Lab 14

Collective Communication

Lab Objective: *Learn how to use collective communication to increase the efficiency of parallel programs*

In the lab on the Trapezoidal Rule [Lab **??**], we worked to increase the efficiency of the trapezoidal rule by parallelizing the code and by load-balancing the parallel processes. However, the algorithm is still far from being optimized.

Look back at the summation that the root process (process 0) does. After each process independently calculates its estimate, the results must be compiled somehow, and the method we have chosen is rather inefficient. While the root process sums the data, the other processors sit idly. A more efficient algorithm would balance the work-load so that there are non-idling processes. Additionally, the root process is solely responsible for communicating with each of the other processes. The communication involved in parallel programs is usually a substantial bottleneck, and this is no exception.

Both of these problems can be somewhat corrected through what is called "Collective Communication." Besides helping to alleviate load imbalances, however, collective communication has a more important purpose: it helps to optimize message passing among separate processes. Communication among processes is expensive. Because each message must be sent over some sort of network, we must minimize and optimize this inter-process communication.

There are two important principles to remember here:

Load Balancing: A program is inefficient if it is not using all of the available resources (e.g., processes are idling because they are waiting for each other)

Communication is Expensive: *Broadcast* and *Reduce* (two MPI functions we will introduce in this lab) are designed to optimize communication among the whole communicator. However, any sort of message passing is extremely expensive and one of the main obstacles to obtaining speedups when parallelizing an algorithm.

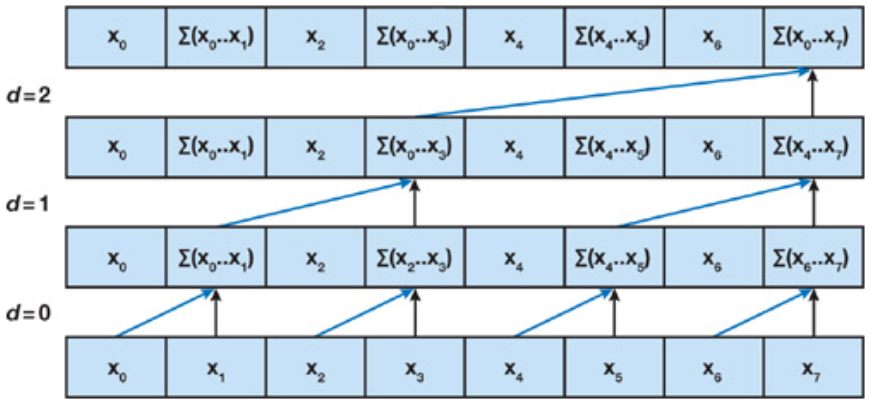

Figure 14.1: Tree structure in fast summation.

Tree-Structured Computation

Suppose we have eight processes, each with a number to be summed. Let the even ranked processes send their data to the odd process one greater than each respective process. The odd processes receive data from directly below them in rank. By so doing, we have in one time-step done half of the work.

From this point we can repeat the process with the odd processes, further partitioning them. A summation done in this manner creates a tree structure (see the figure [14.1]). MPI has implemented fast summations and much more in methods referred to as "Collective Communication." The method shown in the figure is called *Reduce*.

Furthermore, MPI has methods that can also distribute information in efficient ways. Imagine reversing the arrows in the figure [14.1]. This is MPI's *Broadcast* function. We should note, though, that the image is a simplification of how MPI performs collective communication. In practice, it is different in every implementation, and is usually more efficient than the simplified example shown. MPI is designed to optimize these methods under the hood, and so we only need to worry about properly applying these functions.

Knowing this, let's take another look at the trapezoidal rule.

The Parallel Trapezoidal Rule 2.0

Below is the code for our revised edition of the trapezoid rule. (This code requires that the number of trapezoids is evenly divisible by the the number of processes). The main change is that the statement `comm.Reduce(integral, total)` has replaced the entire if-else statement that was used to compile the results, including the for loop. Not only does the code look cleaner, it runs faster.

```
""" trapParallel_2.py
    Example usage:
        $ mpirun -n 10 python.exe trapParallel_2.py 0.0 1.0 10000
        With 10000 trapezoids, the estimate of the integral of x^2 from 0.0 to ↩
            1.0 is:
```

```
            0.333333335
    ***In this implementation, n must be divisble by the number of processes***
"""

from __future__ import division
from sys import argv
from mpi4py import MPI
import numpy as np


COMM = MPI.COMM_WORLD
SIZE = COMM.Get_size()
RANK = COMM.Get_rank()

def integrate_range(fxn, a, b, n):
    ''' Numerically integrates the function fxn by the trapezoid rule
        Integrates from a to b with n trapezoids
        '''
    # There are n trapezoids and therefore there are n+1 endpoints
    endpoints = np.linspace(a, b, n+1)

    integral = sum(fxn(x) for x in endpoints)
    integral -= (fxn(a) + fxn(b))/2
    integral *= (b - a)/n

    return integral

# An arbitrary test function to integrate
def function(x):
    return x**2

# Read the command line arguments
a = float(argv[1])
b = float(argv[2])
n = int(argv[3])


step_size = (b - a)/n
# local_n is the number of trapezoids each process will calculate
# ***Remember, in this implementation, n must be divisible by SIZE***
local_n = n / SIZE

# local_a and local_b are the start and end of this process' integration range
local_a = a + RANK*local_n*step_size
local_b = local_a + local_n*step_size

# mpi4py requires these to be numpy objects:
integral = np.zeros(1)
integral[0] = integrate_range(function, local_a, local_b, local_n)



# This has been the same as trapParallel_1.py up until this line. The rest is new:


total_buffer = np.zeros(1)

# The root node receives results with a collective "reduce"
COMM.Reduce(integral, total_buffer, op=MPI.SUM, root=0)
```

```
64
total = total_buffer[0]
66
# Now the root process prints the results:
68 if RANK == 0:
    print "With {0} trapezoids, the estimate of the integral of x^2 from {1} to ↩
        {2} is: \n\t{3}".format(n, a, b, total)
```

trapParallel_2.py

Some explanation is necessary for the call to `COMM.Reduce`. The first argument, `integral`, is the number which is going to be added up. This number is different for every process. The variable `total_buffer` will hold the result in the root process after this line runs. The parameter `op=MPI.SUM` tells the communicator to use the addition operator to combine the values.

The careful observer may have noticed from the figure 14.1 that in the end of the call to `Reduce`, only the root process holds the final summation. This brings up an important point: *collective communication calls can return different values to each process*. This can be a "gotcha" if you are not paying attention.

So what can we do if we want the result of a summation to be available to all processes? Use the provided subroutine `AllReduce`. It does the same thing as `Reduce`, but it simultaneously uses each process as the root. That way, by the end of the summation, each process has calculated an identical sum. Duplicate work was done, but in the end, the result ends up on each process at the same time.

So when should we use `Reduce` and when should we use `AllReduce`? If it is unimportant for all processes to have the result, as is the case with the Trapezoidal Rule, using `Reduce` should be slightly faster than using `AllReduce`. On the other hand, if every process needs the information, `AllReduce` is the fastest way to go. We could come up with some other solutions to the situation, such as a call to `Reduce`, followed by a call to `Broadcast`, however these other solutions are always less efficient than just using one call to the collective communication routines. In the end, the less communications, the better.

Reduce(...) and Allreduce(...)

Comm.Reduce(sendbuf, recvbuf, Op op=MPI.SUM, root=0) Reduces values on all processes to a single value on the root process. Parameters:

sendbuf (array-like) address of send buffer

recvbuf (array-like) address of recieve buffer (only significant in root process)

op (MPI Op) reduce operation

root (int) rank of root of operation

Example:

```
1 #Reduce_example.py
2 from mpi4py import MPI
  import numpy as np
4
```

```
COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
operand_buffer = np.array(float(RANK))
SIZE_buffer = np.zeros(1)

COMM.Reduce(operand_buffer, SIZE_buffer, op=MPI.MAX)
if RANK == 0:
    SIZE = 1 + int(SIZE_buffer[0])
    print "The size is {}.".format(SIZE)
```

Reduce_example.py

Comm.Allreduce(sendbuf, recvbuf, Op op=MPI.SUM) Reduces values on all processes to a single value on all processes. Parameters:

sendbuf (array-like) address of send buffer

recvbuf (array-like) address of recieve buffer (only significant in root process)

op (MPI Op) reduce operation

Example: Same as the example for Reduce except replace COMM.Reduce with COMM.Allreduce and remove the if statement. Notice that all process now have the reduced value.

The following table contains the predefined operations that can be used for the input parameters *Op*. There are also methods that allow the creation of user-defined operations. Full documentation is found here: `http://mpi4py.scipy.org/docs/apiref/mpi4py.MPI.Op-class.html`.

TODO figure out how to format this table

Problem 1. What is the difference between `Reduce` and `Allreduce`?

Problem 2. Why is `Allreduce` faster than a `Reduce` followed by a `Bcast`?

Parallelizing the Dot Product

Calculating a dot product is a relatively simple task which does not need to be parallelized, but it makes a good example for introducing the other important collective communication subroutines. When provided two vectors, the dot product is the sum of the element wise multiplications of the two vectors:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i$$

In order to parallelize this, we can divide up the work among different processors by sending pieces of the original vectors to different processors. Each processor then

multiplies its elements and sums them. Finally, the local sums are summed using `Reduce`, which sums numbers distributed among many processes in *O(log n)* time.

The code looks like this:

```
'''
Takes a dot product in parallel.
Example usage:
    $ mpirun -n 4 python.exe dot.py 1000
Assumes n is divisible by SIZE

command line arguments: n, the length of the vector to dot with itself
'''

from mpi4py import MPI
import numpy as np
from sys import argv


COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()

ROOT = 0

n = int(argv[1])


if RANK == ROOT:
    x = np.linspace(0, 100, n)
    y = np.linspace(20, 300, n)
else:
    x, y = None, None

# Prepare variables
local_n = n // SIZE
if n % SIZE != 0:
    print "The number of processors must evenly divide n."
    COMM.Abort()

local_x = np.zeros(local_n)
local_y = np.zeros(local_n)

COMM.Scatter(x, local_x)
COMM.Scatter(y, local_y)

local_dot_product = np.dot(local_x, local_y)
buf = np.array(local_dot_product)

result_buf = np.zeros(1) if RANK == ROOT else None
COMM.Reduce(buf, result_buf, MPI.SUM)

if RANK == ROOT:
    print "Parallel Dot Product: ", str(result_buf[0])
    print "Serial Dot Product: ", str(np.dot(x, y))
```

dot.py

The actual distributing of the data occurs with the calls to `Scatter`. `Scatter` takes an array, divides it up, and distributes one piece to each process. Afterward, the

serial dot product (using `np.dot`) is run to calculate a local dot product within each process. Finally, `Reduce` is called to collect the results into the root process.

With this new algorithm and enough processors, the runtime approaches O(log n) time. To help understand how much faster this is compared to the original version, imagine that we have as many processors as we have items in each array: say, 1000. Then, each of these operations requires the same amount of time, the algorithm's runtime would run hypothetically something like this:

TODO figure out how to format this table

The first column shows the serial dot product's time requirements. The final column shows the mpi-version. It is clear that the final version requires less time and that this number drops with the number of processors used. We simply cannot get this speed up unless we break out of the serial paradigm.

The middle column illustrates why we should use *broadcast*, and *scatter*, and *reduce* over using *n send/recv* pairs in a for-loop. We do not include code for this variation. *It turns out that using MPI in this fashion can actually run slower than the serial implementation of our dot product program.*

A Closer Look at Broadcast and Reduce

Let's take a closer look at the call to `COMM.Bcast`. `Bcast` sends data from one process to all others. It uses the same tree-structured communication illustrated in the Fast Sum figure [14.1]. It takes for its first argument an array of data, which must exist on all processors. However, what is contained in that array may be insignificant. The second argument tells Bcast which process has the useful information. It then proceeds to overwrite any data in the arrays of all other processes.

Bcast behaves as if it is synchronous. "Synchronous" means that all processes are in sync with each other, as if being controlled by a global clock tick. For example, if all processes make a call to `Bcast` they will all be guaranteed to be calling the subroutine at basically the same time.

For all practical purposes, you can also think of `Reduce` as being synchronous. The difference is that Reduce only has one receiving process, and that process is the only process whose data is guaranteed to contain the correct value at completion of the call. To test your understanding of Reduce, suppose we are adding the number one up by calling `COMM.Reduce` on 100 processes, with the root being process 0. The documentation of `COMM.Reduce` tells us that the receive buffer of process 0 will contain the number 100. What will be in the receive buffer of process 1?

The answer is we don't know. The reduce could be implemented in one of several ways, dependent on several factors, and as a result it is non-deterministic. During collective communications such as `Reduce`, we have no guarantee of what value will be in the intermediate processes' receive buffer. More importantly, future calculations should never rely on this data except in process root (process *0* in our case).

Problem 3. In our parallel implementation of the calculation of the dot product, `dotProductParallel_1.py`, the number of processes must evenly divide the length of the vectors. Rewrite the code so that it runs regardless

of vector length and number of processes (though for convenience, you may assume that the vector length is greater than the number of processes). Remember the principle of load balancing. Use Scatterv() to accomplish this.

Problem 4. Alter your code from the previous exercise so that it calculates the supremum norm(the maximal element) of one of the vectors (choose any one). This will include changing the operator Op in the call to `Reduce`.

Problem 5. Use `Scatter` to parallelize the multiplication of a matrix and a vector. There are two ways that this can be accomplished. Both use `Scatter` to distribute the matrix, but one uses `Bcast` to distribute the vector and `Gather` to finish while the other uses `Scatter` to segment the vector and finishes with `Reduce`. Outline how each would be done. Discuss which would be more efficient (hint: think about memory usage). Then, write the code for the better one. Generate an arbitrary matrix on the root node. You may assume that the number of processes is equal to the number of rows (columns) of a square matrix. Example code demonstrating scattering a matrix is shown below:

```
#matrix_scatter_example
from mpi4py import MPI
import numpy as np

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
A = np.array([[1.,2.,3.],[4.,5.,6.],[7.,8.,9.]])
local_a = np.zeros(3)
COMM.Scatter(A, local_a)
print "Process {0} has {1}.".format(RANK, local_a)
```

matrix_scatter_example.py