** needs to be committed before it can be used. Call:
  ```
  MPI_Type_COMMIT(newtype)
  INOUT newtype datatype (handle)
  ```
- To deallocate a datatype object:
  ```
  MPI_Type_FREE(datatype)
  INOUT datatype datatype (handle)
  ```

---

Derived Types

- In general different derived types need to be defined at sending and receiving processes.
  - Send process gathers data from a sequence of noncontiguous local memory locations and sends.
  - Receive process takes message and spreads data where it needs to go in its local memory (likely a different pattern).
- But size of the derived types needs to match at both ends of the call (total amount of data sent equals total amount of data received).
- Typically communications calls using derived types are sent with count = 1
  - call MPI_SEND(buf,1,myType,...)
Derived Types Example

```fortran
real, dimension(100,100)::gT
integer::mytype,i,itemp
integer, dimension(50)::blengths,disps

! Create a derived type to send % block of gT
blengths=50
itemp=0
do i=1,50
   disps(i)=itemp
   itemp=itemp+100
end do

! This call will send % of gT starting at gT(1,1) (i.e. Fortran array is column-major)
call MPI_SEND(gT(1,1),1,mytype,1,0,MPI_COMM_WORLD,ierr)
```

Communicators and Groups

- A group is an ordered set of processes, each assigned an integer rank (starting from zero).
- A communicator specifies a communication domain, which includes a group and information to facilitate communication between all members of the group.
- The group associated with a communicator can be accessed (via a handle):

  ```fortran
  MPI_COMM_GROUP(comm,group)
  ````

- A new group can be constructed using:

  ```fortran
  MPI_GROUP_INCL(group,n,ranks,newgroup)
  ```

  * newgroup contains the n processes listed in ranks (new rank numbers assigned in the order processes were listed).
Communicators and Groups

• A new intracommunicator can be created for a specified subgroup:

```c
MPI_COMM_CREATE(comm, group, newcomm)
```

- `comm` communicator (handle)
- `group` a group which is a subset of the group of `comm` (handle)
- `newcomm` new communicator (handle)

- `newcomm` can then be used as the communicator handle for point-to-point or collective communications among the processes in `group`.

Communicators and Groups

• A mechanism for dividing a communication domain into sub-communication domains.

```c
MPI_COMM_SPLIT(comm, color, key, newcomm)
```

- `comm` communicator (handle)
- `color` control of subset assignment (integer)
- `key` control of rank assignment (integer)
- `newcomm` new communicator (handle)

- Partitions the group of `comm` into disjoint subgroups, one for each value of `color`.
- Within each subgroup, processes are ranked in an order defined by the value of `key`.

Process Topologies

• MPI has mechanisms for defining process topologies (grid or graph) and adding this information to an intracommunicator.
• The topological arrangement of processes (eg. the way processes are assigned to subdomains on a grid) must be known in order to facilitate the necessary communication patterns.
• Since Cartesian topologies are common, MPI has a number of routines for this case.
Cartesian Topology

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN comm_old input communicator (handle)
IN ndims number of dimensions of Cartesian grid (integer)
IN dims integer array of size ndims specifying number of processes in each dimension (integer array)
IN periods logical array of size ndims specifying whether grid is periodic (true) or not (false) in each dimension (logical array)
IN reorder re-order ranks (true) or not (false) (logical)
OUT comm_cart communicator with Cartesian topology (handle)

Cartesian Topology Example

<table>
<thead>
<tr>
<th></th>
<th>0 (0,0)</th>
<th>1 (0,1)</th>
<th>2 (0,2)</th>
<th>3 (0,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>1</td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(2,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ndims = 2, dims = (3,4)
- If reorder is false then rank numbers are identical to rank in comm_old, otherwise the function may reorder the processes.
- periods can be used to define the topology of a ring, cylinder, torus, etc.

Cartesian Inquiry Functions

MPI_CART_GET(comm_cart, maxdims, dims, periods, coords)

IN comm_cart Cartesian communicator (handle)
IN maxdims length of vectors dims, periods, coords (integer)
OUT dims number of processes in each dimension (integer array)
OUT periods periodicity for each dimension (logical array)
OUT coords coordinates of calling process in Cartesian structure (integer array)

- Returns position in the Cartesian topology (coords) of the calling process as well as total number of processes in each dimension and periodicity.
Cartesian Inquiry Functions

MPI_CART_RANK(comm_cart, coords, rank)
IN comm_cart Cartesian communicator (handle)
IN coords Cartesian coordinates of a process (integer array)
OUT rank rank of specified process (integer)

• Translates coordinates to rank.

Cartesian Inquiry Functions

MPI_CART_COORDS(comm_cart, rank, maxdims, coords)
IN comm_cart Cartesian communicator (handle)
IN rank rank of process in comm_cart (integer)
IN maxdims length of vector coords (integer)
OUT coords coordinates of specified process (integer array)

• Translates rank number to coordinates in the Cartesian topology.

Cartesian Shift Functions

MPI_CART_SHIFT(comm_cart, direction, disp, source, dest)
IN comm_cart Cartesian communicator (handle)
IN direction coordinate dimension of shift (integer)
IN disp displacement of shift (integer)
OUT source rank of source process (integer)
OUT dest rank of destination process (integer)

• This routine outputs the source and destination process ranks (eg. as input to MPI_SENDRECV()) for a communication shift in the specified coordinate dimension.
• Displacement is >0 for upwards shift, <0 for downwards shift.
References

• “Designing and Building Parallel Programs”, Ian Foster, Addison-Wesley, 1994.