MTH 5050- Intro to Parallel Process
Homework 1: Numerical Integrations
January 17, 2019
Max Lê
ID: 901223283

1 Problem Statement

Our tasks are to numerically integrate the following expressions, using the midpoint rule and compare the accuracy and run time for each.

$$I_1 = \int_0^1 \frac{4}{1+x^2} dx \tag{1}$$

$$I_2 = \int_0^1 4\sqrt{1-x^2} dx \tag{2}$$

The midpoint rule involves splitting up the integration domain into N segments. Then, for each of these equal width subintervals, we are interested in the midpoint values. Using these midpoint values, we can construct sub-rectangles. The final result involves summing up these sub rectangles, hoping that it would give a good approximation to the area under the curve. Denoting the width of the sub rectangle as Δx, and the midpoint values as: $xmid_1$, $xmid_2$,... $xmid_N$, we can write the sum as follow:

$$I \approx \Delta x f(xmid_1) + \Delta x f(xmid_2) + ... + \Delta x f(xmid_N) \tag{3}$$

Although the following assumptions are trivial, it is important to note them. We assume that:

- Area under the curve is non negative. Otherwise, in some intervals, the areas need to be subtracted instead of adding.
- Area is additive, so we can add a finite amount of rectangles to get the integration.
- Our rectangles are perfect $\rightarrow$ area = width*height
- The function is defined on the range 0 to 1. Taking the limit as x goes to 1, we get 2 for the 1st integral and 4 for the second integral. These, in turn, equal the evaluated values at f1(1) and f2(1) respectively. Therefore, we can say the function is defined and continuous on 0 to 1.

2 Analytical Solutions

These given integrals have a trivial solution: the transcendental number π

$$\begin{aligned} I_1 &= \int_0^1 \frac{4}{1+x^2} dx \\ &= 4\int_0^1 \frac{1}{1+x^2} dx \end{aligned}$$

Let x = $tan(u)$

Then dx = $sec^2 u du$

$$\begin{aligned} I_1 &= 4\int_0^{\frac{\pi}{4}} \frac{1}{1+tan^2 u} sec^2 u du \\ &= 4\int_0^{\frac{\pi}{4}} \frac{1}{sec^2 u} sec^2 u du \\ &= 4\int_0^{\frac{\pi}{4}} 1 du \\ &= 4 * \left(\frac{\pi}{4} - 0\right) \\ &= \boxed{\pi} \end{aligned}$$

Likewise:

$$I_2 = \int_0^1 4\sqrt{1-x^2} dx$$

Let x = sin(u)

Then dx = cos(u) du

$$\begin{aligned} I_2 &= 4\int_0^{\frac{\pi}{2}} \sqrt{1-sin^2(u)} cos(u) du \\ &= 4\int_0^{\frac{\pi}{2}} cos^2(u) du \\ &= 4\int_0^{\frac{\pi}{2}} \left(\frac{1}{2} + \frac{1}{2} cos2u\right) du \\ &= 4(\frac{\pi}{4} + \frac{1}{2}(0)) \\ &= \boxed{\pi} \end{aligned}$$

3 Numerical Approach

The functions are given and can be evaluated to find the height of the sub rectangles, what we need is the coordinate of the midpoint. Say our interval spans from a to b, and we divide into n subintervals. Each of these subintervals has a width of "delx", then the algorithm to get the location of the midpoints is as follow (as an example small code segment):

```
DO i = 1,n
     xmid(i) = a + 0.5*delx + (i-1)*delx
END DO
```

This comes from the following sequence, assuming we start at a and end at b, with delx is the distance of each subintervals.

- 1st midpoint = a + delx/2
- 2nd midpoint = a + delx/2 + delx/2 + delx/2 = a + delx/2 + delx
- 3rd midpint = a + delx/2 + 2delx

So on and so forth... ,the sequence has the form a + delx/2 + "a sequence that goes 0,delx, 2delx....". This second sequence translates to (i-1)*delx because the language is FORTRAN, where index starts at 1. Thus, we have the algorithm above.
The width of the subintervals, Δx is calculated by : $\frac{b-a}{N}$, where a and b are lower and upper limit of our integrands respectively, and N is the number of subintervals that we want. This is our mesh, and for simplicity, we are using constant width mesh.
After the midpoints' locations are determined, the function values at these midpoints are determined, then the areas of the sub-rectangles are computed. Finally, it is just a matter of summing up these sub-rectangles.

```
SUM = 0.0
DO i = 1,n
     SUM = SUM + delx*f(xmid(i))
END DO
```

4 Results and Analysis

4.1 Results from Code

RESULTS FOR SOLVING INTEGRAL FROM 0 TO 1 OF 4/(1+x^2)

N	NUMERICAL	ANALYTICAL	ERROR	TIME(s)
2	3.162353038788	3.141592741013	0.020760	0.000002000015
20	3.141800880432	3.141592741013	0.000208	0.000001000008
200	3.141593933105	3.141592741013	0.000001	0.000001999957
2000	3.141593694687	3.141592741013	0.000001	0.000025000016
20000	3.141593217850	3.141592741013	0.000000	0.000238999957
200000	3.141476154327	3.141592741013	0.000117	0.002422000049

RESULTS FOR SOLVING INTEGRAL FROM 0 TO 1 OF 4*sqrt(1-x^2))

N	NUMERICAL	ANALYTICAL	ERROR	TIME(s)
2	3.259367465973	3.141592741013	0.117775	0.000006999937
20	3.145430564880	3.141592741013	0.003838	0.000001000008
200	3.141714811325	3.141592741013	0.000122	0.000003000023
2000	3.141596317291	3.141592741013	0.000004	0.000031000003
20000	3.141593217850	3.141592741013	0.000000	0.000291000004
200000	3.141563892365	3.141592741013	0.000029	0.002939999802

4.2 Verification with n = 2

If n = 2, then $\Delta x = \dfrac{1.00 - 0}{2} = 0.5$. We have two intervals, 0 to 0.5, and 0.5 to 1.00. It is trivial to see that the midpoints' locations are at 0.25 (first interval from 0 to 0.5), and 0.75 (second interval from 0.5 to 1.00). Thus, our area for the two rectangles are as follow:

$$\text{Rectangle } 1 = f(0.25) * \Delta x \tag{4}$$

$$\text{Rectangle } 2 = f(0.75) * \Delta x \tag{5}$$

First, for the integral $\displaystyle\int_0^1 \frac{4}{1+x^2} dx$:

$$\begin{aligned} \text{Rectangle 1, first} &= \frac{4.00}{1 + 0.25^2} * 0.5 \\ &= 1.882352941 \end{aligned}$$

$$\begin{aligned} \text{Rectangle 2,first} &= \frac{4.00}{1 + 0.75^2} * 0.5 \\ &= 1.280000000 \end{aligned}$$

Summing them up, we have: $I_1 = 1.882352941 + 1.280000000 = \boxed{3.162352941}$, which is closed to the results from code with rounding errors (calculator can only do 9 decimal points)

Now, for the integral $\int_0^1 4\sqrt{1-x^2}dx$:

$$\begin{aligned}\text{Rectangle 1, second} &= 4.00 * \sqrt{1-(0.25)^2} * 0.5 \\ &= 1.936491673\end{aligned}$$

$$\begin{aligned}\text{Rectangle 2,second} &= 4.00 * \sqrt{1-(0.25)^2} * 0.5 \\ &= 1.3228756656\end{aligned}$$

Again, we sum them up: $I_2 = 1.936491673 + 1.3228756656 = \boxed{3.259367329}$, which is closed to the results from code with rounding errors (calculator can only do 9 decimal points). Thus, our hand calculations match the code.

4.3 Analysis

4.3.1 Accuracy

We can see that if more subintervals are added (more rectangles), then our numerical solution gets very close to the analytical solution. We can say so because as the N increases, the error between numerical and analytical gets smaller. At the maximum subintervals, N = 200000, the error is on the order of 10^{-6}. Mathematically, this makes sense because as we increase the number of subintervals, N, then the width of our sub-rectangles, delx, becomes smaller and thus we can fit more rectangles to represent the area under the curve. An accurate representation of the area under the curve means a more accurate computation of the integral.

4.3.2 Run time

Unfortunately, accuracy comes at a price: higher sub intervals require more computational power, this is shown by the fact that the GPU takes more time to process. For the first integral, the time for N = 200000 is as much as 1000 times the GPU time for N = 2. If we think about this, the main algorithm to calculate the integral is something like this code snippet.

```
I = 0
DO i = 1,n
    !Calculate the midpoint
    xmid(i) = a + 0.5*delx + (i-1)*delx
    !Calculate the sum
    Integral = Integral + delx*f(xmid(i))
END DO
```

If we think about the lecture notes, where we had something like this:

```
DO i = 1,n
    z(i) = x(i) + y(i)
END DO
```

As n increases, the runtime increases. If n = 1000, then the code takes 1000 clock cycles to run, or 1000 times as long as it takes when n = 1. In our case, the two extreme subintervals are n = 2, and n = 200000. We would expect when n = 200000, the code would take: $\frac{200000}{2} = 100000$. In reality, our runtime is only 1000 times as much. This is better than theory but one explanation for this would be that instead of adding and subtracting arrays, we are actually populating array elements for xmid and for Integral. Therefore, it is faster.

4.3.3 How to parallelize

At first, one can try to code this in parallel via the shared memory approach. Instead of having 1 processor doing all the computation from 1 to 10, we can have 2 processors, each is responsible for half of the For/Do loop. The downside for this is how to communicate. This is a sequential problem, where the Integral result from the previous cycle of the Do/For loop is needed for the next cycle. Therefore, if processor A gets result at from N = 1 to 49, and processor B cannot get results at N = 50 because it doesn't know what A has at the 49th iteration. So.. another way to do this is via message passing. This is done by having 10 processors, each tries to finishes the 1 cycle of the Do/For loop, and then messages are sent in between to get the updated Integral. For example, processor 1 will use all its power to compute result at N = 1, then via message passing, it will transfer its result (the sum) to processor 2, so that processor 2 can use the result and compute its own sum. The lised code below looks complicate because I try to automate the process of repeating the computation for diffrent Ns. However, the core of the program is nothing but performing a number of adds to an array. Therefore, if we can parallelize this core code, then the program will run much faster.
Another reason why communication is important can be seen in the algorithm itself. In class, the concept of parallelizing is shown when two vectors are added to make a third vector. We assume that these two vectors are already populated. The task is to add them up and populate the third vector. Ideally, we can use shared memory and split the for loop: so first processor does the first 49 adds,and second processor does the rest of the adds. There is no need to use the previous values, unlike our midpoint solver.

5 Code

(One single file)

```
program midpoint
    implicit none

    !*********************  INTERFACE FUNCTION*************!
    interface
        function f1 (x)
            real :: f1
            real,intent(in) :: x
        end function f1

    end interface


    interface
        function f2 (x)
            real :: f2
            real,intent(in) :: x
        end function f2

    end interface

    !************************  DECLARE VARIABLES **********!

    ! Declaring basic variables
    real :: lower, upper, delx, n
    real,dimension(:),allocatable :: xmid
    integer :: i,j
    real :: sum
    integer :: status
    integer :: numberofN
    real :: stopT, startT
    real :: timediff
    real :: analytical
    !Declaring arrays for storage of result
    real,dimension(:),allocatable :: narr
    real,dimension(:),allocatable :: resultarr
    real,dimension(:),allocatable :: analyticarr
    real,dimension(:),allocatable :: errarr
    real,dimension(:),allocatable :: timearr
```

```
!************************  PUT NUMBERS IN **************!

! DEFINE THE UPPER AND LOWER BOUND OF THE INTEGRAL
lower = 0.0
upper = 1.0


!DEFINE THE ANALYTICAL RESULT, WE KNOW IT IS PI

analytical = 4.0*atan(1.0)

! ALLOCATE STORAGE ARRAYS

numberofN = 6.00
allocate(narr(numberofN))
allocate(resultarr(numberofN))
allocate(analyticarr(numberofN))
allocate(errarr(numberofN))
allocate(timearr(numberofN))


! POPULATE ANALYTICAL
do j = 1,numberofN
    analyticarr(j) = analytical
end do


!************************  MAIN SOLVER ******************!


n = 2

do while (n < 200000)
    do j = 1,numberofN
        call cpu_time(startT)
        !   calculate dx
        delx = (upper-lower)/n
        allocate(xmid(int(n)))
        do i = 1,int(n)
            !   populate midpoint x
            xmid(i) = lower+(0.5*delx)+(i-1)*delx
            !   midpoint rule sum
            sum = sum + delx*(f2(xmid(i)))
        end do
```

```
            call cpu_time(stopT)
            timediff = stopT-startT
            !STORE INTO ARRAYS

            timearr(j) = timediff
            narr(j) = n
            resultarr(j) = sum
            !   calulate error
            errarr(j)  = abs(resultarr(j)-analyticarr(j))
            n = n*10
            deallocate(xmid,stat=status)
            sum = 0.0
        end do

    end do


    !*********************WRITE TABLE TO FILE ************!

    ! OPEN UP FILE
    open (21, file = 'results.dat',action = 'write')

    ! Write to file

    write(21,*) ''
    write(21,"(6x,a1,10x,a9,16x,a10,10x,a5,15x,a7)") &
        'N', 'NUMERICAL', 'ANALYTICAL', 'ERROR','TIME(s)'

    write(21,*) ''
    do i = 1,6
        write(21,"(T2,I6,T15,F14.12,T40,F14.12,T60,F9.6,T80,F14.12)"
) int(narr(i)),resultarr(i),&
            analyticarr(i),errarr(i),timearr(i)
    end do
    write(21,*) ''

    !DEALLOCATING THE ARRAYS
    deallocate(narr)
    deallocate(resultarr)
    deallocate(errarr)
    deallocate(analyticarr)
    deallocate(timearr)

    ! CLOSE THE FILE
    close(21)
```

```
end program midpoint



!*************************  FUNCTION PROTOTYPES *************!

function f1(x)
    implicit none
    real :: f1
    real, intent(in) :: x
    f1 = 4.00/(1+x**2)
end function f1



function f2(x)
    implicit none
    real :: f2
    real,intent(in) :: x
    f2 = 4.00*sqrt(1-x**2)
end function f2
```