Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

MCS 572 Lecture 10
Introduction to Supercomputing
Jan Verschelde, 18 September 2024

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

the composite trapezoidal rule for π

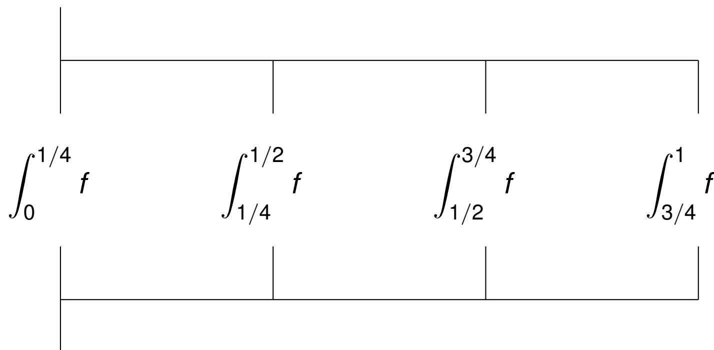
We can approximate π via $\displaystyle\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2} dx$.

The trapezoidal rule for $\displaystyle\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a) + f(b))$.

Using n subintervals of $[a, b]$:

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) + h \sum_{i=1}^{n-1} f(a + ih), \quad h = \frac{b-a}{n}.$$

integration with four threads



Each thread has its own a, b, and $c = \int_a^b f$.

processes and threads

With MPI, we identified processors with processes:
in `mpirun -p` as `p` is larger than the available cores,
as many as `p` processes are spawned.

Main difference between a process and a thread:

- A process is a completely separate program
  with its own variables and memory allocation.
- Threads share the same memory space
  and global variables between routines.

A process can have many threads of execution.

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

the composite trapezoidal rule in Julia

```
"""
    function traprule(f::Function,
                      a::Float64, b::Float64,
                      n::Int)

returns the composite trapezoidal rule to
approximate the integral of f over [a,b]
using n function evaluations.
"""
function traprule(f::Function,
                  a::Float64, b::Float64,
                  n::Int)
    h = (b-a)/n
    y = (f(a) + f(b))/2
    x = a + h
    for i=1:n-1
        y = y + f(x)
        x = x + h
    end
    return h*y
end
```

using Threads in Julia

```
using Printf
using Base.Threads

nt = nthreads()
println("The number of threads : $nt")
subapprox = zeros(nt)

f(x) = sqrt(1 - x^2)
dx = 1/nt
bounds = [i for i=0:dx:1]

timestart = time()
@threads for i=1:nt
    subapprox[i] = traprule(f, bounds[i], bounds[i+1], 1_000_000)
end
approxpi = 4*sum(subapprox)
elapsed = time() - timestart

println("The approximation for Pi : $approxpi")
err = @sprintf("%.3e", pi - approxpi)
println("with error : $err")
println("The elapsed time : $elapsed seconds")
```

on 2020 M1 MacBook Air, 8 GB memory

```
% JULIA_NUM_THREADS=1 julia mtcomptrap.jl
The number of threads : 1
The approximation for Pi : 3.141592652402481
with error : 1.187e-09
The elapsed time : 0.09319114685058594 seconds

% JULIA_NUM_THREADS=2 julia mtcomptrap.jl
The number of threads : 2
The approximation for Pi : 3.1415926532747154
with error : 3.151e-10
The elapsed time : 0.10139918327331543 seconds

% JULIA_NUM_THREADS=4 julia mtcomptrap.jl
The number of threads : 4
The approximation for Pi : 3.1415926533683467
with error : 2.214e-10
The elapsed time : 0.10673189163208008 seconds

% JULIA_NUM_THREADS=8 julia mtcomptrap.jl
The number of threads : 8
The approximation for Pi : 3.1415926534974554
with error : 9.234e-11
The elapsed time : 0.12252998352050781 seconds
```

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

about OpenMP

The collection of

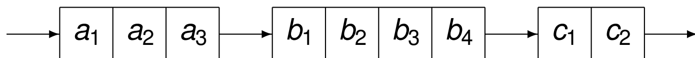
1. compiler directives (specified by `#pragma`)
2. library routines (call `gcc -fopenmp`)
   e.g.: to get the number of threads
3. environment variables
   (e.g.: number of threads, scheduling policies)

defines collectively the specification of the OpenMP API
for shared-memory parallelism in C, C++, and Fortran programs.

OpenMP offers a set of compiler directives to extend C/C++.

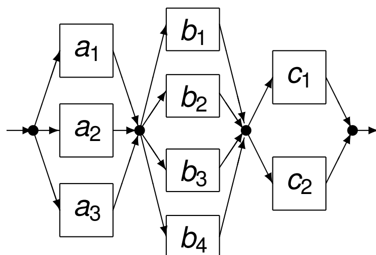
the OpenMP Execution Model

An execution on one thread:



Suppose a_1, a_2, a_3 can be executed in parallel,
and similarly for the execution of b_1, b_2, b_3, b_4, and of c_1, c_2.

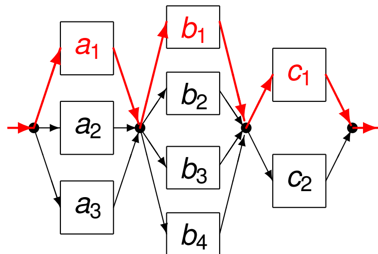
An execution with parallel regions:



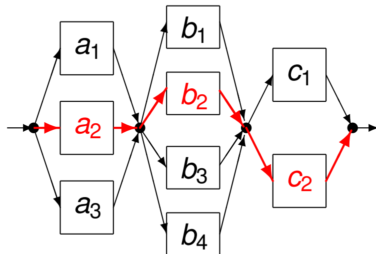
• : threads start and/or join

running a crew of four threads

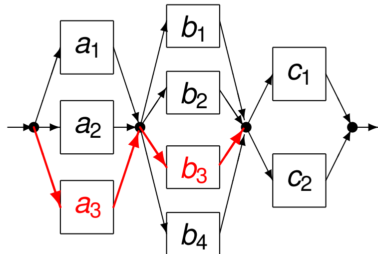
thread 0



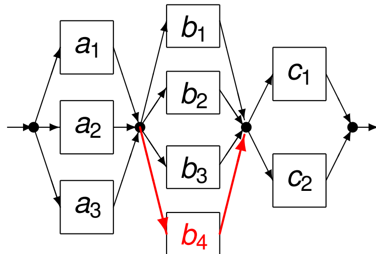
thread 1



thread 2



thread 3



Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

hello world!

```
#include <stdio.h>
#include <omp.h>

int main ( int argc, char *argv[] )
{
   omp_set_num_threads(8);

   #pragma omp parallel
   {
      #pragma omp master
      {
         printf("Hello from the master thread %d!\n",
                omp_get_thread_num());
      }
      printf("Thread %d says hello.\n",
             omp_get_thread_num());
   }
   return 0;
}
```

compiling and running

On Mac OS X, install with `brew install libomp`.

```
% make hello_openmp0
gcc -fopenmp hello_openmp0.c -o hello_openmp0

% ./hello_openmp0
Thread 1 says hello.
Thread 2 says hello.
Thread 3 says hello.
Thread 4 says hello.
Thread 5 says hello.
Thread 6 says hello.
Hello from the master thread 0!
Thread 0 says hello.
Thread 7 says hello.
%
```

Introduction to OpenMP

1 Programming Shared Memory Parallel Computers
- the composite trapezoidal rule for π
- multithreading in Julia

2 the OpenMP Application Program Interface
- running crews of threads
- our first program with OpenMP
- compiler directives and library routines
- passing the number of threads at the command line

3 Numerical Integration with OpenMP
- multithreading using OpenMP
- private variables and critical sections

library routines

We compile with `gcc -fopenmp` and put

```
#include <omp.h>
```

at the start of the program.

The program `hello_openmp0.c` uses two OpenMP library routines:

- `void omp_set_num_threads ( int n );`

  sets the number of threads to be used
  for subsequent parallel regions.

- `int omp_get_thread_num ( void );`

  returns the thread number, within the current team,
  of the calling thread.

the `parallel` construct

We use the `parallel` construct as

```
#pragma omp parallel
{
   S1;
   S2;
   ...
   Sm;
}
```

to execute the statements `S1`, `S2`, ..., `Sm` in parallel.

the `master` construct

```
#pragma omp parallel
{
   #pragma omp master
   {
      printf("Hello from the master thread %d!\n",
             omp_get_thread_num());
   }
   /* instructions omitted */
}
```

The master construct specifies a structured block that is executed by the master thread of the team.

the `single` construct

Extending the `hello_openmp0.c` program with

```
#pragma omp parallel
{
   /* instructions omitted */
   #pragma omp single
   {
      printf("Only one thread %d says more ...\n",
             omp_get_thread_num());
   }
}
```

The single construct specifies that the associated block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.
The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the single construct.

running example of the single construct

```
% ./hello_openmp1
Thread 1 says hello.
Only one thread 1 says more ...
Thread 3 says hello.
Thread 2 says hello.
Thread 4 says hello.
Thread 6 says hello.
Hello from the master thread 0!
Thread 0 says hello.
Thread 7 says hello.
Thread 5 says hello.
%
```

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia

2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line

3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

passing the number of threads at the command line

We start our C programs as

```
#include <stdio.h>
#include <omp.h>

int main ( int argc, char *argv[] )
```

In every C program

1. `argc` is the number of command line arguments,
   the first argument is the name of the executable.
2. `argv` is an array of `argc` strings,
   with the command line arguments.

Application: pass the number of threads when launching the program.

extending our first `hello_openmp0` again

```
% ./hello_openmp2 4
Thread 1 says hello.
Hello from the master thread 0!
Thread 0 says hello.
Thread 3 says hello.
Thread 2 says hello.
%
```

If the user does not specify the number of threads:

```
% ./hello_openmp2
Please specify the number threads,
as the first argument of the program.
%
```

using command line arguments

```
#include <stdio.h>
#include <omp.h>

int main ( int argc, char *argv[] )
{
   if(argc == 1)
   {
      printf("Please specify the number threads,\n");
      printf("as the first argument of the program.\n");

      return 1;
   }
   else
   {
      int nbtreads = atoi(argv[1]);

      omp_set_num_threads(nbtreads);
   }
```

the function traprule

The first argument of the C function for the composite trapezoidal rule is the function that defines the integrand f.

```
double traprule
 ( double (*f) ( double x ), double a, double b, int n )
{
   int i;
   double h = (b-a)/n;
   double y = (f(a) + f(b))/2.0;
   double x;

   for(i=1,x=a+h; i < n; i++,x+=h) y += f(x);

   return h*y;
}
```

the main program

```
double integrand ( double x )
{
   return sqrt(1.0 - x*x);
}

int main ( int argc, char *argv[] )
{
   int n = 1000000;
   double my_pi = 0.0;
   double pi,error;

   my_pi = traprule(integrand,0.0,1.0,n);
   my_pi = 4.0*my_pi; pi = 2.0*asin(1.0); error = my_pi-pi;
   printf("Approximation for pi = %.15e \
          with error = %.3e\n", my_pi,error);

   return 0;
}
```

running on one core on the M1 MacBook Air

Evaluating $\sqrt{1 - x^2}$ one million times:

```
% make comptrap
gcc comptrap.c -o comptrap -lm

% /usr/bin/time ./comptrap
Approximation for pi = 3.141592652402481e+00 \
with error = -1.187e-09

0.01 real
0.01 user
0.00 sys
```

This took 10 milliseconds ...

Last line of `gcc -v` output:

```
gcc version 12.0.1 20220312 (experimental) (GCC)
```

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

the private clause of parallel

```
int main ( int argc, char *argv[] )
{
   int i;
   int p = 8;
   int n = 1000000;
   double my_pi = 0.0;
   double a,b,c,h,y,pi,error;

   omp_set_num_threads(p);

   h = 1.0/p;

   #pragma omp parallel private(i,a,b,c)
   /* each thread has its own i,a,b,c */
   {
```

updating in a critical section

```
    #pragma omp parallel private(i,a,b,c)
    /* each thread has its own i,a,b,c */
    {
       i = omp_get_thread_num();
       a = i*h;
       b = (i+1)*h;
       c = traprule(integrand,a,b,n);
       #pragma omp critical
       /* critical section protects shared my_pi */
          my_pi += c;
    }
    my_pi = 4.0*my_pi; pi = 2.0*asin(1.0); error = my_pi-pi;
    printf("Approximation for pi = %.15e \
            with error = %.3e\n",my_pi,error);

    return 0;
}
```

Introduction to OpenMP

1. Programming Shared Memory Parallel Computers
   - the composite trapezoidal rule for π
   - multithreading in Julia
2. the OpenMP Application Program Interface
   - running crews of threads
   - our first program with OpenMP
   - compiler directives and library routines
   - passing the number of threads at the command line
3. Numerical Integration with OpenMP
   - multithreading using OpenMP
   - private variables and critical sections

private variables

A *private variable* is a variable in a parallel region providing access to a different block of storage for each thread.

```
#pragma omp parallel private(i,a,b,c)
/* each thread has its own i,a,b,c */
{
   i = omp_get_thread_num();
   a = i*h;
   b = (i+1)*h;
   c = traprule(integrand,a,b,n);
```

Thread `i` integrates from `a` to `b`, where `h = 1.0/p` and stores the result in `c`.

the critical construct

The critical construct restricts execution of the associated structured block in a single thread at a time.

```
#pragma omp critical
/* critical section protects shared my_pi */
   my_pi += c;
```

A thread waits at the beginning of a critical region until no threads is executing a critical region.
The critical construct enforces exclusive access.

In the example, no two threads may increase my_pi simultaneously.

running with 8 threads

```
% /usr/bin/time ./comptrap_omp
Approximation for pi = 3.141592653497455e+00 \
with error = -9.234e-11

0.01 real
0.01 user
0.00 sys
```

Compare on one core (`error = -1.187e-09`):

```
0.01 real
0.01 user
0.00 sys
```

Same times but the 8-threaded result is much more accurate.

Summary + Exercises

OpenMP Application Program Interface Version 5.2 Nov 2021 is available at http://www.openmp.org.

Exercises:

1. Modify the `hello world!` program with OpenMP so that the master thread prompts the user for a name which is used in the greeting displayed by thread 5. Note that only one thread, the one with number 5, greets the user.
2. Modify the `comptrap_omp` program so that the number of threads is passed at the command line.
   Run experiments with various number of threads, starting at 2 and then doubling to 4, 8, 16, and 32. Make a table with wall clock times (the reported `real`) and the errors.
3. Compute the `flops` (floating point operations per second) for the `comptrap_omp` run in the lecture slides.