

Introduction to OpenMP

- 1 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - with 8 computing threads using OpenMP
 - private variables and critical sections

MCS 572 Lecture 9
Introduction to Supercomputing
Jan Verschelde, 30 January 2012

Introduction to OpenMP

- 1 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - with 8 computing threads 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++.

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 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - with 8 computing threads 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

```
$ make hello_openmp0  
gcc -fopenmp hello_openmp0.c -o /tmp/hello_openmp0
```

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

Introduction to OpenMP

- 1 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - **compiler directives and library routines**
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - with 8 computing threads 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_omp0.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.

Introduction to OpenMP

- 1 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 **Numerical Integration with OpenMP**
 - **the composite trapezoidal rule for π**
 - with 8 computing threads using OpenMP
 - private variables and critical sections

the composite trapezoidal rule for π

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

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

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

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

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 at 3.47 Ghz

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

```
$ make comptrap
gcc comptrap.c -o /tmp/comptrap -lm

$ time /tmp/comptrap
Approximation for pi = 3.141592652402481e+00 \
with error = -1.187e-09

real    0m0.017s
user    0m0.016s
sys     0m0.001s
```

Introduction to OpenMP

- 1 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - **with 8 computing threads 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 the OpenMP Application Program Interface
 - programming shared memory parallel computers
- 2 using OpenMP
 - our first program with OpenMP
 - compiler directives and library routines
- 3 Numerical Integration with OpenMP
 - the composite trapezoidal rule for π
 - with 8 computing threads 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 on 8 cores

```
$ make comptrap_omp  
gcc -fopenmp comptrap_omp.c -o /tmp/comptrap_omp -lm
```

```
$ time /tmp/comptrap_omp  
Approximation for pi = 3.141592653497455e+00 \  
with error = -9.234e-11
```

```
real    0m0.014s  
user    0m0.089s  
sys     0m0.001s  
$
```

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

```
real    0m0.017s  
user    0m0.016s  
sys     0m0.001s
```

interpretation of the results

Summarizing the results:

	real time	error
1 thread	0.017s	-1.187e-09
8 threads	0.014s	-9.234e-11

In the multithreaded version, every thread uses 1,000,000 subintervals.

The program with 8 threads does 8 times more work than the program with 1 thread.

Summary + Exercises

In the book by Wilkinson and Allen, §8.5 is on OpenMP.
OpenMP Application Program Interface Version 3.1 July 2011
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 Consider the Monte Carlo simulations we have developed with MPI for the estimation of π . Write a version with OpenMP and examine the speedup.

one more exercise

- 3 Write an OpenMP program to simulate the management of a bank account, with the balance represented by a single shared variable. The program has two threads. Each thread shows the balance to the user and prompts for a debit (decrease) or a deposit (increase). Each thread then updates the balance in a critical section and displays the final the balance to the user.

Homework will be collected on Friday 3 February at noon.
Bring your answers to the problems of lectures 6, 7, and 8 to class.
You may work in pairs. Please hand in only one copy per pair.