CSE 160
Lecture 18
Message Passing
% Serial Loop:
for i = 1:n/3-1
    x(2*i) = x(3*i);
end

% Restructured for Parallelism (CORRECT)
for i = 1:3:n/3-1
    y(2*i) = y(3*i);
end
for i = 2:3:n/3-1
    y(2*i) = y(3*i);
end
for i = 3:3:n/3-1
    y(2*i) = y(3*i);
end

% Restructured for Parallelism (CORRECT)
for i = 1:n/3-1
    if (mod((i*2),3))
        a(2*i) = a(3*i);
    end
end
for i = 1:n/3-1
    if (~mod((i*2),3))
        a(2*i) = a(3*i);
    end
end

% Serial Loop:
for i = 1:n/3-1
    x(2*i) = x(3*i);
end

% Restructured for Parallelism (INCORRECT)
for i = 1:n/3-1
    if (mod(i,3) == 0)
        b(2*i) = b(3*i);
    end
end
for i = 1:n/3-1
    if (mod(i,3) ~= 0)
        b(2*i) = b(3*i);
    end
end

% Restructured for Parallelism (INCORRECT)
for i = 1:n/3-1
    if (0 == mod((i*2),3))
        z(2*i) = z(3*i);
    end
end
for i = 1:n/3-1
    if (0 ~= mod((i*2),3))
        z(2*i) = z(3*i);
    end
end
Programming with Message Passing

• Programs execute as a set of P processes (user specifies P)
• Each process assumed to run on a different core
  ‣ Usually initialized with the same code, but has private state
    SPMD = “Same Program Multiple Data”
  ‣ Communicates with other processes by sending and receiving messages
  ‣ Executes instructions at its own rate according to its rank (0:P-1) and
    the messages it sends and receives
• Program execution is often called “bulk synchronous” or “loosely synchronous”
Message passing

- There are two kinds of communication patterns
- **Point-to-point** communication: a single pair of communicating processes copy data between address space
- **Collective communication**: all the processors participate, possibly exchanging information
Point-to-Point message passing

• Messages are like email; to send one, we specify
  ‣ A destination
  ‣ A message body (can be empty)

• To receive a message we need similar information, including a receptacle to hold the incoming data

• Requires a sender and an explicit recipient that must be aware of one another

• Message passing performs two events
  ‣ Memory to memory block copy
  ‣ Synchronization signal on receiving end: “Data arrived”
Send and Recv

• The primitives that implement Pt 2 Pt communication

• When \textbf{Send( )} returns, the message is “in transit”
  ‣ A return doesn’t tell us if the message has been received
  ‣ The data is somewhere in the system
  ‣ Safe to overwrite the buffer

• \textbf{Receive( )} blocks until the message has been received
  ‣ Safe to use the data in the buffer

\begin{tabular}{c|c}
\textbf{Send(}y,1\textbf{)} & \textbf{Recv(}x\textbf{)} \\
\end{tabular}

©2013 Scott B. Baden / CSE 160 / Winter 2013
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- If different processes send messages to the same destination, the order of receipt isn’t defined across sources.
MPI

• We’ll program with a library called MPI “Message Passing Interface”
  ‣ 125 routines in MPI-1
  ‣ 7 minimal routines needed by every MPI program
    • start, end, and query MPI execution state (4)
    • non-blocking point-to-point message passing (3)

• Reference material: see
  http://www-cse.ucsd.edu/users/baden/Doc/mpi.html

• Callable from C, C++, Fortran, etc.

• All major vendors support MPI, but implementations differ in quality
Functionality we’ll will cover today

• Point-to-point communication
• Message Filtering
• Communicators
A first MPI program: “hello world”

#include "mpi.h"

int main(int argc, char **argv ){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Hello, world! I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return(0);
}
MPI’s A minimal interface

• Opening and closing MPI
  ‣ MPI_Init and MPI_Finalize

• Query functions
  MPI_Comm_size( ) = # processes
  MPI_Comm_rank( ) = this process’ rank

• Point-to-point communication
  ‣ Simplest form of communication
  ‣ Send a message to another process
    MPI_Isend( ), MPI_Send( )
  ‣ Receive a message from another process
    MPI_Irecv( ), MPI_Recv( )
  ‣ Wait on an incoming message: MPI_Wait( )
Point-to-point messages

• To send a message we need
  ‣ A destination
  ‣ A “type”
  ‣ A message body (can be empty)
  ‣ A context (called a “communicator” in MPI)

• To receive a message we need similar information, including a receptacle to hold the incoming data

• We can filter messages, enabling us organize message passing activity
Send and Recv

```c
const int Tag=99;
int msg[2] = { rank, rank * rank };
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2, MPI_INT, 1, Tag, MPI_COMM_WORLD, &status);
}
else
    MPI_Send(msg, 2, MPI_INT, 0, Tag, MPI_COMM_WORLD);
```

©2013  Scott B. Baden / CSE 160 / Winter 2013
Communicators

• A communicator is a name-space (or a context) describing a set of processes that may communicate
• MPI defines a default communicator `MPI_COMM_WORLD` containing all processes
• MPI provides the means of generating uniquely named subsets (later on)
• A mechanism for screening messages
MPI Tags

• Tags enable processes to organize or screen messages
• Each sent message is accompanied by a user-defined integer tag:
  ▶ Receiving process can use this information to organize or filter messages
  ▶ `MPI_ANY_TAG` inhibits screening.
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
  ‣ The length in bytes is $\text{sizeof(type)} \times \# \text{ elements}$
  ‣ We don’t specify the as the $\# \text{ byte}$
• MPI specifies a set of built-in types for each of the primitive types of the language
• In C: MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,…
• Also defined types, e.g. structs
What we covered today

• Message passing concepts
• A practical interface
• Next time
  ‣ Applications
  ‣ Asynchronous communication
  ‣ Collective communication