CSE 262
Lecture 6

Programming with Message Passing
MPI
Announcements

• Projects
Today’s lecture

• The Message Passing Programming Model
• The Message Passing Interface - MPI
• A first MPI Application – The Trapezoidal Rule
Architectures without shared memory

- *Shared nothing architecture, or a multicomputer*
- Hierarchical parallelism

uk.hardware.info
tinyurl.com/k6jqag5

Wikipedia

Fat tree
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”
  - Access to local memory only
  - Communicates with other processes by passing messages
  - Executes instructions at its own rate according to its rank (0:P-1) and the messages it sends and receives
Bulk Synchronous Execution Model

• A process is either communicating or computing
• Generally, all processors are performing the same activity at the same time
• Pathological cases, when workloads aren’t well balanced
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 communication

- Messages are like email; to send or receive one, we specify
  - A destination
  - A message body (can be empty)
- 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 at recipient: “Data has arrived”
Send and Recv

- Primitives that implement Pt to Pt communication
- When Send( ) returns, the message is “in transit”
  - Return doesn’t tell us if the message has been received
  - The data is somewhere in the system
  - Safe to overwrite the buffer
- Receive( ) blocks until the message has been received
  - Safe to use the data in the buffer
Causality
Today’s lecture

• The Message Passing Programming Model

• The Message Passing Interface - MPI

• A first MPI Application – The Trapezoidal Rule
MPI

• We’ll use 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
Functionality we’ll will cover today

• Point-to-point communication
• Message Filtering
• Communicators and Tags
• Application: the trapezoidal rule
• Collective Communication
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 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( ) = Isend+Wait`
  - Receive a message from another process
    - `MPI_Irecv( )     MPI_Recv( ) = Irecv +Wait`
  - Wait on an incoming message: `MPI_Wait( )`
Point to Point Communication

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P4</td>
</tr>
<tr>
<td>P1</td>
<td>P5</td>
</tr>
<tr>
<td>P2</td>
<td>P6</td>
</tr>
<tr>
<td>P3</td>
<td>P7</td>
</tr>
</tbody>
</table>

Send(y,1) Recv(x)

Tan Nguyen

Scott B. Baden /CSE 262/ Winter 2015
```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);
```
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 or filtering 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 tag filtering
Message status

- An MPI_Status variable is a struct that contains the sending processor and the message tag.
- This information is useful when we aren’t filtering messages.
- We may also access the length of the received message (may be shorter than the message buffer).

```c
MPI_Recv( message, count,
      TYPE, MPI_ANY_SOURCE,
      MPI_ANY_TAG, COMMUNICATOR,
      &status);

MPI_Get_count(&status, TYPE, &recv_count);

status.MPI_SOURCE      status.MPI_TAG
```

Scott B. Baden / CSE 262/ Winter 2015 22
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
  ♦ The length in bytes is sizeof(type) × # elements
  ♦ We don’t specify the as the # 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
Today’s lecture

• The Message Passing Programming Model
• The Message Passing Interface - MPI
• A first MPI Application – The Trapezoidal Rule
The trapezoidal rule

• Use the trapezoidal rule to numerically approximate a definite integral, area under the curve

• Divide the interval \([a,b]\) into \(n\) segments of size \(h=1/n\)

• Area under the \(i^{th}\) trapezoid
  \[
  \frac{1}{2} \left( f(a+i\times h) + f(a+(i+1)\times h) \right) \times h
  \]

• Area under the entire curve
  \[\approx \text{sum of all the trapezoids}\]
Reference material

- For a discussion of the trapezoidal rule
  http://en.wikipedia.org/wiki/Trapezoidal_rule

- A applet to carry out integration

- Code on Stampede (from Pacheco hard copy text)

  Serial Code
  `$PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/serial.c`

  Parallel Code
  `$PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/trap.c`
main() {
    float f(float x) { return x*x; } // Function we're integrating

    float h = (b-a)/n; // h = trapezoid base width
    // a and b: endpoints
    // n = # of trapezoids

    float integral = (f(a) + f(b))/2.0;

    float x; int i;

    for (i = 1, x=a; i <= n-1; i++) {
        x += h;
        integral = integral + f(x);
    }
    integral = integral*h;
}
Parallel Implementation of the Trapezoidal Rule

- Decompose the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- Processors combine their local integrals into a global one
First version of the parallel code

int local_n = n/p;     // # trapezoids; assume p divides n evenly
float local_a = a + my_rank*local_n*h,
        local_b = local_a + local_n*h,
        integral = Trap(local_a, local_b, local_n);

if (my_rank == ROOT) {  // Sum the integrals calculated by
    // all processes
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, MPI_ANY_SOURCE,
                  tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1,
             MPI_FLOAT, ROOT, tag, WORLD);
Using collective communication

- The result does not depend on the order in which the sums are taken, except to within roundoff
- We can often improve performance by taking advantage of global knowledge about communication
- Instead of using point to point communication operations to accumulate the sum in linear time, use collective communication that runs in log time

```c
local_n = n/p;
float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
    integral = Trap(local_a, local_b, local_n, h);
MPI_Reduce( &integral, &total, 1,
    MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD)
```
Collective communication in MPI

- Collective operations are called by **all** processes within a communicator.
- Broadcast: distribute data from a designated “root” process to all the others
  
  \[
  \text{MPI\_Bcast}(\text{in}, \text{count}, \text{type}, \text{root}, \text{comm})
  \]

- Reduce: combine data from all processes and return to a designated root process
  
  \[
  \text{MPI\_Reduce}(\text{in}, \text{out}, \text{count}, \text{type}, \text{op}, \text{root}, \text{comm})
  \]

- Allreduce: all processes get reduction: \text{Reduce} + \text{Bcast}
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
local_b = local_a + local_n*h,
integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce(&integral, &total, 1,
MPI_FLOAT, MPI_SUM, WORLD)
Message completion

• A `Send()` may or may not complete…
• … before a `Recv()` has been posted
• “May or may not” depends on the implementation
• Some programs may deadlock on certain message passing implementations

<table>
<thead>
<tr>
<th></th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>This program may deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send (x,1)</td>
<td>Send(y,0)</td>
<td></td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Recv(x,0)</td>
<td></td>
</tr>
</tbody>
</table>

This program is “safe”
MPI has pre-allocated storage for the incoming message so there's no possibility of running out of storage

<table>
<thead>
<tr>
<th></th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (x,1)</td>
<td>Recev(x,0)</td>
<td></td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Send(y,0)</td>
<td></td>
</tr>
</tbody>
</table>

Scott B. Baden /CSE 260/ Winter 2014
Asynchronous, non-blocking communication

- With Send or Receive, a return indicates the buffer may be reused, or that the data is ready
- The routines block until completion
- The non-blocking asynchronous forms, do not wait for completion: “Immediate return”
  - Required to express certain algorithms
  - Optimize performance: message flow problems

**Split-phased**
- Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call `IRecv()`, `ISend()`
- Phase 2: synchronize `Wait()`
- Perform unrelated computations between the two phases
Immediate mode send and receive

- Must synchronize with a `Wait()` before reusing buffer (`Send`) or consuming data (`Receive`)
- A `request` argument enables us to refer to a message we are waiting on

```c
MPI_Request request;
MPI_Irecv(buf, count, type, src, tag, comm, &request)
MPI_Wait(&request, &status)
```

- `Irecv + Wait = Recv`
  ```c
  MPI_Recv(buf, count, type, src, tag, comm, &status)
  ```

- `Immediate Send`
  ```c
  MPI_Isend(buf, count, type, dest, tag, comm, &request)
  ```
Restrictions on non-blocking communication

• The message buffer may not be accessed between an `IRecv()` (or `ISend()`)) & its accompanying `Wait()`

  `ISend(data,destination)`
  `Wait()` on `ISend()`

  Use the data

• Each pending `IRecv()` must have a distinct buffer
## Overlap

<table>
<thead>
<tr>
<th>Overlap</th>
<th>No Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x, req)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(..)</td>
<td>Send(…)</td>
</tr>
<tr>
<td>Compute(y)</td>
<td>Wait(x)</td>
</tr>
<tr>
<td>Wait(req)</td>
<td>Compute(x)</td>
</tr>
<tr>
<td>Compute(x)</td>
<td>Compute(y)</td>
</tr>
</tbody>
</table>

A message buffer may not be accessed between an `IRrecv()` (or `ISend()` ) and its accompanying `wait()`
What we covered today

• Message passing concepts
• A practical interface – MPI
  ❖ Collective communication
  ❖ Asynchronous, non-blocking communication
• Next time
  ❖ More collective communication primitives
  ❖ NewApplications