A first application

- Compute a numerical approximation to the definite integral

\[ \int_{a}^{b} f(x) \, dx \]

using the trapezoidal rule
How the trapezoidal rule works

- Divide the interval \([a, b]\) into \(n\) segments of size \(h = 1/n\)
- Approximate the area under an interval using a trapezoid
- Area under the \(i\)th trapezoid
  \[ \frac{1}{2} (f(a + i \times h) + f(a + (i + 1) \times h)) \times h \]
- Area under the entire curve
  \[ \approx \text{sum of all the trapezoids} \]

Reference material

- For a discussion of the trapezoidal rule
  [http://metric.ma.ic.ac.uk/integration/techniques/definite/numerical-methods/trapezoidal-rule/](http://metric.ma.ic.ac.uk/integration/techniques/definite/numerical-methods/trapezoidal-rule/)
- A applet to carry out integration
- Code (from Pacheco hard copy text)
  
  \$(CSE160)/ppmpi_c/chap04/serial.c \quad \text{(Serial code)}$
  \$(CSE160)/ppmpi_c/chap04/trap.c \quad \text{(Parallel code)}$

Where \$CSE160 = valkyrie:~/../public/Pacheco\$
Serial code (Following Pacheco)

```c
main() {  
    float integral; /* Store result in integral */
    float x;
    int i;

    float f(float x) { /* Function we're integrating */
        return x*x;
    } /* f */

    h = (b-a)/n;  /* n = # of trapezoids, h= trapezoid base width; a&b: endpoints */
    integral = (f(a) + f(b))/2.0;
    for (i = 1, x=a; i <= n-1; i++) {
        x += h;
        integral = integral + f(x);
    }
    integral = integral*h;
}
```

The parallel algorithm

- 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

```c
local_n = n/p; /* Number of 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, h);

if (my_rank == 0) { /* Add up the integrals calculated by each process */
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, WORLD);
```

Improvements

- Use wildcard specifier for sending process
- Avoid linear running time algorithm for computing the global sum – use collective communication

```c
for (source = 1; source < p; source++) {
    MPI_Recv(&integral, 1, MPI_FLOAT,
              MPI_ANY_SOURCE, tag,
              WORLD, &status);
    total += integral;
}
```
Under the hood of MPI

- Note that if there is not a pending receive, then an incoming message is placed in an anonymous system buffer
- When the receive gets posted, the message is moved into the user specified buffer
- The extra copying can be expensive
- Non-blocking communication can help ameliorate this problem
- For more information see
  
  *MPI: The Complete Reference*, by Marc Snir et al.
  
  
  “Buffering and Safety”

Message passing implementations

- When a long message is to be sent, MPI first checks if the recipient has sufficient storage to receive the message
- If so, then it sends the message. This is called a *rendezvous* implementation. What are the advantages and disadvantages?
Eager limits

- In an *eager* implementation, we just send the message
- In practice, MPI implementations switch between the two modes
- The *eager limit* is the longest message that can be sent in eager mode

Send Modes

- MPI provides four different *modes* for sending a message
  - Standard: Send *may or may not* complete until matching receive is posted (whether or not the data is buffered is up to the implementation)
  - Synchronous: Send does not complete until matching receive is posted
  - Ready: Matching receive must already have been posted
  - Buffered: data is moved to a user-supplied buffer before sending
- See the handy reference at [http://www-unix.mcs.anl.gov/mpi/sendmode.html](http://www-unix.mcs.anl.gov/mpi/sendmode.html)
Communication modes

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

How does the behavior of this program vary among Standard, Synchronous, Ready, and Buffered send modes?

Communication performance

- Communication performance is often a significant preoccupation in building good quality code
- Let the message have a length n
- The simplest communication cost model is transfer time = $\alpha + \beta n$
  - $\alpha$ = message startup time
  - $\beta = 1/$ peak bandwidth (bytes per second)
  - $n$ = message length
Startup and bandwidth

- The startup term dominates when the message is sufficiently short
  \[ \alpha > \beta n \Rightarrow n < \alpha / \beta \]
- The bandwidth term dominates when the message is sufficiently long
  \[ n > \alpha / \beta \]
- We refer to this message threshold as the half power point \( n_{1/2} \)

Half power point

- Gives a relationship between startup and bandwidth
- \( n_{1/2} = \) message size required to achieve ½ peak bandwidth \((1/\beta)\)
- In theory, this occurs when \( \alpha = \beta n_{1/2} \)
- But look closely!
- For NPACI Blue Horizon, the actual value of \( n_{1/2} = 32 \text{ KB} \)
- Next week we’ll measure \( n_{1/2} \) for Valkyrie
More about modeling

- LogP model (Culler et al, 1993), is more precise, but the $\alpha$, $\beta$ model is often good enough
- All these models ignore important effects: switch and processor contention

Where does the time go?

- Under ideal conditions…
  - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the users message buffer
  - There is no other communication traffic
- Assume a contiguous message
Microbenchmark to determine communication cost model parameters

- Communication cost model:
  transfer time = $\alpha + \beta n$
  $\alpha$ = message startup time
  $\beta$ = inverse peak bandwidth

![Diagram showing network interfaces, sender, and receiver with latency and communication cost parameters]
Determining $n_{1/2}$

$195 \text{ MB/sec}$
$N = 2\text{MB}$

$n_{1/2} \sim 96\text{KB}$

Communication time
Communication time for short messages

Performance measurement

- To measure times in MPI, use MPI_Wtime()
- Measures wall clock time
- We often need to eliminate transient behavior
  - Measure sufficiently long periods of representative steady state behavior
  - “Warm” up the program by running it first without collecting timing information
  - Repeat the measurements several times, and report the shortest times
  - But note any outliers
Evaluating communication performance with the Ring program

- We configure the processors in a logical ring and pass messages around the ring
- Assume there are \( p \) processors
- Neighbors of processor \( k \) are
  - \( (k + 1) \mod p \)
  - \( (k + p - 1) \mod p \)

What does the ring program measure?

- Let \( P_0 \) time the cost of sending a message around the ring \( R \) time
- What are we measuring?
Measurement technique with Ring

for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
if (myid == 0) {
    // (Execute code for warm up)
    const double start = MPI_Wtime( );
    for (int i = 0; i < trips; i++) {
        MPI_Request req;
        MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1) % p,
                   tag, MPI_COMM_WORLD, &req);
        MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p,
                  tag, MPI_COMM_WORLD);
        MPI_Status status;
        MPI_Wait(&req,&status);
    }
    const double delta = MPI_Wtime() - start;
    const long bw = (long)((trips*len*nodes)/delta/1000.0);
}

The Ring program continued

for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
if (myid == 0) {
    // Send side code
    const double delta = MPI_Wtime() - start;
    const long bw = (long)((trips*length*nodes)/delta/1000.0);
} else {
    // (WARM UP CODE)
    for (int i = 0; i < trips; i++) {
        MPI_Status status;
        MPI_Recv(buffer, len, MPI_CHAR, MPI_ANY_SOURCE,
                  tag, MPI_COMM_WORLD, &status);
        MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p,
                  tag, MPI_COMM_WORLD);
    }
}
The Ring program

if (myrank == 0)
    for (int i = 0; i < trips; i++) {
        MPI_Request req;
        MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1)%p, tag, MPI_COMM_WORLD, &req);
        MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p, tag, MPI_COMM_WORLD);
        MPI_Status status;
        MPI_Wait(&req,&status);
    }

else  // myrank != 0
    for (int i = 0; i < trips; i++) {
        MPI_Status status;
        MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p, tag, MPI_COMM_WORLD, &status);
        MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p, tag, MPI_COMM_WORLD);
    }
More on communication cost

• Buffering a message is costly
• Modify the Ring program to copy the message a specified number of times before sending it and note the effect