Shared-memory Programming with OpenMP (MK book, Ch.5)

OpenMP with C/C++ -- higher level than PThreads

```
#include <omp.h>
$>g++ -fopenmp p1.cpp
```

<u>error checking way</u>:

```
#ifdef _OPENMP        //if OpenMP is available, include <omp.h>,
#include <omp.h>      //otherwise, do not;
#endif
```

in the code, use directives (parallel, parallel for, critical, etc.):

```
#pragma omp . . . .
```

directive (within a line)

can be single statement or { block }

parallel block

- Each thread has its own PC and stack.

```
#pragma omp parallel   //system uses the total # of cores (logical cores)
```

vs.

```
#pragma omp parallel num_threads (thread_count)  //any #of cores
```

→ provides automatic joins (implicit barrier).

ex)

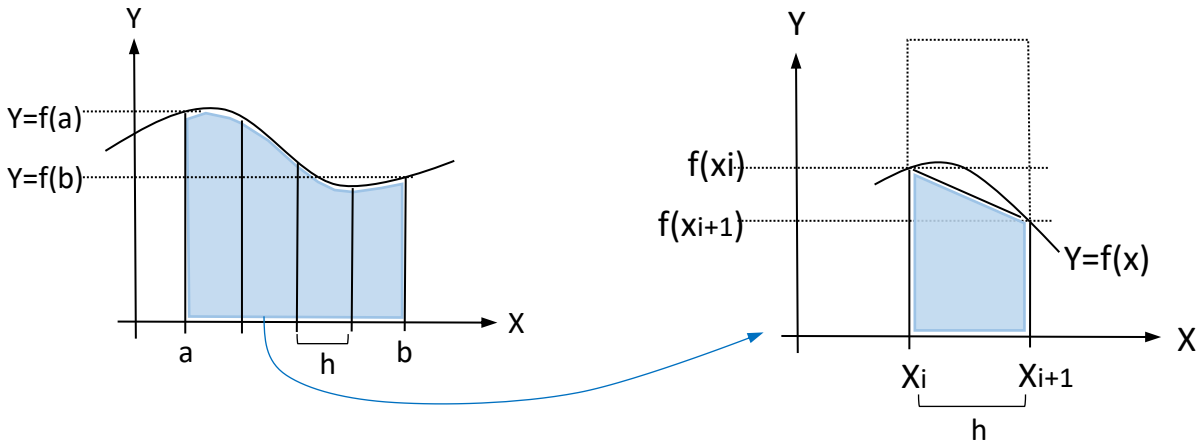
```
. . . .  //main thread
#pragma omp parallel num_threads (4)   //total 5 threads (main + 4 forks)
```

Parallel block

master slaves

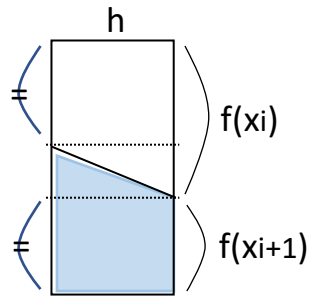
a team

<u>Trapezoid for estimating the area under a curve</u> – (integral $\int_{x=0}^{x=n}$)

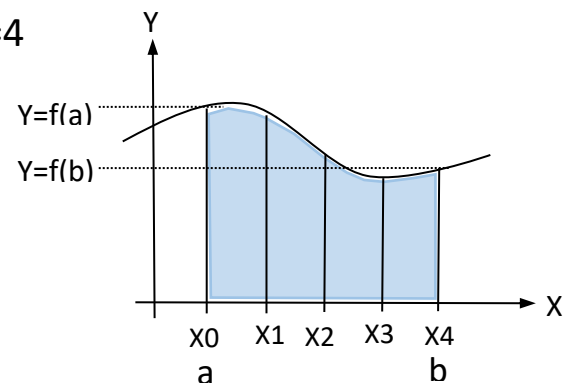


h = $\frac{b-a}{n}$, n = # of trapezoids

area = $\frac{(f(x_i)+f(x_{i+1}))*h}{2}$



ex) n=4



total area = $\frac{h*(f(x0)+f(x1)}{2} + \frac{h*(f(x1)+f(x2)}{2} + \frac{h*(f(x2)+f(x3)}{2} + \frac{h*(f(x3)+f(x4)}{2}$

= $\frac{h}{2}$ [(f(x0)+f(x1)) + (f(x1)+f(x2)) +(f(x2)+f(x3)) +(f(x3)+f(x4))]

= $\frac{h}{2}$ [<u>f(x0)</u> + 2*f(x1) + 2*f(x2) + 2*f(x3) + <u>f(x4)</u>]

→ initial: approx = (f(x0) + f(x4)) / 2.0;

then, for (i=1 ~ n-1) { approx += f(xi);} *//i=1..3*

finally, approx (tot_area) = h * approx;

OpenMP implementation of the trapezoidal rule

reference parameter way
return function way
reduction way
parallel_for loop way

1. reference parameter way

```
in main(..),
. . . .
#pragma omp parallel num_threads (thread_count)
Trap (a, b, n. global_result);  //global_result is ref. para.
. . . .
void Trap (double a, double b, int n, double& global_result)
{ . . . .
  compute local_result;
  #pragma omp critical
  global_result += local_result;
} //global_result computation in the thread function
```

reserved word

scope of variables:

variables declared before parallel directive: shared scope among threads in the team;

vs.

variables declared in the parallel block (locals in thread func): private scope

scope can be changed with other directives;

2. <u>return function way</u>

```
in main(..),
. . . .
global_result = 0.0;
#pragma omp parallel num_threads (thread_count)
{  double local_result = 0.0;  //private to each thread
   local_result += local_Trap (a, b, n);
   #pragma omp critical
   global_result += local_result;
}
```

//global_result computation in main( )

3. <u>reduction way (with return function)</u>

```
in main(..),
. . . .
global_result = 0.0;
#pragma omp parallel num_threads (thread_count) \
    reduction (+: global_result)
global_result += local_Trap (a, b, n);
```

//line continues

//reduction is done on this "+" operation and "global_result" (must be shared var)

4. <u>parallel-for loop way</u>

without using thread function,

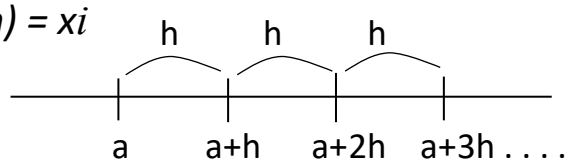
in main(..),

. . . .

h = (b – a) / n;

approx = (f(a) + f(b)) / 2;

```
#pragma omp parallel for num_threads (thread_count) \
    reduction (+: approx)   //parallel for must use for loop
for (i=1; i <= n-1; i++)        //loop var i is private to each thread
   approx += f(a + i*h);      //(a + i*h) = xi
approx. = h * approx;
```



- <u>parallel for</u> must use <u>for loop</u>

  #pragma omp parallel for . . . .

  for (i=0;. . .) . . . .

  iterations are divided into threads, usually block division way, e.g.,

  i=0~10 → Th_0

  i=11~20 → Th_1

  i=21~30 → Th_2

  . . . .

Parallel for directive

Only parallelizes for-loop, but no while/do-while loops;

in fact, while/do-while loops can be converted to for-loop way.

- Cannot use "parallel for" for data dependent loops

  ex) compute first N Fibonacci numbers:

  1, 1, 2, 3, 5, 8, 13, ...

  each loop iteration depends on previous iteration(s);

  trial:

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel for num_threads (thread_count)
for (i=2; i < n; i++)
    fibo[i] = fibo[i-1] + fibo[i-2];
```

  if we use 2 threads (assume N=10),

  output: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 --- if Th_1 starts after Th_0 finishes

  or, 1, 1, 2, 3, 5, 8, 0, 0, 0, 0 --- if Th_1 starts before Th_0 finishes

  → not deterministic output

  OpenMP compiler doesn't check for dependency among iterations.

  → programmer's ersponsibility

ex)

```
for (i=0; i < n; i++)
{  x[i] = a + i*h;
   y[i] = exp(x[i]);
}
//∄ loop dependency
```

→

```
#pragma omp parallel for . . . .
for (i=0; i < n; i++)
{  x[i] = a + i*h;      //executed in the
   y[i] = exp(x[i]);    //same thread in
                        //the same iteration
}                       //→ no problem
```

Loop carried dependency

ex) estimating π (other examples: prefix sum, FIbo#, etc.)

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$$

$$= 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

serial algorithm:

```
double factor = 1.0;
double sum = 0.0;
for (k=0; k < n; k++)
    sum += factor / (2*k + 1);
    factor = - factor;
pi_approx = 4.0 * sum;
```

factor is updated in iteration_i
(ex, assigned to Thread_i);

This affects factor of iteration_i+1
(ex, assigned to Thread_i+1)

→ loop-carried dependency problem
→ so, cannot use parallel-for

trial:

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for num_threads (thread_count) \
        reduction (+: sum)
for (k=0; k <n; k++)
{  sum += factor / (2*k + 1);
   factor = - factor;
}
pi_approx = 4.0 * sum;
```

→ *incorrect result since ∃ loop carried dependency*

correct version:

```
double factor = 1.0;  //factor is shared_var
double sum = 0.0;
#pragma omp parallel for num_threads (thread_count) \
      reduction (+: sum) private (factor)
for (k=0; k <n; k++)
{ if (k % 2 == 0)
      factor = 1.0;
  else
      factor = -1.0;
  sum += factor / (2*k + 1);
}
pi_approx = 4.0 * sum;
```

//Each thread has a private copy of factor (not shared)

- scope of private variables

ex)

```
int x = 5;  //shared (since declared out of parallel block)
#pragma omp parallel num_threads (thread_count) private (x)
{  int my_rank = omp_get_thread_num( );
   cout<<x;  //x? unspecified
   x = 2*my_rank + 2;
   cout<<x;
}
cout<<x;   //x=5
```

//Now, x is private to each thread; Value of x is unspecified at the beginning of parallel or parallel-for block.

Variables declared inside of a parallel block are always private, i.e., each thread has a stack.

Scope of variables – more

In a parallel block, programmer can specify the scope of variables.

#pragma omp parallel (or, parallel for) . . . . default (none) . . . .

//we need to provide scope of variables declared out of the parallel block

Basically,

```
. . . .    //variables declared out of block are shared (to all threads);
parallel . . . .
{  . . . . //variables declared in the block are private (to each thread);
}
```

With parallel . . . . default (none),

→ We need to provide the scope of variables, which are declared out of the block (and used in the block).

ex) . . . .

```
double sum = 0.0;
#pragma omp parallel for num_threads (thread_count) \
      default (none) reduction (+: sum) \
      private (k, factor) shared (n)
for (k=0; k < n; k++)
{ if (k % 2 == 0)  factor = 1.0;
  else  factor = -1.0;
  sum += factor / (2*k + 1);
}
```

//sum is both private and shared, since it is a reduction var. So, don't have to specify scope

shared var – not updated in the block;

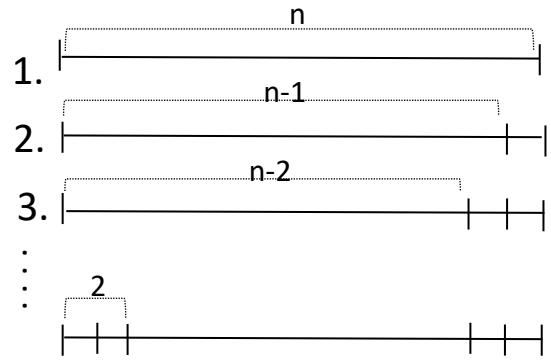
private var – updated in the block;

Loops in OpenMP

ex) bubble sort

algorithm:

```
for (len=n; len >= 2; len--)
    for (i=0; i < len-1; i++)
        if (a[i] > a[i+1])
            swap (a[i], a[i+1]);
```



Q1: Is there loop-carried dependency in the outer loop?

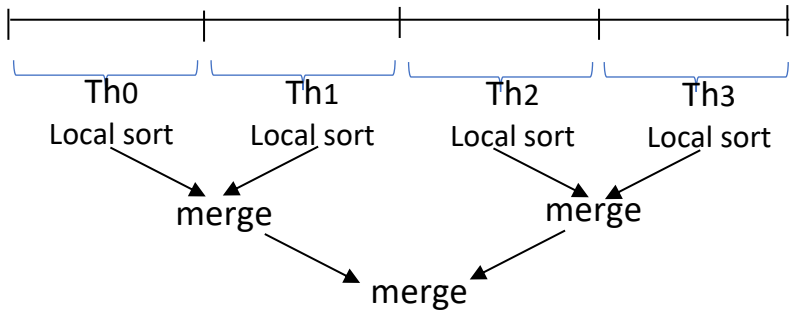
→ Yes, since the current list depends on the previous iterations.

Q2: Is there loop-carried dependence in the inner loop?

→ Yes, since the current pair depends on previous iterations.

So, we cannot apply parallel-for to bubble sort.

<u>Any idea of parallelization?</u>



- parallel-for vs. for

```
#pragma omp parallel num_threads (thread_count) . . . .
{ . . . .
  #pragma omp parallel for
  for (i=0; i<n; i++)
  . . . .
}
```

//with forking a new team of threads, perform parallel-for

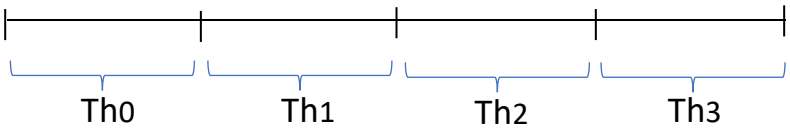
vs.

```
#pragma omp for
for (i=0; i<n; i++)
```

//with current team of threads, perform parallel-for

<u>Scheduling loops</u> – for <u>parallel-for</u> (or, <u>for)</u>

Parallel-for: most systems assign block-divide way;



→ problem: possibly unbalanced load

```
ex)  sum = 0.0;
        for (i=0; i<n; i++)
            sum += f(i);
```

//if f(i) time depends on i linearly, i.e., bigger i yields bigger f(i) time, Th_0 takes the least time while Th_{p-1} takes the longest time; → load unbalanced

→ better to use <u>cyclic assignment</u> (scheduling):

Th_0: 0, 0+P, 0+2P, 0+3P, ...

Th_1: 1, 1+P, 1+2P, 1+3P, ...

Th_2: 2, 2+P, 2+2P, 2+3P, ...

. . . .

Th_{p-1}: P-1, (P-1)+P, (P-1)+2P, (P-1)+3P, ...

→ *still problem!!*

<u>Schedule clause with parallel-for (or, for)</u>

ex) default block scheduling

```
sum = 0.0;
#pragma omp parallel for num_threads (thread-count) reduction (+: sum)
for (i=0; i<=n; i++)
    sum += f(i);
```

ex) cyclic scheduling

```
sum = 0.0;
#pragma omp parallel for num_threads (thread-count) \
      reduction (+: sum)  schedule (static, 1)
for (i=0; i<=n; i++)
   sum += f(i);
```

schedule type *chunk size*

<u>Scheduling types</u>:

1. **static** – iterations are assigned to threads before loop executes;
2. **dynamic** – iterations are assigned to threads at run time (during loop exec.); possibly, some threads more, some threads less;
3. **auto** – compiler and/or run-time system determines and assigns;
4. **run time** – run-time system determines scheduling;

ex) n=12, P=3

schedule (static, 1) → Th_0: 0, 3, 6, 9
Th_1: 1, 4, 7, 10
Th_2: 2, 5, 8, 11

schedule (static, 2) → Th_0: 0, 1, 6, 7
Th_1: 2, 3, 8, 9
Th_2: 4, 5, 10, 11

<u>schedule (static, n/P)</u> → Th_0: 0, 1, 2, 3
//same as default Th_1: 4, 5, 6, 7
//block scheduling Th_2: 8, 9, 10, 11

dynamic scheduling

Each thread requests the next chunk (to the run-time system) when current chunk is done; this repeats until all done.

ex) schedule (dynamic, 1) *//each time, chunk size 1*

schedule (dynamic, 2) *//each time, chunk size 2*

. . .

<u>schedule (guided)</u> //a kind of dynamic scheduling

Each chunk size is reduced by n/P and assigned to available threads at run time.

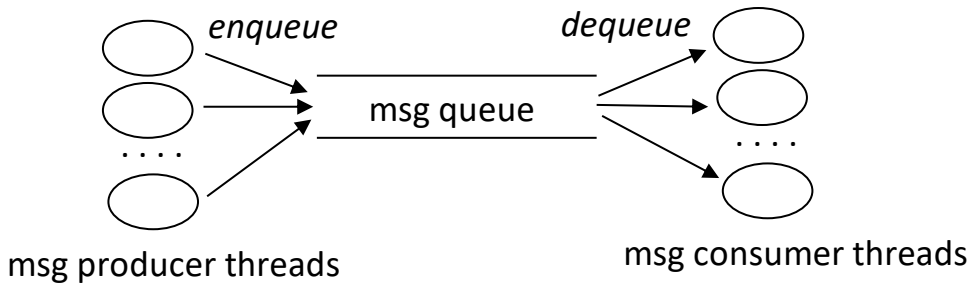
ex) n=1000, P=2

schedule (guided) → Th_0: 500 (1~500)

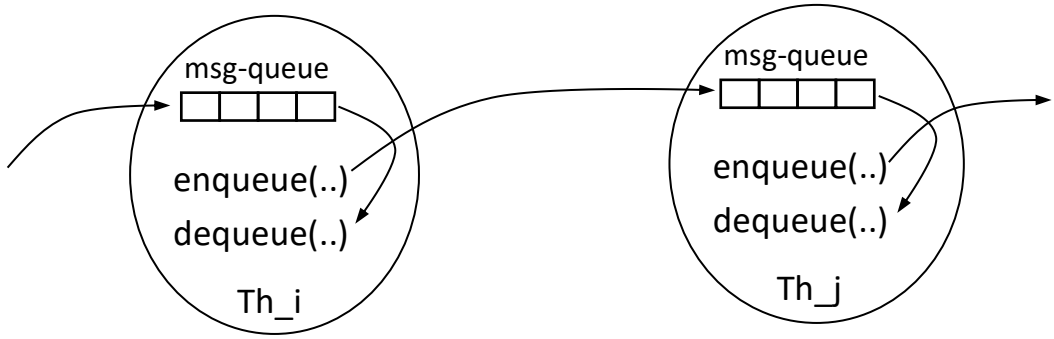
Th_1: 250 (501~750)

Th_1: 125 (751~875)

Producers-consumers with OpenMP



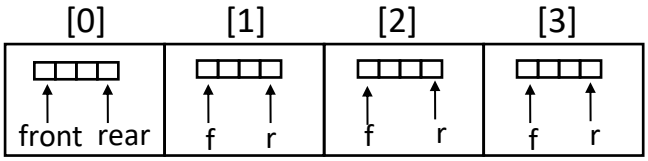
- Message passing mechanism is implemented in shared-mem programming.
- In each thread, enqueue/dequeue operations may repeat; and a thread may send (enqueue) message to any other thread.



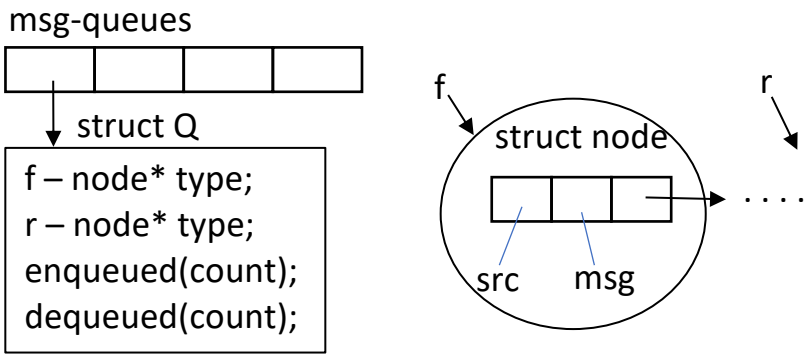
Implementation: 2D dynamic array of msg-queues (one for each thread);
each msg-queue is implemented with linked-list of nodes;

ex) P=4

conceptual view of <u>msg-queues</u>:



implementation:



in master thread (main),

```
declare (new) msg_queues (struc Q** type);  //new 1st dimension
. . . .
#pragma omp parallel . . . .
{ . . . .
  msg_queues [my_rank] = new struct Q;  //in each thread_i, new 2nd dim.
  #pragma omp barrier;  //don't let any threads send msgs until all
                        //msg queues are constructed by all threads
  #pragma omp critical
  enqueue (msg_queues [dest], my_rank, msg);  //send msg –mutex needed
  dequeuer (msg_queues [my_rank], src, msg);  //receive msg
  . . . .
  free/delete msg_queues [my_rank];  //free 2nd dimension in each thread
} //pragma omp parallel
free/delete msg_queues;  //free 1st dimension
```

possibly, repeats

enqueue(..) and dequeuer(..) use linked-list implementation of queue;

enqueue(..) operation is C.S. → needs mutex mechanism;

- #pragma omp <u>barrier</u> – provides barrier synchronization;
- #pragma omp <u>atomic</u> – similar to "critical", but used for only single assignment statement in the form of load-modify-store, e.g., x += 5, which needs load x; modify x; then, store x;

  //more efficient
  //than critical

  ex) x += <exp>; -=, *=, /=

  x++, x--, ++x, --x

dead-lock case with nested critical

ex)
```
#pragma omp critical
y = f(x);
. . . .
double f (double x)
{  #pragma omp critical
   z = g(x);  //z is global (shared)
   . . . .
}
```

//same name

At this moment of time, "critical" is locked by myself (in this thread) and cannot unlock it. → **dead-locked**

→ Sol: using named critical

//any name

ex)
```
#pragma omp critical (one)
y = f(x);
. . . .
double f (double x)
{  #pragma omp critical (two)
   z = g(x);
   . . . .
}
```

//different names