Loop parallelization

Albert DeFusco

December 13, 2012

Compute π by Integration

- By definition

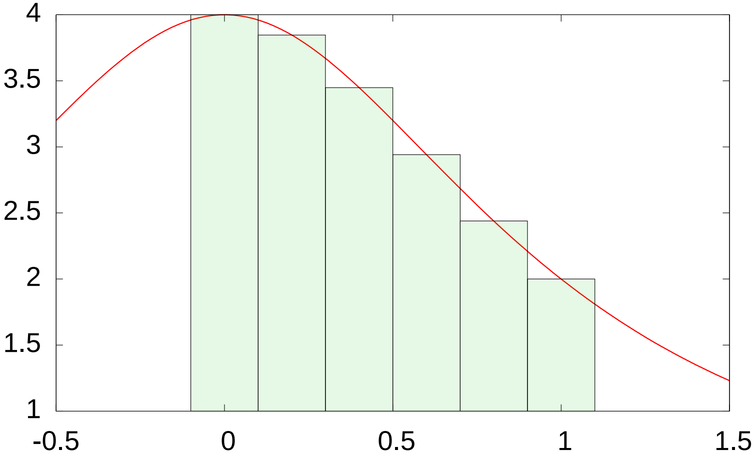
$$\pi = 4\arctan(1) \tag{1}$$

- Many formulas to compute π have centered around approximating $\arctan(1)$ by series

- $\arctan(1)$ is computed by integration

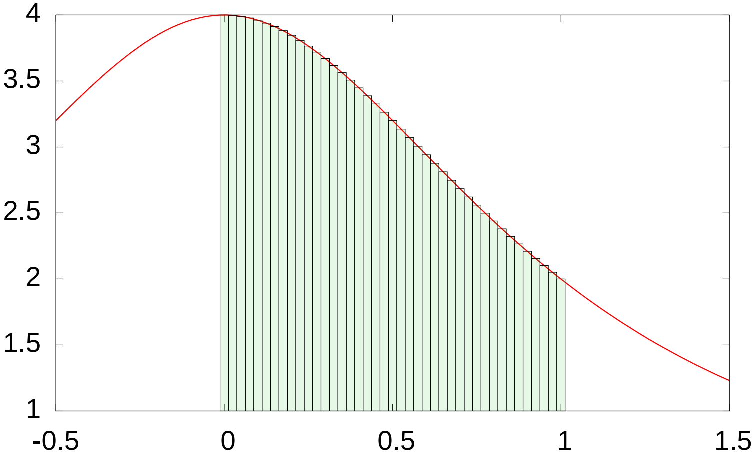
$$\pi = \int_0^1 \frac{4}{1+x^2}dx \tag{2}$$

Compute π by Integration



6 points : $\pi \approx 3.1124$

Compute π by Integration



41 points : $\pi \approx 3.1380$; 10^9 points : $\pi \approx 3.14159265$

Compute π in C++ and Fortran 90

```cpp
#include <iostream>
using namespace std;

double arc(double x);
const long int num_steps=1000000000;
double step;
int main()
{
  double pi, sum=0.0;
  step=1.0/(double) num_steps;

  for (int i=0;i<=num_steps;i++)
  {
    double x=(i+0.5)*step;
    sum+= arc(x);
  }
  pi = sum*step;

  cout.precision(10);
  cout << "pi is probably "
    << fixed << pi << endl;

  return 0;
}
double arc(double x)
{
  double y = 4.0/(1+x*x);
  return y;
}
```

```fortran
program integratePi

  implicit none
  integer(kind=8) :: num_steps,i
  real(kind=8) :: sum,step,pi,x

  num_steps=1000000000
  sum=0.d0
  step=1.d0/num_steps

  do i=1,num_steps
    x=(i+0.5d0)*step
    sum=sum+arc(x)
  enddo

  pi=sum*step

  write(6, &
   '("pi is probably ",f12.10)') &
    pi

  contains
    function arc(x)
      implicit none
      real(kind=8) :: arc,x
      arc = 4.d0/(1.d0+x*x)
    end function arc
end program integratePi
```

Compute π

```
>cp -r /home/sam/training/openmp/basic .
>cd pi
>module load gcc/4.5

#Compile in C++
>CXX pi.cpp -o pi
#Compile in Fortran 90
>FC pi.f90 -o pi

>qsub pi-serial.job
>cat pi.out
pi is probably 3.1415926556

real    0m25.604s
user    0m25.594s
sys     0m0.000s
```

Variable Scopes

- Scope is the lexical context which determines the lifetime of a variable
- In `C++` variables declared in the following regions are "locally" scoped
  - Regions in curly braces
  - Loops
  - Subroutines
- `Fortran` variables are valid within an entire subroutine and `contained` subroutines and functions
- Global variables exist in both languages
  - Variables declared as public in classes or outside of `main` in `C++`
  - Common blocks or modules in Fortran

Variable Scopes

- *lexical scope*
  - The physical text where a variable is valid
- *dynamical scope*
  - The runtime scope of a variable
- Scopes will be extremely important to OpenMP
  - Several clauses exist to control the scoping behavior

Variable Scopes

```cpp
#include <iostream>
using namespace std;

double arc(double x);
const long int num_steps=1000000000;
double step;
int main()
{
  double pi, sum=0.0;
  step=1.0/(double) num_steps;

  for (int i=0;i<=num_steps;i++)
  {
    double x=(i+0.5)*step;
    sum+= arc(x);
  }
  pi = sum*step;

  cout.precision(10);
  cout << "pi is probably "
    << fixed << pi << endl;

  return 0;
}
double arc(double x)
{
  double y = 4.0/(1+x*x);
  return y;
}
```

- `num_steps` and `step` are global
- `pi`, `step` and `sum` are valid for all of `main`
- `i` and `x` are valid only within the `for` loop

Variable Scopes

```fortran
program integratePi

  implicit none
  integer(kind=8) :: num_steps,i
  real(kind=8) :: sum,step,pi,x

  num_steps=1000000000
  sum=0.d0
  step=1.d0/num_steps

  do i=1,num_steps
    x=(i+0.5d0)*step
    sum=sum+arc(x)
  enddo

  pi=sum*step

  write(6, &
   '("pi is probably ",f12.10)') &
    pi

  contains
    function arc(x)
      implicit none
      real(kind=8) :: arc,x
      arc = 4.d0/(1.d0+x*x)
    end function arc
end program integratePi
```

- All variables declared in `program` are available to `arc`
- Variables in `function arc` are private to `arc`

Loop parallelization

- The `omp parallel for` region
  - Reads the loop bounds and divides the work
  - `i` becomes private to each thread in `Fortran` and `C++`
  - What other scopes do we expect?

```c
#pragma omp parallel for
  for (int i=0;i<=num_steps;i++)
  {
    double x=(i+0.5)*step;
    sum+= arc(x);
  }
```

The locally scoped variable `x` becomes private to each thread by default

```fortran
!$omp parallel do
do i=1,num_steps
  x=(i+0.5d0)*step
  sum=sum+arc(x)
enddo
!$omp end parallel do
```

All variables except `i` are shared by default

`parallel` clauses

- Scoping clauses restricted to the lexical extent
  - `private (list)`
    - Independent variables are created for each thread
  - `shared (list)`
    - Can be used for reading and writing by multiple threads
  - `reduction (operator:variable)`
    - A private copy is made for each thread and the reduction operator is applied at the end of the parallel region
    - Operators: `+, *, -, &, |, ^, &&, ||`
  - `default (private|shared|none)`
    - By default all variables are shared
    - Locally scoped variables in C++ are private

Compute π in C++ and Fortran 90 in parallel

```
#pragma omp parallel for reduction(+:sum)
  for (int i=0;i<=num_steps;i++)
  {
    double x=(i+0.5)*step;
    sum+= arc(x);
  }
```

```
  !$omp parallel do reduction(+:sum) &
  !$omp private(x)
  do i=1,num_steps
    x=(i+0.5d0)*step
    sum=sum+arc(x)
  enddo
  !$omp end parallel do
```

`parallel for` region

```
>#edit and compile
>qsub pi-parallel.job
>cat pi.out
pi is probably 3.1415926556

real    0m6.458s
user    0m25.601s
sys     0m0.005s
```

Loop parallelization

- Loops that can be parallelized will have the following
  - All assignments are made to arrays
  - Each element is assigned by at most one iteration
  - No iteration reads elements assigned by another iteration
  - Blocks must have one entrance and exit

Loop parallelization

- Loops with data dependence cannot be parallelized
  - `a[i] = x*a[i-1]`
  - Data dependence exists through the *dynamical* extent of a parallel region
- Loops with an undefined number of iteration cannot be parallelized
  - `while(!converged)`
- Race conditions
  - `sum=sum+i`
  - Many can be cured with the `reduction` clause
- Use control statements and divergent iterations carefully
  - `goto` must be within the block
  - `break, continue` must be within the block
  - `stop` and `exit` are allowed

Loop nesting

- In nested loops, both loops cannot be parallel
  - The inner loop would not be computed in parallel since all of the threads are already used in the outer loop

```
for(int y=0; y<25; ++y)
{
  for(int x=0; x<80; ++x)
  {
    a[y] += b[x];
  }
}
```

- In this example only the outer loop can be parallelized due to data dependence of `a[y]`

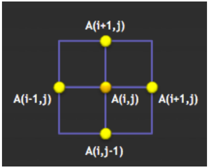
Exercises

Loop parallelization

- Copy `/home/sam/training/openmp/basic/`
- Parallelize the loops in `laplace2d`, `matrixvector`

Jacobi Iterations[1]

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

[1] Adapted from a recent PSC workshop

Loop parallelization

- Parallelize the loops in `matrixvector`

```
>cd matrixvector
>#edit mxm.c
>module purge
>module load gcc/4.5
>CXX mxm.c -fopenmp -o matrixvector
>qsub matrixvector.job
>#inspect omp.out
```