Introduction to Parallel Programming with MPI

Peter Pacheco
University of San Francisco

April 20, 1999

Outline

- Overview and Introduction
- Point-to-point Communication: Dot Product I
- Collective Communication: Dot Product II
- User-defined Types: Matrix Multiplication I
- Communicators and Topologies: Matrix Multiplication II
- Performance
- The Rest of MPI

Introduction

- MPI: a Library of Functions for C, C++, Fortran.
  - MPI-1.0 1994, MPI-1.1 1995, MPI-2 1997
  - MPI-1: 125 functions; MPI-2 150 functions.
- Basic Programming Model: Message Passing.
  - Each process has its own memory
  - Processes communicate by explicitly calling Send or Receive functions.
  - MPI-1: static process model
- Pros: very portable, very fast
- Cons: extremely difficult to program
- Examples in C

Dot Product I

- Vectors distributed by blocks:
  - n component vectors, p processes
  - Assume n divisible by p, $\bar{n} = n/p$
  - qth process gets components

$$q\bar{n}, q\bar{n} + 1, \ldots, (q+1)\bar{n} - 1$$

- Each process computes dot product of components it owns
- "Local" dot products added together by one process

Dot Product I: Main Program

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

int main(int argc, char* argv[]) {
    float *local_x, *local_y, dot;
    int    n, n_bar, p, my_rank;

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

    n = Get_order(my_rank, p);
    n_bar = n/p;
    Allocate_and_read_vectors(&local_x, &local_y, n_bar, my_rank, p);

    dot = Parallel_dot(local_x, local_y, n_bar, my_rank, p);
    if (my_rank == 0) printf("The dot product is %f\n", dot);

    Free_vectors(&local_x, &local_y);

    MPI_Finalize();
    return 0;
}  /* main */
```

- Include `mpi.h` to get function declarations, const defns, etc.
- All MPI functions prefixed with the string "MPI_"
- Call MPI_Init first. Sets up storage, variables, etc.
- *Communicator* a collection of processes that can exchange messages. MPI_Init initializes MPI_COMM_WORLD: all processes started when program execution begins.
- MPI_Comm_size returns number of processes in a comm.
- If there are p processes in a comm, each assigned a unique nonnegative integer rank in range $0, \ldots, p-1$. MPI_Comm_rank returns the process rank.
- MPI_Finalize shuts down MPI. Frees memory, terminates pending ops.

Dot Product I: Parallel_Dot Function

```
float Parallel_dot(float local_x[], float local_y[], int n_bar,
          int my_rank, int p) {
    float      local_dot = 0.0, dot = 0.0;
    int        source, i; MPI_Status status;

    for (i = 0; i < n_bar; i++)
        local_dot += local_x[i]*local_y[i];

    if (my_rank == 0) {
        dot = local_dot;
        for (source = 1; source < p; source++) {
            MPI_Recv(&local_dot, 1, MPI_FLOAT, source, 0,
                     MPI_COMM_WORLD, &status);
            dot += local_dot;
        }
    } else {
        MPI_Send(&local_dot, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    }
    return dot;
} /* Parallel_dot */
```

- First compute local dot product.
- Next:
  - Processes $\neq 0$ send local dot to 0
  - Process 0 receives local dots and keeps running sum
- Result significant only on process 0
- Structure typical of SPMD programs

Syntax of MPI_Send and MPI_Receive

```
int MPI_Send(
        void*           message       /* in */,
        int             size          /* in */,
        MPI_Datatype    type          /* in */,
        int             destination   /* in */,
        int             tag           /* in */,
        MPI_Comm        communicator  /* in */);

int MPI_Recv(
        void*           message       /* out */,
        int             size          /* in  */,
        MPI_Datatype    type          /* in  */,
        int             source        /* in  */,
        int             tag           /* in  */,
        MPI_Comm        communicator  /* in  */,
        MPI_Status*     status        /* out */);
```

- Return values are error codes in C. Error code returned in an argument.

- First argument pointer to block of memory containing data to be sent or block into which data should be received.

- Second argument gives no. of elements in block

- Third argument gives MPI type of elements in message.

- Examples: Use size = 1 and type = MPI_FLOAT in Parallel_Dot. To send an array of 10 ints, use size = 10 and type = MPI_INT.

- Basic MPI types correspond to standard scalar types in C, e.g., MPI_CHAR, MPI_DOUBLE, MPI_LONG.

- Also possible to build complex, structured MPI types.

- Destination rank of process to which message is sent. Source is rank of process from which message is sent.

Tags and Communicators

- Tag a nonnegative integer. Standard guarantees at least 0, 1, . . . , 32767.

- Example: Process 1 sending several floats to process 0. Some to be added into running sum. Some to be printed. Use tags to differentiate.

- Example: Program uses library A to solve differential equations and library B to solve linear systems. Both libraries need to send messages. MPI Solution: communicator.

- Communicator consists of process group and *context* — system-defined tag.

- Messages sent using one communicator cannot be received by process using another communicator.

Status

- Message consists of data and envelope.
- Envelope contains: rank of destination, rank of source, tag, communicator.
- For recipient, source and tag can be wildcards: `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- Source/tag wildcard: get received source/tag with `status`: `status.MPI_SOURCE`, `status.MPI_TAG`.
- For Receive, `size` argument is size of memory block referenced by message argument: received message can be smaller.
- Can get size of received message with call to `MPI_Get_count` function.

- Syntax of MPI_Get_count:

```
int MPI_Get_count(
        MPI_Status*   status    /* in  */,
        MPI_Datatype  datatype  /* in  */,
        int*          count     /* out */);
```

Semantics

- `MPI_Send, MPI_Recv` *blocking* send, receive: function completes, arguments can be modified.

- MPI also has *nonblocking* send/receive: `MPI_Isend, MPI_Irecv.` Calls start communication operation. Arguments cannnot be reused until communication operation completed with call to `MPI_Wait.` Allows overlapping of communication and computation:

```
MPI_Irecv( . . . );
/* Do local calculations */
    .
    .
    .
MPI_Wait( . . . );
```

- `MPI_Send, MPI_Recv` *standard mode* send, receive: buffering of messages up to system.

- MPI also has `MPI_Ssend` and `MPI_Bsend:` *synchronous* mode send and *buffered* mode send.

- Synchronous send completes after matching receive has started.

- Buffered send buffers message in user-defined buffer (Slow!).

I/O

- *Worst* feature of MPI-1: no I/O specification.
- Most implementations allow process 0 access to `stdout`
- Many implementations allow all processes access to `stdout` and `stderr`.
- Many implementations allow process 0 access to `stdin`
- MPI-2 does specify I/O. Not widely available.
- Our assumption: process 0 has access to `stdin`, `stdout`. (All processes have access to `stderr`.)

Dot Product I: Get_order Function

```
int Get_order(int my_rank, int p) {
    int n, dest;
    MPI_Status status;

    if (my_rank == 0) {
        printf("Enter the order of the vectors\n");
        scanf("%d", &n);
        for (dest = 1; dest < p; dest++)
            MPI_Send(&n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
    return n;
}  /* Get_order */
```

Dot Product I: Read_vector Function

```
void Read_vector(char* prompt, float local_v[], int n_bar,
         int my_rank, int p) {
    int i, q; float* temp; MPI_Status status;

    if (my_rank == 0) {
        Allocate_vector(&temp, n_bar, my_rank, "temp");
        printf("Enter %s\n", prompt);
        for (i = 0; i < n_bar; i++)
            scanf("%f", &local_v[i]);
        for (q = 1; q < p; q++) {
            for (i = 0; i < n_bar; i++)
                scanf("%f", &temp[i]);
            MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
        }
        free(temp);
    } else {
        MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            &status);
    }
}  /* Read_vector */
```

Collective Communication

- Three functions in Dot Product I use MPI communications. All processes involved in each instance:
  - Parallel_Dot, "global sum"
  - Get_order, "broadcast"
  - Read_vector, "scatter"
- Structured communications provide great opportunities for optimization: e.g., tree structured broadcast
- Let system designer optimize: many MPI functions for *collective* communication — communication in which all processes in communicator participate.

Dot Product II: Parallel_Dot and Get_order Functions

```
float Parallel_dot(float local_x[], float local_y[], int n_bar) {
    int        i;
    float      local_dot = 0.0, dot = 0.0;

    for (i = 0; i < n_bar; i++)
        local_dot += local_x[i]*local_y[i];

    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT, MPI_SUM, 0,
        MPI_COMM_WORLD);

    return dot;
} /* Parallel_dot */



int Get_order(int my_rank) {
    int n;

    if (my_rank == 0) {
        printf("Enter the order of the vectors\n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    return n;
}  /* Get_order */
```

- In Reduce, note that MPI prohibits aliasing of arguments.

- Many predefined ops — e.g. max, min, product, logical ops, bitwise ops. User can also define his own ops.

- Also note that no tags. This was done mainly to simplify implementation of MPI coll comms. Effect is that if there are a sequence of, e.g., bcasts, then they are matched in the order in which they are started.

- Semantics same as standard send/receive: complete as soon as it is safe to reuse args. Other processes may not have even started.

- Exception MPI_Barrier: returns only after all members in communicator have started call.

Dot Product II: Parallel_read_vector

```
void Read_vector(char* prompt, float local_v[], n_bar,
         int my_rank, int n) {
    int i;
    float* temp;

    if (my_rank == 0) {
        Allocate_vector(&temp, n, my_rank, "temp");
        printf("Enter %s\n", prompt);
        for (i = 0; i < n; i++)
            scanf("%f", &temp[i]);
    }

    MPI_Scatter(temp, n_bar, MPI_FLOAT, local_v, n_bar, MPI_FLOAT,
        0, MPI_COMM_WORLD);

    if (my_rank == 0) free(temp);
}  /* Read_vector */
```

- Once again can't alias arguments.

- Note that sendbuffer count is *not* the size of the sendbuffer: it's the amount of data going to any one process.

- Note that we need a fullsize (n) temp vector to do this. For the other implementation temp only needed local size temp vector (n/p).

Some Additional Collective Communication Functions

- MPI_Barrier: blocks until all processes have entered
- MPI_Gather: gathers the components of a distributed structure onto a single process
- MPI_Allgather: gathers the components of a distributed structure onto all processes
- MPI_Alltoall: each process scatters the contents of a structure to the other processes.
- MPI_Allreduce: result of operation is returned on all processes.

Derived Datatypes

- Derived datatype: variable of type `MPI_Datatype` that can represent structured data

- Created to exploit hardware scatter/gather (older systems pack and unpack structured data)

- Essentially four types of derived datatype constructors in MPI:
  - `MPI_Type_contiguous:` build a datatype from a block of contiguous array entries
  - `MPI_Type_vector:` build a datatype from a sequence of uniformly spaced array entries (e.g, a column in a C array or a row in a Fortran array).
  - `MPI_Type_indexed:` build a datatype from a sequence of irregularly spaced array entries.
  - `MPI_Type_struct:` build a datatype from an arbitrary collection of memory locations of arbitrary types.

- To specify: need type and relative location in memory of each element to be used in the communication function.

Matrix-Matrix Multiplication I

- Matrices distributed by block rows.
  - Matrix order n, p processes
  - Assume n evenly divisible by p, $\bar{n} = n/p$
  - Process q assigned rows

$$q\bar{n}, q\bar{n} + 1, \ldots, (q+1)\bar{n} - 1$$

- Gather block of $\bar{n}$ columns onto each process.
- Each process forms dot product of its rows with the gathered columns.
- Repeat preceding two steps for each successive block of $\bar{n}$ columns.
- Local submatrices stored as linear arrays in row-major order.

Matrix-Matrix Multiplication I: Parallel_matrix_mult

```
void Parallel_matrix_mult(float local_A[], float local_B[],
         float local_C[], int n, int n_bar, int p) {
    float*        B_cols;
    MPI_Datatype  gather_mpi_t;
    int           block;

    Allocate_matrix(&B_cols, n, n_bar, "B_cols");
    MPI_Type_vector(n_bar, n_bar, n, MPI_FLOAT, &gather_mpi_t);
    MPI_Type_commit(&gather_mpi_t);

    for (block = 0; block < p; block++) {
        MPI_Allgather(local_B + block*n_bar, 1, gather_mpi_t,
            B_cols, n_bar*n_bar, MPI_FLOAT, MPI_COMM_WORLD);
        Matrix_mult( local_A, B_cols, local_C, n_bar,
            n, block);
    }
    free(B_cols);
    MPI_Type_free(&gather_mpi_t);
}  /* Parallel_matrix_mult */
```

Block Row

0	0	1	1	2	2	3	3
0	0	1	1	2	2	3	3

Table illustrates a block if $n = 8$ and $p = 4$. During first stage, elements marked 0 are gathered. During second elements marked 1 are gathered, etc.

- We don't want to overwrite B. So we allocate a block of order $n \times \bar{n}$ to store the column block.

- Observe that on any process, the array entries that it contributes to the column block are not contiguous. The entries are grouped into subblocks of size $\bar{n}$ and there are $\bar{n}$ of them. Between the starts of successive rows in any column block, there are n elements.

- Thus, we use the following arguments to MPI_type_vector.
  - First argument is the number of rows or sublocks contributed by the process: $\bar{n}$
  - Second argument is the number of contiguous elements to take from a row or sublock: $\bar{n}$
  - Third argument is the number of elements between the starts of successive blocks: n.
  - Fourth argument is the type of the elements.
  - Fifth argument is storage for the new type.

- After creating the type, with the call to MPI_Type_vector, before the type can be used in communication, it has to be *committed.* This allows the system to make optimizations that wouldn't be necessary if the type were only being used to make a more complex type.

- The Allgather uses the address of the start of the block as its first argument. The count is only 1, since `gather_mpi_t` specifies the entire block.

- Note that in the column block, the entries contributed by a process are contiguous. Hence we just use a count of $\bar{n}^2$ and a type of MPI_Float. This says the received elements will be copied into a contiguous sequence of locations in the destination array.

Matrix-Matrix Multiplication II: Fox's Algorithm

- Matrices distributed by square blocks
  - Matrices square, order n.
  - Number of processes, p, a perfect square, $\sqrt{p} = q$.
  - Processes form a virtual mesh of order $q \times q$.
  - n evenly divisible by q, $n/q = \bar{n}$.
  - Each process assigned a $\bar{n} \times \bar{n}$ submatrix

- Algorithm;
  - Broadcast diagonal block of A across process row.
  - Multiply broadcast block of A by block of B.
  - Carry out "round-the-corner" shift of blocks of B up each process column.
  - Repeat preceding steps each time broadcasting the block of A "to the right" of the preceding block.

Communicators

- Recall: Communicator a group of processes together with a unique, system-defined tag – a context.

- Any collection of MPI processes can be combined into a single group and made a communicator.

- Many functions for creating groups and communicators.

- Most useful: `MPI_Comm_split`

```
int MPI_Comm_split(
        MPI_Comm    old_comm   /* in */,
        int         split_key  /* in */,
        int         rank_key   /* in */,
        MPI_Comm*   new_comm   /* in */);
```

- In Fox's algorithm, we can form a new communicator corresponding to each process row by executing the following code;

```
int my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
    &my_row_comm);
```

- A new communicator is formed for each value of split_key: processes with the same split_key are assigned to the same communicator.

- Ranks in the new communicator are determined by rank_key.

Topologies

- MPI provides facilities for associating or *caching* additional information with a communicator.
- Most important example: process topologies.
- In MPI topology provides mechanism for associating different addressing schemes with processes.
- Important example: processes organized into an N-dimensional virtual mesh. Process topology allows processes to be addressed by coordinates.
- Two types of topologies: cartesian grids and graphs

Topologies: Creating Grids for Fox's Algorithm

```
void Setup_grid(MPI_Comm* grid_comm, MPI_Comm* row_comm,
         MPI_comm* col_comm, int* my_row, int* my_col
         int* my_grid_rank) {
    int old_rank, dimensions[2], wrap_around[2];
    int coordinates[2], free_coords[2], p, q, old_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    q = (int) sqrt((double) p);
    dimensions[0] = dimensions[1] = q;

    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &grid_comm);
    MPI_Comm_rank(grid_comm, &my_grid_rank);
    MPI_Cart_coords(grid_comm, my_grid_rank, 2,
        coordinates);
    my_row = coordinates[0];
    my_col = coordinates[1];

    free_coords[0] = 0; free_coords[1] = 1;
    MPI_Cart_sub(grid_comm, free_coords, &row_comm);

    free_coords[0] = 1; free_coords[1] = 0;
    MPI_Cart_sub(grid_comm, free_coords, &col_comm);
} /* Setup_grid */
```

- MPI_Cart_create: 1 in call — allow reordering
- MPI_Cart_sub: Analog of MPI_Comm_split
- free_coords[i] = 1, let this coordinate vary when creating new communicators
- free_coords[i] = 0, this coordinate is fixed when creating new communicators

Fox's Algorithm: MPI Code

```
void Fox(int n, matrix_t local_A, matrix_t local_B,
        matrix_t local_C) {
    matrix_t   temp_A;
    int        stage, bcast_root, n_bar, source, dest;
    MPI_Status status;

    n_bar = n/q;
    Set_to_zero(local_C);

    source = (my_row + 1) % q;
    dest = (my_row + q - 1) % q;

    temp_A = Local_matrix_allocate(n_bar);

    for (stage = 0; stage < q; stage++) {
        bcast_root = (my_row + stage) % q;
        if (bcast_root == my_col) {
            MPI_Bcast(local_A, n_bar*n_bar, MPI_FLOAT,
                bcast_root, row_comm);
            Matrix_mult(local_A, local_B, local_C);
        } else {
            MPI_Bcast(temp_A, n_bar*n_bar, MPI_FLOAT,
                bcast_root, row_comm);
            Matrix_mult(temp_A, local_B, local_C);
        }
        MPI_Sendrecv_replace(local_B, n_bar*n_bar, MPI_FLOAT,
            dest, 0, source, 0, col_comm, &status);
    } /* for */
} /* Fox */
```

- Recall q = sqrt(p);
- Bcast shouldn't overwrite local block of A.
- Sendrecv: send message to dest, recv message from source.
- replace: overwrite sent message with received message
- 0's are tags.

Performance

- Current systems: communication *much* more expensive than local operations, especially on clusters.

- Floating point operation on 200 MHz Pentium Pro: 65×10^6 floating point operations per second. $t_a \approx 1.5 \times 10^{-8}$ seconds per op.

- Communication commonly measured in terms of
  - Startup or latency: t_s seconds
  - Bandwidth: $1/t_c$ words/second.

  Cost of sending m words $\approx t_s + mt_c$

- On Myrinet, a fast low-cost network for clustering:
  - Latency: 60μsec
  - Bandwidth: 50 Mbytes/sec

- To carry out 1000 arithmetic ops approximately 15μsec. To send 1000 floats approximately 150μsec

- These figures are fairly suspect: floating point performance is highly dependent on the system and the application. For Myrinet, the latency + bytes/bandwidth is reasonable only for fairly narrow ranges of message sizes.

Performance of Matrix-Matrix Multiplication I

- On hypercube, Allgather uses "butterfly" communication structure:
  - $\log_2(p)$ stages: $\log_2(p)t_s$.
  - During kth stage, $k = 0, 1, \ldots, \log_2(p) - 1$, process pairs exchange $2^k\bar{n}^2$ floats.
  - Total cost of sending floats:
$$\sum_{k=0}^{\log(p)-1} 2^k\bar{n}^2 = (p-1)\bar{n}^2$$
  - Total cost of Allgather:
$$\log(p)t_s + (p-1)\frac{n^2}{p^2}t_c \approx \log(p)t_s + \frac{n^2}{p}t_c$$
- Cost of multiplying $\bar{n} \times n$ matrix by $n \times \bar{n}$ matrix:
$$\bar{n} \times \bar{n} \times (2n-1)t_a = \frac{2n^3 - n^2}{p^2}t_a$$
- p repetitions of Allgather-multiply loop. So total time
$$T_\pi \approx p\log(p)t_s + n^2t_c + \frac{2n^3 - n^2}{p}t_a$$
- Serial runtime of "standard" matrix multiply
$$T_\sigma \approx (2n^3 - n^2)t_a$$
- Speedup:
$$S = \frac{T_\sigma}{T_\pi}$$
- Efficiency
$$E = \frac{T_\sigma}{pT_\pi}$$

Fox's Algorithm: Performance

- In hypercube, cost of broadcast:

$$\frac{1}{2}\log_2(p)\left(t_s + \frac{n^2}{p}t_c\right)$$

- Cost of local matrix multiply:

$$\bar{n}^2(2\bar{n}-1)t_a = \left(\frac{2n^3}{p^{3/2}} - \frac{n^2}{p}\right)t_a$$

- Cost of send-receive:

$$2\left(t_s + \frac{n^2}{p}t_c\right)$$

- $\sqrt{p}$ stages. So total parallel time

$$T_\pi = \frac{1}{2}\sqrt{p}\log(4p)t_s + \frac{1}{2}\log(4p)\frac{n^2}{\sqrt{p}}t_c + \left(\frac{2n^3}{p} - \frac{n^2}{\sqrt{p}}\right)t_a$$

The Rest of MPI

- MPI-1
  - Other languages
  - Environmental management: error handling, timing, etc.
  - Profiling interface
- MPI-2
  - Dynamic process creation: intercommunicators (LAM)
  - One-sided communication
  - Additional collective operations
  - External interfaces: info for system developers
  - I/O! (mpich)

Further Info

- *Parallel Programming with MPI,* Pacheco, Morgan Kaufmann
- *Using MPI,* Gropp, Lusk, and Skjellum, MIT Press
- *MPI: The Complete Reference,* Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press
- Slides, code: `http://www.cs.usfca.edu/mpi`
- MPI Specs: `http://www.mpi-forum.org`
- Mpich Implementation: `http://www.mcs.anl.gov/mpi`
- LAM Implementation: `http://www.mpi.nd.edu/lam`