Parallel in C (OpenMP)

Frederick Fung

National Computational Infrastructure, Australia

Acknowledgement of Country

Outline

Goal

This is a step-by-step introductory level OpenMP training session. The goal of this session is to get you familiar with some common practice of OpenMP and explore concurrency in for-loop.

1. The concepts
2. The coding exercises.

Introduction

When do I need OpenMP?

Introduction

When do I need OpenMP?

Among the choices of MPI, OpenCL, OpenAcc, CUDA, OpenSHMEM

Introduction

OpenMP is an Application Program Interface (API) that accommodates easy multithread programming on shared memory.

Introduction

OpenMP is an Application Program Interface (API) that accommodates easy multithread programming on shared memory.

This is ideal for programs running on single node of a cluster.

Introduction

OpenMP is an Application Program Interface (API) that accommodates easy multithread programming on shared memory.

This is ideal for programs running on single node of a cluster.

For programs need to be parallel between multi-processors, consider MPI or hybrid MPI/OpenMP.

Threading: A schematic diagram

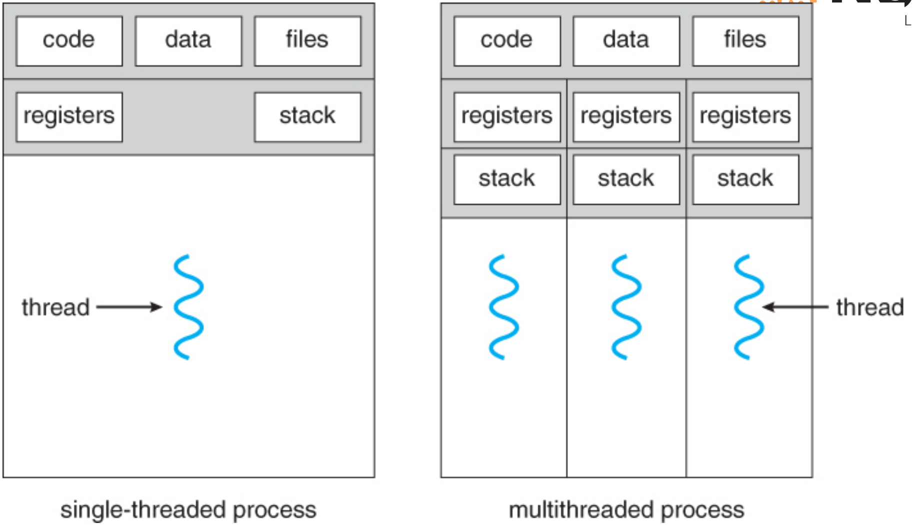

Figure 1: *source: https://www.cs.miami.edu/home/wuchtys/CSC322-21S/Content/UNIXProgramming/UNIXThreads.shtml*

Threading

Threads are NOT physical components of computers. A thread is a group of instructions to be executed by a core. Only one thread can run on a core at a time.

Threading

Threads are NOT physical components of computers. A thread is a group of instructions to be executed by a core. Only one thread can run on a core at a time.

When multithreads fork from a process, a core executes each thread for a short period of time and switch to other threads. Effectively, multiple threads are executed concurrently. (**Concurrency**)

Threading

Threads are NOT physical components of computers. A thread is a group of instructions to be executed by a core. Only one thread can run on a core at a time.

When multithreads fork from a process, a core executes each thread for a short period of time and switch to other threads. Effectively, multiple threads are executed concurrently. (**Concurrency**)

When multiple cores are used, the same number of threads can be executed in parallel. (**Parallelism**)

Threading

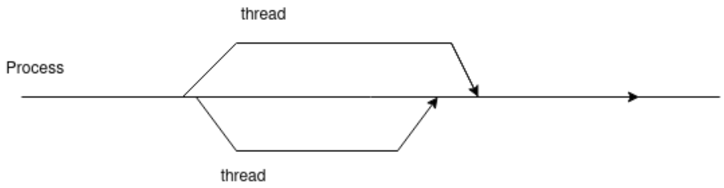

Figure 2: Three threads are forked from a process. They share most of the same execution resources in the process including a shared memory space[Pac11].

OpenMP: Overview

OpenMP is a compiler directive based API. To convert a serial program into a parallel program to run on a shared memory device, OpenMP uses compiler directives to tell the compiler to create a multithreaded version of the program. This limits the overhead of modifying codes to a minimal level for users.

OpenMP: Overview

Following closely to the principles in [MHK19], our hand-on session focus on using OpenMP to write multithreaded programs from three key ingredients:

1. Compiler Directives,
2. OpenMP Library,
3. Data Environment.

Example: Monte-Carlo Approximation of π

In the next part of our session, we use a Monte-Carlo method as a showcase to OpenMP programming.

Example: Monte-Carlo Approximation of π

In the next part of our session, we use a Monte-Carlo method as a showcase to OpenMP programming.

Generally speaking, Monte-Carlo method classifies a group of numerical algorithms that utilise random sampling to acquire numerical approximations.

Example: Monte-Carlo Approximation of π

In the next part of our session, we use a Monte-Carlo method as a showcase to OpenMP programming.

Generally speaking, Monte-Carlo method classifies a group of numerical algorithms that utilise random sampling to acquire numerical approximations.

The particular one that we are going to inspect is using the Monte-Carlo to approximate the value of π. Imagine scattering a handful of rice.

Example: Monte-carlo Approximation of π

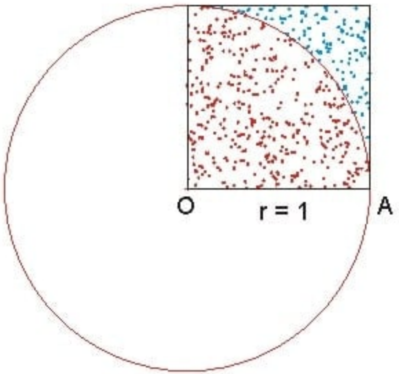

Figure 3: Generate N random sampling points within a square, and count the number h of samples that fall in the circle. Then the approximation $\pi \approx 4h/N$

Example: Monte-Carlo Approximation of π

A serial code

```
seed  = 1; /* seed to generate random numbers */
for (i=0; i<N; i++){
  x = rand_r(&seed)/ (double) RAND_MAX; /* RAND_MAX to normalise */
  y = rand_r(&seed)/ (double) RAND_MAX;

if (x*x + y*y <= 1.0f) count+=1;
}
```

x, *y* are normalised random numbers. Variable declarations are not shown.

Next we are going to walk through step by step to make this a parallel program.

Step 1: Parallel Construct

The parallel region of the code to be executed in multiple threads is specified by a parallel construct in OnpenMP. A parallel construct is composed of the directive and the structured block.

```
seed  = 1;
#pragma omp parallel[clause[[,], clause]...]
{ for (i=0; i<N; i++){
  x = rand_r(&seed)/ (double) RAND_MAX;
  y = rand_r(&seed)/ (double) RAND_MAX;

if (x*x + y*y <= 1.0f) count+=1;
}
}// end of parallel region
```

OpenMP Library Routines

Step 2: Library Routines

1
```
    // request number of threads
    void omp_set_num_threads(int num_threads);
```

OpenMP Library Routines

Step 2: Library Routines

```
1      // request number of threads
       void omp_set_num_threads(int num_threads);
2      // return the thread rank in the team
       int omp_get_thread_num();
```

OpenMP Library Routines

Step 2: Library Routines

```
1   // request number of threads
    void omp_set_num_threads(int num_threads);
2   // return the thread rank in the team
    int omp_get_thread_num();
3   // total number of threads in the team
    int omp_get_num_threads();
```

Parallelism Methodology: Cyclic distribution

Cyclic Distribution

Each thread works on loop iterations by increments determined by the rank of the thread.

```
#pragma omp parallel
{ int id = omp_get_thread_num();
  int numthreads = omp_get_num_threads();
  for (i=id; i<N; i+numthreads){
  seed = numthreads;
  x = rand_r(&seed)/ (double) RAND_MAX;
  y = rand_r(&seed)/ (double) RAND_MAX;

if (x*x + y*y <= 1.0f) count+=1;
}
}// end of parallel region
```

Parallelism Methodology: Block Distribution

Block Distribution

Each threads is given an approximately equal sized block of consecutive loop iterations

```
#pragma omp parallel
{ int id = omp_get_thread_num();
  int numthreads = omp_get_num_threads();
  int start = id * N / numthread;
  int end = (id+1) * N /numthread;
  for (i=start; i<end; i++){
  seed = numthreads;
  x = rand_r(&seed)/ (double) RAND_MAX;
  y = rand_r(&seed)/ (double) RAND_MAX;
if (x*x + y*y <= 1.0f) count+=1;
}
}// end of parallel region
```

SPMD (Single Program Multiple Data)

Both of the two methods we discussed about use the SPMD pattern, i.e., the programmer has to define the copies of data for each thread.

SPMD (Single Program Multiple Data)

Both of the two methods we discussed about use the SPMD pattern, i.e., the programmer has to define the copies of data for each thread.

This can be error-prone since it modifies the code significantly by introducing new variables for each thread.

SPMD (Single Program Multiple Data)

Both of the two methods we discussed about use the SPMD pattern, i.e., the programmer has to define the copies of data for each thread.

This can be error-prone since it modifies the code significantly by introducing new variables for each thread.

Therefore, the two methodologies are difficult to implement. We let OpenMP to handle this for us

Worksharing-loop Construct

Step 3: Worksharing-loop Construct

To avoid SPMD pattern, OpenMP uses worksharing-loop construct to achieve loop-level parallelism. In essence, it leaves the compiler to split the loop iterations and the user does not need to modify the for-loop.

Worksharing-loop Construct

Step 3: Worksharing-loop Construct

To avoid SPMD pattern, OpenMP uses worksharing-loop construct to achieve loop-level parallelism. In essence, it leaves the compiler to split the loop iterations and the user does not need to modify the for-loop.

```
#pragma omp parallel
{seed = omp_get_thread_num();
#pragma omp for
for (i=0; i<N; i++){
    x = rand_r(&seed)/ (double) RAND_MAX;
    y = rand_r(&seed)/ (double) RAND_MAX;
    if (x*x + y*y <= 1.0f) count+=1;
    }
}
```

Combined Worksharing-loop Construct

If the Parallel construct is immediately followed by a worksharing-loop construct, then both directive can be combined. Hence, the following two patterns are equivalent.

```
#pragma omp parallel
{
#pragma omp for
  //for loop
}
```

```
#pragma omp parallel for
  //for loop
```

Synchronisation Construct

Race Condition

When a variable is shared between multiple threads, multiple threads can potentially access the same memory location concurrently. Then the update has unpredictable behaviour. It is seen in the count variable in our example.

```
#pragma omp parallel
{
 /* skipped */
#pragma omp for
     /* skipped */
     if (x*x + y*y <= 1.0f) count+=1;
}
```

Threading: A schematic diagram

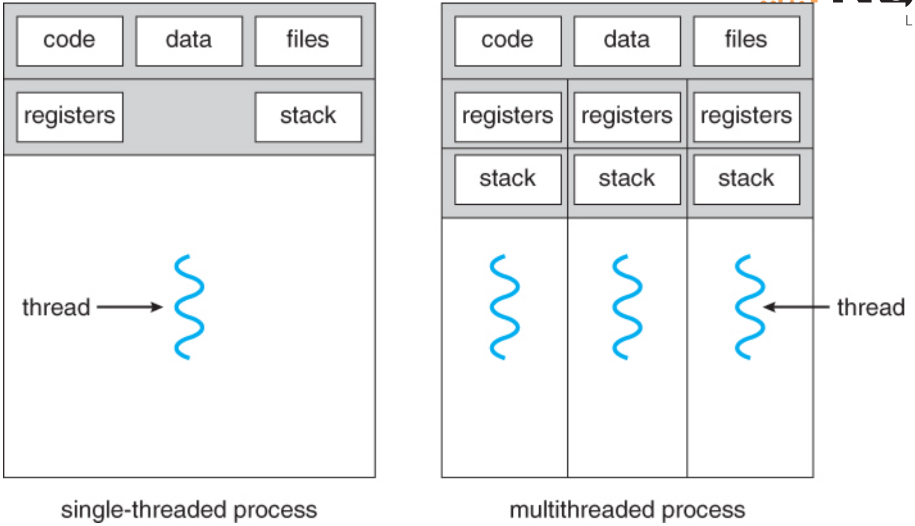

Figure 4: *source: https://www.cs.miami.edu/home/wuchtys/CSC322-21S/Content/UNIXProgramming/UNIXThreads.shtml*

Race Condition

Time	Thread A	count	Thread B
T1	Read	i	
T2		i	Read
T3	Add	i	
T4		i	Add
T5	Write	i+1	
T6		i +1	Write

Table 1: Two thread update a value at the same memory location.

Synchronisation Construct

Step 4.5: Synchronisation

A critical construct enforces the attached block of code to be executed with mutual exclusion.

```
#pragma omp parallel
{
 /* skipped */
#pragma omp for
    /* skipped */
    if (x*x + y*y <= 1.0f)
    {
      #pragma omp critical
      count+=1;
    }
}
```

Critical

Time	Thread A	count	Thread B
T1	Read	i	
T2	Add	i	
T3	Write	i+1	

Table 2: Thread A arrives the critical region first, thread A gets to execute the critical construct exclusively.

Synchronisation Construct

Critical

Although using critical construct renders the thread safety, it increases the serial portion. In some scenarios, if the critical block takes up a considerable amount of time then the code essentially gets serialised.

Synchronisation Construct

Critical

Although using critical construct renders the thread safety, it increases the serial portion. In some scenarios, if the critical block takes up a considerable amount of time then the code essentially gets serialised.

We have a better way to avoid using the expensive synchronisation in this case by smartly allocate data storage attributes. (We will come back on this)

Synchronisation Construct

Atomic

Similar to the critical construct, the atomic construct occurs with the mutual exclusion. Due to the different implementation, atomic has less overhead but the trade-off is that it also has more specific use cases.

Synchronisation Construct

Atomic

Similar to the critical construct, the atomic construct occurs with the mutual exclusion. Due to the different implementation, atomic has less overhead but the trade-off is that it also has more specific use cases.

More precisely, atomic only guarantees the mutual exclusion for operations that directly involves the storage location in memory.

Synchronisation Construct

```
#pragma omp atomic
count+=1; /* thread safety protected */
```

```
#pragma omp atomic
count += calc_count() /* potential data race */
```

```
#pragma omp critical
count += calc_count() /* thread safety protected */
```

Synchronisation Construct

Barrier

A barrier defines a point at which all threads must arrive before any thread may proceed past the barrier.

Synchronisation Construct

Barrier

A barrier defines a point at which all threads must arrive before any thread may proceed past the barrier.

```
#pragma omp barrier
```

Synchronisation Construct

Barrier

A barrier defines a point at which all threads must arrive before any thread may proceed past the barrier.

```
#pragma omp barrier
```

We've been implicitly using barriers! Every worksharing construct has a barrier at the end to guard the thread safety.

Threading

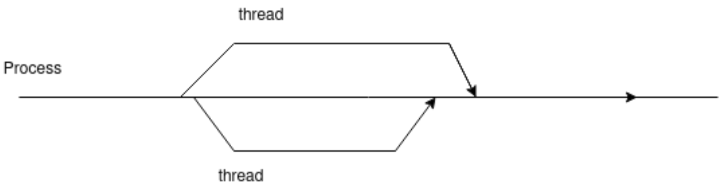

Figure 5: Three threads are forked from a process. A barrier is places before they join as one master thread.[Pac11]

Synchronisation Construct

Nowait

A barrier is very expensive, the waiting time depends on the slowest thread. The nowait construct omits a barrier if it is determined not needed.

```
#pragma omp for nowait /* omits the barrier for
                          the worksharing-loopF */
```

Synchronisation Construct

Nowait

A barrier is very expensive, the waiting time depends on the slowest thread. The nowait construct omits a barrier if it is determined not needed.

```
#pragma omp for nowait /* omits the barrier for
                            the worksharing-loopF */
```

```
#pragma omp parallel
{
#pragma omp for
  {  /* for loop */ } /* compiler places nowait */
}
```

Worksharing-loop Construct

Recall that we used critical construct (or atomic) to avoid the data race from updating the count variable. However, this serialises the multithreaded code.

Worksharing-loop Construct

Recall that we used critical construct (or atomic) to avoid the data race from updating the count variable. However, this serialises the multithreaded code.

The race condition raises when multiple threads try to write to the same memory location. This condition will not present itself if every thread has its private count variable.

Worksharing-loop Construct

Recall that we used critical construct (or atomic) to avoid the data race from updating the count variable. However, this serialises the multithreaded code.

The race condition raises when multiple threads try to write to the same memory location. This condition will not present itself if every thread has its private count variable.

Reduction

Creates copy of a scalar variable local to each thread. Uses the clause operator at end of the worksharing-construct to combine all local copies into a shared global variable.

Worksharing-loop Construct

Step 5: Reduce the Scalar

Reduction

```
reduction (operation:variable)
```

```
#pragma omp parallel
{
 /* not shown */
#pragma omp for reduction(+:count)
    /* for loop not shown */
    if (x*x + y*y <= 1.0f) count+=1;
}
```

Exercise: Add parallel construct and worksharing-loop construct to turn the Monte-Carlo π program into parallel.

Data Storage Attribute

Step 6: Storage Attribute

Shared

A variable is shared if it only has one global copy and can be accessed by all threads in the team.

```
#pragma omp parallel shared(list)
```

Private

The private clause tells the compiler to create a new variable for the same type and the same name for each thread. The private variable is uninitialised in each thread.

```
#pragma omp parallel private(list)
```

Data Storage Attribute

Firstprivate

Like private clause, firstprivate tells compiler to create private copy for each thread. The only difference is that with firstprivate, the new private variable is initialised by copying the value of the original variable.

```
#pragma omp parallel firstprivate(list)
```

What variables in Monte-Carlo π should be created as private? We didn't add the private clause and why it still worked?

```
#pragma omp parallel
{seed = omp_get_thread_num();
#pragma omp for
for (i=0; i<N; i++){
    x = rand_r(&seed)/ (double) RAND_MAX;
    y = rand_r(&seed)/ (double) RAND_MAX;
    if (x*x + y*y <= 1.0f) count+=1;
    }
}
```

Data Storage Attribute

Dafualt(none)

Data storage attribute is a very error-prone component of OpenMP programming (as we see in the Monte-Carlo π), use default(none) to force yourself to explicitly list the correct storage attributes on variables.

```
#pragma omp parallel default(none)
```

More Synchronisation

Ordered

The ordered enforces the execution to run sequentially in the same order as if it was executed in serial.

```
#pragma omp ordered
```

Loop Schedule

Step 7: Scheduling

The default worksharing-loop construct leaves the compiler to decide how to split a loop.

Loop Schedule

Step 7: Scheduling

The default worksharing-loop construct leaves the compiler to decide how to split a loop.

Recall we mentioned two parallel methodologies and they are equivalent in a sense that every thread gets an equivalent amount of work if loop iterations repeat the same execution. It is the case in our Monte-Carlo π exercise.

Loop Schedule

Static Schedule

Static means that the loop schedule is managed at compile time. The static schedule has a chunk parameter. When the chunk is not specified, the loop is split as described in the second methodology (block distribution). If the chunk is given, it is similar to the second methodology (cyclic distribution).

Loop Schedule

Static Schedule

Static means that the loop schedule is managed at compile time. The static schedule has a chunk parameter. When the chunk is not specified, the loop is split as described in the second methodology (block distribution). If the chunk is given, it is similar to the second methodology (cyclic distribution).

```
#prgma omp for schedule(static, chunk)
```

Loop Schedule

As a programmer, we know better than the compiler about the execution of our codes. Can we schedule the loop that is more efficient?

Loop Schedule

As a programmer, we know better than the compiler about the execution of our codes. Can we schedule the loop that is more efficient?

Dynamic Schedule

If we know the work for each loop iteration is a variable and can only be determined at the runtime, we may let the OS to decide the assignment of iterations to the threads. (Be careful, it comes with the cost of overheads.)

```
#prgma omp for schedule(dynamic, chunk)
```

In the next exercise, we will use most of the constructs and clauses that we've learnt today to concurrently generate datasets and visualise some pretty fractal pictures!

Mandelbrot Set

Definition

The Mandelbrot set is the set of complex numbers c for which the function $f_c(z_{i+1}) = z_i^2 + c$ does not diverge to infinity when iterated from $z_0 = 0$

$$M := \{c \in \mathbb{C} \big| z_{i+1} = z_i^2 + c < \infty, z_0 = 0, \quad i = 0, 1, \cdots, N\}$$

Mandelbrot Set

Fractal Structure

The image of Mandelbrot Set exhibits the self-similar fractal structure. It is also an infinite set, more points means higher resolution and we can see more fine-grained details of the fractal image!

Mandelbrot Set

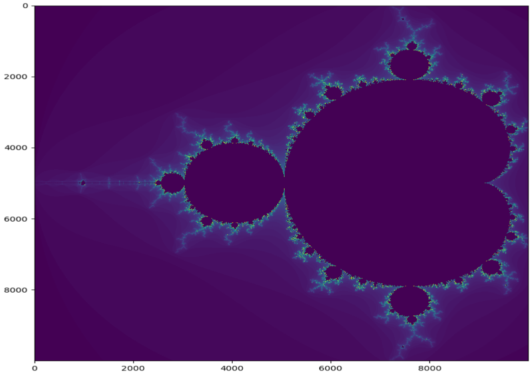

Figure 6: Mandelbrot set

Mandelbrot Set

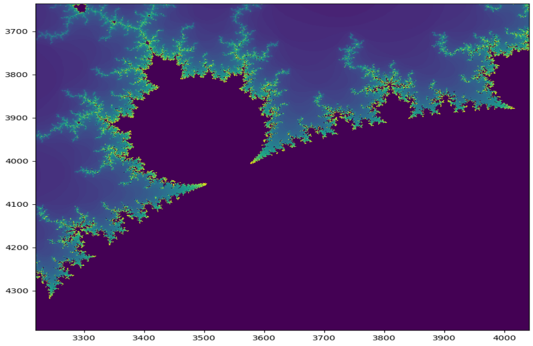

Figure 7: Mandelbrot set zoomed in

Task: Add a parallel construct and appropriate clauses to generate the Mandelbrot set in parallel.

Things to think about:

1. What storage attributes should be given to the variables?

Task: Add a parallel construct and appropriate clauses to generate the Mandelbrot set in parallel.

Things to think about:

1. What storage attributes should be given to the variables?
2. What schedule should be assigned to the for-loop?

Task: Add a parallel construct and appropriate clauses to generate the Mandelbrot set in parallel.

Things to think about:

1. What storage attributes should be given to the variables?
2. What schedule should be assigned to the for-loop?
3. Which part of the code needs to be synchronised?

Task: Add a parallel construct and appropriate clauses to generate the Mandelbrot set in parallel.

Things to think about:

1. What storage attributes should be given to the variables?
2. What schedule should be assigned to the for-loop?
3. Which part of the code needs to be synchronised?

Conjugate Gradient Method

The CG method is a numerical iterative direct solver for solving linear system:

$$Ax = b, \tag{1}$$

where A is a symmetric positive definite matrix.

Conjugate Gradient Method

Algorithm 1: Standard CG (A, b, tolerance)

```
Result: $x$
Compute $r_0 := b - Ax_0, p_0 := r_0$;
for $i = 0, 1, \cdots, dim(A)$ do
    $\alpha_i := (r_i, r_i)/(Ap_i, p_i)$;
    $x_{i+1} := x_i + \alpha_i p_i$;
    $r_{i+1} := r_i - \alpha_i Ap_i$;
    if $r_{i+1} < tolerance$ then
        break
    end
    $\beta_i := (r_{i+1}, r_{i+1})/(r_i, r_i)$;
    $p_{i+1} := r_{i+1} + \beta_i p_i$
end
```

Task: Add a parallel construct and appropriate clauses to accelerate iterations of CG method.

References I

- Timothy G Mattson, Yun Helen He, and Alice E Koniges, *The openmp common core: Making openmp simple again*, MIT Press, 2019.
- Peter Pacheco, *An introduction to parallel programming*, Elsevier, 2011.

Thank you!

Please help us improve!

https://anu.au1.qualtrics.com/jfe/form/SV_0BRRQvSz5oiwOqO